プログラミングプロジェクト (仮) – 画像の生成 (総合実習)

久野 靖 (電気通信大学)

2017.8.16

1　はじめに

今回の内容は「総合実習」であり、自分で計画して「美しいと考える画像」をプログラムで生成して頂きます。したがって課題は「B 課題」のみです。新しい内容は (オプションの位置付けのものを除き) 出て来ません。今回の目標は次のことがらです。

- 自分が生成したいと思う画像のために何が必要かを考える
- 画像の内容に合わせてプログラム構造を実現していく

この資料ではまず皆様が画像を生成するのに当たって役に立つかも知れないオプションを説明し、そのあと前回の演習問題の解説をします。

2　画像の生成に関連する補足

2.1　三角形や凸多角形を塗るには

前回は円の中を塗りつぶすことだけやりましたが、「図形内という条件を記述する」方法についてもう少し具体例を示しましょう。たとえば、XY 軸と平行な長方形だったら、その XY 座標の小さい側の角を (x_0, x_1)、大きい側の角を (x_1, y_1) とした場合、条件は次のように表せますから簡単ですね。

$$\{\ (x,y) \mid x_0 \le x \le x_1 \wedge y_0 \le y \le y_1\ \}$$

しかし、XY 軸に対して傾けたい場合は、もっと一般的に考える必要があります。例として三角形を取り上げましょう。三角形は 3 つの辺で囲まれた領域ですよね (当り前だ)。1 つの直線は、平面を 2 つの半平面に分けます。直線だと指定しにくいので、直線に含まれる線分を $(x_0, y_0) - (x_1, y_1)$ で指定することにして、この半平面の点の集合は次の式で表されます。

$$\{\ (x,y) \mid (x_1 - x_0)(y - y_0) - (y_1 - y_0)(x - x_0) \ge 0\ \}$$

なぜそうなるかというと、上の条件式は、ベクトル $\overrightarrow{(x_0,y_0)-(x_1,y_1)}$ と $\overrightarrow{(x_0,y_0)-(x,y)}$ の外積が正であるという条件であり、一般に起点を共有する位置ベクトル $\vec{\mathrm{a}}$ と $\vec{\mathrm{b}}$ の外積 $\vec{\mathrm{a}} \times \vec{\mathrm{b}}$ の符号は $\vec{\mathrm{a}}$ から見て $\vec{\mathrm{b}}$ が左にある場合には正、右にある場合には負になるからです。

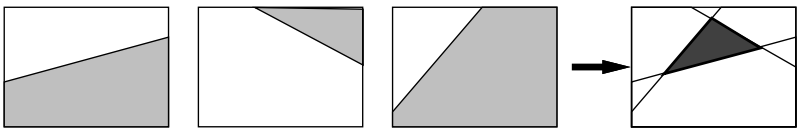

図 1: 三角形は 3 つの半平面の共通集合

そして、半平面が定義できたら、3 つの半平面に「ともに」含まれる点 (つまり共通集合) が三角形となります (図 1)。ということは、条件で言えば上に記したような半平面の条件を 3 つ「すべて」満たす点、ということになります。

三角形に限らず、凸な (へこみの無い) 多角形についても同様の考え方で定義することができます。そして傾いた長方形 (太さのある線分は細長い長方形だと考えることができます) も、こちらの方法で定義することができます。

2.2 おまけ 2: 楕円を塗るには

図形によっては、上述のように直接 2 次元座標上での条件を記述するのは厄介なものがあります。たとえば楕円を考えてみましょう。皆様は楕円の一般式を言えますか?

しかし、よい方法があります。楕円は円を「引き延ばして作る」ことができますね。ですから、原点を中心とした、たとえば横の半径が 3、縦の半径が 2 の楕円があったとして、それを「横に $\frac{1}{3}$ 倍、縦に $\frac{1}{2}$ 倍に縮めると」半径 1 の円になるはずです。そして、半径 1 の円内にあるかどうかは簡単に判定できます。つまり、$p(x, y)$ を $p'(\frac{x}{3}, \frac{y}{2})$ に写像してから円内の判定をすればよいわけです (図 2)。原点にない楕円や軸の回転した楕円は? これらも、原点の移動や座標の回転をしてから判断すればいいわけですね。[1]

