[2023.11.17]渲染流水线和DrawCall

前言

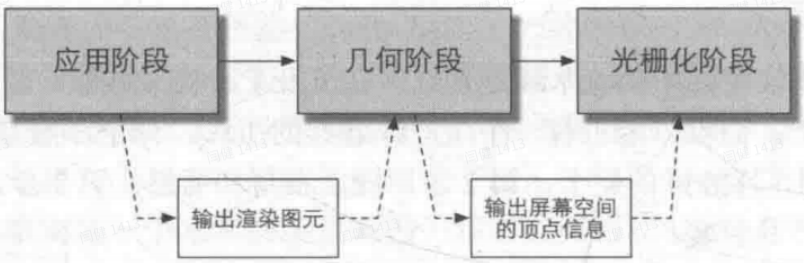
UnityShader入门精要这本书，我读了已经两遍了，第一次读过觉得懵懵懂懂，第二次也只是跟着写了一遍Shader代码，最后发现什么东西都没有剩下；

所以我决定，开始以笔记的方式记录分享自己的shader学习过程，同时也供后续回顾参考；

你应该知道什么是Shader，这点不用过多赘述；关于渲染流水线的部分，它可以被概括为，从CPU出发，将顶点、纹理等等信息传入到GPU最终绘制成图像的过程；

渲染流水线

渲染流程，一般被描述分为3个阶段：应用阶段、几何阶段、光栅化阶段；



▲图 2.2　渲染流水线中的 3 个概念阶段

1. 应用阶段
    - 应用阶段由CPU主导，这个阶段我们需要准备好，场景数据(Camera、视椎体、模型、光源信息等)；此外还可以做初步的剔除(culling)工作，把不可见的物体剔除掉，减少集合阶段的处理；最后，需要设置模型的渲染状态，包括但不限于，材质(漫反射、高光反射)、纹理、着色器等；这一阶段最终会输出渲染所需的几何信息——**渲染图元**；而后这些渲染图元会被传递给几何阶段；
2. 几何阶段
    - 几何阶段用于处理所有和我们要绘制的几何图形相关的事情，例如绘制哪些图元，如何绘制，在哪里绘制等；这一阶段由GPU主导；几何阶段可以分为几个更小的流水线来处理，其中最重要的一个任务就是把顶点坐标转换到屏幕空间，再交由光栅化阶段进行处理；
3. 光栅化阶段

- 这一阶段会使用几何阶段传递的数据来产生屏幕上的像素，并许渲染出最终的图像；这一阶段也是在GPU上运行，光栅化的任务主要是决定渲染图元中哪些像素需要被绘制，它会对几何阶段得到的逐顶点数据(纹理坐标、顶点颜色等)进行插值，再逐像素处理；同样的，光栅化阶段也可以分为几个小流水线来进行；

