Chapter 10 Transition and Transform

1. Introduction

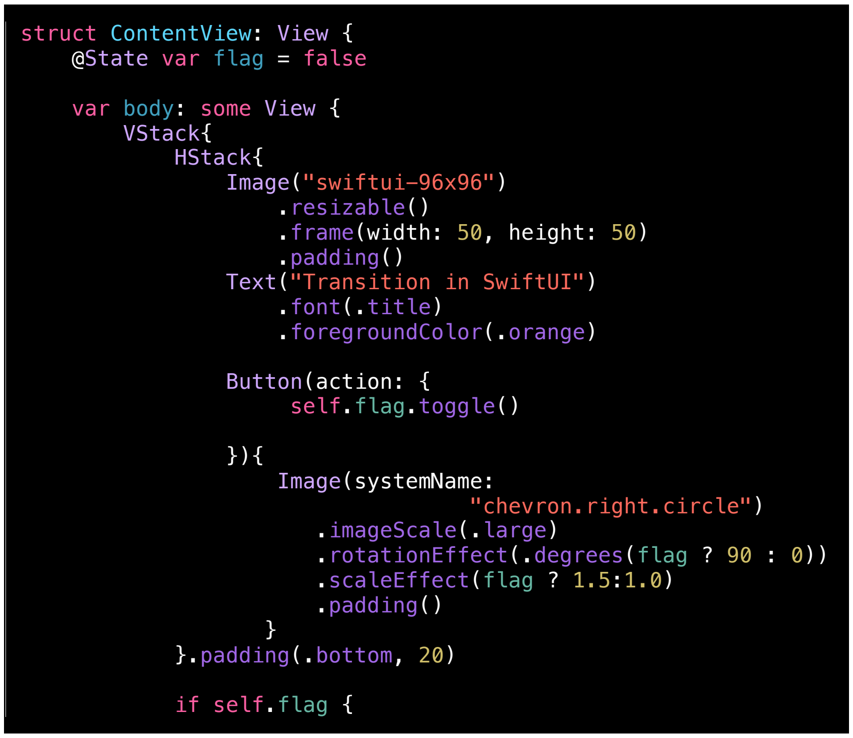
To be added later

Code for this chapter is available on GitHub:

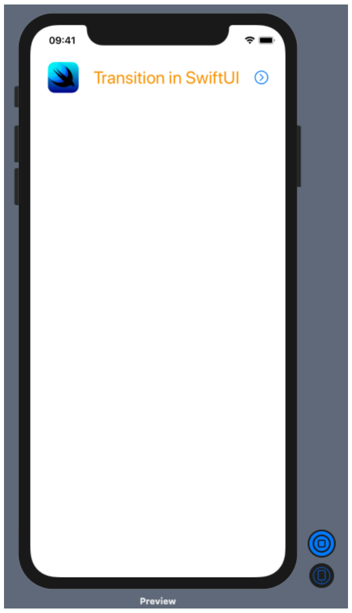
http://www.github.com/mhuq1138/swiftui-chapter-10-transition-and-transform

2. Simple View Transition

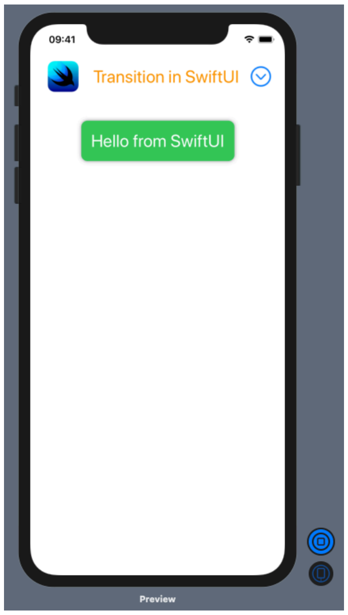
A transition in SwiftUI is what determines how a view is inserted or deleted from the hierarchy. If we don't use transition, when we insert a view in a hierarchy, it appears suddenly. When we delete the view it disappears suddenly. This is illustrated by the following code:



```
struct ContentView: View {
    @State var flag = false

    var body: some View {
        VStack{
            HStack{
                Image("swiftui-96x96")
                    .resizable()
                    .frame(width: 50, height: 50)
                    .padding()
                Text("Transition in SwiftUI")
                    .font(.title)
                    .foregroundColor(.orange)

                Button(action: {
                     self.flag.toggle()

                }){
                    Image(systemName:
                                   "chevron.right.circle")
                       .imageScale(.large)
                       .rotationEffect(.degrees(flag ? 90 : 0))
                       .scaleEffect(flag ? 1.5:1.0)
                       .padding()
                }
            }.padding(.bottom, 20)

            if self.flag {
```



```
                DetailView()
            }
            Spacer()
        }
    }
}
```



