序

這是一本關於 p5.js 的書！從狹義的範圍來說，p5.js 能在網頁進行繪圖、創造互動，能透過視覺化來學習程式設計；廣義而言，p5.js 是創作者表達想法的一種工具，可以從事創意寫碼（creative coding）、生成藝術（generative art）的利器。

本書會談到如何使用 p5.js，然而是基於想創作的主題，擷取 p5.js 裡適當的元素來使用，畢竟使用 p5.js 目的在於創作，而不是想迷失在瑣碎的功能細節裡。

這是一本關於演算法的書！雖然透過 p5.js，任意地寫點程式草稿，弄點隨機、摻些顏色、簡單互動，就能完成乍看效果不錯的塗鴨，然而若沒有進一步的靈感來源，多數人就是失去了興致！

本書會談到一些演算法，會談到一個創作主題，該怎麼分解為數個子任務，讓子任務職責清晰、易於在書裡逐一表達是最主要的目標，因此有些演算法會略為捨棄效能，認識這些演算法，目的是獲取知識，因為知識是靈感的來源，在使用社群實現的相關程式庫時，也能因為掌握相對應的概念而更加得心應手。

這是一本有關數學的書！知識是靈感的來源，而知識來源之一是數學，或許你過去的經驗裡，數學索然無味也無用武之處，然而演算創作的世界，可是數學的伸展台。

本書會談到三角函式、向量、一點點的矩陣以及曲線，這些數學是人類試圖描述自然規律下誕生的產物，因此本書會試著從它們描述了哪些規律來說明，如此一來，才能在掌握了這些數學後，去描述你的創作。

這是一本有關創作的書！無論是 p5.js、演算法或是數學，都是用來描述創作者內心想法的工具、流程或形式，本書藉由一系列的主題，逐步地示範這些知識如何組合，讓心中的想法成形，然而本書只是個開端。

在本書之後，可以看看其他人創作了什麼，不要單純地看著作品，期待靈光一現，可以試著探索別人的作品，從中獲得更多的知識，知道這些知識的應用或組合方式，這個過程等同於探索、累積知識、構造經驗的過程，而這會是從事演算創作時最美妙的部分！

規律與隨機

2

CHAPTER

學習目標

- 運用規律設計圖樣
- 透過觀察尋找規律
- 迭代或遞迴實作規律
- 在規律中穿插隨機

2.1 構築規律

當你手邊有了程式碼作為工具，創作的題材從何而來呢？程式設計本質上，就是觀察人們進行運算的過程，找出規律，使用程式碼加以描述，如果這份規律與圖像有關聯，那就會是創作題材的來源。

2.1.1 魔幻方塊

想建立規律與圖像的關聯，首先要思考的是，**用來建立規律的基本圖像**是什麼，這類基本圖像往往並不複雜，例如，具有互補色的兩個方框：

```
function twoSquare(x, y, width, r, g = r, b = r) {
  push();

  noFill();
  // 外框
  translate(x + width / 5, y + width / 5);
  stroke(r, g, b);
  square(0, 0, width * 4 / 5);

  // 內框
  // 設為互補色
  stroke(255 - r, 255 - g, 255 - b);
  translate(width / 5, width / 5);
  square(0, 0, width * 2 / 5);
```

```
  pop();
}
```

這個 `twoSquare` 函式可以指定位置與 RGB 值，指定 RGB 時如果只指定 `r` 參數，那麼就以 `r` 作為 `g` 與 `b` 的值，也就是灰階了，例如，若使用 `twoSquare(0, 0, 200, 255)`，會畫出以下的黑白方框：

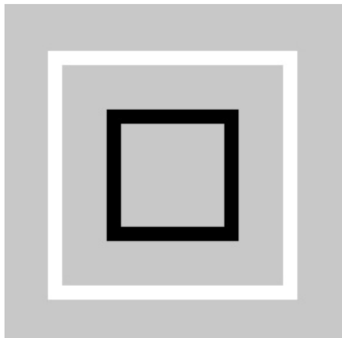

圖 2.1 簡單的黑白方框

接下來**套用某種規律**，例如方塊由上而下、由左而右排列呢？

```
for(let y = 0; y < n; y++) {
  for(let x = 0; x < n; x++) {
    twoSquare(x * w, y * w, w, 0);
  }
}
```

這個程式片段中，`n` 每邊的方塊數，`w` 是方塊的邊長，這會呈現出以下的結果：

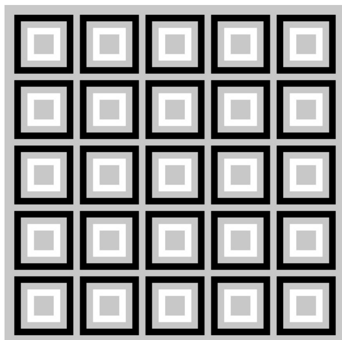

圖 2.2 排列黑白方框

在規律上再加上有規律的變化會如何呢？例如，如果 x+y 是偶數，外框設為灰階度 0，若是奇數就設為 255 呢？

magic-squares phq8wG6n4.js

```
const n = 5;   // 每邊的方塊數
const w = 50;  // 方塊邊長

function setup() {
  createCanvas(n * w + w / 5, n * w + w / 5);
  strokeWeight(5);
  frameRate(2);
}

let flag = 1;
function draw() {
  background(200);;
  for(let y = 0; y < n; y++) {
    for(let x = 0; x < n; x++) {
      const g = (x + y + flag) % 2 === 0 ? 0 : 255; // 奇偶變換
      twoSquare(x * w, y * w, w, g);
    }
  }
  flag = ~flag;
}

function twoSquare(x, y, width, r, g = r, b = r) {
  push();

  noFill();
  // 外框
  translate(x + width / 5, y + width / 5);
  stroke(r, g, b);
  square(0, 0, width * 4 / 5);

  // 內框
  // 設為互補色
  stroke(255 - r, 255 - g, 255 - b);
  translate(width / 5, width / 5);
  square(0, 0, width * 2 / 5);

  pop();
}
```

