渲染优化—By 星云大佬

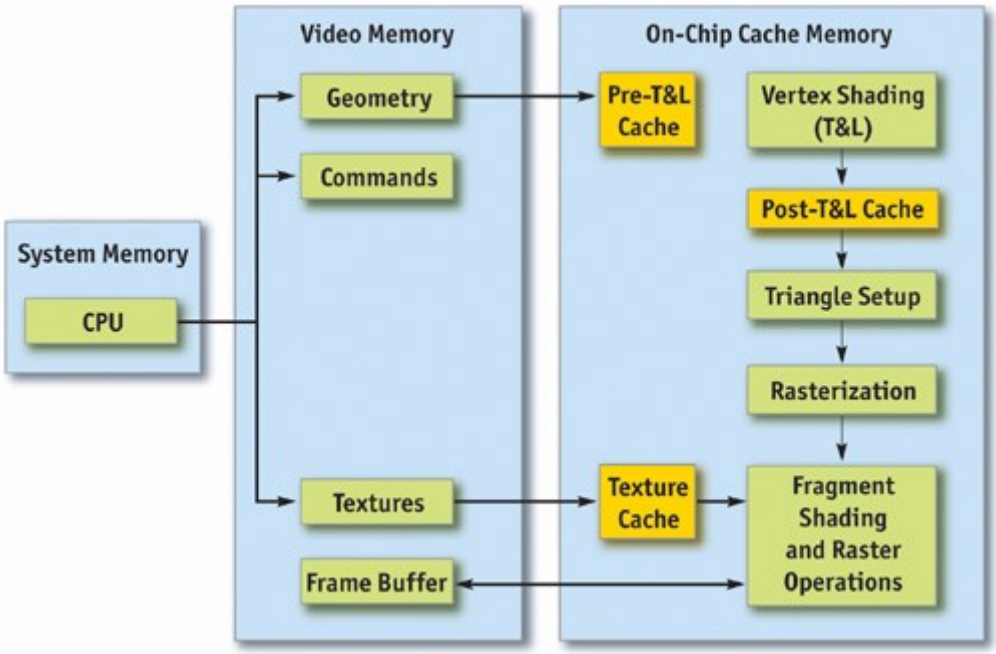
前言

优化时，牢记三句格言、

- “KNOW YOUR ARCHITECTURE”
- "Measure"
- "Premature optimization is the root of all evil"

一、渲染管线的构成

- 应用程序阶段CPU，负责场景管理，渲染排序，粗粒度剔除，设置渲染状态，提交Dc。
- 几何阶段，负责顶点计算。
- 光栅化阶段，对三角形进行光栅化处理，PS计算



基于这样的架构，其中任意的一个阶段，和他们之间的通信中最慢的部分，都会成为性能上的瓶颈。

如：

- 在应用程序阶段，CPU的剔除，排序，会导致性能瓶颈。顶点数太多，或者DC太多，会造成通讯的瓶颈。
- 几何阶段，顶点数太多会导致，顶点计算瓶颈，vertex feath load顶点时，导致miss也会出现瓶颈
- 光栅化阶段，需要光栅化的三角形太多，会出现阻塞，像素计算复杂会有PS瓶颈

以上只是概论，实际上的渲染管线更加复杂，GPU端的计算调度策略可以具体看现代GPU架构

二 、渲染管线的优化概览

优化过程可以归纳为以下基本的确认和优化的循环

- Step 1. 定位渲染瓶颈。对于管线的每个阶段，改变它的负载计算能力（即时钟周期速度）。如果性能发生了改变，即表示发现了一个瓶颈

- Step 2. 进行优化。指定发生瓶颈的阶段，减少这个阶段的负载，直到性能不再改善，或者达到所需要的性能水平。
- Step 3. 重复。重复1步和二步，直到达到所需要的性能水平。

需要注意的是，在经过一次优化步骤后，瓶颈位置可能依然在优化前的位置，也可能不在。比较好的想法是，尽可能对瓶颈阶段进行优化，保证瓶颈位置能够转移到另外一个阶段。在这个阶段再次成为瓶颈之前，必须对其他阶段进行优化处理，这也是为什么不能在一个阶段上进行过多优化的原因。

