Unity* Software Performance Optimizations for Games: Best Practices

This article walk-through illustrates common performance issues in games, as well as some helpful optimization methods. We will show how to optimize draw calls, level of detail (LOD), batching, light baking, and occlusion culling.

When talking about performance there are two main things to consider in a game, the GPU cycles and the CPU cycles. They work together to generate the desired frame rate. The fill rate and high resolutions are based on the available memory bandwidth of the game as well as if the game is run with high amount of details such as ultra-high quality resolution. These details are handled by the GPU. These bottlenecks are more often found in computers with less memory. In contrast, the CPU handles rendering the draw calls.

The considerations for optimizations are resolution, fill rate and the draw calls. There are other bottlenecks in the game to consider such as physics scripts (especially if those scripts are poorly written). Physics requires a lot of calculations, so excessive physics in the game can bring down the frame rate. The Intel® Graphics Performance Analyzers (Intel® GPA) tool and also the Unity profiler are helpful in identifying bottlenecks and how to fix them.

It's also important to see whether GPU and CPU have too many vertices to process. CPU handles the vertices that interact with the object and GPU handles the Vertex shaders. Too Many vertices in the scene can cause the shader to have trouble writing the specific color to the material in that object. These can drag down the frame rate in a game.

In short, this paper discusses the best performance optimizations for designing games using Unity.

Optimizing Geometry

When creating game assets, especially for 3D games, there are some basics to consider.

First, you shouldn't use more triangles than absolutely necessary. If there are many triangles that are not contributing to the shape, then they may not be necessary. If there are lots of edges, triangles, and quads that don't contribute to the shape of the object, then we can try removing them to further optimize the geometry.

Additionally, the vertex count is the primary consideration for CPU and GPU cost. Vertices and triangles are dependent. For A game object with vertices when it is actually created, we are going to see an increased vertex count in the game because as it reads of the UV splits and the smoothing splits. More vertices probably won't impact the FPS significantly, however it's always better to optimize the geometry, keeping the rules of the geometry for games

Batching

Excessive draw calls are one of the major CPU bottlenecks for frame rate. This can lead to significant performance degradation. The more objects on the screen, the more draw calls are made.

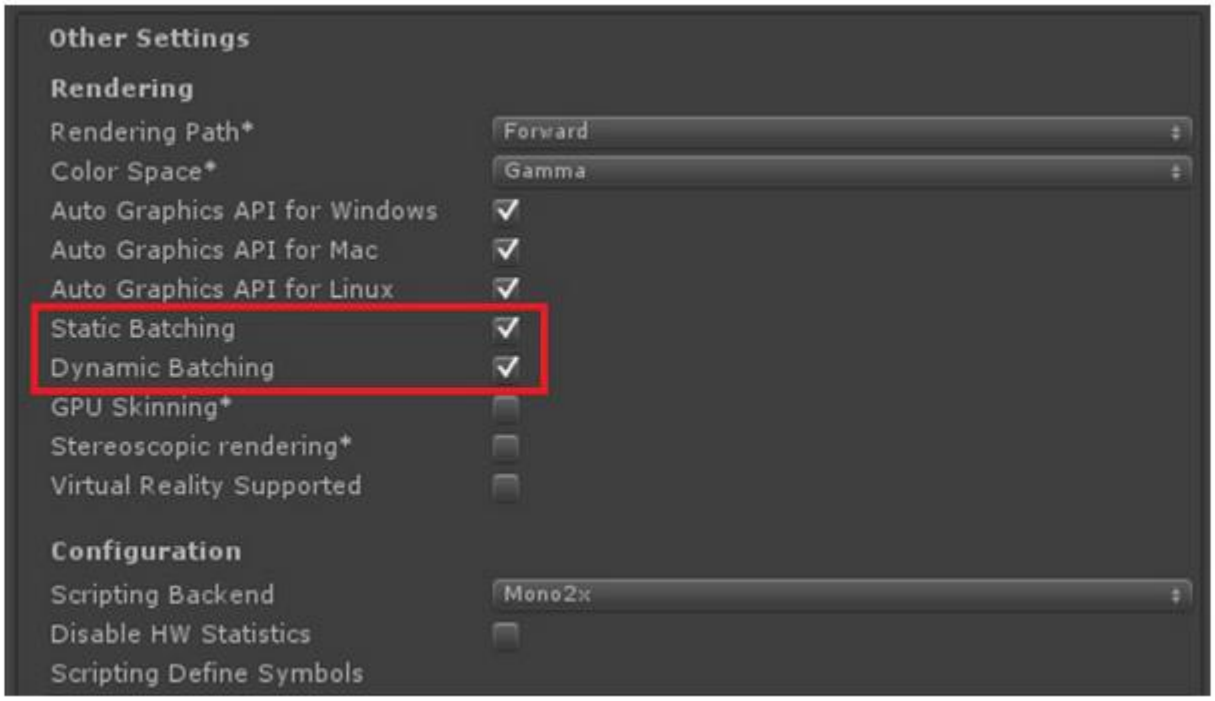
Batching combines game objects into a single draw call. You receive the best benefits from batching when you plan which objects are batched together for single draw calls.

