iOSイベント処理ガイド

2011-03-10

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

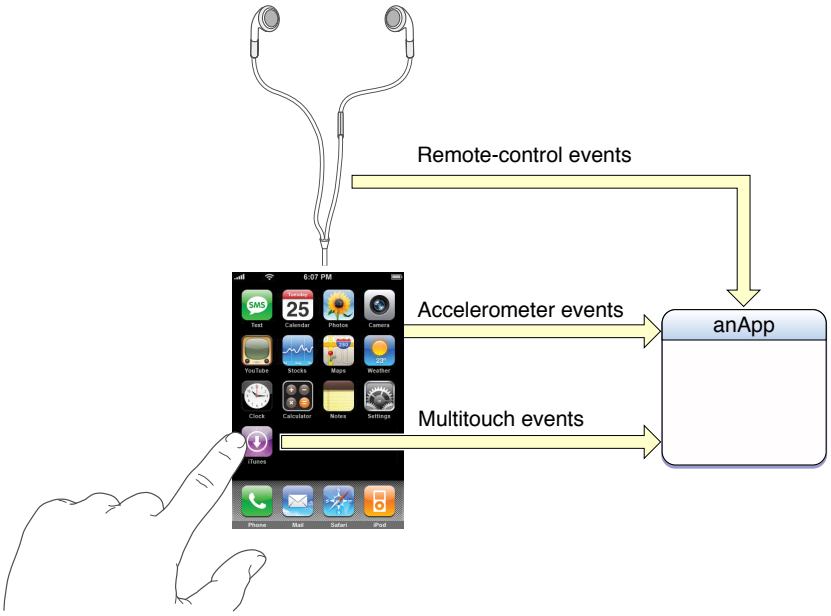
アップルジャパン株式会社
〒163-1450 東京都新宿区西新宿
3 丁目20 番2 号
東京オペラシティタワー
http://www.apple.com/jp/

目次

図、表、リスト

iOSのイベントについて

「イベント」とは、アプリケーションに対し、ユーザの操作（アクション）を通知するために送信されるもの（オブジェクト）のことです。iOSのイベントには、マルチタッチやモーション（たとえばデバイスの加速度センサーが検出して通知）、あるいはマルチメディア再生制御に使われるものなど、さまざまな形態があります。最後に挙げたマルチメディア再生制御用のイベントは、ヘッドセットその他の外付け機器から送られてくるので、リモートコントロールイベントと言います。



UIKitやCore Motionなどのフレームワークは、iOSにおけるイベントの伝達や送信を行います。

概要

ユーザが画面をタッチすると、アプリケーションはマルチタッチイベントを受け取る

iPhone、iPad、iPod touchのマルチタッチインターフェイスは、ユーザがアプリケーション画面にタッチすると、低レベルのイベントを生成します。アプリケーションはこのイベントを（`UIEvent`オブジェクトの形で）、タッチされた画面に送ります。この画面は、多くの場合、各イベントオブジェクトが表す「タッチ」の状況を分析して、適切に応答します。

アプリケーションは一般に、ユーザのタッチ操作を、タップやスワイプなどといった、典型的な「ジェスチャ」に当てはめて解釈しようとします。その手段として、「Gesture Recognizer」というUIKitのクラス群を利用できます。各クラスはそれぞれ、特定のジェスチャを認識するようになっています。

関連する章： 「イベントのタイプと送信」（11 ページ）、「マルチタッチイベント」（17 ページ）、「Gesture Recognizer」（35 ページ）

ユーザがデバイスを動かすと、アプリケーションはモーションイベントを受け取る

モーションイベントの形態は動きの種類によって異なります。このイベントも、各種のフレームワークで処理できます。ユーザがデバイスを振り動かす（シェイクする）と、UIKitは`UIEvent`オブジェクトをアプリケーションに送信します。この「シェイク」モーションイベントを生じさせるジェスチャは、多くの場合、直前のアクションを取り消しまたは再実行する、という意味で使われます。加速度センサーやジャイロスコープのデータを、高頻度かつ連続的に受け取る必要があるアプリケーションには、Core Motionフレームワークを使ってください（ジャイロスコープは一部のデバイスにのみ搭載）。また、`UIAccelerometer`クラスを使って、加速度センサーのデータを受信、処理することもできます。

関連する章： 「イベントのタイプと送信」（11 ページ）、「モーションイベント」（51 ページ）

ユーザがマルチメディアコントローラを操作すると、リモートコントロールイベントが送信される

ヘッドセットその他の外付け機器は、Appleが規定した仕様に従い、リモートコントロールイベントを（UIKitフレームワークを介して）、音声や画像を再生するアプリケーションに送信できるようになっています。マルチメディアを表示するビューは、イベントを受け取り、ユーザの指示（一時停止、早送りなど）に従って音声や画像の再生を制御します。

関連する章： 「イベントのタイプと送信」（11 ページ）,「マルチメディアのリモートコントロール」（69 ページ）

本書の使い方

関心のあるイベントの種類によらず、最初に「イベントのタイプと送信」（11 ページ）を読んでおくとよいでしょう。この章では重要な予備知識を解説しています。

関連項目

iPhoneその他のデバイスにはGPSや方位計が搭載されており、これも低レベルのデータを生成して、アプリケーションに送信するようになっています。『*Location Awareness Programming Guide*』に、このデータを受信し、処理する方法が載っています。

iOS Reference Libraryに収録されているサンプルコードプロジェクトの多くは、マルチタッチイベントの処理や、Gesture Recognizerの使い方の実例になっています。『*Touches*』、『*Metronome*』、『*CopyPasteTile*』、『*SimpleGestureRecognizers*』などのプロジェクトがあります。

イベントのタイプと送信

iPhone、iPad、iPod touchには、入力データストリームを生成するハードウェアがいくつか組み込まれており、このデータにはアプリケーションもアクセスできます。「マルチタッチ」技術により、仮想キーボードなど「画面上に表示されたもの」を直接操作できるようになりました。3つの加速度センサーは、3次元空間の各軸に沿った加速度を測ります。ジャイロスコープ（一部の機種のみ）は、3つの軸の周りの回転速度を測ります。GPS（Global Positioning System、全地球測位システム）と方位計は、位置と方角を測ります。それぞれのハードウェアが、タッチ操作、デバイスの動き、位置の変化を検出し、生成したデータをそのまま、システムフレームワークに渡します。フレームワークはこの生データを「イベント」の形に加工し、アプリケーションに渡して処理を委ねます。

以下のセクションでは、どのようなフレームワークがあるか、イベントをどのような形にしてアプリケーションに送信するか、を説明します。

注： この文書では、タッチイベント、モーションイベント、リモートコントロールイベントの3種類しか扱いません。GPSや方位計が生成するデータの取り扱いについては、『*Location Awareness Programming Guide*』を参照してください。

UIKitのイベントオブジェクトとそのタイプ

イベントとは、ユーザの操作（アクション）を表すオブジェクトのことです。デバイスに組み込まれたハードウェアが検出し、iOSに送ります。指で画面上をタッチする操作、デバイスを振り動かす操作などがあります。多くのイベントは、UIKitフレームワークに実装された、`UIEvent`クラスに属するインスタンスです。`UIEvent`オブジェクトは、タッチ操作などのユーザイベントに関する「状態」をカプセル化しています。また、イベントの生成時刻も記録されています。ユーザが何らかの操作をする（画面を指でタッチする、画面上で指を動かす、など）と、オペレーティングシステムは継続的にイベントオブジェクトをアプリケーションに送信して、処理を委ねます。

現状のUIKitは、タッチイベント、「振り動かす（シェイク）」モーションイベント、リモートコントロールイベントの3種類を認識できます。`UIEvent`クラスには、リスト 1-1に示す`enum`定数が宣言されています。

リスト 1-1　イベントのタイプとサブタイプを表す定数

```
typedef enum {
    UIEventTypeTouches,
    UIEventTypeMotion,
    UIEventTypeRemoteControl,
} UIEventType;

typedef enum {
    UIEventSubtypeNone                              = 0,

    UIEventSubtypeMotionShake                       = 1,
```

```
    UIEventSubtypeRemoteControlPlay                = 100,
    UIEventSubtypeRemoteControlPause               = 101,
    UIEventSubtypeRemoteControlStop                = 102,
    UIEventSubtypeRemoteControlTogglePlayPause     = 103,
    UIEventSubtypeRemoteControlNextTrack           = 104,
    UIEventSubtypeRemoteControlPreviousTrack       = 105,
    UIEventSubtypeRemoteControlBeginSeekingBackward = 106,
    UIEventSubtypeRemoteControlEndSeekingBackward   = 107,
    UIEventSubtypeRemoteControlBeginSeekingForward  = 108,
    UIEventSubtypeRemoteControlEndSeekingForward    = 109,
} UIEventSubtype;
```

各イベントは、いずれかのタイプおよびサブタイプに属しており、`UIEvent`の`type`および`subtype`というプロパティを介してアクセスできます。「タイプ」は、タッチ、モーション、リモートコントロールの各イベントを区別するために使います。iOS 3.0では、モーションイベントのサブタイプとして「シェイク」（`UIEventSubtypeMotionShake`）があるほか、リモートコントロール関連の多数のサブタイプが定義されています。タッチイベントについては、サブタイプは常に`UIEventSubtypeNone`となります。

リモートコントロールイベントは、コマンドの形で、システムトランスポートコントロール、またはAppleの仕様に準拠したヘッドセットなどの外付け機器が生成します。このイベントの目的は、ユーザがマルチメディアコンテンツの再生を制御できるようにすることです。リモートコントロールイベントはiOS 4.0で新たに導入されました。詳しくは「マルチメディアのリモートコントロール」（69 ページ）を参照してください。

`UIEvent`オブジェクトをいつまでも保持するようなコードは避けるべきです。イベントオブジェクトの現在の状態を保存しておき、後で評価する必要がある場合は、状態ビットをインスタンス変数にコピーする、ディレクトリオブジェクトに保存するなど、適切な手段を講じてください。

iOSが稼働するデバイスは、ほかのタイプのイベントもアプリケーションに送信し、処理を委ねることがあります。これは`UIEvent`オブジェクトではありませんが、ハードウェアが生成した何らかの測定値をカプセル化している、という点は同じです。詳しくは「モーションイベントのタイプ」（15 ページ）を参照してください。

イベントの送信

イベントは、所定の経路に沿って、これを処理するオブジェクトに送信されます。「アプリケーションをリモートコントロールイベントに対応させる手順」（69ページ）で説明するように、ユーザがデバイスの画面にタッチすると、iOSは一連のタッチ操作を認識して1つの`UIEvent`オブジェクトにまとめ、アクティブなアプリケーションのイベントキューに入れます。デバイスを振り動かす、という操作を、システムがモーションイベントとして解釈した場合、これを表すイベントオブジェクトも、同様にイベントキューに入れられます。アプリケーションを管理しているシングルトン`UIApplication`オブジェクトは、イベントキューの先頭からイベントを1つ取り出して、処理のためにそれを送信します。通常、イベントは、そのアプリケーションの中心となるウインドウ（その時点でユーザイベントの対象となっているウインドウ）に送信されます。そのウインドウを表すウインドウオブジェクトは、このイベントを、最初にイベントを処理するオブジェクトに送ります。このイベントオブジェクトは、タッチイベントかモーションイベントか、によって異なります。

- **タッチイベント**。ウインドウオブジェクトは、ヒットテストという手法と、レスポンダチェーンを使って、タッチイベントを受け取るべきビューを見つけます。ヒットテストでは、ビュー階層の最上位にあるビューに対して`hitTest:withEvent:`メソッドを呼び出します。このメソッ

ドは、ビュー階層に沿ってサブビューを検索し、それぞれに対してpointInside:withEvent:を呼び出します。これがYESを返すサブビューを再帰的に検索して降りていくと、タッチされた点を境界内に含む、最下層のビューに到達します。これが「ヒットテスト」ビューになります。

「ヒットテスト」ビューがイベントを処理できなければ、レスポンダチェーンを順次たどって、処理できるビューを探します（「レスポンダオブジェクトとレスポンダチェーン」（13 ページ）を参照）。タッチオブジェクト（「イベントとタッチ」（18 ページ）を参照）には、これが生存している間、「ヒットテスト」ビューが関連づけられています。その後、オブジェクトで表される「タッチ」の位置がビュー外に移動してしまっても、この関連づけは変わりません。「ヒットテスト」（30 ページ）では、ヒットテストのプログラム上の注意点について説明します。

- **モーションイベントとリモートコントロールイベント**。ウインドウオブジェクトは、「シェイク」モーションイベントやリモートコントロールイベントをそれぞれ、ファーストレスポンダに送信して処理を委ねます（ファーストレスポンダについては、「レスポンダオブジェクトとレスポンダチェーン」を参照）。

「ヒットテスト」ビューとファーストレスポンダは、同じビューオブジェクトであることが多いのですが、常にそうというわけではありません。

UIApplicationオブジェクトや各UIWindowオブジェクトのsendEvent:メソッドは、適切な送信先にイベントを振り分けます。（いずれのクラスもメソッドのシグネチャは共通です）。これらのメソッドは、アプリケーションが受け取るイベントの入り口であるため、UIApplicationまたはUIWindowのサブクラスを作成して、sendEvent:メソッドをオーバーライドし、イベントを監視できます（ある種のアプリケーションではこれが必要となります）。メソッドをオーバーライドする場合、必ずスーパークラスの実装（[super sendEvent:theEvent]）を呼び出してください。また、イベントの送信経路を変更してはいけません。

レスポンダオブジェクトとレスポンダチェーン

ここでは「レスポンダ」の考え方について説明します。レスポンダオブジェクトとは何か、イベント送信のアーキテクチャにどのように組み込まれているか、といった事柄です。

*レスポンダオブジェクト*とは、イベントに応答し、処理できるオブジェクトのことです。UIResponderはあらゆるレスポンダオブジェクトの基底クラスであり、これを単に「レスポンダ」と呼ぶこともあります。このクラスには、イベント処理だけでなくレスポンダに共通する動作のインターフェイスが定義されています。UIApplication、UIView、およびUIViewから派生したすべてのUIKitクラス（UIWindowを含む）は、UIResponderを直接または間接的に継承しており、したがってそのインスタンスはレスポンダオブジェクトになります。

*ファーストレスポンダ*とは、アプリケーション内で最初にタッチ以外のイベントを受け取るよう設計されたレスポンダオブジェクト（通常は、UIViewオブジェクト）のことです。UIWindowオブジェクトは、メッセージにイベントを含めてファーストレスポンダに送信し、そのイベントを最初に処理する機会を与えます。レスポンダオブジェクトがこのメッセージを受け取るためには、canBecomeFirstResponderメソッドを、YESを返すように実装しなければなりません。さらに、becomeFirstResponderメッセージも受信する（自分自身をファーストレスポンダにする）必要があります。ファーストレスポンダは、次のようなイベントやメッセージを受信できる、ウインドウ内の最初のビューです。

- モーションイベント - UIResponderのモーション処理メソッド（「「シェイク」モーションイベント」（51 ページ）を参照）の呼び出しを介して受信

- リモートコントロールイベント-`UIResponder`のメソッド`remoteControlReceivedWithEvent:`の呼び出しを介して受信
- アクションメッセージ - ユーザが（ボタン、スライダなどの）コントロール部品を操作し、そのメッセージの送信先が指定されていない場合
- 編集メニューメッセージ-ユーザが編集メニューのコマンドをタップしたとき（「編集メニューの表示と管理」を参照）

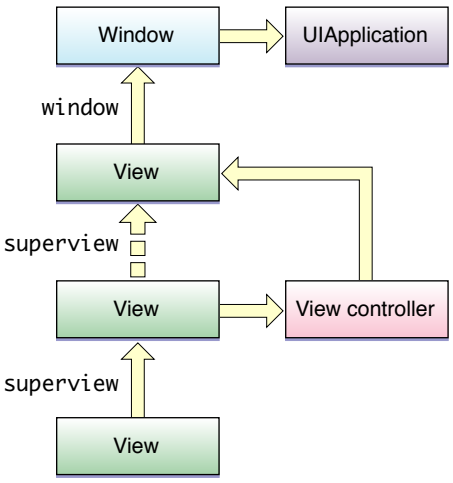
ファーストレスポンダはテキスト編集にも関与します。編集フォーカスがあるテキストビューやテキストフィールドは、ファーストレスポンダになり、仮想キーボードが現れます。

注： モーションイベント、アクションメッセージ、編集メニューメッセージについては、アプリケーションが明示的に、これを処理するファーストレスポンダを設定する必要があります。ユーザがタップするテキストフィールドやテキストビューは、UIKitが自動的に、ファーストレスポンダとして設定します。

ファーストレスポンダやヒットテストビューがイベントを処理しない場合、UIKitは*レスポンダチェーン*にたどり、*次のレスポンダ*に（メッセージの形で）イベントを渡して、処理できるかどうかを調べます。

レスポンダチェーンはレスポンダオブジェクトの連鎖で、イベント、アクションメッセージ、編集メニューメッセージは、この経路に沿って渡されていきます。これによって、レスポンダオブジェクトは、イベント処理の役割をほかの上位レベルのオブジェクトに転送することができます。イベントは、アプリケーションがイベントを処理できるオブジェクトを見つけるまで、レスポンダチェーンをさかのぼります。「ヒットテスト」ビューもレスポンダオブジェクトなので、タッチイベントを処理する際、アプリケーションはレスポンダチェーンを利用できます。レスポンダチェーンは、図 1-1のように、「次の」レスポンダ（`nextResponder`メソッドの戻り値）を順次たどっていけるようになっています。

図 1-1 iOSにおけるレスポンダチェーン



システムは、タッチイベントを送信する際、最初にある特定のビューに送信します。タッチイベントの場合、それは`hitTest:withEvent:`メソッドが返すビューです。一方、「シェイク」モーションイベント、リモートコントロールイベント、アクションメッセージ、編集メニューメッセージの場合は、ファーストレスポンダに送信するようになっています。最初のビューで処理できない場合、イベントは次のように、レスポンダチェーンに沿って順に転送されていきます。

1. 「ヒットテスト」ビューやファーストレスポンダは、ビューコントローラがあれば、それにイベントやメッセージを渡します。なければスーパービューに渡します。
2. ビューまたはビューコントローラがイベントやメッセージを処理できなければ、そのスーパービューにイベントやメッセージを渡します。
3. やはりイベントやメッセージを処理できなければ、階層をさかのぼり、上位のビューについてステップ1〜2を繰り返します。
4. ビュー階層の最上位まで到達しても処理できなかった場合は、ウインドウオブジェクトに渡して処理を委ねます。
5. `UIWindow`オブジェクトも処理できなければ、シングルトンであるアプリケーションオブジェクトに渡します。

アプリケーションオブジェクトも処理できなかった場合、そのイベントやメッセージは廃棄されてしまいます。

「シェイク」モーションイベント、リモートコントロールイベント、アクションメッセージ、編集メニューメッセージを処理するビューを独自に実装する場合、イベントやメッセージを直接`nextResponder`に渡す、という方法で、レスポンダチェーンをたどる処理を組み込んではいけません。スーパークラスに実装されたイベント処理メソッドを呼び出して、UIKitにレスポンダチェーンをたどる処理を委ねてください。

モーションイベントのタイプ

モーションイベントは、3つの加速度センサー、およびジャイロスコープ（一部のデバイスにのみ搭載）という、2種類のハードウェアから送られてきます。加速度センサーは、特定の線形パスに沿って速度の変化を時系列に測定します。この加速度センサーを組み合わせることによって、任意の方向へのデバイスの動きを検出できます。このデータを使用して、デバイスの瞬間的な動きと、重力に対するデバイスの現在の向きの両方を追跡できます。ジャイロスコープは、3つの軸の周りの回転速度を測ります（加速度センサーは軸ごとに1つずつありますが、この文書では1つのものとして説明します）。

