Chapter 9 Animation in SwiftUI

1. Introduction

Animation is an important component in iOS apps. When transitioning from one view to another, we can make the transition fluid by adding some transition. Game apps depends heavily on animation. We can simulate, using animation, an object falling under gravity. And the list goes on and on.

There are two types of animation: implicit and explicit. In implicit animation, we attach an `.animation(_:)` modifier to a view and the animation involves changing one or more states. In explicit animation, we don't attach modifiers to a view in question but instead ask SwiftUI to animate the precise change we want to make. To do this, we wrap our changes in a call to `withAnimation()`.

When SwiftUI animates a view, it goes through a series of views created, each time modifying the animating parameter by a small step. This way it goes progressively from the initial value to the final value.

Behind all SwiftUI animations, there is a protocol `Animatable`. It involves having a computed type that conforms to `VectorArithmetic`. This enables SwiftUI to interpolate values at will.

For example, we consider a position of a View change from an initial position to a final position. SwiftUI will regenerate the view many times, altering the position by little increments. Since position is expressed as `CGFloat`, and because `CGFloat` conforms to `VectorArithmetic`, SwiftUI can interpolate the position values required. So, SwiftUI can animate position, without us to worry about it.

However, this is not always the case. Some of the big exceptions coming to our mind are paths, transform matrices, and arbitrary view changes. In these cases, SwiftUI does not know what to do and we need `Animatable`. (To be edited at the end)

Code for this chapter is available on GitHub:

http://www.github.com/mhuq1138/swiftui-chapter-9-animation-in-swiftui

2. Animation struct

At the heart of all SwiftUI animation is the `Animation` struct with the declaration:

```
@frozen struct Animation
```

It has the following type properties that determine how the animation progresses from start to end.

```
static let `default`: Animation//default is same as linear
static var easeIn: Animation//animation starts slowly
static var easeInOut: Animation//animation starts and ends slowly
static var easeOut: Animation//animation ends slowly
static var linear: Animation//animation progresses linearly
```

All of these yield animation with a default duration of about 0.5 seconds. For the `linear` property the animation is linear, which means that the same change takes place during the same time interval. For the `easeIn` property the animation starts increasing gradually until it becomes linear. For the `easeOut` property the animation starting linearly slows down before ending. For the `easeInOut` property the animation starts and ends slowly. The `default` case is the same as `linear`.

If we want custom duration, we should the type methods instead:

```
static func linear(duration: Double) -> Animation
static func easeIn(duration: Double) -> Animation
static func easeInOut(duration: Double) -> Animation
static func easeOut(duration: Double) -> Animation
```

If we wish to put in a spring in our animation, we have the following three type methods.

The following two methods appear to yield the same spring animation, except that they have different default parameters.

```
static func spring(response: Double = 0.55, dampingFraction: Double = 0.825, blendDuration:
                                                                   Double = 0) -> Animation
```

```
static func interactiveSpring(response: Double = 0.15, dampingFraction: Double = 0.86,
                                                 blendDuration: Double = 0.25) -> Animation
```

The third method makes use of the physics formula for damped simple harmonic oscillation to determine the spring animation:

```
static func interpolatingSpring(mass: Double = 1.0, stiffness: Double, damping: Double,
                                                    initialVelocity: Double = 0.0) -> Animation
```

There is a type method that is used to give us custom time curve for the animation:

```
static func timingCurve(_ c0x: Double, _ c0y: Double, _ c1x: Double, _ c1y: Double, duration:
Double = 0.35) -> Animation
```

We also have four instance methods: The method

```
func delay(_ delay: Double) -> Animation
```

starts the with an initial delay (in seconds).

If we wish to repeat the animation, with or without auto reverse, we can use the instance method

```
func repeatCount(_ repeatCount: Int, autoreverses: Bool = true) -> Animation
```

There is also an instance method for repeating animation forever.

```
func  repeatForever(autoreverses: Bool = true) -> Animation
```

Finally, we have a method to speed up an animation:

```
func speed(_ speed: Double) -> Animation
```

If the animation has a duration of 2.0 seconds and the speed parameter is 0.25, the effective animation duration will be 0.5 seconds.

3. Implicit Animation

Implicit animation of a view is performed by applying the `.animation(_:)` view modifier to the view. This modifier has the definition:

```
func animation(_ animation: Animation?) -> some ViewModifier
```

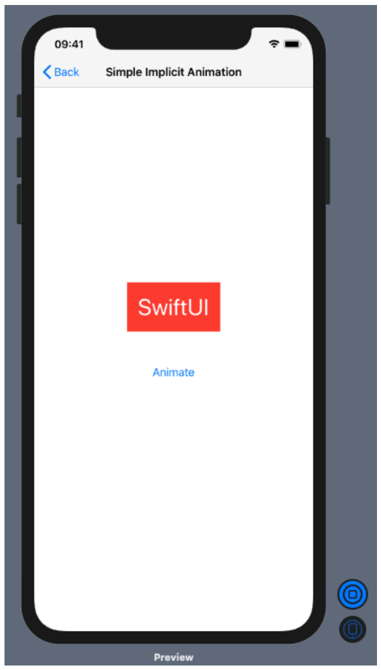
The argument `animation` has the type `Animation`. The way implicit animation works is as follows: We apply the `.animation(_:)` view modifier to a view. If the appearance or behavior of the view depends on a `@State` property, any change made to the `@State` property will be animated.

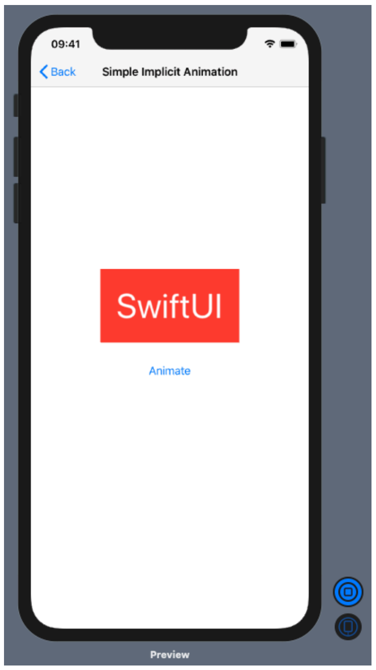
We are going to consider several examples to illustrate how can use implicit animation. All these examples will be put together in a project “Implicit Animation Demo”. Each example will be a row in a `List` embedded in a `VStack`.

We start with the following code to illustrate default animation:

```swift
struct SimpleAnimation: View {
    @State private var scale = false

    var body: some View {
        VStack{
             Text("SwiftUI")
                .font(.largeTitle)
                .foregroundColor(.white)
                .padding()
                .background(Color.red)
                .scaleEffect(scale ? 1.5:1.0)
                .animation(.default)

            Button("Animate"){
                self.scale.toggle()
            }.padding(50)

        }.navigationBarTitle("Simple Implicit Animation", displayMode: .automatic)
    }
}
```

Note that only the `.scaleEffect` view modifier depends on the `@State scale`. Changing the state will change the argument of`.scaleEffect` and thus change the size of the `Text` view. We have applied the`.animation(_:)` view modifier to the `Text` view and a tap on the button will change `scale` triggering an animation that changes the size of the `Text` view. We show below screen shots of live preview before and after a button tap:



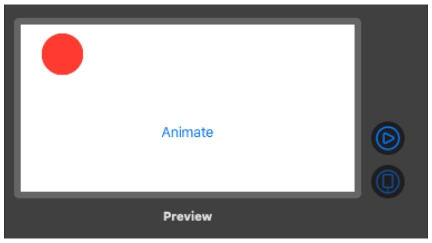


The left screen shot is when `scale` is `false` making the argument of `.scaleEffect` view modifier 1. Tapping the button makes `scale` `true` making the argument of `.scaleEffect` view modifier 1.5. This increases the size of the `Text` view by a factor of 1.5, as shown by the right screen shot. Also note that the first argument of `.scaleEffect` view modifier is of type `CGFloat`, which conforms to `VectorArithmetic`. So, SwiftUI knows how to interpolate between different values of the first argument of `.scaleEffect`.

> Note that the declaration of scaleEffect is:
>
> ```
> func scaleEffect(_ s: CGFloat, anchor: UnitPoint = .center) -> some View
> ```
>
> By default, the view is scaled around the center anchor point.

Note that we are not limited to animating a single characteristics of a view at a time, we can animate multiple characteristics simultaneously. For example, we can animate the fill color of the circle simultaneously with its position by using the following code:

```
struct MoveAndChangeColor: View {
    @State private var flag = true

    var body: some View {
        VStack{
             Circle()
                .fill(flag ? Color.red: Color.blue)
                .frame(width: 50, height: 50)
                .offset(flag ? CGSize(width: -150,
                                      height: 0):
                        CGSize(width: 150, height: 0))
                .animation(.default)

            Button("Animate"){
                self.flag.toggle()
            }.padding(50)

        }.navigationBarTitle("Move and Changer Color",
                                displayMode: .automatic)
    }
}
```



Now as the Circle animates from the leading position to the trailing position, its color changes from red to blue with animation. During reverse animation from the trailing to leading position, it changes back to red.

