第七章

移動

像翻書一樣，你可以透過畫出一個圖像，接著畫出一個稍微不同的圖像，然後再畫出另一個…等等，讓動畫出現在螢幕上。視覺暫留造成了流暢移動的錯覺。當一組類似的圖像出現的速度夠快時，我們的腦袋會將這些圖像轉化為流暢的移動。

範例 7-1：檢視畫面更新率

為了建立平順的移動，Processing 會試著以每秒 60 個畫面（frame）的速度來執行 *draw()* 裡的程式碼。你可以執行下面的程式，檢視印到控制台的值，以確認畫面更新率（frame rate）。

```
void draw() {
  println(frameRate);
}
```

範例 7-2：設定畫面更新率

frameRate() 函式可用於變更程式的執行速度。下面這個例子可用於測試不同版本的 *frameRate()*：

```
void setup() {
  frameRate(30);        // 每秒 30 個畫面
  //frameRate(12);      // 每秒 12 個畫面
```

```
  //frameRate(2);       // 每秒 2 個畫面
  //frameRate(0.5);     // 每 2 秒 1 個畫面
}
void draw() {
  println(frameRate);
}
```

Processing 會試著以每秒 60 個畫面的速度執行程式碼，但如果它花比 1/60 秒還長的時間來執行 *draw()* 函式，那麼畫面更新率將會降低。*frameRate()* 函式只能指定最大的畫面更新率，任何程式實際的畫面更新率取決於執行你的程式碼的電腦。

速度與方向

為了建立流暢的移動，我們使用了一個稱為 *float* 的資料型態。此型態的變數可用於儲存具有小數的數值，於是能夠替移動提供較高的分辨度。舉例來說，當你使用的是 *int* 型態的變數時，最慢可以一次移動一個像素（1, 2, 3, 4, . . .），但當你使用的是 *floats* 型態的變數時，你可以依照你想要的速度來移動像素（1.01, 1.01, 1.02, 1.03, . .）。

範例 7-3：移動一個圖形

下面的例子會透過更新 *x* 變數將一個圖形從左邊移到右邊：

```
int radius = 40;
float x = -radius;
float speed = 0.5;

void setup() {
  size(240, 120);
  smooth();
  ellipseMode(RADIUS);
}
```

```
void draw() {
  background(0);
  x += speed;  // 遞增 x 的值
  arc(x, 60, radius, radius, 0.52, 5.76);
}
```

執行此程式碼，你會在 *x* 變數的值大於視窗的寬度時發現，圖形從螢幕的右邊離開。*x* 的值會繼續增加，但是再也看不到該圖形了。

範例 7-4：繞回

此行為的做法有許多可能性，你可以根據自己的喜好來做選擇。首先我們會示範如何擴充程式碼，讓圖形在螢幕右邊消失之後，把它移回螢幕的左邊。此例中，螢幕被視為一個被扁平化的圓筒，上面的圖形移到螢幕之外時又會回到它的起始點：

```
int radius = 40;
float x = -radius;
float speed = 0.5;

void setup() {
  size(240, 120);
  smooth();
  ellipseMode(RADIUS);
}

void draw() {
  background(0);
  x += speed;                  // 遞增 x 的值
  if (x > width+radius) {      // 若圖形離開螢幕
    x = -radius;               // 則移到左邊
  }
  arc(x, 60, radius, radius, 0.52, 5.76);
}
```

draw() 執行時的每個循環，程式碼會測試 *x* 的值被遞增之後是否會超過螢幕的寬度（加上圖形的半徑）。若超過了，我們會把 *x* 的值設成一個負值，這樣即使它的值繼續遞增，它仍會從左邊進入螢幕。整個運作原理可參考圖 7-1 的說明。

