游戏画面卡顿问题的检测与解决对策

Cem Cebenoyan
NVIDIA开发者技术总监

画面卡顿 – 游戏体验的杀手

- 当别人问你:
  - “游戏每隔几秒就卡一下...”
  - “测出来的帧数很高，但总是觉得不流畅...”
  - “感觉人物动画一顿一顿的...”
  - “有时输入延迟很严重...”
  - ……
- 你知道这是卡顿的问题，可是
  - 哪儿出错了?
  - 严重损害游戏体验
  - 难以查出卡顿的根源，更加难以排除

在本讲座中，

- 我们将讨论：
  - 游戏图形渲染中最常见的卡顿现象
  - 鉴别卡顿根源的方法
  - 消除卡顿的方案

- 但不会涉及：
  - 由网络/磁盘IO，音效，及其它非图形因素造成的卡顿

议程

- 常见卡顿起因一瞥
- 卡顿的诊断
- 起因分析与应对方案
- 垂直同步, SLI及其它情况

常见卡顿起因一瞥

卡顿的各种表象

- 帧率停滞(frame hitching)
  - 表象：每隔一段时间，帧率突然下降或停滞，然后恢复正常
  - 可能起因：shader编译，资源更新，或显存分页交换
- 微型卡顿(micro-stuttering)
  - 表象：统计出的帧率很高，但画面整体感觉不流畅
  - 可能起因：帧与帧的持续时间极度不均匀
- 计时偏差(timing discrepancy)
  - 表象：帧率一切正常，但镜头、动画和模拟等感觉忽快忽慢
  - 可能起因：不合理测量得来的时间间隔，以及对缓冲帧数的不正确处理。

最常见的5种引起卡顿的原因

1. *Shader编译*

- 驱动需要把D3D汇编转化成GPU机器指令，这一过程可能会带来停顿

2. *显存溢出(video memory oversubscription)*

- 当显存用尽时，会出现大量系统内存与显存间的分页交换(paging)，造成拖慢

3. *资源管理不善*

- 在运行时创建，销毁和更新资源容易造成性能颠簸

4. *缓冲帧(queued frames)利用不当*

- CPU与GPU间需要用缓冲帧来解决负载平衡的问题，但对缓冲帧的不合理设置会造成微型卡顿

5. *查询(query)使用不当*

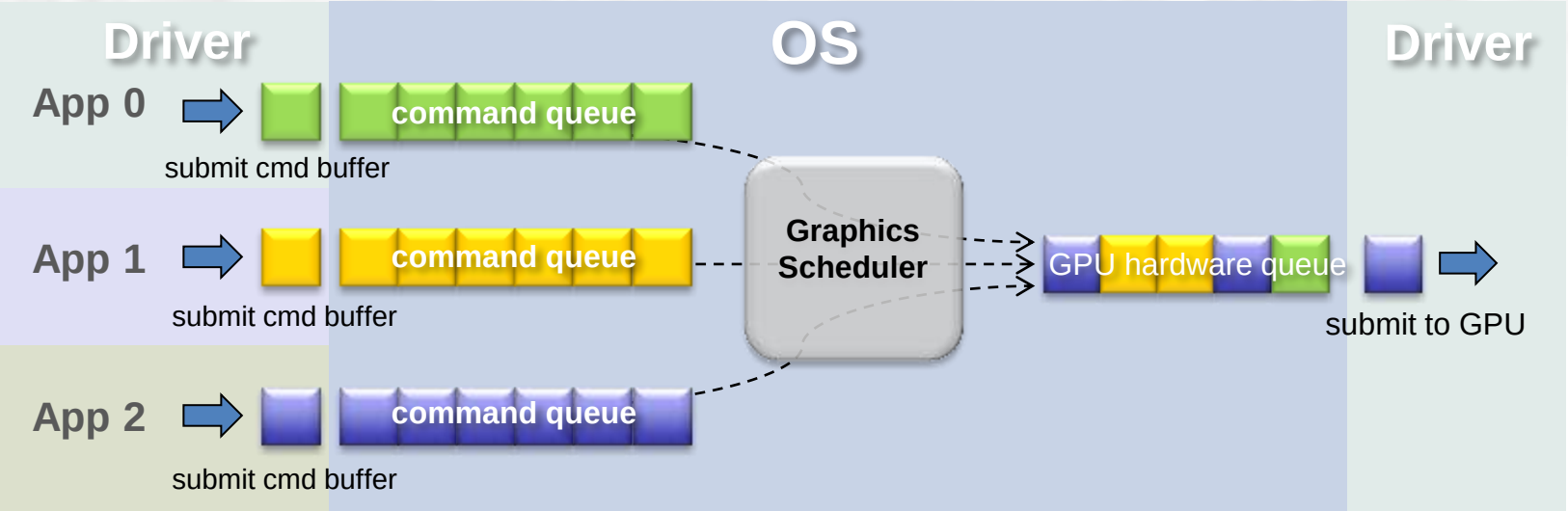
- 事件查询(event query)与遮挡查询(occlusion query)可能会改变驱动的行为，有时会造成渲染管线阻塞

卡顿的诊断

卡顿的定性

- 卡顿的定性是一项艰巨的工作
  - 可能只出现在某一特定环境下，或某个特定的硬件上
  - 难以取得有效的数据来进行分析
- 需要结合多种工具和多项试验来进行定性
- 在讨论细节之前，需要理解一些基础知识：
  - CPU/GPU的通讯方式
  - Windows显示驱动模型(Display Driver Model, WDDM)

