# 2011年图形管道之旅：索引

2012年9月22日

0:01

## 2011年图形管道之旅：索引

2011年7月9日欢迎光临。

这是我目前正在写的一系列关于由GPU实际实现的D3D/OpenGL图形管道的博客文章的索引页。这在图形程序员中是众所周知的，关于它的各个部分有大量的文章，但有一点让我感到恼火的是，虽然有关于各个组件的广泛概述和非常详细的信息，但两者之间的内容并不多，而且几乎没有过时的内容。

本系列面向熟悉现代3D API（至少OpenGL 2.0+或D3D9+）并希望了解其外观的图形程序员。它不是为新手描述图形管道；如果您没有使用3D API，那么大部分（如果不是全部的话）对您来说都是毫无用处的。我还假设你对当代硬件设计有一个有效的理解——你至少应该知道什么是寄存器、FIFO、缓存和管道，并了解它们是如何工作的。最后，您至少需要对基本的并行编程机制有一个实际的了解。GPU是一台大规模并行计算机，没有办法绕过它。

一些读者评论说，这是对图形管道和GPU的低级描述；这取决于你站在哪里。GPU架构师会称之为GPU的高级描述。当新一代GPU出现时，你往往会在硬件审查网站上看到彩色流程图，但没有这些流程图那么高级；但是，老实说，这种报告往往信息密度很低，即使做得很好。归根结底，这并不是为了解释任何东西实际上是如何工作的——它只是试图展示闪亮新玩意儿的科技色情片。好吧，我试着在这里更充实一点，不幸的是，这意味着更少的颜色和更少的基准测试结果，而是大量的文本，一些单色图表，甚至一些（抖动）方程。如果你同意的话，下面是索引：

§  第一部分：导言；软件栈。

§  第2部分：GPU内存体系结构和命令处理器。

§  第3部分：三维管线概述，顶点处理。

§  第4部分：纹理采样器。

§  第5部分：基本体装配、剪裁/消隐、投影和视口变换。

§  第6部分：（三角形）光栅化和设置。

§  第7部分：Z/模板处理，3种不同的方式。

§  第8部分：像素处理-“分叉阶段”。

§  第9部分：像素处理-“连接阶段”。

§  第10部分：几何体着色器。

§  第11部分：流出。

§  第12部分：镶嵌。

§  第13部分：计算着色器。

# 2011年图形管道之旅，第1部分

2012年9月21日

23:51

## 2011年图形管道之旅，第1部分

2011年7月1日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

我在这里发布一些东西已经有一段时间了，我想我可以用这个点来解释一些关于2011年图形硬件和软件的一般观点；你可以找到你电脑中图形堆栈的功能描述，但通常不是“如何”或“为什么”；我会尽量在不太具体说明任何特定硬件的情况下填补空白。我将主要讨论在Windows上运行D3D9/10/11的DX11类硬件，因为这恰好是（PC）我最熟悉的堆栈–API细节等在第一部分之后并不重要；一旦我们真正使用GPU，就都是本机命令了。

## 应用程序

这是你的代码。这些也是你的bug。真的。是的，API运行时和驱动程序都有bug，但这不是其中之一。现在就去修复它吧。

## API运行时

您对API进行资源创建/状态设置/绘制调用。API运行时跟踪应用程序已设置的当前状态，验证参数并执行其他错误和一致性检查，管理用户可见资源，可能验证也可能不验证着色器代码和着色器链接（或者至少D3D是这样的，在OpenGL中，这是在驱动程序级别处理的）可能会进行更多的批处理，然后将其全部交给图形驱动程序——更准确地说，是用户模式驱动程序。

## 用户模式图形驱动程序（或UMD）

这就是CPU端的大部分“魔法”发生的地方。如果你的应用程序因为你做的某个API调用而崩溃，它通常会在这里：）。它被称为“nvd3dum.dll”（NVidia）或“atiumd\*.dll”（AMD）。顾名思义，这是用户模式代码；它与你的应用程序（以及API运行时）在相同的上下文和地址空间中运行它实现了一个由D3D调用的低级API（DDI）；该API与您在表面上看到的API非常相似，但在内存管理等方面更为明确。

此模块是着色器编译之类的事情发生的地方。D3D将预先验证的着色器令牌流传递给UMD–即，已经检查代码是否在语法正确和遵守D3D约束的意义上有效（使用正确的类型，不使用比可用纹理/采样器更多的纹理/采样器，不超过可用常量缓冲区的数量，诸如此类）。这是从HLSL代码编译而来的，通常有很多高级优化（各种循环优化、死代码消除、常量传播、预测ifs等）应用于它–这是个好消息，因为这意味着驱动程序从编译时执行的所有这些相对昂贵的优化中获益。但是，它也有一系列较低级别的优化（例如寄存器分配和循环展开）应用驱动程序更愿意自己做；长话短说，这通常只是立即转化为中间表示（IR），然后再编译一些；着色器硬件与D3D字节码非常接近，编译不需要创造奇迹就能产生好的结果（HLSL编译器已经完成了一些高产量和高成本的优化，这无疑是有帮助的），但仍有许多低级细节（如硬件资源限制和调度约束）D3D既不知道也不关心，因此这不是一个简单的过程。

当然，如果你的应用程序是一款知名的游戏，NV/AMD的程序员可能已经看过你的着色器，并为他们的硬件编写了手动优化的替代品——尽管他们最好能产生相同的结果，以免出现丑闻：）。这些着色器也会被UMD检测和替换。欢迎你。

更有趣的是：一些API状态实际上可能最终被编译到着色器中–举个例子，纹理采样器中可能没有实现纹理边界等相对奇特（或至少很少使用）的功能，但在着色器中使用额外的代码进行模拟（或根本不支持）。这意味着，对于不同的API状态组合，同一着色器有时会有多个版本浮动。

顺便说一句，这也是为什么您在第一次使用新着色器或资源时经常会看到延迟的原因；很多创建/编译工作都是由驱动程序延迟的，只有在实际需要时才执行（您不会相信某些应用程序会创建多少未使用的垃圾！）。图形程序员知道事情的另一面–如果您想确保确实创建了某些内容（而不仅仅是保留内存），则需要发出一个虚拟绘图调用，用它来“预热”.丑陋又烦人，但自从1999年我第一次开始使用3D硬件以来，情况就一直如此——这意味着，到目前为止，这几乎是生活中的一个事实，所以要习惯它。：）

无论如何，继续。UMD还可以处理一些有趣的东西，比如所有D3D9“遗留”着色器版本和固定函数管道–是的，所有这些都会被D3D忠实地传递。3.0着色器配置文件并没有那么糟糕（事实上这是相当合理的），但2.0是粗糙的，各种1.x着色器版本都非常糟糕——还记得1.3像素着色器吗？或者，就这点而言，是带有顶点照明的固定功能顶点管道之类的吗？是的，支持D3D中仍然存在的所有功能和每个现代图形驱动程序的精髓，当然他们只是将其转换为更新的着色器到现在为止（并且已经这样做了很长一段时间）。

还有内存管理之类的东西。UMD将获得纹理创建命令之类的东西，并且需要为它们提供空间。实际上，UMD只是将它从KMD（内核模式驱动程序）获得的一些较大的内存块进行子分配；实际上映射和取消映射页面（管理UMD可以看到的视频内存的哪一部分，反之，GPU可以访问系统内存的哪一部分）是内核模式权限，UMD无法完成。

但是UMD可以做一些事情，比如旋转纹理（除非GPU可以在硬件中这样做，通常使用2D blitting单元而不是真正的3D管道）和安排系统内存和（映射的）视频内存之间的传输等等。最重要的是，它还可以写入命令缓冲区（或“DMA缓冲区”）–在KMD分配并移交后，我将交替使用这两个名称。命令缓冲区包含命令：）。所有状态更改和绘图操作都将由UMD转换为硬件能够理解的命令。许多您无法手动触发的操作也会如此，例如将纹理和着色器上载到视频内存。

一般来说，驱动程序会尝试将尽可能多的实际处理放在UMD中；UMD是用户模式代码，因此在其中运行的任何东西都不需要任何昂贵的内核模式转换，它可以自由分配内存，将工作分配给多个线程，等等——它只是一个常规DLL（即使它是由API加载的，而不是直接由你的应用程序加载的）。这对驱动程序开发也有好处——如果UMD崩溃，应用程序也会随之崩溃，但不会导致整个系统崩溃；在系统运行时可以替换它（它只是一个DLL！）；它可以使用常规调试器进行调试；等等。因此，它不仅高效，而且方便。

但是房间里有一头大象，我还没有提到。

**我说了什么**“用户模式驱动程序”？我的意思是“用户模式驱动程序”。

如前所述，UMD只是一个DLL。好吧，一个恰巧有D3D的祝福和KMD的直接管道的DLL，但它仍然是一个常规DLL，并且在其调用进程的地址空间中运行。

但我们现在使用多任务操作系统。事实上，我们已经使用了一段时间。

我一直在说的这个“GPU”东西？那是一个共享资源。只有一个驱动你的主显示器（即使你使用SLI/Crossfire）。但是我们有多个应用程序试图访问它（并假装只有他们在做）。这不仅仅是自动工作；在过去，解决方案是一次只为一个应用程序提供3D，而当该应用程序处于活动状态时，所有其他应用程序都无法访问。但如果您试图让您的窗口系统使用GPU进行渲染，这并不能真正解决问题。这就是为什么您需要一些仲裁访问权限的组件o打开GPU并分配时间片等。

**输入调度程序。**

这是一个系统组件–请注意“the”有点误导；我这里说的是图形调度器，而不是CPU或IO调度器。这正是你所认为的——它通过在想要使用3D管道的不同应用程序之间进行时间切片来仲裁对3D管道的访问。上下文切换至少会在GPU上引起一些状态切换（这会为命令缓冲区生成额外的命令），也可能会在视频内存中交换一些资源。当然，在任何给定的时间，只有一个进程可以实际向3D管道提交命令。

你经常会发现控制台程序员抱怨PC 3D API的相当高的级别和不干涉的性质，以及由此带来的性能成本。但问题是，PC上的3D API/驱动程序确实比控制台游戏要解决更复杂的问题——例如，他们确实需要跟踪完整的当前状态，因为有人他们也会处理坏掉的应用程序，并试图在背后解决性能问题；这是一种令人讨厌的做法，没有人会满意，当然也包括驱动程序作者自己，但事实是，商业视角在这里占了上风；人们期望的东西不会运行o继续运行（并且做得很顺利）。在应用程序上大喊“但这是错误的！”然后生气，走一条超慢的路，你不会赢得任何朋友。无论如何，继续管道。下一站：内核模式！

## 内核模式驱动程序（KMD）

这是实际处理硬件的部分。一次可能有多个UMD实例运行，但只有一个KMD，如果崩溃，那么你就死了——以前是“蓝屏”死机，但现在Windows实际上知道如何杀死崩溃的驱动程序并重新加载它（进度！）。只要它恰好只是一个崩溃，至少不是内核内存损坏——如果发生这种情况，所有的赌注都没有了。

KMD处理所有只存在一次的东西。只有一个GPU内存，即使有多个应用程序在争夺它。需要有人发号施令，实际分配（和映射）物理内存。同样，必须有人在启动时初始化GPU，设置显示模式（并从显示器获取模式信息），管理硬件鼠标光标（是的，有硬件处理，是的，你真的只有一个！：），编程硬件看门狗定时器，使GPU在某一时间保持无响应时复位，响应中断，等等。这就是KMD所做的。

还有一个完整的内容保护/DRM部分是关于在视频播放器和GPU之间设置一个受保护的/DRM’ed路径，这样，任何肮脏的用户模式代码都看不到真正珍贵的解码视频像素，这可能会做一些可怕的禁止的事情，比如将它们转储到磁盘（…无论如何）。KMD也参与了其中。

对我们来说最重要的是，KMD管理实际的命令缓冲区。你知道，硬件实际消耗的命令缓冲区。UMD产生的命令缓冲区不是真正的交易——事实上，它们只是GPU可寻址内存的随机片。实际发生的是，UMD完成它们，将它们提交给s调度程序，然后等待该进程结束，然后将UMD命令缓冲区传递给KMD。KMD然后将对命令缓冲区的调用写入主命令缓冲区，根据GPU命令处理器是否可以从主内存读取，它可能还需要先将其DMA到视频内存。主命令缓冲区通常是y一个（相当小的）环形缓冲区——在那里唯一被写入的东西是系统/初始化命令和对“真实的”丰富的3D命令缓冲区的调用。

但现在这仍然只是内存中的一个缓冲区。图形卡知道它的位置——通常有一个读指针，GPU在主命令缓冲区中的位置，还有一个写指针，KMD已经写了缓冲区多远（或者更准确地说，它已经告诉GPU它已经写了多远）.这些是硬件寄存器，它们是内存映射的–KMD定期更新它们（通常在提交新的工作块时）…

## 公共汽车

…但当然，写入不会直接进入图形卡（至少除非它集成在CPU芯片上！），因为它需要先通过总线-通常是现在的PCI Express。DMA传输等采用相同的路径。这并不需要很长时间，但这是我们旅程的另一个阶段。直到最后…

**命令处理器！**

这是GPU的前端——实际上是读取KMD编写的命令的部分。我将在下一期文章中从这里继续，因为这篇文章已经足够长了：）

## 小型软件：OpenGL

OpenGL与我刚才描述的非常相似，只是API和UMD层之间没有那么明显的区别着色器编译根本不由API处理，而是由驱动程序完成。一个不幸的副作用是，GLSL前端与3D硬件供应商一样多，它们基本上都实现了相同的规范，但都有自己的bug和特性。这不好玩。这也意味着驱动程序必须进行所有优化无论何时，只要他们看到着色器，都会自动编译—包括昂贵的优化。D3D字节码格式确实是解决此问题的更干净的解决方案—只有一个编译器（因此不同供应商之间没有略微不兼容的方言！），并且它允许进行比通常情况下更昂贵的数据流分析。

## 遗漏和简化

这只是一个概述；我掩盖了很多微妙之处。例如，不仅有一个调度器，还有多个实现（驱动程序可以选择）；CPU和GPU之间的同步是如何处理的，我到目前为止根本没有解释。等等。我可能忘记了一些重要的事情——如果是这样，请告诉我，我会解决的！但是现在，再见，希望下次再见。

# 2011年图形管道之旅，第2部分

2012年9月21日

23:52

## 2011年图形管道之旅，第2部分

2011年7月2日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

**别那么快。**

在上一部分中，我解释了3D渲染命令在实际传递到GPU之前在PC上所经历的各个阶段；简短版本：这比你想象的要多。然后我结束了对命令处理器的命名，以及它最终如何使用我们精心准备的命令缓冲区。好吧，如何我可以这么说吗？我骗了你。在这一期中，我们确实会第一次见到命令处理器，但是请记住，所有这些命令缓冲区都会通过内存——通过PCI Express访问的系统内存，或者本地视频内存。我们正在按顺序通过管道，所以在进入命令处理器之前，让我们先来看看让我们谈谈记忆。

## 内存子系统

GPU没有常规内存子系统–它不同于您在通用CPU或其他硬件中看到的，因为它是为非常不同的使用模式而设计的。GPU的内存子系统与您在常规机器中看到的内存子系统有两种基本方式不同：

第一个是GPU内存子系统很快。非常快。一个核心i7 2600K可能会达到19 GB/s的内存带宽——在好的一天。顺风。下坡。另一方面，GeForce GTX 480的总内存带宽接近180 GB/s——几乎是一个数量级的差异！哇。

第二个原因是GPU内存子系统很慢。速度非常慢。一台计算机上的主内存缓存丢失

Nehalem（第一代核心i7）如果将AnandTechX给出的内存延迟除以时钟速率，则需要大约140个周期。我前面提到的GeForce GTX 480的内存访问延迟为400-800个时钟。因此，我们只需说，以周期为单位，GeForce GTX 480的平均内存延迟是Core i7的4倍多一点。除了我刚才提到的Core i7之外ed的时钟频率为2.93GHz，而GTX 480着色器的时钟频率为1.4GHz——就是这样，又是2倍。呜呜，又是一个数量级的差异！等等，这里发生了一些有趣的事情。我的常识让人感到刺痛。这一定是我在新闻中不断听到的权衡之一！

是的——GPU的带宽大幅增加，但他们为此付出的代价是延迟大幅增加（事实证明，功率消耗也有相当大的提高，但这超出了本文的范围）。这是一般模式的一部分–GPU都是关于延迟的吞吐量；不要等待尚未出现的结果，而要做其他事情！

这几乎是所有你需要知道的GPU内存，除了一个一般的DRAM小贴士，这将是重要的以后：DRAM芯片被组织为一个二维网格-逻辑和物理。有（水平）行线和（垂直）列线。在这些线之间的每个交叉点上都有一个晶体管和一个电容器；如果你想知道如何从这些成分中建立记忆，维基百科是你的朋友。无论如何，这里最突出的一点是，DRAM中某个位置的地址被分为行地址和列地址，DRAM内部的读/写操作最终总是同时访问给定行中的所有列。这意味着访问恰好映射到一个DRAM行的大量内存要比访问分布在多个行上的相同数量的内存便宜得多。现在，这似乎只是一个随机的DRAM琐事，但这将在以后变得重要；换句话说，请注意：这将在考试中出现。但是为了与前面几段中的数字联系起来，让我注意到，仅仅读取内存中的几个字节是无法达到上面的峰值内存带宽数字的；如果您想使内存带宽饱和，最好一次执行一整行DRAM。

## PCIe主机接口

从图形程序员的角度来看，这件硬件并不是非常有趣。事实上，GPU硬件架构师也是如此。问题是，一旦它变得如此缓慢以至于成为瓶颈，你仍然会开始关注它。所以，你要做的就是让好的人正确地去做，以确保这不会发生。除此之外，这给了CPU对视频内存和一堆GPU寄存器的读/写访问权，GPU对（一部分）主内存的读/写访问权，每个人都很头疼，因为所有这些事务的延迟比内存延迟更糟糕，因为信号必须从芯片中出来，进入插槽，在主板上移动一点，大约一周后到达CPU中的某个地方（或者与CPU/GPU速度相比，感觉就是这样）。不过带宽还是不错的–目前大多数GPU使用的16通道PCIe 2.0连接的峰值聚合带宽高达约8GB/s（理论值），因此在聚合CPU内存带宽的一半到三分之一之间；这是一个可用的比率。与AGP等早期标准不同，这是一个对称的点对点链路——带宽双向传输；AGP有一个从CPU到GPU的快速通道，但不是相反。

## 一些最后的记忆片段

老实说，我们现在已经非常接近真正看到3D命令了！如此接近你几乎可以尝到它们。但还有一件事我们需要先让开。因为现在我们有两种内存-（本地）视频内存和映射系统内存。一个是去北方一天的路程，另一个是沿着PCI快速公路去南方一周的路程。我们选择哪条路？

最简单的解决方案是：只需添加一个额外的地址行，告诉你该走哪条路。这很简单，效果很好，已经做了很多次了。或者，您可能使用的是统一的内存体系结构，比如一些游戏机（但不是PC）。在这种情况下，别无选择；只有记忆，就是你要去的地方，句号。如果你想要更新奇的东西，你可以添加一个MMU（内存管理单元），它为你提供了一个完全虚拟化的地址空间，并允许你玩一些不错的把戏，比如在视频内存中频繁访问纹理的某些部分（它们很快），在系统内存中访问其他部分，而其中大部分根本没有映射——从thing air中变出来，或者，更常见的是，通过魔法磁盘读取，只需要大约50年——顺便说一句，这并不夸张；如果你坚持使用“内存访问=1天”的比喻，那就是单次高清读取所需的时间。一个相当快的。磁盘很烂。但我离题了。

所以，嗯。它还允许您对视频内存地址空间进行碎片整理，而无需在视频内存耗尽时实际复制内容。这是件好事。而且让多个进程共享同一个GPU变得更容易。它是绝对允许有一个，但我不确定它是否是一个要求，即使它确实是非常好的有（有人关心帮助我在这里吗？我会更新文章，如果我得到澄清，但tbh现在我只是不能被火查到）。无论如何，MMU/虚拟内存并不是真正可以只在一旁添加的东西（不管怎样，在一个有缓存和内存一致性问题的体系结构中也不是这样），但它确实不是特定于任何特定阶段的——我必须在某个地方提一下，所以我把它放在这里。

还有一个DMA引擎，它可以复制内存，而无需涉及任何宝贵的3D硬件/着色器核心。通常，这至少可以在系统内存和视频内存之间进行复制（在两个方向上）。它通常还可以从视频内存复制到视频内存（如果您必须进行任何VRAM碎片整理，这是一件非常有用的事情）。它通常无法进行系统内存到系统内存的复制，因为这是一个GPU，而不是内存复制单元–在CPU上进行系统内存复制，而不必在两个方向上通过PCIe！

**更新**：我画了一张图（链接，因为这个布局太窄，无法在文本中放置大图）。这还显示了一些更多的细节——到目前为止，您的GPU有多个内存控制器，每个控制器控制多个内存库，前面有一个胖集线器。无论付出什么代价都能获得带宽。：）

好的，清单。我们在CPU上准备了一个命令缓冲区。我们有PCIe主机接口，因此CPU实际上可以告诉我们这一点，并将其地址写入某个寄存器。我们有逻辑将该地址转换为实际返回数据的负载-如果它来自系统内存，则通过PCIe，如果我们决定在视频内存中使用命令缓冲区，则KMD可以设置DMA传输，因此CPU和GPU上的着色器内核都不必主动担心它。然后我们可以通过存储子系统从视频存储器中的拷贝中获取数据。我们已经设置好所有路径，并最终准备好查看一些命令！

**终于，命令处理器！**

我们对命令处理器的讨论从一个词开始，就像现在很多事情一样：

“缓冲…”

如上所述，通向此处的两条内存路径都具有高带宽和高延迟。对于GPU管道中的大多数后期位，解决此问题的选择方法是运行大量独立线程。但在本例中，我们只有一个命令处理器，需要按顺序仔细检查命令缓冲区（因为此命令缓冲区包含需要按正确顺序执行的状态更改和渲染命令等内容）。因此，我们做了下一件最好的事情：添加足够大的缓冲区，并提前足够远的预取，以避免打嗝。

从该缓冲区，它进入实际的命令处理前端，该前端基本上是一个知道如何解析命令（使用特定于硬件的格式）的状态机。有些命令处理2D渲染操作——除非有单独的2D命令处理器，而且3D前端根本看不到。不管怎样，现代GPU上仍然隐藏着专用的2D硬件，就像那个芯片上的VGA芯片一样，仍然支持文本模式、4位/像素位平面模式、平滑滚动等等。祝你在没有显微镜的情况下在模具上找到这些东西。无论如何，这种东西是存在的，但从今以后我不再提它了还有一些命令实际上会将一些基本体交给3D/着色器管道，呜呜呜！我将在接下来的部分介绍它们。由于各种原因（以及在各种管道配置中），还有一些命令进入3D/着色器管道，但从不渲染任何内容；这些都是后来才出现的。

然后是改变状态的命令。作为一个程序员，你认为他们只是在改变一个变量，这就是基本上发生的事情。但GPU是一台大规模并行计算机，你不能仅仅在并行系统中改变一个全局变量，然后希望一切正常——如果你不能保证一切都能通过你正在实施的某个不变量来工作，那么就有一个bug，你最终会遇到它。有几种流行的方法，基本上所有的芯片对不同类型的状态使用不同的方法。

•无论何时更改状态，都需要完成所有可能涉及该状态的未决工作（即基本上是部分管道冲洗）。从历史上看，这就是图形芯片处理大多数状态变化的方式——如果批次数量少、三角形少、管道短，那么这种方式简单且成本不高。唉，批量和三角计数都在增加，管道也越来越长，因此这种方法的成本也急剧上升。对于那些很少更改的内容（在整个框架中，十几个部分管道刷新并不是什么大问题），或者是太贵/太难用更具体的方案实现的内容，它仍然存在。

•您可以使硬件单元完全无状态。只需将状态更改命令传递到关心它的阶段；然后让该阶段在每个周期将当前状态附加到它向下游发送的所有内容。它不存储在任何地方，但它总是存在的，所以如果某个管道阶段想要查看状态中的一些位，它可以，因为它们被传入（然后传递到下一个阶段）。如果你的州恰好只有几块，这就不太便宜和实用了。如果它恰好是一整套活动纹理以及纹理采样状态，则不会太多。