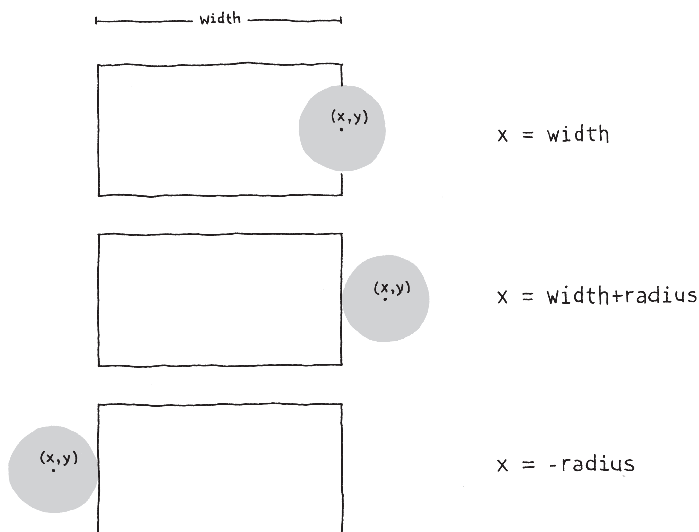


圖 7-1：測試視窗的右邊

範例 7-5：遇到牆壁則反彈

此例中，我們將會擴充範例 7-3，讓圖形在遇到視窗的邊緣時改變移動方向，而不要繞回到左邊。為了做到這一點，我們加入了一個新變數，以便儲存圖形的方向。方向值為 1 時，圖形會向右移動，而方向值為 *–1* 時，圖形會向左移動：

```
int radius = 40;
float x = 110;
float speed = 0.5;
int direction = 1;

void setup() {
  size(240, 120);
  smooth();
  ellipseMode(RADIUS);
}

void draw() {
  background(0);
  x += speed * direction;
  if ((x > width-radius) || (x < radius)) {
    direction = -direction;                    // 翻轉方向
  }
  if (direction == 1) {
    arc(x, 60, radius, radius, 0.52, 5.76);    // 面向右側
  } else {
    arc(x, 60, radius, radius, 3.67, 8.9);     // 面向左側
  }
}
```

當圖形抵達邊緣時，此程式碼會透過變更 *direction* 變數的正負號來翻轉圖形的方向。舉例來說，當圖形抵達邊緣時，若 *direction* 變數為正值，則程式碼會將其翻轉為負值。

兩個位置之間

有時你會想要讓一個圖形從螢幕上的一個點移動到另一個點。只要幾列程式碼，你就可以設置起點位置和終點位置。然後在每個畫面之上計算這兩個位置之間（tween）的移動。

範例 7-6：計算兩個位置之間的移動

為了讓範例程式模組化，我們在程式碼開頭建立了一組變數。試著多執行此程式碼幾次，變更變數的值，看看此程式碼如何以一定範圍的速度，將一個圖形從一個位置移動到另一個位置。變更 *step* 變數的值可以改變速度：

```
int startX = 20;        // 最初的 x 座標
int stopX = 160;        // 最終的 x 座標
int startY = 30;        // 最初的 y 座標
int stopY = 80;         // 最終的 y 座標
float x = startX;       // 當前的 x 座標
float y = startY;       // 當前的 y 座標
float step = 0.005;     // 設定每一步的大小（0.0 到 1.0）
float pct = 0.0;        // 已完成的百分比（0.0 到 1.0）

void setup() {
  size(240, 120);
  smooth();
}

void draw() {
  background(0);
  if (pct < 1.0) {
    x = startX + ((stopX-startX) * pct);
    y = startY + ((stopY-starty) * pct);
    pct += step;
  }
  ellipse(x, y, 20, 20);
}
```

隨機

線性移動常見於電腦圖形，而實體世界的移動則通常有其特殊之處。例如，葉子飄落到地上，或是螞蟻爬過崎嶇的地形。我們可以透過產生隨機數來模擬這個世界不可預期的性質。*random()* 函式可用於計算這些值；在程式中我們可以設置一個範圍來調整這些混亂的值。

