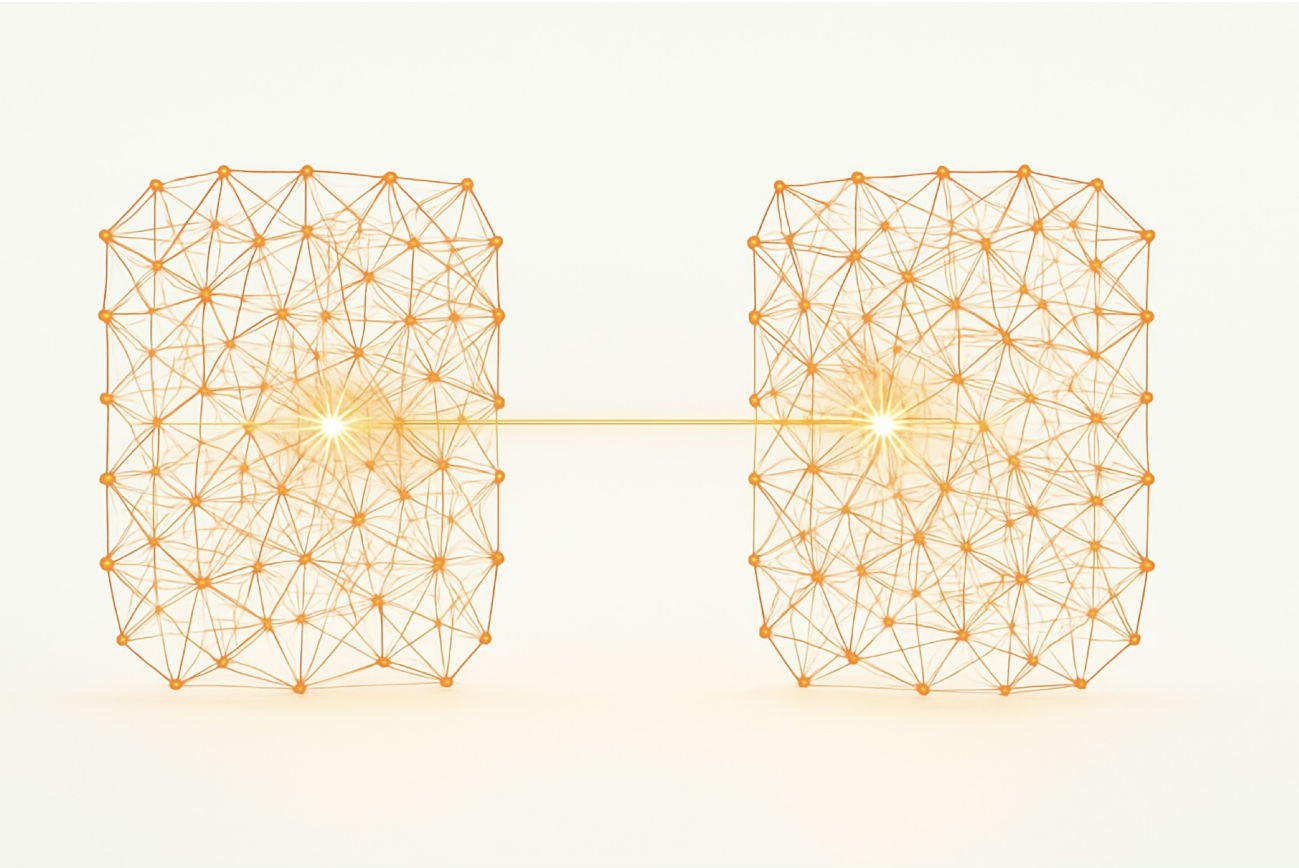
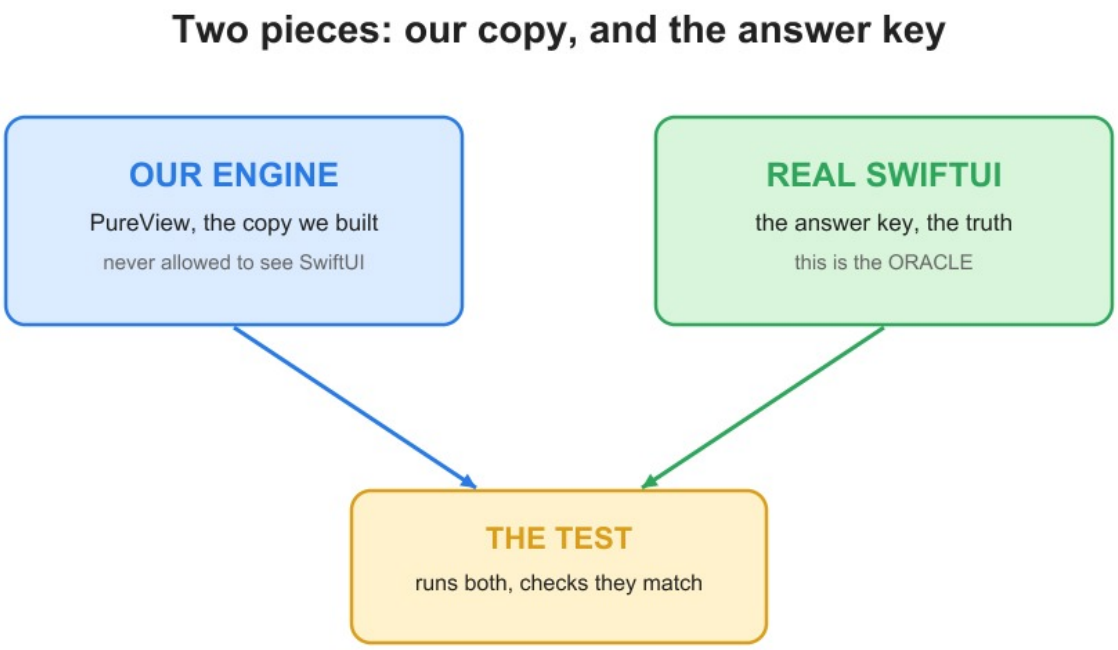
"The SwiftUI Oracle: Measuring a Clean Room Against the Real Thing"



