



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

🚀 Day 23

Visual Revolution with Jetpack Compose

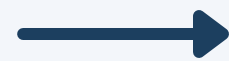
Beyond Aesthetics, Towards Unforgettable User Experience ⚡



Esmaeeil Moradi

@esmaeeil-moradi700

Day 23



Visual Revolution with Jetpack Compose – Beyond Aesthetics, Towards Unforgettable User Experience ⚡

Hey Android Architects, Designers, and Ambitious Developers!

If your app's UI isn't alive, your user experience is incomplete! The era of static, lifeless user interfaces is over. In the modern app world, animations and effects are no longer mere "aesthetics"; they are strategic tools for improving Perceived Performance, guiding users, boosting Engagement, and creating Delight. Today, on Day 23 of the #30DayAndroidChallenge, are you ready to spark a revolution in Android UI design? We're diving deep into the heart of advanced Jetpack Compose animations and effects to uncover the secrets of creating unforgettable user experiences. Let's bring our UIs to life!

Jetpack Compose, with its Declarative approach, paves the way for implementing complex animations with less code that is more readable and maintainable. This powerful framework blurs the line between visual designer and engineering developer, building a bridge that connects art and engineering.



Esmaeel Moradi

@esmaeel-moradi700

Foundational Principles of Animation in Compose: From Basics to Beyond



To create sensational animations, one must start from the foundational principles and then dive into deeper layers. Compose provides both Low-Level and High-Level tools for this purpose.

Composables Animatable: The Powerful Foundation for Manual Control

Animatable in Compose is the backbone of Low-Level animations. This composable allows you to manually animate arbitrary values with full control, from Float to Color, Dp, Size, Offset, and even Custom Data Types.

Code Example:

Animating a Float to change size or alpha.(Imagine this code creating a smooth and engaging animation that changes the size or transparency of a box.)



Esmaeel Moradi

@esmaeel-moradi700

```

@Composable
fun AnimatedScaleOrAlpha(
    targetValue: Float,
    animationSpec: AnimationSpec<Float> = tween(durationMillis = 1000)
) {
    val animatedFloat = remember { Animatable(0f) }

    LaunchedEffect(targetValue) {
        animatedFloat.animateTo(targetValue, animationSpec)
    }

    Box(
        modifier = Modifier
            .size(100.dp * animatedFloat.value)
            .alpha(animatedFloat.value)
            .background(Color.Blue)
    )
}

```

Low-Level vs. High-Level Animations: Why Unrivaled Control?

- **High-Level APIs:** Such as *AnimatedVisibility*, *Crossfade*, and *AnimatedContent* are excellent for common scenarios and simplifying entry/exit animations or content changes.
- **Low-Level APIs:** Including *Animatable* and *AnimationSpecs* (like *tween*, *spring*, *keyframes*, *repeatable*) give you unrivaled control over animation details. For creating truly sensational and custom animations, these APIs are essential.



Esmaeel Moradi

@esmaeeil-moradi700

Transition API:

Orchestrating Multiple Animations

When multiple properties need to be animated simultaneously, `updateTransition` and `Transition` act as a powerful orchestrator. This API allows you to manage several animations that start from one State and end in another.

Code Example: Simultaneously animating the size, alpha, and rotation of an element. (This code controls the animation of a box between "Collapsed" and "Expanded" states, simultaneously changing its size, transparency, and rotation.)

```
enum class BoxState { Collapsed, Expanded }

@Composable
fun MultiPropertyAnimation() {
    var boxState by remember { mutableStateOf(BoxState.Collapsed) }
    val transition = updateTransition(boxState,
        label = "boxStateTransition")

    val size by transition.animateDp(label = "size") { state ->
        if (state == BoxState.Collapsed) 50.dp else 100.dp
    }
    val alpha by transition.animateFloat(label = "alpha") { state ->
        if (state == BoxState.Collapsed) 0.5f else 1f
    }
    val rotation by transition.animateFloat(label = "rotation") { state ->
        if (state == BoxState.Collapsed) 0f else 360f
    }

    Box(
        modifier = Modifier
            .size(size)
            .alpha(alpha)
            .rotate(rotation)
            .background(Color.Red)
            .clickable {
                boxState = if (boxState == BoxState.Collapsed)
                    BoxState.Expanded else BoxState.Collapsed
            }
    )
}
```



Esmaeel Moradi

@esmaeel-moradi700



Crafting Sensational Visual Effects: Beyond Movement

Compose isn't limited to animating Composables movement; you can also implement stunning and creative visual effects.

GraphicsLayer and Transform Power:

Modifier.graphicsLayer is an excellent tool for applying visual Transforms like rotation, scale, translation, shadowElevation, and alpha. These are applied directly to the composable's graphical layer and offer excellent performance.

Advanced Technique: Building a Card with a 3D Flip effect on click.

(This animation smoothly flips a card in 3D to reveal its other side, creating an engaging and interactive experience.)