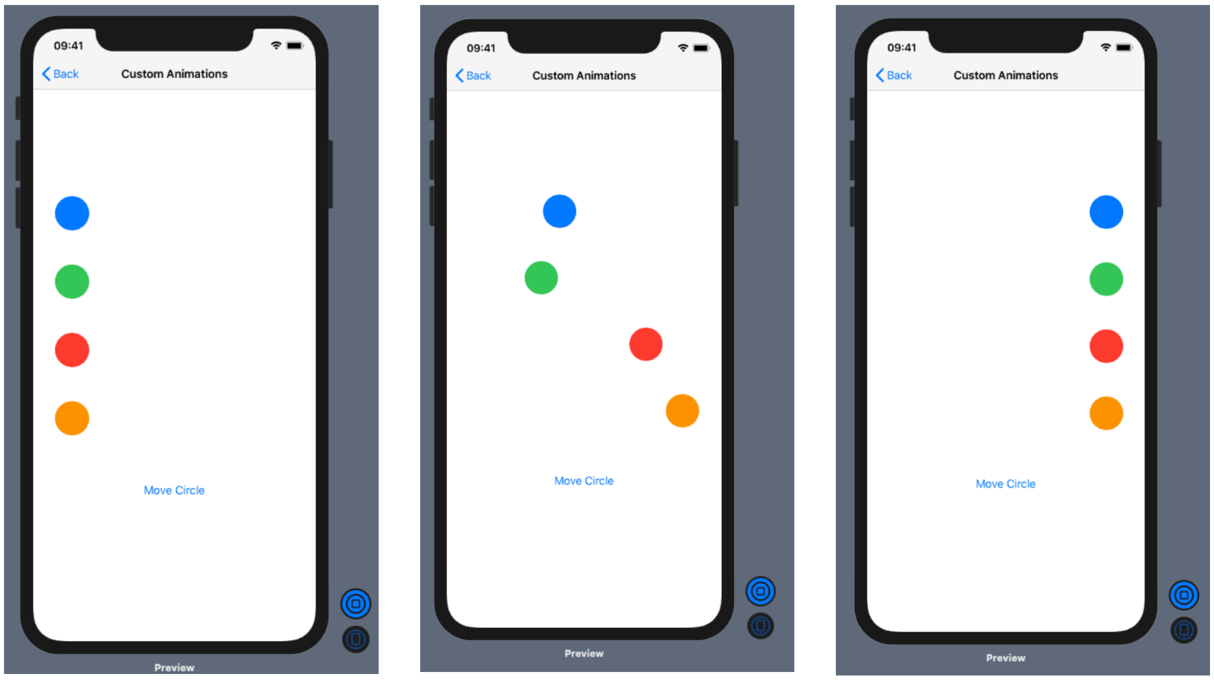
Next, we want demonstrate the different ways animation progresses from start to finish. We place four circles with different colors and animate them using the four different type methods:

```
struct CustomAnimationView: View {
    @State private var flag = true

    var body: some View {
        let leadingOffset = CGSize(width: -150, height: 0)
        let trailingOffset = CGSize(width: 150, height: 0)
```

```
        return VStack(spacing:50){
                    MyCircle(color: .blue)
                        .offset(flag ? leadingOffset:trailingOffset)
                        .animation(.linear(duration:5))

                    MyCircle(color: .green)
                        .offset(flag ? leadingOffset:trailingOffset)
                        .animation(.easeIn(duration:4))

                    MyCircle(color: .red)
                        .offset(flag ? leadingOffset:trailingOffset)
                        .animation(.easeOut(duration:3))

                    MyCircle(color: .orange)
                        .offset(flag ? leadingOffset:trailingOffset)
                        .animation(.easeInOut(duration:2))

                    Button("Move Circle"){
                                              self.flag.toggle()
                    }.padding(20)
               }.navigationBarTitle("Custom Animations", displayMode: .automatic)
      }
}
```

We have given the four circles different animation duration.

In live preview:



The blue circle uses `linear` animation with duration of 5 seconds, the green circle uses `easeIn` animation with duration 4 seconds, the red circle uses `easeOut` animation with duration of 3 seconds, and the orange circle uses `easeInOut` animation with duration of 2 seconds. That explains why during the animation the four circles are not lined up vertically as the middle screen shot shows.

Finally, we demonstrate delay in starting animation and repeating animation with the code:

```
struct DelayAndRepeatingAnimation: View {
    @State private var flag = true

    var body: some View {
        let leadingOffset = CGSize(width: -150, height: 0)
        let trailingOffset = CGSize(width: 150, height: 0)

        return VStack(spacing:50){
                   MyCircle(color: .blue)
                       .offset(flag ? leadingOffset:trailingOffset)
                       .animation(Animation.linear(duration:2).delay(2))

                   MyCircle(color: .green)
                       .offset(flag ? leadingOffset:trailingOffset)
                       .animation(Animation.easeIn(duration:4).repeatCount(4, autoreverses:
                                                                            false))

                   MyCircle(color: .red)
                       .offset(flag ? leadingOffset:trailingOffset)
                       .animation(Animation.easeOut(duration:2).repeatCount(3, autoreverses:
                                                                             true))

                   MyCircle(color: .orange)
                       .offset(flag ? leadingOffset:trailingOffset)
                       .animation(Animation.easeInOut(duration:1)
                       .repeatCount(10).speed(4.0))

                   Button("Move Circle"){
                             self.flag.toggle()
                   }
        }.navigationBarTitle("Delay and Repeating Animation", displayMode: .automatic)
    }
}
```

The blue circle uses the delay method. So, in live preview we will find that it starts animation 2 seconds after the other circles. For the green circle `autoreverses` is set `false`, at the end of each animation the green circle jumps to the leading edge and then start the next animation. While for the red circle `autoreverses` is set `true`, at the end of each animation the red circle animates back to the leading edge and then start the next animation. Finally, for the orange circle we use the `speed(_:)`. So, the effective duration of animation is 4.0 seconds.

It is interesting to note that the `.animation(_:)` can be attached to a SwiftUI binding, which causes the value to animate between its current and new value. This even works if the data in question isn't something that sounds like it can be animated, such as a `Boolean`. For example, we can replace the code for `MoveAndChangeColor` by

```
struct MoveAndChangeColor: View {
    @State private var flag = false

    var body: some View {
        VStack{
             Circle()
                .fill(flag ? Color.red: Color.blue)
                .frame(width: 50, height: 50)
                .offset(flag ? CGSize(width: -150, height: 0): CGSize(width: 150, height: 0))
```

```
            Toggle(isOn: $flag.animation(.linear(duration: 2))){
                Text("Animate")
            }

        }.navigationBarTitle("Move and Changer Color", displayMode: .automatic)
    }
}
```

The code would work fine, now toggling the Toggle will trigger the animation.

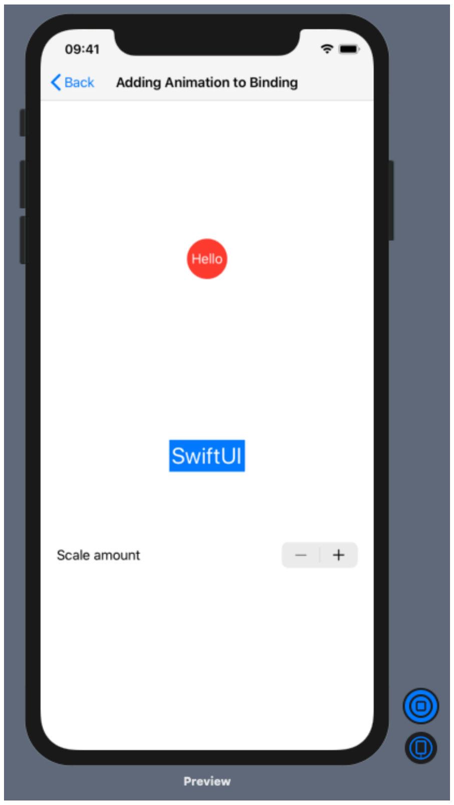
Here we consider an example that works with the `Stepper` control:

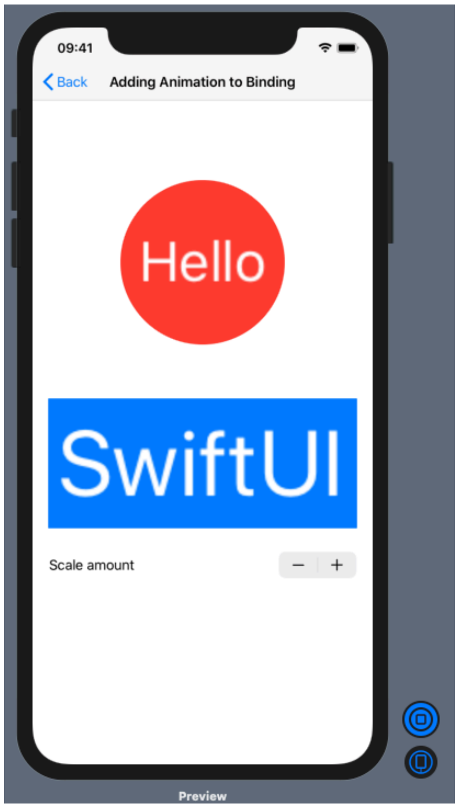
```
struct AddingAnimationToBinding: View {
     @State private var scaleFactor:CGFloat = 1

     var body: some View {
           VStack{
               VStack( spacing: 200){
                   Circle()
                       .fill(Color.red)
                       .frame(width: 50, height: 50)
                       .overlay(Text("Hello").foregroundColor(.white))
                       .scaleEffect(scaleFactor)

                   Text("SwiftUI")
                       .font(.title)
                       .foregroundColor(.white)
                       .padding(3)
                       .background(Color.blue)
                       .scaleEffect(scaleFactor)
               }.padding(.bottom, 50)

               Stepper("Scale amount", value: $scaleFactor.animation(.linear(duration: 2)),
                                                                                      in: 1...10)
               .padding(20)
           }.navigationBarTitle("Adding Animation to Binding", displayMode: .automatic)
      }
}
```

The stepper is bound to `$scaleFactor.animation(.linear(duration: 2))` , which means SwiftUI will automatically animate its changes.

With binding animation we are not setting the animation on a view and implicitly animating the state change. While with binding animation we are explicitly changing the state.

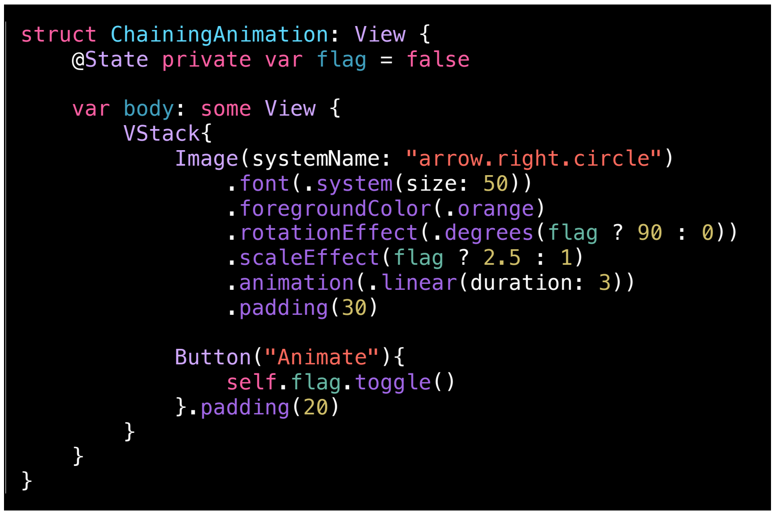
In live preview:



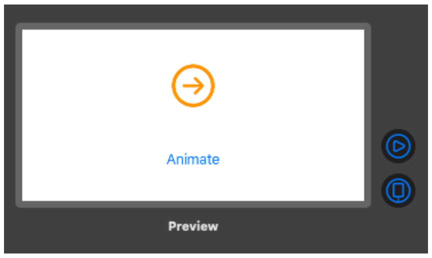


The right screen shot is after a few tap on the + button.

