

# Processing 言語による情報メディア入門

## 文字列と画像の表示と座標変換

神奈川工科大学情報メディア学科 佐藤尚

今までのプログラムでは、図形の表示だけを扱ってきました。色々なプログラムを作っていく際には、図形の表示だけではなく、文字や画像の表示を行いたいことがあります。今回は、文字列や画像の表示を取り扱います。

### 文字列の表示

Processing 言語で、様々な種類のフォントを表示することが出来ます。そのためには、いくつかの手順が必要となります。大雑把に言うと、文字列を表示するためには、1) 使用したいフォントを指定してから、2) 文字列の表示位置と表示文字列を指定するという手順になります。文字列表示の手順を詳しく述べると、以下のようになります。

#### 手順 1: フォントの指定

文字列を表示するためには、まず Processing 内部にフォントの情報を取り込む必要があります。そのために、コンピュータで使えるフォント情報を Processing で扱うことの出来る vlw フォーマットに変換する必要があります。この変換処理は、Tools メニューの「Create Font...」を選ぶと表示されるダイアログボックスで行うことが出来ます。このダイアログボックスで、変換したいフォントを指定し、その大きさ (size) を指定します。OK ボタンをクリックすると、Filename 欄に表示されている名前がファイル名となっている vlw フォーマットのファイルが作成されます。作成されたファイルは、Sketch メニューの「Show Sketch Folder」を選ぶと表示されるフォルダ内にある data フォルダに保存されています。

次に、生成したフォント情報を PFont 型変数の変数に読み込みます。この読み込みのためには、loadFont 関数を使用します。さらに、どのフォント情報を使用して文字列の表示を行うかを定めるために、textFont 関数を使用します。textFont 関数では、表示するフォントの大きさを指定することも出来ます。

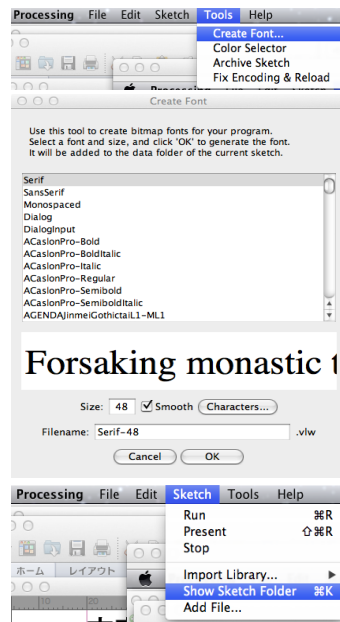


図 6-1 vlw ファイルの作成

vlw フォーマットでは、文字の形の情報をイメージとして保持しています。

作成するフォントのサイズを大きくすると、変換後に作られるファイルのサイズが大きくなります。同様に、漢字などを含むフォントを変換すると変換後に作られるファイルのサイズがかなり大きくなりますので、注意して下さい。

後で、作成したフォント名が必要となるので、Filename 欄のファイル名をコピーしておくと便利です。

Processing のスケッチ (プログラム) で使用されるデータファイルは、この data フォルダ内に保存することが一般的です。

複数の vlw 形式のファイルを読み込むことで、複数のフォントで表示を行うことが出来ます。

表 6-1 文字列表示関連のデータ型と関数その 1

関数名など	説明
PFont	フォント情報を格納するデータ型
String	文字列を格納するデータ型
loadFont(file)	引数 file で指定された vlw ファイルを読み込む関数
textFont(f)	PFont 型の引数 f で指定したフォントを表示に利用する
textFont(f,size)	PFont 型の引数 f で指定したフォントを大きさ size で表示に利用する
text(str,x,y)	引数 str で指定された文字列を位置 (x,y) に表示する関数
text(str,x,y,w,h)	引数 str で指定された文字列を位置 (x,y)、幅 w、高さ h の長方形の内部に表示する関数
textSize(size)	表示に利用するフォントの大きさを size に設定する関数
createFont(fname,size)	大きさ size で引数 fname で指定したフォント情報を作成する。

正方形の指定方法は、実行時の rectMode により変化します。

createFont 関数を使用すると、「Tools > Create Font」で vlw ファイルを作成しなくても大丈夫です。

**手順 2：フォントの表示**

文字列を表示する際には、text 関数を使用します。text 関数では、表示する文字列と表示位置を指定します。文字列の表示色は、fill 関数で指定した色が使用されます。stroke の指定は無視されます。3つの引数をとる text 関数での表示位置の指定は、基本的に図の赤丸の場所を指定します。



赤丸を通る X 軸に平行な線をベースラインと呼んでいます。

図 6-2 フォント表示の際の位置指定

**文字列を表示する単純なサンプル 6-1**

```
PFont font; // フォント情報を保存する変数
String msg = "Riho";
size(400,200);
font = loadFont("Geneva-48.vlw"); // vlw ファイルの読み込み
textFont(font); // 表示するフォントの指定
background(255);
fill(0);
text("Kanagawa Institute Of Technology",10,50); // 文字列を表示
textSize(16); // 表示するフォントの大きさの変更
text("Kanagawa Institute Of Technology",10,100);
textSize(32); // 表示するフォントの大きさの変更
text(msg,10,150); // String 変数の保存されている文字列を表示
```

このサンプルでは、Geneva フォントを使用して、大きさ 48 の vlw ファイルを作成して使用しています。

作成した vlw ファイルのフォントサイズより大きなサイズでは、フォントを表示しない方が望ましいと思います。

次に複数のフォントを表示するサンプルを示します。

### 複数のフォントで表示する単純なサンプル 6-2

```
PFont f1;// フォント情報を保存する変数
PFont f2;
String msg = "The quick brown fox jumps over the lazy dog";
size(400,300);
f1 = loadFont("Serif-48.vlw");// vlw ファイルの読み込み
f2 = loadFont("SansSerif-48.vlw");// vlw ファイルの読み込み
background(255);
fill(50);
textFont(f1,20);// 表示するフォントと大きさの指定
text("Riho",50,100); // 文字列の表示
textFont(f2,18);// 表示するフォントと大きさの指定
text(msg,50,200); // String 変数の保存されている文字列を表示
```

文字列は、長方形の領域を指定して、その内部に表示することが出来ます。この目的のためには、5つの引数をとる text 関数を使用します。この関数は最初の引数で表示する文字列を指定し、残りの引数を利用して、表示を行う長方形を指定します。この長方形領域の指定方法は rect 関数の場合と同じです。従って、現在の rectMode で長方形が指定されます。

