Hybrid Mono Rendering in UE4 and Unity

Written by: Rémi Palandri and Simon Green • Oct 1, 2016

TL;DR: *In the quest for faster Gear VR performance, Oculus has explored monoscopic rendering, where close-up content is stereoscopic (rendered once per eye) but far-away content is rendered only once. Look for a sample in the next release of our Sample Framework for Unity. We're also sharing experimental changes for UE4's Gear VR renderer that you can try out in your own application today: https://github.com/Oculus-VR/UnrealEngine/tree/4.12-gearmono.*

Designing games and experiences for Gear VR is challenging. Due to the constraints of mobile rendering, both triangle counts and shaders have to be carefully optimized. One expensive part of VR rendering is the fact that a different view has to be rendered for each eye. Stereoscopic rendering gives users a sense of depth due to "binocular parallax"—the right and left eyes see objects from slightly different positions and the brain interprets that difference as depth.

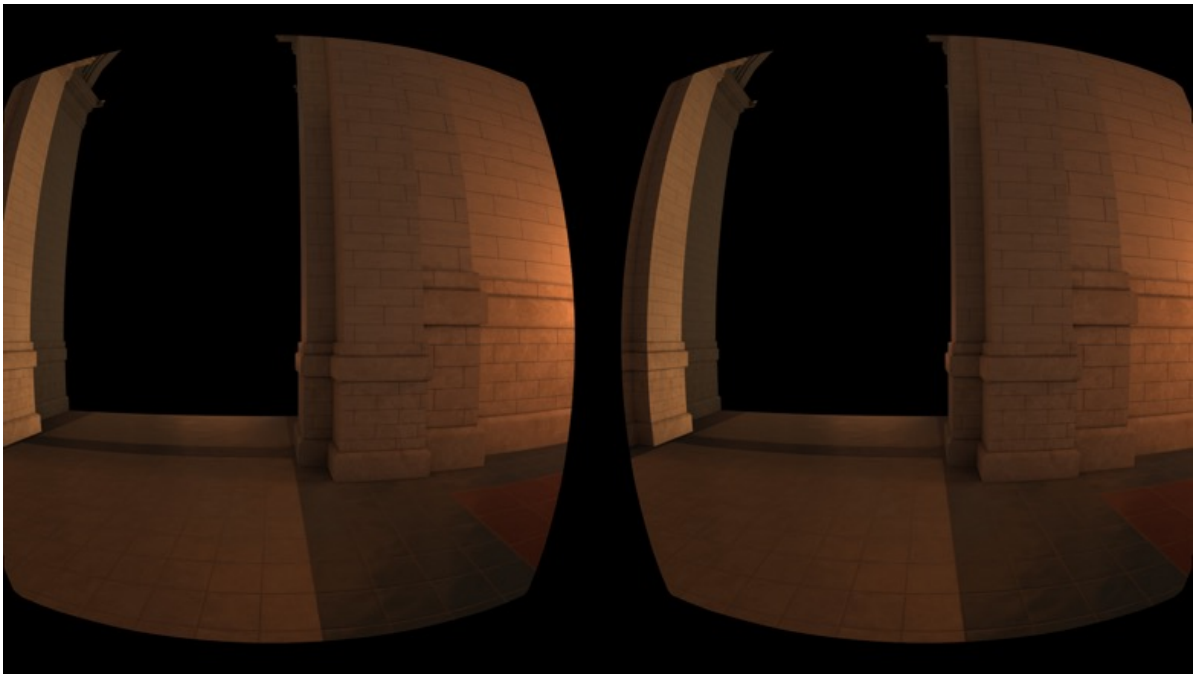
That difference, while significant on nearby objects, becomes smaller and smaller as objects move farther away. The distance between the two eyes—and the generated pixels—becomes insignificant compared to the object's relative position. To take advantage of this, we wrote a Unity sample and modified the UE4 Mobile Forward renderer to render close objects in stereo (once for each eye), and far objects in mono (once for both eyes). Here, we'll explain the choices we made for monoscopic rendering and the related trade-offs.

