DirectX 12与超低驱动层开销的实践

杨雪青
内容技术开发工程师，英伟达(NVIDIA)

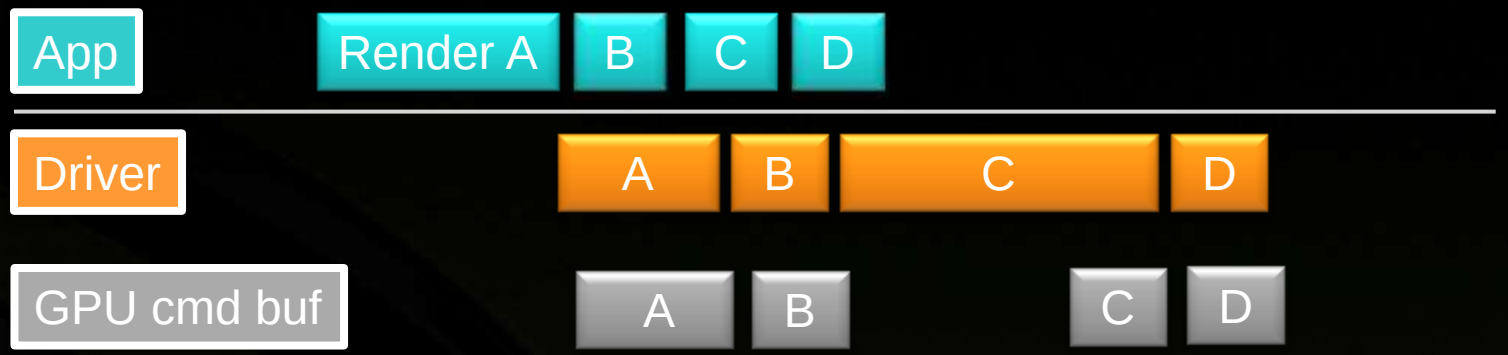
概要

- 为何关注驱动开销
- 实现低驱动开销的方法
- D3D12的相关新特性

概要

- 为何关注驱动开销
- 实现低驱动开销的方法
- D3D12的相关新特性

游戏与驱动的交互



- 应用程序发出渲染调用
- 驱动负责对API的具体实现，生成GPU渲染所用的命令缓冲 (Command Buffer)
- 有些渲染命令的主要开销在驱动层面
  - 渲染物体C的驱动开销大于A/B/D

驱动开销的由来

- 根据API的执行说明来执行和实现API函数的功能
- 验证与编译
  - 绑定对象
    - RTs, shaders, textures, buffers
- 同步与内存分配
  - Buffer的更新
- 冲突回避（Hazard avoidance）
- 累积效应

概要

- 为何关注驱动开销
- 实现低驱动开销的方法
- D3D12的相关新特性

“完整” Draw Call

```
Ctx->IASetInputLayout( … );
Ctx->IASetVertexBuffers( … );
Ctx->IASetIndexBuffer( … );
Ctx->VSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->VSSetConstantBuffers( … );
Ctx->PSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->PSSetConstantBuffers( … );
Ctx->PSSetShaderResources( … );
Ctx->PSSetSamplers( … );
Ctx->DrawIndexed( … );
```

驱动在“完整”draw call 上的开销

- 4个以上的GPU内存对象的绑定
  - Vertices, Indices, Constants, Textures等
- 每个对象引用可能需要
  - 指针间接引用(Pointer indirection)
  - 对象生命周期管理
  - 对象的常驻化管理(Residency management)
- 两次内存管理操作，Map + Discard 需要
  - 对活动对象的重命名(Renaming)
  - 管理内存池

我们可以做些什么?

- **减少非Draw函数的调用**
- **减少Draw Call函数的调用**
- **将工作并行化处理**

减少渲染时的各种设置

- Vertex 和 Index buffer 的再分配（Sub-allocate)
- 超级着色器(Übershaders)
- 优化常数缓冲(Constant Buffer)的使用
- 优化SRV(Shader Resource View)的管理
- 优化采样器(Sampler)的管理

```
Ctx->IASetInputLayout( … );
Ctx->IASetVertexBuffers( … );
Ctx->IASetIndexBuffer( … );
Ctx->VSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->VSSetConstantBuffers( … );
Ctx->PSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->PSSetConstantBuffers( … );
Ctx->PSSetShaderResources( … );
Ctx->PSSetSamplers( … );
Ctx->DrawIndexed( … );
```

Vertex 及 Index 的再分配(Suballocation)

- 多个对象共享相同的Buffer
- 大多数游戏有标准化的顶点格式
  - 需配置的数量较少
- DrawIndexed 提供 BaseIndex/BaseVertex
  - 极大减少对 IASetXXX 等函数的调用