### 長方形領域での文字列表示の単純なサンプル 6-3

```
PFont font;// フォント情報を保存する変数
String msg = "The quick brown fox jumps over the lazy dog";
size(400,300);
font = loadFont("Serif-48.vlw");// vlw ファイルの読み込み
background(255);
fill(0);
textFont(font);// 表示するフォントの指定
text(msg,50,50,350,200);//
```

サンプル 6-4 では、rectMode を CENTER に変更したものを示します。表示の違いを確認して下さい。

### CENTER 指定での長方形領域での文字列表示のサンプル 6-4

```
PFont font;// フォント情報を保存する変数
String msg = "The quick brown fox jumps over the lazy dog";
size(400,300);
font = loadFont("Serif-48.vlw");// vlw ファイルの読み込み
rectMode(CENTER); // rectMode を CENTER に変更
background(255);
fill(0);
textFont(font);// 表示するフォントの指定
text(msg,50,50,350,200);//
```

サンプル 6-5 で、長方形領域の指定方法をマウスボタンが押され

英語の文「The quick brown fox jumps over the lazy dog」の特徴がわかりますか？割と有名なフレーズなのですが。

このサンプルでは、サイズ 48 の Serif フォントと SanSerifu フォントを利用して、vlw ファイルを作成しています。

ているかどうかで変更するものを示します。

### 長方形領域指定方法切り替えでの文字列表示のサンプル 6-5

```
PFont font;// フォント情報を保存する変数
String msg = "The quick brown fox jumps over the lazy dog";
void setup(){
  size(400,300);
  font = loadFont("Serif-48.vlw");// vlw ファイルの読み込み
  rectMode(CENTER);
}
void draw(){
  background(255);
  if(mousePressed ){
    rectMode(CENTER); // マウスボタンが押されていたら、CENTER
  }else{
    rectMode(CORNER);// マウスボタンが押されていないならば、CORNER
  }
  fill(0);
  textFont(font);// 表示するフォントの指定
  // 後ろの4つの引数で表示領域の長方形の指定を行っている
  text(msg,mouseX,mouseY,350,200);
  noFill();
  stroke(255,10,10);
  rect(mouseX,mouseY,350,200); // 表示領域の表示
}
```

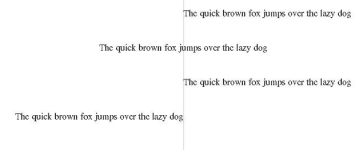
rectMode が CORNER が デフォルトの指定です。

サンプル 6-3 ~ 6-5 のように文字列を表示する長方形領域を指定できるとすると、文字揃えなども行いたくなります。これを実行するのが、textAlign 関数です。サンプル 6-6 で、この textAlign 関数を利用して、文字揃えを変更したものを示します。

### 文字揃えの変更を行う 6-6

```
PFont f;
String msg = "The quick brown fox jumps over the lazy dog";
size(600,300);
f = loadFont("Serif-48.vlw");
background(255);
stroke(200);
line(width/2,0,width/2,height);
fill(50);
textFont(f,16);
text(msg,width/2,60);
textAlign(CENTER);
text(msg,width/2,120);
textAlign(LEFT);
text(msg,width/2,180);
textAlign(RIGHT);
text(msg,width/2,240);
```

### サンプル 6-6 の実行例



The screenshot shows a window with a white background and a vertical line at the center. The text "The quick brown fox jumps over the lazy dog" is displayed in a serif font. The text is centered horizontally in the first instance, left-aligned in the second, and right-aligned in the third. The text is positioned at different vertical offsets from the top of the window.

表 6-2 文字列表示関連のデータ型と関数その 2

関数名など	説明
textAlign(CENTER)	指定した長方形領域に、文字列を中央揃えで表示するようにする。
textAlign(LEFT)	指定した長方形領域に、文字列を左揃えで表示するようにする。
textAlign(RIGHT)	指定した長方形領域に、文字列を右揃えで表示するようにする。
textWidth(str)	現在の表示文字設定で、文字列 str を表示した時の幅を求める関数
textDescent()	現在の表示文字設定で、文字列を表示した時のベースラインからどれだけ下に表示されるかを求める関数。
textAscent()	現在の表示文字設定で、文字列を表示した時のベースラインからどれだけ上に表示されるかを求める関数。

図 6-2 に示すように、文字 g はベースラインの下の方にも文字の一部が出ています。また、文字によって文字の高さが異なっていますし、文字によってその幅も異なっています。これらの情報がわかれば、文字列が描かれる仮想的な長方形を決めることができます。このようなことが出来れば、ウインドウの端で跳ね返るようなプログラムを作成することができます。

サンプル 6-7 は、図 6-1 を描く際に利用したプログラムです。

### テキストの表示位置を求めるサンプル 6-7

```

PFont font;
String msg = "Anegasaki";
void setup(){
  size(1024,512);
  font = loadFont("Serif-128.vlw");
  textFont(font);
}
void draw(){
  background(255);
  fill(0);
  text(msg,mouseX,mouseY);
  stroke(255,10,10);
  fill(255,10,10);
  ellipse(mouseX,mouseY,10,10);
  line(-128+mouseX,mouseY,width,mouseY);
  line(-128+mouseX,mouseY+textDescent(),
    width,mouseY+textDescent());
  line(-128+mouseX,mouseY-textAscent(),width,mouseY-textAscent());
  line(mouseX,mouseY+128,mouseX,0);
  line(mouseX+textWidth(msg),mouseY+textDescent(),
    mouseX+textWidth(msg),0);
}

```

次のサンプル 6-8 は textWidth 関数を使用した例です。左端で表示文字列が消えると、右側から現れてきます。長方形や円の場合と同じように表示位置を少しずつ変化させると、文字列の移動が実現出来ます。このサンプルでは、文字列の最後がウインドウの左端に到達したら、再び右端から文字列を表示するようにしています。

### 文字列の移動サンプル 6-8

```
PFont f;
String msg = "The quick brown fox jumps over the lazy dog";
float x; // 文字列の表示開始位置の x 座標

void setup(){
  size(400,200);
  colorMode(RGB);
  f = loadFont("Serif-48.vlw");
  x = width;
}
void draw(){
  background(255);
  fill(50);
  textFont(f,16);
  float widthOfMsg = textWidth(msg);
  text(msg,x,height/2);
  x -= 2; // 2 ずつ左に移動
  if(x < -widthOfMsg){ // 文字列の最後がウインドウから消えたら
    x = width;
  }
}
```