CPU和GPU的通信

那么我们知道，流水线的起点是CPU主导的应用阶段，它大致可以分为三个步骤；

加载数据到显存

没什么好说的，实际渲染中，包含顶点的位置/颜色信息、法线方向、纹理坐标等等复杂数据都会被上传到显存中，方便GPU知道如何渲染图元；

设置渲染状态

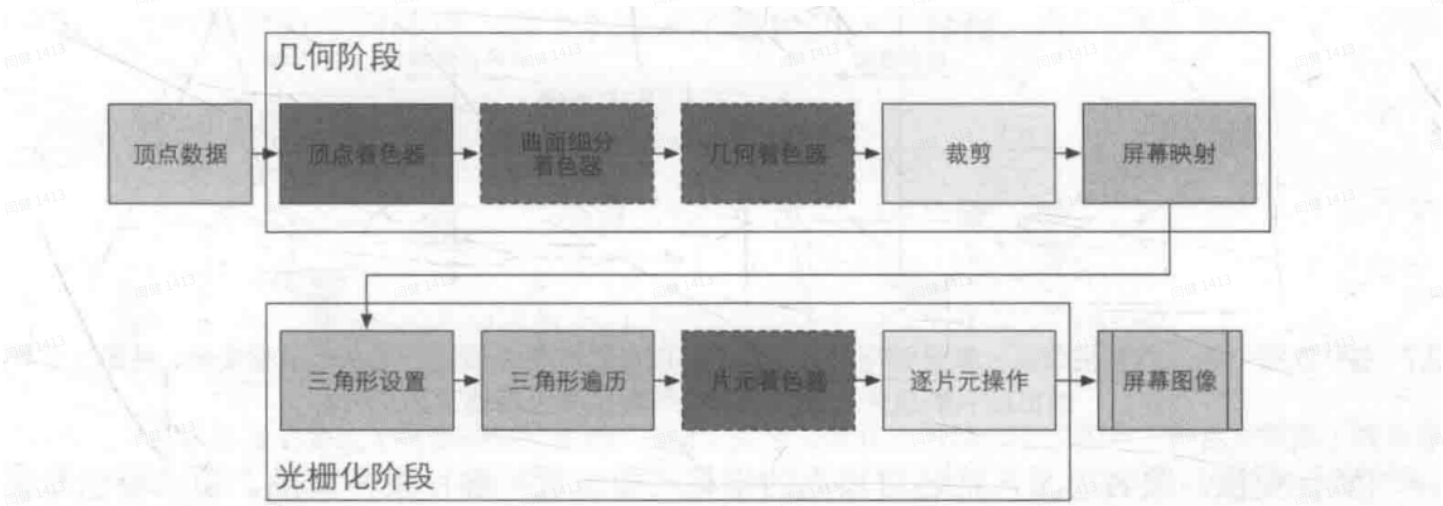
这些状态会告诉GPU渲染哪些网格、用顶点着色器(Vertex Shader)还是片元着色器(Fragment Shader)、光源属性、材质等等；

调用Draw Call

最后，在这些东西都准备好之后，CPU会发起一次DrawCall，通知GPU可以开始渲染我准备好的图元了；我们都知道要优化DrawCall，减少DrawCall，其实优化的是CPU的压力；

GPU流水线

刚刚我们说到的几何阶段和光栅化阶段，都属于GPU流水线，这两个阶段我们只能操作对我们开放了操作权限的部分内容；



可以看到，GPU流水线接受顶点数据作为输入，这些数据由CPU加载进显存，再通过DrawCall指定给GPU渲染；

几何阶段可以做如下细分：

1. 顶点着色器(Vertex Shader)是完全可编程的，它通常用于实现顶点的**空间变换、顶点着色**等功能；

2. 曲面细分着色器，可选，用于细分图元；
3. 几何着色器，可选，用于执行逐图元着色操作，或产生更多图元；
4. 裁剪(Clipping)，这一阶段的目的是将那些不在摄像机视野内的顶点、和三角图元的面片裁剪掉；
5. 屏幕映射，这一阶段由GPU完成，我们也无法干涉，它负责把每个图元的坐标转换到屏幕坐标系中；

光栅化阶段可以做如下细分：

1. 三角形设置，GPU固定的函数；
2. 三角形遍历，GPU固定的函数；
3. 片元着色器(Fragment Shader)是完全可编程的，它用于实现逐片元的着色操作；
4. 逐片元操作，负责修改颜色、深度缓冲、混合等等操作，它可以通过配置来实现具体操作；

接下来，我们针对几个阶段做更详细的了解；

顶点着色器

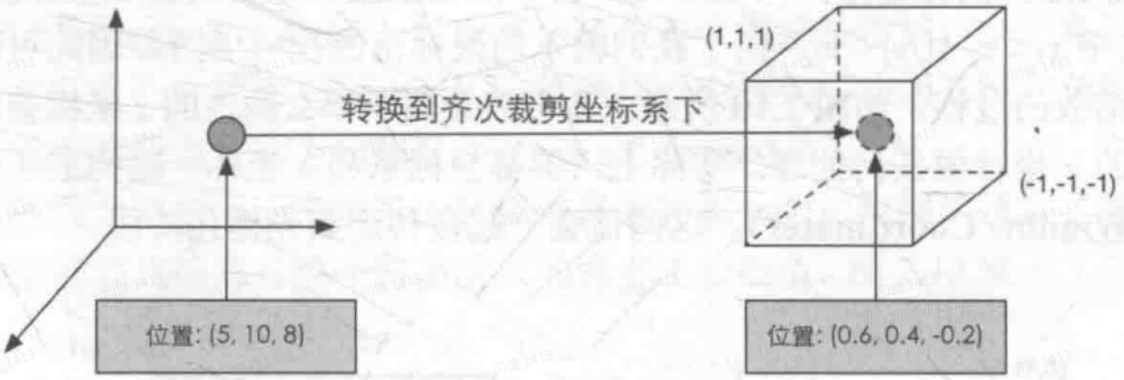
Vertex Shader是流水线的第一个阶段，它的输入来自于CPU；顶点着色器的处理单位是顶点，即输入进来的**每个顶点都会调用一次顶点着色器**；

