Unity Performance Best Practices

v. 0.4

Good performance is critical for VR applications. This section provides simple guidelines to help your Android Unity app perform well. Please review the section titled “Performance Advice for Early Titles” in *Design Guidelines* before reading this guide.

1. General CPU Optimizations

To create a VR application or game that performs well, careful consideration must be given to how features are implemented. Scene should always run at 60 FPS, and you should avoid any hitching or laggy performance during any point that the player is in your game.

- Be mindful of the total number of GameObjects 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 versus allocating new ones at runtime.
- Use pooled AudioSources versus PlayOneShot sounds which allocate a GameObject 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.
- Use the Unity Profiler to identify and eliminate Garbage Collection (GC) allocations that occur each frame.
- Use the Unity Profiler to 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 mobile app and game development generally apply.

2. Rendering Optimization

While building your app, the most important thing to keep in mind is to 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

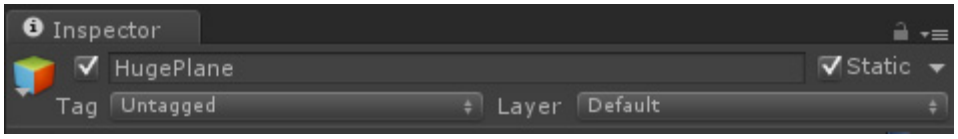
Keep the total number of draw calls to a minimum. A conservative target would be less than 100 draw calls per frame.

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.



[Unity Pro 4.5]

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

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

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:
http://blogs.unity3d.com/2013/12/02/occlusion-culling-in-unity-4-3-the-basics/

Reducing Memory Bandwidth

Texture Compression

Texture compression offers a significant performance benefit. Favor ETC2 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 mobile. 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 mobile-friendly. Typically, assets from the Asset Store are high-fidelity and will need tuning for mobile.

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

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

Bake as much detail into the textures as possible to reduce the computation per vertex, for example, baked bumpmapping as demonstrated in the Shadowgun project:
https://docs.unity3d.com/430/Documentation/Manual/iphone-PracticalRenderingOptimizations.html

Be mindful of GameObject counts when constructing your scenes. The more GameObjects 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 mobile 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.

3. Best Practices

- **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.**

- **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.**

- **Use Texture Compression.**
  Favor ETC2.

- **Do not enable MSAA on the main framebuffer.** MSAA may be enabled on the Eye Render Textures.

4. Design Considerations

Please review *Design Guidelines* if you have not already done so.

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.

An example of how to do this during application startup 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 startup 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.

Universal Menu Handling

Applications will need to handle the Back Key long-press action which launches the Universal Menu as well as the Back Key short-press action which launches the “Confirm-Quit to Home” Menu which exits the current application and returns to the Oculus Home application.

An example of demonstrating this functionality is in the SDKExamples GlobalMenu_Sample scene.

More information about application menu options and access can be found in the *Universal Menu* document.

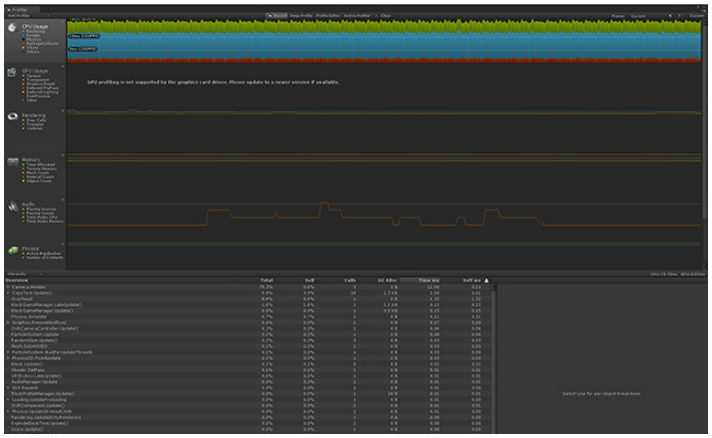
5. Unity Profiling Tools

Even following the guidelines above, you may find you are not hitting a solid 60 FPS. The next section details the various tools provided by Unity to help you diagnose bottlenecks in Android applications. For additional profiling tools, see the the *Performance Analysis* document.

Unity Profiler

Unity Pro comes with a built-in profiler. The profiler provides per-frame performance metrics, which can be used to help identify bottlenecks.

You may profile your application as it is running on your Android device using adb or WIFI. For steps on how to set up remote profiling for your device, please refer to the Android section of the following Unity documentation:
https://docs.unity3d.com/Documentation/Manual/Profiler.html



[Unity Pro 4.5]

The Unity Profiler displays CPU utilization for the following categories: Rendering, Scripts, Physics, GarbageCollector, and Vsync. It also provides detailed information regarding Rendering Statistics, Memory Usage (including a breakdown of per-object type memory usage), Audio and Physics Simulation statistics.

GPU Usage data for Android is not available at this time.

The Unity profiler only displays performance metrics for your application. If your app isn't performing as expected, you may need to gather information on what the entire system is doing.

Show Rendering Statistics

Unity provides an option to display real-time rendering statistics, such as FPS, Draw Calls, Tri and Vert Counts, VRAM usage.

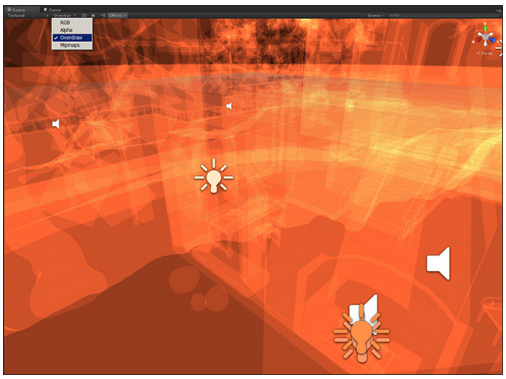
While in the Game View, pressing the Stats button (circled in red in the upper-right of the following screenshot) above the view window will display an overlay showing realtime render statistics.

[VrScene: Tuscany]

Show GPU Overdraw

Unity provides a specific render mode for viewing overdraw in a scene. From the Scene View Control Bar, select **OverDraw** in the drop-down Render Mode selection box.

In this mode, translucent colors will accumulate providing an overdraw “heat map” where more saturated colors represent areas with the most overdraw.



[VrScene: Tuscany]

6. Additional Reading

We recommend reviewing the following documents:

- *Performance Analysis*
- *Performance Guidelines*

Last Update: Nov 10, 2014