Managed DirectX 9.0 SDK Summer 2004 中文文档

译者：Wu Jie & Liu Kang

创建时间：2006-10-23

译者序

本中文文档根据Microsoft公开发布的DirectX SDK Summer 2004 帮助文档英文版翻译，不得用于商业用途，仅供学习交流参考。翻译内容选自官方帮助文档中关于Managed DirectX的章节，略去API Library规范参考。限于译者水平，文档中可能存在错误，欢迎指正，译者将不定期对此进行修正。目前DirectX SDK版本最新版本为October 2006，但文档内容变化不大，如该文档有重大更新，译者将适时发布新版中文文档。由于翻译工作利用业余时间进行，如不能及时更新文档请见谅。译者联系方式：gmmywin@gmail.com（Wu Jie）、luck@nj13z.cn（Liu Kang）。

目录

前言

第1章　托管代码版 DirectX 9.0

第1节　目的

Microsoft DirectX® 9.0 是通往活泼生动的多媒体应用程序的后台。

第2节　可用于何处

DirectX 是一系列低级的应用程序接口(APIs)，它用于创建游戏和其他高执行效率的多媒体程序。它包括对高效的 2-D 和 3-D 图形、音效和音乐、输入设备、力反馈设备、多媒体

流和例如多人游戏的网络通信程序。

第3节　开发者对象

该文档集合是为使用 Microsoft .NET Framework 的开发者准备的。对使用 C/C++语言的开发者另有单独文档可用。

第4节　运行时需求

DirectX 9.0 能运行在 Microsoft Windows 98、Windows Millennium Edition (Windows Me)、Windows 2000 和 Windows XP 环境。

第2章　托管代码版 DirectX 9.0 介绍

Microsoft® DirectX® 是创建游戏和其它高执行效率多媒体应用程序的一系列低级 API。它包括了对于 2-D 和 3-D 图形的支持，声音特效和音乐，输入设备和例如多人游戏这样的网络应用程序。

第1节　托管代码版 DirectX 9.0

在 DirectX 9.0 下，开发者在使用托管代码的时候，能够利用 DirectX 的多媒体功能和硬件加速。托管代码版 DirectX 9.0 允许访问大多数原始的非托管 DirectX 功能。下面是被 DirectX 9.0 和 DirectX 9.0 SDK 支持的托管代码语言：

Microsoft Visual C#
Microsoft Visual Basic .NET
Microsoft Visual C++
Microsoft JScript .NET

第2节　组件

托管代码版 DirectX 9.0 由以下主要组件构成。

Direct3D Graphics 提供了一个单一的 API，你能使用它进行 3-D 图形编程。
DirectDraw 提供直接的低级访问显存和高速渲染，不赞成使用！
DirectInput 提供对于多种输入设备的支持，包括对力反馈技术的完全支持。
DirectPlay 提供多人网络游戏的支持，不赞成使用！
DirectSound 提供播放和捕捉预录制数码采样的支持。
Audio Video Playback 允许回放和简单控制音频/视频媒体。

第3节　托管代码版 DirectX 9.0 的优点

通过消除 COM 组件对象模型的互通层，托管代码版 DirectX 9.0 改善了执行性能。托管代码能减少代码体积和提升工作效率。继承于强大易用的 Microsoft .NET Framework 公共类型的接口更加直观。托管代码也把你从处理很多内存管理的任务中解放了，这些任务比如释放对象。在 SDK 你将能找到托管 Visual C#范例和其它多种非托管副本的指南。

第4节　需求

最少需要支持托管代码版 DirectX 9.0 运行时的操作系统是 Microsoft Windows 98。可是 DirectX 9.0 SDK 中运行范例和工具的最低操作系统是 Windows 2000。

第3章　什么是托管代码？

托管代码使用超过 20 种高级程序语言之一来编写的，这些程序语言在 Microsoft® .NET Framework 是可用的，包括 C#, J#, Microsoft Visual Basic® .NET, Microsoft JScript® .NET, and C++。所有这些语言分享一系列统一的类库，并且能被编码为一种中间语言(IL)。在一个确保类型安全，数组边界和索引检查，异常处理和垃圾收集的托管执行环境里，运行时编译器将 IL 编译为本地执行代码。
通过使用托管代码和在托管执行环境中编译，你能避免许多典型的导致安全漏洞的编码错误和不稳定应用程序。许多没有生产效率的编码任务会被自动照料，例如类型安全检察、内存管理和无用的对象销毁。因此你能集中精神到程序的商务逻辑和使用较少的代码行编写它们。结果是较短的开发时间和更安全稳定的应用程序。

第4章　使用托管代码的提示和技巧

这个话题从基础上讨论了怎样安装和使用 Microsoft® DirectX® 9.0 托管版本及其文档。包括以下篇章。

第1节　托管代码版 DirectX 9.0 起步

安装最新的 Microsoft .NET Framework 运行时。这将 Framework 框架，包括.NET 运行时安装在你本地 PC。.NET 运行时是一套托管代码需要的动态链接库(DLLs)。你可以安装完整的软件开发工具包(SDK)，但是仅运行时是必须的。
从 DirectX站点，或者从MSDN®DirectX Downloads下载DirectX 9.0 SDK。SDK包括了运行DirectX 应用程序必须的DirectX DLLs。
注意：在安装新的发布版前，要卸载旧版本的 DirectX。使用 **Add or Remove Programs** 控制面板来删除一个早期版本。在它们运行之前，被开发的托管代码应用程序应该确定 DirectX 9.0 已经安装。

第2节　使用简单的应用程序

范例程序的预测试代码可以在以下位置找到：(*SDK root*)\Samples\Managed\
范例程序应该使用 Microsoft Visual Studio® .NET 2003 生成。
对于每个 DirectX 技术，你将在下面的文件夹结构中找到。

单独的 sample 范例文件夹： 包含了特别为托管代码设计的多种应用程序。在每个子文件夹下，你将发现一个完整的 Visual Studio .NET 2003 工程和你能快速启动的解决方案。

Bin 子文件夹： 包含了被编译的应用程序，你能通过双击运行之。

Tutorials 指南子文件夹(Microsoft Direct3D® 独有)：包含范例应用程序，那些范例会带你一步一步贯穿于创建一个完整的运行程序。Direct3D C# tutorials 指南的细节描绘在 第 8 章Direct3D 指南。

第3节　类的继承

DirectX 9.0 托管代码版在程序中不支持类的继承。由 DirectX 9.0 托管代码版类的继承不推荐使用，并且其行为结果是不确定的。
虽然你可能发现某些通过继承得到的类能和 DirectX 9.0 托管代码版一同工作，但是这些继承者可能不能在未来版本中工作。

第4节　调试 DirectX 应用程序

有 2 种方法来获得一个 DirectX 托管程序的调试信息。
从 Visual Studio .NET 2003 整合开发环境(IDE)：

在 **Solution Explorer** 面板，右击你的工程（不是解决方案）并且选择属性。

点击在 **Configuration Properties** 左边的文件夹，然后点击 **Debugging** 选项。

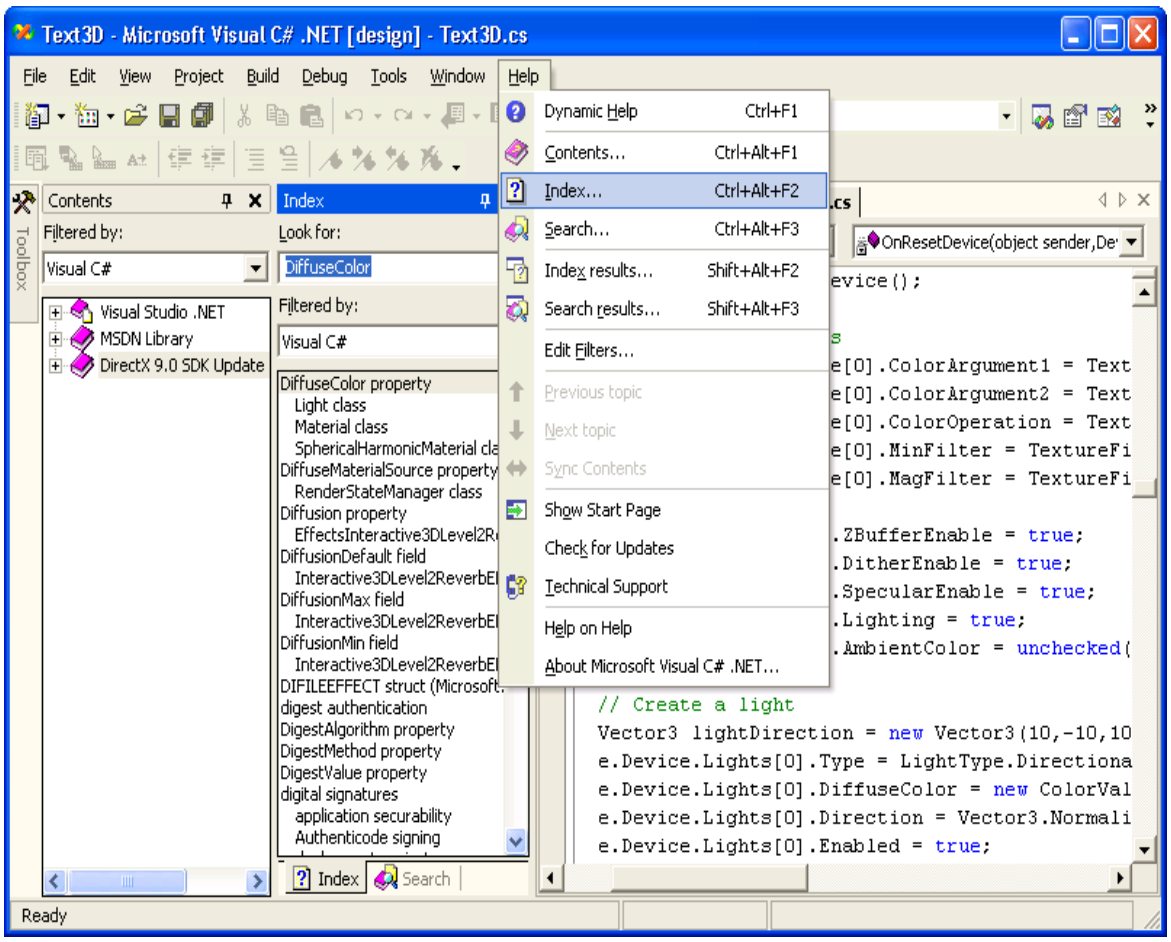
在 Debuggers 选项，设置 Enable Unmanaged Debugging 为 True。它默认设置为 False。

从 Dbmon.exe，它只运行在和 Microsoft Windows® 2000 and Windows XP：

从一个命令行窗体，运行 Dbmon.exe，它位于(*SDK root*)\Utilities\

第5节　使用 Visual Studio .NET 2003 里的 DirectX 托管版文档

当使用以上的手续加载 SDK 时，在 Visual Studio .NET 2003 中文档是易读取的。从 Visual Studio**Help** 菜单，选择 **Contents** 来打开一个内容的表格，**Index** 的搜索通过字母顺序或者文件中的文本，如下所示：



从托管代码里，你也能查看详细的文件和语法，通过点击类、结构体、成员、属性或字段的名称的任何地方。

第6节　使用 DirectX Help 文件

同时从 DirectX SDK 和 MSDN Library 安装 DirectX 文档会导致 Visual Studio .NET 2003 中主题重复。

1. 在 **Control Panel**，点击 **Add/Remove Programs** 图标。
2. 高亮显示 **Microsoft DirectX 9.0 SDK** ，并且单击 **Change**。
3. 当打开 **InstallShield Wizard** ，点击 **Next**。
4. 选择 **Modify**，点击 **Next**。
5. 在 **DirectX** 文件列表里，点击 **This feature will not be available**。
6. 点击 **Next** ，文件将被删除。

第7节　重新发布 DX

单机开发者运行时的安装程序仅为了重新发布目的而安装了 DirectX 开发者和调试运行时。这个安装程序位于：(*SDK root*)\Developer Runtime\Managed DirectX\ DirectX 9.0 托管代码版为了重新发布，包括了一系列完整的 DLLs。要安装全系列，作以下操作。

1. 验证 Framework 框架已被安装。
2. 定位到(*SDK root*)\Redist\DirectX9\。
3. 调用 DXSetup.exe /InstallManagedDX 命令行，如下：

```
DXSetup.exe /InstallManagedDX
```

第1篇 Direct3D 起步

这部分简要介绍了 Direct3D API 的三维图形功能，还附有指南帮助你快速建立程序运行。

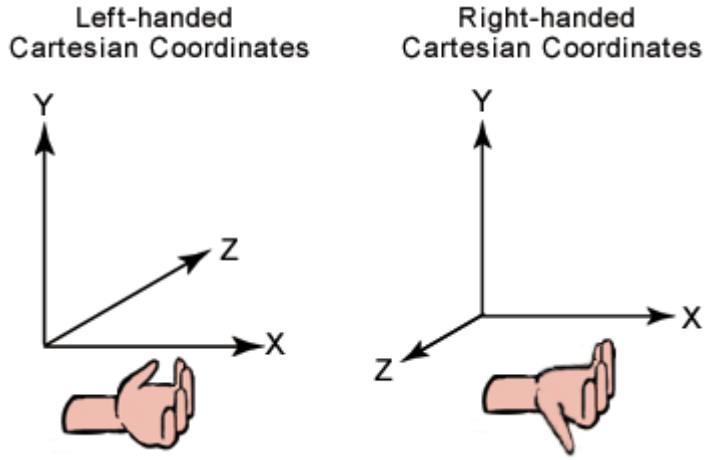
第1章 3-D 坐标系和几何学

Direct3D 编程需要熟悉三维几何学原理。这部分介绍了创建三维场景最重要的几何学概念。下列主题对 Direct3D 程序使用的基本概念作了高级描述。

3-D 坐标系
面和顶点法线
3-D 造型
光栅化规则
矩形
三角形内插值
向量、顶点和四元数

第1节 3-D 坐标系

通常，3-D 图形程序使用 2 种笛卡尔坐标系：左手坐标系和右手坐标系。这二种坐标系的 X 轴向右，Y 轴向上。你可以让手指指向 X 轴，弯曲手指指向 Y 轴，通过左右手的大拇指的方向来确定 Z 轴的方向。



Microsoft Direct3D 使用左手坐标系。如果你将基于右手坐标系的程序导出，必须把传给 Direct3D 的数据做 2 点改变。

翻转三角形顶点顺序，使坐标系从前方按顺时针方向遍历顶点。如顶点是 v0,v1,v2，以 v0,v2,v1 的顺序传给 Direct3D。

使用视角矩阵沿 Z 轴反方向度量世界空间。将你用于视角矩阵的 Matrix 结构体的 M31 、 M32 、 M33 、 M34 标记翻转。

如要获得右手坐标系中的全部世界，应使用 PerspectiveRH和 PerspectiveLH方法定义投影变换。但对应 LookAtRH函数的使用要注意，反序排列隐面消除，然后布置相应的立方贴图。

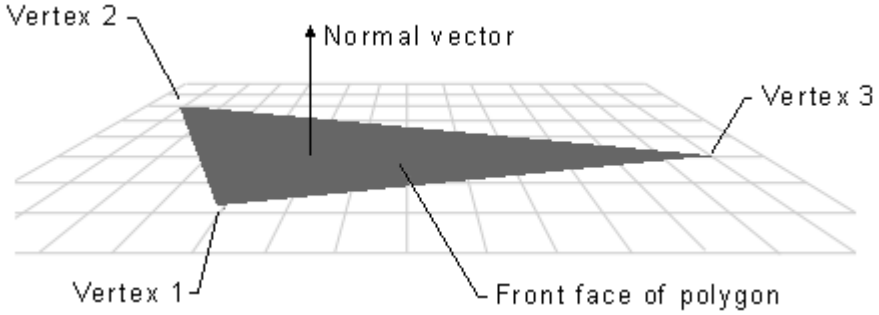
虽然左右手坐标系是常用坐标系，但在三维软件中还会使用多种其他坐标系。如三维建模程序使用一种不常用的 Y 轴指向观察者，Z 轴向上的坐标系统。这里右手被定义为 X、Y 或 Z 任何一正轴指向观察者，左手被定义为 X、Y 或 Z 任何一正轴背向观察者。如果你将基于 Z 轴向上的左手建模程序导出，除前面操作步骤之外，还必须旋转所有的顶点数据。

三维坐标系中定义的物体可执行的基本操作有：平移、旋转和缩放。你可以组合这些基本操作建立一个变换矩阵。

矩阵相乘的顺序是重要的，所以当你组合以上操作时，结果不可互换。

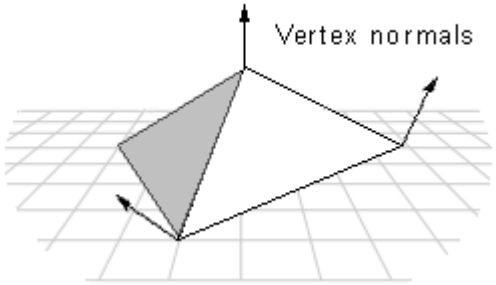
第2节　面和顶点法向量

网格中的每个表面都有一个垂直法线。它的方向取决于定义顶点的顺序和左右手坐标系类型。表面法线背离表面的正面指向外部。Direct3D 中仅表面的正面是可见的。正表面的顶点按顺时针序定义。



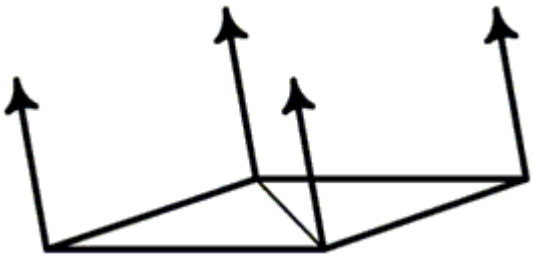
任何表面不是前表面就是后表面。Direct3D 不渲染后表面，因此后表面将被消隐。但你可以改变消隐模式来渲染你想要的背表面。

Direct3D 使用顶点法线进行高洛德着色、光照和纹理特效。



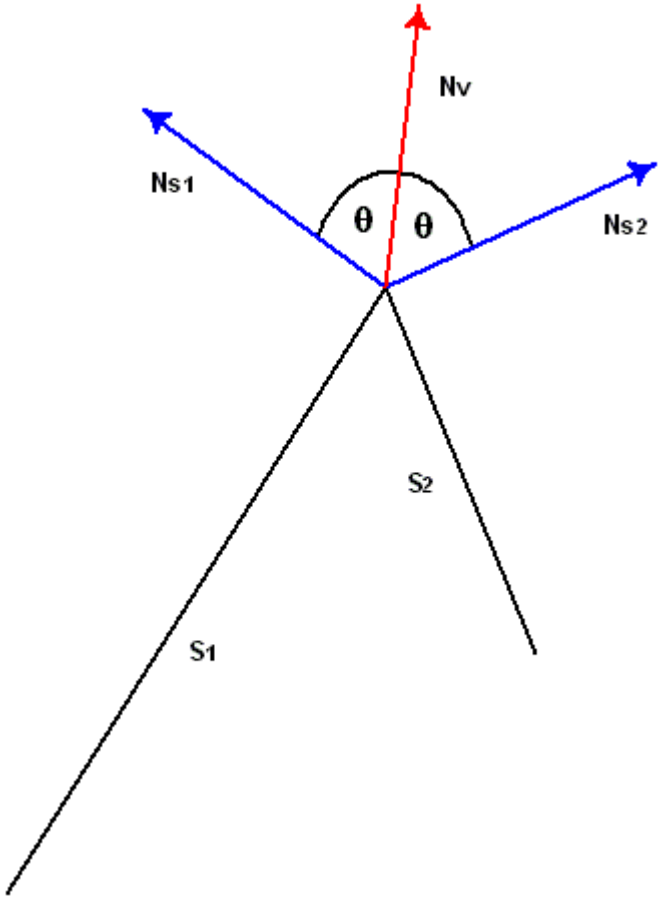
在多边形上使用高洛德着色时，Direct3D 使用法线计算光源和表面之间的角度。Direct3D 计算顶点的色彩和亮度值，并为造型所有表面上的点插值。Direct3D 通过角度计算光线亮度值，角度越大，表面的光线越暗。

如果你建立的是平面物体，设置如下图的表面垂直法线，这就定义了一个由两个三角形组成的平面。

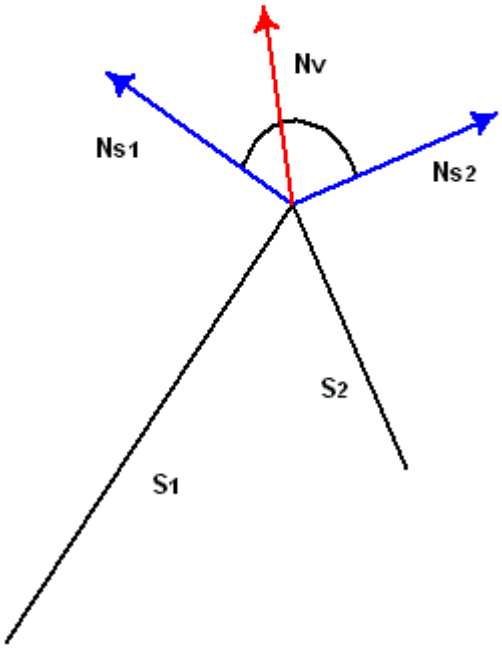


如果你的物体是由不共面的三角形条带组成，为这个条带上所有三角形平滑着色的简单方法是：首先给每个顶点关联的多边形表面计算曲面法线，顶点法线可以通过每个曲面法线设置成一个均等的角度。但这种方法对复杂造型来说效率低下。

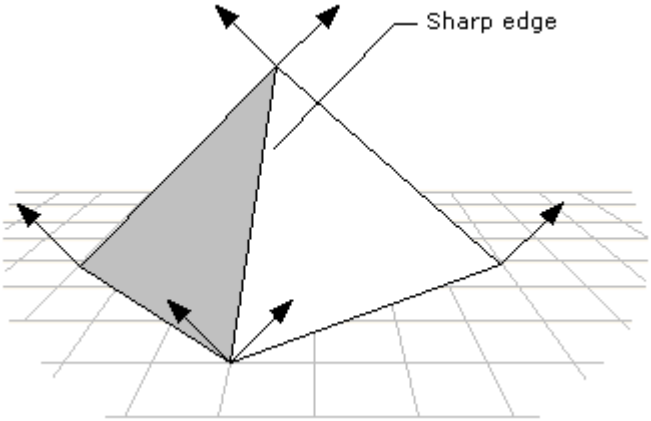
下图描绘了这种方法，有 S1、S2 两个 表面在上方相交。S1 和 S2 的法线用蓝色表示，顶点法线用红色表示。顶点法线和 S1、S2 的曲面法线所成角度相等。当两个表面使用高洛德着色和光照时，将会产生一个平滑的色彩渐变，它们之间会有平滑的过渡边缘。



如果顶点法线偏向它关联的表面中的一个，会导致有 1 个表面的光线亮度增加或减少，这取决于它和光源的角度。下图中顶点法线偏向 S1，导致顶点法线和光源的角度比顶点法线居中时角度减少。



你可以在三维场景中使用高洛德着色为物体显示清晰边界。如果需要一个清晰的边界，可以如下图复制任何相交表面的顶点法线。



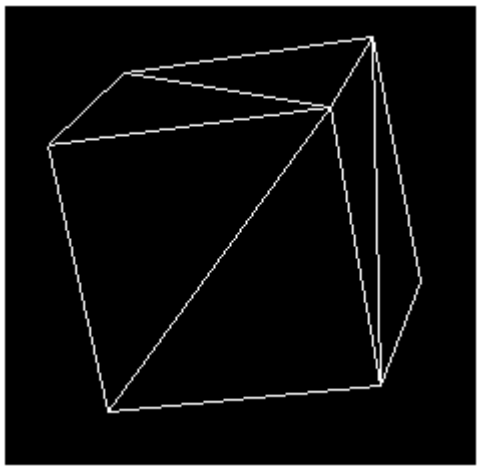
如果使用 DeviceDraw...Primitives方法渲染场景，应将具有清晰边界的物体定义为一个三角形列表，而不是三角形条带。当你将物体定义为三角形条带时，Direct3D认为这是一个由多个三角形表面组成的简单多边形，在多边形的每个表面和相邻表面之间应用高洛德着色。这会导致物体产生从表面到表面的平滑色彩过渡。而三角形列表是一个由一系列互不相连的三角形表面组成的多边形，Direct3D仅在多边形的每个表面应用高洛德着色，而不会从表面到表面。如果有三角形列表中有两个以上三角形相毗邻，这可以在它们之间产生清晰的边界。

渲染具有清晰边界物体的另一个折中方法是使用平面着色法。这是一种计算效率最高的方法，但它渲染场景中物体的效果不如高洛德着色法看上去真实。

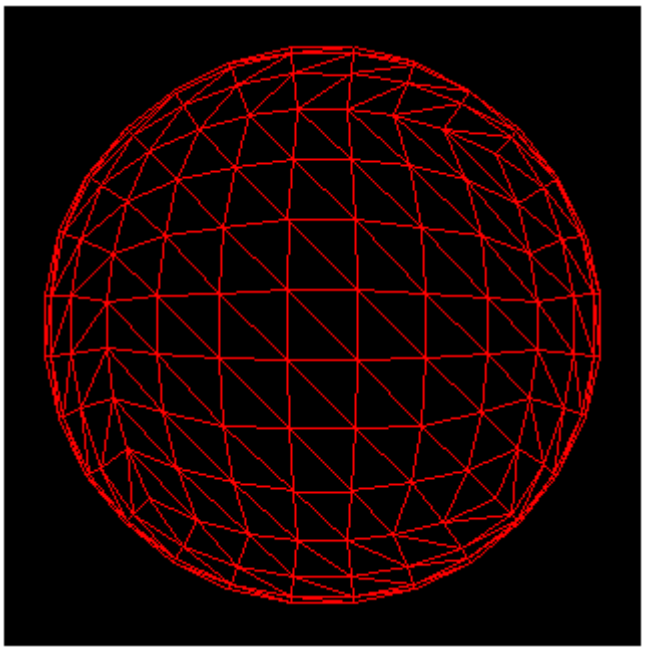
第3节　3-D 造型

3-D 造型是构建简单 3-D 实体的顶点集合。最简单的造型是 3-D 坐标系中的点集合，被称为点列表。

通常三维造型是多边形。多边形是由至少 3 个顶点组成的封闭三维轮廓。最简单的多边形是三角形。因为渲染不共面的顶点效率很低，而三角形中的 3 个顶点保证共面，所以 Direct3D 使用三角形构建大多数多边形。你可以组合三角形构成大而复杂的多边形和网格。下图是个立方体，2 个三角形构成立方体的一个面，全部三角形构成了一个立方体造型。你可以将纹理和材质应用到造型表面上，使它看上去像一个实心固体。更多细节，参看 第 7 章 Direct3D纹理 。



你还可以使用三角形创建表面为平滑曲面的造型。下图为用三角形模拟的球体。当材质应用后，被渲染球体看起来有了曲面感。如果使用高洛德着色，感觉更加真实。更多细节，参看 Gouraud Shading 。

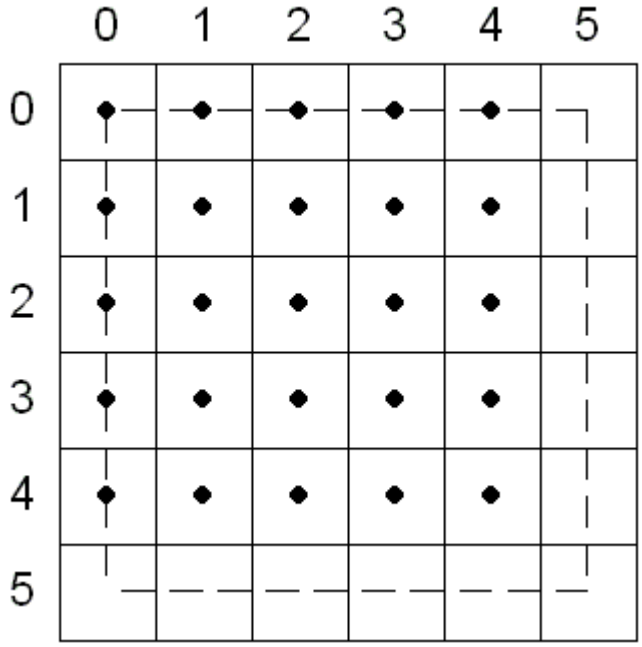


第4节　光栅化规则

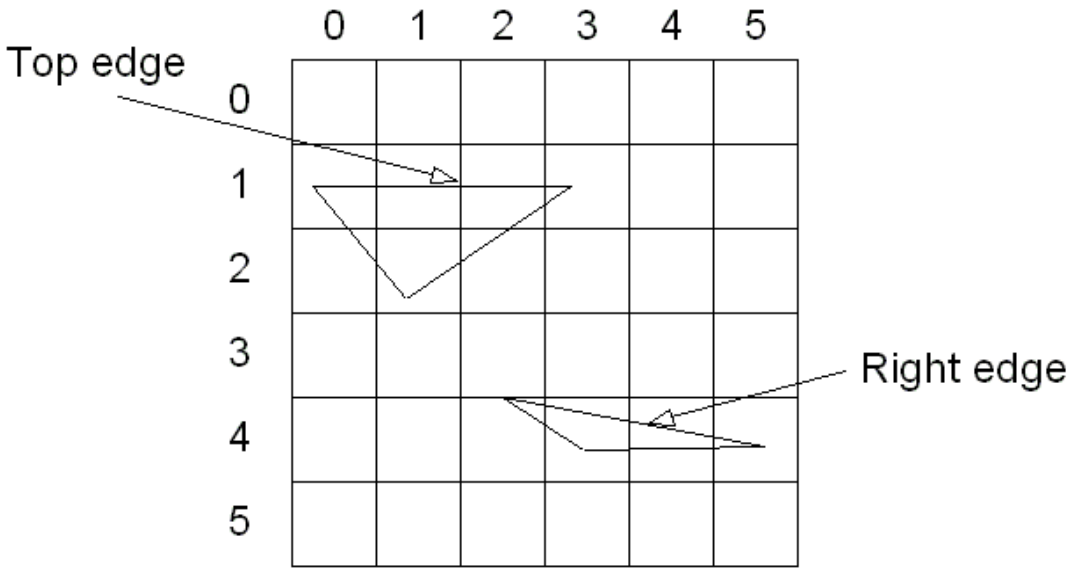
通常顶点不能和屏幕像素精确匹配。当这种情况发生时，Direct3D 使用三角形光栅规则决定哪个像素被用于给定三角形。

4.1 三角形光栅化规则

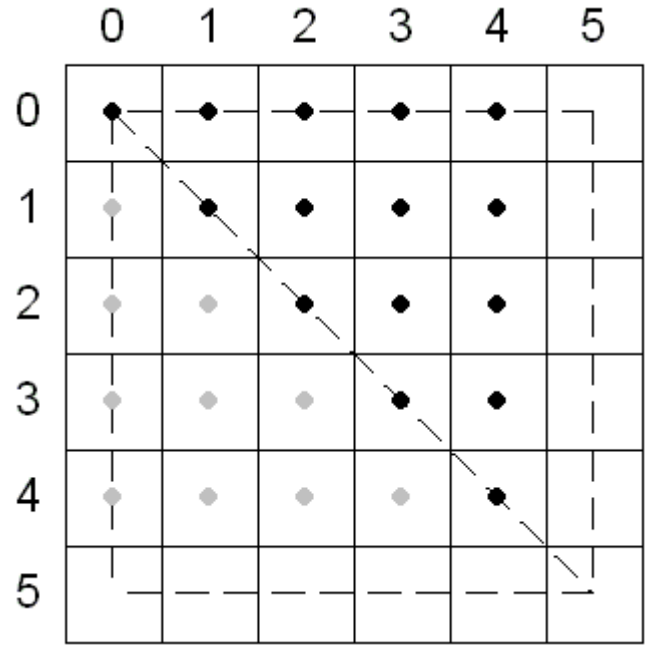
Direct3D 使用自上而下-从左到右的规则进行几何填充，这和 Microsoft Windows 图形设备接口(GDI)、OpenGL 使用的矩形填充规则相同。在 Direct3D 中，像素的中心是决定点，如果中心在三角形内，该像素就属于三角形部分。像素中心是坐标刻度的整数值。
Direct3D 使用的三角形光栅化规则并非用于所有可用硬件。你的测试也许已经发现这些规则的一些微小变化。
下图是一个左上角坐标(0, 0)右下角坐标(5, 5)的矩形，它包含了 25 个像素。矩形宽度定义为自右向左减小，高度定义为自底向上减小。



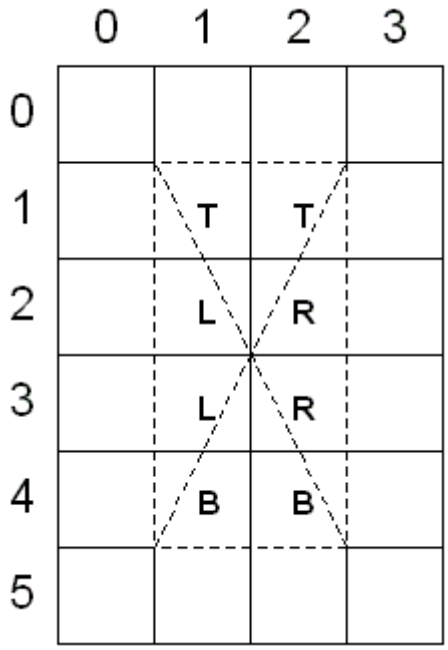
按照自上而下-从左到右的规则，顶部以水平刻度的竖向位置为参照，左部以垂直刻度的横向位置为参照。只有水平线才能成为顶部边界线，通常大多数三角形仅有左右边界。



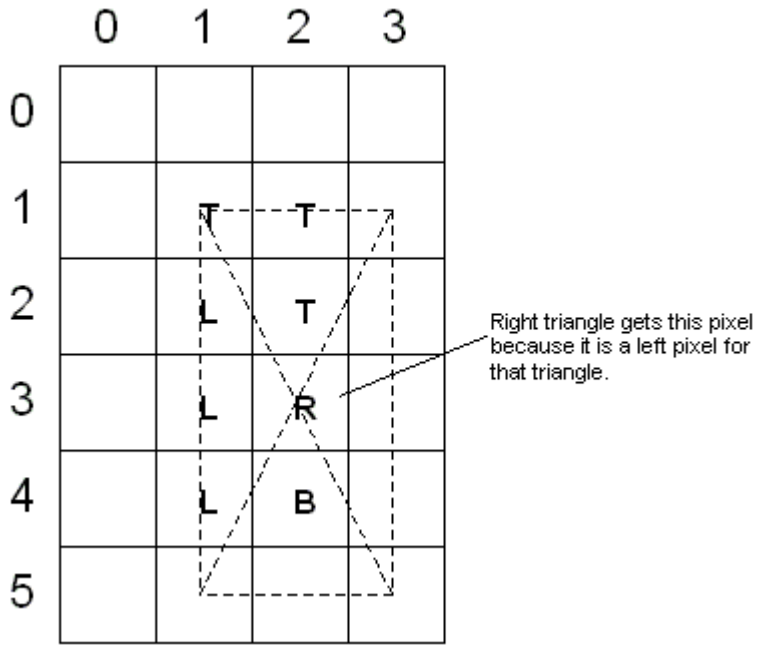
当三角形穿越像素中心时，由自上而下-从左到右的规则决定 Direct3D 的动作。下图有 2 个三角形，一个坐标为(0，0)，(5，0)，(5，5)，另一个坐标为(0，5)，(0，0)，(5，5)。因为共享边是第一个三角形的左边，所以第一个三角形有 15 个像素（黑色表示），反之第二个只有 10 个像素（灰色表示）。



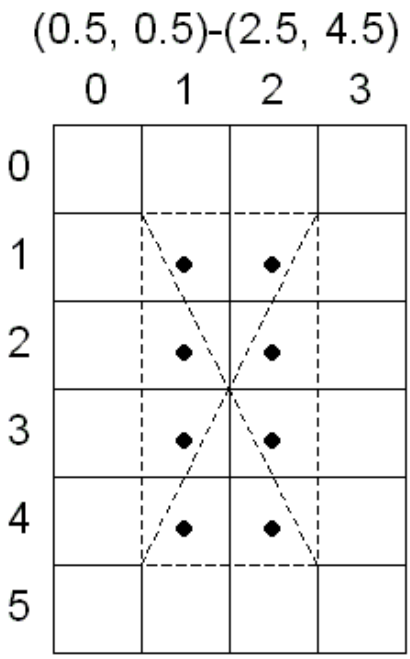
如果你定义一个左上角坐标为（0.5，0.5）右下角坐标为（2.5，4.5）的矩形，矩形的中心点是（1.5，2.5）。当 Direct3D 光栅和矩形镶嵌时，四个三角形中每个像素的中心都显然位于每个三角形内，自上而下-从左到右的规则是不需要的。下图描绘了这种情形，矩形中的像素根据 Direct3D 包含的三角形不同，被分别标记。

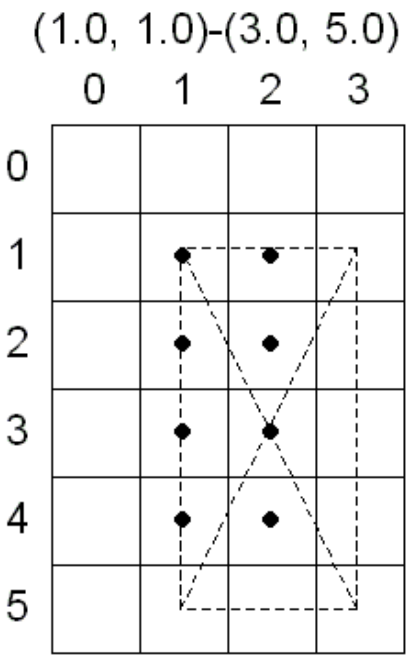


如果你移动前例中的矩形，使它左上角坐标位于(1.0，1.0)右下角坐标位于(3.0，5.0)中心点位于(2.0，3.0)，Direct3D 将应用自上而下-从左到右的规则。如下图所示，矩形中的大部分像素跨越了 2 个或多个三角形的边界范围。