When we tap the button the Boolean `flag` is toggled. When flag is `true`, the `DetailView` appears in the hierarchy. Another tap will remove the `DetailView`. The appearance and disappearance of `DetailView` happens. We can change this behavior by adding a `.transition(_:)` modifier. A transition on its own has no effect, it must be associated with an animation.

The declaration of `.transition(_:)` modifier is

```
.transition(t: AnyTransition)
```

The struct `AnyTranstion` has type properties and type methods determining different types of transitions. There are 5 different types of transition:

1. Slide

   The view is inserted sliding in for the leading side and removed by sliding towards the trailing side. The type property `slide` is used.

2. Move

   The type method

   ```
   static func move(edge: Edge) -> AnyTransition
   ```

   is used. On insertion, the view slides in from the specified edge. On removal, it slides out of the same side.

3. Opacity

   The view fades in when it is inserted and fades out when it is removed. The type property `opacity` is used.

4. Scale

   The view is inserted with the specified scale and reverses the effect when removed . There are two ways we can use scale. Using the type property `scale` the view scales from zero to normal size when inserted and on removal from normal size to zero. We can use custom scale with the type method

```
static func scale(scale: CGFloat, anchor: UnitPoint = .center)
                                                -> AnyTransition
```

5. Offset

   This is like move transition. Now, we can set our x and y coordinates. The type method used is

```
static func offset(x: CGFloat = 0, y: CGFloat = 0) -> AnyTransition
```

   We can also use the type method:

```
static func offset(_ offset: CGSize) -> AnyTransition
```

A transition does nothing unless associated with an animation. We can use implicit animation by applying it to the view on which the transition modifier is applied, or to the parent of the view. Alternatively, we can use explicit animation on the state change that triggers the transition.

To demonstrate all different types of transition, we create a project “Simple Transition Demo”. Each type will be a row in a list embedded in a navigation view. We are going to use the same code described at the beginning of the section. Instead of repeating the entire code, we are going to display only the relevant code for each transition type.

For the slide transition, we use the struct `SlideTransitionView`. The relevant part of the code is:

```
.
.
.
Button(action: {
    self.flag.toggle()

})
.
.
.
if self.flag {
    DetailView()
        .transition(.slide)
        .animation(.easeIn(duration: 2))
}
```

In live preview, we find that the transition happens exactly the way expected: On insertion, the view slides in from the leading edge with `easeIn` animation of 2 seconds duration. On removal, the slides out towards the trailing edge with the same animation.

Now, let us see what happens when we use explicit animation instead:

```
.
```

```
.
.
Button(action: {
    withAnimation(.easeIn(duration: 2)){
         self.flag.toggle()
    }
})
.
.
.
if self.flag {
    DetailView()
        .transition(.slide)
}
```

Now there is no animation when the view is inserted, but on removal the view slides towards the trailing edge with animation. Towards the end of the slide, the view abruptly disappears.

Warning!

Presumably, this is a bug in the Xcode 11.3.1 version. We have to wait and see what Apple does.

