

12-USER INPUT animation

User Interaction:

- Bisogna captare le iterazioni degli utenti

Quando c'è un tocco, **ottengo un MotionEvent** contenente le informazioni che mi forniscono informazioni su cosa è avvenuto (è uno stream, devo considerarlo come una parte di una sequenza).

MotionEvent, inizialmente aveva field Action, successivamente 2 field:

- **ActionMasked:** azione

The **actionMasked** field can contain several values

- **ACTION_DOWN:** a new gesture has started (the first finger has made contact with the screen)
- **ACTION_POINTER_DOWN:** an additional finger was added
- **ACTION_MOVE:** one finger moved, staying in contact with the screen: there can be many such events in a single object
- **ACTION_POINTER_UP:** one finger has moved off the screen (but others are still in contact)
- **ACTION_UP:** the last finger has detached from the screen: the gesture is finished
- **ACTION_CANCEL:** the gesture has been programmatically cancelled

→ notifica di ignorare

- **ActionIndex:** indice

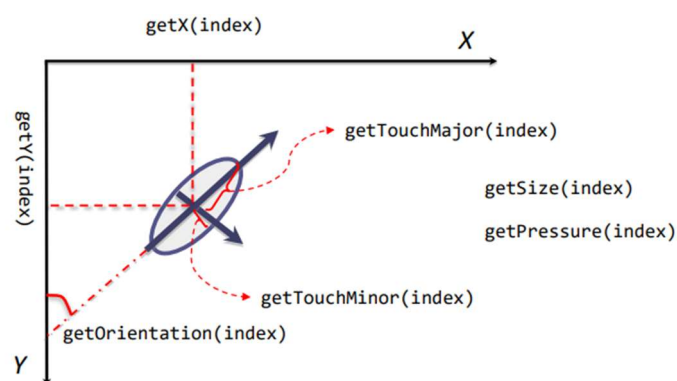
Un **singolo tocco** è modellato dal sensore in una superficie ellittica, si assume il tocco al centro dell'ellisse

- **x, y:** centro dell'ellisse
- **touchMinor, touchMajor :** lunghezza dei mezzi assi dell'ellisse
- **size:** contiene area dell'ellisse
- **pressure:** stima della pressione applicata
- **orientation:** angolo tra l'asse maggiore dell'ellisse e l'asse -Y: quando vale 0, l'asse maggiore è verticale
 - se positivo, è inclinato verso destra; se negativo, a sinistra

Tocchi multipli: partendo dal primo tocco, il MotionEvent riporta le informazioni circa tutte le dita a contatto con lo schermo

- nr tocchi in **pointerCount**
- ogni tocco è associato ad un id che è costante per la durata del tocco
 - accedo ad un determinato tocco con **getPointerId(index)**

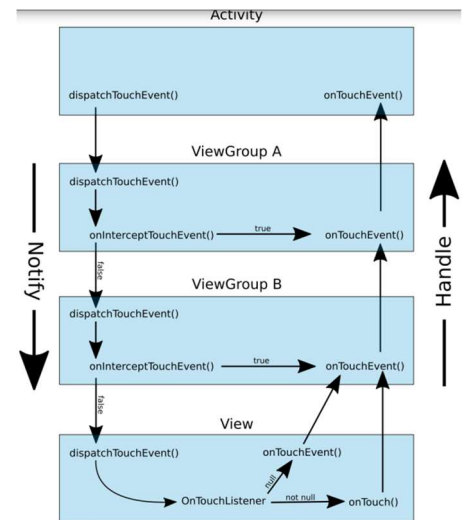
MOTION EVENT:



INPUT PRECEDENTI

Stream di dati sul touch sono bufferizzati:

- Il dispositivo elettronico può operare ad un'elevata frequenza ma non vogliamo che la nostra applicazione senta con quella frequenza
 - Il dispositivo comunica le informazioni con una determinata frequenza in blocchi di informazione
 - Contiene l'azione attuale e un gruppo di azioni precedenti
 - `getHistoricalX()`
 - `getHistoricalY()`
 - `getHistoricalOrientation()`
 - ...
 - Nella nostra applicazione abbiamo bisogno di un oggetto che mantenga questi dati per svolgere determinate azioni
 - Android fornisce un **gestureDetector** per la `viewCompose` e per `JetpackCompose`
- Lo stream di dati inizia con `Activity`
 - `Activity` è notificata con **`DispatchTouchEvent`**
 - Ha la possibilità di comportarsi come filtro e scartare cose in modo tale da non propagare (es. quando un `ModelView` appare sopra un altro (popup) e voglio sentire solo click su di lui) → posso scartare tocchi su altre parti.



