Article

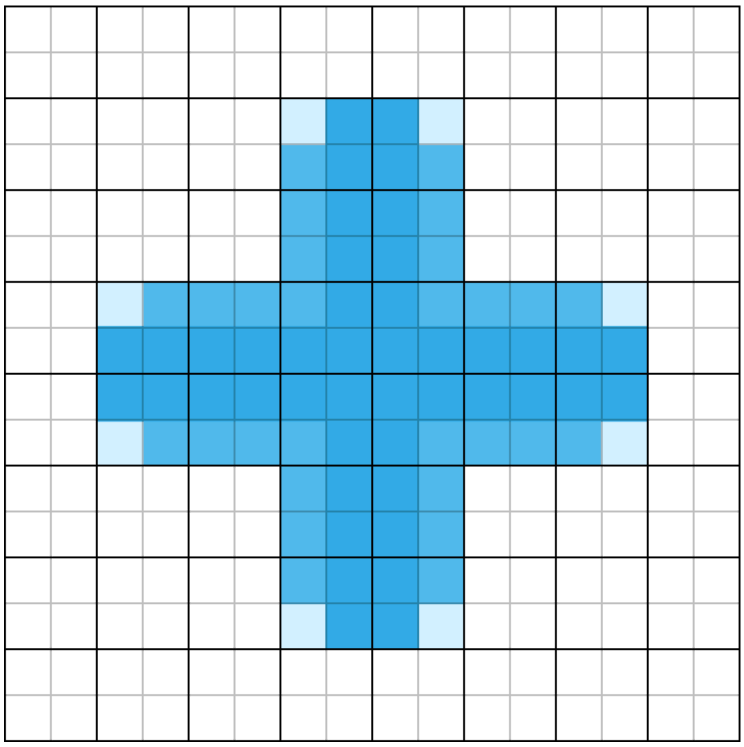
Drawing sharp layer-based content in visionOS

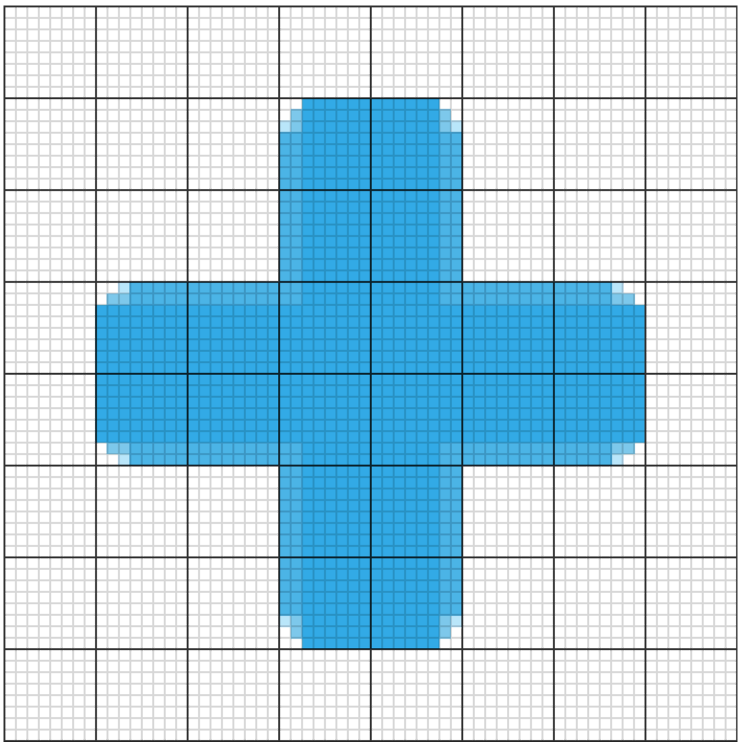
Deliver text and vector images at multiple resolutions from custom Core Animation layers in visionOS.

Overview

If your app uses Core Animation layers directly, update your layer code to draw a high-resolution version of your content when appropriate. SwiftUI and UIKit views use Core Animation layers to manage interface content efficiently. When a view draws its content, the underlying layer captures that content and caches it to improve subsequent render operations.

Core Animation on most Apple platforms rasterizes your layer at the same resolution as the screen, but Core Animation on visionOS can rasterize at different resolutions to maximize both content clarity and performance. The system follows the person’s eyes and renders content immediately in front of them at the highest possible resolution. Outside of this focal area, the system renders content at progressively lower resolutions to reduce GPU workloads. Because the content is in the person’s peripheral vision, these lower resolutions don’t impact the content’s clarity. As the person’s eyes move around, the system redraws content at different resolutions to match the change in focus.


A figure at 2x resolution


A figure at 8x resolution

If you deliver content using custom `CALayer` objects, you can configure your custom layers to support drawing at different resolutions. If you don't perform this extra configuration step, each layer rasterizes its content at a @2x scale factor, which is good enough for most content and matches what the layer provides on a Retina display. However, if you opt in to drawing at different resolutions, the layer rasterizes its content at up to @8x scale factor in visionOS, which adds significant detail to text and vector-based content.

