UNIVERSITY
OF LONDON

Creative computing I: image, sound and motion

Volume 1

M. Casey with

T. Taylor, A. Smaill and C. Brownrigg

CO1112

2014

Undergraduate study in
Computing and related programmes

Goldsmiths
UNIVERSITY OF LONDON

This subject guide was prepared for the University of London by:
Michael Casey, Department of Music, Dartmouth College, USA
Tim Taylor, Department of Computing, Goldsmiths, University of London, UK
Alan Smaill, School of Informatics, University of Edinburgh, UK
Chris Brownrigg, freelance artist and writer, UK

Additional help with production was provided by:
Sarah Rauchas, Department of Computing, Goldsmiths, University of London

This is one of a series of subject guides published by the University. We regret that due to pressure of work the authors are unable to enter into any correspondence relating to, or arising from, the guide. If you have any comments on this subject guide, favourable or unfavourable, please use the form at the back of this guide.

First published 2007
This edition published 2014

In this and other publications you may see references to the 'University of London International Programmes', which was the name of the University's flexible and distance learning arm until 2018. It is now known simply as the 'University of London', which better reflects the academic award our students are working towards. The change in name will be incorporated into our materials as they are revised.

University of London
Publications Office
32 Russell Square
London WC1B 5DN
United Kingdom

london.ac.uk

Published by: University of London

Contents

Preface

This course is about expressing creative ideas through computing. At the end of the course you will understand some foundational creative processes in the form of computer programs that produce audio-visual content to very high standards. The course provides the foundations of programming for creativity coupled with principles of form, structure, transformation and generative processes for image, sound and video. These methods are the conceptual tools that are widely applied in the creative industries. They are used by designers, special effects technicians, animators, games developers and video jockeys alike.

At the end of this course you will have the facility to program your creative ideas. As such, you will understand more deeply the concepts behind creative and commercial software that is in wide use.

The subject guide for **Creative Computing I** is divided into two volumes. The first volume will introduce you to the basic materials that you will need to start your own creative portfolio. The second volume will expand on these foundations so that you can develop your own unique tools and methods via programming. It is therefore very important that you become familiar with the contents of this guide.

By the end of this course, you should be able to implement creative concepts that are not easily realised with commercial software packages and, therefore, you will be enabled to demonstrate a high degree of originality in your own creative work.

The assessment of this course will be formed of two pieces of course work and an exam that you will sit at the end of the first year of the programme. The exam will be an unseen written exam. The exam questions will be about the background, techniques and examples in this subject guide, the second volume of this subject guide, and the essential reading (see below) but not the additional reading. While not required, you should read the items on the additional reading list where possible to increase your understanding of the general subject area.

This subject guide is not a course text. It sets out the logical sequence in which to study the topics in the course. Where coverage in the essential texts is weak, it provides some additional background material. Reading the essential and additional texts is important as you are expected to see an area of study from an holistic point of view, and not just as a set of limited topics.

Some general notes about this guide

Website links

Unless otherwise stated, all websites in this subject guide were accessed in March 2014. We cannot guarantee, however, that they will stay current and you may need to perform an internet search to find the relevant pages.

Colour

A colour version of this subject guide is available as a PDF document in the *CO1112 Creative Computing I* section of the VLE.[1] You may find the colour version easier to follow.

[1] http://computing.elearning.london.ac.uk

Essential reading

Reas, C. and B. Fry *Processing: A Programming Handbook for Visual Designers and Artists* (MIT Press, 2007) [ISBN 0262182629].

Reas, C. and B. Fry *www.processing.org/reference*, on-line *Processing* reference manual.

`http://www.bauhaus.de` (site available in German and English)

`http://www-history.mcs.st-andrews.ac.uk/index.html`

Additional reading

Bayer, H., W. Gropius and I. Gropius (eds) *Bauhaus 1919-1928* (Museum of Modern Art, 1976) [ISBN 0810960133].

Behrens, R. R. *Art, Design and Gestalt Theory*, Leonardo, Vol. 31, No. 4, pp. 299–303, 1998.

Bell, E.T. *Men of Mathematics* (Simon & Schuster, 1986) [ISBN 0671628186].

Berger, J. *Ways of Seeing* (Penguin, reprint edition, 1990) [ISBN 0140135154].

Borchardt-Hume, A. (ed) *Albers and Moholy-Nagy: from the Bauhaus to the New World* (Tate Publishing, 2006) [ISBN 1854376918 (hbk), 1854376381 (pbk)].

Eskilson, S. J. *Graphic Design: A New History* (Yale University Press, 2007) [ISBN 0300120117]. Chapter 6–The Bauhaus and the New Typography.

Fauvel, J., R. Flood and R. Wilson (eds) *Music and Mathematics* (Oxford University Press, 2006) [ISBN 0199298939].

Fauvel, J. and J. Gray (eds) *The History of Mathematics: A Reader*, (MacMillan Education, 1987) [ISBN 0333427912].

Hughes, J. F., A. van Dam, M. McGuire, D. Sklar, J. D. Foley, S. K. Feiner, and K. Akeley *Computer Graphics: Principles and Practice* (Addison-Wesley, 3rd edition, 2013) [ISBN 0321399528].

Hofstadter, D. R. *Gödel, Escher, Bach: An Eternal Golden Braid*. (Basic Books, 20th anniversary edition, 1999) [ISBN 0465026567].

Itten, J. *Design and Form, The Basic Course at the Bauhaus* (Thames and Hudson, 1975) [ISBN 0471289302].

Kandinsky, W. *Point and Line to Plane* (Dover, 1980) [ISBN 0486238083].

Maeda, J. *Creative Code: Aesthetics and Computation* (Thames and Hudson, 2004) [ISBN 0500285176].

Moggridge, B. *Designing Interactions* (MIT Press, 2006) [ISBN 0262134748].

Naylor, G. *The Bauhaus Reassessed* (The Herbert Press, 1985) [ISBN 0906969298, 0906969301(pbk)].

Packer, R. and K. Jordan (eds) *Multimedia: From Wagner to Virtual Reality* (W. W. Norton and Company, expanded edition, 2003) [ISBN 0393323757].

Poling, C. V. *Kandinsky's Teaching at the Bauhaus; Colour Theory and Analytical Drawing* (Rizzoli, illustrated edition, 1986) [ISBN 0847807800].

Rand, P. *A Designer's Art* (Yale University Press, new edition, 2001) [ISBN 0300082827].

Russell, B. *A Critical Exposition of the Philosophy of Leibniz* (London: George Allen and Unwin, 1900; new edition Cambridge University Press, 2013) [ISBN 1107680166].

Shiffman, D. *The Nature of Code: Simulating Natural Systems with Processing* (The Nature of Code, 2012; free online version available at natureofcode.com/book/) [ISBN 0985930802].

Steiner, G. *Grammars of Creation* (Faber & Faber, 2001) [ISBN 0571206816].

Stillwell, J. *Mathematics and its History* (Springer-Verlag, third edition, softcover reprint, 2012) [ISBN 1461426324].

van Campen, C, *Early Abstract Art and Experimental Gestalt Psychology*, Leonardo, Vol. 30, No. 2, pp. 133–136, 1997.

Wong, W. *Principles of Form and Design* (Wiley, 1993) [ISBN 0471285528].

Xenakis, I. *Formalized Music: Thought and Mathematics in Composition* (Pendragon Press, 1992) [ISBN 1576470792].

Zakia, R. D. *Perception and Imaging: Photography—A Way of Seeing* (Focal Press, 4th edition, 2013) [ISBN 0240824539].

Chapter 1

History of Mathematics and Computing in Creativity

Essential reading

`http://www-history.mcs.st-andrews.ac.uk/index.html`. Use the History Topics Index and the Biographies Index on this website as a starting point for digging deeper into the subjects introduced in this chapter.

Additional reading

Bell, E.T. *Men of Mathematics* (Simon & Schuster, 1986) [ISBN 0671628186].

Fauvel, J., R. Flood and R. Wilson (eds) *Music and Mathematics* (Oxford University Press, 2006) [ISBN 0199298939].

Fauvel, J. and J. Gray (eds) *The History of Mathematics: A Reader*, (MacMillan Education, 1987) [ISBN 0333427912].

Hofstadter, D. R. *Gödel, Escher, Bach: An Eternal Golden Braid*. (Basic Books, 20th anniversary edition, 1999) [ISBN 0465026567].

Russell, B. *A Critical Exposition of the Philosophy of Leibniz* (London: George Allen and Unwin, 1900; new edition Cambridge University Press, 2013) [ISBN 1107680166].

Steiner, G. *Grammars of Creation* (Faber & Faber, 2001) [ISBN 0571206816].

Stillwell, J. *Mathematics and its History* (Springer-Verlag, third edition, softcover reprint, 2012) [ISBN 1461426324].

Xenakis, I. *Formalized Music: Thought and Mathematics in Composition* (Pendragon Press, 1992) [ISBN 1576470792].

1.1 Introduction

We begin our study of creative computing with a look at some ways in which mathematics and computing have played roles throughout history in relation to creativity. At different times, these sciences and technologies have opened up creative possibilities; they are also arenas themselves where we see many creative moments throughout history.

We see the latter view from George Steiner, who writes in "Grammars of Creation":

> *It is in mathematics and the sciences that the concepts of creation and of invention, of intuition and of discovery, exhibit the most immediate, visible force.*
>
> *(Steiner (2001), p.145)*

We will look at some moments in history, rather than attempt to trace developments through the centuries.

1.2 Earliest Mathematics

We know that mathematical ideas can be seen in written form as far back as the time of the Babylonians (2000–1500 BC)[1]. We understand these texts as talking of algebraic problems. One case that appears here involves finding three integers a, b, c such that $a^2 + b^2 = c^2$. Why should this be interesting? One aspect of this known to the Babylonians is that if a rope is used to form a triangle whose sides are such lengths a, b, c, then the triangle has a right angle (an angle of 90 degrees). We are used to being surrounded by objects with right angles, that can fit together easily and help in construction of buildings. But how is a right angle to be formed, without one to start with?

The most well-known case of these numbers is

$$3^2 + 4^2 = 5^2.$$

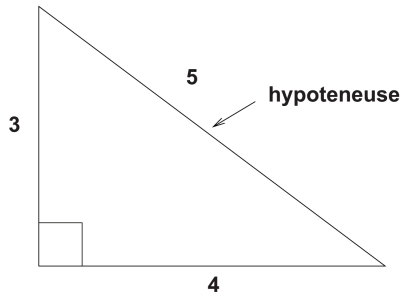


Figure 1.1: The 3,4,5 triangle

A natural question to ask is whether there is a general pattern to such combinations of a, b, c, and if we can be sure we have all such combinations. You can check that if $a^2 + b^2 = c^2$, then if we multiply each number by a fixed other number, the resulting numbers have the same property (e.g. $6^2 + 8^2 = 10^2$). From the Babylonian texts, they found a more interesting way of coming up with such numbers: if p, q are any two positive numbers, then let

$$a = p^2 - q^2, \quad b = 2pq, \quad c = p^2 + q^2.$$

Then $a^2 + b^2 = c^2$ (this can be checked easily); it turns out that this gives all possible numbers without a common factor, although that is a lot harder to show.

There are some characteristic features of mathematics here: there is an initial discovery with useful practical consequences, which prompts more general questions. Finding answers involves mathematical invention – it is a lot harder to dream up the equations above than to check that they give us a good answer. In fact, the work of Gödel and others shows—expressed informally—that we cannot find all of mathematics just by deduction from what is already known, so a creative leap is necessary for finding some new mathematical results.

[1] `http://www-history.mcs.st-andrews.ac.uk/Indexes/Babylonians.html`

1.3 Ancient Greece

Ancient Greece was important for the development of Western mathematical thought; we owe the systematic development of geometry and the introduction of the deductive method to the ancient Greeks (for deduction, see section 1.6), as well as the discovery of irrational numbers.

Let's look at what we call Pythagoras' Theorem [2]. Given any right-angled triangle (the length of each of the sides does not have to be an integer), if we draw a square of each side of the triangle, the sum of the areas of the two smaller squares is the area of the largest square.

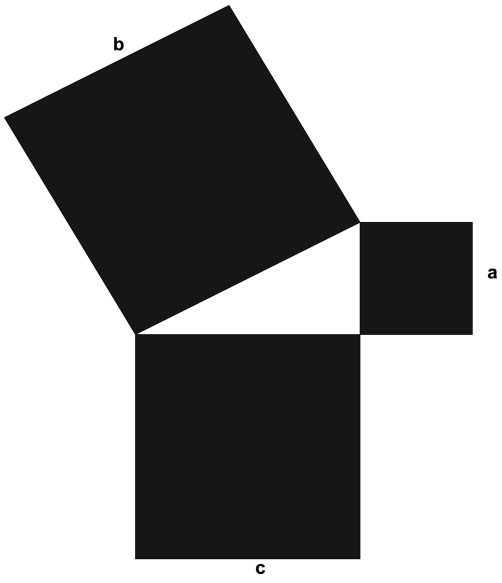


Figure 1.2: Pythagoras's theorem

There are many different ways of showing this; you might try to think how this could be proved in general. There are some animated versions of the proof[3], that work by moving parts of the diagram around so that the different areas (and some copies of them) can be put together in different ways. Remember that the claim is that this relationship holds for all possible right-angled triangles. Coming up with a way of showing that this is the case involves inventing new ways of thinking about area and geometrical shape.

1.4 Arab Mathematics and Computation

In the period 800–1400, there was relatively little new mathematics being developed in the West. However, in the Arab/Islamic part of the world, new ideas were developed that affect the way we think about mathematics and represent it to this

[2] http://www-history.mcs.st-andrews.ac.uk/Mathematicians/Pythagoras.html
[3] http://www.mathsisfun.com/pythagoras.html

day [4]. Not all the mathematicians involved were in fact Muslim (Jews and Christians were involved), and it is worth remembering that North Africa and parts of Spain were part of the Arab world at the time.

It is a tribute to this period that mathematical terms from Arabic have found their way into the English language. Among these are:

zero The Roman numeral system does not have a number "zero"; nowadays, the symbol "0" plays two roles, as a number on its own, and as a part of a numeral system (in base 10, for example). Both these uses came to the West through the influence of Arab mathematics [5].

algebra The beginnings of modern algebra appear with the work of al-Khwarizmi[6]; he treated rational numbers, irrational numbers and geometrical magnitudes as algebraic entities. The word "algebra" itself comes from the title of his book from the 9th century, "Hisab al-jabr w'al-muqabala".

algorithm The term "algorithm" is in fact based on the name "al-Khwarizmi". The word has passed through Latin, originally referring to the Arabic numbering system, but also associated with different ways of working out solutions to arithmetic problems [7].

Arabic numerals The symbols 0,1,2,3,4,5,6,7,8,9, to which we are accustomed, come from Indian sources to the West via Arabic mathematics [8].

It is much easier to work with this numeral system than with, for example, Roman numerals; try to work out an algorithm for the multiplication of two numbers directly in the Roman system and the difference becomes apparent.

1.5 The Renaissance: Geometry and Perspective

During the Renaissance in the West, there was a flowering of thought both in sciences and in the arts, with interaction between them (for example, Leonardo da Vinci's anatomical drawings). It is worth looking at Leonardo's *Vitruvian man*[9], and thinking about how ideas on art, design, architecture and creativity interact, including the use of two of the three basic shapes from Bauhaus design theory (discussed in the next chapter).

We can look at one connection around this time, that between the mathematical analysis of the geometry of vision, and the introduction of perspective into Western art.

The Renaissance brought with it a questioning of received ideas, and an interest in classical thought as a model that could be surpassed. Brunelleschi[10] shows these traits: he trained as a goldsmith and sculptor, while taking an interest in geometry. In 1415, he worked out the principles of perspective: parallel lines in reality should be depicted as converging to a "vanishing point", and objects' sizes should vary inversely with their distance from the plane of the painting.

[4] http://www-history.mcs.st-andrews.ac.uk/Indexes/Arabs.html
[5] http://www-history.mcs.st-andrews.ac.uk/HistTopics/Zero.html
[6] http://www-history.mcs.st-andrews.ac.uk/Biographies/Al-Khwarizmi.html
[7] http://www.peak.org/~jeremy/calculators/alKwarizmi.html
[8] http://www-history.mcs.st-andrews.ac.uk/HistTopics/Arabic_numerals.html
[9] http://en.wikipedia.org/wiki/Vitruvian_Man
[10] http://www-history.mcs.st-andrews.ac.uk/Biographies/Brunelleschi.html

At this time, earlier paintings depicted scenes "as they are", such as in the work of Simone Martini[11], rather than "as they are seen".

The interest in the geometry of three-dimensional space can be seen in the work of the artist Uccello[12]. The artist Dürer is credited with the invention of "perspective machines", that helped artists master the new techniques, as seen in his own etching, in Figure 1.3 below, of 1525. We might think that these devices would lead to "mechanical" drawings, but in fact Dürer's own work is anything but mechanical; the mathematics of space had opened up new artistic possibilities.

Figure 1.3: Dürer: Draughtsman Making a Perspective Drawing of a Woman. From: Bronowski, J, *The ascent of Man*, London : British Broadcasting Corporation, 1973, p.181.

1.6 Inventing Computational Thinking

Computational devices and algorithmic ideas went hand in hand with mathematics from the earliest times; think of the abacus[13], found throughout the world at different times. Astronomy gave ways of predicting the seasons, and methods and tools are associated with it, such as the astrolabe[14].

Leibniz[15] was a German philosopher and mathematician, who lived from 1646 to 1716. As a philosopher, he tried to provide accounts of knowledge, of truth, and of the relation of perception to the external world.

He thought that mathematics gave access to especially clear and uncontroversial conclusions via *calculations*. He built a calculating machine, which he showed to the Royal Society in London in 1673 – so his interests in this topic were not just theoretical.

Leibniz thought that reasoning in general could be dealt with via calculation, if we could express statements in what he called the *Characteristica Universalis* (universal mathematics). Then, given rules to calculate correct conclusions, reasoning could be done in this new language. Once we wrote down our knowledge in this special language, reasoning could be replaced by computation.

He wrote:

> *Telescopes and microscopes have not been so useful to the eye as this instrument would be in adding to the capacity of thought. . . . If we had it, we should be able to reason in metaphysics and morals in much the same way as in geometry and analysis.*
>
> *(Leibniz, quoted in Russell (1900))*

[11] `http://www.ibiblio.org/wm/paint/auth/martini`
[12] `http://abstract-art.com/abstract_illusionism/ai_03_put_into_persp.html`
[13] `http://en.wikipedia.org/wiki/Abacus`
[14] `http://en.wikipedia.org/wiki/Astrolabe`
[15] `http://www-history.mcs.st-andrews.ac.uk/Mathematicians/Leibniz.html`

and also

> *If controversies were to arise, there would be no more need of disputation between two philosophers than between two accountants. For it would suffice to take their pencils in their hand, to sit down to their slates, and say to each other (with a friend as witness, if they liked): Let us calculate.*
>
> *(Leibniz, quoted in Russell (1900), p.169)*

This idea of having machines use logic for themselves eventually became an impetus for work in Artificial Intelligence.

Since the 19th century, other machines have been built to carry out calculations: mechanical devices in the 19th century, up to programmable electronic machines starting around 1945[16]. Back in 1834, Charles Babbage in England conceived a mechanical computer programmable via punched cards (already in use in Jacquard looms as "programs" to control pattern elements in weaving, via a "card reader"). The computer, which was first described in 1837 and more fully in 1843, was called the Analytical Engine[17]. The 1843 description was extensively annotated by Ada Lovelace[18], who worked closely with Babbage. In her notes, Lovelace outlined an algorithm intended to be processed by the Analytical Engine, and is therefore regarded by many as the first computer programmer[19]. The construction of the Analytical Engine was never completed, although a partial trial model was assembled in 1871[20].

It is striking how much of the mathematical theory of computation had been put in place before there were any computers in existence, as we understand the term today. Already in 1936, mathematical characterisations had been worked out, describing which functions (over the natural numbers) could be *computed*. It had been shown that there are some such functions that simply cannot be computed, regardless of which programming language is used and however much time and space is used in the computation. Alan Turing[21] was one of the pioneers here, along with Alonzo Church and Emil Post.

It had been a long-standing philosophical question whether machines can show intelligence, and Alan Turing was also instrumental in provoking work in Artificial Intelligence[22]. His description of *reasoning machines* breathed new life into Leibniz's outline proposal: Turing argued that computers could in principle be made to process sensed data following reasoning patterns as humans do, and would then be to all intents and purposes acting intelligently.

The present-day Loebner competition[23] in Artificial Intelligence is based directly on Turing's work.

[16] http://en.wikipedia.org/wiki/History_of_computing_hardware
[17] http://en.wikipedia.org/wiki/Analytical_Engine
[18] http://www-history.mcs.st-andrews.ac.uk/Mathematicians/Lovelace.html
[19] http://en.wikipedia.org/wiki/Ada_Lovelace
[20] http://www.sciencemuseum.org.uk/objects/computing_and_data_processing/1878-3.aspx
[21] http://www-history.mcs.st-andrews.ac.uk/Mathematicians/Turing.html
[22] http://en.wikipedia.org/wiki/Turing_test
[23] http://www.loebner.net/Prizef/loebner-prize.html

1.7 Mathematics and Music

There has been an interplay between music and mathematics throughout the development of both subjects (Fauvel, Flood and Wilson (2006) cover the history and current state of this interaction). Pythagoras, whom we met above, and his school, investigated how to produce different musical notes (pitches), e.g. by hitting glasses filled with water to different heights, or bells of different sizes (see the illustrations here[24]). It has been argued that this was the basis of the Ancient Greeks' interest in rational numbers: the most important intervals in music can be obtained by taking a vibrating string, e.g. guitar, and then not allowing the string to vibrate at points at $\frac{1}{2}, \frac{1}{3}, \frac{1}{4} \ldots$ of the length of the string, so producing musical harmonics[25].

It was from Pythagorean times that mathematics was considered to be composed of the four related studies: astronomy, geometry, arithmetic and music. This persisted into medieval times, where throughout European universities up to the 16th century the final topics studied at university formed the quadrivium, consisting of just these subjects.

As for visual art, there was a renewed interest in understanding music and the mathematics of sound during the Renaissance and later; an example is in the work of the French mathematician Marin Mersenne[26] who in 1627 published a book on the mathematics of harmony "L'harmonie universelle". (Mersenne is known today because his name is attached to the so-called "Mersenne primes", prime numbers of the form $2^p - 1$ where p is itself prime.)

Later, mathematical analyses were made of the sound waves corresponding to a single note at a fixed frequency; different instruments and different voices give different sound colours (timbres), and it turns out surprisingly that the sound waves involved can be described by a combination of sine waves at frequencies $n, 2n, 3n, \ldots$ where n is the frequency of the base note. This technique of harmonic analysis is due to the French mathematician Fourier[27], and is central to current digital sound production techniques.[28]

We can look at an important recent example of how mathematics enabled new sound worlds to be opened up, in the music of the Greek composer Iannis Xenakis (1922–2001). Many of his ideas are laid out in his book "Formalized Music" (1992). Xenakis was influenced by Ancient Greek culture, and collaborated with Le Corbusier as an architect in France. He saw architecture as being about articulating structures in space, while music involves deploying structures in time, in both cases involving calculation and reasoning. An example in his early piece "Metastasis", shown in Figure 1.4, involves a musical version of creating curves out of a number of straight lines.

Here each line corresponds to an instrument playing a note whose pitch varies uniformly in time (quickly if the line is steep); the effect is of a mass of changing sound which has edges that move in this "curved" way through time.

He introduced many other new musical possibilities – "Pithopratka" treats the

[24] http://www.philophony.com/sensprop/pythagor.html
[25] http://en.wikipedia.org/wiki/Harmonic
[26] http://www-history.mcs.st-andrews.ac.uk/Mathematicians/Mersenne.html
[27] http://www-history.mcs.st-andrews.ac.uk/Mathematicians/Fourier.html
[28] We will discuss these topics in much more detail in Chapter 5 of Volume 2 of this subject guide.

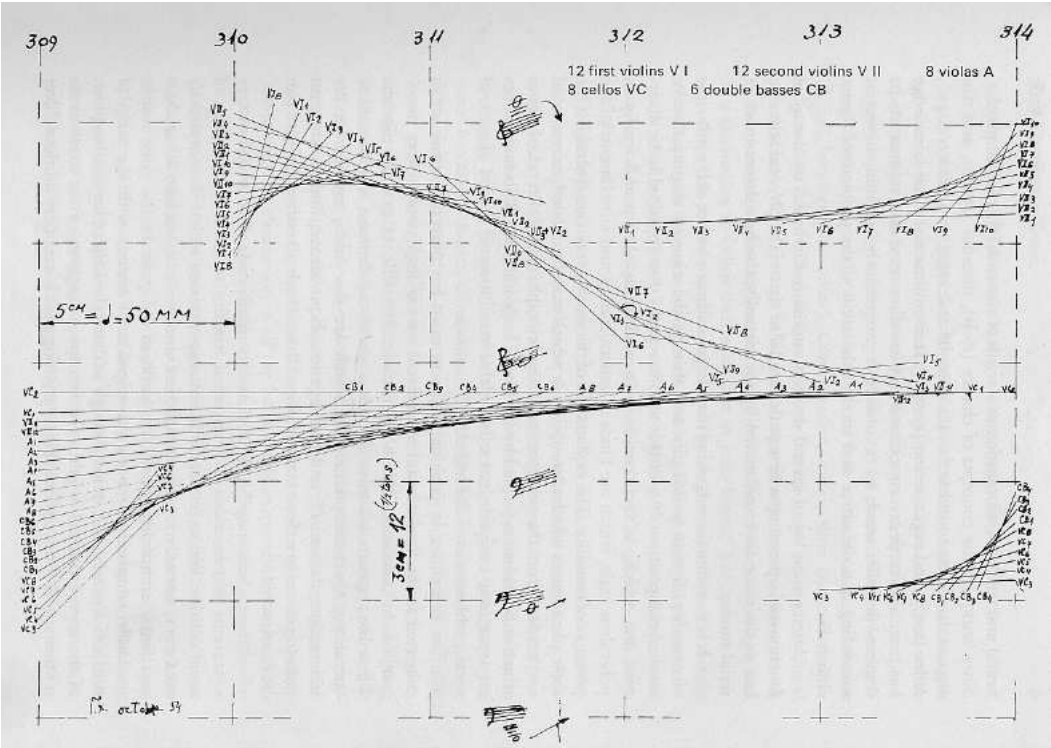


Figure 1.4: Xenakis: sketch for Metastasis. From: Iannis Xenakis, *Formalized Music: Thought and Mathematics in Composition*, Bloomington: Indiana University Press, 1971, pg. 3.

individual musicians like molecules in a gas, moving according to chance though the sound world, but still having predictable global properties, by making use of the statistical mathematics of gases.

Xenakis showed us one way in which mathematical and computational ideas can be liberating for the artist.

1.8 Some notes on additional reading

A good general overview of the history of mathematics is Steiner's "Grammars of Creation"(2001), though it supposes a good mathematical background. An account that focuses more on the personalities is Bell's "Men of Mathematics"(1986). An excellent on-line resource for the history of mathematics is the The MacTutor History of Mathematics archive[29].

We get a different feel for the invention of mathematics by looking at the way mathematicians describe their own work. There is a collection of snippets from mathematics through the ages in Fauvel and Gray's "The History of Mathematics: a Reader" (1987). Poincaré was a famous French mathematician who wrote about mathematical invention; there is an interesting comparison between Poincaré and

[29] `http://www-history.mcs.st-andrews.ac.uk/index.html`

his contemporary the artist Marcel Duchamp by Gerald Holton[30]. Hofstadter's "Gödel, Escher, Bach" (1999) remains the best popular account of the interplay between mathematics, science and the arts, all seen as creative domains in their different ways.

1.9 Summary and learning outcomes

This chapter set the scene for starting to learn about creative computing. It presented some elements from the history of mathematics in a creative context, and described how mathematicians and philosophers through the ages have made conceptual leaps in mathematics by applying creativity and imagination. It also introduced the work of some contemporary artists, and demonstrated how mathematics has been influential in their work.

You should now be able to:

- describe some of the major advances in the development of mathematics
- look at different numeral systems and identify the significant differences between them
- discuss how creativity has influenced the advance of mathematical theory
- identify the use of mathematical concepts in the work of some contemporary artists
- listen to music, look at architecture, examine a painting, and so forth, with a perspective that includes the computational aspects of such artworks.

