UNIVERSITY
OF LONDON

Creative computing I: image, sound and motion Volume 2

M. Casey with T. Taylor and M. Magas

CO1112

2019

Undergraduate study in
Computing and related programmes

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London by:
Michael Casey, Department of Music, Dartmouth College, USA.
Tim Taylor, University of London
Michela Magas, Goldsmiths Digital Studios, University of London

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 2008
This edition published 2019

University of London
Publications Office
32 Russell Square
London WC1B 5DN
United Kingdom
london.ac.uk

Published by: University of London

Reprinted with a revised Chapter 5, 2019

Contents

Preface

This course is about expressing creative ideas through *computing*. At the end of the course you will understand some fundamental 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 also 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 1 is divided into two volumes. The first volume introduced you to some of the history of creativity and the use of technology in a creative domain, and gave you the basic materials needed to start your own creative portfolio. The second volume expands on these foundations so that you can develop your own unique tools and methods via programming, and it also discusses creative processes. It is therefore very important that you become familiar with the contents of this guide.

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

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 main texts is weak, it provides some additional background material. Further reading is essential 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. It is also essential that you do the practical creative exercises, and begin developing a portfolio of work and your own creative style.

The images in this guide are black and white in its print version. There is also an accompanying PDF version, available on the virtual learning environment (VLE),[1] that contains all of the images from the guide, in colour. You should download this document so that you are able to see the colour while you are studying some of the material.

[1] You can access the Goldsmiths Computing VLE via the University of London Student Portal (`https://my.london.ac.uk`), or directly via `https://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 `http://www.processing.org/reference`, online *Processing* reference manual.

Additional reading

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

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].

Greenberg, I. *Processing: Creative Coding and Computational Art* (Apress (Springer-Verlag), 2007) [ISBN 159059617X].

Maeda, J. *Creative Code* (Thames and Hudson, 2004) [ISBN 0500285179].

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

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

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

Prusinkiewicz, P. and Lindenmayer, A. *The Algorithmic Beauty of Plants* (Springer-Verlag, softcover reprint, 1996; free electronic version available at `http://algorithmicbotany.org/papers/#abop`) [ISBN 0387946764].

Roads, C. *The Computer Music Tutorial* (MIT Press, 1996) [ISBN 0262680823].

Shiffman, D. *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction, Second Edition* (Morgan Kaufmann, 2015). Chapter 20. [ISBN 0123944430].

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

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

Chapter 1

Colour

Essential reading

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

Additional reading

Greenberg, I. *Processing: Creative Coding and Computational Art* (Apress(Springer-Verlag) 2007) [ISBN 159059617X].

1.1 Introduction

Colour is an integral aspect of visual art. Whether vibrant shades are used, or a black-and-white image is seen as being more appropriate, it is essential that attention is given to colour when creating an artwork.

Different uses of colour can convey psychological and emotional effect; there are also cultural aspects to colour.

This chapter focuses on the technical implementation of colour concepts, and the creation of colour on a computer screen[1]. In this chapter you will learn how to use the two colour modes in *Processing*: RGB and HSB. Each colour mode provides a different model for manipulating colours and drawing using colours.

1.2 *Processing* `color` type

A simple data type is provided for colours called `color`; note the American spelling. We can declare a variable of this type in the usual way: `color myColor;`

The `color` type actually uses the same base type as `int`. That is, variables of type `color` are stored and manipulated in the computer in the same way as `int` variables. But we use a special type for colours so that we know how to interpret the integer value.

Usually, an integer is used for arithmetic operations such as `int x = y + 2` or `int z = a * b;`. These standard arithmetic operators are not meaningful for the `color`

[1] To complement your technical knowledge of how colour works, you should read about the artistic and design aspects of colour too. There are various books and articles that discuss this, and a useful website is `http://www.colormatters.com/`.

type. (Though it is actually possible to perform these operations on `color` variables in *Processing* without a syntax error, the result is not normally useful.)

The `colorMode()` method is used to tell *Processing* how to interpret the integer values used to represent colours. In colour processing the integers are split into their four individual bytes (where each byte contains 8 bits); you will recall that integers in Java and *Processing* are usually 32 bits. Each byte has a meaning that is independent from the other three, and arithmetic operations must be performed on each byte *independently* and the result subsequently recombined into a single integer. We now look more closely at the way that `color` is represented and how to perform operations on colours in *Processing*.

1.3 RGB colour space

By default, *Processing* provides control over the Red, Green and Blue channel of each pixel using the RGB model: `colorMode(RGB)`. In the RGB colour model, each pixel is composed of red, green and blue colour channels. The intensity of light from each of these channels determines the colour that is seen. The model directly corresponds to the colour channels in a colour cathode ray tube or LCD screen.

The bytes represent Alpha, Red, Green and Blue channels respectively. The Alpha channel determines the opacity of the colour; opacity is the inverse of transparency. An alpha value of 255 means that the object drawn in this colour has no transparency.

Tables 1.1, 1.2 and 1.3 show how the primary colours—red, green and blue—are represented within the 32-bit integer, in `colorMode(RGB)`. The tables show the bytes represented as binary, decimal and hexadecimal respectively. You should be familiar with binary and hexadecimal notation. Recall that hexadecimal can be used as a literal in Java by prefixing the notation with "`0x`", zero-x.

byte	1	2	3	4
Red	11111111	11111111	00000000	00000000
Green	11111111	00000000	11111111	00000000
Blue	11111111	00000000	00000000	11111111

Table 1.1: The binary representation of the primary colours Red, Green and Blue as `color` integers in `colorMode(RGB)`.

byte	1	2	3	4
Red	255	255	0	0
Green	255	0	255	0
Blue	255	0	0	255

Table 1.2: The decimal representation of the primary colours Red, Green and Blue as `color` integers in `colorMode(RGB)`.

The set of colours represented by a model is called the *gamut*. The number of possible colours represented in RGB mode is the total number of colour states. Since there are eight bits for each colour, and there are three colours, that is 2^{24} possible states, or 16,777,216 colours. The Alpha channel determines transparency, so it

byte	1	2	3	4
Red	0xFF	0xFF	0x00	0x00
Green	0xFF	0x00	0xFF	0x00
Blue	0xFF	0x00	0x00	0xFF

Table 1.3: The hexadecimal representation of the primary colours Red, Green and Blue as `color` integers in `colorMode(RGB)`.

represents mixing of colours. The resulting values after mixing will always be one of the 16,777,216 possible states. Due to limitations of physical display devices, the entire range of possible visible colours, perceivable by the eye, is not representable by the RGB model.[2]

1.3.1 Defining `color` values

There are (at least) four ways to define colours in *Processing*. The first is to use hexadecimal "hash" notation which sets only the RGB channels. In this representation the alpha channel is set to maximum opacity 0xFF.

```
color red   = #FF0000;    // Red
color green = #00FF00;    // Green
color blue  = #0000FF;    // Blue
color white = #FFFFFF;    // White
color black = #000000;    // Black
```

This hash notation is widely used in web languages such as HTML. We can use hash notation colour values with any of *Processing*'s colour setting methods:

```
background(#00FF00);      // Green background
stroke(#FF0000);          // Red pen colour
fill(#0000FF);            // Blue fill colour
background(green);        // A hash-defined colour
stroke(#FFFFFF);          // White pen colour
```

The second way to define colour values is to use hexadecimal notation 0xAARRGGBB. In this representation the alpha channel is set explicitly, followed by the RGB channels as in the hash notation:

```
color red   = 0xFFFF0000; // Hex Red
color green = 0xFF00FF00; // Hex Green
color blue  = 0xFF0000FF; // Hex Blue
background(0xFF00FF00);   // Green background
stroke(0xFFFFFFFF);       // White pen colour
```

The third way is to use *Processing*'s `color()` construction method. This method takes three or four parameters representing RGB and Alpha (when using `colorMode(RGB)`):

[2] We will explore colour models, and colour perception, in further detail in the *Creative Computing II* course.

```
color red   = color(255,0,0);              // Red
color green = color(0,255,0);              // Green
color blue  = color(0,0,0xFF);             // Blue (in Hex)
color white = color(255,0x7F+0x80,127+128,0xFF);
color redTransparent = color(255,0,0,127);
```

Note that in the four parameter use (e.g. the last two examples shown above), the first three parameters are RGB, and the final parameter is Alpha.

In the last example shown above, we set `redTransparent` to have a transparency value of 127. We shall investigate how to use transparency in Section 1.3.6.

The fourth method to define colours in *Processing* is to use bitwise operations. We can define the value for each byte and then shift the byte to the correct position in the colour integer using the left shift operator (<<). We use the bitwise OR operator (|) to set the bits:

```
color red   = 0xFF<<24 | 0xFF<<16;         // Red
color green = 255<<24  | 255<<8;           // Green
color blue  = 255<<24  | 255;              // Blue
fill(255<<24|255<<8);                      // fill Green
stroke(255<<24);                           // pen Black
```

Understanding how the individual colour channels are represented within a colour integer is essential for colour processing. To see what is inside a colour integer, we can access each of the individual channel values in the following way:

```
float redVal      = red(0xFFFF7F7F);       // =255
float greenVal    = green(0xFFFF7F7F);     // =127
float blueVal     = blue(0xFFFF7F7F);      // =127
float alphaVal    = alpha(0xFFFF7F7F);     // =255
int   redValAlt   = (0xFFFF7F7F>>16)&0xFF; // =255
int   greenValAlt = (0xFFFF7F7F>>8)&0xFF;  // =127
int   blueValAlt  = (0xFFFF7F7F)&0xFF;     // =127
int   alphaValAlt = (0xFFFF7F7F>>24)&0xFF; // =255
```

Note that the *Processing* methods `red()`, `green()`, `blue()` and `alpha()` return `float` values rather than `ints`. The use of these methods is simpler and produces more readable code in comparison to the direct manipulation of bits (as used in the final four examples above). However, if we wish to work with bitwise operators (such as &, |, <<, >>) we must work with `ints` rather than `floats`.

Learning activity

Write a *Processing* sketch to draw the following rectangles using a `color` integer:

1. red=255, green=127, blue=0, alpha=127
2. red=127, green=0, blue=255, alpha=255
3. red=255, green=127, blue=127, alpha=0

What colour is the rectangle in each of these examples?

Explain the appearance of the third rectangle.

1.3.2 Colour mixing

The RGB colour model represents a three-dimensional colour space with axes Red, Green and Blue. The three axes are called Primary Colours, and are mixed to make all other colours in the RGB gamut. Figure 1.1 illustrates the so-called RGB colour cube. We see seven of the eight vertices; the missing vertex is the colour Magenta, which is hidden on the back of the illustrated cube.

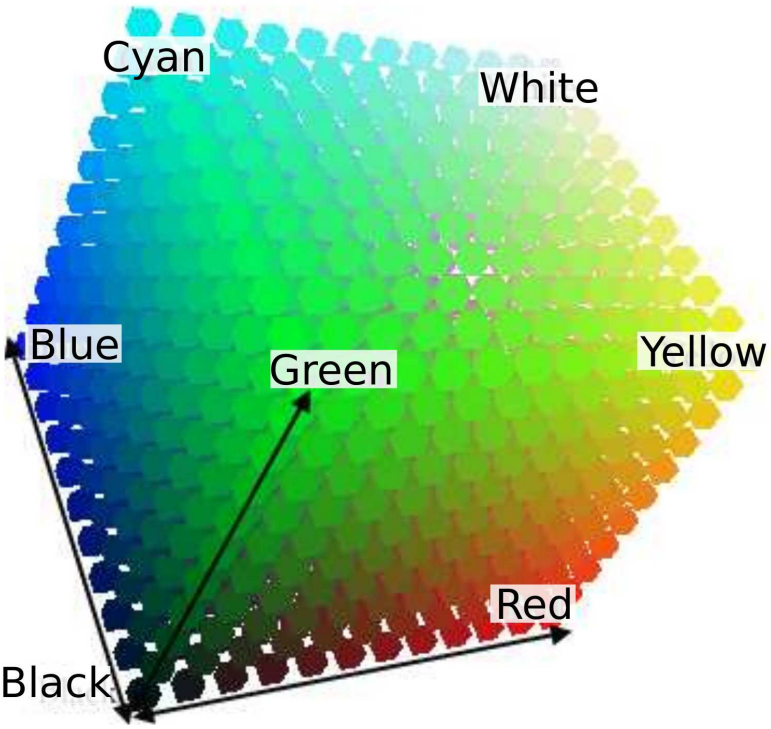


Figure 1.1: `colorMode(RGB)` describes a three-dimensional colour space organised in a cube. The dimensions are red, green and blue as shown in the figure.

We select colours from the RGB colour cube by their position in the three-dimensional space (R,G,B). Figure 1.2 shows an example of using a mixture of primary colours to make new colours. We set the colour mode to RGB with the default range arguments 255, 255, 255. These arguments tell *Processing* the value that we will use to represent the maximum intensity of each colour component. *Processing* automatically scales this value to 255 when we use one of the colour setting methods: `fill(), background(), stroke()`.

In this example we define values for Red, Green and Blue using a selection of the techniques outlined above for illustrative purposes. Normally, we will use a single technique in each of our sketches. We chose the technique that best fits our colour processing requirements. Here, the `Mix` variable is a colour formed out of two primary colours.

Learning activity

Look at Figure 1.2 and answer the following questions:

```
// RGB mode
// Colour representations
int SZ=512;
size(SZ,SZ);
colorMode( RGB, 255, 255, 255);
color Red, Green, Blue, Mix;
Red   = color( 255, 0, 0);
Green = #00FF00;
Blue  = 0xFF0000FF;
Mix   = 0xFFFF7F00;
noStroke();
background(Mix);

fill(Blue);
rect(SZ/6,SZ/6,2*SZ/3,2*SZ/3);

fill(color(255-red(Mix), 255-green(Mix), 255-blue(Mix)));
rect(SZ/3,SZ/3,SZ/3,SZ/3);
```

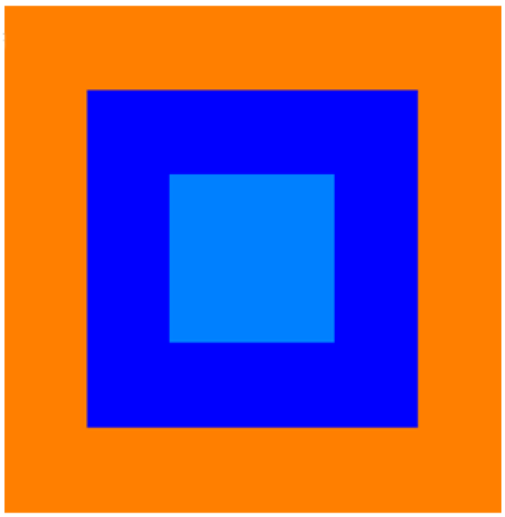


Figure 1.2: In the RGB colour mode each colour channel represents the intensity of red, green and blue light respectively—the *primary* colours. All other colours are made by mixing the amount of red, green and blue light.

Which two primary colours are mixed to make the colour `Mix`?

What colour do they make?

What intensities of each primary colour are used to create the colour `Mix`?

What is the relationship between the outermost square (orange) and the innermost square?

Opposite colours

The inner square of this sketch uses a colour that was constructed out of the Mix colour but using the inverse, or opposite, colour. We constructed the opposite colour by selecting each of the R, G and B channels individually and subtracting their value from 255. In RGB mode, the opposite colour can be constructed as: `Rnew=255-R; Gnew=255-G; Bnew=255-B`:

```
color myCol = 0xFF7FFF00; // mixed colour
color myOppCol=color(255-red(myCol),255-green(myCol),255-blue(myCol));
```

As a general rule, opposite colours blend well together when we have a design using two colours.

Learning activity

1. Make sketches that draw a shape in the opposite colour on the following backgrounds:

```
background(0xFF0000FF);
background(0xFFFF00FF);
background(0xFF00FFFF);
```

2. What is the name of the background colour and opposite colour in each of the above?

3. Make a design using four colours with the `background()`, `stroke()` and `fill()` methods. Your design might be a series of shapes or simply a nested set of rectangles as in Figure 1.2.

1.3.3 Colour gradients

We can use a `for` loop to gradually fade between two colours in a sketch. Figure 1.3 illustrates this by making rectangular strips of colour at regular intervals and gradually fading between green and red.

```
int SZ=512;  // Define screen dimensions
size(SZ,SZ); // Set screen size
// Use screen dimension as colour range
colorMode( RGB , SZ-1 , SZ-1 , SZ-1 );
noStroke();
int step=16;
for( int x=0 ; x < width ; x += step ){
    fill( x , width-x , 0 );
    rect( x , 0 , step , height );
}
```

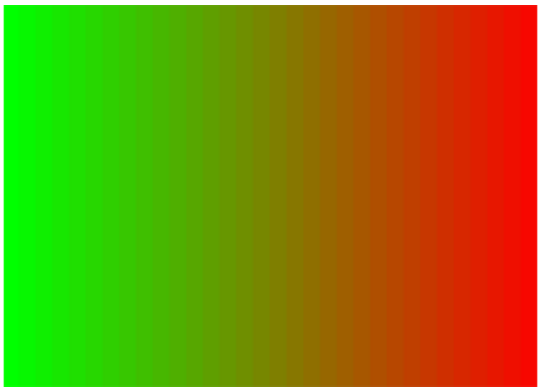


Figure 1.3: A colour *gradient* gradually alters the proportions of the RGB colour channels over a spatial dimension. In this case, the red channel proportion is the `x` position and the green channel is the `width-x` position (the inverse of the red).

The screen width is set to 512 in this example. If we were to use the colour range 0...255, which is the default colour range, then in order to achieve the same colours in the output, we would have to make a second variable, say `colourfade`, in addition to the spatial variable, `x`, to represent the colour. The increment of the colour variable then would be `colourfade*256/512`. This would add more complexity to the code, and would likely make it harder to read and understand. In order to have code that is simple yet flexible, we use the colour range parameters in the `colorMode()` method to set the range to the screen width minus one. Now, the spatial position variable, x, also acts as a colour variable because *Processing* knows to perform the scaling automatically with each call to `fill()`.

This is a very useful technique and will help you to simplify your code enormously when working with colours. It is important to note that the underlying colour representation is still four bytes, each having a range of values from 0...255, but *Processing*'s colour methods scale values that use a different range back to the RGB colour range automatically. This technique also works for values that are less than 255. For example: `colorMode(RGB,1.0,1.0,1.0);` uses the floating point range 0..1 to represent colour values.

So the colour green would be represented as:
```
color green = color(0,1,0);
```

The scaling can be applied to the Alpha channel also:
```
colorMode(RGB,1.0,1.0,1.0,5.0);
```

Here the RGB channels are in the range 0...1, but the Alpha value must be given in the range 0...5.

Figure 1.4 gives an example of mixing two colours in two spatial dimensions. It is a two-dimensional colour gradient with dimensions Red and Green. This example is equivalent to one face of the RGB cube in Figure 1.1.

To achieve the two-dimensional gradient we use a nested `for` loop. The outer loop makes the gradient in the x-dimension, red, and the inner loop makes the gradient in the y-dimension, green. Again we use the range scaling technique on the RGB parameters so that we can use the spatial variables as colour values. This technique saves us having to make a second variable to store colour values or having to make an expression to convert the spatial variables to colour values for every call to `stroke`.

Figure 1.5 illustrates how we can display three faces of the RGB colour cube simultaneously in two dimensions. Each face is represented by one of three concentric circles at each spatial position in the plane. Again, we use the range of spatial variables as our colour parameter range, but we make three different colours for each spatial position in x and y. The outer circle is the mix of red and green based on spatial position. The middle circle is the opposite of the mix of red and blue. The inner circle is the mix of green and blue.

In these examples we use *Processing*'s `stroke()` and `fill()` methods with three parameters. The meaning of the parameters to the colour setting methods is determined by the call to the `colourMode()` method. In this case the three parameters are interpreted as RGB with colour ranges (0... SZ-1, 0... SZ-1, 0... SZ-1). This technique of rescaling the colour ranges to the spatial range greatly simplifies colour programming for sketches that relate colour to spatial position.

Learning activity

Modify the sketch in Figure 1.5 to use a `step=8`. What do you see? How small can `step` be before you cannot see a gradient anymore? How large can `step` be before you cannot see a gradient anymore?

Make a new sketch that displays a gradient in one dimension that moves from a primary colour to its opposite colour.

Make a new sketch that displays a gradient in two dimensions for dimensions: Red-Blue and Blue-Green.

1.3.4 Secondary colours

If we consider the RGB colour space as a cube, with each vertex representing a colour, then the colours on opposite corners of the cube are the opposite colours. Using this knowledge, we can see from Figure 1.1 that the opposite of red is cyan, the opposite of black is white and the opposite of blue is yellow. For the vertex we

```
int SZ=512;
size(SZ,SZ);
int step=16;
colorMode( RGB , SZ-1 , SZ-1 , SZ-1 );
for( int x = step / 2 ; x < width ; x += step )
  for( int y = step / 2 ; y < height ; y += step ){
    strokeWeight( step );
    stroke( x , y , 0 );
    point( x , y );
  }
```

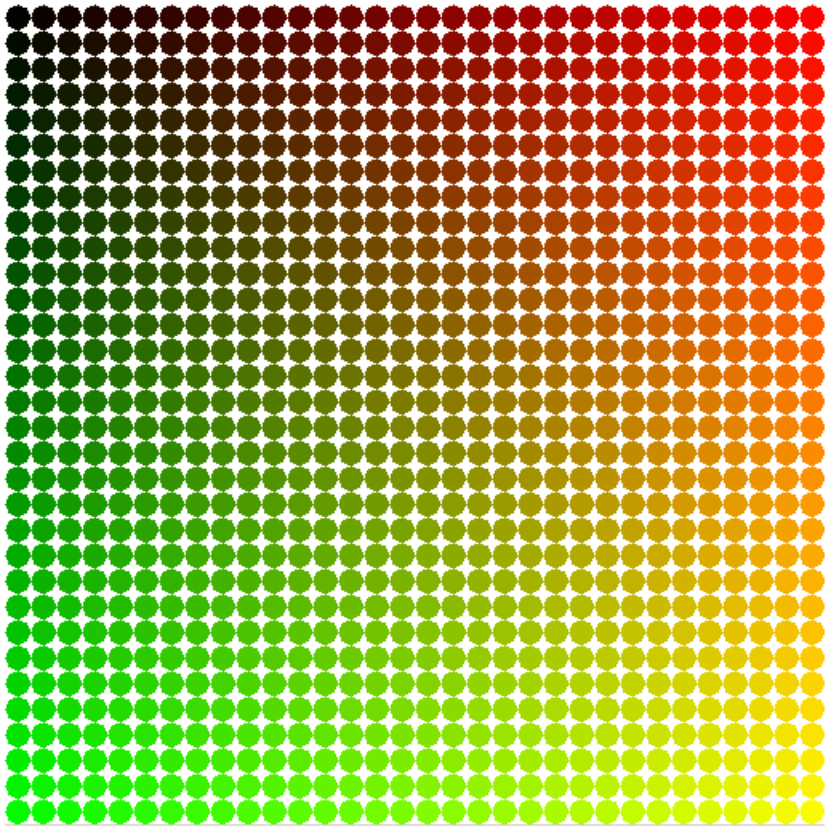


Figure 1.4: Mixing of two colours, red and green, arranged in a plane. The range of colour values is set to the screen `width` and `height` dimensions using `colorMode()`. The vertices of the square (clockwise from top-left) are black, red, yellow and green. Orange is positioned at the mid point of red and yellow, and chartreuse is at the mid point of yellow and green.

```
int SZ=512;
size(SZ,SZ);
colorMode( RGB , SZ-1 , SZ-1 , SZ-1 );
int step=32;
for( int x = step/2 ; x < width ; x += step )
  for( int y = step/2 ; y < height ; y += step ){
    strokeWeight( step );
    stroke( x , y , 0);
    point( x , y );
    strokeWeight( 2 * step / 3. );
    stroke( SZ-x , 0 , SZ-y );
    point( x, y );
    strokeWeight( step / 3. );
    stroke( 0, x , y );
    point( x , y );
  }
```

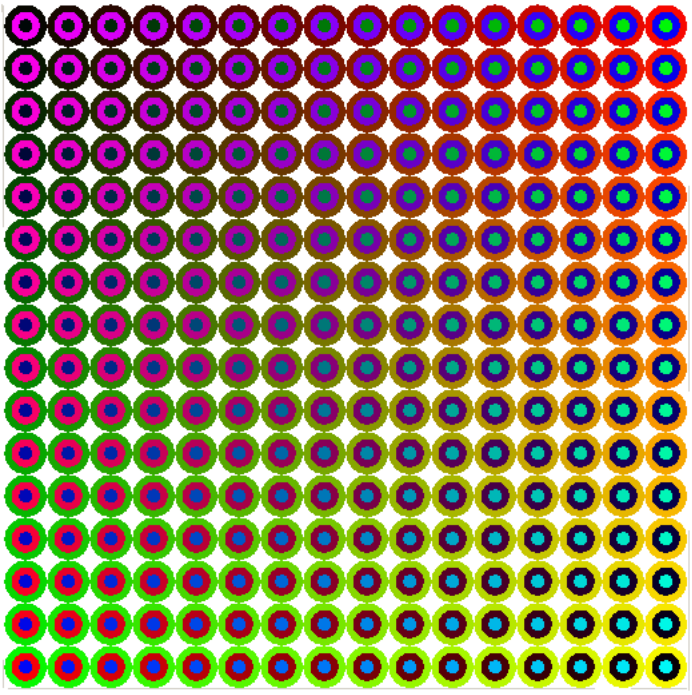


Figure 1.5: Three two-colour mixing processes displayed as concentric circles. The outer circle is the red-green plane, the middle circle is the red-blue plane (inverted so that the values run backwards from high to low) and the innermost circle is the green-blue plane.

cannot see, we infer that the opposite of green is magenta. These are the values at the eight vertices of the RGB cube. Each non-primary vertex is formed by mixing the primaries with full intensity (255). Black is a special case where each colour is mixed with intensity 0.

Consider the three vertices that are formed by mixing pairs of primary colours. We can achieve the correct mixing arithmetic using the bitwise OR (|) operator. These colours are called the *secondary colours*:

```
color Yellow  = 0xFFFF0000|0xFF00FF00;  //    Red | Green
color Magenta = 0xFFFF0000|0xFF0000FF;  //    Red | Blue
color Cyan    = 0xFF00FF00|0xFF0000FF;  //  Green | Blue
```

Figure 1.6 further illustrates the construction of the secondary colours from the primary. Here we use the `color()` method to perform the mixing.

```
// Secondary colours example

color Yellow  = color( 255, 255, 0 );
color Magenta = color( 255, 0, 255 );
color Cyan    = color( 0, 255 ,255 );

void setup() {
  size(512,512);
  noStroke();
}

void draw() {
 int hop=width/3;
 fill(Yellow);  rect(0,0,hop,height); translate(hop,0);
 fill(Cyan);    rect(0,0,hop,height); translate(hop,0);
 fill(Magenta); rect(0,0,hop,height);
}
```

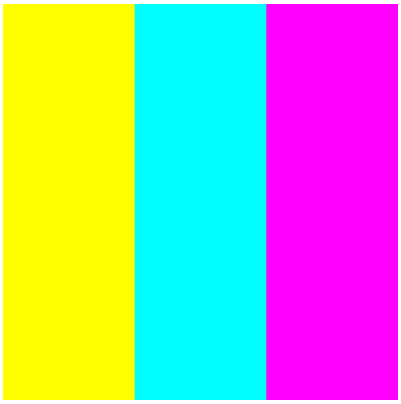


Figure 1.6: The three secondary colours are made by mixing pairs of primary colours: red+green=yellow, red+blue=magenta and green+blue=cyan.

1.3.5 Tertiary colours

Just as the secondary colours were formed by mixing pairs of primaries, the tertiary colours are formed by mixing a primary and a secondary colour. As a secondary colour is formed by the mixing of two primary colours, a tertiary colour can also be considered as the mixture of two primary colours in a 2:1 ratio (with none of the other primary colour). The sketch shown in Figure 1.7 illustrates the construction of the tertiary colours. Table 1.4 enumerates the names and RGB values of each of the tertiary colours. Note that the construction of tertiary colours does not mix a primary colour with its opposite secondary colour. Why?

In Figure 1.7, the construction of tertiary colours is performed by a new method that we defined. This method accepts a primary colour and secondary colour as arguments, but with no checking to see if they are actually primary and secondary. It works by selecting the RGB values of the secondary colour and dividing these quantities by two, then OR-ing the result together with the primary colour. We can also construct tertiary colours directly using hexadecimal notation and following the RGB values in Table 1.4.

```
// Tertiary colours example

// Primary colours
color Red     = color( 255, 0, 0 );
color Green   = color( 0, 255, 0 );
color Blue    = color( 0, 0, 255 );
// Secondary colours
color Yellow  = color( 255, 255, 0 );
color Magenta = color( 255, 0, 255 );
color Cyan    = color( 0, 255 ,255 );

void setup() {
  size(512,512/3);
  noStroke();
  noLoop();
}

// Tertiary colours: mix a primary with non-opposite secondary in 2:1 ratio
color tertiary(color prim, color sec){
   // use bit-wise OR and the color() method to construct a new color
   // using half the red, green and blue values of the secondary
   return prim | color(red(sec)/2 , green(sec)/2 , blue(sec)/2);

   // The following alternative implementation achieves the same result:
   // return color((red(prim)+red(sec))/2, (green(prim)+green(sec))/2,
   //              (blue(prim)+blue(sec))/2);
}

void draw() {
 int hop=width/6;
 // Red-Yellow (Orange)
 fill(tertiary(Red, Yellow));   rect(0,0,hop,height); translate(hop,0);
 // Red-Magenta (Rose)
 fill(tertiary(Red, Magenta));  rect(0,0,hop,height); translate(hop,0);
 // Green-Yellow (Chartreuse)
 fill(tertiary(Green, Yellow)); rect(0,0,hop,height); translate(hop,0);
 // Green-Cyan (Spring Green)
 fill(tertiary(Green, Cyan));   rect(0,0,hop,height); translate(hop,0);
 // Blue-Magenta (Violet)
 fill(tertiary(Blue, Magenta)); rect(0,0,hop,height); translate(hop,0);
 // Blue-Cyan (Azure)
 fill(tertiary(Blue, Cyan));    rect(0,0,hop,height);
}
```

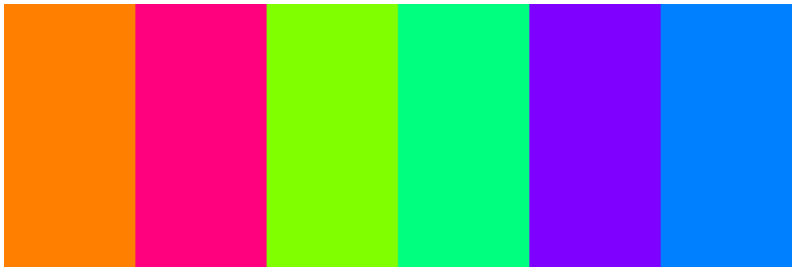


Figure 1.7: The tertiary colours are formed by mixing a primary and a secondary colour, where the secondary colour is not the primary's opposite.

Colour	Primary-Secondary	Red	Green	Blue
Orange	Red-Yellow	0xFF	0x7F	0x00
Chartreuse	Green-Yellow	0x7F	0xFF	0x00
Spring Green	Green-Cyan	0x00	0xFF	0x7F
Azure	Blue-Cyan	0x00	0x7F	0xFF
Violet	Blue-Magenta	0x7F	0x00	0xFF
Rose	Red-Magenta	0xFF	0x00	0x7F

Table 1.4: RGB values for each of the tertiary colours. These are constructed by mixing a primary and non-opposite secondary colour.

Figure 1.8 illustrates a design using all of the tertiary colours. The last two colours, `Color5` and `Color6`, are given transparency values of 0x3F and blended as squares over the triangular design.

Learning activity

Study Figure 1.8. Name each of the colours: `Color1` to `Color6`.

Make a table showing the RGB values and name of the opposite colours for each of the tertiary colours.

How many colours can be made by mixing the secondary and tertiary colours? What are the colour values for each of the mixes of secondary and tertiary colours?

Make a new design using your RGB colour values for mixes of secondary and tertiary colours.

1.3.6 Transparency and colour blending

Artists construct colours by mixing pigments selected from a colour palette. We saw above how to achieve a similar effect by mixing primary, secondary and tertiary colours using the RGB colour model. A further method for constructing colours is available that uses the transparency property, or Alpha channel.

Two colours are mixed in the RGB colour model by making a blend of the RGB values. The amount of each colour included in the blend is determined by the Alpha value. The Alpha channel performs the following arithmetic on each colour channel in RGB mode:

```
color blendAlpha(color Color1, color Color2) {
  float a = (255-alpha(Color2))/255;
  float b = alpha(Color2)/255;
  color RedNew   = (int)(a*red(Color1)   + b*red(Color2)  );
  color GreenNew = (int)(a*green(Color1) + b*green(Color2));
  color BlueNew  = (int)(a*blue(Color1)  + b*blue(Color2) );
  color AlphaNew = (int)(a*alpha(Color1) + b*alpha(Color2));
  return color(RedNew,GreenNew,BlueNew,AlphaNew));
}
```

```
// RGB mode
// Colour representations in Processing
int SZ=512;
size(SZ,SZ);
color Color1, Color2, Color3, Color4, Color5, Color6;
// Web color representation #RRGGBB from hex values
Color1 = #FF7F00;
Color2 = #7F00FF;
Color3 = #00FF7F;
Color4 = #007FFF;
// HEX color representation 0xAARRGGBB
Color5 = 0x3FFF007F;
Color6 = 0x3F7FFF00;
noStroke();
fill(Color1);
triangle(0,0,SZ,0,SZ/2,SZ/2);
fill(Color2);
triangle(0,0,0,SZ,SZ/2,SZ/2);
fill(Color3);
triangle(0,SZ,SZ,SZ,SZ/2,SZ/2);
fill(Color4);
triangle(SZ,SZ,SZ,0,SZ/2,SZ/2);
fill(Color5);
rect(SZ/6,SZ/6,2*SZ/3,2*SZ/3);
fill(Color6);
rect(SZ/3,SZ/3,SZ/3,SZ/3);
```

Figure 1.8: The internal colour representation is four bytes, one each for R, G and B and an Alpha channel, A. The Alpha channel controls the amount of transparency. Here, each colour is a mix of two primary colours in 2:1 ratio forming the tertiary colours. The two inner squares are 50% transparent.

Learning activity

Look at the `blendAlpha()` code above.

What proportion of the RGB values of `Color1` are included in the result if the Alpha value of `Color2` is 127?

What proportion of the RGB values of `Color1` are included in the result if the Alpha value of `Color1` is 127?

Recall that all of the *Processing* colour setting methods accept arguments for the individual colour channels. We can use these arguments to provide transparency information and perform colour mixing using its internal version of the `blendAlpha()` method. The following code implements colour mixing by transparency for tertiary colours:

```
color red   = 0xFFFF0000;
color green = 0xFF00FF00;
// draw a red circle with 100% opacity
fill(red);
ellipse(SZ/2,SZ/2,SZ/2,SZ/2);
// overlay a red-green cicle with 50% opacity
fill(red | green, 0x7F);
ellipse(SZ/2,SZ/2,SZ/2,SZ/2);
// overall result is an orange circle
```

Learning activity

Modify the transparency mixing code above to construct the other five tertiary colours.

Why do we use the OR operator to construct the secondary colours rather than the Alpha value?

There are various other ways in which we can mix colours in *Processing* to achieve effects that are desirable for making good designs. For example, we can use the `PImage` methods `get(), set() and blend()` to mix multiple images in a variety of ways.[3]

Figure 1.9 illustrates how we can use the `blend()` method to produce a colour mix only on the overlap of two filled shapes. The parts of the shapes that do not overlap remain in their original state. This example shows the blending of three layers using the `ADD` blend mode.

The sketch works by drawing three circles independently using the three primary colours. Each circle is captured as an image and blended with the next to create a composite blend of all three. The result is a colour Venn diagram illustrating the mixing process for the secondary colours.

To obtain the tertiary colours we can apply exactly the same principle again, taking the final mixing diagram of Figure 1.9 and blending it with a rotated copy of itself. Figure 1.10, on page 17, illustrates how this is done using *Processing*'s `Pimage` methods.

[3] We will look at these `PImage` methods in more detail in Chapter 4.

```
// Colour mixing using transparency
// Primary colour mix of 3 circles to get secondary colours
// Blend the different layers to achieve correct colour mixing
int SZ=512;
size(SZ,SZ);
background(0);            // Default background
noStroke();
PImage p0=get();          // Save background in p0
float B=SZ/4;
float A=sqrt(3)*(B)/2;
// LAYER 1
fill(#FF0000);            // Red
ellipse(SZ/2,SZ/2-2*A/3,2*B,2*B);
PImage p1=get();          // Save image in p1
image(p0,0,0);            // Restore blank background
// LAYER 2
fill(#00FF00);            // Green
ellipse(SZ/2-B/2,SZ/2+A/3,2*B,2*B);
PImage p2=get();          // Save image in p2
p2.blend(p1,0,0,SZ,SZ,0,0,SZ,SZ,ADD); // blend p2 with p1
image(p0,0,0);            // Restore blank background
// LAYER 3
fill(#0000FF);            // Blue
ellipse(SZ/2+B/2,SZ/2+A/3,2*B,2*B);
PImage p3=get();          // Save impage in p3
p3.blend(p2,0,0,SZ,SZ,0,0,SZ,SZ,ADD); // blend p3 with (blended) p2
image(p3,0,0);
```

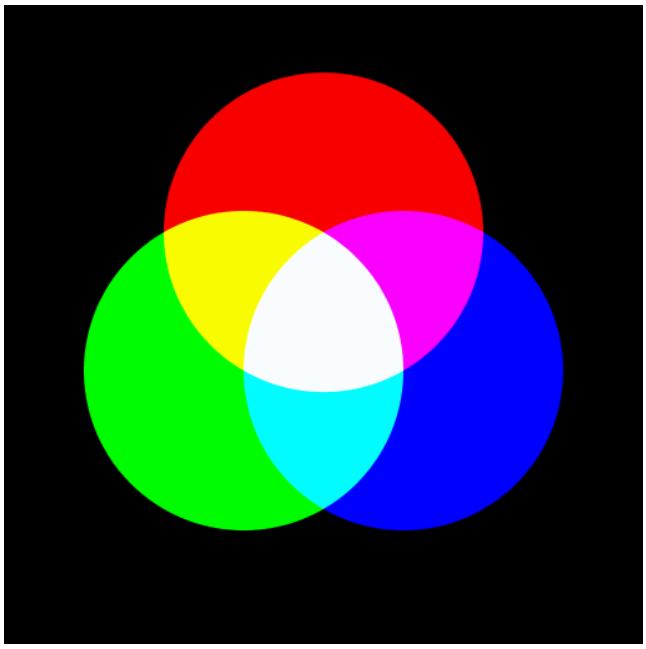


Figure 1.9: Construction of the secondary colours by additive blending, using the `blend()` method with `ADD` mode.

```
// Colour mixing using transparency
// Primary colour mix of 3 circles to get secondary colours
// Blend the different layers to achieve correct colour mixing
// Blend secondary colours with a rotated copy of image to construct
// tertiary colours
int SZ=512;
size(SZ,SZ);
background(0);            // Default background
noStroke();
PImage p0=get();          // Save background in p0
float B=SZ/4;
float A=sqrt(3)*(B)/2;
// LAYER 1
fill(#FF0000);            // Red
ellipse(SZ/2,SZ/2-2*A/3,2*B,2*B);
PImage p1=get();          // Save image in p1
image(p0,0,0);            // Restore blank background
// LAYER 2
fill(#00FF00);            // Green
ellipse(SZ/2-B/2,SZ/2+A/3,2*B,2*B);
PImage p2=get();          // Save image in p2
p2.blend(p1,0,0,SZ,SZ,0,0,SZ,SZ,ADD); // blend p2 with p1
image(p0,0,0);            // Restore blank background
// LAYER 3
fill(#0000FF);            // Blue
ellipse(SZ/2+B/2,SZ/2+A/3,2*B,2*B);
PImage p3=get();          // Save image in p3
p3.blend(p2,0,0,SZ,SZ,0,0,SZ,SZ,ADD); // blend p3 with (blended) p2
// LAYER 4
tint( 0xFFFFFFFF );
image( p3,0,0 );          // display (blended) p3 with full opacity
translate( SZ/2,SZ/2 );   //
rotate( PI/3 );           // rotate axes by 60 degrees around centre point
translate( -SZ/2,-SZ/2 ); //
tint( 0x7FFFFFFF );
image( p3,0,0 );          // display rotated p3 with 50% opacity
```

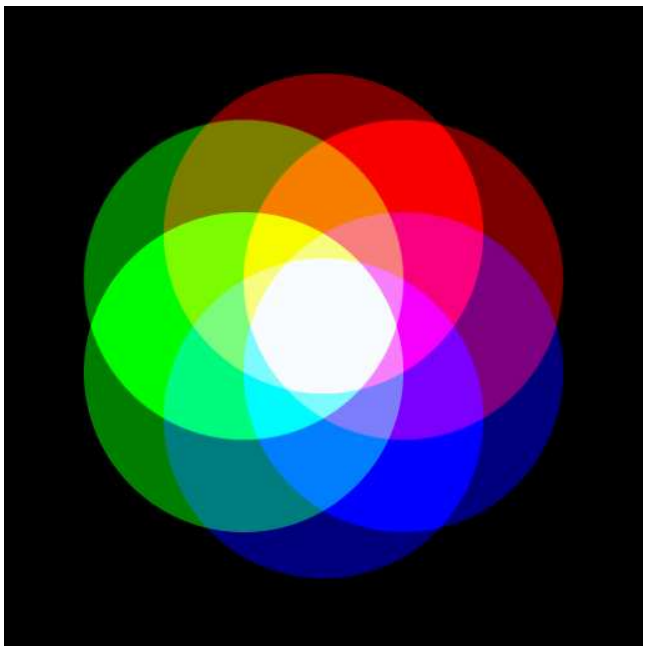


Figure 1.10: Construction of the tertiary colours by blending the secondary colour Venn diagram with a rotated copy of itself. The rotation here is 60 degrees, or $\pi/3$ radians.

We do not have to stop at three levels of mixing. We can construct a colour wheel out of as many levels of mixing as we want. Figure 1.11, on page 19, illustrates the mixing diagram for eight-levels of mixing. Here we have applied the same principle of rotating and blending the output of each level's blend process with the previous level's output.

By applying the principle of mixing the primary colours iteratively in this manner, we have constructed a representation of a different colour space. This is called the HSB colour space and it is the subject of the next section.

1.4 HSB colour space

The second colour mode is Hue Saturation Brightness (HSB), and is set using the `colorMode(HSB)` command. For many purposes the HSB colour space is simpler to use than the RGB colour space. The reason for the difficulty with RGB is that the resulting colour is hard to predict when it isn't one of the primary, secondary or tertiary colours. Furthermore, control over the brightness of the colours is achieved by changing the value of all three colour channels.

The HSB colour model—also referred to as HSV—is thought to be more similar to the way human beings perceive colour than the RGB model. The HSB model attempts to simplify the construction of colours by separating out perceptual attributes of colour: Hue, Saturation and Brightness. Hue is a continuous range of colours arrranged on a circle. Figure 1.12 shows the hue colour wheel for the three primary colours, three secondary colours and the six tertiary colours making a total of twelve sectors. Can you identify the sectors by their colours? What are the start angles of each of the colours? It is good to know these: you might consider committing them to your memory, or at least knowing how to generate this colour wheel for reference.

Brightness is a single value that controls how bright or dark the colour appears and saturation controls how washed out the colour appears, defined as the absence of white. Zero saturation gives 100% white, 50% saturation gives a half mix of the chosen hue and white, and so on.

Figure 1.13 illustrates the construction of HSB colour space using gradients of saturation values decreasing towards the centre of the circle.

In these examples, we use `colorMode(HSB)` with three scale parameters `(TWO_PI, 1.0, 1.0)` representing the ranges of the hue value, the saturation value and the brightness value respectively. The reason we use `TWO_PI` for the hue value is that the colour space organises hue on a circle. This means that the values at 0 and `TWO_PI` are the same colour. All hues around the circle vary smoothly in terms of human perception, i.e. with no jumps (or discontinuities).

