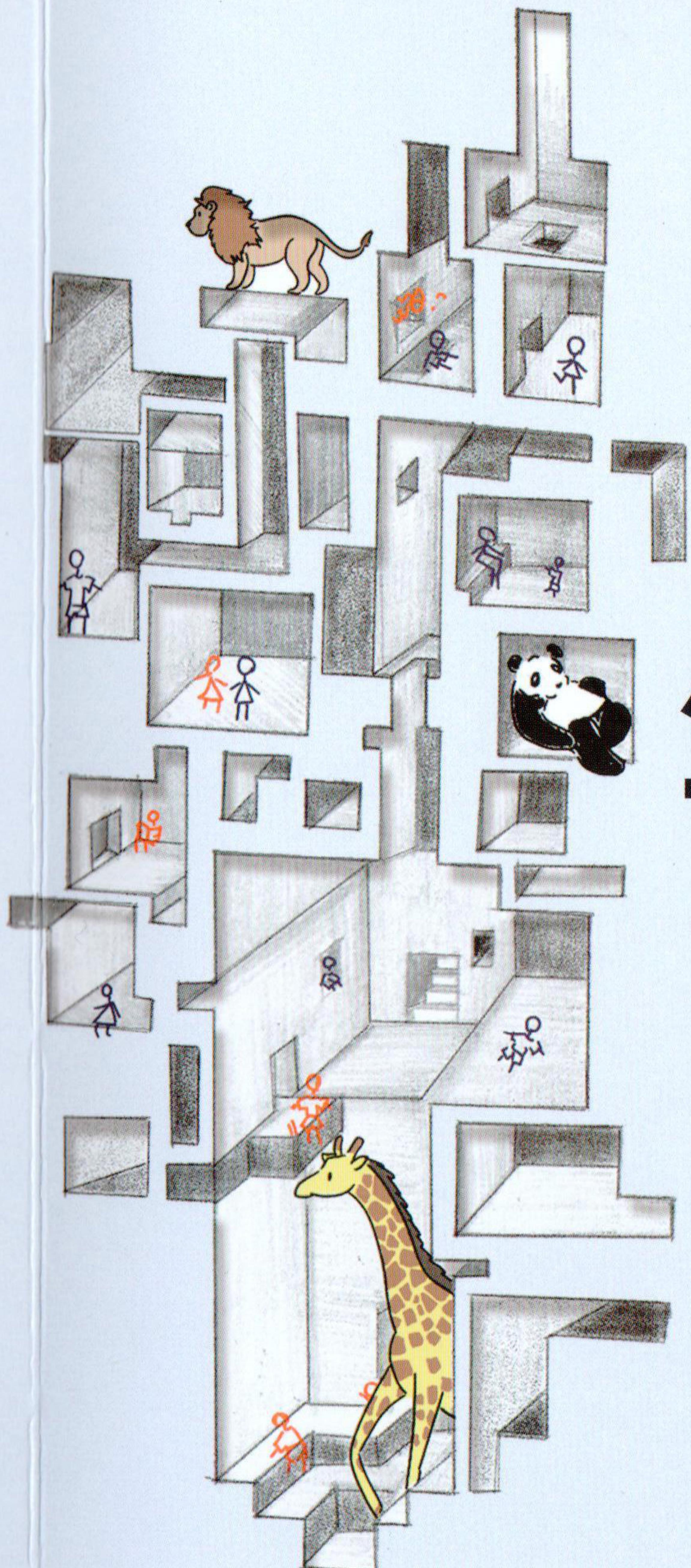


GENERATIVE ART

INTRODUCTION TO GRAPHICS
PROGRAMMING WITH PROCESSING



生成艺术

Processing
视觉创意入门

华 好◎著
刘晨晰◎绘



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>



基础部分

第1章 画布、画笔和颜色	002
1.1 第一个程序	002
1.2 画布与帧	007
1.3 颜色	013
第2章 变化多端的圆形	020
2.1 变量与循环	020
2.2 心	026
2.3 圆的魔术	032
第3章 弹！弹！弹！	040
3.1 弹球	040
3.2 布尔先生	046
3.3 好多弹球	053
第4章 弹！弹！弹！	060
4.1 彩色噪声	060
4.2 自定义方法	065
4.3 滤镜与点彩	069
4.4 图像重绘	077

第 5 章 PVector	083
5.1 类	083
5.2 线性代数	089
5.3 力	095
5.4 线性插值	100
第 6 章 飘	107
6.1 回旋	107
6.2 秩序与随机	112
6.3 奇怪吸引子	122
第 7 章 一石激起千层浪	131
7.1 二维数组	131
7.2 涟漪	137
7.3 化学反应	141
7.4 生命游戏	146
第 8 章 迭代分形	153
8.1 递归	153
8.2 多重画布	158
8.3 悲情朱利亚	165

基础部分





画布、画笔和颜色

1.1 第一个程序

你是否曾经为画画之前要准备一大堆画笔、颜料而烦恼？一旦下载了 Processing 这个“神器”，你就立刻免去了这些麻烦，因为 Processing 开发环境已经在计算机上为你准备好了画布、画笔和所有颜色。

还可以做动画哦！

Processing 是由麻省理工学院的 Ben Fry 和 Casey Reas 为艺术家、设计师、建筑师、学生等开发的图形化编程环境，它使全球掀起了一股编写创意编码的热潮。Processing 诞生于 2001 年，今年已经 20 周岁了。

Processing 是一款免费的开源软件，在 Windows 和 Mac 下运行的效果完全一致。Processing 已经有好几个版本了，但大家还是觉得 2.2 版本用起来更顺手。下载安装完成后打开 Processing，在文本编辑器内迅速写下 6 行神秘的代码：

```
void setup (){  
    size (800,600);  
}  
void draw (){  
    line (400,300, mouseX, mouseY);  
}
```

注意这些字符的不同颜色。如果有些英文单词没有变成彩色，可能是因为拼写错误。代码里面的标点符号是必须的，字母的大小写也不能搞错哦！

Processing 工具栏左侧有两个按钮酷似老式录音机上的播放键（run）和停止键（stop）。单击播放键，你的代码就会开始运行，这时有一个展示窗口（画布）弹出来，你的鼠标就能不断在画布上画直线了，如图 1-1 所示。

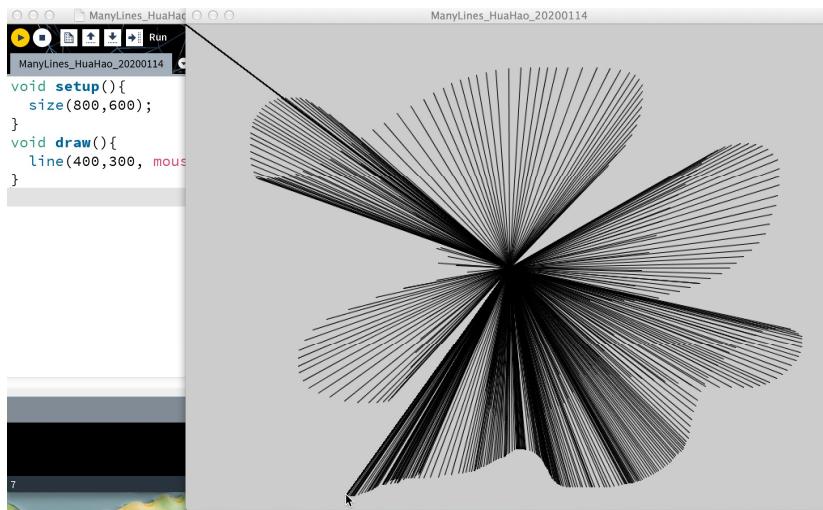


图 1-1 鼠标不断在画布上画直线



平复一下激动的心情，现在可以按停止键关闭程序了。单击“文件/保持”保存代码，给文件（.pde 格式）取一个有含义的名字吧，譬如：

ManyLines_HuaHao_20200114.pde

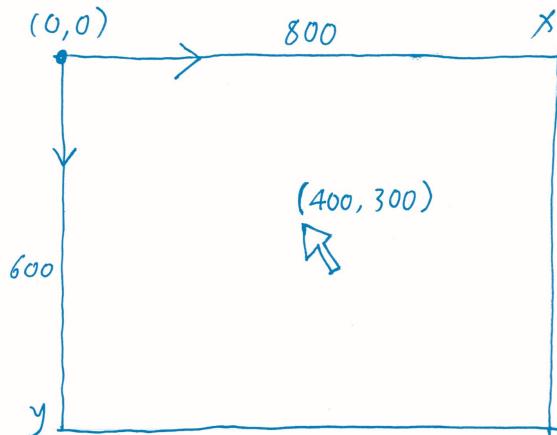
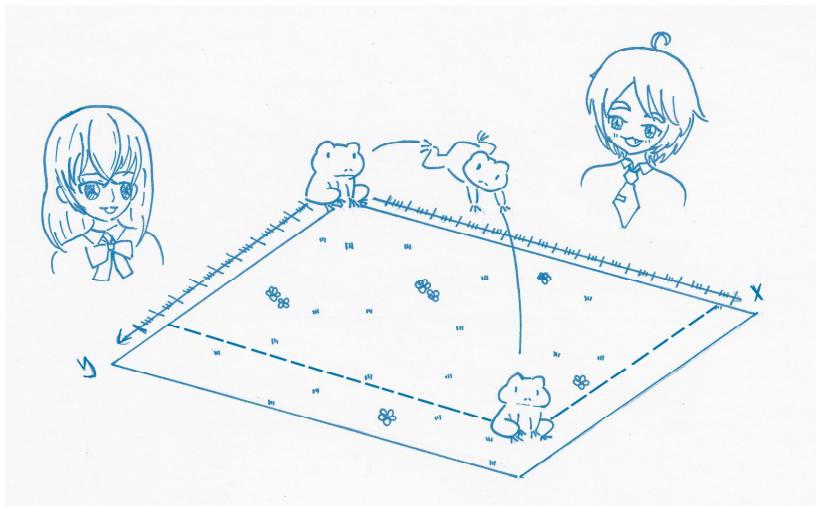
这个名字明确了三点：①程序内容、②作者姓名、③创作日期(yyyymmdd)。文件的命名十分重要！想象一下你已经写了上百个程序，如果这些文件都没有一个有含义的名字，你很难找到那个你想要的文件。当你和他人分享代码的时候，为文件取名也是一种基本的礼仪。此外，建议你在计算机里建一个文件夹，专门存放你的程序。



回到那 6 行代码，我们暂且无视 `void`、`setup`、`draw` 这些词，只看和图形有关的代码：

```
size(800,600); //长, 宽  
line(400,300, mouseX, mouseY); //鼠标位置
```

你可以试着在代码中改变展示窗口的大小和直线起点的位置，看结果是否和你想象的一样。譬如，把画线的那行代码改成 `line(400,300, mouseX, mouseY)`。你也许会惊讶地发现屏幕的原点在左上角，而一般数学书里的坐标原点在左下角。但这不影响我们理解“坐标”意思，假如有一只青蛙从原点起跳，坐到了草地上，我们就能标出它的位置了：



用两个数来表示一个点在平面内的位置，是一项古老的发明，传说是 400 多年前法国大叔勒内 · 笛卡儿 (René Descartes) 在床上睡懒觉时不小心发明的。这个故事告诉我们，在睡觉时也不能停止发明与创造。笛卡儿坐标系把图形与数字，或变化的数字，在程序中用“变量” (variable) 来表示，联系起来，这是所有数字化视觉艺术的基石。Processing 中的笛卡儿坐标系可以是 2 维的或 3 维的，本书专注于 2 维坐标中的图形艺术。

有时，你会发现程序莫名其妙地无法运行了。譬如，把 `mouseX` 这个关键词写成 `mousex`，运行程序的时候文本编辑器下方的消息区域（会变成深褐色）

内会出现一行白色的字：

```
Cannot find anything name "mousex"
```

意思是无法找到 mousex。因为 Processing（Java 语言）是一种严格区分子母大小写的语言，因此 **mouseX** 和 **mousex** 是截然不同的。Processing 内部已经定义了很多关键词，如 **mouseX** 指鼠标在显示窗口中的 X 坐标，这些关键字会自动变成彩色。常见的关键字还有：

Width——展示窗口的宽度；

Height——展示窗口的高度；

frameCount——整数，当前帧的序号。

注意代码的颜色可以有助于较少错误。尽量不要在代码里混入中文字符，否则容易出现一些难以发现的错误。

大部分编程语言允许无用的空行、空格夹杂在代码里，Java 编程语言也一样。在代码任意一处增加一个空行，不会对程序的运行产生任何影响。在各关键词、数字前后增加空行也没有任何问题。推荐大家经常用菜单栏的 Edit/Auto Format 命令，使代码整齐划一，去掉那些冗余的空格。

为了使代码的可读性更强，我们可以用“//”符号添加注释。程序会忽略“//”后面的内容，而且颜色会变灰。下面我们给第一个程序添加注释：

```
void setup(){ //在最开始时运行一次
    size(800,600); //窗口的大小
} // setup 方法到此结束
void draw(){ //此方法每秒运行 60 次
    line(400,300, mouseX, mouseY);
    // 画从(400,300) 到 (mouseX, mouseY)的直线
} // draw 方法到此结束
```

1.2 画布与帧

我们的第一个程序由两个方法 (method) 组成: `setup()`方法和 `draw()`方法。有时人们通俗地把方法称为函数或命令。我们想让程序做事, 就调用 Processing 自定义的方法 (名字会自动变成蓝色粗体) 或者自己定义的方法 (将在第 4 章介绍)。

现在我们在 `line(400,300, mouseX, mouseY);` 上面增添一行代码:

```
void draw(){
    background(255);
    line(400,300, mouseX, mouseY);
}
```

再次按下播放键, 鼠标在展示窗口内移动时的效果和之前不同了, 只有一根线跟随鼠标移动 (见图 1-2)。这是怎么回事呢? 实际上, 程序中的 `draw()` 方法每秒钟会被调用 60 次左右, 因此我们的程序会随着鼠标不断地画下直线。而新增的 `background(255)` 方法先把整个画布 (展示窗口) 刷成了白色, 然后再画一根直线, 因此直线不会随时间叠加在画布上, 如图 1-2 所示。

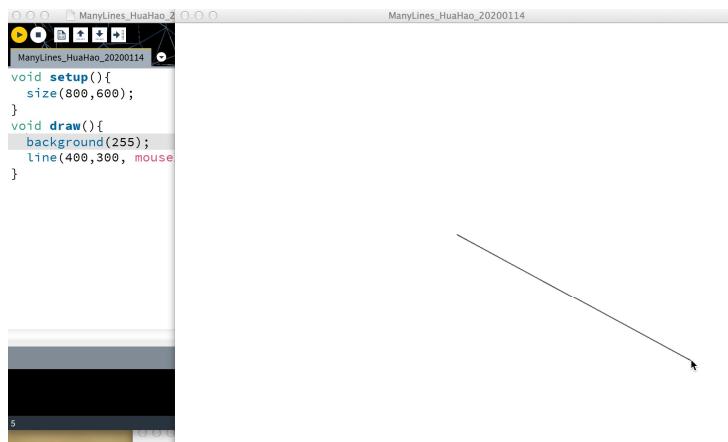


图 1-2 线段一端固定在窗口中央, 另一端为鼠标位置

1.1 节的程序 (线在画布上叠加) 与本节的画线程序 (线不叠加) 代表了图

图模式和动画模式两种动态模式，除此之外，静态模式（不含 `draw` 方法）也很常用。

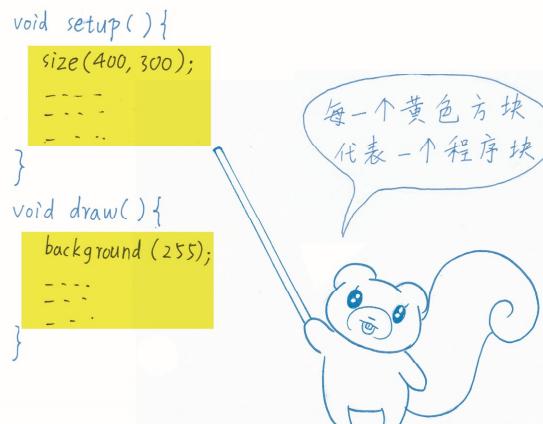


画画模式（不含 `background` 方法）



动画模式（使用 `background` 方法）

Processing 程序以 `setup()` 方法启动，然后不断运行 `draw()` 方法。所以我们把“只需在开头运行一次”的代码放到 `setup()` 程序块里，把“需要反复运行的”的代码放到 `draw()` 程序块里。在每个程序块的内部，代码会一行一行依次运行。



以前美国有个做铁路生意发家的土豪，他和人打赌说马奔跑的某个瞬间四只蹄子都不着地。马跑得太快，人眼无法分辨，于是他花钱聘了一位摄影师

(Eadweard Muybridge) 来拍摄马飞奔过程中的所有瞬间。这个倒霉的摄影师花了好几年才成功, 他的胶片放映机每秒会依次投射 24 张左右胶片 **#特别浪费啊!** 而人眼感受到的是马飞奔的连贯动作。每一张胶片称为一帧 (frame), 其中的某一帧显示: 马的四只蹄子都离开了地面。

Processing 也沿用了帧, `draw()`方法 (包括它的一对 {} 里面的所有代码) 会被程序连续调用, 形成动画的效果。通常, 我们在 `draw` 方法的第一行把屏幕刷白:

```
void draw () {
    background (255);
    // your crazy codes
}
```

这样就形成了一张不断更替的白色画布 (帧)。也许你更爱黑色背景, 那就用 `background(0)`。Processing 采用 0~255 之间的整数 (**#松鼠 抢答题: 从 0 到 255 一共是几个数字? 这个数字是不是有点耳熟?**) 来表示灰度, 0 代表黑色, 255 代表白色, 中间的数字就是各种灰色。那如何设置彩色背景呢? 也很简单, 譬如:



