iOS Scroll Viewプログラミングガイド

User Experience > Cocoa

2011-06-06

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

アップルジャパン株式会社
〒163-1450 東京都新宿区西新宿
3 丁目20 番2 号
東京オペラシティタワー
http://www.apple.com/jp/

目次

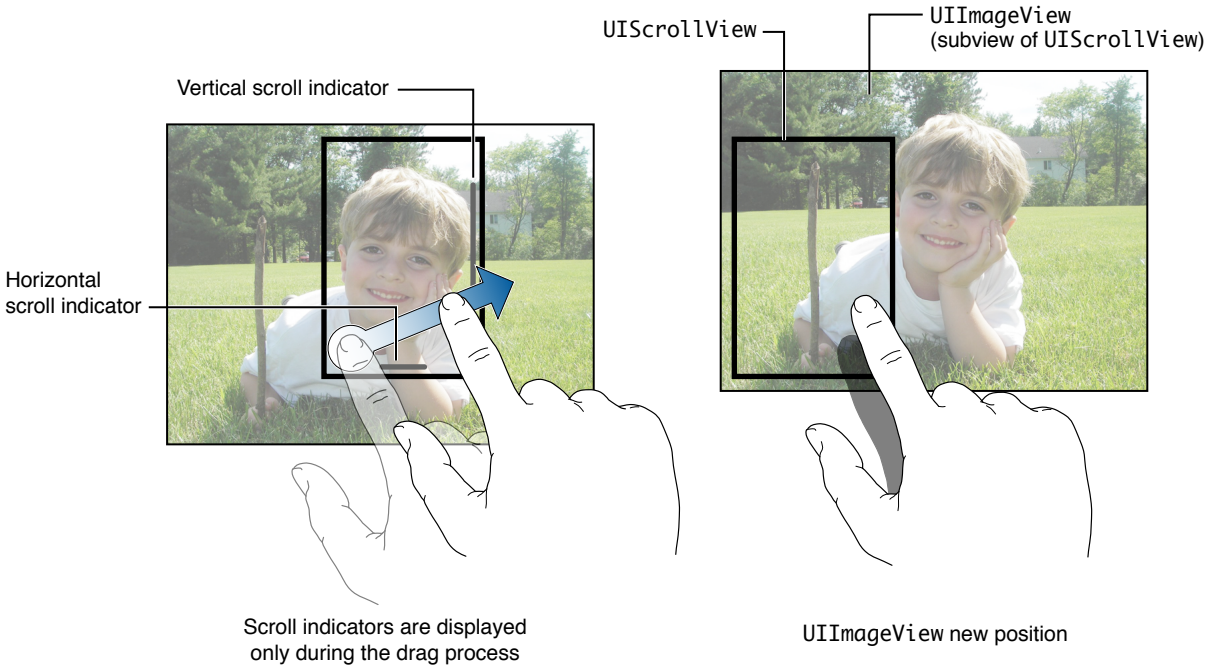
図、リスト

Scroll Viewプログラミングの概要

Scroll Viewは、iOSアプリケーションの中で表示や操作の対象となるコンテンツの全体が画面に入りきらない場合に使われています。Scroll Viewには主に次の2つの目的があります。

- ユーザが表示したいコンテンツの領域をドラッグして表示できるようにするため
- ユーザがピンチジェスチャを使ってコンテンツの表示を拡大縮小できるようにするため

次の図は`UIScrollView`クラスの一般的な使用方法を示します。サブビューは、一人の男の子の画像を含む`UIImageView`です。ユーザが画面上で指をドラッグすると、画像に対するビューポートが移動し、図に示すように、スクロールインジケータが表示されます。ユーザが指を離すとインジケータは消えます。



概要

`UIScrollView`クラスは以下のような機能を提供します。

- 画面に全体が入りきらないコンテンツをスクロールする
- コンテンツを拡大縮小することで、アプリケーションが標準のピンチジェスチャに対応してズームイン（拡大）およびズームアウト（縮小）できるようにする

- コンテンツを一度に1画面ずつスクロール（ページングモード）するのを避ける

UIScrollViewクラスは、表示するコンテンツ用に特別に定義されたビューが含まれているわけではなく、単にサブビューをスクロールするだけです。iOS上のScroll Viewには、ほかにスクロールを開始するための制御はないため、こうした単純なモデルが可能です。

基本的なビュースクロールは実装が最も簡単

ドラッグジェスチャやフリックジェスチャによるスクロールを実現するのに、サブクラス化やデリゲートは必要ありません。UIScrollViewインスタンスのコンテンツサイズをプログラムで設定することを除き、インターフェイス全体をInterface Builderで作成して設計することができます。

関連する章： 「Scroll Viewの作成と設定」 （11 ページ）

ピンチジェスチャ、ズームジェスチャをサポートするにはデリゲートを使用する

基本的なピンチイン／ピンチアウトによる拡大縮小に対応するには、Scroll Viewでデリゲートを使う必要がありますデリゲートクラスはUIScrollViewDelegateプロトコルに準拠し、ズーム対象のScroll Viewのサブクラスを指定するデリゲートメソッドを実装する必要があります。また、拡大縮小倍率係数の最大値と最小値のどちらか、またはその両方を指定する必要があります。

アプリケーションで、（標準のピンチジェスチャに加えて）ダブルタップによる拡大縮小、2本の指のタッチによるズームアウト、および1本の指のタッチによるスクロールとパニングをサポートする必要がある場合、この機能を処理するためにコンテンツビューにコードを実装する必要があります。

関連する章： 「ピンチジェスチャを使った基本的なズーム」 （23 ページ）

ピンチしてズームおよびタップしてズームをサポートするには、コンテンツビューにコードを実装する

アプリケーションで、（標準のピンチジェスチャに加えて）ダブルタップによる拡大縮小、2本の指のタッチによるズームアウト、および1本の指のタッチによるスクロールとパニングをサポートする必要がある場合、コンテンツビューにコードを実装します。

関連する章： 「タップによるズーム」（29 ページ）

ページングモードをサポートするのに必要なのは3つのサブビューのみ

ページングモードをサポートするためには、サブクラス化もデリゲートも必要ありません。単純にコンテンツのサイズを指定してページングモードを有効にするだけです。サブビューを3つだけ使えばほとんどのページングアプリケーションを実装できます。その結果、メモリ空間を節約してパフォーマンスを向上させられます。

関連する注記： 「ページングモードを使用したスクロール」（33 ページ）

お読みになる前に

この文書を読む前に、『*iOS App Programming Guide*』を読んで、iOSアプリケーション開発の基本プロセスを理解しておいてください。Scroll Viewに関連してよく使用するView Controllerについての一般的な情報について、『*View Controller Programming Guide for iOS*』も読むことを検討してください。

この文書の使い方

このガイドの残りの章では、タップしてズームする手法、デリゲートの役割とメッセージシーケンスについての理解、アプリケーションでのScroll Viewのネストなど、複雑さの程度を少しずつ上げながら説明していきます。

関連項目

次のサンプルコードプロジェクトは、自分でTable Viewを実装する際の参考になります。

- *Scrolling*は基本的なスクロールを示します。
- *PageControl*はページングモードでのScroll Viewの使用法を示します。
- *ScrollViewSuite*はサンプルプロジェクトです。これには、タップしてスクロールする手法や、特に高度なその他のプロジェクト（大きい画像や細かな画像をメモリ効率の良い方法で表示するようにタイリングする方法など）を示す高度な例です。

Scroll Viewの作成と設定

Scroll Viewは、ほかのビューと同様にプログラムまたはInterface Builderで作成します。基本的なスクロール機能を実現するために必要な設定は、わずかな追加設定だけです。

Scroll Viewの作成

Scroll Viewは、ほかのビューと同じように作成してコントローラやビュー階層に挿入します。Scroll Viewの設定を完了するのに必要な追加のステップは2つだけです。

