目录

Unity 光照解决方案 2
一、3dsmax 内模型面数优化问题 2
1. 模型制作工具 2
2. 场景模型中存在大量冗余数据。 4
3. 场景中物体打组混乱，命名不清晰 7
4. 场景导出时 Unity 内材质会有所丢失 7
5. Unity 内如何实现打光以及渲染问题 8
6. Unity 内直接进行渲染同样需要耗费大量时间问题 8
二、Unity 光照工作原理 8
1. 实时光照 9
2. 烘焙光照(Baked GI) 10
3. 预计算的实时光照(Precomputed Realtime GI) 10
4. 关于使用烘焙光照和预计算的实时光照 11
5. 逐光源设置 11
6. 烘焙过程 12
7. 选择渲染路径(render path) 14
8. 颜色空间 15
9. HDR 高动态范围 17
10. 反射探针和光照探针 18
11. 阴影与阶梯式阴影 18
三、具体实践步骤 19
1. 3DsMax 内工作 19
2. Unity 内的工作 24
四、参考文献、参考视频和演讲 41

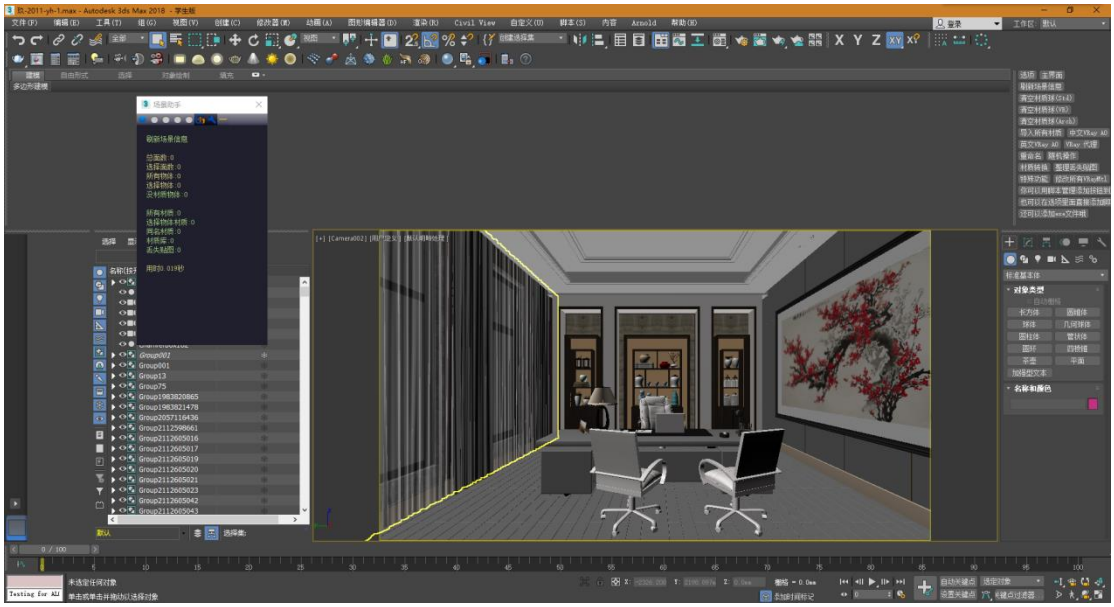
Unity 光照解决方案

本文档描述如何从 3dsmax 中向 Unity 引擎导入场景的过程。

一、3dsmax 内模型面数优化问题

1. 模型制作工具

先稍微了解下 3dmax 的概念及操作。



打开第一个场景：办公室后进入的界面，这里由于加载了 3dmax 场景助手插件，跟一般界面有所不同。

3dmax 工具栏，包含撤销，选择，磁力选项，场景资源管理器，材质编辑器，渲染选项以及其他自定义的常用快捷按钮。

另外在自定义选项面板中可以设置交互模式为 MAYA 模式。

在本工程中不涉及过多建模操作，因此工具栏使用并不多，主要使用的是修改器界面。

3dsmax 右侧的快捷工具栏，可以快捷创建物体，以及对物体进行编辑。

在 3dsmax 中，物体都有一个修改器栈，修改器提供多种工具来完成对模型的各种编辑，在适当时候可以塌陷修改器栈完成，这有点像 photoshop 内的图层，每个栈互不相干，但结束编辑后可以被合并。

但是由于 3dsmax 中每个栈针对的数据对象是相同的，而不像 photoshop 中每个图层有自己的数据对象，因而修改栈中下部的修改器可能会造成上层修改器的崩溃。在工程中修改器也是我主要修改的位置，由于场景内许多部分面数过分精细，因此需要进行删减，这就需要使用“专业优化”修改器。这也是第二个问题所在。

2. 场景模型中存在大量冗余数据。

原始室内场景模型文件大小

可以看到原始室内场景模型文件使用内存较多，达到 200MB 左右。但是这个数据量存在许多问题。

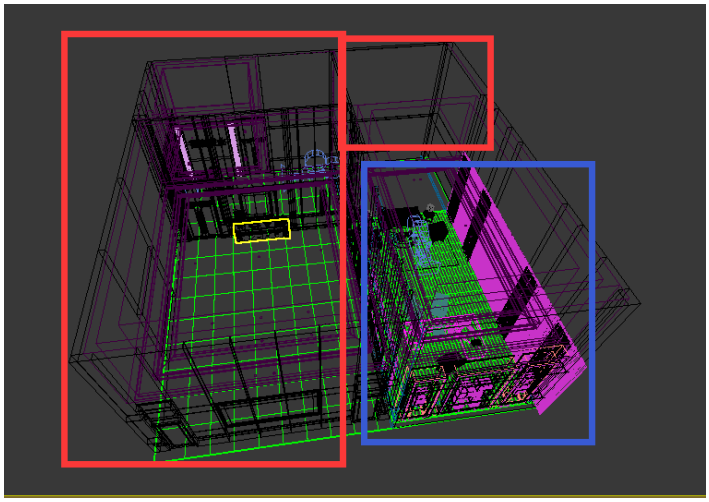
首先如此庞大的文件在 3dmax 内运行时会造成较大运行负担，每次自动存储文件备份时极其容易出现崩溃的情况，不利于工程进行。其次这么大的文件导出为 fbx 文件之后也有几百兆字节的大小，导入 Unity 时明显让 Unity 的运行也变得十分吃力。

显然这个数据量不是必要的，场景中存在许多冗余。

第一点：不可见部分模型冗余

我们的渲染视角主要集中在以下房间内，但是场景中同时还存在着其他房间。

如下图，红色部分是其余房间，蓝色部分是当前房间，显然红色部分的模型数据不是我们所需要的。

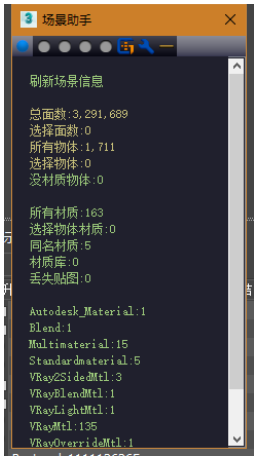


另外除了房间以外，蓝色房间内也会有许多物体不可见，比如关闭的柜子里摆放的物件，柜子里的物体北面的物件，以及花盆的内部，这些面都是通常的摄影机位难以捕捉到的物体，因此也没有必要保存在场景中，可以直接删除。

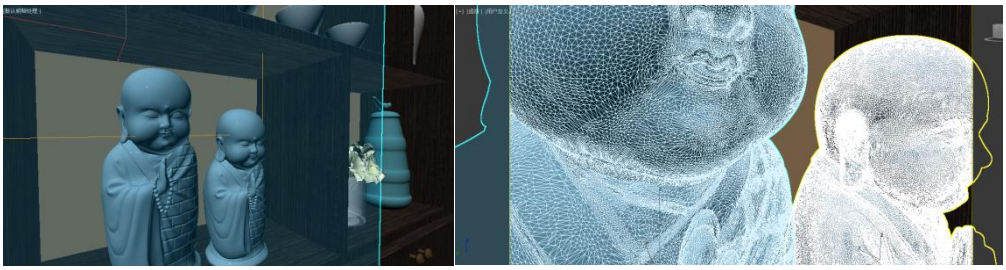
第二点：可见部分模型面数冗余

在上面的线框图中可以看到，实际上房间墙壁这种比较巨大，占画面面积较大的物体，其线框的密集程度不高，但体积更细小的部分，却使用了非常多的顶点和面来描述，参考 LOD 的思想，我们应该适当降低场景中的模型面数，尤其是对一些对整体观感影响不大的部位，要适当减低面数，把数据量使用在关键位置。

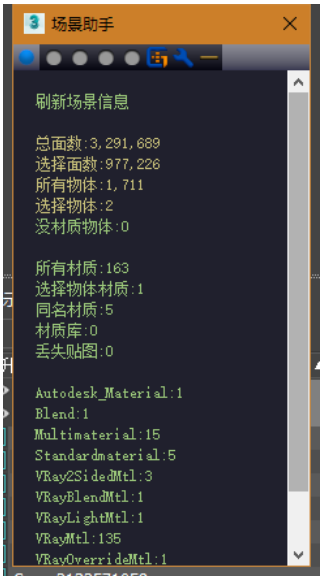
通过场景助手的统计可以看到开始前场景面数大致在三百万左右。



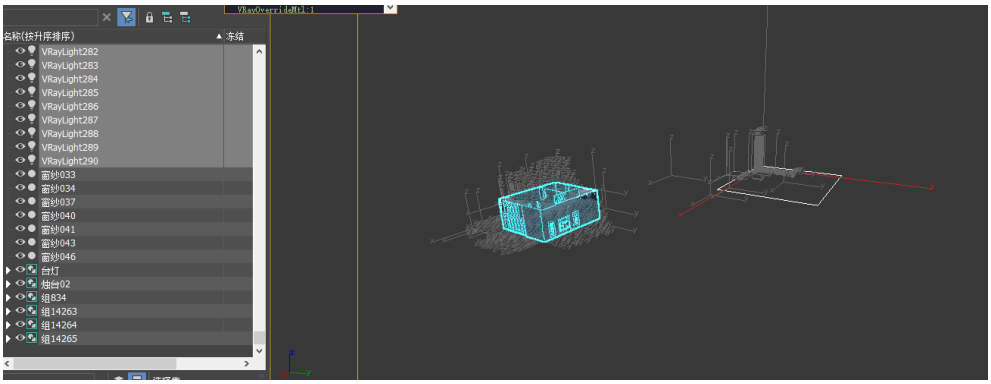
观察这个小佛像，在整体画面中只是一个装饰品，但是我们看他的线框密集程度：



可以看到光这两个小佛像，就占到了大致一百万个面，显然画面中一个简单的装饰使用了整个场景三分之一的数据量，是令人难以接受的。需要对这一类的物体进行面数删减优化。



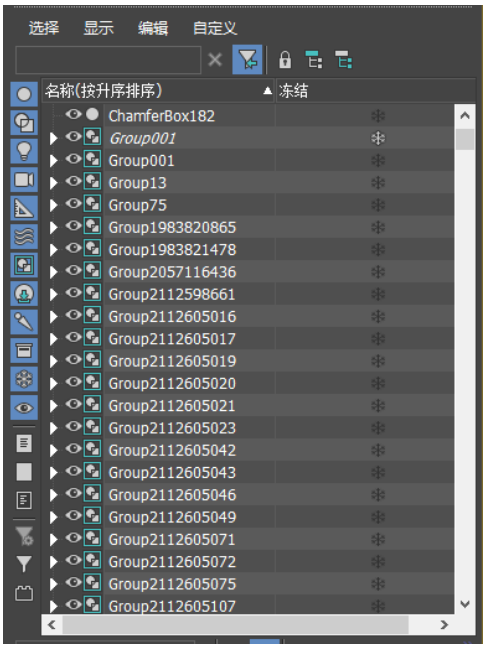
第三点：其他“不明物体”



可以看到场景中有时会残留有建模时使用的辅助物体，以及一些灯光，这些东西导入到 Unity 之后都是没有用处的，因此可以直接删除，从而保证大纲视图的干净，使工作更加顺利。同时删除这些物体也会一定程度上减少数据量。

3. 场景中物体打组混乱，命名不清晰

可以从下图中看到物体和组的命名情况是完全没有逻辑可言的。这对于后续工作来说会造成非常大的混乱，必须根据具体工作需要进行相关的重命名和优化，并合理化分组方式。44



4. 场景导出时 Unity 内材质会有所丢失

这个问题是 Unity 的更新导致的，解决方案会放在后面叙述，只需要两步就能解决，但是需要注意的是，场景中大部分的物体都只是有一个简单的材质和颜色，部分有贴图，这些材质贴图只能说用来帮助肉眼确认导出和渲染情况是否正确，并不能作为最终结果使用。关于材质问题，后续还需要进行细致的材质收集和处理。

5. Unity 内如何实现打光以及渲染问题

Unity 内提供了比较完整的灯光和渲染系统，在默认情况下打开项目使用的都是内置的渲染管线进行渲染，制作过程中还考虑了 HDRP 渲染管线，但是最终没有进行采用。灯光渲染方面的知识学习和工作会在解决方案中着重介绍。