•有时只存储状态的一个副本，并且每次阶段更改时都必须刷新，这会使事情序列化太多，但如果您有两个副本（或者可能有四个副本），情况会很好，这样您的状态设置前端就可以领先一点。假设您有足够的寄存器（“插槽”）来存储每个状态的两个版本，并且一些活动作业引用插槽0。您可以安全地修改插槽1，而无需停止该作业或以其他方式干扰它。现在，您不需要通过管道发送整个状态–每个命令只需一个位，即可选择使用插槽0还是插槽1。当然，如果在遇到状态更改命令时插槽0和1都很忙，您仍然需要等待，但您可以提前一步。同样的技术适用于两个以上的插槽。

•对于采样器或纹理着色器资源视图状态等某些内容，您可以同时设置大量的采样器或纹理着色器资源视图状态，但很可能您没有这样做。您不想仅仅因为跟踪2个正在运行的状态集而为2\*128活动纹理保留状态空间，因此可能需要它。对于这种情况，您可以使用一种寄存器重命名方案——拥有128个物理纹理描述符池。如果有人在一个着色器中需要128个纹理，那么状态更改将会很慢。（艰难的突破）。但在使用少于20种纹理的应用程序中，你有相当大的空间来保持多个版本。

这并不是一个全面的列表——但主要的一点是，看起来很简单的事情，比如在你的应用程序中更改一个变量（甚至在UMD/KMD和命令缓冲区中也是如此！），实际上可能需要大量的支持硬件来支持它，以防止它减慢速度。

## 同步

最后，最后一系列命令处理CPU/GPU和GPU/GPU同步。

通常，所有这些都具有“如果事件X发生，则执行Y”的形式。我将首先处理“doy”部分——这里有两个关于Y的合理选择：它可以是一个推送模型通知，其中GPU对CPU大喊大叫，要求它立即执行某些操作(“Oi！CPU！我现在正在输入显示器0上的垂直消隐间隔，所以如果你想翻转缓冲区而不撕裂，现在是时候这样做了！”），或者它可以是一个拉模型，GPU只记住发生了什么，CPU可以稍后询问(“喂，GPU，您开始处理的最新命令缓冲区片段是什么？”—“让我检查……序列id 303。”）前者通常使用中断来实现，并且只用于不频繁和高优先级的事件，因为中断相当昂贵。后者只需要一些CPU可见的GPU寄存器，以及在某个事件发生后从命令缓冲区写入值的方法。

假设您有16个这样的寄存器。然后您可以将currentCommandBufferSeqId分配给寄存器0。您为提交给GPU的每个命令缓冲区分配一个序列号（这在KMD中），然后在每个命令缓冲区的开头添加一个“如果您在命令缓冲区中到达这一点，写入寄存器0〃。瞧，现在我们知道GPU当前使用的是哪个命令缓冲区！而且我们知道命令处理器严格按照顺序完成命令，因此如果命令缓冲区303中的第一个命令被执行，这意味着所有命令缓冲区都达到并包括sequenc！”e ID302已经完成，现在可以被KMD回收，释放，修改，或者变成一个俗气的游乐园。

我们现在还有一个X可能是什么的示例：“如果你到了这里”-也许是最简单的示例，但已经很有用了。其他示例是“如果所有着色器都在命令缓冲区中该点之前完成了来自批次的所有纹理读取”（这标记了回收纹理/渲染目标内存的安全点），“如果已完成对所有活动渲染目标/UAV的渲染”（这将标记实际可以安全地将其用作纹理的点），“如果在此点之前的所有操作都已完全完成”，等等。

顺便说一句，这类操作通常被称为“fences”。选择写入状态寄存器的值有不同的方法，但就我而言，唯一明智的方法是使用顺序计数器（可能会窃取一些位作为其他信息）.是的，我真的只是在这里不带任何理由地删除了一条随机信息，因为我想你应该知道。我可能会在以后的博客文章中对此进行详细阐述（尽管不是在本系列文章中）：）。

因此，我们得到了一半——我们现在可以从GPU向CPU报告状态，这允许我们在驱动程序中进行合理的内存管理（值得注意的是，我们现在可以确定何时可以安全地回收用于顶点缓冲区、命令缓冲区、纹理和其他资源的内存）。但这还不是全部–缺少一个拼图块。例如，如果我们需要纯粹在GPU端进行同步，该怎么办？让我们回到渲染目标示例。在渲染实际完成之前，我们不能将其用作纹理（并且已经执行了一些其他步骤–一旦我进入纹理单元，将提供更多详细信息）。解决方案是“等待”式指令：“等待，直到寄存器M包含值N”。这可以是相等的比较，也可以是小于（注意，这里需要处理wraprounds！），或更奇特的东西–为了简单起见，我只使用equals。这允许我们在提交批处理之前进行渲染目标同步。它还允许我们构建完整的GPU刷新操作：“如果所有挂起的作业完成，将寄存器0设置为++seqId”/“等到寄存器0包含seqId”.Done和Done.GPU/GPU同步：已解决–在DX11引入具有另一种更细粒度同步的计算着色器之前，这通常是GPU端唯一的同步机制。对于常规渲染，您不需要更多。

顺便说一句，如果你可以从CPU端写这些寄存器，你也可以用另一种方式——提交一个包含等待特定值的部分命令缓冲区，然后从CPU而不是GPU更改寄存器。这种事情可以用来实现D3D11风格的多线程渲染，在这里你可以提交一个批处理at引用仍然锁定在CPU端的顶点/索引缓冲区（可能被另一个线程写入）。您只需在实际渲染调用之前填充等待，然后一旦顶点/索引缓冲区实际解锁，CPU就可以更改寄存器的内容。如果GPU在命令缓冲区中从未达到如此程度，那么等待现在就是一个noop；如果它做到了，它将花费一些时间（命令处理器）时间旋转直到数据实际存在。非常漂亮，不是吗？事实上，如果您可以在提交命令缓冲区后修改命令缓冲区，只要有命令缓冲区“跳转”指令，即使没有CPU可写状态寄存器，您也可以实现这类功能。详细信息留给感兴趣的读者：）

当然，您不一定需要set register/wait register模型；对于GPU/GPU同步，您可以简单地使用“rendertarget barrier”指令确保rendertarget可以安全使用，以及“flush everything”命令。但我更喜欢set register样式的模型，因为它可以杀死两只鸟（将使用中的资源反向报告给CPU，以及GPU自同步）使用一块精心设计的石头。

**更新：**在这里，我为您绘制了一张图表。它有点复杂，所以我将在将来降低细节的数量。基本思想是：命令处理器前面有一个FIFO，然后是命令解码逻辑，执行由与2D单元、3D前端（常规3D渲染）通信的各个块处理或者直接使用着色器单元（计算着色器），然后有一个处理sync/wait命令的块（其中包含我提到的公共可见寄存器），还有一个处理命令缓冲区跳转/调用的单元（它更改进入FIFO的当前提取地址）.我们派遣工作的所有单位都需要向我们发送完成事件，以便我们知道何时不再使用纹理，并且可以回收它们的内存。

## 闭幕词

下一步是第一步做任何实际渲染工作。最后，在我的GPU系列中只有3部分，我们实际上开始查看一些顶点数据！（不，还没有三角形被光栅化。这将需要更多的时间）。

实际上，在这个阶段，已经有一个分支在管道中；如果我们正在运行计算着色器，下一步应该已经是…运行计算着色器。但是我们没有，因为计算着色器是后面部分的主题！首先是常规渲染管道。

小免责声明：再一次，我在这里给你大致的笔划，在必要的地方（或有趣的地方）深入细节，但是相信我，我为了方便（和易于理解）而放弃了很多东西.也就是说，我不认为我遗漏了任何真正重要的东西。当然，我可能有些地方弄错了。如果你发现了任何bug，告诉我！

直到下一部分…

# 2011年图形管道之旅，第3部分

2012年9月21日

23:54

## 2011年图形管道之旅，第3部分

2011年7月3日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

在这一点上，我们已经通过各种驱动层和命令处理器从我们的应用程序发送了draw调用；现在，我们终于要对它进行一些图形处理了！在这一部分中，我将研究顶点管道。但在我们开始之前…

**喝点字母汤！**

我们现在在3D管道中，它由几个阶段组成，每个阶段完成一项特定的工作。我将为我将要讨论的所有阶段命名——为了保持一致性，主要使用“官方”D3D10/11名称——加上相应的首字母缩略词。我们最终会在我们的盛大巡演中看到所有这些，但要看到其中大部分还需要一段时间（以及更多的部分）——说真的，我对我想涵盖的领域做了一个小提纲，这个系列至少会让我忙上两周！总之，这里有一句话总结了每个阶段的工作。

•IA-输入汇编程序。读取索引和顶点数据。

•VS-顶点着色器。获取输入顶点数据，为下一阶段写出已处理的顶点数据。

•PA-基本组件。读取组成基本体的顶点并将其传递。

•HS-船体着色器；接受面片基本体、写入已转换（或未转换）的面片控制点、域着色器的输入，以及一些驱动细分的额外数据。

•TS-镶嵌器阶段。为镶嵌线或三角形创建顶点和连接。

•DS-域着色器；获取着色控制点、HS中的额外数据和TS中的细分位置，并将它们再次转换为顶点。

•GS-几何体着色器；输入基本体（可选地带有邻接信息），然后输出不同的基本体。也是…

•如此-流出。将GS输出（即转换的原语）写入内存中的缓冲区。

•RS-光栅化器。光栅化基本体。

•PS-像素着色器。获取插值顶点数据，输出像素颜色。还可以写入无人机（无序访问视图）。

•OM——产出合并。从PS获取着色像素，进行alpha混合并将其写回backbuffer。

•CS-计算着色器。在它自己的管道中。唯一的输入是常量缓冲区+线程ID；可以写入缓冲区和无人机。

现在，这是我将要讨论的各种数据路径的列表，顺序如下：（我将在这里省略IA、PA、RS和OM阶段，因为出于我们的目的，它们实际上对数据不做任何处理，它们只是重新排列/重新排序-即，它们本质上是粘合的）

1.       VS→PS：古老的可编程管道。在D3D9中，这就是你的全部。到目前为止，这仍然是常规渲染最重要的路径。我将从头到尾地讲述这一点，然后在完成后再回到更奇特的路径。

2.       VS→GS→PS：几何体着色（D3D10新增）。

3.       VS→HS→TS→DS→PS，VS→HS→TS→DS→GS→PS：镶嵌（D3D11中新增）。

4.       VS→那么，VS→GS→那么，VS→HS→TS→DS→GS→SO：流出（有镶嵌和没有镶嵌）。

5.       CS:计算机。在D3D11中新增。

现在，您已经知道接下来会发生什么，让我们开始使用顶点着色器！

## 输入汇编阶段

这里发生的第一件事是从索引缓冲区加载索引——如果它是一个索引批处理。如果不是，就假设它是一个标识索引缓冲区（0 1 2 3 4…），并将其用作索引。如果有索引缓冲区，此时将从内存中读取其内容，而不是直接读取，IA通常有一个数据缓存来利用索引/顶点缓冲区访问的局部性。还要注意，索引缓冲区读取（实际上，D3D10+中的所有资源访问都是边界检查的；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DrawIndexed），则所有越界读取返回零。这（在这种特殊情况下）是完全无用的，但定义明确。类似地，您可以使用空索引缓冲区集发出DrawIndexed–这与设置了大小为零的索引缓冲区的方式相同，即所有读取都超出边界，因此返回零。使用D3D10+，您必须进行更多工作才能进入未定义行为的领域。：）

一旦有了索引，我们就可以从输入顶点流中读取每个顶点和每个实例的数据（当前实例ID只是另一个计数器，在这个阶段非常简单）。这相当简单——我们有一个数据布局声明；只需从缓存/内存中读取它，并将其解压缩为着色器核心需要输入的浮点格式。然而，这种阅读不是立即完成的；硬件正在运行着色顶点缓存，因此，如果一个顶点被多个三角形引用（并且在完全规则的闭合三角形网格中，每个顶点将被大约6个三角形引用！），它不需要每次都着色-我们只需引用已经存在的着色数据！

## 顶点缓存和着色

*注*：本节内容的一部分是猜测。它们基于“知情人士”对当前GPU的公开评论，但这只给出了“什么”，而不是“为什么”，所以这里有一些推断。另外，我只是在猜测一些细节。这就是说，我在这里并不是在胡说八道——我相信我在这里描述的内容是合理和有效的（在一般意义上），我只是不能保证在真实的硬件中是这样的，或者我没有遗漏任何棘手的细节。：）

无论如何在很长一段时间内（直到并包括shader model 3.0代GPU），顶点和像素着色器使用不同的单元实现，这些单元具有不同的性能权衡，顶点缓存是一件相当简单的事情：通常只有一个FIFO用于少量（想想一两打）顶点，使用顶点索引作为标记，为最坏情况下的输出属性数量提供足够的空间。如前所述，相当简单的东西。

然后出现了统一着色器。如果统一两种以前不同的着色器，则设计必然会有所妥协。因此，一方面，你有顶点着色器，在正常使用的情况下，它（当时）一帧最多可以触及100万个顶点。另一方面，你有像素着色器，1920×1200的像素着色器需要每帧至少触摸230万像素才能填满整个屏幕一次——如果你想渲染任何有趣的东西，则需要更多。那么猜猜这两个单位中的哪一个最终拉了短管？

好的，这里的交易是：取代了旧的顶点着色器单元，它一次着色多个或少个顶点，你现在有了一个统一的着色器单元，它是为最大吞吐量而设计的，而不是延迟，因此需要大量的工作（有多大？现在，幻数似乎在一批着色的16到64个顶点之间）。

因此，如果您不希望着色效率低下，您需要16-64个顶点缓存未命中，直到您可以分配一个顶点着色负载。但是，整个FIFO并不能真正实现将顶点缓存未命中进行批处理并一次着色的想法。问题是：如果您一次着色一整批顶点，这意味着您需要只有当所有顶点都完成着色后，才真正开始组装三角形。此时，您刚刚添加了一整批三角形（在这里和下面我们只说32个）到FIFO末端的顶点数，这意味着32个旧顶点现在掉出来了–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次中的一个三角形的顶点缓存命中！哦，这不起作用。显然，我们不能将FIFO中最老的32个顶点作为顶点缓存命中数，因为到那时我们想引用它们，它们会消失的！还有，我们想让这个FIFO有多大？如果我们批量着色32个顶点，它需要至少有32个条目大，但是因为我们不能使用32个最旧的条目（因为我们要将它们移出），这意味着我们将在每个批处理上有效地从一个空的FIFO开始。所以，让它变大，比如说64个条目？这相当大。请注意，每个顶点缓存查找都涉及到比较标记（顶点索引）针对FIFO中的所有标记–这是完全并行的，但也是一个耗电的问题；我们在这里有效地实现了一个完全关联的缓存。另外，在分配32个顶点的着色负载和接收结果之间，我们该怎么做–等等？这个着色将需要几百个周期，等待似乎是一个愚蠢的想法！可能有两个sh在飞行中以并行方式加载？但现在我们的FIFO需要至少64个条目长，并且我们不能将最后64个条目计算为顶点缓存命中，因为当我们收到结果时，它们将被移出。此外，一个FIFO与大量着色器核的对比？阿姆达尔的法则仍然适用–将一个严格串行的组件放入另一个管道中完全并行是使其成为瓶颈的可靠方法。

这整个FIFO的东西真的不能很好地适应这种环境，所以，好吧，把它扔掉。回到绘图板上。我们到底想做什么？得到一批大小合适的顶点进行着色，而不是比必要的更频繁地着色顶点。

因此，保持简单：为32个顶点保留足够的缓冲区空间（=1批），并为32个条目保留类似的缓存标记空间，即所有项无效。对于索引缓冲区中的每个基元，对所有索引进行查找；如果在缓存中命中，则可以。如果未命中，则在当前批中分配一个插槽，并将新索引添加到缓存标记数组中。一旦我们没有足够的空间添加新基元，则将整个批分配给vertex shading、 保存缓存标记数组（即我们刚才着色的顶点的32个索引），然后再次从空缓存开始设置下一批，确保批完全独立。

每个批处理将使着色器单元忙碌一段时间（可能至少几百个周期！）.但这没问题，因为我们有很多这样的组件–只需选择一个不同的单元来执行每个批！Presto parallelism。我们最终会得到结果。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发送到管道中的原语（这就是所谓的“原语组装”）是的，我将在后面部分介绍）。

顺便问一下，当我说“返回结果”时，这意味着什么？它们最终会出现在哪里？有两个主要选择：1.专用缓冲区或2.一些通用缓存/草稿行内存。它过去是1），围绕顶点数据设计了一个固定的组织（每个顶点有16个浮点4个属性向量的空间等等），但最近GPU似乎正朝着2）的方向发展，即“记忆”。它更灵活，并且具有明显的优势，您可以将此内存用于其他着色器阶段，而专门的顶点缓存对于像素着色或计算管道来说是相当无用的，仅举一个例子。

**更新**：这是迄今为止描述的顶点着色数据流的图片。

## 着色器单元内部

简短版本：这与您在查看已反汇编的HLSL编译器输出时所期望的差不多（fxc/dumpbin是您的朋友！）。你猜怎么着，只是处理器真的很擅长运行这种代码，而且这种事情在硬件中的实现方式是构建某种东西，它在精神上相当接近着色器字节码。与我到目前为止所谈论的内容不同，它也有很好的文档记录——如果您感兴趣，请查看AMD和NVidia的会议演示，或者阅读CUDA/Stream SDK的文档。

总之，这里是执行摘要：fast ALU主要围绕一个FMAC（浮动多重累积）单元构建，一些硬件支持（至少）倒数、倒数平方根、log2、exp2、sin、cos，针对高吞吐量和高密度而不是低延迟进行优化，运行大量线程以覆盖所述延迟，每个线程有相当少的寄存器（因为运行的寄存器太多了！），非常擅长执行直线代码，不擅长分支（特别是在它们不一致的情况下）。

这是几乎所有实现的共同点。也有一些不同之处；AMD硬件过去一直坚持使用HLSL/GLSL和着色器字节码所暗示的4宽SIMD（尽管他们最近似乎正在远离这一点），而NVidia不久前决定将4路SIMD转换为标量指令。再说一遍，所有这些都已经在网上了！

但值得注意的是不同着色器阶段之间的差异。简短的说法是，他们中确实有相当少的人；例如，所有的算术和逻辑指令在所有阶段都完全相同。某些构造（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；但最主要的区别在于传入和传出的数据类型（和格式）。

有一个与着色器相关的特殊点，尽管这是一个足够大的主题，值得单独扮演一个角色。该位是纹理采样（和纹理单位）。事实证明，这将是我们下一次的话题！到时候见。

## 闭幕词

再一次，我重复我在“顶点缓存和着色”部分的免责声明：这一部分是我的猜测，所以请恕我直言。或者一磅。我不知道。

我也不会详细介绍如何管理刮擦/缓存内存；缓冲区大小（主要）取决于所处理批次的大小和所需顶点输出属性的数量。缓冲区大小和管理对于性能来说确实很重要，但我不能在这里有意义地解释它，我也不想这样解释；虽然很有趣，但这些东西对于您所谈论的任何硬件都是非常具体的，并且不是很有洞察力。

○ fgiesenpermalink

“因此，我认为顶点缓存发生变化的原因之一不仅是因为着色器的大规模并行性，还因为基本体组装成了瓶颈，1个tri/时钟不再足够。当然，我可能是错的或多余的。”

是的，这就是我用“FIFO是管道的一个串行部分”这一点暗示的；基本上PA需要在某个地方有所有原语的顶点数据，如果那是你的FIFO，那么你只能有一个PA（因为只有一个FIFO）！理论上，可以有多个FIFO，但最终会分别为每个FIFO重新着色顶点，这与顶点缓存的点完全无关。所以，是的，只需阻止它，并在下游大量分派/工作。总体而言，这更容易（也更具可扩展性！）。

“透明的变换后顶点缓存非常酷，但我一直认为，放弃关联查找并使用显式样式将是一个巨大的成功”

所以基本上是两级索引方案；我不确定。总的来说，这无疑是一种更有效的方式来驱动管道，但接口兼容性是一件大事。不管他们做什么，他们仍然需要有效地支持常规索引原语，因为到目前为止每个人都在使用它。在这一点上，在改变这一点上存在着巨大的摩擦；如果这成为一个重大问题，我们最终可能会达到目的，但目前情况似乎并非如此。

另外请注意，例如，使用16宽的着色器单元，您希望拍摄的批次至少与16大，但显然您也可以拍摄更宽的倍数：32、48、64等等。这可以提供更好的命中率（我描述的整个设置在每个块中从头开始重新启动！），但也需要更多的内存进行缓冲，并且具有更高的粒度。如果使用完全关联缓存，则最大顶点着色批处理的大小将有一个固定限制（即该缓存的大小）。但是，使其扩展得更广的更好方法是不完全关联，而是使缓存集关联；现在每个顶点索引在缓存中都有几个位置。如果遇到一个索引将进入一个已满的集合，则无法再将其添加到当前块中，因此必须分派可能是部分块的内容。请注意，对于三角形，您的缓存需要至少具有3路关联性，这样才能取得进展–tri中的所有顶点索引可能映射到同一集合！但我认为目前没有人这样做；首先，非常宽的批处理需要大量的输出缓冲，并且缓冲区空间有限，因此较大的批处理意味着您可以并行运行较少的批处理。如果并行运行的数量太少，则无法覆盖延迟并最终等待。更重要的是，我在这里谈论的是D3D11；最坏情况下需要支持的基本体是32控制点面片，为了组装单个基本体，它绝对需要至少一个32顶点批！

（您需要始终能够组装至少一个基本体，否则可能会被无限期卡住）。这也意味着在集合关联模型中至少需要一个32路关联缓存！“也许这对移动设备（许多实现没有真正的转换后缓存，可能是因为电源问题，仅依赖条带）比台式机更有意义。”

这条带子基本上是免费的。在PA中，无论您做什么，都需要至少3个顶点（单个三角形）的缓冲区。但一旦你缓冲了最后3个顶点，你就可以免费得到有效的三角形条带。另一方面，适当的变换后顶点缓存会占用面积、功耗和设计工作量。如果您不太关心三角形吞吐量，那么它就不值得。

## 2011年图形管道之旅，第4部分

2012年9月22日

0:03

### 2011年图形管道之旅，第4部分

2011年7月4日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

欢迎回来。最后一部分是关于顶点着色器的，一般都会涉及GPU着色器单元。大多数情况下，它们只是向量处理器，但它们可以访问其他向量体系结构中不存在的一种资源：纹理采样器。它们是GPU管道中不可分割的一部分，非常复杂（而且有趣！）足以保证他们自己的文章，所以就这样。

### 纹理状态

在开始实际的纹理操作之前，让我们先看看驱动纹理的API状态。在D3D11部分中，它由3个不同的部分组成：

1.    采样器状态。过滤模式，寻址模式，最大各向异性，诸如此类。这控制纹理采样的一般方式。

2.    底层纹理资源。这归结为指向内存中原始纹理位的指针。该资源还确定它是单个纹理还是纹理数组，以及纹理的多采样格式（如果有），以及纹理位的物理布局–即，在资源级别，尚未决定如何准确解释内存中的值，但它们的内存布局已确定。

3.    着色器资源视图（简称SRV）。这决定了采样器如何解释纹理位。在D3D10+中，资源视图链接到基础资源，因此您永远不会明确指定资源。

大多数情况下，您将创建一个具有给定格式的纹理资源，比如说RGBA，每个组件8位，然后只创建一个匹配的SRV。但是您也可以创建一个“每个组件8位，无类型”的纹理然后对同一资源使用多个不同的SRV，以不同的格式读取基础数据，例如，一次作为UNORM8\_SRGB（SRGB空间中映射为浮点0..1的无符号8位值），一次作为UINT8（无符号8位整数）。

创建额外的SRV一开始似乎是一个恼人的额外步骤，但关键是这允许

API运行时在SRV创建时执行所有类型检查；如果您得到了一个有效的SRV，这意味着SRV和资源格式是兼容的，并且在SRV存在时不需要进行进一步的类型检查。换句话说，这里的一切都是关于API效率的。

