

```
size(720, 480);
bot1 = loadShape("robot1.svg");
bot2 = loadShape("robot2.svg");
bot3 = loadShape("robot3.svg");
landscape = loadImage("alpine.png");
}

void draw() {
// "landscape"を背景にします
// この画像はウィンドウと同じ大きさにする必要があります
background(landscape);

// 左右の間隔を設定し、イージングによって滑らかに移動させます
float targetOffset = map(mouseY, 0, height, -40, 40);
offset += (targetOffset - offset) * easing;

// 左のロボットを描きます
shape(bot1, 85 + offset, 65);

// 右のロボットは少し小さく描き、オフセットも少なくします
float smallerOffset = offset * 0.7;
shape(bot2, 510 + smallerOffset, 140, 78, 248);

// 一番小さいロボットを描きます。オフセットは最小です
smallerOffset *= -0.5;
shape(bot3, 410 + smallerOffset, 225, 39, 124);
}
```

7

動き

Motion

パラパラまんがと同じように、画面上のアニメーションも、まず1枚絵を描き、次にそれとは少し違う絵を描き、さらにまた……と繰り返していくことで作られます。滑らかな動きに見えるのは残像の効果で、わずかずつ異なる一連の画像を十分に速いレートで表示すると、我々の脳はそれが動いていると認識します。

Example 7-1:フレームレートを見る

滑らかな動きを生み出すために、Processingはdraw()内のコードを毎秒60回実行します。そして描かれる画面1枚1枚のことをフレームと呼びます。また、1秒間に何回フレームが更新されるかを表るのがフレームレートです。次のコードを実行するとフレームレートが表示されます。frameRate変数はプログラムのスピードを示しているともいえます。

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

Example 7-2:フレームレートの変更

frameRate()関数を使うとプログラムの実行スピードを変更できます。

```
void setup() {  
    frameRate(30); // 每秒30フレーム  
    //frameRate(12); // 每秒12フレーム  
    //frameRate(2); // 每秒2フレーム  
    //frameRate(0.5); // 2秒ごとに1フレーム  
}  
  
void draw() {  
    println(frameRate);  
}
```

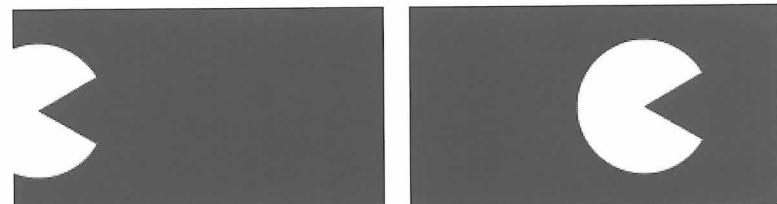
[NOTE] Processingは毎秒60フレームを維持しようとしていますが、draw()の実行に1/60秒以上かかると、フレームレートは低下します。frameRate()関数が指定しているのは最大フレームレートであって、実際のフレームレートはどんなコンピュータがどんなコードを実行しているかによって決まります。

スピードと方向

流れのような動きを表現するためにfloatと呼ばれるデータ型を使います。float型の変数は小数点を持つ数値を格納し、より高精度な動きを可能にします。int型を使って何かを動かすときは、フレームごとに少なくとも1ピクセル移動させないといけませんが、float型を使えば、1.01、1.02、1.03……と、わずかずつ変化させることで、いくらでも細かい動きに対応できます。

Example 7-3:図形の移動

左から右へ図形を移動します。変数xが位置を表しています。

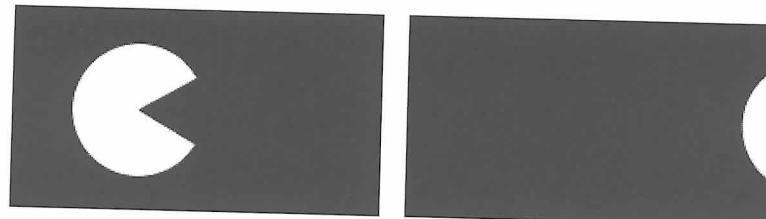


```
int radius = 40;  
float x = -radius;  
float speed = 0.5;  
  
void setup() {  
    size(240, 120);  
    ellipseMode(RADIUS);  
}  
  
void draw() {  
    background(0);  
    x += speed; // xに加算  
    arc(x, 60, radius, radius, 0.52, 5.76);  
}
```

このコードを実行してしばらくすると、xがウィンドウの幅を上回って、図形が右端から外へ出て行ってしまうことに気付くでしょう。xは増加を続け、図形はずっと見えないままです。

Example 7-4:円筒形を回す

違う動きを考えてみましょう。コードを拡張して、右端から出ていった図形が左端から戻ってくるようにします。回転する円筒形の側面を見ているような効果が得られます。



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

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

void draw() {
    background(0);
    x += speed; // xに加算
    if (x > width+radius) { // 図形が画面から消えたら
        x = -radius; // 左端に戻す
    }
    arc(x, 60, radius, radius, 0.52, 5.76);
}
```

draw()が実行されるたびに、図形の位置を表す変数xは増加します。xがウィンドウの幅に図形の半径を加えた値を超えたたら、xに負の値をセットします。そしてまたxに加算していきます。右端から出て左端から入ってくる動きはこのようにして実現できます（図7-1）。

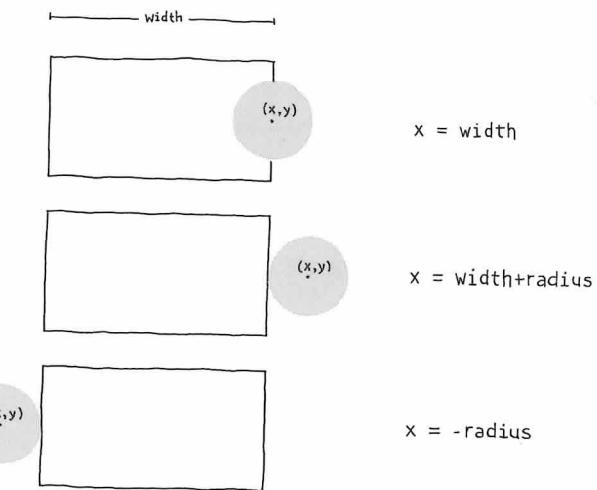
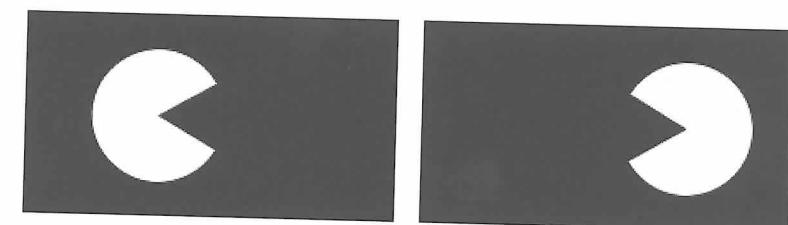


図7-1 ウィンドウの両端での処理

Example 7-5:壁に当たって跳ね返る

Example 7-3を拡張して、端に当たった図形が向きを変えて戻ってくるプログラムを作りましょう。図形の向きを記憶する変数を追加します。方向を表す値が1ならば右向き、-1ならば左向きです。



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

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