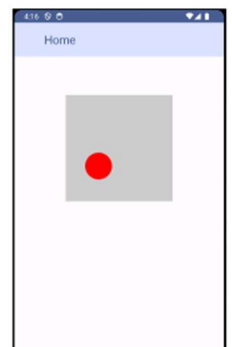
Come implementare touch in una `JetpackCompose` application?

- Cosa più semplice → non usarla e usare solo bottoni
- Usare modifier
 - `Modifier.clickable`** contenente un parametro che è una callback che esegue determinate cose quando avviene touch
 - `Modifier.draggable`** specificando orientation e state
 - Detect quando un componente viene mosso lungo uno dei due assi
 - `Modifier.pointerInput`**
 - Mi permette di captare una iterazione generale, prendendomi la responsabilità di gestire la complessità
 - `detectTapGesture` → mi permette di detect iterazioni

--codice animation

```
@Composable
fun MyTouchableBox(modifier = Modifier.fillMaxSize()) {
    val ctx = LocalContext.current
    val msg by remember { mutableStateOf("") }
    Box(modifier) {
        Text(msg,
            style = MaterialTheme.typography.h3,
            modifier = Modifier.align(Alignment.Center)
                .background(Color.Cyan).size(300.dp)
                .pointerInput(Unit) {
                    detectTapGestures {
                        onPress = { msg = "Press" },
                        onLongPress = { msg = "Long Press" },
                        onTap = { msg = "Tap" },
                        onDoubleTap = { msg = "Double Tap" }
                    }
                }
    )
    }
```

```
@Composable
fun MyDraggableBox(modifier = Modifier.size(200.dp)) {
    Box(modifier) {
        var offsetX by remember { mutableStateOf(0f) }
        var offsetY by remember { mutableStateOf(0f) }
        Box(
            Modifier
                .offset { IntOffset( offsetX.roundToInt(),
                                     offsetY.roundToInt() ) }
                .background(Color.Red, CircleShape)
                .size(50.dp)
                .pointerInput(Unit) {
                    detectDragGestures { change, dragAmount ->
                        change.consume()
                        offsetX += dragAmount.x
                        offsetY += dragAmount.y
                    }
                }
        )
    }
}
```



Di default, quando usiamo `compose`, se non usiamo uno specifico component come `lazyGrid` o `lazyScroll`, componenti non sono scrollabili; per renderli scrollabili dobbiamo usare **`verticalScroll`** o **`horizontalScroll`** che devono ricordarsi la posizione in cui ci trovavamo attraverso **`rememberScrollState`**

```
Box(Modifier.verticalScroll(rememberScrollState()) ) {
    //Content to be scrolled...
}
```

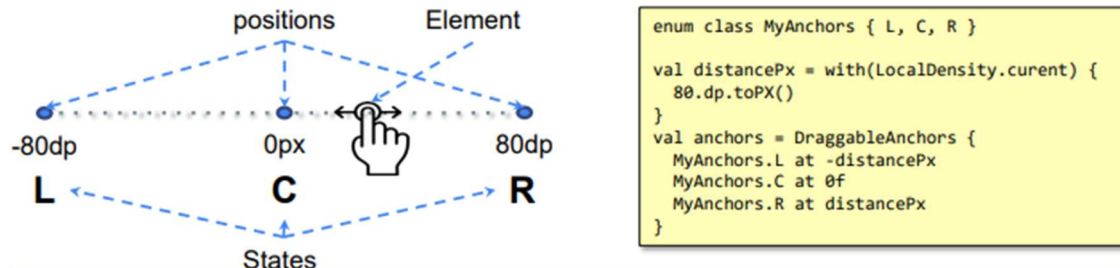
Nota su gesture on scrolling:

