

Unity* Optimization Guide for x86 Android*

By Cristiano Ferriera and Steve Hughes

Contents

Tools.....	2
The Unity Profiler.....	3
The GPA System Analyzer	4
The GPA Frame Analyzer.....	7
Optimizations	12
Scripting Optimizations.....	12
Script Frustum Culling and Co-routines.....	12
Smart Memory Management.....	14
Caching Frequent Objects and Components.....	15
Best Practices for Working with the Unity Physics System	16
Editor Optimizations.....	17
Occlusion Culling (Pro Only)	17
LOD: Level of Detail (Pro Only).....	21
Shadows	23
Render Ordering.....	26
Lightmapping.....	27
Using Simple Colliders Instead of Mesh Colliders for Complex Models.....	29
Compressing Textures	30
Mobile Stock Shaders.....	32
Selecting the Optimal Render Path	33
Static Batching (Pro Only).....	34
Dynamic Batching	35
HDR – High Dynamic Range.....	36
Splitting the Binary	36
Conclusion.....	38

Resources	38
About the Authors.....	38

To get the most out of the Android* x86 platform there are a number of performance optimizations you can apply to your project that help to maximize performance. In this guide, we will show a variety of tools to use as well as features in the Unity* software that can help you enhance the performance of the native x86 code. We will discuss how to handle items like texture quality, batching, culling, light baking, and HDR effects. Additionally, we also will show how to build an x86-specific binary for testing and other needs.

By the end of this guide you will be able to identify performance issues and what they are bound to, key optimizations, and methodologies for good game development in Unity. First we will go over some of the tools available that will make it easy to identify potential hot spots in your application.

Tools

We will explore three main tools in this guide: Unity Profiler, GPA System Analyzer, and GPA Frame Analyzer. Each tool is powerful in its own right in regards to solid game development. If you are able to use all three, you will realize significant progress in streamlining and optimizing your game.

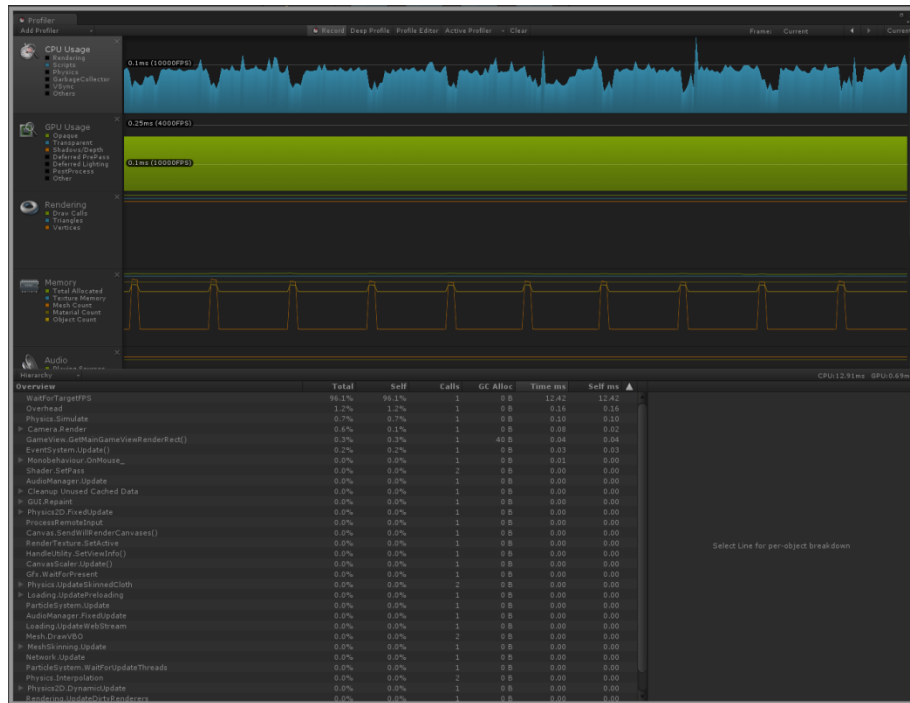


Figure 1. The Unity Profiler main screen

The Unity Profiler

The Unity Profiler (Figure 1) is an extremely powerful tool available in the Pro version of Unity that will help you identify issues in various subsystems in Unity. The profiler graph section has different sub-profilers that show metrics for specific hardware. The current sub-profilers available include CPU usage, GPU usage, Rendering, Memory, Audio, Physics, and Physics 2D. Each of these sub-profilers is further broken down into sections of relevant components that can be isolated to drill down into specifics. For example, CPU usage contains Rendering, Scripts, Physics, GarbageCollector, Vsync, and Others sections.

Below the graph section is the Overview window where you can see a list of metrics including timing info and memory allocations for various Unity subsystems. Everything from rendering to garbage collection is shown here, and it is a good idea to check the sections of your app that take the longest time for optimization opportunities. Clicking on any section of the graph will pause updates in the profiler and allow you to investigate the highlighted frame.

The Unity Profiler can be attached to a running app in the editor or in a standalone build. It is recommended to always attach to a standalone build when trying to get the most accurate timings to avoid the overhead of the editor. This can be done by going to the 'Active Profiler' button towards the top of the window and selecting from the available instances of 'Android Player' that were detected over ADB (Android Debug Bridge) as well as anything on the network.

Another option is to 'Deep Profile' the app. This option is not recommended for ordinary use as it will actually instrument all mono code, which can lead to a lot of overhead when profiling. Luckily, Unity has a way to explicitly instrument any code segment you are interested in. Figure

2 shows how to instrument the code so it will appear in the profiler with whatever label you supply:

```
// Update is called once per frame
0 references
void Update () {
    if(mMyState == STATE.GOOD)
    {
        Profiler.BeginSample("Object fetching the good way");
        for(int i = 0; i < PlainCubeManager.Singleton.mPlainCubes.Count; ++i)
        {
            PlainCubeManager.Singleton.mPlainCubes[i].DoSomething();
        }
        Profiler.EndSample();
    }
    else
    {
        Profiler.BeginSample("Object fetching the bad way");
        PlainCube[] objects = FindObjectsOfType<PlainCube>();
        for(int i = 0; i < objects.Length; ++i)
        {
            objects[i].DoSomething();
        }
        Profiler.EndSample();
    }
}
```

Figure 2. Setting a code segment for use in Profiler

The GPA System Analyzer

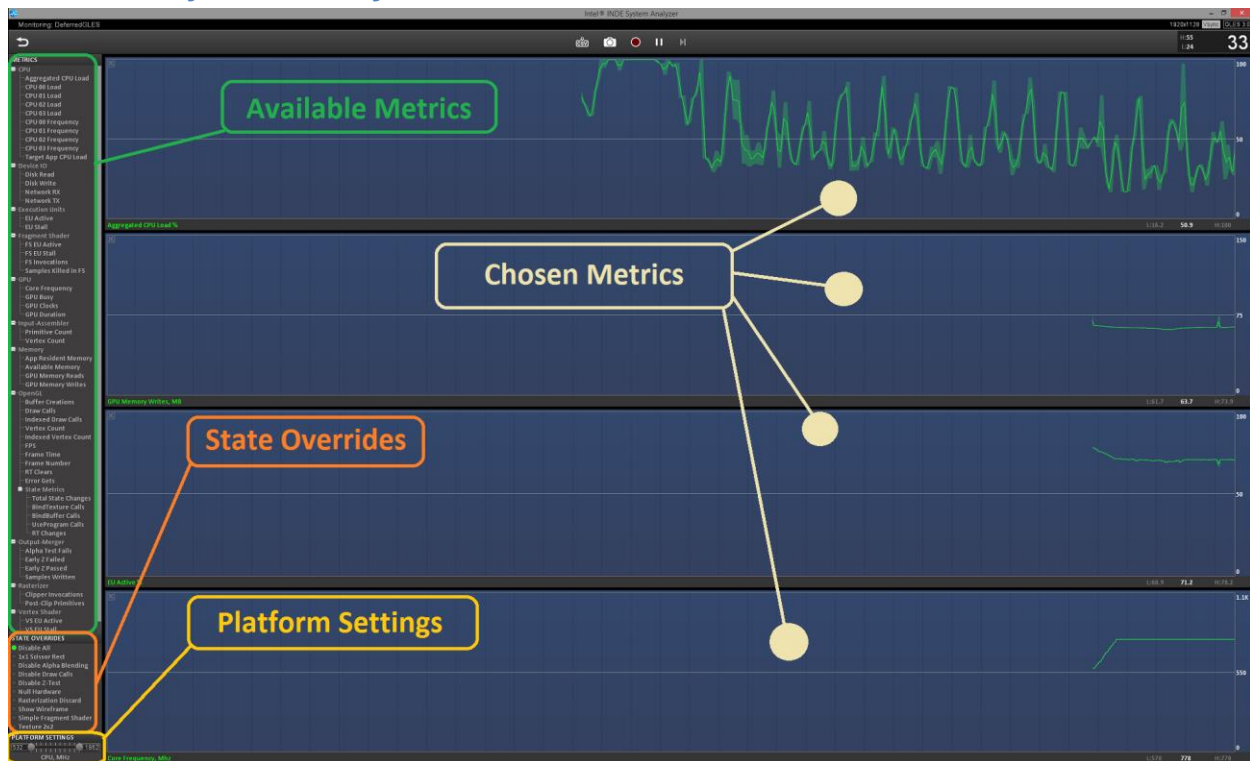


Figure 3. GPA System Analyzer real-time view

The Intel® Graphics Performance Analyzers (Intel® GPA) is a suite of graphics analysis and optimization tools to help game developers make games and other graphics-intensive applications run faster. Intel GPA provides extensive functionality to allow developers to perform in-depth analysis of graphics API calls to determine where the primary performance issues arise. Many of the experiments and metrics shown in this guide are from Intel GPA. Intel GPA lets you study the graphics workload of DirectX* apps on Windows* and OpenGL ES* apps on select Intel® processor systems running Android. While it cannot directly monitor OpenGL* API calls, you can still use GPA System Analyzer to study GPU and CPU metrics as your OpenGL game runs. Regardless of the graphics API, you can also use GPA Platform Analyzer to see the detailed CPU load, including any OpenCL™ activity. If you want a closer look, Intel GPA has an API for adding your own instrumentation. The GPA toolset works on Android as well as desktop, and you can learn more and download Intel GPA here: www.intel.com/software/GPA/

The first step is to use Intel GPA to collect real-time performance information. Intel GPA has two different modes for real-time data display (both shown above): The heads-up display (HUD) that runs on top of your application and the System Analyzer that connects to your test system across the network. Either tool can show metrics from the DirectX pipeline (OpenGL ES pipeline on some Intel processors), CPU utilization, and system power. On supported Intel processor graphics systems, you also get extensive GPU hardware metrics. The HUD and System Analyzer provide simple experiments to help you quickly detect performance issues. See the [Intel GPA documentation](#) for more details on the HUD and System Analyzers features and functionality.

To include a metric's values in the analysis, simply drag it from the left sidebar into the main graphing area. The tools will work on ARM* devices, but will not have all of the metrics that are available on Intel processor-based hardware. For further information, check out the GPA tutorials for [Windows](#) and [OS X](#). The following groups of metrics are available on Intel hardware:

- CPU
- Device IO
- Execution Units
- Fragment Shader
- GPU
- Input-Assembler
- Memory
- OpenGL/DX
 - State Metrics
- Output-Merger
- Power
- Rasterizer
- Vertex Shader

For CPU bottlenecks, you may find Platform Analyzer useful for DirectX and OpenGL workloads. It displays a captured trace of CPU activity. If you add instrumentation to your code, you can correlate individual tasks running on the CPU and watch their progress through DirectX, the driver, and in to the GPU. To help you determine bottlenecks, GPA contains a 'State Overrides' section (Figure 4) that allows you to perform experiments by checking frame rate fluctuations against changing conditions. A few examples:

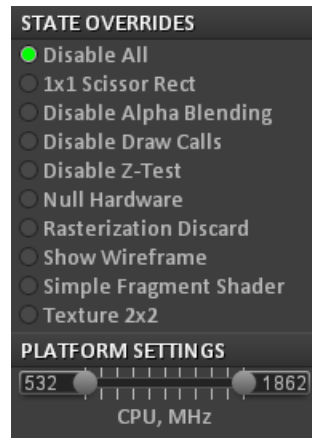


Figure 4. Available overrides

- Texture 2x2
 - Fetching data from high res textures can be expensive. This will replace all textures used in the scene with 2x2 textures. Significant performance changes resulting from checking this option might suggest some textures could be reduced in size to improve frame rate.
- Null Hardware
 - This will simulate an infinitely fast GPU. If this increases frame rate, your code is likely driver or CPU bound.
- Disable Draw Calls
 - This will simulate a very fast driver, indicating that your code may be driver bound if the frame rate fluctuates.
- Simple Fragment Shader
 - This will replace all shaders with a very simple fragment shader. Fluctuation may indicate that shaders should be optimized for a performance bump.