```

}

void draw() {
    background(0);
    x += speed * direction;
    if ((x > width-radius) || (x < radius)) {
        direction = -direction; // 方向を反転
    }
    if (direction == 1) {
        arc(x, 60, radius, radius, 0.52, 5.76); // 右向き
    } else {
        arc(x, 60, radius, radius, 3.67, 8.9); // 左向き
    }
}

```

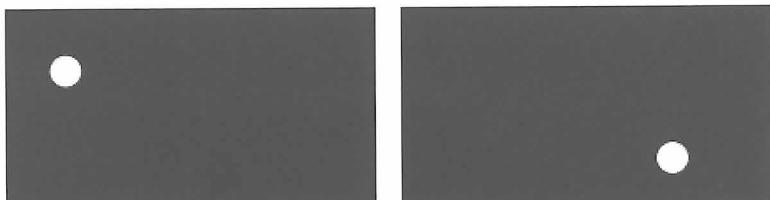
図形が端に達したら direction変数の符号を反転させて、進行方向が変わったことを表現します。direction変数が正なら負に、負なら正に変化します。

2点間の移動

画面上のある点から別の点へ図形が移動するアニメーションは、数行のコードで実現できます。始点と終点を指定し、その間の位置はフレームごとの計算で求めます。この処理をトゥイーニング(tweening)と呼ぶことがあります。

Example 7-6:2点間の経路を計算する

コードの再利用性を高めるため、冒頭に変数の宣言をまとめました。何度も値を変更しながら動かしてみて、このコードがどのように図形を動かすか観察してみましょう。step変数の値を変更すると、移動スピードを変えられます。



```

int startX = 20; // 始点 x 座標
int stopX = 160; // 終点 x 座標
int startY = 30; // 始点 y 座標
int stopY = 80; // 終点 y 座標
float x = startX; // 今の x 座標
float y = startY; // 今の y 座標
float step = 0.005; // ステップごとの移動量 (0.0 to 1.0)
float pct = 0.0; // 移動量 百分率 (0.0 to 1.0)

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

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

```

乱数

コンピュータグラフィックスの線形的でスムーズな動きと違って、現実世界の動きはたいでいいもっと不規則です。地面へ落ちていく枯葉や荒れた地面を這っている蟻を思い浮かべてください。乱数を生成することで、そうした意外性のある動きをシミュレートできます。random()は、設定した範囲内の乱数を作り出す関数です。

Example 7-7:乱数の生成

この短いプログラムは乱数をコンソールに出力します。値の範囲は0からマウスのx座標の間です。random()関数は必ず浮動小数点数を返すので、代入演算子(=)の左側の変数はfloat型です。

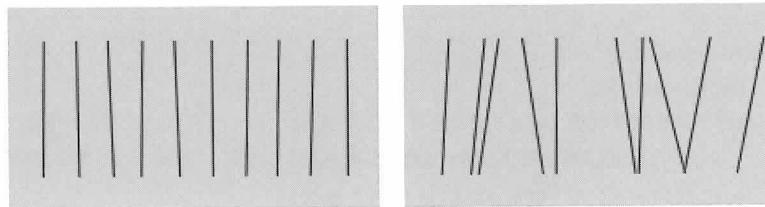
```

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

```

Example 7-8: ランダムに描く

Example 7-7がベースになっている次の例は、random()関数の値を使って線を描きます。マウスカーソルがウィンドウの左へ行くほど変化量は小さくなり、右へ行くほど乱数の効果が強調されます。random()関数はforループのなかにあるため、線を描くたびに新しい乱数が生成されます。



```

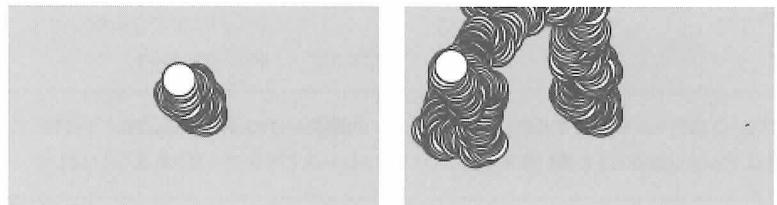
void setup() {
    size(240, 120);
}

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

```

Example 7-9: ランダムに動く

乱数を使って画面上の図形を動かすと、自然な見た目のイメージができるかもしれません。次の例では、円の位置は乱数によってフレームごとに変化します。background()関数を使っていないので、円が移動した跡が残ります。



```

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

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

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

```

このプログラムを長時間観察すると、円がウィンドウの外へ出て行き、また戻ってくることがあるかもしれません。これは偶然起こることですが、if文を追加するか constrain()関数を使うことで、ずっとウィンドウのなかにとどめておくこともできます。xとyの値がウィンドウの境界を越えないよう、constrain()関数による制限を加えるため、先ほどのコードのdraw()ブロックを、次のコードに変更してみましょう。

```

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

```

[NOTE] プログラムを実行するたびに `random()` が同じシーケンスを生成するよう強制したいときは `randomSeed()` を使います。詳しくはリファレンス (205 ページ) を見てください。

タイマー

Processing はプログラムがスタートしてからの経過時間をカウントしています。単位はミリ秒 (千分の一秒) で、1 秒が経過すると 1000、5 秒なら 5000、1 分では 60000 となります。このカウンタの値を返す関数が `millis()` で、時間に合わせてアニメーションを切り替えるトリガーとして使うことができます。

Example 7-10: 時間の経過

このプログラムを実行すると、実行開始からの経過時間がわかります。

```

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

```

Example 7-11: 時限式のイベント

`millis()` を `if` 文と組み合わせることで、イベントやアニメーションをシーケンスとして再生することができます。たとえば次の例では、2 秒経過ごとに、`if` ブロック内のコードが順番に実行されます。`timer1` と `timer2` の値が `x` を書き換えるタイミングを決めています。

