第14回

# Quartzでコントロールを作る

スマートフォンの認知度を一般に広めただけでなく、ソフトウェア開発においても新しい波を作り出してしまったiOS。開発者たちは何を見、どう考えているのか。毎回入れ替わりでiOS向けアプリケーション開発に関わるエンジニアに登場いただき、企画・開発のノウハウやアプリの使いこなし術などを披露してもらいます。

㈱デンソーアイティーラボラトリ　吉田 悠一　YOSHIDA Yuuichi
http://sonson.jp

## はじめに

今回は、プログラマがUIViewを使ったオリジナルのコントロールを作るための描画方法について一考してみます。iOS SDKの特徴の1つにCocoa Touch[注1]のコントロールの美しさが挙げられます。そのコントロールには、一通りすべての機能がそろっていますが、それでも足りないものがあるのが現状です。

たとえば、Map Kit[注2]で使われているポップアップ(**図1**)は、MKMapViewでしか使えません。同等の機能を持つポップアップを一般のUIView上で使うには、コントロールを自作する必要があります。本稿では、コントロールの自作のために、iOS/Mac OS XのAPIであるQuartzを使った描画の方法を説明します。まず、数ある手段の中でQuartzを使う理由から説明しましょう。

**注1)** iOSのアプリケーションフレームワークで、Objective-Cで実装されている。

**注2)** アプリケーションでGoogleの地図ビューを扱うためのフレームワーク。

**▼図1　Map Kitのポップアップ**


## 3つの描画手段

ユーザインターフェースの見栄えは、アプリケーションの使い方をユーザに伝える重要な要素です。スライダーコントロールは、スライダーらしい見栄えを持つ必要があります。ボタンコントロールであれば、通常時と押下時でボタンの外観を切り替える必要があるでしょう。また、インターフェースの反応を提示するために、同様にボタンコントロールであれば、通常時から押下時へ見栄えを切り替える(たとえば、ボタンが引っ込んだような視覚効果を加える)アニメーションを加えることもあります。iOS SDKでこれらを実現するには、次の3つの方法があり、それぞれに長所と短所があります。

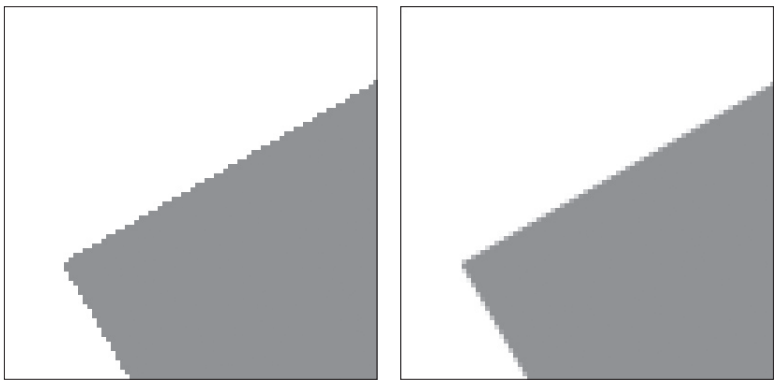
### Cocoa Touchのコントロールと画像リソースの組み合わせ

この手法は、実装の簡易性が長所です。UIButtonの背景画像をオリジナルのものに置き換えれば、見栄えが異なるボタンを簡単に作成できます。どれほど複雑なUIも基本的にUIButtonとUIImageViewの組み合わせで作成

可能です。このことは、意匠デザイナとプログラマが簡単に分業できるという観点からかなり魅力的です。また、CoreAnimationとの組み合わせが簡単であることも大きな長所と言えます。

一方、その短所は、コントロールのスケールやサイズ変更に弱いことです。UIImageの引き伸ばし機能を使って対応することもできますが、Retinaディスプレイが登場したときのようにディスプレイの解像度が変わると、そのたびに画像を作り直すための作業コストが発生します。デバイスの入れ替えやアップデートがいきなり起こる（Appleの秘密主義たる所以ですね）ことを考えると、このリスクは大きくはありませんが、小さくもありません。