1.10 Exercises

1. Here are some parts of modern mathematical notation. Find out who first introduced these symbols, and arrange them by the date they were first used.
   (a) The number e, the base of natural logarithms: $e = 2.72\ldots$.
   (b) The integral sign $\int$, as in $\int_0^1 x^2\,dx = 1/3$.
   (c) The square root sign $\sqrt{}$.
   (d) The square root of -1, $i = \sqrt{-1}$.
   (e) The symbol for infinity: ∞.
   (f) The notation for complex numbers in the form $z = x + iy$, where x and y are the real and imaginary parts of the number z.
2. Sketch out an algorithm for the addition of numbers given using Roman numerals[31]. The algorithm you sketch out should work for input numbers up to MM (which is 2000 in the decimal system).

   This way of representing numbers makes it harder to work out basic arithmetic operations than the modern notation. Describe why it is the case that arithmetic operations are harder to carry out in the Roman numeral system, compared with the decimal system.

[30] http://muse.jhu.edu/journals/leonardo/v034/34.2holton.html
[31] http://www.romannumerals.co.uk/roman-numerals/numerals-chart.html

3. One of the principles of perspective is that parallel straight lines in real life should be depicted as converging to a "vanishing point". This description relates to the point of view of the viewer. Describe in more detail what a vanishing point is, and how this relates to perspective.

   Now explain why parallel lines should be shown in this way, by considering a camera taking a photograph of a railway line, looking along the track, and thinking about the angles involved when light travels in a straight line from the track to the camera. In which cases would a vanishing point not be appropriate?

4. Alan Turing described what is now called the *Turing Test*; he suggested that the question "can machines think?" is too vague to be useful, and could usefully be replaced with the question "can machines pass the Turing Test?".

   Describe the Turing Test. Do you think this is a good test for whether or not a machine can think? (This is a hotly disputed subject, with no agreed answer; what is being asked for here is your own argument, with substantiation, for the viewpoint you are taking.)

5. The composer John Cage often included a mathematical component to his work, and the choreographer Merce Cunningham shared this interest. Look for descriptions of their work together, and discuss how their approaches interact with the topic of creative computing.

6. (a) It is interesting to compare Xenakis's sketch for Metastasis with architectural work he was doing at the same time with Le Corbusier for the Philips Pavilion in the World Fair in Brussels in 1958. Find some images of this building and Xenakis's sketches for that.

   (b) There are recordings of Xenakis's Metastasis on the web. If you are interested, find and listen to this seven-minute piece. The sketch shown in Figure 1.4 above appears about 50 seconds before the end of this piece, and is immediately followed by two bars of silence. In the sketch, time is notated left to right, with pitch marked vertically. So initially we hear a set of low-pitched sounds which get higher and closer together, and then after roughly a second, a cluster of higher-pitched sounds enter. The sketch covers about 7 seconds of music.

Chapter 2

The Bauhaus

Essential reading

`http://www.bauhaus.de` (site available in German and English)

Additional reading

Bayer, H., W. Gropius and I. Gropius (eds) *Bauhaus 1919-1928* (Museum of Modern Art, 1976) [ISBN 0810960133].

Borchardt-Hume, A. (ed) *Albers and Moholy-Nagy: from the Bauhaus to the New World* (Tate Publishing, 2006) [ISBN 1854376918 (hbk), 1854376381 (pbk)].

Eskilson, S. J. *Graphic Design: A New History* (Yale University Press, 2007) [ISBN 0300120117]. Chapter 6–The Bauhaus and the New Typography.

Itten, J. *Design and Form, The Basic Course at the Bauhaus* (Thames and Hudson, 1975) [ISBN 0471289302].

Kandinsky, W. *Point and Line to Plane* (Dover, 1980) [ISBN 0486238083].

Naylor, G. *The Bauhaus Reassessed* (The Herbert Press, 1985) [ISBN 0906969298, 0906969301(pbk)].

Poling, C. V. *Kandinsky's Teaching at the Bauhaus; Colour Theory and Analytical Drawing* (Rizzoli, illustrated edition, 1986) [ISBN 0847807800].

2.1 Background

The importance of the Bauhaus in this course includes its attempts to rationalise design and production. The formalisation of these creative ideas lends itself to implementation in computer-aided design and visualisation tools. To understand this we need to review the work of some important individuals and their interactions.

The Bauhaus was founded by the architect Walter Gropius in Weimar in 1919. This school of architecture and design in a small town in Germany was to have a profound effect on artists, designers and art education in both Europe and the USA, leading to long-term influences on society in terms of architecture, interior design and furnishing.

Germany had undergone an industrial revolution following its unification from a number of independent states in 1871. The speed with which Germany had shifted from an agricultural country to an industrial one caused social problems. The population of Germany greatly expanded over the 19th century. Large cities developed where small villages had been and the small cheap dwellings built for the workers led to slum conditions. Transportation infrastructure was expanded, including new railways and roads. Daimler and Benz built their first motor cars in

the 1880s. Major industries such as Krupps expanded from a small steel works in Essen, to an enormous industrial complex manufacturing armaments.

Germany was now an important trading nation and with this rise in importance, there was a related development in German art. Dresden and Munich, followed by Berlin, emerged as artistic centres. The artists in the German Expressionist movement were influenced by the work of Van Gogh and Gauguin with their use of colour to express emotion. Based in Dresden the 'Brücke' artists—Kirchner, Heckel, Bleyl and Schmidt-Rottluff—were influenced also by the linear quality of Gothic art and the fact that the artist carvers were anonymous members of a guild which did not differentiate between art and craft. The Brücke artists wanted their art to "speak to the people". They published a manifesto calling upon youth to revolt against old established ideas.

In Munich a New Artists' Association was formed. Members included Kandinsky, Jawlinsky, Münter and Franz Marc. They organised an exhibition of work by Picasso, Derain and Vlaminck in 1910, but the group broke up and Kandinsky and Marc formed the 'Blaue Reiter' group. In a publication they produced, Kandinsky wrote that distinctions between different art forms should be broken down.

Kandinsky had arrived in Germany from Russia in 1896. He had studied law, but turned to art and art theories, writing "Concerning the Spiritual in Art", a justification for abstract art. Kandinsky had a deep interest in the relationship between sound and colour (there is debate over whether he had the condition of synesthesia), and this has an effect on the development of his art[1]. He had an interest in colour for its own sake, and over his career he moved away from the representation of recognisable subjects and objects. By 1910 he had produced his first abstract painting of basic shapes, lines and forms. In his "Compositions", Kandinsky carefully arranged shapes and colour to attempt to communicate feelings to the spectator, whilst in his "Improvisations", which were more freely painted, he wished to express experiences and feelings.

New ideas concerning the direction of art were developing in other countries. In Russia, Tatlin pioneered constructivism, an abstract art form that made use of machinery and modern materials. In Holland, a group of artists published a journal called De Stijl. Mondrian was the most famous member of the group. The austere, abstract style had more influence on architecture than painting and also an influence on designers working in the Bauhaus.

Walter Gropius (1863–1969) had been a student at the Weimar School of Arts and Crafts when the Belgian architect Henri van de Velde had been its director. Van de Velde pioneered the Art Nouveau style. He designed a building for the school which was opened in 1907, offering courses in printing, weaving, ceramics, book binding and precious metalwork. With growing xenophobia in Germany, van de Velde left his position in 1915, and the school closed in the same year.

Before the outbreak of the 1914–1918 war, Gropius had worked in the design office of Behrens at AEG where he had developed ideas for standardising components for construction and written, with Behrens, a Memorandum on the Industrial Prefabrication of Houses on a Unified Artistic Basis.

[1] http://en.wikipedia.org/wiki/Synesthesia_in_art

2.2 The Beginning of the Bauhaus

In 1907 an organisation called Werkbund, led by Muthesius, an architect, was formed. Muthesius maintained that industry, not the artist, had the energy to make cultural changes. He held that architecture should move towards standardisation. Gropius disagreed with this theory, as he considered that the artist or architect should determine the forms of buildings.

Gropius and his partner Adolf Mayer were successful architects before the First World War. Gropius had designed the model factory for the Cologne exhibition, the Fagus factory[2,3] , furniture and a locomotive. After the war Gropius was asked by the Weimar State Council to formulate his plans for establishing a school of art and architecture. In 1919 Gropius was appointed as director.

2.2.1 Principles for the Bauhaus

Gropius produced the Bauhaus Manifesto to set out his aims for the school.

He wrote that all creative arts were to return to the crafts and there was to be no difference between the artist and craftsman. Architecture was the supreme art form.

Artists must be trained to work for industry. Artists, architects, sculptors and craftsmen should all work to one common goal.

The Bauhaus staff would consist of a master and a journeyman to each workshop, ensuring that techniques as well as design ideas were brought together.

There were to be six categories of craft training:

Sculpture stonemasons, woodcarvers, ceramic workers and plaster casters

Metalwork blacksmiths, locksmiths, founders and metal turners

Cabinet making

Painting and decorating glass-painters, mosaic workers and enamellers

Printing etching, wood-engravers, lithographers and art printers

Weaving

Apart from studies in these areas, students would experience instruction in drawing and painting, including colour theory, the science of materials and basic business studies.

2.3 Bauhaus developments with new staff

Johannes Itten joined the Bauhaus in 1919. He developed the Basic Course of one term's length. Here the students were taught to develop self-confidence. They were taught theories of form with emphasis on the simple basic forms of circle, triangle and square. Compositions were made employing the three shapes. These shapes

[2] http://www.greatbuildings.com/buildings/Fagus_Works.html
[3] http://www.brynmawr.edu/Acads/Cities/wld/06790/06790m.html

were derived from Cubism and were seen as historically primary in art. Also they are independent of nature and easily produced, appearing in Itten's Wood and Metal Workshops. In addition, students learned colour theory in order to understand the expressive qualities of colour and colour contrasts, and consideration of materials and texture. The latter were considered essential for commercial artists and industrial designers.

The *Processing* package, introduced later in this course, can be thought of as a simple workbench providing a basic stock of elementary shapes and colours, together with the tools to combine and manipulate these basic elements, to design and produce novel products. Correspondingly, Bauhaus staff initiated what are now some standard image manipulation techniques. For example, Itten (1975, p.21) has an interesting 'Light-dark analysis of a picture by Goya', where the breakdown is into a regular array of squares that predates modern 'image pixelation' by more than half a century. Another example is 'Happy Island', which is an oil work on canvas[4]. The same Itten text has examples of image kaleidoscoping (1975, pp.56, 57) using regular photographic darkroom techniques, but which can now be simply carried out in standard image processing packages.

The Metal Workshop was founded to develop prototypes for mass production. Gropius maintained that standardisation of goods was the means by which the masses could acquire items, so designs should be suitable for furnishing a house. The workshop was initially led by Itten, then, from 1923, by László Moholy-Nagy. Marianne Brandt and William Wagensfeld achieved the most successful work. Wagenfeld designed table lamps with straight shafts and an opaque glass shade[5]. Brandt produced metal ashtrays, lamps and other household objects. Her lamp reflectors were made of nickel-plated metal, and had moveable shades and arms for good light dispersion[6,7] .

Brandt, who succeeded Moholy-Nagy as director of the Metal Workshop in 1928, was the only woman on the permanent staff. Most women students joined the Weaving Workshop where they experimented with techniques. They created tapestries using a variety of materials and by 1931 made a range of handmade fabrics in muted colours ideal for mass production[8].

Paul Klee and Wassily Kandinsky joined the Bauhaus in the early 1920s. Klee and Kandinsky had both been members of Der Blaue Reiter. Klee developed an independent theory of colour and an analysis of the creative process. His work was derived from nature/landscapes, plants, sea, stars and buildings. Kandinsky continued to work on his theories concerning the "science of art", the underlying elements and themes in discussing a theoretic approach to analysis and synthesis of painting (see, for example, "Point and Line to Plane", (Kandinsky, 1979)). Another good source for examples of shape and colour work is "Kandinsky's Teaching at the Bauhaus" (Poling, 1982). There were debates within the Bauhaus concerning the relevance of these ideas in an institution that placed technology at the heart of experimentation and an analysis of material. However the painters stayed as their fame contributed to the success of the school.

Sommerfeld House was designed by Gropius and Mayer in 1921 for a timber merchant. Three students worked on the interior designs: Joost Schmidt made relief

[4] http://metropolis.co.jp/tokyo/515/art.asp
[5] http://www.bauhaus.de/english/bauhaus1919/werkstaetten/werkstaetten_metall.htm
[6] http://www.architonic.com/mus/8100111/1
[7] http://www.trocadero.com/MuseXX/items/142321/item142321store.html
[8] http://www.bauhaus.de/english/bauhaus1919/werkstaetten/werkstaetten_weberei.htm

carvings on the staircase, Josef Albers designed the stained glass windows and Marcel Breuer designed the furniture. Breuer's furniture was influenced by Reitveld's Red/Blue chair designed in 1917 and illustrated in De Stijl magazine[9].

2.4 Movement towards Constructivism

In 1923 László Moholy-Nagy was invited to teach at the Bauhaus to replace Itten. He was a Constructivist whose work emphasised the importance of the machine. Constructivism, he declared, could expand from an art form into industrial design. Josef Albers had trained as an art teacher before becoming a Bauhaus student. He now began working with Moholy-Nagy on teaching the Preliminary Course where, without using workshop equipment, tasks were given to explore the nature of materials. Moholy-Nagy directed experiments in form. He emphasised that, in an industrial culture, the need to understand the load-bearing properties and other characteristics of materials was essential, linking design and engineering. Moholy-Nagy was interested in the development of photography and reflected light compositions. He experimented with optical and acoustic equipment to make new creations. A starting point in reading about their influence is Borchardt-Hume's edited collection "Albers and Mohol-Nagy: from the Bauhaus to the New World" (2006).

In 1923, a Bauhaus student, Ludwig Hirschfeld-Mach, wrote a score for a colour sonata of three bars using a combination of light and music. Lights and templates were moved in time to the fugue-like music. In 1924 Hirschfeld-Mach wrote:

> *Yellow, red, green, blue in glowing intensity move about on the dark background of a transparent linen screen—up, down, sideways. They join and overlappings and colour blendings result.*
>
> *(Bayer, Gropius and Gropius (1976), p.65)*

Moholy-Nagy experimented with photography, producing photograms and photomontages. He maintained that traditional painting was finished. The move from working on a canvas to creating art through mechanical means meant that artists were no longer involved with producing a piece of art. In 1922 Moholy-Nagy ordered his 'telephone abstract enamels' from a factory. He described these works as "enamel pictures executed by industrial methods"[10].

Moholy-Nagy was also involved with typography and page layout, which was itself an art form. He moved away from static lay-outs to dynamic ones, especially in poster work in the style of Lisitsky, a Russian Constructivist whose poster of 1919 shows a red, triangular wedge (representing the Communists) being driven into a circle of white (the White Russian Army)[11,12] . These symbols could be easily understood even by an uneducated peasant population.

In 1923, while Germany was in the grip of rising inflation, a competition was held in the Bauhaus to design an experimental house to demonstrate the abilities within the school, the design to be chosen democratically by staff and students. Georg Muche, who had joined the staff as a painter, won with a design for a single storey house,

[9] http://www.terraingallery.org/Anthony-Romeo-Chair.html
[10] http://www.mutualart.com/OpenArticle/Mind-the-Design/CA53B9419178DC4A
[11] http://www.sovr.ru/english/show/virtual1.shtml
[12] http://www.allposters.com/-st/Lazar-Lisitsky-Posters_c81955_.htm

named "Haus am Horn"[13]. Constructed from concrete, the main part of the house was the living room lit by a clear-storey. Other, smaller rooms were set around it including a small, easy-clean kitchen with built-in storage and where everything was within reach. The aim was for economy of space, time and energy. The house was furnished by members of the school.

The Bauhaus mounted an exhibition in 1923. There were lectures by Gropius and Kandinsky and performances of the Triadic Ballet by Schlemmer[14] who had painted murals on the walls of the Bauhaus. Music was provided by the Bauhaus jazz band as well as concerts at which works by Hindemith, Busoni and Stravinsky were played. This made Weimar the focus of the avant-garde, but locally Hitler's National Socialists were gaining popularity and they cut the grant to the Bauhaus, forcing it to re-locate to Dessau in April 1925.

Dessau was then an industrial town where Junkers had their aircraft factory. The Bauhaus was amalgamated with the local trade school. Here the course was re-assessed. Moholy-Nagy and Albers ran the preliminary course and Moholy-Nagy also headed the Metal Workshop.

Marcel Breuer headed the Furniture Workshop while two other Bauhaus trained designers, Herbert Bayer and Joost Schmidt, took on the Printing and Sculpture Workshops. Georg Muche was given responsibility for the Weaving Workshop.

The school's aim was to research the needs of modern households and produce relevant designs that industry could produce in mass.

Gropius wrote:

> *The creation of standard types for all practical commodities of everyday use is a social necessity.*
>
> *(Gropius (1926)[15])*

Muche designed a metal house in 1925 while the architects in Gropius's office designed a new Bauhaus building[16,17] consisting of two L-shaped buildings with flat roofs, one to house the students and the other to house the workshops. The workshop had a curtain wall made of glass, that allowed people outside to see what the students were creating.

Also built at this time were houses for the staff. They were made of concrete with flat roofs, large windows and balconies. Each house had a studio. They were set in landscaped gardens which were ten minutes walk from the Bauhaus[18,19] .

In Dessau, the architects undertook a housing project for workers called the Torten Estate[20]. These were state financed and built at low cost. They were small two-storey buildings made of concrete with flat roofs. Three hundred and sixteen one-family units were built, each having three bedrooms, a kitchen-diner and a living room. Central heating, double glazing and built-in cupboards were provided. Each house had a large garden for growing vegetables. These houses were intentionally experimental. Gropius decided that a national plan for housing was

[13] http://www.hausamhorn.de/
[14] http://www.meisterhaeuser.de/en/bewohner_5_schlemmer.html
[15] http://www.mariabuszek.com/kcai/ConstrBau/Readings/GropPrdctn.pdf
[16] http://www.bauhaus.de/english/bauhaus1919/architektur/index.htm
[17] http://www.tu-harburg.de/b/kuehn/wg21.html
[18] http://www.c20society.org.uk/docs/building/bauhaus.html
[19] http://www.tu-harburg.de/b/kuehn/wg21.html
[20] http://www.creen.demon.co.uk/travel/dessau.html

necessary and should include financial planning, study of methods for industrial production, storage of pre-fabricated units and study of efficient use of materials, as well as standardising building components.

Gropius left the Bauhaus in 1927 and his place was taken by Hannes Meyer (Hans Emil Meyer), whose interest was in social housing. Meyer advocated "a technical, not an aesthetic process" to designing buildings. Past styles were to be rejected in favour of modern. He laid stress on collective rather than individual work. He believed that the new house should be pre-fabricated for building on site. Those involved in building schemes should be economists, statisticians, industrial engineers, standardisation designers, heating engineers and even climatologists, before involving an architect. For Meyer architecture should be functional[21].

Moholy-Nagy insisted that designers should see their ideas through to completion and take note of their impact on individuals and society. He foresaw the time when electrically powered machines would reduce labour hours and the labour force required by industry.

Marcel Breuer experimented with furniture made from tubular steel, for domestic use. He welded pieces of steel together to make a chair[22]. Moholy-Nagy photographed the prototype. When it was published in a newspaper, people wanted to buy the chair because it was light, simple, comfortable and inexpensive.

In the Typography Workshop, Herbert Bayer designed the Universal Type[23]. He argued strongly that the use of two alphabets (capitals and lowercase) was unnecessary:

> *why should we write and print with two alphabets? both a large and a small sign are not necessary to indicate a single sound. A = a. we do not speak a capital 'A' and a small 'a'. we need only a single alphabet.*
>
> *(Bayer (1938) quoted from Eskilson (2007) p.276)*

He aimed to produce guidelines for a more precise visual language.

As the depression worsened in Germany, designers began to feel that Meyer was planning to turn the Bauhaus into a trade school. Before the move to Dessau, painters had developed theories of space, form and colour that they taught to the students. Meyer tried to diminish their influence. He increased the staff with architects and began a programme on research into the requirements of social housing. He re-organised the Bauhaus into four departments. Workshops were now to operate for three days. The new departments were building, advertising, interior design and textiles.

Workshops were to become self-financing through commissions. The interior design department, under former student Alfred Arndt, designed low cost furniture for mass production and wallpaper which became very popular and helped finance the department[24].

The textile department liased with the manufacturing industries and Walter Peterhans joined the advertising department where he focused on teaching photography not as an art form but as a science[25].

[21] http://www.bauhaus.de/english/bauhaus1919/architektur/architektur_meyer.htm
[22] http://www.designmuseum.org/design/marcel-breuer
[23] http://www.type.nu/bayer/univer.html
[24] http://bauhaus-online.de/en/atlas/personen/alfred-arndt
[25] http://www.ifa.de/a/a1/foto/ea1bpebi.htm

Meyer was forced to leave the Bauhaus in 1930 by the Nazis, who accused him of allowing a communist cell into the school.

Mies van der Rohr succeeded Meyer. He had worked with Gropius in Behrens' office. He had begun to design skyscrapers and was a supporter of functionalism. The school closed for a period and when it reopened it was more a school of architecture than design.

2.5 The last phase of the Bauhaus in Germany

The Nazis had little sympathy with van der Rohr's ideas on simplicity and functionalism. The school in Dessau was closed and van der Rohr moved the Bauhaus to Berlin. However, after a number of raids by the Nazis the school closed down. Many of those who had worked there eventually settled in the United States of America.

2.6 Summary and learning outcomes

This chapter has given an introduction to the Bauhaus in terms of its development, its main participants, and its influence on the rationalisation of design, manufacturing and production.

With a knowledge of the contents of this chapter and its associated reading you should now be able to:

- name the main person who drove the formation of the Bauhaus, state his profession early in life, name the main area in which he developed ideas at Behrens, and explain how this background shaped his statement of aims for the Bauhaus
- name the main contributors to the development of the Bauhaus and its courses, and briefly describe the background and interests of each of those contributors
- illustrate the practical orientation of the Bauhaus by listing the six categories of craft training, and briefly state what was involved in each
- list the three simple forms utilised in the Basic Course, state their derivation, and give examples of artefacts, designed at the Bauhaus, that utilise those forms
- name two of the fine artists involved in the Bauhaus who contributed to the theory of colour, and describe those contributions
- state who at the Bauhaus was the driving force behind the idea of creating art by mechanical means, and give examples of his work created in this way
- describe and illustrate the influence of the Bauhaus on the design of housing and household artefacts
- describe and illustrate how the Bauhaus influenced trends in design and practice for manufacturing.

2.7 Exercises

Use the reading for this chapter and other relevant texts when working on the following exercises. In your writing, be sure to place any material from sources in

quotation marks and identify the source at the point of use, and provide a full reference list at the end (this is to ensure you avoid plagiarism—see the *Study Support* section of the VLE for further advice). Long quotations have no value in showing understanding or earning marks. In any assessment it is your own contribution in your own words that matters.

1. Discuss the importance of the contributions of Itten and Albers/Moholy-Nagy in the development of the Foundation Course at the Bauhaus. Include consideration of the contrast between the Nature and Machine approaches and their effects on teaching and outcomes for the students.
2. Discuss which of the Itten and Albers/Moholy-Nagy approaches to design teaching lends itself most easily to implementation and expression, with computer-aided tools for pictorial expression, such as the *Processing* package used in this course. Include an appraisal of the benefits, or otherwise, of possible languages of basic shapes (such as that of triangle, circle, square), and their differing emphases on two- and three-dimensional work.
3. Discuss whether or not the approach of Kandinsky offers greater opportunities, or greater difficulties, for artistic expression with a computer-based drawing package, than the approaches of Itten and Albers/Moholy-Nagy.

2.8 The structure of the rest of this guide

The preceeding two chapters have provided some historical and conceptual context, to allow detailed examination of some of the concepts in the use of computer technology in making creative artefacts.

In the next chapter, we will look at the software package *Processing*, and the following chapters will examine various concepts in visual design, that are illustrated using *Processing*. While you work through the material, you should bear in mind that you are expected to become familiar with using *Processing*, to understand the computational and conceptual issues discussed here and also to consider all of the work in a creative context.

Chapter 3

Introduction to *Processing*

Essential reading

Reas, C. and B. Fry *Processing: A Programming Handbook for Visual Designers and Artists* (MIT Press, 2007) [ISBN 0262182629].

Additional reading

Moggridge, B. *Designing Interactions* (MIT Press, 2006) [ISBN 0262134748]. Chapter 1–The Mouse and the Desktop, Chapter 2–My PC.

Packer, R. and K. Jordan (eds) *Multimedia: From Wagner to Virtual Reality* (W. W. Norton and Company, expanded edition, 2003) [ISBN 0393323757]. Chapter 13–Alan Kay, User Interface: A Personal View.

The year 1984 saw the beginning of a major change in the creative industries which was heralded by the arrival of personal computers with graphical user interfaces (GUIs). The Apple Macintosh was the first widely-available personal computer that could display image, sound, speech, music, video and text. This caught the attention of the design industry and very soon sophisticated audio-visual software packages became the main tools of creative professionals. Among these were Adobe's *Photoshop*, for editing and creating images, and Digidesign's *Sound Designer II* for editing audio and music. The **What-You-See-Is-What-You-Get** (WYSIWYG) interface paradigm offered direct manipulation of media objects simply by pointing and clicking.

All media became part of the everyday desktop computing environment and, as a result, computing became an everyday tool in the creative industries. As the capabilities of computers grew, so did the ease with which media could be manipulated. Multimedia computing became a reality in the late 1980s and early 1990s, followed rapidly by the internet and world wide web.

3.1 *Processing*

We will develop our creative tools in a programming language called *Processing*. *Processing* is an open source programming language and environment for programming images, animation and sound. It is widely used by students, artists, designers, architects, researchers and hobbyists for learning, prototyping and production.

The *Processing* open source project was initiated by Casey Reas (UCLA Design/Media Arts Department) and Ben Fry (School of Design, Carnegie Mellon University). It is an outgrowth of ideas started in the Aesthetics and Computation

Group at the MIT Media Laboratory, and inspired by an earlier Java-based language called *Design By Numbers* by John Maeda, who is a world-renowned graphic designer, visual artist, computer scientist and Professor of Media Arts and Sciences at the MIT Media Laboratory.

Introduced in 2005, *Processing* is based on Java, and provides all the functionality that Java offers. It was created to teach fundamentals of computer programming within a visual context and to serve as a software sketchbook and professional production tool. However, it was designed to be far simpler to use than the standard Java distribution. *Processing* has a user-friendly integrated development environment (IDE) and it has many pre-defined methods for performing graphical and multimedia design tasks with very little user-written code.

As open source software, *Processing* is an ongoing project and is developed by artists and designers as an alternative to proprietary software tools in the same domain.

Do I need to know Java to use Processing?

It is not essential to know how to program in Java in order to use *Processing*. This subject guide and the essential reading will give you an adequate knowledge of the *Processing* programming language. However, as the *Processing* language is based upon Java, knowledge of Java will undoubtedly help you. We assume that you are taking the *CO1109 Introduction to Java and Object-Oriented Programming* course at the same time as this course (or have previously studied Java). You should find that these two courses complement and mutually reinforce each other.

Which version of Processing should I use?

At the start of each academic year, we designate the latest version of *Processing* as our **standard version** for this course. Any coursework that you submit will be tested on the standard version, so it is best if you use this version. To find out what version is the current standard for the course, look at the *CO1112 Creative Computing I* pages on the VLE.

Which version of Processing is used in this subject guide?

The current version of *Processing* at the time of writing this guide was 2.1. Any specific descriptions in this guide refer to this version; if the standard version in your academic year is newer, some of the details may be different. Any important differences will be highlighted on the VLE.

3.2 Installing *Processing*

Learning activity

Method 1. From our VLE

You can download the course's current standard version of *Processing* from the VLE. From the home page at `http://computing.elearning.london.ac.uk`, navigate to *Courses→CO1112 Creative Computing I: Image, Sound and Motion*. From there, click on the relevant link to download the installation file.

Method 2. From processing.org

Alternatively, you can download *Processing* directly from the `http://processing.org` website. However, note that the version available there may be newer than the standard version being used for the course. While you may wish to try out the latest version, remember that any submitted coursework will be tested against the standard version of *Processing*. Therefore, you should always check your coursework against the standard version before submitting it.

Once the installation file is downloaded:

Move the *Processing* installation file to a folder that you want to install *Processing* to. For example, on Windows you might try making a directory `C://Program Files/Processing`. Double click on the file and extract the contents to the new directory. See *processing.org/tutorials/gettingstarted/* for further instructions.

You should now make a shortcut to allow you to easily run the program. For example, on Windows this can be done by right-clicking on the `processing.exe` executable file and selecting the "make shortcut" menu item. Select the resulting "processing.exe shortcut" and move it to the desktop. Now you can just go to the desktop and click the *Processing* icon and it will launch the application.