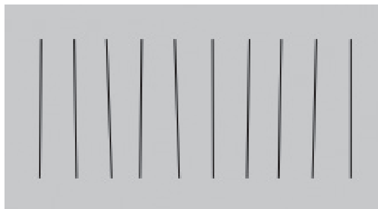
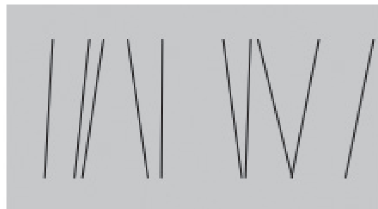
範例 7-7：產生隨機值

下面的例子會將隨機值印到控制台，它的範圍受限於滑鼠游標的位置。*random()* 函式總是會傳回一個浮點值，所以賦值運算符（=）左邊的變數必須是一個 *float*，正如我們在此處所看到的那樣：

```
void draw() {
  float r = random(0, mouseX);
  println(r);
}
```

範例 7-8：隨機作畫

下面的例子建構在範例 7-7 之上；它會使用來自 *random()* 的值來變更螢幕上各線段的位置。當滑鼠游標位於螢幕左邊時，變動比較小；當滑鼠游標移到螢幕右邊時，來自 *random()* 的值就會增加，而使得移動加劇。因為 *random()* 函式被擺在 *for* 迴圈裡，所以會針對每條線段的每個點計算這個新的隨機值：

```
void setup() {
  size(240, 120);
  smooth();
}
```

```
void draw() {
  background(204);
  for (int x = 20; x < width; x += 20) {
    float mx = mouseX / 10;
    float offsetA = random(-mx, mx);
    float offsetB = random(-mx, mx);
    line(x + offsetA, 20, x - offsetB, 100);
  }
}
```

範例 7-9：隨機移動圖形

當我們在螢幕上移動圖形時，隨機值可以產生外觀上較為自然的圖像。在下面的例子中，*draw()* 的每個循環會透過隨機值來修正圓圈的位置。因為沒有用到 *background()* 函式，所以會留下過去的位置：

```
float speed = 2.5;
int diameter = 20;
float x;
float y;

void setup() {
  size(240, 120);
  smooth();
  x = width/2;
  y = height/2;
}

void draw() {
  x += random(-speed, speed);
  y += random(-speed, speed);
  ellipse(x, y, diameter, diameter);
}
```

若此範例程式執行得夠久，你可能會看到圓圈離開視窗，然後又在視窗中重現。這得靠運氣，但是我們可以加入幾個 *if* 結構，或是使用 *constrain()* 函式，以避免圓圈離開螢幕。*constrain()* 函式可將值侷限在特定的範圍，這可用於將 *x* 和 *y* 的值維持在顯示視窗的範圍內。透過把前面程式碼中的 *draw()* 函式代換成下面的 *draw()* 函式，將可讓圓圈維持在螢幕上：

```
void draw() {
  x += random(-speed, speed);
  y += random(-speed, speed);
  x = constrain(x, 0, width);
  y = constrain(y, 0, height);
  ellipse(x, y, diameter, diameter);
}
```

randomSeed() 函式可用於強制 *random()* 在程式每次執行時產生相同的數值序列。相關細節可參考Processing Reference。

計時器

每個 Processing 程式都會計數自從它開始執行到現在經過了多少時間。它會以毫秒（千分之一秒）為計數單位，所以 1 秒之後，計數值為 1,000；5 秒之後，計數值為 5,000；1 分鐘後，計數值為 60,000。我們可以使用這個計數器在特定的時間觸發動畫。*millis()* 函式可用於傳回此計數值。

範例 7-10：經過了多久時間

執行下面的程式，你可以看到經過了多久時間：

```
void draw() {
  int timer = millis();
  println(timer);
}
```

範例 7-11：觸發被計時的事件

透過成對的 *if* 區塊，來自 *millis()* 的值可用於安排動畫和程式中各事件的順序。例如，2 秒後，*if* 區塊裡的程式碼會觸發一個變動。此例中，名為 *time1* 和 *time2* 的變數可用於決定何時變更 *x* 變數的值。

```
int time1 = 2000;
int time2 = 4000;
float x = 0;

void setup() {
  size(480, 120);
  smooth();
}

void draw() {
  int currentTime = millis();
  background(204);
  if (currentTime > time2) {
    x -= 0.5;
  } else if (currentTime > time1) {
    x += 2;
  }
  ellipse(x, 60, 90, 90);
}
```