UE4 Mobile Renderer

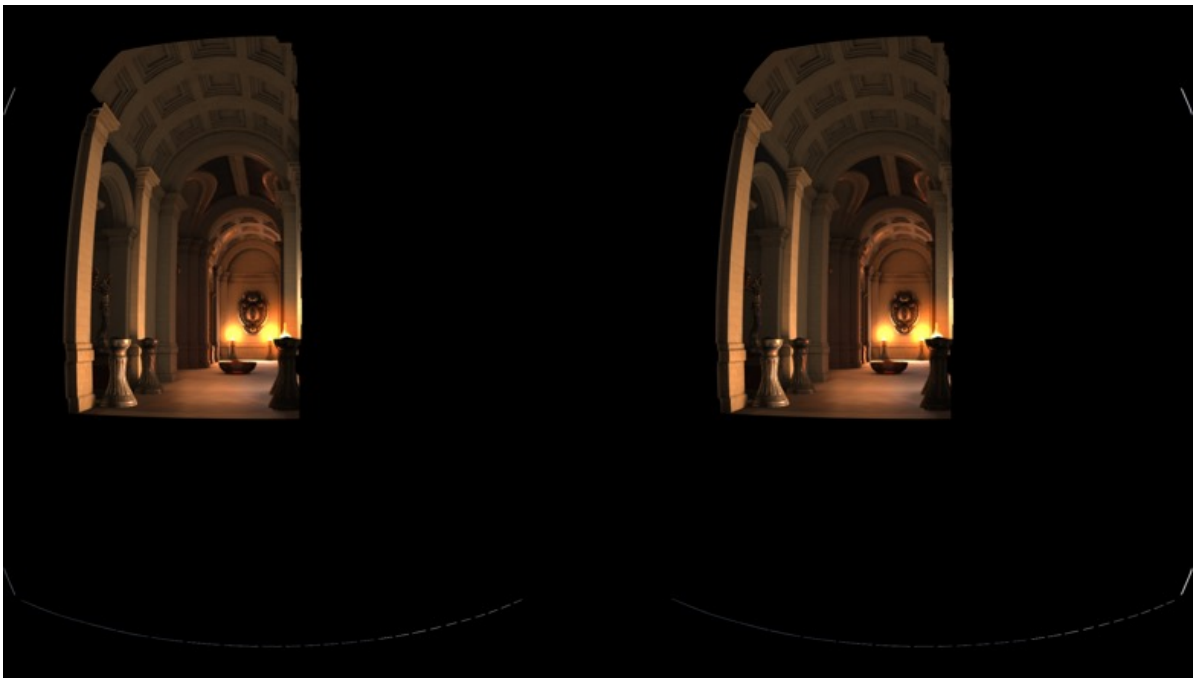
We place a third monoscopic camera between the left and right cameras. We place all three cameras on the same plane to make their perspectives match. On Gear VR, the eye cameras have symmetric frusta , so the monoscopic camera shares the same projection matrix as the stereoscopic cameras. On Rift, the eye cameras are asymmetric (the inward angle towards the nose is actually less than the outward angle), so the monoscopic camera's frustum is the union of both left and right frusta. This covers all objects seen by both eyes, but makes the mono render target slightly bigger than the per-eye render targets.

We add a split plane to choose whether content will be rendered using the stereo cameras or the monoscopic camera. We found that a distance of about 10 meters worked well as a default value. That distance can be modified in-game at any frame and is configurable in the Editor in *World Settings / VR / Mono Culling Distance*.

