# **Android Development**

### **Target device successfully saved**

Your target device is set to: Oculus Quest

This section provides guidelines to help your Unity Android app performance.

Good performance is critical for all VR applications, but the limitations inherent to standalone development warrant special consideration.

## Other Resources

Related resources:

* [Performance Targets](https://developer.oculus.com/documentation/unity/unity-perf/" \l "unity-perf-targets" \o "Before debugging performance problems, establish clear targets to use as a baseline for calibrating your performance.)

## Rendering Optimization

This section describes recommended targets and settings for Android projects.

Be conservative on performance from the start.

* Keep draw calls down.
* Be mindful of texture usage and bandwidth.
* Keep geometric complexity to a minimum.
* Be mindful of fillrate.

### **Reducing Draw Calls**

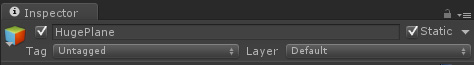
Keep the total number of draw calls to a minimum. A conservative target would be less than ****100 draw calls per frame****.

Unity provides several built-in features to help reduce draw calls such as batching and culling.

### **Draw Call Batching**

Unity attempts to combine objects at runtime and draw them in a single draw call. This helps reduce overhead on the CPU. There are two types of draw call batching: Static and Dynamic.

Static batching is used for objects that will not move, rotate or scale, and must be set explicitly per object. To mark an object static, select the Static checkbox in the object Inspector.



Dynamic batching is used for moving objects and is applied automatically when objects meet certain criteria, such as sharing the same material, not using real-time shadows, or not using multipass shaders. More information on dynamic batching criteria may be found here: [https://docs.unity3d.com/Documentation/Manual/DrawCallBatching.html](https://l.facebook.com/l.php?u=https://docs.unity3d.com/Documentation/Manual/DrawCallBatching.html&h=AT3pHRDD--xkig8fTRe4qeyW9etETf_LCEb5RrNLWx0bzEHfCPMYiDzi78ivqZOazcpT8RLYvNgY13li0OCp2DTe3Ph7lDnHHz_oXg0rwywBRReZ_Jb7doc_nAIJ7J3eck0-7Ruue27Qvs1YYjUE7w" \t "https://developer.oculus.com/documentation/unity/unity-mobile-performance-intro/_blank)

### **Culling**

Unity offers the ability to set manual per-layer culling distances on the camera via Per-Layer Cull Distance. This may be useful for culling small objects that do not contribute to the scene when viewed from a given distance. More information about how to set up culling distances may be found here: [https://docs.unity3d.com/Documentation/ScriptReference/Camera-layerCullDistances.html](https://l.facebook.com/l.php?u=https://docs.unity3d.com/Documentation/ScriptReference/Camera-layerCullDistances.html&h=AT2O1aC82kh0Glws4rw2coQR18bDhxmSfgoVMUdksGTHEASeLcM_H1azmLpk0gd2gKBzkaCxUbomvc6Pl9Hodzy4_FAxO7vw3_dPcYVKmlYvF7h23MhJ01sB2jSqbBeTdgp6cqMu-V8l0gV8s3dgPw" \t "https://developer.oculus.com/documentation/unity/unity-mobile-performance-intro/_blank).

Unity also has an integrated Occlusion Culling system. The advice to early VR titles is to favor modest “scenes” instead of “open worlds,” and Occlusion Culling may be overkill for modest scenes. More information about the Occlusion Culling system can be found here: [https://docs.unity3d.com/Manual/OcclusionCulling.html](https://l.facebook.com/l.php?u=https://docs.unity3d.com/Manual/OcclusionCulling.html&h=AT0wmTJs5HPnzf_fsOZuwL87_t1eRlqa2jt7S8cp3KS9RshvdmcYZFmItB5ZsZV7QYVIYPrrVpNXBbZKITXIicGHZ3t4puBNVm272fzoy3x_Es3tNGWBzDvhHu6-jiqvgI9EEPq_avJFh-7h3qLOmA" \t "https://developer.oculus.com/documentation/unity/unity-mobile-performance-intro/_blank).

