Advanced Animations with UIKit

Session 230

Joe Cerra, UIKit Engineer
Basics

Interactive and Interruptible Animations

New Property Animator Behaviors

Coordinating Animations

Tips and Tricks
Basics
Interactive and Interruptible Animations
New Property Animator Behaviors
Coordinating Animations
Tips and Tricks
Basics
Interactive and Interruptible Animations
New Property Animator Behaviors
Coordinating Animations
Tips and Tricks
Basics
Interactive and Interruptible Animations
New Property Animator Behaviors
Coordinating Animations
Tips and Tricks
Basics
Interactive and Interruptible Animations
New Property Animator Behaviors
Coordinating Animations
Tips and Tricks
Basics
UIView-based Animations
UIView-based Animations
UIView-based Animations

```swift
UIView.animate(withDuration: 5) {
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}, completion: nil
```
UIView-based Animations

```swift
UIView.animate(withDuration: 5) {
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}, completion: nil)
```
UIView.animate(withDuration: 5) {
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}, completion: nil)
UIView-based Animations

```swift
UIView.animate(withDuration: 5) {
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}, completion: nil
```
UIView.animate(withDuration: 5) {
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}, completion: nil)
UIViewPropertyAnimator
UIViewPropertyAnimator
Features

Custom timing
Interactive
Interruptible
Responsive
UIViewPropertyAnimator

![Diagram of UIViewPropertyAnimator with x-axis from 0 to 100]
let animator = UIViewPropertyAnimator(duration: 2.5, curve: .linear) {
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}
animator.startAnimation()
let animator = UIViewPropertyAnimator(duration: 2.5, curve: .linear) {
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}
animator.startAnimation()
let animator = UIViewPropertyAnimator(duration: 2.5, curve: .linear) {
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}
animator.startAnimation()
let animator = UIViewPropertyAnimator(duration: 2.5, curve: .linear) {
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}

animator.startAnimation()
let animator = UIViewPropertyAnimator(duration: 2.5, curve: .linear) {
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}

animator.startAnimation()
let animator = UIViewPropertyAnimator(duration: 2.5, curve: .linear) {
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}
animator.startAnimation()
Linear Curves

% Progress = % Time
Timing Curves

Ease In

Ease Out
Custom Curves

Custom Ease In

Progress

Time
Custom Curves

UI cubic timing parameters:
```
UICubicTimingParameters(controlPoint1: CGPoint(x: 0.75, y: 0.1),
controlPoint2: CGPoint(x: 0.9, y: 0.25))
```
Interactively Animating
Interactively animating
Interactively animating

UIPanGestureRecognizer
Interactively animating
Interactively animating
Interactively animating
Interactively animating
Interactively animating
Interactively animating
var animator: UIViewPropertyAnimator!

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animator = UIViewPropertyAnimator(duration: 1, curve: .easeOut, animations: {
            circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
        })
        animator.pauseAnimation()
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = translation.x / 100
    case .ended:
        animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)
    }
}
var animator: UIViewPropertyAnimator!

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animator = UIViewPropertyAnimator(duration: 1, curve: .easeOut, animations: {
            circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
        })
        animator.pauseAnimation()
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = translation.x / 100
        case .ended:
        animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)
    }
}
var animator: UIViewPropertyAnimator!

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animator = UIViewPropertyAnimator(duration: 1, curve: .easeOut, animations: {
            circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
        })
        animator.pauseAnimation()
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = translation.x / 100
    case .ended:
        animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)
    }
}
var animator: UIViewPropertyAnimator!

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animator = UIViewPropertyAnimator(duration: 1, curve: .easeOut, animations: {
            circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
        })
        animator.pauseAnimation()
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = translation.x / 100
    case .ended:
        animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)
    }
}
var animator: UIViewPropertyAnimator!

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animator = UIViewPropertyAnimator(duration: 1, curve: .easeOut, animations: {
            circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
        })
        animator.pauseAnimation()
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = translation.x / 100
    case .ended:
        animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)
    }
}
var animator: UIViewPropertyAnimator!

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animator = UIViewPropertyAnimator(duration: 1, curve: .easeOut, animations: {
            circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
        })
        animator.pauseAnimation()
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = translation.x / 100
    case .ended:
        animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)
    }
}

Time Conversion

Pausing

Continuing
Time Conversion

Pausing

Continuing
UIViewPropertyAnimator(duration: 1, curve: .easeOut)

- animationState: .inactive
- running: false
- fractionComplete: 0%

Graph showing:
- Progress vs. Time
- A curve indicating the progress over time
`animator.pauseAnimation()`

- **animationState**: `.active`
- **running**: `false`
- **fractionComplete**: `0%`

![Graph](image)
`Animator.pauseAnimation();`

**animationState**
- `active`: false

**fractionComplete**: 0%

![Graph showing animation state and fraction complete](image)
Time Conversion

Pausing

Continuing
animator.fractionComplete = translation.x / distance
$$\text{animator.fractionComplete} = \frac{\text{translation.x}}{\text{distance}}$$
The animation state is `active` and `false` for running, indicating the animation is not running. The fraction complete is 50%. The diagram shows a graph with time on the x-axis and progress on the y-axis, with a point at 50% progress.
`animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)`
` animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)`