Below the experiments section is the platform settings slider. This feature allows you to run the CPU at various frequencies. This will help determine bottlenecks, even if your game / app is running at max frame rate on any device you are using to test. This can also be used to verify that your game / app will run on the widest range of devices.

Finally, you can click the camera icon towards the top of the window to take a frame capture. The system analyzer will then record everything that goes into producing a single frame of your game / app (state changes, timings, textures, etc.). This information will be saved in a file that can be opened by the Frame Analyzer tool to enable a deeper dive.

The GPA Frame Analyzer

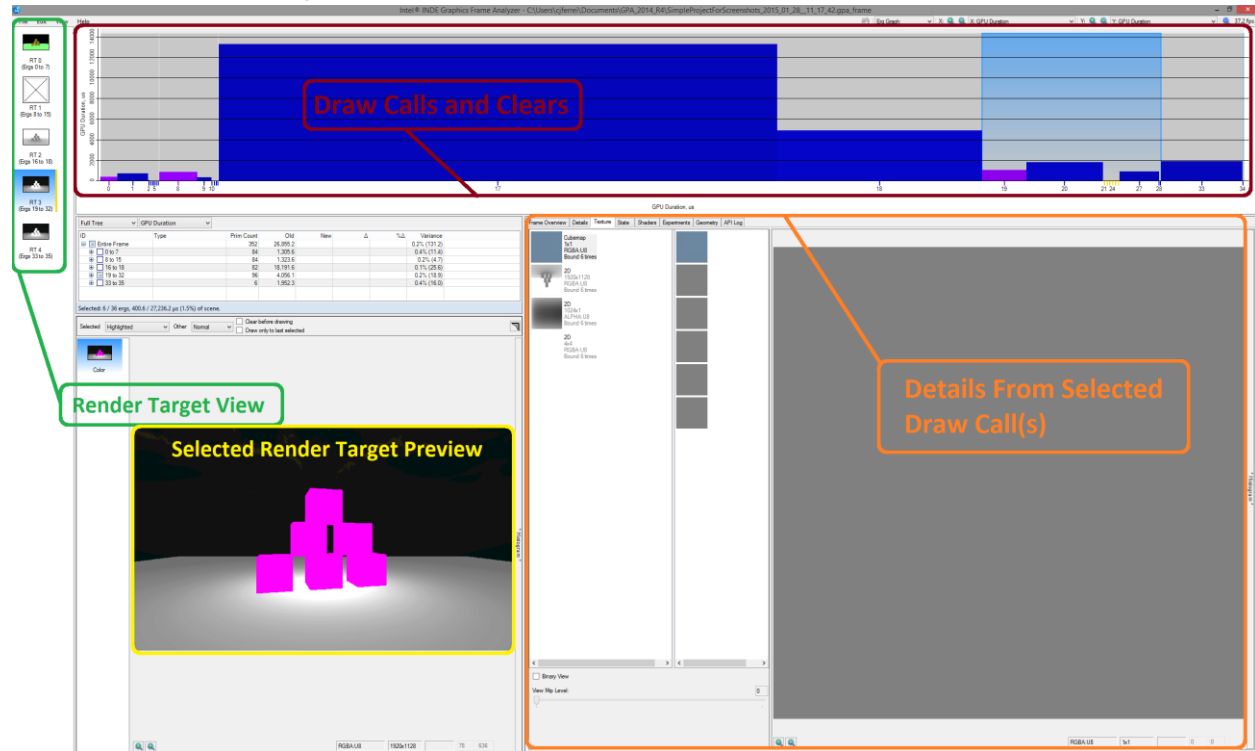


Figure 5. GPA Frame Analyzer showing change records and associated frame info

The Frame Analyzer tool (Figure 5) allows you to open up a single frame capture. The captured frame will contain records of all state changes, resources, timing info, and much more. At the top of the window is a graph that displays each individual draw call recorded in the frame. These draw calls are all separated per render target for easy visualization. The X and Y values of the graph can be changed via the drop-down menus on the top left. On the left is a list of the individual render targets. The lower left section shows a preview of the currently highlighted draw calls and how they appear on the frame. Various options allow you to customize the view, including highlighting the pixels drawn to or keeping them normal. You also have the option to adjust how everything that is not selected affects the preview (hidden or not). In the bottom right is a set of tabs to get more insight into the currently selected draw calls including:

- **Frame Overview (Figure 6)**
 - Timing / state values broken down by stages in the GPU pipeline for the entire frame

Frame Overview						Details	Texture	State	Shaders	Experiments	Geometry	API Log
Metric	/	Old Value	New V...	Delta	% ...	Description						
Execution Units												
- EU Active %		38.1				The percentage of time that the EU array is active.						
- EU Stall %		12.9				The percentage of time that the EU array is stalled.						
Fragment Shader												
- FS EU Active %		37.0				The percentage of time that the EUs were actively executing Fragment Shader instructions.						
- FS EU Stall %		11.4				The percentage of time that the EUs were stalled in the Fragment Shader.						
- FS Invocations		6,483,864.0				The number of times a fragment shader was invoked.						
- Samples Killed in FS		0.0				Number of fragments/samples killed in the fragment shader.						
Input-Assembler												
- Primitive Count		25,692.0				The number of rendering primitives assembled and put into the input assembly stage of the pipeline.						
- Vertex Count		77,072.0				The number of vertices that entered the pipeline.						
Main												
- Core Frequency, Mhz		778.0				GPU frequency during measurement period.						
- GPU Clocks		9,804,142.0				Number of core clock cycles.						
- GPU Duration, us		13,183.1				Total GPU duration for selected work items.						
Memory												
- GPU Memory Reads, MB		7.1				The total number of bytes read from memory						
- GPU Memory Writes, MB		24.1				The total number of bytes written to memory						
Output-Merger												
- Alpha Test Fails		33,234.0				The number of fragments that failed the alpha test.						
- Early Z Failed		1,873,440.0				The number of fragments that failed the early depth/stencil tests.						
- Early Z Passed		6,373,725.0				The number of fragments that passed the early depth/stencil tests.						
- Samples Written		4,872,899.0				Number of fragments successfully rendered and written to render buffer memory. It doesn't account for unli...						
Rasterizer												
- Clipper Invocations		25,692.0				The number of primitives sent to the Clipper.						
- Post-Clip Primitives		18,661.0				The number of primitives that flowed out of the Clipper.						
Vertex Shader												
- VS EU Active %		1.4				The percentage of time that the EUs were actively executing Vertex Shader instructions.						
- VS EU Stall %		1.5				The percentage of time that the EUs were stalled in the Vertex Shader.						
- VS Invocations		33,825.0				The number of times a vertex shader was invoked.						

Figure 6. Values reported in the Frame Overview section

- **Details**
 - Timing / state values broken down by stages in the GPU pipeline for the draw calls currently selected in the graph / tree.
- **Texture** (Figure 7 below)
 - A list of currently bound textures
 - The left sidebar under the texture tab can be used to verify compression, format, mip levels, etc.

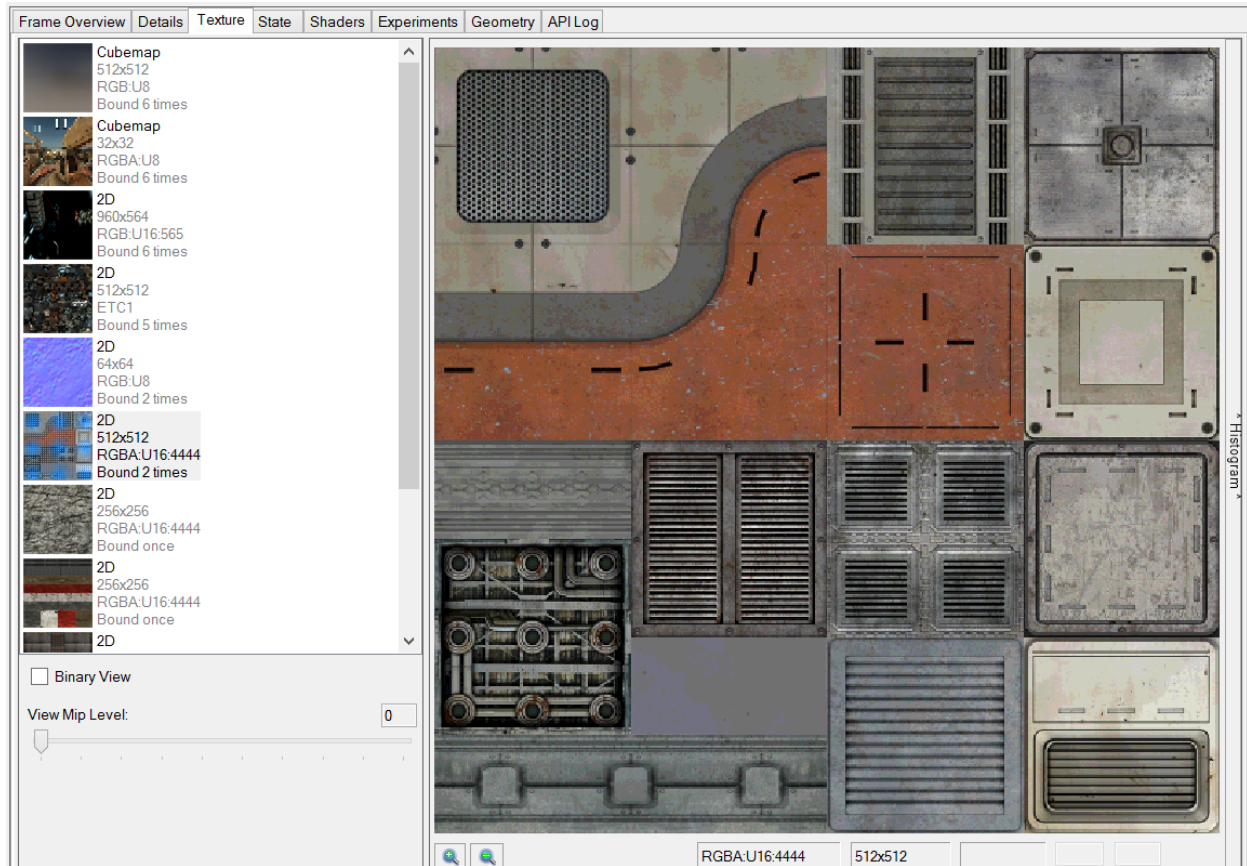


Figure 7. View of textures used in a few draw calls

- **State**
 - The state settings for selected draw call(s)
 - Can be edited to view the effect on render target preview and timings
- **Shaders** (Figure 8)
 - This section allows you to view the shaders used in the selected draw calls.
 - You can edit the shader code and see how the changes affect the scene preview visually. Changes to the shader code will also be reflected in a timing change, so you can see how much a specific optimization is affecting the frame time.

