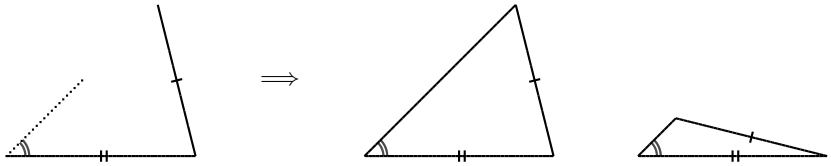
2.5 図形と証明 II

図形の証明を行うとき、基本となるのは三角形の合同条件である。

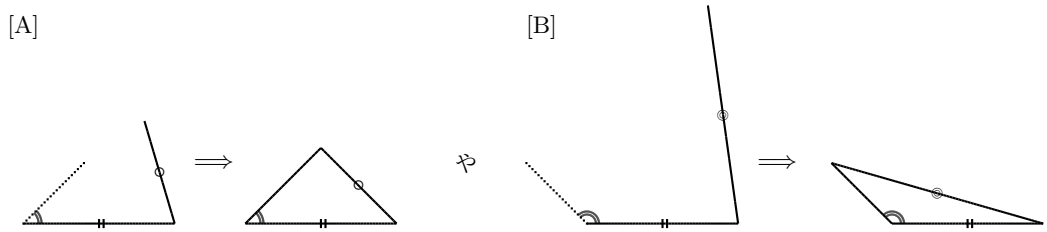
1. 対応する 3 組の辺の長さ（3 辺）がそれぞれ等しいとき
2. 対応する 2 組の辺の長さとその間の 1 組の角の大きさ（2 辺と間の角）がそれぞれ等しいとき
3. 対応する 1 組の辺の長さとその両端の 2 組の角の大きさ（1 辺と両端の角）がそれぞれ等しいとき

正確には “対応する” とか “長さ”、“大きさ” などの言葉を省いてはいけないのだが、面倒なので括弧内の省略した言い回しをすることがほとんどだろうか。2. は “2 辺と間の角” の制限の下で成り立つ条件で、“2 辺とひとつの角” ではいけない。なぜなら



のようなことが考えられるからである。つまり、三角形がひとつに決まらないのである。

ところが、同じような 2 辺とひとつの角でも



のような場合は、三角形がひとつに決まる。

三角形の合同条件とは三角形が必ずひとつに決まることが前提なので、2 辺と 1 角を条件とする場合は、角が 2 辺の間になければならないのである。しかし図のように、2 辺の長さが決まっていて間にない角がわかっている場合でも三角形がひとつに決まることがある。

その例は、固定されてない辺の長さが、もう一辺と**なす角**の延長線までの距離に等しいときである（図 A）。この場合は 1 点で接する状態であるから、三角形はひとつに決まる。また別の例は、2 辺の間にない角が、2 辺のうち短い辺—厳密には長くない辺—にある場合である（図 B）。この場合は角の延長線で一ヶ所しか交わるところがないので、三角形はひとつに決まる。

かと言って、これらを合同条件に含めるには条件が複雑である。しかし、合同条件として使えないわけではないので一考の余地があるかもしれない。

一方で 3. については、“1 辺とふたつの角” でもよい。なぜなら、ふたつの角の大きさがわかれば残りの角の大きさも決まるので、自動的に 1 辺と両端の角が決まるからである。

残念なことに Swift は合同条件を用いて証明をしてくれないので、三角形の合同を見るためには重ね合わせで済ませるしかない。図形の移動を見るためにスライダーを利用するのは手軽だが、せっかくのタッチパネルの性質が活かせていない。自然な操作をするなら、ジェスチャーで移動や回転ができるとよいだろう。