この後に学習する知識などを使用すると、Star Wars のオープニングのようなプログラムを簡単に作る事ができます。

### おまけのサンプル 6-9

```
PFont f;
String msg = "A New Hope¥n¥nA long long time ago¥nIn a galaxy far far away....";
float y=0;
void setup(){
  size(400,400,P3D); // 3D 表示を行う
  f = loadFont("Serif-48.vlw");
  textFont(f);
  textAlign(CENTER);
}
void draw(){
  background(0);
  fill(5,200,255);
  translate(width/2,height/2);
  rotateX(PI/4);
  text(msg,0,y);
  y--;
}
```

文字列の表示位置の X 座標の値は x です。文字列を表示する際に必要となる幅は widthOfMsg なので、文字列を表示した際の右端の X 座標の値は x+widthOfMsg となります。この場所がウインドウの左端から出てしまうのは、x+widthOfMsg < 0 の時です。widthOfMsg を右辺に移項すると、if 命令の条件部分になります。

文字列中の ¥n は表示されることがありません。この ¥n があると、そこで改行が行われます。C 言語系のプログラミング言語では、文字列中に ¥n があると改行を意味しています。このような ¥ と組みになって特別な意味を表すものをエスケープシーケンスと呼んでいます。本によっては、¥ の代わりに \ を使用している場合があります。これは文字コードの問題が関連しています。IT 基礎で関連する話題が出てくると思います。



## 画像ファイルとしての保存

今までのプログラムでは、作成した画像はプログラムの実行を終了すると消えていました。プログラムを実行せずに作成した画像を見るためには、作成した画像を保存することが必要となります。Processingでは、ウインドウの内容を画像ファイルとして保存する `save` 関数と `saveFrame` 関数が用意されています。save 関数は、ウインドウの内容を1つの画像ファイルとして保存します。一方、saveFrame 関数は連番の番号付きファイル名で画像を保存します。保存される画像ファイルは、Tools メニューの「Show Sketch Folder」を選んだ時に出てくるフォルダ内にある data フォルダの中に保存されます。

表 6-3 画像ファイルとしての保存関連の関数

関数	説明
<code>save(filename)</code>	filename で指定したファイル名でウインドウの内容を画像ファイルとして保存します。ファイル名には、保存する画像ファイルの形式を指定する拡張子が必要です。指定できる拡張子は <code>tif,tga,jpg,png</code> です。
<code>saveFrame()</code>	ウインドウの内容を連番の画像ファイルとして保存します。保存されるファイル名は <code>screen-連番.tif</code> です。この関数を呼び出す度に、連番部分の数字が増えていきます。
<code>saveFrame(filename)</code>	ウインドウの内容を連番の画像ファイルとして保存します。引数の filename は <code>filename-####.拡張子</code> の形となります。#### の部分が連番の数字となります。# が4つあれば、4桁の数字で連番の部分の数字が決まります。指定できる拡張子は <code>tif,tga,jpg,png</code> です。

つまり、次の4つのファイル形式で画像をファイルに保存出来ます。

tif : TIFF 形式  
tga : TARGA 形式  
jpg : JPEG 形式  
png : PNG 形式

サンプル 6-10 はサンプル 6-1 の最後に `save` 関数の呼び出しを追加して、ウインドウの内容を画像ファイルとして保存するものです。

### save 関数を利用した サンプル 6-10

```
PFont font; // フォント情報を保存する変数
String msg = "Riho";
size(400,200);
font = loadFont("Geneva-48.vlw"); // vlw ファイルの読み込み
textFont(font); // 表示するフォントの指定
background(255);
fill(0);
text("Kanagawa Institute Of Technology",10,50); // 文字列を表示
textSize(16); // 表示するフォントの大きさの変更
text("Kanagawa Institute Of Technology",10,100);
textSize(32); // 表示するフォントの大きさの変更
text(msg,10,150); // String 変数の保存されている文字列を表示
save("test.jpg"); // ウインドウの内容を test.jpg ファイルに保存
```

draw 関数内で `save` 関数を呼び出し画像ファイルとして保存する場合には、プログラムの実行を終了するタイミングによっては、正しく保存されない場合があります。

画像ファイルがどこに保存されるか、ちゃんと確認しておいて下さい。

## 画像ファイルの読み込み

プログラムを作成していると、外部で作成した画像をファイルの表示や利用などを行いたいことがあります。Processing 言語では、画像データを保存するために PImage 型が用意されています。PImage 型の変数には画像データを記憶させておくことができます。

Processing でファイルに保存されている画像を表示するためには、

1. 画像ファイルに保存されている画像データをコンピュータ内に読み込む、
  2. 読み込んだ画像データを表示する、
- という手順をとります。

### 手順 1：画像ファイルの読み込み

このプログラムで読み込む画像ファイルはどこに置かれているのでしょうか？ Processing では、プログラム中で読み込む画像ファイルなどの保存場所が決まっています。それは、“Show Sketch Folder” で表示されるフォルダ内にある data という名称のフォルダです。もし、その data フォルダが無い場合には、data フォルダを作成して、そこに利用する画像ファイルなどをコピーして下さい。これが面倒な場合には、Processing のプログラムを書いている部分にドラッグ&ドロップすると、自動的に data フォルダにコピーされます。

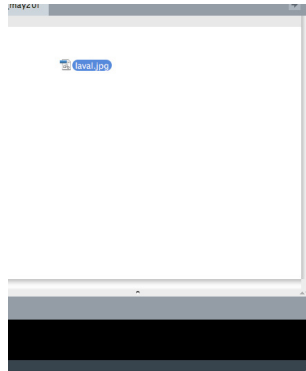


図 6-3 データファイルのドラッグ&ドロップ

画像ファイルを読み込むためには、

loadImage 関数を使用します。読み込んだ情報を PImage 型の変数に代入します。

### 手順 2：画像ファイルの表示

読み込んで PImage 型変数に保存されている画像を表示するためには、image 関数を使用します。

表 6-4 画像データ表示関連の関数など

関数	説明
PImage	画像情報を保存するためのデータ型
loadImage(filename)	filename で指定した画像ファイルを読み込む。TIFF 形式、TARGA 形式、JPEG 形式、PNG 形式の画像ファイルを読み込むことができます。
loadImage(name,ext)	引数 ext では読み込む画像データの種類(png.jpg.gif など)を指定する。
image(img,x,y)	PImage 型引数 img で指定した画像の内容を、引数 x,y で指定された場所に表示する。位置の指定方法は、imageMode 関数で指定します。
image(img,x,y,w,h)	PImage 型引数 img で指定した画像の内容を、引数 x,y で指定された場所に、横方向の大きさを w、縦方向の大きさを h に変更して表示する。