For the move transition, we use the struct `MoveTransitionView`. It behaves just like the slide transition.

```
.
.
.
Button(action: {
    self.flag.toggle()

})
.
.
.
if self.flag {
    DetailView()
        .transition(.move(edge: .bottom))
        .animation(.easeIn(duration: 2))
}
```

For the offset transition, we use the struct `OffsetTransitionView`. It behaves just like the slide transition.

```
.
.
.
Button(action: {
    self.flag.toggle()

})
.
.
.
if self.flag {
    DetailView()
        .transition(.offset(x: 200, y: 300))
        .animation(.easeIn(duration: 2))
}
```

Now, when the view is inserted it starts from the offset point and when remove it ends at the offset point.

For the offset transition, we use the struct `OpacityTransitionView`. It works only with explicit animation

```
.
.
.
Button(action: {
    withAnimation(.easeIn(duration: 2)){
         self.flag.toggle()
    }
})
.
.
.
if self.flag {
    DetailView()
        .transition(.opacity)
}
```

Next, for scale transition with `scale` property, we use the struct `ScaleTransitionView`. It works only with explicit animation

```
.
.
.
Button(action: {
    withAnimation(.easeIn(duration: 2)){
         self.flag.toggle()
    }
})
.
.
.
if self.flag {
    DetailView()
        .transition.scale)
}
```

Finally, for scale transition with `scale` method, we use the struct Custom`ScaleTransitionView`. It works only with explicit animation

```
.
.
.
Button(action: {
    withAnimation(.easeIn(duration: 2)){
         self.flag.toggle()
    }
})
.
```

```
.
.
if self.flag {
    DetailView()
        .transition.scale(scale: 1.5, anchor: .bottomLeading))
}
```

3. Combining Transition

SwiftUI provides a `.combined(with: )` method that allows us to combine two or more animations. We demonstrate in a project "Combining Transition Demo".

> **Warning!**
>
> Because of the bug some of the combinations may not work as expected.

We start with an example of combining scale transition with opacity transition:

```
.
.
.
Button(action: {
    withAnimation(.easeIn(duration: 2)){
         self.flag.toggle()
    }
})
.
.
.
if self.flag {
    DetailView()
        .transition(AnyTransition.scale(scale: 1.5).combined(with: .opacity))
}
```

This works as expected because both scale and opacity transition work with explicit animation.

Next we try slide with scaling:

```
.
.
.
Button(action: {
    withAnimation(.easeIn(duration: 2)){
         self.flag.toggle()
    }
})
.
.
.
if self.flag {
    DetailView()
        .transition(AnyTransition.scale(scale: 1.5).combined(with: .slide))
}
```

This also works as expected.

4. Asymmetric Transition

By default, transitions apply in one way when the view is inserted in the hierarchy. When the view is removed, it will produce the opposite effect. We can change this by using asymmetric transition:

```
static func asymmetric(insertion: AnyTransition, removal: AnyTransition) -> AnyTransition
```

This returns a transition that uses a different transition for insertion versus removal.

To demonstrate we create a project “Asymmetric Transition Demo”. As our first example, we create an asymmetric transition that uses opacity when the view is inserted and scale when it is removed:

```
.
.
.
Button(action: {
    withAnimation(.easeIn(duration: 2)){
         self.flag.toggle()
    }
})
.
.
.
if self.flag {
    DetailView()
        .transition(.asymmetric(insertion: .opacity, removal: .scale(scale: 1.5)))
}
```

As our second example, we use slide on insertion and move on removal:

```
.
.
.
Button(action: {
    self.flag.toggle()
}
})
.
.
.
if self.flag {
    DetailView()
        .transition(.asymmetric(insertion: .slide, removal: .move(edge: .bottom)))
        .animation(.easeIn(duration: 2))
}
```

It is interesting to note that both implicit and explicit animation work in this case.