```

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

```

```

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

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

```

円運動

三角関数が得意な人は、すでにサインとコサインの面白さをよく知っているでしょう。苦手な人は、次の例を試して好きになってください。数学的な説明は省いて、気持ちのいい動きを生み出すプログラムをいくつか紹介します。

図 7-2 はサイン波をグラフ化したもので、角度との関係を示しています。垂直方向の変化に注目すると、波の頂上と底で変化率が小さくなっています。ついには止まって方向を変えます。サイン波が持っているこの性質が興味深い動きを生み出します。

Processing の `sin()` 関数と `cos()` 関数は、指定した角度のサイン (正弦) とコサイン (余弦) を -1 から 1 の間の数として返します。`arc()` と同じように、角度はラジアンで指定してください (ラジアンについては Examples 3-7 を参照)。描画に使う場合は、`sin()` と `cos()` の値に大きな数を掛けることになるでしょう。

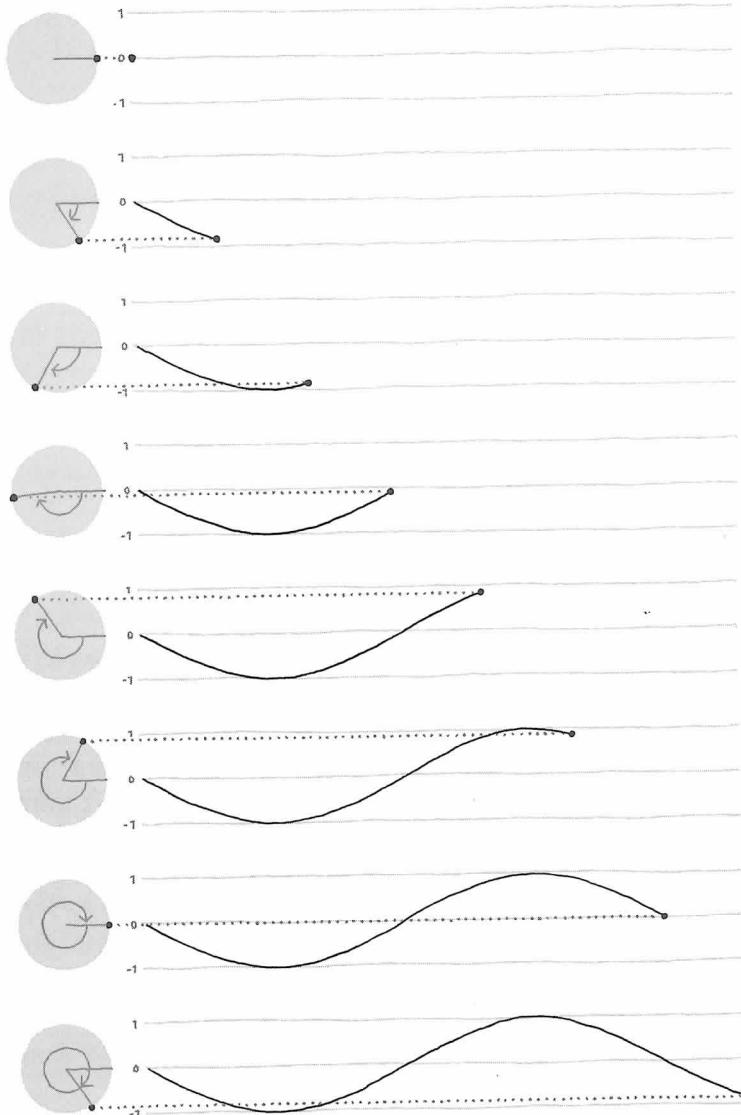


図7-2 サイン波

Example 7-12: サイン波の値

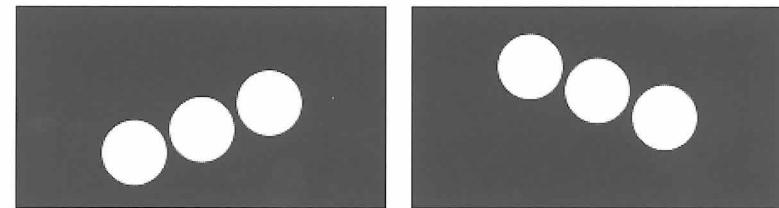
角度が増加すると、`sin()`の値は-1と1の間を行き来します。`map()`関数によって、変数 `sinval` の値は0から255の範囲に変換され、ウインドウの背景色に使われます。

```
float angle = 0.0;

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

Example 7-13: サイン波の動き

`sin()`の値を動きに変換するはどうなるでしょう。



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

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

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

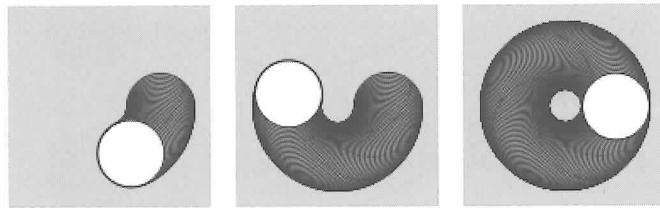
```

float y3 = offset + sin(angle + 0.8) * scalar;
ellipse( 80, y1, 40, 40);
ellipse(120, y2, 40, 40);
ellipse(160, y3, 40, 40);
angle += speed;
}

```

Example 7-14: 円運動

`sin()`と`cos()`を組み合わせると、円運動を作り出すことができます。`cos()`がx座標の、`sin()`がy座標の素となります。2つの値に変数`scalar`を掛けて回転の半径を変更し、変数`offset`を加えて回転の中心を移動します。



```

float angle = 0.0;
float offset = 00;
float scalar = 40;
float speed = 0.05;

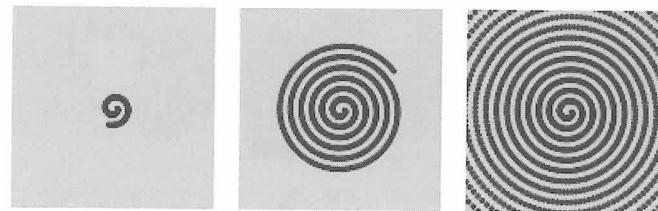
void setup() {
  size(120, 120);
}

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

```

Example 7-15: らせん

`scalar`の値をフレームごとに少しづつ大きくしていくと、円の代わりにらせんが現れます。



```

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

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

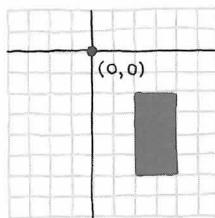
void draw() {
  float x = offset + cos(angle) * scalar;
  float y = offset + sin(angle) * scalar;
  ellipse( x, y, 2, 2);
  angle += speed;
  scalar += speed;
}

