URP管线，当分辨率从100提升至200后，帧率约为原来的1/4（原来渲染的方块数为100×100改变之后为200×200）。

排序、批处理，然后将40000个点的转换矩阵发送给GPU需要大量的时间。一个矩阵由16个浮点数组成，每个浮点数4个字节，每个矩阵总共64B。对于40000个点来说，每次绘制点时都需要将256万字节(约为2.44 mb)复制到GPU上。URP每帧需要做两次，一次是阴影，一次是常规几何。BRP至少要这样做三次，因为它有额外的仅限深度的通道，而且除了主方向的光外，每个光都要再通过一次。CPU和GPU之间的通信和数据传输量最好降到最低(因为DrawCall是CPU发送给GPU的，如果包含了大量的数据就会导致CPU长期忙碌而GPU空闲，导致渲染流水线的效率不够高)。因为我们只需要点的位置来显示它们，如果数据只存在于GPU端，那将是最理想的。这样就省去了大量的数据传输。但是这样CPU就不能再计算位置了，GPU就得代替它。

图片包含 文本

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图示

描述已自动生成

释放缓冲区后将字段引用置为null，如果图形在播放模式中被禁用或者销毁对象，在下一次运行时被Unity的内存垃圾收集进程回收成为可能。可以避免内存阻塞。

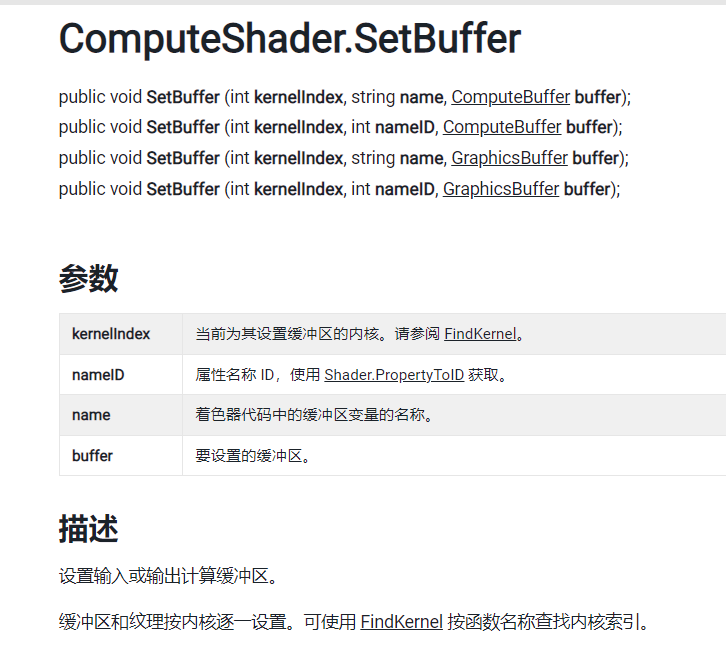
为了计算GPU上的位置，必须为它写一个脚本，它是一个计算着色器。通过Assets / Create / Shader / Compute Shader创建。它将成为我们的FunctionLibrary的GPU等效物。虽然它被称为着色器，并使用HLSL语法，但它的功能是一个通用程序，而不是一个用于渲染事物的常规着色器。

个计算着色器需要包含一个被称为内核的主函数，通过#pragma kernel指令后跟一个名称来表示，比如我们的表面着色器的#pragma surface。将此指令添加为第一行，使用FunctionKernel的名字作为当前唯一一行。#pragma kernel FunctionKernel

通过向内核函数添加numthreads属性来指定每个组应该有多少线程。numthreads的三个参数可用于以一维、二维或三维的方式组织线程。例如，(64,1,1)为我们提供了单维度的64个线程，而(8,8,1)为我们提供了相同数量的线程，但以2D 8×8正方形网格的形式呈现。

为了存储一个位置，我们需要访问位置缓冲区。在HLSL中，计算缓冲区被称为结构化缓冲区。因为我们必须写入它，所以我们需要启用读写的版本，也就是RWStructuredBuffer，并指定缓冲区的元素类型。

Shader.PropertyToID获取计算着色器的属性的标识符。



图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

---------------------------以上都是计算游戏模式下每一帧的所有图形的位置-------------------------

有了GPU上可用的位置，下一步是绘制点，不需要从CPU发送任何转换矩阵到GPU。因此，着色器将不得不从缓冲区中检索正确的位置，而不是依赖于标准矩阵。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

Graphics.DrawMeshInstancedProcedural不使用游戏对象绘制，Unity不知道绘制在场景的什么地方。必须通过提供一个边界框作为附加参数来表示这一点。这是一个轴对齐的框，它表示我们所画物体的空间边界。Unity使用这一点来决定绘图是否可以跳过，因为它最终会出现在摄像机的视野之外。这就是所谓的截锥剔除。不再是计算每个点的边界而是一次计算整个图的边界。

图形用户界面, 文本

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

#pragma target 4.5即目标级别4.5意味着什么?

[这表明我们至少需要OpenGL ES 3.1的功能。](https://docs.unity3d.com/Manual/SL-ShaderCompileTargets.html)它不适用于旧的dx11前gpu，也不适用于OpenGL ES 2.0或3.0。这也排除了WebGL。WebGL 2.0有一些实验性的计算着色器支持，但Unity目前还不支持。  
在支持不足的情况下运行GPU图形最多只能导致所有点重叠，就像现在所发生的那样。所以如果你的目标平台是那样的化，你就必须坚持旧的方法，或者同时包含这两种方法，并退回到低分辨率的CPU图形中

程序化渲染像GPU实例化一样工作，但是我们需要指定一个额外的选项，通过添加#pragma instancing\_options指令（即实例化指令）来表示。在这种情况下，我们必须使用procedural:ConfigureProcedural选项。即#pragma instancing\_options procedural:ConfigureProcedural。这表明表面着色器需要调用每个顶点的ConfigureProcedural函数。它是一个没有任何参数的void函数。把它添加到我们的着色器中。默认情况下，这个函数只会被常规的渲染通道调用。为了在渲染阴影时应用它，我们必须通过添加addshadow到#pragma surface指令来表明我们需要一个自定义的阴影通道。

我们可以通过项目设置关闭异步着色器编译，但这对于Point Surface GPU着色器会是一个问题。幸运的是，我们可以通过添加#pragma editor\_sync\_compilation指令来告诉Unity对一个特定的着色器使用同步编译。这将迫使Unity在第一次使用着色器之前暂停并立即编译它，避免使用虚拟着色器。

函数名的\_float后缀是必需的，因为它表示函数的精度。着色器图形提供了两种精确模式，float或half。后者的大小是前者的一半，因此是两个字节而不是四个字节。节点使用的精度可以显式选择或设置为继承，这是默认值。

文本

描述已自动生成

我们必须为每个图形函数创建一个单独的内核函数，但这是大量重复的代码。我们可以通过创建着色器宏来避免这种情况，就像我们之前定义的PI一样。首先在FunctionKernel函数的上面一行写#define KERNEL\_FUNCTION。

图形用户界面, 文本, 应用程序

描述已自动生成

我们还必须改变核函数的名称。我们将使用function参数作为前缀，后面是Kernel。我们必须保持 function标签分开，否则它不会被识别为着色器参数。要组合两个单词，请使用##宏连接操作符连接它们。

图形用户界面, 文本

描述已自动生成