[Triangle2.playground]

```
import UIKit
import PlaygroundSupport // *

class ViewController: UIViewController {

    var P = CGPoint(x: 250, y: 200)
    var Q = CGPoint(x: 290, y: 200)
    var R = CGPoint(x: 290, y: 170)
    //（挿入予定地 0）
    let triang = CAShapeLayer()

    override func viewDidLoad() {
        super.viewDidLoad()

        //（挿入予定地 1）
        let handle = UILabel(frame: //__
                            CGRect(x: 245, y: 160, width: 50, height: 50))
        handle.addGestureRecognizer(UIPanGestureRecognizer(target: //__
                                self, action: #selector(handleMoved)))
        handle.isUserInteractionEnabled = true
        handle.layer.borderWidth = 1
        handle.layer.borderColor = UIColor.yellow.cgColor
        view.addSubview(handle)

        putTriangle(A: P, B: Q, C: R)
        //（挿入予定地 2）
    }

    func putTriangle(A: CGPoint, B: CGPoint, C: CGPoint) {
        let gline = UIBezierPath()
        gline.move(to: A)
        gline.addLine(to: B)
        gline.addLine(to: C)
        gline.addLine(to: A)

        triang.strokeColor = UIColor.blue.cgColor
        triang.fillColor = UIColor.white.cgColor
        triang.lineWidth = 2.0
        triang.path = gline.cgPath
```

```
38          view.layer.addSublayer(triang)
39      }
40
41      @objc func handleMoved(panGr: UIPanGestureRecognizer) {
42          let p: CGPoint = panGr.location(in: view)
43          let grip: UILabel = panGr.view as! UILabel
44          grip.center = CGPoint(x: p.x, y: p.y)
45
46          putTriangle(A: CGPoint(x: p.x-20, y: p.y+15), //__
                        B: CGPoint(x: p.x+20, y: p.y+15), //__
                        C: CGPoint(x: p.x+20, y: p.y-15))
47      }
48 }
49
50 let viewController = ViewController()                  // *
51 viewController.view.backgroundColor = UIColor.white    // *
52 PlaygroundPage.current.liveView = viewController       // *
53 PlaygroundPage.current.needsIndefiniteExecution = true // *
```

前節のプログラムと比較しながら見るとよいだろう。三角形の 3 個の頂点 P, Q, R と三角形描画のための triang は前節と同じで、回転のための変数はない。また、viewDidLoad() ではスライダーに代えてラベルを配置している。ジェスチャーで動かしたいのは triang のはずだから、ラベルの利用は少し奇異に感じるところである。でも、それなりの理由があるのだ。

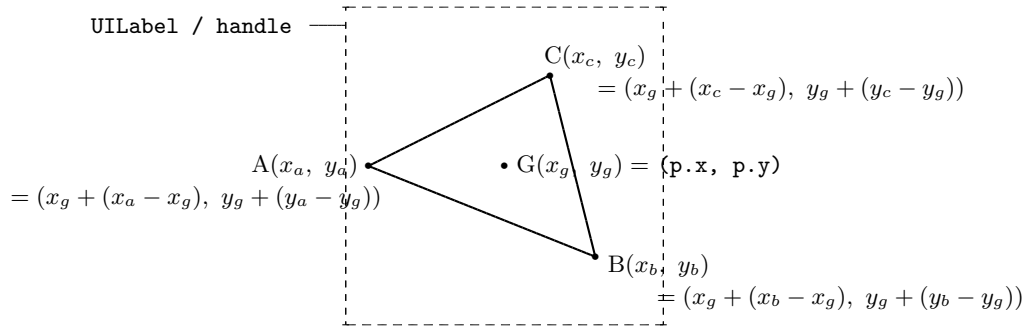
今回、利用するのは 17 行目に見られる UIPanGestureRecognizer() である。ところが、これはどんなものにも付属させられるわけではない。CAShapeLayer() で作成した三角形 triang は.addGestureRecognizer() をメンバーに持たないため、triang.addGestureRecognizer() と書くことができない事情がある。

そこで 16–21 行目において、UILabel() である handle に.addGestureRecognizer() を設定し、handle の移動に伴って CAShapeLayer() で作成した図形を移動することにした。handle の輪郭を黄色い線で見えるようにしているのは説明のためであり、アプリとして作るならこの行は不要だ。そのほうが三角形を直接動かしているように見える。

ただ、このままの handleMoved() では非常にまずい。handleMoved() 内の putTriangle() を見てほしい。46 行目だ。ここで三角形の 3 頂点の座標を指定するのだが、p.x には± 20、p.y には± 15 の調整が入っている。これは、初期値の三角形が底辺 40、高さ 30 の直角三角形であったため、ラベルである handle の中心 (p.x, p.y) から左右・上下に± 20・± 15 の位置が、ちょうど初期値の三角形の頂点と対応するからである。

プログラムはいずれにせよ三角形の初期値を設定しなくてはならないので、同時に (p.x, p.y) に対する調整をすれば問題はない。しかし、将来プログラムを実行してから三角形の頂点を入力す

る仕様にすることがあれば、プログラムの中で (p.x, p.y) に対する調整ができなくては困るだろう。その場合は、handle の中心 (p.x, p.y) から元の三角形の頂点を求める仕組みが必要になる。