- compose supporta due tipi di gesture basati su scrolling:
 - **scroll**: per esplorare contenuto la cui dimensione è maggiore della superficie del display in cui è contenuto
 - **swipe**: per triggerare azioni non reversibili (Es: eliminare item da una lista)

SWIPE

C'è una serie di componenti che devono essere usati, si possono usare una serie di posizioni in cui può essere un componente:

- elemento potrà essere draggable in queste posizioni



Si può usare uno speciale draggable: **AnchoredDraggableState**

```
val state = remember {
    AnchoredDraggableState<MyAnchors>(
        MyAnchors.C, // initial position
        anchors, // set of possible positions
        positionalThreshold = { distance -> distance*0.5f }, // minimum amount of movement
        velocityThreshold = { with (LocalDensity.current) { 100.dp.toPx() } }, // speed in dp/s
        animationSpec = tween<Float>()
    )
}
```

si possono settare delle soglie di velocità e posizione in modo tale da poter gestire meglio i vari casi.

Dragging and swiping

- The dragged component must be decorated with a couple of modifiers
 - One assigning an offset to its natural position, which is derived from the current state (**.offset(...)**)
 - An other specifying that it is draggable (**.anchoredDraggable(...)**), defining its state as well as its direction

```
Box {
    Icon(Icons.filled.Delete, "delete", Modifier.align(Alignment.CenterStart).size(80.dp).padding(8.dp))
    Icon(Icons.filled.Settings, "settings", Modifier.align(Alignment.CenterEnd).size(80.dp).padding(8.dp))
    Box(Modifier
        .fillMaxWidth().height(80.dp)
        .offset(IntOffset(state.requireOffset().roundToInt(), 0))
        .background(Color.LightGray)
        .anchoredDraggable(state, Orientation.Horizontal)
    )
}
```

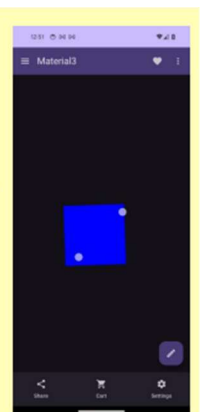
→ Animazione sul pptx

Multi-finger gesture: JetpackCompose supporta questo tipo di interazione (pinch) fornendo, tra i vari rilevatori in PointerInputScope, il metodo `detectTransformGestures(...)`

- L'utilizzo di questo metodo richiede che, nella catena di modificatori, il modificatore `graphicsLayer(...)` sia presente in una posizione precedente

Il metodo consente la manipolazione simultanea della rotazione, del ridimensionamento e della traslazione del componente a cui viene applicato, creando le condizioni per il comportamento pinch-to-zoom

```
@Composable
fun MultiTouchBox(modifier: Modifier = Modifier.fillMaxSize()) {
    Box(modifier) {
        var offset by remember { mutableStateOf(Offset.Zero) }
        var rot by remember { mutableStateOf(0f) }
        var scale by remember { mutableStateOf(1f) }
        Box(
            Modifier
                .graphicsLayer(scaleX = scale, scaleY = scale, rotationZ = rot,
                    translationX = offset.x, translationY = offset.y)
                .background(Color.Blue)
                .size(100.dp)
                .pointerInput(Unit) {
                    detectTransformGestures { _, pan, z, r ->
                        offset += pan; scale *= z; rot += r;
                    }
                }
        )
    }
}
```



ANIMATION

Ci permettono di reagire alle iterazioni, guidare l'utente, evidenziare un cambiamento:

- Eliminazione elemento
- Blinking
- Mostrare relazione tra diversi elementi

Utili perché prendono attenzione dell'utente ma da usare moderatamente.