- **animationState**: `.active`
- **running**: `true`
- **fractionComplete**: `50%`

Graph showing progress over time with a point at `50%` completion.
Progress

Time

animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)

animationState
.active

running
true

fractionComplete
10%

Progress

Time
Animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)

- animationState: active
- running: true
- fractionComplete: 10%

Progress

Time
Progress

Time

90%

`Animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)`
`Animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)`

Duration 2 seconds

![Progress vs Time Graph]

- Progress: 0.5
- Time: 0.9
- Progress Percentage: 90%
animator.continueAnimation(withTimingParameters: nil, durationFactor: 0)

Duration 2 seconds

Progress

Time

Remaining time 1.8 seconds
Interruptible Animations
iPhone
This is 7.
iPhone
This is 7.
iPhone
This is 7.
iPhone
This is 7.
iPhone
This is 7.
iPhone

This is 7.
Interrupting an Animation
Interrupting an Animation

- UIPanGestureRecognizer
Interrupting an Animation
Interrupting an Animation
Interrupting an Animation
Interrupting an Animation
Interrupting an Animation
Interrupting an Animation
func animateTransitionIfNeeded(duration: TimeInterval) {...}

var progressWhenInterrupted: CGFloat = 0

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animateTransitionIfNeeded(duration: 1)
        animator.pauseAnimation()
        progressWhenInterrupted = animator.fractionComplete
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = (translation.x / 100) + progressWhenInterrupted
    case .ended:
        let timing = UICubicTimingParameters(animationCurve: .easeOut)
        animator.continueAnimation(withTimingParameters: timing, durationFactor: 0)
    }
}
```swift
func animateTransitionIfNeeded(duration: TimeInterval) {
}

var progressWhenInterrupted: CGFloat = 0

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animateTransitionIfNeeded(duration: 1)
        animator.pauseAnimation()
        progressWhenInterrupted = animator.fractionComplete
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = (translation.x / 100) + progressWhenInterrupted
    case .ended:
        let timing = UICubicTimingParameters(animationCurve: .easeOut)
        animator.continueAnimation(withTimingParameters: timing, durationFactor: 0)
    }
}
```
func animateTransitionIfNeeded(duration: TimeInterval) {...}

var progressWhenInterrupted: CGFloat = 0

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animateTransitionIfNeeded(duration: 1)
        animator.pauseAnimation()
        progressWhenInterrupted = animator.fractionComplete
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = (translation.x / 100) + progressWhenInterrupted
    case .ended:
        let timing = UICubicTimingParameters(animationCurve: .easeOut)
        animator.continueAnimation(withTimingParameters: timing, durationFactor: 0)
    }
}
func animateTransitionIfNeeded(duration: TimeInterval) {...}

var progressWhenInterrupted: CGFloat = 0

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animateTransitionIfNeeded(duration: 1)
        animator.pauseAnimation()
        progressWhenInterrupted = animator.fractionComplete
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = (translation.x / 100) + progressWhenInterrupted
    case .ended:
        let timing = UICubicTimingParameters(animationCurve: .easeOut)
        animator.continueAnimation(withTimingParameters: timing, durationFactor: 0)
    }
}
func animateTransitionIfNeeded(duration: TimeInterval) {...}

var progressWhenInterrupted: CGFloat = 0

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animateTransitionIfNeeded(duration: 1)
        animator.pauseAnimation()
        progressWhenInterrupted = animator.fractionComplete
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = (translation.x / 100) + progressWhenInterrupted
    case .ended:
        let timing = UICubicTimingParameters(animationCurve: .easeOut)
        animator.continueAnimation(withTimingParameters: timing, durationFactor: 0)
    }
}
func animateTransitionIfNeeded(duration: TimeInterval) {...}

var progressWhenInterrupted: CGFloat = 0

func handlePan(recognizer: UIPanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        animateTransitionIfNeeded(duration: 1)
        animator.pauseAnimation()
        progressWhenInterrupted = animator.fractionComplete
    case .changed:
        let translation = recognizer.translation(in: circle)
        animator.fractionComplete = (translation.x / 100) + progressWhenInterrupted
    case .ended:
        let timing = UICubicTimingParameters(animationCurve: .easeOut)
        animator.continueAnimation(withTimingParameters: timing, durationFactor: 0)
    }
}
Time Conversion

Pausing

Continuing
animator.isRunning

- animationState: .active
- running: false
- fractionComplete: 50%

Progress

Time
`animator.pauseAnimation()`
`animator.pauseAnimation()`

- `animationState.active` is `false`.
- `fractionComplete` is `50%`.

