Article

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.

Overview

Performance tuning is an important part of the development process, regardless of platform. Performance tuning means making your app run as efficiently as possible, so it does more work in less time and with fewer system resources. Efficiency is especially important on devices that can support multiple apps in an immersive experience. Apps that consume too many resources, can push the device beyond thermal limits. When this occurs, the system takes steps to cool down to a more acceptable level. This can have a noticeable visual impact and be disorienting for the wearer.

As you start development, set aggressive goals and evaluate progress throughout the development cycle. Automate the collection of performance metrics as much as possible and look at data over time to see if performance is improving or declining. When you detect a significant decrease in performance, take immediate steps to correct it. When you start fine-tuning early in development, you have more time to make needed changes to algorithms and approaches.

For more information on performance tuning, see Improving your app's performance.

Set performance and power targets

Performance isn't a single metric that you measure and improve. Typically, you choose several metrics and set goals for each of them. For example, consider:

App launch and load times
: Make sure your app launches quickly; this is your first chance to make a good impression.

Responsiveness and latency

Your interface needs to respond quickly to interactions, even while doing other work. Minimize the time it takes to start tasks. For example, make sure audio and video start without noticeable delays.

Graphics rendering
: For an immersive experience with realtime rendering, it's important to maintain consistently high frame rates. Help maintain these rates by avoiding unnecessary changes that result in more frequent updates to the shared render server. Measure things like update rates, stalls, and hangs in both the render server and your app. Only render the content you need, and optimize the textures and other resources you use during drawing.

Power Usage
: When the device begins to reach thermal limits, the system reduces CPU or GPU usage and performance degrades over time. Avoid this thermal ceiling by prioritizing and spreading out work, limiting the number of simultaneous threads your app maintains, and turning off hardware-related features like Core Location when you don't need them.

Task efficiency
: Make the app do as much as possible using the smallest amount of hardware resources. Minimize task-based overhead.

Memory footprint and bandwidth
: Use as little free memory as possible. Don't allocate or deallocate memory during critical operations, which might make your app appear slow.

After you choose the metrics you want, set realistic goals and prioritize them, so you know which ones matter the most. Performance tuning often involves making tradeoffs between competing goals. For example, if you reduce CPU usage by caching computed data or pre-load assets to improve responsiveness, you increase your app's memory usage. Make these kinds of tradeoffs carefully, and always measure the results of any changes to learn whether they were successful. In some cases, you might find the sacrifice isn't worthwhile.

Consider how people will use your app. If your app runs in the Shared Space, consider more conservative targets and goals for system resources. If you expect people to use your app for longer periods of time, factor this extended use into your targets and goals when choosing metrics.

Identify the code flows and user scenarios to test

After you choose the metrics to collect, decide which portions of your app to test. Choose features that are repeatable, measurable, and reliable to test. Repeatable automated tests allow you to compare the results and know the comparisons represent the exact same task. Focus on places where your app executes code, but don't ignore places where your app hands off data to the system and waits. If your app spends a significant amount of time waiting for information, consider eliminating the requests altogether or batching them to achieve better performance.

Focus your tuning efforts on the parts of your app that people use the most, or that have the most impact on overall system performance, including:

- User-facing workflows
- Key algorithms
- Task that allocate or deallocate memory
- Background and network-based tasks
- Custom Metal shaders

Choose actions that people perform frequently or that correspond to important features. For example, if your app lets someone add a new contact, test the workflow for creating the contact, editing the contact, and saving the results. Test your app with a particular feature enabled and disabled to determine whether the feature is solely responsible for any performance impacts. Choose lightweight workflows such as how your app performs at idle time, and also heavyweight workflows, for example, ones that involve user interactions and your app's responses. For launch times, gather metrics for both hot and cold launches — that is, when the app is already resident in memory and when it is not.

Consider thermal and environmental factors

Consider how environmental factors impact your app. The characteristics of your physical environment can affect system load and thermals of the device. Consider the effect that ambient room temperature, the presence of other people, and the number and type of real-world objects can have on the your app's algorithms. Try to test in different settings to get an idea of whether you need to optimize for these scenarios or not.

Use Xcode's thermal inducers to mimic the device hitting its thermal limits and consider how your app responds to fair, serious, and critical thermal notifications. You might need to have different performance goals when under thermal pressure, and prioritize optimizing for power or find ways to dynamically lower your app's complexity in response to thermal pressure to give a smoother experience, even if latency is a bit higher.

Choose tools to collect performance data

There are many tools and APIs you can use to collect performance-related data for your visionOS app. Use a variety of tools to make sure you have the data you need:

Debug gauges
: Monitor the CPU, memory, disk and network gauges in the Debug navigator to track system resources utilization.

Instruments

Profile your app to gather performance data on most metrics. Instruments lets you profile your app's code execution, find memory leaks, track memory allocations, analyze file-system or graphics performance, SwiftUI performance, and much more. Use the RealityKit Trace template to monitoring and investigate render server stalls and bottlenecks on visionOS.

XCTest
: Use XCTest APIs to collect performance data.

MetricKit
: Use MetricKit to gather on-device app diagnostics and generate reports.

Organizer
: Review diagnostic logs for hangs, disk and energy usage, and crashes in the Xcode Organizer.

Reality Composer Pro
: Review statistics on the contents of your RealityKit scenes. Use this information to optimize your 3D models and textures.

Signposts
: Add signposts to your code to generate timing information you can view in Instruments. For more information, see Recording performance data.

Log messages
: Include log messages to report significant events and relevant data for those events. For more information, see Generating log messages from your code.

TestFlight
: Get feedback from testers about their experiences with beta versions of your app. Fill out the Test Information page for your beta version, and request that testers provide feedback about the performance of your app.

Profile on a physical device

In general, profile and analyze performance on a physical device rather than in Simulator. Even if something works well in Simulator, it might not perform as well on devices for all use cases. Simulator doesn't support some hardware features and APIs. There are differences in the rendering pipeline for Simulator running on macOS, so rendering performance characteristics will be different. Other pipelines such as input delivery and audio or video playback are also different. There are, however, some insights you can gain profiling in Simulator, such as CPU stalls, that help you spot areas to investigate and address.

Build automated test cases and run them regularly

Xcode comes with tools to help you automate the collection of performance data:

- Use the XCTest framework to build test cases to collect performance metrics. XCTest lets you gather several different metrics, including the time it takes to perform operations, the amount of CPU activity that occurs during the test, details about memory or storage use, and more.
- Use Instruments to collect metrics for specific interactions with your app. Record those interactions and play them back later to collect a new set of metrics.
- Write custom scripts to gather performance-related data using system command-line tools. Integrate these scripts into your project's build process to automate their execution.

Configure Xcode to run test cases each time you build your app, or create a separate target to run test cases or custom scripts on demand. Integrate your performance tests into your Xcode Cloud workflows, or your own custom continuous integration solution.

> **Note**
>
> Collect performance data using a production version of your app to obtain more accurate results. Debug builds contain additional code to support debugging operations and logging. You can collect data from debug builds too, but keep those metrics separate from production-build metrics.

For information about how to write test cases for your app, see Testing. For information about how to automate testing with Xcode Cloud, see Xcode Cloud.

See Also

Performance

- 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 your UI on visionOS
  Optimize your 2D user interface rendering on visionOS.
- 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.