Now, we are going to see that multiple animations can be chained together. Let us start with the code:



```
struct ChainingAnimation: View {
    @State private var flag = false

    var body: some View {
        VStack{
            Image(systemName: "arrow.right.circle")
                .font(.system(size: 50))
                .foregroundColor(.orange)
                .rotationEffect(.degrees(flag ? 90 : 0))
                .scaleEffect(flag ? 2.5 : 1)
                .animation(.linear(duration: 3))
                .padding(30)

            Button("Animate"){
                self.flag.toggle()
            }.padding(20)
        }
    }
}
```



This would animate both rotation scale effect with the same animation with duration of 3 seconds. Suppose we want the rotation and scale effect to use different animations. We can do that using chain animations:

```
struct ChainingAnimation: View {
    @State private var flag = false

    var body: some View {
        VStack{
            Image(systemName: "arrow.right.circle")
                .font(.system(size: 50))
                .foregroundColor(.orange)
                .rotationEffect(.degrees(flag ? 90 : 0))
                .animation(.easeIn(duration: 5))
                .scaleEffect(flag ? 2.5 : 1)
                .animation(.linear(duration: 3))
                .padding(30)
            Button("Animate"){
                self.flag.toggle()
            }.padding(20)
        }
    }
}
```

Now, the rotation animation will have a longer duration. We cancel the rotation animation with:

```
.rotationEffect(.degrees(flag ? 90 : 0))
    .animation(nil)
```

What would happen if we apply two animations to rotation animation:

```
.rotationEffect(.degrees(flag ? 90 : 0))
    .animation(.easeIn(duration: 5))
    .animation(nil)
```

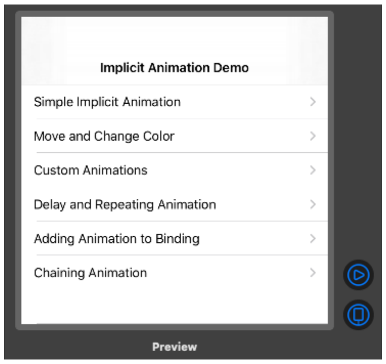
The second animation will be ignored, and the rotation animation will happen. But if we change the order:

```
.rotationEffect(.degrees(flag ? 90 : 0))
    .animation(nil)
    .animation(.easeIn(duration: 5))
```

Now there would be no rotation animation.

We now display the code in the `ContentView` of our project

```
struct ContentView: View {
    var body: some View {
        NavigationView{
            List{
                NavigationLink("Simple Implicit Animation",
                            destination: SimpleAnimation())

                NavigationLink("Move and Change Color",
                         destination: MoveAndChangeColor())

                NavigationLink("Custom Animations",
                        destination: CustomAnimationView())
```


Implicit Animation Demo

Simple Implicit Animation
Move and Change Color
Custom Animations
Delay and Repeating Animation
Adding Animation to Binding
Chaining Animation

```
                NavigationLink("Delay and Repeating
                                    Animation", destination:
                               DelayAndRepeatingAnimation())

                NavigationLink("Adding Animation to
                                  Binding", destination:
                                 AddingAnimationToBinding())

                NavigationLink("Chaining Animation",
                              destination: ChainingAnimation())

            }.navigationBarTitle("Implicit Animation Demo",
                                         displayMode: .inline)
        }
    }
}
```

4. Explicit Animation

We have seen that implicit animation is applied to individual views. It is possible that several views might depend on the same state. In such cases, it would be more appropriate not to apply animation to each view involved, rather apply to the place where we change the state's value to trigger the animation. This is performed by wrapping the change of state in `withAnimation(_:, body:)` method, which has the declaration:

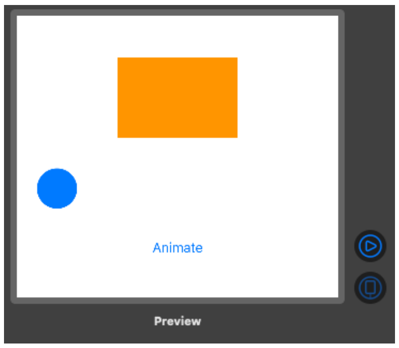
```
func withAnimation<Result>(_ animation: Animation? = .default, _ body: () throws -> Result)
                                                                  rethrows -> Result
```

Explicit animations are often very useful because they cause every affected view to animation, not just those that have implicit animation attached.

Here we are going to consider several examples that we place in the project "Explicit Animation Demo". Different examples can be reached through navigation links.

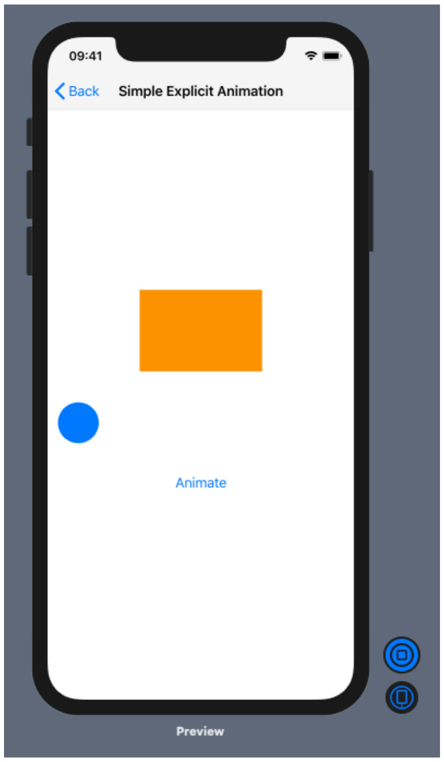
We start with a simple example, where we apply scale effect to a rectangle and offset to a circle:

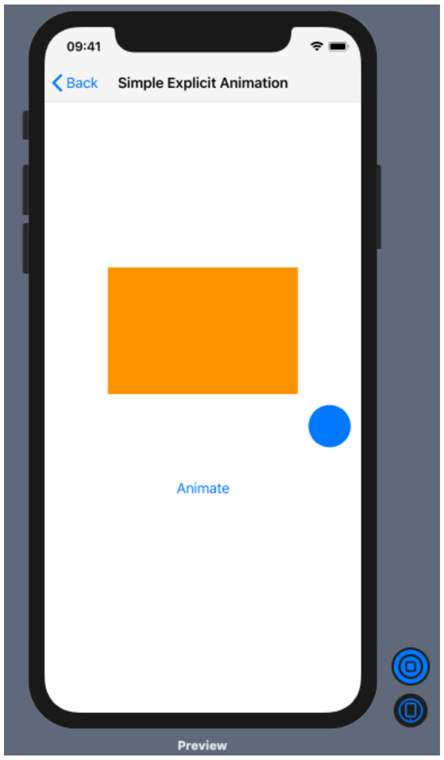
```
@State private var scaleFlag = false
    @State private var moveFlag = false

    var body: some View {
        VStack{
            Rectangle()
                .fill(Color.orange)
                .frame(width: 150, height: 100)
                .scaleEffect(scaleFlag ? 1.5:1.0)
                .padding(.bottom, 30)

            Circle()
                .fill(Color.blue)
                .frame(width: 50, height: 50)
                .offset(moveFlag ? CGSize(width: 150,
                          height: 0):CGSize(width: -150,
                                               height: 0))
```



```
                .padding(.bottom, 30)

            Button("Animate"){
                withAnimation(Animation.linear(duration:
                                                     3)){
                    self.scaleFlag.toggle()
                    self.moveFlag.toggle()
                }
        }
    }
}
```

In live review:

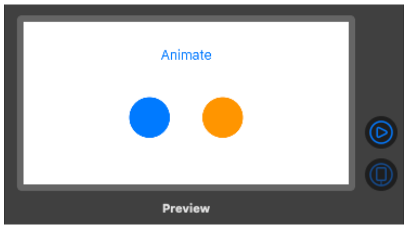




In live preview, when the tap the button the rectangle reduces to 1.5 times its size and the circle moves to the right. On another tap the original state is restored.

In the next example, we have two circles placed in a HStack. Both of them are animated with a `withAnimation(_:, body:)`. But the orange circle also has a `.animation(_:)` applied to it with a shorter duration, which takes precedence over the explicit animation. Therefore, this circles moves down faster.

```
struct TwoCircleMotion_: View {
    @State private var moveFlag = false
    var body: some View {
        VStack{
            Button("Animate"){
                withAnimation(.linear(duration: 3)){
                    self.moveFlag.toggle()
                }
            }.padding(30)

            HStack(alignment: .top, spacing:40){
                MyCircle(color: .blue)
                    .offset(moveFlag ? CGSize(width: 0.0,
                       height: 200.0): CGSize(width: 0.0,
                                             height: 0.0))

                MyCircle(color: .orange)
                    .offset(moveFlag ? CGSize(width: 0.0,
                       height: 400.0): CGSize(width: 0.0,
                                            height: 0.0))
                    .animation(.easeIn(duration: 1.5))
            }
            Spacer()
        }
    }
}
```



5. Spring Animation

In section 2, we have listed three view modifiers we can use to create spring like animation. In such animations the view shoots past the final value and then oscillates around the final value with damped oscillations and ultimately settle down at the final value. The frequency of oscillation, magnitude, and damping depends on the arguments supplied to the function call.