### **Reducing Memory Bandwidth**

* ****Texture Compression****: Texture compression offers a significant performance benefit. Favor ASTC compressed texture formats.
* ****Texture Mipmaps****: Always use mipmaps for in-game textures. Fortunately, Unity automatically generates mipmaps for textures on import. To see the available mipmapping options, switch ****Texture Type**** to ****Advanced**** in the texture inspector.
* ****Texture Filtering****: Trilinear filtering is often a good idea for VR. It does have a performance cost, but it is worth it. Anisotropic filtering may be used as well, but keep it to a single anisotropic texture lookup per fragment.
* ****Texture Sizes****: Favor texture detail over geometric detail, e.g., use high-resolution textures over more triangles. We have a lot of texture memory, and it is pretty much free from a performance standpoint. That said, textures from the Asset Store often come at resolutions which are wasteful for standalone. You can often reduce the size of these textures with no appreciable difference.
* ****Framebuffer Format****: Most scenes should be built to work with a 16 bit depth buffer resolution. Additionally, if your world is mostly pre-lit to compressed textures, a 16 bit color buffer may be used.
* ****Screen Resolution****: Setting Screen.Resolution to a lower resolution may provide a sizeable speedup for most Unity apps.

### **Reduce Geometric Complexity**

Keep geometric complexity to a minimum. 50,000 static triangles per-eye per-view is a conservative target.

Verify model vert counts are standalone-friendly. Typically, assets from the Asset Store are high-fidelity and will need tuning for standalone.

Unity provides a built-in ****Level of Detail**** system, allowing lower-resolution meshes to be displayed when an object is viewed from a certain distance. For more information on how to set up a LODGroup for a model, see the following: [https://docs.unity3d.com/Documentation/Manual/LevelOfDetail.html](https://l.facebook.com/l.php?u=https://docs.unity3d.com/Documentation/Manual/LevelOfDetail.html&h=AT1qqvlsfJsIJUuU0XN89vKhS1mwoAzn2Zv5qTkoBBrjGl8vOP1ENOr9_XdEtvBeTy601FTqe4ZO37V522VNxbE4QEH_pLz05LzmF4gs3bhjsrF1C1XqnT6kmIPsWiaqFVbqan8rkEZXqJvDjm3qrA" \t "https://developer.oculus.com/documentation/unity/unity-mobile-performance-intro/_blank)

Verify your vertex shaders are standalone friendly. And, when using built-in shaders, favor the Mobile or Unlit version of the shader.

Be mindful of Game Object counts when constructing your scenes. The more Game Objects and Renderers in the scene, the more memory consumed and the longer it will take Unity to cull and render your scene.

### **Reduce Pixel Complexity and Overdraw**

****Pixel Complexity****: Reduce per-pixel calculations by baking as much detail into the textures as possible. For example, bake specular highlights into the texture to avoid having to compute the highlight in the fragment shader.

Verify your fragment shaders are standalone friendly. And, when using built-in shaders, favor the Mobile or Unlit version of the shader.

****Overdraw****: Objects in the Unity opaque queue are rendered in front to back order using depth-testing to minimize overdraw. However, objects in the transparent queue are rendered in a back to front order without depth testing and are subject to overdraw.

Avoid overlapping alpha-blended geometry (e.g., dense particle effects) and full-screen post processing effects.

## Managing Power Consumption

Android devices are typically constrained by the processing power of the device and its ability to dissipate heat.

Android devices can manage the heat and their power consumption using the Fixed Clock Level API on Gear VR and Dynamic Clock Throttling. A detailed overview of these two features is available on the [Power Management](https://developer.oculus.com/documentation/native/android/mobile-power-overview/" \l "mobile-power-overview) page in the Mobile SDK guide.

To set your clock level in Unity apps, set OVRManager.cpuLevel ( ) and OVRManager.gpuLevel ( ).

## Best Practices

This section describes best practices for Android projects.

