本文欢迎转载，但是请注明出处：<https://github.com/lygyue/Books>

# 网格模型

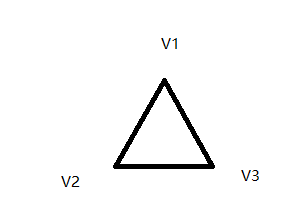
回到图形的基础原理，我依稀记得我提过一条结论：

任何曲面，都可以由N个三角形构成！

这点理解了，那么网格模型，就变成了三角形数据的记录以及优化。

**最简单的顶点优化方式：顶点坐标/索引坐标。**

先假设顶点只有空间坐标，而不考虑UV，法线，切线，权重等等一大堆东东。



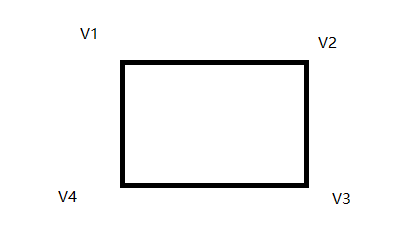
一个三角形，三个顶点坐标。每个顶点坐标为：x，y，z。浮点数。那么，记录一个三角形三个顶点所需要的内存是：

3 \* 3 \* 4 = 36byte

不难算吧？一个顶点三个float，三个顶点不就是9个float吗？每个float4个字节。所以我必须再强调一次基础的重要性。基础不会，后续就是看天书！

一个模型假设有10000个三角形，那么，不考虑其他信息，仅仅是顶点信息，就需要360k内存。叔可忍，婶不可忍。

我们来看看顶点/索引的优化方式：



此处有一个矩形，那么这个矩形可以分割成两个三角形。假设按照之前的存储，那么需要72个字节。

我们来尝试一下最简单的索引压缩大法，不熟悉这个的回头去看图像压缩算法里面的调色板，其实完全是一个套路。我先记录四个顶点的坐标，那么内存占用是3 \* 4 \* 4 = 48字节。

我再记录6个索引，来描述两个三角形。例如上图，我用V1，V2，V3描述一个三角形，V3，V4，V1描述另外一个三角形，那么我只需要记录1,2,3/ 3, 4 ,1就可以描述两个三角形。此处，因为一般模型不会太大，我用16位来描述这个索引。那么一个索引是2个字节。我6个索引，那么用了6 \* 2 = 12个字节

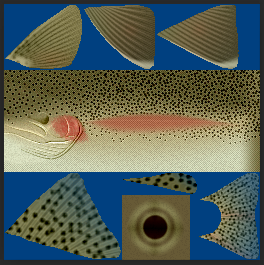
48 + 12 = 60 《 72

由于一个顶点一般不会就一个坐标那么简单，还会有UV坐标，还会有顶点颜色，还会有法线等等，因此，这个压缩比其实还挺高的。

大佬们，是否觉得自己掌握了世界的真理，是不是觉得自己全身充满了力量？

我以前也是这么认为的，然后在几年之后，受到了当头棒喝。

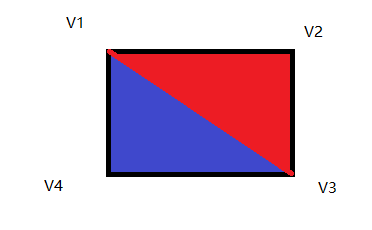
先来了解一下什么是UV坐标。如果完全了解了的，可以忽略：



看上图，我们有了三角形，那么任何一个屏幕上的像素，都是需要图片采样的，这个采样的依据，就是UV坐标。

连个UV坐标都看不懂的话，后面压根没法搞了，这里我不打算讲细节。想看细节的随便找找资料，网上大把。

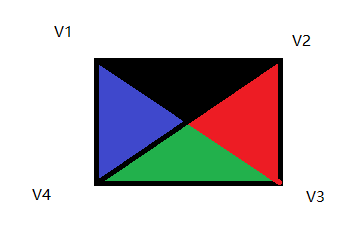
我们看看有什么问题，看图：



还是上图，但是增加了UV坐标，顶点颜色。

假设我们还是用顶点坐标，索引坐标的套路，那么，V1、V3的顶点颜色该是红色还是蓝色？UV坐标到底应该是上三角形的UV坐标还是下三角形的UV坐标？

我知道你想说什么，你想说我多加几套UV坐标对吗？



这种情况，中间的点，你打算做四套UV坐标吗？

所以，新入行的菜逼，看到顶点坐标，索引坐标，以为掌握了模型的真谛。现实却完全不是这样的。

现实是，上面那个矩形，还是6个顶点。这个索引坐标技术，早期的时候确实有用，但是到了后期，很多时候就没什么用了。你想象一下，N个三角形，每个三角形假如要做实时光照计算，每个三角形都要记录自己的法线坐标。重用顶点就很难了。