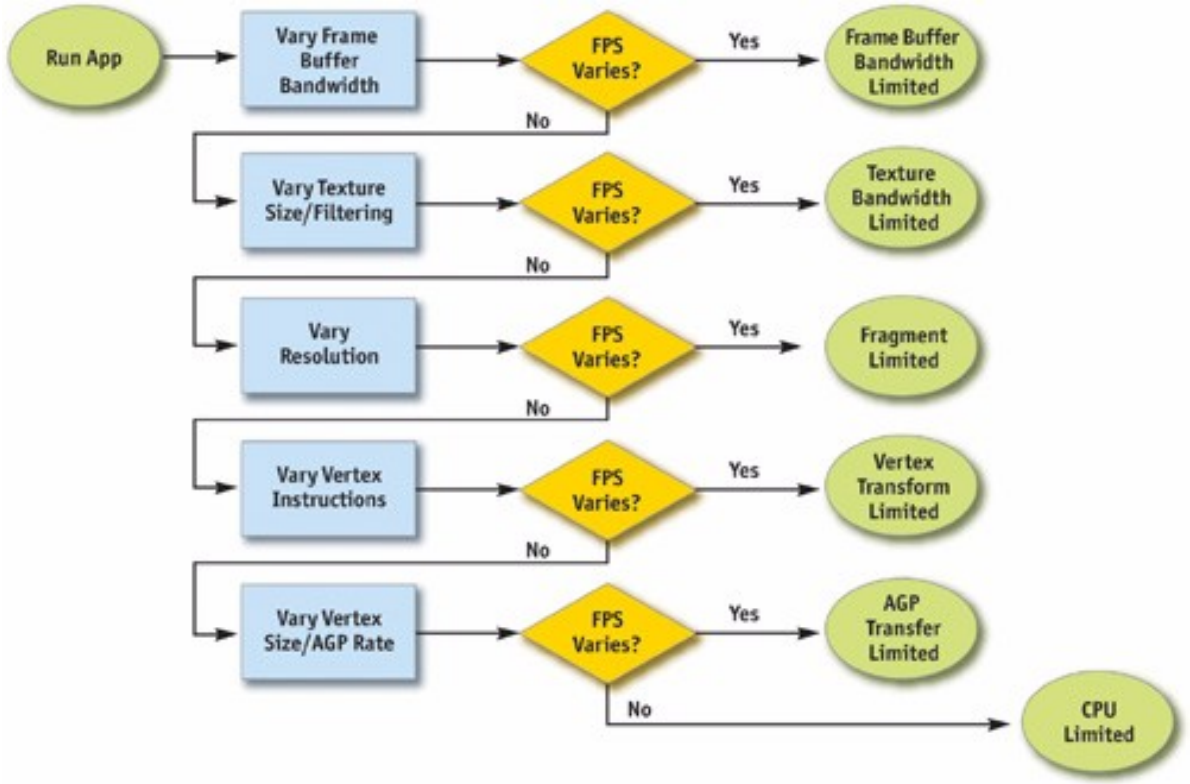
同一帧画面中，瓶颈位置也有可能改变。由于某个时候要渲染很多细小的三角形，这个时候，几何阶段就可能是瓶颈；在画面后期，由于要覆盖屏幕的大部分三角形单元进行渲染，因此这时光栅阶段就可能成为瓶颈。因此，凡涉及渲染瓶颈问题，即是指画面中花费时间最多的阶段。

在使用管线结构的时候应该意识到，如果不能对最慢的阶段进行进一步优化，就要使其他阶段与最慢阶段的工作负载尽可能一样多（也就是既然都要等瓶颈阶段，不妨给其他阶段分配更多任务来改善最终的表现，反正是要等）。由于没有改变最慢阶段的速度，因此这样做并没有改变最终的整个性能。例如，假定应用程序阶段成为瓶颈，需要花费50ms，而其他阶段仅需要花费25ms。这意味着，在不改变管线渲染速度(50ms，即每秒20帧)的情况下，几何阶段和光栅化阶段可以在50ms内完成各自任务。这时，可以使用一个更高级的光照模型或者使用阴影和反射来提高真实感（在不增加应用程序阶段工作负载的前提下）。

