# 4.1 准备工作

Direct3D的初始化过程要求我们熟悉一些基本的Direct3D类型和基本绘图概念；本章第一节会向读者介绍些必要的基础知识。然后我们会详细讲解Direct3D初始化过程中的每一个必要步骤，并顺便介绍一下实时绘图应用程序必须使用的精确计时和时间测量。最后，我们将讨论一下示例框架代码，它是本书所有演示程序使用的统一编程接口。

## 学习目标：

* 对Direct3D在规划调度3D硬件方面所起的作用有一个基本了解。
* 理解COM在Direct3D运行时起到的作用。
* 学习基本的绘图概念，比如2D图像的存储方式、页面翻转、深度缓存和多重采样。
* 学习如何使用性能计数器函数来获取高精度的计时器读数。
* 阐述Direct3D的初始化过程。
* 熟悉本书所有演示程序采用的通用应用程序框架结构。

Direct3D的初始化过程要求我们熟悉一些基本的绘图概念和Direct3D类型。我们会在本节讲解这些概念和类型，以使读者可以顺利地阅读之后的章节。

## 4.1.1 Direct3D概述

Direct3D是一种底层绘图**API（application programming interface，应用程序接口）**，它可以让我们可以通过3D硬件加速绘制3D世界。从本质上讲，Direct3D提供的是一组软件接口，我们可以通过这组接口来控制绘图硬件。例如，要命令绘图设备清空渲染目标（例如屏幕），我们可以调用Direct3D的**ID3D11DeviceContext::ClearRenderTargetView**方法来完成这一工作。Direct3D层位于应用程序和绘图硬件之间，这样我们就不必担心3D硬件的实现细节，只要设备支持Direct3D 11，我们就可以通过Direct3D 11 API来控制3D硬件了。

支持Direct3D 11的设备必须支持Direct3D 11规定的整个功能集合以及少数的额外附加功能（有一些功能，比如多重采样数量，仍然需要以查询方式实现，这是因为不同的Direct3D硬件这个值可能并不一样）。在Direct3D 9中，设备可以只支持Direct3D 9的部分功能；所以，当一个Direct3D 9应用程序要使用某一特性时，应用程序就必须先检查硬件是否支持该特性。如果要调用的是一个不为硬件支持Direct3D函数，那应用程序就会出错。而在Direct3D 11中，不需要再做这种设备功能检查，因为Direct3D 11强制要求设备实现Direct3D 11规定的所有功能特性。

## 4.1.2 COM

组件对象模型（COM）技术使DirectX独立于任何编程语言，并具有版本向后兼容的特性。我们经常把COM对象称为接口，并把它当成一个普通的C++类来使用。当使用C++编写DirectX程序时，许多COM的底层细节都不必考虑。唯一需要知道的一件事情是，我们必须通过特定的函数或其他的COM接口方法来获取指向COM接口的指针，而不能用C++的**new**关键字来创建COM接口。另外，当我们不再使用某个接口时，必须调用它的**Release**方法来释放它（所有的COM接口都继承于**IUnknown**接口，而**Release**方法是**IUnknown**接口的成员），而不能用**delete**语句——COM对象在其自身内部实现所有的内存管理工作。

当然，有关COM的细节还有很多，但是在实际工作中只需知道上述内容就足以有效地使用DirectX了。

**注意**：COM接口都以大写字母“I”为前缀。例如，表示2D纹理的接口为**ID3D11Texture2D**。

## 4.1.3 纹理和数据资源格式

2D纹理（texture）是一种数据元素矩阵。2D纹理的用途之一是存储2D图像数据，在纹理的每个元素中存储一个像素颜色。但这不是纹理的唯一用途；例如， 有一种称为法线贴图映射（normal mapping）的高级技术在纹理元素中存储的不是颜色，而是3D向量。因此，从通常意义上讲，纹理用来存储图像数据，但是在实际应用中纹理可以有更广泛的用途。1D纹理类似于一个1D数据元素数组，3D纹理类似于一个3D数据元素数组。但是在随后的章节中我们会讲到，纹理不仅仅是一个数据数组；纹理可以带有多级渐近纹理层（mipmap level），GPU可以在纹理上执行特殊运算，比如使用过滤器（filter）和多重采样（multisampling）。此外，不是任何类型的数据都能存储到纹理中的；纹理只支持特定格式的数据存储，这些格式由**DXGI\_FORMAT**枚举类型描述。一些常用的格式如下：