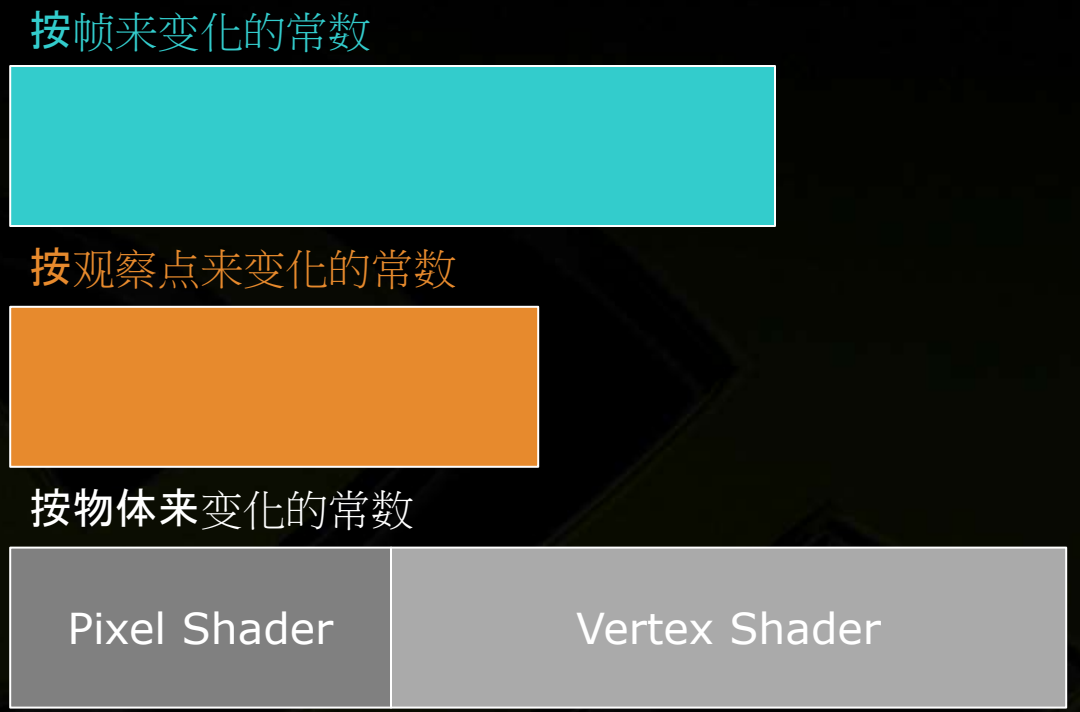
```
Ctx->IASetInputLayout( … );
Ctx->IASetVertexBuffers( … );
Ctx->IASetIndexBuffer( … );
Ctx->VSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->VSSetConstantBuffers( … );
Ctx->PSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->PSSetConstantBuffers( … );
Ctx->PSSetShaderResources( … );
Ctx->PSSetSamplers( … );
Ctx->DrawIndexed( … );
```

超级着色器(Übershaders)

- 在不同条件间变换Shader会引起额外的开销
  - Shader关系到渲染的很多方面
- 条件语句本身开销不大
  - `if (hasSpec)  / for( numLayers )`
- Übershader: 在一个shader中，用静态条件语句来区分不同材质的实现方法
- 可能会对GPU的性能造成负面影响
  - 比如对增加寄存器的使用量

```
Ctx->IASetInputLayout( … );
Ctx->IASetVertexBuffers( … );
Ctx->IASetIndexBuffer( … );
Ctx->VSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->VSSetConstantBuffers( … );
Ctx->PSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->PSSetConstantBuffers( … );
Ctx->PSSetShaderResources( … );
Ctx->PSSetSamplers( … );
Ctx->DrawIndexed( … );
```

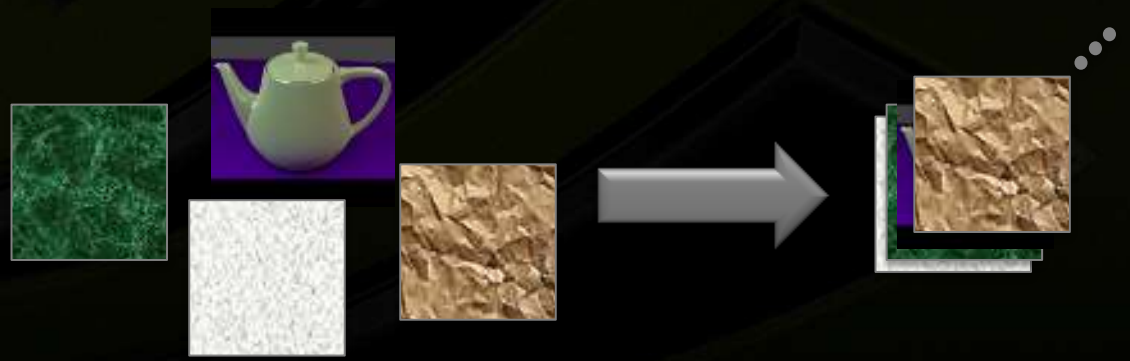
减少常数缓冲(Constant Buffer)的开销




```
Ctx->IASetInputLayout( … );
Ctx->IASetVertexBuffers( … );
Ctx->IASetIndexBuffer( … );
Ctx->VSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->VSSetConstantBuffers( … );
Ctx->PSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->PSSetConstantBuffers( … );
Ctx->PSSetShaderResources( … );
Ctx->PSSetSamplers( … );
Ctx->DrawIndexed( … );
```