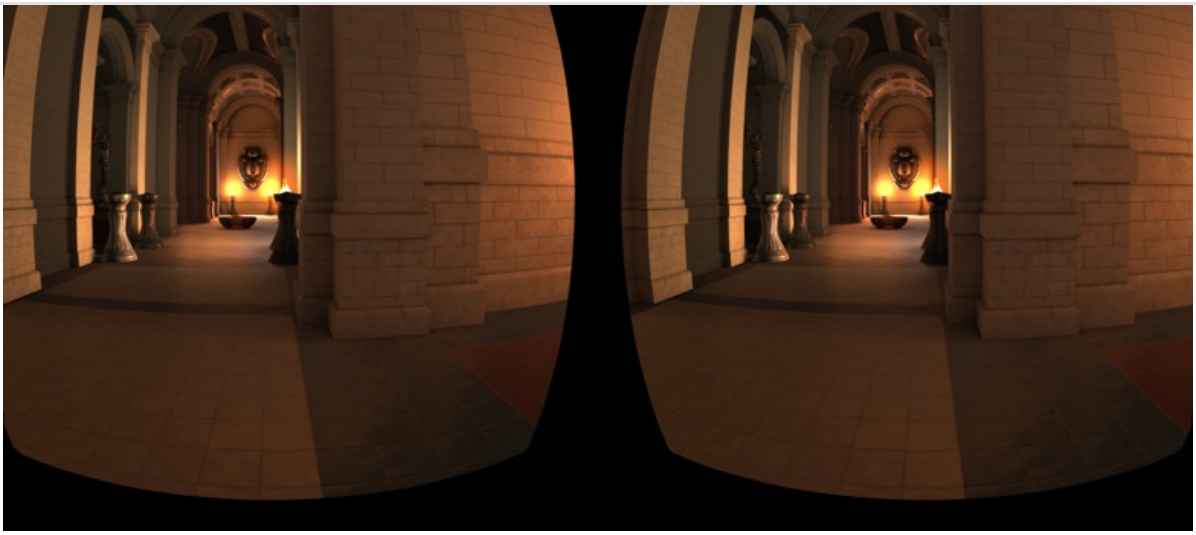
2. Shift and combine the output to create a monoscopic occlusion mask, which pre-populates the monoscopic depth buffer.
3. Render non-transparent content with the monoscopic camera.
4. Composite the monoscopic camera's result into the stereo buffers.
5. Render all transparent content and perform all post processing in stereo.



Result of step 1, stereo-only buffer with close depth culling.



Result of step 3, computed for one eye only but composited in both for the screenshot.



Result of step 4, composition of 3 into 1 with close depth culling.

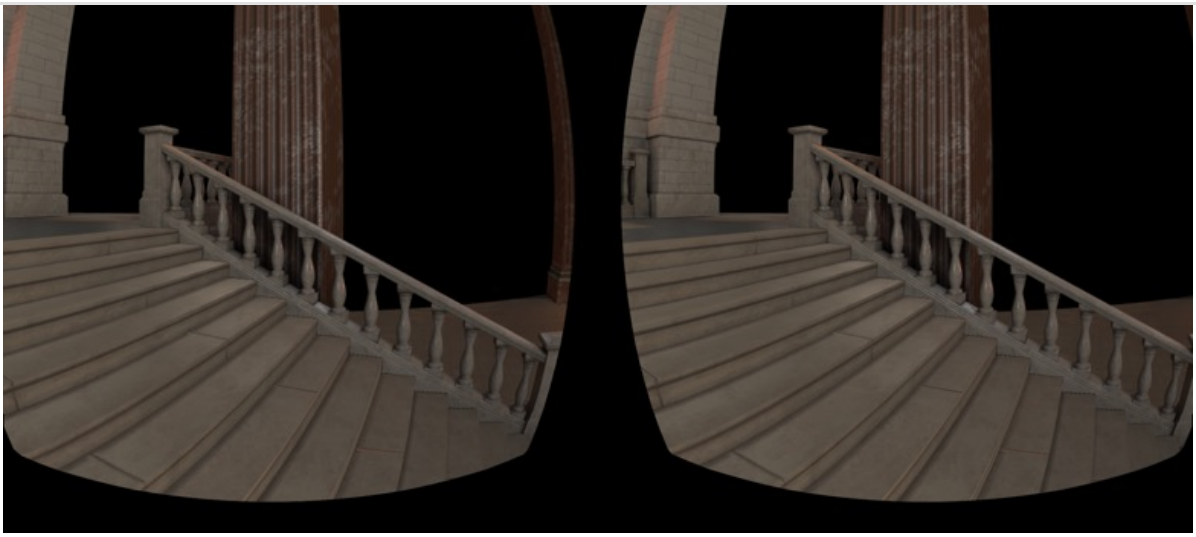
Interfacing the cameras

To separate what content gets rendered in stereo and mono, we went with a depth buffer approach: all pixels after a certain predefined depth will be discarded in the stereo view by clearing the stereo depth buffer to that depth. The monoscopic projection's near plane is also initialized at that depth, to discard fragments that are already rendered in stereo.

