Creating Custom Animated UIViews: Mastering Animations

Kristóf Kálai

January 1, 2024

Welcome to the world of custom UIViews and seamless animations! Have you ever seen those cool, smooth animations in apps and wondered how they're made? In this article, we'll delve into the art of creating the code that powers these captivating animations allowing you to quickly utilize eye-catching animations that make your apps look awesome.

The missing piece of information

This article assumes a basic understanding of Swift, UIKit, and the fundamental animation concepts. I'll guide you through building robust UIViews and CALayers that are both easy to implement and resilient to misuse. Whether you're a seasoned iOS developer seeking to refine your animation expertise or a newcomer eager to explore the realm of animated views, join me as we uncover the secrets behind crafting custom animated UIViews!

. . .

Within UIKit, numerous controls offer animatable properties that seamlessly integrate into animation contexts like UIView.animate(...). These properties dynamically update views during animations without requiring additional developer intervention.

However, how do we replicate this behavior for custom properties? 🤔

. . .

The article is divided into four parts:

- firstly, I'll demonstrate basic animation usage that we aim to mimic,
- secondly, we'll delve into replicating these styles in detail with practical demonstrations,
- next, we'll tackle a more complex example that includes some counterintuitive aspects, and
- finally, I provide a brief summary and notes for further exploration.

Note: all examples, including the final code, are based on Xcode 15.0 and Swift 5.9, with a minimum OS version is set to iOS 17.0. At all codeblocks there will be a GitHub link for that stage of code.

. . .

1 How to create a basic animation in UIKit? How did Apple make our work so easy?

The description of the animate method might seem concise (animate changes to one or more views using the specified duration), but let's delve deeper. When you utilize this method, it accepts a TimeInterval (a floating-point value representing the duration of the animation in seconds) and a closure that defines the final state of the animation. Yes, you read that correctly: there's an implicit starting state (the current visible state), and you need only specify explicitly the subsequent state - UIKit takes care of the rest. What a time to be alive! Moreover, you can refine it further by specifying a delay before the animation or customizing its timing curve. However, if our UIView works with the straightforward .animate(withDuration:animations:) method, then all other methods will work seamlessly as well.

Wondering how to use it? Let's craft a simple UIViewController housing a square in the center with a blue background. We'll animate its background color to green. Surprisingly, the animation can be accomplished in just three lines of code (link):

```
final class ViewController: UIViewController {
    private let subview: UIView = {
        let view = UIView()
        view.translatesAutoresizingMaskIntoConstraints = false
        view.backgroundColor = .blue
        view.heightAnchor.constraint(equalToConstant: 100).isActive = true
        view.widthAnchor.constraint(equalToConstant: 100).isActive = true
        return view
    }()

    override func viewDidLoad() {
        super.viewDidLoad()

        view.addSubview(subview)
        subview.centerYAnchor.constraint(equalTo: view.centerYAnchor).isActive = true
        subview.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true

        DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
            self.animate()
        }
    }

    private func animate() {
        UIView.animate(withDuration: 3) {
            self.subview.backgroundColor = .green
        }
    }
}
```

. . .

Let's say we're in a similar situation: we have a text in the square, and we want to animate its color. Like this (link):

```
final class ViewController: UIViewController {
    private let subview: UILabel = {
        let view = UILabel()
        view.translatesAutoresizingMaskIntoConstraints = false
        view.backgroundColor = .blue
        view.heightAnchor.constraint(equalToConstant: 100).isActive = true
        view.widthAnchor.constraint(equalToConstant: 100).isActive = true
        return view
    }()

    override func viewDidLoad() {
        super.viewDidLoad()

        view.addSubview(subview)
        subview.centerYAnchor.constraint(equalTo: view.centerYAnchor).isActive = true
        subview.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true

        subview.text = "TEST"
        subview.textColor = .red

        DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
            self.animate()
        }
    }

    private func animate() {
        UIView.animate(withDuration: 3) {
            self.subview.textColor = .green
```

```
        }
    }
}
```

It seems quite similar to what we've done before, but strangely enough, it's not functioning as expected. If only there were a way to mimic the previous behavior...

. . .