文字列の表示と同じような手順となっています。

実は、loadImage 関数はかなり強力な機能を持っています。

画像は長方形になっているので、rect 関数で長方形を描画するのと同じ方法で、画像の描画位置を指定します。

image 関数のように、引数の数やデータ型によって処理の内容によって異なる関数定義を行うことを多重定義（オーバーロード）と呼びます。以



関数	説明
imageMode(mode)	引数 mode で指定された方法で、表示する画像位置を指定できるようになります。引数 mode には、値 CORNER,CENTER,CORNERS を指定することが出来ます。それぞれの値の意味は、rectMode の場合と同じです。

前から使用していた fill 関数や stroke 関数も多重定義されている関数です。

サンプル 6-11 は loadImage 関数を imae 関数を利用した単純なプログラムです。

### image 関数を利用した サンプル 6-11

```
PImage src; // 画像データを保存するための変数
void setup(){
  size(400,400);
  src = loadImage("laval.jpg");// ファイル名は適当なものに変えること
}
void draw(){
  image(src,random(width),random(height)); // デタラメな位置に表示
}
```

Processing では、異なる変数に画像データを記録させておけば、複数の画像を扱うことが出来ます。サンプル 6-12 は複数の画像を取り扱うサンプルプログラムです。このプログラムは、マウスボタンを押した状態と離れた状態で表示する画像を切り替えています。また、src2.width と src2.height のようにすると、src2 に保存されている画像の横方向の画素数と高さ方向の画素数を取り出すことが出来ます。

### 複数の画像を利用した サンプル 6-12

```
PImage src1,src2;
void setup(){
  size(600,600);
  src1 = loadImage("2cv.jpg");// ファイル名は適当なものに変えること
  src2 = loadImage("laval.jpg");
}
void draw(){
  background(255);
  if(mousePressed==true){
    image(src1,mouseX,mouseY);
  }else{
    // 画像を半分の大きさにして表示
    image(src2,mouseX,mouseY,src2.width/2,src2.height/2);
  }
}
```

表 6-5 画像データの取得

フィールド名	意味
PImage 型変数 .width	記録されている画像の横方向の画素数を記憶している。
PImage 型変数 .height	記録されている画像の縦方向の画素数を記憶している。

## QR コードの作成（おまけ）

画像ファイルを表示するために使用した loadImage 関数は単にパソコン内の画像を読み込むだけではなく、もう少し高度なことも出来ます。その一例として、loadImage 関数を利用して、QR コードを作ることに挑戦します。Google では Chart API と呼ばれる web を利用してグラフを描く機能を提供しています。この機能の中に QR コードを描くものがあるので、これを利用します。サンプル 6-13 は QR コードを表示するものです。

### QR コードの表示 サンプル 6-13

```
PImage qrcode_img; // QR コード画像を保存する変数
String uri; // 文字列を保存する変数
String qrcode_google_api = "http://chart.apis.google.com/chart?";
int qrcode_size = 300; // 表示する QR コードの大きさを決める値
String data = "www.kait.jp"; // QR コードの中に埋め込みたい情報

size(qrcode_size, qrcode_size);
// Google Chart API を呼び出すための URL を作る。
uri = qrcode_google_api + "chs=" + qrcode_size + "x" + qrcode_size
+ "&cht=qr&chl=" + data;
// Google Chart API の機能を利用して、QR コード作成。
qrcode_img = loadImage(uri,"PNG");// PNG 型で画像ファイル
image(qrcode_img,0,0); // 作成した QR コードを表示
save("myqrcode.png"); // 表示した QR コードを保存、ファイル名は変更可能
```

鋭い人は気がついたかもしれませんが、Processing の loadImage 関数は data フォルダ内の画像だけでなく、ファイル名を URI で指定すると、web サイトなどに置かれている画像を読み出すことも出来ます。この機能を利用したものがサンプル 6-14 です。

### URI 指定での画像の表示 サンプル 6-14

```
PImage img;

void setup(){
  size(300,300);
  // setup 内で画像を読み込まないと、ちょっと面倒なことが起きるかも
  img = loadImage("http://www.kait.jp/images/top2011/index.jpg");
}

void draw(){
  background(255);
  image(img,0,0);
}
```

この方法は、クラウドと呼ばれている情報処理を利用して見ると見ることも出来ます。

ここで使用している + も多重定義されています。int 型や float 型の際には加算として機能しますが、String 型の際には文字列の連結となっています。

Uniform Resource Identifier

ネット越しに画像データの取得を行っているので、loadImage 関数の実行が終わっても画像データの読み込みが完全に終了していない場合があります。その場合には、上手く表示が行われないことになります。

このような処理の仕方をノンブロッキング処理と呼びます。ネット系のプログラムでは良く使用される方法です。

## 座標軸の移動

今までの知識で、図 6-4 のような画像を作ろうとすると、ちょっと大変です。このような画像を作るために、必要となる座標軸の移動（座標変換）について説明します。

Processing では、座標軸は図 6-5 のように決まっています。この座標軸を基準に図形の描画が行われています。そこで、図 6-4 のような斜めになった図形を描くためには、描画の基準となる座標軸が傾いていれば可能です。このように、基準となる座標軸を傾けたり、移動させたりすることを座標変換と呼んでいます。平面の場合にの座標変換は、下の式のようなもので表すことが出来るものです。しかし、コンピュータグラフィックなどで

