Stanford CS193p

Developing Applications for iOS
Fall 2017-18

Today

- Core Motion
  - Detecting the position and movement of the device
  - Demo: Gravity-driven Playing Card
- Camera
  - How to take pictures in your app
  - Demo: Taking a picture to be our background image in EmojiArt

Core Motion

- API to access motion sensing hardware on your device
- Primary inputs: Accelerometer, Gyro, Magnetometer
  Not all devices have all inputs (e.g. only later model devices have a gyro)
- Class used to get this input is `CMMotionManager`
  Use only one instance per application (else performance hit)
  It is a "global resource," so getting one via a class method somewhere is okay
- Usage
  1. Check to see what hardware is available
  2. Start the sampling going and poll the motion manager for the latest sample it has

  ... or ...

  1. Check to see what hardware is available
  2. Set the rate at which you want data to be reported from the hardware
  3. Register a closure (and a queue to run it on) to call each time a sample is taken

Core Motion

- Checking availability of hardware sensors

  `var {accelerometer,gyro,magnetometer,deviceMotion}Available: Bool`

  The "device motion" is a combination of all available (accelerometer, magnetometer, gyro).

  We'll talk more about that in a couple of slides.

- Starting the hardware sensors collecting data

  You only need to do this if you are going to poll for data.

  Generally used when some architecture in your app is already periodic (e.g. animation frames).

  `func start{Accelerometer,Gyro,Magnetometer,DeviceMotion}Updates()`

- Is the hardware currently collecting data?

  `var {accelerometer,gyro,magnetometer,deviceMotion}Active: Bool`

- Stop the hardware collecting data

  It is a performance hit to be collecting data, so stop during times you don't need the data.

  `func stop{Accelerometer,Gyro,Magnetometer,DeviceMotion}Updates()`

Core Motion

- Checking the data (from existing periodic mechanism)

```
var accelerometerData: CMAccelerometerData?
```

`CMAccelerometerData` object provides `var acceleration: CMAcceleration`

```
struct CMAcceleration {
     var x: Double   // in g (9.8 m/s/s)
     var y: Double   // in g
     var z: Double   // in g
}
```

This raw data includes acceleration due to gravity

So, if the device were laid flat, `z` would be `1.0` and `x` and `y` would be `0.0`

Core Motion

- Checking the data (from existing periodic mechanism)

```
var gyroData: CMGyroData?
```

CMGyroData object provides `var rotationRate: CMRotationRate`

```
struct CMRotationRate {
     var x: Double   // in radians/s
     var y: Double   // in radians/s
     var z: Double   // in radians/s
}
```

Sign of the rotation data follows right hand rule

The data above will be biased

Core Motion

- Checking the data (from existing periodic mechanism)

```
var magnetometerData: CMMagnetometerData?
```

CMMagnetometerData object provides `var magneticField: CMMagneticField`

```
struct CMMagneticField {
    var x: Double   // in microteslas
    var y: Double   // in microteslas
    var z: Double   // in microteslas
}
```

The data above will be biased

CMDeviceMotion

- CMDeviceMotion is a "combined" motion data source

  It uses information from all the hardware to improve the data from each.

  ```
  var deviceMotion: CMDeviceMotion?
  ```

- Acceleration Data in CMDeviceMotion

  ```
  var gravity: CMAcceleration
  var userAcceleration: CMAcceleration // gravity factored out using gyro
  ```

- Other information in CMDeviceMotion

  ```
  var rotationRate: CMRotationRate // bias removed from raw data using accelerometer
  var attitude: CMAttitude         // device's attitude (orientation) in 3D space
  class CMAttitude: NSObject       // roll, pitch and yaw are in radians
      var roll: Double     // around longitudinal axis passing through top/bottom
      var pitch: Double    // around lateral axis passing through sides
      var yaw: Double      // around axis with origin at CofG and ⊥ to screen directed down
  }
  var heading: Double // in degrees, where 0 is north (true or magnetic depending on frame)
  ```

CMDeviceMotion

- Reference Frame

  Magnetometer use in CMDeviceMotion can be controlled by setting its reference frame.
  Specify this when calling startDeviceMotionUpdates.

  xArbitraryZVertical // the default, does not use magnetometer
  xArbitraryCorrectedZVertical // uses magnetometer (if available) to correct yaw over time
  xMagnetic/TrueNorthZVertical // uses magnetometer for device position/heading in world
  These last two may require the user to calibrate the magnetometer.
  And for TrueNorth, location information (e.g. GPS/Wifi/Cellular) will also be required.
  North frames are necessary for apps that use things like Augmented Reality.
  To get heading, for example, you must use a MagneticNorth or TrueNorth reference frame.