```

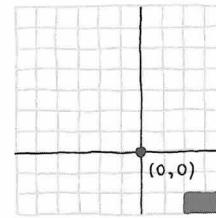
移動、回転、拡大、縮小

座標をずらすことでの動きを作り出すテクニックがあります。たとえば、図形を右に50ピクセル動かすことと、座標(0, 0)の位置を50ピクセル右に動かすことは、同じ視覚効果をもたらします。デフォルトの座標系を変更して、移動、回転、拡大、縮小といった変形を試してみましょう(図7-3)。

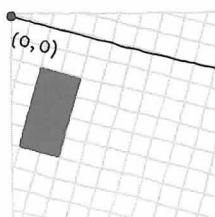
```
translate(40, 20);  
rect(20, 20, 20, 40);
```



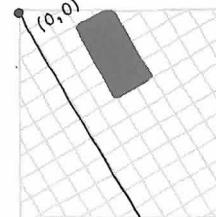
```
translate(60, 70);  
rect(20, 20, 20, 40);
```



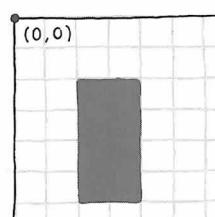
```
rotate(PI/12);  
rect(20, 20, 20, 40);
```



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



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



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

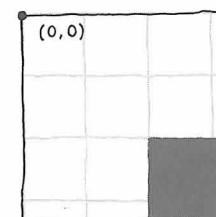
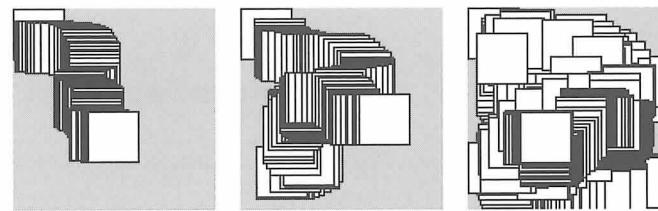


図7-3 座標の移動、回転、拡大、縮小

座標の移動はトリッキーな操作に感じられるかもしれません。一番わかりやすい `translate()` 関数からスタートしましょう。この関数は2つのパラメータをもとに座標系を上下左右にずらします。

Example 7-16: 位置の変更



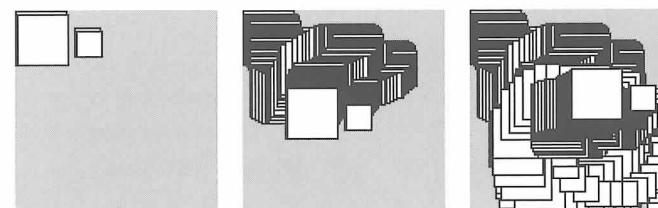
四角形は常に(0, 0)の位置に描かれるのですが、`translate()` の効果で画面上をあちらこちらへ移動します。

```
void setup() {  
    size(120, 120);  
}  
  
void draw() {  
    translate(mouseX, mouseY);  
    rect(0, 0, 30, 30);  
}
```

`translate()` 関数によって、座標(0, 0)はマウスカーソルがある位置へ移動します。次の行の `rect()` は座標(0, 0)を指定していますが、実際にはマウスカーソルがある位置に四角形が描かれます。

Example 7-17: 複数の移動

座標の移動は、続いて行われる描画にも適用されます。2回目の `translate()` によって、2つ目の四角形がどこに現れるかを確認してください。



```

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

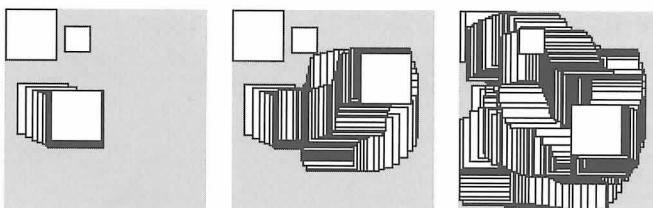
void draw() {
  translate(mouseX, mouseY);
  rect(0, 0, 30, 30);
  translate(35, 10);
  rect(0, 0, 15, 15);
}

```

小さい方の四角形はマウスの位置から右へ35ピクセル、下へ10ピクセル移動したところに表示されます。

Example 7-18: 座標系の復元

座標移動の影響が後続のコードへ及ばないようにしたい場合があります。pushMatrix()関数とpopMatrix()関数を使うことで、変更した座標系を復元することができます。pushMatrix()を実行するとそのときの座標系が記録され、次にpopMatrix()を実行するとその座標系に戻ります。



```

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

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

  popMatrix();
  translate(35, 10);
}

```

```

    rect(0, 0, 15, 15);
}

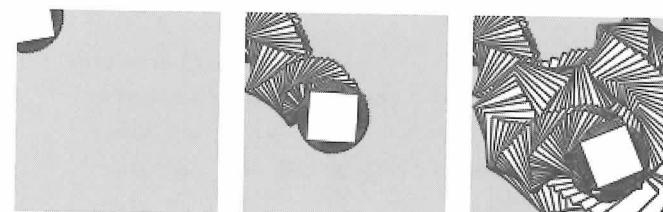
```

popMatrix()がtranslate(mouseX, mouseY)の効果を帳消しにするので、小さい方の四角形は常に同じ位置で表示されます。

[NOTE] pushMatrix()関数とpopMatrix()関数は常にペアで使います。popMatrix()には必ず対応するpushMatrix()が必要です。

Example 7-19: 回転

rotate()関数は座標系を回転させます。パラメータは1つで、回転角（ラジアン）を指定します。回転の軸は常に(0, 0)なので、図形の中心を軸にして回したいときは、まずtranslate()を使って座標をずらし、次にrotate()を呼んで、それから(0, 0)が中心となるように図形を描きます。



```
float angle = 0.0;
```

```

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

```

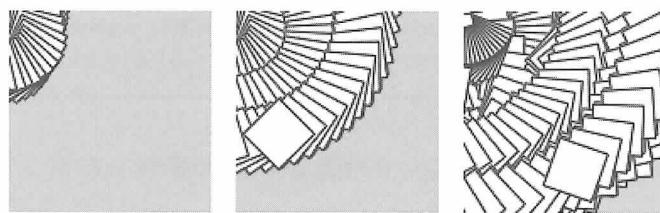
```

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