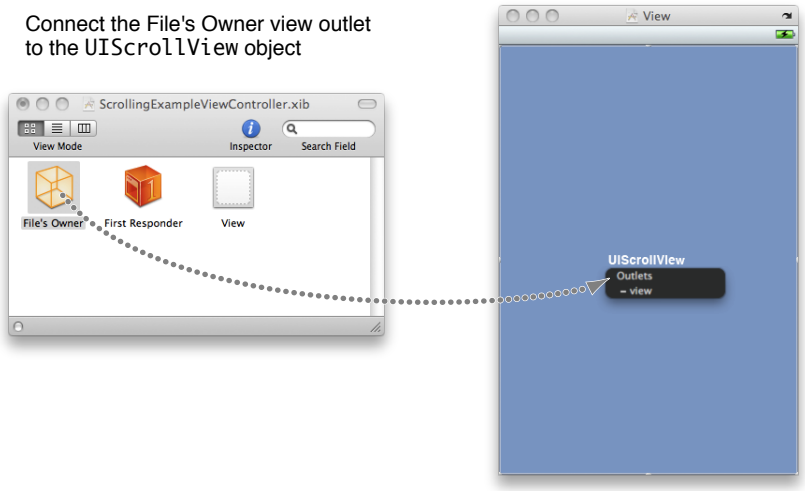
1. `contentSize`プロパティにスクロール可能なコンテンツのサイズを設定する必要があります。これにより、スクロール可能な領域のサイズが指定されます。
2. Scroll Viewで表示してスクロールする1つまたは複数のビューも追加する必要があります。これらのビューは表示するコンテンツを提供します。

必要に応じて、アプリケーションで必要な視覚的なヒント（水平スクロールインジケータや垂直スクロールインジケータ、ドラッグバウンス、ズームバウンス、スクロールの向きに対する制約など）を設定できます。

Interface BuilderでのScroll Viewの作成

Interface BuilderでScroll Viewを作成するには、Libraryパレットの**Library->Cocoa Touch->Data Views**セクションにある`UIScrollView`アイコンをView“ウインドウ”にドラッグします。その後、`UIViewController`サブクラスのViewアウトレットをScroll Viewに接続します。File's Ownerが`UIViewController`サブクラスであると想定（一般的なデザインパターンです）した接続を、図 1-1に示します。

図 1-1 Scroll ViewへのUIViewControllerサブクラスの接続方法



Interface BuilderのUIScrollViewインスペクタを使ってScroll Viewインスタンスの多くのプロパティを設定できますが、アプリケーションコードの中で、スクロール可能な領域のサイズを定義するcontentSizeプロパティを設定する責任は常にデベロッパにあります。Controllerインスタンス（通常はFile's Owner）のviewプロパティにScroll Viewを接続すると、リスト 1-1に示すControllerのviewDidLoadメソッドでcontentSizeプロパティの初期化が行われます。

リスト 1-1 Scroll Viewのサイズの設定

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIScrollView *tempScrollView=(UIScrollView *)self.view;
    tempScrollView.contentSize=CGSizeMake(1280,960);
}
```

Scroll Viewのサイズを設定した後は、ビューコンテンツを提供する必要なサブビューを、プログラムによって、またはInterface BuilderでScroll Viewにそれらを挿入するなどして、アプリケーションに追加できます。

プログラムによるScroll Viewの作成

Scroll Viewを完全にコードで作成することも可能です。これは、Controllerクラスで、具体的にはloadViewメソッドの実装の中で、通常行います。リスト 1-2にサンプルの実装を示します。

リスト 1-2 プログラムによるScroll Viewの作成

```
- (void)loadView {
    CGRect fullScreenRect=[[UIScreen mainScreen] applicationFrame];
    scrollView=[[UIScrollView alloc] initWithFrame:fullScreenRect];
    scrollView.contentSize=CGSizeMake(320,758);

    // Scroll Viewへのその他の設定を行う
    // Scroll Viewのサブビューとしてビュー（1つまたは複数）を追加する

    // self.viewが保持するためscrollViewを解放する
```

```
    self.view=scrollView;
    [scrollView release];
}
```

このコードでは、フルスクリーン（ステータスバーを除く）サイズのScroll Viewを作成して、scrollViewオブジェクトをControllerのビューとして設定し、contentSizeプロパティを320×758ピクセルに設定しています。このコードは垂直にスクロールするScroll Viewを作成します。

このメソッド実装には、たとえば、必要に応じてサブビューやビューを挿入してそれらを設定するなど、ほかのコードも考えられます。また、このコードでは、Controllerにviewが設定されていないものと仮定しています。すでに設定されている場合、Scroll ViewをControllerのビューとして設定する前に既存のviewを解放するのはデベロッパの責任です。

サブビューの追加

Scroll Viewを作成して設定したら、コンテンツを表示する1つまたは複数のサブビューを追加する必要があります。Scroll Viewの中で直接使用するサブビューを1つにすべきか複数にすべきかは、一般的にはScroll Viewでズームをサポートする必要があるかどうかによって決まる設計上の判断事項です。

Scroll Viewでズームをサポートする場合、最も一般的な手法はScroll ViewのcontentSize全体を含むサブビューを1つ使用し、そのビューに追加のサブビューを追加することです。こうすると、1つ用意した"コレクション"コンテンツビューをズーム用のビューとして指定することができ、そのサブビューすべてはコレクションコンテンツビューの状態に応じてズームするようになります。

ズームが必須要件でない場合は、Scroll Viewで1つのサブビュー（それ自身がサブビューを持つかどうかは問わない）を使用するか複数のサブビューを使用するかは、アプリケーションごとに決める判断事項となります。

注： 1つのサブビュー（複数のサブビューを持つ場合も含む）をズーム用ビューとして返すのが最も一般的ですが、アプリケーションで、ズームをサポートするために同じScroll View内に複数のビューを持てるようにする必要があるとすることもできます。この場合、デリゲートメソッドviewForZoomingInScrollView:を使用して適切なサブビューを返すことになります。詳細については「ピンチジェスチャを使用した基本的なズーム」（23 ページ）で説明します。

Scroll Viewのコンテンツサイズ、コンテンツのインセット、スクロールインジケータの設定

contentSizeプロパティは、Scroll Viewに表示する必要があるコンテンツのサイズです。「Interface BuilderでのScroll Viewの作成」では、これを幅320ピクセル、高さ758ピクセルに設定しています。図1-2の図に、Scroll ViewのコンテンツとcontentSizeの幅と高さを示します。

図 1-2 コンテンツとcontentSizeの寸法ラベル

コントローラとツールバーが、Scroll Viewのコンテンツ全体の表示の妨げにならないように、Scroll Viewコンテンツのそれぞれの縁（通常はコンテンツの最上部と最下部）の周辺に余白を追加できます。アプリケーションに余白を追加するにはScroll ViewのcontentInsetプロパティを設定する必要があります。contentInsetプロパティはScroll Viewのコンテンツの周辺に余白領域を指定します。これを見る1つの見方は、これにより、サブビューのサイズやビューのコンテンツサイズが変更されずにScroll Viewのコンテンツ領域が広くなるということです。

contentInsetプロパティは、top、bottom、left、rightの各フィールドを持ったUIEdgeInsets構造体です。図 1-3にcontentInsetとcontentSizeが指定されたコンテンツを示します。

図 1-3 contentSizeおよびcontentInsetが指定されたコンテンツ

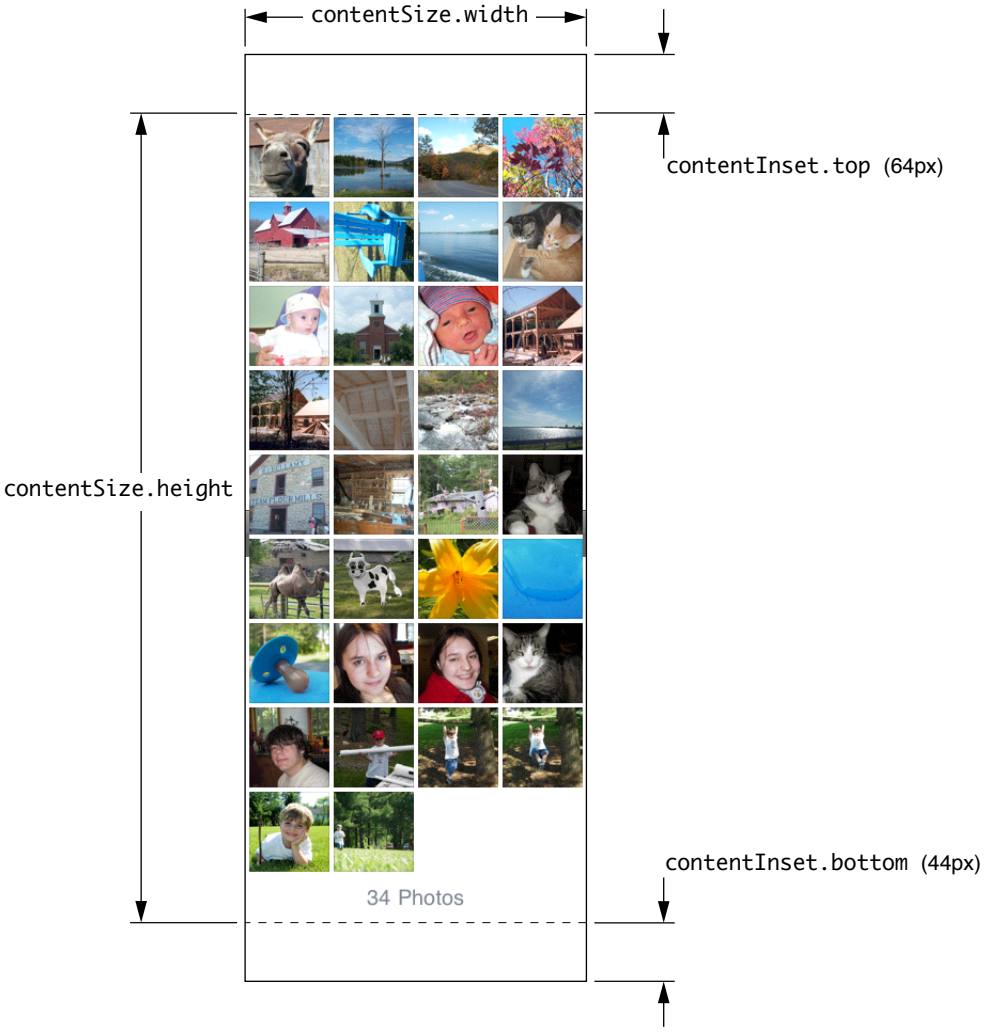


図 1-3 （15 ページ）に示すように、contentInsetプロパティに(64,44,0,0)を指定した結果、コンテンツの最上部に64ピクセル（ステータスバーに対し20ピクセルとNavigation Controllerに対し44ピクセル）と、最下部に44ピクセル（ツールバーの高さ）の余白領域が追加されます。contentInsetをこれらの値に設定することで、画面上にナビゲーションコントロールとツールバーを表示したまま、Scroll Viewのコンテンツ全体をスクロールして表示できます。

リスト 1-3 contentInsetプロパティの設定

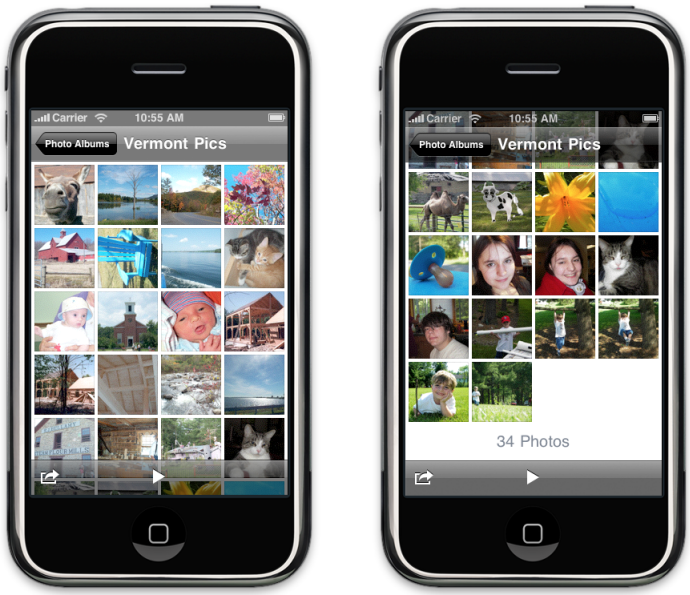
```
- (void)loadView {
    CGRect fullScreenRect=[[UIScreen mainScreen] applicationFrame];
    scrollView=[[UIScrollView alloc] initWithFrame:fullScreenRect];
    self.view=scrollView;
    scrollView.contentSize=CGSizeMake(320,758);
    scrollView.contentInset=UIEdgeInsetsMake(64.0,0.0,44.0,0.0);