▼図2　OpenGL ESで描画した場合（左）とQuartzで描画した場合（右）



## OpenGL ES

OpenGL ES[注3]を使って描画する方法です。OpenGL ESの長所は何と言っても高速に描画できる点と、シェーダ[注4]などの高い表現力を描画に応用できる点です。

しかし、コーディングが煩雑であること、プログラマが多くはないこと、デザイナから見てOpenGL ESのコードや概念が簡単ではないことなどを考慮すると、開発コストの観点から、コントロールを作るのにあまり良い方法とは言えません。また、**図2**に示すように、Quartzで描画するときれいに描画できる直線も、OpenGL ESでは自前でアンチエイリアス[注5]をかける必要があるなど、低レイヤならではのコーディングの手間もあります。

注3）組込みシステム向けの3Dグラフィックスエンジン。
注4）陰影やグラデーションのレンダリングを行う機能。
注5）オブジェクトの輪郭が滑らかにきれいに見えるように、画像に対して施す処理。

## Quartz

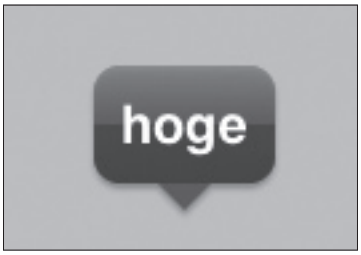
Quartzは、iOS/Mac OS Xの描画用のAPIです。長所は、OpenGLライクな文法でありながら、OpenGL ESほど低レイヤではない高レイヤのAPIが用意されている点です。たとえば、グラデーション、点線、多角形、ベジェ曲線のAPIが用意されており、Mac OS Xの Aquaインターフェースに似た質感を表現できます。また、すべてベクトルグラフィックスで表現できるため、Retinaディスプレイへの対応などもほとんど意識する必要がなく、自動でアンチエイリアスも行ってくれる、かなり便利なAPIと言えます。

一方で、その短所は、コードの移植性の低さと煩雑さです。OpenGL ESであれば、他のプラットフォームへの移植も部分的には簡単にできますが、Quartzで書いたコードは、iOSあるいはMac OS Xでしか使うことはできません。また、Quartzのプログラミングスタイルや、ベクトルグラフィックスでの描画に慣れる必要があり、簡易であるとは言いがたく、意匠デザイナと共同で作業を行うにもこの点は障壁になり得ると考えます。

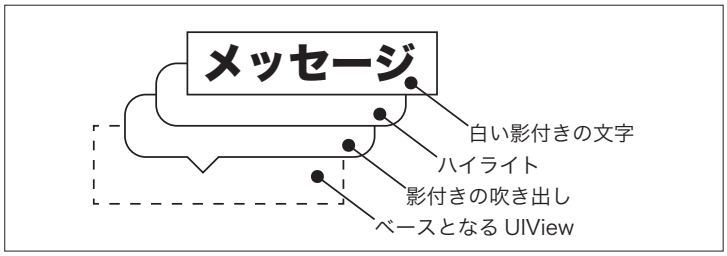
◆ ◆ ◆

以上のように、どの手法にも長短があり、甲乙をつけがたいのですが、本稿ではQuartzを使ってコントロールを作ることにします。その理由は、Quartzのレンダリングが美しいことと、Retinaディスプレイへの対応が不要であるこ

▼図3　完成したポップアップ




▼図4　ポップアップのデザイン




▼リスト1　UIViewのdrawRectメソッド

```
- (void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();
    // レンダリングコードをここに書く
}
```

▼リスト2　drawRectメソッドによる描画の例

```
- (void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();
    // レンダリングコードをここに書く
    CGContextSetRGBFillColor(context, 0.0, 1.0, 0.0, 1.0);
    CGContextFillRect(context, CGRectMake(10, 10, 100, 100));
}
```

▼リスト3　パスによる四角形の描画

```
CGContextRef context = UIGraphicsGetCurrentContext();
CGContextMoveToPoint(context, 10, 10);
CGContextAddLineToPoint(context, 20, 10);
CGContextAddLineToPoint(context, 20, 20);
CGContextAddLineToPoint(context, 10, 20);
CGContextAddLineToPoint(context, 10, 10);
CGContextClosePath(context);
CGContextDrawPath(context, kCGPathStroke);
```

