

## 第5章

# PVector

## 5.1 类

“两只老虎，两只老虎，跑得快，跑得快。一只没有耳朵，一只没有尾巴，真奇怪！真奇怪！”人们在幼儿时期就明白了“类”的概念：众多的例属于一种事物。这个概念如此显而易见，以至于人们往往会忽视它。而在编程的世界里，类（class）是一个必不可少的概念，属于同一类的个体被称为该类的实例（instance）。



“类”是人为规定的结果，未必完全符合客观事实，也可能引发“白马不是马”的悖论。在程序语言的虚拟世界中，人们完全可以根据自己的需要来定义各种各样的类。但我们首先要分清“类”的名称和“实例”的名称，如导入图片时用的代码（见 4.3 节）：

```
PIImage img=loadImage( "hot.jpg" );
```

其中，`PImage` 是类的名称，而 `img` 是该类一个实例的名称。`PImage` 这个橙色的名字是固定的，它是由 Processing 程序预先定义好的类。而 `img` 这个实例的名称是可以随意修改的，它可以叫 `pic`，也可以叫 `tuPian`，等等。如果在同一处创建了某个类的多个实例，通常可以采用数字后缀的方式来区分不同的实例，如：

```
PImage img0=loadImage("hot.jpg");
PImage img1=loadImage("alp.jpg");
PImage img2=loadImage("...");
```



类的好处在于，每个实例可以直接使用这个类的属性（field）和方法（method）。属性表明这个类（的实例）的各种状态，而方法表明这个类（的实例）能干什么。譬如，Processing 的官方说明文档中显示 `PImage` 类有以下三个属性：

- (1) `height`: `int` 型，表示图片的高度（竖直方向上的像素点数量）。
- (2) `width`: `int` 型，表示图片的宽度（水平方向上的像素点数量）。
- (3) `pixels[]`: `color`型的数组，包含图片中每个像素的颜色。

`PImage` 类的方法包括以下几种：

- (1) `updatePixels()`: 用 `pixels[]` 中的数据来更新图片。
- (2) `get(x, y)`: 获取(x, y)处的颜色，返回一个 `color` 值。
- (3) `set(x, y, col)`: 设置(x, y)处的颜色为 `col`，等同于 `pixels[y*width+x]=col`。

(4) `filter()`: 应用滤镜, 如反相 `filter(INVERT)`、半径为 3 的高斯模糊 `filter(BLUR, 3)`, 等等。

(5) `save(filename)`: 把图片保存在本地文件夹内, 如 `save("my.png")`。

(6) `resize(w, h)`: 把图片缩放到宽度为 w, 高度为 h。

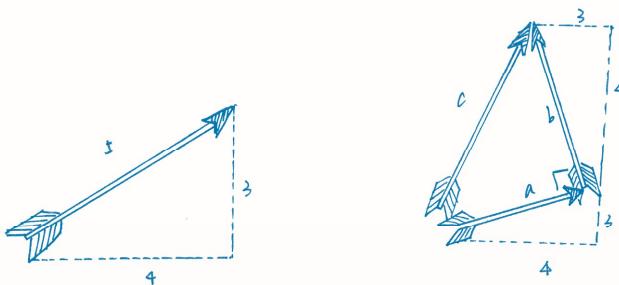
这里面有些方法没有参数, 如 `updatePixels()`; 有些方法需要参数; 而有些方法的参数数量不是唯一的(如 `filter()`可以有一个参数或两个参数)。下面这个程序导入两张图片, 应用滤镜, 再保存图片。

```
void setup(){
    PImage img0=loadImage("hot.jpg");
    PImage img1=loadImage("alp.jpg");
    img0.filter(BLUR, 10);
    img1.filter(INVERT);
    img1.resize(img1.width/2, img1.height/2 );
    img0.save("hot_blur.tif");
    img1.save("alp_inv.png");
}
```

其中有点复杂的一行代码是: `img1.resize(img1.width/2, img1.height/2);`, 因为它先提取了图片 `img1` 的属性 (`width` 和 `height`) 进行数学运算(除以 2), 最后由 `img1` 这个实例调用 `resize()` 方法来实现缩放。

使用类可以给编程带来很多便利, 特别是程序员可以创建自己的类。但我们就暂且只讨论如何使用 Processing 或 Java 既有的类。Processing 自带了一个非常有用的类 `PVector` (Processing Vector 的缩写), 代表二维空间或三维空间中的向量。二维向量代表平面中的箭头。如下图中的箭头长度是 5, 在水平方向的投影长度是 4, 在垂直方向的投影长度是 3。代码如下:

```
PVector a=new PVector(4,3);
float mag=a.mag();
println(mag);
```



程序运行后会在控制台打印出 5.0 这个值。创建实例时需要用 `new` 关键字。  
`new PVector(4,3)` 这部分代码创建了 `Pvector` 的一个实例，这个实例通过 “=” 赋予变量 `a`，这个变量的名字可以自己定，换句话说，我们可以任意命名一个新创建的实例。而 `mag()` 是 `Pvector` 类获取箭头长度的方法。

选中代码中的 `Pvector` 字样右击鼠标，弹出菜单，选择 `find in reference`，就能打开 `Pvector` 的说明文档。`Pvector` 有以下三个属性：

- (1) `x`: `float` 型，箭头在水平方向上的投影长度。
- (2) `y`: `float` 型，箭头在垂直方向上的投影长度。
- (3) `z`: `float` 型，箭头在 `z` 坐标轴上（三维空间）的投影长度。

`Pvector` 的方法非常多，常用的有加法 `add()`、减法 `sub()`、求向量长度 `mag()`、归一化 `normalize()`、线性差值 `lerp()`，等等。用箭头表示向量，我们来了解一下向量的加法。如下图所示，两个箭头 `a` 和 `b` 头尾相接，那么箭头 `c` 等于箭头 `a`+箭头 `b`。这里 `a`、`b` 两个箭头正好垂直，因此代码可以这样写：

```
PVector a=new PVector(4,3);
PVector b=new PVector(-a.y,a.x);
PVector c= PVector.add(a,b);
println(a,b,c);
```

想要让两个长度相同的箭头相互垂直，只要交换它们的 `x`, `y` 值并添加一个负号即可。因此箭头 `b` 有 `(-a.y,a.x)` 或 `(a.y,-a.x)` 两种可能性，前者符合图中的 `b`。在 5.2 节“线性代数”中，我们会用点乘（dot product）的方式来说明相互垂直的关系。

`PVector.add(a,b)`这一行代码把两个箭头相加，但不会改变 a 或 b 的值。最后打印出来的三个向量的数据为：[4.0, 3.0, 0.0] [-3.0, 4.0, 0.0] [1.0, 7.0, 0.0]。另外一种加法的代码为：

```
PVector a=new PVector(4,3);
PVector b=new PVector(-a.y,a.x);
b.add(a);
println(a,b);
```

这里箭头 b 自身加上了箭头 a，箭头 b 最后变成了[1.0, 7.0, 0.0]，与上一种方式里的向量 c 相等。

下面我们用 `PVector` 数组来表示北斗七星的形状，北斗七星一年四季绕着北极星旋转。下面的程序分两步走：

(1) 把北斗七星中的第一颗星（天枢）放在坐标原点(0,0)，用从原点出发的六个箭头来确定其他六颗星的位置，程序如下：

```
PVector[] vs=new PVector[7];
void setup() {
    size(560, 450);
    vs[0]=new PVector();
    vs[1]=new PVector(32, 0);
    vs[2]=new PVector(44, 45);
    vs[3]=new PVector(21, 58);
    vs[4]=new PVector(17, 90);
    vs[5]=new PVector(12, 115);
    vs[6]=new PVector(30, 150);
}
```

(2) 把北斗七星平移到正确的位置（以北极星为中心，即北极星在坐标原点），并使它们绕着北极星旋转，程序如下：

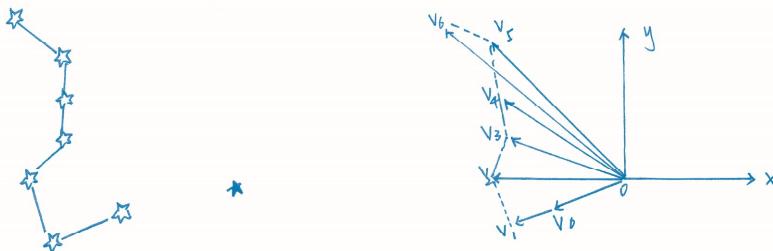
```
void draw() {
```

```

background(0);
fill(255,255,220);
translate(width/2, height/2);
float d= random(4,13);
ellipse(0, 0, d, d);
PVector shift= new PVector(vs[1].x*5, 0);
for (PVector v : vs) {
    PVector p= PVector.add(v, shift);
    p.rotate(-frameCount*0.005);
    d= random(4,13);
    ellipse(p.x, p.y, d, d);
}
}

```

因为天枢星（vs[0]）与北极星之间得距离大约是天枢星与天璇星（vs[1]）之间距离的 5 倍。因此上面的代码中指定了一个平移向量 shift，它的 x 值是 vs[1].x 的 5 倍。然后在 for 循环内部，用向量 shift 分别对七颗星进行平移，再用 rotate 方法实现旋转。



代码 `p.rotate(-frameCount*0.005);` 让实例 `p`（每颗星平移后的位置）调用 `rotate` 方法，`rotate` 方法的参数为弧度（对应旋转的角度），我们让该参数和不断变化的帧数（`frameCount`）关联起来，从而实现匀速旋转。`rotate()`内参数的正负号决定了旋转的方向（顺时针或逆时针）。

在 for 循环的最后，我们用画圆的方式把七颗星显示出来。让圆圈的大小

等于一个实时产生的随机数，产生一闪一闪的效果，（见图 5-1）。

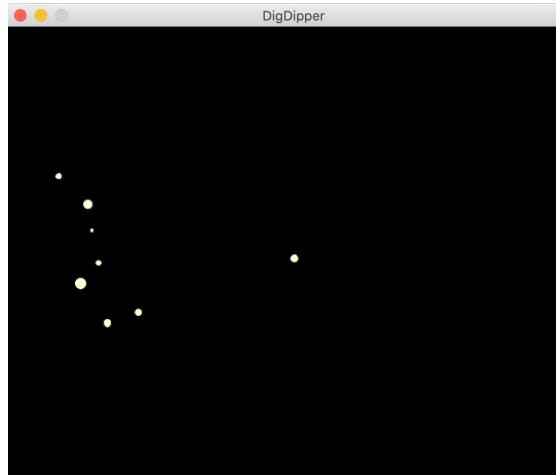
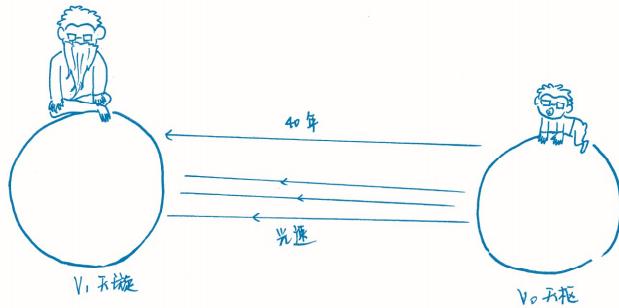


图 5-1 旋转的北斗七星

本节介绍了“类”的使用。除了我们用过的 `PImage` 和 `PVector` 两个类，Processing 自带的类还包括 `ArrayList`（长度可变的数组）、`XML`、`PFont`（字体）、`Table`（csv 表格），等等。本书默认使用 Processing 2 版本的类，而 Processing 3 和 Processing 4 的类有一些微小的区别。



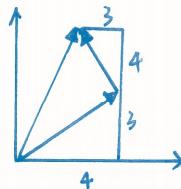
## 5.2 线性代数

接着 `PVector` 类的话题，本节介绍与向量密切相关的线性代数。人们比较

习惯用一个数来描述点的位置，对向量则比较陌生。在二维的平面内，用 x、y 两个数来表示点和用一个向量来表示点，似乎没有什么本质区别。但线性代数促使人“偷懒”。譬如，如果要完整表达两个向量的相加，需要用矩阵格式（下图中的方括号表示法），但把这些矩阵用 a、b、c 符号表示后整个公式就会变得异常简洁。

$$\begin{bmatrix} 1 \\ 7 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} + \begin{bmatrix} -3 \\ 4 \end{bmatrix}$$

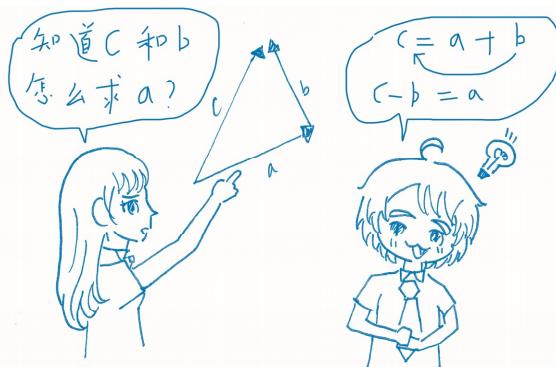
$$c = a + b$$



以上这两种表达方式，在 Processing 中有对应的写法。矩阵格式的运算等价于分别计算 x、y 值：

```
float x = 4 + (-3);
float y = 3 + 4;
```

而公式  $c=a+b$  的代码为 `PVector c=PVector.add(a, b);`。其中，a、b、c 均为



向量。也就是说，一旦点的数据储存在 `PVector` 类的一个实例中，我们就可以直接调用 `add()`、`sub()` 等方法进行各类数学运算了，而不必操心具体的运算过程。

这能使代码变短，减少书写错误。

如果将上图中的问题看作一个几何问题，则需要考虑角度、方位、大小等因素。但如果把这个问题看作一个代数（algebra）问题，那么它就是一个简单的移位变号操作。而且，实现 `a=c-b` 这个运算的代码也非常简单：`PVector a=PVector.sub(c, b);`。注意，加法有以下两种情况：

- (1) `c` 减 `b`，产生一个新的向量 `a`，但 `c` 不变，就要采用 `PVector.sub()` 方法。
- (2) `c` 减去 `b`，就采用 `c.sub(b)`。

古希腊数学家、哲学家毕达哥拉斯（Pythagoras）发现了一个定理：直角三角形斜边的平方=两条直角边的平方和，即我们熟知的勾股定理。对一个二维向量 `v` 来说就是：

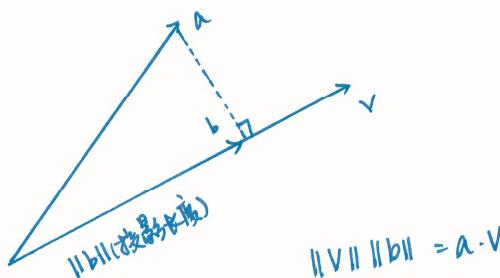
$$\|v\|^2 = v_1^2 + v_2^2$$

这是数学表达式，`\|v\|` 表示向量 `v` 的长度；`v_1` 和 `v_2` 分别表示向量 `v` 在 x 坐标轴和 y 坐标轴上的投影。在上述等式两边加上根号后等式依然成立，再用代码来表示：

```
v.mag() == sqrt(v.x*v.x + v.y*v.y)
```

代码 `v.mag()` 将返回向量的长度。而 `sqrt()` 是开平方操作。

在熟悉了向量的加减法和长度之后，大家或许会想：两个向量可以相乘吗？可以！而且线性代数中有两种乘法：点乘（dot product，数学中用小圆点“•”表示）和叉乘（cross product，数学中用叉号“×”表示）。这里我们只介绍点乘（点积或内积），它主要用来表示一个向量在另一个向量上的垂直投影，如下图所示。

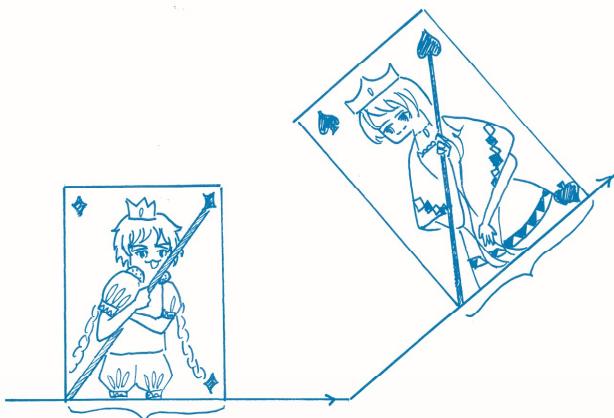


上图中有一个奇妙的公式，它的意思是：向量  $v$  的长度乘以投影的长度 = 向量  $a$  点乘向量  $v$ 。换句话说，投影长度 =  $a$  点乘  $v/\|v\|$ 。因此，求  $a$  在  $v$  上垂直投影的长度的代码为：

```
float dot=a.dot(v);           //点乘
PVector b=v.get();           //把 v 复制一份，赋给 b
b.setMag(dot/v.mag());       //把 b 的长度设置为投影的长度
```

下面这个程序先随机产生一个向量  $v$ ，然后根据鼠标位置来确定向量  $a$ ，再用上面这个方法来求投影向量  $b$ ，最后绘制垂直线。

```
PVector v;
void setup() {
    size(600, 600);
    v=PVector.random2D();
    v.mult(random(100, 300));
}
void draw() {
    background(255);
    translate(width/2, height/2);
    PVector a=new PVector(mouseX-width/2, mouseY-height/2);
    stroke(0);
    line(0, 0, a.x, a.y);
    stroke(200);
    line(0, 0, v.x, v.y);
    float dot=a.dot(v);
    PVector b=v.get();
    b.setMag(dot/v.mag());
    stroke(255, 0, 0);
    line(a.x, a.y, b.x, b.y);
}
```



上述程序的 `draw()` 部分首先采用 `translate(width/2, height/2)` 命令把整个画布的原点移动到屏幕中央，在创建鼠标控制的向量 `a` 时也要进行相应的处理：`new PVector(mouseX-width/2, mouseY-height/2)`。

上述程序的 `setup()` 部分采用 `PVector.random2D()` 方法来创建一个长度为 1 的随机向量（方向是随机的），然后用 `mult()` 方法把这个向量缩放到一个随机长度。现在我们可以总结一下创建向量的方式：

- (1) `PVector a=PVector.random2D(); //方向随机，长度为 1。`
- (2) `PVector a=new PVector(); //数值为 0、0、0。`
- (3) `PVector a=new PVector(x, y); //数值为 x、y、0。`
- (4) `PVector a= v.get(); //复制其他向量的数值。`