```

Example 7-20: 移動と回転を組み合わせる

translate() と rotate() を実行する順番が結果に影響します。次の例は、Example 7-19 とほとんど同じコードですが、translate() と rotate() の順番だけが違います。回転の軸はウィンドウの左上隅のままで、図形は translate() の効果により軸から離れたところに描かれます。



```
float angle = 0.0;

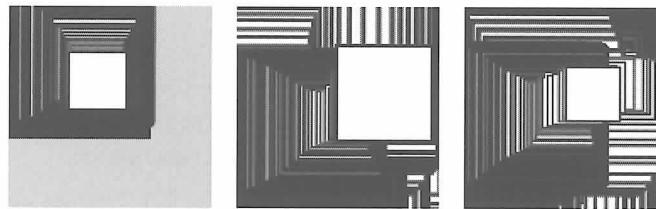
void setup() {
    size(120, 120);
}

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

[NOTE] 中心点を基準に図形を描きたいときは、rectMode()、ellipseMode()、imageMode()、shapeMode() といった関数を使うと簡単になります。

Example 7-21: 伸縮

scale() 関数は座標系を伸縮させます。rotate() と同様に (0, 0) が基準点となるので、図形の中心から伸び縮みさせたい場合は、translate()、scale()、rect() の順に実行し、図形を (0, 0) が中心となるように描きます。



```
float angle = 0.0;

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

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

Example 7-22: 線の太さを一定に保つ

実行中のExample 7-21をよく見ると、`scale()`関数の影響で輪郭線までが太くなってしまうのがわかります。太さを一定に保つため、`strokeWeight()`のパラメータを拡大率(`scalar`)で割ります。

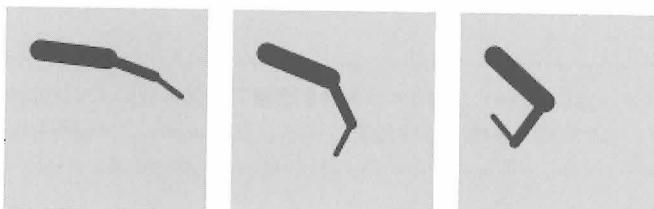
```
float angle = 0.0;

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

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

Example 7-23: 関節のある腕

これが本章の最後にして最長のプログラムです。3組の`translate()`と`rotate()`をつなげて、くねくね曲がる関節を作ってみましょう。それぞれの`translate()`で線の始点へ移動し、`rotate()`で回転を加えていきます。



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

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

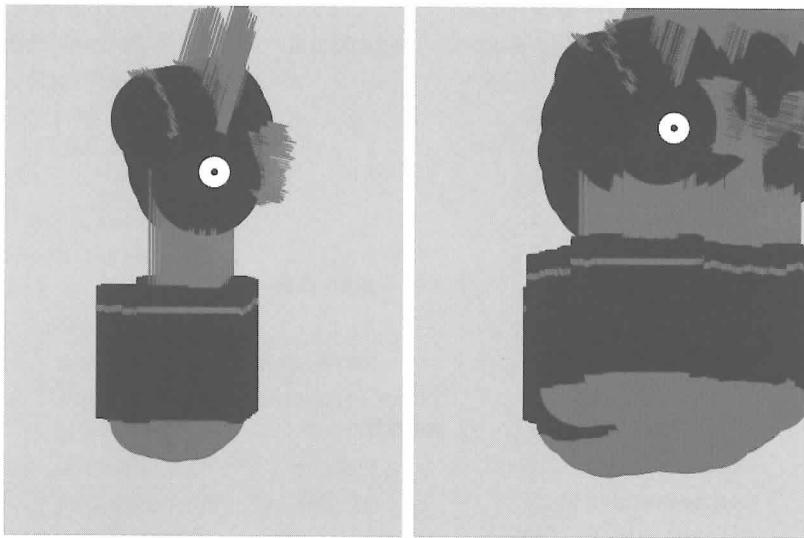
void draw() {
    background(204);
    translate(20, 25); // スタート地点へ移動
    rotate(angle);
    strokeWeight(12);
    line(0, 0, 40, 0);
    translate(40, 0); // 次の関節へ
    rotate(angle * 2.0);
    strokeWeight(6);
    line(0, 0, 30, 0);
    translate(30, 0); // さらに次の関節へ
    rotate(angle * 2.5);
    strokeWeight(3);
    line(0, 0, 20, 0);

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

この例で`pushMatrix()`と`popMatrix()`を使っていないのは、座標移動の結果を引き継ぐためです。それぞれの座標移動は、前回の座標移動で設定された座標を元にしています。

座標系はフレームごとにリセットされるので、`draw()`が始まるときはデフォルトの状態に戻っています。

Robot 5: Motion



この章のロボットには不規則な円運動をさせてみましょう。位置と形の変化がよくわかるように `background()` は省略しました。

毎フレーム、-4 から 4 の間の乱数を `x` 座標に、-1 から 1 の間の乱数を `y` 座標に加えます。その結果、上下よりも左右に激しく動くロボットになりました。`sin()` 関数を使った計算により、首の長さは 50 から 110 ピクセルの範囲で変化します。

```
float x = 180;           // x座標
float y = 400;           // y座標
float bodyHeight = 153; // 胴の高さ
float neckHeight = 56;  // 首の高さ
float radius = 45;      // 頭の半径
float angle = 0.0;       // 動きの角度

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

```
void draw() {
    // 小さな乱数の蓄積により位置を変える
    x += random(-4, 4);
    y += random(-1, 1);

    // 首の長さを変える
    neckHeight = 80 + sin(angle) * 30;
    angle += 0.05;

    // 頭の高さを調整
    float ny = y - bodyHeight - neckHeight - radius;

    // 首
    stroke(102);
    line(x+2, y-bodyHeight, x+2, ny);
    line(x+12, y-bodyHeight, x+12, ny);
    line(x+22, y-bodyHeight, x+22, ny);
    // アンテナ
    line(x+12, ny, x-18, ny-43);
    line(x+12, ny, x+42, ny-99);
    line(x+12, ny, x+78, ny+15);
    // 胴
    noStroke();
    fill(102);
    ellipse(x, y-33, 33, 33);
    fill(0);
    rect(x-45, y-bodyHeight, 90, bodyHeight-33);
    fill(102);
    rect(x-45, y-bodyHeight+17, 90, 6);
    // 頭
    fill(0);
    ellipse(x+12, ny, radius, radius);
    fill(255);
    ellipse(x+24, ny-6, 14, 14);
    fill(0);
    ellipse(x+24, ny-6, 3, 3);
}
```

8

関数

Functions

関数は Processing プログラムの基本的な構成要素です。これまでに紹介したほぼすべてのプログラムで使われています。size()、line()、fill() といった関数は何度も登場しました。この章では新しい関数を作って Processing の能力をさらに拡張する方法を説明します。