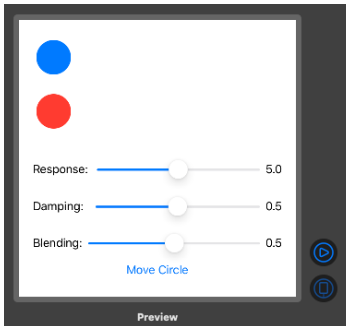
We create the project "Spring Animation Demo" to demonstrate the use of these view modifiers. Since we are going to use `Slider` control with appropriate labels to choose the method parameters, it would be convenient to create the struct:

```
struct SliderDisplay: View {
    @Binding var sliderValue:Double
    let label:String
    let lower:Double
    let upper:Double

    var body: some View {
        let value = String(format: "%0.1f", arguments: [sliderValue])

       return HStack{
            Text(label)
            Slider(value: $sliderValue, in: lower...upper)
            Text(value)
        }.padding(EdgeInsets(top: 5, leading: 20, bottom: 5, trailing: 20))
    }
}
```

In our first example, we are going consider the methods that has a response parameter

On the first tab, we are going to demonstrate the first two methods using two circles with different fill color. The code we use is

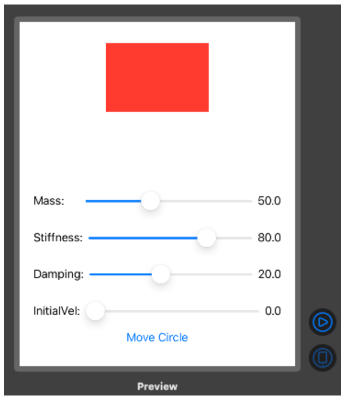
```swift
struct SpringAnimationWithResponse: View {
    @State var response: Double = 5.0
    @State var damping: Double = 0.5
    @State var blending: Double = 0.5
    @State var sign = 1

    @State private var offset = CGSize(width: -150, height: 0)

    var body: some View {
        VStack{
           MyCircle(color:.blue)
               .offset(offset)
               .padding(.bottom, 20)
               .animation(.spring(response: response,
                                dampingFraction: damping,
                                    blendDuration: blending))

           MyCircle(color:.red)
               .padding(.bottom, 30)
               .offset(offset)
               .animation(.interactiveSpring(response:
                          response, dampingFraction: damping,
                                     blendDuration: blending))

           SliderDisplay(sliderValue: $response, label:
                      "Response: ", lower: 0.0, upper: 10.0)
           SliderDisplay(sliderValue: $damping, label:
                      "Damping:  ", lower: 0.0, upper: 1.0)
           SliderDisplay(sliderValue: $blending, label:
                         "Blending:", lower: 0.0, upper: 1.0)
            Button("Move Circle"){
                self.sign = -self.sign
                self.offset = CGSize(width: -150*self.sign,
                                                 height: 0)
            }
        }
    }
}
```


Response: 5.0
Damping: 0.5
Blending: 0.5
Move Circle

On live preview, we would discover that both methods produce the same animation. Smaller value of response produces more rapid oscillations.

As our second example, we would consider the oscillation of a rectangle about its center anchor point

```swift
struct MassSpringAnimation: View {
    @State var mass: Double = 50.0
    @State var stiffness: Double = 80.0
    @State var damping: Double = 20.0
    @State var initialVel: Double = 0.0
    @State var rotationFlag = false

    @State private var offset = CGSize(width: -150, height: 0)

       var body: some View {
           VStack{
              Rectangle()
                .fill(Color.red)
                .frame(width: 150, height: 100)
                .rotationEffect(rotationFlag ? Angle(degrees:
                                       90):Angle(degrees: 0))
                .padding(.bottom, 100)
```


Mass: 50.0
Stiffness: 80.0
Damping: 20.0
InitialVel: 0.0
Move Circle

```
              SliderDisplay(sliderValue: $mass, label:
                    "Mass:      ", lower: 20.0, upper: 100.0)
              SliderDisplay(sliderValue: $stiffness, label:
                    "Stiffness:", lower: 20.0, upper: 100.0)
              SliderDisplay(sliderValue: $damping, label:
                     "Damping:", lower: 5.0, upper: 40.0)
              SliderDisplay(sliderValue: $initialVel, label:
                       "InitialVel:", lower: 0.0, upper: 30.0)

               Button("Move Circle"){
                     withAnimation(.interpolatingSpring(mass:
                        self.mass, stiffness: self.stiffness,
                                 damping: self.damping,
                       initialVelocity: self.initialVel)){
                    self.rotationFlag.toggle()
                }
            }
        }.navigationBarTitle("Spring Animation with Mass-
                               Spring", displayMode: .inline)
    }
}
```

6. AnimatableData

So far, we have considered animations for which SwiftUI knows how to interpolate between the initial and the final value involved in the animation. For example, when we animate opacity between 0.2 and 0.8, SwiftUI knows how to regenerate the view many times, altering the opacity by small increments. Now opacity is expressed as a `Double` and because `Double` conforms to `VectorArithmetic`.

But there are some big exceptions, such as paths, transform matrices, and arbitrary vie changes where SwiftUI doesn't know what to do. This is where `Animatable` protocol comes to our rescue. This protocol has a single requirement:

```
var animatableData: Self.AnimatableData { get set }
```

Here we are going to consider animating `Shape`. Fortunately, `Shape` already conforms to `Animatable`. This means that there is a computed property (`animatableData`), but this is set to `EmptyAnimatableData`. So, we have to create our own.

We are going to consider several examples, which we place in the project "AnimatableDate Animation Demo". Let us start with a simple example that demonstrates spring like animation of a line with the length of the line oscillating.

First of all we create the line we wish to animate:

```
struct MyLine: Shape {
    var width: CGFloat

    func path(in rect: CGRect) -> Path {
        var p = Path()
        p.move(to: CGPoint(x: rect.size.width/2-width, y: 0))
        p.addLine(to: CGPoint(x: rect.size.width/2+width, y: 0))

        return p
    }
}
```

This creates a line of specified horizontally centered on the interface. Let try to animate using the code:

```
struct SpringingLine: View {
    @State private var width:CGFloat = 30.0

    var body: some View {
        GeometryReader{ g in
            VStack{
                MyLine(width: self.width)
                    .stroke(Color.blue, lineWidth: 10)
                    .frame(width: 300, height: 50)
                    .animation(Animation.interpolatingSpring(stiffness: 23, damping: 0.8))
                    .padding(.bottom, 50)
                Button("Tap Me"){
                    self.width = g.size.width/4
                }
            }
        }
    }
}
```

In live preview, we don't see any animation, the line width jumps to its final value. This because even though Shape conforms to `Animatable` protocol it returns `EmptyAnimatableData`. So, we need to add `AnimatableData` we want explicitly:

```
struct MyLine: Shape {
    var width: CGFloat

    var animatableData: CGFloat{
        get{ width }
        set{ width = newValue }
    }

    func path(in rect: CGRect) -> Path {
        var p = Path()
        p.move(to: CGPoint(x: rect.size.width/2-width, y: 0))
        p.addLine(to: CGPoint(x: rect.size.width/2+width, y: 0))

        return p
    }
}
```

Now we get the animation we want.

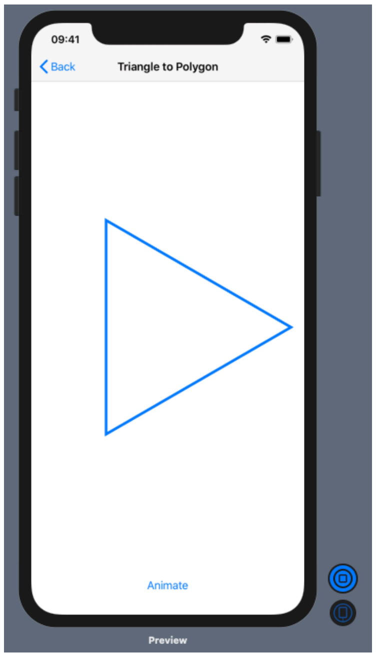
As our second example, we would like to animate a triangle into a eight-sided polygon. The question is how do you interpolate between two integral number of sides in small increments. Obviously, you need to go through polygon with 4.5 sides and so on. We can teach SwiftUI how to do this with `AnimatableData`. First let us see how we can draw a polygon:

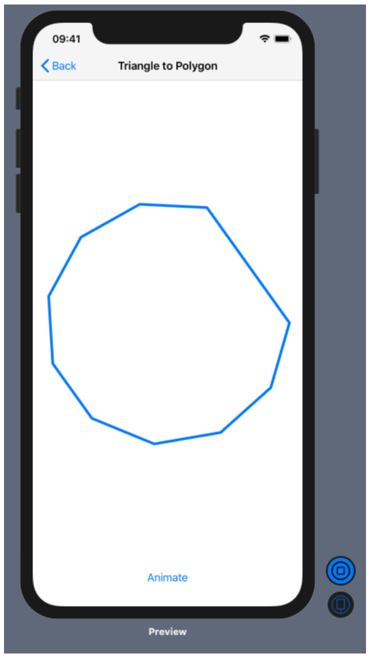
```
struct MyPolygon: Shape {
    var sides: Int
    var doubleSides:Double

    init(sides:Int){
        self.sides = sides
        self.doubleSides = Double(sides)
    }

    var animatableData: Double{
        get{ doubleSides}
        set {doubleSides = newValue}
    }

    func path(in rect: CGRect) -> Path {
        let h = min(rect.size.width,rect.size.height)/2.0
        let c = CGPoint(x: rect.size.width/2.0, y: rect.size.height/2.0)

        var p = Path()
        for i in 0..<sides {
            let angle = 2.0 * .pi*Double(i)/doubleSides
            let pt = CGPoint(x: c.x + CGFloat(cos(angle))*h, y: c.y + CGFloat(sin(angle))*h)
            if i == 0{
                p.move(to:pt)
            }
            else{
                p.addLine(to: pt)
            }
        }
        p.closeSubpath()
        return p
    }
}
```

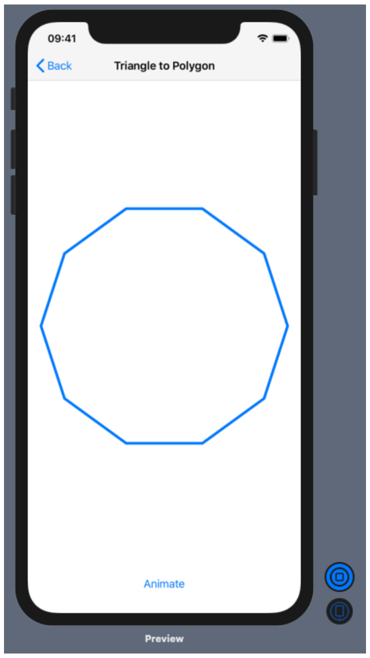
Note that `AnimatableData` needs to be a type that conforms to `VectorArithematic`. We use this in our animation code:

```
struct TriangleToPolygon: View {
    @State private var sides = 3

    var body: some View {
        VStack{
            MyPolygon(sides: sides)
                .stroke(Color.blue, lineWidth: 4)
                .padding(20)

            Button("Animate"){
                withAnimation(.interpolatingSpring(stiffness: 5, damping: 0.80)){

                    self.sides = 10
                    }
            }
        }
    }
}
```

In live preview:







The animation goes through some bizarre intermediate steps. One need run a live preview to see some of that.

