Article

Reducing the rendering cost of RealityKit content on visionOS

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

Overview

The complexity of the assets and features you use in a `RealityView` have a big impact on the work your app and the render server do to render each frame. On visionOS, the system continuously renders 3D content in response to changes in head position and also incorporates any changes you make through system updates. Performance bottlenecks that prevent timely rendering and responsive feedback interfere with the spatial experience.

To identify any performance bottlenecks, use the RealityKit Trace template to profile your app. The RealityKit Metrics instrument that it includes collects data on 3D mesh rendering, spatial systems, entity commits, animations, physics, and particles effects work. If you encounter performance bottlenecks in your RealityKit scenes, attempt to reduce the overhead in the bottleneck area to improve the rendering and responsiveness of your app. With careful balancing, your app can maintain utility and realism without exhausting the resources available on the system. Minimize overhead in less noticeable areas while prioritizing aspects of your app that provide critical functionality and an engaging experience.

For information on how input propagates through the system and updates content on visionOS, see Understanding the visionOS render pipeline. For more information on using the RealityKit Trace template in Instruments to profile your app, see Analyzing the performance of your visionOS app.

Related sessions from WWDC23

Session 10099: Meet RealityKit Trace

Session 10100: Optimize app power and performance for spatial computing

Session 10274: Create 3D models for Quick Look spatial experiences

Session 10160: Demystify SwiftUI performance

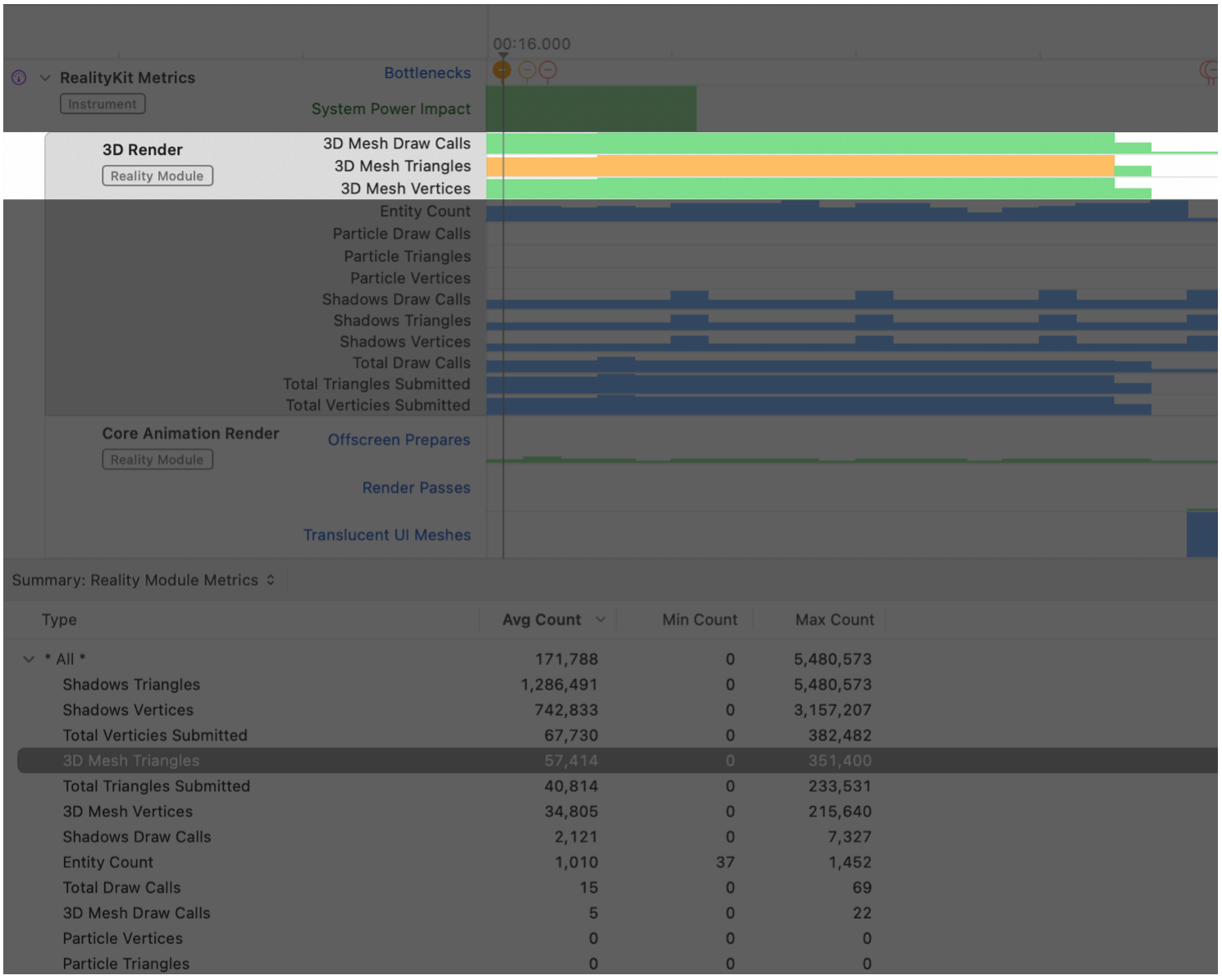
Session 10080: Build spatial experiences with RealityKit