The diagram shows the progress of the animation against time, with the current fraction complete marked on the graph.
`animator.pauseAnimation()`
animeator.pauseAnimation()

animationState = .active
running = false
fractionComplete = 10%

Progress vs Time graph:
- Progress axis ranges from 0.0 to 1.0
- Time axis ranges from 0.0 to 1.0
- The graph shows a linear and a nonlinear progression of time with respect to progress.
animator.fractionComplete = 0.1

animationState
.active
false

running
false

fractionComplete
10%
`animat.or.continueAnimation(... animationCurve: .easeOut ...)`

![Graph showing progress over time with an easeOut curve.](image-url)
`Animator.continueAnimation(...) animationCurve: .easeOut ...)`

**Performance Metrics:**
- **animationState**: `.active`
- **running**: `true`
- **fractionComplete**: `10%`

**Diagram:**
- **Progress** vs **Time** graph showing the animation curve with a point indicating 10% completion.
```
@State
var animationState: AnimationState = .active

@State
var running = true

@State
var fractionComplete = 0.0
```

```
animator.continueAnimation(... animationCurve: .easeOut ...)
```
animator.continueAnimation(... animationCurve: .easeOut ...)

- animationState: .active
- running: true
- fractionComplete: 5%

![Graph showing animation progress and time]
New Animator Behaviors
UIViewPropertyAnimator
New in iOS 11

var scrubsLinearly: Bool
var pausesOnCompletion: Bool
UIViewPropertyAnimator
New in iOS 11

var scrubsLinearly: Bool
var pausesOnCompletion: Bool

Starting as Paused
Linear scrubbing

Non-linear scrubbing
Non-linear scrubbing

- Linear scrubbing
- Non-linear scrubbing
Non-linear scrubbing

linear scrubbing

non-linear scrubbing
Non-linear scrubbing
The diagram shows an animation state machine with two states: `.Inactive` and `.Active`. When the animation finishes, it transitions to the `.Active` state. If the `pausesOnCompletion` property of the animator is set to `true`, the animation will pause on completion. The code snippet `animator.pausesOnCompletion = true` indicates that this property is set to `true`. The diagram illustrates the transition with an arrow labeled `Start / pause` and a circle arrow labeled `Animations finish`. The state `NEW` is also indicated at the top right of the diagram.
Manhattanhenge 2017: Where and when to view New York City's most striking sunset of the year

The event happens only twice a year and attracts thousands of spectators. A view of the ‘Manhattanhenge’ sunset from Hunters Point South Park in Queens, New York (Drew Angerer/Getty Images) This summer, bustling New York...
Manhattanhenge 2017: Where and when to view New York City's most striking sunset of the year

The event happens only twice a year and attracts thousands of spectators. A view of the 'Manhattanhenge' sunset from Hunters Point South Park in Queens, New York (Drew Angerer/Getty Images) This summer, bustling New York...
Manhattanhenge 2017: Where and when to view New York City's most striking sunset of the year

The event happens only twice a year and attracts thousands of spectators. A view of the 'Manhattanhenge' sunset from Hunter's Point South Park in Queens, New York (Drew Angerer/Getty Images) This summer, bustling New York...
Manhattanhenge 2017: Where and when to view New York City’s most striking sunset of the year

The event happens only twice a year and attracts thousands of spectators. A view of the ‘Manhattanhenge’ sunset from Hunters Point South Park in Queens, New York (Drew Angerer/Getty Images). This summer, bustling New York...
Manhattanhenge 2017: Where and when to view New York City's most striking sunset of the year

The event happens only twice a year and attracts thousands of spectators. A view of the 'Manhattanhenge' sunset from Hunters Point South Park in Queens, New York (Drew Angerer/Getty Images) This summer, bustling New York...
Manhattanhenge 2017: Where and when to view New York City's most striking sunset of the year

The event happens only twice a year and attracts thousands of spectators. A view of the 'Manhattanhenge' sunset from Hunters Point South Park in Queens, New York (Drew Angerer/Getty Images) This summer, bustling New York...
Manhattanhenge 2017: Where and when to view New York City's most striking sunset of the year

The event happens only twice a year and attracts thousands of spectators. A view of the 'Manhattanhenge' sunset from Hunters Point South Park in Queens, New York (Drew Angerer/Getty Images). This summer, bustling New York...
Manhattanhenge 2017: Where and when to view New York City's most striking sunset of the year

The event happens only once a year and attracts thousands of spectators. A view of the 'Manhattanhenge' sunset from Hunters Point South Park in Queens, New York. (Drew Angerer/Getty Images) This summer, bustling New York...
.pausesOnCompletion

Inactive

Start / pause

Animations finish

Active
.pausesOnCompletion

```
 animator.addObserver(self, forKeyPath: "running", options: [.new], context: nil)
```
let animator = UIViewPropertyAnimator(duration: 1, curve: .easeIn)
animator.startAnimation()
// ...
animator.addAnimations {
    // will run immediately
    circle.frame = circle.frame.offsetBy(dx: 100, dy: 0)
}
Springs
Spring Animations