循環

如果你是個精於三角函數的人，想必你已經知道正弦與餘弦函數是多麼驚人。若你對三角函數不熟，我們希望以下的例子將會引發你的興趣。我們在此處不會探討數學的細節，但是我們會示範幾個能夠產生流暢移動的應用程式。

圖 7-2 展現了正弦波的值與角度的關係。檢視波形的頂部和底部，注意縱軸的變化率：緩慢下降、停止，然後切換方向。這樣的曲線特性可以產生有趣的移動。

對於所指定角度的正弦或餘弦，Processing 中的 *sin()* 和 *cos()* 函式會傳回範圍 –1 到 1 的值，角度必須以弳度值來指定（參見範例 3-7 和 3-8 對弳度所做的說明）。為了用於作畫，*sin()* 和 *cos()* 所傳回的浮點值通常會乘以一個較大的值。

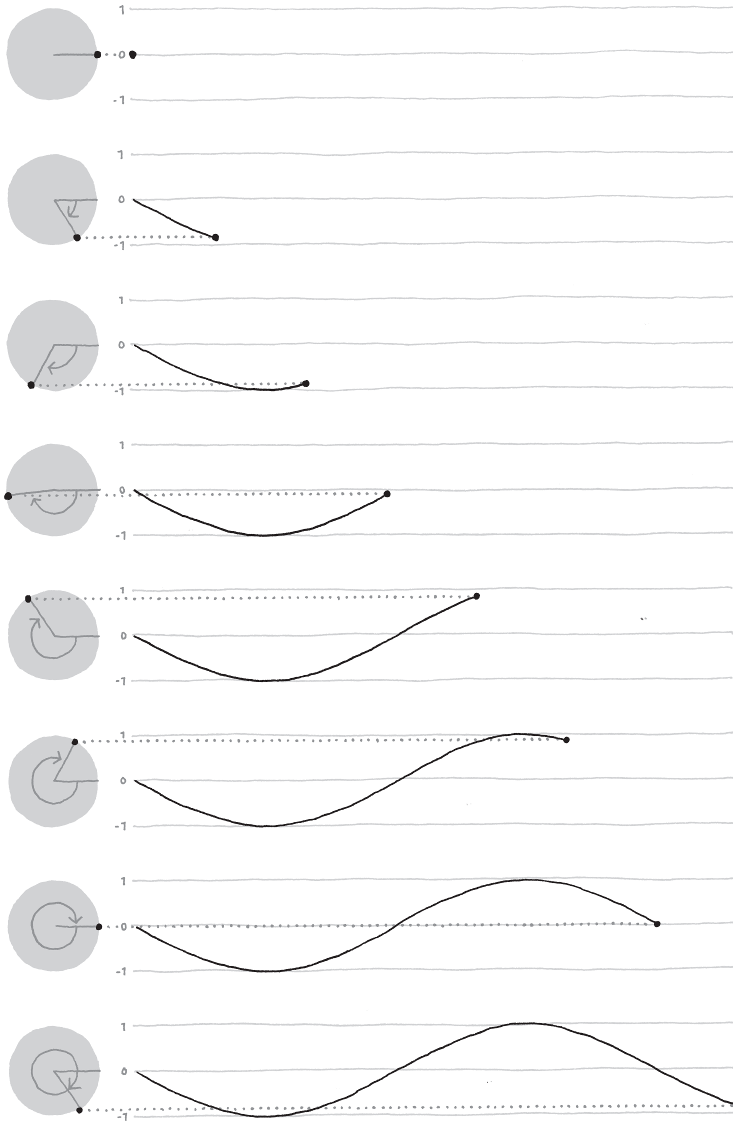


圖 7-2：正弦和餘弦值

範例 7-12：正弦波的值

這個例子展示了當角度增加時 *sin()* 週期的值如何從 –1 變化到 1。使用 *map()* 函式，可以把 *sinval* 變數的值從這個範圍轉換至範圍 0 到 255 的值。這個新產生的值會被拿來設定視窗的背景顏色：

```
float angle = 0.0;

void draw() {
  float sinval = sin(angle);
  println(sinval);
  float gray = map(sinval, -1, 1, 0, 255);
  background(gray);
  angle += 0.1;
}
```