下面我们对已有代码进行一个简单扩展，把随机向量 `v` 扩展成多个随机向量（数组 `PVector[] vs`），代码和运行结果如图 5-2 所示。

图 5-2 中的红色线段呈现出一个圆形轮廓，以同一根线段为斜边的所有直角三角形自然形成一个圆（圆的直径即为三角形斜边）。

之前都是用鼠标来确定向量 `a` 的，我们也可以从数组 `PVector[] vs` 中任选一个向量 `vs[i]`，把这个向量投影到其他所有向量上。代码大致如下：

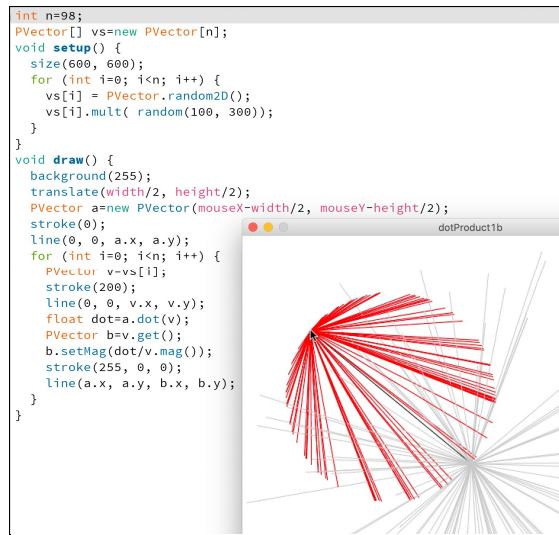


图 5-2 红色垂线构成一个图形

```
PVector a=vs[i]; //从数组中挑选第 i 个向量（如 i=0）
for (int j=0; j<n; j++) { //遍历数组内的每个向量
    PVector v=vs[j];
    //把 a 投影到 v 上
}
```

完整的代码可在\*\*\*dotProduct2b 下载，程序运行的某一时刻如图 5-3 所示。

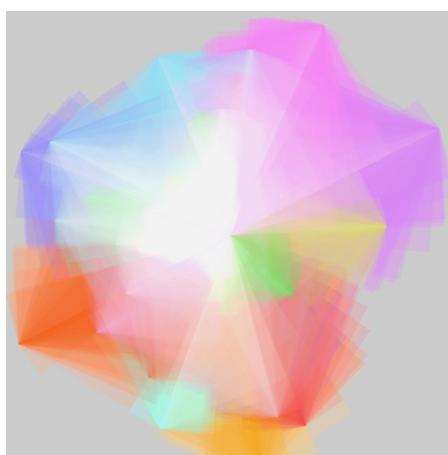


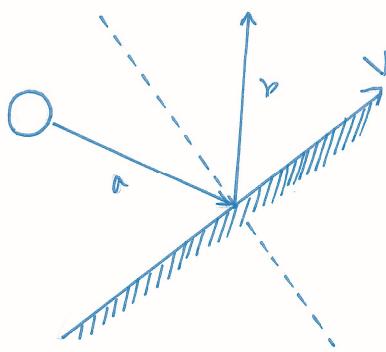
图 5-3 向量投影产生的图形

练习题：

一个乒乓球撞击到墙面后，反弹的方向类似镜面反射（暂时忽略空气阻力、球自转等问题）。如下图所示，乒乓球的初始速度向量是  $\mathbf{a}$ ，反弹后的速度向量是  $\mathbf{b}$ ，墙面的方向用向量  $\mathbf{v}$  来表示。

问题：如何用  $\mathbf{a}$  和  $\mathbf{v}$  来表示  $\mathbf{b}$ ？即把等式  $\mathbf{b}=?$  写完整。

写出表达式后，用代码验证得出的公式是否正确。



### 5.3 力

前面两节介绍了 Processing 中向量的用法，5.2 节的练习题讨论了一种常见的物理现象：乒乓球在任意角度的墙面上反弹的问题。实际上，向量可以帮助我们模拟很多物理现象。3.1 节的弹球程序模拟了一种非常简单的力：小球遇到墙壁就被反弹回来。原来的代码（处理左右两侧的碰撞）是这样的：

```
if (x>w || x<0) {
    dx*=-1;
}
```

其中， $x$  表示小球的  $x$  坐标； $dx$  表示小球在水平方向上的速度。如果用 **PVector** 来表示小球，代码会变为：

```

if (p.x>w || p.x<0) {
    d.x*=-1;
}

```

其中，向量 `p` 表示小球的位置；向量 `d` 表示小球的速度。力的作用被抽象成数学表达式 `d.x*=-1`。按照以上方式，弹球程序的向量版程序写为：

```

PVector p=new PVector(350,400);
PVector d;
int w=700;
int h=800;
void setup() {
    size(w, h); //size(700,800)
    d=PVector.random2D();
    d.mult(10); //速度为 10
}
void draw() {
    fill(0, 10);
    rect(0, 0, w, h);
    fill(255);
    noStroke();
    ellipse(p.x, p.y, 20, 20);
    p.add(d);
    if (p.x>w || p.x<0)
        d.x*=-1;
    if (p.y>h || p.y<0)
        d.y= -d.y;
}

```

在初始化小球的速度向量时，我们先用 `PVector.random2D()` 创建一个方向随机但长度为 1 的向量，再把它的长度设为 10。处理速度时，原始的代码为 `x+=dx; y+=dy;`，而 `PVector` 版本只需要一句 `p.add(d);`。



在 3.3 节中，我们实现了多个弹球的同时运行（见程序\*\*\*），程序开头用了四个数组：

```
float[] x=new float[n];
float[] y=new float[n];
float[] dx=new float[n];
float[] dy=new float[n];
```

如果改用 **PVector** 来表示位置和速度，只需要创建两个数组：

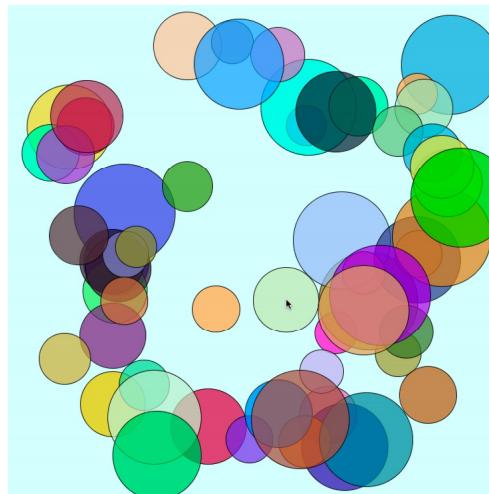
```
PVector[] ps=new PVector[n];
PVector[] ds=new PVector[n];
```

整个弹球程序的 **PVector** 版本可以在 (\*\*\* ) 下载。该程序的运行效果和 3.3 节的一样，但是代码更有条理了。

下面这个例子模拟了一个稍微有点复杂的物理场景，程序运行的结果如图 5-4 所示。

(1) 一推彩色气球在空中随机漂浮，其中一个气球的位置由鼠标控制。

(2) 由鼠标控制的气球撞到其他气球时，其他气球会被推开。



代码可以分为三个部分，首先创建一组圆形，它们的位置、半径、颜色都是随机的，程序如下：

```
int w=800;  
int h=800;  
int n=60;  
  
PVector[] ps= new PVector[n];  
float[] rs=new float[n];  
color[] cs=new color[n];  
  
void setup() {  
    size(w, h);  
    for (int i=0; i< n; i++) {  
        ps[i]= new PVector( random(w), random( h));  
        rs[i]=random(30, 80);  
        cs[i]=color(random(255), random(255), random(255) ,180);  
    }  
}
```

代码的第二部分在 `draw()`方法中实现气球的随机抖动，程序如下：

```

void draw() {
    background(220, 255, 255);
    for (int i=0; i<n; i++) {
        PVector v= ps[i];
        fill(cs[i]);
        ellipse(v.x, v.y, 2*rs[i], 2*rs[i]);
        PVector target= new PVector( random(w), random( h));
        PVector arrow= PVector.sub(target, v);
        arrow.setMag(0.3);
        v.add(arrow);
    }
}

```

其原理是给每个气球设定一个随机的目标位置 target，然后每个气球按照这个目标点移动 0.3 个像素。因为每帧产生的目标位置都是随机的，因此气球表现出随机抖动的感觉。这里我们调用了 PVector 类的 add()、 sub()、 setMag() 等方法。

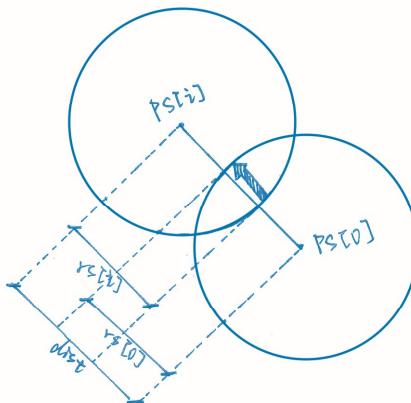
代码的第三部分，是将下列代码放置在 draw() 方法的末尾（完整代码见\*\*\*）。

```

ps[0].set(mouseX, mouseY);
for (int i=1; i<n; i++) {
    float dist= ps[i].dist(ps[0]);
    if (dist < rs[0]+rs[i]) {
        PVector arrow = PVector.sub(ps[i], ps[0]);
        arrow.setMag(rs[0]+rs[i] - dist );
        ps[i].add(arrow);
    }
}

```

ps[0]所在的圆形（位置由鼠标控制）把第 i 个圆形推开的向量分析可参考下图。



以  $ps[i]$  为圆心的圆形将被推开，需要移动的量可以用图中的小箭头表示。这个小箭头的方向与大箭头平行，长度为  $rs[0]+rs[i]-dist$ （两个圆形的半径之和减去两个圆心之间的距离）。另外需要注意的是，这种推开的动作只有在两个圆形相交的前提下才会发生，因此在代码中要用 `if(dist < rs[0]+rs[i])` 语句把与推开相关的代码都包起来。

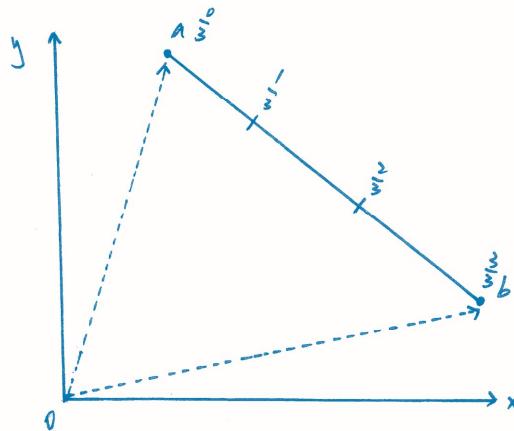
练习题：

基于上述例子，如果两者相交的话，如何让所有小球之间两两相斥。

## 5.4 线性插值

`PVector` 类有一个非常有用的 `lerp()` 方法，它是 linear interpolation 的缩写，即线性插值。譬如，把线段  $ab$  三等分将得到两个等分点（见下图），如果把线段自身的两个端点算进去，则共有四个点。这四个点可以依次表示为：

```
PVector.lerp(a, b, 0/3.0)
PVector.lerp(a, b, 1/3.0)
PVector.lerp(a, b, 2/3.0)
PVector.lerp(a, b, 3/3.0)
```



`lerp()`方法的前两个参数是线段的两个端点，第三个参数是新增点  $v$  所在位置的比例（ $av$  的长度除以  $ab$  的长度）。在线性代数中，线性插值可以写为：

$$v = a + s(b - a)$$

其中， $s$  就是插值点的比例，如  $1/3$ 、 $2/3$  等，等式也可以写为：

$$v = (1-s)a + sb$$

因此 `PVector v = PVector.lerp(a, b, s)` 等价于以下三行代码：

```
a.mult(1-s);
b.mult(s);
PVector v= PVector.add(a,b);
```

1.1 节的第一个程序，可以很快改写为包含插值点的绘图程序，如图 5-5 所示。

2.2 节创建了一个生成心形的动态程序\*\*\*，我们首先可以把它改写成 `PVector` 版本，程序如下：

```
float r=240;
int n=360;
void setup() {
  size(600, 600);
}
```

```
void draw() {  
    background(255);  
    translate(300, 300);  
    float s=2+frameCount/100;//1+0.01*frameCount;  
    for (int i=0; i<n; i++) {  
        float ta= 2*PI*i/n;  
        float tb= s*ta;  
        PVector a= new PVector(cos(ta), sin(ta));  
        PVector b= new PVector(cos(tb), sin(tb));  
        a.mult(r);  
        b.mult(r);  
        line(a.x, a.y, b.x, b.y);  
    }  
}
```

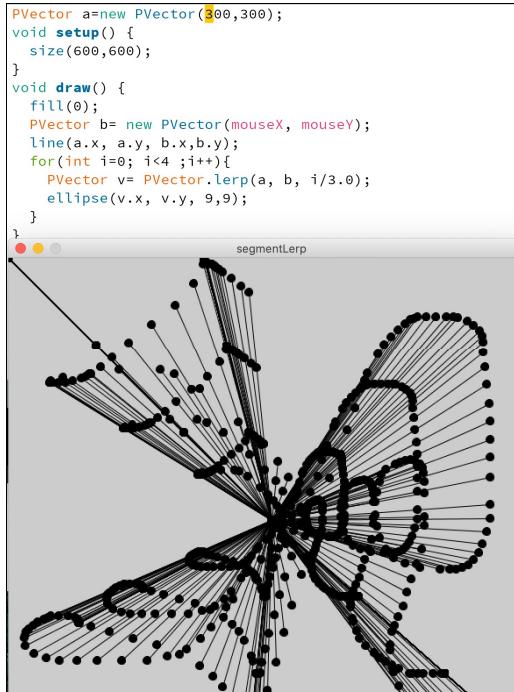


图 5-5 线段上增加了插值点

这里的关键参数 `s` 与之前稍有不同，从 `0.01*frameCount` 变成了 `frameCount/100`，后者为整数乘法。譬如，`frameCount` 从 0 到 99 时除法的结果始终为 0，从 100 到 199 时的结果始终为 1，从 200 到 299 时的结果始终为 2。因此，整个图形每隔 100 帧才变化一次。

现在删除最后一句 `line(a.x, a.y, b.x, b.y);`，再加上如下两行实现线性插值的代码：

```
PVector v= PVector.lerp(a, b, 0.5);
ellipse(v.x, v.y, 8,8);
```

运行结果如图 5-6 所示，图中第一行为 `lerp(a, b, 0.3);` 的运行结果，第二行为 `lerp(a, b, 0.5);` 的运行结果，第三行为 `lerp(a, b, 0.7);` 的运行结果。

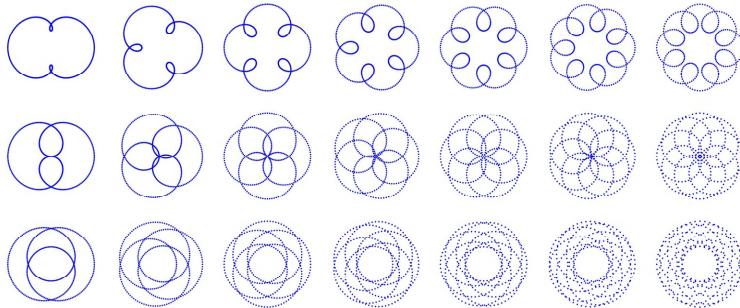
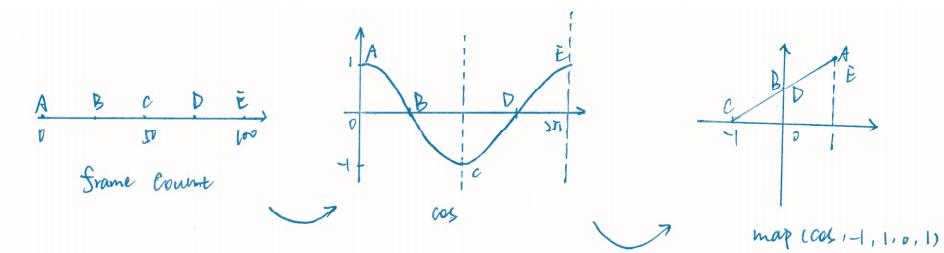


图 5-6 三组线性插值参数产生了三组图形

不难看出，`lerp()`中的系数（第三个参数）决定了具体的图形。我们可以让该系数连续变化，从而让图形连续变化。关键是要在每 100 帧内，让该系数从 1 变到 0 再变回 1，代码可以写为：

```
float cos= cos(2*PI*frameCount/100.0);
float sc = map(cos, -1, 1, 0, 1);
PVector v= PVector.lerp(a, b, sc);
```

注意，其中 `frameCount/100.0` 采用了小数除法。前两行代码把 `frameCount` 转换成一个在 0~1 范围内变化的小数，原理如下图。



完整的代码如下：

```

float r=240;
int n=360;
void setup() {
    size(600, 600);
}
void draw() {
    background(255);
    translate(300, 300);
    fill(0, 0, 255);
    noStroke();
    float s=2+frameCount/100;//1+0.01*frameCount;
    for (int i=0; i<n; i++) {
        float ta= 2*PI*i/n;
        float tb= s*ta;
        PVector a= new PVector(cos(ta), sin(ta));
        PVector b= new PVector(cos(tb), sin(tb));
        a.mult(r);
        b.mult(r);
        float sc = map(cos(2*PI*frameCount/100.0), -1, 1, 0, 1);
        PVector v= PVector.lerp(a, b, sc);
        ellipse(v.x, v.y, 8, 8);
    }
}

```

```

    }
}

```

运行程序，我们将看到一个连续跳动的图形，而且永远不会重复。

下面我们来看本节的最后一个例子。5.3 节编写了弹球程序的 **PVector** 版本，现在我们把它和线性插值结合起来。假设场景中有 5 个弹球，我们可以打开 **\*\*\*bouncePVector** 程序，把其中的 n 设为 5。我们在相邻编号的小球之间连线：ps[0]连接 ps[1]、ps[1]连接 ps[2]、ps[2]连接 ps[3]、ps[3]连接 ps[4]、ps[4]连接 ps[0]。最后一根连线比较特殊，它将最后一个编号和第一个编号连接起来。幸运的是，我们可以用 for 循环统一处理编号问题，程序如下：

```

for (int i=0; i<n; i++) {
    PVector a= ps[i];
    PVector b= ps[(i+1)%n];
    //在线段 ab 内进行插值
}

```

假设我们要在每根连线上插入 63 个点，只需要把下面这段代码加到 **\*\*\*bouncePVector** 中 **draw()** 方法的末尾。

```

float dots=63;
for (int i=0; i<n; i++) {
    PVector a= ps[i];
    PVector b= ps[(i+1)%n];
    for (int j=0; j<dots; j++) {
        PVector c= PVector.lerp(a, b, j/dots);
        ellipse(c.x, c.y, 2, 2);
    }
}

