Article

Reducing the rendering cost of your UI on visionOS

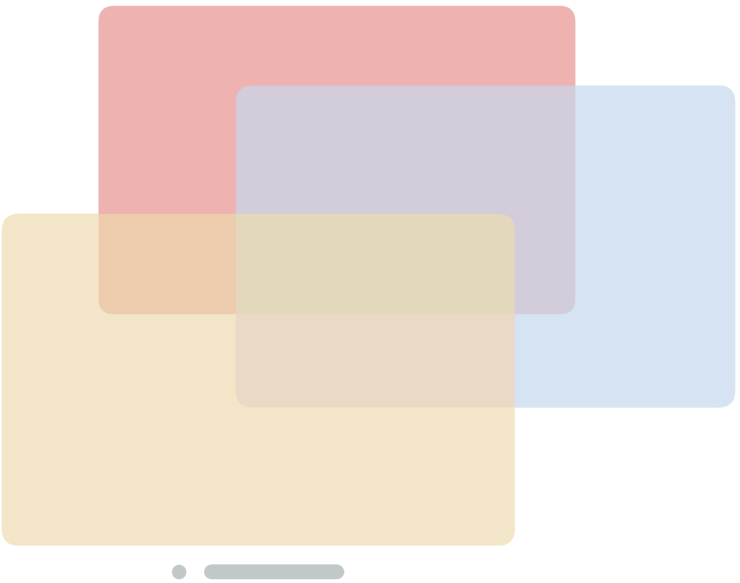
Optimize your 2D user interface rendering on visionOS.

Overview

Provide an enjoyable experience and maintain a sense of immersion on Apple Vision Pro by minimizing visual choppiness and interaction latency. Performance bottlenecks that prevent timely rendering and responsive feedback interfere with the spatial experience overall and can cause disorientation or discomfort if they persist over an extended period of time. Your app's processing and its views have an impact on the work the system does and its ability to meet rendering deadlines. To address bottlenecks in your SwiftUI or UIKit interfaces, first analyze your apps performance, then implement some of the following strategies to reduce CPU and GPU overhead. For more information on performance analysis, see Analyzing the performance of your visionOS app.

Minimize transparency in overlapping views

If you have overlapping UI windows, avoid adding translucency to them. In SwiftUI, avoid setting the value of your view's `opacity(_:)` below 1. In UIKit, avoid setting the value of your view's `alpha` below 1. Subviews inherit `alpha`.



When a view with transparent pixels overlaps another view, the system performs extra work to composite those views together. The GPU always renders foreground content, but when foreground content is transparent the GPU also renders the UI behind it. When the foreground content is fully opaque, the GPU doesn't do this rendering.

The Translucent UI Meshes metric in the Core Animation section of the RealityKit Metrics instrument timeline helps you identify translucent UI content your app uses:

Other visual effects that involve *overdraw*, the need to draw pixels multiple times to produce a final result, also increase rendering work. In the Shared Space, overdraw can result from interactions with the content from other apps, so minimizing translucent content might have a greater impact. For design guidance, see Human Interface Guidelines > Windows. Visual effects can also cause offscreen passes. To reduce offscreen passes, see Reducing the rendering cost of your UI on visionOS.

Reduce the size of static UI views

To lower the GPU overhead of static UI content, reduce the size of UI elements in your view hierarchy. Making your UI content appear larger in a space requires rendering more pixels which requires more rendering work. People still have control over re-sizing content, so use sizes that are large enough for people to be comfortable interacting with, but not so large that they require unnecessary demand from the GPU.

To identify areas in your app where rendering your static UI content creates a lot of overhead for the GPU, check the RealityKit Metrics instrument for 3D Render GPU bottlenecks. Content you display in 2D windows still renders in a space with a 3D mesh. If you expect people to frequently launch multiple windows, account for the rendering cost of this in your design and profile your app with multiple windows open. For design guidance, see Human Interface Guidelines > Spatial layout.

