# Assets Streaming学习笔记

### Mesh Complexity

For small levels we recommend no more than 500,000 triangles in view at the same time， Larger levels should have no more than 200,000 triangles

The geometry can be more complex in smaller, enclosed levels due to draw distances

being shorter and occlusion culling working more effectively. After culling, most triangles will no longer be rendered, even if technically still within the view frustum. In larger, open levels the complexity of the geometry has to be adjusted because culling is generally less effective. Keep in mind that both of these depend largely on how complex your shaders are.

Finally, it’s important to reduce the amount of small triangles on screen. Not only does this reduce the vertex count, it also reduces the amount of extra samples being rendered when MSAA is enabled.

### Shader Complexity

To get a more precise measure of shader complexity, you can select the shader in the project browser and then select **Compile and show code** for D3D. This outputs the shader disassembly and some basic statistics. Disassembly is harder to read for a human, but it makes it fairly easy to see which shaders are expensive since more complex shaders will require more operations. As a general rule, higher stats and more lines of disassembly code means the shader will be slower.

Another way to express the complexity of a shader is to run it through the Mali shader compiler. The Quest does not use a Mali chip, but the metrics returned by the shader compiler are a pretty good estimate. To do this, first **Compile and show code** for GLES3x, find the correct shader variant, and copy it to a file. Then run malisc on this file through the command line. This will show you an estimate of how many cycles it will take to execute the shader and whether the cycles are arithmetic, memory, or texture operations. A fairly basic shader with direct lighting and multiple texture maps will have roughly 60 arithmetic, 2 load/store, and 5 texture operations.

## Measuring and Reducing Draw Calls

A top priority when trying to get the most performance out of the GPU is to reduce the number of draw calls required per frame. For more information, refer to the following articles:

* [Down the Rabbit Hole, Quest Developer Best Practices](https://developer.oculus.com/blog/down-the-rabbit-hole-w-oculus-quest-developer-best-practices-the-store/)
* [Mobile Draw Call Analysis](https://developer.oculus.com/documentation/native/android/mobile-draw-call-analysis/)
* [How to Optimize your Quest App with RenderDoc (Part 1)](https://developer.oculus.com/blog/how-to-optimize-your-oculus-quest-app-w-renderdoc-walkthroughs-of-key-usage-scenarios-and-optimization-tips-part-1/)
* [How to Optimize your Quest App with RenderDoc (Part 2)](https://developer.oculus.com/blog/how-to-optimize-your-oculus-quest-app-w-renderdoc-walkthroughs-of-key-usage-scenarios-and-optimization-tips-part-2/)

如果您的场景中有 10 个具有 10 种不同纹理的对象，那么这些对象中的每一个都将是一个具有唯一数据集的唯一绘制调用。不同的网格、不同的纹理、不同的着色器参数，甚至可能是不同的着色器。但是，如果您将所有这些对象组合成一个纹理（“atlas 纹理”）并调整对象的 UV 映射以指向该单个纹理，您的绘制调用开销将会减少，因为纹理只需要绑定一次绘制所有 10 个对象。如果您可以进一步为这些对象使用相同的着色器和着色器参数，您将显着降低场景成本。如果您是 Unity 开发人员，您可以通过在所有对象上使用相同的材​​质、相同的着色器和相同的纹理来实现此目的。如果您可以将这些对象标记为“批处理静态，”您甚至可以为所有对象使用相同的网格。

We were able to reduce the number of set pass calls by forcing these meshes into the same lightmap. We did this through [Lightmap Parameters Assets](https://www.oculus.com/lynx/?u=https%3A%2F%2Fdocs.unity3d.com%2FManual%2Fclass-LightmapParameters.html&e=AT0efdjO5Paov91rwgzoV_CCP_ha5GTFIbpd4_DDOPloPjo-wh0bbrmqCpixJ_h3RcMmUV6kL56z7q8yA-ZLyRg2ze6wt3M2XvOamLShmrNYE0ZQt9l9L_RCGw9ndA8jFMuMmI-Qg4dfKXdP8rHaYg). This allows you to assign objects into a group by setting the **System Tag** parameter. You can also limit how many lightmaps the objects using the lightmap parameter asset are allowed to generate. We forced the terrain into lightmap 0 and the rest of the meshes into lightmap 1. For our purposes, this created good looking lightmaps. It also reduced the number of setpass calls from 148 to just 16. The total number of batches were reduced from 156 to 94.

## Mesh Batching and LOD Generation

 Mesh batching refers to combining meshes into one big mesh. Static batching refers to Unity’s batching of draw calls. A static batch has a single set pass call but can have multiple draw calls. A batched mesh only has one draw call.