在這個範例中，額外又加上一個 flag 變數，令每次影格繪製時 flag 的值會是 0 與 1 交相變換，從而達到奇偶數判斷變換的效果，如此一來畫面就會不

斷地變動，看來就有魔幻方塊的風格，下圖是將交相變換的兩個影格放在一起的示意：

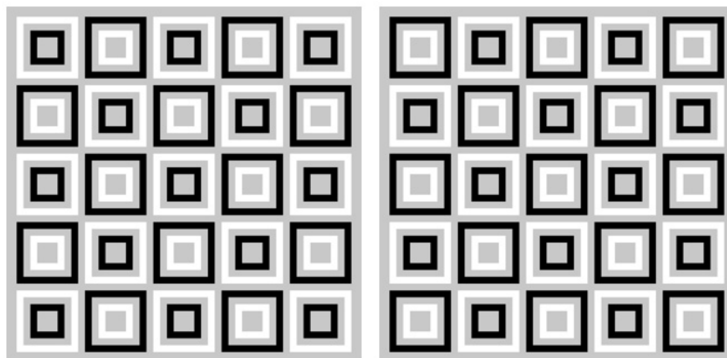

圖 2.3 交換變換的兩個影格

看起來會魔幻，是因為大腦產生錯覺，以為方塊會錯落歪斜，其實每個方塊都是端正地擺放；如果想要更魔幻，可以用 0 到 255 的隨機值作為 `twoSquare` 的 `r`、`g`、`b` 參數值，你可以自行嘗試，看看會有什麼效果！

2.1.2 線的交織

用來建立規律的基本圖像往往並不複雜，就算是直線也可以，規律一開始也不用複雜，單純地從由上而下畫水平線也可以：

圖 2.4 單純由上而下繪製

由左而右畫出垂直線也可以：

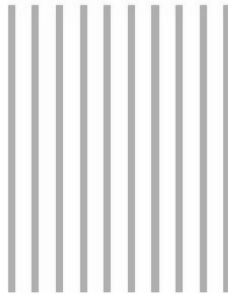

圖 2.5　單純由左而右繪製

設計一個簡單的圖案，然後以一定的規律排列，本身就能構成美麗的圖樣，也許你覺得單純地直線與橫線不夠美？那就將兩種規律疊加在一起吧！

lines iY2iTEh93.js

```
const weight = 10;
const halfWeight = weight / 2;

function setup() {
  createCanvas(400, 400);
  noLoop();
  strokeWeight(weight);
}

function draw() {
  background(200);
  for (let i = 0; i < height; i += 20) {
    // 畫水平線
    stroke(255, 0, 0);
    line(0, i + halfWeight, width, i + halfWeight);

    // 畫垂直線
    stroke(0, 255, 0);
    line(i + halfWeight, 0, i + halfWeight, height);
  }
}
```

跟「點」一樣，「線」在數學上並沒有寬度，不過繪圖上會用筆刷概念，透過筆刷大小來設定畫出的線寬，`line` 函式的四個參數分別是起點的 x、y 與終點的 x、y，範例完成的效果會呈現編織般的美感：

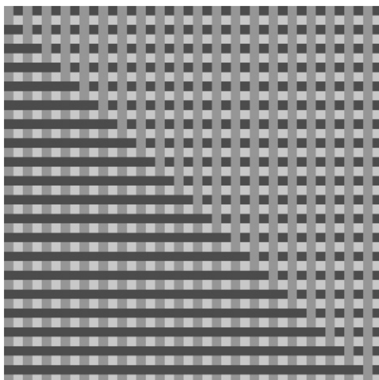

圖 2.6　交織的線

基本的規律構成基本的圖樣，規律與規律之間的組合，會構成新的圖樣，有本《Pattern Design 圖解圖樣設計》就專門討論了這類設計，有興趣可以研究一下。

你也許看過許多 p5.js 的精采作品，出發點其實都是簡單的圖樣，**逐步加入規律**，讓圖樣逐步變化，更多的情況下是嘗試，在不經意間發現可呈現美的作品。

例如，在方才編織般美感的範例上，進一步加上動畫變化如何？這並不難，只要改一下迴圈的邊界就可以了：

lines2 jp8bFZaYQ.js

```
const weight = 10;
const halfWeight = weight / 2;

function setup() {
  createCanvas(400, 400);
  strokeWeight(weight);
  frameRate(24);
}

let to = 0;
function draw() {
  background(200);
  to = (to + 20) % height; // 迴圈邊界
```

```
  for (let i = 0; i <= to; i += 20) {
    stroke(255, 0, 0);
    line(0, i + halfWeight, width, i + halfWeight);

    stroke(0, 255, 0);
    line(i + halfWeight, 0, i + halfWeight, height);
  }
}
```

改變迴圈邊界是程式上的說法，從另一個觀點來說，是改變在畫布上繪圖的範圍，一開始是畫 20 x 20，接著是 40 x 40，再來是 60 x 60、80 x 80… 把一連串的畫框接連播放，就會有漂亮的動畫了，下圖是擷取動畫過程中的兩個影格作為示意：