    // Scroll Viewへのその他の設定を行う
    // Scroll Viewのサブビューとしてビュー（1つまたは複数）を追加する
```

```
    // self.viewが保持するためscrollViewを解放する
    self.view=scrollView;
    [scrollView release];
}
```

図 1-4にcontentInsetのtopパラメータとbottomパラメータをこれらの値に設定した結果を示します。最上部までスクロールすると（左側の図）、Navigation Barおよびステータスバーのために空間が残ります。右側の図は、最下部までコンテンツをスクロールしたとき、ツールバー用の空間が残っている様子を示します。どちらの場合もスクロールされた時にNavigation Barおよびツールバーを透かしてコンテンツが見えます。コンテンツが最上部または最下部へいっぱいにスクロールされた場合も、コンテンツの全部が見えます。

図 1-4 contentInsetのtopとbottomへの値の設定結果



Scrolled to top　　Scrolled to bottom

ただし、contentInsetの値を変更すると、Scroll Viewがスクロールインジケータを表示する場合に予期せぬ副次的な影響があります。ユーザが画面の最上部または最下部へコンテンツをドラッグすると、スクロールインジケータが、contentInsetで定義された領域内の領域に表示されているすべてのコンテンツ（たとえば、ナビゲーションコントロールやツールバーなど）に重ねてスクロールしてしまいます。

これを修正するには、scrollIndicatorInsetsプロパティを設定する必要があります。contentInsetプロパティと同様に、scrollIndicatorInsetsプロパティはUIEdgeInsets構造体として定義されています。垂直インセット値を設定すると、垂直スクロールのインジケータがそのインセット値を超えて表示されるのを制限します。またこれにより、水平スクロールのインジケータがcontentInsetの矩形領域の外側に表示されるようになります。

scrollIndicatorInsetsプロパティも設定せずにcontentInsetを変更すると、スクロールインジケータがNavigation Controllerおよびツールバーに重ねて描画され、望まない結果となります。しかし、scrollIndicatorInsetsの各値をcontentInsetの値に一致するように設定すると、この状況は解消されます。

リスト 1-4に示すloadViewの修正版の実装は、Scroll Viewの設定に必要な追加のコード（scrollIndicatorInsetsの初期化の追加）を示します。

リスト 1-4 Scroll ViewのcontentInsetプロパティおよびscrollIndicatorInsetsプロパティの設定

```
- (void)loadView {
    CGRect fullScreenRect=[[UIScreen mainScreen] applicationFrame];
    scrollView=[[UIScrollView alloc] initWithFrame:fullScreenRect];
    scrollView.contentSize=CGSizeMake(320,758);
    scrollView.contentInset=UIEdgeInsetsMake(64.0,0.0,44.0,0.0);
    scrollView.scrollIndicatorInsets=UIEdgeInsetsMake(64.0,0.0,44.0,0.0);