对下面二个矩形而言，相同的像素受到了影响。





4.2 点和线规则

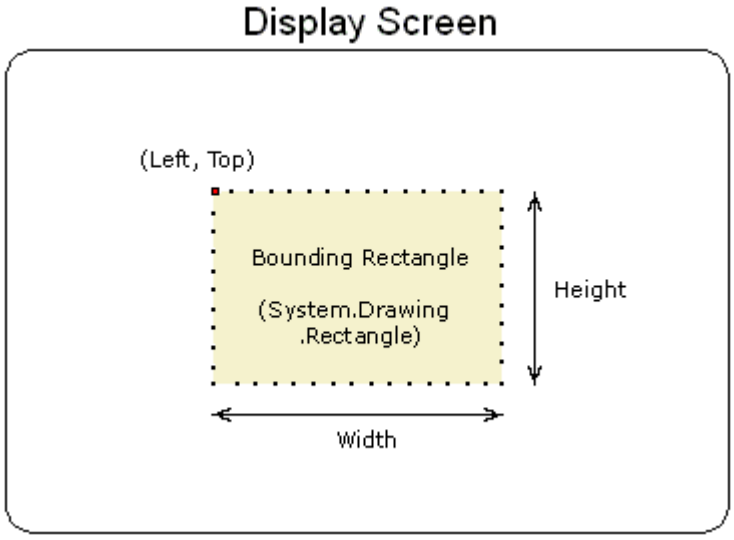
点和点精灵以相同方式渲染，点精灵作为屏幕上排列的四角被渲染，它遵循和多边形渲染相同的规则。
非抗锯齿 线条在 Direct3D 中的渲染规则和在 GDI 中完全相同。
了解抗锯齿线条渲染，参看 Line。

4.3 点精灵规则

和造型初次被三角形镶嵌并导致三角形光栅化类似，点精灵和面片造型被光栅化。

第5节　矩形

包含在有界矩形内的屏幕对象，贯穿在Direct3D和Windows编程中。有界矩形的边总是平行于屏幕，托管代码程序利用 System.Drawing命名空间下的 Rectangle结构体，存储矩形信息、位图传送或碰撞检测。
多数非托管C++程序使用 RECT结构体，它需要矩形左上角(x，y)和右下角(x，y)坐标。托管代码程序提供 Rectangle属性，指明矩形左上角(x，y)坐标和它的高度、宽度，如下面的图表所示。



出于效率、一致性和易用性的考虑，所有 Direct3D 表达显示函数都使用了矩形。

第6节　三角形内插值

在渲染期间，流水线为每个三角形插入顶点数据。可以插入的数据类型有：

漫反射色彩
镜面反射色彩
漫反射 alpha （三角不透明度）
镜面反射 alpha
雾化因子（来自固定的 镜面反射透明 函数顶点管道和来自可编程顶点管道的雾化注册）
纹理坐标

顶点数据插值取决于下表所示的着色模式。

着色模式	描述
Flat	仅雾化因子在该模式下插值。三角形第一个顶点的色彩被应用在整个表面。
Gouraud	在所有 3 个顶点之间进行线性插值。

色彩和镜面插值根据色彩模式区别对待处理。在 RGB 模式下，系统在插值时使用红绿蓝色彩组分。
因为设备驱动可以两种不同方式（纹理混合或点刻法）进行透明处理，所以色彩的不透明度组分作为一个独立的内插值。
使用 Caps结构体的 ShadeCaps属性决定当前设备驱动支持何种方式插值。

第7节　向量、顶点和四元数

在整个 Direct3D 中，顶点用于描述位置和方向。造型的每个顶点通过一个向量给出它的位置、色彩和纹理坐标，通过法线给出它的方向。
四元数给定义了[x，y，z]三组分的向量加入第四组分，这是 3-D 旋转常用矩阵方法的一种选择。四元数表达了 3-D 空间中的一个轴和围绕该轴的旋转。如一个四元数可能代表了(1，

1，2)轴线，并围饶轴线旋转 1 度。四元数携带了有价值的信息，但其真正能力在于你可以对其进行 2 种操作：合成与插值。
四元数的合成类似于它们的组合。2 个四元数合成用符号表示如下：

$$Q = q_1 \circ q_2$$

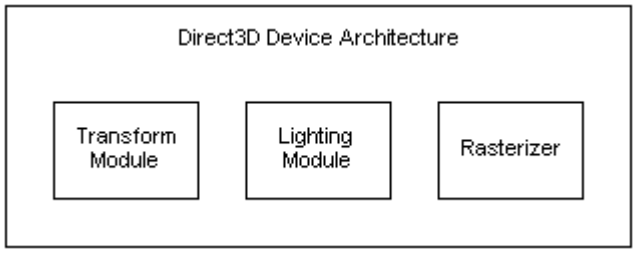
2 个四元数的合成在几何学上意义是"按照 rotation 2 这步将几何体围绕 axis2 轴旋转，然后按照 rotation 1 这步，旋转几何体围绕 axis1 轴"。这里，Q 表示了一个绕单轴的旋转，其结果与对几何体进行 q_2 旋转，然后进行 q_1 旋转是一样的。
程序使用四元数插值，计算从一个轴和方向到另一个轴和方向过渡的合理平滑的路径。因此在 q_1 和 q_2 间插值为实现方向的转换提供了简单的方法
当同时使用合成和插值时，它们为你提供了操纵看似复杂的几何体的简单方法。例如，假设你想把一个几何体旋转到一个给定方向。你想绕 axis2 轴旋转 r2 度，然后绕 axis1 轴旋转 r1 度，但你不知道最后的四元数。通过合成，你就能在几何体上组合 2 个旋转操作得到一个单独的四元数作为结果。然后你对合成后的四元数插值就得到从一个方向到另一个方向的平滑过渡。
Microsoft.DirectX 命名空间包含的 Quaternion 结构，让你可以使用四元数。如 RotateAxis 方法对一个给定转轴的向量赋予旋转值，并更新原四元数的属性和域。Multiply 方法对四元数进行合成操作，Slerp 方法在 2 个四元数之间进行球形线性插值。
Direct3D 应用程序可以使用 Microsoft.DirectX.Vector2、Vector3 和 Vector4 结构体的方法，来分别简化 2 组、3 组和 4 组分向量的操作。

第2章　设备

Direct3D 设备是 Direct3D 的渲染组件。它封装存储了渲染状态。另外 Direct3D 设备执行变换和光照操作，将图像光栅化成为表面。下列主题提供了设备使用的细节。

设备类型
建立设备
选择设备
失效设备
测定硬件支持
处理顶点数据
设备支持的造型种类

Direct3D 设备在架构上包含 1 个变换模块，1 个光照模块，1 个光栅化模块，如下图所示。



DirectX 当前支持 2 种 Direct3D 设备类型：一种是硬件设备，具有硬件加速光栅和软硬件顶点处理着色；另一种是引用设备。
你可以将这些设备看成 2 种独立的驱动程序。软件驱动程序代表软件和引用设备，硬件驱动程序代表硬件设备。通常做法是使用硬件设备装运程序，引用设备测试特性。这些第三方提供的用于模拟特殊设备，如开发中尚未发布的硬件。

程序创建的 Direct3D 设备必须符合运行程序的硬件所具备的功能。Direct3D 通过访问安装在计算机上的三维硬件或软件模拟三维硬件功能提供渲染能力。因此 Direct3D 为设备提供了硬件访问和软件模拟二种方式。
硬件加速设备比软件设备具有更好的性能。硬件设备类型在所有支持 Direct3D 的图形适配器上都是可用的。大多数程序运行的目标计算机具备硬件加速，而依靠软件模拟适应低端计算机。
除引用设备外，软件设备并不一定能够支持和硬件设备完全相同的特性。程序始终应该查询设备功能以决定支持何种特性。
因为由 DirectX9.0 提供的软件和引用设备的行为和硬件设备是一样的，所以使用硬件设备工作的程序代码无需改动就可以工作在软件或引用设备上。注意二者行为虽然相同，但是设备功能有差别，特定的软件设备也许只能完成一小部分功能。
Direct3D 允许你指定设备的行为和类型。Device(Int32,DeviceType,IntPtr,CreateFlags,PresentParameters[])方法可以将 1 个或多个 CreateFlags行为标识组合起来控制Direct3D设备的整体行为。这些行为指出了那些是或不是Direct3D运行时的部分，设备类型指定了使用何种驱动程序。虽然某些设备行为的组合非法，但在所有的设备类型上使用全部的设备行为是可能的。如指定 DeviceType.Software（指明了一个软件设备）和设备建立标识 CreateFlags.PureDevice（指定设备不支持顶点处理）的组合设置是合法的。

第1节　设备类型

1.1　硬件设备

主设备类型是硬件设备，它支持硬件加速光栅和软硬件二种顶点处理。如果你运行程序的计算机配置了支持 Direct3D 的图形适配器，程序就可以用它来进行三维操作。Direct3D 硬件设备将通过硬件的全部或部分变换、光照和光栅化模块来执行。
程序不直接访问 3D 加速卡。它们调用 Direct3D 的函数和方法。Direct3D 通过一个硬件抽象层访问硬件。如果运行程序的计算机支持这个抽象层，那么通过使用硬件设备将获得最高性能。
要指定硬件光栅和着色，应使用 DeviceType.Hardware常量创建一个 Device对象。
注意：硬件设备无法渲染 8 位渲染目标表面。

1.2 引用设备

Direct3D 支持另一种叫做引用设备或引用光栅的设备类型。和软件设备不同，引用光栅支持所有 Direct3D 特性。因为这些特性可以通过软件准确完成，只是速度不快。引用光栅尽其所能使用了特殊的 CPU 指令，但并不打算供零售版程序使用。引用光栅仅用于特性测试和示范目的。
要建立一个引用设备，应使用 DeviceType.Reference常量创建一个 Device对象。

第2节　建立 1 个设备

注意：由一个给定 Direct3D 对象建立的所有渲染设备共享相同的物理资源。虽然你的程序可以从一个单独的 Direct3D 对象建立多个渲染设备，但它们共享相同的硬件，这会极大的影响性能。

要创建一个Direct3D设备的话，你的托管代码程序应首先建立一个 PresentParameters 对象，它包含了有关该设备的描述参数。下面的C#代码范例包含了一个使用 Device(Int32,DeviceType,Control,CreateFlags,PresentParameters[])构造器的简单初始化过程。这个构造函数接收了一个 PresentParameters对象。

```
        [C#]



using Microsoft.DirectX.Direct3D;

.

.

.

// Global variables for this project

Device device = null;                  // Create rendering device

PresentParameters presentParams = new PresentParameters();



device = new Device(0, DeviceType.Hardware, this,

                    CreateFlags.SoftwareVertexProcessing, presentParams);
```

这条函数调用指明了缺省适配器，一个硬件设备和软件顶点处理方式。

注意对设备创建、释放和重置的调用仅应发生在和焦点窗口的窗口处理程序相同线程上。

相关主题

指南 1：建立 1 个设备

第3节　选择 1 个设备

程序可以查询硬件检测支持 Direct3D 的设备类型。这部分信息包含了与列举显示适配器和选择 Direct3D 设备有关的主要任务。

一个程序必须执行一系列任务来选择一个合适的 Direct3D 设备。注意下面步骤适用于一个全屏程序。大多数情况下，窗口程序可以跳过大部分步骤。

1. 开始，程序必须将系统内的显示适配器列举出来。一个适配器就是一个硬件物理块。注意图形卡也许包含了不止一个适配器，比如双头显示的情形。参看(SDK root)\Samples\Managed\Common\dxmutenum.cs 获取一个列举适配器的

详细范例。不支持多显示器连接的程序可以忽略这步，使用缺省适配器的列举。

2. 程序用 _Manager对象的 Adapters属性列举出每个适配器支持的显示模式。可以通过 AdapterListCollection对象查询适配器集合。
3. 程序在必要时检查 _Manager.CheckDeviceType的checkType 参数返回值，为每个列举的显示模式检查是否具有硬件加速。CheckDeviceType的其他参数可以为设备当前显示和后台缓存格式提供额外信息。
4. 程序使用 _Device.DeviceCaps属性来检查适配器上的设备能够达到的功能级别。该属性返回一个描述设备功能的 Caps结构。那些无法支持所需功能的设备将不包含一个合法结构。在 CheckDeviceType验证的所有显示模式下，由 DeviceCaps返回的设备功能保证在该设备上是不变的。
5. 设备总是渲染它支持的列举显示模式的格式表面。如要渲染不同格式的表面，程序应调用 _CheckDeviceType。如果设备能够渲染该格式，那它就能保证所有通过 DeviceCaps返回的功能可用。
6. 最后，程序使用 CheckDeviceMultiSampleType方法可以确定渲染格式是否支持诸如全屏抗锯齿等多重采样技术。

完成上述步骤后，程序应该有了一个可以操纵的显示模式列表。最后步骤是验证有否足够的设备可用内存数量以满足缓存和抗锯齿的需要。如果没有验证过程将无法预测模式和多重采样的内存消耗，因此这个测试是必须的。而且，某些显示适配器在架构上也许没有固定数量的设备可用内存。这意味着程序应能够在进入全屏模式时，报告显存溢出错误。通常，程序应将全屏模式从提供给用户的模式列表中除去，或减少后台缓存数量、使用简化的多重采样技术以降低内存消耗。

窗口程序执行如下所示的类似任务。

1. 它测定被窗口客户区覆盖的桌面矩形。
2. 它列举适配器，寻找显示器上覆盖的客户区所属适配器。如果客户区为多个适配器共属，程序会选择独立驱动每个适配器，或驱动一个单独的适配器并让Direct3D将像素从一个设备传输到另一个设备。程序也可以忽略以上 2 步，使用缺省适配器，虽然当窗口置于辅显示器时会导致操作速度降低。缺省适配器是由 DeviceCreationParameters.AdapterOrdinal识别的第一个序号。适配器序号值可以通过 GetDeviceCaps方法查询或由 AdapterInformation.Adapter属性返回。

在桌面模式下，程序应调用 _CheckDeviceType来确定设备是否支持渲染指定格式的后台缓存。AdapterInformation.CurrentDisplayMode属性可以用于测定桌面显示格式。

第4节　丢失设备

Direct3D设备处于运作或丢失二种状态之一。运作状态是设备的正常状态，设备可以按照预期运行和显示所有渲染。在诸如全屏程序键盘焦点丢失的事件中，设备会转换成丢失状态，导致无法进行渲染。丢失状态表现为所有渲染操作无记录的失败，这意味着渲染方法即使操作失败也返回成功代码。这时 Device.DeviceLost事件将被触发。

设计上，没有指出可能导致设备丢失的所有情形。典型例子是当用户按下ALT+TAB或系统对话框初始化时焦点丢失。设备丢失应归于动力管理事件或另一个程序采取了全屏操作。另外，Device.Reset的任何失败也将导致设备丢失。

下列 2 个异常对象也许会遇到：

DeviceLostException这个对象指明设备已经丢失，不能重置，因此无法进行渲染。

DeviceNotResetException这个对象指明设备已经丢失，但现在可以重置。

4.1 响应丢失设备

失效设备在重置后必须重建资源（包含显存资源）。如果设备失效，程序应查询设备看是否能恢复到运作状态。如果不能，程序应等待直到设备能够恢复。
如果设备能够恢复，程序为设备准备销毁所有的显存资源和交换链表。然后程序调用 Reset 方法。Reset是当设备丢失时唯一有效的方法，也是程序将丢失设备转换到运作状态的唯一可用方法。如果程序没有释放分配在 Default中的所有资源（包含由 CreateRenderTarget 和 CreateDepthStencilSurface方法建立的资源），该方法将会失败。
在极大程度上，Direct3D的高频率调用不会返回任何关于设备是否丢失的信息。程序可以继续调用渲染方法，如 Device.DrawPrimitives，而不会收到丢失设备的通知。实际上，这些操作在设备重置为运作状态前都被丢弃了。
程序可以通过查询 CheckCooperativeLevel方法的返回值确定设备是否丢失。

4.2 锁定操作

当设备丢失后，Direct3D 内部作了大量工作以保证锁定操作成功。然而在锁定期间不能保证显存资源的数据正确。它仅保证无错代码返回，这使得书写程序时不用关心锁定操作期间的设备丢失。

4.3 资源

资源会消耗显存。因为丢失设备与适配器所属显存分离，所以当设备丢失时不能保证显存的分配。结果是所有创建资源的方法仅能分配虚拟系统内存。因为在设备重新调整大小前任何显存资源必须被销毁，所以不会产生显存分配溢出的问题。这些虚拟表面可以执行锁定和复制操作，看上去运行正常，但直到程序调用了 Device.Present才发现设备已经丢失。
全部显存必须在设备从丢失状态重置为运作状态前被释放。这意味着程序应释放 SwapChain建立的交换链表和放在 Pool.Default 内存中的任何资源。程序无需释放在 Managed或 SystemMemory内存中的资源。其他状态数据在向运作状态的转换过程中自动销毁。
鼓励你开发具有单一代码路径以响应设备丢失的程序。如果该代码路径不同，那么可能和启动时设备初始化的代码路径类似。

4.4 获得返回数据

Direct3D允许程序使用 ValidateDevice验证纹理和硬件是否违反单通道渲染的状态。这种方法通常在程序初始化时调用，如果设备丢失，将返回 DeviceLostException对象。
Direct3D 还允许程序从显存资源向非可变系统内存资源复制生成的或预先绘制的图像。因为这种传输方式的源图像可能会随时丢失，所以 Direct3D 允许在设备丢失时进行这种复制操作。
当源对象位于可变内存(Pool.Default)和目标对象位于非可变内存（SystemMemory或 Managed）时，使用 UpdateSurface和 GetFrontBufferData的复制操作将返回 1 个

DeviceLostException对象。一个 GetFrontBufferData的调用失败应归于从主表面找回的数据。除 Present，TestCooperativeLevel 和 Reset 方法以外，这是导致 DeviceLostException返回的仅有情形。

4.5 可编程着色器

在DirectX9.0 中，当 Device.Reset调用后，Vertex Shader 1_1 和 Pixel Shader 1_X 无需重建，它们会被记忆。先前版本的DirectX中，当设备丢失时，着色器必须被重建。

第5节　检测硬件支持

Direct3D 提供下列方法测定硬件支持。

CheckDeviceFormat

用于验证一个表面格式能否用作纹理，一个表面格式能否用作纹理和渲染目标，或一个表面格式能否用作模版缓存深度。另外这个方法用于验证支持何种缓存深度格式和对模版缓存格式的支持。

CheckDeviceType

用于验证设备执行硬件加速的能力，设备建立一个显示用交换链表的能力，或设备渲染当前显示格式的能力。

CheckDepthStencilMatch

用于验证一个模版缓存深度格式是否与目标渲染格式兼容。注意在该方法调用前，程序应在模版深度和渲染目标格式上都调用 CheckDeviceFormat 。

第6节　处理顶点数据

Device类支持软硬件 2 种顶点处理方式。通常，软硬件设备的顶点处理功能是不同的。硬件功能随显示适配器和驱动程序有所不同，而软件功能是固定的。

下列 CreateFlags枚举的常量控制硬件和引用设备的顶点处理行为。

软件顶点处理

硬件顶点处理

混合顶点处理

BehaviorFlags结构也可使用访问相同常量值的属性来控制行为。

当调用 Device构造器时应指定这些标记中的一个。MixedVertexProcessing混合模式标记允许设备可以同时执行软硬件顶点处理。同一时间仅能为一个设备设置 3 个顶点处理标记中的 1 个；它们之间是互斥的。注意当建立了一个纯设备(PureDevice)时，必须将标识设置为 HardwareVertexProcessing。

为了消除在一个单独设备上的两种顶点处理功能，在运行时程序仅查询硬件顶点处理功能，而不会查询固定的软件顶点处理功能。

你可以参考 VertexProcessingCaps结构的属性测定设备的硬件顶点处理功能。软件顶点处理支持下列 VertexProcessingCaps功能（这是 2 种顶点处理都支持的功能）。

SupportsDirectionalLights

SupportsLocalViewer

SupportsMaterialSource

SupportsPositionalLights
SupportsTextureGeneration
SupportsTweening

另外下表列出了在软件顶点处理模式下，可以为设备功能(Caps)设置的值。

成员	软件顶点处理功能
MaxActiveLights	Unlimited
MaxUserClipPlanes	6
MaxVertexBlendMatrices	4
MaxStreams	16
MaxVertexIndex	int(0xffffffff)

软件顶点处理提供了一套可以保用的顶点处理功能，包含一个大量光照和完整支持可编程顶点的着色器。在使用硬件设备时，你可以随时将软硬件顶点处理方式组合。这是仅有的同时支持软硬件顶点处理的设备类型，而这只需要使用 Pool枚举的 SystemMemory常量在系统内存中分配用于软件顶点处理的顶点缓存。

注意：硬件顶点处理的性能和软件方式差不多。因此在一个单独的设备类型内，同时提供硬件和软件模拟顶点处理功能是一个好主意。而主处理器比专用图形硬件速度慢很多，所以这种情形不适用于光栅化过程，因此也就没有为单一设备类型同时提供硬件和软件模拟光栅化。在单一设备上，软件顶点处理仅是由运行时和硬件（driver 驱动程序）共同提供的功能镜像的实例。所有其他的设备功能则描述了由驱动程序提供的潜在的变化的功能。

第7节　设备支持的造型类型

Direct3D 设备可以建立并操纵以下类型的造型。

点列表
线列表
线条带
三角形列表
三角形条带
三角形扇

你可以使用任一种 Device渲染方法对托管代码程序中造型的类型进行渲染。

7.1 点列表

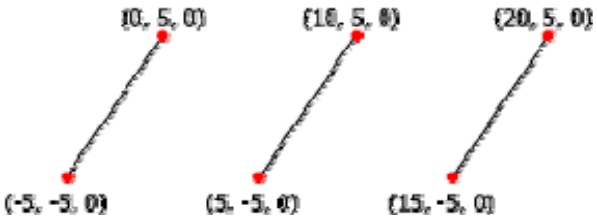
点列表是一个每点独立进行渲染的顶点集合。你的程序可以在 3-D 场景中使用它作为星空视野，或作为多边形表面的点化线。

下图是一个经过渲染的点列表。

程序可以将材质和纹理应用于点列表。材质和纹理的色彩看上去紧沿点边缘，却不在任一点上。

7.2 线列表

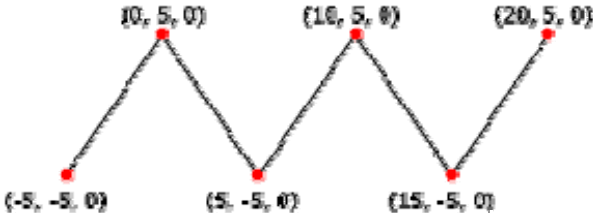
线列表是一系列的独立线段。它用于为 3-D 场景加入冰雹或大雨诸如此类的任务。程序通过填充一个顶点数组来建立一个线列表，线列表中顶点的数量必须是大于等于 2 的偶数。
下图是一个经过渲染的线列表。



你可以将材质和纹理应用于线列表。材质和纹理的色彩看上去紧沿线条，却不在线条的任一点上。

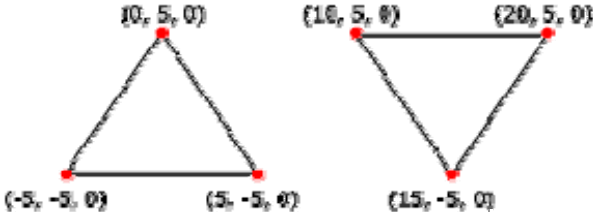
7.3 线条带

线条带是一个由相互连接的线段组合成的造型。程序可以用它产生不封闭的多边形。封闭多边形是指其最后一个顶点通过线段与第一个顶点连接。如果你的程序使用的多边形是基于线条带的，顶点就无法保证共面。
下图是一个经过渲染的线条带。



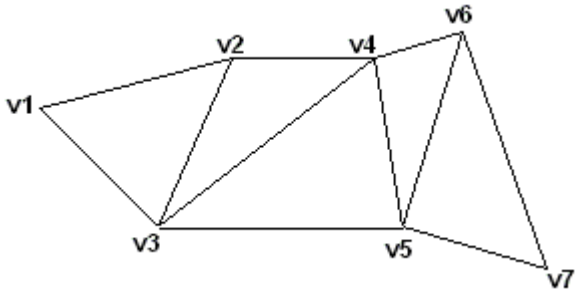
7.4 三角形列表

三角形列表是一系列独立的三角形。它们之间可能靠的很近，也可能不是。一个三角形列表必须至少具备三个顶点。顶点总数必须能被 3 整除。
使用三角形列表建立一个由离散面片组成的物体。如在三维游戏中建立力场墙的一种方法就是指定一个由互相不相连的小三角形组成的大型列表，然后将一个可以发散光线的材质和纹理应用到这个三角形列表。
三角形列表对于建立具有清晰边界并使用高洛德着色的造型有用。参看 面和顶点法向量。
下图是一个经过渲染的三角形列表。



7.5 三角形条带

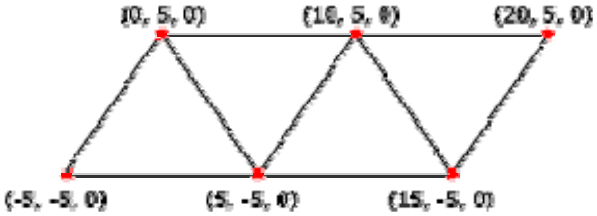
三角形条带是一连串相连接的三角形。因为三角形是相连的，程序不必为每个三角形重复指定 3 个顶点。如定义下图中的三角形条带，仅需 7 个顶点。



系统使用顶点 v1，v2，v3 画第一个三角形，v2，v4，v3 画第二个三角形，v3，v4，v5 画第三个，v4，v5，v6 画第四个，以此类推。注意第二个和第四个三角形的顶点顺序是不正确的，这就需要确认所有三角形是按照顺时针序画的。

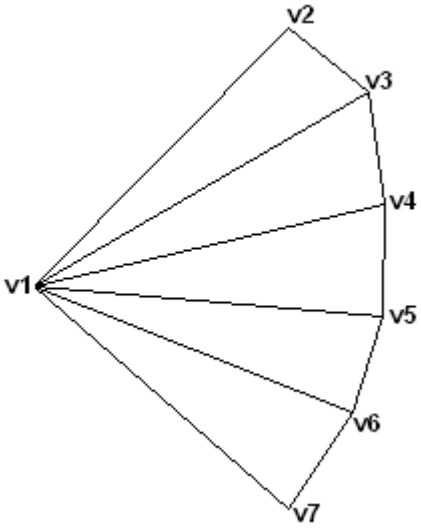
三维场景中的大多数物体由三角形条带组成。这是因为使用三角形条带表达复杂物体在内存占用和处理时间开销上效率更高。

下图是一个经过渲染的三角形条带。



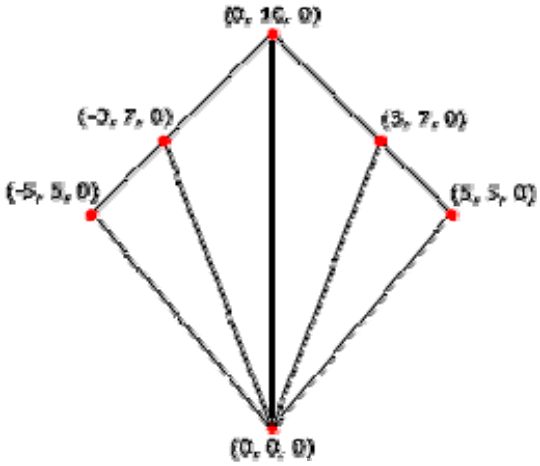
7.6 三角形扇

三角形扇和三角形条带类似，但其所有三角形共享一个顶点。如下图所示。



系统使用顶点 v2，v3，v1 画第一个三角形，v3，v4，v1 画第二个三角形，v4，v5，v1

画第三个三角形，以此类推。当平面着色激活时，系统使用它的第一个顶点色彩为三角形着色。
下图是一个经过渲染的三角形扇。



第3章　资源

资源是常用于渲染场景的纹理和缓存。程序需要建立，加载，复制和使用资源。这部分主要介绍了资源和通过程序使用资源的方法和步骤。
包括几何体资源 IndexBuffer和 VertexBuffer在内的所有资源，都是由 Resource类继承而来。纹理资源 Texture，CubeTexture和 Volume也是从 BaseTexture类继承而来。
附加信息分为以下主题：

资源属性
操纵资源
锁定资源
管理资源
应用程序托管资源和分配策略

第1节　资源属性

所有资源共享下列属性，这些属性通过同名或类似名的枚举指定。

属性	枚举值	描述
Usage	Usage	资源被使用的方式，例如，作为一个纹理或者渲染目标。
Format	Format	数据的格式，例如一个 2-D 表面的像素格式。
Pool	Pool	资源被分配所在内存的种类。
Type	ResourceType	资源的种类，例如，一个顶点 buffer 缓冲器或者渲染目标。

资源利用是强迫的。在某个固定操作中将要使用资源的程序必须在资源建立的时候指定操作。
Usage枚举的 RTPatches，NPatches和 Points常量为驱动程序指明了这些缓存中的数据可能分别用于三角形或栅格patch，N-patches或点纹理。这些标识符被提供用于万一没有

主机处理而硬件不能执行这些操作的情况。因此驱动程序通常应该将这些表面分配在系统内存中，使得CPU可以访问它们。如果驱动程序可以在硬件上完整执行这些操作，就可以把这些表面分配到显存或加速图形端口（AGP）内存中，避免一份主机拷贝并至少获得双倍的性能提升。注意由这些标识符提供的信息并非绝对需要。驱动程序可以检测正在数据上执行的操作，并将后续帧的缓存移回系统内存。
包含在一个单独资源中的不同对象不能混合在一个池中。就是说当选定一个资源池后，mipmap 中的 mip 级别和池都不能改变。
当程序调用诸如 CubeTexture之类的资源构造函数时，资源类型在运行时将被隐式设置。程序可以在运行时查询这些类型；然而我们所期望的是大多数情形下都不会要求这种运行时检查。

第2节　操纵资源

为了渲染场景，你的程序是这样操纵资源的：首先程序通过初始化 CubeTexture，Texture或VolumeTexture对象来建立纹理资源。注意这些纹理类都由BaseTexture基类继承而来。返回的纹理对象是表面或体积的容器；这些容器一般被认为是缓存。属于资源的缓存继承了资源的用法、格式和池，但具有它们自己的类型。更多信息参看 Resource Properties。通过调用 CubeTexture.LockRectangle，，Texture.LockRectangle，或VolumeTexture.LockBox方法，程序获得了对包含用于装载艺术品的表面的访问。详见Locking Resources。锁定方法带有指示所包含表面的参数，并返回指向像素的指针。这种参数的范例就是mipmap的子级别或纹理的立体表面。典型程序从不直接使用 1 个表面对象。
另外当程序初始化 IndexBuffer或 VertexBuffer对象时，它就建立了面向几何体的资源。你的程序通过调用 IndexBuffer.Lock或 VertexBuffer.Lock方法来锁定并填充缓存资源。如果你的程序允许 Direct3D 运行时管理这些资源，那么资源创建进程就在此终止。否则，程序将对系统内存资源的管理提升为设备可访问资源，而硬件加速卡可以通过调用 Update.Texture 方法来使用它们。
为了显示从资源中渲染得来的图像，程序还需要色彩和深度模版缓存。对于典型程序来说，色彩缓存属于设备的交换链表，该链表是 1 个由设备隐式建立的后台缓存表面集合。深度模版缓存可以被隐式建立，或通过 CreateDepthStencilSurface方法显式建立。程序将设备和它的深度、色彩缓存与 1 个 SetRenderTarget调用相关联起来。

第3节　锁定资源

锁定 1 个资源意味着批准CPU对它存储区的访问。锁定步骤通过 LockFlags枚举值常量定义。然而这些锁定标记仅是一种提示，运行时并不检查程序遵循由这些标记所指定的功能。一个指定 LockFlags.ReadOnly的应用程序，但是对资源进行了写入，将会得到不确定的结果。通常违背锁定标记（包含锁定用法标记）的工作，将在以后的版本中无法保证运作，并也许会导致明显的性能损失。
解除锁定操作紧接着锁定操作之后。比如锁定 1 个纹理之后，通过解除锁定，程序随后放弃了对锁定纹理的直接访问。除了批准处理器的访问之外，其他任何关于资源的操作在锁定期间都是可以连续执行的。允许仅对资源进行 1 个单独的锁定，即使对不重叠区域。当锁定操作正作用于表面时，该表面上没有加速卡支持的操作都可以执行。

每个资源类都有被包含缓存的锁定方法。每个纹理资源也可以锁定资源的一部分。二维资源（表面）允许对子矩形锁定，体积资源允许对子体积或盒锁定。每个锁定方法返回 1 个 System.Array。该数组包含的值表示了数据行或位面之间的距离，这取决于资源类型。返回数组总是开始于被锁定子区域的左上角字节。
当使用索引和顶点缓存时，你可以安排多个锁定调用；然而你必须确保锁定调用语句的数量和解除锁定语句的数量相匹配。
压缩纹理格式的 DirectX 变换集，按照 4x4 块的编码方式存储像素，仅能被锁定在 4x4 的边界处。

第4节　管理资源

资源从系统内存存储区提升为设备可访问存储区，然后从设备可访问存储区丢弃的过程就是资源管理。Direct3D运行时有基于近期最少使用优先技术的自身管理算法。当Direct3D检测到有比共存于设备可访问内存中更多的资源用于 1 个独立帧的时候，它就在 Device.BeginScene和 EndScene调用之间，切换至近期最常使用优先技术。
在建立时期使用 Pool枚举值中的 Managed常量来指定 1 个托管资源。托管资源在设备的丢失和运作状态之间持续转换。设备可以通过调用 Reset重置，资源继续正常运行而不会被重新加载。然而如果设备必须被销毁并重新建立，所有使用 Managed建立的资源则必须被重新建立。
在建立时候使用 Default标记指定资源将要被放置在缺省的池中。缺省池中的资源在设备丢失和运作状态之间不会持续转换。这些资源必须在调用 Reset之前被释放然后必须被重建。更多关于丢弃设备状态的信息，参看 Lost Devices 。
注意资源管理不支持所有类型和用法。比如通过 RenderTarget标记建立的对象就不被支持。另外，不推荐对那些内容高频率变化的对象进行资源管理。比如改变每帧的托管顶点缓存会在某些硬件上显著降低性能。然而对于纹理资源来说这不是一个问题。

第5节　应用程序托管资源和分配策略

动态顶点缓存用于渲染动态几何体，由二叉空间分割（BSP）树得来的数据，或其他可见的数据结构。这种类型的渲染可以通过 Usage枚举的 Dynamic常量这样所期望格式的预分配缓存完成。这些资源通过程序中的资源管理器分配以支持程序的需要。
因为程序同时使用的顶点仅有很小的间距差异，而对于每个唯一的间距都需要一个不同的顶点缓存，所以动态顶点缓存的总数很小。当以这种方式管理动态资源时，确保对资源的高频率需求而不显著降低程序性能就很重要了。
托管的顶点缓存或索引缓存资源不能通过在创建时设置 Dynamic进行动态声明。这会要求对每个顶点缓存内容的改变进行 1 个额外的复制。
通过Direct3D和程序使用托管资源时，在 Pool.Default内存中的程序托管资源分配优先于建立Direct3D托管的资源。这使内存管理器可以保持一个精确数量的可用内存。

第4章　变换

变换常用于把几何体对象从 1 个空间坐标转化成另 1 个空间坐标。最常用的变换是使用矩

阵。矩阵本质上是一种容纳变换值并将它们应用到数据的工具。下列主题介绍了矩阵并解释如何使用它们产生世界，视点和投影变换。

视点变换
世界变换
矩阵
投影变换

第1节　视点变换

这部分介绍了视点变换的基本概念并提供了如何在Direct3D程序中设立1个视点变换矩阵的详情。信息由以下部分组成。

什么是1个视点变换
建立1个视点矩阵

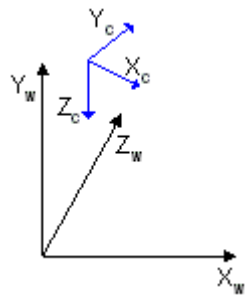
1.1 什么是视点变换？

视点变换确定观察者在世界空间中的位置，将顶点转换成照相机空间。在照相机空间中，照相机或观察者位于源点，朝着z轴正方向看去。回忆一下Direct3D使用了左手坐标系统，因此z正轴指向屏幕内。视点矩阵围绕照相机的位置--照相机空间的源头和方向--重新确定了物体在世界中的位置。

有很多种建立视点矩阵的方法。总的来说，照相机在世界空间中具有某个逻辑位置和方向，它们用作创建视点矩阵的起始点，该视点矩阵会在场景中应用到模型上。视点矩阵转化并旋转物体，把它们放置在以照相机为源点的在照相机空间中。一种建立视点矩阵的方法是将每个轴向的变换矩阵和旋转矩阵组合起来。这种方法要应用到下面的矩阵公式。

$$V = T \cdot R_z \cdot R_y \cdot R_x$$

公式中，V是被建立的视点矩阵，T是重新定位物体在世界中位置的变换矩阵，R_x到R_z是沿x-，y-，z-轴旋转物体的旋转矩阵。变换和旋转矩阵基于照相机在世界空间中的逻辑位置和方向。因此如果照相机在世界中的逻辑位置是<10,20,100>，变换矩阵的目标就是将物体沿x-轴移动-10单位，沿y-轴移动-20单位，沿z-轴移动-100单位。公式中的旋转矩阵基于照相机的方向，它根据照相机空间的轴线旋转偏离了世界空间多少的程度。比如如果早先提及的照相机直指向下方，它的z-轴就偏离世界空间的z-轴90度，如下图所示。



旋转矩阵向场景中的模型应用等值、反向的旋转。该照相机的视点矩阵包含了1个围绕x-轴-90度的旋转。旋转矩阵与变换矩阵组合建立了1个视点矩阵，它可以调整场景中物体的位置和方向。因此物体的上部面向照相机，看上去照相机位于模型的上方。

1.2 建立 1 个视点矩阵

Matrix.LookAtLH 和 Matrix.LookAtRH 帮助函数建立了 1 个基于照相机位置和观察朝向的视点矩阵。
下列 C#代码范例建立了 1 个左手坐标系的视点矩阵。

```
[C#]



Vector3 eye = new Vector3(0,3,-5);

Vector3 at = new Vector3(0,0,0);

Vector3 up = new Vector3(0,1,0);


Matrix viewMatrix = Matrix.LookAtLH(eye,at,up);
```

