Chapter 10. Pipeline Optimization
流水线优化

金小刚 **Email: jin@cad.zju.edu.cn**
*浙江大学**CAD&CG**国家重点实验室*

简　介

- *“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.”* — Donald Knuth

- 绘制图象时，采用的是一个基 于流水线的结构。包含三个概念层：**application, geometry, rasterizer.**

- 流水线结构决定了其中最慢的一层永远是瓶颈，也是我们优化中最应重视的。

- 优化绘制流水线与优化流水线处理器类似,主要包含两步:

  (1). 找出流水线的瓶颈

  (2). 优化该层。如果性能仍未达到要求，则重复第一步。

- 瓶颈的位置是随不同帧动态变化的。在某一帧，由于有大量的细小三角形需要绘制，几何层可能是瓶颈；而在另一帧，由于三角形覆盖了屏幕的大半部分，光栅化层可能变成瓶颈。

利用流水线构造的另一个要素

- 当最慢的层不能再优化时，则尽量让其它层满负荷工作，直至速度接近最慢的层。

- 这不会改变系统的性能，因为最慢层的速度没有变化。例如，假设应用层为瓶颈，耗时**50**毫秒，其它层耗时**25**毫秒。这意味着在不改变绘制流水线速度的情况下，几何和光栅化层可以充分利用这**50**毫秒时间。我们可以采用更精致的光照明模型、加入阴影、加入反射等来提高绘制的真实程度。

- 把流水线划分成**application, geometry and rasterizing**三层只是概念上的划分。根据系统的结构，真正的流水线层可能与此不同。例如，假如应用层和几何层都在单个**CPU**上实现，则优化时应把他们当成一个层来看待。

- 在优化流水线过程中，应先尽量提高绘制速度，然后让不是瓶颈的层耗费与瓶颈层一样的时间。

- 在流水线优化时，

  永远记住**“Know Your Architecture”**。

找出瓶颈

- 优化过程是一个非常费时的过程，因此对图形流水线中的每一层都进行优化是不合适的。

(1). 如果对每一层都进行优化，整个绘制性能肯定能得到提高，但需要耗费程序员大量的时间，并有可能对绘制精度进行不必要的折衷。

(2). 优化每一层的另一个缺点是我们不能确定流水线中的哪些层不是瓶颈层。我们可以在不影响整体性能的情况下，充分利用这些非瓶颈层来得到更高的绘制质量。

优化流水线的第一步为找出瓶颈

- 对于任意给定的一帧，总是存在一个瓶颈，而这个瓶颈决定了绘制的速度。我们的目的是查到这个瓶颈并消除它。
- 瓶颈不能简单地通过对进程计时来找到。因为这样得到的是从绘制开始到结束，也即数据经过整个流水线的时间。
- 比较合理的方法为在源代码的某处开始计时（例如，在一清屏操作之后），并在该位置下次被执行到时终止计时。这样我们可以得到瓶颈层的执行时间，但并非瓶颈位置。
- 由于绘制一幅图象的总时间为瓶颈层的时间，而瓶颈可以是应用层、几何层、光栅化层或者层之间的通信（例如：**CPU**和加速卡之间的总线完全有可能是瓶颈）。我们可以对每个层进行计时，但这通常是很困难的，因为光栅化层和几何层通常在图形硬件中执行。

- 一种寻找瓶颈的方法为设立一系列测试，其中每次测试在某一时刻只影响一层。如果整个绘制时间受到影响，则我们已找到瓶颈。
- 我们可以保持测试层的负荷(**Workload**)不变，而减少其它层的负荷。如果系统的性能没有发生变化，则负荷保持不变的层即为瓶颈层。
- 下面我们将具体讲述测试方法。

测试应用层

- 如果所用的平台提供度量处理器负荷的工具，则该工具可以用来检测你的程序是否百分之百地利用了CPU的处理能力。

- 对于Unix操作系统，可用top或者osview命令来显示进程在CPU上的负荷。对于Windows操作系统，可用任务管理器来显示进程在CPU上的负荷。如果CPU的使用率接近100％，则程序为CPU受限的（CPU-Limited）。

- 但这也不是十分安全的。因为你可能在等待硬件完成一帧，而等待操作通常执行为一“**busy-wait**”操作。采用代码仿真器(**code profiler**)是一种更好的选择，它可以用来找出时间花在何处，分析代码的潜在问题或瓶颈所在 。

- 如果几何层也在**CPU**上执行，则在采用该技术时，我们无法辨别应用层和几何层。

- **AMD**提供了**CodeAnalyst**，可以用来分析和优化**CPU**上运行的代码。**Intel**提供了**VTune Analyzer**，可以用来分析时间花在何处。