2 Part II — How to replicate the style of the animations made by Apple?

Let's proceed with the previous example where our aim was to change the color of the text within the square. Following thorough exploration, it becomes evident that the default UILabel does not support the animation of text color. So, what's the next step? We can mimic the public interface of the UILabel, or at least a part of it, and by implementing a custom UILabel that allows the color to be animated similarly to backgroundColor, we can tackle this issue. Since the core issue lies in UILabel utilizing a CALayer beneath its surface, I created a basic UILabel implementation leveraging a CATextLayer. This decision stems from the official documentation suggesting that the CATextLayer's foregroundColor property is indeed animatable. For the sake of simplicity, I opted not to copy UILabel's entire interface but rather made the layer's properties accessible from the external scope (link):

```
final class TextLabel: UIView {
    enum Text {
        case string(String)
        case attributedString(NSAttributedString)
        case none
    }

    var text: Text {
        get {
            if let attributedString = layer.string as? NSAttributedString {
                .attributedString(attributedString)
            } else if let string = layer.string as? String {
                .string(string)
            } else {
                .none
            }
        }
        set {
            switch newValue {
            case let .string(string): layer.string = string
            case let .attributedString(attributedString): layer.string = attributedString
            case .none: layer.string = nil
            }
        }
    }

    var font: UIFont? {
        get {
            func cast(_ cfString: CFString?) -> String? {
                if let cfString {
                    cfString as NSString as String
                } else {
                    nil
                }
            }

            let name: String? = {
                switch layer.font {
                case let ctFont as CTFont: cast(CTFontCopyName(ctFont, kCTFontPostScriptNameKey))
                case let cgFont as CGFont: cast(cgFont.postScriptName)
                case let string as NSString: string as String
                case let string as String: string
                default: nil
                }
            }()

            return name.flatMap { UIFont(name: $0, size: layer.fontSize) }
        }
        set {
            layer.font = newValue
        }
    }

    var fontSize: CGFloat {
```

```
        get {
            layer.fontSize
        }
        set {
            layer.fontSize = newValue
        }
    }

    var foregroundColor: UIColor? {
        get {
            layer.foregroundColor.map { .init(cgColor: $0) }
        }
        set {
            layer.foregroundColor = newValue?.cgColor
        }
    }

    var isWrapped: Bool {
        get {
            layer.isWrapped
        }
        set {
            layer.isWrapped = newValue
        }
    }

    var truncationMode: CATextLayerTruncationMode {
        get {
            layer.truncationMode
        }
        set {
            layer.truncationMode = newValue
        }
    }

    var alignmentMode: CATextLayerAlignmentMode {
        get {
            layer.alignmentMode
        }
        set {
            layer.alignmentMode = newValue
        }
    }

    var allowsFontSubpixelQuantization: Bool {
        get {
            layer.allowsFontSubpixelQuantization
        }
        set {
            layer.allowsFontSubpixelQuantization = newValue
        }
    }

    override class var layerClass: AnyClass {
        CATextLayer.self
    }

    override var layer: CATextLayer {
        super.layer as! CATextLayer
    }

    override init(frame: CGRect = .zero) {
        super.init(frame: frame)
        commonInit()
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        commonInit()
    }
}

extension TextLabel {
    private func commonInit() {
        layer.contentsScale = UIScreen.main.scale
    }
}
```

And with that:

```
final class ViewController: UIViewController {
    private let subview: TextLabel = {
        let view = TextLabel()
        view.translatesAutoresizingMaskIntoConstraints = false
        view.backgroundColor = .blue
        view.heightAnchor.constraint(equalToConstant: 100).isActive = true
        view.widthAnchor.constraint(equalToConstant: 100).isActive = true
        return view
    }()
```

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

        view.addSubview(subview)
        subview.centerYAnchor.constraint(equalTo: view.centerYAnchor).isActive = true
        subview.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true

        subview.text = .string("TEST")
        subview.foregroundColor = .red

        DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
            self.animate()
        }
    }

    private func animate() {
        UIView.animate(withDuration: 3) {
            self.subview.foregroundColor = .green
        }
    }
}
```

We'll try it out... but it doesn't work either. What's going on here? 🤯

. . .

Although the CATextLayer's foregroundColor is animatable, there seems to be a missing action associated with it. To address this and make it functional, we need to subclass the CATextLayer and provide a custom action implementation (link):

```
final private class TextLayer: CATextLayer {
    override func action(forKey event: String) -> CAAction? {
        switch event {
        case #keyPath(foregroundColor): // 0
            let context = action(forKey: #keyPath(backgroundColor)) as? CABasicAnimation // 1
            guard let animation = context?.copy() as? CABasicAnimation else { return nil } // 2

            animation.keyPath = event
            animation.fromValue = presentation()?.value(forKeyPath: event) // 3
            animation.toValue = nil
            return animation
        default:
            return super.action(forKey: event)
        }
    }
}

// ...

