The SwiftUI Lab

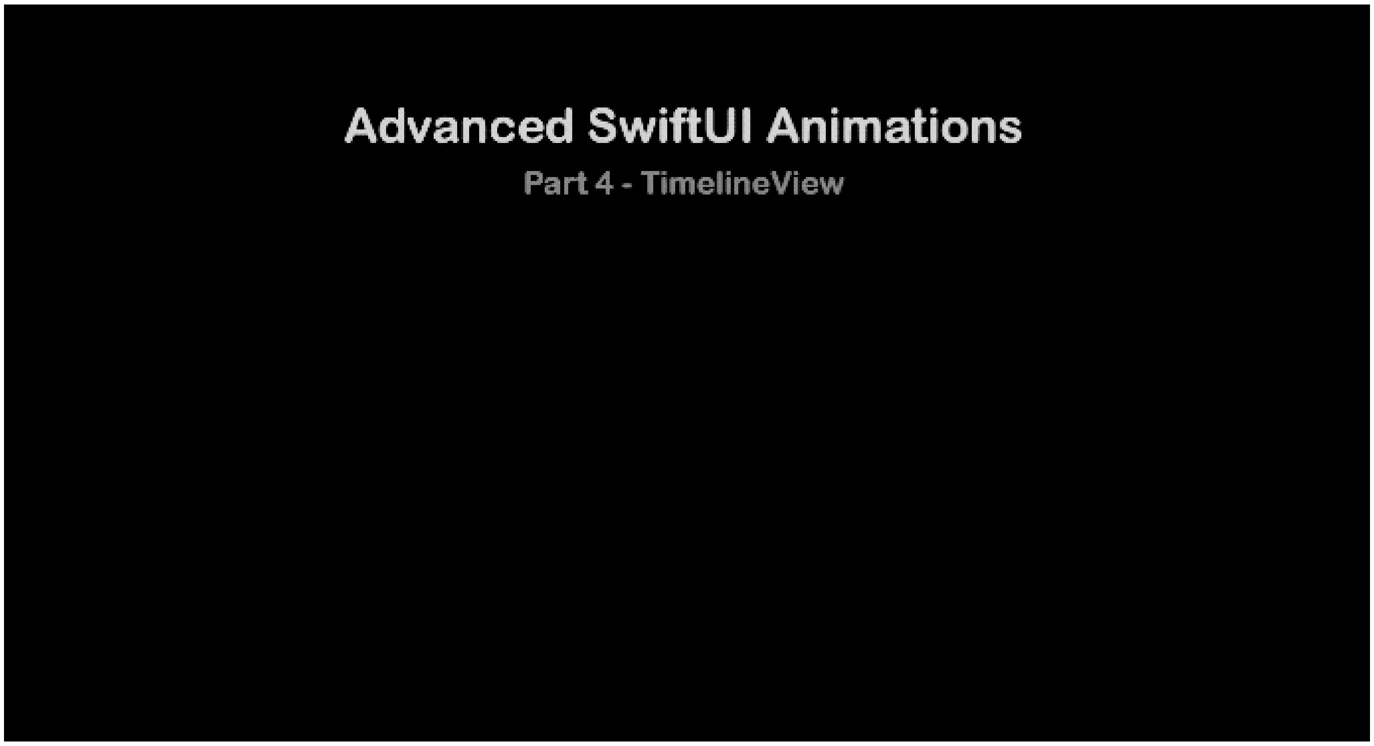
When the documentation is missing, we experiment.

Menu

Advanced SwiftUI Animations — Part 4: TimelineView

June 28, 2021 by javier

It's been two years since I published Part 3 of this series of articles about **Advanced SwiftUI Animations**. I'm super excited about this year's WWDC introduction of `TimelineView` and `Canvas`. It opens a whole new range of possibilities that I will try to lay out in this and the next part of the series.



In this post, we will explore `TimelineView` in detail. We will start slow, with the most common uses. However, I think the biggest potential comes with combining `TimelineView` and the existing animations we already know. By being a little creative, among other things, this combo will let us finally do "keyframe-like" animations.

In part 5, we will explore the `Canvas` view, and how great it is in combination with our new friend `TimelineView`.

The animation shown above has been created using the techniques explained in this article. The full code of that animation is available in this gist.

Components of a TimelineView

TimelineView is a container view that re-evaluates its content with a frequency determined by the associated scheduler:

```
TimelineView(.periodic(from: .now, by: 0.5)) { timeline in

    ViewToEvaluatePeriodically()

}
```

The `TimelineView` receives a scheduler as a parameter. We will look at them in detail later, for now, the example uses a scheduler that fires every half a second.

The other parameter is a content closure that receives a `TimelineView.Context` parameter that looks something like this:

```
struct Contex
    let cadence: Cadence
```

```
        let date: Date

        enum Cadence: Comparable {
            case live
            case seconds
            case minutes
        }
    }
```

A `Cadence` is an enum we can use to make some decisions on what to show in our view. Possible values are: `live`, `seconds`, and `minutes`. Take this as a hint to avoid showing information irrelevant for the cadence. The typical example is to avoid showing milliseconds on a clock that has a scheduler with a `seconds` or `minutes` cadence.

