**4 Direct3D的初始化**

**4.1 预备知识**

**4.1.1 Direct3D 12概述**

Direct3D 12较之上一个版本的主要改变在于其性能优化方面，在大大减少了CPU开销的同时，又改进了对多线程的支持。Direct3D 12的API较Direct3D 11更偏于底层。

**4.1.2 组件对象模型**

组件对象模型(COM)是一种令DirectX不受编程语言束缚，并且使之向后兼容的技术。我们通常将COM对象视为一种接口，但考虑当前编程的目的，遂将它当作一个C++类来使用。用C++语言编写DirectX程序时，COM帮我们隐藏了大量底层细节。我们只需知道：要获取指向某COM接口的指针，需借助特定函数或另一COM接口的方法，而不是用C++语言中的new去创建一个COM接口。另外，COM对象会统计其引用次数，因此，在使用完某接口时，我们便应调用它的Release方法，而不是用delete来删除，当COM对象的引用计数为0时，它将执行释放自己所占用的内存。

Windows运行时库提供了ComPtr类，我们可以把它当作COM对象的智能指针。当一个ComPtr实例超出作用域范围时，它便会自动调用相应COM对象的Release方法，继而省掉了我们手动调用的麻烦。

本书常用的3个ComPtr方法：

Get：返回一个指向此底层COM接口的指针；

GetAddressOf：返回指向此底层COM接口的指针的地址；

Reset：将此ComPtr实例设置为nullptr释放与之相关的所有引用；

**4.1.3 纹理格式**

2D纹理是一种由数据元素构成的矩阵。它的用途之一存储2D图像数据，这种情况下，纹理的每个元素存储的是一个像素的颜色。纹理的用处并非仅此而已。简单来讲，1D、2D、3D纹理就相当于特定数据元素所构成的1D、2D、3D数组。

纹理只能存储DXGI\_FORMAT枚举类型中描述的特定格式的数据元素。

DXGI\_FORMAT\_R32G32B32\_FLOAT

DXGI\_FORMAT\_R16G16B16\_UNORM

DXGI\_FORMAT\_R32G32\_UINT

DXGI\_FORMAT\_R8G8B8A8\_UNORM

DXGI\_FORMAT\_R8G8B8A8\_SNORM

DXGI\_FORMAT\_R8G8B8A8\_SINT

DXGI\_FORMAT\_R8G8B8A8\_UINT

有无类型格式的纹理，我们仅用它来预留内存，待纹理被绑定到渲染流水线之后，再具体解释它的数据类型。例如：DXGI\_FORMAT\_R16G16B16A16\_TYPELESS。

**4.1.4 交换链和页面翻转**

为了避免动画中出现画面闪烁的现象，最好将动画帧完整地绘制在一种称为后台缓冲区的离屏纹理内。为此，需要利用硬件管理两种纹理缓冲区：前台缓冲区和后台缓冲区。前后台缓冲区的交换操作称为呈现。

前后台缓冲区构成了交换链，在Direct3D中用IDXGISwapChain接口来表示。

使用两个缓冲区(前台和后台)的情况称为双缓冲。当然，也可以运用更多的缓冲区。例如三重缓冲。

**4.1.5 深度缓冲**

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。深度值的范围为0-1。0代表观察者在视锥体中能看到离自己最近的物体。深度缓冲区的元素与后台缓冲区内的像素呈一一对应关系。所以，如果后台缓冲区的分辨率为1280x1024，那么深度缓冲区中就应当有1280x1024个深度元素。

若使用了深度缓冲，则物体的绘制顺序也就变得无关紧要了。

深度缓冲可用的格式：

DXGI\_FORMAT\_D32\_FLOAT\_S8X24\_UINT

DXGI\_FORMAT\_D32\_FLOAT

DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT

DXGI\_FORMAT\_D16\_ UNORM

**4.1.6 资源与描述符**

GPU资源并非直接与渲染流水线相绑定，而是通过一种名为描述符的对象来对它间接引用，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。从本质上来讲，它实际上即为一个中间层，若指定了资源描述符，GPU将既能获得实际的资源数据，也能了解到资源的必要信息。