测试CPU受限的另一更好方法

- 方法：将导致其它层做很少工作或不工作的数据向下发送。对于一些API来说，这可以通过用空驱动器（接受调用，但不做任何事情的驱动器）来实现。
- 因为我们没有用图形硬件，该方法可以得到整个程序在CPU上运行的上限。
- 而且，可以得到不在应用层运行的其它层的改进空间。

注意：采用空驱动器方法可能隐藏层通讯引起的瓶颈问题。

OpenGL中可采用的方法

- 把glVertex3fv和glNormal3fv调用用glColor3fv替换。这样并没有改变CPU的负荷，但几何和光栅化层的负荷得到大大减少。
- 几何层：不需要光照计算，不需要裁剪，不需要顶点和法向变换等。
- 光栅化层：没有从几何层接收到任何顶点，因此不会绘制任何物体。

- 如果性能没有得到提高，则程序为CPU受限的。(因为几何层和光栅化层没有做任何任务)

测试几何层

- 几何层是最难测试的层。这是因为若几何层的负荷发生变化，通常其它层的负荷也会发生改变。
- 只影响几何层负荷的一个参数为光源的类型和数目。如果把所有的光源去除或**disable**后，绘制的性能得到提高，则瓶颈为几何层，而程序为变换受限的（**transform-limited**）。
- 同理：光照、纹理坐标的生成、雾等也可以**disable**，因为这些操作也在几何层完成。
- 另一个方法：增加光源的数目，或把光源设成计算比较费时的类型（如聚光灯）。如果性能没有改变，则瓶颈不在几何层。

若硬件支持可编程Vertex Shader …

- 可以用一个只变换每个顶点的顶点程序，而不做任何光照计算，从而降低几何层的负荷，如果性能有提高，则瓶颈在几何层。

测试光栅层

- 对光栅层的测试只需简单地降低图象绘制窗口的分辨率，这样就相应减少了需要填充的象素数。如果总的绘制性能提高了，则可确定流水线是填充受限的，瓶颈就是光栅层。
- 此外，还可以通过关闭混合操作和深度缓冲进行测试，它们只影响光栅层。如果绘制时间减少，瓶颈就在光栅层。

性能测定

- 一种表示几何层性能的方法为：“每秒顶点(Vertices Per Second)”。
- 一种表示光栅层性能的方法为：“每秒象素(Pixels Per Second)”。
- 图形硬件生产商给出的通常是峰率，在实际应用过程中很难达到。
- 当涉及到CPU的性能测试时，会优先使用时钟周期计数器。大多数独立的测试平台(Benchmark)一般测定给定某一场景和屏幕分辨率下的实际帧率。

- 实际操作中要根据峰率所表示的具体含义确定测定方法。

最优化方法

- 确定瓶颈位置后，就可以对瓶颈所处层进行优化，从而提高整体性能。下面将介绍各个层的优化技术：有些优化技术是以牺牲质量来提高执行速度的，而有些是专门加速特定层的。
- 在优化过程中，如果实际性能已接近图形硬件厂商所宣称的性能指标，就不需要对流水线中的硬件加速部分进行优化，而需采用其它不同的加速技术。

优化应用层

- 应用层的优化可以通过提高代码的执行速度以及提高程序的存储访问速度来实现。
- 对不同**CPU**生产商，最优化技术也不尽相同，具体可以参考相应计算机的**CPU**使用手册。
- 下面我们介绍一些通用的优化技术，适用于大多数的**CPU**。

通用优化技术

- 打开编译器优化标志。如果可能的话，也可以尝试不同的编译器，因为不同的编译器是按不同的方式进行优化的。
- 编译器选项在某些情况下非常有用，但是一般还需要结合CPU体系结构进行优化处理。可以借助于代码模拟器找到代码的执行热点进行优化处理。
- 优化的基本规则就是尝试各种策略，尽可能多地尝试各种变化情况。
- 下面我们给出一些编写快速代码的方法和技巧，并对内存问题进行阐述。

编写快速代码的技巧

- 单指令多数据指令集(如**Intel** 的**SSE**和**SSE2**，**AMD**的**3DNow!)**可以用于许多情形并获得很好的性能。
- 尽可能地避免使用除法。新型的**CPU**通常提供一些快速计算低精度倒数和平方根倒数的特殊指令。
- 条件分支的开销非常高。尽管大部分处理器具有分支预测功能，但错误的分支预测仍会导致高的计算开销。因此要尽可能删除条件分支，消除分支的错误预测，提高速度。具体可以用条件移动指令**(Conditional Move)**，比如奔腾处理器上可以使用**CMOVxx**和**FCMOVxx**。
- 为了去掉过多的循环，可以将小的循环展开。但这会导致代码变大，降低高速缓冲存储器的性能。

