Merge-Instancing

先从合批技术说起

静态合批

将场景中相同材质的静态物体的Mesh在世界坐标空间下合并成一个大的Mesh。但同时保留单个的Instance。记录每个SubMesh在大Buffer中的索引，通过剔除计算选择渲染。

静态合批有时并不会减少DC。如场景中有五个相同材质的Instance，A，B，C，D，E。

五个Instance合并成了一个大的Vertex和Index。

当相机能同时看到五个Instance时，就整个提交。一个DC就提交五个物体

但当只能看到A，B时，只提交整个的Vertex，提交两个DC，每个DC只提交各自Instance的Index。

优点：

- 减少Vertex的提交，特定情况下减少DC。
- 可以将不同的Mesh合并到一起

缺点：

- 增大包体内存
- 顶点Buffer中可能有多个重复的顶点和索引（在Obj空间下。
- 合并的索引不能超过65536（Unity的限制，因为采用了16位的Index，可以用脚本放开这个限制）

Instancing

硬件技术。

各种图形API都提供了Instancing技术。即相同Shader相同Mesh只在CPU端提交一次，在GPU端重复绘制的技术。可以通过Buffer来将不同Instance之间不同的参数传递给GPU，如位置，颜色，贴图等。适合大量重复物体的绘制，如草，树，石头等。顶点数尽量多一点，不然对GPU并行不友好。