The SwiftUI Oracle: Measuring a Clean Room Against the Real Thing

In the last post I claimed I had measured SwiftUI until its behavior was predictable. This post is the receipts. It is for Swift developers, so it is mostly code: the test harness, the firewall that keeps it honest, and the differential tests that hold a from-scratch engine against the real framework. If you have ever wondered what it would take to *prove* you understand SwiftUI rather than assert it, this is one answer.

The engine is called PureView. It is a clean-room reimplementation of SwiftUI's semantics: the attribute graph, identity, layout, animation, and lowering to a display list. It never imports SwiftUI. The proof that it is faithful is a separate test target that *does* import SwiftUI and compares the two, value for value.

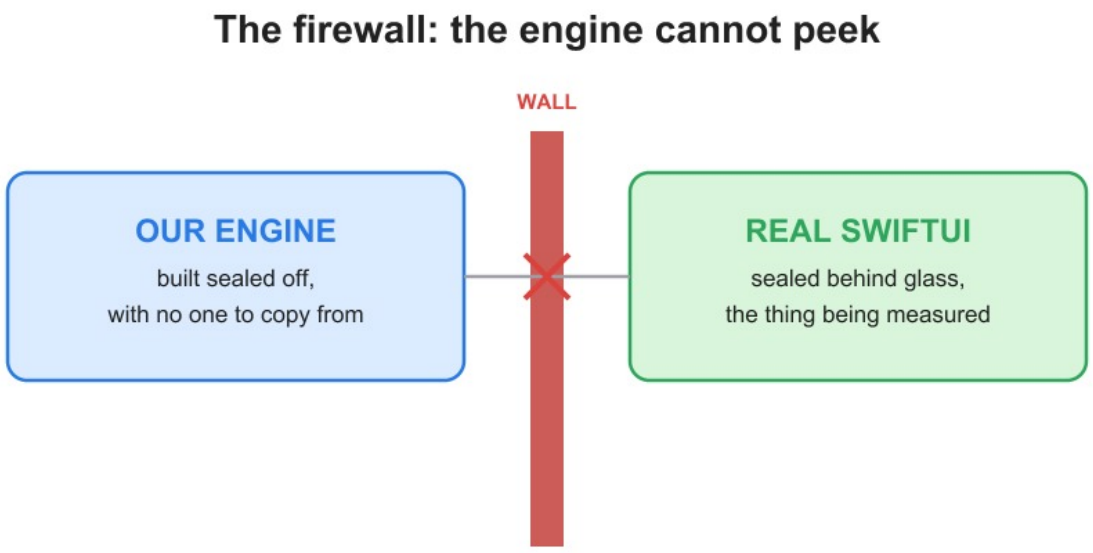


What "oracle" means here

In testing, an **oracle** is the source of the correct answer: the thing that tells you whether a result is right. Our oracle is real SwiftUI itself. For any scene, we ask the real framework what it does, and that answer is the ground truth we grade the engine against. The engine is the work; SwiftUI is the answer key.

The firewall: the engine may never import SwiftUI

A measurement is worthless if the instrument can cheat. If the engine were allowed to call SwiftUI, any agreement would be suspect, because the engine could simply forward to the framework it claims to reproduce. So the engine is sealed off, and the seal is not a convention. It is a test that fails `swift test` if a UI-framework import ever appears in the engine sources.



A build-failing test enforces the wall: no SwiftUI import ever enters the engine.

