# **Unity实现无缝大世界--植被**

大世界除了地形，最重要的就是植被了，一般植被都跟地形放在一起。目前来说，渲染植被，一般采用两种办法，静态合批和DrawInstance。几年前，静态合批用得比较多，因为当时植被整体数量比较少，DrawInstance的概念也还没普及。最近几年，DrawInstance的火爆给植被系统提供了最优解。unity提供的植被系统也是这两个方向，但如果说unity移动上地形系统在开了DrawInstance能勉强使用，那植被系统就是完全无法使用。

**存在问题**

unity的植被有两种，tree和detail，两种。

这两种都有非常致命的问题。

首先，tree就是普通的DrawInstance，存在以下几个问题。

1. 他的剔除居然是for循环，不清楚为啥不直接用unity的场景剔除那套（莫名其妙）。不过，就算是场景多线程剔除也是不行的，在植被系统这种情况，四叉树才是正解。
2. 如果prefab有穿插，貌似DrawInstance也会被打断，最终性能会变得非常差。

如果有unity的源码修改权限，倒是可以简单修改一下就能凑合使用（草除外）。

其次，detail这个工具，如果这几年一直在做手游的话，肯定自己项目都实现过一次，就是简单的暴力静态合批。把所有数据，按照区块和类型预处理成模型。存在以下几个问题。

1. 模型顶点爆炸，显存压力大，显示100米，显存就直接炸了，就算不炸，带宽也受不了。
2. 颜色缩放分布，无法精确控制，作为大世界，这点是必须要满足的。

以上，我们急需一套自己的植被系统。

RenderBatchGroup貌似是一个不错的解决方案，CPU多线程剔除+DrawInstance

多年前，我写过一篇这个博客

[狗哥老司机：Unity大世界超多物体渲染---BatchRendererGroup173 赞同 · 42 评论文章](https://zhuanlan.zhihu.com/p/105616808" \t "https://zhuanlan.zhihu.com/p/_blank)

但是，实际测下来，存在以下几个问题。

1. 多线程在移动上，很多时候大概率还是会阻塞，造成CPU某些时候压力过大。
2. 无法遮挡剔除（可以PVS，但是数据过大）。

基于此，加上GPUDriven技术的流行让我发现完美契合植被系统，于是，一套GPUDriven的植被系统由此诞生。

首先，我们将植被的类型分成三种。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 显示范围 | 是否产生阴影 | Lod级数 | 是否摆动 |
| 树 | 3200\*3200（25\*25） | 是 | 4（3级+插片） | 是 |
| 石头 | 640\*640（5\*5） | 否 | 2 | 否 |
| 草 | 384\*384（3\*3） | 否 | 3 | 否 |

## **数据管理**

这里，建议是从Houdini不要区分类型，Houdini只记录ID（无需使用Houdini到Unity的Instance），然后在Unity里弄一个编辑器，维护ID到Prefab的字典，好处是unity能直接显示ID对应的Prefab缩略图，且点击能跳转到Project对应的路径，方便地编经常增删改查prefab。

然后，数据块的粒度建议是128一块，太小文件会太多，太大，粒度不好把控。从Houdini生成到非Asset目录下（减少Unity的Import），做好预览工具。植被数据在大世界里如果不做压缩是非常恐怖的，我们可以做很多逻辑上的压缩。

**Tree数据**

**public** **struct** **InstanceDataAsset**{**public** **byte** Type,Scale,RotX,RotY,RotZ;**public** half3 Pos;}**public** **struct** **TreeBlockDataImport**{**public** int2 BlockId;**public** InstanceDataAsset[] TreeDataList;**public** float3 Center;}

这里注意的是，每个Block存一个Center，这样就能用Half来存位置了，能省不少。然后旋转只需存RotX，然后通过255/360进行运算，Scale的话，可以确定一个范围，然后用255表示范围内的缩放，这样能做到极致压缩。

**石头数据**

同Tree

**草数据**

**public** **struct** **InstanceDataAsset**{**public** **byte** Type,Scale,ColorR,ColorG,ColorB;**public** half3 Pos;}**public** **struct** **ClusterDataAsset**{**public** **uint** ClusterID;**public** InstanceDataAsset[] GrassDataList;**public** float3 Center;}**public** **struct** **GrassBlockDataImport**{**public** int2 BlockId;**public** List<ClusterDataAsset> ClusterDataList;}