顶点着色器本身不参与创建或销毁顶点，也无法获得和其他顶点的关系(比如是否同属于一个三角网格)，基于这种特性，GPU可以并行处理每个顶点，这表示着顶点阶段的处理速度会快到飞起；

顶点着色器的主要工作为：坐标变换、逐顶点光照；此外，顶点着色器还可以输出后续阶段所需的数据；

坐标变换，就是对顶点的坐标进行变换，顶点着色器可以在这一步中改变顶点的位置，例如可以用顶点动画来模拟水面、布料等；但无论如何，顶点坐标系最基础的一项工作是，把顶点坐标从模型空间转换到齐次裁剪空间，例如：

```
//这个操作在目前的Unity版本中已经由函数统一了；
o.pos = mul(UNITY_MVP, v.position);
```

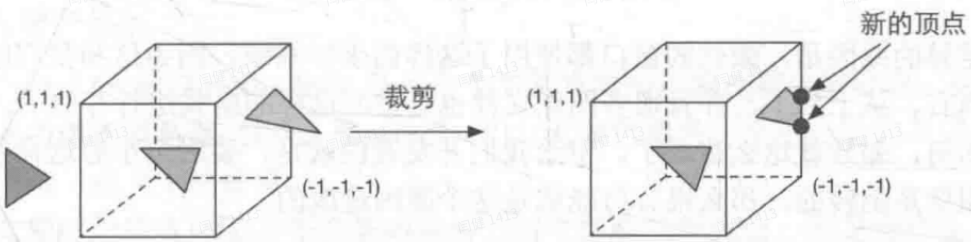


▲图 2.8　顶点着色器会将模型顶点的位置变换到齐次裁剪坐标空间下，进行输出后再由硬件做透视除法得到 NDC 下的坐标

裁剪

由于我们的场景很大，而摄像机的视锥范围肯定不会覆盖场景的所有物体，所以裁剪，就是为了剔除不在摄像机视锥范围内的物体的部分；

对于图元来说，它和摄像机的关系有3种：完全在视野内、部分在视野内、完全在视野外；所以明显能看出，只有部分在视野内需要裁剪，完全在视野外的图元不会继续向下传递，因为它不需要被渲染；

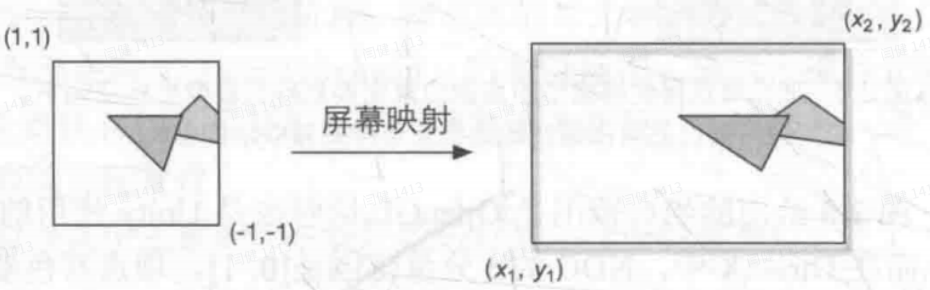


▲图 2.9　只有在单位立方体的图元才需要被继续处理。因此，完全在单位立方体外部的图元（红色三角形）被舍弃，完全在单位立方体内部的图元（绿色三角形）将被保留。和单位立方体相交的图元（黄色三角形）会被裁剪，新的顶点会被生成，原来在外部的顶点会被舍弃