Note that the cadence is not something you can change, but something that reflects a device state. The documentation provides only one example. On watchOS, the cadence slows down when lowering the wrist. If you find other cases where the cadence changes, I would very much like to know. Please comment below.

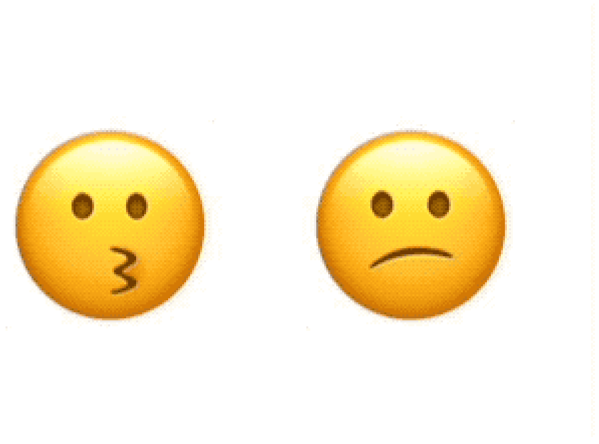
Well, this all looks great, but there are many subtleties we should be aware of. Let's start building our first `TimelineView` animation to see what these are…

Understanding How TimelineView Works

Look at the following code. We have two emoji characters that change randomly. The only difference between the two is that one is written in the content closure, while the other has been put on a separate view for better readability.

```
    struct ManyFaces: View {
        static let emoji = ["😀", "😬", "😄", "🙂", "😗", "🤓", "😏", "😕", "😟", "😎", "😜", "😍", "

        var body: some View {
            TimelineView(.periodic(from: .now, by: 0.2)) { timeline in

                HStack(spacing: 120) {

                    let randomEmoji = ManyFaces.emoji.randomElement() ?? ""

                    Text(randomEmoji)
                        .font(.largeTitle)
                        .scaleEffect(4.0)

                    SubView()

                }
            }
        }

        struct SubView: View {
            var body: some View {
                let randomEmoji = ManyFaces.emoji.randomElement() ?? ""

                T
                    .font(.largeTitle)
                    .scaleEffect(4.0)
            }
        }
    }
```

Now, let's see what happens when we run the code:



Shocked? Why does the left emoji change, but the other remains sad at all times? It turns out, the `SubView` is not receiving any changing parameters, which means it has no dependencies. SwiftUI has no reason to recompute the view's body. A great talk from this year's WWDC is Demystify SwiftUI. It explains view identify, lifetime and dependency. All these topics are very important to understand why the timeline behaves as it does.

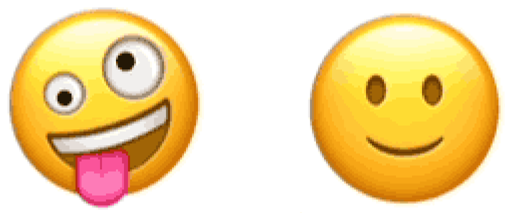
To solve the problem, we change the `SubView` view to add a parameter that will change with every timeline update. Note that we do not need to use the parameter, it just has to be there. Nevertheless, we will see that this unused value will be quite useful later.

```
struct SubView: View {
    let date: Date // just by declaring it, the view will now be recomputed apropriately.

    var body: some View {

        let randomEmoji = ManyFaces.emoji.randomElement() ?? ""

        Text(randomEmoji)
            .font(.largeTitle)
            .scaleEffect(4.0)
    }
}
```

Now the `SubView` is created like this:

```
SubView(date: timeline.date)
```

Finally, both our emoji can experience a whirlwind of emotions:



Acting Upon a Timeline

Most examples about TimelineView (at the time of this writing), are usually about drawing clocks. That makes sense. The data provided by the timeline is, after all, a `Date`.

The easiest TimelineView clock ever:

```
TimelineView(.periodic(from: .now, by: 1.0)) { timeline in

    Text("\(timeline.date)")

}
```

Clocks may become a little more elaborate. For example, using an analog clock with shapes, or drawing the clock with the new Canvas view.

However, `TimelineView` is here for more than just clocks. In many cases, we want our view to do something every time the timeline updates our view. The perfect place to put this code is the `onChange(of:perform)` closure.

In the following example, we use this technique, to update our model every 3 seconds.