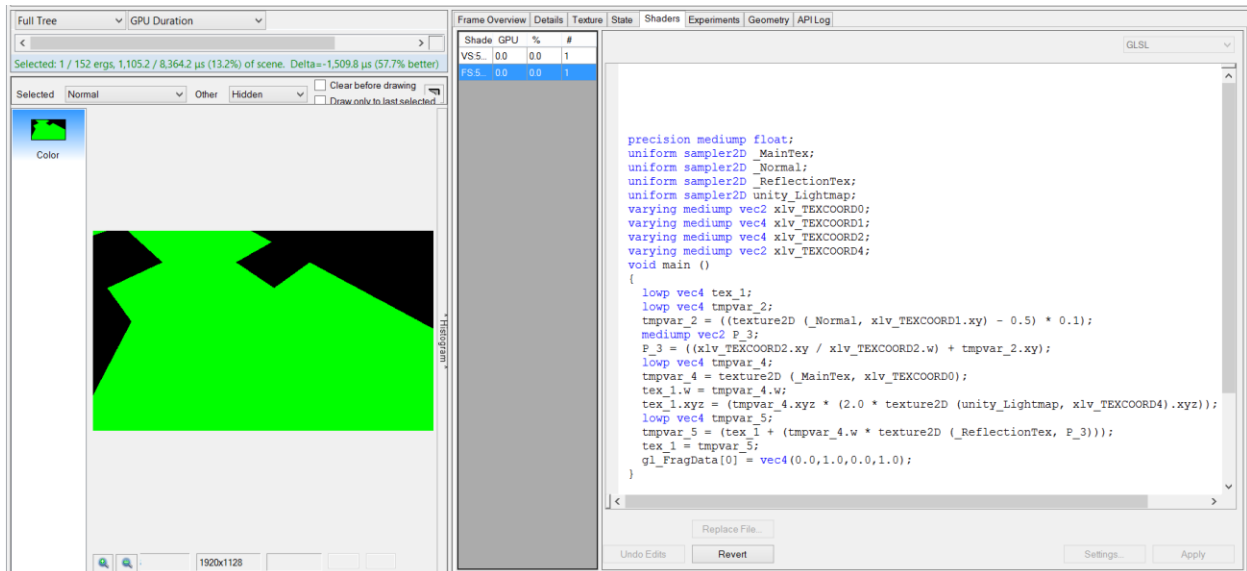


Figure 8. Editing a shader to output a hardcoded green value, making the draw call 57.7% faster

- Experiments (Figure 9)
 - Similar to the experiments section in the System Analyzer, but can be used on a per-draw-call basis.

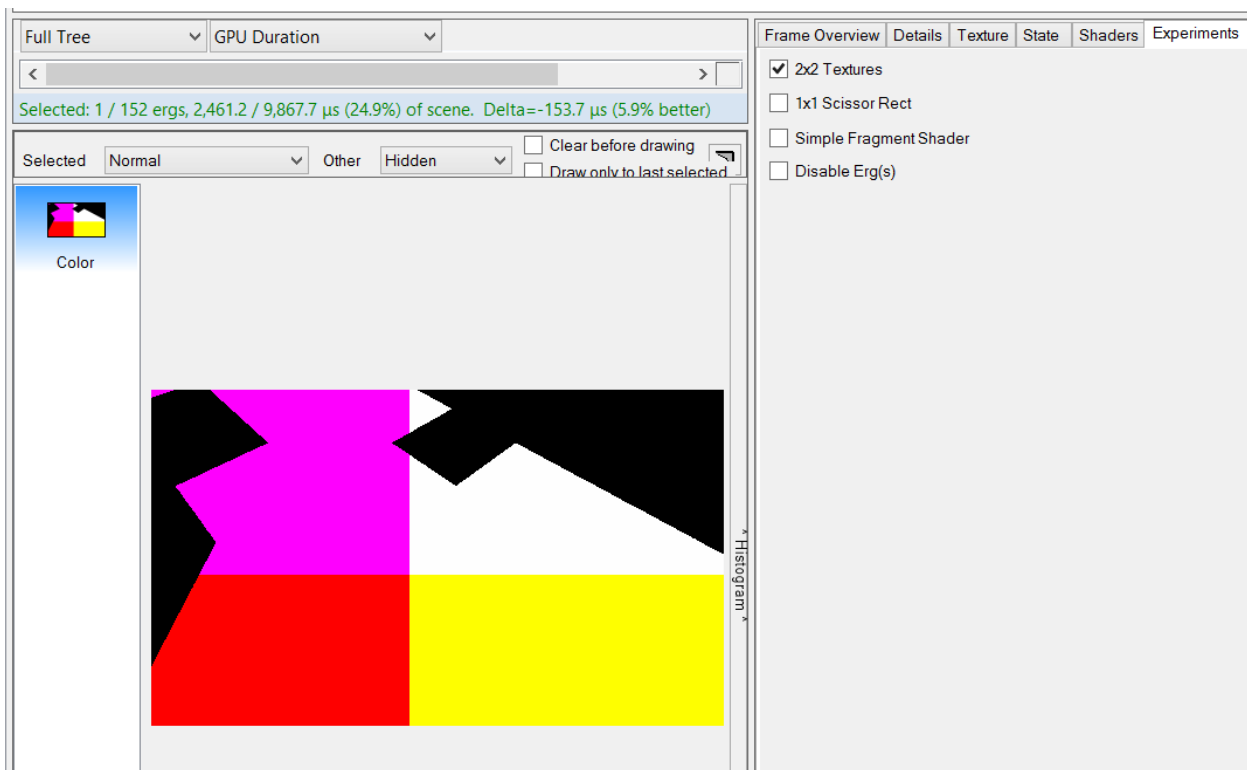


Figure 9. An experiment that shows how substituting a 2x2 texture would make this draw call 5.9% faster. Changes can also be seen on individual points in the graphics pipeline in the Details tab.

- Geometry (Figure 10)

- This tab displays a 3D representation of the geometry data in a window for the selected draw calls.

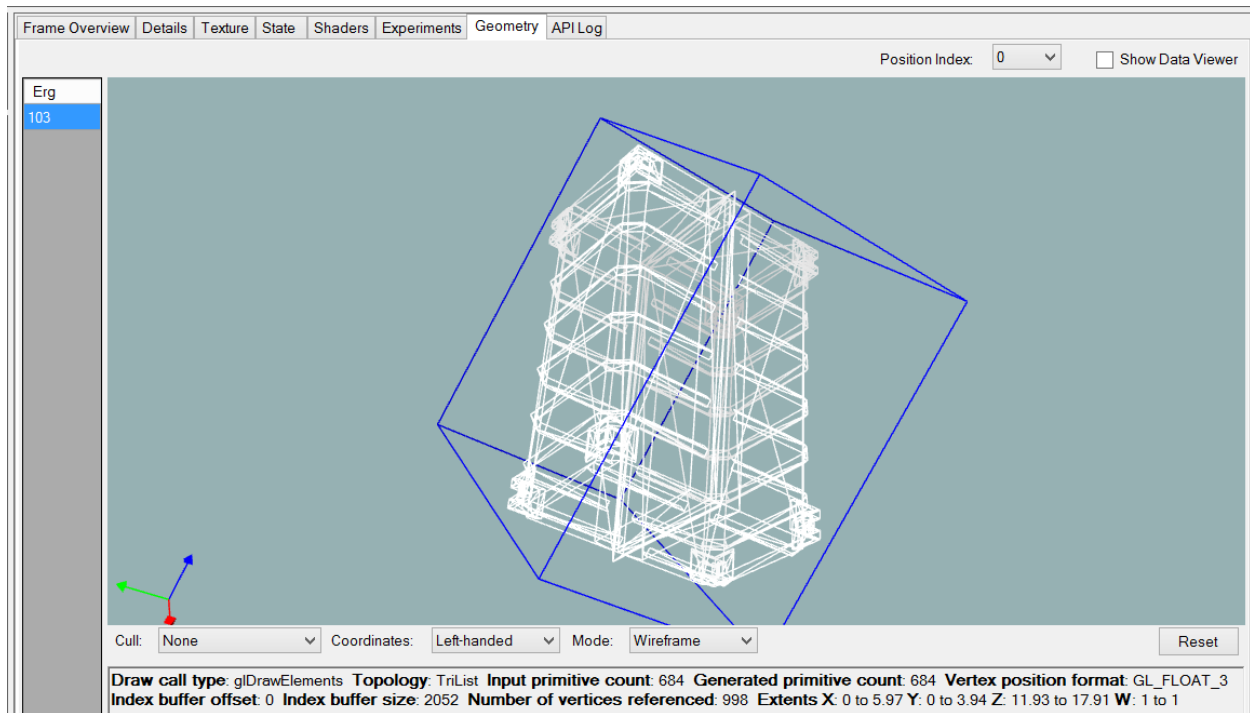


Figure 10. *Viewing model geometry in the Geometry tab*

- **API Log**

- Displays all of the API calls used for the selected draw calls. This can be immensely useful in tracking down unnecessary state changes that can impact performance.

Optimizations

We will explore two main areas of optimizations in this guide: Scripting and Editor-based. You will see a number of specifics for each optimization based on the roots in which it functions.

Scripting Optimizations

Script Frustum Culling and Co-routines

When you are profiling your application and see that a script's Update() function does not need to be called every frame, you have a few great methods to reduce the amount of updates:

- **Script Frustum Culling**

Use the following MonoBehaviour callbacks to cull scripts outside of the camera frustum that do not need to update when not in focus.

```
O references
void OnBecameVisible()
{
    enabled = true;
}

O references
void OnBecameInvisible()
{
    enabled = false;
}
```

- **Co-routines**

Co-routines are essentially functions with the ability to pause and resume execution. The power of co-routines can be leveraged by removing the original Update() function in your script and replacing it with a co-routine. You can then set how often you would like your co-routine to execute using the **yield** command. This snippet shows how to create a custom smart update that is called every 2 seconds:

0 references

```
void Start()
{
    StartCoroutine("MyCoroutine");
}
```

0 references

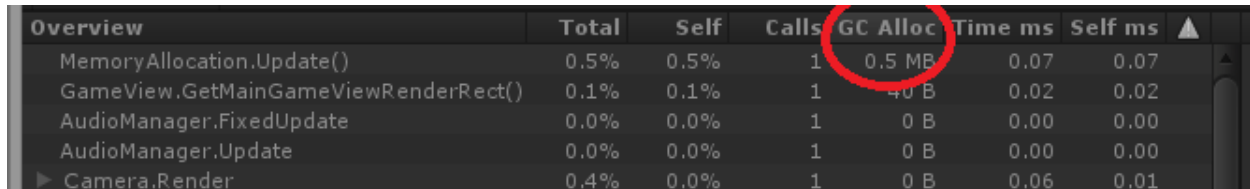
```
IEnumerator MyCoroutine()
{
    while (true)
    {
        MoreEfficientUpdate();
        yield return new WaitForSeconds(2.0f);
    }
}
```

1 reference

```
void MoreEfficientUpdate()
{
    Debug.Log("I'm so efficient!");
}
```

Smart Memory Management

When looking for ways to optimize your memory usage, it is helpful to check the Unity profiler first. A great way to get an overview of how you are managing memory is to check the 'GC Alloc' section of the Overview window (Figure 11) and step through your frames until you see a significant allocation.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms	
MemoryAllocation.Update()	0.5%	0.5%	1	0.5 MB	0.07	0.07	
GameView.GetMainGameViewRenderRect()	0.1%	0.1%	1	40 B	0.02	0.02	
AudioManager.FixedUpdate	0.0%	0.0%	1	0 B	0.00	0.00	
AudioManager.Update	0.0%	0.0%	1	0 B	0.00	0.00	
Camera.Render	0.4%	0.0%	1	0 B	0.06	0.01	

Figure 11. By inspecting many frames in a row, you can determine when GC will occur and adjust

It is also helpful to check how frequent garbage collection is being invoked. To see this, isolate the GarbageCollector field in the 'CPU Usage' sub-profiler (Figure 12):

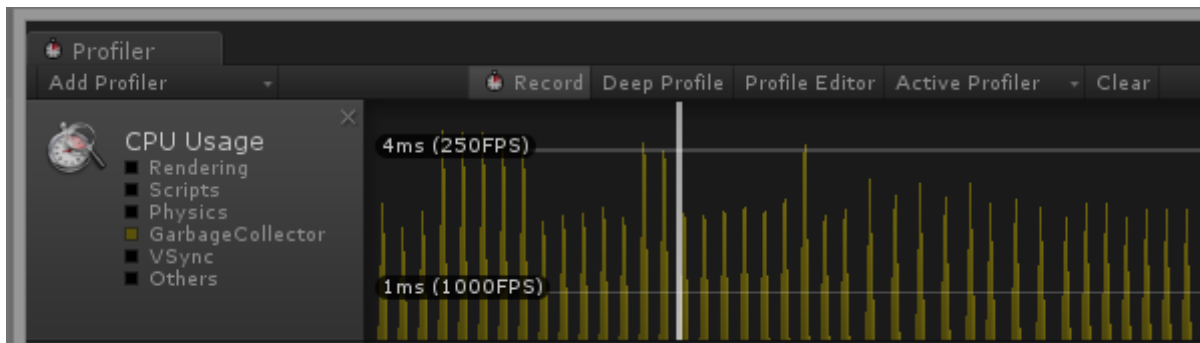
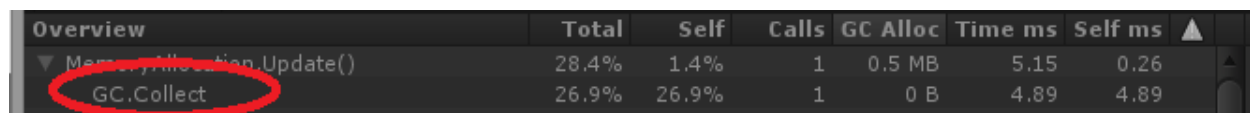


Figure 12. Point identified in CPU usage when GC occurs

When collects are displayed, you can then click on a peak in the graph and look for the call to GC.Collect (Figure 13). By doing this you can see how much time each collect takes:



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms	
MemoryAllocation.Update()	28.4%	1.4%	1	0.5 MB	5.15	0.26	
GC.Collect	26.9%	26.9%	1	0 B	4.89	4.89	

Figure 13. Statistics on GC

To avoid frequent allocations, it is advantageous to use structs instead of classes to have allocations be done on the stack, instead of in the heap. Multiple allocations to the heap can lead to significant memory fragmentation and frequent garbage collections.