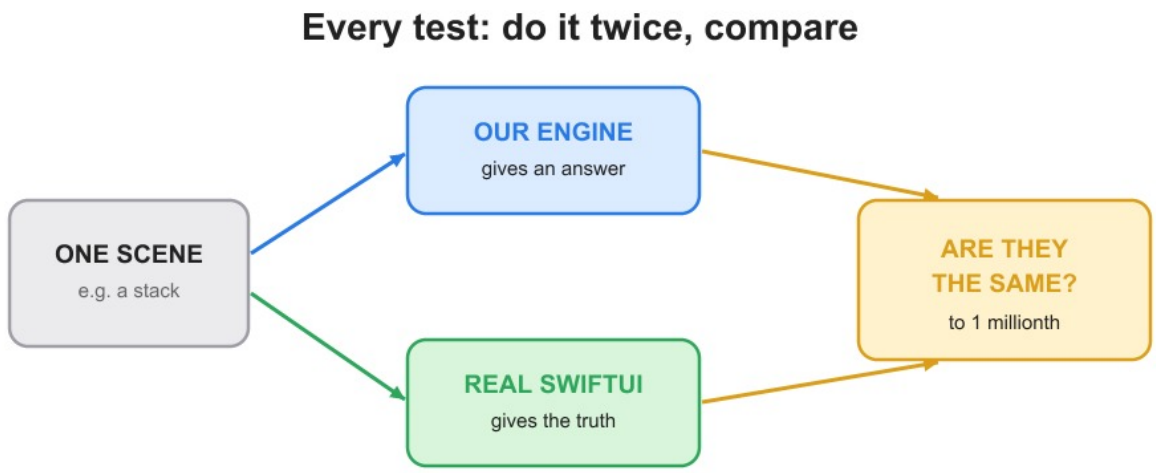
```
@Suite("Engine import confinement")
struct EngineImportConfinementTests {
    @Test("Sources/PureView imports no UI framework")
    func engineImportsNoUIFramework() throws {
        let forbidden = ["SwiftUI", "UIKit", "AppKit", "WatchKit"]
        // #filePath = <root>/Tests/PureViewTests/<this file>; climb to
<root>/Sources/PureView.
        let engineRoot = URL(filePath: #filePath)
            .deletingLastPathComponent() // PureViewTests
            .deletingLastPathComponent() // Tests
            .deletingLastPathComponent() // <root>
            .appending(path: "Sources/PureView")

        let fileManager = FileManager.default
        let enumerator = try #require(
            fileManager.enumerator(at: engineRoot, includingPropertiesForKeys:
nil),
            "engine source root must exist at \(engineRoot.path)"
        )

        func importsForbidden(_ line: String) -> Bool {
            forbidden.contains { framework in
                line == "import \(framework)"
                    || line == "@testable import \(framework)"
                    || line.hasPrefix("import \(framework).")
            }
        }

        var offenders: [String] = []
        for case let url as URL in enumerator where url.pathExtension ==
"swift" {
```

```
            let source = try String(contentsOf: url, encoding: .utf8)
            for (index, line) in source.split(separator: "\n",
omittingEmptySubsequences: false).enumerated() {
                let trimmed = line.trimmingCharacters(in: .whitespaces)
                if importsForbidden(trimmed) {
                    offenders.append("\(url.lastPathComponent):\(index + 1):
\(trimmed)")
                }
            }
        }

        #expect(offenders.isEmpty, "engine must not import a UI framework,
found: \(offenders)")
    }
}
```

SwiftUI, UIKit, and AppKit are system frameworks, so a stray `import SwiftUI` in an engine file would compile green and ship undetected. The convention is not enough; this test makes the leak fail the build. That is the whole integrity story in thirty lines.

The shape of every test: do it twice, compare

Every oracle test has the same skeleton. Describe a scene. Run it in real SwiftUI and record the answer. Run the same scene in the engine. Assert the two agree, to the floating-point floor.



The harness: running real SwiftUI headless

To use SwiftUI as the oracle we have to run it for real, with no app and no screen. The harness hosts a view in an offscreen window, lets it run a genuine update cycle, then reads the resolved geometry back out through a preference-key probe.

```
private struct FrameKey: SwiftUI.PreferenceKey {
    static let defaultValue: [String: CGRect] = [:]
    static func reduce(value: inout [String: CGRect], nextValue: () -> [String:
CGRect]) {
        value.merge(nextValue(), uniquingKeysWith: { _, new in new })
    }
}

extension SwiftUI.View {
    /// Tag a view so the oracle records its resolved frame, in root
coordinates.
    func probe(_ id: String) -> some SwiftUI.View {
        background(
            GeometryReader { proxy in
                Color.clear.preference(key: FrameKey.self, value: [id:
proxy.frame(in: .named(kRoot))])
            }
        )
    }
}

@MainActor
func oracleFrames(_ content: some SwiftUI.View, rootSize: CGSize) -> [String:
CGRect] {
    let box = FrameBox()
    let view = ZStack(alignment: .topLeading) { content }
        .frame(width: rootSize.width, height: rootSize.height,
alignment: .topLeading)
        .coordinateSpace(name: kRoot)
        .onPreferenceChange(FrameKey.self) { box.frames = $0 }
        .environment(\.displayScale, 2)   // pin the pixel grid for
cross-machine determinism

    let host = NSHostingView(rootView: view)
    host.frame = CGRect(origin: .zero, size: rootSize)
    let window = NSWindow(contentRect: host.frame, styleMask: [.borderless],
                          backing: .buffered, defer: false)
    window.contentView = host
    host.layoutSubtreeIfNeeded()
    pump(until: { !box.frames.isEmpty }, timeout: 2)
    return box.frames
}

/// Spin the main run loop in 10 ms slices until `condition` holds or `timeout`
elapses.
@MainActor
```

