GPU渲染管线之旅|06 三角形的生成和建立

欢迎回来。这次我们去看看三角形的光栅化。但在光栅化三角形之前，我们需要执行三角形设置，并且在设置三角形之前，我还要解释一下我们做的准备是为了什么，最后我们来聊聊三角形硬件光栅化算法。

1.如何画一个三角形

首先，给很熟悉这部分并自己写过软 纹理 映射的人一点小提示：三角形光栅器一次要处理一堆东西：跟踪三角形的形状，插值出坐标u和v（对于透视矫正映射，是u/z，v/z和$1/z$），执行Z缓冲测试（对于透视矫正映射，可以用$1/z$缓冲替代），然后处理实际的纹理（还有着色），以上步骤都在一个安排好可用寄存器的大循环里。在硬件中，这些东西都被打包成很整齐的小模块，这便于设计以及独立测试。硬件中的"三角形光栅器"是告诉你三角形覆盖了哪些像素的块；某些情况下，它也会给出三角形中这些像素的重心坐标。但仅此而已。不仅没有给出u和v，甚至没有$1/z$。当然也没有纹理和着色，但通过使用专用的纹理和shader单元这些都不是个事。

其次，如果你写过自己的三角形映射器，你就可能会用过像 Chris Hecker 这种透视纹理映射的增量式扫描线光栅算法。在没有SIMD单元的处理器上这是一个非常好的方法，但是它对于拥有高速SIMD单元的现代处理器并不很适合，对于硬件甚至更糟糕。就像是放在角落里的过时了的游戏主机，现在根本没人感兴趣了。就好比是三角形光栅器对于屏幕底部和右侧的边的保护带裁剪非常快，而对于顶部和左侧的边就没这么快了。只是打个比方而已。

那么，对硬件来说这个算法到底哪里不好？首先，它确实是通过逐条扫描线来光栅化三角形。当进行像素着色时就出现了问题，我们想要光栅器输出成组的2x2个像素点（所谓的"方块quads"——不要与"四边形quad"图元相混淆，quad图元在管线中被分解为一对三角形）。因为我们不仅要并行的运行两个"实例instances"，还要从他们各自的扫描线上的第一个像素开始绘制，它们可能离的很远而导致不能很好的生成我们想要的2x2的块，这就是扫描线算法的尴尬之处。而且很难高效的并行化，在x和y的方向上不对称——这意味着画一个宽8像素高100像素的三角形与一个宽100像素高8像素的三角程度是截然不同的。现在得让"x"和"y"的步进"循环"同样快来避免瓶颈——但我们要是在"y"的步进上执行所有工作，那"x"的循环就不重要了！这就有点麻烦了。

2.一个更好的方法

在1988年 Pineda 的 论文里提到了一个非常简单（对硬件更加友好）的渲染三角形的方法。这个方法可以归结为两句话：到直线的符号距离可以通过2D点积来计算（相乘再相加）——就像到平面的符号距离可以通过3D点积来计算一样。以及三角形本质上可以被定义为三条边正确侧面上的所有点的集合。所以只用遍历所有像素的坐标并且测试他们是否在三角形里就行了。这就是最基本的算法。

注意，比如当我们移动一个像素到右边，我们在X上加上一个数并同时保持Y不变。我们的边的公式有如下形式：

a，b，c 是三角形常量，所以对于X+1就是：$E(X+1,Y) = a(X+1) + bY + c = E(X,Y) + a.$

换句话说，一旦得到边的公式在已知点上的值，对于邻接像素的值仅作一些相加就可得出。还要注意，这很容易并行化：比如像AMD的硬件一次可以光栅化8x8=64像素（或是Xbox360，参考《Real-Time Rendering》第三版）。你只用计算 其中。一次计算每个三角形（和边）并保存在寄存器中。然后只需 计算左上角的三边公式
来光栅化一个8x8的像素块，之后测试结果符号位来

部。这样计算三条边，非常快，一个8x8的 三角形光栅块很适合并行化的方式，并且除了做大量的整数加法操作就没什么更复杂的了！这就是为什么在上一部分里要对齐到定点（fixed-point）网格——这样我们就可以在这用整数运算了。整数累加器比浮点运算单元可简单多了。当然我们可以选择累加器的宽度来刚好支持我们想要的视口大小，有足够的子像素精度，以及大概2~4倍的合适尺寸的保护带。