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} f \\ g \end{pmatrix}$$

は余り一般的な座標変換は扱わずに、次の 4 種類の特別な座標変換とそれを組み合わせたものを扱います。CG では、最初の 3 つを良く使用します。

1. 平行移動：translate
2. 回転：rotate
3. 拡大・縮小：scale
4. 剪断（傾け）：shear

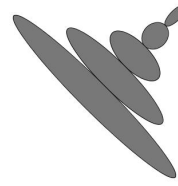
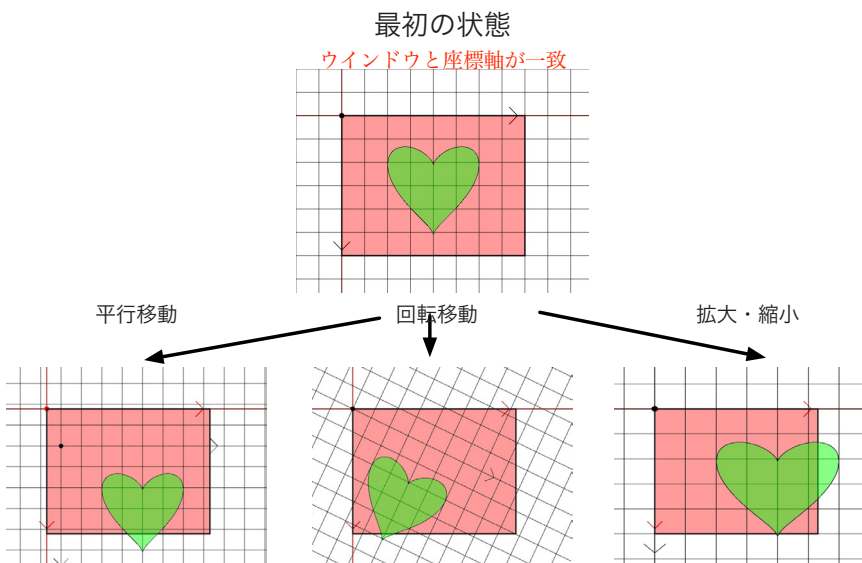


図 6-4

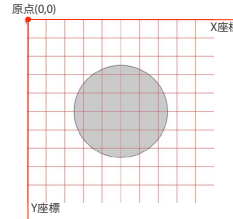


図 6-5

画用紙を傾けて絵を描き、その画用紙をものと向きに戻すと、傾いた絵になっていますよね。

座標変換の詳しい話は、線形代数学やグラフィクス基礎論で学びます。

正確には、逆行列をもっているようなものに限定されますが。

Processing でも、applyMatrix 関数を利用すると一般的な座標変換も取り扱うことが出来ます。

Processing では、scale は余り利用することが少ないような気が。図形の枠線の大きさも拡大・縮小されてしまうので。

実は、Processing では 3 次元物体の表示を行うことができます。そのため、3 次元空間での座標変換の関数も持っています。関数名はほとんどおなじです。

表 6-6 座標変換に関わる関数その 1

関数	意味
translate(x,y)	「現在の原点」を移動させる関数。「現在の X 軸」方向に x、「現在の Y 軸」方向に y だけ「現在の原点」を移動させる。移動した先が新たな「現在の原点」となります。座標軸の向きは変わりません。
rotate(angle)	「現在の原点」を中心に X 軸と Y 軸を回転させる関数。引数 angle は、回転角度の指定はラジアンで行います。
scale(s)	「現在の座標軸」を s 倍する。つまり、現在の 1 の長さが s となる。
scale(sx,sy)	「現在の X 軸」の長さを sx 倍、「現在の Y 軸」の長さを sy 倍する。
randians(angle)	angle 度をラジアンでの値に変換する関数。

### 平行移動

平行移動が一番簡単な座標変換です。平行移動では、原点の位置を移動させます。原点の位置を移動させるだけです。X 軸や Y 軸の向きは変化しません。座標変換を行う関数を実行すると、その度に座標軸が移動していきます。サンプル 6-15 は、これを確認するプログラムです。

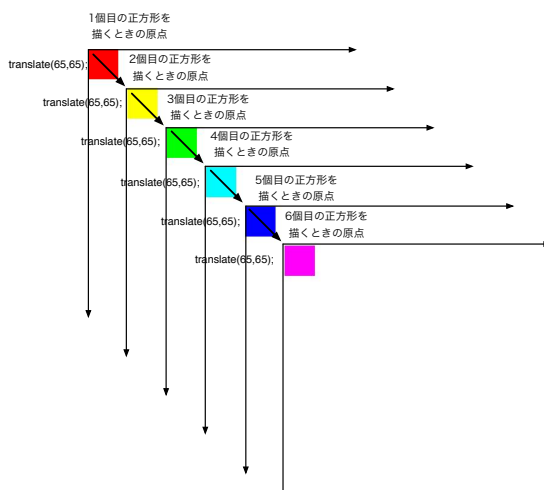


図 6-6

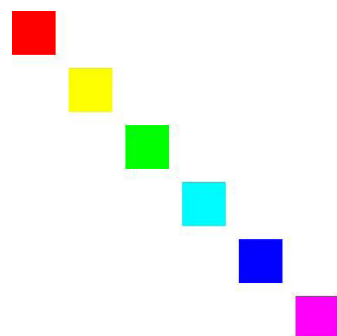
### translate を利用した例その 1 サンプル 6-15

```
size(400,400);
colorMode(HSB,359,99,99);
background(0,0,99);
noStroke();
for(int x=0;x <6;x++){
  fill(60*x,99,99);
  rect(0,0,50,50); // 原点 (0,0) で正方形を描く
  translate(65,65); // 原点を X 軸方向に 65、Y 軸方向に 65 移動させる
}
```

サンプル 6-15 は for 命令を利用して、色を変えながら 6 個の正方形を描くプログラムです。for 命令の繰り返し部分の rect 関数は、毎回同じ場所 (0,0) で正方形を描いています。ところが、このプログラムを実行してみると、異なった場所に正方形を描かれています。

数学的には、変換の合成（簡単にいうと、行列のかけ算）になっています。

実は、この座標軸の移動ですが、image 関数などの実行にも有効です。



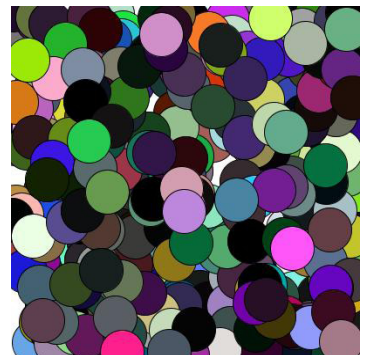
これは、なぜでしょうか？それは、rect 関数の後に実行している translate 関数に理由があります。つまり、rect 関数で正方形を描いた後に、「translate(65,65);」で原点の位置 (0,0) を動かしているからです。