Haikus are easy. But sometimes they don't make sense. Refrigerator.

```
struct ExampleView: View {
    var body: some View {
        TimelineView(.periodic(from: .now, by: 3.0)) { timeline in
            QuipView(date: timeline.date)
        }
    }

    struct QuipView: View {
        @StateObject var quips = QuipDatabase()
        let d

        var body: some View {
                Text("_\(quips.sentence)_")
                    .onChange(of: date) { _ in
                        quips.advance()
                    }
            }
        }
    }

    class QuipDatabase: ObservableObject {
        static var sentences = [
            "There are two types of people, those who can extrapolate from incomplete data",
            "After all is said and done, more is said than done.",
            "Haikus are easy. But sometimes they don't make sense. Refrigerator.",
            "Confidence is the feeling you have before you really understand the problem."
        ]

        @Published var sentence: String = QuipDatabase.sentences[0]

        var idx = 0

        func advance() {
            idx = (idx + 1) % QuipDatabase.sentences.count

            sentence = QuipDatabase.sentences[idx]
        }
    }
```

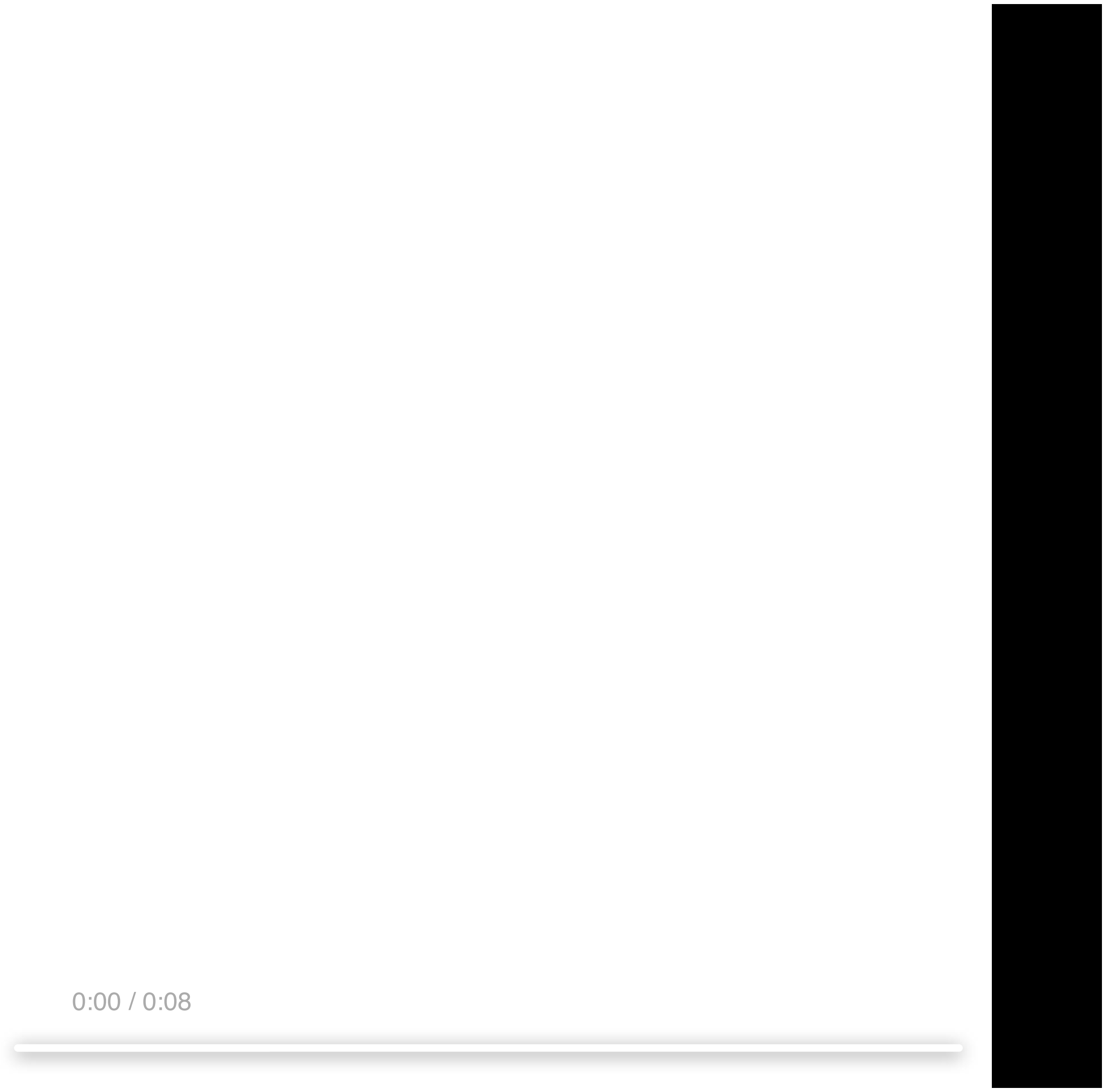
It's important to notice that for every timeline update, our `QuipView` is refreshed twice. That is, once when the timeline updates, and then again immediately after, because by calling `quips.advance()` we are causing the @Published value of `quips.sentence` to change and trigger the view update. This is perfectly fine, but it is something to be aware of, as it will become more important later on.

> *An important concept we take from this is that although the timeline may produce a certain number of updates, the content of the view will most likely update even more times.*

Combining TimelineView with Traditional Animations

The new `TimelineView` brings a lot of new opportunities. Combining it with `Canvas`, as we will see in a future post, is a great addition. But that puts the load of writing all the code for each frame of the animation on us. The technique I am going to expose in this section, uses the animations we already know and love to animate views from one timeline update to the next. This will finally let us create our own keyframe-like animations purely in SwiftUI.

