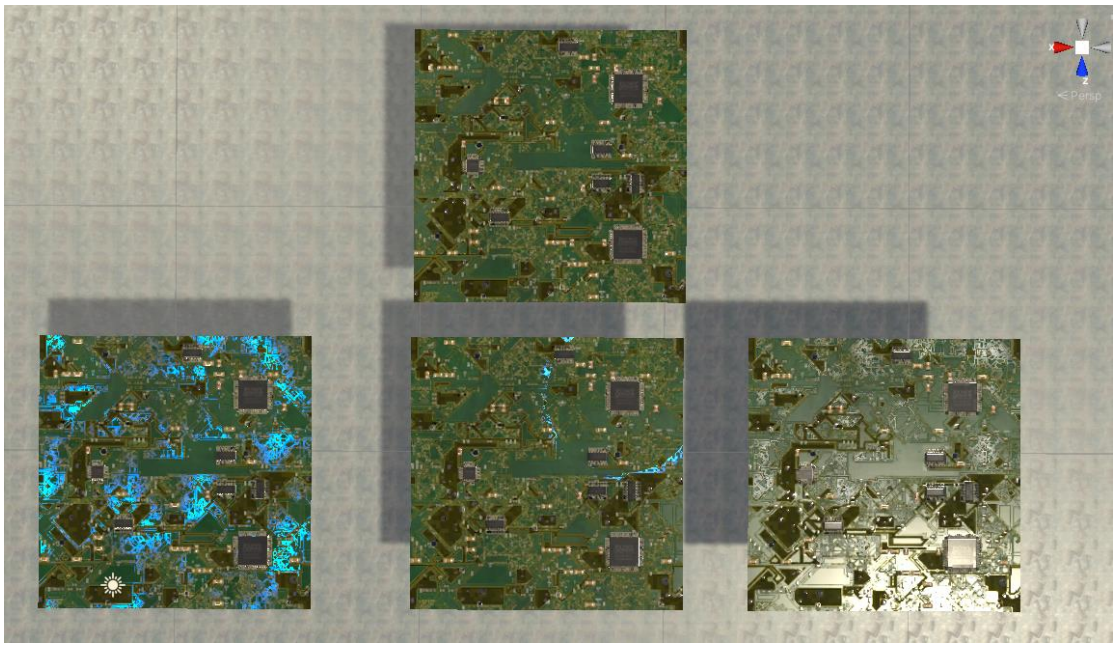
# 游戏引擎原理与实践

深互动媒体硕 20　鄢磊 2020214426

如图所示，PBR 的材质实践效果如图 1 所示。可执行文件在 Build 目录中。

自定义材质实现的电路流动效果分为三类，第一类是类 voronoi 图的随机流动方式（下左），第二类是电流流向右上角芯片中（下中），第三类是电流从右向左流动。

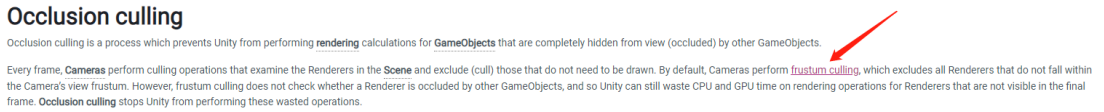


图一 材质系统实践效果

**遮挡剔除与视锥剔除：**

**1. 视锥剔除**

所使用的 unity 版本为 2020.1.5f1c1，查阅 unity manual 可知，unity 在渲染中默认开启了视锥剔除.如下图所示。



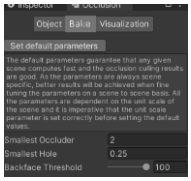
Occlusion culling

Occlusion culling is a process which prevents Unity from performing rendering calculations for GameObjects that are completely hidden from view (occluded) by other GameObjects.

