3.5 円と相似

ふたつの図形が**相似**であるとは、一方を拡大・縮小した上で平行・回転・対称移動をして重なる場合を言う。拡大・縮小は iPhone などの画面上でピンチ操作をするとできる。しかし、これでは画面が拡大・縮小されたのであって、あるひとつの図形が拡大・縮小されたわけではない。対象となる図形だけ拡大・縮小するようなプログラムを書いてみよう。

[Magnification.playground]

```
import UIKit
import PlaygroundSupport // *

class Triangle: UIViewController {

    var iA = CGPoint(x: 0, y: 0) // 初期（initial）配置の頂点
    var iB = CGPoint(x: 0, y: 0)
    var iC = CGPoint(x: 0, y: 0)

    var  A = CGPoint(x: 0, y: 0) // 拡大・縮小するの三角形の頂点
    var  B = CGPoint(x: 0, y: 0)
    var  C = CGPoint(x: 0, y: 0)

    func sync(P: CGPoint, Q: CGPoint, R: CGPoint) {
        iA = P; iB = Q; iC = R
         A = P;  B = Q;  C = R
    }

    func draw() -> CGPath {
        let line = UIBezierPath()
        line.move(to: A)
        line.addLine(to: B)
        line.addLine(to: C)
        line.addLine(to: A)
        return line.cgPath
    }
}

class ViewController: Triangle {

    let W: CGFloat = 400
    let H: CGFloat = 400
    let E: CGFloat = 20
    var tri = [Triangle(), Triangle()]
    var fig = [CAShapeLayer(), CAShapeLayer()]

    override func viewDidLoad() {
        super.viewDidLoad()

```

```
        tri[0].sync(P: CGPoint(x: 100, y: 100), //__
                    Q: CGPoint(x: 200, y: 100), //__
                    R: CGPoint(x: 200, y:  50))
        tri[1].sync(P: CGPoint(x: 150, y: 300), //__
                    Q: CGPoint(x: 200, y: 300), //__
                    R: CGPoint(x: 200, y: 200))

        var slider = [UISlider(), UISlider()]
        for i in 0...1 {
            slider[i] = UISlider(frame: //__
                    CGRect(x: 0, y: H+CGFloat(45*i), width: W, height: 30))
            slider[i].addTarget(self, action: //__
               #selector(sliderChanged), for: UIControlEvents.valueChanged)
            slider[i].minimumValue = 0
            slider[i].maximumValue = 2
            slider[i].setValue(1.0, animated: true)
            slider[i].tag = i
            view.addSubview(slider[i])
        }

        for i in 0...1 {
            fig[i].strokeColor = UIColor.red.cgColor
            fig[i].fillColor = UIColor.white.cgColor
            fig[i].lineWidth = 2
            fig[i].path = tri[i].draw()
            view.layer.addSublayer(fig[i])
        }
    }

    @objc func sliderChanged(sender : UISlider) {
        let r = CGFloat(sender.value)
        let t = sender.tag
        tri[t].A = CGPoint(x: tri[t].iA.x * r, y: tri[t].iA.y * r)
        tri[t].B = CGPoint(x: tri[t].iB.x * r, y: tri[t].iB.y * r)
        tri[t].C = CGPoint(x: tri[t].iC.x * r, y: tri[t].iC.y * r)
        fig[t].path = tri[t].draw()
        view.layer.addSublayer(fig[t])
    }
}

let viewController = ViewController()                  // *
viewController.view.backgroundColor = UIColor.white    // *
PlaygroundPage.current.liveView = viewController       // *
PlaygroundPage.current.needsIndefiniteExecution = true // *
```

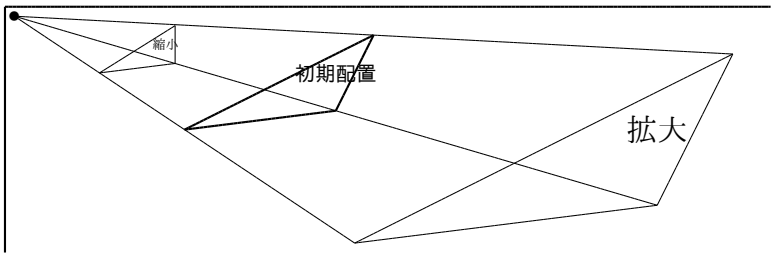
初めに `class Triangle: UIViewController` という見慣れないものが登場しているね。続けて `class ViewController: Triangle` というものもある。なんで似たようなものが 2 個あるんだ？いままでは、`class ViewController: UIViewController` が 1 個あっただけのはずなのに。