Esmaeel Moradi
@esmaeeil-moradi700



```
@OptIn(ExperimentalAnimationGraphicsApi::class)
@Composable
fun FlipCard() {
    var flipped by remember { mutableStateOf(false) }
    val rotationY by animateFloatAsState(
        targetValue = if (flipped) 180f else 0f,
        animationSpec = tween(durationMillis = 800)
    )

    Card(
        modifier = Modifier
            .graphicsLayer {
                rotationY = rotationY
                cameraDistance = 8 * density // Perspective effect
            }
            .clickable { flipped = !flipped }
            .size(200.dp, 120.dp),
        elevation = CardDefaults.cardElevation(4.dp)
    ) {
        Box(
            modifier = Modifier.fillMaxSize(),
            contentAlignment = Alignment.Center
        ) {
            Text(
                text = if (rotationY <= 90f) "Front" else "Back",
                fontSize = 24.sp,
                color = Color.White,
                modifier = Modifier.graphicsLayer {
                    alpha = if (rotationY <= 90f) 1f else 0f
                    rotationY = if (rotationY <= 90f) 0f else 180f
                }
            )
        }
    }
}
```



Esmaeel Moradi
@esmaeeil-moradi700

Custom Drawing with Canvas and DrawScope:

For creating entirely custom shapes or animations that fall outside the default Composables, Canvas and DrawScope give you direct access to Android's low-level drawing capabilities.

Code Example: Animating a Custom Progress Indicator. (This code creates a custom circular progress indicator, where the filled portion smoothly increases as the progress value changes.)

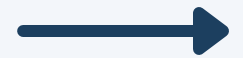
```
@Composable
fun AnimatedProgressIndicator(progress: Float) {
    Canvas(modifier = Modifier.size(100.dp)) {
        drawArc(
            color = Color.LightGray,
            startAngle = 0f,
            sweepAngle = 360f,
            useCenter = false,
            style = Stroke(width = 8.dp.toPx())
        )
        drawArc(
            color = Color.Green,
            startAngle = -90f,
            sweepAngle = 360 * progress,
            useCenter = false,
            style = Stroke(width = 8.dp.toPx())
        )
    }
}
```



Esmaeel Moradi

@esmaeeil-moradi700

Shaders and Pixel Effects:



Shaders allow you to apply complex visual effects directly to a *Paint* or *Composable*, such as animated gradients or shimmering effects.

Conceptual Example: *Creating Text with an Animated Gradient or Shimmer Effect.* (This capability allows you to create text with animated gradient colors or a shimmering effect, similar to what you see in loading placeholders.)

```
// Conceptual example; actual implementation involves
// a custom modifier and a shader.
@Composable
fun AnimatedGradientText(text: String) {
    Text(
        text = text,
        style = MaterialTheme.typography.headlineLarge,
        // Modifier.animatedGradientBrush(...)
    )
}
```



Esmaeel Moradi

@esmaeeil-moradi700

Path Animations: Magical Tracings:



Animating Paths enables you to draw, morph, or trace shapes. This is excellent for logo animations, animated icons, or line-drawing effects.

Conceptual Example: *A Line Drawing animation for an icon or logo. (This code conceptually shows how a Path can be gradually drawn, as if tracing a line with a pen.)*

```
// Conceptual example
@Composable
fun PathDrawingAnimation() {
    val path = remember { Path().apply { /* define your path */ } }
    var animProgress by remember { mutableStateOf(0f) }

    LaunchedEffect(Unit) {
        Animatable(0f).animateTo(1f, tween(2000)) {
            animProgress = value
        }
    }

    Canvas(modifier = Modifier.size(150.dp)) {
        val pathMeasure = PathMeasure().apply {
            setPath(path, false)
        }
        val segment = Path()
        pathMeasure.getSegment(0f, pathMeasure.length * animProgress,
            segment, true)
        drawPath(
            path = segment,
            color = Color.Black,
            style = Stroke(width = 4.dp.toPx())
        )
    }
}
```



Esmaeel Moradi

@esmaeeil-moradi700

Interactive Animations and Gestures:

Live UI in the User's Hands



A truly alive UI responds to user interaction and simulates natural movements.

Gestures and Responsive Animations:

By combining `Modifier.pointerInput` with `Animatable`, you can create animations directly driven by user gestures. Decay Animations (with fling) are essential for simulating natural motion after an element is released.