HSB mode is not supported by the `color` datatype in *Processing*. All colours are represented as RGB at the lowest level. The colour setting methods, however, do support HSB mode and interpret the parameters according to the `colorMode(HSB,...)` arguments.

```
// Colour mixing using transparency
// Primary colour mix of 3 circles to get secondary colours
// Blend the different layers to achieve correct colour mixing
// Blend secondary colours with a rotated copy of image to construct
// tertiary colours. Repeat rotation a number of times.
int SZ=512;
size(SZ,SZ);
background(0);              // Default background
noStroke();
PImage p0=get();            // Save background in p0
float B=SZ/4;
float A=sqrt(3)*(B)/2;
// LAYER 1
fill(#FF0000);              // Red
ellipse(SZ/2,SZ/2-2*A/3,2*B,2*B);
PImage p1=get();            // Save image in p1
image(p0,0,0);              // Restore blank background
// LAYER 2
fill(#00FF00);              // Green
ellipse(SZ/2-B/2,SZ/2+A/3,2*B,2*B);
PImage p2=get();            // Save image in p2
p2.blend(p1,0,0,SZ,SZ,0,0,SZ,SZ,ADD); // blend p2 with p1
image(p0,0,0);              // Restore blank background
// LAYER 3
fill(#0000FF);              // Blue
ellipse(SZ/2+B/2,SZ/2+A/3,2*B,2*B);
PImage p3=get();            // Save image in p3
p3.blend(p2,0,0,SZ,SZ,0,0,SZ,SZ,ADD); // blend p3 with (blended) p2
image(p3,0,0);              // Restore p3 to display window
// LAYER 4
for (float a=PI/3; a>=PI/24; a/=2) {
  p0=get();                 // save current image in p0
  tint( 0xFFFFFFFF );
  image( p0,0,0 );          // display p0 with full opacity
  translate( SZ/2,SZ/2 );   //
  rotate( a );              // rotate axes around centre point
  translate( -SZ/2,-SZ/2 ); //
  tint( 0x7FFFFFFF );
  image( p0,0,0 );          // display rotated p0 with 50% opacity
}
```

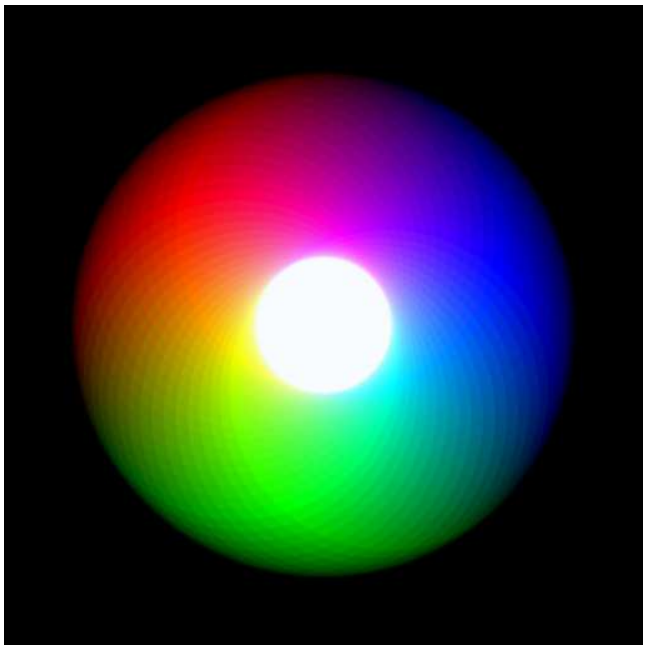


Figure 1.11: A colour wheel of continuous variation between the primary colours is constructed by applying the blending principle iteratively. At each iteration we take a snap shot of the colour wheel, then blend it with a rotated copy of itself to get the next colour wheel.

```
// The twelve primaries, secondaries and tertiaries
// on the HSB colour wheel.

int SZ=512;
float ang,dAng;
void setup(){
  size(SZ,SZ);
  colorMode(HSB,TWO_PI,1,1);
  noStroke();
  ang=0;
  dAng=PI/6;
}
void draw(){
  fill(ang,1,1);
  arc(SZ/2,SZ/2,SZ,SZ,ang,ang+dAng);
  ang+=dAng;
  ang%=TWO_PI;
}
```

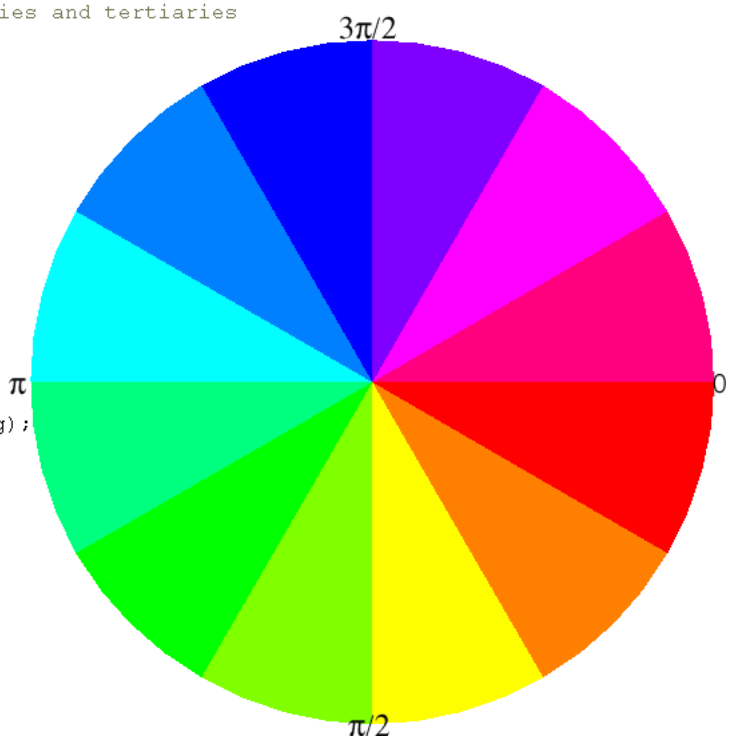


Figure 1.12: Sketch illustrating the circularity of the HSB colour space for fixed brightness and saturation. The twelve sectors (pie wedges) are coloured by each of the three primary colours, interlaced with each of the secondary colours, interlaced with each of the tertiary colours.

```
// HSB Colour Wheel
// This example illustrates the HSB colour space with ranges
// TWO_PI, 1.0, 1.0. We express Hue in radians so its maximum
// value is TWO_PI

size(512,512);                // size of the window that we will draw into
background(0);                // black background
colorMode(HSB,TWO_PI,1.0,1.0); // the colour model
noStroke();

int HALFW=width/2;  // Half screen width global variable
int HALFH=height/2; // Half screen height global variable
int N=16;           // Total number of Hue segments

// We now compute the angle required to cut a 'pie' into N equal wedges.
// So 2*PI/N is the angle between equal-sized 'pie' wedges
float AngN=2*PI/N; // angN is in radians,(a>=0.0)&&(a<2*PI), hence
                   // requires a float

// Draw concentric arcs filled with Hue (angle), and Saturation by (radius)
// colour pallette leaving the Brightness fixed.
// Nested for(...) loops are used to achieve nested circles
float Bri=1.0; // brightness is fixed to maximum (1.0)
for(float Hue=0.0; Hue<2*PI; Hue+=AngN) {
  for(float Sat=1.0; Sat>0.0; Sat-=2.0/N) {
      // Set the colour using the HSB colour model
      fill(Hue,Sat,Bri);
      // Local variable 'Rad' maps from 'Sat' to radius of the arc
      float Rad=Sat*width;
      // Each arc segment is a pie wedge extending from angle Hue to
      // Hue+AngN radians. Local variable d used for diameter of arc
      arc(HALFW,HALFH,Rad,Rad,Hue,Hue+AngN);
   }
}
```

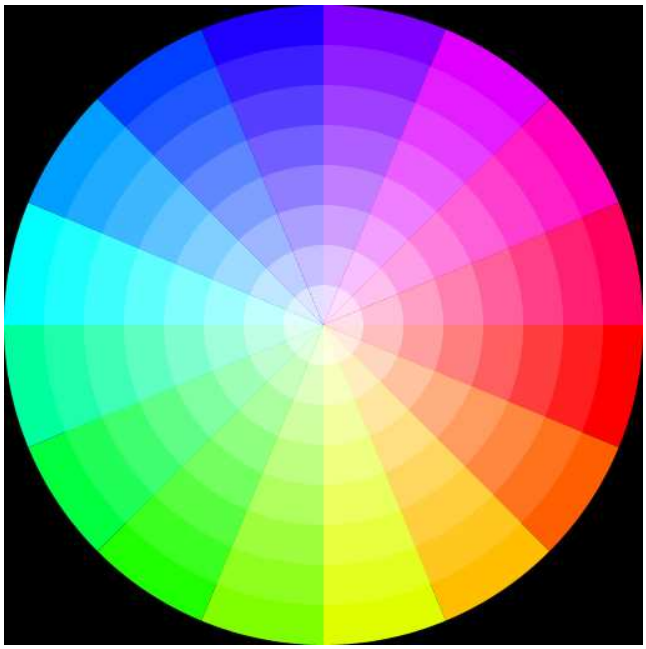


Figure 1.13: Sketch illustrating HSB colour space for fixed brightness at eight levels of saturation represented as concentric arc sections.

From the colour wheel it is easy to see how to construct the primary colours:

```
colorMode(HSB, TWO_PI, 1.0, 1.0);
color red   = color(0, 1, 1);
color green = color(4 * TWO_PI/12, 1, 1);
color blue  = color(8 * TWO_PI/12, 1, 1);
```

It is also clear where the secondary colours are on the wheel—they are between the primary colours:

```
colorMode(HSB, TWO_PI, 1.0, 1.0);
color yellow  = color(2 * TWO_PI/12, 1, 1);
color cyan    = color(6 * TWO_PI/12,1,1);
color magenta = color(10 * TWO_PI/12,1,1);
```

To get the tertiary colours we find the mid points of the primary and secondary colours. We use a denominator of twelve to find all of the three primaries, three secondaries and six tertiary colours.

```
colorMode(HSB, TWO_PI, 1.0, 1.0);
float step = TWO_PI/12;
color red         = color(0*step,1,1);
color orange      = color(1*step,1,1);
color yellow      = color(2*step,1,1);
color chartreuse  = color(3*step,1,1);
color green       = color(4*step,1,1);
color springgreen = color(5*step,1,1);
color cyan        = color(6*step,1,1);
color azure       = color(7*step,1,1);
color blue        = color(8*step,1,1);
color violet      = color(9*step,1,1);
color magenta     = color(10*step,1,1);
color rose        = color(11*step,1,1);
```

Learning activity

Construct the HSB colour wheel for the quaternary colours. This requires drawing 24 segments instead of 12.

Control over the brightness is simply a single parameter. To get dark green we use:

```
colorMode(HSB, TWO_PI, 1, 1);
color darkGreen = color(4 * TWO_PI/12 , 1, 0.5);
```

The commands that we are able to use in HSB mode are the colour setting commands—`background()`, `fill()`, `stroke()`, `color()`—and three special commands that allow us to access the HSB values individually for a given colour—`hue()`, `saturation()`, `brightness()`. Note that Alpha—`alpha()`—is also defined for HSB mode; however, the blending is performed on the RGB equivalents of the HSB colours when applying transparency operations.

1.5 Colour schemes

In HSB mode it is easy to construct a colour scheme to be used in a design. A colour scheme is a principled organisation of colours that both helps the designer to control the colour aspects of the design and also establishes a visual language that the viewer will find easier to understand.

Most colour schemes are organised around the hue wheel from HSB mode. Colour schemes define a fixed number of colours that will be used in the design and then select from the hue wheel, beginning with a given starting colour.

A common colour scheme for two-colours is the use of opposite colours. We already saw (page 6) that opposite colours required us to compute the inverse of each colour channel in RGB mode.

Figure 1.14 shows a colour scheme based on opposite colours. This example uses RGB mode, and the inverse is selected using the bitwise NOT operator in this example.

In HSB mode it is a little simpler to select the opposite colour, as it is the colour on the opposite side of the hue wheel.

For example, the colour red is at angle 0 and its opposite, cyan, is at 180 degrees or `PI` radians (if we use `TWO_PI` as the hue colour range). This is true for all colours. The opposite of the primary colour blue is the secondary colour yellow. The primary colour blue is at angle `2*TWO_PI/3` and its opposite is at angle `PI/3`—which is `2*TWO_PI/3 - PI` or `(2*TWO_PI/3 + PI)%TWO_PI`. We use the modulus operator (`%`) to wrap the angle around to zero when the value reaches `TWO_PI`. In this way we implement the circularity of the hue colour wheel.

1.5.1 Two-colour schemes

To illustrate this circularity, and how to select opposite colours in a two-colour HSB scheme, Figure 1.15 shows a two-colour dial that rotates around the hue wheel and selects opposite colours for the two triangular arms.

The colour selection code is greatly simplified by the fact that the position of the arms around the circle is also the position of the colour on the hue wheel. We achieve this by using the parameter range technique scaled to `TWO_PI`.

1.5.2 Three-colour schemes

As we select more colours for a colour scheme we use different spacings around the hue wheel. Evenly-spaced colours are considered to be pleasing to the eye when used in a colour scheme. In general, it is best to avoid selecting colours from a small region of the hue wheel, and instead, select colours that use the entire wheel or a good proportion of it.

Figure 1.16 gives an example of a three-colour scheme constructed by selecting colours at even intervals around the hue wheel for a given starting colour.

```
// Two-colour schemes using opposite colours
int SZ=512;
size(SZ,SZ);
noStroke();
ellipseMode(CORNER);
for (int j=0; j<2; j++) {          // produce two rows of output
  color ff = 0x00FF0000;           // store current background colour
                                   //  (starting with red)
  if (j==1)                        // invert colour scheme for
    ff = ~ff;                      //  second row
  for(int i=0; i<3; i++) {         // produce three columns of output
    fill(ff|0xFF000000);           // fill colour, fully opaque
    rect(i*SZ/3, j*SZ/2, SZ, SZ); // draw background rectangle
    fill((~ff)|0xFF000000);        // fill colour is opposite of ff
    ellipse(i*SZ/3, j*SZ/2, SZ/3, SZ/2); // draw ellipse
    ff>>=8;                        // move to next primary colour
  }
}
```

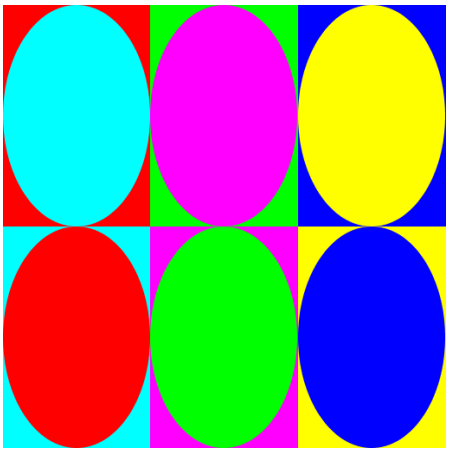


Figure 1.14: Two-colour scheme generation in RGB mode. Here arithmetic and bitwise-logical operations in Java are available to use because the fundamental datatype is an integer (int). This example uses procedural construction to make a series of two-colour schemes based on opposite colours.

```
// Two-colour dial
// Illustrates circularity of hue wheel and selection of
// opposite colours

int SZ  = 512;  // size of window
float a = 0;    // current angle of dial hand
float B = SZ/3;
float A = sqrt(3)*B/2.0;

void setup() {
  size(SZ,SZ);
  noStroke();
  colorMode(HSB, TWO_PI, 1, 1);
  // white background (saturation 0)
  background(1,0,1);
  // draw colour wheel for dial face
  float dAng=TWO_PI/128;
  for (float ang=PI/2; ang<TWO_PI+PI/2; ang+=dAng) {
    fill((ang-PI/2),1,1);
    arc(SZ/2,SZ/2,SZ,SZ,ang,ang+dAng);
  }
}

void draw() {
  // draw central white circle on dial face
  // (erases previous dial hands)
  fill(1,0,1);
  ellipse(SZ/2,SZ/2,SZ-128,SZ-128);
  // rotate axes around dial centre to current hand angle
  translate(SZ/2,SZ/2);
  rotate(a);
  translate(-SZ/2,-SZ/2);
  // select fill colour according to current hand angle
  fill((a+PI)%TWO_PI,1,1);
  // draw large triangle of hand
  triangle(SZ/2-B/2,SZ/2+A/3,SZ/2+B/2,SZ/2+A/3,SZ/2,SZ/2-4*A/3);
  // rotate 180 degrees
  translate(SZ/2,SZ/2);
  rotate(PI);
  translate(-SZ/2,-SZ/2-2*A/3);
  // select opposite colour for drawing smaller triangle
  fill((a)%TWO_PI,1,1);
  // draw small triangle of hand
  triangle(SZ/2-B/2,SZ/2+A/3,SZ/2+B/2,SZ/2+A/3,SZ/2,SZ/2-2*A/3);
  // and advance hand angle for next time
  a+=PI/256;
  a%=TWO_PI;
}
```

Figure 1.15: Two-colour scheme generation in HSB mode. The example is a dial that rotates around the hue circle and selects the pointed-to colour and its opposite. The arms of the dial are then coloured with these selected values and the rotation continues. Colour schemes are based on sampling from the hue circle at even intervals. For N colours the sample interval is generally `TWO_PI/N`.

```
// Generation of three-colour schemes by even
// sampling of hue wheel using HSB mode

int SZ=512;            // size of window
float a=0;             // current position on hue wheel
float dAng=TWO_PI/512;
float B=SZ*0.61;
float A=sqrt(3)*B/2.0;
float n=SZ;
float m=SZ*.61;

void setup() {
  size(SZ,(int)(SZ*0.61));
  noStroke();
  colorMode(HSB, TWO_PI, 1, 1);
  background(0);
  frameRate(10);
  ellipseMode(CORNER);
}

void draw() {
  // select three colours evenly spaced at 2PI/3
  // intervals around the hue wheel, starting at angle
  // specified by a
  color c1 = color(a%TWO_PI,1,1);
  color c2 = color((a+TWO_PI/3)%TWO_PI,1,1);
  color c3 = color((a+2*TWO_PI/3)%TWO_PI,1,1);
  // draw some shapes with the colours
  background(c1);
  fill(c2);
  rect(0,0,m,m);
  fill(c3);
  triangle(m/2-B/2,A,m/2+B/2,A,m/2,0);
  fill(c3);
  rect(m,n-m,n-m,n-m);
  fill(c1);
  ellipse(m,n-m,n-m,n-m);
  fill(c2);
  triangle(m,0,n,0,n,n-m);
  // and advance colour angle for next time
  a+=dAng;
}
```

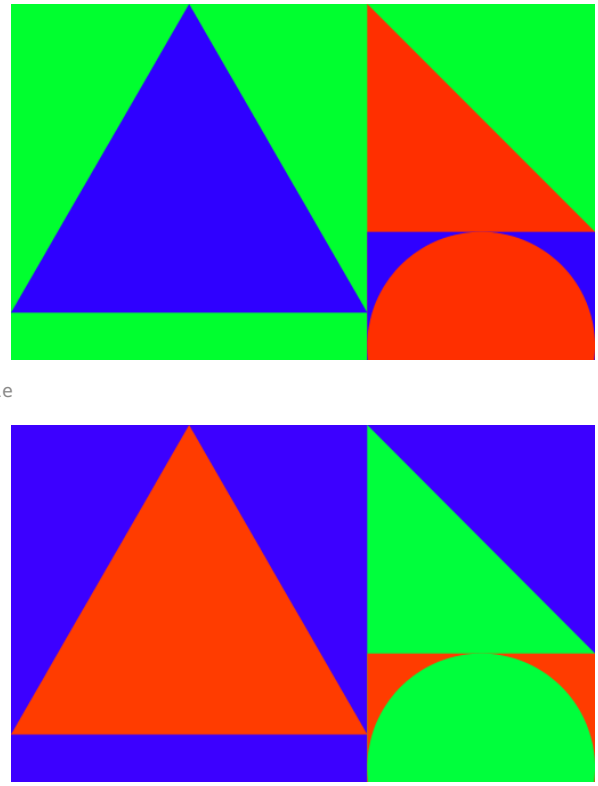


Figure 1.16: Generation of three-colour schemes in HSB mode. The colours are selected by evenly sampling from the hue wheel. In this case the distance between colours is `TWO_PI/3`. This example cycles through the colour when selecting three points spaced around the hue wheel.

1.6 Summary

In this chapter we learnt how to use the two colour modes in *Processing*: RGB and HSB. The RGB colour model is the fundamental colour model in Java, and many computer packages for graphics, images, video and other visual media. As such, there are basic data types that hold colour values and perform arithmetic and bitwise operations on them in RGB mode. However, we saw that colour selection was complicated in RGB mode and that it is simplified if we use the HSB model instead. HSB mode is supported by all of *Processing*'s colour setting methods, but it is not supported by the low-level operations of Java. The HSB colour mode lends itself very well to constructing colour schemes from which we can make coherent designs; these were the subject of the last part of this chapter. You should now be able to construct a colour scheme using either RGB or HSB mode and apply it to your own designs.

You should now be able to:

- distinguish between the RGB and HSB colour modes
- create interesting images in *Processing* using the `color` type
- describe how the red, green, blue and Alpha channels together contribute to colour in images
- use the concept of transparency, and implement the concepts of colour blending and colour gradients
- discuss colour schemes, and the use of primary, secondary and tertiary colours.

1.7 Exercises

1. Make different three-colour schemes using colours chosen at intervals around the hue circle. Draw three different filled shapes (e.g. triangle, rectangle, ellipse) using your scheme including each of the following colours in your scheme:
   (a) red
   (b) chartreuse
   (c) violet.
2. Make a four-colour scheme using equal intervals around the hue circle and use it to lay out a design with four objects.
3. (a) Make a non-uniform interval five-colour scheme. A non-uniform scheme might take three colours from the upper half of the hue circle, and two from the lower half. Use principles of trigonometry to space the colours evenly in each half of the hue circle.
   (b) Use your colour scheme to make a five-colour design.
   (c) Rotate the hue wheel for your scheme using a loop so that the colours change but the hue angles between the colours remain constant.
4. Make a colour gradient, with two tertiary colours as the end-points, that changes horizontally across the screen.
5. Make a colour gradient, with two different tertiary colours as the end-points, that changes vertically down the screen.

6. Make a colour gradient, with two different tertiary colours as the end-points, that changes diagonally across the screen.
7. In Section 1.3 of this guide, there is a sentence: "*Due to limitations of physical display devices, the entire range of possible visible colours, perceivable by the eye, is not representable by the RGB model.*" Explain why this is the case. Is it possible to represent the entire range of possible visible colours using the HSB model? Explain why or why not.

Chapter 2

3D Graphics

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 `http://www.processing.org/reference`, online *Processing* reference manual.

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].
Rand, P. *A Designer's Art* (Yale University Press, new edition 2001) [ISBN 0300082827].
Wong, W. *Principles of Form and Design* (Wiley, 1993) [ISBN 0471285528].

2.1 Introduction

Some of the first uses for 3D computer graphics came from military simulation and aircraft design in the 1970s and 1980s, though work in 3D graphics began even prior to this in the 1960s. Early in the 1980s, Ken Perlin and other famous computer graphics artists brought computer graphics to the film industry with films such as *Tron* leading the way.

A decade later the first all-3D computer animated feature film was produced by Steve Jobs' company Pixar, in association with the Walt Disney company: *Toy Story*. At around the same time, the first version of the popular 3D first-person shooter game called *Doom* was released, which brought interactive 3D graphics to desktop computers with modest resources for the first time.

At the height of its popularity in 2007, the online virtual environment *Second Life*, shown in Figure 2.1, reached an activity level of several million active users.

3D applications are now commonplace on desktop, mobile and web applications. The ever growing appetite for 3D applications is demonstrated by the level of interest shown in the development of the *Oculus Rift* virtual reality head-mounted display.[1] In August 2012, a developers' version of the hardware was funded by a *Kickstarter* campaign that raised its target of US$ 250,000 within four hours of bidding and eventually attracted funding of almost ten times that amount.[2,3] The company was subsequently bought by *Facebook* for over US$2 billion in March 2014.[4]

[1] `http://www.oculusvr.com/`
[2] `http://www.bbc.com/news/technology-19085967`
[3] `https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game`
[4] `http://en.wikipedia.org/wiki/Oculus_Rift`

Figure 2.1: A scene from the online 3D interactive virtual environment *Second Life*.

In this chapter, you will explore the capabilities of the OpenGL 3D graphics interface provided in *Processing*. The fundamentals of 3D graphics are similar to the concepts you have learned in 2D drawing in Volume 1 of the Subject guide.

The OpenGL interface was developed by Silicon Graphics Ltd (SGI) as an open specification for software to use 3D graphics hardware. It has since been widely adopted in 3D games, flight simulators, virtual reality and many other areas. There are competing proprietary specifications for using graphics hardware, but these are simply alternate ways of expressing the same concepts. So we shall focus only on the OpenGL implementation in *Processing*.

2.2 3D coordinate system

The three-dimensional coordinate system introduces a depth variable so that all points are of the form `(x,y,z)`, with `x` width, `y` height and `z` depth.

2.2.1 The screen (viewport)

Figure 2.2 shows the position of the 2D *Processing* screen positioned in the 3D coordinate system. The viewer is looking into the `x-y` plane. In 2D drawing, objects are placed on this plane. We call this the viewing rectangle or *viewport*.

In 3D drawing, however, we can place the viewport at any position and orientation. This gives us the ability to see the same scene from many different viewports. We use the concept of a `camera` to define the position and orientation from which a scene is viewed. In Section 2.4.4 we shall explore how to control the viewport by changing the location and orientation of the camera in 3D.

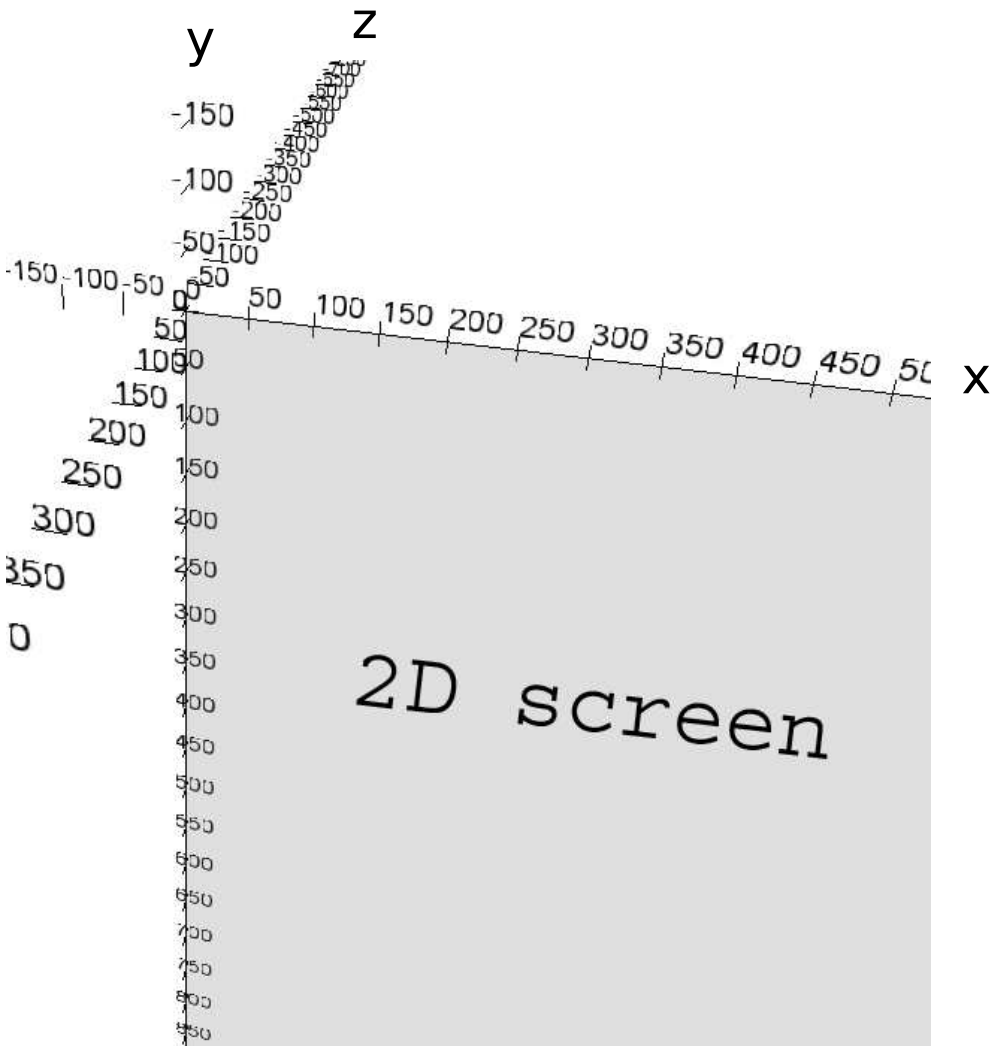


Figure 2.2: The 2D screen shown in a 3D space showing it to be the lower quadrant of the x–y plane.

2.2.2 3D rendering

Objects that we see in a 3D scene are projected onto the 2D viewport by a process called *rendering*. The rendering algorithm calculates which points from the 3D scene are visible through the current viewport; only those points are rendered.

3D rendering is the mechanism that is used to place objects from a 3D scene in perspective. That is, objects that are far away appear smaller compared to similar-sized objects that are close to the viewport.

In 3D perspective, parallel lines appear to meet at a point at infinity. Hence parallel lines are drawn so that they converge. The mathematics of projective geometry, which is used in the mathematics of perspective, makes a postulate that all parallel lines appear to meet at a point at infinity. This formal theory is due to the 17th-century mathematician Desargues and is the basis for 3D graphics.

Learning activity

Do a little research and reading (using the web or the additional reading list) and answer the following:

What are Euclid's postulates?

How is projective geometry different from Euclidean geometry?

2.2.3 3D lines

Previously we explored 2D graphics using methods such as `line(x1, y1, x2, y2)`. To use `line()` in 3D we add a third dimension to each point: `line(x1, y1, z1, x2, y2, z2)`. There are still two points defined, but they are located in a three dimensional coordinate system.

Figure 2.3 shows how we can use the 3D coordinate system to construct a simple scene. The ground plane extends away from the viewer in the `-z` direction. The origin remains in the top left corner as can be seen by the sphere, centred at the origin `(0,0,0)`, partially off the screen from the viewing point. This line of view is parallel to the `z`-axis, and the `x` and `y` coordinates retain their meaning as left-right up-down dimensions respectively.

The 3D coordinate grid is constructed using a `for` loop. Each pair of calls to the `line` method constructs one of the plane grids. In two dimensions there was one coordinate grid. However, in three dimensions there is a coordinate cube. Here we have drawn three faces of the coordinate cube to illustrate the relationships of the dimensions. There are three planes in the coordinate cube, the `x-y` plane, the `x-z` plane and the `y-z` plane. Can you identify each of these planes in the sketch shown in Figure 2.3?

There is a pattern to the grid drawing technique shown in this example. The first pair of lines draws the ground plane; note that the `x` and `y` variables are `>= 0` and the `z` coordinate is `<= 0`. We shall use the convention that positive depth is toward the viewer and negative depth extends away from the viewer.

```
// Simple 3D drawing example
int SZ = 512;
// select display size and use the P3D
// renderer for hardware acceleration
// with OpenGL
size(SZ, SZ, P3D);
background(255);
noFill();
// draw some grid lines
int step = 64;
for (int k=0 ; k<=SZ; k+=step) {
  line(k, SZ,   0,  k, SZ, -SZ);
  line(0, SZ,  -k, SZ, SZ, -k );
  line(0,  k, -SZ, SZ,  k, -SZ);
  line(k, SZ, -SZ,  k,  0, -SZ);
  line(0, SZ,  -k,  0,  0, -k );
  line(0,  k,   0,  0,  k, -SZ);
}
// draw a sphere (will be placed at 0,0,0)
sphereDetail(20);
sphere(128);
```

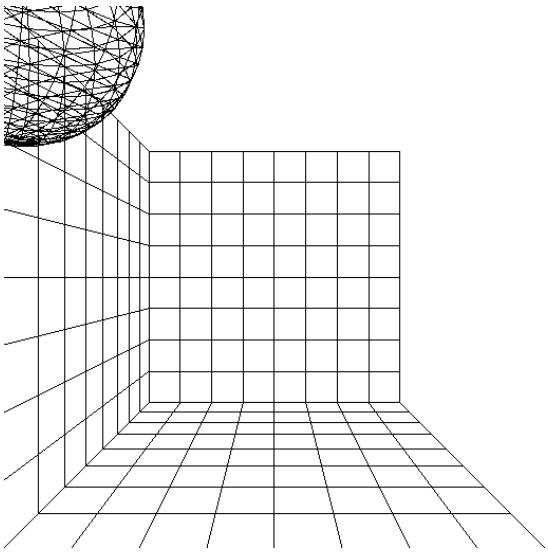


Figure 2.3: The three dimensional coordinate system adds a third component to each point, a depth dimension.

2.2.4 Using OpenGL

By specifying `P3D` as the third parameter to the `size()` method at the top of the sketch, we are telling *Processing* to use its 3D graphics renderer that makes use of OpenGL classes and methods. By using OpenGL routines, *Processing* binds our graphics methods to the specialised hardware on our computers that handles 3D graphics. Almost all modern desktop and laptop computers are shipped with some form of specialised graphics hardware. This hardware accelerates 3D operation, and is accessed via programs using one of several application programming interfaces (APIs). OpenGL is an API specification, as are Microsoft's Direct3D, Oracle's JavaFX 3D graphics API and the WebGL API for Javascript. If we use *Processing*'s 3D renderer on a system that does not support hardware accelerated graphics, it will revert to using the default software-only methods which generally run more slowly than the hardware accelerated versions.

Learning activity

Alter the sketch in Figure 2.3 to put the ceiling grid and right-hand wall grid in the scene. You can do this by adding only four calls to `line()`.

2.3 3D drawing

2.3.1 3D primitives

In *Processing*, the two 3D primitive shapes (primitives) are the `sphere()` and the `box()` (which is a cube). Figure 2.3 features a wire-frame sphere that is drawn using the `sphere()` method. Primitive shapes are drawn using `sphere(radius)` for spheres and `box(size)` for cubes.

There are no coordinate arguments for these 3D primitives. Spheres and boxes are always drawn at the origin. At first this may seem like very limited interface, allowing objects to be drawn only at the origin. There is a very good reason for this that we will introduce in Section 2.4.

3D tessellation

In 2D drawing, seen previously, ellipses are constructed out of lines; the more lines we use, at closer spacing, the better the approximation. The `sphere()` and `box()` shape primitives construct a 3D surface by a process called `tessellation` using triangles. Clearly, a `box()` can be exactly represented by 12 triangles; two for each face of the cube. For spheres, however, there is no finite number of triangles that can make a sphere. We must approximate it using a small number of triangles; the more triangles we use in the tessellation, the greater the level of detail of the approximation.

We can control the level of detail for `Processing`'s tessellation of spheres using the `sphereDetail()` method. The integer argument controls the number of faces used to approximate the sphere. The more faces we use, the longer the tessellation takes and the slower our sketch will run.

2.3.2 `beginShape()`, `endShape()`

We can draw any shape (either two or three dimensional) using the `beginShape()`, `endShape()` and `vertex()` methods.

Figure 2.4 shows one technique for drawing 2D polygons in a 3D space. Here we see the familiar polygons of a rectangle and triangle drawn on the `y-z` plane. Notice that although we specify that the `x`-coordinate is `0` for each vertex, the `x`-position is not always at the left hand edge of the screen. This is because the depth dimension moves the viewer to a position that can be distant from the drawing plane. It is only when the viewer is at zero distance from the drawing plane that the `x`-coordinate's `0` coincides with the screen left edge.

The `vertex()` method takes two or three arguments that are the coordinates of the point at which the vertex will be placed. The order of vertices is unimportant as the shape tessellation algorithm sorts the vertices into its own internal ordering before rendering the shape.

```
// Using beginShape(), vertex() and
// endShape() in 3D
int SZ=512;
size(SZ, SZ, P3D); // use 3D renderer
background(200);   // light grey background
// draw a square
fill(255);         // white fill
beginShape();
vertex(0,  0,   0);
vertex(0, SZ,   0);
vertex(0, SZ, -SZ);
vertex(0,  0, -SZ);
endShape(CLOSE);
// draw a triangle
fill(80);          // dark grey fill
float B=SZ/2;
float A=sqrt(3)*B/2;
beginShape();
vertex(0, SZ/2 - 2*A/3, -B);
vertex(0, SZ/2 + A/3, -SZ/2-B/2);
vertex(0, SZ/2 + A/3, -SZ/2+B/2);
endShape(CLOSE);
```

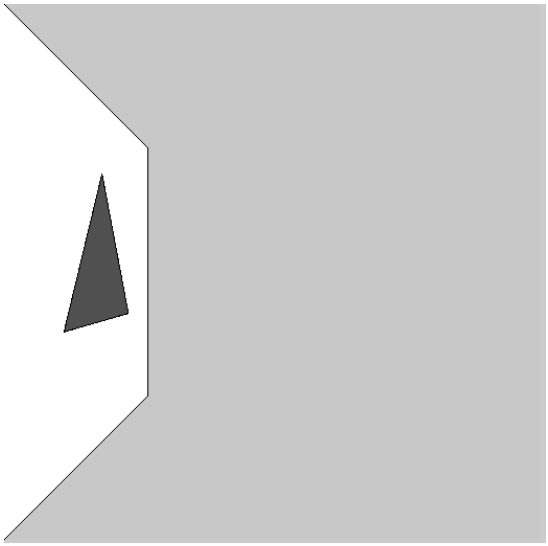


Figure 2.4: 2D polygonal shapes placed in a 3D scene using `beginShape()`, `endShape()` and `vertex()`. This illustrates perspective drawing.

```
// Perspective drawing in 3D
// Two rectangles extending backwards
// in z plane
int SZ=512;
size(SZ, SZ, P3D);
background(200);  // light grey back
// rectangle 1
fill(255);        // white fill
beginShape();
vertex(SZ/2,     0, 0);
vertex(SZ/2,    SZ, 0);
vertex(-2.5*SZ, SZ, -5*SZ);
vertex(-2.5*SZ,  0, -5*SZ);
endShape(CLOSE);
// rectangle 2
fill(100);        // mid-grey fill
beginShape();
vertex(SZ/2,  0, 0);
vertex(SZ/2, SZ, 0);
vertex(2*SZ, SZ, -5*SZ);
vertex(2*SZ,  0, -5*SZ);
endShape(CLOSE);
```

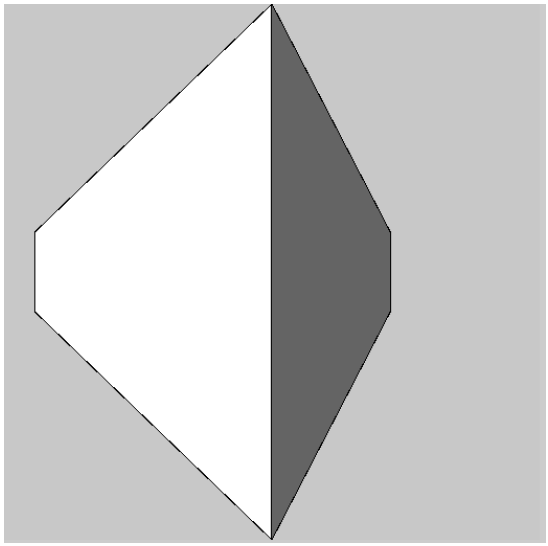


Figure 2.5: A perspective drawing using two rectangles drawn in three dimensions. Note that the parallel lines appear to get closer together as they extend into the distance. This drawing is an example of two-point perspective; to the viewer, it can appear that there are two walls, at a right angle (orthogonal), extending into the distance.

The `beginShape()` method takes one optional argument, the tessellation type. We can specify any one of: `POINTS, LINES, TRIANGLES, TRIANGLE_FAN, TRIANGLE_STRIP, QUADS, QUAD_STRIP`. Each of these tessellation types uses a different algorithm to draw the shape by filling in the faces between the given vertices.

See the *Processing* reference for examples of different tessellations for the `beginShape()` and `endShape()` drawing methods.

2.3.3 Perspective

In 3D perspective drawing, which OpenGL and other interfaces implement, parallel lines seem to meet at a point at infinity. This is demonstrated in Figure 2.5.

2.4 3D transformations

It is natural to think of drawing objects at specific locations by providing the coordinates for their positions. However, an alternate way to think of the drawing process is to transform the origin to the location we desire, then draw the object at the (new) origin. Then move the origin back to its original position and repeat the transformation, draw, undo transformation process for each object.[5] For example, to draw two rectangles we might use point coordinates to locate the objects:

```
int SZ = 512;
size(SZ, SZ);
rect(9, 9, 246, 246);
rect(265, 265, 246, 246);
```

Or instead, we can transform the coordinate system between each call to `rect()`.

```
int SZ = 512;
size(SZ, SZ);
translate(9,9);        // new coordinates #1
rect(0, 0, 246, 246);
translate(-9,-9);      // undo translate  #1
translate(265, 265);   // new coordinates #2
rect(0, 0, 246, 246);
translate(-265, -265); // undo translate  #2
```

Now the two calls to `rect()` have exactly the same form, which greatly simplifies the object drawing code. When we come to draw much more complicated objects with multiple lines of drawing code, we may wish to place them in their own method. If we do this, then we can simply transform the coordinate system to the desired location and then call the drawing method with no arguments:

[5] We have already come across a similar method when we looked at *Turtle Graphics* in Chapter 6 of Volume 1. However, in that case we did not perform the undo transformation after each draw.

```
int SZ = 512;
void setup() {
   size(SZ, SZ);
}
void draw(){
  translate(9, 9);
  myShape();
  translate(-9, -9);
  translate(73, 73);
  myShape();
  translate(-73, -73);
}
void myShape(){
  rect(0, 0, 246, 246);
  rect(246, 246, 100, 100);
}
```

The method of transforming the coordinate system is extremely powerful and is the basis for the OpenGL way of doing 3D graphics. Using this method, we can draw objects on transformed planes so that they appear as drawing in a 3D perspective view.

There are nine coordinate-system transformations available. The first is translation, the second is scaling, and the third is rotation: each has a transformation in relation to each axis `x`, `y`, `z`, making a total of nine transformation primitives.

The `translate(x,y,z)` and `scale(x,y,z)` methods each take three arguments and operate on all three dimensions at the same time; this is possible because the operations in each dimension are independent of each other. For example, no matter how we translate `x`, the translation of `y` and `z` are not affected. Similarly, scaling the `z` dimensions does not affect scaling in the `y` and `x` dimensions. However, rotation has dependencies between the axes. Rotating the three axes in a different order changes the final result. For example `(rotateY(PI/4); rotateX(PI/3);` is not the same as `rotateX(PI/3); rotateY(PI/4)`. Hence the rotation methods are separated into their three axes. So there are a total of five method calls for transformations: `translate(x,y,z)`, `scale(x,y,z)`, `rotateX(angle)`, `rotateY(angle)` and `rotateZ(angle)`. The next sections describe the use of these five transformation methods.

2.4.1 `translate(x,y,z)`

To place an object in a 3D scene we use the `translate()` method. The axes that we are translating are those shown in Figure 2.6. We move the coordinate system by shifting it along the following directions: left/right, up/down and forwards/backwards. These directions correspond to: `-x/+x, -y/+y, +z/-z` respectively.

Figure 2.7 shows the placement of two objects in a 3D scene using coordinate system transformations in `(x, y, z)`.

The full sketch to produce this output is shown in Figure 2.8. It is rather lengthy, because it draws the labelled axes as well as the objects. A simplified snippet to just draw the two objects might be as follows:

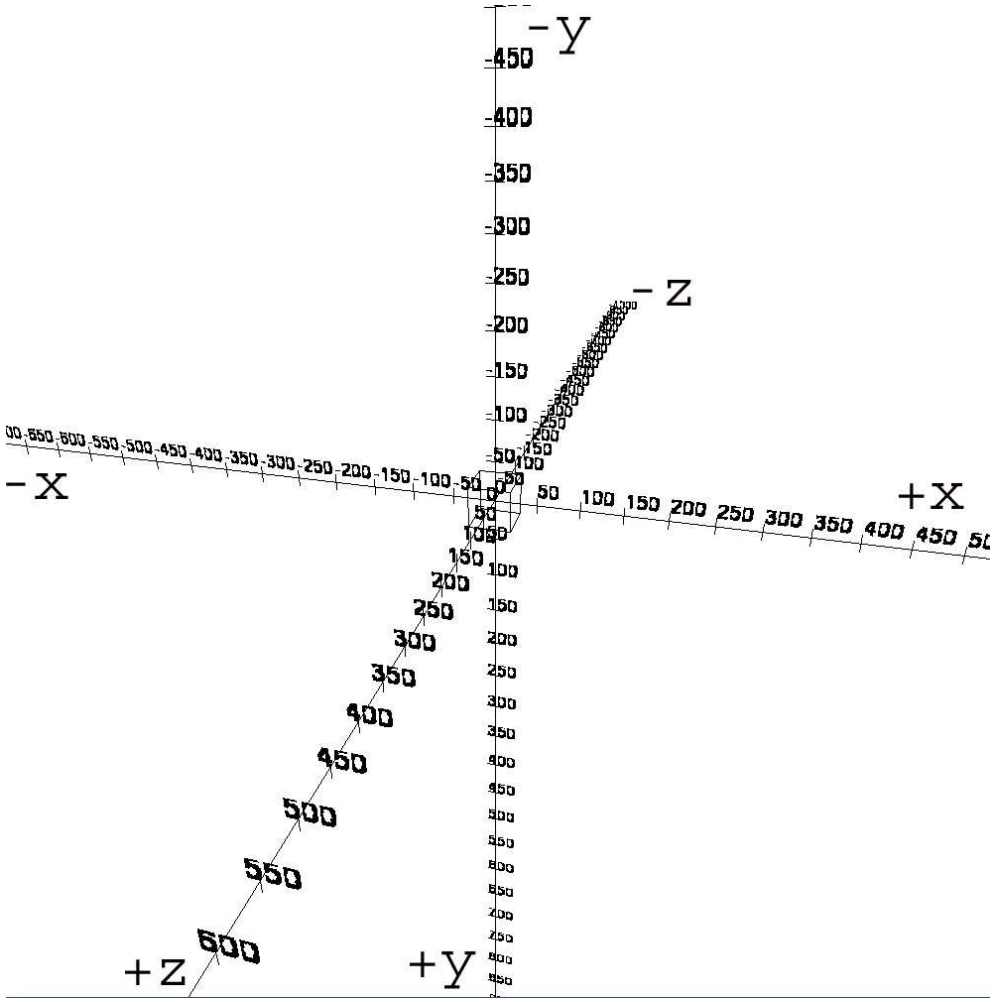


Figure 2.6: The three dimensional coordinate system as viewed from off-axis. We can see both positive and negative positions in all axes. The view of the coordinate system depends on its position and rotation, and the viewer's position and rotation.

```
float x,y,z;
x=-400; y=200; z=200;
noFill();
translate(x, y, z);
sphereDetail(9);
stroke(0, 255);
sphere(100);
translate(-x, -y, -z);  // inverse transform
x=400; y=-200; z=-300;
translate(x, y, z);     // new transform
box(100);
translate(-x, -y, -z);  // inverse transform
```

Looking at the code above, notice that the `box()` is in a final position of `(400, -200, -300)`. To get it there we had to first undo the transform of the sphere by `translate(-x, -y, -z)`. This type of transform is called an inverse transform because it restores the origin to its original "zero" position; it inverts the coordinate transform. Once the origin is back at `(0, 0, 0)`, we can now specify the new coordinate for the box and move the coordinate system to the new position using `translate(x, y, z)`.

In the sketch shown in Figure 2.8, we actually use `pushMatrix()` and `popMatrix()` to achieve the same result as using inverse transformations. We are getting a little bit ahead of ourselves here; we briefly came across these methods in Chapter 8 of Volume 1 of the Subject guide, but we will discuss them in more detail in Section 2.4.5 later in this chapter.

By moving the coordinate system around we are able to draw the primitive shapes without explicit coordinates. The shapes are always drawn at the local position `(0, 0, 0)`.

Figure 2.9 shows how a series of transformations can be driven by variables in a `for` loop. Three nested `for` loops are used to iterate over the three axis variables `(i, j, k)` for the three dimensions `(x, y, z)` respectively.

Each of the `for` loops controls a single dimension. The order of drawing is not important for multiple axes because they are linearly independent. Try and switch two `for` loops: do you get the same answer? Is this the case for all three types of transformation?

2.4.2 `scale(sx,sy,sz)`

The scaling transformation works by stretching each axis independently from the other axes. The scale factor is a floating point value.

```
scale(0.5, 1, 1);     // Scale x-axis by 0.5
scale(0.5, 0.5, 1);   // Scale x- and y-axis by 0.5
scale(0.5, 0.5, 2.0); // Scale x- and y-axis by 0.5, z-axis by 2.0
```

A scale factor of 2 makes everything in the direction of the scaled axis two times larger. The remaining axis directions remain unaffected. This technique allows us to stretch or shrink the appearance in any direction. Figure 2.10 shows the effect of scaling the `x` and `y` axes by 0.5 and the `z` axis by 2.0.