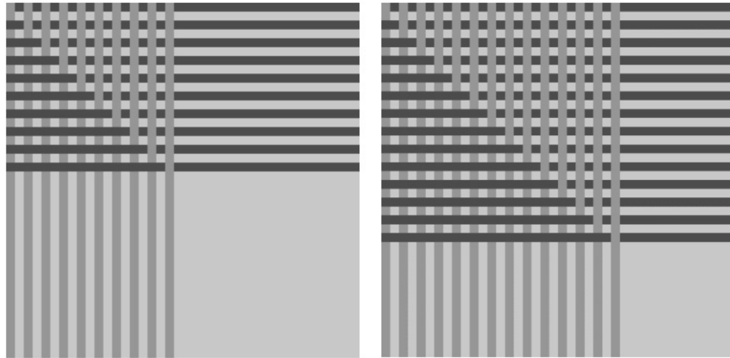

圖 2.7　變動而交織的線

隨意做些變化，還可能還會有驚喜，例如，若不清除前一次的繪圖結果會如何呢？試著將範例中的 background 該行註解掉看看吧！

2.1.3　謝爾賓斯基三角形

剛開始試著使用程式碼作為創作工具時，不用想太多，只需要多嘗試一些規律的組合與變化，有些你心中可能已經知道一些方式，例如先前試過的奇偶變換，有時需要一些運氣，有時需要一些**觀察**，從中發現一些意外的驚喜，例如，試著在底下這個程式片段「?」處可以隨便放上一些運算子，將運算結果用文字畫出來之類的：

```
const n = 32;
const w = 10;
for(let y = 0; y < n; y++) {
```

圖片處理

5
CHAPTER

學習目標

- 圖片拼接／裁剪
- 製作圖片動畫
- 地圖建立與互動
- 像素存取與應用

5.1 拼接／裁剪

除了自行撰寫程式進行繪圖，運用圖片檔案也是個創作的方式，p5.js 可以載入圖片，透過程式碼的控制圖片的顯示、裁剪、拼接，甚至是與圖片進行互動。

5.1.1 圖片載入／顯示

p5.js 可以透過 **`loadImage`** 函式載入圖片，然而要怎麼載入圖片呢？由於透過程式碼載入圖片等資源，會伴隨著安全性的隱憂，畢竟有心人士若惡意地載入有害的資源，那就糟糕了，為了避免安全方面的問題，瀏覽器對程式碼載入資源這件事，做了一些限制。

本書不假設你瞭解瀏覽器的安全限制等問題（畢竟那很複雜），因此你的 p5.js 程式碼從哪個網站取得，圖片就放在該網站，`loadImage` 函式基本上就能載入了。

由於本書使用 p5.js 官方的 Web 編輯器，就將圖片上傳至 Web 編輯器吧！這並不麻煩，只要你註冊、登入為使用者後，可以執行選單的「草稿／新增資

料夾」，接著建立名稱為「images」的資料夾，然後按下編輯器的 > 圖示，在「images」資料夾按右鍵，就可以上傳檔案：

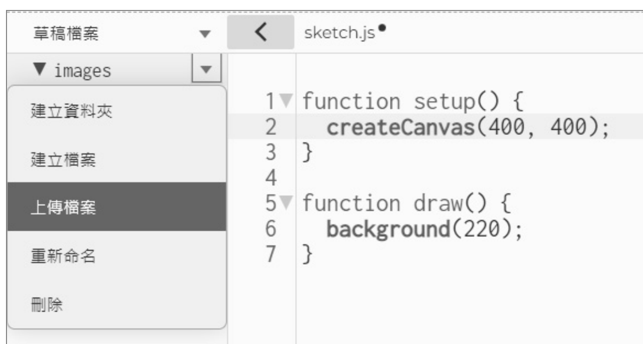

圖 5.1 p5.js 官方編輯器支援檔案上傳

如果想使用 `loadImage` 函式載入圖片，**必須在 `preload` 函式中呼叫 `loadImage`**，`preload` 函式可用來確保圖片都載入了，才呼叫 `setup` 函式。

提示》》》 p5.js 還提供了 `loadJSON`、`loadFont`、`loadStrings` 等下載相對應類型資源的函式，這些函式必須在 `preload` 函式裡呼叫；相對地，其他函式別寫在 `preload`。

`loadImage` 函式執行會傳回 **`p5.Image`** 實例，將之傳給 **`image`** 函式，指定圖片 `x`、`y` 位置，就可以顯示圖片，你可以運用圖片作為媒材，展現更豐富的畫面效果。

例如，2.2.2 曾經談過 Truchet 拼接，當時只是單純地畫圓弧作為拼接塊，如果你有以下的水管圖片作為拼接塊：

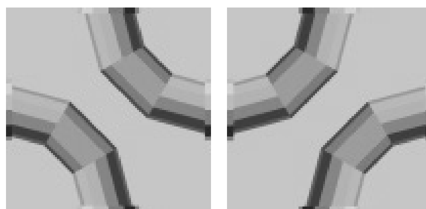

圖 5.2 使用圖片作為拼接塊

5.1.3 圖片動畫

image 函式可以裁剪圖片，在 image 函式官方文件[2]裡有底下這張圖，說明了 image(img,dx,dy,dw,dh,sx,sy,[sw],[sh])版本各參數之作用：

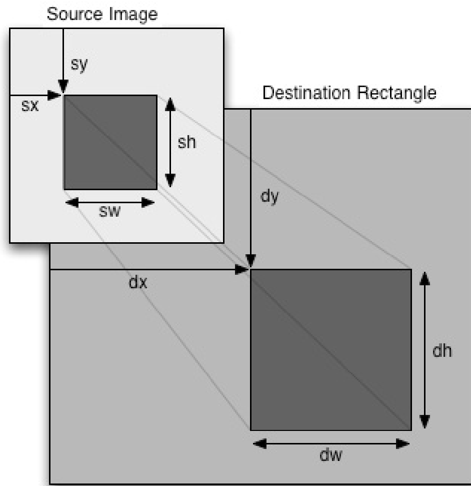

圖 5.9 image 函式參數對照

也就是說，可以指定從來源圖片的(sx,sy)位置開始，選擇寬 sw 高 sh 的部分，在畫布的(dx,dy)位置，以寬 dw 高 dh 繪製出來；如果不指定 sw、sh，那就是從(sw,sh)至右下角的圖片範圍全部畫出來。

創造遠近感

image(img,dx,dy,dw,dh,sx,sy,[sw],[sh])版本的各參數，可以用來單純地進行圖片裁剪，或者是製作圖片動畫，例如方才談到的無限捲動背景效果，例如，來捲動這些樹如何？