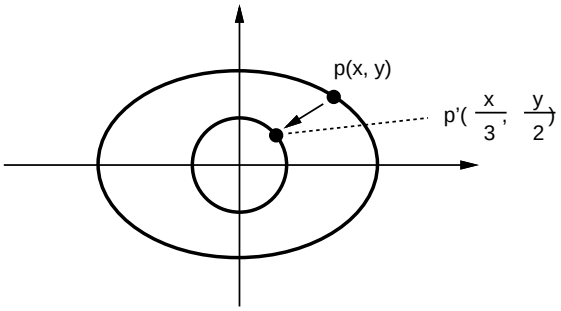

図 2: 楕円の内部かどうかの判定

2.3 おまけ 3: フラクタル

フラクタルな図形というのは、再帰性のある図形 (その図形の一部分が、全体と相似になっているような図形) を言います。たとえば、図 3 を見ると、「正方形の 4 隅に小さい正方形がくっついている」という構造が繰り返されています。

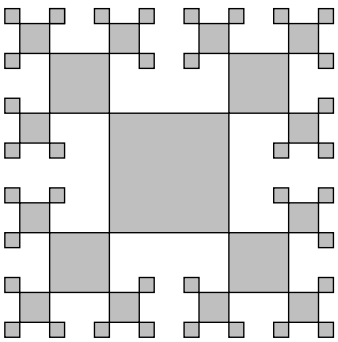

図 3: フラクタルな図形の例

こういうのは、再帰的なメソッドで作れます。どのみち、図形が小さくなりすぎたら描けないので、それが終了条件となります。たとえば次のような擬似コードを考えてみてください (正方形を描くメソッド fillsquare は別にあるものとします。)

- squares: 中心 x, y, 1 辺 2 × len の正方形フラクタルを描く

[1] こういう変換のことを総称してアフィン変換とか呼びますが、まあ用語はそれとして、適切な行列を作って掛けるとかでできます。

- もし `len < 1` ならば 戻る。
- fillsquare(x, y, len)。
- half ← len / 2。
- squares(x+len+half, y+len+half, half)。
- squares(x+len+half, y-len-half, half)。
- squares(x-len-half, y+len+half, half)。
- squares(x-len-half, y-len-half, half)。

なお、このようにきっちり機械的にやると人工物ぽくなりますが、乱数を使って「大きさがランダムに変動したり」「子供が確率的にできたりできなかったり」すると、自然物ぽくなります (自然界はフラクタルだと言われている)。Ruby では乱数は次の 2 つの方法で使えます。

- `rand()` — 0 以上 1 未満の実数値の一様乱数が得られる。
- `rand(`N`)` — N は整数として、0 から $N-1$ までの整数の一様乱数が得られる。

3 前回演習問題の解説

3.1 2 次元配列

これは簡単なのでコードだけ示します。(c) や (d) は if 式を使うのが素直でしょう。

```
Array.new(5) do|i| Array.new(5) do|j| j-i end end   # (a)
Array.new(5) do|i| Array.new(5) do|j| i**j end end  # (b)
Array.new(5) do|i| Array.new(5) do|j| if i==j then 1 else 0 end end end
Array.new(5) do|i| Array.new(5) do|j|
               if (i-2).abs == (j-2).abs then 1 else 0 end
             end end      # (d)
```

3.2 様々な図形

細かい演習問題は省略して、今回は図 4 のようなさまざまな図形を描くプログラムを説明します (このために、透明度の機能も追加しました)。

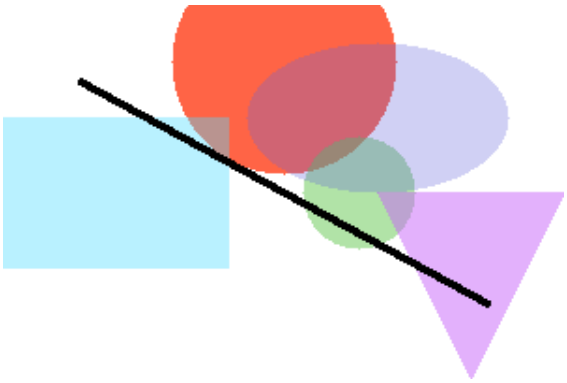

図 4: 生成されたさまざまな図形

レコード定義、画像の書き出しについては変更はありません。色に「透明度」をつけて塗れるようにするには、現在の色と塗りたい色を α (透明度) に応じて比例配分すればよいでしょう。それを行うように `pset` を変更し、あとはすべてこれを使っています。

```
def pset(x, y, r, g, b, a)
  if x < 0 || x >= 300 || y < 0 || y >= 200 then return end
```

```
    $img[y][x].r = ($img[y][x].r * a + r * (1.0 - a)).to_i
    $img[y][x].g = ($img[y][x].g * a + g * (1.0 - a)).to_i
    $img[y][x].b = ($img[y][x].b * a + b * (1.0 - a)).to_i
  end
```

