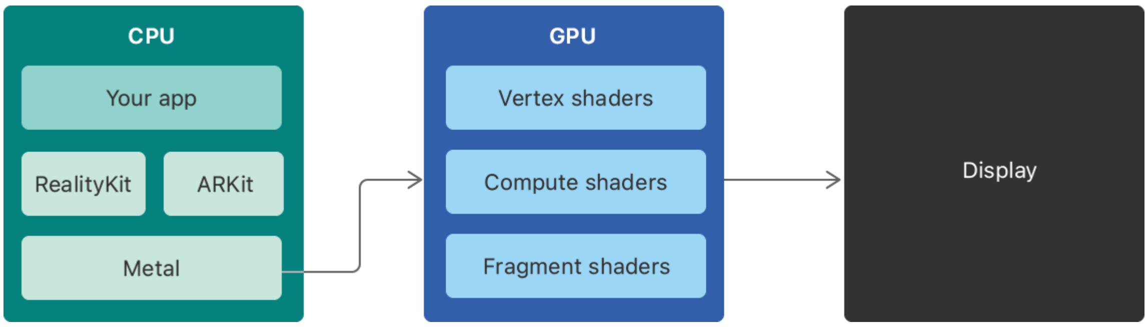
Improving the Performance of a RealityKit App

Measure CPU and GPU utilization to find ways to improve your app's performance.

Overview

You use the RealityKit framework to add 3D content to an ARKit app. The framework runs an entity component system (ECS) on the CPU to manage tasks like physics calculations, animations, audio processing, and network synchronization. It also relies on the Metal framework and GPU hardware to perform multithreaded rendering.



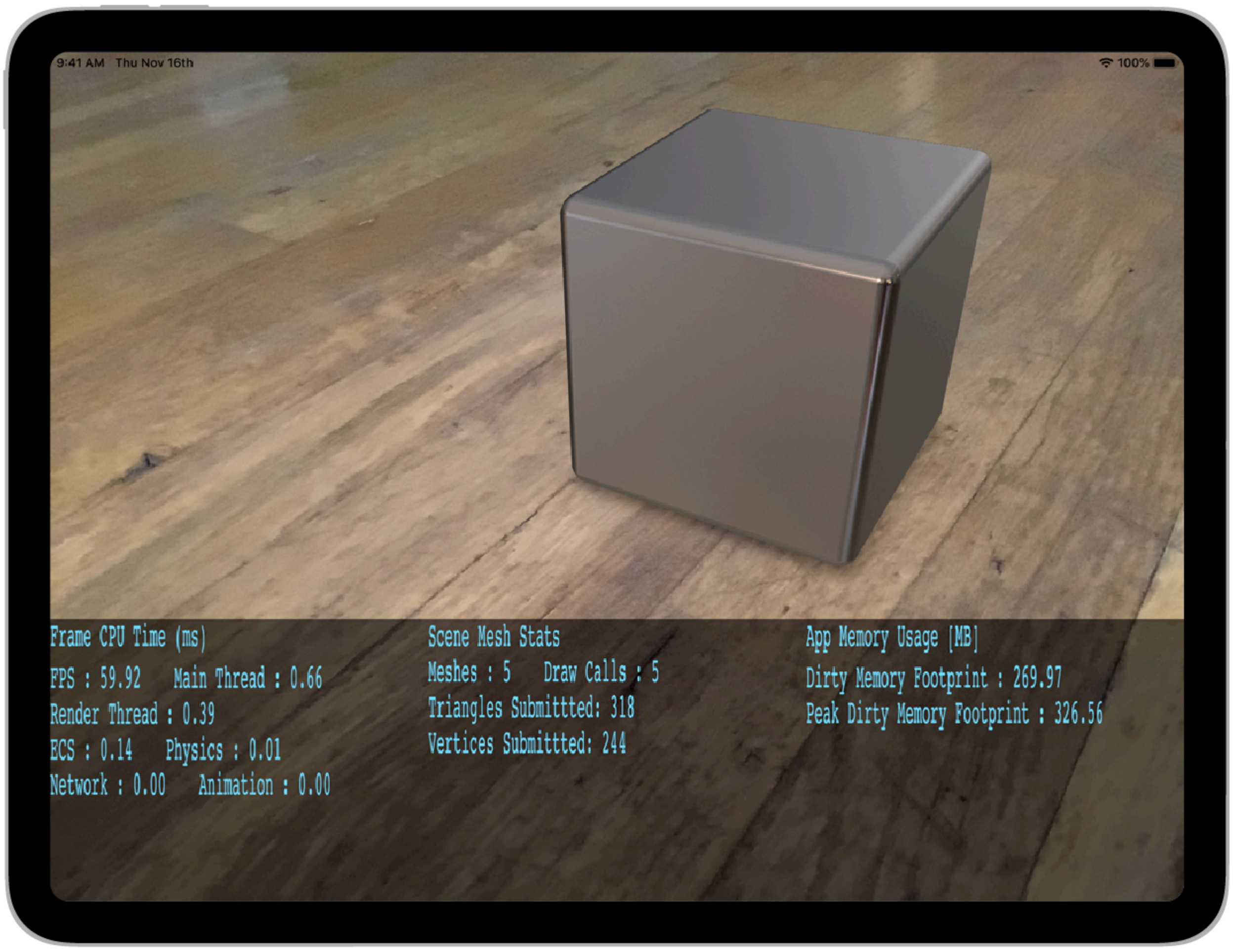
Although RealityKit handles much of the complexity of this system for you, it's still important to optimize your app for performance. Use debugging features built in to RealityKit — along with standard tools like Xcode and Instruments — to pinpoint the causes of reduced frame rate. Then make data-driven adjustments to your assets or to the way you use the framework to improve performance.