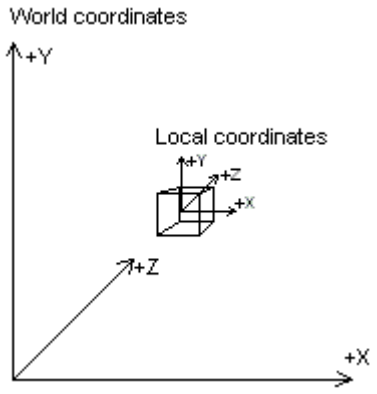
Direct3D 使用了世界和视点矩阵，你要为它们配置一些内部数据结构。每次当你设置 1 个新的世界或视点矩阵时，系统重新计算关联的内部结构。频繁设置这些矩阵是要花费计算开销的。你可以通过将你的世界和视点矩阵连接到 1 个世界-视点矩阵中使得计算量需求最小。世界-视点矩阵就是你先设置世界矩阵，然后将视点矩阵设置为相同。保存单独的世界和视点矩阵隐藏拷贝，使得你可以在需要时改变、连接和重置世界矩阵。但是范例很少使用这种优化。

第2节　世界变换

关于世界变换的讨论介绍了基本概念并提供了如何在Direct3D程序中设立1个世界变换矩阵的详情。

2.1 什么是世界变换？

世界变换将模型世界的坐标改变为世界空间的坐标。模型世界中的顶点是相对于模型的局部位置定义的，世界空间中的顶点是相对于场景中所有物体的公共位置定义的。本质上，世界变换是将 1 个模型放入世界，这就和它的名称一样。下图描绘了世界坐标系统和模型的局部坐标系统之间的关系。



世界变换可以包含任何变换、旋转和缩放的组合。关于变换的数学讨论，参看 3-D变换

2.2 建立 1 个世界矩阵

和任何其他变换一样，你可以将一系列变换矩阵连接成 1 个单独的矩阵来建立世界变换，该矩阵包含了全部矩阵的效果。最简单的情况，当模型位于世界源点，并且它的坐标轴和世界空间的朝向一致，世界矩阵就是单位矩阵。更一般的情况，世界矩阵是 1 个变换为世界空间的组合，并有 1 个或多个旋转按照需要应用到了模型上。
下面的 C#范例，由用 C#书写的 1 个假想 3-D 模型而来，建立了 1 个世界矩阵。它包含了 3 次面向模型的旋转和变换，重新定位了它相对于在世界空间中的位置。

```
            [C#]


public class ModelClass
{
    private float xPos=0;
    private float yPos=0;
    private float zPos=0;

    private float Pitch=0;
    private float Yaw=0;
    private float Roll=0;

    //...Other model properties and methods

    public Matrix MakeWorldMatrix(Matrix worldMatrix)
```

```
    {
        worldMatrix.Translate(xPos,yPos,zPos);

        Matrix matRot = Matrix.Identity;

        matRot.RotateYawPitchRoll(Yaw,Pitch,Roll);

        Matrix.Multiply(matRot, worldMatrix);

        return worldMatrix;

    }
}
```

在你准备世界变换矩阵之后，调用 SetTransform方法来设置它，为第 1 个参数指定 TransformType.World值。你可以将世界变换矩阵间隔传递到 Device.Transform.World 属性。

注意：Direct3D 使用了世界和视点矩阵，你要为它们配置一些内部数据结构。每次当你设置 1 个新的世界或视点矩阵时，系统重新计算关联的内部结构。频繁设置这些矩阵是要花费计算开销的。你可以通过将你的世界和视点矩阵连接到 1 个世界-视点矩阵中使得计算量需求最小。世界-视点矩阵就是你先设置世界矩阵，然后将视点矩阵设置为相同。保存单独的世界和视点矩阵隐藏拷贝，使得你可以在需要时改变、连接和重置世界矩阵。但是范例很少使用这种优化。

第3节　矩阵

Direct3D 使用矩阵执行三维变换。这部分阐述了矩阵如何建立三维变换，描述了一些变换的常见用途，详细说明你如何将矩阵组合产生 1 个包含了多个变换的单独矩阵。信息被分为下列主题。

3-D 变换
平移和缩放
旋转
矩阵串联

3.1 3-D 变换

在使用三维图形的程序中，几何变换常用于以下用途：

表达 1 个物体相对于另 1 个物体的位置。
旋转和调整物体尺寸。

改变观察位置，方向和透视图。

你可以使用 1 个 4 x 4 矩阵将任意点(x,y,z)变换为另 1 个点(x', y', z') 。

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

在(x, y, z)和矩阵上执行下列操作产生点(x', y', z')。

$$x' = (x \times M_{11}) + (y \times M_{21}) + (z \times M_{31}) + (1 \times M_{41})$$
$$y' = (x \times M_{12}) + (y \times M_{22}) + (z \times M_{32}) + (1 \times M_{42})$$
$$z' = (x \times M_{13}) + (y \times M_{23}) + (z \times M_{33}) + (1 \times M_{43})$$

最常用的变换是平移、旋转和缩放。你可以将产生这些效果的矩阵组合为 1 个单独的矩阵，1 次计算多个变换。比如，你可以构建 1 个单独矩阵平移和旋转一系列点。更多信息参看 Matrix Concatenation 。

矩阵按照行-列顺序写入。沿着每个轴均匀缩放顶点（该处理被称为等值缩放）的矩阵，通过下列使用数学符号的矩阵来表达。

$$\begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Direct3D使用 Matrix结构体，将矩阵声明为 1 个二维数组。下列C#代码描述了如何建立 1 个具有等值缩放矩阵特性的 Matrix结构体。

```
[C#]


// In this example, s is a variable of type float.
Matrix scale = new Matrix();

scale.M11=s;     scale.M12=0.0f; scale.M13=0.0f; scale.M14=0.0f;
scale.M21=0.0f; scale.M22=s;     scale.M23=0.0f; scale.M24=0.0f;
scale.M31=0.0f; scale.M32=0.0f; scale.M33=s;     scale.M34=0.0f;
scale.M41=0.0f; scale.M42=0.0f; scale.M43=0.0f; scale.M44=1.0f;
```

3.2 平移和缩放

3.2.1 平移

下列变换将点(x, y, z)平移到新的点(x', y', z')。

$$[x'y'z'1]=[x\ y\ z\ 1]\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ T_x & T_y & T_z & 1\end{bmatrix}$$

你可以在托管代码中，手工建立 1 个平移矩阵。下列 C#代码范例描述了 1 个函数的源代码，它用于建立 1 个可以平移顶点的矩阵。

```
        [C#]



private Matrix TranslateMatrix(float dx, float dy, float dz)
{
   Matrix ret;

   ret = Matrix.Identity;

   ret.M41 = dx;
   ret.M42 = dy;
   ret.M43 = dz;

   return ret;
}
```

为方便起见，托管 Direct3D 提供了 Translation 方法。

3.2.2 缩放

下列变换通过 x-，y-，z-方向的任意值来缩放点(x, y, z)到新的点(x', y', z')。

$$[x'y'z'1] = [x\ y\ z\ 1]\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.3 旋转

这里描述的变换是用于左手坐标系的，可能与你在其他地方看到的变换矩阵有所不同。更多信息参看 3-D 坐标系。
下列变换将点(x, y, z)围绕 x 轴旋转，产生 1 个新的点(x', y', z')。

$$[x'y'z'1] = [x\ y\ z\ 1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

下列变换将点围绕 y 轴旋转。

$$[x'y'z'1] = [x\ y\ z\ 1]\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

下列变换将点围绕 z 轴旋转。

$$[x'y'z'1] = [x\ y\ z\ 1]\begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

在这些范例矩阵中，希腊字母 theta 代表用弧度表示的旋转角度。当沿旋转轴向原始位置观察时，角度是顺时针测量的。
在托管程序中，使用 Matrix.RotationX, Matrix.RotationY, 和 Matrix.RotationZ方法来建立旋转矩阵。下列C#代码范例示范了Matrix.RotationX方法是如何执行 1 个旋转的。

```
    [C#]


private Matrix MatrixRotationX(float angle)
{
   double sin, cos;
   sin = Math.Sin(angle);
   cos = Math.Cos(angle);
```

```
    Matrix ret;
    ret.M11 = 1.0f; ret.M12 = 0.0f; ret.M13 = 0.0f; ret.M14 = 0.0f;
    ret.M21 = 0.0f; ret.M22 = (float)cos; ret.M23 = (float)sin; ret.M24 = 0.0f;
    ret.M31 = 0.0f; ret.M32 = (float)-sin; ret.M33 = (float)cos; ret.M34 = 0.0f;
    ret.M41 = 0.0f; ret.M42 = 0.0f; ret.M43 = 0.0f; ret.M44 = 1.0f;
    return ret;
}
```

3.4 矩阵串联

使用矩阵的 1 个优点就是你可以通过矩阵相乘将 2 个或多个矩阵组合起来。这意味着旋转 1 个模型然后将它平移到某个位置，你不需要应用 2 个矩阵。你可以将旋转和平移矩阵相乘来代替产生 1 个包含了所有效果的合成矩阵。这个过程称为矩阵串联，可以用下列公式书写。

$$C = M_1 \cdot M_2 \cdot M_{n-1} \cdot M_n$$

公式中 C 是被建立的合成矩阵，M_1 到 M_n 是矩阵 C 包含的独立变换。大多数情况下，仅有 2 个或 3 个矩阵连接，但这是没有限制的。
使用 Matrix.Multiply 方法执行矩阵乘法。
矩阵乘法执行的顺序是至关重要的。前面的公式表示了矩阵串联的从左到右的规则。那就是说，你用来建立合成矩阵的那些矩阵在视觉效果上是按照从左到右的顺序发生的。下面的范例表示了 1 个典型的世界变换矩阵。想象一下你正在为 1 个立体造型的飞碟建立世界变换矩阵。你可能会想围绕它的中心--模型空间的 y 轴旋转飞碟，并将它平移到场景中的某个其他位置。为了完成这些效果，你首先建立 1 个旋转矩阵，然后将它与 1 个平移矩阵相乘，如下列公式所示。

$$W = R_y \cdot T_w$$

公式中 R_y 是关于 y 轴的旋转矩阵，T_w 是世界坐标系中某个位置的平移。
你将矩阵相乘的顺序是重要的，因为不同于 2 个标量值相乘，矩阵乘法不是不可交换的。以相反顺序将矩阵相乘，视觉效果将是平移飞碟到世界空间位置，然后围绕世界原始位置旋转。
不管你建立了何种类型的矩阵，记住从左到右的规则以确保你完成所期望的效果。

第4节　投影变换

投影变换可以认为是对照相机内部的控制；它类似于为照相机选择 1 个镜头。这是三种变换中最复杂的一种。投影变换的讨论由以下主题构成。

什么是投影变换？
建立 1 个投影矩阵

1 个 W-友好的投影矩阵

4.1 什么是投影变换？

投影变换通常是 1 种缩放和透视投影。投影变换将观察平截头体转化为 1 个立方体外形。观察平截头体的近端小于远端，这有 1 种将靠近照相机的物体扩大的效果。这就是透视图如何应用到场景中的。

在观察平截头体中，照相机和观察变换空间之间的距离被任意定义为 D，因此投影矩阵看上去就像下面描述的那样。

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

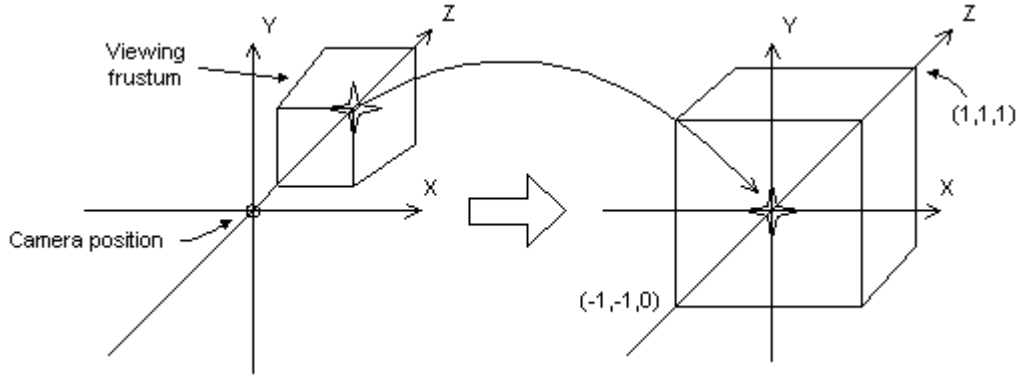
观察矩阵通过在 z 方向上平移-D，将照相机平移到原始位置。平移矩阵如下所示。

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -D & 1 \end{bmatrix}$$

将平移矩阵和投影矩阵(T*P)相乘得到合成投影矩阵。它看上去如下所示。

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & -D & 0 \end{bmatrix}$$

下图描绘了透视图变换是如何将 1 个观察平截头体转化为 1 个新的坐标空间。注意平截头体变成了立方体并且源点从场景的右上角移到了中心。



在透视图变换中，x-和 y-方向的限制为-1 和 1。z-方向的限制在前位面是 0，后位面是 1。该矩阵基于 1 个指定的从照相机到近端剪裁位面之间的距离来平移和缩放物体，但它不认为视野范围（fov）和为物体产生的 z-值在距离上是相同的，因这使得深度比较变得困难。下面的矩阵表述了这些问题，并调整顶点以适应观察窗口的宽高比，这使得它成为 1 个透视投影的好选择。

$$\begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & Q & 1 \\ 0 & 0 & -QZ_n & 0 \end{bmatrix}$$

下面的矩阵中，Z_n 是近端剪裁位面的 z-值。变量 w，h 和 Q 具有以下含义。注意 fov_w 和 fov_k 用弧度表示观察窗口的水平和垂直视野。

$$w = \cot\left(fov_w/2\right)$$

$$h = \cot\left(fov_k/2\right)$$

$$Q = \frac{Z_f}{Z_f - Z_n}$$

在你的程序中使用视野角度来定义 x-和 y-缩放比例系数也许不如使用观察窗口的水平和垂直维数（在照相机空间中）方便。作为数学计算，下列 2 个为 w 和 h 准备的公式使用了观察窗口维数，并且和前面的公式是等价的。

$$w = \frac{2 \cdot Z_n}{V_w}$$

$$h = \frac{2 \cdot Z_n}{V_k}$$

在这些公式中，Z_n 代表了近端剪裁位面的位置，V_w 和 V_h 变量代表了照相机空间中观察窗口的宽度和高度。

在托管程序中，这些二维数直接对应于 Viewport结构体的 Width和 Height成员。

z-值变换不大。这使得使用 16 位 z-缓存的深度比较变得稍微有些复杂。

你可以使用世界和观察变换，并调用 Device.SetTransform方法来设置投影变换。

4.2 建立 1 个投影矩阵

下面的ProjectionMatrix范例函数设置了前剪裁和后剪裁位面，以及水平和垂直视野角度。这段C#代码导入了 What Is the Projection Transformation?中讨论的方法。视野范围应比pi度数要小一些。

```
    [C#]



private Matrix ProjectionMatrix(float near_plane, float far_plane,
                                float fov_horiz, float fov_vert)
{
   float h,w,Q;
```

```
    w =  1/(float)Math.Tan(fov_horiz*0.5); // 1/tan(x) == cot(x)
    h =  1/(float)Math.Tan(fov_vert*0.5); // 1/tan(x) == cot(x)
    Q = far_plane/(far_plane - near_plane);

    Matrix ret = new Matrix();

    ret.M11 = w;
    ret.M22 = h;
    ret.M33 = Q;
    ret.M43 = -Q*near_plane;
    ret.M34 = 1;

    return ret;

}
```

在建立矩阵后，通过 SetTransform设置它，指定 TransformType.Projection，或将矩阵传递给 Device.Transform.Projection属性。
托管代码的 DirectX 9.0 提供下列函数帮助你设立 1 个投影矩阵。

- Matrix.PerspectiveLH
- Matrix.PerspectiveRH
- Matrix.PerspectiveFovLH
- Matrix.PerspectiveFovRH
- Matrix.PerspectiveOffCenterLH
- Matrix.PerspectiveOffCenterRH

4.3 1 个 W-友好的投影矩阵

Direct3D可以在深度缓存或雾化效果中使用已经过世界变换的顶点w-分量、view和投影矩阵来执行基于深度的计算。计算这些需要 1 个投影矩阵，将w等价的规范转化为世界空间中z值。简而言之，如果 1 个投影矩阵包含 1 个（3.4）系数而不是 1，你就必须通过将（3，4）系数反转以生成 1 个正确的矩阵来缩放所有系数。如果你没有提供 1 个合适的矩阵，雾化效果和深度缓冲将不能正确应用。在 What Is the Projection Transformation?中推荐的投影矩阵是适合基于w值的计算的。
下图描绘了 1 个不适合的投影矩阵和 1 个同样缩放了的矩阵，它可以启用肉眼相关的雾化。

Non-compliant

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & e \\ 0 & 0 & d & 0 \end{bmatrix}$$

Compliant

$$\begin{bmatrix} a/e & 0 & 0 & 0 \\ 0 & b/e & 0 & 0 \\ 0 & 0 & c/e & 1 \\ 0 & 0 & d/e & 0 \end{bmatrix}$$

在前面的矩阵中，所有变量都被假设为非零值。更多关于肉眼相关的雾化信息，参看 Eye-Relative vs. Z-Based Depth。更多关于基于w值的深度缓冲信息，参看 Depth Buffers。

注意：Direct3D 使用当前设置的投影矩阵用于基于 w 值的计算中。结果程序必须设置 1 个合适的投影矩阵来接收所期望的基于 w 值的特性，即使它们没有使用 Direct3D 变换流水线。

第5章　Direct3D 渲染

程序使用一组绘制造型的方法渲染三维场景。

第1节　着色

这部分描写了 Direct3D 中使用的控制三维多边形着色的技术。

1.1 着色模式

用于渲染多边形的着色模式在外观上具有一个深刻的效果。着色模式决定多边形表面任一点的色彩强度和亮度。Direct3D 支持 2 种着色模式：平面法和高洛德法。

1.1.1 平面着色

在平面着色模式下，Direct3D 渲染流水线使用多边性第一个顶点的材质色彩作为整个多边形的色彩来渲染该多边形。如果多边形不共面，被平面法渲染的三维物体在多边形之间将具有可视的清晰边缘。

下图描绘了一个由平面着色法渲染的茶壶。每个多边形的轮廓清晰可见。平面着色是最快的着色方式。

1.1.2 高洛德着色

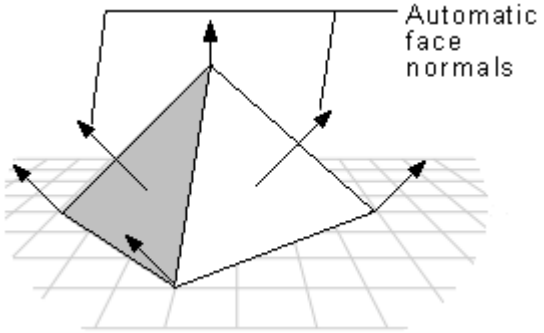
当 Direct3D 使用高洛德着色渲染多边形时，它使用顶点法线和光照参数为每个顶点计算色

彩。然后，它穿越多边形的表面进行插值。插值以线性方式完成。如果顶点 1 的红色色彩分量是 0.8，顶点 2 的红色色彩分量是 0.4，Direct3D 光照模块使用高洛德着色模式和 RGB 色彩模型，为这些顶点连线中点位置上的像素赋予一个 0.6 的红色色彩分量。
下图示范了高洛德着色。这个茶壶由许多平面的，三角形的多边形组成。然而，高洛德着色使物体的表面看上去弯曲和平滑。

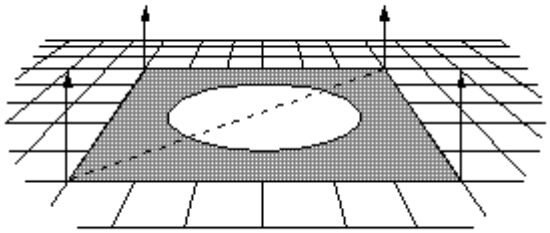
高洛德着色也能常被用于显示具有清晰边缘的物体。
更多信息，参看 面和顶点法向量。

1.2 比较着色模式

在平面着色模式下，下面显示了在毗邻表面具有清晰边缘的金字塔。然而在高洛德着色模式下，着色值是穿越边界插值的，最终外观具有一个曲面。



高洛德着色法照亮的平面比平面着色法更加真实。平面着色模式下的表面具有统一格式的色彩，但高洛德着色允许光线更加正确的落于表面。如果附近有 1 个点光源，这种效果将尤其明显。
高洛德着色使那些原本在平面着色中看上去清晰的多边性交界边缘更加平滑。然而这会导致 Mach bands 效应。程序可以通过增加物体中多边形的数量，增加屏幕分辨率，或增加程序色彩深度来降低 Mach bands 效应带来的外观影响。
高洛德着色可能会损失某些细节。一个例子如下图所示，一个聚光灯完全被包含在一个多边形表面内部。



这种情况下，高洛德着色法在顶点之间进行插值，会完全损失掉聚光灯，渲染出来的表面就像聚光灯不存在一样。

1.3 设置着色模式

Direct3D允许一次选择一种着色模式。缺省选择是高洛德着色法。在托管代码的DirectX 9.0 中，着色模式可以通过设置 RenderStateManager对象的 ShadeMode属性进行改变，该属性由 Device.RenderState属性返回。
下列 C#代码范例描述了 Direct3D 程序的当前着色模式是如何被设置为平面或高洛德模式的。

[C#]

```
// This code example assumes that device is already initialized .
// Set to flat shading.

device.RenderState.ShadeMode = ShadeMode.Flat;

// Set to Gouraud shading. This is the default for Direct3D.
device.RenderState.ShadeMode = ShadeMode.Gouraud;
```

第2节　显示场景

这部分介绍了描绘表达类 APIs 并讨论了关于将一个场景呈现到显示器上的问题。

2.1 显示 1 个场景的简介

描绘表达类(APIs)是一组控制可以让用户看到显示器上呈现内容的设备状态的方法。它包含了用于设置显示模式和常用于向用户呈现图像的 once-per-frame 等方法。

Device.Present
Device.Reset
Device.GetGammaRamp
Device.SetGammaRamp
Device.GetRasterStatus

熟悉下列术语有助于理解描绘表达类 APIs。

front buffer 前台缓存：内存中的一块矩形，通过图形适配器翻译，并被呈现在显示器或其它输出设备上。
back buffer 后台缓存：一个内容可以被提升为前台缓存的表面。
swap chain 交换链表：一个可以连续被显示为前台缓存的后台缓存集合。通常，一个全屏交换链表通过翻动设备驱动界面（ DDI ）来显示后面的图像，一个窗口

交换链表则通过位图传输 DDI 来显示图像。

因为 DirectX 9.0 下的 Direct3D 将交换链表作为设备的一个属性，所以每个设备总是有至少一个交换链表。设备类包含了一组操纵隐式交换链表的方法，这些方法是交换链表自身界面的一个拷贝。程序可以创建额外的交换链表；然而，对于典型的单窗体或全屏程序来说这不是必须的。

在DirectX 9.0 中，前台缓存不直接暴露在Direct3D API中。结果，程序不能锁定或渲染前台缓存。更多信息参看 访问前台缓存色彩。

注意：Direct 7.0 提供了大量同时调用的描述表达类 APIs。一个好的例子是 IDirectDraw7::SetCooperativeLevel，IDirectDraw7::SetDisplayMode，和 IDirectDraw7::CreateSurface 序列。另外，IDirectDrawSurface7::Flip 和 IDirectDrawSurface7::Blt 方法通知被渲染帧传输到显示器。DirectX 9.0 废除了这几组 APIs 合并为 2 种方法，Reset 和 Present。 Reset 包含了 SetCooperativeLevel，SetDisplayMode, CreateSurface，还有某些用于 flip 的参数。Present 包含了翻动和使用了 blit 的描述表达。

对 Manager 的调用描述了一个对设备隐式的重置。DirectX 9.0 API 没有主表面的概念；因为主表面被认为是一个设备的内部属性，所以不可能建立一个描述主表面的对象。

伽玛ramp和一个交换链表有关，并由 GetGammaRamp和 SetGammaRamp方法操纵。

2.2 窗口模式下的多视角

除了属于 Device 对象并由其操纵的交换链表之外，程序可以使用 SwapChain.SwapChain方法来建立额外的交换链表从相同的设备上进行多视角显示。

通常，程序建立为每个视角建立一个交换链表，并将每个交换链表与一个特定的视角相关联。程序在每个交换链表的后台缓存中渲染图像，然后使用 SwapChain.Present方法来分别显示它们。注意在每个适配器上每次仅能有一个交换链表为全屏。

2.3 多显示器操作

当设备在全屏操作中成功重置（通过 Device.Reset）或建立（通过 Device.Device），由设备建立的Direct3D对象被标识为系统的所有适配器独有。这个状态被认为是独占模式，并属于Direct3D对象所有。独占模式意味着通过其他任何Direct3D对象建立的设备都不能采取全屏操作或分配显存。另外，当Direct3D对象采取独占模式时，除了该设备可以进入全屏外，其他所有设备都被设置为丢失状态。更多信息参看 Lost Devices。

设备使用的焦点窗口将连同通知 Direct3D 对象和独占模式。当那个属于 Direct3D 对象的最终全屏设备被重置为窗口模式或被销毁时，独占模式将被释放。

当一个 Direct3D 设备属于独占模式的时候，设备可以分成 2 种。第一种设备有下列特性。

由和建立全屏设备的相同 Direct3D 对象所建立。

和全屏设备具有相同的焦点窗口。

代表了一个与任何全屏设备所不同的适配器。

这种设备没有关于被重置或被建立的能力限制，它们不会被设置为丢失状态。它们甚至能进入全屏模式。

不归于第一种的设备-那些由其他 Direct3D 对象建立的设备，具有一个不同的焦点窗口，代表了已经全屏设备的适配器-不能重置并处于丢失状态直到独占模式消失。结果是一个多显示器程序可以在全屏模式下放置多个设备，但它们全部仅是代表了不同的适配器，它们由相同 Direct3D 对象建立并共享相同的焦点窗口。

2.4 操纵深度缓存

深度缓存和设备相关联。当程序设置了渲染状态时，要求程序移动深度缓存。Device.DepthStencilSurface属性常用于操纵深度缓存。

2.5 访问前台缓存色彩

通过 Device.GetFrontBufferData方法可以允许对前台缓存的访问。这个方法是获得抗锯齿场景的屏幕快照的唯一方式

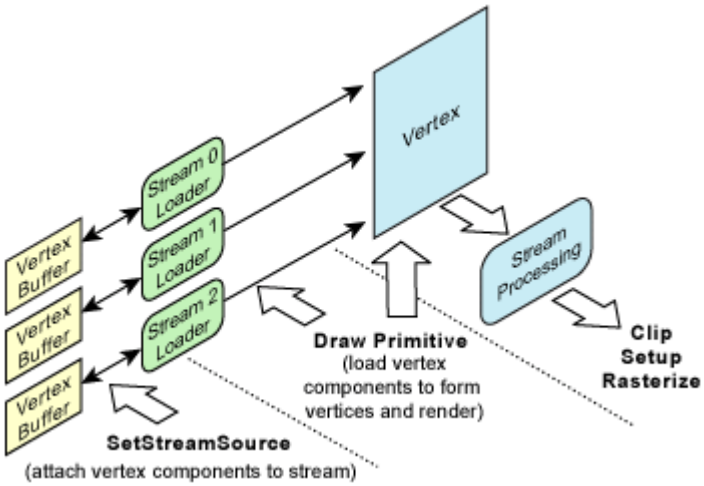
第3节　渲染造型

下列主题提供了关于在程序中绘制造型的信息。

顶点数据流
设置流源
从顶点和索引缓存渲染
从用户内存指针渲染

3.1 顶点数据流

通过存储于 1 个或多个数据缓存中的顶点数据对造型进行渲染的方法组成了 Direct3D 渲染界面。顶点数据是由顶点元素组合起来的顶点分量所构成。顶点元素-最小的顶点单位-表达了诸如位置，法线或色彩的实体属性。
顶点分量是 1 个或多个连续存储在单独内存块中的顶点元素。1 个完整的顶点包含了 1 个或多个分量，每个分量都位于一块分离的内存缓冲区。在渲染造型时，顶点的多个分量被读入并装配组合使完成后的顶点可用于顶点处理。
下图描绘了使用顶点分量时造型渲染的处理过程。



造型渲染由 2 个步骤组成：设立 1 个或多个顶点分量流，然后调用 Device.DrawPrimitives 方法对这些流进行渲染。对这些分量流内部的顶点元素验证是通过顶点着色器说明的。
Device.DrawPrimitives方法指定了顶点数据流中的一个偏移量。因此所有绘制命令都可

以对具有一组顶点数据的造型的任意相邻子集进行渲染。这使得在相同顶点缓存渲染得来的造型组之间进行设备渲染状态的变更成为可能。
索引和非索引方法都被支持。更多信息，参看 Rendering from Vertex and Index Buffers。

3.2 设置流源

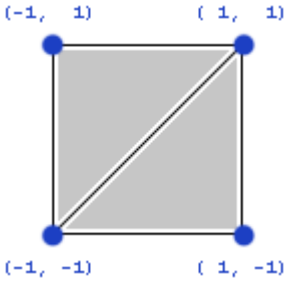
Device.SetStreamSource方法将一个顶点缓存绑定到一个设备数据流上，在顶点数据和用于传入造型处理函数的数据流端口之间建立了关联。只有调用了诸如 Device.DrawPrimitives 绘制方法，才会发生对流数据的实际引用。
流被定义为一个有格式的分量数据的数组，数组中每个分量由 1 个或多个元素组成，它们代表了 1 个单独的实体属性，诸如位置，法线或色彩。Device.SetStreamSource方法的 Stride参数按比特位指定了分量的尺寸。

3.3 从顶点和索引缓存渲染

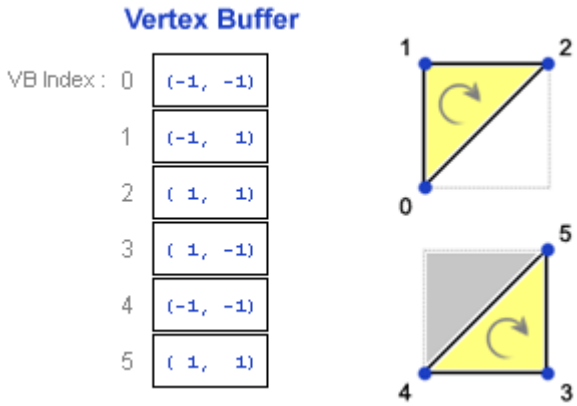
Direct3D 支持索引和非索引 2 种绘制方法。索引方法为所有顶点分量使用了一组单独的索引。顶点数据存储在顶点缓存中，索引数据存储在索引缓存中。下面列出的是使用顶点和索引缓存，在绘制造型时的一小部分常用情形。

3.3.1 无索引绘制 2 个三角形

假设你想绘制下列四方形。



如果你使用三角形列表造型渲染这 2 个三角形，每个三角形都作为 3 个独立顶点存储，在一个顶点缓存中的结果和下面所示类似。



执行绘制调用时，在顶点缓存内从位置 0 开始绘制 2 个三角形。如果启用了消隐功能，顶

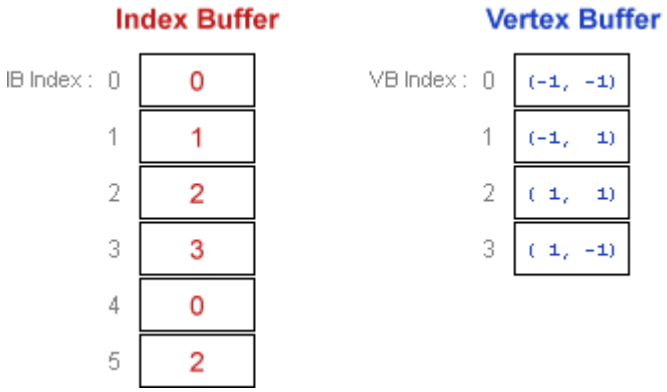
点顺序是重要的。假设这个范例采取了缺省的逆时针消隐状态，因此必须按照顺时针序绘制可视三角形。三角形列表造型类型简单的从每个三角形的缓存中线性读取三个顶点，因此下列 C#调用绘制了三角形(0, 1, 2)和(3, 4, 5)。

```
[C#]


Device.DrawPrimitives(PrimitiveType.TriangleList,     // PrimitiveType
                      0,                              // StartVertex
                      2);                             // PrimitiveCount
```

3.3.2 索引绘制 2 个三角形

你可能注意到，顶点缓存在位置 0 和 4，位置 2 和 5 包含了重复数据。因为 2 个三角形共享 2 个公用顶点，所以这是有意义的。重复数据是浪费的，可以使用一个索引缓存来压缩顶点缓存。一个较小的顶点缓存降低了必须传输给图形适配器的顶点数据量。更重要的是，使用一个索引缓存可以让适配器在一个顶点高速缓存中存储顶点；如果被绘制的造型包含了一个最近使用过的顶点，那个顶点就可以不通过从顶点缓存中读取而从高速缓存中获得，这在性能上有显著的增长。
一个索引缓存‘索引’到顶点缓存，因此每个唯一的顶点必须在顶点缓存中仅被存储一次。这是一个绘制上述情形的索引方法。



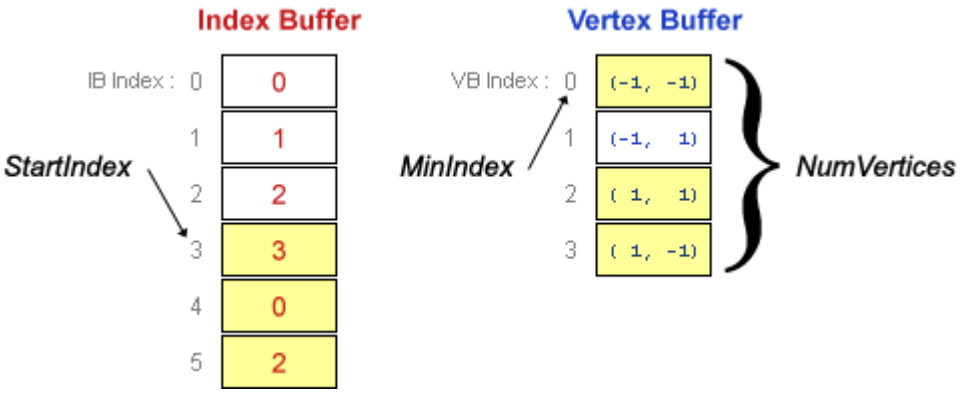
索引缓存存储了 VB 索引值，该值在顶点缓存中引用一个特定的顶点。顶点缓存可以看成是一个顶点数组，因此 VB 索引仅是一个顶点缓存内部用于目标顶点的索引。同样的，IB 索引是一个索引缓存内的索引。如果不仔细一点，这些可能会很快使你糊涂，因此要确定你对于使用的术语词汇很清楚：VB 索引值索引到顶点缓存内部，IB 索引值索引到索引缓存内部，索引缓存自身存储了 VB 索引值。
绘制调用语句如下所示。最后作为下一个绘制情形，将讨论到全部参数的意义；现在，仅注意这个调用语句再次通知 Direct3D 渲染一个包含有 2 个三角形的三角形列表，在索引缓存中开始于位置 0。下列 C# 调用和前面完全相同的顺序绘制了相同的 2 个三角形，确保了一个正确的顺时针方向。

```
[C#]


Device.DrawIndexedPrimitives(PrimitiveType.TriangleList,  // PrimitiveType
                             0,                           // BaseVertexIndex
                             0,                           // MinIndex
                             4,                           // NumVertices
                             0,                           // StartIndex
                             2);                          // PrimitiveCount
```

3.3.3 索引绘制 1 个三角形

现在假设你想仅绘制第 2 个三角形，但你想使用在绘制整个四方形时使用的相同顶点缓存和索引缓存。



这个绘制调用使用的第 1 个 IB 索引是 3；该值叫做 StartIndex 。使用的最小 VB 索引是 0；该值叫做 MinIndex 。即使仅需要用 3 个顶点绘制该三角形，这 3 个顶点也将会被播撒在顶点缓存的四个相邻位置上；绘制调用语句需要顶点缓存邻近内存块中被称为 NumVertices 的位置数量，它在这个调用中被设置为 4。在软件顶点处理期间，MinIndex 和 NumVertices 值实际上仅仅提示帮助 Direct3D 优化内存访问，它们仅能够以性能为代价来换取包含整个顶点缓存。

这是一个 C#绘制单个三角形情况下的调用语句。BaseVertexIndex 参数的意义将在下面解释。

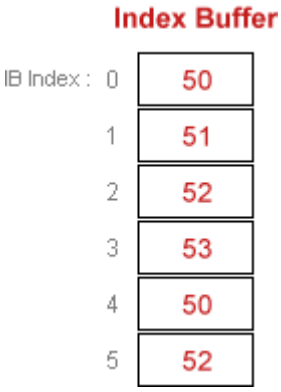
```
[C#]


Device.DrawIndexedPrimitives(PrimitiveType.TriangleList,  // PrimitiveType
```

```
                                                    0,
// BaseVertexIndex
                                                    0,
// MinIndex
                                                    4,
// NumVertices
                                                    3,
// StartIndex
                                                    1);
// PrimitiveCount
```

3.3.4 偏移量索引绘制 1 个三角形

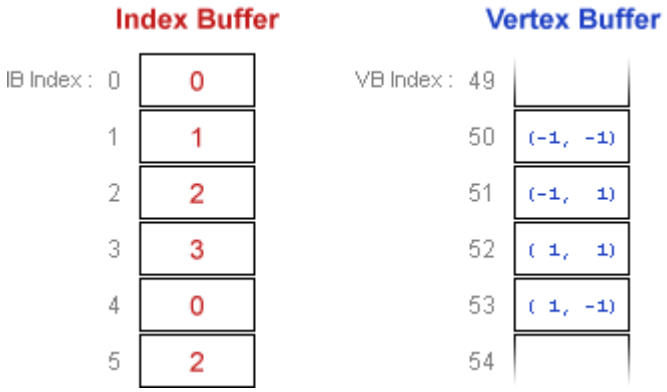
BaseVertexIndex 是一个有效加入到每个VB索引中的值，它存储于索引缓存中。比如，如果在早先调用期间，有一个 50 的值被作为BaseVertexIndex 传入，它和用于Device.DrawIndexedPrimitives调用期间的下列索引缓存具有相同的功能