Unity's batching mechanism comes in two forms, Static and Dynamic. Static Batching affects static objects, and Dynamic Batching is for those that move. Dynamic Batching happens automatically, if all requirements are met (see batching documentation), whereas Static Batching needs to be created.

Static batching can be turned on by selecting Edit > Project Settings > Player. Then, under the Other Settings tab, turn the static batching on for all the objects that do not move (Figure 1). 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. Dynamic batching is the same concept as static batching but instead batches draw calls for dynamic objects.

Many objects in a scene leads to many draw calls leads to a drop in frame rate. In batching, sharing materials is very important. When two objects are using the same material, Unity automatically senses and batches them together as same material. When objects use a different material, the draw calls increase.

If it happens that we should apply multiple materials then the best methods to optimize is using the texture atlases. A texture atlas groups multiple textures of multiple objects into a single texture file. We can take multiple related textures and combine them into single texture to reduce the number of materials into one material. This can then apply that texture to the related objects to reduce the draw calls.



Other Settings

Rendering

Rendering Path*	Forward
Color Space*	Gamma
Auto Graphics API for Windows	✓
Auto Graphics API for Mac	✓
Auto Graphics API for Linux	✓
Static Batching	✓
Dynamic Batching	✓
GPU Skinning*	
Stereoscopic rendering*	
Virtual Reality Supported	

Configuration

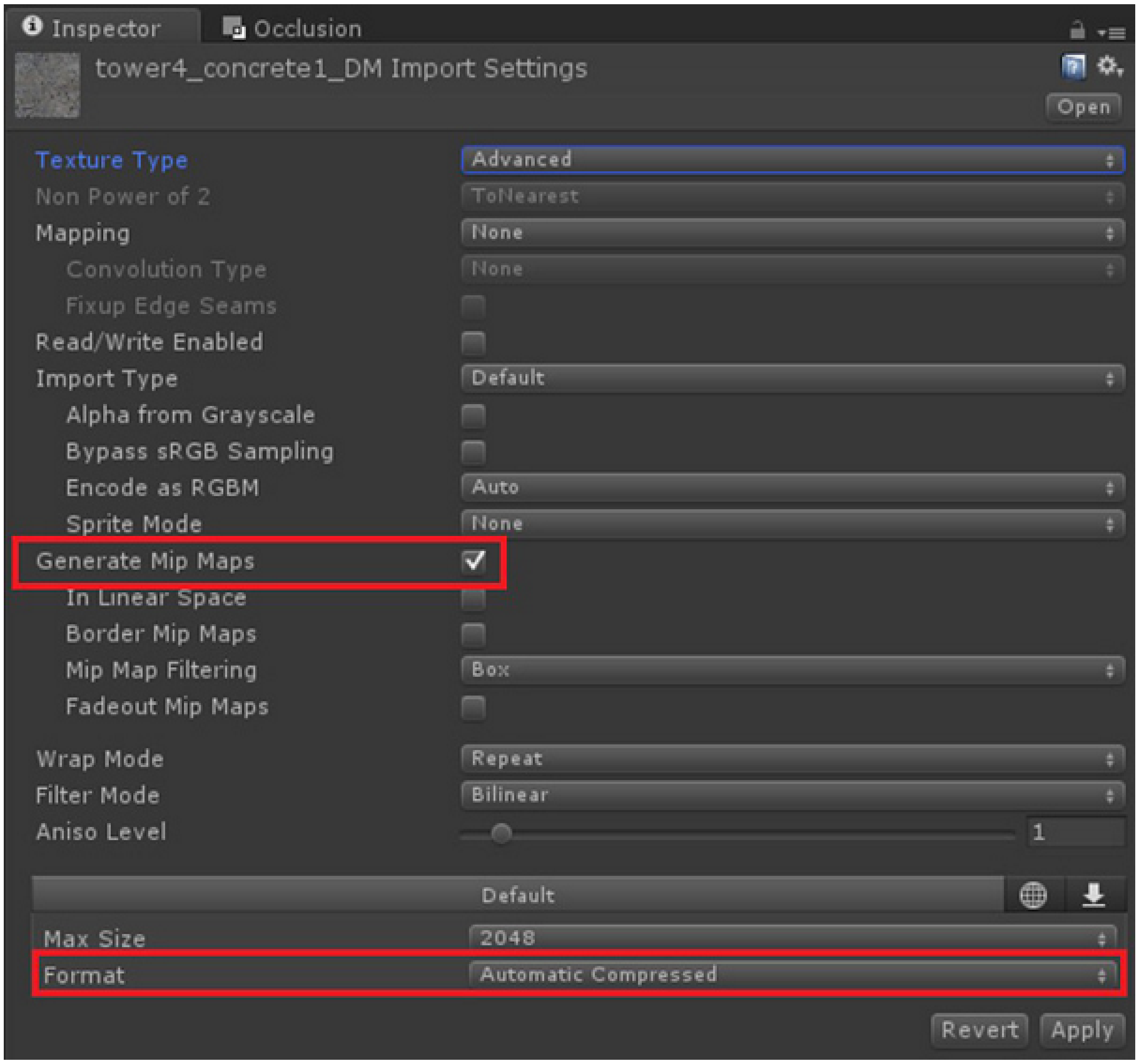
Scripting Backend	Mono2x
Disable HW Statistics	
Scripting Define Symbols	