---

[2] image 函式官方文件：p5js.org/reference/#/p5/image

圖 5.10　作為背景的樹

圖片可以不只一張，像是將樹作為近景，以下的山作為遠景：

圖 5.11　作為背景的山

在捲動背景時，讓作為遠景的山步進值較小，作為近景的樹圖片步進值較大，就可以製造出具有遠近感的背景捲動。例如：

travel Zy-3dz6id.js

```
let trees;
let mountains;

function preload() {
  trees = loadImage('images/trees.png');           // 樹景，PNG 支援透明背景
  mountains = loadImage('images/mountains.png');   // 山景，PNG 支援透明背景
}

function setup() {
  createCanvas(1000, 200);
}

let tx = 0;  // 樹景圖片繪製的 x 座標
let mx = 0   // 山景圖片繪製的 x 座標

function draw() {
  background(200);

  image(mountains, mx, 0);                        // 繪製山景
  image(mountains, width + mx, 0, 1000, 0);       // 剪裁左邊、拼到右邊

  image(trees, tx, 0);                            // 繪製樹景
```

```
  image(trees, width + tx, 0, 1000, 0);      // 剪裁左邊、拼到右邊

  mx = (mx - 0.5) % 1000;   // 步進 0.5，山景較慢
  tx = (tx - 4) % 1000;     // 步進 4、樹景較快
}
```

這邊使用了支援透明背景的 PNG 圖片，山景與樹景可以重疊在一起， 樹景的背景不會蓋住山景，執行時的參考圖如下：

圖 5.12　具有遠近感的背景捲動

管理圖片素材

談到圖片與動畫，會讓人聯想到 GIF 檔案，若圖片來源是 GIF，執行 `image` 函式時，就看繪圖時正好播放到哪個 GIF 中哪張圖片 ，以這種方式來展現圖片動畫的效果並不好，畢竟若 `frameRate` 設定值較低時，GIF 裡選中的圖片就容易不連續。

`p5.Image` 提供了 `pause`、`play` 等方法，可以暫停、播放 GIF，不過要留意 GIF 的每一個影格品質，確認沒有破圖之類的問題。

除了利用 GIF，也可以將每張圖片存為獨立的檔案，然後依序載入、播放，例如若有 16 張圖檔，`frameRate` 可以設為 16：

clock BgpAH1vzT.js

```
let clocks = [];

function preload() {
  // 載入 16 張圖檔
  for(let i = 0; i < 16; i++) {
    clocks.push(loadImage('images/clock' + i + '.png'));
  }
}

function setup() {
```

```
  createCanvas(400, 350); // 圖片大小為 400x350
  frameRate(16);          // 每秒 16 張
}

let i = 0;

function draw() {
  background(200);

  // 依序繪製
  image(clocks[i], 0, 0);
  i = (i + 1) % 16;
}
```

這種方式可以確實可以達到圖片動畫的效果，下圖是擷取播放中的三張圖：

圖 5.13　播放中的三張圖

不過就程式而言，需要載入 16 張圖片，也就是需要發出 16 次網路請求，若網路速度不佳，在看到動畫前就會需要較久的載入時間；若不想這麼做，可以將 16 張圖檔合併為一張大圖：

圖 5.14　16 張圖集成大圖

然後藉由控制圖片來源的範圍，每次僅繪製出其中一格：

clock2 FgmwnaBJ6.js

```
let clock;

function preload() {
  // 載入大圖
  clock = loadImage('images/clock.png');
}

function setup() {
  createCanvas(400, 350);
  frameRate(16);
}

let i = 0;
function draw() {
  background(200);

  const sx = i % 4;          // 其中一張圖片的 x 座標
  const sy = floor(i / 4); // 其中一張圖片的 y 座標

  // 在畫布上繪製其中一張圖
  image(
    clock,
    0, 0, width, height,
```

```
    sx * width, sy * height, width, height
  );

  i = (i + 1) % 16;        // 每 16 次重置為 0
}
```

繪製出來的效果，與上一個範例是相同的，然而只需要下載一個圖片檔案，就可以取得全部的圖片內容，這是管理動畫圖片素材的常見方式之一。

5.1.4　平面／斜角地圖

有時候在製作一些小遊戲時，需要簡單的地圖，像是利用簡單的陣列標示可行走路線、障礙物等，再根據標示選擇對應的圖片來繪製地圖。

平面地圖

以底下兩張圖來說：

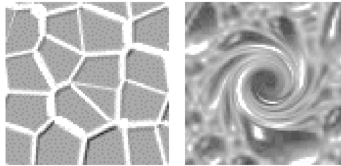

圖 5.15　岩路與岩漿

這兩張圖只要結合以下的程式，就可以繪製簡單的平面地圖：

simple-map　DzPglc6JZ.js

```
let road;
let magma;
function preload() {
  road = loadImage("images/road.jpg");    // 岩路圖片
  magma = loadImage("images/magma.jpg");  // 岩漿圖片
}

function setup() {
  createCanvas(480, 480);
  angleMode(DEGREES);
}

// 地圖資料
const ROAD = 0;   // 岩路
```

```
const MAGMA = 1;  // 岩漿
const map = [
  [1, 0, 1, 1, 1, 1],
  [1, 0, 0, 0, 0, 1],
  [1, 1, 1, 1, 0, 1],
  [0, 0, 0, 1, 0, 1],
  [1, 1, 0, 0, 0, 0],
  [1, 1, 0, 1, 1, 1],
];

function draw() {
  for(let yi = 0; yi < map.length; yi++) {
    for(let xi = 0; xi < map[yi].length; xi++) {
      // 根據地圖資料繪製對應的圖片
      const img = map[yi][xi] == ROAD ? road : magma;
      image(
        img,
        xi * img.width, yi * img.height
      );
    }
  }
}
```