Nota:

- dove abbiamo focus visivo: notiamo tutto
- contorno: notiamo che qualcosa succede ma non distinguiamo colori e etc
 - veloci

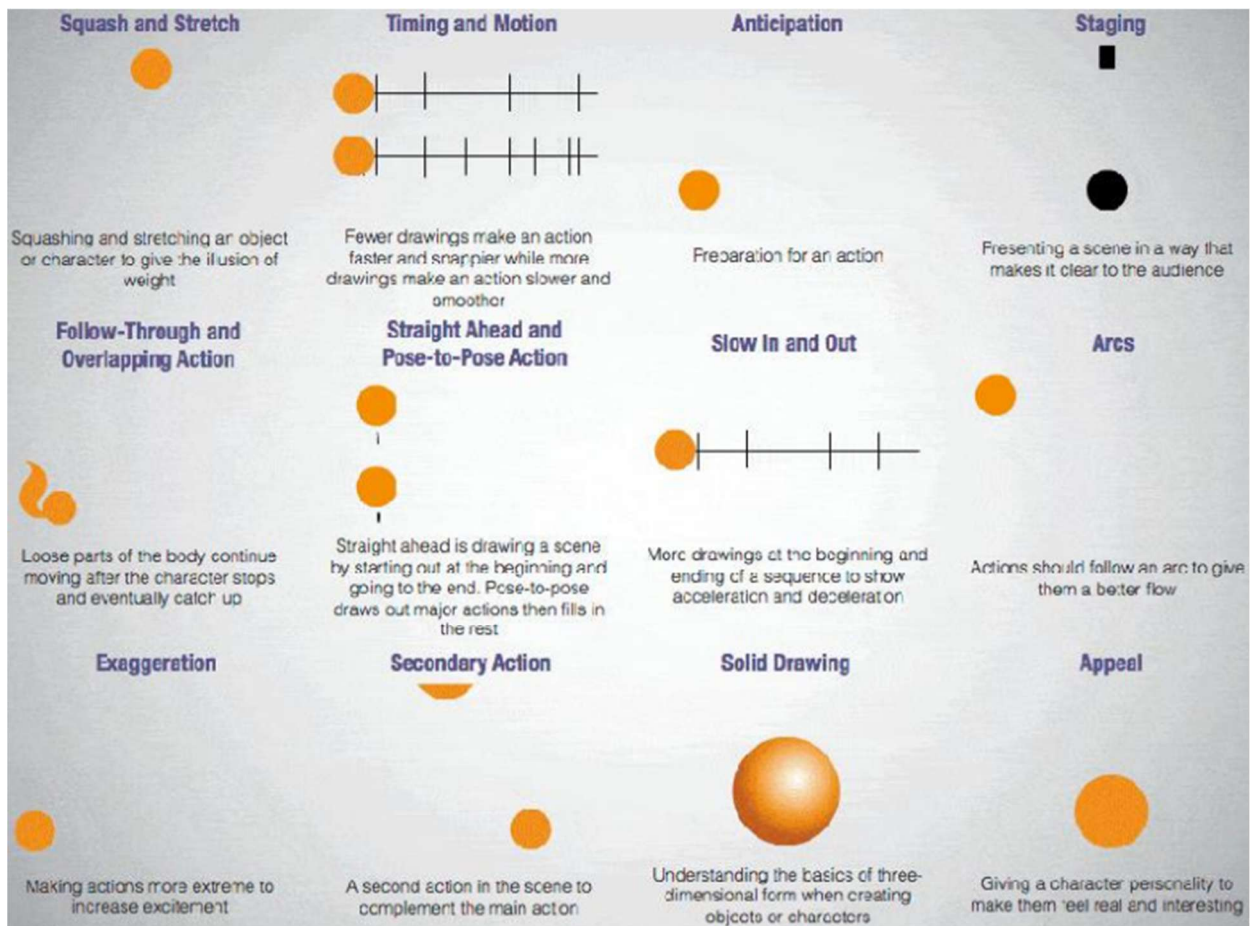
si effettua una **distinzione** tra:

- **Animazioni funzionali:** utilizzato per guidare e informare l'utente in tempo reale e può essere il risultato di un simulatore interattivo
- **Animazioni decorative:** supporto per storytelling → fornire informazioni all'utente in modo che mantenga l'attenzione

4 maggiori aree:

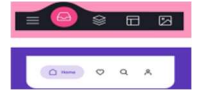
- micro-interaction
- loading and advancement
- navigation
- story-telling