But let's begin slow, with our little project: the metronome shown below. Play the video with the volume up, to appreciate how the beat sound is synchronized with the pendulum. Also, as metronomes do, a bell sounds every few beats:



First, let’s see what our timeline looks like:

```
struct Metronome: View {
    let bpm: Double = 60 // beats per minute

    var body: some View {
        TimelineView(.periodic(from: .now, by: 60 / bpm)) { timeline in
            MetronomeBack()
                .overlay(MetronomePendulum(bpm: bpm, date: timeline.date))
                .overlay(MetronomeFront(), alignment: .bottom)
        }
    }
}
```

Metronomes speeds are usually specified in bpm (beats per minute). The example uses a periodic scheduler, that repeats every 60/

minute.

The Metronome view is composed of three layers: The MetronomeBack, MetronomePendulum, and MetronomeFront. They are overlaid in that order. The only view that will have to refresh with every timeline update, will be the MetronomePendulum, which swings from side to side. The other views won't refresh, because they have no dependencies.

The code for MetronomeBack and MetronomeFront is very simple, they use a custom shape called RoundedTrapezoid. To avoid making this page too long, the code for the custom shape is in this gist.

```
struct MetronomeBack: View {
    let c1 = Color(red: 0, green: 0.3, blue: 0.5, opacity: 1)
    let c2 = Color(red: 0, green: 0.46, blue: 0.73, opacity: 1)

    var body: some View {
        let gradient = LinearGradient(colors: [c1, c2],
                                      startPoint: .topLeading,
                                      endPoint: .bottomTrailing)

        RoundedTrapezoid(pct: 0.5, cornerSizes: [CGSize(width: 15, height: 15)])
            .foregroundStyle(gradient)
            .frame(width: 200, height: 350)
    }
}

struct MetronomeFront: View {
    var body: some View {
        RoundedTrapezoid(pct: 0.85, cornerSizes: [.zero, CGSize(width: 10, height: 10)])
            .foregroundStyle(Color(red: 0, green: 0.46, blue: 0.73, opacity: 1))
            .frame(width: 180, height: 100).padding(10)
    }
}
```

The MetronomePendulum view, however, is where things start to get interesting:

```
struct MetronomePendulum: View {
    @State var pendulumOnLeft: Bool = false
    @State var bellCounter = 0 // sound bell every 4 beats

    let bpm: Double
    let date: Date

    var body: some View {
        Pendulum(angle: pendulumOnLeft ? -30 : 30)
            .animation(.easeInOut(duration: 60 / bpm), value: pendulumOnLeft)
            .onChange(of: date) { _ in beat() }
            .onAppear { beat() }
    }

    func beat() {
        pendulumOnLeft.toggle() // triggers the animation
        bellCounter = (bellCounter + 1) % 4 // keeps count of beats, to sound bell every 4th

        // so
        if bellCounter == 0 {
            bellSound?.play()
        } else {
            beatSound?.play()
        }
    }

    struct Pendulum: View {
        let angle: Double

        var body: some View {
            return Capsule()
                .fill(.red)
                .frame(width: 10, height: 320)
                .overlay(weight)
                .rotationEffect(Angle.degrees(angle), anchor: .bottom)
        }

        var weight: some View {
            RoundedRectangle(cornerRadius: 10)
                .fill(.orange)
                .frame(width: 35, height: 35)
                .padding(.bottom, 200)
        }
    }
}
```

Our view needs to keep track of where we are in the animation. I will call this, the animation phase. Since we need to track these phases, we will use @State variables:

1. `pendulumOnLeft`: keeps track of which way the pendulum is swinging.
2. `bellCounter`: it keeps count of the number of beats, to determine if a beat or a bell should be heard.

The example uses the `.animation(_:value:)` modifier. This version of the modifier, applies an animation when the specified value changes. Note that it is also possible to use an explicit animation. Instead of calling `.animation()`, simply toggle the `pendulumOnLeft` variable inside a `withAnimation` closure.

To make our view advance through the animation phases, we monitor changes in `date`, using the `onChange(of:perform)` modifier, as we did with the previous quip example.

In addition to advancing the animation phase every time the date value changes, we also do it in the `onAppear` closure. Otherwise, there would be a pause at the beginning.

The final piece of code, non-SwiftUI related, is creating the NSSound instances. To avoid overcomplicating the example, I just created a couple of global variables:

```
let bellSound: NSSound? = {
    guard let url = Bundle.main.url(forResource: "bell", withExtension: "mp3") else { return nil }
    return NSSound(contentsOf: url, byReference: true)
}()

let beatSound
    guard let url = Bundle.main.url(forResource: "beat", withExtension: "mp3") else { return nil }
    return NSSound(contentsOf: url, byReference: true)
```

```
}()
```

If you need sound files, there is a large database available at: https://freesound.org/

The ones in the example are:

- **bell sound**: metronome_pling under license CC BY 3.0 (m1rk0)
- **beat sound**: metronome.wav under license CC0 1.0 (Druminfected)

The TimelineScheduler

As we've seen already, a `TimelineView` needs a `TimelineScheduler` to determine when to update its contents. SwiftUI provides some predefined schedulers, like the ones we used. However, we can also create our own custom scheduler. I will elaborate more on that in the next section. But let's start with the pre-existing ones.