Figure 1. Static Batching and Dynamic Batching checkboxes under Rendering in Other Settings

Textures

Setting up textures and considering the resolution of texture when creating them and its compression are important considerations. Setting up texture compression is important to create smaller size games specifically creating mobile or web player games. High resolution textures can easily become a bottleneck in mobile games and slower hardware. Texture leaves a lot of memory foot print in the game if there are going to be lots of textures, It’s important to make sure the file size is appropriate for the texture

It is always worth verifying that the textures you use in your scene are in compressed format, and that you selected the Generate Mip Maps checkbox (Figure 2) to enable Mip Mapping. Mip Mapping is a good in-game method of reducing the texture memory overhead. If any object you are drawing is far from the camera, it’s not necessary to use a higher resolution texture, as the object may only be covered by a few pixels, and these Mip maps will reduce the resolution of texture and blurring them and saving the memory.



tower4_concrete1_DM Import Settings

Texture Type	Advanced
Non Power of 2	ToNearest
Mapping	None
Convolution Type	None
Fixup Edge Seams	
Read/Write Enabled	
Import Type	Default
Alpha from Grayscale	
Bypass sRGB Sampling	
Encode as RGBM	Auto
Sprite Mode	None
Generate Mip Maps	✓
In Linear Space	
Border Mip Maps	
Mip Map Filtering	Box
Fadeout Mip Maps	
Wrap Mode	Repeat
Filter Mode	Bilinear
Aniso Level	1

Default

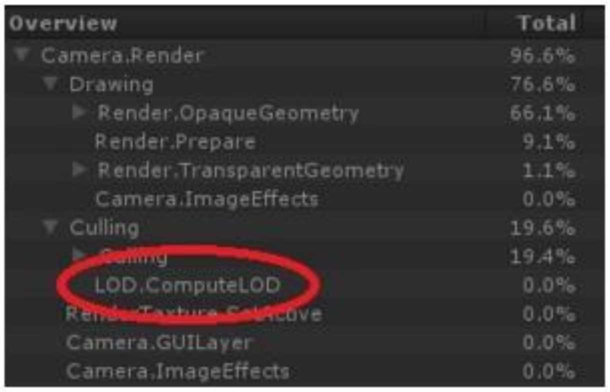
Max Size	2048
Format	Automatic Compressed

Figure 2. Compressing textures and generating mips on the Inspector tab for selected texture

Level of Detail

Level of Detail (LOD) is a feature available in the pro version of Unity. LOD allows multiple meshes to attach to a game object and provides the ability to switch between object meshes based on camera distance. This can be beneficial for complex game objects that are really far away from the camera.