```

完整的代码可在\*\*\*下载，其运行效果如图 5-7 所示。

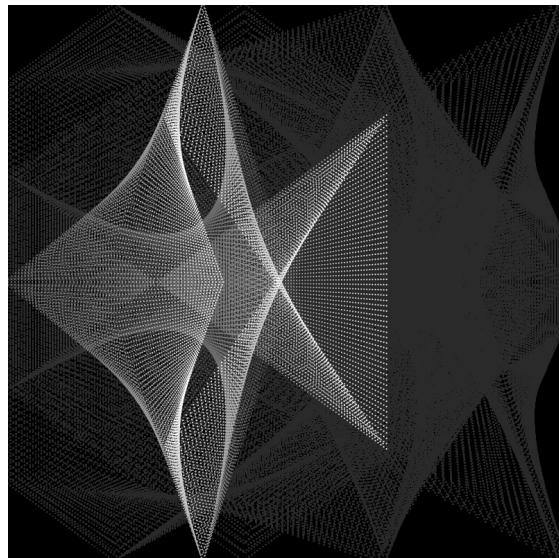


图 5-7 弹球程序的插值版本

## 第6章

# 飘

## 6.1 回旋

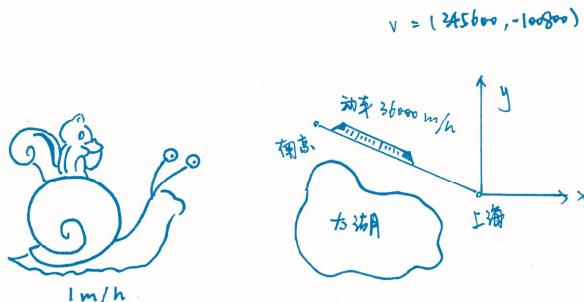
一把沙子在地上散开，一滴墨汁在水中扩散，浩瀚的星尘在太空中聚散，都可以看作是很多点受力运动的结果。这些现象看似非常复杂，但它们背后的原理或许很简单。

三百多年前，牛顿总结了物体的运动的奥秘：

(1) 速度是位置的导数，换句话说，物体位置在每个微小时段内的变化量是一个向量(速度)，在没有外力作用的情况下，该速度向量是恒定不变的。

(2) 加速度是速度的导数：速度在每个微小时段内的变化量是一个向量(加速度)，加速度与物体所受的力成正比。

值得注意的是，日常生活中所说的速度是一个数字，可以看作是速度向量的长度。譬如，一辆从南京开往上海的火车，速度是360千米每小时，它的速度向量为  $v = (345.6, -100.8)$  千米每小时。



在 6.1 节和 6.2 节中，我们将专注于速度；而 6.3 节将通过加速度让速度发生变化。在 5.3 节的单个弹球程序中，我们已经用

```
PVector p;
PVector d;
```

来表示小球的位置与速度。然后在 `draw()` 方法中用 `p.add(d);` 使小球的位置发生变化。

现在我们沿用这种思路，设速度的水平分量为 `cos(f)`，其中 `f` 是一个随帧数增长的数。完整的代码如下：

```
PVector p=new PVector(200, 200);
void setup() {
    size(400, 400);
    background(255);
    strokeWeight(3);
}
void draw() {
    float f= 0.01*frameCount;
    PVector v= new PVector(cos(f), 0 );
    v.mult(1.5); //speed
    p.add(v);
    point(p.x, p.y);
}
```

运行程序，小黑点沿水平方向来回运动，它的速度是周期性变化的。如果我们对速度向量稍作改动：

```
PVector v= new PVector(0, cos(f) );
```

那么小黑点会沿垂直方向徘徊。如果速度向量为：

```
PVector v= new PVector(cos(f), cos(f) );
```

那么小黑点会沿  $45^\circ$  的方向来回振荡。这三种情况如图 6-1 所示。

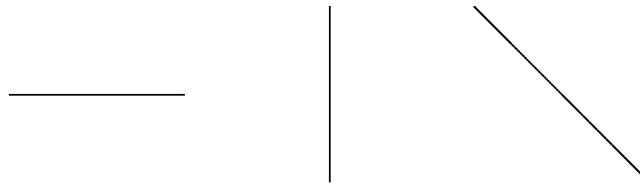


图 6-1 黑点来回振荡产生的轨迹

当速度向量  $v$  的水平分量和垂直分量的变化周期不同时，小黑点的轨迹如何？我们用以下代码来试一试：

```
float f1= 0.01*frameCount;
float f2= 0.02*frameCount;
PVector v= new PVector(cos(f1), cos(f2));
```

结果小黑点的运行轨迹呈“8”形。可以想象，当  $f1$  和  $f2$  中的系数不同时，小黑点的轨迹就会呈现出不同的形状。图 6-2 列出了六种系数组合对应的图形，大家也可以试试其他的系数组合。

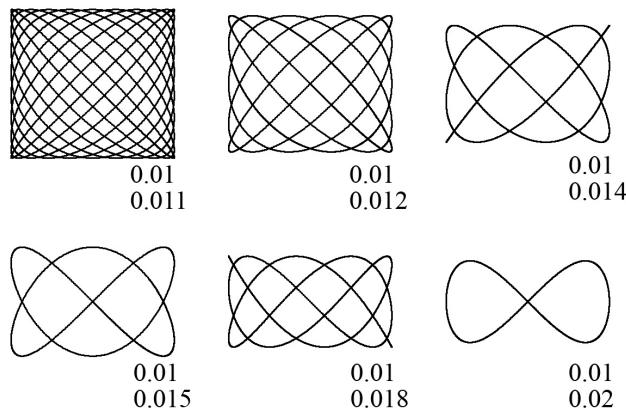


图 6-2 不同参数设置下黑点的运动轨迹

如果大家想在程序运行的某一时刻把窗口显示的内容保存下来，只需要在程序的最后添加一个 `keyPressed()` 方法，代码如下：

```
void keyPressed(){  
    save("shot.png");  
}
```

名为“shot.png”的截图会保存在程序所在的文件夹中。后保存的图片会覆盖之前保存的图片。如果不想被覆盖，我们可以用帧数为截图命名，代码如下：

```
void keyPressed(){  
    save(frameCount+".png");  
}
```

其实 `keyPressed()` 方法与 `setup()` 和 `draw()` 方法一样，都是 Processing 内部已经定义好的方法，但它们的触发机制不同。

`setup()` 方法是在程序启动时运行一次；`draw()` 方法是当 `setup()` 执行完成之后，每帧都执行一次；`keyPressed()` 方法是在每次按下键盘时执行一次。



最后我们把一个点扩展为多个点，营造出大量微粒随风而动的感觉，代码如下：

```
int w=800;
int h=600;
int n=800;
PVector[] ps= new PVector[n];
void setup(){
    size(800,600);
    for(int i=0;i<n;i++){
        ps[i]=new PVector(random(w), random(h));
    }
    background(0);
}
void draw(){
    fill(0,4);
    noStroke();
    rect(0,0,w,h);
    stroke(255);
    strokeWeight(2);
    float f1= 0.01*frameCount;
    float f2= 0.015*frameCount;
    for(int i=0;i<n;i++){
        PVector p= ps[i];
        PVector v= new PVector(cos(f1), cos(f2));
        p.add(v);
        if( 0.005 >random(1.0) )
            ps[i]=new PVector(random(w), random(h));
        point(p.x, p.y);
    }
}
```

该程序采用了两个小技巧。首先，在绘图前用一个透明的黑方块把整个屏幕盖住，代码如下：

```

fill(0,4);
noStroke();
rect(0,0,w,h);

```

这样就造成了每个白点都拖着一个尾巴的效果，如图 6-3 所示。其次，随机选择一小部分点，分别给它们一个随机的位置，代码如下：

```

if( 0.005 >random(1.0) )
ps[i]=new PVector(random(w), random(h));

```

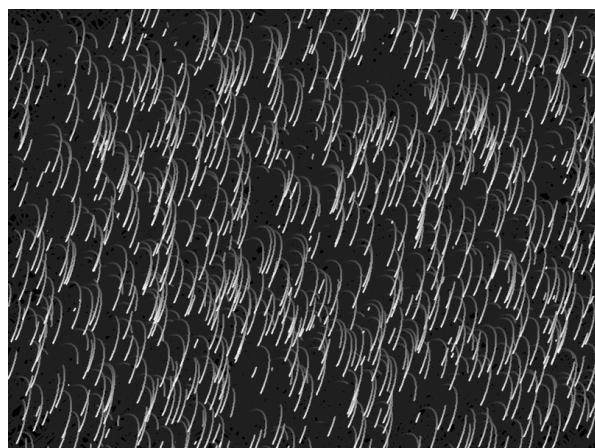


图 6-3 随风而动的白点的运动轨迹

这样就增加了整个画面的动感。其中，`0.005>random(1.0)`表示 5% 的概率，因此每帧大约有四个点被移动到随机的位置。

## 6.2 秩序与随机

在 6.1 节中，点的运动很有规律，具有周期性。每个周期，点的运动轨迹都会完全重复一次。现在我们来试验一种完全随机的运动，即速度向量的长度为 1，但方向随机，代码如下：

```
PVector v= PVector.random2D();
p.add(v);
```

这种随机运动有点像微粒在空气中的布朗运动，或是醉汉毫无规律的的步伐。假设一名醉汉离悬崖的距离是 10 步，随着步伐的增加，他跌下悬崖的概率有多大？在 Processing 里很容易模拟这种随机步伐：让一个点从屏幕中央开始随机移动，监测它何时跑到屏幕外面。代码如下：

```
if( p.x<0 || p.x>w || p.y<0 || p.y>h) {
    //点在屏幕外面
}
```

完整的代码如下：

```
int w=800;
int h=800;
PVector p=new PVector(w/2, h/2);
int count=0;
void setup() {
    size(w, h);
    background(255);
    stroke(0,40);
}
void draw() {
    PVector v= PVector.random2D();
    p.add(v);
    point(p.x, p.y);
    if( p.x<0 || p.x>w || p.y<0 || p.y>h) {
        println( "done" );
        noLoop();
    }
}
```



程序需要运行很久，才能看到随机点走出屏幕，如图 6-4 所示。有一种加速的办法，就是把原来 `draw()` 中的内容写在一个自定义方法（参见 4.2 节）中，然后在 `draw()` 中多次调用该自定义方法。譬如，每帧走 200 步（[代码\\*\\*\\*](#)），代码如下：

```
void draw() {
    for (int i=0; i<200; i++)
        display();
}

void display() {
    //原来 draw()中的代码
}
```

这是一种常用的技巧，每当我们觉得程序运行得太慢时，就可以尝试这种加速方式。

从图 6-4 中可以看出，如果速度向量的方向是完全随机的，那么点的移动轨迹则非常无序。现在我们用 `noise()` 方法来确定速度向量的方向，代码如下：

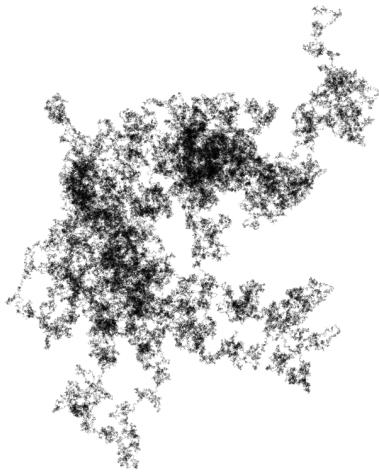


图 6-4 随机运动的微粒留下的轨迹

```
PVector p=new PVector(400, 400);
void setup() {
    size(800, 800);
    background(255);
    strokeWeight(2);
}
void draw() {
    float theta=noise( 0.0006*p.x , 0.0006*p.y )*8*PI;
    PVector v= new PVector(cos(theta), sin(theta) );
    p.add(v);
    point(p.x, p.y);
}
```

运行上述代码，点的移动轨迹变成了一条平滑的曲线，图 6-5 第一排为程序运行三次的结果。4.1 节已经介绍过 `noise(x,y)`方法：输入 `x`、`y` 两个坐标参数，`noise` 方法将返回一个随机值。但这个随机值不是完全随机的，而是沿 `x` 和 `y` 方向平滑变化的。如果把系数变成 `noise( 0.004*p.x , 0.004*p.y )`，程序运行的结果如图 6-5 中第二排所示；如果把系数变成 `noise( 0.04*p.x , 0.04*p.y )`，点的轨迹就更加随机了（见图 6-5 中第三排）。可以发现，所用的系数越小，轨迹越平滑；

系数越大，轨迹越曲折、随机。

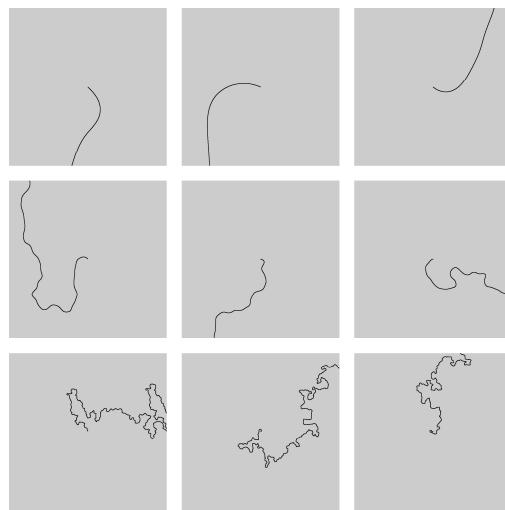


图 6-5 不同系数时 noise 方法下的点的轨迹

下面我们把很多条轨迹叠合在一起：一条轨迹走出屏幕边缘时，下一条轨迹从屏幕中央出发。每条轨迹启动时，需要用 `noiseSeed()` 来设定一个随机种子，否则所有的轨迹会完全重合在一起。完整的代码见 ([\\*\\*\\* wind\\_noise\\_single](#))，运行结果如图 6-6 所示。

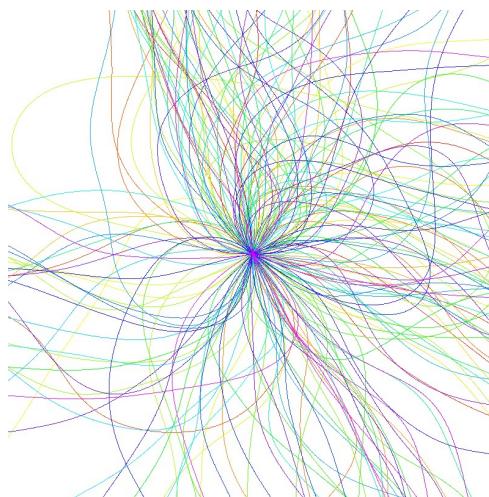


图 6-6

当系数为 `noise( 0.003*p.x, 0.003*p.y )` 时，产生的轨迹如图 6-7 所示。

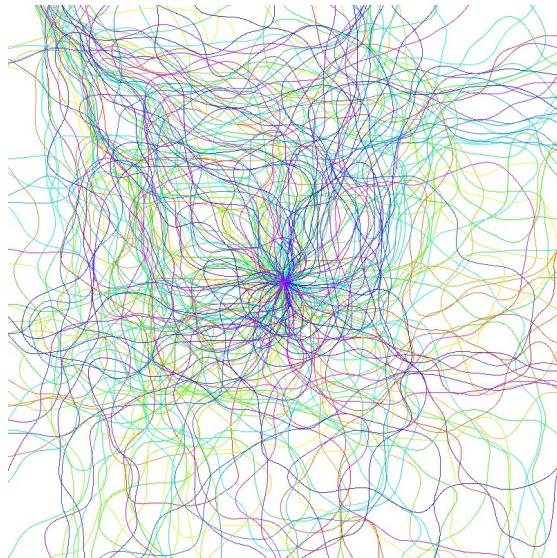


图 6-7

当系数为 `noise( 0.015*p.x, 0.015*p.y )` 时，结果如图 6-8 所示。

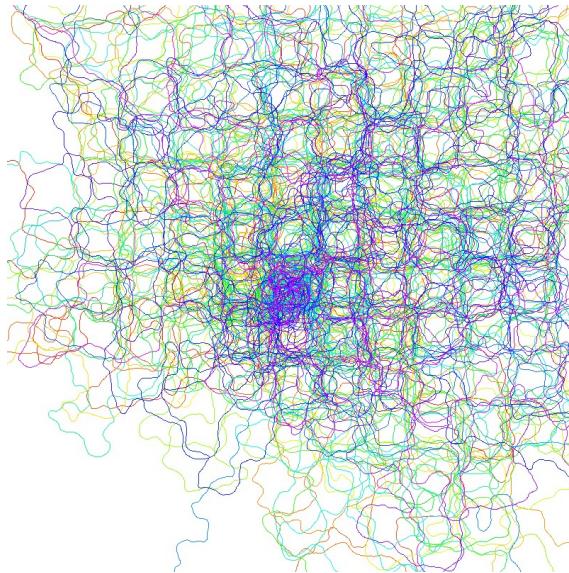


图 6-8

之前的几个例子都只是考虑了单个点的运行轨迹，现在我们用 noise 方法让很多点运动起来。下面这个程序和 6.1 节的最后一个例子十分相似，但是下面的程序使用 noise 方法来确定速度向量的方向。

```
int w=800;
int h=600;
int n=800;
PVector[] ps= new PVector[n];
void setup(){
    size(800,600);
    for(int i=0;i<n;i++)
        ps[i]=new PVector(random(w), random(h));
    background(0);
}
void draw(){
    fill(0,4);
    noStroke();
    rect(0,0,w,h);
    stroke(255);
    strokeWeight(2);
    for(int i=0;i<n;i++){
        PVector p= ps[i];
        float theta=noise( 0.003*p.x , 0.003*p.y )*4*PI;
        PVector v= new PVector(cos(theta), sin(theta) );
        v.mult(0.5); //speed
        p.add(v);
        if( 0.005>random(1.0) || 0>p.x || w<p.x || 0>p.y || h<p.y){
            ps[i]=new PVector(random(w), random(h));
        }
        point(p.x, p.y);
    }
}
```

```

    }
}

