Sample Code

Controlling the timing and movements of your animations

Build sophisticated animations that you control using phase and keyframe animators.

Download

iOS 17.0+ | iPadOS 17.0+ | macOS 14.0+ | Xcode 15.0+

Overview

SwiftUI provides a collection of useful animations that you can use in your app. These animations help enhance the user experience of your app by providing visual transitions of views and user interface elements. While these standard animations provide a great way to enhancement the user interaction of your app, there are times when you need to have more control over the timing and movement of a visual element. `PhaseAnimator` and `KeyframeAnimator` help give you that control.

A phase animator allows you to define an animation as a collection of discrete steps called phases. The animator cycles through these phases to create a visual transition. With keyframe animator, you create keyframes that define animation values at specific times during the visual transition.

Create a simple bounce animation

To better understand how to create animations using a `PhaseAnimator` or `KeyframeAnimator`, start with a simple example that uses standard SwiftUI animations. The following code moves an emoji upwards by setting its offset to `-40.0`. To provide a smooth transition of the movement, the code uses the `withAnimation(_:_:)` function to apply a `bouncy` animation after someone taps the emoji.

```swift
struct SimpleAnimationView: View {
    var emoji: String
    @State private var offset = 0.0

    var body: some View {
        EmojiView(emoji: emoji)
            .offset(y: offset)
            .onTapGesture {
                withAnimation(.bouncy) {
                    offset = -40.0
                }
            }
    }
}
```

This animation has a single, discrete step: move the emoji upward. However, an animation can have multiple steps, such as moving an emoji upwards then back to its original position. For example, the following code sets the offset to `-40.0` to move the emoji upward, and then sets the offset (`0.0`) to return the emoji back to its original position:

```swift
struct SimpleAnimationView: View {
    var emoji: String
    @State private var offset = 0.0

    var body: some View {
        EmojiView(emoji: emoji)
            .offset(y: offset)
            .onTapGesture {
                withAnimation(.bouncy) {
                    offset = -40.0
                } completion: {
                    withAnimation {
                        offset = 0.0
                    }
                }
            }
    }
}
```

This code uses the `withAnimation(_:completionCriteria:_:completion:)` function to animate the two steps of the visual transition. The first step occurs in the body closure of the function, setting the offset to −40.0. The second step occurs in the completion closure, setting the offset to 0.0.

Play ⊙

However, EmojiView actually goes through three steps. The first step happens when the view appears for the first time. The offset of the EmojiView view is 0.0. When someone taps the view, the offset changes to −40.0; this is the second step. When that animation completes, the third step changes the offset back to 0.0. However, there are only two discrete steps, based on the offset values (0.0 and −40.0).

While this implementation certainly works as expected, using a PhaseAnimator is a more convenient way to define discrete steps as phases of an animation.

Bounce with a phase animator

A PhaseAnimator automatically advances through a set of given phases to create an animated transition. Use the phaseAnimator(_:content:animation:) modifier to provide the animator the phases for changing the animation value. For example, the emoji bounce animation shown earlier has two phases: move up and move back. You can represent these phases using the Boolean values, true and false. When the phase is true, the emoji moves up to −40.0. When the phase is false, the emoji moves back to the original position by setting the offset to 0.0.

```
struct TwoPhaseAnimationView: View {
    var emoji: String

    var body: some View {
        EmojiView(emoji: emoji)
            .phaseAnimator([false, true]) { content, phase in
                content.offset(y: phase ? -40.0 : 0.0)
            }
    }
```

The phase animator cycles through the list of phases in the order that you provide to the phaseAnimator(_:content:animation:) modifier. When the view first appears, the phase animator invokes the `content` closure passing in the first phase. Then the animator calls the closure with the value of the second phase. The animator continues to call the `content` closure for each additional phase. After reaching the last phase, the animator calls `content` one more time with the value of the first phase.

This means that in the previous code, the phase animator calls `content` with the phase value of `false` when the view first appears. This sets the emoji's offset to `0.0`. The phase animator then calls `content` with the `true` phase. This phase sets the offset to `−40.0`, causing the emoji to move upwards. After reaching that offset position, the animator calls `content` with the phase of `false`. This causes the emoji to move back to its original position by setting its offset to `0.0`.