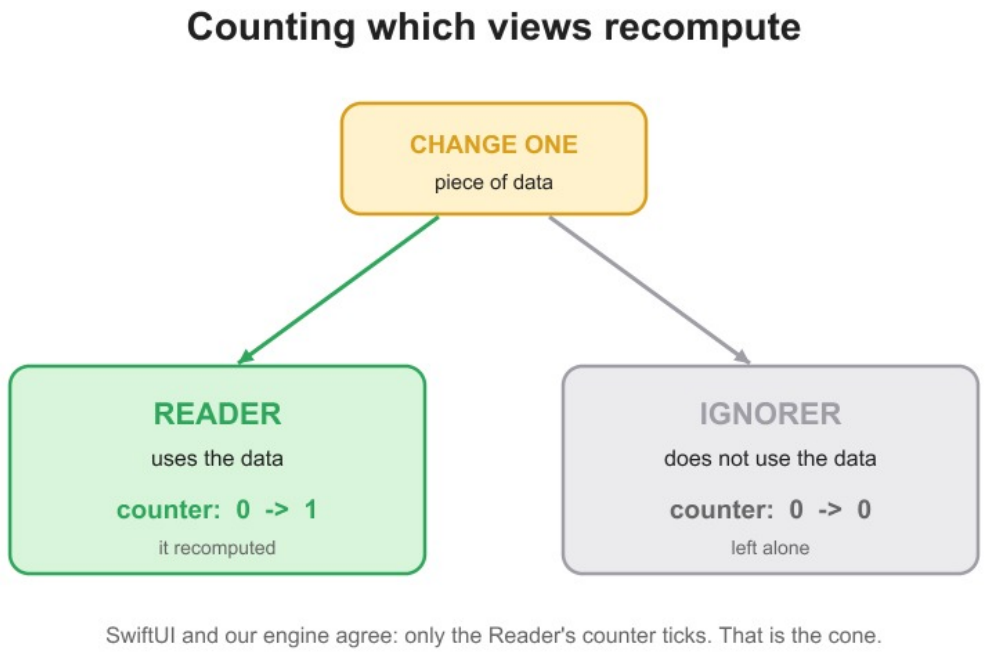
```
func pump(until condition: () -> Bool, timeout: TimeInterval) {
    let deadline = Date().addingTimeInterval(timeout)
    while !condition(), Date() < deadline {
        RunLoop.main.run(until: Date().addingTimeInterval(0.01))
    }
}
```

`displayScale` is pinned to 2 so SwiftUI's frame snapping is identical on a Retina machine and a CI box. When we say "we measured what SwiftUI does," this is the literal mechanism: SwiftUI runs, and a `GeometryReader` reports its own layout system's answer.

Measuring the attribute graph: count the body re-runs

The headline claim about SwiftUI is the cone: change one value, and only the views that read it recompute. The problem is that a `body` re-run leaves no trace. So we give it one. Each probe view bumps a counter inside its `body`. The invisible event becomes an integer.

```
/// A reference counter mutated only on the main actor (inside `body`).
private final class BodyCounter: @unchecked Sendable {
    private(set) var runs = 0
    func bump() { runs += 1 }
}

private final class OracleModel: SwiftUI.ObservableObject {
    @SwiftUI.Published var value: Int = 0
}

/// Reads `model.value`, so it depends on the state.
private struct ReaderView: SwiftUI.View {
    @SwiftUI.ObservedObject var model: OracleModel
    let counter: BodyCounter
    var body: some SwiftUI.View {
        counter.bump()
        return Text(verbatim: "\(model.value)")
    }
}

/// Reads nothing from the model, so it must not re-evaluate when the state
changes.
private struct IgnorerView: SwiftUI.View {
    let counter: BodyCounter
    var body: some SwiftUI.View {
        counter.bump()
        return Text(verbatim: "static")
    }
}
```



SwiftUI and our engine agree: only the Reader's counter ticks. That is the cone.