  Always check to make sure the reference frame you want is available on the device …

  ```
  static func availableAttitudeReferenceFrames() -> CMAttitudeReferenceFrame
  ```

Core Motion

- Registering a block to receive Accelerometer data

```
func startAccelerometerUpdatesToQueue(queue: OperationQueue,
                                withHandler: CMAccelerometerHandler)
typealias CMAccelerationHandler = (CMAccelerometerData?, Error?) -> Void
```

  queue can be an `OperationQueue()` you create or `Operation.main` (or `current`)

- Registering a block to receive Gyro data

```
func startGyroUpdatesToQueue(queue: OperationQueue,
                       withHandler: CMGyroHandler)
typealias CMGyroHandler = (CMGyroData?, Error?) -> Void
```

- Registering a block to receive Magnetometer data

```
func startMagnetometerUpdatesToQueue(queue: OperationQueue,
                               withHandler: CMMagnetometerHandler)
typealias CMMagnetometerHandler = (CMMagnetometerData?, Error?) -> Void
```

Core Motion

- Registering a block to receive DeviceMotion data

```
func startDeviceMotionUpdates(using: CMAttitudeReferenceFrame,
                              queue: OperationQueue,
                        withHandler: (CMDeviceMotion?, Error?) -> Void)
```

queue can be an `OperationQueue()` you create or `Operation.mainQueue` (or `currentQueue`)

Errors … `CMErrorDeviceRequiresMovement`
`CMErrorTrueNorthNotAvailable`
`CMErrorMotionActivityNotAvailable`
`CMErrorMotionActivityNotAuthorized`

Core Motion

- Setting the rate at which your block gets executed

```
var accelerometerUpdateInterval: TimeInterval
var gyroUpdateInterval: TimeInterval
var magnetometerUpdateInterval: TimeInterval
var deviceMotionUpdateInterval: TimeInterval
```

- It is okay to add multiple handler blocks

  Even though you are only allowed one `CMMotionManager`
  However, each of the blocks will receive the data at the same rate (as set above)
  (Multiple objects are allowed to poll at the same time as well, of course.)

Accelerometer Over Time

- Historical Accelerometer Data

  Sometimes you don't need to look at the accelerometer in real time.
  You just want to know what happened over a period of time in the past.
  For example, if you want an idea of the user's physical movement pattern.
  The class `CMSensorRecorder` can record (at 50hz) and then play back accelerometer data.

  Not all devices are capable of this (iPhone 7 and later, Apple Watch).

```
isAccelerometerRecordingAvailable() -> Bool    // whether this device can record
```

  Start recording data …

```
func recordAccelerometer(forDuration: TimeInterval) // keep this short for performance
```

  Retrieving the recorded data …

```
func accelerometerData(from: Date, to: Date) -> CMSensorDataList // 3 day max
```

  You enumerate over the `CMAccelerometerData` objects in a `CMSensorDataList` with `for in` …

```
for dataPoint: CMRecordedAccelerometerData in sensorDataList { . . . }
```

Activity Monitoring

- Rough estimate of what the user is doing

  For example, `stationary`, `walking`, `running`, `automotive`, or `cycling`.

  You track this with a `CMMotionActivityManager` (not a `CMMotionManager`!).

  ```
  func startActivityUpdates(to: OperationQueue, withHandler: (CMMotionActivity?) -> Void)
  ```

  `CMMotionActivity` is one of the above activities.