`background(255,170,0);`

就是一个橙色背景，括号里的三个数字分别表示红色、绿色、蓝色的成分，
我们会在 1.3 节详细介绍颜色。

这里是第二遍！

重要的事情要说三遍！当你开始写任何一个程序之前，首先要挑选一种
模式。

(1) 静态模式。静态模式只有 `setup()`方法，没有 `draw()`方法，譬如

```
void setup(){  
    size(200,400);  
    background(255,170,0);  
    line(0,0,width,height);  
    line(0, height,width,0);  
}
```

(2) 画画模式。在画画模式下，每帧画的内容在一张画布上不断叠加（见
1.1 节程序）。

(3) 动画模式。在动画模式下，每帧均在一张空白画布上进行绘图，即在
`draw()`方法第一行用 `background()`指定背景颜色。

静态模式、画画模式和动画模式使 Processing 视觉艺术变得丰富多彩。静
态模式可以输出大幅精美图像；画画模式像画笔一样，不断在画布上留下笔触；
动画模式能产生无穷无尽的不重复的视频。

有时一个程序可以写成两种模式，如画画模式：

```
void setup(){  
    size(800,200);  
    background(255,160,160);  
}  
void draw(){  
    ellipse((frameCount*8)%width, 100, 50,100);  
}
```

以及动画模式：

```

void setup(){
    size(800,200);
}

void draw(){
    background(255,160,160);
    ellipse((frameCount*8)%width, 100, 50,100);
}

```

上面两组代码的唯一区别就是 `background(255,160,160)` 方法是在 `setup()` 里还是 `draw()` 里。代码中运用到了模运算（% 符号），当 % 符号前面的数字增加到等于 `width` 时，模运算的结果为 0。所以当椭圆移动到窗口最右侧后，它会跳回到窗口的最左侧（见图 1-3）。

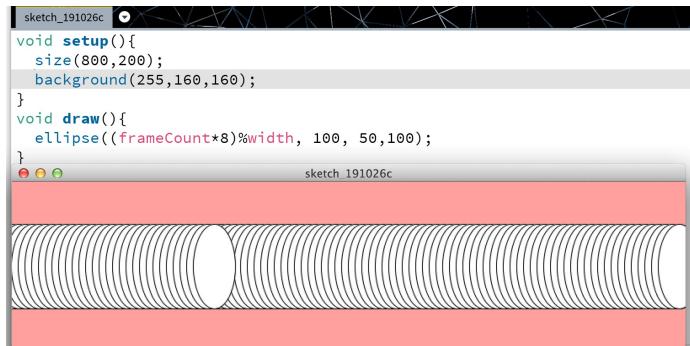
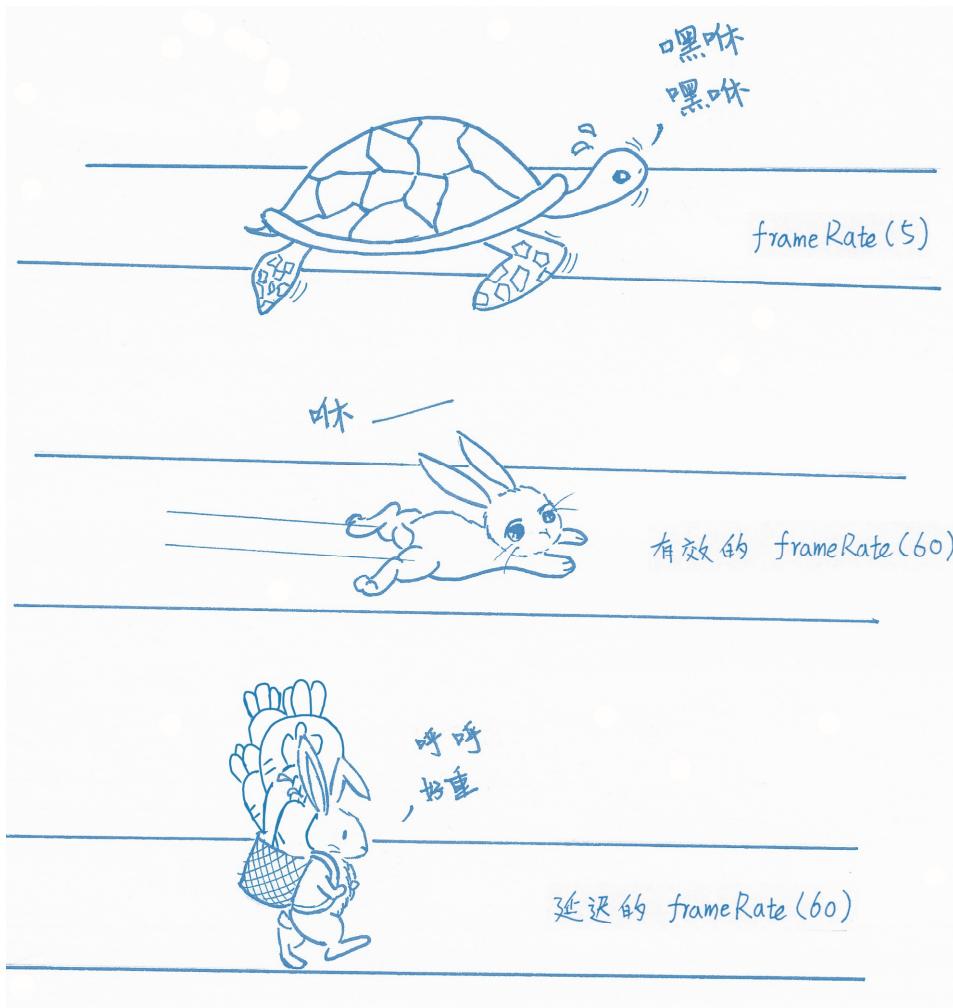
(a) 不调用 `background` 方法(b) 调用 `background` 方法

图 1-3 一个程序写成两种模式

我们可以用 `frameRate()` 方法来控制帧的更新速度，如在上面这个程序中 `setup()` 方法的内部加入“`frameRate(5);`”，小球的移动速度就会非常慢。Processing 的默认速度是 `frameRate(60)`，比电影的 24 帧/秒快很多。还有一种特殊情况，当每帧需要绘制的内容（或所需的运算）特别多时，帧的实际更新速度会比设定的要慢。



1.3 颜色

1.2 节用%运算符使椭圆在画布内循环往复。现在我们来认识一下四个常见的运算符，即加、减、乘、除。在一个空白的程序内写出以下代码：

```
print(1+3);
```

按下工具栏内的播放键，底部的控制台（console）会打印出一个“4”，即 $1+3$ 的计算结果等于4。实际上这行代码做了三件事：①做 $1+3$ 等于4的数学计算；②把答案“4”作为参数传递给print()方法；③print()方法将答案在控制台中打印出来。注意：这行代码要以分号结束。**#调用任何方法都要以分号结束。**
我们再试试减法、乘法和除法：

```
print(1-3);
print(7*3);
print(7/3);
print(7.0/3);
```

控制台会打印出4-22122.3333333。print()打印时不会带空格。为了更清楚地看到结果，可以将第一行程序改写为“print(1-3+",");”，控制台会打印出“-2,”，即在数字后面加了一个逗号。我们还可以使用println()方法，这样每次打印的时候都会换行：

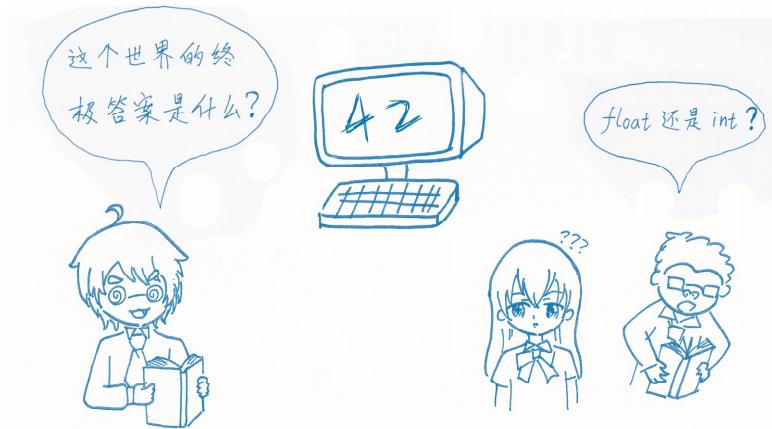
```
println(1-3);
println(7*3);
println(7/3);
println(7.0/3);
```

控制台会分四行打出四个结果。但为什么 $7/3=2$? 实际上，当除号“/”两侧的数字都是整数时，Processing会做整数除法，即“7除3等于2余1”。

$7/3 = 2*3 + 1$

如果不想做整数除法，需要把数字写成小数，如 $7.0/3$ 、 $7.0/3.0$ 、 $7/3.0$ 皆可。

对于 Processing 来说，小数（float）与整数（int）截然不同。我们将在 2.1 节深入了解 float 和 int 这两种数据类型。对于数学家来说，3.0 与 3 是截然不同的。德国数学家利奥波德·克罗内克（Leopold Kronecker）曾说：自然数是上天给的，小数则是人们造出来的。



算术中括号可以控制运算的优先级，这对程序而言同样适用。譬如， $(4+5)*2$ 与 $4+5*2$ 的结果不同，但 $4+5*2$ 与 $4+(5*2)$ 是一回事。

那如何用数学方法和代码来表示五彩斑斓的颜色呢？这要从彩虹说起。日光透过水汽呈现出连续变化的颜色。牛顿在实验室里用三棱镜把白光分解成连续的光谱，后来人们将光谱中的 7 种颜色（按光的波长排序）命名为：红（red）、橙（orange）、黄（yellow）、绿（green）、青（cyan）、蓝（blue）、紫（violet）。每个人对颜色的感受与理解会略有偏差，幸好计算机程序可以用数字来精确表示颜色。譬如：

```
void setup(){
    size(256,256);
    colorMode(HSB);
    background(191, 255,255)
}
```

上面这个程序运行后，将产生一种有点偏紫的蓝色。这段代码首先把程序的颜色模式设为 HSB 模式，即 Hue（色相）、Saturation（饱和度）、Brightness

(亮度)。其中，色相正好对应彩虹从红到紫的光谱，用 0~255 中的一个数字来指定。我们可以移动鼠标来产生 0~255 中的任意一个数字，并产生对应的色相：

```
void setup(){
    size(256,256);
    colorMode(HSB);
}
void draw(){
    print(mouseY+",");
    background(mouseY, 255,255);
}
```

显示窗口颜色的色相 (Hue) 与鼠标 Y 坐标相关联，底部的控制台随时打印出 mouseY 值，即色相的值 (在 0~255 范围内)。

frameCount (当前帧的序号) 与%运算符的组合可以动态地展示七彩颜色：

```
void draw(){
    background(frameCount%256, 255,255);
}
```

红色和黄色给人温暖的感觉，绿色和蓝色则是“冷色”。德国大文豪歌德曾经专门研究人对颜色感知，反对牛顿单纯用数学和物理方法对颜色进行分析。计算机领域则采用了牛顿的思想，最经典的理论就是把所有颜色解释为三原色 (红、绿、蓝) 之间特定比例的组合。这就是 RGB 模式：

```
void setup(){
    size(256,256);
    colorMode(RGB);
}
void draw(){
    background(mouseX, mouseY, 0);
}
```

鼠标在左上角时显示黑色（红色、绿色成分都为 0），在右上角时显示红色，在左下角时显示绿色，在右下角时显示黄色（红色、绿色成分都为 255）。由于 GRB 模式是 Processing 的默认模式，因此 `colorMode(RGB)` 这行代码可以省略。



我们可以采用不同的颜色来绘制不同的物体，譬如：

```
void setup(){
    size(600,400);
    rectMode(CENTER);
}

void draw(){
    colorMode(HSB);
    background(frameCount%256, 255,255);
    colorMode(RGB);
    fill(0,255,255); //cyan
    rect(mouseX, mouseY, 200, 200);
    fill(255,255,0); //yellow
    ellipse(mouseX, mouseY, 200, 200);
}
```

矩形为青色（R:0, G:255, B:255），圆形为黄色（R:255, G:255, B:0）。在调用 `colorMode(RGB)` 之后，`fill(0,255,255)` 内的三个数值就代表红色、绿色、蓝色三种颜色的成分。`fill()` 方法用来指定画形状时填充的颜色。`rect(x,y,w,h)` 是画矩形的方法：

```
#解释 rect      rectMode(CORNER);
```

其中，w 是矩形的宽，本例中 w 取 200，h 是矩形的高，本例中 h 取 200。

我们不但可以指定形状的填充颜色，还能改变形状边缘的颜色：

```
void setup(){
    size(600,400);
}

void draw(){
    stroke(0);      //black
    fill(0,255,255); //cyan
    rect(mouseX, mouseY, 200, 200);
    stroke(255,0,0); //red
    fill(255,255,0); //yellow
    ellipse(mouseX, mouseY, 200, 200);
}
```

其中，矩形的边缘为黑色，圆形的边缘为红色，如图 1-4 所示。该程序中矩形的左上角总是与鼠标对齐，因为程序默认的模式是 `rectMode(CORNER)`。在任意一个程序块内部，代码是一行一行依次执行的，因此第 5、6 行的 `stroke` 和 `fill` 影响了第 7 行 `ellipse` 的颜色，而第 8、9 行的 `stroke` 与 `fill` 只能影响第 10 行 `rect` 的颜色。

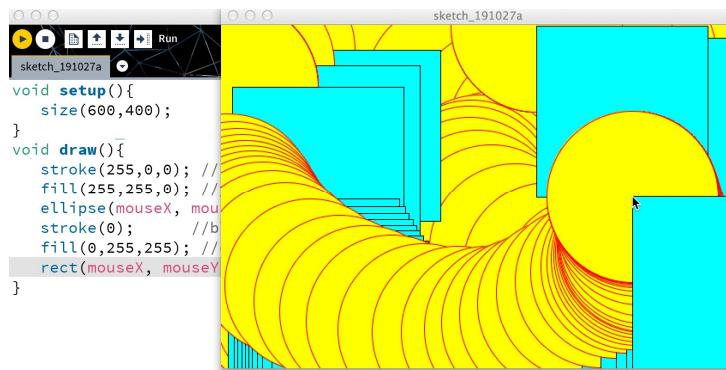


图 1-4 黑色边缘的方块，红色边缘的圆形

在编程的时候，经常需要暂时性地删掉一段代码，我们一般用一对/* */来把一段代码“注释掉”，如下面的程序段把矩形暂时去掉了：

```
void setup(){
    size(600,400);
}

void draw(){
    stroke(255,0,0); //red
    fill(255,255,0); //yellow
    ellipse(mouseX, mouseY, 200, 200);
    /* stroke(0);      //black
    fill(0,255,255); //cyan
    rect(mouseX, mouseY, 200, 200);*/
}
```

当我们再次需要这段代码时，把/* */符号删除即可。

水彩画中的颜色可以呈现出半透明的状态。Processing 中的颜色也可以，我们只要把 `fill()`、`stroke()` 等方法内的三个参数变成四个即可，其中第四个参数表示透明度（取值范围同样是 0~255，0 为完全透明，255 为不透明）。譬如：

```
void setup(){
    size(600,400);
}

void draw(){
    colorMode(RGB);
    background(255);
    fill(0,255,0,100); //green, transparent
    ellipse(mouseX, mouseY, 200, 200);
    colorMode(HSB);
    fill(25,255,255,50); //orange, transparent
    rect(mouseX, mouseY, 200, 200);
}
```

运行结果如图 1-5 所示。

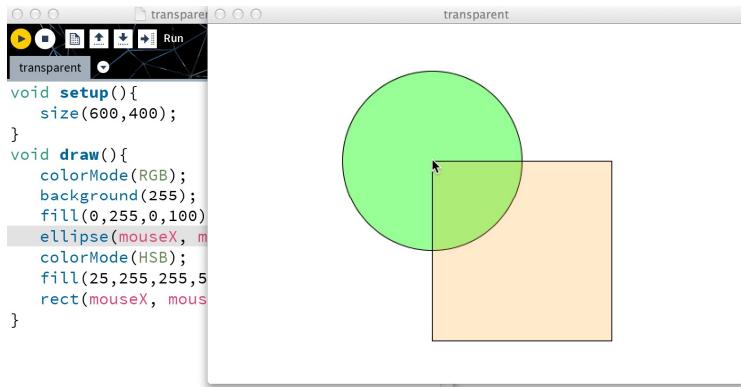


图 1-5 半透明的绿色圆形和半透明的橙色方块

其中，圆形的填充颜色为 RGB 模式下的 `fill(0,255,0,100)`，纯绿色结合了 100 的透明度之后变成了浅绿色。正方形的填充颜色为 HSB 模式下的 `fill(25,255,255,50)`，橙色结合了 50 的透明度之后显得非常通透。

你也许已经发现 `background()`、`stroke()`、`fill()`这几个方法的括号内可以有一个、三个或四个参数，如 `fill(255)`、`fill(0,255,0)`、`fill(0,255,0,100)`。在 Processing 内部，其实已经定义好了三个不同的版本，它们名称相同，但参数不同，这称为方法的重载(overload)。

红色 → (0, 255, 255) HSB
→ (255, 0, 0) RGB



第2章

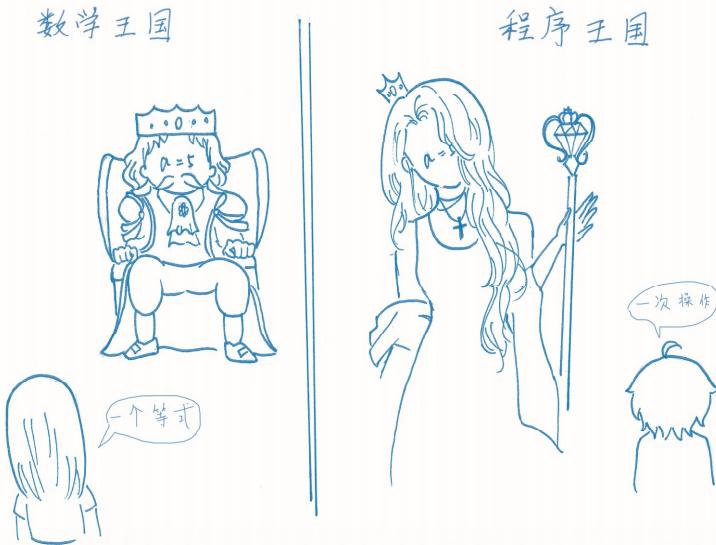
变化多端的圆形

2.1 变量与循环

在第1章我们体验了Processing的画布、画笔和颜色。不难发现，Processing用数字表示图形和颜色，而变化的数字可以产生变化的图形和颜色。变量(variable)就是用来表示变化的数，譬如：

```
int a=-1; //声明一个整数变量 a, 把-1 赋予 a  
println(a);  
a=-3; // 把-3 赋予 a  
println(a);
```