预备知识：CPU/GPU的通讯方式

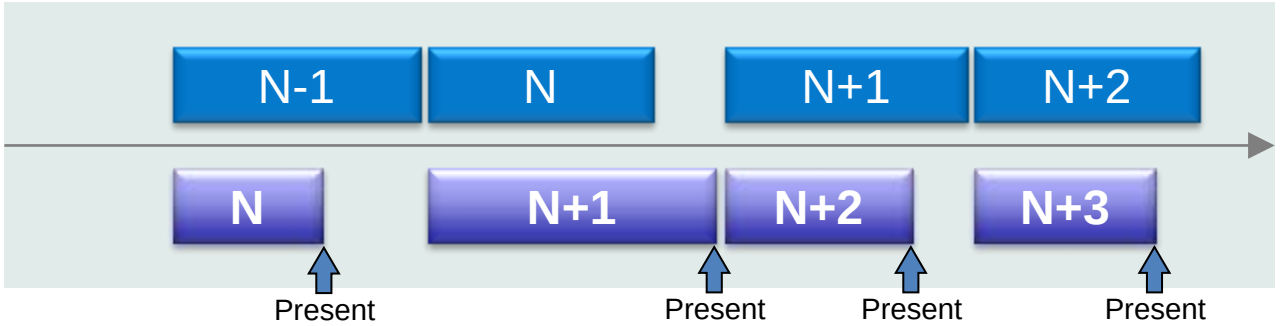


- 每个D3D设备(D3D device)都有一个图形上下文(graphics context)
  - 负责维护一个*命令队列*(*command queue*)
  - 驱动把接收到的API调用放入命令缓冲(*command buffers*)，在合适的时候提交到command queue中
- GPU的硬件资源由多个应用程序共享
  - 操作系统的全局*图形调度器*(*graphics scheduler*) 从多个command queue中选取合适的命令包，放入*GPU硬件队列*(*GPU hardware queue*)
  - GPU按顺序处理命令包，完成后，从队列中删除

预备知识：CPU/GPU的通讯方式(续)

- Command buffer的提交
  - 通常，驱动会在Present函数调用后开始提交
  - 偶尔会在其它地方进行提交

- 帧延迟(frame latency)
  - 在没有帧缓冲(frame queuing)时，GPU比CPU慢一帧



  - 实际上，驱动可能在提交之前最多缓冲3帧(4帧延迟)

预备知识：WDDM

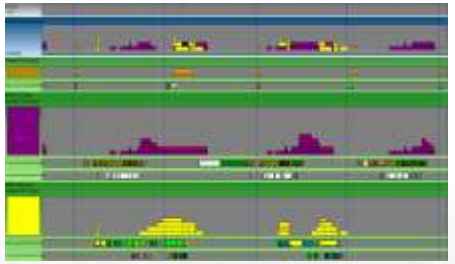
- Windows显示驱动模型
  - 从Vista开始使用
  - 将显存纳入虚拟内存体系，改善的容错性，由操作系统进行图形任务调度，等等
- 实际上有两个驱动
  - UMD：用户模式驱动
    与应用程序和D3D runtime协同工作
    负责创建和提交command buffers
  - KMD：内核模式驱动
    运行于操作系统的内核模式下
    为操作系统提供管理硬件资源的接口
  - 操作系统负责操作command queues，联结UMD和KMD

卡顿诊断所需的工具

- Fraps
  - 记录帧率
  - 快速收集统计数据

- Nsight
  - 静态和动态分析
  - GPU管线调查

- GPUView
  - 深层次分析

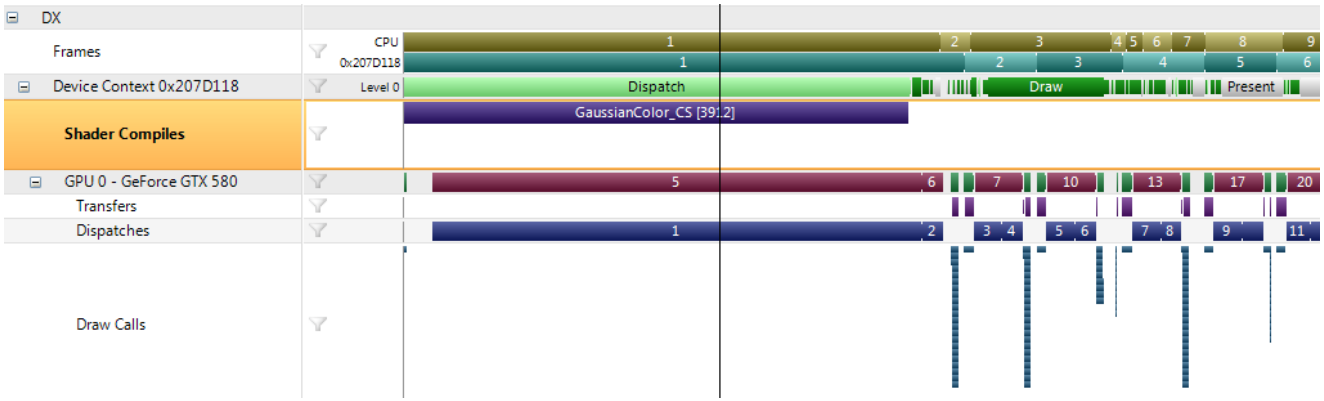


帧率停滞(Framerate Hitching)的诊断

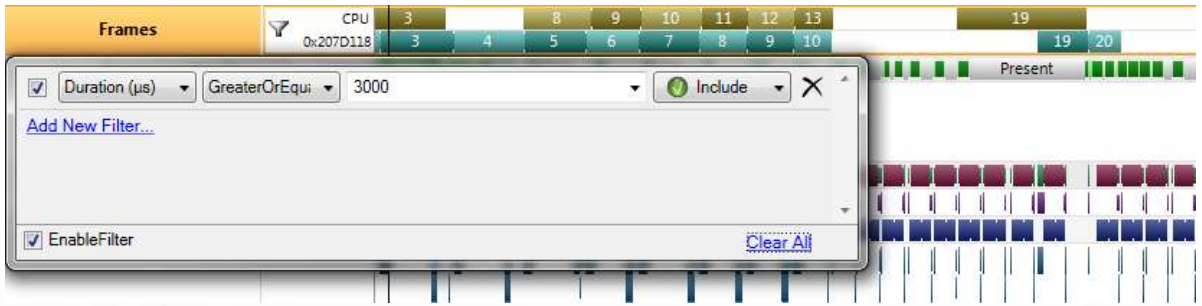
- 现象
  - 每隔一段时间，帧率突然下降或停滞，然后恢复正常
- 先进行帧率的记录
  - 在游戏引擎中加入测时代码，记录每一帧的持续时间 (每两个present调用之间)
  - 或者，使用Fraps的*frametimes*功能进行记录
  - 可以很容易从结果中找到停滞的帧
- 检查停滞帧的问题：
  - 是否有创建shader? 有新材质加载?
  - 是否首次引用了一块较大的资源? (texture, render target, buffers, etc.)
  - 渲染线程是不是因为资源更新被阻塞了?
  - CPU或GPU处理了不同其它帧的大量任务?