3.3 A Quick Tour of *Processing*

In Computer Science, a *program* is the list of instructions that is written by a human for a computer to execute. In *Processing* this list of instructions is called a *sketch*. This is to emphasize that this programming language is designed for creative computing.

To see what this means, click on the program shortcut that you made in the activity above. Once *Processing* is running click on the File menu and select Examples to bring up a window that lists a large number of example sketches; see Figure 3.1.

Try opening one of the examples, such as Basics→Form→ShapePrimitives, by double clicking on it. When the sketch opens, click on the *Run* button (which looks like a conventional video or audio Play button—an equilateral triangle with a point to the right) at the top left of the window. There are many example sketches for you to try; see the exercises at the end of this chapter.

3.4 Code examples

The following chapters present many examples of *Processing* code to demonstrate particular topics and features of the language. You are encouraged to experiment with these examples, by making changes to see how the output is affected, and by using them as the starting point for developing more complex programs. Many of the example programs can be downloaded from the *CO1112 Creative Computing I* pages on the VLE.

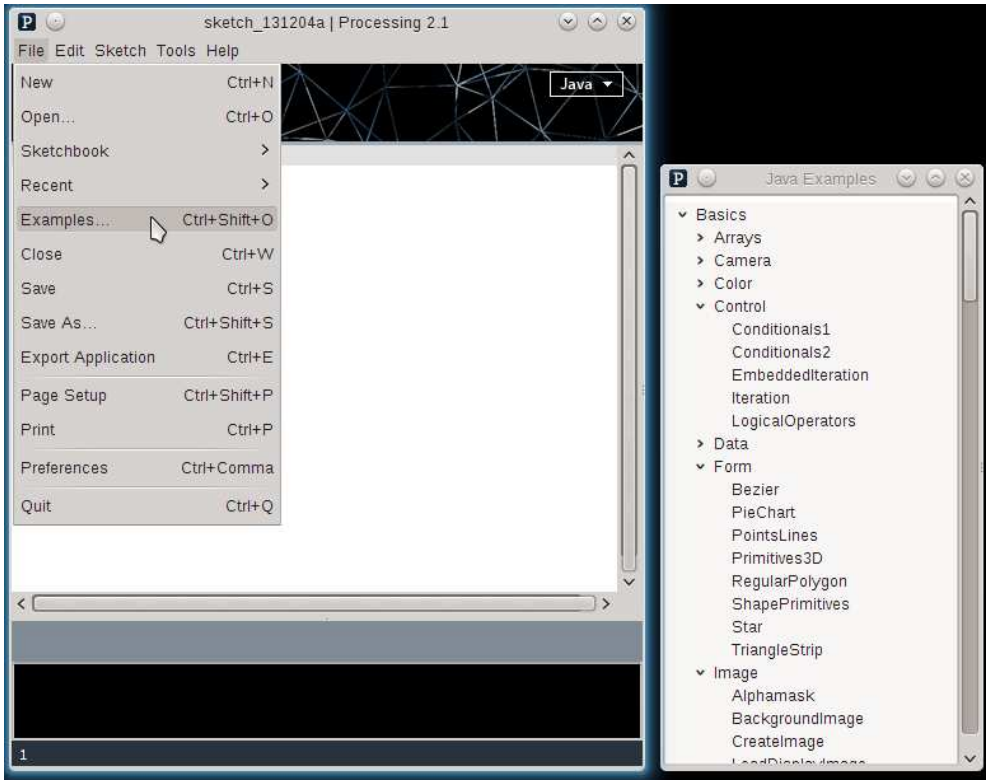


Figure 3.1: Opening the Examples folder in *Processing*.

3.5 Summary and learning outcomes

This chapter has been primarily about introducing you to *Processing*, and describing how to download, install and use it.

You should now be able to:

- explain what *Processing* is and who wrote it
- download *Processing*, install and run it
- open and run the example sketches that are bundled with *Processing*.

3.6 Exercises

Open each of the five sketches listed below, and run them by pressing the *Run* button at the top of the *Processing* application.

Try to understand how they work by looking at the code for each sketch in the text area. You should recognise some of the commands from Java, but you will also notice that *Processing* simplifies standard Java syntax.

Write a brief explanation (two to three sentences) of what each sketch does.

I) Basics→Form→ShapePrimitives
II) Basics→Structure→WidthHeight
III) Topics→Drawing→ContinuousLines
IV) Topics→Drawing→Pattern
V) Basics→Input→MousePress

Chapter 4

Origins

Essential reading

Reas, C. and B. Fry *Processing: A Programming Handbook for Visual Designers and Artists* (MIT Press, 2007) [ISBN 0262182629]. Structure 1: Code Elements, Shape 1: Coordinates, Shape 1: Primitive shapes; Gray values, Development 1: Sketching software.

Additional reading

Wong, W. *Principles of Form and Design* (Wiley, 1993) [ISBN 0471285528]. Chapters 1 and 2.

4.1 Introduction

This chapter introduces the foundations of two-dimensional drawing in *Processing*. The chapter assumes that you have succesfully installed *Processing*, understand how to launch the application, and are able to type in the editing box.

4.2 The *Processing* display window

Visual artists create on a canvas, composers create on staff paper; in *Processing* we create on a *display window*. The display window is a flat rectangular surface with hundreds of thousands of *pixels* (picture elements) that can be controlled individually to produce light at different intensities. In *Processing* we type commands into a text document called a *sketch*. After we make a *Processing* sketch we can *play* it by pressing the **Run** button, as if it were a music player or a video player, for example. Via sketches, we control the individual pixels on the display window. So, the pixels are the raw material that we manipulate in a visual sketch. Note that we may create sound instead of a visual image, but for now we will concern ourselves only with the visual domain.

We must be careful to make a distinction between the display window in *Processing* and the display device that is attached to your computer, be it a laptop, a tablet, a monitor on your desktop or maybe a wide-screen high-definition plasma display. We will call these physical displays.

In *Processing*, the display window is an interface to a window that is displayed automatically, see Figure 4.1. This window is sent to the display device via the operating system. In contrast to Java, where we must explicitly program every

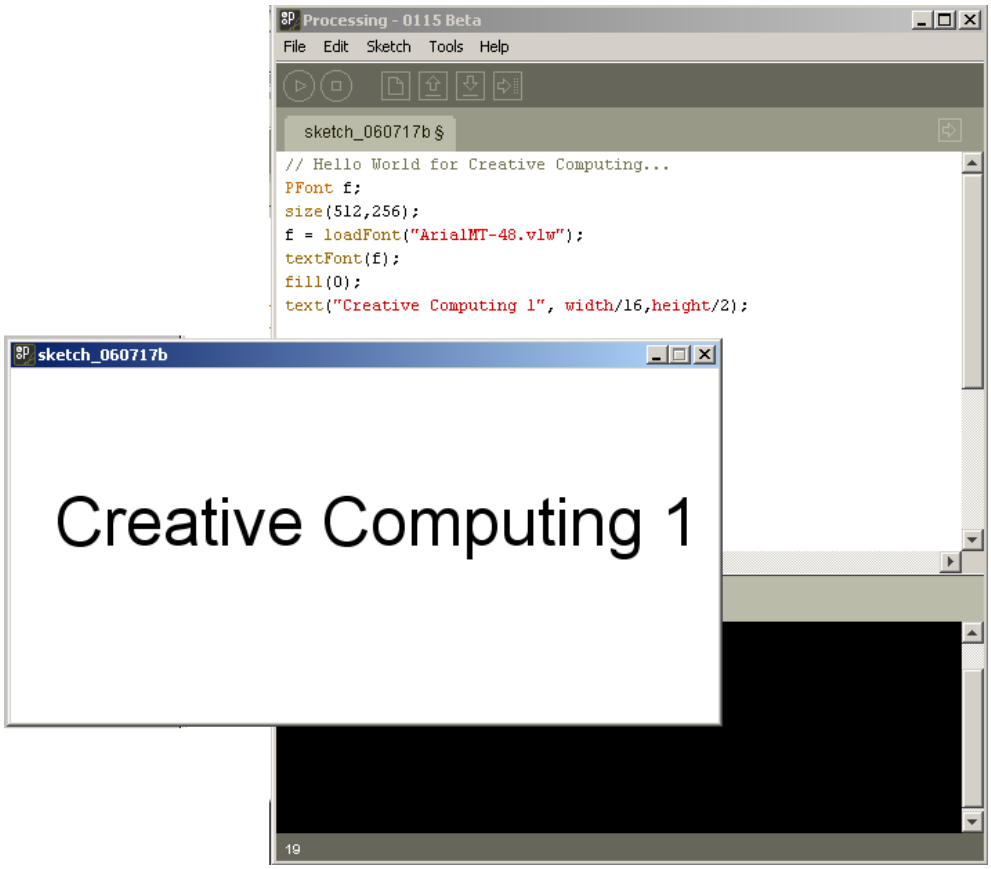


Figure 4.1: The *Processing* environment showing a *sketch* and its corresponding display window.

graphical interface element ourselves, the *Processing* environment will create a window for us automatically, but we must provide the details of the size of the window that we want.

For brevity, throughout this guide we will sometimes refer to the *Processing* display window as the *screen*. However, when we do this, do keep in mind the distinction between this and the physical display, as described above.

Learning activity

In any programming language, it is interesting to see what happens when we provide the system with no input whatsoever. Let's start by seeing what *Processing* does if we *do nothing*.
Make a new *Processing* sketch.
Save the blank sketch to your *Processing* folder.
Try pressing *Run* on your blank sketch.
What happens?

Comments on the activity

We might expect nothing to happen, but something happens. A blank display window is drawn to the screen. What size do you think this display is?

4.3 size()

To change the size of the display window we use a built-in *Processing* command called `size()`. You should go to the *Processing* `Help→Reference` menu item. This launches your web browser with the *Processing* help pages. It is often useful to look at the complete form of the Help pages rather than the abridged form. That way you can see all the built-in commands in *Processing* (see Figure 4.2). Note that to find the Help page entry for any of the built-in commands or the language elements, type the command that you are looking for into a sketch window, highlight the text, right-click and select `Find in Reference` from the pop-up menu.

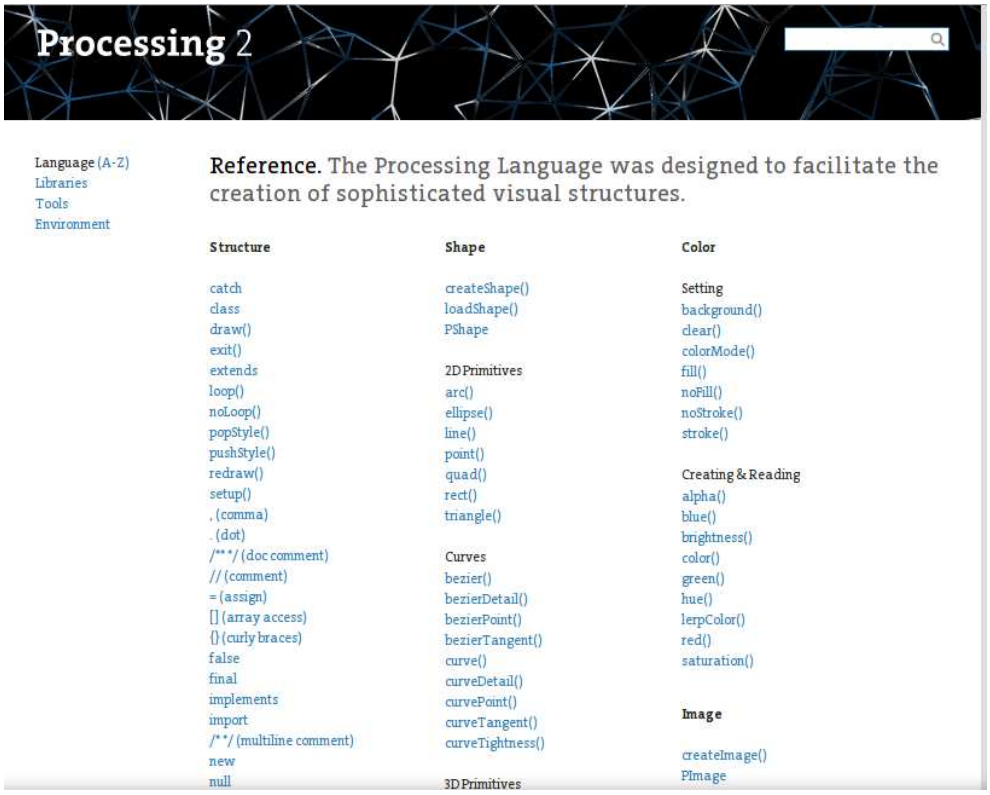


Figure 4.2: The built-in Help pages in *Processing*.

The parentheses () mean that this command is a method that takes arguments. Arguments are simply values that we use to control the behaviour of a method. The full name of the `size()` method is `size(int width, int height)`. This tells us that the `size()` method requires two arguments, both of which are integers, and they control the `width` and `height` of the *Processing* display window.

The width is simply the number of pixels horizontally across the window, and the height is the number of pixels the window will display vertically. For example, the following *Processing* sketch makes a display window that is 128 pixels wide and 512 pixels high:

```
size(128,512);
```

This sketch consists of a single line of code. Try running it. In *Processing*, like many other computer languages, individual chunks of code must be terminated by semi-colons. This makes it easy for the *Processing* program to identify where each statement ends. Line breaks are allowed between parts of a single statement, but statements must be terminated with a semicolon.

Here are some other sizes for you to try. How they appear on your physical display device depends on the type of device you are using, especially its *physical dimensions* and *screen resolution*.

Learning activity

Launch *Processing*.
Make a new folder for your sketches.
Make a series of sketches consisting of display windows of the following different sizes:

```
size(512,128);

size(128,512);

size(1024,128);

size(1024,768);

size(8192, 8192);

size(-1, -1);
```

Did all of these commands produce useful results?
If not, do you understand why not?
Save these sketches to your file system using the `File→Save` menu item.
Re-load the sketches using the `File→Open` menu item.
Try modifying and re-saving some of your sketches.

Comments on the activity

Processing *automatically makes a new folder for your sketch and creates a special file called a* Processing Development Environment (PDE) file *and places it in that folder. You only need to choose the name and the top-level directory where you want to save the sketch;* Processing *does the rest.*

It is good practice to organise your files into directory trees in some systematic way. For example, you might make a directory called `CC1`*, and inside it make a directory called* `Chapter6`*, then save your sketches for that chapter within that folder. This way, it is easier to remember where your sketches are when you want to refer back to them at a later time.*

When developing code in Processing *(or any other language), you should get into the habit of saving copies of your work (“*backing up*”) at regular intervals. The simplest way to do this is to decide on a naming convention for your sketches—for example, you might call the first version of your sketch* `mysketch-version1.pde`*, the next* `mysketch-version2.pde`*, etc.*[1] *That way, if you make changes in your code that you later wish to remove (or, worse, if you accidentally delete your sketch!), you can easily revert to the most recently saved backup version.*

4.4 background()

We now start to explore how to draw in our sketch. Just like paper, or a canvas, we start with a background colour that we specify using the `background()` method.

Learning activity

Try typing and running the following sketch:

```
size(512,512);
background(0);
```

What do you see? Now try the following sketch:

```
size(512,512);
background(100);
```

and finally try:

```
size(512,512);
background(255);
```

Type in the sketches above and run them. What is the meaning of the number in the brackets in the `background()` statements?

Comments on the activity

The number in the brackets is the level of light intensity that is emitted from each pixel in the display window. It can take on values between 0 and 255 where 0 is zero light intensity (black) and 255 is maximum light intensity (white). Values in between are various shades of grey that get progressively lighter as the number gets larger.

Processing's display window is a model for the physical display that you are viewing. So increasing the value of the number argument of the background() method increases intensity of light on the physical display. You are controlling the display using a simple, yet powerful, software interface.

[1] There are more sophisticated ways of backing up your code, for example by using a *Version Control System* such as *Git* (`http://git-scm.com`).

4.5 Coordinates

4.5.1 Cartesian Coordinate System

A `point` is the smallest unit of drawing in *Processing*. Each position on the display window is labeled by its *coordinate pair*, (x, y):

Example

```
size(512,512);
background(255);
stroke(0);
point(256,256);
```

In this example, we have made the window 512×512 pixels, set the background colour to white and the stroke colour to black. Type in this example. Can you see the pixel in the centre?

The single pixel is a little too small for our purposes because the pixels are very tiny, see Section 4.9. Instead, let us increase the size of what we can see by changing the thickness of the stroke:

Example

```
size(512,512);
background(255);
stroke(0);
strokeWeight(10);
point(256,256);
```

Type in this example, what do you see? From now on we shall use a large stroke to make points visible enough for us to see them.

4.6 The Origin

The most important point in the Cartesian coordinate system is the origin. This is the point at (0,0), that is zero in the x-dimension and zero in the y-dimension. In *Processing*, where is the point (0,0)? We shall call the point (0,0) the origin, or O, for short. Not to be confused with 0 (zero).

Once we know the position of the origin, we can draw in the plane using points and lines. The origin in *Processing* is the point in the top left corner.

This placement of the origin at the top left of the screen, with x-values increasing from left to right and y-values increasing down the screen, is common in computer graphics and image processing applications such as *Processing* and the Java two-dimensional graphics libraries. However, this is a different position for the

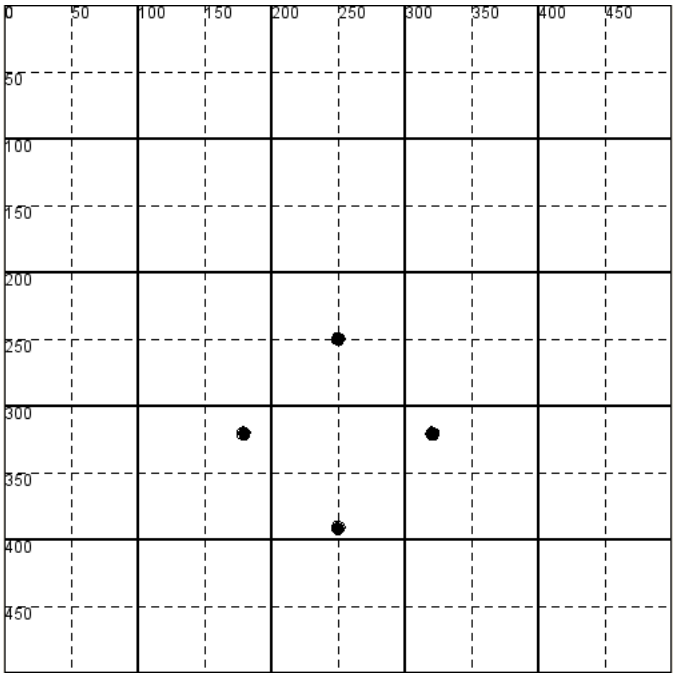


Figure 4.3: The origin in *Processing* is the point in the top left corner, that is point (0,0). The grid is made by placing lines at a regular interval in both the horizontal and vertical directions.

origin than the usual location in geometry, as taught in mathematics courses. In mathematics, the origin is usually at the bottom left with y-values increasing as we move up the page. That is, here we use left-hand axes, while in mathematics it is conventional to use right-hand axes.

4.7 Plane Geometry

In plane geometry, a plane is the area onto which lines and points are drawn. This can simply be thought of as an infinitely large sheet, where each position on the sheet is indexed by a number from zero to the screen width minus one for the x dimension and from zero to the screen height minus one for the y dimension. Cartesian coordinates in two dimensions always occur as pairs of numbers such as $(x1, y1)$ and $(x2, y2)$ for two different points p1 and p2. Figure 4.3 shows a plane with origin at the top left-hand corner.

4.7.1 `point()`

Now that we have a background, we can add foreground content. The concepts of foreground and background will become much more developed later. For now, the background is an initially blank solid-coloured canvas and the foreground is anything that is drawn upon it.

To draw a point we call the `point()` method:

```
size(512,512);
background(0);
stroke(255);
strokeWeight(1);
point(256,256);
```

What do you see? Each `point` on the *Processing* screen corresponds to a pixel on your computer's monitor. A pixel is the smallest unit of change that is possible on your computer's screen. Now try the following:

```
size(512,512);
background(0);
stroke(255);
strokeWeight(1);
point(256,256);
point(256,257);
point(256,258);
point(256,259);
point(256,260);
```

What do you see? What are the dimensions of what you see? Remember, dimensions are measurements of an object's width and height. In this case the dimensions are width=1 pixel, height=5 pixels.

Learning activity

Figure 4.4: *PointSketch*, a simple mouse-controlled point drawing sketch.

Type in the sketch shown in Figure 4.4.
Run the sketch. What do you think it does?
(Hint: try pressing the left mouse button while moving the mouse pointer over the display window.)
What is the name of the method defined in this sketch?

We will discuss the use of methods in later chapters, but how is the method being used by *Processing*?

Figure 4.5: Output of the *PointSketch* sketch.

Comments on the activity

`draw()` is a built-in method. You should look at the help pages for a description of what it is and how it works. Processing *calls the `draw()` method automatically in a loop. This allows us to make interactive drawing applications such as the one shown in this example.*

4.8 Lines

In *Processing*, a line is defined by two points that are joined by filling in all the points in between. The `line()` method allows us to define the two end-points of a straight line:

```
size(512,512);
background(255);
stroke(0);
strokeWeight(10);
line(100,200,200,200);
```

It doesn't matter in which order the two points are specified. We would get the same results with:

```
size(512,512);
background(255);
stroke(0);
strokeWeight(10);
line(200,200,100,200);
```

These two describe the same line. We can check our intuition by looking just at the end points alone:

```
size(512,512);
strokeWeight(10);
point(100,200);
point(200,200);
```

```
size(512,512);
strokeWeight(10);
point(200,200);
point(100,200);
```

It is the same two points, so the same line will be drawn, the direction of drawing is different, but that has no influence on what we see, the same pixels will get coloured in both cases.

4.8.1 Zero-Based Indexing

The screen we've been using is 512×512 pixels. But, the pixels are addressed in the range 0–511. This type of indexing is called zero-based indexing and it is the form of addressing used in Java and in some other programming languages such as C and C++. We must remember that the pixel indexed by the number 512 is actually the 513th pixel; this is off the screen because the last visible pixel is 511. How can we make a screen where pixel number 512, in either the x-dimension or y-dimension, is included on the screen? The answer is that we make the screen at least 513 pixels in its width dimension, x, and height dimension, y. For example:

```
size(513,513);
line(512,0,512,512);
```

Here, all the pixels at position 512 in x or 512 in y are drawn because the screen is 513 pixels wide and 513 pixels tall.

4.9 Size of a pixel

Pixels can be seen as the individual light sources that are packed together on your computer's display to make an image. They are the smallest unit of control that we have in making an image. We start our creative computing journey by learning how to manipulate pixels to draw images on the display device.

Before we embark on drawing to our display device, it will be useful to know how big the pixels are and, therefore, how big the screen that we are drawing to is. In short, we would like to know the *dimensions* of our display and of our created images.

To work out these sizes, we need to know two things. The first is the size of the physical display, and the second is the display resolution. The size of a pixel, then, is its width and height in standard physical units.

Computer displays are measured by the length of the diagonal. For example, I am currently writing this subject guide on a laptop computer that has a 15-inch screen, however, let us work in centimeters for the purpose of international standardisation. An inch is 2.54 centimeters (cm) so 15 inches is $15 \times 2.54 = 38.1$cm, this is the distance from the top left to the bottom right of the screen, a diagonal line.

Figure 4.6 shows the monitor and the diagonal length of 38.1 cm. From the specifications of my laptop given in its user manual, I know that its display has an *aspect ratio* of 4/3; this is the ratio of the width to the height of the screen. This means that the width is 4/3 or 33.33% larger than the height.

Even if we are unable simply to measure the height and width of the display directly, we can calculate these values as long as we know the (diagonal) screen size and aspect ratio.

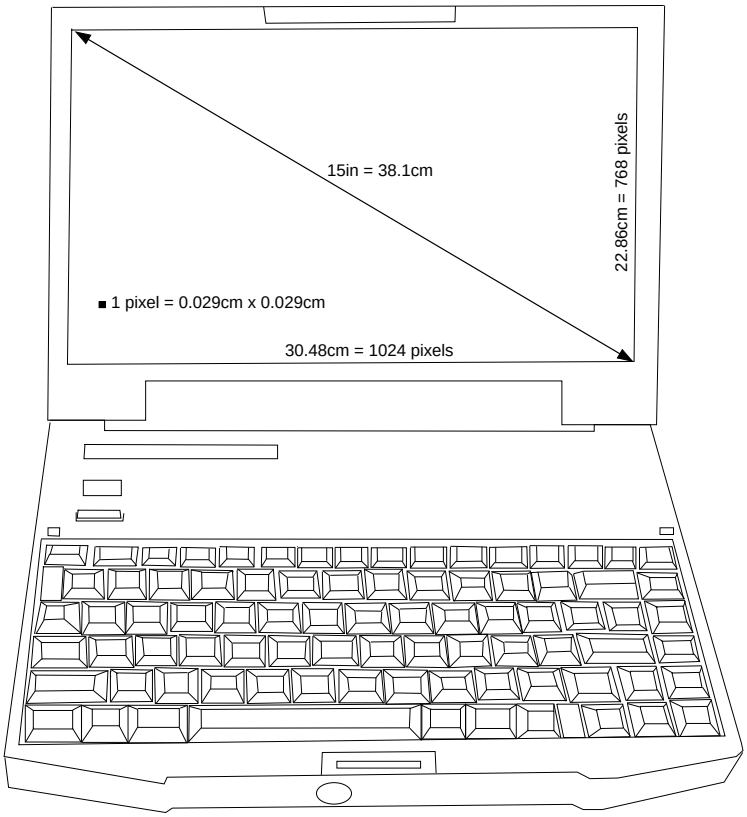


Figure 4.6: The dimensions of a laptop display.

By *Pythagoras' Theorem* we know that the sum of the squares of the width and the height equal the square of the diagonal length. We also know the aspect ratio, so we can write down the following equation:

$$width^2 + height^2 = diagonal^2$$

We know that *width* and *height* are related by $width = aspectRatio \times height$. So we can substitute for width:

$$(aspectRatio \times height)^2 + height^2 = diagonal^2 = (38.1cm)^2$$

Now we rearrange the equation to obtain the height of the screen:

$$(1 + aspectRatio^2) \times height^2 = (38.1cm)^2$$

and finally:

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

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

I know that my laptop display has a resolution of 1024×768 pixels. Now, we can calculate the size of a pixel by dividing the width of the display by 1024, the total number of pixels across the screen, or the height by 768, the total number of pixels down the screen. We find that the width and height measurements of a pixel are each 0.0298cm. That means there are 33.59 pixels active to make a line of pixels that is a centimeter across and $33.59^2 \approx 1129$ pixels to fill a square centimeter ($\approx$ means approximately; the actual answer is 1128.678 pixels but we have rounded the answer to the nearest whole number of pixels).

Learning activity

How many pixels are there in the whole screen assuming the 1024×768 resolution? If stretched out in one long line, how long would this single-line display be?

Comments on the activity

To work out how many pixels there are in total we multiply the dimensions together. For example, $1024 \times 768 = 786432$ *pixels.*
To work out how many centimeters a line of this many pixels would be, we calculate $786432 \times 0.0298 = 23435.6$ *cm– that is* ≈ 234 *meters or 0.23 kilometers.*

As an illustration of how to use *Processing* to work out the answers to the display dimension problem, Figures 4.7 and 4.8 show a *Processing* sketch that computes these dimensions, and its output. Don't worry if you don't understand all the *syntax* for now, you will learn what it all means in your Java course and in later sections of this subject guide. But see if you can find where the diagonal dimension and aspect ratio are specified, and where the answers to the screen width and height, and dimensions of a single pixel are calculated.

While we are looking at this sketch, it is worth highlighting one aspect that is often a source of error for novice programmers. Notice that the variable `A` is defined as:

```
float A = 4.0/3.0; // correct
```

You might wonder why we did not just use:

```
float A = 4/3;     // incorrect!
```

The reason is that 4 and 3 are treated as integers by *Processing* and Java, and the result of dividing two integers is also an integer. So in this case 4/3 would be rounded down to the closest integer, giving a result of 1. This then gets assigned to variable `A`, which is a float, so `A` takes the floating point value 1.0. In order to avoid these integer arithemic problems when we desire an answer as a floating point number, we specify the numbers to be divided as floats from the start. Hence, 4.0/3.0 in this example gives us the answer that we want, which is 1.33333.

```
// Solve the screen pixel size problem
float D = 38.1;                       // Length of diagonal
float A = 4.0/3.0;                    // Aspect ratio
float h = sqrt(pow(D,2)/(pow(A,2)+1)); // Height (cm)
float hPix = 768;                     // Height (pixels)
// Go backwards to check our result
float d = sqrt(pow(A*h,2)+pow(h,2));
// Now print the results
println("All Screen Units in Centimeters");
println("screen diagonal = " + d);
println("screen width    = " + A*h);
println("screen height   = " + h);
println("pixel width     = " + h/hPix);
println("pixel height    = " + h/hPix);
println("pixels per cm   = " + 1/(h/hPix));
println("total pixels    = " + A*hPix*hPix);
println("total pixel cms = " + A*hPix*h);
```

Figure 4.7: screenMath, a *Processing* sketch that calculates the size of pixels for given screen diagonal length D, aspect ratio A and vertical height in pixels $hPix$.

```
All Screen Units in Centimeters
screen diagonal = 38.1
screen width    = 30.48
screen height   = 22.859999
pixel width     = 0.029765623
pixel height    = 0.029765623
pixels per cm   = 33.595802
total pixels    = 786432.0
total pixel cms = 23408.639
```

Figure 4.8: Text output of screenMath sketch.