这个值很少设置为除 0 以外的其它值，但是如果你想减弱索引缓存对顶点缓存的依赖时，它就有用了：假设在为一个特定网格填充索引缓存时，顶点缓存内的网格位置仍然未知，你只能假装网格顶点将会位于顶点缓存的开始处。当调用绘制语句时，仅将 BaseVertexIndex 作为实际开始位置进行传递。

当使用 1 个单独索引缓存来绘制 1 个网格的多个实例时，这种技术也是有用的；比如，如果顶点缓存包含了 2 个具有相同绘制顺序但顶点略有不同（可能是漫反射色彩或纹理坐标不同）的网格，则 2 个网格都可以使用不同的值作为 BaseVertexIndex 。进一步深化这个概念，你可以使用 1 个索引缓存来绘制 1 个网格的多个实例，每个实例都包含在一个不同的顶点缓存中，仅使用激活的顶点缓存并根据需要来调整 BaseVertexIndex 。注意 BaseVertexIndex 值是自动加入到 MinIndex 参数中的，该参数只有在你了解它是如何被使用时才有意义。

假设现在你想再次使用和前面相同的索引缓存，仅绘制四方形的第 2 个三角形；然而位于 VB 索引值 50 的四方形使用了 1 个不同的顶点缓存。四方形顶点的相对顺序没有改变，仅顶点缓存中的启示位置有所不同。索引缓存和顶点缓存看上去就像这样。



这是合适的 C#绘制调用；注意 BaseVertexIndex 是在前面的情形中唯一改变的值：

```
[C#]



Device.DrawIndexedPrimitives(PrimitiveType.TriangleList,  // PrimitiveType
                                                          50,
// BaseVertexIndex
                                                          0,
// MinIndex
                                                          4,
// NumVertices
                                                          3,
// StartIndex
                                                          1);
// PrimitiveCount
```

3.4 从用户内存指针渲染

一组次要的渲染界面支持直接由用户内存指针传递顶点和索引数据。这些界面仅支持一个单独的顶点数据流。更多信息参看下列引用主题。

Device.DrawUserPrimitives

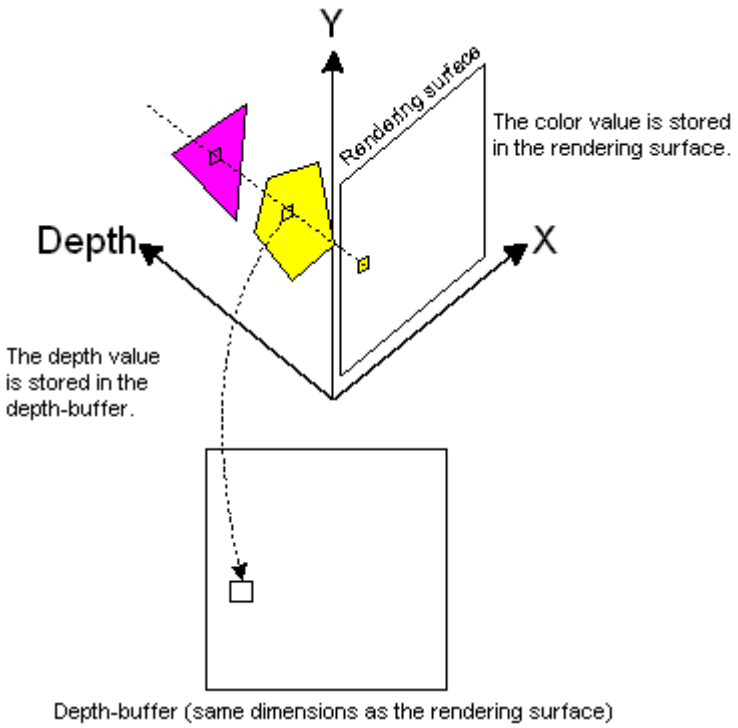
Device.DrawIndexedUserPrimitives

这些方法通过用户内存指针进行渲染而非通过顶点和索引缓存。

第4节　深度缓存

深度缓存，经常被称为 z-缓存或 w-缓存，是设备的一种属性，存储了 Direct3D 所使用的深度信息。当 Direct3D 将一个三维场景渲染到一个目标表面时，它能够使用深度缓存表面关联的内存作为一个工作空间来决定光栅化多边形的像素如何彼此闭塞。Direct3D 使用一

个离屏 Direct3D 表面作为填充最终色彩值的目标表面。和渲染目标表面关联的深度缓存表面常用于存储深度信息，告诉 Direct3D 场景中的每个可视像素有多深。
当一个启用了深度缓存的三维场景被光栅化时，渲染表面的每个点都被测试。深度缓存中的值可以是来自于投影空间内的点（x，y，z，w）的 z 坐标或它的同系 w 坐标。使用 z 值的深度缓存经常叫做 z-缓存，使用 w 值的叫做 w-缓存。每种类型的深度缓存都有优缺点，接下来将会讨论它们。
测试开始时，深度缓存中的深度值被设置为场景的可能最大值。渲染表面的色彩值被设置为该点的背景色或背景纹理色的二者之一。场景中的每个多边形都被测试以判断是否与渲染表面的当前坐标（x，y）相交。如果相交，当前点上的深度值（将会成为 z-缓存中 z 坐标和 w-缓存中的 w 坐标）被测试判断是否小于存储在深度缓存中的深度值。如果多边形的深度值小，它就被存储在深度缓存中，源于多边形的色彩值将被填充到渲染表面的当前点上。如果在该点的多边形深度值大，将测试列表中的下一个多边形。这个过程如下图所示。



注意：虽然大多数程序不使用这个特性，但它可以改变Direct3D用于决定哪个值放在深度缓存和随后的渲染目标表面的比较过程。做到这些，要改变 RenderStateManager.ZBufferFunction属性的值。
几乎市面所有的三维图形加速卡都支持 z-缓冲，将 z-缓存作为目前最普通的深度缓存类型。然而普遍存在的 z-缓存具有它们弱点。这与数学有关，z-缓存中生成的 z 值并非 倾向在 z-缓存范围中均匀分布（通常包含从 0.0 到 1.0）。尤其是远端和近端剪裁位面之间的比率极大的影响了 z 值的不均衡分布。使用一个远端位面距离和近端位面距离为 100 的比率，深度缓存范围的 90%都花费在场景深度的第一个 10%范围内。典型的娱乐用或具有户外场景的可视模拟程序经常要求任意地点的远端位面/近端位面比率在 1000 到 10000 之间。比率 1000 时，98%的范围花费在深度的第一个 2%范围内，更高比率的分布将变得更糟。这将会导致远处物体的表面被隐藏，尤其在使用通常支持的 16 位深度缓存时。
另一方面，一个基于 w 值的深度缓存通常比 z-缓存更加均匀的分布在近端和远端位面之间。关键的好处是远端和近端位面距离比率不再是问题。虽然这仍然只能获得相对准确接近于肉眼视点的深度缓存，但这让程序能够支持最大限度的范围。基于 w 值的深度缓存不是完美的，有时可能会显示出近处物体隐藏表面的假象。这种方法的另一个缺点是涉及到硬件支持：

w-缓存不像 z-缓存被硬件广泛支持。
Z-缓存在渲染期间需要天花板。当使用 z-缓存时，可以应用各种技术优化渲染。当确定场景是从前到后渲染的，程序使用 z-缓存和纹理能够提升性能。带有纹理和 z-缓存的造型将忽略 z-缓存，在一根屏幕扫描基线上进行预测试。如果扫描线被先前渲染过的多边形隐藏，系统就及时有效的丢弃它。Z-缓存能够改进性能，但这种技术在场景不止一次的描绘相同像素时最有用处。虽然难以准确计算，但通常可以估计出近似值。如果相同像素绘制少于 2 次，通过关闭 z-缓存并从后到前渲染场景，就可以获得最佳性能。
注意：对一个深度缓存值的实际阐述属于三维渲染器的内容。
下列主题表达了关于深度缓存用于隐藏线和隐藏表面消除的信息。

查询深度缓存支持
建立深度缓存
启用深度缓存
找回深度缓存
清除深度缓存
改变深度缓存的写入访问
改变深度缓存的比较函数
使用 z-斜线

4.1 查询深度缓存支持

程序使用的驱动程序，即使具备所有特性，也许不支持所有类型的深度缓存。因此始终要对驱动程序的功能进行检查。虽然大多数驱动程序支持基于 z 值的深度缓存，而不支持基于 w 值的深度缓存（w-缓存）。如果你尝试启用一个不支持的配置计划，驱动程序会失败。它们将会求助于另一个深度缓存方法作为替代，或有时完全禁用深度缓存，这可能会导致被渲染场景极大程度上的深度排序假象。
要检查深度缓存的常用支持，在建立一个 Direct3D 设备前，应向 Direct3D 查询你程序所使用的显示设备。如果 Direct3D 对象报告它支持深度缓冲，由它建立的任何硬件设备都将支持 z-缓存。
使用 Manager.CheckDeviceFormat方法来查询深度缓存的支持，如下列C#代码范例所示。

[C#]

```
AdapterInformation ai = Manager.Adapters.Default;

// Verify that the depth format exists.
if (Manager.CheckDeviceFormat(ai.Adapter,
                              DeviceType.Hardware,
                              adapterFmt,
```

```
                                    Usage.DepthStencil,
                                    ResourceType.Surface,
                                    DepthFormat.D16,
                                    out result))
{
    return true;
}

return false;    // If the call fails, the error code is passed back in the result.
```

Manager.CheckDeviceFormat方法允许你选择建立一个基于该设备功能的设备。既然这样，不支持 16 位深度缓存的设备就被丢弃。
使用 Manager.CheckDepthStencilMatch来检测深度模版与渲染目标兼容性的方法如下面C#代码范例所示。

```
            [C#]


// Reject devices that cannot create a render target of RTFormat while
// the back buffer is of RTFormat and the depth-stencil buffer is
// at least 8 bits of stencil.
AdapterInformation ai = Manager.Adapters.Default;

if (Manager.CheckDepthStencilMatch(ai.Adapter,
                                   DeviceType.Hardware,
                                   adapterFmt,
                                   RTFormat,
                                   DepthFormat.D24S8))
{
    return true;     // Device can create the render target.
}
```

```
return false;    // Reject the device; it cannot create the render target.
```

当你知道驱动程序支持深度缓存，你可以验证 w-缓存支持。虽然深度缓存为所有软件光栅所支持，但 w-缓存仅被引用光栅支持，而这种光栅不适用于真实世界程序。不管程序使用的设备类型，在启用基于 w 值的深度缓存的任何尝试之前，都必须先验证对 w-缓存的支持。下面的过程示范了如何验证对 w-缓存的支持。

1. 在建立设备后，从 Device.DeviceCaps属性中找回 Caps结构。
2. Caps.LineCaps属性包含了一个对 LineCaps结构的引用，它包含了关于驱动程序对渲染造型的支持信息。
3. 如果设备支持基于w值的用于造型类型的深度缓存，检查包含在 Caps.RasterCaps属性下的 RasterCaps结构体的 RasterCaps.SupportsWBuffer属性。

4.2 建立 1 个深度缓存

深度缓存是设备的一个属性。要建立一个由Direct3D托管的深度缓存，应按下列C#代码范例为 PresentParameters类设置 合适的成员。

```
        [C#]


PresentParameters presentParams = new PresentParameters();


presentParams.Windowed = true;
presentParams.SwapEffect = SwapEffect.Copy;
presentParams.EnableAutoDepthStencil = true;
presentParams.AutoDepthStencilFormat = DepthFormat.D16;
```

将 PresentParameters.EnableAutoDepthStencil成员设置为true通知Direct3D为程序管理深度缓存。注意 PresentParameters.AutoDepthStencilFormat必须设置成一个合法的深度缓存格式。DepthFormat.D16 值指定了一个可用的 16 位深度缓存。
下列对 Device设备构造器的C#调用语句依次建立了一个产生深度缓存的设备。

```
        [C#]


Device device;


// Create a device using the PresentParameters previously set.
device = new Device(0,
```

```
                    DeviceType.Hardware,
                    this,
                    CreateFlags.SoftwareVertexProcessing,
                    presentParams);
```

深度缓存自动被设置为设备的渲染目标。当设备重置时，深度缓存自动销毁并以新尺寸大小重新建立。使用 Device.CreateDepthStencilSurface方法建立一个新的深度缓存表面。使用 Device.DepthStencilSurface方法为设备设置一个新的深度缓存表面。
要在程序中使用深度缓存，必须启用深度缓存。更多信息参看 Enabling Depth Buffering。

4.3 启用深度缓存

在建立一个深度缓存之后（如 Creating a Depth Buffer 一节所述），你通过将 Device.RenderState 属性的 RenderStateManager.ZBufferEnable 属性设置为 true 来启用深度缓存。要启用 w-缓存，应将设备渲染器状态 RenderStateManager.UseWBuffer 属性设置为 true。这些步骤如下面 C#代码范例所示。

```
            [C#]


using System;
using Microsoft.DirectX.Direct3D;


    // Declare a rendering device.
    Device device = null;


    // Initialize the device.
    .
    .
    .


    // Get the device's render state object.
    RenderStates rs = device.RenderState;
```

```
    // Enable depth buffering.
    rs.ZBufferEnable = true;

    // Render the scene.
    .
    .
    .

    // Disable depth buffering
    // (note the different syntax; both forms are acceptable)
    device.RenderState.ZBufferEnable = false;

    // Use w-buffering.
    device.RenderState.UseWBuffer = true;
```

注意：要使用w-缓存，即使没有使用Direct3D变换流水线，程序必须设置一个合适的投影矩阵。关于一个合适投影矩阵的更多信息提供，参看 1 个W-友好的投影矩阵。在 什么是投影变换?中讨论的投影矩阵是适合的。

4.4 获取深度缓存的返回值

下列C#代码范例示范了如何使用 Device.DepthStencilSurface属性来找回一个 Surface 对象，这用于描述属于设备的深度缓存表面。

```
            [C#]


Device device = null;
Surface surface = null;

// Create the device.
.
.
.
```

```
surface = device.DepthStencilSurface;
```

4.5 清除深度缓存

许多C#程序在渲染每个新帧前清除深度缓存。在Direct3D中，可以显式的调用Device.Clear和为flags参数指定 ClearFlags.ZBuffer来清除深度缓存。Device.Clear方法可以规范zdepth 参数中的任意深度值。
下列C#代码范例示范了如何清除深度缓存。

```
        [C#]


using Microsoft.DirectX.Direct3D;


Device device = null;


// Create a rendering device.
.
.
.


// Clear the depth buffer.
device.Clear(ClearFlags.ZBuffer, 0, 0.5f, 0);
```

4.6 改变深度缓存的写入访问

Direct3D 缺省允许对深度缓存进行写入。大多数程序对深度缓存启用写操作，但是在Direct3D 系统中要关闭这项功能实现某些特殊效果。
在C#中禁用深度缓存，应将 Device.RenderState 属性的RenderStateManager.ZBufferWriteEnable属性设置为false。下列C#代码范例进行了描述。

```
        [C#]


using System;
```

```
using Microsoft.DirectX.Direct3D;

    // Declare a rendering device.
    Device device = null;

    // Initialize the device.
    .
    .
    .

    // Get the device's render state object.
    RenderStates rs = device.RenderState;

    // Disable depth buffer writes.
    rs.ZBufferWriteEnable = false;

    // Render the scene.
    .
    .
    .

    // Enable depth buffer writes
    // (note the different syntax; both forms are acceptable)
    device.RenderState.ZBufferWriteEnable = true;
```

4.7 改变深度缓存的比较函数

当在一个渲染表面上执行深度测试时，如果每个点的对应深度值（z或w）小于深度缓存中的值，Direct3D缺省将更新渲染目标表面。在一个C#程序中，关于系统是如何对深度值进行比较的变化是由 Compare 枚举类型的一个值的 Device.RenderState 属性的 RenderStateManager.ZBufferFunction 属性设置的。下面用C#代码范例进行了描述。

[C#]

```
using System;
using Microsoft.DirectX.Direct3D;

    // Declare a rendering device.
    Device device = null;

    // Initialize the device.
    .
    .
    .

    // Get the device's render state object.
    RenderStates rs = device.RenderState;

    // Set the ZBuffer comparison function.
    rs.ZBufferFunction = Compare.Greater;

    // Render the scene.
    .
    .
    .

    // Change the comparison function to some other value
    // (note the different syntax; both forms are acceptable)
    device.RenderState.ZBufferFunction = Compare.LessEqual;
```

4.8 使用 Z-斜线

在三维场景中，可以通过为每个共面多边形加入一个 z-斜线使它们看上似乎是不共面的。这是一种用于保证场景中的阴影正确显示的常用技术。例如墙上的一个阴影可能具有和墙体相同的深度值。如果墙体在阴影前被渲染，阴影也许就不可视，或者会看见一种深度假象。我们希望反转被渲染的共面物体的顺序可以改变这种影响，但是深度假象可能仍然存在。
在系统渲染一组共面多边形时，一个C#程序可以向系统使用的z值加入 一个斜线，以帮助确定共面多边形渲染的顺序。要向一组多边形加入一个斜线，应将Device.RenderState 属性的 RenderStateManager.DepthBias属性设置为一个 0 到 16 之间的值。较高z-斜线值的被渲染多边形，在和其他共面多边形一起显示时，其被看见的可能性越大。
下面的 C#代码范例示范了如何设置 z-斜线值。

```
        [C#]


using System;
using Microsoft.DirectX.Direct3D;

    // Declare a rendering device.
    Device device = null;

    // Initialize the device.
    .
    .
    .

    // Get the device's render state object.
    RenderStates rs = device.RenderState;

    // Set the depth bias value.
    rs.DepthBias = 7.0f;

    // Render the scene.
    .
```

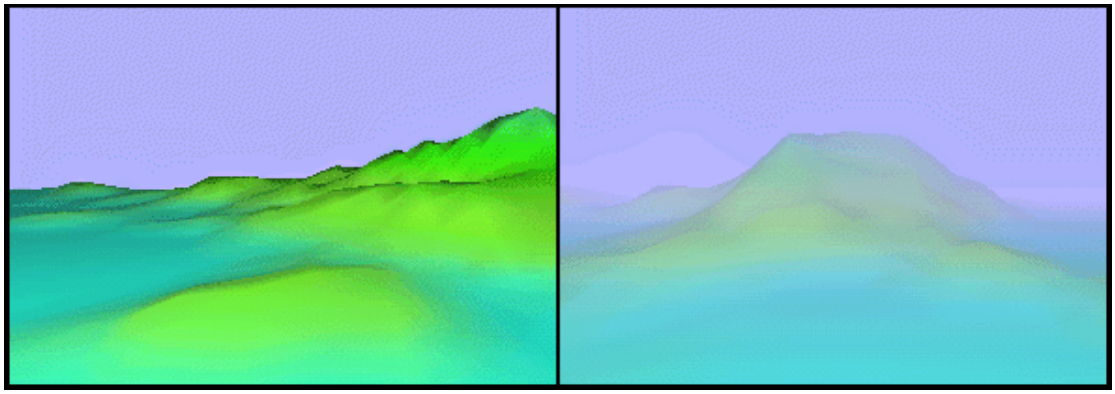
```
.
.


// Change the depth bias value
      // (note the different syntax; both forms are acceptable)
device.RenderState.DepthBias = 3.0f;
```

第5节　雾化

为三维场景加入雾可以提高真实度，提供气氛或设置一种情境，有时会产生使远处进入视线的几何体模糊的假象。Direct3D 支持 2 种雾化模型，像素雾化和顶点雾化，它们每一种都拥有自己的特性和编程界面。

实质上，雾是由场景中的物体色彩与一种选定的雾化色彩的混合产生的，这种混合基于场景中物体的深度或它距离视点的远近。当物体越来越远时，它们的原始色彩与选定的雾化色彩混合也越来越更多，这产生了一种物体被场景中悬浮的微小颗粒逐渐模糊的幻觉。下图描绘了一幅雾化前后渲染场景的对比。



左图中的场景有一个清晰的地平线。超过了它，那么即使在真实世界中可见的场景也将看不见。右图中的场景使用一种和背景色相同的雾化色彩模糊了地平线，使得多边形看上去淡入远方。在创造性的场景设计中，结合不连续的雾化效果，能够在场景中加入情境和柔化物体的色彩。

Direct3D提供了 2 种为场景加入雾的方法：像素雾化和顶点雾化，名称来源于应用雾化效果表达的方式。更多信息参看 Pixel Fog和 Vertex Fog。简而言之，像素雾化又叫表格雾化，是在设备驱动程序上完成的，而顶点雾化是在Direct3D光照引擎上完成的。程序可以在需要时同时使用二种类型的雾化。

注意：不管使用的是像素还是顶点雾化，程序必须提供一个合适的投影矩阵保证雾化效果的正确应用。这种限制甚至还用于没有使用Direct3D变换和光照引擎的程序。更多关于如何和提供一个合适矩阵的信息，参看 1 个W-友好的投影矩阵 。

下列主题介绍了雾化并描述了关于在 Direct3D 程序中使用各种雾化特性的信息。

雾化公式
雾化参数
雾化混合

雾化色彩
顶点雾化
像素雾化

雾化混合由渲染状态控制；它不是可编程像素流水线中的一部分。

5.1 雾化公式

通过改变 Direct3D 计算雾化效果穿过距离的方式，C#程序可以控制雾如何影响场景中物体的色彩。FogMode 枚举类型包含了识别三种雾化公式的成员（线性雾化和 2 种指数雾化）。所有公式都将距离函数作为雾化因子来计算，该因子是程序设置的给定参数。

5.1.1 线性雾化

$$f = \frac{end - d}{end - start}$$

start 是雾化效果开始的距离。
end 是雾化效果不再增加的距离。
d 代表深度，或离开视点的距离。对于基于范围的雾来说， d 值是照相机位置和顶点之间的距离。对于非基于范围的雾来说， d 值是照相机空间内的 z 坐标绝对值。

5.1.2 指数雾化

线性和指数公式都为像素雾化和顶点雾化所支持。

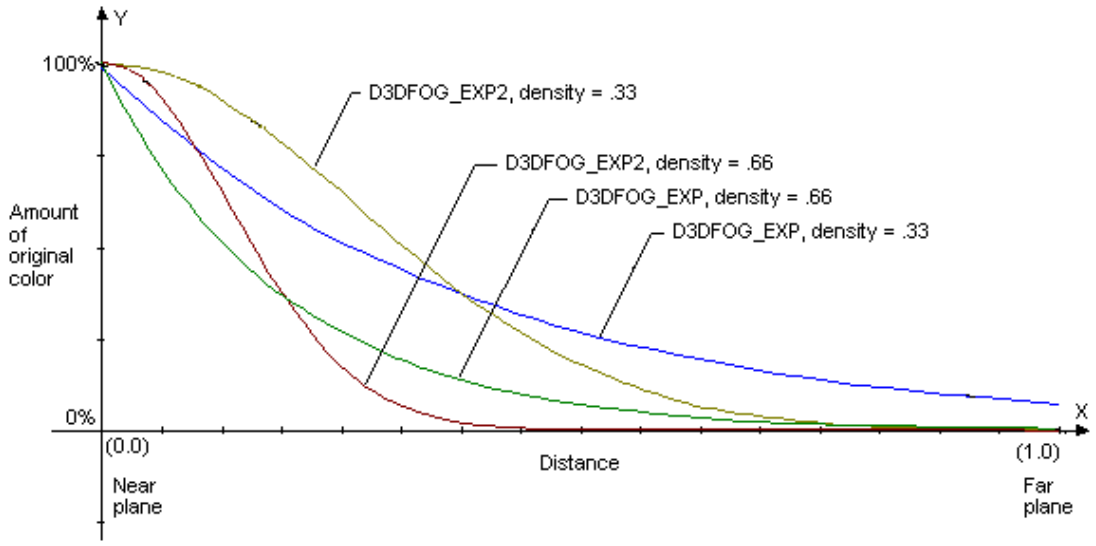
$$f = 1 / e^{d \times density}$$

$$f = 1 / e^{(d \times density)^2}$$

e 是自然对数的基数（约为 2.71828）。
density 是一个范围从 0.0 到 1.0 的任意雾化密度。
如上所述， d 代表了深度，或离开视点的距离。

注意：系统在顶点的镜面反射色彩的 alpha 分量中存储了雾化因子。如果程序执行了自己的变换和光照，那么在渲染期间，可以手工插入系统应用的雾化因子值。

下面的曲线图使用公式参数中的常用值来描述这些公式。



线性 Linear开始于 1.0 终止于 0.0。它相对于近端或远端位面是不整齐规则的。
Direct3D 计算雾化效果时，使用了雾化因子，它是由前面所述的混合公式中的一种方程式得来的，这里表示为：

$$C = f \cdot C_i + (1-f) \cdot C_f$$

通过雾化因子 f，这个公式可以有效地按比例确定当前多边形的色彩 C，并将该乘积与由雾化因子的位倒数按比例确定的雾化色彩 C 相加。结果色彩是一种与距离因素有关的原始色彩与雾化色彩的混合。公式可以应用到支持 DirectX 7.0 以上版本的所有设备。对于以前遗留下来的 ramp 设备，雾化因子在 0.0 到 1.0 之间测量漫反射和镜面反射色彩分量。雾化因子通常在近端位面开始于 1.0，在远端位面终止于 0.0。

5.2 雾化参数

雾化参数由设备渲染状态所控制。像素和顶点雾化类型均支持在 Fog Formulas中介绍的所有雾化公式。FogMode枚举类型定义的常量可以用于识别你想要Direct3D使用的雾化公式。RenderStateManager.FogTableMode属性控制Direct3D用于像素雾化的雾化模型，RenderStateManager.FogVertexMode属性控制用于的顶点雾化的模型。
当使用线性雾化公式时，通过 RenderStateManager.FogStart 和 RenderStateManager.FogEnd属性来设置开始距离和终止距离。系统如何插值取决于程序使用的雾化类型--像素或顶点，以及当使用像素雾化时，使用的深度--是基于z值还是 w 值。下表总结了雾化类型及它们的开始、终止单元。

雾化类型	雾化开始 / 结束单元
Pixel (Z)	Device space [0.0,1.0]
Pixel (W)	Camera space
Vertex	Camera space

当启用了指数雾化公式时，RenderStateManager.FogDensity属性控制雾化密度的应用。雾化密度实质是一个按指数比例测定距离值的额外因子，它的范围从 0.0 到 1.0（包含该值）。
系统用于雾化混合的色彩由 RenderStateManager.FogColor设备属性控制。更多信息参看 Fog Color和 Fog Blending。

5.3 雾化混合

雾化混合应用雾化色彩和物体色彩等雾化因子来生成呈现在场景中的最终色彩（在 Fog Formulas中讨论）。RenderStateManager.FogEnable属性控制雾化混合。其缺省值为false；将该渲染状态设为true可以启用雾化混合，如下面C#代码范例所示。

```
            [C#]


// For this example, device is a valid Device object.


// Get the device's render states object.
RenderStates rs = device.RenderState;


rs.FogEnable = true;
```

雾化混合必须同时为像素雾化和顶点雾化启用。更多关于这些雾化类型的信息，参看 Pixel Fog和 Vertex Fog 。

5.4 雾化色彩

像素雾化和顶点雾化的雾化色彩都是通过 RenderStateManager.FogColor属性设置的。当渲染状态值被指定为一个RGBA色彩时，它只能是一个RGB色彩。Alpha分量将被忽略。下列 C#代码范例将雾化色彩设置为白色。

```
            [C#]


// For this example, device is a valid Device object.


// Get the device's render states object.
RenderStates rs = device.RenderState;


rs.FogColor = System.Color.White;
```

通过修正功能流水线和可编程流水线应用雾化有所区别。

5.5 顶点雾化

当系统执行顶点雾化时，它对每个多边形的每个顶点进行雾化计算，然后在光栅化期间为多边形的所有面插入计算值。顶点雾化效果由Direct3D光照和变换引擎计算。更多信息，参看 Fog Parameters。

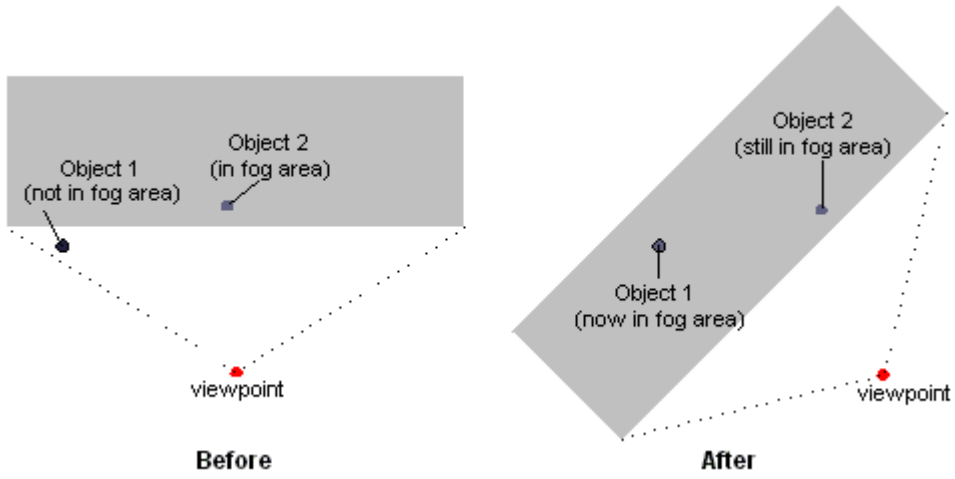
如果程序不使用Diect3D进行变换和光照，那么它必须执行雾化计算。因此，要为每个顶点放置通过镜面反射色彩的alpha分量计算出来的雾化因子。可以使用任何基于范围的、体积的或其它的公式。Direct3D使用供给的雾化因子来为每个多边形的表面进行插值。执行自身变换和光照操作的程序也必须执行它们自己的顶点雾化计算。结果，这样一个程序仅需要启用雾化混合并通过关联的渲染状态设置雾化色彩，详见 Fog Blending和 Fog Color。

注意：当使用顶点着色器时，必须使用顶点雾化。这是通过使用顶点着色器将每顶点雾化强度写入 oFog 寄存器完成的。在像素着色器完成后，oFog 数据常用于和雾化色彩线性插值。该强度在像素着色器上是不可用的。

5.5.1 基于范围的雾化

注意：仅当通过 Direct3D 变换和光照引擎来使用顶点雾化时，Direc3D 使用了基于范围的雾化计算。这是因为像素雾化是在设备驱动程序上完成的，通常没有硬件存在支持基于范围的每像素雾化。如果程序执行自身的变换和光照，它必须执行自己的基于范围或其它的雾化计算。

有时，使用雾化可能会引入导致物体直觉上与雾化色彩相混合的图形假象。比如，设想一个具有 2 个可视物体的场景：一个距离足够远至受到雾化影响，另一个距离足够近至不受影响。如果视线区域适当旋转，那么即使物体是固定的，其雾化效果的外观也可能会改变。下图描绘了一幅从上往下观看的视图情况。



基于范围的雾化是另一种更加准确决定雾化效果的方式。在基于范围的雾化中，Direct3D 使用从视点到顶点的实际距离来进行雾化计算。Direct3D 随着 2 点之间距离的增加，而非随着场景中顶点的深度，来增加雾化的效果，从而避免了旋转中出现的假象。

如果当前设备支持基于范围的雾化，它设置 Caps结构体的 RasterCaps成员下的SupportsFogRange属性，该属性通过调用 Manager.GetDeviceCaps方法被返回。要启用基于范围的雾化，应将 RenderStateManager.RangeFogEnable属性设置为true。

在变换和光照期间，基于范围的雾化由 Direct3D 计算。没有使用 Direct3D 变换和光照引擎的程序也必须执行它们自身的顶点雾化计算。因此，应为每个顶点提供在镜面反射分量 alpha 分量中的基于范围的雾化因子。

5.5.2 使用顶点雾化

在程序中使用下列步骤启用顶点雾化。

1. 将 RenderStateManager.FogEnable设置为true来启用雾化混合。
2. 在 RenderStateManager.FogColor属性中设置雾化色彩
3. 将 RenderStateManager.FogVertexMode属性设置为 FogMode枚举类型的一个成员来选择所期望的雾化公式。
4. 在渲染状态中，将雾化参数设置为所期望的被选择雾化公式。

下列 C#代码范例用代码描述了这些步骤看上去的样子。

```
[C#]



// For the purposes of this example, device is a valid Device object.
//
void SetupVertexFog(Color color, FogMode mode, bool bUseRange, float
fDensity)
{
    float Start = 0.5f;      // Linear fog distances.
    float End   = 0.8f;

    // Enable fog blending.
    device.RenderState.FogEnable = true;

    // Set the fog color.
    device.RenderState.FogColor = color;

    // Set fog parameters.
    if (mode == FogMode.Linear)
    {
        device.RenderState.FogVertexMode = mode;
        device.RenderState.FogStart = Start;
        device.RenderState.FogEnd = End;
```

```
    }
    else
    {
        device.RenderState.FogVertexMode = mode;
        device.RenderState.FogDensity = fDensity;
    }

    // Enable range-based fog if desired (only supported for
    // vertex fog). For this example, it is assumed that bUseRange
    // is set only if the driver exposes the
    // RasterCaps.SupportsFogRange capability.
    // Note: This is slightly more performance-intensive
    // than non-range-based fog.

    if (bUseRange)
        device.RenderState.RangeFogEnable = true;
}
```

5.6 像素雾化

像素雾化得名于在设备驱动程序上基于每个像素的计算。这与在变换和光照计算期间通过流水线来计算的顶点雾化方式有所不同。像素雾化有时也叫做雾化表，因为一些驱动程序使用一个预先计算好的查找表格来确定雾化因子，该因子使用了每个像素的深度应用于混合计算。它可以应用于任何由 FogMode枚举类型成员所识别的雾化公式。这些公式都是由驱动程序指定完成的。如果驱动程序不支持一个复杂的雾化公式，它会被降为复杂度较低的公式来处理。

5.6.1 眼睛相关的 VS 基于 Z 轴的缓存

为了减轻由于深度缓存中z值不均匀分布带来的雾化相关图形的假象，大多数硬件设备使用了眼睛相关深度来代替基于z值的深度值用于像素雾化。眼睛相关深度本质上是在同一坐标系中设置的w元素。Direct3D从一个设备空间坐标系中取得RHW元素的倒数，以再生真正的w值。如果设备支持眼睛相关雾化，它设置 Caps结构体的成员 RasterCaps的SupportsWFog属性，使其通过调用 Manager.GetDeviceCaps返回。除了引用光栅外，软件设备总是使用z值计算像素雾化效果。

当眼睛相关雾化被支持时，如果提供的投影矩阵在世界空间内产生的z值等于设备空间内的w值，系统自动使用眼睛相关深度而非基于z值的深度。投影矩阵的设置要通过调用Device.SetTransform方法，使用 TransformType.Projection值和传递一个用于描述所需要的矩阵的 Matrix结构体。如果投影矩阵不适合需要，雾化效果就不能被完全应用。生成一个合适矩阵的细节，参看 A W-Friendly Projection Matrix。在 What Is the Projection Transformation?提供的透视投影矩阵可以生成一个合适的投影矩阵。
Direct3D 通常在基于 w 值的深度计算中使用设置好的投影矩阵。结果，程序必须设置一个合适的投影矩阵接收所需要的基于 w 值的特性，即使它没有使用 Direct3D 变换流水线。Direct3D 检查投影矩阵的第四列。如果系数是[0,0,0,1]（作为一个仿射投影），系统使用基于 z 值的深度值进行雾化。因此线性雾化效果的开始和终止距离必须在设备空间中指定，其范围从 0.0（在指向用户的最近点）到 1.0（在最远点）。

5.6.2 使用像素雾化

在程序中使用下列步骤启用像素雾化。
1.将 RenderStateManager.FogEnable设置为true来启用雾化混合。
2.在 RenderStateManager.FogColor 属性中设置所期望的雾化色彩。
3.设置 FogMode枚举类型下对应成员的 RenderStateManager.FogTableMode属性来选择使用的雾化公式。
4.在关联的渲染状态中，将雾化参数设置为所期望被选择的雾化模式。这包含了线性雾化的开始、终止距离和指数雾化模式下的雾化密度。
下列 C#代码范例用代码方式描述了这些步骤看上去的样子。

```
[C#]


// For the purposes of this example, device is a valid Device object.
//
void SetupPixelFog(Color color, FogMode mode)
{
    float Start   = 0.5f;    // For linear mode.
    float End     = 0.8f;    //

    float Density = 0.66f;   // For exponential modes.

    // Enable fog blending.
    device.RenderState.FogEnable = true;
```

```
    // Set the fog color.
    device.RenderState.FogColor = color;

    // Set fog parameters.
    if (mode == FogMode.Linear)
    {
        device.RenderState.FogTableMode = mode;
        device.RenderState.FogStart = Start;
        device.RenderState.FogEnd = End;
    }
    else
    {
        device.RenderState.FogTableMode = mode;
        device.RenderState.FogDensity = Density;
    }
}
```

第6节　Alpha 混合

Alpha 混合用于显示具有透明度或半透明度像素的图像。除红绿蓝色彩通道外，alpha 位图中的每个像素具有一个透明分量，被称为 alpha 通道。它通常包含的比特位数和一个色彩通道相同。如一个 8 位 alpha 通道能表示 256 级透明度，从 0（意味着整个像素是透明的）到 255（意味着整个像素是不透明的）。

色彩定义可以有或没有 alpha 值。没有 alpha 的色彩是 RGB；具有 alpha 的色彩以 ARGB 存储。顶点数据，材质数据和纹理数据常用于给出物体的透明度。帧缓存也常用于产生透明效果。

