Core Motion

Core Motion

Core Motion is a framework that allows your application to receive motion data from device hardware.

For an iOS developer, this means you can create applications that can observe and respond to the motion and orientation of an iOS device.

Important note:

You can only test or use the functionality of Core Motion on an actual device. The simulator does not have any facilities for reproducing physical motion for your app.

Hardware Elements of Core Motion

Accelerometer
- Measures acceleration in all three dimensions

Gyroscope
- Calculates orientation and rotation in all three dimensions

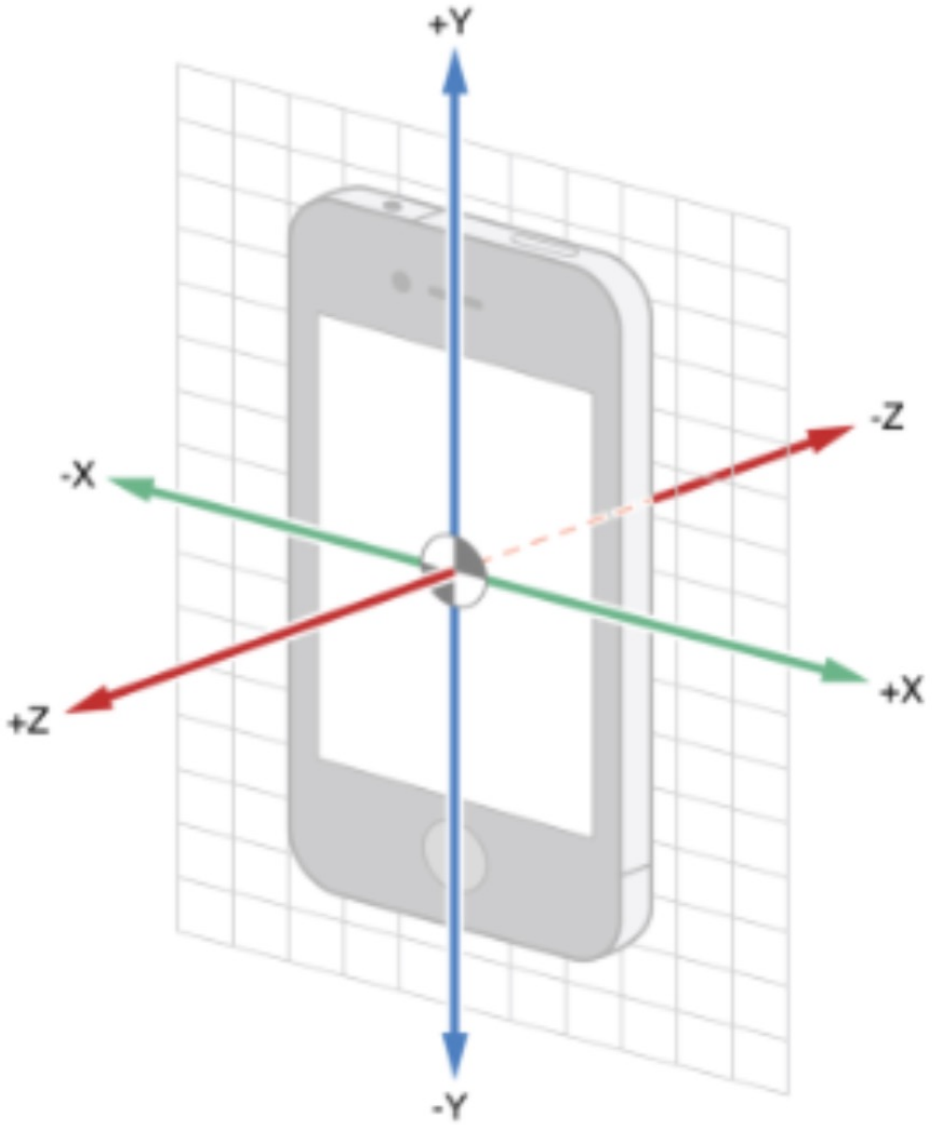
Magnetometer
- Measures magnetic forces

Coordinate System

+x is towards the right of the screen

+y is towards the top of the screen

+z is towards the user when the screen is faceup

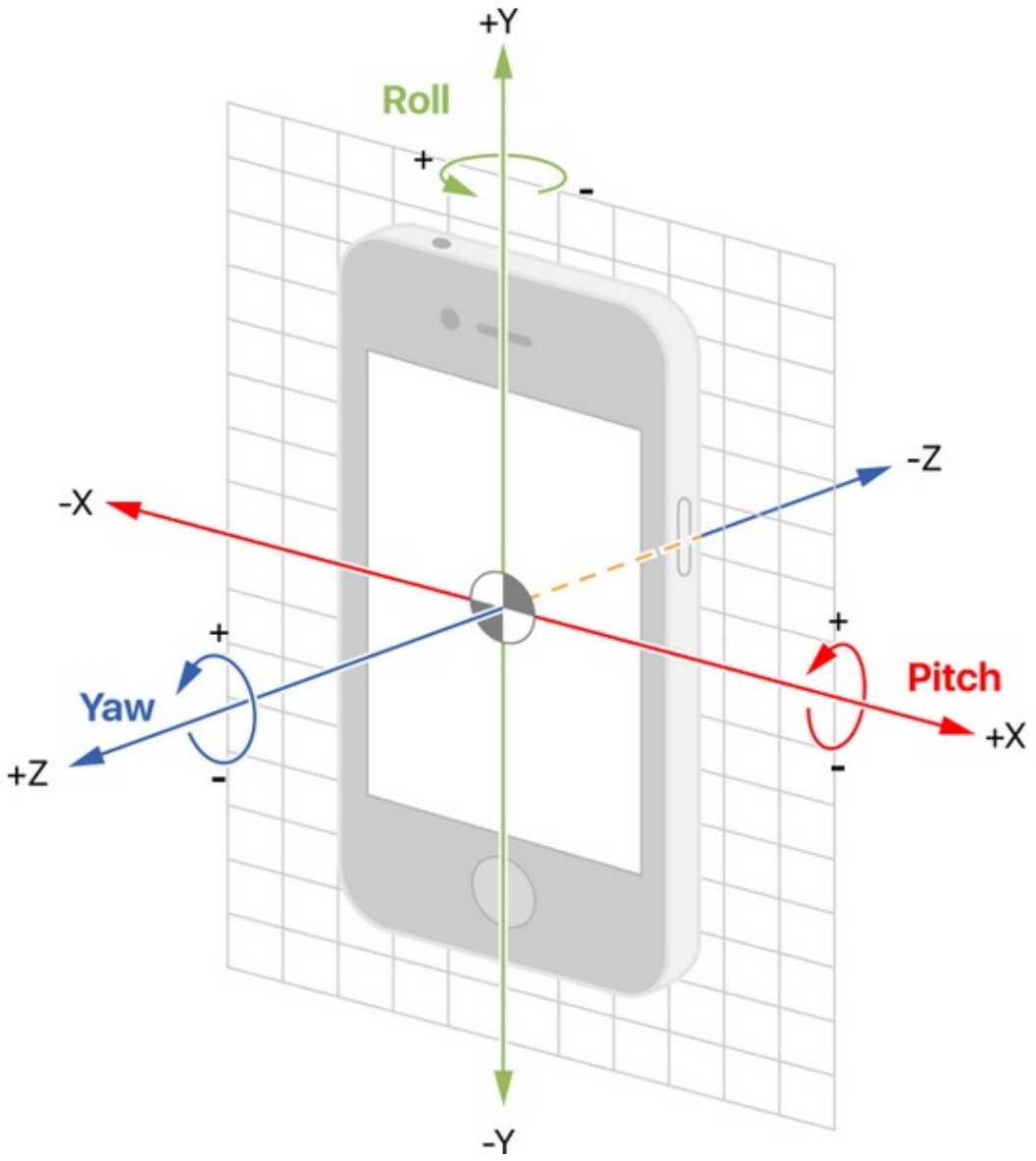


Rotations Within the Coordinate System

Pitch: rotation around the x axis

Roll: rotation around the y axis

Yaw: rotation around the z axis



CMDeviceMotion

A `CMDeviceMotion` object contains the following objects as properties:

- **`attitude:`** `CMAttitude`
  - Returns the orientation of the device
  - Can access the data in any of 3 representations
- **`rotationRate:`** `CMRotationRate`
  - Returns the rotation rate of the device for devices with a gyro
  - x, y, z values in radians per second
- **`gravity:`** `CMAcceleration`
  - Returns the gravity vector expressed in the device's reference frame
  - x, y, z values in g's (gravitational force)
- **`userAcceleration:`** `CMAcceleration`
  - Returns the acceleration that the user is giving to the device
  - x, y, z values in g's (gravitational force)
- **`magneticField:`** `CMCalibratedMagneticField`
  - Returns the magnetic field vector with respect to the device for devices with a magnetometer

`CMAttitude`

`CMAttitude` contains three different representations of the device's orientation:

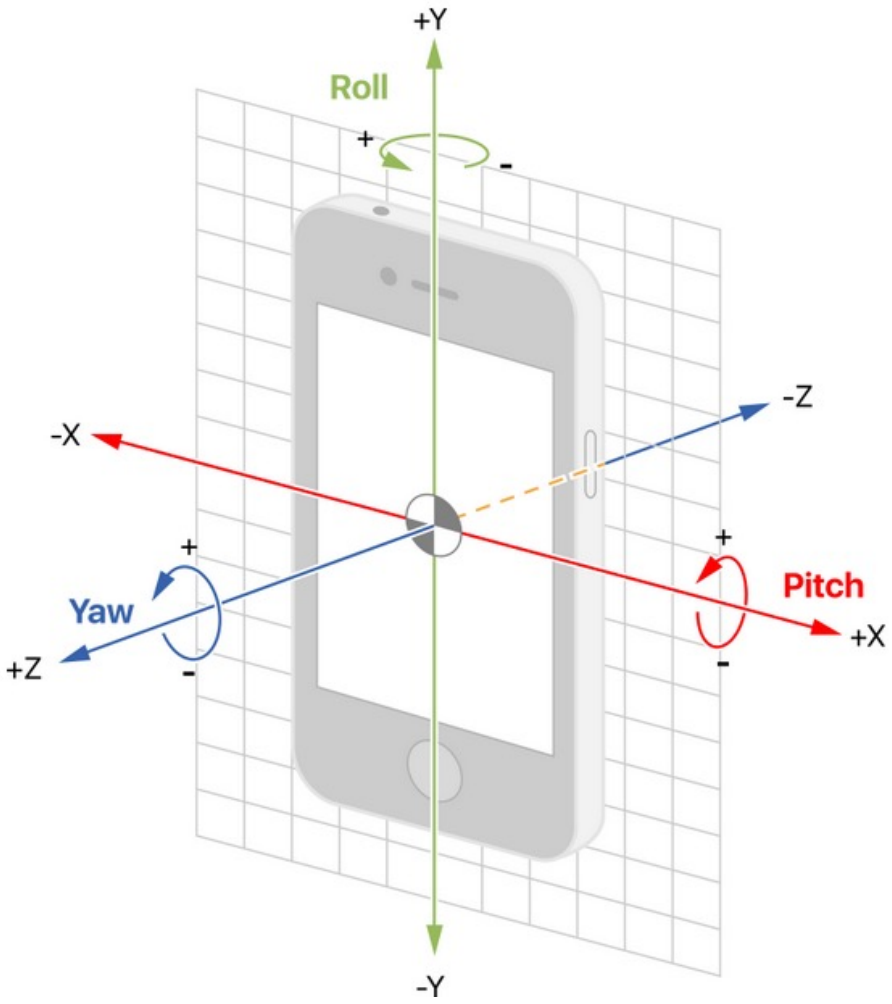
- Euler angles (pitch, roll, yaw)
- Rotation matrices
- Quaternions

Each of these is in relation to a given reference frame.

Euler Angles

Euler Angles are the most readily understood of the 3 representations, as they simply describe rotation around each of the axes.

- **Pitch** - is rotation around the **x-axis**, increasing as the device tilts toward you, decreasing as it tilts away
- **Roll** - is rotation around the **y-axis**, decreasing as the device rotates to the left, increasing to the right
- **Yaw** - is rotation around the **z-axis**, decreasing clockwise, increasing counter-clockwise



CMMotionManager

`CMMotionManager` provides a consistent interface for each of the four motion data types:

- Attitude (rotation)
- Acceleration
- Gravity
- Magnetic Field

Although you can access data for each of these motion types individually, it's simplest to create a `CMMotionManager` instance to access all of the above.