编写快速代码的技巧（续）

- 在PC上，可以将经常用到的数据结构对齐为32字节的倍数，最佳利用高速缓冲线，从而提高总体性能。
- 许多数学函数，如**sin**、**cos**、**tan**、**exp**、**arcsin**等计算开销很高，在精度不高时只需使用麦克劳林或泰勒级数的前几项。
- 对于经常调用的小函数使用内联代码。
- 在适当的时候降低实型数据的精度，尽可能使用低精度。
- 对同一算法尝试不同的代码编写方式，如使用索引避免指针增加等。
- 尽可能使用**restrict**和**const**，编译器可以用它们进行优化。
- 虚函数方法、动态转换、（继承）构造，以及按值传递都会对效率造成一定的影响。

内存问题

- 当涉及系统内存程序时，不要做任何假设。例如，如果知道复制方向，或者复制源和复制目标没有重叠时，使用现有的最大寄存器的汇编循环就是一种最快的复制方式。
- 在代码中连续访问的存储内容在内存中也应该连续存储。
- 对数据结构尝试不同的组织方式。
- 要尽可能利用体系结构中的高速缓冲存储器。

内存问题（续）

- 高速缓冲存储器的预取。对于好的代码性能，要执行的代码和下一个要访问的存储内容最好都在高速缓冲存储器中。某些体系结构提供预取指令，如**APARC**的**prefetch, PowerPC**的**DCBT**，**AMD Athlons**的**PREFETCH**和**PREFETCHW**, 奔**3**和奔**4**的**PrefetchNTA**和**PrefetchTx**。奔**4**具有一种特殊的逻辑单元，可以识别重复的存储访问并进行相应的预取操作。
- 尽量避免间接指针、跳转，以及函数调用，因为它们很容易降低**CPU**中高速缓冲的性能。
- 在一些系统中，函数**malloc()**和**free()**执行的速度比较慢。可以在开始时分配一块较大的内存池，然后使用自己的分配和释放程序来处理这个内存池。

优化几何层

- 几何层主要负责变换、光照、裁剪、投影，以及屏幕映射。
- 变换和光照过程比较容易优化，而剩余的部分优化就比较困难了，甚至不可能优化。

连接和压缩图元

- 索引顶点缓冲器是一种最快的图形加速器数据提供方式。**AGP**内存、**DMA pull**，以及视频内存中的数据存储都是影响性能的一些因素，而变换、光照、裁剪、投影，以及屏幕映射都可以从紧凑的数据存储中受益。

- 索引顶点缓冲器的效率非常高，不同的方案会有不同的效率，某些顶点缓冲器方案对于特定的机器来说具有最好的性能，因此必须“**know your hardware**”。此外在内存的分配与维护、最佳的顶点记录大小以及如何使用栅栏来避免数据复制等方面也存在很多变化。

- 以压缩格式存储顶点数据是另外一种可以减少数据阻塞的方法，同时还可以降低应用程序方面的内存使用。

光 照

- 可以用很多方法对光照计算进行优化：首先，应该考虑所使用的光源类型。并不是所有的多边形都需要光照，有时候一个模型仅需要纹理贴图，或者是具有顶点颜色的纹理贴图，或者只需要在顶点处具有简单的颜色；
- 如果必须使用光源，那么平行光源要比点光源的速度快，而点光源要比聚光源的速度快；
- 光源的数量也会影响几何层的性能。某些图形加速器的优化仅仅是针对只使用一个光源的光照而言的；光源使用的数量越多，速度就越慢；双面光照要比单面光照的计算开销要大；
- 当对光源使用距离衰减时，可以根据物体到光源的距离，在每个物体的基础上进行光源的关闭和打开，这样做非常有用且很难察觉。

- 第二，在一些API中，可以使用非局部视点(Nonlocal Viewer)，它只对镜面高光的计算和环境贴图的计算产生影响，对视图本身没有影响。

- 其基本思想就是认为相机位于无限远处，从而避免从视点到光照顶点之间的向量计算并对之进行归一化处理，使用这个特征会导致镜面高光位置的微小偏移，但是通常很难觉察得到。

- Direct3D提出了非局部视点概念的一种变化形式：利用点光源建立一个从光源自身到顶点网格中心的方向向量，然后将这个向量用于网格的所有顶点。这样光源就成为一种平行光源，从而可以简化光照计算。