這邊手動設置了地圖資料 map，你也可以透過特定演算法來產生，根據以上 map 可以繪製出來的地圖如下：

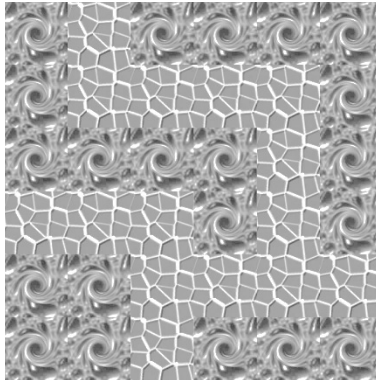

圖 5.16　簡單的平面地圖

斜角地圖

如果想要有些變化，可以嘗試做 45 度視角的斜角地圖：

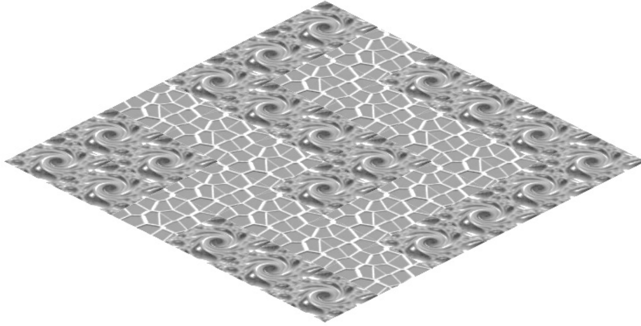

圖 5.17　簡單的斜角地圖

視覺上看來會有 3D 的感覺，然而在圖片處理上，只是將圖片轉 45 度，然後高度縮為一半就能達到，例如，將以上範例的地板圖片改為斜角的版本就會是：

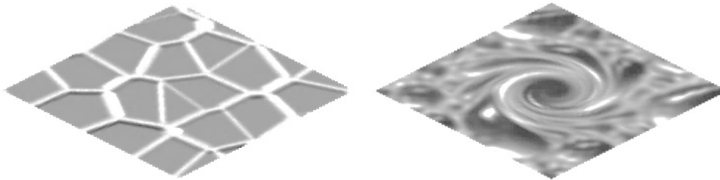

圖 5.18　用於斜角地圖的菱形圖片

如果你只有圖 5.15 的拼接小圖，別急著拿出影像處理軟體，因為透過 `rotate`、**`scale`** 函式，也可以繪製出圖 5.18 的小圖；以下的函式可以接受 `p5.Image` 實例、圖片的中心座標 `x`、`y`，它會將正方形圖片，轉換為斜角地圖可用的菱形圖片：

```
// 平面圖片轉菱形圖片
function diamondTransform(img, x, y) {
  push();
  imageMode(CENTER);  // x、y 對齊圖片中心
  scale(1, 0.5);      // 接下來畫的圖寬縮放為 1、高縮放為 0.5
  rotate(45);         // 接下來畫的圖都旋轉 45 度
  image(img, x, y);
  pop();
}
```

在 1.2.1 時談過，translate、rotate 等函式，會改變座標系統，然而在需要連續進行多個座標系統轉換時，似乎感覺很複雜？其實技巧在於，不要去想著 translate、rotate 等函式整個如何組合，只要想著某個轉換函式與下個繪圖指令間的關係就可以了。

例如，方才談到「在圖片處理上，只是將圖片轉 45 度，然後高度縮為一半」，其實你不用想著接下來是不是繪製圖片，只要想著要將後續的繪圖旋轉 45 度：

```
// 旋轉座標系統
rotate(45);
// 再繪圖
image(img, x, y);
```

你不用想著後面的圖片有沒有被旋轉 45 度，只要想著將後續的繪圖高度縮為一半：

```
// 縮放座標系統
scale(1, 0.5);
// 再繪圖
rotate(45);
image(img, x, y);
```

這就是為什麼方才的 diamondTransform 函式，會構成 scale、rotate 的堆疊流程了。

有了菱形拼接圖片後要怎麼拼接呢？利用轉動、平移公式來計算座標嗎？這太麻煩了，圖片格式必須支援透明背景，若以直角座標來看，菱形拼接圖片的寬為 w、高為 h 的話，對於第一列拼接，只要如下平移就可以了：

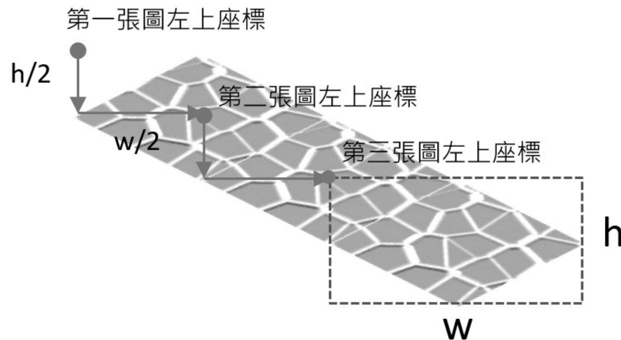

圖 5.19 斜角地圖拼接

類似地，你可以計算第二列、第三列…這是一個方式，不過還有更簡單的方式，**基於向量來計算**。

例如平面地圖繪製時，若拼接圖片寬為 w、高為 h，可以看成每個拼接塊的位置，是基於向量(w,0)與(0,h)在計算：

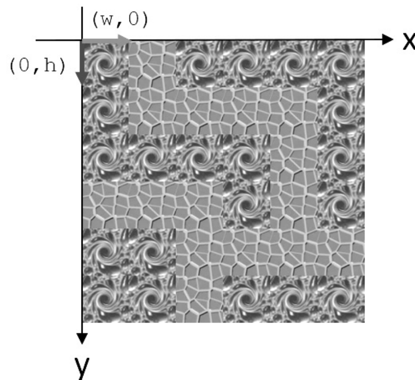

圖 5.20　基於向量思考平面地圖拼接

對於第 x 行、第 y 列的拼接塊，位置若以向量表示，會是 x*(w,0)+y*(0,h)；那麼斜角地圖呢？若拼接圖片寬為 w、高為 h，可以看成是基於(w/2,h/2)與(-w/2,h/2)在計算：