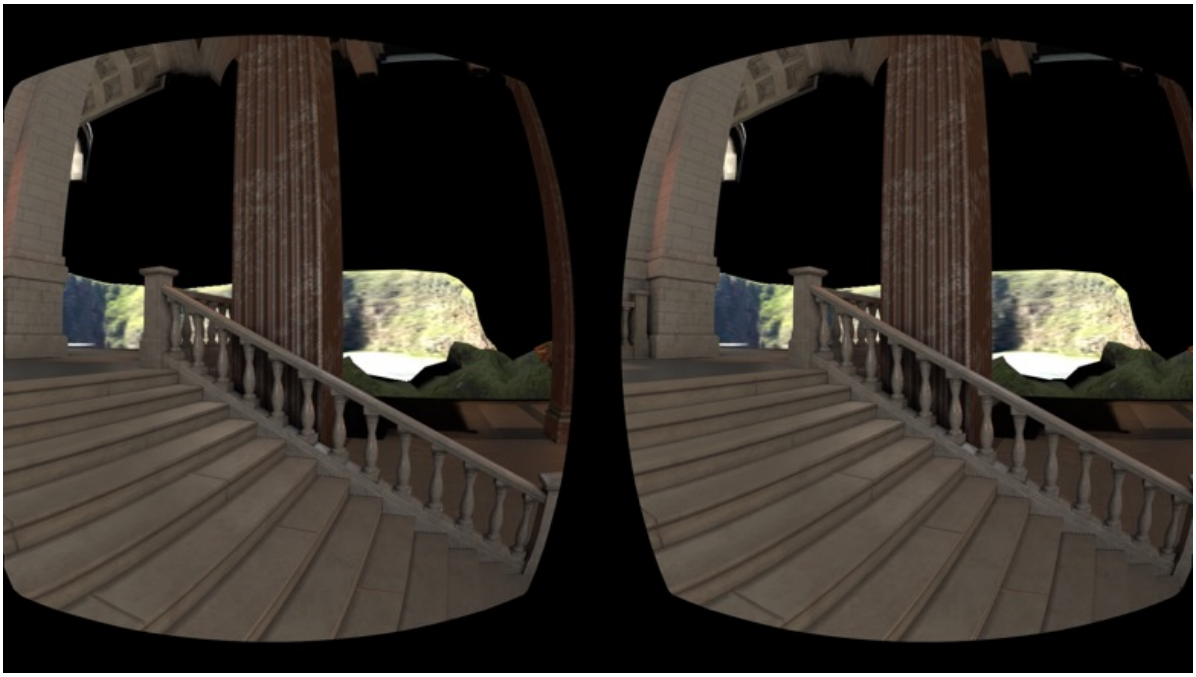
This approach permits us to have a clear depth ordering between stereo pixels and mono pixels: we know all stereo pixels have a depth that is strictly less than all mono pixels. This allows us to avoid an expensive depth comparison during the composition stage. No pixels get rendered in stereo that are further than this plane, minimizing the number of pixel shader calls.

The main drawback of this pixel-based plane approach, compared to an object-based approach, is the number of draw calls: objects that go through the split plane have to be drawn in both stereo and mono, even if no pixel is drawn twice. It is also non-trivial, using normal frustum culling techniques, to minimize the number of draw calls to the stereo buffers, even with a very close far plane in the frustum culling: a far object with a big bounding box, like an environment cubemap, will always be drawn through frustum culling as its bounding box will intersect the camera frustum even if none of its pixels are displayed due to the short far plane.

To avoid such cases, we incorporated a way to manually flag objects in our monoscopic rendering engine to never be rendered in the stereo buffers. To know which objects to flag, we also added two rendering modes: one that only displays the stereo buffers without monoscopic compositing, and one that shows the stereo buffers with the far plane set to the mono/stereo clip plane for frustum culling, but not depth testing. In short: everything getting rendered to the second image but not the first one gets draw called but doesn't show up due to the short far plan.



Stereo-only buffer.