Locate performance bottlenecks

To address performance problems, you need data. RealityKit provides a debugging option to collect a basic set of statistics, like CPU utilization, ECS operations, and memory footprint. Add the `showStatistics` option to the `debugOptions` option set of your `ARView`:

```
arView.debugOptions.insert(.showStatistics)
```

As a result, the view draws an overlay that displays statistics, updated in real time.



```
Frame CPU Time (ms)                 Scene Mesh Stats                  App Memory Usage [MB]
FPS : 59.92   Main Thread : 0.66    Meshes : 5   Draw Calls : 5       Dirty Memory Footprint : 269.97
Render Thread : 0.39                Triangles Submittted: 318         Peak Dirty Memory Footprint : 326.56
ECS : 0.14   Physics : 0.01         Vertices Submittted: 244
Network : 0.00   Animation : 0.00
```

RealityKit typically limits the refresh rate — the rate at which the framework renders updates for the screen — to 60 frames per second (fps). This rate limits each of the main and render threads, as well as the GPU, to 16.6 ms to complete all computations per rendered frame. The overlay reports the observed frame rate, as well as the measured main and render thread times. It also provides granular detail about time spent in the main thread, such as when performing tasks like ECS updates and physics calculations. For more information about the visible metrics, see `showStatistics`.

If the main thread consumes more than 16.6 ms, then the app is CPU limited. If not, but the frame rate remains consistently below 60 fps, then the app is probably GPU limited. Use this information and the other data in the overlay to drive changes in your app.

Tip

If you need more detailed data, use tools like Xcode's debug gauges and the Instruments app. Combined with OS signposts, available among the Logging methods of the os framework, you can identify processor utilization with a great deal of precision, as described in Improving your app's performance.

Reduce CPU utilization

For an app that's CPU limited, examine what RealityKit task or tasks — like rendering, ECS, physics, network, and so on — consume the most time during each frame. Also consider your app's custom logic, which contributes to the main thread time. Then target the appropriate areas for optimization, as described in Reducing CPU Utilization in Your RealityKit App.

Reduce GPU utilization

If your app is GPU limited, look for ways to reduce render work. For example, you can reduce the complexity of your content, like textures, meshes, and materials. Alternatively, you can disable or scale back certain render features, like depth of field, shadows, and environmental texturing. For more information, see Reducing GPU Utilization in Your RealityKit App.

Make runtime adjustments for older devices

Be sure to test your app on all supported devices before shipping your app. If you find that older devices can't achieve the full frame rate, experiment with complexity reductions that you can make dynamically. For example:

- Scale back effects, as described in Choose render effects carefully. Depth of field and motion blur tend to be particularly expensive.
- Substitute simpler models with fewer polygons in place of your standard models.
- Reduce the rendering resolution by scaling the `contentScaleFactor` property of the view, whose default value depends on the device:

```
// Capture the default value after you initialize the view.
let defaultScaleFactor = arView.contentScaleFactor

// Scale as needed. For example, here the scale factor is
// set to 75% of the default value.
arView.contentScaleFactor = 0.75 * defaultScaleFactor
```

Determine which adjustments you need for different kinds of hardware, and then choose different code paths based on the hardware you detect at runtime. To learn about identifying the available GPU hardware, see Detecting GPU features and Metal software versions.

Topics

Optimization targets

Reducing CPU Utilization in Your RealityKit App

Target specific CPU metrics with adjustments to your app and its content.

Reducing GPU Utilization in Your RealityKit App

Prevent the GPU from limiting your app's frame rate by reducing the complexity of your render.

See Also

Performance improvements

Reducing GPU Utilization in Your RealityKit App

Prevent the GPU from limiting your app's frame rate by reducing the complexity of your render.

Reducing CPU Utilization in Your RealityKit App

Target specific CPU metrics with adjustments to your app and its content.

{} Construct an immersive environment for visionOS

Build efficient custom worlds for your app.

Passing Metal command objects around your application

Build a system that creates and passes Metal command objects to entities dispatching Metal compute shaders.

`protocol Resource`

A shared resource you use to configure a component, like a material, mesh, or texture.