下面列出了可以使用 alpha 混合产生的一些特殊效果。

顶点 Alpha
材质 Alpha
纹理 Alpha
帧缓存 Alpha
渲染目标 Alpha

下例示范了 alpha 混合的应用。

公告板
云、烟和水汽
火焰、闪光和爆炸

6.1 顶点 Alpha

顶点数据可以提供alpha数据。要启用顶点alpha，应将设备渲染状态从RenderStateManager.DiffuseMaterialSource设置为 ColorSource.Color1，使Direct3D运行时从漫反射色彩而不是材质中获取漫反射值。
Alpha 数据可以加入到一个顶点的一个色彩分量中。下列代码建立一个球形网格的 2 份拷贝，将一个色彩分量加入到顶点定义中，并指定了每个顶点的色彩。这段代码可以加入到由 DirectX 9.0 Visual C# Wizard 生成的 GraphicsClass.InitializeDeviceObjects 方法中进行功能示范。

```
            [C#]



// Create the sphere geometry.

rearSphere = Mesh.Sphere(device,1f,10,10);



// Copy the sphere, adding a diffuse color to the vertices.

sphere =
rearSphere.Clone(MeshFlags.Managed,VertexFormats.PositionNormal |
VertexFormats.Diffuse,device);



// Add the same color componant to the original sphere.

rearSphere =
rearSphere.Clone(MeshFlags.Managed,VertexFormats.PositionNormal |
VertexFormats.Diffuse,device);



// Lock the vertex buffer of the rearSphere to obtain the vertex information.

CustomVertex.PositionNormalColored[] Source =


(CustomVertex.PositionNormalColored[])rearSphere.LockVertexBuffer(typeof(
CustomVertex.PositionNormalColored),

                    0,rearSphere.NumberVertices);



// Loop through the vertices, adding a red transparent color to each.

for(int x=0; x < (Source.GetLength(0)); x++)

{
```

```
    Source[x].Color = Color.FromArgb(0, 255, 0, 255).ToArgb();
}
rearSphere.UnlockVertexBuffer();


// Add a color component with alpha to the second sphere.
CustomVertex.PositionNormalColored[] Destination =

(CustomVertex.PositionNormalColored[])sphere.LockVertexBuffer(typeof(Cust
omVertex.PositionNormalColored),
                0,rearSphere.NumberVertices);


// Instead of adding the same alpha value to each of the vertices,
// this code increments the value, creating a gradient effect across the sphere.
int num = 256/Destination.GetLength(0);
for(int x=0; x < (Destination.GetLength(0)); x++)
{
    Destination[x].Color = Color.FromArgb(150, 255, 255, num*x).ToArgb();
}
sphere.UnlockVertexBuffer();
```

已被加入到顶点中的色彩alpha值现在 可以用于绘制具有透明度的球体。下列代码设立了一个渲染设备并在rearSphere 前方绘制球体来表示透明感。这段代码可以加入到由DirectX 9.0 Visual C# Wizard生成的GraphicsClass.Render 方法中进行功能示范。将这段代码放置在 Device.Transform.Projection矩阵设置之后。

```
            [C#]


// A StateBlock is used to reset the device to its former state after we draw
transparency.
StateBlock sb = new StateBlock(device, StateBlockType.All);
sb.Capture();
```

```
// Use the diffuse vertex color as the DiffuseMaterialSource for color lighting calculations
// (as opposed to a material).
device.RenderState.DiffuseMaterialSource = ColorSource.Color1;

// Set up the device to render the alpha information.
device.RenderState.AlphaBlendEnable = true;
device.RenderState.SourceBlend = Blend.SourceColor;
device.RenderState.DestinationBlend = Blend.InvSourceAlpha;

// Translate the rearSphere so that it is drawn behind the sphere.
Matrix m2 = m;
m.Translate(0,0,2);
device.Transform.World = m;

rearSphere.DrawSubset(0);

// Rotate the sphere so we can see the gradient a little better.
device.Transform.World = m2;
Vector3 rot = new Vector3(90,0,0);
m2.RotateAxis(rot,-45);
device.Transform.World = m2;

sphere.DrawSubset(0);

// Restore the device to its original settings.
sb.Apply();
```

这段代码结果产生了 2 个透明球体，其中 1 个位于另 1 个的稍前方。

6.2 材质 Alpha

材质也可提供 alpha 数据。要启用材质 alpha，设置漫反射材质渲染状态让运行时使用材质漫反射色彩分量而非顶点漫反射色彩分量，如下面 C#代码所示。

```
[C#]



// Enable material alpha.
device.RenderState.DiffuseMaterialSource = ColorSource.Material;
```

接下来的 C#代码用一个 alpha 值材质初始化材质，并且在绘制前设置材质。

```
[C#]



Material mtrl = new Material();
Color c = Color.FromArgb(127, 255, 0, 0);

mtrl.Diffuse = c;
mtrl.Ambient = c;
mtrl.Specular = c;
mtrl.Emissive = c;

device.Material = mtrl;
```

6.3 纹理 Alpha

纹理也可以提供 alpha 数据。首先必须建立纹理，然后加入 alpha 值。要渲染这个纹理，应将其设置为一个纹理场景并选择合适的纹理场景操作和操作数。当调用绘制命令时，造型以具有透明度的方式被渲染。
下列 C#代码范例描述了这些步骤。

```
[C#]



// Declare a Device.
```

```
Device device = null


// Declare a global texture variable.
Texture texture = null;


// Initialize the device.
.
.
.


// Create an alpha texture.
texture = new Texture(device, 128, 128, 0, 0, Format.A8R8G8B8,
Pool.Managed);


LoadGradient();
```

通过建立一个空纹理，Texture纹理类设立好了全部。在纹理资源建立后，调用LoadGradient加载alpha通道。

```
        [C#]


protected void LoadGradient()
{
    uint yGrad, xGrad;
    uint dx = 128;       // width
    uint dy = 128;       // height


    uint[] buffer = new uint[dy * dx];


    GraphicsStream gs = null;
```

```
try
{
    gs = texture.LockRectangle(0, LockFlags.Discard);

    for (uint y = 0; y < dy; y++)
    {
        uint offset = y * dx;
        yGrad = (uint)(((float)y / (float)dx) * 255.0f);

        for (uint x = 0; x < dx;)
        {
            xGrad = (uint)(((float)x / (float)dx) * 255.0f);

            uint b = (uint)(xGrad + (255 - yGrad)) / 2 & 0xFF;
            uint g = (uint)((255 - xGrad) + yGrad) / 2 & 0xFF;
            uint r = (uint)(xGrad + yGrad) / 2 & 0xFF;
            uint a = (uint)(xGrad + yGrad) / 2 & 0xFF;

            buffer[offset + x] = ((a << 24) + (r << 16) + (g << 8) + (b));
            x++;
        }
    }

    gs.Write(buffer);
    texture.UnlockRectangle(0);
}
catch(InvalidCallException)
{
    return;
```

```
    }
}
```

Alpha 值的计算基于在纹理尺寸内的当前像素相对 x/y 坐标位置。
下一步，将纹理赋予一个纹理场景并设立纹理场景。

```
            [C#]



 // Assign texture.
device.SetTexture(0, texture)


// Texture stage states.
device.TextureState[0].ColorOperation = TextureOperation.Modulate;
device.TextureState[0].ColorArgument1 = TextureArgument.TextureColor;
device.TextureState[0].ColorArgument2 = TextureArgument.Diffuse;


device.TextureState[0].AlphaOperation = TextureOperation.Modulate;
device.TextureState[0].AlphaArgument1 = TextureArgument.TextureColor;
device.TextureState[0].AlphaArgument2 = TextureArgument.Diffuse;
```

这段代码结果将产生一个具有梯度透明的造型。梯度范围是从透明（所在位置 x=0）到不透明（所在位置 x 具有最大值）。

6.4 帧缓存 Alpha

帧缓存 alpha 混合和顶点 alpha，材质 alpha，纹理 alpha 有点不同。顶点，材质和纹理 alpha 设置透明值仅用于当前造型，并不会影响其他的造型。帧缓存 alpha 混合控制当前造型如何与帧缓存中的已存在像素相结合，形成一个最终的像素色彩。
当执行 alpha 混合时，2 种色彩被结合：源色彩和目标色彩。源色彩来源于透明物体，目标色彩来源于像素位置的已有色彩。目标色彩是透明物体之后的一些其他物体的渲染结果；这就是说，该物体将通过透明物体可见。Alpha 混合使用下面的公式控制源物体和目标物体之间的比率。

```
Final Color = ObjectColor * SourceBlendFactor
                    + PixelColor * DestinationBlendFactor
```

ObjectColor 来源于当前像素位置的被渲染造型。PixelColor 来源于当前像素位置的帧缓

存。
可以使用下列一组混合因子。

混合模式因子	描述
Zero	(0, 0, 0, 0)
One	(1, 1, 1, 1)
SourceColor	(R_s,G_s,B_s,A_s)
InvSourceColor	(1 - R_s, 1 - G_s, 1 - B_s, 1 - A_s)
SourceAlpha	(A_s, A_s, A_s, A_s)
InvSourceAlpha	(1 - A_s, 1 - A_s, 1 - A_s, 1 - A_s)
DestinationAlpha	(A_d, A_d, A_d, A_d)
InvDestinationAlpha	(1 - A_d, 1 - A_d, 1 - A_d, 1 - A_d)
DestinationColor	(R_d, G_d, B_d, A_d)
InvDestinationColor	(1 - R_d, 1 - G_d, 1 - B_d, 1 - A_d)
SourceAlphaSat	(f, f, f, 1); f = min(A_s, 1 - A_d)
BothSourceAlpha	Obsolete. To achieve the same effect, set the source and destination blend factors to SourceAlpha and InvSourceAlpha in separate calls.
BothInvSourceAlpha	Obsolete. To achieve the same effect, set the source and destination blend factors to SourceAlpha and InvSourceAlpha in separate calls. Source blend factor is (1 - A_s, 1 - A_s, 1 - A_s, 1 - A_s), and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden. This blend mode is supported only if the RenderStateManager.SourceBlend property is set to **true**.
BlendFactor	Constant color blending factor used by the frame-buffer blender. Uses color.r, color.g, color.b, and color.a obtained from RenderStateManager.BlendFactor. This blend mode is supported only if the BlendCaps.SupportsBlendFactor property is set to **true**.
InvBlendFactor	Inverted constant color blending factor used by the frame-buffer blender. Uses 1-color.r, 1-color.g, 1-color.b, and 1-color.a obtained from RenderStateManager.BlendFactor. This blend mode is supported only if the BlendCaps.SupportsBlendFactor property is set to **true**.

Direct3D使用 RenderStateManager.AlphaBlendEnable允许alpha透明混合。alpha混合的类型取决于 RenderStateManager.SourceBlend 和RenderStateManager.DestBlend渲染状态。源混合状态和目标混合状态配对使用。下面C#代码片断将源混合状态设置为 Blend.SourceColor，目标混合状态设置为Blend.InvSourceColor。

```
            [C#]



// This code fragment assumes that device is a

//    valid Device object.
```

```
// Enable alpha blending.
device.RenderState.AlphaBlendEnable = true;

// Set the source blend state.
device.RenderState.SourceBlend = Blend.SourceColor;

// Set the destination blend state.
device.RenderState.DestBlend = Blend.InvSourceColor;
```

这段代码在源色彩（造型在当前位置被渲染的色彩）和目标色彩（当前位置帧缓存中的色彩）之间执行 1 个线性混合。结果看上去有点儿类似于目标对象的色彩通过源对象穿过有色玻璃的样子；剩余的部分看上去被吸收了。
这许多混合因子需要纹理中的 alpha 值来正确操作。因此当使用顶点或材质 alpha 时，程序将受到模式产生有趣效果的限制。

6.5 渲染目标 Alpha

这种控制，可以使用一个新的渲染状态，RenderStateManager.SeparateAlphaBlendEnabled。
当 RenderStateManager.SeparateAlphaBlendEnabled设置为false（缺省条件）时，那些同样为混合色彩通道而定义的alpha值，将被应用渲染目标混合因子和操作。驱动程序需要设置 MiscCaps.SupportsSeparateAlphaBlend以指明它可以支持渲染目标alpha混合。一定要设置 RenderStateManager.AlphaBlendEnable告诉流水线需要进行alpha混合。
下面定义了新的渲染状态，以控制渲染目标混合器 alpha 通道中的因子。

RenderStateManager.AlphaDestinationBlend
RenderStateManager.AlphaSourceBlend

链接RenderStateManager.SourceBlend和RenderStateManager.DestinationBlend一样，这些状态可以在列举的 Blend值中设置为其中一种。依靠 BlendCaps对象内嵌的Caps.DestinationBlendCaps和 Caps.SourceBlendCaps设置，源和目标混合设置能以多种方式组合。
Alpha 混合如下所示完成：
renderTargetAlpha = ($alpha_{in}$ * srcBlendOp) BlendOp ($alpha_{rt}$ * destBlendOp)

$alpha_{in}$ 是输入 alpha 值.
srcBlendOp 是 Blend中的 1 个混合因子
BlendOp是 Blend中的 1 个混合因子
$alpha_{rt}$ 是渲染目标的 alpha 值
destBlendOp 是 Blend中的 1 个混合因子
renderTargetAlpha 是最后混合过的 alpha 值.

6.6 公告板

当建立三维场景时，程序有时在渲染看上去像三维的二维物体时具有更高的性能。这种基本观点源于公告板技术。

对公告板在世界中的正常理解就是沿路上的标记。Direct3D 程序可以通过定义一个应用纹理的矩形固体来渲染公告板的这种形式。在更多三维图形中的专用场景中，公告板是这种形式的扩展。它目的是建立看上去像三维的二维物体，这种技术将包含了物体图像的纹理应用到一个矩形造型上。造型被旋转使其总是面向用户。物体的图像是否矩形无关紧要。公告板的一部分可以被透明化，因此你不想看见的那部分公告板图像是不可视的。

许多游戏使用公告板作为动画精灵。比如，当用户在三维迷宫中穿梭时，他或她也许看见了可以捡取武器或奖金。这些就是典型的矩形造型上的二维纹理图像。公告板经常被游戏用于渲染树木、灌木和云的图像。

当图像应用于公告板时，矩形造型首先必须旋转使图像面向用户。然后程序必须把它转化为坐标位置，之后可以将纹理应用于造型。

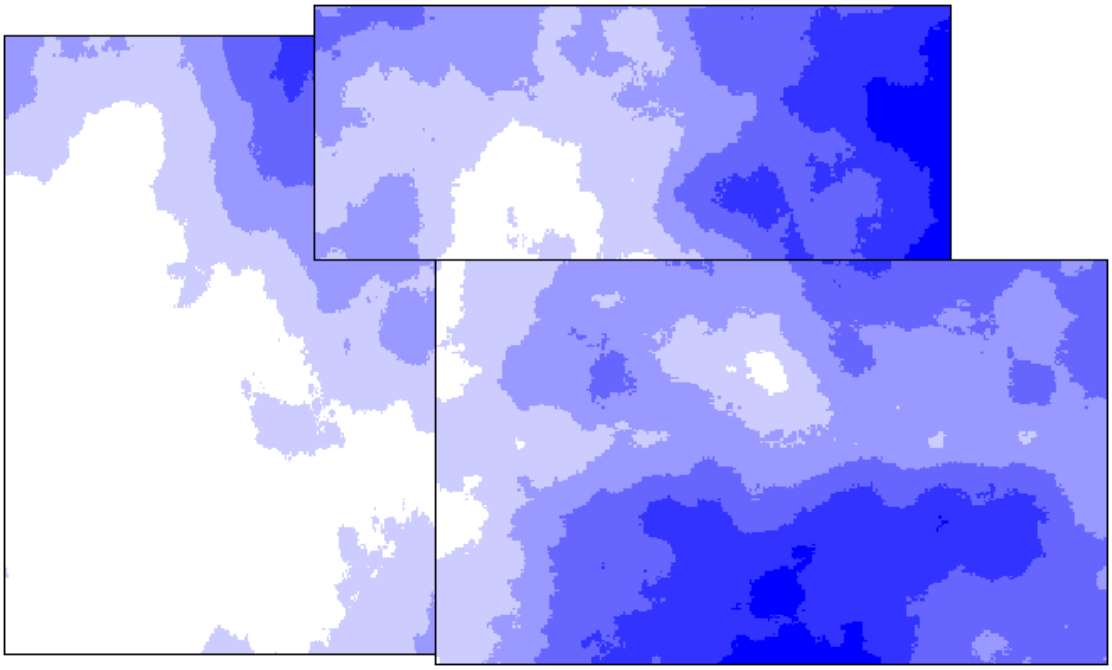
对于对称物体，公告板工作得最好，尤其是沿垂直轴线对称的物体。它还要求视点高度不能增加太多。如果用户可以从上方观察公告板对象，就可以明显看出它是 2-D 而非 3-D 的。

6.7 云、烟和水汽

云、烟雾和水汽都可以通过公告板技术的扩展形式建立（参看 Billboarding）。通过将公告板的单轴旋转改为双轴，程序可以让用户从任何角度观察公告板。通常，程序在水平和垂直轴线方向上旋转公告板。

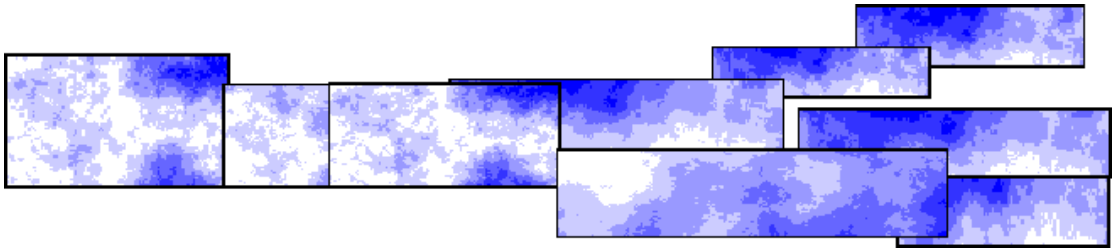
程序可以绕单轴或双轴旋转一个矩形造型使其面向用户来建立一个简单的云层。然后可以将一个透明云状纹理应用到造型上。通过应用一系列纹理，云就被逐渐展现出来。

程序可以通过一组造型来建立更多复杂的云。云的每部分是一个矩形造型。这些造型可以随时间而独立移动看上去像一层动态的薄雾。下图描绘这个想法。



烟雾的外观和云的显示方式类似。它和复杂的云一样，通常需要多个公告板。烟雾通常会随

时间变化而翻腾升涨，因此构成烟雾羽片的公告板需要相应的移动。由于羽片的升涨和消散，也许需要添加更多的公告板。
水汽是不升涨的烟雾羽片。然而，和烟雾羽片一样，它随时间消散。下图描绘了使用公告板模拟水汽的技术。



6.8 火焰、闪光和爆炸

Direct3D 惯于模拟诸如能量释放之类的自然现象。比如程序可以将火焰状纹理应用到一组公告板来产生火焰的外观。如果程序使用一个火焰状纹理序列来描绘火焰中每块公告板上的火焰，这尤其有效。改变公告板到公告板的动画回放速度会增加火焰外观的真实感。通过放置公告板和公告板上的纹理，可以产生出混合三维火焰的外观。
对场景中所有造型应用连续不断加亮的光线贴图，可以模拟出闪耀和闪光。虽然这种技术的计算开销很大，但它让程序能够模拟出局部的闪耀或闪光。就是说，闪耀或闪光发源的部分场景优先变亮。
另一种技术是在场景前方安排一个公告板，覆盖整个渲染目标区域。程序在公告板上应用连续不断的白色纹理并随时间变化降低透明度。随时间流逝，整个场景趋向白色。这是一种建立闪光的低开销方法。然而，使用这种技术难以生成由一个单独点光源发出明亮闪光的外观效果。
在三维场景中显示爆炸的程序和火焰、闪耀、闪光类似。例如，当爆炸发生时，程序可以使用一个公告板显示冲击波和上涨的烟雾羽片。与此同时，程序可以使用一组公告板模拟火光。另外它还能够决定场景前方的一个单独公告板的位置，给整个场景加上一个闪光。
使用公告板可以模拟能量射线束。程序也可以使用被定义为线列表或线条带的造型来显示它们。更多信息参看 Line Lists和 Line Strips。
程序可以使用公告板或被定义为三角形列表的造型来建立力场。要从三角形列表建立一个力场，应在一个三角形列表中定义一组互不相连的三角形，它们将由力场覆盖的区域平等分隔。在看向力场时，也许和期望的一样，三角形之间的缝隙让用户可以看见三角形后面的场景。给三角形赋予炽热能量外观的三角形应用纹理。更多信息参看 Triangle Lists。

第6章　Direct3D 表面

一个表面代表了一个显示内存的线性区域。表面通常位于显示卡的显存中，虽然它们也可以存在于系统内存中。它们由 Surface类托管。
熟悉下列术语对于理解表面是必须的。

front buffer 前台缓存。内存中的一个可以通过图形适配器被翻译并在显示器上显示的矩形。在 Direct3D 中，程序从不能直接对前台缓存进行写入。

back buffer 后台缓存。内存中的一个程序可以直接写入的矩形。后台缓存从不能直接显示在显示器上。

flipping 翻转表面。将后台缓存移动至前台缓存的处理过程。

swap chain 交换链表。可以连续出现在前台缓存的 1 个或多个后台缓存的集合。

获得 1 个表面

要建立一个表面，应调用以下任一种方法。

Device.CreateDepthStencilSurface
Device.CreateOffscreenPlainSurface
Device.CreateRenderTarget

表面格式决定了表面内存中的每个像素数据如何插值。Direct3D使用 SurfaceDescription 结构体的 Format属性来描述表面格式。1 个已存在表面的 SurfaceDescription结构体可以通过它的 Surface.Description属性找回。

一旦建立了 1 个表面，通过调用下列任一种方法可以获得 1 个指针。

CubeTexture.GetCubeMapSurface
Device.GetBackBuffer
Device.DepthStencilSurface
Device.GetFrontBufferData
Device.GetRenderTarget
SwapChain.GetBackBuffer
Texture.GetSurfaceLevel

Surface类允许通过 UpdateSurface方法对内存进行直接访问。该方法允许像素矩形区域从一个 UpdateSurface类复制到另一个类。UpdateSurface类还有方法直接访问显存。如 Surface.LockRectangle方法常用于锁定显存的一块矩形区域。重要的是在使用过表面的锁定矩形区域后，要调用 Surface.LockRectangle。

附加表面主题

下列主题提供了关于如何使用表面的信息。

表面格式
什么是 1 个交换链表
宽度 vs 间距
翻转表面
页面翻转和后台缓冲
复制到表面
复制表面
直接访问表面内存
私有表面数据
伽玛控制

第1节　表面格式

在 Direct3D 中，所有的 2-D 图像是由 1 个称作表面的线性范围内存所表示的。1 个表面可以看成 1 个 2-D 数组，其中每个元素包含了 1 个像素，它表示图像一小部分的色彩值。1 幅图像的细节层次由表达图像所需要的像素数量和表达图像色彩频谱的位数来定义。例如 1 幅 800×600×32 位色深的图像具有比 1 幅 640×480×16 位色深的图像更多的细节。同样，更多细节的图像需要更大的表面来存储数据。1 幅 800×600×32 位色深的图像，表面的数组维数是 800×600，每个元素包含了 1 个 32 位值表示它的色彩。

所有表面具有 1 个尺寸大小和 1 个用来表示色彩的指定数量的位数。这些位被划分为几个

色彩元素：红绿蓝。在托管代码的DirectX 9.0 中，所有色彩元素通过 Format枚举型定义。1 个色彩格式按照每种色彩的保留位数被分解。比如，1 个 16 位色彩格式被定义为 Format.R5G6B5，保留 5 位用于红色（R），保留 6 位用于绿色（G），5 位用于蓝色（B）。所有格式从左到右列出，从最高有效位(MSB)到最低有效位(LSB)。比如 ARGB 是按照 MSB A(alpha)通道到 LSB B（蓝色 ）通道顺序排列的。在表面数据移动期间，数据在内存中是从 LSB 到 MSB 存储的，这意味着内存中的通道顺序是从 LSB（蓝色）到 MSB（alpha）。对于包含了未定义通道的格式来说缺省值为 1。唯一例外是 A8 格式，它将 3 个色彩通道初始化为 000。

1.1 无符号格式

在无符号格式中的数据必须是正数。无符号格式使用了红色（R），绿色（G），蓝色（B），透明度（Alpha），亮度（L）和调色板（P）数据的组合。因为调色板数据常用于索引 1 个色彩调色板，所以它作为色彩索引数据被引用。

无符号格式标识	格式
R8G8B8	A 24-bit RGB pixel format that uses 8 bits per channel.
A8R8G8B8	A 32-bit ARGB pixel format, with alpha, that uses 8 bits per channel.
X8R8G8B8	A 32-bit RGB pixel format in which 8 bits are reserved for each color.
R5G6B5	A 16-bit RGB pixel format that uses 5 bits for red, 6 bits for green, and 5 bits for blue.
X1R5G5B5	A 16-bit pixel format in which 5 bits are reserved for each color.
A1R5G5B5	A 16-bit pixel format in which 5 bits are reserved for each color and 1 bit is reserved for alpha.
A4R4G4B4	A 16-bit ARGB pixel format that uses 4 bits for each channel.
R3G3B2	An 8-bit RGB texture format that uses 3 bits for red, 3 bits for green, and 2 bits for blue.
ii	An 8-bit format that uses alpha only.
A8R3G3B2	A 16-bit ARGB texture format that uses 8 bits for alpha, 3 bits each for red and green, and 2 bits for blue.
X4R4G4B4	A 16-bit RGB pixel format that uses 4 bits for each color.
A2B10G10R10	A 32-bit pixel format that uses 10 bits for each color and 2 bits for alpha.
A8B8G8R8	A 32-bit ARGB pixel format, with alpha, that uses 8 bits per channel.
X8B8G8R8	A 32-bit RGB pixel format in which 8 bits are reserved for each color.
G16R16	A 32-bit pixel format that uses 16 bits each for green and red.
A2R10G10B10	A 32-bit pixel format that uses 10 bits each for red, green, and blue, and 2 bits for alpha.
A16B16G16R16	A 64-bit pixel format that uses 16 bits for each component.
A8P8	An 8-bit color format indexed with 8 bits of alpha.
P8	An 8-bit color indexed format.

L8	An 8-bit luminance-only format.
L16	A 16-bit luminance-only format.
A8L8	A 16-bit format that uses 8 bits each for alpha and luminance.
A4L4	An 8-bit format that uses 4 bits each for alpha and luminance.

1.2 有符号格式

有符号格式中的数据可以为正数或负数。标记格式使用（U）（V）（W）（Q）数据的组合。

有符号格式标识	格式
V8U8	A 16-bit **bump-map** format that uses 8 bits each for U and V data.
Q8W8V8U8	A 32-bit bump-map format that uses 8 bits for each channel.
V16U16	A 32-bit bump-map format that uses 16 bits for each channel.
Q16W16V16U16	A 64-bit bump-map format that uses 16 bits for each component.
CxV8U8	A 16-bit normal compression format. The texture sampler computes the C channel from C = sqrt(1 - U^2 - V^2).

1.3 混合格式

混合格式中的数据可以包含有符号和无符号数据的组合。

混合格式标识	格式
L6V5U5	A 16-bit bump-map format, with luminance, that uses 6 bits for luminance and 5 bits each for v and u data.
X8L8V8U8	A 32-bit bump-map format, with luminance, that uses 8 bits for each channel.
A2W10V10U10	A 32-bit bump-map format that uses 2 bits for alpha and 10 bits each for w, v, and u data.

1.4 四字符编码格式

Data in a four-character code (FOURCC) format is compressed.
在四-字符编码(FOURCC)格式中的数据被压缩了。

1.5 MAKEFOURCC

下列格式的枚举值是基于 1 个由多个正数值组合产生的数。这允许硬件设计者使用这里没有显式列出以指明 FOURCC 格式的自定义枚举值。
下列 C#代码片断产生 FOURCC 代码。

```
[C#]
```

```
static int MakeFourCC(int ch0, int ch1, int ch2, int ch3)

{

    return ((int)(byte)(ch0)|((int)(byte)(ch1) << 8)| ((int)(byte)(ch2) << 16) |
((int)(byte)(ch3) << 24));

}
```

定义的 FOURCC 格式如下所示。

四字符编码格式标识	值	格式
Multi2Argb8	MakeFourCC('M','E','T','1')	MultiElement texture (not compressed).
G8R8G8B8	MakeFourCC('G', 'R', 'G', 'B')	A 16-bit packed RGB format that is analogous to YUY2 (Y0U0, Y1V0, Y2U2, etc.). It requires a pixel pair to properly represent the color value. The first pixel in the pair contains 8 bits of green (in the high 8 bits) and 8 bits of red (in the low 8 bits). The second pixel contains 8 bits of green (in the high 8 bits) and 8 bits of blue (in the low 8 bits). The two pixels share the red and blue components, while each has a unique green component (G0R0, G1B0, G2R2, etc.). When looking up into a pixel shader, the texture sampler does not normalize the colors; they remain in the range of 0.0f to 255.0f. This is true for all programmable pixel shader models. For the fixed-function pixel shader, the hardware should normalize to the 0.f to 1.f range and treat it as the YUY2 texture. Hardware that exposes this format must have the Caps.PixelShader1xMaxValue member set to a value capable of handling that range.
R8G8B8G8	MakeFourCC('R', 'G', 'B', 'G')	A 16-bit packed RGB format that is analogous to UYVY (U0Y0, V0Y1, U2Y2, etc.). It requires a pixel pair ito properly represent the color value. The first pixel in the pair contains 8 bits of green (in the low 8 bits) and 8 bits of red (in the high 8 bits). The second pixel contains 8 bits of green (in the low 8 bits) and 8 bits of blue (in the high 8 bits). The two pixels share the red and blue components, while each has a unique green component (R0G0, B0G1, R2G2, etc.). When looking up into a pixel

		shader, the texture sampler does not normalize the colors; they remain in the range of 0.0f to 255.0f. This is true for all programmable pixel shader models. For the fixed-function pixel shader, the hardware should normalize to the 0.f to 1.f range and treat it as the YUY2 texture. Hardware that exposes this format must have the Caps.PixelShader1xMaxValue member set to a value capable of handling that range.
Dxt1	MakeFourCC('D', 'X', 'T', '1')	DXT1 compression texture format.
Dxt2	MakeFourCC('D', 'X', 'T', '2')	DXT2 compression texture format.
Dxt3	MakeFourCC('D', 'X', 'T', '3')	DXT3 compression texture format.
DXT4	MakeFourCC('D', 'X', 'T', '4')	DXT4 compression texture format.
Dxt5	MakeFourCC('D', 'X', 'T', '5')	DXT5 compression texture format.
Uyvy	MakeFourCC('U', 'Y', 'V', 'Y')	UYVY format (PC98 compliance).
Yuy2	MakeFourCC('Y', 'U', 'Y', '2')	YUY2 format (PC98 compliance).

1.6 Buffer 缓冲器格式

深度，模版，顶点和索引 buffer 缓存器，每个都具有一个唯一格式。

缓存标识	格式
D16Lockable	A 16-bit z-buffer bit depth.
D32	A 32-bit z-buffer bit depth.
D15S1	A 16-bit z-buffer bit depth in which 15 bits are reserved for the depth channel and 1 bit is reserved for the stencil channel.
D24S8	A 32-bit z-buffer bit depth that uses 24 bits for the depth channel and 8 bits for the stencil channel.
D24X8	A 32-bit z-buffer bit depth that uses 24 bits for the depth channel.
D24X4S4	A 32-bit z-buffer bit depth that uses 24 bits for the depth channel and 4 bits for the stencil channel.
D32SingleLockable	A lockable format in which the depth value is represented as a standard Institute of Electrical and Electronics Engineers (IEEE) floating-point number.
D24SingleS8	A nonlockable format that contains 24 bits of depth (in a 24-bit floating-point format - 20e4) and 8 bits of stencil.
D16	A 16-bit z-buffer bit depth.

VertexData	Describes a vertex buffer surface.

除 D16Lockable 外，所有的深度模版格式指明每个像素都没有特殊的位顺序，并允许驱动程序消耗比每深度通道指定的更多位数（而不是每模版通道）。

1.7 浮点格式

这些标识用于浮点表面格式。它们使用 16 位/通道并作为 s10e5 格式。

浮点标识	格式
R16F	A 16-bit floating-point format that uses 16 bits for the red channel.
G16R16F	A 32-bit floating-point format that uses 16 bits each for the red and green channels.
A16B16G16R16F	A 64-bit floating-point format that uses 16 bits for each channel (alpha, blue, green, and red).

1.8 IEEE 格式

这些标记用于浮点表面格式。它们使用 32 位/通道并作为 s23e8 格式。

浮点标识	格式
R32F	A 32-bit floating-point format that uses 32 bits for the red channel.
G32R32F	A 64-bit floating-point format that uses 32 bits each for the red and green channels.
A32B32G32R32F	A 128-bit floating-point format that uses 32 bits for each channel (alpha, blue, green, and red).

1.9 其他

这个标记用于未定义格式。

其他标识	格式
Unknown	Surface format is unknown.