```

图 6-9 为程序运行某一时刻的截图。

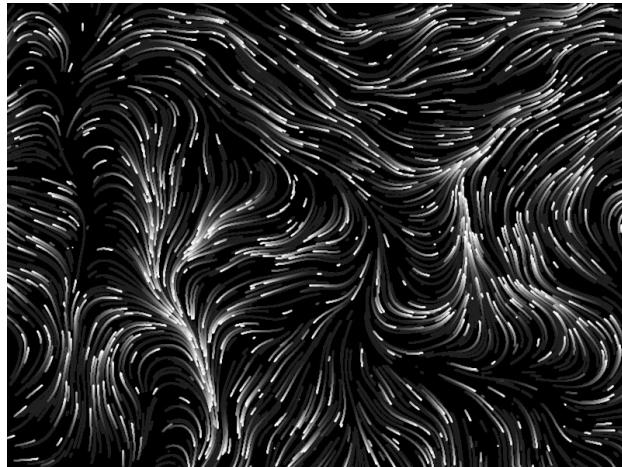


图 6-9 用 noise 方法控制每个微粒的速度向量

最后我们要把以下两种速度结合起来：

(1) 6.1 节最后一个例子中的回旋速度向量采用了 cos 函数：

```

float f1= 0.01*frameCount;
float f2= 0.015*frameCount;
PVector v= new PVector(cos(f1), cos(f2));

```

(2) 本节的 noise 速度向量，代码如下：

```

float theta=noise( 0.003*p.x , 0.003*p.y )*4*PI;
PVector v= new PVector(cos(theta), sin(theta));

```

如何把两个速度向量结合起来？其实很简单，用向量加法（见 5.1 节、5.2 节）即可，代码如下：

```

PVector a= new PVector(cos(theta), sin(theta));
PVector b= new PVector(cos(f1), cos(f2));

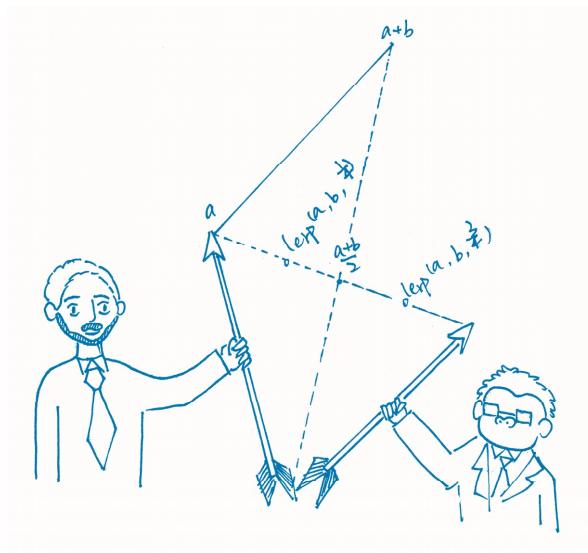
```

```
PVector v= PVector.add (a,b);  
v.mult(0.5);
```

最后两句代码等价于线性插值（见 5.4 节）：

```
PVector v= PVector.lerp(a,b,0.5);
```

实际上，线性插值可以有很多自由度，它可以实现从  $a$  (noise 速度向量) 到  $b$  (cos 回旋速度向量) 的任意比重。譬如，`PVector v= PVector.lerp(a,b, 1/4.0);`侧重于 noise 速度向量，而 `PVector v= PVector.lerp(a,b, 3/4.0);`侧重于 cos 回旋速度向量。



以下是两种速度相结合的完整代码：

```
int w=1100;  
int h=800;  
int n=800;  
PVector[] ps= new PVector[n];  
void setup() {  
    size(w, h);  
    for (int i=0; i<n; i++)
```

```

ps[i]= new PVector(random(w), random(h));
background(0);
colorMode(HSB);
}

void draw() {
    fill(0, 4);
    noStroke();
    rect(0, 0, w, h);
    stroke(255);
    float f1= 0.015*frameCount;
    float f2= 0.01*frameCount;
    for (int i=0; i<n; i++) {
        PVector p= ps[i];
        float theta=noise( 0.003*p.x , 0.003*p.y )*4*PI;
        PVector a= new PVector(cos(theta), sin(theta) );
        PVector b= new PVector(cos(f1), cos(f2) );
        PVector v= PVector.lerp(a,b,0.4);
        p.add(v);
        if ( 0.005>random(1.0) ||p.x<0 || p.x>w || p.y<0 || p.y>h)
            ps[i]= new PVector(random(w), random(h));
        float mag= v.mag();
        strokeWeight(1 + 0.6/(0.01+mag));
        stroke(100*mag, 255, 255);
        point(p.x, p.y);
    }
}

```

在绘图时，速度向量 `v` 的长度决定了笔触的宽度，`strokeWeight(1+0.6/(0.01+mag))`，而颜色 `stroke(100*mag, 255, 255)` 使整个画面更加丰富，如图 6-10 和图 6-11 所示。

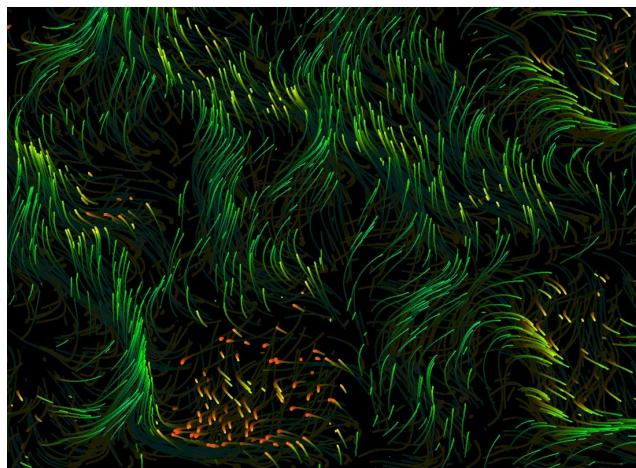


图 6-10 增加笔触和颜色的变化 (一)

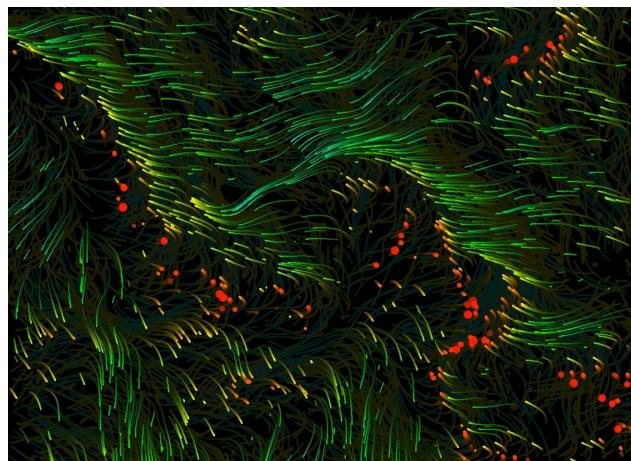


图 6-11 增加笔触和颜色的变化 (二)

### 6.3 奇怪吸引子

银河、云朵、火焰等是由无数微粒构成的，它们的形状似乎永远不会重复。我们可以用引力来模拟这类复杂的现象。首先考虑一个非常简单的场景：一个

粒子被两个吸引子吸引。6.1 节提到过，力（用加速度表示）改变速度向量，准确的说，加速度是速度的导数。我们先设定两个吸引子的位置，代码如下：

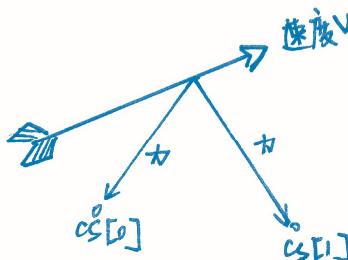
```
PVector [] cs= {new PVector(0, -150), new PVector(0, 150)};
```

在 `draw()` 方法中，首先，依次用两个吸引子来改变速度向量 `v`，代码如下：

```
for (int j=0; j<cs.length; j++) {
    PVector accel= PVector.sub(cs[j], p);
    accel.setMag(0.01);
    d.add(accel);
}
```

这里的向量 `accel` 表示引力的方向，而力的大小为 0.01，这里默认粒子的质量为 1。然后，再用速度向量来改变粒子的位置，代码如下：

```
p.add(d);
```



完整的代码如下：

```
PVector p;
PVector d=new PVector();
PVector [] cs= {new PVector(0, -150), new PVector(0, 150)};
void setup() {
    size(400, 500);
    p= PVector.random2D();
    p.mult(180);
```

```

background(255);
}

void draw() {
    translate(200, 250);
    for (int j=0; j<cs.length; j++) {
        PVector accel= PVector.sub(cs[j], p);
        accel.setMag(0.01);
        d.add(accel);
    }
    p.add(d);
    point(p.x, p.y);
}

```

粒子的初始位置是在一个半径为 180 像素的圆上，粒子的初始速度为 0。图 6-12 截取了程序运行的几种典型结果，有的像枕头，有的像弓。有趣的是，粒子的轨迹很难完全重复先前的路径；或者说，相似的轨迹之间总有一点微小的差距，这就是奇怪吸引子（strange attractor）在作怪。



图 6-12 奇怪吸引子作用下的粒子运动轨迹

如果吸引子固定在其他位置，产生的图形也会不同。譬如，下面这个程序把两个吸引子分别放在 x 轴和 y 轴上，然后同时让三个粒子运动起来。

```

int n=3;
PVector [] ps=new PVector[n];
PVector [] ds=new PVector[n];
PVector [] cs= {new PVector(120, 0), new PVector(0, -120)}; //120
void setup() {

```

```
size(1000, 800);
for (int i=0; i<n; i++) {
    ds[i]=new PVector();
    PVector v= PVector.random2D();
    v.mult(300);
    ps[i]= v;
}
background(255);
colorMode(HSB);
}
void draw() {
    translate(500-40,400+40);
    for (int i=0; i<50; i++)
        display();
}
void display() {
    for (int i=0; i<n; i++) {
        for (int j=0; j<cs.length; j++) {
            PVector accel= PVector.sub(cs[j], ps[i]);
            accel.setMag(0.008);
            ds[i].add(accel);
        }
        ps[i].add(ds[i]);
        stroke(255*i/n, 255, 255);
        point(ps[i].x, ps[i].y);
    }
}
```

运行上述程序，生成的图形呈 45° 倾斜，由于 45° 线是两个吸引子的对称轴。循环往返，但不会完全重复之前的路径，这是奇怪吸引子的魔法（见图 6-13）。

此外，微粒初始位置（或速度）的微小变化，也可能使后来的轨迹截然不同。1963年，洛伦兹（Edward Lorenz）发表了一篇名叫《一只在巴西的蝴蝶拍一下翅膀会不会在得克萨斯州引起龙卷风？》的论文，论述系统中如果初期条件差一点点，结果会很不稳定，他把这种现象戏称为“蝴蝶效应”，自此奇怪吸引子被大众所熟知。

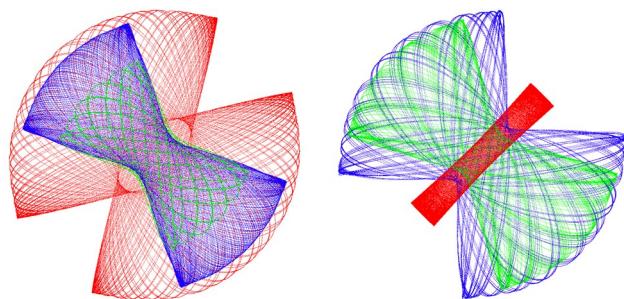


图 6-13 奇怪吸引子导致粒子的轨迹很不稳定



下面我们来模拟大量粒子在两个吸引子作用下的运动。这些粒子一开始随机分布在一个半径为380像素的圆上，初始速度为0。任意时刻，每个粒子的速度向量都受到两个力的作用。

```
int n=2345;
PVector [] ps=new PVector[n];
PVector [] ds=new PVector[n];
PVector [] cs= {new PVector(-300,0), new PVector(300, 0)};
void setup() {
    size(1000, 800);
    for (int i=0; i<n; i++) {
```

```

ds[i]=new PVector();
PVector v= PVector.random2D();
v.mult(380);
ps[i]= v;
}
background(0);
}

void draw() {
  fill(0, 8);
  noStroke();
  rect(0, 0, 1000, 800);
  translate(500,400);
  stroke(255);
  for (int i=0; i<n; i++) {
    for (int j=0; j<cs.length; j++) {
      PVector accel= PVector.sub(cs[j], ps[i]);
      accel.setMag(0.008);
      ds[i].add(accel);
    }
    ps[i].add(ds[i]);
    point(ps[i].x, ps[i].y);
  }
}

```

对于每个粒子来说，它的运动规律为：

$$\frac{dp}{dt} = v$$

$$\frac{dv}{dt} = k \sum_i \frac{c_i - p}{\|c_i - p\|}$$

其中，向量  $p$  是位置；向量  $v$  是速度；向量  $c_i$  表示第  $i$  个吸引子的位置；

$k$  是一个常数 ( $k=0.008$ )。图 6-14 截取了运行过程中的六个画面。

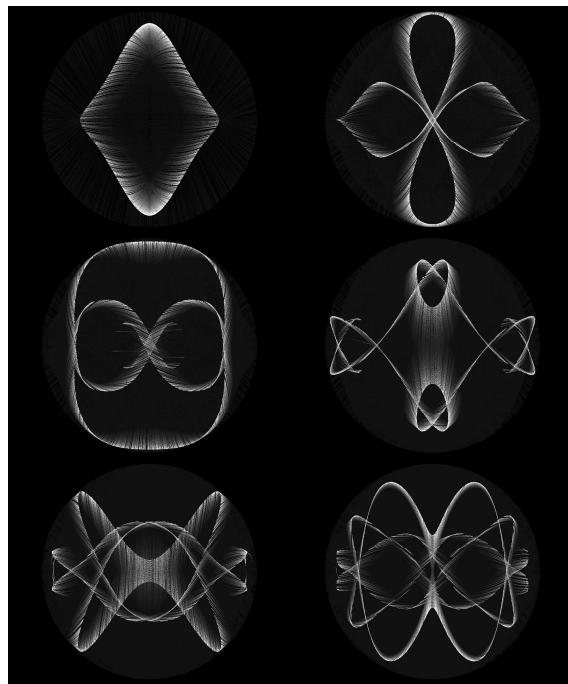


图 6-14 左右两个吸引子作用下产生的轨迹



如果把两个吸引子改成 `PVector[] cs= {new PVector(120, 0), new PVector(0, -120)}`; 就会得到完全不同的效果, 如图 6-15 所示。

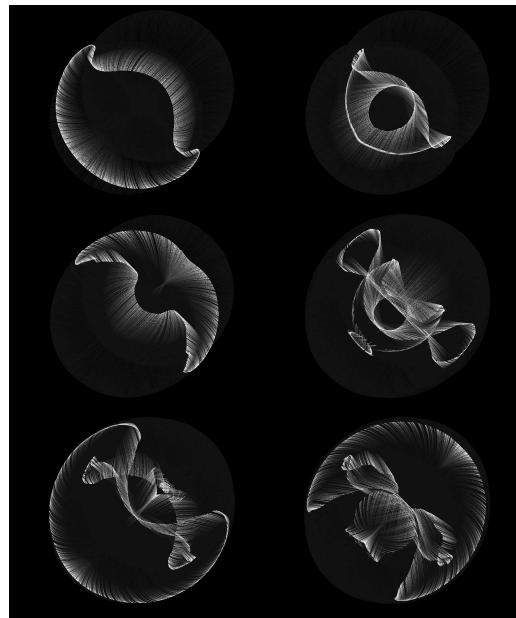


图 6-15 两个倾斜的吸引子作用下产生的轨迹

在之前的几个例子中，粒子的初速度都是 0，即一开始每个粒子都是静止的。而奇怪吸引子的有趣之处在于：稍微不同的初始状态有可能极大地影响后面的运行结果。下面我们为每个粒子设置一个初速度，其（初始）速度向量和（初始）位置向量相互垂直，在 `setup()` 中设定，代码如下：

```
for (int i=0; i<n; i++) {
    PVector p= PVector.random2D();
    PVector v=new PVector(p.y, -p.x);
    v.mult(1.8);
    ds[i]=v;
    p.mult(random(10, 300));
    ps[i]=p;
}
```

其中，`ds[i]=new PVector(p.y, -p.x);`确保了速度向量 `ds[i]` 垂直于位置向量 `p`(5.1 节讨论了两个向量垂直的数学关系)。程序 (\*\*\*) 运行的状态如图 6-16 所示。

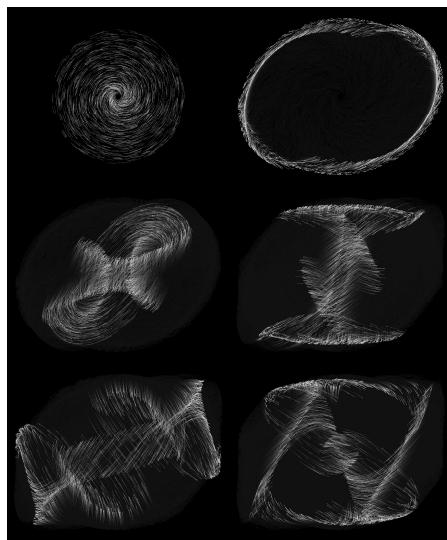
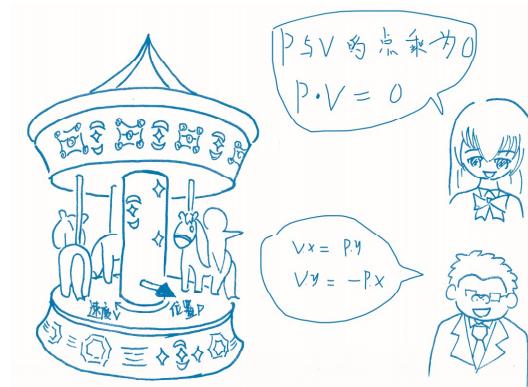


图 6-16 有初速度的粒子产生的轨迹

吸引子控制的粒子运动包含多种变数：

- (1) 每个粒子的初始位置和初始速度向量。
- (2) 吸引子的位置和个数。以上几个例子中都采用了两个吸引子，三个以上吸引子或许更有趣。我们之前都采用了位置固定的吸引子，实际上吸引子也可以实时移动。
- (3) 吸引子对粒子的拉力。在本节的例子中，力的大小与粒子和吸引子之间的距离无关。但根据万有引力，力的大小与距离的平方成反比。



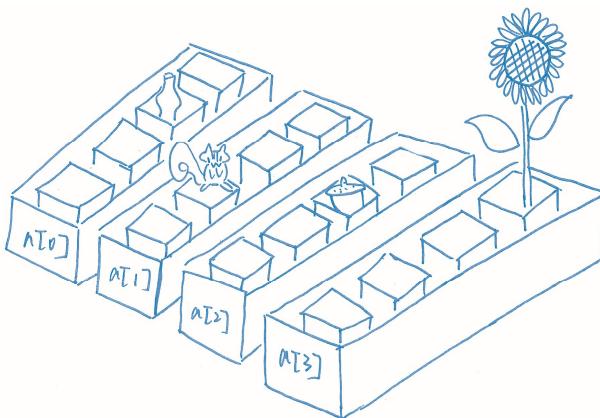
## 第7章

# 一石激起千层浪

## 7.1 二维数组

第4章中用 `int` 型（或 `color` 型）的数组来表示窗口中的所有点，某一行某一列的点在这个数组中有一个唯一的编号（需要通过公式计算），这种方式并不是很直观。窗口内的点分布在二维平面中，有没有更自然的方式来表示这样的二维阵列呢？答案是采用二维数组（或嵌套数组）。实际上，Processing（Java）还支持三维、四维等高维度的数组。需要澄清的是，数组的维度和现实世界的空间维度并不是一回事，只不过我们恰巧用二维数组来表示二维的阵列而已。