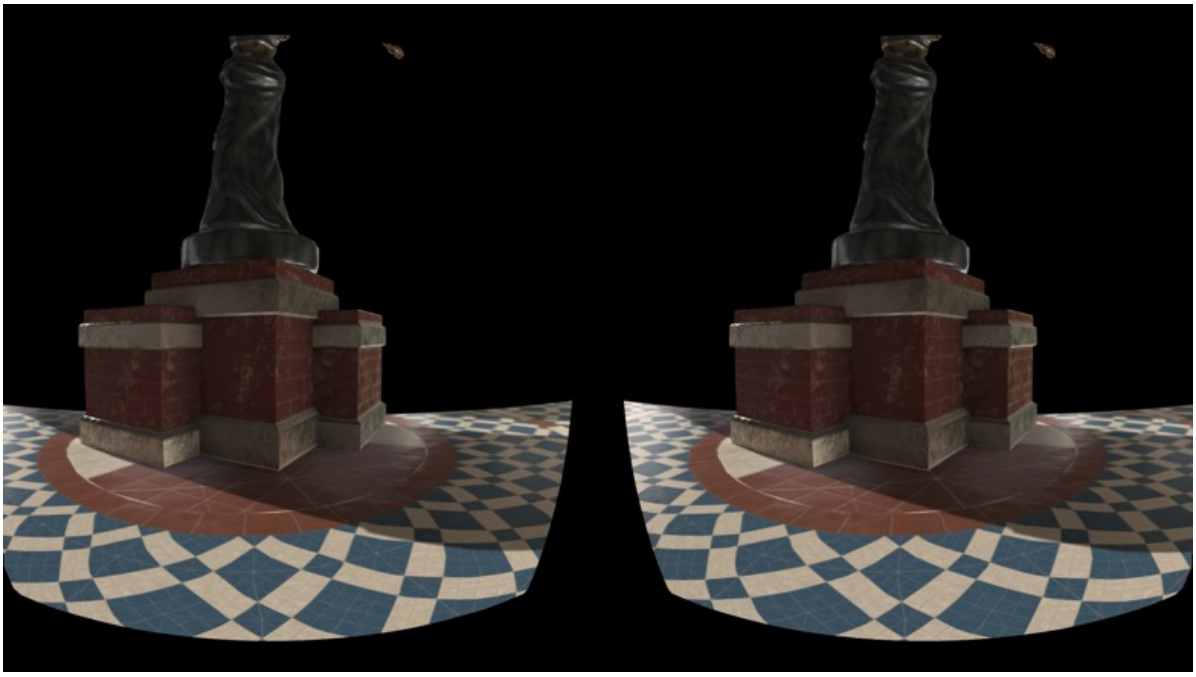
Draw calls submitted to the stereo-only buffer.

In this example, we can notice that the far away terrain passes the frustum culling test due to its big bounding sphere, but shouldn't be rendered because its pixels are way further than the 30 ft depth split plane: we should flag it to force mono-only render in order to gain precious draw calls. That flag is in the rendering section of the object details in the UE4 Editor, named "Force Mono."

Avoiding over-rendering

One of the main issues with monoscopic rendering is that its goal to minimize pixel shader work can be compromised by the lack of occlusion between the stereo and mono layers. Since we're not rendering the objects close to the camera in the monoscopic view, pixels that will not be visible after composition are still shaded in the monoscopic buffer as nothing is occluding them there. To avoid this, we chose

camera. The result is shown below: the huge pillar in the stereo views is completely masked in the mono buffer, which avoids all those pixels being rendered.



Stereo-only buffer.

Mono-only buffer after composition.

Compositing

To avoid discontinuities between the stereoscopic buffers and the monoscopic buffers at the split plane, we calculate the difference of the projection of a 3D point on that plane onto the monoscopic camera and onto the left and right stereoscopic cameras. That difference is minimal: two pixels at a distance of 30 ft for a resolution of 1024 pixels with a 90-degree field of view. We still take it into account to offset

Stereo-mono transition without offset.

With offset.

As we have a clear depth ordering, all monoscopic pixels being behind all stereoscopic pixels, we then just composite the monoscopic pixels if nothing has been written in the stereoscopic buffer at that position by using (1, 1-dest_alpha) as the blending function with a full-screen compositing pass.

Results

The benefits of monoscopic rendering are very scene-dependent, but the results can be impressive on forward rendering pipelines. On Epic's SunTemple sample, we were able to get from 45ms frames to 34ms frames consistently—a 25% decrease in rendering times. Even inside the temple, where objects are both close and far, there was no noticeable drop in quality or depth perception. Similar wins were had

However, scenes that have lots of expensive close-up content can see a performance decrease while using monoscopic rendering due to the fixed costs of rendering a third view and the monoscopic passes of depth masking and composition. As activating and deactivating monoscopic mode doesn't introduce stuttering or frame drops, it can be only turned on when the scenes are favorable to it.