有大佬有疑问：我不是可以用法线贴图吗？这么说的大佬，好歹知道得更多一些。法线贴图是一个比较有技术含量的东东，后续有单独的章节细讲。这里，先做一个总结：

1. 顶点坐标/索引坐标的套路，并非万能，绝对没有一统江湖。
2. Autodesk是一家伟大的公司，他们给全世界解决了很多问题。

好了，顶点坐标和索引坐标的问题打住。大概介绍一下信息的构成。

一般而言，顶点会包含以下信息：

顶点坐标（必须）；

UV坐标（绝大部分都有）；

顶点颜色，早期很有用，现在一般没什么用，反正我很少用到；

**法线：**（需要计算光照，需要做一些特效的时候必须有，例如描边）；

**切线：**（用到法线贴图的时候必须有，其他时候就。。。）；

其他一些用到的例如权重，还有自定义的一些东东。

写几次shader，基本上就熟悉这些套路了。

这部分也没太多可以介绍的，就法线部分值得深入介绍一下，后续会在其他地方介绍到。

# 渲染批次（Draw Call）

一次渲染过程大概是这样子的：



这里，大概是渲染一个Mesh的代码，可能多一些，可能少一些。

渲染一个Mesh，统称为一次DrawCall。也称为一次渲染调用。这部分，有人说是同步的，有人说是异步的。这些，先不谈，以后章节有机会我再谈。

很容易看出来。一个for循环，渲染的mesh越多，大概率帧率越低，注意，我只说大概率，没说一定。

假设现在要渲染10000个三角形，怎么渲染是最快的？当然是一次渲染10000个三角形。如果你渲染100次，一次渲染100个三角形，理论上肯定效率更低。这是由GPU的机制决定的，不理解的回头去看“认识显卡”章节。

那么，怎么降低渲染的Mesh数量？方案很多，例如什么最常见的镜头裁切就是一个，还有什么其他一堆方案，intel多年前我记得貌似还出过一个什么鬼软光栅AABB裁切的。这些，这里都不谈，我们只谈Mesh相关的。

可以预见的一套方案是：我把场景中比较小的Mesh，合并在一起。你想啊，假如我房间里有一把尺子，有一个橡皮擦，有五本书。正常的做法，一样一个mesh。但是现在这些东西都是静止不动的， 我全部放在一个mesh里面不行吗？