Critically damped spring

Under damped spring
Spring Animations

Critically damped spring

Under damped spring
Spring Animations

Critically damped spring

Under damped spring
Spring Animations

Critically damped spring

Under damped spring
Spring Animations

Critically damped spring

Under damped spring
Spring Animations

Critically damped spring

Under damped spring
Spring Animations

Critically damped spring
Damping ratio = 1.0

Under damped spring
Damping ratio < 1.0
Spring Animations
Why they always animate from current state
Spring Animations
Why they always animate from current state

Remapping onto cubic may be undefined
Spring Animations
Why they always animate from current state

2D velocity desynchronization
Spring Animations
Why they always animate from current state

2D velocity desynchronization
Best Practices When Interrupting Springs
Best Practices When Interrupting Springs

Stop and create a new property animator
Best Practices When Interrupting Springs

Stop and create a new property animator

Use critically damped spring without velocity
Best Practices When Interrupting Springs

Stop and create a new property animator

Use critically damped spring without velocity

Decompose component velocity with multiple animators
Coordinating Animations
Overview

Build a fully interactive, interruptible animated transition
Coordinate across multiple uniquely timed animators
// Tracks all running animators
var runningAnimators = [UIViewPropertyAnimator]()

// Perform all animations with animators if not already running
func animateTransitionIf Needed(state: State, duration: TimeInterval) { ... }

// Starts transition if necessary or reverses it on tap
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) { ... }

// Starts transition if necessary and pauses on pan .begin
func startInteractiveTransition(state: State, duration: TimeInterval) { ... }

// Scrubs transition on pan .changed
func updateInteractiveTransition(fractionComplete: CGFloat) { ... }

// Continues or reverse transition on pan .ended
func continueInteractiveTransition(cancel: Bool) { ... }
// Tracks all running animators

```swift
var runningAnimators = [UIViewPropertyAnimator]()
```

// Perform all animations with animators if not already running

```swift
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
...
}
```

// Starts transition if necessary or reverses it on tap

```swift
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
...
}
```

// Starts transition if necessary and pauses on pan .begin

```swift
func startInteractiveTransition(state: State, duration: TimeInterval) {
...
}
```

// Scrubs transition on pan .changed

```swift
func updateInteractiveTransition(fractionComplete: CGFloat) {
...
}
```

// Continues or reverse transition on pan .ended

```swift
func continueInteractiveTransition(cancel: Bool) {
...
}
```
// Tracks all running animators
var runningAnimators = [UIViewPropertyAnimator]()

// Perform all animations with animators if not already running
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
 ...
}

// Starts transition if necessary or reverses it on tap
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
 ...
}

// Starts transition if necessary and pauses on pan .begin
func startInteractiveTransition(state: State, duration: TimeInterval) {
 ...
}

// Scrubs transition on pan .changed
func updateInteractiveTransition(fractionComplete: CGFloat) {
 ...
}

// Continues or reverse transition on pan .ended
func continueInteractiveTransition(cancel: Bool) {
 ...
}
// Perform all animations with animators if not already running
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
    if runningAnimators.isEmpty {
        let frameAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
            switch state {
            case .Expanded:
                self.control.frame = CGRect(...)
            case .Collapsed:
                self.control.frame = CGRect(...)
            }
        }
        frameAnimator.startAnimation()
        runningAnimators.append(frameAnimator)
    }
}
// Perform all animations with animators if not already running
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
    if runningAnimators.isEmpty {
        let frameAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
            switch state {
            case .Expanded:
                self.control.frame = CGRect(...)
            case .Collapsed:
                self.control.frame = CGRect(...)
            }
        }
        frameAnimator.startAnimation()
        runningAnimators.append(frameAnimator)
    }
}
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
    if runningAnimators.isEmpty {
        let frameAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
            switch state {
            case .Expanded:
                self.control.frame = CGRect(...)
            case .Collapsed:
                self.control.frame = CGRect(...)
            }
        }
        frameAnimator.startAnimation()
        runningAnimators.append(frameAnimator)
    }
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
    if runningAnimators.isEmpty {
        let frameAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
            switch state {
            case .Expanded:
                self.control.frame = CGRect(...)
            case .Collapsed:
                self.control.frame = CGRect(...)
            }
        }
        frameAnimator.startAnimation()
        runningAnimators.append(frameAnimator)
    }
}
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
    if runningAnimators.isEmpty {
        let frameAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
            switch state {
            case .Expanded:
                self.control.frame = CGRect(...)
            case .Collapsed:
                self.control.frame = CGRect(...)
            }
        }
        frameAnimator.startAnimation()
        runningAnimators.append(frameAnimator)
    }
}

// Perform all animations with animators if not already running
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
// Perform all animations with animators if not already running
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
// Tracks all running animators
var runningAnimators = [UIViewPropertyAnimator]()

// Perform all animations with animators if not already running
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
...
}

// Starts transition if necessary or reverses it on tap
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
...
}

// Starts transition if necessary and pauses on pan .begin
func startInteractiveTransition(state: State, duration: TimeInterval) {
...
}

// Scrubs transition on pan .changed
func updateInteractiveTransition(fractionComplete: CGFloat) {
...
}