いままでのことから説明しておこう。Swift はプログラムを**クラス**単位で管理している。いままで私たちが書いてきたプログラムは `class ViewController { }`というクラスにまとめられていたのだ。そして Swift は、ここに書かれた命令群を忠実に実行していたのである。また、画面にテキストフィールドを配置できたのも私たちの指示に Swift が従ったからである。そしてテキストフィールド自体もプログラムを書いて作成されている。でも私たちは、そんなの書いてこなかったはずだ。なぜなら、そのために必要なプログラム `UIViewController` を**継承**したからである。

どうやって継承したかといえば、こうやった（→ `class ViewController: UIViewController`）のである。つまり私たちは、必要なプログラムが継承されたクラスに、さらに自分たちが実行させたい命令を付け加えてプログラムを書いていたことになる。たとえば `aTextF = UITextField()` は `aTextF` にはテキストフィールドの機能が与えられる。なぜなら、代入することで `UITextField` クラスを継承したからだ。Swift では、たとえば `UpperCamelCase()` のように大文字から書き始めるものはクラスを表すので、`UITextField()` はクラスである（型変換 `String()` などと混同しないように）。よって `UITextField` クラスを代入された変数 `aTextF` は、クラスの機能を持ったのだ。

で `class Triangle` だが、これは三角形を作成するための自作クラスで、`UIViewController` の機能を継承している。そして `class ViewController` は `:` によって `Triangle` を継承していることがわかる。結局 `ViewController` は、`Triangle` クラスと `UIViewController` クラスの機能を継承したのである。

ここでは `Triangle` クラスに三角形を書く命令をまとめたので、複数の三角形を作る際、その都度線を引かなくて済むのが利点だろう。しかも三角形 `tri` を配列で扱うようにしたので、`tri[番号]` で一括して処理できる。`triA`、`triB`、`triC`、...などと書くより効率的なことに注目してほしい。



そうは言っても `Triangle` クラスはちょっと変だ。こうなってしまったのは、拡大・縮小の中心を原点—画面の左上角—にとり、そこから初期配置の三角形に投射するようにしたからである。

つまり、初期配置の座標が固定されていないと、拡大・縮小の座標を求められないからである。要するに `Triangle` クラスは固定座標と移動座標を両方扱うことになるのだ。初期配置用の変数 `iA`

などの他に、拡大・縮小用の変数 A などがあるのはそのためである。いま変数と言ったが、iA や A などのクラス内変数は**プロパティ**と呼ばれている。したがってこれまでプロパティを使ったときと同様に、Triangle クラスが持つ特性として初期配置の頂点 iA の情報を知りたければ.iA を付けて取得することになる。

では Triangle クラスは何ができるのだろうか。続きを見れば、そこには 2 個の関数 sync と draw があることがわかる。sync() は、単に初期配置の頂点と拡大・縮小の頂点に代入しているだけだ。また、draw() は、変数 line を UIBezierPath() で初期化して線を引いていることがわかる。直後に line.cgPath を返しているので三角形を配置まではしない。要するに Triangle クラスは、三角形の頂点の座標を元に線を引く機能を持っているに過ぎない。ここでも関数と称したが、sync() などのクラス内関数は**メソッド**と呼ばれている。したがってプロパティ同様、クラスに仕事—たとえば三角形の頂点を線で結ぶこと—をしてもらうには.draw() を付けて実行することになる。

```
class クラス名 { // クラス名は大文字で始まる
    var 変数 = （初期値）
    func 関数名() -> 戻り型 { }
}
```