理论上可行，现实中一定程度也可行，但是，有前提条件。

假设你的尺子是塑料的，是半透明的。而你的书本是不透明的，你这么干，就不能合并了。

显然，我们轻易知道，假设渲染状态不一样的，不能合并！

渲染状态是指：光照函数，带不带透明，模板缓冲……反正就是上图中的各种设置，要一样的。

这个，就是简单的合并渲染批次优化。一般3d Max可以做，一些也可以自己程序做，还有一些编辑器可以做，例如UE4。原理都是一样的。

但是，合并渲染批次有那么一些需要注意的地方：

例如你的尺子在房间的一头，而书本在房间的另外一头，那么当你只看到尺子的时候，书本也是要被渲染的，因为你们两个已经被合为一体，要么都渲染，要么都不渲染。

**因此，距离比较近，渲染状态一致，不需要分开的Mesh，才适宜合并渲染批次。**

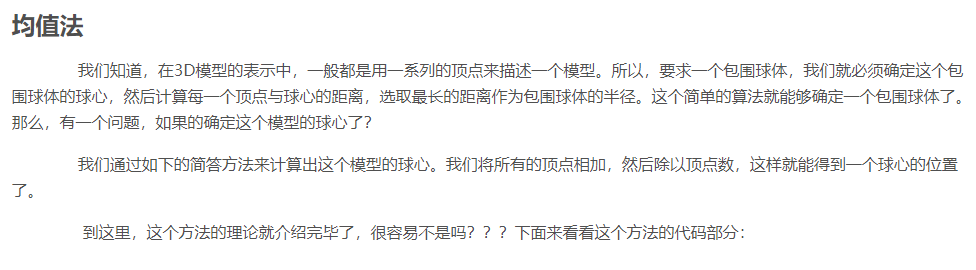
# 包围球/包围盒

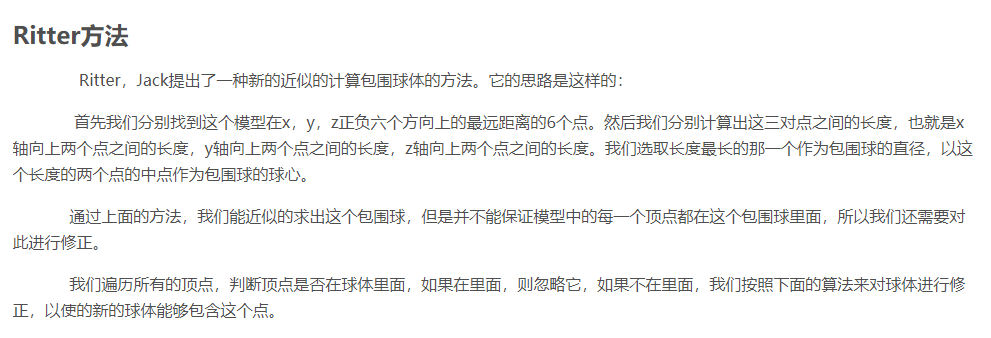
估计大部分人，只要搞3d一段时间，应该都听过这两个东东。

本来这种比较简单的烂大街的东东，我是不大打算讲的，无奈看到网上垃圾太多，我觉得我必要正本清源，是时候表演真正的技术了！

包围球，绝大部分人都是这样理解的：求一个最小的球，这个球能把所有的mesh顶点都包含进来。

那么这个球，怎么算出来呢？各位大佬能混到一定地步，这点数学基础还是有的。百度随手搜了一下，看到了几种流传甚广的算法，这些算法不知道谁抄谁的，也没见到转载字眼，连错别字抄的都是一样的：现截图如下：





随便搜到了两种算法，并且附带源码，还附带有算出来的效果图，牛逼不？我要做引擎了，我要把这两种算法放进我的引擎里，谁也不要拦我！

等你真把这两种算法放进去，大概率你会发现你傻逼了。所以说，做学问，最怕就是一知半解，似是而非。

使用这样的算法，至少有两个问题：

第一个问题：这两个算法的开销太大。如果启动的时候加载1000个MESH，算一千个包围球，需要各种算中心点，算半径，遍历多次。而一个游戏场景，1000个MESH一点也不多。要是GIS之类的应用，就更多了。一万个都不算多。

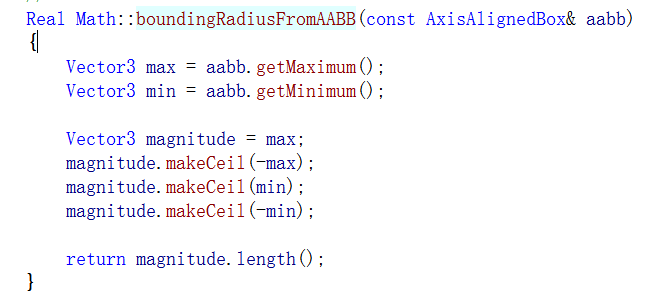
第二个问题，MESH在空间之中，旋转是很常见的需求。静态的MESH你这样算出来的包围球没问题，但是一旦这个MESH旋转了，算出来的结果，很可能会有问题，需要重新算。试问，你旋转一次算一次，如果你这个包围球还参与镜头剔除之类的计算，那么是每一帧都需要计算一次，这个开销能承受吗？

因此，我不说这些算法没有价值，而是说，这类算法，大部分只是理想状态，写个demo，发个paper之类的可以。真正做引擎，要考虑到的东西太多。

好了，现在讲一讲一般引擎中这类东西的套路做法。

首先，我们来算一个AABB。这个怎么算呢？加载的时候，遍历所有顶点，算一个最大值，一个最小值。这里，只需要遍历一遍，做一个verctor3的最大值最小值比较即可，效率还是很快的。

算出了最大值，最小值，跟包围球有什么关系吗？



这是OGRE的算法，根据最大值，最小值算包围球的半径。这算法最大的好处是简单暴力。但是问题也是显而易见。

首先，如果建模不规范，这个算法会坑死。显而易见。只要稍稍玩过一些3D引擎，会很容易发现，一旦物体旋转，包围盒跟着变化大小，有时候包围盒会非常大，造成极大的误差。这是因为，不可能每个MESH每一帧重新算一遍AABB，AABB在旋转的时候，其实也是跟着旋转的。想象一下，一个Vector3，做一个简单的空间变换，很可能会变大很多或者变小很多，直接导致AABB包围盒的大变更。一旦建模不规范，例如模型的中心点不对，这个影响就大到你怀疑人生了。完全可能出现那种：你希望一个中心点+半径描述一个包围球，现实是半径比实际的直径还要大，造成了包围球的巨大浪费，那不是得哭死。