You can verify that the LOD feature is 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 3).



```
Overview                          Total
▼ Camera.Render                   96.6%
  ▼ Drawing                       76.6%
    ▶ Render.OpaqueGeometry       66.1%
      Render.Prepare               9.1%
    ▶ Render.TransparentGeometry   1.1%
      Camera.ImageEffects          0.0%
  ▼ Culling                       19.6%
    ▶ Culling                     19.4%
      LOD.ComputeLOD               0.0%
    RenderTexture.SetActive        0.0%
    Camera.GUILayer                0.0%
    Camera.ImageEffects            0.0%
```

Figure 3. Verifying LOD usage in the Unity software profiler*

The LOD system reduces unnecessary vertices drawn at one time and works with the distance from the camera. If player gets far away from the camera the full details or all the vertices are not relevant. You can experiment with the Unity editor by exporting all the LODs to Unity, create the LOD group component with LODs, and observe from the inspector in Unity by sliding the camera to observe the transition between different LODs. When you switch the model you will notice the LOD objects change out based upon camera distance. 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.

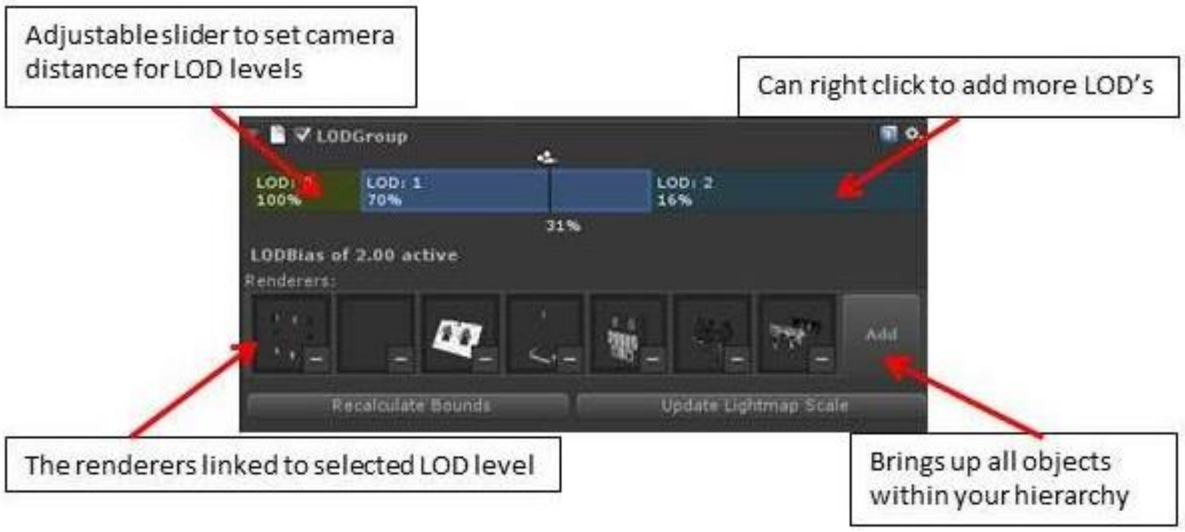

Adjustable slider to set camera distance for LOD levels
Can right click to add more LOD's
LODGroup
LOD: 0 100%
LOD: 1 70%
LOD: 2 16%
31%
LODBias of 2.00 active
Renderers:
Add
Recalculate Bounds
Update Lightmap Scale
The renderers linked to selected LOD level
Brings up all objects within your hierarchy