範例 7-13：正弦波的移動

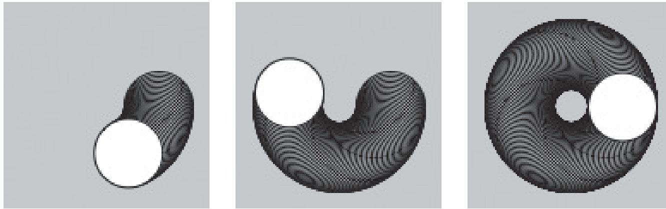
這個例子展示了這些值如何被轉化成移動：

```
float angle = 0.0;
float offset = 60;
float scalar = 40;
float speed = 0.05;

void setup() {
  size(240, 120);
  smooth();
}

void draw() {
  background(0);
  float y1 = offset + sin(angle) * scalar;
  float y2 = offset + sin(angle + 0.4) * scalar;
  float y3 = offset + sin(angle + 0.8) * scalar;
```

```
  ellipse( 80, y1, 40, 40);
  ellipse(120, y2, 40, 40);
  ellipse(160, y3, 40, 40);
  angle += speed;
}
```

範例 7-14：循環移動

當 *sin()* 和 *cos()* 一起使用時，可以產生循環移動（circular motion）。*cos()* 的值提供了 x 座標，而 *sin()* 的值提供了 y 座標。這兩個值都會乘以一個名為 *scalar* 的變數，以變更移動的半徑，並提供一個偏移值，以設置循環移動的中心：



```
float angle = 0.0;
float offset = 60;
float scalar = 30;
float speed = 0.05;

void setup() {
  size(120, 120);
  smooth();
}

void draw() {
  float x = offset + cos(angle) * scalar;
  float y = offset + sin(angle) * scalar;
  ellipse( x, y, 40, 40);
  angle += speed;
}
```

範例 7-15：螺旋

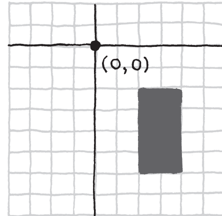
稍微修改一下，在每個畫面遞增 *scalar* 的值，這會產生螺旋形，而非圓形：

```
float angle = 0.0;
float offset = 60;
float scalar = 2;
float speed = 0.05;


void setup() {
  size(120, 120);
  fill(0);
  smooth();
}

void draw() {
  float x = offset + cos(angle) * scalar;
  float y = offset + sin(angle) * scalar;
  ellipse( x, y, 2, 2);
  angle += speed;
  scalar += speed;
}
```

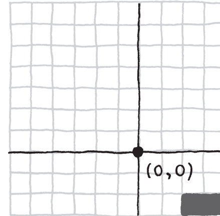
轉譯、旋轉、縮放

變更螢幕座標是建立動畫的另一種技術。例如，你可以將一個 50 像素的圖形右移，或是你可以將位於座標 (0,0) 的 50 像素右移 — 螢幕上所看到的視覺效果並無不同。透過修改預設的座標系統，我們可以建立各種轉換，包括轉譯、旋轉和縮放。參見圖 7-3 所做的說明。

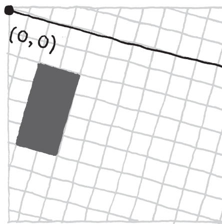
```
translate(40, 20);
rect(20, 20, 20, 40);
```



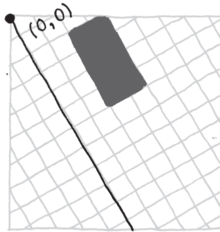

```
translate(60, 70);
rect(20, 20, 20, 40);
```