这一步是不可编程的，但我们可以自定义一个裁剪操作来对这一步进行配置；

屏幕映射

屏幕映射的任务是把每个图元的x和y坐标转换到屏幕坐标系下，屏幕坐标系和z坐标共同组成了窗口坐标系，这些值会一起被传递到光栅化阶段；



▲图 2.10　屏幕映射将 x、y 坐标从（−1, 1）范围转换到屏幕坐标系中

三角形设置

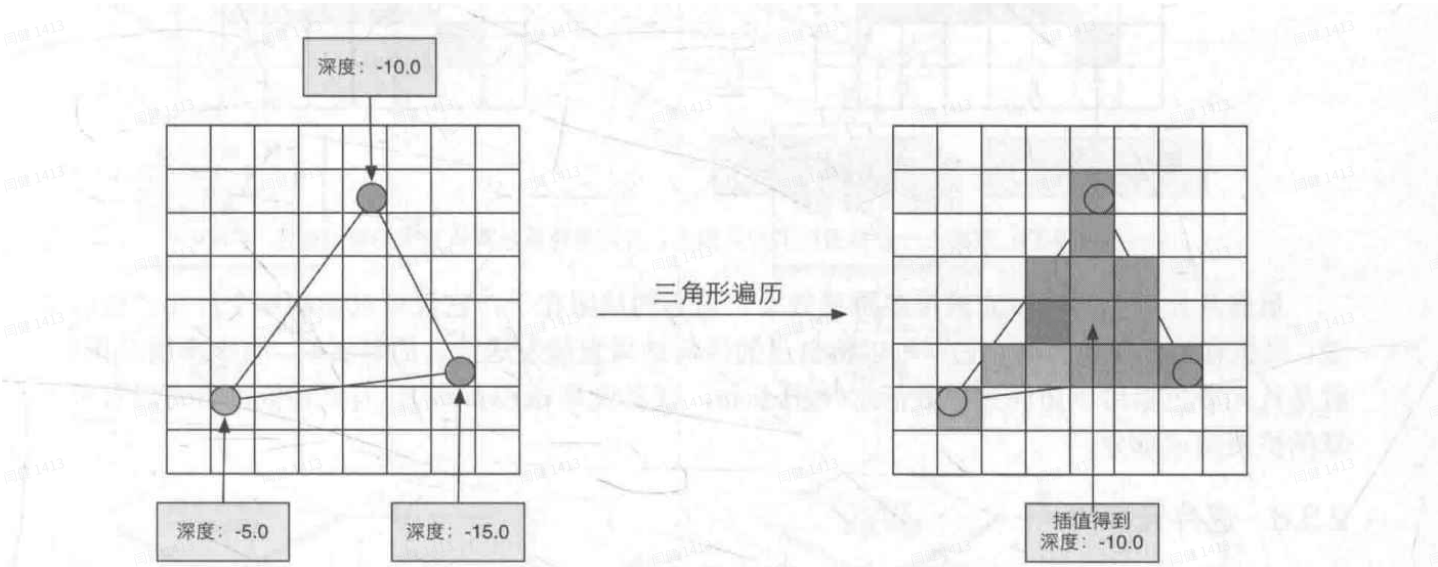
从这一步开始就进入了光栅化阶段，上一阶段输出的信息是屏幕坐标系下的顶点位置，以及和它们相关的额外信息，如深度(z坐标)、法线方向、视角方向等；

三角形设置是光栅化的第一个阶段，这个阶段会计算光栅化一个三角形网格所需的信息(个人理解光栅化就是把东西画在屏幕上的一个过程)；上一个阶段输出的是三角网格的各个顶点，但如果我们要绘制出一个三角形，还需要计算整个三角网络对限速的覆盖情况，所以要得到每条边的像素坐标；

这样一个计算三角网格表示数据的过程就是三角形设置，它的输出是为了给下一个阶段做准备；

三角形遍历

三角形遍历阶段将会检查每个像素是否被一个三角网格所覆盖，如果是就会生成一个片元；而这个阶段也被称为扫描变换；它会根据上一个阶段(三角形设置)的计算结果来判断一个三角网格覆盖了哪些像素，并使用三角形3个顶点的顶点信息对整个覆盖区域的像素进行插值；