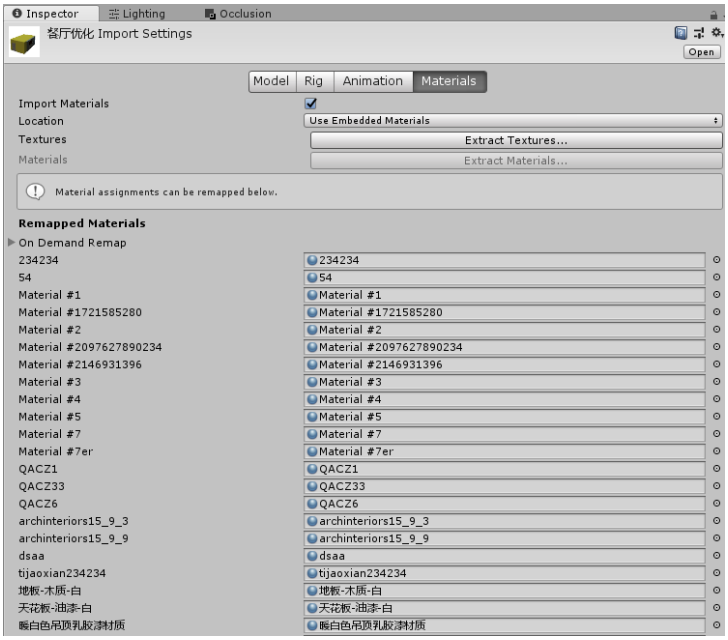
6. Unity 内直接进行渲染同样需要耗费大量时间问题

由于场景中细节众多，物体丰富，渲染所有物体的光照消费的时间几乎是不可忍受的，而且稍微太高渲染质量就会导致过长的渲染时间而最终无法完成。

二、Unity 光照工作原理

首先单独解决一下 Unity 内材质和贴图丢失的问题。

Unity2018 之后，材质和贴图不再是随着 fbx 文件导入 asset 的同时自动导入的了，需要在 asset importer 的 inspector 处点开 materials 标签手动载入。



载入时注意两点：

记得先载入贴图，后载入材质，这样材质载入时会自动加载贴图数据，否则材质上会没有贴图。

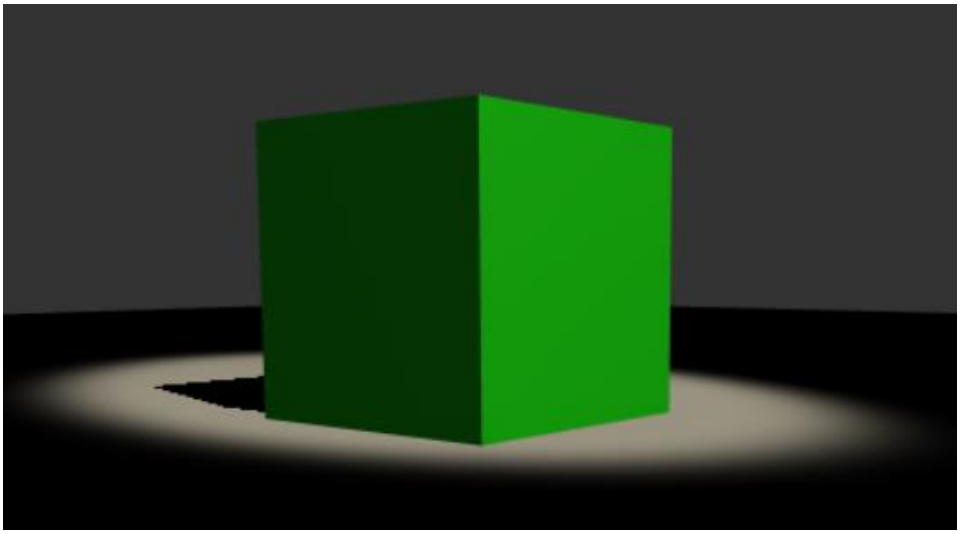
载入贴图和材质时需要选择文件夹，最好在 asset 文件夹下新建文件夹来存放这些数据。

我们的终极目标是为了解决 Unity 内场景渲染和打光方法的问题。

首先在 Unity 内实现光照，有实时光照(Realtime lighting)，烘焙光照(Baked lighting)，预烘焙实时光照(Precomputed realtime global illumination)三种方式。其中实时光照是完全实时的，而后两种则都是有一定预计算的。

1. 实时光照

一般来说，Unity 内有三种实时光，分别为平行光，点光和聚光灯，实时光会在场景内进行每帧的更新，因此当物体在光的作用范围内移动时，引擎会自动计算光和阴影的位置。



实时光照

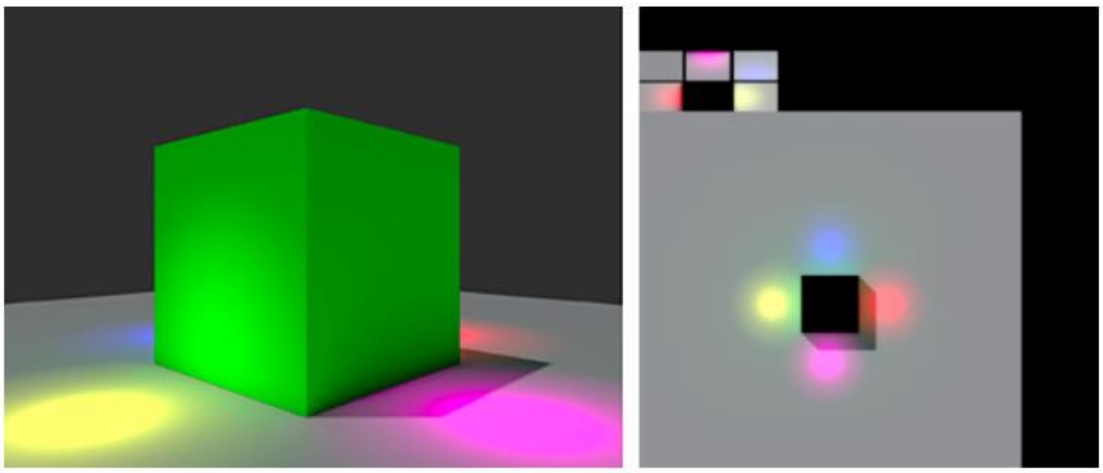
但实时光照使用比较简单的传统模型，不会和周围环境产生作用，即光线并不会在周围环境弹射，因此，如果需要达到较好的效果，还需要辅以间接光，即全局照明(global illumination)。

全局照明在 Unity 内大致可以分为环境光(ambient light)和上述两种预计算光照，环境光较为简单不做介绍，下面描述两种预计算。

2. 烘焙光照(Baked GI)

事实上，现实生活中的许多灯光，在场景不变化的情况下，是比较稳定的，固定的光源位置会与固定的场景进行静态的交互，最后的成像结果并不会因为观察角度而改变，这部分灯光实时上完全可以通过预先的场景设计和计算渲染出来。

Unity 将这些静态物体的光照信息保存到一些贴图上，这些贴图就成为光照贴图，这个过程称为烘焙。



左：简单场景在添加光照贴图后 右：根据这个场景所产生的光照贴图，在图中的模式下光照贴图包含了阴影

光照贴图能与模型的其他贴图(例如反照率贴图和法线贴图)叠加，经由着色器(shader)处理后得到最终结果。

在 Unity 中参与烘焙光照的物体需要被标记为 static，而实时光照可以在烘焙光照的基础上再进行叠加，从而做到直接光和间接光同时存在的效果。使用烘焙光照实际上是一种用内存换时间的思想，使用一些内存将光照信息预先保存，保证运行时的轻量，因此烘焙光照在移动平台上应用广泛。

3. 预计算的实时光照(Precomputed Realtime GI)

为了满足实时光照也能有一定交互性而不是完全静态的需要，Unity 提供了另外一套可交互的光照方案。

通过预计算的实时光照，Unity 能够实时的更改灯光的信息，同时为场景提供实时的全局照明，这种方案能够用来模拟一天中光照由于太阳光位置改变而产生变化的情形，这是烘焙光照做不到的。

但是这样的光照并不意味着物体能够随意走动，实际上物体仍然需要是固定在原位的，因为在这个过程中 Unity 需要预先计算物体表面与表面之间光照交互的关系与可能性，并且将这种关系作为数据存储起来，供运行时使用。因此物体的相对关系必须保持不变，但是物体的材质和颜色信息以及光的位置角度和光照颜色都是可以更改的。

事实上，我们最经常希望通过全局照明获得的是间接光，而间接光的频率，即在光照贴图的一定范围内光照颜色的变化速度一般比较慢，而 Unity 的预计算实时光照则利用了这一点。

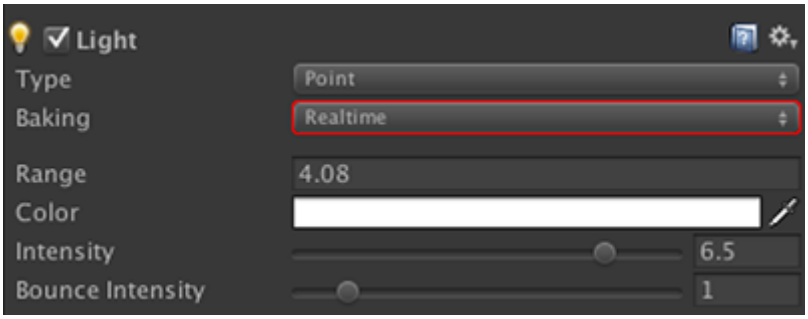
光照中的高频部分，也就是较为锋利的细节，例如硬边的阴影，一般来说使用实时光照会有比较好的效果，因此假设烘焙光照不需要生成这些复杂的细节，就能大大的降低光照贴图所需要的分辨率和计算量。

4. 关于使用烘焙光照和预计算的实时光照

默认情况下两者在 Unity 内都是打开的，可以先找到 window->rendering->lighting Settings，在这里打开 lighting 的设置窗口，在 lighting->scene 里面可以找到两者的开关。如下图所示。

5. 逐光源设置

在 Unity 中光源的默认烘焙模式为实时。



当全局设置仅勾选了实时照明时，光源的烘焙模式只能选择实时，而当全局设置勾选了烘焙照明之后，这里的选项可以被更改为混合模式或烘焙模式。烘焙模式的光源将不会产生实时光，而混合模式在影响静态光照贴图的同时，还会起到实时光照的作用。

如果希望某一个光源对全局照明有所影响，同时又希望动态角色被该光源影响的话，就可以使用混合光照模式。

6. 烘焙过程

在 Unity 中，烘焙是在后台进行的，可以被设置为自动烘焙或者手动烘焙，烘焙中我们仍可以进行其他工作而不被打断。

烘焙过程中在 lighting 窗口下方会与烘焙的进度条，显示当前正在执行的步骤，以及本步骤内还剩余多少工作。

烘焙全局光和预计算实时全局光的烘焙过程和算法是不同的，他们经历的主要阶段如下表所示：

Precomputed Realtime GI			Baked GI
01 - Create Geometry			01 - Create Geometry
02 - Layout Systems			02 - Atlassing
03 - Create Systems			03 - Create Baked Systems
04 - Create Atlas			04 - Baked Resources
05 - Clustering			05 - Bake AO

Precomputed Realtime GI			Baked GI
06 - Visibility			06 - Export Baked Texture
07 - Light Transport			07 - Bake Visibility
08 - Tetrahedralize Probes			08 - Bake Direct
09 - Create ProbeSet			09 - Ambient and Emissive
			10 - Create Bake Systems
Probes			11 - Bake Runtime
			12 - Upsampling Visibility
01 - Ambient Probes			13 - Bake Indirect
02 - Baked/Realtime Ref. Probes			14 - Final Gather
			15 - Bake ProbesSet
			16 - Compositing

主要说明一下预计算的实时光照，因为将来的工程中会主要应用这种光照方式，对于预计算的实时光照来说，第一步是创建几何体，即将场景内的 static 物体纳入光照的计算中。

第二第三步是创建系统(System)，系统是一个抽象概念，这里决定了哪些物体将被归为同一个系统而其他一些物体将被归为另外一个，同一个系统下的物体将公用一张光照贴图，一般来说系统的创建是自动的，但是我们也可以通过光照贴图参数(LightMap Parameter)给希望被放到同一张贴图的物体打标签来控制这个过程。

第四步即创建 Atlas,地图册，这里就是光照贴图分块的意思，上一步已经决定好集合体分类了，这一步就把烘焙贴图用的分块给创建出来了。

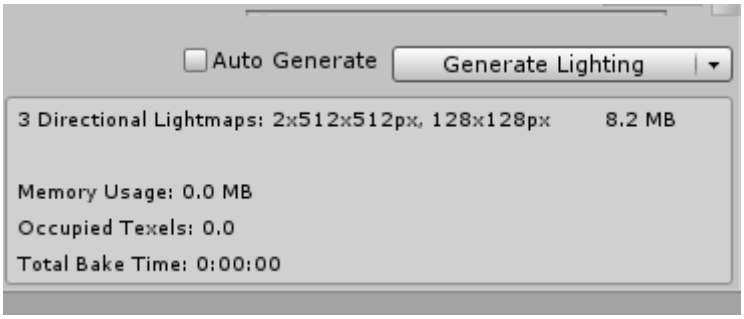
第五步创建 cluster 簇，通过将物体的表面分块，能够大致计算出块与块之间的光照关系，实际上使用簇是为了减低计算量，避免对每个像素进行计算光照关系。