図の座標は数学座標で表している。(p.x, p.y) だけが Swift 座標の意味で用いた。G の座標から A、B、C の座標を求めるにはどうすればよいだろう。たとえば A、G がどれだけ離れているかは、x 座標の差と y 座標の差を調べればよい。このとき、距離を考えるなら絶対値を計算することになるが、A が G からどれだけ離れているかは $A - G$ を計算すればよいことに注意しよう。すると負の値になることがあるが、それは A が G の左または下にあることを意味する。A の x 座標 x_a を例に取ると、G からの差は $(x_a - x_g)$ だから x_a は $x_g + (x_a - x_g)$ で求められることになる。実際、$x_g + (x_a - x_g) = x_a$ である。

以上のことをビュー座標で実装しよう。まず、CGPoint(x: p.x, y: p.y) は panGr が受け取った座標である。これはジェスチャー操作で指が触れている場所になる。指が画面を移動するたびにビューの位置が panGr に反映される。そして、ラベルの中心である grip.center へ代入されるわけだ。この結果、指は常にラベルの中心にあることになるので、三角形の中心が grip.center に一致していれば、指で三角形を動かしているように見えるだろう。

では、三角形の中心とはどこだろうか。三角形には五心と呼ばれる**外心**、**内心**、**重心**、**垂心**、**傍心**の他に、いくつもの “心” がある。どれもそれなりに三角形の臍（へそ）に違いないが、ここでは計算も楽な重心を使うことにする。つまり、重心を grip.center に一致させるのである。重心の計算は簡単だ。

```
let G = CGPoint(x: (P.x + Q.x + R.x)/3, y: (P.y + Q.y + R.y)/3)
```

でよい。これは（挿入予定地 1）に追加しよう。ただ、重心だけでは元の三角形の位置を特定できないので、三角形の頂点が重心からどれだけ離れているか知る必要がある。したがって、それらを

格納しておく変数が必要になる。最初に var dP = CGPoint(x: 0, y: 0) のように準備しておくとよい。他に dQ、dR も同様なので

```
dP = CGPoint(x: 0, y: 0)
dQ = CGPoint(x: 0, y: 0)
dR = CGPoint(x: 0, y: 0)
```

と初期化しておく。これらは（挿入予定地 0）に追加する。dP 等の実際の計算は三角形を置いたあとに計算すればよいだろう。計算は

```
dP = CGPoint(x: P.x-G.x, y: -P.y+G.y)
dQ = CGPoint(x: Q.x-G.x, y: -Q.y+G.y)
dR = CGPoint(x: R.x-G.x, y: -R.y+G.y)
```

として（挿入予定地 2）に追加しよう。y 座標の正・負が逆なのはビューの y 座標が数学のそれとは違うことを思い出してほしい。

これで指の位置から三角形を特定できるようになった。handleMoved() 内の putTriangle() は

```
putTriangle(A: CGPoint(x: p.x+dP.x, y: p.y-dP.y),
            B: CGPoint(x: p.x+dQ.x, y: p.y-dQ.y),
            C: CGPoint(x: p.x+dR.x, y: p.y-dR.y)) // このように修正
```

でいいだろう。ここも y 座標の符号に注意だ。

最後にラベルの位置と大きさを修正しておこう。当初、ラベルの位置は

```
frame: CGRect(x: 245, y: 160, width: 50, height: 50)
```

としていたが、基点は枠の左上ではなく中心であり、それが三角形の重心 G であるから、枠の大きさを考えて

```
frame: CGRect(x: G.x-25, y: G.y-25, width: 50, height: 50) // このように修正
```

としておかないといけない。

関数 handleMoved は、簡単のため単に平行移動をする例を示しているに過ぎない。このとき grip.center の画面中心からの角度を求めれば、角度に応じた直角三角形の位置を計算できる。その場合は、前回のように計算が面倒だが回転移動を実装できる。また線対称移動なら、たとえば画面中央の縦線に対して裏返すような計算をすればよい。それだと任意の軸に対して対称移動することにはならないが、縦線対称のあと、回転移動と平行移動をすれば任意の軸に対して対称移動したことにできる。ある処理を一度で済ますより、基本的な数種類の処理を組み合わせて実現できるなら是非そうしよう。図形の任意の移動がみっつの基本移動の組み合わせで実現できるように、複雑な処理をするプログラムも単機能の組み合わせで実現できることは多い。

ex. **1** ジェスチャーによる平行移動のプログラムをジェスチャーによる回転移動のプログラムに修正してみよう。少し大変な作業かもしれない。

ところで、図形は単に`.move(to: )`と`.addLine(to: )`を使って描いているだけなので、単調な線画しか書けない。もう少し見栄えのする図形を描くにはグラフィックスコンテキストを利用するとよい。そうなるとコードが長くなるが、細かい設定が可能になるので状況に応じて使えばよいことである。この砂遊びでは見栄えは二の次で、なんとか動作するコードを短く書くことに主眼を置いているので、貧相な画像を許してもらいたい。