法向归一化的处理

- 第三，在计算光照时，对模型法向归一化的处理。

◎ 对在绘制阶段没有进行比例缩放变换的模型应该以预处理的方式进行法向归一化。在完成对模型的所有变换后，对法向通过比例缩放来进行归一化；

◎ 应将法线已经归一化的信息通知给相关的图形**API,**以确保图形引擎不再浪费时间进行归一化处理。

◎ 对不同的体系结构，归一化的代价也有所不同。

- 第四，如果不需要多边形双面的光照，就可以关闭这个特征。
- 第五，如果把光源或材质的镜面反射分量设为(0,0,0)，就可以避免光照方程中的高光计算，这部分的计算开销相当高。
- 第六，采用预着色法。相对于几何物体来说，如果光源是静止的，同时多边形数据没有包含镜面反射参数的材质，就可以将漫反射和环境光照预先计算出来，并作为颜色存储到顶点中。阴影处理：为了增加真实感，可以通过光源与顶点间的可见性判断来获得简单的阴影效果，然后再与环境光照和漫反射光照结合起来。最终就可以关闭这些物体的光照计算。

Vertex Shader优化

- Vertex Shader是一个很小的程序，可以将其发送到几何层的单元中，然后针对发送到图形流水线中的每个顶点，由这个单元来执行该程序。
- 对这些程序进行优化的一个明显的方法就是尽量使这些程序比较简短，即尽可能使用较少的指令。
- 此外，由于切换Shader程序也会耗费一定的时间，所以要尽可能按照Vertex Shader对物体进行分组。
- 现今的图形硬件通常含有多个Vertex Shader单元，可以让它们以并行方式运行，从而提高整体性能。

优化光栅层

- 对封闭的物体和无法看到背面的物体应该打开背面剔除开关，这样可以将需要光栅处理的三角形数量减少近**50%**。需要记住的是，虽然背面剔除可以减少不必要的图元处理，但是需要花费一定量的计算来判断图元是否朝向视点。

- 另一种不影响绘制质量的优化技术是在特定的时间关闭Z缓冲器。诸如多边形对齐的**BSP**树之类的算法就不需要Z缓冲器；此外，还可以使用局部**BSP**树在绘制过程中对它进行实时合并。

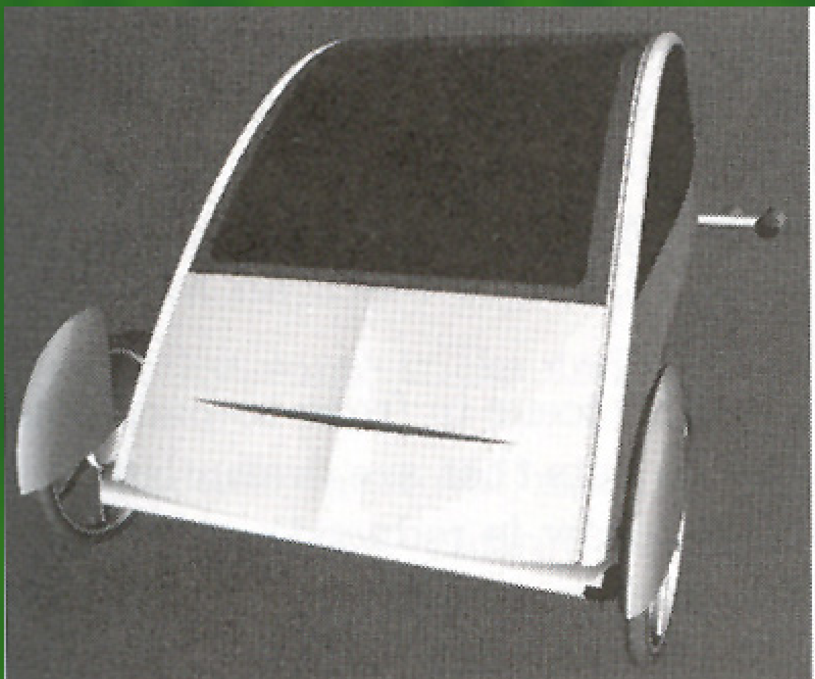
- 如果可以保证屏幕上的每个象素都被某个物体覆盖，就不需要清除颜色缓冲器。
- 还可以通过牺牲一个比特的深度精度避免对Z缓冲器进行清除。诸如**HyperZ**这样的技术可以节省在硬件中**Z**缓冲器的清除时间。
- 使用内部的纹理和象素格式，避免从一种格式转换到另一种格式。当需要更新纹理时，可以使用**glTexSubImage2D**代替**glTexImage2D**，这样就可以避免内存的分配与回收问题。
- 使用纹理压缩，加快发送到纹理内存的速度。如果可以的话最好进行离线压缩，这样可以对纹理质量进行提前检查。压缩纹理还可以提高高速缓存的效率，从而提高系统性能。

