Sample Code

Construct an immersive environment for visionOS

Build efficient custom worlds for your app.

Download

visionOS 1.0+ | Xcode 15.3+

Overview

You can implement immersive environments for your app that people can fade in and out using the Digital Crown, just like the provided system environments. However, custom immersive environments can cause performance and thermal problems if you're not careful about how you build them.

This article describes ways to address these potential problems, and the sample provides a demonstration of some of these methods in action.

Avoid very dark or light environments

If possible, avoid creating environments that are very dark or very light. With very dark environments, you may notice banding artifacts. You can mitigate this effect to some extent by using 16-bit textures instead of 8-bit textures, but doing so adds nontrivial overhead because Reality Composer Pro doesn't compress the texture. You may also be able to implement dithering in your shader to reduce banding.

With very bright textures, the lightest parts of your environment may look clipped when viewed on a device.

This sample's garden scene is a well-balanced environment.

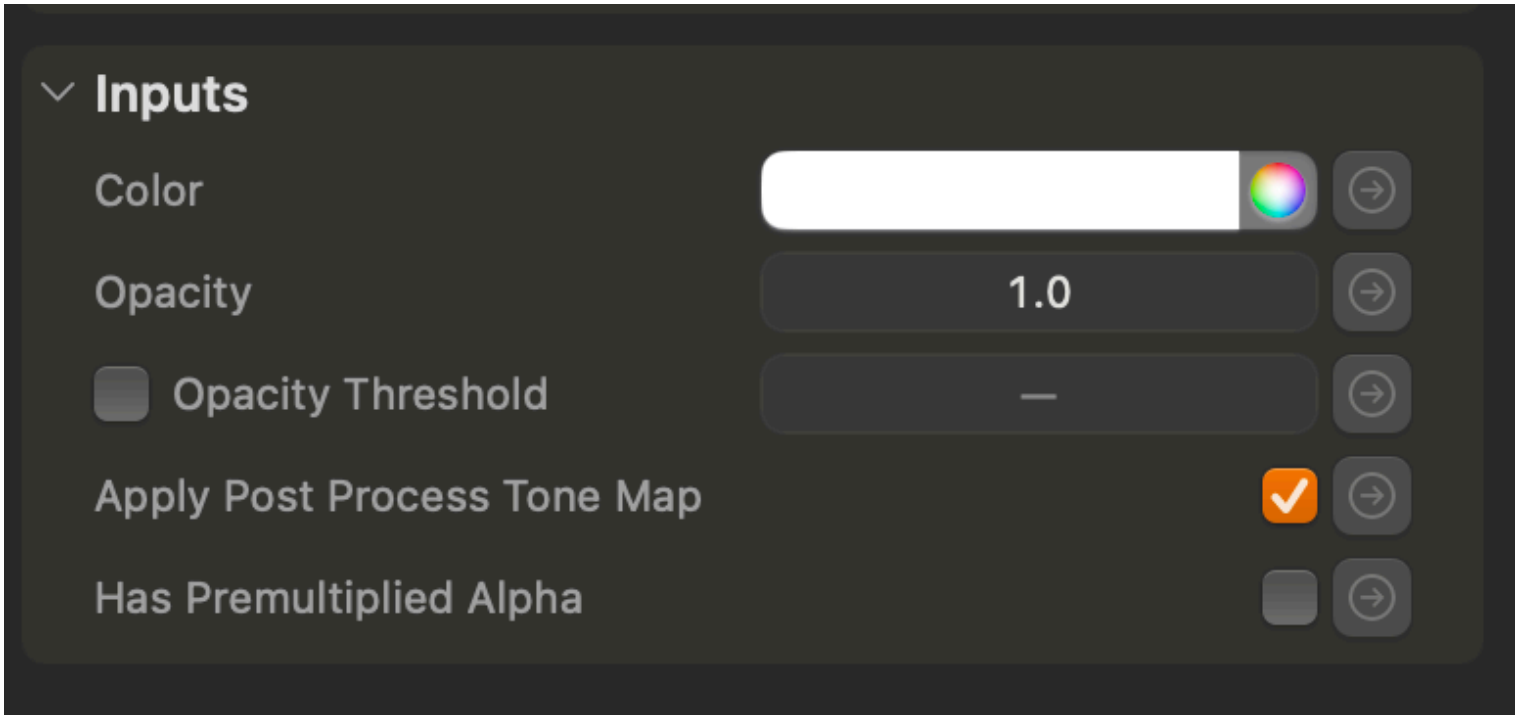
Minimize shader complexity

All shaders aren't created equal. Physically based rendering (PBR) materials are considerably more complex than unlit materials, which just map a texture or color onto an object without requiring lighting calculations. Most 3D modeling applications provide a way to bake lighting into a texture, creating the illusion of a lighted entity. Baking lighting into your model's textures isn't always possible, but when you use an unlit material instead of a PBR material, you improve your environment's performance. The sample app uses unlit materials as much as possible to limit pixel shader cost.

When creating `UnlitMaterial`, pass `false` to the intializer's `applyPostProcessToneMap` parameter for better performance and more accurate colors; for example:

```
let material = UnlitMaterial(color: .white,
                             applyPostProcessToneMap: false)
```

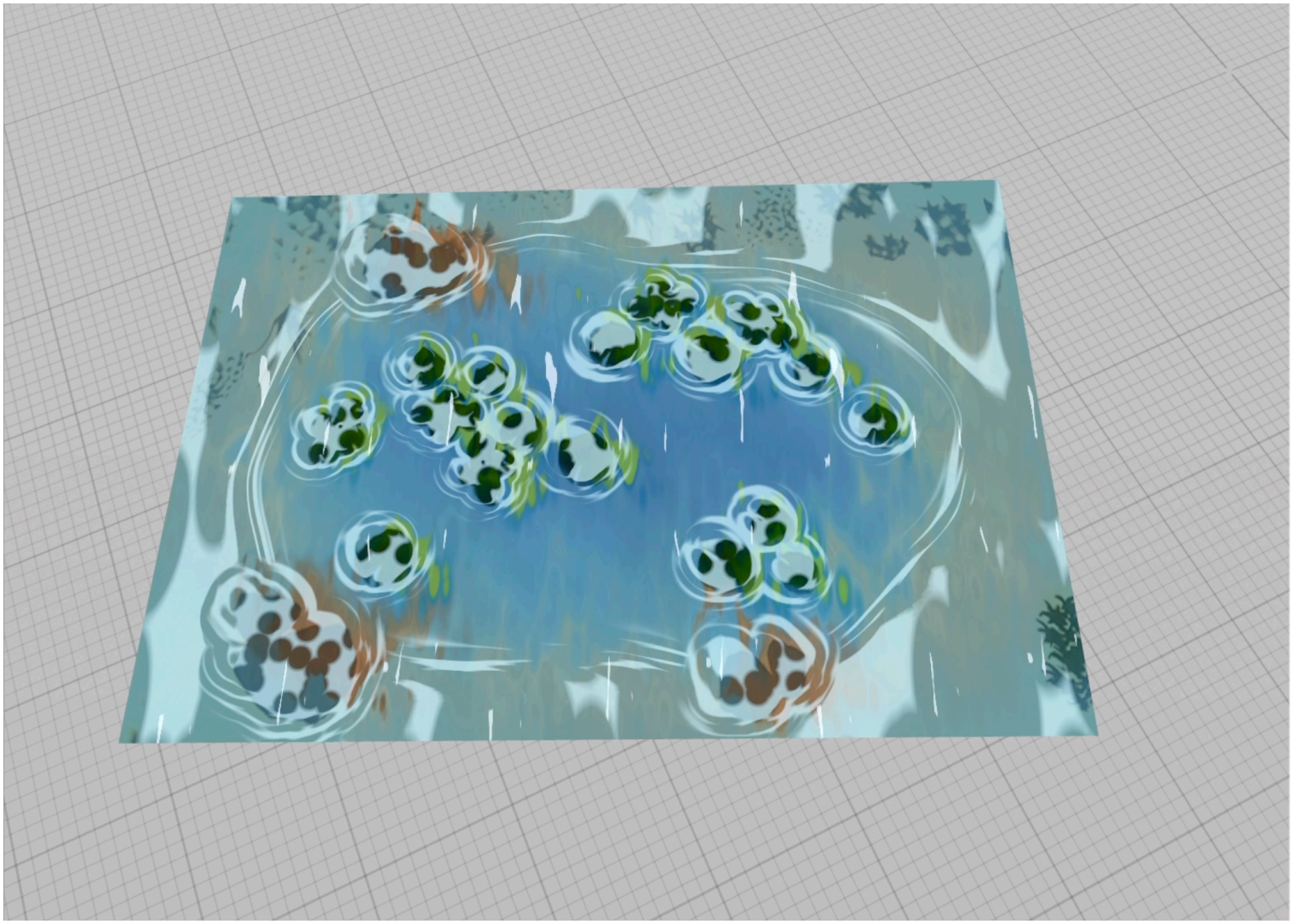
You can also set `applyPostProcessToneMap` to `false` in Reality Composer Pro's inspector by selecting a material in the hierarchy browser, and then selecting that material's UnlitSurface node in the Shader Graph.