12 principi di animation



vedere esempi animati su pptx

Squash and stretch: modo per fornire idea di movimento



Anticipation: fornisce un'anticipazione su cosa sta accadendo

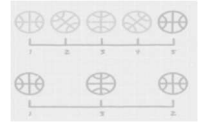
- es: riduco icona e mostro cosa si avrebbe (si può fare con mouse, non con touch)

Staging: Il movimento viene utilizzato per dirigere l'attenzione dello spettatore ed evidenziare gli elementi chiave della scena

- usato in navigation per mostrare il nuovo screen che entra nello schermo

Straight ahead:

- possono essere disegnate in diversi modi
- **pose-to-pose:** versione ora, cosa sta succedendo, versione finale
 - frame intermedi sono realizzati in modo automatico



Follow Through and Overlapping Action: quando le varie parti dell'oggetto non si muovono tutte allo stesso tempo, ma una va prima e altre seguono

Slow in and slow out: Come un oggetto richiede tempo per accelerare e decelerare mentre si muove

Arc: Le cose si muovono secondo il moto dell'arco, più velocemente qualcosa si muove, più piatto è l'arco e più ampia è la curva

Secondary Actions: complementare o evidenziare l'azione principale che si svolge in una scena

Timing: Descrive la velocità con cui qualcosa si muove e per quanto tempo rimane fermo

Exaggeration: ci sono alcune situazioni in cui si vuole operare sulla parte emotiva del cervello

- intrattenere persone e mostrare elementi

Solid drawing: per rappresentare alcune icone che sembrano tridimensionali

Appeal: si possono usare le animazioni per creare Appeal → rendere le mie cose interessanti

- raccontare una storia

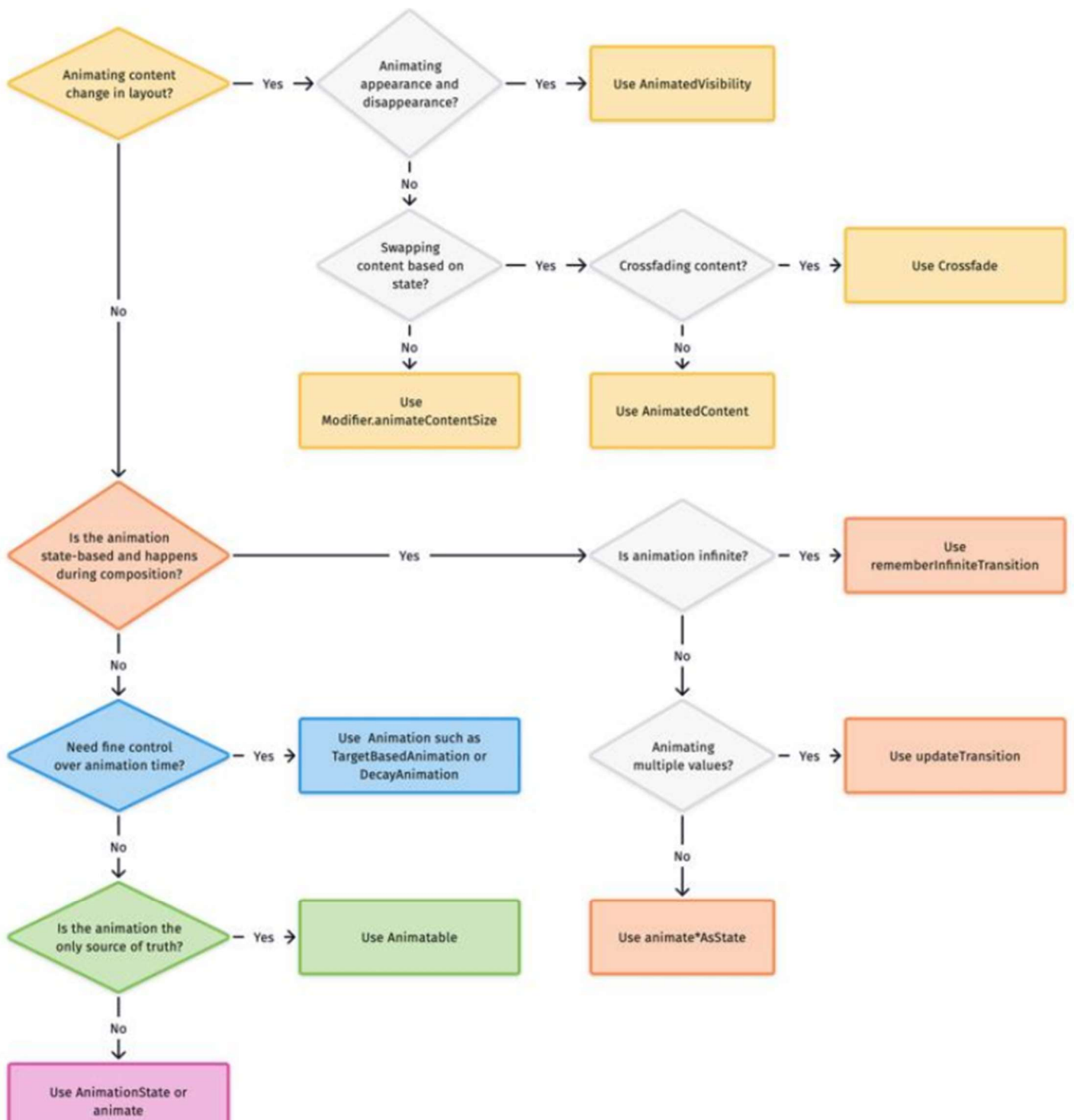
Come farlo?