无论如何，在硬件层面，这归结为与纹理采样操作相关联的一包状态——采样器状态、要使用的纹理/格式等——需要保留在某个地方（有关在流水线体系结构中管理状态的各种方法的说明，请参阅第2部分）。同样，也有各种方法，从“每次状态改变时管道刷新”到“在采样器中完全无状态，并随每个纹理请求发送完整集”，中间有各种选项。您无需担心——这是硬件架构师快速进行成本效益分析，模拟一些工作负载，然后采用前面给出的任何方法——但值得重复的是：作为PC程序员，不要假设硬件遵循任何特定模型。

不要认为纹理开关很昂贵——它们可能完全由无状态纹理采样器管道连接，因此基本上是免费的。但也不要假设它们是完全自由的——可能它们没有完全管道化，或者在任何给定时间管道中不同纹理状态集的最大数量都有上限。除非你在一个有固定硬件的控制台上（或者你针对每一代图形硬件都对你的引擎进行了优化），否则就没有办法说了。因此，在进行优化时，要做一些显而易见的事情——尽可能按材质排序，以避免不必要的状态更改等——这当然至少可以为您节省一些API工作，然后就不做了。不要根据硬件正在做的任何特定模型做任何花哨的事情，因为它可以（也将！）在硬件的不同代之间在眨眼之间改变。

### 纹理请求的剖析

那么，我们需要随纹理样本请求发送多少信息？这取决于纹理类型和我们使用的采样指令。现在，让我们假设一个2D纹理。如果我们想使用（比如）多达4倍的各向异性采样进行2D纹理采样，我们需要发送哪些信息？

各向异性采样？

§  2D纹理坐标–2个浮动，按照本系列中的D3D术语，我将它们称为u/v，而不是s/t。

§  u和v沿屏幕“x”方向的偏导数：。

§  同样，我们也需要y方向的偏导数：。

所以，对于一个相当普通的2D采样请求（SampleGradvariety），这是6个浮点数——可能比您想象的要多。这4个梯度值既用于mipmap选择，也用于选择各向异性过滤内核的大小和形状。您还可以使用纹理采样指令，明确指定mipmap级别（在HLSL中，即SampleLevel）——这些不需要渐变，只需要包含LOD参数的单个值，但它们也不能进行各向异性过滤——最好的效果是三线性！不管怎样，让我们和那6辆车呆一会儿。这看起来真是太多了。我们真的需要将它们与每个纹理请求一起发送吗？

答案是：视情况而定。在除像素着色器之外的所有着色器中，答案都是肯定的，我们确实必须这样做（如果我们想要各向异性过滤的话）。在像素着色器中，我们没有；有一个技巧可以让像素着色器给你梯度指令（你可以计算一些值，然后问硬件“这个值的屏幕空间梯度近似值是多少？”），纹理采样器也可以使用同样的技巧从坐标中获得所有需要的偏导数。因此，对于PS 2D“sample”指令，您只需要发送2个坐标，这意味着其余坐标，前提是您愿意在采样器单元中进行更多的数学运算。

只是为了好玩：最坏情况下，单个纹理样本需要多少参数？在当前的D3D11管道中，它是Cubemap阵列上的SampleGrad。让我们看看统计数字：

§  3D纹理坐标–u、v、w:3个浮动。

§  Cubemap数组索引：一个int（让我们在这里以与float相同的成本计算）。

§  屏幕x和y方向（u、v、w）的梯度：6个浮动。

对于每个采样像素总共10个值，如果您真的这样存储它，那就是40字节。现在，您可能会决定不需要全部32位（对于数组索引和渐变来说，这可能有点过头了），但仍然需要发送大量数据。

事实上，让我们看看我们在这里讨论的带宽是什么。让我们假设我们的大多数纹理都是2D（带有一些立方体贴图），我们的大多数纹理采样请求来自像素着色器，顶点着色器中几乎没有纹理采样，并且常规采样类型请求最频繁，其次是SampleLevel（所有这些对于你在游戏中看到的实际渲染来说都是非常典型的）。这意味着每个像素发送的32位浮点值的平均数量将介于2（u+v）和3（u+v+w/u+v+lod）之间，比如说2.5或10字节。

假设中等分辨率–比如说1280×720，大约92万像素。你的平均游戏像素着色器有多少纹理样本？我想至少有3个。假设我们有适度的透支量，所以在3D渲染阶段，我们在屏幕上触摸每个像素大约两次。然后我们用一些纹理起伏来完成它y全屏传递进行后处理。考虑到某些后处理将以较低的分辨率进行，这可能会使每个像素至少再增加6个样本。加起来，我们得到0.92\*（3\*2+6）=每帧约1100万个纹理样本，每秒30帧，约3.3亿个。每请求10字节，仅纹理请求有效负载为3.3 GB/s。下限，因为涉及到一些额外的开销（我们将在一秒钟内实现）。请注意，我在所有这些数字中“咳嗽”都有“一点”错误：）。在一张好的DX11卡上，一款真正的现代游戏将以更高的分辨率运行，具有比我列出的更复杂的着色器，可比的透支量甚至更少（将着色/照明延迟到救援！），更高的帧速率，以及更复杂的后处理方式–继续，做一个快速的信封背面计算多少纹理请求带宽一个体面的质量SSAO通过在四分之一分辨率与双边上采样需要…

重点是，这整个纹理带宽的东西不是你可以用手挥动的东西。纹理采样器不是着色器核心的一部分，它们在芯片上的某个距离上是独立的单元，每秒洗牌数千兆字节并不仅仅是它自己发生的事情。这是一个实际的体系结构问题——幸好我们没有在Cubemap阵列上使用SampleGrad来处理所有问题：）

**但是谁要求一个纹理样本呢？**

答案当然是：没有人。我们的纹理请求来自着色器单元，我们知道它一次处理16到64个像素/顶点/控制点/…之间的某个位置。因此，我们的着色器不会发送单个纹理样本，而是一次发送一组纹理样本。这一次，我将使用16作为数字，因为我上次选择的32是非平方的，这看起来很简单

使用16作为数字——这仅仅是因为我上次选择的32是非正方形的，这在谈论2D纹理请求时看起来很奇怪。因此，一次16个纹理请求–构建该纹理请求负载，在开始时添加一些命令字段以便采样器知道要做什么，添加更多字段以便采样器知道要使用哪个纹理和采样器状态（同样，请参见上面关于状态的备注），并将其发送到某个纹理采样器。

这需要一段时间。

不，说真的。纹理采样器的管道非常长（我们很快就会知道原因）；纹理采样操作花费的时间太长，着色器单元无法一直处于空闲状态。再说一遍，跟我说：吞吐量。因此，在纹理样本上，着色器单元只会悄悄地切换到另一个线程/批处理并执行其他工作，然后在结果出现时切换回。只要着色器单元有足够的独立工作，就可以正常工作！

### 一旦纹理坐标到达…

首先要做一系列计算：（在这里和下面，我假设一个简单的双线性样本；三线性和各向异性需要更多的工作，见下文）。

§  如果这是采样或SampleBias类型请求，请首先计算纹理坐标梯度。

§  如果未给出明确的mip水平，则计算要从梯度中采样的mip水平，并添加LOD偏差（如果指定）。

§  对于每个生成的采样位置，应用地址模式（包裹/夹紧/镜像等），以在归一化的[0,1]坐标中获得要采样的纹理中的正确位置。

§  如果这是立方体贴图，我们还需要确定从哪个立方体面采样（基于u/v/w坐标的绝对值和符号），并进行除法以将坐标投影到单位立方体上，使其处于[-1,1]间隔。我们还需要删除3个坐标中的一个（基于立方体面），并缩放/偏移其他2个坐标，以便它们位于与常规纹理样本相同的[0,1]标准化坐标空间中。

§  接下来，取[0,1]标准化坐标并将其转换为定点像素坐标进行采样–我们需要一些分数位用于双线性插值。

§  最后，根据整数x/y/z和纹理数组索引，我们现在可以计算从中读取纹理的地址。嘿，在这一点上，朋友之间的几倍和几倍是什么？

如果你认为这听起来很糟糕，那么让我来提醒你，这是一个简化的观点。上面的总结甚至没有涵盖有趣的问题，例如纹理边界或采样立方体贴图边/角。相信我，现在听起来可能很糟糕，但如果你真的为这里需要发生的一切编写代码，你肯定会感到震惊。幸好我们有专门的硬件为我们做这件事。：）无论如何，我们现在有了一个内存地址来获取数据。只要有内存地址，附近就有一两个缓存。

### 纹理缓存

现在似乎每个人都在使用两级纹理缓存。第二级缓存是一个完全bog标准的缓存，发生在包含纹理数据的缓存内存中。第一级缓存不是很标准，因为它有额外的智能。它也比你预期的要小——每个采样器大约4-8kb。让我们先讨论一下尺寸，因为它往往会让大多数人感到惊讶。

事情是这样的：大多数纹理采样是在启用mip映射的像素着色器中完成的，并且专门选择了用于采样的mip级别，以使屏幕像素：纹理比例大约为1:1–这就是全部要点。但这意味着，除非您一次又一次地命中纹理中完全相同的位置，否则每个纹理采样操作平均将丢失约1 texel–双线性过滤的实际测量值约为1.25 misses/request（如果您单独跟踪像素）。即使更改纹理缓存大小，该值也会在很长一段时间内或多或少保持不变，然后在纹理缓存足够大以包含整个纹理时（通常在几百KB到几MB之间，一级缓存的大小完全不现实）该值会急剧下降。

关键是，任何纹理缓存都是一个巨大的胜利（因为它将每个双线性采样的内存访问次数从大约4次降至1.25次）。但与CPU或着色器内核共享内存不同的是，从4k缓存到16k缓存几乎没有什么好处；不管发生什么，我们都会通过缓存传输更大的纹理数据。

第二点：由于未命中率/样本平均值为1.25，纹理采样器管道需要足够长，以便在不暂停的情况下支持每个样本从内存中的完全读取。让我换一种说法：足够支持每个样本的完整内存读取，而不会延迟。让我换一种说法：纹理采样器管道足够长，即使需要400-800个周期，也不会暂停内存读取。这是一条非常长的管道——从字面意义上讲，它实际上是一条管道，将数据从一个管道寄存器传递到下一个管道寄存器几百个周期，在内存读取完成之前不进行任何处理。

所以，一级缓存小，管道长。“额外的智能”呢？嗯，有压缩纹理格式。你在PC上看到的那些——S3TC又名DXTC又名BC1-3，然后是D3D10引入的BC4和5，它们只是DXT的变体，最后是D3D11引入的BC6H和7——都是基于块的方法，分别对4×4像素的块进行编码。如果在纹理采样期间对其进行解码，则意味着您需要能够在每个周期解码最多4个这样的块（如果您的4个双线性采样点恰好位于跨越4个块的最坏配置中），并从每个块中获取单个像素。坦率地说，这太糟糕了。因此，取而代之的是，当4×4块被带进一级缓存时，它被解码：在BC3（又名DXT5）的情况下，您从纹理L2中获取一个128位的块，然后在纹理缓存中将其解码为16个像素。突然之间，你不再需要对每个样本进行多达4个块的部分解码，你现在只需要对每个样本解码1.25/（4\*4）=大约0.08个块，至少如果你的纹理访问模式足够连贯，能够击中你解码的其他15个像素，以及你实际要求的像素：）。即使在它再次退出L1之前，您只使用了它的一部分，这仍然是一个巨大的改进。这种技术也不限于DXT块；您可以在缓存填充路径中处理D3D11所需的>50种不同纹理格式之间的大部分差异，其命中频率约为实际像素读取路径的三分之一–很好。例如，可以通过将sRGB像素转换为纹理缓存中的16位整数/通道（或16位浮点/通道，如果需要，甚至32位浮点）来处理UNORM sRGB纹理之类的事情。然后，过滤在线性空间中正确地对其进行操作。请注意，这最终会增加一级缓存中texel的足迹，因此您可能希望增加一级纹理大小；不是因为需要缓存更多的texel，而是因为缓存的texel更胖。和往常一样，这是一种权衡。

### 过滤

在这一点上，实际的双线性滤波过程相当简单。从纹理缓存中获取4个采样，使用分数位置在它们之间进行混合。这是我们常用的备用设备，乘法累加装置的几个。（实际上还有很多–我们同时为4个频道进行此操作…）

三线性滤波？两个双线性采样和另一个线性插值。只需在堆中添加更多的多重累积。

各向异性滤波？现在，这实际上需要在管道的前面做一些额外的工作，大约在我们最初计算要采样的mip级别的点上。我们所做的是查看梯度，不仅确定面积，还确定texel空间中屏幕像素的形状；如果它大致和它的高度一样宽，我们只需要做一个常规的双线性/三线性样本，但是如果它在一个方向上拉长，我们会在这条线上做几个样本，然后将结果混合在一起。这会生成多个样本位置，因此我们最终会在整个双线性/三线性管道中循环数次，而样本的实际放置方式及其相对权重的计算对于每个硬件供应商来说都是一个严格保密的秘密；他们多年来一直在解决这个问题，到目前为止，他们都以合理的硬件成本开发出了非常好的产品。我不会猜测他们在做什么；说实话，作为一名图形程序员，你不需要关心底层的各向异性过滤算法，只要它没有被破坏，不会产生可怕的伪影或可怕的减速。

无论如何，除了循环所需样本的设置和排序逻辑之外，这不会给管道增加大量计算。在这一点上，我们有足够多的累加单元来计算各向异性滤波中涉及的加权和，而不需要在实际滤波阶段使用大量额外的硬件。：）

### 纹理返回

现在我们几乎到了纹理采样器管道的末端。这一切的结果是什么？每个请求的纹理样本最多4个值（r、g、b、a）。与纹理请求不同的是，在纹理请求中，请求的大小会有很大的变化，到目前为止，最常见的情况是着色器消耗了所有4个请求

价值观请注意，从带宽的角度来看，发送4个浮点数并不是什么可以忽视的事情，而且在某些情况下，您可能还想刮胡子。如果着色器正在采样32位浮点/通道纹理，则最好返回32位浮点，但如果它正在读取8位未格式化的SRGB纹理，则32位返回是多余的，可以通过在返回路径上使用较小的格式来节省带宽。

就是这样–着色器单元现在有了它的纹理采样结果，并且可以继续处理您提交的批次，该批次将结束本部分。在下一期中，当我

您提交的批次–结束本部分。在下一期中，我们将再次见面，届时我将讨论在我们真正开始光栅化原语之前需要完成的工作。更新：这里有一张纹理采样管道的图片，包括一个有趣的错误，我像专业人士一样在帖子中纠正了这个错误！

### 通常的后期脚本

这一次，没有重大的免责声明。我在带宽示例中提到的数字实际上是当场编造的，因为我无法查到当前游戏的一些实际数字：），但除此之外，我在这里描述的内容应该与您的GPU上现在的内容非常接近，尽管我在过滤等方面用手挥过了一些角落案例（主要是因为细节令人恶心，而非启发性）。

至于包含解压纹理数据的纹理一级缓存，据我所知，这对于当前硬件来说是准确的。一些较旧的硬件即使在一级纹理缓存中也会保持某些格式的压缩，但这是因为“对于大范围的缓存大小，1.25次未命中/采样”规则，这不是一个很大的胜利，可能不值得这么复杂。我认为这些东西现在都消失了。

一个有趣的部分是嵌入式/电源优化的图形芯片，例如PowerVR；在本系列中，我将不详细介绍各种芯片，因为我在这里的重点是您在PC中看到的高性能部件，但如果您感兴趣，我在前面部分的评论中有一些关于它们的注释。

无论如何，PVR芯片有自己的纹理压缩格式，它不是基于块的，并且与过滤硬件紧密集成，因此我假设它们即使在一级纹理缓存中也会保持纹理压缩（实际上，我不知道它们是否有第二级缓存！）。这是一种有趣的方法，就每个区域所做的有用工作和所消耗的能量而言，它可能处于一个相当有利的位置。但我认为“退回到一级缓存”方法总体上提供了更高的吞吐量，而且正如我经常提到的，这都与高端PC GPU的吞吐量有关：）

## 2011年图形管道之旅，第5部分

2012年9月22日

0:05

### 2011年图形管道之旅，第5部分

2011年7月5日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

在上一篇关于纹理采样器的文章之后，我们现在回到3D前端。我们已经完成了顶点着色，现在我们可以开始实际渲染了，对吗？嗯，不完全是。你看，在我们真正开始光栅化基本体之前，还有很多事情要做。事实上，我们不会在这篇文章中看到任何光栅化——这将不得不等到下一次。

### 基本装配

当我们离开顶点管道时，我们刚刚从着色器单元中得到一块着色顶点，并隐式承诺该块包含整数个基本体–即，我们不允许三角形、直线或面片在多个块之间分割。这一点很重要，因为这意味着我们可以真正独立地处理每个块，并且不需要缓冲多个着色器输出块–当然，我们可以，但我们不必。

下一步是组装属于单个基本体的所有顶点（因此称为“基本体组装”）。如果该基本体恰好是一个点，则只读取一个顶点并将其传递。如果是直线，则读取两个顶点。如果是三角形，三个。对于控制点数量较多的面片，依此类推。

简而言之，这里发生的一切就是我们聚集顶点。我们可以通过读取原始索引缓冲区并保存顶点索引->缓存位置贴图的副本（如我所述）来实现这一点，或者我们可以将完全展开的基本体的索引与着色顶点数据一起存储，这可能会为输出缓冲区占用更多的空间，但这意味着我们不必在这里再次读取索引。两种方法都可以。

现在我们已经展开了组成基本体的所有顶点。换句话说，我们现在有了完整的三角形，而不仅仅是一堆顶点。那么我们可以把它们光栅化了吗？不完全是。

### 视口剔除和剪裁

哦，是的，那个。是的，我想我们最好先这么做，嗯？这是管道的一部分，它真正做到了您所期望的，与您所期望的几乎一样（即，文档中解释的方式）。所以我不打算在这里一般地解释多边形裁剪，你可以在任何计算机图形学教科书中找到它，尽管大多数都把它弄得一团糟；如果你想得到一个好的解释，可以使用Jim Blinn的（本书第13章），尽管现在你可能想传递他的备选[0，w]剪辑空间，以避免混淆。

不管怎样，剪辑。简短的版本是：顶点着色器返回齐次剪辑空间上的顶点位置。选择剪辑空间，使描述视锥体的方程尽可能简单；就D3D而言，它们是



,



,



，及



; 注意，最后一个方程实际上排除了齐次点（0,0,0,0），这是一种退化情况。

我们首先需要确定三角形是否部分甚至完全位于这些裁剪平面之外。这可以通过使用Cohen Sutherland风格的代码非常有效地完成。您可以为每个顶点计算剪裁代码（或仅计算剪裁代码）（例如，这可以在顶点着色时完成，并与位置一起存储）。然后，对于每个基本体，剪辑代码的按位AND将告诉您所有的视图平截头体平面

您可以查看基本体中所有顶点都位于错误一侧的所有视锥平面（如果有，这意味着基本体完全位于视锥之外，并且可能会被丢弃），剪辑代码的位或将告诉您需要对基本体进行剪辑的平面。考虑到clipcodes，所有这些都只是几门硬件——简单的东西。

此外，着色器还可以生成一组“消隐距离”（如果所有顶点的任何一个消隐距离小于零，则三角形将被丢弃）和一组“剪裁距离”（定义其他剪裁平面）。原始拒绝/剪辑测试也会考虑这些因素。

如果调用，实际的剪裁过程可以采用两种形式之一：我们可以使用实际的多边形剪裁算法（添加额外的顶点和三角形），也可以将剪裁作为额外的边方程添加到光栅化器中（如果你觉得这听起来像胡言乱语，那就等到下一部分我解释光栅化的时候——它最终会问你是否有意义）。后者更优雅，根本不需要实际的多边形裁剪器，但我们需要能够将所有规范化的32位浮点值作为有效的顶点坐标来处理；可能有一个技巧可以构建一个快速的HW光栅化器来实现这一点，但至少说起来似乎很棘手。所以我假设有一个实际的裁剪器，带有这涉及到（生成额外的三角形等）。这是一个讨厌的问题，但它也很少发生（比你想象的要多，我马上就会谈到），所以这没什么大不了的。不确定这是否是特殊的硬件，或者这条路径是否需要一个着色器单元来进行实际的剪裁；这取决于在这个阶段分配一个新的顶点着色负载是否笨拙，一个专用的剪裁单元有多大，以及您需要多少。我不知道这些问题的答案，但至少从性能方面来说，这没什么大不了的：我们并不经常剪辑，因为我们可以使用保护带剪辑。

### 保护带限幅

这个名字有点用词不当；它不是一种奇特的剪辑方式。事实上，它正好相反：一种直接的不剪辑方式。：）

其基本思想非常简单：部分位于左、右、上和下剪裁平面之外的大多数基本体根本不需要剪裁。GPU上的三角形光栅化实际上是通过扫描全屏区域（或者更准确地说，是剪刀形矩形）并询问每个像素：“这个像素被当前的三角形覆盖了吗？”（事实上，它有点复杂，效率更高，但这是一般的想法）。对于完全位于视口中的三角形，这与对于延伸过（例如）右侧和顶部剪裁平面的三角形一样有效。只要我们的三角形覆盖测试是可靠的，我们根本不需要对左，右，顶部和底部平面剪辑！

这个测试通常是用整数算法完成的，精度是固定的。最后，当你移动一个三角形顶点越来越远时，你会得到整数溢出和错误的测试结果。我想我们都同意，光栅化器产生的像素实际上不在三角形内部，至少是非常令人厌恶的行为，应该是非法的！实际上是这样的——这样做的硬件违反了规范。

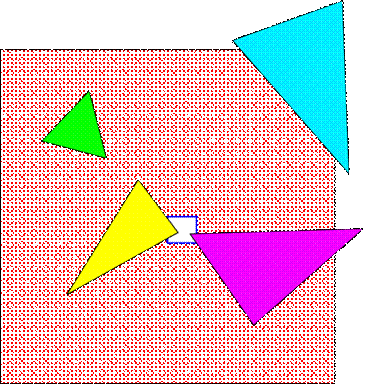
这个问题有两种解决方案：第一种是确保三角形测试永远不会产生错误的结果，不管输入三角形看起来如何。如果你做到了这一点，那么你就永远不需要夹在前面提到的四个平面上。这被称为“无限保护带”，因为保护带实际上是无限的。解决方案二是最终剪裁三角形，就像它们即将超出光栅化器计算不能溢出的安全范围一样。例如，假设光栅化器有足够的内部位来处理具有



,



（注意，我使用大写字母X和Y表示屏幕空间位置；我将坚持这个惯例）。您仍然使用常规视图平面执行视口剔除测试（即“此三角形是否位于视锥台之外”），但实际上仅对选定的保护带剪裁平面进行剪裁，以便在投影和视口变换后，生成的坐标处于安全范围内。我想是时候拍张照片了：



保护带限幅

小的白色矩形，蓝色轮廓大致在中间代表我们的视口，而大鲑鱼周围的区域是我们的保护带。在这张图片中，它看起来像一个小视口，但实际上我选择了一个大视口，这样你就可以看到任何东西！用我们的-32768。。32767 guardband剪辑范围，该视口大约5500像素宽–是的，这里有一些巨大的三角形：）。不管怎样，三角形展示了一些重要的案例。黄色三角形是最常见的情况–延伸到视口之外但不延伸到保护带的三角形。这只是直接通过，无需进一步处理。绿色三角形完全位于保护带内，但不在视口区域内，因此它永远不会到达此处–它已被上面的视口剔除拒绝。蓝色三角形延伸到保护带剪辑区域之外，需要进行剪裁，但它完全位于视口区域之外，并且被视口消隐拒绝。最后，紫色三角形延伸到视口内部和防护带外部，因此实际上需要剪裁。

正如你所看到的，你需要在四边平面上裁剪的三角形是非常极端的。如前所述，这种情况很少发生——别担心。

### 旁白：正确剪裁

所有这些都不应该令人感到非常惊讶；这听起来也不应该太难，至少如果你熟悉算法的话。但魔鬼总是在细节中。下面是一些不明显的规则，三角形裁剪器在实践中必须遵守。如果它违反了这些规则中的任何一条，在某些情况下，它会在共享一条边的相邻三角形之间产生裂缝。这是不允许的。