Learning activity

Type in the *Processing* sketch of Figure 4.7. Look in your computer's User Manual to find the diagonal length and aspect ratio of your display device. Change these values in the *Processing* sketch and run it.
a) How big is a pixel on your display device?
b) How many pixels per centimeter fit on your display device?
c) For a 38.1cm display, with resolution 1024×768 pixels, what is the size of a 512×512 *Processing* sketch in cm^2?

Comments on the activity

Hint: The first two lines define the diagonal length and aspect ratio of the screen. The fourth line defines the screen resolution in the y-dimension.

Learning activity

Type in the following short *Processing* sketch:

```
void setup() {
  println("display: " + displayWidth + "x" + displayHeight);
  println("toolkit: " +
    java.awt.Toolkit.getDefaultToolkit().getScreenResolution() + " dpi");
}
```

Run the sketch and look at the text output.

Comments on the activity

This sketch makes use of the Processing *variables* `displayWidth` *and* `displayHeight`*, which give the dimensions of the entire physical display (not just the* Processing *display window).*

The sketch also calls a library method in the underlying Java AWT toolkit to obtain the Dots Per Inch *(dpi) density of the physical display.*

These methods can be useful when writing code for deployment on devices of unknown display size.

4.10 Summary and learning outcomes

In this chapter, we have seen that we can turn the display of individual pixels on and off using the `point(x,y)` method. Each pixel on the display window has a unique two number address; just like we use latitude and longitude to identify locations on the globe, we need an x-value and a y-value. These values are the number of pixels to count in the x direction from left to right starting at 0, and the number of pixels to count in the y-dimension starting from the top at 0 pixels. Just as the combination of a particular latitude and longitude reading uniquely identifies a specific place on the globe, both the x-value and y-value together uniquely identify a pixel on the display window. The same pair of x-value and y-value will always identify that same specific place on the *Processing* window.

It is possible to turn each individual display pixel on and off to make points on the display window. A series of points in a row makes a line. But the points are very small (how small?) so it takes a lot of points in a row to produce a line that is significant.

To avoid the repetitive turning on and off of pixels to make lines, or other shapes, we use the pre-defined line command as part of a *Processing* sketch. A sketch is entered as statements separated by a semicolon. The semicolon is required for the machine to be able to read and interpret the sketch, and a new line at the end of a statement is optional and is for human readability of the sketch.

The `line()` method takes two pairs of numbers that represent the start point and the end point of the line, and joins them by filling in the pixels in between. So, when we draw lines in computing, we are really drawing a series of points. In this case

we let an algorithm decide which points to fill in to draw the line, so there is already computing going on under our noses. The algorithm for drawing lines is quite complicated and is covered in a later section. For now, it suffices that lines are really a collection of points with the property that the points join the end points by a straight line, the shortest possible pattern of connected turned-on pixels between two points on the display window.

We now have a method for joining two points with a line. In the next chapter, we will see how lines can be made to have many different meanings.

You should now be able to:

- describe what Cartesian coordinates are, and how the origin relates to these
- explain what the *Processing* methods `size()` `background()`, `point()` and `line()` do
- describe what a pixel is and how this relates to the *Processing* display window and to the physical display
- write *Processing* sketches to draw pictures using points and simple lines.

4.11 Exercises

1. In section 4.2 you created a blank sketch, and were asked what size you thought the sketch was. Without guessing or simply using judgement, how would you find out the actual size? Why do you think this particular size is used as a default? Note that when we talk about size, we could be referring to actual lengths, or to numbers of pixels; it is important to become comfortable about both of these uses.
2. In *Processing*, what is the difference between a *sketch* and a display window? Why does *Processing* need both?
3. In Section 4.5.1 you saw an example where we used `strokeWeight()` to change the thickness of the stroke. What is actually happening when this is done? What is the relation between the pixels and the point? At the end of that section we say that we will use a large stroke to make points more visible. Why then is it useful for us to have pixels as small as they are? Would not making the physical size of a pixel bigger improve visibility? What do we lose when we do this?
4. Write a *Processing* sketch that covers the display window with vertical lines. Now modify it to do horizonal ones as well. Think about the distance between the vertical lines, and the distance between the horizontal lines. You could make them evenly spaced, or you could try to make them less regular. See what kinds of pictures you can come up with just through innovative use of drawing lines — this does not purely have to be through spacing; there are many other creative dimensions that you can explore. Also, think about this in terms of the previous chapters on the Bauhaus, and creativity in mathematics. In the next chapter you will learn more about lines in *Processing* and lines as a creative vehicle, but for now, try to see how much you can do with what you already know.
5. Does a bigger physical display size mean there are more or less (or the same number of) pixels on the screen? Dicuss this, and think about what it means for *Processing* and its implementation — will *Processing* run in the same way on computers with different sized screens and screen resolutions?

Chapter 5

`background()`, `stroke()` and `line()`

Essential reading

Reas, C. and B. Fry *Processing: A Programming Handbook for Visual Designers and Artists* (MIT Press, 2007) [ISBN 0262182629]. Shape 1: Primitive shapes; Drawing attributes, Input 1: Mouse I.

5.1 Introduction

The elements of drawing on a computer consist of setting a background colour, setting a foreground colour and selecting the size and shape of pen with which to draw. The same elements are required whether using a graphical drawing tool such as Adobe Illustrator or a programming language such as *Processing*.

We will focus on lines in this chapter because a thorough understanding of line forms the basis of every artist's repertoire of drawing techniques. Line is also a structural concept; once we know how to draw a line we can understand how to arrange other elements into lines, and we can start to use line in our sketches and compositions.

5.2 `line()`

In *Processing*, drawing is defined by a total of seven numbers: the background colour, the stroke colour, the stroke weight (width of the pen) and two points to connect (requiring a total of four coordinates).

As we saw in Chapter 3, *Processing* provides us with a default screen, background colour and stroke colour. The simplest line we can possibly make uses default values for all of these. Such a line requires only four numbers: the two sets of (x,y) coordinates of the end points. For example, the single statement `line(0,0,width-1,height-1)` used in a sketch draws a line in the default stroke colour (black) against the default background colour (a light gray) using the default screen size, 100×100 pixels—see learning activity below. Try it; see Figure 5.1.

The default line is a very useful line because it tells us many things about a drawing program. Perhaps the most important thing it tells us is whether the origin, the point $(0, 0)$, is at the top or bottom of the screen. We know that the x-dimension will always have (0) at the left-hand side by convention. But the vertical, y-dimension (0), can either be at the top or bottom depending on the drawing program. The default line, which is drawn from the origin to the maximum extent of the screen, tells us the answer to where vertical (0) is. The line in Figure 5.1 extends from the

```
line(0,0,width-1,height-1);
```

Figure 5.1: The simplest line in processing using all the default values.

top left to bottom right which immediately tells us the origin is at the top. What type of line would we see if the origin were at the bottom left corner of the screen (as it is in some drawing programs)?

5.2.1 Vertical, Horizontal and Diagonal Lines

The coordinates of the end points of a line determine its orientation on the screen. Horizontal and vertical lines leave one of the coordinates fixed in each end point but vary the other. For example, the vertical line in Figure 5.2 has a fixed x-coordinate for both end points but varies the y-coordinate.

```
size(256,256);
background(128);
stroke(0);
line(128,0,128,255);
```

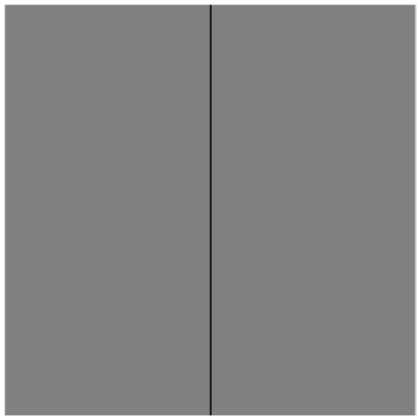


Figure 5.2: A vertical line fixes the x-coordinate but varies the y-coordinate.

Conversely, a horizontal line fixes the y-coordinate but varies the x-coordinate. A line that varies in both coordinates at both its end points will be oriented in a direction other than horizontal or vertical, so it will be diagonal.

Learning activity

In this activity you should make a sketch starting with `size(512,512)`.

Draw a vertical line down the left-hand side of the screen at x-coordinate (0).

Draw a vertical line down the right-hand side of the screen.

What is the x-coordinate of the last column of pixels on the right hand side of the screen?

Draw a horizontal line across the middle of the screen.

Draw a horizontal line across the bottom of the screen.

What is the y-coordinate of the last row of pixels at the bottom of the screen?

Draw a diagonal line that is 20 pixels to the right of the line shown in Figure 5.1.

Draw a diagonal line that is 50 pixels below the line shown in Figure 5.1.

Put all your lines in a single sketch.

5.2.2 background()

There are four method calls in this chapter's examples that control the appearance of a line: `background()`, `stroke()`, `strokeWeight()` and `strokeCap()`.

The `background()` sets the background colour of the screen with 0 meaning black and 255 meaning white. Values in between are shades of grey that are continuous from black to white.

5.2.3 stroke()

The `stroke()` method also sets a colour but it is the colour of the pen. We must set the colour of the pen before drawing with it. The range of possible colours for the pen are the same as for the background. What will happen if we set `background()` and `stroke()` to the same colour?

Learning activity

In a new sketch using `size(512,512)` try to find the value of the default background colour by setting `stroke()` and drawing a line.

How can you tell when you have found the background colour's value?

5.2.4 strokeWeight()

The `strokeWeight()` method selects the size of the pen in pixels; the greater the number of pixels the thicker the line that is drawn. We must set the `strokeWeight()` before drawing with the pen for it to have any effect.

Figure 5.3 on the previous page shows many possible configurations of `background()`, `stroke()` and `strokeWeight()` in a grid of sketches. The lines are drawn with random end points, but can you see the relationship between the

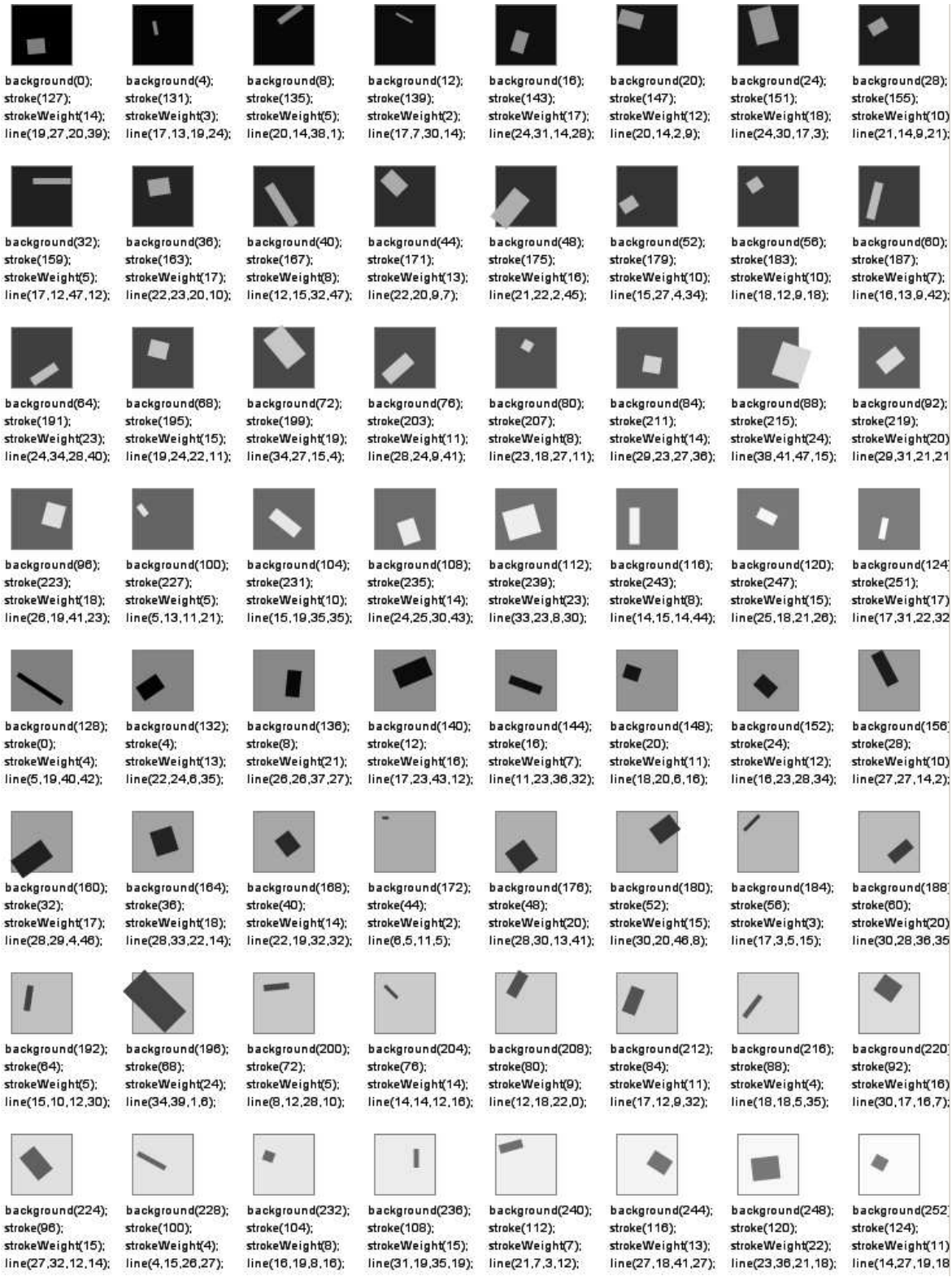


Figure 5.3: `background()`, `strokeWeight()`, `stroke()` and `line()` used in many configurations of a single line.

background() and stroke() colours? To see for yourself look at the code that generated these examples, which is shown in Figure 5.12 on page 54.

5.2.5 Many lines

While there are many possibilities for a single line, things get a lot more interesting with multiple lines. Figure 5.4 shows one possible relationship between two lines and Figure 5.5 shows a simple composition with three lines.

```
size(256,256);
background(192);
stroke(0);
line(128,0,128,255);
strokeWeight(15);
stroke(64);
line(0,161,255,161);
```

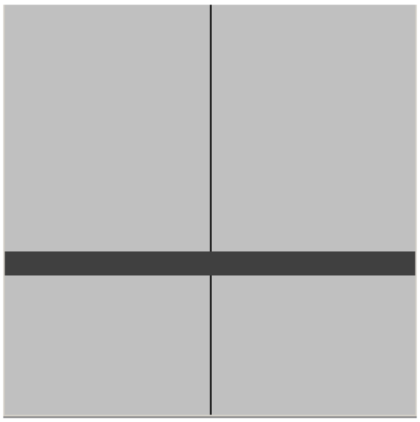


Figure 5.4: Two Lines.

```
size(256,256);
background(64);
stroke(128);
strokeWeight(5);
line(128,0,255,161);
strokeWeight(1);
stroke(255);
line(0,0,width-1,height-1);
strokeWeight(17);
stroke(32);
line(0,128,161,255);
```

Figure 5.5: Three Lines.

5.2.6 strokeCap()

In addition to drawing multiple lines in a composition we can change the way each line looks by changing the shape of the pen. Figure 5.6 shows the three types of strokeCap that are available: ROUND, SQUARE, PROJECT. The first two are straightforward, except that ROUND projects beyond the end of the line, using a semicircle that has a diameter of the current strokeWeight in pixels. For this reason

the third `strokeCap()` type, namely `PROJECT`, is used to create a `SQUARE` end to a line that projects the same amount as a `ROUND` line end. Figure 5.6 demonstrates that lines that end with `ROUND` or `PROJECT` are more likely to fall outside of the shaded background area; this is because the randomly selected end points are chosen to fall within the background area but some of the lines project beyond their end points, due to the `strokeCap()`.

5.3 Snap To Grid

Grids are extremely important tools for drawing, structuring and compositional processes. A grid usually consists of evenly spaced horizontal and vertical lines that extend across the screen; see Figure 5.7 on page 50.

The important aspects of this sketch are the relationships between the coordinates. First take a look at the vertical lines, then the horizontal lines. Do you notice any patterns in the coordinates between the two types of lines? These patterns are fundamental; they are due to the geometry of grids. You should become familiar with them and try to use them whenever arranging a sketch with multiple elements.

Often we don't actually want to draw a grid, but use it as a guide for arranging a composition or sketch. Figure 5.8, also on page 50, shows how we can use a grid to constrain the drawing of a set of lines by hand. Contrast this example with the example in Chapter 4, using mouse movement to draw points to the screen.

Learning activity

Starting with the sketch in Figure 5.8, remove the drawing of the grid but leave the constraint of the grid in the code.

Try increasing the number of lines in the grid by changing `int GL=16` to other values such as 32, 64, and 128. What do you notice as you increase the number of grid lines?

Try decreasing the number of grid lines to 8 and 4. What happens?

How might you draw with the mouse and save the coordinates of the lines that you create?

5.4 Observing and Drawing

Once we have some mastery over basic line drawing techniques a very large world of possibility opens up. Consider Figure 5.9. This is an arrangement of lines that represents a complex shape. In this case, we naturally perceive the shape as a pyramid which is a three-dimensional object. However, we have only drawn seven lines on a two-dimensional screen to achieve the effect.

Knowing where to put lines in order to create a representation of an object, or impression of an object, is what artists are very good at. It is a skill that can take many years to master. However, starting with the basic principles of this chapter,

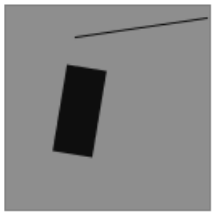
```
background(0);
stroke(127);
strokeWeight(3);
strokeCap(PROJECT);
line(9,12,112,127);
strokeWeight(34);
strokeCap(ROUND);
line(37,35,16,121);
```

```
background(28);
stroke(155);
strokeWeight(14);
strokeCap(PROJECT);
line(23,16,102,109);
strokeWeight(29);
strokeCap(PROJECT);
line(75,34,94,25);
```

```
background(56);
stroke(183);
strokeWeight(2);
strokeCap(ROUND);
line(14,49,63,42);
strokeWeight(13);
strokeCap(ROUND);
line(29,54,98,92);
```

```
background(85);
stroke(212);
strokeWeight(1);
strokeCap(ROUND);
line(36,29,95,112);
strokeWeight(7);
strokeCap(PROJECT);
line(35,34,75,25);
```

```
background(113);
stroke(240);
strokeWeight(8);
strokeCap(PROJECT);
line(35,34,90,70);
strokeWeight(29);
strokeCap(PROJECT);
line(37,39,26,24);
```



```
background(142);
stroke(14);
strokeWeight(1);
strokeCap(PROJECT);
line(44,20,126,8);
strokeWeight(25);
strokeCap(SQUARE);
line(51,39,42,93);
```

```
background(170);
stroke(42);
strokeWeight(3);
strokeCap(PROJECT);
line(30,34,37,65);
strokeWeight(28);
strokeCap(SQUARE);
line(45,49,115,64);
```

```
background(199);
stroke(71);
strokeWeight(5);
strokeCap(ROUND);
line(45,25,127,29);
strokeWeight(5);
strokeCap(ROUND);
line(43,38,70,120);
```

```
background(227);
stroke(99);
strokeWeight(35);
strokeCap(PROJECT);
line(37,58,113,51);
strokeWeight(30);
strokeCap(SQUARE);
line(35,67,68,35);
```

Figure 5.6: Pairs of lines with different `strokeWeight()` and `strokeCap()` values.

```
// A Grid
size(256,256);
background(255);
stroke(0);
line(0,16,255,16); // Horizontal
line(16,0,16,255); // Vertical
line(0,32,255,32); // Horizontal
line(32,0,32,255); // Vertical
line(0,48,255,48); // Horizontal
line(48,0,48,255); // Vertical
line(0,64,255,64); // Horizontal
line(64,0,64,255); // Vertical
// A Border
line(0,0,width-1,0);
line(width-1,0,width-1,height-1);
line(width-1,height-1,0,height-1);
line(0,height-1,0,0);
```

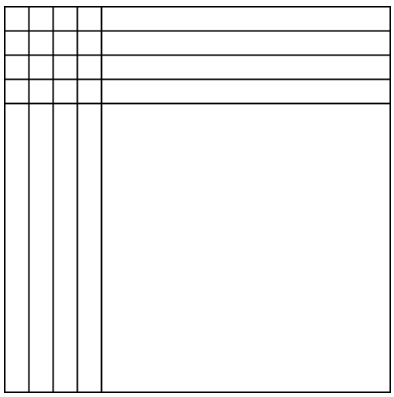


Figure 5.7: A grid can be composed by equally spaced lines in both horizontal and vertical directions.

```
// SNAP TO GRID: A Grid Drawing Tool

int SZ=512;       // The size of the screen
int GL=16;        // The number of grid lines we want (try changing this!)

float GSZ=SZ/GL;  // Calculate the size of each square on the grid
int a=0, b=0;     // Variables for storing previous position of mouse

void setup() {
  // Perform initial set up of the screen and draw the grid
  size(SZ,SZ);
  background(255);
  drawGrid();
  stroke(0);
  strokeWeight(2);
}

void draw() {
  // If mouse pressed, draw a line from the end of the previously drawn line
  // to the grid position nearest to the current mouse position
  if (mousePressed) {
    line(snapToGrid(a),snapToGrid(b),snapToGrid(mouseX),snapToGrid(mouseY));
  }
  // record the current mouse position for future use
  a=mouseX;
  b=mouseY;
}

int snapToGrid(int d) {
  // Round down the given coordinate d to the nearest
  // grid position
  return (int)(round(((float)d)/GSZ)*GSZ);
}

void drawGrid() {
  stroke(200);
  // A single for-loop performs the magic
  for(int k=0; k<SZ; k+=GSZ){
    // A line across the screen, offset in Y
    line(0,k,SZ,k); // X dimension
    // Reverse the coordinates for offset in X
    line(k,0,k,SZ); // Y dimension
  } // Close the loop
}
```

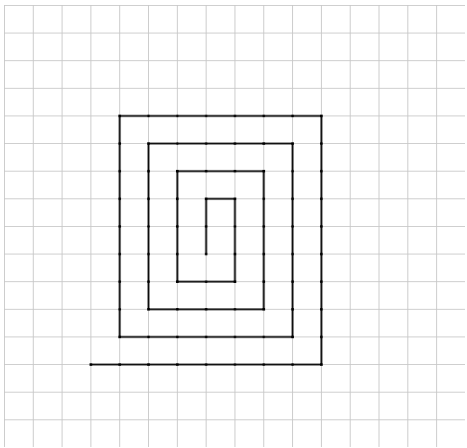


Figure 5.8: Drawing on a grid using constraints on end point positions.

```
size(512,512);
background(255);
stroke(0);
line(204,262,358,337);
line(102,362,256,437);
line(358,337,256,437);
line(204,262,102,362);
line(204,262,256,162);
line(358,337,256,162);
line(102,362,256,162);
```

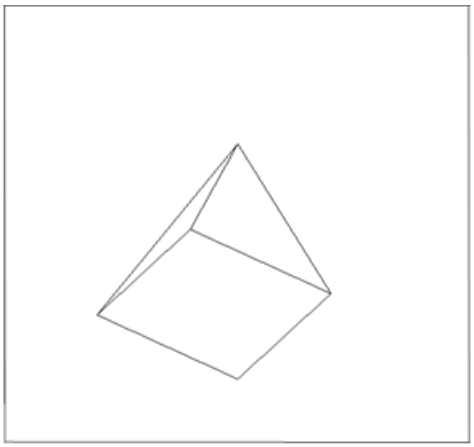


Figure 5.9: Pyramid, adapted from John Maeda's *Design By Numbers*.

you have at your disposal enormous potential to create images and impressions of things that you see.

The most important lesson is to look carefully at the world, and images created by other artists, and to learn from what you see. Try getting into the habit of seeing the lines in everyday scenes, or objects, and try to recreate them in your own sketches.

The drawings in Figure 5.10 and Figure 5.11, on the following pages, are adapted from John Maeda's *Design by Numbers*. Take a careful look at the numbers that have been used to construct the lines. Can you see patterns between lines that are joined versus those that are not joined?

5.5 Observation and practice

It is very important to take an organized approach to creating sketches when they have many elements as in these complex line drawings. A good strategy is to start at the point closest to the origin and work your way around the drawing in clockwise rotation so that each pair of connected lines are next to each other in your code.

It is also important that you keep viewing your sketch as you are coding so that you can quickly spot any errors as they occur. It is sometimes difficult to look at a finished sketch that is incorrect and then go back and find the errors in the code (errors here are lines that are drawn in a different location than you intended).

Learning activity

Using the SnapToGrid sketch, shown in Figure 5.8, with a number of grid lines of your choice, draw some familiar objects such as an open door in a room, a table, a chair and a car in a garage.

Your can print the coordinates of your grid drawing sketches to the *Processing* output console using the `println()` method call in your sketch.

```
size(512,512);
background(255);
stroke(0);
line(271,152,256,122);
line(256,122,276,102);
line(276,102,291,107);
line(291,107,302,127);
line(302,127,286,147);
line(286,147,271,152);
line(250,162,256,282);
line(296,167,307,287);
line(250,162,307,167);
line(250,162,215,197);
line(215,197,168,142);
line(307,167,327,227);
line(327,227,327,282);
line(256,282,307,287);
line(256,287,235,357);
line(235,357,250,447);
line(307,287,312,367);
line(312,367,322,457);
```

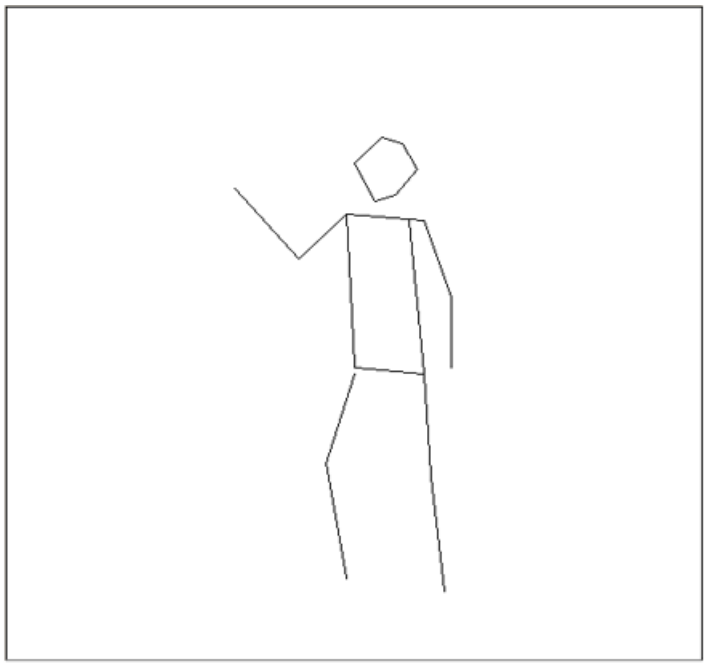


Figure 5.10: Human Figure Drawing, adapted from John Maeda's *Design By Numbers*.

Now make sketches that consist only of lines using the coordinates that you printed (and wrote down) from your grid drawings.

This technique of drawing lines with a mouse, on a grid or not, and then writing down the coordinates is a good way to build up your technique in line drawing.

Practise line drawing every day for at least four weeks; after this, you should continue to practise regularly. Choose simple subjects at first and then develop more complex observations and responses based on what you see.

You will quickly build a style of working that you will find very valuable as you continue your work in creative computing.

5.6 Summary and learning outcomes

In this chapter we first looked at the necessary elements to create a single line: the `background()`, `stroke()`, `strokeWeight()`, `strokeCap()` and coordinates for the `line()`.