If `deviceMotion` is a `CMDeviceMotion` object:

```
deviceMotion.gravity.x
deviceMotion.gravity.y
deviceMotion.gravity.z

deviceMotion.userAcceleration.x
deviceMotion.userAcceleration.y
deviceMotion.userAcceleration.z

deviceMotion.attitude.pitch
deviceMotion.attitude.roll
deviceMotion.attitude.yaw

deviceMotion.magneticField.field.x
deviceMotion.magneticField.field.y
deviceMotion.magneticField.field.z
```

Using Core Motion in Your App

1. Create a motion manager:

In your `ViewController` class:

```
let motionManager = CMMotionManager()
```

2. Start receiving updates at the desired frequency:

```
override func viewDidLoad() {
    super.viewDidLoad()

    motionManager.deviceMotionUpdateInterval = 0.1

    motionManager.startDeviceMotionUpdates(to:
                          OperationQueue.current!) {
        (deviceMotion, error) -> Void in

        if(error == nil) {    // you write these methods
            self.handleUpdate(deviceMotion: deviceMotion!)
        } else {
            self.handleError()
        }
    }
}
```

Using Core Motion in Your App

3. Write code specifying what you want to happen at each update

```
func handleUpdate(deviceMotion:CMDeviceMotion) {

    let acceleration = deviceMotion.userAcceleration
    let xAcc = acceleration.x
    let yAcc = acceleration.y
    let zAcc = acceleration.z

    print("Acceleration in the x direction: \(xAcc) ")
    print("Acceleration in the y direction: \(yAcc) ")
    print("Acceleration in the z direction: \(zAcc) ")
}

func handleError() {
    print("An error occurred")
```

Alert Views

Alert Views

- Alert views are an easy way to display concise and informative information to the user.
- The kind of UI that is displayed in a UI Alert Controller is specified by the controller's `preferred` style when creating the controller
- You customize the UI by identifying what buttons or text fields you want to include

Key classes

The primary classes used in an Alert are:

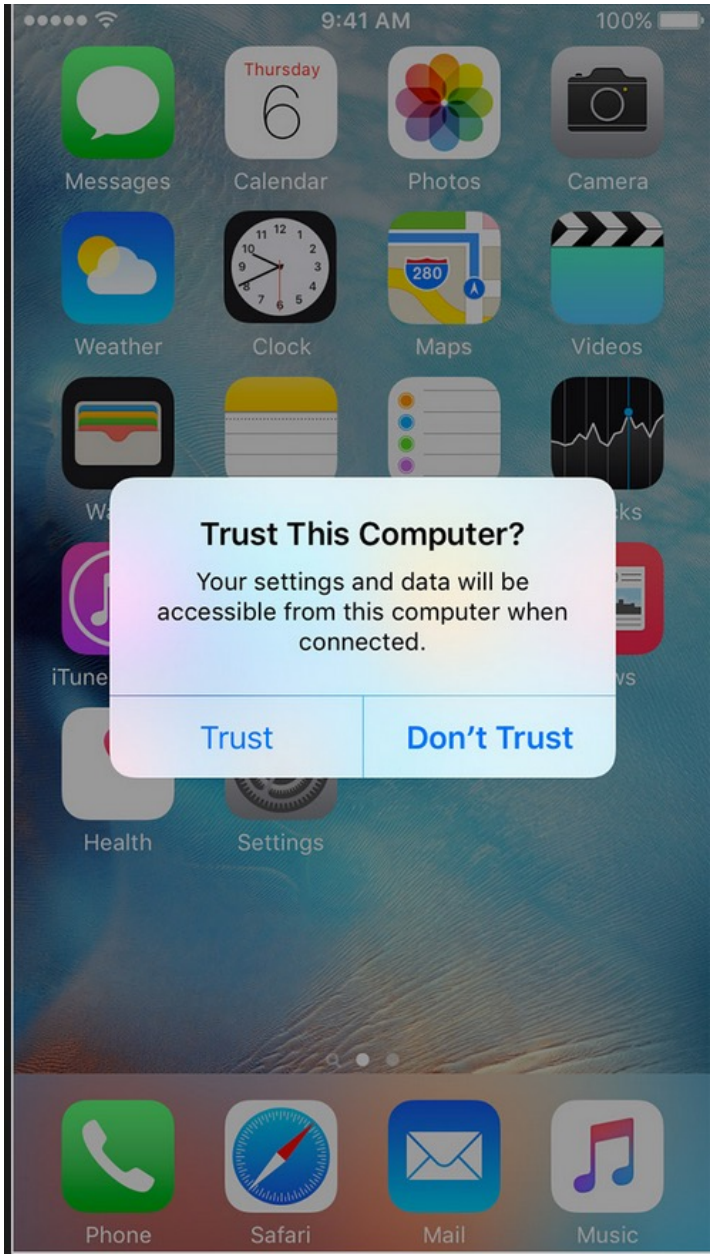
- `UIAlertController`
  is a VC that displays an alert message to the user
- `UIAlertAction`
  represents an action that can be taken when tapping a button in an alert