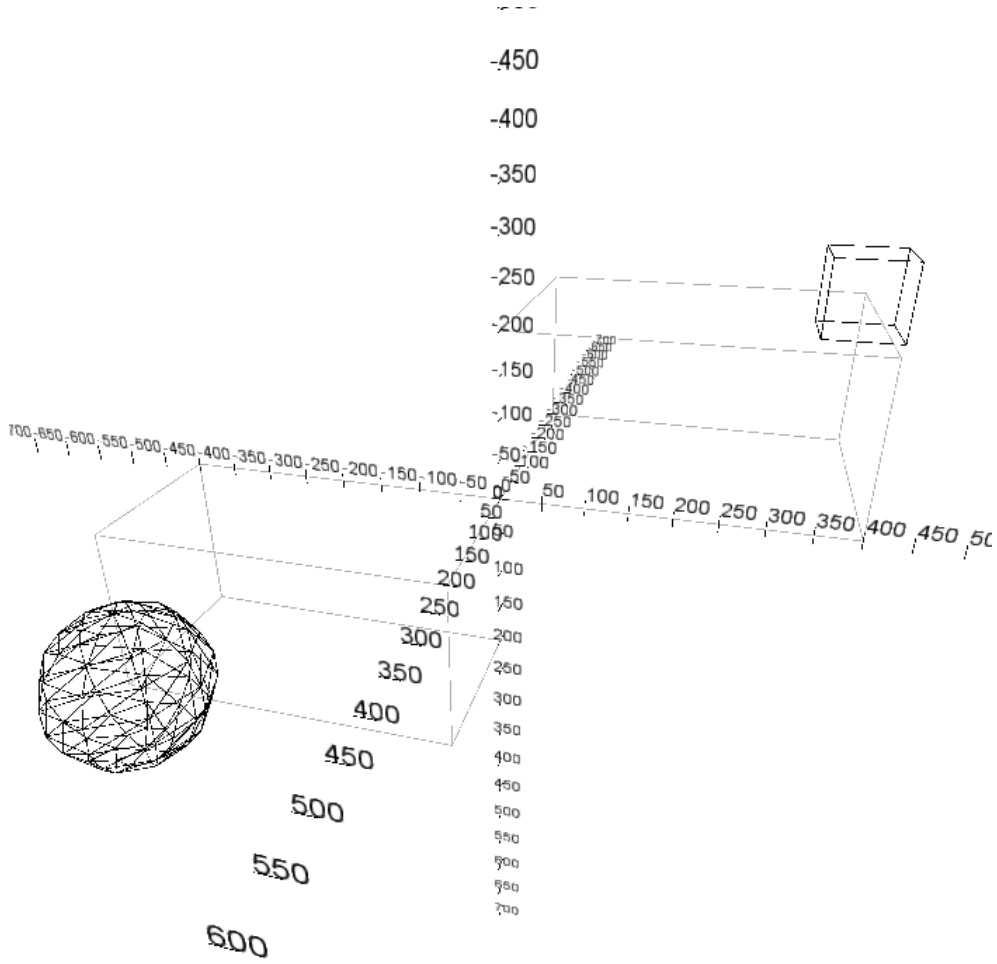


Figure 2.7: Placing objects in a 3D scene requires specifying their coordinate system transformations by translation in each of the three axes. Here a sphere is placed at position `(-400, 200, 200)` and a box at position `(400, -200, -300)`.

```
PFont f;
int SZ=700;

void setup() {
  size(SZ,SZ,P3D);
  f = loadFont("ArialMT-24.vlw");
  background(255);
  stroke(0);
  fill(0);
  camera(0,0,SZ,0,0,-1,0,1,0);
  textFont(f);
  noLoop();
}

void draw() {
  float x, y, z;
  scale(0.7);
  rotateX(-PI/7);
  rotateY(-PI/12);
  drawAxes();
  // draw sphere
  x=-400; y=200; z=200;
  noFill();
  pushMatrix();
  translate(x,y,z);
  sphereDetail(9);
  stroke(0,255);
  sphere(100);
  popMatrix();
  drawOriginBox(x,y,z);
  // draw box
  x=400; y=-200; z=-300;
  pushMatrix();
  translate(x,y,z);
  stroke(0,255);
  box(100);
  popMatrix();
  drawOriginBox(x,y,z);
}

void drawOriginBox(float x, float y, float z) {
  stroke(63,63);
  pushMatrix();
  translate(x/2.0, y/2.0, z/2.0);
  box(x,y,z);
  popMatrix();
}

void drawAxes() {
  float a=10;
  // X axis
  for(int x=-SZ; x<=SZ; x+=50) {
    pushMatrix();
    translate(x, 0, 0);
    line(0,-a,0,a); text(""+x,0,-a,0);
    popMatrix();
  }
  // Y axis
  for(int y=-SZ; y<=SZ; y+=50) {
    pushMatrix();
    translate(0, y, 0);
    line(0,0,-a,0,0,a); text(""+y,-a,0,0);
    popMatrix();
  }
  // Z axis
  for(int z=-SZ; z<=SZ; z+=50) {
    pushMatrix();
    translate(0, 0, z);
    line(-a,0,0,a,0,0); text(""+z,-a,0,0);
    popMatrix();
  }
}
```

Figure 2.8: Sketch to produce the scene shown in Figure 2.7.

```
int SZ=512;
int boxVol=2*SZ/3;
int nBoxes=8;
int boxSZ=boxVol/nBoxes;

void setup() {
  size(SZ,SZ,P3D);
  noFill();
  noLoop();
}

void draw() {
  background(255);
  translate( SZ/4, 0, -SZ/4 ); // Position of drawing
  for( int i=0; i < boxVol; i += boxSZ ) {
    for( int j=0; j < boxVol; j += boxSZ ) {
      for( int k=0; k > -boxVol; k -= boxSZ ) {
        translate( i, j, k );
        box( boxSZ );
        translate( -i, -j, -k );
      }
    }
  }
}
```

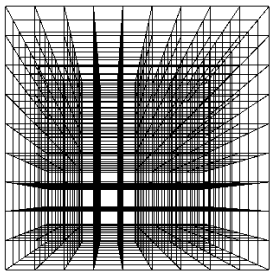


Figure 2.9: Using a `for` loop to place boxes in a larger cube.

2.4.3 `rotateZ()`, `rotateY()`, `rotateX()`

The final set of transformations are rotations. As we saw above, translation and scaling transformations operate on each axis indepently of each other axis. This is not the case with rotations, where the order of the axes determines the final rotation.

`rotateZ()`

We have already used, extensively, rotation around the `z`-axis because this is the type of rotation used in a 2D sketch. The `z`-axis is the depth dimension which protrudes from the screen out, conceptually. Therefore, `rotateZ()` behaves in the same way as `rotate()` in two dimensions (provided the camera is looking on-axis into the `z` dimension: see Section 2.4.4).

`rotateY()`

Figure 2.11 illustrates how we can use the standard 2D shape methods to construct a 3D scene. The method is to transform the coordinate system; in this case the entire coordinate system is rotated around the left hand edge of the screen, inwards.

We achieve the rotation around the `y`-axis using the `rotateY()` method. This method accepts one argument, which is the rotation angle in radians. A positive value rotates the coordinate system inwards. Now, all `x` coordinates extend into the scene with depth.

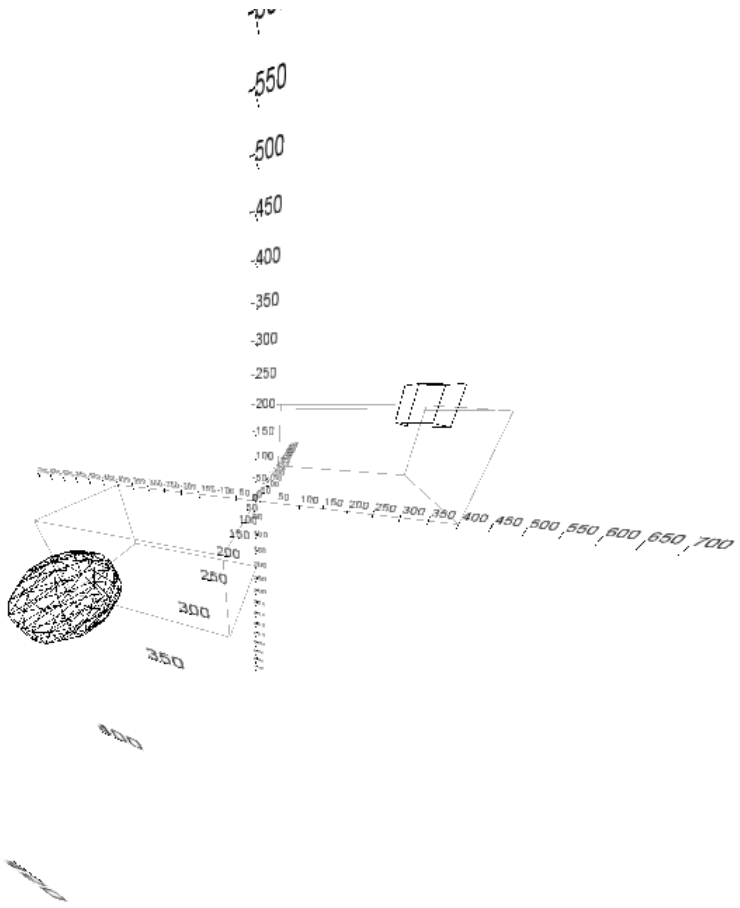


Figure 2.10: Axes scaling. The scene from Figure 2.7 is reproduced here with the `x` and `y` axes scaled by 0.5 (compressed) and the `z` axis scaled by 2.0 (stretched).

```
// Perspective and vanishing points with 2D drawing
// and rotateY

size(512,512,P3D);
float faraway = 1000000;

// We need a little bit of magic here to make
// Processing display much further into the distance
// than it normally would, so we can see the
// vanishing point.
// To understand the next two lines, look at
// http://processing.org/reference/perspective_.html
float cameraZ = ((height/2.0)/tan(PI*60.0/360.0));
perspective(PI/3.0, width/height,cameraZ/10.0,faraway);

// rotate the whole scene around Y axis
rotateY(PI/3);
//rotateY(1.4);
noStroke();
background(255);
fill(0);
// now draw some (2D) shapes on the X-Y plane
ellipse(256,256,512,512);        // a circle
rect(512,0,512,512);             // some squares
rect(1024,512,512,512);
rect(1536,0,512,512);
rect(2048,512,512,512);
rect(2560,0,512,512);
rect(3072,512,512,512);
stroke(0);
line(0,0,faraway,0);             // some lines
line(0,height,faraway,height);
line(0,2*height,faraway,2*height);
```

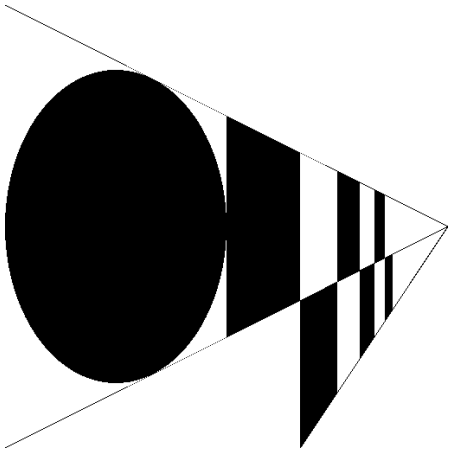


Figure 2.11: In projective geometry, parallel lines appear to meet at a point at infinity. This point is called the vanishing point. In this example there is one such point, at the right-hand edge of the screen.

`rotateX()`

In addition to rotating around the vertical axis and the depth axis, we may also rotate around the horizontal `x` axis. Rotations around this axis incline the ground plane either towards or away from the viewer.

```
int SZ=512;
int boxVol=2*SZ/3;
int nBoxes=8;
int boxSZ=boxVol/nBoxes;
float ang1=PI/(4/sqrt(2));
float ang2=PI/(4/((1+sqrt(5))/2));

void setup() {
  size(SZ,SZ,P3D);
  noFill();
  noLoop();
}

void draw() {
  background(255);
  translate( SZ/4, 0, -SZ/4 ); // Position of drawing
  for( int i=0; i < boxVol; i += boxSZ ) {
    for( int j=0; j < boxVol; j += boxSZ ) {
      for( int k=0; k > -boxVol; k -= boxSZ ) {
        rotateY(ang1);
        rotateX(ang2);
        translate( i, j, k );
        box( boxSZ );
        translate( -i, -j, -k );
        rotateX(-ang2);
        rotateY(-ang1);
      }
    }
  }
}
```

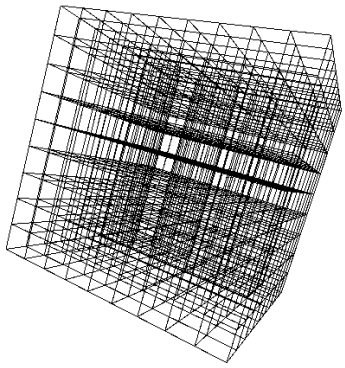


Figure 2.12: Using `rotateY() and rotateX()` to change the view of the cube.

To understand what is happening more generally, imagine drawing a large rectangle on the inside of a closed window. Now imagine opening the window outwards. If the window is hinged at the left edge, this is the same as rotating the coordinate system around the `y`-axis. If the window is hinged at the top, this is the same as rotating the coordinate system around the `x`-axis. Rotation around the `z`-axis is the same as rotation in 2D: the coordinate system is rotated around the origin like rotating a painting on a wall fixed at its top-left corner.

It is also possible to combine rotation of axes. Figure 2.12 shows an extended version of the sketch shown in Figure 2.9 with the additional use of `rotateY` followed by `rotateX` transformations. After the two rotations, a translation is applied to correctly position each of the small boxes, just like it was in Figure 2.9.

2.4.4 Camera transformations

As well as placement of drawing elements in a 3D scene, the viewing position is a critical factor in a 3D design. The *Processing* `camera()` method moves the camera to a specified position in 3D space. That is the position from which the scene will be viewed. For example, we may be viewing a concert stage from different locations in

the concert hall, or watching a football match from different parts of the stadium. A change in viewing position is a camera transformation.

To position a camera in 3D space we need to decide the camera's translation coordinates `Cx, Cy, Cz` and the point to which the camera is facing, the *lookat* point, `Lx, Ly, Lz`—*the camera automatically faces the specified lookat point*. Finally, the `camera` method requires three arguments to indicate the orientation of the vertical axis; that is, which way is up. To indicate that the `y` axis is the vertical axis and points down, use `(0,1,0)`.

For example, to place the camera at the origin facing in the negative `z` direction, we use:

```
camera(0, 0, 0, 0, 0, -1, 0, 1, 0);
```

It is often more convenient to place the camera at a distance from the origin, so that we are looking facing forwards at the scene which is drawn at the origin. To do this we need to translate the camera in the `z` dimension, away from the screen, which is in the positive `z` direction.

In Section 3.2.1 we will look at additional techniques that provide further control of the camera.

2.4.5 `pushMatrix(), popMatrix()`

In the previous examples of using transformations, we have been careful to undo all our coordinate system transformations using the inverse transform after each object is drawn. We did this to preserve the global coordinate system, so that we have a coordinate system to place objects in relation to each other.

We can rewrite the 3D scene sketch from Figure 2.12 so that there are far fewer inverse transforms to perform. Note that in the previous example, we translated and rotated every box into its final position starting from the origin in the world coordinate system. The world coordinate system was preserved by undoing the rotations and translations after every placement of a box.

Figure 2.13 shows the construction of the same 3D scene but we have *factored* the transformations. Factoring uses algebraic manipulation of the transformations: $Ax + Ay = A(x + y)$.

In Figure 2.12, each transformation is performing the same sequence of operations except for an increment in one of the `x,y,z` dimensions. In contrast, in Figure 2.13 the world coordinate transformation that is global to the scene is factored out. The global transformation is the translation of the large cube to its final position, and the rotation of the scene around the `y`-axis and `x`-axis. Now we perform a global rotation of the world coordinate system, followed by incremental transformations along each axis. Each box is placed next to the last one. To do this, we have to be careful about the order in which we place the boxes: we draw one column at a time and then loop back to draw the next column in the opposite direction. This is achieved by the calls to `pow(-1,n)` in the code, which switch the direction between positive and negative directions each time `n` is increased.

The sketch in Figure 2.13 has successfully factored out the global transformations in the scene. However, it has done so at a heavy price to the readability of the sketch: it is fairly hard to understand what the sketch is doing at a casual glance!

```
int SZ=512;
int boxVol=2*SZ/3;
int nBoxes=8;
int boxSZ=boxVol/nBoxes;
float ang1=PI/(4/sqrt(2));
float ang2=PI/(4/((1+sqrt(5))/2));

void setup() {
  size(SZ,SZ,P3D);
  noFill();
  noLoop();
}

void draw() {
  background(255);

  // First perform global transformations on the scene
  translate(SZ/4, 0, -SZ/4); // Position of drawing
  rotateY(ang1);             // Factored out rotateY
  rotateX(ang2);             // Factored out rotateX

  // Loop over all box positions along the X axis
  for (int i=0; i < nBoxes; i++) {
    translate(boxSZ, 0, 0);
    // Now draw all boxes in the Y-Z plane for the
    // current X position
    for (int j=0; j < nBoxes; j++) {
      if (j>0) {
        // switch direction of drawing for each
        // successive move along the X axis
        translate(0, pow(-1,i)*boxSZ, 0);
      }
      // Draw all boxes in the Z column for the current
      // X-Y position
      for (int k=0; k < nBoxes; k++) {
        if (k>0) {
          // switch direction of drawing for each
          // successive move along the Y axis
          translate(0, 0, pow(-1,j+1)*boxSZ);
        }
        box( boxSZ );
      }
    }
  }
}
```

Figure 2.13: Construction of cube of boxes by factoring the transformations. The translations are performed independently in different levels of the nested loops. Construction is performed in an order that does not require inverse transforms. The global coordinate system is transformed, before the loop structures, to the final position of the cube.

Fortunately, there is another technique to manage transformations of the coordinate system, that is in general easier to use and that produces code that is easier to read. The technique makes use of two of *Processing*'s data structures: the *transformation matrix* and the *stack*. A stack is a special data structure with two methods: `push()` and `pop()`. The `push()` method places an object on the stack, which is an ordered list like the stack of messages in your e-mail inbox. However, stacks must be read in reverse order of arrival. In other words, the last item placed onto the stack will be the first item taken off it.

We use a stack to store the current coordinate system before transforming it. This is a `push()` operation. Then after any transformations, we restore the coordinate system to its state before the transformation using the `pop()` operation. *Processing* provides two methods called `pushMatrix()` and `popMatrix()` to store and restore the current coordinate system. A matrix is simply a transformation, so we are storing and restoring transformation matrices of the coordinate system when we use the coordinate system stack.

We can nest `pushMatrix()` operations, as long as there are as many calls to `popMatrix()` as `pushMatrix()`. *Processing* imposes a limit of 32 calls to `pushMatrix()` before a `popMatrix()` is required. Figure 2.14 shows the use of `pushMatrix()` and `popMatrix()` to save and restore coordinate transformations.

We shall use the coordinate system stack extensively when making our 3D sketches.

2.4.6 Lights

The sketch in Figure 2.14 also illustrates the use of lights in a 3D scene. The use of lighting can enhance the 3D effect. We can light the entire scene using ambient or directional light. In this example we have used a directional light and have lit the scene from the front. Note that the light is specified on the same coordinate system as the scene. The `directionalLight()` method tells *Processing* to add light from the specified direction.

Processing provides a variety of lighting methods, in addition to `directionalLight()`, to achieve different effects. These include `ambientLight()`, `spotLight()`, and the general `lights()` method. Further details can be found in the online *Processing* reference manual.

When rendering with lights, an algorithm calculates which surfaces in the 3D scene are facing in the direction of any light in the scene, and then colours the scene based on the colours of these lights and the amount of light based on the surfaces' angle to the light source.

In Figure 2.14 we see that not all parts of the spheres are coloured the same. Those parts of the spheres that are facing away from the light source are coloured dark gray, and those that face the light source are coloured white.

2.5 Texture mapping

In the previous examples we have used 3D primitives to construct a surface in 3D space. The appearance of the surface was either wireframe or filled. If wireframe it was transparent, showing us only the strokes of the tessellation. If filled, the

```
// Example of pushMatrix(), popMatrix() and directionalLight()

int SZ=512;
float boxVol=SZ;
float nBoxes=8.0;
float boxSZ=boxVol/nBoxes;
float ang1=PI/(4/((1+sqrt(5))/2));
float i, j, k;

void setup() {
  size(SZ,SZ,P3D);
  fill(200,200,255);
  noStroke();
  noLoop();
  sphereDetail(12);
}

void draw() {
  background(255);
  rotateY(ang1);
  translate(SZ/2,SZ/2,SZ/2);
  directionalLight(255,255,255,1,1,-1); // add some lighting
  for (j=0; j < nBoxes; j++) {
   for (i = SZ/2 - j*boxSZ/2; i <= SZ/2 + j*boxSZ/2; i+=boxSZ) {
     for (k = -SZ/2 + j*boxSZ/2; k >= -SZ/2 - j*boxSZ/2; k-=boxSZ) {
       // first save the current coordinate system before
       // doing any transformations
       pushMatrix();
       translate( i, j*sqrt(2)*boxSZ/2, k );
       sphere( boxSZ / 2 );
       // we do not need to perform an inverse transform
       // here, we simply restore the previous coordinate
       // system with the following call
       popMatrix();
     }
   }
  }
}
```

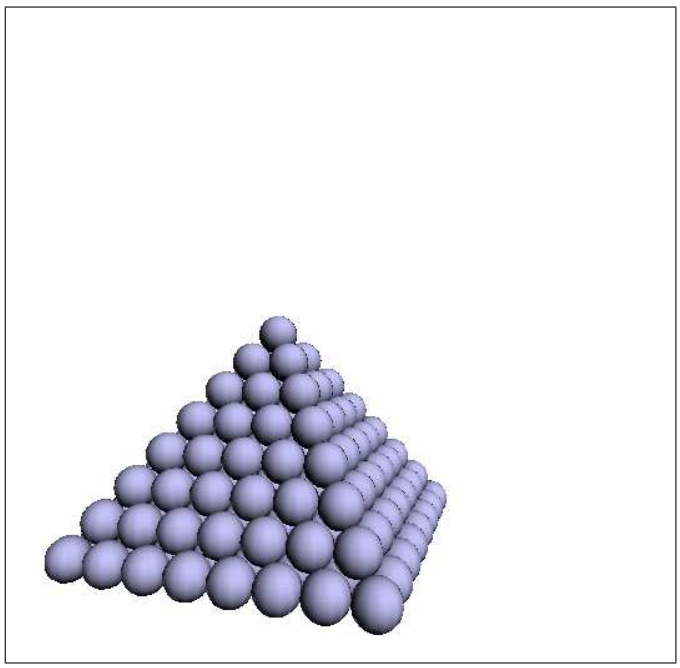


Figure 2.14: Using `pushMatrix()`, `popMatrix()` to preserve local coordinate systems and using `directionLight()` to light the scene from the front.

appearance was a solid colour, and if the scene was lit, the colour's lightness was proportional to the angle of rotation from the light source.

The process of *texture mapping* allows us to make the appearance of surfaces much more interesting and varied than filling with solid colour. Figure 2.15 and Figure 2.16 show scenes from the classic 3D games *Doom* for the PC, and *Super Mario Brothers 64* for the Nintendo 64. Here we see that the scenes are constructed from all kinds of details such as the appearance of the surfaces.

In Figure 2.15, both the 3D model of the monster and the walls, ceiling and floor surfaces have intricate details drawn on them. Similarly, in Figure 2.16, each surface is painted by its material appearance: bricks, wooden planks, roof slates, etc.

Figure 2.15: Texture mapping gives an appearance to surfaces. In this example from the classic game *Doom*, the walls and 3D monsters are texture mapped with intricate details.

The technique used to paint surfaces in this way is called texture mapping. We can use texture mapping by loading an image (such as a JPG, GIF or PNG file) using the `loadImage()` method. We then map the texture to a surface by using the `beginShape()` and `endShape()` methods along with a call to the `texture()` method and the texture-map version of the `vertex()` method. This version takes two extra arguments which are the points of the texture image to map to each vertex of the 3D shape.

The rendering algorithm will now perform a set of image processing operations on the texture to stretch it to fit the surface at the points specified. Figure 2.17 gives an example of texture mapping an image of a wall of bricks onto planar walls in a 3D scene. The scene is constructed by moving the coordinate system to various locations, rotating, and drawing a section of wall. This is an example of a modular construction of a 3D scene.

Figure 2.18 and Figure 2.19 illustrate how to map a larger image, in this case 800x400 pixels, onto surfaces. This shows that the position of the surface determines the appearance of the texture in the scene. In this case, we see the scene from *on-axis* so that we are facing the x-y plane.

Figure 2.16: The *Super Mario Brothers 64* game for the Nintendo 64 console, released 1996. The player navigates a 3D world of obstacles and puzzles. All the objects are texture mapped; the walls with bricks. Notice the non-alignment of textures between the top of the walls and the vertical sides.

```
// Simple demonstration of texture mapping

PImage bricks;
float a;

void setup() {
  size(512,512,P3D);
  bricks=loadImage("bricks.gif");
  noStroke();
  noLoop();
}

void draw() {
  background(255);
  translate(width,2*height,-3*width);
  rotateY(2*PI/9);
  maze();translate(width,0,0);
  maze();translate(width,0,0);
  maze();translate(0,0,-width);
  maze();translate(0,0,-width);
  maze();translate(0,0,-width);
  maze();translate(-width,0,0);
  maze();translate(-width,0,0);
  maze();translate(0,-2*height,4*width);
  rotateX(-PI/3);maze();
}

void maze() {
  // draw a cross structure with four walls
  wall(width/2);rotateY(-PI/2);
  wall(width/2);rotateY(-PI/2);
  wall(width/2);rotateY(-PI/2);
  wall(width/2);rotateY(-PI/2);
}

void wall(int L) {
  // draw an individual wall
  beginShape();
  texture(bricks);
  vertex(0,0,0,0);
  vertex(L,0,L,0);
  vertex(L,L,L,L);
  vertex(0,L,0,L);
  endShape();
}
```

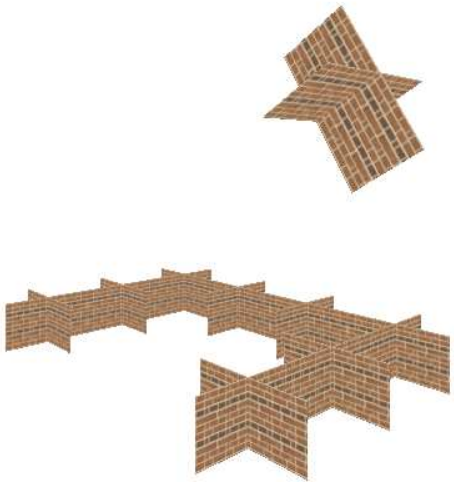


Figure 2.17: Texture mapping gives an appearance to surfaces. In this example, the walls appear to be composed of bricks.

Learning activity

Modify the code in Figure 2.18 so that you see the scene from the back right hand side of the scene. You should try positioning the camera at translation `(800, -800, 800)`. How many walls do you now see?

```
PImage Doom;
float a;

void setup() {
  size(800,400,P3D);
  Doom=loadImage("Doom.png");
  noStroke();
  noLoop();
}

void draw() {
  background(0);
  translate(0,0,-width);
  maze();translate(width,0,0);
  maze();translate(width,0,0);
  maze();translate(0,0,-width);
  maze();translate(0,0,-width);
  maze();translate(0,0,-width);
  maze();translate(-width,0,0);
  maze();translate(-width,0,0);
  maze();translate(0,-2*height,4*width);
  rotateX(-PI/3);maze();
}

void maze() {
  wall(width,height);rotateY(-PI/2);
  wall(width,height);rotateY(-PI/2);
  wall(width,height);rotateY(-PI/2);
  wall(width,height);rotateY(-PI/2);
}

void wall(int L, int M)  {
  beginShape();
  texture(Doom);
  vertex(0,0,0,0);
  vertex(L,0,L,0);
  vertex(L,M,L,M);
  vertex(0,M,0,M);
  endShape();
}
```

Figure 2.18: Source code for texture mapping the 800 pixels x 400 pixels *Doom* image.

2.5.1 Transparent textures

The use of transparency for texture mapping is a very useful technique to enhance the object-like quality of a texture-mapped surface. In Figure 2.20 (page 55) a texture is used that has a transparent background. The rendering algorithm in *Processing* is able to render the backgrounds as transparent so that we can see through the object at these points in the texture. Here we have mapped a complex 2D shape (a dune buggy) onto planes. The background of the dune buggy is transparent, so that we can see the dune buggies behind through those in the front. The output of the sketch is shown in Figure 2.21.

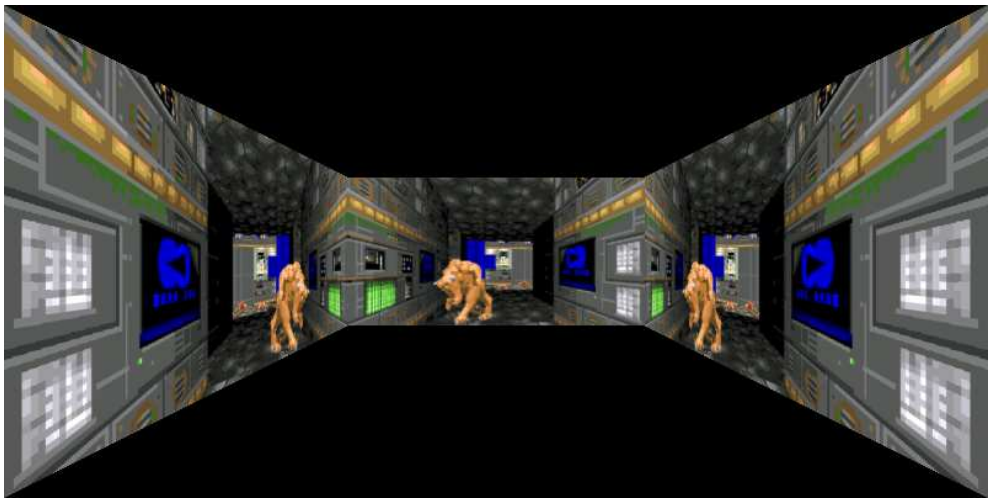


Figure 2.19: *Doom* image texture mapped onto three planar walls. The left and right walls are parallel; what we see is a perspective view.

Changing the viewing position of the scene changes the appearance of the textures also. Figure 2.22 (page 56) shows an alternate view of the previous sketch from a different camera position. The planes now do not overlap but the view is from a position above the scene and we see only a sliver of each object. That is because we constructed these as planar surfaces; they have no depth.

Learning activity

Modify the code for the scene in Figure 2.21 (page 56) so that a complete car is texture mapped. Find images that are of the left/right/front and back sides, roof and undercarriage of a car. Map these new views onto a 3D object that has six faces, one for each of left, right, top, bottom, front, back.

Make the windows of your car transparent by editing the image files to fill glass areas with transparency.

View your newly constructed car object from different viewing angles.

2.6 Algebra for perspective and affine transformations; point and line

The material below may already be known from another unit or from other study, in which case it may be used for revision or passed over as required.

2.6.1 Vectors and matrices; addition and multiplication

A **vector**, in general terms, can be thought of as a quantity with both magnitude and direction, or having as many components (of the same type) as there are dimensions of the space under consideration.

```
// Texture mapping using a texture with a transparent background

int distance=10000;
int L=200;
float ang=0.0;
PImage a;
int NUMCARS=1000;
float[] x;
float[] z;
int GH=200; // Ground height

void setup() {
  size(768,512,OPENGL);
  noStroke();
  // LOAD IMAGE
  a = loadImage("dunebuggy.gif");
  x = new float[NUMCARS];
  z = new float[NUMCARS];
  // Generate the CAR positions
  for(int k=0; k<NUMCARS; k++) {
    x[k]=random(2000)-1000; // left-right
    z[k]=2*(random(10000)-5000); // depth
  }
}

void draw() {
  background(250,250,255);
  if (mousePressed) { // mouse control
    float accel=100*(height/2-mouseY)/height;
    ang-=PI/32*(width/2-mouseX)/width;
    distance-=accel;
  }
  camera(width/2,GH-400,distance,width/2,GH-200,distance-50000,0,1,0);
  // Viewpoint rotation
  rotateY(ang);
  // Draw the ground plane
  beginShape(QUADS);
  int P=1000000; // Size of world
  fill(30,80,30); // Colour
  vertex(-P,GH,P); // Ground Plane
  vertex(+P,GH,P);
  vertex(+P,GH,-P);
  vertex(-P,GH,-P);
  endShape();

  noStroke();
  fill(255);

  // Car rendering
  for (int k=0;k<NUMCARS;k++) {
    pushMatrix();
    translate(x[k],0,z[k]);
    drawCAR();
    popMatrix();
  }
}

// Texture mapping the cars
void drawCAR() {
  for (int i=0; i<2; i++) {
    beginShape();
    rotateY(PI);
    texture(a);
    vertex(0, 0, 0, 0, 0);
    vertex(L/2, 0, 0, L/2, 0);
    vertex(L, 0, 0, L, 0);
    vertex(L, L, 0, L, L);
    vertex(0, L, 0, 0, L);
    endShape();
  }
}
```

Figure 2.20: Source code for texture mapping using a transparent background. The output of the sketch is shown in Figure 2.21.

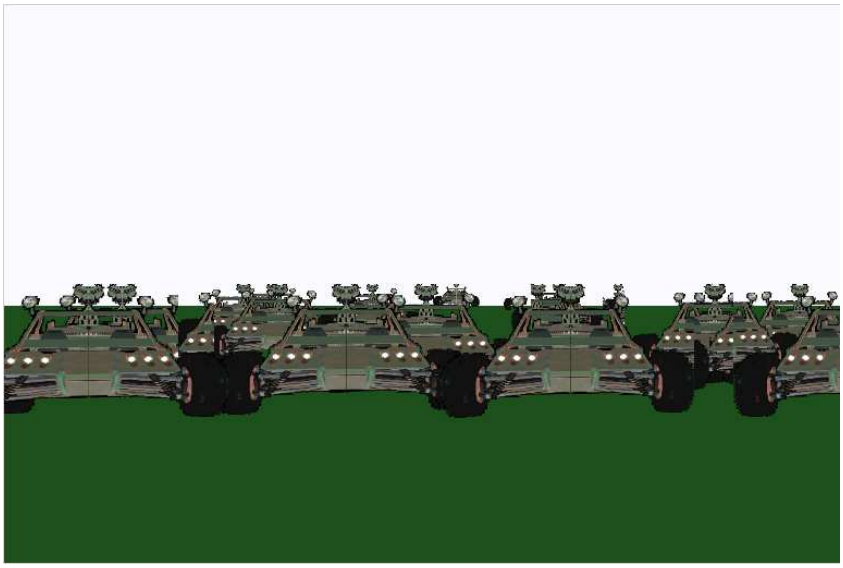


Figure 2.21: Texture mapping using a transparent background. We can see the dune buggies further away through the transparent parts of those that are closest. This is achieved using a texture map with transparency (alpha=0) on the background colour. Source code is shown in Figure 2.20.

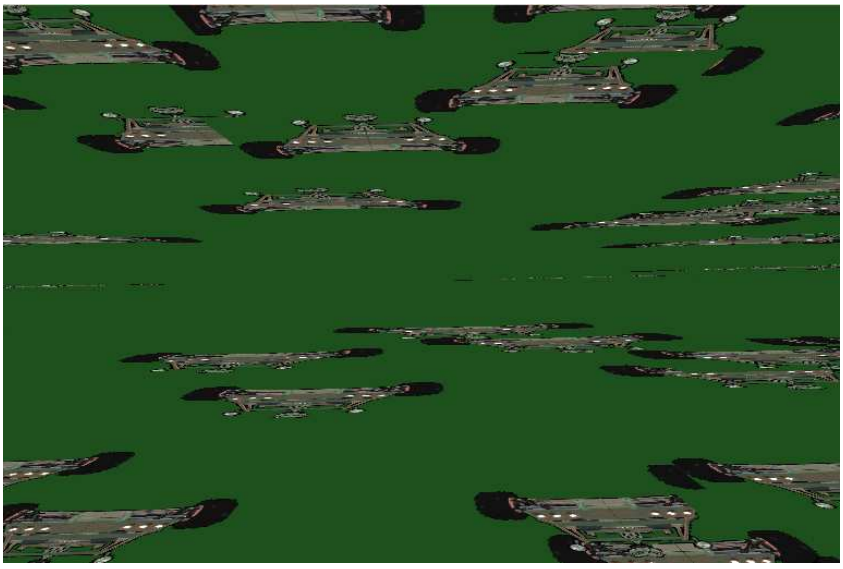


Figure 2.22: A different view of the texture-mapped scene shown in Figure 2.21 from a new camera position.

Consider the example of a runner on flat ground—that is, in two dimensional space. Say they are running at 10 miles per hour (their speed) in a direction 30° south of east (their direction). In *Processing* terms, this is in a direction rotated 30°, or $\pi/6$ radians, from that of the x-axis in a clockwise direction.

Another way of thinking about a vector is in terms of its components in coordinate space. For our example, in an hour the runner will travel 10 miles. In the x-direction (east) they will have moved $10.cos(30) = 10.(\sqrt{3}/2) = 5.\sqrt{3}$ miles, while in the y direction (south) they will have moved $10.sin(30) = 10.(1/2) = 5$ miles. This is illustrated in Figure 2.23. Thus if the runner began at the origin, their position would now be

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5.\sqrt{3} \\ 5 \end{pmatrix}$$

miles in relation to the origin. This is the **direction** vector of travel irrespective of where the runner started. More generally, if the runner began from a general position

$$\underline{\mathbf{P}} = \begin{pmatrix} X \\ Y \end{pmatrix}$$

their position after an hour would be

$$\begin{pmatrix} X \\ Y \end{pmatrix} + \begin{pmatrix} 5.\sqrt{3} \\ 5 \end{pmatrix} = \begin{pmatrix} X + 5.\sqrt{3} \\ Y + 5 \end{pmatrix}$$

in relation to the origin.

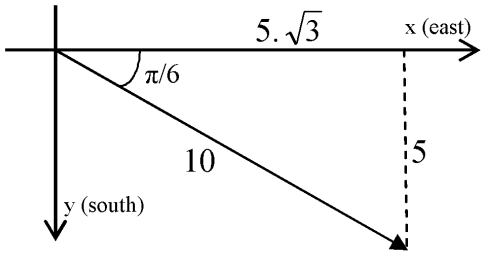


Figure 2.23: Distance moved by the runner.

The x and y components of the runner's velocity are $5.\sqrt{3}$ and 5 mph respectively. That is, the position change is in the same direction as that of the velocity, which is constant.

An $m \times n$ **matrix** is a table of values of the same type, arranged in m rows of n columns. A row vector is a matrix with $m = 1$, or one row; a column vector is a matrix with $n = 1$, or one column.

Addition of two matrices, **A** and **B**, each of m rows and n columns, with (row i, column j) components $a_{i,j}$ and $b_{i,j}$ respectively, is specified by the sum of **A** and **B** being a matrix **C** with components $c_{i,j} = a_{i,j} + b_{i,j}, i = 1 \ldots m, j = 1 \ldots n$. Thus **C** = **A** + **B** = **B** + **A**, which is commutative under addition. Addition is only defined for matrices with the same number of rows as each other and the same number of columns as each other.

Multiplication of two matrices **A** ($m \times K$) and **B** ($K \times n$) is defined when the number of columns in the first matrix is the same as the number of rows in the second.

For **C** = **A.B**,

$$c_{i,j} = \sum_{k=1}^{K} a_{ik}.b_{kj}$$

and **C** is $m \times n$.
In general **A.B** $\neq$ **B.A**, so matrix multiplication is not commutative.

2.6.2 Translation, scaling and rotation of objects in 2-dimensional coordinate space

For objects defined by a set of points, or vertices, such as line segments and polygons, we can move, magnify or turn the object by considering the operation for a general vertex and applying the same operation to each and every vertex. Consider a general point (x, y), that may be one such vertex.

Moving the point by TX in the x direction and TY in the y direction is called **translation**, and results in the point moving to a new position $(x\prime, y\prime)$. This is illustrated in Figure 2.24. In terms of the original position and movement the new position will be

$$x\prime = x + TX$$

$$y\prime = y + TY$$

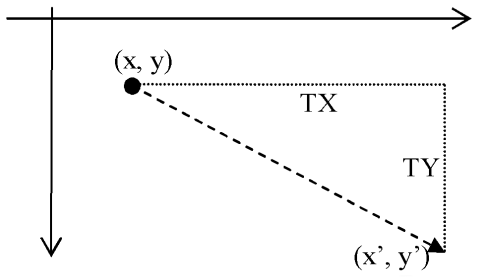


Figure 2.24: Translation of the point (x, y).

In matrix terms this may be written

$$\begin{pmatrix} x\prime \\ y\prime \end{pmatrix} = \begin{pmatrix} 1 & 0 & TX \\ 0 & 1 & TY \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \mathbf{T}'.\underline{\mathbf{P}}$$

where $\mathbf{T}'$ is a matrix with the components for the translation.

We will wish to combine operations such as translation, rotation and scaling, by the operations of addition and multiplication, so all related matrices need to have equal numbers of columns and rows. However $\mathbf{T}'$ has two rows and three columns. To resolve this, we can note that the first equation—which corresponds to the first row in the matrix—relates to an x change and that the second equation—which corresponds to the second row in the matrix—relates to a y change. The first column gives x-multipliers, the second gives y-multipliers and the third has (unit) constant

multipliers. Thus the matrix can be made square by adding in a (unit) constant 'equation', so the equation set becomes

$$x\prime = x + TX$$

$$y\prime = y + TY$$

$$1 = 1$$

In matrix terms, this is

$$\begin{pmatrix} x\prime \\ y\prime \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & TX \\ 0 & 1 & TY \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

or $\underline{\mathbf{P}}' = \mathbf{T}.\underline{\mathbf{P}}$

Rotation, by convention, is measured in radians and can be clockwise or counter-clockwise for increasing angle. In this course unit, and in relation to use in *Processing*, positive rotation is taken as clockwise about the origin. In general the line from the origin to the point will rotate from angle α, by an amount θ. The distance of the point from the origin—the radius of rotation, r—is constant.

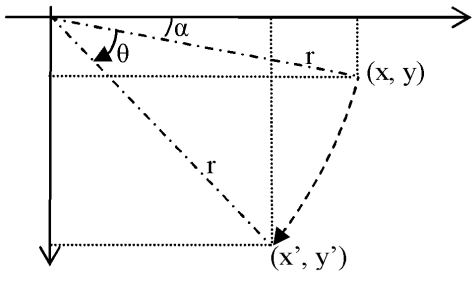


Figure 2.25: Rotation from the point (x, y).

Thus $r = \sqrt{x^2 + y^2} = \sqrt{x\prime^2 + y\prime^2}$.

With reference to Figure 2.25, we can see that

$$x\prime = r.cos(\alpha + \theta) = r.cos(\alpha).cos(\theta) - r.sin(\alpha).sin(\theta) = x.cos(\theta) - y.sin(\theta)$$

$$y\prime = r.sin(\alpha + \theta) = r.cos(\alpha).sin(\theta) + r.sin(\alpha).cos(\theta) = x.sin(\theta) + y.cos(\theta)$$

In matrix form this gives

$$\begin{pmatrix} x\prime \\ y\prime \\ 1 \end{pmatrix} = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

or $\underline{\mathbf{P}}' = \mathbf{R}.\underline{\mathbf{P}}$

Scaling corresponds to magnification of an object in relation to the origin and so for a point (x, y) it corresponds to a multiplication of the x component value by some factor SX and of the y component value by some factor SY. That is, it need not be the same factor in each direction so that, for example, a square may be transformed

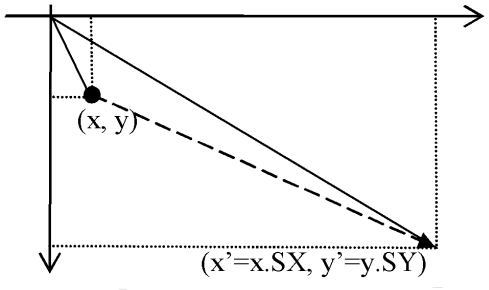


Figure 2.26: Scaling an object by a factor S.

into a rectangle or into a lozenge if the sides of the original square are not aligned with the coordinate axes. This is shown in Figure 2.26.

Thus $x\prime = SX.x$ and $y\prime = SY.y$, or in matrix terms,

$$\begin{pmatrix} x\prime \\ y\prime \\ 1 \end{pmatrix} = \begin{pmatrix} SX & 0 & 0 \\ 0 & SY & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

or $\underline{\mathbf{P}}' = \mathbf{S}.\underline{\mathbf{P}}$

Parenthetically, if $SX = SY = 1$, then $(x\prime, y\prime) = (x, y)$ and the point is unchanged in position. The matrix form of the equations is then

$$\begin{pmatrix} x\prime \\ y\prime \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

or $\underline{\mathbf{P}}' = \mathbf{I}.\underline{\mathbf{P}}$, so $\underline{\mathbf{P}}' = \underline{\mathbf{P}}$

The matrix **I**, with unit elements on the diagonal and zero elements elsewhere, is called the *identity* matrix, and is characterised by it leaving anything changed under multiplication—as does the identity 1 when multiplying a real number.

Combination of transformations is often needed in manipulation of graphical objects. Again we can consider a typical point, such as for a general polygon vertex.

As illustration, consider the following rotation and then scaling of a point:

$$\begin{pmatrix} x\prime \\ y\prime \\ 1 \end{pmatrix} = \begin{pmatrix} SX & 0 & 0 \\ 0 & SY & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} SX.cos(\theta) & -SX.sin(\theta) & 0 \\ SY.sin(\theta) & SY.cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

or $\underline{\mathbf{P}}' = \mathbf{S}.\mathbf{R}.\underline{\mathbf{P}}$.

Correspondingly, scaling of a point, followed by rotation, is given by

$$\begin{pmatrix} x\prime \\ y\prime \\ 1 \end{pmatrix} = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} SX & 0 & 0 \\ 0 & SY & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} SX.cos(\theta) & -SY.sin(\theta) & 0 \\ SX.sin(\theta) & SY.cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

or $\underline{\mathbf{P}}' = \mathbf{R.S.}\underline{\mathbf{P}}$.

Note that in general, $\mathbf{R.S} \neq \mathbf{S.R}$. Under what circumstances is it the case that $\mathbf{R.S} = \mathbf{S.R}$? The answer is when $SX = SY$, as then the corresponding elements of **R.S** and **S.R** are equal.

Rotation about a general point

Note that rotation about some general point (x_0, y_0) can be achieved by translating the point to the origin—that is, translating all objects by that amount—carrying out the rotation, and then translating the point back to its original position—that is, taking objects back to their positions rotated about (x_0, y_0). The matrix combination is:

$$\begin{pmatrix} 1 & 0 & x0 \\ 0 & 1 & y0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} 1 & 0 & -x0 \\ 0 & 1 & -y0 \\ 0 & 0 & 1 \end{pmatrix}$$

2.6.3 Perspective

The viewpoint of an observer, or the **camera** in *Processing*, within a scene, determines the relative positions and apparent sizes of objects. This can be made explicit by considering how the eye to object line intersects with the **viewplane** that contains the **viewport** which is mapped to the actual picture region on the viewing device or screen. More will be presented formally on this later in the course as needed, and so a very simple introduction is given here. Note that creative adaptation may include a curved path of light, as might be encountered due to gravitational effects.

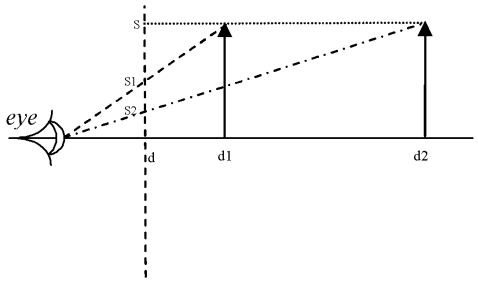


Figure 2.27: Projected sizes of objects (solid arrows) onto a viewplane (dotted line) dependent on distance from eye.

Consider an eyeline perpendicular to the viewplane which is distance d from the observer. As an example consider two objects, each of size S, perpendicular to the eyeline and with their bases on the eyeline, and at distances $d1$ and $d2$ from the observer, as in Figure 2.27. The projected sizes of the objects on the viewplane are $S1$ and $S2$. In the diagram, representation has been reduced to two dimensions by using an orthographic (view from infinity, or parallel projection) view, perpendicular to the eyeline and objects. By similar triangles, $S/d1 = S1/d$ so $S.d = S1.d1$. Also, $S/d2 = S2/d$ so $S.d = S2.d2$.

Thus

$$S1.d1 = S2.d2$$

That is, the product of distance and apparent size is constant in a perspective projection. The advances in Western representational art from Mediaeval times depended partly on increasing the understanding of this relationship.

To the observer the apparent sizes of the objects differ in the ratio $S1 : S2 = d2 : d1$ in relation to their projections onto the viewplane, and so the displayed picture will show the objects' displayed sizes in this proportion. That is, the size of an object as drawn on the viewplane depends inversely on its distance from the observer.

2.6.4 Translation, scaling and rotation of objects in 3-dimensional co-ordinate space

As an introductory treatment we shall consider a simple extension of the equations for two dimensions to three dimensions. Here we consider objects, defined by a set of points or vertices in three dimensions, such as line segments and polygons. We can move, magnify or turn an object by considering the operation for a general vertex and applying the same operation to each and every vertex.

Consider a general point (x, y, z) = **P** that may be one such vertex. This is illustrated in Figure 2.28. As in two dimensions, we will wish to combine operations such as translation, rotation and scaling, by the operations of addition and multiplication, so all related matrices need to have equal numbers of columns and rows.

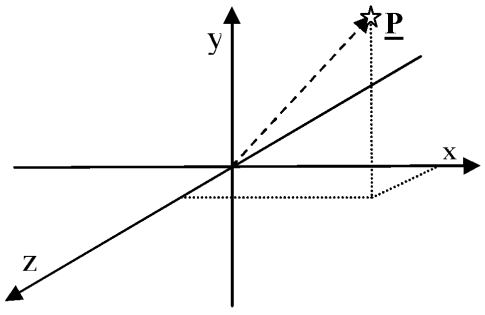


Figure 2.28: Point **P** in three dimensions.