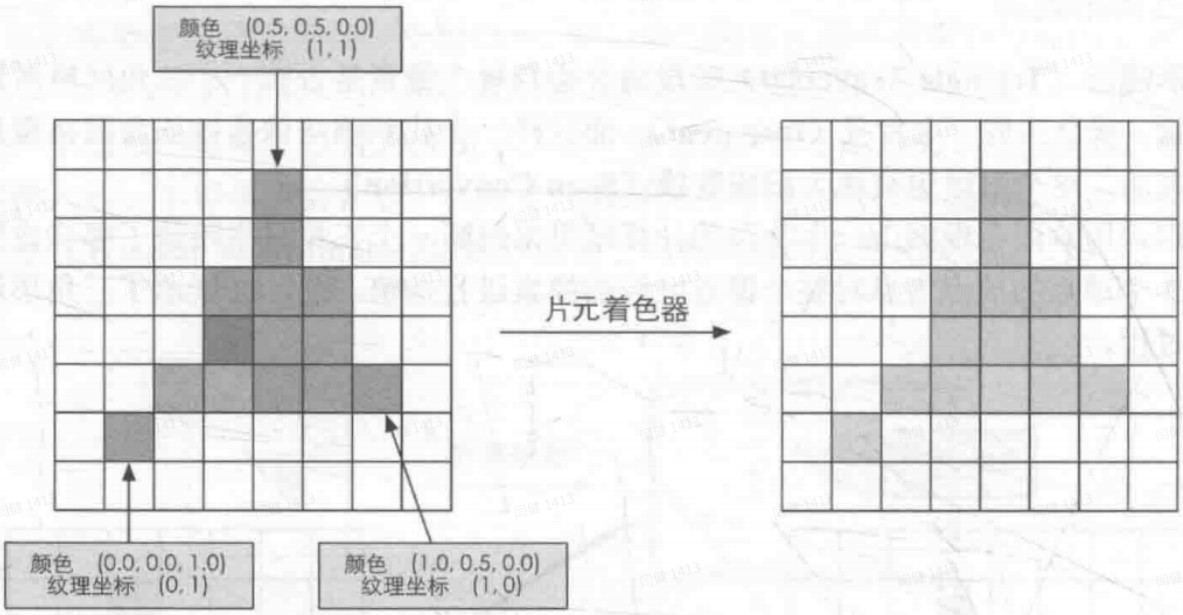
▲图 2.12　三角形遍历的过程。根据几何阶段输出的顶点信息，最终得到该三角网格覆盖的像素位置。对应像素会生成一个片元，而片元中的状态是对 3 个顶点的信息进行插值得到的。例如，对图 2.12 中 3 个顶点的深度进行插值得到其重心位置对应的片元的深度值为−10.0

这一步最终会得到一个片元序列，但片元并不是真正意义上的像素，而是包含了很多状态(包括但不限于屏幕坐标、深度信息、几何阶段输出的法线、纹理坐标等)的集合，这些状态用于计算像素的最终颜色；