つまり、一番目の赤色の正方形を描くときには、通常のウィンドウの左上に原点がある状態で rect(0,0,50,50) が実行されるので、左上に正方形が描画されます。その後、transalte(65,65) が実行されるので、「現在の原点」(ウィンドウの左上) が「現在の X 軸」方向に 65、「現在の Y 軸」方向に 65 だけ移動します。つまり、ウィンドウの左上から横方向に 65、縦方向に 65 だけ移動した場所に「現在の原点」が移動します。ですから、2 番目の黄色の正方形を描くときには、この「現在の原点」を基準に描画を行うので、rect(0,0,50,50) を実行すると、赤色の正方形より少し右下の部分に描かれることとなります。その後、再び translate(65,65) が実行されるので、「現在の原点」が「現在の X 軸」方向に 65、「現在の Y 軸」方向に 65 だけ移動します。つまり、ウィンドウの左上から横方向に 130、縦方向に 130 だけ移動した場所に「現在の原点」が移動します。3 番目の緑色の正方形を描くときには、この「現在の原点」を基準に描画を行うので、rect(0,0,50,50) を実行すると、黄色の正方形より少し右下の部分に描かれることとなります。この後、translate(65,65) を実行するので、「現在の原点」が「現在の X 軸」方向に 65、「現在の Y 軸」方向に 65 だけ移動します。つまり、ウィンドウの左上から横方向に 195、縦方向に 195 だけ移動した場所に「現在の原点」が移動します。このような操作を繰り返すので、徐々に右下に移動しながら、正方形が描かれるようになります。

translate 関数を利用した、別のサンプルを示します。このサンプル 6-16 では、ellipse 関数を利用して、毎回「現在の原点」に直径 50 の円を描いています。ただし、ellipse 関数を実行するまえに、translate 関数を呼び出して、「現在の原点」の位置を変更しています。そのために、異なった位置に円が描画されています。なお、draw 関数の一番先頭では、「現在の原点」はウィンドウに左上の位置に初期化されています。

### translate を利用した例その 2 サンプル 6-16

```
void setup(){
  size(400,400);
  colorMode(HSB,359,99,99);
  smooth();
  background(0,0,99);
}
// 次ページに続く
```





```
void draw(){
// この場所では、「現在の原点」はウインドウに左上に初期化されている
fill(random(360),random(100),random(100)); // 色はランダム
transalte(random(width),random(height)); // 「現在の原点」を移動
ellipse(0,0,50,50); // 「現在の原点」を中心に円を描画
}
```

## 回転移動

平行移動だけだと、図形を描画する位置を変更するだけ対応出来るので、あまりありがたみがありません。ここで説明をする回転移動と組み合わせると描画できる形状がより豊富になります。

rotate 関数による回転移動は、「現在の原点」を中心として座標軸の回転を行います。回転を考える際には、回転方向の向きが問題となりますが、Processing では図 6-7 のようになっています。

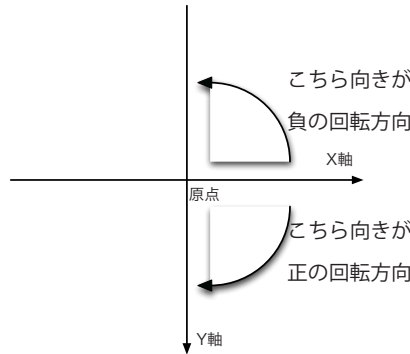


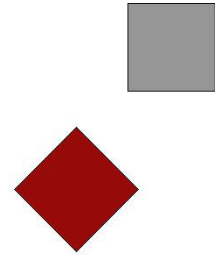
図 6-7 回転の向き

rotate 関数を実行しただけでは、「現在の原点」の位置は移動しません。座標軸の傾きだけが変わります。

まず、シンプルな rotate 関数を使用したサンプル 6-17 を示します。

### rotate を利用した例 サンプル 6-17

```
size(400,400);
background(255);
stroke(0);
fill(150);
rect(200,0,100,100);
rotate(PI/4);
fill(150,10,10);
rect(200,0,100,100);
```



サンプル 6-17 では、2 回 rect(200,0,100,100) を実行することで、2 つの正方形を描いていますが、異なった場所に描かれています。1 番目の灰色の正方形が描かれたときには、座標変換の関数を一切実行していないので、ウインドウの左上に原点があり、横方向の左から右方向に X 軸が、上から下方向に Y 軸が位置しています。その座標軸の状態です。rect(200,0,100,100) を実行するので、灰色の正方形が、ウインドウの上に水平の描画が

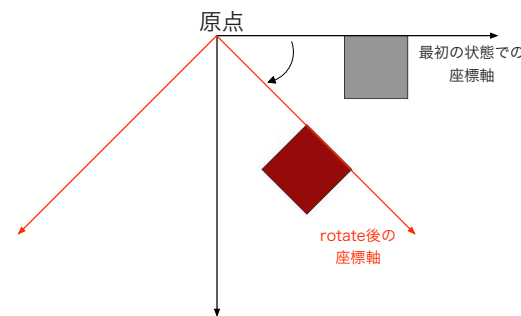


図 6-8 rotate 前後の座標軸

されます。この後、rotate(PI/4) を実行するので、「現在の原点」の位置は変わりませんが、「現在の原点」を中心に PI/4 (=45 度) だけ、X 軸と Y 軸を回転させます。その後、再び rect(200,0,100,100) を実行するので、画面の中央部分に傾いた赤い正方形が描かれることにな

前にも説明しましたが、PI は円周率を表す定数です。

ります。

rotate 関数は「現在の原点」を中心に「現在の座標軸」を回転させるので、rotate 関数単体では、使い道が限られてしまいます。座標変換を行う関数を実行するたびに、「現在の原点」や「現在の座標軸」が移動していくので、translate 関数を利用して「現在の原点」を回転の中心に移動させ、その後、rotate 関数を実行すると、任意の場所で座標軸の回転を行うことができます。

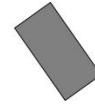
サンプル 6-18 は、マウスカーソルの位置で長方形を回転させるものです。

### translate と rotate を利用した例その 1 サンプル 6-18

```
float angle = 0; // 「現在の座標軸」の回転角度

void setup(){
  size(400,400);
  rectMode(CENTER);
  smooth();
  fill(128);
  stroke(0);
}

void draw(){
  // この時点では、「現在の原点」と「現在の座標軸」は初期位置
  background(255);
  // 「現在の原点」をマウスカーソルの位置に移動
  translate(mouseX,mouseY);
  rotate(angle); // 「現在の座標軸」を angle だけ回転させる
  rect(0,0,50,100);
  angle = angle + PI/180; // 回転角度を増やす
}
```



translate 関数で、回転の中心となる「現在の原点」を適切な場所に移動させます。その後、rotate 関数で「現在の座標軸」を回転させます。これにより、希望する場所での回転を実現できます。

サンプル 6-18 では、最初に translate 関数で「現在の原点」の位置をマウスカーソルの場所に移動させます。その後、rotate 関数で「現在の座標軸」を angle だけ回転させます。その後、回転角度を示す変数 angle の値を少しだけ増加 ( $PI/180 = 1$  度) させます。draw 関数が呼び出されるたびに、回転角度が増加するので、マウスカーソルの位置で長方形が回転しているように見えます。