The shaders also use no direct lighting. The lighting comes entirely from the lightmaps. Note that lightmaps can break static batches, so it’s important that meshes use the same lightmap when possible. It’s also important that all LODs of a mesh share the same lightmap. This saves in memory usage (fewer lightmaps), eliminates inconsistencies in the lightmap when switching LODs, and you only have to bake the lightmap once instead of per LOD. If your assets come with LODs, make sure their lightmap UVs match the LOD0 (highest detail) mesh.

From experience, we know that on the original Quest, using the GLES API, you should target less than 100 draw calls for the static geometry in your level, with around 200 draw calls in total, including all dynamic draw calls. When targeting the Quest 2 or when using the Vulkan API, these targets can be increased slightly.

The target triangle count is dependent on the complexity of the shaders. In our case, we have a very simple fragment shader which leaves us with a lot of headroom on the GPU. This allows us to have a lot of triangles. From experience, we know that on the original Quest, when using a fragment shader with lightmapping, direct lighting from a single directional light, and normal and specular maps, we’d be limited to around 300k triangles. In this case, where we only have lightmapping, we can easily handle over one million triangles.

One thing to be particularly careful of are the set pass calls. Set pass calls can be very expensive on the rendering thread, especially when using GLES. A set pass call happens every time the material is switched in between draw calls. Our aim is to only have a single set pass call for all static geometry in the level, excluding things like the skybox and terrain, since these require very different shaders compared to the normal geometry. To do this you’ll have to combine your textures into either a texture atlas or a texture array and modify the meshes and shaders to sample from the correct texture coordinates.

Our world is a combination of four sub-levels which are then repeated to create one big world. We wanted to be able to bake each sub-level individually. This allows each sub-level to use different LOD settings. Also, if one of the sub-levels changes we only need to bake that one instead of all four again.

### Our Approach

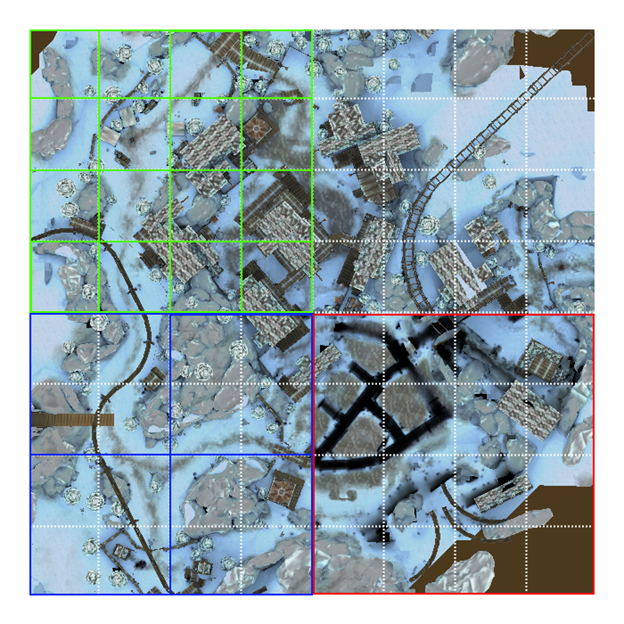
For each of the sub-levels, we take the following steps: Bake materials, generate lightmap mesh, bake lightmaps, generate LODs, and create the LOD structure.

#### Generate LODs

Our system combines meshes by grouping them into grid cells. Each grid cell will create a batched mesh of all the meshes it contains. Each LOD level creates cells double the size of the previous level. This allows us to use them in a quadtree-based hierarchical LOD system later. To reduce complexity on distant meshes, higher LOD levels remove small meshes before batching. Then we batch the meshes to create an LOD for that grid cell. To further reduce the triangle count you can choose to remove all triangles below the terrain.

After generation, we copy the lightmap UVs from our lightmap mesh to our LOD mesh. At this point all vertices have an identical vertex in the lightmap mesh because all we’ve done is remove vertices from the original mesh.

Next we decimate the batched mesh on all levels except LOD0, with each level allowing greater and greater error. This has to be done after copying the lightmap UVs since this process can alter vertex position.



Here the white lines are the cells. The green squares are the LOD0 meshes. The blue squares are the LOD1 meshes. The red square is an LOD2 mesh. Each square is a single mesh. In LOD0 you can still see a lot of small objects (campfire, logs). In LOD1 these small objects have been removed, but trees and buildings are still visible. In LOD2 only the large objects remain (buildings, rocks). You can see that the lightmap of LOD2 still has shadows where the meshes have been removed. This happens because the lightmaps are shared between all LOD levels.