Translation of the point by TX in the x direction, TY in the y direction and TZ in the z direction, results in the point moving to a new position $(x\prime, y\prime, z\prime)$. In terms of the original position and movement the new position will be

$$x\prime = x + TX$$

$$y\prime = y + TY$$
$$z\prime = z + TZ$$

Extending from two dimensions, we can note that the first equation (and 1st row in the matrix) relates to an x change, that the second equation (2nd row in the matrix) relates to a y change, and the third relates to a z change. Writing the equations as for two dimensions would lead to a 4th, constant term (or 4th column in a matrix of coefficients). Thus the matrix can be made square by adding in a (unit) constant 'equation', so the equation set becomes

$$x\prime = x + TX$$
$$y\prime = y + TY$$
$$z\prime = z + TZ$$
$$1 = 1$$

or, in matrix terms,

$$\begin{pmatrix} x\prime \\ y\prime \\ z\prime \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & TX \\ 0 & 1 & 0 & TY \\ 0 & 0 & 1 & TZ \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

or $\underline{\mathbf{P}}' = \mathbf{T}.\underline{\mathbf{P}}$

Rotation can be considered about any line as an axis, in a way that is analogous to rotation about any point in two dimensions. For three dimensions, the basic rotations we need consider are about each of the coordinate axes—x, y and z. Rotation about a general line can be achieved by some combination of translations and axial rotations. You should read up on this in Hughes et al (2013), or in another computer graphics textbook. Rotation in two dimensions, extended to three dimensions, corresponds to rotation about the z axis, as shown in Figure 2.29.

As before, in general the line from the origin to the point will rotate from angle α, by an amount θ.

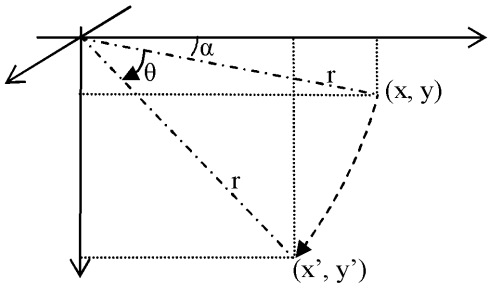


Figure 2.29: Rotation in three dimensions.

The x and y changes are as for 2 dimensions, while the z value is unchanged, corresponding to an equation $z\prime = z$. In matrix form this gives

$$\begin{pmatrix} x\prime \\ y\prime \\ z\prime \\ 1 \end{pmatrix} = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

or $\underline{\mathbf{P}}' = \mathbf{R}_z.\underline{\mathbf{P}}$

For rotation about the x axis, we can aid visualisation by rearranging our view of the axes so x is out of the y-z viewplane. Thus the x axis rotation is like that for z axis rotation, but x for z, y for x and z for y. For rotation about the y axis, we can arrange our view so that y is out of the z-x viewplane. Thus the y axis rotation is like that for x axis rotation, but with y for x, z for y and x for z.

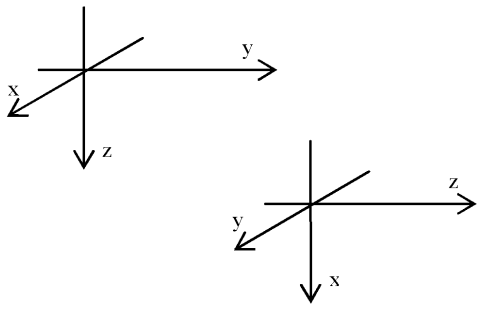


Figure 2.30: Cyclical permutation of x, y and z for rotation.

In both cases we have *cyclically permuted* x, y and z progressively in line with the progressive—z to x to y—cyclical permutation of the rotation axis. See Figure 2.30 for an illustration of this.

Thus the x-axis rotation equations are $y\prime = y.cos(\theta) - z.sin(\theta)$, $z\prime = y.sin(\theta) + z.cos(\theta)$ and $x\prime = x$, giving a matrix form

$$\begin{pmatrix} x\prime \\ y\prime \\ z\prime \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

or $\underline{\mathbf{P}}' = \mathbf{R}_x.\underline{\mathbf{P}}$

The y-axis rotation equations are $z\prime = z.cos(\theta) - x.sin(\theta)$, $x\prime = z.sin(\theta) + x.cos(\theta)$ and $y\prime = y$, giving a matrix form

$$\begin{pmatrix} x\prime \\ y\prime \\ z\prime \\ 1 \end{pmatrix} = \begin{pmatrix} cos(\theta) & 0 & sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

or $\underline{\mathbf{P}}' = \mathbf{R}_y.\underline{\mathbf{P}}$

Scaling in three dimensions is a simple extension of the two dimensional case. For a point (x, y, z), it corresponds to a multiplication of the x component value by some factor SX, of the y component value by some factor SY, and of the z component by a factor SZ. As in two dimensions, it need not be the same factor in each direction.

Thus $x\prime = SX.x, y\prime = SY.y$, and $z\prime = SZ.z$, or in matrix terms

$$\begin{pmatrix} x\prime \\ y\prime \\ z\prime \\ 1 \end{pmatrix} = \begin{pmatrix} SX & 0 & 0 & 0 \\ 0 & SY & 0 & 0 \\ 0 & 0 & SZ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

or $\underline{\mathbf{P}}' = \mathbf{T}.\underline{\mathbf{P}}$

By extension from 2 dimensions, if $SX = SY = SZ = 1$, then $(x\prime, y\prime, z\prime) = (x, y, z)$ and the point is unchanged in position. The matrix form of the equations is then

$$\begin{pmatrix} x\prime \\ y\prime \\ z\prime \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

or $\underline{\mathbf{P}}' = \mathbf{I}.\underline{\mathbf{P}}$, $\underline{\mathbf{P}}' = \underline{\mathbf{P}}$, and $\mathbf{I}$ is now the 4x4 identity matrix.

2.6.5 Point and line in two and three dimensions

Mathematical representation differs from **drawn** representation. Mathematically a point has no size, whilst for an artist a point has physical extent. When drawing on a raster display, and in *Processing*, a point cannot be represented by anything less than a pixel in extent.

In the same way, mathematically a line has no width, whilst an artist's concept of a line is something narrow in proportion to its length, and on a raster device or in *Processing* a line has to be represented as at least one pixel wide. Thus it is important to remember the distinction between points and lines, and their drawn representations on displays. It is partly because of these restrictions in representation that *Processing* usually uses the terms 'point' and 'line' in an artist's sense.

The equation of a line in two dimensions (in the x-y plane) can be written $a.x + b.y + c = 0$, for some real numbers a, b and c where:

- the slope of the line is $-a/b = m$
- the line cuts the x-axis (where $x = 0$) at $y = -c/b = k$ and
- the line cuts the y-axis (where $y = 0$) at $x = -c/a$.

Thus one representation of a line is $y = m.x + k$. There is a difficulty in use of this form though. When $b = 0$ the line is vertical and the slope, m, of the line does not have a finite value. For this reason it is common to define a line as

- $a.x + b.y + c = 0$ (in two dimensions), or
- $a.x + b.y + c.z + d = 0$ (three dimensions), or
- in terms of two points through which it passes, or
- in terms of one point through which it passes and the direction vector of the line.

The two-point and point-direction forms apply to a line in any number of dimensions, with consistent notation. As before, in two dimensions a point can be defined in terms of its coordinates (x, y) while in three dimensions it is (x, y, z). In

these and for a point in any number of dimensions we can describe a point by its vector form

$$\underline{\mathbf{P}} = (x, y)$$

in two dimensions or

$$= (x, y, z)$$

in 3D.

Consider two points **P1** and **P2**, say with coordinates $(x1, y1, z1)$ and $(x2, y2, z2)$, for 3 dimensions, as shown in Figure 2.31.

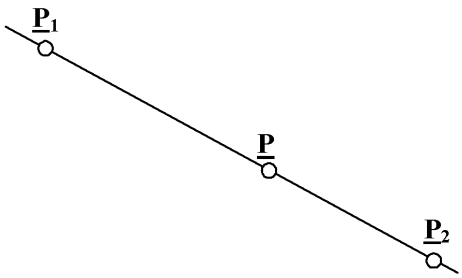


Figure 2.31: Line in three dimensions.

A general point, **P**, on the line can be given by the two-point form,
P = $(1 - \lambda)$.**P1** + λ.**P2**, where λ is a real number.

Rearranging, **P** = **P1** + λ.(**P2** − **P1**) which gives the point-direction form **P** = **P1** + λ.**d** where **d** = **P2** − **P1** is the direction vector of the line.

Note that when $\lambda = 0$, **P** = **P1**, and when $\lambda = 1$, **P** = **P2**. Thus for $\lambda \varepsilon [0, 1]$, **P** ranges between **P1** and **P2**, for $\lambda < 0$ **P** is to the left of **P1**, and for $\lambda > 1$ **P** is to the right of **P2**. The full range of λ is $(-\infty, \infty)$.

Left and right of a line in two dimensions, and left or right of a plane in three dimensions, is an important concept. It can be used in determining if a point is inside or outside a polygon (two dimensions) or a faceted body (three dimensions). For example, consider the form of a line, $f(x, y) = a.x + b.y + c = 0$. A notional viewer looking along a line in the same sense as its direction vector will find that for any point (xa, ya) to the left of the line, it is the case that $f(xa, ya) > 0$, while for any point (xb, yb) to the right of the line, $f(xb, yb) < 0$.

2.7 Summary

This chapter is about 3D graphics, and how we present images in order to give the impression of three dimensions. It extends many of the 2D concepts that you saw in Volume 1 of the study guide, but includes a depth dimension. Some of the concepts are very similar, but there are also new aspects to consider, such as perspective and viewports. The final section is a reference section about matrix algebra and its use in representing and manipulating images in both two and three dimensions.

You should now be able to:

- explain how 3D images are represented and displayed on a two dimensional screen
- describe and use image rendering
- use various OpenGL-based *Processing* methods in creating 3D images
- discuss the representation of lines in three dimensions, and also how this can give the viewer the perception of depth and perspective
- perform various manipulations, including scaling, rotation, and transformation, to change the way an image is presented, in both 2D and 3D
- incorporate texture into images.

2.8 Exercises

Many of the examples below are given in terms of two dimensions for ease of visualisation, but can be extended to three dimensions quite simply.

1. Consider the texture mapping code in Figure 2.17. Modify the code so that it texture maps something other than bricks, such as waves, or any other texture of your choice.
2. Calculate **R.S** and **S.R** when $\theta = \pi/4, SX = 2$ and $SY = 3$. Leave your answer in exact closed form; that is, utilise the length of the diagonal of a unit square, expressed as a square root.

   Hence find the components of $\underline{\mathbf{P}}\prime$ when $x = y = 1$, so $\underline{\mathbf{P}} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ for $\underline{\mathbf{P}}\prime = \mathbf{S.R}.\underline{\mathbf{P}}$ and for $\underline{\mathbf{P}}\prime = \mathbf{R.S}.\underline{\mathbf{P}}$.
3. By computing the matrix products, show that—in general—translation followed by scaling does not give the same result as scaling followed by translation.
4. In the particular case that $TX = TY$, we have shown that **R.S** = **S.R**. For $TX = TY$, find whether or not **T.S** = **S.T** and/or **R.T** = **T.R**, by calculating the product matrices.
5. Are there any combinations of translation, scaling and rotation in pairs, where the order of operation does not matter? Explain your answer.
6. In relation to Figure 2.27 above, regarding perspective, what happens in drawing terms as $d1$, the distance of the first object from the observer, decreases and reaches zero? What happens in drawing terms, in terms of pixel representation in a *Processing* sketch, as $d2$ increases without bound? What is your understanding of what a sensible representation would be?
7. For a viewplane 25 cm from the eye, consider two objects presented as discussed in Section 2.6.3 above. If one object, A, is of size 100 cm at distance 200 cm from the observer and the other object, B, is size 200 cm at distance 500 cm from the observer, calculate the ratio of apparent sizes as seen by the observer, and calculate the projected size of each object on the viewplane.
8. A standard operation in film or animation work is called ‘maintaining the vertical’ which means that the vertical as perceived by an observer, or camera, corresponds to the vertical in the scene. In terms of computer graphics, this means that when an observer, or camera, observes the y axis, that axis should appear vertical. Read up on this topic in a textbook, such as Hughes et al (2013), or on the internet and state your sources in any answer to the questions below.

(a) Under what circumstances does ‘maintaining the vertical’ have no meaning? When these circumstances arise transitionally in an animation, what choices may be made in drawing the scene?

(b) Recall films or computer animations that you have seen. List two or three instances when ‘maintaining the vertical’ had no usual meaning (at least one from a standard film) and briefly describe the way or ways that the issue was addressed in terms of camera operation (e.g. pan or tilt).

(c) On the basis of your answer to 8b and your own investigations, describe distinct ways of managing loss of meaning to ‘maintaining the vertical’, including what each involves in terms of computer or camera operation.

(d) There is a particular film genre that was first in not always ‘maintaining the vertical’ to give a particular effect. Find out the name of this genre and what effect the director was trying to achieve.

Chapter 3

3D Motion and Control

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 `http://www.processing.org/reference`, online *Processing* reference manual.

Additional reading

Greenberg, I. *Processing: Creative Coding and Computational Art* (Apress (Springer-Verlag) 2007) [ISBN 159059617X].
Maeda, J. *Creative Code* (Thames and Hudson, 2004) [ISBN 0500285179].
Moggridge, B. *Designing Interactions* (MIT Press, 2006) [ISBN 0262134748].

3.1 Introduction

In the previous chapter we looked at how to construct 3D scenes by placing objects in the scene and by locating the camera for the viewing position. The scenes and viewing position were static.

To make a 3D scene animated, the same principles of motion apply as we dicussed in Chapter 8 of Volume 1 for the 2D case. That is, persistence of coordinates by global variables or class fields, `setup()` and `draw()` methods, resetting the background and re-drawing the scene.

Working in 3D creates new possibilities for motion which we discuss in this chapter.

3.2 Motion in a straight line

To create motion in a straight line the component velocities, `dx, dy and dz`, are constant. The velocities are in units of pixels per frame. To change the units to pixels per second, metres per second or other dimensional units, we can control the `frameRate()`. By setting this to 10, each frame will last 0.1s (seconds) so velocities will be measured in pixels per 0.1s (or 10 times velocity pixels per second).

In the sketch shown in Figure 3.1, we set the initial position to `(0, SZ/2, -2*SZ)` which is left, half screen height and twice the screen height in the distance. In the `setup()` method, we call `frameRate(10)` and set the velocities to `(0, 1, 10)`. The camera position is set so that it is in the centre of the screen and looking directly forward.

The sketch moves a simple scene, consisting of a box and a sphere, toward the viewer at a rate of 100 pixels per second in the `z` dimension and shifts it by 10 pixels per second in the `y` dimension.

```
// Motion in a straight line by moving the objects in
// the scene

int SZ=512;
float x=0.0, y=SZ/2, z=-2*SZ; // object positions
float dx, dy, dz;             // object velocities
float cx, cy, cz;             // camera position
float lx, ly, lz;             // camera direction

void setup() {
  size(SZ, SZ, P3D);
  cx=SZ/2; cy=SZ/2; cz=SZ/2;
  lx=cx; ly=cy; lz=-cz;
  camera(cx,cy,cz,lx,ly,lz,0,1,0);
  frameRate(10);
  noFill();
  sphereDetail(10);
  dx=0;
  dy=1;
  dz=10;
}

void draw() {
  background(255);
  translate(x, y, z);
  box(100);
  translate(width,0,0);
  sphere(100);
  x += dx; // update object x position
  y += dy; // update y
  z += dz; // update z
}
```

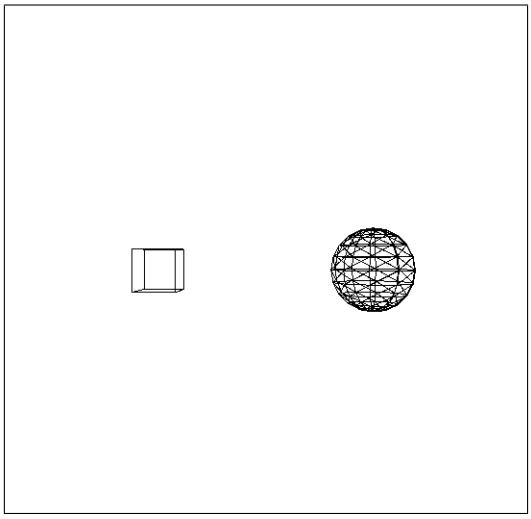


Figure 3.1: Motion by moving the objects in a scene.

Learning activity

Initialise the scene at a position of `(0, -SZ, -2*SZ)` with velocities `(0, 10, 10)`. What do you see?

Initialise the position in the above sketch to three other locations and velocities in 3D space.

When can you see the scene and when is it invisible?

3.2.1 Camera motion

Motions can be constructed either by moving the scene to the viewer or by moving the viewer to the scene. They are equivalent in terms of what is seen[1] but different in the way they are constructed. The sketch in Figure 3.2 illustrates motion of the camera through the scene to create the same motion as in Figure 3.1—compare the different implementations of the `draw()` method in these two examples.

[1] This is true at least for the case where there is no relative motion between objects in the scene.

```
// Motion in a straight line by moving the camera

int SZ=512;
float x=0.0, y=SZ/2, z=-2*SZ; // object positions
float dx, dy, dz;             // camera velocity
float cx, cy, cz;             // camera position
float lx, ly, lz;             // camera direction

void setup() {
  size(SZ, SZ, P3D);
  cx=(SZ/2)-x; cy=(SZ/2)-y; cz=(SZ/2)-z;
  lx=cx; ly=cy; lz=-cz;
  camera(cx,cy,cz,lx,ly,lz,0,1,0);
  frameRate(10);
  noFill();
  sphereDetail(10);
  dx=0;
  dy=1;
  dz=10;
}

void draw() {
  camera(cx,cy,cz,lx,ly,lz,0,1,0);
  background(255);
  box(100);
  translate(width,0,0);
  sphere(100);
  cx -= dx; // update camera x position
  cy -= dy; // update y
  cz -= dz; // update z
}
```

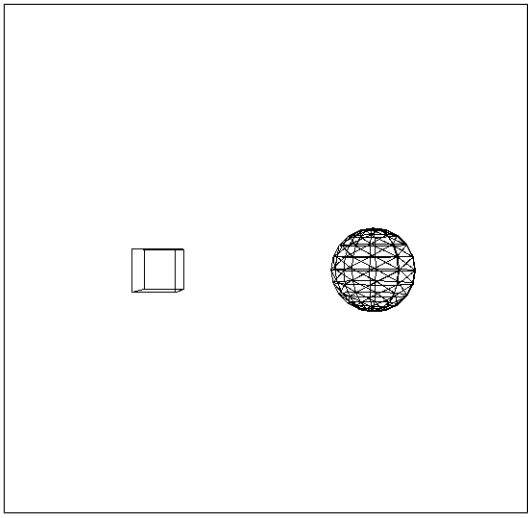


Figure 3.2: Motion by moving the camera.

The negative velocity values when updating the camera position in Figure 3.2 are to make the 3D scene equivalent to the example in Figure 3.1. The camera and the scene are facing in opposite directions because the camera is looking at the scene. So a step in a positive direction for the scene, with a stationary camera looking at it, produces the same screen image as a step in the negative direction for the camera looking at a stationary scene.

The global position variables `(x, y, z)` are only used in this sketch to initialise the camera position to the *inverse* of the initial position: i.e. `(-x, -y, -z)`. For comparison with the previous example, we have used the same initial position coordinates but we have negated them.

Learning activity

Modify the sketch above to use the following camera initial positions:

camera(-SZ, -y, -z, SZ ,-y , 0 , 0, 1, 0);
camera(SZ/2, -y-SZ, -z, SZ/2 ,-y+SZ , 0 , 0, 1, 0);
camera(0, +y, +z, 0 ,0 , 0 , 0, 1, 0);

What do you observe in each case?

Interpret (explain) these results with a sentence on each.

3.2.2 Camera transformations

In addition to the `camera()` method, *Processing* provides methods which allow more sophisticated control of the camera position. The camera actually has its own transformation matrix, independent of the scene's transformations. These can be controlled using the `beginCamera()` and `endCamera()` methods.

Camera transformations work in a different way from the standard graphical transformations. The first important difference is that the transformation is not reset with each call to the `draw()` method; the camera's transformation is persisted by *Processing*.

The second important difference between camera transformations and scene transformations is that the coordinate systems are kept separate. The camera coordinate-system transformation does not affect the scene coordinate system. Likewise, the scene coordinate system transformation does not affect the camera coordinate system transformation.

The first property, of transformation persistence, means that the absolute position (and rotation) of the camera does not have to be stored by the programmer in a set of global variables as in other kinds of motion.

Thus, within the `draw()` method we only need to update from the current persisted camera position. This simplifies the main draw loop significantly because we no longer need the global position variables to be updated. Instead, we use the velocities in the camera `translate()` transform.

Figure 3.3 is an example of a sketch that uses the `beginCamera()` and `endCamera()` methods.

3.2.3 Motion parallax

When moving in a 3D scene, one of the phenomena we observe is *motion parallax*. This is the phenomenon where objects which are at different distances from the viewer appear to change their relative positions as the viewer moves. We observe this effect when looking out of a train window and we see a distant church or skyscraper appear behind a near object, such as a tree, then, a few moments later, it appears to have moved with the train and now appears behind a different object.

Motion parallax occurs because the relative angles from the viewer to each of the objects changes as the viewer moves. It is most noticable when the viewer is moving at a right angle to the scene but looking directly at the scene. This is the situation in looking out of a train window which is a sideways view from forward motion.

In Figure 3.3, all the objects in the scene are aligned in the `x` and `y` axis. However, the motion of the camera makes objects at different distances appear at different relative positions as it sweeps past the scene. Distant objects appear to move more slowly than near objects.

A scene is constructed such that the camera is moving at right angles to the viewing direction. There are three layers spaced at `-SZ`, `-2*SZ` and `-3*SZ` from the viewer and consisting of a row of squares, circles and triangles respectively.

The camera sweeps along the `x` axis from left to right looking into the scene in the `z`

```
// 3D Motion parallax
// Objects in the distance appear to move more slowly

int SZ=512;
float dx=10;

void setup() {
  size(SZ,SZ,P3D);
  rectMode(CENTER);
  ellipseMode(CENTER);
  frameRate(30);
  fill(255);
  // set initial camera position
  camera(10*SZ,0,SZ,10*SZ,0,-1,0,1,0);
}

void draw() {
  background(0);

  // move the camera position each time the draw
  // method is called
  beginCamera();
  translate(dx,0,0);
  endCamera();

  // draw a series of squares
  for (int i=-10*SZ ; i < 10*SZ ; i+=SZ/2) {
    rect(i, 0, SZ/4, SZ/4);
  }

  // draw a series of circles behind the squares
  translate(0,0,-SZ);
  for (int i=-10*SZ ; i < 10*SZ ; i+=SZ/2) {
    ellipse(i, 0, SZ/4, SZ/4);
  }

  // draw a series of triangles behind the circles
  translate(0,0,-SZ);
  float B=SZ/4; // triangle edge length
  float A=sqrt(3)*B/2; // triangle height
  for (int i=-10*SZ ; i < 10*SZ ; i+=SZ/2) {
    triangle(i-B/2, A/3, i+B/2, A/3, i, -2*A/3);
  }
}
```

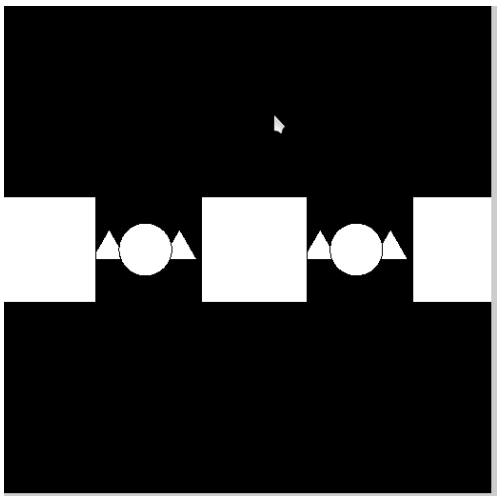


Figure 3.3: An illustration of *motion parallax*.

dimension. We observe the motion parallax as the position relationships between the squares, circles and triangles appear to change even though they are all in fixed rows.

3.3 Circular motion

Circular motion is achieved by rotation. To rotate a scene around an axis we translate the scene to the centre of rotation and then perform the rotation.

The velocity of the rotation is called the *angular velocity* with units in radians per second. Figure 3.4 illustrates the use of a timed rotation to create a simple planet-satellite system. The `frameRate(10)` method allows us to control the speed of motion in the scene. The coordinate system is translated to the centre of the screen, where the default camera position is looking, and then a coordinate system rotation is performed around the `y` axis.

The planet sphere is drawn at the centre of the screen, which is the current origin, followed by a translation and a second smaller sphere is drawn. Finally, the angle of rotation is updated so that a rotation is completed every 10 seconds: `TWO_PI / (10 * frameRate)`, which is one tenth of the once-per-second rotation speed expressed by `TWO_PI / frameRate`.

```
// Rotating planet-satellite system

float angle = 0.0;

void setup() {
  size(512,512,P3D);
  frameRate(10);
  sphereDetail(10);
  noFill();
}

void draw() {
  background(255);
  translate(width/2,height/2,0);
  rotateY(angle);
  stroke(255,0,0);
  sphere(100);         // planet
  translate(200,0);
  stroke(0,0,255);
  sphere(10);          // satellite
  angle += TWO_PI/(10*frameRate);
}
```

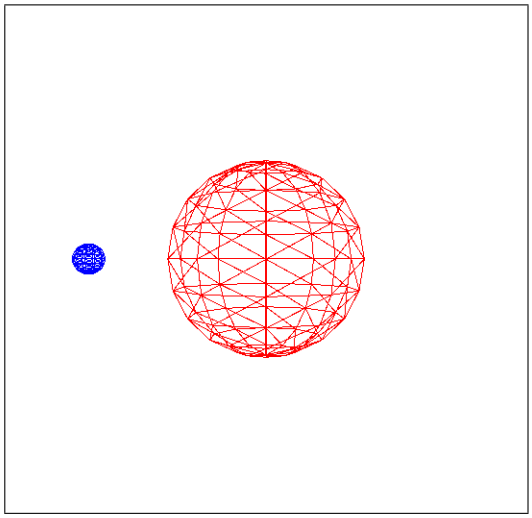


Figure 3.4: A planet-satellite system using coordinate transformations to animate the scene. The scene is timed to complete a rotation every 10 seconds.

Learning activity

Taking Figure 3.4 as a starting point, make the satellite rotate on its own axis using a second rotation around the `y` axis. Perform this operation on the line before drawing the satellite `sphere()`.

Expand the previous example to create a simple solar system in which the planet rotates around a sun, which is at the centre, and the satellite rotates around the planet, with each sphere rotating around its own central axis.

3.4 Motion control

A feature of interactive graphics systems, such as 3D games and simulation, is that there is a high degree of interactive control. In *Processing* we may use the mouse, keyboard or any device that can be connected to a serial port on the computer.

Interactive 3D graphics systems divide the input modes into manipulation and navigation. A manipulation affects the state of an object in the 3D environment. For example, we may click on an object to change its colour or texture or we may drag it to a new position in the 3D scene. This type of interaction is called *direct manipulation* and is an advanced type of graphical user interface (GUI). Direct manipulation of objects in a 3D environment is a complex type of interaction. The online social virtual environment *Second Life* includes a system for modifying and constructing objects by direct manipulation using mouse (and keyboard) control.

3D navigation is the process of moving the camera in a 3D scene in a manner that allows the viewer to explore different parts of the scene from multiple viewpoints. We often use the analogy of flying to design 3D navigation motions.

3.4.1 Types of navigation

Planar navigation motion in a straight line is aligned with the coordinate system axes such as left-right, up-down, forwards-backwards. We may use a combination of planar motions in our sketches to allow exploration of the 3D space. Planar motions are implemented using the techniques of motion in a straight line. That is, we make three persistent variables for position `(x, y, z)` and velocity `(dx, dy, dz)`, and we control the motion by updating these variables using `x+=dx; y+=dy` and `z+=dz`.

This type of motion is somewhat like a 3D lift (or elevator). We can move in the three dimensions but we cannot rotate to look left or right or look up or down. The camera always faces in one direction under these types of motion.

In addition to the planar navigation motions there are three degrees of freedom in rotation. Figure 3.5 illustrates the types of rotation that are available for navigation in a 3D space: *pitch*, *roll* and *yaw*. Pitch is a rotation around the `x` axis so that the nose of the aeroplane is pitched up or down. Yaw is a rotation around the `y` axis such that the nose of the aeroplane points to the left or right. Roll is a rotation around the `z` axis such that the right or left wing of the aeroplane is raised and the opposite wing is lowered.

In the following examples we shall use a combination of planar motion and rotations to create movement of the camera in a 3D space.

3.4.2 Mouse

Input by the user using the mouse is represented in *Processing* using a combination of method callbacks, called when a mouse event occurs, and global system variables, which hold the mouse position and button values. Table 3.1 lists the available methods and system variables associated with mouse input.

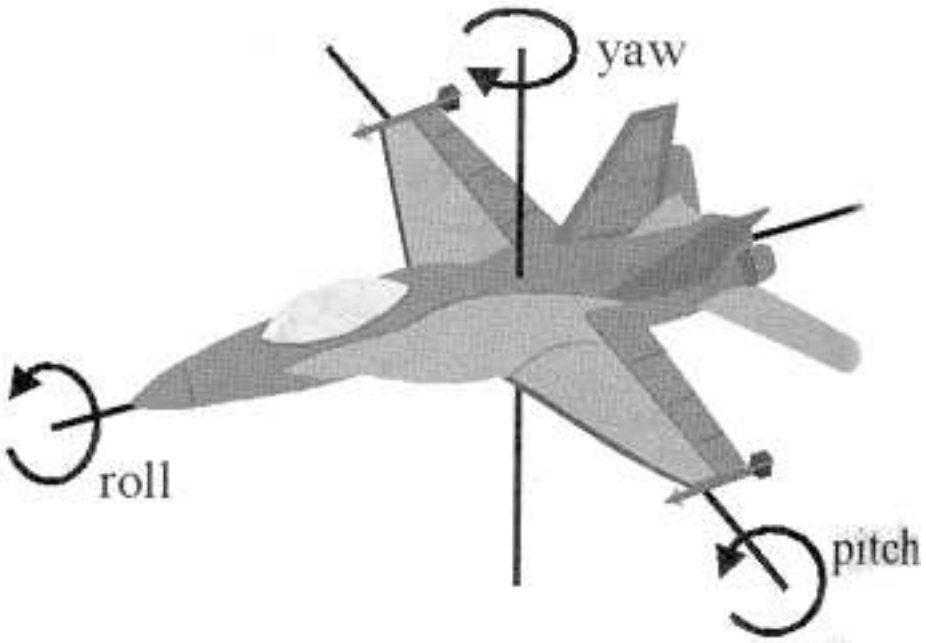


Figure 3.5: Rotations used for 3D navigation.

The central problem in providing control and navigation in a 3D environment is that the mouse does not have many degrees of freedom. It can only move left-right and forward-backward, and has one, two or three buttons and sometimes a wheel. We must make a choice as to which motions will be controlled by which mouse actions and then program our sketches with the necessary mouse logic to implement these motions.

3.4.3 Mouse flythrough

Figure 3.6 illustrates the use of the mouse for navigating a 3D scene. The left-right motion of the mouse rotates the camera around the `y` axis (yaw). The forward-backward motion of the mouse rotates the camera around the `x` axis (pitch). Motion is activated by pressing one of the mouse buttons: LEFT for forward motion (in the direction the camera is facing) and RIGHT for backwards motion.

We could also remove the `if(mousePressed)` statements from between `beginCamera()` and `endCamera()` calls and instead use a mouse differential position to control the rotations. To do this we use the difference between the previous mouse position and the current mouse position as the amount to rotate. If the current mouse position is greater than the previous position, then there is a positive rotation. If the current mouse position is less than the previous mouse position, then there is a negative rotation. The system variables `pmouseX` and `pmouseY` encode the previous frame's `mouseX` and `mouseY` positions. So the expressions `mouseX-pmouseX` and `mouseY-pmouseY` are differential positions that are zero unless the mouse is moved between calls to `draw()`.

```
// Example of 3D navigation by mouse control

int SZ=512;
float x,y,z=1000;
float dx,dy,dz;

void setup() {
  size(SZ, SZ, P3D);
  noFill();
  frameRate(10);
  camera(SZ/2,SZ/2,z,SZ/2,SZ/2,-1,0,1,0);
}

void draw() {
  background(255);
  // Forward-backward control
  if(mousePressed){
    if(mouseButton==LEFT)
      dz=10;
    else if(mouseButton==RIGHT)
      dz=-10;
  }
  else {
    dz=0;
  }

  beginCamera();
  translate(-dx, -dy, -dz);
  if (mousePressed) {
    rotateX((PI/16)*(0.5-(float)mouseY/SZ));  // Pitch
    rotateY((PI/16)*(-0.5+(float)mouseX/SZ)); // Yaw
  }
  endCamera();

  box(100);
  translate(width,0,0);
  sphere(100);
}
```

Figure 3.6: Sketch to implement 3D navigation using pitch, yaw and forward-backward motion with the mouse. The LEFT and RIGHT buttons control forward-backward motion in the direction the camera is facing. Pitch and Yaw are controlled by up-down, left-right mouse position respectively.

method	description
mouseDragged()	user-defined method called when mouse is moved with a button down
mouseMoved()	user-defined method called when mouse is moved
mouseButton	when button is pressed contains either `LEFT, RIGHT` or `CENTER`
mouseX	x-axis position of the mouse (with `z=0` and no coordinate rotations)
mouseReleased()	user-defined method called when mouse button released
pmouseX	previous frame mouse x-axis position
mousePressed()	user-defined method called when mouse button pressed
mouseY	y-axis position of the mouse (with `z=0` and no coordinate rotations)
mousePressed	boolean set to true if a mouse button is pressed
pmouseY	previous frame mouse y-axis position
mouseClicked()	user-defined method called when mouse button pressed then released
mouseWheel()	user-defined method called when mouse wheel rotated

Table 3.1: Mouse input methods and *Processing* system variables.

Learning activity

Starting with Figure 3.6, re-write the camera motion part of the code to use differential mouse position as the rotation control instead of `mousePressed` and absolute position.

Re-write your expressions so that a mouse movement to the left rotates the scene right, and a mouse movement upwards rotates the scene downwards.

It might take a little bit of getting used to the control. It is actually quite difficult to navigate a 3D scene. In this example we have mapped left mouse movement to a left rotation. Sometimes the sense of rotation is reversed, so a left mouse motion creates yaw as right rotation. The same is true for pitch: when the mouse is moved up the screen, the pitch may be positive or negative depending on the application.

3.4.4 Keyboard

The mouse offers the possibility of mapping movement with the hand to movement on screen, but as we discussed above, it is limited in the range of motion that is offered. Often we want to implement many different motions in a single sketch so we need to find a way to allow the user to input them all.

The keyboard has over a hundred keys, each or which can be detected by *Processing*. We can map keys to particular motions to expand the scope of motion within our sketches. Table 3.2 shows the system variables and method calls available to handle input from the keyboard.

The sketch in Figure 3.7 shows six motions under the control of the keyboard. Each motion has two directions, such as forwards / backwards. We have used terms borrowed from cinematography to describe all the motions: pedestal (up-down),

method	description
keyCode	detect pressing of special keys such as the UP, DOWN, LEFT, RIGHT arrow keys.
keyReleased()	user-defined method called when any key is released
keyPressed()	user-defined method called when any key is pressed
key	the `char` value of the pressed key: 'a', 'B', '=', etc.
keyPressed	boolean set to true if a key is pressed
keyTyped()	user-defined method called once when any key is pressed

Table 3.2: Keyboard input methods and *Processing* system variables.

dolly (forwards-backwards), truck (left-right), pan (`rotateY`), tilt (`rotateX`) and roll (`rotateZ`).

Learning activity

Re-write some of the sketches from Chapter 2 to use the camera motion controls from Figure 3.7. You may have to practise to navigate around the scene.

Make your own keyboard controls by changing the keys that cause the motions.

Add a mouse control to your sketches so that motion is controlled by both keyboard and mouse. The mouse controls should control different motions from the keyboard so that they can be used together simultaneously. For example, you may want to use the mouse for Pitch, Yaw and Dolly motions, and the keyboard for the others.

3.5 Creative applications of 3D motion

Motion is a feature in many creative computing applications such as games, animations, screen savers and film special effects. The following examples illustrate some techniques that draw upon the 3D motion principles covered in this chapter.

3.5.1 A 3D paint brush

Figure 3.8 demonstrates how a paint brush can be simulated using 3D animation. An abstract painting is created by moving a textured brush around the 2D screen. The textures brush varies in colour, size and 3D orientation so it creates layers of paint. The colour of the paint brush is deterministic and is calculated by the `x` and `y` position of the brush using the HSB colour model. The size of the paint brush is determined by scaling the `x` and `y` axes.

The texture effect, giving the impression of brush strokes, is created using a series of rectangles that are individually rotated around the `x`,`y` and `z` axes. The rectangles are planes, so when they are rotated around the `y` or `x` axis, they protrude from the `x`–`y` plane (where `z`==`0`) by different amounts. This is what creates the effect of layers of paint visible through a complex feathered texture, giving the impression of a paint brush stroke. The code that achieves the image is shown in Figure 3.9.

```
// Example of camera motions controlled by the keyboard

void setup() {
  size(512,512,P3D);
  background(0);
  noFill();
  stroke(255);
  camera(width/2,height/2,0,width/2,height/2,-1,0,-1,0);
}

void draw() {
  beginCamera();
  if (keyPressed) {
    if (keyCode==UP)
      translate(0,1,0);  // pedestal up
    if (keyCode==DOWN)
      translate(0,-1,0); // pedestal down
    if (keyCode==LEFT)
      translate(1,0,0);  // truck left
    if (keyCode==RIGHT)
      translate(-1,0,0); // truck right
    if (key=='f')
      translate(0,0,-1); // dolly forward
    if (key=='b')
      translate(0,0,1);  // dolly backward
    if (key=='q')
      rotateY(.01);      // pan left
    if (key=='w')
      rotateY(-.01);     // pan right
    if (key=='a')
      rotateX(-.01);     // tilt up
    if (key=='s')
      rotateX(.01);      // tilt down
    if (key=='z')
      rotateZ(-.01);     // roll left
    if (key=='x')
      rotateZ(.01);      // roll right
  }
  endCamera();

  background(0);
  pushMatrix();
  translate(width/2,height/2,-500);
  box(100);
  popMatrix();
}
```

Figure 3.7: Camera motions controlled by the keyboard. A range of camera motions with terms borrowed from cinematography for each of the motions.

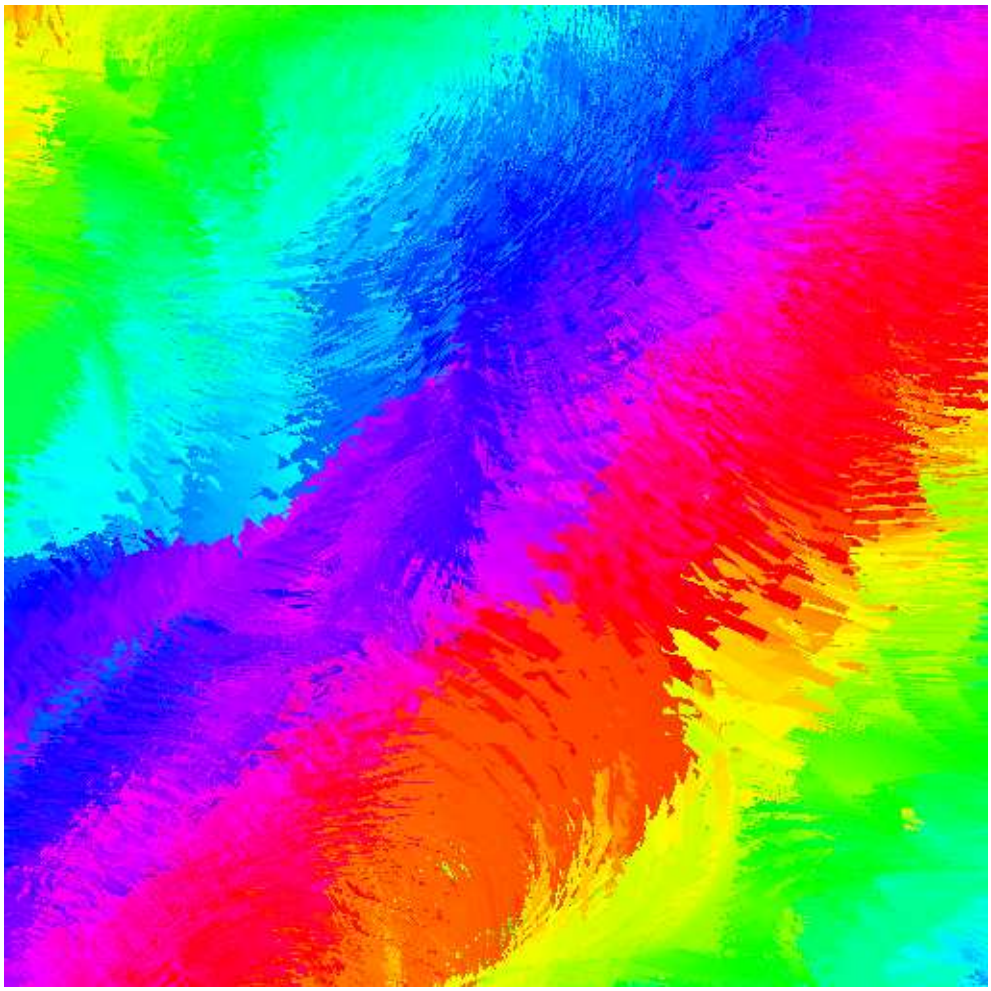


Figure 3.8: A 2D abstract painting created using 3D brush strokes to create the illusion of brush feathering.

```
// 3D Paint Brush demo

int SZ=512;
float x,y,dx,dy; // brush coordinates
float s,ds;      // brush scaling factor

void setup() {
  size(SZ,SZ,P3D);
  x=SZ/2; y=SZ/2;
  s=1; ds=.002; dx=1; dy=1;
  colorMode(HSB,TWO_PI,1,1);
  background(0,0,0);
  noStroke();
  rectMode(CENTER);
}

void draw() {
  fill((x+y)/100.%TWO_PI, 1, 1); // fill colour determined by x and y
  translate(x,y);
  scale(s,s);

  // draw the brush as a series of 20 rectangles at different orientations
  for (int k=0;k<20;k++) {
    rotate((x+y)/100.0);
    rotateX(x/100.0);
    rotateY(y/100.0);
    rect(0,0,100,50);
  }

  // update coordinates and scaling factor
  s=s+ds;
  x=x+dx;
  y=y+dy;
  if (s > 2  || s < 0.75     ) ds=-ds;
  if (x <= 0 || x >= width-1 ) dx=-dx;
  if (y <= 0 || y >= height-1) dy=-dy;
  if (random(100)<3) {
    dx+=random(1)-0.5;
    dy+=random(1)-0.5;
  }
}
```

Figure 3.9: The source code for Figure 3.8.

3.5.2 Painting by 3D swarms

You will recall from Chapter 8 of Volume 1 that we investigated Perlin noise motion in 2D. We can also create Perlin noise motion in 3D, giving the impression of life-like motion, or group behaviours such as swarming. To do this we need to generate a third motion variable for the `z` dimension.

Figure 3.10 shows a painting generated by a cloud of 3D boxes moving under the control of Perlin noise motion. The coloured boxes leave a trail that gradually fades using a background rectangle filled with a transparent colour. By increasing the density of the swarm, or the size of the motion, or the length of the trail by increasing transparency, we can create many different abstract painting effects. The source code is shown in Figure 3.11. The offsets for the `y` and `z` noise values are related to the `x` offset value by 1.41 and 1.61 which are approximations to `sqrt(2)` and `(1+sqrt(5))/2` respectively. In general, offsets that are very large, say in the 10s or 100s, will create a greater degree of randomness or jerkiness in the motion.

3.5.3 Painting by gestures

Paintings that rely on motion to build up a pattern of colours on the screen have the property that we can see the gestures, or the processes, that create the final result. Simply looking at the final result allows us to see evidence of the dynamic gestural content, even if we have no knowledge of the processes that created them.

Figure 3.12 (page 86) illustrates an application of 3D motion under the control of the keyboard for creating a gestural painting. This painting was constructed by flying through the coloured boxes moving in 3D Perlin motion. Transparency values were set on the colours such that they persisted for a few seconds, leaving a coloured trace. The speed of the fly-through determines the density of colour in the painting and also the shape of the trail. Different velocities and accelerations through the scene create a painting with different dynamic brush strokes.

The technique of using motion gestures to leave colour trails creates an effect that alludes to the famous 20th-century painter Jackson Pollock, whose abstract paintings were constructed using various gestures, such as *throwing* and *spilling* paint from brushes, paint cans or various shapes of stick onto a canvas. Figure 3.13, on page 87, shows a photograph of one of Pollock's abstract gestural paintings that is hung in the Museum of Modern Art in New York City, USA.

3.6 Summary

In this chapter we learned how to create 3D motion in a straight line, on circular paths and using Perlin noise motion. We also learned how to control motion using keyboard and mouse input. The types of motion that are available include planar motions and rotations in all three dimensions. We saw that transformations of a 3D scene are equivalent (inversely) to transformations of a camera viewpoint, and that control of the camera is the way to implement first-person control in 3D graphics. Finally, we looked at some examples of creative 3D graphics that simulated the painting process, but went beyond what is available in 2D drawing.

Figure 3.10: 3D Perlin motion used with transparent colours in order to create a moving swarm of objects that leave a trace. This trace creates an image that we could regard as a type of painting.

```
// 3D motion with Perlin noise

int SZ=512;
float xincrement = 0.005;
float xoff = 0.0;

void setup() {
  size(SZ,SZ,P3D);
  colorMode(HSB);
  background(0,0,0);
  frameRate(30);
  noStroke();
}

void draw() {
  // Create an alpha blended background to implement
  // fading traces
  fill(0,0,0,20);
  rect(0,0,width,height);

  // Draw 100 cubes
  for (int k=1; k<=100; k++) {
    // Get a noise value based on xoff and
    // scale it by the screen width
    float n1 = noise(xoff+k)*width;
    float n2 = noise(xoff+1.41*k)*height;
    float n3 = noise(xoff+1.61*k)*height;

    // Draw the ellipse at the value produced by perlin noise
    fill((k*10)%255,255,255,100); // notice the Alpha channel
    pushMatrix(); // Translate each cube individually
    translate(n1,n2,n3);
    box(3);
    popMatrix(); // And pop the coordinate system back
  }
  // With each cycle, increment xoff
  xoff += xincrement;
}
```

Figure 3.11: Source code for the image in Figure 3.10.