Caching Frequent Objects and Components

As a rule of thumb, you should analyze your apps to find the most frequently used GameObjects and Components and make sure that these values are being cached. Any time you see an object being fetched every scene is an opportunity for caching and saving unnecessary computation.

The same rule applies to GameObject **instantiation**. In general, instantiation is a relatively slow call that should be avoided. If creating and destroying the same object types repeatedly in every scene, it is advantageous to maintain a list of those objects in a game manager script.

Unity recommends a central game manager to maintain lists of all of your cached game objects. After implementing this technique, you can include the following code snippet to compare the performance delta between the two methods by toggling a state button, or other control mechanism while viewing the **CPU Usage profiler** in real time. Here is the snippet (Figure 14) showing the difference in usage:

```
// Update is called once per frame
Oreferences
void Update () {
    if(mMyState == STATE.GOOD)
    {
        Profiler.BeginSample("Object fetching the good way");
        for(int i = 0; i < PlainCubeManager.Singleton.mPlainCubes.Count; ++i)
        {
            PlainCubeManager.Singleton.mPlainCubes[i].DoSomething();
        }
        Profiler.EndSample();
    }
    else
    {
        Profiler.BeginSample("Object fetching the bad way");
        PlainCube[] objects = FindObjectsOfType<PlainCube>();
        for(int i = 0; i < objects.Length; ++i)
        {
            objects[i].DoSomething();
        }
        Profiler.EndSample();
    }
}
```

Figure 14. Using STATE to toggle objects

Best Practices for Working with the Unity Physics System

When working with dynamic objects in Unity, there are a few well-known optimizations and pitfalls to avoid. Whether you plan to move the object yourself or allow Unity to take control of the physics of an object, add a **Rigidbody** component to your object. This tells the Unity physics system that the object is moveable. When you wish to move the object manually, simply check the **isKinematic** flag (Figure 15). You also want to make sure that the **static** checkbox at the top right corner of the inspector is unchecked for that object (Figure 16).

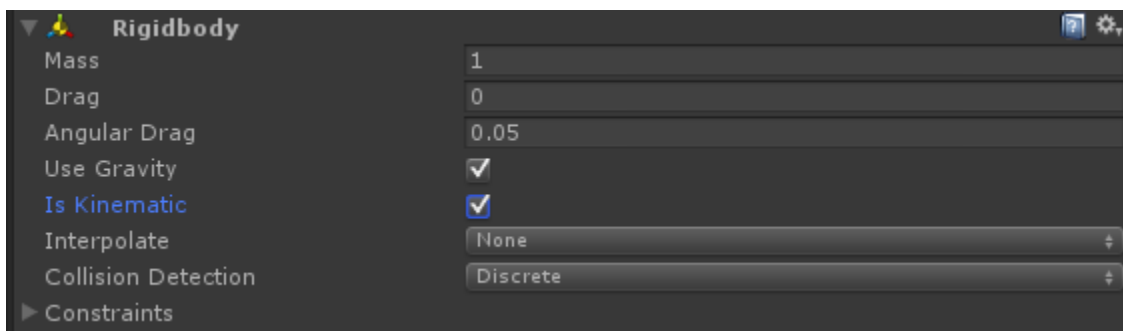


Figure 15. *isKinematic* checked to take control over objects movement

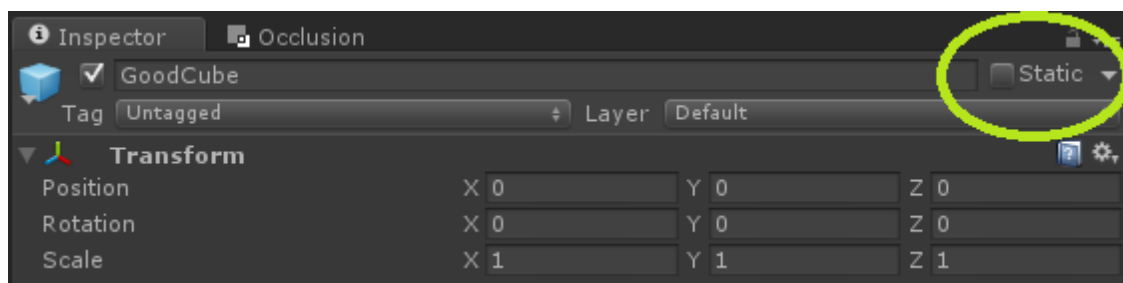


Figure 16. *Static* property unchecked to keep dynamic object out of static set

To make sure that you are handling dynamic objects properly in your app, open up the profiler, isolate the **physics** subsection of the **CPU Profiler**, highlight a frame that lands on the physics time step (24 updates per second by default), and verify that you do not see any “**Static Collider.Move (Expensive delayed cost)**” entries (Figure 17) in the overview window under the object’s FixedUpdate() call. The lack of a Static Collider.Move message indicates the physics in this section is working appropriately.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms	
Physics.Simulate	48.7%	48.7%	2	0 B	5.91	5.91	
▼ BadCube.FixedUpdate()	1.2%	29.8%	1250	0 B	3.79	3.62 .250	
Static Collider.Move (Expensive delayed cost)	1.3%	1.3%	1250	0 B	0.16	0.16 .250	

Figure 17. *Static Collider.Move (Expensive delayed cost)* appears when you are not managing dynamic objects properly

Editor Optimizations

Occlusion Culling (Pro Only)

Occlusion culling is a feature available in the pro version of Unity that enables you to cull out objects that are occluded by other objects with respect to the camera. Let's use a fantasy MMORPG as a simple example. If the player were to walk up to a giant castle containing a sprawling city, would you really want to spend system resources to render all of the occluded shops / players within the city walls? Didn't think so. Occlusion culling is the answer (Figure 18)

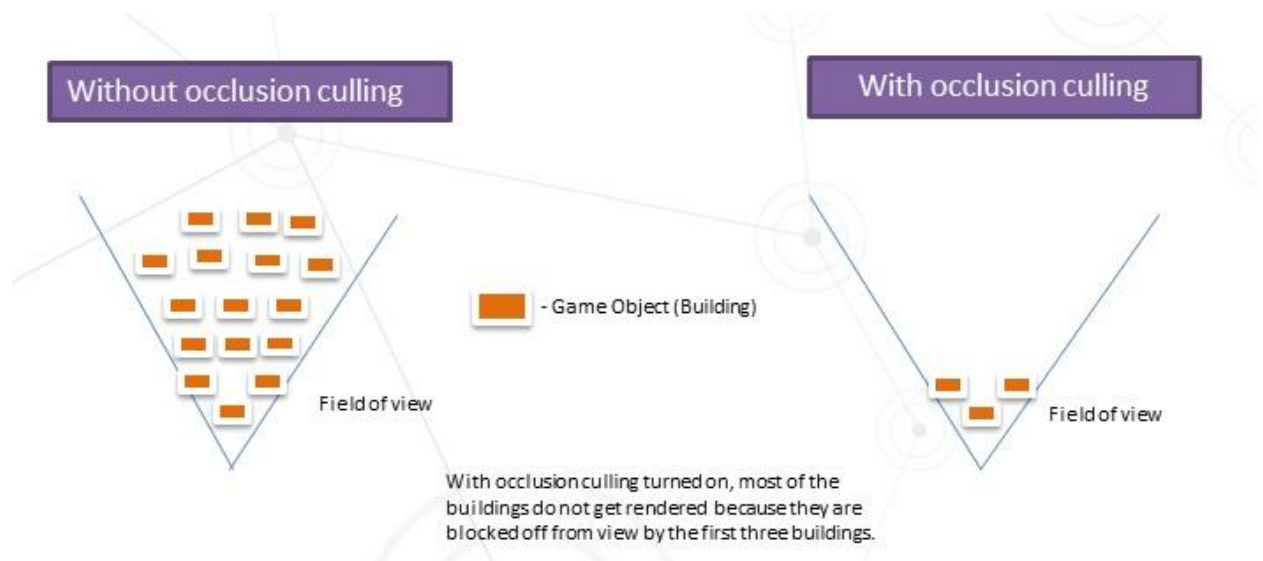


Figure 18. Occlusions explained

Occlusion culling alleviates GPU processing (and maybe CPU if the occlusion culling calculation takes less time than the driver calls saved) by sending less draw calls overall. To set occlusion culling, it helps to know some of the vocabulary Unity uses as it will help you set up your scene to do the culling.

- Occluder – Any object that acts as a barrier and prevents visibly blocked objects (occludees) from being rendered.
- Occludees – Any object that will not be rendered to the screen because it is blocked by an occluder.

Most objects that you come across will have the potential to be included as both an occludee and an occluder depending on the camera orientation and game boundaries. It's recommended to go through your entire scene to multi-select any objects that should be included in occlusion culling calculations and mark them as "Occluder Static" and "Occludee Static" (Figure 19).

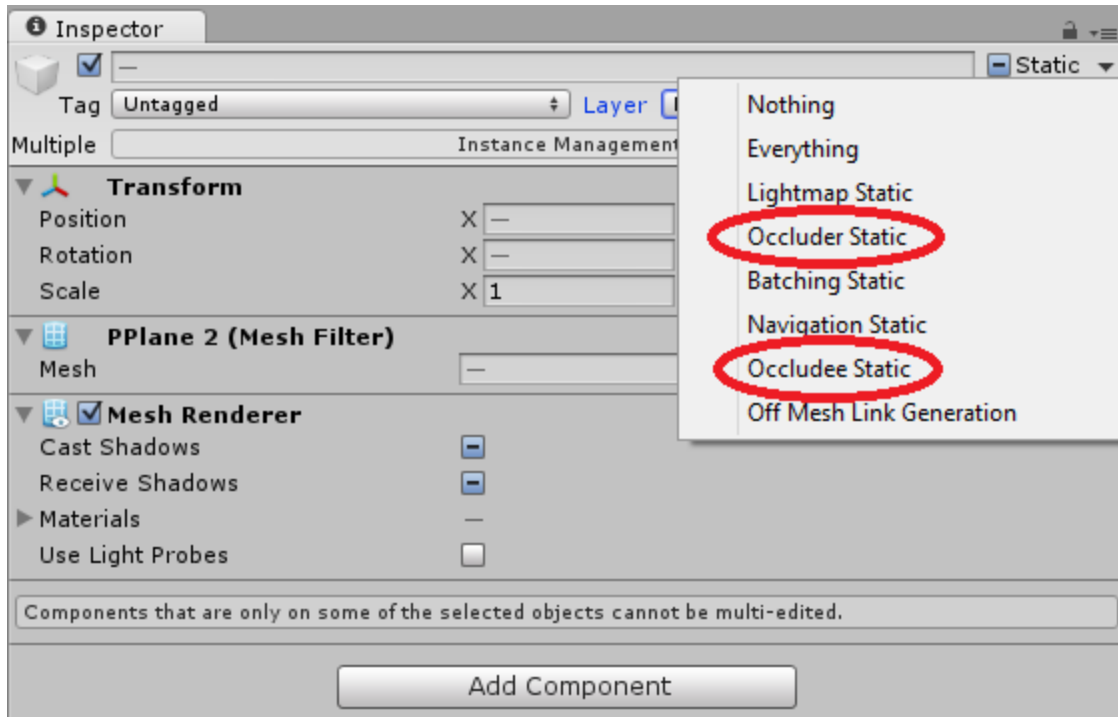


Figure 19. *How to set occluder and occludee in Inspector*

The last step required to complete the process of adding occlusion culling is to bake the scene. This can be done by opening the Occlusion Culling Window located by selecting Window->Occlusion Culling. You will see a window like the one shown in Figure 20 with different baking techniques that range from higher performance / lower accuracy to lower performance / higher accuracy. You should use the “minimum effective dose” technique for your app.