  You can also query historical activity with …

  ```
  func queryActivityStarting(from: Date, to: Date, to: OperationQueue, withHandler: …)
  ```

Pedometer

- Pedometer

  Getting the user's "step" information only makes sense over time.
  Create a `CMPedometer` and then send it the message …

  ```
  func startUpdates(from: Date, withHandler: (CMPedometerData?, Error?) -> Void)
  ```

  The `from Date` is allowed to be in the past (but only last 7 days are recorded).
  Your handler will be called periodically with the `struct` `CMPedometerData` which has …
  `startDate` and `endDate` of the data
  `numberOfSteps`, `distance`, `averageActivePace`, and `currentPace` during the time
  also `floorsAscended` and `floorsDescended`

- Altimeter

  Get relative altitude changes.

  ```
  func startRelativeAltitudeUpdates(to: OperationQueue, withHandler: (CMAltitudeData?, Error?))
  ```

  `CMAltitudeData` has both change in altitude in meters and raw atmospheric pressure data.

Core Motion

- Checking the authorization status of hardware sensors

Some information is considered "private" to the user (e.g. fitness data).
Specifically `CMPedometer`, `CMSensorRecorder`, `CMMotionActivityManager` and `CMAltimeter`.
iOS will automatically ask the user (once) for permission to access this information.
You can find out what the status is at any time with this `static func` on each of these.

```
static func authorizationStatus() -> CMAuthorizationStatus
struct CMAuthorizationStatus {
    case notDetermined   // user has not yet been asked
    case restricted      // fitness data access disabled in Settings
    case denied          // user has explicitly denied your app access
    case authorized      // ready to go!
}
```

Lack of authorization may also show up as an error when you request data.

Demo

- Playing Card

  We'll make our playing cards be affected by "real gravity" using the accelerometer.

UIImagePickerController

- Modal view controller to get media from camera or photo library
  i.e., you put it up with `present(_:animated:completion:)`
- Usage
  1. Create it & set its `delegate` (it can't do anything without its `delegate`)
  2. Configure it (source, kind of media, user edibility, etc.)
  3. Present it
  4. Respond to `delegate` methods when user is done/cancels picking the media
- What the user can do depends on the platform
  Almost all devices have cameras, but some can record video, some can not
  You can only offer camera <u>or</u> photo library on iPad (not both together at the same time)
  As with all device-dependent API, we want to start by check what's available …
  `static func isSourceTypeAvailable(sourceType: UIImagePickerControllerSourceType) -> Bool`
  Source type is `.photoLibrary` or `.camera` or `.savedPhotosAlbum` (camera roll)

UIImagePickerController

- But don't forget that not every source type can give video

  So, you then want to check …

  `static func availableMediaTypes(for: UIImagePickerControllerSourceType) -> [String]?`

  Depending on device, will return one or more of these …

  `kUTTypeImage` // pretty much all sources provide this, hardly worth checking for even

  `kUTTypeLivePhoto` // must also say kUTTypeImage for this one to work

  `kUTTypeMovie` // audio and video together, only some sources provide this

- You can get even m

  (Though usually this is not

  `static func isCameraDeviceAvailable(UIImagePickerControllerCameraDevice) -> Bool`

  returns `.rear` or `.front`

  There are other camera-specific interrogations too, for example …

  `static func isFlashAvailableForCameraDevice(UIImagePickerControllerCameraDevice) -> Bool`

These are declared in the MobileCoreServices framework.
`import MobileCoreServices`

UIImagePickerController

- Set the source and media type you want in the picker

  Example setup of a picker for capturing video (kUTTypeMovie) …

  (From here out, UIImagePickerController will be abbreviated UIIPC for space reasons.)

```
let picker = UIImagePickerController()
let mediaTypeMovie = kUTTypeMovie as String
picker.delegate = self // self must implement UINavigationControllerDelegate too
if UIIPC.isSourceTypeAvailable(.camera) {
    picker.sourceType = .camera
    if let availableTypes = UIIPC.availableMediaTypesForSourceType(.camera) {
        if availableTypes.contains(mediaTypeMovie) {
            picker.mediaTypes = [mediaTypeMovie]
            // proceed to put the picker up
        }
    }
}
```

UIImagePickerController

- Editability

```
var allowsEditing: Bool
```

If true, then the user will have opportunity to edit the image/video inside the picker.
When your delegate is notified that the user is done, you'll get both raw and edited versions.