サンプル 6-19 では、translate(width/2, height/2) で「現在の原点」をウインドウの中心に移動させます。その後、色を変えながら ellipse(150,0,60,60) で円を描画します。円の描画後、rotate 関数を使って、24 度ずつ「現在の原点」を中心に「現在の座標軸」を回転させます。これにより、円周上に円を配置することが出来ます。図 6-9 では、3 種類の座標軸が表示されています。1 つ目は translate 直後の

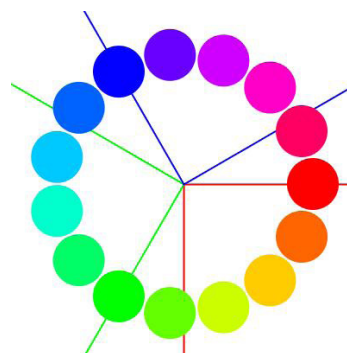


図 6-9 rotate 後の座標軸

座標軸、2つ目は rotate 関数を 5 回実行した直後の座標軸、3つ目は rotate 関数を 10 回実行した直後の座標軸となっています。

### translate と rotate を利用した例その 2 サンプル 6-19

```
size(400,400);
colorMode(HSB,359,99,99);
smooth();
background(0,0,99);
noStroke();
// 「現在の原点」をウインドウの中心に移動
translate(width/2,height/2);
for(int angle = 0;angle < 360;angle += 24){
  fill(angle,99,99); // 描画色の変更
  ellipse(150,0,60,60); // 円の描画
  rotate(radians(24)); // 現在の座標軸を 24 度回転させる
}
```

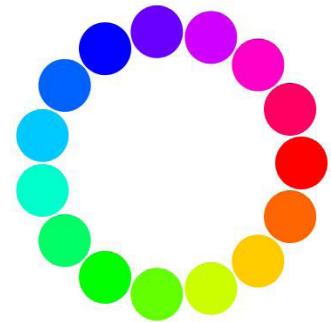
サンプル 6-20 は、translate 関数と rotate 関数を上手く利用して、画像を作成したものです。今までのサンプルプログラムとは異なり、「rotate → translate」が組みになって、繰り返し実行しています。つまり、X 軸方向にちょっと移動して、少し向きを変えろという処理になっています。

### translate と rotate を利用した例その 3 サンプル 6-20

```
size(400,400);
colorMode(HSB,359,99,99);
smooth();
noStroke();
background(0,0,99);
translate(240,60); // スタート位置に移動
for(int i=0;i<12;i++){
  fill(i*30,99,99); // 描画色の設定
  stroke(i*30,99,99);
  line(0,0,150,0); // 座標軸を表示
  line(0,0,0,150);
  rect(0,0,40,40); // 正方形の描画
  rotate(radians(30)); // 「現在の原点」を中心に座標軸を回転
// 「現在の原点」を「現在の X 軸」の正の方向に 80 移動
  translate(80,0);
}
```

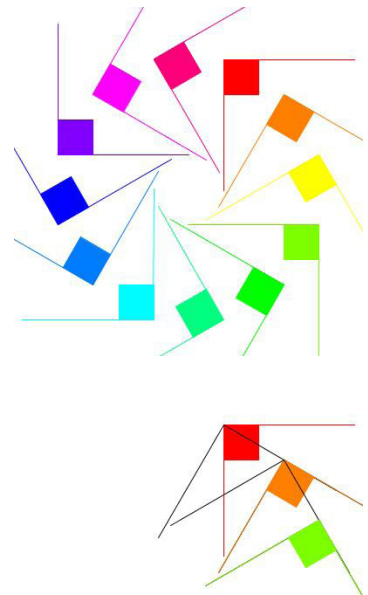
### 拡大・縮小

scale 関数は、translate 関数や rotate 関数に比べると、利用の機会は少ないですが、座標軸の拡大・縮小が行えます。この関数は、座標軸の目盛りを拡大・縮小します。scale 関数の実行結果は、「現在の座標軸」に対して有効です。従って、scale 関数を呼び出す前の図形は関わりません。「現在の座標軸」の目盛りの大きさを変えてしまうの



回転角度はラジアンで指定する必要がありますので、radians 関数を用いて、ラジアンに変換しています。

この処理は円運動の簡易的なモデルになっています。



### 途中までの描画状況

で、線の太さなども変わってしまいます。一応、サンプルをのせておきます。

### scale を利用した例 サンプル 6-21

```
size(400,400);
smooth();
background(255);

for(int x = 80; x < width;x += 80){
  fill(x % 256,10,10);
  ellipse(x,height/4,50,50);
}

scale(0.5); // 「現在の座標軸」 の目盛りを半分の長さにする
for(int x = 80; x < width;x += 80){
  fill(x % 256,10,10);
  ellipse(x,height,50,50);
}
```

### 座標軸の記憶

何回も座標変換を行っていくと、どんどん「現在の座標軸」が移動していきます。プログラムによっては、「前の座標軸」の状態に戻りたいということがおきます。これを実現するために、Processing 言語では pushMatrix 関数と popMatrix 関数が用意されています。pushMatrix 関数は、「現在の座標軸」の状況を一時的に記憶します。逆に、popMatrix 関数は、「現在の座標軸」を一時的に記憶されている「過去の座標軸」に変更します。

表 6-7 座標変換に関わる関数その 2

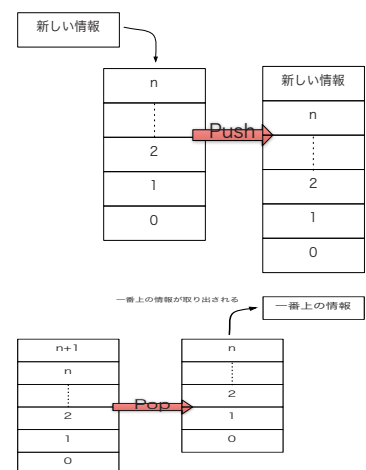
関数名	意味
pushMatrix()	「現在の座標軸」を一時的に記憶する。
popMatrix()	「現在の座標軸」を一時的に記憶されている「過去の座標軸」に変更します。つまり、pushMatrix 関数で記憶させた座標軸に戻すということを行います。
resetMatrix()	「現在の座標軸」を初期状態に戻します。

pushMatrix 関数と popMatrix 関数による座標軸の記憶は、普通の変数によるものとは、ちょっと異なっています。次のような制限があります。

1. popMatrix 関数の説明で書かれている「過去の座標軸」とは、この popMatrix 関数を呼び出す直前に呼び出された pushMatrix 関数が保存した「現在の座標軸」です。
2. popMatrix 関数を実行すると「過去の座標軸」は消えてしまいます。
3. pushMatrix 関数と popMatrix 関数が一対のものになっている。この辺りの話をやり出すと、ちょっと面倒なので、今回はあまり