A timeline scheduler is basically a struct that adopts the `TimelineScheduler` protocol. The existing types are:

- `AnimationTimelineSchedule`: Updates as fast as possible, giving you the chance to draw each frame of the animation. It has parameters that let you limit the frequency of updates, and pause the updates. This one will be very useful when combining `TimelineView` with the new `Canvas` view.
- `EveryMinuteTimelineSchedule`: As the name implies, it updates every minute, at the start of the minute.
- `ExplicitTimelineSchedule`: You may provide an array with all the times you want the timeline to update.
- `PeriodicTimelineSchedule`: You may provide a start time and a frequency at which updates occur.

Although you could create a timeline in this fashion:

```
Timeline(EveryMinuteTimelineSchedule()) { timeline in
    ...
}
```

Since Swift 5.5 and the introduction of SE-0299, we now have support for enum-like syntax. This makes the code more readable and improves autocompletion. It is recommended that we use this syntax instead:

```
TimelineView(.everyMinute) { timeline in
    ...
}
```

Note: You may have heard, but this has also been introduced with styles this year. And better yet, for styles, as long as you are using Swift 5.5, you may back-deploy it with previous versions.

For each of the existing schedulers, there may be more than one enum-like option. For example, these two lines create a schedul

```
TimelineView(.animation) { ... }

TimelineView(.animation(minimumInterval: 0.3, paused: false)) { ... }
```

And you may even create your own (do not forget the static keyword!):

```
extension TimelineSchedule where Self == PeriodicTimelineSchedule {
    static var everyFiveSeconds: PeriodicTimelineSchedule {
        get { .init(from: .now, by: 5.0) }
    }
}

struct ContentView: View {
    var body: some View {
        TimelineView(.everyFiveSeconds) { timeline in
            ...
        }
    }
}
```

Custom TimelineScheduler

If none of the existing schedulers fit your needs, you may create your own. Consider the following animation:

In this animation, we have a heart emoji that changes its scale, at irregular intervals and irregular amplitudes:

It starts with a scale of 1.0, 0.2 seconds later grows to 1.6, 0.2 seconds later, grows to 2.0, then shrinks back to 1.0 and stays there for 0.4 seconds, before starting all over. In other words:

Scale changes: 1.0 → 1.6 → 2.0 → start again
Time between changes: 0.2 → 0.2 → 0.4 → start again

We could create a `HeartTimelineSchedule` that updates exactly as the heart requires. But in the name of reusability, let's do something more generic that can be reused in the future.

Our new scheduler will be called: `CyclicTimelineSchedule` and will receive an array of time offsets. Each offset value will be relative to the previous value in the array. When the scheduler has exhausted the offsets, it will cycle back to the beginning of the array and start all over.

```
struct CyclicTimelineSchedule: TimelineSchedule {
    let timeOffsets: [TimeInterval]
    func entries(from startDate: Date, mode: TimelineScheduleMode) -> Entries {
        Entries(last: startDate, offsets: timeOffsets)
    }

    struct Entries: Sequence, IteratorProtocol {
        var last: Date
        let offsets: [TimeInterval]

        var idx: Int = -1

        mutating func next() -> Date? {
            idx = (idx + 1) % offsets.count

            last = last.addingTimeInterval(offsets[idx])

            return last
        }
    }
}
```

There are a couple of requirements to implement a TimelineSchedule:

- Provide the `entries(from:mode:)` function.
- Our `Entries` type must conform to `Sequence` where `Entries.Element == Date`

There are several ways in which you can conform to `Sequence`. This example implements `IteratorProtocol` and declares conformance to both `Sequence` and `IteratorProtocol`. You can read more about Sequence conformance here.

For `Entries` to implement `IteratorProtocol`, we must write the `next()` function, which produces the dates in the timeline. Our scheduler remembers the last date and adds the appropriate offset. When no more offsets remain, it cycles back to the first in the array.

Finally, the icing on the cake for our scheduler is to create an enum-like initializer:

```
extension TimelineSchedule where Self == CyclicTimelineSchedule {
    static func cyclic(timeOffsets: [TimeInterval]) -> CyclicTimelineSchedule {
            .init(timeOffsets: timeOffsets)
    }
}
```

Now that we have our `TimelineSchedue` type ready, let's put some life into our heart:

```
struct BeatingHeart: View {
    var body: some View {
        TimelineView(.cyclic(timeOffsets: [0.2, 0.2, 0.4])) { timeline in
            Heart(date: timeline.date)
        }
    }
}
```

```
struct Heart: View {
    @State private var phase = 0
    let scales: [CGFloat] = [1.0, 1.6, 2.0]

    let date: Date

    var body: some View {
        HStack {
            Text("❤️")
                .font(.largeTitle)
                .scaleEffect(scales[phase])
                .animation(.spring(response: 0.10,
                                   dampingFraction: 0.24,
                                   blendDuration: 0.2),
                           value: phase)
                .onChange(of: date) { _ in
                    advanceAnimationPhase()
                }
                .onAppear {
                    advanceAnimationPhase()
                }

        }
    }

    func advanceAnimationPhase() {
        phase = (phase + 1) % scales.count
    }
}
```

You should be familiar with this pattern now, it is the same one we used with the metronome. Advance the animation with onChange and onAppear, use `@State` variables to keep track of the animation, and set an animation that will transition our view from one timeline update, to the next. In this case, we employ a `.spring` animation, giving it a nice shake effect.