压缩方法同上，但是这里我是按32\*32一个Cluster再分了一次，主要是因为算法需要，后面谈及算法了，就知道为啥了。

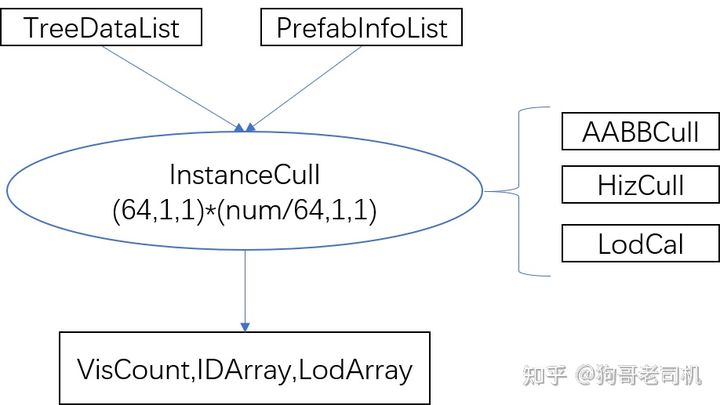
## **树**

按照128为粒度，加载前后左右各12个格子数据，构建一个数组，这里记得把压缩数据转成正常数据（使用JobSystem），这里得到了一个TreeData的数组。

然后根据TreeData的Type，我们能知道这些TreeData都有哪些Type，获取这些Type的AABB以及LOD信息，保存成一个数组，然后建立一个Type到Index的字典。

准备好了数据后，输送到computerShader进行剔除和渲染数据生成。最终我们要得到，一个按Type和Lod排序的列表，以及每个Type和Lod对应的DrawIndirect，这里树Lod分四级，其中最后一级单独合并，用插片树一起画出来，所以是分成3\*Type + 1堆数据，然后进行渲染。大概分成一下这几步。

**1，初步剔除**



通过包围盒，Hiz剔除，并计算出Lod，用InterlockedAdd得到两个数组，分别存放在字典中的位置Id和Lod，这里建议压缩成一个数组，用4位存Lod，int剩下的字节存Id，这里得到的被剔除后依然可见的id，但是由于是多线程计算，他们是被一股脑扔到一个数组里了。

**2，计算每个Type的lod个数和偏移，并算出下一步CS的参数。**

上一步我们得到一个混乱的剔除后的数组列表，现在我们要把这些数据按照Type和LOD排序分成Type\*LOD段数据，这在CPU串行下比较好计算，在GPU，我们得分成两步进行，首先，我们得计算出每一段的数量是多少。

[numthreads(THREAD\_GROUP\_SIZE\_X, 1, 1)]**void** CalcIndexOffsetCS(**in** uint3 dispatchThreadID : SV\_DispatchThreadID){

**uint** tid = dispatchThreadID.x;

**if** (tid < TreeRenderTypeCount){

**uint** nowId = tid \* LOD\_LEVLE;

**uint** needAdd = IDArray[nowId];

**for** (**uint** i = 1; i < LOD\_LEVLE; i++){

**if** (needAdd > 0){

InterlockedAdd(DrawIndirectArgsBuffer[(nowId + i) \* 5 + 4], needAdd);

}

needAdd += VisCount[nowId + i];

}

**if** (needAdd > 0){

**for** (i = tid + 1; i < TreeRenderTypeCount; i++){

**for** (**uint** lod = 0; lod < LOD\_LEVLE; lod++){

InterlockedAdd(DrawIndirectArgsBuffer[(i \* LOD\_LEVLE + lod) \* 5 + 4], needAdd);

}

}

InterlockedAdd(DrawIndirectArgsBuffer[(TreeRenderTypeCount \* LOD\_LEVLE \* PREFAB\_RENDER\_COUNT) \* 5 + 4], needAdd);