との2点です。誌面の都合上、Quartzのすべて、あるいは**図3**に示すポップアップの描画方法のすべてを書くことはできませんが、図3に示すコントロールを書くための基礎について説明していきたいと思います。

## Quartzによるポップアップの自作

図3に示すポップアップは、**図4**のようなパーツに分けることができます。図4のデザインに従って、Quartzで描画していけばポップアップコントロールを自作できるわけです。

このポップアップを描画するには、角丸の四角形、グラデーション(ハイライトを表現する)、影付け、テキストの描画コードが必要です。次に、基本的なコーディングから始め、図4で解説した、4つの部品の描画コードについて説明していきます。

▼図5　Quartzで描画した吹き出し付きの角丸の四角形

### Quartzの基本的な描画コード

Quartzのレンダリングコードは、UIViewのdrawRectメソッドに記述します(**リスト1**)。

Quartzでは、コンテキスト(リスト1のcontext)を使ってレンダリングを行います。たとえば、座標の(10, 10)に大きさ(100, 100)で緑色で塗りつぶされた四角形を描画するには、**リスト2**のように書きます。

それでは、図のコントロールを作るために必要なプリミティブの描画コードを追っていきましょう。これ以降、本稿で紹介するAPIの詳細は、Appleのドキュメント[注6]を参照してください。まずは、角丸の四角形です。

### パスと角丸の四角形

**図5**のような角丸の四角形は、直線と90度分の円弧を組み合わせて、パスを作り、描画します。Quartzでは複雑な形状を描画する場合、

注6)『Quartz 2D Programming Guide』(http://developer.apple.com/library/mac/#documentation/graphicsimaging/conceptual/drawingwithquartz2d/Introduction/Introduction.html)など。

パスを使います。たとえば、四角形のパスを使って、線で四角形を描く場合は、**リスト3**のようになります。

パスを「追加して」「閉じて」「描画する」のが一連の流れです。CGContextDrawPath は、現在クローズしたパスを描画するAPIで、塗りつぶし、線のみ、塗りつぶしと線などの描画方法を指定できます。角丸の四角形を描画する場合も前述のとおりパスを追加していくわけですが、円弧を追加する際はCGContextAddArcToPointという便利なAPIを使って描画します。このAPIは、現在の点と円弧の始点(第2引数と第3引数で表される点)を通る直線と、円弧の始点と終点(第4引数と第5引数で表される点)を通る直線を接線とし、第6引数で指定された数値を半径とする円を描きます。

これらのAPIを組み合わせると、**リスト4**のように角丸の四角形を描くメソッドを実装できます。

**▼リスト4　角丸の四角形の描画**

```
- (void)drawRoundCornerRect:(CGRect)rect mode:(CGPathDrawingMode)mode
    radius:(float) radius {
    CGContextRef context = UIGraphicsGetCurrentContext();

    CGFloat minx = CGRectGetMinX(rect);
    CGFloat midx = CGRectGetMidX(rect);
    CGFloat maxx = CGRectGetMaxX(rect);
    CGFloat miny = CGRectGetMinY(rect);
    CGFloat midy = CGRectGetMidY(rect);
    CGFloat maxy = CGRectGetMaxY(rect);

    CGContextMoveToPoint(context, minx, midy);
    CGContextAddArcToPoint(context, minx, miny, midx, miny, radius);
    CGContextAddArcToPoint(context, maxx, miny, maxx, midy, radius);
    CGContextAddArcToPoint(context, maxx, maxy, midx, maxy, radius);
    CGContextAddArcToPoint(context, minx, maxy, minx, midy, radius);
    CGContextClosePath(context);

    CGContextDrawPath(context, mode);
}
```