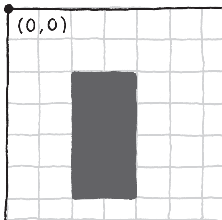

```
rotate(PI/12);
rect(20, 20, 20, 40);
```




```
rotate(-PI/3);
rect(20, 20, 20, 40);
```




```
scale(1.5);
rect(20, 20, 20, 40);
```




```
scale(3);
rect(20, 20, 20, 40);
```

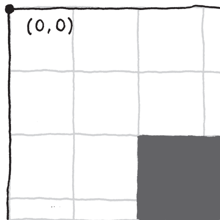
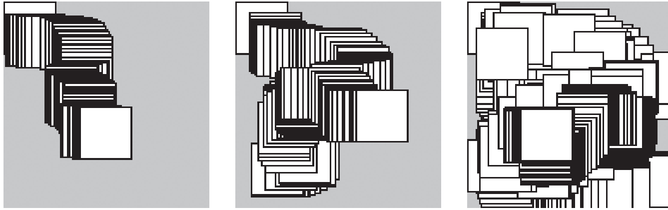


圖 7-3：轉譯、旋轉和縮放座標

轉換的工作相當棘手，但是 *translate()* 函式卻相當簡單，所以讓我們來使用它。此函式有兩個參數，可用於將座標系統左移、右移、上移和下移。

範例 7-16：轉譯位置

此例中，注意座標 `(0,0)` 上所畫出的每個矩形，你可以在螢幕上到處移動它們，因為它們受到了 *translate()* 的影響：



```
void setup() {
  size(120, 120);
}

void draw() {
  translate(mouseX, mouseY);
  rect(0, 0, 30, 30);
}
```

translate() 函式將螢幕的 `(0,0)` 座標設成滑鼠游標的位置。程式碼的下一列，在新的 `(0,0)` 座標所繪製的矩形（*rect()*）其實是畫在滑鼠游標的位置上。

範例 7-17：多重轉譯

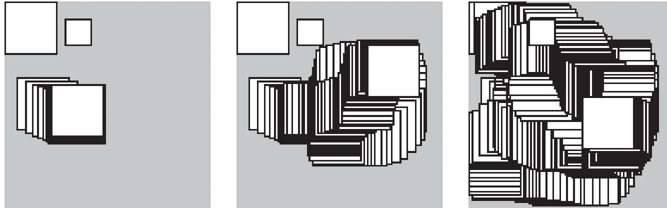
建立一個轉換之後，會影響後續執行的所有繪圖函式。注意當為了控制第二個矩形而加入第二道轉譯命令會發生什麼事：

```
void setup() {
  size(120, 120);
}

void draw() {
  translate(mouseX, mouseY);
  rect(0, 0, 30, 30);
  translate(35, 10);
  rect(0, 0, 15, 15);
}
```

較小的矩形會被轉譯成 *mouseX* + 35 和 *mouseY* + 10。

範例 7-18：隔離轉換

使用 *pushMatrix()* 和 *popMatrix()* 等函式，可以隔離轉換的效果，使其不會影響後續的命令。*pushMatrix()* 執行時會保存當前的座標系統，而 *popMatrix()* 執行後會恢復之前的座標系統：



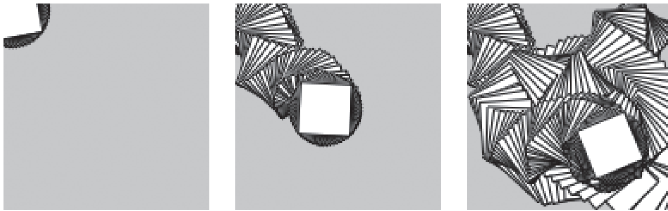
```
void setup() {
  size(120, 120);
}

void draw() {
  pushMatrix();
  translate(mouseX, mouseY);
  rect(0, 0, 30, 30);
  popMatrix();
  translate(35, 10);
  rect(0, 0, 15, 15);
}
```

此例中。較小的矩形總是會被畫在左上角，因為 *translate(mouseX, mouseY)* 的效果已被 *popMatrix()* 隔離。

pushMatrix() 和 *popMatrix()* 函式的使用總是成對的。對於你所使用的每個 *pushMatrix()*，你需要提供一個對應的 *popMatrix()*。