如果一般的数组是一排盒子，那么二维数组就是箱子套箱子，如下图所示。



四个大盒子为 `a[0]`、`a[1]`、`a[2]`、`a[3]`，这四个大盒子里面又放了小盒子。可

乐瓶所在的盒子的编号是  $a[0][2]$ ，松鼠躲在编号为  $a[1][1]$  的盒子里面，而向日葵长在编号为  $a[3][3]$  的盒子里面。如果在编程过程中，需要在二维盒子阵列里面储存整数，可以声明这样的二维数组，代码如下：

```
int[][] a= new int[3][2];
```

这是一个 3 行 2 列的整数型二维数组。可以通过两个方括号内的数字来指定某一个盒子，代码如下：

```
a[0][0]=-9;  
a[0][1]=0;  
a[1][0]=-7;  
a[1][1]=4;  
a[2][0]=-5;  
a[2][1]=6;
```

如果预先知道每个元素的值，可以非常简洁地把以上代码写在一行中，如下所示：

```
int[][] a= {{-9, 0}, {-7, 4}, {-5, 6}};
```

其中，两层花括号对应二维数组的两个维度。`a.length` 表示大盒子的总数，`a[0].length` 表示第一个大盒子中小盒子的总数。我们可以用下面的代码确认：

```
int[][] a= {{-9, 0}, {-7, 4}, {-5, 6}};  
println(a.length);  
for(int i=0; i<a.length;i++)  
    println(a[i].length);
```

对于二维数组，人们经常需要对每行（或每列）进行运算，如求和、平均值、归一化（normalization）等。我们先针对一般数组写出代码，然后再尝试把这些代码用到二维数组的每行（或每列）。

计算数组 `float[] a={1,-2,3};` 的和，代码如下：

```
float x=0;
```

```
for(int i=0;i<a.length;i++)
    x+= a[i];
```

求数组 a 的平均值，代码如下：

```
float x=0;
for(int i=0;i<a.length;i++)
    x+= a[i];
x/=a.length;
```

把数组 a 归一化，代码如下：

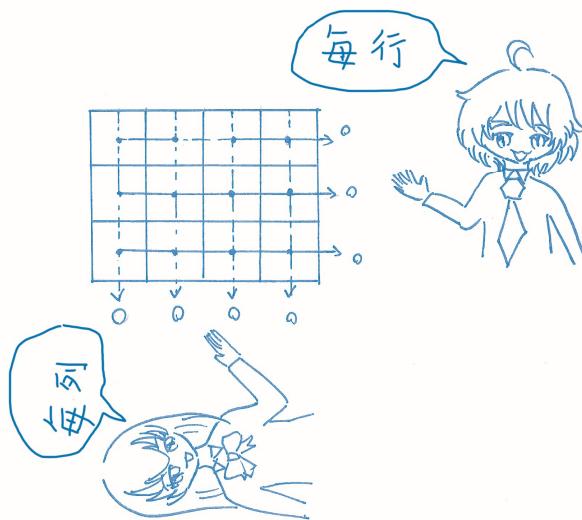
```
float x=0;
for(int i=0;i<a.length;i++)
    x+= a[i]* a[i];
x=sqrt(x);
```

下面我们先创建一个二维数组，并用 for 循环给每个元素赋值，代码如下：

```
float[][] a= new float[3][4];
for(int i=0; i<a.length;i++){
    for(int j=0; j<a[i].length;j++){
        a[i][j]= i-2*j;
        print(a[i][j]+",");
    }
    println();
}
```

打印的结果（3 行 4 列的小数）为：

```
0.0,-2.0,-4.0,-6.0,
1.0,-1.0,-3.0,-5.0,
2.0,0.0,-2.0,-4.0,
```



现在我们求每行的平均值，并把结果放在一个数组 b 内，代码如下：

```
float[] b= new float[a.length];
for(int i=0; i<a.length;i++){
    float x=0;
    for(int j=0; j<a[i].length;j++)
        x+= a[i][j];
    b[i]=x/a[i].length;
}
println(b);
```

打印的结果为：

[0] -3.0 [1] -2.0 [2] -1.0

我们也可以求每列的平均值，并把结果放在一个数组 c 内，代码如下：

```
float[] c= new float[a[0].length];
for(int j=0; j<a[0].length;j++){
    float x=0;
    for(int i=0; i<a.length;i++)
        x+= a[i][j];
    c[j]=x/a.length;
}
println(c);
```

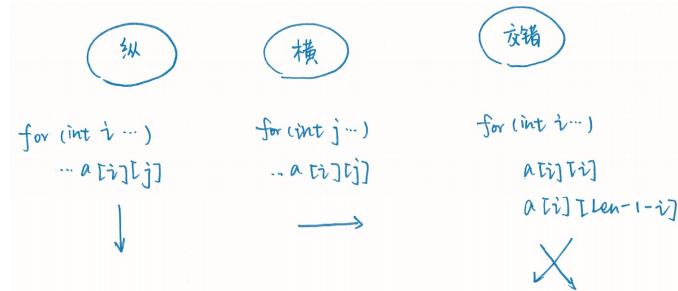
```

x+= a[i][j];
c[j]=x/a.length;
}
println(c);

```

打印的结果为：

[0] 1.0 [1] -1.0 [2] -3.0 [3] -5.0



有了二维数组之后，人们就可以很方便地来表示程序窗口内的二维整列。

图 7-1 所示的例子创建了纵横排列的彩色方块，每个方块的色相（hue）依次递增。每次 `draw()`方法运行的时候，我们让每个方块的色相值增加 1，从而产生彩色流动的错觉（见图 7-1）。

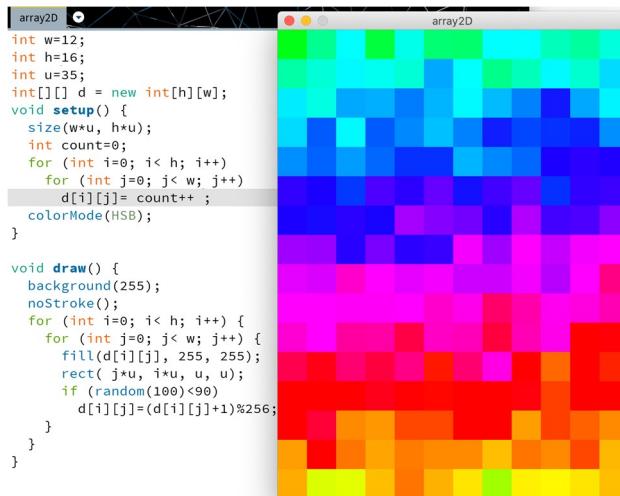


图 7-1 用二维数组组织色块

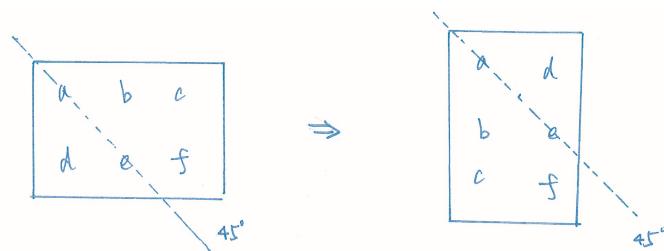
其中，`d[i][j] = count++;` 这句代码比较让人费解，其实它相当于下面这两行代码：

```
d[i][j] = count;  
count++;
```

此外，`draw()`方法中的 `if(random(100)<90)`语句增加了一点颜色变化的随机性，每个色块有 90% 的概率会发生颜色渐变。



例习题 1：把一个二维数组的行与列调换，数学上称为矩阵的转置，即把二维数组沿着  $45^\circ$  斜线进行翻转。



我们可以定义一个方法来实现该功能，其格式为：

```
int[][] transpose(int[][] a){  
}  
}
```

或

```
float[][] transpose (float[][] a){  
}
```

等等。

例习题 2：定义一个方法，把一个二维数组中的所有元素放到一个普通数组（即一维数组）内。

该方法的格式为：

```
int[] put(int[][] a){  
}
```

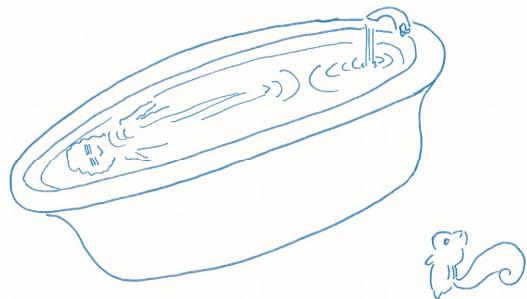
或

```
float[] put(float[][] a){  
}
```

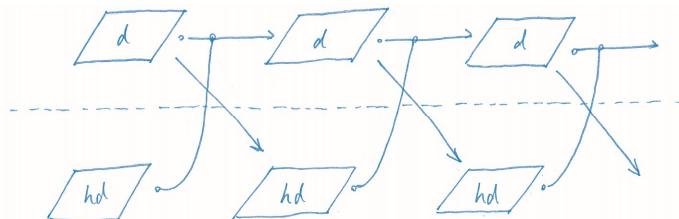
等等。

## 7.2 涟漪

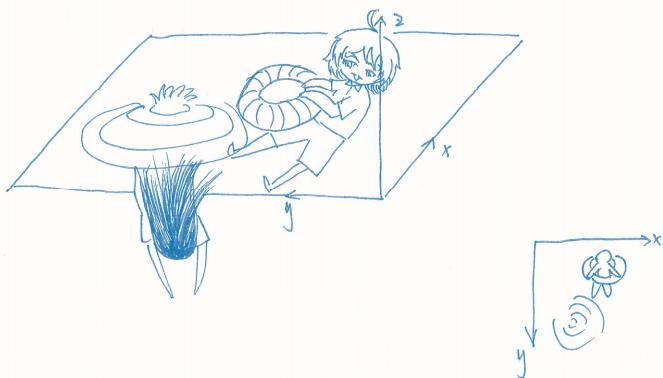
大人和孩子都爱玩水，因为水的形状、光影和手感实在是太奇妙了。传说有一天，阿基米德在浴缸里洗澡时忽然大喊“Eureka！”原来他发现了浮力的大小等于他身体排开的水的重量，这就是著名的浮力定理。这个例子暗示我们：水种种难以琢磨的现象背后，可能隐藏着简单的数学原理。本节介绍一种形成涟漪的数学原理，并用非常简短的代码来生成动态的涟漪图案。



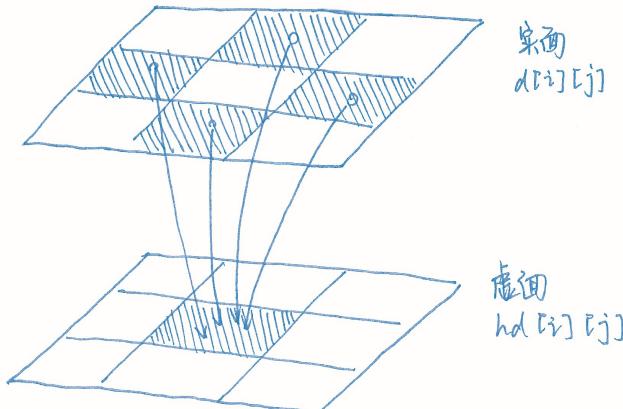
首先，想象一下这样的场景：一对孪生的水面，其中一个水面是我们能看到的面，我们暂且把它称为实面；另一个水面是看不到的，我们暂且把它称为虚面。这两个水面相互叠加，从而形成“推波助澜”的效果。具体来说就是：①实面变成下一时刻的虚面；②实面与虚面相互叠加产生下一时刻的实面。把实面标记为 d（data 的缩写），把虚面标记为 hd（意为 hidden data），两者的交互过程如下图所示。



从现代物理学来看，所有波似乎都是由虚、实两个量构成的。譬如，在量子力学中，波的运动规律（如薛定谔方程）是由复数来表示的，而复数（complex number）有实数部分和虚数部分。



如果我们垂直向下（或从水底向上）观察水面，并把一个矩形范围内的水面划分为细密的正交网格，实面与虚面的在每个局部的叠加规律如下图所示。



具体的数值运算为：

$$d[i][j] = (d[i-1][j] + d[i+1][j] + d[i][j-1] + d[i][j+1]) * 0.5 - hd[i][j]$$

在实面中， $d[i-1][j]$ 、 $d[i+1][j]$ 、 $d[i][j-1]$ 、 $d[i][j+1]$ 分别表示上、下、左、右四个相邻元素的值。这种交互规则是局部的，它只涉及每个元素的四个邻居。有趣的是，这种局部的数学原理能够在整个窗口中生成动态、连续的波纹（见图 7-2）。完整代码如下：

```

int w=800;
int h=600;
float[][] d = new float[h][w];
float[][] hd = new float[h][w];
void setup(){
    size(w, h);
}
void mousePressed(){
    d[mouseY][mouseX]=100;
}
void draw(){

```

```

loadPixels();
float[][] t= new float[h][w];
for(int i=1; i< h-1 ; i++) {
    for(int j=1; j< w-1 ; j++) {
        t[i][j]=(d[i-1][j]+ d[i+1][j]+ d[i][j-1]+d[i][j+1])*0.5 -hd[i][j];
        t[i][j]*=0.999;
        pixels[i*w+j]= color(160*d[i][j]);
    }
}
updatePixels();
hd=d;
d=t;
}

```

该程序采用了鼠标事件方法，只要用鼠标单击窗口内的任意位置，该处的值 `d[mouseY][mouseX]` 就会变成 100，从而在该处形成一个新的涟漪。

在虚实面交互的代码中，我们参照 Daniel Shiffman 的教程，加了一句代码 `t[i][j]*=0.999;`，从而使波纹以一种非常缓慢的速度趋于平静。如果把代码变成 `t[i][j]*=0.99` 使衰减变快，那么所有波纹会很快消失。

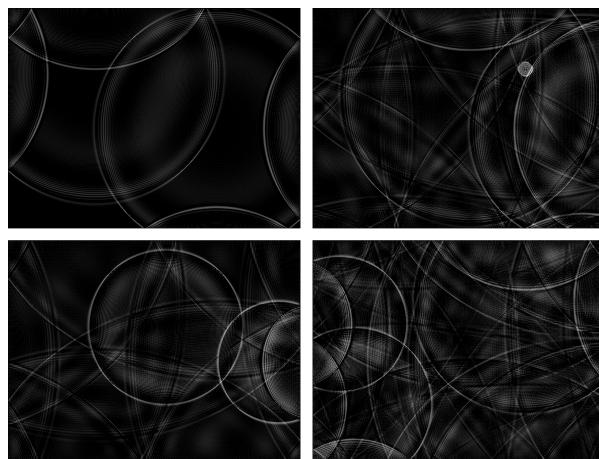


图 7-2 涟漪程序运行过程截图

以上程序非常简短，但生成的波纹相当复杂。我们可能看到：①波纹遇到边界会反弹；②波与波之间有干涉效果（interference）。但为什么那几行简短的代码会创造这样复杂的行为，是一个很难完全解释清楚的现象。这个例子体现了生成艺术的一个痛点，也是有趣的一点：

理解了每行代码，但生成的结果依然令人费解。

或许这就是科学与艺术相遇的边界。7.3 节我们会继续遇到这样的例子。

练习题：如果你学过高等数学，把程序中与

$$d[i][j] = (d[i-1][j] + d[i+1][j] + d[i][j-1] + d[i][j+1]) * 0.5 - h \cdot d[i][j]$$

对应的微分方程写出来。

## 7.3 化学反应

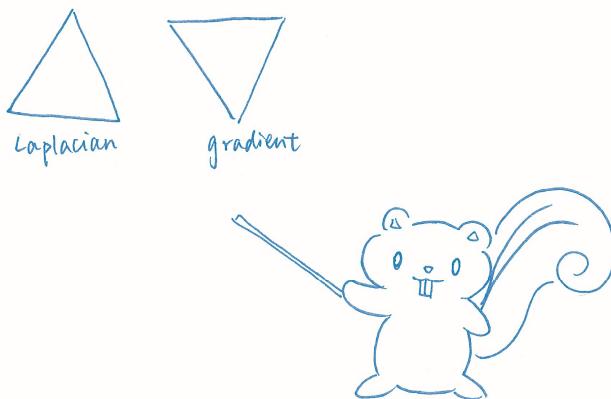
7.2 节的涟漪效果来源于虚实面的交互，而本节将模拟两种液体之间的反应。世界上绝大部分化学反向是单向的（过程是不可逆的），如点燃氢气，氢气与氧气反应瞬间就变成了水。但我们很难把水分解成氢气和氧气，否则氢能源车一定会流行起来。但世界无奇不有，在 20 世纪五六十年代化学家发现了可以“双向”进行的化学反应，如丙二酸  $\text{CH}_2(\text{COOH})_2$  与溴酸钾  $\text{KBrO}_3$  的反应。这种可逆的化学反应可以在培养皿中产生有趣的纹理，后来被人们称为“反应扩散系统”（Reaction-diffusion system）。



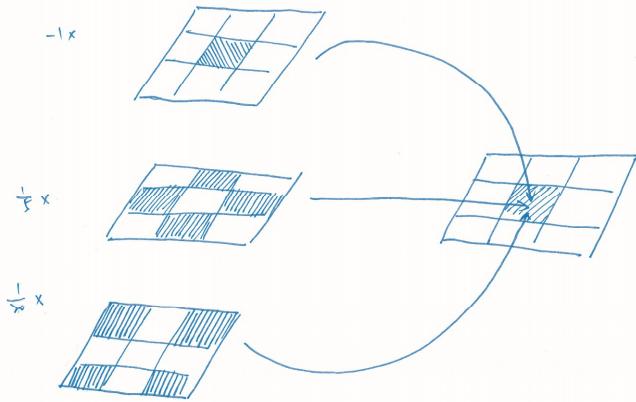
与涟漪程序中的虚实两面类似，我们要模拟的化学反应包括一种可见的液体  $u$  和一种不可见的液体  $v$ 。如果我们将程序窗口想象成培养皿中的一层液体薄膜，并将其划分成很细的网格，那么  $u$  与  $v$  相互反应的公式（代码）为：

```
float u = us[i][j];
float v = vs[i][j];
float newu=0.8*lap(us, i, j) - u*v*v + 0.99*u +0.015;
float newv=0.52*lap(vs, i, j) +u*v*v +0.935*v;
us[i][j] = constrain(newu, 0, 1);
vs[i][j]= constrain(newv, 0, 1)
```