§  裁剪器必须保留视锥体内的顶点位置（位精确）。

§  在平面上剪切边AB必须产生与在该平面上剪切边BA（方向反转）相同的结果，精确到位。（这可以通过使数学完全对称，或始终在同一方向剪裁边（例如从外到内）来确保）。

§  在多个平面上剪裁的基本体必须始终以相同的顺序在平面上剪裁。（一次对所有平面进行剪切或剪切）

§  如果使用防护带，则必须紧靠防护带平面；不能对某些三角形使用保护带，但如果确实需要剪裁，则可以对原始视口平面进行剪裁。同样，如果不这样做，将导致裂缝——如果我没记错的话，实际上有一块图形

在过去的糟糕日子里，伴随着这个缺陷的硬件被封装在硅中。哎呀。：）

### 那些讨厌的远近飞机

好的，我们有一个非常好的4个侧面平面的快速解决方案，但是近面和远面呢？尤其是近平面是令人烦恼的，因为处理的所有东西都只是稍微超出视口的范围，这就是我们进行大部分剪辑的平面。那么我们能做什么呢？一个z形的保护带？但这是如何工作的呢？我们根本没有沿着z轴光栅化！事实上，这只是我们在三角形上插值的值，该死！

我们在三角形上插值的值，该死！

不过，从好的方面来说，这只是我们在三角形上插值的一些值。事实上，z-近测试(



)一旦你插值Z，这真的很容易——它只是符号位。z-far(



)这是一个额外的比较（不是我在这里使用的Z不是Z，也就是说，这些是“屏幕”或投影后坐标）。不过，我们还是在做每像素的Z比较（Z测试！），所以这并不是一笔很大的额外费用。视情况而定，但以这种方式进行z-clip绝对是一种选择。如果你想支持像NVidias“深度钳”OpenGL扩展这样的东西，你需要能够跳过z-near/z-far剪裁；事实上，我认为这个扩展的存在是一个很好的暗示，表明他们正在这样做，或者至少在一段时间内习惯了这样做。

现在我们来看一个常规的剪辑平面：



. 我们能把这个也处理掉吗？答案是肯定的，使用在齐次坐标系下工作的光栅化算法，例如，此算法。不过，我不确定硬件是否使用了那个。这是一个优雅的好方法，但似乎很难使用该算法严格遵守D3D11光栅化规则。但也许有一些我不知道的很酷的技巧。不管怎样，剪辑就是这样。

### 投影和视口变换

投影只取x、y和z坐标，并将它们除以w（除非您使用的是不实际投影的齐次光栅化器，但我将在下面忽略这种可能性）。这为我们提供了-1和1之间的标准化设备坐标或NDC。然后，我们应用视口变换，将投影的x和y映射到像素坐标（我称之为x和y），并将投影的z映射到范围[0,1]（我称之为该值z），以便在z近平面z=0，在z远平面z=1。

此时，我们还将像素捕捉到亚像素网格上的分数坐标。从D3D11开始，对于三角形坐标，硬件要求具有精确的8位亚像素精度。此捕捉将一些非常薄的碎片（否则会导致问题）转换为退化三角形（根本不需要渲染）。

### 背面和其他三角形剔除

一旦我们有了所有顶点的X和Y，我们就可以使用边向量的交叉来计算有符号三角形的面积。如果区域为负数，则三角形逆时针旋转（此处，负数区域对应于逆时针方向，因为我们现在在像素坐标空间中，而在D3D像素空间中，y向下而不是向上增加，因此符号是反转的）。如果该区域为正，则顺时针旋转。如果它是零，它是退化的，并且不覆盖任何像素，因此可以安全地剔除它。此时，我们知道三角形方向，因此可以执行背面剔除（如果启用）。

就这样！我们现在已经准备好光栅化了…差不多了。实际上我们必须先做三角形的设置。但要做到这一点，需要了解光栅化将如何执行，所以我将推迟到下一部分…到时候见！

### 最后发言

同样，我跳过了一些部分并简化了其他部分，因此这里有一个通常的提示，即现实中的情况有点复杂：例如，我假装您只使用常规的齐次剪裁算法。大多数情况下是这样的–但是可以将某些顶点着色器属性标记为使用屏幕空间线性而不是透视正确插值。现在，常规齐次剪辑总是进行透视校正插值；对于屏幕空间的线性属性，您实际上需要做一些额外的工作，以使其透视图不正确。：）

我有时会谈到原语，但我主要集中在三角形上。点和线并不难，但老实说，它们也不是我们在这里的目的。如果你感兴趣的话，你可以计算出细节。：）

有大量的光栅化算法，其中一些（比如我引用的Olanos 2DH方法）允许您跳过几乎所有的剪辑，但正如我所提到的，D3D11对引用有非常严格的要求）允许您跳过几乎所有的剪辑，但正如我所提到的，D3D11对三角光栅化器有非常严格的要求，因此硬件实现没有太大的回旋余地；我不确定这些方法是否可以调整到完全符合规范的程度（下一次我将介绍很多细微之处）。因此，在这里和下面我假设你不能做超光滑的事情；再说一次，我正在运行的不太圆滑的方法在光栅化器中的每像素数学数稍微少一些，因此它们可能会在硬件实现中获胜。当然，我可能错过了解决所有这些问题的神奇精灵尘埃。这在图形中经常出现，令人惊讶。如果你知道一个很棒的解决方案，请在评论中给我留言！

最后，我在这里描述的三角形剔除是最小值；例如，在光栅化时生成零像素的三角形类比零面积三角形大得多，如果您能够足够快地找到它（或使用足够少的门），您可以立即删除三角形，而无需进行三角形设置。这是最后一点，在进行三角形设置和至少一些光栅化之前，您可以便宜地进行筛选-找到其他方法提前拒绝tris在这里获得丰厚回报。

## 2011年图形管道之旅，第6部分

2012年9月22日

0:06

### 2011年图形管道之旅，第6部分

2011年7月6日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

欢迎回来。这一次我们将看到三角形被光栅化——最后！但在我们可以光栅化三角形之前，我们需要进行三角形设置，在我可以讨论三角形设置之前，我需要解释我们设置的目的；换句话说，让我们讨论硬件友好的三角形光栅化算法。

### 如何不渲染三角形

首先，向那些已经在这个游戏中玩了很长时间并编写了自己的优化软件纹理贴图器的人提醒一下：首先，你可能习惯于认为三角形光栅化器是一个合并的blob，它可以同时做很多事情：跟踪三角形形状，插值u和v坐标（或者，对于透视正确贴图，u/z、v/z和1/z），执行z缓冲区测试（对于透视正确贴图，您可能使用了1/z缓冲区），然后执行实际纹理（加上着色），都在一个大循环中，精心安排，可能会使用所有可用的寄存器。你知道我说的是什么，对吧？是的，忘了这一点。这是硬件。在硬件中，你把东西打包成漂亮整洁的小模块，易于单独设计和测试。在硬件中，“三角光栅器”是一个块，告诉你一个三角形覆盖了什么（亚）像素；在某些情况下，它也会给你三角形内这些像素的重心坐标。但就是这样。没有u或v，甚至不是1/z。当然也没有纹理和着色，通过专用的纹理和着色单元，这应该不会让人感到意外。

第二，如果你已经写了自己的三角形映射器“回到过去”，您可能使用了Chris Hecker的透视纹理贴图系列中所述的增量扫描线光栅化器。这恰好是在没有SIMD单元的处理器上在软件中实现这一点的一个好方法，但它不能很好地映射到具有快速SIMD单元的现代处理器，甚至更糟的是映射到硬件，这并不是说它阻止了人们使用我在努力。特别是，有一个过时的游戏机站在角落里，现在正努力让自己看起来很淡定。那个有三角形光栅的游戏机在屏幕的底部和右侧边缘有很快的保护带剪辑，而在顶部和左侧边缘没有那么快的保护带剪辑（朋友们，这就是我们所说的“告诉”）。只是说说而已。

那么，这种硬件算法有什么不好的地方呢？首先，它真的光栅化了一行一行扫描的三角形。由于在像素着色后会变得明显的原因，我们希望光栅化器以2×2像素的组输出（所谓的“四边形”-不要与“四边形”原语混淆，该原语在管道中的这一阶段已分解为一对三角形）。对于扫描线算法来说，这是非常棘手的，因为我们现在不仅需要并行运行它的两个“实例”，它们还从各自扫描线中三角形覆盖的第一个像素开始，这可能相距很远，并且不会很好地生成我们想要的2×2四边形。它也很难有效地并行化，在x和y方向上是不对称的——这意味着一个8像素宽、100像素的三角形与一个100像素宽、8像素高的三角形在光栅化器的各个部分上的应力非常不同。真的很烦人，因为现在你必须使“x”和“y”步进“循环”同样快，以避免瓶颈-但我们所有的工作都是在“y”步上进行的，“x”中的循环是微不足道的！如前所述，这是一片混乱。

### 更好的方法

Pineda在1988年的一篇论文中提出了一种更简单（且更硬件友好）的栅格化三角形的方法。一般方法可以总结为两句话：到直线的有符号距离可以用2D点积（加上一个加法）计算——就像到平面的有符号距离可以用3D点积（加上一个加法）计算一样。三角形的内部可以定义为所有点的集合，这些点位于所有三条边的正确边上。所以…只要在所有候选像素上循环，测试它们是否真的在三角形内。就这样。这是基本算法。

请注意，当我们向右移动一个像素时，我们将一个像素添加到X，而Y保持不变。我们的边方程的形式是



，a，b，c是每个三角形的常数，所以对于X+1

，a，b，c是每个三角形的常数，所以对于X+1



. 换句话说，一旦在给定的点上有了边缘方程的值，相邻像素的边缘方程的值只需加几次就可以了。还要注意的是，并行化非常简单：假设您想一次光栅化8×8=64像素，就像AMD硬件喜欢做的那样（或者根据第三版实时渲染，至少Xbox 360是这样做的）。好吧，你只需要计算



每个三角形（和边）一次，并将其保存在寄存器中；然后，要光栅化8×8像素块，

您只需计算左上角的3边方程，对我们刚刚计算的常数进行8×8并行加法，然后测试得到的符号位，看看8×8像素中的每一个是在该边的内部还是外部。对3条边这样做，一个8×8的三角形块以真正令人尴尬的并行方式光栅化，没有比一堆整数加法器更复杂的了！顺便说一下，这就是为什么在前一部分中会捕捉到定点网格的原因——所以我们可以在这里使用整数数学。整数加法器比任何浮点数学单元都简单得多。当然，我们可以选择合适的加法器宽度来支持我们想要的视口大小，具有足够的亚像素精度，并且可能还有一个2x-4x因子，因此我们可以得到一个大小适中的保护带。

顺便说一下，这里还有一个棘手的问题，那就是填充规则；您需要有打破平局的规则，以确保对于共享一条边的任何一对三角形，该边附近的任何像素都不会被跳过或光栅化两次。D3D和OpenGL都使用所谓的“左上”填充规则；有关详细信息，请参见相关手册。我在这里不讨论它，只是要注意，使用这种整数光栅化器，它可以归结为在三角形设置过程中从某些边上的常数项中减去1。这就保证了它的防水性，一点也不大惊小怪——与Chris在他的文章中为了使它正常工作而不得不经历的扭曲相比！有时候事情会完美地结合在一起。

但我们有一个问题：我们如何找出要测试的8×8像素块？Pineda提到了两种策略：1）只扫描三角形的整个边界框，或2）一个更聪明的方案，一旦发现不再命中任何三角形样本，就停止“旋转”。好吧，如果你一次测试一个像素就可以了。但我们现在做的是8×8像素！做64个并行加法，最后却发现它们完全没有达到任何像素，这是浪费大量的工作。所以…别那么做！

### 我们需要的是更多的等级制度

我刚才描述的是“精细”光栅化器的功能（即实际输出样本覆盖率的功能）。现在，为了避免在像素级浪费工作，我们要做的是在前面添加另一个光栅化器，它不会将三角形光栅化为像素，而是“平铺”——我们的8×8块（本文作者

麦科马克和麦克纳马拉有一些细节，格林的“覆盖遮罩的分层多边形平铺”也有一些细节，这一点从逻辑上得出了结论）。将边缘方程光栅化为覆盖瓷砖的工作原理与光栅化像素非常相似；我们要做的是计算全瓷砖上边缘方程的上下界；由于边方程是线性的，这样的极值出现在瓷砖的边界上——事实上，它足以在4个角点上循环，并且从边方程中“a”和“b”项的符号，我们可以确定哪个角点。总之，它实际上并不比我们已经讨论过的昂贵多少，并且需要完全相同的机器——一些并行整数加法器。作为奖励，如果我们评估瓷砖一角的边缘方程，我们不妨将其传递给精细光栅化器：它需要每8×8块一个参考值，记得吗？很不错的。

所以我们现在要做的是首先运行一个“粗糙”光栅化器，它告诉我们哪些瓷砖可能被三角形覆盖。这个光栅化器可以做得更小（在这个级别上8×8看起来真的太夸张了！），而且它不需要那么快（因为它只针对每个8×8块运行）。换句话说，在这个级别上，发现空块的成本相对较低。

我们可以进一步思考这个想法，就像格林的论文或迈克·艾布拉什在《拉拉拉比》中描述的那样，并进行完整的分层光栅化。但对于硬件光栅化器，几乎没有意义：它实际上增加了小三角形的工作量（除非您可以跳过层次结构的级别，但这不是设计硬件数据流的方式！），如果您有一个足够大的三角形，可以实际产生显著的光栅化工作，我描述的体系结构应该已经足够快，以比着色器单元消耗像素位置更快的速度生成像素位置。

事实上，这里的实际问题首先不是大三角形；对于几乎任何算法（当然包括扫描线光栅化器），它们都很容易有效地处理。问题是小三角形！即使你有一堆生成0或1个可见像素的小三角形，你仍然需要经过三角形设置（我还没有描述，但我们已经接近了），至少一步粗光栅化，然后至少一步细光栅化8×8块。对于小三角形，很容易获得三角形设置或粗光栅边界。

需要注意的一点是，使用这种算法，碎片（长的、非常薄的三角形）是一个严重的坏消息——您需要遍历成吨的瓷砖，并且每个瓷砖只能获得很少的覆盖像素。所以，他们很慢。尽可能避免它们。

**那么三角形设置做什么呢？**

好了，现在我已经描述了光栅化算法是什么，我们只需要看看我们在整个过程中使用了什么样的每边常数；这正是我们在三角形设置过程中需要设置的。

在我们的案例中，列表如下：

§  所有3条三角形边的边方程–a、b、c。

§  一些派生值，比如我提到的for；请注意，您实际上不会在硬件中存储完整的8×8矩阵，如果您要为其添加其他值，则肯定不会。最好的方法是在HW中，可能只是计算和，使用进位保存加法器（aka 3:2 reducer，我以前写过），将表达式减少为一个和，然后用常规加法器完成。或者类似的东西。

§  用于获取粗光栅化器边缘方程的上/下界的瓷砖的哪个参考角。

§  粗光栅化器（调整后的填充规则）第一个参考点处边缘方程的初始值。

…这就是三角形设置计算的结果。它可以归结为边方程及其初始值的几个大整数倍数，阶跃值的几个小倍数，以及其他一些廉价的组合逻辑。

### 其他光栅化问题和像素输出

到目前为止，有一件事我没有提到，那就是剪刀式直肠。这只是一个屏蔽像素的屏幕对齐矩形；光栅化器不会生成该矩形外的像素。这是相当容易实现的–粗光栅化器可以直接拒绝不完全重叠剪刀矩形的分片，细光栅化器使用“光栅化”剪刀矩形（此处的“光栅化”归结为每行和每列一个整数比较以及一些位and）生成所有覆盖遮罩。简单的东西，继续前进。

另一个问题是多样本抗锯齿。现在的变化是，您必须每像素测试更多样本–从DX11开始，硬件需要支持至少8x MSAA。请注意，每个像素内的采样位置不在规则网格上（这对于近水平或近垂直边缘来说表现不佳），而是分散在多个边缘方向的大范围内，以提供良好的结果。这些不规则的采样位置在扫描线光栅化器中是一个很难处理的问题（不使用它们的另一个原因！），但在Pineda风格的算法中很容易支持：它归结为在三角形设置中计算更多的每边偏移和每像素的多个添加/符号测试，而不仅仅是一个。

对于4x MSAA，您可以在8×8光栅化器中做两件事：您可以将每个样本视为一个不同的“像素”，这意味着您的有效分片大小在MSAA解析后现在是4×4实际屏幕像素，精细光栅栅格中2×2位置的每个块在解析后现在对应一个像素，或者你可以坚持使用8×8的实际像素，只需在其中运行四次。8×8对我来说似乎有点大，所以我假设AMD做的是前者。其他MSAA级别的工作方式类似。

无论如何，我们现在有了一个精细的光栅化器，它为我们提供了8×8块的位置以及每个块中的覆盖掩模。很好，但这只是故事的一半——当前的硬件在运行像素着色器之前也会进行早期Z和分层Z测试（如果可能的话），并且Z处理与实际的光栅化交织在一起。但出于说教的原因，似乎最好将其分开；所以在下一部分中，我将讨论各种类型的Z处理、Z压缩和一些更多的三角形设置–到目前为止，我们刚刚介绍了光栅化的设置，但是还有各种我们想要的Z和像素着色的插值量，它们也需要设置！直到那时。

### 警告

我已经链接了一些光栅化算法，我认为它们代表了各种方法

（它们也恰好都在网络上）。还有很多。我甚至没有试着在这里给你一个关于这个主题的全面介绍；这将是一个（冗长的！）严肃的关于它全面介绍这个主题的帖子；这本身就是一个（冗长的！）严肃的帖子——我担心，经过一段时间后，它会变得相当枯燥。

这篇文章中另一个隐含的假设（我已经多次提到了这一点，但这是提醒您的地方之一）是我们使用的是高端PC硬件；很多部分，特别是在移动/嵌入式范围内，都是所谓的分块渲染器，它将屏幕分为多个分块，并分别渲染每个分块。这些与我在本文中使用的用于光栅化的8×8瓷砖不同。平铺渲染至少需要另一个“超粗”光栅化阶段，该阶段提前运行，并找出每个三角形所覆盖的（大）平铺；这一阶段通常称为“装箱”。平铺渲染器的工作方式不同，设计参数也不同于我在这里描述的“最后排序”体系结构（这是官方名称）。当我使用完D3D11管道（还有很长的路要走！）后，我可能会在平铺渲染器上添加一两个帖子（如果有兴趣的话），但现在我只是忽略了它们，所以请注意，例如，智能手机中经常出现的PowerVR芯片处理这些问题的方式有所不同。

8×8块（其他块大小也有同样的问题）意味着小于某个大小的三角形，或者具有不方便的纵横比的三角形，需要比您想象的多得多的光栅化工作，并且在这个过程中使用率很低。我很想告诉你，有一种神奇的算法，它很容易并行化，对切片之类的东西也很好，但是如果有，我不知道，因为硬件供应商仍然定期提醒我们切片不好，显然他们也不知道。所以就目前而言，这似乎是HW光栅化的一个现实。也许有人最终会想出一个很好的解决办法。

我为粗糙光栅描述的“边缘函数下限”效果很好，但在某些情况下会产生误报（误报是指它要求在不覆盖任何像素的块中进行精细光栅化）。有一些技巧可以减少这种情况，但同样，检测一些罕见的情况比只是光栅化偶尔没有任何像素点亮的精细块要复杂/昂贵。另一个权衡。

最后，光栅化过程中使用的块通常在网格上捕捉（这将有助于在下一部分中变得更清楚）。如果是这样的话，即使是一个仅覆盖2个像素的三角形也可能横跨2个瓷砖，并使您光栅化两个8×8的块。效率更低。

重点是：是的，所有这些都相当简单和优雅，但并不完美，实际三角形的实际光栅化远未接近理论峰值光栅化率（通常假设所有精细块都已完全填充）。记住这一点。

## 2011年图形管道之旅，第7部分

2012年9月22日

0:06

### 2011年图形管道之旅，第7部分

2011年7月8日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

在本期中，我将讨论（早期）Z管道以及它如何与光栅化交互。与上一部分一样，本文不会按照实际的管道顺序进行；再次，我将首先描述底层算法，然后在事实发生后填充管道阶段（以相反的顺序，因为这是最简单的解释方法）。

### 插值

Z是在三角形上插值的，顶点着色器输出的所有属性也是如此。因此，让我花一分钟来解释它是如何工作的。在这一点上，我最初有一节介绍了插值背后的数学是如何推导的，以及透视插值为何以这种方式工作。我为此奋斗了几个小时，因为我试图限制在一到两段（因为这是一个旁白），我现在能说的是，如果我想正确地解释它，我需要更多的空间，至少需要一到两张图片；一张图片可能会说上千字，但一张好的图表需要我花上大约一千字的时间来准备文字，所以从我的角度来看，这不一定是一个胜利：）。无论如何，这是一个探根不管怎么说，我把它添加到我的“与图形相关的东西中，以便在某个时候正确地写出来”。现在，我给你一个执行摘要：

仅在屏幕空间三角形上线性插值属性（颜色、纹理坐标等）不会产生正确的结果（除非插值模式是“无透视”模式之一，在这种情况下忽略我刚才写的内容）。但是，假设我们要插值2D纹理坐标对



.如果你用线性插值，你会得到正确的结果



,



和



在屏幕空间中（这里是从顶点位置开始的均匀剪辑空间），然后是每像素

取



得到w，最后将其他两个插值分数乘以w，得到s和t。实际线性

插值归结为建立一个平面方程，然后插入屏幕空间坐标。如果你正在编写一个软件透视纹理贴图器，那就到此为止。但是如果你要插值两个以上的值，更好的方法是计算（使用透视插值）重心坐标–我们称之为



–对于原始剪辑空间三角形中的当前像素，在此之后，可以使用常规线性插值对实际顶点属性进行插值，而无需将所有内容乘以w。

那么，这会给三角形设置增加多少工作量呢



和



因为三角形需要4个倒数，三角形面积（我们已经为背面计算过了）

剔除！），以及一些减法、乘法和加法。使用重心方法设置插值的顶点属性非常便宜–每个属性两次减法（如果不使用重心，这里会有更多的乘法和加法操作）.跟我来吗？可能不会，除非你以前实现过。对此很抱歉，但如果你不理解，忽略所有这些是相当安全的。

让我们回到这里的原因：我们现在要插值的一个值是Z，因为我们将Z计算为



在顶点级别，作为投影的一部分（参见前一部分），因此它已经被w除以，我们可以

在屏幕空间中线性插值。很好。我们得到的是一个平面方程



我们可以把X和Y插入其中得到一个值。所以，这是我愤怒的挥手的妙语

在最后几段中：在任意给定点插值Z归结为两个乘法相加。（开始了解GPU为什么有快速乘法累加单位？这种东西绝对无处不在！）。

### 早期Z/模板

现在，如果你相信图形API传统上把Z/模板处理放进的地方——就在alpha混合之前，就在像素管道的底部——你可能会有点困惑。为什么我还要在管道中现在讨论Z？我们甚至还没有开始给像素着色！答案很简单：Z和stencil测试拒绝像素。可能大多数都是。你真的，真的不想用复杂的材料对一个细节网格进行完全着色，然后扔掉你刚刚做的95%的工作，因为这个网格恰好大部分隐藏在墙后面。这真是对带宽、处理能力和能量的愚蠢浪费。而且在大多数情况下，这是完全不必要的：大多数着色器不会执行任何会影响Z测试结果或写入Z/模具缓冲区的值的操作。

因此，GPU在可以的情况下实际所做的工作被称为“早期Z”（与后期Z相反，后者实际上处于传统API模型通常显示的管道的后期阶段）。这正是它听起来的样子——在三角形光栅化之后，在开始向着色器发送像素之前，尽早执行Z/模具测试和写入。这样，我们就可以及早发现所有被拒绝的像素，而不会浪费大量的计算时间。但是，我们不能总是这样做：像素着色器可能忽略插值的深度值，而是提供自己的深度以写入Z缓冲区（例如深度精灵）；或者它可能会使用discard、alpha test或alpha to coverage，所有这些都会在像素着色器执行期间“杀死”像素/样本，这意味着我们无法提前更新Z缓冲区或模具缓冲区，因为我们可能会更新稍后在着色器中丢弃的样本的深度值！