// Continues or reverse transition on pan .ended
func continueInteractiveTransition(cancel: Bool) {
...
}
// Tracks all running animators
var runningAnimators = [UIViewPropertyAnimator]()  

// Perform all animations with animators if not already running
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
...
}

// Starts transition if necessary or reverses it on tap
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
...
}

// Starts transition if necessary and pauses on pan .begin
func startInteractiveTransition(state: State, duration: TimeInterval) {
...
}

// Scrubs transition on pan .changed
func updateInteractiveTransition(fractionComplete: CGFloat) {
...
}

// Continues or reverse transition on pan .ended
func continueInteractiveTransition(cancel: Bool) {
...
}
// Starts transition if necessary or reverses it on tap
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
// Starts transition if necessary or reverses it on tap

func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
    if runningAnimators.isEmpty {
        animateTransitionIf Needed(state: state, duration: duration)
    } else {
        for animator in runningAnimators {
            animator.isReversed = !animator.isReversed
        }
    }
}
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
    if runningAnimators.isEmpty {
        animateTransitionIfNeeded(state: state, duration: duration)
    } else {
        for animator in runningAnimators {
            animator.isReversed = !animator.isReversed
        }
    }
}
// Starts transition if necessary or reverses it on tap

func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
    if runningAnimators.isEmpty {
        animateTransitionIfNeeded(state: state, duration: duration)
    } else {
        for animator in runningAnimators {
            animator.isReversed = !animator.isReversed
        }
    }
}
// Starts transition if necessary or reverses it on tap

func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
// Starts transition if necessary or reverses it on tap
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
Tracks all running animators

```swift
var runningAnimators = [UIViewPropertyAnimator]()
```

Perform all animations with animators if not already running

```swift
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
    ... }
```

Starts transition if necessary or reverses it on tap

```swift
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
    ... }
```

Starts transition if necessary and pauses on pan `.begin`

```swift
func startInteractiveTransition(state: State, duration: TimeInterval) {
    ... }
```

Scrubs transition on pan `.changed`

```swift
func updateInteractiveTransition(fractionComplete: CGFloat) {
    ... }
```

Continues or reverse transition on pan `.ended`

```swift
func continueInteractiveTransition(cancel: Bool) {
    ... }
```
// Tracks all running animators
var runningAnimators = [UIViewPropertyAnimator]()

// Perform all animations with animators if not already running
func animateTransitionIfNeeded(state: State, duration: TimeInterval) { ... }

// Starts transition if necessary or reverses it on tap
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) { ... }

// Starts transition if necessary and pauses on pan .begin
func startInteractiveTransition(state: State, duration: TimeInterval) { ... }

// Scrubs transition on pan .changed
func updateInteractiveTransition(fractionComplete: CGFloat) { ... }

// Continues or reverse transition on pan .ended
func continueInteractiveTransition(cancel: Bool) { ... }
// Tracks all running animators
var runningAnimators = [UIViewPropertyAnimator]()

// Perform all animations with animators if not already running
func animateTransitionIfNeeded(state: State, duration: TimeInterval) {
    ... }

// Starts transition if necessary or reverses it on tap
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) {
    ... }

// Starts transition if necessary and pauses on pan .begin
func startInteractiveTransition(state: State, duration: TimeInterval) {
    ... }

// Scrubs transition on pan .changed
func updateInteractiveTransition(fractionComplete: CGFloat) {
    ... }

// Continues or reverse transition on pan .ended
func continueInteractiveTransition(cancel: Bool) {
    ... }
// Tracks all running animators
var runningAnimators = [UIViewPropertyAnimator]()

// Perform all animations with animators if not already running
func animateTransitionIfNeeded(state: State, duration: TimeInterval) { ... }

// Starts transition if necessary or reverses it on tap
func animateOrReverseRunningTransition(state: State, duration: TimeInterval) { ... }

// Starts transition if necessary and pauses on pan .begin
func startInteractiveTransition(state: State, duration: TimeInterval) { ... }

// Scrubs transition on pan .changed
func updateInteractiveTransition(fractionComplete: CGFloat) { ... }

// Continues or reverse transition on pan .ended
func continueInteractiveTransition(cancel: Bool) { ... }
Animating a Blur
class UIVisualEffectView: UIView {
    var effect: UIVisualEffect // animatable
}

class UIBlurEffect: UIVisualEffect {
    init(style: UIBlurEffectStyle)
}

class UIVibrancyEffect: UIVisualEffect {
    init(blurEffect: UIBlurEffect)
}
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let blurAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
        switch state {
        case .Expanded:
            self.blurEffectView.effect = UIBlurEffect(style: .dark)
        }
        case .Collapsed:
            self.blurEffectView.effect = nil
        }
    }
    blurAnimator.startAnimation()
    runningAnimators.append(blurAnimator)
    // ...
}
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let blurAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
        switch state {
        case .Expanded:
            self.blurEffectView.effect = UIBlurEffect(style: .dark)
        }
        case .Collapsed:
            self.blurEffectView.effect = nil
        }
    }
    blurAnimator.startAnimation()
    runningAnimators.append(blurAnimator)
    // ...
}
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let blurAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
        switch state {
        case .Expanded:
            self.blurEffectView.effect = UIBlurEffect(style: .dark)
        }
        case .Collapsed:
            self.blurEffectView.effect = nil
        }
    }
    blurAnimator.startAnimation()
    runningAnimators.append(blurAnimator)
    // ...
}
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let blurAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
        switch state {
            case .Expanded:
                self.blurEffectView.effect = UIBlurEffect(style: .dark)
            case .Collapsed:
                self.blurEffectView.effect = nil
        }
    }
    blurAnimator.startAnimation()
    runningAnimators.append(blurAnimator)
    // ...
}