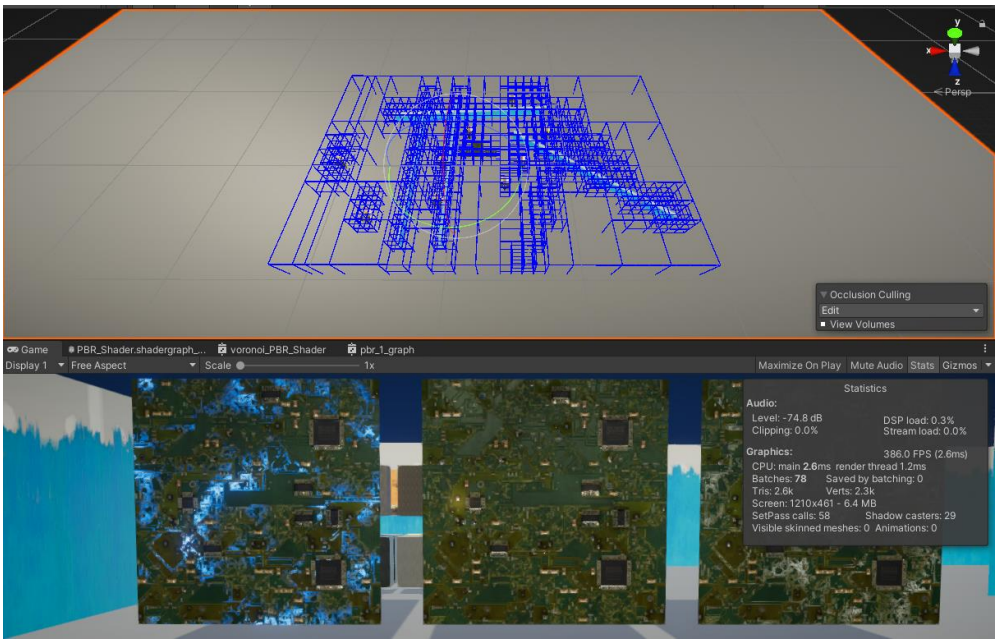
Every frame, Cameras perform culling operations that examine the Renderers in the Scene and exclude (cull) those that do not need to be drawn. By default, Cameras perform frustum culling, which excludes all Renderers that do not fall within the Camera's view frustum. However, frustum culling does not check whether a Renderer is occluded by other GameObjects, and so Unity can still waste CPU and GPU time on rendering operations for Renderers that are not visible in the final frame. Occlusion culling stops Unity from performing these wasted operations.

**2. 遮挡剔除**

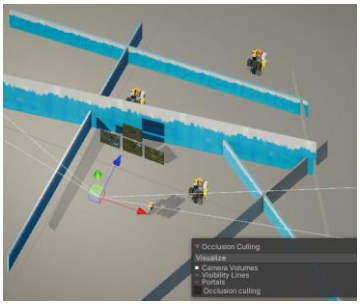
为了开启遮挡剔除，将场景中不动物体（即所有物体）标记为 occlusion static 后，采用了如下参数 bake:

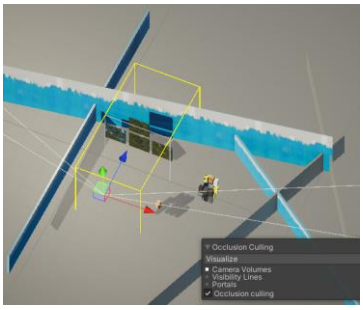



可以得到场景中当前的 occluder 与 stats 如图：




开启遮挡剔除前后的对比图：



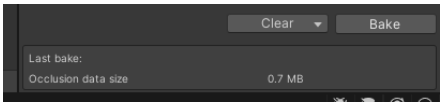




可以看出，在开启遮挡剔除后，后方的物体不再被渲染。

此处已经去掉了默认的天空盒，以减少 verts 数量。

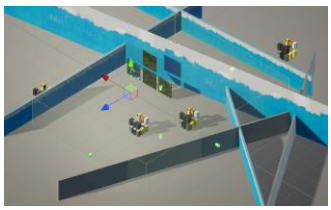
### 3. Occlusion Area

此时尚未采用 Occlusion Area,因此 Bake 出的数据较大，也更花时间，如图所示。

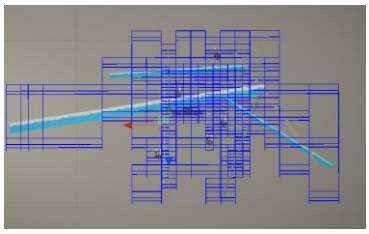



Bake 后数据量 0.7MB

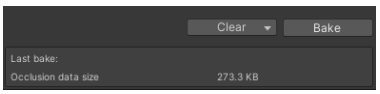
此后，在场景中可移动区域设置了 Occlusion Area,进行 Bake,如图所示：