运行程序后，控制台打印出-1 和-3 两个结果。代码中的 int a=-1 语句做了两件事：①声明一个整数型(int) 变量 a；②把一个具体的值-1 赋予 a。因为 a 是变量，所以我们可以随时赋给它新的值，如 a=-3。一般来说，变量的名字用小写，如果变量的名字由几个单词组成，可以用 the_first_variable 或 theFirstVariable 这两种形式。



值得注意的是，`a` 是一个 `int` 型变量，因此不能把小数赋给它。例如，`a=0.5` 语句会触发程序异常（Exception），消息区域会变成深褐色，并显示：cannot convert from float to int。

当等号右侧的数据类型无法自动转换为等号左侧的数据类型时，就会触发这类异常。

另一种常用的数据类型是小数（`float`），现在我们来创建一个名为 `x` 的 `float` 型变量：

```
float x=-1.6; //声明一个小数变量 x, 把-1.6 赋予 x
println(x);
x+=2; // x=x+2
println(x);
```

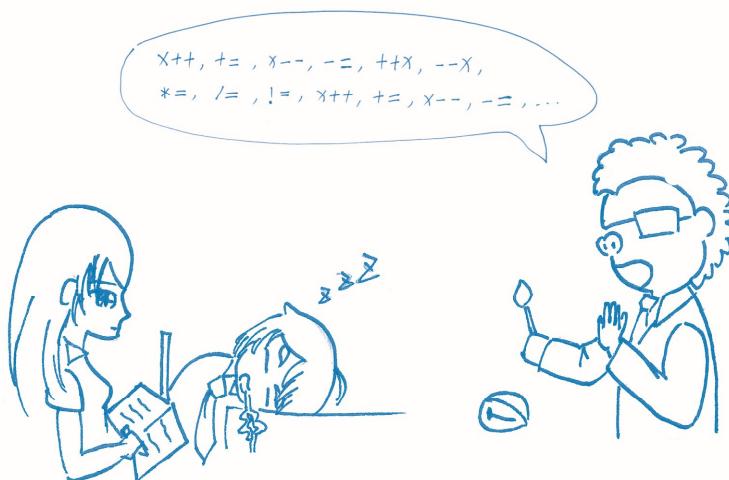
其中，`+=` 运算符的含义是：把运算符左边的变量加上右边的数，并计算最终结果。类似的还有`=`、`*=`和`/=`，你不妨自己试一试，看看结果是否和想象的一样。运行上面的程序会得到 `0.5`、`-1.6`、`0.39999998` 三个结果。最后一个结果看上去很奇怪，因为正确结果应该是 `0.4`。**#好可怕！程序吃掉了 0.0000002！** 实际上 `float` 数据类型所表示的数字的精度是有限的，这和我们熟悉的“绝对精确”

的数学”完全不同。但对于生成艺术来说精度似乎不是问题。

变量之间也可以做运算，例如：

```
float x= 6.9;  
x++; // x=x+1  
float pigPeppa=11.25;  
float pigGeorge=8.5;  
float result = (pigPeppa +pigGeorge)/x;  
println( "result is " +result);
```

其中，++运算符的含义是给左边的变量加上 1。“程序猿们”热爱偷懒，发明了很多简化的运算符，类似的还有--运算符。程序的运行结果为“result is 2.5”。最后一句 `println()` 中的参数为"result is "+result，即把一个小数（result 值为 2.5）与字符串“result is”拼接成一段字符串。



现在我们在屏幕中央画一个圆形，使圆的边缘始终与鼠标光标的对齐：

```
int w=800;
```

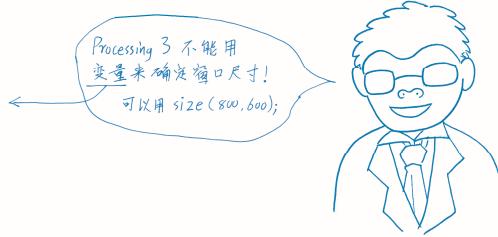
```
int h=600;
```

```

void setup() {
    size(w, h);
    colorMode(HSB);
}

void draw() {
    fill(mouseX*256/w, mouseY*256/h, 255);
    float r=dist(mouseX, mouseY, w/2, h/2);
    ellipse(w/2, h/2, 2*r, 2*r);
    line(mouseX, mouseY,w-mouseX, h-mouseY );
}

```



代码运行结果如图 2-1 所示。

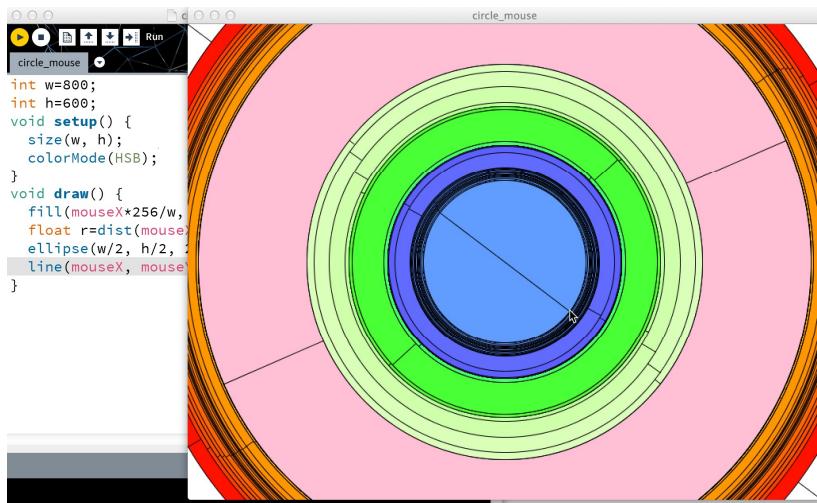


图 2-1 一组同心圆，移动鼠标可控制圆的大小

在指定 HSB 颜色模式时，我们采用了变量之间的运算：`mouseX*256/w`，即把鼠标的 x 坐标（从 0 到 w）缩放到 0~255 的范围。代码中的 `dist()` 是 Processing 自带的求两点之间距离的方法，四个参数分别为第一个点的 x 与 y 坐标，以及第二个点的 x 与 y 坐标。运用勾股定理，代码中的 `dist()` 语句可以用以下代码代替：

```

float dx=mouseX-w/2;
float dy=mouseY-h/2;
float r= sqrt(dx*dx+ dy*dy);

```

其中，`sqrt()`是求平方根的方法，`sqrt` 是 square root 的缩写。



下面来认识一下编程语言中常用的 `for` 循环语句：

```

for(int i=0; i<8;i++){
    int square= i*i;
    println( "square of " +i+ " is " + square );
}

```

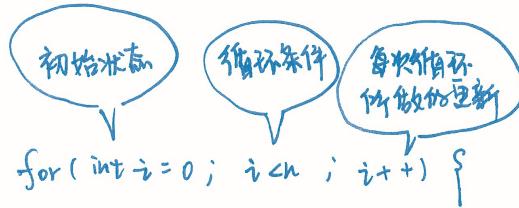
该循环从变量 `i` 等于 0 开始，每次循环给 `i` 加上 1，直到 `i` 为 8（不再满足 `i<8` 的循环条件）的时候停止循环。全世界的“程序猿”都喜欢从 0 开始计数，所以

```

for(int i=0; i<n;i++){
    //do anything you want
}

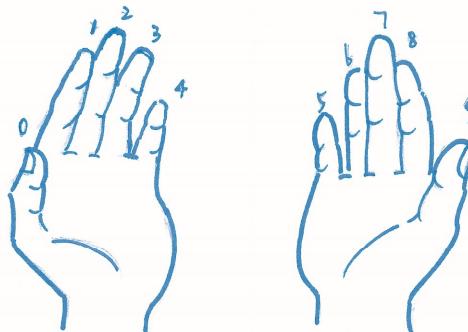
```

中的 i 会从 0 变到 $n-1$ ，因此花括号内的代码不会经历“ $i=n$ ”的情况。现在我们来数一下自己有几根手指。



写在这里的代码会被循环执行

}



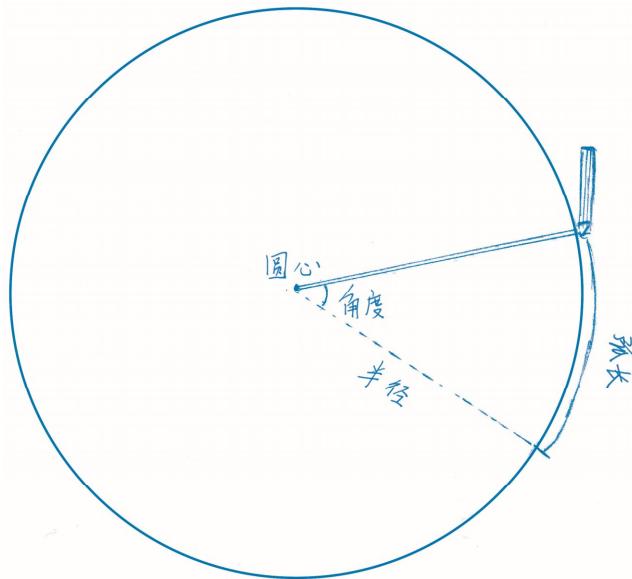
for 循环有很多灵活的用法，譬如计算 5~25 范围内的奇数的平方根：

```
for(int i=5; i<=25;i+=2){
    float sq_root= sqrt(i);
    println( "square root of " +i+ " is " + sq_root );
}
```

这里的初始状态条件是 $i=5$ (不是从 0 开始)，继续循环的条件是 $i \leq 25$ (允许 i 等于 25)，每次循环会把 i 增加 2。

2.2 心

圆形既是宇宙中最简单最容易理解的图形，又是神秘且耐人寻味的图形。本节我们将利用圆形来生成富于变化的图案。首先注意圆形的几个基本特征：圆心、半径、角度、弧长。在铅笔上拴一根绳子，绷紧绳子并把绳子的另一端固定在纸上，绕着固定点转动，铅笔就能画出光滑的圆弧。



绳子一端的固定点即为圆心，绳子的长度就是半径（直径是半径的 2 倍）。铅笔转过的角度决定了圆弧的长度，当转过 360° 时就形成了一个完整的圆。当圆的半径为 1（单位圆）时，整个圆弧的长度为

#祖冲之在 1500 多年前算到了这一位

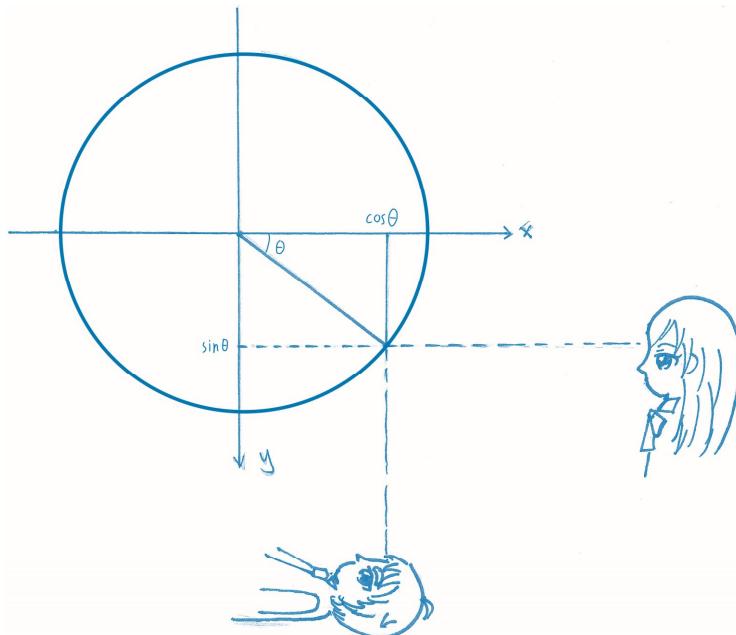
3.14159265358979323846264338327950288419716939937510582097494459
23078164062862089986280348253421170679821480865132823066470938446095
50582231725359408128481117450284102701938521105559644622948954930381
96442881097566593344612847564823378678316527120190914564856692346034
86104543266482133936072602491412737245870066063155881748815209209628

29254091715364367892590360011330530548820466521384146951941511..... #
这不就是 π 吗？

“程序猿”和数学家爱用弧度（radian）来表示角度的大小，弧度与角度之间的关系为： 弧度= 角度* $\pi/180^\circ$ 。该公式基于这样一个事实： 180° 和 π 代表的角度是一样大的。Processing 用 PI 这个常量来表示 π ：

```
println(PI);
float rad=90*PI/180;
println(rad);
```

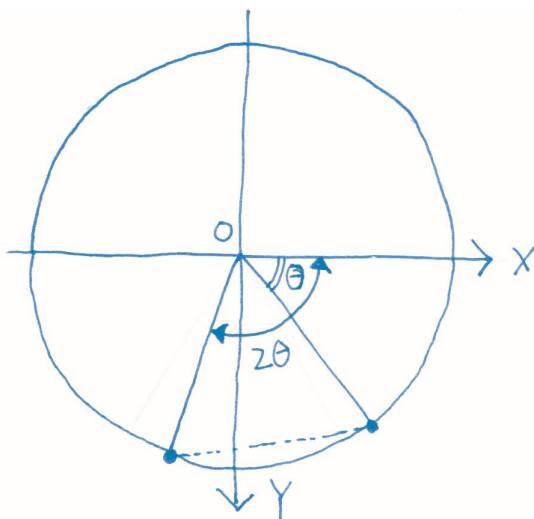
本书偏好用弧度来表示角度，其中一个原因就是仰慕欧拉公式 $e^{i\theta} = \cos\theta + i\sin\theta$ （其中 θ 是弧度制的角度），它被誉为世界上最完美的公式，在本书的最后我们将用它生成极其繁复的分形（fractal）图案。该公式包含了 cos 与 sin 这两个三角函数。实际上 cos θ 与 sin θ 函数是单位圆上一点的水平投影与垂直投影，而对应的角度就是 θ ，如下图所示。



因为点在圆环上的运动是周而复始的，每当一个点转过 2π （对应 360° ）就会回到初始位置，所以 cos 和 sin 都是周期函数，周期为 2π 。运用勾股定理，

设斜边的长度为 1，就可以发现 $\cos^2\theta + \sin^2\theta = 1$ 。当圆的半径为 r 时，圆上一点的 x 坐标为 $r\cos\theta$ ，y 坐标为 $r\sin\theta$ ，即在笛卡儿坐标系下，圆可以用三角函数来表示。

现在我们对一个半径为 400 的圆进行 432 等分，将其中任意一点（对应角度 θ ，在程序中写作 theta）与另一点（对应角度 2θ ）连线，如下图所示。



利用 `for` 循环语句可以方便地画出所有连线：

```
float r=400;
int n=72*6;
void setup() {
    size(800, 800);
}
void draw() {
    background(255);
    translate(400, 400);
    for (int i=0; i<n; i++) {
        float theta= 2*PI*i/n;
        float y= r*sin(theta);
        float x= r*cos(theta);
```

```

    float y2= r*sin(2*theta);
    float x2= r*cos(2*theta);
    line(x, y, x2, y2);
}
}

```

其中, `translate(400, 400)`语句把整个画布移动到屏幕中央, 否则图形会偏向左上角。`for` 循环内部的 6 行代码完成了三件事: ①计算第 i 个点对应的弧度 θ ; ②计算弧度为 θ 的点的 x 坐标和 y 坐标, 以及弧度为 2θ 的点的 x 坐标和 y 坐标; ③连线。

运行程序将得到一个心形 (`cardiod`), 如图 2-2 所示。该心形中每根线的规律是: 一端对应的角度是另一端对应的角度的 2 倍。那如果是 3 倍的关系会怎么样呢? 我们对代码稍作修改:

```

float y2= r*sin(3*theta);
float x2= r*cos(3*theta);

```

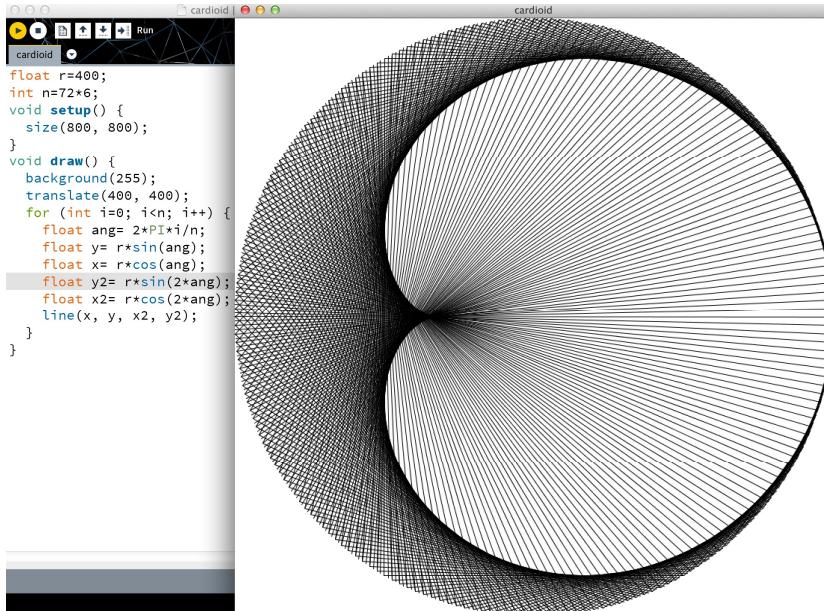


图 2-2 `cardiod` 图形

就能得到一个新的图案，呈现出两瓣的对称性。同理也能编写 4 倍、5 倍、6 倍的图形，能分别显示出 3 瓣、4 瓣、5 瓣的旋转对称性。图 2-3 列出了一系列不同倍数下的图案。