1.10 后台缓存或显示格式

这些格式是后台缓存或显示的仅有合法格式。

格式	后台缓存	显示
A2R10G10B10	X	x (full-screen mode only)
A8R8G8B8	X	
X8R8G8B8	X	X
A1R5G5B5	X	
X1R5G5B5	x	X
R5G6B5	x	X

1.11 备注

位顺序以最高有效位开始，因此 A8L8 指明这个 2 字节格式的高位是 alpha。D16 格式指明了 1 个 16 位整数值和 1 个程序可锁定表面。
像素格式可以启用由硬件生产商定义的扩展格式表达，并包含确定好的FOURCC方法。这套由Direct3D运行时所理解的格式通过 Caps结构体定义。
注意格式是由独立硬件生产商（IHVs ）供应的，许多 FOURCC 代码没有列出。因为枚举的这些格式可以被运行时所支持，所以它们是唯一的。这意味着可以对所有这些类型进行引用光栅操作。IHVs 提供的格式由独立硬件生产商来支持，而这是基于该图形卡的。

1.12 相关主题

Format Enumeration

第2节　什么是 1 个交换链表？

图形适配器包含了 1 个指向表面的指针，该表面被称为前台缓存，它表示了显示器上显示的图像。当显示器刷新时，图形卡将前台缓存的内容发送到显示器上显示。然而，当选渲染实时图形时这会导致 1 个问题。问题的核心是显示器刷新率和计算机的相比非常慢。常用刷新率范围从 60Hz（60 次/秒）到 100Hz。当显示器刷新到中间时，如果程序正在更新前台缓存，显示的图像就会被切成两半。显示出来的上半部分包含了旧图像，而下半部分包含了新图像。这种问题被称为撕裂。
Direct3D 通过 2 种方式避免撕裂发生：1 种是允许显示器仅在垂直方向进行扫描（或垂直同步）操作，还有 1 种是称为后台缓冲的技术。显示器通过阴极射线管（CRT）的电子束 Z 字形扫描来刷新图像。扫描起始于显示器的左上角，终止于右下角。当扫描束到达底部时，显示器将它移回至左上角重新校准电子束，使得扫描过程能够重新进行。这个重新校准过程称为 1 个垂直同步。在垂直同步过程期间，显示器不绘制任何东西。因此任何对前台缓存的更新在显示器再次开始绘制前，都无法看见。垂直同步相对较慢，但还不足以慢到等待渲染 1 个复杂场景。避免撕裂现象和渲染复杂场景所需要的是后台缓冲处理。
后台缓冲是将 1 个场景绘制到 1 个被称为后台缓存的离屏表面上的处理过程。注意除了前台表面以外的任何表面都称为离屏表面，因为不能从显示器上直接看见它。通过使用后台缓存，只要系统处于空闲时程序就可以自由渲染场景（就是说没有窗口信息处于等待）而不必考虑显示器的刷新率。后台缓冲给如何及何时将后台缓存移动到前台缓存带来了额外的复杂因素。
将后台缓存移动到前台缓存的过程称为表面翻转。记住图形卡仅使用 1 个指向表面的指针来表示前台表面。这是因为 1 个简单的指针改变是将后台缓存设置为前台缓存所需要的全部操作。当程序请求 Direct3D 将后台缓存显示为前台缓存时，Direct3D 仅仅交换 2 个表面的指针。结果后台缓存现在成了新的前台缓存，旧的前台缓存成了新的后台缓存。只要在程序请求 Direct3D 设备显示后台缓存时，就调用了 1 个表面翻转；然而，Direct3D 可以设置为列队响应请求直到垂直同步发生。这个选项作为 Direct3D 设备的显示间隔。注意新的后台缓存中的数据也许不可再使用，这取决于程序如何指定 Direct3D 应该如何处理表面翻转。表面翻转是多媒体，动画和游戏软件的关键。这相当于通过 1 沓纸完成的动画方式。画家在每张纸上稍稍改变外形轮廓，使得你在快速翻动纸张时图案呈现出动画效果。

相关主题

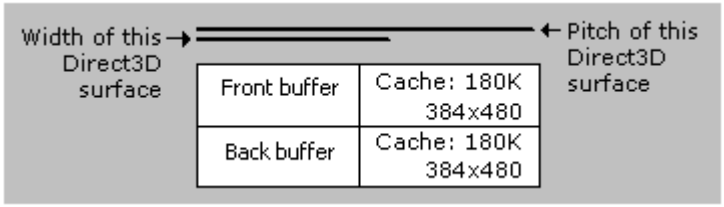
Flipping Surfaces

第3节　宽度 VS 间距

虽然术语宽度和间距经常交换使用，但它们的意义有明显不同。理解这些含义及如何解释 Direct3D 用于描述它们的值是非常重要的。

Direct3D使用 SurfaceDescription结构体装载描述 1 个表面的信息.在多数情形下，该结构被定义为：包含了关于表面维数及那些维数在内存中是如何表达的的信息。该结构使用了 Height和 Width成员来描述表面的逻辑维数。像素中的这 2 个成员都将被测量。因此，1 个 640×480 表面的 Height和 Width值是相同的，无论该表面是 8 位或 24 位的RGB。

当使用 Surface.LockRectangle方法锁定 1 个表面时，该方法返回 1 个 GraphicsStream 对象或 1 个填充了表面数据的 Array数组，这取决于该方法的重载调用。表面的间距由pitch参数传回。由pitch参数中返回的该值描述了表面的内存间距，也称为步幅。Pitch表示了 2 幅相邻位图的首行内存地址之间距离的字节数。因为pitch是用字节而不是用像素测量的，1 幅 640×480×8 的表面和 1 个维数相同但像素格式不同的表面具有完全不同的pitch值。另外，pitch值有时也被Direct3D保留作为高速缓存，因此假设pitch仅由宽度乘以每像素的字节数是不安全的。宁可形象化的用下图来表示宽度和间距之间的不同。



本图中，前台缓存和后台缓存都是 640×480×8，并且高速缓存是 384x480x8。

直接访问表面时，要注意停留在为表面维数分配的内存中并置于为高速缓存保留的任何内存之外。还有仅锁定 1 个表面的一部分时，你必须在锁定期间停留在指定的矩形中。不遵循这些指导方针的话将会导致不可预期的结果。当直接在表面内存中渲染时，总应使用 Surface.LockRectangle 方法返回的 pitch。不要采用单独基于显示模式的 pitch。如果程序在某些适配器上可以工作而在另一些适配器上看上去工作异常，这就有可能是该问题所导致的。

相关主题

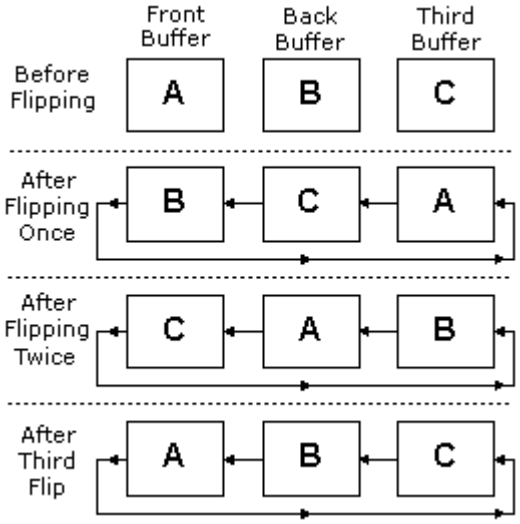
Accessing Surface Memory Directly

第4节　翻转表面

Direct3D 程序通常通过在后台缓存中生成动画的帧并按顺序显示它们的方式来显示一个动画序列。后台缓存被组织为交换链表。交换链表是一系列连续翻转到屏幕上的缓存。交换链表常用于渲染内存中的场景，当渲染结束后把场景翻转到屏幕上。它阻止了一种被称为撕裂的现象产生，并可以生成更平滑的动画。

在Direct3D中建立的每个设备至少具有 1 个交换链表。当你初始化第一个Direct3D设备时，你设置了 PresentParameters结构体的 BackBufferCount成员，该成员告诉了Direct3D将要在交换链表中使用的后台缓存数量。然后 Device构造函数创建Direct3D设

备和相应的交换链表。
当你使用 Device.Present方法请求一个表面翻转操作时，用于前台缓存和后台缓存的表面内存指针相互交换。翻转操作是通过交换显示设备用于引用内存的指针执行的，而不是通过复制表面内存。当翻转链表包含了 1 个前台缓存和多于 1 个后台缓存时，指针按照如下图所示的环形顺序进行交换。



调用 SwapChain构造器可以建立设备的额外交换链表。程序为每个视角建立 1 个交换链表并将每个交换链表与一个特定的窗口关联。程序渲染每个交换链表的后台缓存中的图像，然后分别显示它们。SwapChain带来的 2 个参数分别对 1 个 PresentParameters对象和 1 个合法的 Device对象进行引用。SwapChain.Present方法常用于显示前台缓存的下一个后台缓存中的内容。注意 1 个设备仅能有 1 个全屏交换链表。
要获得对 1 个指定后台缓存的访问，应调用 Device.GetBackBuffer 或 SwapChain.GetBackBuffer方法。该方法将返回 1 个 Surface对象，它代表了被返回的后台缓存表面。
记住 Direc3D 通过在交换链表内部交换表面内存指针来翻转表面，而非交换表面本身。这意味着你将总是要对下一个会被显示的后台缓存进行渲染。
重要的是，要注意通过显示适配器驱动程序执行的"翻转操作"和 SwapEffect.Flip建立的交换链表上所应用于的"显示操作"之间的区别。
术语flip传统上表示了改变显示适配器用来产生输出信号的显存地址范围的一个操作，因为这可以让一个先前隐藏的后台缓存被显示出来。在DirectX 9.0 中，这个术语更多用于描述通过 SwapEffect.Flip交换效果建立的任一交换链表中后台缓存的显示表达。
然而当交换链表为全屏时，这样的"显示"操作几乎总是通过翻转操作完成的；当交换链表是窗口时，它们必须通过复制操作来完成。而且，一个显示适配器驱动程序可以在非全屏交换链表下，使用基于 SwapEffect.Discard和 SwapEffect.Copy的翻转来完成"显示"操作。
更多关于窗口和全屏交换链表交换效果区别的讨论，参看 SwapEffect 。

第5节　页面翻转和后台缓冲

页面翻动是多媒体、动画和游戏软件的关键要点。这有点类似于通过翻动一沓纸张表现动画的方式；这就是艺术家略微改变在每个页面上的轮廓外形，因此当你在页面之间快速翻动时，绘制就表现出了动画效果。
Direct3D 通过 1 个交换链表完成翻动功能，该链表是设备的 1 个属性。开始，你应设置一

系列翻动到屏幕上的 Direct3D 缓存，就像艺术家翻到下一页的纸张。第 1 个缓存被引用为色彩前台缓存。它后面的缓存称为后台缓存。你的程序应写入 1 个后台缓存中，然后翻动色彩前台缓存让后台缓存出现在屏幕上。当系统显示图像时，软件再次对 1 个后台缓存进行写入。这个处理过程随动画而继续进行，让你可以有效率的动画显示图像。
Direct3D 简化了建立翻转页面方案，从简单的双缓存方案（就是 1 个具有后台缓存的色彩前台缓存）到更多的具有多个后台缓存的复杂方案。

第6节　复制到表面

当使用 Device.UpdateSurface复制到一个表面时，会应用某些限制条件。你可以传递源表面上的一个矩形，或使用null指定整个表面。你也可以传递目标表面上的一个指针，该指针指向被复制源图像上的矩形左上角位置。
Device.UpdateSurface方法不支持剪裁。该操作会失败，除非源矩形和相应的目标矩形分别完全包含在源表面和目标表面内。注意目标和源表面必须是明显不同的。该方法也不支持alpha混合，色彩键值，或格式转换。
下列方法对于将图像复制到 Direct3D 表面也是可用的。

SurfaceLoader.FromFile
SurfaceLoader.FromStream
SurfaceLoader.FromSurface

相关主题

Device.StretchRectangle

第7节　复制表面

术语blit 是"位块传输"的缩写，这是将数据块从内存中的一处传输到另一处的处理过程。位块传输设备驱动程序界面（DDI）在DirectX 9.0 中继续作为由像素组成的大矩形的基于每帧的主要移动机制。这种机制隐藏在面向复制的 Device.Present方法之后。在blit 操作中，美术工艺品的运输是由 Device.UpdateTexture方法执行的。美术工艺品也可以通过 Device.UpdateSurface在DirectX 9.0 中进行复制，这将复制出一个由像素组成的矩形子集。

相关主题

Device.StretchRectangle

第8节　直接访问表面内存

使用 Surface.LockRectangle方法可以直接访问表面内存。该方法调用时，描述直接访问表面上矩形的 Rectangle结构体将被rect 参数引用。如果要求锁定整个表面，可以调用 Surface.LockRectangle重载方法中不获取 Rectangle结构体的一种，或将rect 参数设置为整个表面的维数。也可以指定仅覆盖表面部分的一个 Rectangle结构。如提供 2 个不交迭的矩形，则通过 2 个线程或进程可以同时锁定一个表面上的多个矩形。注意不能锁定一个多重采样的后台缓存。
依赖于 Surface.LockRectangle重载方法的调用，这个方法返回 1 个 GraphicsStream对

象或一个描述锁定区域的 Array。调用该方法返回在pitch参数中的pitch值。当对表面内存访问结束后，必须调用 Surface.UnlockRectangle方法解除锁定。
当表面被锁定时，可以对表面中的内容直接进行操作。下面的列表提供了一些关于在直接渲染表面内存时，如何避免经常遇到有关问题的小技巧。

绝不要采用一个恒定的显示点距。总是要检查 Surface.LockRectangle方法返回的点距信息。这个点距可能随多种原因发生变化，比如表面内存的位置，显示卡类型，或者甚至是Direct3D驱动程序的版本。更多信息参看 Width vs. Pitch 。

确定要复制到非锁定表面。在一个被锁定表面上调用 Direct3D 复制方法会失败。当表面锁定时限制程序的行为。

如果一个表面属于一个被分配给 Pool.Default 内存池的资源，则该表面就不能被锁定，除非它是一个动态纹理或是通过使用 Device.CreateOffscreenPlainSurface 建立的。仅当交换链表通过将 PresentParameters 结构体的 PresentFlag 属性设置为 PresentFlag.LockableBackBuffer 的方式被建立之后，通过 Device.GetBackBuffer 和 SwapChain.GetBackBuffer 方法访问的后台缓存表面才能被锁定。

第9节　私有表面数据

任何种类的程序指定数据都可以通过表面存储。例如，游戏中表示 1 幅地图的表面也许包含了地形信息。
一个表面可以具有 1 个以上的私有数据缓存。当数据附加到表面上时，每个缓存将通过 1 个由你提供的全局唯一标识符(GUID)进行识别。
要存储私有表面数据，应使用 Surface.SetPrivateData（由 Resource类继承而来），将一个程序定义的GUID和 1 个包含私有数据的 Byte型数组传递到关联的表面。数据通过值传递，并且 1 个资源可以多种设置相关联。
SetPrivateData方法为数据分配 1 个内部缓存并复制它。一旦完成，就可以安全释放源缓存或对象。该内部缓存在 FreePrivateData调用时，被释放。当表面释放时这将会自动发生。
要获得表面的私有数据，应声明 1 个 Byte型数组然后调用 Surface.GetPrivateData方法（也由 Resource类继承而来），将分配给数据的GUID传递到添加的数组。

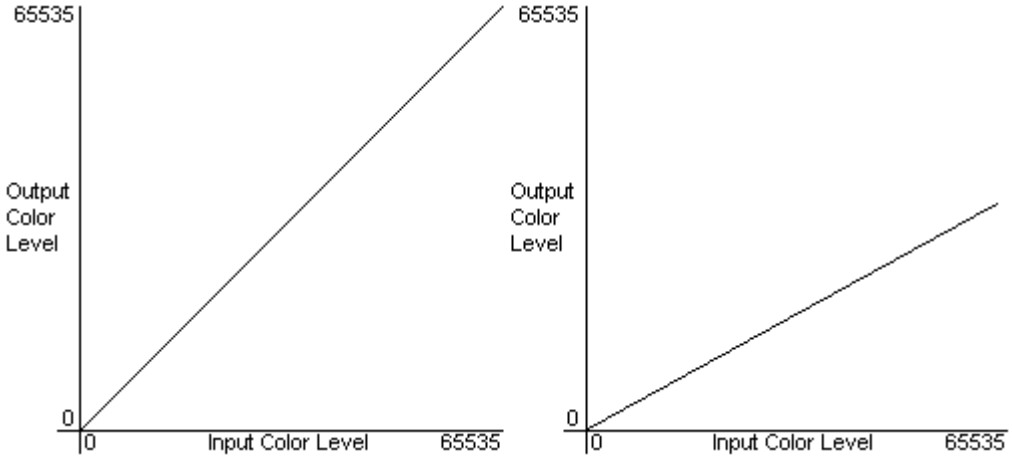
第10节　伽玛控制

伽玛控制允许你改变系统如何显示表面的内容，而不影响其中的内容。可以把这些控制看作是 Direct3D 对数据进行了非常简单的过滤，就像数据被渲染在屏幕上之前已经离开了表面。
伽玛控制是交换链表的一种属性。它们使得动态改变表面的红绿蓝级别如何映射为系统显示的真实级别成为可能。通过设置伽玛级别，你可以在用户角色被击中时，屏幕闪现红色，当角色拣取新物件时闪现绿色等等--而无须将新图像复制到帧缓存中以完成该效果。或者，你也许可以调整色彩级别给后台缓存中的图像应用一个偏色。
因为 DirectX 9.0 中的 Direct3D 将交换链表作为设备的 1 个属性，所以每个设备总是至少有 1 个交换链表（隐式的交换链表）。因为 gamma ramp 是交换链表的 1 个属性，所以

当交换链表为窗口时，它可以被应用。Gamma ramp 立即生效；它不会等待垂直同步操作。
在表面像素的红绿蓝色彩分量被送往数-模显示转换器(DAC)之前，Device.SetGammaRamp和 Device.GetGammaRamp方法可以操纵对像素有影响的ramp级别。

10.1 伽玛控制级别

在 Direct3D 中，术语 gamma ramp 描述了一组值。它们将 1 个特殊色彩分量--在帧缓存中所有像素的红绿蓝--级别映射为 1 个被 DAC 所接收并用于显示的新级别。这个再映射过程是通过 1 个专用于每个色彩分量的查询表执行的。
这是它如何工作的：Direct3D 从帧缓存中获得 1 个像素并分别求出它的红绿蓝色彩分量值。每个分量是用 1 个从 0 到 65565 的值来表示的。Direct3D 获取原始值并使用它来索引 1 个具有 256 个元素的数组（ramp），这里的每个元素都包含了 1 个值用于替代原始值。Direct3D 为帧缓存中每个像素的每个色彩分量执行这个查找和替换过程，因此屏幕上所有的像素都改变了最终色彩。
可以很容易的通过图形化来显示 ramp 值。下图中，左图描绘了 1 个完全没有改变色彩的 ramp。右图描绘了 1 个色彩分量加上负偏色的 ramp。



左图中数组元素包含的值与其索引号相同--0 位于元素索引 0，65535 位于元素索引 255。这是一种缺省类型的 ramp，因为在显示前它没有改变输入值。右图提供了一些变化；它的 ramp 包含的值范围从 0（第 1 个元素）到 32768（最后 1 个元素）。在这中间的值范围是均一。使用这个 ramp 显示出来的的色彩分量看上去效果暗淡。使用线性图形是没有限制的；如果需要，你的程序可以分配任意的映射。你甚至能将起点全部设为 0，从显示中彻底删除 1 个色彩分量。

10.2 设置并获取 Gamma Ramp 级别的返回值

Gamma ramp级别是Direct3D用来将帧缓存中的色彩分量映射为要显示新级别的有效查询表。你可以调用 Device.SetGammaRamp和 Device.GetGammaRamp方法设置并获得主表面的ramp级别。
Device.SetGammaRamp方法接收 3 个参数，Device.GetGammaRamp接收 1 个参数。对 Device.SetGammaRamp来说，第 1 个参数是指定交换链表的 1 个整数；第 2 个参数

calibrate，是 1 个控制伽玛校正是否应用的布尔值；第 3 个参数ramp，是 1 个GammaRamp结构体，该结构包含了 3 个 256 元素的短整形数组（每个数组包含了红、绿、蓝gamma ramp）。Device.GetGammaRamp方法具有的 1 个参数可以得到 1 个从GammaRamp中获取的代表交换链表的整数。
当设置新的伽玛级别时，你可以将 Device.SetGammaRamp方法的calibrate参数设为true以调用测量器。测量gamma ramp会产生一些处理开销，不应频繁使用。可以不管适配器和显示器，设置 1 个校准的gamma ramp为用户提供绝对一致的伽马值。
并非所有系统都支持伽马校正。要决定是否支持，应调用 Manager.GetDeviceCaps并检查返回的 Caps结构体中的 DriverCaps成员。如果 DriverCaps.CanCalibrateGamma属性被设为true，就是支持伽玛校正的。
当设置了新的 ramp 级别，紧记仅在程序处于全屏独占模式下时使用你在数组中设置的级别。只要程序变为正常模式，ramp 级别就将被取消，当程序恢复到全屏模式时再次生效。
如果设备不支持在交换链表当前显示模式（窗口或全屏）下的gamma ramp，不返回错误值。程序可以检查 DriverCaps.SupportsFullscreenGamma 和DriverCaps.CanCalibrateGamma属性以决定设备的功能和测量器是否安装。

第7章　Direct3D 纹理

纹理是一种建立计算机生成真实三维图像的有力工具。DirectX 9.0 托管代码版支持一个广泛的纹理特性集，让开发者可以使用高级纹理技术。

第1节　基本纹理概念

早期计算机生成的 3-D 图像，虽然通常在那个时代是先进的，但看上去有着光亮的塑料外观。它们缺乏诸如摩擦、破裂、指纹、污迹之类的标记类型，这可给 3-D 物体赋予真实形象的复杂外观。近年来，纹理作为一种提升计算机生成 3-D 图像真实感的工具在开发者中广泛流行。
日常生活中，单词"纹理"指的是一个物体的平滑度或粗糙度。然而在计算机图形中，1 个纹理就是 1 幅用像素色彩描绘物体纹理外观的位图。
因为 Direct3D 纹理是位图，任何位图都可以应用于 Direct3D 造型。比如程序可以建立并操纵一些看上去具有木纹感的对象。草、泥、岩石都可以应用于构成山峰的一组三维造型。结果产生了 1 个具有真实外观的山坡。纹理还常用于建立诸如沿途上的路标、峭壁中的岩层、地板的大理石外观等等之类的效果。
另外 Direct3D 支持更多高级纹理特性，诸如纹理混合--带有或不带有透明度的、光影贴图。如果程序建立了 1 个硬件抽象层（HAL）设备或 1 个软件设备，它就可以使用 8，16，24，32，64 或 128 位纹理。

第2节　纹理寻址模式

用于托管代码的 DirectX 9.0 可以将纹理坐标分配给任何造型的任意顶点。通常你赋给顶点的 u-纹理和 v-纹理坐标范围在 0.0 到 1.0（包含该值）。然而如果赋予超出该范围的纹理坐标，你就可以创造出特定的纹理效果。

你可以通过设置纹理寻址模式来控制 Direct3D 处理那些超出[0.0，1.0]范围的纹理坐标。比如你可以让程序设置纹理寻址模式将 1 个纹理平铺到 1 个造型上。
Direct3D 允许程序执行纹理外包。重点注意的是：将纹理寻址模式设置为 TextureAddress.Wrap 和执行纹理外包是不一样的。将纹理寻址模式设置为 TextureAddress.Wrap 导致源纹理的多份拷贝被应用到当前造型上；而启用纹理外包则改变了系统对应用纹理的多边形的光栅化方式。
启用纹理外包使得超出[0.0，1.0]范围的纹理坐标失效，并且非法纹理坐标这种光栅化行为在这儿没有定义。当纹理外包启用时，没有使用纹理寻址模式。因此启用纹理外包时，应确认你的程序没有指定低于 0.0 或高于 1.0 的纹理坐标。

2.1 设置寻址模式

你可以通过设置由 Device.SamplerState 属性中的 SamplerStateManagerCollection.SamplerState 属性返回的 SamplerStateManager 对象，为独立纹理设置纹理寻址模式。寻址模式可以设置为 TextureAddress 枚举型中的任何成员。下列C#代码范例描述了如何设置并获取寻址模式。

[C#]

```
// For this example, device is a valid Device object.
//
using System;
using Microsoft.DirectX.Direct3D;

// Load a texture.
Texture tx = new Texture(device, 4, 4, 0, 0, Format.X8R8G8B8,
Pool.Managed);

// Set the texture in stage 0.
device.SetTexture(0, tx);

// Set some sampler states.
device.SamplerState[0].AddressU = TextureAddress.Clamp;
device.SamplerState[0].AddressV = TextureAddress.Border;
```

```
// Retrieve a sampler state.

TextureAddress ta = device.SamplerState[0].AddressU;
```

2.2 设备限制

虽然通常系统允许纹理坐标超出 0.0 到 1.0 的范围，但硬件限制经常会影响到纹理坐标能够超出范围多少。当你重新获得设备功能时，渲染设备通过 Caps 结构的 MaxTextureRepeat属性中传达这种限制。该成员的值描述了设备允许的纹理坐标全部范围。比如，如果该值为 128，输入纹理坐标就必须保持在-128.0 到+128.0 的范围内。将超出该范围的纹理坐标传递给顶点是非法的。自动生成的纹理坐标和变换的纹理坐标作为生成的纹理坐标结果有着相同的限制。

Caps.MaxTextureRepeat 的解释方式也受到 TextureCaps.SupportsTextureRepeatNotScaledBySize属性的影响。当该属性设置后，Caps.MaxTextureRepeat 中的值就正好用作描述。然而当 TextureCaps.SupportsTextureRepeatNotScaledBySize没有设置时，纹理重复的限制就取决于由纹理坐标索引的纹理尺寸。因此 Caps.MaxTextureRepeat必须通过当前纹理在最高细节级别下尺寸计算出来的合法纹理坐标范围来测量。比如，给定 1 个 32 维纹理和 1 个值为 512 的 Caps.MaxTextureRepeat，真实合法纹理坐标范围是 512/32=16，因此用于该设备的纹理坐标必须在-16.0 到+16.0 范围内。

关于纹理寻址模式的附加信息包含在下列主题中。

外包纹理寻址模式
镜像纹理寻址模式
夹钳纹理寻址模式
边框色彩纹理寻址模式

2.3 外包纹理寻址模式

外包纹理寻址模式由 TextureAddress.Wrap枚举值 识别，它使Direct3D在每个正数连接处重复纹理。假设程序建立了 1 个正方造型并指定了纹理坐标(0.0,0.0)，(0.0,3.0)，(3.0,3.0)和(3.0,0.0)。将纹理寻址模式设置为 TextureAddress.Wrap导致纹理在u-方向和v-方向都被应用三次，如下图所示。

这种纹理寻址模式的效果有点类似于镜像模式，但有明显不同。更多信息参看 Mirror Texture Address Mode。

2.4 镜像纹理寻址模式

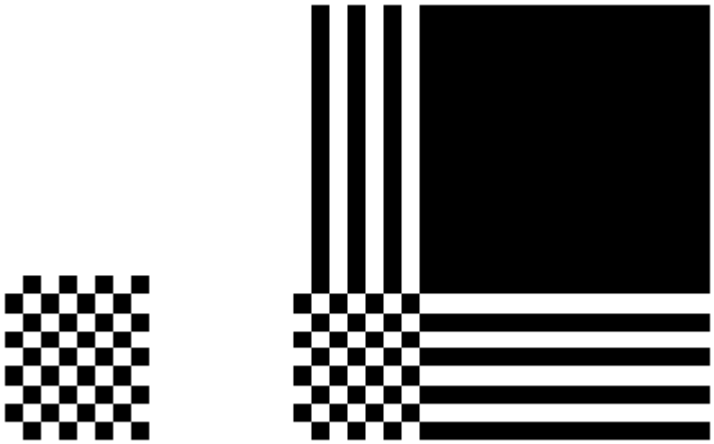
镜像纹理寻址模式由 TextureAddress.Mirror枚举值 识别。这使得Direct3D在每个正数边界线对纹理做镜像。假设程序建立了 1 个正方造型并指定纹理坐标(0.0,0.0), (0.0,3.0), (3.0,3.0)和(3.0,0.0)。将纹理寻址模式设置为 TextureAddress.Mirror导致纹理在u-方向和v-方向上被应用三次。该纹理应用的每隔行或每隔列都是前面行或前面列的镜像，如下图所示。

这种纹理寻址模式的效果有点类似于外包模式，但又有明显不同。更多信息参看 Wrap Texture Address Mode。

2.5 夹钳纹理寻址模式

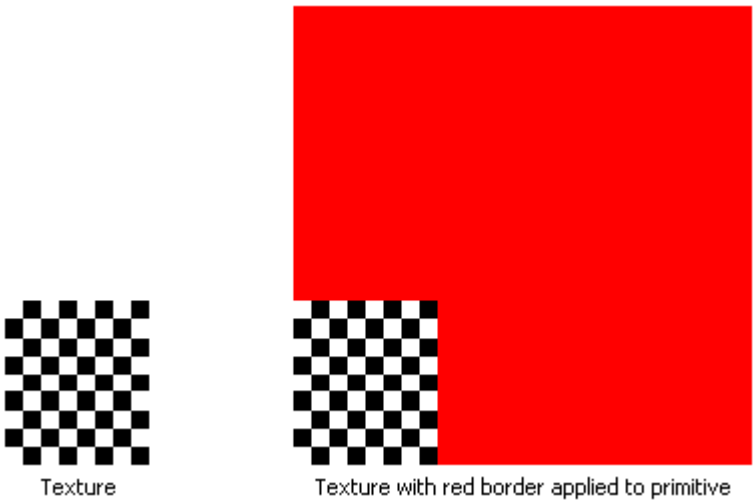
夹钳纹理寻址模式，由 TextureAddress.Clamp枚举的值识别，这使得Direct3D将纹理坐

标在[0.0，1.0]范围内夹住。就是说它应用纹理一次，然后就将边界像素的色彩抹去。
比如假设程序建立了 1 个正方造型并将纹理坐标（0.0，0.0），（0.0，0.3），（3.0，3.0）和（3.0，0.0）赋给造型的顶点。将纹理寻址模式设置为 TextureAddress.Clamp导致纹理应用一次。在列上端和行末段的像素色彩将分别被扩展到造型的上端和右端，如下图所示：



2.6 边框色彩纹理寻址模式

边框色彩纹理寻址模式通过 TextureAddress.Border枚举的值来识别。这使得Direct3D为任何超出 0.0 到 1.0 范围（包含该值）的纹理坐标使用任意一种已知边框色彩。
下图中，程序指定将使用红色边框的纹理应用到造型上。



程序通过设置从 Device.SamplerState 属性中获取的SamplerStateManagerCollection.SamplerState属性返回的SamplerStateManager对象来设置边框色彩。在下列 C# 代码范例中，SamplerStateManagerCollection.SamplerState索引参数被设置为所期望的纹理标识符，SamplerStateManager.BorderColor属性被设置为新的RGBA边框色彩。

```
[C#]
```

```
// For this example, device is a valid Device object.


using System;

using Microsoft.DirectX.Direct3D;


// Load a texture.

Texture tx = new Texture(device, 4, 4, 0, 0, Format.X8R8G8B8,
Pool.Managed);


// Set the texture in stage 0.

device.SetTexture(0, tx);


// Set the address mode for the u-coord.

device.SamplerState[0].AddressU = TextureAddress.Border;


// Set the border color.

device.SamplerState[0].BorderColor = System.Drawing.Color.Blue.ToArgb();


// Get the border color.

int color = device.SamplerState[0].BorderColor;
```

第3节　纹理噪点区域

程序可以通过指定纹理上的噪点区域优化被复制的纹理子集。通过调用 Device.UpdateTexture，仅复制了那些被标记为噪点的区域。然而噪点区域可以被扩大为优化队列。当纹理建立时，整个纹理被考虑是有噪点的。仅下列操作才会影响这种状态。

向 1 个纹理加入 1 个噪点区域。

锁定纹理中的某些缓存。这个操作将被锁定区域作为 1 个噪点区域加入。如果程序可以更好的了解实际噪点区域的话，程序就能关闭这种自动噪点 - 区域更新。

使用纹理的 1 个表面级别作为 Device.UpdateSurface 的目标，将整个纹理标记为有噪点的。

使用纹理作为 Device.UpdateTexture 的来源，清除源纹理上的所有噪点区域。

使用 Surface.GetGraphics 返回 1 个设备的内容。

调用 BaseTexture.GenerateMipSubLevels ，将整个纹理标记为有噪点的。

设置 BaseTexture.AutoGenerateFilterType ，将整个纹理标记为有噪点的。噪点区域设置在 1 个MIP映射纹理的最高级别之上，Device.UpdateTexture可以把噪点区域扩大到MIP链表，使得为每个子级别所复制的字节数最少。注意噪点区域坐标的子级别环绕在外；也就是说它们的分数部分围绕向着纹理最近处的边缘。
因为纹理的每种类型都有不同类型的噪点区域，每个纹理类型上都有下表所示的方法。二维纹理使用噪点矩形，体积纹理则使用盒。

CubeTexture.AddDirtyRectangle
Texture.AddDirtyRectangle
VolumeTexture.AddDirtyBox

忽略上面方法的 rect 或 box 参数会将噪点区域扩大到覆盖整个纹理。
所有的锁定方法都可以获取防止对纹理噪点状态进行改变的 LockFlags.NoDirtyUpdate 方法。更多信息参看 Locking Resources。
当锁定操作期间，关于被改变的真正区域组的更多信息可用时，程序应使用 LockFlags.NoDirtyUpdate。注意对纹理子级别的锁定或复制（不是对最高级别的锁定或复制）仅不更新纹理的噪点区域。当程序锁定了较低级别而没有锁定最高级别时，程序同样负责对噪点区域进行更新。

第4节　纹理调色板

托管代码的DirectX 9.0 通过与 Device对象关联的一组 256 入口（8 位）调色板来支持调色板纹理。一个调色板通过设置 Device.CurrentTexturePalette属性而形成。当前调色板用于为所有活动的纹理进程翻译全部调色板纹理。Device.SetPaletteEntries方法更新调色板全部的 256 入口。每个入口都是 1 个 Format.A8R8G8B8 类型的 PaletteEntry结构。所有入口缺省为 0xFFFFFFFF。
Device调色板包含 1 个alpha通道。当 TextureCaps.SupportsAlphaPalette设备功能标识符设置时，该通道用于指明设备支持从调色板得来的alpha。在纹理格式没有alpha通道时，调色板alpha通道将被使用。如果设备不支持从调色板得来的alpha并且纹理格式没有alpha通道，则将 0xFF将作为alpha值。
最大可以有 65536 (0x0000FFFF)个 调色板。因为与调色板相关联的内存资源和程序引用的最大调色板数量是成正比的，所以使用的毗邻调色板数量应该从 0 开始。

第5节　加载 1 个纹理

使用托管代码的DirectX 9.0 可以很简单的装载 1 个纹理。下列C# 代码范例就这样做了。
使用托管代码的DirectX 9.0 装载 1 个纹理，首先需要建立 1 个合法的 Device对象和 1 个 Texture对象。

```
        [C#]



using Microsoft.DirectX.Direct3D;

.
```

```
.
.
// Global variables for this project.
Device device = null; // Rendering device.
Texture texture = null;


// Initialize the device.
.
.
.
// Device reset method.
public void OnResetDevice(object sender, EventArgs e)
{
    Device dev = (Device)sender;


    // Now create the texture.
    texture = TextureLoader.FromFile(dev, Application.StartupPath +
                                          @"\..\..\banana.bmp");
}
```

相关主题

Tutorial 5: Using Texture Maps
TextureLoader

第8章　Direct3D 指南

下列 C#指南使用了 Microsoft Direct3D 的基本功能来帮助你开始建立单机图形程序。

指南	描述
Tutorial 1: Creating a Device	初始化 Direct3D，渲染 1 个简单的蓝色屏幕，并最终关闭。
Tutorial 2: Rendering Vertices	初始化 Tutorial 1 的过程，建立最简单的形状（1 个三角形），并将它渲染到显示器上。
Tutorial 3: Using Matrices	介绍矩阵的概念并展示了如何使用它们来变换顶点坐标和

	设置照相机、视口。
Tutorial 4: Using Materials and Lights	在 Direct3D 对象中加入光照和材质以建立更多的真实感。
Tutorial 5: Using Texture Maps	向 Direct3D 对象中加入纹理。
Tutorial 6: Using Meshes	介绍网格主题并展示了如何载入、渲染和卸载 1 个网格。

这些指南是循序渐进的，每个指南都保留了前一个指南的大部分代码但加入了更多的功能。所有的指南都位于(*SDK root*)\Samples\Managed\Direct3D\Tutorials\。

第1节　建立 1 个设备

指南项目 CreateDevice 初始化 Micriosoft Direct3D，渲染 1 个简单的蓝色屏幕，最后关闭

1.1 路径

Source location: (*SDK root*)\Samples\Managed\Direct3D\Tutorials\Tutorial1

1.2 过程

1.2.1 建立 1 个程序窗体

在运行时，任何Microsoft Windows程序必须首先做的是建立 1 个程序窗体。为了完成这个操作，下列范例代码*Main()*函数中的首次调用就是程序定义的*CreateDevice*类构造函数，它设置显示窗体尺寸、窗体标题和窗体图标。*CreateDevice*是由Microsoft .NET框架中用来表示程序窗体的 System.Windows.Forms.Form类建立的。

```
            [C#]


using System;

using System.Drawing;

using System.Windows.Forms;

using Microsoft.DirectX;

using Microsoft.DirectX.Direct3D;


namespace DeviceTutorial
```

```
{
    public class CreateDevice : Form
    {
        // Our global variables for this project
        Device device = null; // Our rendering device

        public CreateDevice()
        {
            // Set the initial size of our form
            this.ClientSize = new System.Drawing.Size(400,300);
            // And its caption
            this.Text = "D3D Tutorial 01: CreateDevice";
        }
        .
        .
        .
    }
    .
    .
    .
}
```

1.2.2 初始化 Direct3D 对象

在建立程序窗体后，你即将要初始化用来渲染屏幕的 Direct3D 对象。这个过程包含了建立对象、设置描述参数，最后建立 Direct3D 设备。

```
          [C#]

public bool InitializeGraphics()
{
    try
```

```
    {
        // Now  setup our D3D stuff
        PresentParameters presentParams = new PresentParameters();
        presentParams.Windowed=true;
        presentParams.SwapEffect = SwapEffect.Discard;
        device = new Device(0, DeviceType.Hardware, this,
                  CreateFlags.SoftwareVertexProcessing, presentParams);
                  return true;
    }
    catch (DirectXException)
    {
         return false;
    }
}
```

上面的代码范例依赖于 1 个常用于设置窗体显示特性的 PresentParameters对象。比如将 Windowed属性设置为true，显示的窗体尺寸就小于全屏。默认small-window 格式没有菜单或子窗体，但它为窗体程序保留了最小化、最大化和关闭按钮。这种情况下，由于 SwapEffect.Discard标识，快速交换后台buffer 缓冲器与系统内存的功能将被禁用。如果 Windowed属性设置为false，那么建立的窗体将替代所有非前端显示窗体并驻留于它们之上，即使当窗体没有被激活。

初始化过程的最后步骤是建立Direct3D 设备。在这个范例中输入给 Device(Int32,DeviceType,Control,CreateFlags,PresentParameters[]) 的标识指定了首选硬件设备、以软件方式进行顶点处理。注意如果你通过指定 CreateFlags.HardwareVertexProcessing告诉系统使用硬件顶点处理，你将会在支持硬件顶点处理的显示卡上看见很大的性能改进。

1.2.3 渲染 Direct3D 对象

程序使用 Application.DoEvents方法在循环中保持运行，该方法具有 1 个叫做*frm.*的*CreateDevice*对象。DoEvents在当前线程上运行 1 个标准Windows程序消息循环。

```
        [C#]

static void Main()
{
```

```
    using (CreateDevice frm = new CreateDevice())
    {
        if (!frm.InitializeGraphics()) // Initialize Direct3D
        {
            MessageBox.Show(
                  "Could not initialize Direct3D.  This tutorial will exit.");
            return;
        }
        frm.Show();

        // While the form is still valid, render and process messages
        while(frm.Created)
        {
            frm.Render();
            Application.DoEvents();
        }
    }
}
```

当建立的CreateDevice Form对象有效时，1 个程序定义的Render方法就被调用来渲染Direct3D对象。首先通过 Device.Clear方法，视口（打开的窗体）被设置为同一的蓝色。使用 Device.BeginScene方法开始进行场景渲染。渲染完成后，场景将以 EndScene和Present方法的成功调用而结束。

```
          [C#]

private void Render()
{
    if (device == null)
        return;
```

```
    //Clear the backbuffer to a blue color
    device.Clear(ClearFlags.Target, System.Drawing.Color.Blue, 1.0f, 0);
    //Begin the scene
    device.BeginScene();

    // Rendering of scene objects can happen here

    //End the scene
    device.EndScene();
    device.Present();
}
```

第2节　渲染顶点

用Microsoft Direct3D书写的程序使用了顶点来绘制几何外形。每个三维（3-D）场景包含 1 个或多个这些几何外形。指南项目*Vertices*以 Tutorial 1: Creating a Device的初始化过程开始，然后建立 1 个最简单的形状三角形，将它渲染到显示器上。

2.1 路径

Source location: (*SDK root*)\Samples\Managed\Direct3D\Tutorials\Tutorial2

2.2 过程

这个指南使用三个顶点渲染 1 个 2-D三角形。它包含了顶点缓存的概念，通常 1 个 VertexBuffer对象常用于存储和渲染顶点。
顶点可以通过 CustomVertex自定义顶点类中的可用结构体以多种方式来定义。这里顶点是经过变换的，因此顶点已经是 2-D窗体坐标。所以点（0，0）位于左上角，x-轴正向朝右，y-轴正向朝下。这些顶点也没有使用Direct3D光照，但是它们提供了自身漫反射色彩来代替。通过使用 CustomVertex.CustomVertex.TransformedColored结构体初始化顶点缓存可以指定这个特性，如下列代码片断所示。

```
            [C#]

public class Vertices : Form
{
    // Global variables for this project
```

```
Device device = null; // Rendering device
VertexBuffer vb = null;
.
.
.
public void OnCreateDevice(object sender, EventArgs e)
{
Device dev = (Device)sender;

// Now create the vertex buffer
vertexBuffer = new VertexBuffer(
    typeof(CustomVertex.TransformedColored), 3, dev, 0,
    CustomVertex.TransformedColored.Format, Pool.Default);
vertexBuffer.Created +=
    new System.EventHandler(this.OnCreateVertexBuffer);
this.OnCreateVertexBuffer(vb, null);
}

public void OnCreateVertexBuffer(object sender, EventArgs e)
{
VertexBuffer vb = (VertexBuffer)sender;
GraphicsStream stm = vb.Lock(0, 0, 0);
CustomVertex.TransformedColored[] verts =
    new CustomVertex.TransformedColored[3];
.
.
.
vb.Unlock();
}
```

```
}
```