顺带一提，这里还有一个棘手点，就是填充规则：你需要保证任何一对共用一条边的三角形，共用的边附近没有像素被漏掉或者被光栅化两次。D3D和OpenGL都使用所谓的"左上角"填充规则；具体细节在各自的用户手册上有解释。我就不在这里赘述了，不过要注意这种整数光栅器，在三角形设置过程中从一些边的常数项中减去1。使其保证不会出现问题—— 相比较，Chris在他的文章中的做法就适用这项工作了。两种方法结合起来就很棒了。

仍然存在一个问题：我们如何找到要测试哪些8x8的像素块呢？Pineda提出了两种策略：1）只扫描整个三角形包围盒，或者2）一个更聪明的方案：一旦没有命中任何三角形采样点，就停止反复了。好吧，如果一次只测试一点像素点是没有问题的。但是我们现在要处理8x8个像素！同时执行64次并行相加，最后却发现没命中任何像素，太浪费了。所以，千万别这么干。

3.我们这里需要的是更多的层级

我刚才讲的是适合光栅器工作 （实际输出的采样量）的方式。为了避免像素级上的多余工作，我们应该在它之前添加另一个光栅器，这个光栅器不把三角形 光栅化成像素，只是将8x8像素块分成tiles（McCormack和McNamara的 论文中有一些详细内容，以及Greene的"Hierarchical Polygon Tiling with Coverage Masks"的结论中用到了这个想法）。光栅化边的方程到覆盖的tile的工作很类似于光栅化像素；我们要做的是按照边的方程计算整个tile的上下边界；因为方程是线性的，所以极值是在tile的边界上——实际上，可以循环4个拐角点，从公式中a和b因数的符号可以判断出是哪个拐角。底部的线相比之下计算量就很小了，也需要同样的层级——一些并行的整数累加器。如果要估算tile一个拐角的边方程，不如传到细粒度光栅器中执行：每个8x8的块需要一个参考值，还记得吗？

所以要先执行一次粗粒度光栅化来得到可能被三角形覆盖的tiles，这个光栅器可以做的小一点（8x8都足够用了），它不需要速度非常快（因为它只用来执行每个8x8的块），在这个层次，找到空的块的开销是比较小的。

可以参考Greene的论文和Mike Abrash的《Rasterization on Larrabee》，实现一个完整的层级光栅器。但对于硬件光栅器来说：实际上增加了一些对小三角形的处理工作（除非你可以跳过层次级别，但硬件数据流不是那样设计的），如果三角形非常大，要做大量的光栅化工作。这种架构下生成像素位置非常快，比Shader单元的处理速度要快。

然而，实际的问题不是处理大三角形：它们对于任何算法都很有效（当然包括扫描线光栅算法）。主要的问题在于小三角形。假如有一堆生成0或1个可见像素的小三角形，也需要执行三角形设置（马上就要讲到了），对于8x8的块至少要执行一步粗粒度光栅化和一步细粒度光栅化。小三角形很容易执行三角形设置，以及粗粒度光栅化边界。

需要注意的是，这种算法对于薄片形（又长又窄的三角形）是开销很大的——你得遍历大量的tiles，却只能得到很少的覆盖像素。所以这种情况非常慢，要尽可能的避免。

4.三角形设置阶段都做了什么?

我已经讲过了三角形的光栅化算法，在三角形设置过程中，仅需要看一下每条边使用的常量：

边方程中的三角形三条边a, b, c。
之前提到的一些派生值；如果不是要加上另一个值的话，一般不会将8x8的矩阵全部存储进硬件里。最好的方法就是只在硬件中计算，使用进位保留累加器（又名3:2 reducer，我之前写到过）来减少单独的和公式计算，然后完成常规加法。
参考获取tile的四个角的方法来获取边方程的上下边界做粗粒度光栅化。
在第一个粗粒度光栅化的参考点上，边方程的初始值（调整填充规则）。