For a few reasons, we found monoscopic rendering less helpful for UE4's deferred renderer. First, deferred lighting uses lots of screen-space effects that must be done in stereo, after composition. Many deferred apps are also bandwidth-bound, not pixel-shader bound, so adding another render target and composition pass only made the problem worse. For that reason, we implemented monoscopic rendering on UE4 on their forward renderer that currently is only used on mobile.

This work uses a modified version of UE 4.12 that is not directly supported by Oculus or Epic, but we encourage you to try it out. We are also working with Epic towards a potential future implementation on the main version of UE4.

Unity Sample

The Unity implementation of hybrid mono rendering follows the same basic approach as the UE4 version, but is implemented entirely using C# scripts and shaders.

Two cameras are used—one which renders the nearby scene in stereo, and one which renders the far scene in mono (only once). The near and far clip planes are used to restrict rendering to the appropriate depth range, and we rely on Unity's view frustum culling to skip draw calls for objects outside the frustums.

The basic procedure is:

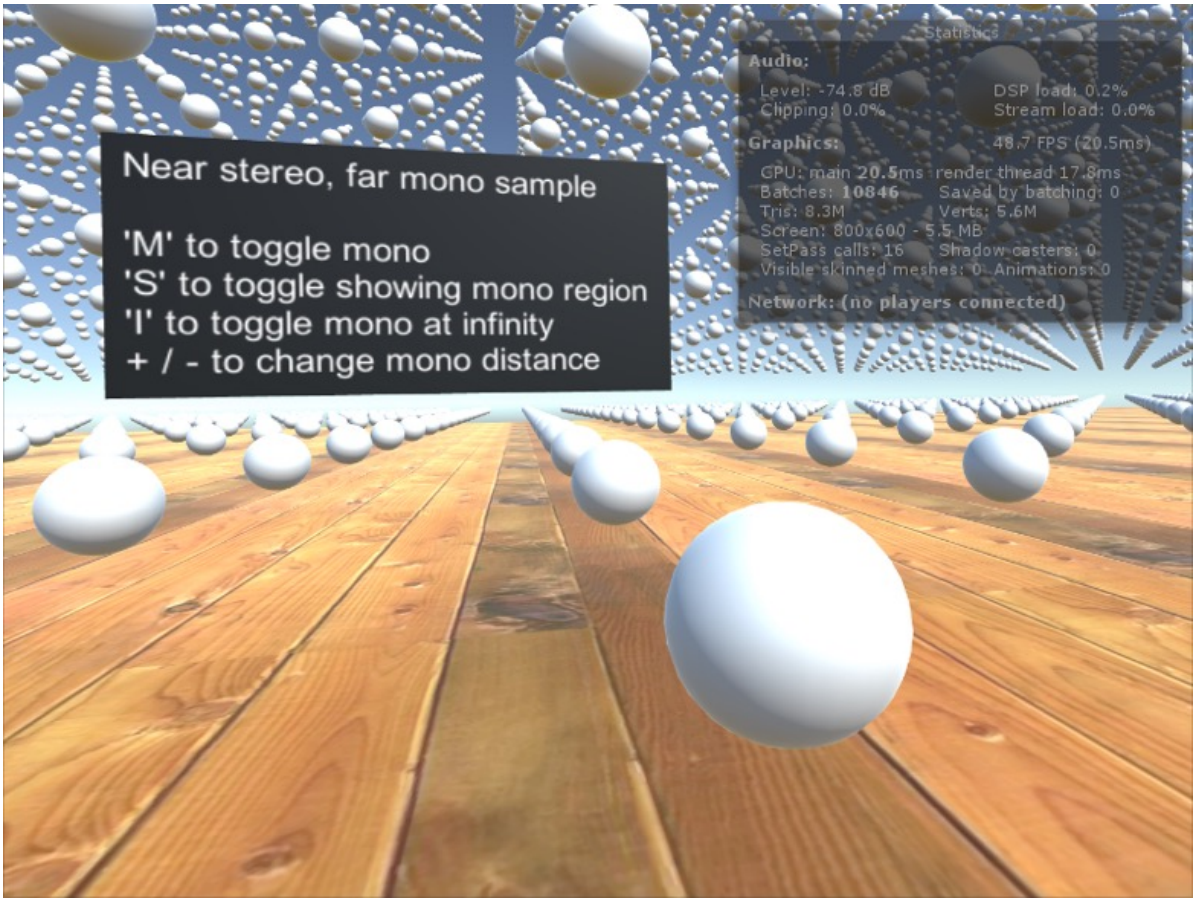
- Render distant parts of scene to texture (mono camera).
- Render close parts of scene in stereo—once for each eye (left and right stereo cameras).
- For each eye, composite the stereo eye image on top of the mono image using a Unity image effect.