Consider the impact of dynamic content scaling

The system can provide sharper visuals from any angle and distance by altering the resolution of text or vector-based UI content based on where people are looking. Doing so requires drawing content more frequently and at potentially higher scales. SwiftUI and UIKit views enable this by default. To opt in with your custom Core Animation or Core Graphics rendering, enable the `wantsDynamicContentScaling` property of any `CALayer` object. Consider the performance and memory trade-offs and profile your app with this feature on and off to determine if the cost is worth the quality improvement.

For more information on dynamic content scaling, see Drawing sharp layer-based content in visionOS and the section on Dynamic content scaling in the video Explore rendering for spatial computing.

Reduce redraw and offscreen rendering

The performance cost of numerous redraws and offscreen render passes adds up. Offscreen passes can also contribute to additional overdraw. Some complex renderers require offscreen passes but you can avoid them in many cases. Learn more about offscreen render passes in Customizing render pass setup. To reduce the number of redraw and offscreen render passes your app performs:

- Lower the update rates of animations.
- Pause or stop animations.
- Reduce the number of different views your app uses.
- Avoid layouts that require redrawing overlapping layers when redrawing a different layer.

- Use more efficient alternatives than `UIVisualEffectView`. For example, use background colors in your SwiftUI and UIKit apps.

Similar to other Apple platforms, app updates done in response to input events, timed animations, or other sources initiate UI redraw work in the render server. On visionOS, dynamic content scaling can cause redraw to be frequent in the absence of these updates. This makes it more important to reduce updates and animation timers.

To reduce the cost of redraws and render passes:

- Avoid large layers that might require redrawing a large number of pixels.
- Minimize your use of vector-based media content during active scrolling and animated interactions. Animating large vector-based media content is often expensive.
- Reduce the resolution, type, and number of images in your content that scrolls or animates during interactions.

The number of offscreen render passes can negatively impact performance. Visual effects, like shadows, masking, rounded rectangles, blurs, and vibrancy increase the number of offscreen passes the system requires to render your content.

To identify frequent or expensive UI redraws, check the RealityKit Metrics instrument for Core Animation Encoding (CPU), Core Animation GPU (GPU), Core Animation Server Update (CPU), and Core Animation Client Commit (CPU) bottlenecks.

