



Stanford CS193p

Developing Applications for iOS
Fall 2017-18



CS193p
Fall 2017-18

Today

👁 Animation

UIViewPropertyAnimator

Transitions

Dynamic Animator



UIView Animation

- Changes to certain UIView properties can be animated over time

`frame/center`

`bounds` (transient size, does not conflict with animating `center`)

`transform` (translation, rotation and scale)

`alpha` (opacity)

`backgroundColor`

- Done with `UIViewPropertyAnimator` using closures

You define some animation parameters and pass an animation block (i.e. closure).

The animation block contains the code that makes the changes to the UIView(s).

The animation block will only animate changes to the above-mentioned UIView properties.

The changes inside the block are made immediately (even though they will appear “over time”).

Most also have another “completion block” to be executed when the animation is done.



UIViewPropertyAnimator

- Easiest way to use a UIViewPropertyAnimator

```
class func runningPropertyAnimator(  
    withDuration: TimeInterval,  
    delay: TimeInterval,  
    options: UIViewAnimationOptions,  
    animations: () -> Void,  
    completion: ((position: UIViewAnimatingPosition) -> Void)? = nil  
)
```

Note that this is a static (class) function. You send it to the UIViewPropertyAnimator type. The animations argument is a closure containing code that changes center, transform, etc. The completion argument will get executed when the animation finishes or is interrupted.



UIView Animation

👁 Example

```
if myView.alpha == 1.0 {  
    UIViewPropertyAnimator.runningPropertyAnimator(  
        withDuration: 3.0,  
        delay: 2.0,  
        options: [.allowUserInteraction],  
        animations: { myView.alpha = 0.0 },  
        completion: { if $0 == .end { myView.removeFromSuperview() } }  
    )  
    print("alpha = \(myView.alpha)")  
}
```

This would cause myView to “fade” out over 3 seconds (starting 2s from now).

Then it would remove myView from the view hierarchy (but only if the fade completed).

If, within the 5s, someone animated the alpha to non-zero, the removal would not happen.

The output on the console would immediately be ... **alpha = 0.0**

... even though the alpha on the screen won't be zero for 5 more seconds!



UIView Animation

• UIViewAnimationOptions

<code>beginFromCurrentState</code>	// pick up from other, in-progress animations of these properties
<code>allowUserInteraction</code>	// allow gestures to get processed while animation is in progress
<code>layoutSubviews</code>	// animate the relayout of subviews with a parent's animation
<code>repeat</code>	// repeat indefinitely
<code>autoreverse</code>	// play animation forwards, then backwards
<code>overrideInheritedDuration</code>	// if not set, use duration of any in-progress animation
<code>overrideInheritedCurve</code>	// if not set, use curve (e.g. ease-in/out) of in-progress animation
<code>allowAnimatedContent</code>	// if not set, just interpolate between current and end "bits"
<code>curveEaseInEaseOut</code>	// slower at the beginning, normal throughout, then slow at end
<code>curveEaseIn</code>	// slower at the beginning, but then constant through the rest
<code>curveLinear</code>	// same speed throughout



UIView Animation

- Sometimes you want to make an entire view modification at once

In this case you are not limited to special properties like alpha, frame and transform

Flip the entire view over `UIViewAnimationOptions.transitionFlipFrom{Left,Right,Top,Bottom}`

Dissolve from old to new state `.transitionCrossDissolve`

Curling up or down `.transitionCurl{Up,Down}`



UIView Animation

👁 Example

Flipping a playing card over ...

```
UIView.transition(with: myPlayingCardView,  
                 duration: 0.75,  
                 options: [.transitionFlipFromLeft],  
                 animations: { cardIsFaceUp = !cardIsFaceUp }  
                 completion: nil)
```

Presuming myPlayingCardView draws itself face up or down depending on cardIsFaceUp
This will cause the card to flip over (from the left edge of the card)



Dynamic Animation

- A little different approach to animation

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.