Core Motionフレームワークは主として、加速度センサーやジャイロスコープから得られる生のデータにアクセスし、これを処理するアプリケーションに転送するようになっています。これに加え、特別なアルゴリズムでこの生データを組み合わせて処理し、「動き」を表すデータとしてアプリケーションに渡します。Core Motionから送られてくるモーションイベントは、次の3つのデータオブジェクトから成り、いずれもいくつかの測定値をカプセル化しています。

- `CMAccelerometerData`オブジェクトは、3次元空間の各軸に沿った加速度をカプセル化しています。
- `CMGyroData`オブジェクトは、3つの軸の周りの回転速度をカプセル化しています。

- CMDeviceMotionオブジェクトは、デバイスの「姿勢」と、回転速度や加速度をより使いやすい形にしたものをカプセル化しています。

Core Motionは、UIKitのアーキテクチャや規約とは独立しています。UIEventモデルとの関連はなく、ファーストレスポンダやレスポンダチェーンについて考慮しているわけでもありません。モーションイベントを、これを要求するアプリケーションに、直接送信するだけです。

CMMotionManagerクラスは、Core Motionにアクセスするための「集中窓口」です。このクラスのインスタンスを生成し、更新間隔を（明示して、または暗黙で）指定し、更新を開始するよう要求して、モーションイベントが届いたらそれを処理することになります。「Core Motion」（56 ページ）に、その具体的な手順を説明します。

少なくとも加速度センサーのデータに関して、Core Motionの代替として利用できる、UIAccelerometerというクラスがUIKitフレームワークにあります。このクラスを使う場合、加速度センサーイベントはUIAccelerationオブジェクトの形で送信されます。UIAccelerometer はUIKitの一部を成すものですが、やはりUIEventやレスポンダチェーンのアーキテクチャとは独立しています。UIKitの機能の使い方については、「UIAccelerometerを介した加速度センサーイベントへのアクセス」（54 ページ）を参照してください。

メモ： UIAccelerometerクラス、UIAccelerationクラスとも、将来の版では廃止になります。加速度センサーイベントを扱うアプリケーションは、今後、Core Motion APIを使うようにしてください。

iOS 3.0以降では、シェイクモーションなどの特定の種類のモーションをジェスチャとして検出しようとする場合、加速度センサーインターフェイスの代わりに、モーションイベントUIEventTypeMotionを使用してそのモーションを処理することを検討するべきです。高い頻度で連続的にモーションデータを受信し処理する場合は、代わりにCore Motionの加速度センサーAPIを使ってください。モーションイベントについては「「シェイク」モーションイベント」（51 ページ）で解説します。

マルチタッチイベント

注： この章の内容は、『*iOS App Programming Guide*』に記載されていたものです。特にiOS 4.0向けに改訂した事項はありません。

iOSのタッチイベントは、Multi-Touchモデルに基づいています。ユーザは、マウスとキーボードを使用する代わりに、デバイスの画面をタッチしてオブジェクトを操作したりデータを入力したりその他の意図を伝えたりします。iOSは、1本以上の指で画面にタッチする操作を *マルチタッチシーケンス*の一部として認識します。このシーケンスは、最初の指が画面に触れたときに始まって、最後に指が画面から離れたときに終わります。iOSは、マルチタッチシーケンスの間中、画面に触れている指を追跡して、それぞれのタッチの特徴を記録します。それには、画面上での指の位置と、タッチが発生した時間が含まれます。通常、アプリケーションは、複数のタッチの特定の組み合わせをジェスチャとして認識して、ユーザに直感的にわかる方法でそれに応答します。たとえば、ピンチジェスチャに応答してコンテンツを拡大したり、フリック（はじく）ジェスチャに応答してコンテンツをスクロールしたりします。

メモ： 画面上の指は、マウスポインタとはかなり異なるレベルの精度を提供します。ユーザが画面に触れたとき、その接触領域は実際には楕円形で、ユーザ自身が触れたと考えている点の下方にずれている傾向があります。この「接触面」はまた、どの指が画面に触れたか、指のサイズ、画面上の指の圧力、指の向き、その他の要因などによってもサイズや形状が変わります。基盤のMulti-Touchシステムは、このすべての情報を分析し、単一のタッチポイントを計算します。

iOS 4.0は今でも、iPhone 4（および将来の高解像度デバイス）に対するタッチ操作を320×480の座標系で通知することにより、互換性を維持していますが、iOS 4.0以降向けに構築されたアプリケーションの場合、実際の解像度は縦横とも2倍になっています。具体的に言うと、iOS 4向けに構築したアプリケーションをiPhone 4上で動かした場合は、タッチ操作の位置座標の値に0.5の端数が生じることがあります。旧デバイス上で動かした場合は、端数は生じません。タッチ操作を処理するパスに、整数値に丸めるコードがあると、精度が失われる可能性があります。

UIKitのクラスの多くは、そのクラスのオブジェクト特有の方法でマルチタッチイベントを処理します。このことは、特に、`UIControl`のサブクラス（`UIButton`、`UISlider`など）に当てはまります。これらのサブクラスのオブジェクト（コントロールオブジェクトと呼ばれる）は、タップや特定の方向へのドラッグなど、特定のタイプのジェスチャを受け付けます。正しく設定されていれば、そのジェスチャが発生したときに対象となるオブジェクトにアクションメッセージを送信します。UIkitのその他のクラスは、ジェスチャを別のコンテキストで処理します。たとえば、`UIScrollView`は、Table View、Text View、そして大きなコンテンツ領域を持つその他のビューに対してスクロール動作を提供します。

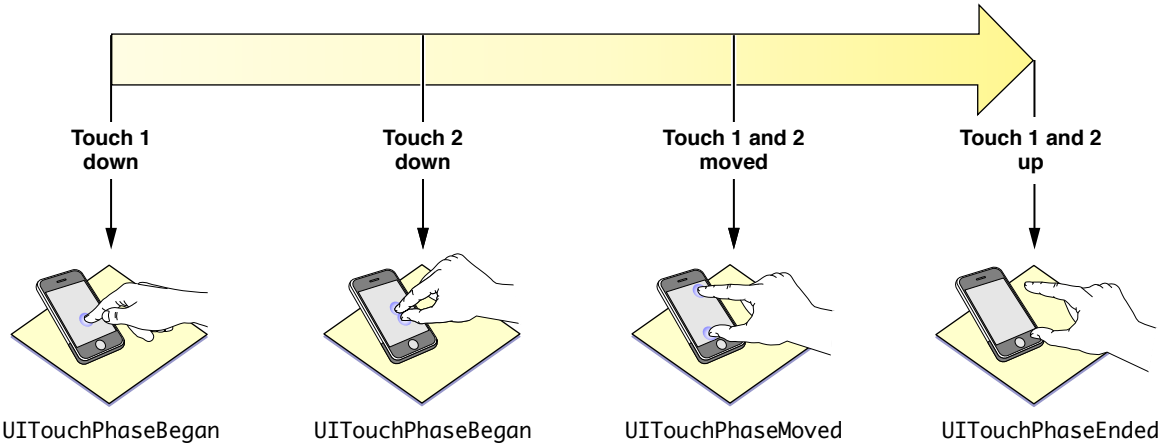
アプリケーションによっては、自分で直接イベント処理を行わずに、この動作をUIKitのクラスに任せることもできます。ただし、`UIView`のカスタムサブクラスを作成し（iOSの開発ではよくあるケースです）、そのビューを特定のタッチイベントに応答させる場合は、それらのイベントを処理するために必要なコードを実装しなければなりません。さらに、UIKitオブジェクトに、イベントに対して異なる対応をさせる場合は、そのフレームワーククラスのサブクラスを作成して適切なイベント処理メソッドをオーバーライドする必要があります。

イベントとタッチ

iOSでは、タッチとは画面上での1本の指の存在または移動を表し、一意のマルチタッチシーケンスに属します。たとえば、ピンチ-クローズジェスチャには、画面上で2本の指がそれぞれ反対方向から互いに近づくという2つのタッチが含まれています。タップ、ダブルタップ、ドラッグ、フリック（画面上を1本の指ですばやくはじく）など、1本の指の単純なジェスチャもあります。アプリケーションが、それよりも複雑なジェスチャを認識することもあります。たとえば、アプリケーションにダイヤル型のカスタムコントロールを備え、ユーザがそれを複数の指で回して何らかの変数の微調整を行えるようにすることが考えられます。

`UIEventTypeTouches`型の`UIEvent`オブジェクトは、タッチイベントを表します。指が画面にタッチし、表面上を動いている間、システムは継続的に、このタッチイベントオブジェクト（あるいは単に「タッチイベント」ともいう）をアプリケーションに送信します。イベントは、1つのマルチタッチシーケンスの間のすべてのタッチのスナップショットを提供します。最も重要なのは、特定のビューへの新規のタッチや、変化のあったタッチです。図 2-1に示すように、マルチタッチシーケンスは、1本の指が最初に画面に触れたときに始まります。それに続けて、その他の指が画面に触れたり、すべての指が画面上を移動したりする場合もあります。シーケンスは、最後の指が画面を離れたときに終了します。アプリケーションは、すべてのタッチの各フェーズで、イベントオブジェクトを受け取ります。

図 2-1　マルチタッチシーケンスとタッチフェーズ

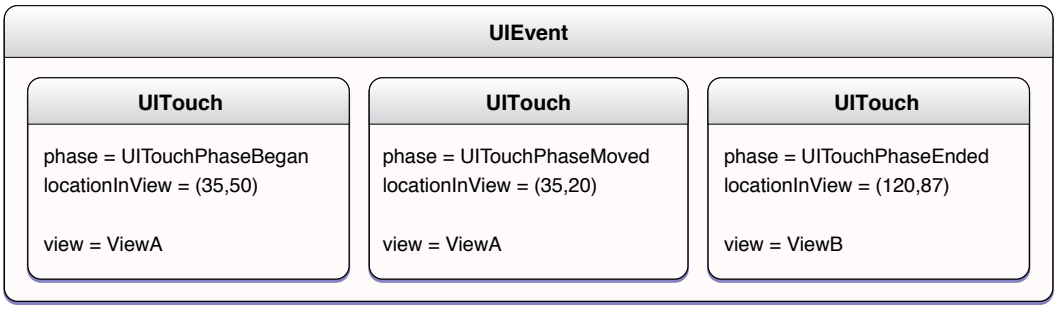


タッチには、`UITouch`オブジェクトで表されるように、時間的な側面と空間的な側面の両方があります。フェーズと呼ばれる時間的な側面は、タッチが始まったばかりのとき、移動しているか静止しているか、タッチが終わったとき（指が画面から離れたとき）を表します。

一方、タッチの空間的な側面とは、タッチされたオブジェクトとの関連付け、およびオブジェクト内の位置に関するものです。1本の指が画面に触れるとそのタッチは、その下にあるウインドウおよびビューに関連付けられます。その関連付けは、イベントが存続している間ずっと維持されます。複数のタッチが同時に行われた場合は、それらが同じビューに関連付けられている場合にのみまとめて扱われます。同様に、2つのタッチがすばやく立て続けに行われた場合はそれらが同じビューに関連付けられている場合にのみ、マルチタップとして扱われます。タッチオブジェクトは、ビューまたはウインドウ内におけるタッチの現在位置と、（もしあれば）直前の位置を保持しています。

1つのイベントオブジェクトには、現在のマルチタッチシーケンスを構成するすべてのタッチオブジェクトが含まれ、1つのビューまたはウインドウに固有のタッチオブジェクトを提供します（図2-2を参照）。1つのタッチオブジェクトは、1つの指に対応してシーケンスの間中、存続します。また、UIKitは、その間ずっとその指を追跡しながらタッチオブジェクトを変遷させます。変遷するタッチ属性は、タッチのフェーズ、ビュー内の位置、直前の位置、およびタイムスタンプです。イベント処理コードは、これらの属性を評価してタッチイベントへの対応方法を決定します。

図 2-2　`UIEvent`オブジェクトとそれに対応する`UITouch`オブジェクトの関係



システムは、いつでもマルチタッチシーケンスをキャンセルできるため、イベント処理アプリケーションでは適切に対応できるように備えておく必要があります。キャンセルは、電話の着信など優先されるシステムイベントによって発生します。

タッチイベント処理のアプローチ

カスタムビューに対するタッチ操作を受け付けるアプリケーションでは、扱い方が定まったジェスチャを検出し、それに応じた処理をすることも多いでしょう。ここに言うジェスチャには、タップ（1回または複数回）、ピンチ（表示の拡大/縮小）、スワイプ、画面のパンやドラッグ、2本指でビューを回転する操作などがあります。

タッチイベントの処理コードを実装して、ジェスチャを認識、処理することもできなくはありませんが、複雑でバグが入り込みやすく、開発にはかなりの時間を要します。代わりに、iOS 3.2で取り入れられた「Gesture Recognizer」クラス群を使えば、一般的なジェスチャの解釈や処理を簡単に実現できます。Gesture Recognizerを使うためには、そのインスタンスを生成し、タッチ操作を受け付けるビューにアタッチし、必要な設定をしたうえで、アクションセレクタおよび対象オブジェクトに割り当てることになります。Gesture Recognizerはジェスチャを認識すると、アクションメッセージを対象オブジェクトに送信して、これに応じた処理を委ねます。

独自のGesture Recognizerが必要ならば、`UIGestureRecognizer`のサブクラスを作って実装します。マルチタッチシーケンスの一連のイベントを分析して、独自のジェスチャを認識するプログラムを実装することになりますが、そのためには、この章に述べる予備知識が不可欠です。

Gesture Recognizerの詳細については、「Gesture Recognizer」（35 ページ）を参照してください。

タッチイベントの送信の統制

UIKitは、アプリケーションでのイベント処理を容易にしたり、`UIEvent`オブジェクトのストリームを完全にオフにするためのプログラミング手段を提供しています。以下に、これらのアプローチの概要を示します。

- **タッチイベントの送信を停止する**。デフォルトでビューはタッチイベントを受け取りますが、`userInteractionEnabled`プロパティを`NO`と設定することにより、イベントの受け取りを停止することができます。ビューはまた、非表示であったり透過的であったりすると、これらのイベントを受け取りません。

- **一定時間タッチイベントの送信を停止する**。アプリケーションは、`UIApplication`の`beginIgnoringInteractionEvents`メソッドを呼び出して、その後で`endIgnoringInteractionEvents`メソッドを呼び出すことができます。最初のメソッドによって、アプリケーションはタッチイベントの受け取りを完全に停止することができます。2番目のメソッドを呼び出すと、マルチタッチイベントの受け取りが再開されます。アニメーションを実行している間、イベントの送信を停止したい場合などに便利です。

- **複数タッチの送信を有効にする**。デフォルトでは、ビューは、マルチタッチシーケンスの間、最初のタッチ以外のすべてのタッチを無視します。ビューで複数のタッチを処理する場合は、ビューに対するこの機能を有効にする必要があります。これは、プログラムでビューの`multipleTouchEnabled`プロパティに`YES`を設定するか、Interface Builderで、関連するビューのInspectorに対し、関連する属性を設定することにより行います。

- **イベントの送信先を1つのビューに限定する**。デフォルトでは、ビューの`exclusiveTouch`プロパティは`NO`となっています。これは、ウインドウ中のほかのビューがタッチを受信することを妨げない、ということを意味します。このプロパティを`YES`に設定すると、このビューがタッチを追跡している間、これがウインドウの中でタッチを追跡する唯一のビューとなります。ウインドウ内のほかのビューはこれらのタッチを受け取れません。ただし、このように「排他的タッチ」のマークが付いたビューは、同じウインドウ内の別のビューに関連付けられているタッチは受け取りません。排他的タッチのビューに指が触れると、そのビューがウインドウ内で指を追跡している唯一のビューである場合にのみ、そのタッチが送信されます。非排他的ビューに指が触れると、排他的タッチのビューで別の指が追跡されていない場合にのみタッチが送信されます。

- **イベントの送信先をサブビューに限定する**。`UIView`のカスタムクラスで、`hitTest:withEvent:`をオーバーライドして、マルチタッチイベントの送信先をサブビューに限定することができます。この手法に関する詳細な解説については、「ヒットテスト」（30 ページ）を参照してください。

マルチタッチイベントの処理

マルチタッチイベントを処理するためには、まずレスポンダクラスのサブクラスを作成する必要があります。これは次のいずれかです。

- カスタムビュー（`UIView`のサブクラス）

- `UIViewController`、またはそのUIKitサブクラスのサブクラス

- UIKitのビュークラスまたはコントロールクラス（UIImageView、UISliderなど）のサブクラス
- UIApplicationまたはUIWindowのサブクラス（この作り方をすることは稀）

ビューコントローラは通常、レスポンダチェーンを通して、当初はビューに送信されたタッチイベントを受信します。ただし、当該ビューがタッチ処理メソッドをオーバーライドしている場合を除きます。

先に作成したサブクラスのインスタンスがマルチタッチイベントを受信するためには、このサブクラスに、以下に挙げるような、タッチイベントを処理するUIResponderメソッドをいくつか実装する必要があります。さらに、ビューは可視（透明でも非表示でもない）であり、userInteractionEnabledプロパティがYES（これがデフォルト値）でなければなりません。

以降の各セクションでは、タッチイベントの処理メソッドについて説明し、一般的なジェスチャの処理方法を示します。さらに、複雑なマルチタッチイベントを処理するレスポンダオブジェクトの例も示します。最後に、イベント転送について説明し、イベント処理のテクニックをいくつか提案します。

イベント処理メソッド

1つのマルチタッチシーケンスの間、アプリケーションは一連のイベントメッセージを対象であるレスポンダにディスパッチします。これらのメッセージを受け取って処理するには、レスポンダオブジェクトのクラスに、UIResponderで宣言されている以下のメソッドを少なくとも1つ、場合によってはすべて、実装している必要があります。