次に、円を描く (正確には円の形に色を塗る) メソッド fillcircle を示します。ただし、前回は「画像全部の点」に対して判定していましたが、今回は処理を軽くするため、必要にしてできるだけ少ない範囲の点だけを列挙して判定します。それには、円に含まれ得る点の X 座標、Y 座標の範囲 (中心 (x_c, y_c) 半径 r として $x_c \pm r$ と $y_c \pm r$) をまず考え、その範囲内の各点 (x, y) について $(x - x_c)^2 + (y - y_c)^2 \leq r^2$ を満たすなら円内にあるものとしてその点の色を設定します:

```
  def fillcircle(x, y, rad, r, g, b, a)
    j0 = (y-rad).to_i; j1 = (y+rad).to_i
    i0 = (x-rad).to_i; i1 = (x+rad).to_i
    j0.step(j1) do |j|
      i0.step(i1) do |i|
        if (i-x)**2+(j-y)**2<rad**2 then pset(i,j,r,g,b,a) end
      end
    end
  end
```

XY 座標や半径に小数点つきの値が入れられても動作するように、調べる範囲を計算した時に結果をメソッド to_i で整数にしています。

長方形を描く fillrect は、円よりもっと簡単で、単にその範囲全部を pset するだけです。[2]

```
  def fillrect(x, y, w, h, r, g, b, a)
    j0 = (y-0.5*h).to_i; j1 = (y+0.5*h).to_i
    i0 = (x-0.5*w).to_i; i1 = (x+0.5*w).to_i
    j0.step(j1) do |j|
      i0.step(i1) do |i| pset(i, j, r, g, b, a) end
    end
  end
```

楕円を描く fillellipse は、円と同様で、ただし縦横をそれぞれ縦横の半径で割ってから半径 1 の円に入っているかどうかで判定すればよいでしょう:

```
  def fillellipse(x, y, rx, ry, r, g, b, a)
    j0 = (y-ry).to_i; j1 = (y+ry).to_i
    i0 = (x-rx).to_i; i1 = (x+rx).to_i
    j0.step(j1) do |j|
      i0.step(i1) do |i|
        if (i-x).to_f**2/rx**2 + (j-y).to_f**2/ry**2 < 1.0
          pset(i, j, r, g, b, a)
        end
      end
    end
  end
```

三角形を描く filltriangle は、凸多角形を塗る fillconvex というのを作ってそれを呼ぶようにしました:

[2]回転させたい場合は後の「直線」を援用してください。

```
def filltriangle(x0, y0, x1, y1, x2, y2, r, g, b, a)
  fillconvex([x0, x1, x2, x0], [y0, y1, y2, y0], r, g, b, a)
end
```

fillconvex は X 座標、Y 座標をそれぞれ配列で渡し、最後には最初と同じ要素を重複して入れておくことにします。また、点を指定する順序は「左回り」である必要があります (これらの理由は後述します)。

fillconvex では、まず座標の範囲は配列に入っている X 座標や Y 座標の最大と最小を求め (最大と最小は前に演習でやったようなものですが、実は配列にはメソッド max と min があって最大と最小を計算してくれるのでそれを使っています)、その後各点についてそれが図形の内側にあるなら塗ります:

```
def fillconvex(ax, ay, r, g, b, a)
  xmax = ax.max.to_i; xmin = ax.min.to_i
  ymax = ay.max.to_i; ymin = ay.min.to_i
  ymin.step(ymax) do |j|
    xmin.step(xmax) do |i|
     if isinside(i, j, ax, ay) then pset(i, j, r, g, b, a) end
    end
  end
end
```

図形の内側にあるかどうかは isinside で判定します。isinside は、与えられた点が「いずれかの辺の右側にある」なら図形の外にある、そうでなければ内側にあるか線上にある、と判断します。

```
def isinside(x, y, ax, ay)
  (ax.length-1).times do |i|
    if oprod(ax[i+1]-ax[i], ay[i+1]-ay[i], x-ax[i], y-ay[i]) < 0
      return false
    end
  end
  return true
end
```

右側にあるかどうかは、辺の線分の**ベクトル** (vector) と、線分の起点から調べたい点までのベクトルの**外積** (outer product) を計算して、負なら右側と判定します (このために左回りで周囲を指定するという条件が必要なのでした):

```
def oprod(a, b, c, d)
  return a*d - b*c;
end
```