第六步 visibility，这一步目前没有看到解释的比较清晰的资料，我自己猜测这一步是对画面中物体进行类似摄影机裁剪变换一样的剪裁？避免渲染可见性较低或不可见的物体？

第七步 lighty transport，主要进行光照的计算，当然实际上光照的计算在第五步已经进行了相当一部分。

后面几步都是针对光照探针(light probes)和反射探针(reflection probes)进行的，因此不多做解释。

而开始烘焙事实上是很简单的，只需要打开自动烘焙或者点击生成烘焙按钮即可。



7. 选择渲染路径(render path)

Unity 支持复数的渲染技术，主要包括前向渲染和延迟渲染，在开始渲染之前我们需要选择合适的渲染路径。

前向渲染(forward rendering)

在前向渲染中，每个物体对于多盏灯光，都有一个 pass 对应渲染，因此每个物体根据灯光数目不同会被渲染多次。这意味着当灯光数量增加，运行效率也会相应降低，一般可以通过舍弃视锥体以外的灯光来优化，但优化效率有限。

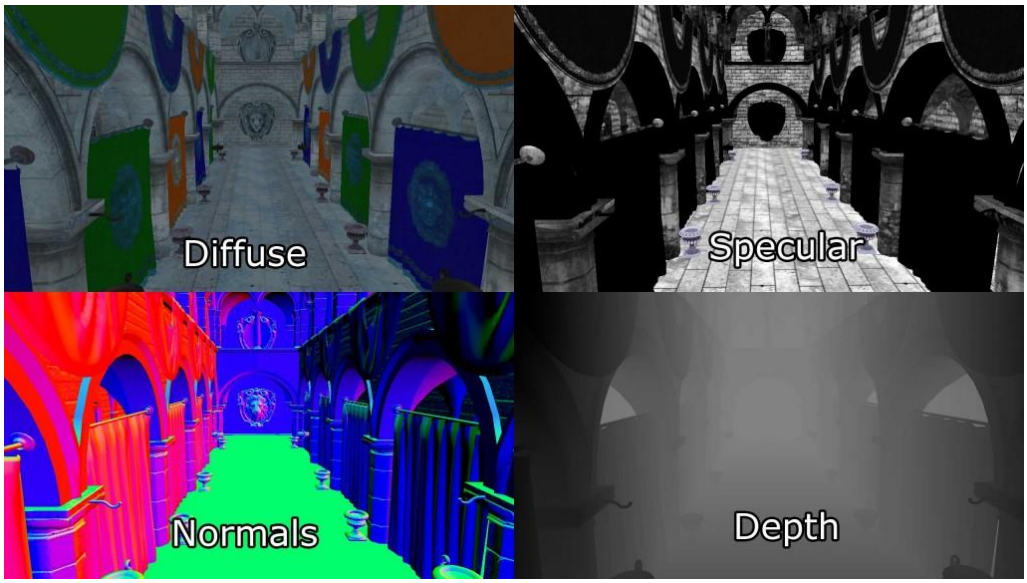
前向渲染的优点是，作为比较传统的渲染方式，其被各种图形硬件支持，尤其在移动端，目前还局限在前向渲染。且前向渲染事实上在场景内实时灯光很少的情况下可以运行的非常迅速，同时还能支持各种自定义的渲染方式，对透明物体的支持也非常友好，因为前向渲染只需要按照渲染顺序来渲染半透明和透明物体就可以了。

但是缺点也每场明显，一旦灯光增加过多，前向渲染就会很吃力。

延迟渲染(deferred rendering)

延迟渲染，顾名思义是因为它延后了光照的计算，转而首先使用一个 pass 将场景内后续渲染所需要的信息都存储到几张屏幕空间坐标下的纹理中，然后进行光照。这些纹理包含的信息有顶点位置，法线，材质，反照率，深度等等。

类似下图：



这些纹理构成了 G-Buffer: Diffuse(左上)，高光(右上)，法线(左下)和深度(右下)，高光因子存储在高光纹理(右上)的 alpha 通道。

这些纹理称为 Geometry-Buffer(G-Buffer)。

之后对于每一盏实时灯光，会使用上面的信息对场景进行着色。

延迟渲染的优点是对于光照，只需要对每个光源以及其覆盖的像素计算一次，也就是说计算的复杂度显著下降，因此相比前向渲染能够支持更多的实时光数量。

但是缺点是对于半透明的物体，由于 G-buffer 中存储的只能是不透明物体的信息，因此光照计算时无法将半透明物体纳入考虑范畴，只能考虑在渲染半透明物体时切换到前向渲染上。另外，延迟渲染中使用的灯光 pass 是统一的，因此如果要使用多个光照模型，就会产生问题。

8. 颜色空间

除了渲染技术的选择，另一个非常重要的设置是选择颜色空间。

这里包括线性空间(linear space)和 Gamma 空间(Gamma Space)两种。

下面开始解释这几个概念。

在物理世界中，光强和亮度是线性关系。但是显然人眼并不是这样感知的，由于人眼对亮度差的感知与当前亮度有关，越亮的部分，要让人眼感觉出不同，则需要相对于暗部更大的变化。也就是说，同样的光强差，在黑暗环境下人眼能够辨认，在光亮的环境下可能就会认为是没有差别的。

因此从编码的角度来说，暗部的信息对人眼来说更有意义，因此希望提高暗部亮度的编码水平，需要人为地抬升暗部，压缩亮部信息。

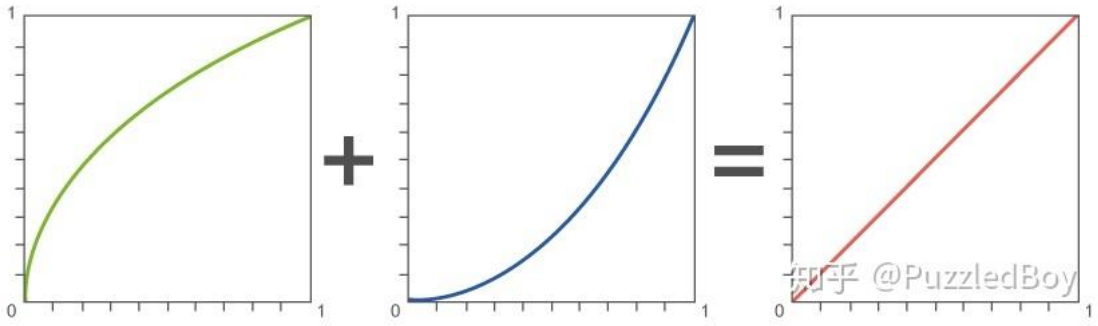
这可以通过一个Gamma矫正来完成，所谓Gamma矫正其实就是一个幂函数，而函数的幂次称为 Gamma 值。

例如 Gamma0.45 矫正的函数即为：

$$output = input^{0.45}$$

而 sRGB 使用的正是 Gamma0.45 的 Gamma 矫正。

但是这相当于对图像的亮度进行一次提升，直接输出的话会造成图像变亮的现象。但是恰好，早期的显示器，阴极射线管，显示图像时，由于成像原理并不理想，因此天然地会产生一次 Gamma2.2 的畸变，这就正好和 Gamma0.45 的 sRGB 编码中和了。

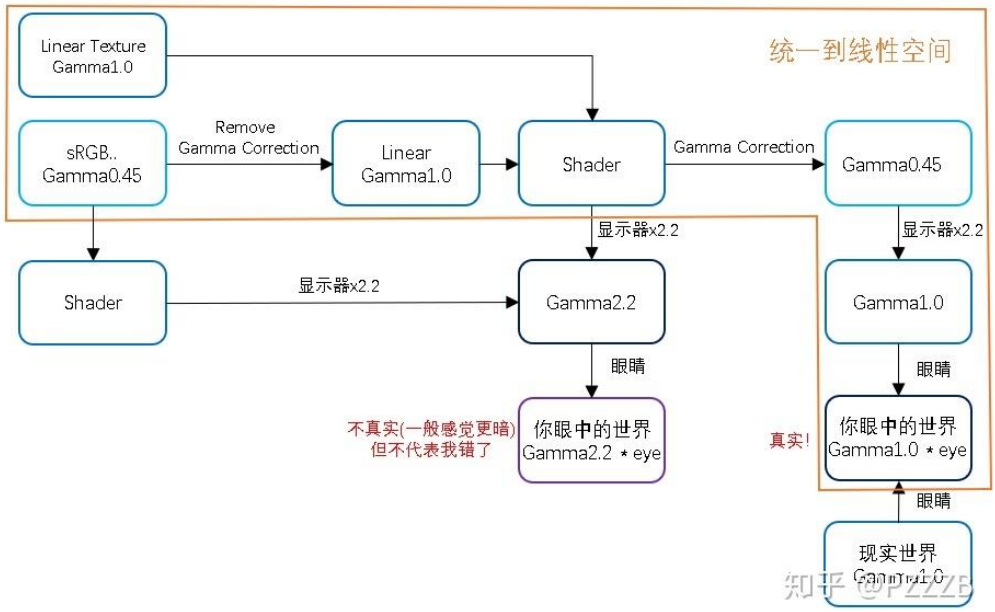


实际上 0.45 和 2.2 这两个 Gamma 值应该是早期阴极射线管显示器带来的，而其又刚好符合了对暗部编码的需要，因此这个矫正就被保留下来，1996 年微软和惠普一起开发了标准 sRGB 色彩空间，对应 Gamma0.45 所在的色彩空间，同时得到业界支持，现代的显示器配合这套标准以及早期的阴极射线管显示器，都从硬件层面具备 Gamma2.2 的矫正。

拓展到 Unity 中，由于计算机图形学的使用的颜色以及光照规律都是依照物理世界在线性空间描述的，因此如果所有的输入输出都统一到线性空间，那么出现的结果也会最真实。

Unity 内的 gamma 颜色空间不会对输入进行任何处理，因此所有输入的颜色都会在线性空间下被计算，然后通过显示器的 gamma2.2 输出，这对于 sRGB 这样的纹理本身是没有问题的，因为 sRGB 已经在 Gamma0.45 下。但是对于 Unity 内的光照数据，以及 Unity 内生成的程序纹理来说，它们都是在 gamma1.0 下的，跟 gamma0.45 下的 sRGB 纹理一起参与运算，本身就是有问题的。

而在 Unity 内的线性空间下，首先会将 sRGB 纹理进行 remove Gamma Correction 的操作，然后再参与运算，然后在输出之前会统一进行 Gamma0.45 的矫正，最后通过显示器之后还原到 Gamma1.0 线性空间。



颜色空间的正确保证了摄像机 HDR 的正常运转，也为后续 post processing 提供了正确的数据。

但是不幸的是线性空间不被某些移动设备支持，因此 Gamma 仍然是有存在的必要的，因为 Gamma 空间不需要引擎做额外的计算，所以某种程度上也减轻了程序的负担。

9. HDR 高动态范围

通常来说图像每个像素每个通道存储 8bit，rgba 四个通道总和 32bit，而 rgb 三个通道合起来能够编码的颜色只有 24bit，这是远低于自然界真实存在的颜色数目的，因此也叫做低动态范围 LDR。但对于极端的亮部和暗部来说，对编码的动态范围有很高的要求。

实际上 Unity 内是能够捕捉亮度值远超 LDR 的数据的，这些数据可以被保存在浮点数中，使得颜色的动态范围有巨大的提升，但这些数据需要经过转换后才能在 LDR 的范围内显示出来。

这就需要应用到 Tonemapping 技术。使用则个技术可以通过一个映射，将 HDR 映射到 LDR，这样可以使得 HDR 内捕捉的许多超出可视范围内的细节被 LDR 表现出来，也可以利用 HDR 的数据来实现光晕之类的效果。

10. 反射探针和光照探针

默认情况下，Unity 内新建的物体会使用 Unity 的 standard shader 来渲染，这个 shader 是一个基于物理的 shader，而物理材质一般来说会根据其 metalness 参数来决定反射外界光线的多少。但是由于硬件限制，往往做不到实时的光线追踪，因此就需要预先计算好反射信息，并将其储存在 reflection probe，反射探针中。

反射探针本质上是一个六面的 cubemap，反射探针会在其被放置的位置上架设一台计算机来渲染周围的环境，并存储到 cubemap 中，由此数据，位于这个位置的附近的物体在反射时就可以通过对这个探针采样而达到反射效果。

默认情况下物体使用天空盒子作为反射采样的依据，实际上天空盒子也可以被视为反射探针的一种。

而光照探针则使用与反射探针不同的原理，用来照亮场景中的动态物体。可以注意到，全局光只对静态的物体起作用，因此动态的物体很可能无法接收到全局光照的影响，因此需要光照探针来解决这个问题。