片元着色器

片元着色器是另一个非常重要的可编程着色器阶段；前面的光栅化阶段不会影响像素的颜色值，只是产生信息描述三角网格的像素覆盖情况；

片元着色器的输入是上一个阶段对顶点信息插值得到的结果，具体来说，是从顶点着色器中输出的数据插值得到的，而片元着色器的输出是一个或者读个颜色值；



▲图 2.13 根据上一步插值后的片元信息，片元着色器计算该片元的输出颜色

比如纹理采样就在这一阶段完成，片元着色器之前的阶段已经为纹理采样准备好了数据：

1. 顶点着色器输出顶点的纹理坐标；
2. 光栅化阶段对顶点的纹理坐标进行插值，获得覆盖的片元的纹理坐标；
3. 最后，片元着色器计算并输出片元对应的颜色；

片元着色器和顶点着色器是类似的，它仅可以影响单个片元，也不可以将自身的结果传输给相邻的或其他的片元(导数信息除外)；

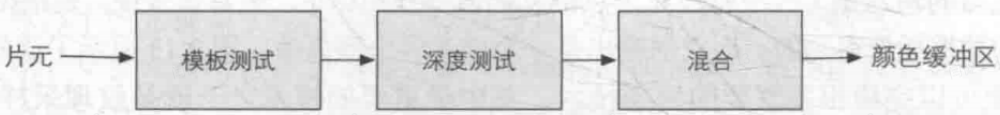
逐片元操作

逐片元操作是渲染流水线的最后一步；

逐片元操作是OpenGL中的说法，它直观地表达了这个阶段的操作单位和方式——逐片元；而在DirectX中，这一阶段被称为输出合并阶段，而这个名字则直观表达了这个步骤的目的——合并；

这一阶段有几个主要任务：

1. 决定每个片元的可见性，这就涉及了许多测试工作，如深度测试、模板测试等；
2. 如果一个片元通过了所有的测试，就需要把这个片元的颜色值和已经存储在颜色缓冲区中的颜色进行合并，或者说是混合(blend)；对于不透明的物体，可以关闭混合，直接覆盖掉缓冲区中的像素值，对于半透明物体，我们就需要通过混合来让这个物体看起来是透明的；



▲图 2.14 逐片元操作阶段所做的操作。只有通过了所有的测试后，新生成的片元才能和颜色缓冲区中已经存在的像素颜色进行混合，最后再写入颜色缓冲区中

关于Draw Call的说明

Draw Call是什么

DrawCall本身的含义很简单，就是CPU调用图像编程接口，如OpenGL中的**glDrawElements**命令(查看过UnityProfiler的GPU部分，应该能频繁看到这个函数)，通过调用函数来命令GPU进行渲染的操作就是DrawCall；

DrawCall过多会造成CPU的压力，为什么?