    // Scroll Viewへのその他の設定を行う
    // Scroll Viewのサブビューとしてビュー（1つまたは複数）を追加する

    // self.viewが保持するためscrollViewを解放する
    self.view=scrollView;
    [scrollView release];
}
```

Scroll Viewコンテンツのスクロール

Scroll Viewのスクロールを開始する最も一般的な方法は、ユーザが画面に触れて指でドラッグすることによる直接操作です。スクロールコンテンツはそのアクションに応えてスクロールします。このジェスチャのことを*ドラッグジェスチャ*と言います。

ドラッグジェスチャのバリエーションには*フリックジェスチャ*があります。フリックジェスチャは、最初に画面に触れ、スクロールしたい方向へドラッグして画面から指を離す、ユーザの指の素早い動きです。このジェスチャによってスクロールが行われるだけでなく、ユーザのドラッグアクションの速度に応じた勢いが生じて、ジェスチャが完了した後もスクロールが継続します。その後、スクロールは所定の時間の経過とともに減速します。フリックジェスチャを使うと、ユーザは一度のアクションで大きく移動することができるようになります。減速している間、ユーザはいつでもスクロールしている画面を適当な位置でタッチして止めることができます。この動作はすべて`UIScrollView`に組み込まれており、デベロッパ側での実装は必要ありません。

しかし、たとえばドキュメントの特定の場所を表示する場合など、アプリケーションでコンテンツをスクロールさせるためのプログラミングが必要なことがあります。このような場合、`UIScrollView`が必要なメソッドを提供します。

`UIScrollView`のデリゲートプロトコルである`UIScrollViewDelegate`は、アプリケーションでスクロールの進行を追跡し、アプリケーションの特定のニーズに応じて応答できるようにするメソッドをいくつか提供します。

プログラムによるスクロール

Scroll Viewのコンテンツのスクロールは、必ずしもすべてがユーザの指のドラッグや画面のフリックに対応した動作ではありません。時には、アプリケーションで特定の矩形領域を表示するためにコンテンツの特定のオフセットまでスクロールしたり、Scroll Viewの最上部へスクロールしたりする必要が生じる場合もあります。`UIScrollView`はこのようなアクションすべてを実行するためのメソッドを提供します。

特定のオフセットへのスクロール

特定の位置を左上角とする位置（`contentOffset`プロパティ）へのスクロールは、2通りの方法で実行できます。`setContentOffset:animated:`メソッドは、指定されたコンテンツオフセットへコンテンツをスクロールします。アニメーション用のパラメータが`YES`の場合、現在の位置から指定の位置まで一定の速度で進むようにスクロールがアニメーション化されます。アニメーション用パラメータが`NO`の場合、スクロールは即座に行われ、アニメーション化は行われません。どちらの場合もデリゲートには、`scrollViewDidScroll:`メッセージが送られます。アニメーションが無効の場合、または直接`contentOffset`プロパティに設定することでコンテンツオフセットを設定した場合、デリゲートは`scrollViewDidScroll:`メッセージを1つ受信します。アニメーションが有効の場合、デリゲートはアニメーションが進行中の間、一連の`scrollViewDidScroll:`メッセージを受信します。アニメーションが完了すると、デリゲートは`scrollViewDidEndScrollingAnimation:`メッセージを受信します。

矩形範囲の表示

ある矩形領域が表示されるようにスクロールすることも可能です。これは特に、アプリケーションで、現在可視領域の外にあるコントロールを、表示されているビュー内に移動して表示する必要がある場合に有用です。`scrollRectToVisible:animated:`メソッドは、Scroll View内でちょうど目にすることができるように指定の矩形領域をスクロールします。アニメーション用パラメータが`YES`の場合、矩形領域はビュー内に表示されるまで一定の速度でスクロールされます。`setContentOffset:animated:`と同様に、アニメーションが無効の場合は、デリゲートには`scrollViewDidScroll:`メッセージが1つ送信されます。アニメーションが有効の場合、デリゲートにはアニメーションが進行中の間、一連の`scrollViewDidScroll:`メッセージが送られます。`scrollRectToVisible:animated:`の場合、Scroll Viewの追跡プロパティとドラッグプロパティはどちらも`NO`です。

アニメーションが`scrollRectToVisible:animated:`で有効の場合、デリゲートは、Scroll Viewが指定の位置にたどり着き、アニメーションが完了した通知を提供する`scrollViewDidEndScrollingAnimation:`メッセージを受信します。

最上部へのスクロール

ステータスバーが表示されている場合、ステータスバーへのタップに応答してScroll Viewもコンテンツの最上部へスクロールします。この方法は、データを垂直方向に示すアプリケーションでは一般的です。たとえば、「写真(Photos)」アプリケーションは、アルバム選択のTable Viewおよびアルバム写真のサムネイル閲覧時の両方で最上部までのスクロールをサポートしています。また、`UITableView`（`UIScrollView`のサブクラス）のほとんどの実装は、最上部へのスクロールをサポートしています。

アプリケーションでは、Scroll Viewのプロパティを設定するデリゲートメソッド`scrollViewShouldScrollToTop:`を実装して、`YES`を返すことでこの動作を有効にします。このデリゲートメソッドは、画面上に同時に複数のScroll Viewが存在する場合に、スクロールさせるScroll Viewを返すことで、最上部にスクロールするScroll Viewを細かく制御することを可能にします。

スクロールが完了すると、デリゲートは、Scroll Viewを指定する`scrollViewDidScrollToTop:`メッセージを送られます。

スクロール中に送られるデリゲートメッセージ

スクロールが発生すると、Scroll Viewは`tracking`、`dragging`、`decelerating`、および`zooming`の各プロパティを使用して状態を追跡します。また、`contentOffset`プロパティは、Scroll Viewの境界の左上角に表示されているコンテンツ内の点を定義します。それぞれの状態プロパティについて次の表で説明します。

状態プロパティ	説明
`tracking`	ユーザの指がデバイス画面と接触している場合、`YES`
`dragging`	ユーザの指がデバイス画面と接触していて動いた場合、`YES`

状態プロパティ	説明
`decelerating`	フリックジェスチャ、またはScroll Viewのフレームを超えてドラッグしてバウンスした結果、Scroll Viewが減速している場合、`YES`
`zooming`	`zoomScale`プロパティを変更するためにScroll Viewがピンチジェスチャを追跡している場合、`YES`
`contentOffset`	Scroll Viewの境界の左上角を定義する`CGPoint`の値

実行中のアクションを判断するためにこれらのプロパティをポーリングする必要はありません。Scroll Viewは詳細なメッセージシーケンスをデリゲートに送信し、スクロールアクションの進行状況を伝えるからです。これらのメソッドによりアプリケーションは状況に応じた応答が可能になります。デリゲートメソッドでは、これらの状態プロパティを照会して、メッセージを受信した理由やScroll Viewの現在の位置を判断できます。

簡単なアプローチ：スクロールアクションの開始と完了の追跡

アプリケーションで必要な情報がスクロール処理の開始と終わりについてのみであれば、実装可能なデリゲートメソッドの小さなサブセットのみを実装することで済ませることができます。

ドラッグ開始の通知を受け取るには、`scrollViewWillBeginDragging:`メソッドを実装します。

スクロールの完了を知るには、`scrollViewDidEndDragging:willDecelerate:`と`scrollViewDidEndDecelerating:`の2つのデリゲートメソッドを実装する必要があります。スクロールが完了するのは、デリゲートが、減速パラメータが`NO`である`scrollViewDidEndDragging:willDecelerate:`メッセージと受け取ったか、`scrollViewDidEndDecelerating:`メソッドを受け取ったときです。どちらの場合もスクロールは完了しています。

デリゲートメッセージシーケンスの全体

ユーザが画面をタッチすると追跡シーケンスが開始されます。`tracking`プロパティがすぐに`YES`に設定され、ユーザの指が動いているかどうかに関わらず画面に接触している間は`YES`のままです。

ユーザの指が静止しており、コンテンツビューがタッチイベントに応答するものである場合、ビューはタッチを処理すべきであり、シーケンスは完了しています。

ただし、ユーザが指を動かせばシーケンスは継続します。

ユーザが指を動かしてスクロールを開始しようとすると、Scroll Viewは、まず進行中のタッチ処理があれば、それらをすべて取り消すことを試みます（Scroll Viewがデフォルト値で設定されている場合）。

注： メッセージシーケンスを通して、trackingプロパティとdraggingプロパティが常にNOのままになり、zoomingプロパティがYESである可能性があります。これは、スクロールがジェスチャによって開始されたかプログラムにって開始されたかに関わらず、ズームアクションの結果としてスクロールが発生した場合に起こります。ズームまたはスクロールの結果としてデリゲートメソッドが送られた場合、アプリケーションでは異なるアクションを取ることも可能です。

Scroll Viewのdraggingプロパティは`YES`に設定され、そのデリゲートには`scrollViewWillBeginDragging:`メッセージが送信されます。

ユーザが指をドラッグすると`scrollViewDidScroll:`メッセージがデリゲートに送られます。このメッセージは、スクロールが続いている間、継続的に送られます。このメソッドの実装でScroll Viewの`contentOffset`プロパティを照会し、Scroll Viewの境界の左上角の位置を知ることができます。`contentOffset`プロパティは、スクロールが進行中であるかどうかに関わらず、常にスクロール境界の左上角の現在位置を示します。

ユーザがフリックジェスチャを行うと、trackingプロパティは`NO`に設定されます。これは、フリックジェスチャを行うために、最初のジェスチャによってスクロールが開始された後、ユーザの指が画面から離れるからです。この時デリゲートは`scrollViewDidEndDragging:willDecelerate:`メッセージを受信します。減速パラメータは、スクロールが減速すると`YES`になります。減速スピードは`decelerationRate`プロパティで制御します。デフォルトでは、このプロパティは`UIScrollViewDecelerationRateNormal`に設定されており、スクロールはかなり長い時間続きます。速度を`UIScrollViewDecelerationFast`に設定すれば、減速の時間を大幅に短縮して、フリックジェスチャ後のスクロールする距離をだいぶ短くできます。ビューが減速すると、Scroll Viewの`decelerating`プロパティは`YES`になります。

ユーザがドラッグし、ドラッグを止めて指を画面から離すと、デリゲートは`scrollViewDidEndDragging:willDecelerate:`メッセージを受信します（ただし、減速パラメータは`NO`です）。これは、Scroll Viewに勢いが加えられなかったためです。ユーザの指はすでに画面と接触しているわけではないため、trackingプロパティの値は`NO`になります。

`scrollViewDidEndDragging:willDecelerate:`メッセージの減速パラメータが`NO`の場合、Scroll Viewのデリゲートはこのドラッグアクションに対するデリゲートメッセージをもう受信しません。Scroll Viewの減速プロパティも以降は`NO`の値を返します。

静止している状態でユーザが指を離しても、デリゲートに`scrollViewDidEndDragging:willDecelerate:`メッセージが送られる状態はもう1つあります。ユーザがスクロール領域の縁を超えてコンテンツをドラッグしたときに視覚的にバウンスする（跳ね返る）様子を表示するようにScroll Viewが設定されている場合、減速パラメータが`YES`に設定された`scrollViewDidEndDragging:willDecelerate:`メッセージがデリゲートに送られます。バウンスは、`bounces`プロパティが`YES`の場合（デフォルト設定）に有効です。`alwaysBounceVertical`プロパティと`alwaysBounceHorizontal`プロパティは、`bounces`が`NO`の場合、Scroll Viewの動作に影響しません。`bounces`が`YES`の場合、`contentSize`プロパティの値がScroll Viewの境界より小さいときはバウンスが可能です。

Scroll Viewが`scrollViewDidEndDragging:willDecelerate:`メッセージを受信する原因となった状況に関わらず、減速パラメータが`YES`であれば、Scroll Viewには`scrollViewWillBeginDecelerating:`が送られます。trackingの値とdraggingの値がどちらも`NO`であっても、減速している間はデリゲートは`scrollViewDidScroll:`メッセージを受信し続けます。deceleratingプロパティは引き続き`YES`です。

最終的に、Scroll Viewの減速が完了すると、デリゲートには`scrollViewDidEndDecelerating:`メッセージが送られます。deceleratingプロパティの値は`NO`となり、スクロールシーケンスは完了します。

ピンチジェスチャを使った基本的なズーム

UIScrollViewを使って、ピンチジェスチャによるズームを簡単にサポートすることができます。アプリケーションでは拡大縮小率（コンテンツをどの程度拡大縮小できるかを表す）を指定し、デリゲートメソッドを1つ実装します。わずかにこれだけの手順を踏むことで、Scroll Viewでピンチジェスチャによるズームをサポートすることができます。

ピンチズームジェスチャのサポート

ピンチイン／ピンチアウトズームジェスチャは、iOSアプリケーションユーザが表示を拡大縮小するときに行うものと想定している標準のジェスチャです。図 3-1に、ピンチジェスチャの例を示します。

図 3-1 標準的なピンチイン／ピンチアウトジェスチャ

ズームをサポートするには、Scroll Viewに対してデリゲートを設定する必要があります。このデリゲートオブジェクトは、UIScrollViewDelegateプロトコルに従わなければなりません。多くの場合、デリゲートはScroll Viewのコントローラクラスです。そのデリゲートクラスは viewForZoomingInScrollView:メソッドを実装し、ズーム対象のビューを返す必要があります。以下に示すデリゲートメソッドの実装は、コントローラのimageViewプロパティの値を返します。これはUIImageViewのインスタントです。これにより、ズームジェスチャや何らかのプログラミングによるズームに応じてimageViewプロパティがズームされることが指定されます。