**▼リスト5　角丸の四角形に影を付ける**

```
- (void)drawRectWithShadow {
    CGContextRef context = UIGraphicsGetCurrentContext();

    // 影を付ける前の状態を保存する
    CGContextSaveGState(context);
    UIColor *blackColor = [UIColor blackColor];
    CGContextSetShadowWithColor(context,
        CGSizeMake(0, 1), 2.0, [blackColor CGColor]);
    CGContextSetRGBFillColor(context, 0.0, 1.0, 0.0, 1.0);
    CGContextFillRect(context, CGRectMake(10, 10, 100, 100));

    // 影を付ける前の状態を復帰する
    CGContextRestoreGState(context);

    // 他のオブジェクトをここで描画する
}
```

## 影を付ける

次に、**図6**のように影を付けてみましょう。

影は、CGContextSetShadowWithColorというAPIで簡単に設定できます(**リスト5**)。第1引数はコンテキスト、第2引数は影を作る光源の位置のオフセット、第3引数はブラー(ぼかし)の程度を表す数値、第4引数は影の色になります。

注意しなければならないのは、その前後で利用する2つのAPI、CGContextSaveGStateとCGContextRestoreGStateです[注7]。CGContextSetShadowWithColorで影を設定してしまうと、それ以降に描画したものすべてに影が付いてしまいます。そのため、CGContextSaveGStateで

**▼図6　角丸の四角形に影を付ける**

**注7)** OpenGLに詳しい人は、glPushMatrixとglPopMatrixを思い出してください。

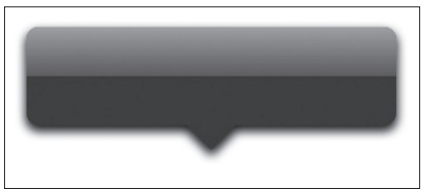
▼リスト6　クリッピングの実行

```
- (void)testClipping {
    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSaveGState(context);
    CGContextAddArc(context, 100, 100, 30, 0, 360, 0);
    CGContextClosePath(context);
    CGContextClip(context);

    UIColor *greenColor = [UIColor greenColor];
    CGContextSetFillColorWithColor(context, [greenColor CGColor]);
    CGContextFillRect(context, CGRectMake(100, 100, 50, 50));
    CGContextRestoreGState(context);
}
```

▼リスト7　グラデーションの色の定義

```
CGContextRef context = UIGraphicsGetCurrentContext();
CGColorSpaceRef space = CGColorSpaceCreateDeviceRGB();
CGFloat colors[] = {
    155.0 / 255.0, 155.0 / 255.0, 155.0 / 255.0, 1.0,
    70.0 / 255.0, 70.0 / 255.0, 70.0 / 255.0, 1.0,
};
CGGradientRef gradient =
    CGGradientCreateWithColorComponents(
        space, colors, NULL,
        sizeof(colors)/(sizeof(colors[0])*4));
CGColorSpaceRelease(space);
```

▼図7　グラデーションによるハイライトの表現



影を付ける前の状態を保存しておき、影が不要になった時点でCGContextRestoreGStateによってその保存していた状態に戻します。こうすると、影を付ける対象を正しく切り分けられます。

## グラデーションとクリッピング

次に説明するのはグラデーションです(**図7**)。Quartzでは、複雑なグラデーションを簡単に描画できます。ただし、任意の形状をグラデーションで塗りつぶすには、パスを設定してクリッピングを行う必要があります。

まず、クリッピングから説明します。円でクリッピングした状態で、緑色で塗りつぶした四角形を描画するサンプルコードを**リスト6**に示します。

リスト6を見るとわかるように、ここでもCGContextSaveGStateとCGContextRestoreGStateを使います。これは、クリッピングを一度設定してしまうと、これ以降すべての描画内容がクリッピングされてしまうためです。影を付けるときと同様に、クリッピングを設定する前の状態を保存しておき、クリッピングが不要になったときに保存しておいた状態に戻るようにコーディングします。

## グラデーションの描画

次に、グラデーションの描画を説明します。まずグラデーションの色を定義するCGGradientのインスタンスを**リスト7**のようなコードで生成します。

配列colorsには、グラデーションに使いたい色を指定します。2色以上を指定可能です。また、CGGradientCreateWithColorComponentsで生成したCGGradientのインスタンスは自動的に解放されないので、不要になったときにCGGradientReleaseを使って忘れずに解放してください。