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
```

タッチフェーズにおいて新しいタッチまたは変化したタッチがあったときに、アプリケーションは以下のようなメッセージを送信します。

- 1本以上の指が画面に触れたときは、touchesBegan:withEvent:メッセージを送ります。
- 1本以上の指が移動したときは、touchesMoved:withEvent:メッセージを送ります。
- 1本以上の指が画面から離れたときは、touchesEnded:withEvent:メッセージを送ります。
- タッチシーケンスが電話の着信などのシステムイベントによってキャンセルされたときは、touchesCancelled:withEvent:メッセージを送ります。

以上のメソッドはそれぞれ、たとえばtouchesBegan:withEvent:はUITouchPhaseBeganに、というように、タッチフェーズに対応しています。UITouchオブジェクトのフェーズは、phaseプロパティを評価することにより調べることができます。

イベント処理メソッドを起動するメッセージは、2つのパラメータを渡します。第1のパラメータは、該当するフェーズの新規タッチ、または変化のあったタッチを表すUITouchオブジェクトのセットです。第2のパラメータは、この特定のイベントを表すUIEventオブジェクトです。このイベントオブジェクトから、イベントに対応するすべてのタッチオブジェクトを取得したり、特定のビュー

またはウインドウに対応するタッチオブジェクトだけを抽出したサブセットを取得したりできます。これらのタッチオブジェクトの中には、前回のイベントメッセージ以降変化のないタッチや、変化はあったもののフェーズが異なるタッチを表すオブジェクトも含まれます。

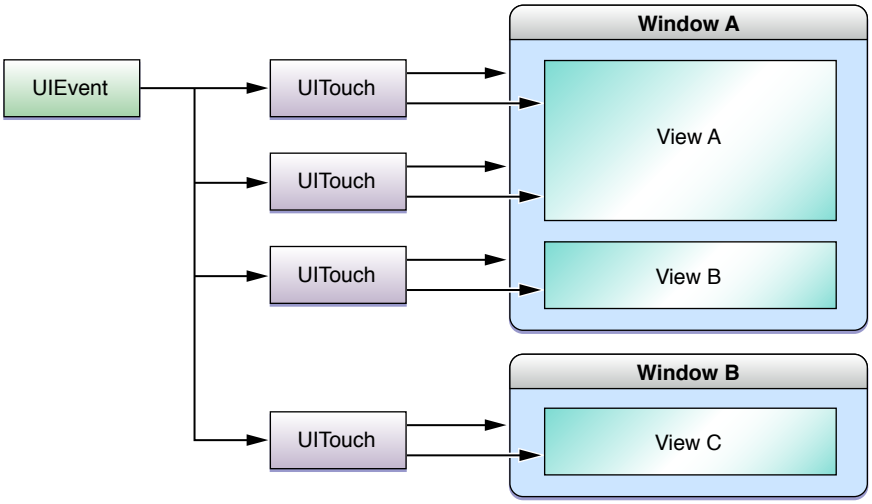
タッチイベント処理の基本

あるフェーズのイベントを処理する際には、UITouchオブジェクト群を受け取って、そのプロパティを評価し、位置を取得し、それに従って次のフェーズに進む、という形になることが多いでしょう。ここで渡されるオブジェクト群は、イベント処理メソッドに対応するフェーズで新たに生じたタッチ、あるいは前のフェーズから状態が変化したタッチを表します。どのタッチオブジェクトもそうであれば、NSSetオブジェクトにanyObjectメッセージを送信して構いません。これは、ビューがマルチタッチシーケンスの最初のタッチだけを受け取る（すなわち、multipleTouchEnabledプロパティがNOである）場合です。

UITouchの重要なメソッドとしてlocationInView:があります。これにパラメータとしてselfを渡すと、受信したビューの座標系で表したタッチ位置を取得できます。同様のメソッドによって、そのタッチの以前の位置を取得できます（previousLocationInView:）。UITouchインスタンスのプロパティによって、タップの回数(tapCount)、そのタッチが作成また最後に変化した時点(timestamp)、およびそのタッチがどのフェーズ(phase)に属しているかがわかります。

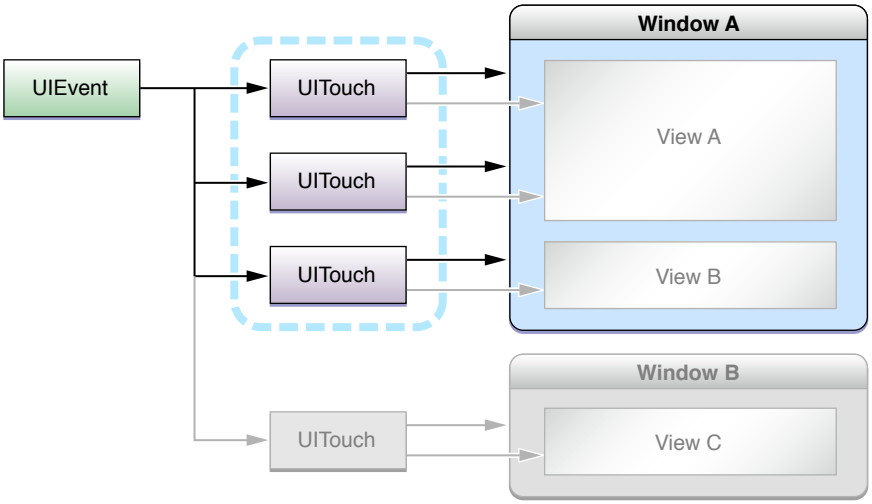
何らかの理由で、現在のマルチタッチシーケンス内のタッチのうち、最終フェーズ以降変化していないもの、あるいは渡されたセットのフェーズ以外に属するものを調べたい場合は、渡されたUIEventオブジェクトに問い合わせてください。図 2-3に、あるUIEvent オブジェクトの状態を示します。ここにはタッチオブジェクトが4つ含まれています。このタッチオブジェクトをすべて取得したい場合は、イベントオブジェクトのallTouchesメソッドを実行します。

図 2-3　あるタッチイベントに関連するすべてのタッチ



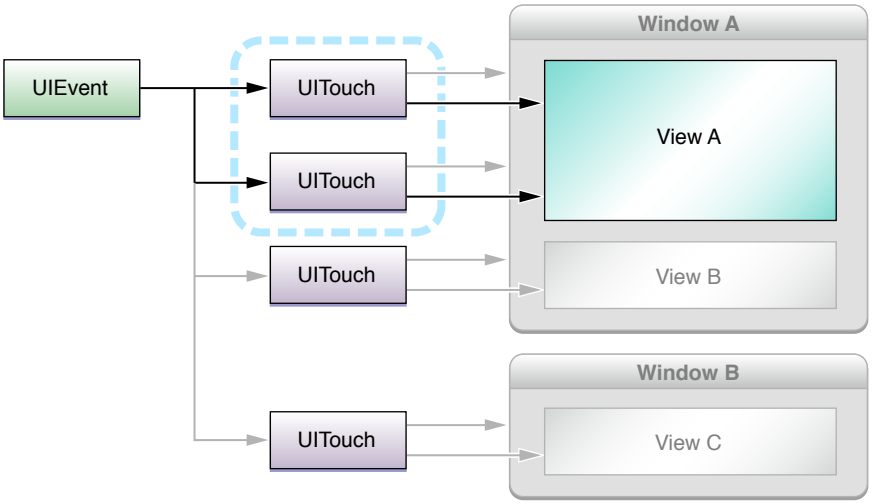
一方、特定のウインドウ（たとえば図2-4の「ウインドウA」）に属するタッチのみ取得したい場合は、UIEventオブジェクトに、touchesForWindow:メッセージを送信してください。

図 2-4 特定のウインドウに属するすべてのタッチ



特定のビューに属するタッチのみ取得したい場合は、イベントオブジェクトの`touchesForView:`メソッドを、ビューオブジェクト（たとえば図 2-5の「ビューA」）を引数として呼び出します。

図 2-5 特定のビューに属するすべてのタッチ



レスポンダがマルチタッチシーケンスでイベントを処理している間に永続オブジェクトを作成する場合は、システムがそのシーケンスをキャンセルしたときに、そのオブジェクトを破棄できるように、`touchesCancelled:withEvent:`を実装する必要があります。通常は、外部からのイベント（たとえば、電話の着信）によって現在のアプリケーションのイベント処理が中断されたときにキャンセルが発生します。レスポンダオブジェクトは、マルチタッチシーケンスの最後の`touchesEnded:withEvent:`を受け取ったときにも、これらのオブジェクトを破棄しなければならない点に注意してください（マルチタッチシーケンス内の最後の`UITouchPhaseEnded`タッチオブジェクトを判断する方法については、「タッチイベントの転送」（31 ページ）を参照してください）。

重要： カスタムレスポンダクラスが`UIView`または`UIViewController`のサブクラスであれば、「イベント処理メソッド」（21 ページ）に示したメソッドをすべて実装しなければなりません。一方、ほかのUIKitレスポンダクラスのサブクラスであれば、イベント処理メソッドをすべてオーバーライドしなくても構いませんが、オーバーライドするのであれば、その中から、当該メソッドのスーパークラスにおける実装を呼び出す必要があります（たとえば`super touchesBegan:touches withEvent:theEvent];`のように）。このガイドラインを設けている理由は単純です。タッチ操作を処理するビューは、独自に作成したものを含めすべて、タッチイベントストリーム全体を受け取ります（そう想定しなければなりません）。UIKitのレスポンダオブジェクトが、イベント中のあるフェーズのタッチを受け取れなければ、その場合の振る舞いは未定義になります。これはおそらく、望ましいものではないでしょう。

タップジェスチャの処理

iOSアプリケーションでよく使われるジェスチャはタップです。タップは、ユーザが自分の指で、画面上のオブジェクトを軽くたたくことを指します。レスポンダオブジェクトは、シングルタップ、ダブルタップ、場合によってはトリプルタップを、それぞれ別の方法で処理できます。ユーザがレスポンダオブジェクトをタップした回数を判断するには、`UITouch`オブジェクトの`tapCount`プロパティの値を取得します。

この値を調べるのに最適な場所は、`touchesBegan:withEvent:`メソッドと`touchesEnded:withEvent:`メソッドです。多くの場合、ユーザがタップして指を離したときのタッチフェーズに対応しているため、後者のメソッドの方が適しています。このタッチアップフェーズ（`UITouchPhaseEnded`）でのタップ回数を調べることによって、たとえば、指が画面に触れた後でドラッグしているのではなく、指が本当にタップしていることを確認できます。

リスト 2-1に、ビューのいずれかにダブルタップが発生したかを検出する方法を示します。

リスト 2-1　ダブルタップジェスチャの検出

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    for (UITouch *touch in touches) {
        if (touch.tapCount >= 2) {
            [self.superview bringSubviewToFront:self];
        }
    }
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
}
```

難しいのはレスポンダオブジェクトで、シングルタップおよびダブルタップジェスチャを異なる方法で処理する場合です。たとえば、シングルタップでオブジェクトを選択し、ダブルタップで、ダブルタップされた項目を編集するためのビューを表示するようなケースです。レスポンダオブジェクトで、シングルタップと、ダブルタップの最初のタップをどうやって区別すればよいのでしょう。リスト 2-2（25 ページ）に、イベント処理メソッドの実装例を示します。ダブルタップまたはシングルタップのジェスチャに応じて、ビューを拡大または縮小する、という処理を行います。

このコードについて説明します。

1. touchesEnded:withEvent:で、タップ回数が1の場合、レスポンダオブジェクトは自分自身にperformSelector:withObject:afterDelay:メッセージを送ります。セレクタは、このシングルタップジェスチャを処理するために、このレスポンダによって実装されている別のメソッドを識別します。2番目のパラメータは、それに関連するUITouchオブジェクトの状態を格納するNSValueオブジェクトまたはNSDictionaryオブジェクトです。遅延は、シングルタップジェスチャとダブルタップジェスチャの間の妥当な間隔にします。

   注： タッチオブジェクトの中身は、マルチタッチシーケンスを処理する過程で変化するので、タッチを保持し、その状態がずっと変わらないと想定することはできません（タッチオブジェクトをコピーすることもできません。UITouchはNSCopyingプロトコルを採用していないからです）。したがって、タッチオブジェクトの状態を保持しておきたい場合は、NSValueオブジェクト、ディクショナリなどに、必要な状態データを保存してください（リスト 2-2のコードでは、タッチ位置情報をディクショナリに保存しています。もっとも、これは単にコード例として示しただけで、情報を後で使ってはいません）。

2. touchesBegan:withEvent:で、タップ回数が2の場合、レスポンダオブジェクトはNSObjectのcancelPreviousPerformRequestsWithTarget:メソッドを、自分自身を引数として呼び出し、保留中の遅延実行呼び出しをキャンセルします。タップ回数が2でない場合は、前のステップでシングルタップジェスチャ用としてセレクタによって識別されたメソッドが遅延の後に実行されます。

3. touchesEnded:withEvent:で、タップ回数が2の場合は、レスポンダはダブルタップジェスチャを処理するために必要なアクションを実行します。

リスト 2-2　シングルタップ/ダブルタップジェスチャの処理

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *aTouch = [touches anyObject];
    if (aTouch.tapCount == 2) {
        [NSObject cancelPreviousPerformRequestsWithTarget:self];
    }
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *theTouch = [touches anyObject];
    if (theTouch.tapCount == 1) {
        NSDictionary *touchLoc = [NSDictionary dictionaryWithObject:
            [NSValue valueWithCGPoint:[theTouch locationInView:self]]
forKey:@"location"];
        [self performSelector:@selector(handleSingleTap:) withObject:touchLoc
afterDelay:0.3];
    } else if (theTouch.tapCount == 2) {
        // ダブルタップ: 画像を10%拡大
        CGRect myFrame = self.frame;
        myFrame.size.width += self.frame.size.width * 0.1;
        myFrame.size.height += self.frame.size.height * 0.1;
        myFrame.origin.x -= (self.frame.origin.x * 0.1) / 2.0;
        myFrame.origin.y -= (self.frame.origin.y * 0.1) / 2.0;
        [UIView beginAnimations:nil context:NULL];
```

```
        [self setFrame:myFrame];
        [UIView commitAnimations];
    }
}

- (void)handleSingleTap:(NSDictionary *)touches {
    // シングルタップ：画像を10%縮小
    CGRect myFrame = self.frame;
    myFrame.size.width -= self.frame.size.width * 0.1;
    myFrame.size.height -= self.frame.size.height * 0.1;
    myFrame.origin.x += (self.frame.origin.x * 0.1) / 2.0;
    myFrame.origin.y += (self.frame.origin.y * 0.1) / 2.0;
    [UIView beginAnimations:nil context:NULL];
    [self setFrame:myFrame];
    [UIView commitAnimations];
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    /* クリーンアップの状態ではないので、nullを実装 */
}
```

スワイプジェスチャ、ドラッグジェスチャの処理

水平および垂直のスワイプは、コードの中で簡単に追跡してアクションに使用できる単純なジェスチャです。スワイプジェスチャを検出するには、望みの移動軸に沿ってユーザの指の動きを追跡する必要がありますが、何がスワイプを構成するのかを決めるのはデベロッパ自身です。つまり、ユーザの指が十分な距離を移動したか、きちんと直線的に移動したか、また十分な速度で移動したかを判断する必要があります。これは、最初のタッチの場所を保存し、それを引き続くタッチ移動イベントで報告される位置と比較して行います。

リスト 2-3に、ビュー内での水平スワイプを検出するために使用できる基本的な追跡手法を示しています。この例では、ビューが、タッチの開始位置を`startTouchPosition`インスタンス変数に保存します。ユーザの指の移動に伴って、コードはタッチの現在位置と開始位置とを比較してそれがスワイプかどうかを判断します。タッチの垂直方向の移動が大きすぎる場合は、スワイプではないとみなされて別の処理が行われます。しかし、指が水平の軌道をたどっている場合は、それがスワイプであるとみなしてコードの処理が継続します。スワイプが水平方向に十分に移動してジェスチャが完了したと見なされたときに、処理ルーチンはアクションをトリガします。垂直方向のスワイプジェスチャを検出するには、xコンポーネントとyコンポーネントを入れ替えただけの同様のコードを使用します。

リスト 2-3　ビュー内のスワイプジェスチャの追跡

```
#define HORIZ_SWIPE_DRAG_MIN  12
#define VERT_SWIPE_DRAG_MAX    4

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    // startTouchPositionはインスタンス変数
    startTouchPosition = [touch locationInView:self];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    CGPoint currentTouchPosition = [touch locationInView:self];
```

```
    // スワイプであるためには、タッチの方向が水平で、十分な長さがなければならない
    if (fabsf(startTouchPosition.x - currentTouchPosition.x) >=
HORIZ_SWIPE_DRAG_MIN &&
        fabsf(startTouchPosition.y - currentTouchPosition.y) <=
VERT_SWIPE_DRAG_MAX)
    {
        // スワイプだとみなす
        if (startTouchPosition.x < currentTouchPosition.x)
            [self myProcessRightSwipe:touches withEvent:event];
        else
            [self myProcessLeftSwipe:touches withEvent:event];
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    startTouchPosition = CGPointZero;
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    startTouchPosition = CGPointZero;
}
```

リスト 2-4に、シングルタッチを処理する、より簡単な実装を示します。今回は、画面の周りにビューをドラッグする、という処理です。この例で、レスポンダクラスに完全に実装されているのはtouchesMoved:withEvent:メソッドだけです。ここでは、ビューにおけるタッチの現在位置と、直前の位置との差分を計算します。次に、この差分の値に基づき、ビューのフレームの原点をリセットしています。

リスト 2-4　シングルタッチによりビューをドラッグ

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *aTouch = [touches anyObject];
    CGPoint loc = [aTouch locationInView:self];
    CGPoint prevloc = [aTouch previousLocationInView:self];

    CGRect myFrame = self.frame;
    float deltaX = loc.x - prevloc.x;
    float deltaY = loc.y - prevloc.y;
    myFrame.origin.x += deltaX;
    myFrame.origin.y += deltaY;
    [self setFrame:myFrame];
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
}
```

複雑なマルチタッチシーケンスの処理

タップ、ドラッグ、スワイプはいずれも単純なジェスチャで、一般に、1回のタッチ操作しかありません。2回以上のタッチ操作を伴うタッチイベントは、より複雑な処理を必要とします。全フェーズを通してすべてのタッチを追跡して、変化のあったタッチ属性を記録し、内部状態を適切に変更しなければならない場合もあります。マルチタッチを追跡、処理するためには、いくつか必要な事項があります。

- ビューの `multipleTouchEnabled`プロパティを`YES`と設定します。
- Core Foundationのディクショナリオブジェクト（`CFDictionaryRef`）を使って、イベントの各フェーズを通して、タッチ状態の変化を追跡します。

複数回のタッチ操作を伴うイベントを処理する場合、各タッチの最初の状態を保存しておき、後で、変化した`UITouch`インスタンスと比較することが多くなります。たとえば、各タッチの終了位置を開始位置と比較する場合を考えます。`touchesBegan:withEvent:`メソッド内で、`locationInView:`メソッドから各タッチの開始位置を取得し、`UITouch`オブジェクトのアドレスをキーとして使用して`CFDictionaryRef`オブジェクトに格納できます。次に、`touchesEnded:withEvent:`メソッド内で、渡された各`UITouch`オブジェクトのアドレスを使用して、そのオブジェクトの開始位置を取得し現在位置と比較します（その際、`NSDictionary`オブジェクトではなく、`CFDictionaryRef`型を使用する必要があります。`NSDictionary`はキーをコピーしますが、`UITouch`クラスは、オブジェクトのコピーに必要な`NSCopying`プロトコルを採用しません）。

リスト 2-5に、`UITouch`オブジェクトの開始位置を、Core Foundationディクショナリに保存するコード例を示します。

リスト 2-5　マルチタッチの開始位置を保存

```
- (void)cacheBeginPointForTouches:(NSSet *)touches
{
    if ([touches count] > 0) {
        for (UITouch *touch in touches) {
            CGPoint *point = (CGPoint *)CFDictionaryGetValue(touchBeginPoints,
 touch);
            if (point == NULL) {
                point = (CGPoint *)malloc(sizeof(CGPoint));
                CFDictionarySetValue(touchBeginPoints, touch, point);
            }
            *point = [touch locationInView:view.superview];
        }
    }
}
```

リスト 2-6に、ディクショナリに保存した開始位置を検索するコード例を示します。ここでは同じタッチの、現在の位置も取得しています。この値を使ってアフィン変換を施します（コードは省略）。

リスト 2-6　タッチオブジェクトの開始位置を検索

```
- (CGAffineTransform)incrementalTransformWithTouches:(NSSet *)touches {
   NSArray *sortedTouches = [[touches allObjects]
sortedArrayUsingSelector:@selector(compareAddress:)];
```

```
    // その他のコードをここに入れる...

    UITouch *touch1 = [sortedTouches objectAtIndex:0];
    UITouch *touch2 = [sortedTouches objectAtIndex:1];

    CGPoint beginPoint1 = *(CGPoint *)CFDictionaryGetValue(touchBeginPoints,
touch1);
    CGPoint currentPoint1 = [touch1 locationInView:view.superview];
    CGPoint beginPoint2 = *(CGPoint *)CFDictionaryGetValue(touchBeginPoints,
touch2);
    CGPoint currentPoint2 = [touch2 locationInView:view.superview];

    // アフィン変換の計算...
}
```

リスト 2-7のコード例では、ディクショナリを使ってタッチの変化を追跡してはいませんが、イベント中のマルチタッチも処理しています。UIViewカスタムオブジェクトがどのようにタッチに応答するかを示します。ここでは、指の動きに合わせて「Welcome」という掲示が画面の中を動き回る様子をアニメーション化し、ユーザのダブルタップジェスチャに応じて「Welcome」の言語を変化させることでタッチに対応します（このコード例は、『*MoveMe*』サンプルコードプロジェクトに由来します。このサンプルプロジェクトを参照することで、イベント処理のコンテキストについての理解を深めることができます）。