光照探针放置在场景中需要被照亮或者场景光线变化比较明显的位置，然后会在烘焙过程中被一起烘焙。在这个过程中探针会采集周围的全局光数据，并且通过频域变换的方式（球谐函数）存储光照信息中的低阶部分，这对于全局光来说已经足够了，使用低阶的光照足以将物体融入周围环境使之不至于过于突兀，对于凸表面物体来说尤其适用。

11. 阴影与阶梯式阴影

Unity 内使用类似贴图的方式来实现阴影，也就是说阴影是存在一定分辨率的，在同一分辨率下，同样大小但是位于远处的阴影，其与位于近处的阴影使用的阴影遮罩的像素是一样的，但是在相机视角内则只占据很小的一部分。因此近处的阴影虽然在视口内看起来很明显，但分辨率却不够高，同样的远处的阴影则使用了过多的分辨率。为了优化这一点，Unity 采用了阶梯式阴影的方式，按照

层级，里摄像机越近的阴影拥有越高的分辨率，这一点可以在 Unity 的 Quality Settings 内设置，Unity 允许最多使用四个阴影层级。

三、具体实践步骤

毫无疑问我们主要做的工作分为两部分，一是场景的优化，二是场景的打光渲染。其中场景优化可以分为在 3dsmax 内的优化和 Unity 内的优化。

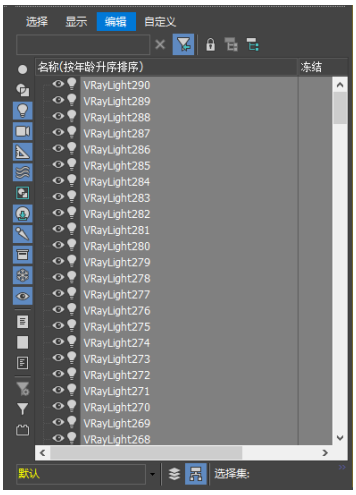
首先，最开始时在第六点问题中提到，Unity 内渲染场景会耗费大量时间，经过查阅资料和研究之后知道，这是由于场景过于复杂，面数过多，以及 Unity 内的渲染设置不当所致。如何在 Unity 内进行良好的场景设置会在后续使用更多篇幅来解释。

1. 3DsMax 内工作

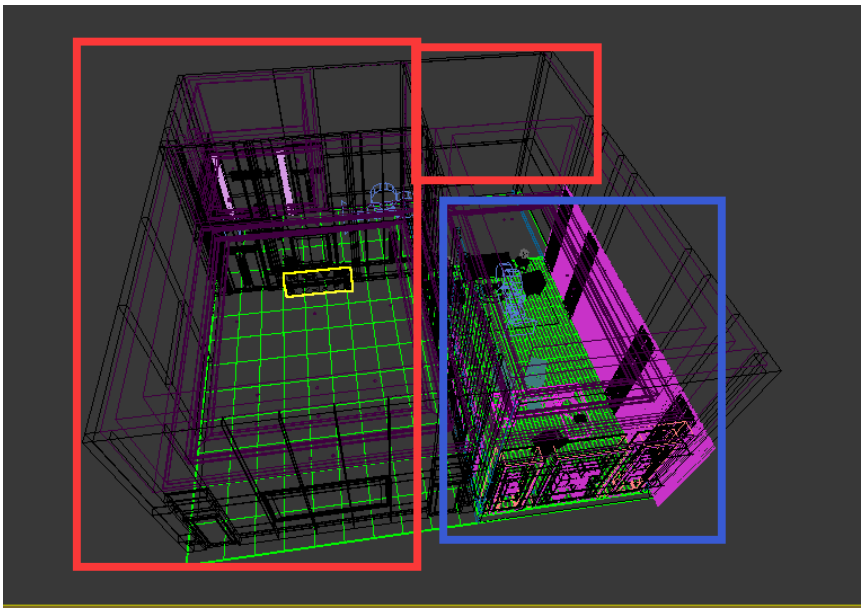
第一步：删除 3dsmax 内所有不可见物体

3dmax 内的工作实际上并不困难，主要是需要一定的工作量。第一步就需要首先剔除渲染无关的物体，减轻运行负担。

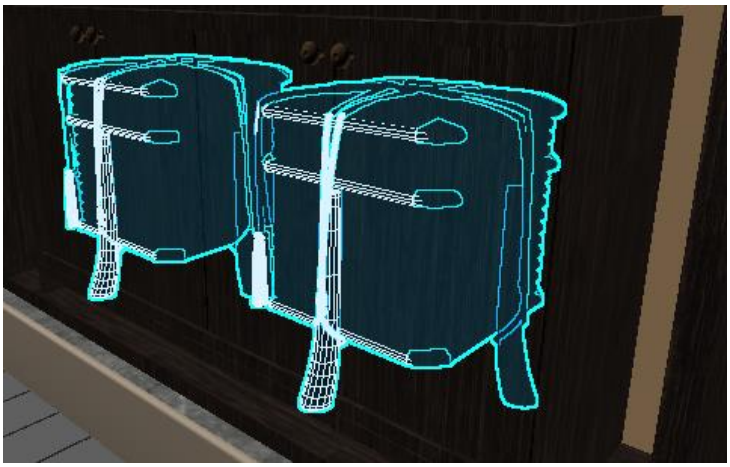
删除所有非几何体，因为我们在 Unity 中完全不需要这些物体。



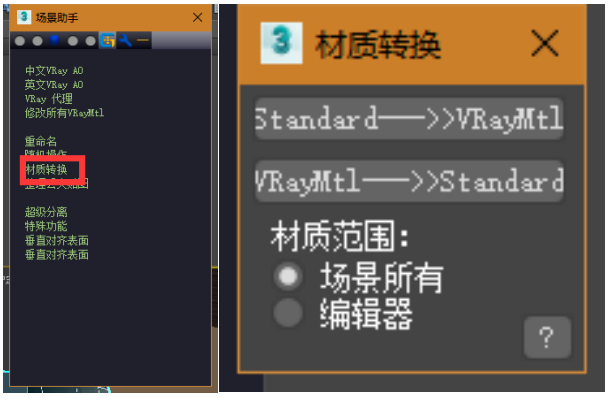
删除不需要的场景，本例中为场景中红色部分。



删除这类完全不可见或者对场景帮助不大的物体。



第二步，将所有 Vray 材质转换为 standard 材质。

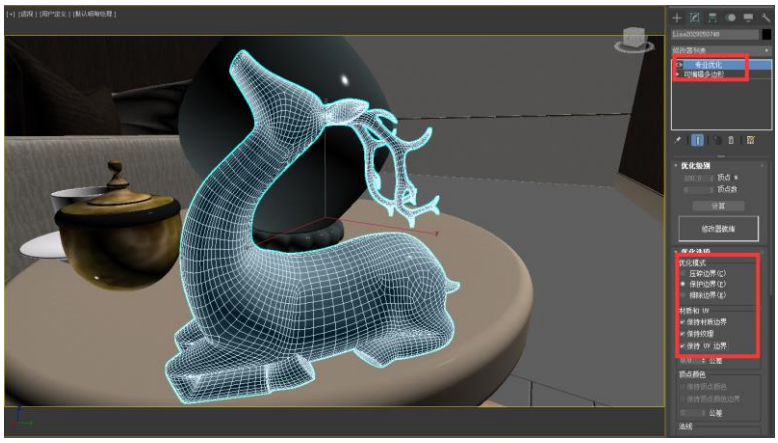


第三步，将所有组解组

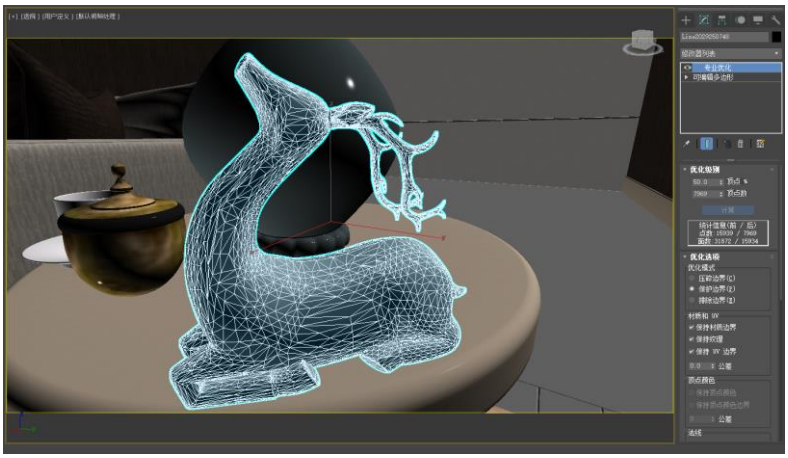
因为场景中的组可能十分混乱，并且不利于面数优化操作。这里只需要不断全选所有组，然后解组即可，最后可能会有几个组无法解开，这时候只需要选中并关闭组，再次解组就可以成功了。

第四步，减少没有必要的模型精度

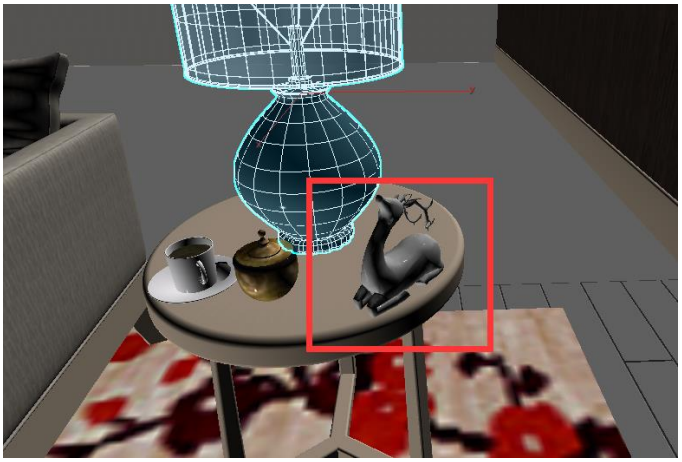
过于浪费空间又没有效果的模型可以直接删掉，例如最开始时指出的小佛像，就可以直接删除。这里主要使用修改器中的专业优化工具，可以在保持 UV 边界和几何边界的情况下，按照要求的精度减少模型面数。



这里以这只小鹿模型为例，选中专业优化修改器，选中保护边界模式，并保持所有材质和 UV 边界，就可以开始优化了，上图是优化前的模型，下面是将顶点减少大约一半之后的效果。

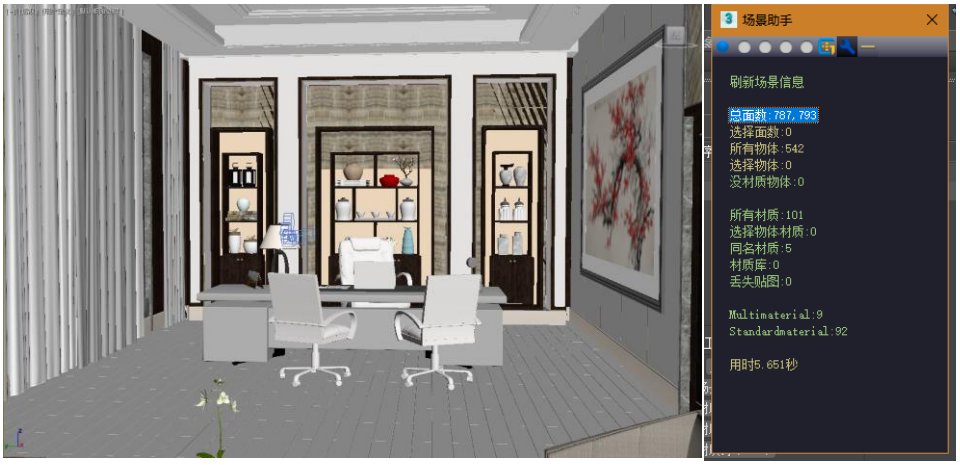


事实上，由于这类装饰模型在场景中并不重要，所以即使把这个模型的顶点数降低到 30%也不会有太大的影响，效果如下图所示。

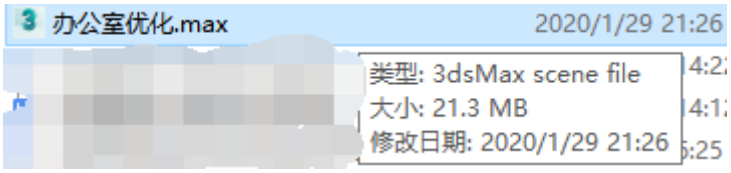


之后就是将每个物体都经历一遍这个步骤，根据模型重要程度可以做不同程度的删减，特别明显而又没有过分精细的模型可以不动，保持它拓扑结构的完整性，后续光照的完整性也会更好。