Certain other settings can affect the complexity and performance of the shaders that RealityKit generates. For example, enabling transparency can add nontrivial overhead to the generated shader, as can using trilinear filtering on textures. For more information, see Improving the Performance of a RealityKit App.

Use perpendicular geometry

Try to use geometry that's perpendicular to the viewer. For objects more than a moderate distance from the user, consider mapping a texture containing the rendered object and map it to a rectangle or other two-dimensional polygon using an unlit shader. You can also use normal maps and use UV animation on your textures to create more convincing flat objects, like the pond shown in this video. This sample has the pond located behind you. In the visonOS Simulator, once you have entered the immersive space, use the I/O > Set immersion to > 100% and then turn around to see the pond.

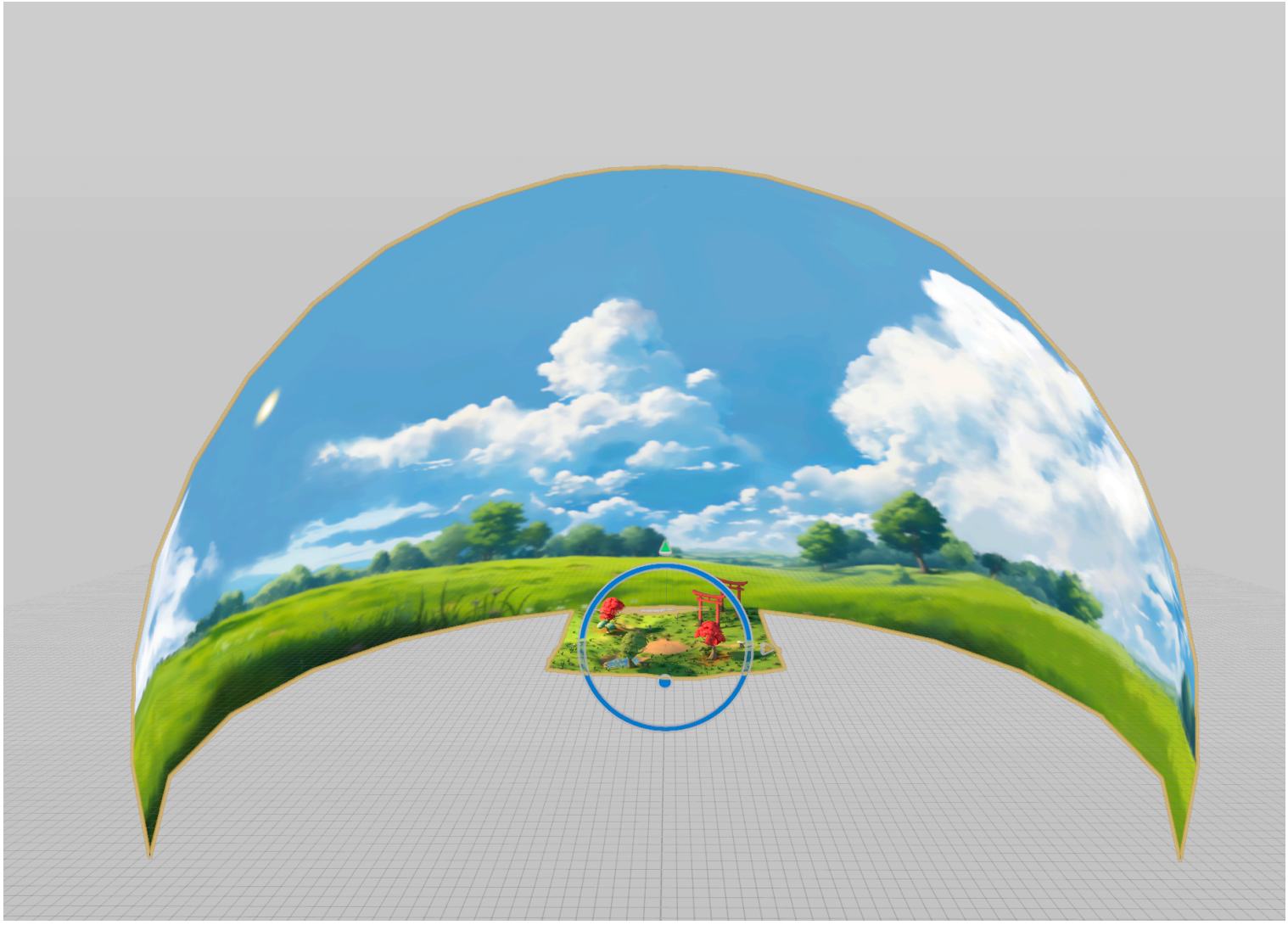


Play ⊙

Use a sky dome

To create your environment's sky and distant objects, place a large dome over the viewer with the dome's normals pointing inward, and map an environment texture onto it with an unlit material, as shown in this sample's garden asset. If your environment has no floor or ground, use a sphere instead of a dome.

Beyond a certain distance, a viewer can't tell the difference between an entity and a rendered version of that entity mapped to the sky dome, but the texture-mapped version adds no vertices to the scene other than those needed to create the sky dome.



Use geometry efficiently

Although the geometry detail in the stylized content of the sample isn't dense enough to require much optimization, vertex count is one of the biggest sources of performance problems in rendered 3D scenes, so use the smallest number of vertices you need to achieve the effect that you want. Objects that are further away from from the viewer generally need fewer vertices than those closer to the camera.

Most 3D-modeling programs provide a way to capture fine surface details for a higher-resolution version of the same model using a texture instead of increasing the number of polygons. This process is referred to as *baking*, and the output is called a *normal map*. A normal map works by storing surface normals, which are vectors perpendicular to the surface of the model, from the higher resolution version of the object, by storing the vectors' X, Y, and Z values as the R, G, and B components of the corresponding pixel in the UV-mapped image. Apply baked normal maps to a lower-resolution entity by assigning it to the `normal` property of the material or shader node.

Using normal maps does have a performance cost, however. Not as much as the extra vertices it would take to show fine details, but not neglible either. Avoid using them on large or distant objects and use them selectively when building immersive environments. Most 3D programs provide a way to bake other kinds of maps besides normal maps. For larger or more distant

objects, if you need more detail, consider baking the details into the emission map and using an unlit shader.