You create a `UIAlertController` object first, and then add as many `UIAlertAction` objects as needed, typically based on the number of buttons defined.

`UIAlertController` Style Settings

Alert: a UI that displays over and grays out the current UI.

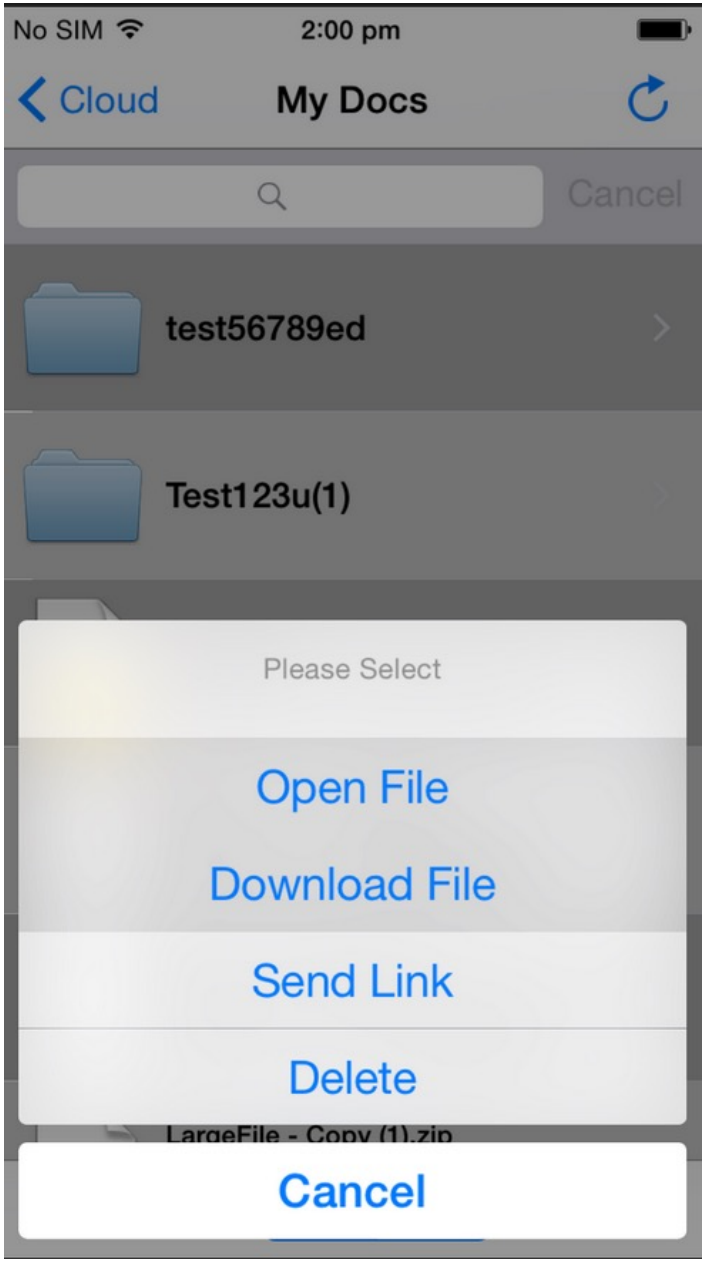
- "Trust" and "Don't Trust" are the two `UIAlertAction` objects.