関数はLEGOブロックに似ています。それぞれのブロックは固有の役割を持っていて、複雑なモデルを作るときは、役割の異なるブロックをいくつもつなぎあわせます。単独ではなく、組み合わせて使うことが前提になっている点は関数も同じです。また、ブロックは再利用が可能で、1セットの部品からたくさんの形が生まれます。宇宙船の製造に使ったブロックでトラックや高層ビルを組み立てることができます。関数の真価も、部品化されたコードの再利用にあるといつてもいいでしょう。

少し複雑な图形、たとえば木を何本も繰り返し描きたいとします。そういうときは関数を作って効率よく処理しましょう。まず、既存の機能(たとえば`line`)を使い、木を1本描くコードを書きます。それを新たな名前(たとえば`tree`)を持つ関数にまとめてしまえば、それ以降は新しい関数の名前を書くだけで、何本でも簡単に描くことができます。再度同じコードを書く必要はありません。また、関数には複雑なシーケンスを抽象化する働きがあり、`line()`関数で木の形を作るといった実装の詳細ではなく、「木を描く」というより高いレベルの思考に集中することが可能になります。

関数の基礎

コンピュータはプログラムを1行ずつ実行します。関数を実行するとき、コンピュータはまずその関数が定義されている部分へジャンプし、関数内のコードを実行してから、もう一度ジャンプして元の場所へ戻ってきます。

Example 8-1: サイコロを振る

関数のふるまいを`rollDice()`という例題を通じて見ていきましょう。プログラムがスタートすると、すぐに`setup()`内のコードが実行されます。`setup()`内では`rollDice()`が複数回呼び出されていますが、そのたびにプログラムは回り道をして`rollDice()`関数に飛び、そのなかのコードが実行されます。`setup()`内のコードがすべて実行されると、このプログラムは停止します。

```
void setup() {  
    println("Ready to roll!");  
    rollDice(20);  
    rollDice(20);  
    rollDice(6);  
    println("Finished.");  
}  
  
void rollDice(int numSides) {  
    int d = 1 + int(random(numSides));  
    println("Rolling... " + d);  
}
```

`rollDice()`関数の2行のコードは1から`numSides`の間の乱数をコンソールに表示します。この乱数はサイコロを振って出た目を意味していて、プログラムを実行するたびに違う数字が表示されます。

```
Ready to roll!  
Rolling... 20  
Rolling... 11  
Rolling... 1  
Finished.
```

`setup()`内で`rollDice()`が呼び出されるたびに、関数内のコードが上から下へ実行され、また`setup()`に戻って次の行へ処理が移ります。

`random()`関数は0より大きく、パラメータとして指定した数より小さい数を返します。たとえば、`random(6)`と指定した場合は0以上6未満の数です。データ型が`float`なので、結果は5.99999...のようなサイコロらしくない数(浮動小数点数)になり、0が出てしまう可能性もあります。そこで、`int()`を使って整数にコンバートしてから、さらに1を足して1、2、3、4、5、6のうちのどれかを得ています。`random()`の結果が0から始まるのは、本書の他の例でも示されているように、そのほうが計算に使いやすいからです。

Example 8-2: もうひとつの振り方

同様のプログラムを rollDice() 関数を使わずに書くと、こんなふうになるでしょう。

```
void setup() {  
    println("Ready to roll!");  
    int d1 = 1 + int(random(20));  
    println("Rolling... " + d1);  
    int d2 = 1 + int(random(20));  
    println("Rolling... " + d2);  
    int d3 = 1 + int(random(6));  
    println("Rolling... " + d3);  
    println("Finished.");  
}
```

rollDice() 関数を使った Example 8-1 のコードのほうが読みやすく、メンテナンスも容易です。一方で、setup() のなかに random() 関数が並んでいるこの例のようなプログラムは、わかりやすいとはいえません。

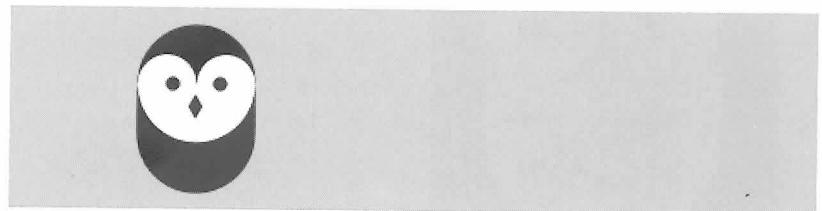
関数を使うと、その名前が目的を説明してくれるので、プログラムが明確になります。サイコロの面数を表す 6 という数字も理解を助けてくれます。rollDice(6) というコードを見れば、6 面体のサイコロを振る、という機能を示していることは明らかです。Example 8-1 のほうがメンテナンスしやすい理由をもうひとつあげましょう。2 つめのプログラムには "Rolling..." というフレーズが 3 回も出てきます。もし、この表記を変えたくなったら 3 か所を修正する必要がありますが、関数化されていれば 1 か所直すだけで済みます。関数を使ってプログラムを短くすることは、バグが発生する可能性を小さくすることにもつながります。

関数を作る

ここからは、関数の作り方を、フクロウを描きながら、ステップごとに説明します。

Example 8-3: フクロウを描く

まずははじめに、関数を作らずに描いてみましょう。



```
void setup() {  
    size(480, 120);  
}  
  
void draw() {  
    background(204);  
    translate(110, 110);  
    stroke(0);  
    strokeWeight(70);  
    line(0, -35, 0, -65); // 胴  
    noStroke();  
    fill(255);  
    ellipse(-17.5, -65, 35, 35); // 右目のまわり  
    ellipse(17.5, -65, 35, 35); // 左目のまわり  
    arc(0, -65, 70, 70, 0, PI); // 頸  
    fill(0);  
    ellipse(-14, -65, 8, 8); // 右目  
    ellipse(14, -65, 8, 8); // 左目  
    quad(0, -58, 4, -51, 0, -44, -4, -51); // くちばし  
}
```

描画を始める前に、translate() で座標(0, 0)を右に 110 ピクセル、下に 110 ピクセル移動します。フクロウの絵は(0, 0)を基準に描かれ、座標の指定には負の値と正の値の両方が使われます。

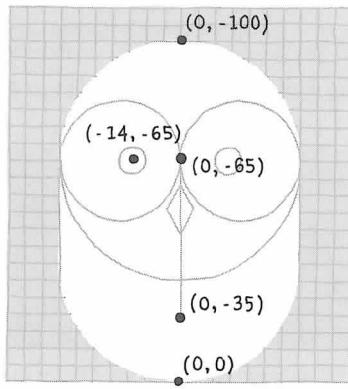
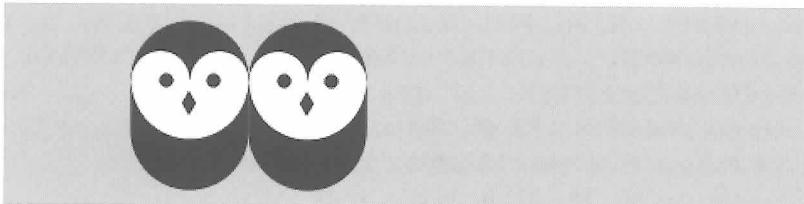