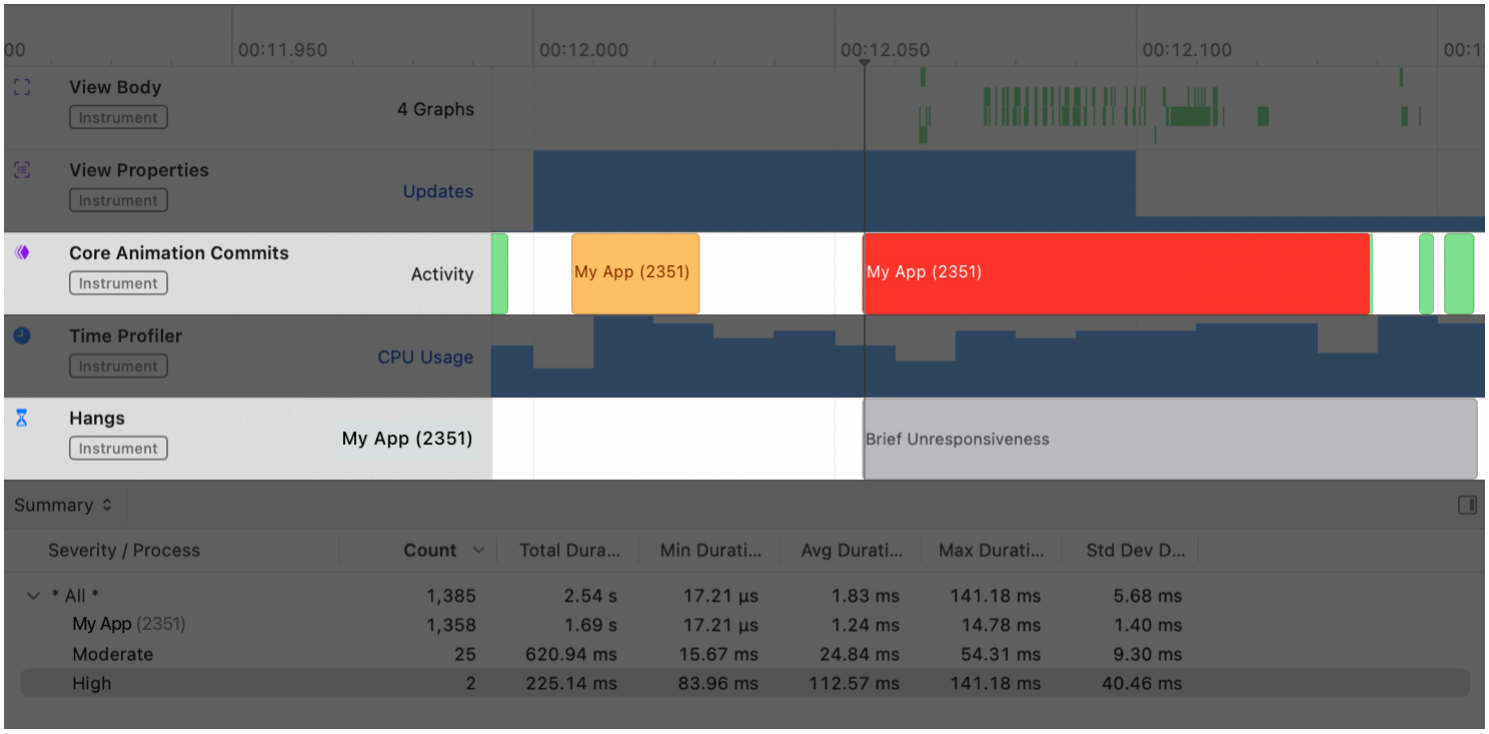
Minimize view layouts and updates in your SwiftUI, UIKit, or custom Core Animation and Core Graphics rendering when you see a bottleneck related to UI redraw to make your app more efficient for the system to render.

Review the Offscreen Prepares and Render Passes metrics in the Core Animation Render section of the RealityKit Metrics Instrument to identify the number of potential offscreen render passes and total render passes the render server does on behalf of your app. The section includes metrics on render passes for both offscreen visual effects and for each region of UI content to redraw.

To learn more about these offscreen passes, watch the video Demystify and eliminate hitches in the render phase.