As our final example, we will consider asymmetric and combine transition

```
.
.
.
Button(action: {
    withAnimation(.easeIn(duration: 2)){
         self.flag.toggle()
    }
})
.
.
.
if self.flag {
    DetailView()
        .transition(.asymmetric(insertion: AnyTransition.slide.combined(with: .opacity),
                                                            removal: .move(edge: .bottom)))
}
```

5. Custom Transition

Under the hood, a transition needs a modifier for the beginning and the end of the animation. SwiftUI will figure out the rest, provided the difference between both modifiers is animatable. `AnyTransition` provides us a type method for this purpose:

```
static func modifier<E>(active: E, identity: E) -> AnyTransition where E : ViewModifier
```

To demonstrate, we create a project “Custom Transition Demo”. As our first example, we create a custom opacity transition:

```
struct MyOpacityModifier: ViewModifier {
    let opacity: Double

    func body(content: Content) -> some View {
        content.opacity(opacity)
    }
}

extension AnyTransition{
    static var myOpacity: AnyTransition{
        get{
            AnyTransition.modifier(
                active: MyOpacityModifier(opacity: 0),
                identity: MyOpacityModifier(opacity: 1))
        }
    }
}
```

We then make use of it in

```swift
.
.
.
Button(action: {
    withAnimation(.easeIn(duration: 2)){
         self.flag.toggle()
    }
})
.
.
.
if self.flag {
    DetailView()
        .transition(.myOpacity)
}
```

In the next example, we create a custom transition with a type method:

```swift
struct MyScaleModifier: ViewModifier{
    let scale:CGFloat

    func body(content:Content)-> some View {
        content.scaleEffect(scale)
    }
}


extension AnyTransition{

    static func myScale(scale:CGFloat)->AnyTransition{
        return AnyTransition.modifier(
            active:MyScaleModifier(scale: 1),
            identity: MyScaleModifier(scale: scale))
    }
}
```

Finally, we make use of GeometryEffect to create custom animatable data

```swift
extension AnyTransition {
    static var rotate: AnyTransition { get {
        AnyTransition.modifier(active: RotateTransition(pct: 0),
                               identity: RotateTransition(pct: 1))
        }
    }
}

struct RotateTransition: GeometryEffect {
    var pct:Double

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

    func effectValue(size: CGSize) -> ProjectionTransform {
        let rotationPercent = pct
        let a = CGFloat(Angle(degrees: 90 * (1 - rotationPercent)).radians)

        var transform3d = CATransform3DIdentity
        transform3d.m34 = -1/max(size.width, size.height)
        transform3d = CATransform3DRotate(transform3d, a, 0, 1, 0)
        transform3d = CATransform3DTranslate(transform3d, -size.width/2.0, 0, 0)
        let affineTransform1 = ProjectionTransform(CGAffineTransform(translationX:
                                                  size.width/2.0, y: size.height / 2.0))

        return ProjectionTransform(transform3d).concatenating(affineTransform1)
    }
}
```

This creates a transition by rotating the view about the y axis as it is inserted and removed.

6. CGAffineTransform

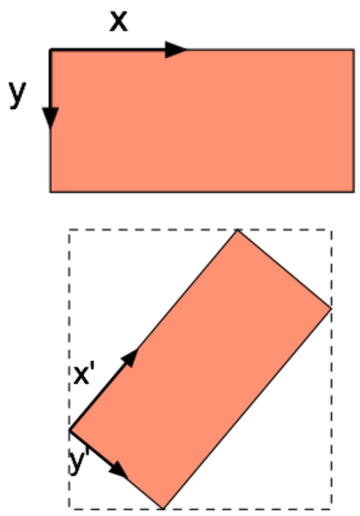
`CGAffineTransform` is a struct provided by Core Graphics that provides an affine transform matrix for use in drawing 2D graphics.

An affine transform is a 3 x 3 matrix:

$$\begin{pmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{pmatrix}$$

We use this matrix to relate the transformed and original coordinates by

$$x' = ax + cy + t_x$$
$$y' = bx + dy + t_y$$



The elements a, b, c and d are for rotation, scaling, and shear transforms. The elements t_x and t_y are for translations. For rotation, the matrix has the form

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Where θ is the angle of rotation.

For scaling,

$$\begin{pmatrix} a & 0 & 0 \\ 0 & d & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

While for shear

$$\begin{pmatrix} 1 & b & 0 \\ c & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

`CGAffineTransform` has an initializer that takes all six matrix elements as parameters:

```
init(a: CGFloat, b: CGFloat, c: CGFloat, d: CGFloat, tx: CGFloat, ty: CGFloat)
```

It also has three convenience initializers that are useful for some common transform types:

For rotation, we use:

```
init(rotationAngle angle: CGFloat)
```

The angle is in radians.