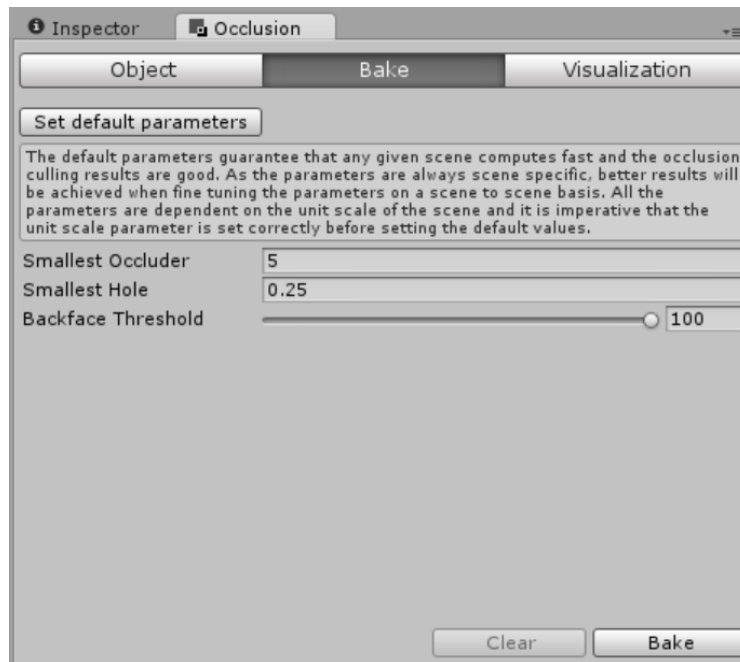


Figure 20. *The Occlusion window and Bake button*

When setting up your occlusion culling system, set your **occlusion areas** carefully. By default, Unity uses the entire scene as the occlusion area, which can lead to frivolous computation. To make sure that the entire scene isn't used, create an occlusion area manually and surround only the area to be included in the calculation.

Unity allows you to visualize each part of the occlusion culling system. To view your camera volumes, visibility lines, and portals, simply open the occlusion culling window (Window > Occlusion Culling) and click on the Visualization tab (Figure 21). You will now be able to visualize all of these components in the scene view.

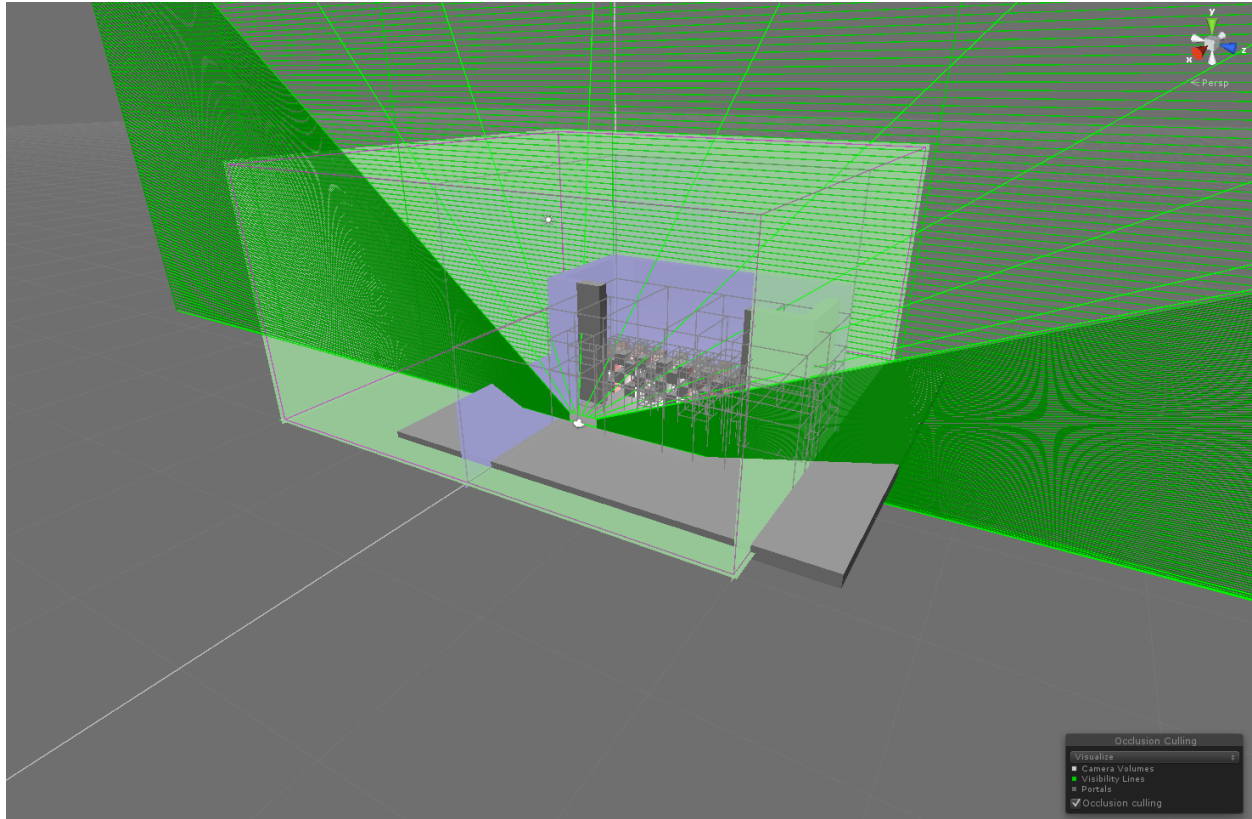


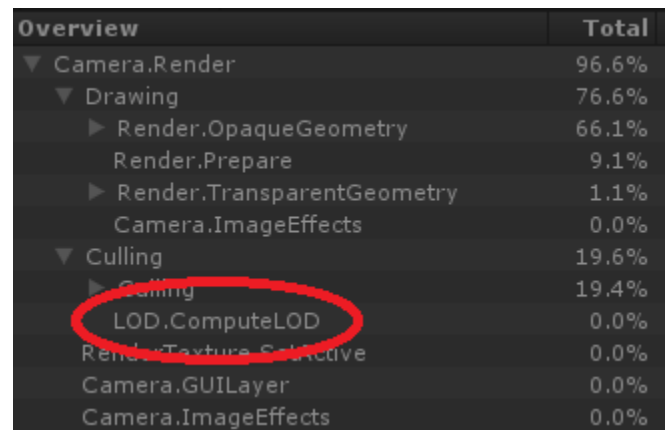
Figure 21. *Occlusion Culling visualization in the scene view*

For more info: <http://docs.unity3d.com/Manual/class-OcclusionArea.html>

LOD: Level of Detail (Pro Only)

The level of detail (LOD) component allows game objects to switch out meshes at varying levels of detail depending on the object's distance from the camera. Using this LOD feature can dramatically reduce memory requirements for a frame with very little impact on visual fidelity. Adjusting LOD alleviates the input assembler and the vertex shader by supplying less geometry on lower LOD levels.

You can verify that the LOD feature is actually being used by checking the Unity profiler. To do this, open up the 'CPU usage' profiler and navigate down to Camera.Render > Drawing > Culling and check to see if "LOD.ComputeLOD" is displayed (Figure 22).



Overview	Total
▼ Camera.Render	96.6%
▼ Drawing	76.6%
▶ Render.OpacityGeometry	66.1%
Render.Prepare	9.1%
▶ Render.TransparentGeometry	1.1%
Camera.ImageEffects	0.0%
▼ Culling	19.6%
▶ Culling	19.4%
LOD.ComputeLOD	0.0%
Render.Texture.Solve	0.0%
Camera.GUILayer	0.0%
Camera.ImageEffects	0.0%

Figure 22. Verifying LOD usage in the Unity Profiler

You can also verify that the correct model is being used by taking a frame capture using GPA, selecting the model's corresponding draw call, and then clicking the Geometry tab. This will give you a visual representation of the actual model's geometry submitted along with other useful stats, such as the vertex count. You can verify that the vertex count matches the desired model to be used at the camera distance of the capture.

Level of Detail is normally vertex bound. Bottlenecks can occur from too much computation per vertex. Using the mobile version of Unity shaders can help alleviate significant computations per vertex. When objects are small or far away, limit the vertex count in the LODGroup when those details are not needed (Figures 23-26).

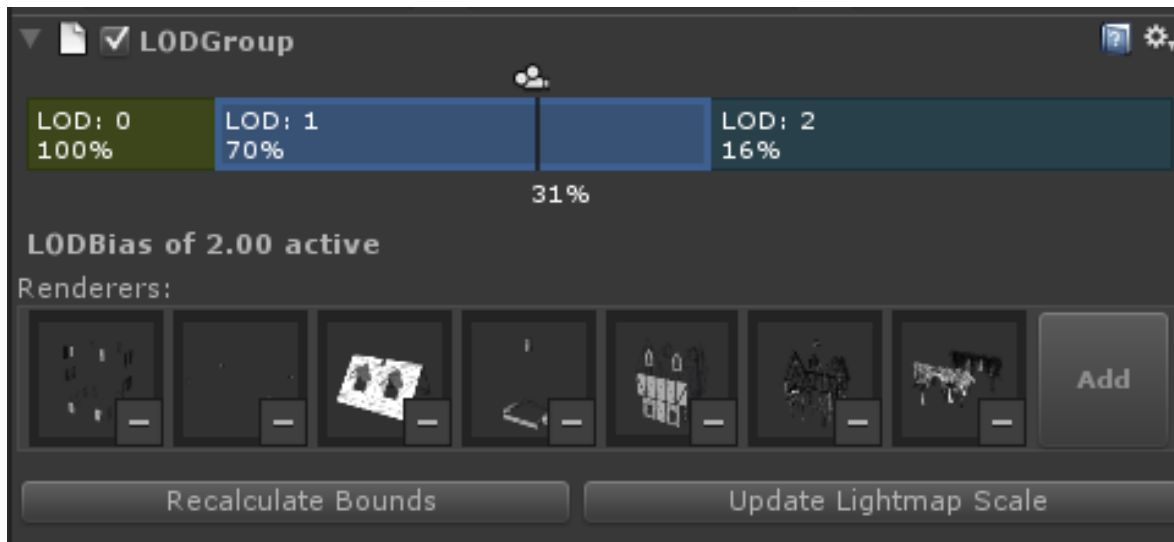


Figure 23. To add an LOD component to a game object click Component->Rendering->LOD Group