所以GPU实际上有两个Z/模具逻辑的副本；一个位于光栅化器的正后方，一个位于像素着色器（执行早期Z）的前面，另一个位于着色器（执行晚期Z）的后面。请注意，原则上，我们仍然可以在早期Z阶段进行深度测试，即使着色器使用了一些samplekilling机制。我们必须小心的只是文字。唯一真正阻止我们进行任何早期Z测试的情况是当我们在像素着色器中写入输出深度时——在这种情况下，早期Z单元根本不需要处理。

传统上，API只是假装这些早期逻辑都不存在；Z/Stencil在原始API模型中处于后期阶段，任何优化（如early-Z）都必须以与该模型100%功能一致的方式进行；i、 e.驾驶员必须检测early-Z何时适用，并且只能在没有明显差异时打开。到目前为止，API已经缩小了这一差距；从DX11开始，着色器可以声明为“force early-Z”，这意味着即使着色器使用对early-Z不一定“安全”的基本体，它们也可以使用完整的early-Z处理运行，并且写入深度的着色器可以声明插值的Z值是保守的（即，仍然可能发生early Z拒绝）。

### Z/stencil写道：全部真相

好的，等等。正如我所描述的，我们现在在管道中有两个部分–早期Z和晚期Z–都可以写入Z/模具缓冲区。对于我们看到的任何给定着色器/渲染状态组合，这将在稳定状态下工作。但实际情况并非如此。实际上，我们每帧渲染几百到几千批，定期切换着色器和渲染状态。大多数着色器都允许早期Z，但有些着色器不允许。从执行早期Z的着色器切换到执行晚期Z的着色器没有问题。但是从Z晚期回到Z早期，如果Z早期写过任何东西的话：Z早期比Z早期在管道中更早——这就是全部要点！因此，我们可以为一个着色器启动early-Z处理，愉快地写入深度缓冲区，而仍然有东西可以为一个着色器启动early-Z处理，愉快地写入深度缓冲区，而我们的旧着色器仍在运行late-Z，并且可能试图同时写入同一位置-经典竞态条件。那么我们如何解决这个问题呢？有很多选择：

§  在帧内从早Z处理到晚Z处理（或至少对同一渲染目标执行一系列操作）后，将一直保持晚Z，直到下一个刷新管道的点。这可以工作，但在不必要地禁用early-Z时可能会浪费大量着色器周期。

§  从late-Z着色器切换到early-Z着色器时触发（像素）管道刷新–也可以工作，也不完全是精细的。这一次，我们不会浪费着色器周期（或内存带宽），而是暂停—没有太大的改进。

§  但在实践中，在两个地方都有Z字只是个坏消息。另一个选择是永远不要在早期Z阶段写入Z；总是在Z后期写Z。注意，如果你这样做的话，你需要小心在Z早期做出保守的Z测试决定！这避免了竞争条件，但意味着早期的Z-test结果可能会过时，因为当前调度像素的Z-write要过一段时间才会发生。

§  使用一个单独的单元为我们跟踪Z写操作，并强制执行正确的顺序；早Z和晚Z都必须通过这个单元。

所有这些方法都有效，并且都有各自的优点和缺点。同样，我不确定当前的硬件在这些情况下会做什么，但我有充分的理由相信它是最后两种选择之一。特别是，我们将在稍后的路上（和管道）遇到一个功能单元，这将是实现最后一个选项的好地方。

但我们仍然在做每像素的测试。我们不能做得更好吗？

### 分层Z/模板

这里的想法是，我们可以再次使用光栅化的瓷砖技巧，并尝试一次Z拒绝整个瓷砖，甚至在我们下降到像素级之前！我们在这里做的是一个严格保守的测试；它可能会告诉我们，当没有像素时，“在这个图块中可能有通过Z/模具测试的像素”，但它永远不会声称所有像素都被拒绝，而事实上它们没有。

这里假设我们使用“less”、“less-equal”或“equal”作为Z-compare模式。然后，我们需要存储为该磁贴写入的最大Z值，每个磁贴。在栅格化三角形时，我们计算活动三角形将写入当前分幅的最小Z值（一个简单的保守近似是取当前分幅四个角处插值Z值的最小值）。如果我们的三角形最小-Z大于当前磁贴存储的最大-Z，则保证三角形完全被遮挡。这意味着我们现在需要跟踪maximum-Z pertile，并在写入新像素时保持该值为最新值——不过，如果该信息不是完全最新的，也没关系；因为我们的Z-test是“较少”的，所以Z缓冲区中的值只会随着时间的推移而变小。如果我们使用一个有点过时的每瓷砖最大值-Z，这只意味着我们的早期拒绝率会比我们可能的稍差；它不会引起任何其他问题。

如果我们使用“更大”、“更大相等”或“相等”Z-测试之一，同样的事情也会发生（最小/最大和比较方向交换）。我们不容易做到的是从一个“较少”的测试变成一个“更大”的测试在帧的中间，因为这将使我们一直在跟踪的信息无用（对于基于较少的测试，我们需要最大的Z每瓦片，对于更大的测试，我们需要最小Z每瓦片）。我们需要在整个深度缓冲区上循环，以重新计算所有分幅的最小/最大值，但GPU实际做的是在执行此操作后关闭Hierarchy-Z（直到下一次清除）。所以：不要那样做。

与我描述的Hierarchy-Z逻辑类似，当前的GPU也有分层模板处理。然而，与Hierarchy-Z不同的是，我没有看到多少关于这一主题的出版文献（也就是说，我没有遇到过它——可能有关于它的论文，但我不知道它们）；作为一名游戏机开发人员，您可以访问低级GPU文档，其中包括对底层算法的描述，但坦率地说，我绝对不喜欢在这里写一些东西，因为我拥有的唯一好的来源是各种GPU文档，这些文档附带了大量的NDA。相反，我只是模糊不清地注意到，有一种神奇的精灵尘埃可以在受控环境下非常有效地进行某些类型的模板测试，让你思考这可能是什么以及它是如何工作的，在不太可能的情况下，你会非常关心这件事——大概是因为你的父亲被一个等级制的模具单位杀死了，而你现在正在收集关于它的弱点的信息以进行报复，或者类似的事情。

### 把它们放在一起

好的，我们现在有了我们需要的所有算法和理论——让我们看看如何将我们的新玩具集与我们已有的连接起来！

玩具和电线与我们已经有了！

首先，我们现在需要为Z/属性插值做一些额外的三角形设置。没有太多的工作要做-更多的三角形设置工作；事情就是这样。之后是粗光栅化，我在前面的部分已经讨论过。

然后是分层Z（我假设这里的样式比较较少）。我们希望在粗光栅化和细光栅化之间运行。首先，我们需要逻辑来计算每个图块的最小Z估计。我们还需要存储每块瓷砖的最大Zs，这不需要精确：只要我们总是四舍五入，我们就可以刮胡子！和往常一样，在使用空间和早期拒绝效率之间存在一个权衡。理论上，你可以将Z-max信息放入常规内存中。实际上，我认为没有人会这样做，因为您希望在没有大量额外延迟的情况下做出分层Z决策。另一种选择是将用于分层Z的专用内存放在芯片上——通常作为SRAM，这种内存也可以用于缓存。对于24位Z，您可能需要每个磁贴10-14位，以便在紧凑编码中存储合理精度的Z-max。假设8×8块，这意味着支持2048×2048分辨率的SRAM少于1MBit（128k）——对我来说，这听起来是一个合理的数量级。请注意，这些东西是固定大小的，并为整个芯片共享；如果进行上下文切换，则会丢失。如果将错误的深度缓冲区分配给该内存，则无法在实际重要的深度缓冲区上使用分层Z，并且会丢失。事情就是这样的。这就是为什么硬件供应商经常告诉您首先创建最重要的渲染目标和深度缓冲区；他们提供的这种类型的内存有限（正如你所看到的，还有更多类似的内存），当它用完时，你就没有运气了。请注意，他们不一定需要做这一切或什么都不做；例如，如果您有一个非常大的深度缓冲区，那么您可能只会在左上角2048×1536像素处获得分层Z，因为这是Zmax内存的容量。这并不理想，但仍然比完全禁用hierarchical-Z要好得多。

顺便说一下，“实时渲染”在这一点上提到“GPU很可能是具有两个以上级别的分层Z缓冲区”。我怀疑这是真的，这与我怀疑他们使用多级层次光栅化器的原因是一样的：添加更多级别会使简单的案例（大三角形）更快，同时为小三角形添加延迟和无用的工作：如果你正在绘制一个适合单个8×8平铺的三角形，任何粗糙的层次级别都是纯粹的开销，因为即使是在8×8级别，您也只需要做一个测试就可以简单地拒绝三角形（或者不拒绝）。再说一次，对于硬件来说，性能问题并没有那么大；只要你不消耗额外的带宽或其他稀缺资源，做比严格必要的更多的计算工作并不是什么大问题，只要它在合理的范围内，

分层模板也存在，并且应该在精细rast之前出现，最有可能与分层Z并行。我们已经确定，这在空气、爱情和魔法精灵尘埃上运行，因此它不需要任何实际硬件，并且其预测可能总是完全正确的。啊哼。继续。

在这之后是精细光栅化，然后是早期Z。对于早期Z，我还需要指出两个更重要的点。

### API命令的复仇

在过去的几部分中，我一直在处理原语提交的顺序。到目前为止，这并不重要；不用于顶点着色，也不用于基本体装配、三角形设置或光栅化。但是Z是不同的。对于“less”或“lessequal”等Z-compare模式，像素到达的顺序非常重要；如果我们搞砸了，我们就有可能改变结果，引入不确定性行为。更重要的是，根据规范，我们可以自由地以任何顺序执行操作，只要它对应用程序不可见；正如我刚才所说，对于Z处理，顺序很重要，所以我们需要确保三角形以正确的顺序到达Z处理（这适用于早Z和晚Z）。

在这种情况下，我们要做的就是回到管道中，寻找一个合理的地点，把事情重新整理好。在我们当前的路径中，最佳候选位置似乎是原始装配；因此，当我们开始从着色顶点块组装基本体时，我们确保严格按照应用程序提交给API的原始顺序组装它们。这意味着如果PA缓冲区保存输出顶点块，我们可能会暂停多一点，但它不是正确的，我们需要等待，并且还不能开始设置原语），但这是正确性的代价。

### 内存带宽与Z压缩

第二个要点是Z/Stencil严重占用带宽。这有两个原因。首先，这是我们为光栅化器生成的所有样本真正运行的一件事（当然，假设Z/Stencil没有关闭）。着色器、混合等都受益于我们所做的早期拒绝；但即使是Zrejected像素也会先读取Z缓冲区（除非它们被分层Z杀死）。这就是它的工作原理。另一个重要原因是，当启用多重采样时，Z/模具缓冲区是每个样本的缓冲区；那么4x MSAA意味着4倍于Z的内存带宽成本？对于一些即使没有MSAA也需要大量内存带宽的东西来说，这是一个严重的坏消息。

所以GPU做的是Z压缩。有各种各样的方法，但总的想法总是一样的：假设三角形大小合理，我们期望很多瓷砖只包含一个或两个三角形。如果发生这种情况，那么我们不存储整个平铺的Z值，而是存储填充该平铺的三角形的平面方程。这个平面方程（希望）比实际的Z数据小。如果没有MSAA，一个瓷砖覆盖8×8个实际像素，因此三角形需要相对较大才能覆盖整个瓷砖；但使用4倍的MSAA，一块瓷砖有效地缩小到4×4像素，覆盖整个瓷砖变得更容易。还有一些扩展可以支持2个三角形等，但是对于大小合理的瓷砖，你不能比2-3个三角形大很多，而且还可以节省带宽：额外的平面方程和覆盖遮罩不是免费的！

无论如何，重点是：这种压缩，当它工作时，是完全无损的，但它不适用于所有的瓷砖。因此，我们需要一些额外的空间来表示瓷砖是否被压缩。我们可以将其存储在常规内存中，但这意味着我们现在需要等待两次完整内存往返延迟来执行Z-read。那太糟糕了。因此，我们再次添加了一些专用SRAM，它允许我们在每个磁贴上存储一些（1-3）位。最简单的说，它只是一个“compressed”或“notcompressed”标志，但您可以选择添加多种压缩模式等等。Z压缩的一个很好的副作用是，它允许我们快速进行Z清除：例如，当清除到Z=1时，我们只需将所有瓷砖设置为“压缩”，并存储恒定Z=1三角形的平面方程。

所有Z压缩的东西，就像纹理采样器中的纹理压缩一样，都可以折叠到内存访问/缓存逻辑中，并且对其他人完全透明。如果您不想将平面方程（或添加插值逻辑）发送到Z内存访问块，它可以从Z数据推断它们，并使用一些整数增量编码方案。这种方法通常需要每个样本额外的比特数才能实现无损重建，但它可以使数据路径更简单，单元之间的接口更好，这正是硬件爱好者所喜欢的。

今天就到此为止！下一步：像素着色和它周围发生的事情。

### 后记

正如我前面所说的，设置插值属性的主题实际上本身就是一篇不错的文章。我现在跳过这一步——也许以后会决定填补这个空白，谁知道呢。

Z处理在3D管道中已经存在很长时间了，对于大多数人来说，这是一个严重的带宽问题，他们对这个问题已经思考了很久，并且有无数的技巧可以使用

用于GPU的“生产质量”Z缓冲，有些大，有些小。再一次，我只是在这里触及表面；我试着把自己限制在对图形程序员有用的方面。这就是为什么我没有花太多时间在分层Z计算或Z压缩之类的细节上；所有这些都是非常具体的硬件细节，在每一代中都会发生轻微的变化，而且最终，大多数情况下，您无法有效地利用这些细节：如果给定的Z压缩方案适合您的场景，那么您可以在其他事情上花费一些内存带宽。如果没有，你打算怎么办？更改几何体和相机位置，以便Zcompression更高效？不太可能。对于硬件设计师来说，这些都是每一代人都需要改进的算法，但对于程序员来说，它们只是生活中需要处理的事实。

这一次，我不会详细介绍内存访问在管道的这一阶段是如何工作的。那是故意的。高通量像素着色和其他逐像素或过采样处理有一个关键点，但这是后期的管道，我们还没有实现。一切都将在适当的时候公布：）

## 2011年图形管道之旅，第8部分

2012年9月22日

0:07

### 2011年图形管道之旅，第8部分

2011年7月10日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

在这一部分中，我将讨论像素处理的前半部分：调度和实际的像素着色。事实上，这正是大多数图形程序员在谈论像素处理时所考虑的；我们将在下一部分中遇到的alpha混合和后期Z阶段似乎只不过是事后思考。在硬件方面，这个故事是一个b更复杂的是，正如我们将看到的，我将像素处理分为两部分是有原因的。但我已经超越了自己。在我们进入这一阶段的时候，像素的坐标（或者，实际上，四边形）要着色，加上相关的覆盖遮罩，从光栅化器/early-Z单元到达–三角形的顺序与应用程序提交的顺序完全相同，正如我上次指出的。我们需要做的是将线性、连续的工作流分到数百个着色器单元，然后一旦结果返回，我们需要使其返回到一个线性内存更新流中。

这是一个fork/join并行性的教科书示例。这一部分涉及fork阶段，我们将在其中进行扩展；下一部分将解释join阶段，我们将数百个流合并回一个流。但首先，我有几句关于光栅化的话要说，因为我刚才告诉过你，只有一个quad流进入这不完全是真的。

### 在光栅化过程中变宽

为我辩护，我告诉你的在相当长的一段时间内都是正确的，但这是管道的一个串行部分，一旦你在一个问题上投入超过300个着色器单元，管道的串行部分就有可能成为瓶颈。因此GPU架构师开始使用多个光栅器；从2010年开始，NVidia使用四个光栅器和AMD uses二。作为旁注，NV演示文稿也有一些关于保持API有序的要求。特别是，您确实需要在光栅化/early-Z之前对原语进行排序，就像我上次提到的那样；在alpha blend之前进行排序（您可能倾向于这样做）是行不通的。

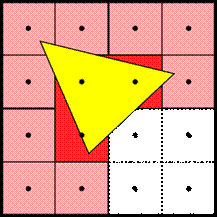
光栅化器之间的功分布基于我们已经看到的early-Z和粗糙光栅化的瓦片。帧缓冲区被划分为平铺大小的区域，每个区域被分配给一个光栅化器。设置后，将参考三角形的边界框，以确定要将哪些三角形移交给哪些光栅化器；大三角形将始终发送到所有光栅化器，但较小的三角形只能命中一个磁贴，并且只发送到拥有它的光栅化器。

此方案的优点在于，它只需要更改功分布和粗光栅化器（遍历瓷砖）；只看到单个瓷砖或四边形（即，从层次Z向下的管道）的所有内容都不需要修改。问题是你现在正在根据屏幕位置划分作业；这可能会导致光栅化器之间出现严重的负载不平衡（想象一下，在一个磁贴中有几百个小三角形），而您实际上对此无能为力。但好的是，向管道添加排序约束的所有内容（Z-test/写入顺序、混合顺序）都附加到特定的帧缓冲区位置，因此屏幕空间细分可以在不破坏API顺序的情况下工作–如果不是这样，平铺渲染器将无法工作。

**你需要走得更远！**

好的，所以我们不只是得到一个四坐标加上覆盖掩模的线性流，而是在2到4之间。我们仍然需要将它们分成数百个着色器单元。是时候换一个调度组了！这首先意味着另一个缓冲区。但是我们发送给着色器的批次有多大？在这里，我再次提到NVidia的数字，仅仅是因为他们在公共白皮书中提到了这个数字；AMD可能也在某处声明了这些信息，但我不熟悉他们的术语，所以我无法直接搜索。无论如何，对于NVidia，分配到着色器单元的单位是32个线程，他们称之为“扭曲”。每个四边形有4个像素（每个像素可以作为一个线程处理），因此对于我们发出的每个着色批处理，我们需要从光栅化器中获取8个传入四边形，然后才能将批处理发送到着色器单元（如果有着色器开关或管道刷新，我们可能会发送更少）。

另外，这是一个很好的解释为什么我们处理的是2×2像素的四边形，而不是单个像素。最大的原因是衍生品。纹理采样器依赖于纹理坐标的屏幕空间导数来进行mip贴图选择和过滤（正如我们在第4部分中看到的）；并且，作为着色器坐标，进行mip贴图选择和过滤（正如我们在第4部分中看到的）；而且，从shader model 3.0及更高版本开始，相同的机制可以以派生指令的形式直接用于像素着色器。在四边形中，每个像素在同一四边形中同时具有水平和垂直相邻；这可用于使用有限差分法（归结为几个减法）估计参数在x和y方向上的导数。这为您提供了一种非常便宜的获取导数的方法，但代价是每次必须对2×2像素的组进行着色。这在大三角形的内部没有问题，但意味着为三角形边生成的四边形的着色工作浪费了25-75%。这是因为四边形中的所有像素，甚至遮罩的像素，都会被着色。这对于为四边形中可见的像素生成正确的导数是必要的。不可见但仍着色的像素称为“辅助像素”。下面是一个小三角形的图示：



三角形与4个四边形相交，但仅在其中3个四边形中生成可见像素。此外，在3个四边形中的每个四边形中，只有一个像素被实际覆盖（每个像素区域的采样点被描绘成黑色圆圈）——被填充的像素被描绘成红色。每个partiallycovered四边形中的其余像素是辅助像素，并使用较浅的颜色绘制。这幅插图应该清楚地表明，对于小三角形，着色像素总数的很大一部分是辅助像素，这引起了一些关于如何合并相邻三角形四边形的研究关注。然而，尽管这种优化很聪明，但当前的API规则不允许这样的优化，而且当前的硬件也不允许这样做。当然，如果硬件供应商在某一点上决定，在四边形上浪费的着色工作是一个严重的问题，足以迫使该问题，这可能会改变。

### 属性插值

像素着色器的另一个独特功能是属性插值–所有其他着色器类型，包括我们到目前为止看到的类型（VS）和我们仍然要讨论的类型（GS、HS、DS、CS），都直接从前面的着色器阶段或内存获取输入，但像素着色器前面还有一个额外的插值步骤。在上一部分讨论Z时，我已经讨论了一些这方面的内容，Z是我们看到的第一个插值属性。

其他插值属性的工作方式大致相同；在三角形设置期间（GPU可能会选择稍微推迟此计算，例如，直到知道三角形的至少一个平铺通过分层Z测试，但这与我们无关），然后在像素着色期间，计算它们的平面方程，有一个单独的单元，它使用四边形的像素位置和我们刚刚计算的平面方程执行属性插值。

**更新**：Marco Salvi（在下面的评论中）指出，虽然过去有专门的插值器，但现在的趋势是让它们返回重心坐标以插入平面方程。实际计算（每个属性两次乘法相加）可以在着色器单元中完成。

所有这些都不应该令人惊讶，但是还有一些额外的插值类型需要讨论。首先，有“常量”插值器，它们（令人惊讶！）在整个基本体上都是常量，并从“前导顶点”（在基本体设置过程中确定的顶点）获取每个顶点属性的值。硬件可能对此有一个快速路径，或者只是建立一个相应的平面方程；两种方法都可以。

然后就没有透视插值了。这通常会以不同的方式建立平面方程；通过将每个顶点处的属性值除以相应的w，为基于X、Y的插值建立透视校正插值的平面方程，或通过构建三角形边向量，为重心插值建立平面方程。但是，如果将非透视插值属性的平面方程设置为基于X、Y的插值，而不将每个顶点处的值除以相应的w，则非透视插值属性的计算成本最低。

### “质心”插值很棘手

接下来，我们有“质心”插值。这是一个标志，不是一个单独的模式；它可以与透视模式和无透视模式相结合（但不能与常量插值相结合，因为这是毫无意义的）。它还被命名为“无操作”，除非启用了多重采样。对于多采样ob，这是一种解决实际问题的有点粗糙的方法。问题是，使用多重采样时，我们在光栅化器中的多个采样点上评估三角形覆盖率，但我们只对每个像素进行一次实际着色。纹理坐标等属性将在像素中心位置进行插值，就好像整个像素被基本体覆盖一样。这可能会在以下情况下导致问题：



这里，我们有一个部分被原语覆盖的像素；四个小圆圈描绘了4个采样点（这是默认的4xMSAA模式），而中间的大圆圈描绘了像素中心。请注意，大圆在基本体之外，它的任何“插值”值实际上都是线性外推；例如，如果应用程序使用纹理地图集，这就是一个问题。根据三角形的大小，像素中心的值实际上可能非常遥远。质心采样解决了这个问题。最初的解释是GPU获取原语覆盖的所有样本，计算它们的质心，并在该位置采样（因此得名）。这之后通常会添加一个附加内容，即这只是一个概念模型，只要GPU选择的采样点在基本体中，GPU就可以自由地以不同的方式进行采样。

如果您认为硬件不太可能对覆盖的样本进行计数，那么将它们相加，然后除以计数，然后加入俱乐部。下面是实际发生的情况：

§  如果所有采样点都覆盖了基元，则按常规进行插值，即在像素中心（恰好是所有合理采样模式的所有采样位置的质心）。

§  如果并非所有采样点都覆盖三角形，则硬件将拾取其中一个覆盖三角形的采样点，并在该采样点处进行计算。所有覆盖的采样点（根据定义）都位于基本体内部，因此这是可行的。

该拾取过去是任意的（即留给硬件）；我相信到目前为止，DX11实际上规定了具体的操作方式，但这更多的是在不同的硬件之间获得一致的结果，而不是API用户真正关心的事情。如前所述，它有点粗糙。它也会搞乱部分覆盖像素的四边形的导数计算——真倒霉。我能说什么呢，它可能是工业强度的管道胶带，但它仍然是管道胶带。

最后（DX11中新增！）还有“拉模型”属性插值。常规属性插值在像素着色器启动之前自动完成；pull model interpolation向像素着色器添加执行插值的实际指令。这允许着色器计算其自身的位置以在处采样值，或仅在某些分支中插值属性，而不在其他分支中插值属性。归根结底，像素着色器能够在着色器运行时向插值单元发送附加请求。