所以，各位做程序的大佬，做美术规范的时候，必然要规定建模的时候，模型的中心点尽量就在模型的中心。这是基础的规范。

好了，这里顺便讲一些烂大街的东西。**AABB有什么用？**

**一般来说，做镜头裁切，射线检测的时候会用到。由于是烂大街的东西，我不打算讲太多，打字也是很累的。**

**镜头裁切的意思是：渲染之前，先用镜头跟模型的AABB做一个相交计算，如果相交，再渲染，不然不渲染。镜头怎么跟AABB做相交？我靠，镜头是一个正梯形啊，就是算两个几何体相交啊。这是数学问题，**千万别觉得3D里的镜头是现实中相机拍照的镜头，那么想的人搞图形学注定是要没有前途的，你这种形象思维方式，该去搞艺术，例如人体艺术。及早转行才是王道。

**射线检测是指：屏幕上点击选取一个物体，先算一条屏幕射线，然后用射线跟AABB相交，算出来一堆MESH，再排序，然后再遍历MESH的三角形，做三角形跟射线的相交，判断点中了哪一个MESH。**

# 网格模型的LOD（Level of Detail）

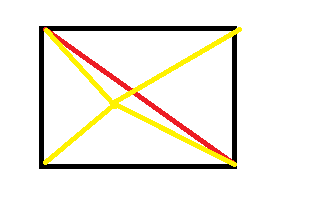
这东西有什么用？常见的几种用法：

1. 一个大型游戏，模型很精美，面数很多。但是你好歹要照顾一下低端玩家。那么，你可以高端玩家用高模，低端玩家用低模也能玩。
2. 你显卡很牛逼，但是一个MESH距离你很遥远的时候，其实细节你是看不清楚的。这个时候，你就不需要浪费了，就渲染粗模即可。

这个东西是怎么实现的？最简单粗暴的做法，是让美术建几套模型，渲染的时候，判断模型跟镜头的距离，距离近的，渲高模，远的，渲低模。当然了，这种做法，大概率是会被鄙视的，因为没什么技术含量，而且切换的时候比较生硬，但是，我其实见过更多更糟糕的效果。例如早期的游戏由于镜头远平面比较近，走近了突然出来个NPC，吓死人：）。

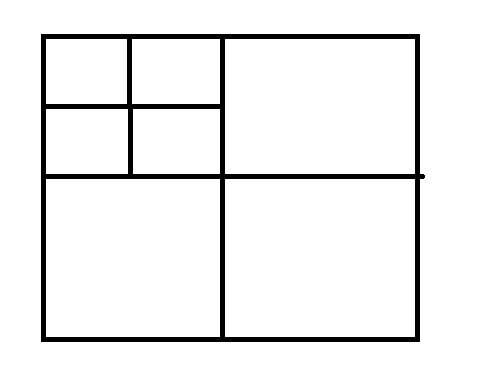
好一点的做法，会有技术含量一点。其实就是**公用顶点和贴图，只切换索引坐标。**如果不熟悉索引坐标，顶点坐标的，需要回去复习一下。

我随便画个图示意一下：



这里，是五个顶点，四个角还有中间那个黄色那个。精模渲染，那么就是四个三角形。粗模呢，简单啊，红线一条线，顶点坐标不变，但是索引变了，变成了两个三角形。

大佬们看出来了，地形四叉树不是差不多的套路吗？你没有看错，地形四叉树就是这种套路，看图：



国内的游戏，做LOD的我知道的很少，估计只有部分大厂做的。小厂都是小场景，能出来效果再说，别扯那些有的没的优化。而地形这个四叉树LOD，基本就是标配了，一般小厂也能做。为毛呢？因为这个优化都有公开算法的，而且地形本来就比较标准，实在不行用引擎，引擎自带。

为什么除了地形之外的LOD很少？很少就是很少啊，哪里有什么原因。我从来不说美术不行，这是你们说的。程序也不行？去死吧，我会这么自绝于群众吗？当然不会。

至于这几年大火的Tessellation，其实是高级版的LOD，这涉及到一个全新的建模方式：贝塞尔曲面建模。这个就更有技术含量了，我后续有专门的章节介绍。

好了，关于网格模型的，暂时没有想到有什么要讲的。先这样吧。