The mono render is achieved by setting the camera "Target Eye" to "Left" rather than "Both." The camera object is offset so that it is actually rendered from the center eye.

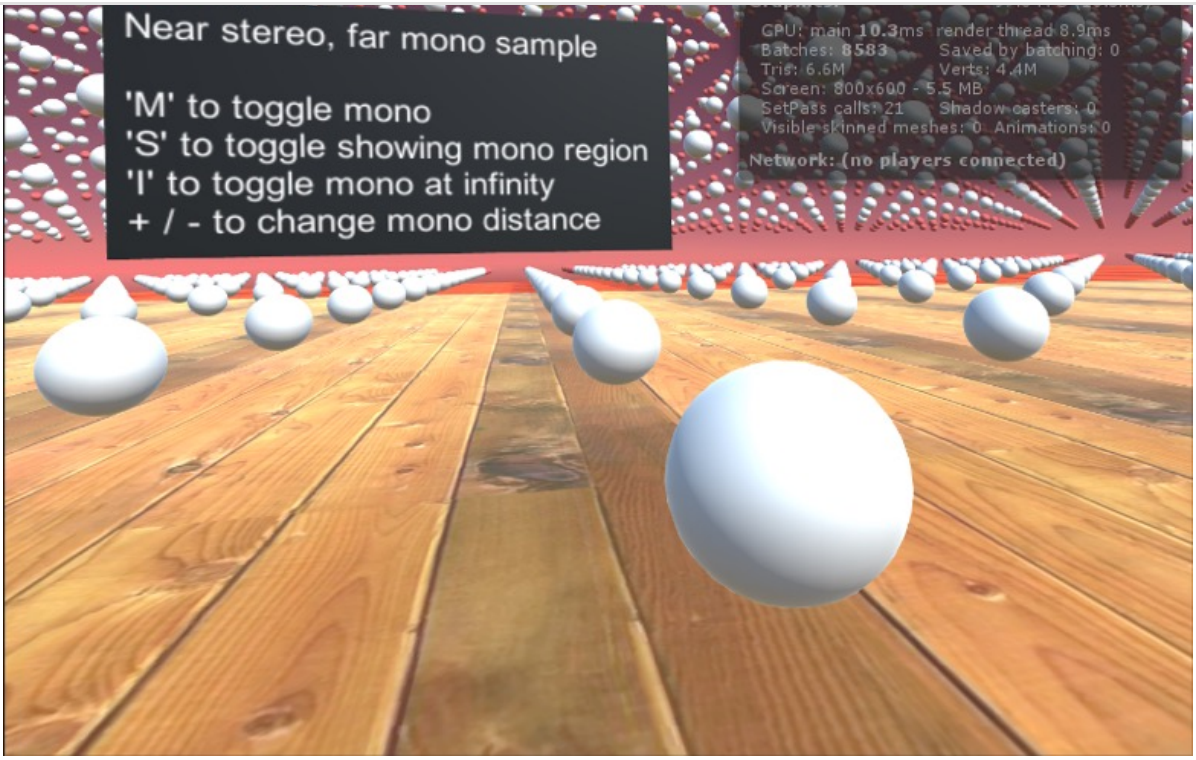
This is all wrapped up in a prefab that can be used as a replacement for the Oculus camera rig.

Meta Quest

The images below show the results of hybrid stereo / mono rendering on a synthetic test scene with a large number of sphere objects.



Mono disabled—10846 draw calls, 48.7 FPS.



Hybrid stereo / mono (mono region shown in red)—8583 draw calls, 97.7 FPS.

This is obviously a contrived scene, but it demonstrates the best-case scenario. With the mono distance set to 10 meters, it is almost impossible to tell the difference between full stereo and hybrid stereo / mono.

We are working on making this into a built-in feature in Unity, which will make it easier to use for all developers.

We are continuing to research other techniques for optimizing stereo rendering in all engines.

Meta Quest

Forums