### 实际着色器主体

同样，API文档中很好地解释了一般的着色器原理，所以我不打算讨论单个指令是如何工作的；一般来说，答案是“正如你所期望的那样”。不过，关于像素着色器执行，有一些有趣的地方值得讨论。

第一个是：纹理采样！等等，我不是已经在第4部分讲了很久纹理采样器了吗？是的，但这是纹理采样器方面的事情——如果你还记得的话，有一点是关于纹理缓存未命中如此频繁，以至于采样器通常被设计为在每个请求（16-32像素，记住！）中至少保留一次对主内存的未命中，而不会延迟。这是一个很大的周期——数百个周期。而在这一切发生的时候，让这些完美的ALU闲置将是对它们的巨大浪费。

所以着色器单元实际上做的是在发出纹理样本后切换到不同的批次；然后，当该批次发出纹理样本（或完成）时，它将切换回以前的批次之一，并检查纹理样本是否存在。只要每个着色器单元在任何给定时间都有几个批次可以处理，就可以很好地利用可用资源。不过，它确实增加了完成单个批处理的延迟——这也是延迟与吞吐量之间的折衷。现在你应该知道哪一方在GPU上获胜了：吞吐量！总是。这里要注意一件事。现在你应该知道哪一方在GPU上获胜了：吞吐量！总是。这里需要注意的一点是，保持多个批次（或NVidia硬件上的“扭曲”，或AMD的“波前”）同时运行需要更多寄存器。如果着色器需要大量寄存器，则着色器单元可以保持较少的扭曲；如果它们的数量较少，那么在某一点上，不等待纹理结果的可运行批次用完的可能性更高。如果没有可运行的批处理，那么您就倒霉了，必须暂停，直到其中一个批处理返回结果。这是很不幸的，但这类事情的硬件资源是有限的——如果你的内存不足，你的内存也就不足了。

还有一点我还没有提到：着色器中的动态分支（即循环和条件）。在着色器单元中，对每个批处理的所有元素的处理通常按锁定步骤进行。所有“线程”在同一时间运行相同的代码。这意味着ifs有点棘手：如果任何线程想要执行If的“then”分支，所有线程都必须执行，尽管它们中的大多数可能最终会使用一种称为谓词的技术忽略结果，因为它们一开始不想下降到那里。。对于“else”分支也是如此。如果条件句在各个元素之间趋于一致，那么这种方法非常有效；如果条件句或多或少是随机的，那么效果就不太好了。最糟糕的情况是，您总是执行每个if的两个分支。哎哟循环的工作原理类似——只要至少有一个线程希望继续运行循环，该批处理/扭曲/波前中的所有线程都将继续运行。

另一个特定于像素着色器的是discard指令。像素着色器可以决定“杀死”当前像素，这意味着它不会被写入。同样，如果批中的所有像素都被丢弃，则着色器单元可以停止并转到另一批；但是，如果至少有一条线还未完成，其余的线就会被拖走。DX11通过从像素着色器写入输出像素覆盖率，在此处添加了更细粒度的控制（始终与原始三角形/Z-test覆盖率进行AND，以确保着色器不能在其基本体之外写入，以确保完整性）。这允许着色器丢弃单个样本，而不是整个像素；例如，它可以用于使用着色器中的自定义抖动算法实现Alpha到覆盖。

像素着色器还可以写入输出深度（该功能已经存在了相当长的一段时间）。根据我的经验，这是一种很好的方法，可以消除早期Z、分层Z和Z压缩，并且通常可以获得最慢的路径。到目前为止，您已经对这些东西的工作原理有了足够的了解，并了解了原因。：）

像素着色器生成多个输出–通常，每个渲染目标有一个4分量向量，其中（当前）最多可以有8个。然后着色器将结果沿管道向下发送到D3D所称的“输出合并”。这将是我们下次的主题。

但在我结束之前，像素着色器可以从D3D11开始做最后一件事：它们可以写入无序访问视图（UAV）——这是只有计算和像素着色器才能做到的。一般来说，在计算着色器执行期间，无人机将取代渲染目标；但与渲染目标不同，着色器可以确定写入自身的位置，并且没有隐式的API顺序保证（因此名称的“无序访问”部分）。现在，我只想提到这个功能的存在——当我开始计算着色器时，我会更多地讨论它。

**更新**：在评论中，Steve向我介绍了正确的AMD术语（文章的第一个版本没有“Wavefronts”的名称，因为我记不起来了），还发布了一个链接，指向Kayvon Fatahaliana的这篇精彩演讲，解释了GPU上的着色器执行，有很多漂亮的图片，我可以费心制作：）。如果你对着色器核心的工作方式感兴趣，你应该去看看。

就这样！这次没有太多的警告。如果这里遗漏了什么，那是因为我真的忘记了它，而不是因为我觉得它太神秘或太具体了，所以不能写在这里。请随意指出评论中的遗漏，我会看看我能做些什么。

## 2011年图形管道之旅，第9部分

2012年9月22日

0:07

### 2011年图形管道之旅，第9部分

2011年7月12日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

欢迎回来！这篇文章讨论像素处理的后半部分，即“连接阶段”。前一阶段是获取少量输入流，并将它们转换为着色器单元的大量独立任务。现在我们需要将大量独立计算重新组合为一个（正确排序）内存操作流。正如我在关于光栅化和早期Z的文章中所做的那样，我将首先快速描述在一般级别上需要做什么，然后我将讨论如何将其映射到硬件。

### 再次合并像素：混合和延迟Z

在管道的底部（D3D称之为“输出合并”阶段），我们有后期的Z/模板处理和混合。这两个操作在计算上都相对简单，它们都分别更新渲染目标/深度缓冲区。“更新”这里的操作意味着它们属于读-修改-写类型。因为所有这些都发生在每一个通过管道的四边形上，所以它也是带宽密集型的。最后，它是顺序敏感的（混合和Z处理都需要按API顺序进行），所以我们需要确保首先将处理过的四边形按顺序排序。

我已经解释过Z-处理，混合是其中一种工作方式，与您预期的非常相似；它是一个固定的函数块，每个渲染目标首先执行乘法、乘法和可能的减法运算。该块刻意保持简单；它与着色器单元分离，因此需要自己的ALU，我们需要使用真实的ALU我更希望它尽可能小：我们想花费我们的芯片面积（和电源预算）在着色器单元中的ALU上，它们使GPU上运行的所有代码受益，而不是仅在像素管道末端使用的固定功能单元上。此外，我们需要它具有短而可预测的延迟：管道的这一部分需要处理数据才能正确。这就限制了我们的选择，即以吞吐量换取lat我们仍然可以并行处理不重叠的四边形，但如果我们画很多小三角形，每个屏幕位置都会有多个四边形，我们最好能尽快写出它们，否则我们所有大规模并行的像素处理都是徒劳的。

### 遇到ROPs

ROP是处理这部分管道的硬件单元（正如您可以从复数形式看出的，有不止一个）。根据您的要求，首字母缩写表示“渲染输出单元”、“光栅操作管道”或“光栅操作处理器”。实际名称相当古老–它源自纯2D硬件加速的时代，硬件的主要目的是实现快速位位位闪烁。经典2D ROP设计有三个输入–当前（目标）帧缓冲区中的像素值、源数据和掩码输入–然后计算3个值中的某些函数并将结果写回帧缓冲区。注意，这是在真彩色显示之前：图像数据通常是位平面格式，函数是一些二进制逻辑函数。然后在某个点位平面消失（为了支持将像素的位保持在一起的“粗块”表示法），真彩色成为标准，开-关掩模被alpha通道取代，按位操作被混合取代，但名称仍然存在。因此，即使在2011年，当原始架构的最后一个残余是“逻辑运算”时在OpenGL中，我们仍然称它们为

ROPs。

那么，对于blend/late Z，我们需要在硬件方面做些什么呢？一个简单的计划：

1.    从内存读取原始渲染目标/深度缓冲区内容–内存访问，延迟时间长。可能还涉及深度缓冲区和渲染目标解压缩！（稍后我将解释渲染目标压缩）

2.    将传入的着色四边形按正确（API）顺序排序。这需要一些缓冲，所以当四边形没有以正确的顺序完成时（考虑循环/分支、丢弃和可变纹理提取延迟），我们不会立即暂停。注意，这里我们只需要根据原语ID进行排序–来自同一原语的两个四边形永远不会重叠，如果它们不重叠，就不需要排序！

3.    执行实际的混合/后期Z/模具操作。这是一个数学问题——可能需要几十个周期，即使是采用深度流水线的单元。

4.    再次将结果写回内存，压缩等等。一路上——再次出现很长的延迟，尽管这次我们不等待结果，所以在这方面问题不大。

所以，构建late-Z/混合单元，添加一些压缩逻辑，将其连接到一侧的内存，并对另一侧的着色四边形进行缓冲，我们就完成了，对吗？

好吧，不管怎样，在理论上。

但我们需要以某种方式覆盖长延迟。事实上，所有这些都发生在每一个四边形上）。所以我们也需要担心内存带宽…内存带宽？内存带宽不是有问题吗？现在仔细观察我在第二部分把兔子放在帽子里之后，从帽子里拉出兔子的过程（哦，那是一个多星期前的事了，希望小兔子在里面还可以…）。

### 内存带宽冗余：DRAM页

在第2部分中，我描述了DRAM的2D布局，以及如何更快地保持在一行中，因为更改活动行需要时间——因此为了获得理想的带宽，您希望在两次访问之间保持在同一行中。问题是，单DRAM行有点大。如今，单个DRAM芯片的尺寸已达到千兆位，虽然它们不一定是正方形（事实上，2:1的纵横比似乎是首选），但您仍然可以粗略计算出将有多少行和列；对于512MB（=64MB），我们期望类似16384×32768的数据，也就是说，一行大约是32k位或4k字节（或者可能是2k或8k，但在大致范围内的某个地方——你知道了）。这对于进行内存事务来说是一个相当不方便的大小。

因此，有一个折衷方案：页面。DRAM页是一行中大小更方便的部分（到目前为止，通常为256或512位），通常在单个突发中传输。现在让我们以512位（64字节）为例。每像素32位–深度缓冲区的标准，对于渲染目标来说仍然相当常见，尽管渲染工作负载肯定正在转向64位/像素格式–这足够存储16像素的数据。嘿，这很有趣–我们通常以16到64个像素为一组对像素进行着色！（NV更接近于小型终端，AMD更倾向于大型终端）。事实上，我在光栅化器/早期Z部件中引用的8×8瓷砖尺寸来自AMD；如果NV在4×4分片上做了粗略的遍历（和分层Z，他们称之为“Z-cull”），我不会感到惊讶，尽管快速的网络搜索没有发现任何东西来证实这一点或排除它。不管是哪种方式，情节都会变得更加复杂。可能是因为我们试图以一种能够提供良好DRAM页面一致性的顺序遍历像素？当然是的。请注意，这对内部呈现目标布局也有影响：我们希望确保像素的存储使得单个DRAM页面实际上具有有用的形状；出于着色目的，4×4或8×2像素的DRAM页面比16×1像素的页面（请记住–四边形）有用得多。这就是渲染目标在内存中通常没有完全线性布局的原因。

这给了我们另一个在组中对像素进行着色的理由，也给了我们另一个进行两级遍历的理由。但是我们可以再挤些牛奶吗？当然我们可以：我们仍然有内存延迟需要覆盖。通常的免责声明：这是一个我没有关于GPU实际功能的详细信息的地方，所以我在这里描述的是猜测，而不是事实。无论如何，一旦我们光栅化了一块瓷砖，我们就知道它是否会生成任何像素。在这一点上，我们可以选择一个ROP来处理该分幅的四边形，并将一个命令排队以将相关的帧缓冲区数据提取到缓冲区中。当我们从着色器单元返回着色四边形时，数据应该在那里，我们可以毫不延迟地开始混合（当然，如果混合禁用或标识，我们可以完全跳过此加载）。类似地，对于Z数据–如果我们在像素着色器之前运行早期Z，我们可能需要更早地分配ROP并获取深度/模具数据，可能是在瓷砖通过粗略Z测试之后。如果我们晚运行Z，我们可以在获取帧缓冲区像素的同时预取深度缓冲区数据（除非Z完全关闭）。

所有这些都足够早，可以避免除最快的像素着色器（通常是内存带宽受限）之外的所有像素着色器的延迟暂停。还存在输出到多个渲染目标的像素着色器的问题，但这取决于该功能的实现方式。可以多次运行着色器（如果有固定大小的输出缓冲区，则效率不高，但最简单），或者可以通过相同的ROP运行所有渲染目标（但最多可以运行8个渲染目标，最高可达128位/像素–这是我们谈论的大量缓冲空间），或者可以为每个输出渲染目标分配一个ROP。

当然，如果我们在ROPs中有这些缓冲区，我们不妨将它们当作一个小缓存（即，将它们保留一段时间）。如果你画了很多小三角形，这会很有帮助——不管怎样，只要它们在空间上是局部的。同样，我不确定GPU是否真的这样做，但这似乎是一个合理的做法（您可能希望每批刷新这些缓冲区一次左右，以避免完全写回缓存带来的同步/一致性问题）。

带来）。

好的，这解释了事情的记忆方面，以及我们已经讨论过的计算部分。下一步：压缩！

### 深度缓冲和颜色缓冲压缩

我已经在第7部分中解释了它的基本工作原理，同时谈到了Z；事实上，关于深度缓冲区压缩，我没有太多补充。但是我提到的所有带宽问题也存在于颜色值上；这对于常规渲染来说并不坏（除非像素着色器输出像素的速度足以达到内存带宽限制），但对于MSAA来说，这是一个严重的问题，在MSAA中，我们突然将每个像素存储2到8个采样。像Z一样，我们需要一些无损压缩方案来在常见情况下节省带宽。与Z不同，每个平铺的平面方程不适合纹理像素数据。

但是，这没有问题，因为实际上，MSAA像素数据更容易优化：

请记住，像素着色器仅在每个像素上运行一次，而不是在每个采样上运行一次–除非您使用的是samplefrequency着色，但这是D3D11的一个功能，而且还不常用（尚未？）。因此，对于一个原语完全覆盖的所有像素，存储的2-8个样本通常是相同的。这就是常见的颜色缓冲区压缩方案背后的想法：写入标志位（每像素、每四元组或更大的粒度），表示对于压缩块中的所有像素，所有每采样颜色实际上是否相同。如果是这样的话，我们只需要每像素存储一次颜色。这在写回期间很容易检测到，同样（很像深度压缩），它需要一些标记位，我们可以将它们存储在小型片上SRAM中。如果有一条边穿过像素，我们需要全部带宽，但是如果三角形不是太小（而且它们基本上从来都不小），我们至少可以在部分帧上节省大量带宽。同样，我们可以使用同样的机器来加速清算。

关于清除和压缩，还有一件事要提：一些GPU

“分层Z”式机制，用于存储最近清除的大块像素（光栅化块，可能更大）。然后，您只需要在内存中为整个瓷砖（或更大的块）存储一个颜色值。这将为某些缓冲区提供非常快速的颜色清除（同样，您需要一些标记位来完成此操作！）。然而，一旦任何非清晰颜色的像素被写入平铺（或更大的块），就需要清除“这是刚刚清除”标志。但是我们确实在clear本身和第一次从内存读取磁贴时节省了大量内存带宽。

这就是我们的第一个渲染数据路径：只是顶点和像素着色器（最常见的路径）。在下一部分中，我将讨论几何体着色器以及管道的外观。但在我结束这篇文章之前，我有一个适合本节的小主题。

**旁白：为什么没有完全可编程的混合？**

*每个人*编写渲染代码的人有时会对这一点感到好奇——常规的混合管道有时会让人感到非常痛苦。那么为什么我们不能得到完全可编程的混合呢？毕竟，我们有完全可编程的着色！好吧，我们现在有了必要的框架来适当地研究这个问题。我已经看到了这方面的两个主要建议——让我们依次看看这两个建议：

1.    混合在像素着色器中–即像素着色器读取帧缓冲区，计算混合方程，写入新的输出值。

2.    可编程混合单元–“混合着色器”，必要时带有完整着色器指令集的子集。发生在PS后的单独阶段。

### 1.混合像素着色器

这看起来很简单：毕竟，我们在着色器中已经有了加载和纹理样本，对吗？那么，为什么不允许读取当前渲染目标呢？结果表明，无约束读取是一个非常糟糕的主意，因为这意味着每个被着色的像素都可能（潜在地）影响其他被着色的像素。那么，如果我在左边的四边形中引用一个像素呢？好吧，这个四边形的着色器可能正在运行。或者，我可以对当前四元组的一半和另一半当前处于活动状态的四元组进行采样–我现在该怎么做？在这方面，正确的结果到底是什么，更不用说我们可能必须按顺序对所有四边形进行着色才能可靠地获得它们？不，那是一罐虫子。从像素着色器中的帧缓冲区进行的无约束读取将被输出。但是，如果我们得到一条特殊的渲染目标读取指令，对当前位置的一个活动渲染目标进行采样，会怎么样？现在，这要好得多——现在我们只需要担心写入当前四元组的位置，这是一个更容易处理的问题。

然而，它仍然引入了排序约束；我们必须检查光栅化器生成的所有四边形与当前像素着色的四边形。如果光栅化器刚刚生成的四边形想要写入将由当前正在运行的其中一个像素着色器写入的样本，我们需要等待PS完成，然后才能分派新四边形。这听起来不太糟糕，但我们如何跟踪它？我们可以有一个“此示例当前正在着色”位标志…那么我们需要多少位？在1920×1080和8x MSAA的情况下，大约2MB的内存（即字节而非比特）是全局的，共享的，并决定了我们可以发出新的四元组的速率（因为我们需要在发出四元组之前将其标记为忙碌）。更糟糕的是，对于分级Z等标记位，它们只是一个提示；如果我们用完了它们，我们仍然可以渲染，尽管速度要慢一些。但是这个内存不是可选的。我们不能保证正确性，除非我们真的跟踪每个样品！如果我们只是跟踪每像素（甚至四像素）的“忙”状态，对像素的任何写入都会阻止所有其他此类写入，会怎么样？这是可行的，但它会极大地损害我们的MSAA性能：如果我们跟踪每个样本，我们可以并行地对相邻的、不重叠的三角形进行着色，这没有问题。但是，如果我们按像素跟踪（或以较低的粒度跟踪），我们可以有效地序列化所有边四边形。我们的填充率会发生什么变化，例如，粒子系统有大量的透支？使用我描述的管道，这些渲染（或多或少）与ROPs可以将传入像素合并到存储缓冲区的速度一样快。但是如果我们需要避免冲突，我们最终会按顺序对单个重叠粒子进行着色。这对我们的着色器单元来说不是好消息，它们设计为用延迟换取吞吐量，一点也不。

好的，所以整个追踪工作是个问题。如果我们只是强制着色按顺序执行呢？也就是说，保持整个过程管道化，所有着色器同步运行；现在我们不需要跟踪，因为像素的完成顺序与我们放入管道的顺序相同！但这里的问题是，我们需要确保批处理中的着色器实际上始终使用完全相同的时间，这会带来不幸的后果：对于每个纹理采样，您始终必须等待最坏情况下的延迟时间，并且需要始终执行每个分支的两侧（有些人可能在某个时候需要then/else分支，我们需要所有的东西都占用相同的时间！），总是以相同的迭代次数运行所有循环，不能在丢弃时停止着色…不，这听起来也不像是赢家。

好的，是时候面对现实了：我所描述的体系结构中的像素着色器混合带来了一系列非常棘手的问题。那么第二种方法呢？

### 2. “混合着色器”

我现在就说：这是可以实现的，但是…

让我们说它有它自己的问题。这一次，我们现在需要在ROPs中使用另一个完整的ALU+指令解码器/序列器等。这不是一个小变化——不是在设计上，也不是在面积上，也不是在功率上。第二，正如我在这篇文章开始时提到的，我们的常规“只需广泛使用”对于blend来说，战术效果不太好，因为在这个地方，我们可能会有一堆四边形连续击中相同的像素，需要按顺序处理它们，所以我们需要低延迟。这是一个与常规统一着色器单元非常不同的设计点，所以我们不能使用它们（这也意味着混合着色器中的纹理采样/内存访问是一个很大的禁忌，但我怀疑在这一点上会让任何人感到震惊）.第三，纯串行执行在这一点上已经过时了–吞吐量太低。因此我们需要对其进行管道处理。但要进行管道处理，我们需要知道管道的长度！对于常规混合单元，管道的长度是固定的，所以很容易。混合着色器可能是相同的。事实上，由于设计限制，您不太可能获得混合着色器–更像是一个混合寄存器组合器，实际上，指令数量的上限（可能相对较低），由管道长度决定。

重点是，这里的串行执行确实限制了我们的设计，这些设计仍然是相对较低的水平；远没有我们喜欢的完全可编程的着色器单元。一个更好的混合单元和一些额外的混合模式，你肯定可以得到；一个更开放的寄存器组合器风格的设计，可能，尽管API的家伙和硬件都没有uys会非常喜欢它（API是因为它是一个固定的功能块，硬件是因为它很大，需要一个大的ALU+控制逻辑，而他们不想拥有它）。完全可编程，带有分支、循环等——不会发生。在这一点上，你不妨咬紧牙关，采取一切措施来获得“混合像素着色器”场景要正常工作。

…这篇文章到此为止！下次见。

## 2011年图形管道之旅，第10部分

2012年9月22日

0:08

### 2011年图形管道之旅，第10部分

2011年7月20日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

欢迎回来。上一次，我们深入到像素管道的底部。这次，我们将切换到管道的中间，看看D3D10中最明显的添加：几何体着色器。但首先，我要介绍一下本系列中如何分解图形管道，以及这与视图t有何不同他将向你展示。

### 有多个管道/管道阶段的解剖结构

这可以追溯到第3部分，但重复这一点非常重要：例如，如果您查看D3D10文档，您会发现“D3D10管道”的图表，其中包括所有可能处于活动状态的阶段。“D3D10管道”包括几何体着色，即使您没有设置几何体着色，对于Stream Out也是如此。在D3D10的纯功能模型中，几何体着色阶段始终存在；如果您没有设置几何体着色，它只是非常简单（而且很无聊）：数据只是在未经修改的情况下通过下一个管道阶段（光栅化/流式输出）。

这是指定API的正确方式，但在本系列中考虑它的方式是错误的，我们关心的是如何在硬件中实际实现该函数模型。那么，到目前为止，我们看到的两个着色器阶段是如何实现的呢？对于VS，我们使用了输入汇编程序，它为着色准备了一块顶点，然后使用dispatc将该批处理添加到着色器单元（着色器单元会对其进行一段时间的咀嚼），然后稍后我们将结果返回，将其写入缓冲区（用于基本体组装），确保其顺序正确，然后将其发送到下一个管道阶段（消隐/剪裁等）。对于PS，我们从光栅化器接收要着色的四边形，对它们进行批处理，将它们缓冲一段时间，直到着色器单元可以自由地接受新的批处理，将批处理分配给着色器单元（着色器单元会对其咀嚼一段时间），然后一段时间后，我们将结果返回，将它们写入缓冲区（用于ROPs），确保它们的顺序正确，然后混合/延迟Z并将结果发送到内存。听起来有点熟悉，不是吗？

事实上，当我们想要通过着色器单元完成某些事情时，它总是这样：我们需要一个缓冲区在前面，然后是一些调度逻辑（实际上对于所有着色器类型都是通用的，可以共享），然后我们广泛地并行运行一组着色器，最后我们需要另一个缓冲区和一个将结果（我们从着色器单元接收到的结果可能是无序的）排序回API顺序的单元。

我们已经看到了着色器单元（和着色器执行）和分派；事实上，现在我们已经看到了像素着色器（具有一些特性，如导数计算、辅助像素、丢弃和属性插值），在我们开始计算着色器之前，我们不会看到着色器单元功能有任何大的增加，它们具有专门的缓冲区类型和原子。因此，在接下来的几个部分中，我将不讨论着色器单元；不同着色器类型的真正不同之处在于输入和输出的数据的形状和解释不处理IO（算术、纹理采样）的部分保持不变，所以我不再谈论它们。