优点：

- CPU端只提交一次GPU端绘制多次
- 不占用CPU端合并的时间

缺点：

- 仅支持相同Mesh和同一个shader的物体

Merge-Instancing

受限于硬件的限制，只能对相同Mesh的物体做Instance。**Merge-instancing**为了解决这个问题被提了出来。

雪崩工作室在2012年的Siggraph提出了这种技术。链接在没有状态切换的而情况下，理论上可以一个DC绘制所有模型。

现在记录一下这篇文章的内容。

文章第一部分的主题是粒子特效来带的ROP带宽问题，和MergeInstanc无关

现代GPU中，ALU和TEX的增长速度要远超ROP

> ROP即Warp内数据和显存交换的地方，如深度，模板测试，Blende混合。是带宽问题的最大头。

一些典型会造成ROP出现性能瓶颈的情况：

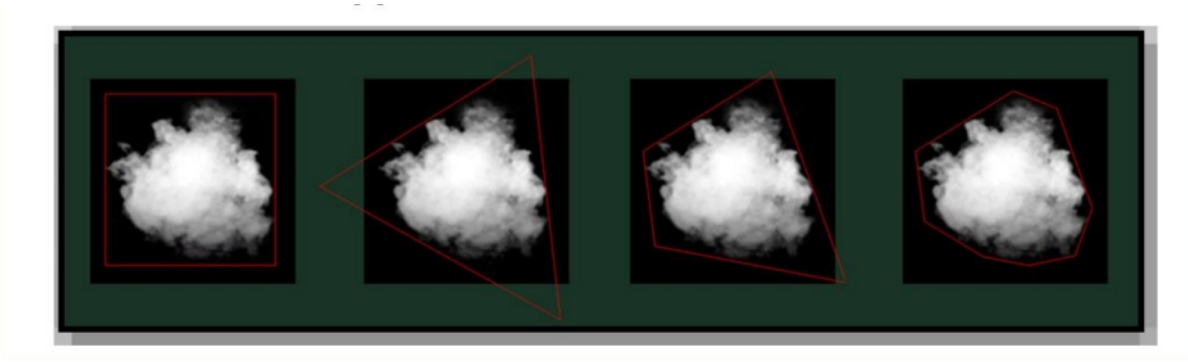
- 粒子

  模型简单，shader复杂，且是半透明
- 云
- 广告牌算法
- GUI

一些常见的解决办法：

- 减少渲染分辨率
- 文章提出使用MSAA来提高ROP的吞吐量

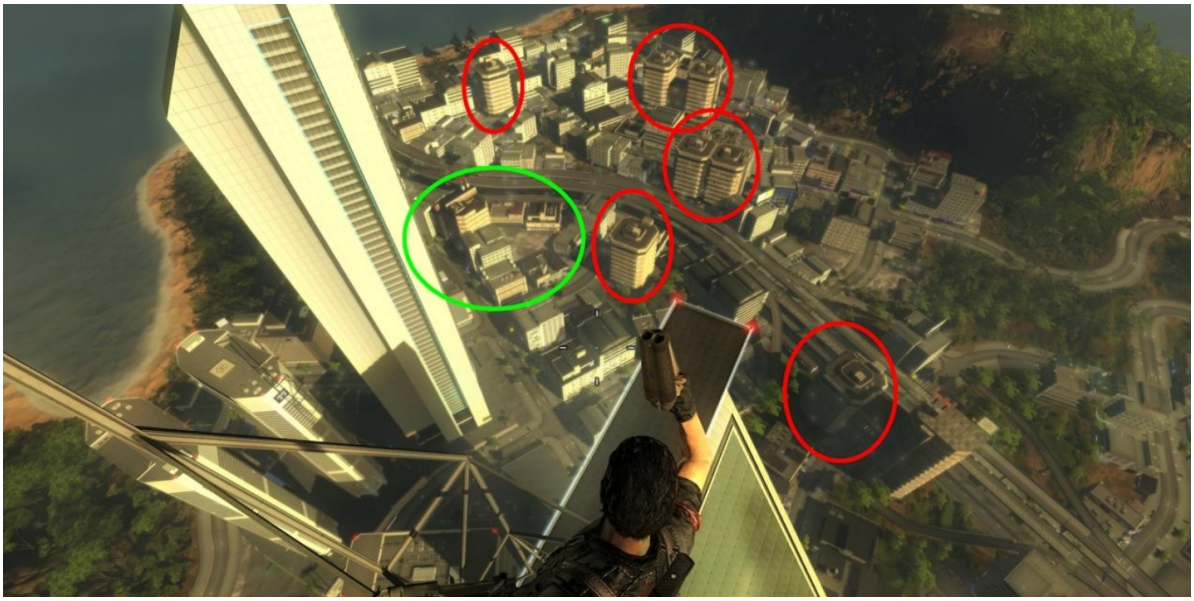
在粒子的渲染中有很多alpha = 0的地方，会造成ROP填充的浪费，可以通过修剪粒子的几何来减少这个问题



Merge-Instancing部分

减少Draw的调用，有两个标准方法，静态合批，Instancing。然而它们都有各自的局限性，文章提出了一种结合静态合批和实例化的方法，称为**Merge-Instancing**.

> 这种方法也是后面GPU Pipeline的前提。



上面这张图是《正当防卫2》的游戏截图。

在这张图中，你可能认为红色圈起来的建筑应该是绘制实例化的主要目标。因为它们是同一个模型，只是不同的位置而已。然而这意味着在CPU端剔除时，它们必须合并到一个巨大的包围盒中来参与剔除，这将大大降低剔除的有效性。当然，也可以一个个剔除后再在CPU端记录位置使用Instancing绘制。但是这样做，我们要在选择阶段付出更高的代价，甚至超过了减少DC带来的好处。

Instancing的理想情况下，我们喜欢绿色圈起来的建筑，因为它们彼此很接近，即使作为单个实例参与剔除，也无所谓，因为它们很接近。但是，相邻的房子很少都是相同的网格。所以基于这种情况，Merge-Instancing来了。

Instancing伪代码

```
for(int instance = 0;instance < instance_count;instance++)
{
    for(int index = 0; index<index_count;index++)
    {

 VertexShader(VertexBuffer[IndexBuffer[index]],InstanceBuffer[instance]);
    }
}
```

过于抽象，我们举个例子。

场景中的实例。

A：Vertex{{0.0，0.0，0.0}，{0.0，0.5，0.0}，{0.5，0.0，0.0}}；Index{0，1，2}。画10个

B：Vertex{{0.0，0.0，0.0}，{0.0，0.5，0.0}，{0.5，0.0，0.0}，{0.5，0.5，0.0}}；Index{0，1，2，1，3，2}。画5个

对于A来说：

instance_count = 10，index_count = 3。

对于B来说：

instance_count = 5，index_count = 6。

Merge-Instancing伪代码

```
//顶点着色器一共需要处理vertex_cout = instance_count * freq 个顶点。
//instance_count是总的实例数
for(int vertex = 0;vertex < vertex_count;vertex++)
{
    //计算InstanceID
    int instance = vertex / freq;
    // 该点在实例中
    int instance_subindex = vertex % freq;
    //获取该实例在索引缓存中的偏移
    int indexbuffer_offest = InstanceBuffer[instance].IndexOffset;
    //计算该点的索引
    int index = IndexBuffer[indexbuffer_offest+instance_subindex];
    //vertexbuff[index] 从顶点缓存中获取顶点信息(fetch vertex)
    //InstanceBuffer[instance_id] 从实例缓存获取实例信息
    //执行顶点着色器
    VertexShader(VertexBuffer[index],InstanceBuffer[instance]);
}
```