We then looked at how patterns in the coordinates of the end points of lines create patterns of lines on the screen. There are lots of ways to draw a line, and a single line can have a lot of meaning, such as dividing the screen or creating part of a border.

```
// John Maeda desk lamp drawing from Design by Numbers
size(512,512);
background(204);
stroke(0);
line(512,472,128,402);
line(102,387,256,332);
line(286,322,348,302);
line(102,387,102,422);
line(102,422,460,487);
line(225,442,225,512);
line(256,472,256,512);
line(276,457,276,512);
line(261,342,286,337);
line(256,337,179,212);
line(256,312,209,237);
line(276,312,209,212);
line(174,207,327,112);
line(209,212,312,137);
line(348,82,373,112);
line(348,82,332,92);
line(373,112,358,122);
line(322,97,343,137);
line(343,137,337,187);
line(337,187,419,137);
```

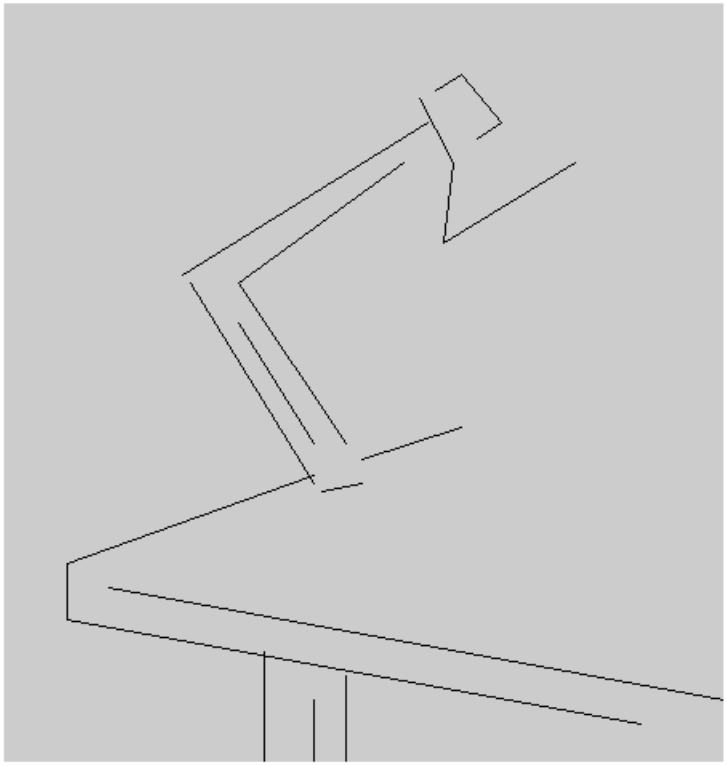


Figure 5.11: Desk Lamp, adapted from John Maeda's *Design By Numbers*.

```
PFont P = loadFont("LiberationSans-10.vlw");
size(768,1024);
strokeCap(SQUARE);
textMode(MODEL);
smooth();
int N=8;
int dx=width/N;
int dy=height/N;
textFont(P);
for (int x=0; x<N*N; x++) {
   stroke(127);
   int f=floor(((float)x/(N*N))*256);
   strokeWeight(1);
   fill(f);
   rect(((x%N)+0.25)*dx,((x/N))*dy,0.5*dx,0.5*dx);
   int sw=floor(random(dx/4))+1;
   int s=(floor(((float)x/(N*N))*256)+127)%255;
   strokeWeight(sw);
   stroke(s);
   int x1=floor(random(0.37*0.5*dx))+sw;
   int y1=floor(random(0.37*0.5*dx))+sw;
   int x2=floor(random(0.5*dx));
   int y2=floor(random(0.5*dx));
   line(((x%N)+0.25)*dx+x1,((x/N))*dy+y1,((x%N)+0.25)*dx+x2,((x/N))*dy+y2);
   fill(0);
   text("background("+f+");", ((x%N)+0.2)*dx, ((x/N)+0.5)*dy);
   text("stroke("+s+");", ((x%N)+0.2)*dx, ((x/N)+0.5)*dy+14);
   text("strokeWeight("+sw+");", ((x%N)+0.2)*dx, ((x/N)+0.5)*dy+28);
   text("line("+x1+","+y1+","+x2+","+y2+");", ((x%N)+0.2)*dx, ((x/N)+0.5)*dy+42);
}
```

Figure 5.12: *Processing* code to generate many configurations of a single line, this code uses the `random()` method to create random end points within the constraints of each background rectangle. To use this code you must first generate the `font` by selecting `Tools→Create Font` from the *Processing* editor's menu bar.

We then investigated how to use multiple lines to make simple compositions in our sketches, and extended the use of multiple lines to drawing with a mouse and constraining the end points of lines to a grid.

Finally, we looked at how complex drawings can be produced with just the right collection of lines and discussed how you can set about learning to develop your drawing skills by observing and drawing things that you see in the world around you.

You should now be able to:

- describe how lines are drawn in *Processing*
- use *Processing* to create both simple and complex drawings, using lines in a variety of ways

5.7 Exercises

1. Figures 5.9, 5.10 and 5.11 are recognisable images of objects, which are made of a series of lines. Try drawing an image of your choice, using *Processing*, that is made of lines. You could, for example, try to draw a table, a cat, a cube, a doorway. In this exercise you are trying to develop both your technique using *Processing*, and your creativity.
2. With the object you have created in the previous exercise, try out different stroke weights, shapes, etc. to experiment with the different properties available for lines in *Processing*. See where this can take you creatively.

Chapter 6

Shape

Essential reading

Reas, C. and B. Fry *Processing: A Programming Handbook for Visual Designers and Artists* (MIT Press, 2007) [ISBN 0262182629]. Shape 1: Primitive shapes, Math 3: Trigonometry, Transform 1: Translation, Transform 2: Rotation, Shape 2: Vertex, Math 4: Unexpected numbers.

Additional reading

Wong, W. *Principles of Form and Design* (Wiley, 1993) [ISBN 0471285528]. Chapters 1 and 2.

6.1 Introduction

In this chapter, you will learn how to use shapes as building blocks for drawing more complex compositions. Shape is an extension of the idea of line because we combine many lines in a closed *loop* to make a shape. Shape is fundamental to visual design and there are many books about construction of complex designs by combining simple shapes that are called *Unit Forms*.

6.2 Unit Forms

Unit forms are the material from which compositions are made; see Figure 6.1, for instance. In this composition, the unit forms are all rectangles. The unit forms are repeated within a composition but there are many variations on the basic shapes over the entire composition.

There are methods built into *Processing* for drawing primitive shapes such as rectangles and ellipses, and there are different methods for more complex shapes.

6.3 Construction of Simple Polygons

A shape bounded by a sequence of straight line segments is called a polygon. Simple polygons include triangles, rectangles, squares, pentagons and so forth. In the field of mathematics, a great deal of effort has been put into understanding the construction of various types of polygon. In classical Greek mathematics, for

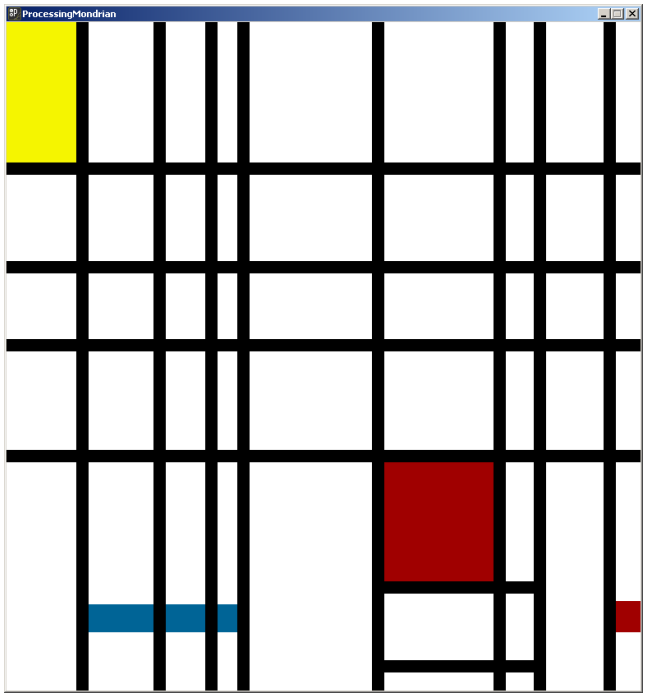


Figure 6.1: Composition with Red, Yellow and Blue 1921, after Piet Mondrian. Reproduced using *Processing* (originally Oil on canvas. 39 x 35 cm).

example, polygons were studied by a system of construction that employed only a straight edge and compass to accurately draw them.

To begin with, we will study the built-in methods in *Processing* that draw the regular polygons.

6.3.1 `rect()`

The `rect(x,y,len1,len2)` method under default conditions draws rectangles with top left position `(x,y)`, width `len1` and height `len2`. The default appearance is a white rectangle with a black border. Use the `background()`, `stroke()`, `strokeWeight()` and `fill()` methods to control the appearance of the rectangle. Use the `rectMode()` method to control the interpretation of the `x` and `y` parameters of `rect()`; for example, it can be used to specify that `x` and `y` define the centre point of the rectangle rather than its top left corner.

Figure 6.1 is a reproduction of a painting by the artist Piet Mondrian called "Composition with Red, Yellow and Blue" painted in 1921. The entire painting is composed of rectangles, most of which appear as thick black lines. The lines are arranged to form an irregular grid pattern; they are not evenly spaced but are placed in a non-simple arrangement with respect to each other. Some of the rectangles are filled with colour, the rest are filled with the white background. The three colours Red, Yellow and Blue are sometimes called the artist primaries as they are often used as the basis for colour in visual arts practice.[1]

[1] We discuss colour and colour perception in more detail in Chapter 1 of Volume 2 of this subject guide. The topic is explored even further in Chapter 1 of Volume 2 of the *Creative Computing II* subject guide.

```
// Piet Mondrian : Composition with Red, Yellow and Blue
size(772,814);
background(255);          // White background
noStroke();               // No outline
// Coloured patches
fill(245,245,0);          // Yellow Patch
rect(0,0,91,177);
fill(0,100,150);          // Blue Patch
rect(92,709,200,34);
fill(160,0,0);            // Red Patches
rect(452,528,150,160);
rect(734,705,66,38);
stroke(0);                // Draw the lines in black
strokeWeight(15);         // Make a ten pixel thick pen
// Vertical lines (these lines are rectangles; that is, thick lines)
line(92,0,92,height-1);
line(186,0,186,height-1);
line(249,0,249,height-1);
line(288,0,288,height-1);
line(452,0,452,height-1);
line(600,0,600,height-1);
line(649,0,649,height-1);
line(734,0,734,height-1);
// Horizontal lines
line(0,178,width-1,178);
line(0,298,width-1,298);
line(0,393,width-1,393);
line(0,528,width-1,528);
line(452,688,648,688);    // These lines are shorter
line(452,784,648,784);
```

Figure 6.2: *Processing* code for Mondrian sketch.

Figure 6.2 shows the sketch which is composed of both `line()` and `rect()` elements. In the sketch we see that lines with thickness appear as filled rectangles; i.e. the thick black lines. In this sketch we have made the black lines thick using the `strokeWeight(15)` and `line()` method calls. We could also have made the coloured rectangles using the `line()` command. For example, the first coloured rectangle was made using `rect(0,0,91,177)` and using the `fill()` method to make the fill colour yellow. The same rectangle would result using either a horizontal line of length 91 pixels and `strokeWeight(177)`, or a vertical line of length 177 pixels and `strokeWeight(91)`.

Learning activity

Using the sketch in Figure 6.2, replace the `line()` method calls with the `rect()` method.
Apart from renaming the method call, what other steps must you take to make `rect()` draw the correct rectangles?
Processing does not have a `square()` method. How can you draw squares?
Can you draw a transparent (non-filled) square using the `line()` method? Explain your answer.

6.3.2 `ellipse()`

The `ellipse(x,y,len1,len2)` method is similar to the `rect()` method, except that it produces a curved shape. Figure 6.3 gives an example of constructing ellipses that

```
size(300,300);
noFill();
strokeWeight(2);
// Circles
ellipse(161,161,161,161);
ellipse(161,161,261,261);
ellipse(161,161,61,61);
//Ellipse
ellipse(161,161,261,161);
ellipse(161,161,161,261);
ellipse(161,161,61,161);
ellipse(161,161,161,61);
```

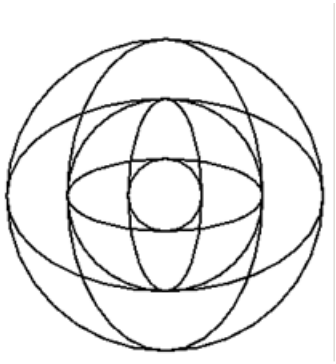


Figure 6.3: Nested ellipses.

are nested inside each other. Notice that an ellipse with equal width and height is a circle.

Similar to `rectMode()`, the `ellipseMode()` method controls the intrepretation of `x` and `y` in the `ellipse()` method. Note, however, that by default `x` and `y` specify the *centre point* of the ellipse; this is in contrast to the `rect()` method, where `x` and `y` specify the top left corner by default.

6.3.3 `arc()`

The method `arc(x,y,len1,len2,ang1,ang2)` is almost the same as for an ellipse but adds two more parameters. Its use is illustrated in Figure 6.4. If the parameters are set to `arc(x,y,len1,len2,0,TWO_PI)` then the method is the same as `ellipse()`. The last two parameters of the `arc()` method are the starting angle and the final angle for drawing a segment of an ellipse. Angles are given in radians rather than degrees; so the angles `(0,TWO_PI)` describe an arc that starts at angle 0, the right-most point of the ellipse, extending all the way around to $2 \times \pi$ radians in a clockwise direction, which is the full ellipse. To go half-way around the angles would be `(0,PI)`. Note that *Processing* uses the pre-defined constants `PI` and `TWO_PI` to mean π and $2 \times \pi$ respectively.

Figure 6.5 shows the four compass-points on a circle and their values in radians as represented in *Processing*. Note that angles `0` and `TWO_PI` are the same point on the circle. In fact, all angles that are an integer multiple of `TWO_PI` ($2 \times \pi$) apart are the same angle. However, the `arc()` method requires the fifth argument to be strictly less than the sixth argument for the arc to be drawn.

The *Processing* methods `radians()` and `degrees()` can be used to convert between radians and degrees.

6.3.4 Construction of Regular Polygons using Turtle Graphics

Regular polygons are special cases where the edges are of equal length, as are the angles between them. Apart from the special case of a square and an equilateral triangle, *Processing* does not have built-in commands to construct regular polygons.

```
size(420,420);
strokeWeight(2);
// Arcs
stroke(255,0,0);
arc(161,261,261,420,PI,TWO_PI);
stroke(0,255,255);
arc(161,261,261,161,0,PI);
stroke(0,255,0);
arc(161,261,100,161,PI,TWO_PI);
stroke(255,0,255);
arc(161,261,100,61,0,PI);
stroke(0,0,255);
arc(161,261,37,61,PI,TWO_PI);
stroke(255,255,0);
arc(161,261,37,22.5,0,PI);
```

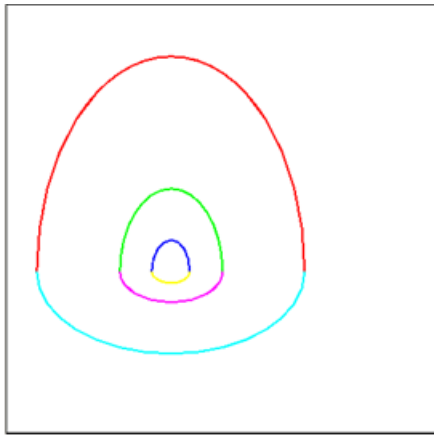


Figure 6.4: Nested arc sections.

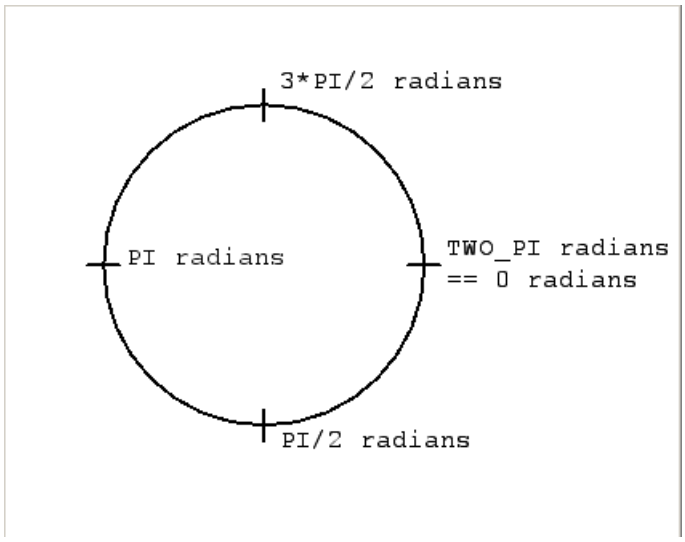


Figure 6.5: Positions around a circle (or ellipse) in radians in *Processing*. Positive angles describe *clockwise* rotations.

For example, there are no `pentagon()`, `hexagon()` or `dodecagon()` methods for creating polygons with five, six and twelve edges respectively.

Instead, we can construct regular polygons by drawing a sequence of lines of equal length and rotating the lines through equal angles. In principle, this sounds easy. However, in practice it requires some geometry to calculate the points required for drawing polygons. There are many textbooks and websites that discuss construction of polygons in two dimensions using Cartesian coordinates. *Processing* offers a very simple and convenient method to construct regular polygons by rotations and line drawing; the method is similar to the *Turtle Graphics* approach introduced by Seymour Papert in the 1970s with the Logo programming language. This approach is widely used for teaching graphics programming to beginning students.

To draw any regular polyhedron, a shape with *N* edges of equal length, we first find the rotation angle for each edge; this is simply *TWO_PI*/*N*. Now we choose a length for our edges, say 100 pixels and a number of edges, say 5. Then we repeat the following process 5 times:
`line(0,0,100,0);translate(100,0);rotate(TWO_PI/5);`.
Note that the position of the origin in *Processing* determines that the y-dimension values increase as we go down the screen. Consequently, positive angles describe clockwise rotations in *Processing* as opposed to the counter-clockwise direction of angles in most mathematics texts. Once you have mastered the sequence of line lengths and rotations for drawing a polygon, you can draw any regular polygon using the above recipe. Figure 6.6 shows a sketch for drawing a pentagon using the turtle graphics method.

```
// Pentagon using "turtle graphics"
size(200,200);
background(255);
noFill();
rect(0,0,width-1,height-1);
translate(37,37);
strokeWeight(2);
// 1
line(0,0,100,0);
translate(100,0);
rotate(TWO_PI/5);
// 2
line(0,0,100,0);
translate(100,0);
rotate(TWO_PI/5);
// 3
line(0,0,100,0);
translate(100,0);
rotate(TWO_PI/5);
// 4
line(0,0,100,0);
translate(100,0);
rotate(TWO_PI/5);
// 5
line(0,0,100,0);
translate(100,0);
rotate(TWO_PI/5);
```

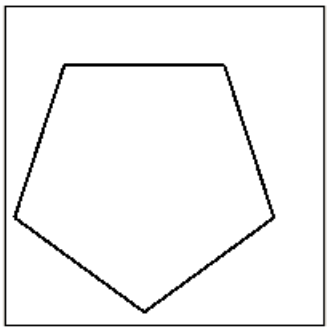


Figure 6.6: Construction of a pentagon using `line()`, `translate()` and `rotate()`.

The figure shows the construction of a pentagon using the `translate()` and `rotate()` methods. To understand these methods, it is useful to think of a turtle (a small animal) that is initially facing right positioned at the top left corner. Each translation and rotation moves the turtle by the set amount. Now each line is drawn *relative to the turtle's position*, so if we rotate the turtle and translate her by 100 pixels, then a line at the *turtle's origin* is different from a line at the screen origin. The turtle's origin is wherever the turtle happens to be.

Learning activity

Using Figure 6.6 as a guide, construct the following regular polygons:

i) Hexagon (six sides)

ii) Heptagon (seven sides)

iii) Octagon (eight sides)

iv) Nonagon (nine sides)

v) Decagon (ten sides)

How many sides can you get up to? Which shape does your many-sided polygon start to resemble as the number of sides increases?

6.3.5 Construction of Irregular Polygons

Irregular polygons have one or more non-similar edge lengths or angles. For these, it is not possible to repeat exactly the same sequence of `line()`, `translate()` and `rotate()` statements to draw the shape.

Processing offers three convenient methods to calculate the angle and length of the last connecting line given a sequence of points that describe the edges of a shape. They are `beginShape()`, `endShape()` and `vertex(x,y)`. In Figure 6.7, a series of vertices (points) describe an irregular shape that is then completed automatically to create a closed shape.

Learning activity

Using Figure 6.7, what happens when you use `fill(0)` before calling `beginShape()`?

```
// Irregular polygon
size(512,512);
background(255);
noFill();

beginShape();
vertex(61,161);
vertex(61,61);
vertex(161,100);
vertex(261,161);
vertex(161,261);
endShape(CLOSE);


beginShape();
vertex(random(512),random(512));
vertex(random(512),random(512));
vertex(random(512),random(512));
vertex(random(512),random(512));
endShape(CLOSE);
```

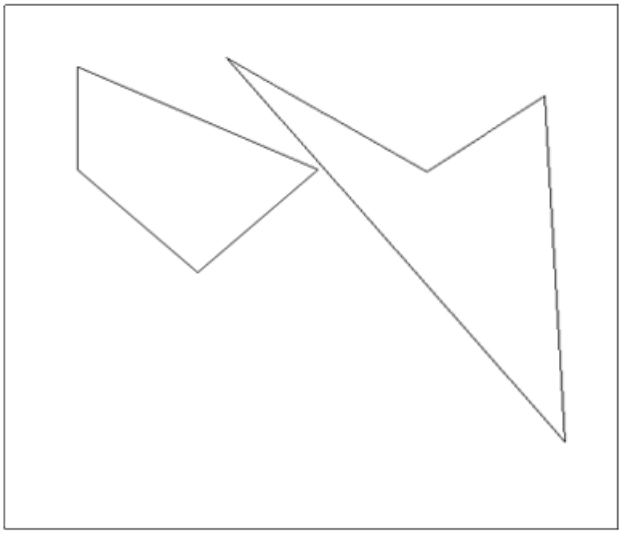


Figure 6.7: Construction of irregular polygons using automatic closure.

`beginShape()` accepts an argument, such as `beginShape(TRIANGLES)`. Try each of the arguments from the following list and describe what they do.

i) `TRIANGLES`
ii) `QUADS`
iii) `POINTS`
iv) `LINES`
v) `TRIANGLE_STRIP`
vi) `QUAD_STRIP`
vii) `TRIANGLE_FAN`

Construct some of the regular polygons described above using `beginShape()` and `endShape()`. The `translate()`, `line()` and `rotate()` methods do not work between the `beginShape()` and `endShape()` method calls. You will need to perform some geometry to calculate the vertex coordinates for your regular polygons.

6.4 Summary and learning outcomes

In this chapter, we discussed various types of shape and how to construct them using *Processing*. We have also investigated how unit forms can be combined to yield more complex forms, and that these more complex forms can be structured into elaborate compositions. It is with this structuring process that we are concerned in the next chapter.

You should now be able to:

- distinguish between, and use appropriately, the methods `rect()` and `line()` for constructing rectangular shapes
- use and descibe methods for constructing curved shapes
- construct a variety of polygons, both regular and irregular, using *Processing*
- use lines and shapes to construct creative artefacts.

6.5 Exercises

1. In Section 6.2, a program that draws a likeness of Mondrian's *"Composition with Red, Yellow and Blue"* was presented. You will look in detail at colour later in the course, but now is a good time for you to begin to understand how colour works, both creatively and from a medium perspective.

   Find out what you can about *additive* colour and *subtractive* colour. What is meant by each of these terms? What are the true additive primary colours? What are the true subtractive primary colours? (While it is true that red, yellow and blue are often called the primary colours, and they are also asethetically pleasing in combination, you should have discovered that they together are neither the additive nor the subtractive primary colours, but elements of both. Look also at Figure 6.4, which contains both the additive and the subtractive primary colours.)

   Now take the code for the Mondrian sketch and modify it to use, for the coloured patches, either the additive or the subtractive primary colours. What part of the code do you need to modify to do so? And how do these changes affect the visual impact of the image?
2. In Figure 6.7, the `endShape()` statement is given the parameter `CLOSE`, while `beginShape()` has no parameters. What happens when you use `endShape()` with no parameter? Look at the *Processing* reference pages to see what should happen, and then try some examples to test it out.

   Can you use `fill()` to put colour into an irregular polygon? Can you do this for a regular polygon constructed using turtle graphics? How do you do this?
3. Can you `fill()` a shape that is not closed? What happens if you try to do this?

Chapter 7

Structure

Essential reading

Reas, C. and B. Fry *Processing: A Programming Handbook for Visual Designers and Artists* (MIT Press, 2007) [ISBN 0262182629]. Shape 1: Drawing order, Data 1: Data types; Variables; Processing variables, Math 1: Arithmetic; Operator precedence; Constraining numbers, Math 3: Trigonometry, Control 2: Iteration, Transform 1: Translation, Transform 2: Rotation, Image 1: Display, Structure 2: Continuous evaluation; Controlling the flow, Structure 3: Abstraction; Creating functions, Shape 3: Recursion, Appendix D: Bit, Binary, Hex.

Additional reading

Behrens, R. R. *Art, Design and Gestalt Theory*, Leonardo, Vol. 31, No. 4, pp. 299–303, 1998.

van Campen, C, *Early Abstract Art and Experimental Gestalt Psychology*, Leonardo, Vol. 30, No. 2, pp. 133–136, 1997.

Wong, W. *Principles of Form and Design* (Wiley, 1993) [ISBN 0471285528]. Chapters 3 and 4.

Zakia, R. D. *Perception and Imaging: Photography—A Way of Seeing* (Focal Press, 4th edition, 2013) [ISBN 0240824539]. Chapters 1 and 2.

7.1 Introduction

In the last chapter, you explored methods to draw different types of shape. These individual forms can be used to build up larger and more complex drawings. In this chapter, you will learn about the principles of structuring shapes in relation to each other.

7.2 Gestalt Principles

The Gestalt school of psychology was founded by Max Wertheimer in the early 20th century and has profoundly influenced the way modern philosophers and psychologists think about how humans see objects in relation to each other. There are many correspondences between the development of Gestalt psychology and the development of the Bauhaus—discussed in Chapter 2—at around the same time; see van Campen (1997) and Behrens (1998) for some interesting discussion on this topic. Psychologists also discovered that the Gestalt principles apply to sound in speech and music perception (see Zakia (2004)).

The most important guiding principle upon which Gestalt pyschology is based is that the human brain sees, or hears, the *whole* form, not just the parts. The

observations made by the Gestalt school about the way humans see and hear are often thought of as rules of perception. Gestalt principles are widely used in the visual and musical arts and are the basis of composition: putting forms together to make a *whole* picture.

Figure 7.1 shows one of the most well-known observations from Gestalt psychology. The circles are drawn using an arc that does not extend the entire way around. Instead, a carefully drawn pattern of triangular notches are left in each circle. The effect is to create the perception of a large triangle between the circles, that is not drawn at all. The Gestalt principle is at work here: the triangle is simply implied by the forms that are drawn. Figure 7.2 shows a similar effect. Here a sphere is perceived but is not drawn; the conical forms (implied by triangles with curved bases) imply the spherical form which is perceived. This type of relationship is called *Reification* in Gestalt psychology.

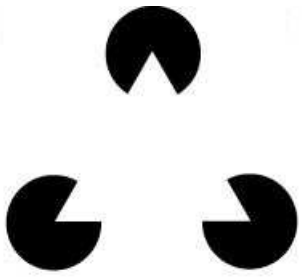


Figure 7.1: Gestalt Principle of Reification. The circles are drawn to suggest the presence of a triangle, even though the triangle is not drawn.

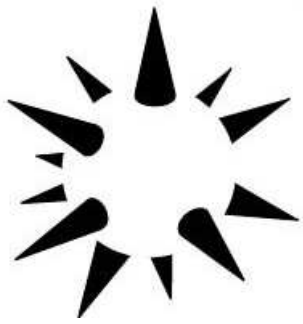


Figure 7.2: Reification of a 3D form. The triangular forms are arranged to suggest the presence of a sphere, even though the sphere is not drawn.

Figure 7.3 shows how the figure/ground relationships in a composition can be obscured. The Gestalt Principle of Multistability is the observation that this figure can be perceived either as a white vase (figure) seen against a black background (ground) or we can see this as a pair of heads (figure) against a white background (ground). Control over the perception of figure and ground is an important skill for the artist and applies just as much in the digital domain as in the traditional visual arts.