This animation starts when the view appears. To start the animation based on an event, use the phaseAnimator(_:trigger:content:animation:) modifier and provide a `trigger` value that animator observes for changes. The animator starts the animation when the value changes. For example, the following code increments the state variable `likeCount` each time a person taps the emoji. The code uses `likeCount` as the value that the phase animator observes for changes. Now whenever someone taps the emoji, it moves up and returns to its original position.

```swift
struct TwoPhaseAnimationView: View {
    var emoji: String
    @State private var likeCount = 1

    var body: some View {
        EmojiView(emoji: emoji)
            .phaseAnimator([false, true], trigger: likeCount) { content, phase in
                content.offset(y: phase ? -40.0 : 0.0)
            }
            .onTapGesture {
                likeCount += 1
            }
    }
}
```

Play ⦿

So far, the phase animator uses the default animation to move the emoji. You can change that behavior by providing the `phaseAnimator` modifier an animation closure. In this closure, specify the type of animation to apply for each phase. For instance, the following code applies a bouncy animation when the phase is `true`; otherwise, it applies the default animation:

```
struct TwoPhaseAnimationView: View {
    var emoji: String
    @State private var likeCount = 1

    var body: some View {
        EmojiView(emoji: emoji)
            .phaseAnimator([false, true], trigger: likeCount) { content, phase in
                content.offset(y: phase ? -40.0 : 0.0)
            } animation: { phase in
                phase ? .bouncy : .default
            }
            .onTapGesture {
                likeCount += 1
            }
    }
}
```

Play ⦿

Add more phases to the animation

While this bounce effect is nice, you can add more pizzazz to it. For instance, you could make the emoji increase in size as it moves upward, and then shrink back to normal size. To do this, you'll add a third phase to the animation: scale.

To define the phases, create a custom type that lists the possible phases; for example:

```swift
private enum AnimationPhase: CaseIterable {
    case initial
    case move
    case scale
}
```

Next, to help simplify logic and reduce complexity, define computed properties that return the values to animate. For instance, to set the vertical offset to move the emoji, create a computed property that returns the offset based on the current phase:

```swift
private enum AnimationPhase: CaseIterable {
    case initial
    case move
    case scale

    var verticalOffset: Double {
        switch self {
        case .initial: 0
        case .move, .scale: -64
        }
    }
}
```

When at the initial phase, the offset is `0`, which is the original screen location for the emoji. But when the phase is `move` or `scale`, the offset is `−64`.

You can use the same approach (creating a computed property) for the scale effect to change the size of the emoji. Initially, the emoji appears at its original size, but increases in size during the move and scale phase, as shown here:

```swift
private enum AnimationPhase: CaseIterable {
    case initial
    case move
    case scale

    var verticalOffset: Double {
        switch self {
        case .initial: 0
        case .move, .scale: −64
        }
    }

    var scaleEffect: Double {
        switch self {
        case .initial: 1
        case .move, .scale: 1.5
        }
    }
}
```

To animate an emoji, apply the `phaseAnimator(_:trigger:content:animation:)` modifier to the `EmojiView`. Provide the animator all cases from the custom `AnimationPhase` type. Then change the content based on the phase by applying the `scaleEffect(_:anchor:)` and `offset(x:y:)` modifiers. The values passed into these modifiers come from the computed properties, which helps keep the view code more readable.

```swift
struct ThreePhaseAnimationView: View {
    var emoji: String
    @State private var likeCount = 1

    var body: some View {
        EmojiView(emoji: emoji)
            .phaseAnimator(AnimationPhase.allCases, trigger: likeCount) { content, p
```

```
                content
                    .scaleEffect(phase.scaleEffect)
                    .offset(y: phase.verticalOffset)
            } animation: { phase in
                switch phase {
                case .initial: .smooth
                case .move: .easeInOut(duration: 0.3)
                case .scale: .spring(duration: 0.3, bounce: 0.7)
                }
            }
            .onTapGesture {
                likeCount += 1
            }
    }
}
```

The code also applies different animation types based on the phase in the `animation` closure to give the full animation that pizzazz you were looking for.

> **Note**
>
> Use the canvas preview in Xcode to help determine the animation types and values to apply for the phase animations. Make changes to the code and see those changes reflected in the canvas preview.

A [`PhaseAnimator`] gives you control of an animation based on discrete phases, which helps you add extra polish to an animation. But if you find that you need even more control over the timing and movement of an animation, use a [`KeyframeAnimator`].

Gain more control with a keyframe animator

You can define complex, coordinated animations with complete control over timing and movement using a `KeyframeAnimator`. This animator allows you to create keyframes that define values at specific times during an animation. The animator use these values to generate interpolated values in between each frame of the animation.