Issues
Issues

Too fast animating in
Still too fast animating in / out
Custom Timing
Symmetric pacing

Custom Ease In
Animates our blur in slowly
\((0.75, 0.1) \ (0.9, 0.25)\)

Custom Ease Out
Animates our blur out quickly
\((0.1, 0.75) \ (0.25, 0.9)\)
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    let timing: UITimingCurveProvider
    switch state {
    case .Expanded:
        timing = UICubicTimingParameters(controlPoint1: CGPoint(x: 0.75, y: 0.1),
                                           controlPoint2: CGPoint(x: 0.9, y: 0.25))
    case .Collapsed:
        timing = UICubicTimingParameters(controlPoint1: CGPoint(x: 0.1, y: 0.75),
                                           controlPoint2: CGPoint(x: 0.25, y: 0.9))
    }
    let blurAnimator = UIViewPropertyAnimator(duration: duration, timingParameters: timing)
    blurAnimator.scrubsLinearly = false
    // ...
}
View Morphing

Hello
Hello
View Morphing

Scaling, translation, and opacity blending of two views
This is a fresh photo. Nice kicks.

Your ideal sneaker. Hard wearing sole and upper. Most shoes only offer one.
Strategy
Strategy

.transform: CGAffineTransform

Compute transform.scale and transform.translation

Prepare views and animate .transform and .alpha
Computing Scale

\[
\text{.scale.width} = \frac{W}{w}
\]

\[
\text{.scale.height} = \frac{H}{h}
\]

\[
\frac{W}{w} \quad \frac{h}{H}
\]
Computing Scale

\[
\frac{w}{h} \quad \text{Hello}
\]

\[
\text{.scale.width} = \frac{W}{w}
\]

\[
\text{.scale.height} = \frac{H}{h}
\]

\[
\frac{W}{w} = \frac{1}{\text{.scale.width}}
\]

\[
\frac{h}{H} = \frac{1}{\text{.scale.height}}
\]
Computing Translation

Hello

Hello
Computing Translation

\[ \text{translation.y} = Y - y' \]
Animations

- Critically Damped Spring
- `.easeIn`
- `.easeOut`

Graphs showing progress and time for different animation effects. The graphs illustrate the behavior of each effect over time, with labels indicating `transform`, `Incoming view.alpha`, and `Outgoing view.alpha` Non linear scrubbing.
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let transformAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
        inLabel.transform = CGAffineTransform.identity
        outLabel.transform = inLabelScale.concatenating(inLabelTranslation)
    }
    // ...
    let inLabelAnimator = UIViewPropertyAnimator(duration: duration, curve: .easeIn) {
        inLabel.alpha = 1
    }
    inLabelAnimator.scrubsLinearly = false
    // ...
    let outLabelAnimator = UIViewPropertyAnimator(duration: duration, curve: .easeOut) {
        outLabel.alpha = 0
    }
    outLabelAnimator.scrubsLinearly = false
    // ...
}
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let transformAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
        inLabel.transform = CGAffineTransform.identity
        outLabel.transform = inLabelScale.concatenating(inLabelTranslation)
    }
    // ...
    let inLabelAnimator = UIViewPropertyAnimator(duration: duration, curve: .easeIn) {
        inLabel.alpha = 1
    }
    inLabelAnimator.scrubsLinearly = false
    // ...
    let outLabelAnimator = UIViewPropertyAnimator(duration: duration, curve: .easeOut) {
        outLabel.alpha = 0
    }
    outLabelAnimator.scrubsLinearly = false
    // ...
}
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let transformAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
        inLabel.transform = CGAffineTransform.identity
        outLabel.transform = inLabelScale.concatenating(inLabelTranslation)
    }
    // ...
    let inLabelAnimator = UIViewPropertyAnimator(duration: duration, curve: .easeIn) {
        inLabel.alpha = 1
    }
    inLabelAnimator.scrubsLinearly = false
    // ...
    let outLabelAnimator = UIViewPropertyAnimator(duration: duration, curve: .easeOut) {
        outLabel.alpha = 0
    }
    outLabelAnimator.scrubsLinearly = false
    // ...
}
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let transformAnimator = UIViewPropertyAnimator(duration: duration, dampingRatio: 1) {
        inLabel.transform = CGAffineTransform.identity
        outLabel.transform = inLabelScale.concatenating(inLabelTranslation)
    }
    // ...
    let inLabelAnimator = UIViewPropertyAnimator(duration: duration, curve: .easeIn) {
        inLabel.alpha = 1
    }
    inLabelAnimator.scrubsLinearly = false
    // ...
    let outLabelAnimator = UIViewPropertyAnimator(duration: duration, curve: .easeOut) {
        outLabel.alpha = 0
    }
    outLabelAnimator.scrubsLinearly = false
    // ...
}
Tips and Tricks
Animating Corner Radius
This is a fresh photo. Nice kicks.