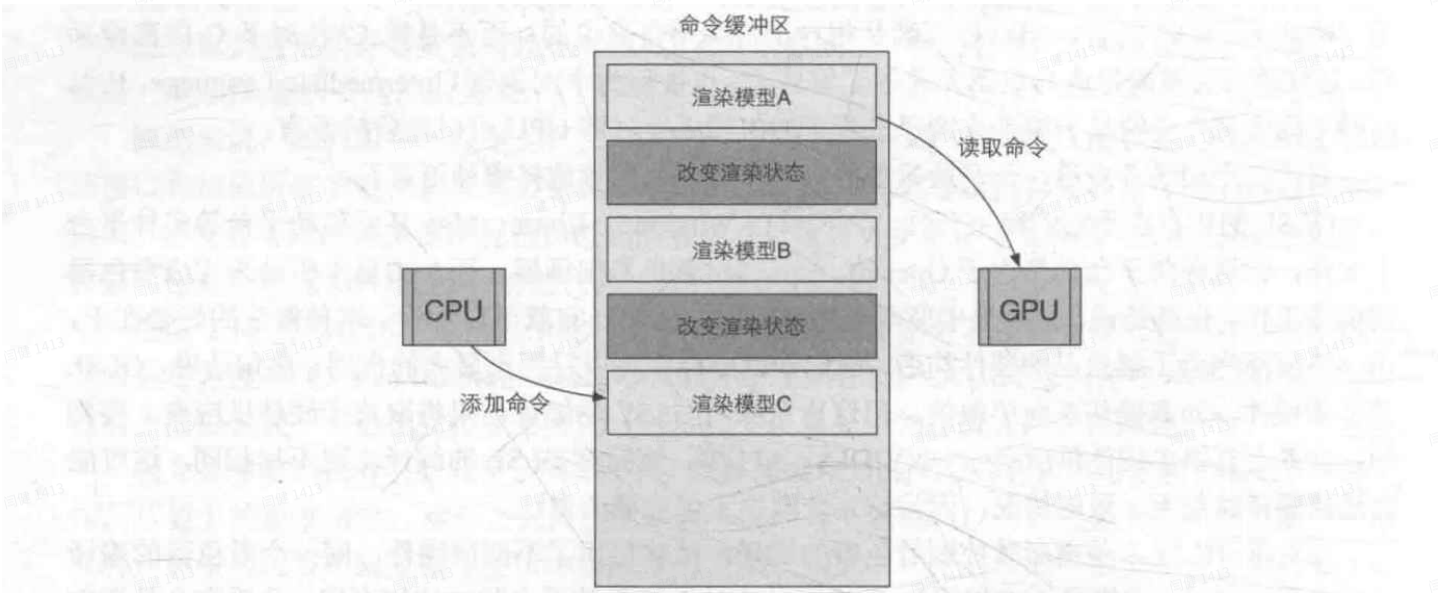
Command Buffer是什么

你应该知道CPU和GPU是并行工作的，因为如果不这样，CPU在发送渲染命令后，要等待GPU完成上一个渲染任务，才能发送下一个命令，这显然会浪费大量的时间在等待上，是低效的；

所以，CPU和GPU的并行工作，依靠**命令缓冲区(Command Buffer)**来实现；

命令缓冲区包含了一个命令队列，由CPU向其中添加命令，GPU从中读取命令；添加和读取的过程是互相独立的，所以二者可以互相独立；

当CPU需要渲染对象时，它就向命令缓冲区添加命令，GPU完成了渲染任务后，它就从队列中读取下一个命令执行任务；命令缓冲区中的命令有很多种类，DrawCall只是其中一种；



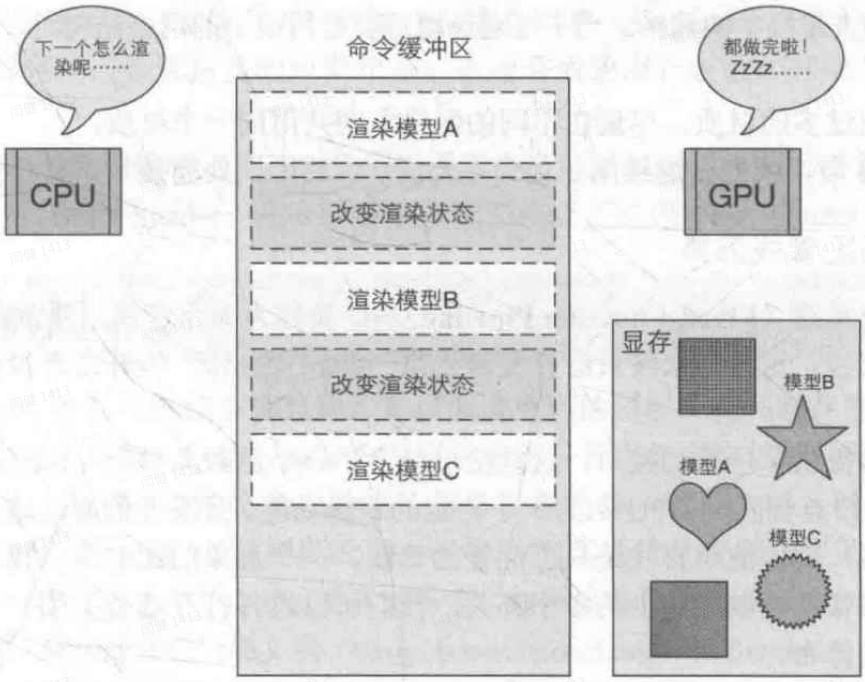
▲图 2.19　命令缓冲区。CPU 通过图像编程接口向命令缓冲区中添加命令，而 GPU 从中读取命令并执行。黄色方框内的命令就是 Draw Call，而红色方框内的命令用于改变渲染状态。我们使用红色方框来表示改变渲染状态的命令，是因为这些命令往往更加耗时

Draw Call如何影响帧率

你应该知道移动1GB的文件所需的时间远少于移动100w个1KB的文件；

渲染的过程虽然和文件操作天差地别，但从逻辑角度上，是类似的；在每次调用DrawCall前，CPU要向GPU发送很多内容，包括数据、状态和命令等；在这一阶段，CPU需要完成很多工作，例如检查渲染状态等；

而一旦CPU完成了这些准备工作，GPU就可以开始本次渲染；GPU的处理速度是很快的，渲染200个还是2w个三角网格的耗时差异(可能)是纳秒级的；因此，如果DrawCall的数量过多，CPU就会把大量时间花费在提交DrawCall上，造成CPU的过载；