```
- (UIView *)viewForZoomingInScrollView:(UIScrollView *)scrollView
{
    return self.imageView;
}
```

ユーザがズームできる量を指定するには、minimumZoomScaleプロパティとmaximumZoomScaleプロパティの値を設定します。この2つのプロパティは、初期設定では1.0に設定されています。プロパティの値はInterface BuilderのUIScrollViewインスペクタパネルか、またはプログラミングによって設定できます。リスト 3-1に、ズームをサポートするためにUIViewControllerのサブクラスで必要な

コードを示します。このコードでは、コントローラサブクラスのインスタンスはデリゲートであり、上記のviewForZoomingInScrollView:デリゲートメソッドを実装しているものと想定しています。

リスト 3-1　UIViewControllerサブクラスにおける最小限必要なズームメソッドの実装

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.scrollView.minimumZoomScale=0.5;
    self.scrollView.maximumZoomScale=6.0;
    self.scrollView.contentSize=CGSizeMake(1280, 960);
    self.scrollView.delegate=self;
}
```

ズーム倍率の指定と、viewForZoomingInScrollView:メソッドを実装するデリゲートオブジェクトの指定は、ピンチジェスチャを使ったズームのサポートに最低限必要な要件です。

プログラミングによるズーム

Scroll Viewは、ダブルタップやその他のタップジェスチャなどのタッチイベントに反応して、あるいは、ピンチジェスチャ以外のユーザアクションに反応して拡大縮小しなければならないことがあります。そのために、UIScrollViewは、setZoomScale:animated:とzoomToRect:animated:という2つのメソッドの実装を用意しています。

setZoomScale:animated:は、現在の拡大縮小率を指定された値に設定します。値は、minimumZoomScaleとmaximumZoomScaleで指定された範囲内に入っていなければなりません。アニメーションパラメータがYESの場合、ズーム操作により一定のアニメーションがズーム完了まで実行されます。それ以外の場合、拡大縮小率の変更が即座に実行されます。zoomScaleプロパティを直接設定することもできます。これは、アニメーションパラメータとしてNOを指定してsetZoomScale:animated:を呼び出すことと同じです。このメソッドで、またはプロパティを直接変更してズーム操作を行うと、ビューはその中心を固定した状態で拡大縮小されます。

zoomToRect:animated:メソッドは、指定された矩形内に納まるようにコンテンツを拡大縮小します。setZoomScale:animated:の場合と同様に、このメソッドには、位置と拡大縮小率の変化によってアニメーションが実行されるかどうかを決めるアニメーションパラメータがあります。

アプリケーションでは、特定の位置でのタップに反応して拡大縮小率と位置を設定しなければならないこともよくあります。setZoomScale:animated:は、表示されているコンテンツの中心の周りをズームするため、特定の位置と拡大縮小率を受け取って、それをzoomToRect:animated:に適した矩形に変換する関数が必要になります。リスト 3-2に、Scroll View、拡大縮小率、ズーム矩形の中心点を受け取るユーティリティメソッドを示します。

リスト 3-2　指定された拡大縮小率とズームする矩形の中心点を変換するユーティリティメソッド

```
- (CGRect)zoomRectForScrollView:(UIScrollView *)scrollView withScale:(float)scale
 withCenter:(CGPoint)center {

    CGRect zoomRect;

    // ズーム矩形はコンテンツビューの座標系で表される。
    // 拡大縮小率1.0の場合、ズーム矩形のサイズは
    // imageScrollViewの境界矩形と同じ。
```

```
    // 拡大縮小率が小さくなるほど、より多くのコンテンツが表示され、
    // 矩形のサイズは大きくなる。
    zoomRect.size.height = scrollView.frame.size.height / scale;
    zoomRect.size.width  = scrollView.frame.size.width  / scale;

    // 中心となるように原点を選択する
    zoomRect.origin.x = center.x - (zoomRect.size.width  / 2.0);
    zoomRect.origin.y = center.y - (zoomRect.size.height / 2.0);

    return zoomRect;
}
}
```