`UIAlertController` Style Settings

Action Sheet: a UI that slides up from the bottom of the screen and grays out the current UI.

- In this example, there are five `UIAlertAction` objects.



`UIAlertAction`

- A `UIAlertAction` represents an action that can be taken when tapping a button in an alert
- You use this class to configure information about a single action, including
  - The *title* to display in the button
  - Any *style* information
  - A *handler* to execute when the user taps the button

`UIAlertAction` Style Settings

- *Default*:
  - Apply the default style to the action's button
  - Normal text
- *Cancel*:
  - Apply a style that indicate the action cancels the operation and leaves things unchanged
  - Can only have one of these. (App crashes if you define more than one for a given button)
  - Bold text
- *Destructive*:
  - Apply a style that indicates the action might change or delete data
  - Red text color

Calendar and EventKit

Event Kit

Event Kit is a set of classes for accessing and manipulating a user's calendar events and reminders, which live in the *Event Store* database on a device.

You can, among other things:

- Create a calendar
- Delete a calendar
- Get a list of calendars
- Get the attributes of a given calendar
- Create an event
- Modify an event
- Delete an event

Event Kit

At the heart of EventKit is the class `EKEventStore`.

An instance of EKEventStore provides access to an API for performing read and write operations on the user's calendars and reminder lists.

```
let eventStore = EKEventStore()
```

Event Kit Authorization

Your app must ask for permission to access the calendars and/or reminders.

- Check to see if your app is authorized:

```
authorizationStatus(
    for entityType: EKEntityType) ->
    EKAuthorizationStatus
```

`entityType`: either `.event` or `.reminder`

returns `EKAuthorizationStatus`:

```
.authorized
.denied
.notDetermined
.restricted
```

Event Kit

- If your app isn't authorized, you must request access.

```
requestAccess(
    to entityType: EKEntityType,
    completion: <completion handler>)
```

`entityType`: either `.event` or `.reminder`
`completion`: code to execute when the request completes.

- Your app is not blocked while the user decides.
- The completion handler executes regardless of what the user's choice was.

Note that the user can change the calendar access state at any time. Consequently, include this code in `viewWillAppear` to make sure that the current state of authorization is used each time the user sees the application interface.

Event Kit

To use Event Kit:

- `import EventKit`
- Create an instance of `EKEventStore`
- Through the `EKEventStore` object:
  - Verify that your app has permission to access the event store
  - Include handling if you don’t have access
- Read and write calendars / events from and to the event store

Event Kit

To check to see if your app is authorized to access the event store:

```
if (EKEventStore.authorizationStatus(for: .event) !=
                    EKAuthorizationStatus.authorized) {
    < handle error >
} else {
    < do stuff >
}
```

If the status returned is `Authorized`, you can start reading and writing from or to the Event Store.

If the status returned is `NotDetermined` (as in the first execution), then ask the user for access to the calendars:

```
eventStore.requestAccess(to: .event,
    completion: {(accessGranted: Bool, error: NSError?) in

if accessGranted == true {
   <we can access the event store>
   } else {
   <help the user give you access>
   }

})
```

Event Kit

Once you've been given access to the calendars, you can get a list of them:

```
eventStore.calendarsForEntityType(EKEntityType.Event)
```

This returns an array of `EKCalendar` objects.

Managing Calendars

Creating calendars:

- Create an `EKCalendar` object.
- Set various attributes.
- After saving, store the key associated with that calendar.

Deleting a calendar:

- Get the calendar to delete using the stored key.
- Remove the calendar.

Creating events:

- Get the calendar you want to add an event to.
- Create an `EKEvent` object.
- Set various attributes.
- Save.

Events

To create an event:

- create an instance of `EKEvent` for the appropriate `eventStore`:

```
let event = EKEvent(eventStore:eventStore)
```

- set the properties of the event:

```
event.title = “UT vs. Oklahoma”
event.startDate = Date(“2019-10-12”)
event.calendar = calendarKey
```