视图与描述符是同义词，例如，常量缓冲区视图与常量缓冲区描述符表达的是同一个事物。

本书常用的描述符如下：

1. CBV/SRV/UAV描述符分别表示的是常量缓冲区视图、着色器资源视图和无序访问视图；
2. 采样器描述符表示的是采样器资源；
3. RTV描述符表示的是渲染目标视图资源；
4. DSV描述符表示的是深度/模板视图资源；

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。

我们能用多个描述符来应用同一个资源。

创建描述符的最佳时机为初始化期间。最好不要在运行时才创建描述符。

**4.1.7 多重采样技术的原理**

超级采样，可简记作SSAA，它使用4倍与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时，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。

Direct3D还支持多重采样，可简记为MSAA。这种技术通过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。现假设采用X4多重采样，并同样使用4倍于屏幕分辨率的后台缓冲区和深度缓冲区。值得注意的是，这种技术并不需要对每一个子像素进行计算，而是仅计算一次像素中心处的颜色，再基于可视性(每个子像素经深度/模板测试)和覆盖性(子像素的中心在多边形的里面还是外面)将得到的颜色信息分享给其子像素。

**4.1.8 利用Direct3D进行多重采样**

填写DXGI\_SAMPLE\_DESC结构体。Count指定每个像素的采样次数，Quality用于指示图像质量级别。

**4.1.9 功能级别**

功能级别为不同级别所支持的功能进行了严格的界定。只要了解所支持的功能集，就能知道有哪些Direct3D功能可供使用。

**4.1.10 DirectX图形基础结构**

DirectX图形基础结构(DXGI)是一种与Direct3D配合使用的API。设计DXGI的基本理念是使多种图形API中所共有的底层任务能借助一组通用API来进行处理。

IDXGIFactory是DXGI中的关键接口之一，主要用于创建IDXGISwapChain接口以及枚举显示适配器。而显示适配器则真正实现了图形处理能力。通常来说，显示适配器是一种硬件设备(例如独立显卡)，然而系统也可以用软件显示适配器来模拟硬件的图形处理功能。一个系统中可能会存在数个适配器。适配器用接口IDXGIAdapter表示。

一个系统也可能装有数个显示设备。我们称每一台显示设备都是一个显示输出实例，用IDXGIOutput接口来表示。每个适配器都与一组显示输出相关联。

每种显示设备都有一系列它所支持的显示模式，用DXGI\_MODE\_DESC结构体表示。

**4.1.11 功能支持的检测**

我们通过ID3D12Device::CheckFeatureSupport方法，检测当前图形驱动对功能的支持。

**4.1.12 资源驻留**

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。

一般来说，资源在创建时就会驻留在显存中，而当它被销毁时清出。但是通过MakeResident我们可以自己控制资源的驻留。

**4.2 CPU与GPU间的交互**

CPU和GPU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同时工作，少同步。同步意味着一种处理器要以空闲的状态等待另一个处理器完成某些任务。

**4.2.1 命令队列和命令列表**

每个CPU都至少维护者一个命令队列(本质上是环形缓冲区)。借助API，CPU可利用命令列表将命令提交到这个队列中去。当一系列命令被提交到命令队列之时，它们并不会被GPU立即执行，理解这一点至关重要。

假如命令队列中空空如也，那么没有任务可执行的GPU只能空闲下来；相反，如果命令队列被填满，那么CPU必将随着GPU的工作步伐在某些时刻保持空闲。这两种情况都是我们不希望碰到的。

在Direct3D 12中，命令队列被抽象为ID3D12CommandQueue接口来表示。通过填写D3D12\_COMMAND\_QUEUE\_DESC结构体来描述队列，在调用CreateCommandQueue方法创建队列。