Animatable Pair

Quite often we will find ourselves needing to animate more than one parameter. In case, when we need to animate two parameters, we can use `AnimatablePair<First, Second>`. Here we need both `First` and `Second` to conform to `VectorArithmetic`. SwiftUI framework uses `AnimatablePair<First, Second>`quite extensively. For example, `CGSize`, `CGPoint`, `CGRect`. Although these types don't conform to `VectorArithmetic`, they can be animated because they do conform to `Animatable`.

For demonstration, we create an wedge shape that can change in angular width and its angle of offset from a fixed direction. The wedge and its animatable data is created by the code:

```swift
struct MyWedge: Shape {
    var angleOffset: Double // in radians
    var wedgeWidth: Double // in radians

    public var animatableData: AnimatablePair<Double, Double> {
        get {
           AnimatablePair(Double(angleOffset), Double(wedgeWidth))
        }

        set {
            self.angleOffset = newValue.first
            self.wedgeWidth = newValue.second
        }
    }

    func path(in rect: CGRect) -> Path {
        return Path { path in
            let width = Double(rect.width)
            let height = Double(rect.height)

            let middlePoint = CGPoint(x: width/2, y: height/2)
            let startingPoint = CGPoint(x: width/2 + cos(self.angleOffset)*width/2,
                                        y: height/2 + sin(self.angleOffset)*height/2)
            path.move(to: middlePoint)
            path.addLine(to: startingPoint)
            path.addArc(center: middlePoint, radius: rect.width/2, startAngle:
                          Angle(radians: self.angleOffset), endAngle: Angle(radians:
                        self.angleOffset+self.wedgeWidth), clockwise: false,
                        transform: CGAffineTransform.init(scaleX: 1, y:
                        rect.height/rect.width).translatedBy(x: 0,
                                                       y: (rect.width-rect.height)/2))
            path.addLine(to: middlePoint)
        }
    }
}
```

This then animated in code:

```
struct OscillatingWedge: View {
    @State var offset = 0.0
    @State var width = 1.0
    var body: some View {
        VStack{
            MyWedge(angleOffset: offset, wedgeWidth: width)
            .fill(Color.red)
            .frame(width: 350, height: 400)

            Button("Animate"){
                withAnimation(.interpolatingSpring(stiffness: 5.0, damping: 0.5)){
                    self.offset = 1.0
                    self.width = 1.5
                }
            }
        }
    }
}
```

Custom Animatable Type

What should we do if we have more than two parameters to animate. One solution would be to cascade multiple `AnimatablePair` like:

```
AnimatablePair<CGFloat, AnimatablePair<CGFloat, AnimatablePair<CGFloat, CGFloat>>>
```

It is actually used within the implementation of `EdgeInsets` type. It is obviously very flexible.

Most of the time the existing types offer enough flexibility to any animate anything. However, we may find ourselves with a complex type to animate. In such cases, we can provide our own custom implementation.

As an example, we consider the case when the parameters we want to animate is given by members of an array. That is we are dealing with a vector with number of components specified by the number of elements in the array. Let us start with the code:

```swift
struct CustomAnimatableVector {

    var values: [Double]

    init(count: Int = 1){
        self.values = [Double](repeating: 0.0, count: count)
    }

    init(with values: [Double]){
        self.values = values
    }
}
```

We want animate the parameters contained in values. This would require `CustomAnimatableVector` to conform to `VectorArithmetic` protocol. So, we add the extension:

```swift
extension CustomAnimatableVector:VectorArithmetic{

    var magnitudeSquared: Double {
        var sum: Double = 0.0

        for index in 0..<self.values.count {
            sum += self.values[index]*self.values[index]
        }
        return Double(sum)
    }

    mutating func scale(by rhs: Double) {
        for index in 0..<values.count {
            values[index] *= rhs
        }
    }

    static var zero: CustomAnimatableVector {
        CustomAnimatableVector()
    }

    static func - (lhs: CustomAnimatableVector, rhs: CustomAnimatableVector)
                                                        -> CustomAnimatableVector {
        var retValues = [Double]()

        for index in 0..<min(lhs.values.count, rhs.values.count) {
            retValues.append(lhs.values[index] - rhs.values[index])
        }
        return CustomAnimatableVector(with: retValues)
    }

    static func -= (lhs: inout CustomAnimatableVector, rhs: CustomAnimatableVector) {
        lhs = lhs - rhs
    }

    static func + (lhs: CustomAnimatableVector, rhs: CustomAnimatableVector)
                                                       -> CustomAnimatableVector {
           var retValues = [Double]()

           for index in 0..<min(lhs.values.count, rhs.values.count) {
               retValues.append(lhs.values[index] + rhs.values[index])
            }
            return CustomAnimatableVector(with: retValues)
        }

    static func += (lhs: inout CustomAnimatableVector, rhs: CustomAnimatableVector) {
        lhs = lhs + rhs
    }
}
```

The top three properties and method are required by the `VectorArithmetic` protocol. Since `VectorArithmetic` inherits from `AdditiveArithmetic` protocol, we require the remaining method.

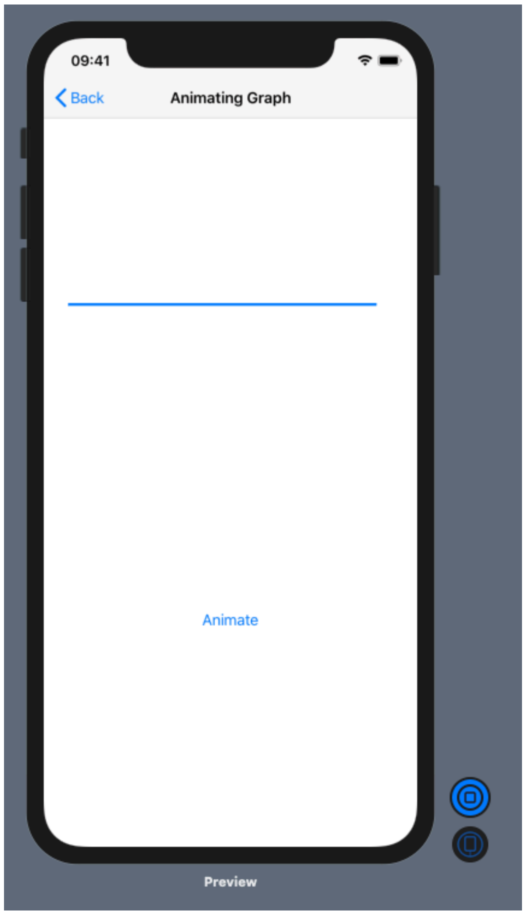
We use the above custom animatable vector in creating a graph:

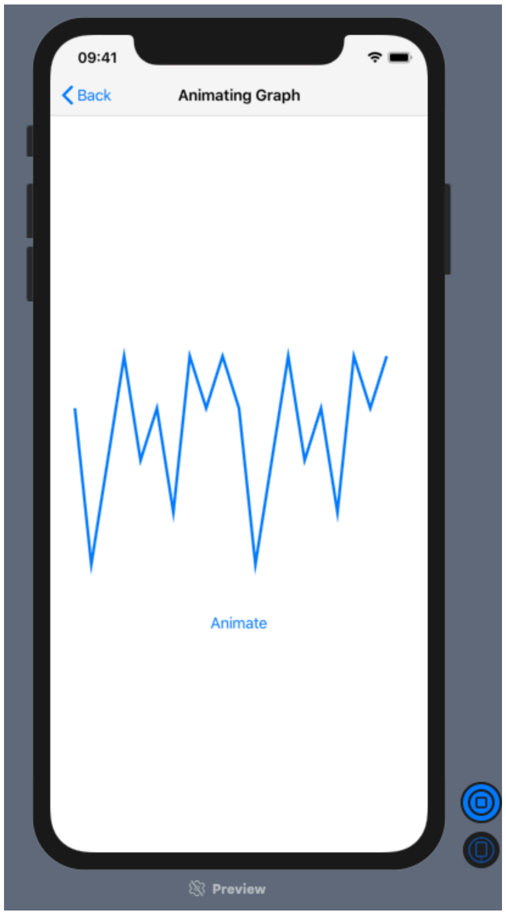
```
struct AnimatableGraph: Shape {
    var controlPoints: CustomAnimatableVector

    var animatableData: CustomAnimatableVector {
        set { self.controlPoints = newValue }
        get { return self.controlPoints }
    }

    func point(index: Int, rect: CGRect) -> CGPoint {
        let value = self.controlPoints.values[index]
        let x = Double(index)/Double(self.controlPoints.values.count)*Double(rect.width)
        let y = Double(rect.height)*value
        return CGPoint(x: x, y: y)
    }

    func path(in rect: CGRect) -> Path {
        return Path { path in

            let startPoint = self.point(index: 0, rect: rect)
            path.move(to: startPoint)

            var i = 1
            while i < self.controlPoints.values.count {
                path.addLine(to:  self.point(index: i, rect: rect))
                i += 1
            }
        }
    }
}
```

Now we are ready to animate:

```
struct AnimatingGraphView: View {
    @State var vector: CustomAnimatableVector =  CustomAnimatableVector(count: 20)

    var body: some View {
        return VStack{

            AnimatableGraph(controlPoints: self.vector)
                .stroke(Color.blue, lineWidth: 3)

            Button("Animate"){
                withAnimation(.interpolatingSpring(stiffness: 2.3, damping: 0.2)){
                        self.vector = CustomAnimatableVector(with: [0.2, 0.5, 0.3,0.1, 0.3,
                        0.2,0.4, 0.1, 0.2,0.1,0.2, 0.5, 0.3,0.1, 0.3, 0.2,0.4, 0.1, 0.2,0.1])
                    }
                }
        }.padding(20)
        .frame(width: 400, height: 400)
    }
}
```

In live preview:





7 Geometry Effect

In the previous section we have used `Animatable` protocol to animate `Paths`. Here we are going to use the same protocol to animate transformation matrices, using a new tool: `GeometryEffect`. `GeometryEffect` is a protocol that conforms to `Animatable` and `ViewModifier`. To conform to `GeometryEffect`, we need to implement the method:

```
func effectValue(size: CGSize) -> ProjectionTransform
```

The return value of `effectValue` method is a `ProjectionTransform` struct which is basically a 3x3 transformation for various transformations, such as scale transformation, translation, and rotation. Note that the `effectValue` method is called continuously during the interpolation of the `offsetValue` property.

We are going to consider several examples of using `GeometryEffect` to produce different types of animation. The examples will be presented in the project "GeometryEffect Animation Demo".