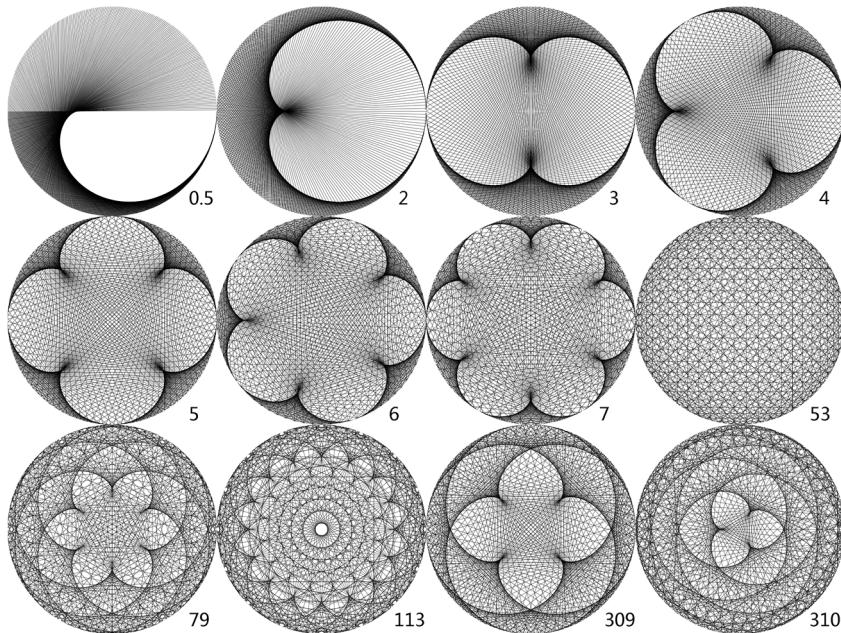


图 2-3 不同倍数产生不同图形

不难发现，输入不同的倍数就能得到不同的图形。在视觉实验中，我们经常会尝试不同的参数值，往往能得到意想不到的效果。很多有意思的作品就是在不断尝试中发现的。现在我们让上一个程序中的倍数连续变化(与 `frameCount` 关联)，从而产生动画效果，程序如下：

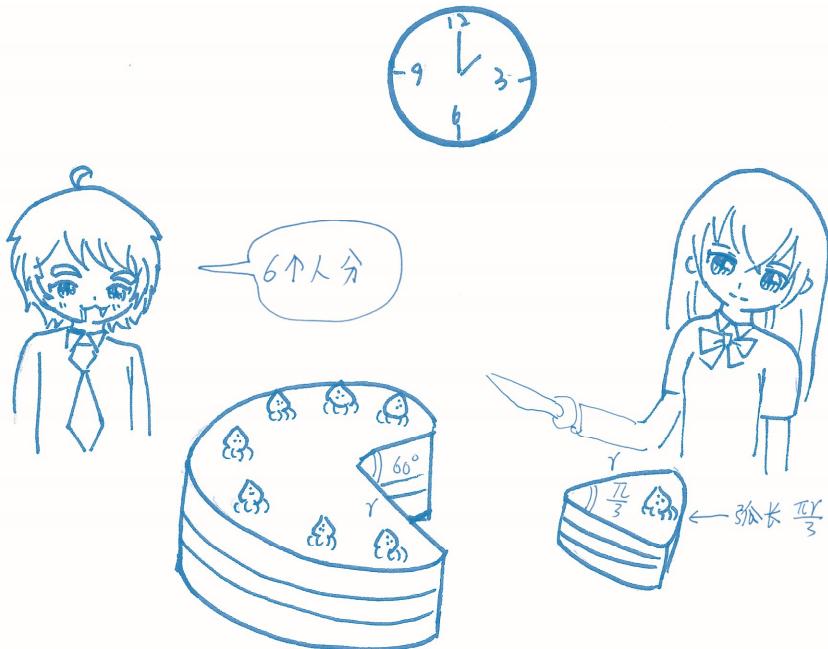
```
float r=400;
int n=72*6;
void setup(){
    size(800,800);
}
void draw(){
    background(255);
```

```

translate(400,400);
float s=1+0.05*frameCount;
for(int i=0;i<n;i++){
    float theta= 2*PI*i/n;
    float y= r*sin(theta);
    float x= r*cos(theta);
    float y2= r*sin(s*theta);
    float x2= r*cos(s*theta);
    line(x,y, x2,y2);
}
}

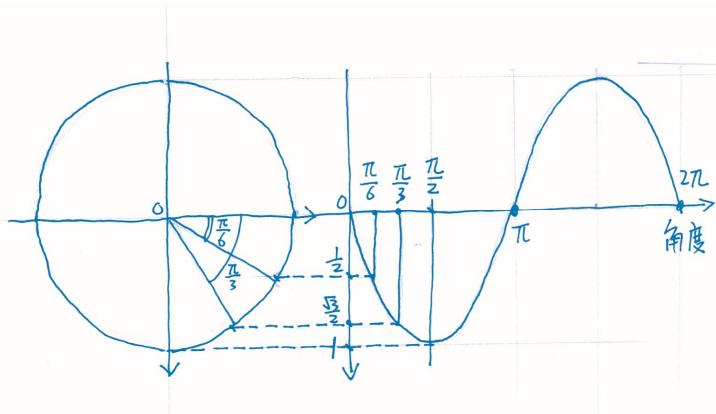
```

其中，**float** 型的变量 s 决定了每根线两端对应角度之间的倍数，当 s 随着帧数不断变化时，整个图形也就动起来了。



2.3 圆的魔术

2-2 节介绍了 `cos` 函数和 `sin` 函数是圆上一点的水平和垂直投影。而下图中的 `sin` 函数表达了弧度与垂直投影之间的关系。



太阳每天从东方升起，而点在圆上每转 2π （对应 360° ）就会复位，因此 `sin` 函数是一个周期为 2π 的函数。`cos` 函数的周期同样是 2π 。从上图可以看出，`sin` 函数的最大值为 1，最小值为 -1。Processing 可以方便地绘制三角函数：

```
void setup() {
    size(800, 600);
}

void draw() {
    background(255);
    for (int x=0; x<800; x+=2) {
        float theta= x* 2*PI /800;
        float y= 300* (sin(theta)+1) ;
        stroke(255, 0, 0);
        line( x, 0, x, y );
    }
}
```

该程序绘制了 400 根垂直线，注意 `x+=2` 语句使 `for` 循环只产生 0、2、4、6 等偶数。每根垂直线一端的 y 坐标为 0（屏幕的上边缘），另一端的 y 坐标为 $300 * (\sin(\theta) + 1)$ 。由于 `sin` 函数的值在 $-1 \sim 1$ 之间变化，因此 `sin(theta) + 1` 的值在 $0 \sim 2$ 之间，最终 $300 * (\sin(\theta) + 1)$ 的值会在 $0 \sim 600$ 之间变化，从而撑满整个屏幕高度。

在 `float theta=x*2*PI/800` 语句中 x 的最大取值为 800 (#严格来说是 798)，因此 $x * 2 * \pi / 800$ 这个表达式的最大取值为 2π ，正好是 `sin` 函数的一个周期。如果我们想让曲线重复出现两次，可以写为：

```
float theta= x* 4*PI /800;
```

我们在 `for` 循环中再加入绿色的 `cos` 形状：

```
for (int x=0; x<800; x+=2) {
    float theta= x* 4*PI /800;
    float y= 300* (sin(theta)+1) ;
    stroke(255, 0, 0);
    line( x, 0, x, y );
    y= 300* (cos(theta)+1) ;
    stroke(0, 255, 0);
    line( x+1, 0, x+1, y );
}
```

绿线和红线是完美错开的，奇妙的是，我们会看到黄色（见图 2-4）。也许大家还记得，在 `RGB` 颜色模式下， $(255, 255, 0)$ 表示黄色，而上面这个程序细密地混合了红线和绿线。

印度尼西亚的小天使凤蝶，翅膀上闪烁着鲜亮而难以用语言准确描述的绿色，但在显微镜下人们却发现它的翅膀上密布了黄色和蓝色两种颜色。

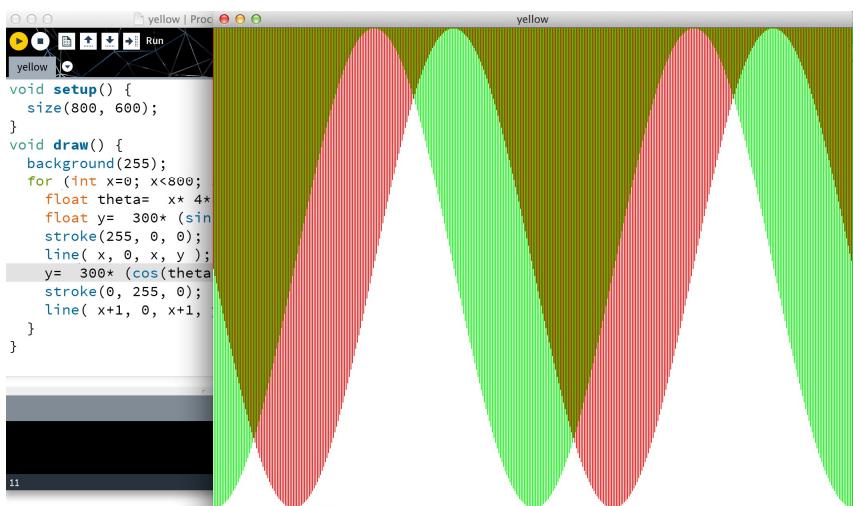
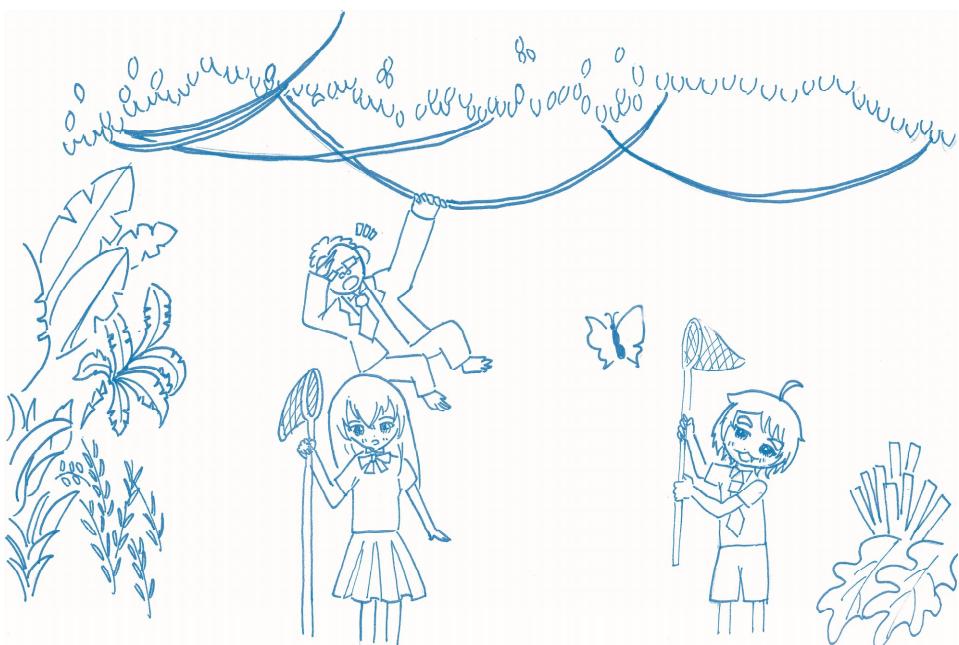
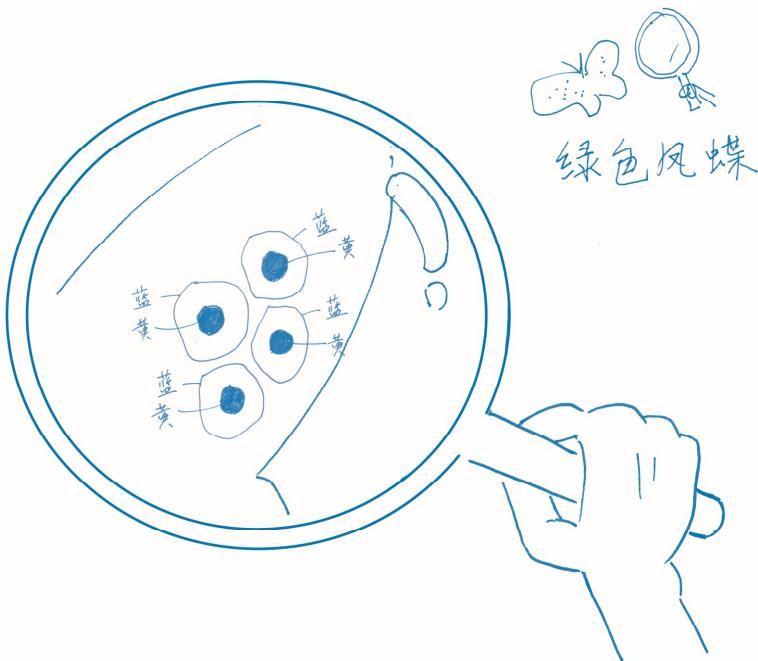


图 2.4 红色波浪与绿色波浪的叠加产生了黄色的错觉





最后，我们添加一个 `float` 型的变量 `shift`，让它的每帧递减 0.1，从而让绿色曲线向左移动、让红色曲线向右移动。

```

float shift=0;
void setup() {
    size(800, 600);
}
void draw() {
    background(255);
    for (int x=0; x<800; x+=2) {
        float theta=  shift + x* 4*PI /800;
        float y=  300* (sin(theta)+1) ;
        stroke(255, 0, 0);
        line( x, 0, x, y );
        theta= -shift + x* 4*PI /800;
        y=  300* (cos(theta)+1) ;
    }
}

```

```

    stroke(0, 255, 0);
    line( x+1, 0, x+1, y );
}
shift -= 0.1;
}

```

圆的一个本质特征是：圆上任意一点到圆心的距离都一样，这个距离就是圆的半径。下面这个程序在半径为 290 的圆上画了 720 个小圆圈。

```

void setup() {
    size(600, 600);
}

void draw() {
    background(255);
    fill(0);
    int n=720;
    for (int i=0; i<n; i++) {
        float theta = i* 2*PI /n ;
        float r = 290;
        float x= 300+r*cos(theta);
        float y= 300+r*sin(theta);
        ellipse(x,y, 6,6);
    }
}

```

现在我们来打破这个圆：让每个点的半径（到圆心的距离）对应它的角度 theta，譬如

```
float r = 290* sin(0.5*theta);
```

把这句代码替代原代码中的 `float r = 290;` 原来的圆形就变成了一个心形（见图 2-5），看上去和 2.2 节中的心形相似，如图 2-5 所示。

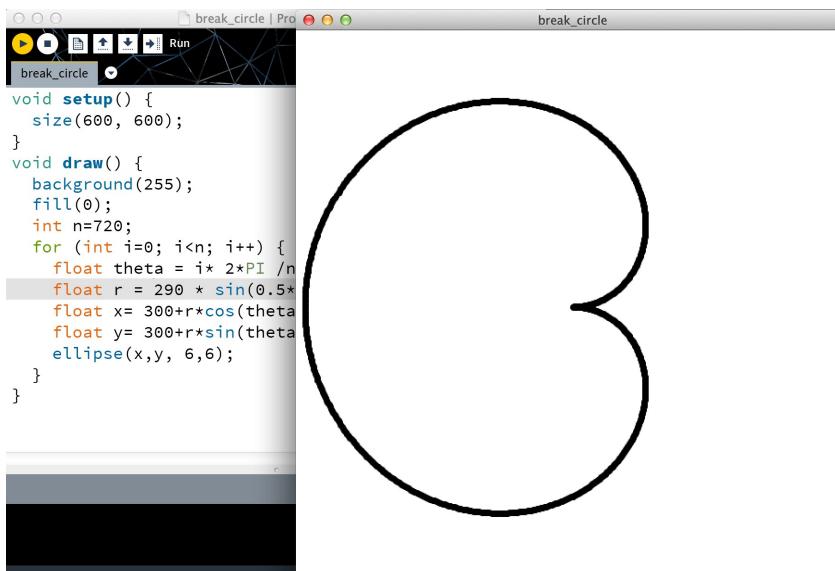
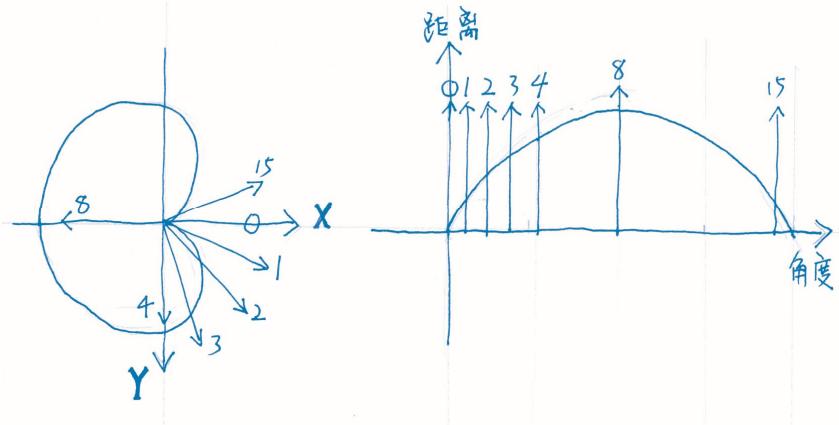