SRV(Shader Resource View)的优化管理

- 给“全局”贴图分配指定专用的Slot
  - Shadow maps, environment maps
- 不要清空 Slot
  - 绑定 NULL 也是有开销的
- 使用贴图数组来进行再分配



```
Ctx->IASetInputLayout( … );
Ctx->IASetVertexBuffers( … );
Ctx->IASetIndexBuffer( … );
Ctx->VSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->VSSetConstantBuffers( … );
Ctx->PSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->PSSetConstantBuffers( … );
Ctx->PSSetShaderResources( … );
Ctx->PSSetSamplers( … );
Ctx->DrawIndexed( … );
```

采样器(Sampler)的优化管理

- 同 SRV 的优化方法，但是没有类似的数组可用
- Samplers 并不经常被改变
  - 通常就那么几套设置

```
Ctx->IASetInputLayout( … );
Ctx->IASetVertexBuffers( … );
Ctx->IASetIndexBuffer( … );
Ctx->VSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->VSSetConstantBuffers( … );
Ctx->PSSetShader( … );
Ctx->Map/Unmap( … );
Ctx->PSSetConstantBuffers( … );
Ctx->PSSetShaderResources( … );
Ctx->PSSetSamplers( … );
Ctx->DrawIndexed( … );
```

优化后的 Draw Call

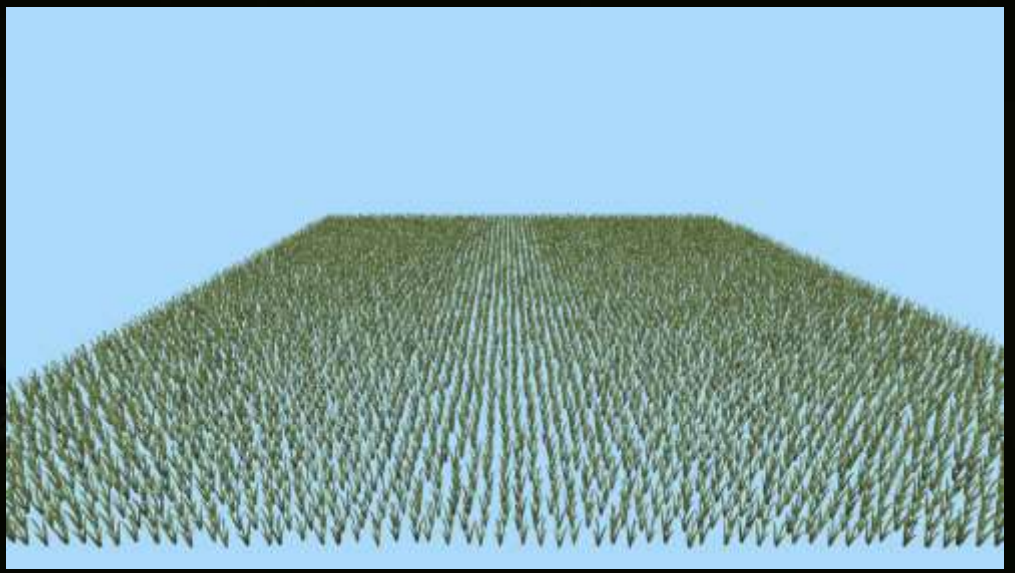
```
// Setup:
//   ubershader constants
//   texture array indices
//   normal constants, like transform
Ctx->Map/Unmap( … );

// Don’t need to rebind CBs

// Offsets in multiuse IB/VB
//  StartIndexLocation -> offset to IB
//  VertexOffset -> offset for vertices
Ctx->DrawIndexed( … );
```

我们可以做些什么?

- 减少非Draw函数的调用
- **减少Draw Call调用**
- 将工作并行化处理

减少 Draw Call 的数量

- 已经是老生常谈了
  - 实例化（Instancing）
- 几乎每个游戏都可以受益
  - 相同的模型可能在一帧里被渲染多次
  - 非常适合应用于粒子系统



我们可以做些什么?

- 减少非Draw函数的调用
- 减少Draw Call的调用
- 将工作并行化处理

延迟上下文(Deferred Context)

- 可以将工作量分散到多个线程
- 常数缓冲的一些更新操作（调用Map等）的处理受益最大
- 驱动还是要做许多串行化工作

在OpenGL中实现超低驱动开销

- 持久映射缓冲(Persistent-mapped buffer)
  - 更高效的动态几何数据更新
- MultiDrawIndirect (MDI)
  - 更高效的多个Draw Call提交方式
- 将 2D 贴图打包到数组
  - 不会因为需要更改贴图，而增加渲染批次

在OpenGL中实现超低驱动开销

- 持久映射缓冲(Persistent-mapped buffer)
- MultiDrawIndirect (MDI)
- 将 2D 贴图打包到数组

BufferStorage 与持久映射(Persistent Map)

- 使用glBufferStorage()来分配Buffer

```
glBufferStorage(GL_ARRAY_BUFFER, ringSize, NULL, flags);
```

- 通过以下标记来打开持久映射

```
GLbitfield flags = GL_MAP_WRITE_BIT
                 | GL_MAP_PERSISTENT_BIT
                 | GL_MAP_COHERENT_BIT;
```