下面是优化过后的场景，外观上并不会产生非常巨大的不同，但是面数从最初的大约 300 万个面减少到了大约 80 万：



而此时文件大小也减小到了 20MB 左右，足够轻量。

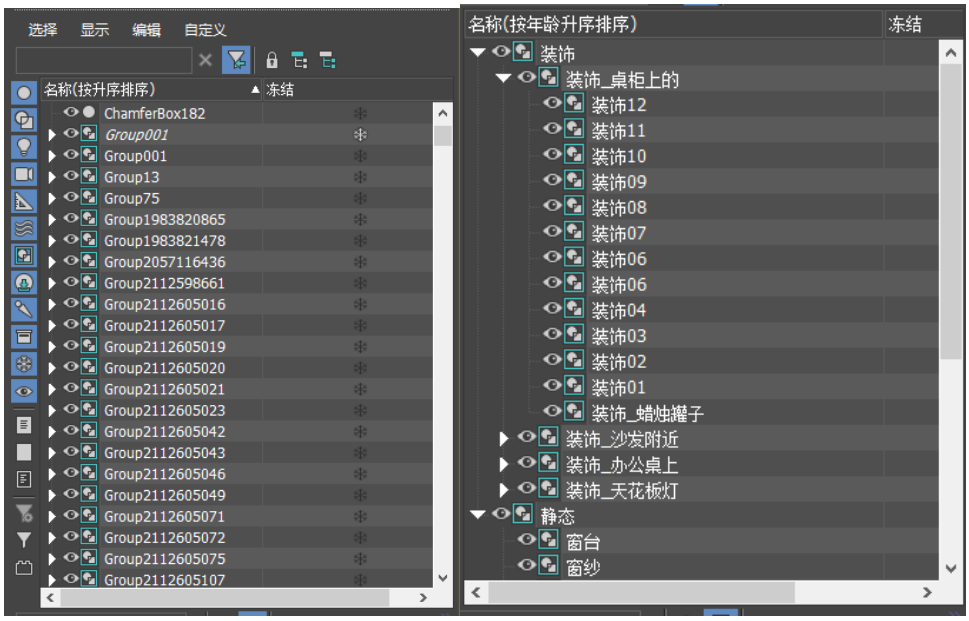


第五步：给所有模型重命名并进行恰当的分组

这里的分组是为了 Unity 内能够便利地选中自己想要的物体，我们知道 Unity 内需要为物体打标签，标记哪些物体是 static 哪些不是，为了到时候不需要手动选择，最好在三维软件内做好这个工作。另外，虽然目前还没有进行到颜色编辑

的阶段，但是这一步最好能够一起考虑一下颜色编辑时的分类，是否将同一材质的物体都归类到一起也是这一步需要考虑的问题。

这里直接放出结果：



左：分类重命名前 右：重命名并且完整打组之后，根据是否要被标记为 static 做了明确的分类。

第六步，塌陷所有物体的修改器，并将枢轴居中

第七步，导出场景

这一步要记得在导出时选择包含嵌入的媒体，否则模型的材质和贴图将不会被包含进 fbx 文件中。

至此就完成了一个场景在 3dsmax 内的工作了，接下来转到 Unity 内进行工作。

2. Unity 内的工作

实际上在本工程尝试过程中我尝试了 precomputed Realtime GI，Baked GI 和 HDRP 三种工作方式，其中 HDRP 是高清渲染管线，主要支持的也是 Baked GI，但是能够带来更好的图形效果，但是最后决定还是使用 Precomputed Realtime GI，原因在于本工程需要实现颜色重编辑的效果，这意味着我们需要实时更改场景中物体的材质和颜色，而在烘焙照明中，烘焙后的光照贴图事实上已经考虑了烘焙时物体的固有色和材质信息了，因而如果烘焙后再进行更改材质，这个材质变化是不会影响光照贴图的，这会导致场景中的照明看起来不正确，因此最终还是决定使用预计算的实时全局光照。

这里主要详细说明预计算的实时全局光照的实施细节

在之前已经提到过，实时全局光会计算场景内静态集合体(static Geometry)的光照信息，即光在场景内会有什么样的弹射路径，当这些信息被计算出来之后，会被存储起来用于实时的光照计算，这个过程能够显著减少运行时计算光照的计算量，同时满足全局照明和实时可交互的更改。

我们知道在烘焙照明中，光照信息会被烘焙到光照贴图上，并且不能在运行时被更改。但是在预计算的实时光照中，Unity 并不会创建光照贴图，取而代之的是 Lighting Data Asset。Lighting Data Asset 中存储了在运行时生成一系列低分辨率的光照贴图所需要的数据。

不难想象，这个计算过程将会耗费大量的时间，因此我们需要使用一定的手段来进行优化。

首先要进行设置的是全局的光照分辨率(Realtime Resolution)

Realtime Resolution 指的是 Unity 在每个单位上会渲染多少个纹素(texel)，纹素实际上就是纹理的像素，想象一下屏幕上的一个像素和纹理上的一个像素并不是同一个大小的，因为纹素会被拉伸，而一个物体使用的贴图分辨率越高，包含纹素越多，每个纹素在屏幕上也就越接近一个像素大小。

实际上最为精确的渲染是对每一个像素都进行光线追踪，从而得到高精度的结果，但是实际上这是完全没有必要的，在模型表面上相近的点之间接受的光照

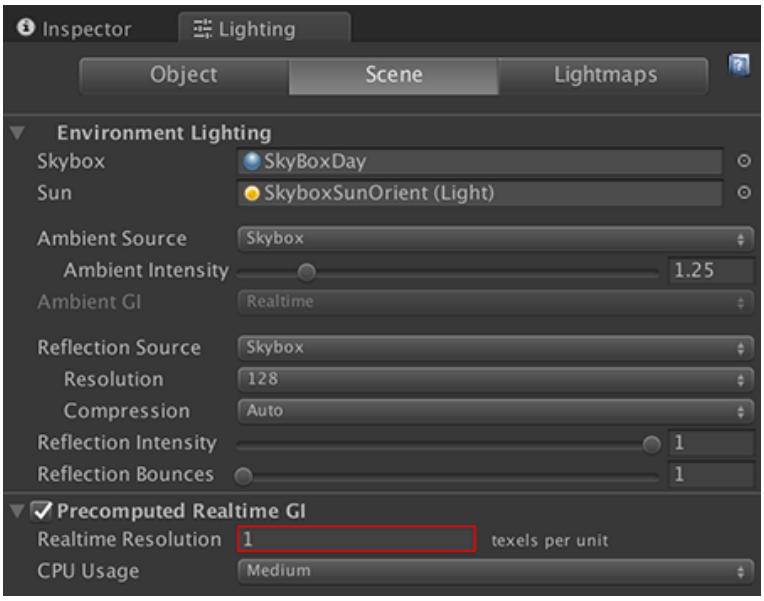
信息一般不会有很巨大的变化，甚至这个变化是微乎其微的，因此只要对很多个像素统一计算就可以了。

这里 Unity 就会对每一个纹素进行光线追踪，然后通过 UV 展开铺到物体的表面。

而为了更明确地让人感受到这个纹素大小的变化，Unity 使用“纹素每单位”(texels (texture pixels) used per world unit)来规定分辨率。

这里我们知道 Unity 内的一个单位(Unit)默认指代 1m 的物理距离。

设置 realtime resolution 可以找到 Lighting Window()Lighting window (Window > Lighting)，选择 Scene 选项卡，确保 Precomputed Realtime GI 被勾选，然后在 realtime resolution 参数内设置分辨率。



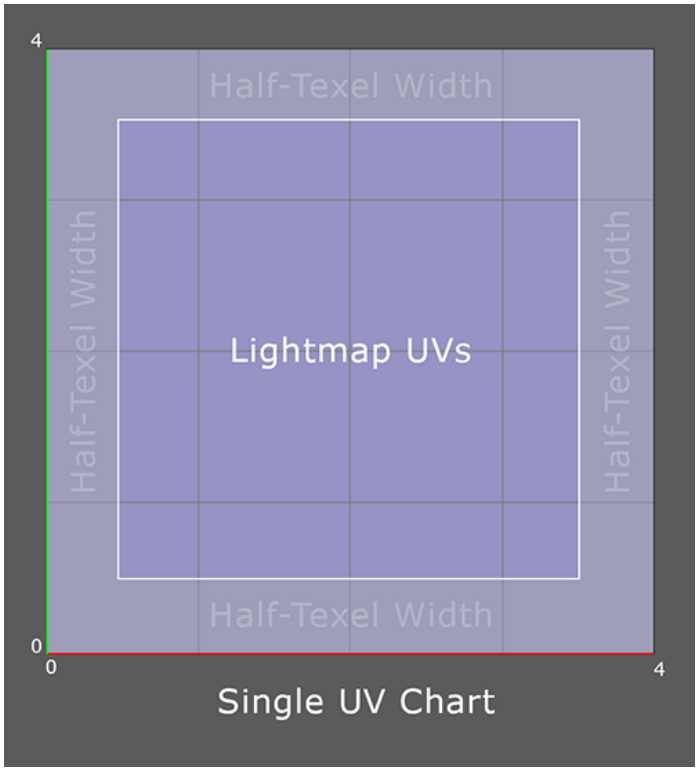
在设置这个值之前，需要先确定场景大小和类型，最好能在三维软件内设置好单位之后再导出到 Unity。一般来说，体量不大但是细节丰富的场景，例如室内场景，可以使用 2-3 的分辨率。而对于比较打的室外场景来说，则没有必要使用这么大的分辨率，因为室外光照变化不大，使用这么大的分辨率会导致内存和 cpu 运行的浪费，因此室外场景下，物体使用 0.5-1，地形使用 0.1-0.5 的分辨率就足够了。

接下来需要理解两个比较重要的概念，charts 和 cluster。

Charts

我们知道物体的表面在贴图时一般被展开成多块 UV 壳，这些 UV 壳一般来说是不规则的多边形，而在 Unity 内，进行预计算的实时光照时，为了避免这些

UV 壳产生混叠，Unity 使用一个比这些 UV 壳略大的矩形来包裹这些 UV 壳，而这些矩形称为 UV charts。UV charts 的概念在预计算的实时全局照明和烘焙全局照明内都有。UV chart 分为两部分，第一部分用来存储光照颜色信息(irradiance)，另一部分用来储存光照方向(directionality)。



当生成实时全局光时，每个光照纹素的计算都是在 UV charts 内进行的，每个 Chart 默认情况下的最小大小是 4*4 个纹素，因此每个 Chart 至少会占用 16 个纹素的空间，无论这个 Chart 所代表的表面在场景中占据多大的大小。

另外 UVcharts 会比 UV 壳在上下左右至少各有半个纹素的缓冲区保证 UV 混叠不会发生，但也会造成纹素的冗余，因为每个 Chart 都会产生一个很小的实际上无用但是需要被计算的边界。

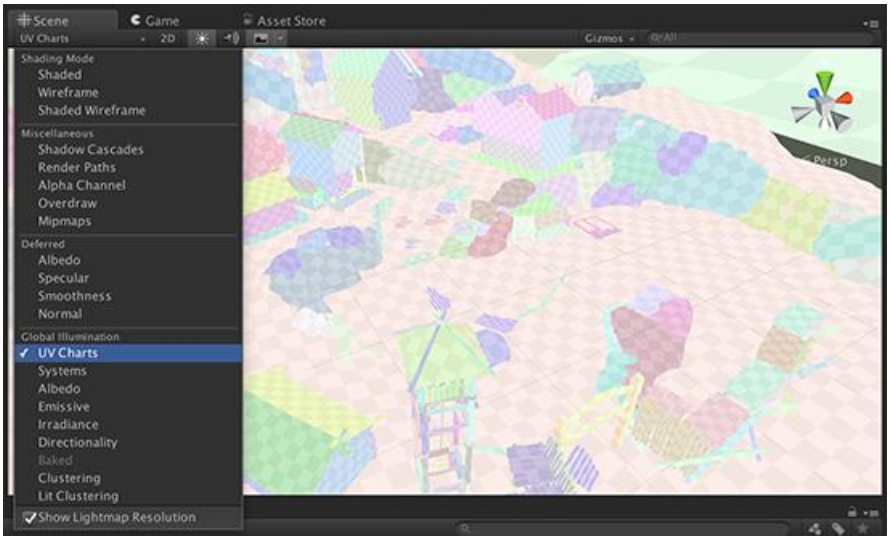
这两个特点意味着即使一个物体很小，但如果它拥有 50 个 UV Charts，则至少要占据 50*16 个，也就是 800 个纹素来烘焙光照，显然这是不合理的。因为之前提到过，Unity 的光照计算是逐纹素的，所以 lightmap 里的纹素越多，光照计算的循环就越多，尤其在复杂场景里，如果 UVchart 非常多的话就更容易产生严重的运行问题。

所以，减少运算量的终极目标时间少纹素，而减少 chart 的数目则直接关系到这一目标。

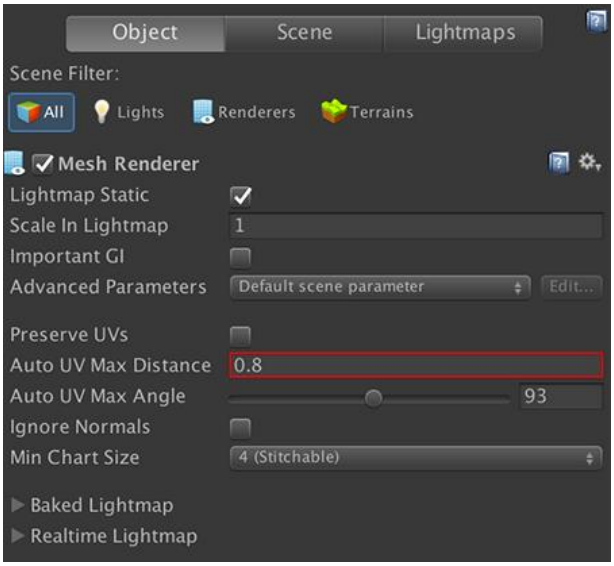
但是在这之前，还可以通过减少 static 物体来实现减少参与实时全局光运算的物体，这一点对于其他全局照明来说也是一样的。