For translation, we use:

```
init(translationX tx: CGFloat,y ty: CGFloat)
```

The parameters `tx` and `ty` determine amount of translation in the x and y direction.

For scaling we use:

```
init(scaleX sx: CGFloat, y sy: CGFloat)
```

We also have the following instance methods to change affine transform.

The following method is used to return an affine transform by combining two affine transforms:

```
func concatenating(_ t2: CGAffineTransform) -> CGAffineTransform
```

We can rotate the existing transform to a new transform using:

```
func rotated(by angle: CGFloat) -> CGAffineTransform
```

We can translate the existing transform to a new transform using:

```
func translatedBy(x tx: CGFloat, y ty: CGFloat) -> CGAffineTransform
```

We can translate the existing transform to a new transform using:

```
func scaledBy(x sx: CGFloat, y sy: CGFloat) -> CGAffineTransform
```

We can invert an existing transform using:

```
func inverted() -> CGAffineTransform
```

7. CATransform3D

We get `CATransform3D` from Core Animation. This is the standard transformation matrix used throughout Core Animation. Note that transforms in 2D can be treated as subset of 3D transforms.

`CATransform3D` matrix is a 4 x 4 matrix with sixteen elements denoted by `m11`, `m12`, etc. It has an initializer with all sixteen elements as parameters:

```
init(m11: Float, m12: Float, m13: Float, m14: Float, m21: Float, m22: Float, m23: Float,
     m24: Float, m31: Float, m32: Float, m33: Float, m34: Float, m41: Float, m42: Float,
     m43: Float, m44: Float)
```

`CATransform3D` struct has 16 instance properties for the 16 elements.

8. View Transform in SwiftUI

For view transform in 2D there are two view modifiers we can use:

For rotation:

```
func rotationEffect(_ angle: Angle, anchor: UnitPoint = .center) -> some View
```

For all affine transform:

```
func transformEffect(_ transform: CGAffineTransform) -> some View
```

We demonstrate use of these view modifiers in a project “2D View Transform Demo”. As our first example, we create a view that explores all range of values for the affine transform using the code:

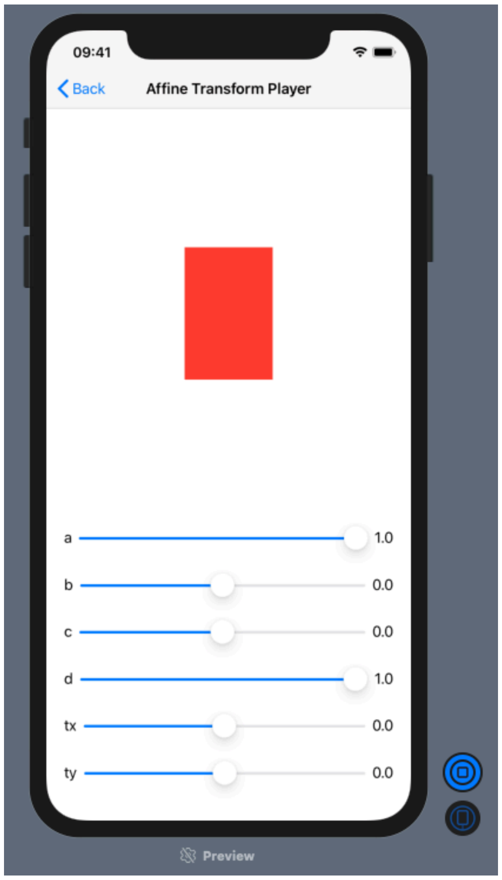
```
struct AffineTransformPlayer: View {
    @State private var a:Double = 1.0
    @State private var b:Double = 0.0
    @State private var c:Double = 0.0
    @State private var d:Double = 1.0
    @State private var tx:Double = 0.0
    @State private var ty:Double = 0.0


    var body: some View {
        VStack{
            Spacer()
            Color.red
               .frame(width: 100, height: 150)
```

