Stanford CS193p

Developing Applications for iOS
Winter 2015

Today

- Animation
  Animating using simulated physics
- Demo
  Dropit

Dynamic Animation

- A little different approach to animation than last week
  - Set up physics relating animatable objects and let them run until they resolve to stasis.
  - Easily possible to set it up so that stasis never occurs, but that could be performance problem.
- Steps
  - Create a `UIDynamicAnimator`
  - Add `UIDynamicBehavior`s to it (gravity, collisions, etc.)
  - Add `UIDynamicItem`s (usually `UIView`s) to the `UIDynamicBehavior`s
    (`UIDynamicItem` is an `protocol` which `UIView` happens to implement)
  - That's it! Things will instantly start animating!

Dynamic Animation

- Create a UIDynamicAnimator

  ```
  var animator = UIDynamicAnimator(referenceView: UIView)
  ```

  If animating views, all views must be in a view hierarchy with `referenceView` at the top.

- Create and add UIDynamicBehavior instances

  e.g., `let gravity = UIGravityBehavior()`

  ```
  animator.addBehavior(gravity)
  ```

  e.g., `collider = UICollisionBehavior()`

  ```
  animator.addBehavior(collider)
  ```

Dynamic Animation

- Add UIDynamicItems to a UIDynamicBehavior

```
let item1: UIDynamicItem = … // usually a UIView
let item2: UIDynamicItem = … // usually a UIView
gravity.addItem(item1)
collider.addItem(item1)
gravity.addItem(item2)
```

item1 and item2 will both be affect by gravity

item1 will collide with collider's boundaries, but not with item2

Dynamic Animation

- UIDynamicItem protocol

  Any animatable item must implement this …

```
protocol UIDynamicItem {
    var bounds: CGRect { get }  // note that the size cannot be animated
    var center: CGPoint { get set }  // but the position can
    var transform: CGAffineTransform { get set }  // and so can the rotation
}
```

  UIView implements this protocol

  If you change center or transform while the animator is running,
  you must call this method in UIDynamicAnimator …

```
func updateItemUsingCurrentState(item: UIDynamicItem)
```

Behaviors

- UIGravityBehavior

```
var angle: CGFloat  // in radians; 0 is to the right; positive numbers are counter-clockwise
var magnitude: CGFloat  // 1.0 is 1000 points/s/s
```

- UIAttachmentBehavior

```
init(item: UIDynamicItem, attachedToAnchor: CGPoint)
init(item: UIDynamicItem, attachedToItem: UIDynamicItem)
init(item: UIDynamicItem, offsetFromCenter: CGPoint, attachedToItem/Anchor…)
var length: CGFloat  // distance between attached things (this is settable while animating!)
var anchorPoint: CGPoint  // can also be set at any time, even while animating
```

The attachment can oscillate (i.e. like a spring) and you can control frequency and damping

Behaviors

- UICollisionBehavior

```
var collisionMode: UICollisionBehaviorMode // .Items, .Boundaries, or .Everything
```

If .Items, then any items you add to a UICollisionBehavior will bounce off of each other

If .Boundaries, then you add UIBezierPath boundaries for items to bounce off of …

```
func addBoundaryWithIdentifier(identifier: NSCopying, forPath: UIBezierPath)
func removeBoundaryWithIdentifier(identifier: NSCopying)
var translatesReferenceBoundsIntoBoundary: Bool // referenceView's edges
```

Behaviors

- UICollisionBehavior

  How do you find out when a collision happens?

  ```
  var collisionDelegate: UICollisionBehaviorDelegate
  ```

  … this delegate will be sent methods like …

  ```
  func collisionBehavior(behavior: UICollisionBehavior,
      began/endedContactForItem: UIDynamicItem,
      withBoundaryIdentifier: NSCopying) // withItem:atPoint: too
  ```

  The `withBoundaryIdentifier` is the one you pass to `addBoundaryWithIdentifier()`
  It is an `NSCopying` (`NSString` & `NSNumber` are both `NSCopying`, so `String` & `Int`/`Double` work)
  In this delegate method you'll have to cast (with `as` or `as?`) the `NSCopying` to what you want

Behaviors

- UISnapBehavior

  `init(item: UIDynamicItem, snapToPoint: CGPoint)`

  Imagine four springs at four corners around the item in the new spot.