对于非 static 的物体来说，无法使用全局光照，但是在上面已经提到过，对于动态的物体，Unity 提供光照探针(light probes)技术来实现光照解决方案，使用光照探针，动态物体能够从场景中获取光照信息。虽然动态物体不会向周围环境贡献全局光照，但是通过人为地选择适合作为动态物体的物体可以让这个影响降低到几乎不可查觉。一般来说，较小的物体，尤其是凸面体较为合适，而凑巧的是，这些物体数量庞大且很琐碎，为他们生成 UVCharts 往往会造成很大程度的内存浪费，因此更不适宜全局照明。

下面就是如何优化 UVchart 数目了，首先，我们可通过编辑器内的绘制模式，切换到 UV charts 模式来观察场景中到底有多少不同的 UVcharts。



对于每一个物体，在他的 MeshRenderer 内，一旦勾选了 lightmap static(在新版本内改名为 Contribute GI)，就会出现该物体实时全局光的相关设置。如下图：



图中有四个参数相关：

Preserve UVs(新版本内为 Optimize UVs，是相反的意思)：是否使用手动展开的 UV 作为实时全局光使用的 UV。

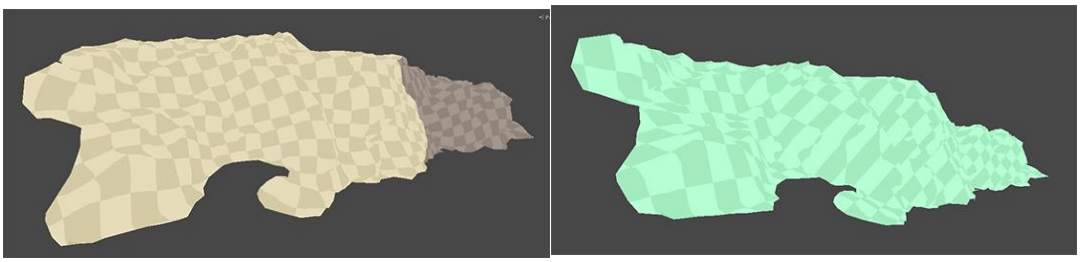
Auto UV Max Distance：提升这个值会减少一个物体自动生成的 UV charts 的数量，相对地会使 UV 产生一定的畸变。

Auto UV Max Angle：同上。

Ignore Normals：打开之后有时可以减少 UV Charts 数量，但不一定，同时也会产生畸变。

一般调整中间两个参数即可。

调整 UV charts 的过程需要在减少计算量和增加 UV 畸变之间权衡，需要做的工作是尽量在保证 UV 畸变可接受的情况下最大限度减少 UVCharts 数量。例如下图，在 UV Charts 渲染模式下，如果物体上纹素大小相对统一，意味着畸变较少，反之相对即便较多。

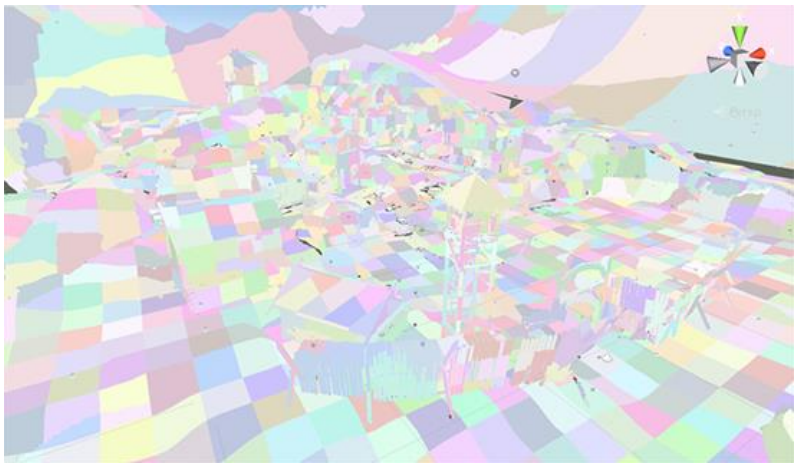


左：畸变较少 右：畸变较多

上面我们花费篇幅解释了 UV Charts，下面要介绍另外一个跟预计算时间相关的概念。

Cluster

Cluster 与 Chart 类似，至少看起来十分相似，也是把光照计算分块进行的一种实践，Unity 把场景内的物体分块，Cluster 实际上也是一种物体表面的补丁，主要用于复杂的光照计算，下图展示了 Unity 官方一个示例场景内的 Cluster 情况。



关于 cluster 的具体原理 Unity 官方的文档内并没有说明的非常清楚，但是这不妨碍它需要被优化的事实，实际上，cluster 的优化原则和 UV Charts 十分类似。都需要遵循近实远虚的原则，远处的物体由于没必要使用那么高的分辨率，因此可以适当降低 UV 和 Cluster 的个数，当然，调整精度要求较低的物体的分辨率也是可行的。

Cluster 是真正的光照被计算出来前的最后一步，实际上 Unity 内可以查看光照下的 cluster 的情况。可以在 Clustering 或者 Lit Clustering 渲染模式下观察。

因此总的来说优化一个场景的思路有四个个。

第一个是剔除不希望参与全局光照运算的物体，直接减少了工程量。

第二是如果可能的话，降低场景的分辨率，这样能直接减少所有物体对纹素的使用，同时也会减少 cluster 的数量，因为 cluster 的数目和纹素是相关的。

第三是尽可能减少 UV Charts，这样可以减少冗余，同样可以减少纹素的使用。

第四是通过减少一些不希望使用太多 cluster 的物体的 cluster 数目，从而减少 cluster 总数，这样不仅提高了预计算的速度，同时对运行时的效率也会有一定提升，因为需要被运算的数据也会相应减少。

我们已经说明过如何实现第一和第三种，而第二和第四中方法还没说明，这两种方法的优化方式都被集成在 Unity 的 Lightmap Parameters asset 中。

Lightmap Parameters Asset

在 Unity 中，可以创建 Lightmap Parameters asset，只需要在项目窗口右击创建即可。

可以看到前两个参数就是设定分辨率和 cluster 参数。

解释一下这里的参数

这里的 resolution 参数会和场景设定的 realtime resolution 参数相乘，得到最终用于渲染的 resolution。

Cluster Resolution，同上，是一个用于相乘的系数，实际上 cluster 的数目就是根据 resolution 得到的，一般来说 cluster 的数目不需要超过 resolution 本身，而是小于等于 resolution 就可以了，所以 cluster 在场景中看起来的分块总是要比 texel 的分块要大一些，默认情况下 cluster 的长宽会是 texel 的两倍，面积则是四倍。在这个参数里我们可以进一步调整场景中 cluster 的数目，但是一般来说也不能降得太低，甚至对于一些我们希望能够得到更好的光照细节的物体还可以提升这个数值。

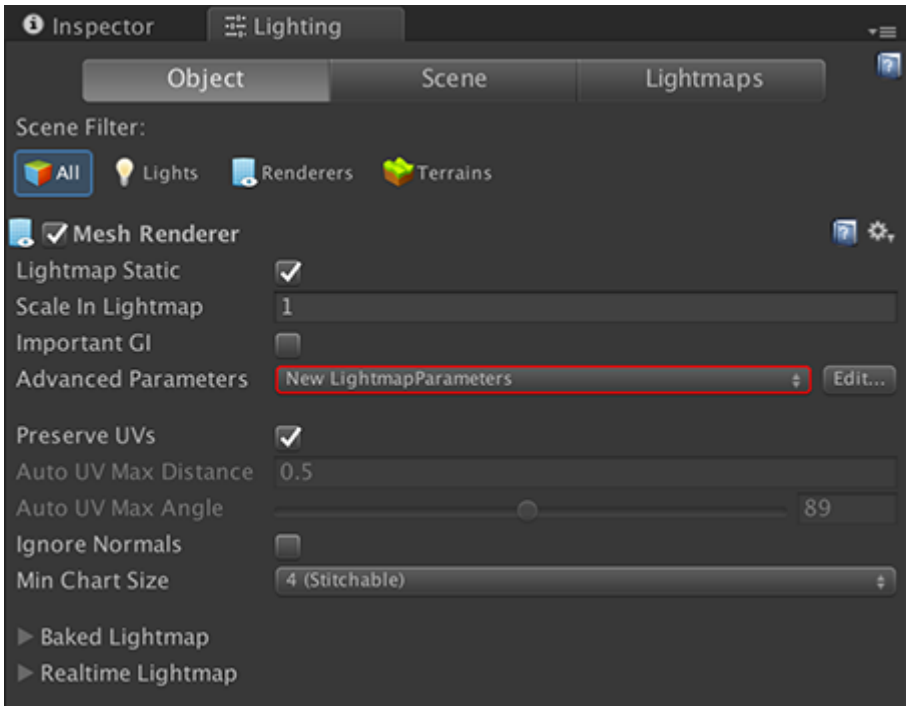
Irradiance Budget 决定了运行时每个纹素所属能够获得的用于计算光照的内存的大小，提升这个数值会提升运行时光照的质量，但是会造成一定负担。但是这个数值对预计算的时间没有太大影响。

Irradiance Quality 则决定了每个纹素在对光线采样的时候能够射出多少采样射线，这可以用来提升光照的准确性，如果某些地方的光线产生明显的错误，那

可能是由于采样数不够导致的。这个选项并不会影响运行时的效率，它只会增加预计算的时间。

其余参数在工程中并没有起到较有意义的作用，在这里就不做阐述了。

Lightmap Parameter asset 最终可以被应用到物体的 Mesh Renderer 里，如下：

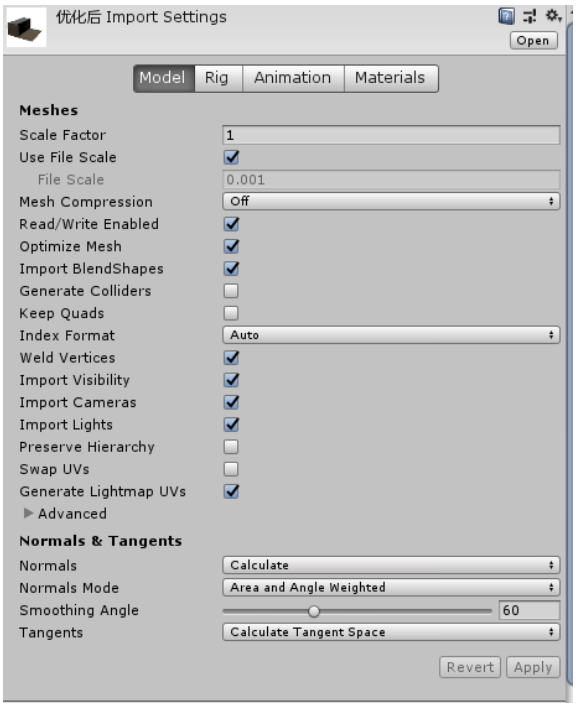


接下来就进行 Unity 内的场景优化

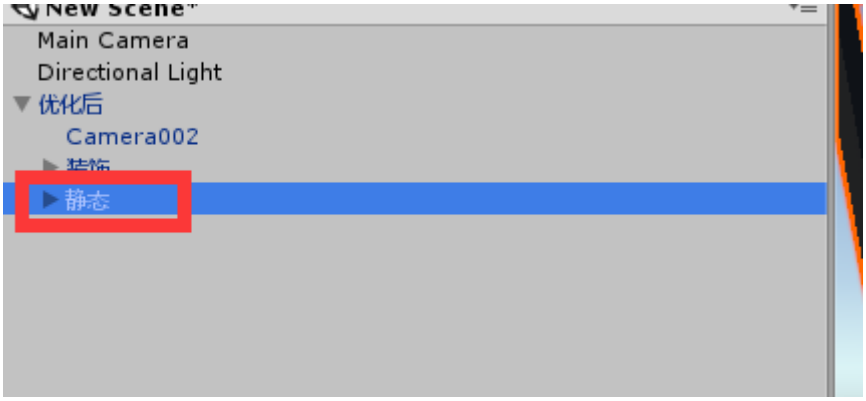
我们要做的步骤大致为

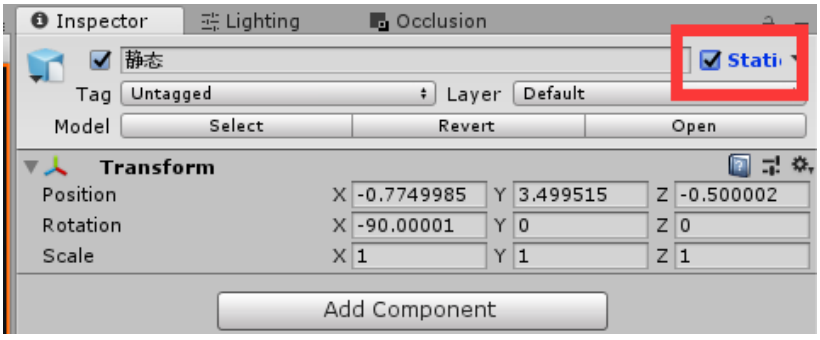
1. 导入模型资源，手动提取贴图和材质
2. Lightmap UVs 以供烘焙光照。
3. 分别设置好静态物体和动态物体
4. 设置场景烘焙参数
5. 设置渲染路径和颜色空间
6. 根据情况进行 UVcharts 和 Cluster 数量的优化
7. 设置光照探针与反射探针
8. 设置好光源照明的位置
9. 烘焙场景
10. 架设摄影机位置，应用屏幕后处理

