**Direct3D的初始化**

**Direct3D**

**What:**

底层图形应用程序编程接口

**Why:**

通过Direct3D可对图形处理器GPU进行控制和编程

**How:**

Direct3D层和硬件驱动会协作将Direct3D命令转换为系统中GPU可以执行的本地机器指令。

**组件对象模型**

**What:**

组件对象模型是一种令DirectX不受编程语言束缚，并且使之向后兼容的技术。

**Why:**

**How:**

**如何获取COM接口的指针 ？**

借助特定函数或另一COM接口的方法，而非使用关键字new去创建一个COM接口。

**ComPtr类的作用**

COM对象会统计引用计数，在使用完某接口时，我们便应调用它的Release方法。为了辅助用户管理COM对象的生命周期，Windows运行时库专门提供了ComPtr类，我们可以把它当作COM对象的智能指针。

**常用的3个ComPrt方法**

1．Get：返回一个指向此底层COM接口的指针。

2．GetAddressOf：返回指向此底层COM接口指针的地址。

3．Reset：将ComPtr实例设置为nullptr释放与之相关的所有引用。

**纹理格式**

**What:**

2D纹理是一种由数据元素构成的矩阵。纹理中的每个元素存储的都是一个像素的颜色。然而，纹理的用处并非仅此而已。

**Why:**

**How:**

**常见的纹理格式**

1．DXGI\_FORMAT\_R32G32B32\_FLOAT

2．DXGI\_FORMAT\_R16G16B16A16\_UNORM

3．DXGI\_FORMAT\_R32G32\_UINT

4．DXGI\_FORMAT\_R8G8B8A8\_UNORM

5．DXGI\_FORMAT\_R8G8B8A8\_SNORM

6．DXGI\_FORMAT\_R8G8B8A8\_SINT

7．DXGI\_FORMAT\_R8G8B8A8\_UINT

**交换链和页面翻转**

**What:**

前台缓冲区和后台缓冲区构成交换链。

前后台缓冲的互换操作称为呈现。

**Why:**

避免动画中出现画面闪烁的现象。

**How:**

**交换链的工作原理**

前台缓冲区存储的是当前显示在屏幕上的图像数据，而动画的下一帧则被绘制在后台缓冲区里。当后台缓冲区中的动画帧绘制完成之后，两种缓冲区的角色互换。

**深度缓冲**

**What:**

深度缓冲区这种纹理资源存储的并非图像数据，而是特定像素的深度信息。

**Why:**

**How:**

**深度缓冲的工作原理**

深度缓冲区中的元素与后台缓冲区内的像素呈一一对应关系。计算每个像素的深度值，并执行深度测试，具有最小深度值的像素会获得最终的胜利。

**深度缓冲可用的格式**

1. DXGI\_FORMAT\_D32\_FLOAT\_S8X24\_UINT
2. DXGI\_FORMAT\_D32\_FLOAT

3．DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT：

4．DXGI\_FORMAT\_D16\_UNORM

**资源与描述符**

**What:**

我们可以把描述符视为一种对送往GPU的资源进行描述的轻量级结构。视图和描述符是同义词。

**Why:**

GPU资源实质都是一些普通的内存块。

**How:**

**常用的描述符类型**

1．CBV/SRV/UAV：常量缓冲区视图、着色器资源视图和无序访问视图。

2．采样器描述符

3．RTV：渲染目标视图资源

4．DSV：深度/模板视图资源

**什么是描述符堆**

描述符堆中存有一系列描述符，我们需要为每一种类型的描述符都创建出单独的描述符堆。另外，也可以为同一种描述符类型创建出多个描述符堆。

**多重采样**

**What:**

在不提升显示器分辨率时，运用的反走样，抗锯齿技术。

**Why:**

抗锯齿

**How:**

**超级采样(SSAA)**

使用4倍于屏幕分辨率大小的后台缓冲区和深度缓冲区。当数据要从后台缓冲区调往屏幕显示的时候，会将后台缓冲区按4个像素一组进行解析：每组用求平均值的方法得到一种相对平滑的像素颜色。

**多重采样(MSAA)**

通过跨子像素共享一些计算信息，从而使它比超级采样的开销更低。这种技术并不需要对每一个子像素都进行计算，而是仅计算一次像素中心处的颜色，再基于可视性和覆盖性将得到的颜色信息分享给其子像素。

**功能级别**

**What:**

“功能级别”为不同级别所支持的功能进行了严格的界定。

**Why:**

便捷的知道哪些Direct3D功能可以使用。

**How:**

**功能级别的枚举**

enum D3D\_FEATURE\_LEVEL

{

D3D\_FEATURE\_LEVEL\_9\_1 = 0x9100,

D3D\_FEATURE\_LEVEL\_9\_2 = 0x9200,

D3D\_FEATURE\_LEVEL\_9\_3 = 0x9300,

D3D\_FEATURE\_LEVEL\_10\_0 = 0xa000,

D3D\_FEATURE\_LEVEL\_10\_1 = 0xa100,

D3D\_FEATURE\_LEVEL\_11\_0 = 0xb000,

D3D\_FEATURE\_LEVEL\_11\_1 = 0xb100

}D3D\_FEATURE\_LEVEL;

**资源驻留**

**What:**

在Direct3D 12中，应用程序通过控制资源在显存中的去留，主动管理资源的驻留情况。

**Why:**

**How:**

**如何控制驻留**

HRESULT ID3D12Device::MakeResident(UINT NumObjects,

ID3D12Pageable \*const \*ppObjects);

HRESULT ID3D12Device::Evict(UINT NumObjects,

ID3D12Pageable \*const \*ppObjects);