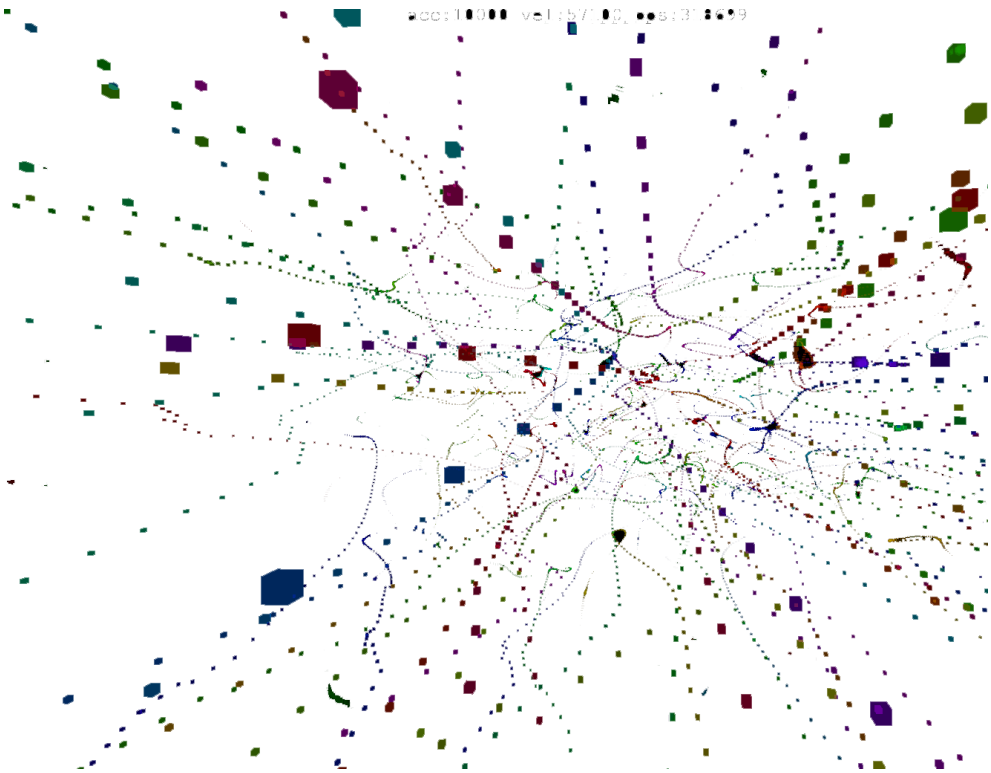


Figure 3.12: A 3D abstract painting. The image is generated by a 3D swarm of brushes in motion. The painter operates the painting by flying a canvas through the space and allowing paint boxes to hit it as they are in motion. This technique alludes to painter Jackson Pollock's technique of throwing paint onto a canvas with very precise gestures, to create abstract forms with dynamic patterns.

Figure 3.13: A photograph of an abstract painting by Jackson Pollock, which is displayed in the East Wing of the Museum of Modern Art in New York City, USA. Photo courtesy of C.K.H. at `http://flickr.com` photo repository.

You should now be able to:

- describe the difference between camera transformations and screen transformations, and decide which are appropriate to use in which circumstances
- describe the phenomenon of motion parallax, and make use of it in 3D motion where appropriate
- implement circular motion, or rotation
- implement camera motion control using a range of inputs, including the mouse and the keyboard
- use 3D motion in a creative context, to implement things such as animation and special effects.

3.7 Exercises

1. Use the code in Figure 3.3 to explore the concept of motion parallax. Modify the code to implement different aspects and examples of the phenomenon.
2. Consider the code in Figure 3.9. Modify the code to give different creative effects. You could vary the colours, rotations, perspective, etc., to obtain different effects.
3. Find out what you can about the work of Jackson Pollock. Figure 3.12 shows an image in the style of some of Pollock's work. Other pieces of Pollock's display different themes. Write another piece of code that will create images in a theme or style that is derivative of a different Pollock work.

Chapter 4

Image

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 `http://www.processing.org/reference`, online *Processing* reference manual.

Additional reading

Greenberg, I. *Processing: Creative Coding and Computational Art* (Apress (Springer-Verlag) 2007), [ISBN 159059617X].
Maeda, J. *Creative Code* (Thames and Hudson, 2004) [ISBN 0500285179].
Moggridge, B. *Designing Interactions* (MIT Press, 2006) [ISBN 0262134748].

4.1 Introduction

Digital images are created using a combination of computer-generated graphics and digital photographs. In this chapter you will learn how to load, display and manipulate images stored in files. These image files might contain photographic images or screen shots of sketches that you have captured to an image file, and wish to process further.

Image processing applications enable transformations of images. In this chapter you will learn how to manipulate images using *Processing* methods. Simple transformations include: changing the size of an image, or its canvas; changing from colour to black and white; inverting the colours in an image; and rotating the image by a given angle. More complex transformations are used to remove unwanted artifacts from images, such as flash reflections or red-eye, or they can be used to create a specific look or style for an image such as the soft focus style of portrait photography.

As you start to build a portfolio of your own sketches you should become familiar with at least one image processing tool. A good tool is *Gimp* which is available from `http://www.gimp.org` as a free download.

4.2 `PImage`

In *Processing*, the `PImage` class handles the display of images. The screen pixels of a sketch are represented internally using an instance of the `PImage` class.

4.2.1 Image formats and encoding algorithms

Images are stored in files that contain all the colour values for each pixel. A variety of different encoding algorithms can be used to convert an image into a representation suitable for storage in a file. Different image file formats employ different encoding algorithms. Therefore, when reading the contents of an image file, the format of the file determines how the bits in that file should be read and interpreted.

There is a general distinction between *lossless* and *lossy* encoding algorithms. Lossless algorithms encode the colour pixels exactly; the colour values of all pixels are recorded individually in the file. In *Processing* and similar systems, the colour values are stored in a 32-bit integer and the image dimensions determine the size of the file. For example, a 1024 x 768 image requires 1024 x 768 x 4 = 3145728 bytes of storage, about 3 Megabytes (MB). Image formats that use lossless encodings include GIF and PNG.

Lossy encoding algorithms are used to compress the image information into fewer bits. Information is lost in the compression of information. The information that is lost is that which the encoding algorithm determines is the least important from a perceptual perspective. So image encoders preserve the information in the image that is perceptually the most important. The higher the compression ratio (i.e. the fewer bits we use) the more information that is lost. If we keep increasing the amount of compression of an image, eventually it will be degraded to the point of being unusable in a design. However, the transformation caused by compression may also produce desirable artifacts that can be used creatively. Image formats that use lossy encoding algorithms include JPG, TGA and TIF.

Processing can load the following file formats: GIF, JPG, TGA, PNG.

4.2.2 `PImage` methods

The methods that are available in the `PImage` class are listed in Table 4.1.

The low-level storage for image data in a `PImage` is a member variable called `pixels[]`. Colour is encoded as RGB values and stored using the *Processing* `color` datatype as in Chapter 1.

There are many ways to define a new image. To make a new `PImage` we can do one of the following:

```
PImage myImage1 = loadImage("myImage.jpg");
PImage myImage2 = createImage(512,512,RGB);
PImage myImage3 = myImage2.get(256,256,100,100);
PImage myImage4 = new PImage(512,512);
```

The first image is constructed from a file in JPG format that is in the `data` directory of the current sketch. The second is a new blank image with dimensions 512 x 512. The third image is a copy of a 100 x 100 square patch of `myImage2` with the patch's top left corner at the copied image's centre point. The last image is a new blank image with dimensions 512 x 512. This produces an image that is the same as `myImage2`.

Fields	description
width	Image width
height	Image height
pixels[]	Array containing the colour of every pixel in the image
Methods	
loadPixels()	Loads the pixel data for the image into its pixels[] array
updatePixels()	Updates the image with the data in its pixels[] array
get()	Reads the colour of any pixel or grabs a rectangle of pixels
set()	Writes a colour to any pixel or writes an image into another
copy()	Copies a region of pixels from one image into another.
resize()	Changes the size of an image to a new width and height
mask()	Masks part of the image from displaying
blend()	Copies a pixel or rectangle of pixels using different blending modes
filter()	Converts the image to grayscale or black and white
save()	Saves the image to a TIFF, TARGA, PNG, or JPEG file
Constructors	
PImage()	Default constructor
PImage(width, height)	New image with specified width and height
PImage(img)	New image copying img
Non-PImage methods	
createImage()	Create a new (empty) image with specified width, height, format
loadImage()	Load an image from a file into a new PImage object
image()	Display all or part of an image on the display window
tint()	Sets the fill value for displaying images
loadPixels()	Loads the pixel data for the display window into the pixels[] array
updatePixels()	Updates the display window with the data in the pixels[] array

Table 4.1: Image processing methods in *Processing*. All of these methods manipulate a rectangular block of pixels.

4.3 Image display

The *Processing* display window itself is an instance of the `PImage` class. It is an implicit instance; we do not need to reference it using a variable. The pixel data of the display window is stored in the global array `pixels[]`, and many of the `PImage` methods are replicated as global methods that act on the display window (e.g. `blend()`, `copy()` and `filter()`).

To display an image we need to copy its pixels to the display window. We do this using the `set()` or `image()` methods:

```
PImage myImage = loadImage("image.jpg");
size(myImage.width, myImage.height);  // set display window dimensions
set(0, 0, myImage);                   // load image to display window
image(myImage, 0, 0);                 // equivalent to previous line
```

We must set the *Processing* display window to the correct size before displaying an image. The screen does not automatically resize itself when we use `set()` or `image()`.

The `set()` method accepts three arguments; the first two are a screen coordinate and the third is either a `color` value (to set the individual pixel) or a `PImage` to copy the image rectangle to the screen at the specified point. We can use the `set()` method for both images and individual pixels.

The `image()` method also accepts three arguments, but in a different order. The first argument is a `PImage` and the second two arguments are the position on the screen to display the image.

You will recall that rectangles can be displayed using the `rect()` method. The position of the rectangle given its `(x,y)` coordinates is determined by the `rectMode()` which can be `CORNER`, `CORNERS` or `CENTER`. Images have the same type of control over the meaning of the coordinates. We use the `imageMode()` method to tell *Processing* to place the image's top left corner at specified coordinates (`CORNER` and `CORNERS`), or to centre the image on the specified coordinates (`CENTER`). `CORNER` and `CORNERS` differ in that the former allows you to specify the image's width and height, whereas the latter allows you to specify the coordinates of the bottom right corner.

Figure 4.1 shows an image, taken from the Creative Commons[1], which we use to illustrate various manipulations in the following sections. The dimensions of the image are 600×800 pixels.

4.3.1 Image crop

We can use the `copy()` method to crop an image to a defined rectangular region:

```
copy(myImage, 300,550,90,100, 0,0,90,100);
```

The image `myImage` here is cropped from a rectangle with top left corner at `(300,550)` width 90 and height 100, and it is copied to the display window, as shown in Figure 4.2.

4.4 Image transformation

4.4.1 Image scaling

The `image()` method accepts two additional arguments which are the width and height to display the image. This can be different from the dimensions of the image. For example, for an image `myImage` with dimensions 600×800 pixels, we can alter the display dimensions using:

```
image(myImage, 0, 0, 300, 400);
```

In this example we have shrunk the image by a factor of 2, preserving the ratio between the width and the height (the aspect ratio).

[1] `http://creativecommons.org/about/`

Figure 4.1: We can use *Processing* to load and display images such as this image, by jimg944 at `http://flickr.com`, which is under a Creative Commons attribution licence.

Figure 4.2: Using the `copy()` method we can crop a small region from an image.

The following example changes the aspect ratio from 1:1 to 2:1.

```
image(myImage, 0, 0, 600, 400);
```

When we change the dimensions of an image we are performing a complex signal processing operation. If we stretch the image too much it will become pixelated[2], because the information in the image is too sparse to fill the new space. If we compress the image too much it becomes unrecognisable, because all the information is packed into a small space. The effects of scaling an image in an extreme way can lead to very interesting and creative results.

Figure 4.3 shows the result of combining the operations of cropping and scaling on our example image. The crop operation extracts a small portion of the image from Figure 4.1 and the `image()` method is used to scale the cropped portion by 500% in the width and height dimensions.

Figure 4.3: Using the `image()` method we can scale a cropped image back to a suitable size.

Learning activity

Select an image from the internet or your computer's filespace, and load it into a new `PImage` instance using the `loadImage()` method. If you do not have any images there are millions of suitable ones available on the internet. Some of these are specifically made for free use. For example, see `creativecommons.org` for a large repository of freely available images that you can download and even use in your own designs.

Shrink and stretch the image in each of its dimensions using the `image()` method. Try scaling the image by:

```
(2*myPimage.width, myImage.height)
(myPimage.width, 2*myImage.height)
(2*myPimage.width, 0.5*myImage.height)
(0.5*myPimage.width, 2*myImage.height)
```

[2] See Exercise 1 at the end of the chapter for more information about pixelation.

Save one of the new scaled images using the following command:

```
save("myNewImage.jpg");
```

This will make a new image file on disk.

Load the new image file that you created and repeat the stretching operations above. This sequence of operations, process / save / load / re-process, is called a feedback loop. We can save and re-process the image many times to create a wide range of visual effects.

4.4.2 Coordinate system transformations

In contrast to the `set()` method, the `image()` method is affected by *Processing*'s 2D and 3D coordinate system transformations. This means that a combination of `translate()`, `scale()` and `rotate()` can be used before calling the `image()` method, and the image will appear translated, scaled and rotated as specified.

`scale()`

For example, we can perform the scaling operations we discussed above using the `scale()` method:

```
PImage img = loadImage("oasis.png");
size(img.width, img.height);
scale(1, 0.5); // Scale the y-axis
image(img, 0, 0);
```

`translate()`

We can paste an image many times in different locations using either different coordinates with each call to `set()` or `image()`. Alternatively, it is often convenient to use coordinate transformations to perform a series of position operations.

Figure 4.4 shows the sequence of operations of scaling the coordinate system by 0.25 in each dimension, then pasting the image at four locations in a line. Translation by the image width translates by the scaled image's width, so the images are placed in a contiguous sequence.

```
PImage img = loadImage("oasis.png");
size(img.width, img.height/4);
scale(0.25, 0.25);
image(img, 0, 0);
translate(img.width,0);
image(img, 0, 0);
translate(img.width,0);
image(img, 0, 0);
translate(img.width,0);
image(img, 0, 0);
```

Figure 4.4: Repeating an image by coordinate system transformations using the `scale()` and `translate()` methods.

`rotate()`

We can also rotate an image using the `rotate()` method as in Figure 4.5.

Figure 4.5: Coordinate system transformation of an image using the `translate()` and `rotate()` methods.

```
PImage img = loadImage("oasis.png");
int SZ = max(img.width, img.height);
size(SZ, SZ);
translate(width/2, height/2);
rotate(PI/2);
translate(-width/2, -height/2);
image(img, 0, 0);
```

In this example the screen size is set to the maximum dimension of the image, to allow the rotated image to fit on the screen. The `translate()` method is called twice, the first call translates the origin to the centre of the screen, then the rotation

is performed around the origin, and the second call to `translate()` restores the origin to its starting position (0,0).

4.4.3 3D transformations

We may also perform coordinate system transformations in 3D before calling the `image()` method. This has the same effect of texture mapping onto a plane that is the same size as the image and can be used as a substitute for texture mapping using `beginShape()` and `texture()` when it is simpler or more convenient to do so.

Figure 4.6 shows a set of 3D transformations of an image forming a box.

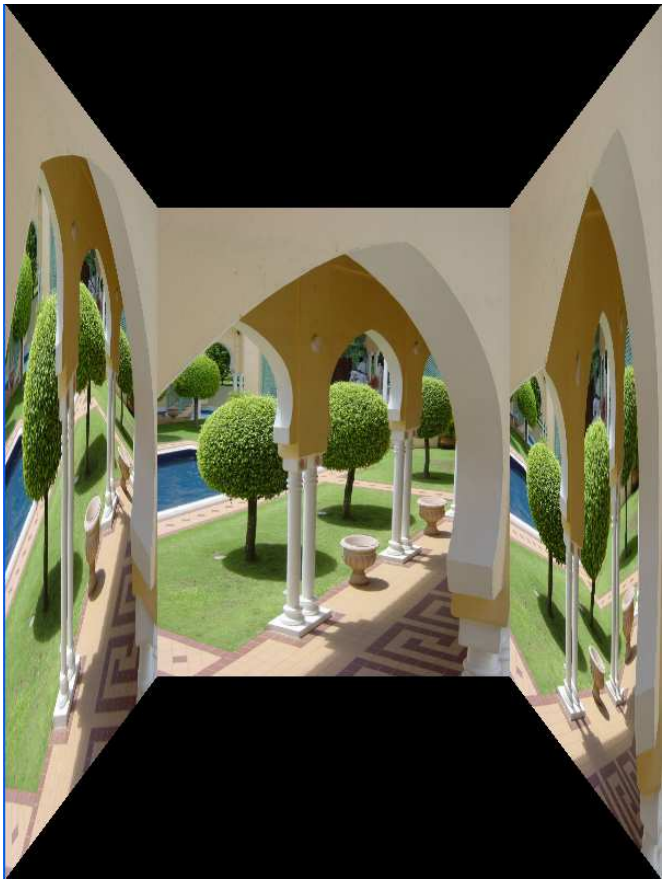


Figure 4.6: 3D coordinate system transformation of an image to form a box.

To use 3D transformations we must specifiy `P3D` in the `size()` method. In this example we first rotate the image around the `y` axis by `PI/2` radians and paste. Then we restore the coordinate system by undoing the rotation. Next we translate to a depth that is the same as the image width and paste the image. Then, remaining at that depth, we translate across the width of the image, rotate around the `y` axis by `-PI/2` and paste the image again:

```
PImage img = loadImage("oasis.png");
size(img.width, img.height, P3D);
background(0);
rotateY(PI/2);
image(img, 0, 0);
rotateY(-PI/2);
translate(0,0,-width);
image(img, 0, 0);
translate(width,0,0);
rotateY(-PI/2);
image(img, 0, 0);
```

Using the 2D and 3D coordinate transformations in *Processing*, an image composition of great complexity can be constructed.

Learning activity

Load some of your digital photos into different `PImage` instances. Display all of your images together using an appropriate layout with the images scaled and translated into a tidy presentation, to create a 2D "photobook".

Now display your images as a 3D photobook. How can you use the extra dimension to display even more images?

4.5 Layers

Digital image editing software, such as Adobe *Photoshop*, uses a concept called layers to organise many image parts, one on top of the other in an arrangement known as a stack, that are flattened together to make a finished image. Layers can be transparent so that the parts are blended together when the image is flattened.

We can use the `image()` and `blend()` methods to achieve layers in *Processing*. The basic technique is to load several images into different `PImage` instances and then use the blend method to mix them together.

Using a combination of cropping, scaling, pasting, image transformations and blend, we can achieve much of the utility of a commercial digital image editing package, such as *Photoshop*, in just a few lines of *Processing* code. Figure 4.9 (page 101) shows the application of blending to the images in Figures 4.7 and 4.8.

To use the blend method, we need two image instances: the target image and the source image. The source is blended with the target image and the target image is updated to contain the blend:

```
PImage img1 = loadImage("medialab.jpg");
PImage img2 = loadImage("rainforest.jpg");
size(img1.width, img1.height);
img1.blend(img2, 0,0,img2.width,img2.height, 0,0,width,height, SUBTRACT);
image(img1,0,0);
```

The `blend()` method takes 10 arguments in addition to the instance from which it is

Figure 4.7: Image of a rainforest by tauntingpanda from the Creative Commons repository at `http://flickr.com`.

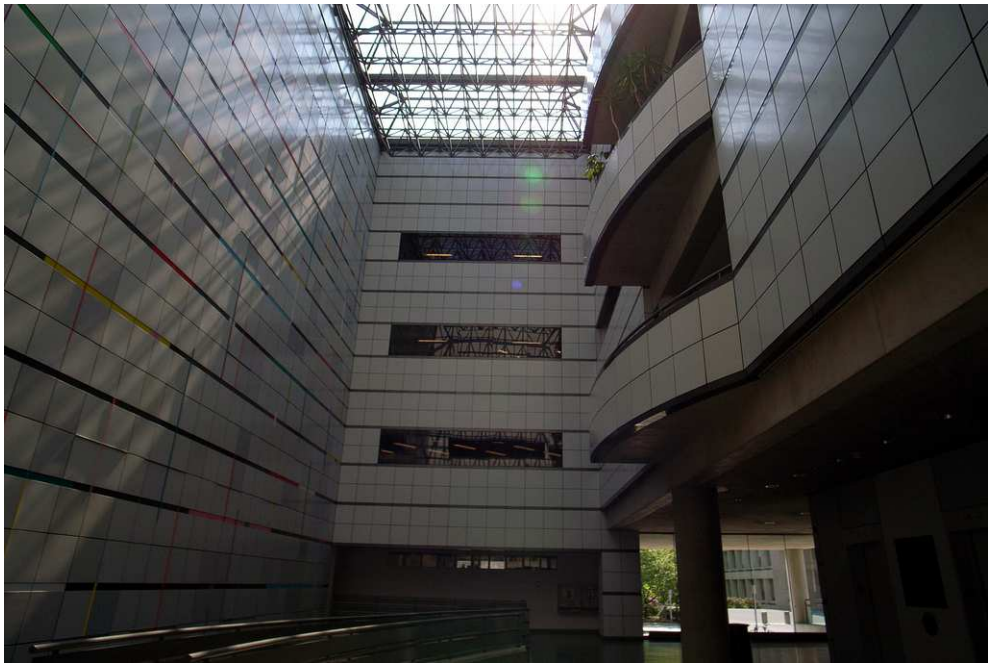


Figure 4.8: Image of the Weisner building at MIT in Cambridge, Massachusetts from the Creative Commons repository at `http://flickr.com`.

called (`img1`). The first argument is the source image, the image with which to blend the target image. The top left corner coordinates of a rectangular blend region in the source image are the next two parameters, followed by the width and height of this region. The next four parameters define the region of the target to blend with the source region. Again the parameters are the coordinates of the top left corner of the region (in the target image) and the width and height of the region.

The final argument of the `blend()` method is the blend MODE. This defines which algorithm will be used for blending. The blend modes are shown in Table 4.2.

MODE	description
BLEND	Linear interpolation of colours: C = A*factor + B
ADD	Additive blending with white clip: C = min(A*factor + B, 255)
SUBTRACT	Subtractive blending with black clip: C = max(B - A*factor, 0)
DARKEST	Only the darkest colour succeeds: C = min(A*factor, B)
LIGHTEST	Only the lightest colour succeeds: C = max(A*factor, B)
DIFFERENCE	Subtract colors from underlying image
EXCLUSION	Similar to DIFFERENCE, but less extreme
MULTIPLY	Multiply the colors, result will always be darker
SCREEN	Opposite multiply, uses inverse values of the colors
OVERLAY	A mix of MULTIPLY and SCREEN. Multiplies dark values, and screens light values
HARD_LIGHT	SCREEN when greater than 50% gray, MULTIPLY when lower
SOFT_LIGHT	Mix of DARKEST and LIGHTEST. Works like OVERLAY, but not as harsh
DODGE	Lightens light tones and increases contrast, ignores darks
BURN	Darker areas are applied, increasing contrast, ignores lights

Table 4.2: Blend MODES for the `blend()` method.

Figure 4.9 shows the effect of combining two images using the `blend()` method with the blend mode set to `ADD`. Figures 4.10, 4.11, 4.12 show the same two images blended using the `LIGHTEST`, `DARKEST` and `SUBTRACT` blend MODES respectively.

Using the same technique, we can blend many images together. Figure 4.13 (page 103) shows the result of blending all three of the Creative Commons licensed images in this chapter. To do this, we simply perform a second blend on the composite blend of the first two images. Each blend operation stores the result back to `img1` in this sketch:

```
PImage img1 = loadImage("oasis.jpg");
PImage img2 = loadImage("rainforest.jpg");
PImage img3 = loadImage("medialab.jpg");
size(img1.width,img1.height);
img1.blend(img2,0,0,img2.width,img2.height,0,0,width,height,LIGHTEST);
img1.blend(img3,0,0,img3.width,img3.height,0,0,width,height,LIGHTEST);
image(img1, 0, 0);
```

Figure 4.9: img.blend(img2,0,0,img2.width,img2.height,0,0,width,height,ADD);

Figure 4.10:
`img1.blend(img2,0,0,img2.width,img2.height,0,0,width,height,LIGHTEST);`

Figure 4.11:
`img1.blend(img2,0,0,img2.width,img2.height,0,0,width,height,DARKEST);`

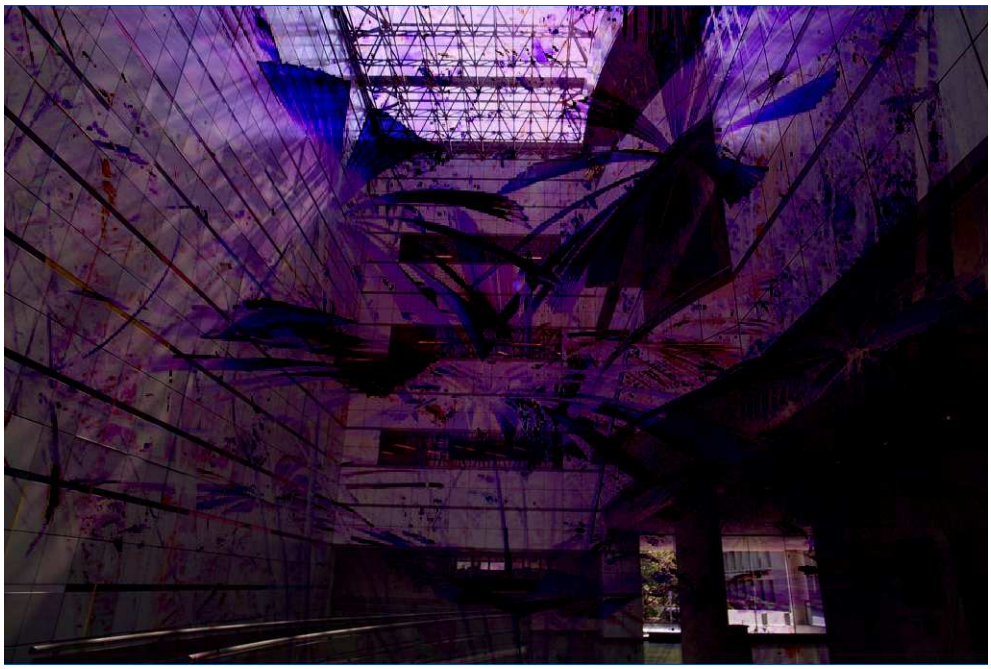


Figure 4.12:
`img1.blend(img2,0,0,img2.width,img2.height,0,0,width,height,SUBTRACT);`

Figure 4.13: `img1.blend(img2,0,0,img2.width,img2.height,0,0,width,height,ADD);`
`img1.blend(img3,0,0,img3.width,img3.height,0,0,width,height,ADD);`

4.6 Summary

In this chapter we first looked at how images are represented as files on disk and how to load the files into *Processing*. We looked at the methods for image manipulation and how these are used to perform simple image operations such as crop, paste and scale. We also looked at how image display positions can be manipulated using the same transformations we use with 2D and 3D graphics. Finally, we considered how to work with layers by using the `blend()` method.

You should now be able to:

- make use of the methods in the *Processing* class `PImage` to manipulate images
- transform images using translation, rotation, scaling and cropping, in order to make new images
- incorporate 3D into images
- use blending and layers to produce interesting images that are composites of existing ones.

4.7 Exercises

1. What is pixellation? Find out what you can about this phenomenon, which is specific to digital technology, and what techniques exist to avoid it. Why is it specific to digital technology?
2. In the blending example shown in Figure 4.9, which is the source image, and which is the target? Does it make a difference if they are the other way round? Illustrate this using two example images of your own. You should also try to find the images shown in this guide, from the `http://flickr.com` website, and apply the blending as shown, for yourself.

   Think also about the order in which the blending for Figure 4.13 is done, and work out whether this order has any impact on what the final image looks like. Does it have any impact on any properties (whether visibly detectable or not) of the final image?

Chapter 5

Sound

Essential reading

DuBois, R. L. and W. Thoben. `https://processing.org/tutorials/sound/`, online tutorial covering the *Processing* Sound library.

Additional reading

Shiffman, D. *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction, Second Edition* (Morgan Kaufmann, 2015). Chapter 20. [ISBN 0123944430].

Roads, C. *The Computer Music Tutorial* (MIT Press, 1996) [ISBN 0262680823]. Chapters 1–4, 9–10, 12–13, 18–19.

5.1 Introduction

In the physical world, vibrating objects in a physical medium are the sources of sound. Sound is a progressive wave that propagates spherically from the source through the transmission medium—usually air or water. Sound carries energy from the source to the receiver in a succession of changes in air pressure (called rarefactions and expansions) that cause the ear drum to vibrate. This change in air pressure over time is an acoustic waveform.

In the digital world, sound can be represented entirely in terms of numbers and computational functions. These functions can be chosen to emulate physical sources, such as a saxophone, drum or voice, or new sounds can be created that emulate no physical object.

As with digital photography, we have the option to take sound recordings of the physical world and store them digitally as samples. These samples can then be played back and manipulated, again via computational functions, to create sound effects or new musical *timbres* (the tone quality of the sound).

5.2 Digital audio

When we represent sound in a computer we refer to the data as digital audio. This is to distinguish it from pressure waves propagating in air or water. Digital audio is a series of many numbers, that are obtained by sampling the air pressure wave using a microphone and analogue-to-digital converter (ADC).

To manipulate digital audio we must use special software that understands the meaning of the numbers and can interpret them. *Audacity*, illustrated in Figure 5.1,

is a freely available digital audio editor with mixing and signal processing capabilities. It can be downloaded from `https://www.audacityteam.org/`.

5.2.1 Sampling

Sampling is the process of measuring a sound signal at regular intervals. The sample rate (SR) of the audio signal is the number of samples per second. For CD-quality audio the sample rate must be 44100 samples per second of audio.

When sound is converted from the physical world into digital form, a *bit depth* has to be chosen. The bit depth is the number of bits of information in each sample, and corresponds to the resolution of the sample. It is most usual in sound to use either 8 bits, 16 bits or 24 bits. A depth of 8 bits, for example, provides a resolution of $2^8 = 256$ different values that a single sample can take.

Most sound implementations use "shorts", which are signed 16-bit integers with values in the range -32768 to 32767; that is $-(2^{16})/2$ to $(2^{16})/2 - 1$. We may scale the `short` to a `float` in the range `[-1,1]` by multiplying with `1.0/32768.0`. In many applications it is convenient to use the `float` representation.

Digital audio is organised into *tracks*, where each track represents an individual piece of audio (which can be either *stereo* or *mono*). *Channels* refer to individual speakers, so a mono track requires only a single channel, whereas a stereo track requires two different channels. Figure 5.2 shows a screen shot of stereo audio. The waveform is displayed up close so that we can see the detail of individual samples. The selected portion of the waveform is 0.155 seconds.

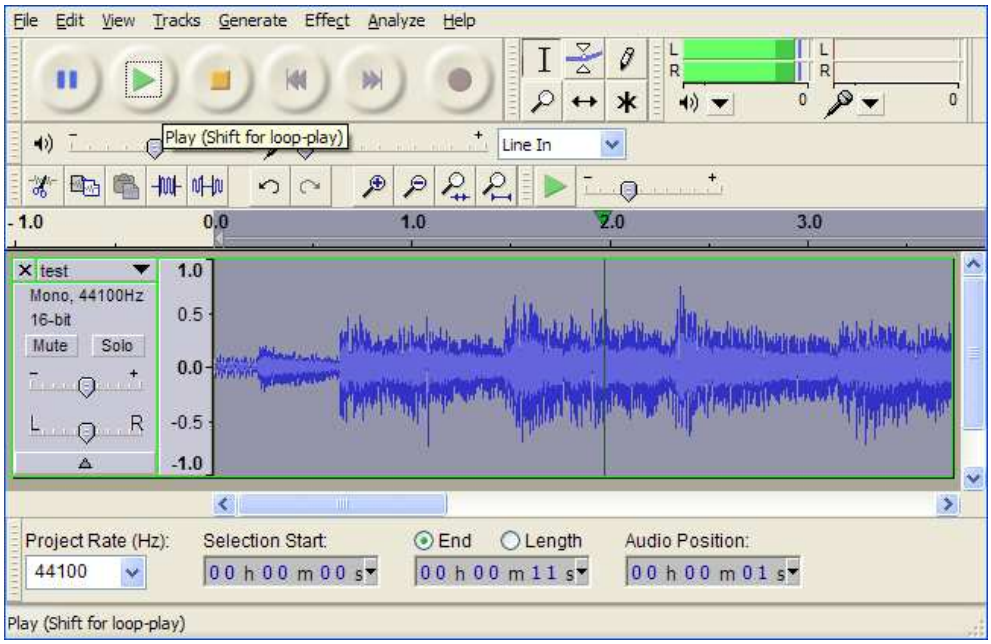


Figure 5.1: *Audacity*, a digital audio editor that is freely available. In this program we can inspect digital audio files and construct new files by mixing and processing using digital signal processing methods.

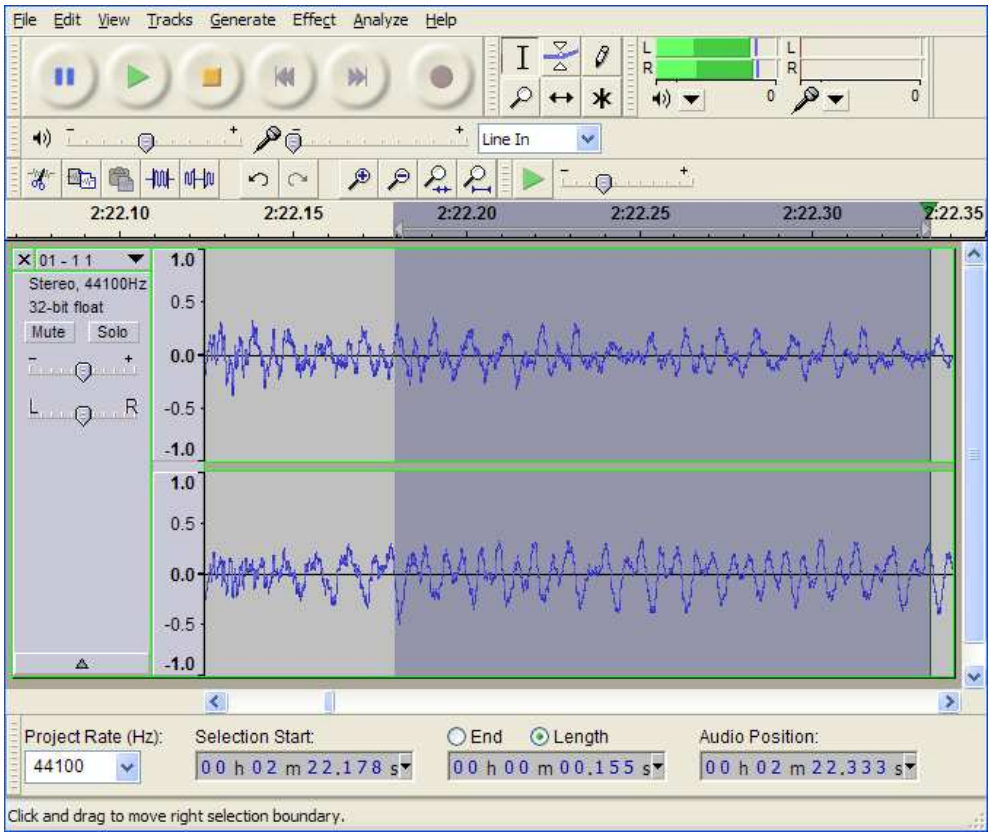


Figure 5.2: A stereo file displayed in *Audacity*. This view of the tracks is magnified relative to Figure 5.1; the selected region is only 0.155 seconds in duration.

Amplitude

In the previous two examples the amplitude was displayed in the range $[-1, +1]$. Some editors also display power, the sum of the squares of the signal values.

The human ear relates the loudness of a sound to the amplitude by the transformation of a logarithm:

$$loudness = 20 \times \log_{10}(amp)$$

This measure is called a deci-Bel(dB) and is one tenth of a Bel, which is $200 \times \log_{10}(amp)$.

10dB is approximately the change in sound pressure level for a perception of twice the loudness. This corresponds to multiplying the amplitude by 3.162, which is $\sqrt{10}$. To get 20dB (four times the loudness) we need to square the multiplier on the amplitude which gives 10.0 times the amplitude. For eight times the loudness, we need 31.62 times the amplitude and so forth.

Mixing

Mixing is the process of combining waveforms to make a new waveform, such as combining the bass track, drum tracks and vocal tracks in a pop song. Each track is a separate waveform, and mixing combines them into either a single waveform (mono), two waveforms (stereo) or more for theatre and surround sound systems.

The process of mixing is the arithmetic process of addition. We must be careful when mixing waves that the sums of amplitudes do not exceed the minimum and maximum values. For floating point representation this is $[-1, 1]$.[1]

To avoid overflow, we divide each signal by the total number of signals to be added before adding them together.

Figures 5.3 and 5.4 show the mixing process for two sinusoidal waveforms. These waveforms have the same wave shape (the sinusoid) but they occur at different rates, or *frequencies*. In each example the second sinusoid has twice the frequency of the first. The first undergoes three complete cycles in the 100 samples shown and the second undergoes six complete cycles in 100 samples.

In the second of these two examples, the second sinusoid is shifted to the left by $\pi/2$ radians. The resulting waveform is different to the one shown in the first example. This shows that the phase of waveform components affects the shape of the resulting waveform.

We compute energy from the amplitude by summing the squares of all the samples and dividing by the length of the window. In this case the window is 100 samples long. Note that phase shifting does not affect the overall energy of the mixture of the two sinusoids.

[1] Note that this notation, using [,], is for a *closed* interval, which means that the range includes both -1 and 1.

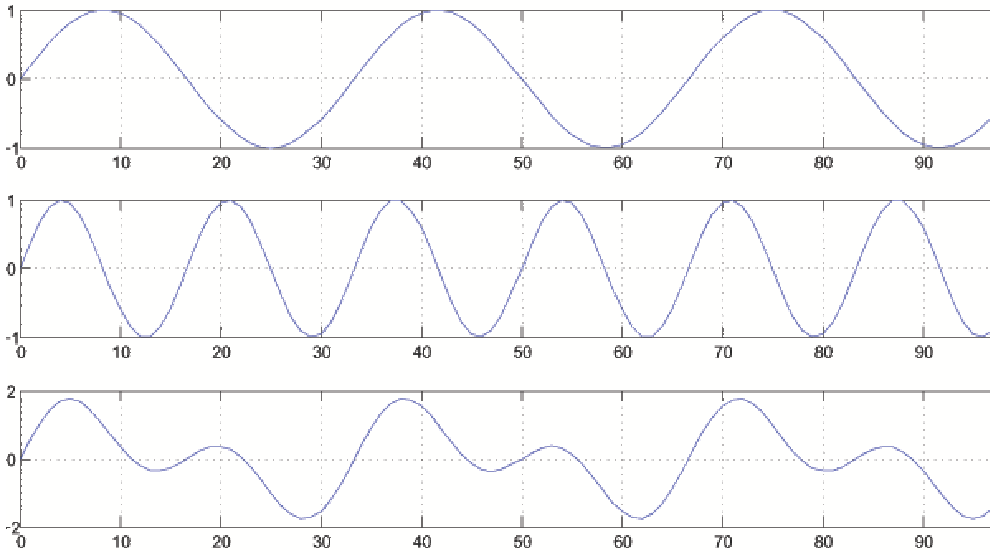


Figure 5.3: Mixture of sinusoidal waveforms. The sum of two sinusoids makes a new complex waveform that has a greater amplitude than the component sinusoids. We scale the sinusoids before summing to avoid overflow when using integer representations.

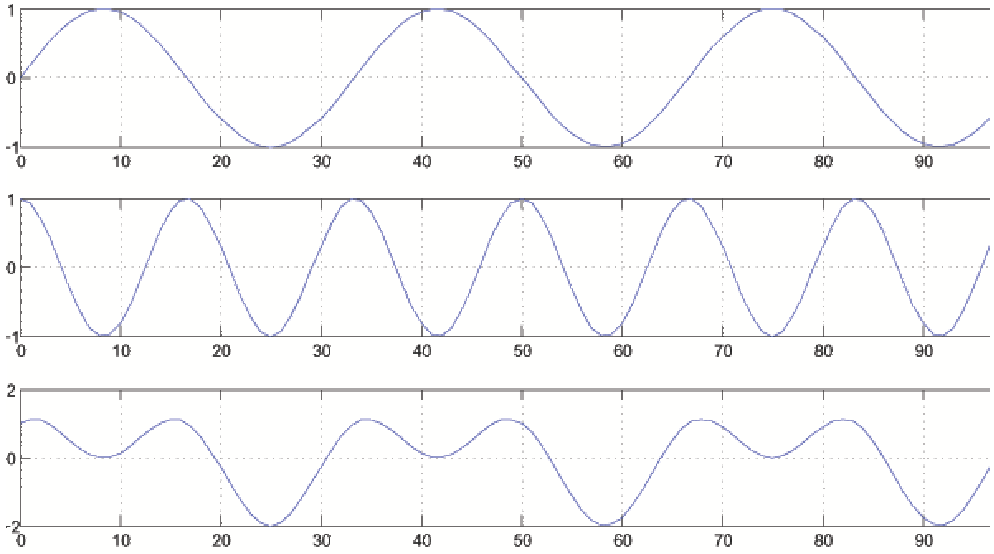


Figure 5.4: Mixture of two sinusoids with the same frequencies as the previous example. The second sinusoid has a phase offset of `PI/2`.

Frequency

Sound is made of frequency components. These can be simple tones, called sine waves, or other atomic sound waves, such as noise bursts.

Frequency is a property of a waveform that repeats; these are called periodic sounds. When a waveform is periodic the number of repetitions (cycles) per second is called the frequency. Cycles per second are called Hertz; 1 Hz is one cycle per second and 1 kHz is a kilo-Hertz which is 1,000 cycles per second.

Naturally occurring sound contains many frequency components often extending beyond the range of human hearing, both below and above. The sounds that we can hear consist of frequency components that are in the range 20 Hz to 20,000 Hz.

Sampling theory tells us that the highest *frequency* component that we can measure for a sampling rate of SR is SR/2. This frequency is called the Nyquist frequency and it tells us that to measure a signal up to 10 kHz, we require a sample rate of 20 kHz; 10 kHz is the Nyquist frequency of a 20 kHz sampling rate.

Sample rates are chosen for their specific application. For telephone speech the sample rate is usually 8 kHz. Thus the highest frequency represented is 4 kHz. Most of the important frequency information for speech is below 4 kHz. For CD-quality, however, we require 44.1 kHz (44100 Hz) and for studio-master quality we would normally use sample rates of 48 kHz or 96 kHz, with bit depths of 24-bit or 32-bit.

A sound that is periodic in the audible range is heard as a single pitched percept. However, there are typically many frequency components that make up a periodic waveform, so which one is heard as the pitch of the sound?

A periodic sound has a spectrum that may be represented as equally spaced frequency components:
$amp1 \times sin(freq \times n) + amp2 \times sin(2 \times freq \times n) + amp3 \times sin(3 \times freq \times n) + \ldots$
The components have frequencies that are integer multiples of the first `freq` in the series. This lowest frequency is called the *fundamental* frequency, and—if it is in the audible range—it is the frequency that is heard by the ear as the pitch of the entire waveform. The remaining frequency components are fused into the perception of *timbre*, which is the tone quality of the sound.

Discrete Fourier transform

The Fourier transform, and the associated family of mathematical tools, are the basis of most digital image processing and digital audio processing algorithms. They are the tools that allow us to manipulate digital media via the concept of frequency components. A finite Fourier Transform is performed on a series of N values from a sequence, such as a sampled digital audio waveform, and produces the amplitudes and phases of N/2 sinusoidal components that, when summed together, reproduce the sampled waveform exactly.

The main contribution of Fourier theory is that the waveform representation of a sound is equivalent to the frequency representation, phases and amplitudes over frequency. The sampled sound gives a finite number of discrete frequencies, corresponding to those in the discrete Fourier transform (DFT). This representation is called the spectrum. For sounds with a pitch, the human ear is not sensitive to the phase of these sinusoidal components. Therefore the frequency representation is

simply amplitude as a function of frequency. It is usual in digital audio editors to display the amplitude-only spectrum.

Figure 5.5 shows the time-frequency spectrum of a sound file. This view is a series of DFTs of the sound taken at regular intervals, the frame rate (FR). The DFT calculates the contribution to the sound of a set of sinusoids that are equally spaced in frequency from 0 Hz to the sample rate. The number of sinusoids determines the sampling density for the spectrum and is usually a power of two (512; 1024; 2048; . . .). This is the window on the audio signal that will be analysed. The window is advanced one half of this value for the next frame, the DFT performed, and so on.

For sounds that we hear as fixed musical pitches, the waveform of each note is periodic. This means that it repeats at regular intervals determined by the note's frequency. This repetition also gives rise to the step ladder that we see in Figure 5.5 for each note played.

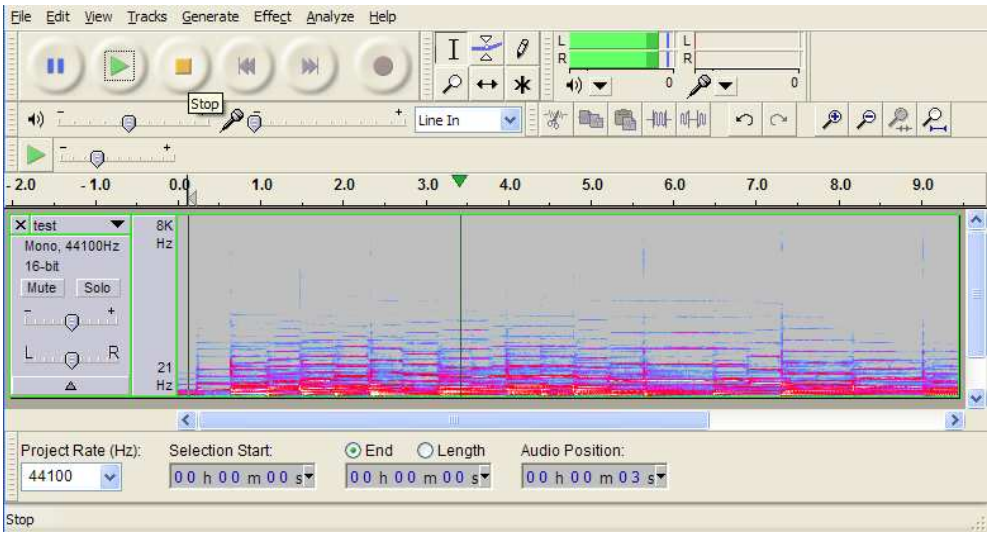


Figure 5.5: The spectrum of an audio signal displayed in *Audacity*. This graph shows the magnitude, `abs(amplitude)`, in a range of frequency bands as a function over time. The spectrum view shows us that sound is made of many frequency components that vary in magnitude over time.

5.3 Audio file formats

To persist audio clips, or store very long audio clips, we must write the sampled audio to a file. To do this we use a file format to store additional information such as the sample rate and bit depth of the sound. Common formats include aac, mp3, wav, wma, aif(f) and ogg.

Some of these formats contain compressed audio and require an encoder to create them.

Uncompressed digital audio data is often referred to as being in Pulse Code Modulation (PCM) format. PCM is a type of sampling process. CD-quality audio is in PCM format. With a sample rate of 44.1 kHz, a bit depth of 16-bit, and two

independent channels for stereo, it occupies 10MB of disk space per minute of audio. By contrast, compressed audio is about 1-2MB per minute depending on the compression ratio. The rate of a compressed audio signal is measured in kbits per second. For CD quality (uncompressed) audio, there are 1411 kbits per second. MP3 has a bit rate of between 8 and 320 kbits per second, depending on the desired quality. The bit rate does not change the sample rate of the audio signal, it only tells us how many bits are needed to reconstruct the full-bandwidth signal using the appropriate decoder.

When processing audio files, it is good practice to keep as much information as possible through any processing chain, with any loss of information (through compression by encoding, downsampling, etc.) happening only at the very end of the process.

5.4 Audio in *Processing*

In order to play, generate and manipulate audio files in *Processing*, we need to use the *Sound* library.[2]

Learning activity

Before proceeding, check that the *Sound* library is installed on your computer. To do this, open the *Processing* editor and click on the `Sketch` menu. Hover your mouse pointer over `Import Library...` and check whether `Sound` appears in the list of installed libraries in the central part of the pop-up menu. If you do not see it, select `Add library...` from the top of the pop-up menu to open the *Contribution Manager*. Scroll down the list of libraries until you find the one named `Sound`, click on it, then press the `Install` button at the bottom of the screen. The library will then be downloaded and installed within a few seconds.