リスト 2-7　複雑なマルチタッチシーケンスの処理

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];
    // タッチがplacardビューの内側で生じた場合にのみplacardビューを移動する
    if ([touch view] != placardView) {
        // placardビューの外側でダブルタップが生じた場合、placardの表示文字列を更新する
        if ([touch tapCount] == 2) {
            [placardView setupNextDisplayString];
        }
        return;
    }
    // placardビューを拡大してから縮小することによって、placardビューの「脈動」を表す
    // UIViewの組み込みアニメーション機能を使用する
    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:0.5];
    CGAffineTransform transform = CGAffineTransformMakeScale(1.2, 1.2);
    placardView.transform = transform;
    [UIView commitAnimations];

    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:0.5];
    transform = CGAffineTransformMakeScale(1.1, 1.1);
    placardView.transform = transform;
    [UIView commitAnimations];

    // placardViewをタッチの下に移動する
    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:0.25];
    placardView.center = [self convertPoint:[touch locationInView:self]
fromView:placardView];
    [UIView commitAnimations];
}
```

```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];
    // タッチがplacardView内で生じた場合には、その位置までplacardViewを移動する
    if ([touch view] == placardView) {
        CGPoint location = [touch locationInView:self];
        location = [self convertPoint:location fromView:placardView];
        placardView.center = location;
        return;
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];
    // タッチがplacardView内で生じた場合には、中央に戻す
    if ([touch view] == placardView) {
        // インタラクティブ操作を無効にして、以降のタッチがアニメーションに影響しないように
する
        self.userInteractionEnabled = NO;
        [self animatePlacardViewToCenter];
        return;
    }
}
```

注： 一般に、処理対象のイベントに応答して自分自身を再描画するようなカスタムビューは、イベント処理メソッドの中では描画状態の設定だけを行い、すべての描画処理を`drawRect:`メソッドで実行するようにします。ビューコンテンツの描画の詳細については、『*View Programming Guide for iOS*』を参照してください。

マルチタッチシーケンスの最後の指がビューを離れたことを検出するには、渡された`UITouch`オブジェクトの数と、渡された`UIEvent`オブジェクトが管理しているビューのタッチ数を比較します。同じであれば、マルチタッチシーケンスはこれで終わりです。リスト 2-8に、そのコード例を示します。

リスト 2-8　マルチタッチシーケンスの最後のタッチが終わったことを検出する

```
- (void)touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event {
    if ([touches count] == [[event touchesForView:self] count]) {
        // 最後の指が離れた...
    }
}
```

すでに説明したように、引数**touches**には、ビューに関連づけられたタッチオブジェクトのうち、当該フェーズで新たに生成されたもの、変化したものがすべて含まれます。一方、`touchesForView:`から返されるタッチオブジェクトには、当該ビューに関連づけられたすべてのオブジェクトが含まれます。

ヒットテスト

カスタムレスポンダは、「ヒットテスト」という方法で、タッチ「下」にあるサブビューまたはそれ自身のサブレイヤを見つけ、イベントを適切に処理できます。この処理には、`UIView`の`hitTest:withEvent:`メソッド、または`CALayer`の`hitTest:`メソッド、あるいはこれをオーバーライドしたメソッドを使います。レスポンダは、イベントを転送する前に、ヒットテストを実施することがあります（「タッチイベントの転送」　(31 ページ)）を参照）。

注： hitTest:withEvent:メソッドとhitTest:メソッドの振る舞いには若干の違いがあります。

hitTest:withEvent:または hitTest:に渡された点がビューの境界外にあれば、単に無視されます。したがって、スーパービューの外部にあるサブビューは、タッチイベントを受け取らないことになります。

カスタムビューがサブビューを持つ場合は、タッチをサブビューレベルで処理するか、スーパービューレベルで処理するかを判断する必要があります。サブビューにおいて、touchesBegan:withEvent:、touchesEnded:withEvent:、またはtouchesMoved:withEvent:を実装してタッチを処理しない場合は、これらのメッセージはレスポンダチェーンに沿ってさかのぼり、スーパービューに送られます。ところが、マルチタップやマルチタッチは、それらが最初に発生したサブビューに関連付けられているため、スーパービューではこれらのタッチを受け取ることができません。すべての種類のタッチを確実に受け取れるようにするには、スーパービューでhitTest:withEvent:をオーバーライドして、サブビューではなく自分自身を返すようにします。

リスト2-9に、カスタムビューのレイヤ内にある「Info」画像がタップされたことを検出するコード例を示します。

リスト 2-9 ビューのCALayerオブジェクトのhitTest:を呼び出し

```
- (void)touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event {
    CGPoint location = [[touches anyObject] locationInView:self];
    CALayer *hitLayer = [[self layer] hitTest:[self convertPoint:location
fromView:nil]];

    if (hitLayer == infoImage) {
        [self displayInfo];
    }
}
```

リスト 2-10に、レスポンダのサブクラス（この場合はUIWindowのサブクラス）で、hitTest:withEvent:をオーバーライドしている例を示します。まず、スーパークラスの実装を使って、「ヒットテスト」ビューを取得します。次に、これが自分自身であれば、ビュー階層をたどって、最も下位のビューで置き換えます。

リスト 2-10 hitTest:withEvent:のオーバーライド

```
- (UIView*)hitTest:(CGPoint)point withEvent:(UIEvent *)event {
    UIView *hitView = [super hitTest:point withEvent:event];

    if (hitView == self)
        return [[self subviews] lastObject];
    else
        return hitView;

}
```

タッチイベントの転送

ある種のアプリケーションで使われる技法として、イベント転送というものがあります。タッチイベントの転送は、ほかのレスポンダオブジェクトのイベント処理メソッドを呼び出すことによって行います。効果的な技術ですが、利用に当たっては注意も必要です。UIKitフレームワークのクラス

は、その管轄外で起こったタッチを受信できる設計になっていません。プログラムの実装の都合上、UITouchオブジェクトは、タッチを処理するため、viewプロパティにフレームワークオブジェクトの参照を保持しているからです。アプリケーション中で、条件に応じてほかのレスポンダにタッチを転送する場合、転送先のレスポンダはすべて、UIViewのサブクラスのインスタンスでなければなりません。

たとえば、アプリケーションに、A、B、Cという3つのカスタムビューがあるとします。ユーザがビューAにタッチすると、アプリケーションのウインドウはこれが「ヒットテスト」ビューであると判断し、最初のタッチイベントをここに送ります。所定の条件に従い、ビューAはイベントをビューBまたはCに転送します。この場合、ビューA、B、Cは、この転送がなされたことを認識していなければならず、また、ビューB、Cは、自分自身の範囲外で起こったタッチを処理できなければなりません。

イベント転送を行うためには、多くの場合、タッチオブジェクトを分析して、どこに転送するべきか判断する必要があります。分析方法は、次のようにいくつか考えられます。

- 「オーバーレイ」ビュー（共通のスーパービューなど）が、ヒットテストの機会にイベントを遮断し、サブビューに転送する方法（「ヒットテスト」 （30 ページ）を参照）。
- UIWindowのカスタムサブクラスで、sendEvent:をオーバーライドする方法。この中でタッチを分析し、適切なレスポンダに転送する。オーバーライドしたメソッド内で、スーパークラスに実装されている sendEvent:を呼び出すようにしなければならない。
- タッチ分析が必要ないようにアプリケーションを設計する方法。

リスト 2-11に、2つ目の方法による実装例を示します。UIWindowのサブクラスで、sendEvent:をオーバーライドする方法です。この例では、タッチイベントの転送先オブジェクトはカスタム「ヘルパ」レスポンダで、対応するビュー上でアフィン変換を行います。

リスト 2-11　タッチイベントを「ヘルパ」レスポンダオブジェクトに転送

```
- (void)sendEvent:(UIEvent *)event
{
    for (TransformGesture *gesture in transformGestures) {
        // 該当するタッチをすべて、イベントから収集
        NSSet *touches = [gesture observedTouchesForEvent:event];
        NSMutableSet *began = nil;
        NSMutableSet *moved = nil;
        NSMutableSet *ended = nil;
        NSMutableSet *cancelled = nil;

        // タッチをフェーズ順に整列して、通常のイベントディスパッチと同様に処理できるように
する
        for(UITouch *touch in touches) {
            switch ([touch phase]) {
                case UITouchPhaseBegan:
                    if (!began) began = [NSMutableSet set];
                    [began addObject:touch];
                    break;
                case UITouchPhaseMoved:
                    if (!moved) moved = [NSMutableSet set];
                    [moved addObject:touch];
                    break;
                case UITouchPhaseEnded:
                    if (!ended) ended = [NSMutableSet set];
                    [ended addObject:touch];
```

```
                    break;
                case UITouchPhaseCancelled:
                    if (!cancelled) cancelled = [NSMutableSet set];
                    [cancelled addObject:touch];
                    break;
                default:
                    break;
            }
        }
        // タッチを処理するメソッドを呼び出し
        if (began)     [gesture touchesBegan:began withEvent:event];
        if (moved)     [gesture touchesMoved:moved withEvent:event];
        if (ended)     [gesture touchesEnded:ended withEvent:event];
        if (cancelled) [gesture touchesCancelled:cancelled withEvent:event];
    }
    [super sendEvent:event];
}
```

この例では、オーバーライドしているサブクラスが、タッチイベントストリームの整合性を保つために重要な処理をしています。すなわち、スーパークラスで実装されている、`sendEvent:`を呼び出しています。

UIKitのビューやコントロールのサブクラスでイベントを処理

イベント処理の振る舞いを変える、または拡張するために、UIKitフレームワークにおける、ビューやコントロール（`UIImageView`、`UISwitch`など）のサブクラスを作成する場合、次の点に注意してください。

- カスタムビューとは違い、イベント処理メソッドをすべてオーバーライドする必要はありません。
- イベント処理メソッドをオーバーライドする場合、その中から、スーパークラスの実装を必ず呼び出してください。
- イベントをUIKitフレームワークオブジェクトに転送することはできません。

マルチタッチイベント処理のベストプラクティス

イベント（タッチイベント、モーションイベントとも）を処理する際、推奨される技術やパターンがいくつかあります。

- イベント取り消しメソッドは必ず実装してください。

  このメソッドでは、ビューの状態を、現在処理していたマルチタッチシーケンス以前に戻し、イベント処理用に確保した一時リソースセットを解放しなければなりません。イベント取り消しメソッドを実装しないでおくと、ビューの状態が不安定になるおそれがあります。場合によっては、ほかのビューが取り消しメッセージを受け取ってしまうかも知れません。

- `UIView`、`UIViewController`、（ごく稀に）`UIResponder`のサブクラスでイベントを処理する場合:

- イベント処理メソッドはすべて実装してください（中身が空であっても）。
- スーパークラスに実装されたメソッドを呼び出さないでください。

- ほかのUIKitレスポンダクラスのサブクラスでイベントを処理する場合:
  - イベント処理メソッドをすべて実装する必要はありません。
  - しかし、実装する場合は、必ずスーパークラスの実装を呼び出してください。たとえば、次のように記述します。

    ```
    [super touchesBegan:theTouches withEvent:theEvent];
    ```

- UIKitフレームワークにおけるほかのレスポンダオブジェクトには、イベントを転送しないでください。

  イベントの転送先レスポンダは、`UIView`のサブクラスのインスタンスでなければなりません。さらに、どのオブジェクトも、転送されたイベントを適切に処理できること、また、タッチイベントの場合、自分自身の範囲外で起こったタッチも処理できることが必要です。

- イベントに応答して自分自身を再描画するようなカスタムビューは、イベント処理メソッドの中では描画状態の設定だけを行い、すべての描画処理を`drawRect:`メソッドで実行するようにします。

- レスポンダに明示的に（`nextResponder`を介して）イベントを送信するのではなく、スーパークラスの実装を呼び出し、UIKitがレスポンダチェーンをたどるようにしてください。

Gesture Recognizer

注： この章の内容は、『*iPad Programming Guide*』に記載されていたものです。特にiOS 4.0向けに改訂した事項はありません。

iOSアプリケーションは、主として、アプリケーションのユーザインターフェイス内のボタン、ツールバー、テーブルビューの行、その他のオブジェクトをユーザがタップしたときに生成されるイベントによって駆動されます。UIKitフレームワークのクラスは、これらのほとんどのオブジェクトに対するデフォルトのイベント処理動作を提供します。ただし、独自のイベント処理を実装しなければならないアプリケーションもあります（主に、カスタムビューを持つアプリケーション）。このようなアプリケーションでは、マルチタッチのシーケンスにおけるタッチオブジェクトの流れを解析して、ユーザの意図を判断しなければなりません。

ほとんどのイベント処理ビューは、ユーザが画面上で行う一般的なジェスチャ（トリプルタップ、タッチアンドホールド（「長押し」とも呼ばれる）、ピンチ、回転ジェスチャなど）を検出しようとします。マルチタッチイベントの流れをそのまま解析して、1つ以上のジェスチャを検出するコードは、通常は複雑です。iOS 3.2より前のバージョンでは、このようなコードを別のプロジェクトにコピーして適宜修正する方法以外に、コードを再利用できません。

iOS 3.2では、アプリケーションでのジェスチャ検出を支援するために、Gesture Recognizerが導入されました。このオブジェクトは、`UIGestureRecognizer`クラスを直接継承します。以降の各セクションでは、このオブジェクトの仕組み、使用法、およびアプリケーション間で再利用可能なカスタムGesture Recognizerの作成方法について説明します。

イベント処理を簡単にするGesture Recognizer

`UIGestureRecognizer`は、Gesture Recognizer具象サブクラス（単に、Gesture Recognizerと呼ぶ）のための抽象基底クラスです。`UIGestureRecognizer`クラスには、プログラムインターフェイスが定義されており、ジェスチャ認識動作の土台が実装されています。UIKitフレームワークでは、最も良く使われるジェスチャに対応する6つのGesture Recognizerを提供しています。それ以外のジェスチャについては、独自のGesture Recognizerを設計して実装できます（詳細については、「カスタムGesture Recognizerの作成」（45 ページ）を参照）。

認識可能なジェスチャ

UIKitフレームワークでは、表 3-1に示すジェスチャの認識をサポートしています。ここに挙げた各クラスは、`UIGestureRecognizer`を直接継承したサブクラスです。

表 3-1 UIKitフレームワークのGesture Recognizerクラスで認識可能なジェスチャ

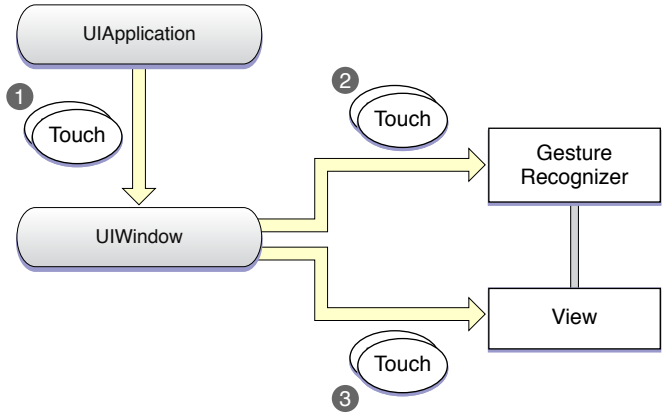
ジェスチャ	UIKitのクラス
タップ（タップ数は任意）	`UITapGestureRecognizer`
ピンチインとピンチアウト（ビューのズーム拡大縮小用）	`UIPinchGestureRecognizer`
パニング（ドラッグ）	`UIPanGestureRecognizer`
スワイプ（任意の方向に対応）	`UISwipeGestureRecognizer`
回転（互いに反対の方向への指の移動）	`UIRotationGestureRecognizer`
長押し（「タッチアンドホールド」とも呼ばれる）	`UILongPressGestureRecognizer`

Gesture Recognizerの使用を決定する前に、その使い方を検討してください。ユーザの想定に合った方法でジェスチャに応答するようにします。たとえば、ピンチジェスチャは、ビュー内容の拡大縮小に対応させるべきであって、たとえば選択要求として解釈するべきではありません。選択要求にはタップの方が適切です。適切なジェスチャの使用についてのガイドラインは、『*iOS Human Interface Guidelines*』を参照してください。

Gesture Recognizerをビューに添付する

ジェスチャを検出するには、ユーザがタッチするビューにGesture Recognizerをアタッチしなければなりません。このビューをヒットテストビューと呼びます。iOSのイベントは`UIEvent`オブジェクトで表現されることを思い出してください。また、各イベントオブジェクトは、現在のマルチタッチシーケンスを表す`UITouch`オブジェクトをカプセル化しています。これらの`UITouch`オブジェクトの集合は、特定のフェーズのマルチタッチシーケンスに固有のものです。イベントの配信は、最初は、通常の経路（オペレーティングシステムからアプリケーションオブジェクトを経て、タッチが発生しているウインドウを表すウインドウオブジェクトへ）をたどります。しかし、ヒットテストビューへのイベントの送信の前に、そのビューまたはそのビューのいずれかのサブビューにアタッチされているGesture Recognizerにイベントが送信されます。図 3-1に、この一般的な経路を示します。図中の番号は、タッチが受信される順番を表します。

図 3-1 Gesture Recognizerがビューにアタッチされている場合のタッチオブジェクトの経路



したがって、Gesture Recognizerは、対応するビューまたはビュー階層に送信されるタッチオブジェクトを監視する役割を果たします。ただし、Gesture Recognizerはそのビュー階層には含まれないため、レスポンダチェーンには加わりません。Gesture Recognizerは、ジェスチャの認識処理を行っている間、ビューへのタッチオブジェクトの配信を遅らせる場合があります。また、デフォルトでは、ジェスチャを認識すると、ビューへの残りのタッチオブジェクトの配信を取り消します。Gesture Recognizerからビューへのイベント配信で考えられるシナリオの詳細については、「ビューへのタッチ配信の制御」（43 ページ）を参照してください。

ジェスチャによっては、`UIGestureRecognizer`の`locationInView:`メソッドや`locationOfTouch:inView:`メソッドを利用して、クライアントが、対応するビューやサブビューでのジェスチャや特定のタッチの位置を検出できるものもあります。詳細については、「ジェスチャへの応答」（40 ページ）を参照してください。

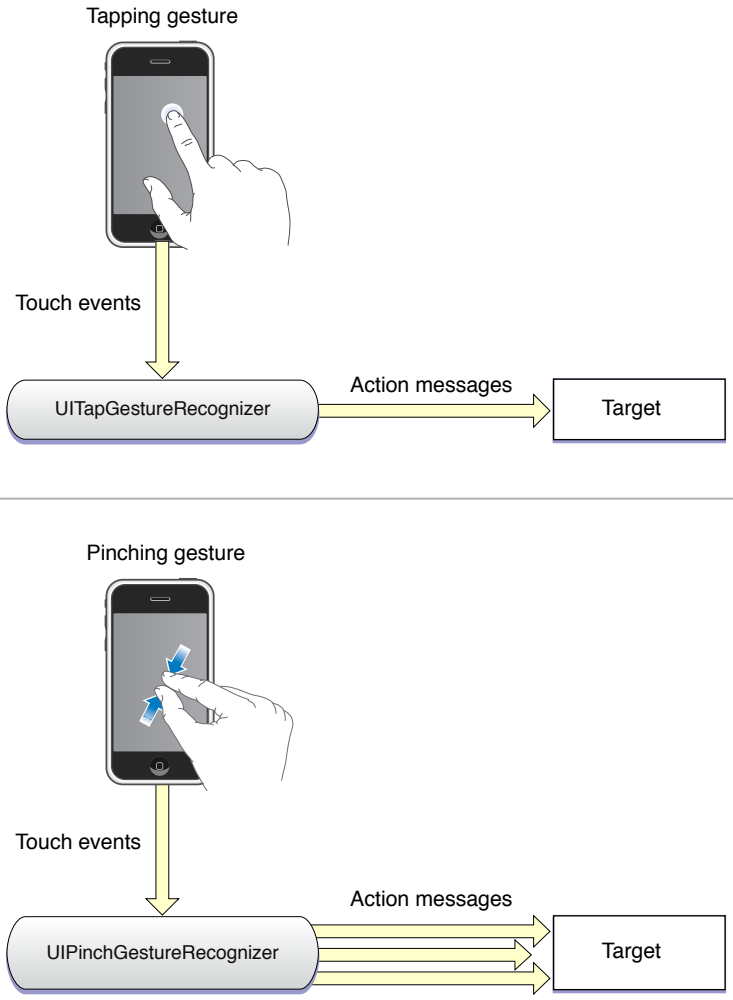
ジェスチャによってアクションメッセージが発行される

Gesture Recognizerは、対応するジェスチャを認識すると、1つ以上のターゲットに1つ以上のアクションメッセージを送信します。Gesture Recognizerを作成するときは、アクションとターゲットで初期化します。以後、ターゲット／アクションのペアを追加していけます。ターゲット／アクションのペアは加算的ではありません。つまり、アクションは、あらかじめ関連付けられていたターゲットにのみ送信され、それ以外のターゲットには送信されません（別のターゲット／アクションのペアで指定されていない限り）。

単発のジェスチャと連続的なジェスチャ

Gesture Recognizerは1つのジェスチャを認識すると、ターゲットに1つだけアクションメッセージを送信するか、そのジェスチャが終了するまで複数のアクションメッセージを送信するかのいずれかの動作をします。この動作は、ジェスチャが単発か連続的かによって決まります。単発のジェスチャ（ダブルタップなど）は1回だけ発生します。Gesture Recognizerは、単発のジェスチャを認識すると、ターゲットに1つだけアクションメッセージを送信します。連続的なジェスチャ（ピンチなど）は、一定時間にわたって発生し、ユーザがマルチタッチシーケンスでの最後の指を離したときに終了します。Gesture Recognizerは、マルチタッチシーケンスが終了するまで、短い間隔で複数のアクションメッセージをターゲットに送信します。

図 3-2 単発のジェスチャと連続的なジェスチャ



各Gesture Recognizerクラスのリファレンスドキュメントには、そのクラスのインスタンスが単発のジェスチャを検出するか、連続的なジェスチャを検出するかが明記されています。

ジェスチャ認識の実装

ジェスチャ認識を実装するには、Gesture Recognizerのインスタンスを作成して、ターゲットとアクションを割り当てます。また、場合によっては、ジェスチャ固有の属性も割り当てます。このGesture Recognizerオブジェクトをビューにアタッチしたら、ジェスチャを処理するターゲットオブジェクトにアクションメソッドを実装します。

Gesture Recognizerの準備

Gesture Recognizerを作成するには、UIGestureRecognizer具象サブクラスのインスタンスを割り当てて初期化しなければなりません。初期化するときには、次のコードのように、ターゲットオブジェクトとアクションセレクタを指定します。