### 这三个字的形状要变暗

让我们看看几何体着色器的IO缓冲区是什么样子的。让我们从输入开始。好吧，这相当简单——这只是我们从顶点着色器编写的内容！或者，不完全是这样；几何体着色器查看的是基元，而不是单个顶点，所以我们真正需要的是基元汇编（PA）的输出。请注意，有多种方法可以处理此问题；PA可以将基本体展开（如果多次引用顶点，则复制顶点），也可以将一块顶点（我将继续使用我之前使用的32个顶点）与相关联的小“索引缓冲区”交给我们（因为我们将索引到一个32个顶点的块中，每个索引只需要5位）。任何一种方法都很好；前者是我在PA之后讨论的剪辑/剔除的自然输入格式，但后者在运行GS时需要更少的缓冲区空间，所以我将在这里使用该模型。

您需要担心GS的缓冲区空间量的一个原因是，它可以在一些相当大的基本体上工作，因为它不仅支持普通线或三角形（每个基本体分别支持2个和3个顶点），还支持具有邻接信息的线/三角形（每个基本体4/6个顶点）。D3D11还添加了更胖的输入原语——GS还可以使用多达32个控制点作为输入的补丁。复制例如16个控制点面片的顶点，每个面片最多可以有16个矢量属性（32个带有D3D11）？那将是一些严重的内存浪费。所以我假设这条路径的顶点是非重复的，索引的。它为一批原语生成输入：VS输出，加上一个（相对较小的）索引缓冲区。

现在，几何体着色器将按基本体运行。对于顶点着色器，我们需要收集一批顶点，我们使用一个简单的贪婪算法选择批次大小，该算法尝试将尽可能多的顶点打包到一个批次中，而无需在多个批次中拆分一个基本体，这很公平。

对于像素着色，我们从光栅化器中获得大量的四边形，并将它们全部打包成批。几何体着色器有点不方便–我们的输入块保证至少包含一个完整的基本体，可能包含几个基本体–但除此之外，该块中的基本体数量完全取决于顶点缓存命中率。如果它很高，我们使用三角形，我们可能会得到40-43；如果我们使用带有邻接信息的三角形，如果我们运气不好，我们可能只有5个。

当然，我们可以尝试从几个输入块中收集原语，但这也有点尴尬。现在，我们需要为单个GS批处理保留多个输入块和索引缓冲区，如果单个批处理可以引用多个索引缓冲区，这意味着该批处理中的每个原语现在需要知道从何处获取索引和顶点数据–更多的存储需求、更多的管理和更多的开销。也很丑。当然，即使有两个输入块，如果您以较低的顶点缓存命中率命中两个输入批，您的利用率仍然很差。您可以支持更多的输入块，但这会消耗内存——请记住，您也需要为输出几何体留出空间（稍后我会介绍）。

这是我们的第一个障碍：使用VS，我们基本上可以选择我们的目标批量大小，我们选择不总是生成完整的批量，以便使我们在PA（这里是GS，后面也是HS）中的生活更轻松。对于PS，我们总是对四边形进行着色，即使是相当小的Tri通常也会击中多个四边形，所以我们得到了四边形数与Tri数的一个不错的比率。但是对于GS，我们没有对管道两端的完全控制（因为我们在中间！），并且我们需要每个原语的多个输入顶点（相对于每一个输入三角形的多个Quad），因此缓冲大量的输入是昂贵的（无论是在内存方面还是在管理开销上）。

在这个阶段，您基本上可以选择要合并多少个输入块，以获得一个用于几何体着色的基本体块；由于内存需求，这个数字将会很低（我会非常惊讶地看到超过4个），并且根据您判断GS的重要性，您甚至可以选择1，即根本不跨输入块合并，并且在GS着色块/扭曲/波前上的利用率很差！这对于三角形来说不太好，对于具有更多顶点的基本体来说也不太好，但当GS的主要使用案例实际上是将点扩展到四边形（点精灵）时，可能会渲染偶尔的立方体阴影贴图（使用视口阵列索引/渲染目标索引–我将稍后介绍）。

### GS输出：这里也没有玫瑰园

那么从输出端看情况如何？同样，这比普通的VS数据流更复杂。事实上要复杂得多；虽然VS只输出一件事（着色顶点），未着色顶点和着色顶点之间的对应关系为1:1，但GS输出可变数量的顶点（最多可达编译时指定的最大值），并且从D3D11开始，它还可以有多个输出流–但是，最多可以在管道的其余部分上发送一个流，这就是我现在谈论的道路。GS数据的另一个目的地（流输出）将在下一部分中介绍。

GS产生可变大小的输出，但它需要在有限的内存需求下运行（除其他外，缓冲区可用的内存量决定了可以并行几何着色的原语数量），这就是为什么在编译时输出顶点的最大数量是固定的。这（连同写入的输出属性的数量）决定了分配了多少缓冲区空间，从而间接决定了并行GS调用的最大数量；如果这个数字太低，延迟就不能完全隐藏，GS将在一定百分比的时间内暂停。

还请注意，GS输入基本体（例如点、线、三角形或面片，可选择使用邻接信息），但输出顶点–即使我们将基本体发送到光栅化器！如果输出原语类型为points，则这是微不足道的。然而，对于直线和三角形，我们需要再次将这些顶点重新组合成基本体。这是通过使输出顶点分别形成直线或三角形条带来实现的。这可以很好地处理3种最重要的情况：单线、三角形或四边形。如果GS试图很好地进行一些实际的拉伸：单线、三角形或四边形，就不那么方便了。如果GS尝试进行一些实际拉伸或生成其他“复杂”几何体，这就不太方便了，因为这些几何体通常需要几个“重新启动条带”标记（可以归结为每个顶点一个位，表示当前条带是继续还是启动新条带）。那么为什么会有限制呢？在API级别，它似乎相当随意——为什么GS不能只输出一个顶点列表和一个小的索引缓冲区？

答案可以归结为两个词：原始汇编。这就是我们在这里要做的——取一些顶点，并将它们组合成一个完整的基本体，然后沿着管道发送。但是我们已经在这个数据路径中使用了这个功能块，就在GS前面。因此，对于GS，我们需要第二个基本装配阶段，我们希望保持简单，装配三角形条实际上非常简单：三角形总是按顺序从输出缓冲区中取出3个顶点，只有一点粘合逻辑来跟踪当前的缠绕顺序。换句话说，条带支持的复杂程度并不比最简单的基本体（非索引线/三角形）复杂多少，但它们仍然为四边形等典型基本体节省了输出缓冲区空间（从而为我们提供了更大的并行性潜力）。

### 再次订购API

然而，这里有一些问题：在常规顶点着色路径中，我们确切地知道一个批次中有多少基本体以及它们在哪里，甚至在着色顶点到达PA缓冲区之前——所有这些都是从我们设置批次着色的点开始修复的。例如，如果我们有多个单元用于cull/clip/triangle设置，它们都可以并行启动；他们知道从哪里获取顶点数据，并且他们可以提前知道三角形将具有哪个“序列号”，这样就可以将所有的顶点数据按顺序排列。

对于GS，我们通常不知道在得到输出之前要生成多少个原语——事实上，我们可能没有生成任何原语！但我们仍然需要尊重API顺序：首先是从GS调用0生成的所有原语，然后是从调用1生成的所有原语，依此类推，直到批处理结束（当然，批处理也需要按顺序处理，与VS相同）。所以对于GS，一旦我们得到结果，我们首先需要扫描输出数据，以确定完整原语开始的位置。只有这样，我们才能开始进行剔除、剪辑和三角形设置（可能并行）。更多的额外工作！

### VPAI与RTAI

使用GS添加的这两个功能实际上不会影响几何体着色器的执行，但会对下游的处理产生一些影响，因此我想在这里提及它们：视口数组索引（这里简称VPAI）和渲染目标数组索引（RTAI）。首先是RTAI，因为它更容易解释：正如您希望知道的，D3D10增加了对纹理数组的支持。RTAI为您提供了“渲染到纹理数组”支持：您可以将纹理数组设置为“渲染目标”，然后在GS中，您可以选择每个基本体，该基本体应该指向哪个数组索引。请注意，由于GS写入的是顶点而不是基本体，因此我们需要拾取一个顶点来选择每个基本体的RTAI（以及VPAI）；这始终是“前导顶点”，即属于基本体的第一个指定顶点。RTAI的一个示例用例是在一次过程中呈现立方体贴图：GS决定每个基本体应该将其发送到哪个立方体面（可能有几个）。VPAI是一种正交功能，允许您设置多个视口和剪切矩形（最多15个），然后根据每个基本体决定使用哪个视口。例如，这可以用于在单个过程中渲染级联阴影贴图中的多个级联，也可以与RTAI结合使用。

如前所述，这两个功能都不会对GS处理产生显著影响——它们只是附加到基本体上的额外数据，然后在以后使用：VPAI在视口变换过程中被消耗，而RTAI则一直使用到像素管道。

### 迄今为止的总结

好的，所以在输入端有一些麻烦-我们不能完全选择我们的输入数据格式，所以我们需要对输入数据进行额外的缓冲，即使这样，我们也有一个可变数量的输入原语，我们不一定能够将其划分成很好的大批量。在输出端，我们再次组装了数量可变的原语，不一定知道哪个GS将提前生成多少个原语（尽管对于某些GSs，我们可以通过编译的代码静态地确定这一点，例如，因为所有顶点发射都在流控制之外，或者在循环内部，具有已知的迭代计数，并且没有提前退出），并且必须花一些时间解析输出，然后才能将其发送到triangle设置。

如果这听起来比我们在VS-only的情况下更复杂，那是因为它是。这就是为什么上面提到的将GS视为始终运行的东西是一个错误-即使是一个非常简单的GS，除了通过当前三角形外，什么都不做，还要经过两个缓冲阶段，一轮额外的基本装配，所有这些都是有成本的，而且往往是累加的：我在D3D10硬件相当新的时候检查过它，在AMD和NVidia硬件上，甚至一个纯穿透GS都是相当新的3倍，在AMD和NVidia硬件上，甚至一个纯穿透GS都比它慢3倍到7倍完全没有GS（即在几何体受限的情况下）。我没有在较新的硬件上重新运行此实验；我假设它现在已经变得更好了（这是实现GS的第一代，而功能在实现GS的第一代GPU中通常没有良好的性能），但问题依然存在：仅仅通过GS管道发送一些东西，即使那里根本没有发生任何事情，也有非常明显的成本。

GSs按条带顺序生成基本体也无济于事；对于顶点着色器，我们对每个顶点进行一次调用，读取一个顶点并写入一个顶点（尼斯）。但是对于GS，我们可能最终只运行一批11个基本体（因为输入缓冲区中没有足够的基本体），每一个都运行相当长的时间，并产生8个输出顶点。这是一个以低利用率运行的很长时间！（请记住，我们需要16到16之间的某个位置）

如果GS主要由一个循环组成（例如，在“渲染到立方体贴图”（render to cube map）中），那就更麻烦了我提到的RTAI案例，我们在立方体中的6个面上循环，检查该面上是否可见三角形，如果是，则输出一个三角形。6个面的计算是真正独立的；如果可能，我们希望并行运行它们！

### 奖励：GS实例

好吧，请输入GS Instanceing，D3D11中的另一个新特性—遗憾的是，文档记录得很差（我不确定SDK中是否有关于它的好例子）。不过，解释起来相当简单：对于每个输入原语，GS不仅运行一次，而且运行多次（这是在编译时选择的静态计数）.它基本上相当于将整个着色器包装在

|  |  |
| --- | --- |
| 对于（int i=0；i<N；i++） | |
|  | |
| // ... |  |
|  |

{

}

块，通过实际为每个输入原语生成多个GS调用，仅在着色器外部处理循环，这有助于我们获得更大的批处理大小，从而提高利用率。i作为系统生成的值导出到着色器（在D3D11中，使用语义SV\_GSInstanceID）.因此，如果您有这样一个GS，只需去掉外部循环，添加一个[instances（N）]声明，并将i声明为具有正确语义的输入，您只需做很少的工作，它就可能运行得更快–这就是为大规模并行机提供更多独立作业的魔力！

无论如何，关于几何体着色器就是这样。我已经跳过了Stream Out，但这篇文章已经足够长了，而且这是一个足够大的主题（并且足够独立于GS！）来保证它自己的文章。下一篇文章，更准确地说。在那之前！

## 2011年图形管道之旅，第11部分

2012年9月22日

0:08

### 2011年图形管道之旅，第11部分

2011年8月14日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

欢迎回来！这一次，焦点将是流式输出（SO）。这是一种将几何体着色器阶段的输出存储到内存的工具，而不是将其发送到管道的其余部分。

这可用于缓存蒙皮顶点数据，或作为D3D10级硬件上使用D3D10 API的穷人计算着色器（请注意，对于D3D11，您可以仅使用CS 4.0，即使是在D3D10硬件上）。就像我上次提到的GS实例一样，API文档中对其中一些内容的描述非常糟糕，因此我将对API的使用进行一些评论，尽管它在技术上超出了本系列的预期范围。

### 顶点着色器流输出（即，使用NULL GS时为SO）

这是D3D10（或D3D11）文档中没有正确解释的功能之一；事实上，除了“开始使用流输出阶段（Direct3D 10）”中的一句简单的评论外，这里根本没有提到它。您应该从示例中找出答案-这些示例本身并没有明确说明发生了什么。遗憾的是，VS Stream out比GS SO更容易，并且本身有一些非常有用的应用程序（例如缓存蒙皮顶点）。

在D3D10和D3D111中是这样做的：您只需将顶点着色器字节码（而不是GS字节码）传递给CreateGeometryShaderWithStreamOutput。是的，文档中提到了“编译的几何体着色器的大小”这里–忽略它。你得到的是一个几何体着色器对象，你可以将其传递给toGSSetShader。实际上，这是一个空几何体着色器–它实际上没有经过GS处理。它只是一些包装（更像是管道胶带）为了使其适合API模型，所有渲染都要经过GS阶段，因此就在GS之后–尽管我上次已经解释过，当没有GS集时，实际的硬件往往会完全跳过GS阶段。

因此，着色顶点会像以前一样组装成基本体，但不会像前面所描述的那样被发送到管道的其余部分，而是被转发到Stream Out，它们会像往常一样到达缓冲区中。它们到底会发生什么取决于Stream Out声明（在创建时传递）。在Stream Out声明中，应用程序可以指定它希望每个输出向量在Stream Out目标（简称SO目标）中结束的位置。如果SO声明“匹配”顶点着色器输出声明（即相同顺序的相同属性），输入缓冲区中的数据可以或多或少地流式处理到内存中。如果它与声明不完全匹配，则可能会跳过着色器写入的某些属性，或者以不同的顺序写入这些属性。无论哪种方式，都会涉及一些额外的重新排序。这可能涉及专用的重新排序单元（基本上从SO输入缓冲区实现聚集类型操作），或者它可能涉及生成大量的小内存写入而不是大的突发写入，或者类似的操作。无论哪种方式，这都是额外的工作，而且通常速度较慢；具体触发慢路径的细节取决于硬件的具体情况，但实际上，这并不重要。如果您想要最佳的SO性能，只需确保SO declaration和输出声明一致。

另一点是，SO通常无法访问到内存子系统的高性能路径。与ROPs不同，SO不是真的（还没有？）在当前的GPU设计中，它是一个完全的公民，所以它通常只能访问一个内存通道或类似的东西。如果你通过so生成大量数据，这一点需要记住。so输出总是完全浮动，这使得so输出更加复杂，因此使用压缩顶点数据类型无法节省带宽。

关于VS SO的最后一点意见：正如我前面提到的，SO对集合的基本体而不是单个顶点进行操作。请注意，如果基本体集合在管道中的距离太远，它会丢弃邻接信息，并且由于这种情况发生在之前，与邻接信息相对应的顶点也不会将其放入SO缓冲区。因此，对基本体进行操作并非单个顶点与实例化单蒙皮网格（以单个姿势）等用例相关好几次。如果您像往常一样绘制三角形网格，然后使用SO，这会导致数据爆炸–每个输入基本体有3个未打包、未共享的顶点。这是可行的，但在SO和稍后的顶点输入端都不能有效地利用带宽。相反，您应该绘制三角形网格作为第一遍中的一个（非索引）点列表，因此每个顶点只着色一次。SO网格作为一个（非索引）第一次扫描中的点列表，从而对每个顶点精确着色一次。然后SO缓冲区与原始顶点缓冲区以1:1的对应关系结束，仅使用蒙皮顶点而不是非蒙皮顶点。然后，您可以将该顶点缓冲区与原始基元拓扑和索引缓冲区一起使用。

### 几何体着色器SO：多个流

对于空GS，这基本上是这样工作的，除了涉及到一个几何体着色器，这增加了一些新功能（和复杂性）。在VS的情况下，我们只有一个输出流（注意，流是D3D11+功能–它们在D3D10级别HW上不存在）。该流可以发送到SO或not，也可以沿管道发送到viewport/clip/cull或not，但仅此而已。但几何体着色器允许多个流，这使得输出路由有点困难。

基本上，每个GS最多可以写入4个流（从D3D11开始）。每个流可以发送到SO目标–是的，复数：单个流可以写入多个SO目标，但单个SO目标只能从一个流接收值，即，这是一种一对多关系，而不是完全通用的多对多关系。流的存在对SO缓冲有一定的影响–我们现在可能有多个输入缓冲区，每个流一个，而不是像我在NULL GS案例中描述的那样只有一个输入缓冲区。除SO目标外，最多可沿管道发送一个流，即常规渲染管道，并可同时使用SO。

与NULL GS的情况一样，SO适用于基本体，而不是单个顶点–也就是说，在GS中输出的条带在进入SO之前会展开为完整的直线或三角形。

### 跟踪输出大小

这里还有另一个问题：我们不一定知道有多少输出数据将由SO生成。对于GS，这是因为每次GS调用都可能产生数量可变的输出原语；但即使在简单的VS情况下，一旦涉及到索引原语，应用程序可能会在其中插入一些“原语剪切”索引，从而影响实际写入的原语数量。这是一个问题，如果我们以后想从那个缓冲区中绘制，因为我们不知道到底有多少顶点在那个里！我们确实有一个上限——创建时缓冲区的最大容量——但仅此而已。现在，这可以通过某种查询机制来解决，但一旦你想清楚了，这似乎是相当倒退的：在我们使用SO缓冲区进行绘图时，我们显然知道我们实际编写了多少个原语——毕竟SO单元需要跟踪其当前输出位置！如果我们采用某种查询机制，我们最终将通过总线将单个32位值传输回驱动程序，驱动程序将其传递给API，API将其传递给应用程序，然后应用程序立即发送另一个绘图，以相反的方向再次遍历所有层。

这就是问题的解决方法。取而代之的是DrawAuto。这个想法非常简单——GPU已经知道它实际写入输出缓冲区的有效顶点数量；SO单元在写入时会跟踪它，最后一个计数器也会保存在内存中（与缓冲区一起），因为应用程序可能会在多个过程中呈现给SO缓冲区。然后，这个计数器用于DrawAuto，而不是让应用程序自己提交一个明确的计数——这大大简化了事情，完全避免了昂贵的往返。请注意，此查询机制确实存在–用于检查写入的顶点数和确定是否发生溢出。但它不在从SO缓冲区进行渲染的关键路径上，这使驱动程序开发人员的工作变得简单得多。

就这样，真的。这篇文章中没有太多的硬件信息，从管道的角度来看，也不是一个非常有趣的话题，这就是为什么我花了这么长时间才完成的原因；很抱歉。下一步是细分-这应该快得多，因为这是一个有趣的话题：）

## 2011年图形管道之旅，第12部分

2012年9月22日

0:09

### 2011年图形管道之旅，第12部分

2011年9月6日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

欢迎回来！这一次，我们将调查什么可能是“海报男孩”D3D11/Shader 5.x硬件生成中引入的功能：细分。这一功能很有趣，因为它是一个有趣的主题，也因为它标志着很长一段时间以来第一次在图形管道中添加了一个重要的用户可见组件，而该组件不可编程。

与概念上非常简单的几何体着色器不同（它只是一个可以看到整个基本体而不是单个顶点的着色器），“细分”主题需要更多的解释。细分几何体的方法有很多种——仅举最流行的几种，有几十种风格的样条线面片、各种类型的细分曲面和置换贴图——因此从要点“细分”开始单是GPU为我们提供了什么服务，以及它们是如何实现的，这一点并不明显。

为了描述硬件镶嵌的工作原理，在中间开始可能是最容易的——使用实际的原始镶嵌步骤，以及应用于它的各种要求。（D3D11术语中的外壳着色器和域着色器、OpenGL 4.0行话中的细分控制着色器和细分评估着色器）稍后。

### 镶嵌-与您预期的不太一样

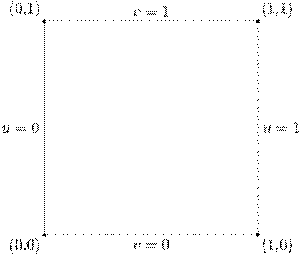
由Shader 5.x class HW实现的细分属于“基于面片”的种类。CG文献中的面片类型主要根据用于从控制点（B样条面片、Bézier三角形等）构造细分点的函数来命名.但我们现在将忽略这一部分，因为它是在新的着色器类型中处理的。实际的固定函数细分单元仅处理输出网格的拓扑（即有多少个顶点以及它们如何相互连接）；从这个角度来看，基本上只有两种不同类型的面片：基于四边形的面片，它们定义在具有两个正交坐标轴的参数域上（这里我称之为u和v，两者都在[0,1]中）通常构造为两个单参数基函数和基于三角形的面片的张量积，这些面片使用基于重心坐标的三个坐标（u、v、w）的冗余表示（即。

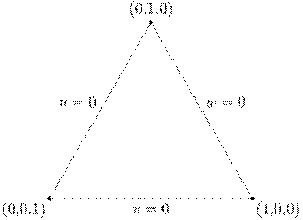


).用D3D11的说法，它们分别是“四”域和“三”域。还有一个

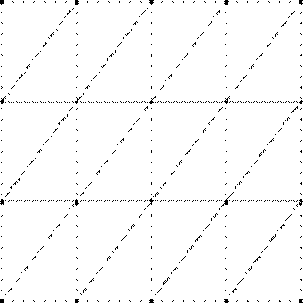
“等值线”域，它代替二维曲面生成一条或多条1D曲线；我将以与本系列中处理直线和点基本体相同的方式来处理它：我承认它的存在，但不会进一步详述。

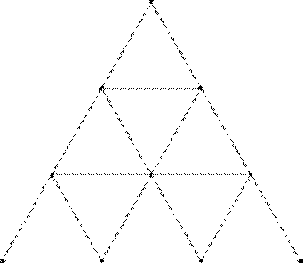
镶嵌基本体可以在各自的域坐标系中自然绘制。对于四边形，绘制域的明显选择是作为一个单位正方形，所以这就是我将使用的；对于三角形，我将使用等边三角形来可视化事物。以下是我将在本文中使用的顶点和边坐标系标记：



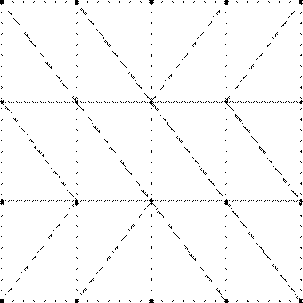


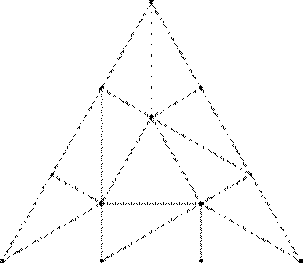
不管怎样，三角形和四边形都有我认为是“自然”的方式来镶嵌它们，如下所示。但事实证明，这并不是你所得到的网状拓扑结构。





以下是细分器将为给定输入参数生成的实际网格：





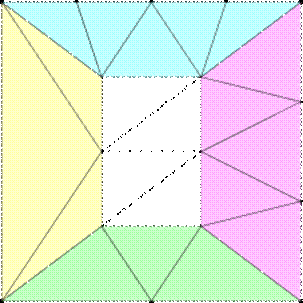
对于四边形，这（大致上）是我们所期望的——除了一些翻转的对角线，我将在一分钟内讨论。但三角形是一个完全不同的野兽。它的拓扑结构与我上面展示的“自然”细分非常不同，包括不同数量的顶点（12而不是10）.很明显，这里发生了一些有趣的事情，而且一些事情恰好与处理不同镶嵌级别之间的过渡方式有关。