Once the *Sound* library is installed, include the following import line at the start of your sketches in order to use it:

```
import processing.sound.*;
```

5.5 Playing a PCM sound file

Figure 5.6 shows how to use the `SoundFile` class in the *Sound* library to load and play back a sound file. The constructor for this class is called with the filename of the audio file. As with any other digital asset associated with a *Processing* sketch, the audio file should be located in the sketch's `data` folder.

Note that the `SoundFile` constructor takes two arguments. The first argument should be a reference to the current sketch, represented by the `this` keyword.

[2]The *Sound* library was developed by the Processing Foundation and introduced in *Processing* version 3.0. Older versions of *Processing* require the use of a third-party library as an alternative, such as *Minim* (`http://code.compartmental.net/minim/`), *Sonia* (`http://sonia.pitaru.com/`), or *Beads* (`http://www.beadsproject.net/`). The example sketches described in this chapter assume that you are running *Processing* version 3.0 or higher and have the *Sound* library properly installed.

Similarly, most other classes in the *Sound* library also take `this` as the first argument of their constructors.

```
/**
 * Simple example of using the Processing Sound
 * library to play a sound file
 */

import processing.sound.*;
SoundFile mySample;

void setup() {
  size(512, 512);
  background(255);

  // load a soundfile from the sketch's data
  // folder and play it back
  mySample = new SoundFile(this, "piano.wav");
  mySample.play(); // play the sample once
}

void draw() {
  if (mySample.isPlaying()) {
    // white background when the sample is playing
    background(255);
  }
  else {
    // black background when sample has finished
    background(0);
  }
}
```

Figure 5.6: Digital audio is stored in an audio file and can be loaded into memory for sound playback. In this example, the whole sound file is played back. The background colour changes from white to black when playback has ended.

5.5.1 Adjusting the playback rate

Pitch shifting is achieved by altering the speed of playback of a sample. For a soundfile with sample rate of SR, playing back at a rate of SR/2 will halve the playback frequency. This shifts all the frequencies down by an octave and doubles the duration of the sound file.

Similarly, doubling the playback speed (2*SR) will shift all the frequencies up by an octave and will halve the duration of the audio file.

It is usual to discuss frequency in terms of *pitch class* when working with musical pitch. In Western tonal music, pitch is selected from a set of 12 pitch classes. Each pitch class can occur at many different pitch heights (the octaves). So there are 12 pitches and about 10 audible octaves, where each octave is divided into the 12 pitch classes. The centre frequencies of these pitch classes are computed as follows:

$$A_0 = 55Hz, A\sharp_0 = A_0.2^{(1/12)}, B_0 = A_0.2^{(2/12)}, C_0 = A_0.2^{(3/12)}, \ldots$$

The relation $2^{(n/12)}$ is the frequency multiplication factor for a change in frequency of n 12-th of-an-octave steps. Such a division of an octave into 12 equal parts is known as *twelve-tone equal temperament*.

The SoundFile class

Table 5.1 shows the list of all of the available methods of the `SoundFile` class. To play back a sound at a different sample rate, we used the `rate()` method. This method takes a single floating point number as an argument, which is a relative playback rate. For example, an argument of 1.5 will result in the sample playing one and a half times faster than normal, while 0.75 will be three quarters of the original speed. A slower rate means a longer duration.

Learning activity

Modify the code in Figure 5.6 to play back at half speed (double the duration).

Modify the code to play back at double speed (half the duration).

In both cases, how does the output sound in comparison to the original version?

Method	*Description*
channels()	Returns the number of channels of the soundfile.
cue()	Cues the playhead to a fixed position in the soundfile.
duration()	Returns the duration of the soundfile in seconds.
frames()	Returns the number of frames of this soundfile.
play()	Starts the playback of the soundfile. Only plays to the end of the audiosample once.
jump()	Jump to a specific position in the soundfile while continuing to play.
pause()	Stop the playback of the file, but cue it to the current position so that the next call to play() will continue playing where it left off.
isPlaying()	Check whether this soundfile is currently playing.
loop()	Starts playback which will loop at the end of the soundfile.
amp()	Change the amplitude/volume of this audiosample.
pan()	Move the sound in a stereo panorama. Only works for mono soundfiles!
rate()	Set the playback rate of the soundfile.
stop()	Stops the playback.

Table 5.1: The `SoundFile` class's available methods. We can play back and manipulate digital audio files by calling these methods with suitable parameters. The information in this table is from the *Sound* library online reference at `https://processing.org/reference/libraries/sound/SoundFile.html`, used under the Creative Commons CC BY-NC-SA 4.0 license.

5.6 Digital audio synthesis

We generate sound digitally by computing its waveform. This is called digital audio synthesis. A number of waveform functions are commonly used in audio synthesis. They are the sine wave, square wave, triangle wave, sawtooth wave, pulse wave and white noise. Figure 5.7 illustrates the first four of these. The *Sound* library provides classes to automatically generate a variety of commonly used waveforms, including all of those just mentioned. We can also use any other kind of waveform by writing code to generate an appropriate data array to represent the waveform and using it to

construct an `AudioSample` object. In the following sections we will first look at using the inbuilt waveform classes, then at manually generating our own waveforms.

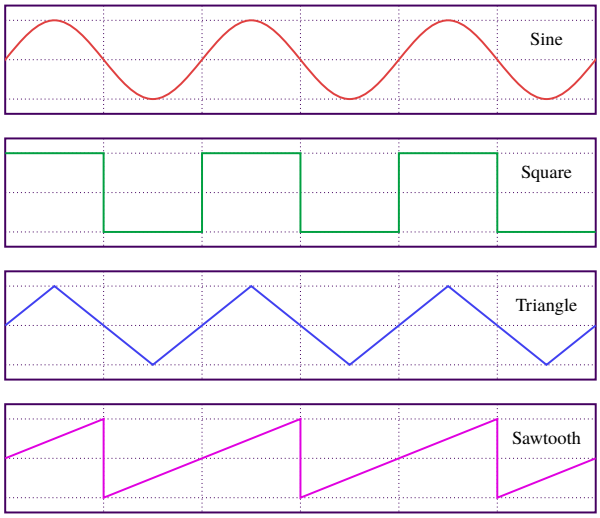


Figure 5.7: Four waveforms commonly used for audio synthesis: sine wave, square wave, triangle wave and sawtooth wave. Also commonly used are pulse waves and noise bursts which are not shown in this figure. Image from `https://commons.wikimedia.org/wiki/File:Waveforms.svg`, used under the Creative Commons CC BY-SA 3.0 license.

5.6.1 Synthesis using the inbuilt oscillator and noise classes

Table 5.2 lists the classes provided by the *Sound* library for generating commonly used audio waveforms.

Oscillators	
SinOsc	Sine Wave Oscillator
SawOsc	Saw Wave Oscillator
SqrOsc	Square Wave Oscillator
TriOsc	Triangle Wave Oscillator
Pulse	Pulse Oscillator
Noise	
WhiteNoise	White Noise Generator
PinkNoise	Pink Noise Generator
BrownNoise	Brown Noise Generator

Table 5.2: The *Sound* library's inbuilt classes for generating common kinds of waveform and noise.

Oscillators

A very basic example of using the `SinOsc` class to generate and play a sine wave oscillator is shown in Figure 5.8. Table 5.3 lists all of the methods provided by the `SinOsc` class. The other oscillators listed in Table 5.2 each provide a similar set of methods.

```
/**
 * Simple example of using the Processing Sound library
 * to play a Sine Wave Oscillator
 *
 * Based upon the example code in the Processing
 * online manual: https://processing.org/
 *   reference/libraries/sound/SinOsc.html
 */

import processing.sound.*;
SinOsc sin;

void setup() {
  size(512, 512);
  background(255);

  // create the sine wave oscillator
  sin = new SinOsc(this);
  // specify a frequency of 500Hz
  sin.freq(500);
  // and play it (oscillators play continuously until
  // the stop() method is called)
  sin.play();
}

void draw() {
}
```

Figure 5.8: Basic usage of the `SinOsc` class to generate and play a sine wave oscillator.

Method	*Description*
play()	Starts the oscillator.
set()	Set multiple parameters at once.
freq()	Set the frequency of the oscillator in Hz.
amp()	Change the amplitude/volume of this sound.
add()	Offset the output of this generator by given value.
pan()	Move the sound in a stereo panorama.
stop()	Stop the oscillator.

Table 5.3: The `SinOsc` class's available methods. The information in this table is from the *Sound* library online reference at `https://processing.org/reference/libraries/sound/SinOsc.html`, used under the Creative Commons CC BY-NC-SA 4.0 license.

Learning activity

Run the sketch shown in Figure 5.8 on your computer and listen to the output. Try changing the frequency of the sine wave to higher and lower values, and notice how this affects the generated sound.

Modify the sketch to use each of the other oscillator classes listed in Table 5.2, in place of the `SinOsc` class. Notice the difference in output sound in each case.

Experiment with playing multiple oscillators at the same time, with different frequencies and/or different waveforms.

The Digital Theremin

We can use some of the methods provided by the `SinOsc` class to simulate one of the earliest kinds of Electronic Musical Instrument (EMI). This instrument is called the *Theremin*, after its inventor Leon Theremin (b. 1896). The Theremin was invented in 1919, and first used in concert by Clara Rockmore at New York City's Carnegie Hall in the 1930s.

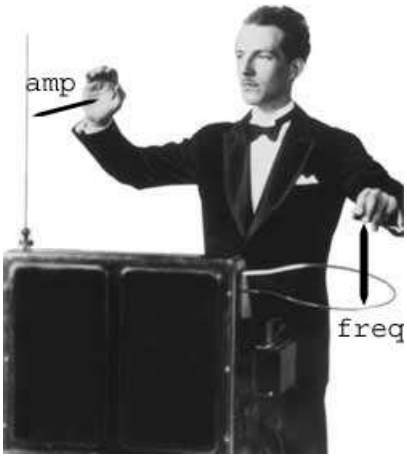


Figure 5.9: Leon Theremin playing his invention: *The Theremin*.

The Theremin contains a sine wave oscillator,[3] controlled by a human performer who uses their hands to affect an electric field around the device's antennae. Figure 5.9 shows Theremin playing his invention. In the figure the antennae are labelled `freq` and `amp`. Distance from the frequency antenna (left hand of operator) maps to decreasing frequency and distance from the amplitude antenna (right hand of operator) maps to decreasing amplitude.

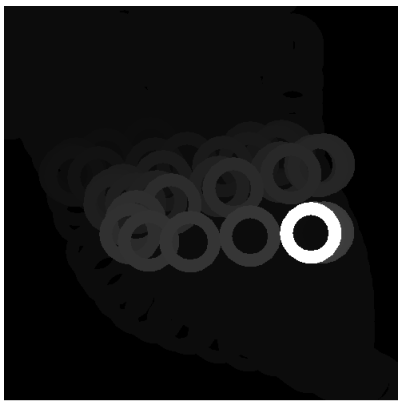


Figure 5.10: Digital Theremin control: the position of the white circle on the `x`-axis determines the frequency and the `y`-axis determines the amplitude of the sine wave oscillator.

We can make a digital version of the Theremin by extending the basic sine wave synthesis sketch shown in Figure 5.8 to allow us to use the mouse to control

[3] In fact, to produce sounds in the audible range, the Theremin uses two oscillators in an arrangement known as a *heterodyne oscillator*.

```
/**
 * Example of basic usage of SinOsc and SqrOsc oscillators
 * from the Processing Sound library.
 *
 * Move the mouse horizontally across the window to change the
 * output frequency, and vertically to change the amplitude.
 * Use keyboard buttons 1 and 2 to switch between the Sine Wave
 * Oscillator and the Square Wave Oscillator.
 */

import processing.sound.*;

SinOsc sin;
SqrOsc sqr;
float freq = 640.0;
float freqMin = 70.0;
float freqMax = 1200.0;
float x1 = 0.5;
float amp = 0.4;

void setup() {
  size(512, 512);
  background(0);
  stroke(255);
  noFill();
  strokeWeight(16);
  rectMode(CENTER);

  // Create a sine wave and a square wave oscillator
  sin = new SinOsc(this);
  sqr = new SqrOsc(this);

  // Set the initial amplitude and frequency of
  // both of the oscillators
  sin.set(freq, amp, 0.0, 0.0);
  sqr.set(freq, amp, 0.0, 0.0);

  // Start playing the sine wave oscillator
  sin.play();
}

void draw() {
  // Fade previous graphics on screen with a semi-transparent fill
  fill(0, 10);
  rect(256, 256, width, height);

  // Check for mouse interaction to change frequency and amplitude
  if (mousePressed) {
    // Set amplitude according to mouse Y position
    // -> map mouseY from 0 to 1
    amp = constrain(map(height-mouseY, 0, height, 0.0, 1.0), 0.0, 1.0);
    // -> set the oscillator amplitude to desired value
    sin.amp(amp); sqr.amp(amp);

    // Set frequency according to mouse X position
    // -> map mouseX from 0 to 1
    x1 = constrain(map(mouseX, 0, width, 0.0, 1.0), 0.0, 1.0);
    // -> map x1 to the specified frequency range
    freq = freqMin + (freqMax-freqMin) * x1;
    // -> set the oscillator frequency to desired value
    sin.freq(freq); sqr.freq(freq);
  }

  // Display shape on screen according to which
  // oscillator is currently active
  if (sin.isPlaying()) {
    ellipse(x1*width, (1.0-amp)*height, 60, 60);
  }
  else {
    rect(x1*width, (1.0-amp)*height, 60, 60);
  }

  // Output information about current sound
  println((sin.isPlaying()?"Sine":"Square")+": freq="+freq+", amp="+amp);
}

void keyPressed() {
  // Allow user to switch between the two oscillators
  if (key == '1') {
    sqr.stop(); sin.play();
  }
  else if (key == '2') {
    sin.stop(); sqr.play();
  }
}
```

Figure 5.11: The Digital Theremin uses a `SinOsc` oscillator. The mouse is used to control the amplitude and frequency of the oscillator. The 1 and 2 keys on the keyboard can be used to switch between a sine wave oscillator and a square wave oscillator.

frequency and amplitude. In the sketch shown in Figure 5.11, the mouse `x`-position is mapped to frequency and `y`-position to amplitude. The sine wave synthesizer is the same as in the previous example; we have added user control via the mouse input and by mapping screen coordinates to Theremin parameters. We use the oscillator's `freq()` and `amp()` methods to set these parameters accordingly. We have further extended the sketch to go beyond the capabilities of a standard Theremin by allowing the user to switch between a sine wave oscillator and a square wave oscillator. This is achieved by pressing the 1 and 2 keys on the keyboard respectively.

Figure 5.10 shows an example of the visual interface of the sketch, illustrating how the Theremin controller is moved around to change the amplitude and frequency of the sine wave oscillator.

Noise synthesis

In addition to the various oscillators, the *Sound* library also provides classes for generating three different types of noise (see Table 5.2).

Figure 5.12 shows a basic example of generating noise using the `WhiteNoise` class. This class creates a waveform comprising uniform random numbers in the range $[-1, +1]$. As with the oscillator classes, the amplitude of the output can be adjusted by calling the `amp()` method.

```
/**
 * Simple example of using the Processing Sound library
 * to play white noise
 */

import processing.sound.*;
WhiteNoise noise;

void setup() {
  size(512, 512);
  background(255);

  // create the white noise generator
  noise = new WhiteNoise(this);
  // and play it (the generator will play continuously
  // until the stop() method is called)
  noise.play();
}

void draw() {
}
```

Figure 5.12: Basic usage of the `WhiteNoise` class to generate noise with a flat power spectrum.

The *Sound* library also provides classes for generating *pink noise* and *Brownian (brown) noise*. While white noise has a flat power spectrum at all frequency intervals, pink and brown noise have more powerful components at lower frequencies.

Learning activity

Modify the sketch in Figure 5.12 to play white noise when key 1 is pressed, pink noise when key 2 is pressed, and brown noise when key 3 is pressed. Run the sketch and listen to the difference between these three types of noise.

Search online for further information about the difference between these three kinds of noise. How do their power spectra differ? Find examples of advantages of pink noise over white noise. Find examples of advantages of brown noise over white noise.

We can use noise synthesis to create many kinds of special effect and as a component of many musical instruments. For example, the sound of a flute contains the sound of the breath travelling across the mouthpiece; this is noise. Also, wind sounds consist of noise that is filtered so that only some of the frequencies are audible.

5.6.2 Synthesis by generating other waveforms

In addition to the inbuilt oscillators and noise generators, it is also possible to create and play any waveform we like by explicitly defining it within our sketch. To do this, we use the *Sound* library's `AudioSample` class. This is a more general version of the `SoundFile` class that we have already seen.[4]

The `AudioSample` object contains an array of floating point numbers that defines the waveform. The simplest way to initialise an `AudioSample` object is to create a `float` array specifying the amplitude of the desired waveform at successive sample positions, then pass the `float` array into the `AudioSample` constructor.

An example of initialising and using an `AudioSample` object is shown in Figure 5.13. In this example, we generate a *reverse saw wave*, i.e. a saw wave with amplitude that starts positive and ends negative, like a mirror image of the one illustrated at the bottom of Figure 5.7. The sketch also creates a visualisation of a section of the generated waveform, as shown in Figure 5.14.

In this sketch, the `float` array named `reverseSaw` is initialised in the `setup()` method with 400 amplitude values sampled from the desired waveform (the number 400 being specified by the variable named `resolution`). The first sample has a value of 1.0 and the final one has a value of –1.0. An `AudioSample` object is then created (also in the `setup()` method), passing in the `reverseSaw` array as an argument.

Notice that we pass three arguments into the `AudioSample` constructor in `setup()`. The first is the `this` pointer, in common with most of the classes in the *Sound* library. The second argument is our `float` array as just described. The final argument specifies the waveform's *frame rate*, which is the default playback rate of the waveform expressed in frames per second.[5] One *frame* of an `AudioSample` is a single value from the float array. The array contains 400 frames, as specified by the variable named `resolution`, and this represents a single waveform. We would like to play this waveform many times per second to bring it into the audible range. In the sketch we use the variable named `defaultFreq` to specify that the waveform should be played 100 times per second (i.e. 100Hz). The number of *frames* per second should therefore be `defaultFreq*resolution`, so this is the value we supply as the third argument of the `AudioSample` constructor.

The `keyPressed()` method at the end of Figure 5.13 allows the user to change the playback rate of the sample by using the Z and X keys to decrease or increase the

[4] In object-oriented terms, `SoundFile` is a subclass of `AudioSample`.

[5] Be careful not to confuse the `AudioSample` frame rate, which specifies the speed of the audio playback, with a *Processing* sketch's frame rate, which specifies the number of times the `draw()` method is called per second.

```
/**
 * Example of audio synthesis by generating a data array
 * and using it to initialise an AudioSample object.
 *
 * In this example, we generate a reverse saw wave, i.e. one
 * that starts high and ends low. We sample 1024 points from
 * the wave and store them in the reverseSaw array. This
 * array is passed into the AudioSample constructor, where we
 * also specify that the wave should be repeated 100 times
 * per second during normal playback. The rate of playback
 * can be adjusted interactively using the Z and X keys.
 *
 * The waveform is visualised on screen by reading data from
 * the AudioSample object. The screen shows 0.1 seconds of data.
 */

import processing.sound.*;

AudioSample sample;
int  resolution = 400;  // number of data points to represent wave
int  defaultFreq = 100; // default number of waves to play per sec
float rate = 1.0;        // adjustment factor for playback speed
float displaySecs = 0.1; // visual display shows 0.1s of audio

void setup() {
  size(512, 512);
  stroke(255);
  strokeWeight(4);

  // Create an array and manually write a single reverse saw
  // wave into it
  float[] reverseSaw = new float[resolution];
  for (int i = 0; i < resolution; i++) {
    reverseSaw[i] = 1.0 - ((2.0*i) / resolution);
  }

  // Create the audio sample based on the data, set frame rate
  // to play defaultFreq (100) wavelengths per second
  sample = new AudioSample(this, reverseSaw, defaultFreq*resolution);

  // Play the sample in a loop (but don't make it too loud)
  sample.amp(0.2);
  sample.loop();
}

void draw() {
  background(0);

  int numFrames = sample.frames();
  int x1=0, y1=0;

  // calculate how many pixels to allocate to a single wave
  float pixelsPerWave = width/(defaultFreq*rate*displaySecs);

  // draw the waveform
  for (int x = 0; x < width; x++) {
    // calculate where this x point lies on the wave (where
    // 0.0 is the leftmost point and 1.0 is the rightmost point)
    float posOnWave = (x/pixelsPerWave)%1.0;
    // calculate the frame index of the AudioSample object that
    // corresponds to the calculated position on the wave
    int frameIdx = (int)(posOnWave*numFrames);
    // read the value of the identified data frame from the sample
    float frameValue = sample.read(frameIdx);
    // finally, calculate the required y coordinate, mapping a data value
    // of 1.0 to 0.25*height and a data value of -1.0 to 0.75*height
    int y = (int)((0.5-0.25*frameValue)*height);

    // draw a line from the previous calculated point to this one
    // (skip if x==0 because we don't have a previous point yet)
    if (x > 0) { line(x1,y1,x,y); }

    // store the current x and y coordinates to be used as the starting
    // point for the line to be drawn in the next iteration of the loop
    x1 = x; y1 = y;
  }
}

void keyPressed() {
  // change the sample playback rate if the Z or X keys are pressed
  if (key == 'x' || key == 'X') {
    rate = constrain(rate+0.1, 0.1, 3.0);
  }
  else if (key == 'z' || key == 'Z') {
    rate = constrain(rate-0.1, 0.1, 3.0);
  }
  sample.rate(rate);
}
```

Figure 5.13: Synthesis of a reverse saw wave using the `AudioSample` class.

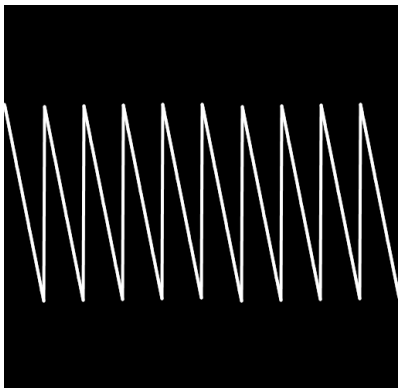


Figure 5.14: Output of the `AudioSample` Synthesis sketch shown in Figure 5.13, demonstrating a synthesized reverse saw wave.

rate, respectively. After checking for key presses, the rate is actually changed in the final line of the method by calling the sample's `rate()` method. We constrain the rate parameter to lie in the range 0.1 to 3.0. A rate of 0.1 means the sample is played ten times slower than its default speed, and a rate of 3.0 means three times faster.

In the sketch's `draw()` method we read data values from the `AudioSample` object and use them to draw a visualisation of the waveform on the screen. We do this by calling the sample's `read()` method in the line:

```
float frameValue = sample.read(frameIdx);
```

This line is called within a `for` loop that calculates the appropriate `y`-value to plot for each position across the `x`-axis. The data read from the sample gives us the shape of the waveform, but we need to decide how this data should be scaled along the `x`-axis (the time axis) when we plot it. This is achieved in the sketch by using the variable `displaySecs` to specify how many seconds should be represented across the entire width of the `x`-axis. We initialise `displaySecs` with a value of 0.1, meaning that we wish to display 0.1 seconds of waveform across the screen. From there, at the top of the `draw()` method we calculate how many pixels a single wavelength should occupy along the `x`-axis and assign it to the variable `pixelsPerWave`. This calculation takes into account the value of `displaySecs` as well as the current playback rate (given by the default frequency `defaultFreq` and the current rate modifier `rate`). Further calculations are then performed within the `for` loop to determine for each individual position along the `x`-axis which data value should be plotted and how it should be scaled along the `y`-axis. See the comments in the sketch for further details.

5.7 Amplitude analysis of audio samples

The *Sound* library provides the `Amplitude` class, which allows you to query the amplitude of a specified audio source. A very basic example of using the class to query a `SoundFile` object is shown in Figure 5.15.

The `Amplitude` class provides only two methods. The first of these is `input()`, which allows you to specify the audio source to be analysed. This method accepts various different kinds of source, including a `SoundFile` object and all of the oscillators and noise generators listed in Table 5.2.

```
/**
 * Simple example of using the Amplitude class to
 * query the current amplitude of a SoundFile
 * object.
 */

import processing.sound.*;

Amplitude amp;
SoundFile mySample;

void setup() {
  size(600, 300);
  fill(150);
  noStroke();

  // initialise the Amplitude and Sound File objects
  amp = new Amplitude(this);
  mySample = new SoundFile(this, "piano.wav");

  // set the SoundFile object as the input for amp
  amp.input(mySample);

  // play the SoundFile in a continuous loop
  mySample.loop();
}

void draw() {
  background(0);

  // query the current root mean square amplitude of the source
  float rms = amp.analyze();

  // draw an ellipse with size proportional to the RMS amplitude
  ellipse(width/2, height/2, rms*200, rms*200);
}
```

Figure 5.15: Basic usage of the `Amplitude` class to query the output of a `SoundFile` audio sample.

The only other method provided by `Amplitude` is `analyze()`. This returns the current root mean square amplitude of the selected input source, measured over a short fixed time window.

5.8 Music synthesis

We can put together the principles of sound learned in this Chapter to make a piece of music. Note, however, it is more common to use an audio editor and sequencer to perform music synthesis due to the complexity of the tasks involved.

In this section we construct a simple percussive rhythm that emulates an early drum machine such as the Roland TR808; these used analogue circuitry to generate simple percussive sounds and organise them into beats and patterns.

5.8.1 Rhythm

When we think of sound, many of us probably think of a rhythm, or groove, or perhaps a melody. Rhythm is the component of sound that organises it in time as a series of events with different durations.

We can compose a rhythmic sequence by triggering events at certain time points. A periodic rhythm is one that repeats, exactly, over and over, until it is stopped. Periodic rhythms are very common and are the basis for most of the world's music.

```
/**
 * A simple drum machine. This example demonstrates how
 * to generate simple percussive rhythm combining several
 * different instruments.
 *
 * The example also uses the Amplitude class to measure
 * and visualise the output of each of the instruments.
 */

import processing.sound.*;

// Define the underlying rate of the sequencer (used
// to determine the sketch's frame rate and hence the
// number of times the draw() method is called per second)
int triggerFreq = 24;

SinOsc bassDrum;
SinOsc hiHat;
WhiteNoise snareDrum;

Amplitude bassAmp;
Amplitude hiHatAmp;
Amplitude snareAmp;

float rms;

void setup() {
  size(600,300);
  background(0);
  noStroke();

  frameRate(triggerFreq);

  // Initialise our three instruments with appropriate
  // frequencies and amplitudes
  bassDrum = new SinOsc(this);
  bassDrum.set(60, 1.0, 0.0, 0.0);

  hiHat = new SinOsc(this);
  hiHat.set(5000, 0.1, 0.0, 0.0);

  snareDrum = new WhiteNoise(this);
  snareDrum.set(0.75, 0.0, 0.0);

  // Connect each instrument to its associated Amplitude analyser
  bassAmp = new Amplitude(this);
  bassAmp.input(bassDrum);

  hiHatAmp = new Amplitude(this);
  hiHatAmp.input(hiHat);

  snareAmp = new Amplitude(this);
  snareAmp.input(snareDrum);
}

void draw() {
  // Determine which instrument(s), if any, to play now
  boolean playBass  = ( frameCount   %8 == 0);
  boolean playHiHat = ((frameCount+4)%8 == 0);
  boolean playSnare = ((frameCount+8)%16 == 0);

  // Play or stop each instrument as required
  if (playBass)  bassDrum.play();  else bassDrum.stop();
  if (playHiHat) hiHat.play();     else hiHat.stop();
  if (playSnare) snareDrum.play(); else snareDrum.stop();

  background(0);

  // Query the current root mean square amplitude of each
  // instrument, and use the measured value to determine
  // the size of a circle to draw on the screen
  rms = hiHatAmp.analyze();
  fill(150,50,50); // draw the hi-hat in red
  ellipse(width/4, height/2, rms*3000, rms*3000);

  rms = bassAmp.analyze();
  fill(50,150,50); // draw the bass drum in green
  ellipse(width/2, height/2, rms*300, rms*300);

  rms = snareAmp.analyze();
  fill(50,50,150); // draw the snare drum in blue
  ellipse(3*width/4, height/2, rms*1500, rms*1500);
}
```

Figure 5.16: Using the *Sound* library to synthesize sine waves and noise bursts and sequence them into a glitch electronica rhythm.

Figure 5.16 shows how we can compose a simple rhythm using just simple tones (sine waves) and noise bursts. The early electronic drum machines worked by triggering simple tones and noise bursts in regular patterns using a machine called a sequencer.

To make a bass drum sound, we create a `SinOsc` object in the sketch's `setup()` method and set its frequency to 60Hz to generate a suitable low frequency sine wave. We make a hi-hat sound using a higher frequency sine wave. Finally, we make a snare drum sound by creating a `WhiteNoise` object to generate a noise burst. In the calls to the `set()` method of these three objects we also specify suitable amplitudes for each one (1.0, 0.1 and 0.75, respectively).

The timing mechanisms for these sounds are designed so that a standard techno-style rhythm is played. In the `setup()` method we set the sketch's frame rate to 24 frames per second; this determines the number of times per second the `draw()` method will be called, and hence gives us an underlying rate of operation of our sequencer. In the `draw()` method we query the `frameCount` variable, which is an inbuilt variable in *Processing* that is incremented each time the `draw()` method is called. In this example the sequencer uses some modulo arithmetic. We treat each sample as a beat and for each beat index (frameCount) we test to see if it is exactly divisible by some quantity that we call the beat period (such as 8 for the bass drum). If it is exactly divisible by this beat period, i.e. `frameCount%beatPeriod==0`, then we call the instrument's `play()` method.

The result is a repeating rhythmic pattern that you can edit by changing the parameters and modulo arithmetic of the example. In the `draw()` method we also visualise the current amplitude of each of the instruments by calling their associated `Amplitude` object's `analyze()` method, just as we did in Figure 5.15.

Learning activity

Run the sketch in Figure 5.16 on your computer and listen to the output. Try altering the rhythm by changing the modulo arithmetic at the top of the `draw()` method.

Experiment with adding different types of oscillator and noise to the sketch.

5.9 Other facilities provided by the Sound library

The *Sound* library includes several other classes that we will not cover in this course. We mention them here for completeness, in case you wish to explore them yourself.

In addition to the `Amplitude` analyser, there is also an `FFT` class that performs a Fast Fourier Transform on an audio source and returns the calculated frequency spectrum. The `FFT` class can be used with the same variety of audio sources as accepted by the `Amplitude` class.

The `Env` class allows you to apply an *attack-sustain-release* envelope to an audio source.

Finally, a number of classes allow you to apply effects to audio sources. There are three classes that filter out specific ranges of frequencies from a source: `LowPass`,

`HighPass` and `BandPass`. `Delay` and `Reverb` classes are also available.

Further information about these and all other classes in the *Sound* library can be found in the online reference manual available at `https://processing.org/reference/libraries/sound/`.

5.10 Summary

In this chapter we discussed the components of sound and its digital audio representation. Various audio file formats are used to store sounds in either uncompressed (PCM) or compressed form. Using the *Sound* library, we can play back sounds from a file using the `SoundFile` class, or we can generate sound synthetically using the `AudioSample` class. The library also provides various generators for creating simple oscillators (e.g. the `SinOsc` class) and noise (e.g. the `WhiteNoise` class). We can read values from the underlying data of an `AudioSample` object by calling its `read()` method, and we can analyse the amplitude of an audio source using the `Amplitude` class.

The basis of music synthesis and sound manipulation is an understanding of the sound's spectrum which comprises the frequency components of the sound. When we synthesize a sound we create these frequency components from computational functions driven by some parameters: usually amplitude and frequency. We can add sounds together in a process called mixing to make musical sequences consisting of rhythmic and pitched elements. This is the basis for music synthesis, which is an advanced topic that we shall cover in Year 2.

You should now be able to:

- describe the digital audio concepts of sampling and mixing, and how all sound waves are composed of sinusoidal waveforms
- play sound from PCM sound files using the *Sound* library's `SoundFile` class
- synthesize digital audio using the *Sound* library's inbuilt oscillator and noise classes
- synthesize arbitrary digital audio waveforms using the *Sound* library's `AudioSample` class
- read the underlying data of an `AudioSample` object by using its `read()` method
- use the `Amplitude` class to analyze the root mean square amplitude of an audio source
- compose simple sonic fragments, using the *Sound* library and incorporating principles of rhythm.

5.11 Exercises

1. Plot two sine waves, one at 440 Hz and the other at 880 Hz, both with amplitude 0.5.
2. Sum the two sinusoids from Exercise 1 and plot the result.
3. What is compression? Lossy compression can be acceptable for some media applications, while lossless compression is more appropriate for data and text. Why is this the case? Explain the difference between these two categories of compression, and give some examples of each.

4. Find and describe two different compression formats that are suitable for audio data, and give a critical comparison of them.
5. Write a sketch to continuously play the waveform of Exercise 2. How many pitches do you hear?
6. Modify the sketch in Figure 5.16 to play different rhythms. Make a pattern with three beats. Make one with five beats.

Chapter 6

Generative Systems

Essential reading

Reas, C. and B. Fry *Processing: A Programming Handbook for Visual Designers and Artists* (MIT Press, Sep. 2007) [ISBN 0262182629].
Reas, C. and B. Fry `http://www.processing.org/reference`, online *Processing* reference manual.

Additional reading

Bentley, P. J. and D. W. Corne (eds) *Creative Evolutionary Systems* (Morgan Kaufmann Publishers Inc., 2001) [ISBN 1558606734].
Prusinkiewicz, P. and Lindenmayer, A. *The Algorithmic Beauty of Plants* (Springer-Verlag, softcover reprint, 1996; free electronic version available at `http://algorithmicbotany.org/papers/#abop`) [ISBN 0387946764].
Shiffman, D. *The Nature of Code: Simulating Natural Systems with Processing* (The Nature of Code, 2012; free online version available at `http://natureofcode.com/book/`) [ISBN 0985930802].

6.1 Introduction

An important development in creative computing over the last three decades has been an increasing understanding of how particular types of complex system can be generated as the emergent result of very simple underlying rules.

Many researchers have explored subjects such as fractals, iterated function systems and rule-based systems for generating rich and complex graphics. Sometimes these graphics can imitate lifeforms and other times they show us a hidden universe within number systems.

In this chapter we will explore some of these systems from the perspective of creative computing, looking at how we can use generative systems to create visual images for simulating natural-looking complex forms.

6.2 Fractals

In the 1970s and 1980s, mathematicians discovered that complex patterns can be generated by simple numerical systems. One class of such systems are called *fractals*. Fractals exhibit the property of self-similarity, meaning that the shape of the fractal is the same at all scales. A fractal seen from a distance possesses the same shape characteristics as the fractal seen in a close up or zoomed-in view.

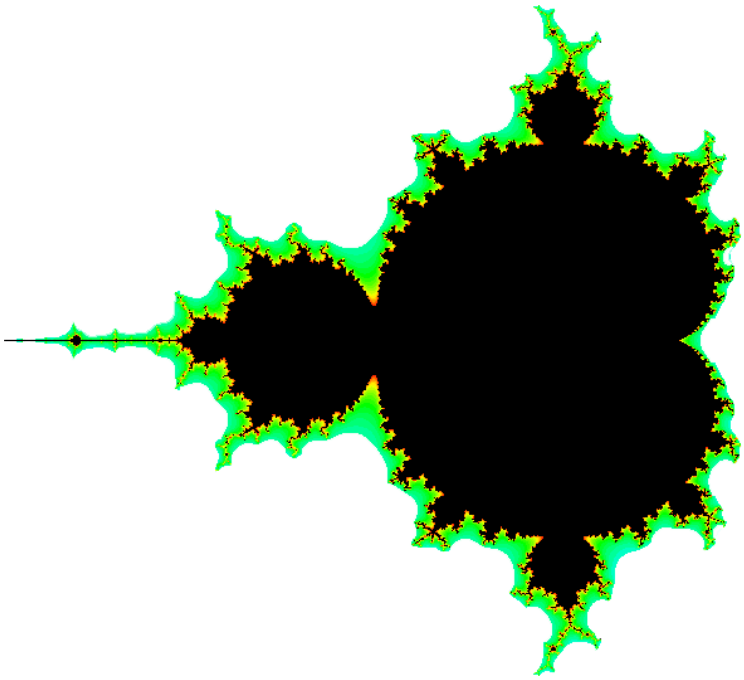


Figure 6.1: The well-known *Mandelbrot set* fractal discovered by mathematician Benoit Mandelbrot. Each of the side lobe arms, hanging off the central shape, contains an infinite number of copies of the central shape.

A common example of a fractal is the coastline. Seen from a map view, or satellite image as seen in *Google Maps*, the coastline of a country consists of a continuous rough curve. The coastline seen from such an image appears rather similar to the coastline seen from a different closer view—a birds-eye view perhaps. If we keep getting closer to the coastline, we see that the rocks and cliffs that define the coastline also exhibit the same roughness of shape.

This property of similar shape at different scales is called self-similarity and is the defining property of a fractal. Figure 6.1 shows the Mandelbrot set fractal, within which all of the side lobes are composed of an infinite number of copies of the central black shape. Fractals have many applications in computer graphics and in the sciences and computer simulation.

First studied in depth by mathematicians such as Benoit Mandelbrot, fractals have become an important component of creative computing.

6.2.1 Iterated function systems and substitution systems

One class of fractals are *iterated function systems* (IFS). These systems work by taking an input, transforming it in some way, and feeding the output back into the input to re-transform it in a recursive pattern.

Substitution systems are a particular type of iterated function system. They work by transforming their input by substitution according to a set of rules. Substitution systems operate on strings, transforming an initial predecessor string (or word) into more and more complex words according to a set of rules.

Iterated function systems and substitution system have found particular application in modelling the appearance of natural phenomena such as clouds, coastlines, mountains, plants and the textures of flora and fauna.

6.2.2 L-systems

L-systems (named after their inventor, Aristid Lindenmayer) are a type of substitution system. An L-system starts with an initial string called an *initiator*. For example, we may start with the string *ab*. For each letter in the alphabet, we define a *generator* that is a substitution rule. For example $a \rightarrow ba$ reads "substitute each occurrence of a in the predecessor string with the letters ba". Likewise, the generator $b \rightarrow ccb$ reads "substitute each occurrence of b in the predecessor string with the letters ccb".

As an example, consider an L-system with an alphabet $\{a, b, c\}$, an initiator a and the following three generators:

$$a \rightarrow ba$$
$$b \rightarrow ccb$$
$$c \rightarrow a$$

The first iteration of the substitution system replaces a using the generator $a \rightarrow ba$ so the output is ba. Following this method, the sequence of substitutions for the first 4 iterations is:

$$a \rightarrow ba \rightarrow ccbba \rightarrow aaccbccbba \rightarrow babaaaccbaaccbccbba$$

We can interpret the string of letters generated by a substitution system as a set of commands for drawing using coordinate system transformations. We do this by mapping each letter in the L-system alphabet to a drawing command. A convenient way of doing this is to use *turtle graphics*[1] so that the state of the turtle is changed to a new position and orientation after each drawing command. Table 6.1 shows an example of a list of commands could control the turtle using *Processing*'s drawing and coordinate system transformation commands.

Character	Command	Transformation
F	Draw a line and move forward	`line(0,0,d,0);translate(d,0);`
f	Move forward without drawing a line	`translate(d,0);`
+	Rotate by `PI/2` radians (90 degrees)	`rotate(PI/2);`
-	Rotate by `-PI/2` radians (-90 degrees)	`rotate(-PI/2);`

Table 6.1: Mapping of characters in an L-system alphabet to turtle commands and *Processing* drawing and transformation commands.

As an example, to draw a square of length `d` pixels using this system, we specify the turtle's initial position and orientation, the line length `d`, and a suitable string, which, for a square, might be `F-F-F-F`. The string reads:

> *Draw a line (of length `d`) and move forward by `d` pixels. Rotate by `-PI/2`, and repeat three times.*

The end result of this procedure is a square of length d.

[1] Recall that we discussed turtle graphics in Chapter 6 of Volume 1 of the Subject guide.

6.2.3 The Koch snowflake

The sequence of images shown in Figure 6.2 illustrates the process of substitution of an initiator with a generator to make successively more complex self-similar curves. This example is called the Koch snowflake and was first proposed in 1905 by the mathematician Helge von Koch. Each iteration replaces every line with a scaled version of the generator. The result is a complex curve exhibiting a high degree of roughness and self-similarity, properties that are often associated with fractals.

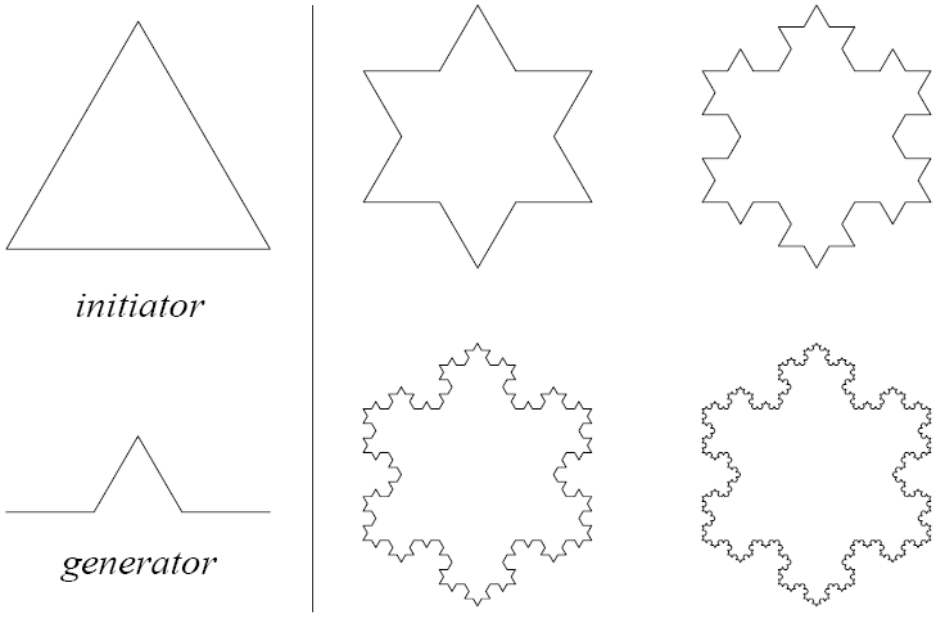


Figure 6.2: Initiator, generator and four iterations of the substitution system to generate the Koch snowflake. This self-similar curve is an example of a fractal.

Figure 6.3 is an example of a substitution system used for drawing another complex fractal curve called the *quadratic Koch island*. Using the alphabet notation introduced above, the initiator is a square defined by the string `F-F-F-F` and the generator is $F \rightarrow F + F - F - FF + F + F - F$. The images correspond to iterations 0 to 3 of the substitution system. The turtle angle is equal to 90 and the turtle line length `d` is decreased by a factor of 4 between iterations. The implementation is shown in Figure 6.4.

In this sketch, the substitution is applied to strings of letters that are stored in the `String` variable named `state`. Substitution is performed using a method called `substitute` which accepts two strings as arguments, `s` and `r`, and produces a third string as output that is the result of substituting all the occurrences of the letter `F` with the entire substitution string:

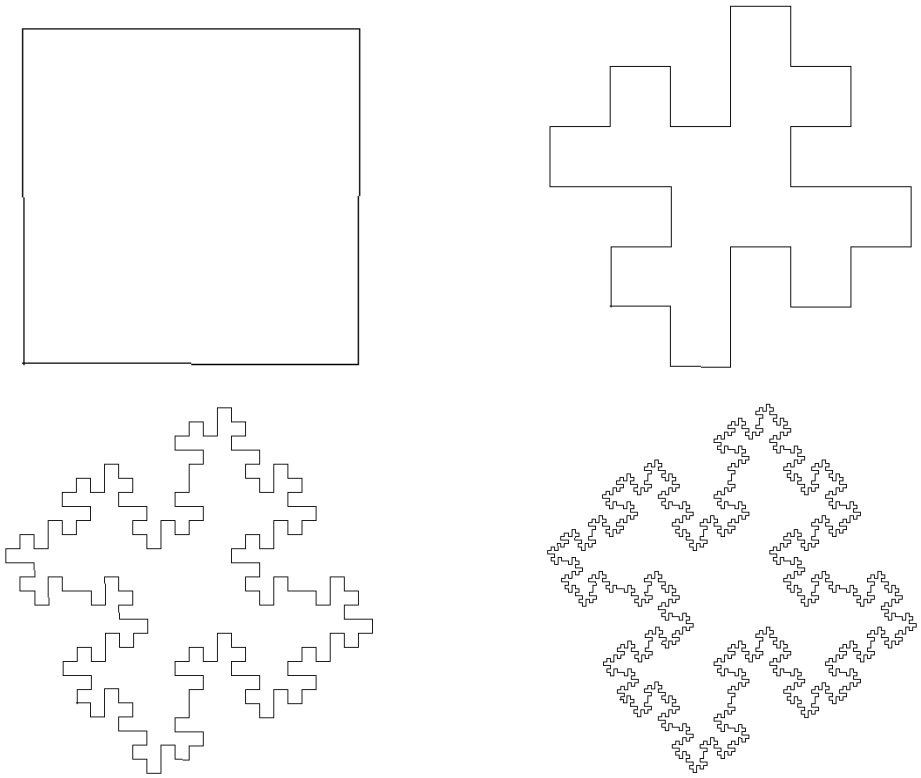


Figure 6.3: Initial state and three iterations of L-system substitution for the quadratic Koch island.

```
String substitute(String s, String r) {
  String s2 = new String();
  for (int j=0; j<s.length(); j++)
    if (s.charAt(j)==’F’)
      s2=s2+r;
    else
      s2=s2+s.charAt(j);
  return s2;
}
```

6.2.4 Two-handed substitutions

It is sometimes useful to divide the line-drawing command `F` into two different types that we shall call left- and right-handed lines, denoted by the letters `F` and `H` respectively. Each of these commands draws a line in exactly the same way, but they differ in their substitution rules. The following version of the `substitute()` method illustrates how we can implement substitution rules for two letters in our alphabet:

```
// L-System to generate the quadratic Koch island

int SZ=512, sz=SZ/2;           // screen size and half screen size
float d = sz;                  // turtle line length
float x = sz-d/2, y = sz+d/2;  // initial x and y position
float ang = HALF_PI;           // initial rotation
String state="F-F-F-F";        // initial state
String ruleF="F-F+F+FF-F-F+F"; // substitution rule
int L = 3;                     // number of times to substitute

void setup() {
  size(SZ,SZ);
  background(255);
  stroke(0);
  noLoop();
  // Perform L rounds of substitutions on the initial state
  for (int k=0; k < L; k++) {
    state = substitute(state, ruleF);
    d/=4; // divide turtle line length
  }
  // The variable "state" now contains the final state of the L-system,
  // which will be drawn to the display window in the draw() method.
}

void draw() {
  translate(x,y); // initial position
  rotate(-ang);   // initial rotations
  // now walk along the state string, executing the corresponding
  // turtle command for each character
  for (int i=0; i < state.length(); i++)
    turtle(state.charAt(i));
}

// Turtle command definitions for each character in our alphabet
void turtle(char c) {
  switch(c) {
  case 'F':
    line(0, 0, d, 0);
    translate(d,0);
    break;
  case 'f':
    translate(d,0);
    break;
  case '-':
    rotate(ang);
    break;
  case '+':
    rotate(-ang);
    break;
  }
}

// Apply substitution rule r to string s and return the resulting string
String substitute(String s, String r) {
  String s2 = new String();
  for (int j=0; j<s.length(); j++)
    if (s.charAt(j)=='F')
      s2=s2+r;
    else
      s2=s2+s.charAt(j);
  return s2;
}
```

Figure 6.4: Source code for the quadratic Koch island shown in Figure 6.3.

```
String substitute(String s, String F, String H) {
  String s2 = new String();
  for (int j=0; j<s.length(); j++) {
    if (s.charAt(j)=='F')
      s2=s2+F;
    else if (s.charAt(j)=='H')
      s2=s2+H;
    else
      s2=s2+s.charAt(j);
  }
  return s2;
}
```

Figures 6.5 and 6.6 illustrate the application of two-handed substitution systems to drawing more complex fractal curves. The entire source code for these systems is shown in Figure 6.7. This code differs from the previous version, shown in Figure 6.4, in adding the `H` command to the `turtle` method and a substitution method for the three letters `F`, `H` and `f`.