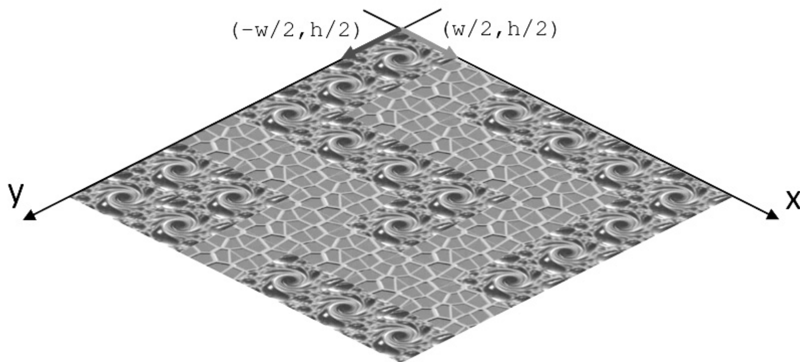

圖 5.21　基於向量思考斜角地圖拼接

這其實是看事情的不同角度，也印證了 3.2.1 談過的，**有些需求若以向量來思考會比較方便**，以上的思考可以透過函式描述如下：

```
// 計算菱形圖片的拼接位置
function coord(x, y, w, h) {
    const basisX = createVector(w / 2, h / 2);  // 基於 x 方向的向量
    const basisY = createVector(-w / 2, h / 2); // 基於 y 方向的向量
    return p5.Vector
             .mult(basisX, x)                   // 有 x 個單位的 basisX 向量
             .add(p5.Vector.mult(basisY, y))    // 加上 y 個單位的 basisY 向量
             .sub(createVector(0, h / 2));      // 菱形最上頂點為原點
}
```

類似地，若以向量來思考直角座標，可以看成位置是基於`(1,0)`、`(0,1)`的向量在計算，因此可以基於 simple-map 範例，結合方才的 `diamondTransform` 與 `coord` 函式，來完成圖 5.17 的斜角地圖拼接：

tiled-map _Qf1t4n3D.js

```
let road;
let magma;
function preload() {
  road = loadImage("images/road.jpg");    // 岩路圖片
  magma = loadImage("images/magma.jpg");  // 岩漿圖片
}

function setup() {
  createCanvas(700, 350);
  angleMode(DEGREES);
}

// 地圖資料
const ROAD = 0;   // 岩路
const MAGMA = 1;  // 岩漿
const map = [
  [1, 0, 1, 1, 1, 1],
  [1, 0, 0, 0, 0, 1],
  [1, 1, 1, 1, 0, 1],
  [0, 0, 0, 1, 0, 1],
  [1, 1, 0, 0, 0, 0],
  [1, 1, 0, 1, 1, 1],
];

function draw() {
  background(255);

  const diamondW = road.width * sqrt(2); // 菱形圖片寬
  const diamondH = diamondW / 2;         // 菱形圖片高
```

```
  translate(width / 2, diamondH / 2); // 拼接塊的原點
  for(let yi = 0; yi < map.length; yi++) {
    for(let xi = 0; xi < map[yi].length; xi++) {
      const {x, y} = coord(xi, yi, diamondW, diamondH);
      const img = map[yi][xi] == ROAD ? road : magma;
      diamondTransform(
        img,
        xi * img.width, yi * img.height
      );
    }
  }
}

// 平面圖片轉菱形圖片
function diamondTransform(img, x, y) {
  push();
  imageMode(CENTER);  // x、y 對齊圖片中心
  scale(1, 0.5);      // 接下來畫的圖寬縮放為 1、高縮放為 0.5
  rotate(45);         // 接下來畫的圖都旋轉 45 度
  image(img, x, y);
  pop();
}

// 計算菱形圖片的拼接位置
function coord(x, y, w, h) {
    const basisX = createVector(w / 2, h / 2);  // 基於 x 方向的向量
    const basisY = createVector(-w / 2, h / 2); // 基於 y 方向的向量
    return p5.Vector
             .mult(basisX, x)                   // 有 x 個單位的 basisX 向量
             .add(p5.Vector.mult(basisY, y))    // 加上 y 個單位的 basisY 向量
             .sub(createVector(0, h / 2));      // 菱形最上頂點為原點
}
```

斜角地圖互動

基於向量來計算，有時可以簡化不少程式的撰寫。例如，若想判定滑鼠在哪個拼接塊按下，對於平面地圖來說非常容易實現，如果是斜角地圖呢？若是基於圖 5.18 來拼接，計算上就會複雜許多，然而基於向量的話，事情就單純多了：

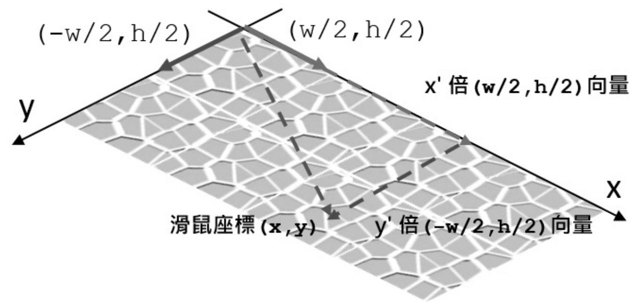

圖 5.22 基於向量思考座標轉換

上圖中，(x,y)與(x',y')間的關係為(x,y)=x'*(w/2,h/2)+y'*(-w/2,h/2)，整理一下並以用矩陣表示的話就是：

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{w}{2} & -\frac{w}{2} \\ \frac{h}{2} & \frac{h}{2} \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix}$$

圖 5.23 斜角地圖座標與滑鼠座標的關係

若已知 x、y，試著求解 x'與 y'，並使用矩陣表示的話就會是：

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \frac{1}{w} & \frac{1}{h} \\ -\frac{1}{w} & \frac{1}{h} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