绘制过程中始终处于map状态,
无需Unmap

写入的数据可以直接反映到GPU

动态更新几何数据

- 在创建时做一次Map

```
data = glMapBufferRange(ARRAY_BUFFER, 0, ringSize, flags);
```

- 不需要在渲染中调用Map/Unmap
  - 但是需要应用程序负责同步操作

```
WriteGeometry( data, ... );
data += dataSize;
```

glFenceSync() 和 glClientWaitSync()

- 使用Fence机制来保证在写入新数据前原先的数据已被渲染完毕

在OpenGL中实现超低驱动开销

- 持久映射缓冲(Persistent-mapped buffer)
- MultiDrawIndirect (MDI)
- 将 2D 贴图打包到数组

渲染多个对象时的典型操作

```
foreach( object )
{
    WriteUniformData( object, &uniformData );

    glDrawElementsBaseVertex(
        GL_TRIANGLES,
        object->indexCount,
        GL_UNSIGNED_SHORT,
        object->indexDataOffset,
        object->baseVertex );
}
```

改用间接绘制(Indirect Draw)

```
DrawElementsIndirectCommand command;
foreach( object )
{
    WriteUniformData( object, &uniformData );
    WriteDrawCommand( object, &command );
    glDrawElementsIndirect(
        GL_TRIANGLES,
        GL_UNSIGNED_SHORT,
        &command );
}
```

每个对象的参数从内存中获取

```
typedef struct {
    uint count;
    uint instanceCount;
    uint firstIndex;
    uint baseVertex;
    uint baseInstance;
} DrawElementsIndirectCommand;
```

同时提交多个间接绘制处理

```
DrawElementsIndirectCommand* commands = ...;
foreach( object )
{
    WriteUniformData( object, &uniformData[i] );
    WriteDrawCommand( object, &commands[i] );
}
glMultiDrawElementsIndirect(
    GL_TRIANGLES,
    GL_UNSIGNED_SHORT,
    commands,
    commandCount,
    0 );
```

填入每个对象的数据(**可以通**过并行化处理或者使用GPU计算优化这一步)

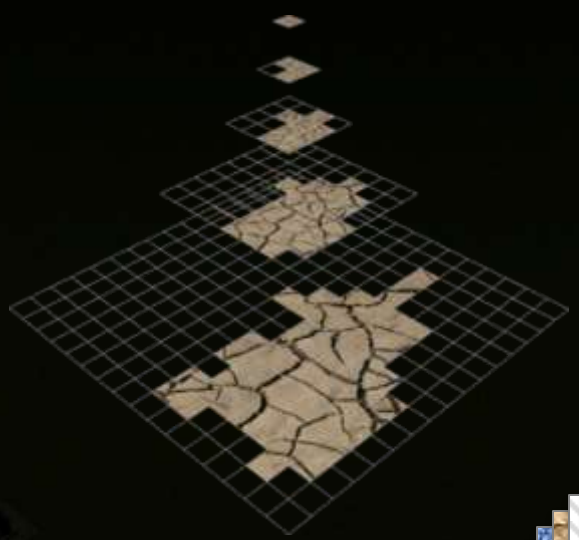
把存好的各对象送交渲染

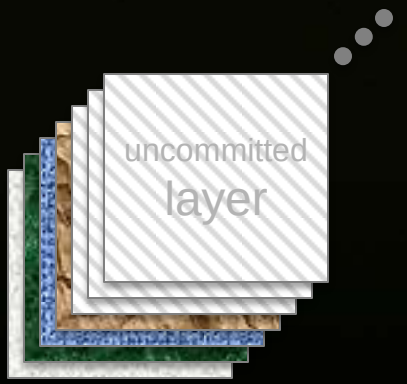
在OpenGL中实现超低驱动开销

- 持久映射缓冲(Persistent-mapped buffer)
- MultiDrawIndirect (MDI)
- 将 2D 贴图打包到数组

将2D贴图打包到数组中去

- D3D中对 SRV优化方法在此同样适用
- 免绑定贴图(Bindless texture)
  - 翻新了传统的贴图绑定模式
  - 无需设置贴图单元(Slot)
- 稀疏贴图(Sparse texture)
  - 贴图由多个分块(Tile)组成
  - 每个分块可以是常驻的也可以不是
  - DX11.2 也有支持
- 稀疏免绑定贴图数组(Sparse Bindless Texture Array)





概要

- 为何关注驱动开销
- 实现低驱动开销的方法
- D3D12的相关新特性

DirectX12的新特点

- 大幅减少大量任务提交的开销
  - 应用程序将负担起更多的责任：资源冲突监测，CPU-GPU 同步，内存管理等
- 更适合GPU构架的最新发展状况
- 能充分利用多核系统的优势
- 提供主机式的执行环境

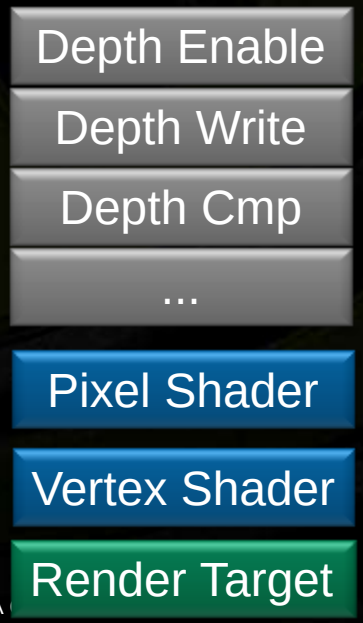
DX12 的新特性

- 状态管理模型
- 命令列表(Command list)
- 资源绑定及冲突监测
- 数据常驻管理(Residency management)与内存模式(memory model)
- CPU/GPU 同步