**命令队列和命令列表**

**What：**

命令队列：每个GPU都至少维护着一个命令队列（command queue）。

命令列表：CPU可利用命令列表（command list）将命令提交到这个队列中去。

命令分配器：记录在命令列表内的命令，实际上是存储在与之关联的命令分配器。

**Why：**

**How：**

**围栏**

**What:**

强制CPU等待，直到GPU完成所有的命令处理，达到某个指定的围栏点为止。这种方法称为刷新命令队列，通过围栏实现。

**Why:**

CPU和GPU同步

**How:**

**CPU与GPU间的交互的策略**

在进行图形编程的时候，有两种处理器在参与处理工作，即CPU和GPU，两者并行工作，但时而也需同步。为了获得最佳性能，最好的情况是让两者尽量同时工作，少同步。同步意味着一种处理器要以空闲状态等待另一种处理器完成某些任务。

**资源转换**

**What:**

**Why:**

GPU的写操作还没有完成抑或甚至还没有开始，却开始读取资源，便会导致资源冒险。

**How:**

**帮助文件**

以CD3DX12作为前缀的变体全都定义在d3dx12.h头文件当中，这个文件并不属于DirectX 12 SDK的核心部分，但是可以通过微软的官方网站下载获得。

**初始化Direct3D**

**What:**

**Why:**

**How:**

**Direct3D初始化的9个步骤：**

1．用D3D12CreateDevice函数创建ID3D12Device接口实例。

2．创建一个ID3D12Fence对象，并查询描述符的大小。

3．检测用户设备对4X MSAA质量级别的支持情况。

4．依次创建命令队列、命令列表分配器和主命令列表。

5．描述并创建交换链。

6．创建应用程序所需的描述符堆。

7．调整后台缓冲区的大小，并为它创建渲染目标视图。

8．创建深度/模板缓冲区及与之关联的深度/模板视图。

9．设置视口（viewport）和裁剪矩形（scissor rectangle）。

**性能计时器**

**What:**

**Why:**

精确地度量时间

**How:**

**性能计时器如何使用？**

QueryPerformanceCounter函数获取性能计时器测量的当前时刻值（以计数为单位）

再用QueryPerformanceFrequency函数来获取性能计时器的频率（单位：计数/秒）

**实践：Direct3D初始化框架**

**常用的API**

**功能支持的检测**

HRESULT ID3D12Device::CheckFeatureSupport(D3D12\_FEATURE Feature,

void \*pFeatureSupportData, UINT FeatureSupportDataSize);

**IID\_PPV\_ARGS辅助宏**

#define IID\_PPV\_ARGS(ppType) \_\_uuidof(\*\*(ppType)), IID\_PPV\_ARGS\_Helper(ppType)

**创建队列**

ID3D12Device::CreateCommandQueue

**将命令列表里的命令添加到命令队列之中**

void ID3D12CommandQueue::ExecuteCommandLists(

UINT Count, ID3D12CommandList \*const \*ppCommandLists);

**创建命令分配器**

HRESULT ID3D12Device::CreateCommandAllocator( D3D12\_COMMAND\_LIST\_TYPE type, REFIID riid,void \*\*ppCommandAllocator);

**创建命令列表**

HRESULT ID3D12Device::CreateCommandList( UINT nodeMask,

D3D12\_COMMAND\_LIST\_TYPE type, ID3D12CommandAllocator \*pCommandAllocator,

ID3D12PipelineState \*pInitialState,REFIID riid,void \*\*ppCommandList);

**创建围栏对象**

HRESULT ID3D12Device::CreateFence(

UINT64 InitialValue,D3D12\_FENCE\_FLAGS Flags, REFIID riid, void \*\*ppFence);

**资源转换**

static inline CD3DX12\_RESOURCE\_BARRIER Transition(

\_In\_ ID3D12Resource\* pResource,D3D12\_RESOURCE\_STATES stateBefore,

D3D12\_RESOURCE\_STATES stateAfter,

UINT subresource = D3D12\_RESOURCE\_BARRIER\_ALL\_SUBRESOURCES,

D3D12\_RESOURCE\_BARRIER\_FLAGS flags = D3D12\_RESOURCE\_BARRIER\_FLAG\_NONE)

**创建设备**

HRESULT WINAPI D3D12CreateDevice(IUnknown\* pAdapter,

D3D\_FEATURE\_LEVEL MinimumFeatureLevel,REFIID riid, void\*\* ppDevice );

**创建交换链**

mdxgiFactory->CreateSwapChain(

mCommandQueue.Get(), &sd, mSwapChain.GetAddressOf()));

**创建描述符堆**

md3dDevice->CreateDescriptorHeap(

&rtvHeapDesc, IID\_PPV\_ARGS(mRtvHeap.GetAddressOf())));

**获得存于交换链中的缓冲区资源**

HRESULT IDXGISwapChain::GetBuffer(

UINT Buffer, REFIID riid, void \*\*ppSurface);

**创建渲染目标视图**

void ID3D12Device::CreateRenderTargetView(ID3D12Resource \*pResource,

const D3D12\_RENDER\_TARGET\_VIEW\_DESC \*pDesc,

D3D12\_CPU\_DESCRIPTOR\_HANDLE DestDescriptor);

**创建一个资源与一个堆，并把该资源提交到这个堆中**

ID3D12Device::CreateCommittedResource

**设置Direct3D中的视口**

ID3D12GraphicsCommandList::RSSetViewports

**设置裁剪矩形**

ID3D12GraphicsCommandList::RSSetScissorRects