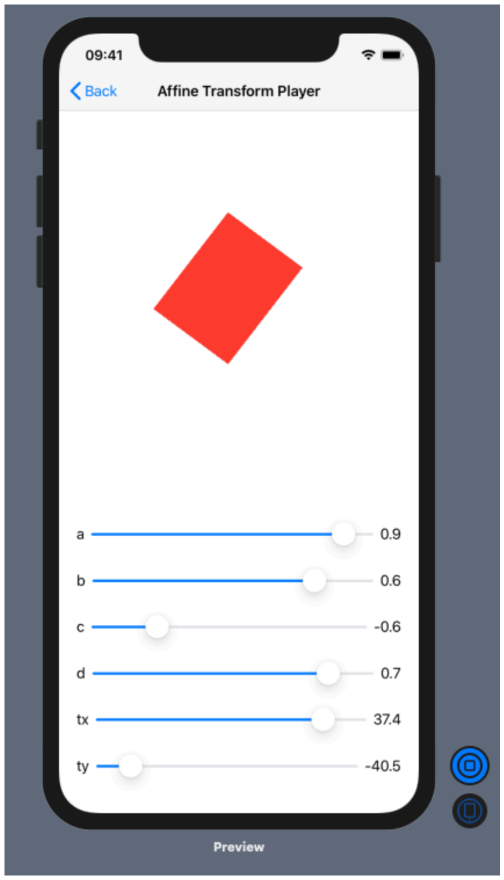
```
                .transformEffect(CGAffineTransform(a: CGFloat(a), b: CGFloat(b), c:
                           CGFloat(c), d: CGFloat(d), tx: CGFloat(tx), ty: CGFloat(ty)))
             Spacer()
             SliderDisplay(sliderValue: $a, label: "a", lower: -1.0, upper: 1.0)
             SliderDisplay(sliderValue: $b, label: "b", lower: -1.0, upper: 1.0)
             SliderDisplay(sliderValue: $c, label: "c", lower: -1.0, upper: 1.0)
             SliderDisplay(sliderValue: $d, label: "d", lower: -1.0, upper: 1.0)
             SliderDisplay(sliderValue: $tx, label: "tx", lower: -50.0, upper: 50.0)
             SliderDisplay(sliderValue: $ty, label: "ty", lower: -50.0, upper: 50.0)
         }.navigationBarTitle("Affine Transform Player", displayMode: .automatic)
     }
}
```