* **DXGI\_FORMAT\_R32G32B32\_FLOAT**：每个元素包含3个32位浮点分量。
* **DXGI\_FORMAT\_R16G16B16A16\_UNORM**：每个元素包含4个16位分量，分量的取值范围在[0,1]区间内。
* **DXGI\_FORMAT\_R32G32\_UINT**：每个元素包含两个32位无符号整数分量。
* **DXGI\_FORMAT\_R8G8B8A8\_UNORM**：每个元素包含4个8位无符号分量，分量的取值范围在[0,1]区间内。
* **DXGI\_FORMAT\_R8G8B8A8\_SNORM**：每个元素包含4个8位有符号分量，分量的取值范围在[−1,1] 区间内。
* **DXGI\_FORMAT\_R8G8B8A8\_SINT**：每个元素包含4个8位有符号整数分量，分量的取值范围在[−128, 127] 区间内。
* **DXGI\_FORMAT\_R8G8B8A8\_UINT**：每个元素包含4个8位无符号整数分量，分量的取值范围在[0, 255]区间内。

注意，字母R、G、B、A分别表示red（红）、green（绿）、blue（蓝）和alpha（透明度）。每种颜色都是由红、绿、蓝三种基本颜色组成的（例如，黄色是由红色和绿色组成的）。alpha通道（或alpha分量）用于控制透明度。不过，正如我们之前所述，纹理存储的不一定是颜色信息；例如，格式**DXGI\_FORMAT\_R32G32B32\_FLOAT**包含3个浮点分量，可以存储一个使用浮点坐标的3D向量。另外，还有一种**弱类型（typeless）**格式，可以预先分配内存空间，然后在纹理绑定到管线时再指定如何重新解释数据内容（这一过程与C++中的数据类型转换颇为相似）；例如，下面的弱类型格式为每个元素预留4个8位分量，且不指定数据类型（例如：整数、浮点数、无符号整数）：

**DXGI\_FORMAT\_R8G8B8A8\_TYPELESS**

## 4.1.4 交换链和页面翻转

为了避免在动画中出现闪烁，最好的做法是在一个离屏（off-screen）纹理中执行所有的动画帧绘制工作，这个离屏纹理称为**后台缓冲区（back buffer）**。当我们在后台缓冲区中完成给定帧的绘制工作后，便可以将后台缓冲区作为一个完整的帧显示在屏幕上；使用这种方法，用户不会察觉到帧的绘制过程，只会看到完整的帧。从理论上讲，将一帧显示到屏幕上所消耗的时间小于屏幕的垂直刷新时间。硬件会自动维护两个内置的纹理缓冲区来实现这一功能，这两个缓冲区分别称为**前台缓冲区（front buffer）**和后台缓冲区。前台缓冲区存储了当前显示在屏幕上的图像数据，而动画的下一帧会在后台缓冲区中执行绘制。当后台缓冲区的绘图工作完成之后，前后两个缓冲区的作用会发生翻转：后台缓冲区会变为前台缓冲区， 而前台缓冲区会变为后台缓冲区，为下一帧的绘制工作提前做准备。我们将前后缓冲区功能互换的行为称做**呈现（presenting）**。提交是一个运行速度很快的操作，因为它只是将前台缓冲区的指针和后台缓冲区的指针做了一个简单的交换。图4.1说明了这一过程。



**图4.1：我们首先渲染缓冲区B，它是当前的后台缓冲区。一旦帧渲染完成，前后缓冲区的指针会相互交换，缓冲区B会变为前台缓冲区，而缓冲区A会变为新的后台缓冲区。之后，我们将在缓冲区A中进行下一帧的渲染。一旦帧渲染完成，前后缓冲区的指针会再次进行交换，缓冲区A会变为前台缓冲区，而缓冲区B会再次变为后台缓冲区。**