状态管理

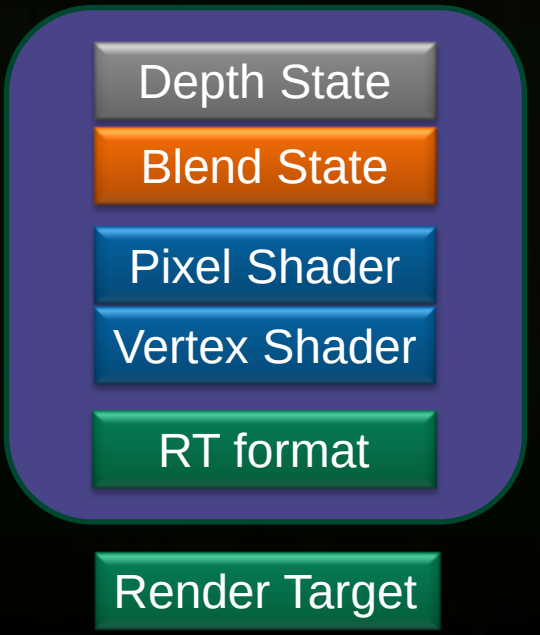
- 管线状态对象(Pipeline state object) – 更大范围的状态组合
- D3D12以前: 独立设置各种各样的状态
- D3D12以后: 设置一个几乎涵盖所有状态的大对象

D3D9 – 大量独立的小状态



D3D11– 小范围内相关状态打包
在绘制时解决相关性

PSO – 大范围内状态打包
在创建时解决相关性



命令列表(Command list)

- 每个命令列表都是一串渲染命令流
- 先渲染命令的集合，可供以后可多次使用

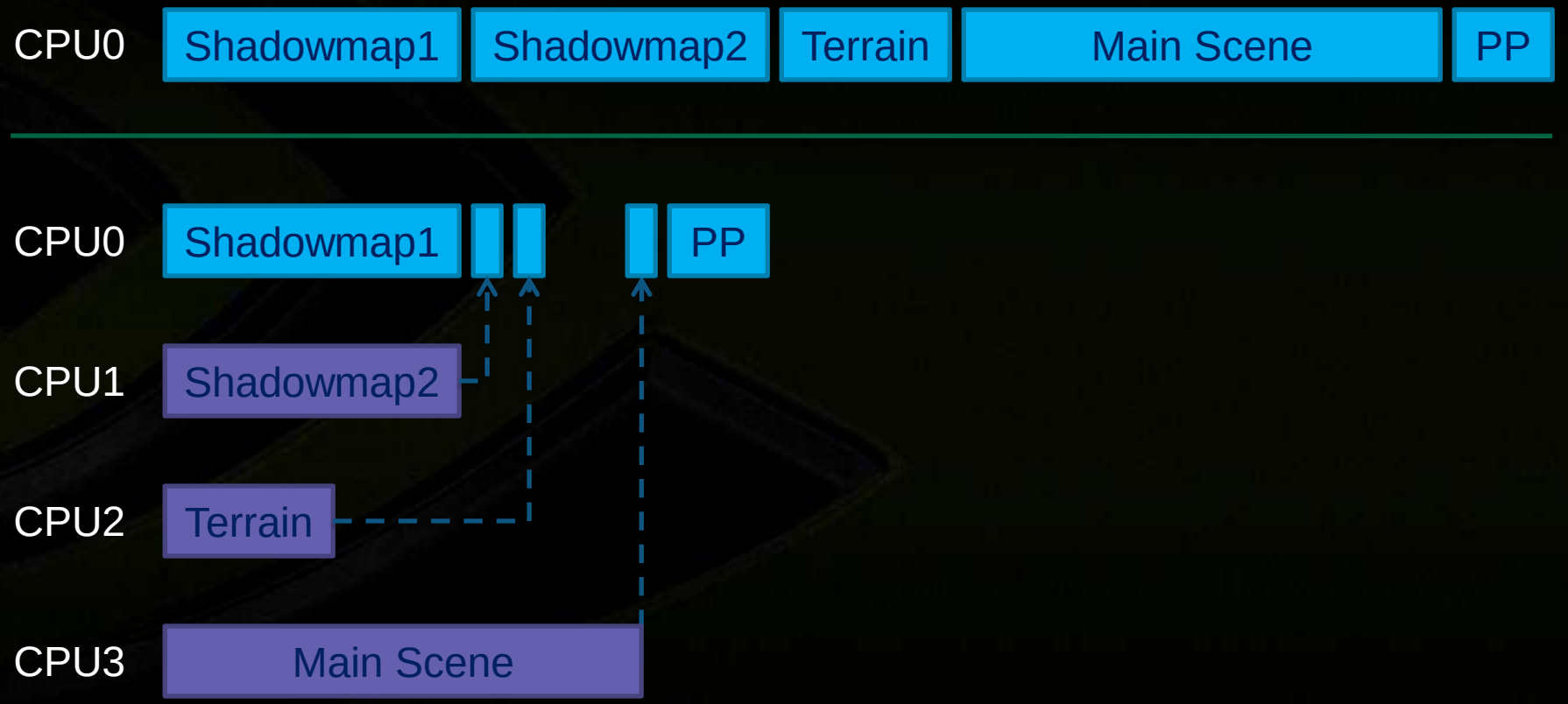
CmdList X
SetRenderTarget
SetVB, SetIB
SetTexture
SetViewport
SetPSO
Draw