Our first example will produce a shaking animation: a view shakes several times before settling down at its initial position. The view modifier conforming to `GeometryEffect` protocol to produce the shaking animation is

```
struct ShakeEffect: GeometryEffect {
    var amplitude: CGFloat
    var shakeCount:Int
    var animatableData: CGFloat

    func effectValue(size: CGSize) -> ProjectionTransform {
        ProjectionTransform(CGAffineTransform(translationX:
            amplitude * sin(animatableData * .pi * CGFloat(shakeCount)), y: 0))
    }
}
```

Note that the sin function will go through a number of complete cycle depending on the

We make use of this to produce shake animation:

```
struct ContentView: View {
    var body: some View {
        NavigationView{
            List{
                NavigationLink(destination: ShakingAnimationView()){
                    Text("Shaking View Animation")
                }
            }.navigationBarTitle("GeometryEffect Animation Demo", displayMode: .inline)
        }
    }
}
```

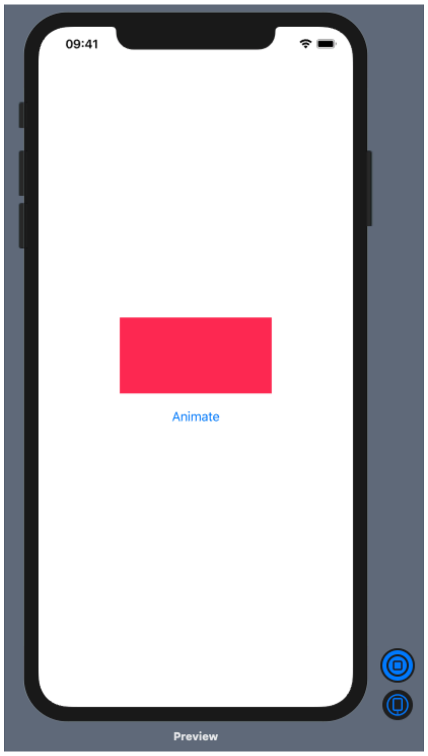
We can create very interesting animations by choosing the transformation matric. In the next example, we animate the skew of a view. The `GeometryEffect` code is now:

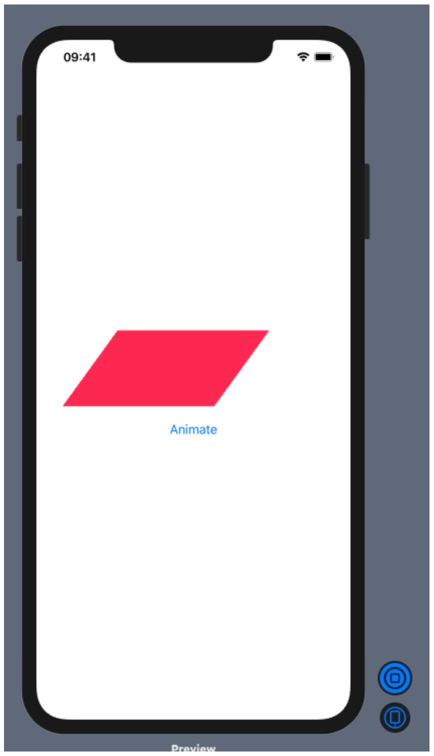
```
struct SkewEffect: GeometryEffect {
    var offset:CGFloat
    var cycle:CGFloat

    init(offset:CGFloat,cycle:CGFloat){
        self.offset = offset
        self.cycle = cycle
    }

    var animatableData: AnimatablePair<CGFloat,CGFloat>{
        get { return AnimatablePair<CGFloat,CGFloat>(offset,cycle)}
        set{
            offset = newValue.first
            cycle = newValue.second
        }
    }

    func effectValue(size: CGSize) -> ProjectionTransform {
        let skew = offset * sin( .pi * cycle)
        return ProjectionTransform(CGAffineTransform(a: 1, b: 0, c: skew, d: 1, tx: 0, ty: 0))
    }
}
```

We apply this modifier in the following:

```
struct SkewAnimationView: View {
    @State var cycle: CGFloat = 0.0

    var body: some View {
        VStack(spacing:20) {
            Rectangle()
               .fill(Color.pink)
                .frame(width: 200, height: 100)
                .modifier(SkewEffect(offset: 0.8, cycle: cycle))

            Button("Animate"){
                withAnimation(.linear(duration: 2)) {
                    self.cycle += 2.0
                   }
               }
        }.navigationBarTitle("Shaking View Animation", displayMode: .automatic)
      }
}
```

In live preview:





We can combine standard modifiers with `GeometryEffect`. In the following example, we have 5 buttons arranged horizontally with a small circle on top of the selected button. When we tap on a different button, the circle jumps to the new selection with animation. The selected button is pushed down a little vertically.

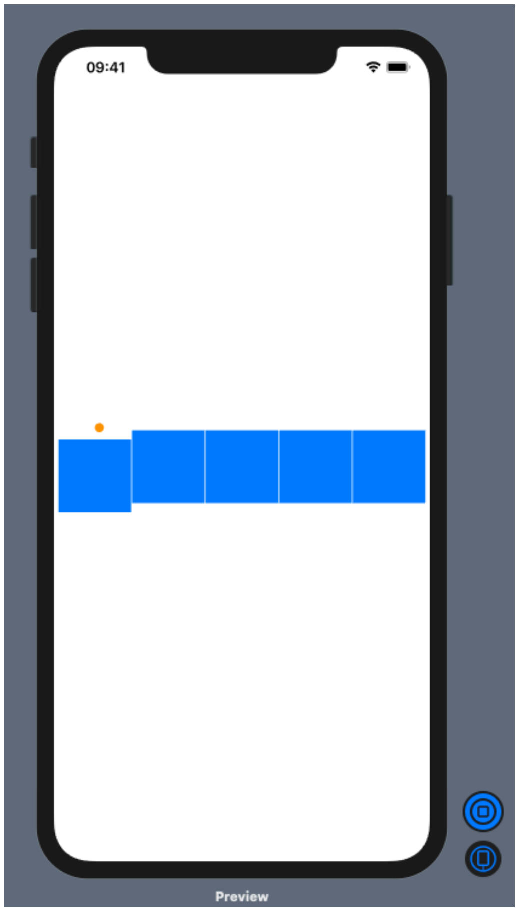
The jump effect is produced by:

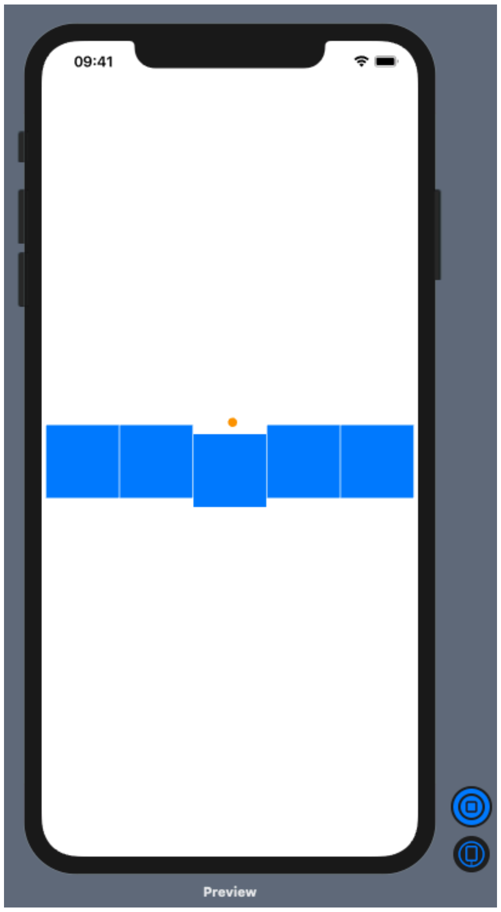
```swift
struct JumpEffect: GeometryEffect {

    var offsetValue: Double

    var animatableData: Double {
        get { offsetValue }
        set { offsetValue = newValue }
    }

    func effectValue(size: CGSize) -> ProjectionTransform {

        let value = 1.0-(cos(2*offsetValue*Double.pi)+1)/2
        let translation   = CGFloat(-50*value)
        let affineTransform = CGAffineTransform(a: 1, b: 0, c: 0, d: 1, tx: 0, ty:
                                                                    translation)

        return ProjectionTransform(affineTransform)
    }
}
```

We use it in the following code:

```swift
struct JumpingAnimationView: View {
    @State var selectedOption : Int = 0
    @State var menuOffset : Double = 0

    let itemWidth: CGFloat = 80

    var body: some View {
        VStack(alignment: .leading) {
            Circle()
                .fill(Color.orange)
                .frame(width:10, height:10)
                .offset(x: CGFloat(self.selectedOption) * self.itemWidth + itemWidth/2.0,
                                                                       y: 10.0 )
                .modifier(JumpEffect(offsetValue: menuOffset))
            HStack(spacing: 1) {
                ForEach(0..<5) { index in
                    Button(action:{
                        withAnimation(.spring()) {
                            self.selectedOption = index
                            self.menuOffset += 1
                        }
                    }) {
                        Rectangle()
                    }
                    .frame(width: self.itemWidth, height: 80)
                    .offset(x: 0, y: self.selectedOption == index ? 10:0)
                }
            }
        }
    }
}
```

In live preview:





In the final example, we will consider a hare running along a closed path. There are two things we need to keep up to date as the animation progresses:

1. The hare's position along the path from the starting point.
2. The orientation of the hare to follow the path direction.

We will express the distance from the starting point by percentage, `pct`: 0 at starting point and 1 at end point.