KeyFrame Animations

The heart and metronome examples are, in a way, keyframe animations. We defined several key points in the whole animation, where we change the parameters of our view, and let SwiftUI animate the transition between these points. The following example will try to generalize that idea, and make it more evident. Meet our new project friend, the jumping guy:

If you observe the animation carefully, you will notice that this emoji character has many of its parameters changed at different points in time. These parameters are: y-offset, rotation and y-scale. And also important, the different segments of the animation, have different animation types (`linear`, `easeIn` and `easeOut`). Since these are the parameters we change, it is a good idea to put them together in an array. Let's begin:

```
struct KeyFrame {
    let offset: TimeInterval
    let rotation: Double
    let yScale: Double
    let y: CGFloat
    let animation: Animation?
}

let keyframes = [
    // Initial state, will be used once. Its offset is useless and will be ignored
    KeyFrame(offset: 0.0, rotation: 0, yScale: 1.0, y: 0, animation: nil),

    // Animation keyframes
    KeyFrame(offset: 0.2, rotation:   0, yScale: 0.5, y:  20, animation: .linear(duration: 0.2)),
    KeyFrame(offset: 0.4, rotation:   0, yScale: 1.0, y: -20, animation: .linear(duration: 0.4)),
    KeyFrame(offset: 0.5, rotation: 360, yScale: 1.0, y: -80, animation: .easeOut(duration: 0.5)),
    KeyFrame(offset: 0.4, rotation: 360, yScale: 1.0, y: -20, animation: .easeIn(duration: 0.4)),
    KeyFrame(offset: 0.2, rotation: 360, yScale: 0.5, y:  20, animation: .easeOut(duration: 0.2)),
    KeyFrame(offset: 0.4, rotation: 360, yScale: 1.0, y: -20, animation: .linear(duration: 0.4)),
    KeyFrame(offset: 0.5, rotation:   0, yScale: 1.0, y: -80, animation: .easeOut(duration: 0.5)),
    KeyFrame(offset: 0.4, rotation:   0, yScale: 1.0, y: -20, animation: .easeIn(duration: 0.4)),
]
```

It is important to know that when `TimelineView` appears, it will draw our view, even if there are no scheduled updates, or if they are in the future. When the `TimelineView` appears, it needs to show something so it does draw our view. We are going to use the first keyframe for the state of our view at that point, but when we loop, that frame will be ignored. This is an implementation decision, you may need or want to do it differently.

Now, let's look at our timeline:

```
struct Jumpin
    // Use al
    let offse
```

By continuing to use the site, you agree to the use of cookies. more information Accept

```
    var body: some View {
        TimelineView(.cyclic(timeOffsets: offsets)) { timeline in
            HappyEmoji(date: timeline.date)
        }
    }
}
```

We are already benefiting from the work we did on the previous example, and reusing the `CyclicTimelineScheduler`. As mentioned, we do not need the offset of the first keyframe, so we discard it.

Now the fun part:

```
struct HappyEmoji: View {
    // current keyframe number
    @State var idx: Int = 0

    // timeline update
    let date: Date

    var body: some View {
        Text("😃")
            .font(.largeTitle)
            .scaleEffect(4.0)
            .modifier(Effects(keyframe: keyframes[idx]))
            .animation(keyframes[idx].animation, value: idx)
            .onChange(of: date) { _ in advanceKeyFrame() }
            .onAppear { advanceKeyFrame()}
    }

    func advanceKeyFrame() {
        // advance to next keyframe
        idx = (idx + 1) % keyframes.count

        // skip first frame for animation, which we
        // only used as the initial state.
        if idx == 0 { idx = 1 }
    }

    struct Effects: ViewModifier {
        let keyframe: KeyFrame

        func body(content: Content) -> some View {
            content
                .scaleEffect(CGSize(width: 1.0, height: keyframe.yScale))
                .rotationEffect(Angle(degrees: keyframe.rotation))
                .offset(y: keyframe.y)
        }
    }
}
```

For better readability, I put all the changing parameters inside a modifier, called `Effects`. As you can see, it is again the same p keyframe segment. Nothing new there.

Don't! It's a Trap!