在上面代码中，VertexBuffer.Lock方法被调用允许CPU直接访问顶点缓存数据。每个锁定调用必须跟随 1 个解除锁定调用。参看 Locking Resources获取更多关于锁定和解除锁定的信息。
在上面代码中建立的verts对象是 1 个具有 3 个 CustomVertex.TransformedColored结构体元素的数组，三角形三个顶点中的每一个都对应 1 个结构体。每个顶点结构体的(X,Y,X), Rhw 和 Color字段在下面代码中被初始化。ToArgb方法常用于按照需要的ARGB32 位格式提供色彩数据。

```
        [C#]

verts[0].X=150; verts[0].Y=50; verts[0].Z=0.5f; verts[0].Rhw=1;
verts[0].Color = System.Drawing.Color.Aqua.ToArgb();


verts[1].X=250; verts[1].Y=250; verts[1].Z=0.5f; verts[1].Rhw=1;
verts[1].Color = System.Drawing.Color.Brown.ToArgb();


verts[2].X=50; verts[2].Y=250; verts[2].Z=0.5f; verts[2].Rhw=1;
verts[2].Color = System.Drawing.Color.LightPink.ToArgb();


stm.Write(verts);
```

GraphicsStream类（stm对象）在上面 2 段代码片断中进行了介绍。1 个图形流常用于将数据绑定到输入寄存器上，以便着色器使用。这个代码提供了 GraphicsStream对象直接对顶点缓存访问，或者换一种方式理解，它将顶点缓存的内容替换进图形流之中，并通过顶点buffer缓冲器的长度在流中提前了当前位置。
下列代码片断描述了私有Render方法是如何从Tutorial 1 中扩展来初始化流源、顶点格式和 Device对象造型渲染的。和Tutorial 1 中一样，只要当前Vertices对象（frm）是有效的或者如果系统绘制事件被触发了，Render就渲染到显示器上。

```
        [C#]

private void Render()
{
    if (device == null)
    return;
```

```
    // Clear the back buffer to a blue color (ARGB = 000000ff)
    device.Clear(ClearFlags.Target, System.Drawing.Color.Blue, 1.0f, 0);
    // Begin the scene
    device.BeginScene();

    // New for Tutorial 2
    device.SetStreamSource(0, vertexBuffer, 0);
    device.VertexFormat = CustomVertex.TransformedColored.Format;
    device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);

    // End the scene
    device.EndScene();
    device.Present();
}
```

第3节　使用矩阵

指南项目 Matrices 介绍了矩阵的概念并展示了如何使用它们。矩阵常用于变换顶点坐标和设置照相机、视口。

3.1 路径

Source location: (*SDK root*)\Samples\Managed\Direct3D\Tutorials\Tutorial3

3.2 过程

注意：关于初始化Microsoft Direct3D、处理Microsoft Windows消息、渲染或关闭的信息，参看 Tutorial 1: Creating a Device。

Tutorial 2: Rendering Vertices 将 2-D顶点进行渲染后绘制出 1 个三角形。该指南向 Tutorial 2 中加入了通过使用 3-D顶点变换来旋转三角形的代码。

因为这个项目将变换应用到三角形对象，替代了在Tutorial 2 使用的已经过变换的 2-D窗体坐标，顶点缓存通过 CustomVertex.CustomVertex.PositionColored结构体被初始化，如下代码片断所示。

```
        [C#]

Device dev = (Device)sender;
```

```
// Now create the vertex buffer.
vertexBuffer = new VertexBuffer(typeof(CustomVertex.PositionColored),
                                3,
                                dev,
                                0,
                                CustomVertex.PositionColored.Format,
                                Pool.Default);
```

另外，在私有Render 方法中，设备顶点格式被初始化为 CustomVertex.PositionColored 格式，如下列代码所示。在几何体被渲染前，程序定义的建立和设置三角形对象 3-D矩阵变换的SetupMatrices方法，将从Render中被调用。

```
        [C#]

private void Render()
{
    .
    .
    .
    // Set up the world, view, and projection matrices.
    SetupMatrices();

    device.SetStreamSource(0, vertexBuffer, 0);
    device.VertexFormat = CustomVertex.PositionColored.Format;
    device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);

    // End the scene.
    device.EndScene();
    device.Present();
}
```

通常有 3 种类型的变换被设置用于 1 个 3-D场景。这些变换全部定义为 Transforms对象的属性，从 Device.Transform属性访问。它们全部使用 1 个Direct3D典型的左手坐标系统；参看 3-D Coordinate Systems。

1. World Transformation Matrix:这里通过调用 Matrix.RotateY方法，三角形围绕y-轴旋转，如下列代码范例所示。注意 Matrix是 Microsoft.DirectX命名空间的多种用途的一部分。该调用使用了系统 Environment.TickCount方法，被 1 个缩放值来除，以给出用弧度表示的 RotateY参数。这个步骤可以围绕y-轴进行 1 个平滑变化的旋转。
2. View Transformation Matrix:这个范例代码中通过调用 Matrix.RotateY方法，视点变换矩阵给出了场景的照相机的视角。3 个 Vector3 向量构成了 LookAtLH方法的参数，该方法构建 1 个左手（LH）观看的矩阵。3 个向量分别描述了眼睛的位置、照相机观看目标（本例中的源点）和当前世界上行方向。
3. Projection Transformation Matrix:投影变换矩阵定义了几何体是如何从 3-D可视空间被转换为 2-D视口空间的。这个范例代码中，它是由从左手 PerspectiveFovLH方法返回的矩阵中所构成的。该方法的参数有用弧度表示的视野范围、高宽比（可视空间高度除以宽度）、近端剪裁位面距离和远端剪裁位面距离。

这些变换矩阵建立的顺序不会影响场景中对象的设计编排。然而 Direct3D 是按上面的顺序将矩阵应用到场景中的。

[C#]

```
private void SetupMatrices()
{
    // For our world matrix, we will just rotate the object about the y-axis.

    // Set up the rotation matrix to generate 1 full rotation (2*PI radians)
    // every 1000 ms. To avoid the loss of precision inherent in very high
    // floating point numbers, the system time is modulated by the rotation
    // period before conversion to a radian angle.
    int   iTime  = Environment.TickCount % 1000;
    float fAngle = iTime * (2.0f * (float)Math.PI) / 1000.0f;
    device.Transform.World = Matrix.RotationY( fAngle );

    // Set up our view matrix. A view matrix can be defined given an eye
    // point, a point to lookat, and a direction for which way is up. Here,
    // we set the eye five units back along the z-axis and up three units,
    // look at the origin, and define "up" to be in the y-direction.
```

```
    device.Transform.View = Matrix.LookAtLH(
        new Vector3( 0.0f, 3.0f,-5.0f ),
        new Vector3( 0.0f, 0.0f, 0.0f ),
        new Vector3( 0.0f, 1.0f, 0.0f ) );

    // For the projection matrix, we set up a perspective transform (which
    // transforms geometry from 3D view space to 2D viewport space, with
    // a perspective divide making objects smaller in the distance). To build
    // a perpsective transform, we need the field of view (1/4 pi is common),
    // the aspect ratio, and the near and far clipping planes (which define
    // at what distances geometry should be no longer be rendered).
    device.Transform.Projection = Matrix.PerspectiveFovLH(
        (float)Math.PI / 4,
        1.0f,
        1.0f,
        100.0f );
}
```

渲染特性通过设置 RenderStateManager类的属性控制。这是在OnResetDevice程序定义方法中完成的，如下代码片断所示。

```
          [C#]

public void OnResetDevice(object sender, EventArgs e)
{
    Device dev = (Device)sender;
    // Turn off culling, so the user sees the front and back of the triangle
    dev.RenderState.CullMode = Cull.None;
    // Turn off Direct3D lighting, since object provides its own vertex colors
    dev.RenderState.Lighting = false;
}
```

这里，背面消除和 Direct3D 光照都被关闭。这些设置允许 3-D 对象的全部深度可见并且

对象可以提供自身的色彩。

第4节　使用材质和光照

指南项目 Lights 为 Microsoft Direct3D 对象建立更多的真实感加入了光照和材质。基于位置和使用的光照类型，场景中的每个对象将被照亮。材质描述了多边形是如何反射周围环境和漫反射色彩的，它们的镜面高光是如何呈现的，多边形看上去是否发出光线。

4.1 路径

Source location: (*SDK root*)\Samples\Managed\Direct3D\Tutorials\Tutorial4

4.2 过程

注意：关于初始化Direct3D，处理Microsoft Windows消息或关闭的信息，参看 Tutorial 1: Creating a Device
Tutorial 3: Using Matrices 在 3-D中经过变换的对象顶点。该指南向Tutorial 3 中加入了建立 1 个材质和 1 个光源的代码。

4.2.1 初始化 1 个深度模版

如下列代码片断所示，该项目也向 Tutorial 3 initialization 加入了 1 个初始化过程以允许使用 1 个 z-缓存（深度缓存）和 1 个深度模版。深度模版允许程序掩饰被渲染图像的一部分，使得它们不被显示出来。这儿深度模版首先被启用，然后格式被设置为 1 个 16 位 z-缓存深度。

```
        [C#]

public bool InitializeGraphics()
{
    .
    .
    .
    // Turn on a depth stencil
    presentParams.EnableAutoDepthStencil = true;

    // Set the stencil format
    presentParams.AutoDepthStencilFormat = DepthFormat.D16;
```

```
    .
    .
    .
}
```

4.2.2 初始化顶点缓存和渲染状态

使用光照的要求之一就是每个表面具有 1 个法向量。因此该项目使用了 1 个与前面指南中不同的自定义顶点类型 CustomVertex.PositionNormal结构体，它包含了Direct3D内部用于光照计算的 1 个 3-D位置和 1 个表面法线。

```
            [C#]

public void OnCreateDevice(object sender, EventArgs e)
{
    Device dev = (Device)sender;

    // Now create the vertex buffer
    vertexBuffer = new VertexBuffer(typeof(CustomVertex.PositionNormal),
                                    100,
                                    dev,
                                    Usage.WriteOnly,
                                    CustomVertex.PositionNormal.Format,
                                    Pool.Default);
    vertexBuffer.Created +=
                        new System.EventHandler(this.OnCreateVertexBuffer);
    this.OnCreateVertexBuffer(vertexBuffer, null);
}
```

范例代码也允许使用 1 个z-缓存（深度缓存）来高效存储场景中的几何体，并通过RenderStateManager属性启用Direct3D光照，如下所示。

```
            [C#]

public void OnResetDevice(object sender, EventArgs e)
```

```
{
    Device dev = (Device)sender;

    // Turn off culling, so the user sees the front and back of the triangle
    device.RenderState.CullMode = Cull.None;

    // Turn on the z-buffer
    device.RenderState.ZBufferEnable = true;
    device.RenderState.Lighting = true;      // Make sure lighting is enabled
}
```

4.2.3 建立圆柱体对象

在设备初始化时（上面所示的 OnCreateDevice），程序定义的 OnCreateVertexBuffer 方法被调用建立 1 个圆柱体对象。1 个存储了圆柱体点的顶点缓存初始化，然后圆柱体上每个点的位置和法线被载入顶点缓存，如下面范例代码所示。

```
            [C#]

public void OnCreateVertexBuffer(object sender, EventArgs e)
{
    VertexBuffer vb = (VertexBuffer)sender;
    // Create and lock a vertex buffer (which will return the structures)
    CustomVertex.PositionNormal[] verts =
                        (CustomVertex.PositionNormal[])vb.Lock(0,0);

    for (int i = 0; i < 50; i++)
    {
        // Fill up the structs
        float theta = (float)(2 * Math.PI * i) / 49;
        verts[2 * i].Position = new Vector3(
                    (float)Math.Sin(theta), -1, (float)Math.Cos(theta));
```

```
        verts[2 * i].Normal = new Vector3(
                (float)Math.Sin(theta), 0, (float)Math.Cos(theta));
        verts[2 * i + 1].Position = new Vector3(
                (float)Math.Sin(theta), 1, (float)Math.Cos(theta));
        verts[2 * i + 1].Normal = new Vector3(
                (float)Math.Sin(theta), 0, (float)Math.Cos(theta));
    }
    // Unlock (and copy) the data
    vb.Unlock();
}
```

在 SetupMatrices 私有方法中使用世界变换矩阵，圆柱体被旋转，和 Tutorial 3 中一样。

4.2.4 建立 1 个材质

材质定义了当 1 个光源照射时几何体对象表面反射的色彩。下列代码片断使用 Material结构体来建立 1 个白色材质。材质的漫反射和环境色彩属性被设置为白色。在该调用完成后，每个造型将用该材质渲染直到色彩属性被设置为另一个值。

```
          [C#]

private void SetupLights()
{
    System.Drawing.Color col = System.Drawing.Color.White;

    // Set up a material. The material here just has the diffuse and ambient
    // colors set to white. Note that only one material can be used at a time.
    Direct3D.Material mtrl = new Direct3D.Material();
    mtrl.Diffuse = col;
    mtrl.Ambient = col;
    device.Material = mtrl;
    .
    .
```

```
    .
}
```

4.2.5 建立 1 个光源

Direct3D中有 3 种类型的可用光源：点光源、方向光源和聚光灯。该指南项目建立了 1 个光线照向某个方向的方向光源，光线的方向会振荡改变。下列代码片断使用 1 个 Light对象来建立 1 个暗绿宝石色的方向光源。场景中的所有对象都被 1 种低级的单色（灰度）环境光线所照亮。

```
        [C#]

private void SetupLights()
{
    .
    .
    .
    // Set up a colored directional light, with an oscillating direction.
    // Note that many lights may be active at a time (but each one slows down
    // the rendering of the scene). However, here just one is used.
    device.Lights[0].Type = LightType.Directional;
    device.Lights[0].Diffuse = System.Drawing.Color.DarkTurquoise;
    device.Lights[0].Direction = new Vector3(
                    (float)Math.Cos(Environment.TickCount / 250.0f),
                    1.0f,
                    (float)Math.Sin(Environment.TickCount / 250.0f));

    device.Lights[0].Enabled = true; // Turn it on

    // Finally, turn on some ambient light.
    // Ambient light is light that scatters and lights all objects evenly.
   device.RenderState.Ambient = System.Drawing.Color.FromArgb(0x202020);
```

第5节　使用纹理映射

指南项目*Textures*向Microsoft Direct3D对象中加入了纹理。虽然光照和材质为场景添加了很多真实感，但没有任何东西比向表面添加纹理更能表现出真实感了。纹理可以认为是在表面上包裹的墙纸。你可以把 1 个木纹理放置在 1 个立方体上使它看上去确实象由木质构成。该指南项目将 1 个"剥落香蕉"的纹理加入 Tutorial 4: Using Materials and Lights 中建立的圆柱体对象。

5.1 路径

Source location: (*SDK root*)\Samples\Managed\Direct3D\Tutorials\Tutorial5

5.2 过程

注意：关于初始化Direct3D，处理Microsoft Windows消息或关闭的信息，参看 Tutorial 1: Creating a Device。
Tutorial 4: Using Materials and Lights 在Direct3D对象上建立了材质和光照。该指南向Tutorial 4 中加入了用于加载纹理、设置顶点和显示具有纹理的对象的代码。

5.2.1 建立纹理

要使用纹理，建立的顶点缓存必须在它自定义顶点格式中具有纹理坐标。纹理坐标告诉 Direct3D 在哪里为造型中的每个向量放置纹理。纹理坐标范围从 0.0 到 1.0，(0.0, 0.0) 表示了纹理的左上角，(1.0, 1.0)代表纹理的右下角。
在初始化 1 个新 Texture 对象后，下面范例代码使用 CustomVertex.PositionNormalTextured结构体设置了顶点缓存的格式。这个初始化预备让顶点缓存接收（x，y，z）位置和（x，y，z）法线数据及 1 组（tu，tv）纹理坐标。

```
        [C#]

public class Textures : Form
{
    // Our global variables for this project
    Device device = null; // Our rendering device
    VertexBuffer vertexBuffer = null;
    Texture texture = null;
    .
    .
    .
```

```
    public void OnCreateDevice(object sender, EventArgs e)
    {
        Device dev = (Device)sender;

        // Now create the vertex buffer
        vertexBuffer = new VertexBuffer(
                            typeof(CustomVertex.PositionNormalTextured),
                            100,
                            dev,
                            Usage.WriteOnly,
                            CustomVertex.PositionNormalTextured.Format,
                            Pool.Default);
        vertexBuffer.Created += new System.EventHandler(
                            this.OnCreateVertexBuffer);
        this.OnCreateVertexBuffer(vertexBuffer, null);
    }
}
```

5.2.2 载入纹理数据

渲染状态设置（显示参数）与Tutorial 4 中的类似，除了Direct3D光照被禁用以允许显示圆柱体对象的色彩，使用 TextureLoader.FromFile方法将从 1 幅位图中载入 1 个"剥落香蕉"纹理。

```
        [C#]

public void OnResetDevice(object sender, EventArgs e)
{
    Device dev = (Device)sender;

    // Turn off culling, so the user sees the front and back of the triangle
```

```
    dev.RenderState.CullMode = Cull.None;

    // Turn off Direct3D lighting
    dev.RenderState.Lighting = false;

    // Turn on the z-buffer
    dev.RenderState.ZBufferEnable = true;

    // Now create the texture
    texture = TextureLoader.FromFile(dev,
                        Application.StartupPath + @"\..\..\banana.bmp");
}
```

通过圆柱体对象的属性载入顶点缓存的步骤和在 Tutorial 4 中程序定义的 OnCreateVertexBuffer 方法是一样的。为每个点加入额外的纹理坐标（tu，tv），如下面代码范例所示。该过程导致在渲染时纹理平滑的包裹在圆柱体上，并且纹理始终保持与原始圆柱体的绑定。

```
          [C#]

public void OnCreateVertexBuffer(object sender, EventArgs e)
{
    VertexBuffer vb = (VertexBuffer)sender;

    // Create a vertex buffer
    // Lock the buffer (which will return the structures)
    CustomVertex.PositionNormalTextured[] verts =
                (CustomVertex.PositionNormalTextured[])vb.Lock(0,0);
    for (int i = 0; i < 50; i++)
    {
        // Fill up the structures
        float theta = (float)(2 * Math.PI * i) / 49;
```

```
        verts[2 * i].Position = new Vector3(
                (float)Math.Sin(theta), -1, (float)Math.Cos(theta));
        verts[2 * i].Normal = new Vector3(\
                (float)Math.Sin(theta), 0, (float)Math.Cos(theta));
        verts[2 * i].Tu = ((float)i)/(50-1);
        verts[2 * i].Tv = 1.0f;
        verts[2 * i + 1].Position = new Vector3(
                (float)Math.Sin(theta), 1, (float)Math.Cos(theta));
        verts[2 * i + 1].Normal = new Vector3(
                (float)Math.Sin(theta), 0, (float)Math.Cos(theta));
        verts[2 * i + 1].Tu = ((float)i)/(50-1);
        verts[2 * i + 1].Tv = 0.0f;
    }
}
```

5.2.3 渲染场景

5.2.3 渲染场景

纹理阶段允许你定义 1 个或多个纹理如何渲染的行为。比如，你可以把多个纹理混合在一起。该指南项目以Tutorial 4 中的私有*Render*方法和纹理阶段 0 开始，0 是Direct3D设备用来渲染的纹理阶段。1 个设备最多可以有 8 组纹理，因此最大阶段值为 7，但是该指南项目仅有 1 个纹理并把它放在 0 阶段。0 既用作 Device.SetTexture方法的**stage**参数，又用作 Device.TextureState属性的数组值。

TextureOperation和 TextureArgument枚举类型的常量值控制了纹理如何渲染。比如，该代码中的设置导致输出色彩成为纹理色彩与漫反射色彩的乘积。更多关于在纹理中混合色彩的信息。

```
        [C#]

private void Render()
{
    .
    .
    .
    // Set up our texture. Using textures introduces the texture stage states,
```

```
    // which govern how textures get blended together (in the case of multiple
    // textures) and lighting information. In this case, modulate (blend)
    // the texture with the diffuse color of the vertices.
    device.SetTexture(0,texture);
    device.TextureState[0].ColorOperation = TextureOperation.Modulate;
    device.TextureState[0].ColorArgument1 =
TextureArgument.TextureColor;
    device.TextureState[0].ColorArgument2 = TextureArgument.Diffuse;
    device.TextureState[0].AlphaOperation = TextureOperation.Disable;
    .
    .
   .
}
```

第6节　使用网格

复杂几何体通常使用 3-D 建模软件构造模型并保存为 1 个文件。这里一个例子就是.x 文件格式。Microsoft Direct3D 使用网格从这些文件中载入对象。指南项目 *Meshes* 介绍了网格主题和如何载入、渲染、卸载 1 个网格。
1 个网格包含了 1 个复杂模型的数据。它是 1 个抽象数据容器，包含了诸如纹理和材质之类的资源、诸如位置数据和邻接数据之类的属性。虽然网格有些复杂，但 Direct3D 包含了可以让使用网格变得容易的方法。

6.1 路径

Source location: (*SDK root*)\Samples\Managed\Direct3D\Tutorials\Tutorial6

6.2 过程

注意：关于初始化Direct3D，处理Microsoft Windows消息或关闭的信息，参看 Tutorial 1: Creating a Device。
Tutorial 5: Using Texture Maps 在Direct3D对象上建立了纹理。该指南向Tutorial 5 代码中加入了从文件中处理 1 个网格的过程，但它放弃了OnCreateDevice程序定义的方法并与常用来建立 1 个顶点缓存的设备建立调用语句相关联。这儿使用的过程在网格方法调用内隐式的处理顶点缓存。

6.2.1 初始化 1 个网格对象

如下所示，该指南项目初始化多维材质和纹理数组。还要注意using 语句声明的System.ComponentModel和 System.IO命名空间。

```
        [C#]

using System;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;
using System.IO;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
using Direct3D = Microsoft.DirectX.Direct3D;
public class Meshes : Form
{
    Device device = null;               // Rendering device
    Mesh mesh = null;                   // Mesh object in system memory
    Direct3D.Material[] meshMaterials; // Materials for the mesh
    Texture[] meshTextures;             // Textures for the mesh
    PresentParameters presentParams = new PresentParameters();
    .
    .
    .
}
```

程序定义的*OnResetDevice*方法初始化 1 个 ExtendedMaterial对象，该对象常用于将网格文件数据捕获到 1 个 Material结构体中。该方法设置了网格文件的目录路径，初始化 1 个设备，并打开z-缓存和白色环境灯光。

```
        [C#]

public void OnResetDevice(object sender, EventArgs e)
```

```
{
    ExtendedMaterial[] materials = null;

    // Set the directory up to load the right data, because the
    // default build location is Bin\debug or Bin\release.
    Directory.SetCurrentDirectory(Application.StartupPath +  @"\..\..\");

    Device dev = (Device)sender;

    // Turn on the z-buffer.
    dev.RenderState.ZBufferEnable = true;

    // Turn on ambient lighting.
    dev.RenderState.Ambient = System.Drawing.Color.White;
    .
    .
    .
}
```

6.2.2 载入 1 个网格对象

如下代码片断所示，OnResetDevice方法下一次从tiger.x文件中载入了 1 个网格，该网格描绘了 1 个经过纹理贴图的 3-D老虎。在这个 Mesh.FromFile方法调用中，MeshFlags 枚举类型的 SystemMemory常量指明网格将被载入系统RAM而不是通常的设备可访问 RAM。在网格载入后，1 个meshMaterials Material对象的成员将被来自于网格文件的 Materials ExtendedMaterial对象的 Material结构体所填充。材质也被设置为环境色。最后通过对 TextureLoader.FromFile方法的调用，1 个具有纹理的meshTextures Texture对象被载入。meshMaterials 和 meshTextures 都被初始化为文件中的 Materials 结构体的维数（Length）。

```
        [C#]

public void OnResetDevice(object sender, EventArgs e)
{
```

```
    .
    .
    .
    // Load the mesh from the specified file.
    mesh = Mesh.FromFile("tiger.x",
                         MeshFlags.SystemMemory,
                         device,
                         out materials);
    if (meshTextures == null)
    {
        // Extract the material properties and texture names.
        meshTextures  = new Texture[materials.Length];
        meshMaterials = new Direct3D.Material[materials.Length];
        for( int i=0; i<materials.Length; i++ )
        {
            meshMaterials[i] = materials[i].Material3D;

            // Set the ambient color for the material. Direct3D
            // does not do this by default.
            meshMaterials[i].Ambient = meshMaterials[i].Diffuse;

            // Create the texture.
            meshTextures[i] = TextureLoader.FromFile(dev,
                                      materials[i].TextureFilename);
        }
    }
}
```

6.2.3 渲染 1 个网格对象

在网格载入后，私有 *Render* 方法被调用来渲染网格对象。首先它开始场景并调用 *SetupMatrices* 程序定义的方法，和 Tutorial 5 中做的一样。为了渲染网格，它被分成很多子集，每个对应于 1 个载入的材质。如下列代码片断所示，运行 1 个循环以渲染每个材质子集。循环为每个材质作了以下工作：

设备的 Material属性被设置为meshMaterials Material结构体。
设备纹理阶段 0 被设置为meshTextures Texture结构体。
材质子集通过 DrawSubset方法来绘制，该方法从 BaseMesh类中继承而来。

```
[C#]

private void Render()
{
    .
    .
    .
    for( int i=0; i<meshMaterials.Length; i++ )
    {
        // Set the material and texture for this subset.
        device.Material = meshMaterials[i];
        device.SetTexture(0, meshTextures[i]);

        // Draw the mesh subset.
        mesh.DrawSubset(i);
    }

    // End the scene.
    device.EndScene();
    device.Present();
}
```

第9章　Direct3D 托管代码范例

这部分的范例展示了如何在C#托管代码程序中使用Microsoft® Direct3D®。一旦你已经掌握了 tutorials指南，你就准备跳入到范例学习中。一些其它的例子示范了Microsoft DirectX® 9.0 托管版的最新特色。其它例子示范了现有的Direct3D的基本功能。所有的范例建立在范例框架的顶部。参看 The Sample Framework 和 Sample Framework Reference 。

范例

这些例子示范了融合进 DirectX 的最新技术。虽然 DirectX 范例包含 Microsoft Visual Studio® .NET 解决方案文件，你可能需要在你的开发环境验证其它设置来确保范例的正确编译。

范例	描述
BasicHLSL Direct3D Sample	同时采用Microsoft Visual C#® 和 Microsoft Visual Basic® .NET编码，这是 1 个使用高级着色语言（HLSL）和Effect类的简单范例。
CustomUI Direct3D Sample	1 个用户界面（UI）子系统的简单示范，是用托管代码版 DirectX 9.0 范例框架完成的。
EmptyProject Direct3D Sample	1 个新 Direct3D 应用程序的开始点。
EnhancedMesh Direct3D Sample	示范了 Direct3D 中的网格镶嵌。
FragmentLinker Direct3D Sample	使用FragmentLinker类将着色器源代码分成一系列着色器片断，它们被分别编译并连接在一起形成 1 个完整的着色器。
HDRCubeMap Direct3D Sample	使用浮点纹理和高动态范围（HDR）光照示范了立方体环境映射。
HLSLwithoutEffects Direct3D Sample	提供了 1 个如何使用 HLSL 而不使用 D3DX 特效界面的范例。
ProgressiveMesh Direct3D Sample	展示了1个应用程序如何使用D3DX渐进网格功能来简化网格以加速渲染。
PrtPerVertex Direct3D Sample	示范了如何使用 PrtEngine，1 个使用了低次序球调和函数（SH）的预先计算次表面散射（PRT）模拟器。
C# Scripting Direct3D Sample	示范了使用 C#代码作为你非托管应用程序中脚本的 1 种可能技术，并包括了这样的技术。它可以确保被认为不安全脚本不能执行。
SimpleAnimation Direct3D Sample	展示了如何动画 1 个具有骨骼结构的 3-D 模型。
Text3D Direct3D Sample	展示了如何绘制 1 个 3-D 场景中的 2-D 文本。

第1节　BasicHLSL 范例

这是个使用了 Effect特效类的高级着色语言(HLSL)简单的范例，既有C#代码，也有VB代码。

1.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\BasicHLSL
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\csBasicHLSL.exe

1.2 范例概览

范例简单的加载了一个 Mesh网格，创建了一个来自文件的 Effect特效，并且然后使用 Effect来渲染 Mesh。那个被使用的 Effect效果是一个根据时间变化来渲染顶点的简单顶点着色器，它可能具有多个方向性光源。

1.3 范例是如何工作的

首先范例用 Mesh.FromFile方法加载了几何体，并且调用 ComputeNormals方法来建立网格，如果还没有的话。例子然后调用 OptimizeInPlace来为顶点缓存初始化网格。然后使用 **CreateTextureFromFile** 方法加载它的纹理。最后，调用它的 **CreateEffectFromFile** 方法来编译特效的文本文件到一个叫做**effect**的 Effect对象。注意**CreateEffectFromFile**调用获得了那些需要用Microsoft Visual Studio® .NET shader debugger调试的标识。调试顶点着色需要引用或着软件顶点处理，并且调试像素着色需要引用顶点处理。 ForcePixelShaderSoftwareNoOptimizations 标识和 ForceVertexShaderSoftwareNoOptimizations标识改善了shader debugger的调试体验。它们允许源级别的调试，防止指令重安排、防止死代码、强制编译器编译下一个最可能利用的软件目标，确保非优化的着色不会超越受限的着色模式。设置这些标识引起较慢的渲染，因为着色器不将优化以及强制运行在软件。要在范例代码上打开这些标识，简单的从靠近文件顶部的#define DEBUG_VS或者#define DEBUG_PS行删除注释标记即可。通过加一些代码到**ModifyDeviceSettings** 回调方法，打开这些标识之一也会强制让设备进入软件处理。

OnFrameRender 回调方法设置适当的技术在**effect**.Technique属性，经过一个 EffectHandle（本案例是一个string）。有可能经过一个 EffectHandle取代string，会改进执行效率，因为这个很高频率的调用不是必须花费时间进行字符串比较。

OnFrameRender 方法然后设置技术要使用的变量，例如世界视图射影矩阵，它调用各种各样的**effect**.SetArrayRange。要渲染场景，**OnFrameRender** 调用返回pass的**effect**.Begin，pass的数目由技术需要决定，然后它循环通过每个调用**effect**.BeginPass(pass)的pass和用**mesh**.DrawSubset渲染网格。在每个pass之后，**OnFrameRender**调用**effect**.EndPass。当所有的pass被完成，**OnFrameRender**调用**effect**.End。

第2节　CustomUI 范例

这是 1 个通过托管代码的 Microsoft DirectX 9.0 范例框架完成的简单用户界面（UI）子系统示范。范例框架为大多数 DirectX 9.0 托管代码程序提供了代码和底层通用基础结构。框架的 1 个区域是用户界面支持。范例框架包含常用的控制对象，如按钮和检查框，窗口和全屏 Microsoft Direct3D 程序都能使用它们来完成用户界面。

2.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\CustomUI
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\csCustomUI.exe

2.2 范例是如何工作的

这个例子示范了使用托管代码的 DirectX 9.0 范例框架的程序是如何利用框架 UI 支持的。

控制类型	完成它的类
Statics	Microsoft.Samples.DirectX.UtilityToolkit.StaticText
Buttons	Microsoft.Samples.DirectX.UtilityToolkit.Button
Radio buttons	Microsoft.Samples.DirectX.UtilityToolkit.RadioButton
Check boxes	Microsoft.Samples.DirectX.UtilityToolkit.Checkbox
Combo boxes	Microsoft.Samples.DirectX.UtilityToolkit.ComboBox
List boxes	Microsoft.Samples.DirectX.UtilityToolkit.ListBox
Sliders	Microsoft.Samples.DirectX.UtilityToolkit.Slider
Scrollbars	Microsoft.Samples.DirectX.UtilityToolkit.Scrollbar
Edit boxes	Microsoft.Samples.DirectX.UtilityToolkit.EditBox

范例框架的所有控件与 Microsoft Windows 控件工作类似。它们是完全随意拖放生成并执行的，除了编辑框没有使用上面任何 Windows 控件。它们也由 Direct3D 进行渲染，因此可用于窗口和全屏 Direct3D 程序。

这个范例通过定义 2 个对话框对象（hud 和 sampleUi）开始。1 个对话框用作封装 1 个或多个控件的容器。它位于程序和控件之间，因此程序可以将消息和渲染调用传递给对话框，对话框将确保它的全部控件接收到消息并得到正确的渲染。控件自身被设计成 1 个并非单独的对话框。程序并不限于 1 个对话框对象。它们可以一次使用多个对话框，如这个范例所示。

当用户与 UI 控件交互时，控件能够将重要事件通报给范例，使得它可以正确响应。这时范例还初始化所需要的额外字体。这是通过 SetFont 方法完成的。

范例必须做的下一步骤是建立它需要的控件。这是通过调用诸如 AddButton 之类的对话框 AddXXX 方法完成的。该例中，hud 包含提供了标准功能的 3 个按钮，这些功能是 Direct3D 范例所具有的：在硬件和引用设备之间切换和改变 3D 设备设置。smpleUi 包含了剩余的

控件，它们是范例的详细类别。在 1 个控件加入后，它的外观可以改变。在范例中对改变静态控件和组合框的外观进行了示范。适当的事件在建立时被每个控件所绑定并在事件处理方法中进行响应。
下一步，在 OnResetDevice 中，是大量的对话框和控件的 SetSize 和 SetLocation 方法的调用。这是因为当用户调整程序窗口大小时，某些对话框和控件需要重新调整尺寸和定位。值得注意的是对话框位置与程序窗体有关，而 1 个控件的位置与它所属的对话框位置有关。要渲染用户界面，范例在它的 OnFrameRender 函数中调用了 2 个对话框对象的 OnRender 方法。对话框将以正确的顺序渲染它所包含控件。
要求 UI 完成的最后一步是将消息传递给控件。这让它们从用户那里接收键盘和鼠标事件。在 MsgProc 中，范例调用对话框 MsgProc 方法传递消息。如果对话框的 MsgProc 返回 true 指明消息已经被处理，范例将 noFurtherProcessing 参数设置为 true 并立即返回就是非常重要的。没有这样做的话将会导致消息被不止一次的处理。

2.3 事件

在用户与 UI 控件交互时，必然有诸如检查或不检查 1 个检查框、在组合框中选择 1 个新项目、在单选按钮群中选择 1 个新项目等等之类的事件发生。程序也许对这些事件感兴趣，并由控件事件通报。该例中，一系列事件处理函数被定义并当控件建立时被绑定。结果无论用户是否通过 UI 控件执行了感兴趣的操作，这些函数都将被调用。这些可以触发回调的操作是：

单击 1 个按钮
选择组合框中的 1 个新项目
选择单选按钮群中的 1 个新项目
改变 1 个检查框的检查状态
改变滑块的值
在编辑框中按下回车键

在事件处理句柄中，发出事件的控件和任何必需的额外数据对程序都是可用的。范例通过发生事件的控件返回内容和它在屏幕上的显示来对事件进行响应。

第3节　EmptyProject 范例

这篇范例是新版 Microsoft® Direct3D®应用程序的起点。

3.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\EmptyProject
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\csEmptyProject.exe

3.2 范例概览

这个例子是一个 Direct3D 应用程序的骨架，为你自己的工程提供了一个方便的起点。

3.3 范例是如何工作的

这个例子使用了范例框架来初始化并且运行一个基本的 Direct3D 应用程序。

第4节　EnhancedMesh 范例

这个例子示范了 Microsoft Direct3D 中的网格镶嵌。网格镶嵌对网格三角形进行了细分。这产生了 1 个具有微小几何细节的网格，即使采用逐顶点光照，它也可以产生更好的光照效果。网格镶嵌通常用于完成 1 个细节级别（LOD）技术，它对靠近观察者的网格采用较多细节进行渲染，对远离观察者的网格采用较少细节进行渲染。

4.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\EnhancedMesh
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\csEnhancedMesh.exe

4.2 范例是如何工作的

范例能够以 2 种模式中的 1 种运行：硬件镶嵌或软件镶嵌。用户可以将镶嵌级别设置为不同的值并发现网格的变化相对于级别的调整是如何起作用的。
当运行在硬件镶嵌模式下时，范例通过设置 Device.NPatchMode属性镶嵌网格，该属性将镶嵌片断的数量设置成设备为每个网格片断镶嵌的数值。比如，指定 3.0 将导致输入网格中的每个原始片断被镶嵌成 3 个片断。在渲染循环中的网格绘制调用语句之后，该镶嵌操作实时发生。
当运行在软件镶嵌模式下时，范例不依赖于硬件来完成高速镶嵌。所以范例必须处理网格并在渲染它之前获取所期望的细节级别。代码通过调用 Mesh.TessellateNPatches完成该操作，这个方法带有 1 个输入网格和 1 个片断数量，然后输出另 1 个新版本网格，它代表了已经过镶嵌的输入网格。然后范例就能够使用任何标准机制来渲染这个镶嵌过的网格。

第5节　FragmentLinker 范例

本例描述了如何使用 FragmentLinker类。着色器源代码可以分成一系列着色器片断，这

些片断被单独编译并连接起来形成 1 个完整的着色器。这个连接阶段非常高效，使它适合在运行时使用。这样，Microsoft Direct3D程序可以自定义生成 1 个适合当前图形卡的着色器。

5.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\FragmentLinker
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\csFragmentLinker.exe

5.2 范例概览

大规模 Direct3D 程序通常使用多组冗余着色器，它们中的每个都采用了可支持回退功能的容器代替某种特定技术（经常用统一参数生成）。在运行时它们将根据被选择的当前图形硬件来挑选合适的着色器。这种方法结果导致大量的编译过的着色器被包含到程序中来，在 1 台特定机器上仅有一小部分的着色器被使用到。使用着色器片断，可以在运行时生成当前图形卡所期望的着色器。

5.3 范例是如何工作的

本例中有 2 个顶点着色器片断，范例将其中 1 个用于处理投影（顶点绘制或统计）的片断和另 1 个用于处理顶点光照（仅启用漫反射或环境色）的片断连接在一起。这些片断在初始化期间一次编译然后在运行时连接在一起。

当在你的程序中使用着色器片断时，仅需要遵循下列很少的几个步骤：

1. 使用 FragmentLinker Constructor建立 FragmentLinker对象。
2. 通过调用以下的 FragmentLinker 方法来加载和编译着色器片断：GatherFragmentsFromFile, GatherFragmentsFromStream, 或 GatherFragmentsFromString。
3. 调用 FragmentLinker.AddFragments向连接器的内部列表加入编译过的片断。
4. 通过调用 FragmentLinker.GetFragmentHandle获取你想连接的返回片断句柄。
5. 通过将片断句柄传递给以下 FragmentLinker 方法：LinkVertexShader, LinkPixelShader, 或 LinkShader来把加入的片断子集连接到一起。这些方法返回 1 个对象，该对象包含了 1 个编译过的可以用于渲染的着色器。