このユーティリティメソッドは、ダブルタップのジェスチャをサポートするカスタムサブクラスでのダブルタップに反応する際に役立ちます。このメソッドを使用するには、対象となるUIScrollViewインスタンス、新しい拡大縮小率（たいていは、ズームする量を加算または乗算することによって既存の`zoomScale`から導き出される）、およびズームの中心とする点を単に渡すだけです。ダブルタップジェスチャに反応するときには、通常はタップの位置が中心点になります。メソッドから返される矩形は、`zoomToRect:animated:`メソッドへの受け渡しに適しています。

デリゲートへのズーム完了の通知

ユーザがズームピンチジェスチャを終えたり、プログラミングによるScroll Viewのズームが完了したりすると、`UIScrollView`デリゲートには`scrollViewDidEndZooming:withView:atScale:`メッセージによって通知されます。

このメソッドには、Scroll Viewインスタンス、スクロールされたScroll Viewサブビュー、ズーム完了時の拡大縮小率がパラメータとして渡されます。このデリゲートメッセージを受信すると、アプリケーションは適切なアクションをとることができます。

ズーム後のコンテンツを鮮明に保つ

Scroll Viewのコンテンツが拡大縮小されると、ズームビューのコンテンツは、拡大縮小率の変化に応じて単に拡大縮小されます。これにより、よりコンテンツが拡大または縮小されますが、再描画はされません。そのため、表示されているコンテンツは鮮明でないことがあります。ズーム対象のコンテンツが画像で、アプリケーションが新たにより詳細なコンテンツを表示しないのであれば（「マップ(Maps)」アプリケーションなど）、これは問題にはなりません。

しかし、ズームに反応してより詳細なビットマップ画像を表示する必要があるアプリケーションの場合は、『*ScrollViewSuite*』のサンプルコードの「Tiling」の例が参考になります。このサンプルコードでは、ズーム対象のコンテンツを小さな単位で事前レンダリングする手法を使い、Scroll View内の別々のビューにコンテンツを表示します。

しかし、ズームしたコンテンツを実時間で描画し、ズームした際にも明瞭に表示しようとすると、ズームしたビューを描画するアプリケーションクラスは、Core Animationを使う必要があります。クラスは、`UIView`クラスのレイヤとして使われているCore Animationクラスを、`CATiledLayer`に変更し、Core Animationの`drawLayer:inContext:`メソッドで描画する必要があります。

リスト 3-3 （26 ページ）に、十字を描画し、ズームに対応するサブクラスの完全な実装を示します。ズーム操作の間も画像は鮮明なままです。Zoomableビューは、コンテンツサイズ（460,320）のScroll Viewのサブビューとして追加されたUIViewサブクラスで、Scroll ViewデリゲートメソッドのviewForZoomingInScrollView:によって返されます。ズーム発生時にZoomableビューを再描画するために、デベロッパ側で必要なアクションはありません。

リスト 3-3　ズーム時にコンテンツを鮮明に描画するUIViewサブクラスの実装

```
#import "ZoomableView.h"
#import <QuartzCore/QuartzCore.h>

@implementation ZoomableView


// UIViewレイヤをCATiledLayerに設定
+(Class)layerClass
{
    return [CATiledLayer class];
}


// バイアスのlevelsOfDetailBiasと
// タイル画像を描画するレイヤのlevelsOfDetailを設定して
// レイヤを初期化
-(id)initWithFrame:(CGRect)r
{
    self = [super initWithFrame:r];
    if(self) {
        CATiledLayer *tempTiledLayer = (CATiledLayer*)self.layer;
        tempTiledLayer.levelsOfDetail = 5;
        tempTiledLayer.levelsOfDetailBias = 2;
        self.opaque=YES;
    }
    return self;
}

// UIViewクラスが正しく動作するように-drawRect:を実装
// 実際の描画処理は-drawLayer:inContextで行われる
-(void)drawRect:(CGRect)r
{
}

-(void)drawLayer:(CALayer*)layer inContext:(CGContextRef)context
{
    // コンテキストは適切に拡大縮小と変換が行われ、あたかもレイヤ全体に描画しているかのように
    // このコンテキストへの描画ができ、正しいコンテンツが再描画される。
    // ここでは現在のCTMは、回転なしで均一に拡大縮小されるアフィン変換であると想定。

   // これは、暗黙的に以下を意味する。
    // a == d and b == c == 0
    // CGFloat scale = CGContextGetCTM(context).a;
    // ここでは使われていないが、ほかの状況で役立つ場合がある。

    // クリップ境界矩形は、レンダリングが要求されているコンテキストの
    // 領域を示す。ここでは使われていないが、
    // アプリケーションでは、ほかの状況で拡大縮小のために
    // これが必要になることがある。
```

```
    // CGRect rect = CGContextGetClipBoundingBox(context);

    // レイヤ全体の背景色の設定と描画
    // あるいは、レイヤをopaque=NO;と設定してもよい。
    // その場合はこの後の2行のコードを削除し
    // Scroll Viewの背景色を設定する。
    CGContextSetRGBFillColor(context, 1.0,1.0,1.0,1.0);
    CGContextFillRect(context,self.bounds);

    // 簡単なプラス記号を描画
    CGContextSetRGBStrokeColor(context, 0.0, 0.0, 1.0, 1.0);
    CGContextBeginPath(context);
    CGContextMoveToPoint(context,35,255);
    CGContextAddLineToPoint(context,35,205);
    CGContextAddLineToPoint(context,135,205);
    CGContextAddLineToPoint(context,135,105);
    CGContextAddLineToPoint(context,185,105);
    CGContextAddLineToPoint(context,185,205);
    CGContextAddLineToPoint(context,285,205);
    CGContextAddLineToPoint(context,285,255);
    CGContextAddLineToPoint(context,185,255);
    CGContextAddLineToPoint(context,185,355);
    CGContextAddLineToPoint(context,135,355);
    CGContextAddLineToPoint(context,135,255);
    CGContextAddLineToPoint(context,35,255);
    CGContextClosePath(context);

    // 単純な図形をストローク
    CGContextStrokePath(context);


}
```

警告： 注意：この描画メソッドには重大な制限事項があります。UIKitの描画メソッドはスレッドセーフではなく、`drawLayer:inRect:`コールバックはバックグラウンドスレッドで呼び出されます。このため、UIKitの描画メソッドではなくCoreGraphicsの描画関数を使用する必要があります。

タップによるズーム

基本クラスである`UIScrollView`クラスは、ごく少量のコードを使ってピンチインおよびピンチアウトジェスチャをサポートしていますが、タップ検出を使ってより充実したズーム体験に対応するには、アプリケーション側で多くの作業が必要です。

『*iOS Human Interface Guidelines*』に、ズームイン/ズームアウトするためのダブルタップ操作が定義されています。ただし、ガイドラインではいくつか特定の制約を想定しています。「写真(Photos)」アプリケーションのようにビューのズームのレベルが1つであったり、連続したダブルタップで最大限までズームしたり、次のダブルタップに達すると全画面表示に戻ったりするなどです。しかし、アプリケーションによっては、たとえば「マップ(Maps)」アプリケーションのように、タップしてズームする機能に対応するときに、より柔軟な動作が必要になることもあります。「マップ(Maps)」ではダブルタップでズームインし、追加のダブルタップでさらにズームインします。連続的に一定の刻みでズームアウトするには、「マップ(Maps)」では、2本の指を狭い間隔に保ってタッチする方法によって、段階的にズームアウトします。このジェスチャは『*iOS Human Interface Guidelines*』では規定されていませんが、アプリケーションにこの機能が必要な場合には、「マップ（Maps）」を模倣して取り入れることができます。

アプリケーションでタップによるズームの機能をサポートするには、`UIScrollView`クラスをサブクラス化する必要はありません。代わりに、`UIScrollView`のデリゲートメソッド`viewForZoomingInScrollView:`から返されるクラスに、必要なタッチ処理を実装します。そのクラスが、画面上の指の数とタップ回数を追跡します。シングルタップ、ダブルタップ、2本指のタッチを検出すると、それに応じて適切に対応します。ダブルタップと2本指のタッチの組み合わせの場合、プログラムによって適切な倍率でScroll Viewをズームする必要があります。

タッチ処理コードの実装

`UIView`（またはこの子孫）のサブクラスのタッチコードでタップ、ダブルタップ、2本指でのタップをサポートするには、`touchesBegan:withEvent:`、`touchesEnded:withEvent:`、`touchesCanceled:withEvent:`の3つのメソッドを実装する必要があります。さらに、対話操作の初期化、複数のタッチ、追跡のための変数が必要になることもあります。次のコードは、`UIImageView`のサブクラスである`TapDetectingImageView`に含まれており、サンプルコードプロジェクト『*ScrollViewSuite*』の*TapToZoom*の例から抜粋したものです。