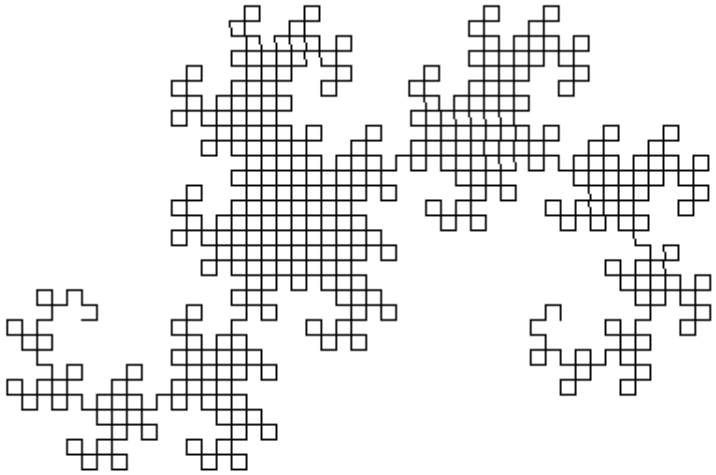


Figure 6.5: L-system with initiator `F`, generators $F \rightarrow F + H+$ and $H \rightarrow -F - H$, angle `PI/2` and 10 substitution iterations. This fractal is known as the Dragon Curve.

6.2.5 Plant modelling with bracketed L-systems

A classic use of L-systems is the generation of plant-like structures, such as trees, that can be produced with many slightly different variations. To produce such structures, we require a slight modification to the previous L-system's rules to model the branching structure of plants.

We model branching by allowing the L-system to save and restore the current turtle state. This is the same as saving and restoring the current coordinate system transformation which is achieved with *Processing*'s `pushMatrix()` and `popMatrix()` commands.

We denote these operations in the L-system production strings using brackets [and] to represent `pushMatrix()` and `popMatrix()` respectively. By adding these two new

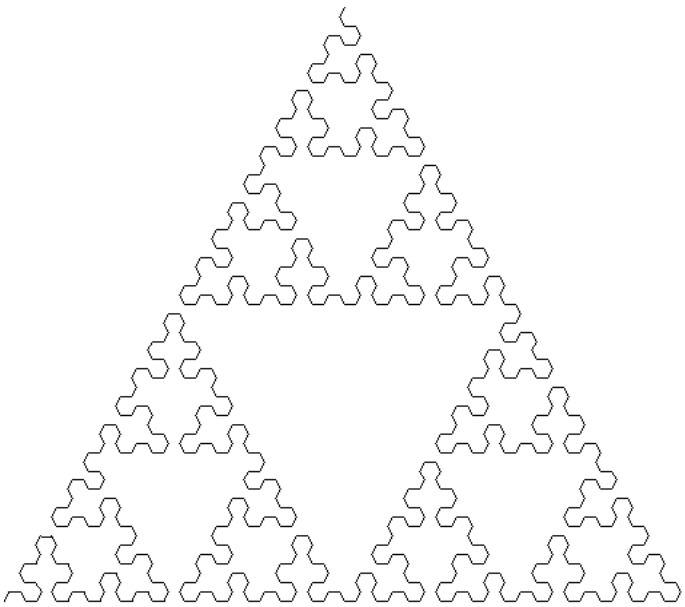


Figure 6.6: L-system with initiator `H`, generators $F \rightarrow H + F + H$ and $H \rightarrow F - H - F$, angle `PI/3` and 6 substitution iterations.

strings, we are able to describe the drawing of a branching structure using a string of letters.

For example, the sequence of operations `F[+F][-F]F` used with angle `PI/8` produces the tree shown in Figure 6.8 (page 138). The commands describe an initial line, followed by saving the coordinate system. Then a positive rotation followed by a line, then a restoration of the coordinate system to just after the initial line, followed by a negative rotation and a line, coordinate system restoration and a final line in the same direction as the initial stroke but placed on top of it.

Figure 6.9 shows the same production process using the generator string `F[+F[-F][+F]][-F[+F][-F]]F`. This string describes a branching process with two levels of branching. In this way we can build up complex branching structures that have many levels of branching and are not necessarily symmetrical.

Figures 6.10, 6.11 and 6.12 show three branching plant-like structures that were generated using bracketed L-systems as described in this section.

The source code for bracketed L-system generation is shown in Figure 6.13 (page 141). There are two extra production rules added to deal with the brackets, one for each of the brackets `[` and `]` generating a `pushMatrix()` and a `popMatrix()` respectively. The sketch also incorporates a new character `s` to allow strings to modify their own scaling factor, and an `f-rule` which moves the turtle forward by a distance `d` but without drawing a line (these features are not used in the initiator and generators shown in Figure 6.13, but you might like to experiment with them).

```
// Two-handed L-System

int SZ=512, sz=SZ/2;              // screen size and half screen size
float d = sz/32;                  // turtle line length
float x = sz+sz/2, y = x;         // initial x and y position
float ang = HALF_PI, a = -HALF_PI; // turtle rotation and initial rotation
String state = "F";               // initial state
String F_rule = "F+H+";           // F-rule substitution
String H_rule = "-F-H";           // H-rule substitution
int L = 10;                       // number of times to substitute

void setup() {
  size(SZ,SZ);
  background(255);
  stroke(0);
  noLoop();
  // Perform L rounds of substitutions on the initial state
  for (int k=0; k < L; k++)
    state = substitute(state, F_rule, H_rule);
}

void draw() {
  translate(x,y); // initial position
  rotate(a);      // initial rotation
  // now walk along the state string, executing the corresponding
  // turtle command for each character
  for (int i=0; i < state.length(); i++)
    turtle(state.charAt(i));
}

// Turtle command definitions for each character in our alphabet
void turtle(char c) {
  switch(c) {
  case 'F': // drop through to next case
  case 'H':
    line(0, 0, d, 0);
    translate(d,0);
    break;
  case 'f':
    translate(d,0);
    break;
  case '-':
    rotate(ang);
    break;
  case '+':
    rotate(-ang);
    break;
  }
}

// Apply substitution rules F and H to s and return the resulting string
String substitute(String s, String F, String H) {
  String s2 = new String();
  for (int j=0; j<s.length(); j++)
    if (s.charAt(j)=='F')
      s2=s2+F;
    else if (s.charAt(j)=='H')
      s2=s2+H;
    else
      s2=s2+s.charAt(j);
  return s2;
}
```

Figure 6.7: Source code for a two-handed L-system with two types of line (left `F` and right `H`) and two substitution rules.

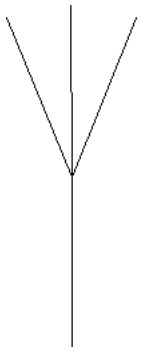


Figure 6.8: A bracketed L-system is capable of generating branching structures such as this simple tree. The string that generated this tree is `F[+F][-F]F` using an angle of `PI/8`. There are no substitutions applied in this simple example.

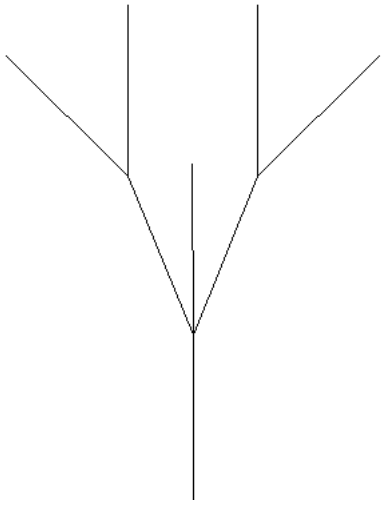


Figure 6.9: A branching bracketed L-system showing two-levels of branching. The production string that generated this figure is `F[+F[-F][+F]][-F[+F][-F]]F`.

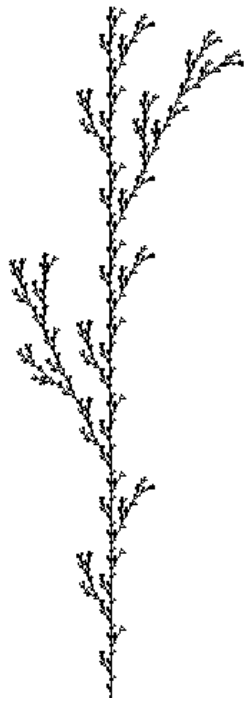


Figure 6.10: A plant-like structure (variation 1) generated by a bracketed L-system.

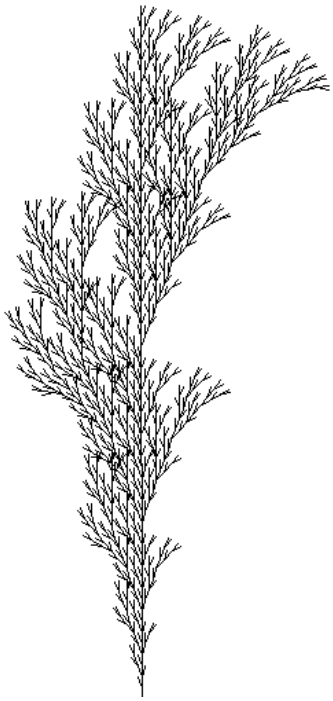


Figure 6.11: A plant-like structure (variation 2) generated by a bracketed L-system.

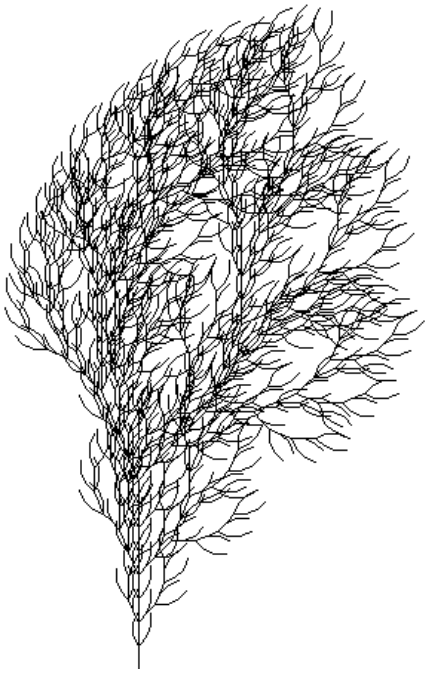


Figure 6.12: A plant-like structure (variation 3) generated by a bracketed L-system.

6.3 Genetic algorithms

At the same time that fractals were being discovered and developed, an equally important and influential body of biologically-inspired research started in computing. Genetic algorithms were pioneered by John Holland in the 1960s, based on earlier work by biologists wishing to model evolutionary processes.[2]

In his book *The Blind Watchmaker*, Richard Dawkins used genetic algorithms as a way to explore a parameter space of organism morphologies (body shapes) by simulating the inheritance and random mutation properties of reproducing organisms.

6.3.1 The Blind Watchmaker algorithm

The basic idea behind Dawkins' use of genetic algorithms is that complex designs, with properties which we desire, can be automatically generated by incrementally building them up from an initial state, much like we saw in the L-system examples above. At each iteration, the current state is modified using small random changes on the parameters of the generating function.

The crucial link to genetics and biological systems is the use of a *population* of slightly different versions of the design, generated using slightly different parameter

[2]Various other researchers were developing similar work at around the same time. See `http://en.wikipedia.org/wiki/Evolutionary_computation` for an introduction.

```
// Bracketed L-System

int SZ=512, sz=SZ/2;           // screen size, half screen size
float d = sz/128;              // turtle line length
float x = SZ/2, y = SZ;        // initial x and y position
float ang = (25.7/180.0)*PI;   // turtle rotation
float a = -HALF_PI;            // initial rotation
float s = 1;                   // branch scale factor
String state = "F";            // initial state
String F_rule = "F[+F]F[-F]F"; // F-rule substitution
String H_rule = "";            // H-rule substitution
String f_rule = "";            // f-rule substitution
int L = 5;                     // number of times to substitute

void setup() {
  size(SZ,SZ);
  background(255);
  stroke(0);
  noLoop();
  // Perform L rounds of substitutions on the initial state
  for (int k=0; k < L; k++) {
    state = substitute(state, F_rule, H_rule, f_rule);
  }
  println(state);
}

void draw() {
  translate(x,y); // initial position
  rotate(a);      // initial rotation
  // now walk along the state string, executing the
  // corresponding turtle command for each character
  for (int i=0; i < state.length(); i++)
    turtle(state.charAt(i));
}

// Turtle command definitions for each
// character in our alphabet
void turtle(char c) {
  switch(c) {
  case 'F': // drop through to next case
  case 'H':
    line(0, 0, d, 0);
    translate(d,0);
    break;
  case 'f':
    translate(d,0);
    break;
  case 's':
    scale(s);
    break;
  case '-':
    rotate(ang);
    break;
  case '+':
    rotate(-ang);
    break;
  case '[':
    pushMatrix();
    break;
  case ']':
    popMatrix();
    break;
  default:
    println("Bad character: " + c);
    exit();
  }
}

// Apply substitution rules F, H and f to
// string s and return the resulting string
String substitute(String s, String F,
                  String H, String f) {
  String s2 = new String();
  for (int j=0; j<s.length(); j++)
    if (s.charAt(j)=='F')
      s2=s2+F;
    else if (s.charAt(j)=='H')
      s2=s2+H;
    else if (s.charAt(j)=='f')
      s2=s2+f;
    else
      s2=s2+s.charAt(j);
  return s2;
}
```

Figure 6.13: Source code for a bracketed L-system.

values. The user *selects* from the population the design that best fits what they are looking for. This design is then used as the starting point for the next generation of the population. The new population is filled up with many slightly different variations of the selected design(s) from the previous generation. This process of selecting the best design(s) from the current population and using them to generate a new population is repeated many times.

In the language of genetic algorithms, a selected design from the current population is called a *parent*. Each of the random variations that are generated from it are called *children* (or *progeny*). The process of the Blind Watchmaker algorithm is to select from the among progeny, at each iteration, the design with the most desirable characteristics (from the perspective of the human viewer).

6.3.2 User-guided genetic algorithms versus fitness functions

As stated above, the Blind Watchmaker algorithm involves selection of the best design by a human viewer at each iteration. It is therefore an example of what is know as a *user-guided genetic algorithm*. Other forms of genetic algorithm exist where the selection step is performed by a pre-specified *fitness function* rather than by human selection. User-guided genetic algorithms are useful in cases where it is hard to formalise the desired characteristics of a design into an exact fitness function.

6.3.3 Biomorphs

Dawkins called the forms that were generated in his system *Biomorphs*, after the life-like images in the paintings of the artist Desmond Morris. Each Biomorph is a life-like structure generated algorithmically, just like in the L-systems we saw earlier in the chapter. In the remainder of this chapter we will discuss how the Biomorph system might be implemented in *Processing*.

A Biomorph has three important processes associated with it: *genes*, *development* and *reproduction*.

The *genes* are the parameters that control the appearance of the Biomorph. For example, in the L-system language, the genes might be the individual letters of the initiator and generator strings, as well as the line-length and angle parameters. These are the individual components that, when used with the drawing commands, control the appearance of the object. In our sketch we will intialise a Biomorph's genes in a method called `initGenes()`.

Development is the process of generating a full design based upon the parameter values specified in the genes. It is the mapping from genes to a set of commands for producing a drawing. In our Biomorph sketch we will implement development in a slightly different way to our previous L-system sketches—instead of preforming multiple iterations of string substitutions, we will make use of a recursive drawing method that we shall call `tree()`.

Finally, there is the process of *reproduction*. The system generates multiple copies of the selected predecessor (the parent) with random variations (called *mutations*) so that each of the progeny will differ slightly from the parent. We implement this process in a method called `reproduction()`.

The combination of all three of these processes—genes, development and reproduction—enables a sequence of generative drawing where, at each iteration, the user controls the desired properties of drawing by selecting the progeny that have the required traits.

Figure 6.14 shows a sequence of Biomorphs generated by selecting, at each iteration, the progeny with desired characteristics.

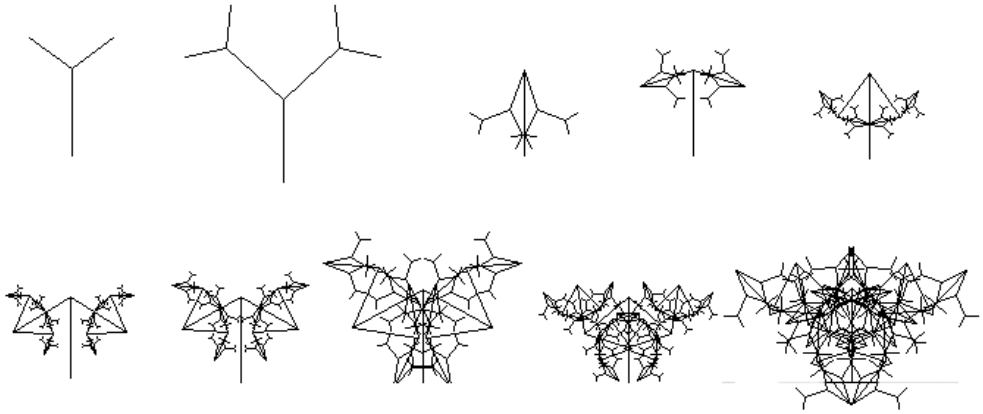


Figure 6.14: A sequence of Biomorphs generated by successive iterations of the Blind Watchmaker algorithm. The initial state is a simple tree with one level of branching. After nine iterations a complex beetle-like shape has emerged. The user has chosen, at each iteration, which Biomorph will be used to create the next generation from a population of Biomorphs.

Figure 6.15 shows the user interface for creating Biomorphs using the Blind Watchmaker algorithm. The parent (current Biomorph) is highlighted in the top left corner of the grid. The 15 progeny are random variations on the parent and they are displayed in the other grid positions. The user selects the child to be used for the next generation by clicking on its square in the grid.

Figure 6.16 shows the source code for our full implementation of the Blind Watchmaker algorithm as a *Processing* sketch. In the following sections we look in more detail at each section of the sketch and how they implement the various processes involved in the Biomorph system.

6.3.4 Modelling genetic processes

The process of reproduction with mutations is modelled by changing the values of the parent genes by a small amount using a random number generator.

The genes are implemented as four arrays:

```
int[]     G_Branching    = new int[NPROG];
float[][] G_Angle        = new float[NPROG][LMAX];
float[][] G_ShrinkFactor = new float[NPROG][LMAX];
float[]   G_Height       = new float[NPROG];
```

Here the constant `NPROG` denotes the number of Biomorphs (number of progeny plus

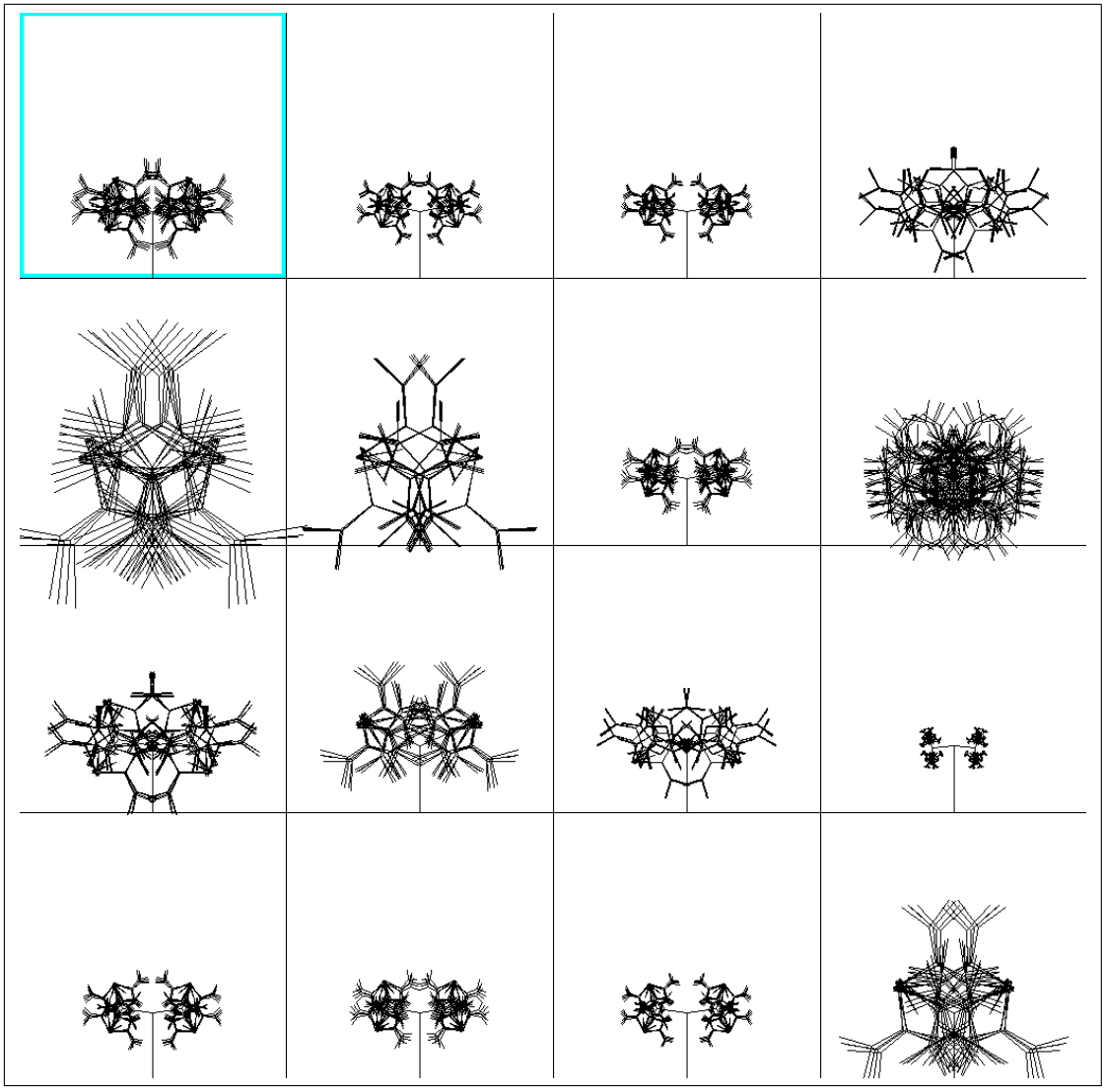


Figure 6.15: Biomorphs generated by the Blind Watchmaker algorithm.

the parent) and is set to 16 in the source code: the parent and 15 progeny. The parent is always located at index 0 in these four arrays, and the 15 progeny occupy the remaining positions in the arrays. The constant `LMAX` is the maximum number of levels of branching to use and is set to 20 in the source code.

The first array, `G_Branching[]` is the level of branching to use for each of the parent and progeny Biomorphs. As with all the arrays used for storing genes, the parent's genes are stored in the first element of the array: `G_Branching[0]`.

The second array, `G_Angle[][]` stores the angles for each Biomorph at each level of branching. So, for example, a Biomorph with three levels of branching can have a different angle for each of the levels up to a maximum of `LMAX` levels for each progeny. Therefore, this is a two-dimensional array.

The `G_ShrinkFactor[][]` array stores the amount by which to reduce the line length at each level of branching. So, for a Biomorph with three levels of branching, each level could have a different line length relative to the initial state. If the shrink factors are `(1.0, 1.5, 2.0)` then the length of the first branch is the same as the initial length `sz`, the length of the second level of branches is `sz/1.5` and the third is `(sz/1.5)/2.0`. Again, there are up to `LMAX` levels of branching allowed, so there are up to `LMAX` elements in the `G_ShrinkFactor[][]` array for each Biomorph, requiring a two-dimensional array.

The final array is `G_Height[]` which is the initial line length for the first level (zeroth branch) for each of the progeny. This gene controls the overall size of the Biomorph.

6.3.5 Modelling the development process

The drawing process is controlled by the `development()` method and is performed on a 4×4 grid in this example. This allows the drawing of a parent and 15 progeny. The `development()` method calls the `tree()` method to draw each individual Biomorph. `tree()` is a recursive method which accepts 7 arguments corresponding to the Biomorph position and the values of its genes. The first argument is the index of the Biomorph that is being drawn, which allows the `tree()` method to access the Biomorph's genes from within the method, since they are global arrays.

```
// Draw the parent and all of its children
void development() {
  for (int k=0; k<NPROG; k++) {
    int x = (int)((k%N + 0.5) * width/N);
    int y = (k/N + 1) * height/N;
    pushMatrix();
    translate(x,y);
    tree(k,0,0,0,-1,G_Height[k],G_Branching[k]);
    popMatrix();
  }
}
```

The tree method is a variant of the recursive tree drawing method from Volume 1, which is also the basis of the L-system production method discussed earlier in this chapter. In this example we explicitly perform rotations and translations using trigonometric methods. These could also be implemented using the coordinate system transformation methods, or turtle graphics methods, used in the L-systems above.

```
// tree is a recursive method to draw a single Biomorph according
// to the parameter values specified by its genes
void tree(int k, float x1, float y1, float dx, float dy, float sz, int n) {
  if (n==0) { // check for exit point of recursion
    return;
  }
  float x2 = x1 + sz*dx;
  float y2 = y1 + sz*dy;

  line(x1,y1,x2,y2);

  n--; // Change level of branching

  // Rotate the branches about an angle
  float dx2 = cos(G_Angle[k][n])*dx + sin(G_Angle[k][n])*dy;
  float dy2 = -sin(G_Angle[k][n])*dx + cos(G_Angle[k][n])*dy;

  // New branch at position and new direction vector
  // by recursively calling the tree method
  tree(k,x2,y2,dx2,dy2,sz/G_ShrinkFactor[k][n],n);

  // Symmetry: Rotate the branches about an angle
  dx2 = cos(-G_Angle[k][n])*dx + sin(-G_Angle[k][n])*dy;
  dy2 = -sin(-G_Angle[k][n])*dx + cos(-G_Angle[k][n])*dy;

  // New branch at position and new direction vector
  tree(k,x2,y2,dx2,dy2,sz/G_ShrinkFactor[k][n],n);
}
```

6.3.6 Modelling the reproduction process

The `reproduction()` method makes a copy of the parent's genes and probabilistically mutates them. The expression `if(random(10)<3.3)` makes each mutation conditional and gives it a 33% chance of occurring. Each change is incremental, that is, it adds or subtracts a small amount from the current value of the gene rather than completely altering the value. This is important in order to provide some sense of control over the evolutionary process to the user. The direction of the change in value is determined by the value of `sign` which is set to one of `(+1, -1)` with a 50% chance each. In the implementaion of the `reproduction()` method shown, in order to retain some control of the overall size of the Biomorphs we do not mutate the `G_Height` gene—although this could easily be added in a similar fashion to the other mutations if desired.

```
// Reproduction makes mutated copies of the parent genes
void reproduction() {
  // iterate over all of the child positions in the arrays
  // (array indices > 0)
  for (int k=1; k<NPROG; k++) {
    // first copy the parent (array index 0)
    for (int j=0; j<LMAX; j++) {
      G_Angle[k][j] = G_Angle[0][j];
      G_ShrinkFactor[k][j] = G_ShrinkFactor[0][j];
    }
```

```
    G_Branching[k] = G_Branching[0];
    G_Height[k] = G_Height[0];

    // now apply some random mutations to the children
    // (in this implementation we do not mutate G_Height)
    int sign = random(2)<1?-1:+1;
    // ... first consider angle mutations
    for (int j=0; j< LMAX; j++)
      if (random(10) < 3.3)
        G_Angle[k][j] = G_Angle[k][j]+sign*(2*PI/128);
    // ... now consider shrink factor mutations
    for (int j=0; j< LMAX; j++)
      if (random(10) < 3.3)
        G_ShrinkFactor[k][j] = G_ShrinkFactor[k][j]+sign*.25;
    // ... and finally consider branching number mutations
    if (random(10) < 3.3)
      G_Branching[k] = max(1,min(G_Branching[k]+sign*1,14));
  }
}
```

6.3.7 Selecting from a population

The user selects from the grid of progeny (or the parent) to start the next generation of mutation and Biomorph drawing using the selected Biomorph as the starting point. Selection on a grid is performed using some integer arithmetic on the mouse `(x,y)` position. The grid position is converted into an array index using the expression `winner = y*N+x`, with the winner being the index of the selected Biomorph. The genes of this Biomorph are copied into the array slot of the parent. It is these genes that will be used to create the next generation of Biomorphs.

```
// Allow user selection by mouse press of the best design to be
// used as the parent of the next generation
void mousePressed() {
  int winner;
  // map the mouse coordinates to the index of the selected Biomorph
  int x = mouseX/(width/N);
  int y = mouseY/(height/N);
  winner = y*N+x;
  Selected = true;
  println(winner);
  // copy the genese of the selected Biomorph to become the parent
  // (array index 0) of the next generation
  for (int j=0; j< LMAX; j++) {
    G_Angle[0][j] = G_Angle[winner][j];
    G_ShrinkFactor[0][j] = G_ShrinkFactor[winner][j];
  }
  G_Branching[0] = G_Branching[winner];
  G_Height[0] = G_Height[winner];
}
```

Figure 6.16 shows the entire source code for the Blind Watchmaker sketch. Each of the elements of genes, development and reproduction is put together with a simple

```
// Simple implementation of the Blind Watchmaker
// user-guided genetic algorithm

int N = 4;                // size of selection grid
int NPROG = N*N;          // number of progeny
int SZ = 680;             // screen size
int LMAX = 20;            // maximum branching levels
boolean Selected = true;

// Arrays for genes
int[]     G_Branching    = new int[NPROG];
float[][] G_Angle        = new float[NPROG][LMAX];
float[][] G_ShrinkFactor = new float[NPROG][LMAX];
float[]   G_Height       = new float[NPROG];

void setup() {
  size(SZ,SZ);
  initGenes();
}

void draw() {
  background(255);
  stroke(0);

  // if the user has just selected a design, call
  // reproduction() to create a new generation
  if (Selected) {
    reproduction();
    Selected = false;
  }

  grid();        // draw user interface
  development(); // draw the Biomorphs
}

// Draw a blank grid on which the Biomorphs will be displayed
void grid() {
  // draw the grid lines
  for (int k=0; k<N; k++) {
    line(0,k*height/N,width,k*height/N);
    line(k*width/N,0,k*width/N,height);
  }
  // draw a red hightlight around the parent Biomorph
  stroke(255,0,0);
  noFill();
  for (int k=1; k<5; k++)
    rect(k,k,width/N-2*k,height/N-2*k);
  stroke(0);
}

// Initialise the genes of the parent Biomorph
// (array index 0) with random numbers
void initGenes() {
  G_Branching[0] = (int)random(3)+2;
  for (int k=0; k<LMAX; ++k) {
    G_Angle[0][k]=random(64)*PI/64;
    G_ShrinkFactor[0][k]=random(20)/20+1.1;
  }
  G_Height[0]=height/(4*N);
}

// Draw the parent and all of its children
void development() {
  for (int k=0; k<NPROG; k++) {
    int x = (int)((k%N + 0.5) * width/N);
    int y = (k/N + 1) * height/N;
    pushMatrix();
    translate(x,y);
    tree(k,0,0,0,-1,G_Height[k],G_Branching[k]);
    popMatrix();
  }
}

// tree is a recursive method to draw a single
// Biomorph according to the parameter values
// specified by its genes
void tree(int k, float x1, float y1, float dx, float dy,
          float sz, int n) {
  if (n==0) { // check for exit point of recursion
    return;
  }
  float x2 = x1 + sz*dx;
  float y2 = y1 + sz*dy;

  line(x1,y1,x2,y2);

  n--; // Change level of branching

  // Rotate the branches about an angle
  float dx2 = cos(G_Angle[k][n])*dx + sin(G_Angle[k][n])*dy;
  float dy2 = -sin(G_Angle[k][n])*dx + cos(G_Angle[k][n])*dy;

  // New branch at position and new direction vector
  // by recursively calling the tree method
  tree(k,x2,y2,dx2,dy2,sz/G_ShrinkFactor[k][n],n);

  // Symmetry: Rotate the branches about an angle
  dx2 = cos(-G_Angle[k][n])*dx + sin(-G_Angle[k][n])*dy;
  dy2 = -sin(-G_Angle[k][n])*dx + cos(-G_Angle[k][n])*dy;

  // New branch at position and new direction vector
  tree(k,x2,y2,dx2,dy2,sz/G_ShrinkFactor[k][n],n);
}

// Reproduction makes mutated copies of the parent genes
void reproduction() {
  // iterate over all of the child positions in the arrays
  // (array indices > 0)
  for (int k=1; k<NPROG; k++) {
    // first copy the parent (array index 0)
    for (int j=0; j<LMAX; j++) {
      G_Angle[k][j] = G_Angle[0][j];
      G_ShrinkFactor[k][j] = G_ShrinkFactor[0][j];
    }
    G_Branching[k] = G_Branching[0];
    G_Height[k] = G_Height[0];

    // now apply some random mutations to the children
    // (in this implementation we do not mutate G_Height)
    int sign = random(2)<1?-1:+1;
    // ... first consider angle mutations
    for (int j=0; j< LMAX; j++)
      if (random(10) < 3.3)
        G_Angle[k][j] = G_Angle[k][j]+sign*(2*PI/128);
    // ... now consider shrink factor mutations
    for (int j=0; j< LMAX; j++)
      if (random(10) < 3.3)
        G_ShrinkFactor[k][j] = G_ShrinkFactor[k][j]+sign*.25;
    // ... and finally consider branching number mutations
    if (random(10) < 3.3)
      G_Branching[k] = max(1,min(G_Branching[k]+sign*1,14));
  }
}

// Allow user selection by mouse press of the best design to
// be used as the parent of the next generation
void mousePressed() {
  int winner;
  // map the mouse coordinates to the index of the selected
  // Biomorph
  int x = mouseX/(width/N);
  int y = mouseY/(height/N);
  winner = y*N+x;
  Selected = true;
  println(winner);
  // copy the genese of the selected Biomorph to become the
  // parent (array index 0) of the next generation
  for (int j=0; j< LMAX; j++) {
    G_Angle[0][j] = G_Angle[winner][j];
    G_ShrinkFactor[0][j] = G_ShrinkFactor[winner][j];
  }
  G_Branching[0] = G_Branching[winner];
  G_Height[0] = G_Height[winner];
}
```

Figure 6.16: Source code for the Blind Watchmaker algorithm.

user interface, operated by mouse clicks on a grid, to make a powerful evolutionary design application.

The general principles, and code, for this system can be adapted to many different applications and styles of drawing for a general purpose creativity engine to assist and inspire your own design processes.

6.4 Summary

In this chapter we have explored methods for creating complex drawings by generative methods. Each system had parameters that were updated according to some rules and translated into a drawing by interpreting the rules. Such systems are in common use in today's creative computing applications, and are commonly used for visual effects in films. Given these generative system foundations, you should explore further what computation is capable of producing in terms of creative output. By gaining control over the drawing process with parameters and rules, we can automatically try out many different possibilities until we discover a combination that produces an exciting new creative output.

You should now be able to:

- use *Processing* to create pictures using fractals, including Koch curves
- use L-systems for plant modelling, including two-handed and bracketed systems
- use genetic algorithms for image creation.

6.5 Exercises

1. Write a *Processing* sketch to draw Koch snowflakes.
2. Take the source code shown in Figure 6.4, that draws a simple quadratic Koch island. Figure 6.3 shows three iterations of the program. Run the code for more iterations, and see what you get. Also, modify some of the parameters to change the kinds of images created. This should help you to get a feel for ways in which you can be creative using fractals.
3. Use the code given in Section 6.3 and in Figure 6.16 to create a picture using genetic algorithm approaches. Try to include a creative element, so that you don't only produce a Biomorph, but you use, for example, combinations of Biomorphs in a composition, or you include colour.
4. The concept of fractals can be applied to music creation, as well as to image creation. Find out what you can about this. Which musicians have used generative approaches in their work?

Chapter 7

Introduction to Creative Thinking

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].

The ideal is the works of engineering inspired by the soul of an artist.

[architect Santiago Calatrava, from an interview with Charlie Rose, May 22nd, 2002]

The following chapter provides you with an initial framework to help you develop your own individual creative thinking method and apply it to the programming skills you have acquired so far.

Learning how to draw and animate shapes on the screen enables you to start experimenting with your own creative ideas[1]. Your ideas may initially be influenced by programs and applications you have seen before. However, each student has the ability to develop their own creative process and original thinking. We will start here by defining which qualities are necessary for a successful creative outcome.

7.1 Essential components: technology, usability, aesthetics

The Spanish architect Santiago Calatrava defines his creativity with reference to his skills as an artist, an architect and an engineer:

I am interested to emphasize the relation between art, architecture and engineering.

In this context, a Calatrava bridge is conceived as three things simultaneously: a durable engineering construction capable of resisting the elements, an architectural design functioning according to the necessities of usability, and an artistic creation which enhances the landscape. An example of one of Calatrava's bridge designs is shown in Figure 7.1.

[1] For an insight on how other artists and programmers work with their creative ideas, see "Interviews 1: Print" (p.155); "Interviews 2: Software Web" (p.261); "Interviews 3: Animation, Video" (p.377); "Interviews 4: Performance, Installation" (p.501), all from: *Processing: A Programming Handbook for Visual Designers and Artists* by Casey Reas and Ben Fry, with introduction by John Maeda, The MIT Press, 2007.

Figure 7.1: The bridge leading to the Quadracci Pavilion at the Milwaukee Art Museum, designed by Santiago Calatrava. Photograph published under Wikimedia Commons at `http://en.wikipedia.org/wiki/File:Milwaukee_Art_Museum_1_(Mulad).jpg`.

The problem of creating an instance of interaction in a digital medium can similarly be viewed as a combination of three fundamental elements:

- a mathematical problem which needs to execute seamlessly within its digital environment
- a piece of design which addresses usability issues efficiently
- a piece of art which generates aesthetic impact within its cultural context.

It is therefore essential to understand the opportunities offered by technology at a particular moment in time, the user behaviour related to the subject matter, and the cultural context within which one is operating.

The website for Magnum photographer Carl de Keyzer[2], an example page of which is shown in Figure 7.2, was designed by the Belgian design studio Group94[3] and has received several awards for successfully uniting art, usability and programming.

Group94 have here considered all of the relevant elements that make for a successful project. They have designed the site with full awareness of available technologies, downloading times, user behaviour and artistic language appropriate for a Magnum photographer:

- Technically, the site runs smoothly despite the fact that it contains thousands of quality images. It is administered by the photographer at his own convenience through a specially designed content management system. Loading times are reduced as much as is feasible.

[2] `http://www.carldekeyzer.com/`
[3] `http://www.group94.com/`

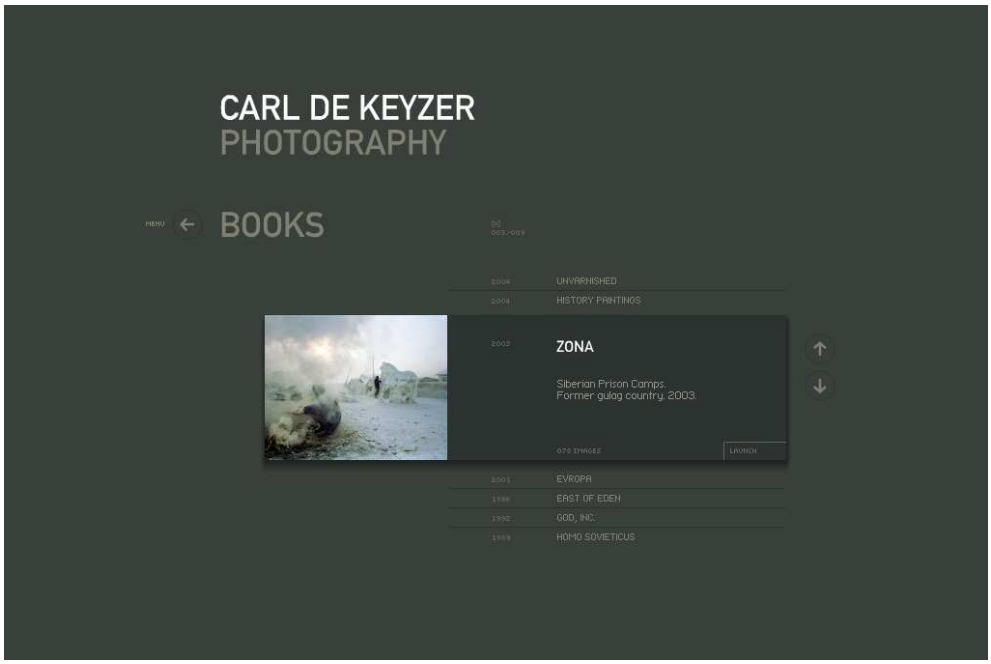


Figure 7.2: Screenshot from the Carl de Keyzer website designed by Group94, courtesy of Pascal Leroy. `http://www.carldekeyzer.com`

- The navigation of the site is very clear and easy to use, despite the extensive content of news, guestbook, biography pages and a huge portfolio section. The user is given the option, when viewing photographs, of either scrolling by numbers or via a series of thumbnails.
- Magnum is one of the most reputable co-operatives of photographers in the world, all of whom have been recognised for their highly artistic achievements. Group 94 have approached the task of creating a website for one of those highly creative photographers, by making the navigation and information monochrome and subtle, and thus allowing the focus to stay on the principal subject matter, i.e. the photographs; their composition, colour and contrast.

7.2 Technology: Flash Professional, Blender and HTML5

The *Processing* programming techniques covered in this course provide a good background for coding in a variety of different media. Over the last decade, one of the most successful tools for creative coding has been the *Adobe Flash Professional* authoring environment. It not only has provision for coding, but also a moving image timeline and video editing capabilities, all of which enable a variety of creative activities.

Flash is an integrated development environment, originally designed as a vector-based web animation tool for efficient deployment of linear animation on the web, by providing drawing tools and a timeline. It also provides motion tweening[4] and streaming of MP3s, as well as scripting capabilities in the form of the *ActionScript* language which considerably boosts the program's capabilities. The

[4] `http://en.wikipedia.org/wiki/Tweening`

versatility of the Flash timeline, combined with the scripting environment, provides new opportunities for the design of a variety of interactive web tools, including various types of navigation, games and presentation of photographic and video content.

ActionScript is a scripting language which has syntax similar to JavaScript, though a different programming framework and set of class libraries. Used in conjunction with the XML capabilities of the browser it enables the deployment of rich content to the browser.

In addition to the extensive opportunities offered by ActionScript, Adobe Flash Professional has a forms-based development environment, filter effects and blending modes, bitmap caching, a type-rendering engine, the ability to import a variety of files, mobile and device development and integration of video editing content.

One of the most successful websites dedicated to Flash creativity is `http://www.yugop.com/`. The author Yugo Nakamura is one of the pioneers of Flash interaction, notable for successfully deploying the use of code to a high artistic level, as illustrated in Figure 7.3. It is instructive to visit the website and see this art in action. Other examples of Flash interactions can be found on the Adobe website (`http://www.adobe.com/`) and on the Infosthetics website (`http://www.infosthetics.com/`).

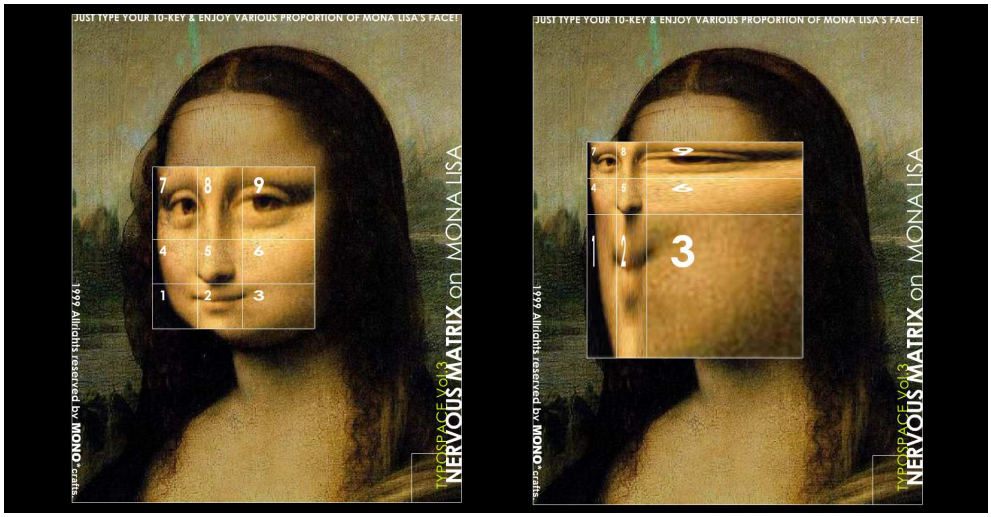


Figure 7.3: Screenshots from Yugo Nakamura's website, courtesy of Yugo Nakamura.
`http://www.yugop.com`

Adobe Flash Professional is a commercial product for which you require a licence for normal use. You can explore the potential of the package by downloading a free trial version from `https://creative.adobe.com/products/flash`.

Flash is not appropriate in all situations, and various alternatives are available. A major limitation of Flash is that Apple does not support it on any of its iOS devices. An alternative to Flash, with particular emphasis on 3D animation, is the free, open-source *Blender* software, available at `http://www.blender.org/`. A variety of other packages, both commercial and free, are also available. In addition, as the HTML5 standards and associated API technologies[5] mature, it is becoming

[5] `http://en.wikipedia.org/wiki/HTML5`

increasingly possible to code sophisticated creative applications using native web technology. As always, it is sensible to investigate the various options and decide which one best suits your needs.

7.3 User behaviour

How does technology affect us and how do we affect technology with our behaviour?

According to Tony Dunne, of Dunne and Raby, and Professor of Design Interactions at the Royal College of Art, London,

> *. . . we see ourselves as complex individuals moving through an equally complex, technologically mediated, consumer landscape. Interaction may be our medium in this department, but people are our primary subject, and people cannot be neatly defined and labelled. We are contradictory, volatile, always surprising. To remember this is to engage fully with the complexities and challenges of both people and the field of interaction design*[6].

While in the process of designing an instance of interaction, we need to consider the user's needs, wants and limitations, as well as their likely age, cultural background and education. We need to imagine, and also test, how the new product we have created will impact upon their perceptions and experiences.

A low-tech project which makes excellent use of the knowledge of user behaviour is "Railings", by Greyworld, an art group which focuses on interactive art installations, mostly using modern technology.

Figure 7.4: An image from the Greyworld project "Railings", courtesy of Andrew Shoben.

When Greyworld were asked to liven up a run-down inner urban area in Britain, rather than altering the environment through the use of shape or colour, they considered how children imaginatively use mundane things to amuse themselves,

[6] `http://www.design-interactions.rca.ac.uk/programme`

and, particularly in this case, how they pick up a stick and run it along the metal railings of a playground, in order to produce a sound.

With this behaviour in mind, Greyworld set to physically tune the railings of the urban area in question, so that when a child runs a stick along them, they play "Girl from Ipanema".

Figure 7.4 is one image from the project; look at the `http://greyworld.org` website to see other images, and other projects.

You should focus on this kind of thinking when creating any type of interaction, regardless of the level of complexity of the technology used. From the design of the first computer mouse to the search facilities on Google, interaction designers have always taken user behaviour as a starting point.

In *Designing Interactions*, Bill Moggridge looks at "how the needs and desires of people can inspire innovative designs and how prototyping methods are evolving for the design of digital technology". You can look at the variety of different ways in which interaction designers consider human behaviour on the book's website[7].