To determine the x and y position of the hare at a given `pct` value, we are going to use the `trimmedPath()` modifier of the `Path` struct. Given a starting and ending `pct` value, the method returns a `CGRect`. It contains the bounds of that segment of the path. We make this a very small rectangle, and use its canter as our x and y point. We put this together in the code:

```
func percentToPoint(_ percent:CGFloat)-> CGPoint{
    let delta: CGFloat = 0.01
    let comp: CGFloat = 1 - delta

    let pct = percent > 1 ? 0: (percent < 0 ? 1:percent)
    let f = pct > comp ? comp : pct
    let t = pct > comp ? 1: pct + delta
    let tp = path.trimmedPath(from: f, to: t)

    return CGPoint(x: tp.boundingRect.midX, y: tp.boundingRect.midY)
}
```

The following code will determine the direction for the line joining two points:

```
func getDirection(_ pt1:CGPoint, _ pt2:CGPoint)-> CGFloat{

    let a = pt2.x - pt1.x
    let b = pt2.y - pt1.y

    let angle = a < 0 ? atan(Double(b / a)) : atan(Double(b / a)) - Double.pi

    return CGFloat(angle)
}
```

The complete code for `GeometryEffect` modifier is

```
struct FollowEffect: GeometryEffect {
    var pct: CGFloat
    let path:Path
    let rotate:Bool

    var animatableData: CGFloat{
        get { return pct}
        set {pct = newValue }
    }


    func effectValue(size: CGSize) -> ProjectionTransform {
        if !rotate { // Skip rotation login
            let pt = percentToPoint(pct)

            return ProjectionTransform(CGAffineTransform(translationX: pt.x, y: pt.y))
        } else {
            let pt1 = percentToPoint(pct)
            let pt2 = percentToPoint(pct - 0.01)

            let angle = getDirection(pt1, pt2)
            let transform = CGAffineTransform(translationX: pt1.x, y: pt1.y).rotated(by:
                                                                                     angle)

            return ProjectionTransform(transform)
        }
    }

    func percentToPoint(_ percent:CGFloat)-> CGPoint{
        let delta: CGFloat = 0.01
        let comp: CGFloat = 1 - delta

        let pct = percent > 1 ? 0 : (percent < 0 ? 1 : percent)

        let f = pct > comp ? CGFloat(1-delta) : pct
        let t = pct > comp ? CGFloat(1) : pct + delta
        let tp = path.trimmedPath(from: f, to: t)

        return CGPoint(x: tp.boundingRect.midX, y: tp.boundingRect.midY)
    }
    func getDirection(_ pt1:CGPoint, _ pt2:CGPoint)-> CGFloat{

        let a = pt2.x - pt1.x
        let b = pt2.y - pt1.y

        let angle = a < 0 ? atan(Double(b / a)) : atan(Double(b / a)) - Double.pi

        return CGFloat(angle)
    }

}
```

The figure of eight path is drawn by

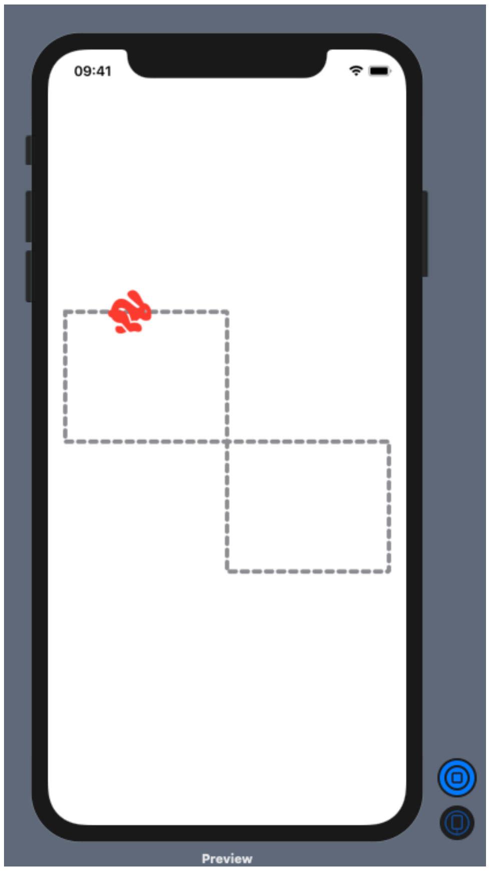
```
struct FigureEightShape: Shape {

    func path(in rect: CGRect) -> Path {
        return FigureEightShape.createPath(in: rect)
    }

    static func createPath(in rect: CGRect) -> Path {
        let height = rect.size.height
        let width = rect.size.width

        var path = Path()

        path.move(to: CGPoint(x:0, y: 0))
        path.addLine(to: CGPoint(x: width/2, y: 0))
        path.addLine(to: CGPoint(x: width/2, y: height))
        path.addLine(to: CGPoint(x: width, y: height))
        path.addLine(to: CGPoint(x: width, y: height/2))
        path.addLine(to: CGPoint(x: 0, y: height/2))
        path.addLine(to: CGPoint(x: 0, y: 0))

        return path
    }
}
```

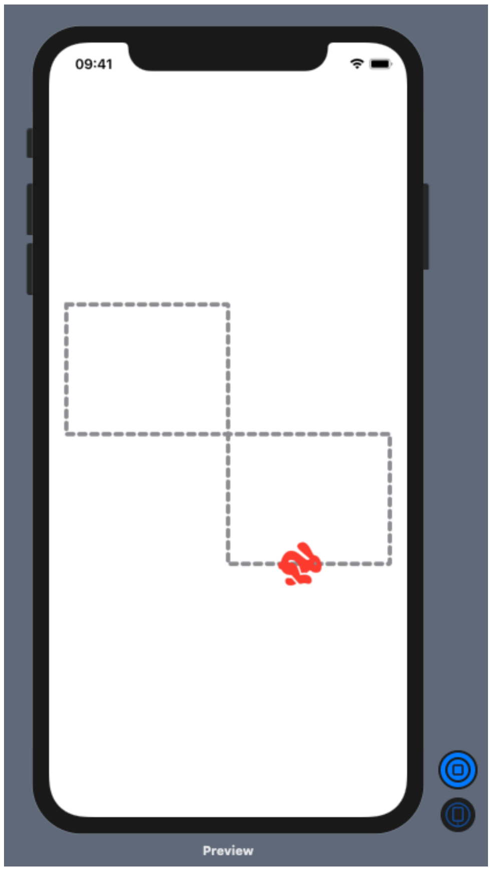
The animation is taken care of by

```
struct RunningHareView: View {
    @State private var flag = false

    var body: some View {
        GeometryReader{ g in
            ZStack(alignment: .topLeading){
                FigureEightShape()
                   .stroke(Color.gray, style: StrokeStyle(lineWidth: 5, lineCap: .round,
                              lineJoin: .miter, miterLimit: 0, dash: [7, 7], dashPhase: 0))
                   .foregroundColor(.blue)
                   .frame(width: g.size.width, height: 300)

                Image(systemName: "hare.fill")
                    .resizable()
                    .foregroundColor(Color.red)
                    .frame(width: 50, height: 50).offset(x: -25, y: -25)
                    .modifier(FollowEffect(pct: self.flag ? 1: 0, path:
                               FigureEightShape.createPath(in: CGRect(x: 0, y: 0, width:
                                                 g.size.width, height: 300)), rotate: true))
                .onAppear {
                    withAnimation(Animation.linear(duration: 24.0).repeatForever(autoreverses:
                                                                                  false)) {
```

```
                    self.flag.toggle()
                }
            }
        }
    }.padding(20)
     .navigationBarTitle("Hare Running Along Path", displayMode: .automatic)
  }
}
```

In live preview:





`GeometryEffect` has a method `.ignoredByLayout()`. Here we show a simple use of the method. The `GeometryEffect` file is

```
struct IgnoreEffect: GeometryEffect {
    var x: CGFloat = 0

    var animatableData: CGFloat {
        get { x }
        set { x = newValue }
    }

    func effectValue(size: CGSize) -> ProjectionTransform {
        return ProjectionTransform(CGAffineTransform(translationX: x, y: 0))
    }
}
```

We use it here

```
struct IgnoreLayoutView: View {
    @State private var animate = false

        var body: some View {
            VStack {
                RoundedRectangle(cornerRadius: 5)
                    .foregroundColor(.green)
                    .frame(width: 300, height: 50)
                    .overlay(ShowSize())
                    .modifier(IgnoreEffect(x: animate ? -10 : 10))

                RoundedRectangle(cornerRadius: 5)
                    .foregroundColor(.blue)
                    .frame(width: 300, height: 50)
                    .overlay(ShowSize())
                    .modifier(IgnoreEffect(x: animate ? 10 : -10).ignoredByLayout())

            }.onAppear {
                withAnimation(Animation.easeInOut(duration: 1.0).repeatForever()) {
                    self.animate = true
                }
            }
        }
    }

struct ShowSize: View {
    var body: some View {
        GeometryReader { proxy in
            Text("x = \(Int(proxy.frame(in: .global).minX))")
                .foregroundColor(.white)
        }
    }
}
```

Since the lower rectangle has the `.ignoredByLayout()`is applied to the size information is not updated.

8. AnimatableModifier

Now we come to the most powerful tool for animation. It is the `AnimatorModifier`, which is a view modifier that conforms to `Animatable` protocol. The `AnimatorModifier` is continuously called during the animation. So, it would continuously update an view created inside the modifier.

As our first example, we consider the case of a `Text` view being continuously updated with the progress of a progress bar. The `AnimatorModifier` code used is

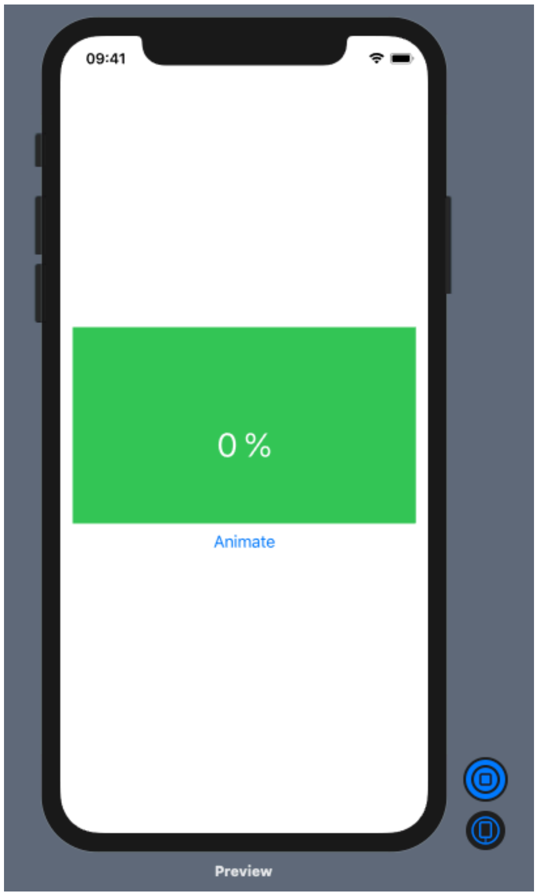
```
struct ProgressBarUpdater: AnimatableModifier {
    var pct: CGFloat = 0

    var animatableData: CGFloat {
        get { pct }
        set { pct = newValue }
    }

    func body(content: Content) -> some View {
        content
            .overlay(LineShape(pct: pct))
            .overlay(LabelView(pct: pct))
    }

    struct LineShape: Shape {
        let pct: CGFloat

        func path(in rect: CGRect) -> Path {

            var p = Path()
            p.move(to: CGPoint(x: 0, y: 30))
            p.addLine(to: CGPoint(x: rect.size.width*pct , y: 30.0))

            return p.strokedPath(.init(lineWidth: 10, dash: [6, 3], dashPhase: 10))
        }
    }

    struct LabelView: View {
        let pct: CGFloat

        var body: some View {
            Text("\(Int(pct * 100)) %")
                .font(.largeTitle)
                .foregroundColor(.white)
                .padding(.top, 40)
        }
    }
}
```