Reduce the complexity of meshes and materials

To reduce the render server's CPU overhead, lower draw call counts. One way to do this is to combine parts of your mesh that share a material. The render server performs a draw call for each individual part and uses the CPU to setup each call it sends to the GPU. While combining parts can improve performance, avoid combining parts that are far apart in your scene or that become too large to stay in the field of view when combined. The system renders the entire mesh for a part that is only partially in the field of view. It doesn't render a part that completely falls outside the field of view.

To reduce both the CPU time the render server spends to create data for the draw calls and rendering work on the GPU, lower vertex and triangle counts, and optimize your meshes for overdraw. *Overdraw* is drawing pixels multiple times to produce the final result.

To identify areas of complex mesh rendering, check the RealityKit Metrics instrument for 3D Render Encoding (CPU), 3D Render GPU (GPU), and GPU Work Stall (CPU) bottlenecks. The instrument collects metrics on the number of 3D Mesh Draw calls, 3D Mesh Triangles, and 3D Mesh Vertices. Expand the view of the RealityKit Metrics instrument in the timeline pane to reveal graphs of this data under the 3D Render section. Select the section to view these metrics under Summary: Reality Module Metrics in the detail pane.



In general, use less than:

- 250 draw calls in the Shared Space (or 500 in a Full Space)
- 250k vertices and 250k triangles in the Shared Space (or 500k in a Full Space)

Find a balance between the other work your app needs to complete and the features of your 3D meshes. Complex meshes and materials in addition to frequent mesh draw calls can create CPU performance bottlenecks. The GPU overhead of visual effects the system applies to 3D content can make the effects of these complexities worse. Under certain conditions, the system applies visual effects automatically, for example, when an app's scenes overlap a scene from another app or a person's surroundings. In these cases, you can turn off other effects, such as grounding shadows. To turn off grounding shadows, toggle the `GroundingShadowComponent` setting in Reality Composer Pro. You can efficiently use Physically Based materials in Reality Composer Pro for smaller, fully opaque content. For larger content or content you choose to make transparent, the environmental lighting these materials use is expensive. Consider using a custom material with an unlit surface and add lighting textures or other less expensive effects to produce a similar effect.

Load assets efficiently

Complex assets take a long time to load, contribute to longer app launches, and can trigger expensive render updates. To reduce asset loading times and their impact:

- Export files from Reality Composer Pro to use in your visionOS project. Reality Composer Pro optimizes the content in these files for fast loading and lower memory costs.
- Minimize the number and size of expensive file types in your assets, such as textures, meshes, audio, and video files. Reality Composer Pro performs texture compression automatically when you use it to export your assets.
- Minimize the number of entities, textures, and primitives that USD models contain and the number of distinct and custom materials they use.
- Preload assets that contain custom materials compiling shaders at runtime.
- Use asynchronous loading APIs to avoid blocking the main thread. This can be especially useful when you need to load multiple assets together.
- Reuse assets as much as possible for entities that can share the same resources. Entities that use the same assets and models can load the file once and share instances.

The complexity of an asset is primarily depends on its size and the amount of sub-assets it contains. When a RealityView loads an asset, your app's process performs the work to load the asset but the system still registers the asset entities with the render server. The RealityKit Metrics instrument might detect a bottleneck, or stall, in the render server while loading an expensive asset.