The lesson to be learnt from the Gestalt princples is that the placement of forms in relation to each other has a profound impact on the way we perceive the whole

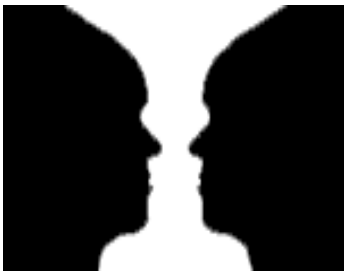


Figure 7.3: Gestalt Principle of Multistability. The image can be seen as a vase or as two human heads facing each other by a switch in figure and ground perception.

picture. This chapter explores ways of putting together unit forms that express a higher concept or shape.

7.2.1 Proximity

The simplest of the Gestalt principles is that of relationships by proximity. Those forms that are closer are seen as grouped together. The principle applies hierarchically, so that many levels of grouping are sometimes seen. Figure 7.4 shows an arrangement of circles such that a large square is perceived on the left of the figure. On the right, we see three vertical columns of circles that are two-circles wide. The hierarchical aspect is that there is a clear division between left and right parts of the composition so there are two sides, each made up of 36 circles, and the right-hand side has three groups of 12 circles. All of these groupings are caused by some circles being closer to each other than others. Do you see any other groupings in this picture?

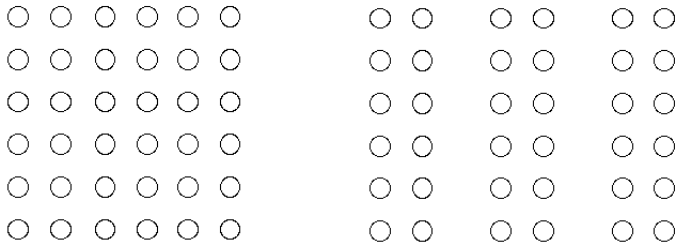


Figure 7.4: Gestalt Principle of Proximity. Forms that are closer are seen as grouped together.

7.2.2 Similarity

The Gestalt principle of similarity is more complex in that it requires there to be something that is unchanged, or changed very little, between forms for them to be perceived as similar. Figure 7.5 shows an arrangement of circles such that the perceived grouping is caused by the similar colours in each row of circles. It is close to impossible to see the picture any other way.

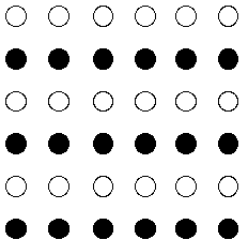


Figure 7.5: Gestalt Principle of Similarity. Forms that are similar are seen as grouped together. Here we see rows instead of a square of circles.

7.2.3 Closure

The principle of closure (or good continuation) is the observation that lines do not have to be continuous to define a form. This is illustrated in Figure 7.6: the lines are broken, yet the perceived forms seem complete. We see a circle and a rectangle as complete percepts. This principle suggests that we can play with line as a way to suggest form without explicitly drawing the entire object.

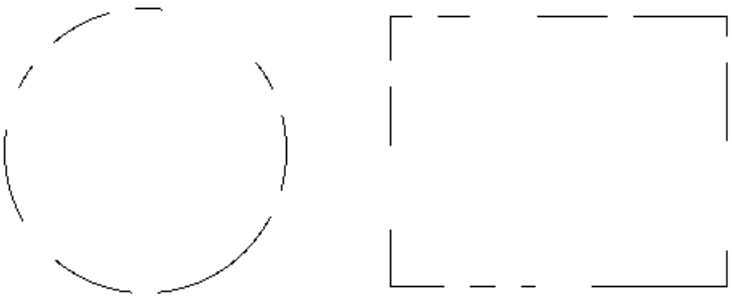


Figure 7.6: Gestalt Principle of Closure. We tend to see whole shapes as closed forms even if the lines that define them are broken.

Learning activity

Make a *Processing* sketch of the first figure in this chapter showing the Gestalt Principle of Reification, Figure 7.1.

Hint: use the `arc()` method to draw the incomplete circles; from a starting angle `StartAngle` the distance around the circle is `StartAngle+5*TWO_PI/6`. An equilateral triangle has equal-length sides, use a variable such as `float length`. The horizontal position of the top vertex, relative to one of the bottom vertices, is `length/2`. The vertical distance from the base to the top is `-sqrt(3)*length/2`, in *Processing* this distance is negative relative to the base line because the upwards direction corresponds to decreasing the y-coordinate.

7.3 Interrelationship of Unit Forms

We now go on to look at how to program the basic ways in which unit forms are placed in relation to each other using proximity relationships. We go on to examine structuring principles that build on these relationships. The proximity relationships are *Disjoint, Proximal, Overlapping and Conjoined*. Each of these concepts should be considered as examples of Gestalt principles as shown above.

7.3.1 Disjoint

```
// Disjoint forms
size(256,256);
background(255);
fill(0);
ellipse(width/4,height/2,width/3,width/3);
ellipse(3*width/4,height/2,width/3,width/3);
```

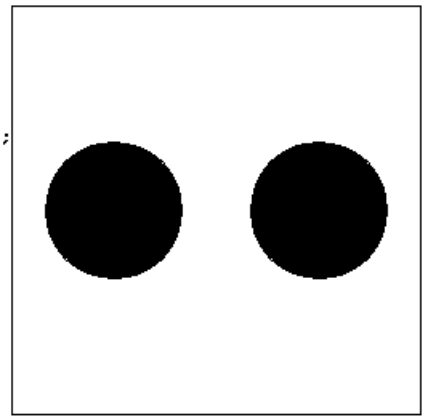


Figure 7.7: Disjoint forms.

Two forms are said to be disjoint if they appear as two separate entities, as in Figure 7.7. The notion of disjointness is inextricably bound to the Gestalt principles since it requires that the forms be placed at a certain distance from each other, but the distance for separateness is dependent upon the size of the forms and the size of the composition.

You will learn by observation, and by your own judgement, how to place forms such that they appear disjoint.

7.3.2 Proximal

When two forms come into non-overlapping contact we call them proximal. In Figure 7.8 it is clear that there is a proximity relationship by virtue of their closeness. Unequal sizes with proximal interrelationships of form suggests *hierarchy*, and can be used to signify parent/child or other unequal relationships.

```
// Proximal forms
size(256,256);
background(255);
fill(0);
ellipse(width/3,height/2,width/3,width/3);
ellipse(2*width/3,height/2,width/3,width/3);
```

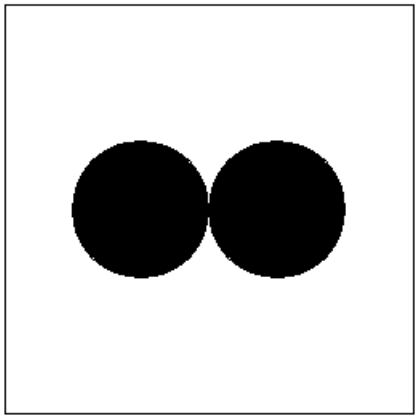


Figure 7.8: Proximal forms.

7.3.3 Overlapping

The idea of overlapping forms is conveyed by drawing form boundaries with stroke colours that are different from their interior fill colours, as in Figure 7.9. Note that the stroke colour is the same as the background colour in this example. The form that is drawn last appears to be on top of its predecessor.

```
// Overlapping forms
size(256,256);
background(255);
fill(0);
stroke(255);
ellipse(width/3,height/2,width/2,width/2);
ellipse(2*width/3,height/2,width/2,width/2);
```

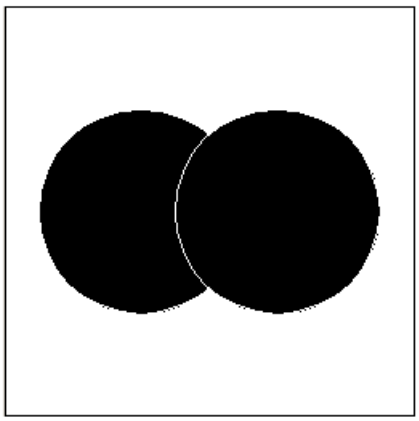


Figure 7.9: Overlapping forms.

7.3.4 Conjoined

If we remove the stroke in the overlapping portion of the two forms then we obtain a new form that is the union of the unit forms. This is a way to combine unit forms into more complex forms, as shown in Figure 7.10.

```
// Conjoined forms
size(256,256);
background(255);
fill(0);
ellipse(width/3,height/2,width/2,width/2);
ellipse(2*width/3,height/2,width/2,width/2);
```

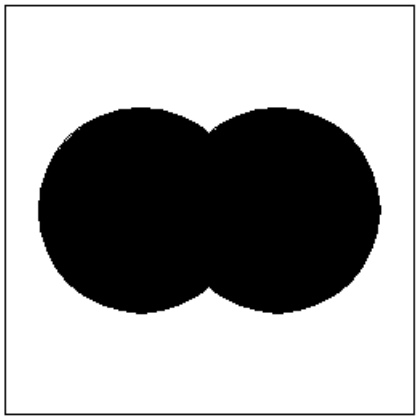


Figure 7.10: Conjoined forms.

7.4 Logical Combination

More complex forms arise when we colour the areas of intersection of two unit forms in different ways. In this section we look at doing this using binary logic.

7.4.1 A brief introduction to colour representation

We will explore in detail the use of colour in *Processing* in Volume 2 of the subject guide. Until then, the following quick introduction will help you to understand the examples presented in the remainder of Volume 1.

The colour value for each pixel is represented in hexadecimal notation as a 32-bit number of the form AARRGGBB; these are the alpha value (AA), red (RR), green (GG) and blue (BB) colour values respectively. The alpha value, which controls opacity, is described more fully in Section 9.2.

If you are unfamiliar with hexadecimal (base 16) notation, use a web search engine to find out about it. Each hexadecimal character, represented by a symbol from the set $\{0-9, A-F\}$, corresponds to a decimal number from 0 to 15. In other words, it represents 4 bits of information ($2^4 = 16$). Each pair of hexadecimal characters therefore represents 8 bits of data, or one byte. In *Processing*, hexadecimal numbers can be specified by using the prefix '0x' (e.g. 0xFF represents the decimal number 255).

We can therefore represent a 32-bit colour value using an 8-symbol hexadecimal number; for example, 0xFFFF0000 represents an alpha value of (decimal) 255, red value of 255, and green and blue values of 0, in other words, a fully opaque red.

7.4.2 Bitwise logical operations on colour values

Processing provides bitwise operators such as OR (|), AND (&), XOR (^) and NOT (~). (At the time of writing, the bitwise XOR and NOT operators are not listed in the *Processing* reference manual, but they are defined by the underlying Java language upon which *Processing* is built.) These operations can be applied to RRGGBB colour values to perform conventional set-theoretic operations such as union, intersection and complement. In the examples that follow, we are drawing the Venn diagrams that represent the set-theoretic operations being illustrated.

7.4.3 Or

The bitwise OR operator ($a|b$) is used for the union operator. Each pixel in the output image gets the logical OR of each pair of input pixels, see Figure 7.11. The bitwise OR operator acting on an 8-bit word, used for colour representation, is applied component-wise on each bit. So if we start with 11000111 and 00000100 as input sequences we get 11000111 as the output.

```
// Image OR Two Layers

PImage P1,P2;
int bg=0xFF000000; // Background colour
int fg=0xFFFFFFFF; // Foreground colour
int len1,len2;

void setup() {
  size(512,512);
  noLoop();
  noStroke();
  len1=(8*width/13);
}

void draw() {
  // Draw the first ellipse
  background(bg);
  fill(fg);
  ellipse(width/3,height/2,len1,len1);
  P1=get();        // store first ellipse in P1

  // Draw the second ellipse
  background(bg);
  fill(fg);
  ellipse(2*width/3,height/2,len1,len1);
  P2=get();        // store second ellipse in P2

  // Combine the two images
  imageOR(P1,P2);  // P1 = P1 | P2

  // And finally draw the result on the display window
  image(P1,0,0);
}

// imageOR method
// The method imageOR modifies the left image (first argument)
// by OR-ing each pixel with a pixel in the right image (second argument)
void imageOR(PImage I1, PImage I2) {
  for (int i=0; i<min(I1.width,I2.width); i++)
    for(int j=0; j<min(I1.height,I2.height); j++)
      I1.set(i,j, I1.get(i,j) | I2.get(i,j) );
}
```

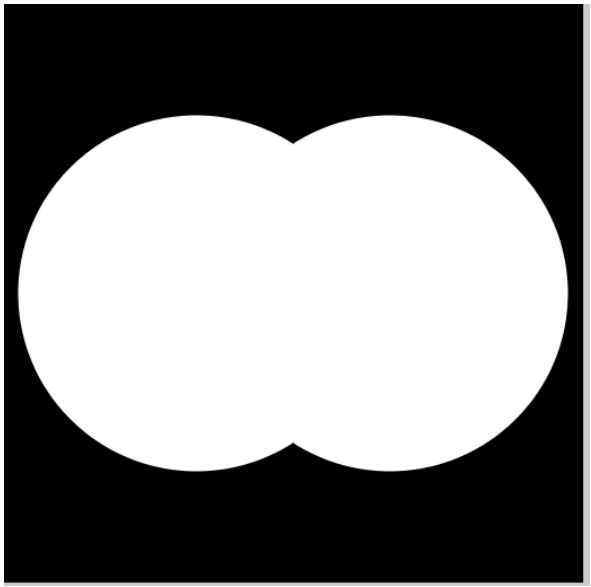
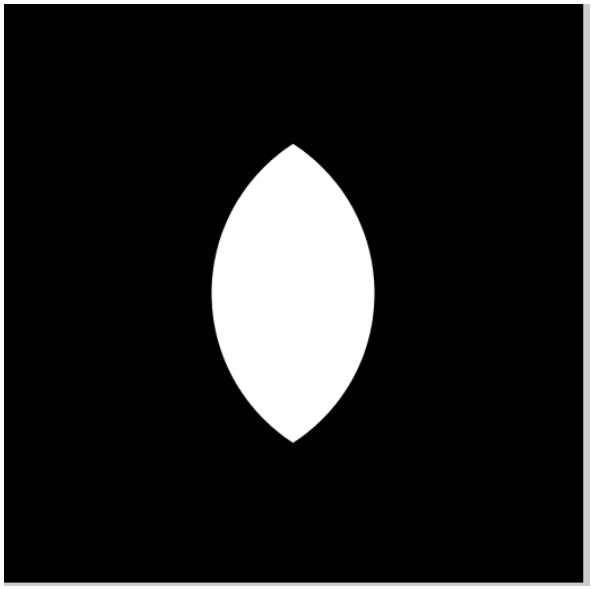


Figure 7.11: Using the bitwise OR operator (|) to blend layers.

7.4.4 And

The bitwise AND operator (*a*&*b*) performs the intersection of two forms, as in Figure 7.12. If the two bytes to compare were 11000111 and 00000100 the output would be 00000100; only one bit is in common between the two.



```
// Image AND Two Layers

PImage P1,P2;
int bg=0xFF000000; // Background colour
int fg=0xFFFFFFFF; // Foreground colour
int len1,len2;

void setup() {
  size(512,512);
  noLoop();
  noStroke();
  len1=(8*width/13);
}

void draw() {
  // Draw the first ellipse
  background(bg);
  fill(fg);
  ellipse(width/3,height/2,len1,len1);
  P1=get();        // store first ellipse in P1

  // Draw the second ellipse
  background(bg);
  fill(fg);
  ellipse(2*width/3,height/2,len1,len1);
  P2=get();        // store second ellipse in P2

  // Combine the two images
  imageAND(P1,P2);  // P1 = P1 & P2

  // And finally draw the result on the display window
  image(P1,0,0);
}

// imageAND method
// The method imageAND modifies the left image (first argument)
// by AND-ing each pixel with a pixel in the right image (second argument)
void imageAND(PImage I1, PImage I2) {
  for (int i=0; i<min(I1.width,I2.width); i++)
    for(int j=0; j<min(I1.height,I2.height); j++)
      I1.set(i,j, I1.get(i,j) & I2.get(i,j) );
}
```

Figure 7.12: Using the bitwise AND operator (&) to blend layers.

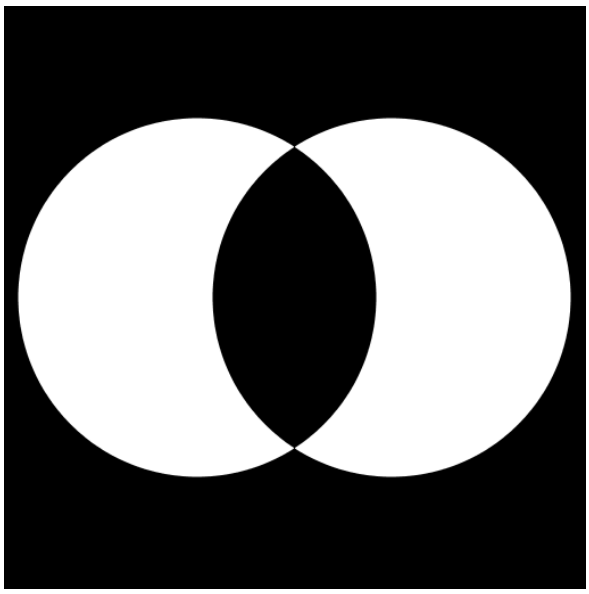
7.4.5 Exclusive Or (XOR)

The combined operation of union minus intersection is a very interesting case. It is so useful in art, mathematics, computer science and machine learning theory that the operation has a special name; it is called the Exclusive Or operator (XOR).

The bitwise XOR operator (a^b) gives the output value 1 when only one of its inputs is 1. Which bit is 1 does not matter, as long as there is only one. The XOR of 11000111 and 00000100 is 11000011.

We see that the bitwise XOR operator acting on two images is the same as the union (OR) minus the intersection (AND), which is the expression `(a|b) & ~(a&b)`, see

Figure 7.13. We use the XOR operator to invert the region of intersection in overlapping forms; this can be used to create many interesting effects in visual compositions; see Figure 7.14 on page 77.



```
// Image XOR Two Layers

PImage P1,P2;
int bg=0xFF000000; // Background colour
int fg=0xFFFFFFFF; // Foreground colour
int len1;

void setup() {
  size(512,512);
  noLoop();
  noStroke();
  len1=(8*width/13);
}

void draw() {
  // Draw the first ellipse
  background(bg);
  fill(fg);
  ellipse(width/3,height/2,len1,len1);
  P1=get();        // store first ellipse in P1

  // Draw the second ellipse
  background(bg);
  fill(fg);
  ellipse(2*width/3,height/2,len1,len1);
  P2=get();        // store second ellipse in P2

  // Combine the two images
  imageXOR(P1,P2);  // P1 = P1 ^ P2

  // And finally draw the result on the display window
  image(P1,0,0);
}

// imageXOR method
// The method imageXOR modifies the left image (first argument)
// by XOR-ing each pixel with a pixel in the right image (second argument)
// The alpha channel of each pixel is reset to FF after applying the XOR
// operation to ensure the resulting image is opaque; this is done by
// OR-ing the result with 0xFF000000.
void imageXOR(PImage I1, PImage I2) {
  for (int i=0; i<min(I1.width,I2.width); i++)
    for(int j=0; j<min(I1.height,I2.height); j++)
      I1.set(i,j, (I1.get(i,j) ^ I2.get(i,j)) | 0xFF000000);
}
```

Figure 7.13: Using the bitwise XOR operator (^) to blend layers.

7.4.6 Not (Inversion)

As a final example of logical operations on images we can use the bitwise NOT operator to invert or complement an image. If the input is 11000111 then the bitwise NOT operator (~a) flips the bits so that the output will be 00111000. Figure 7.15 shows the effect of inverting the image of Figure 7.13.

```
// Multiple image XORs combined with rotation

PImage P1,P2;
int bg=0xFFFFFFFF; // Background colour
int fg=0xFF000000; // Foreground colour
int len1,len2;

void setup() {
  size(512,512);
  noLoop();
  noStroke();
  len1=(8*width/13);
  len2=(8*height/13)/3;  // len2 is shorter than len1
}

void draw() {
  // Draw the first ellipse (long in x dimension)
  background(bg);
  fill(fg);
  ellipse(width/3,height/3,len1,len2);
  P1=get(); // store first ellipse in P1

  // Draw the second ellipse (long in y direction)
  background(bg);
  fill(fg);
  ellipse(2*width/3,height/3,len2,len1);
  P2=get(); // store second ellipse in P2

  // Combine the two images and store result in P1
  imageXOR(P1,P2); // P1 = P1 ^ P2

  // Rotate the coordinate frame
  translate(width/2,height/2);
  rotate(-PI/4);
  translate(-width/2,-height/2);

  // Draw the rotated image and store result in P2
  image(P1,0,0);
  P2=get();

  // Now XOR The Original and Rotated Images
  imageXOR(P1,P2);
  image(P1,0,0);
}

// imageXOR method
// The method imageXOR modifies the left image (first argument)
// by XOR-ing each pixel with a pixel in the right image (second argument)
// The alpha channel of each pixel is reset to FF after applying the XOR
// operation to ensure the resulting image is opaque; this is done by
// OR-ing the result with 0xFF000000.
void imageXOR(PImage I1, PImage I2){
  for (int i=0; i<min(I1.width,I2.width); i++)
    for(int j=0; j<min(I1.height,I2.height); j++)
      I1.set(i,j, (I1.get(i,j) ^ I2.get(i,j)) | 0xFF000000 );
}
```

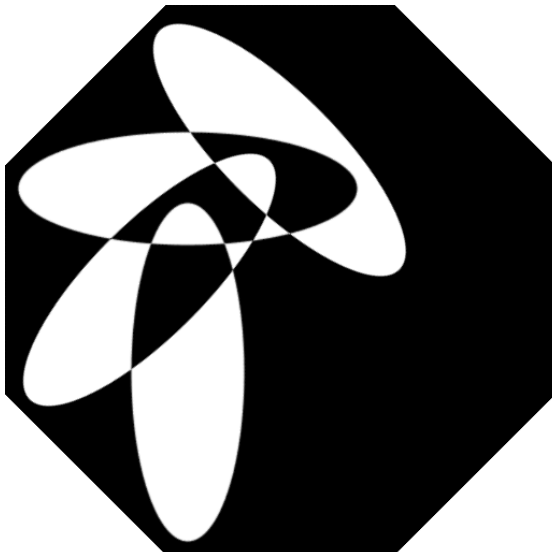


Figure 7.14: Using XOR to blend rotated layers, the intersections are always inverted even upon multiple pastes. This creates an interesting alternating pattern that has been used in mosaics through the centuries.

```
// Image NOTXOR Two Layers

PImage P1,P2;
int bg=0x00;   // Background colour
int fg=0xFF;   // Foreground colour
int len1,len2;

void setup() {
  size(512,512);
  noLoop();
  noStroke();
  len1=(8*width/13);
}

void draw() {
  // Draw the first ellipse
  background(bg);
  fill(fg);
  ellipse(width/3,height/2,len1,len1);
  P1=get(); // store first ellipse in P1

  // Draw the second ellipse
  background(bg);
  fill(fg);
  ellipse(2*width/3,height/2,len1,len1);
  P2=get(); // store second ellipse in P2

  // Combine the two images
  imageNOTXOR(P1,P2); // P1 = ~(P1 ^ P2)

  // And finally draw the result on the display window
  image(P1,0,0);
}

// imageNOTXOR method
// The method imageNOTXOR modifies the left image (first argument)
// by performing NOT-XOR on each pixel with a pixel in the right
// image (second argument)
void imageNOTXOR(PImage I1, PImage I2){
  for (int i=0; i<min(I1.width,I2.width); i++)
    for(int j=0; j<min(I1.height,I2.height); j++)
      I1.set(i,j, ~(I1.get(i,j) ^ I2.get(i,j)) );
}
```

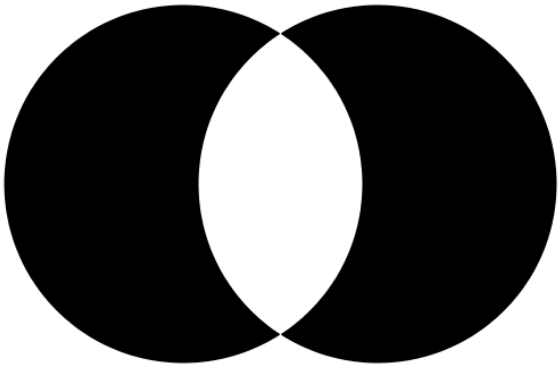


Figure 7.15: Using the bitwise unary NOT operator (˜) to invert an image.

7.5 Repetition

A powerful method of composing a design is to employ repetition of unit forms. One way of repeating a form is to use a `for()` loop to iterate a position variable.

Recall, from Java, that the syntax of a `for()` loop is:

```
for(initialization; termination; increment){
    statement(s);
}
```

The initialization expression is executed once when the loop begins.
The boolean termination expression will halt the loop if it evaluates to `false`.
The increment expression is evaluated at the end of each loop iteration.

7.5.1 Rows

In Figure 7.16, a `for` loop is used to iterate in the x-dimension while leaving the y-dimension fixed. The number of steps to iterate is set to 10, and the amount of each jump along the x-dimension is set to `width/10`. This results in 10 equal jumps across the width of the screen.

```
size(512,512);
background(0);
fill(255);
for(int i=0; i<10; i++)
  rect(i*width/10,height/2,20,20);
```

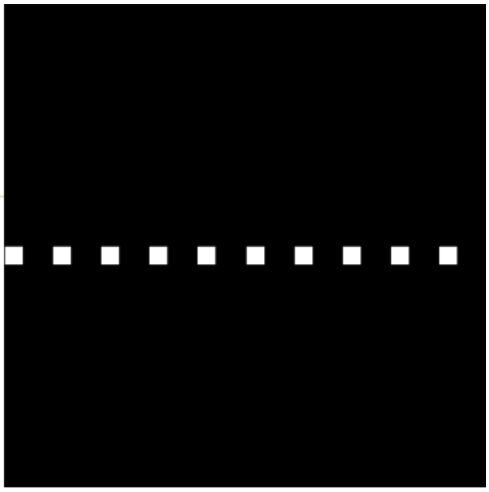


Figure 7.16: Repeating a Unit Form along a row.

7.5.2 Columns

We can apply the same technique as above to repeat along the y-dimension by transposing the position variable and the fixed position. Compare Figure 7.17 on the next page, with the rows example, to see what has changed.

```
size(512,512);
background(0);
fill(255);
for(int i=0; i<20; i++)
  rect(width/1.61,i*height/20,10,10);
```

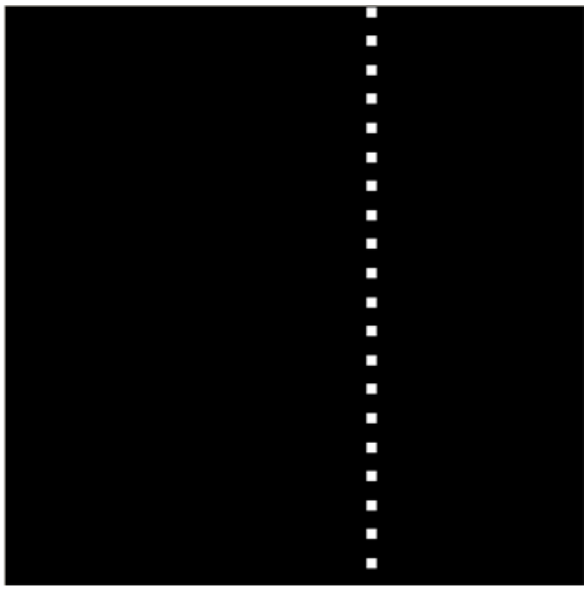


Figure 7.17: Repeating a Unit Form along a column.

7.5.3 Diagonals

Using a single `for` loop, a repetition structure that spans the x-dimension and the y-dimension can be generated. In Figure 7.18, the iteration variable, `i`, is used both for both `x` and `y` coordinates. However, we have used a different function of the variable for each coordinate. Notice the use of the modulus operator `%` in Figure 7.18, to wrap the x-dimension coordinate onto the screen width.

```
size(512,512);
background(0);
fill(255);
for(int i=0; i<20; i++)
  rect((i*width/5)%width,i*height/20,10,10);
```

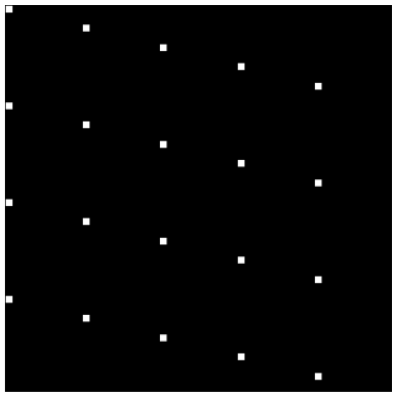


Figure 7.18: Repeating a Unit Form along diagonals.