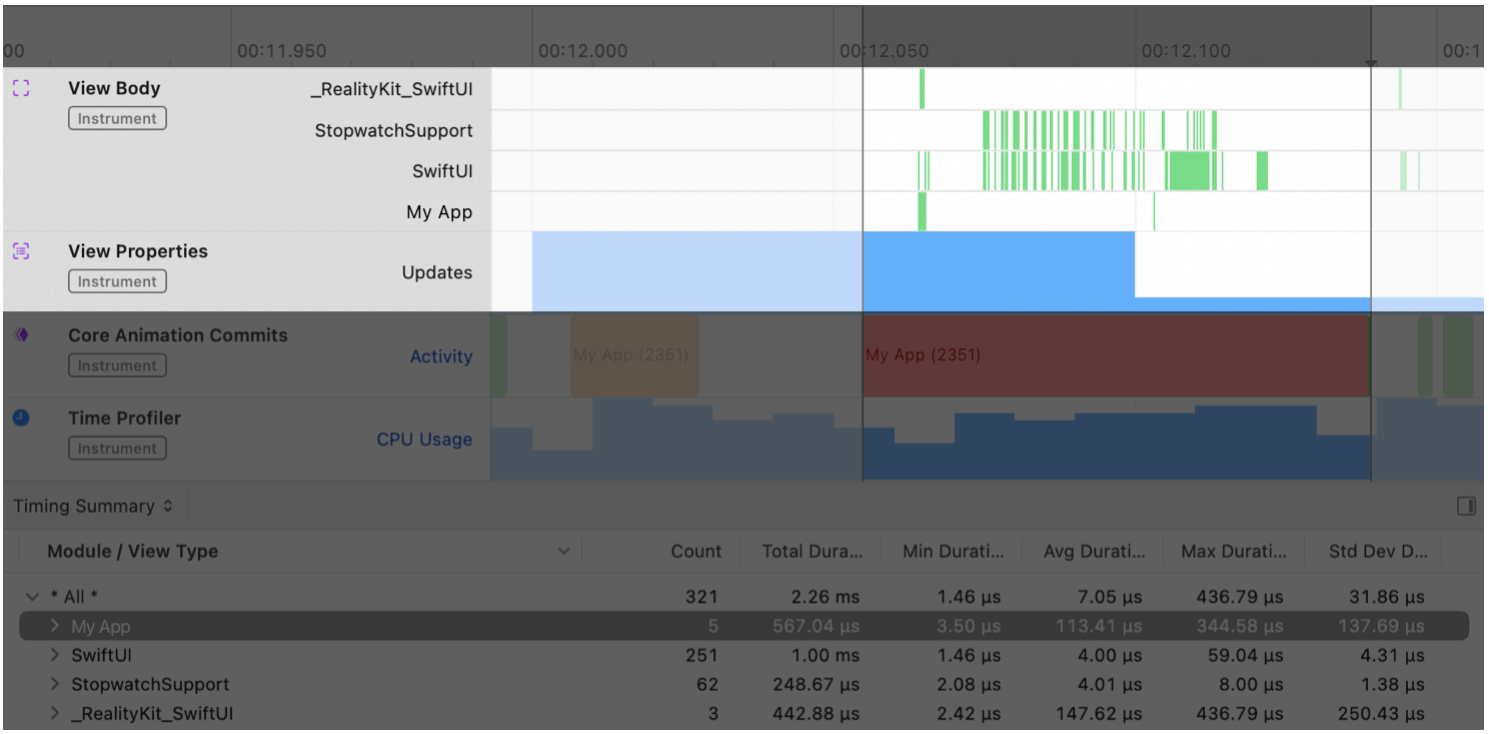
Avoid expensive updates on the main thread

To avoid visible hitches and delays while people scroll and resize, reduce time consuming layout and view creations that occur on the app's main thread. The SwiftUI Instrument template includes the View Body, View Properties, Core Animation Commits, Time Profiler, and Hangs instruments. Use the data these instruments collect to debug expensive commits and updates on your app's main thread. The Core Animation Commits and Hangs instruments can help you locate areas of expensive work and updates on the app's main thread that cause rendering delays:



To learn more about analyzing Hangs with Instruments, watch Analyze hangs with Instruments.

The View Body and View Properties instruments provide metrics on the number of view body creations and property updates that occur:



For debugging purposes, you can add a call to the internal method `Self._printChanges()` from the body of the view to log information about the property that caused the view to update.

```
var body: some View {
    let _ = Self._printChanges()
    // View code
}
```

See Also

Related articles

- Creating a performance plan for your visionOS app

  Identify your app's performance and power goals and create a plan to measure and assess them.

- Analyzing the performance of your visionOS app

  Use the RealityKit Trace template in Instruments to evaluate and improve the performance of your visionOS app.

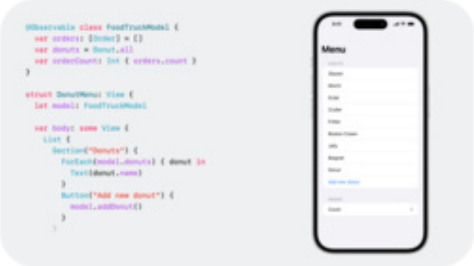
- Reducing the rendering cost of RealityKit content on visionOS

  Optimize your app's 3D augmented reality content to render efficiently on visionOS.

- Understanding the visionOS render pipeline

Compare how visionOS handles events and manages its rendering loop differently from other Apple platforms.

Related videos



Discover Observation in SwiftUI

Demystify SwiftUI performance

Demystify SwiftUI