帧率停滞的诊断(续1)

- Nsight可以用来检查停滞帧
  - 使用“Trace Application”来录制有问题的游戏部分



  - 可以在时间线上看到shader的编译情况



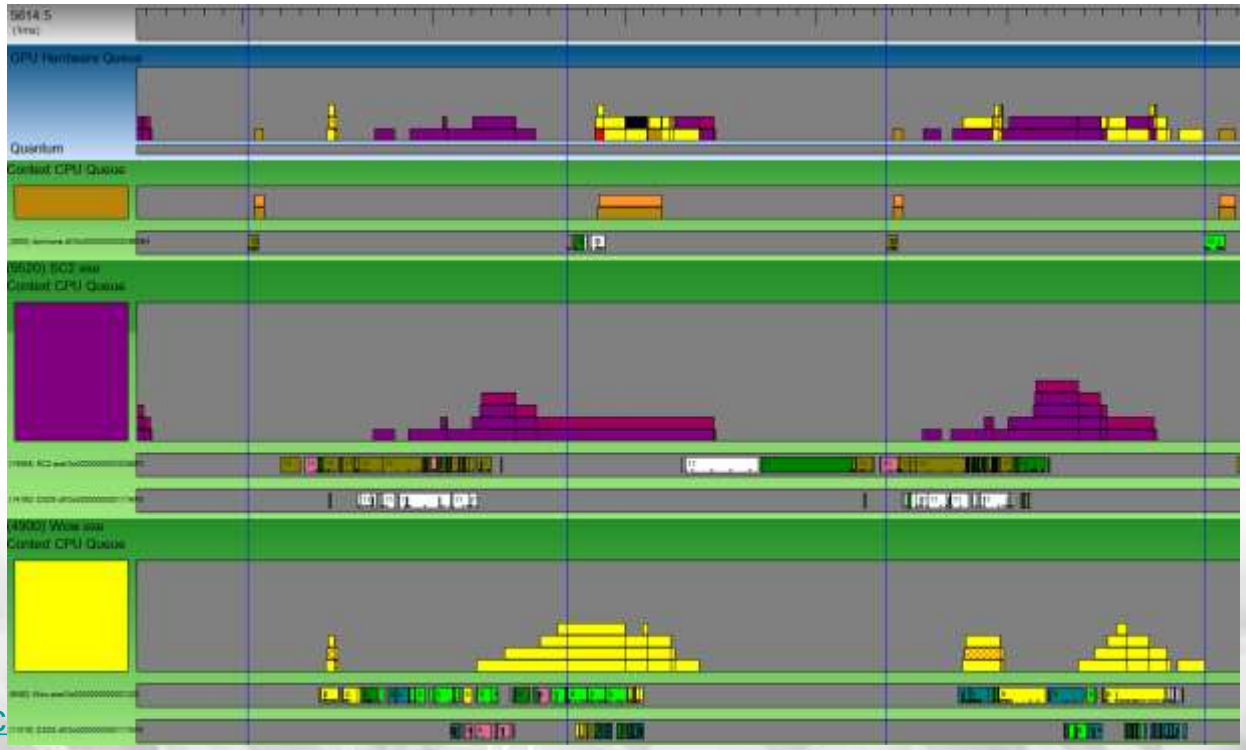
  - 可使用过滤功能来选出有问题的帧，或者特别关注的API调用

帧率停滞的诊断(续2)

- 对常见的可能起因进行实验
  - 如果发现停滞帧中有shader编译存在, 去除这些shader
  - 如果发现停滞帧中首次引用某种大资源, 去除这些资源
  - 如果发现停滞帧中有阻塞现象的Lock*, Map*, Update*函数, 去除这些函数调用
- 实验结果可以显示卡顿的起因
  - Shader编译的问题, 资源管理的问题, 等等.
  - 我们会在后面详细讨论每一种起因

帧率停滞诊断(续3)

- GPUView是一个更高级的检测工具
  - *[优势]* 海量信息：所有command queues和GPU hardware queue的详细内容，每个包的内容和时间，等等
    在后台录制时不会对应用程序造成影响
  - *[缺点]* 对新用户来说是个很大的挑战
    数据量太大
  - GPUView可以监测整个系统．例如：可以帮助检查为何某个游戏的Present调用被windows桌面阻塞?