图 2-5 心形

这是为什么呢？我们可以想象， \sin 函数图像在 16 支向上的箭上留下了刻痕，如果把这 16 支箭呈放射状布置，这些刻痕刚好组成一个心形。



如果代码改成了 `float r = 290 * sin(2*theta);`; 原来的圆形又变成了四叶草的形状，如图 2-6 所示。

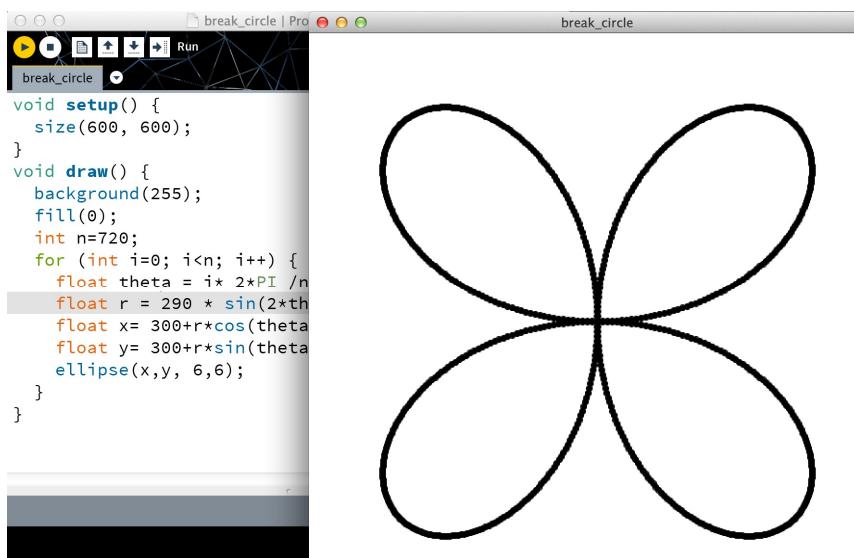
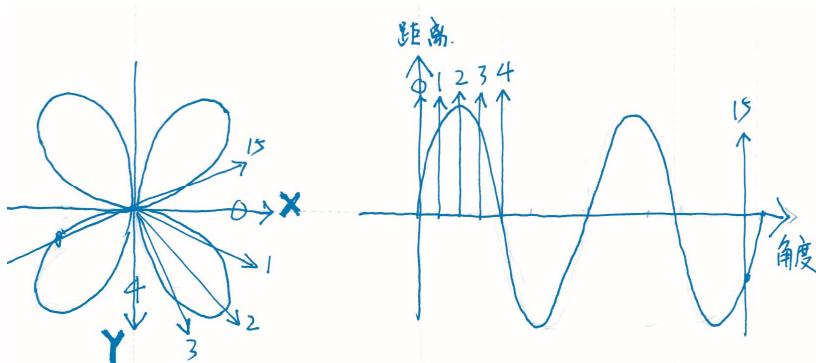


图 2-6 四叶草

我们依然可以用 16 支箭头来解释四叶草。



我们还可以在 `float r = 290 * sin(2*theta)` 这句代码中尝试其他倍数，如 `sin(3*theta)`、`sin(4*theta)` 等，看看会得到什么形状。不难发现，这个倍数对形状的影响非常大，现在我们让这个倍数随 `frameCount` 连续变化，从而让形状连续地变化，程序如下：

```

void setup() {
    size(600, 600);
}

```

```
void draw() {  
    background(255);  
    fill(0);  
    int n=720;  
    for (int i=0; i<n; i++) {  
        float theta = i* 2*PI /n ;  
        float r = 290 * sin(0.02*frameCount*theta);  
        float x= 300+r*cos(theta);  
        float y= 300+r*sin(theta);  
        ellipse(x,y, 6,6);  
    }  
}
```

整个第 2 章的主角是圆，它简单纯粹又暗藏丰富的变化。在笛卡儿坐标系下的圆又可以转化为 `cos` 函数和 `sin` 函数，帮助我们进行数学思考和编程实验。本章用变量、`for` 循环、三角函数发掘了圆的各种变化，希望大家也可以发现圆不为人知的一面。

第3章

弹！弹！弹！

3.1 弹球

生成艺术的一大特色是动画，而且这种动画往往可以无限地运行下去而不会在内容上重复。听上去是不是很神奇？而且 Processing 可以用随机数的原理让动画每次的运行都不一样。我们先绘制一个小球并让它动起来，程序如下：

```
float x=0;  
float y=300;  
void setup(){  
    size(800,600);  
}  
void draw(){  
    background(255);  
    fill(0,255,0);  
    ellipse(x, y, 30, 30);  
    x+=3;  
}
```

运行程序，我们可以看到一个绿色的小球向左移动，最后飞出了展示窗口。使小球运动的代码是 `x+=3` (`x=x+3`)，意思是：把当前的 `x` 值加上 3，然后把计算获得的值重新赋给 `x`。因此，每当 `draw()` 方法运行一次，变量 `x` 的值就会增

加 3。实际上，小球是一顿一顿地跳动，但由于每秒 `draw()` 会运行近 60 次，人眼感受到的画面是连续运动的，其实这也是一般的动画或电影的基本原理。

该程序有三大部分：

(1) 首先声明两个变量 `x` 和 `y`（均为 `float` 型），代表了小球中心的位置，它们的值将会在程序运行的过程中发生改变。

(2) 其次是 `setup()` 方法块，其内部的代码会在程序刚启动时执行一次，在这里用 `size()` 来指定窗口的大小。

(3) 最后是 `draw()` 方法块，其内部代码会被反复执行来产生动画效果，这里用了 `ellipse()` 来绘制小球，最后让它的横坐标递增 (`x+=3`)，递增的效果在下一帧才能体现出来。

简单的数理知识可以帮助我们编写代码。小球运动的速度可以用两个参数来表示：`dx` 表示沿水平方向的运动速度，`dy` 表示沿垂直方向的运动速度。根据高等数学，物体的位置 `x` 对时间的导数（derivative）就是物体的运动速度，而 `dx` 就表示这个导数。在程序的开头设 6 个变量：

```
float x=300;
float y=300;
float dx=3;
float dy=-0.5;
int w=800;
int h=800;
void setup() {
    size(w,h); //在 Processing 3 版本下使用 size(800,600)
}
```

而 `draw()` 方法中的代码让小球的位置根据速度产生变化：

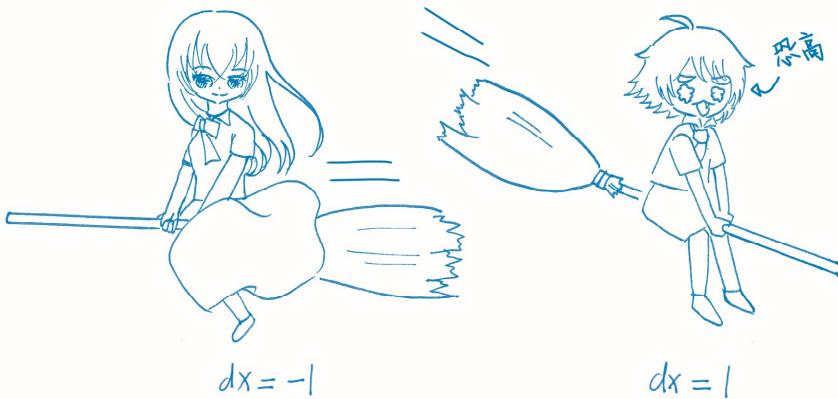
```
void draw(){
    background(255);
    fill(0,255,0);
```

```

ellipse(x, y, 30, 30);
x+=dx;
y+=dy;
}

```

在坐标系中速度是有符号的（可能是正的，也可能是负的）。在日常生活中速度总是正的，因为人们一般说的是速度的大小，不包含方向信息。譬如，2008年京津城际列车的最高速度是105米/秒。速度的大小为 (dx^2+dy^2) 的平方根。 dx 是速度的水平分量，而 dy 是速度的垂直分量。上面例子中小球的速度大小约等于3.041像素/帧。



如果要让小球撞到窗口的边缘就弹回来，应该怎么做呢？首先，程序需要检测到“小球撞到窗口的边缘”这种状态。如果只考虑小球飞出右边框的情况，可以用if判断语句：

```

if(x>w){
    dx*=-1;
}

```

也就是说，一旦 x 的值大于窗口的宽度，水平方向上的运动速度就会反转（正变成负，负变成正）。而 $dx *= -1;$ 这句代码很简练地实现了“符号反转”的操作，它等价于 $dx = -1 * dx;$

if判断语句在Java语言中十分常见，关键词if后面需要跟一对圆括号()和

一对花括号{}。圆括号()内写判断条件，而在条件满足的情况下，花括号{}内写需要执行的代码。Java 语言对格式的要求非常严格，其中圆括号()、花括号{}都必须成对出现，否则编译过程就会报错。

以上代码能让小球被右边框弹回，但是随着它一直向左运动，最终会飞出左边框。所以需要再添加一个判断：

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

在左边框处也让速度反转。为了让程序更简短，可以把两个判断条件合并在一个 if 语句中：

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

其中，“||” 符号表示“或者”关系，即当 $x < 0$ 或 $x > w$ 时，速度 dx 需要变符号。Java 语言中还有“ $\&\&$ ”操作符，表示“并且”关系，我们在后面的程序中会用到。现在小球的运动被限制在窗口的左右区间内，对上下边框也可以进行类似的处理：

```
if(y>h || y<0){  
    dy*=-1;  
}
```

最后对画面风格稍作调整，得到图 3-1 所示的代码。

弹球程序终于完成啦！小球拖着一条尾巴在窗口内跳动（见图 3-1）。这条逐渐淡化的尾巴，实际上是因为有半透明的黑色不断蒙在整个窗口上。`draw()`方法内第一行 `fill(0, 10);` 其中数值 0 表示黑色（255 表示白色），数值 10 表示颜色的透明度（255 表示完全不透明）。第二行 `rect(0,0,w,h);` 用这种透明的黑色绘制了一个与窗口一样大的矩形。

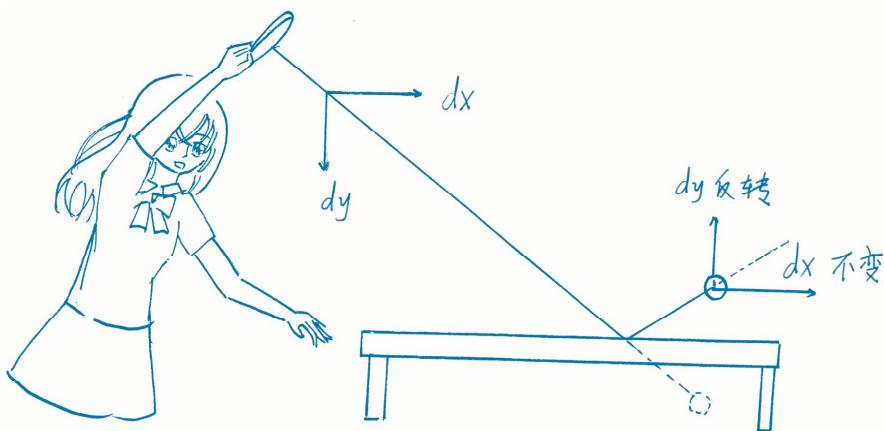
```

float x=300;
float y=300;
float dx=3;
float dy=-0.5;
int w=800;
int h=800;
void setup() {
    size(w,h); // size(a
}
void draw(){
    fill(0, 10); // transp
    rect(0,0, w,h); //mas
    fill(255);
    noStroke();
    ellipse(x, y, 20,20);
    x+=dx;
    y+=dy;
    if (x>w || x<0){
        dx*=-1;
    }
    if (y>h || y<0){
        dy= -dy;
    }
}

```



图 3-1 拖着尾巴的探求



你一定掷过垒球、铅球或石子，当你用最大力气掷出时，无论向哪个方向投球，物体的初速度（大小）是一样的。在 Processing 程序里如何表达“速度大小相同而方向不同”呢？我们可以用第 2 章所讲的三角函数（请脑补一个半径为 speed 的圆），程序如下：

```

float speed=5;
float ang= random(-PI, PI);

```

```
dx=speed*cos(ang);
dy=speed*sin(ang);
```

在上面的程序中，第一行设定速度的大小为 5，第二行 `random(-PI, PI)`;随机产生一个角度（这里使用弧度来表示角度），第三、四行用三角函数来计算速度的水平与垂直分量。由于引入了随机数进行初始化，因此程序每次运行的结果都不一样。最后的弹球程序如下：

```
float x=350;
float y=400;
float dx, dy;
int w=700;
int h=800;
void setup() {
    size(w,h); //在 Processing 3 版本下使用 size(700,800)
    float speed=10;
    float ang= random(-PI, PI);
    dx=speed*cos(ang);
    dy=speed*sin(ang);
}
void draw(){
    fill(0, 10);
    rect(0,0, w,h);
    fill(255);
    noStroke();
    ellipse(x, y, 20,20);
    x+=dx;
    y+=dy;
    if (x>w || x<0)
        dx*=-1;
    if (y>h || y<0)
```

```
dy= -dy;  
}
```

这里使用了一个小技巧：当 `if` 判断语句内只有一行代码时，可以省略花括号。

练习题 1：如何绘制两个小球，并让它们各自运动？

练习题 2：如何绘制很多（如 356 个）小球，并让它们各自运动？

3.2 布尔先生

随着代码长度的增加，需要让代码排列整齐，特别是要整理每一行的缩进。下一个层级的代码需要向右缩进，菜单中的“Edit / Auto Format”自动格式化命令可以方便地让代码缩进。当代码有错的时候（如少写一个括号），Auto Format 命令可能会给出一个错误的格式。代码的整齐不仅关乎美观，更重要的是便于阅读和编辑。

本节重点介绍各种逻辑判断，下面我们先用代码来判断鼠标是否在圆形内部：

```
float x=220;  
float y=250;  
float r=180;  
void setup() {  
    size(500, 700);  
}  
void draw() {  
    background(255);  
    if ( r>dist(x, y, mouseX, mouseY)) {  
        fill(0, 0, 0,100);
```

```

} else {
    fill(0,255,0,100);
}
ellipse(x, y, 2*r, 2*r);
}

```

当鼠标移动到圆圈内时（鼠标到圆心的距离小于半径），圆的颜色会从原来的绿色变成灰色。填充颜色中都使用了透明度（`fill` 中的第四个参数为透明度，0 表示完全透明，255 表示完全不透明）。代码中的 `dist()` 是 Processing 内部定义好的函数，用来计算平面中两个点之间的距离，其中前两个参数表示第一个点的 x、y 坐标（在这里我们填入了圆心的坐标），后两个参数表示第二个点的坐标（在这里我们填入了光标的位置）。

最常见的 `if-else` 判断语句是两段式的：

```

if( // condition A ) {
    //codes A
} else {
    //codes B
}

```

当 `if` 后面圆括号内的条件得到满足时，后面花括号内的代码将会运行，否则 `else` 后面花括号内的代码会运行。`if-else` 语句还可以处理多重条件，如下所示。

```

if( // condition A ) {
    //codes A
} else if( // condition B){
    //codes B
} else if( // condition C){
    //codes C
} else {
    //codes D
}

```

```

    }
}

```

图 3-2 所示的程序同时处理了鼠标与两个圆形的几何关系（复制修改之前的代码即可）。

```

float x1=220;
float y1=250;
float r1=180;
float x2=600;
float y2=500;
float r2=400;
void setup() {
    size(500, 700);
}
void draw() {
    background(255);
    if (r1>dist(x1, y1, mouseX, mouseY)) {
        fill(0, 0, 0, 100);
    } else {
        fill(0, 255, 0, 100);
    }
    ellipse(x1, y1, 2*r1, 2*r1);

    if (r2>dist(x2, y2, mouseX, mouseY)) {
        fill(0, 0, 0, 100);
    } else {
        fill(0, 0, 255, 100);
    }
    ellipse(x2, y2, 2*r2, 2*r2);
}

```

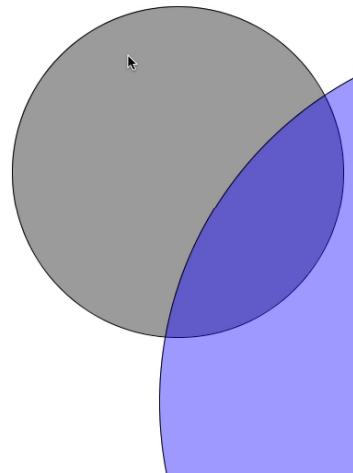


图 3-2 鼠标的位置决定了图形的颜色

“是与否”的逻辑判断在日常生活中比比皆是，人们甚至都不一定会意识到我们的大脑在进行逻辑判断。譬如，你听到“他不会不去”的时候，大脑会自动领会“他会去”的意思。大约 170 年前，在英格兰有位善于自学的天才乔治·布尔（George Boole）， he 觉得人类思维的一项基本法则是“布尔运算”（后人以他的名字来命名）。布尔运算基于布尔型变量（之前我们已经熟悉了 float 型、 int 型变量），如：

```

boolean a=true;
boolean b=false;

```

其中 a 的值为 true（真），b 的值为 false（假），任何布尔型变量只能取这两种值。针对布尔型变量的运算符号有：!（非）、||（或）、&&（并且）和==（比较）。譬如：

```
println(!a);
```

```
    println(!b);
```

从打印的结果可以看出，!a 的值变成了 `false`，而!b 的值为 `true`。俗话中的“负负得正”可以用以下代码来表示：

```
    println( !( !a) );
```

`!(!a)` 的值与 a 的值永远是一样的。这就是当年布尔先生想要的“把人的思维用明确的符号表达出来”。人们经常把数字之间大小的比较结果储存在布尔型变量内，如：

```
boolean a= 5<3*2;
boolean b= 5<3-2;
boolean c= (5==2+3);
println(a+","+b+","+c);
```

下面我们来解决一个数学问题：在 1~10000 的整数中，有哪些数字是 3 的倍数而且它的平方能被 1787 整除？如果用笔算工作量非常大，但如下程序可以瞬间给出答案：

```
for (int a=1; a<=10000; a++) {
    boolean s = 0==(a*a) %1787;
    boolean t = 0==a %3;
    if ( s && t)
        println(a);
}
```

代码用到了`&&`（并且）运算符。大家要注意`=`和`==`之间的区别，`=`是赋值符号（把右边的值赋给左边的变量）；而`==`是比较符号（得到的结果是布尔型值，即 `true` 或 `false`），不会改变变量的值。