CmdList Y
SetTex
SetUAV
Dispatch

Cmd Queue
ExecCmdList(X)
AdvanceFence()
ExecCmdList(Y)
AdvanceFence()

命令列表(Command List) – 多线程

- 把在命令列表的创建工作分配到CPU的多个核上

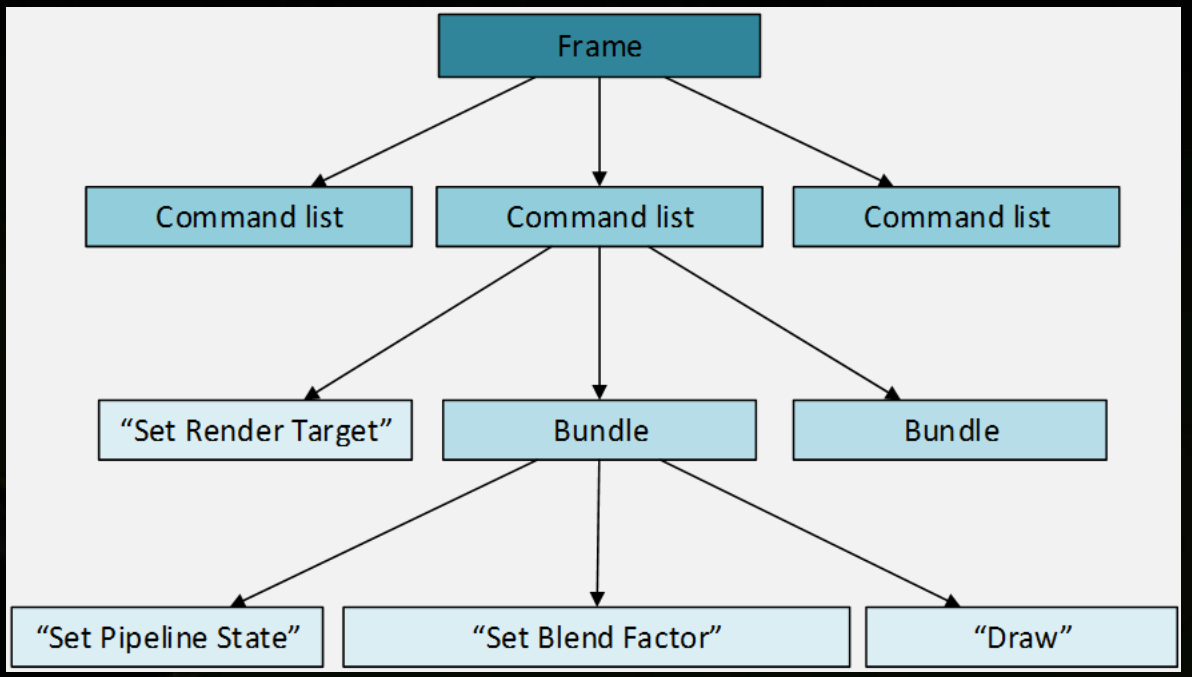


与 D3D11 命令列表的区别

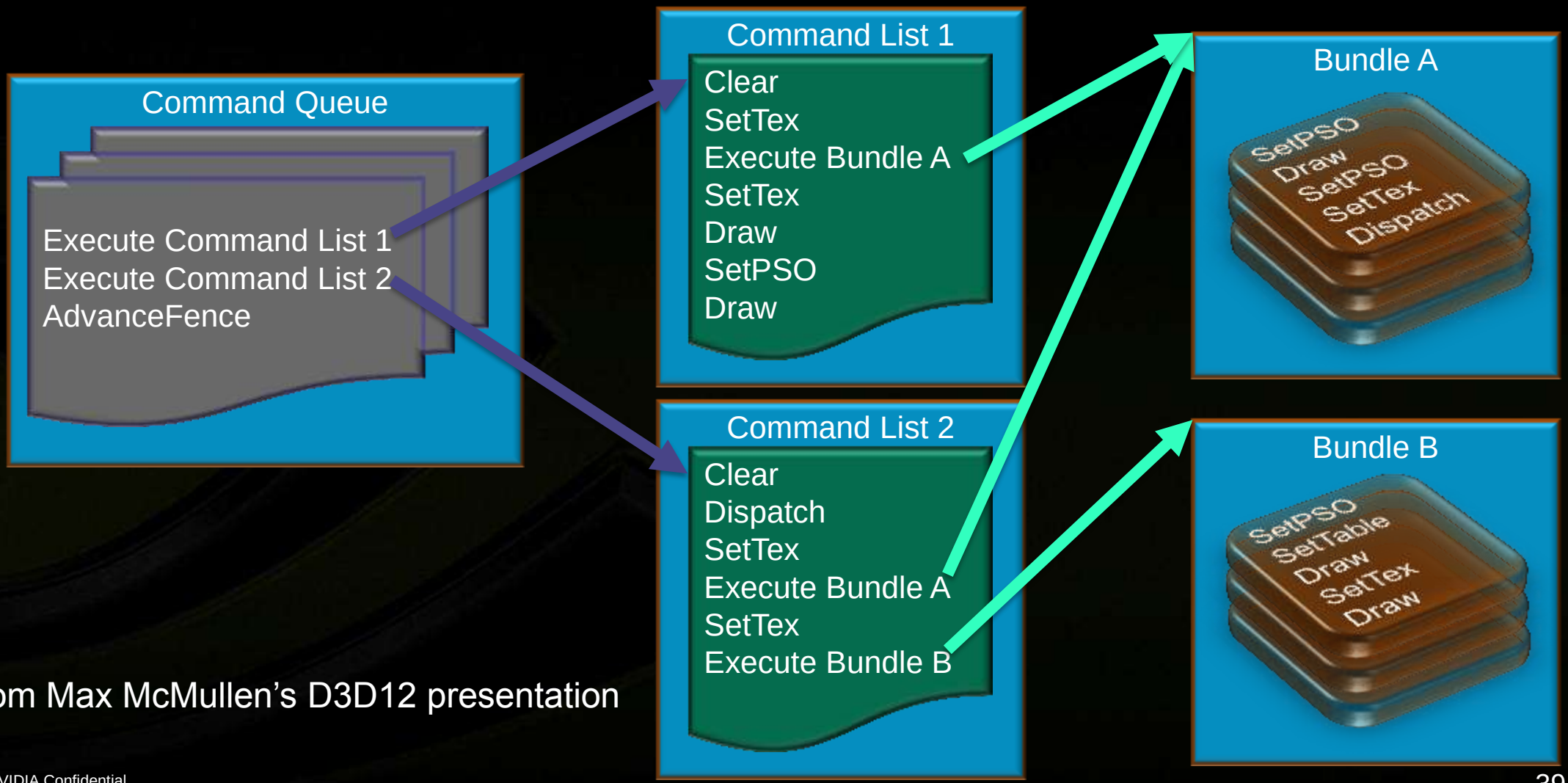
- 概念十分相似：
  - 渲染命令在多个线程中并行创建完成
  - 串行执行命令
    - D3D11 – immediate context