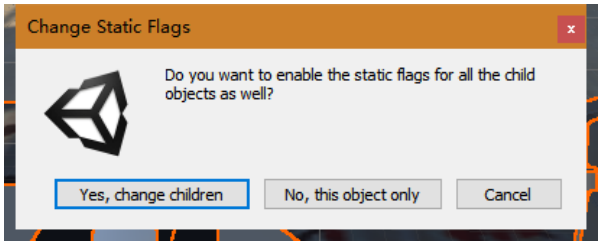
首先在导入模型资源时要提取贴图和光照，这一点在文章开头已经提及。另外要切换到导入模型的 model 面板，勾选 Generate LightMap UVs，然后应用设置。



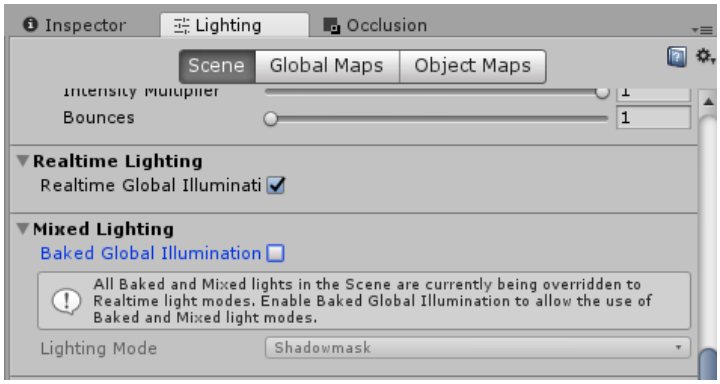
可以看到我们已经在 3dmax 中做好了静态物体的分类，这里直接选中静态物体父对象，勾选 static，然后在弹窗中选择 yes，change children。



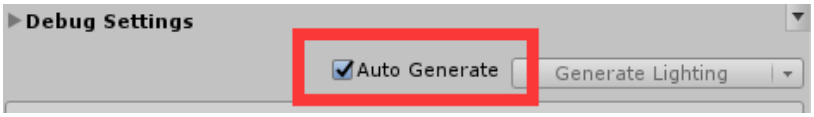




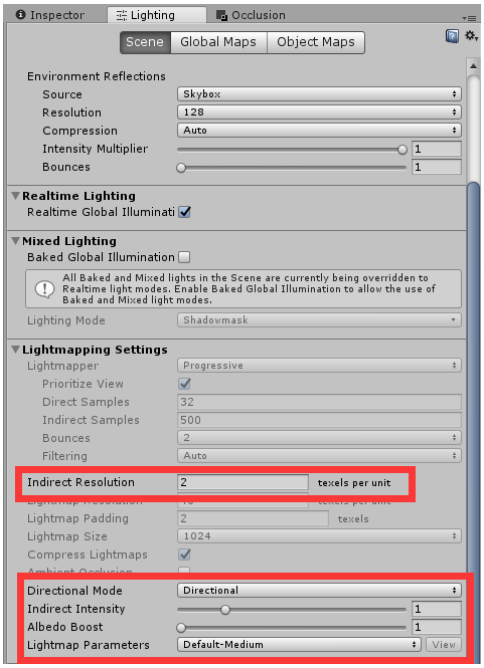
然后转到灯光设置 window->Rendering->Light Settings->Scene，打开 Precomputed realtime GI，关掉 Baked GI。



并勾选 Auto Generate，方便优化。

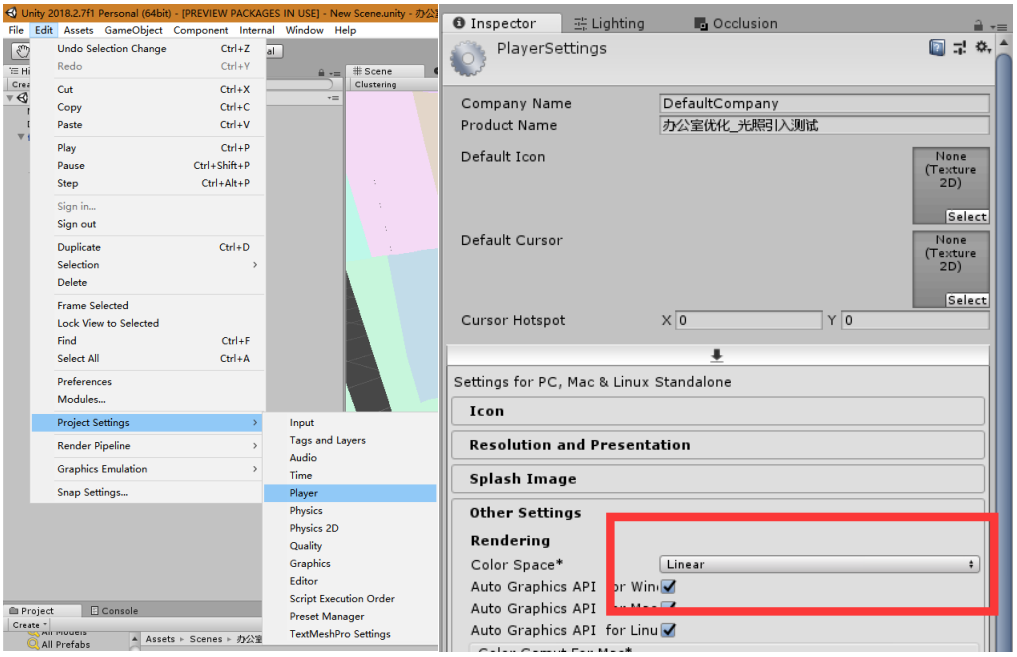


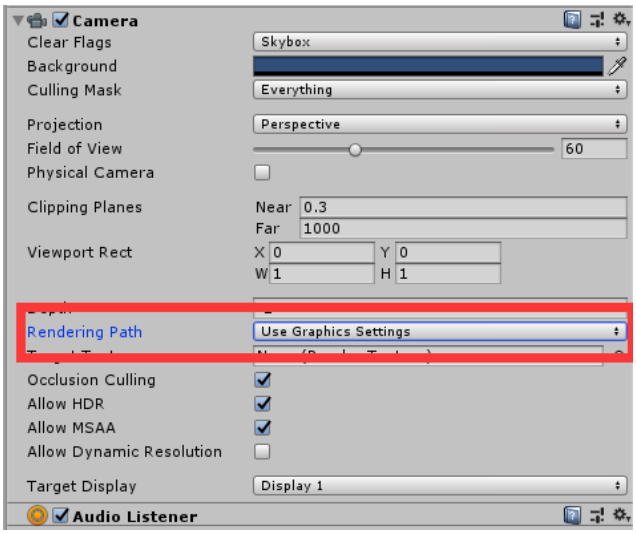
在 Scene 标签页内，要设定场景分辨率、是否烘焙光照方向以及选择场景默认的 Lightmap Parameter。这里主要设定场景分辨率，在室内场景中一般使用 2 或三即可，可以根据需要进行逐物体的优化。



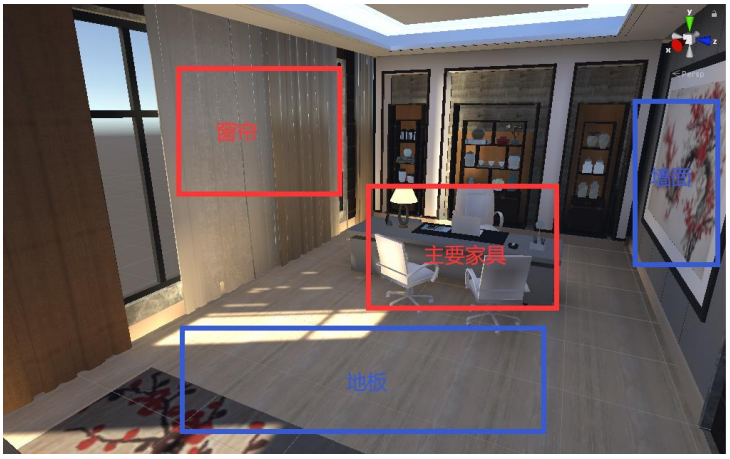
这里就按照默认值设定了，接下来要设置颜色空间和渲染路径。

打开 Edit->Project Settings->Player->Other Settings，更改颜色空间为线性，然后找到场景中的摄像机的 Inspector，选择渲染路径，这里选择了延迟渲染路径，可以支持更多的实施灯光，并且能够支持 Screen Space Reflection。





下面开始优化场景中的 UV Charts 数目，先看一下场景截图：

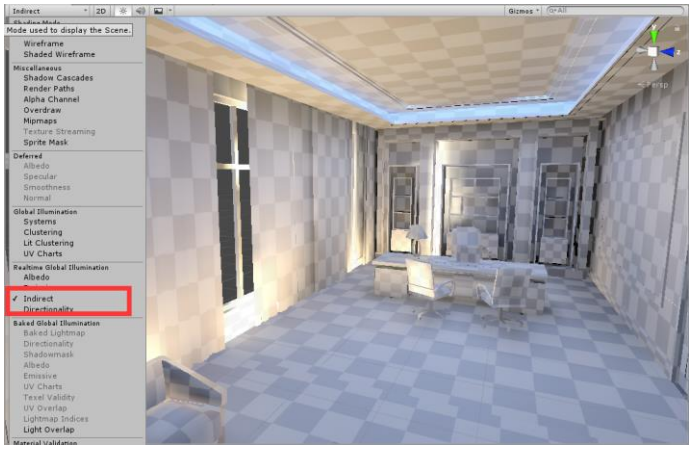


在这里经过测试，窗帘虽然不是我们光照的关注的重点，但是容易出现光照错误，因此需要提升光照精度，主要家具也需要适当提升，尤其对椅子沙发上的曲面来说需要提高精度。而墙壁和地板并不需要那么高的分辨率，因此会适当降低精度。

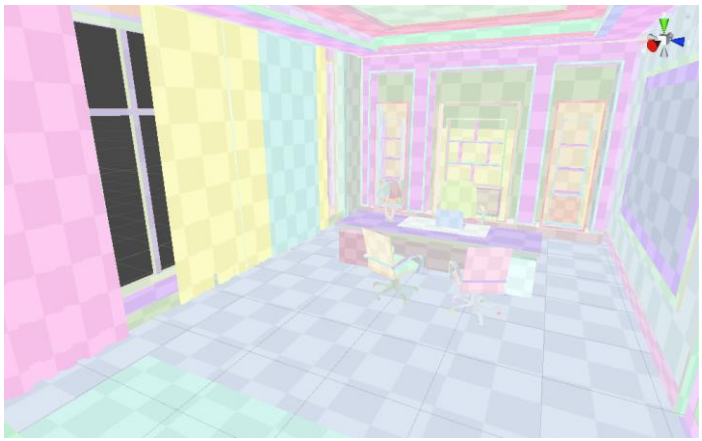
另外像下图这种区域，也容易出现光照错误，因此也要提升分辨率。

这里体现了自动生成烘焙的好处，可以早期帮助我们发现问题并优化，当然这要求我们先使用较低的分辨率，否则可能会导致烘焙失败。

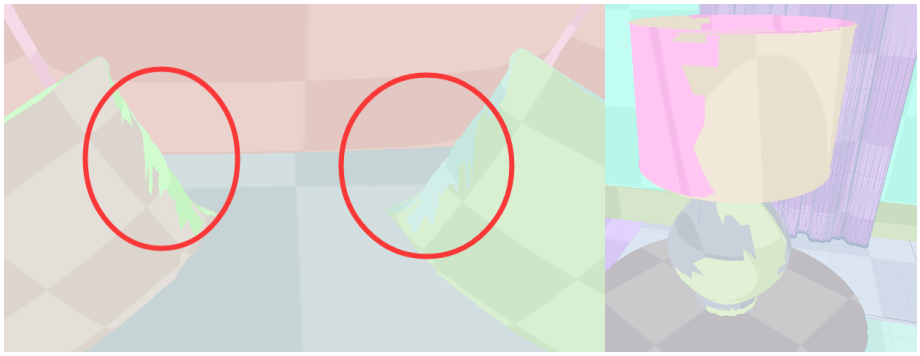
或者也可以按照先验的经验先进行优化，然后在更快的烘焙速度下进行问题修复，提高效率。也可以灵活使用场景内的渲染模式，例如通过间接光渲染模式(Indirect)来观察场景中间接光的受光情况来判断问题。