三、渲染管线瓶颈定位策略

两个办法

- Profiler工具。查看API的耗时信息
- 控制变量法。控制一个阶段的数据不变，增加或者减少另一个阶段的工作量，查看帧率。



管线的每个阶段都依赖于GPU频率（分为GPU Core Clock，GPU核心频率，以及GPU Memory Lock，GPU显存频率），这个信息可以配合工具 PowerStrip（EnTech Taiwan 2003），减小相关的时钟速度，并在应用中观察性能的变化。

3.1 光栅化阶段的瓶颈定位

3.1.1 光栅化操作的瓶颈定位

光栅操作的瓶颈主要与帧缓冲带宽有关。光栅化操作可能涉及到深度缓冲，模板缓冲的比较，颜色读写。这些操作都依赖与ROP，来与显存进行交互。但是移动端在Tile上完成这些操作，会快很多。总之这一块的消耗主要是在带宽上，除了修改缓冲的位数如从32到16位，还可以修改GPU的时钟周期来判断是否是瓶颈。

3.1.2 纹理读写的带宽问题

使用mipmap来确定纹理读取带宽是否是性能瓶颈。同样也可以修改GPU频率来测试

3.1.3 片元着色的瓶颈定位

改变分辨率是确定片元着色是否为瓶颈的第一步。因为在上述光栅化操作步骤中，已经通过改换不同的深度缓冲位，排除了帧缓冲区带宽是瓶颈的可能性。所以，如果调整分辨率使得性能改变，片元着色就可能是瓶颈所在。而辅助的鉴别方法可以是修改片元长度，看这样是否会影响性能。但是要注意，不要添加可以被一些"聪明"的设备驱动轻松优化的指令。

片元着色的速度与GPU核心频率有关。

3.2 几何阶段的瓶颈定位

在几何阶段有两个主要区域可能出现瓶颈：顶点与索引传输(Vertex and
Index Transfer)和顶点变换阶段（Vertex Transformation Stage）。要看瓶颈是否是由于顶点数据传输的原因，可以增加顶点格式的大小。这可以通过每个顶点发送几个额外的纹理坐标来实现，例如。如果性能下降，这个部分就是瓶颈。
但是在移动端要注意，现代移动端基于TBR架构，所有顶点处理完才会进入光栅化和Ps阶段，如果顶点数量过多，片上内存放不下就会转移到片下，导致内存访问周期变长，成为性能瓶颈。

3.2.1 顶点与索引传输的瓶颈定位

GPU渲染第一步是获取顶点数据，这时会从系统内存通过AGP或者PCI Express总线传送到GPU。
可以通过调整顶点格式的大小，来确定得到顶点或索引传输是否是应用程序的瓶颈。
如果数据放在系统内存内，得到顶点或索引的性能与AGP或PCI Express总线传输速率有关；如果数据位于局部缓冲内存，则与内存频率有关。

如果上述测试对性能都没有明显影响，那么顶点与索引传输阶段的瓶颈也可能位于CPU上。我们可以通过对CPU降频来确认这一事实，如果性能按比例进行变化，那么CPU就是瓶颈所在。

3.3 应用程序阶段的瓶颈定位

以下是应用程序阶段的瓶颈定位的一些策略的总结：

- **可以用Profiler工具查看CPU的占用情况。**主要是看当前的程序是否使用了接近100%的CPU占用。比如AMD出品的Code
  Analyst代码分析工具，可以对运行在CPU上的代码进行分析和优化。Intel也出品了一个称为Vtune的工具，可以分析在应用程序或驱动器（几何处理阶段）中时间花费的位置情况。
- *一种巧妙的方法是发送一些其他阶段工作量极小甚至根本不工作的数据*。对于某些API而言，可以通过简单地使用一个空驱动器（就是指可以接受调用但不执行任何操作）来取代真实驱动器来完成。这就有效地限制了整个程序运行的速度，因为我们没有使用图形硬件，因此CPU始终是瓶颈。通过这个测试，我们可以了解在应用阶段没有运行的阶段有多大的改进空间。也就是说，请注意，使用空驱动程序还隐藏了由于驱动程序本身和阶段之间的通信所造成的瓶颈。