圖 5.24 滑鼠座標與斜角地圖座標的關係

求得 x'、y'後，想知道滑鼠點選位置，應該算在哪一個拼接塊的話，只要透過 floor(x')、floor(y')就可以了，實作成函式的話會如下：

```
// 指定滑鼠點選座標、拼接塊寬高以及原點
// 傳回向量代表拼接塊索引
function tileVector(mx, my, imgWidth, imgHeight, orgX, orgY) {
  const v = createVector(mx, my).sub(createVector(orgX, orgY));
  return createVector(
    floor(v.x / imgWidth + v.y / imgHeight),
    floor(-v.x / imgWidth + v.y / imgHeight)
  );
}
```

```
function mousePressed() {
  const diamondW = road.width * sqrt(2); // 菱形圖片寬
  const diamondH = diamondW / 2;         // 菱形圖片高

  // 選擇的拼接塊索引
  const { x, y } = tileVector(
    mouseX, mouseY, diamondW, diamondH, width / 2, 0
  );
  selectedX = x;
  selectedY = y;
}
```

可以試著將這段程式碼加入方才的 tiled-map 範例，至於被點選的拼接塊，可以使用 **tint** 函式，指定顏色與透明度等，為接下來的繪製「著色」，例如：

tiled-map2 _Qf1t4n3D.js

```
let road;
let magma;
function preload() {
  road = loadImage("images/road.jpg");   // 岩路圖片
  magma = loadImage("images/magma.jpg"); // 岩漿圖片
}

function setup() {
  createCanvas(700, 350);
  angleMode(DEGREES);
}

// 地圖資料
const ROAD = 0;  // 岩路
const MAGMA = 1; // 岩漿
const map = [
  [1, 0, 1, 1, 1, 1],
  [1, 0, 0, 0, 0, 1],
  [1, 1, 1, 1, 0, 1],
  [0, 0, 0, 1, 0, 1],
  [1, 1, 0, 0, 0, 0],
  [1, 1, 0, 1, 1, 1],
];

function draw() {
  background(255);

  const diamondW = road.width * sqrt(2); // 菱形圖片寬
  const diamondH = diamondW / 2;      // 菱形圖片高

  translate(width / 2, diamondH / 2); // 拼接塊的原點
  for(let yi = 0; yi < map.length; yi++) {
    for(let xi = 0; xi < map[yi].length; xi++) {
```

```
      const { x, y } = coord(xi, yi, diamondW, diamondH);
      const img = map[yi][xi] == ROAD ? road : magma;

      // 點選的拼接塊著色
      if(xi === selectedX && yi === selectedY) {
        tint(255, 128, 255);
      }
      diamondTransform(img, xi * img.width, yi * img.height);
      noTint();
    }
  }
}

// 平面圖片轉菱形圖片
function diamondTransform(img, x, y) {
  push();
  imageMode(CENTER); // x、y 對齊圖片中心
  scale(1, 0.5);     // 接下來畫的圖寬縮放為 1、高縮放為 0.5
  rotate(45);        // 接下來畫的圖都旋轉 45 度
  image(img, x, y);
  pop();
}

// 計算菱形圖片的拼接位置
function coord(x, y, w, h) {
  const basisX = createVector(w / 2, h / 2);  // 基於 x 方向的向量
  const basisY = createVector(-w / 2, h / 2); // 基於 y 方向的向量
  return p5.Vector
          .mult(basisX, x)                // 有 x 個單位的 basisX 向量
          .add(p5.Vector.mult(basisY, y)) // 加上 y 個單位的 basisY 向量
          .sub(createVector(0, h / 2));   // 菱形最上頂點為原點
}

// 指定滑鼠點選座標、拼接塊寬高以及原點
// 傳回向量代表拼接塊索引
function tileVector(mx, my, imgWidth, imgHeight, orgX, orgY) {
  const v = createVector(mx, my).sub(createVector(orgX, orgY));
  return createVector(
    floor(v.x / imgWidth + v.y / imgHeight),
    floor(-v.x / imgWidth + v.y / imgHeight)
  );
}

let selectedX = -1;
let selectedY = -1;

function mousePressed() {
  const diamondW = road.width * sqrt(2); // 菱形圖片寬
  const diamondH = diamondW / 2;      // 菱形圖片高
```

```
  // 選擇的拼接塊索引
  const {x, y} = tileVector(
    mouseX, mouseY, diamondW, diamondH, width / 2, 0
  );
  selectedX = x;
  selectedY = y;
}
```

指定給 tint 的值會除以 255，再與接下來繪製時每個像素的 RGB 值相乘，也就是 tint(255,128,255)的話，會是 R*255/255、G*128/255、B*255/255，得到新的顏色後繪製，結果就是執行後點選地圖任一格，該格就會以不同的顏色顯示：

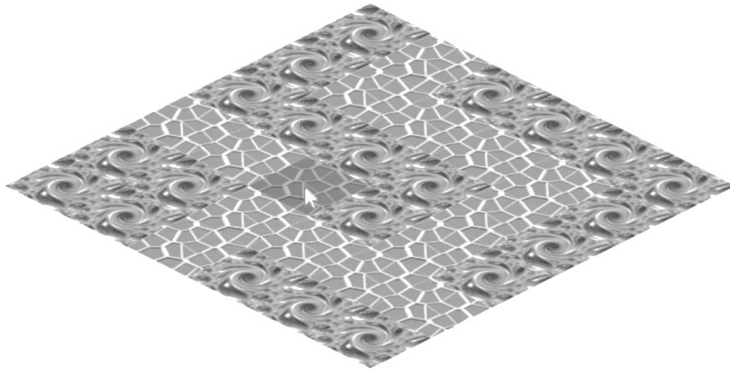

圖 5.25　使用滑鼠點選拼接塊

5.2　像素控制

p5.js 除了可以對圖片進行裁剪／拼接之外，也可以取得圖片像素資訊，基於像素進行計算，除了能從事影像處理軟體相關任務外，還能基於程式設計進行影像轉換等更多樣化的任務。