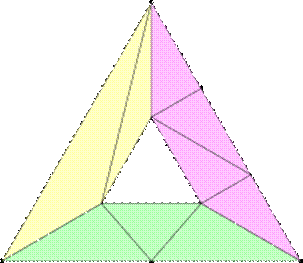
### 收支平衡

房间里的大象正在处理面片之间的过渡。细分单个三角形（或四边形）很容易，但我们希望能够确定每个面片的细分因子，因为我们只需要四边形）这很容易，但我们希望能够确定每个面片的细分因子，因为我们只希望在需要的地方使用三角形，而不是在远处（可能是背面剔除）浪费大量三角形除此之外，我们希望能够在不占用额外内存的情况下快速、理想地完成此操作；这意味着不可能进行全局修复后通行证或类似的操作。

解决方案——如果您编写了外壳或域着色器，您已经遇到过——是让所有实际的细分工作完全在本地进行，并将确保生成网格的水密性的负担推到着色器上。这是一个单独的主题，除其他外，需要在域着色器代码中非常小心；我将跳过着色器中表达式求值的所有细节，坚持使用基本方法。基本机制是每个面片都有多个细分因子（TF），在外壳着色器中计算：一个或两个用于面片的实际内部，每个边加一个。面片内部的TFs可以自由选择；但是如果两个面片共享一条边，它们最好沿该边计算完全相同的TFs，否则将出现裂缝。硬件不在乎-它将通过它处理每个面片赛尔夫。如果你做的每件事都正确，你会得到一个很好的水密网格，否则——好吧，那是你的问题。硬件需要确保的是，有可能得到水密网格，最好是具有合理的效率。这本身在某些地方是很棘手的；我稍后会谈到这一点。

因此，这里有一些新的参考面片–这一次沿每条边有不同的TF，因此我们可以看到它是如何工作的：





我对由不同的边缘镶嵌因子影响的区域着色；中间的未着色的中心部分仅取决于内部的TFS。在这些图像中，U＝0（黄色）边缘具有Tf为2，V＝0（绿色）边缘具有Tf为3，U＝1 /W＝0（粉红色）边缘具有TF 4，V＝1（Quad仅，青色）。edge的TF为5–正好是对应外边上的顶点数。从这两张图像中可以明显看出，拓扑的基本构建块只是一种很好的方法，可以将两条具有不同顶点数的细分边相互缝合。此操作的细节是将两条具有不同顶点数的细分边缝合在一起顶点之间相互连接。这方面的细节有点棘手，但不是特别有趣，所以我不会深入讨论。

至于内部TF，四边形相当简单：上面的四边形的内部TF沿u方向为3，沿v方向为4。除了第一行和最后一行/列被各自的缝合三角形替换外，基本上是该尺寸的规则网格的几何图形（如果任何边的TF为1，则生成的网格将具有与u/v的内部TF均为2相同的结构，即使它们小于该值）。三角形稍微复杂一些。我们已经看到了奇怪的TF，TF为



，它们生成一个由



同心环，最里面是一个三角形。对于TFs，我们得到

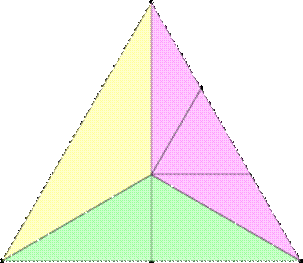


具有中心顶点而不是中心三角形的同心环。下面是最简单的

即使如此，



，它仅由边缝合和中心顶点组成。



最后，在对四边形进行三角剖分时，通常选择对角线远离面片中心（在域坐标空间中），并遵循一致的断连规则。这只是为了确保生成网格的最大旋转对称性——如果有额外的自由度，不妨使用它们！

### 分数细分因子和总体管道流量

到目前为止，我只讨论了两种所谓的“分区类型”中的整数TFs

“Integer”和“Pow2”，这就是细分器看到的所有内容。如果着色器生成非整数（或分别为非2次幂）TF，它将简单地向上取整到下一个可接受的值。更有趣的是剩下的两种分区类型：分数奇数和分数偶数细分。而不是从细分因子跳到细分因子（这将导致可见POP），新顶点从与网格中现有顶点相同的位置开始，然后随着TF的增加逐渐移动到其新位置。

例如，对于分数奇数细分，如果对上述三角形使用3.001的内部TF，生成的网格将非常类似于TF为3的网格–但从拓扑上看，它与TF为5的情况相同，即它是一个具有3个同心环的面片，即使中间环非常窄。然后，TF变为clo到5，中间环扩张直到最终到达TF 5的最后位置。一旦你提高了TF超过5，网格将在拓扑上与TF为7相同，但在中间又有一些几乎退化的三角形，等等。

然后，细分器的输出由两部分组成：第一，细分顶点在域坐标中的位置，第二，相应的连接信息–基本上是索引缓冲区。

现在，在解释了固定函数细分单元的基本功能之后，让我们后退一步，看看我们需要做些什么来实际生成基本体：首先，我们需要输入一组输入控制，我们需要做些什么来实际生成基本体：首先，我们需要输入一组输入控制点，这些控制点包含一个补丁到外壳Sh中HS然后计算输出控制点和“面片常数”（两者都会传递给域着色器）加上所有细分因子（本质上只是更多面片常数）。然后我们运行固定函数tessellator，它为我们提供了一组域位置以运行域着色器，以及相关的索引。在运行DS之后，我们再进行一轮基本体组装，然后将基本体发送到GS管道（如果它处于活动状态）或视口变换、剪裁和消隐（如果没有）。

让我们看一看HS阶段。

### 外壳着色器执行

与几何体着色器一样，外壳着色器将完整（面片）原语用作输入–所有导致输入缓冲头痛的问题。头痛的程度完全取决于输入面片的类型。如果面片类型类似于立方Bézier面片，则每个面片需要4×4=16个输入点，并且可能只产生一个四分之一的输出（如果剔除了面片，甚至一点也没有）；显然，这是一个需要处理的数据量，并且不适合进行非常有效的着色。另一方面，如果细分将普通三角形作为输入（很多人都这样做），输入缓冲非常简单，不太可能成为问题或瓶颈的来源。

更重要的是，与几何体着色器（为每个基本体运行）不同，外壳体着色器不经常运行–它们在每个面片上运行一次，只要有任何实际的细分（即使在适度的TFs下）进行换句话说，即使HS输入有点低效，问题也不会像GS那样严重，因为我们不经常点击它。

外壳着色器的另一个很好的属性是，与几何体着色器不同，它们没有可变数量的输出数据；它们生成固定数量的控制点，每个控制点具有固定数量的关联属性，加上固定数量的面片常量。所有这些在编译时都是静态已知的；没有动态运行时缓冲区管理没有必要。如果一次对16个外壳进行外壳着色，我们甚至在开始执行着色器之前就知道每个外壳的数据将在哪里结束。这绝对是几何体着色器的优势；对于许多几何体着色器，可以静态地知道将生成多少个输出顶点（例如，因为所有导致发出/剪切指令的控制流都可以在编译时进行静态计算），对于所有这些，都有一个保证的最大输出顶点数，但是对于HS，我们有一个保证的固定数量的输出数据，不需要额外的分析。简言之，输出缓冲区管理没有问题，除了一个事实，即根据基元类型，我们可能需要大量的输出缓冲区空间限制我们可以实现的并行量（由于内存/寄存器限制）。

最后，外壳着色器在D3D11中编译的方式有些特殊；所有其他着色器类型基本上由一个代码块组成（可能有一些子程序），但外壳着色器生成时会分解为多个阶段，每个阶段可以由多个（独立的）代码块组成执行线程。驱动程序和着色器编译器程序员主要对细节感兴趣，但只需说，您的平均HS以一种暴露大量潜在并行性（如果有的话）的形式打包就足够了。当然，微软似乎真的很想避免这一次困扰几何着色器的瓶颈。

无论如何，外壳着色器在每个面片上都会产生一系列输出；除了TFs（被发送到细分器单元）之外，大部分输出都会保留到相应的域着色器运行之前。如果任何TFs小于或等于零（或NaN），面片被剔除，相应的控制点和面片常数被悄悄地丢弃。否则，细分器（实现上述功能）将启动，读取刚刚着色的面片，并开始大量生成域点位置和三角形索引，我们需要为DS执行做好准备。

### 域着色器

就像顶点着色一样，我们希望将多个域顶点收集到一个批次中，然后一起着色，然后传递到PA。固定函数细分器可以处理这个问题：“Just”处理它，同时生成顶点位置和索引（我将“Just”放在在这里引用，因为这确实涉及一些簿记）。

就输入和输出而言，域着色器确实非常简单：它们得到的唯一实际随顶点变化的输入是域点u和v坐标（w，在使用时，不需要计算或由tesselator传入；因为



，可计算为

，可计算为



)。其他所有内容都是面片常量、控制点（面片上的所有控制点都相同）或常量缓冲区。输出基本上与顶点着色器相同。

简言之，一旦我们到达DS，生活就好了；数据流几乎和VS一样简单，这是一条我们知道如何高效运行的路径。这也许是D3D11细分管道相对于几何体着色器的最大优势：实际的三角形放大不会发生在着色器中，我们浪费了宝贵的ALU循环，需要保留缓冲区空间用于最坏情况下的顶点估计，但在基本上是状态机的本地化元素（细分器）中，只获得很少的输入（几个TF）并产生非常紧凑的输出（实际上是索引缓冲区，每个输出顶点加一个2D坐标）.正因为如此，我们需要更少的内存进行缓冲，并且可以让我们的着色器单元忙于实际的着色工作，而不是内务处理。

这篇文章就到此为止——下一篇：计算着色器，也就是我在本系列原始大纲中的最后一部分！直到那时。

### 最后发言

像往常一样，我抄了一些弯路。有一种“等值线”面片类型，我根本没有讨论过（如果有任何需求，我可以写出来）。镶嵌器具有各种对称性和精度要求；就顶点域位置而言，您基本上可以期望不同硬件供应商之间的位精确结果，因为D3D11规范确实确定了这一点。有意不确定的是顶点或三角形的生成顺序–一个实现可以做它想做的事情，只要它始终如一地这样做（即，相同的输入必须始终产生相同的输出）。这也有一系列微妙的约束——例如，由细分器编写的所有域位置都需要同时具有u和1-u（以及v和1-v）完全可以表示为float，有很多必要的条件

然后，着色器可以生成水密网格（此规则特别重要，以便两个面片之间的共享边AB（AB到一个面片，BA到另一个面片）可以以相同的方式对两个面片进行细分）。

编写域着色器使其实际上不会产生裂缝是一件棘手的事情，需要非常小心；我故意回避这个话题，因为它超出了本系列的范围。我没有提到的另一个更为琐碎的问题是由细分器生成的三角形的缠绕顺序（回答：这取决于应用程序-支持顺时针和逆时针方向）。

对外壳和域着色器的输入/输出缓冲的描述有点简洁，但它与我们已经看到的阶段非常相似，因此我宁愿保持简短，避免额外的混乱；如果速度过快，请重新阅读顶点着色器和几何体着色器上的文章。

最后，由于细分管道可以输入GS，因此存在一个问题，即它是否可以生成邻接信息。对于面片的“内部”而言，这是可以想象的（只需编写更多的细分单元索引），但一旦到达面片边缘，它就会变得很难看，因为交叉面片邻接正需要一种全局“网格感知”，而这种全局“网格感知”正是细分管道设计极力避免的。因此，长话短说，不，镶嵌器不会为GS生成邻接信息，只会生成普通三角形。

## 2011年图形管道之旅，第13部分

2012年9月22日

0:10

### 2011年图形管道之旅，第13部分

2011年10月9日

*这篇文章是这个系列的一部分*“2011年图形管道之旅”。

欢迎回到最后一个“官方”节目本系列的一部分–我将在将来做更多的GPUPOST，但本系列已经足够长了。我们已经浏览了图形管道的所有常规部分，直到不同的细节级别。这就省去了DX11中引入的一个主要新功能：计算着色器。所以这将是我这次的主题。

### 执行环境

对于本系列，重点放在架构级别的整体数据流上，而不是着色器执行（在其他地方有很好的解释）.对于目前为止的阶段，这意味着关注每个阶段的输入和输出；内部工作的方式通常由数据的形状决定。计算着色器是不同的–它们自己运行，而不是作为图形管道的一部分，因此其界面的表面积要小得多。

事实上，在输入端，实际上根本没有任何用于输入数据的缓冲区。除了绑定常量缓冲区和资源等API状态之外，唯一的输入计算着色器获得的是它们的线程索引。这里存在巨大的混淆可能性，因此要记住最重要的一点：一个“线程”是CS环境中调度的原子单元，它与操作系统提供的线程有着本质上的不同，您可能会将其与术语联系起来。CS线程有自己的标识和寄存器，但它们没有自己的程序计数器（指令指针）或堆栈，也没有单独调度。

事实上，CS中的“线程”取代了顶点着色时单个顶点的位置，或像素着色时单个像素的位置。它们的处理方式相同：将一束线程（通常，在16到64之间）组合成“扭曲”或“波前”让他们在lockstep中运行相同的代码。CS线程不会被调度——扭曲和波前会被调度（在本文的其余部分，我将坚持使用“扭曲”，用“波前”代替AMD）。为了隐藏延迟，我们不会切换到不同的“线程”（用CS的说法），但指向不同的扭曲，即不同的线程束。扭曲中的单个线程不能单独执行分支；如果此类捆绑中至少有一个线程希望执行某段代码，则该代码将被捆绑中的所有线程处理，即使大多数线程最终会丢弃结果。简而言之，CS“线程”更像SIMD通道，而不是编程中其他地方看到的线程；请记住这一点。

这就解释了“线程”和“扭曲”级别。上面是“线程组”级别，它处理–谁会想到？–线程组。线程组的大小在着色器编译期间指定。在DX11中，线程组可以包含1到1024个线程之间的任何位置，并且线程组大小不是指定为单个数字，而是指定为给出线程x、y和z坐标的3元组。此编号为scheme主要是为了方便处理2D或3D资源的着色器代码，尽管它也允许遍历优化。在宏级别，CS执行以线程组的倍数进行调度；D3D11中的线程组ID再次使用3D组ID，与线程ID相同，原因也差不多。

线程ID——可以以各种形式传递，具体取决于着色器的偏好——是计算着色器的唯一输入，但并非所有线程都是相同的；与我们以前看到的其他着色器类型截然不同。不过，这只是冰山一角。

### 线程组

上面的描述听起来像是线程组在这个层次结构中是一个相当任意的中间级别。然而，缺少一个重要的位，这使得线程组非常特殊：线程组共享内存（TGSM）.在DX11级硬件上，计算着色器可以访问32k的TGSM，这基本上是同一组中线程之间通信的草稿行。这是不同CS线程通信的主要（也是最快）方式。

那么这在硬件中是如何实现的呢？很简单：一个线程组中的所有线程（嗯，实际上是扭曲）都由同一个着色器单元执行。着色器单元只有至少32k（通常多一点）的本地内存。而且因为所有分组的线程共享同一个着色器单元（以及相同的ALU集等），不需要为共享内存访问包含复杂的仲裁或同步机制：在任何给定周期中只有一个扭曲可以访问内存，因为在任何周期中只有一个扭曲可以发出指令！现在，当然这个过程通常是管道化的，但这不会改变基本不变量：每个着色器单元，我们有只有一个TGSM；访问TGSM可能需要多个管道阶段，但对TGSM的实际读取（或写入）将只发生在一个管道阶段内，并且该周期内的内存访问都来自同一个扭曲。

但是，这还不足以实现实际的共享内存通信。问题很简单：

上面的不变量保证每个周期只有一组对TGSM的访问，即使我们不添加任何互锁来防止并发访问。这很好，因为它使硬件更简单更快。但是，从着色器程序的角度来看，它不保证内存访问以任何特定顺序发生，因为扭曲可以或多或少地随机安排；这完全取决于在特定时间点谁是可运行的（不等待内存访问/纹理读取完成）。更微妙的是，正是因为整个过程是流水线式的，所以可能需要一些周期才能使对TGSM的写入变为“可见”读取；当TGSM的实际读取和写入操作发生在不同的管道阶段（或同一阶段的不同阶段）时，就会发生这种情况。因此，我们仍然需要某种同步机制。输入屏障。屏障有不同类型，但它们只由三个基本组件组成：

*1.    组同步*。组同步障碍会强制当前组中的所有线程到达该障碍，然后它们中的任何线程都可能消耗超过该障碍。一旦某个扭曲到达该障碍，它将被标记为不可运行，就像它正在等待内存或纹理访问完成一样。一旦最后一个扭曲到达该障碍，剩余的扭曲将被删除重新激活。这一切都发生在Warp调度级别上；它添加了额外的调度约束，这可能会导致暂停，但不需要原子内存事务或类似的事情；除了在微观级别上丢失利用率之外，这是一个相当便宜的操作。

*2.    群体记忆障碍*。由于组中的所有线程都在同一个着色器单元上运行，因此这基本上相当于管道刷新，以确保所有挂起的共享内存操作都已完成。无需与当前着色器单元外部的资源同步，这意味着它的成本也相当低。

*3.    设备存储器屏障*。这会阻止组中的所有线程，直到所有内存访问完成为止–直接或间接（例如通过纹理样本）。正如本系列前面所述，GPU上的内存访问和纹理样本具有很长的延迟–可以认为超过600次，且经常超过1000次循环–因此，这种障碍将非常有害。

DX11提供了不同类型的屏障，将上述几个组件组合成一个原子单元；语义应该是显而易见的。

### 无序访问视图

我们现在已经处理了CS输入并了解了一些CS执行。但是我们将输出数据放在哪里呢？答案是“无序访问视图”，简称UAV。UAV看起来有点类似于像素着色器中的渲染目标（事实上，除了像素着色器中的渲染目标外，还可以使用UAV），但有一些非常重要的语义差异：

§  最重要的是，正如这一点所表明的，无人机的访问是“无序的”，即API不保证访问以任何特定顺序可见。在渲染原语时，保证按API顺序对四边形进行Z测试、混合和写回（如本系列第9部分中详细讨论的），或者至少产生与实际相同的结果-这需要大量的努力。无人机不需要这样的努力-无人机访问会在着色器中遇到时立即发生，这可能与API顺序非常不同。它们是

但并非完全无序；虽然在API调用中没有保证操作顺序，但API和驱动程序仍将协作以确保在API调用中保留感知到的顺序。因此，如果您有一个复杂的计算着色器（或像素着色器），则会立即向UAV写入第二个（更简单的）从同一底层资源读取的CS，第二个CS将看到完成的结果，而不是部分写入的输出。

§  无人机支持随机访问。像素着色器只能写入每个渲染目标的一个位置–其对应的像素。同一像素着色器可以写入其绑定的任何无人机中的任意位置。

§  无人机支持原子操作。在经典的像素管道中，没有必要；我们保证不会发生任何冲突。但是无人机提供的自由形式执行，不同的线程可能试图同时访问一块内存，我们需要同步机制来处理这一问题。

所以从一个“CPU程序员”从他的观点来看，无人机相当于sharedmemory多处理系统中的常规RAM；它们是内存中的窗口。更有趣的是原子操作问题；这是当前GPU与CPU设计有很大差异的一个领域。原子操作；这是当前GPU与CPU设计有很大差异的一个领域。

### 原子学

在当前的CPU中，共享内存处理的大部分魔法都是由内存层次结构（即缓存）来处理的。要写入内存，活动内核必须首先声明相应缓存线的独占所有权。这是使用所谓的“缓存一致性协议”来完成的，通常是MESIand后裔。细节与本文无关；重要的是，因为写入内存需要获得独占所有权，所以从来不会有两个内核同时尝试写入某个位置的风险。在这种模型中，可以通过在持续时间内保持独占所有权来实现原子操作如果我们一直拥有独占所有权，那么在我们执行原子操作时，就不可能有其他人试图写入同一个位置。同样，这一操作的实际细节很快就会变得毛茸茸的（特别是当涉及到分页、中断和异常时），但30000英尺的视图就足以满足本文的目的。

在这种类型的模型中，原子操作是使用常规的核心ALU和加载/存储单元执行的，大多数“有趣的”工作都发生在缓存中常规内存访问，尽管有一些额外的要求。但是，有几个问题：最重要的是，缓存一致性的标准实现“窥探”要求协议中的所有代理相互通信，这有严重的可伸缩性问题。有一些方法可以绕过此限制（主要使用所谓的基于目录的一致性协议），但它们增加了内存访问的复杂性和延迟。另一个问题是，所有锁和内存事务实际上都发生在缓存线级别；如果两个不相关但经常更新的变量共享同一缓存线，则可能最终导致“乒乓”在多个核心之间，导致大量的一致性事务（以及相关的减速）。这个问题称为“虚假共享”.软件可以通过确保不相关的字段不落入同一缓存线来避免此问题；但在GPU上，应用程序既不知道也不控制执行期间的缓存线大小和内存布局，因此此问题将更加严重。

当前的GPU通过不同的内存层次结构来避免这个问题。而不是在着色器单元内处理原子操作（这再次引发了“谁拥有哪个内存”的问题），有专用的原子单元直接与共享的最低级别缓存层次结构通信。只有一个这样的缓存，因此不存在一致性问题；缓存线存在于缓存中（这意味着它是当前的）或不存在于缓存中（这意味着内存中的副本是当前的）.原子操作包括首先将各自的内存位置放入缓存（如果还没有），然后使用原子单元上的专用整数ALU直接对缓存内容执行所需的读-修改-写操作。当原子单元在内存位置上忙时，对该位置的所有其他访问都将暂停。由于存在多个原子单元，因此有必要确保它们从不尝试访问同一内存位置同时，实现这一点的一个简单方法是使每个原子单元“拥有”一组特定的地址（静态地，而不是像缓存线所有权那样动态地）。这是通过将负责的原子单元的索引计算为要访问的内存地址的一些散列函数来完成的。（请注意，我无法确认当前的GPU是如何工作的；我在官方文档中没有找到关于原子单元如何工作的详细信息）。

如果着色器单元希望对给定内存地址执行原子操作，它首先需要确定哪个原子单元负责，等待它准备好接受新命令，然后提交操作（如果需要原子操作的结果，则可能等待操作完成）。原子单元一次可能只处理一个命令，或者可能有一小部分未完成的请求；当然，还有各种各样的分配和排队细节需要正确处理，以便原子操作处理是合理的，这样着色器单元将始终取得进展。再次，我在这里不再详细介绍。

最后一句话是，当然，出色的原子操作被视为“设备内存”访问，与内存/纹理读取和UAV写入相同；着色器单元需要跟踪其出色的原子操作，并确保在遇到设备内存访问障碍时完成。

### 结构化缓冲区和附加/使用缓冲区

除非我遗漏了什么，否则这两种缓冲区类型是我还没有提到的最后一个与CS相关的功能。而且，从硬件的角度来看，实际上没有太多要说的。结构化缓冲区对驱动程序内部着色器编译器的提示比其他任何东西都多；它们给驱动程序一些关于它们将如何运行的提示o可以使用–即，它们由可能一起访问的具有固定跨距的元素组成–但它们最终仍会编译为常规内存访问。结构化缓冲区部分最终可能会影响驾驶员对位置内存访问的决定。结构化缓冲区部分可能会影响驾驶员对位置内存访问的决定它们在内存中的位置和布局，但它不会向模型添加任何基本的新功能。

追加/消耗缓冲区类似；它们可以使用现有的原子指令实现。事实上，它们有点类似，除了追加/消耗指针不在资源中的显式位置之外，它们是资源外部的边带数据，可以使用特殊的原子指令访问。（与结构化缓冲区类似，它们的用法声明为append/consume buffer，这一事实允许驱动程序适当地选择它们在内存中的位置）。

**收工。**

而且……就这样。下一部分不再预览，本系列已完成：），但这并不意味着我已经完成了。我要做一些重组和部分重写工作——这些博客文章都是未经处理的原始文章，我打算把它们转换成一个单独的文档。与此同时，我会在这里写一些其他的东西。如果还有其他问题，我会尝试整合到目前为止得到的反馈，更正或评论，现在是时候告诉我了！我不想确定这个系列的最终清理版本的预计到达时间，但我会尽量在年底前确定下来。我们拭目以待。在此之前，感谢阅读！