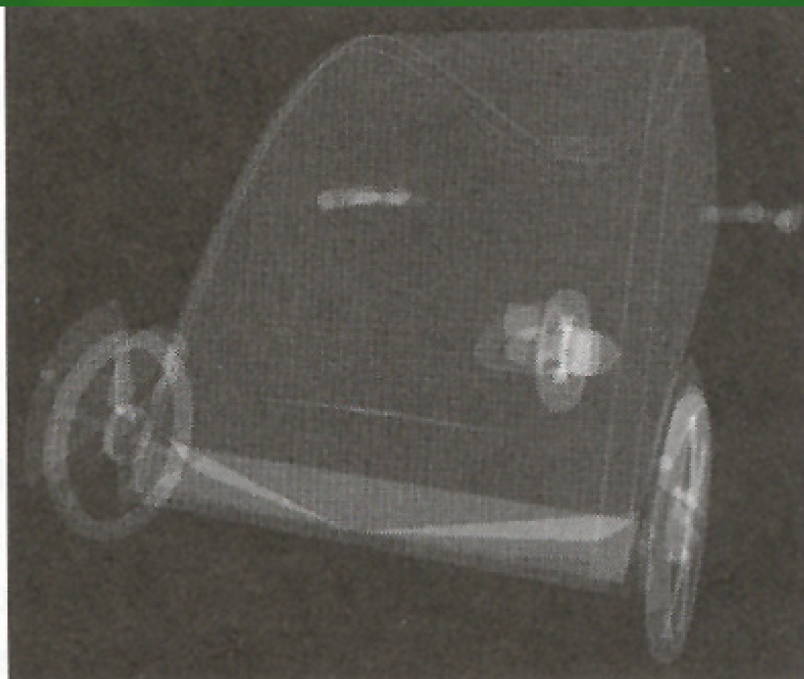
- 降低绘制窗口的图象分辨率是一种提高性能、牺牲质量的常用方法。具体可以通过减少图象的实际绘制量来完成。当准备生成图象时，可以用线性插值滤波器将较小的图象放大到所期望的尺寸。

- 在一些硬件系统上（**SGI InfiniteReality**），有硬件支持的用来测量图形流水中不同层工作负载的流水测量设备。流水测量设备在精确定时的生成和任务优化的简化方面具有非常重要的作用。

- 为了更好的理解一个程序的行为，特别是光栅层的负荷，可以对深度复杂度(即一个象素被接触的次数)，进行可视化。

深度复杂度的可视化





(a). 三轮车模型

(b). 深度复杂度的可视化表示，象素越亮，被重写的次数越多。

- 深度复杂度图象可以简单地通过使用一种类似于**OpenGL**中的**glBlendFunc(GL_ONE,GL_ONE)**的函数调用来生成，在此期间关闭**Z**缓冲器。

  (1) 将图象清除成黑色

  (2) 对场景中所有物体，均用颜色(0,0,1)进行绘制

- 这种混合函数设置的效果就是对每个绘制的图元来说，可以将写入的象素值增加(0,0,1)。深度复杂度为**0**的象素为黑色，深度复杂度为**255**的是纯蓝色。

- 可以使用双通路方法对那些通过或者没通过Z缓冲器深度测试的象素进行计数。
- 在第一个通路中激活Z缓冲器，并使用上述方法对通过深度测试的象素进行计数。
- 对那些没有通过深度测试的象素的计数，既可以通过增加模板缓冲来实现；也可以通过关闭Z缓冲器获得深度复杂度后，减去第一个通路的结果来实现。
- 这些方法可以用来确定：

  (1)场景中深度复杂度的平均值、最小值和最大值

  (2)每个图元的象素数目

  (3)通过或者没有通过深度测试的象素数目
- 通过深度复杂度可以知道每个象素覆盖的表面数量。

- 按照从前到后的顺序绘制场景可以提高性能，对于被遮挡的物体来说，可以避免更多的复杂填充操作。
- 在有些硬件上， 在光栅层存在一种简单形式的遮挡裁剪功能，因此一种大致的排序对此会有很大作用。
- 如果排序很困难，则可以使用复杂的**Pixel Shader**。首先将物体表面绘制到**Z**缓冲器中，在**Z**缓冲器中完成整个场景的绘制后，使用**Pixel Shader**绘制所有场景表面。由于象素**Fragment**在进入**Pixel Shader**之前，遮挡裁剪硬件会将不可见的象素**Fragment**筛选掉，从而可以节省大量的填充率。