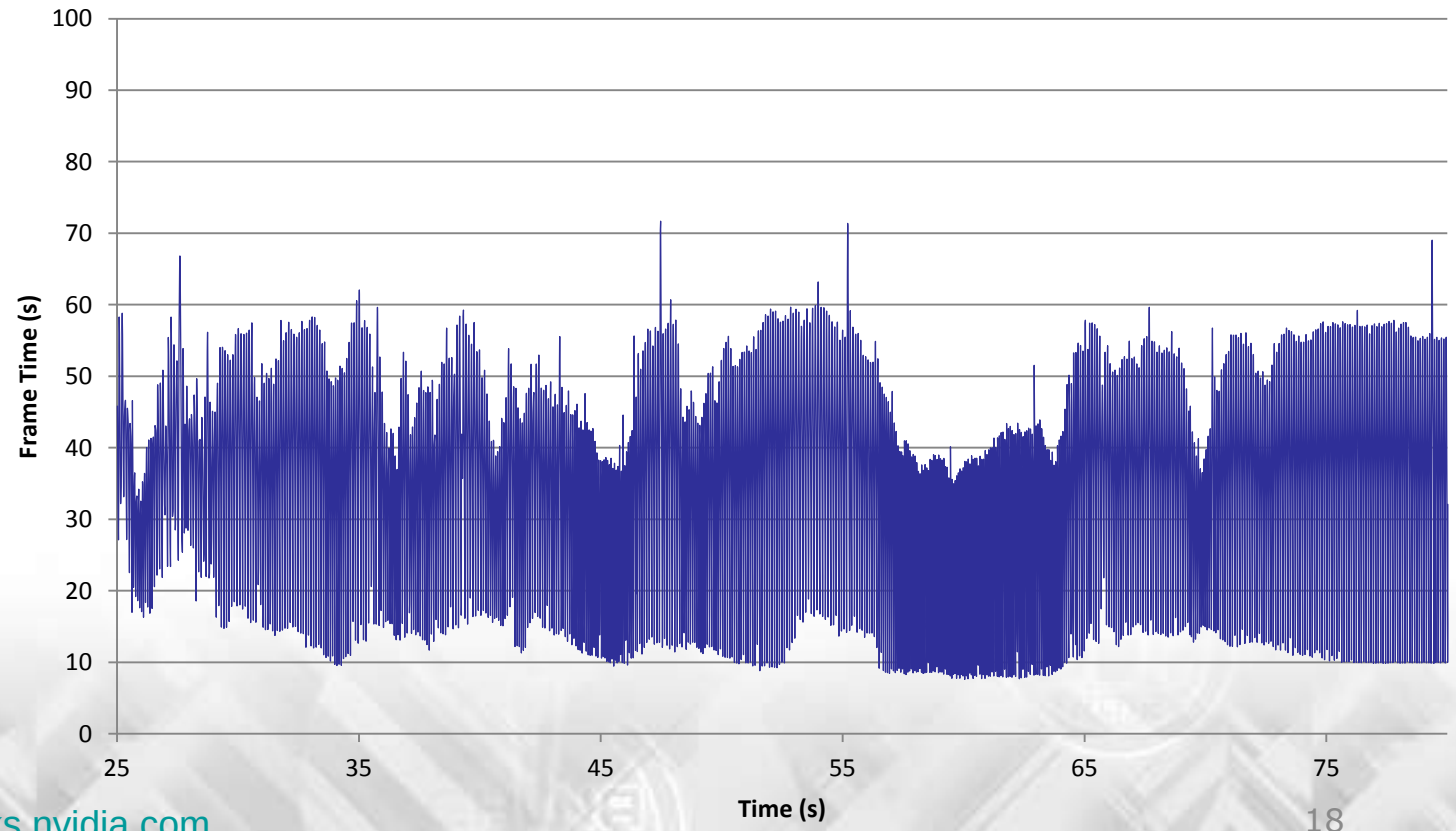


gameworks.nvidia.c

Courtesy of
Matt Fisher

微型卡顿(Micro-Stuttering)的诊断

- 现象
  - 统计出的帧率很高, 但画面整体感觉不流畅
- 问题: 帧时间(frametime)严重不均



微型卡顿的诊断(续1)

- 可能的起因
  - 负载不均：AI、动画等任务隔帧执行或多帧执行一次，而不是每帧都在CPU上执行一次
  - 游戏引擎通过某些方式限制缓冲帧(queued frames)的数量，造成驱动无法通过调度来隐藏不均匀的Present调用
  - 大量的资源更新
  - 显存溢出后，频繁发生显存分页交换(paging)

微型卡顿的诊断(续2)

- 使用Nsight或GPUView进行监测
  - Nsight: 查看GPU frametime一栏
  - GPUView: 查看GPU hardware queue部分

- 检查以下问题:
  - 帧与帧的CPU负载是否严重不均匀?
  - 游戏引擎是否通过某些途径来限制缓冲帧的数量?

  - CPU帧负载不均 + 无缓冲帧 -> 微型卡顿

微型卡顿的诊断(续3)

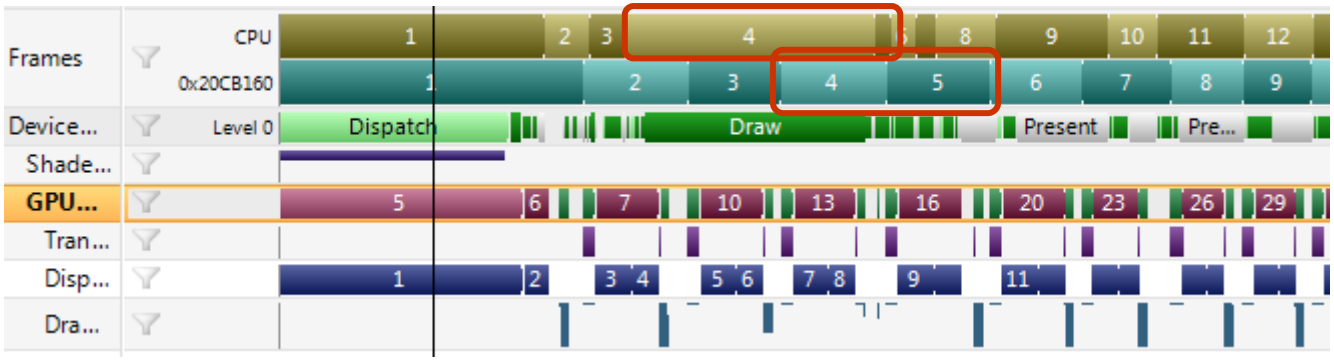
- 检查资源更新时, CPU是否被长时间阻塞
  - 对Lock*, Map*和StretchRect函数, 用测时代码检测其阻塞CPU的时间. 对这些函数来说, 如果指定的待更新资源正在被GPU使用, CPU可能被阻塞到GPU使用结束
  - CPU长时间被阻塞 + 缓冲帧->微型卡顿
- 在运行时估算显存的使用量
  - 结合WMI接口和游戏引擎自身的显存统计
  - 显存大幅度溢出 -> 频繁的显存页交换 ->微型卡顿
  - 频繁的显存页交换只会发生在显存大幅度溢出后, 少量的溢出不会带来严重的卡顿

计时偏差(Timing Discrepancy)的诊断

- 现象
  - 帧率一切正常，但镜头、动画和模拟等部分感觉忽快忽慢
- 可能的起因
  - 游戏引擎的计时方式有误或有缺陷，用于镜头和动画更新等的时间间隔不准确

计时偏差的诊断(续1)

- 检查游戏引擎的计时系统
  - 是不是用了每两个Present调用的时间间隔来作为下一帧动画更新的步进时间?
  - 如果是, 用Nsight查看时间线



  - 如上图, CPU的步进时间可能和GPU的步进时间差别很大 -> 动画卡顿
  - 在这种情况下, CPU一端的Present到Present时间间隔并非真正的两帧之间的时间间隔!