首先将场景中所有Mesh的顶点放到同一个顶点缓存（VettexBuffer）中。把所有Mesh的索引也合并到一个索引缓存中（IndexBuffer）。实例缓存中保存每个实例特有的数据，另外需要额外存储一个该实例在索引缓存中的偏移，用于表示该Instance属于哪个Mesh。该技术要求每个Mesh的索引数相同，记为freq，用于手动计算InstanceID。

Merge-Instance不会减少顶点着色器的开销，应该在剔除和LOD应用之后使用，减少不可见顶点处理的规模。同时，该技术需要在实例缓存中额外存储IndexOffest，数据量的增加不算多。合并实例的缺点在于Mesh的索引数量固定，换句话说每个Mesh的三角形数固定。如果一个Mesh不足freq/3个三角形需要退化三角形补足；如果一个Mesh多于freq/3三角形，需要将该Mesh拆分成两个更小的Mesh。这自然会增加顶点缓存的内存的开销。如果目标平台DC调用占用不高，用实例化绘制更简单，对美术制作也更加友好。

开始实现文章

解决第一个问题：如何在VS中获取顶点

从三角形的数据结构说起：

虎书中使用**Manifold**来定义Mesh。一个**Manifold**大体上没有缝隙，并且能够将空间完整的分割位里面和外面。Manifold描述了一种网格中三角形彼此连接的关系，这种关系可以让我预测其形状而不必在代码里进行繁琐的检测，因此其定义和网格中的顶点以及边相关。

对于**边**来说，manifold要求所有使用这个边的三角形能够“扯平”成一个平面，那一条边最多只能同时被两个三角形使用。

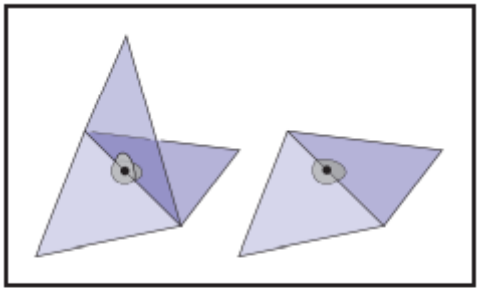


Figure 12.1. Non-manifold (left) and manifold (right) interior edges.

对顶点来说，manifold要求所有使用这个顶点的三角形也能够“扯平”成一个平面。

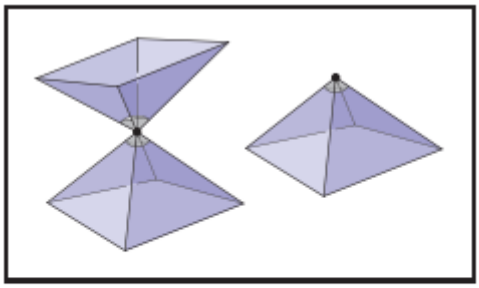


Figure 12.2. Non-manifold (left) and manifold (right) interior vertices.

即：

- 每个边都必须被一个或者两个三角形使用
- 每个顶点都必须被一组相连的三角形共享

网格最简单的形式就是存储所有三角形，如此我们只需要定义一个三角形类和一个顶点类：

```
class Triangle
{
   Vertex v[3];
}
class Vertex
{
    vector3 pos;
}
//举个例子，一个平面有最少由两个三角形组成，那平面的表示既是
// vertex a,b,c,d
//Plane {a,b,c,a,c,d}
//可以看到a和c两个顶点被存储了两次，当一个模型很大时，这种存储方式重复存的顶点数量会巨大，造成浪费。
```

做一个简单的改进，三角形类只存储索引，而顶点存储到另一个数组中。

```
index = {0,1,2,1,2,3}
vertex = {a,b,c,d};
```

对应到GPU中递交渲染指令时，两种存储方式调用的指令也有所不同，以OpenGl为例

第一种绘制方式为：**glDrawArrays**,gpu会从vertexBuffer中依次三个三个的取顶点数据。

第二种对应的绘制方式为：**glDrawElements**，gpu会根据IndexBuffer中的索引去Vertex中取顶点数据。

了解了GPU绘制的两种方式以后，来看两个在VS中的关键参数

- SV_VertexID
    - 当使用**glDrawArrays**渲染时，该ID的值时是，当前处理的顶点数是整个顶点数组中的第几个。
    - 当使用**glDrawElements**渲染时，该ID的值是正在绘制的顶点的当前的索引。
- SV_InstanceID

该ID的值是当前绘制的Instance是第几个Inseance。

看两个Unity中的API

- Graphics.DrawProcedural(Maerial mar,Bounds bounds,MeshTopology topology,int vertexCount,int InstanceCount.....)