- 如果上述方法都不行，就需在速度与质量之间寻求一种平衡，关闭具有速度补偿能力的所有特征：计算开销大的纹理过滤、雾化、混合、线条绘制、深度测试、模板缓冲操作，以及多采样（反走样）。

- 此外，还需了解当前的体系结构。

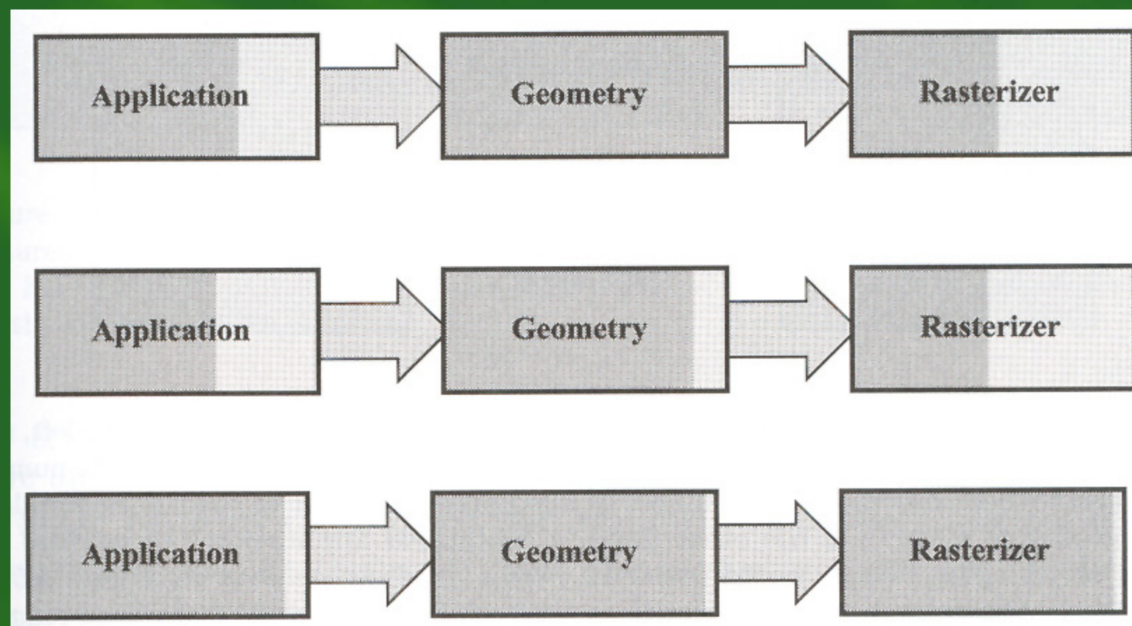
总体优化

- 尽可能减少通过流水线的绘制单元数量，具体可以通过模型简化或裁剪技术。
- 尽可能选择低精度的顶点、法线、颜色和纹理坐标。
- 在特定的应用环境中，应该对需要绘制的模型进行预处理。（如自相交、凹多边形）
- 关闭那些不再使用的特性。那些已经激活却不使用的特征可能会降低系统的性能。
- 尽可能减少**API**调用数量。
- 按照相近的绘制状态对物体进行分组，从而将状态变化最小化。
- 寻找并使用快速路径，并按照组大小为当前的体系结构建立最优化的图元。所谓快速路径就是一条通过图形硬件且经过高度优化的路径。
- 要尽可能确保所有的纹理都在纹理内存中，从而避免数据交换。

- 要将三维操作与二维操作分开，因为在两者之前进行切换需要很大的开销。
- 在设置阶段调用**glGet()**，而在运行阶段要避免调用**glGet()**。
- 要确保充分利用图形硬件所具有的优势，有些图形硬件对深度缓冲器和颜色缓冲器同时进行了优化。
- 当顶点和很多骨架相互混合时，最好预先计算出混合矩阵并将其通过流水线进行发送。
- 多通路所需要的开销通常比较高，可以使用多重纹理贴图或**Pixel Shader**，尽可能避免过多的通路。
- 帧缓冲器读取的开销通常比较高，最好避免使用。
- 如果显示列表对体系结构有益，就使用显示列表。
- 坚持“越少越好”原则，即从应用程序到图形加速卡之间经过的数据越少，系统的运行速度就越快。

图形流水线的平衡

- 为了观察流水优化的效果，可以在单缓冲模式下测定每个帧画面的绘制时间。
- 打开双缓冲器的话，由于只有在与监视器电子束同步时才可能发生缓冲器交换。这其中就可能会有空闲时间，我们可以加以利用提高绘制质量，这就是流水线平衡的目的。
- 只要不超过监视器刷新时间t的下一个倍数，就可以由瓶颈层来完成多余的工作，提高绘制图象的质量。
- 在不降低性能的条件下，要充分利用非瓶颈层，使其多做工作。