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 other items or boundaries, but not with item2



Dynamic Animation

- UIDynamicItem protocol

Any animatable item must implement this ...

```
protocol UIDynamicItem {  
    var bounds: CGRect { get } // essentially the size  
    var center: CGPoint { get set } // and the position  
    var transform: CGAffineTransform { get set } // rotation usually  
    var collisionBoundsType: UIDynamicItemCollisionBoundsType { get set }  
    var collisionBoundingPath: UIBezierPath { get set }  
}
```

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 clockwise
 - `var magnitude: CGFloat` // 1.0 is 1000 points/s/s

- UIAttachmentBehavior

- `init(item: UIDynamicItem, attachedToAnchor: CGPoint)`

- `init(item: UIDynamicItem, attachedTo: UIDynamicItem)`

- `init(item: UIDynamicItem, offsetFromCenter: CGPoint, attachedTo[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 addBoundary(withIdentifier: NSCopying, for: UIBezierPath)
```

```
func addBoundary(withIdentifier: NSCopying, from: CGPoint, to: CGPoint)
```

```
func removeBoundary(withIdentifier: NSCopying)
```

```
var translatesReferenceBoundsIntoBoundary: Bool // referenceView's edges
```

`NSCopying` means `NSString` or `NSNumber`, but remember you can `as` to `String`, `Int`, etc.



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/endedContactFor: UIDynamicItem,  
                      withBoundaryIdentifier: NSCopying // with:UIDynamicItem too  
                      at: CGPoint)
```

The withBoundaryIdentifier is the one you pass to addBoundary(withIdentifier:).



Behaviors

- UISnapBehavior

`init(item: UIDynamicItem, snapTo: 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 positive numbers are clockwise

`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 linearVelocity(for: UIDynamicItem) -> CGPoint`

`func addLinearVelocity(CGPoint, for: UIDynamicItem)`

`func angularVelocity(for: 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 add 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 willMove(to: 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 an `.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` var which takes a closure ...

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

But 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!



Aside: Closure Capturing

- You can define local variables on the fly at the start of a closure

```
var foo = { [x = someInstanceOfaClass, y = "hello"] in  
  // use x and y here  
}
```

- These locals can be declared weak

```
var foo = { [weak x = someInstanceOfaClass, y = "hello"] in  
  // use x and y here, but x is now an Optional because it's weak  
}
```

- Or they can even be declared "unowned"

unowned means that the reference counting system does count them (or check the count)

```
var foo = { [unowned x = someInstanceOfaClass, y = "hello"] in  
  // use x and y here, x is not an Optional  
  // if you use x here and it is not in the heap, you will crash  
}
```



Closure Capture

- This is all primarily used to prevent a memory cycle

```
class Zerg {  
    private var foo = {  
        self.bar()  
    }  
    private func bar() { . . . }  
}
```

This, too, is a memory cycle. Why?

The closure assigned to foo keeps self in the heap.

That's because closures are reference types and live in the heap

and they "capture" variables in their surroundings and keep them in the heap with them.

Meanwhile self's var foo is keeping the closure in the heap.

So foo points to the closure which points back to self which points to the closure.

A cycle.

Neither can leave the heap: there's always going to be a pointer to each (from the other).



Closure Capture

- We can break this with the local variable trick

```
class Zerg {  
    private var foo = { [weak weakSelf = self] in  
        weakSelf?.bar() // need Optional chaining now because weakSelf is an Optional  
    }  
    private func bar() { . . . }  
}
```

Now the closure no longer has a strong pointer to self.
So it is not keeping self in the heap with it.
The cycle is broken.



Closure Capture

- The local closure variable is allowed to be called `self` too

```
class Zerg {  
    private var foo = { [weak self = self] in  
        self?.bar() // still need chaining because self is a (local) Optional  
    }  
    private func bar() { . . . }  
}
```

There are two different “self” variables here.

The yellow one is local to the closure and is a weak Optional.

The green one is the instance of Zerg itself (and is obviously not weak, nor an Optional).



Closure Capture

- The local closure variable is allowed to be called `self` too

```
class Zerg {  
    private var foo = { [weak self] in  
        self?.bar() // still need chaining because self is a (local) Optional  
    }  
    private func bar() { . . . }  
}
```

There are two different “self” variables here.

The yellow one is local to the closure and is a weak Optional.

The green one is the instance of Zerg itself (and is obviously not weak, nor an Optional).

You don’t even need the “= self”.

By default, local closure vars are set equal to the var of the same name in their surroundings.



Memory Cycle Avoidance

👁 Example of using action and avoiding a memory cycle

Even more dramatically, we could use `unowned` to break a cycle.

The best example of this is back in our push behavior ...

```
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  
}
```

The action closure no longer captures `pushBehavior`.

And we can even use the local `pushBehavior` without any Optional chaining (it's not Optional).

It is safe to mark it `unowned` because if the action closure exists, so does the `pushBehavior`.

But we'd better be right. Or kaboom!



Demo Code

Download the demo code from today's lecture.