起因分析与应对方案

起因

- 卡顿的5种最常见起因:
  1. *Shader编译*
  2. *显存溢出(video memory oversubscription)*
  3. *资源管理不善*
  4. *缓冲帧(queued frames)利用不当*
  5. *查询(query)使用不当*

Shader编译的基本知识

- 驱动为何在运行时编译shader?
  - D3D汇编必须转化为GPU机器指令才能够执行
  - 每一代GPU的指令集都大相径庭

- 何时编译? 如何编译?
  - 在调用Create***Shader后
  - 驱动中的编译器生成机器指令, 存储下来为后续使用
  - 对于复杂的shader, 驱动可能会先生成一套优化较少的指令, 然后在晚些时候替换上一套优化的指令

Shader编译的基本知识(续)

- 驱动需要多长时间编译?
  - 依shader复杂度的不同, 需要几十到几千毫秒
- 有什么办法预编译后存盘么?
  - 没有
- 每个shader只编译一次?
  - 不一定. 有些D3D9的状态改变会引发shader被重新编译

状态相关编译(State Dependent Recompile)

- D3D9的状态与GPU硬件的状态缺乏良好的对应关系
  - 很多D3D9状态的改变会引发shader的重编译
  - 特别是一些早期的GPU系列(D3D9级别的GPU) 有大量此类问题
  - D3D10.x and D3D11.x没有重编译的问题

状态相关编译(续)

- “危险” 状态 (按严重性排列)
  - 绑定/解除绑定shadow map
  - 在同一个sampler上切换浮点数格式与非浮点数格式的贴图
  - 绑定一个与编译时格式不同的资源
  - 将sRGB状态应用到texture或render target上
  - 对同一个pixel shader, 改变COLORWRITEENABLE状态
  - 含有静态分支的shader(用布尔变量), 每个静态分支的组合都需要一次编译

在D3D9级别的显卡上,

  - 用户自定义的剪裁平面(user clip plane)
  - 固定管线中使用的雾的相关参数
  - MRT相关的状态

Shader编译：应对方案

- 老办法依然是好办法：
  - 在加载地图的时候创建所有的shader
  - 把所有的物体至少渲染一遍，每个至少渲染一个三角形
  - 如果使用动态加载，可以设置隐藏的物体，把常用材质放上去，在加载点将其渲染一遍
- 如果上述方法无法做到，
  - 可以让驱动在运行时编译，但不要在创建shader后马上就使用它
  - 在调用Create***Shader和调用Set***Shader之间，给驱动留出500毫秒 ~ 1000毫秒的时间进行编译

Shader编译：应对方案(续)

- 对于状态相关编译：
  - 把物体按照危险状态分组
  - 尽量减少或避免危险状态的变更
  - 确保让shader编译和运行在那些最常用的状态值下
- 若使用D3D11，使用异步创建可有效缓解编译带来的开销
  - 但注意，不要有时用异步创建机制，有时又不用
- 最后不要忘了用Nsight来查看shader编译的情况

资源管理的基本知识

- 资源的创建与销毁
  - 资源未必会在创建时分配显存，而是在第一次被引用时(在使用WDDM的操作系统上)
  - 在运行时创建大块资源会带来巨大的开销
  - 调用Release函数不会马上销毁资源，只是把资源的引用计数减1．当引用计数下降到0时，资源被销毁
  - 频繁创建/销毁资源会造成内存碎片
    - 尽量重用

资源管理的基本知识(续1)

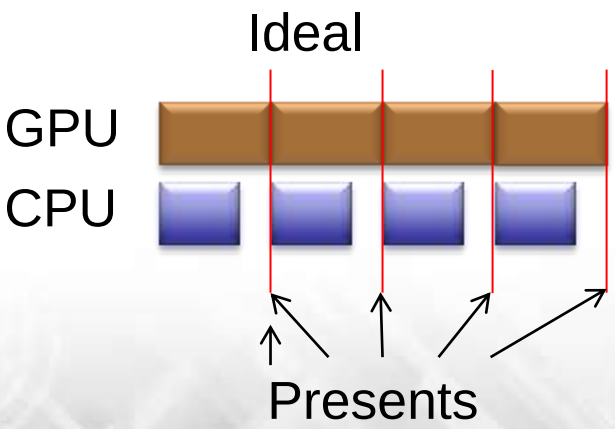
- CPU-GPU同步点(sync point)
  - 如果CPU要求GPU在某个API调用返回之前就完成当前的工作，就会产生一个同步点
  - 1个糟糕的同步点就可能让游戏的帧数**减半**

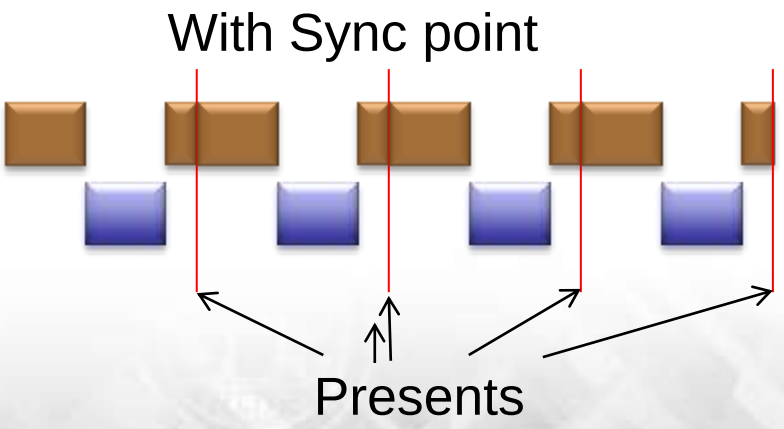
- 各种同步点
  - CPU立即更新一个GPU还在使用中的资源
  - 从一个render target中读回刚刚渲染的数据
  - 在释放了一大块资源后，立即分配一大块资源

  ……

资源管理的基本知识(续2)

- 为什么同步点这么糟糕?
  - 理想状态下, 每帧的持续时间应该是max(CPU时间, GPU时间)
  - CPU-GPU同步点把这个持续时间变成CPU Time + GPU Time.
  - 一个长时间的同步点会产生一次卡顿





资源管理的基本知识(续3)

- 回想一下, GPU较CPU有1~4帧的延迟. 一个随机出现的同步点, 对CPU来说意味着:
  - 清空所有的command buffer
  - 等GPU完成最多4帧的工作!
  - 出现卡顿

- 在D3D9调用Lock需要额外注意
  - 锁定任何buffer, 如果flags=0, 那极可能是一个同步点(GPU正在使用该buffer).