其中， $us$  是 float 型的二维数组（见 7.1 节），用来表示物质  $u$  在整个平面中的分布。而  $u=us[i][j]$ ; 则表示平面某处  $u$  的浓度。变量  $newu$  储存了当前化学反应后  $u$  的浓度。最后需要把这个新浓度的大小限制在 0~1 之间，因此用了 `constrain()` 方法。



最让人费解的代码是 `lap(us,i,j)`，它表示拉普拉斯算子（Laplacian）。在数学中，拉普拉斯算子是多个空间维度中二次导数的和。在液体薄膜的化学反应中，拉普拉斯算子表示浓度在每一点上聚集的程度。如果周围的浓度比点(i, j)的浓度高，那么  $lap(us, i, j)$  的值就小（负值）；反之， $lap(us, i, j)$  的值就大。在正交网格中， $lap(us, i, j)$  的计算原理如下图所示。



就是把该点的值乘以-1；上下左右的值相加再乘以 1/5；四个斜对角的值相加再乘以 1/20；最后把它们相加作为该点的（算子运算后获得的）值。需要注意的是，拉普拉斯算子的输入是九个数字（九宫格），但只输出一个数字。对应的代码如下：

```
float lap(float[][] a, int i, int j) {
    float x= -a[i][j]+0.2*(a[i-1][j] + a[i+1][j] + a[i][j-1]+ a[i][j+1]);
    x+= 0.05*(a[i-1][j-1] + a[i-1][j+1] + a[i+1][j-1]+ a[i+1][j+1]);
    return x;
}
```

这是一个自定义的方法，可以在程序的任意位置来调用它，完整的代码如图 7-3 所示。

上述程序运行后，用鼠标单击窗口，单击处立即出现涟漪，动态的纹理随即出现，如图 7-4 所示。

我们模拟的反应扩散系统对数学公式中的参数非常敏感，如果把公式中的参数稍作改动，产生的图案就会截然不同，如下面这一组参数会产生如图 7-5 所示的效果。

```

int w=200;
int h=160;
float[][] us = new float[h][w];
float[][] vs = new float[h][w];
void setup() {
    size(3*w, 3*h);
    colorMode(HSB);
}
void draw() {
    noStroke();
    for (int i=1; i< h-1; i++) {
        for (int j=1; j< w-1; j++) {
            float u = us[i][j];
            float v = vs[i][j];
            float newu = 0.8*lap(us, i, j) - u*v*v + 0.99*u + 0.015;
            float newv = 0.52*lap(vs, i, j) + u*v*v + 0.935*v;
            us[i][j] = constrain(newu, 0, 1);
            vs[i][j] = constrain(newv, 0, 1);
            fill( 200*us[i][j], 255, 255 );
            rect(j*3, i*3, 3, 3);
        }
    }
}
float lap(float[][] a, int i, int j) {
    float x=-a[i][j]+0.2*(a[i-1][j]+a[i+1][j]+a[i][j-1]+a[i][j+1]);
    x+=0.05*(a[i-1][j-1]+a[i-1][j+1]+a[i+1][j-1]+a[i+1][j+1]);
    return x;
}
void mousePressed() {
    for (int i=mouseY-30; i< mouseY+30; i++)
        for (int j=mouseX-30; j<mouseX+30; j++)
            vs[i/3][j/3]=1;
}

```

图 7-3 模拟两种液体 u、v 之间的转换

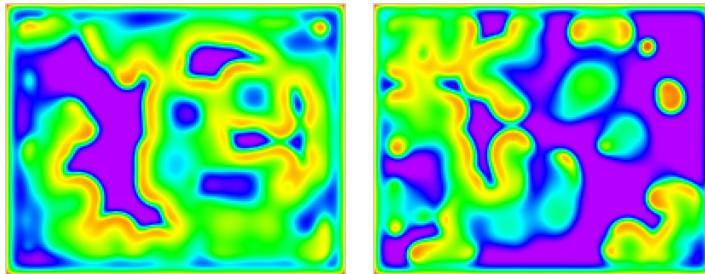


图 7-4 反应扩散系统（一）

```

float newu= 1.0*lap(us, i, j) - u*v*v +0.98*u + 0.021;
float newv= 0.19*lap(vs, i, j) + u*v*v +0.93*v;

```

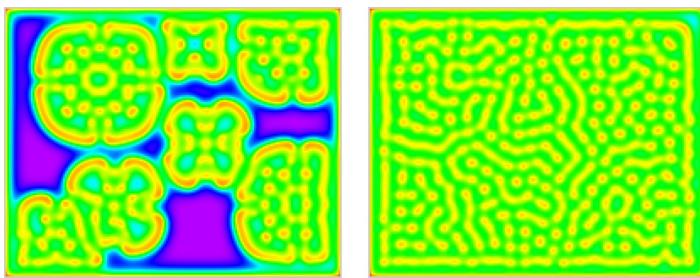


图 7-5 反应扩散系统（二）

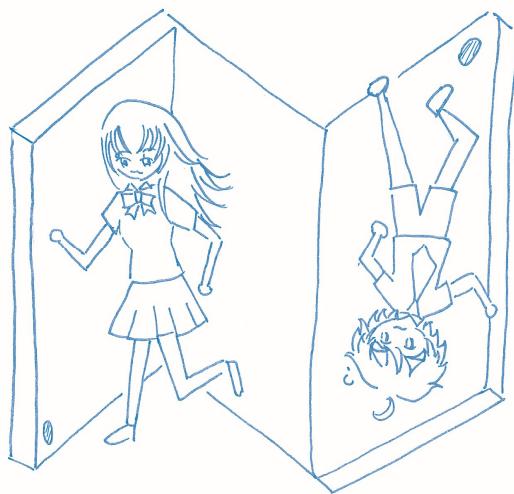
最早用化学来诠释“生成形态”的人可能是艾伦·图灵，他在 1952 年发表了一篇当年很少有人懂的论文《形态发生的化学基础》。而当今的反应扩散系统可以用具体的数学模型和代码来模拟各种化学反应，生成复杂纹理。反应扩散系统还有一个特点，即每个平面点上的参数可以是相互独立的，我们可以在整个窗口内设置不同的参数。譬如，让参数从左至右、从上至下线性地变化，就可以生成如图 7-6 所示的复杂图案。



图 7-6 反应扩散系统产生的复杂有机图案

反应扩散系统与 7.3 节的涟漪程序有很多类似之处。首先，两者都由局部“死板”的数学公式来控制，但在整体上生成了不可预测的“自然”的复杂图像。其次，两者的（局部的）数学公式都与每个点周围的九宫格有关。最后，这两个程序都依赖于一虚一实的两个量之间的互动，而我们只需要观察其中一个量

形成的图形。



## 7.4 生命游戏

2020年4月，英国数学家约翰·康威（John Conway）因为感染了新冠肺炎不幸去世。50多年前，他发明了震惊世界的“生命游戏”（Game of Life）——一种生活在二维网格中的虚拟生命。后来大家把这类神奇的系统称为“细胞自动机”（Cellular Automaton）。

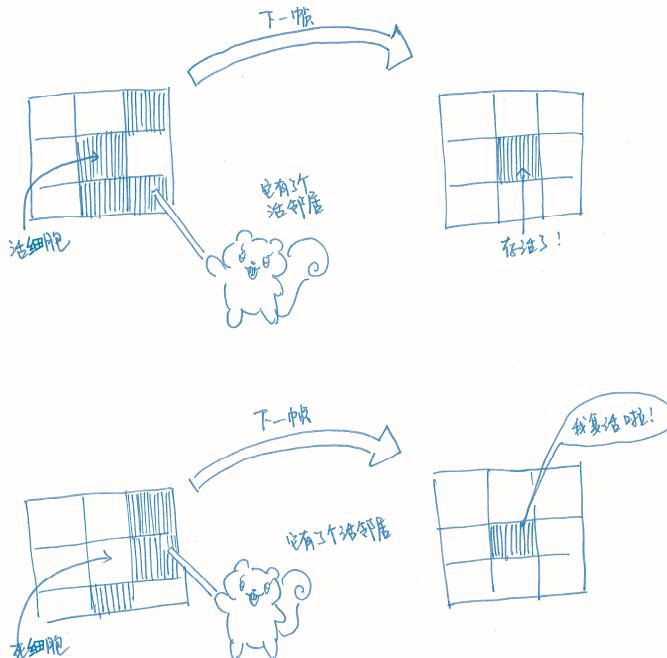


大自然中的各种生命基本上都是由细胞（cell）构成的，而细胞由各种分子构成，分子又由原子构成。那么在计算机的 0、1 世界里，虚拟的生命似乎应该由 0 和 1 构成。康威的细胞自动机试图用简单的 0、1 数据来创造能够不断重生与消亡的“生命”，实现从最简单的元素到复杂生命现象的飞跃。

假设在二维网格中每个格子都有“活”与“死”两个状态，康威用非常简单的规则来激发一种难以预测的生死演化过程。在每帧中，每个格子（细胞）的八个邻居（九宫格除去中央的格子）中活细胞的个数将决定该细胞在下一帧的命运：

如果一个活细胞有 2~3 个活邻居，则下一帧存活，否则死亡。

如果一个死细胞有 3 个活邻居，则下一帧复活。



如果用整数 0 代表死、1 代表活，那么我们可以用一个二维的数组 `int[][] d` 来表示平面中所有格子的生死状态。下面的代码就可以用来统计八个邻居中活细胞的数目：

```
int count = d[i-1][j-1]+ d[i-1][j]+ d[i-1][j+1]+ d[i][j-1]+ d[i][j+1]+d[i+1][j-1]+  
d[i+1][j]+ d[i+1][j+1];
```

决定每个格子命运的代码为：

```
c[i][j]=d[i][j];  
if (0== d[i][j]) { //死格子  
    if (3==count )  
        c[i][j]=1; // 复活  
    } else { //活格子  
        if (1==count || 3<count )  
            c[i][j]=0; // 死亡  
    }
```

完整的代码如图 7-7 所示。

```
int w=360;  
int h=360;  
int[][] d = new int[h][w];  
void setup() {  
    size(2*w, 2*h);  
    int a=166;  
    for (int i=a; i< h-a; i++)  
        for (int j=a; j< w-a; j++)  
            d[i][j]=1;  
}  
void draw() {  
    background(255);  
    noStroke();  
    fill(0);  
    int[][] c = new int[h][w];  
    for (int i=1; i< h -1; i++) {  
        for (int j=1; j< w -1; j++) {  
            int count=d[i-1][j-1]+d[i-1][j]+d[i-1][j+1]+d[i][j-1]  
            + d[i][j+1]+d[i+1][j-1]+d[i+1][j]+d[i+1][j+1];  
            c[i][j]=d[i][j];  
            if (0== d[i][j]) { //dead  
                if (3==count )  
                    c[i][j]=1; // revive  
            } else { //live  
                if (1==count || 3<count )  
                    c[i][j]=0; // die  
                rect(2*j, 2*i, 2,2);  
            }  
        }  
    }  
    d=c;  
}
```

图 7-7 生命游戏代码

某两个时刻的截图如图 7-8 所示。

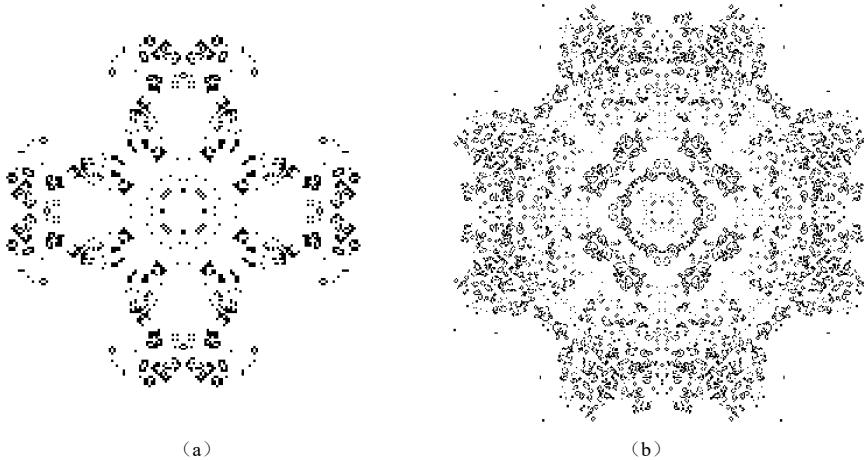


图 7-8 生命游戏（规则 1）运行过程截图

如果在 `setup()` 中设 `int a=69`; 在 `draw()` 中设置生死规则为:

```
c[i][j]=d[i][j];
if (0== d[i][j] ) { //dead
    if (3==count || 4==count )
        c[i][j]=1; // revive
} else { //live
    if (1<count)
        c[i][j]=0; // die
    rect(2*j, 2*i, 2,2);
}
```

就会生成如图 7-9 所示的完全不一样的图案。

“生命游戏”好玩的地方在于，我们可以任意尝试各种生死规则，期待程序生成意想不到的动图。制定规则的基本思路是：不能让细胞过分扩张，否则整个屏幕很快会被活细胞填满；也不能让细胞过快死亡，否则整个屏幕将变得一片空白。

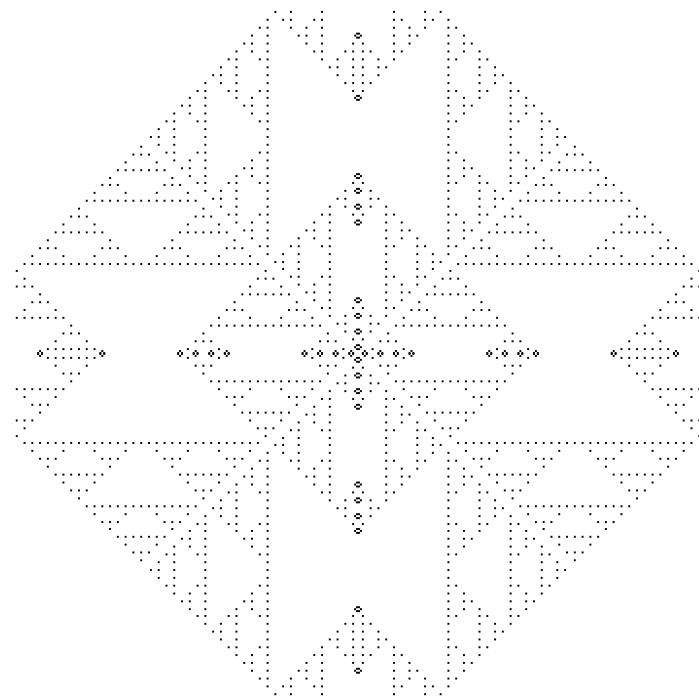
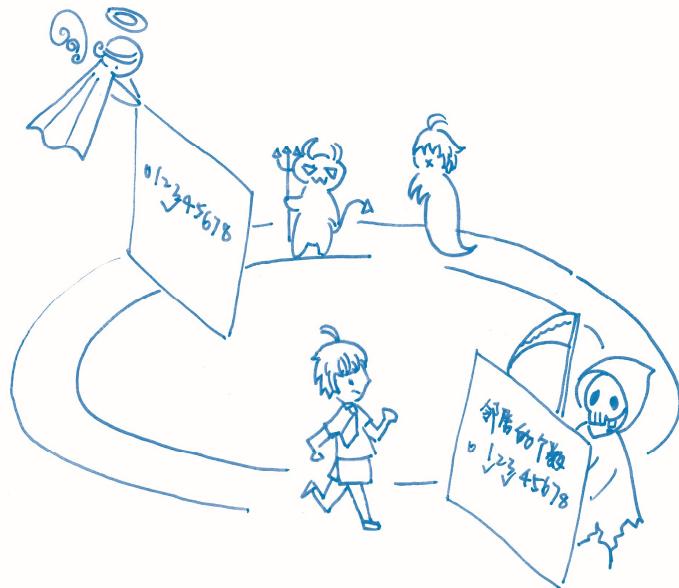


图 7-9 生命游戏（规则 II）

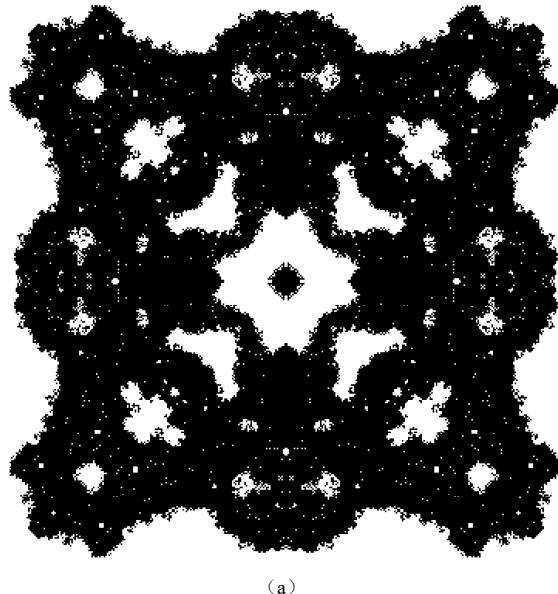


另外一个重要因素是初始化，即在一开始决定哪些细胞是活的。本节的例子都是在屏幕中央设一块充满活细胞的矩形。

最后我们再来尝试另一种新的规则，设 `int a=0;` 生死转换的规则为：

```
c[i][j]=d[i][j];
if (0== d[i][j] ) { //dead
    if (3==count || 6==count || 7==count )
        c[i][j]=1; // revive
} else { //live
    if (3>count || 5==count)
        c[i][j]=0; // die
    rect(2*j, 2*i, 2,2);
}
```

就会生成如图 7-10 所示的复杂图案。



(a)

