Getting Processed Device-Motion Data

On This Page

The device-motion service offers a simple way for you to get motion-related data for your app. Raw accelerometer and gyroscope data needs to be processed to remove bias from other factors, such as gravity. The device-motion service does this processing for you, giving you refined data that you can use right away. For example, this service provides separate values for user-initiated accelerations and for accelerations caused by gravity. Therefore, this service lets you focus on using the data to manipulate your content rather than on processing that data.

The device-motion service uses the available hardware to generate a `CMDeviceMotion` object, which contains the following information:

- The device's orientation (or attitude) in three-dimensional space relative to a reference frame
- The unbiased rotation rate
- The current gravity vector
- The user-generated acceleration vector (without gravity)
- The current magnetic field vector

You access the device-motion service using the `CMMotionManager` class of the Core Motion framework. Before enabling the service, always check the value of the `deviceMotionAvailable` property to verify that the service is available for you to use. You have several options for receiving data from the device-motion service. You can get the motion data only when you need it, or you can ask the framework to push updates to your app at regular intervals. Each technique involves different configuration steps and has a different use case.

> IMPORTANT
>
> If your app relies on the presence of accelerometer and gyroscope hardware, configure the `UIRequiredDeviceCapabilities` key of its `Info.plist` file with the `accelerometer` and `gyroscope` values. For more information about the meaning of this key, see *Information Property List Key Reference*.

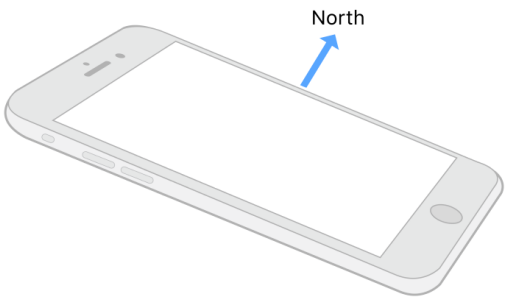
For general information about the classes of the Core Motion framework, see *Core Motion Framework Reference*.

Understanding Reference Frames and Device Attitude

The device attitude (or device orientation) reported by the device-motion service is always specified relative to a fixed reference frame. You decide which reference frame to use based on the interfaces you use to start the service. You can transform attitude values to other reference frames as needed.

The `CMAttitudeReferenceFrame` type defines several fixed initial reference frames that you may use. Each reference frame assumes a device that is lying on a flat surface and is rotated in a particular direction. For example, the `XMagneticNorthZVertical` reference frame corresponds to a device whose x axis points toward magnetic north, as illustrated in Figure 16-1. Other reference frames assume a slightly different device orientation.

Figure 16-1 An initial reference frame where the device's x axis points to magnetic north

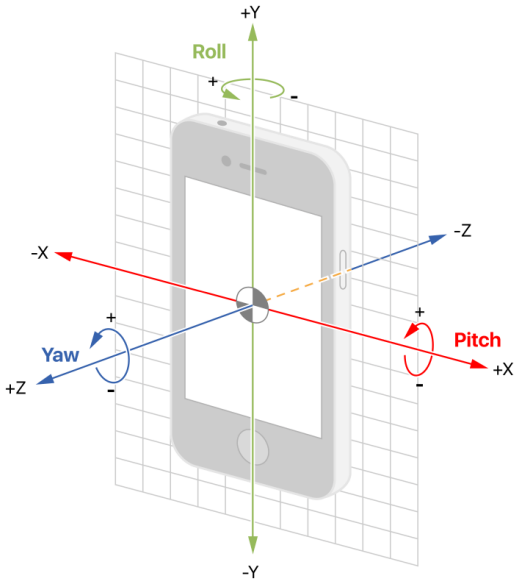


The attitude property of the `CMDeviceMotion` object contains pitch, roll, and yaw values for the device that are relative to the reference frame. These values provide the device's orientation in three-dimensional space. When all three values are 0, the device's orientation matches the orientation of the reference frame. As the device rotates around each axis, the pitch, roll, and yaw values reflect the amount of rotation (in radians) around the given axis. Rotation values may be positive or negative and are in the range -∏ to ∏.

Figure 16-2 shows how pitch, roll, and yaw are reported relative to the initial reference frame. Pitch reports rotations around the device's x axis, roll reports rotations around the y axis, and yaw reports rotations around the z axis.

On This Page

Figure 16-2 Pitch, roll, and yaw axes



For some types of apps, you can simplify your calculations by using the device's current attitude as the starting point for tracking changes. For example, a baseball app that measures the user's bat swing might use the current attitude as the starting bat position. As the user swings the bat, the app would then track only the changes from this initial start position.