}

}

**if** (tid == 0){

**uint** RenderNum = min(RenderRearrangeDispatchIndirectArgsBufferRW[3], TreeRenderTotalCount);

RenderRearrangeDispatchIndirectArgsBufferRW[4] = RenderNum;

RenderRearrangeDispatchIndirectArgsBufferRW[0] = (RenderNum + THREAD\_GROUP\_SIZE\_X - 1) / THREAD\_GROUP\_SIZE\_X;

RenderRearrangeDispatchIndirectArgsBufferRW[1] = 1;

RenderRearrangeDispatchIndirectArgsBufferRW[2] = 1;

RenderRearrangeDispatchIndirectArgsBufferRW[3] = 0;

}}

**3，按Type和Lod重新排布Visible列表，并计算DrawIndirect参数。**

在知道每一段数目是多少后，就能将第一步的无序列表按照Type和Lod，组织到各个对应的数据段中去了。

[numthreads(THREAD\_GROUP\_SIZE\_X, 1, 1)]**void** GPUInstanceRearrangeCS(**in** uint3 dispatchThreadID : SV\_DispatchThreadID){

**uint** tid = dispatchThreadID.x;

**if** (tid < RenderRearrangeDispatchIndirectArgsBuffer[4])

{

**uint** renderInfo = VisArray[tid];

**uint** lodLevel = renderInfo & 0xf;

**uint** RenderInstanceIndex = renderInfo >> 4;

**if** (lodLevel < LOD\_LEVLE)

{

**uint** RenderIndex = (GetInstanceTypeIndex(RenderInstanceIndex) \* LOD\_LEVLE + lodLevel) \* 5;

**uint** CurTypeLODInstanceOffset = DrawIndirectArgsBuffer[RenderIndex + 4];

**uint** CurIndex;

InterlockedAdd(DrawIndirectArgsBuffer[RenderIndex + 1], 1, CurIndex);

\_RenderIndex[CurTypeLODInstanceOffset + CurIndex] = RenderInstanceIndex; [unroll]

**for** (**int** i = 1; i < PREFAB\_RENDER\_COUNT; i++){

**uint** baseIndex = RenderIndex + 1 + i \* TreeRenderTypeCount \* LOD\_LEVLE \* 5;

InterlockedAdd(DrawIndirectArgsBuffer[baseIndex], 1);

DrawIndirectArgsBuffer[baseIndex + 3] = CurTypeLODInstanceOffset;

}

}

**else**{

**uint** RenderIndex = TreeRenderTypeCount \* LOD\_LEVLE \* PREFAB\_RENDER\_COUNT \* 5;

**uint** CurTypeLODInstanceOffset = DrawIndirectArgsBuffer[RenderIndex + 4];

**uint** CurIndex;

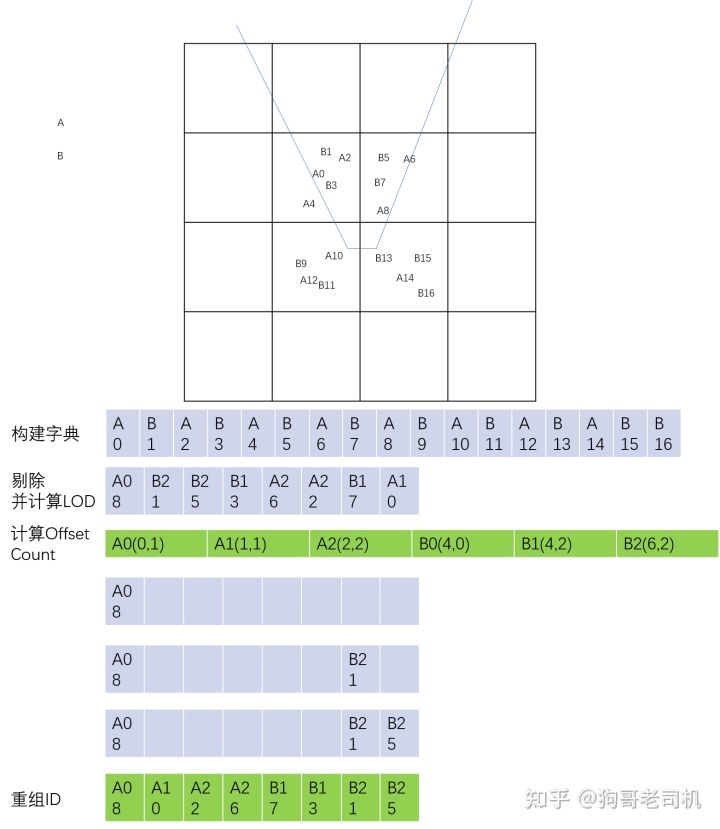
InterlockedAdd(DrawIndirectArgsBuffer[RenderIndex + 1], 1, CurIndex);