前后缓冲区形成了一个**交换链（swap chain）**。在Direct3D中，交换链由**IDXGISwapChain**接口表示。该接口保存了前后缓冲区纹理，并提供了用于调整缓冲区尺寸的方法（**IDXGISwapChain::ResizeBuffers**）和呈现方法（**IDXGISwapChain::Present**）。我们会在4.4节中详细讨论些方法。

使用（前后）两个缓冲区称为**双缓冲（double buffering）**。缓冲区的数量可多于两个；比如，当使用三个缓冲区时称为**三缓冲（triple buffering**）。不过，两个缓冲区已经足够用了。

**注意**：虽然后台缓冲区是一个纹理（纹理元素称为texel），但是我们更习惯于将纹理元素称为像素（pixel），因为后台缓冲区存储的是颜色信息。有时，即使纹理中存储的不是颜色信息，人们还是会将纹理元素称为像素（例如，“法线贴图像素”）。

## 4.1.5 深度缓冲区

**深度缓冲区（depth buffer）**是一个不包含图像数据的纹理对象。在一定程度上，深度信息可以被认为是一种特殊的像素。常见的深度值范围在0.0到1.0之间，其中0.0表示离观察者最近的物体，1.0表示离观察者最远的物体。深度缓冲区中的每个元素与后台缓冲区中的每个像素一一对应（即，后台缓冲区的第ij个元素对应于深度缓冲区的第ij个元素）。所以，当后台缓冲区的分辨率为1280×1024时，在深度缓冲区中有1280×1024个深度元素。



**图4.2 彼此遮挡的一组物体**

图4.2是一个简单的场景，其中一些物体挡住了它后面的一些物体的一部分区域。为了判定物体的哪些像素位于其他物体之前，Direct3D使用了一种称为**深度缓存（depth**

**buffering）**或**z缓存（z-buffering）**的技术。我们所要强调的是在使用深度缓存时，我们不必关心所绘物体的先后顺序。

**注意**：要处理深度的问题，有人可能会建议按照从远至近的顺序绘制场景中的物体。使用这种方法，离得近的物体会覆盖在离得远的物体之上，这样就会产生正确的绘制结果，这也是画家作画时用到的方法。但是，这种方法会导致另一个问题——如何将大量的物体和相交的几何体按从远到近的方式进行排序？此外，图形硬件本身就提供了深度缓存供我们使用，因此我们不会采用画家算法。

为了说明深度缓存的工作方式，让我们来看一个例子。如图4.3所示，它展示的是观察者看到的立体空间（左图）以及该立体空间的2D侧视图（右图）。从这个图中我们可以发现，3个不同的像素会被渲染到视图窗口的同一个像素点P上。（当然，我们知道只有最近的像素会被渲染到P上，因为它挡住了后面的其他像素，可是计算机不知道这些事情。）首先，在渲染之前，我们必须把后台缓冲区清空为一个默认颜色（比如黑色或白色），把深度缓冲区清空为默认值——通常设为1.0（像素所具有的最远深度值）。



**图4.3：视图窗口相当于从3D场景生成的2D图像（后台缓冲区）。我们看到，有3个不同的像素可以被投影到像素P上。直觉告诉我们，P1是P的最终颜色，因为它离观察者最近，而且遮挡了其他两个像素。深度缓冲区算法提供了一种可以在计算机上实现的判定过程。注意，我们所说的深度值是相对于观察坐标系而言的。实际上，当深度值存入深度缓冲区时，它会被规范到[0.0,1.0]区间内。**

现在，假设物体的渲染顺序依次为：圆柱体、球体和圆锥体。下面的表格汇总了在绘制些物体时像素P及相关深度值的变化过程；其他像素的处理过程与之类似。

**表 4.1**

|  |  |  |  |
| --- | --- | --- | --- |
| 操作 | P | d | 说明 |
| 清空 | 黑色 | 1.0 | 初始化像素以及相应的深度元素。 |
| 绘制圆柱体 | P3 | d3 | 因为d3≤d＝1.0，所以深度测试通过，更新缓冲区，设置P＝P3、d＝d3。 |
| 绘制球体 | P1 | d1 | 因为d1≤d＝d3，所以深度测试通过，更新缓冲区，设置P＝P1、d＝d1。 |
| 绘制圆锥体 | P1 | d1 | 因为d2＞d＝d1，所以深度测试未通过，不更新缓冲区。 |