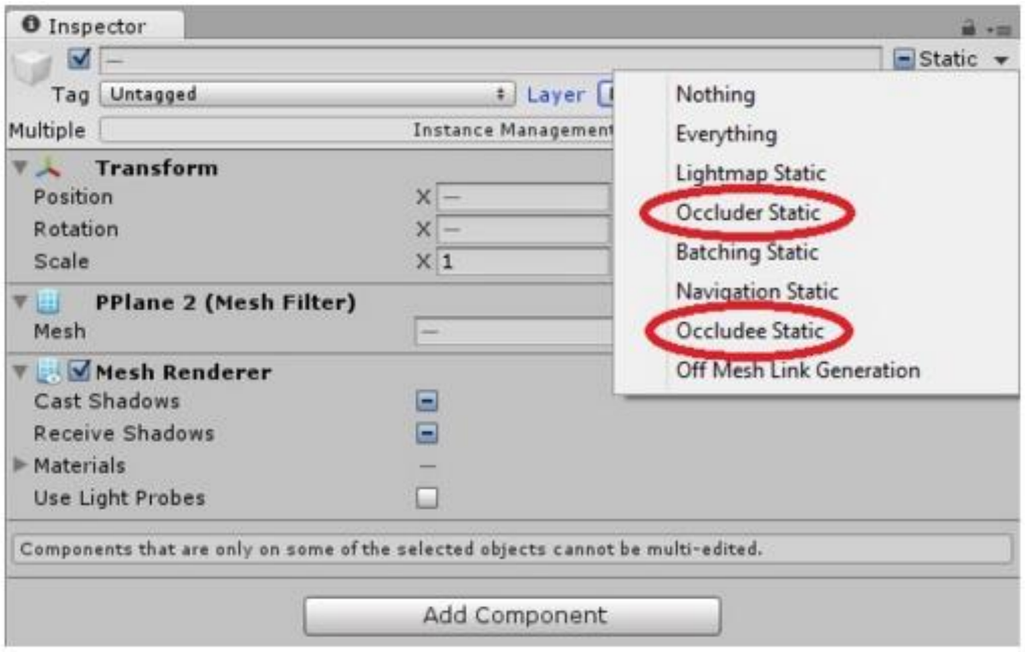
Figure 4. Adjusting in the LOD group

Occlusion Culling:

Occlusion culling is a feature available in the pro version of Unity that enables you to remove objects that are blocked by other objects. In a game when there are lots of objects that are drawn in a scene, there are many draw calls, and the objects are drawn from farthest object to closest in the order from the camera.

In cases where objects are not seen by the camera, drawing them results in wasted draw calls. In these cases, the occlusion culling mechanism turns off the objects that are not in view of camera, reducing the vertices that are not drawn, optimizing the memory usage.

In occlusion culling, the occludee is an object that will not be rendered to the screen because it is visibly blocked by another object. An occluder is an object that acts as a barrier and prevents visibly blocked objects (occludees) from being rendered.