\_RenderIndex[CurTypeLODInstanceOffset + CurIndex] = RenderInstanceIndex;

}

}}

总结前三步



**4，按照123步重新处理阴影**

这里阴影无需使用LOD，直接用LOD0就好了，然后如果用到了Cascade，那么Lod就相当于CascadeLevel了。这里记得从URP里取出各级Cascade的参数，然后利用这些参数进行CalInShadow，替换掉第一步的Cull和LodCal。

inline **bool** CalcInShadow(float4 objSphere, float4 shadowSphere){

float3 dir = objSphere.xyz - shadowSphere.xyz;

**float** projectDis = dot(dir, \_LightDirection);

**float** pointToLineDis = sqrt(dir.x \* dir.x + dir.y \* dir.y + dir.z \* dir.z - (projectDis \* projectDis));

**float** totalLen = objSphere.w + shadowSphere.w;

**return** abs(pointToLineDis) < totalLen;}

**5，提交渲染DrawCall**

在Update里循环调用Graphics.DrawMeshInstancedIndirect渲染各个Prefab的各级Mesh和材质（ShadowCastingMode.Off），然后再统一调用一次插片树的渲染。在VS里，我们现在有了字典TreeDataList以及RenderIndex，通过InstanceID，我们就能定位到TreeDataList的数据，然后就能正确设置平移旋转缩放颜色等信息。这里注意，DrawMeshInstancedIndirect的IndirectBuffer的第四位能保存InstanceID的偏移，但是只在vulkan上有效，在GLES和IOS得自己从DrawIndirectBuffer里取出第四位偏移进行计算。

**6，提交阴影DrawCall**

监听Cascade的ShadowDraw调用CommandBuffer.DrawMeshInstancedIndirect渲染对应Slice的Mesh。

## **石头**

石头的流程跟树一样，只是LOD为2级，且没有插片，也无需阴影，这里不作过多叙述。

## **草**

草的数量远远大于树和石头，如果采用上述办法，那么会出现两个问题。

第一，数据的更新太大，每次重新构建SSBO比较费劲。

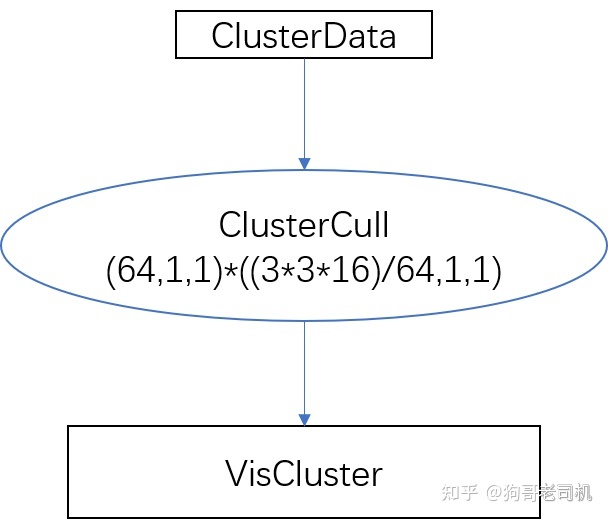
这个问题可以为每个块申请3\*3个固定大小的数组，每次更新只把改变了的数据更新，基本每次更新三块数据，至多5块。

第二，每颗草走一次剔除，cs的线程数有点吃不消。

这就是为啥我们要针对草做Cluster了，我们可以先做一级Cluster的剔除，得到一个粗略的剔除（1/3-1/2），然后再对cluster里的草做进一步剔除。步骤如下：