    - D3D12 – command queue
- D3D12
  - 驱动不再负责冲突监测
  - 去掉了immediate mode渲染

Bundles

- Bundle适用于较短小的，会在一帧中或多帧间被重复使用的渲染操作
- 可以被重复利用的渲染命令包
- 亦可提高单线程下的渲染效率
- 保留了灵活性可以处理不同数据



示例 Command List + Bundle



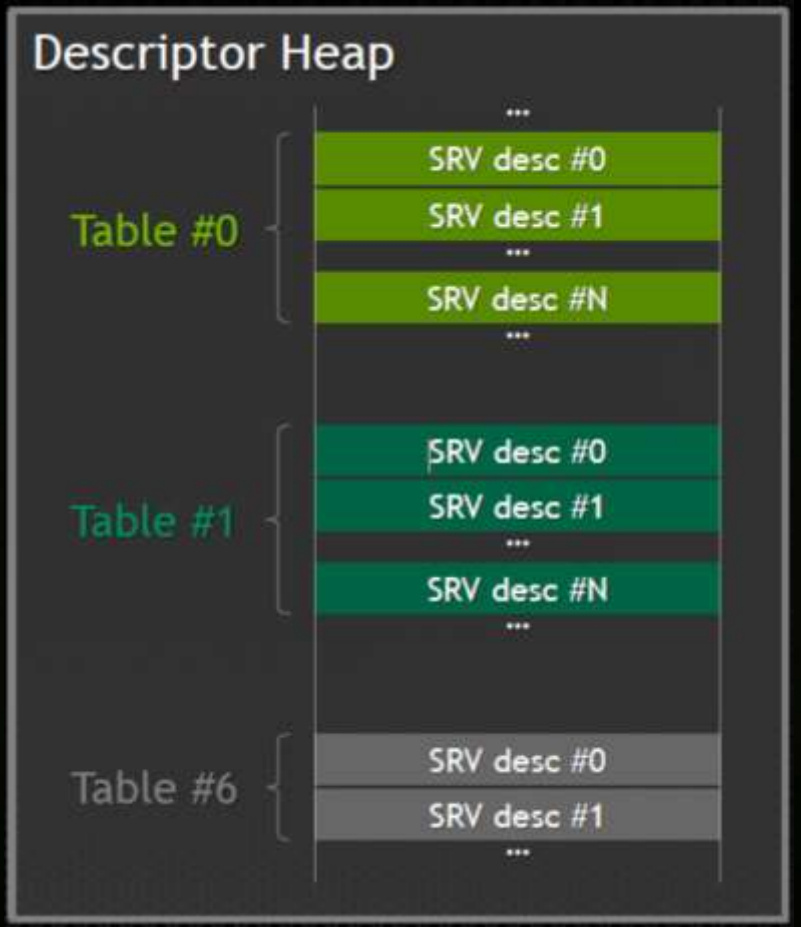
From Max McMullen's D3D12 presentation

免绑定贴图与采样器(Bindless Texture & Sampler)