In your path to the `TimelineView` discovery, you may encounter this error:

```
Action Tried to Update Multiple Times Per Frame
```

Let's see an example that generates this message:

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

    var body: some View {

        TimelineView(.periodic(from: .now, by: 2.0)) { timeline in

            Text("Hello")
                .foregroundStyle(flag ? .red : .blue)
                .onChange(of: timeline.date) { (date: Date) in
                    flag.toggle()
                }

        }
    }
}
```

The code looks harmless, and it is supposed to change the text color every two seconds, alternating between red and blue. So what could be going on? Just pause for a second and see if you can spot the reason behind it.

We are not dealing with a bug. The problem is, in fact, predictable.

It is important to remember that the first update of a timeline is when it appears for the first time, then it follows the scheduler rules to trigger the following updates. So even if our scheduler produces no update, the TimelineView content is generated at least once.

In this specific example, we monitor for changes in the `timeline.date` value, and when it does change, we toggle the `flag` variable, which produces the color change.

The `TimelineView` will first appear. Two seconds later, the timeline will update (e.g., due to the first scheduler update), triggering the onChange closure. This will in turn change the `flag` variable. Now, since our `TimelineView` has a dependency on it, it will need to refresh immediately, triggering another toggle of the `flag` variable, forcing another `TimelineView` refresh, and so on, and so on… You got it: multiple updates per frame.

So how do we fix it? Solutions may vary. In this case, we simply encapsulate the contents and move the `flag` variable inside the encapsulated view. Now the `TimelineView` no longer depends on it:

```
struct Exampl
    var body: some View {
        TimelineView(.periodic(from: .now, by: 1.0)) { timeline in
            SubView(date: timeline.date)
        }

    }
}

struct SubView: View {
    @State private var flag = false
    let date: Date

    var body: some View {
        Text("Hello")
            .foregroundStyle(flag ? .red : .blue)
            .onChange(of: date) { (date: Date) in
                flag.toggle()
            }
    }
}
```

Exploring New Ideas

Refreshing Once Per Timeline Update: As mentioned before, this pattern makes our views to compute their bodies twice per update: first when the timeline updates, and then again when we advance our animation state values. In this type of animation where we have spaced key points in time, that is perfectly fine.

In animations where those points in time are too close together maybe you need/want to avoid that. If you need to change a stored value, but avoid a view refresh… there's a trick you can do. Instead of `@State`, use `@StateObject`. Make sure you DO NOT make such value `@Published`. If at some point, you want/need to tell your view to refresh, you can always call `objectWillChange.send()`

Matching Animation Duration and Offsets:In the keyframe example, we use different animations for each of the animation segments. To do so, we store the Animation values in our array. If you look closer, you'll see that in our specific example, the offsets and the animation durations match! It makes sense, right? So instead of having the Animation value in your array, you may define an enum with the kind of animations. Later in your view, you create the Animation value, based on the animation kind, but instantiate it with the duration from the offset value. Something like this:

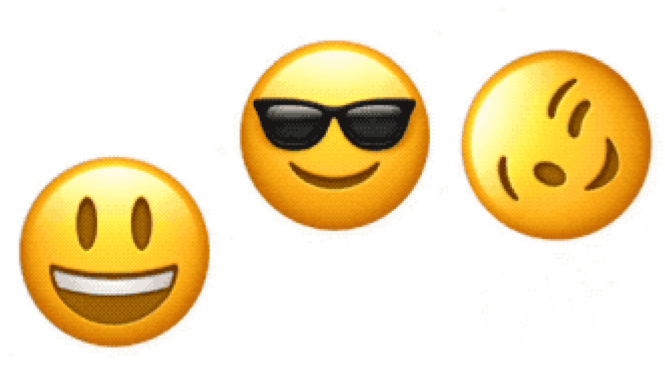
```
enum KeyFrameAnimation {
    case none
    case linear
    case easeOut
    case easeIn
}

struct KeyFrame {
    let offset: TimeInterval
    let rotation: Double
    let yScale: Double
    let y: CG
    let anima
```

```
        var animation: Animation? {
            switch animationKind {
            case .none: return nil
            case .linear: return .linear(duration: offset)
            case .easeIn: return .easeIn(duration: offset)
            case .easeOut: return .easeOut(duration: offset)
            }
        }
    }

    let keyframes = [
        // Initial state, will be used once. Its offset is useless and will be ignored
        KeyFrame(offset: 0.0, rotation: 0, yScale: 1.0, y: 0, animationKind: .none),