このほか、線分が直交かどうか調べるにはベクトルの**内積** (inner product) が 0 かどうか調べればよいなど、図形処理においてベクトルの考え方はさまざまに活用できます。このような、プログラムで幾何学的な図形の計算を行うものを一般に**計算幾何学** (computational geometry) と呼びます。

線を描く fillline ですが、2 点の XY 座標と「線の幅」を指定します:

```
def fillline(x0, y0, x1, y1, w, r, g, b, a)
  dx = y1-y0; dy = x0-x1; n = 0.5*w / Math.sqrt(dx**2 + dy**2)
  dx = dx * n; dy = dy * n
  fillconvex([x0-dx, x0+dx, x1+dx, x1-dx, x0-dx],
```

```
                [y0-dy, y0+dy, y1+dy, y1-dy, y0-dy], r, g, b, a)
  end
```

線分のベクトルからそれと直交するベクトルを計算し、その長さが線の幅の半分になるようにします。あとは線分の両端点と幅ベクトルを加減することで細長い長方形ができますから、それを `fillconvex` で塗ればよいわけです。

では最後に、さまざまな絵を描くメソッドを示します:

```
  def mypicture
    fillcircle(150, 30, 60, 255, 100, 70, 0.0)
    fillcircle(190, 100, 30, 100, 200, 80, 0.5)
    fillrect(60, 100, 120, 80, 80, 220, 255, 0.6)
    fillellipse(200, 60, 70, 40, 100, 100, 220, 0.7)
    filltriangle(200, 100, 300, 100, 250, 200, 200, 100, 250, 0.5)
    fillline(40, 40, 260, 160, 4, 0, 0, 0, 0.0)
    writeimage("t.ppm")
  end
```

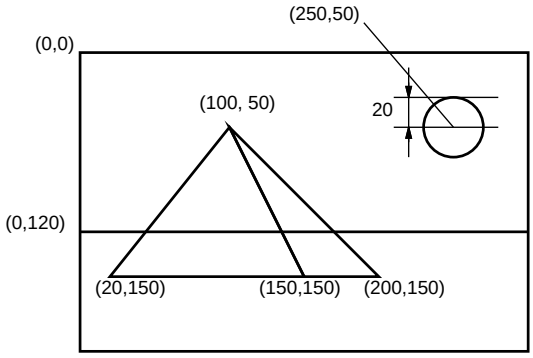

図 5: ピラミッドの絵の設計図

演習 7 の「美しい」については皆様にお任せしているのですが、たとえば風景みたいに構図のある絵を描くとしたら、やっぱり何らかの設計が必要ではと思います。たとえば「海にうかぶピラミッドと太陽」という絵を描くものとします。まず、図 5 のように方眼紙などで構図の設計をして、それからそれぞれの図形を色指定して入れていく、みたいにすればそれらしくなるのではないでしょうか (図 6)。

```
  def mypicture1
    fillrect(150, 60, 300, 120, 180, 240, 250, 0.0);
    fillrect(150, 160, 300, 80, 20, 90, 200, 0.0);
    filltriangle(100, 50, 150, 150, 20, 150, 120, 70, 20, 0.0);
    filltriangle(100, 50, 200, 150, 150, 150, 160, 90, 80, 0.0);
    fillcircle(250, 50, 20, 255, 0, 0, 0.0);
    writeimage("t1.ppm")
  end
```

なかなか大変でしたが、このように手続きを次々に作っていくことで、大きなプログラムでも見通しよく組み立てて行けることが納得いただけたかと思います。

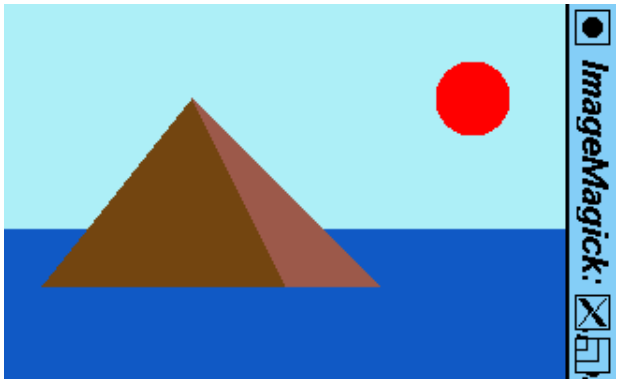

図 6: ピラミッドの絵の画像