**图形渲染管线之旅——3、3D流水线概述，顶点处理**

至此，我们已经从应用程序通过各种驱动程序层和命令处理器发送了绘图调用。 现在，我们将对其进行一系列图形处理！ 在这一部分中，我将介绍顶点管线。 但是在我们开始之前…

**名称定义**

现在，我们讨论3D渲染管线。3D渲染管线由几个阶段组成，每个阶段都完成一项特定的工作。 我将给所有要讨论的阶段命名，名称将与“官方” D3D10 / 11名称保持一致性，并加上相应的首字母缩写词。 我们最终将在整篇文章中看到所有这些内容，但需要花点时间（以及其他几个部分）-这个系列至少会让我忙2周！

* IA —输入汇编器。 读取索引和顶点数据。
* VS —顶点着色器。 获取输入的顶点数据，为下一阶段写出已处理的顶点数据（包括坐标转换和顶点光照等）。
* PA —图元（Pirmitive）装配。 读取组成基本体的顶点并将其传递。
* HS —Hull着色器，也称TESS Control Shader， 接受修补程序原语，编写转换的（或不提供）修补程序控制点，域着色器的输入，以及一些驱动细分的额外数据。
* TS —镶嵌阶段。 为棋盘形线或三角形创建顶点和连通性。
* DS —域着色器，也称TESS Evaluation Shader， 获取阴影的控制点，HS的额外数据和TS的细分位置，然后将它们再次转换为顶点。
* GS —几何着色器； 输入图元（可带有邻接信息），然后输出不同的图元。
* SO —流输出。 将GS输出（即转换后的基元）写入内存中的缓冲区。
* RS —栅格化器。 栅格化图元。
* PS —像素着色器。 获取插值的顶点数据，输出像素颜色。 也可以写入UAV（无序访问视图）。
* OM —输出合并。 从PS获取像素，进行alpha混合，然后将其写回到后缓冲区。
* CS —计算着色器。 在其自己的管道中全部靠自己。 只有输入是常量缓冲区+线程ID。 可以写入缓冲区和无人机。

既然这已经不成问题了，那么下面是我将要讨论的各种数据路径的列表：（在此，我将省略IA，PA，RS和OM阶段，因为出于我们的目的，实际上不对数据做任何事情，它们只是重新排列/重新排序-即它们本质上是胶合的）

1. VS→PS：最早的可编程管线。 在D3D9中，这就是您所拥有的。 到目前为止，仍然是常规渲染的最重要途径。
2. VS→GS→PS：添加几何着色（D3D10的新增功能）。
3. VS→HS→TS→DS→PS，VS→HS→TS→DS→GS→PS：添加镶嵌（D3D11中的新增功能）。
4. VS→SO，VS→GS→SO，VS→HS→TS→DS→GS→SO：添加流输出（带或不带细分）。
5. CS：Compute Shader。 D3D11的新功能。

现在，让我们从顶点着色器开始！

**输入组装阶段（Input Assembler）**

在这个阶段首先是从索引缓冲区加载索引-如果它是已索引的批处理。 如果不带索引，只需为它生成一个索引缓冲区（0 1 2 3 4…），然后将其用作索引。 如果存在索引缓冲区，则此时将从内存中读取其内容–尽管不是直接读取，IA通常会为索引/顶点缓冲区访问建立一个本地数据缓存。 还要注意，索引缓冲区读取（实际上，D3D10 +中的所有资源访问）都要边界检查； 如果引用原始索引缓冲区之外的元素（例如，从5索引缓冲区发出IndexCount == 6的索引），则所有越界读取均返回零。 虽然返回零完全没有用，但这是定义好的。 同样，您可以给DrawIndexed的索引缓冲区指针传NULL –行为与您设置大小为零的索引缓冲区相同，即所有读取都越界并因此返回零。 在D3D10 +中，你很难触发未定义的行为。 🙂

一旦有了索引，我们就可以从输入顶点数据流中读取每个顶点数据和每个实例数据（当前实例ID是另一个计数器，无论如何在这个阶段都相当简单）。 这非常简单–我们已经声明了数据布局； 只需从缓存/内存中读取它，然后将其解压缩为着色器核心需要输入的float格式即可。 但是，该读取操作不会立即完成。 硬件有一个已着色顶点的缓存，因此，如果一个顶点被多个三角形引用（通常在一个闭合三角形网格中，每个顶点被大约6个tris引用！），不需要多次为这个顶点着色–只需引用已经存在的着色数据！