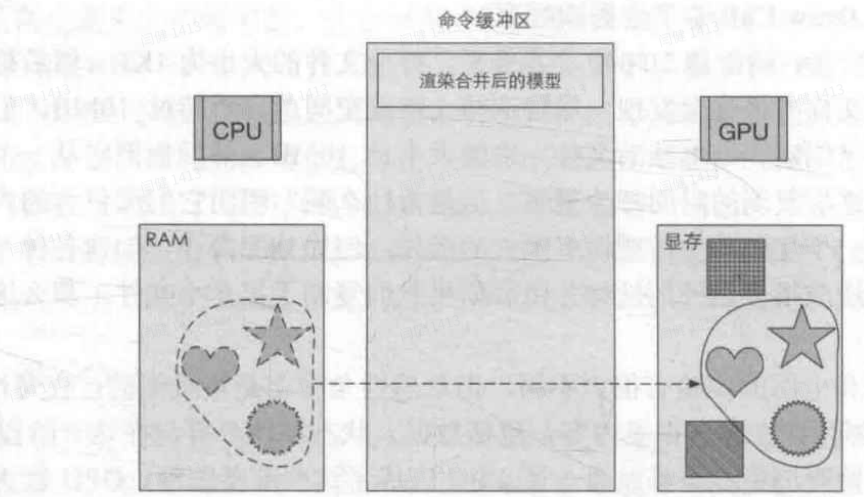


▲图 2.20　命令缓冲区中的虚线方框表示 GPU 已经完成的命令。此时，命令缓冲区中没有可以执行的命令了，GPU 处于空闲状态，而 CPU 还没有准备好下一个渲染命令

Draw Call怎么减少

减少DrawCall的方式有很多，但被大家熟知的方式之一就是**批处理**；你应该知道Unity本身就有动态批处理和静态批处理两种最基本的方式，当前要合批也是有限制的，这些要求无论是Unity文档还是网上都有其他人在说明，无须赘述；

批处理的思想就是把很多小的DrawCall合并为一个大DrawCall，减少CPU包装DrawCall本身的消耗；



▲图 2.21　利用批处理，CPU 在 RAM 把多个网格合并成一个更大的网格，再发送给 GPU，然后在一个 Draw Call 中渲染它们。但要注意的是，使用批处理合并的网格将会使用同一种渲染状态。也就是说，如果网格之间需要使用不同的渲染状态，那么就无法使用批处理技术

但是，由于我们需要合并网格，而合并也会消耗时间，所以批处理技术在静态物体上的优势会更大，比如大地，石头等；

对于动态批处理，由于动态物体的位置每帧都在发生变化，所以网格每一帧都需要重新合并，那你可能要斟酌一下是DrawCall带来的消耗高，还是动态批处理本身的消耗高了；

同时，因为需要在CPU的内存中进行网格合并，(对于静态批处理来说)为了避免每帧都合并不会改变的静态物体，就会存储已经合并的物体的网格，这就会占据额外的内存；所以静态批处理不适合大量重复出现的物体，比如树和草这些；

所以，针对树和草的问题，你还可以使用GPU Instancing技术，如果你想了解可以参阅：[11.03]Unity GPU Instancing调研 ，如果需要权限，申请就好了，Don't be shy!

此外，GPU Instancing和动态批处理是冲突的，你不能将两者同时使用；

综上，对于DrawCall的优化，有两点通用的建议：

1. 避免大量使用很小的网格，如果一定要这么做的话，考虑是否可以合并；
2. 避免使用过多的材质，尽量在不同的网格之间共用同一个材质，方便合批；

这就够了吗

然而，DrawCall只是减少了CPU的压力，如果GPU侧没出现问题，那还无伤大雅；

游戏行业发展至今，有的游戏也面临着不小的渲染压力，手机发烫、功耗高，有相当一部分原因就是带宽占用过高导致的；而带宽占用高又有可能是更复杂的原因导致的，也有可能是Shader过于复杂或材质球过多等等原因导致的SetPassCall过多；

所以渲染的优化，也有许多不同纬度的问题导致的，不能单一地去看；