Quando si vuole cambiare la gerarchia:

- `AnimatedVisibility`, `Crossfade`, `AnimatedContent`, `Modifier.animateContentSize`

Quando si vuole alterare qualcosa che sia già sullo schermo bisogna usare ricomposizione:

- `TargetBaseAnimation`, `DecayAnimation`, `Animatable`
- `rememberInfiniteTransition()`, `updateTransition()`, `animate*AsState()` `animate()`



Animated visibility


Elemento modulare che mostra/nasconde il contenuto in base ad un valore booleano

- permette di cambiare il comportamento dell'animazione per mostrare/nascondere il contenuto attraverso i parametri `enter:EnterTransition` e `exit:ExitTransition`
- all'interno della lambda funzione che racchiude il contenuto da mostrare, è possibile definire ulteriori transizioni che risultano collegate allo stato visibile
 - o questo permette di creare effetti di animazioni concatenate

```
var isVisible by remember { mutableStateOf(false) }
var rot by remember { mutableStateOf(0f) }

Column(
    modifier = Modifier.fillMaxWidth(),
    horizontalAlignment = Alignment.CenterHorizontally) {
    AnimatedVisibility( isVisible ) {
        val r by transition.animateFloat(label = "rotation") { }
        when (it) {
            EnterExitState.PreEnter -> 0f
            EnterExitState.Visible -> 180f
            EnterExitState.PostExit -> 0f
        }
    }
    rot = r

    Text("Lorem ipsum dolor sit amet...",
        modifier = Modifier.padding(8.dp))
} //...continues
```



Crossfade

```
var state by remember { mutableStateOf("a")}
Column(Modifier.fillMaxSize()) {
    Crossfade(
        targetState = state,
        animationSpec = tween(durationMillis = 1000),
        modifier = Modifier.padding(8.dp).fillMaxWidth()
    ) {
        state ->
        when (state) {
            "a" -> { ComponentOne() }
            "b" -> { ComponentTwo() }
        }
    }
    Spacer(modifier = Modifier.weight(1f))
    Button(onClick = { state = if (state == "a") "b" else "a" },
        Modifier.padding(top = 8.dp)) {
        Text("Change")
    }
}
```

AnimatedContent

È possibile definire transizioni specializzate attraverso elementi composabile `AnimatedContent(..)`.