override class var layerClass: AnyClass {
    TextLayer.self
}
```

And it works! 🎉 But how, and more importantly, why?

Given that the CATextLayer does indeed support the animation of the foregroundColor, our task narrows down to one essential step — implementing the action for foregroundColor during animation. To achieve this, we:

- checked if there is a pending change to the foregroundColor (0),
- retrieved the current animation context (1) and generated an identical copy of that (2) to retain the current animation properties, and
- configured the desired properties, specifically the new foregroundColor (3).

  You may have noticed an unexpected appearance of backgroundColor during the process. This is due to the fact that handling animation contexts is Apple's private functionality. Thus, this is the only way for us to ascertain if an animation is taking place. As backgroundColor is a part of UIKit from the beginning, we have empirically determined that it returns a CABasicAnimation.

. . .

3 Part III — How to make use of it in the real world?

CAGradientLayer, another CALayer subclass, supports displaying gradient colors. While its colors and locations properties are straightforward to animate and manage, there's an interdependence between the startPoint and endPoint properties. In certain scenarios, it becomes essential to manage these independently, however, in most cases, a simple linear gradient is favored, where the angle serves as the primary source, allowing derivation of both startPoint and endPoint. To animate the angle (rotation) of such a gradient, we can craft a helper method coupled with additional functionalities (link):

```
extension CAGradientLayer {
    var location: [CGFloat]? {
        get {
            locations?.compactMap { CGFloat(exactly: $0) }
        }
        set {
            locations = newValue?.map { Double($0) }.map { NSNumber(value: $0) }
        }
    }

    var color: [UIColor]? {
        get {
            colors?.map { $0 as! CGColor }.compactMap { UIColor(cgColor: $0) }
        }
        set {
            colors = newValue?.map(\.cgColor)
        }
    }

    private func calculatePoints(degrees angle: CGFloat) -> (startPoint: CGPoint, endPoint: CGPoint) {
        let x = angle / 360
        let a = pow(sinf(Float(2 * .pi * ((x + 0.75) / 2))), 2)
        let b = pow(sinf(Float(2 * .pi * ((x + 0.00) / 2))), 2)
        let c = pow(sinf(Float(2 * .pi * ((x + 0.25) / 2))), 2)
        let d = pow(sinf(Float(2 * .pi * ((x + 0.50) / 2))), 2)

        return (CGPoint(x: CGFloat(a), y: CGFloat(b)), CGPoint(x: CGFloat(c), y: CGFloat(d)))
    }

    func apply(degrees angle: CGFloat /* .zero means top to bottom gradient */, on layer: CAGradientLayer? = nil) {
        let (startPoint, endPoint) = calculatePoints(degrees: angle)

        (layer ?? self).endPoint = endPoint
        (layer ?? self).startPoint = startPoint
    }
}

extension CALayer {
    var currentLayer: Self {
        presentation() ?? self
    }
}
```

It uses mathematical computations under the hood to calculate the startPoint and the endPoint based on the provided angle. Subsequently, we can create a layer that exposes an angle variable and a callback while handling its functionality behind the scenes. This functionality mirrors the previous example, but necessitates more caution due to the introduction of new properties (link):

- firstly, the new property must be marked with @NSManaged. This prevents the compiler from synthesizing the property and allows seamless cooperation with UIKit and CoreAnimation. In Objective-C, the @dynamic annotation should be used for a similar effect, although it differs from the dynamic keyword in Swift. In a nutshell, you must mark all animatable properties with @NSManaged,
- secondly, all three initializers must be implemented. While the latter two can be left empty, the first initializer, which accepts a layer as an argument, should copy the properties of the parameter to itself,
- thirdly, the needsDisplay(forKey:) method should return true if the key matches with our new property, and
- lastly, besides implementing the action(forKey:) similarly to the previous example, the draw(in:) method needs to be overridden, in order to actually draw the current angle, that is, the startPoint and endPoint, on the layer. Here any additional work can be done, e.g. calling the angleDidUpdate callback, but be aware that this method will be invoked for every frame rendering.

```
final class GradientLayer: CAGradientLayer {
    @NSManaged var angleInDegrees: CGFloat
    var angleDidUpdate: ((Angle) -> Void)?