  You can control the damping of these "four springs" with `var damping: CGFloat`

- UIPushBehavior

  ```
  var mode: UIPushBehaviorMode  // .Continuous or .Instantaneous
  var pushDirection: CGVector
  ```

  … or …

  ```
  var angle: CGFloat // in radians and …
  var magnitude: CGFloat // magnitude 1.0 moves a 100x100 view at 100 pts/s/s
  ```

  Interesting aspect to this behavior

  If you push `.Instantaneous`, what happens after it's done?

  It just sits there wasting memory.

  We'll talk about how to clear that up in a moment.

Behaviors

- UIDynamicItemBehavior
  - Sort of a special "meta" behavior.
  - Controls the behavior of items as they are affected by other behaviors.
  - Any item added to this behavior (with `addItem`) will be affected by …

```
var allowsRotation: Bool
var friction: CGFloat
var elasticity: CGFloat
```

  - … and others, see documentation.

  - Can also get information about items with this behavior …

```
func linearVelocityForItem(UIDynamicItem) -> CGPoint
func addLinearVelocity(CGPoint, forItem: UIDynamicItem)
func angularVelocityForItem(UIDynamicItem) -> CGFloat
```

  - Multiple `UIDynamicItemBehaviors` affecting the same item(s) is "advanced" (not for you!)

Behaviors

- `UIDynamicBehavior`
  - Superclass of behaviors.
  - You can create your own subclass which is a combination of other behaviors.
  - Usually you override `init` method(s) and `addItem` and `removeItem` to call …
  - `func addChildBehavior(UIDynamicBehavior)`
  - This is a good way to encapsulate a physics behavior that is a composite of other behaviors.
  - You might also have some API which helps your subclass configure its children.
- All behaviors know the `UIDynamicAnimator` they are part of
  - They can only be part of one at a time.
  - `var dynamicAnimator: UIDynamicAnimator { get }`
  - And the behavior will be sent this message when its animator changes …
  - `func willMoveToAnimator(UIDynamicAnimator)`

Behaviors

- UIDynamicBehavior's action property
  - Every time the behavior acts on items, this block of code that you can set is executed ...
  - `var action: (() -> Void)?`
  - (i.e. it's called `action`, it takes no arguments and returns nothing)
  - You can set this to do anything you want.
  - But it will be called a lot, so make it very efficient.
  - If the `action` refers to properties in the behavior itself, watch out for memory cycles.

Stasis

- UIDynamicAnimator's delegate tells you when animation pauses
  - Just set the delegate …
  - `var delegate: UIDynamicAnimatorDelegate`
  - … and you'll find out when stasis is reached and when animation will resume …
  - `func dynamicAnimatorDidPause(UIDynamicAnimator)`
  - `func dynamicAnimatorWillResume(UIDynamicAnimator)`

Memory Cycle Avoidance

- Example of using `action` and avoiding a memory cycle

  Let's go back to the case of a `.Instantaneous UIPushBehavior`

  When it is done acting on its items, it would be nice to remove it from its animator

  We can do this with the `action` method, but we must be careful to avoid a memory cycle …

```
if let pushBehavior = UIPushBehavior(items: […], mode: .Instantaneous) {
    pushBehavior.magnitude = …
    pushBehavior.angle = …
    pushBehavior.action = {
        pushBehavior.dynamicAnimator!.removeBehavior(pushBehavior)
    }
    animator.addBehavior(pushBehavior) // will push right away
}
```

  The above has a memory cycle because its `action` captures a pointer back to itself

  So neither the action closure nor the pushBehavior can ever leave the heap

Memory Cycle Avoidance

- Example of using `action` and avoiding a memory cycle
  - Let's go back to the case of a `.Instantaneous UIPushBehavior`
  - When it is done acting on its items, it would be nice to remove it from its animator
  - We can do this with the `action` method, but we must be careful to avoid a memory cycle …

```
if let pushBehavior = UIPushBehavior(items: […], mode: .Instantaneous) {
    pushBehavior.magnitude = …
    pushBehavior.angle = …
    pushBehavior.action = { [unowned pushBehavior] in
        pushBehavior.dynamicAnimator!.removeBehavior(pushBehavior)
    }
    animator.addBehavior(pushBehavior) // will push right away
}
```

  - Now it no longer captures `pushBehavior`
  - This is safe to mark `unowned` because if the `action` closure exists, so does the `pushBehavior`
  - When the `pushBehavior` removes itself from the animator, the `action` won't keep it in memory
  - So they'll both leave the heap because the animator no longer points to the behavior