……这些就是三角形设置阶段要做的计算。它可以归结为用于边方程的几个大整数的乘法计算，以及它们的初始赋值，一些步进值的乘法计算，

5.其它光栅化问题和像素输出

有一件事到目前还没有提，那就是裁剪矩形（scissor rect）。这只是一个屏幕对齐的矩形掩码像素。光栅器不会生成矩形之外的像素。这相当容易实现——粗粒度光栅器可以直接拒绝不与scissor rect重叠的tiles，并且细粒度光栅器 将通过"光栅化"的scissor rect的覆盖像素掩码进行AND逻辑与运算（此处的"光栅化"指的是逐行逐列的整数比较，以及一些位的AND运算）。

还有一个问题是多重抗锯齿。现在最大的挑战是需要测试每个像素的多个采样点——DX11中硬件需要至少支持8x MSAA。注意，每个像素中的采样位置不是在规则的网格里（这对于近似水平或近似垂直的边效果很不好），但大多数方向的边都可以得到不错的结果。这些不规则的采样位置是扫描线光栅算法的致命点（这是不使用它们的另一个原因！），但却很容易支持 `Pineda-style` 算法：即在三角形设置阶段计算每个边上的一些偏移量，然后对每个像素上的这些偏移量进行并行 相加和测试符号，来替代只计算一个点的方法。

比如说4x MSAA，在一个8x8的光栅器上可以做两件事情：可以将每个采样点当作是一个特别的"像素"，它表示有效的tile大小是4x4个实际的屏幕像素，细粒度光栅格中的每个块有2x2个位置对应一个"像素"，或者可以用8x8个实际像素运行4次。8x8似乎有点大了，我假设AMD是这种工作方式，其它的MSAA也都差不多。

无论如何，我们现在得到了一个细粒度的光栅器，它可以给出每个块上的8x8块的位置加上覆盖区域的掩码。非常好，不过这只是故事的一半——当今的硬件在执行 `pixel shader` 之前还要执行 `early Z` 和 `hierarchica Z` 测试，实际的光栅化与Z处理过程是交织在一起的。但最好分开来讲；所以在下一部分里，将会讲多种Z处理过程，Z比较，以及一些三角形设置——就是我们刚刚将的光栅化设置，但还有多个Z和像素着色的内 插值 ，它们也需要在之前进行设置。

6.注意事项

我把一些我认为有代表性的光栅化算法联系到了一起（这些在网上都有资料）。还有一些我没尝试过的算法都在这给出了介绍；恐怕这块内容写的有点复杂了。

本文假设为使用高端PC硬件平台。在大多数领域，特别是移动/嵌入式中，被称为tile渲染器，屏幕被分成若干tiles单独渲染。这和我讲过的8x8tile光栅化有所不同。基于tile的渲染器还至少需要一个非常粗粒度的光栅化阶段，它会预先找到被每个三角形覆盖的大块的tile；这个阶段通常被称为"装箱（Binning）"。基于tile的渲染器的工作方式有所不同，它相比"后排序（sort-last）"架构有不同的设计参数。讲完D3D11的管线，我有可能会用一到两篇文章讲一下基于tile的渲染器（如果感兴趣的话），但是现在先忽略它们，比如在常用的智能手机上的PowerVR芯片，它的处理方式是有些不同的。

在8x8的块中（其它尺寸的块也有同样的问题），当三角形小于一定尺寸或者是不合适的比例时，需要做大量的光栅化工作，并且在处理过程中会得到很糟糕的效果。我很想告诉给你一个神奇的易于并行化的算法，不过我不知道，一些硬件厂商也做的不是很好。所以就目前而言，这些都是硬件光栅化的难题。或许未来会有一个不错的解决方案。

我讲到的"边方程的下边界"适合于粗粒度光栅化，但是在某些情况下会出现错误（即需要在不覆盖任何像素的块中执行细粒度光栅化）。是有技巧减少这种情况的，但检测这些特殊情况比起在不覆盖任何像素的块中执行光栅化往往开销更大。这也是一种权衡。

在光栅化过程中用到的块通常都是固定在一个网格上的（下一篇会讲的更详细）。如果一个三角形覆盖的两个像素跨过了两个tile，就得光栅化两个8x8的块。这是非常低效的。

以上内容看似简单，但并不完美，实际的三角形光栅化是达不到理论峰值的（理论上总是假设所有的块都被全部填充）。请记住这一点。

vigilant-system:基于Larrabee的软件光栅化器