Code Example:

A Draggable Card that returns to its original position or flings off-screen upon release. (This code creates a draggable card that responds to user finger movements, and upon release, it either smoothly returns to its original position or flings off-screen with a natural animation.)

```
@OptIn(ExperimentalFoundationApi::class)
@Composable
fun DraggableCard() {
    val offsetX = remember { Animatable(0f) }
    val offsetY = remember { Animatable(0f) }
    val decay = rememberSplineBasedDecay<Float>()

    Card(
        modifier = Modifier
            .offset { IntOffset(offsetX.value.roundToInt(),
                offsetY.value.roundToInt()) }
            .pointerInput(Unit) {
                coroutineScope {
                    while (true) {
                        val pointerId = awaitPointerEventScope {
                            awaitFirstDown().id
                        }
                    }
                    offsetX.stop()
                    offsetY.stop()
                }
            }
    )
}
```



Esmaeel Moradi

@esmaeel-moradi700

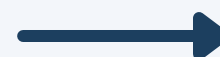


```
        val tracker = VelocityTracker()
        awaitPointerEventScope {
            horizontalDrag(pointerId) { change ->
                val originalX = offsetX.value
                val originalY = offsetY.value
                launch {
                    offsetX.snapTo(originalX +
                        change.position.x - change.previousPosition.x)
                    offsetY.snapTo(originalY +
                        change.position.y - change.previousPosition.y)
                }
                tracker.addPosition(change.uptimeMillis,
                    change.position)
            }
        }

        val velocity = tracker.calculateVelocity()
        val targetX = if (abs(velocity.x) > 200) {
            with(decay) {
                decay.calculateTarget(offsetX.value, velocity.x)
            }
        } else 0f
        val targetY = if (abs(velocity.y) > 200) {
            with(decay) {
                decay.calculateTarget(offsetY.value, velocity.y)
            }
        } else 0f

        launch {
            offsetX.animateTo(targetX, decay,
                initialVelocity = velocity.x)
        }
        launch {
            offsetY.animateTo(targetY, decay,
                initialVelocity = velocity.y)
        }
    }
}

.size(150.dp)
.background(Color.Cyan),
elevation = CardDefaults.cardElevation(8.dp)
) {
    Text("Drag Me!", modifier = Modifier.align(Alignment.CenterHorizontally))
}
}
```



Shared Element Transitions with Compose (Navigation Component):

Creating Smooth Transitions between Composables during navigation, especially with shared elements, significantly enhances the user experience. Compose, with the Navigation Component, provides this capability.

Conceptual Example:

Animating an image transition between two screens.(This feature allows an image or common element to smoothly move from one location to another when navigating between two screens.)



Esmaeel Moradi
@esmaeeil-moradi700

```
// Conceptual example for shared element transition setup
@Composable
fun DetailScreen(imageUrl: String, navHostController: NavHostController) {
    Image(
        painter = rememberImagePainter(imageUrl),
        contentDescription = null,
        modifier = Modifier
            .size(200.dp)
            // .sharedElement(...) // Used for shared elements
    )
}

@Composable
fun ListScreen(navHostController: NavHostController, items: List<String>) {
    LazyColumn {
        items(items) { imageUrl ->
            Image(
                painter = rememberImagePainter(imageUrl),
                contentDescription = null,
                modifier = Modifier
                    .size(50.dp)
                    .clickable {
                        // navHostController.navigate(...) // Navigation logic
                    }
                    // .sharedElement(...) // Used for shared elements
            )
        }
    }
}
}
```

MotionLayout and Compose (Interoperability):

In highly complex scenarios requiring precise Motion control that Compose's native APIs might not fully cover, you can use MotionLayout (from the View System) within Compose via AndroidView. This approach is less common but useful for gradual migration or leveraging specific MotionLayout capabilities.



Esmaeel Moradi

@esmaeeil-moradi700

Best Practices and Performance Optimization for Animations: Beauty Without Sacrificing Speed

Beautiful animations are useless if they cause lag or slow down the application. Optimizing performance in Compose is crucial.

Avoiding Unnecessary Recomposition:

The importance of Stable Types, remember, derivedStateOf, and key to ensure that Composables recompose only when necessary, keeping animations smooth.

Hardware Acceleration and Layers:

Modifier.graphicsLayer is used not only for Visual Effects but also for applying Hardware Acceleration to Composables, which significantly improves Rendering Performance.

Profiling and Debugging Animations:

Android Studio tools like Layout Inspector and CPU Profiler are essential for identifying performance Bottlenecks in animations.

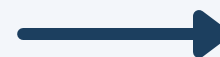
Accessibility in Animations:

Always consider users with special needs. Providing text alternatives for important animations or allowing control over animation speed via Accessibility settings is crucial for an Inclusive UX.



Esmaeel Moradi

@esmaeel-moradi700



Conclusion:

Beyond Aesthetics, Crafting Unforgettable Experiences

Jetpack Compose is not just a UI framework; it's a powerful canvas for creating unparalleled user experiences.

Animations are no longer just for "beauty"; they are strategic tools for **User Engagement, Storytelling** within the application, and even visual user guidance.

Are you ready to transform lifeless UIs into dynamic works of art? Jetpack Compose is in your hands to turn dull UIs into living, unforgettable experiences. Start now and amaze the Android world with your creativity!

Join us and share your stunning animation showcases built with Compose! Which Compose animation has excited you the most?



Esmaeel Moradi

@esmaeel-moradi700

Looking forward to your
insights and experiences!
Let's elevate Android
development together.



Esmaeeil Moradi

@esmaeeil-moradi700