**1，cluster的剔除**

通过视椎体和Hiz剔除得到可见Cluster列表。



**2，Cluster里草的数量和偏移计算，以及下一步CS的IndirectBuffer计算。**

这一步主要是为了将可见Cluster里的草数据平展开，需要计算出每个cluster的偏移。

[numthreads(THREAD\_GROUP\_SIZE\_X, THREAD\_GROUP\_SIZE\_Y, THREAD\_GROUP\_SIZE\_Z)]

inline void CalcVisClusterOffsetCS(in uint3 \_dispatchThreadID : SV\_DispatchThreadID)

{

uint threadID = \_dispatchThreadID.x;

uint clusterOffset = 0;

uint visClusterNum = dispatchIndirectArgsBufferRW[2];

if (threadID < visClusterNum){

for (uint cidx = 0u; cidx < threadID; cidx++){

int prevVisClusterIndex = visibleClusterIndexBuffer[cidx];

clusterOffset += clusterCullDataBuffer[prevVisClusterIndex].grassInstanceCount;

}

int grassNum = clusterCullDataBuffer[visibleClusterIndexBuffer[threadID]].grassInstanceCount;

InterlockedAdd(dispatchIndirectArgsBufferRW[1], grassNum);

}

visClusterOffsetBufferRW[threadID] = clusterOffset;

GroupMemoryBarrierWithGroupSync();

if (threadID == 0){

uint grassNum = dispatchIndirectArgsBufferRW[1];

dispatchIndirectArgsBufferRW[0] = (grassNum + THREAD\_GROUP\_SIZE\_X - 1) / THREAD\_GROUP\_SIZE\_X;

dispatchIndirectArgsBufferRW[3] = grassNum; // grass num

dispatchIndirectArgsBufferRW[4] = dispatchIndirectArgsBufferRW[2]; // cluster num

dispatchIndirectArgsBufferRW[1] = 1;

dispatchIndirectArgsBufferRW[2] = 1;

drawArgsComputeBuffer[GrassTypeNum \* LOD\_LEVLE \* 5] = 0;

}

}

**3，Grass剔除，获得VisibleIndex和Lod等级。**

跟树的第一步类似，只是数据组织形式不一样。

[numthreads(THREAD\_GROUP\_SIZE\_X, THREAD\_GROUP\_SIZE\_Y, THREAD\_GROUP\_SIZE\_Z)]

inline void GrassCullCS(in uint3 \_dispatchThreadID : SV\_DispatchThreadID)

{

uint threadID = \_dispatchThreadID.x;

uint visClusterNum = dispatchIndirectArgsBuffer[4];

int curRange = 0;

for (uint visClusterOffsetIdx = 1; visClusterOffsetIdx < visClusterNum; visClusterOffsetIdx++){

curRange += (threadID >= (visClusterOffsetBuffer[visClusterOffsetIdx]));

}

int clusterIndex = visibleClusterIndexBuffer[curRange];

int grassIndexInCluster = threadID - visClusterOffsetBuffer[curRange];

int grassIndex = clusterIndex \* GRASS\_COUNT\_PER\_CLUSTER + grassIndexInCluster;

uint grassType = grassInstanceTypeBuffer[grassIndex];

uint typeOffset = instanceTypeTableBuffer[grassType];

if (typeOffset < GrassTypeNum){

AABB aabb = grassAABBDataBuffer[typeOffset];

aabb = TransGrassAABB(aabb, grassIndex);

if (TestAABB(aabb))

{

float3 center = (aabb.Min + aabb.Max) \* 0.5;

float dist = distance(LodCamearPos, center);

int lod = calcGrassLOD(dist);

if (lod < LOD\_LEVLE)

{

int instOffset = CalcTypeLODIndirectArgStartOffset(typeOffset, lod) \* 5 + 1; // add instance count

int visIndex = 0;

InterlockedAdd(drawArgsComputeBuffer[instOffset], 1);

InterlockedAdd(drawArgsComputeBuffer[GrassTypeNum \* LOD\_LEVLE \* 5], 1, visIndex);

visGrassInstanceIndexBufferRW[visIndex] = (grassIndex << 3) + lod;