Listing 16-1 shows how a baseball app would establish a new initial attitude for tracking the user's bat swing. The `startPitch` method saves the device's current attitude immediately before the swing begins. As the user swings, the `drawView` method multiplies the device's current attitude by the inverse of the reference attitude, which yields the amount of change between the two attitudes. It then uses this transformed data to update the bat's position onscreen.

Listing 16-1 Getting the change in attitude prior to rendering

```
-(void) startPitch {
    // referenceAttitude is a property
    self.referenceAttitude = self.motionManager.deviceMotion.attitude;
}

- (void)drawView {
    CMAttitude *currentAttitude = self.motionManager.deviceMotion.attitude;
    [currentAttitude multiplyByInverseOfAttitude: self.referenceAttitude];
    // Render bat using currentAttitude
    [self updateModelsWithAttitude:currentAttitude];
    [renderer render];
}
```

Getting Motion Data Only When You Need It

For apps that process device-motion data on their own schedule, such as games, use the `startDeviceMotionUpdates` or `startDeviceMotionUpdatesUsingReferenceFrame(_:)` method of `CMMotionManager` to start the delivery of motion data. When you call this method, the system enables the needed hardware and begins updating the `deviceMotion` property of your `CMMotionManager` object. However, the system does not notify you when it updates that property. You must explicitly check the value of the property when you need the motion data.

Before you start the delivery of motion updates, specify an update frequency by assigning a value to the `deviceMotionUpdateInterval` property. The maximum frequency at which you can request updates is hardware dependent but is usually at least 100 Hz. If you request a frequency that is greater than what the hardware supports, Core Motion uses the supported maximum instead.

Listing 16-2 shows a method that configures the device-motion service to deliver new motion data 60 times per second. The method then configures a timer to fetch those updates at the same frequency and do something with the data. You could configure the timer to fire at a lower frequency, but doing so would waste power by causing the hardware to generate more updates than were actually used. Setting the `showsDeviceMov` ensure an accurate starting point for motion tracking.

On This Page

Listing 16-2 Fetching device-motion data on demand

```
func startDeviceMotion() {
    if motion.isDeviceMotionAvailable {
        self.motion.deviceMotionUpdateInterval = 1.0 / 60.0
        self.motion.showsDeviceMovementDisplay = true
        self.motion.startDeviceMotionUpdates(using: .xMagneticNorthZVertical)

        // Configure a timer to fetch the motion data.
        self.timer = Timer(fire: Date(), interval: (1.0/60.0), repeats: true, block:
  { (timer) in
            if let data = self.motion.deviceMotion {
                // Get the attitude relative to the magnetic north reference frame.
                let x = data.attitude.pitch
                let y = data.attitude.roll
                let z = data.attitude.yaw

                // Use the motion data in your app.
            }
        })

        // Add the timer to the current run loop.
        RunLoop.current.add(self.timer!, forMode: .defaultRunLoopMode)
    }
}
```

Processing a Steady Stream of Motion Updates

When you want to capture all of the device-motion data being generated, perhaps so you can analyze it for movement patterns, use the `startDeviceMotionUpdatesUsingReferenceFrame(_:toQueue:withHandler:)` or `startDeviceMotionUpdatesToQueue(_:withHandler:)` method of `CMMotionManager`. These methods push each new data set to your app by executing your handler block on the specified queue. The queueing of these blocks ensures that your app receives all of the motion data, even if your app becomes busy and is unable to process updates for a brief period of time.

Before you start the delivery of motion updates, specify an update frequency by assigning a value to the `deviceMotionUpdateInterval` property. The maximum frequency at which you can request updates is hardware dependent but is usually at least 100 Hz. If you request a frequency that is greater than what the hardware supports, Core Motion uses the supported maximum instead.

Listing 16-3 shows a method that receives incoming motion data at a rate of 60 times per second. Because the incoming data is delivered using an operation queue, the app is guaranteed to receive all of the data generated by Core Motion. Setting the `showsDeviceMovementDisplay` property to `YES` lets the system display the standard calibration interface to ensure an accurate starting point for motion tracking.

Listing 16-3 Accessing queued motion data

```
func startQueuedUpdates() {
    if motion.isDeviceMotionAvailable {
        self.motion.deviceMotionUpdateInterval = 1.0 / 60.0
        self.motion.showsDeviceMovementDisplay = true
        self.motion.startDeviceMotionUpdates(using: .xMagneticNorthZVertical,
                                             to: self.queue, withHandler: { (data,
  error) in
                                                // Make sure the data is valid
  before accessing it.
                                                if let validData = data {
```

```
9                                               // Get the attitude relative to
   the magnetic north reference frame.
10                                              let roll =
   valic                                                                          On This Page
11                                              let pitch =
   validData.attitude.pitch
12                                              let yaw = validData.attitude.yaw
13
14                                              // Use the motion data in your
   app.
15                                          }
16          })
17      }
18  }
```