The slider display is produced by the code

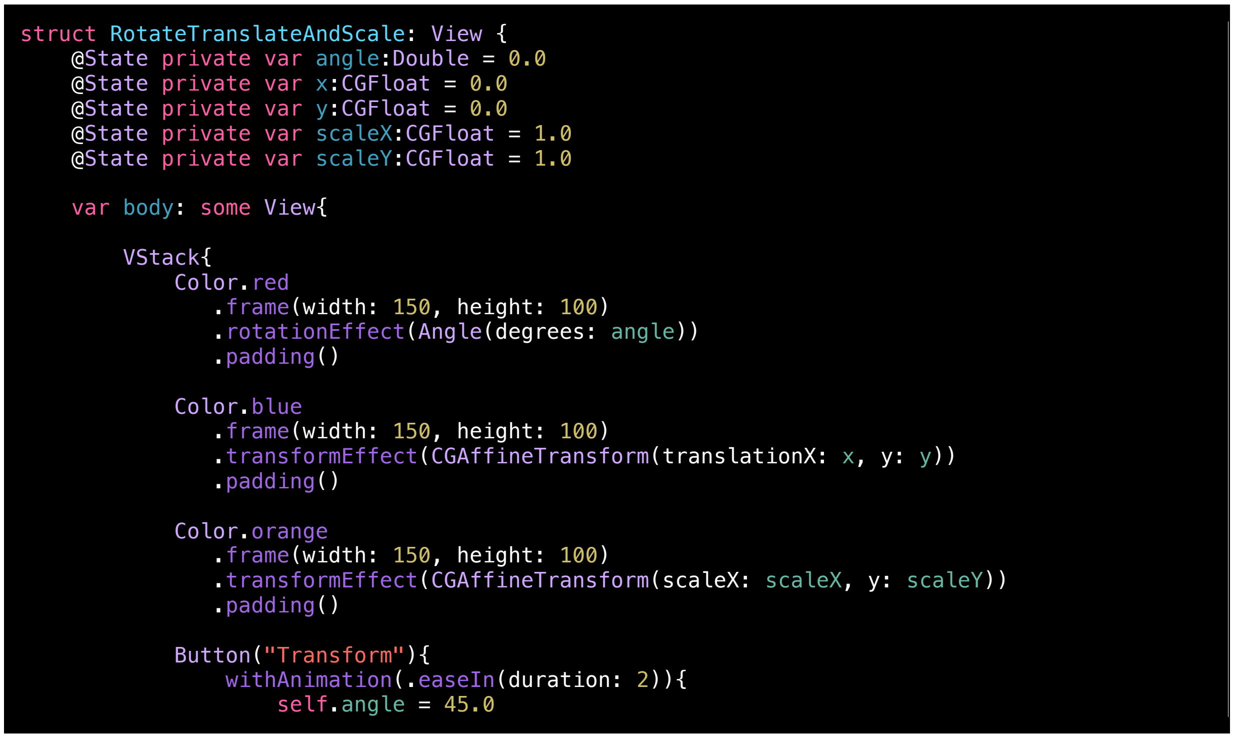
```
struct SliderDisplay: View {
    @Binding var sliderValue:Double
    let label:String
    let lower:Double
    let upper:Double

    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 live preview:





Next we examine specific transforms with



```
struct RotateTranslateAndScale: View {
    @State private var angle:Double = 0.0
    @State private var x:CGFloat = 0.0
    @State private var y:CGFloat = 0.0
    @State private var scaleX:CGFloat = 1.0
    @State private var scaleY:CGFloat = 1.0

    var body: some View{

        VStack{
            Color.red
               .frame(width: 150, height: 100)
               .rotationEffect(Angle(degrees: angle))
               .padding()

            Color.blue
               .frame(width: 150, height: 100)
               .transformEffect(CGAffineTransform(translationX: x, y: y))
               .padding()

            Color.orange
               .frame(width: 150, height: 100)
               .transformEffect(CGAffineTransform(scaleX: scaleX, y: scaleY))
               .padding()

            Button("Transform"){
                withAnimation(.easeIn(duration: 2)){
                    self.angle = 45.0
```

```
                     self.x = -100
                     self.y = 30
                     self.scaleY = 1.2
                     self.scaleY = 0.5
                 }
             }
         }.navigationBarTitle("Rotate Translate and Scale", displayMode: .automatic)
     }
 }
```

When we run the code we find that only the rotation transform takes place with animation.

To animate the other transforms, we have to create custom modifiers with GeometryEffect. For example for translation:

```
struct CustomTranslateView: View {
    @State var x:CGFloat = 0.0
    @State var y:CGFloat = 0.0
    var body: some View {
        VStack{
            Rectangle()
               .fill(Color.red)
               .frame(width: 200, height: 150)
               .modifier(translationEffect(x: x, y: y))
               .animation(.easeIn(duration: 2))

            Button("Animate"){
                self.x = 45
                self.y = -30
            }
        }.navigationBarTitle("Custom Translate Effect", displayMode: .automatic)
    }
}

struct transEffect: GeometryEffect  {
    var x:CGFloat
    var y:CGFloat

    var animatableData: AnimatablePair<CGFloat,CGFloat>{
        get { AnimatablePair(x, y)}
        set {
            x = newValue.first
            y = newValue.second
        }
    }
    func effectValue(size: CGSize) -> ProjectionTransform {
        let affineTransform = CGAffineTransform(translationX: x, y: y)
        return ProjectionTransform(affineTransform)
    }
}

func translationEffect(x:CGFloat,y:CGFloat)-> some GeometryEffect{
    return transEffect(x: x, y: y)
}
```

3D transform to be added later