生成したCGGradientのインスタンスを使って描画を行ってみましょう。描画はCGContextDrawLinearGradientを使って、**リスト8**のように簡単に実装できます。また、CGContextDrawRadialGradientを使うと放射状のグラデーションを描画できます。

リスト8のコードを実行した結果を見ると、グラデーションが塗られる領域が固定されることがわかります。そのため、クリッピングを用いて、任意の形状をグラデーションで塗りつぶします。

**リスト9**は、グラデーションをひし形で塗りつぶすサンプルコードです。これで、グラデーションで任意の形状を塗りつぶせるようになりました。ハイライトや光の反射といった微妙な表現をこれにより再現できます。

## 文字の描画

最後は、文字の描画です(図8)。文字列の描画は、文字列自体の処理もからんでくるので、話がややこしくなりがちです。そこで、NSStringを使った描画方法だけに話を絞り、文字の描画に関する基本的なAPIを紹介することにします。NSStringの描画メソッドの詳細は、『NSString UIKit Additions Reference』[注8]というドキュメントを参照してください。

文字を描画するには、次のように実装します。

```
NSString *str = @"Hello, world.";
[str drawAtPoint:CGPointMake(10, 10)
    withFont:[UIFont
boldSystemFontOfSize:12]];
```

drawAtPoint:withFont:を使えば、任意のフォントで、任意の点にNSStringの内容をレンダリングできます。その他にも、領域を指定して文字をレンダリングしたり、文字列の折り返し幅を指定してレンダリングしたりするAPIが用意されています。また、文字列を描画したときのレンダリングのサイズを得るAPIもあります。これらを使って、文字列のレイアウトを行いながら、レンダリングすることが可能です。

リスト10に、文字列を折り返しながらレンダリングし、さらにその背景を黄色に設定するサンプルコードを示します(図9)。これまでに説明したAPIを用いて文字列に影を加えたり、文字の形でクリッピングすることもできます。

リスト10では、UIColorの珍しいメソッドを使っ

注8) http://developer.apple.com/library/ios/#documentation/uikit/reference/NSString_UIKit_Additions/Reference/Reference.html を参照。

▼図8 ポップアップの文字の描画


▼リスト8 グラデーションの描画

```
- (void)drawGradientColor {
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGColorSpaceRef space = CGColorSpaceCreateDeviceRGB();
    CGFloat colors[] = {
        155.0 / 255.0, 155.0 / 255.0, 155.0 / 255.0, 1.0,
        70.0 / 255.0, 70.0 / 255.0, 70.0 / 255.0, 1.0,
    };
    CGGradientRef gradient =
        CGGradientCreateWithColorComponents(
            space, colors, NULL,
            sizeof(colors)/(sizeof(colors[0])*4));
    CGColorSpaceRelease(space);

    CGContextDrawLinearGradient(
        context, gradient,
        CGPointMake(100, 100), CGPointMake(100, 200),
        kCGGradientDrawsBeforeStartLocation |
        kCGGradientDrawsAfterEndLocation);

    CGGradientRelease(gradient);
}
```

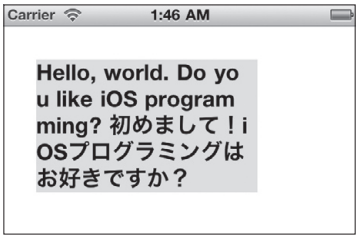
▼リスト9 ひし形のグラデーションの描画

```
- (void)drawGradientRhombus {
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGColorSpaceRef space = CGColorSpaceCreateDeviceRGB();
    CGFloat colors[] = {
        155.0 / 255.0, 155.0 / 255.0, 155.0 / 255.0, 1.0,
        20.0 / 255.0, 20.0 / 255.0, 20.0 / 255.0, 1.0,
    };
    CGGradientRef gradient =
        CGGradientCreateWithColorComponents(
            space, colors, NULL,
            sizeof(colors)/(sizeof(colors[0])*4));
    CGColorSpaceRelease(space);

    CGContextSaveGState(context);

    CGContextMoveToPoint(context, 160,160);
    CGContextAddLineToPoint(context, 240, 240);
    CGContextAddLineToPoint(context, 160, 320);
    CGContextAddLineToPoint(context, 80, 240);
    CGContextAddLineToPoint(context, 160, 160);
    CGContextClosePath(context);
    CGContextClip(context);

    CGContextDrawLinearGradient(
        context, gradient,
        CGPointMake(160, 160), CGPointMake(160, 320),
        kCGGradientDrawsBeforeStartLocation |
        kCGGradientDrawsAfterEndLocation);

    CGGradientRelease(gradient);
    CGContextRestoreGState(context);
}
```