}

}

}

}

**4，计算每个Type的lod个数和偏移，并算出下一步CS的参数。（同树第2步）**

**5，按Type和Lod重新排布Visible列表，并计算DrawIndirect参数。（同树第3步）**

**6，在Update里循环调用Graphics.DrawMeshInstancedIndirect渲染各个Prefab的各级Mesh和材质。（同树第5步）**

上述流程做完，主体功能完成，下面说一下能优化或者需要额外注意的点

## **贴图代替SSBO**

在某些机型（华为）GLES下，VS访问SSBO的个数是0，这里可以用贴图替换SSBO，用贴图替换SSBO除了有兼容性好处，还有一点是，cs是无法使用half这种半精度类型的，我们可以做进一步优化，使SSBO的大大内存减少，不仅能优化显存还能使Cache命中更高。

## **Pre-Z**

现在移动端各个平台都不推荐使用Pre-Z，因为芯片自带背面消隐，但是由于树和草OverDraw实在是太高了，实际测下来，开Pre-Z确实要好一点，特别是高通机器上。这个可以以项目测下来的实际数据为准，甚至针对不同平台设置Pre-Z开关。

开Pre-Z也比较简单，在渲染不透明物体之前，先打开AlphaTest，写一遍深度，ColorMask置成0（有人说设置这个反而效率低，我没测试过）。在渲染物体本身时，把深度测试设置成Equal，关闭AlphaTest。

## **DrawCall调用时机**

可以看到物体的渲染是在Update里调用Graphics的方法，而没有在URP里单开一个RenderFeature或者RenderPass来处理，明明我这里无需进行场景剔除，为啥还要这样。主要原因是灯光和LightProbe这些跟Mask有关的渲染数据，如果要正确设置好，最好还是走一次Unity的场景流程，不然自己手动设置这些数据太麻烦了。

## **CrossFade**

树的CrossFade采用Dither的方法，就是在cs计算lod时，在lod计算卡在边缘时，可以把两级lod都加入到可见列表，并保存weight，然后在渲染的时候，再使用传统的dither来处理CrossFade的办法。

草的CrossFade不建议采用dither方法，或者说只开dither依然解决不了切换LOD时的断裂，因为草分布比较密集，这样就会看到有一条LOD线往前推，这里就需要在计算lod时，做一个扰动，然后针对远处突然出现的草，可以让草在远处的时候把草的Y值降低，做一个靠近相机的lerp，这样可以让草是慢慢从地下冒出来的感觉。

## **Feedback**

如果不做任何处理使用DrawMeshInstancedIndirect，我们使用的是DrawBuffer，无法得知到底需不需要调用这个Draw，那么我们需要调用全量的DrawCall，即使渲染0个物体。

Vulkan和Ios平台是支持异步回读的，unity的接口为

[AsyncGPUReadbackRequest​docs.unity3d.com/ScriptReference/Rendering.AsyncGPUReadbackRequest.html](https://link.zhihu.com/?target=https://docs.unity3d.com/ScriptReference/Rendering.AsyncGPUReadbackRequest.html" \t "https://zhuanlan.zhihu.com/p/_blank)

可以通过异步回读上一帧的结果来确定是否需要提交这个DrawCall。这里还需要做一些防御手段，如：一个Prefab只要有一级LOD显示，那么这整个Prefab的LOD都需要提交，不然往前跑，切换LOD会出现植被有一帧消失。即使做了很多优化手段，还是会在转动视角的时候出现植被的显隐，好在对画面没有那么大的崩坏，勉强能忍。其实玩的很多游戏，甚至主机游戏都有这个问题。

## **DrawProcedural**

即使采用GPU植被，性能消耗依然很大，最主要还是OverDraw的问题，再一个即使在使用FeedBack做优化后，由于种类偏多，DrawCall数依然不小。如果我们能通过DrawProcedural实现真正的GPUDriven就能让一个DrawCall画完所有植被，且做到Cluster级别的剔除，大大减少OverDraw。前提是我们要处理很多事情，如：模型拆分成Cluster，植被贴图VT，材质参数最好一样等。整个工作流会非常复杂，且DrawProcedural似乎对移动端支持得不是很好，也有可能是我没用对（写了个简单demo测试），最终放弃了这个方法。