このあたりの設計は、OpenGL と呼ばれる 3D-CG 用の API の影響を強く受けています。

このような情報の記憶の仕方をスタック (stack) と呼んでいます。スタックを利用して、情報を記憶することをプッシュ (push)、記憶されている情報を取り出すことをポップ (pop) と呼びます。

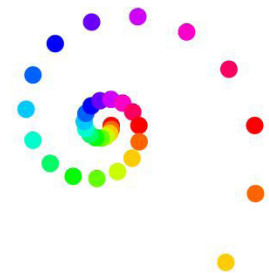


深入りはしません。

### pushMatrix と popMatrix 利用した例 サンプル 6-22

```
size(400,400);
smooth();
colorMode(HSB,359,99,99);
background(0,0,99);
noStroke();

// 「現在の原点」をウインドウの中心に移動させる
translate(width/2,height/2);
float len = 10;
for(int angle=0;angle < 1080;angle += 24){
  pushMatrix(); // 「現在の座標軸」を一時的に記憶
  rotate(radians(angle)); // 「現在の座標軸」を回転させる
  translate(len,0); // 「現在の原点」を X 軸方向に len だけ移動させる
  fill(angle % 360,99,99); // 塗りつぶし色の決定
  ellipse(0,0,20,20); // 円の描画
  popMatrix(); // 直前に記憶した座標軸の状況を「現在の座標軸」にする
  len *= 1.1; // 移動量を 1 割増やす
}
```



### resetMatrix 利用した例 サンプル 6-23

```
size(200,200);

smooth();
background(255);
translate(width/2,height/2); // 「現在の原点」をウインドウ中央に移動
fill(255,10,10); // 「現在の原点」を中心に赤色の円を描く
ellipse(0,0,100,100);

resetMatrix(); // 「現在の原点」を初期状態（ウインドウの左上）に移動
fill(10,255,10); // 「現在の原点」を中心に緑色の色の円を描く
ellipse(0,0,100,100);
```

## 応用：時計の制作

△  
7 回学習した内容に加えて、時間を取得する方法があれば、時計を作ることが出来ます。Processing では、現在の時刻を取得できる関数が用意されています。

表 6-8 時間に関わる関数

関数名	意味
hour()	現在の時間の時 (0 ~ 23 の整数) を返す関数。
minute()	現在の時間の分 (0 ~ 59 の整数) を返す関数。
second()	現在の時間の秒 (0 ~ 59 の整数) を返す関数。
year()	現在の年を返す関数。
month()	現在の月 (1 ~ 12 の整数) を返す関数。



関数名	意味
day()	現在の日 (1 ~ 31 の整数) を返す関数。
millis()	プログラムを実行してからの時間をミリ秒単位で返す。

1 秒 = 1000 ミリ秒です。

サンプル 6-24 ではデジタル時計のサンプルです。text 関数では、文字列 (String) しか表示できません。そこで、str 関数を利用して、強制的に int 型を String 型に変換しています。実は、draw 関数の中は、サンプル 6-25 のように書いても同じ動作となります。これは、式「hour()+":"+minute()+":"+second()」を見ると、数値データ同士の加算ではなく、文字列の連結と判断できるので、自動的に hour() などの値を String 型に変換してくれます。

hour()+":" などのように、数字と文字列に対して、+ を計算しようとしているので、+ は加算ではなく、文字列の連結と判断しています。

### デジタル時計 サンプル 6-24

```

PFont font;

void setup(){
  size(400,64*3);
  smooth();
  font = loadFont("Helvetica-128.vlw");
  textFont(font,64);
  textAlign(CENTER);
  rectMode(CENTER);
  fill(0);
}

void draw(){
  background(255);
  String h = str(hour()); // 時間を String 型に変換
  String m = str(minute()); // 分を String 型に変換
  String s = str(second()); // 秒を String 型に変換
  String t = h + ":" + m + ":" + s; // 表示する文字列作成
  int hs = textAscent()+textDescent(); // 表示する文字列の高さを取得
  text(t,width/2,height/2,width,hs);
}

```

### デジタル時計の一部サンプル 6-25

```

void draw(){
  background(255);
  // 表示する文字列作成
  String t = hour() + ":" + minute() + ":" + second();
  int hs = textAscent()+textDescent();
  text(t,width/2,height/2,width,hs);
}

```

サンプル 6-26 では、アナログ時計の秒針の部分だけを表示するサンプルです。0 秒時には鉛直上方向に秒針が来る必要があります。初期状態での座標軸は、原点を中心に  $-\pi/2$  だけ回転させた位置にあることに注意が必要です。

## 秒針だけのアナログ時計 サンプル 6-26

```
void setup(){
  size(400,400);
  smooth();
}
void draw(){
  background(255);
  // 「現在の原点」をウインドウの中心に移動
  translate(width/2,height/2);
  float angle = -90+6*second();// 現在の秒から、秒針の角度を求める
  rotate(radians(angle)); // 「現在の x 軸」を秒針方向に傾ける
  fill(50);
  triangle(0,5,180,0,0,-5); // 秒針を三角形として表示
}
```

### おまけのサンプル

**座**標変換を利用すると、複雑な図形を表示出来るようになります。  
やっていることは単純で、「現在の座標軸」を色々と移動させながら、`line(0,0,0,-100)` で直線を描いているだけです。

フラクタルと呼ばれる図形の描画の基礎となるプログラムです。フラクタルはCGにおける自然物を再現するにしばしば用いられる手法です。

### おまけ サンプル 6-23

```
size(400,400);
smooth();
stroke(0);
background(255);
translate(width/2,height);

line(0,0,0,-100);
translate(0,-100);
pushMatrix(); // (a)
rotate(-PI/6);
line(0,0,0,-100);
translate(0,-100);
pushMatrix(); // (b)
rotate(-PI/6);
line(0,0,0,-100);
popMatrix(); // (b)
// 右上に続く

rotate(PI/6);
line(0,0,0,-100);
popMatrix(); // (a)
rotate(PI/6);
line(0,0,0,-100);
translate(0,-100);
pushMatrix(); // (c)
rotate(-PI/6);
line(0,0,0,-100);
popMatrix(); // (d)
rotate(PI/6);
line(0,0,0,-100);
```