5.2.1　取得／設定像素

loadImage 函式會傳回 p5.Image 實例，可以透過 p5.Image 實例的 **get**、**set** 方法存取圖片像素資料。

get 方法可以指定圖片的像素座標 x、y，採繪圖座標向右為正、向下為正，get 會傳回陣列，包含了該位置的**[r,g,b,alpha]**資料，代表 **R**、**G**、**B**、**透明度**。

set 方法可以指定圖片的像素座標 x、y 以及像素資料，像素資料可以是數值（指定灰階）、陣列（指定[r,g,b,alpha]）、p5.Color（也可透過 color 函式建立），set 方法操作後必須使用 **updatePixels** 方法，才會真正更新圖片像素。

例如，5.1.4 最後使用過 tint 函式，指定給 tint 的值會除以 255，再與接下來繪圖時每個像素的 RGB 值相乘，達到著色的效果，以下的範例會對指定的圖片，簡單地模仿了這個處理過程：

pixels AIxFsS09o.js

```
let img;

function preload() {
  img = loadImage("images/girl.jpg");
}

function setup() {
  createCanvas(400, 200);
  noLoop();  // 不重複繪製
}

function draw() {
  background(200);

  image(img, 0, 0);             // 顯示原圖
  coloring(img, 0, 150, 150);   // 著色
  image(img, 200, 0);            // 顯示著色後的圖
}

// 對指定的 img 著色
function coloring(img, r, g, b) {
  // 逐一走訪像素
  for(let y = 0; y < img.height; y++) {
    for(let x = 0; x < img.width; x++) {
      const px = img.get(x, y);  // 取得指定位置的像素
      const newPx = [
        px[0] * r / 255,         // R
        px[1] * g / 255,         // G
        px[2] * b / 255,         // B
        px[3]                    // Alpha
      ];
      img.set(x, y, newPx);      // 設定新像素資料
    }
  }
  img.updatePixels();
}
```

範例中的 coloring 函式，運用了 get、set 與 updatePixels 方法，這會對指定的 p5.Image 實例進行像素更新，執行結果中左邊是原圖，右邊是著色後的圖：

圖 5.26　模仿 **tint** 著色

p5.Image 實例有個 **pixels** 特性，預設是空陣列，執行 **loadPixels** 方法後，可以將像素資料載入 pixels 陣列，pixels 是個一維陣列，每四個元素為一組像素資料，也就是 pixels 陣列會是[r1,b1,g1,a1,r2,b2,g2,a2,r3,b3,g3,a3,...]的資料結構，由上而下逐列記錄像素資料。

若是基於效率等理由，也可以自行存取 pixels 特性，例如若 img 是 p5.Image 實例，執行 loadPixels 方法後，以下的程式片段是從 pixels 特性，取得 x、y 位置處像素的方式：

```
const idx = y * 4 * img.width + x * 4; // 乘 4 是因為像素資料每四個元素為一組
const r = img.pixels[idx];
const g = img.pixels[idx + 1];
const b = img.pixels[idx + 2];
const alpha = img.pixels[idx + 3];
```

無論是透過 set 或直接存取 pixels 特性，最後都要呼叫 p5.Image 實例的 updatePixels 方法，目的是更新 p5.Image 實例相對應的內部畫布，方才的範例設定了 noLoop，就是為了避免不斷地執行 coloring 函式，而不斷地更新同一個 p5.Image 實例。

可以透過 **createImage** 函式建立 `p5.Image`，自行設定像素資料，作用之一是可以重複利用圖片處理結果，或者也可以用來複製圖片，例如方才的 pixels 範例，可以改寫如下：

```
let img;

function preload() {
  img = loadImage("images/girl.jpg");
}

function setup() {
  createCanvas(400, 200);
}

function draw() {
  background(200);

  const w = img.width;
  const h = img.height;

  image(img, 0, 0);             // 顯示原圖

  // 建碟新圖片後複製 img
  let copiedImage = createImage(w, h);
  copiedImage.copy(img, 0, 0, w, h, 0, 0, w, h);

  coloring(copiedImage, 0, 150, 150);  // 著色
  image(copiedImage, 200, 0);          // 顯示著色後的圖
}

...coloring 函式不變，故略
```

`p5.Image` 實例的 **copy** 方法，可以指定來源圖片、來源的原點 `x`、`y`、寬、高，以及目的圖片的原點 `x`、`y`、寬、高。

5.2.2 濾鏡實現

既然 p5.js 可以存取像素，那麼就可以取得像素資料，進行各種運算，例如，可以取得 RGB 後，透過灰階公式，例如`(r*38+g*75+b*15)>>7` 將圖片轉為灰階，或者進一步基於灰階值，超過門檻值設為白色，低於某值設為黑色，實現影像二值化，也就是將圖片轉為黑白，通常稱像素轉換操作為濾鏡套用。

其實 `p5.Image` 實例本身就提供了 **`filter`** 方法，可以套用常見的幾種濾鏡，例如，若要轉灰階，只要指定 **`GRAY`** 引數，若要轉黑白，可以指定 **`THRESHOLD`**，預設的門檻值為 0.5，可以自行指定 0 到 1 的門檻值。

例如，若 `img` 為 pixels 範例中的變數，在 `draw` 函式撰寫以下片段的話：

```
img.filter(THRESHOLD, 0.6);
image(img, 0, 0);            // 顯示原圖
```

就可以得到以下的黑白效果：

圖 5.27　圖片轉黑白

若 p5.js 沒有你想要的濾鏡效果，就可以自行基於像素實現，例如，來想個問題，有沒有辦法只使用黑白兩種顏色，就令圖片在視覺上看來有灰階的效果呢？你應該已經看過這類圖片的應用，像是漫畫上的網點，下圖中每個點都是黑色，然而視覺上會像是灰階：

圖 5.28　半色調繪製