**顶点缓存和着色**

注意 ：本节的内容部分是猜测。 它们基于人们对当前GPU的公开讨论，但这只是告诉我“什么”，而不是“为什么”，因此这里有一些推断。 当然，我只是在一些细节有所猜测，就是说，我并没有完全在猜测-我有信心，我在这里描述的东西既合理又行得通（从一般意义上来说），我只是不能保证硬件上确实如此，或者没有遗漏。 🙂

长期以来（直到并包括着色器模型3.0时代GPU），顶点和像素着色器都是由具有不同性能折衷的不同单元实现的，并且顶点缓存是一件相当简单的事情：通常只是一个FIFO用于使用顶点索引作为标记，使用很少数量的顶点（想像一两个），为最坏情况的属性输出留出足够的空间。 如前所述，相当简单。

然后出现了统一的着色器。 如果您将以前不同的两种着色器统一起来，则设计必然会有所妥协。 因此，一方面，你拥有顶点着色器，（在当时）在正常使用中，它可能一帧计算不多于一百万个顶点。 另一方面，您有像素着色器，在1920×1200的像素着色器下，一帧至少要计算 230万像素才能填充整个屏幕 –考虑模型互相重叠，实际需要更多。 那么，猜猜这两个单元中的哪个拉了后腿？

好的，这就是问题所在：不像以前旧的顶点着色器单元那样一次或多或少地着色顶点，统一的着色器单元旨在最大的吞吐量而不是延迟，因此需要大批次的计算工作（有多大？现在通常一个批次为16到64个顶点着色）。

因此，如果你不想低效地着色，当存在16-64个顶点高速缓存未命中状态时需要分派一个顶点着色单元。 但是，整个FIFO并没有真正实现这种将顶点缓存未命中进行批处理并一次性着色的机制。 问题是这样的：如果你一次着色一整批顶点，就意味着只有在所有这些顶点都完成着色之后才能开始组装三角形。 此时，你刚刚在FIFO的末尾又添加了一整批的顶点（在这里和下面仅说32个），这意味着现在有32个老的顶点弹出-但是这32个顶点中的每一个都可能是我们正在尝试组装的当前批中的一个三角形的顶点！ 嗯，那不行。 显然，我们无法将FIFO中32个最老的顶点算作顶点缓存命中，因为到我们要引用它们的时候，它们已经消失了！ 另外，我们要使该FIFO多大？ 如果我们要批量着色32个顶点，则它的大小至少需要32个，但是由于我们不能使用32个最老的条目（因为我们将它们移出了），所以这意味着我们将有效地开始每个批次都有一个空的FIFO。 那么，将其扩大，比如说64个条目？ 很大 还要注意，每个顶点缓存查找都涉及将标记（顶点索引）与FIFO中的所有标记进行比较–这是完全并行的，但是它也是耗电的。 我们在这里有效地实现了完全关联的缓存。 另外，在分配32个顶点的着色负载与接收结果之间我们该怎么做–只是等待？ 这种阴影将花费数百个周期，等待似乎是一个愚蠢的主意！ 也许在飞行中有两个平行的阴影负载？ 但是现在我们的FIFO需要至少有64个条目长，并且我们不能将最后64个条目算作顶点缓存命中，因为在我们收到结果时它们将被移出。 另外，一个FIFO与大量着色器核心吗？ 阿姆达尔定律仍然成立-将一个严格串行的组件放置在原本完全并行的管道中，无疑是使其成为瓶颈的方法。

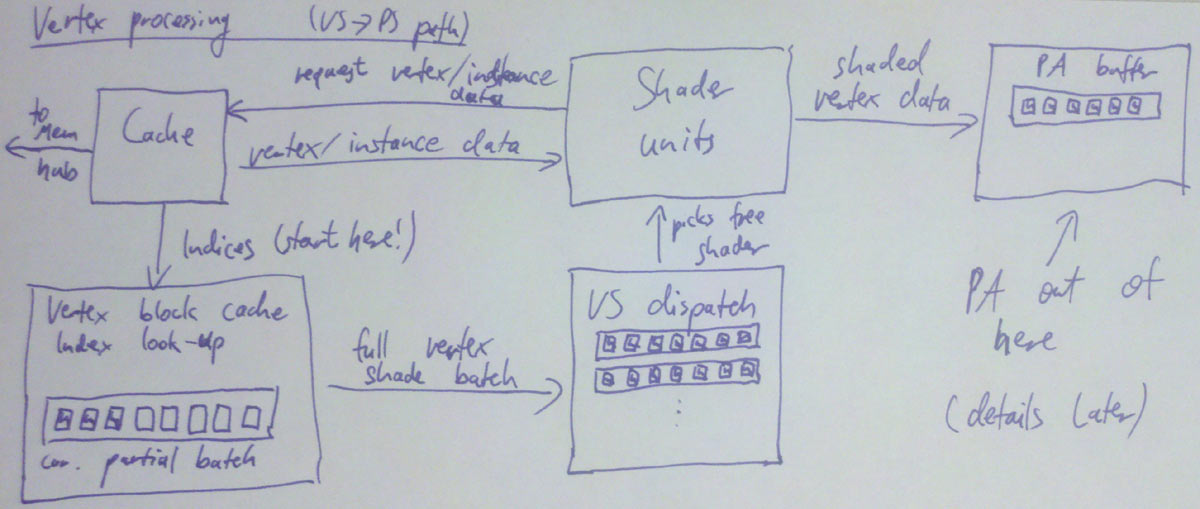
FIFO确实不能很好地适应这种环境，因此，就把它扔掉。 我们实际上想做什么？ 获取一批适当大小的顶点以进行着色，而不是（不必要地）频繁地对顶点进行着色。