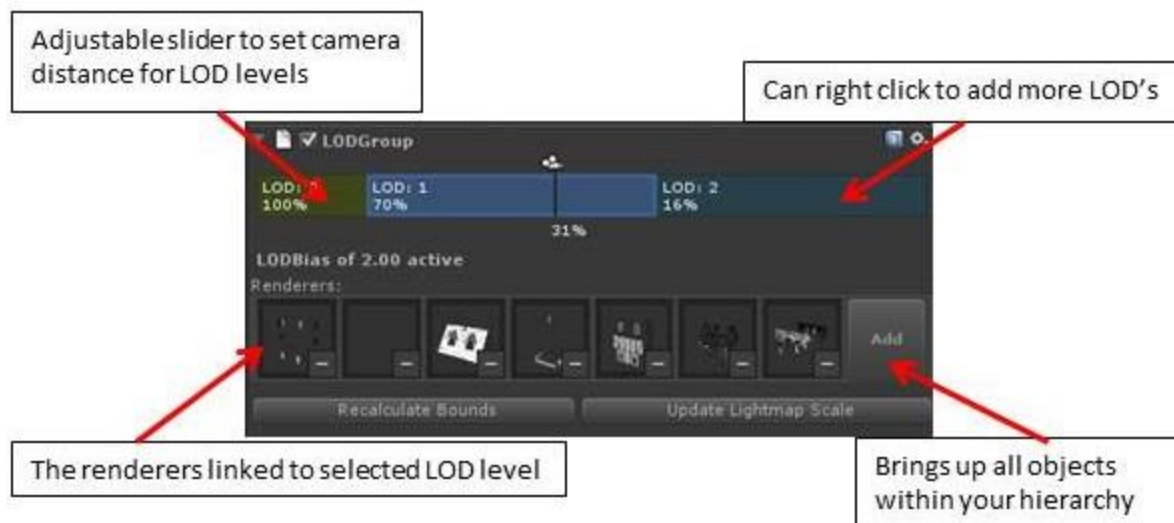


Figure 24. Adjusting in the LOD Group



Figure 25. High Quality

Figure 26. Low Quality

Shadows

Shadows can be a very significant GPU performance hog. To view just how many system resources your shadows are using, check out the Profiler > GPU > Shadows section. There are various optimizations you can do to maximize shadow performance depending on the layout of your scene. For instance, if most of the scene shadows are caused by a directional light, it can significantly help to reduce the Shadow Distance setting in Project Settings > Quality (Figure 27). Shadow distance is most closely associated with fragment shader performance. In a GPA frame capture you can view the fragment shader execution unit stall / active metrics when selecting a draw call that samples from the shadow map. The shadow distance value can also be set in code on the fly. For point lights, adjusting the shadow resolution helps alleviate memory bandwidth bottlenecks as this can be very costly on mobile.

Here is a brief overview of each of the shadow options available under Project Settings > Quality ([for more information check out the Unity Quality Settings guide](#)):

- **Shadow Filtering** – Method used to filter shadows
 - Hard - When sampling from the shadow map, Unity takes the nearest shadow map pixel
 - Soft – Averages several shadow map pixels to create smoother shadows. This options is more expensive, but creates a more natural looking shadow
- **Shadow Resolution** – Resolution of the generated shadow map
 - Can significantly affect performance if using many **point / spot lights**
- **Shadow Projection** – Method used to project shadows
 - Stable – Renders lower resolution shadows that do not cause wobbling if the camera moves
 - Close Fit – Renders higher resolution shadow maps that can wobble slightly if the camera moves
- **Shadow Cascades** – How many cascade levels to use
 - Can significantly impact **directional light** performance
- **Shadow Distance** – Max distance from object that shadows can project
 - Can significantly affect fragment shader performance if using **directional light**
 - Can be changed on the fly via script

Performance results will vary as the GPU usage is dependent on the scene and how many objects are casting/receiving shadows. As always, it is important to use the lowest quality

settings required to achieve the desired look. It is generally recommended to change the default shadow distance to a lower value.

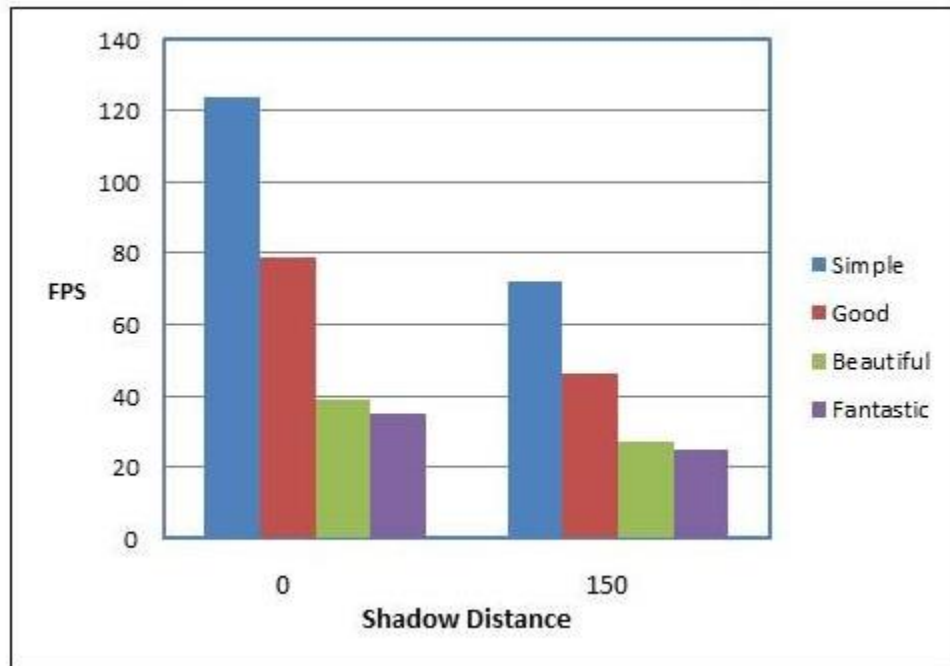


Figure 28. FPS based on shadow distance within Unity Bootcamp demo.

You can view the shadow map generated by Unity by taking a frame capture of the scene and then viewing it in the GPA Frame Analyzer. Go to the final render target and navigate to the Textures tab to view the shadow map (Figure 29).



Figure 18. *Shadow map generated as seen in GPA*

Render Ordering

There is a concept in graphics programming called **overdraw** that refers to a pixel being unnecessarily overdrawn multiple times, thereby wasting graphics resources. Unity offers a way to define the order in which to render different models called the **Render Queue** property. The RenderQueue property is a numerical value that can be set through a GameObject's Material.

To see why this can be very beneficial, picture a floor with a bunch of objects on it. Let's say the ground is rendered first, touching every pixel on half the screen. Next, all of the objects are rendered on top of that ground. This is a lot of unnecessary work. In this example, any pixel that is touched by an object is being drawn twice.

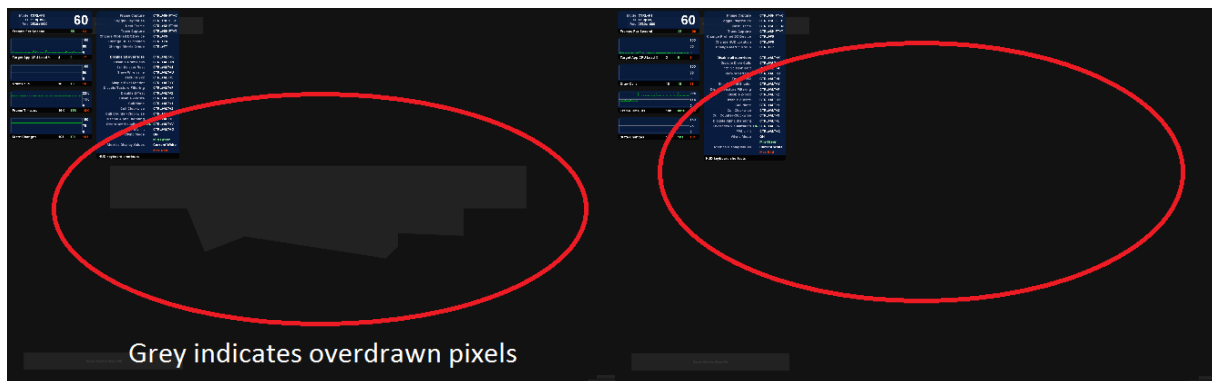


Figure 30. *Overdraw visualization using GPA System Analyzer. Grey areas indicate overdraw. Left side is not using the RenderQueue to order and right side is.*

Lightmapping

Lightmapping is the concept of first baking all scene lights into a light map (a texture with pre-calculated lighting data) to be sampled from shaders for objects in your scene instead of dynamically calculating lighting values in a shader every frame. Using this technique can result in some serious performance improvements when memory bandwidth is not the main bottleneck. Unity offers the ability to bake your lights into scenery in this fashion.

Unity also enables you to bake light maps for dynamic objects using **Light Probes**. Light probes are points that you can place in your scene that will sample surrounding lighting and shadowing conditions (Figure 31). The texture generated by these light probes is sampled when a dynamic object passes by those probes points. The light/shadow values used on passing objects is interpolated between all surrounding probes. Probes placed around a scene should form a 3D volume and should be scattered more densely around areas that your dynamic object could potentially cover.

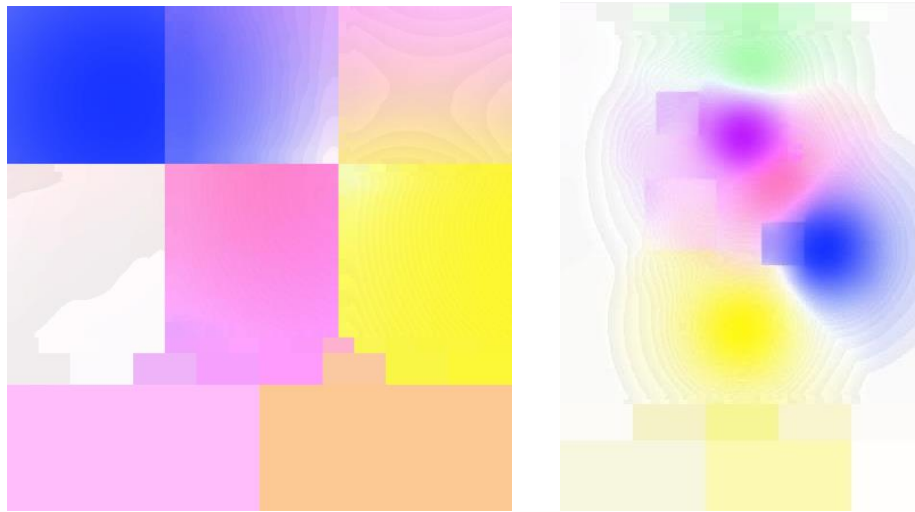


Figure 31. Left: Static light map generated by Unity. Right: Dynamic light map generated using light probes. (As shown in the GPA Frame Analyzer.)

To bake light data, mark all static geometry as static in the inspector (same checkbox as mentioned in Occlusion Culling section) and place your light probes around the scene to form a 3D volume covering all potential paths of dynamic objects where light data is to be received. Once objects are marked and light probes are placed, open the Lightmapping window via Window->Lightmapping and click the “Bake Scene” button (Figure 32).

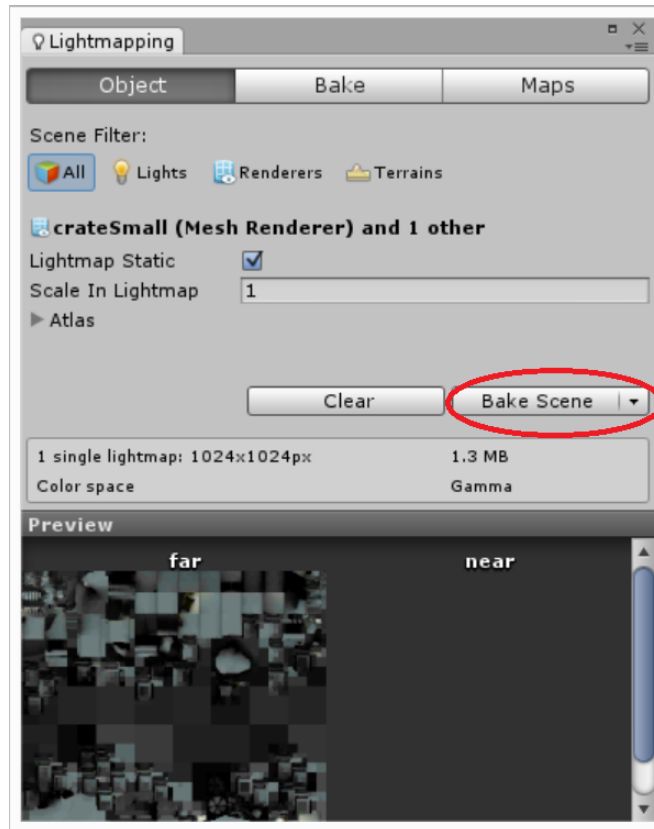


Figure 32. *Baking a scene from Lightmapping*

You will see a small loading bar at the bottom right portion of the window. When the baking is complete, you're done! You can remove/disable all dynamic lights that aren't needed from your scene. The baked light will automatically be applied.