从上表可以看到，当我们发现某个像素具有更小的深度值时，就更新该像素以及它在深度缓冲区中的相应深度值。通过一方式，在最终得到的渲染结果中只会包含那些离观察者最近的像素。（如果读者对此仍有疑虑，那么可以试着交换本例的绘图顺序，看看得到的计算结果是否相同。）

综上所述，深度缓冲区用于为每个像素计算深度值和实现深度测试。深度测试通过比较像素深度来决定是否将该像素写入后台缓冲区的特定像素位置。只有离观察者最近的像素才会胜出，成为写入后台缓冲区的最终像素。这很容易理解，因为离观察者最近的像素会遮挡它后面的其他像素。

深度缓冲区是一个纹理，所以在创建它时必须指定一种数据格式。用于深度缓存的格式如下：

* **DXGI\_FORMAT\_D32\_FLOAT\_S8X24\_UINT**：32位浮点深度缓冲区。为模板缓冲区预留8位（无符号整数），每个模板值的取值范围为[0,255]。其余24位闲置。
* **DXGI\_FORMAT\_D32\_FLOAT**：32位浮点深度缓冲区。
* **DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT**：无符号24位深度缓冲区，每个深度值的取值范围为[0,1]。为模板缓冲区预留8位（无符号整数），每个模板值的取值范围为[0,255]。
* **DXGI\_FORMAT\_D16\_UNORM**：无符号16位深度缓冲区，每个深度值的取值范围为[0,1]。

**注意**：模板缓冲区对应用程序来说不是必须的，但是如果用到了模板缓冲区，那么模板缓冲区必定是与深度缓冲区存储在一起的。例如，32位格式**DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT**使用24位用于深度缓冲区，8位用于模板缓冲区。 所以，将深度缓冲区称为“深度/模板缓冲区”更为合适。模板缓冲区是一个比较高级的主题，我们会在第10章讲解模板缓冲区的用法。

## 4.1.6 纹理资源视图

纹理可以被绑定到**渲染管线（rendering pipeline）**的不同**阶段（stage）**；例如，比较常见的情况是将纹理作为渲染目标（即，Direct3D渲染到纹理）或着色器资源（即，在着色器中对纹理进行采样）。当创建用于这两种目的的纹理资源时，应使用绑定标志值：

**D3D11\_BIND\_RENDER\_TARGET | D3D10\_BIND\_SHADER\_RESOURCE**

指定纹理所要绑定的两个管线阶段。其实，资源不能被直接绑定到一个管线阶段；我们只能把与资源关联的资源视图绑定到不同的管线阶段。无论以哪种方式使用纹理，Direct3D始终要求我们在初始化时为纹理创建相关的**资源视图（resource view）**。这样有助于提高运行效率，正如SDK文档指出的那样：“运行时环境与驱动程序可以在视图创建执行相应的验证和映射，减少绑定时的类型检查”。所以，当把纹理作为一个渲染目标和着色器资源时，我们要为它创建两种视图：渲染目标视图（**ID3D11RenderTargetView**）和着色器资源视图（**ID3D11ShaderResourceView**）。资源视图主要有两个功能：（1）告诉Direct3D如何使用资源（即，指定资源所要绑定的管线阶段）；（2）如果在创建资源时指定的是弱类型（typeless）格式，那么在为它创建资源视图时就必须指定明确的资源类型。对于弱类型格式，纹理元素可能会在一个管线阶段中视为浮点数，而在另一个管线阶段中视为整数。