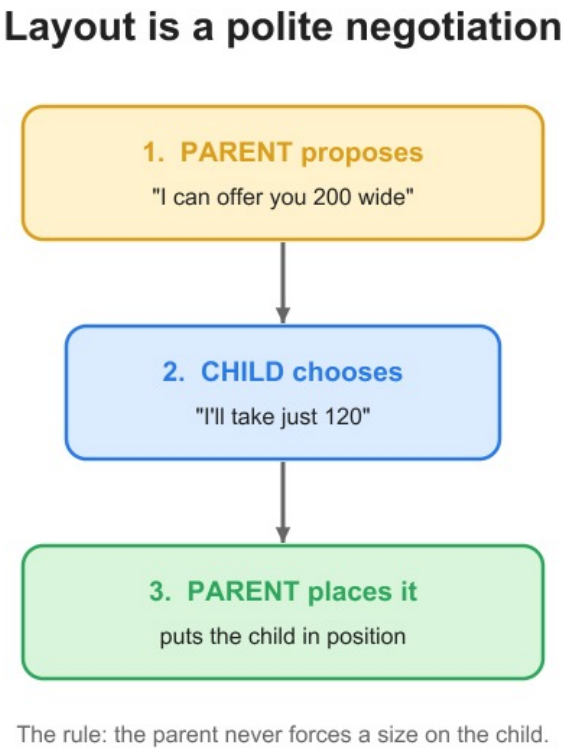
The test reads both counters, changes one value, waits for the reader to re-run, and then drains the run loop a little longer so a stray re-run of the ignorer would be caught. Then it builds the same shape in the engine's `AttributeGraph` and asserts the cones match.

```
@Test("only the view that reads state re-evaluates, and the engine cone agrees")
func bodyEvalConeMatchesEngine() {
    // Ground truth from real SwiftUI.
    let swiftUI = swiftUIBodyEvalDifferential()
    #expect(swiftUI.readerReran,  "SwiftUI: the reader's body should re-run on
the state change")
    #expect(!swiftUI.ignorerReran, "SwiftUI: the ignorer's body should not
re-run")

    // The same shape modeled in the engine: a source, a reader, an ignorer.
    let graph = AttributeGraph()
    let state = Attribute(graph: graph, value: 0)
    let reader = Attribute(graph: graph) { _ in state.value }
    let ignorer = Attribute(graph: graph) { _ in 0 }
    _ = reader.value
    _ = ignorer.value
    let readerBefore = reader.evaluationCount
    let ignorerBefore = ignorer.evaluationCount

    state.setValue(1)
    _ = reader.value
    _ = ignorer.value

    // The differential: the engine's invalidation cone matches SwiftUI's.
    #expect((reader.evaluationCount  > readerBefore)  == swiftUI.readerReran)
    #expect((ignorer.evaluationCount > ignorerBefore) == swiftUI.ignorerReran)
}
```

Proving an *absence* is the subtle part. To show the ignorer stayed put, the test waits for the expected re-run and then drains briefly for a stray. The engine side is deterministic and is the real regression catcher; the bounded wait on the SwiftUI side can only catch a stray, never manufacture a false pass.

Measuring layout: probe the resolved frame

Layout is the largest area, because there is the most to pin down. Each test lays the scene out in real SwiftUI, reads back every subview's frame through the probe, and asserts the engine places each node at the same coordinates.




```
@Test("Frame centering matches real SwiftUI")
func frameCentering() {
    let leaf = Color.clear.frame(width: 20, height: 20).probe("child")
    let container = leaf.frame(width: 30, height: 30).probe("container")
    let oracle = oracleFrames(container, rootSize: CGSize(width: 100, height:
100))
    #expect(!oracle.isEmpty, "oracle produced no frames (headless measurement
failed)")

    let node = FrameLayout(width: 30, height: 30, FixedLeaf(Size(width: 20,
height: 20)))
    let engine = node.layout(in: Size(width: 100, height: 100))

    if let oc = oracle["container"] {
        #expect(close(oc, cg(engine.frame)), "container: oracle \(oc) vs engine
\(cg(engine.frame))")
    }
```

```
    if let ch = oracle["child"] {
        #expect(close(ch, cg(engine.children[0].frame)), "child: oracle \(ch)
vs engine \(cg(engine.children[0].frame))")
    }
}
```

`close` is where the rigor lives. The tolerance absorbs only `CGFloat` round-trip, nothing more.

```
/// The engine and SwiftUI must produce the *same* frame, not a close one. This
1e-6
/// absorbs only `CGFloat` round-trip representation. A residual above it is a
real
/// divergence, never slack: re-derive the rule, do not widen the bound.
let exactFloor: CGFloat = 1e-6

func close(_ a: CGRect, _ b: CGRect, tol: CGFloat = exactFloor) -> Bool {
    abs(a.minX - b.minX) <= tol && abs(a.minY - b.minY) <= tol &&
        abs(a.width - b.width) <= tol && abs(a.height - b.height) <= tol
}
```

When a layout test fails, the rule is to find why the behavior differs and fix the engine, never to widen `exactFloor` until the failure disappears. A tolerance you are allowed to relax is a wish, not a measurement.