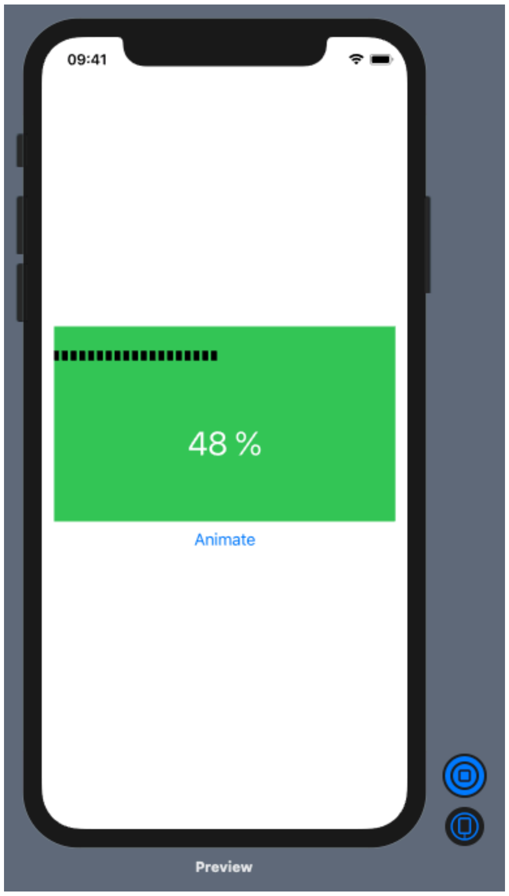
The struct `LineShape` is used to draw the progress bar and the struct `LabelView` updates a `Text` view.

We apply the modifier to a `Rectangle` view:

```
struct TextViewUpdateView: View {
    @State private var pct:CGFloat = 0.0

    var body: some View {
        VStack{

            Rectangle()
                .fill(Color.green)
                .frame(width: 350, height: 200)
                .modifier(ProgressBarUpdater(pct: pct))

            Button("Animate"){
                withAnimation(.linear(duration: 2)){
                    self.pct = 1.0
                }
            }
        }.frame(width: 300, height: 300)
    }
}
```

In live preview:





Note that we don't need to use `content` in the `body` method:

```swift
func body(content: Content) -> some View {

     LineShape(pct: pct)
        .overlay(LabelView(pct: pct))
}
```

Now the green rectangle will disappear (it would be necessary to change text font color in `LabelView` to make it visible against the white background).

Animation of `.foregroundColor(_:)` works fine for views other than `Text` view. It is possibly a bug, which may be remedied in future. For the time being, we can use `AnimatorModifier` to animate `Text` view font color.

To implement animation of `.foregroundColor(_:)` for `Text` view, we need a way to interpolate between two colors. A simple implementation is

```swift
func colorMixer(c1: UIColor, c2: UIColor, pct: CGFloat) -> Color {
    guard let cc1 = c1.cgColor.components else { return Color(c1) }
    guard let cc2 = c2.cgColor.components else { return Color(c1) }

    let r = (cc1[0] + (cc2[0] - cc1[0]) * pct)
    let g = (cc1[1] + (cc2[1] - cc1[1]) * pct)
    let b = (cc1[2] + (cc2[2] - cc1[2]) * pct)

    return Color(red: Double(r), green: Double(g), blue: Double(b))
}
```

Then our `AnimatorModifier` is

```
struct TextColorAnimationModifier: AnimatableModifier {
    let from: UIColor
    let to: UIColor
    var pct: CGFloat
    let text: Text

    var animatableData: CGFloat {
        get { pct }
        set { pct = newValue }
    }

    func body(content: Content) -> some View {
        return text.foregroundColor(colorMixer(c1: from, c2: to, pct: pct))
    }

    func colorMixer(c1: UIColor, c2: UIColor, pct: CGFloat) -> Color {
        guard let cc1 = c1.cgColor.components else { return Color(c1) }
        guard let cc2 = c2.cgColor.components else { return Color(c1) }

        let r = (cc1[0] + (cc2[0] - cc1[0]) * pct)
        let g = (cc1[1] + (cc2[1] - cc1[1]) * pct)
        let b = (cc1[2] + (cc2[2] - cc1[2]) * pct)

        return Color(red: Double(r), green: Double(g), blue: Double(b))
    }
}
```

We use it in the code:

```
struct TextColorAnimationView: View {
    @State private var flag = false

    var body: some View {
        VStack(spacing: 20){
            Text("")
                .modifier(TextColorAnimationModifier(from: .red, to: .blue,
                                   pct: flag ? 1.0 : 0.0,
                                         text: Text("Hello from SwiftUI").font(.largeTitle)))

            Button("Animate"){
                withAnimation(.linear(duration: 2)){
                   self.flag.toggle()
                }
            }
        }
    }
}
```

On tap of the button the color will change from red to blue with animation, another tap will change it back to red.

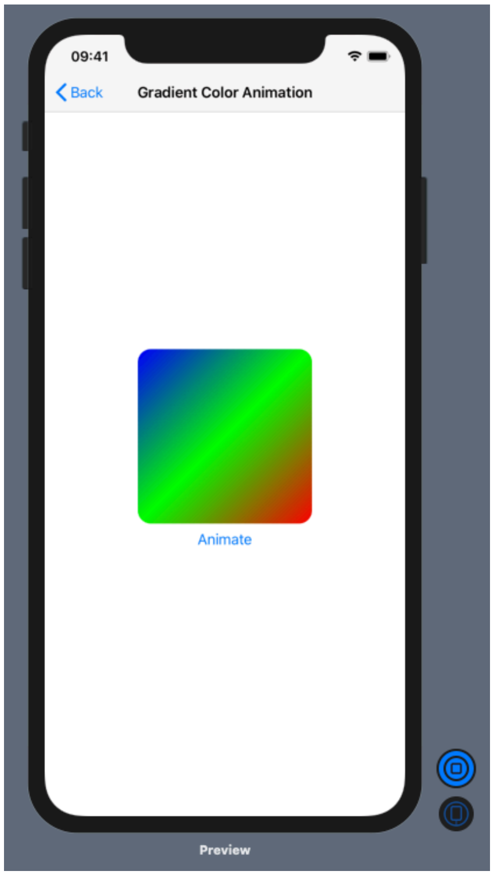
Animation of gradient is limited. We can animate the start and end points, but not the gradient color. We can use the color mixture above to animate the gradient color as well.

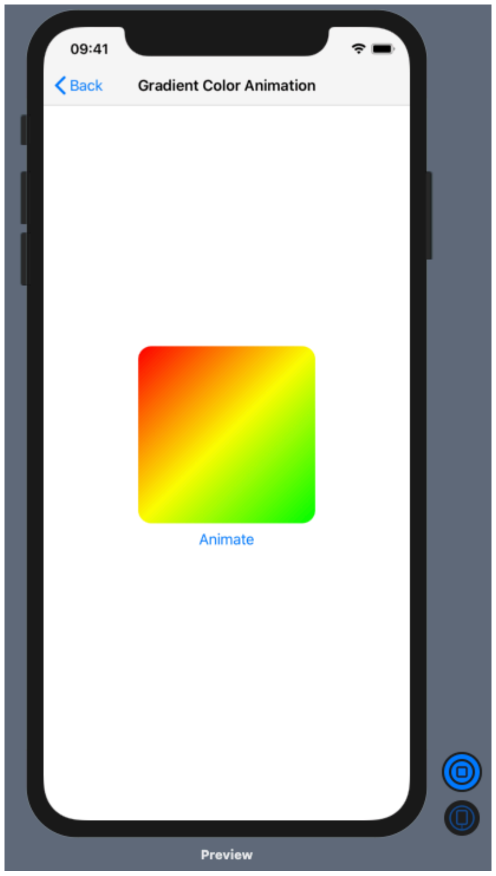
```swift
struct GradientModifier: AnimatableModifier {
    let from: [UIColor]
    let to: [UIColor]
    var pct: CGFloat = 0

    var animatableData: CGFloat {
        get { pct }
        set { pct = newValue }
    }

    func body(content: Content) -> some View {
        var gColors = [Color]()

        for i in 0..<from.count {
            gColors.append(colorMixer(c1: from[i], c2: to[i], pct: pct))
        }

        return RoundedRectangle(cornerRadius: 15)
            .fill(LinearGradient(gradient: Gradient(colors: gColors),
                                 startPoint: UnitPoint(x: 0, y: 0),
                                 endPoint: UnitPoint(x: 1, y: 1)))
            .frame(width: 200, height: 200)
    }

    func colorMixer(c1: UIColor, c2: UIColor, pct: CGFloat) -> Color {
        guard let cc1 = c1.cgColor.components else { return Color(c1) }
        guard let cc2 = c2.cgColor.components else { return Color(c1) }

        let r = (cc1[0] + (cc2[0] - cc1[0]) * pct)
        let g = (cc1[1] + (cc2[1] - cc1[1]) * pct)
        let b = (cc1[2] + (cc2[2] - cc1[2]) * pct)

        return Color(red: Double(r), green: Double(g), blue: Double(b))
    }
}
```

We use it in the following code:

```swift
struct GradientAnimationView: View {
    @State private var flag = false

    var body: some View {
        let grad1: [UIColor] = [.blue, .green, .red]
        let grad2: [UIColor] = [.red, .yellow, .green]

        return VStack{
            Color.clear
                .frame(width: 200, height: 150)
                .modifier(GradientModifier(from: grad1, to: grad2, pct: flag ? 1: 0))

            Button("Animate"){
                withAnimation(.linear(duration: 2)){
                    self.flag.toggle()
                }
            }
        }
    }
}
```

In live preview:


09:41
Back
Gradient Color Animation
Animate
Preview


09:41
Back
Gradient Color Animation
Animate
Preview