为了给资源创建一个特定视图，我们必须在创建资源时使用特定的绑定标志值。例如，如果在创建资源没有使用**D3D11\_BIND\_DEPTH\_STENCIL**绑定标志值（该标志值表示纹理将作为一个深度/模板缓冲区绑定到管线上），那我们就无法为该资源创建**ID3D11DepthStencilView视图**。只要你试一下就会发现Direct3D会给出如下调试错误：

**ERROR: ID3D11Device::CreateDepthStencilView: A DepthStencilView cannot be created of a Resource that did not specify D3D10\_BIND\_DEPTH\_STENCIL.**

我们会在本章的4.2节中看到用来创建渲染目标视图和深度/模板视图的代码。在第8章中看到用于创建着色器资源视图的代码。本书随后的许多例子都有会把纹理用作渲染目标和着色器资源。

**注意**：2009年8月的SDK文档指出：“当创建资源时，为资源指定强类型（fully-typed）格式，把资源的用途限制在格式规定的范围内，有利于提高运行时环境对资源的访问速度……”。所以，你只应该在真正需要弱类型资源时（使用弱类型的优点是可以使用不同的视图将数据用于不同的用途），才创建弱类型资源；否则，应尽量创建强类型资源。

## 4.1.7 多重采样

因为计算机显示器上的像素分辨率有限，所以当我们绘制一条任意直线时，该直线很难精确地显示在屏幕上。图4.4中的第一条直线说明了“阶梯”（**aliasing，锯齿**）效应，当使用像素矩阵近似地表示一条直线时就会出现这种现象，类似的锯齿也会发生在三角形的边缘上。



**图4.4：我们可以看到，第一条直线带有明显的锯齿（当使用像素矩阵近似地表示一条直线时就会出现阶梯效应）。而第二条直线使用了抗锯齿技术，通过对一个像素周围的邻接像素进行采样得到该像素的最终颜色；这样可以形成一条较为平滑的直线，使阶梯效果得到缓解。**

通过提高显示器的分辨率，缩小像素的尺寸，也可以有效地缓解一问题，使阶梯效应明显降低。

当无法提高显示器分辨率或分辨率不够高时，我们可以使用**抗锯齿（antialiasing）**技术。其中的一种技术叫做**超级采样（supersampling）**，它把后台缓冲和深度缓冲的大小提高到屏幕分辨率的4倍。3D场景会以这个更大的分辨率渲染到后台缓存中，当在屏幕上呈现后台缓冲时，后台缓冲会将4个像素的颜色取平均值后得到一个像素的最终颜色。从效果上来说，超级采样的工作原理就是以软件的方式提升分辨率。

超级采样代价昂贵，因为它处理的像素数量和所需的内存数量增加为原来的4倍。Direct3D支持另一种称为**多重采样（multisampling）**的抗锯齿技术，它通过对一个像素的子像素进行采样计算出该像素的最终颜色，比超级采样节省资源。假如我们使用的是4X多重采样（每个像素采样4个邻接像素），多重采样仍然会使用屏幕分辨率4倍大小的后台缓冲和深度缓冲，但是，不像超级采样那样计算每个子像素的颜色，而是只计算像素中心颜色一次，然后基于子像素的可见性（基于子像素的深度/模板测试）和范围（子像素中心在多边形之外还是之内）共享颜色信息。图4.5展示了这样的一个例子。



图4.5：如图（a）所示，一个像素与多边形的边缘相交，像素中心的绿颜色存储在可见的三个子像素中，而第4个子像素没有被多边形覆盖，因此不会被更新为绿色，它仍保持为原来绘制的几何体颜色或**Clear**操作后的颜色。如图（b）所示，要获得最后的像素颜色，我们需要对4个子像素（3个绿色和一个白色）取平均值，获得淡绿色，通过这个操作，可以减弱多边形边缘的阶梯效果，实现更平滑的图像。

**注意**：supersampling与multisampling的关键区别在于：使用supersampling时，图像的颜色需要通过每个子像素的颜色计算得来，而每个子像素颜色可能不同；使用multisampling（图4.5）时，每个像素的颜色只计算一次，这个颜色会填充到所有可见的、被多边形覆盖的子像素中，即这个颜色是共享的。因为计算图像的颜色是图形管线中最昂贵的操作之一，因此multisampling相比supersampling而言节省的资源是相当可观的。但是，supersampling更为精确，这是multisampling做不到的。