```
UITapGestureRecognizer *doubleFingerDTap = [[UITapGestureRecognizer alloc]
   initWithTarget:self action:@selector(handleDoubleDoubleTap:)];
```

ジェスチャを処理するアクションメソッド（および、それを識別するセレクタ）は、次の2つのシグニチャのいずれかに従うことが前提になります。

```
- (void) handleGesture
- (void) handleGesture :(UIGestureRecognizer *) sender
```

*handleGesture*と*sender*には、任意の名前を使用できます。2番目のシグネチャを有するメソッドを利用すると、ターゲットがGesture Recognizerに補足情報を照会できるようになります。たとえば、UIPinchGestureRecognizerオブジェクトのターゲットは、ピンチジェスチャに関連する現在の倍率をそのGesture Recognizerオブジェクトに問い合わせることができます。

Gesture Recognizerを作成したら、UIViewのaddGestureRecognizer:メソッドを使用して、タッチを受け取るビュー（つまり、ヒットテストビュー）にそれをアタッチしなければなりません。現在ビューにアタッチされているGesture Recognizerは、gestureRecognizersプロパティを通じて知ることができます。また、removeGestureRecognizer:を呼び出して、Gesture Recognizerをビューからデタッチすることもできます。

リスト 3-1に示すメソッド例では、1本指によるダブルタップ、パニングジェスチャ、回転ジェスチャに対応する3つのGesture Recognizerの作成と初期化を行います。その後、それぞれのGesture Recognizerオブジェクトを同じビューにアタッチします。singleFingerDTapオブジェクトでは、そのジェスチャを認識するために2つのタップが必要であることを、コードで指定しています。それぞれのメソッドでは、作成したGesture Recognizerをビューに追加した後は、それを解放します（ビューがGesture Recognizerを保持しているため）。

リスト 3-1　単発ジェスチャおよび連続ジェスチャに対応するGesture Recognizerの作成と初期化

```
- (void)createGestureRecognizers {
    UITapGestureRecognizer *singleFingerDTap = [[UITapGestureRecognizer alloc]
        initWithTarget:self action:@selector(handleSingleDoubleTap:)];
    singleFingerDTap.numberOfTapsRequired = 2;
    [self.theView addGestureRecognizer:singleFingerDTap];
    [singleFingerDTap release];

    UIPanGestureRecognizer *panGesture = [[UIPanGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePanGesture:)];
    [self.theView addGestureRecognizer:panGesture];
    [panGesture release];

    UIPinchGestureRecognizer *pinchGesture = [[UIPinchGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePinchGesture:)];
    [self.theView addGestureRecognizer:pinchGesture];
    [pinchGesture release];
}
```

UIGestureRecognizerのaddTarget:action:メソッドを使用して、1つのGesture Recognizerに複数のターゲットとアクションを追加することもできます。各ターゲット／アクションペアに対応するアクションメッセージは、そのペアに限定されることを忘れないでください。複数のターゲット／アクションを持つ場合、それらは加算的ではありません。

ジェスチャへの応答

ジェスチャを処理するには、Gesture Recognizerのターゲットが、そのGesture Recognizerの初期化時に指定されたアクションセレクタに対応するメソッドを実装していなければなりません。タップジェスチャのような単発のジェスチャの場合、Gesture Recognizerは、ジェスチャを認識するたびにこのメソッドを呼び出します。一方、連続的なジェスチャの場合、Gesture Recognizerは、ジェスチャが終了する（つまり、Gesture Recognizerのビューから最後の指が離れる）まで一定間隔で繰り返しこのメソッドを呼び出します。

ジェスチャ処理メソッドでは、しばしばターゲットオブジェクトがGesture Recognizerからジェスチャについての補足情報を取得します。それには、Gesture Recognizerで定義されているプロパティ（scale（拡大縮小の倍率）、velocityなど）の値を取得します。また、（必要な場合は）Gesture Recognizerにジェスチャの位置を問い合わせることもできます。

リスト 3-2に、ピンチジェスチャ（handlePinchGesture:）とパニングジェスチャ（handlePanGesture:）の2つの連続的なジェスチャ用の処理を示します。また、単発のジェスチャ用の処理の例も示します。この例では、ユーザが1本の指でビューをダブルタップしたときに、ハンドラ（handleSingleDoubleTap:）はそのビューをダブルタップされた位置にセンタリングします。

リスト 3-2　ピンチ、パニング、ダブルタップの各ジェスチャの処理

```
- (IBAction)handlePinchGesture:(UIGestureRecognizer *)sender {
    CGFloat factor = [(UIPinchGestureRecognizer *)sender scale];
    self.view.transform = CGAffineTransformMakeScale(factor, factor);
}

- (IBAction)handlePanGesture:(UIPanGestureRecognizer *)sender {
    CGPoint translate = [sender translationInView:self.view];

    CGRect newFrame = currentImageFrame;
    newFrame.origin.x += translate.x;
    newFrame.origin.y += translate.y;
    sender.view.frame = newFrame;

    if (sender.state == UIGestureRecognizerStateEnded)
        currentImageFrame = newFrame;
}

- (IBAction)handleSingleDoubleTap:(UIGestureRecognizer *)sender {
    CGPoint tapPoint = [sender locationInView:sender.view.superview];
    [UIView beginAnimations:nil context:NULL];
    sender.view.center = tapPoint;
    [UIView commitAnimations];
}
```

これらのアクションメソッドは、次のような特有の方法でジェスチャを処理します。

- handlePinchGesture:メソッドでは、ターゲットがGesture Recognizer（sender）とやり取りをして拡大縮小の倍率（scale）を取得します。このメソッドは、Core Graphicsの関数にこの倍率値を使用します。この関数は、ビューを拡大縮小して、その計算結果をそのビューのアフィンtransformプロパティに割り当てます。
- handlePanGesture:メソッドは、Gesture Recognizerから取得したtranslationInView:値を対応するビューのキャッシュ済みのフレーム値に適用します。ジェスチャが終了したときには、最新のフレーム値がキャッシュされます。
- handleSingleDoubleTap:メソッドでは、ターゲットがlocationInView:メソッドを呼び出して、Gesture Recognizerからダブルタップジェスチャの位置を取得します。次に、この位置（スーパービューの座標に変換されている）を使用して、ビューの中心をダブルタップの位置にアニメーションで移動します。

handlePinchGesture:メソッドで取得した拡大縮小倍率は、回転角度やその他の連続的なジェスチャのGesture Recognizerに関連する変換値と同様に、ジェスチャが最初に認識された時点のビューの状態に適用されます。これは、特定のジェスチャのハンドラが呼び出されるたびに連結される差分値ではありません。

Gesture Recognizerがアタッチされているヒットテストビューは、タッチイベントが入ってきたときに、受け身である必要はありません。代わりに、gestureRecognizersプロパティを調べて、特定のUITouchオブジェクトに関与するGesture Recognizerを決定できます（対応するGesture Recognizerがあれば）。同様に、UIEventのtouchesForGestureRecognizer:メソッドを呼び出して、特定のGesture Recognizerが特定のイベントに対して解析中のタッチを知ることもできます。

ほかのGesture Recognizerとのやり取り

ビューには複数のGesture Recognizerがアタッチされている場合もあります。デフォルトの動作では、マルチタッチシーケンス内のタッチイベントは、最終的にビューに配信されるまでに、Gesture RecognizerからGesture Recognizerへ非決定的な順番で渡されます。通常は、このデフォルトの動作で十分です。しかし、次のような動作が必要になる場合もあります。

- 最初のGesture Recognizerがタッチイベントの解析に失敗すると、次のGesture Recognizerが解析を開始する。
- ほかのGesture Recognizerが、特定のマルチタッチシーケンスやそのシーケンス内のタッチオブジェクトを解析するのを禁止する。
- 2つのGesture Recognizerが同時に動作できるようにする。

UIGestureRecognizerクラスでは、クライアントのメソッド、デリゲートのメソッド、およびサブクラスでオーバーライドしたメソッドを提供することで、これらの動作を可能にします。

Gesture Recognizerの失敗を条件とする

2つのGesture Recognizerの間に、一方が失敗した場合にだけ、もう一方が動作可能になるような関係を持たせたい場合があります。たとえば、Gesture Recognizer Aは、Gesture Recognizer Bが失敗するまでマルチタッチシーケンスの解析を開始しないようにする場合などです。逆に、Gesture Recognizer Bがジェスチャの認識に成功した場合は、Gesture Recognizer Aはそのマルチタッチシーケンスを解析

しません。このような関係を指定する場合の例としては、シングルタップ用のGesture Recognizerとダブルタップ用のGesture Recognizerを1つずつ持つ場合が考えられます。シングルタップ用のGesture Recognizerは、ダブルタップ用のGesture Recognizerが失敗するまでは、マルチタッチシーケンスの解析を開始しません。

このような関係を指定するために呼び出すメソッドが、`requireGestureRecognizerToFail:`です。このメッセージを送信すると、それを受信したGesture Recognizerは、指定されたGesture Recognizerが`UIGestureRecognizerStateFailed`に遷移するまでは、`UIGestureRecognizerStatePossible`状態に留まっていなければなりません。指定されたGesture Recognizerが`UIGestureRecognizerStateRecognized`または`UIGestureRecognizerStateBegan`に遷移した場合は、処理を進めることができますが、ジェスチャを認識してもアクションメッセージは送信されません。

注： シングルタップジェスチャとダブルタップジェスチャの認識について考えてみましょう。シングルタップのGesture Recognizerが、ダブルタップのRecognizerが「認識失敗」と判断するのを待つことなく、自分自身の処理を進めてしまう設定になっているとします。すると、ダブルタップの操作をしても、（その1回目のタップ操作の時点で）シングルタップに応じたアクションが起こり、その後で、ダブルタップに応じたアクションが起こる、と想定していなければなりません。これは当然想定しておくべき動作であり、ある意味で望ましい振る舞いともいえます。ユーザにとっては、（1回、2回とタップ操作を繰り返すのに応じて）アクションが「積み上がって」いって欲しいからです。2つのGesture Recognizerに排他的なアクションをさせたいのであれば、シングルタップの操作をしたとき、（2回目のタップ操作がなかったことを認識した時点で）ダブルタップのGesture Recognizerが失敗するようにしなければなりません。逆に、ダブルタップの操作をしたときには、シングルタップに応じたアクションは起こらないことになります。しかしながら、シングルタップの操作をしたとき、タッチ入力後、（ダブルタップではないと判断するまで）若干の遅れが生じます。すなわち、ダブルタップかどうかは、（1回目のタップ操作から）所定の遅延時間が経過するまで判断できないので、シングルタップのGesture Recognizerはそれまでの間、対応するアクションを実行できないのです。

ジェスチャ認識の状態と、それらの状態間に起こり得る遷移の詳細については、「状態遷移」（45 ページ）を参照してください。

Gesture Recognizerのタッチ解析を禁止する

Gesture Recognizerが特定のタッチを解析するのを禁止したり、ジェスチャを認識するのを禁止できます。このような「禁止」関係は、デリゲーションのメソッド、または`UIGestureRecognizer`クラスで宣言されているメソッドのオーバーライドを使用して指定できます。

`UIGestureRecognizerDelegate`プロトコルには、ケースバイケースで特定のGesture Recognizerのジェスチャ認識を禁止する、次の2つのオプションメソッドが宣言されています。

- `gestureRecognizerShouldBegin:`—このメソッドは、Gesture Recognizerが`UIGestureRecognizerStatePossible`から出ようとするときに呼び出されます。`UIGestureRecognizerStateFailed`に遷移させるには、`NO`を返します（デフォルト値は`YES`です）。

- `gestureRecognizer:shouldReceiveTouch:`—このメソッドは、1つ以上の新規のタッチが発生したときに、ウインドウオブジェクトがGesture Recognizerの`touchesBegan:withEvent:`を呼び出す前に呼び出されます。Gesture Recognizerがこれらのタッチを表すオブジェクトを解析するのを禁止するには`NO`を返します（デフォルト値は`YES`です）。

さらに、これらのデリゲーションメソッドと同じ振る舞いをする2つの UIGestureRecognizer メソッド（UIGestureRecognizerSubclass.hで宣言されている）があります。サブクラスでこれらのメソッドをオーバーライドすると、クラスレベルの禁止規則を定義できます。

```
- (BOOL)canPreventGestureRecognizer:(UIGestureRecognizer
*)preventedGestureRecognizer;
- (BOOL)canBePreventedByGestureRecognizer:(UIGestureRecognizer
*)preventingGestureRecognizer;
```

同時のジェスチャ認識を許可する

デフォルトでは、2つのGesture Recognizerが同時にジェスチャ認識を試みることはできません。しかし、UIGestureRecognizerDelegateプロトコルのオプションメソッドの gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:を実装することによって、この動作を変更できます。このメソッドは、それを受信したGesture Recognizerの認識が、指定されたGesture Recognizerの動作をブロックする（または、その逆の）場合に呼び出されます。両方のGesture Recognizerに同時にジェスチャ認識を許可する場合は、YESを返します。

注： YESを返すと、同時認識ができることは保証されますが、NOを返したからといって、同時認識が禁止される保証はありません。ほかのジェスチャのデリゲートがYESを返す場合があるからです。

ビューへのタッチ配信の制御

一般に、ウインドウは、UITouchオブジェクト（UIEventオブジェクトにカプセル化されている）をGesture Recognizerに配信してから、それに対応するヒットテストビューに配信します。ただし、ジェスチャが認識されるかどうかによって、この一般的な経路が回り道をしたり、行き止まりになったりします。アプリケーションのニーズに合うように、この配信経路を変更できます。

デフォルトのタッチイベント配信

デフォルトでは、マルチタッチシーケンス内のウインドウは、ヒットテストビューへの「Ended」フェーズのタッチオブジェクトの配信を遅らせます。そして、ジェスチャが認識された場合は、ビューへの現在のタッチオブジェクトの配信を禁止して、それまでにビューが受信したタッチオブジェクトを取り消します。実際の動作は、タッチオブジェクトのフェーズと、Gesture Recognizerがマルチタッチシーケンス内でそのジェスチャを認識できたかどうかによって決まります。

この動作をわかりやすく説明するために、2つのタッチ（つまり、2本の指）を伴う単発のジェスチャ用の架空のGesture Recognizerを考えます。タッチオブジェクトがシステムに入ると、それは、UIApplicationオブジェクトから、ヒットテストビューに対応するUIWindowオブジェクトに渡されます。ジェスチャが認識されると、次のようなシーケンスが発生します。

1. ウインドウは、「Began」フェーズ（UITouchPhaseBegan）の2つのタッチオブジェクトをGesture Recognizerに送信します。しかし、Gesture Recognizerはこのジェスチャを認識しません。ウインドウは、これと同じタッチを、このGesture Recognizerに対応するビューに送信します。

2. ウインドウは、「Moved」フェーズ（UITouchPhaseMoved）の2つのタッチオブジェクトをGesture Recognizerに送信します。しかし、Gesture Recognizerは依然としてジェスチャを認識しません。次に、ウインドウは、これらのタッチをGesture Recognizerに対応するビューに送信します。

3. ウインドウは、「Ended」フェーズ（UITouchPhaseEnded）の*1*つのタッチオブジェクトをGesture Recognizerに送信します。このタッチオブジェクトはジェスチャに関する十分な情報をもたらしませんが、ウインドウはそのオブジェクトをGesture Recognizerに対応するビューに渡さずに保留します。

4. ウインドウは、「Ended」フェーズのもう1つのタッチオブジェクトをGesture Recognizerに送信します。ここでようやくGesture Recognizerはジェスチャを認識します。その結果、Gesture Recognizerの状態はUIGestureRecognizerStateRecognizedに設定されます。最初の（唯一の）アクションメッセージが送信される直前に、ビューはtouchesCancelled:withEvent:を受信して、それまで（「Began」フェーズと「Moved」フェーズで）送信されたタッチオブジェクトを無効にします。「Ended」フェーズのタッチが取り消されます。

今度は、最後のステップのGesture Recognizerが、それまで解析してきたマルチタッチシーケンスは、自身が担当するジェスチャではないと判断したとします。その場合は、Gesture Recognizerの状態はUIGestureRecognizerStateFailedに設定されます。そして、ウインドウは「Ended」フェーズの2つのタッチオブジェクトを、touchesEnded:withEvent:メッセージとして、Gesture Recognizerに対応するビューに送信します。

連続的なジェスチャ用のGesture Recognizerも同様のシーケンスをたどります。ただし、タッチオブジェクトが「Ended」フェーズに到達する前に、ジェスチャを認識する可能性が高くなります。ジェスチャを認識すると、Gesture Recognizerの状態はUIGestureRecognizerStateBeganに設定されます。ウインドウは、マルチタッチシーケンス内のそれ以降のすべてのタッチオブジェクトを、Gesture Recognizerに対応するビューではなく、Gesture Recognizerに送信します。

注： ジェスチャ認識の状態と、それらの状態間に起こり得る遷移の詳細については、「状態遷移」（45 ページ）を参照してください。

ビューへのタッチ配信の操作

UIGestureRecognizerの3つのプロパティの値を変更することによって、ビューへのタッチオブジェクトのデフォルトの配信経路をある程度変更できます。これらのプロパティとデフォルトは次のとおりです。

cancelsTouchesInView（デフォルト値はYES）
delaysTouchesBegan（デフォルト値はNO）
delaysTouchesEnded（デフォルト値はYES）

これらのプロパティのデフォルト値を変更すると、動作に次のような違いが生じます。

- cancelsTouchesInViewをNOに設定する—認識されたジェスチャに含まれるタッチについて、touchesCancelled:withEvent:がビューに送信*されません*。その結果、対応するビューが「Began」フェーズや「Moved」フェーズでそれまでに受信したタッチオブジェクトは無効になりません。

- delaysTouchesBeganをYESに設定する—Gesture Recognizerがジェスチャを認識すると、そのジェスチャに含まれていたタッチオブジェクトが、対応するビューに配信されません。この設定は、UIScrollViewのdelaysContentTouchesプロパティで提供されるものと同様の動作を提

供します。スクロールビューの場合は、タッチが始まるとすぐにスクロールが始まるときは、そのスクロールビューオブジェクトのサブビューはタッチを受信しません。したがって、瞬時の視覚的なフィードバックはありません。ユーザインターフェイスの反応が鈍いように感じられやすくなるので、この設定には注意が必要です。

- `delaysTouchesEnded`を`NO`に設定する—タッチが終了した後にジェスチャを認識したGesture Recognizerは、ビューに対してそのタッチを取り消せなくします。たとえば、あるビューに、`numberOfTapsRequired`が2に設定された`UITapGestureRecognizer`オブジェクトがアタッチされている場合に、ユーザがそのビューをダブルタップしたとします。このプロパティが`NO`に設定されていると、このビューは、`touchesBegan:withEvent:`、`touchesEnded:withEvent:`、`touchesBegan:withEvent:`、`touchesCancelled:withEvent:`の順番でメッセージを受け取ります。このプロパティが`YES`に設定されていると、ビューは、`touchesBegan:withEvent:`、`touchesBegan:withEvent:`、`touchesCancelled:withEvent:`、`touchesCancelled:withEvent:`の順番でメッセージを受け取ります。このプロパティの目的は、Gesture Recognizerによって後から取り消される可能性があるタッチのために、ビューがアクションを実行しないようにすることです。

カスタムGesture Recognizerの作成

カスタムGesture Recognizerを作成する場合は、Gesture Recognizerの仕組みを十分に理解しておく必要があります。次のセクションでは、Gesture Recognizerのアーキテクチャ面の背景について説明します。また、その次のセクションでは、実際にGesture Recognizerを作成する際の詳細について説明します。

状態遷移

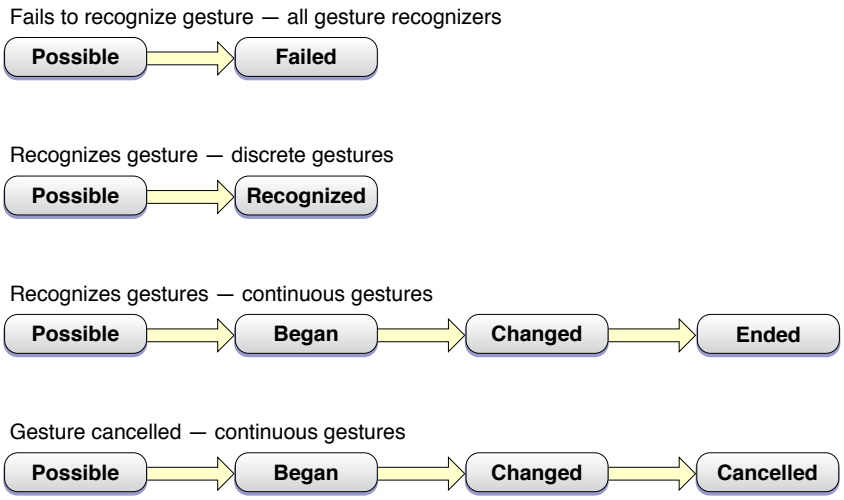
Gesture Recognizerは、定義済みの状態マシンとして動作します。Gesture Recognizerは、特定の条件が満たされるかどうかに応じて、ある状態から別の状態に遷移します。`UIGestureRecognizer.h`の次の`enum`定数は、Gesture Recognizerの状態を定義しています。