        // Animation keyframes
        KeyFrame(offset: 0.2, rotation:   0, yScale: 0.5, y:  20, animationKind: .linear),
        KeyFrame(offset: 0.4, rotation:   0, yScale: 1.0, y: -20, animationKind: .linear),
        KeyFrame(offset: 0.5, rotation: 360, yScale: 1.0, y: -80, animationKind: .easeOut),
        KeyFrame(offset: 0.4, rotation: 360, yScale: 1.0, y: -20, animationKind: .easeIn),
        KeyFrame(offset: 0.2, rotation: 360, yScale: 0.5, y:  20, animationKind: .easeOut),
        KeyFrame(offset: 0.4, rotation: 360, yScale: 1.0, y: -20, animationKind: .linear),
        KeyFrame(offset: 0.5, rotation:   0, yScale: 1.0, y: -80, animationKind: .easeOut),
        KeyFrame(offset: 0.4, rotation:   0, yScale: 1.0, y: -20, animationKind: .easeIn),
    ]
```

If you wonder why I didn't do it like this in the first place, I just wanted to show you that both ways are possible. The first case is more flexible, but more verbose. That is, we are forced to specify the duration for each animation, however, it is more flexible, because we are free to use a duration that does not match the offset.

When using this new approach, however, you could easily add a customizable factor, that could let you slow down or speed up the animation, without having to touch the keyframes at all.

Nesting TimelineViews: Nothing prevents you from nesting one `TimelineView` inside another. Now that we have our JumpingEmoji, we can put three `JumpingEmoji` views inside a `TimelineView` that makes them appeared one at a time with a delay:



For the full code of the emoji wave, check out this gist.

GifImage

I originally had one more example, but it got scrapped when I published the article. The reason it didn't made the cut, is because the concurrency API was not yet stable. Fortunately, it is now safe to publish it. The code uses TimelineView to implement a view for animated gifs. The view loads the gif asynchronously from a URL (which can be both local or remote). All the code is available in this gist.

Summary

Congratulations for reaching the end of this long post. It was a ride! We went from the simplest `TimelineView` example, to some creative uses of the view. In part 5, I will explore the new *Canvas* view, and how well it combines with `TimelineView`. By putting them both together, we will extend even more what is possible in the world of SwiftUI animations.

Feel free to follow me on twitter, if you want to be notified when new articles are published. Until then!

Animations, SwiftUI

< Random Lessons from the SwiftUI Digital Lounge

> Advanced SwiftUI Animations – Part 5: Canvas

6 thoughts on "Advanced SwiftUI Animations — Part 4: TimelineView"

Even Northug

June 29, 2021 at 1:21 pm

Thank you for an interesting and well written article. I'm making a notational sequencer: piano roll with musical events along a timeline, having a hard time with scrolling, zooming, snapping, and synching it all. It's about time to put some structural information into the old piano roll. Anyway … Perhaps Timeline can come in handy, in addition to Timer.

Reply

Helge Heß

July 3

Note that collections feature `randomElement`, e.g. instead of the boilerplate'y:

```
ManyFaces.emoji[Int.random(in: 0..<ManyFaces.emoji.count)]
```

you can do

```
Self.emoji.randomElement()! // or ?? ""
```

(it returns nil for empty collections)

https://developer.apple.com/documentation/swift/array/2994747-randomelement

Reply

javier

July 6, 2021 at 6:30 am

Brilliant! Updated already 😉

Reply

Helge

July 6, 2021 at 10:26 am

The `Self` doesn't work in Subview though, you need a `ManyFaces` there (I often just use `fileprivate`'s for such shared statics)

Reply

javier

July 6, 2021 at 12:29 pm

Oops! I fell victim of a blind search and replace. 😉 I changed it back to ManyFaces, in both places. I made a whole point of both pieces of code being identical after all 😉 Thanks again!

Reply

nylki

September 8, 2021 at 2:17 pm

Thank you for the article, appreciate your thorough and illustrated exploration of the TimelineView!

Reply

Leave a Comment

Name *

Email *

Website

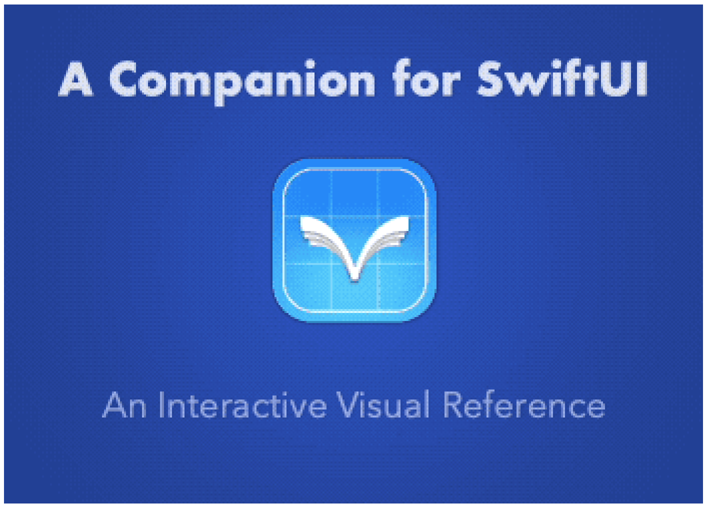
I'm not a robot

reCAPTCHA
Privacy - Terms

Replies to my comments Notify me of followup comments via e-mail. You can also subscribe without commenting.

Post Comment

Search ...



Categories

Animations
Drawing
Layout

SwiftUI

User Interface

Archives

September 2021

June 2021

November 2020

September 2020

July 2020

June 2020

March 2020

February 2020

December 2019

November 2019

October 2019

September 2019

August 2019

July 2019

June 2019

javier@swiftui-lab.com

@SwiftUILab