Unlike a phase animator, in which you model separate, discrete states, a keyframe animator generates interpolated values of the type that you specify. While an animation is in progress, the animator provides you with a value of this type on every frame so you can update the animating view by applying modifiers to it.

You define the type as a structure that contains the properties that you want to independently animate. For example, the following code defines four properties that determine the scale, stretch, position, and angle of an emoji:

```
private struct AnimationValues {
    var scale = 1.0
    var verticalStretch = 1.0
    var verticalOffset = 0.0
    var angle = Angle.zero
}
```

Note

`KeyframeAnimator` can animate any value that conforms to the `Animatable` protocol.

To create a animation using a keyframe animator, apply either the `keyframeAnimator(initialValue:repeating:content:keyframes:)` or `keyframeAnimator(initialValue:trigger:content:keyframes:)` modifier to the view that you want to animate. For instance, the following code applies the second modifier to `EmojiView`. The initial value for the animation is a new instance of `AnimationValues`, and the state variable `likeCount` is the value that the animator observes for changes as it did in the previous phase animation example.

```
struct KeyframeAnimationView: View {
    var emoji: String
    @State private var likeCount = 1

    var body: some View {
        EmojiView(emoji: emoji)
            .keyframeAnimator(
                initialValue: AnimationValues(),
                trigger: likeCount
            ) { content, value in
                // ...
            } keyframes: { _ in
                // ...
            }
            .onTapGesture {
                likeCount += 1
            }
    }
}
```

To apply modifiers to a view during the animation, provide a `content` closure to the keyframe animator. This closure includes two parameters:

`content`
: The view that's animating.

`value`
: The current interpolated values.

Use these parameters to apply modifiers to the view that SwiftUI is animating. For example, the following code uses these parameters to rotate, scale, stretch, and move an emoji:

```
struct KeyframeAnimationView: View {
    var emoji: String
    @State private var likeCount = 1

    var body: some View {
        EmojiView(emoji: emoji)
            .keyframeAnimator(
                initialValue: AnimationValues(),
                trigger: likeCount
            ) { content, value in
                content
                    .rotationEffect(value.angle)
                    .scaleEffect(value.scale)
                    .scaleEffect(y: value.verticalStretch)
                    .offset(y: value.verticalOffset)
            } keyframes: { _ in
                // ...
            }
            .onTapGesture {
                likeCount += 1
            }
        }
    }
```

Important

SwiftUI calls the keyframe animator’s `content` closure on every frame of the animation, so avoid performing any expensive operations directly within it.

Next, define the keyframes. Keyframes let you build sophisticated animations with different keyframe for different properties. To make this possible, you organize the keyframes into tracks. Each track controls a different property of the type that you are animating. You associate a property to a track by providing the key path to the property when creating the track. For example, the following code adds a `KeyframeTrack` for the `scale` property:

```
struct KeyframeAnimationView: View {
    var emoji: String
    @State private var likeCount = 1

    var body: some View {
        EmojiView(emoji: emoji)
            .keyframeAnimator(
                initialValue: AnimationValues(),
                trigger: likeCount
            ) { content, value in
                content
                    .rotationEffect(value.angle)
                    .scaleEffect(value.scale)
                    .scaleEffect(y: value.verticalStretch)
                    .offset(y: value.verticalOffset)
            } keyframes: { _ in
                KeyframeTrack(\.scale) {
                    // ...
                }
            }
            .onTapGesture {
                likeCount += 1
            }
    }
}
```

When creating a track, you use the declarative syntax in SwiftUI to add keyframes to the track. There are different kinds of keyframes, such as `CubicKeyframe`, `LinearKeyframe`, and `SpringKeyframe`. You can mix and match the different kinds of keyframes within a track. For example, the following code adds a track for the `scale` property that performs a combination of linear and spring animations:

```swift
struct KeyframeAnimationView: View {
    var emoji: String
    @State private var likeCount = 1

    var body: some View {
        EmojiView(emoji: emoji)
            .keyframeAnimator(
                initialValue: AnimationValues(),
                trigger: likeCount
            ) { content, value in
                content
                    .rotationEffect(value.angle)
                    .scaleEffect(value.scale)
                    .scaleEffect(y: value.verticalStretch)
                    .offset(y: value.verticalOffset)
            } keyframes: { _ in
                KeyframeTrack(\.scale) {
                    LinearKeyframe(1.0, duration: 0.36)
                    SpringKeyframe(1.5, duration: 0.8,
                        spring: .bouncy)
                    SpringKeyframe(1.0, spring: .bouncy)
                }
            }
            .onTapGesture {
                likeCount += 1
            }
    }
}
```