第一行：系统只使用了一个缓冲器，其中几何层的使用率为100%，应用层的使用率为75%。光栅层的使用率为50%。

中间行：启用双缓冲，意味着绘制必须与监视器的刷新频率同步，因此所有层的空闲时间都增加了（如果绘制时间不能与刷新频率保持同步）。

最后一行：平衡之后的管线。利用了所有层的空闲时间，却没有对帧率产生影响。

瓶颈和非瓶颈层空闲时间的使用

- 增加三角形数目（这会影响所有层）。
- 如果应用层处于空闲，那么尽量多做如下工作。
  (1). 计算更真实的动画;
  (2). 执行更精确的碰撞检测;
  (3). 使用更复杂的加速算法;
  (4). 执行其他的类似任务。
- 如果几何层空闲，那么可以使用较多的光源和开销较大的光源类型和顶点着色器等。

瓶颈和非瓶颈层空闲时间的使用(续)

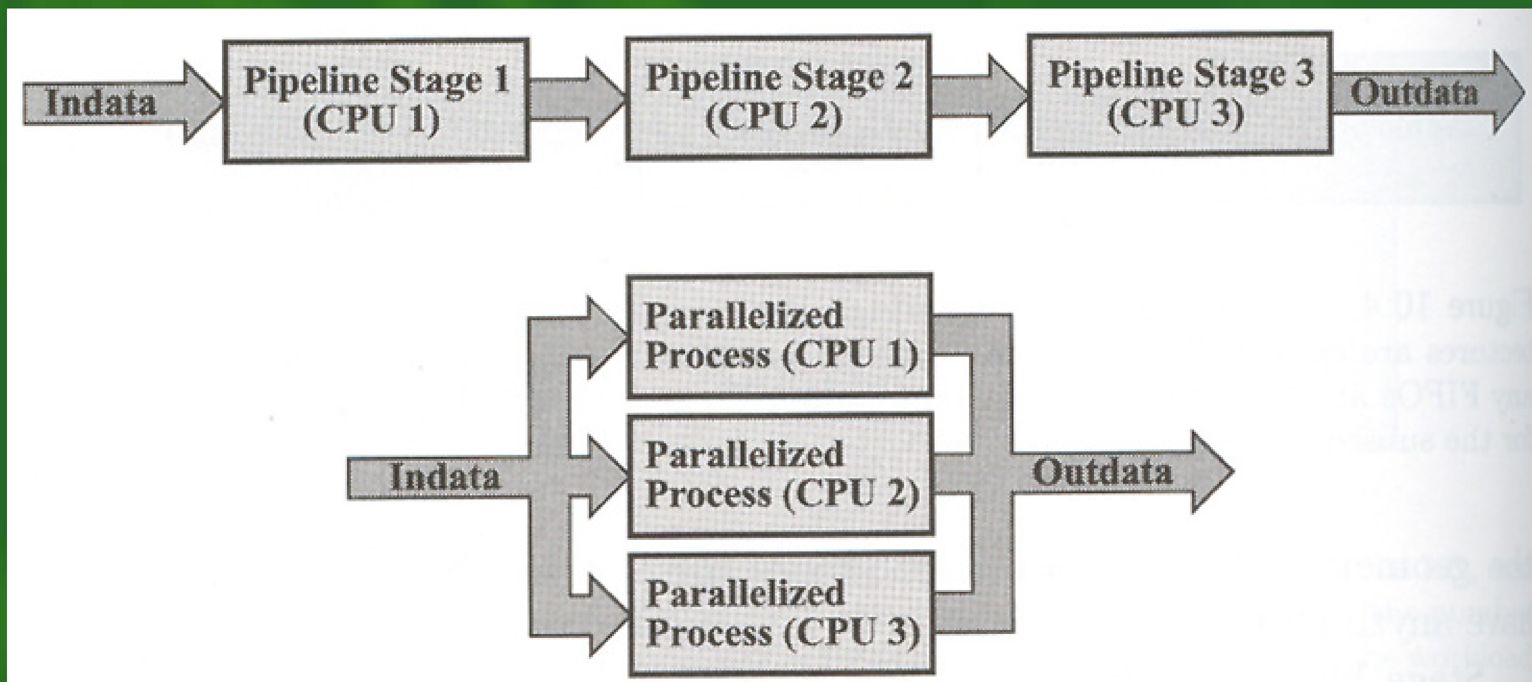
- 如果光栅层空闲，那么可以使用开销较大的纹理过滤、雾化、混合、**Pixel Shader**等。
- 在清屏后，不要立即使用过多的图形命令来填充图形流水线。这是因为图形硬件已经忙于清除缓冲器（这需要花费大量时间）。因此，在清屏之后，最好做一些应用相关的作业。
- 只要填充象素的数目没有改变，填充受限的应用程序就能很容易地增加多边形的数目，而不会影响光栅层的性能。
- 如果绘制不是填充受限的，就可以对绘制图象所在的窗口进行放大。