- Limiting Video Capture

```
var videoQuality: UIImagePickerControllerQualityType
.typeMedium                 // default
.typeHigh
.type640x480
.typeLow
.typeIFrame1280x720         // native on some devices
.typeIFrame960x540          // native on some devices

var videoMaximumDuration: TimeInterval    // defaults to 10 minutes
```

UIImagePickerController

- Present the picker

  ```
  present(picker, animated: true, completion: nil)
  ```

  On iPad, if you are not offering Camera (just photo library), you must present with popover.
  If you are offering the Camera on iPad, then full-screen is preferred.
  Remember: on iPad, it's Camera OR Photo Library (not both at the same time).

- Delegate will be notified when user is done

  ```
  func imagePickerController(UIIPC, didFinishPickingMediaWithInfo info: [String:Any]) {
      // extract image/movie data/metadata here from info, more on the next slide
      picker.presentingViewController?.dismiss(animated: true) { }
  }
  ```

- Also dismiss it when cancel happens

  ```
  func imagePickerControllerDidCancel(UIIPC) {
      picker.presentingViewController?.dismiss(animated: true) { }
  }
  ```

UIImagePickerController

- What is in that info dictionary?

```
UIImagePickerControllerMediaType          // kUTTypeImage, kUTTypeMovie
UIImagePickerControllerOriginalImage      // UIImage
UIImagePickerControllerEditedImage        // UIImage
UIImagePickerControllerImageURL           // URL (in a temp location, so move it to keep it)
UIImagePickerControllerCropRect           // CGRect (in an NSValue)
UIImagePickerControllerMediaMetadata      // Dictionary of info about the image
UIImagePickerControllerLivePhoto          // a PHLivePhoto
UIImagePickerControllerPHAsset            // a PHAsset (see PHPhotoLibrary)
UIImagePickerControllerMediaURL           // URL of the video if kUTTypeMovie
```

UIImagePickerController

- Saving taken images or video into the device's photo library

  You can save to the user's Camera Roll …

```
func UIImageWriteToSavedPhotosAlbum(
    _ image: UIImage,
    _ target: Any?,                // the object to send selector to when finished writing
    _ selector: Selector?          // selector to send to target when finished writing
    _ context: UnsafeMutableRawPointer?   // passed to the selector
)
```

  It's a bummer that this isn't closure-based, but it is what it is.
  This is a very simple and convenient way to do this.
  But this only makes sense if the user only occasionally would want to save an image.
  Otherwise, you'll want to integrate with the Photos application: checkout `PHPhotoLibrary`.
  Of course, you could also save the image into your own sandbox.
  You'd do that if the captured images only make sense inside your own app.

UIImagePickerController

- In general, much more sophisticated media capture is available

  This UIImagePickerController API is pretty simple, but more powerful API exists.

  Check out both PHPhotoLibrary and AVCaptureDevice.

UIImagePickerController

- Overlay View

  `var cameraOverlayView: UIView`

  Be sure to set this view's `frame` properly.

  Camera is always full screen, so use `UIScreen.main`'s `bounds` property.

  If you use the built-in controls at the bottom, you might want your view to be smaller.

- Hiding the normal camera controls (at the bottom)

  `var showsCameraControls: Bool`

  Will leave a blank area at the bottom of the screen (camera's aspect 4:3, not same as screen's).

  With no controls, you'll need an overlay view with a "take picture" (at least) button.

  That button should send `takePicture()` or (`startVideoCapture()`) to the picker.

  Don't forget to `dismiss` when you are done taking pictures.

- You can zoom or translate the image while capturing

  `var cameraViewTransform: CGAffineTransform`

  For example, you might want to scale the image up to full screen (some of it will get clipped).

Core Image and Vision

- Processing Images

  Core Image is a powerful and efficient framework for applying filters to your images.
  Has a couple of hundred filters to choose from (blur, depth, comparison, colors, smoothing, etc.).
  Vision framework provides powerful feature detection in images (e.g. faces, barcodes, etc.).
  Core Image also has some feature detection API.
  Check out Core Image and Vision in the documentation.

Demo Code

Download the Emoji Art demo from today's lecture.

Download the Playing Card demo from today's lecture.