资源管理: 应对方案

- 通用原则 1
  - 调用Lock和Map的时候一定要使用DISCARD标记

    (在使用DISCARD标记后, 得到的资源可能是一个新创建的资源, 不是原来的资源. 这个新资源在其后会替换原有的资源. 这个优化避免了同步点, 但增加了显存用量.)

  - 对频繁上锁的资源, 使用DYNAMIC类型. 在锁定时, 设法使用NOOVERWRITE标记

    (驱动倾向于把DYNAMIC资源存放在系统内存中. 对于vertex/index buffers和小texture来说, 放在系统内存中不会对性能产生很大影响.)

资源管理：应对方案(续2)

- 通用原则 2
  - 避免在运行时创建/销毁资源
  - 尽量在启动时分配资源，运行时进行重用
  - 在重用一个资源之前，调用查询功能(query)来确保GPU已经使用完毕该资源.

资源管理：应对方案(续3)

- 仔细管理小buffer
  - 包括动画，粒子系统，界面元素，等等.
  - 游戏引擎可以自行管理一个显存池，用于buffer的重用和更新

- 自行管理的无冲突显存池
  - 游戏引擎分配一块显存用来做缓冲池，按照堆(heap)或环形缓冲(circular buffer)来管理
  - 设置3个链表：可用空间，占用空间和待释放空间
  - 在释放一块显存时，先把它放入待释放空间链表，调用一个查询(query)来确认GPU是否已经使用完毕. 确认后，从待释放空间链表中删除，放入可用空间链表

显存溢出：应对方案

- 如果一定需要在运行时创建/销毁资源：
  - 先销毁，再创建. 避免内存碎片.
  - 短暂的显存溢出可能会引起驱动在内存管理策略上的改变

- 显存的分配原则：
  先到先占用

显存溢出：应对方案(续1)

- 按照资源种类的重要性来调整分配顺序：
  1. Depth-stencil surface
  2. Render target
  3. 可随机访问的只读资源：
     Textures
  4. 流式访问的只读资源：
     Vertex buffer, index buffer
     或尺寸较小的texture

- 对同类资源来说，按尺寸和格式来分配：
  - 尺寸越大，抗锯齿级别越高和浮点格式的资源应优先分配

显存溢出：应对方案(续2)

- 显存溢出并不总是产生卡顿
  - 如果关键性的资源(GPU可写资源)能够全部放入显存，把部分只读资源放入系统内存不会造成太大的性能问题. 在这种情况下，只读资源不会产生很多显存分页交换.
- GPUView可以用来跟踪显存分页交换

  - 红色块代表分页交换

缓冲帧(Queued Frames)

- 帧缓冲的必要性
  - 为什么驱动总是试图缓冲更多帧?

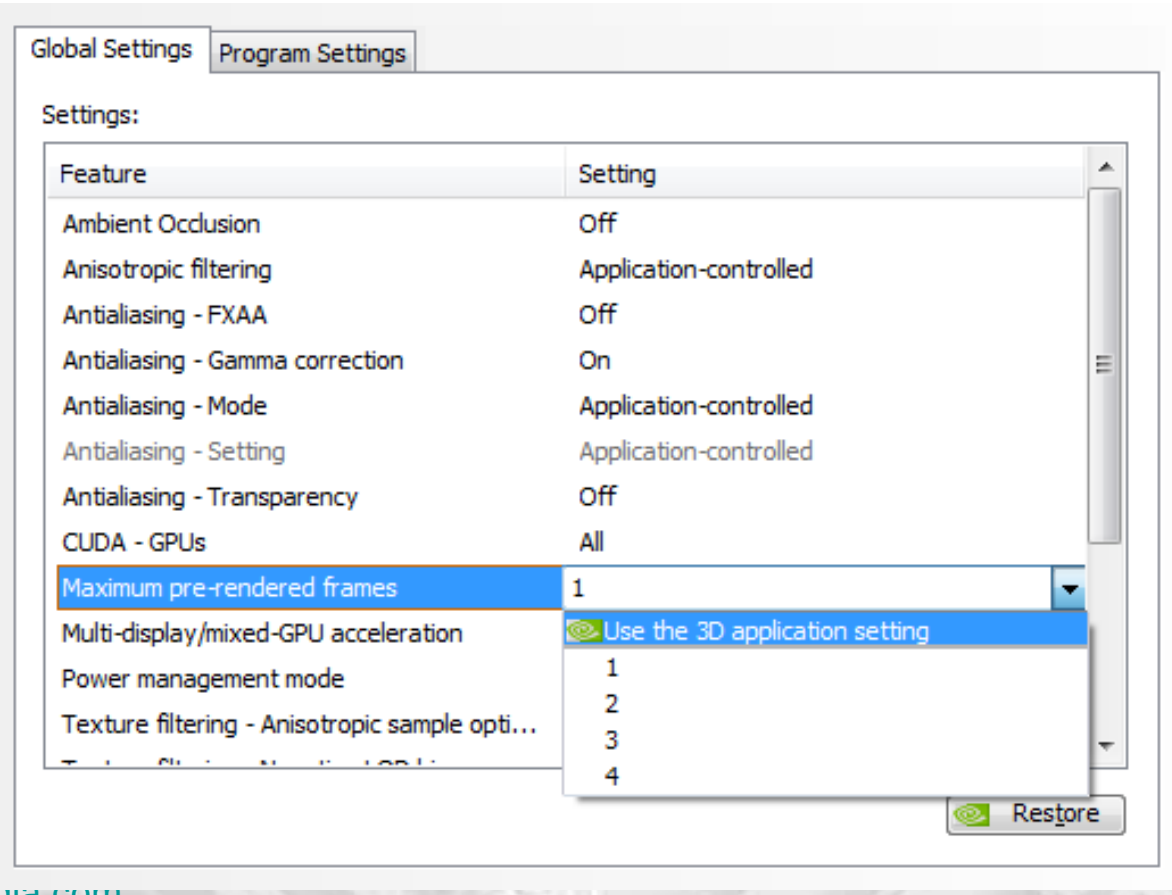
  - 缓冲的帧越多, CPU被阻塞在Present调用上的机率就越少 (换句话说, CPU时间线上的气泡也越少)
  - 同样, GPU时间线上的气泡也越少
  - 驱动可以提前多帧进行调度, 避免CPU负载不平均带来的颠簸. 在合适的时间提交command buffer
  - 在正常情况下,帧缓冲 -> 更高、更平滑的帧率