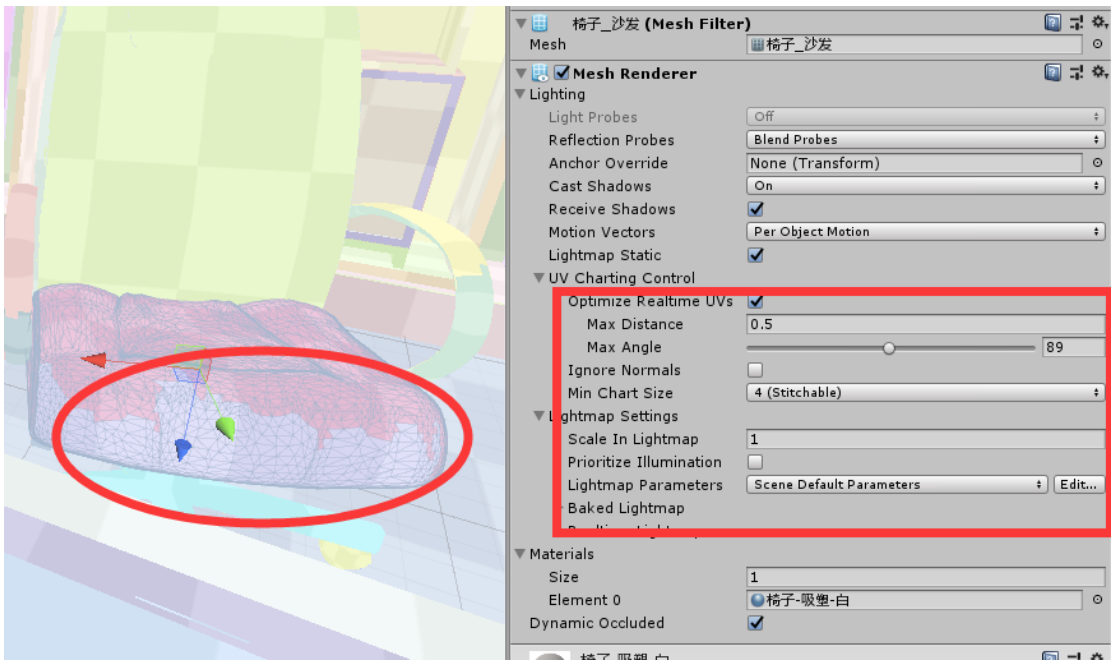
显然上述说过的窗户边上有明显的漏光现象，这些地方可以通过增加遮挡或者提高渲染精度来修复。



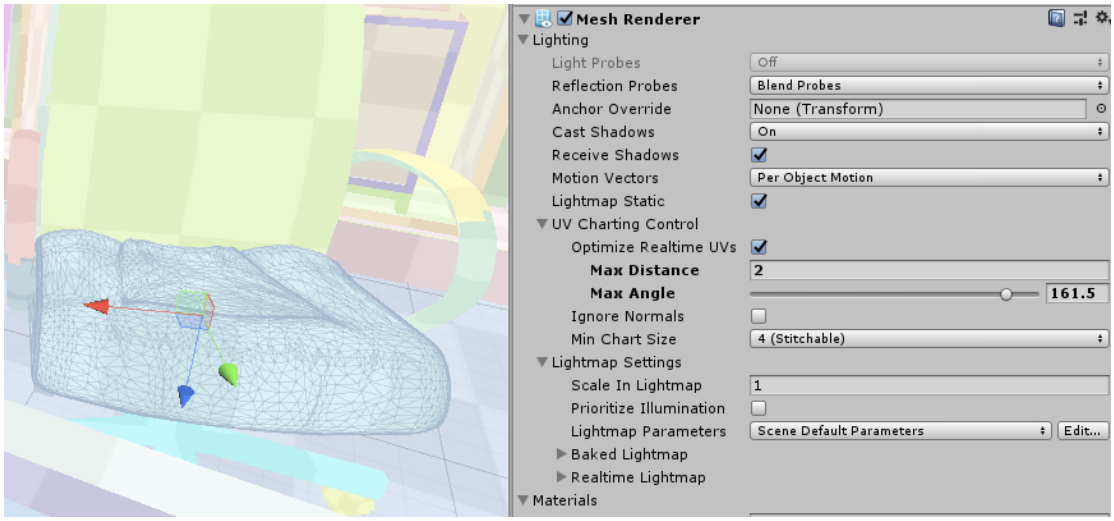
场景中默认情况下的 UVchart 情况。注意到场景中有些部分会产生 UV 接缝的问题，需要重点修复。



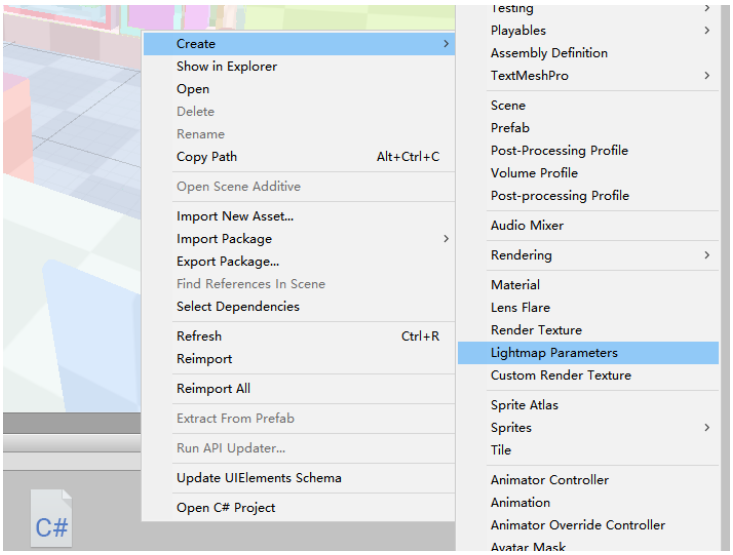
UVchart 和 cluster 的优化原则是类似的，因此需要同步抬升，原则就是在减少这两类分块数目的同时保证场景不出现严重的光照问题。

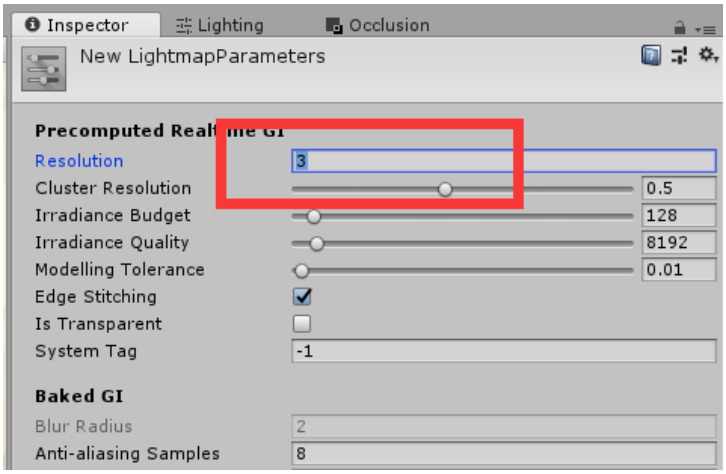


默认情况下会出现的接缝问题，可以通过调整 Optimize Realtime UVs 选项下的 Max Distance 和 Max Angle 的数值来调整。

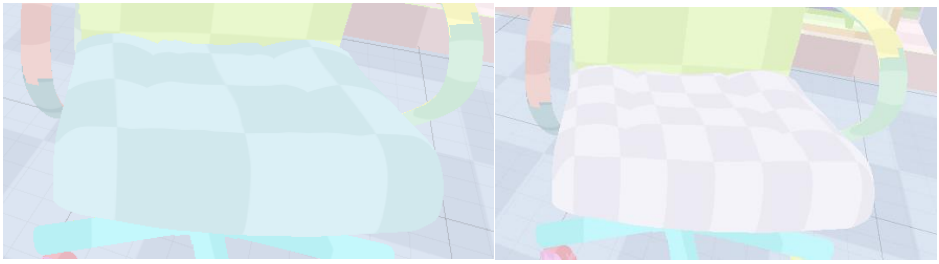


另外可以通过创建 Lightmap Parameter 来调整 texel 分辨率和 cluster 大小，这里我们希望抬升这个沙发座垫的分辨率，可以创建一个高分辨率的 Lightmap Parameter，然后将它赋值给这个物体的 Mesh Renderer。





创建一个新的 Lightmap Parameter，并调整参数



左：使用场景默认的 Lightmap Parameter 右：使用新建的高分辨率 Lightmap Parameter

对场景中的物体都进行这两项优化后，就完成了场景的优化的主要工作了。

优化后的场景

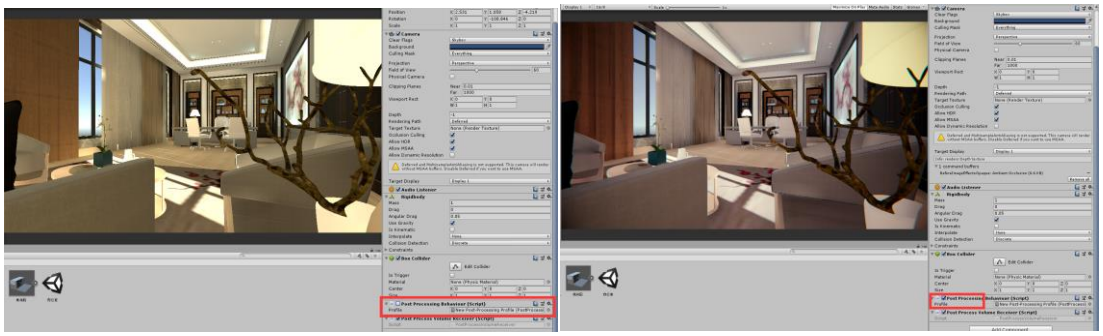
接下来要在场景中摆放光照探针和反射探针，由于反射探针需要材质的配合，而目前的工作进度下，我们的材质还未就位，因此我们考虑先进放置光照探针。

反射探针出于示范作用，也放置了三个：

同时可以注意到，图中已经摆放好灯光照明了。至此我们可以整体提升烘焙质量，然后进行最后的烘焙。

最后一步是添加屏幕后处理。屏幕后处理是利用摄影机在渲染过程中产生的各种画面信息，也就是延迟渲染中存放在 G-Buffer 内的信息，来进行后期的图像处理。屏幕后处理并不是 Unity 内置的功能，我们可以通过 Unity 的插件管理器来加载屏幕后处理包，或者在 Asset Store 内下载免费的屏幕后处理包，这里暂时使用了 Asset Store 内的，实际上官方的包内的功能更加丰富。

使用方式大同小异，只需要给摄像机组件添加 Post Processing Behavior 组件，并创建 Post Processing Profile 文件来实现对应的屏幕后处理即可。



左：屏幕后处理前 右：屏幕后处理后场景更符合人类的审美标准

屏幕后处理包含的内容有：

屏幕空间雾效，抗锯齿，环境光遮罩，屏幕空间反射，景深效果，运动模糊，光晕，颜色映射，颜色索引，镜头色相差效果，胶卷颗粒效果，影晕效果，以及 dithering 技术。

通过这些效果的整合，渲染出来的画面能够得到更好的处理，使之看起来更接近真实空间照片渲染效果。

四、参考文献、参考视频和演讲

【1】Introduction to Lighting and Rendering
https://learn.unity.com/tutorial/introduction-to-lighting-and-rendering?language=en

【2】Creating Believable Visuals
https://learn.unity.com/tutorial/creating-believable-visuals?language=en#5c7f8528edbc2a002053b54c

【3】Precomputed Realtime GI (Global Illumination)
https://learn.unity.com/tutorial/precomputed-realtime-gi-global-illumination?language=en#

【4】Unite Europe 2017 - Bake it 'til you make it: An intro to Lightmaps
https://www.youtube.com/watch?v=u5RTVMBWabg&t=2106s

【5】Unite Berlin 2018 - An Introduction to Lightmapping in Unity
https://www.youtube.com/watch?v=tN33YqhfVtI

【6】LIGHTING in Unity
https://www.youtube.com/watch?v=VnG2gOKV9dw&t=708s

【7】REALTIME LIGHTING in Unity
https://www.youtube.com/watch?v=wwm98VdzD8s&t=189s

【8】REFLECTIONS in Unity
https://www.youtube.com/watch?v=lhELeLnynI8&list=PLPV2KyIb3jR4GH32npxmkXE-AHnlamcdG&index=29&t=0s

【9】How to get GOOD GRAPHICS - Upgrading to HDRP
https://www.youtube.com/watch?v=12gkcdLc77s&list=PLPV2KyIb3jR4GH32npxmkXE-AHnlamcdG&index=50&t=181s

【10】Gamma、Linear、sRGB 和 Unity Color Space，你真懂了吗？
https://zhuanlan.zhihu.com/p/66558476

【11】FBX Importing to Unity 2017/2018+ | Where did my textures go?
https://www.youtube.com/watch?v=xOeodlLTx8g

【12】游戏渲染技术：前向渲染 vs 延迟渲染 vs Forward+渲染(一)
https://www.jianshu.com/p/0994555aefb5