Minimize updates

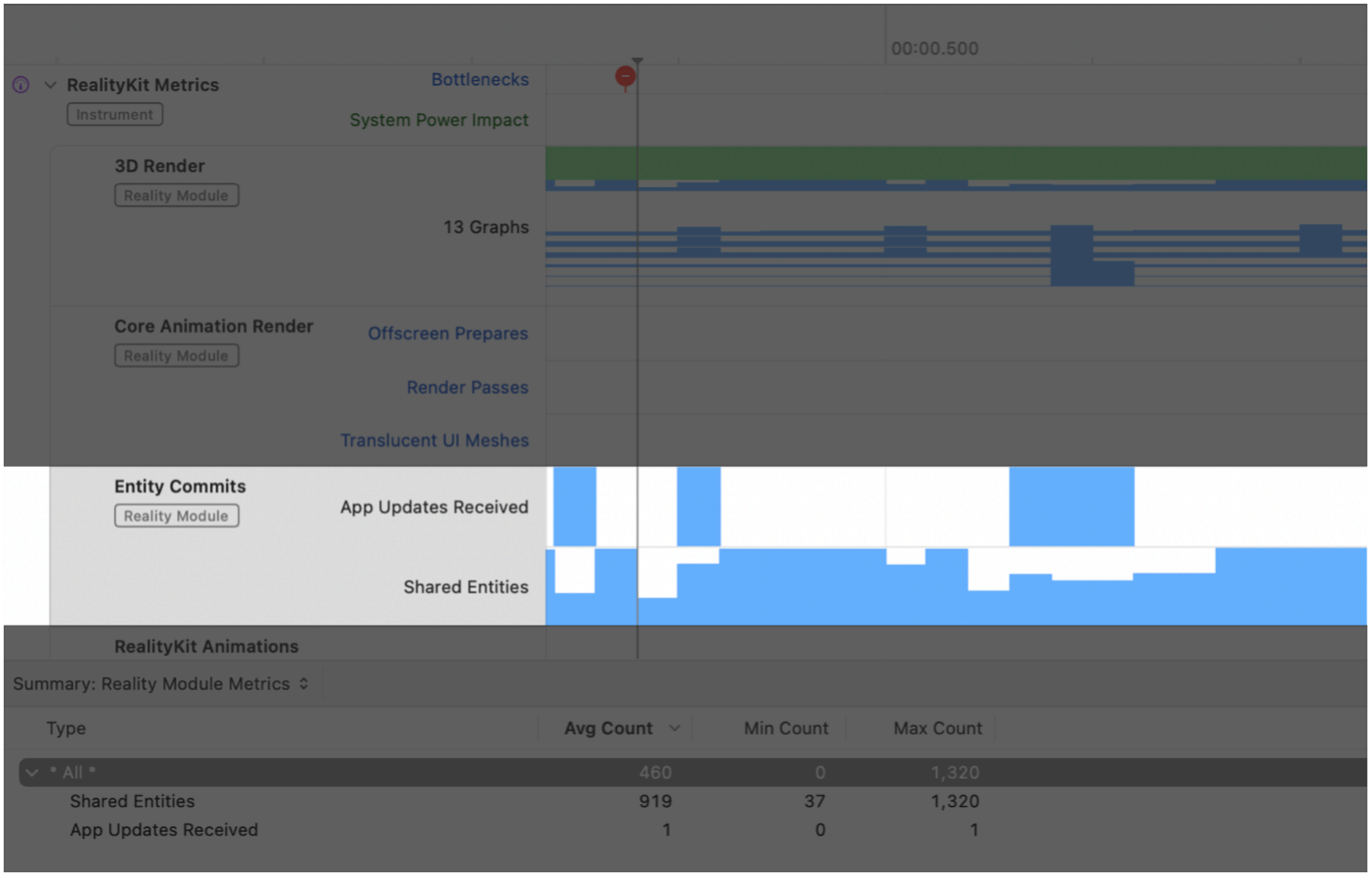
When your custom systems modify entities and audio components, your app uses the CPU to encode the update. Then the render server performs additional work on the CPU to decode it and apply the changes to your content. To reduce the number of updates that your 3D RealityKit scenes make per frame:

- Minimize the amount of entity creation and destruction that your app does per frame, especially for attachments. Create entities in advance, and hide or show them as needed by adding or removing them from the hierarchy or toggling the value of `isEnabled`.
- Perform work in custom systems only when you require it, rather than on every single frame.
- Change properties deliberately and avoid altering them unnecessarily.
- Reduce the number of entities your app updates in response to events and actions. One way to accomplish this is to flatten your mesh entity hierarchies.

- Lower the update rates of code based animations and reduce the number of entities that animations update.
- Avoid triggering excessive SwiftUI redraw when updating RealityKit entities.

> **Note**
>
> Certain actions initiate updates indirectly. For example, a physics simulation might cause transform updates.

To identify areas of frequent and complex updates, check the RealityKit Metrics instrument for Entity Commits (CPU) and Custom RealityKit Systems (CPU) bottlenecks. The instrument collects metrics on the number of app updates the render server receives and the number of entities that it creates and destroys. Expand the view of the RealityKit Metrics instrument in the timeline pane to reveal graphs of this data under the Entity Commits section. Select the section to view a summary of these metrics in the detail pane.



App Updates Received
: The number of updates the render server gets from all apps over a particular interval.

Shared Entities
: The number of entities the render server creates and destroys across frames.

> **Note**
>
> Depending on the type of content, you might see additional bottlenecks that result from these updates. Transform updates, animations, material updates, asset loading, and view hierarchy updates also cause the render server to redraw 3D content.

Optimize animations

Minimize the CPU overhead of your animations and the total number of entities your animation affects.