範例 7-19：旋轉

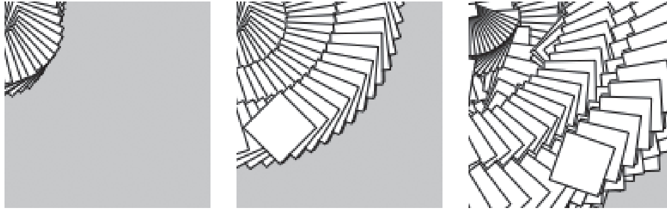
rotate() 函式的功能是旋轉座標系統。它具有一個參數，用於指定所旋轉的角度（單位為弳度）。它的旋轉總是相對於 (0,0) 座標，也就是繞著原點旋轉。要讓一個圖形繞著它的中心點自旋，首先需要使用 *translate()* 將圖形移到你想擺放的位置，接著呼叫 *rotate()*，然後相對於它的中心（位於座標 (0,0)）作畫：



```
float angle = 0.0;

void setup() {
  size(120, 120);
  smooth();
}

void draw() {
  translate(mouseX, mouseY);
  rotate(angle);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

範例 7-20：結合轉換

當 *translate()* 與 *rotate()* 結合時，它們出現的順序會影響到最後的結果。下面的例子如同範例 7-19，但是 *translate()* 與 *rotate()* 的順序顛倒了。圖形現在會繞著顯示視窗的左上角旋轉，而且圖形出現的位置在 *translate()*所設置的角度上：



```
float angle = 0.0;

void setup() {
  size(120, 120);
  smooth();
}

void draw() {
  rotate(angle);
  translate(mouseX, mouseY);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

使用 *rectMode()*、*ellipseMode()*、*imageMode()* 和 *shapeMode()* 等函式也能讓以圖形的中心來繪圖的工作更容易些。

範例 7-21：縮放

scale() 函式可用於伸縮螢幕上的座標。如同 *rotate()*，它的轉換也是相對於原點。因此，如同 *rotate()*，要以圖形的中心點來縮放一個圖形，你需要先轉譯它的位置，接著呼叫縮放功能，然後相對於它的中心點（位於座標 (0,0)）作畫：

```
float angle = 0.0;

void setup() {
  size(120, 120);
  smooth();
}

void draw() {
  translate(mouseX, mouseY);
  scale(sin(angle) + 2);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

範例 7-22：維持筆劃的一致性

你可以從範例 7-21 中線段的粗細看到 *scale()* 函式對筆劃粗細（stroke weight）所造成的影響。為了在縮放圖形時維持筆劃粗細的一致性，我們將想要的筆劃粗細除以縮放值：

```
float angle = 0.0;

void setup() {
  size(120, 120);
  smooth();
}

void draw() {
  translate(mouseX, mouseY);
  float scalar = sin(angle) + 2;
  scale(scalar);
  strokeWeight(1.0 / scalar);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