图 7-10 生命游戏（规则 III）运行过程截图

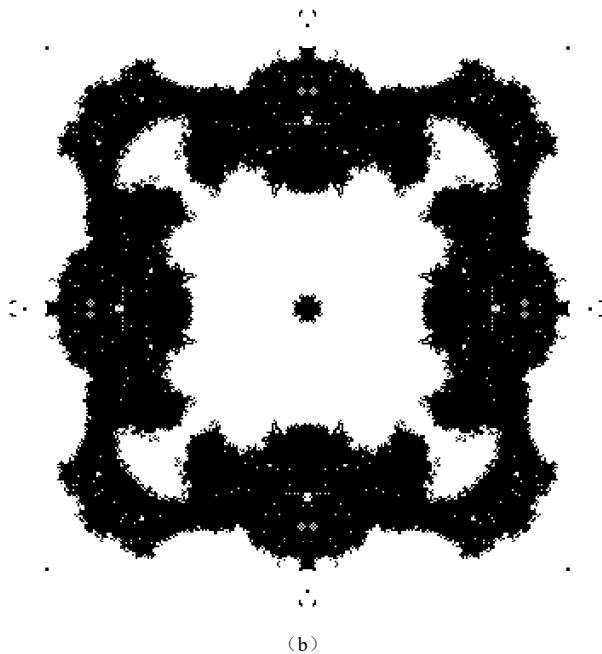
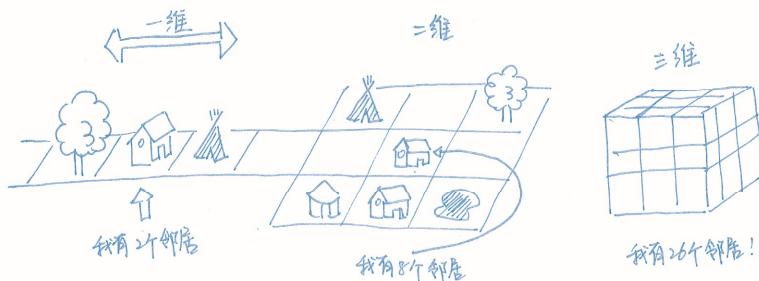


图 7-10 生命游戏（规则Ⅲ）运行过程截图（续）

“生命游戏”是变化多端的，生死规则的不同、初始化的不同都会导致不一样的演化过程。除正交网格之外，在三角形、蜂窝状的网格上也能运行细胞自动机。不仅如此，细胞自动机还可以运行在一维网格（每个细胞只有左右两个邻居）和三维网格上，读者可以自行试验。



## 第8章

# 迭代分形

## 8.1 递归

我们在 4.2 节了解了自定义方法(method)，这次我们要尝试一种神奇的“自己调用自己”的方法，这种特殊情况称为递归(recursion)。究竟什么是递归呢？其实就是在定义方法时调用自己，譬如：

```
void mul(int a){  
    print(a, " ");  
    if(a<1000 )  
        mul(a*2);  
}
```

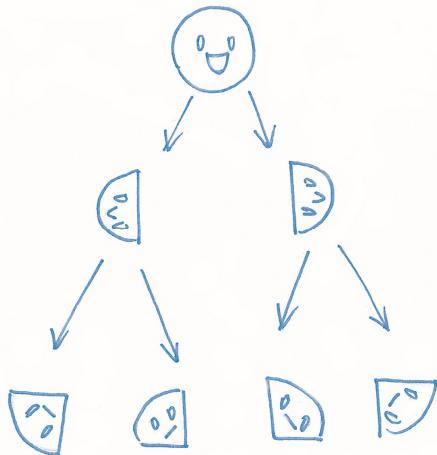
如此定义方法看上去很奇怪，自己还没定义结束呢，就要调用自己了。但 Processing（或者说 Java）确实允许我们这样做。下面的代码是在 `setup()` 中调用这个方法：

```
void setup(){  
    mul(3);  
}
```

输出结果为：

3 6 12 24 48 96 192 384 768 1536

需要提醒的是，递归方法需要一个让递归继续下去的条件，否则就会导致死循环。在上面这个方法中， $a < 1000$  就是继续的条件，当  $a > 1000$  时递归就停止了。



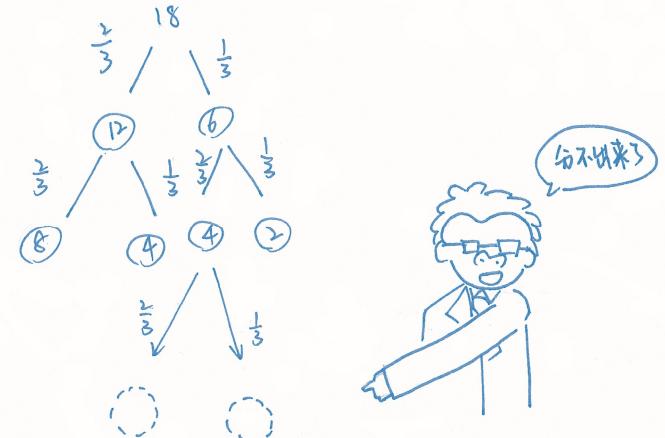
该递归是“1 拖 1”的情况，而递归也可以是“1 拖 2”或“1 拖多”的情况。譬如，下面这个代码就是在内部两次调用自己：

```
void divide(int a){
    print(a, "");
    if( 0!=a && 0==a%3){
        divide( a*2/3);
        divide( a/3);
    }
}
```

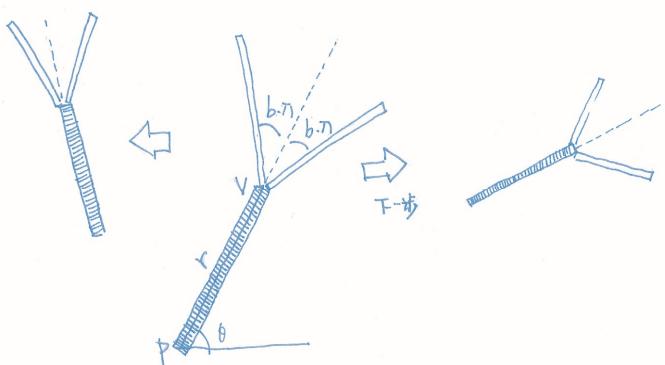
它把一个输入的数字  $a$  分成两份：一份为  $2/3$ ，另一份为  $1/3$ 。譬如，数字 12 就会被拆分成 8 和 4。这个递归过程的继续条件有两个：（1） $a$  不等于零；（2） $a$  可以被 3 整除。运行以下代码：

```
void setup(){
    divide(6);
}
```

就会打印 6、4、2 三个数字。其中，4 和 2 不能再细分，因此递归停止。运行 `divide(18);` 就会生成“18 12 8 4 6 4 2”这一串数字。运行 `divide(81);` 就会生成“81 54 36 24 16 8 12 8 4 18 12 8 4 6 4 2 27 18 12 8 4 6 4 2 9 6 4 2 3 2 1”。



下面我们用递归的方式来模拟树枝分叉，一根枝条一分为二，其中每根枝条再一分为二，如下图所示。



如果当前枝条的倾角为  $\theta$ ，分叉的两根枝条的倾角为  $\theta - b\pi$  和  $\theta + b\pi$ 。此外，分叉出去的枝条长度也会依次按比例递减。写成如下代码：

```
void branch(PVector p, float r, float theta, int age) {
    if (0==age)
        return;
```

```

PVector v= new PVector(r*cos(theta), r*sin(theta));
v.add(p);
line(p.x, p.y, v.x, v.y);
line(w-p.x, p.y, w-v.x, v.y);
branch(v, a*r, theta + b*PI, age-1 );
branch(v, a*r, theta - b*PI, age-1 );
}

```

该方法的四个参数分别为：枝条的起点 `p`（关于 `Pvector` 的知识可参见第 5 章）；长度 `r`；倾角及年龄 `age`。该方法用参数 `age` 来控制递归的深度，其中

```

if (0>age)
return;

```

为终止语句。其中，`return` 关键字直接终止当前 `branch()` 方法的运行。需要注意的是，在 `branch()` 内部调用 `branch()` 的时候，需要用 `age-1` 指定下一代的年龄（比上一代减 1）。绘图命令 `line(p.x,p.y, v.x, v.y);` 画了一根线段，代表当前的枝条；而 `line(w-p.x,p.y, w-v.x, v.y);` 绘制了一根与水平枝条对称的线段，完整的程序如图 8-1 所示。

```

int w=1000;
int h=800;
float a=0.5;
float b=0.3;
void setup() {
  size(w, h);
}
void draw() {
  background(255);
  branch(new PVector(w/2, h/2), 160, 0, 9);
}
void branch(PVector p, float r, float theta, int age) {
  if (0>age)
    return;
  PVector v= new PVector(r*cos(theta), r*sin(theta));
  v.add(p);
  line(p.x, p.y, v.x, v.y);
  line(w-p.x, p.y, w-v.x, v.y);
  branch(v, a*r, theta + b*PI, age-1 );
  branch(v, a*r, theta - b*PI, age-1 );
}
void mouseMoved() {
  a = map(mouseX, 0, w, 0.4, 0.8);
  b = map(mouseY, 0, h, 0, 1);
}

```

图 8-1 用 `branch` 递归方法生成树状图案的完整代码

鼠标可以控制分叉的角度和长度参数，生成如图 8-2 所示的对称树形。

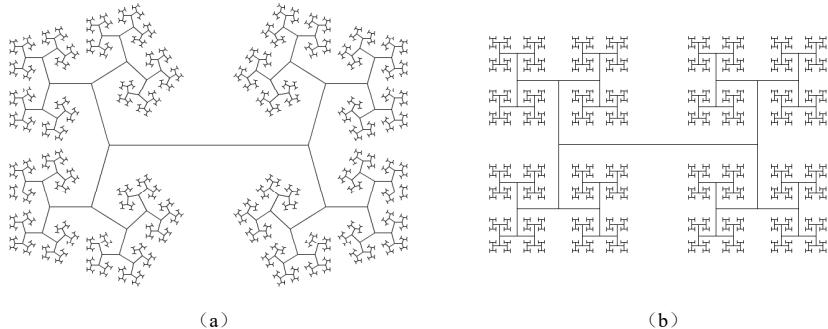


图 8-2 递归方法生成的两种树状图案（参数不同）

也可以产生如图 8-3 所示的奇怪形状。

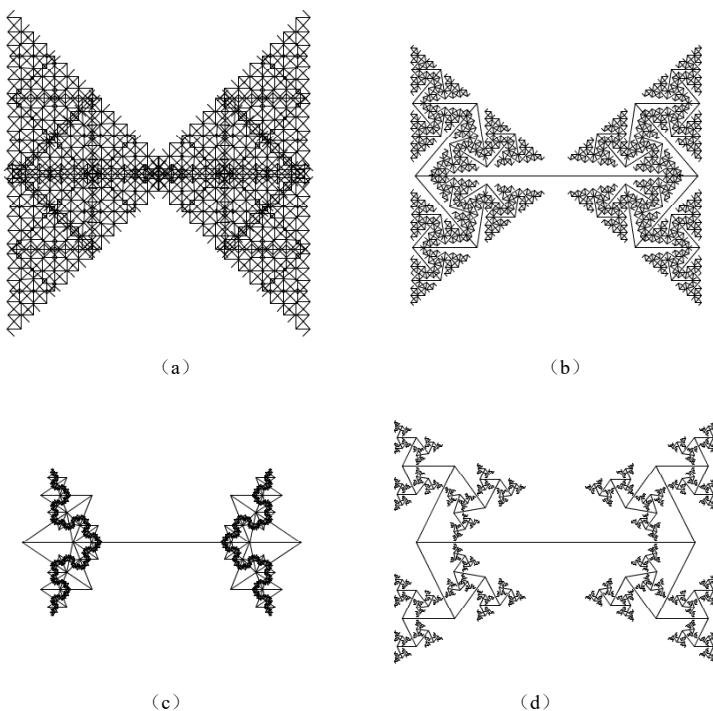


图 8-3 不同参数设定产生更多图案

这个例子很好地诠释了“参数变化导致形状变化”。我们在构思一个程序时，往往不是在设计一个特定的形状，而是要思考一系列相互关联且可以连续变化的形状。而变量、自定义方法可以很好地帮助我们实现图形的变化。

练习题：

本节的程序在绘制枝条时，线宽都是默认值 1。尝试让线宽 `strokeWeight()` 与参数 `age` 相关联，从而形成“老枝条粗、新枝条细”的效果。



## 8.2 多重画布

世界上很多事物都具有自相似性，如地球绕着太阳转，而月亮又绕着地球转。树也具有明显的自相似性，8.1 节介绍的递归方法可以很自然地生成自相似的结构。实际上，抽象的数字也具有自相似性，如著名的斐波那契数列(Fibonacci sequence)，即下一个数字等于前两个数之和，该数列从 0、1 这两个数字开始。

用递归来实现该数列，程序如下：

```
void setup() {
    println("golden ", 0.5*(sqrt(5)+1));
    add(0, 1);
}

void add(int a, int b) {
    println(a, "/", b, "=", (float)a/b);
    if (b<100)
        add(b, a+b);
}
```

golden 1.618034

$0 / 1 = 0.0$

$1 / 1 = 1.0$

$1 / 2 = 0.5$

$2 / 3 = 0.6666667$

$3 / 5 = 0.6$

$5 / 8 = 0.625$

$8 / 13 = 0.61538464$

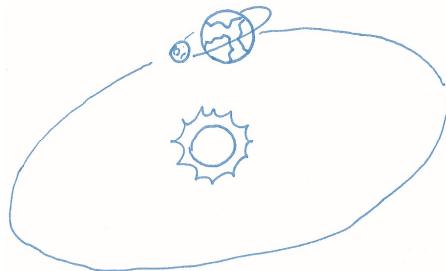
$13 / 21 = 0.61904764$

$21 / 34 = 0.61764705$

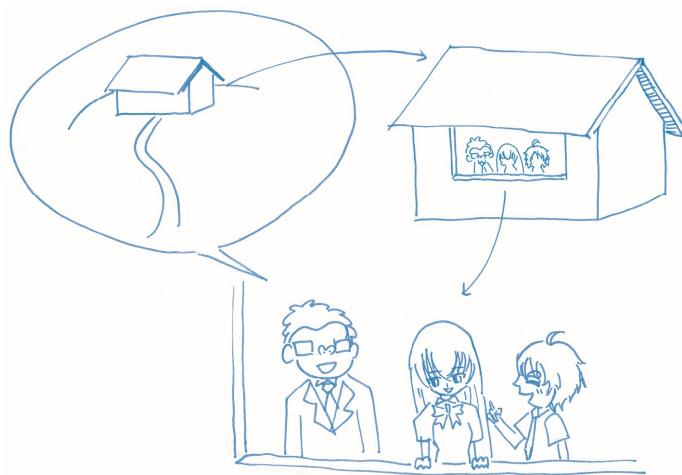
$34 / 55 = 0.6181818$

$55 / 89 = 0.6179775$

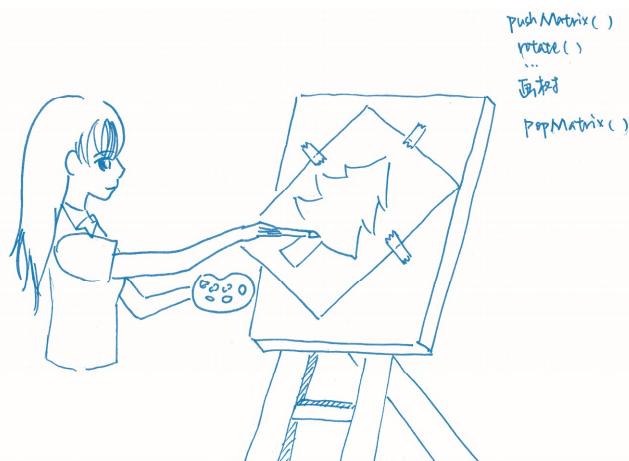
$89 / 144 = 0.6180556$



不难发现，相邻两个数字的比例慢慢趋近于黄金分割比例。

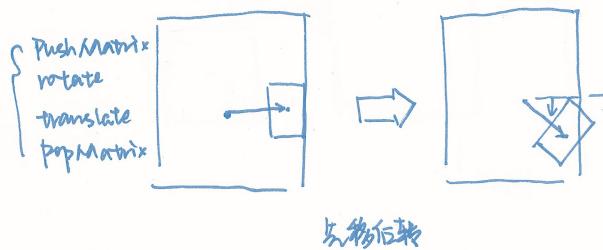
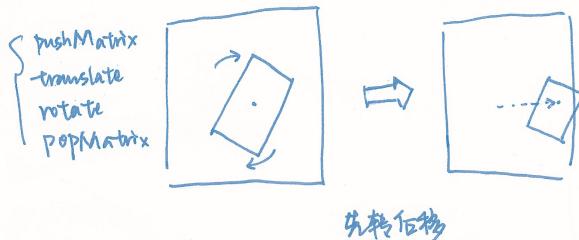


下面这个例子将把递归和画布结合起来。Processing 中的 `pushMatrix()` 与 `popMatrix()` 会临时创建一个虚拟画布，如下图所示。



上图中 `rotate()` 命令后面的绘图都在旋转过后的画布上。然而，`pushMatrix()` 之前、`popMatrix()` 之后的绘图是不受影响的。因此，`pushMatrix()` 与 `popMatrix()` 的主要作用就是把旋转、平移、缩放等画布操作限制在代码的某个局部，这样就有可能在同一个程序中创建多个互不干涉的画布。

当多个旋转、平移、缩放命令在代码中先后出现时，通常包裹在 `pushMatrix()`、`popMatrix()` 之内，效果是从下向上依次实现的（与直观相反），如下图所示。



换句话说，旋转、平移等画布操作命令的先后顺序不同，则效果不同。下面的程序是“一拖三”画方块的递归方法：

```
void square(float x, float y, float w, int age){
    if (0 > age)
        return;
    fill(age * 20, 255, 255, 255 - age * 25);
    pushMatrix();
    translate(x, y);
    rect(0, 0, w, w);
    rotate(theta * PI);
    square(-w / 2, w / 2, w * s, age - 1);
    square(w / 2, -w / 2, w * s, age - 1);
    square(w / 2, w / 2, w * s, age - 1);
    popMatrix();
}
```

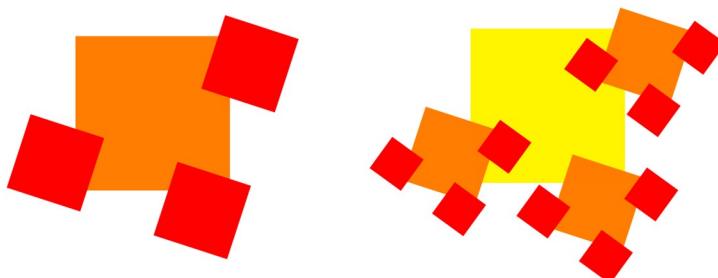
它先把当前的画布平移到(x, y)位置画方块，而内部的三次 square()调用将

在一块先旋转  $\text{theta} * \text{PI}$  后平移(x, y)的画布上进行。

运行下列主程序：

```
float theta=0.1;
float s=0.5;
void setup() {
    size(1000, 800);
    rectMode(CENTER);
    colorMode(HSB);
}
void draw() {
    background(255);
    noStroke();
    square(500, 400, 256, 1);
}
```