クラスは実際にどう使われるのだろうか。まず [Magnification.playground] は 2 個の三角形の拡大・縮小の見本である。そこで class ViewController で最初に定義するのは、いつもどおりのビューサイズの他に、三角形の情報を持つ変数 tri とその情報を元に三角形を描くための台紙にあたる fig である。これまでにも、たとえば変数 aText をテキストフィールドとして初期化する際 aText = UITextField(あれこれ) と代入したことを思い出してほしい。同じく、変数 tri を三角形のさまざまな情報で初期化するには tri = Triangle() とするのだ。ただし、いまは 2 個の三角形の拡大・縮小を想定しているので、配列として 2 個まとめて初期化している。fig も同じだ。もし、2 個以上の三角形を扱いたければ配列の要素を増やせばよい。

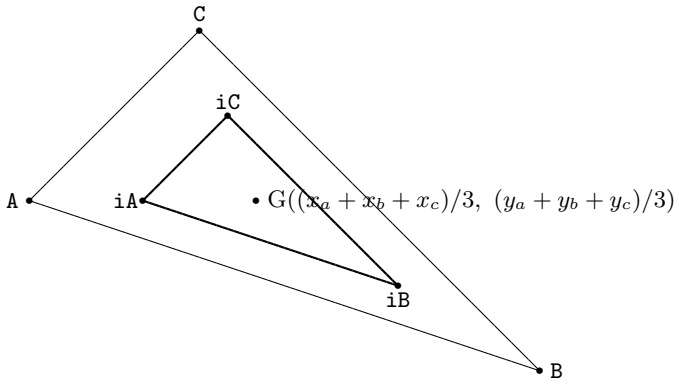
続いて viewDidLoad() を見よう。ここで 2 個の三角形の初期値を与えている。三角形は配列で定義したはずだから、2 個の三角形は tri[0] と tri[1] である。三角形の頂点は Triangle クラスのメンバーである関数 sync によって代入される。tri[0].sync() の引数に頂点の座標を与えると、初期配置の頂点と拡大・縮小の頂点に同時に代入される。これを見ればクラスも結構便利なものだと感じるだろう。

次はスライダーだ。この位置で初期化を行っているのは、`viewDidLoad()` が最初に実行されたときに配置すればよく、他の関数などで利用することがないからである。拡大・縮小が個別の三角形に行われるように、三角形ひとつにつきスライダーひとつを割り当ててある。三角形を増やしたら `for` 文で回す回数 `i` の範囲を変えるだけでよい。しかしコードを書き足す必要はない。

スライダーの機能はどの三角形のものでも同じなので、使用する`#selector()` はひとつでよい。では、`#selector()` はどうやってスライダーの区別をするのだろう。それはタグ付けで解決できる。`slider[i].tag = i` によって、`slider[i]` には番号 `i` がタグ付けされる。このとき動かす三角形は `tri[i]` である。すべてが番号 `i` で統一されているのがわかるだろうか。

スライダーによる拡大・縮小は、画面の左上を中心にして行なうことにしていたので、三角形の頂点の座標は単に元の頂点をスライダーが示す倍率 `sender.value` だけ乗ずればよいので簡単だ。

ex. **1** たとえば三角形の重心を中心に、拡大・縮小ができるようにプログラムを修正してみよう。



あとは三角形をビューに配置すれば完成である。三角形の配置も `for` 文で行なっているので、三角形が増えれば指定範囲を変えればよい。三角形の頂点の座標はすでに与えていたので、ここでは配置だけをする。三角形の色や線の太さの指定はこれまでどおりだが、描画は `Triangle` クラスのメソッドを使う。それが `fig[i].path = tri[i].draw()` だ。

ex. **2** ビューに表示される三角形の色は皆同じである。三角形ごとに異なる色にするにはどうすればよいだろうか。実際にコードを修正して、異なる三角形が異なる色で表示されるようにしてみよう。

プログラム全体を見渡した感想はどうかな？ クラスのために面倒なことやコードの量が増えた感じがしただろうか。そうかもしれない。でも、プログラムはひとつの入れ物に何でもかんでも詰め込んではいけないのだ。保守が困難になるからである。関数やクラスは扱いに苦労することが

あっても、コードの再利用や管理をしやすくする。でも私が書くプログラムはインチキくさいので、正統なプログラムの書き方は別のどこかで学んでもらうのがよい。