Using Simple Colliders Instead of Mesh Colliders for Complex Models

It's important to use a combination of **primitive colliders** for complex objects that can be collided with instead of just throwing **mesh colliders** on everything. A primitive collider is a simple 3D shape (capsule, sphere, box, etc.); while a mesh collider takes the form of the mesh that you are trying to enable collision on. When possible, choose primitive colliders over mesh colliders (Figure 33).

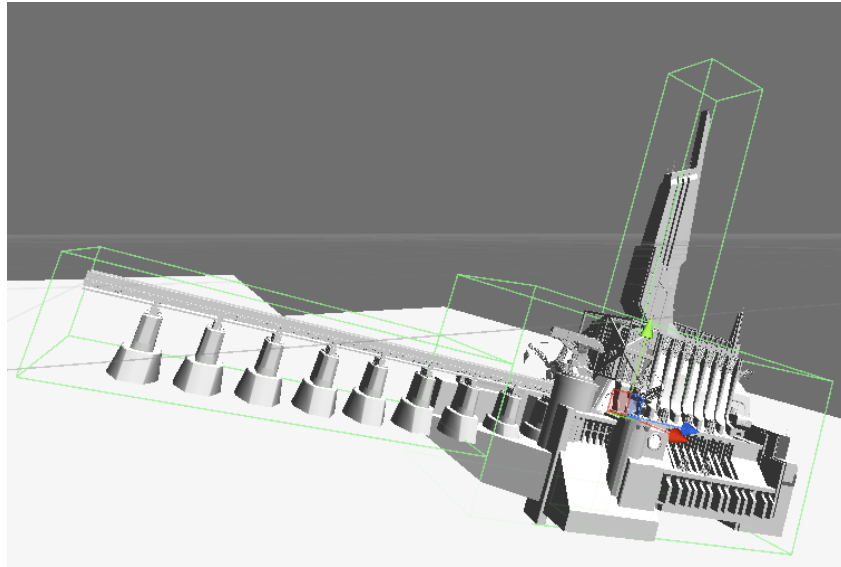


Figure 19. *Three primitive colliders used for this massive structure. Far greater performance than using the mesh collider.*

Compressing Textures

Unnecessarily high resolution textures can easily become a bottleneck in mobile apps. It is always worth verifying that the textures you use in your scene are in compressed format and that you are selecting the Generate Mip Maps checkbox to enable mip mapping (Figure 34). Mip mapping is similar to the LOD system discussed earlier, but for texture resolutions. If any object you are drawing is super far from the camera, it's not necessary to use a 1024 x 1024 texture to get the detail, as the object may only be covered by a few pixels.

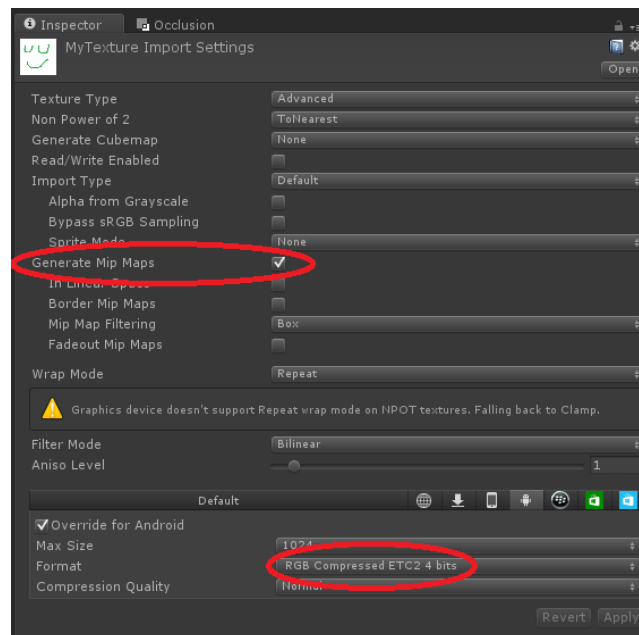


Figure 34. *Compressing textures and generating mips on the Inspector tab for selected texture*

You can verify that your textures are being compressed and mips are being generated by taking a frame capture in GPA and looking at the **Texture** tab for the draw calls you wish to investigate (Figure 35). It should be mentioned that generating mip maps can actually result in poorer performance in some cases due to generating additional data. As always, verify this option in your app.

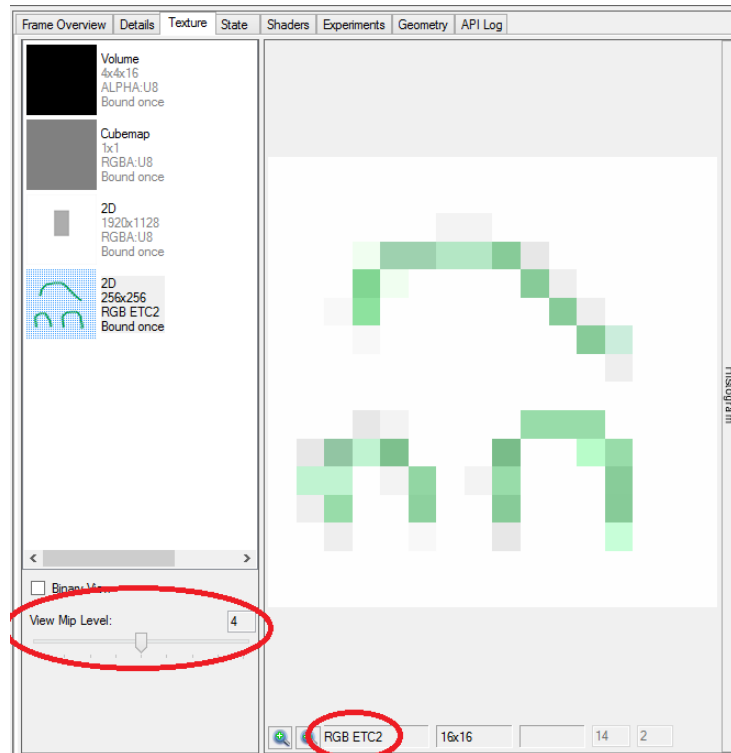


Figure 35. Viewing the 4th MIP level and format of the texture in the Frame Analyzer

Mobile Stock Shaders

When using stock shaders in your Android app, it is usually worth switching to the mobile version counterparts provided by Unity to ensure that lower precision floating point values and mobile specific optimizations are being utilized. Try them out in your app by selecting your materials and finding the Mobile section in the drop down menu (Figure 36).

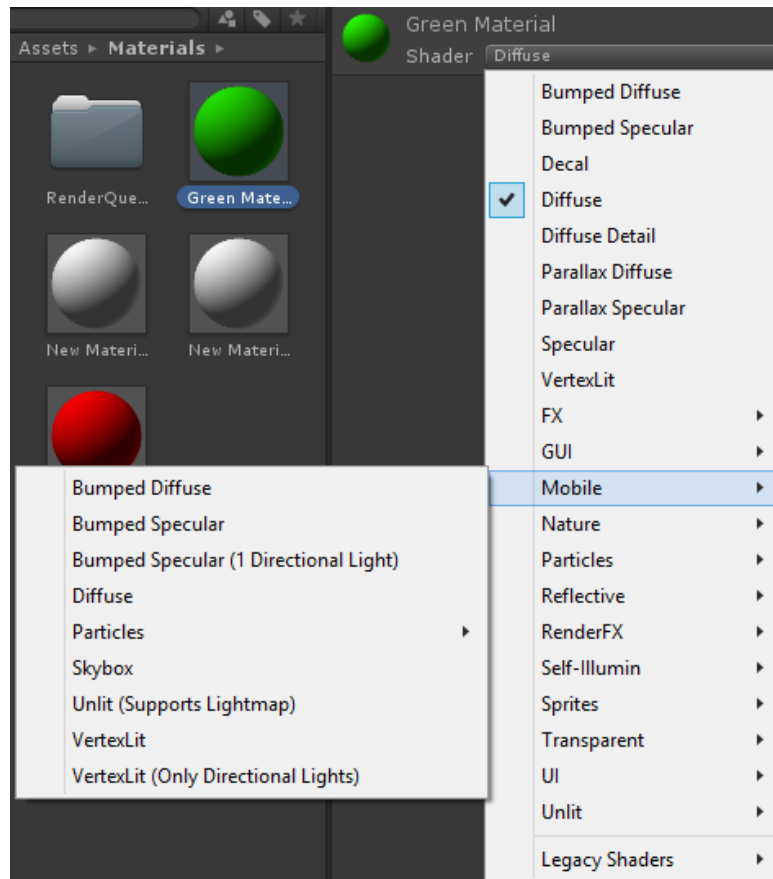


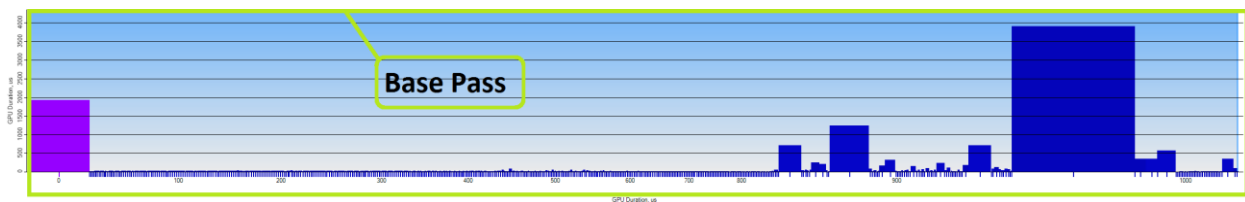
Figure 36. Stock shaders in Unity

Selecting the Optimal Render Path

Selecting the optimal render path for your app is highly dependent on what you are trying to do. The following is a brief overview of each of the render paths that Unity provides with their pros and cons. Hopefully with this information, you will know which path to choose based on your project specifics, but as with the rest of these optimizations, testing each option is always the way to go! A great way to see which rendering path is optimal for your game is to write a switch in code that will change rendering paths at the push of a button, and then observing the effect of each path in real time via the Unity Profiler and GPA.

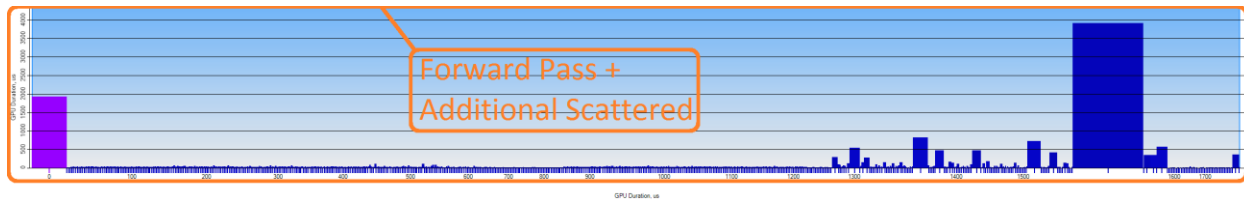
Vertex-Lit Rendering

- Best for mobile
- Per-vertex lighting only
- Real-time shadows not supported
- Everything done in the “base pass”



Forward Rendering

- Lighting done with a combination of per-pixel, per-vertex, and spherical harmonic techniques
- Real-time shadows supported
 - Breakdown
 - Base pass
 - First per-pixel light reserved for brightest directional light.
 - Next, up to 3 other per-pixel lights that are marked as **important** are drawn. If no lights are marked as important, the next 3 brightest from the scene are chosen. If there are more lights marked as important that exceed the “**per-pixel light count**” setting value in **Project->Quality**, then these are done in additional passes.
 - Next, up to 4 lights are rendered per-vertex.
 - Finally, remaining lights are drawn using spherical harmonic calculations (these values are always calculated, so essentially free on GPU).
 - Additional passes
 - An additional pass done for each per-pixel light remaining after the base pass.
 - An additional pass done for semi-transparent objects.