図8-1 フクロウの座標

Example 8-4:2羽のフクロウ

Example 8-3の描き方は、フクロウが1羽だけなら合理的です。しかし、もう1羽描くと、コードはほとんど倍の長さになってしまいます。



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

void draw() {
    background(204);
    // 左のフクロウ
    translate(110, 110);
    stroke(0);
    strokeWeight(70);
    line(0, -35, 0, -65); // 脳
    noStroke();
```

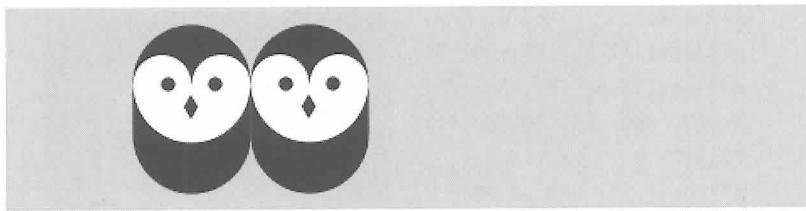
```
fill(255);
ellipse(-17.5, -65, 35, 35); // 右目のまわり
ellipse(17.5, -65, 35, 35); // 左目のまわり
arc(0, -65, 70, 70, 0, PI); // 頸
fill(0);
ellipse(-14, -65, 8, 8); // 右目
ellipse(14, -65, 8, 8); // 左目
quad(0, -58, 4, -51, 0, -44, -4, -51); // くちばし

// 右のフクロウ
translate(70, 0);
stroke(0);
strokeWeight(70);
line(0, -35, 0, -65); // 脳
noStroke();
fill(255);
ellipse(-17.5, -65, 35, 35); // 右目のまわり
ellipse(17.5, -65, 35, 35); // 左目のまわり
arc(0, -65, 70, 70, 0, PI); // 頸
fill(0);
ellipse(-14, -65, 8, 8); // 右目
ellipse(14, -65, 8, 8); // 左目
quad(0, -58, 4, -51, 0, -44, -4, -51); // くちばし
}
```

このプログラムの21行目から34行目は、最初のフクロウのコードをカット&ペーストし、右に70ピクセル移動する`translate()`を追加しただけです。この退屈かつ非効率な方法で3羽目を描くことを考えると、頭痛がしてきます。コードの複製はやめましょう。この状況は、関数が解決してくれます。

Example 8-5:フクロウ関数

ひとつのコードで2羽のフクロウを描くために、関数を導入します。フクロウを描くコードを関数に入れてしまえば、そのコードはもうプログラム中に1度しか現れません。



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

void draw() {
    background(204);
    owl(110, 110);
    owl(180, 110);
}

void owl(int x, int y) {
    pushMatrix();
    translate(x, y);
    stroke(0);
    strokeWeight(70);
    line(0, -35, 0, -65); // 脳
    noStroke();
    fill(255);
    ellipse(-17.5, -65, 35, 35); // 右目のまわり
    ellipse(17.5, -65, 35, 35); // 左目のまわり
    arc(0, -65, 70, 70, 0, PI); // 頬
    fill(0);
    ellipse(-14, -65, 8, 8); // 右目
    ellipse(14, -65, 8, 8); // 左目
    quad(0, -58, 4, -51, 0, -44, -4, -51); // くちばし
    popMatrix();
}
```

Example 8-4とこの例の出力画像を見てください。まったく同じです。しかし、コードはこの例のほうが短くなっています。owl()という適切な名前を与えられた関数に、フクロウを描くコードがまとめられているからです。そのコードはdraw()から2回呼び出されるので、2回実行されます。呼び出しの際、2つのパラメータ(x座標とy座標)を関数に渡しています。

これにより、2羽のフクロウが異なる位置に表示されます。

パラメータは関数に柔軟性を与える大事な要素です。rollDice()関数にもパラメータが1つありました。サイコロの面数を表すnumSidesという名前のパラメータで、6面体なら6、20面体なら20と指定しました。この値を変えるだけで、どんな面数のサイコロも実現可能です。こうした柔軟性はProcessingの他の関数にも見られます。たとえば、line()関数はパラメータしだいで画面上のあらゆる2点間に線を引くことができます。もしこのパラメータがなかったら、たった1種類の線しか引けませんね。

このプログラムでowl()関数は2回実行されますが、1回目のパラメータxの値は110、2回目は180となっている点に注意してください。関数に渡すパラメタによって、その関数のなかの変数の値が置き換えられます。ひとつのコードが毎回異なる初期値で実行されるわけです。

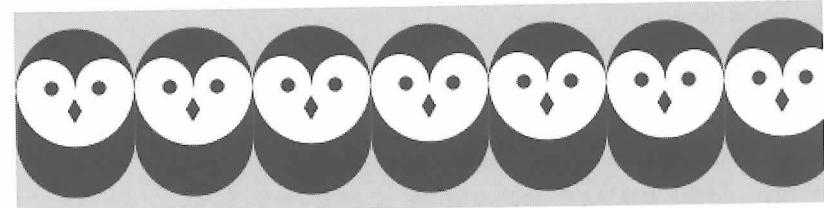
各パラメータには、intやfloatといったデータ型があります。関数で定義されているパラメータの型と、関数に渡す値の型が一致していることを確認してください。もし、Example 8-5で次のようなコードを実行すると、エラーになります。

```
owl(110.5, 120.2);
```

その理由は、xパラメータとyパラメータはint型なのに対し、110.5と120.2はfloat型なのでマッチしないからです。

Example 8-6: 人口増加

フクロウを好きな位置に描く関数ができあがりました。この関数をforループと組み合わせて、効率良くてたくさんのフクロウを描いてみましょう。owl()関数の第1パラメータを変更するだけで、うまく機能します。