    override init(layer: Any) {
        super.init(layer: layer)
        if let layer = layer as? Self {
            angleInDegrees = layer.angleInDegrees
            angleDidUpdate = layer.angleDidUpdate
        }
    }

    override init() {
        super.init()
    }

    required init?(coder: NSCoder) {
        super.init(coder: coder)
    }

    override class func needsDisplay(forKey key: String) -> Bool {
        key == #keyPath(angleInDegrees) ? true : super.needsDisplay(forKey: key)
    }

    override func action(forKey event: String) -> CAAction? {
        switch event {
        case #keyPath(angleInDegrees):
            let context = action(forKey: #keyPath(backgroundColor)) as? CABasicAnimation
            guard let animation = context?.copy() as? CABasicAnimation else { return nil }

            animation.keyPath = event
            animation.fromValue = presentation()?.value(forKeyPath: event)
            animation.toValue = nil
            return animation
        default:
            return super.action(forKey: event)
        }
    }

    override func draw(in ctx: CGContext) {
        super.draw(in: ctx)
        apply(degrees: currentLayer.angleInDegrees, on: model())
        angleDidUpdate?(.degrees(currentLayer.angleInDegrees))
    }
}
```

Then, since this is only a lightweight layer, we can construct a UIView that is backed by this layer (link):

- we expose all properties we want (in this case, I mimicked the CAGradientLayer, so there is a colors and a locations property, but instead of startPoint and endPoint I introduced angle (along with angleDidUpdate)),
- implement the layerClass property to ensure our layer acts as the root layer in the UIView,
- implement the layer property, in order to be able to access its values more easily, and
- implement a basic initialization. Although I've assigned default values to each property, the initializer could be left empty.

```
final class GradientView: UIView {
    var colors: [UIColor]? {
        get {
            layer.color
        }
        set {
            layer.color = newValue
        }
    }

    var locations: [CGFloat]? {
        get {
            layer.location
        }
        set {
            layer.location = newValue
        }
    }

    var angle: Angle /* .zero means top to bottom gradient */ {
        get {
            .init(degrees: layer.angleInDegrees)
        }
        set {
```

```
            layer.angleInDegrees = .init(newValue.degrees)
        }
    }

    var angleDidUpdate: ((Angle) -> Void)? {
        get {
            layer.angleDidUpdate
        }
        set {
            layer.angleDidUpdate = newValue
        }
    }

    override class var layerClass: AnyClass {
        GradientLayer.self
    }

    override var layer: GradientLayer {
        super.layer as! GradientLayer
    }

    init(frame: CGRect = .zero, colors: [UIColor]? = nil, locations: [CGFloat]? = nil, angle: Angle = .zero,
         angleDidUpdate: ((Angle) -> Void)? = nil) {
        super.init(frame: frame)
        self.colors = colors
        self.locations = locations
        self.angle = angle
        self.angleDidUpdate = angleDidUpdate
    }

    required init?(coder: NSCoder) {
        super.init(coder: coder)
    }
}

// ...

final class ViewController: UIViewController {
    private let subview: GradientView = {
        let view = GradientView()
        view.translatesAutoresizingMaskIntoConstraints = false
        view.backgroundColor = .blue
        view.heightAnchor.constraint(equalToConstant: 100).isActive = true
        view.widthAnchor.constraint(equalToConstant: 100).isActive = true
        return view
    }()

    override func viewDidLoad() {
        super.viewDidLoad()

        view.addSubview(subview)
        subview.centerYAnchor.constraint(equalTo: view.centerYAnchor).isActive = true
        subview.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true

        subview.colors = [.red, .green]
        subview.angle = .zero

        DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
            self.animate()
        }
    }

    private func animate() {
        UIView.animate(withDuration: 3) {
            self.subview.angle = .degrees(270)
        }
    }
}
```

We've just built a UIView from the ground up with a brand new animatable property! 🎉

. . .

4 Part IV — Miscellaneous. Things to consider

At the start, you might wonder why to use animatable properties when you're already constructing large, high-quality iOS applications without them. While animatable properties can be replaced, they undeniably offer several advantages that are worth understanding.

Here's a brief list highlighting their benefits:

- **Compatibility**: native animatable properties work seamlessly eliminating any additional third-party dependencies, ensuring smooth integration,

- **Openness**: integrating animatable properties within your animation context can effortlessly blend with other animations, enabling simultaneous manipulation of multiple properties without additional code, or even use in any of the UIKit's animation methodologies (i.e. you don't have to write a single line of code to be able to use your property in keyframe animations),
- **Clarity**: since it's a native solution, a new team member might already know this technique, and it helps in reducing the learning curve for them,
- **Stability**: if a new iOS version is coming out, or a new UI framework is released, you can be sure, that this type of animatable properties will still be working, and be supported - as Apple heavily builds upon this for several years now.

Here is the link to the repo, where you can find all the code above.

. . .

I hope you found this article engaging and insightful. If it resonated with you and you found it helpful, please consider giving it a clap and share it to make it more discoverable for others! Should you have any questions, feel free to ask or follow me, and I'll endeavor to address them ⚡