Deferred Lighting (Pro only)

- Lighting performance is unrelated to scene complexity
- Trade heavy lighting computation (FLOPS) for more memory usage leading to a higher chance of being memory bound
- Real-time shadows supported
- Semi-transparent rendering not supported directly. These objects are drawn with an additional forward pass.
- Breakdown
 - Base pass
 - Renders all objects, saving normal x, normal y, and normal z into RGB render target channels and specular power into the alpha channel for use by the lighting pass.
 - Lighting pass
 - Uses the texture generated by the base pass to do per-pixel lighting calculations. Unity will pass in geometric bounding volumes to Z-test against, making it easy to detect occluded / partially occluded lights.
 - This pass will generate a single texture with RGB channels holding diffuse lighting values and the alpha channel containing the monochrome specular color.
 - Final pass
 - The final pass draws all objects again, using the texture generated by the lighting pass to apply lighting to each object.
 - Additional passes
 - Semi-transparent objects require additional forward passes.

Static Batching (Pro Only)

Batching combines similarly shaded geometry into single draw calls. This leads to less driver overhead that would normally accompany the extra calls. Static batching is:

- Combines objects sharing the same materials into single draw calls
- Turn on static batching – The setting is located by selecting Edit->Project Settings->Player under the Other Settings tab (Figure 37)

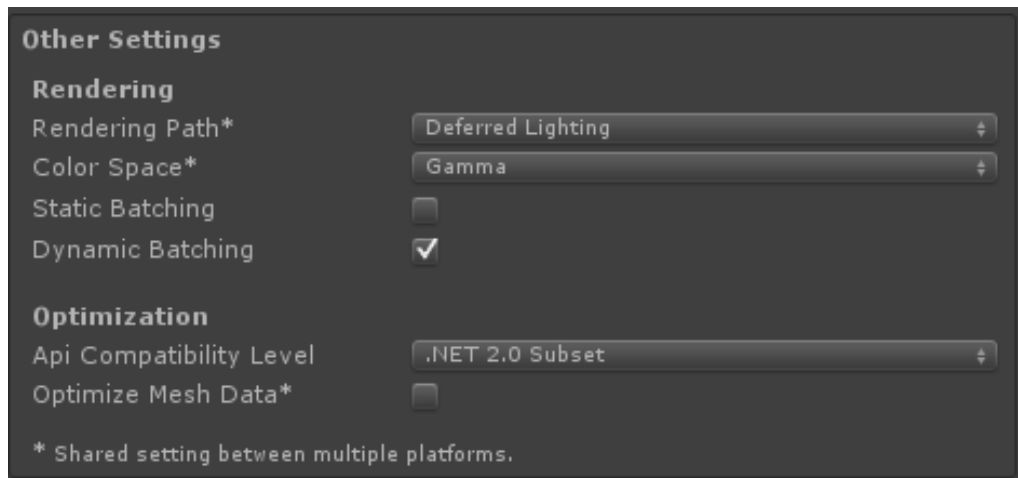


Figure 20. Static and Dynamic Batching checkboxes under Rendering in Other Settings

- Check the static checkboxes for all game objects that will not move.

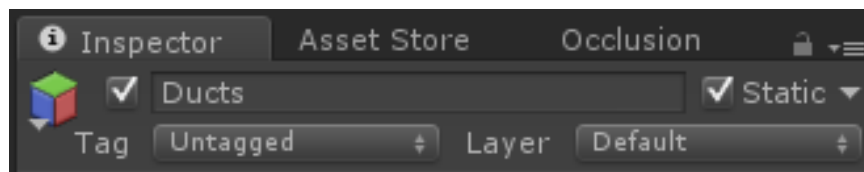


Figure 21. Batching checkmarks based on object

- Make sure you use `Renderer.sharedMaterial` over `Renderer.material` to keep material shared.
- When possible, combine multiple textures used by the same material (Diffuse, Bumped Specular, etc.) into a texture atlas to increase the number of batched objects.

Most of the time, static batching will provide a tremendous benefit to your application, but there can be situations where it is best not to use it. If you need to lower your memory usage, it can be disadvantageous to use static batching. Even objects that share geometry data will have to pack a copy of each instances vertex / index buffers into the draw call's vertex and index buffers. Unity gives an example of when static batching can go wrong with a dense forest scene. In this extreme situation, each tree with the same material is packed into these buffers before issuing the call and that can cause some serious performance issues.

Dynamic Batching

Dynamic batching is the same concept as static batching but instead batches draw calls for dynamic objects (objects that move).

- Dynamic batching is only applied to objects that have less than a total of 900 vertex attributes (subject to change).
- Objects receiving real-time shadows will not be batched.

HDR – High Dynamic Range

If you have a scene with HDR effects like Bloom and Flare using deferred rendering, you will see a large decrease in draw calls by checking the HDR box in the Camera Settings. It's important to note that each camera has its own HDR checkbox. It's best to use HDR when using DirectX 10 or better with deferred rendering. HDR is most closely related to the fragment shader. Follow these guidelines:

- Use Deferred Rendering
- Check the HDR box (Figure 39)

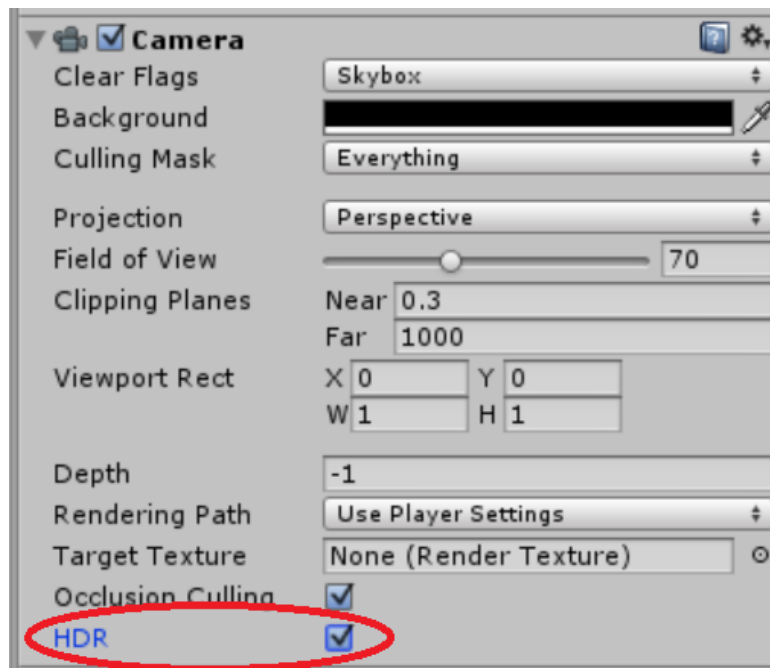


Figure 39. HDR option in Camera settings

Splitting the Binary

The latest Unity releases have the ability to produce a Fat Binary or split the binary into separate ARM and x86 portions. You can use the same process to choose either x86 or ARM to test various aspects of a deployment. Evaluating compression, code, and other specifics allows you to troubleshoot or even benchmark your builds.

Building FAT APKs does not significantly increase the binary size. You can build slim binaries by simply choosing x86 or ARMv7; however, it would be necessary to maintain two separate builds.

In Player settings:

1. Open/expand other settings .

2. In Configuration, find Device Filter and choose: FAT (ARMv7+X86). See Figure 41.

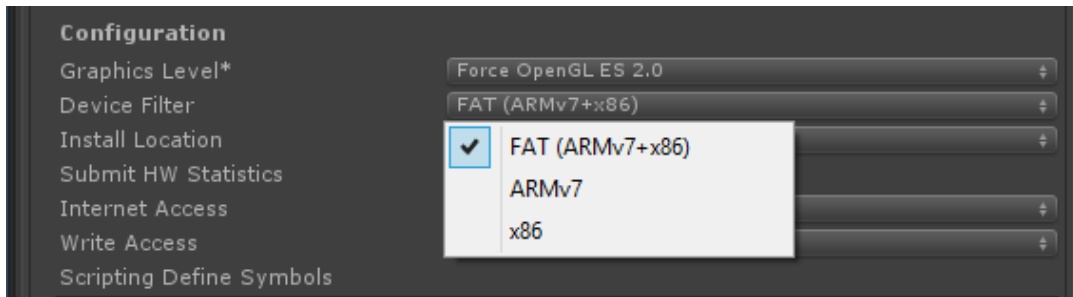


Figure 22. Configuration in Other Settings. Three options shown in Device Filter.

3. Choose build (on Build Settings screen) to begin the process of creating the selected binary (Figure 42).

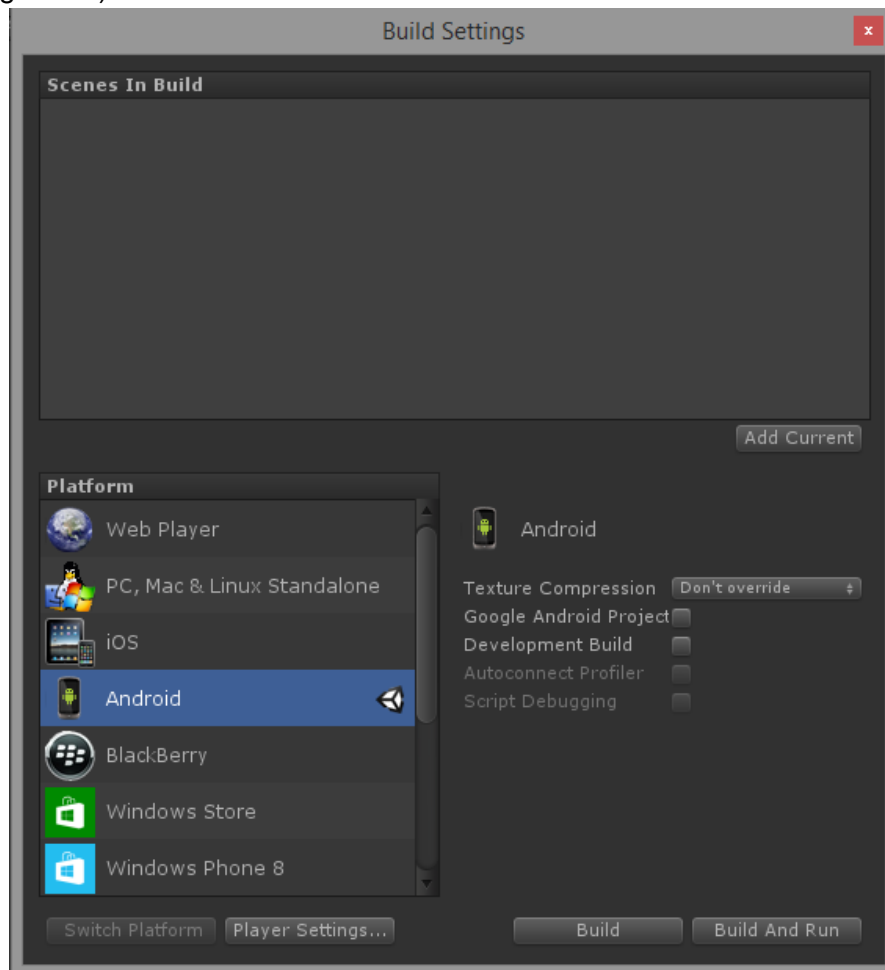


Figure 23. Choose Build to begin the process of creating the new binary

That's it! You are now supporting x86 in your Unity game deployments.

A special Unity x86 developer page is available at www.intel.com/software/unity for additional support.

Conclusion

Optimization is a job in and of itself for achieving high levels of performance from your graphic intensive game. Combinations of the above techniques can help you gain significant ground. Using these tools in the base- or Pro-level versions of Unity will allow you to dive deeper and make further adjustments.

Resources

Keyframe Animation: https://www.youtube.com/watch?v=BVH-CCLzD_E

Reducing textures and sizes: <http://docs.unity3d.com/Manual/ReducingFilesize.html>

Shadows: <http://docs.unity3d.com/Manual/Shadows.html>

About the Authors

Cristiano Ferreira is a software engineer working in Intel's Developer Relations Division with a specialization in mobile games and graphics. Cristiano helps game developers give their customers the greatest experience possible on Intel hardware.

Steve Hughes is a Senior Application Engineer with Intel, focusing on support for game development on x86 platforms from desktops and tablets to phones. Prior to Intel, Steve has 12 years of experience as a game developer, working on all aspects of game development for a number of companies.

Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL

INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: <http://www.intel.com/design/literature.htm>

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

Copyright © 2015 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.

OpenCL and the OpenCL logo are trademarks of Apple Inc and are used by permission by Khronos.