Each keyframe type receives a value. The animator uses this value to generate interpolated values between frames and sets the property specified in the track's key path before calling the animator's content closure. For instance, in the previous code listing, the scale value is `1.0` during the linear keyframes, which keeps the emoji at its original size. Then the scale changes to `1.5` during the first spring keyframe. This causes the emoji to grow in size. The final spring keyframe sets the scale to `1.0`, which returns the emoji back to its original size.

Note

SwiftUI preserves the velocity (that is, the speed of the animation) across multiple keyframes for continuous motion within a track.

When implementing a keyframe animation, include a track for each property that you want to animate. For instance, `AnimationValues` has four properties:

- `scale`
- `verticalStretch`
- `verticalOffset`
- `angle`

To animate all four, the animator needs four keyframe tracks as shown in the following code:

```
struct KeyframeAnimationView: View {
    var emoji: String
    @State private var likeCount = 1

    var body: some View {
        EmojiView(emoji: emoji)
            .keyframeAnimator(
                initialValue: AnimationValues(),
                trigger: likeCount
            ) { content, value in
                content
                    .rotationEffect(value.angle)
                    .scaleEffect(value.scale)
                    .scaleEffect(y: value.verticalStretch)
                    .offset(y: value.verticalOffset)
            } keyframes: { _ in
                KeyframeTrack(\.scale) {
                    LinearKeyframe(1.0, duration: 0.36)
                    SpringKeyframe(1.5, duration: 0.8, spring: .bouncy)
                    SpringKeyframe(1.0, spring: .bouncy)
                }

                KeyframeTrack(\.verticalOffset) {
                    LinearKeyframe(0.0, duration: 0.1)
                    SpringKeyframe(20.0, duration: 0.15, spring: .bouncy)
                    SpringKeyframe(-60.0, duration: 1.0, spring: .bouncy)
                    SpringKeyframe(0.0, spring: .bouncy)
                }

                KeyframeTrack(\.verticalStretch) {
                    CubicKeyframe(1.0, duration: 0.1)
                    CubicKeyframe(0.6, duration: 0.15)
                    CubicKeyframe(1.5, duration: 0.1)
                    CubicKeyframe(1.05, duration: 0.15)
                    CubicKeyframe(1.0, duration: 0.88)
                    CubicKeyframe(0.8, duration: 0.1)
                    CubicKeyframe(1.04, duration: 0.4)
                    CubicKeyframe(1.0, duration: 0.22)
                }

                KeyframeTrack(\.angle) {
                    CubicKeyframe(.zero, duration: 0.58)
                    CubicKeyframe(.degrees(16), duration: 0.125)
                    CubicKeyframe(.degrees(-16), duration: 0.125)
                    CubicKeyframe(.degrees(16), duration: 0.125)
                    CubicKeyframe(.zero, duration: 0.125)
                }
            }
            .onTapGesture {
                likeCount += 1
            }
    }
}
```

The combination of these keyframe tracks creates an animation that squishes and stretches the emoji, before bouncing it upwards. As the emoji moves towards its peak, it grows larger. When the emoji reaches its peak, it gives a little wiggle. Then the emoji returns to its original location with a slight bounce as it settles back into its original position.

Play ▶

> **Note**
>
> As with phase animations, use the canvas preview in Xcode to help determine the animation types and values to apply for the keyframe animations. Make changes to the code and see those changes reflected in the canvas preview.

See Also

Creating phase-based animation

```
func phaseAnimator<Phase>(some Sequence, content: (PlaceholderContent
View<Self>, Phase) -> some View, animation: (Phase) -> Animation?) ->
some View
```

Animates effects that you apply to a view over a sequence of phases that change continuously.

```
func phaseAnimator<Phase>(some Sequence, trigger: some Equatable,
content: (PlaceholderContentView<Self>, Phase) -> some View, animation:
(Phase) -> Animation?) -> some View
```

Animates effects that you apply to a view over a sequence of phases that change based on a trigger.

```
struct PhaseAnimator
```

A container that animates its content by automatically cycling through a collection of phases that you provide, each defining a discrete step within an animation.