Measuring animation: hold the spring against SwiftUI's Spring

Animation is the most quantitative area, because motion is numbers over time. A spring is a damped harmonic oscillator. Released from rest, its progress curve is

$$p(t) = 1 - e^{-\text{zeta omega}_0 t}[\cos(\text{omega}_d t) + \frac{\text{zeta omega}_0}{\text{omega}_d} \sin(\text{omega}_d t)]$$

which starts at 0, sweeps past 1, and settles back. We sample the engine's spring and SwiftUI's own `Spring` at the same times and assert they agree across all three damping regimes.

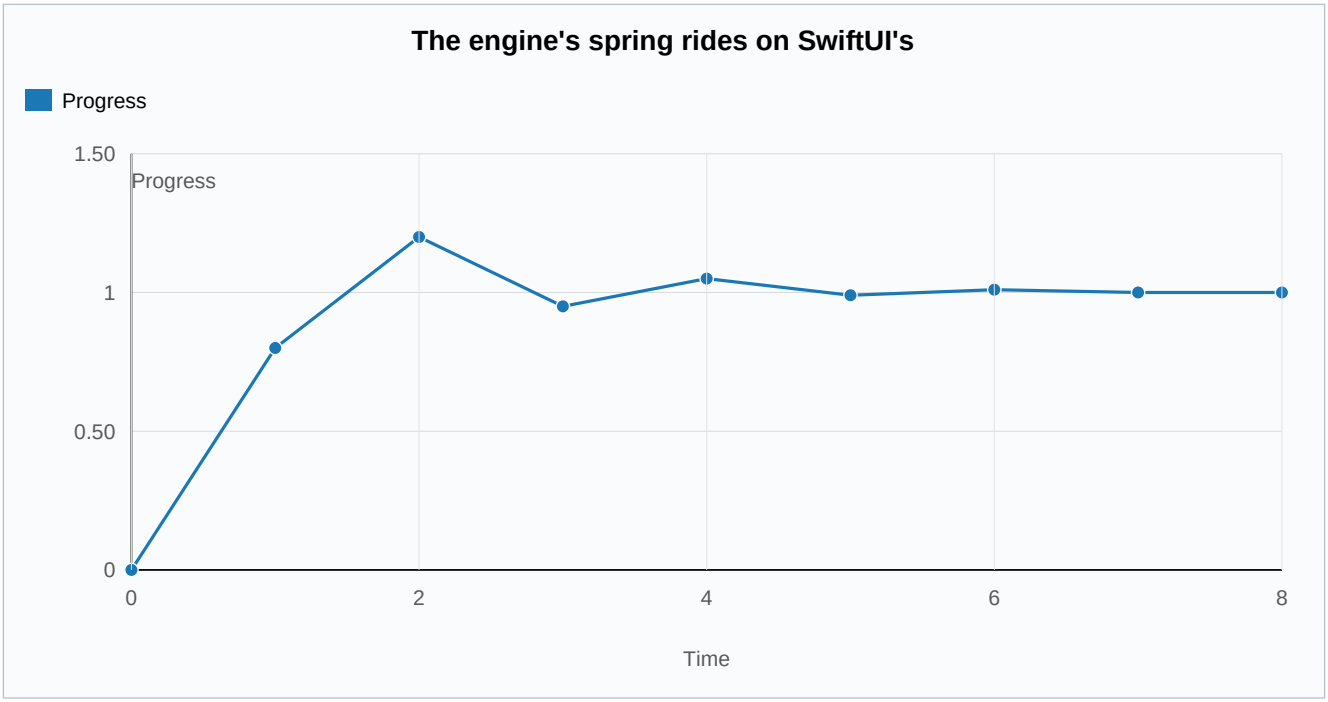
```
@available(macOS 14, iOS 17, *)
private func assertSpringMatches(
    response: Double,
    dampingFraction: Double,
    tolerance: Double = 1e-6
) {
    let engine = PureView.Spring(response: response, dampingFraction:
dampingFraction)
    let oracle = SwiftUI.Spring(response: response, dampingRatio:
dampingFraction)
    let samples = 120
    for step in 0 ... samples {
        let t = engine.settleDuration * Double(step) / Double(samples)
        let engineValue = engine.unitStepResponse(at: t)
        let oracleValue = oracle.value(target: 1.0, initialVelocity: 0.0, time:
```

```
t)
        #expect(
            abs(engineValue - oracleValue) <= tolerance,
            "spring(response=\(response), damping=\(dampingFraction)) at
t=\(t): " +
            "engine \(engineValue) vs SwiftUI \(oracleValue)"
        )
    }
}
```

The assertion is the whole point, in one line:

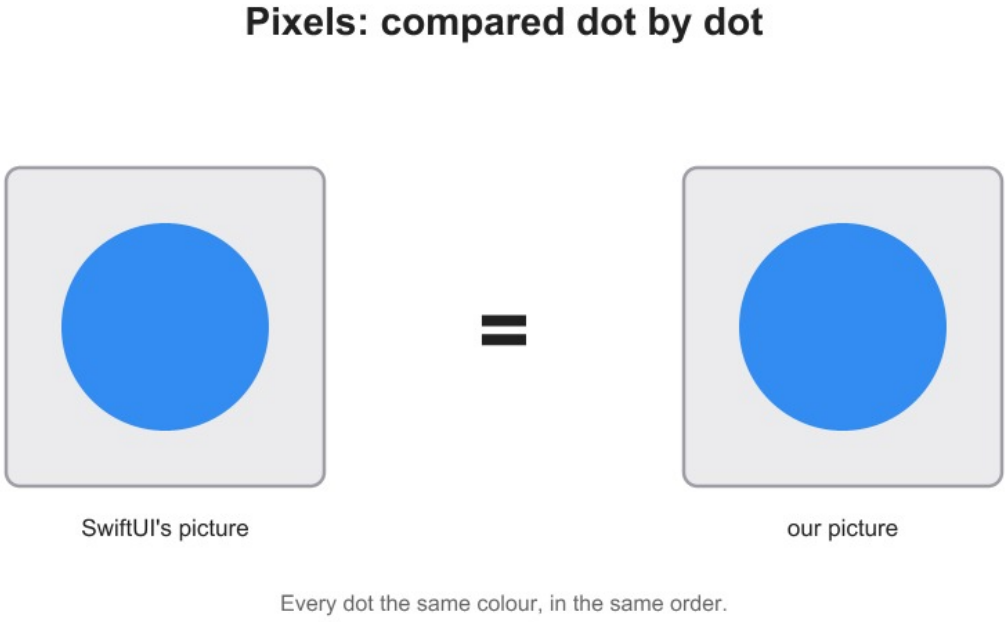
$$
\max_{t}\; \bigl|\, v_{\text{engine}}(t) - v_{\text{SwiftUI}}(t) \,\bigr| \;\le\; 10^{-6}
$$

Sampled point by point, the two springs are the same number to six decimals.



Measuring pixels: compare the rendered bytes

Frames prove position. Pixels prove the rest: the actual color drawn, and the paint order of overlapping fills. SwiftUI's own `ImageRenderer` rasterizes the view; the engine's display list is rasterized by a reference renderer; we diff the buffers channel by channel.