**注意**：在图4.5中，我们用标准的网格图形表示一个像素的4个子像素，但由于硬件的不同，实际的子像素放置图形也是不同的，Direct3D并不定义子像素的放置方式，在特定情况下，某些放置方式会优于其他的放置方式。

## 4.1.8 Direct3D中的多重采样

在下一节中， 我们要填充一个**DXGI\_SAMPLE\_DESC**结构体。该结构体包含两个成员，其定义如下：

typedef struct DXGI\_SAMPLE\_DESC {

UINT Count;

UINT Quality;

} DXGI\_SAMPLE\_DESC, \*LPDXGI\_SAMPLE\_DESC;

**Count**成员用于指定每个像素的采样数量，**Quality**成员用于指定希望得到的质量级别（不同硬件的质量级别表示的含义不一定相同）。质量级别越高，占用的系统资源就越多，所以我们必须在质量和速度之间权衡利弊。质量级别的取值范围由纹理格式和单个像素的采样数量决定。我们可以使用如下方法，通过指定纹理格式和采样数量来查询相应的质量级别：

HRESULT ID3D11Device::CheckMultisampleQualityLevels(

DXGI\_FORMAT Format, UINT SampleCount, UINT \*pNumQualityLevels);

如果纹理格式和采样数量的组合不被设备支持，则该方法返回0。反之，通过pNumQualityLevels参数返回符合给定的质量等级数值。有效的质量级别范围为0到pNumQualityLevels−1。

采样的最大数量可以由以下语句定义：

#define D3D11\_MAX\_MULTISAMPLE\_SAMPLE\_COUNT(32)

采样数量通常使用4或8，可以兼顾性能和内存消耗。如果你不使用多重采样，可以将采样数量设为1，将质量级别设为0。所有符合Direct3D 11功能特性的设备都支持用于所有渲染目标格式的4X多重采样。

**注意**：我们需要为交换链缓冲区和深度缓冲区各填充一个**DXGI\_SAMPLE\_DESC**结构体。当创建后台缓冲区和深度缓冲区时，必须使用相同的多重采样设置；具体的代码会在下一节给出。

## 4.1.9 特征等级

Direct3D 11提出了特征等级（feature levels，在代码中由枚举类型**D3D\_FEATURE\_LEVEL**表示）的概念，对应了定义了d3d11中定义了如下几个等级以代表不同的d3d版本：

typedef 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;

特征等级定义了一系列支持不同d3d功能的相应的等级（每个特征等级支持的功能可参见SDK文档），用意即如果一个用户的硬件不支持某一特征等级，程序可以选择较低的等级。例如，为了支持更多的用户，应用程序可能需要支持Direct3D 11，10.1，9.3硬件。程序会从最新的硬件一直检查到最旧的，即首先检查是否支持Direct3D 11，第二检查Direct3D 10.1，然后是Direct3D 10，最后是Direct3D 9。要设置测试的顺序，可以使用下面的特征等级数组（数组内元素的顺序即特征等级测试的顺序）：

D3D\_FEATURE\_LEVEL featureLevels [4] =

{

D3D\_FEATURE\_LEVEL\_11\_0, // First check D3D 11 support

D3D\_FEATURE\_LEVEL\_10\_1, // Second check D3D 10.1 support

D3D\_FEATURE\_LEVEL\_10\_0, // Next,check D3D 10 support

D3D\_FEATURE\_LEVEL\_9\_3 // Finally,check D3D 9.3 support

} ;

这个数组可以放置在Direct3D初始化方法（4.2.1节）中，方法会输出数组中第一个可被支持的特征等级。例如，如果Direct3D报告数组中第一个可被支持的特征等级是**D3D\_FEATURE\_LEVEL\_10\_0**，程序就会禁用Direct3D 11和Direct3D 10.1的特征，而使用Direct3D 10的绘制路径。本书中我们要求必须能支持**D3D\_FEATURE\_LEVEL\_11\_0**。