- 这种新的资源模式常常被称为“免绑定(Bindless)”
- Bindless指向低层的资源所在
  - 贴图和采样器直接通过指针直接引用而不是通过绑定Slot来引用
  - 资源的使用数量不受Slot数量的限制

描述符(Descriptor), 描述符表(Descriptor Table)与描述符堆 (Descriptor Heap)

- 描述符封装了指向资源的句柄
  - 字面上等同于 ID3D11*ResourceView 对象
- “描述符表”是描述符组成的连续数组
  - 每个表代表了一系列资源
  - GPU可以对表中单个的描述符进行索引
  - 可一次性设置一系列贴图(或其它资源)
  - 快速同时绑定大量资源
- 描述符表定义在描述符堆中
  - 驱动提供描述符堆所用的内存
  - “堆”的概念后面会讨论



描述符(Descriptor), 描述符表(Descriptor Table)与描述符堆（ Descriptor Heap)

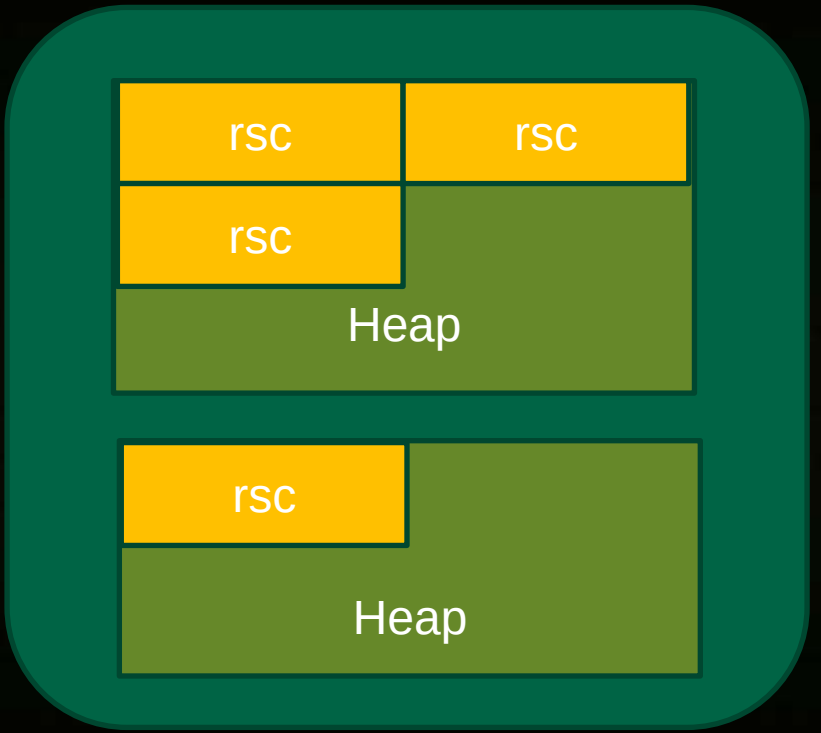
- 表的切换开销很低
  - 理想状态下仅仅是替换一下指针
  - 在进行大量状态变换时十分高效
- 可以同时被设置多个表
  - Shader可以先选择一个表，然后选择表中的各种资源
- 应用程序负责管理堆中的这些表

冲突监测（Hazard Tracking）

- 对渲染进行隐性的冲突监测变得非常困难
- 应用程序要使用显式的屏障(Barrier)来避免冲突
  - 例如资源从RTV 转变为SRV的时候 (如Shadow mapping)
  - 可以利用应用程序对自身的了解来减少保守的安全处理带来的开销

数据常驻管理和堆(Residency Management & Heaps)

- 把内存的使用管理从资源绑定模式中显式地分离出来
  - 不再需要对单个的资源绑定做追踪管理
- 堆(Heap)代表着一大块内存区域
  - OS / 驱动内存管理以堆为单位
- 对堆中的内存单位进行数据常驻管理
  - 应用程序负责对堆内的内存进行管理



CPU/GPU 同步

- 用Fence来避免CPU和GPU之间的数据使用冲突
  - CPU 同 GPU 共享数据
- 应用程序负责设置和监测这些Fence
- 由应用程序负责重命名等优化工作
  - 可以对动态资源进行有效地持久映射

对DX12的总结 (1/2)

- 大幅减少CPU花费在驱动上的开销从而大幅提升性能
- 大幅降低了驱动的复杂度从而使驱动更加稳定可靠
- 不再有由驱动造成的卡顿现象

对DX12的总结 (2/2)

- 应用程序担负更多的责任
  - 资源冲突监测
  - CPU-GPU同步
  - 内存管理
- 如果处理不慎，容易给自己挖坑
- 如果处理得当，应用程序将从中获得巨大的好处！

谢谢 ！

提问?

youngy@nvidia.com