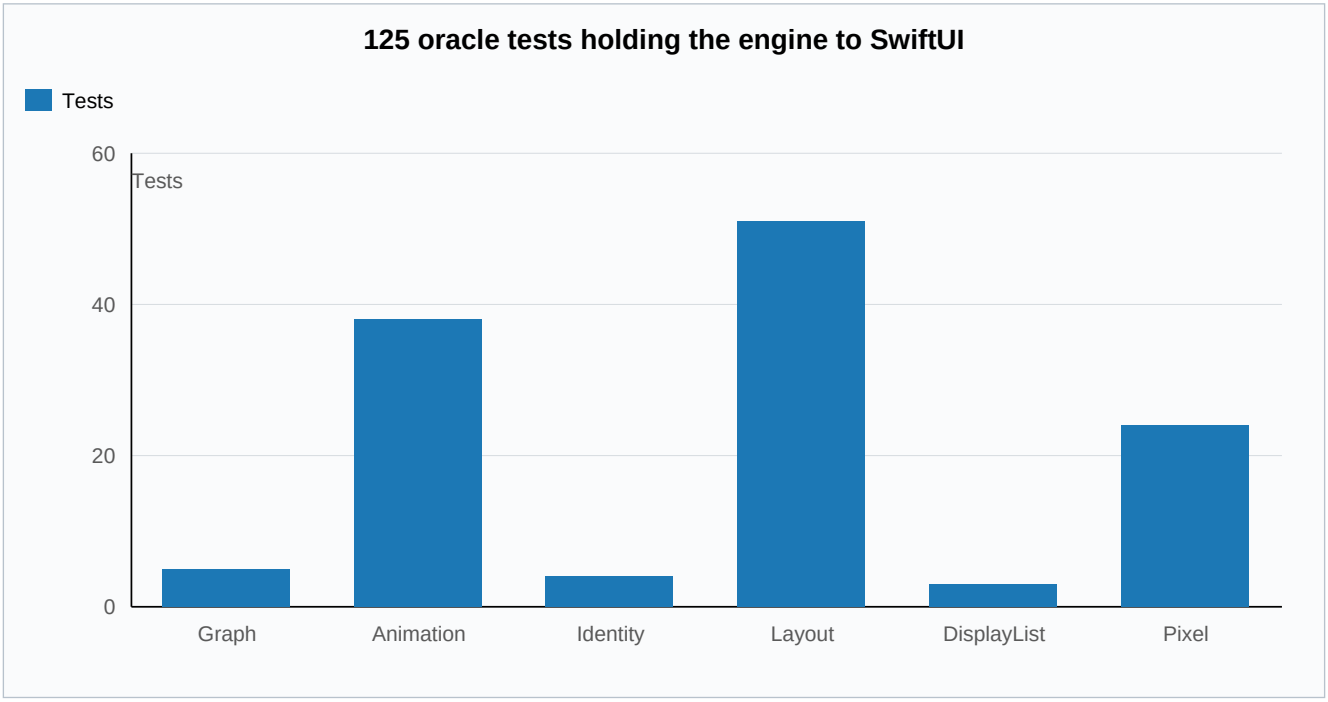

```
/// Render a SwiftUI view to an RGBA8 buffer in the same
space/scale/orientation.
private func snapshot(_ view: some SwiftUI.View, size: CGSize, scale: CGFloat)
throws -> [UInt8] {
    let renderer = ImageRenderer(content: view.frame(width: size.width, height:
size.height))
    renderer.scale = scale
    renderer.colorMode = .nonLinear
    let image = try #require(renderer.cgImage, "ImageRenderer produced no
image")
    let w = Int(size.width * scale), h = Int(size.height * scale)
    let (ctx, raw) = try context(width: w, height: h, scale: 1, flipForDrawing:
true)
    defer { raw.deallocate() }
    ctx.draw(image, in: CGRect(x: 0, y: 0, width: CGFloat(w), height:
CGFloat(h)))
    return bytes(raw, count: w * 4 * h)
}

private func assertPixelsMatch(
    _ name: String,
    list: DisplayList,
    view: some SwiftUI.View,
    size: CGSize,
    scale: CGFloat = 2,
    tolerance: Int = 2
) throws {
    let engine = try rasterize(list, size: size, scale: scale)
    let swiftui = try snapshot(view, size: size, scale: scale)
    #expect(!engine.isEmpty && engine.count == swiftui.count, "\(name): buffer
size mismatch")
```

```
    var maxDiff = 0, exceeding = 0
    for i in 0 ..< min(engine.count, swiftui.count) {
        let diff = abs(Int(engine[i]) - Int(swiftui[i]))
        maxDiff = max(maxDiff, diff)
        if diff > tolerance { exceeding += 1 }
    }
    #expect(exceeding == 0, "\(name): \(exceeding) channels exceed tol
\(tolerance) (maxDiff \(maxDiff))")
}
```

The tolerance here is 2 on an 8-bit channel: the quantization floor is one least-significant bit, and integer-aligned fills at scale 2 have crisp edges, so there is no anti-aliasing slack to hide behind.

What it all adds up to

There are 125 of these tests today, across 19 files. The same do-it-twice method holds every layer the engine rebuilds.



What measuring caught that reading never would

The payoff: the measurement caught a bug in the engine that reading the code could not. The spring decided it had come to rest using the bare decay envelope, which undercounts the long tail of a critically damped or overdamped spring. The code looked correct. Measured against SwiftUI, the spring clamped a few percent short of its target, a visible snap. The oracle failed with the exact residual, and the settling time was rederived from the real basis of the oscillator. Reading gives you the happy path. Measuring gives you the edges, where the interesting bugs live.

One honest caveat about the animation path

A committed animation keeps running smoothly when the main thread is busy only for one of two paths. Layer-level properties, opacity and transform, lower to Core Animation and are interpolated on the render server's own clock, off the main thread. Value-driven animations, a spring on a custom value, are re-pulled through the graph every frame on the main thread, and those do stutter if you block it. The oracle measures the value and velocity of the motion; which thread interpolates it is a property of the lowering path. Worth stating precisely, since the rest of this post is about precision.

Colophon: every figure was made by my own software

No third-party drawing or charting tool was used anywhere in this post.

The conceptual diagrams (the wall, the two pieces, the counters, the negotiation, the pixel comparison) were drawn by **PureDraw**, my dependency-free Swift-native 2D vector engine, through its `GraphicsContext` API. PureDraw rasterized them to the JPEGs shown here with its own `BitmapRenderer` and `JPEGEncoder`, and also produced scalable SVG and PDF masters. No external image tool touched them.

The bar chart, the spring plot, and every mathematical formula were rendered by **TileDown**, my static-site engine, from plain-text fences in this Markdown file.

This page was assembled by **TileDown**, and its PDF was typeset by **MarkdownPDF**, my Markdown-to-PDF engine.

PureDraw is part of **SlayerMotion**, the graphics engine family this series is about. **TileDown** and **MarkdownPDF** are separate projects of mine, not part of SlayerMotion. All three are my own software; none of it is anyone else's.