缓冲帧的矛盾

- 矛盾#1
  - 缓冲帧增加输入延迟. 降低缓冲帧的数量有益于更快的响应时间
  - 但是, 降低缓冲帧的数量会暴露不均匀的CPU负载, 增大微型卡顿的机率. 同时, 降低CPU和GPU的使用效率
- 矛盾 #2
  - 增加缓冲帧的数量有益于平滑的帧率
  - 但是, 如果有不良的CPU-GPU同步点出现, 会产生更严重的卡顿.
- 如果决定要限制缓冲帧的数量, 那么
  - 应保证游戏引擎能够均匀的分配负载
  - 加强资源管理, 避免出现不良的同步点

缓冲帧：应对方案

- 对缓冲帧的数量进行实验
  - 在NVDIA控制面板中调整"最大与渲染帧数"
  - 检查卡顿是否有好转

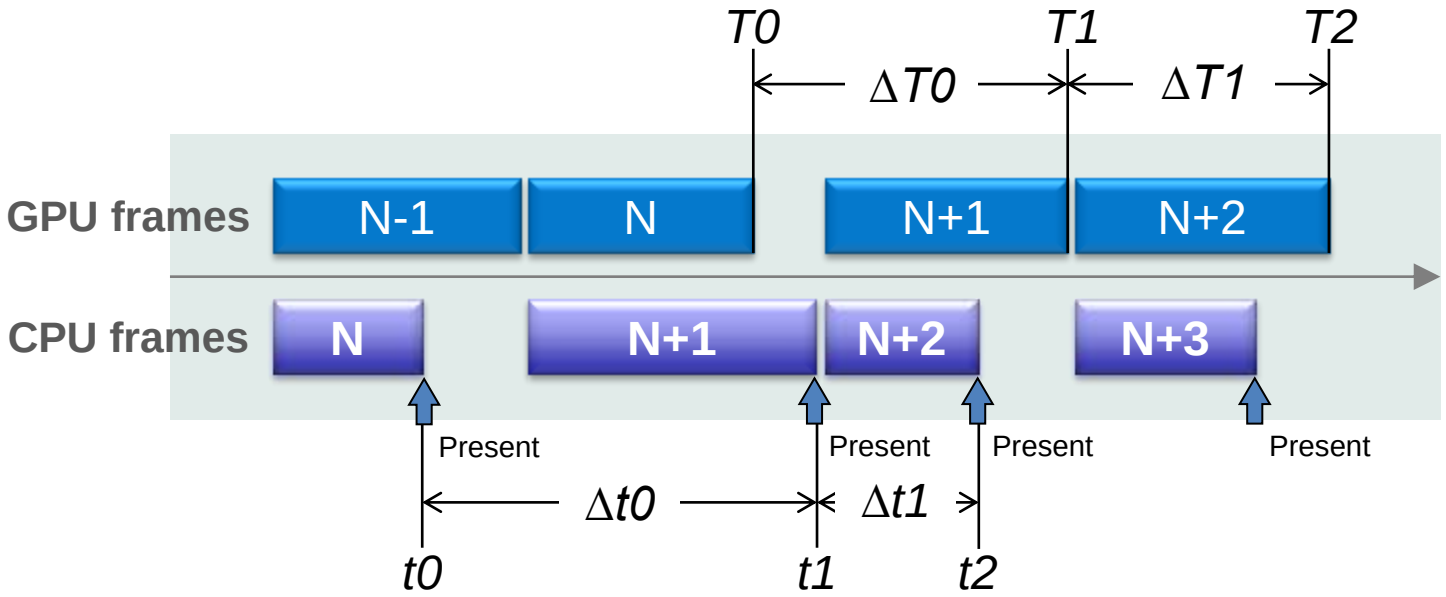


缓冲帧：应对方案(续)

- 限制缓冲帧的方法
  - 不要直接从NVIDIA控制面板中强行设置!
    会影响整个系统和其它游戏的性能
  - 使用事件查询(event query, 方法见DXSDK文档)
    但这不是最有效的方法. CPU的效率会受损
  - 使用API函数:
    IDirect3DDevice9Ex::SetMaximumFrameLatency
    IDXGIDevice1:: SetMaximumFrameLatency