Your ideal sneaker. Hard wearing sole and upper. Most shoes only offer one.
.cornerRadius
Now animatable in UIKit

CALayer
var .cornerRadius: CGFloat
.cornerRadius
Now animatable in UIKit

cALayer

`var .cornerRadius: CGFloat`

circle.clipsToBounds = true
UIViewPropertyAnimator(duration: 1, curve: .linear) {
    circle.layer.cornerRadius = 12
}.startAnimation()
This is a fresh photo. Nice kicks.

Your ideal sneaker. Hard wearing sole and upper. Most shoes only offer one.
This is a fresh photo. Nice kicks.

Your ideal sneaker. Hard wearing sole and upper. Most shoes only offer one.
New in iOS 11

**.maskedCorners**

```
var .maskedCorners: CACornerMask
```
New in iOS 11

**.maskedCorners**

```swift
var .maskedCorners: CACornerMask
```
New in iOS 11

**CALayer**

```swift
var .maskedCorners: CACornerMask
```

```swift
circle.layer.maskedCorners = [.layerMinXMinYCorner, .layerMaxXMinYCorner]
```
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let cornerAnimator = UIViewPropertyAnimator(duration: duration, curve: .linear) {
        switch state {
        case .Expanded:
            self.control.layer.cornerRadius = 12
        case .Collapsed:
            self.control.layer.cornerRadius = 0
        }
    }
    // ...
}
Component Timing

![Graph showing Component Timing with axes labeled Progress and Time]
Component Timing

![Component Timing Diagram]
Component Timing

![Graph showing component timing with axes labeled Progress and Time]

- Progress range: 0.0 to 1.0
- Time range: 0.0 to 1.0

The graph illustrates the relationship between component timing and progress over time.
Component Timing

Progress

Time
Component Timing
Keyframe Animations
UIView

```swift
func animateKeyframes(withDuration duration: TimeInterval, delay: TimeInterval,
                      options: ..., animations: ..., completion: ...)

func addKeyframe(withRelativeStartTime frameStartTime: Double,
                  relativeDuration frameDuration: Double,
                  animations: ...)```
**Keyframe Animations**

```swift
UIView

func animateKeyframes(withDuration duration: TimeInterval, delay: TimeInterval,
    options: ..., animations: ..., completion: ...)

func addKeyframe(withRelativeStartTime frameStartTime: Double,
    relativeDuration frameDuration: Double,
    animations: ...)```
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let buttonAnimator = UIViewPropertyAnimator(duration: duration, curve: .linear) {
        UIView.animateKeyframes(withDuration: 0.0, delay: 0.0, options: [], animations: {
            switch state {
            case .Expanded:
                UIView.addKeyframe(withRelativeStartTime: 0.5, relativeDuration: 0.5) {
                    // Start with delay and finish with rest of animations
                    detailsButton.alpha = 1
                }
            case .Collapsed:
                UIView.addKeyframe(withRelativeStartTime: 0.0, relativeDuration: 0.5) {
                    // Start immediately and finish in half the time
                    detailsButton.alpha = 0
                }
            }
        }, completion: nil)
    }
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let buttonAnimator = UIViewPropertyAnimator(duration: duration, curve: .linear) {
        UIView.animateKeyframes(withDuration: 0.0, delay: 0.0, options: [], animations: {
            switch state {
            case .Expanded:
                UIView.addKeyframe(withRelativeStartTime: 0.5, relativeDuration: 0.5) {
                    // Start with delay and finish with rest of animations
                    detailsButton.alpha = 1
                }
            }
            case .Collapsed:
                UIView.addKeyframe(withRelativeStartTime: 0.0, relativeDuration: 0.5) {
                    // Start immediately and finish in half the time
                    detailsButton.alpha = 0
                }
            }
        }, completion: nil)
    }
```swift
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let buttonAnimator = UIViewPropertyAnimator(duration: duration, curve: .linear) {
        UIView.animateKeyframes(withDuration: 0.0, delay: 0.0, options: [], animations: {
            switch state {
            case .Expanded:
                UIView.addKeyframe(withRelativeStartTime: 0.5, relativeDuration: 0.5) {
                    // Start with delay and finish with rest of animations
                    detailsButton.alpha = 1
                }
            case .Collapsed:
                UIView.addKeyframe(withRelativeStartTime: 0.0, relativeDuration: 0.5) {
                    // Start immediately and finish in half the time
                    detailsButton.alpha = 0
                }
            }
        }, completion: nil)
    }
}
```
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let buttonAnimator = UIViewPropertyAnimator(duration: duration, curve: .linear) {
        UIView.animateKeyframes(withDuration: 0.0, delay: 0.0, options: [], animations: {
            switch state {
            case .Expanded:
                UIView.addKeyframe(withRelativeStartTime: 0.5, relativeDuration: 0.5) {
                    // Start with delay and finish with rest of animations
                    detailsButton.alpha = 1
                }
            case .Collapsed:
                UIView.addKeyframe(withRelativeStartTime: 0.0, relativeDuration: 0.5) {
                    // Start immediately and finish in half the time
                    detailsButton.alpha = 0
                }
            }
        }, completion: nil)
    }
func animateTransitionIfNeeded(forState state: State, duration: TimeInterval) {
    // ...
    let buttonAnimator = UIViewPropertyAnimator(duration: duration, curve: .linear) {
        UIView.animateKeyframes(withDuration: 0.0, delay: 0.0, options: [], animations: {
            switch state {
            case .Expanded:
                UIView.addKeyframe(withRelativeStartTime: 0.5, relativeDuration: 0.5) {
                    // Start with delay and finish with rest of animations
                    detailsButton.alpha = 1
                }
            case .Collapsed:
                UIView.addKeyframe(withRelativeStartTime: 0.0, relativeDuration: 0.5) {
                    // Start immediately and finish in half the time
                    detailsButton.alpha = 0
                }
            }
        }, completion: nil)
    }
Additive Animations
Additive Animations
Additive Animations
Additive Animations