Request dynamic scaling for custom layers

Dynamic content scaling is off by default for all Core Animation layers, and frameworks or apps must turn on this support explicitly. If your interface uses only SwiftUI or UIKit views, you don't need to do anything to support this feature. SwiftUI and UIKit enable it automatically for views that benefit from the added detail, such as text views and image views with SF Symbols or other vector-based artwork. However, the frameworks don't enable the feature for all views, including `UIView` and `View`.

If your visionOS interface includes custom Core Animation layers, you can enable the `wantsDynamicContentScaling` property of any `CALayer` objects that contain vector-based content. Setting this property to `true` tells the system that you support rendering your layer's content at different resolutions. However, the setting is not a guarantee that the system applies dynamic content scaling to your content. The system can disable the feature if your layer draws using incompatible functions or techniques.

The following example shows how to enable this feature for a `CATextLayer` object. After configuring the layer, set the `wantsDynamicContentScaling` property to `true` and add the layer to your layer hierarchy.

```
let layer = CATextLayer()

layer.string = "Hello, World!"
layer.foregroundColor = UIColor.black.cgColor
layer.frame = parentLayer.bounds

// Setting this property to true enables content scaling
// and calls setNeedsDisplay to redraw the layer's content.
layer.wantsDynamicContentScaling = true

parentLayer.addSublayer(layer)
```

Dynamic content scaling works best when the layer contains text or vector-based content. Don't enable the feature if you do any of the following in your layer:

- You set the layer's content using the `contents` property.
- You draw primarily bitmap-based content.
- You redraw your layer's contents repeatedly over a short time period.

The `CAShapeLayer` class ignores the value of the `wantsDynamicContentScaling` property and always enables dynamic content scaling. For other Core Animation layers, you must enable the feature explicitly to take advantage of it.

Draw the layer's content dynamically

Dynamic content scaling requires you to draw your layer's contents using one of the prescribed methods. If you define a custom subclass of `CALayer`, draw your layer's content in the `draw(in:)` method. If you use a `CALayerDelegate` object to draw the layer's content, use the delegate's `draw(_:in:)` method instead.

When you enable dynamic content scaling for a layer, the system captures your app's drawing commands for playback later. As the person's eyes move, the system draws the layer at higher resolutions when someone looks directly at it, or at lower resolutions otherwise. Because the redraw operations implicitly communicate what the person is looking at, the system performs them outside of your app's process. Letting the system handle these operations maintains the person's privacy while still giving your app the benefits of high-resolution drawing.

Some Core Graphics routines are incompatible with dynamic content scaling. Even if you enable dynamic content scaling for your layer, the system automatically disables the feature if your layer uses any of the following:

- Core Graphics shaders.

- APIs that set intent, quality, or other bitmap-related properties. For example, don't call `CGContextSetInterpolationQuality`.
- A `CGBitmapContext` to draw content.

If your app creates timer-based animations, don't animate layer changes using your drawing method. Calling `setNeedsDisplay()` on your layer repeatedly in a short time causes the system to draw the layer multiple times in quick succession. Because visionOS needs a little extra time to draw a layer at high resolution, each redraw request forces it to throw away work. A better option is to animate layer-based properties to achieve the same effect, or use a `CAShapeLayer` to animate paths when needed.

Modify layer hierarchies to improve performance

The backing store for a layer consumes more memory at higher resolutions than at lower resolutions. Measure your app's memory usage before and after you enable dynamic content scaling to make sure the increased memory cost is worth the benefit. If your app's memory usage increases too much, limit which layers adopt dynamic content scaling. You can also reduce the amount of memory each layer uses in the following ways:

- Make your layer the smallest size possible. Larger layers require significantly more memory, especially at higher resolutions. Make the size of the layer match the size of your content by eliminating padding or extra space.
- Separate complex content into different layers. Instead of drawing everything in a single layer, build your content from multiple layers and arrange them hierarchically to achieve the same result. Enable dynamic content scaling only in the layers that actually need it.
- Apply special effects using layer properties whenever possible. Applying effects during drawing might require you to increase the layer's size. For example, apply scale and rotation effects to the layer's `transform` property, instead of during drawing.
- Don't draw your layer's content at different resolutions in advance and cache the images. Maintaining multiple images requires more memory. If you do cache images, draw them only at @2x scale factor.
- Don't use your drawing code to draw a single image. If your layer's content consists of an image, assign that image to the layer's `contents` property directly.

Complex drawing code can also lead to performance issues. A layer with many strokes can render quickly at lower scale factors, but might be computationally too complex to render at larger scales. If a complex layer doesn't render correctly at higher resolutions, turn off dynamic content scaling and measure the render times again.

See Also

App construction

Creating your first visionOS app

Build a new visionOS app using SwiftUI and add platform-specific features.

Adding 3D content to your app

Add depth and dimension to your visionOS app and discover how to incorporate your app’s content into a person’s surroundings.

Creating fully immersive experiences in your app

Build fully immersive experiences by combining spaces with content you create using RealityKit or Metal.

Introductory visionOS samples

Learn the fundamentals of building apps for visionOS with beginner-friendly sample code projects.