```
typedef enum {
    UIGestureRecognizerStatePossible,
    UIGestureRecognizerStateBegan,
    UIGestureRecognizerStateChanged,
    UIGestureRecognizerStateEnded,
    UIGestureRecognizerStateCancelled,
    UIGestureRecognizerStateFailed,
    UIGestureRecognizerStateRecognized = UIGestureRecognizerStateEnded
} UIGestureRecognizerState;
```

Gesture Recognizerが遷移する状態の順番は、認識されるジェスチャが単発ジェスチャか連続的ジェスチャかどうかに応じて異なります。すべてのGesture Recognizerは、Possible状態（`UIGestureRecognizerStatePossible`）から始まります。次に、Gesture Recognizerは、対応するヒットテストビューで発生したマルチタッチシーケンスを解析し、ジェスチャの認識に成功するか、失敗します。Gesture Recognizerがジェスチャを認識しなかった場合は、Failed状態（`UIGestureRecognizerStateFailed`）に遷移します。これは、ジェスチャが単発か連続的かに関係なく、すべてのGesture Recognizerに当てはまります。

一方、ジェスチャが認識された場合は、ジェスチャが単発か連続的によって、状態遷移が異なります。単発のジェスチャ用のGesture Recognizerは、PossibleからRecognized（`UIGestureRecognizerStateRecognized`）に遷移します。一方、連続的なジェスチャ用のGesture Recognizerは、最初にジェスチャを認識したときは、PossibleからBegan（`UIGestureRecognizerStateBegan`）に遷移します。次に、BeganからChanged（`UIGestureRecognizerStateChanged`）に遷移します。その後は、ジェスチャに変化があるたびにChangedからChangedに遷移します。最終的に、マルチタッチシーケンスの最後の指がヒットテストビューから離れたときに、Gesture RecognizerはEnded状態（`UIGestureRecognizerStateEnded`）に遷移します。この状態は、`UIGestureRecognizerStateRecognized`状態の別名です。連続的なジェスチャ用のGesture Recognizerは、認識中のジェスチャが、期待するジェスチャのパターンにこれ以上当てはまらないと判断したら、Changed状態からCancelled状態（`UIGestureRecognizerStateCancelled`）に遷移することもできます。図3-3に、これらの遷移を示します。

図 3-3 Gesture Recognizerで起こり得る状態遷移



注： Began、Changed、Ended、Cancelledの各状態は必ずしも同じタッチフェーズ内の`UITouch`オブジェクトに関連付けられているとは限りません。これらの状態は、認識中のタッチオブジェクトではなく、厳密にジェスチャ自体のフェーズを表します。

ジェスチャが認識されると、それ以降のすべての状態遷移によって、ターゲットにアクションメッセージが送信されます。Gesture RecognizerがRecognized（またはEnded）状態に達すると、新たなジェスチャ認識動作に備えるために、内部状態がリセットされます。そして、`UIGestureRecognizer`クラスはGesture Recognizerの状態をPossibleに戻します。

カスタムGesture Recognizerの実装

カスタムGesture Recognizerを実装するには、まず、Xcodeで`UIGestureRecognizer`のサブクラスを作成します。次に、このサブクラスのヘッダファイルに次のようなimportディレクティブを追加します。

```
#import <UIKit/UIGestureRecognizerSubclass.h>
```

そして、次のメソッド宣言をUIGestureRecognizerSubclass.hからこのサブクラスのヘッダファイルにコピーします。これらは、サブクラスでオーバーライドするメソッドです。

```
- (void)reset;
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event;
```

オーバーライドするすべてのメソッド内で、必ずスーパークラスの実装（super）を呼び出さなければなりません。

UIGestureRecognizerSubclass.hの stateプロパティの宣言を見てみましょう。（UIGestureRecognizer.hでは）readonlyだったものが、readwriteオプションになっている点に注目してください。サブクラスでは、このプロパティにUIGestureRecognizerState定数を割り当てることによって、状態を変更できるようになっています。

UIGestureRecognizerクラスは、自動的にアクションメッセージを送信して、ヒットテストビューへのタッチオブジェクトの配信を制御します。この作業を自分で実装する必要はありません。

マルチタッチ用のイベント処理メソッドの実装

Gesture Recognizerの実装の中心は、touchesBegan:withEvent:、touchesMoved:withEvent:、touchesEnded:withEvent:、およびtouchesCancelled:withEvent:の4つのメソッドです。これらのメソッドを実装する作業は、カスタムビュー用のメソッドを実装する場合と同様です。

注： マルチタッチシーケンスの間に配信されるイベントの処理の詳細については、「Document Revision History」の「Handling Multi-Touch Events」 in *iOS App Programming Guide*を参照してください。

Gesture Recognizer用のメソッドを実装する場合の主な違いは、適切なタイミングで状態を遷移させる点です。それには、stateプロパティの値を適切なUIGestureRecognizerState定数に設定しなければなりません。Gesture Recognizerが単発のジェスチャを認識した場合は、stateプロパティをUIGestureRecognizerStateRecognizedに設定します。ジェスチャが連続的な場合は、まず、stateプロパティをUIGestureRecognizerStateBeganに設定し、次に、ジェスチャの位置が変化するたびにstateプロパティをUIGestureRecognizerStateChangedに設定します（またはリセットします）。ジェスチャが終了したら、stateをUIGestureRecognizerStateEndedに設定します。Gesture Recognizerは、このマルチタッチシーケンスが自分の担当するジェスチャではないと判断した場合は、いつでもstateをUIGestureRecognizerStateFailedに設定できます。

リスト3-3は、単発のシングルタッチの「チェックマーク」ジェスチャ（実際には、任意のV字を描くジェスチャ）用のGesture Recognizerの実装です。このGesture Recognizerは、ジェスチャの中心点（上向きのストロークが始まる位置）の値をクライアントが取得できるようにその中心点を記録します。

リスト 3-3　「チェックマーク」Gesture Recognizerの実装

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesBegan:touches withEvent:event];
    if ([touches count] != 1) {
        self.state = UIGestureRecognizerStateFailed;
        return;
    }
```

```
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesMoved:touches withEvent:event];
    if (self.state == UIGestureRecognizerStateFailed) return;
    CGPoint nowPoint = [[touches anyObject] locationInView:self.view];
    CGPoint prevPoint = [[touches anyObject] previousLocationInView:self.view];
    if (!strokeUp) {
        // 下向きのストロークでは、xとyの両方が正の方向に増加する
        if (nowPoint.x >= prevPoint.x && nowPoint.y >= prevPoint.y) {
            self.midPoint = nowPoint;
            // 上向きのストロークでは、xの値は増加するが、yの値は減少する
        } else if (nowPoint.x >= prevPoint.x && nowPoint.y <= prevPoint.y) {
            strokeUp = YES;
        } else {
            self.state = UIGestureRecognizerStateFailed;
        }
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesEnded:touches withEvent:event];
    if ((self.state == UIGestureRecognizerStatePossible) && strokeUp) {
        self.state = UIGestureRecognizerStateRecognized;
    }
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesCancelled:touches withEvent:event];
    self.midPoint = CGPointZero;
    strokeUp = NO;
    self.state = UIGestureRecognizerStateFailed;
}
```

Gesture Recognizerは、自分の担当するジェスチャには含まれないと判断したタッチ（UITouchオブジェクトで表される）を検出した場合、それを直接ビューに渡すことができます。それには、自身のignoreTouch:forEvent:を呼び出して、そのタッチオブジェクトを渡します。無視されたタッチは、cancelsTouchesInViewプロパティの値がYESに設定されていても、対応するビューに渡されます。

状態のリセット

Gesture RecognizerがUIGestureRecognizerStateRecognized状態またはUIGestureRecognizerStateEnded状態に遷移すると、UIGestureRecognizerクラスは、Gesture Recognizerの状態がUIGestureRecognizerStatePossibleに戻る直前に、そのGesture Recognizerのresetメソッドを呼び出します。Gesture Recognizerクラスは、すべての内部状態をリセットして、新たなジェスチャを認識動作に備えるために、このメソッドを実装しなければなりません。Gesture Recognizerは、このメソッドから戻った後は、すでに始まっているがまだ終了していないタッチに関して更新を受け取らなくなります。

リスト 3-4　Gesture Recognizerのリセット

```
- (void)reset {
    [super reset];
    self.midPoint = CGPointZero;
    strokeUp = NO;
```

```
}
```

モーションイベント

iPhone、iPad、iPod touchデバイスは、ユーザが特定の方法（デバイスをシェイクする、傾けるなど）でデバイスを動かした場合にモーションイベントを生成します。モーションイベントの発生元はすべて、デバイスの加速度センサーまたはジャイロスコープです。

モーション（特にシェイクモーション）をジェスチャとして検出するためには、「「シェイク」モーションイベント」（51 ページ）のように、モーションイベントを処理する必要があります。高頻度で継続的にモーションデータを受信し、処理するためには、「Core Motion」（56 ページ）や「UIAccelerometerを介した加速度センサーイベントへのアクセス」（54 ページ）で説明する方針に従わなければなりません。

メモ： この章の内容は、『*iOS App Programming Guide*』に記載されていたものです。特にiOS 4.0向けに改訂した事項はありません。

「シェイク」モーションイベント

ユーザがデバイスを振り動かすと、システムは加速度センサーのデータを評価し、それが一定の基準を満たす場合に、シェイクジェスチャとして解釈します。システムは、このジェスチャを表す`UIEvent`オブジェクトを作成し、処理するために現在アクティブなアプリケーションに対してイベントオブジェクトを送信します。

注： モーションイベントは、`UIEvent`の一種として、iOS 3.0で導入されました。現在のところ、ジェスチャとして解釈されてモーションイベントになるのはシェイク動作だけです。

モーションイベントはタッチイベントよりもはるかに単純です。システムは、モーションの始まったときと止まったときをアプリケーションに通知し、個々のモーションが発生したタイミングは伝えません。さらに、タッチイベントには、タッチとそれに関連する情報のセットが含まれているのに対して、モーションイベントには、イベントタイプ、イベントサブタイプ、およびタイムスタンプ以外の状態は含まれていません。システムは、向きの変化と混同しないような方法でモーションジェスチャを解釈します。

モーションイベントを受け取るためには、これを処理するレスポンダオブジェクトがファーストレスポンダでなければなりません。リスト 4-1に、レスポンダが自分自身をファーストレスポンダにするコード例を示します。

リスト 4-1　ファーストレスポンダになる方法

```
- (BOOL)canBecomeFirstResponder {
    return YES;
}

- (void)viewDidAppear:(BOOL)animated {
    [self becomeFirstResponder];
```

```
}
```

モーションイベントを処理するには、UIResponder から派生したクラスに、motionBegan:withEvent:メソッドとmotionEnded:withEvent:メソッドのいずれか、または両方を実装する必要があります（「マルチタッチイベント処理のベストプラクティス」 （33 ページ）を参照）。たとえば、アプリケーションで水平シェイクと垂直シェイクに異なる意味を割り当てたい場合は、motionBegan:withEvent:で現在の加速度軸の値をキャッシュし、motionEnded:withEvent:でキャッシュされた値と同じ軸の値を比較して、その結果に応じた処理を実行します。また、レスポンダはmotionCancelled:withEvent:メソッドを実装して、システムがモーションイベントをキャンセルするために送信するイベントに対応する必要があります。これらのイベントはしばしば、モーションが結局は有効なジェスチャではなかったという、システムの判断を反映します。

リスト 4-2に、シェイクモーションイベントを処理するコード例を示します。（変換、回転、拡大縮小により）変化したビューを、元の位置、向き、大きさにリセットしています。

リスト 4-2　モーションイベントの処理

```
- (void)motionBegan:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
}

- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
    [UIView beginAnimations:nil context:nil];
    [UIView setAnimationDuration:0.5];
    self.view.transform = CGAffineTransformIdentity;

    for (UIView *subview in self.view.subviews) {
        subview.transform = CGAffineTransformIdentity;
    }
    [UIView commitAnimations];

    for (TransformGesture *gesture in [window allTransformGestures]) {
        [gesture resetTransform];
    }
}

- (void)motionCancelled:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
}
```

アプリケーションとそのキーウインドウは、モーションイベントをウインドウのファーストレスポンダに処理のために送信します。ファーストレスポンダがそれを処理しない場合は、タッチイベントと同様に、イベントはそれが処理されるか無視されるまでレスポンダチェーンをさかのぼります（詳細については「イベントの送信」 （12 ページ）を参照）。ただし、タッチイベントとシェイクモーションイベントには重要な違いが1つあります。ユーザがデバイスのシェイクを開始すると、システムはmotionBegan:withEvent:メッセージによってモーションイベントをファーストレスポンダに送信します。ファーストレスポンダがそのイベントを処理しない場合、そのメッセージはレスポンダチェーンをさかのぼります。シェイクの持続時間が約1秒未満の場合、システムはファーストレスポンダにmotionEnded:withEvent:メッセージを送信します。一方、シェイクの持続時間がそれよりも長い場合やシステムがそのモーションをシェイクだと判断しなかった場合は、ファーストレスポンダはmotionCancelled:withEvent:メッセージを受信します。

シェイクモーションイベントが処理されないままレスポンダチェーンをさかのぼってウインドウに到達したときに、UIApplication のapplicationSupportsShakeToEditプロパティがYESに設定されている場合、iOSは「取り消し（Undo）」コマンドと「やり直し（Redo）」コマンドを持つシートを表示します。デフォルトではこのプロパティはYESに設定されています。

現在のデバイスの向きの取得

向きの正確なベクトルではなく、単にデバイスの大まかな向きを知る必要がある場合、UIDeviceクラスのメソッドを使用してその情報を取得します。UIDeviceインターフェイスを使うのが簡単で、開発者自身が向きのベクトルを計算する必要がありません。

現在の向きを取得する前に、beginGeneratingDeviceOrientationNotificationsメソッドを呼び出すことによって、デバイス向きの通知の生成を開始するようにUIDeviceクラスに指示する必要があります。これを行うことで、加速度センサーハードウェアがオンになります。そうでなければ、節電のためにオフになっている可能性があります。

向きの通知を有効にしたらすぐ、UIDevice共有オブジェクトのorientationプロパティから現在の向きを取得できます。また、大まかな向きの変化が生じたるたびに送信される、UIDeviceOrientationDidChangeNotification通知の受け取りを登録することもできます。デバイスの向きは、デバイスが横長モードか縦長モードか、デバイスの表面が上に向いているか下に向いているかを示す、UIDeviceOrientationの定数を使ってレポートされます。これらの定数は、デバイスの物理的な向きを示し、アプリケーションのユーザインターフェイスの向きに対応する必要はありません。

デバイスの向きを知る必要がなくなったら、必ずUIDeviceのendGeneratingDeviceOrientationNotificationsメソッドを呼び出して、向きの通知を無効にするようにします。これを行うことで、システムはほかで加速度センサーハードウェアを使用していなければ、加速度センサーハードウェアを無効にできます。

加速度センサーやジャイロスコープのイベントを処理できるよう、必要なハードウェア能力を設定

アプリケーションの実行にデバイス関連の機能（加速度センサーのデータを受信するなど）が必要な場合は、必要な機能のリストをアプリケーションに追加する必要があります。宣言されている機能がデバイス上に存在する場合に限り、iOSは実行時にアプリケーションを起動します。さらに、App Store側では、ユーザのデバイス向けに要件リストを生成して、ユーザが自分のデバイスでは実行できないアプリケーションをダウンロードするのを防ぐために、この情報を使います。

UIRequiredDeviceCapabilitiesキーをアプリケーションの Info.plistファイルに宣言することによって、必要な機能のリストを追加します。iOS 3.0以降に導入されているこのキーの値は、配列または辞書のいずれかです。配列を使用する場合は、与えられたキーが存在すればそれに対応する機能が必要であることを表します。辞書を使用する場合は、その機能が必要かどうかを表すブール値をキーごとに指定しなければなりません。どちらの場合も、キーがなければその機能は必要ないことを示します。

次のUIRequiredDeviceCapabilitiesキーは、ハードウェアごとに、対応するモーションイベントが必要かどうかを表します。

- accelerometer（加速度センサーイベントの場合）

  アプリケーションでデバイスの向きの変化のみを検出する場合、あるいはアプリケーションがUIEventオブジェクト経由で送信されるシェイクモーションイベントを処理する場合は、このキーを含める必要はありません。