Tag: Untagged

Scale: X 1

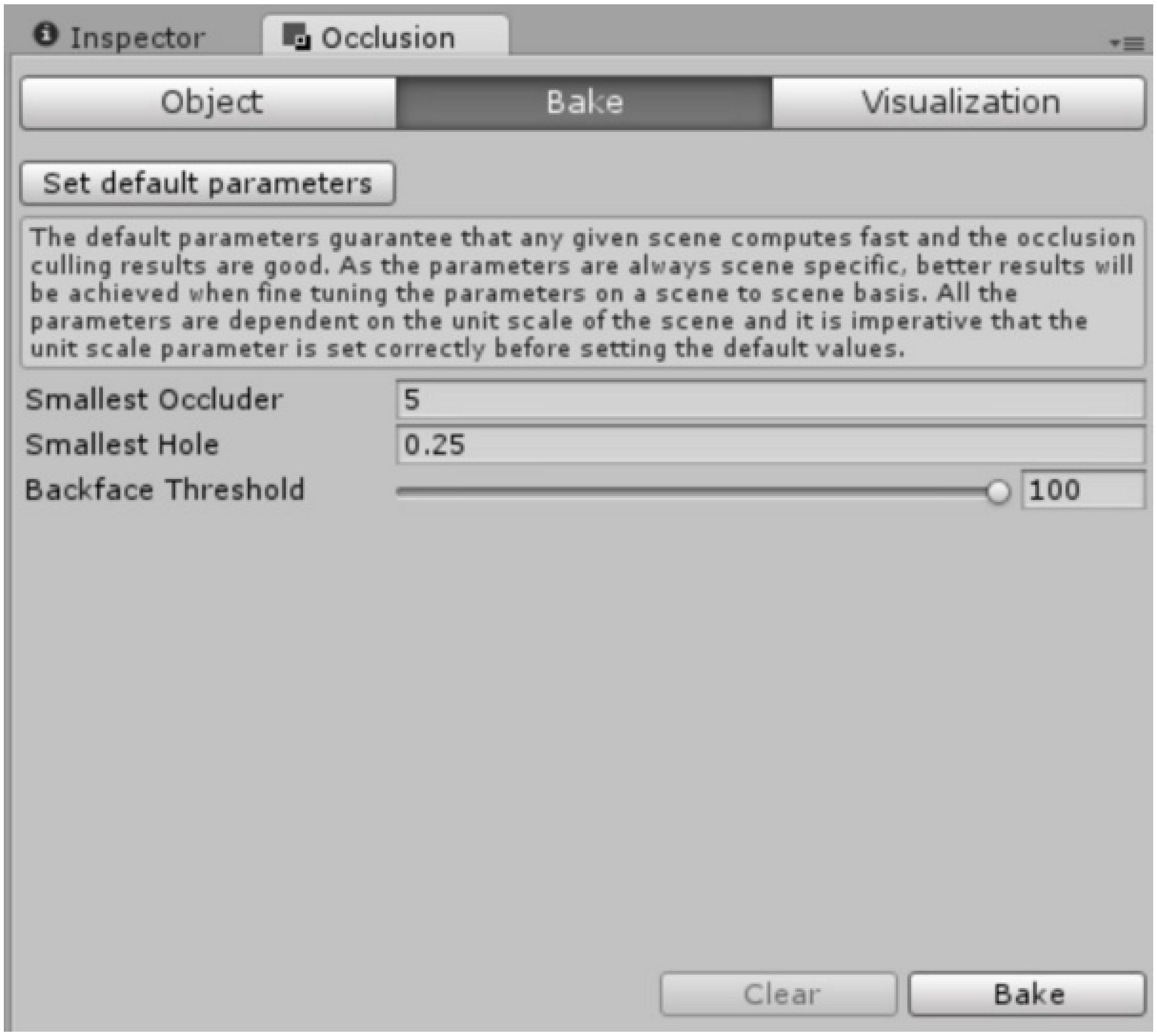
Components that are only on some of the selected objects cannot be multi-edited.

Figure 5. How to set occluder and occludee in Inspector

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 extra overhead. 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. You will now be able to visualize all of these components in the scene view.

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 (Figure 6). If the camera can see the cell, it will render objects in the vicinity of the cell. If it is outside of the viewing area, the object is occluded.



The default parameters guarantee that any given scene computes fast and the occlusion culling results are good. As the parameters are always scene specific, better results will be achieved when fine tuning the parameters on a scene to scene basis. All the parameters are dependent on the unit scale of the scene and it is imperative that the unit scale parameter is set correctly before setting the default values.

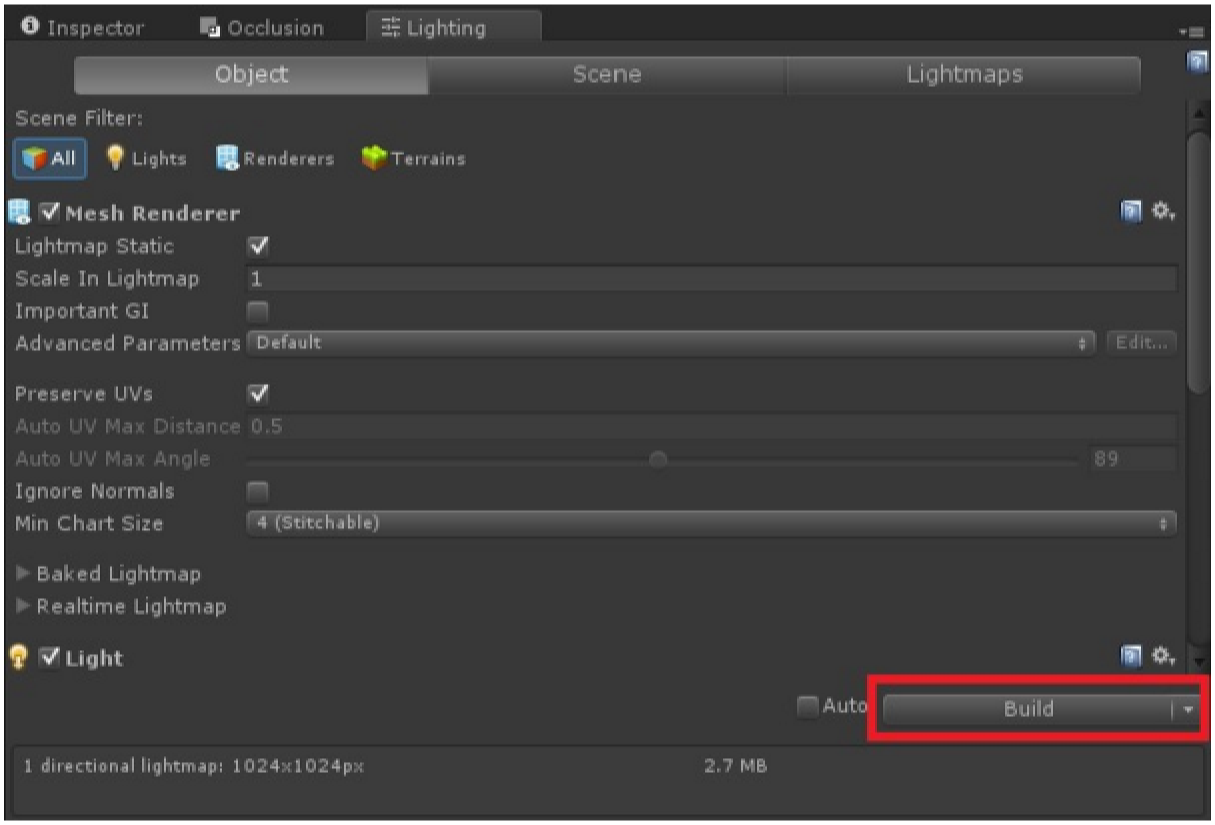
Smallest Occluder 5
Smallest Hole 0.25
Backface Threshold 100