回到两个圆形的程序，我们将鼠标是否在第一个圆圈内的判断结果储存在变量 `in1` 中，将鼠标是否在第二个圆圈内的判断结果放在变量 `in2` 中。当两个条件都满足时 (`in1&& in2`)，可使两个圆圈放大或缩小，如图 3-3 所示。

```

float x1=220;
float y1=250;
float r1=180;
float x2=600;
float y2=500;
float r2=400;
void setup() {
    size(500, 700);
}
void draw() {
    background(255);
    boolean in1=r1>dist(x1, y1, mouseX, mouseY);
    boolean in2=r2>dist(x2, y2, mouseX, mouseY);
    if (in1 && in2) {
        r1+=0.5;
        r2-=0.4;
    }
    fill(0, in1?0:255, 0, 100);
    ellipse(x1, y1, 2*r1, 2*r1);
    fill(0, 0, in2? 0:255, 100);
    ellipse(x2, y2, 2*r2, 2*r2);
}

```

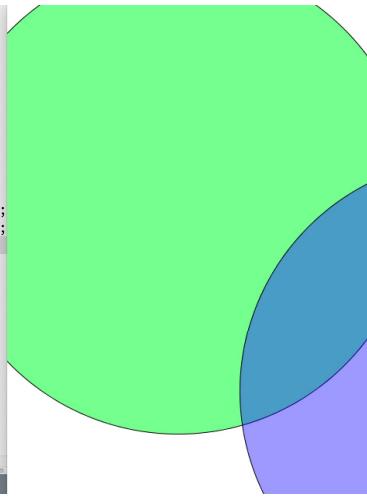


图 3-3 鼠标的位置决定了圆的放或缩小

相对于圆形，判断点在矩形内部就有些复杂了。在 3.1 节中，小球与四个边框的碰撞条件需要分别写清楚。首先，我们用四个数字来定义一个矩形的范围，程序如下：

```

float xa=100;
float ya=150;
float xb=320;
float yb=500;
void setup() {
    size(400, 600);
}
void draw() {
    background(255);
    rect(xa, ya, xb-xa, yb-ya);
}

```

`xa` 和 `ya` 是矩形左上角的坐标，而 `xb` 和 `yb` 是矩形右下角的坐标。因此，矩形的宽度是 `xb-xa`，高度是 `yb-ya`。鼠标需要满足四个条件：在左边框的右边，在右边框的左边，在上边缘的下面，在下边缘的上面，这样才能说明它在矩形

内。程序如图 3-4 所示。

```
float xa=150;
float ya=100;
float xb=500;
float yb=320;
void setup() {
    size(600, 400);
}
void draw() {
    background(255);
    fill(0,255,255);
    if(xa<mouseX && mouseX<xb && ya<mouseY && mouseY<yb){
        fill(255,0,0);
    }
    rect(xa, ya, xb-xa, yb-ya);
}
```

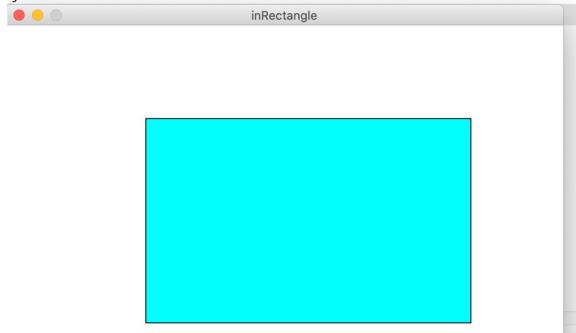
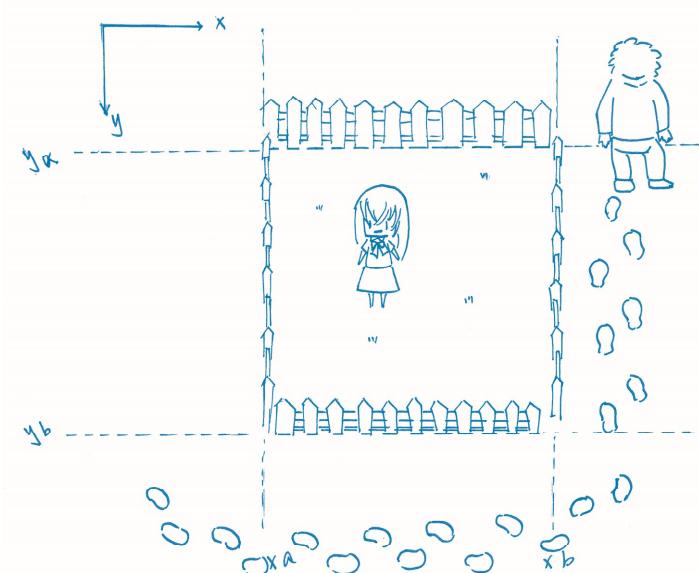


图 3-4 矩形的颜色反映鼠标的位置是否在矩形内



然而，“程序猿”有另外的思路：“在矩形内 == !在矩形外”，即判断点是否在矩形外就能判断它是否在矩形内。因此“程序猿”写了如下的奇怪代码：

```
boolean outx= (xa-mouseX)*(xb-mouseX)>0;
boolean outy= (ya-mouseY)*(yb-mouseY)>0;
if( !(outx||outy)){
    fill(255,0,0);
}
```

其中 $(xa-mouseX)*(xb-mouseX)>0$ 概况了两种情况：鼠标在左边框左侧，或者在右边框右侧（ $xa-mouseX$ 与 $xb-mouseX$ 的符号相同）。

最后介绍一种布尔型表达式，由? 和: 组合而成，譬如：

```
int x = a? 0 : 1;
```

如果 a（必须是 boolean 型的数据）的值为 true，右侧表达式的值为 0；如果 a 为 false，右侧表达式的值为 1。再试一下以下代码：

```
for (int i=0; i<5; i++) {
    boolean a = i%3==0;
    String x = a? "to be" : "not to be" ;
    println(i, x);
}
```



练习题：把 3.1 节中运动小球的坐标替换为本节中的 `mouseX`, `mouseY`。

布尔先生的思考题：预测下面这个程序的打印结果。

```
for(int i=0;i<4;i++){
    boolean a= i%2==0;
    boolean b= i/2==0;
    boolean compare= !(a&&b) == !a || !b;
    println(a+","+b+": "+compare);
}
```

如何用日常用语来解释 `!(a&&b)` 和 `!a || !b` 的关系？

3.3 好多弹球

与用纸笔画图不同，用程序来画图可以批量处理大量重复性的动作。譬如，我们之前已经用了 `for` 循环来反复运行一段代码，而 `for` 循环经常需要和数组 (`array`) 一起使用才更有效。我们可以把数组想象成一排箱子，里面装的是同一类东西，譬如：

```
boolean [] a = new boolean [5];
```

上面的程序定义了五个箱子（元素），里面只能装 `boolean` 型的值。第一个箱子的编号是 0，第二个的箱子的编号是 1，以此类推，程序可写为：

```
a[0]=true;
a[1]=false;
a[2] = a[0]==a[1];
a[3] = a[0]!=a[1];
a[4] = a[2] || a[3];
```



赋值给数组的某个元素，就相当于把东西放进该元素对应的箱子。当然，每个箱子里面的东西以后还可以替换，就像变量的值可以不断修改一样。在程序员的世界里，数是从 0 开始的。数组的 `length` 属性表示数组的长度（箱子的个数），对应程序如下：

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

这样，就可以把所有箱子里的内容依次打印出来。如果我们预先知道每个箱子里的值，可以在一行代码中同时声明数组并完成元素的赋值，代码如下：

```
boolean [] a= { true, false, false, true, true };
```

`for` 循环可以很方便地赋值给数组的元素，以下代码把函数 $y=\sin(x)/x$ 的结果储存在一个 `float` 型的数组里，如下所示：

```
float[] a = new float[300];
for (int i=0; i<a.length; i++) {
    float x= 0.1*( i-0.5*(a.length-1));
    a[i] = cos(x)/x;
}
```

下面我们把 $\sin(x)/x$ 和 $\cos(x)/x$ 的值分别赋予数组 **a** 和数组 **b**, 随后在 **draw()** 中把两个数组的值显示出来。数组 **a** 和数组 **b** 的长度均为 300 (包含 300 个数字), 而显示窗口的宽度是 600, 将数组 **a** 的值画在偶数列上, 将数组 **b** 的值画在奇数数列上, 如图 3-5 所示。

```
float[] a = new float[300];
float[] b = new float[300];
void setup() {
    size(600, 400);
    for (int i=0; i< a.length; i++) {
        float x= 0.1*(i-0.5*(a.length-1));
        a[i] = cos(x)/x;
        b[i] = sin(x)/x;
    }
}
void draw() {
    background(255);
    for (int i=0; i< a.length; i++) {
        line( 2*i, 0, 2*i, 200+200*a[i]);
        line( 2*i+1, 400, 2*i+1, 200+200*b[i]);
    }
}
```

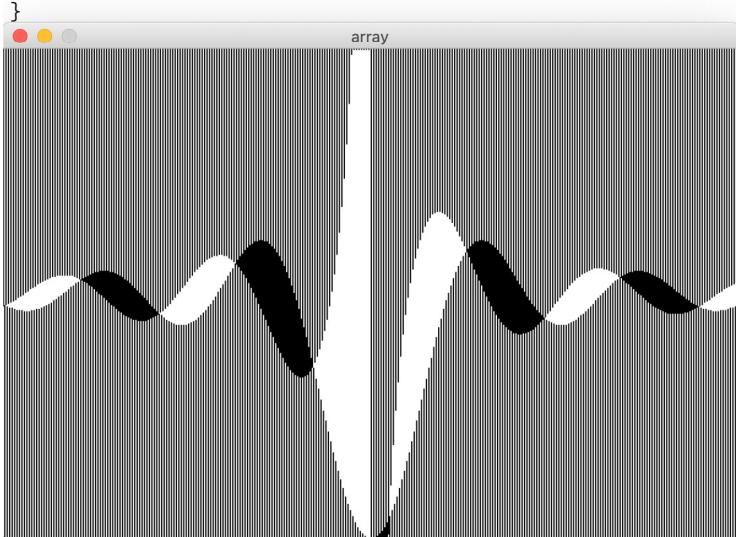


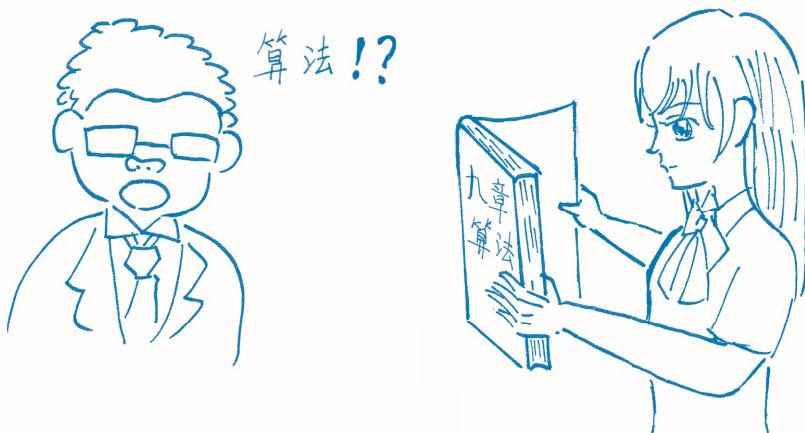
图 3-5 两组波的叠加

因为 **a[i]** 与 **b[i]** 的值很小 (基本在 -1~1), 所以我们在绘图时把 **a[i]** 与 **b[i]** 的值在垂直方向放大了 200 倍, 基本撑满了整个画面。图 3-5 所示的程序展示了在 Processing 中使用数组的一种常见方式: 先给数组赋值 (只需要进行一次,

因此写在 `setup()` 中), 然后把数组中的数据绘制出来(重复运行, 因此写在 `draw()` 中)。

使用数组的难点是正确处理数组的长度 (元素的总数) 及元素的编号。南宋数学家杨辉在 1261 年所著的《详解九章算法》发明了下面这样的数字阵列, 下面我们尝试用数组来生成杨辉三角。

```
1  
1 2 1  
1 3 3 1  
1 4 6 4 1
```



如果我们把每一行看作一个数组, 那么每一行数组的长度总是比上一行增加 1。另外, 每个数是上一行与之相邻的两个数之和 (头和尾除外)。因此, 如果上一行的数组是 `int[] a`, 那么下一行的数组 `b` 可以通过如下程序来创建:

```
int[] b= new int[a.length+1];  
for (int i=0; i<a.length; i++) {  
    b[i]+=a[i];  
    b[i+1]+=a[i];  
}
```

有两点需要说明:

(1) 一个整数型数值在刚创建时（第一行代码），每个元素都是 0。`float` 型的数组也一样，`boolean` 型数组默认值是 `false`。

(2) 该程序很小心地处理了数组元素的编号（特别是 `b[i+1]+=a[i];` 这一句），把头尾的特殊情况都包含了。杨辉三角完整的代码和运行结果如图 3-6 所示。

```

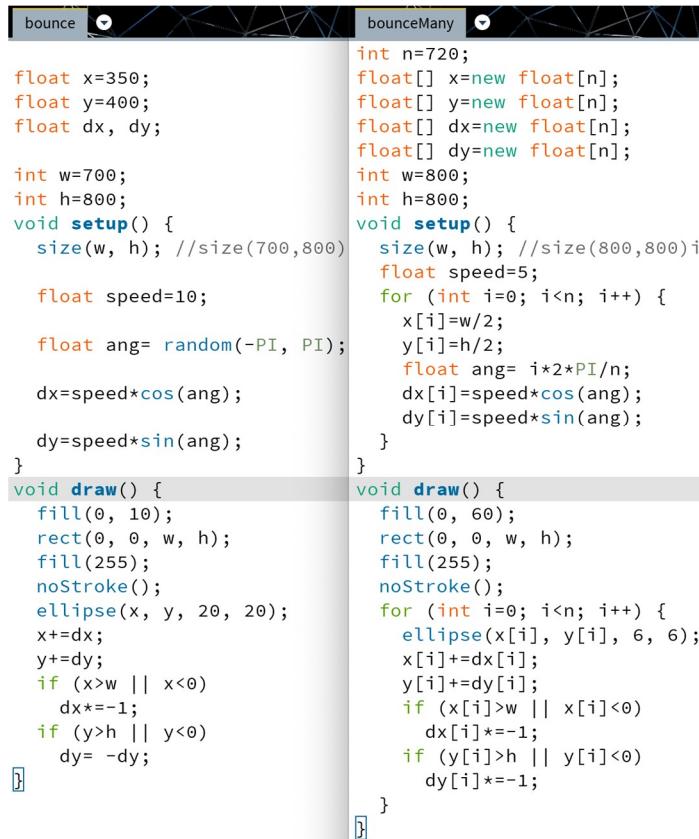
int[] a= {1};
int[] b;
void setup() {
    frameRate(1);
}
void draw() {
    for(int v:a)
        print(v+" ");
    println();
    b= new int[a.length+1];
    for (int i=0; i<a.length; i++) {
        b[i]+=a[i];
        b[i+1]+=a[i];
    }
    a=b;
}

```

图 3-6 杨辉三角完整的代码和运行结果

`a=b;` 的意思是：把下面一行（数组 `b`）的值赋给数组 `a`，这样在下一轮计算中数组 `a` 的值就整体更新了。

熟悉了数组的使用之后，我们可以改写 3.1 节的弹球程序，把一个小球变成很多小球。为了方便对照，图 3-7 把两组代码叠在了一起。



```

bounce
float x=350;
float y=400;
float dx, dy;

int w=700;
int h=800;
void setup() {
    size(w, h); //size(700,800)

    float speed=10;

    float ang= random(-PI, PI);
    dx=speed*cos(ang);
    dy=speed*sin(ang);
}

void draw() {
    fill(0, 10);
    rect(0, 0, w, h);
    fill(255);
    noStroke();
    ellipse(x, y, 20, 20);
    x+=dx;
    y+=dy;
    if (x>w || x<0)
        dx*=-1;
    if (y>h || y<0)
        dy=-dy;
}

```



```

bounceMany
int n=720;
float[] x=new float[n];
float[] y=new float[n];
float[] dx=new float[n];
float[] dy=new float[n];
int w=800;
int h=800;
void setup() {
    size(w, h); //size(800,800);
    float speed=5;
    for (int i=0; i<n; i++) {
        x[i]=w/2;
        y[i]=h/2;
        float ang= i*2*PI/n;
        dx[i]=speed*cos(ang);
        dy[i]=speed*sin(ang);
    }
}

void draw() {
    fill(0, 60);
    rect(0, 0, w, h);
    fill(255);
    noStroke();
    for (int i=0; i<n; i++) {
        ellipse(x[i], y[i], 6, 6);
        x[i]+=dx[i];
        y[i]+=dy[i];
        if (x[i]>w || x[i]<0)
            dx[i]*=-1;
        if (y[i]>h || y[i]<0)
            dy[i]*=-1;
    }
}

```

图 3-7 单个弹球程序与多个弹球程序的对比

图 3-7 左侧是 1 个弹球的代码，右侧是 720 个弹球的代码。原来的 `float x`（一个数）变成了 `float[] x`（一个数组），原来的速度 `dx`（一个数）也变成了数组 `dx`。从一个小球变成几百个小球，代码却只增加了一点点，这是因为我们很好地利用了数组与 `for` 循环。程序运行的部分截图如图 3-8 所示。

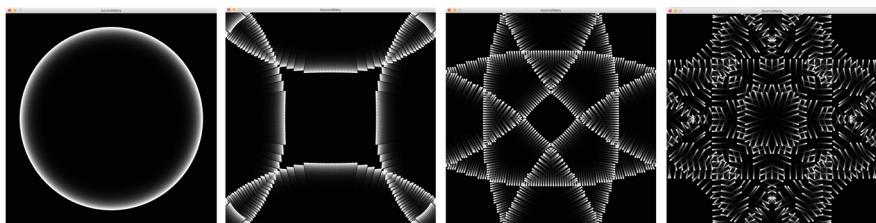


图 3-8 “幻影版”弹球程序运行过程的部分截图

在进入下一章之前，我们先回顾一下三种数据类型，如下：

```
int a = -5;  
float b = 3.14;  
boolean c = true;
```

布尔型数据的四种运算符为：!、||、&&， ==。

判断语句的两种常见方式如下：

```
if ( // condition A ) {  
    // codes A  
}
```

```
if ( // condition A ) {  
    // codes A  
} else {  
    // codes B  
}
```