场景中的 Occlusion Area

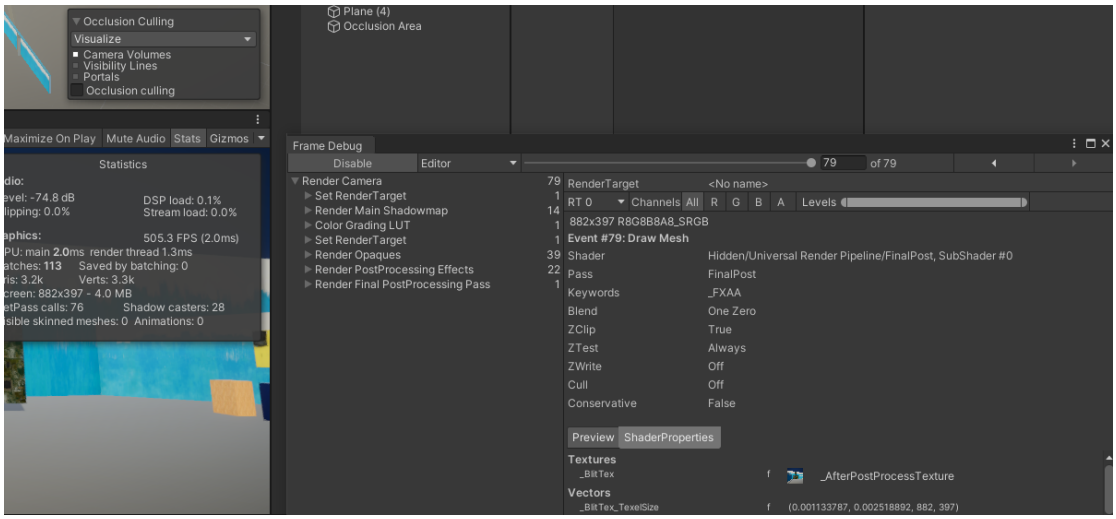


Bake 后的 Occluder 盒



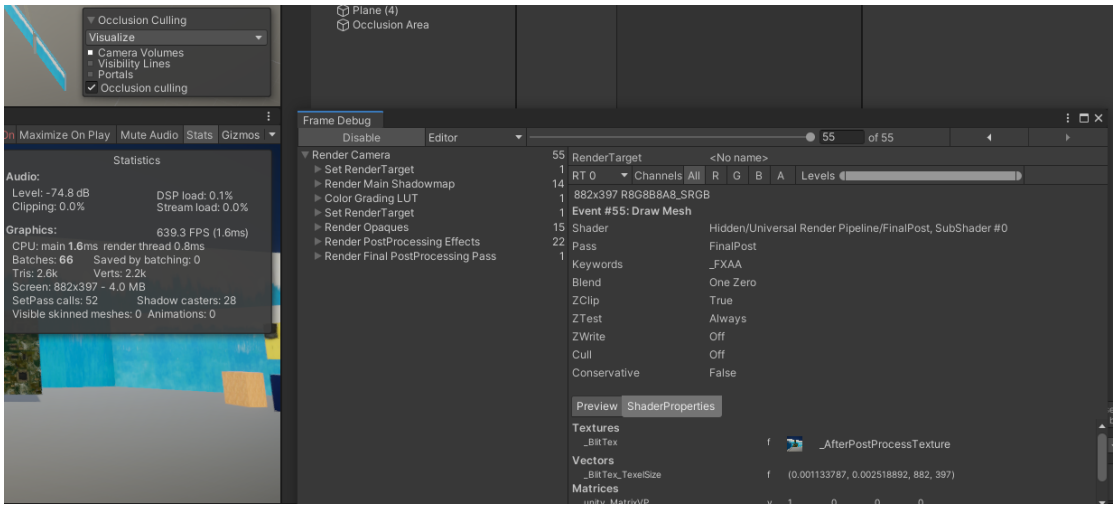

设置 Occlusion Area 后的 data size:273.3KB,缩小 3 倍

首先查看场景中只有一个立方体，一个平面的信息，由于这两个物体的 scale 均经过调整，在 unity 中 batch 数量与物体数量并不一致。立方体之后挡住了 15 个立方体，即其为不可见。现在还将刚才启用的 Occuler 进行 Clear，开启 frameDebugger 与 stats 窗口，可以查看到当前的渲染信息：




可以看到当前的 batchs 为 113，setPass calls 为 73，而渲染事件包含 79 个（包含后处理等），由于项目整体采用的是 URP 的渲染管线，并不能从 Scene 视图中查看到 overdraw 的叠加图，只能从 FrameDebugger 中查看信息，可以看到其中 Render Opaques 部分数量明显远大于场景中可见物体数量。

在保持场景物体位置不变的情况下，对遮挡进遮挡剔除，可以看到当前的 batchs 降到了 66，setPass calls 为 52，而渲染事件包含 55 个，其中 render opaques 的数量明显下降，可见其 overdraw 进行了较大的优化。并且可以看到场景中 Verts 与 Tris 的数量都有所降低，可见确实少处理了一些物体。




# Overdraw

URP performs several optimizations within a Camera, including rendering order optimizations to reduce overdraw. However, when you use a Camera Stack, you effectively define the order in which those Cameras are rendered. You must therefore be careful not to order the Cameras in a way that causes excessive overdraw.

When multiple Cameras in a Camera Stack render to the same render target, Unity draws each pixel in the render target for each Camera in the Camera Stack. Additionally, if more than one Base Camera or Camera Stack renders to the same area of the same render target, Unity draws any pixels in the overlapping area again, as many times as required by each Base Camera or Camera Stack.

You can use Unity's Frame Debugger, or platform-specific frame capture and debugging tools, to understand where excessive overdraw occurs in your Scene. You can then optimize your Camera Stacks accordingly.

**总结：**

由于采用了 URP 渲染管线（URP 在管线中对单个相机会进行渲染顺序的优化从而降低 overdraw，见上图），自定义了多 Pass 的渲染方式，且改变了物体的材质，导致不易定量计算当前视图中 SetPassCall,Batch 数量，overdraw 与物体数量的相对关系。但可以明显看到，随着开启遮挡剔除，各项数值均有明显下降，可证明遮挡剔除对于渲染优化的重要意义。