- gyroscope（ジャイロスコープイベントの場合）

UIAccelerometerを介した加速度センサーイベントへのアクセス

どのアプリケーションも、加速度データを受け取るために使用できるUIAccelerometerというシングルトンオブジェクトを持っています。このクラスのインスタンスを取得するには、UIAccelerometerクラスの sharedAccelerometerクラスメソッドを使用します。このオブジェクトを使用して、レポートの間隔や、加速度イベントを受け取るカスタムデリゲートを設定します。レポートの間隔は、最短で10ミリ秒（100Hzの更新レートに相当）に設定できます。ただし、ほとんどのアプリケーションはもっと長い間隔で十分に動作します。デリゲートオブジェクトを割り当てるとすぐに、加速度センサーはそのデリゲートオブジェクトにデータを送り始めます。その後、デリゲートオブジェクトは指定の更新間隔でデータを受け取ります。

リスト 4-3に、加速度センサーを設定するための基本的な手順を示します。この例では、更新頻度は50Hzです。これは、更新間隔20ミリ秒に相当します。myDelegateObjectはカスタムオブジェクトです。このオブジェクトは、加速度データを受け取る方法を定義するUIAccelerometerDelegateプロトコルをサポートする必要があります。

リスト 4-3　加速度センサーの設定

```
#define kAccelerometerFrequency        50.0 //Hz
-(void)configureAccelerometer
{
    UIAccelerometer*  theAccelerometer = [UIAccelerometer sharedAccelerometer];
    theAccelerometer.updateInterval = 1 / kAccelerometerFrequency;

    theAccelerometer.delegate = self;
    // デリゲートイベントがすぐに開始する
}
```

共有の加速度センサーオブジェクトは、一定の間隔でイベントデータをデリゲートのaccelerometer:didAccelerate:メソッドに送信します。その様子をリスト 4-4に示します。このメソッドを使用して、任意の方法で加速度データを処理できます。一般的には、何らかのフィルタを使用して、関心のあるデータ成分を分離する処理をお勧めします。

リスト 4-4　加速度センサーイベントの受け取り

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration
{
    UIAccelerationValue x, y, z;
    x = acceleration.x;
    y = acceleration.y;
    z = acceleration.z;
```

```
    // 受け取った値を使って何か処理をする
}
```

加速度イベントの送信を停止するには、`UIAccelerometer`共有オブジェクトのデリゲートに`nil`を設定します。デリゲートオブジェクトに`nil`を設定すると、システムに、必要に応じて加速度センサーハードウェアをオフにしてバッテリー持続時間を節約できることを知らせます。

デリゲートメソッドで受け取る加速度データは、加速度センサーハードウェアによってレポートされる瞬間的な値を表します。デバイスがたとえ完全に静止していても、ハードウェアからレポートされる値はわずかに変動する可能性があります。これらの値を使用する場合、時間の経過とともに値を平均化したり、受け取ったデータを補正したりして、必ずこの変動を計算に入れる必要があります。たとえば、Bubble Levelサンプルアプリケーションでは、既知の表面に対して現在の角度を補正するコントロールを提供しています。補正後の計測では、補正後の角度に対する相対角度がレポートされます。自分のコードで同じレベルの精度が必要な場合、そのユーザインターフェイスにも補正手段を含めるべきです。

適切な更新間隔の選択

加速度イベントの更新間隔を設定するとき、アプリケーションのニーズを満たしながら、送信されるイベントの数を最小限に抑える間隔を選択するのが最善です。1秒間に100回も加速度イベントの送信を必要とするアプリケーションはほとんどありません。低い頻度を採用することでアプリケーションが頻繁に動作せずに済むため、バッテリー持続時間を向上させることができます。表4-1に、一般的な更新頻度とその頻度で生成された加速度データを使って何ができるかを示します。

表 4-1　加速度イベントの一般的な更新間隔

イベント頻度（Hz）	使用方法
10–20	デバイスの現在の向きを表すベクトルを確認するのに適しています。
30–60	ゲーム、またはリアルタイムユーザ入力用に加速度センサーを使用するその他のアプリケーションに適しています。
70–100	高い頻度でモーションを検出する必要があるアプリケーションに適しています。たとえば、この間隔を使って、ユーザがデバイスをたたいたり高速でゆすったりすることを検出する場合が考えられます。

加速度データからの重力成分の分離

加速度センサーのデータを使用してデバイスの現在の向きを検出する場合は、デバイスの動きによる加速度データ成分から、重力による加速度データ成分を取り除く必要があります。それには、ローパスフィルタを使用して、加速度センサーのデータに対する瞬間的な変化の影響を減らすことができます。フィルタを適用した結果得られた値は、より安定した重力の影響を反映しています。

リスト 4-5に、簡単なローパスフィルタを示します。この例では、低い値のフィルタ係数を使用して、フィルタリングされていない加速度データの10%と、直前にフィルタリングした値の90%を使用した値を生成します。直前の値は、このクラスのメンバ変数である`accelX`、`accelY`、および`accelZ`に格納されています。加速度データは規則的になるので、これらの値はすぐに安定し、瞬間的な動きの変化に対する反応は鈍くなります。

リスト 4-5　加速度センサーのデータからの重力の影響の分離

```
#define kFilteringFactor 0.1

- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration {
    // 基本的なローパスフィルタを使用して、各軸の重力成分だけを保持する
    accelX = (acceleration.x * kFilteringFactor) + (accelX * (1.0 -
kFilteringFactor));
    accelY = (acceleration.y * kFilteringFactor) + (accelY * (1.0 -
kFilteringFactor));
    accelZ = (acceleration.z * kFilteringFactor) + (accelZ * (1.0 -
kFilteringFactor));

   // この加速度データを使用する
}
```

加速度データからの瞬間的な動きの分離

加速度センサーのデータを使用してデバイスの瞬間的な動きだけを検出する場合は、一定である重力の影響から瞬間的な動きの変化を分離する必要があります。それには、ハイパスフィルタを使用します。

リスト 4-6に、簡単なハイパスフィルタの計算を示します。直前のイベントの加速度値は、このクラスのメンバ変数である`accelX`、`accelY`、および`accelZ`に格納されています。この例では、瞬間的な動きの成分だけを取り出すために、ローパスフィルタの値を計算してそれを現在の値から引いています。

リスト 4-6　加速度センサーのデータからの瞬間的な動きの取得

```
#define kFilteringFactor 0.1

- (void)accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration {
    // ローパス値を現在の値から引いて、簡単なハイパスフィルタを取得する
    accelX = acceleration.x - ( (acceleration.x * kFilteringFactor) + (accelX
* (1.0 - kFilteringFactor)) );
    accelY = acceleration.y - ( (acceleration.y * kFilteringFactor) + (accelY
* (1.0 - kFilteringFactor)) );
    accelZ = acceleration.z - ( (acceleration.z * kFilteringFactor) + (accelZ
* (1.0 - kFilteringFactor)) );

   // この加速度データを使用する
}
```

Core Motion

Core Motionとは、モーションデータをデバイス上のセンサーから取得し、アプリケーションに渡して処理を委ねるシステムフレームワークのことです。センサーデータの処理と、関連するアルゴリズムを実装したアプリケーションは、Core Motion独自のスレッド上で実行されます。このモーションイベントを検出、送出するハードウェアコンポーネントとして、加速度センサーとジャイロス

コープがあります（ジャイロスコープはiPhone 4にのみ搭載）。Core MotionにはObjective-Cのプログラムインターフェイスがあります。アプリケーションはこれを介して、デバイスから各種のモーションデータを受け取り、適切な方法で処理することになります。

図 4-1 （57 ページ）に示すように、Core Motionにはマネージャクラスである`CMMotionManager`に加えて3つのクラスがあり、そのインスタンスは各種のモーションデータ（測定値）をカプセル化するようになっています。

- `CMAccelerometerData`オブジェクトは、3つの軸に沿った加速度を記録するデータ構造をカプセル化しています。データは加速度センサーから得られるものです。

  `CMAccelerometerData`について詳しくは、「Core Motionによる加速度センサーイベントの処理」 （58 ページ）を参照してください。

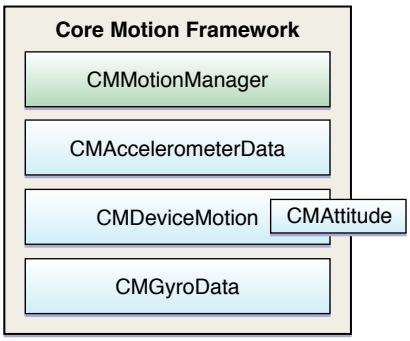
- `CMGyroData`オブジェクトは、3つの軸の周りの回転速度に「バイアス」をかけて評価し、その値を記録するデータ構造をカプセル化しています。「生の」データはジャイロスコープから得られたものです（「バイアス」とは、ここでは回転速度からのオフセットを表します。そのため、デバイスが回転していなくても、評価値は0になりません）。

  `CMGyroData`について詳しくは、「回転速度データの処理」 （61 ページ）を参照してください。

- `CMDeviceMotion`オブジェクトは、加速度センサーおよびジャイロスコープから得られた、処理済みのデバイスモーションデータをカプセル化します。Core Motionのセンサー融合アルゴリズムは、加速度センサーとジャイロスコープのデータを処理し、デバイスの姿勢、（バイアスを加えていない）回転速度、重力の方向、ユーザがデバイスに与えた加速度の、精密な測定値をアプリケーションに渡します。`CMAttitude`オブジェクトは、`CMDeviceMotion`のインスタンスに含まれており、ロール（回転角）、ピッチ（垂直角）、ヨー（水平角）を表すオイラ角をはじめ、姿勢に関する各種の測定値をプロパティとして保持しています。

  「姿勢」とは、デバイス外にある参照フレームを基準とした、3次元空間上のデバイスの向きを表します。姿勢および`CMDeviceMotion`について詳しくは、「加工済みのデバイスモーションデータの処理」 （64 ページ）を参照してください。

図 4-1 Core Motionクラス



Core Motionでは、データをカプセル化して保持するクラスはいずれも、`CMLogItem`のサブクラスになっています。これにはタイムスタンプが定義されており、モーションデータにイベント時刻をタグの形で添えてファイルに記録できます。アプリケーションでは、このタイムスタンプを以前のモーションイベントと比較して、イベント間の実際の更新間隔を判断できます。

上述のデータモーションタイプそれぞれについて、CMMotionManagerクラスには、モーションデータを取得する、プッシュ型とプル型という2種類の手段が組み込まれています。

- **プッシュ型。** アプリケーションは、更新間隔を指定し、モーションデータを処理する、（あるタイプの）ブロックを実装します。次いで、Core Motionにオペレーションキューとブロックを渡して、該当するタイプのモーションデータの更新を始めるよう指示します。Core Motionは（指定された更新間隔で）更新データをブロックに渡してオペレーションキューに入れます。ブロックはオペレーションキューから順次取り出され、タスクとして実行されることになります。
- **プル型。** アプリケーションは、あるタイプのモーションデータの更新を始めるよう指示し、定期的に、直近の測定値をサンプリングします。

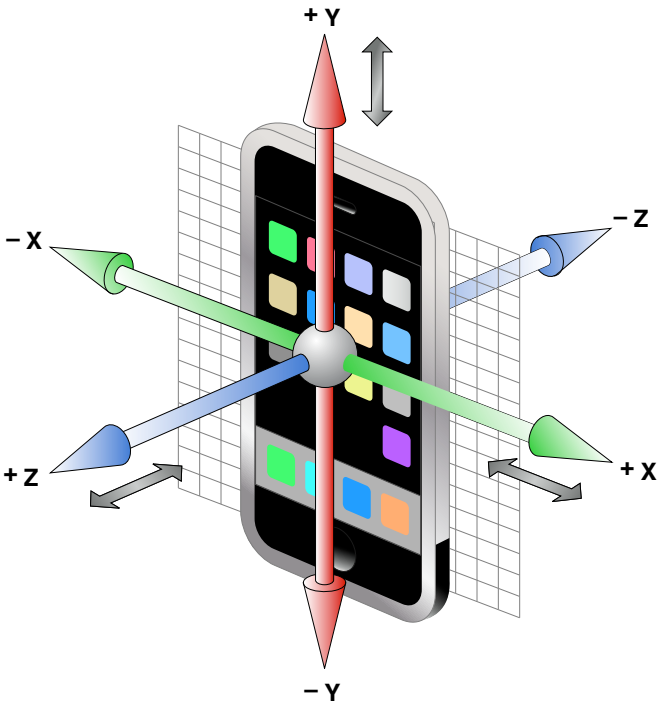
多くのアプリケーション、特にゲームでは、プル型を推奨します。一般に効率がよく、記述するコードも少なくて済みます。一方、プッシュ型は、データ収集アプリケーションなど、測定漏れがあっては困るものに向いています。どちらもスレッドセーフです。プッシュ型の場合、ブロックはオペレーションキューのスレッド上で動作します。一方、プル型の場合、Core Motionがスレッドを中断することはありません。

重要： アプリケーションでは、CMMotionManagerクラスのインスタンスをひとつだけ生成するようにしてください。複数のインスタンスがあると、アプリケーションが加速度センサーやジャイロスコープから受信するデータの速度に影響する場合があります。

Core Motionに関連するアプリケーション機能を、シミュレータ上で動かすことはできません。アプリケーションのテストやデバッグは、デバイス上で行う必要があります。

Core Motionによる加速度センサーイベントの処理

Core Motionには、加速度センサーイベントにアクセスする代替のプログラムインターフェイスとして、UIAccelerometerがあります。イベントはすべてCMAccelerometerDataのインスタンスであって、ここには加速度センサーの測定データが、CMAcceleration型のデータ構造としてカプセル化されています。



加速度センサーデータの受信と処理を始めるには、まずCMMotionManagerのインスタンスを生成し、次のいずれかのメソッドを呼び出してください。

- `startAccelerometerUpdates`

  このメソッドを呼び出すと、Core Motionは継続的に、CMMotionManagerのaccelerometerDataプロパティを、加速度センサーの直近の測定値で更新するようになります。アプリケーションは定期的に（ゲームなどでは通常、レンダリングループごとに）、このプロパティ値を取得することになります。このポーリング型の方針を採用する場合は、更新間隔を表すプロパティ（accelerometerUpdateInterval）に、最大でも何秒以内にCore Motionが更新処理をして欲しいか、を設定してください（Core Motionの能力としては、これより短い間隔で更新できるかも知れません）。

  このセクションで紹介したコード例は、この方針に基づいています。

- `startAccelerometerUpdatesToQueue:withHandler:`

  このメソッドを呼び出す前に、アプリケーションでは、accelerometerUpdateIntervalプロパティに更新間隔を設定しておきます。また、NSOperationQueueのインスタンスを生成し、さらに、CMAccelerometerHandler型のブロックに、加速度センサーのデータ更新を受けて実行する処理を実装します。こうしておいて、オペレーションキューとブロックを渡して、モーションマネージャオブジェクトのstartAccelerometerUpdatesToQueue:withHandler:を呼び出します。Core Motionは、指定された更新間隔で、加速度センサーの直近のデータをブロックに渡してキューに入れます。ブロックは、キューから順次取り出して実行されます。

アプリケーションがデータ処理を終えたら、ただちにモーションデータの更新を停止してください。するとCore Motionはモーションセンサーを停止するので、電池の消耗が抑えられます。

Core Motionの加速度センサーイベントに対しては、UIAccelerometerに設定したのと同じ更新間隔を設定してください。更新間隔はアプリケーションの処理内容に応じて決め、その値（秒単位）をaccelerometerUpdateIntervalプロパティに設定します。更新間隔を毎秒の回数（ヘルツ）として表す方が考えやすければ、その逆数を更新間隔の値としてください。リスト 4-3 （54 ページ）にその例があります(（「適切な更新間隔の選択」 （55 ページ）に、適切な更新間隔を決めるためのヒントを示します）。

次のコード例は、XcodeのOpenGL ESプロジェクトテンプレートがもとになっています。OpenGL ESアプリケーションは、ビューの描画用のレンダリングループを利用して定期的に、デバイスモーションの更新値を取得します。アプリケーションでは最初に、加速度の値を保持するインスタンス変数（3つの要素から成るCの配列）を宣言しています。

```
double filteredAcceleration[3];
```

リスト 4-7に示すように、アプリケーションはCMMotionManagerのインスタンスを、レンダリングループのタイミング機構を設定、スケジューリングするのと同じテンプレートメソッド（startAnimation）で生成します。次に、加速度センサーの適切な更新間隔をモーションマネージャに設定し、C配列のメモリ領域を確保した上で、加速度センサーの更新を開始します。なお、加速度センサーの更新を止める際も、レンダリングループのタイミング機構を無効にする際と同じテンプレートメソッド（stopAnimation）を使います。

リスト 4-7　モーションマネージャの設定と更新の開始

```
- (void)startAnimation {
    if (!animating) {
        // CADisplayLinkやタイマーを設定、スケジューリングするコードをここに置く...
    }
    motionManager = [[CMMotionManager alloc] init]; // motionManagerはインスタン
ス変数
    motionManager.accelerometerUpdateInterval = 0.01; // 100Hz
    memset(filteredAcceleration, 0, sizeof(filteredAcceleration));
    [motionManager startAccelerometerUpdates];
}

- (void)stopAnimation {
    if (animating) {
        // CADisplayLinkやタイマーを無効にするコードをここに置く ...
    }
    [motionManager stopAccelerometerUpdates];
}
```

OpenGL ESアプリケーションテンプレートでは、drawViewメソッドが、レンダリングループの各サイクルで呼び出されます。リスト4-8のコード例に示すように、アプリケーションはこの同じメソッド内で、直近の加速度センサーデータを取得し、ローパスフィルタを適用します。次に、フィルタ処理後の加速度値を用いて描画モデルを更新し、ビューを描画しています。

リスト 4-8　加速度データの取得とフィルタ処理

```
- (void)drawView {
    // alphaはフィルタ値（インスタンス変数）
    CMAccelerometerData *newestAccel = motionManager.accelerometerData;
    filteredAcceleration[0] = filteredAcceleration[0] * (1.0-alpha) +
newestAccel.acceleration.x * alpha;
    filteredAcceleration[1] = filteredAcceleration[1] * (1.0-alpha) +
newestAccel.acceleration.y * alpha;
```