For transform and material animations, minimize the number of entities you impact to lower the cost your app incurs. For example, you might use a custom RealityKit system to trigger and run these animations and call `move(to:relativeTo:)` or make changes to materials, rotation, scale and transform properties in your app's update loop. Each entity you impact adds to the work your custom systems do and results in additional entity commits between your app and the render server.

For code based animations, synchronize updates with the device's display refresh rate. Inconsistent animation update intervals create a poor visual experience. Avoid creating material animations by generating a sequence of new materials at runtime. Instead, use a single material and animate a set of parameters using `ParameterSet` and `MaterialParameters`. Retain a copy of the material structs in your app to reuse across frames. This approach avoids redundant allocations and unnecessary overhead.

For Skeletal animations, reduce the total number of vertices in mesh geometries, weights per vertex, and separate the animations in content where possible. The render server does most of the work to implement these animations running shader deformers on the GPU. Limiting the cost of these operations is important; their overhead is greater when operating on more complex mesh geometries.

To identify areas of frequent and complex updates, check the RealityKit Metrics instrument for RealityKit Animations (GPU) and RealityKit Animations (CPU) bottlenecks. The RealityKit Metrics instrument collects metrics on number of skeletal animations running. Expand the view of the RealityKit Metrics instrument in the timeline pane to reveal graphs of this data under the RealityKit Animation section. Select the section to view a summary of these metrics in the detail pane.

Reduce collisions

Collisions between objects in the Shared Space result in expensive physics calculations. To reduce the number of collisions and the work the system performs to compute collisions:

- Remove colliders and dynamic rigid bodies from a scene that aren't necessary to produce the experience you want to create.
- Space objects apart or organize objects into groups to avoid clustering. For example, 200 rigid bodies bouncing about in a single box results in more collisions than 50 rigid bodies bouncing around in 4 separate boxes.
- Try different shapes. Choose collision shapes that minimize contact. Simpler shapes require less compute time. Avoid using `generateCollisionShapes(recursive:static:)` when you can pick a simple shape. This function might create a more complex shape to fit a mesh.
- Minimize the number of colliders. Using collider components for tap interactions is expensive when you create a lot of colliders.
- Reduce the number of physics objects stacked on top of one another. Stacked and overlapping physics shapes require more memory to track the overlapping contact pairs.

> **Tip**
>
> Take into account a mesh's structure when you use `generateCollisionShapes(recursive:static:)` to generate physics shapes. This function recursively generates shapes for the meshes, but the shapes might be inefficient for collisions. Set `isStatic` to true to generate static colliders that are more efficient.

To identify areas with high CPU overhead resulting from collisions and physics calculations, check the RealityKit Metrics instrument for RealityKit Physics (CPU) bottlenecks. Expand the RealityKit Metrics instrument in the timeline pane to reveal graphs of this data under the RealityKit Physics section. Select the section to view a summary of this data in the detail pane.

Lower the impact of particle effects

To reduce the overhead of particle systems and the number of draw calls they generate, minimize the the total number of particles you have in your app. Often, you can create a similar effects with fewer particles. Without increasing the total number of particles, try to produce more particles per particle emitter rather than running several particle emitters synchronously with a small number of particles each. Loading particle effects is also an expensive operation, so preload them whenever possible and avoid loading several at the same time.

Experiment with different particle effects, shapes, and shaders when attempting to reduce the overhead of your particles on the GPU. The total number of vertices and triangles for all particles, and their material and transparency, affect the amount of GPU work and overdraw the particles require and the number of pixels your effects take up.

To identify areas with high GPU overhead resulting from particle effects simulation, check the RealityKit Metrics instrument for 3D Render GPU bottlenecks. To identify areas with high CPU overhead, check for Particles Update (CPU) bottlenecks. The instrument collects metrics on the number of Particle Draw Calls, Particle Triangles, and Particle Vertices. Expand the RealityKit Metrics instrument in the timeline pane to reveal graphs of this data under the 3D Render section. Select the section to view a summary of this data in the detail pane.

View scene statistics in Reality Composer Pro

Take advantage of the statistics Reality Composer Pro provides when you use it to author RealityKit scenes. To view this data:

1. Open the project's Reality Composer Pro (`.realitycomposerpro`) file in Reality Composer Pro.
2. To select a scene to inspect, click on its tab at the top of the project window.
3. Choose View > Statistics to display the statistics at the bottom of the window.

The tool provides information about entity counts, physics, animations, particle emitters, materials, lighting effects, and mesh geometry that effect rendering overhead. For more information about Reality Composer Pro, see Composing interactive 3D content with RealityKit and Reality Composer Pro.

See Also

Performance

- 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 your UI on visionOS
  Optimize your 2D user interface rendering on visionOS.
- Understanding the visionOS render pipeline

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