7.6 Recursion

Figure 7.19 shows a fractal tree. A fractal is a form that has self similarity; in this case, each branch of the tree is a smaller version of the previous branch to which it is

```
// Fractal tree using recursion

int Branching=7; // number of times to repeat

void setup() {
  size(512,512);
  background(0);
  stroke(255);
  strokeWeight(4);
  noFill();
}

void draw() {
  // move to a suitable starting point
  translate(width/2.,7*height/8.);
  // and draw the tree
  tree(height/4,PI/5,1.41,Branching);
}

// make a tree by recursion
//  sz is the length of the first branch
//  a  is the (half) angle at each branching point
//  sf is the scaling factor for each successive branch
//  n  is the number of recursive branches to draw
void tree(float sz, float a, float sf, int n) {
  if(n==0)
    return;           // terminate if n is zero
  line(0,0,0,-sz);    // draw a line
  if(n==1)            // if end of tree draw a leaf
    ellipse(0,-sz,16,8);
  --n;                // decrement branching
  pushMatrix();       // save coordinate system
  translate(0,-sz);   // move to end of branch
  rotate(a);          // rotate by angle
  tree(sz/sf,a,sf,n); // draw a tree (recursive call)
  rotate(-2*a);       // rotate other way
  tree(sz/sf,a,sf,n); // draw a tree (recursive call)
  popMatrix();        // restore coordinates
}
```

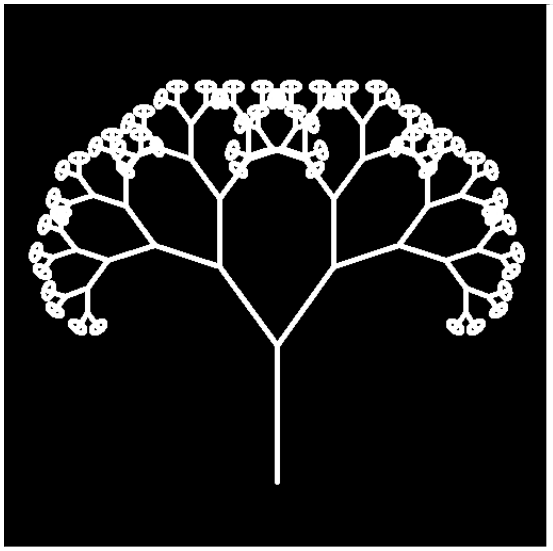


Figure 7.19: A fractal tree drawn using recursion.

connected. Fractal forms are relatively common in nature and can be used to describe many natural phenomena such as mountains, coast lines and clouds. The method of structuring we have used for the fractal tree involves a different type of iteration than a `for()` loop. In this case each line of a tree is drawn by the `tree( )` method which then calls itself to draw the next branch of the tree. Thus each branch is just a copy of the last branch but with a different starting point, length and angle of rotation.

This process of performing a simple operation and calling upon the operation again within the operation is called recursion and is a very powerful form of expression in computer science. This example shows how a composition can be organised recursively so that the different levels of the composition relate to each other by hierarchy.

Finally, Figure 7.20 shows the same sequence of operations with different parameters and using rectangles instead of lines and ellipses to draw the Unit Forms. Note that in both of these examples, the relationships of forms to each other is by hierarchical composition using recursion. The visual perception of such structures includes the concept of nesting, that is, forms inside or subordinate to each other.

We will return to the topics of recursion and fractals in Chapter 6 of Volume 2 of this subject guide, where we will see that recursive drawing rules can be used to generate a wide variety of forms beyond the tree structures we have seen here.

```
// Fractal tree using recursion and rectangles

int Branching=7; // number of times to repeat

void setup() {
  size(512,512);
  background(0);
  stroke(255);
  strokeWeight(2);
  rectMode(CENTER);
  noFill();
}

void draw() {
  // move to a suitable starting point
  translate(width/2.0,7*height/8.0);
  // and draw the tree
  tree(height/4,PI/4,1.41,Branching);
}

// make a tree by recursion
//  sz determines the size of the first rectangle
//  a  is the (half) angle at each branching point
//  sf is the scaling factor for each successive branch
//  n  is the number of recursive branches to draw
void tree(float sz, float a, float sf, int n) {
  if(n==0)
    return;              // terminate if n is zero
  rect(0,-sz,sz/3,sz*2); // draw a rectangle
  --n;                   // decrement branching
  pushMatrix();          // save coordinate system
  translate(0,-sz);      // move to end of branch
  rotate(a);             // rotate by angle
  tree(sz/sf,a,sf,n);    // draw a tree (recursive call)
  rotate(-2*a);          // rotate other way
  tree(sz/sf,a,sf,n);    // draw a tree (recursive call)
  popMatrix();           // restore coordinates
}
```

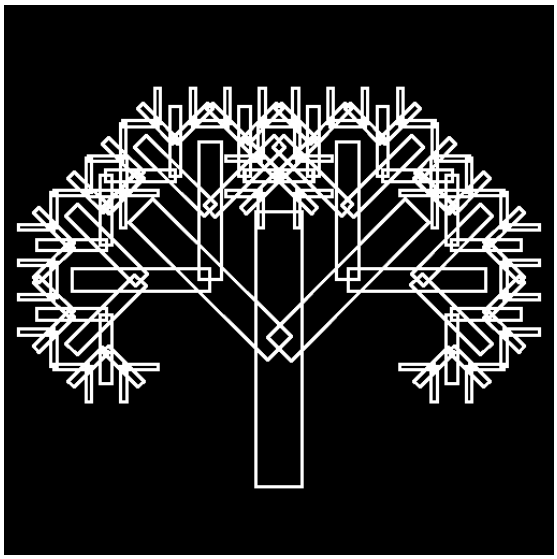


Figure 7.20: Another hierarchical composition using rectangles.

Learning activity

Modify the sketch in Figure 7.19 so that the leaves are triangles instead of ellipses.

What happens if you change the angle to `PI/2` in the `draw()` method's call to `tree()`? The angle is the second argument to `tree()`.

What happens if you change the angle to `PI`?

What happens if you change the angle to `TWO_PI/3`?

Change the branching factor to `Branching=10`. What happens?

Change the scale factor (`sf`), the third parameter to `tree()`, to 1.61, 2, and 1.0 respectively. What happens in each case?

Remove the second method call to `tree()` inside the `tree()` method. What happens? Why?

7.7 Summary and learning outcomes

In this chapter we have investigated how unit forms can be structured into larger forms, concepts and compositions using a number of principles of combination. We started by considering the Gestalt principles that describe our ability to see whole aspects of a set of forms rather than a collection of individual forms.

We then looked at how to achieve some of the effects of proximity and the relationships between forms by the way in which they overlapped or intersected. A new technique of blending image layers using bit-wise logical operators was introduced.

A number of iterative structuring methods were demonstrated that repeat a basic form in rows, columns or diagonals. We finally looked at the concept of hierarchy and how to implement it as a recursive procedure in *Processing*.

You should now be able to:

- discuss and distinguish between the Gestalt principles of perception
- use *Processing* to create images that demonstrate some of these principles
- discuss and distinguish between different proximity relationships
- discuss how the logical connectives `and`, `or`, `not` and `xor` relate to the proximity relationships, and how they can be used to produce *Processing* sketches
- use the concepts of repetition and recursion practically in *Processing*, and to create interesting images
- describe what a fractal is, and use *Processing* to create images that contain fractals.

The examples and methods in this chapter serve only as a guide to your own exploration of composing forms into larger concepts. You are strongly encouraged to take the examples in this chapter and modify them to create your own

compositions. You should try making larger structures out of the simple ideas presented here. You should also try combining the ideas in this chapter with ideas from previous chapters, such as putting together different shapes and structuring them in new ways.

7.8 Exercises

1. Use *Processing* to draw as many of the examples in Section 7.2 as you can. Which is the hardest to create using *Processing*? For this one, can you create a different image that still captures the Gestalt Principle that it exemplifies?
2. Figure 7.10 shows what is effectively the union of two sets, as does Figure 7.11. Yet Figure 7.10 has much shorter and simpler code than Figure 7.11. What is really the difference between these two approaches? In what circumstances would you choose one over the other? Can you draw the same picture as in Figure 7.12 without using the bitwise approach, but instead taking an approach more like that of Figure 7.10?
3. The key to recursion is the *terminating condition*. What is this? In Figures 7.19 and 7.20, which statements are the terminating conditions? What happens if there is no terminating condition?
4. Find out as much as you can about the use of recursion in making pictures. Find some examples of artists whose work uses this technique.
5. Find out what you can about the Sierpinsky triangle, and Sierpinsky carpets. Try to draw approximations of Sierpinsky coverings using *Processing*.

Chapter 8

Motion

Essential reading

Reas, C. and B. Fry *Processing: A Programming Handbook for Visual Designers and Artists* (MIT Press, 2007) [ISBN 0262182629]. Structure 2: Continuous evaluation; Controlling the flow, Structure 3: Function overloading, Control 1: Decisions, Input 1: Mouse data, Input 2: Keyboard, Synthesis 2: Input and Response, Motion 1: Lines, Curves, Transform 1: Controlling transformations, Simulate 2: Motion simulation, Math 4: Noise.

Additional reading

Hughes, J. F., A. van Dam, M. McGuire, D. Sklar, J. D. Foley, S. K. Feiner, and K. Akeley *Computer Graphics: Principles and Practice* (Addison-Wesley, 3rd edition, 2013) [ISBN 0321399528]. Chapter 10, Chapter 35

8.1 Introduction

So far you have been studying methods for drawing stationary images. In this chapter you will learn how to create sketches with objects that move, guided by functions that you will design. These processes are called *animations*.

There are four requirements for creating a simple animation. These are the `draw()` method, the `background()` method, global variables for persistence of position and, finally, the *component velocities*.

8.2 `setup()` and `draw()`

In the previous chapters we have been using the `draw()` method to create a still image on the display window. However, behind the scenes, *Processing* actually calls the `draw()` method repeatedly, a fixed number of times per second, until the sketch is stopped or the `noLoop()` method is called. We can create an animation by changing the output of the `draw()` method each time it is called.

When the `draw()` method is used we must also use the `setup()` method to define the screen size, background colour, stroke colour, fill colour and other useful parameters that will not change during the sketch. The `setup()` method is called exactly once at the beginning of the sketch, then the `draw()` method is repeatedly called.

These methods use the Java technique of *inheritance*. That is, they are already

defined within the *Processing* environment, but if we want to add our own functionality we must override these methods. All sketches that use the `setup()` or `draw()` methods must redefine them in the following way:

```
void setup() {
  size(mySizeX, mySizeY);
  // Your setup code
}

void draw() {
  // Your draw code here
}
```

The simple example in Figure 8.1 illustrates how these elements can be used to create a basic animation. (Note that example output shown in Figure 8.1 shows a trace to indicate the motion of the ball. The code shown in the figure does not produce such a trace on the display window; we will look at how this effect can be achieved in the Learning Activity on p. 92.)

```
// Simple animation of ball movement in x direction only

float x;        // Global variable to store x position of ball

void setup() {
 size(512,512);
 fill(255);
 frameRate(10); // frame rate, 10Hz
}

// draw called repeatedly at the frame rate, 10 Hz
void draw() {
 background(0); // redraw background to erase previous output
 ellipse(x,height/2,10,10);
 x=x+2;         // amount to change x position at each step
}
```

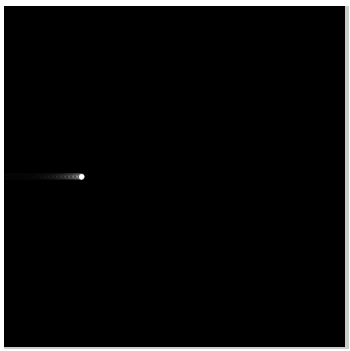


Figure 8.1: Basic animation: the ball appears to move in a straight line.

8.3 Persistence

The apparent motion of an object or scene on a screen is a trick of the eye. The process of animating an object is to draw it, erase it, draw it in a slightly different position, and so on. This sequence of operations creates the illusion of movement of an object on the screen. Clearly, nothing is really moving. The pixels remain in their permanent locations.

There are two unrelated concepts of persistence at play in this example. The first is the *persistence of vision* which is a property of our eye-brain system that makes a sequence of small jumps such as erase, move, re-draw, appear as a smooth motion rather than a sequence of jumps. This property of the eye-brain system only works for relatively small jumps, if we make the jumps too large then the motion would appear jerky.

The second type of persistence that we employ is a memory of the location of the object from the previous step. The `draw()` method must remember the new location

of the ball to draw it in the correct position. This is persistence of a variable and it is achieved using a global variable `float x` which is defined at the top of the sketch before the `setup()`. In general, global variables are defined anywhere outside of the methods defined in your sketch. See Figure 8.1.

We will explore persistence of vision in greater detail in Chapter 1 of Volume 2 of the *Creative Computing II* subject guide.

8.4 Velocity

The speed and direction of the velocity are controlled by the number of pixels that we shift the object by each time it is re-drawn. Shifting by two pixels each step results in motion that is twice as fast as one pixel each step. Using two global variables, one each for the x-dimension and y-dimension, we can create motion in two dimensions. Figure 8.2 shows the resulting motion of moving a ball with component velocities `x=x+2` and `y=y+1`.

```
// Simple animation of ball movement in x and y directions

float x,y;        // Global variables to store x and y positions

void setup() {
 size(512,512);
 fill(255);
 frameRate(10); // frame rate, 10Hz
}

// draw called repeatedly at the frame rate, 10 Hz
void draw() {
 background(0); // redraw background to erase previous output
 ellipse(x,y,10,10);
 x=x+2;         // amount to change x position at each step
 y=y+1;         // amount to change y position at each step
}
```

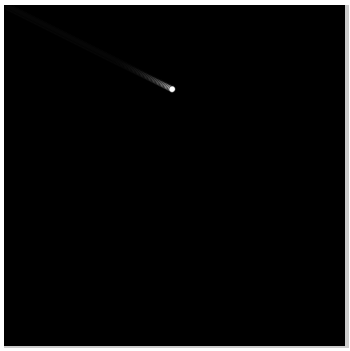


Figure 8.2: Using component velocities in both x and y makes diagonal motion.

8.5 Motion by Coordinate Transformations

Recall in Chapter 6 we used *Processing*'s built-in transformation methods `translate()` and `rotate()` to draw shapes using the 'turtle graphics' concepts of Seymour Papert's Logo language. The sketch in Figure 8.3 uses the `translate(x,y)` method to change the position of the origin of the coordinate system and then draws the ball at the origin.

In addition to the global variables for remembering the position of the coordinate system, the component velocities are also defined using two global variables `float dx=1,dy=2;`. Compared to Figure 8.2, this example shows that setting the jumps in x and y to different values changes the direction of travel. If the size of the jumps is the same each time, we call the `draw()` method, then the motion will always be in a straight line. Eventually, depending on the size of the jumps, the ball reaches the edge of the screen and moves beyond it. The ball continues to be drawn but it is drawn off the screen so we do not see it.

```
// Motion by coordinate transformation

float x=256, y=256; // position of ball
float dx=1, dy=2;   // change in x and y position at each step

void setup() {
 size(512,512);
 fill(255);
 frameRate(10); // frame rate, 10Hz
}

// draw called repeatedly at the frame rate, 10 Hz
void draw() {
 background(0);     // redraw background to erase previous output
 translate(x,y);    // translate the coordinate system
 ellipse(0,0,10,10);// draw at 0,0 in new coordinate system
 x+=dx;             // amount to change x position at each step
 y+=dy;             // amount to change y position at each step
}
```

Figure 8.3: Setting the component velocities to different values changes the angle (direction) of motion.

8.6 Reflection at Boundaries

To make the ball reverse direction when it reaches one of the edges of the screen we use a conditional; in Figure 8.4, two `if(...)` statements test if the ball is at, or past, the edge of the screen and reverse the component velocity direction if it is. If the ball is at, or past, the left or right edge of the screen, the x component velocity `dx` is reversed by negating its value `dx=-dx`. If the ball is at, or past, the top or bottom of the screen, then the y component velocity is reversed by negating its value in the same way. This logical test and the reversing of component velocities ensures that the ball always stays within the boundary of the screen.

```
// Motion with relfection at boundaries

float x=256, y=256; // position of ball
float dx=11, dy=13; // change in x and y position at each step
int sz=10;          // size of ball

void setup() {
 size(512,512);
 fill(255);
 frameRate(10);     // frame rate, 10Hz
}

// draw called repeatedly at the frame rate, 10 Hz
void draw() {
 background(0);     // redraw background to erase previous output
 translate(x,y);    // translate the coordinate system
 ellipse(0,0,sz,sz);// draw at 0,0 in new coordinate system
 if (x<=sz || x>=width-dx)
   dx=-dx;          // reflect off left and right screen edge
 if (y<=sz || y>=height-dy)
   dy=-dy;          // reflect off top and bottom screen edge
 x+=dx;             // amount to change x position at each step
 y+=dy;             // amount to change y position at each step
}
```

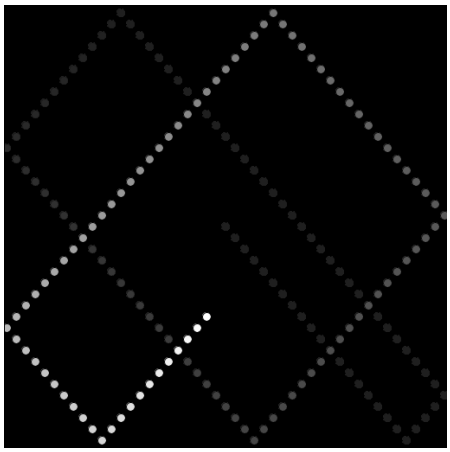


Figure 8.4: Detecting when the ball is at, or past, one of the edges of the screen and reversing the component velocity.

8.7 Interaction

```
// Simple collision detection

float x=256, y=256; // position of ball
float dx=11, dy=13; // change in x and y position at each step
int sz=10;          // size of ball

void setup() {
 size(512,512);
 fill(255);
 noStroke();
 frameRate(10);     // frame rate, 10Hz
}

// draw called repeatedly at the frame rate, 10 Hz
void draw() {
 background(0);     // redraw background to erase previous output
 rect(0.6*width, mouseY-50, 20, 100); // draw bat
 translate(x,y);    // translate the coordinate system
 ellipse(0,0,sz,sz);// draw at 0,0 in new coordinate system
 if (x<=sz || x>=width-dx)
   dx=-dx;          // reflect off left and right screen edge
 if (x>=0.6*width && x<=0.6*width+20 && y<=mouseY+50 && y>=mouseY-50)
   dx=-dx;          // reflect off bat
 if (y<=sz || y>=height-dy)
   dy=-dy;          // reflect off top and bottom screen edge
 x+=dx;             // amount to change x position at each step
 y+=dy;             // amount to change y position at each step
}
```

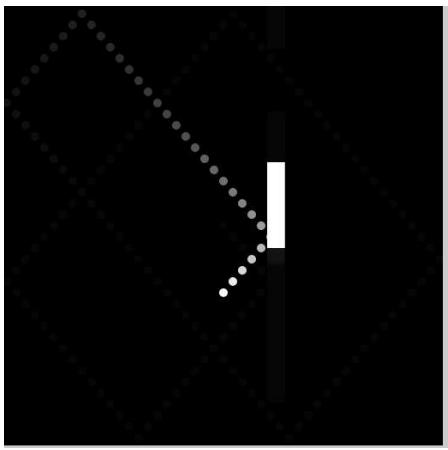


Figure 8.5: Detecting when the ball has hit a bat controlled by the user.

One of the first computer games was *Pong*, an arcade game made by the company Atari founded by Nolan Bushnell. In the game, the players control a bat which they must move to hit a ball. The winner is the player who hits the ball the most times before missing the ball three times. Figure 8.5 shows the basics of such a game. In this sketch, the user interacts with the sketch by moving the mouse which controls the height of the bat. When the ball hits the bat, the ball's velocity in the x direction is reflected.

This is achieved by starting with Figure 8.4 and adding two extra lines. One line draws the bat, `rect(0.6*width,mouseY-50,20,100);` which is drawn at 0.6 the width of the screen, with its centre placed at the mouse's Y position. Then to detect if the bat has hit the ball, we must check both the x-dimension boundaries of the bat and the y-dimension boundaries of the bat; this is done via the line `if(x>=0.6*width && x<=0.6*width+20 && y<=mouseY+50 && y>=mouseY-50) dx=-dx;`. If the expression evaluates to `true`, this reverses the x-dimension velocity component.

Learning activity

Modify the sketch in Figure 8.5 so that a score is kept of the number of times the player has hit the ball with the bat.

Use the `println()` method to print the current score each time the bat is hit.

Modify your game so that the game ends if the ball disappears off the right edge of the screen. All other edges should still reflect the ball.

8.8 Gravity and Acceleration

Rates of change of position are determined by the component velocities. Figure 8.6 illustrates what happens when the velocity itself is changing. In this case the y-component of motion is subject to gravitational acceleration. At each step, the y-component of the velocity is decreased by a small amount. This amount is constant and is called the acceleration.

The acceleration is in the downward direction which in *Processing* is positive in the y component velocity. The effect is to create a bouncing ball that moves on a curve called a parabola.

```
// Gravity and acceleration

float x=128, y=128;  // position of ball
float dx=5, dy=10;   // rate of change of position (velocity)
float gravity = 5;   // rate of change of velocity (acceleration)
float ballSize = 20; // size of ball

void setup() {
 size(512,512);
 fill(255);
 frameRate(20);      // frame rate, 20Hz
}

// draw called repeatedly at the frame rate, 10 Hz
void draw() {
  background(0);     // redraw background to erase previous output
  translate(x,y);    // translate the coordinate system
  ellipse(0,0,ballSize,ballSize);
  if (x>width-ballSize/2 || x<ballSize/2)
    dx=-dx;          // bounce off screen sides
  if (y>height-ballSize/2)
    dy=-dy;          // bounce off screen bottom edge
  else
    dy+=gravity;     // apply acceleration (change of velocity)
   x+=dx; // apply x-component velocity (change of x position)
   y+=dy; // apply y-component velocity (change of y position)
}
```

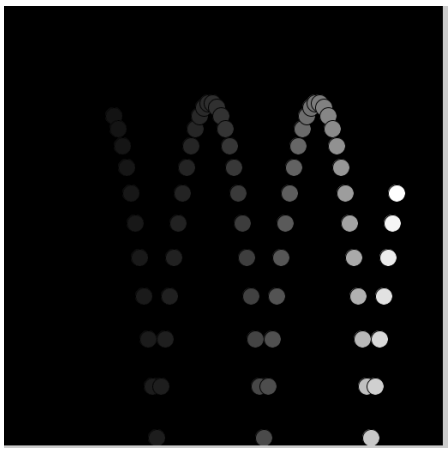


Figure 8.6: Motion of a projectile under gravity, with reflection (bouncing).

8.8.1 Rotation and Acceleration

Figure 8.7 illustrates acceleration in all directions. The spaceship can freely rotate around its centre point using the LEFT and RIGHT arrow keys. The UP arrow key applies thrust which is acceleration in the direction that the spaceship is pointing. By rotating the spaceship and applying thrust in different directions, very complex motions can be created. In this example the motion is contained within the screen by wrapping the spaceship around the screen edges. This example is based on the popular early computer game called *Asteroids*.

The novel aspect of the motion is the rotation of the spaceship around its centre using `translate(x+sz/2,y+sqrt(3)*sz/4)`. Here the `x,y` coordinates are the position of the bottom left vertex of the spaceship. The mid-point of the spaceship is calculated as the mid-point of a triangle. We translate to that point, rotate the

```
// Rotation and acceleration

float x=128, y=128; // initial position
float dx=0, dy=0;   // initial velocity
float angle=0;      // initial rotation
float sz=61;        // size of spaceship

void setup() {
  size(512,512);
  noFill();
  stroke(255);
  frameRate(20);
}

// draw called repeatedly at the frame rate
void draw() {
  background(0);

  // deal with keyboard interaction
  if (keyPressed && keyCode==LEFT)       // LEFT ARROW ROTATE LEFT
    angle=(angle-PI/16)%TWO_PI;
  else if (keyPressed && keyCode==RIGHT) // RIGHT ARROW ROTATE RIGHT
    angle=(angle+PI/16)%TWO_PI;
  if (keyPressed && keyCode==UP) {       // UP ARROW THRUST (ACCELERATE)
    dx+=-sin(angle);
    dy+=cos(angle);
  }

  // rotate coordinate frame around centre of spaceship
  translate(x+sz/2,y+sqrt(3)*sz/4);
  rotate(angle);
  translate(-sz/2,-sqrt(3)*sz/2);

  // draw the spaceship
  beginShape();
  vertex(0,0);
  vertex(sz/2,sqrt(3)*sz);
  vertex(sz,0);
  vertex(sz/2,sqrt(3)*sz*.37);
  endShape(CLOSE);

  // update spaceship's position
  x=(x+dx)%width;      // apply x-component velocity, wrap at right edge
  y=(y+dy)%height;     // apply y-component velocuty, wrap at bottom edge
  if (x<-sz) x=width;  // wrap at left edge
  if (y<-sz) y=height; // wrap at top edge
}
```

Figure 8.7: Motion of a space ship with thrust (acceleration) and rotation (direction).

coordinate system, then translate back to the bottom left vertex to start drawing. The effect of the translation, rotation, and translation are to rotate the spaceship around its centre.

Learning activity

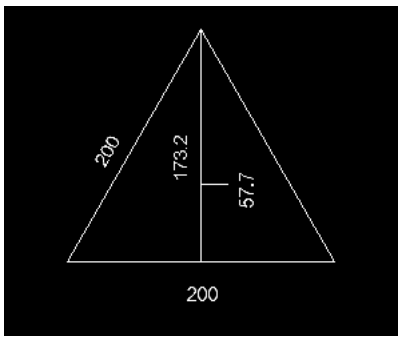


Figure 8.8: Construction of an equilateral triangle with edge length 200. The height of the triangle above its baseline is `sqrt(3)*200/2` and the mid-point is at one third of its height on the centre line.

What happens if you replace the lines:

```
translate(x+sz/2,y+sqrt(3)*sz/4);
rotate(angle);
translate(-sz/2,-sqrt(3)*sz/2);
```

in Figure 8.7 with the following code?

```
translate(x,y);
rotate(angle);
```

Modify the code to make the spaceship an equilateral triangle with edge length 200 and rotate it about its centre. The necessary measurements are shown in Figure 8.8.

For more on the geometric construction of triangles see :
`mathworld.wolfram.com/EquilateralTriangle.html`
`en.wikipedia.org/wiki/Triangle`

Replace the line in the `draw()` method that calls `background(0)` with the following code:

```
fill(0,20);
rect(0,0,width,height);
```

What effect does this have? The second argument to `fill()` is a transparency value. Why do you see the effect that you see?

Modify all of the examples in this chapter to create the same effect. You may need to add a call to `background(0)` in the `setup()` method to ensure that the background is black at the start of the animation. You may also need to reset the fill colour to its normal value (e.g. `fill(255)`) after you have added the new lines shown above to the `draw()` method.

8.9 Random Motion

8.9.1 Brownian Motion

```
// Brownian motion

float x=256, y=256;  // initial position
float sz=20;         // size of ball
float range=10;

void setup() {
  size(512,512);
  background(0);
  noFill();
  stroke(255);
  frameRate(30);
}

void draw() {
  // fading trace of previous movement
  fill(0,10);
  rect(0,0,width,height);
  noFill();

  // draw ball in current position
  translate(x,y);
  ellipse(0,0,sz,sz);

  // update ball's x and y position with uniform random numbers
  x+=random(-range,range);
  y+=random(-range,range);
}
```

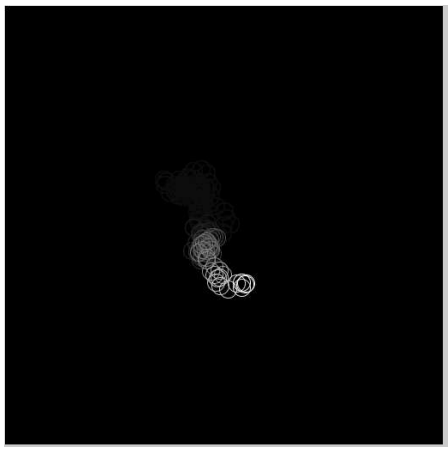


Figure 8.9: An approximation to Brownian motion consisting of random jumps in x and y, which can be positive or negative. This corresponds to having random component velocities at each step.

Sometimes we want motion that is not deterministic; i.e. it does not follow a predictable path. We can achieve this by incorporating randomness into our sketches. Figure 8.9 illustrates the type of motion that occurs when the jumps in position are random at each step. In this case, the random value is drawn from the range $[-10, 10]$; this means that values can be both negative and positive. The `random()` method returns real-valued numbers `(float)` so the jumps are whole pixels plus fractional parts. The trace effect is created by using the alpha channel (transparency) as in the previous learning activity.