```
    filteredAcceleration[2] = filteredAcceleration[2] * (1.0-alpha) +
newestAccel.acceleration.z * alpha;
    [self updateModelsWithAcceleration:filteredAcceleration];
    [renderer render];
}
```

注： 加速度値にローパスフィルタやハイパスフィルタを適用することにより、重力成分と、ユーザの操作による加速度の成分を分離できます。

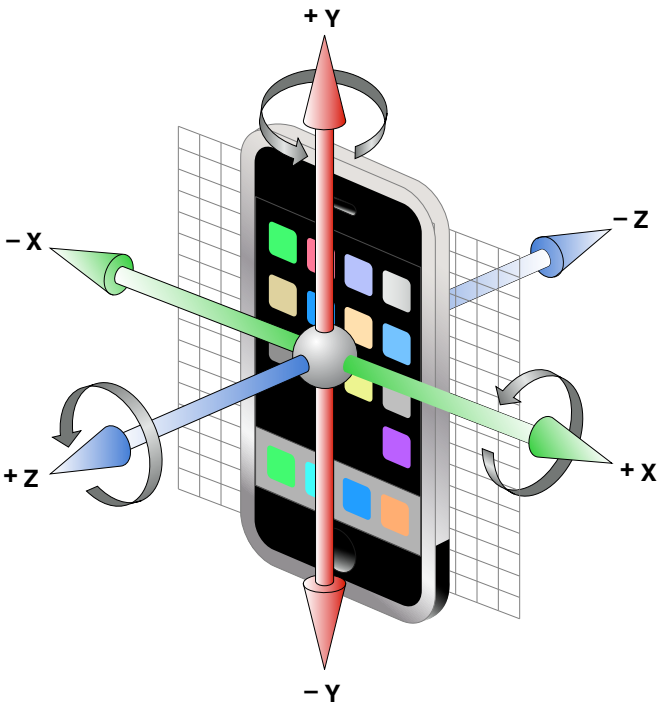
- ローパスフィルタを適用して重力成分を分離する方法については、「加速度データからの重力成分の分離」（55 ページ）を参照してください。
- ハイパスフィルタを適用して、ユーザの操作による加速度の成分を分離する方法については、「加速度データからの瞬間的な動きの分離」（56 ページ）を参照してください（ここではユーザの操作による加速度の成分を「瞬間的な動き」と呼んでいます）。

アプリケーションでは、加速度の代わりにデバイスモーションの更新値を受け取って処理することにより、加速度のうち重力に由来する成分、ユーザに由来する成分を、Core Motionから直接取得することも可能です。詳しくは「加工済みのデバイスモーションデータの処理」（64 ページ）を参照してください。

回転速度データの処理

ジャイロスコープは、3つの空間軸の周りの、デバイスの回転速度を測定します（これに対し、加速度センサーは、3つの空間軸に沿った*加速度*を測定するのでした）。ジャイロスコープの測定値の更新を求められるごとに、Core Motionは回転速度にバイアスをかけて評価し、`CMGyroData`オブジェクトの形でこの情報をアプリケーションに返します。このオブジェクトには`rotationRate`というプロパティがあります。これを通して、3つの軸の周りの回転速度（ラジアン/秒単位）を格納した、`CMRotationRate`というデータ構造にアクセスできます。

注： CMGyroDataオブジェクトにカプセル化されている回転速度の測定値は、バイアスを加えた値です。より正確な（バイアスを加えていない）測定値は、CMDeviceMotionオブジェクトのrotationRateプロパティを介して取得できます。



回転速度データの受信と処理を始めるには、まずCMMotionManagerクラスのインスタンスを生成し、次のいずれかのメソッドを呼び出してください。

- startGyroUpdates

  このメソッドを呼び出すと、Core Motionは継続的に、CMMotionManagerのgyroDataプロパティを、ジャイロスコープの直近の測定値で更新するようになります。アプリケーションは定期的に（ゲームなどでは通常、レンダリングループごとに）、このプロパティ値を取得することになります。このポーリング型の方針を採用する場合は、更新間隔を表すプロパティ（gyroUpdateInterval）に、最大でも何秒以内にCore Motionが更新処理をして欲しいか、を設定してください（Core Motionの能力としては、これより短い間隔で更新できるかも知れません）。

- startGyroUpdatesToQueue:withHandler:

  このメソッドを呼び出す前に、アプリケーションでは、gyroUpdateIntervalプロパティに更新間隔を設定しておきます。また、NSOperationQueueのインスタンスを生成し、さらに、CMGyroHandler型のブロックに、ジャイロスコープのデータ更新を受けて実行する処理を実装します。こうしておいて、オペレーションキューとブロックを渡して、モーションマネージャオブジェクトのstartGyroUpdatesToQueue:withHandler:を呼び出します。Core Motionは、指定された更新間隔で、ジャイロスコープの直近のデータをブロックに渡してキューに入れます。ブロックは、キューから順次取り出して実行されます。

このセクションで紹介したコード例は、この方針に基づいています。

アプリケーションがデータ処理を終えたら、ただちにモーションデータの更新を停止してください。するとCore Motionはモーションセンサーを停止するので、電池の消耗が抑えられます。

回転速度（ジャイロスコープ）イベントの更新間隔は、アプリケーションの処理内容に応じて決め、その値（秒単位）をgyroUpdateInterval プロパティに設定します。更新間隔を毎秒の回数（ヘルツ）として表す方が考えやすければ、その逆数を更新間隔の値としてください。「適切な更新間隔の選択」（55 ページ）のリスト 4-3（54 ページ）に、逆数を求めて更新間隔を設定する例を示します（これは加速度センサーの更新間隔ですが、考え方は同じです）。

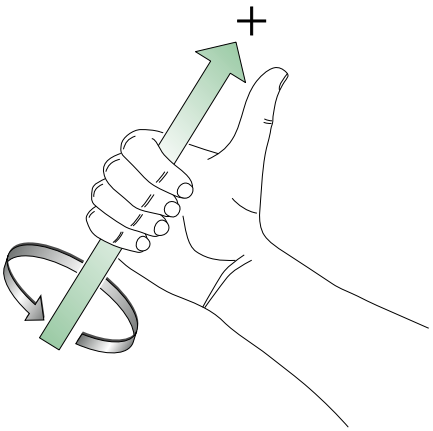
startGyroUpdatesToQueue:withHandler:メソッドを呼び出して、ジャイロスコープの更新を開始する例を示します。リスト 4-9に示すように、ビューコントローラはviewDidLoadで、CMMotionManagerオブジェクトのインスタンスを生成し、ジャイロスコープデータの更新間隔を設定しています。デバイスにジャイロスコープが組み込まれていれば、ビューコントローラはNSOperationQueueオブジェクトを生成し、ジャイロスコープデータの更新処理用のブロックハンドラを定義します。

リスト 4-9　CMMotionManagerオブジェクトを生成し、ジャイロスコープ更新用に設定

```
- (void)viewDidLoad {
    [super viewDidLoad];
    motionManager = [[CMMotionManager alloc] init];
    motionManager.gyroUpdateInterval = 1.0/60.0;
    if (motionManager.gyroAvailable) {
        opQ = [[NSOperationQueue currentQueue] retain];
        gyroHandler = ^ (CMGyroData *gyroData, NSError *error) {
            CMRotationRate rotate = gyroData.rotationRate;
            // 回転速度データをここで処理 ......
        };
    } else {
        NSLog(@"No gyroscope on device.");
        toggleButton.enabled = NO;
        [motionManager release];
    }
}
```

回転速度データ（すなわちCMRotationMatrixデータ構造の各フィールド）を分析する際、回転方向は「右手則」に従って判断します（図 4-2（64 ページ）を参照）。たとえばX軸の周りを右手で包むように摑み、親指の先がX軸の正の方向を指すようにすると、正の向きの回転は、ほかの4本の指先が指し示す方向になります。負の向きの回転は、この反対の方向です。

図 4-2　右手則



ユーザがボタンをタップすると、アクションメッセージがビューコントローラに送信されます。ビューコントローラには、更新開始と更新停止を交互に切り替えるアクションを実装します。リスト 4-10に実装例を示します。

リスト 4-10　ジャイロスコープの更新の開始と停止

```
- (IBAction)toggleGyroUpdates:(id)sender {
    if ([[(UIButton *)sender currentTitle] isEqualToString:@"Start"]) {
        [motionManager startGyroUpdatesToQueue:opQ withHandler:gyroHandler];
    } else {
        [motionManager stopGyroUpdates];
    }

}
```

加工済みのデバイスモーションデータの処理

デバイスに加速度センサーとジャイロスコープが組み込まれていれば、Core Motionは、この2つから「生」のモーションデータを読み取る、デバイスモーションサービスを提供します。このサービスは、センサー融合アルゴリズムにより生データを処理し、デバイスの姿勢、（バイアスを加えていない）回転速度、重力の方向、ユーザがデバイスに与えた加速度の、精密な測定値を生成します。`CMDeviceMotion`クラスのインスタンスには、このデータがカプセル化されています。

姿勢データには、`CMDeviceMotion`オブジェクトの`attitude`プロパティを介してアクセスできます。`CMAttitude`クラスのインスタンスには、姿勢の測定値がカプセル化されています。姿勢は次の3とおりの方法で表すようになっています。

- 四元数
- 回転行列
- 3つのオイラ角（ロール、ピッチ、ヨー）

デバイスモーションサービスは、加速度の重力成分とユーザ成分を分離して返すので、加速度データに改めてフィルタを適用する必要はありません。

デバイスモーションの更新データの受信と処理を始めるには、まずCMMotionManagerのインスタンスを生成し、次のいずれかのメソッドを呼び出してください。

- startDeviceMotionUpdates

  このメソッドを呼び出すと、Core Motionは継続的に、CMMotionManagerのdeviceMotionプロパティを、加速度センサーおよびジャイロスコープの直近の測定値で更新するようになります（CMDeviceMotionオブジェクトにカプセル化されているのと同様に）。アプリケーションは定期的に（ゲームなどでは通常、レンダリングループごとに）、このプロパティ値を取得することになります。このポーリング型の方針を採用する場合は、更新間隔を表すプロパティ（deviceMotionUpdateInterval）に、最大でも何秒以内にCore Motionが更新処理をして欲しいか、を設定してください（Core Motionの能力としては、これより短い間隔で更新できるかも知れません）。

  このセクションで紹介したコード例は、この方針に基づいています。

- startDeviceMotionUpdatesToQueue:withHandler:

  このメソッドを呼び出す前に、アプリケーションでは、deviceMotionUpdateIntervalプロパティに更新間隔を設定しておきます。また、NSOperationQueueのインスタンスを生成し、さらに、CMDeviceMotionHandler型のブロックに、加速度センサーのデータ更新を受けて実行する処理を実装します。こうしておいて、オペレーションキューとブロックを渡して、モーションマネージャオブジェクトのstartDeviceMotionUpdatesToQueue:withHandler:を呼び出します。Core Motionは、指定された更新間隔で、（CMDeviceMotionオブジェクトで表される）加速度センサーおよびジャイロスコープの直近のデータをブロックに渡してキューに入れます。ブロックは、キューから順次取り出して実行されます。

アプリケーションがデータ処理を終えたら、ただちにモーションデータの更新を停止してください。するとCore Motionはモーションセンサーを停止するので、電池の消耗が抑えられます。

デバイスモーションイベントの更新間隔は、アプリケーションの処理内容に応じて決め、その値（秒単位）をdeviceMotionUpdateIntervalプロパティに設定します。更新間隔を毎秒の回数（ヘルツ）として表す方が考えやすければ、その逆数を更新間隔の値としてください。「適切な更新間隔の選択」（55 ページ）のリスト 4-3（54 ページ）に、逆数を求めて更新間隔を設定する例を示します（これは加速度センサーの更新間隔ですが、考え方は同じです）。

デバイスモーションデータを処理する例

次のコード例は、XcodeのOpenGL ESプロジェクトテンプレートがもとになっています。OpenGL ESアプリケーションは、ビューの描画用のレンダリングループを利用して定期的に、デバイスモーションの更新値を取得します。リスト 4-11に示すように、アプリケーションはinitWithCoder:で、CMMotionManagerのインスタンスを生成し、このオブジェクトをインスタンス変数に代入します。さらに、デバイスモーションデータの最小更新間隔を指定しています。次にアプリケーションは、OpenGLビューのレンダリングループを利用して、デバイスモーションの更新処理を開始します。また、ループが無効になれば（ビューの描画が停まれば）、デバイスモーションの更新処理を停止します。

リスト 4-11　デバイスモーションの更新処理の開始と停止

```
- (id)initWithCoder:(NSCoder*)coder {
```

```
    if ((self = [super initWithCoder:coder])) {
        motionManager = [[CMMotionManager alloc] init];
        motionManager.deviceMotionUpdateInterval = 0.02; // 50 Hz
        // その他の初期化コードをここに入れる...
    }
}

- (void)startAnimation {
    if (!animating) {
        // CADisplayLinkやタイマーを設定、スケジューリングするコードをここに置く...
    }
   if ([motionManager.isDeviceMotionAvailable])
       [motionManager startDeviceMotionUpdates];
}


- (void)stopAnimation {
    if (animating) {
        // CADisplayLinkやタイマーを無効にするコードをここに置く ...
    }
    if ([motionManager.isDeviceMotionActive])
        [motionManager stopDeviceMotionUpdates];
}
```

なお、デバイスモーションサービスが使えない（おそらくジャイロスコープが組み込まれていない）場合は、加速度センサーデータを処理する方法でデバイスモーションに応答する、代替手段を実装するとよいかも知れません。

デバイスの姿勢と参照フレーム

CMDeviceMotionオブジェクトが提供する情報の中でも特に有用なのは、デバイスの姿勢に関するものでしょう。実用的な見地からは、デバイスの姿勢の*変化*の方が、より重要です。デバイスの姿勢、すなわち空間における方向は、参照フレームを基準として測定するようになっています。Core Motionは、アプリケーションがデバイスモーションの更新処理を開始した時点で、参照フレームを確定します。CMAttitudeのインスタンスは、この当初の参照フレームから、デバイスの現在の参照フレームに移るために必要な回転を求めます。Core Motionの参照フレームは、z軸が鉛直で、x軸とy軸が重力方向に対して直交するように選ばれます。したがって、Core Motionの参照フレームを基準とすると、重力方向は常にベクトル [0, 0, -1] で表されることになります。これを重力参照と言います。CMAttitudeオブジェクトから取得した回転行列を重力参照に掛けると、デバイスのフレームにおける重力方向が得られます。これを数式で表すと、次のようになります。

$$\texttt{deviceMotion.gravity} = R \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

CMAttitudeのインスタンスが基準として使う参照フレームは、変更可能です。そのためには、参照フレームを格納している姿勢オブジェクトをキャッシュしておき、これを引数として multiplyByInverseOfAttitude: に渡してください。このメッセージを受け取る姿勢引数は、新しい参照フレームを基準とした、姿勢の変化を表すようになります。

これが有用であることを例示するため、野球ゲームを考え、ユーザがバット代わりにデバイスを振ることを想定しましょう。通常、投球が始まった時点では、バットはある静止方向を向いています。その後、デバイスの姿勢が投球開始時からどのように変化したか、に応じた向きで、バットが描画されていくことになります。リスト 4-12にその実装例を示します。

リスト 4-12　姿勢の変化に応じて描画

```
-(void) startPitch {
    // referenceAttitudeはインスタンス変数
    referenceAttitude = [motionManager.deviceMotion.attitude retain];
}

- (void)drawView {
    CMAttitude *currentAttitude = motionManager.deviceMotion.attitude;
    [currentAttitude multiplyByInverseOfAttitude: referenceAttitude];
    // currentAttitudeに応じた位置にバットを描画 ...
    [self updateModelsWithAttitude:currentAttitude];
    [renderer render];
}
```

`multiplyByInverseOfAttitude:`の処理が終わると、この例の`currentAttitude`は、`referenceAttitude`で表される姿勢から、`CMAttitude`インスタンスが表す直近の姿勢に移るために必要な変化（すなわち回転）を表すようになります。

マルチメディアのリモートコントロール

リモートコントロールイベントを使えば、ユーザはアプリケーション上のマルチメディアを、システムトランスポートコントロールや外付け機器で制御できるようになります。アプリケーションに音声や画像の再生機能を組み込む場合、リモートコントロールイベントに応答するコードを記述する必要が生じるでしょう。このイベントは、トランスポートコントロールから送信されるほか、Appleが定める仕様に準拠する外付け機器（ヘッドセットなど）が、コマンドとして発行することもあります。iOSはこのコマンドを`UIEvent`オブジェクトに変換し、アプリケーションに送信します。アプリケーションはこれをファーストレスポンダに送信し、ファーストレスポンダが処理できなければ、レスポンダチェーンをたどって順次渡されていきます。

以下のセクションでは、リモートコントロールイベントを受信できるようアプリケーション側で準備する方法、受信して処理する方法を説明します。コード例は『*Audio Mixer (MixerHost)*』サンプルコードプロジェクトに由来します。

アプリケーションをリモートコントロールイベントに対応させる手順

リモートコントロールイベントを受信するためには、ビュー、またはマルチメディアコンテンツの再生を管理するビューコントローラが、ファーストレスポンダでなければなりません。さらに、ファーストレスポンダ（またはアプリケーションのほかのオブジェクト）が、アプリケーションオブジェクトに対して、リモートコントロールイベントを受信できる旨を通知する必要があります。

ビューまたはビューコントローラは、自分自身がファーストレスポンダになれるようにするため、`UIResponder`のメソッド`canBecomeFirstResponder`をオーバーライドして、`YES`を返すように記述します。さらに、適当な時点で自分自身に`becomeFirstResponder`を送る必要もあります。たとえば（ビューコントローラの場合）、`viewDidAppear:`メソッドをオーバーライドして、このメッセージを送信する記述を加えます。リスト 5-1に、この呼び出しをするコード例を示します。ここではさらに、ビューコントローラは`UIApplication`の`beginReceivingRemoteControlEvents`メソッドを呼び出して、リモートコントロールイベントの送信を「オンに」しています。

リスト 5-1　リモートコントロールイベントを受信する準備

```
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    [[UIApplication sharedApplication] beginReceivingRemoteControlEvents];
    [self becomeFirstResponder];
}
```

ビューまたはビューコントローラが音声/画像の管理をやめる場合には、リモートコントロールイベントの送信を停め、さらに`viewWillDisappear:`を実装して、ファーストレスポンダとしての役割を降りる必要があります（リスト 5-2を参照）。

リスト 5-2　リモートコントロールイベントの受信の終了

```
- (void)viewWillDisappear:(BOOL)animated {
    [[UIApplication sharedApplication] endReceivingRemoteControlEvents];
    [self resignFirstResponder];
    [super viewWillDisappear:animated];
}
```

リモートコントロールイベントの処理

リモートコントロールイベントを処理するためには、ファーストレスポンダに、`UIResponder`で宣言された`remoteControlReceivedWithEvent:`メソッドを実装する必要があります。このメソッドでは、引数として渡される各`UIEvent`オブジェクトの`subtype`を評価し、音声/画像を再生するオブジェクトに、適切なメッセージを送信しなければなりません。リスト 5-3のコード例では、音声オブジェクトに、再生、一時停止、停止の各メッセージを送信しています。

リスト 5-3　リモートコントロールイベントの処理

```
- (void) remoteControlReceivedWithEvent: (UIEvent *) receivedEvent {

    if (receivedEvent.type == UIEventTypeRemoteControl) {

        switch (receivedEvent.subtype) {

            case UIEventSubtypeRemoteControlTogglePlayPause:
                [self playOrStop: nil];
                break;

            case UIEventSubtypeRemoteControlPreviousTrack:
                [self previousTrack: nil];
                break;

            case UIEventSubtypeRemoteControlNextTrack:
                [self nextTrack: nil];
                break;

            default:
                break;
        }
    }
}
```

`UIEvent`にはほかにもリモートコントロールに関するサブタイプがあります。詳細については、『*UIEvent Class Reference*』を参照してください。

アプリケーションが正しくリモートコントロールイベントを受信し、処理しているかのテストには、「Now Playing」コントロールが便利です。これはiOS 4.0以降が動作する最近の機種（iPhoneであればiPhone 3GS以降）で使えます。「ホーム」ボタンをダブルプレスし、画面の底部で左か右にフリックしていくと、音声の再生コントロールが現れます。このコントロールは、音声を現在再生しているか、または直近に再生していたアプリケーションに、リモートコントロールイベントを送信します。再生コントロールの右側にあるアイコンがアプリケーションを表します。

プログラムでアプリケーションを操作して音声を再生しておき、「Now Playing」コントロールをタップしてリモートコントロールイベントをアプリケーションに送信し、動作をテストするとよいでしょう。アプリケーションを配布する際は、プログラムで再生を始める機能を無効にしてください。ユーザが何も操作しないのに再生が始まる、ということがあってはなりません。

書類の改訂履歴

この表は「*iOSイベント処理ガイド*」の改訂履歴です。

日付	メモ
2011-03-10	細かな訂正をいくつか行いました。
2010-09-29	細かな訂正をいくつか行って、わかりやすくしました。
2010-08-12	「マルチメディアのリモートコントロール」のコードを修正しました。
2010-08-03	「マルチメディアのリモートコントロール」の章で、サンプルコードおよび関連するテキストを修正しました。その他の細かな訂正を行いました。
2010-07-09	『iPhone OSイベント処理ガイド』から文書名を変更し、「iPhone OS」の記述を「iOS」に変更しました。Core Motionフレームワークについてのセクションを更新しました。
2010-05-18	アプリケーションがマルチタッチイベント、モーションイベント、およびその他のイベントを処理する方法について解説した文書の初版。

書類の改訂履歴