- fornisce il parametro `transitionSpec` che è responsabile per definire quali transizioni devono essere usate per far comparire e scomparire componenti basandosi sulla coppia formata dallo stato iniziale e dallo stato finale del cambiamento
- il valore preso da questo parametro è il `composition` di 2 transizioni usando l'operatore infisso `with`

```
AnimatedContent(targetState = currentPage,
    transitionSpec = {
        ( transitionInAnimations with transitionOutAnimations )
    }
)
```

```
AnimatedContent(
    targetState = count,
    transitionSpec = {
        // Compare the incoming number with the previous number
        if (targetState > initialState) {
            // If the target is larger, slide up and fade in
            slideInVertically { height -> height } + fadeIn() with
            slideOutVertically { height -> -height } + fadeOut()
        } else {
            // If the target is smaller, slide down and fade in
            slideInVertically { height -> -height } + fadeIn() with
            slideOutVertically { height -> height } + fadeOut()
        }
    }.using(
        // Disable clipping
        SizeTransform(clip = false)
    )
)
} { targetCount ->
    Text(text = "$targetCount")
}
```

Animatable

La funzione Animatable ritorna un container che incapsula un singolo valore di un tipo generico T che può evolvere nel tempo

- Offre il metodo **animateTo(..)** che permette di animare il valore corrente verso quello specificato come destinazione, specificando il tipo di interpolazione da adottare e l'eventuale velocità iniziale
 - o This method is of the suspend fun type and can only be invoked in the context of a coroutine

```
val angle = remember { Animatable(0f) }
val color = remember { Animatable(Color.Green,
    typeConverter = Color.VectorConverter(ColorSpaces.LinearSrgb)) }
}
LaunchedEffect(angle, color) {
    launch { angle.animateTo(360f, animationSpec = tween(3000)) }
    launch { color.animateTo(Color.Blue, animationSpec = tween(3000)) }
}
Canvas(Modifier
    .padding(16.dp)
    .size(200.dp),
    onDraw = {
        rotate(angle.value) {
            drawRoundRect(color.value,
                cornerRadius = CornerRadius(16.dp.toPx()))
        }
    }
)
```

→ animazione su pptx

Animate *AsState

Insieme di funzioni componibili che permettono di animare un singolo valore dall'indicatore del valore da ottenere

- Numeri, colori, dimensioni
- La durata dell'animazioni e il suo progresso può essere controllata attraverso il parametro opzionale di tipo AnimationSpec
 - o Various animation types are supported: spring, tween, keyframes, repeatable, infiniteRepeatable, snap

```
var state by remember { mutableStateOf(false) }
val startColor = Color.Blue
val endColor = Color.Green

val backgroundColor by animateColorAsState(
    if (state) endColor else startColor
)

Column(
    modifier = Modifier.fillMaxSize()
        .background(backgroundColor),
    verticalArrangement = Arrangement.Center
) {
    Button (onClick = { state = !state }) {
        Text("Change color")
    }
}
```



AnimationSpec

Interfaccia che rappresenta i parametri relativi all'animazione:

- Tipo del dato animato
- Flag che indica se animazione è infinita o no
- Altri dettagli

Funzioni che gestiscono animationSpec:

- **Tween**: anima un valore, per una data durata, possibilmente introducendo un ritardo iniziale, easing è la curva che definisce il valore durante l'interpolazione
 - `tween(durationMillis: Int, delayMillis: Int, easing: Easing)`
- **Spring**: crea un'animazione basandosi sul modello fisico di una molla
 - `spring(dampingRatio: Float, stiffness: Spring.Stiffness)`
- **Keyframes**: anima un valore basandosi su posizioni intermedie ad un dato tempo
 - o È possibile specificare una curva di interpolazione per definire la modalità di interpolazione in ogni intervallo
- o **repeatable(iterations: Int, animation: DurationBasedAnimationSpec, repeatMode: RepeatMode)**
 - performs a duration-based animation several times; it allows to indicate whether at the end of an interaction the next one should start again from the beginning (RepeatMode.Restart) or revert from the bottom (RepeatMode.Reverse)
- o **infiniteRepeatable(animation: DurationBasedAnimationSpec, repeatMode: RepeatMode)**
 - performs the past animation an indefinite number of times
- o **snap(delayMillis: Int)** – brings the animated value to the set target, without performing any interpolation

```
val value by animateFloatAsState(
    targetValue = 1f,
    animationSpec = keyframes {
        durationMillis = 375
        0.0f at 0 with LinearOutSlowInEasing // for 0-15 ms
        0.2f at 15 with FastOutLinearInEasing // for 15-75 ms
        0.4f at 75 // ms
        0.4f at 225 // ms
    }
)
```