7.4 Cultural context

A piece of interaction can stimulate social, ethical and cultural debate about issues as well as address the practical needs of users. It is essential to develop an acute awareness of your environment and the culture within which you operate. When creating an instance of on-screen interaction, you are making a piece of visual communication. The visual language you use has to be comprehensible to your target audience.

In order to create successful visual communication, you need to be aware of your audience's social and cultural heritage, as well as their current status and cultural preferences.

Look at a picture of the 1993 work by Damien Hirst entitled "*Mother and Child, Divided*" on the Tate Gallery website.[8] It uses a dissected cow and calf in formaldehyde. Consider how this piece could be perceived in different cultural contexts:

- on a street in Bombay, particularly in view of the fact that in Hinduism the cow is considered sacred
- in the middle of a Mediaeval battle, surrounded by violence and chaos
- next to an Eskimo skinning seals, surrounded by blocks of ice
- in a shopping trolley, an item typically used in the Western consumer environment.

Damien Hirst's piece uses a language which is conditioned by time, space and the culture which created it. It can be understood within the context of Western consumer culture, and within the Western heritage of 20th Century Art movements such as Surrealism, Dada and Pop Art.

[7] `http://www.designinginteractions.com/`
[8] `http://www.tate.org.uk/art/artworks/hirst-mother-and-child-divided-t12751`

Cultural history is a fundamental part of the cultural awareness of our present. The works of Salvador Dali, René Magritte, Man Ray, Marcel Duchamp and Andy Warhol are now being commonly used as part of the Western visual language and can often be seen as influences in advertising and commercials.

In an early example of art meeting advertising, the Belgian Surrealist painter René Magritte painted "The Treachery of Images" which shows a drawing of a pipe but underneath states "Ceci n'est pas une pipe" or "This is not a pipe". The statement appears to be a paradox at first, though it stimulates thought and makes the viewer eventually question whether a drawing or pictorial representation of a pipe can be defined by the object it represents.

Magritte became well known for witty images which played on convention and our habitual perception. The style he used was representational and already widely used in advertising in 1929 when the image was created. Magritte was here questioning the status of the image within the context of advertising and the emerging consumer culture.

The debate initiated by this painting was later explored in philosophical works by Michel Foucault[9] and Douglas Hofstadter, as well as by a variety of artists and designers, either as a direct reference, or as a way of using imagery to stimulate thought.

There are other examples which show clearly the influences Magritte and other Surrealists have had on more recent works by artists and designers.

The famous advertising campaign by Collett Dickenson Pearce[10] for Benson & Hedges[11] from the 1970s borrows heavily from Salvador Dali's surrealist language, such as his "The Persistence of Memory", shown in Figure 7.5[12]. Guinness[13] and Smirnoff [14] advertising campaigns have also become famous for using surrealist elements.

The same language can also be seen in the work of a variety of animators and film makers, such as Terry Gilliam's animations for Monty Python[15] or the work of director David Lynch[16].

In 2006 Dan Kurtz created a laser-etched PowerBook entitled "Ceci n'est pas un PowerBook", shown in Figure 7.7. The work makes very clear references to its cultural heritage. It is a re-working of Magritte's signature painting "The Son Of Man" (Figure 7.6[17]) and the title refers to the text in "Treachery of Images", with the additional surrealist twist of utilising a well-known modern brand identity as part of the composition.

[9] http://foucault.info/documents/foucault.thisIsNotaPipe.en.html
[10] http://en.wikipedia.org/wiki/Collett_Dickenson_Pearce
[11] http://www.alastairmcintosh.com/images/bensons.htm
[12] The use of this image falls under the MoMA "fair use" policy. http://www.moma.org
[13] http://www.youtube.com/watch?v=ueKvBThaqR4
[14] http://www.youtube.com/watch?v=cbfDGMfXimA
[15] http://www.youtube.com/watch?v=D1BKtrG7qxQ
[16] http://www.youtube.com/watch?v=T4tOvkvCvwQ
[17] The use of this image falls under the Wikipedia "fair use" policy. For further information, see http://en.wikipedia.org/wiki/File:Magritte_TheSonOfMan.jpg

Figure 7.5: Salvador Dali's "The Persistence of Memory", 1931, oil on canvas.

Figure 7.6: René Magritte's "The Son of Man", 1964, oil on canvas.

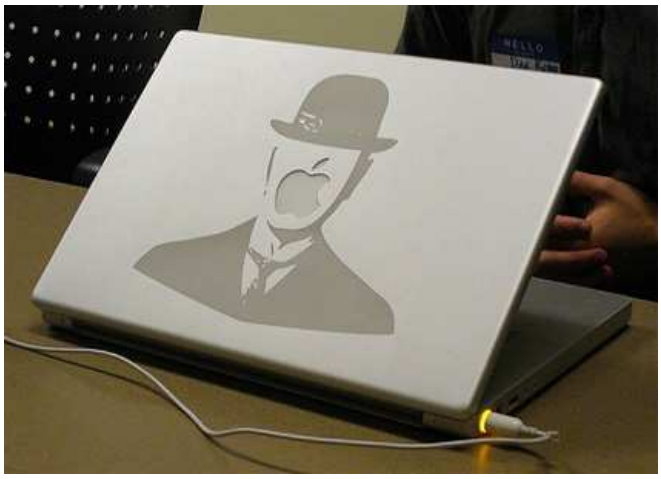


Figure 7.7: Dan Kurtz's laser-etched PowerBook, reproduced courtesy of Dan Kurtz. Photo by Steve Rhodes.

7.5 Summary

This chapter has focused on the creative aspects related to the technical work that the rest of the guide has developed. It has given some insights into how you can begin to create your own artistic works, using the programming skills you have acquired in the rest of the course.

You should now be able to:

- describe the way art, design and technology interact in creative artefacts such as architectural works
- understand the potential of packages such as Adobe Flash Professional and Blender for creative coding
- incorporate an understanding of user behaviour into your design
- discuss how cultural context impacts on the experience of an artwork, and reciprocally, incorporate an understanding of this into your own creative works
- work through the creative brief in Appendix A.

7.6 Exercises

1. The following short exercises are provided to help you develop your understanding of the material in the guide. It is however most important that you work through the Creative Brief in Appendix A, as described in Exercise 2, in order to develop your ability and skills in the creative domain.

   (a) Find another example of an architectural work that satisfies the three components mentioned by Calatrava in Section 7.1. Describe how this new example demonstrates these properties. Find an architectural work that you believe does not demonstrate all of the properties, and critically evaluate it.

(b) Use Blender, a trial version of Adobe Flash Professional, or the native HTML5 APIs, to take an artwork from the creative commons, and add a new dimension to it. Describe how your new dimension extends the work.

(c) The guide shows some examples of where well-known art works have informed media campaigns. There are other examples—find some more of these and describe how they are connected to the art that has inspired them.

2. Appendix A is a brief for you to develop a creative work or works, using what you have seen in the preceding chapters of the guide. Work through the brief to produce a portfolio of work in the theme of your choice, but in line with that directed by the constraints of the brief.

   Once you have done this, and also while you are doing it, you should also do some critical evaluation of the work you are producing. Part of this includes describing it in relation to existing work you have seen, in this guide and also in the additional reading and investigations of your own that you will have done.

Appendix A

Creative Brief

The following chapter will provide you with some inspiration to interact with your environment, investigate and discover, and develop your own way of looking at the world around you. You will then be encouraged to analyse your discoveries and interpret them using all the skills this course has provided you with so far.

Our environment generates a playground for a variety of behaviours. These can be observed as distinct patterns, some governed by strict rules of law, some by social etiquette, some by simple human interactions.

By observing carefully we can discover interesting patterns of behaviour, interpret them and generate new meaning. We can observe real and imaginary boundaries, map them out, describe them or critique them. We can analyse people's contradictions and complexities, as well as define their similarities.

The exercises in this chapter are designed to help you think about, and develop, your own personal process of creativity.

A.1 Rules of the playground

The aim of this exercise is to use your habitat as your playground. The city or area you live in provides ample grounds for a series of interesting observations. The task is to walk and observe attentively with the aid of either photographic equipment, video footage or sound recordings.

The following is a series of themes one can focus on while making observations:

- rhythm of a network, sound of a pattern
- timing events, observing patterns in time
- measuring distances, taking measures of our behavioural patterns
- spatial relationships, scale, distribution, congestion
- ethical, good and proper
- poetic, symbolic, imaginative, aesthetics, harmony
- sequence, a series of frames
- weather, environmental effects on networks
- mechanical, natural and cyber elements
- networks of material activity, work patterns
- chance, error, luck, richness, diversity and cultural freedom
- representation, maps, movies, text, sound recordings
- impact of networks on society, a perspective on networks

A.2 Observing behaviours

The pictorial, video or audio data gathered forms the core of the project's research. Here are some examples from the visual domain:

[flickr, creative commons, by Banalities]

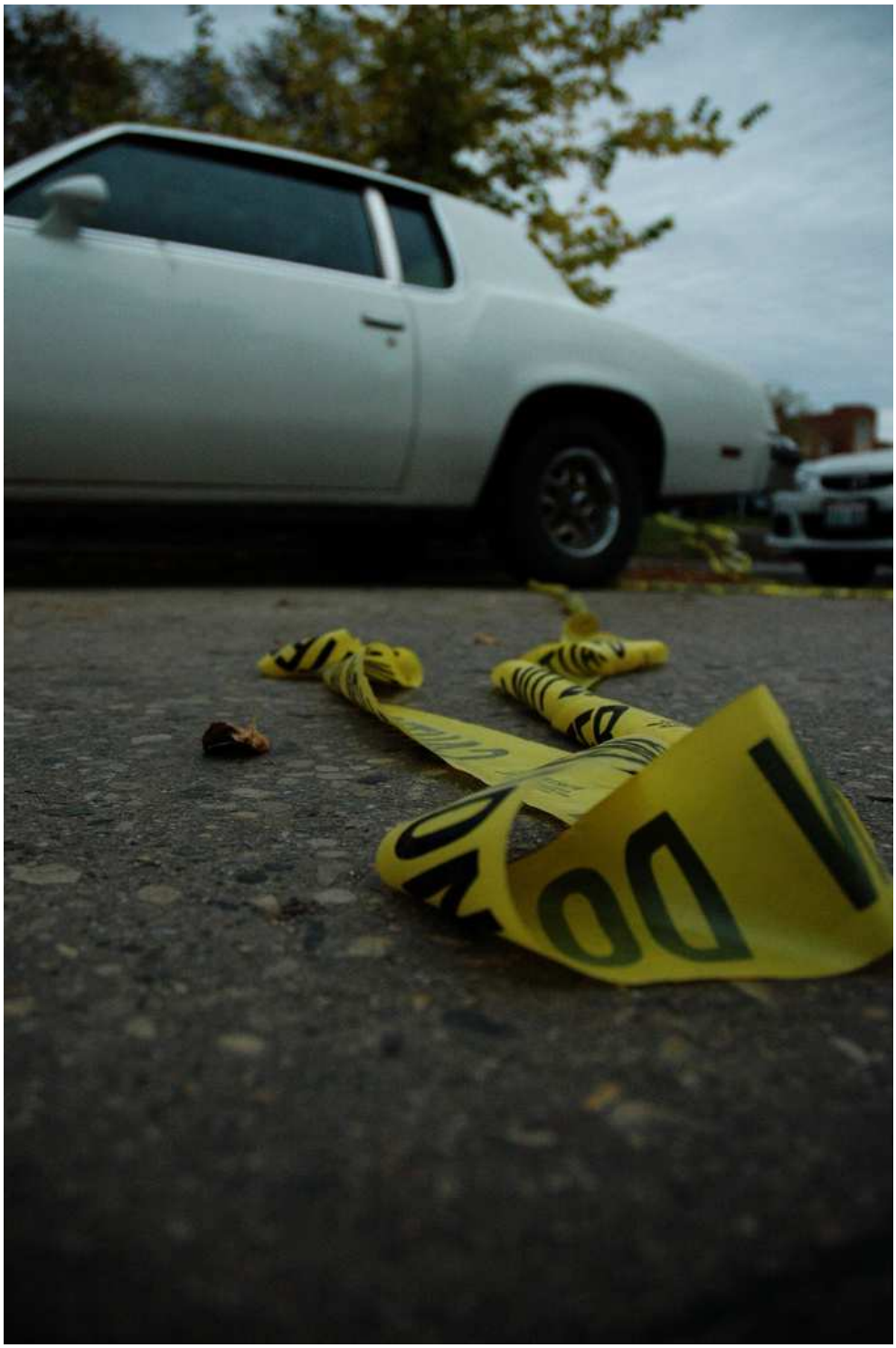
Starting with simple observations, people can be seen forming lines while queuing for public transport or purchasing tickets. They can be seen positioning themselves at a station in front of the expected vehicle entry door position. They can be seen spread with even spacing between them when sitting on benches in public spaces.

[flickr, creative commons, by Jasoon]

The local culture influences pattern formations. On the London underground system it is customary to stand on the right on the escalators, while commuters in a hurry can use the left for fast access.

[flickr, creative commons, by Idolum_Visions]

A group of people going to a concert can often be identified by their style of clothing as well as their direction. United by their common interest, they can be followed like street signage.



[flickr, creative commons, by Rickabbo]

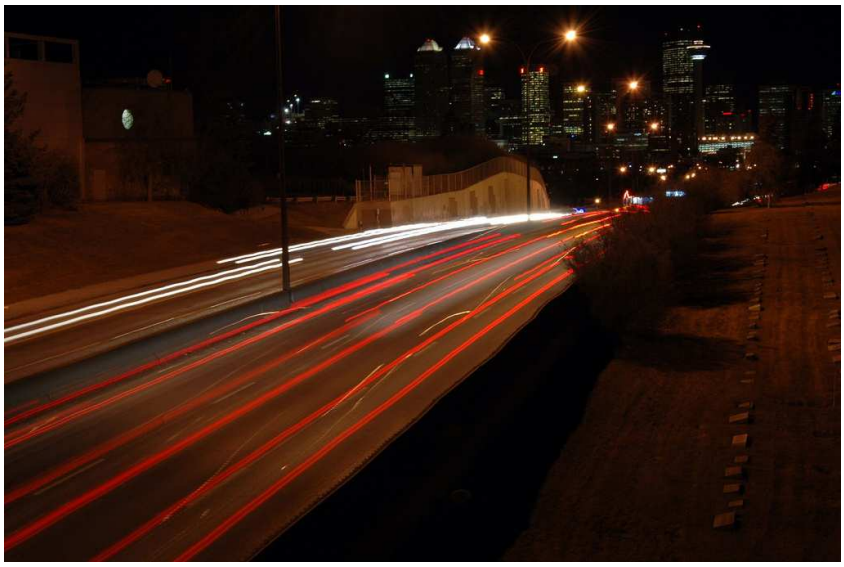
Once we have observed a pattern, we can also start identifying exceptions. Those can be caused by sudden environmental changes, accidents or deliberate decisions by individuals. People will normally avoid areas marked with fluorescent tape as they associate it with a hazard or a restricted area. This can be tested by placing some fluorescent tape in a public area which is not hazardous and observing the

results. It is likely that we will notice a pattern of avoidance.

[reproduced courtesy of Annie Mole,
`http://london-underground.blogspot.co.uk`]

[flickr, creative commons, by Annie Mole]

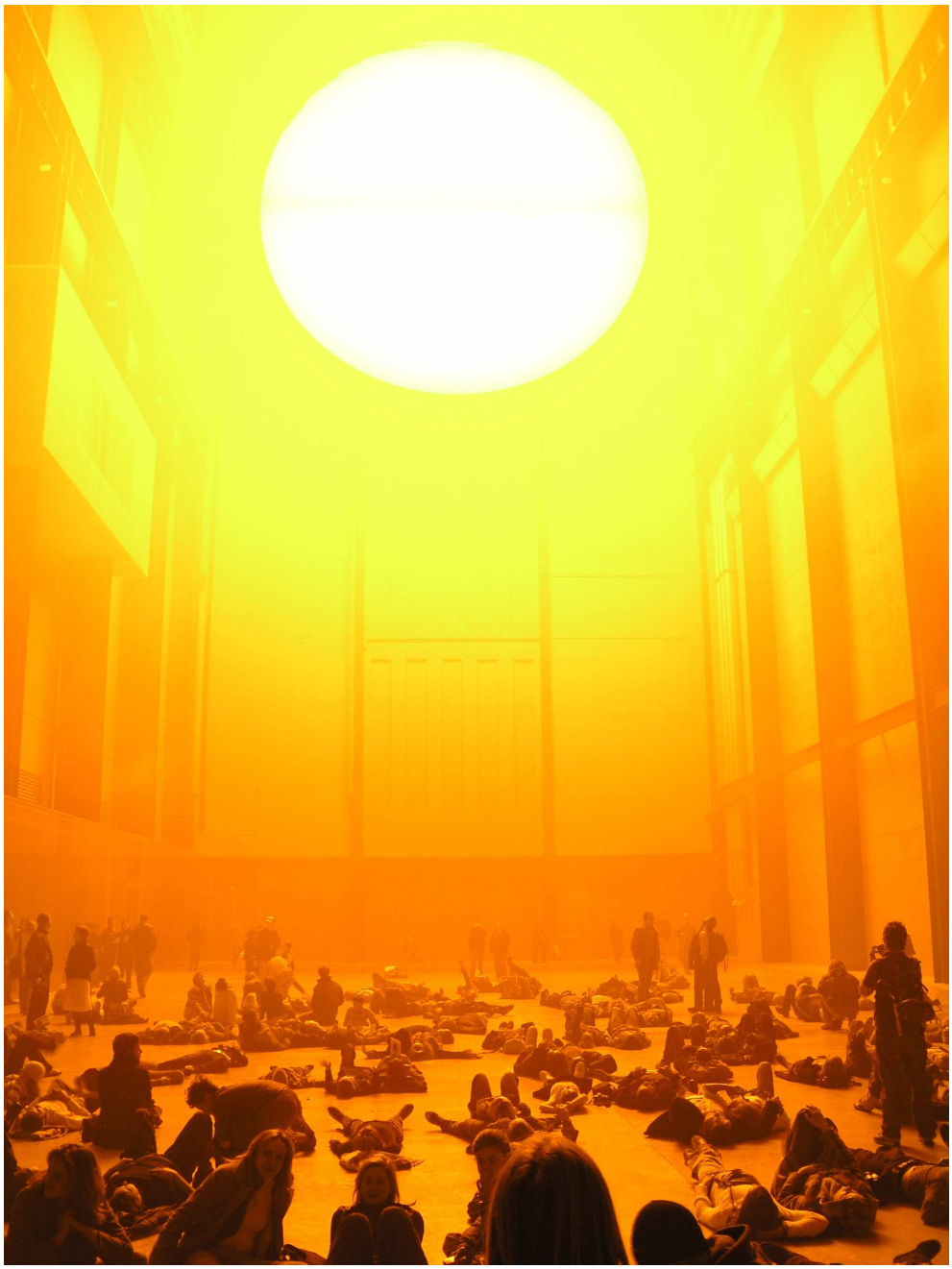
Different types of behaviour can be observed in reaction to restrictions.



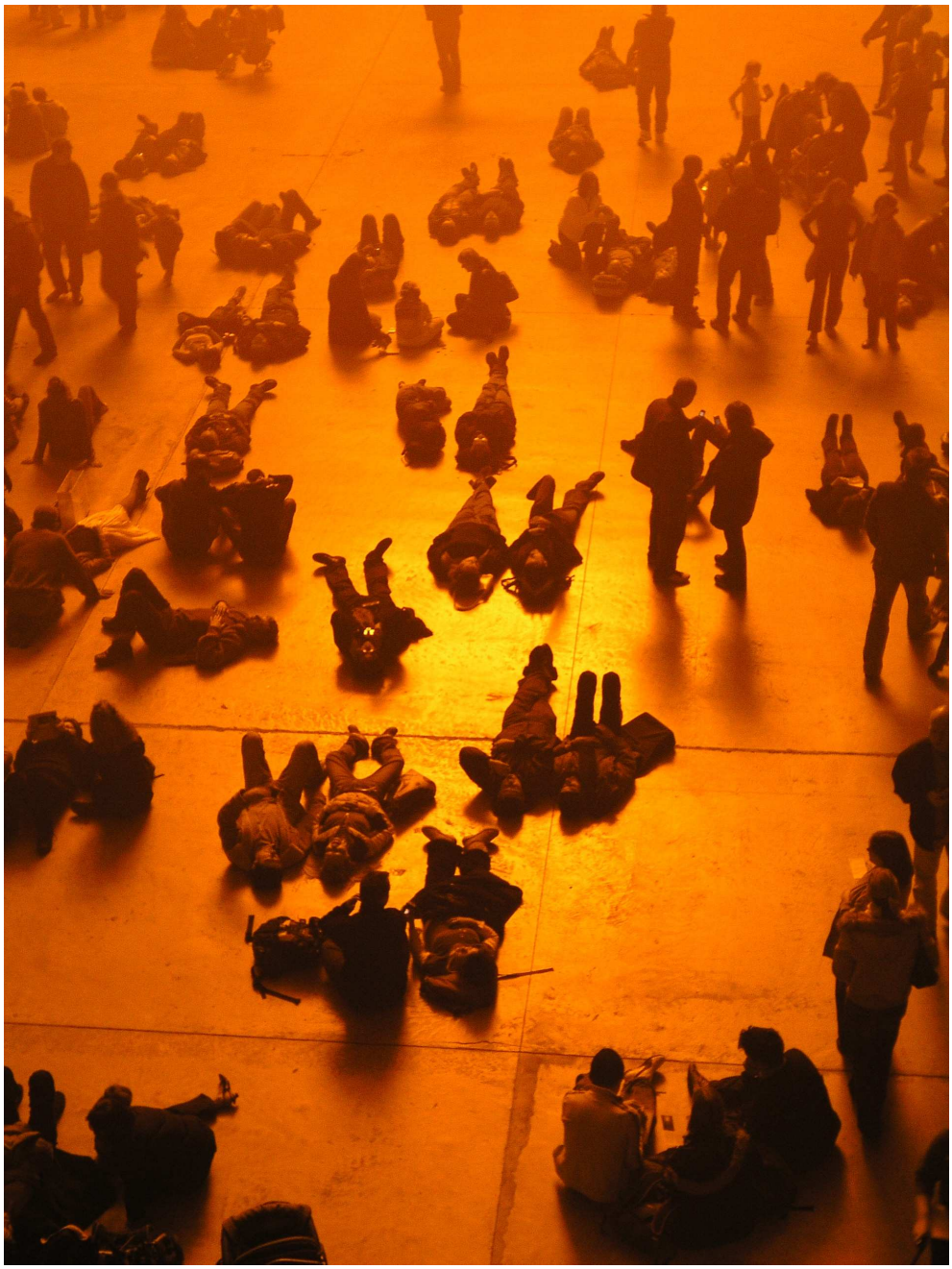
[flickr, creative commons, by Brian U]

[flickr, creative commons, by Djrue]

Certain patterns emerge over time such as the light patterns of the flow of traffic, or at certain times of the day, such as crowds moving in a certain direction during rush-hour.



[flickr, creative commons, by Simiant]



[flickr, creative commons, by Simiant]

Successful art installations, such as Olafur Eliasson's "The Weather Project" in the Turbine Hall at the Tate Modern, London 2004, may encourage intricate behavioural patterns.

A.3 Interpreting your observations

Once you have gathered your data and made your observations, you can develop your project further. Your project should both inform and influence people's perception of the subject matter.

- Analyse all of the images, film footage or sound data that you have gathered. Interpret what you have discovered.
- Identify areas of interest. Find any exceptions, similarities or contradictions.
- Write down all your ideas, sketch out any maps, diagrams or symbols.
- Create any visuals which will help you communicate your ideas. You can use Adobe Photoshop or Illustrator, or any other visual tools available.
- Use the programming skills you have acquired on this course. Write a piece of code based on a discovered behavioural pattern. The visual aspect of it can be entirely abstract. The pattern and motion, however, ought to replicate the organic quality of human behaviour.
- Test your code on a few of your friends and colleagues and note their reactions.

Remember:

> *even the simplest operations can have unexpected consequences, especially when executed with feedback and interactivity. By using programming to harness feedback and interactivity it is possible to be repeatedly rewarded with emergent phenomena and happy surprises.*

[Ed Burton of Sodaplay, from *Processing, a Programming Handbook for Visual Designers and Artists*, p. 264]

The more you observe, the more you will be able to explain and interpret the world around you. In time you will develop your own patterns of discovery. As your thinking evolves, you will create your individual methodology of research and your own process in creativity.

Appendix B

Example Examination Questions

Usually an examination paper will require answering 4 unseen questions out of 6, in 3 hours. Question 1 of the paper will usually be a compulsory multiple choice question. This must be answered along with 3 of the remaining 5 questions. Calculators are normally allowed. The following give examples of the kinds of (non multiple choice) questions that can be expected in an examination; also included are some comments about the kinds of answers that might be given. For many other examples, including multiple choice questions, refer to the archive of past exam papers on the Goldsmiths Computing VLE[1]. Also available on the VLE are examiners' commentaries on students' performance on previous exams—you are strongly encouraged to refer to these in addition to the past exam papers.

Question 1

(a) List 4 ways in which a computer, with appropriate software and peripherals, can aid creativity in visual arts or music.

Discuss, from your own experience, which of the 4 ways you have listed are most productive and which are least helpful, giving examples. [9]

(b) Briefly discuss whether art and mathematics can each be self-contained, separate from any practical function, or whether they are only of value in terms of some application, such as helping in the design of a bridge, a house or another artefact. [8]

(c) Describe some ways in which the internet and tools such as Adobe Flash, have aided artists in devising and presenting their work. Also describe how such tools have changed or extended the nature of the works that artists can devise. [8]

Question 1—notes As a general point, the directions 'discuss' and 'describe' emphasise the requirement for answers to be in a standard essay form, rather than a set of bullet points or ill-connected notes. The ideas given below are examples; simple reproduction in an answer to an actual question would gain little credit.

(a) Ways include: creating images with aspects not possible with traditional media (such as detail in some areas finer than possible with a brush, or simulating non-realistic brush forms, etc.), image processing with precision of effects (e.g. in image duplication for patterning effect can be far more accurate than in a photographic darkroom, or in recursive/fractal effects), translation of a method to image (via program code), creation of artificial instruments with sounds not attainable on any existing real instrument, etc.

Discussion of your own experience would account for 5 of the marks. These marks would be given for coherence of explanation, feasibility and degree of effect, and clarity of demonstrating how the most productive ways would

[1] direct URL: `https://computing.elearning.london.ac.uk`

substantially ease production compared to traditional methods or enable something not possible in any other way.

(b) Reference should be made to topics covered in the Subject guide and essential reading, to outcomes from assignments carried out as part of the unit, and any personal experience or development. For example, in prehistory it is unclear if cave paintings of animals were a simple recording and celebration of past hunts or a magical (functional) device to help with future hunts; e.g. most mediaeval Western religious paintings had the intentional function to educate (e.g. the illiterate) about the scriptures and enhance religious or moral messages (e.g. the goodness of god or the consequences of sin); much Western modern art is about expressing the feelings or experience of the painter (but then this has an effect on the viewer and may change behaviour in positive or negative ways).

In mathematics, much of the early work was practical, e.g. for land measurement, geometry for land allocation, etc., while a modern mathematician may consider work in the context of some extension or generalisation of ideas.

Thus the quality of the answer will partly depend on the coherence of arguments as to the intention of the artist or mathematician in their creation as an item in its own right versus the need for funding to provide the opportunity of creation (where the funder may have expectations of applicability). Mention can be made of conscious, systematic attempts (e.g. Bauhaus) to 'return' the artist to the role of a valued craftsperson.

(c) Some issues of image manipulation from part (a) may be rephrased here as part of the answer, regarding packages such as Flash and Blender, and HTML5. Issues also include the manner of acquiring (via digital images from camera, video, scanner, internet, etc.) images or image elements. An answer might also include discussion of how far reuse of others' image elements must go in order for the image to be the artist's own work (cf Warhol's reworking of photographs of Marilyn Monroe and Elvis Presley: Warhol explained "I wanted something stronger that gave more of an assembly line effect" — this might also be related to the intentions of the Bauhaus, etc.).

Question 2

(a) A general point in 2-dimensional coordinate space may be denoted in column vector form as:

$$\begin{pmatrix} x \\ y \end{pmatrix}$$

However, when representing a point **P** as a vector for standard manipulation, such as rotation, scaling or translation, the usual representation is:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Why is this form convenient to use? [3]

(b) Derive or directly write down the matrix representations:

(i) **T** for translating a point **P** by TX and TY in the $x-$ and $y-$directions respectively. [2]

(ii) **T** for scaling a point **P** by factors SX and SY in the $x-$ and $y-$directions respectively. [2]

(c) Consider $SX = 2$, $SY = 2$, $TX = 1$, $TY = 1$, and

$$\underline{\mathbf{P}} = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}.$$

Using matrix representation, calculate $x\prime$ and $y\prime$ for the position

$$\underline{\mathbf{P}}' = \begin{pmatrix} x\prime \\ y\prime \\ 1 \end{pmatrix},$$

where

(a) $\underline{\mathbf{P}}\prime$ = **S.T.**$\underline{\mathbf{P}}$ [2]

(b) $\underline{\mathbf{P}}\prime$ = **T.S.**$\underline{\mathbf{P}}$ [2]

(d) Explain under what conditions scaling followed by translation has the same effect as translation followed by scaling. [4]

(e) Consider the following fragment of *Processing* code, where `width` and `height`, both value 512, are the dimensions of the output screen.

```
size(512,512,P3D);
noFill();
background(255);
translate(width*0.75, height*0.75, 0);
stroke(0,255,0);
box(50);
translate(-width/2, -width/4.0, 0);
stroke(0,0,255);
box(100);
```

(i) State the colours used for drawing the first and second boxes. [2]

(ii) Sketch a boundary line for the output screen and within it sketch the objects drawn by *Processing* (exact positions are not needed, just the relative positions, sizes and appearances). [4]

(iii) If the line
`box(100);`
were to have the following additional line inserted just before it:
`scale(1,3,2);`
sketch the new appearance of the output screen. [4]

Question 2—notes

(a) See Subject guide. Particularly because of the translation operation, the number of columns in a transformation matrix is one greater than the number of spatial dimensions. To make the matrices commensurable, a dummy equation 1=1 is added to the transformation set and leads to a supplementary value of 1 in the position vector.

(b) See Subject guide, Volume 2, Section 2.6.2.

(c) As in all calculation questions, show your intermediate working (e.g. result of translation on x and then the result of scaling on that value) so that if you make an arithmetic error you can gain partial marks. If you just write an answer with no working then it will only gain marks if the answer is completely right. No more than half marks if working is not shown in matrix form.

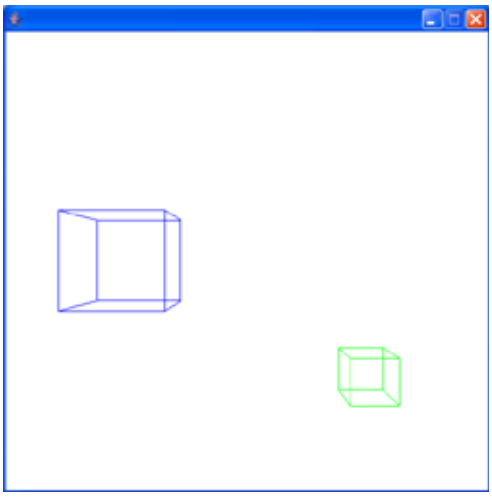
(i) $x\prime = 6, y\prime = 4$

(ii) $x\prime = 5, y\prime = 3$

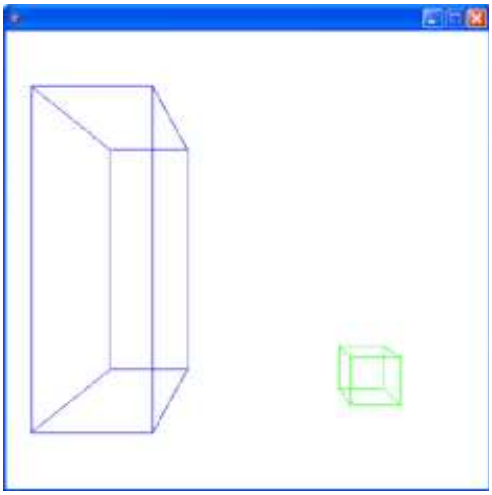
(d) Here it is best to work out the symbolic form of **S.T** and **T.S** and then explain or calculate what combinations of values of SX, SY, TX, TY mean that **S.T** and **T.S** have the same value. For example, trivially $SX = SY = TX = TY = 0$ will leave all points at the origin in both cases.

(e) (i) First box green, second box blue.

(ii) The screen should look as follows:



(iii) The screen should look as follows:



Question 3

(a) (i) If a tree branches into two new branches at each level, how many branches are there at 6 levels of branching? You may assume that the first level is the trunk. [3]

(ii) What is recursion and how is it performed in *Processing*? [3]

(iii) What is the length of a branch at three levels of branching with shrink factor 2 and first level branch length 100 (the trunk)? [3]

(b) Modify the following (supplied) code to draw a tree with two branches at each level [6]

```
int Branching=7;

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

void draw(){
  translate(width/2.0,7*height/8.0);
  tree(height/4,PI/5,1.41,Branching);
}

// make a tree by recursion
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
  --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
  popMatrix();        // restore coordinates
}
```

(c) Modify the code in (b) to draw the tree such that the angle of branching is controlled by the x position of the mouse (range 0 to PI) and the shrink factor is controlled by the y-position of the mouse (range 1.0 to 2.0). [10]

Question 3—notes

(a) (i) If first level is trunk (1 stem = 2^0), 2nd level will have 2 (2^1) branches, so nth level will have $2^{(n-1)}$ branches.
Thus 6th level will have $2^5 = 32$ branches. Thus total branches is $32+16+8+4+2+1 = 2^6 - 1 = 63$ branches.

(ii) See Subject guide, Volume 1, Section 7.6. Recursion occurs when a process calls upon itself with some condition for terminating the process. In *Processing* a method calls itself until a terminating condition is met.

(iii) Shrink factor of 2 halves length compared to previous stage. So length 100 at stage 1 means length 50 at stage 2 and length 25 at stage 3.

(b)

```
int Branching=7;

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

void draw(){
  translate(width/2.0,7*height/8.0);
  tree(height/4,PI/5,1.41,Branching);
}
```

```
// make a tree by recursion
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
  line(0,0,-sz/2,-sz);  // draw second line from same point -- added
  --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
  rotate(-a);           // this and next line reset position -- added
  translate(0,sz);      // added
  translate(-sz/2,-sz); // move to end of second branch -- added
  rotate(a/2);          // rotate a different angle -- added
  tree(sz/sf,a,sf,n);   // added
  popMatrix();          // restore coordinates
}
```

(c)

```
int Branching=7;
float ang, shrink;  // added

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

void draw(){
  fill(0);
  rect(0,0,512,512);
  // simple mouse position use as in Subject guide examples - ADDED PART
  // tree redrawn while mouse pressed and stays as last mouse position
  // when mouse button released
  if (mousePressed) {                // added
    ang=PI*mouseX/width;             // added
    shrink=2.0-float(mouseY)/height; // added
  }                                  // added
  translate(width/2.0,7*height/8.0);
  tree(height/4,ang,shrink,Branching);
}

// make a tree by recursion
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
  line(0,0,-sz/2,-sz);  // draw second line from same point
  --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
  rotate(-a);           // this and next line reset position
```

```
    translate(0,sz);      //
    translate(-sz/2,-sz); // move to end of second branch
    rotate(a/2);          // rotate a different angle
    tree(sz/sf,a,sf,n);   //
    popMatrix();          // restore coordinates
  }
```

Question 4

(a) What are the three components of the RGB colour model and how are they represented in *Processing*? [3]

(b) Explain, line by line, what the following code produces: [2]

```
size(100,100);
colorMode(RGB);
fill(0xFFFF00FF);
rect(0,0,100,100);
```

(c) Write *Processing* code to make the following:

A magenta square with edge length 100 pixels at the top-left of a screen of width and height 200. Add to this a yellow triangle, with transparency 50%, with edge length 100 pixels, overlapping the upper right half of square, so that the hypotenuse of the triangle is a diagonal of the square.

[3]

(d) Consider the *Processing* class called `PImage`.

(i) How many bits represent each pixel in a `PImage` variable? [1]

(ii) What does the `PImage` method `get()` do? [2]

(iii) What does the `PImage` method `set()` do? [2]

(iv) What does the `PImage` method `image()` do? [2]

(e) Images `FigA.gif` and `FigB.gif`, which are the same size as each other, are taken to be in the data folder of the current sketch.

(i) Write a program to load `FigA.gif` and display it in a correctly-sized window. [3]

(ii) Write a program to load `FigB.gif` the image and display it upside down. [3]

(iii) Write a program to load `FigA.gif` and `FigB.gif` and combine them so that the darkest colour is shown at each position. [4]

Question 4—notes

(a) RGB: see Subject guide Volume 2, Chapter 1 Red; Green; Blue (as successive hexadecimal values in *Processing* colour value, each of value 0 to 255).

(b) No transparency, red+blue(magenta), 100x100 rectangle.

(c)
```
size(200,200);
fill(0xFFFF00FF);
rect(0,0,100,100);
fill(0x7FFFFF00);
triangle(0,0,100,0,100,100);
```

(d) (i) 32

(ii) Reads the colour of any pixel or grabs a section of an image, etc. (see reference).

(iii) Changes the colour of any pixel or writes an image directly into the display window. The x and y parameter specify the pixel to change and the colour parameter specifies the colour value.

(iv) Diplays images to the screen.

(e) (i)
```
PImage test1 = loadImage("FigA.gif");
size(test1.width,test1.height);
set(0,0,test1);
```

(ii)
```
PImage test1 = loadImage("FigB.gif");
size(test1.width,test1.height);
translate(width/2,height/2);
rotate(PI);
translate(-width/2,-height/2);
image(test1,0,0);
```

(iii)
```
PImage test1 = loadImage("FigA.gif");
PImage test2 = loadImage("FigB.gif");
test1.blend(test2,0,0,test2.width,test2.height,0,0,
            test1.width,test1.height,DARKEST);
size(test1.width,test1.height);
set(0,0,test1);
```

Question 5

(a) (i) Briefly explain each of the following *Processing* commands; include in your answer an explanation of the parameters for method calls:

1. `size(x,y,P3D);` [2]
2. `popMatrix();` [2]
3. `vertex(x,y,u,v);` [2]

(ii) Write a *Processing* sketch to draw a rotating planet sphere with radius 100 pixels, and a geo-stationary satellite sphere with radius 10 pixels, with distance between the centres of the spheres of 200 pixels. Geo-stationary means that the satellite rotates about the planet at the same rate as the planet's rotation. The planet/satellite system should be centred in the screen. The planet should rotate on its own axis once per minute at a frame rate of at least 10 frames per second. [5]

(b) (i) Describe what is meant by the term 'painting by gestures' as used in this course. [4]

(ii) Briefly describe some of the gesture techniques used by Jackson Pollock. Include examples of the devices and media that he used. [5]

(iii) Briefly describe at least one way in which some of the methods used by Jackson Pollock can be simulated using Processing. Include mention of any event detection that needs to take place for 'painting' to take place. [5]

Question 5—notes

(a) (i) See *Processing* reference guide and books.

1. Sets the width and height of the display window to `x` and `y`, respectively, and specifies the `P3D` renderer for drawing 3D scenes.

2. Pops the current transformation matrix off the matrix stack. The `pushMatrix()` function saves the current coordinate system to the stack and `popMatrix()` restores the prior coordinate system. `pushMatrix()` and `popMatrix()` are used in conjuction with the other transformation methods and may be embedded to control the scope of the transformations.
3. All shapes are constructed by connecting a series of vertices. `vertex()` is used to specify the vertex coordinates for points, lines, triangles, quads, and polygons and is used exclusively within the `beginShape()` and `endShape()` function.

(ii) (as in Subject guide, Volume 2, Chapter 3)

```
float a;
void setup(){
  size(512,512,P3D);
  frameRate(10);
  sphereDetail(10);
  noFill();
}
void draw(){
  background(255);
  translate(width/2,height/2);
  rotateY(a);
  stroke(255,0,0);
  sphere(100);
  translate(200,0);
  stroke(0,0,255);
  sphere(10);
  a+=2*PI/(10*frameRate);
}
```

(b) In all parts, credit given for enriching your answer with your own added understanding and examples. Also, see Subject guide, Volume 2, Section 3.5.3.

(i) Gestures are the dynamical processes that create the effects in a picture that relies on motion for its creation.

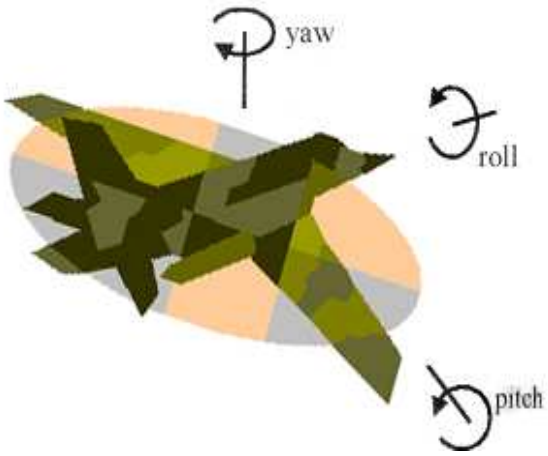
(ii) JP used dripping, throwing, etc. from brushes, cans, etc. onto canvas (e.g. horizontal or vertical).

(iii) The example 'flies' a 'canvas' through a space with paint element objects. In this case detection of when the canvas intersects with the paint is needed to know when to register the colour on the 'canvas'.

Question 6

(a) Outline the operation of the function `camera` in *Processing*, including specifying the meaning and use of each of the nine parameters. [6]

(b) A flight simulation program allows the following motions in addition to thrust and reverse thrust:



The following code is provided; it is a 3D world consisting of a pilot view—as if from a plane—and a runway. The plane is in forward motion.

```
float velocity=2;
float z=1000;

void setup(){
  size(512,512,P3D);
  stroke(0);
}

void draw(){
  background(255);
  noFill();
  // DRAW PILOT's VIEW FROM PLANE
  beginCamera();
  camera(0,0,0,0,0,-1000,0,1,0);
  translate(width/2,height-200,z);
  endCamera();

  // DRAW RUNWAY
  pushMatrix();
  translate(width/2,height,0);
  rotateX(-PI/2);
  stroke(0);

  // RUNWAY EDGE
  rect(-100,0,200,10000);

  // RUNWAY GUIDE LINES
  for (int k=0;k<10000;k+=50){
    if(k%100==0)
      rect(-1,k,2,50);
  }
  popMatrix();

  // MOVE PLANE FORWARD
  z-=velocity;
}
```

For each of (i), (ii) and (iii) below, all changes needed can be made before the `endCamera()` line. Thus in your answers rewrite the given code, with your changes, only from the start up to `endCamera()`.

(i) Modify the given code to make the plane *roll* in response to a suitable mouse command. [6]

(ii) Modify the given code, with your changes in (i), to make the plane pitch in response to a suitable mouse command. [4]

(iii) Modify the given code, with your changes in (ii), to show a red 'X' cross in the fixed ahead direction (centre of screen) for the pilot. [4]

(iv) For the code above, including your added cross, draw an outline sketch of the scene at the beginning of execution of the program. [5]

Question 6—notes

(a) Sets the position of the camera through setting the eye position, the centre of the scene, and which axis is facing upward. Moving the eye position and the direction it is pointing (the centre of the scene) allows the images to be seen from different angles.

The version without any parameters sets the camera to the default position, pointing to the centre of the display window with the Y axis as up. The default values are camera(width/2.0, height/2.0, (height/2.0) / tan(PI*60.0 / 360.0), width/2.0, height/2.0, 0, 0, 1, 0).

Syntax:

```
camera()
camera(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)
```

Parameters

eyeX float: x coordinate for the eye
eyeY float: y coordinate for the eye
eyeZ float: z coordinate for the eye
centerX float: x coordinate for the centre of the scene
centerY float: y coordinate for the centre of the scene
centerZ float: z coordinate for the centre of the scene
upX float: usually 0.0, 1.0, or -1.0
upY float: usually 0.0, 1.0, or -1.0
upZ float: usually 0.0, 1.0, or -1.0 [6]

(b) (i)

```
float velocity=2;
float z=1000;
float roll=0; // initial roll -- added line

void setup(){
  size(512,512,P3D);
  stroke(0);
}

void draw(){
  background(255);
  noFill();
  beginCamera();
  camera(0,0,0,0,0,-1000,0,1,0);
  translate(width/2,height-200,z);
```

```
  // roll (Z) control by mouse -- added line
  if (mousePressed) {                              // added line
    roll=roll + 0.1 * (PI*mouseX/width - PI/2);  // added line
    rotateZ(roll);                                 // added line
  }                                                // added line
  endCamera();
```

(ii)

```
float velocity=2;
float z=1000;
float roll=0;  // initial roll
float pitch=0; // initial pitch -- added line
void setup(){
size(512,512,OPENGL);
stroke(0);
}

void draw(){
  background(255);
  noFill();
  beginCamera();
  camera(0,0,0,0,0,-1000,0,1,0);
  translate(width/2,height-200,z);
  // roll (Z) and pitch (X) control by mouse
  if (mousePressed) {
    roll=roll + 0.1 * (PI*mouseX/width - PI/2);
    rotateZ(roll);
    pitch=pitch + 0.01*(PI/2 - PI*float(mouseY)/height); // added line
    rotateX(pitch);                                       // added line
  }
  endCamera();
```

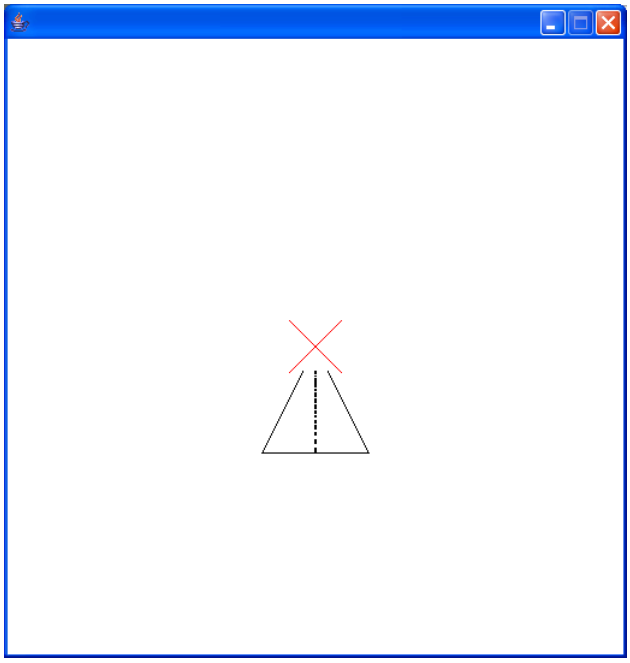
(iii)

```
float velocity=2;
float z=1000;
float roll=0;  // initial roll
float pitch=0; // initial pitch
void setup(){
size(512,512,P3D);
stroke(0);
}

void draw(){
  background(255);
  noFill();
  beginCamera();
  camera(0,0,0,0,0,-1000,0,1,0);
  //draw red cross ahead of pilot (neg z) for heading  // added line
  stroke (255,0,0);                                    // added line
  line(-5,-5,-100,5,5,-100);                           // added line
  line(-5,5,-100,5,-5,-100);                           // added line
  translate(width/2,height-200,z);
  // roll (Z) and pitch (X) control by mouse
  if (mousePressed) {
    roll=roll + 0.1 * (PI*mouseX/width - PI/2);
    rotateZ(roll);
    pitch=pitch + 0.01*(PI/2 - PI*float(mouseY)/height);
    rotateX(pitch);
```

```
    }
    endCamera();
```

(iv) The scene should look as follows:



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 University of London.
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, Stewart House, 32 Russell Square, London WC1B 5DN, UK.