Remove geometry the user can't see, including vertices inside an object or behind one, or on the back side of static objects. Keep in mind that an environment is viewed from near the origin of the scene. If somebody moves more than about a meter from their starting position, visionOS automatically turns immersive mode off, so you don't need to worry about how environment objects look from other vantage points.

Try to avoid using long, thin triangles in your models. To create items like wires and chain link fences, either break up the geometry into multiple smaller triangles, or consider using an alpha map, which is a texture you apply to your model that identifies which parts of the object are transparent. You can use an alpha map to "cut out" thin shapes from a simple, larger polygon rather than building them from vertices.

Reduce draw calls

Each model entity in your scene generates one draw call per material for noninstanced geometry. On the other hand, with instanced entities, RealityKit only generates a single draw call for each material used on the original instanced entity. As a result, using instanced entities can reduces the number of draw calls your app makes.

Sharing textures between entities by using *texture atlases*, which are textures that contain images for multiple materials, also reduces your app's draw calls, as can combining multiple models into a single entity with shared materials. When combining model entities, be careful not to make the combined entities too large; if the entities are too large, they won't be culled during frustum culling, when objects that are completely off-camera are removed from the rendering process — resulting in no draw calls at all. If any part of the combined entity is visible on screen, every material on that entity generates a draw call, even materials that are completely off screen.

Generate efficient textures

The way you create the textures for your model entities can have a significant impact on shader performance. For best results, use 8 bits per channel, sRGB-encoded textures that use the Display P3 color space. Select the smallest texture size you can without sacrificing quality. Also, use UV map geometry so the entity is consistently sized based on the entity's relative distance to the viewer. In other words, create texture maps so that objects closer to the viewer use more UV space than distant objects of roughly the same size.

Provide a custom, image-based lighting texture

To control the lighting of your scene, provide an image-based lighting (IBL) texture. For best results, use a 1024x512 pixel HDRI image saved as an `.exr` file. Load the image, then use it to create an `ImageBasedLightComponent` on the root node of your `RealityView`, as shown below:

```
  if let resource = try? EnvironmentResource.generate(fromEquirectangular: myHDRImag
    rootNode.components.set(ImageBasedLightComponent(
        environment: resource,
        intensityExponent: overallIntensityExponent
    ))
}
```

When using IBL in your scene, add an `ImageBasedLightReceiverComponent` to any entity in the scene using PBR materials, such as `PhysicallyBasedMaterial`, `SimpleMaterial`, or materials created in Reality Composer's Shader Graph of type `Physically Based`, or of type `Custom` that uses a `MaterialXPreviewSurface` shader node. If you don't provide an IBL texture when using PBR materials, those materials render using RealityKit's default IBL texture when in immersive mode.

Instrument your environment

Use the RealityKit Trace tool in Instruments to identify performance bottlenecks caused by your environment. With your project open in Xcode, select Product -> Analyze and choose the RealityKit Trace instrument. Click the red record button on the left side of the toolbar to start your app running. Use your app normally for a few minutes, then hit the same button again to stop recording. That button now shows a white square instead of the red circle to indicate the recording is in process. It takes a little bit of time for Instrument to process the recorded data after you stop it. For more information on the RealityKit Trace tool, see the Meet RealityKit Trace video from WWDC23.

Once Instruments finishes processing, expand the tool called RealityKit Metrics (1), and select the Reality Module called 3D Render (2). Look on the left side of the window and confirm that you can see Summary: Reality Module Metrics. The metrics in the bottom pane (3) provide information about the work your app's shaders are doing behind the scenes.

The three most important metrics to keep an eye on are Entity Count, 3D Mesh Draw Calls, and 3D Mesh Vertices. It's important to make sure you leave enough compute power for the rest of your app, so try to keep your environment's maximum value for these three metrics to the following values.

Metric	Target Max
Entity Count	< 20
3D Mesh Draw Calls	< 100
3D Mesh Vertices	< 100000

See Also

Related articles

Improving the Performance of a RealityKit App

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

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.

Related videos

Optimize your custom environments for visionOS