初期化

タップによるズームの実装に必要なジェスチャでは、対象のビューでユーザとの対話操作と複数タッチが有効であることと、この機能を有効にするメソッドが`initWithImage:`メソッドから呼び出されることが求められます。このメソッドはまた、タッチメソッドの中で状態を追跡する2つのインスタンス変数の初期化も行います。`twoFingerTapIsPossible`プロパティはブール値で、デバイスの画面に3本以上の指が触れていない限り`YES`となります。`multipleTouches`プロパティは、2回以上のタッチイベントが検出されない限り、値は`NO`です。3番目のプロパティの`tapLocation`は、ダブルタップの位置、または2本指のタッチが検出されたときの2本の指の中間点を追跡するための

CGPointです。この点はその後、「プログラミングによるズーム」（24 ページ）で説明しているように、プログラミングによるズームのメソッドを使ったズームインまたはズームアウトの中心点として使われます。

```
- (id)initWithImage:(UIImage *)image {
    self = [super initWithImage:image];
    if (self) {
        [self setUserInteractionEnabled:YES];
        [self setMultipleTouchEnabled:YES];
        twoFingerTapIsPossible = YES;
        multipleTouches = NO;
    }
    return self;
}
```

初期化が行われると、クラスはタッチイベントの受信に備えることができます。

touchesBegan:withEvent:の実装

`touchesBegan:withEvent:`メソッドはまず、1本指のタップ、すなわち`handleSingleTap`メッセージの処理を開始しようとするすべての試みを取り消します。メッセージを取り消すのは、それが送信されると、これが追加のタッチイベントとなるためシングルタップと認められずに無効だからです。これが最初のタッチであるためメッセージが送信されていなかった場合、実行の取り消しは無視されます。

メソッドは次に、追跡用変数の状態を更新します。2回以上のタッチイベントが受信された場合、これは2本指のタッチの可能性があるため`multipleTouches`プロパティは`YES`に設定されます。3回以上のタッチイベントが発生した場合、`twoFingerTapIsPossible`プロパティは`NO`に設定され、一度に3本以上の指によるタッチは無視されるジェスチャになります。

このメソッドのコードを以下に示します。

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // 保留中のhandleSingleTapメッセージをすべて取り消す
    [NSObject cancelPreviousPerformRequestsWithTarget:self
                                             selector:@selector(handleSingleTap)
                                               object:nil];

    // タッチの状態を更新
    if ([[event touchesForView:self] count] > 1)
        multipleTouches = YES;
    if ([[event touchesForView:self] count] > 2)
        twoFingerTapIsPossible = NO;

}
```

touchesEnded:withEvent:の実装

このメソッドはタップ処理の中心的な役割の部分で、多少複雑ですが、コードには十分な説明がなされており、以下にこれをそのまま示します。`midPointBetweenPoints`関数は、`handleTwoFingerTap`メソッドが呼び出されたときに2本指のタッチの中心点を特定するために使われ、その結果ビューは一段階ズームアウトします。

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    BOOL allTouchesEnded = ([touches count] == [[event touchesForView:self]
count]);

    // まず、単純なシングル／ダブルタップを確認する。複数回のタッチが行われていない場合のみ、
シングル／ダブルタップの可能性がある
    if (!multipleTouches) {
        UITouch *touch = [touches anyObject];
        tapLocation = [touch locationInView:self];

        if ([touch tapCount] == 1) {
            [self performSelector:@selector(handleSingleTap)
                       withObject:nil
                       afterDelay:DOUBLE_TAP_DELAY];
        } else if([touch tapCount] == 2) {
            [self handleDoubleTap];
        }
    }

    // 複数回のタッチが行われていた場合は、それが2本指のタップだったかどうか、
    // また、そうである状況が否定されていないか確認する
    else if (multipleTouches && twoFingerTapIsPossible) {

        // ケース1：これが両方のタッチの同時の終了の場合
        if ([touches count] == 2 && allTouchesEnded) {
            int i = 0;
            int tapCounts[2];
            CGPoint tapLocations[2];
            for (UITouch *touch in touches) {
                tapCounts[i] = [touch tapCount];
                tapLocations[i] = [touch locationInView:self];
                i++;
            }
            if (tapCounts[0] == 1 && tapCounts[1] == 1) {
                // 両方ともシングルタップであれば、これは2本指のタップ
                tapLocation = midpointBetweenPoints(tapLocations[0],
                                                    tapLocations[1]);
                [self handleTwoFingerTap];
            }
        }

        // ケース2：これが1回のタッチの終了で、もう1つはまだ終了していない場合
        else if ([touches count] == 1 && !allTouchesEnded) {
            UITouch *touch = [touches anyObject];
            if ([touch tapCount] == 1) {
                // タッチが1回のタップであれば、その位置を保存し、
                // 2番目のタッチの位置との平均値を求められるようにする
                tapLocation = [touch locationInView:self];
            } else {
                twoFingerTapIsPossible = NO;
            }
        }

        // ケース3：これが2つのタッチの2番目の終了である場合
        else if ([touches count] == 1 && allTouchesEnded) {
            UITouch *touch = [touches anyObject];
            if ([touch tapCount] == 1) {
```

```
                // 最後のタッチがシングルタップであれば、これは2本指のタップ
                tapLocation = midpointBetweenPoints(tapLocation,
                                                    [touch locationInView:self]);
                [self handleTwoFingerTap];
            }
        }
    }

    // すべてのタッチが終了した場合、タッチの監視状態をリセットする
    if (allTouchesEnded) {
        twoFingerTapIsPossible = YES;
        multipleTouches = NO;
    }
}
```

touchesCancelled:withEvent:の実装

指が移動してスクロールの状態になったために、タップ処理はもう関係しないことをScroll Viewビューが検出すると、ビューは`touchesCancelled:withEvent:`メッセージを受信します。このメソッドは単純に状態変数をリセットします。

```
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    twoFingerTapIsPossible = YES;
    multipleTouches = NO;
}
```

ScrollViewスイートの例

*ScrollViewSuite*サンプルコードプロジェクトは、タップジェスチャを使ったズームの実装の優れた例です。このスイートの`TapToZoom`の例は、「マップ(Maps)」アプリケーションで見られるようなタップ動作を使ったズームをサポートする`UIImageView`のサブクラスを実装します。サンプルコードではタップ、ダブルタップ、2本指のタッチの実際の処理を実装するためにデリゲート（通常はScroll Viewを管理するコントローラ）を使っていますが、実装は十分に汎用性があるため、コードを簡単に適用してデベロッパ独自のビューを設計できます。

`TapDetectingImageView`クラスは、タッチ処理を実装する`UIImageView`のサブクラスです。タップとタッチに対する実際の応答を処理するデリゲートとして`RootViewController`クラスと、`UIScrollView`の初期状態を設定するコントローラを使います。

ページングモードを使用したスクロール

UIScrollViewクラスは、ユーザが開始したスクロールアクションを、一度に1画面分のコンテンツをスクロールするように制限するページングモードをサポートします。このモードは、たとえばeBookや一連の手順など、逐次的なコンテンツを表示する際に使います。