True Brownian motion—originally observed with smoke particles under a microscope—is produced using a random angle of rotation and random jump-length (radius) for each step. Thus it is a circular non-uniform distribution of velocities. To do this in *Processing* we must first calculate a random angle and radius, then convert these circular values into rectangular coordinates. The relationships between an angle, a, a radius, r, and x and y coordinates are given by basic trigonometry:

$$x = r * cos(a), y = r * sin(a) \tag{8.1}$$

The circular coordinates, (a, r), are called *polar coordinates* in mathematics, and the standard (x, y) coordinates are known as *rectangular coordinates*.

Learning activity

Modify the approximate Brownian motion sketch in Figure 8.9 so that it uses a true circular random distribution.
Hint: at each step you should make a random angle, `a`, in the range `float a=random(TWO_PI)` and a random radius in the range `float r=random(10)`.
Now use the trigonometric relations above, using the `cos()` and `sin()` methods in *Processing* to convert the circular values `a` and `r` to rectangular coordinates `x` and `y`.

8.9.2 Perlin Noise

```
// Perlin noise

float xoff = 0.0;  // value to be passed to noise() method
int   sz = 20;     // size of ball

void setup() {
  size(512,512);
  frameRate(30);
  background(0);
  stroke(255);
}

void draw() {
  // Make a trace using alpha channel
  fill(0,20);
  rect(0,0,width,height);

  // Get a noise value based on xoff and scale
  float x = noise(xoff)*width;
  float y = noise(xoff+1.41)*height;
  translate(x,y);
  ellipse(0,0,sz,sz);

  // With each cycle, increment xoff
  // Refer to the Processing reference manual for further
  // details about the noise() method and xoff parameter
  xoff += 0.01;
}
```

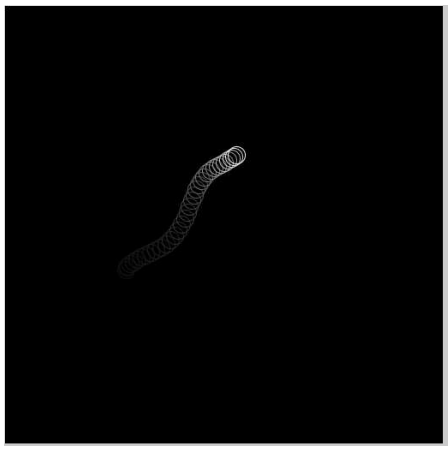


Figure 8.10: Perlin noise generates motion that looks more life-like than Brownian motion. It is still randomly generated, but using a special `noise()` method that generates the Perlin noise.

Sometimes we want to generate motion that is indeterminate, but that has a more life-like quality than the motion that is generated by Brownian motion. Ken Perlin invented a technique to generate noise that is more natural in appearance than that of the `random()` method. The method is called `noise()` in *Processing* and takes one argument that is an offset value. This value simply determines where in a pre-computed function to look up the noise value. Small changes to the offset parameter between successive calls to `noise()` will result in small jumps in the returned noise values, and larger changes will results in larger jumps. The returned value will always lie in the range of 0.0 to 1.0. Using this method we can create a smooth random motion. Figure 8.10 illustrates Perlin noise used in this manner.

Notice in this example that the `x` and `y` positions are not persistent. Instead, the `xoff` noise table look-up parameter is persistent and the `x` and `y` positions are generated independently at each step by the `noise()` method. The change in the `xoff` parameter at each step determines the average velocity of the motion and the difference between the parameter in the two calls to `noise()` generates the difference between the `x` and `y` velocity components.

Learning activity

Modify the `range` parameter in Figure 8.9 to the values 1, 20 and 100 respectively.

What happens if you replace `random(-range,range)` with `random(range)` in Figure 8.9?

In Figure 8.10, modify the second call to `noise(xoff+1.41)`. For example, try `float y=noise(xoff+4)`, `float y=noise(xoff+20)`, `float y=noise(xoff+100)` and `float y=noise(xoff)`.

What do you observe in each case?

Now modify the line `xoff+=0.01` to increment by a different amount. For example, `xoff+=0.001`, `xoff+=0.1`, `xoff+=10`.

What do you observe in each case?

Write down an explanation of the behaviour of the `noise()` method in your own words. Do a web search for "Perlin Noise". Try to understand what is happening behind the `noise()` method.

8.10 Motion of Multiple Objects

Multiple objects can be set in motion on individual paths by creating arrays of positions and component velocities. The method for creating the motion is exactly the same as for a single object, we just have to include the code within a `for(...)` loop so that each object in the array can be moved in turn. When using coordinate transformations to perform the motion, we must call the `pushMatrix()` method before the transformations, and call the `popMatrix()` method after the transformations, to ensure that all objects are drawn in their correct absolute position. If we do not do this, then each object gets a position that is relative to the last object, and many of them will be drawn off the screen.

Figure 8.11 shows a sketch that draws 10 circles in motion, each with individual positions, component velocities and sizes. Compare this example with Figure 8.2, to see how the same technique of motion is used but contained within a loop structure. If you are not familiar with Java arrays, then now is a good time to look them up in your Java textbook.

Learning activity

Modify the number of objects drawn in Figure 8.11 to 100 and 1000 respectively. Do you notice a difference in the speed of drawing?

```
// Motion of multiple objects

int     nObj=10; // number of objects
float[] x=new float[nObj], y= new float[nObj];
float[] xVel=new float[nObj], yVel=new float[nObj];
float[] sz = new float[nObj];

void setup() {
  size(512,512);
  background(0);
  noFill();
  stroke(255);
  // initialise data for each object
  for(int i=0; i<nObj; i++){
    x[i]=random(width);     // random x position
    y[i]=random(height);    // random y position
    xVel[i]=random(-10,10); // random x component velocity
    yVel[i]=random(-10,10); // random y component velocity
    sz[i]=random(40)+5;     // random size of object
  }
}

void draw() {
  // make a trace using alpha channel
  fill(0,20);
  rect(0,0,width,height);

  // Draw each object and update its position
  for(int i=0; i<nObj; i++){
    // draw object i
    pushMatrix();       // Save current coordinate system
    translate(x[i],y[i]);
    ellipse(0,0,sz[i],sz[i]);
    popMatrix();        // Restore coordinates

    // handle reflections at window boundaries
    if(x[i]>width-abs(xVel[i]) || x[i]<abs(xVel[i]))
      xVel[i]=-xVel[i]; // Reflect off left and right
    if(y[i]>height-abs(yVel[i]) || y[i]<abs(yVel[i]))
      yVel[i]=-yVel[i]; // Reflect off top and bottom

    // update position of object i
    x[i]+=xVel[i];      // Apply x component velocity
    y[i]+=yVel[i];      // Apply y component velocity
  }
}
```

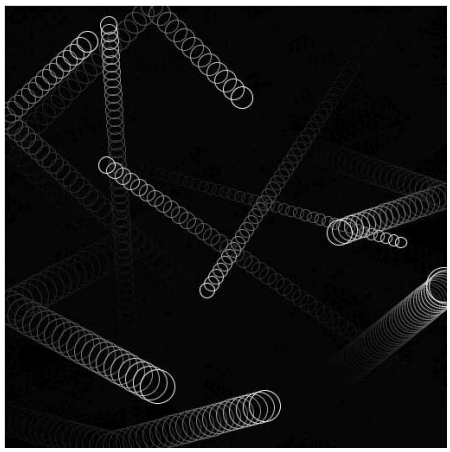


Figure 8.11: Motion of multiple objects with individual component velocities and sizes.

Modify the sketch of Figure 8.5 to draw multiple balls to be hit by the user's single bat. Start the balls in different initial positions and with different component velocities as shown in Figure 8.11.

Extend the sketch of Figure 8.10 to draw multiple different objects using Perlin noise motion. In this example, each object will have a random offset `xoff[i]` and a random increment `xoff[i]+=xInc[i]`.

Extend the sketch of Figure 8.7 to play a simple version of the game *Asteroids*. You will need multiple asteroids moving in straight line motion, each with its own random component velocities, and a number of rockets that can be fired (using the SPACE bar) to destroy the asteroids. The ship should be destroyed if it hits one of the asteroids. The rockets, spaceship and asteroids should all wrap around the screen edges as in Figure 8.7.

8.11 Summary and learning outcomes

The appearance of motion in *Processing* can be created by repeatedly erasing, moving and redrawing a set of objects. The type of motion can either be in a straight line (Figure 8.1, Figure 8.2 and Figure 8.3) or a more complex motion based on curved paths (Figure 8.6 and Figure 8.7) or a motion based on different types of randomness (Figure 8.9 and Figure 8.10).

In applying motion to our sketches, we can give the impression of movement that we might see in nature such as bouncing (gravity) or life-like motion (Perlin noise). When we make a creative sketch it is useful to base our programming on observations of the real world, so that we learn to choose the correct type of motion to model the situation we want to create.

You should now be able to:

- explain how animation and the appearance of movement are achieved, in general and using *Processing*
- write *Processing* sketches that depict objects moving at different speeds and through different angles, as well as incorporate bouncing
- incorporate acceleration into motion
- explain the difference between Brownian motion and Perlin noise, and decide which is appropriate to use in which circumstances
- write *Processing* sketches that include multiple moving objects.

You should look carefully at different types of motion now that you know the basics of how to imitate them, and see if you can refine your sketches to create more realistic, lifelike and compelling creative works.

8.12 Exercises

1. Take the sketch in Figure 8.6 and modify it such that the height of the bounce lessens as the bounce progresses as a fixed fraction of the speed is lost at each bounce.

2. The `noise()` method in *Processing* can take 1, 2 or 3 parameters. Try to modify the examples in this chapter where `noise()` is used, to establish how it works with the different numbers of parameters. You should also look at the *Processing* reference to understand what is being done. You will see more on Perlin noise in later parts of this course.
3. In Figure 8.11 you saw a sketch that demonstrates the motion of multiple objects. Develop a sketch that contains only two objects, that move randomly, but change the direction of their motion when they collide.

Chapter 9

Cellular Automata in 1D and 2D

Essential reading

Reas, C. and B. Fry *Processing: A Programming Handbook for Visual Designers and Artists* (MIT Press, 2007) [ISBN 0262182629]. Simulate 1: Biology; Cellular automata, Image 3: Pixels, Synthesis 3: Motion and Arrays, Appendix D: Bit, Binary, Hex.

9.1 Introduction

When we make a sketch, we usually start with a clear picture in our minds of what we are going to draw. Whether it is a pattern based on grids, symmetry or motion, we are responsible for placing all of the elements in the scene.

In Section 7.6 we took an introductory look at the topics of recursion and fractals. These methods allow us to create complex patterns by the repeated application of simple rules. In this chapter we will explore other techniques that similarly produce complex patterns and animations based upon simple rules. We will draw using primitive operations that interact to create complex patterns. To do this we must first consider how to represent primitive elements as code in our rule-based drawing system.

9.2 Bits and Pixels

In this chapter, we will draw by setting the individual pixels on the screen. Although it is a slow way to do things, compared to *Processing*'s built-in shape drawing methods, setting individual pixels allows us complete control over the screen. Figure 9.1 shows how to switch all the individual elements on the screen using a bit-wise operation.

A global variable, `y`, holds the height of the row to be drawn. Then the `setup()` method draws the sketch window and makes the background black. The `draw()` method uses a `for` loop to visit every pixel in each row of the sketch. Each pixel is accessed using the `set(x,y,c)` method which sets the pixel at screen position `(x,y)` to the value `c`.

In Section 7.4.1 we looked at the representation of colours in *Processing*. The colour value of each pixel is represented by a 32-bit integer. Black is all bits set to zero and white is all bits set to one. We can quickly set all the bits to one using the unary bitwise NOT operator, which flips all the bits in a binary representation of a number. For the number 0, the bits in a 32-bit integer are 00000000000000000000000000000000

```
// Switching individual pixels to white, one at a time
// (this is very slow and is for demonstration purposes only!)

int y=0; // record current y position

void setup() {
  size(512,512);
  background(0);
}

void draw() {
  // switch all pixels on current row to white
  for (int x=0; x<width; x++)
    set(x,y,~0);

  // move to next row, and stop when we reach the
  // bottom of the window
  if (++y>height)
    noLoop();
}
```

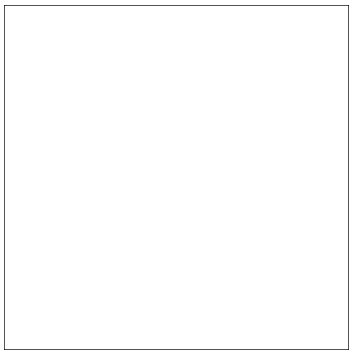


Figure 9.1: Each individual pixel in the sketch is switched from black to white using the `set()` method.

and the value ~0 makes the bits in the integer 11111111111111111111111111111111. So the method call `set(x,y,~0)` sets all bits to one which is white.

The 32-bit integer representing colour is split into four bytes; 11111111 11111111 11111111 11111111. These bytes represent the alpha, red, green and blue colour channels respectively: AAAAAAAA RRRRRRRR GGGGGGGG BBBBBBBB. When the R, G and B values are the same we see a shade of gray. If they are all set to 1s then we see white. In this chapter we will only consider black and white sketches; in the next volume we will learn how to use the colour channels to make colour sketches.

Note: The alpha value is the degree of opacity (opposite of transparency). When it has the value 255 (11111111), the pixel is completely opaque, while if given the value 0, it is completely transparent and thus the colour values of the pixel retain their current values and are not modified by new values. We must be careful to exclude the alpha channel from any colour value arithmetic that is intended for the RGB channels. We do this by bit-masking which is explained below.

9.3 Images out of Bits

We can use the bits in each pixel to compare colour values between adjacent pixels and set the next row's colour values based on simple adjacency rules. Figure 9.2 uses the Exclusive-Or operator on three adjacent pixels in the current row `(y)` to set a pixel value in the next row `(y+1)`.

The first row has its middle pixel set to white in the `setup()` method: `set(width/2,0,~0)`.

In the `draw()` method, the rule for setting each pixel in the next row is the bitwise Exclusive-Or operator (XOR) applied to the RGB bits only. Using the bitwise AND (&) operator between a colour value and the alpha mask returns an integer with only the colour bits represented. The alpha mask is created by right-shifting the all-ones bits by 8 bits: `int c=~0>>8`, which is the binary value

```
// Complex patterns from simple rules

int y=0;     // current row
int c=~0>>8; // alpha mask (00000000111111111111111111111111)

void setup() {
  size(256,256);
  background(0);
  set(width/2,0,~0); // set middle pixel in top row to white
}

void draw() {
  // Set pixels in next row according to pixels in current row
  // (see Section 9.3 for details)
  for(int x=1 ; x<width-1 ; x++)
    set(x, y+1, ((get(x-1,y)&c) ^ (get(x,y)&c) ^ (get(x+1,y)&c)) | ~c);
  // Move to next row, and stop when we reach the bottom
  if (++y>height)
    noLoop();
}
```

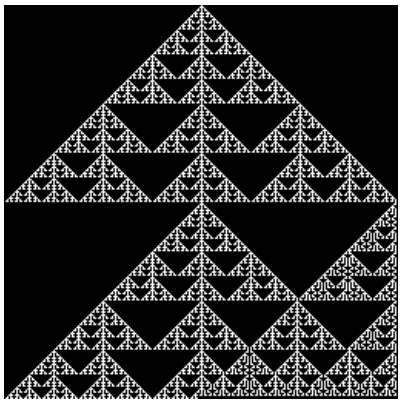


Figure 9.2: The bitwise Exclusive-OR (XOR) operator is applied to the colour values of each row to get the pattern for the next row.

00000000111111111111111111111111. Compare this with the ARGB bit positions above, to see why this selects colour values only and ignores the alpha value.

The `set(x,y+1,c)` method is called using the `get(x,y)` method as an argument, instead of a variable `c`, and the returned colour bits are XOR'd together using the bitwise XOR operator `^`. This is done for three adjacent pixels starting at `x-1` in the current row, `y`. Finally the alpha value is re-applied (because we need it to display the pixel) by OR'ing it with the inverted alpha mask:

```
... | ~c
```

The entire line that does the drawing is:

```
set(x, y+1, ((get(x-1,y)&c) ^ (get(x,y)&c) ^ (get(x+1,y)&c)) | ~c)
```

The remarkable pattern that is generated from these simple operations on the pixels is a fractal. It is a repeating pattern that exhibits self-similarity at all scales. There are many such patterns that can be made out of rule-based drawing. We now go on to explore one particular class of them, three-bit Cellular Automata.

9.4 Three-bit 1D Cellular Automata

The number of possible patterns of on's (white) and off's (black) for three adjacent pixels is $2^3 = 8$. A cellular automaton (CA) *rule* defines the centre pixel in the next row given the three adjacent pixels in the current row.

current three pixel pattern	111	110	101	100	011	010	001	000
new state for centre pixel	1	0	0	0	0	0	0	1

Table 9.1: Rule 129 Cellular Automaton Table

Table 9.1 shows the cellular automaton table for Rule 129. The possible current three state patterns are shown on the top row and their corresponding outputs (for the pixel in the middle of the three cells on the next row) is given on the bottom row. It

current three pixel pattern	111	110	101	100	011	010	001	000
new state for centre pixel	0	0	0	1	1	1	1	0

Table 9.2: Rule 30 Cellular Automaton Table

is called Rule 129 because the pattern of bits for the output (bottom) row is the binary for the decimal number 129 (this naming convention was introduced by the British scientist Stephen Wolfram). Table 9.2 is the table for Rule 30. Figure 9.3 is a completely general three pixel cellular automata generator. By changing the line that reads `int rule=129` to assign any value between 0 . . . 255, we can explore the entire set of possible three pixel cellular automata.

```
// Simple 3 bit, 1 dimensional CA, with direct pixel setting

int y=0;      // current row
int rule=129; // rule number (using Wolfram numbering scheme)

void setup() {
  size(256,256);
  background(0);
  set(width/2,0,~0); // set middle pixel in top row to white
}

void draw() {
  // Set pixels in next row according to pixels in current row
  for(int x=1 ; x<width-1; x++) {
    // inspect neighbouring bits in current row
    int pix1 = (get(x-1,y)&1)<<2; // gives 100 if x-1 on, else 000
    int pix2 = (get(x,y)&1)<<1;   // gives 010 if x   on, else 000
    int pix3 = (get(x+1,y)&1);    // gives 001 if x+1 on, else 000

    // combine results into a single 3-bit number
    // (for example, if x-1 is on, x is off and x+1 is on,
    // this will result in 101)
    int bits = pix1+pix2+pix3;

    // if the rule has a 1 in the position specified by
    // bits, set the pixel in the next row to on.
    // (Look at the examples of Rule number in Figures 9.1
    // and 9.2 to understand how the following line works)
    if ((rule&(1<<(bits)))>0)     // Test pattern in rule
      set(x, y+1, ~0);            // Set next row if true
  }

  // Move to next row, and stop when we reach the bottom
  if (++y>height)
    noLoop();
}
```

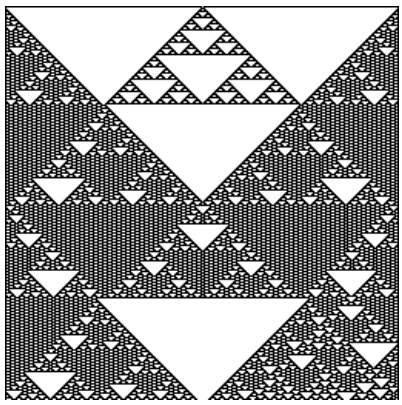


Figure 9.3: Three adjacent pixel cellular automata generated using Rule 129 from the Wolfram naming scheme.

Learning activity

Set the rule in the sketch shown in Figure 9.3 to `int rule=30`.

Also try the following rules: 22, 110, 255.

Explore the complete set of all 255 cellular automata rules. Which ones are the most interesting? Why?

9.5 Two-dimensional Cellular Automata

9.5.1 Conway's Game of Life

The Game of Life is a cellular automaton that was invented in 1970 by the British mathematician John Conway. Although not an interactive game, it has rules that are applied to a two-dimensional grid, much like the rules that are applied to the one-dimensional rows in the cellular automata discussed above.

The rules are deceptively simple:

1. Any live cell with fewer than two live neighbours dies, as if by loneliness.
2. Any live cell with more than three live neighbours dies, as if by overcrowding.
3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbours comes to life.

From these rules emerges extremely complex patterns of behaviour such as self-replication, motion, growth and death.

The sketch in Figure 9.5 is a simple implementation of two *gliders* operating under the rules of Conway's Game of Life. An array variable holds the state of the current and next generation. The above rules are applied using a 3×3 grid centred on every pixel on the screen. We simply count the number of neighbours for each pixel and check against the rules whether the pixel should be switched on or off in the next generation.

The code in `setup()` initialises two patterns called *gliders*. These are patterns that have a 'walking' motion across the screen under the rules of the cellular automaton. The initial pattern for a glider is shown in Figure 9.4(a).

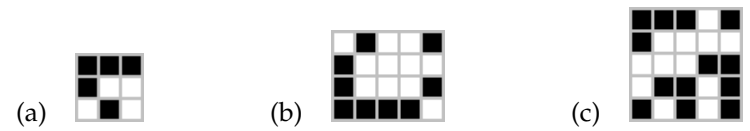


Figure 9.4: Initial states (patterns) for (a) a glider, (b) a spaceship, (c) infinite growth.

Another type of glider is known as the *spaceship*; its initial pattern is not a 3×3 grid, but instead must be constructed on a 5×4 grid. However, the glider works using the standard 3×3 grid rules of the Game of Life. Figure 9.4(b) shows the spaceship initial pattern.

There are also patterns that grow indefinitely and reproduce. One such pattern is shown in Figure 9.4(c). The initial pattern requires a 5×5 cell.

```
// Conway's Game of Life

int s=8;          // scale factor for zooming display window output
int wSz=512;      // display window size
int sx, sy;       // scaled dimensions of window
int world[][][]; // record of current and next state of world
int genA, genB;  // used to indicate which records in world array
                  // refer to current state, and which to next state
void setup() {
  size(wSz,wSz);
  frameRate(10);

  // initialise various variables
  sx = wSz/s;
  sy = wSz/s;
  world = new int[sx][sy][2];
  genA = 0; // current state recorded in world[][][0]
  genB = 1; // next state recorded in world[][][1]

  // set initial conditions: two gliders
  // Note: all other values in world[][][] are initialised
  // by Processing to zero by default, as in Java
  int w=sx/2, h=sy/2;
  world[w-1][h-1][genA]=1; world[w][h-1][genA]=1; world[w+1][h-1][genA]=1;
  world[w-1][h][genA]=1;   world[w][h+1][genA]=1;
  w=sx/3; h=sy/3;
  world[w-1][h-1][genA]=1; world[w][h-1][genA]=1; world[w+1][h-1][genA]=1;
  world[w-1][h][genA]=1;   world[w][h+1][genA]=1;

  // display the initial state of the world
  displayWorld(genA);
}

void draw() {
  // calculate the next state of the world by updating every entry in
  // world[][][genB] according to the Game of Life rules applied to the
  // current state as recorded in world[][][genA]
  for (int y=1; y<sy-1; y++) {
    for (int x=1; x<sx-1; x++) {
      // for each cell in world[][][genA]...
      int sum=0;
      // count the number of 'on' bits in 3x3 neighbourhood
      for (int xx=-1; xx<2; xx++) {
        for (int yy=-1; yy<2; yy++) {
          if (!(xx==0&&yy==0))
            sum += world[x+xx][y+yy][genA];
        }
      }
      // set the corresponding cell in world[][][genB]
      // according to the Game of Life rules
      if (world[x][y][genA]==1) {
        if (sum==2 || sum==3)
          world[x][y][genB]=1;
        else
          world[x][y][genB]=0;
      }
      else {
       if (sum==3)
         world[x][y][genB]=1;
       else
         world[x][y][genB]=0;
      }
    }
  }

  // We've now finished calculate the next state of the world,
  // so we can now display it
  displayWorld(genB);

  // Finally, for the next pass, we'll use the current world[][][genB]
  // as the current state, and record the next state in the current
  // world[][][genA]. We can do this simply by swapping the values of
  // genA and genB.
  genA=1-genA;
  genB=1-genB;
}

void displayWorld(int gen) {
  // display the world recorded in world[][][gen], using scaling factor s
  background(0);
  scale(s);
  stroke(255);
  for (int x=0; x<sx; x++) {
    for (int y=0; y<sy; y++) {
      if (world[x][y][gen]==1)
        point(x,y);
    }
  }
}
```

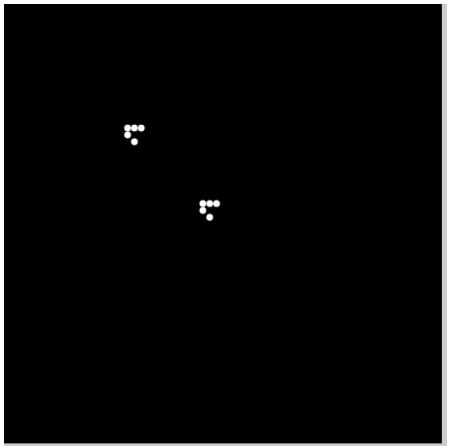


Figure 9.5: Two gliders in Conway's Game of Life.

Learning activity

Add the spaceship and self-replicating patterns to the initial conditions of the sketch in Figure 9.5. You should be careful to place the patterns away from any of the other initial patterns so they don't interfere in the first generation.

Try changing the display window size and the scale factor variables to experiment with worlds of different sizes.

Find a different set of rules for two-dimensional cellular automata. Explore the initial patterns and resulting behaviours for the rules that you found.

9.6 Summary and learning outcomes

The purpose of this chapter was to demonstrate that it is possible to draw complex patterns out of very simple rules. We shall visit this concept again in Creative Computing I: *Image, Sound, Motion Volume 2*, in particular in Chapter 6 on *Generative Systems*.

We first looked at how pixel values could be manipulated as bit patterns, then we looked at a method for implementing one-dimensional cellular automata rules from the bit patterns.

This chapter concluded with a version of Conway's Game of Life, a two-dimensional cellular automaton that is widely known. It is often used to teach fundamental principles of self organisation or distributed intelligence, both of which are advanced subjects in computer science.

You should now be able to:

- contrast the approaches of using line and shape elements, with the approach of manipulating individual pixels, in creating images
- describe Conway's Game of Life
- develop one- and two-dimensional cellular automata, that perform according to specified rules.

9.7 Exercises

1. In this chapter, you saw a one-dimensional cellular automaton and a two-dimensional cellular automaton. What is the difference between these? What would a three-dimensional cellular automaton do? Can you find or create examples of one? Is it possible to have 4-dimensional cellular automaton?
2. This chapter focussed on the use of pixel manipulation to create images. What aspects of a pixel can be manipulated (changed)?
3. You will see more on colour in the next volume of material for this course, but now is also a good time to experiment with the colour values that can be changed by using methods such as `set()`. Modify the sketches in Figures 9.2 and 9.3 to include colour rather than greyscale shading. Again, you should bring a creative component as well as a techinical component to this exercise.

9.8 Looking forward

We have now reached the end of Volume 1 of the subject guide. This volume has introduced some of the history of creativity and the use of technology in a creative domain, and has provided the basic materials needed to start your own creative portfolio. We have discussed the concepts of *shape*, *structure* and *motion* from theoretical and practical perspectives, and have seen how simple rules can be employed to generate complex patterns and behaviours.

You should now be comfortable with using *Processing* to implement creative works based upon the concepts we have covered.

Volume 2 of the subject guide expands on these foundations. It explores in greater depth some of the topics that have been introduced here, such as *colour* and *generative systems*. A variety of new topics are introduced, including *3D graphics*, *3D motion*, the display and manipulation of *image files*, and the use and manipulation of *digital audio*. Volume 2 also discusses creative processes and ways to develop your creative thinking.

Now that you have completed Volume 1, you should find that the range of works you can produce increases rapidly as you progress through the topics covered in Volume 2. We look forward to seeing the results of your work!

Comment form

We welcome any comments you may have on the materials which are sent to you as part of your study pack. Such feedback from students helps us in our effort to improve the materials produced for the International Programmes.
If you have any comments about this guide, either general or specific (including corrections, non-availability of Essential readings, etc.), please take the time to complete and return this form.

Title of this subject guide: .....

.....

Name .....

Address .....

.....

Email .....

Student number .....

For which qualification are you studying? .....

Comments

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Please continue on additional sheets if necessary.

Date: .....

Please send your completed form (or a photocopy of it) to:
Publishing Manager, Publications Office, University of London International Programmes,
Stewart House, 32 Russell Square, London WC1B 5DN, UK.