let animator = UIViewPropertyAnimator(duration: 5, curve: .easeInOut) {
    square.transform = CGAffineTransform(rotationAngle: CGFloat(Double.pi * 20))
}
animator.startAnimation()
CGAffineTransform(rotationAngle: CGFloat(Double.pi * 20))
CGAffineTransform(rotationAngle: CGFloat(-Double.pi))
CGAffineTransform(rotationAngle: CGFloat(-Double.pi))
CGAffineTransform(rotationAngle: CGFloat(-Double.pi))
CGAffineTransform(rotationAngle: CGFloat(-Double.pi))
Options

Use Core Animation
  • Low level
  • No scrubbing
Options

Use Core Animation
• Low level
• No scrubbing

Decompose into several smaller additive rotation animations
Additively Animatable Properties

```swift
var transform: CGAffineTransform // affine only
var frame: CGRect
var bounds: CGRect
var center: CGPoint
var position: CGPoint
```
Decomposed Additive Animations

```swift
let animator = UIViewPropertyAnimator(duration: 5, curve: .easeInOut, animations: {
    for _ in 0..<20 {
        let rotation = CGAffineTransform(rotationAngle: CGFloat(Double.pi))
        square.transform = square.transform.concatenating(rotation)
    }
})
animator.startAnimation()
```
let animator = UIViewPropertyAnimator(duration: 5, curve: .easeInOut, animations: {
    for _ in 0..<20 {
        let rotation = CGAffineTransform(rotationAngle: CGFloat(Double.pi))
        square.transform = square.transform.concatenating(rotation)
    }
})

animator.startAnimation()
Summary

Modern methods for making animations interactive and interruptible.

Coordinating several animations during interactive transition.
<table>
<thead>
<tr>
<th>Session</th>
<th>Room</th>
<th>Day</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>What's New in Cocoa Touch</td>
<td>Hall 3</td>
<td>Tuesday</td>
<td>10:20AM</td>
</tr>
<tr>
<td>Introducing Drag and Drop</td>
<td>Hall 3</td>
<td>Tuesday</td>
<td>11:20AM</td>
</tr>
<tr>
<td>Mastering Drag and Drop</td>
<td>Executive Ballroom</td>
<td>Wednesday</td>
<td>11:00AM</td>
</tr>
<tr>
<td>Modern User Interaction on iOS</td>
<td>Grand Ballroom B</td>
<td>Wednesday</td>
<td>4:10PM</td>
</tr>
<tr>
<td>Drag and Drop with Collection and Table View</td>
<td>Hall 2</td>
<td>Thursday</td>
<td>9:00AM</td>
</tr>
<tr>
<td>Data Delivery with Drag and Drop</td>
<td>Hall 2</td>
<td>Thursday</td>
<td>10:00AM</td>
</tr>
<tr>
<td>Session Title</td>
<td>Year</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------</td>
<td>--------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core Animation Essentials</td>
<td>WWDC 2011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Custom Transitions Using View Controllers</td>
<td>WWDC 2013</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Building Interruptible and Responsive Interactions</td>
<td>WWDC 2014</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advanced Graphics and Animations for iOS Apps</td>
<td>WWDC 2014</td>
<td></td>
<td></td>
</tr>
<tr>
<td>View Controller Advancements in iOS 8</td>
<td>WWDC 2014</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advances in UIKit Animations and Transitions</td>
<td>WWDC 2016</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Labs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cocoa Touch and Haptics Lab</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Technology Lab C</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Friday 12:00PM</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More Information