- **另一个更直接的方法是对CPU 进行降频(Underclock）**。如果性能与CPU速率成正比，则应用程序的瓶颈与CPU相关。但需要注意，降频的方法可以帮助识别瓶颈，也有可能导致一个之前不是瓶颈的阶段成为瓶颈。

四、渲染管线的优化策略

4.1 对CPU的优化策略

4.1.1 减少资源锁定

有时候Cpu和Gpu需要一些同步操作，类似DX12里的围栏，Cpu需要等GPU给信号才能继续执行。
而减少资源锁定的方法，可以尝试避免访问渲染期间GPU正在使用的资源。

4.1.2 合批

不多说

4.2 光栅化阶段的优化策略

- **善用背面裁剪。**对封闭（实心）的物体和无法看到背面的物体（例如，房间内墙的背面）来说，应该打开背面裁剪开关。这样对于封闭的物体来说，可以将需光栅化处理的三角形数量减少近50%。但需要注意的是，虽然背面裁剪可以减少不必要的图元处理，但需要花费一定的计算量来判断图元是否朝向视点。例如，如果所有的多边形都是正向的，那么背向裁剪计算就会降低几何阶段的处理速度。
- **另一种适用于光栅化阶段的优化技术是进行合适的纹理压缩。**如果在送往图形硬件之前已经将纹理压缩好，那么将它发送到纹理内存中的速度将会非常迅速。压缩纹理的另一个优点是可以提高缓存使用率，因为经过压缩的纹理会使用更少的内存。
- **另一种有用的相关优化技术是基于物体和观察者之间的距离，使用不同的像素着色器** LOD
- **理解光栅化阶段的行为。**为了很好地理解光栅阶段的负荷，可以对深度复杂度进行可视化，所谓的深度复杂度就是指一个像素被接触的次数。生成深度复杂度图像的一种简单方法就是，使用一种类似于OpenGL的glBlendFunc(GL ONE,GL ONE)调用，且关闭Z缓冲。首先，将图像清除成黑色；然后，对场景中所有的物体，均使用颜色(0,0,1)进行渲染。而混合函数（blend function）设置的效果即是对每个渲染的图元来说，可以将写入的像素值增加(0,0,1)。那么，深度复杂度为0的像素是黑色，而深度复杂度为255的像素为全蓝色（0, 0, 255）。
- **可以通过计数得到通过Z缓冲与否的像素的数量，从而确定需进一步优化的地方。**使用双通道的方法对那些通过或没通过Z缓冲深度测试的像素进行计数。在第一个通道中，激活Z缓冲，并对那些通过深度测试的像素进行计数。而对那些没有通过深度测试的像素进行计数，可以通过增加模板缓冲的方式。另一种方法是关闭Z缓冲进行渲染来获得深度复杂度，然后从中减去第一个通道的结果。
  通过上述方法得到结果后，可以确认：
  （1） 场景中深度复杂度的平均值、最小值和最大值
  （2） 每个图元的像素数目（假定已知场景中图元的数目）；
  （3） 通过或没有通过深度测试的像素数目。
  而上述这些像素数量对理解实时图形应用程序的行为、确定需要进一步优化处理的位置都非常有用。
  通过深度复杂度可以知道每个像素覆盖的表面数量，重复渲染的像素数量与实际绘制的表面的多少是相关的。假设两个多边形覆盖了一个像素，那么深度复杂度就是2。如果开始绘制的是远处的多边形，那么近处的多边形就会重复绘制整个远处的多边形，重绘数量也就为1。如果开始绘制的是近处的多边形，那么远处的多边形就不会通过深度测试，从而也就没有重绘问题。假设有一组不透明的多边形覆盖了一个像素，那么平均绘制数量就是调和级数。
  通过以上信息，过滤掉不必要的ps开销。

jia