定义数组的两种方式如下：

```
float[] arr = new float[64];  
int[] five = {1, 2, 3, 4, 5};
```

for 循环的两种方式如下：

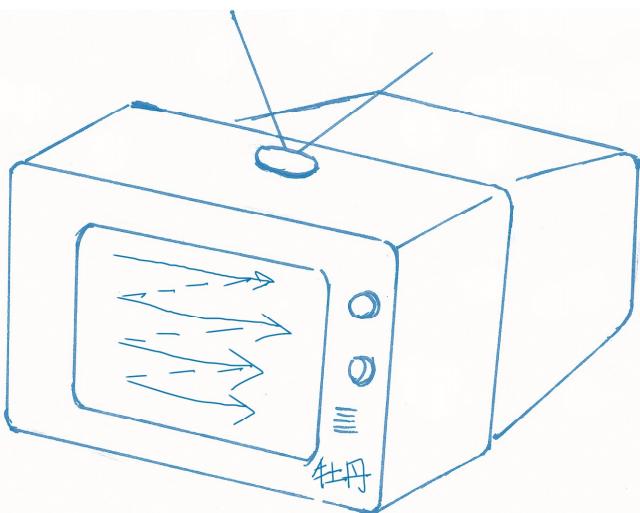
```
for (int i = 0; i < arr.length; i++) {  
    println(arr[i]);  
}  
for (float v : arr) {  
    println(v);  
}
```

第4章

弹！弹！弹！

4.1 彩色噪声

大家是否想过面前的计算机屏幕是如何显示千变万化的图像的？如果凑近屏幕仔细观察，会发现屏幕是由很多点构成的，一个点发光就构成了整幅图像中的一个像素（pixel）。如今显示器的分辨率一般都在 1920×1080 左右，即包含 200 多万个像素。以前的老式黑白电视机采用逐行扫描的方式来显示图像：电子束先从左上角向右移动，然后跳跃到第二行，以此类推一行一行向下扫描。电子束打在每个位置上时的强度不同，从而在屏幕上显示出灰度图像。这个过程非常快，因此人的眼睛感受到的是连续的画面。



Processing 处理像素的方式和老式黑白电视机的扫描原理类似。如果设定窗口 `size(4,3)`, 即有 12 个像素被放置在一个称为 `pixels` 的数组里 (Processing 在后台自动创建, 不需要自己写代码)。这 12 个像素在屏幕上的排列方式如下。

第0行	0	1	2	3
第1行	4	5	6	7
第2行	8	9	10	11

编辑像素时需要与 `loadPixels()` 和 `updatePixels()` 配合使用, 格式为:

```
loadPixels();
pixels[0]= color( 0, 0, 255);
pixels[1]= color( 0);
pixels[2]= color( 0, 255, 0);
//...
updatePixels();
```

编程的时候常常需要将行、列的序号转换为 `pixels` 中元素的序号, 公式为:

$$\text{元素序号} = i * w + j$$

其中, `i`、`j` 为行、列的序号; `w` 为屏幕的宽度。下面的代码采用了这种编号规律来生成彩色条纹:

```
int w=256;
int h=256;
void setup() {
    size(w, h); //或 size(256,256)
    colorMode(HSB);
}
void draw() {
    loadPixels();
```

```

for (int i=0; i<h; i++)
    for (int j=0; j<w; j++)
        pixels[i*w + j]= color(j, 255, 255);
    updatePixels();
}

```

这里出现了两个嵌套的 `for` 循环。第一个 `for (int i=0; i<h; i++)` 循环遍历了所有行，第 2 个 `for (int j=0; j<w; j++)` 循环遍历了所有列。这两个循环的运行过程和老式黑白电视机扫描屏幕的方式一样：

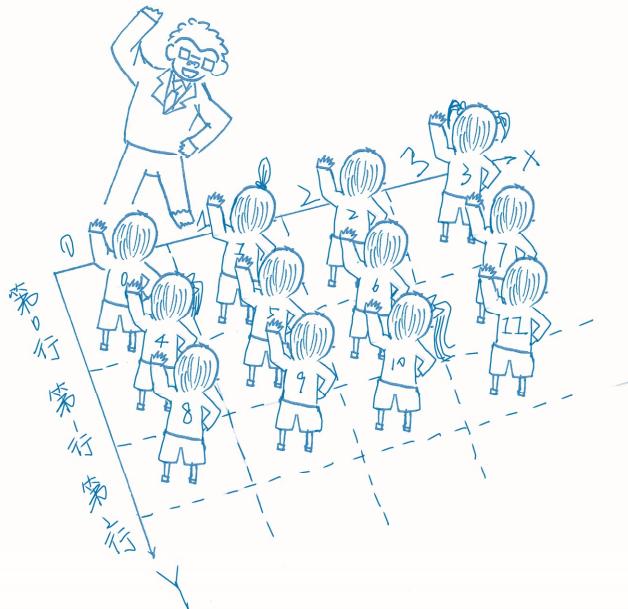
$i=0$ 时， $j=0, j=1, j=2, \dots$ 直到 $j=h-1$ (第一行走完)；

$i=1$ 时， $j=0, j=1, j=2, \dots$ 直到 $j=h-1$ (第二行走完)；

.....

$i=h-1$ 时， $j=0, j=1, j=2, \dots$ 直到 $j=h-1$ (最后一行走完)。

在处理颜色时，代码 `pixels[i*w + j]= color(j, 255, 255);` 把窗口内像素的色相 (hue) 与列数 j 关联，这里大家可以编写自己的涂色方式。



下面我们用 Processing 内置的 `noise()`方法来为窗口填色。20 世纪 80 年，纽约大学的佩林（Ken Perlin）发明了一种数学方法来模拟云朵、水面、石材等的不规则纹理，随后在计算机图形学领域广泛采用，被称为 Perlin noise（噪声）。为了与 Processing 的 `noise(x,y)`方法对接，首先需要把行和列的序号转化为一定范围内的小数，程序如下：

```
float x= map(j, 0, w-1,0,1);
float y= map(i, 0, h-1,0,1);
float v= noise(x,y);
pixels[i*w + j]= color( 230*v, 255, 255);
```

本来 `j` 在 `0~w-1` 范围内变化的，而对应的 `x` 在 `0` 到 `1` 之间变化。我们可以很方便地查看 `map()` 方法的详细解释，在程序中选中 `map`，右击鼠标选中“`find in reference`”，对应的详细文档就会跳出来。大家可以用同样的方式查看 `noise()` 方法的文档。获得合适范围内的 `x`、`y` 之后，计算对应的噪声值 `v= noise(x,y)`，最后与像素的色相对应起来。

有趣的是，程序每次运行的结果都不一样，如图 4-1 所示。如果想要每次运行结果一样，需调用 `noiseSeed()` 方法。Perlin 噪声的另一个特点是：不同的 `x`、`y` 取值范围将得到大小不同的纹理，如把参数值放大 3 倍，程序如下：

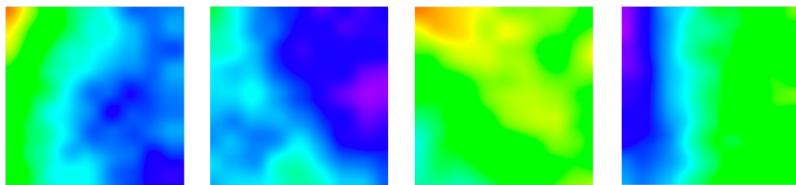


图 4-1 每次运行程序产生的随机图像都不同

```
noiseSeed(66);
for (int i=0; i<h; i++) {
    for (int j=0; j<w; j++) {
        float x= map(j, 0, w-1,0,1);
        float y= map(i, 0, h-1,0,1);
        float v= noise(3*x,3*y);
```

```

pixels[i*w + j] = color( 230*v, 255, 255);

}
}

```

得到的图像会很不一样。图 4-2 所示中 4 个截图的放大比例分别是 1、3、9、27。

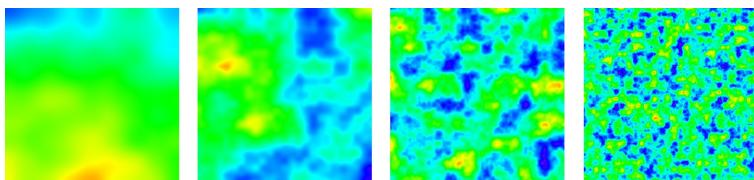


图 4-2 不同比例下的 noise 图案

简单来说， x 与 y 范围的越小，纹理就越平滑；反之则纹理越粗糙（实际上是把观察范围扩大了）。利用逐渐递增的 `frameCount` 值可以动态显示不同尺寸下的彩色噪声，程序如图 4-3 所示。

```

int w=720;
int h=720;
void setup() {
    size(w, h); //size(720,720)
    colorMode(HSB);
}
void draw() {
    loadPixels();
    for (int i=0; i<h; i++) {
        for (int j=0; j<w; j++) {
            float x= map(j, 0, w-1, 0,1);
            float y= map(i, 0, h-1, 0,1);
            float sc=0.02*frameCount;
            float v= noise(sc*x, sc*y);
            pixels[i*w + j] = color( 230*v, 255, 255);
        }
    }
    updatePixels();
}

```

图 4-3 利用 `frameCount` 改变 noise 图案的比例

Processing 内部编写了很多好用的方法，譬如：

```
map(value, start1, stop1, start2, stop2)
dist(x1,y1, x2, y2)
noise(x,y)
```

可以在 Processing 3 的官方文档上查阅，Processing 2 的绝大部分文档与 Processing 3 相同。

4.2 自定义方法

4.1 节介绍了组合使用 `loadPixels()`、`pixels` 数组、`updatePixels()` 的绘图方式，对窗口中的每个像素直接赋色。下面我们用 `sin` 函数来指定所有像素的颜色，程序如下：

```
C42_1:
int w=720;
int h=720;
void setup() {
    size(w, h); //size(720,720)
    colorMode(HSB);
}
void draw() {
    loadPixels();
    float r =exp(0.005*frameCount);
    for (int i=0; i<h; i++) {
        for (int j=0; j<w; j++) {
            float x= map(j, 0, w-1, -r, r);
            float y= map(i, 0, h-1, -r, r);
            float hue= map(sin(x*y), -1, 1, 10, 150);
            pixels[j+w*i]= color(r, hue, 100);
        }
    }
    updatePixels();
}
```

```

pixels[i*w + j] = color( hue, 255, 255);
}
}
updatePixels();
}

```

上述代码采用了两个嵌套的 `for` 循环来遍历所有像素（`i` 为行序号，`j` 为列序号），行和列的序号被转换成数组元素的编号，即 `pixels[i*w+j]`。代码中用到了三次 `map()` 方法，它把一个数从原来的取值范围映射到一个新的取值范围。譬如，`sin(x*y)` 本来的取值范围是从 -1 到 1，而代码 `hue = map(sin(x*y), -1, 1, 10, 150)` 把 `hue` 的取值范围变成了从 10 到 150（该数值用于确定颜色）。为了使整个画面连续缩小（观察范围变大），代码 `r = exp(0.005*frameCount)`；产生了一个逐渐变大的值 `r`，作为 `x`、`y` 的取值范围。程序运行的部分截图如图 4-4 所示。

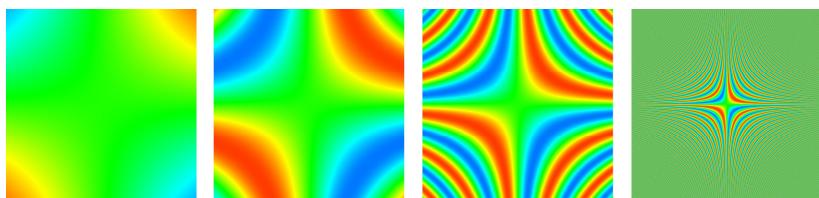
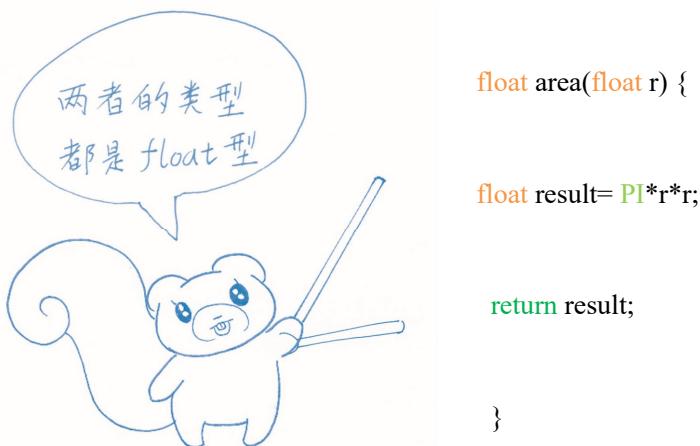


图 4-4 `sin` 函数产生的动画截图

为了更高效地写代码，经常需要定义自己所需的方法（method）。譬如我们需要计算圆的面积，就可以编写以下方法：



声明方法的第一行要写明返回数据的类型(这里是 `float` 型)、方法的名字、输入参数的类型和个数(这里只有一个 `float` 型参数)。方法内的最后一句“`return`”后面的数据类型必须与第一行写明的返回类型相同。一种更简洁的写法是：

```
float area(float r) {
    return PI*r*r;
}
```

如果要计算矩形的面积，可以再定义一个方法：

```
float area(float w, float h) {
    return w*h;
}
```

这两个方法的名字都是 `area`，但两者的参数不同，Processing 会把它们看作两个互不相干的方法，这种情况称为方法的重载(`overloading`)。在这个例子中，我们可以简单地把方法(`method`)理解为数学函数，但程序中的方法还可以做很多其他事，未必是为了进行数学运算。如果你发现一段代码在程序里出现了两次以上，就应该考虑把这段代码写在一个方法内。并不是所有的方法都有返回类型，下面的程序创建了一个用来打印小数数组的方法。

```
import java.text.DecimalFormat;
DecimalFormat df= new DecimalFormat ("#.##");
void setup() {
    float[] pis= {PI, 2*PI, 3*PI};
    print(pis);
}
void print(float[] arr) {
    for (int i=0; i<arr.length; i++)
        println( df.format(arr[i]));
}
```

`print()`方法最后一句没有用 `return`，因此在声明方法的第一行开头要写 `void`(表示该方法不返回任何数据)。代码中的 `DecimalFormat ("#.##")`可以使小数保

留两位小数，如果要保留三位小数就写 DecimalFormat ("#.###")，以此类推。为了使用 Java 库里的 DecimalFormat 类，需要在程序开头写 import java.text.DecimalFormat 来导入这个类。Processing 基于 Java 语言，该语言于 1996 年发布，后由甲骨文（Oracle）公司管理，可以在其网站上找到 Java 语言的详细文档。

Java 方法还能返回复杂的数据类型，如数组。下面的方法返回了给定范围内的一串整数：

```
void setup() {
    println(make_array (-2, 3));
}

int[] make_array (int min, int max) {
    int[] a=new int[max-min+1];
    for (int i=0; i<a.length; i++)
        a[i]= i+min;
    return a;
}
```

make_array 方法在开头写明了返回类型是 int 型，在方法的最后一行要返回一个 int 型的数据。

熟悉了自定义方法后，我们再次回到屏幕像素的编程，把程序 C42_1 中的 draw() 方法改写为：

```
void draw() {
    loadPixels();
    float r =exp(0.005*frameCount);
    for (int i=0; i<h; i++)
        for (int j=0; j<w; j++)
            pixels[i*w + j]= colorAt(i, j, r);
    updatePixels();
}
```

```

color colorAt (int i, int j, float r) {
    float x= map(j, 0, w-1, -r, r);
    float y= map(i, 0, h-1, -r, r);
    float hue= map(sin(x*y), -1, 1, 10, 150);
    return color(hue, 255, 255);
}

```

程序运行的结果不变，只是把计算像素颜色的代码集中放在自定义的 `colorAt()` 方法内。该方法的返回类型是 `color`，需要输入三个参数（像素的位置和参数 `r`）。`colorAt()` 方法可以用来生成极其复杂的图像，在 8.3 节“悲情朱利亚”中我们会继续这个话题。

4.3 滤镜与点彩

Processing 处理图片与处理展示窗口像素（`pixels` 数组）的方式是基本相同的。Processing 可以很方便地导入或导出图片，在程序中用 `PImage` 型表示图片。首先创建一个空程序并保存，再把心仪的图片放到程序文件夹中（与.pde 文件并列），然后就可以用 `loadImage("****.jpg")` 来导入图片了，程序如下：

```

PImage img; //声明变量 img
int w=800; //图片宽度
int h=627; //图片高度
void setup(){
    size(w,h);
    img = loadImage( "alp.jpg" );
}
void draw() {
    image(img,0,0);
}

```

如果预先知道图片的长度和宽度，在程序的开头就可以定义窗口的宽度 w 和高度 h，这样 `draw()` 显示的图片就正好充满窗口。显示图片用 `image(img,x,y);` 命令，其中的 x、y 表示图片左上角的坐标。

我们可以使用 `for` 循环，让第 num 行以下的像素颜色等于第 num 行的像素颜色，代码及运行效果如图 4-5 所示。

```
PImage img; //declare a variable
int w=800; //width of image
int h=627; //height of image
void setup(){
    size(w,h);
    img = loadImage( "alp.jpg");
}
void draw(){
    color[] ps = img.pixels;
    int num=h-frameCount;
    for(int i= num+1; i< h; i++) // row
        for(int j=0; j< w; j++) //col
            ps[ i*w +j ] =ps[ num*w +j ];
    img.updatePixels();
    image(img, 0,0);
}
```