  该API可以不需要绑定vertex和Index buffers就可以渲染，其中**vertexCount**对应了**SV_VertexID**，**InstanceCount**对应了**SV_InstanceID**。所需要的顶点数据和索引数据从Commdbuffer中获取。

- Graphics.DrawProceduralIndirect

  该API不需要指定渲染绘制调用的次数，可以直接从ComputeBuffer中读取。该API可以结合CS做剔除后再绘制。

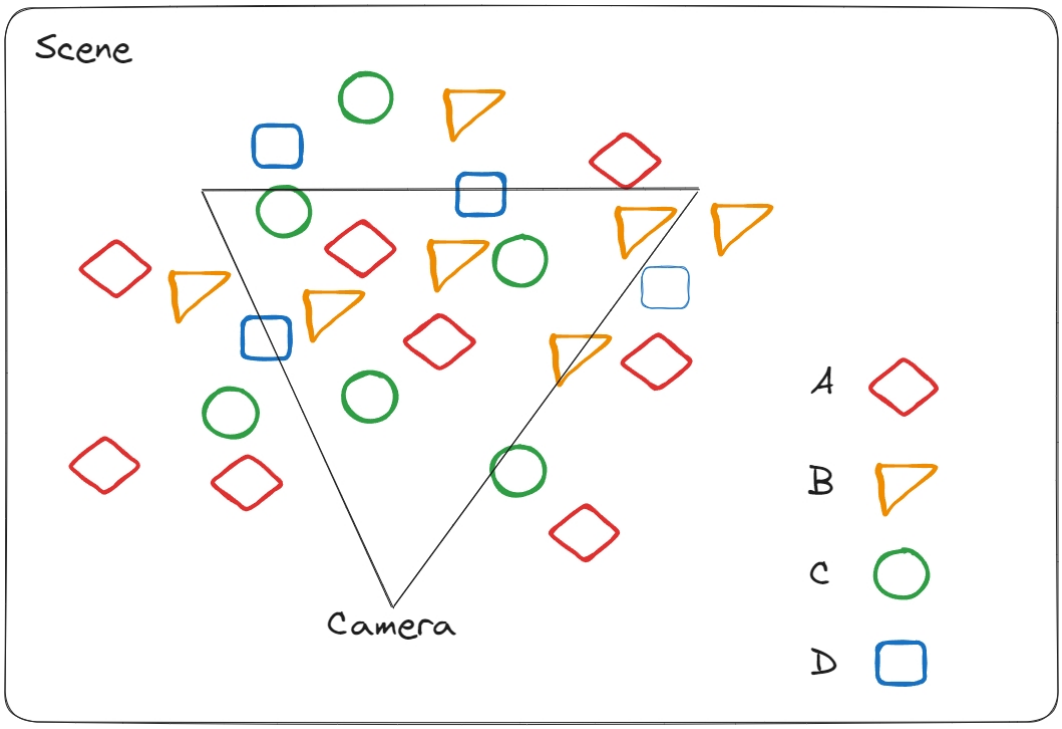
熟悉相关的参数和API后，从实际的例子出发，在不考虑GPU剔除的情况下，看如何使用Merge-Instancing完成绘制。

开始绘制

假设场景中的物体如表格所示

	Vertex	Index	Count
A	V_a	I_a	C_a
B	V_b	I_b	C_b
C	V_c	I_c	C_c
D	V_d	I_d	C_d

示例如图所示：



Scene中有A,B,C,D四种物体，它们在整个场景中有多个实例，相机看到了它们中的一部分。

1. 视锥体剔除,收集可以看见的物体
2. 将可以看见的物体，按照Mesh类型进行Vertex和Index合并(这一步可以离线做，将整个场景提前做好)
3. 只测试一个材质的情况所以我们准备了三个CommandBuffer
    1. 合并的VertexBuffer
    2. 合并的IndexBuffer
    3. 每个实例的Index在IndeBuffer中的offest
4. 使用Graphics.DrawProcedural进行渲染

> 这里忽略了一些工具性的东西。如，实际项目中，基本上美术制作的模型Index不可能相同的，如何做这一部分处理。离线处理的模型怎么做序列化和反序列化，场景很大时，怎么做显存的管理，等。