因此，简单起见：为32个顶点（= 1个批处理）保留足够的缓冲区空间，并为32个条目类似地缓存标签空间。 从一个空的“缓存（Cache）”开始，即所有条目无效。 对于索引缓冲区中的每个图元，对所有索引进行查找； 如果它在缓存中命中，那就好。 如果未命中，请在当前批次中分配一个插槽，然后将新索引添加到缓存标签数组。 一旦我们没有足够的空间来添加新的图元，就调度整个批次进行顶点着色，保存缓存标签数组（即已经着色顶点的32个索引），然后开始设置下一个批次，再次从空Cache开始–确保批次完全独立。

每一批将让着色器单元繁忙一段时间（可能至少几百个Cycle！）。 但这没问题，因为我们有很多着色器单元–只需每批用不同的单元来执行！ Presto并行性。 在这一点上，我们可以使用保存的缓存标记和原始索引缓冲区数据来组装要沿管道发送的图元（这是“图元组装”的工作，我将在后面的部分中介绍）。

顺便说一句，着色器的结果到哪里去了？ 有两个主要选择：1.专用缓冲区或 2.一些常规的缓存/便笺式存储器。 以前结果输出到1，围绕顶点数据结构设计了一个固定模块（每个顶点具有16个float4属性属性的空间，依此类推），但是最近GPU似乎正在向2迈进，即就是“内存”。 它具有更大的灵活性，并具有明显的优势，举例来说，该内存可以用于其它着色器阶段，而专门的顶点缓存之类的东西对于像素着色或计算管线来说是毫无用处的。

更新 ：这是到目前为止所描述的顶点着色数据流的[图片](http://www.farbrausch.de/~fg/gpu/vertex_shade.jpg) 。



**着色器单元内部**

一句话：查看反汇编的HLSL编译器输出，就了解了（ 可使用fxc /dumpbin）。 只有处理器才真正能够运行这种代码，在硬件中实际上把这些代码编译成着色器字节码。 shader指令的文档相当好-如果你有兴趣，只需查看AMD和NVidia的会议演示文稿或阅读CUDA / Stream SDK的文档即可。

简要的说：快速ALU主要围绕FMAC（浮点累加）单元构建，从硬件层面支持求倒数，倒数平方根，log2，exp2，sin，cos，并针对高数据吞吐量和高数据密度进行了优化，但没有降低延迟的优化，运行大量线程以覆盖延迟，每个线程的寄存器数量很少（因为运行着很多线程！），非常擅长执行线性代码，在分支方面则很糟糕（特别是复杂的分支）。

这几乎是所有实现所共有的。 但也有一些区别。 AMD硬件过去一直直接使用HLSL / GLSL和着色器字节码所隐含的4-wide SIMD（128bit，如vec4，但最近已不再使用），而NVidia不久前决定将4路SIMD转换为标量指令。这些都在网上都能查到！

各个着色器阶段之间的差异很少。 例如，所有算术和逻辑指令在所有阶段都完全相同。 虽然某些结构（例如像素着色器中的派生指令和插值属性）仅在某些阶段存在，但是大多数情况下，它们的差异仅在于数据传入和传出的类型（和格式）。

有一个与着色器有关的特殊之处，就是是纹理采样（和纹理单位）。 这是一个足够大的话题，将成为下一篇的主题！ 回头见。

**结束语**

我要再重复一次“顶点缓存和着色”部分的免责声明：部分内容来源于我的猜想，因此你大可不必采信。 我也不会详细介绍如何管理暂存/缓存。 缓冲区的大小（主要）取决于待处理的批次的大小以及期望的顶点输出属性的数量。 缓冲区大小和管理对于性能而言确实很重要，但是我在这里不能也不想做有意义的解释。 尽管很有趣，但是在不同的硬件上是完全不同的。