Figure 6. The Occlusion window and Bake button

Light Mapping

Light mapping is creating a surface cache with pre-calculated lighting data (a light map). The values some from sampling shaders after baking all scene lights, rather than dynamically calculating lighting values in the shader. Using this technique can result in some serious performance improvements when memory bandwidth or sampler usage are not the bottlenecks.

Light baking is a good way to optimize the game to create level without lights. Lights in a scene create draw calls. The CPU must determine what lights are in the scene, what objects the light affects, and it has to send the details to the GPU. If there are many lights in a game it can lead to performance degradation on the CPU. We need to tell Unity the static objects in the scene. We do this by selecting Static in the inspector of all the objects that are static (Figure 7).



Mesh Renderer ✓

Lightmap Static ✓
Scale In Lightmap 1
Important GI
Advanced Parameters Default

Preserve UVs ✓
Auto UV Max Distance 0.5
Auto UV Max Angle 89
Ignore Normals
Min Chart Size 4 (Stitchable)

Baked Lightmap
Realtime Lightmap

Light ✓

1 directional lightmap: 1024x1024px 2.7 MB

Figure 7. Baking a scene using light mapping

To bake light data, mark all static geometry as static in the inspector and place your light probes around the scene to form a 3D space. This space should cover all potential paths of dynamic objects where light data is received. Once objects are marked and light probes are placed, open the Light mapping window by selecting Window->Lighting and click the "Bake Scene" button.

When the baking is complete, you can remove or disable all unnecessary dynamic lights from your scene. It is not necessary if you have your lights marked as baked in the inspector. The baked light automatically applied. An easy way to keep track of only the baked lights in your scene is to keep them under a blank Game Object for quick activation/deactivation for when you need to re-bake. Verify that your light map baking workflow mode is not set to auto if you go this route.

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 will allow you to dive deeper and make further adjustments.

References

https://software.intel.com/sites/default/files/managed/c8/14/Unity-Optimizations-for-x86-Android-V2_92.pdf

https://software.intel.com/en-us/node/542152

https://software.intel.com/en-us/android/articles/unity-optimization-guide-for-x86-android-part-1

https://software.intel.com/en-us/android/articles/unity-optimization-guide-for-x86-android-part-2

https://software.intel.com/en-us/android/articles/unity-optimization-guide-for-x86-android-part-3

https://software.intel.com/en-us/android/articles/unity-optimization-guide-for-x86-android-part-4

About the Author

Praveen Kundurthy works in the Intel® Software and Services Group. He has a master's degree in Computer Engineering. His main interests are mobile technologies, Microsoft Windows*, and game development.

Notices