* Be batch friendly. Share materials and use a texture atlas when possible.
* Prefer lightmapped, static geometry.
* Prefer lightprobes instead of dynamic lighting for characters and moving objects.
* Always use baked lightmaps rather than precomputed GI.
* Use Non-Directional Lightmapping.
* Bake as much detail into the textures as possible. E.g., specular reflections, ambient occlusion.
* Only render one view per eye. No shadow buffers, reflections, multi-camera setups, et cetera.
* Keep the number of rendering passes to a minimum. No dynamic lighting, no post effects, don’t resolve buffers, don’t use grabpass in a shader, et cetera.
* Avoid alpha tested / pixel discard transparency. Alpha-testing incurs a high performance overhead. Replace with alpha-blended if possible.
* Keep alpha blended transparency to a minimum.
* Be sure to use texture compression. We recommend using ASTC texture compression as a global setting.
* Check the ****Disable Depth and Stencil****\* checkbox in the ****Resolution and Presentation**** pane in ****Player Settings****.
* Be sure to initialize GPU throttling, and avoid dangerous values (-1 or >3) See [Power Management](https://developer.oculus.com/documentation/native/android/mobile-power-overview/) in our Mobile SDK Developer Guide for more information.
* Avoid full screen image effects.
* Be careful using multiple cameras with clears - doing so may lead to excessive fill cost.
* Avoid use of LoadLevelAsync or LoadLevelAdditiveAsync. This has a dramatic impact on framerate, and it is generally better to go to black and load synchronously.
* Avoid use of Standard shader or Standard Specular shader.
* Avoid using Projectors, or use with caution - they can be very expensive.
* Avoid Unity’s Default-Skybox, which is computationally expensive for standalone. We recommend setting Skybox to None (Material), and Ambient Source to Color in Window > Lighting. You may also wish to set Camera.clearFlags to SolidColor (never Skybox).

### **CPU Optimizations**

* Be mindful of the total number of Game Objects and components your scenes use.
* Model your game data and objects efficiently. You will generally have plenty of memory.
* Minimize the number of objects that actually perform calculations in Update() or FixedUpdate().
* Reduce or eliminate physics simulations when they are not actually needed.
* Use object pools to respawn frequently-used effects or objects instead of allocating new ones at runtime.
* Use pooled AudioSources versus PlayOneShot sounds, as the latter allocate a Game Object and destroy it when the sound is done playing.
* Avoid expensive mathematical operations whenever possible.
* Cache frequently-used components and transforms to avoid lookups each frame.
* Use the Unity Profiler to:
  + Identify expensive code and optimize as needed.
  + Identify and eliminate Garbage Collection allocations that occur each frame.
  + Identify and eliminate any spikes in performance during normal play.
* Do not use Unity’s OnGUI() calls.
* Do not enable gyro or the accelerometer. In current versions of Unity, these features trigger calls to expensive display calls.
* All best practices for standalone apps and game development generally apply.

## Startup Sequence and Reserved Interactions

This section describes startup sequence recommendations and mandatory behaviors required for standalone applications.

### **Startup Sequence**

For good VR experiences, all graphics should be rendered such that the user is always viewing a proper three-dimensional stereoscopic image. Additionally, head-tracking must be maintained at all times. We recommend considering using a cubemap overlay for your startup screen (see [VR Compositor Layers](https://developer.oculus.com/documentation/unity/unity-ovroverlay/)), which will render at a consistent frame rate even if the application is unavailable to update the scene.

An example of how to do this during application start up is demonstrated in the SDKExamples Startup\_Sample scene:

* Solid black splash image is shown for the minimum time possible.
* A small test scene with 3D logo and 3D rotating widget or progress meter is immediately loaded.
* While the small start up scene is active, the main scene is loaded in the background.
* Once the main scene is fully loaded, the start scene transitions to the main scene using a fade.

### **Reserved Interactions**

For more information about the Oculus reserved interactions, see [Universal Menu and Reserved User Interactions](https://developer.oculus.com/documentation/native/android/mobile-umenu-intro/) in our Mobile Developer Guide.

See the class description of OVRPlatformMenu in our [Unity Scripting Reference](https://developer.oculus.com/documentation/unity/unity-reference-scripting/) for details about the relevant public members.

### **Volume**

The volume buttons are reserved, and volume adjustment on the Samsung device is handled automatically. Volume control dialog is also handled automatically by the VrApi as of Mobile SDK 1.0.3, supported by Utilities for Unity versions 1.5.0 and later. Do not implement your own volume handling display, or users will see two juxtaposed displays.

It is possible to override automatic volume display handling by setting VRAPI\_FRAME\_FLAG\_INHIBIT\_VOLUME\_LAYER as an ovrFrameParm flag.