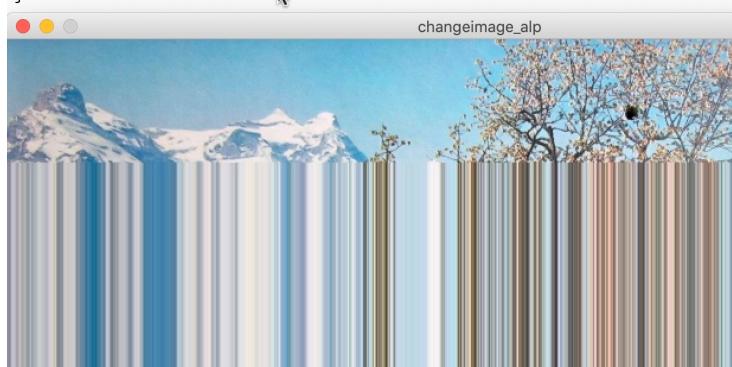


图 4-5 垂直条纹逐渐侵蚀图片

在这里 num 这个数会随着帧数（`frameCount`）的增长而变小。在操作像素时，第一个 `for` 循环对应每一行，而第二个 `for` 循环对应每一列。其中，`ps[i*w+j] = ps[num*w+j]` 把第 num 行的像素颜色赋予了第 i 行的像素，i 的范围是从 num+1 到图片像素最后一行(h-1)。

Processing 内部定义了不少滤镜，如 `img.filter(BLUR,r)` 对图片进行模糊处理，参数 r 是模糊的半径（越大越模糊）。`img.filter(INVERT);` 把图片反相，即把

每个像素的颜色换成它的补色。滤镜还可以进行叠加，如图 4-6 所示的代码先对图片进行反相，再进行模糊处理。

```
PImage img;
int w=600;
int h=450;
void setup() {
    size(w, h); //size(600,450)
    img=loadImage("hot.jpg");
    img.filter(INVERT);
    img.filter(BLUR,20);
}
void draw() {
    image(img, 0, 0);
}
```

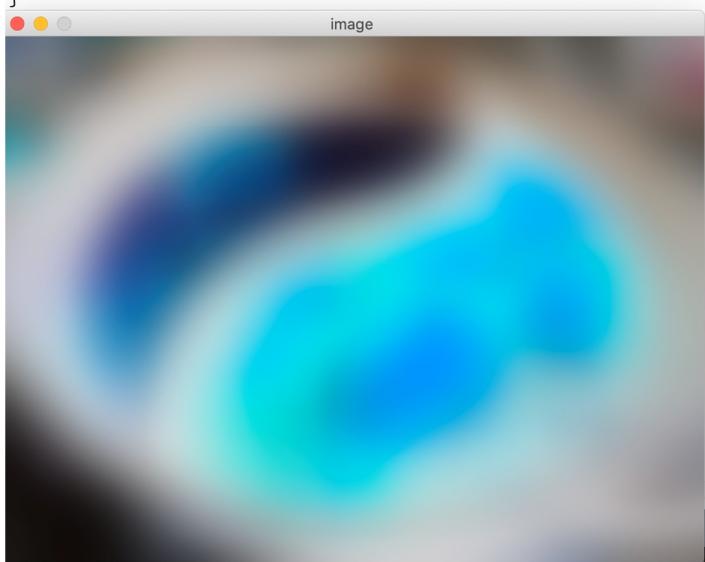


图 4-6 将图片反相再增加模糊滤镜

滤镜代码写在 `setup()` 方法中，因此滤镜的作用仅发挥一次。如果把滤镜代码写在 `draw()` 方法中，图片就会连续地发生变化。

我们经常需要把自己的作品保存成图片，这时就会用到 `keyPressed()` 方法。首先通过下面这段简短的代码来了解 `keyPressed()` 方法：

```
boolean active=false;
void setup() {
    size(600, 450);
```

```
}

void draw() {
    background(0, active? 255:0, 0);
}

void keyPressed() {
    active= !active;
}
```

其中，`background(0, active? 255:0, 0)`是一种浓缩式代码，它相对于以下代码：

```
if(active)
    background(0, 255, 0);
else
    background(0, 0, 0);
```

按下键盘上的任意键，窗口的颜色就会转换（绿色或黑色）。程序开头定义了 `boolean` 型变量 `active`，而按键会触发 `keyPressed()` 方法，进而执行 `active= !active`（把 `active` 的值反转）。窗口的背景颜色取决于 `active` 的值。



`setup()`、`draw()`和 `keyPressed()`三种方法被触发的机制各不相同。`setup()`方法在程序启动时（展示窗口弹出时）运行一次；而 `draw()`方法时时刻刻反复运行（约每秒 60 次）；`keyPressed()`只有在键盘被按下时才会运行一次。

想要把窗口显示的内容保存成图片，可以在 `keyPressed()`方法中调用 `save()` 方法，程序如下：

```
PImage img;
int w=600;
int h=450;
boolean saveNow=false;
void setup() {
    size(w, h); //size(600,450)
    img=loadImage( "hot.jpg" );
}
void draw() {
    image(img, 0, 0);
    img.filter(BLUR, 2);
}
void keyPressed() {
    if( 's' ==key || 'S'==key)
        save( "hot"+frameCount+ ".jpg" );
}
```

一旦按下 `s` 键，窗口内显示的内容就会被保存到文件夹内。`keyPressed()`方法内部的 `if` 语句用来判断 `s` 键（区分大小写）是否被按下，如果是就保存图片。可以多次按下 `s` 键，从而保存多张图片。为了不让后面保存的图片覆盖之前保存的图片，我们用 `frameCount` 给每张图片取一个不同的名称。

图片是由像素构成的，所有像素都存放在 `img.pixels` 数组里（`img` 是 `PImage` 型的实例）。`img.pixels` 中的每个元素（像素）都是一个 `color` 型的变量，我们可以提取它的红、绿、蓝成分，程序如下：

```

color col= img.pixels [i];
float r= red(col);
float g= green(col);
float b= blue(col);

```

改变这些颜色成分就能改变整张图片的样子，这种批量化处理图片的程序就是滤镜。如图 4-7 所示的程序实现了红 $< 255 - 3 \times$ 绿、绿 $< 0.7 \times$ 红的滤镜：

```

PImage img1, img2;
int w=600;
int h=450;
void setup() {
    size(w/2, h); //size(300,450)
    img1=loadImage("hot.jpg");
    img2=createImage(w, h, RGB);
    color[] ps1=img1.pixels;
    color[] ps2=img2.pixels;
    for (int i=0; i<ps1.length; i++) {
        color col= ps1[i];
        float r= red(col);
        float g= green(col);
        float b= blue(col);
        ps2[i]= color(255-3*g, 0.7*r, b);
    }
    img2.filter(BLUR, 7);
}
void draw() {
    background(255);
    image(img1, 0, 0, w/2, h/2);
    image(img2, 0, h/2, w/2, h/2);
}

```

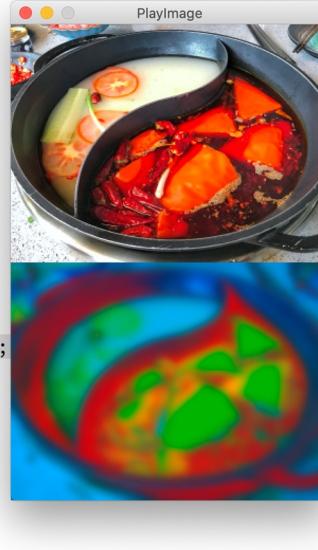


图 4-7 通过自定义的滤镜来改变图片

该程序导入了一张图片 `img1`，同时又创建了一张图片“`img2 = createImage(w, h, RGB);`”，把图片 `img1` 的像素颜色进行处理后赋给 `img2` 的像素。最后 `draw()` 方法把两张图片都显示出来。

我们可以在程序中自定义衡量颜色的函数（方法），如下面这个方法可以衡量颜色的红、绿、蓝成分之间的差异：

```

float dif(color col) {
    float r = red(col);
    float g = green(col);
    float b = blue(col);

```

```

    return abs(r-g)+abs(g-b)+abs(b-r);
}

```

一种颜色的 dif 值越大，说明颜色越鲜艳，反之颜色就越黯淡。

用像素来绘图的艺术传统可以追溯到 19 世纪末的巴黎。年轻的乔治·修拉（Georges Seurat）离开法国美术学院加入军队，归来后，他开始尝试一种“点彩”画法，创作了《大碗岛的星期天下午》，细看画面上都是杂乱的彩点，但远看却呈现出身临其境的光感。他不是在调色盘里混合颜色，而是让观察者的眼睛来混合颜色。

下面我们来编写一个交换像素的游戏：先选择任意一列中的任意两个像素（它们在一条竖线上），如果上方像素的鲜艳度高于下方像素，就交换这两个像素。这里的判断条件可以写为：

```

if( (i1-i2) * (dif(c1)-dif(c2)) < 0 ){
    //交换两个像素的颜色
}

```



其中，`i1`、`i2` 是两个像素所在行的序号，`c1`、`c2` 是两个像素的颜色。

如何随机地产生行的序号呢？我们可以利用 `random()`方法：

```
int i1= int(random(h));  
int i2= int(random(h));
```

有一点麻烦的是，`random()`返回的是 `float` 型数字（小数），而行的序号是 `int` 型数字（整数），因此需要用 `int()`方法把小数转化为整数。为了更好地组织代码，我们把交换像素的代码都放在 `swapPixels()`方法中，然后在 `draw()`方法中调用 `swapPixels()`方法，程序如下：

```
PIImage img;  
int w=600;  
int h=450;  
void setup() {  
    size(w, h); //size(600,450)  
    img=loadImage( "hot.jpg" );  
}  
void draw() {  
    for (int i=0; i<30; i++)  
        swapPixels();  
    image(img, 0, 0);  
}  
void swapPixels() {  
    for (int j=0; j<w; j++ ) {  
        int i1= int(random(h));  
        int i2= int(random(h));  
        color c1= img.pixels[ i1*w+j ];  
        color c2= img.pixels[ i2*w+j ];  
        if ( (i1-i2) * (dif(c1)-dif(c2))<0 ){  
            img.pixels[i2*w+j] = c1;
```

```

    img.pixels[i1*w+j] = c2;
}
}
img.updatePixels();
}

float dif(color col) {
    float r = red(col);
    float g = green(col);
    float b = blue(col);
    return abs(r-g)+abs(g-b)+abs(b-r);
}

```

程序运行时，可以看到，颜色鲜艳的像素会往下沉，颜色黯淡的会上浮，最后逐渐形成一张抽象画（见图 4-8）。整张图片的像素并没有发生变化，但这些像素的位置发生了变化。

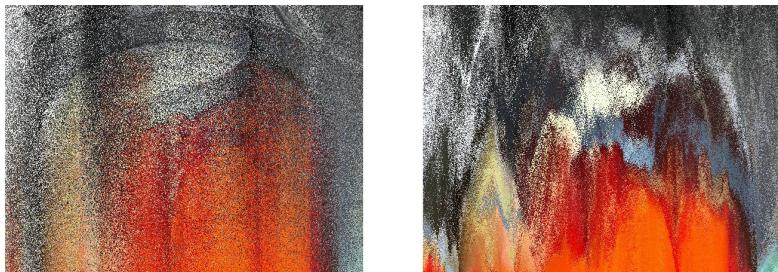


图 4-8 鲜艳的像素下沉，淡色的像素上浮

4.4 图像重绘

画素描时，我们要根据所观察物体每个局部的明暗来决定排线的深浅和疏密。基于一张图片，我们可以用程序来进行排线，形成一种类似素描的视觉效果。如果每个像素上都绘制一根线段，那整个画面会太密实，所以要间隔地进行像素采样与线段绘制：

```
for (int i=0; i<h; i+=2) {  
    for (int j=0; j<w; j+=2) {  
        // 像素采样与线段绘制  
    }  
}
```

注意：其中的 `i+=2` 和 `j+=2`。如果图片较大也可以采用更大的间距，如 `i+=4` 和 `j+=4`。提取图片上任意一点的颜色可以用代码 `color col= img.pixels[i*w+j];`，而获取颜色的亮度可以用 `brightness()` 方法。此外 `red()`、`green()`、`blue()` 方法可以提取颜色的红、绿、蓝成分。下面这个程序根据颜色的亮度来确定线段的长度，根据红色成分来决定线段的方向：

```
PImage img;  
int w=800;  
int h=627;  
void setup() {  
    size(w, h); //size(800,627)  
    img=loadImage( "alp.jpg" );  
    img.filter(BLUR, 2);  
}  
void draw() {  
    background(255);  
    stroke(0, 35);  
    for (int i=0; i<h; i+=2) {  
        for (int j=0; j<w; j+=2) {  
            color col= img.pixels[i*w+j];  
            float r= 0.08*(255-brightness(col));  
            float ang=red(col)*PI/255;  
            line(j, i, j+r*cos(ang), i+r*sin(ang));  
        }  
    }  
}
```

线段的方向在 $0^\circ \sim 180^\circ$ 之内变化，红色成分越多则角度越大：
`ang=red(col)*PI/255`。而 `r=0.08*(255-brightness(col))` 这句代码说明：像素越亮线段越短，0.08 这个系数可以根据图面效果进行调节。这句代码也可以写为：
`map(brightness(col), 0,255,20,0);`。

此外，程序对原始图片进行了模糊处理 `img.filter(BLUR, 2)`，可以使素描线条更整齐（同时将损失细节），模糊半径为 2 时产生的结果如图图 4-9 (a) 所示，模糊半径为 4 的结果如图 4-9 (b) 所示。此外，我们还可以把 `filter(BLUR, 2)` 写在 `draw()` 方法中，运行时图片会逐渐变得模糊，而素描效果也会随之变化，代码如图 4-10 所示。

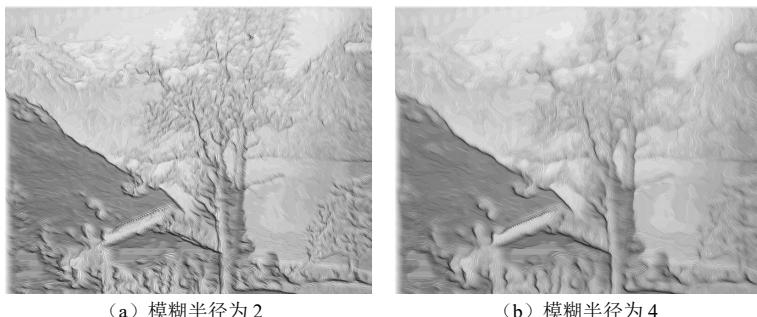


图 4-9 不同模糊半径的素描效果

```
PImage img;
int w=800;
int h=627;
void setup() {
    size(w, h); //size(800,627)
    img=loadImage("alp.jpg");
}
void draw() {
    background(255);
    stroke(0, 35);
    for (int i=0; i<h; i+=2) {
        for (int j=0; j<w; j+=2) {
            color col= img.pixels[i*w+j];
            float r= 0.08*(255-brightness(col));
            float ang=red(col)*PI/255;
            line(j, i, j+r*cos(ang), i+r*sin(ang));
        }
    }
    img.filter(BLUR, 2);
}
```

图 4-10 动态变化素描效果的代码

用程序生成素描的效果，相当于创建了一种图片滤镜。我们可以用手机拍摄一张照片，然后在 Processing 中进行处理，就能得到一张素描作品。我们不仅可以用线段来表现明暗，还可以用圆形、三角形、矩形等图元来重新绘制图片。下面这个程序就采用了最简单的圆形来绘制图片，采样的间距为 12（很稀疏）：

```
PImage img;  
int w=800;  
int h=627;  
void setup() {  
    size(w, h); //size(800,627)  
    img=loadImage( "alp.jpg" );  
}  
void draw() {  
    background(255);  
    fill(0);  
    noStroke();  
    for (int i=0; i<h; i+=12) {  
        for (int j=0; j<w; j+=12) {  
            color col= img.pixels[i*w+j];  
            float r= 15*sin(0.5*PI*brightness(col)/255);  
            ellipse(j, i, r, r);  
        }  
    }  
}
```



圆形的大小由像素的亮度决定：

```
float r= 15*sin(0.5*PI*brightness(col)/255);
```

因此得到的直径 r 是可正可负的。其中，系数 0.5 在很大程度上决定了整个图面的效果。系数为 0.5 时的效果如图 4-11 (a) 所示，系数为 1.4 时的效果如图 4-11 (b) 所示。比较图 4-11 中的两个图可以发现，明暗的分布规律大相径庭，这是 `sin()` 函数作用的结果。我们可以把这个关键系数设为 s ：

```
float r= 15*sin(s*PI*brightness(col)/255);
```

然后把 s 与鼠标位置对应起来：

```
float s=map(mouseX,0,w, 0.5,1.5);
```

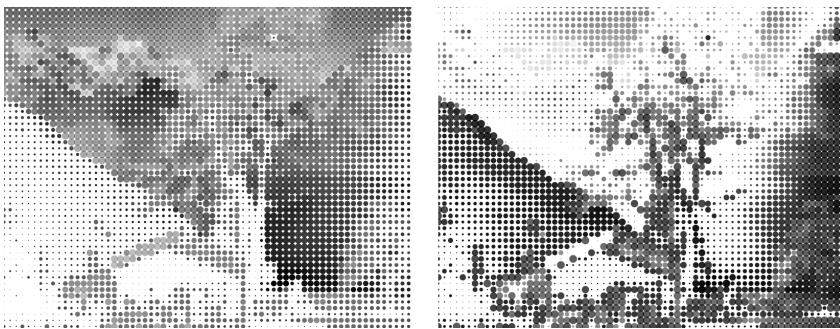


图 4-11 不同系数产生不同的画面效果

最终的代码如图 4-12 所示。

```
PImage img;  
int w=800;  
int h=627;  
void setup() {  
    size(w, h); //size(800,627)  
    img=loadImage("alp.jpg");  
    img.filter(BLUR, 2);  
}  
void draw() {  
    background(255);  
    fill(0);  
    noStroke();  
    float s=map(mouseX,0,w, 0.5,1.5);  
    for (int i=0; i<h; i+=12) {  
        for (int j=0; j<w; j+=12) {  
            color col= img.pixels[i*w+j];  
            float r= 15*sin(s*PI*brightness(col)/255);  
            fill(0, 255-red(col));  
            ellipse(j, i, r,r);  
        }  
    }  
}
```

图 4-12 鼠标控制图像的代码

该程序把（导入的原始图片的）像素的明暗转换为圆形的尺寸，而圆形的尺寸又决定了图像局部的明暗效果。这种转换可以用来改变原始图片的整体风格或细部特征。