**▼リスト10　折り返し文字列の描画**

```
- (void)layoutAndDrawText {
    CGContextRef context = UIGraphicsGetCurrentContext();

    // 引数
    CGPoint p = CGPointMake(30, 30);
    float width = 200;
    float height = 480;
    float fontSize = 20;

    NSString *str =
        @"Hello, world. Do you like iOS programming? "
         "初めまして!iOSプログラミングはお好きですか?";
    UIColor *yellow = [UIColor yellowColor];
    UIColor *black = [UIColor blackColor];
    UIFont *font = [UIFont boldSystemFontOfSize:fontSize];

    // 文字列のサイズ
    CGSize strSize =
        [str sizeWithFont:font
            constrainedToSize:CGSizeMake(width,height)
            lineBreakMode:UILineBreakModeCharacterWrap];
    CGRect renderingRect;
    renderingRect.origin = p;
    renderingRect.size = strSize;

    // 背景色のレンダリング
    [yellow setFill];
    CGContextFillRect(context, renderingRect);

    // 文字列のレンダリング
    [black setFill];
    [str drawInRect:renderingRect
        withFont:font
        lineBreakMode:UILineBreakModeCharacterWrap];
}
```

**▼図9　折り返し文字列の表示**




ています。UIColorには、次の3つのメソッドが用意されており、UIColorからQuartzの色の設定ができます。UIColorは、CGColorRefよりいくぶん扱いやすいので、これらのメソッドをうまく利用することをお勧めします。

```
- (void)set;
- (void)setFill;
- (void)setStroke;
```

## Quartz学習のために

Quartzを学ぶための情報源として、まずAppleのサンプルコード、QuartzDemo[注9]があります。このサンプルコードには、Quartzで描画できる典型的な例が多数実装されているので、Quartzを使って何ができて、何ができないかをさっと理解できる点と、基本的な描画コードをすべて読めるという点において、かなりお勧めです。また、これと併せて、Xcodeの開発者向けのドキュメントを読めば、たいていのことは理解できると思います。その他には、英語ですが、『Programming with Quartz: 2D and PDF Graphics in Mac OS X』[注10]という解説本もあります。

今回は、QuartzのベーシックなAPIのみを紹介しましたが、これらとCore Animationを組み合わせると、さらにUIKitらしいコントロールを作成できます。筆者は、github[注11]で拙作のポップアップビューのソースコードを公開しています。このポップアップビューは、Core Animationを組み合わせ、ポップアップアニメーションで開くなどのアニメーション効果も実装しています。BSD Licenseに留意いただいて、ソースコードを眺めてみてください。SD

注9）http://developer.apple.com/library/ios/#samplecode/QuartzDemo/Introduction/Intro.htmlを参照。
注10）David Gelphman、Bunny Laden 著、Morgan Kaufmann、2005年、ISBN978-0-1236-9473-7
注11）https://github.com/sonsongithub/を参照。

● 吉田 悠一（よしだ ゆういち）㈱デンソーアイティーラボラトリ

本業はコンピュータビジョン、ヒューマンインターフェースの研究。iOS用2ちゃんねるビューア「2tch」の中の人。共著書にiOSのAPIハックを集めた「iOS SDK HACKS」（オライリー・ジャパン刊、ISBN978-4-87311-472-9）がある。