IID\_PPV\_ARGS宏详解。

ExecuteCommandLists接口可将命令列表中的命令添加到命令队列之中。

ID3D12GraphicsCommandList接口封装了一系列图形渲染命令，他有数种方法向命令列表中添加命令。

调用ExecuteCommandLists方法将命令真正地送入命令队列，供GPU在合适的时机处理。当命令都被加入命令列表后，调用Close方法结束命令的记录。调用ExecuteCommandLists之前要先调用Close方法。

还有一种与命令列表相关的名为ID3D12CommandAllocator的内存管理类接口。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上。当通过ExecuteCommandLists方法执行命令列表时，命令队列就会引用分配器中的命令。命令分配器由ID3D12Device的CreateCommandAllocator接口创建。

我们可以创建多个关联于同一命令分配器的命令列表，但是不能同时用它们记录命令。因此，当其中的一个命令列表在记录命令时，必须关闭同一个分配器的其他命令列表。

重置命令列表并不会影响命令队列中的命令，因为相关的命令分配器仍在维护着其内存中被命令队列引用的系列命令。

ID3D12CommandAllocator::Reset方法重置命令分配器的内存。

**4.2.2 CPU与GPU间的同步**

假设有一个资源R，存有几何体位置信息。现在，令CPU对R中数据进行更新，把位置信息改为p1，再向队列中添加绘制资源命令，如果CPU在执行绘制前将位置改为p2，就会造成一个严重的错误。

解决此问题的一个方法是：强制CPU等待，知道GPU完成所有命令的处理，达到某个指定的围栏点为止。我们将这种方法称为刷新命令队列，我们可以通过围栏来实现这一点。围栏用ID3D12Fence接口来表示，此技术用于实现GPU和CPU间的同步。

**4.2.3 资源转换**

我们经常会通过GPU对某个资源R按顺序进行先写后读这两种操作。然而，如果写操作还没有完成抑或没有开始，却开始读取资源，便会导致资源冒险。为此，Direct3D专门针对资源设计了一组相关状态。

通过命令列表设置转换资源屏障数组，可指定资源的转换：当我们希望以一次API调用来转换多个资源的时候，这种数组就派上了用场。资源屏障用D3D12\_RESOURCE\_BARRIER结构体来表示。

Direct3D 12中许多结构体都有其对应的扩展辅助结构体，更便于使用。以CD3DX12作为前缀的变体全都定义在d3dx12.h头文件当中，这个文件可以从官网下载获得。

**4.2.4 命令与多线程**

Direct3D 12的设计目标是为用户提供一个搞笑的多线程环境，命令列表也是一种发挥Direct3D多线程优势的途径。对于一个庞大的场景，我们可以创建4条线程，每条线程分别负责构建一个命令列表来绘制25%的场景物体。

读者可以通过查阅SDK中的Multithreading12示例学习如何并行生成命令列表。如果希望应用程序充分利用系统资源，应该通过多线程技术来发挥CPU多核心的并行处理能力。

**4.3 初始化Direct3D**

初始化步骤：

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. 设置视口和裁剪矩形；

**4.3.1 创建设备**

**4.3.2 创建围栏并获取描述符的大小**

**4.3.3 检测对4X MSAA质量级别的支持**

**4.3.4 创建命令队列和命令列表**

**4.3.5 描述并创建交换链**

**4.3.6 创建描述符堆**

**4.3.7 创建渲染目标视图**

**4.3.8 创建深度/模板缓冲区及视图**

**4.3.9 设置视口**

**4.3.10 设置裁剪矩阵**

**4.4 计时与动画**

**4.4.1 性能计时器**

为了精确地度量时间，我们将采用性能计时器。

性能计时器所用的时间度量单位叫做计数。可调用QueryPerformanceCounter函数来获取当前时刻值。

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

**4.4.2 游戏计时器类**

**4.4.3 帧与帧之间的时间间隔**

**4.4.4 总时间**

**4.5 应用程序框架示例**

Win32编程参考附录A。

**4.5.1 D3DApp类**

**4.5.2 非框架方法**

**4.5.3 框架方法**

**4.5.4 帧的统计信息**

**4.5.5 消息处理函数**

**4.5.6 初始化Direct3D演示程序**

**4.6 调试Direct3D应用程序**