- 大多数流水线都具有先进先出队列，其基本思想就是用**FIFO**原则对几何层和光栅层的作业进行排序。也就是说，即使流水线中的下一个层还没准备好，当前层依然可以继续自己的作业，同时将相关结果先存储起来，等待下一层准备就绪。这种缓冲机制可以对流水线的平衡性进行平滑处理，从而使较小的瓶颈变得不可见。
- 在层中可能会出现空闲或阻塞状态。
- 在同一画面期间，瓶颈位置也会随着时间的变化而变化，在优化期间需要对此进行考虑。
- 在加载流水平衡时，还要考虑不同画面之间的加载随着时间而变化。可以测量绘制一幅画面所需要的时间，如果测量的时间比用户定义的最小阈值要小，则可增加工作负荷，反之降低工作负荷。

多处理器技术

- 在大的范围内，可以将多处理器计算机分为消息传递体系结构和共享存储器多处理器。
- 实时图形学应用中使用多处理器的两种方法：多处理器流水线方法，也称时域并行方法；并行处理方法，也称空间并行方法。

使用多处理器的两种方法



上图为多处理器流水线，使用了3个处理器。

下图所示为3个CPU上并行执行作业。

多处理器流水线方法

- 流水线是一种将任务分割为以并行方式运行的流水阶段来加速任务执行的方法，一个流水阶段的结果可以传递到下一个流水阶段。
- 目前，对于单**CPU**系统，可以使用流水以并行方式执行应用层、几何层和光栅层。在主机上使用多处理器时，也可以使用这种技术，在这种情况下，就称为多处理器流水线方法。
- 由于通常在硬件中进行光栅化处理，因此即使在主机上使用多余的处理器，也无法对该层流水化。应用层通常会使用额外的处理器；如果几何层不是在硬件中实现，那么也可以使用这些额外处理器。

应用层流水线化

- 应用层可以再划分为3个层：APP、CULL、DRAW。
- APP层是流水中的第一个层，它控制着其它层，程序员可以在这个层中添加额外的代码，比如碰撞检测，这个层还可以对视点进行更新。
- CULL层主要执行如下作业：

  (1). 遍历场景图并进行分层视域四棱锥裁剪。

  (2). 细节层次的选取。

  (3). 状态分类。（为了使得状态变化最小化）

  (4). 将所有需要绘制的物体生成一个简单的列表。
- DRAW层从CULL层取出列表并在这个列表中发送所有图形调用。

- 这种技术的优点在于可以提高吞吐量，即提高绘制速度，不足之处就是与并行处理相比，延迟时间增加了。
- 延迟时间指从用户操作开始到图象最终形成所花费的时间。
- 帧率是指每秒钟可以显示的画面数量。（注意：不要混淆帧率和延迟时间）。

减少延迟时间的技术

- 在APP层的末端更新视点和其他影响延迟时间的关键参数。
- 让CULL和DRAW重叠执行，只要准备好绘制所需的相关事项，就可以将CULL的结果发送到DRAW中。在这些层之间必须存在某种缓冲（如FIFO）。当缓冲器满时，CULL必须停止；而缓冲器空时，DRAW必须停止。

并行处理方法

- 多处理器流水技术使得延迟时间变得更长了，对有些应用如飞行模拟器，快速游戏等是很难接受的。
- 如果存在多个处理器，就可以对代码进行并行化处理，从而缩短延迟时间。前提是程序必须有并行操作特征。

算法的并行化处理方法

- 使用静态赋值，假设有n个处理器，可以将一个总的作业分成n个小的作业包，每个处理器负责一个作业包，并行执行。当所有处理器完成各自的作业包后，对处理结果进行合并。这要求所有的工作负载是可以预测的。
- 动态分配算法可以适应不同的工作负载：建立一个或多个作业池，作业生成后就放置到作业池中。当CPU完成自己的当前作业时，就可以从作业队列中取得一个或多个作业。必须保证一个CPU只能取得一个特定的作业且不影响系统整体性能。如果作业比较大，则维持队列的开销就相对小，但会导致系统的不平衡性，从而降低系统性能。

Further Reading and Resources

- 《**Game Programming Gems**》系列丛书（如何高效利用图形硬件）
- 《**Graphics Gems**》（优化算法）