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

void draw() {
    background(204);
```

```

for (int x = 35; x < width + 70; x += 70) {
    owl(x, 110);
}

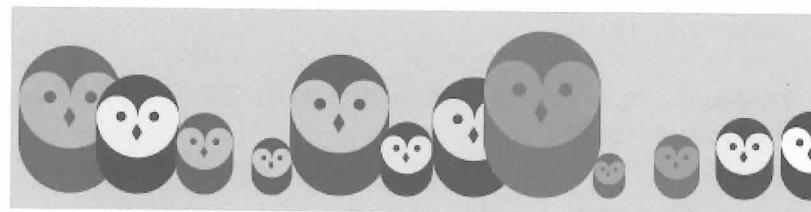
// ここに Example 8-5 の owl() 関数を挿入してください

```

もっといろいろな形のフクロウを描くために、owl()関数のパラメータを増やすことができます。たとえば、大きさ、傾き、色、目のサイズを変更するパラメータが考えられます。

Example 8-7: いろんな大きさのフクロウたち

この例では、2つのパラメータを追加します。フクロウたちの色（グレースケール）と大きさを変えてみましょう。



```

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

void draw() {
    background(204);
    randomSeed(0);
    for (int i = 35; i < width + 40; i += 40) {
        int gray = int(random(0, 102));
        float scalar = random(0.25, 1.0);
        owl(i, 110, gray, scalar);
    }
}

void owl(int x, int y, int g, float s) {
    pushMatrix();
    translate(x, y);

```

```

    scale(s); // 大きさをセット
    stroke(g); // グレー値をセット
    strokeWeight(70);
    line(0, -35, 0, -65); // 体
    noStroke();
    fill(255-g);
    ellipse(-17.5, -65, 35, 35); // 右目のまわり
    ellipse(17.5, -65, 35, 35); // 左目のまわり
    arc(0, -65, 70, 70, 0, PI); // 頸
    fill(g);
    ellipse(-14, -65, 8, 8); // 右目
    ellipse(14, -65, 8, 8); // 左目
    quad(0, -58, 4, -51, 0, -44, -4, -51); // くちばし
    popMatrix();
}

```

値を返す

関数は計算結果を呼び出し元のプログラムに返すことができます。これまでに使った関数の中では、random() や sin() などが該当します。関数からの値は変数に代入することが多いでしょう。

次のコードは、random() が返す1から10の間の値を変数 r に代入します。

```
float r = random(1, 10);
```

値を返す関数を、別の関数のパラメータとして使用することもよくあります。

```
point(random(width), random(height));
```

この例では、random() の値を変数に入れる代わりに、point() 関数へパラメータとして渡し、描画に利用しています。

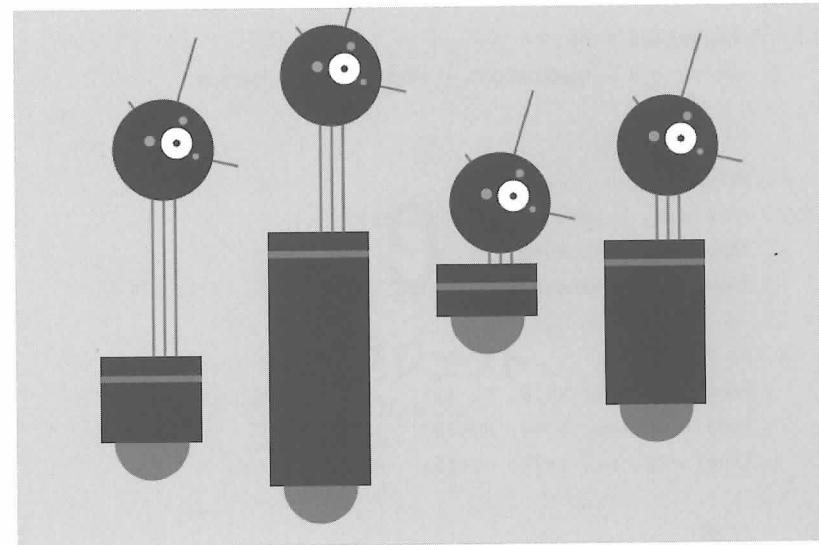
Example 8-8: 値を返す

これまでに定義した関数はどれも、関数名の前に `void` というキーワードがついていました。値を返す関数を作るときは、`void` の代わりにその関数が返す値のデータ型を書きます。関数の中には、返すデータを指定する `return` 文があります。次の例の関数 `calculateMars()` は火星表面における体重を計算し、その結果を返します。

```
void setup() {  
    float yourWeight = 132;  
    float marsWeight = calculateMars(yourWeight);  
    println(marsWeight);  
}  
  
float calculateMars(float w) {  
    float newWeight = w * 0.38;  
    return newWeight;  
}
```

この関数は浮動小数点数を返すので、関数名の前には `float` があります。ブロックの最終行は変数 `newWeight` を返すという意味です。`setup()` の2行目で、その値は変数 `marsWeight` に代入されます。コードを修正して自分の火星体重を調べてみましょう。

Robot 6: Functions



見た目は4章の Robot 2と同じですが、このロボットは関数を使って1つのコードを4回実行することで描かれています。`drawRobot()` 関数は `draw()` 内で4回呼ばれ、毎回異なるパラメータが渡されて、位置や形に変化が生じます。

Robot 2 のプログラムでは冒頭で宣言されていた `radius` や `bodyHeight` といった変数は、この例ではすべて `drawRobot()` ブロックの中だけに存在します。いつも同じ値の `radius` については、パラメータとして渡されることもなく、ブロック内で定義されています。

```
void setup() {  
    size(720, 480);  
    strokeWeight(2);  
    ellipseMode(RADIUS);  
}  
  
void draw() {  
    background(204);  
    drawRobot(120, 420, 110, 140);  
    drawRobot(270, 460, 260, 95);  
    drawRobot(420, 310, 80, 10);  
    drawRobot(570, 390, 180, 40);  
}
```

```

void drawRobot(int x, int y, int bodyHeight, int neckHeight) {

    int radius = 45;
    int ny = y - bodyHeight - neckHeight - radius;

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

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

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

    // 頭
    fill(0);
    ellipse(x+12, ny, radius, radius);
    fill(255);
    ellipse(x+24, ny-6, 14, 14);
    fill(0);
    ellipse(x+24, ny-6, 3, 3);
    fill(153);
    ellipse(x, ny-8, 5, 5);
    ellipse(x+30, ny-26, 4, 4);
    ellipse(x+41, ny+6, 3, 3);
}

}

```



9

オブジェクト

Objects

オブジェクト指向プログラミングは新たな思考法です。言葉の印象から難解に感じるかもしれませんのが心配はいりません。実はもうすでにあなたはオブジェクトを使ったプログラミングを体験済みで、6章以降で登場した PImage、PFont、String、PShape の正体はオブジェクトなのです。boolean、int、floatといった1つの値しか記憶できないプリミティブなデータ型と違い、オブジェクトはたくさんの値を持つことができます。複数の変数とそれに関連する関数を集約することで、より理解しやすいパッケージとして扱えるようになります。