我们能看到三个红色的方块整体被旋转了，如图 8-4 (a) 所示，可以想象它们在一块旋转过后的透明画布上。如果把迭代深度增加，即 `square(500, 400, 256, 2);` 就会得到如图 8-4 (b) 所示的图案，其中三个红方块和橙色方块一组，该组的形状等同于三个橙色方块和中间黄色方块构成的形状(体现出自相似性)。整个图中有四个生成的画布：一个在黄色方块中间（旋转了  $\text{theta} * \text{PI}$ ），三个分别在橙色方块中间（在原来基础上又旋转了  $\text{theta} * \text{PI}$ ）。



(a) 两种颜色的方块绘制在两层画布上

(b) 三种颜色的方块绘制在三层画布上

图 8-4 “一拖三”画方块的递归方法

下面我们把递归的深度增加到 8，程序如图 8-5 所示。

```

float theta=0.37;
float s=0.52;
void setup() {
    size(1000, 800);
    rectMode(CENTER);
    colorMode(HSB);
}
void draw() {
    background(255);
    noStroke();
    square(500, 400, 256, 8);
}
void square(float x, float y, float w, int age) {
    if (0>age)
        return;
    fill(age*20, 255, 255, 255-age*25);
    pushMatrix();
    translate(x, y);
    rect(0, 0, w, w);
    rotate(theta*PI);
    square(-w/2, w/2, w*s, age-1);
    square(w/2, -w/2, w*s, age-1);
    square(w/2, w/2, w*s, age-1);
    popMatrix();
}

```

图 8-5 square 递归方法的完整代码

运行图 8-5 所示程序，就生成了如图 8-6 所示的复杂图案。

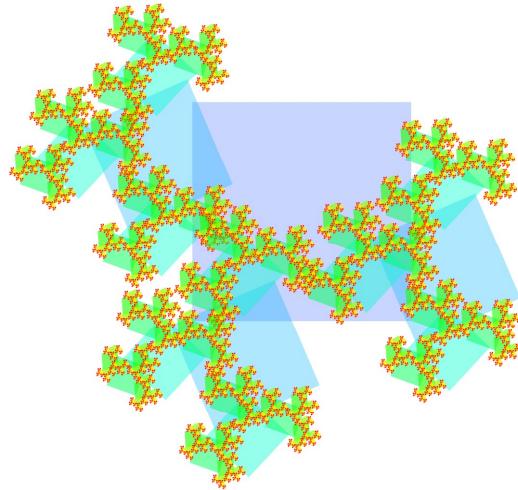


图 8-6 递归方块分形图案 (theta=0.37、s=0.52)

中间方块的年龄最大为 8，末端红色方块的年龄最小为 0。这个图形对参数也很敏感，当参数为  $\text{theta}=0$ 、 $s=0.5$  时，图形如图 8-7 所示。

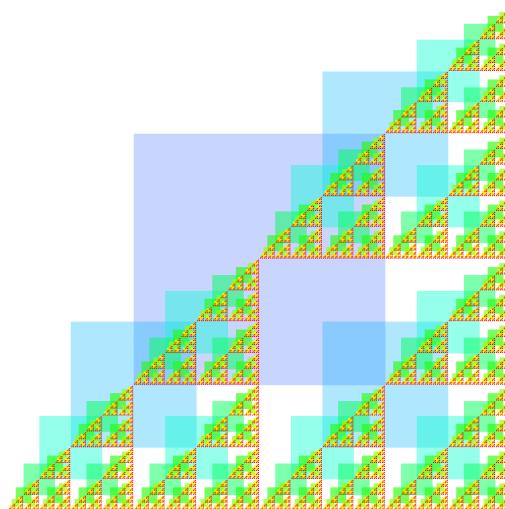


图 8-7 递归方块分形图案 (  $\text{theta}=0$ 、 $s=0.5$  )

当参数为  $\text{theta}=0.5$ 、 $s=0.5$  时，图形如图 8-8 所示。

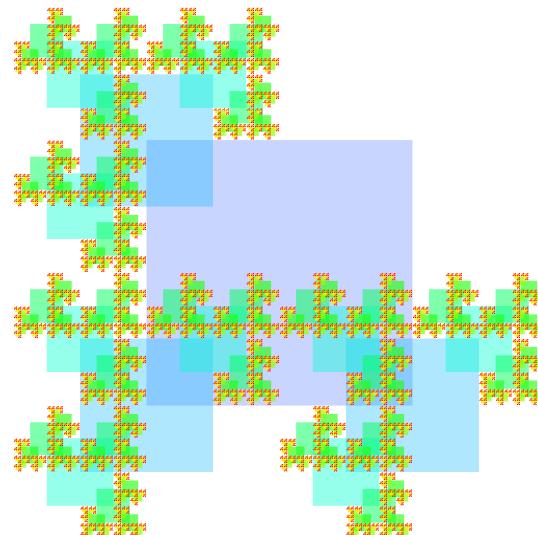


图 8-8 递归方块分形图案 (  $\text{theta}=0.5$ ， $s=0.5$  )

递归的反复循环造成了画布的多重叠加，可以产生繁复的图形。这个过程有点像反复使用复印机，如把一个印刷在透明 A2 纸上的图形缩小复印在两张透明 A3 纸上，再将每张 A3 纸上的图形再缩小复印两张 A4 纸上……

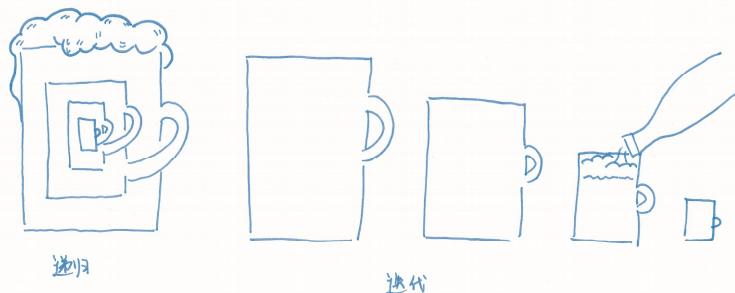


## 8.3 悲情朱利亚

递归（recursion）是一种非常特殊的迭代（iteration）方式。普通的迭代方式包括 `for` 循环、`while` 语句、`do-while` 结构等。其实递归都可以改写成普通的迭代，如 8.2 节的斐波那契数列的代码，就很容易用 `while` 语句来写：

```
int a=0;
int b=1;
while (a<100) {
    println(a, "/", b, "=", (float)a/b);
    int sum=a+b;
    a=b;
    b=sum;
}
```

其中，`a<100` 是该循环继续下去的条件。因此，当 `a>100` 时，该循环就终止了。



把“一拖二”或“一拖多”的递归改写成一般迭代，就稍微有点复杂了。譬如，我们把 8.1 节中的 divide 递归改写成 do-while 的形式，代码如下：

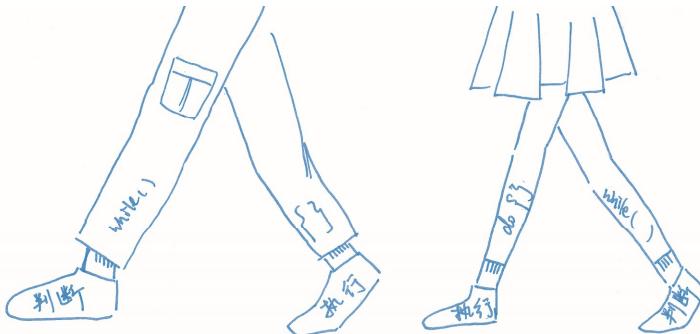
```
ArrayList<Integer> list=new ArrayList();
list.add(9);
do {
    ArrayList<Integer> tmp=new ArrayList();
    for (int a : list) {
        print(a, " ");
        if (0!=a && 0==a%3) {
            tmp.add(a*2/3);
            tmp.add(a/3);
        }
    }
    list=tmp;
    println(" end");
}
while (list.size ()>0);
```

第一轮循环把数字 9 拆分成 6 和 3，而 tmp 列表保存了 6 和 3 两个数字。

第二轮循环把 6 分成 4 和 2，把 3 分成 3 和 1；而 tmp 列表保存了 4、2、2、1 这四个数字。

第三轮循环遇到的 4、2、2、1 这四个数字，都不满足  $0!=a \&& 0==a \% 3$  条件，因此 `list.size ()>0` 条件不成立，所以循环终止了。

`while` 语句与 `do-while` 语句稍有区别：`while` 语句在每次循环中先判断（是否满足执行条件）后执行（花括号内的代码），而 `do-while` 语句先执行再判断是否执行下一轮循环。

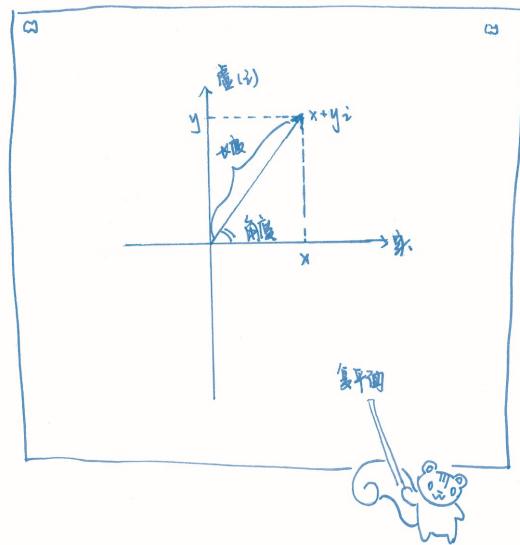


从 8.1 节和 8.2 节可以看出，递归可以很方便地生成自相似（self-similar）的图形。实际上，同一段代码的反复运行就可以自然地对应于几何形状的自相似性。而本节将用普通的 `for` 循环来生成一种奇妙的自相似图形——复数分形。

复数分形的想法最早出现在一位法国数学家的脑海中。朱利亚(Gaston Julia)的在年轻时卷入了第一次世界大战，甚至不幸地失去了自己的鼻子，但在战争结束时他发明了一种神秘的运算，代码如下：

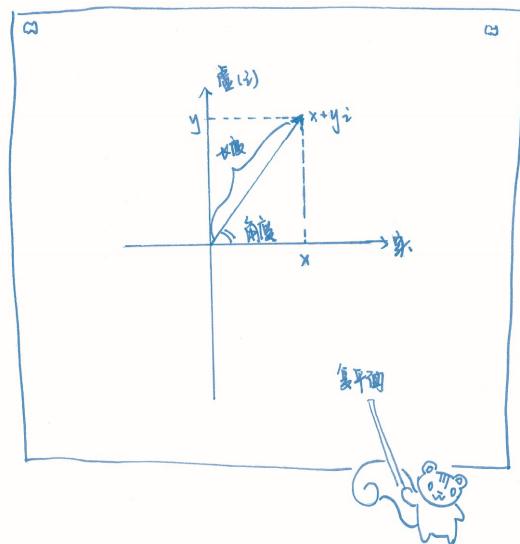
$$z=z^*z+c$$

即在变量  $z$  的平方上加上常数  $c$ ，再把得到的值赋给  $z$ 。这里  $z$  和  $c$  都是复数（complex number），我们可以把复数想象成平面中的一个点（与二维向量相似），一般形式为  $z=x+yi$ 。式中， $x$  称为实部； $y$  称为虚部； $i$  表示虚数单位。



运用勾股定理，计算出复数  $x+yi$  的长度为：

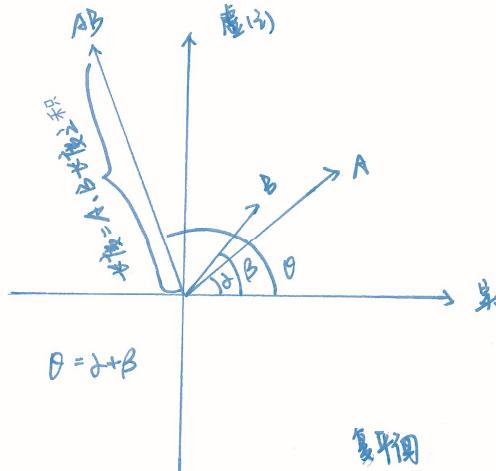
$$\sqrt{x^2 + y^2}$$



两个复数相加的运算十分简单，把它们的实部与虚部分别相加即可。而两个复数的乘法比较特殊，假设有复数  $A = x+yi$ 、 $B = a+bi$ ，那么它们的积  $AB$  可以如下这样计算：

$$(x+yi) * (a+bi) = ax + ay i + bxi + byii = ax - by + (ay + bx)i$$

这里用到了“ $i^2=-1$ ”这个等式。有趣的是，这个乘法可以用下图来解释：AB 的角度等于 A 和 B 的角度之和，而 AB 的长度是 A 和 B 的长度之积。



假设有两个复数  $z=x+yi$  和  $c=-0.6+0.4i$ ，那么朱利亚的  $z=z^*z+c$  算法在 Processing 中可以如下这样实现：

```
float nx = x * x - y * y - 0.6;
float ny = 2 * x * y + 0.4;
```

需要输入的两个复数为  $z=x+yi$  和  $c=-0.6+0.4i$ ，运算后得到新的复数为  $z=nx+nyi$ 。在 4.2 节处理屏幕像素时，我们曾经用 `colorAt()` 方法为每个像素填色。现在我们把朱利亚的这种复数运算写到 `colorAt()` 方法中，代码如图 8-9 所示。

`colorAt()` 方法中的 `for` 循环把  $z=z^*z+c$  反复运行了 7 次，然后计算复数  $z$  的长度的平方 ( $x*x+y*y$ )，最后用 `color(150*v, 255, 255)` 这个值来确定像素的颜色。如果值太大（超过 255）就把白色赋给像素 `color(255)`。

程序运行的结果如图 8-10 所示，该类图像被称为朱利亚集合（Julia set）。可惜的是，朱利亚那个年代还没有计算机，而手算这样复杂的图像几乎不可能，该图形一直到 1978 年才第一次在计算机上绘制出来。

```
int w=900;
int h=700;
void setup() {
    size(w, h); //size(900,700)
    colorMode(HSB);
}
void draw() {
    loadPixels();
    for (int i=0; i<h; i++) {
        for (int j=0; j<w; j++) {
            pixels[i*w + j] = colorAt(i, j);
        }
        updatePixels();
    }
}
color colorAt(int i, int j) {
    float x= 2*(j-w*0.5)/h;
    float y= 2*(i-h*0.5)/h;
    for (int n = 0; n < 7; n++) {
        float nx = x * x -y * y - 0.6;
        float ny = 2 * x * y + 0.4;
        x = nx;
        y = ny;
    }
    float v=x*x+y*y;
    if (150*v>255)
        return color(255);
    else
        return color(150*v, 255, 255);
}
```

图 8-9 创建朱利亚集合的完整代码

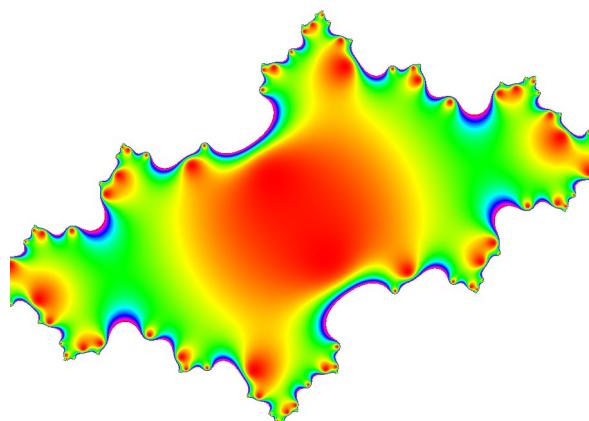


图 8-10 朱利亚集合的彩色显示

之前的程序采用了一个固定的  $c$  值，我们可以用鼠标的实时位置来产生一个不断变化的  $c$ （复数形式为  $a+bi$ ），代码如图 8-11 所示，这样就能生成动态的朱利亚图像了。

```

int w=900;
int h=700;
void setup() {
    size(w, h); //size(900,700)
    colorMode(HSB);
}
void draw() {
    float a = map(mouseX, 0, h, -1, 1);
    float b = map(mouseY, 0, h, -1, 1);
    loadPixels();
    for (int i=0; i<h; i++)
        for (int j=0; j<w; j++)
            pixels[i*w + j] = colorAt(i, j, a, b);
    updatePixels();
}
color colorAt(int i, int j, float a, float b) {
    float x= 2.4*(j-w*0.5)/h;
    float y= 2.4*(i-h*0.5)/h;
    for (int n = 0; n < 7; n++) {
        float nx = x * x-y * y + a;
        float ny = 2 * x * y + b;
        x = nx;
        y = ny;
    }
    float v=x*x+y*y;
    return 150*v>255?color(255):color(150*v, 255, 255);
}

```

图 8-11 用鼠标位置来控制朱利亚集合的参数

图 8-11 中程序最后一句的表达式 `return 150*v>255 ? color(255):color(150*v, 255, 255);` 是 `if-else` 语句的一种简化写法，它等价于以下代码：

```

if (150*v>255)
    return color(255);
else
    return color(150*v, 255, 255);

```

利用类似的语法，将最后一句改写成 `return color(x*y<0?255:0);`，这样就能生成如图 8-12 所示的黑白图形了。

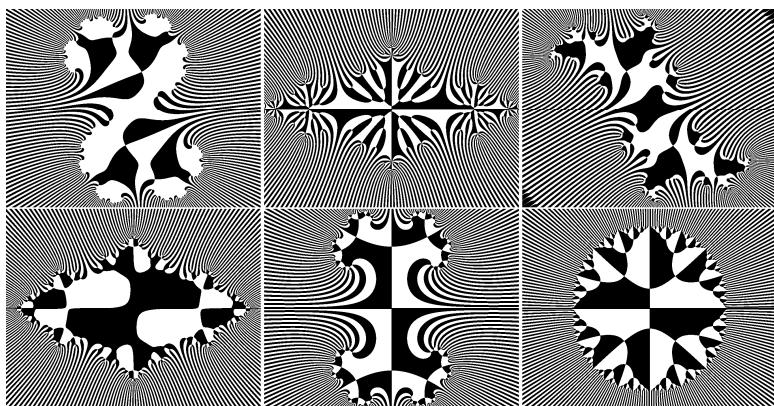


图 8-12 不同参数设定下的黑白朱利亚集合图像

在没有计算机的年代，像朱利亚这样的天才数学家也无法清晰地展现复数分形的全貌。到了 20 世纪七八十年代计算机逐渐普及的时候，用代码来编写这样的图形依然很困难；但有了 Processing 一切都变得简单快捷了。