ページングモードの設定

ページングモードをサポートするためにScroll Viewを設定するには、Scroll ViewのControllerクラスにコードを実装する必要があります。

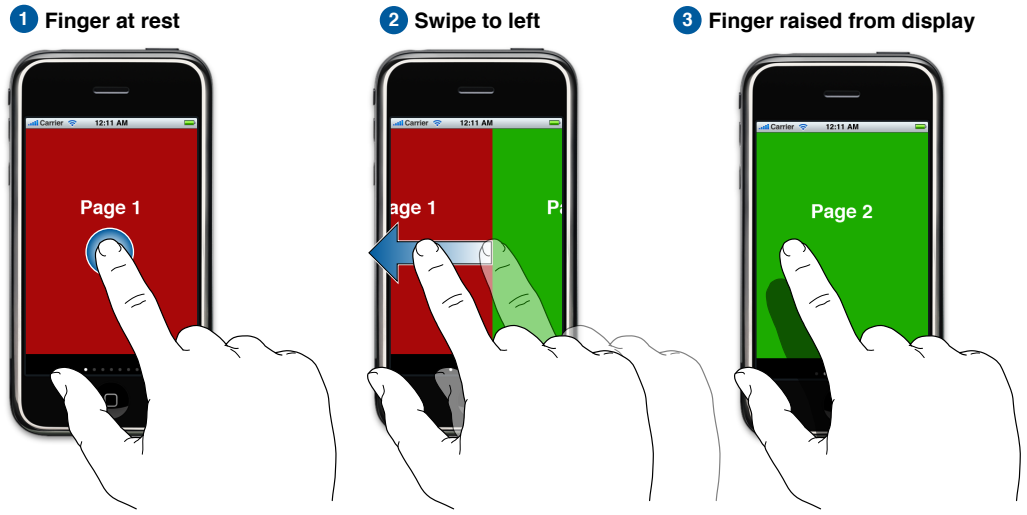
「Scroll Viewの作成と設定」（11 ページ）で説明した、標準のScroll Viewの初期化のほかに、pagingModeプロパティをYESに設定する必要もあります。

ページングモードのScroll ViewのcontentSizeプロパティは、画面の高さいっぱいになるように、幅は表示するページの数にデバイス画面幅を乗じた倍数になるように設定します。

また、画面上でユーザがタッチしている相対位置は重要ではないため、あるいはUIPageControlを使って表示されるため、スクロールインジケータは無効にしておくべきです。

図 5-1にページングモードに設定したScroll Viewの例を示します。ここで示しているアプリケーションの実装は、『*PageControl*』のサンプルコードに含まれています。

図 5-1　ページングモードのScroll Viewとスクロールアクションの結果



ページングモードのScroll Viewのサブビューの設定

ページングモードのScroll Viewのサブビューは2つのうちどちらかの方法で設定できます。コンテンツが小さければ、Scroll Viewの`contentSize`の大きさの1つビューに、コンテンツ全体を一度に描画できます。これは最も簡単な実装方法ですが、大きなコンテンツ領域を扱ったり、描画するのに時間のかかるページコンテンツを扱ったりする場合には効果的ではありません。

アプリケーションで大量のページを表示する必要がある、あるいはページコンテンツの描画に時間がかかる可能性がある場合は、アプリケーションでは、各ページごとに1つのビューを使うなど、複数のビューを使用してコンテンツを表示すべきです。こうすることはより複雑ではありますが、パフォーマンスを著しく向上させ、アプリケーションでもっと多くの画面セットをサポートできるようになります。*PageControl*の例では、この複数ビューの手法を使用しています。サンプルコードを研究することで、この手法をどのように実装できるか正確に把握できるでしょう。

ページングモードのScroll Viewで大量のページをサポートするには、デバイス画面と同じサイズの3つのビューインスタンスを使えば実現できます。1つのビューは現在のページを、もう1つのビューは直前のページを、そして3つ目のビューは次のページを表します。ビューは、ユーザがページをスクロールするのに合わせて再利用されます。

Scroll View Controllerを初期化するとき、3つのビューすべてが作成され初期化されます。これらのビューは通常、UIViewのカスタムサブクラスですが、アプリケーションでは必要に応じて`UIImageView`のインスタンスを使用することもできます。その後ビューは、ユーザがスクロールした際に、次のまたは前のページがいつでも適当な場所に配置されてコンテンツの表示準備が整っているように、互いに相対的に配置されます。Controllerは、どのページが現在のページなのか追跡し続ける役目を担います。

ユーザがコンテンツをスクロールしたためにページを構成しなおす必要がある場合を判断するために、Scroll Viewでは`scrollViewDidScroll:`メソッドを実装するデリゲートを必要とします。このメソッドの実装では、Scroll Viewの`contentOffset`を追跡します。そして、現在のビューの幅の中間点を越えたら、ページを構成しなおして、画面上に表示されなくなったビューを、（ユーザがスクロールした向きに対応して）次のページまたは前のページを表す位置に移動する必要があります。その後デリゲートはビューに対して、それぞれの新しい位置にふさわしいコンテンツを描画する必要があることを伝えます。

この手法を使えば、最小限のリソースで大量のコンテンツを表示できます。

ページコンテンツの描画に時間がかかる場合、アプリケーションはビュープールにビューをさらに追加して、スクロールの発生に合わせて次のページまたは前のページのどちら側にもそれらのビューをページとして配置し、現在のコンテンツがスクロールされるのに合わせてこれらの追加ページのページコンテンツを描画することができます。

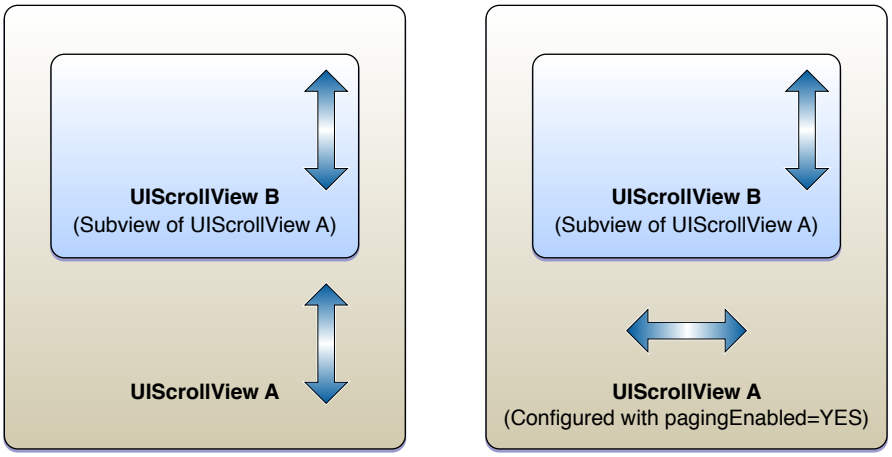
Scroll Viewのネスト

豊かなユーザ体験を提供するために、アプリケーションでScroll Viewをネストすると良いでしょう。iOS 3.0より前では、これを実現するのは不可能ではなかったにしても難しいものでした。iOS 3.0で、この機能は完全にサポートされ、自動的に機能します。

同じ向きのスクロール

同じ向きのスクロールが起こるのは、`UIScrollView`のサブビューである`UIScrollView`が、どちらも同じ向きにスクロールする場合です。この様子を図 6-1の左側に示します。

図 6-1 同じ向きでスクロールするScroll Viewと十字スクロールのScroll View



注： この同じ向きのスクロールはサポートされており、機能として定義された明確な動作があります。ただし、この動作はiOSの将来のバージョンで変更になる可能性があります。

十字スクロール

十字スクロールとは、別のスクロールビューのサブビューであるScroll Viewが90度の角度でスクロールする場合に使われる用語です（図 6-1の右側の図参照）。

十字スクロールの1つの例としては「株価（Stocks）」アプリケーションがあります。最上位のビューはTable Viewですが、最下部のビューはページングモードを使用して構成された水平Scroll Viewです。3つのサブビューのうち2つはカスタムビューですが、3つ目のビュー（ニュース記事を含む）は水平Scroll Viewのサブビューである`UITableView`（`UIScrollView`のサブクラス）です。水平にニュースビューまでスクロールした後、その中身を上下にスクロールできます。

前述のとおり、アプリケーションではスクロールのネストをサポートするために必要なことは何もありません。機能はデフォルトでサポートされ、提供されています。

書類の改訂履歴

この表は「*iOS Scroll View プログラミングガイド*」の改訂履歴です。

日付	メモ
2011-06-06	リスト1-2でコンテンツサイズを訂正しました。
	「Scroll ViewのcontentInsetプロパティおよびscrollIndicatorInsetsプロパティの設定」（17 ページ）の誤りを修正。
	さらに、分かりやすくなるよう編集。
2010-07-10	ピンチジェスチャを使った基本的なズームの色定義を修正しました。
2010-07-07	『iPhone OS Scroll Viewプログラミングガイド』から文書名を変更しました。
	『iPhone OS Scroll Viewプログラミングガイド』から文書名を変更しました。
2010-06-14	誤字を訂正しました。
2010-03-24	導入部分を書き換えました。図と細かな誤字を訂正しました。
2010-02-24	Scroll Viewを使用してスクロールと拡大縮小が可能なユーザインターフェイスを実装する方法について説明した新規文書。

書類の改訂履歴