计时问题：解决计时偏差

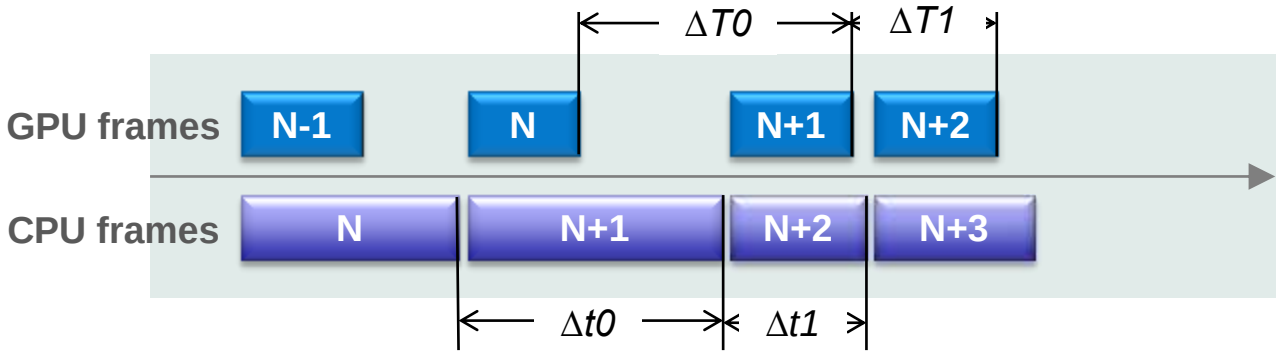
- CPU帧时间 vs. GPU帧时间



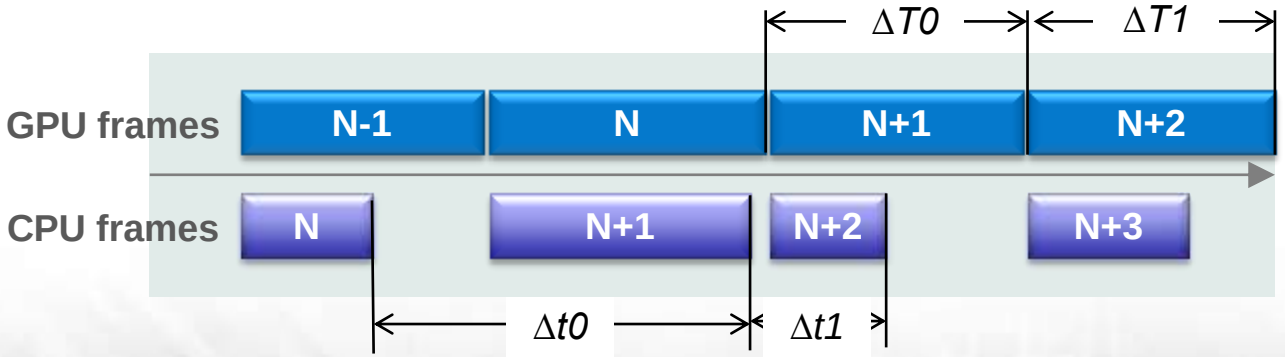
  - 游戏在时间点*t0*, *t1*, *t2*, …调用Present
  - 用户在时间点*T0*, *T1*, *T2*, …看到相应的帧
  - *Δt1- Δt0*不能用来作为动画等系统的步进时间，因为真正的帧时间是*ΔT1 - ΔT0*

计时问题：解决计时偏差(续)

- 两种情况
  - CPU是性能瓶颈，采用CPU帧时间不会带来太大偏差



  - GPU是性能瓶颈，采用CPU帧时间可能会带来极大的偏差



计时偏差：应对方案

- 使用GPU时间戳(time stamps)
  - 调用Present后，调用时间戳查询(time stamp query)来获取GPU完成Present的确切时间点.
  - 但GPU落后于CPU数帧，这个时间戳要一定帧数后才能得到. 所以只能用来做估算用途
  - 为更快得到结果，可在Issue调用后立即调用一次GetData，并设置 FLUSH标记
- 估算帧时间
  - 简易方法：取最近几帧的平均值
  - 高级方法：对比CPU调用Present的时间点和GPU完成Present的时间点，查看最近几帧的性能瓶颈是CPU还是GPU. 然后计算加权平均值.

合理使用查询(Query)

- D3D中的异步查询(async query)
  - D3D9开始引入异步查询, 主要用来解决GPU落后于GPU数帧的问题
  - 反复调用GetData来获取异步查询的结果, 会产生一个CPU-GPU同步点

    while (S_FALSE == pQuery->GetData(…, D3DGETDATA_FLUSH));

  - 这种做法可能会造成卡顿

事件查询(Event Queries)

- 事件查询可用于限制缓冲帧
  - 有利于降低输入延迟，但是...
  - 同时会暴露不均匀的CPU负载，产生微型卡顿
  - CPU必须等待查询返回，降低了CPU和GPU的并行性，导致低性能
  - 驱动无法进行多种优化和调度，因为没有提前多帧的数据

遮挡查询(Occlusion Query)

- 遮挡查询往往会有很高的延迟
  - 查询结果在1~3帧后返回
  - 避免反复调用GetData直到结果返回的做法，因为会到来非常严重的停滞：

    首先，CPU等待GPU返回结果，
    然后，GPU等待CPU发送当前帧
    CPU-GPU处于串行工作的状态

    这种做法可能会抵消使用遮挡查询带来的性能提升.

遮挡查询：应对方案

- 使用查询时需额外小心
  - 保证查询的使用不会在时间线上引入额外的气泡
  - 理想状态下，优化资源的管理，不要使用限制缓冲帧的方法.
  - 有效使用非阻塞方式进行遮挡查询(本讲座中未作讨论)

检查使用的中间件

- 中间件往往都是在”真空”里开发的
  - 小环境里写出来的系统未必能够适应实际环境
- 特别需要检查中间件是否造成了CPU-GPU同步点

垂直同步，SLI及其它情况

垂直同步(Vsync)

- 垂直同步可能会带来微型卡顿
  - 帧率在垂直同步点之间跳跃: 60fps, 30fps, 20fps, …
  - 游戏可以自行实现限帧机制, 避免垂直同步带来的瞬间大幅度帧率跳跃
- 最新的NVIDIA控制面提供自适应垂直同步选项
  - 当帧率降低到指定的垂直同步点以下时, 自动关闭垂直同步功能

SLI

- 在多GPU环境下，微型卡顿更容易出现
  - 多颗GPU可能在不均匀的时间点上完成Present
  - 同步点造成的阻塞问题更加难以消除
  - GPU间的数据传输会增加更多的同步点
- 驱动有义务为SLI消除卡顿
  - 但游戏也须为此进行优化
- 深入的多GPU情况讨论不在本讲座的讨论范围内

其它卡顿的起因

- 一些较为少见的卡顿起因:
  - GPU状态切换
    如果游戏使用了CUDA, compute shader等利用GPU通用计算, 那么在图形和通用计算的切换时, command buffer可能会在不合适的事件点被清除
  - 多个D3D设备间竞争硬件资源
    对于需要多开的游戏, 可能会在多个渲染设备间产生这种问题
  - 系统paged/non-paged pool不够用
    在XP 32bit下, paged/non-paged pool不够用有一定可能造成卡顿, 特别是那些同时使用大量资源的游戏
  - 还有更多可能的起因, 限于篇幅, 此处不作讨论.

谢谢!

Q & A