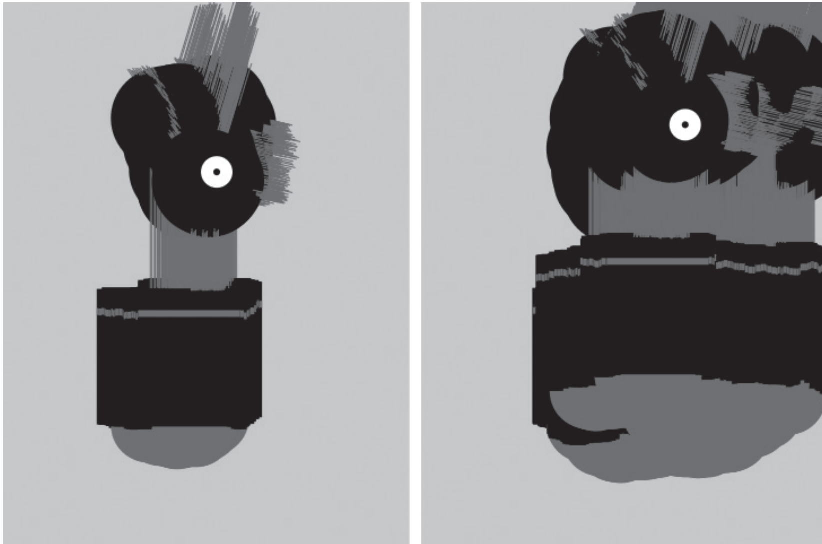
範例 7-23：一個用關節連接的手臂

在最後這個最長的轉換例子中，我們將一系列的 *translate()* 和 *rotate()* 函式放在一起，以便建立一個會來回彎曲之被鏈結起來的手臂。每個 *translate()* 會進一步移動線段的位置，而每個 *rotate()* 會加到前一個旋轉結果，以便讓手臂更彎曲：

```
float angle = 0.0;
float angleDirection = 1;
float speed = 0.005;

void setup() {
  size(120, 120);
  smooth();
}

void draw() {
  background(204);
  translate(20, 25);        // 移動到開始位置
  rotate(angle);
  strokeWeight(12);
  line(0, 0, 40, 0);
  translate(40, 0);         // 移動到下一個關節
  rotate(angle * 2.0);
  strokeWeight(6);
  line(0, 0, 30, 0);
  translate(30, 0);         // 移動到下一個關節
  rotate(angle * 2.5);
  strokeWeight(3);
  line(0, 0, 20, 0);

  angle += speed * angleDirection;
  if ((angle > QUARTER_PI) || (angle < 0)) {
    angleDirection *= -1;
  }
}
```

我們在此處並未用到 *pushMatrix()* 和 *popMatrix()*，因為我們想要傳播轉換的效果 — 對於建構在前一次轉換效果的每一個轉換而言。當 *draw()* 開始繪製每個畫面時，座標系統會被自動重置為預設狀態。

第 5 個機器人範例：移動



此例中，我們對機器人用到隨機和循環移動的技術。拿掉 *background()* 可以讓我們比較容易看到機器人的位置和身體的變化方式。

每個畫面上，x 座標會被加入範圍 –4 到 4 的隨機數值，而 y 座標會被加入範圍 –1 到 1 的隨機數值。這會讓機器人從左到右的移動超過從上到下的移動。由 *sin()* 函式計算而得的數值用於變更脖子的高度，因此會讓它在範圍 50 到 110 的像素之間來回擺動：

```
float x = 180;           // X 座標
float y = 400;           // Y 座標
float bodyHeight = 153;  // 身體的高度
float neckHeight = 56;   // 脖子的高度
float radius = 45;       // 頭部的半徑
float angle = 0.0;       // 移動的角度


void setup() {
  size(360, 480);
  smooth();
  ellipseMode(RADIUS);
  background(204);
}
```

```
void draw() {
  // 透過一個小的隨機數值來變更位置
  x += random(-4, 4);
  y += random(-1, 1);

  // 變更脖子的高度
  neckHeight = 80 + sin(angle) * 30;
  angle += 0.05;

  // 調整頭部的高度
  float ny = y - bodyHeight - neckHeight - radius;

  // 脖子
  stroke(102);
  line(x+2, y-bodyHeight, x+2, ny);
  line(x+12, y-bodyHeight, x+12, ny);
  line(x+22, y-bodyHeight, x+22, ny);

  // 天線
  line(x+12, ny, x-18, ny-43);
  line(x+12, ny, x+42, ny-99);
  line(x+12, ny, x+78, ny+15);

  // 身體
  noStroke();
  fill(102);
  ellipse(x, y-33, 33, 33);
  fill(0);
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);
  fill(102);
  rect(x-45, y-bodyHeight+17, 90, 6);

  // 頭部
  fill(0);
  ellipse(x+12, ny, radius, radius);
  fill(255);
  ellipse(x+24, ny-6, 14, 14);
  fill(0);
  ellipse(x+24, ny-6, 3, 3);
}
```