#### Lightmapping

Unity stores all lightmapping information in a lightmap data asset. This asset can not be created or modified. This means that when we combine the sub-levels into the final level, we lose the lightmap data. To work around this, we combine all lightmaps into a texture array. We modify the shaders to sample from this texture array instead of using the default lightmap texture. We also modify the lightmap UVs on the meshes to include the lightmap offset and scale.

Note that the lightmap texture array can take up a lot of memory. Because of this, it is important that the lightmap texture array is in a compressed format. This can be achieved by first compressing the source texture to the desired format, then blitting it to the texture array.

It is important to store the lightmap offset, scale, and index as a vertex attribute to avoid creating unnecessary set pass calls.

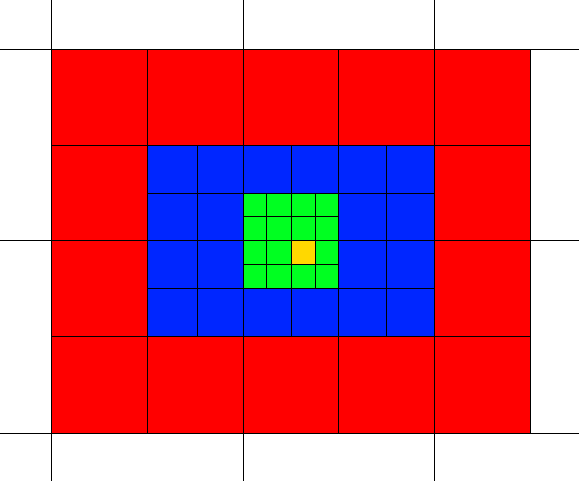
#### Scenes

To allow for streamed loading, each LOD mesh should be put in its own scene. Each scene should only contain a single mesh. The mesh should already be offset to its final position. Scenes themselves do not have a transform so if the meshes don’t get offset they will all spawn at the origin of the level.

Each scene should be added to the build settings. The LOD system should then be given the index of the scene that contains the matching LOD mesh in the build settings.

## LOD System

We switched to a hierarchical approach based on a [quadtree](https://www.oculus.com/lynx/?u=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FQuadtree&e=AT0efdjO5Paov91rwgzoV_CCP_ha5GTFIbpd4_DDOPloPjo-wh0bbrmqCpixJ_h3RcMmUV6kL56z7q8yA-ZLyRg2ze6wt3M2XvOamLShmrNYE0ZQt9l9L_RCGw9ndA8jFMuMmI-Qg4dfKXdP8rHaYg). This allows us to cover large areas very efficiently. If we calculate that a cell should be rendering LOD2 then we can skip checking all LOD1 and LOD0 cells entirely, as they would be contained by the LOD2 area. This means most cells never get checked. A hierarchical structure also simplifies streaming. A lazy streaming system can easily be implemented by showing the parent LOD until all LODs have finished loading. Alternatively, you can always load the child cells of the one currently being shown, at the cost of more memory.



If the camera occupies the yellow grid cell, you get this LOD structure. Green is LOD0, blue is LOD1 and red is LOD2. In this case twenty LOD2 meshes have been loaded but only fourteen are visible. Twenty-four LOD1 meshes have been loaded and twenty are visible. Sixteen LOD0 meshes have been loaded and they’re all visible. The white cells on the outside are nodes from the octree structure, these don’t have meshes.

### LOD State

Each time the camera walks into a different cell, we traverse the tree from top to bottom. At each level of the tree, we calculate the distance from the camera in cells. The cell size at each level is half that of the previous level. If the distance from the camera is less than 1 cell, we load but don’t show the mesh and go down to the next level. If we don’t have to go down or if we’ve reached a leaf node then we load and show the mesh associated with that node. Nodes which are loading notify their parent node.

We first do a pass in which we calculate the state we want each node to be in. Then we do another pass to apply the correct state. The reason for this is that we don’t want any of the child nodes to become visible until all child nodes have finished loading. Otherwise you could see meshes pop in. If a node knows any of its child nodes are loading it forces itself to be visible while keeping the child nodes invisible. Each time a child node has finished loading it will notify the parent node. The parent node will enable the child nodes and disable itself once all have finished loading.

To stop the LOD state from changing rapidly when the camera is on the edge between cells, we stop the LOD system from updating until the camera has moved at least 1 meter from the position where we were when we last calculated the LOD states.

For more detail, refer to the following files:

* LODManager.cs
* LODTreeNode.cs

Addressable group窗口中的路径前名称，可以选中预制体的时候更改