在设备建立时期，下列范例程序在 OnCreateDevice 函数内部执行这些步骤：

```
        [C#]


private void OnCreateDevice( object sender, DeviceEventArgs e )

.
```

```
.
.
// Create the fragment linker interface
fragmentLinker = new FragmentLinker( e.Device, 0 );

// Compile the fragments to a buffer. The fragments must be linked
// together to form a shader before they can be used for rendering.
path = Utility.FindMediaFile( "FragmentLinker.fx" );
compiledFragments =
    Microsoft.DirectX.Direct3D.FragmentLinker.GatherFragmentsFromFile(
        path, null, null, ShaderFlags.None );

// Build the list of compiled fragments
fragmentLinker.AddFragments( compiledFragments );

// Store the fragment handles
ComboBox cb1 = sampleUI.GetComboBox( Lighting );
cb1.Clear();
cb1.AddItem( "Ambient",
    fragmentLinker.GetFragmentHandle( "AmbientFragment" ) );
cb1.AddItem( "Ambient & Diffuse",
    fragmentLinker.GetFragmentHandle( "AmbientDiffuseFragment" ) );

ComboBox cb2 = sampleUI.GetComboBox( Animation );
cb2.Clear();
cb2.AddItem( "On" ,
    fragmentLinker.GetFragmentHandle( "ProjectionFragment_Animated" ) );
cb2.AddItem( "Off",
    fragmentLinker.GetFragmentHandle( "ProjectionFragment_Static" ) );
```

```
// Link the desired fragments to create the vertex shader
LinkVertexShader();
```

最后的连接步骤在 FragmentLinker.LinkVertexShader方法中执行。除了在设备建立时期被调用之外，在用户改变片断组合框选择的任何时候该方法还被调用。当前选择的片断从用户界面控件返回值中获得，着色器的数据被更新。

```
          [C#]

private void LinkVertexShader()
{
    const int NumberFragments = 2;
    Device device = sampleFramework.Device;
    EffectHandle[] aHandles = new EffectHandle[NumberFragments];

    aHandles[0] =
        sampleUI.GetComboBox( Animation ).GetSelectedData() as
EffectHandle;
    aHandles[1] =
        sampleUI.GetComboBox( Lighting ).GetSelectedData() as
EffectHandle;

    if ( vertexShader != null )
    {
        vertexShader.Dispose();
    }

    string errors;
    using (GraphicsStream code = fragmentLinker.LinkShader(
        "vs_1_1", ShaderFlags.None, aHandles, out errors ) )
    {
        vertexShader = new VertexShader( device, code );
```

```
            constTable = ConstantTable.FromShader( code );
        }
        // Set the global variables
        device.VertexShader = vertexShader;
        if ( constTable != null )
        {
            constTable.SetValue( device, "g_vMaterialAmbient",
MaterialAmbient );
            constTable.SetValue( device, "g_vMaterialDiffuse", MaterialDiffuse );
            constTable.SetValue( device, "g_vLightColor", LightColor );
            constTable.SetValue( device, "g_vLightPosition",LightPosition );
        }
            }
```

第6节　HDRCubeMap 范例

本例示范了通过浮点纹理和高动态范围（HDR）光照进行立方体环境映射。在 Microsoft® DirectX® 9.0 中，纹理已经可以使用浮点格式。这些纹理可以存储大于 1.0 的色彩值。当材质吸收了部分光线时，这可以让环境映射上的光照效果更加具有真实感。
注意并不是所有的图形卡都可以支持环境映射和 HDR 光照技术全部特性的。

6.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\HDRCubeMap
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\csHDRCubeMap.exe

6.2 使用手册

这个例子中定义的常用控件：

关键字	动作
E,D	Adjust material reflectivity

W,S	Adjust light intensity
F	Change cubic map format
R	Reset lighting parameters
N	Change to another mesh
F2	Change device
Esc	Quit

6.3 范例概览

HDRCubeMap 展示了 2 种渲染技术：立方体环境映射和 HDR 光照。立方体环境映射是这样 1 种技术：将围绕 1 个 3-D 对象的环境渲染到 1 个立方体纹理贴图中，使得对象可以使用立方体贴图而不用花费昂贵的光照计算开销来完成复杂的光照效果。HDR 光照是这样 1 种技术：通过使用浮点纹理和高亮度光线来渲染非常真实的光照效果。浮点纹理格式在 DirectX 9.0 中介绍。与整数格式的传统纹理不同，浮点纹理可以存储 1 个广阔范围的色彩值。因为浮点纹理中的色彩值不会被限制在[0，1]中，和真实世界中的光线非常相似，这些纹理常可用于达到更高的真实感。
HDRCubeMap 渲染的场景是 1 个环境映射网格和许多围绕该网格补上环境的其他对象。网格具有 1 个从 0 到 1 的可调整反射率，0 意味着光线完全被吸收，1 意味着没有吸收光线并全部反射。在场景中使用的光线由 4 个点光源构成，它们的亮度可由用户调整。

6.4 执行

HDRCubeMap 遵循了 DirectX 范例框架的架构。
范例渲染的场景由这些对象组成：

1 个室内 Mesh对象。这包含了墙体、地板和天花板。
4 个代表光源的小球体 Mesh对象。
位于室内中央并应用了环境映射贴图的 1 个 Mesh对象。
2 个双翼飞机 Mesh对象。这些轨道围绕环境映射网格并在环境映射贴图中被反射。它们可视化的暗示出了环境映射是动态的，并随着周围环境的改变而变化。

当范例加载时，它建立了 2 个立方体纹理，1 个是 **A8R8G8B8** 格式，另 1 个是 **A16B16G16R16F** 格式。范例在渲染时，基于用户的选择使用其中的 1 个来构造 1 个环境映射贴图。当使用 HDR 光照时，这被用来展示整数和浮点纹理之间的视觉差异。1 个尺寸等于 1 个立方体纹理表面的的模版表面也在同时被建立。当范例将场景渲染到立方体纹理上时候，该模版表面将被用作模版缓存。范例还建立了 1 个包含了所有着色器和渲染场景所需要技术的特效对象。
范例中使用的照相机是 1 个 ModelViewerCamera。这个照相机总是看向环境映射网格所在的原点。用户可以围绕网格移动照相机的位置并通过鼠标控制来旋转网格。照相机对象为渲染计算出视点、投影矩阵和环境映射网格的世界矩阵。

6.5 渲染代码

范例的渲染发生在 3 个函数中：**RenderSceneIntoCubeMap()**，**OnFrameRender()**，和 **RenderScene()**，使用了顶点和像素着色器。

RenderScene()是实际将场景渲染到当前渲染目标上的函数。它被调用同时将场景渲染到立方体纹理和设备后台缓存上去。它带有 6 个参数：用于渲染的设备、1 个视点矩阵、1 个投影矩阵、当前被选择渲染模式的技术群、程序的当前时间、1 个指明是否应该渲染环境映射网格的标识。需要该标识是因为在构造环境映射贴图时，没有绘制环境映射网格。该函数的剩余部分很直观。它首先通过最新的变换矩阵来更新特效对象和视点空间中的光源位置。然后它为每个网格使用 1 个合适的技术渲染场景中的每个网格对象。

RenderSceneIntoCubeMap() 的任务是将整个场景（负的环境映射网格）渲染到立方体纹理上去。首先，它保存当前渲染目标和模版缓存，并为立方体纹理设置模版表面作为设备的模版缓存。下一步，函数声明立方体纹理的 6 个面。它为每个面设置合适的朝向表面作为渲染目标。然后它计算出视点矩阵用于特定的面，在面向立方体面的原点处放置照相机。然后它调用 **RenderScene()**，将 **bRenderEnvMappedMesh** 设置为 false，并沿着计算出的视点矩阵和 1 个特殊的投影矩阵来传递。这个投影矩阵具有 90 度视野和值为 1 的宽高比，这是由于渲染目标是 1 个正方形。在全部 6 个面都完成这个过程之后，函数恢复原来的渲染目标和模版缓存。现在就为该帧完全构造好了 1 个环境贴图。

OnFrameRender()是通过范例框架每帧被调用一次的最高级别渲染函数。它调用 **RenderSceneIntoCubeMap()**构造立方体纹理以映射该帧中的环境。在这之后，它使用由照相机得来的视点和投影矩阵并通过调用 **RenderScene()** 来渲染场景，将 **renderEnvMappedMesh** 设置为 true。

6.6 着色器

本例中的所有渲染都是通过可编程着色器完成的，它们被分成 3 种特效技术：**RenderLight**，**RenderHDREnvMap** 和 **RenderScene**。

RenderLight 技术用于渲染表示光源的球体。顶点着色器进行 1 个常用的世界-视点-投影变换，然后将光线亮度分配给输出漫反射。像素着色器将漫反射传播给流水线。

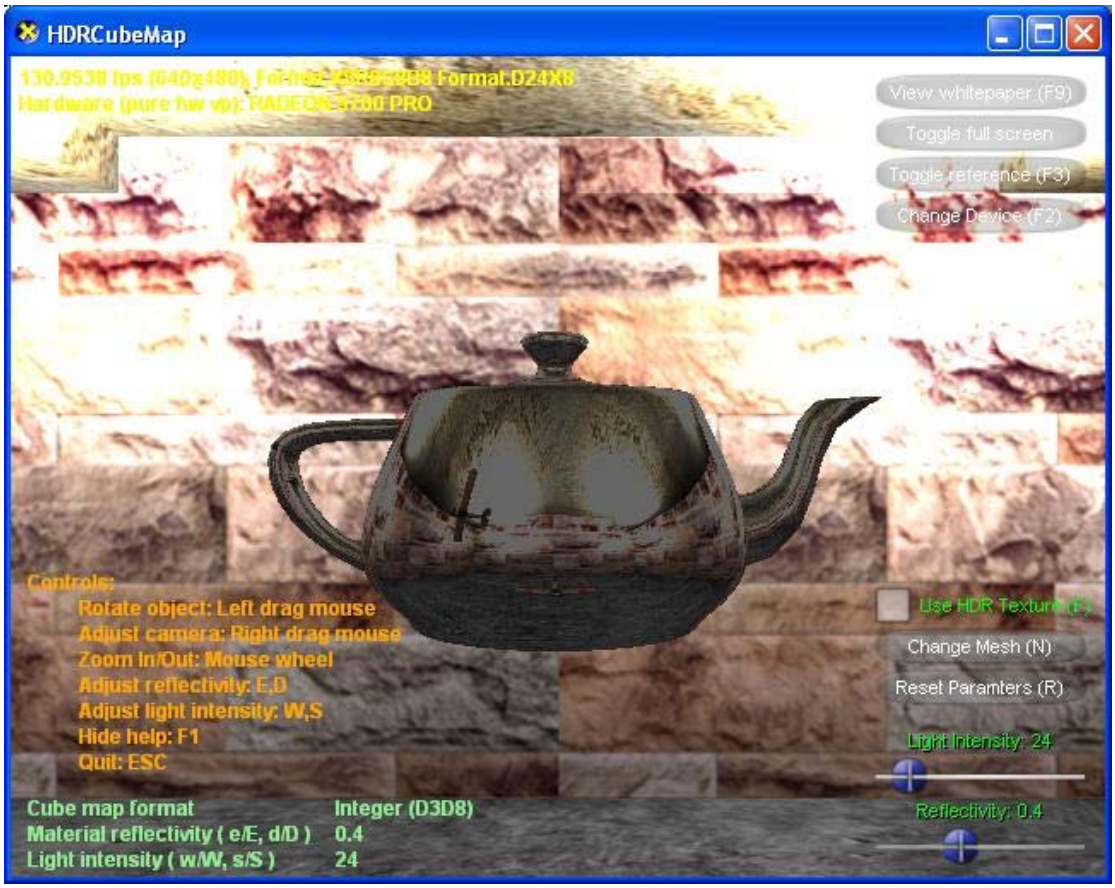
RenderHDREnvMap 技术用于渲染场景中的环境映射。除将位置从对象空间变换到屏幕空间，顶点着色器计算在视点空间中眼睛反射向量（眼睛到顶点的向量的反射）。该向量被作为 1 个立方体纹理坐标传递到像素着色器。像素着色器抽取了立方体纹理，将反射率应用到抽取值上，并将结果返回给流水线。

RenderScene 技术用于渲染其他任何东东。由该技术渲染的对象使用了逐像素的漫反射光照。顶点着色器将位置从对象空间变换到屏幕空间。然后它计算视点空间中的顶点位置和法线，将它们作为纹理坐标传递给像素着色器。像素着色器使用该信息来执行逐像素光照。1 个 for 循环用于计算从 4 个光源中返回的光线总数。像素的漫反射项目通过获取法线的点乘积和像素到光线的单位向量来计算。衰减项目作为像素和光线之间距离的平方的倒数来计算。这 2 个项目然后被调整来表述像素从光源中的 1 个所接收到的光线数量。一旦场景中的所有光线都完成了这个过程，数值被累加并通过光线亮度和纹素调整以形成输出。

6.7 高动态范围真实感

在 DirectX 9.0 之前，Microsoft Direct3D 仅支持整数纹理格式。当使用了这种类型的 1 个纹理时，如果 1 个色彩值高于 1.0，它就被在被写入前限制为 1.0，使得该值可以适用于 1 个 8 位整数值。这意味着当立方体纹理被写入时，它的纹素色彩值将会有 1 个最大值 1.0。当后来在 1 个不能 100%反射的材质（可能仅有 30%被反射）上使用这个纹理用作环境映射贴图时，即使纹理中最大亮度的纹素在网格中也将呈现为 0.3，而不管它原始光线亮度有

多高。因此，真实感打了折扣。下图示范了这种情况。



当使用 DirectX 9.0 中光线亮度高于 1 的新型浮点立方体纹理时，纹理可以存储高于 1.0 的色彩值，因此即使反射率增加，也可以保留真实的亮度。下图展示了这种情况。

6.8 折中选择

在 1 个 HDR 光照环境中不使用浮点立方体纹理来表现 1 个高度的真实感是有可能的。程序可以在构造立方体环境映射贴图时应用材质反射率乘法器来完成。然后在渲染期间，像素着色器将抽取立方体纹理并将数值直接向后面的流水线输出。这种方法通过比例缩放的方式围绕色彩夹钳限制运作，它应用在夹钳限制之前而非之后。这种方法的缺点是如果场景具有多个不同反射率的环境映射贴图对象，这些每个对象一般都需要用 1 个不同的环境贴图来渲染。

第7节　HLSLwithoutEffects 范例

这个范例展示了 1 个如何使用Microsoft® Direct3D®高级着色器语言（HLSL），而不使用D3DX特效界面。不使用特效界面而使用HLSL是 1 种更加困难的方法。详见为 1 个使用HLSL简单方法准备的 BasicHLSL Direct3D Sample。

7.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\HLSLwithoutEffects
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\csHLSLwithoutEffects.exe

7.2 范例概览

例子示范了使用 HLSL 编写出顶点着色器而不使用 D3DX 特效界面。HLSL 是 1 种接近于类似 C 语法和构造方法的语言。通过用 HLSL 代替汇编语言编写着色器，开发者不仅可以利用他们已经熟悉的语言特性和元素，而且可以利用 D3DX 着色器编译器提供的强大优化功能。

7.3 范例是如何工作的

这个范例渲染的场景由依赖于XZ位面的三角形的 1 个方格组成。在每帧中，这个格子的顶点将基于它们到原点的距离函数和时间函数，沿着Y方向上或下移动。顶点还使用了另 1 个距离和时间的函数被照亮。时间随每帧而增长。因为顶点的Y坐标和色彩在每帧中都被生成，所以它们不需要被存储。因此顶点声明仅包含了 1 个 Vector2 成员，该成员包含了X和Z坐标。
在初始化期间，范例调用 CompileShaderFromFile将范例存储在 1 个文件中的着色器函数编译为 3-D设备能够理解的二进制着色器代码。在这个过程之后，1 个包含了着色器代码的 GraphicsStream对象和 1 个允许程序读取并设置着色器全局变量的 ConstantTable对象将被建立。然后，范例使用着色器代码建立了 1 个新的顶点着色器来获取 1 个能够在执行期间传递给 3-D设备的 VertexShader对象。
在**OnFrameMove()**，为了更新着色器的视点+投影变换和时间参数，范例使用从初始化过程中获得的 ConstantTable来设置着色器全局变量*worldViewProj* 和 *appTime*。在渲染时，范例将 VertexShader属性设置为启用顶点着色器在设备上渲染。当这之后调用了 DrawIndexedPrimitives，顶点着色器将被每个处理的顶点调用 1 次。
范例中，顶点着色器编写在 1 个叫做 HLSLwithoutEffects.fx 的文本文件中。在这个文件中，有 2 个全局变量和 1 个叫做 Ripple 的着色器函数。Ripple 带有 1 个 **float2** 作为输入值（为顶点的 X 和 Z），并输出 2 个描述屏幕空间位置和顶点色彩的 float4。是用这个公式产生 Y 坐标：

```
Y = 0.1 * sin( 15 * L * sin(T) );
```

L 是在 Y 调整之前顶点和原点之间的距离，因为顶点依赖于 XZ 位面，所以它是(X^2 + Z^2)平方根的值。T 是 appTime 全局变量。
顶点色彩将基于这个公式具有某些灰度阴影：

```
C = 0.5 - 0.5 * cos( 15 * L * sin(T) );
```

C 是为红、绿、蓝和 alpha 全部使用的值，范围从 0 到 1，分别将给出顶点色彩从黑色到

白色。结果看上去有点像在狭窄和宽阔处之间来回改变宽度的波纹，并基于倾斜程度被适当着色。

第8节　ProgressiveMesh 范例

这个例子展示了 1 个程序是如何使用 D3DX 渐进网格功能来简化网格以快速渲染的。渐进网格是 1 个专门网格对象，它可以增加或减少它的几何复杂度，因而在绘制网格时提供了灵活性使得性能可以维持在 1 个稳定的水平。在程序中提供了细节级别（LOD）时候，这个特性是有用的。

8.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\ProgressiveMesh
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\csProgressiveMesh.exe

8.2 范例是如何工作的

渐进网格的功能由 ProgressiveMesh对象提供。这个 ProgressiveMesh对象类似于增加了管理网格复杂度方法的 Mesh对象。渐进网格可以和 1 个常规网格一样使用。要渲染它，范例在它的材质循环并调用 DrawSubset将几何体子集发送给设备。要调整渐进网格的 LOD，范例将 ProgressiveMesh.NumberVertices属性设置为所期望的顶点数量。1 个渐进网格将简化或提升它的几何特征来尽可能的匹配顶点数量。

范例还展示了一种通过整理多重渐进网格来对渐进网格进行优化的技术。整理限制了 1 个渐进网格可以具有的最小和最大顶点或表面数量。范例将渐进网格顶点的范围（最大-最小）分成 10 个子范围。在子范围被计算后，范例通过调用 ProgressiveMesh.Clone在原始网格上建立 10 个渐进网格。然后范例在每个使用不同子范围的渐进网格上调用 ProgressiveMesh.TrimByVertices 。在设置好顶点范围之后，范例调用 ProgressiveMesh.OptimizeBaseLevelOfDetail来优化网格顶点和索引缓存。无论用户何时改变顶点数目，新的顶点数目都要被优化的渐进网格所设置的范围进行检查，那些范围包含了所期望顶点数目的网格将被选择 ProgressiveMesh.NumberVertices 属性。

使用多重渐进网格的好处在于调整 LOD 比使用 1 个单独的渐进网格更有效率。改变 LOD 的性能负载和复杂度的差异（由本例中的顶点数目描述）直接成比例。通过减少顶点数量到 10 比减少顶点数目到 100 花费的时间更少可以简化 1 个网格。这就是为什么通过整理某些覆盖了 1 个更小 LOD 范围的渐进网格，本例可以达到更好的性能。

第9节　PrtPerVertex 范例

本例示范了如何使用 PrtEngine，1 个使用了低次序球调和函数（SH）的预计算次表面散射（PRT）模拟器。本例还示范了如何使用这些结果来完成动态光线传递，它使用了 1 个动态光照环境和 1 个**vs_1_1** 顶点着色器。

9.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\PrtPerVertex
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\csPrtPerVertex.exe

9.2 为什么这个例子很有趣？

使用低次序 SH 基础函数的 PRT 比典型的漫反射(*N* * *L*)光照具有更多的优势。诸如相互反射、软阴影、自身阴影和子表面散射之类的区域光源和全局特效可以通过预计算次表面散射模拟之后被实时渲染。聚簇主要组件分析(CPCA)允许模拟器的结果被压缩，因此着色器不需要一样多的常量或逐-顶点数据。

9.3 范例概览

基本概念是首先运行 1 个 PRT 模拟器脱机作为艺术加工内容建立过程的一部分并为后期实时使用来保存压缩过的结果。光线传递模拟器模仿通常实时环境下非常困难的全局特效。实时引擎根据 SH 基础函数估计光线并将光线累加到 1 组单独的 SH 基础系数上，该系数描述了整个光照环境。然后它通过组合压缩模拟器结果和光照环境，使用 1 个顶点着色器来达到顶点的漫反射色彩。因为脱机模拟器通过计算相互反射和软阴影来完成这项工作，所以该技术在视觉上是真实深刻且高效的，并可以用于实时光照。

9.4 这个例子是如何工作的

范例执行 PRT 的脱机和实时部分二者。启动对话框询问用户执行哪一个步骤。用户可以运行脱机模拟器或者看见 1 个网格，该网格使用了前面从 PRT 模拟器得来的保存结果。脱机步骤通常在 1 个独立的工具中完成，但是这个范例在相同的可执行文件中都完成了。

9.4.1 步骤 1：脱机处理

第一个步骤是在源文件*PRTSimulator.cs*中运行脱机逐顶点PRT模拟器。该代码接受了大量参数来控制操作模拟器、1 个网格数组和 1 个 SphericalHarmonicMaterial结构体数组。

每个网格允许 1 个材质，因此假定每个网格是同一个坐标系的。模拟器的输入参数和SH材质结构体的成员普遍由范例对话框工具提示来解释。如果你想传入 1 个以上的网格，那么在你传递之前网格就需要被变换为相同的坐标系空间。
大多数模拟器输入参数不影响结果如何使用。特别的，范例的 *Order* 参数通过指定常用于近似次表面散射的 SH 基础函数的顺序，确实影响了如何使用结果。
除了 Order 参数之外，光谱参数(spectralCB) 也影响了结果。如果选择了光谱模拟，就会有 3 种色彩通道-红绿蓝。然而，有时仅使用 1 种通道（比如当对阴影建模时）工作是有用的。如果你选择无光谱模拟，当你调用 SH 函数时，就简单使用了红色通道，因为其他通道是可选项。
模拟器将运行一段时间，通常是几分钟，这取决于网格的复杂度、射线的数量和模拟设置。输出是1个GraphicsStream对象，它为网格的每个顶点都包含了1个内部标题和1个float值类型的数组。
每个顶点的 float 值类型叫做次表面散射向量，可以通过 1 个顶点着色器用于将源散射变换为离去散射。然而，既然每个通道都有 $Order^2$ 转换系数，那么带有光谱的并且 *Order* = 6 就会是每顶点 3 x 36 或者 108 个标量。幸运的是，你可以通过 CPCA 算法压缩大量的标量。 每顶点系数的数量将被降低至主要组件分析(PCA)向量的数量。对好的结果来说，这个数量不需要被夸大。比如，4 个或 8 个通常适合于好的结果。比如，如果有 8 个 PCA 向量和 *Order* = 6，然后你将仅需要 8 个系数/每顶点而不是 108 个。PCA 向量的数量必须小于 $Order^2$

9.4.2 步骤 2：实时渲染

渲染压缩 PRT 数据的方程式为：

$$R_p \approx (M_k \bullet L') + \sum_{j=1}^{N} w_{pj} (B_{kj} \bullet L')$$

参数	描述
R_p	1 个在顶点 *p* 处离去散射的简单通道并被在网格的每个顶点上求值。
M_k	簇数目 *k*.的平均数。这是 1 个系数的 $Order^2$ 向量。
k	顶点 *p* 的簇验证（ID）。
L'	进入 SH 基础函数的源散射近似值。这是 1 个系数的 $Order^2$ 向量。
j	1 个将 PCA 向量数量累加的整数。
N	PCA 向量的数量。

w_{pj}	点 *p* 的第 *j* 个 PCA 重量。这是 1 个单独的系数。
B_{kj}	簇 *k* 的第 *j* 个 PCA 基础向量。这是 1 个系数的 Order^2 向量。

范例的源文件 *PRTMesh.cs* 收集了这个方程式需要的所有数据并将近似值传递给 1 个执行方程式的顶点着色器，如下所示：

1. 范例的**prtMesh.LoadMesh**方法常用于加载网格，但既然CPCA需要每个顶点的 (PrtCompressedBuffer.NumberPcaVectors + 1) 标量，通过 1 个提供了足够内存来存储该数据的 VertexElement声明克隆了网格。另外还需要为顶点着色器提供 1 个对聚簇数据的索引。范例使用 BlendWeight常量值来存储CPCA数据，但是这个语义是任意的并且被选择是因为蒙皮操作和PRT不能一起工作。通过 VertexElement构造函数对输入顶点数据进行初始化过程中，usageIndex = 0，范例将 DeclarationType定义为 Float1 并使用它来存储 1 个对常量数组的索引。用法上索引从 1 到 6，声明类型为 Float4。范例因此可以在顶点缓存中存储 24 个PCA重量。
2. 范例从 1 个文件载入模拟器的SH PRT结果并将这个数据放入 1 个 PrtCompressedBuffer对象中。范例的**prtMesh.CompressBuffer** 方法然后调用 PrtCompressedBuffer构造函数来应用CPCA，它使用了某些数量的PCA向量和某些数量的簇。输出是 1 个叫做**prtCompBuffer** 的 PrtCompressedBuffer 对象，它包含了上面方程式所需要的数据。依靠**ReloadState** 方法你可以选择PCA向量和簇的数量而不再次运行模拟器。
3. 范例首先通过调用 PrtCompressedBuffer.ExtractClusterIDs来获取定顶点的簇ID从**prtCompBuffer** 中提取CPCA数据。这个方法对 1 个 Int32 值类型的数组写入，该数组中顶点N的簇ID是 clusterIds[N]。在顶点缓存中将为每个顶点计算并存储 1 个数组偏移量。顶点着色器使用该偏移量并允许它直接对当前顶点簇的数据进行索引。该偏移量是由CPCA数据填充的常量数组的步幅和簇ID的简单倍数。
4. 范例调用 PrtCompressedBuffer.ExtractToMesh 并使用 usage = BlendWeight 和 usageIndexStart = 1 来指示方法存储网格中每顶点的PCA重量，从语义上开始于 BlendWeight[1]，继续 BlendWeight[2]等等，直到所有的PCA重量都已经被写入。既然程序已经定义 BlendWeight索引从 1 到 6 作为float4类型，那么如果有 20 个PCA向量，这个方法将写入 BlendWeight索引从 1 到 5。注意这些顶点元素不必是float4 类型的；它们仅需要是有符号的。
5. 和渲染方程式展示的一样，要计算着色器中的离去散射，你需要逐顶点压缩的传递向量和光照环境（也叫做源散射），它们是使用SH基础函数求值的。SphericalHarmonics类的某些方法对帮助这个步骤也是可用的。
    - EvaluateDirectionalLight
    - EvaluateSphericalLight
    - EvaluateConeLight
    - EvaluateHemisphereLight
    - ProjectCubeMap

你可以使用这些函数中的 1 个，为每个光照每个通道获取 1 个Order^2 float 数组的输出值。然后使用 SphericalHarmonics.Add将这些数组一起加入以为每通道达到 1 组单独的 Order^2 SH系数，该系数描述了场景的源散射，即渲染方程式中的L'。注意这些方法在对象空间中带有光线的方向，因此你通常必须通过翻转世界矩阵来变换光线的方向。

6. 范例从ExtractToMesh方法所需要的最后数据片是簇的平均数(M)和PCA基础向量(B)。范例在 1 个巨大的float值类型数组中存储这个数据使得当光线改变时它可以对光线再次求值并重新计算M * L and B * L。要完成这些，它调用了PrtCompressedBuffer.ExtractBasis ，这每次提取 1 个簇的基础。每个簇的基础由 1 个平均数和PCA基础向量组成。因而存储所有簇基础所需要的数组尺寸大小是。

```
7.      int clusterBasisSize = (numberPcaVectors + 1)
                                * numberCoefficients * numberChannels;
```

注意 PCA 向量的数量加 1 以存储簇平均值。还要注意既然(M_k * L') 和 (B_{kj} * L') 都是常量，范例在 CPU 上计算这些值并将它们作为常量传递给顶点着色器。范例在顶点缓存中存储这个逐-顶点的数据。

8. 范例的 PrtMesh.ComputeShaderConstants 方法使用渲染方程式执行 M * L 和 B * L 计算，并在 prtConstants 数组中存储计算好的 CPCA 常量值，该数组定义如下：

```
9.      prtConstants = new float[ numberClusters
                        * (4 + numberChannels * numberPcaVectors) ];
```

这个数组通过 Effect.SetValue 方法直接传递给顶点着色器。注意顶点着色器使用 float4 因为每个寄存器可以存放 4 个 float 值，因此装有数组的顶点着色器尺寸为：

```
int numberVConsts = numberClusters
                        * (1 + numberChannels * numberPcaVectors / 4) + 4;
```

对光线求值，计算和设置常量表是快速步骤，每帧可以完成 1 次或多次，但是出于优化目的范例仅在光线移动时对光照效果求值。

10. 现在范例已经提取了它所需要的全部数据，它可以通过 CPCA 使用 SH PRT 渲染场景。渲染循环使用特效文件 *PRTPerVertex.fx* 来渲染场景。范例的顶点着色器技术 **PrtDiffuseVS** 执行了渲染方程式以满足离去散射。

9.5 限制条件

因为这个技术使用了低次序球调和函数，光照环境被假定为是低频率的
传递向量被预先计算，因此预先计算的场景相关空间关系不能改变。既然刚性操作不会改变传递向量，所以 1 个网格可以被旋转、平移或缩放。然而，如果网格被毁形或蒙皮，那么渲染就会错误。相同的逻辑还应用到由许多网格组成的场景。比如，如果你将 1 个具有 3 个网格的场景传递给模拟器，实时引擎可以把它们作为 1 个整体旋转、平移和缩放，但是它不能脱离其他网格而旋转 1 个单独的网格，这会产生错误的渲染。
为了通过这个基于顶点技术进行准确渲染，网格必须被高度镶嵌。然而，基于顶点的技术可以运行在 **vs_1_1** 硬件上，虽然基于纹理的技术需要 **ps_2_0** 硬件。
如果你把具有离开其他表面就无法散射的子表面进行混合的话，那么你也许需要为被网格散射的子表面缩放传递系数，因为散射光线通常要暗淡 3 倍。你可以简化对

1 个单独网格缩放投影光线系数。你可以在压缩数据前通过使用 PrtEngine.ScaleMeshChunk缩放传递系数。

9.6 图像资源

(*SDK root*)\Samples\Light Probes

第10节 C# Scripting 范例

例子示范了在你的非托管程序中使用 C#代码作为 1 种"脚本"的可能技术，还包含了一种技术可以确保被认为非"安全"的脚本不能执行。例子包含了 3 个脚本，1 个简单的"无状态"脚本，1 个维持状态并对环境有反应的更高级的脚本，最后还有 1 个模拟从你的硬盘上删除文件的"恶意"脚本。

10.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\Scripting
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\Scripting.exe

10.2 范例概览

在你的游戏标题中具有 1 个强有力的脚本引擎，可以为你的标题开发出许多新方法。游戏外观上的许多差异可以通过这些"脚本"控制（虽然这些东东实际将被编译并执行，但"脚本"一词并不完全准确）。你可以在游戏内部、游戏 AI 或者游戏环境的许多方面，为你的照相机制作 1 个"分镜头"脚本。更加让人兴奋的前景之一就是让用户添加内容（比如自定义单元），这要使用脚本来进行自定义动作并对环境产生恰当的反应。

这个特殊例子产生于 3 种不同类型的脚本，每个脚本都有 1 个独特的特性：

脚本 #1 - 1 个简单的无状态脚本，当旋转时让角色围绕场景缓慢移动。

脚本 #2 - 1 个维持状态并使用该信息来决定执行何种动作的高级脚本。在旋转发生改变时角色可以弹过墙去。

坏孩子黑客脚本 - 这个脚本设计用于模拟"坏孩子黑客"尝试从你的计算机删除文件。如该脚本不允许运行，应归咎于被允许运行脚本上的安全设置。

10.3 范例是如何工作的

这个例子使用了可能是最直接的技术将 C#代码高速编译为脚本。它开始使用了 ShadowVolume C++软件开发套件（SDK）的范例，并作了改变允许 C#脚本控制场景中的角色。首先通过加入/CLR 编译器切换，范例被更新为 1 种混合编译模式。这让你可以

有托管代码来控制脚本引擎，该引擎被直接嵌入到你的程序中。
你会注意到范例的非托管代码文件也包含了 1 个新的#pragma 非托管指示，让编译器知道这段代码仅为非托管代码。在启动时安全策略被设置，然后当运行时允许用户从装配了 SDK 的 3 种脚本中选择 1 个（或者根本不选择脚本）。
当从用户界面中选择了脚本中的 1 个，编译和加载脚本代码的工作就发生了。CSharpCodeProvider类的 1 个实例被建立，脚本的陌生代码被填入该类中进行编译。汇编脚本是在临时文件夹中进行编译和存储的，然后临时文件夹中的内容被载入到程序中。一旦汇编内容被载入，脚本的主类（该类是所有脚本必须执行的）就被加载和存储；之后从脚本中将调用方法来使用它。
注意：本例中，这些脚本中的 1 个任何编译错误都会被忽略，并且角色的行为表现很简单，似乎和没有脚本运行一样。
一旦 1 个脚本被编译并载入，4 种潜在方法将会被连续调用（这取决于脚本是否执行了它们）。角色的旋转（沿每个轴）和角色的位置一样可以改变。

10.4 安全考虑

退一步说，允许未知代码在你的游戏引擎中运行会引起很大的惊慌。虚构一下：将你的重要文档通过电子邮件寄给你的全球地址簿中每个人，然后格式化你的硬盘？这恐怕将会阻止脚本作者决定写 1 个这样的脚本。C#作为 1 种语言功能非常强大并允许你在完全信任的环境下做出这些事情中的任何一种。
幸运的是托管代码有一种东东叫做代码访问安全，这是专门设计用于上述情况的。默认运行在本机上（大多数客户端程序都是）任何代码都被认为是完全信任的，这基本上意味着允许代码做出语言能力所及的事情来（像格式化硬盘）。要确保不允许这些非安全操作，应用程序域的安全策略要被更新。在启动期间，范例在脚本引擎上调用 **Initialize()**方法，该引擎包含了下列代码片断：

```
// Create a new, empty permission set so we don't mistakenly grant some
permission we don't want

PermissionSet* pPermissions = new PermissionSet(PermissionState::None);



// Set the permissions that you will allow, in this case we only want to allow
execution of code

pPermissions->AddPermission(new
SecurityPermission(SecurityPermissionFlag::Execution));



// Make sure we have the permissions currently

pPermissions->Demand();



// Create the security policy level for this application domain

PolicyLevel* pSecurityLevel = PolicyLevel::CreateAppDomainLevel();
```

```
// Give the policy level's root code group a new policy statement based

// on the new permission set.

pSecurityLevel->RootCodeGroup->PolicyStatement = new
PolicyStatement(pPermissions);



// Update the application domain's policy now

AppDomain::CurrentDomain->SetAppDomainPolicy(pSecurityLevel);
```

上面代码使用 PermissionState::None 许可状态首先建立 1 个 PermissionSet，该状态本质是"不允许这段代码做任何事"。显然这限制有点儿太严格了，既然你有一点儿想让运行脚本的代码被执行。你可以向这个允许列表加入 1 个新的许可，和你在代码片断中看到的一样，SecurityPermissionFlag::Execution 被加入了（以这种方式可以设置相当多不同的安全许可）。在 PolicyLevel 建立后，RootCodeGroup 被更新，应用程序域的安全策略被更新。在该点之后（比如稍后的脚本）任何被加载的汇编内容将强制具有这种安全策略。注意当你运行范例时，最后的脚本运行失败就归咎于它违反了安全性。

10.5 性能考虑

本例中，你会注意到为了执行脚本方法调用了 **InvokeMember()**方法。既然该方法执行很快并运行正常，这种技术也就依赖于对该调用的反应，该调用会比直接调用方法慢 1 个数量级。和这个范例做的一样，在游戏关卡中你每帧进行了 1 次（或多次）调用，你可能想以 1 个更有效率的方式来调用这些方法。

有一种更加容易的执行方式是定义 1 个脚本必须执行的界面，它比这种技术具有更好的性能。你可以把这个界面放入 1 个托管的汇编块中，然后你可以在程序中向这个块加入 1 个引用。当编译脚本时，你可以获得在托管汇编块中声明的界面类型并直接调用方法。这会删除 **InvokeMember()**方法的开销，而替代让你直接调用进入编译过的脚本中。

10.6 执行考虑

也许对你的程序来说像这个范例所做的一样，通过/CLR 切换到混合编译模式（它包含了托管和非托管代码二者）下是不可能（或不实用的）。你有许多其他方法来执行范例所达到的类似技术：

同时包含脚本引擎代码（就像本例的）和非托管代码将调用的非托管互通层的独立混合编译模式。

独立的纯托管汇编（比如 1 个 C#的汇编），它包含了脚本引擎代码，其中的类被标记为[ComImport]。该方法需要更多的设置时间，因为面向组件模型（COM）需要使用经由 RegAsm 工具注册的托管汇编。

要求脚本代码自身执行[ComImport]属性。这显然是最糟糕的游戏关卡。

第11节 SimpleAnimation 范例

这个例子展示了一个有骨骼结构的 3-D 模型怎样成为动画。

11.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\SimpleAnimation
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\csSimpleAnimation.exe

11.2 范例概览

这个例子展示了怎样在一个网格层次访问骨骼和皮肤信息，并且渲染成为动画的网格来展示。

11.3 范例是如何工作的

Sample Framework范例框架被用来初始化和运行一个简单的Microsoft® Direct3D®动画程序。

第12节 Text3D 范例

这个例子展示了怎么样在 3-D 场景里绘制 2-D 文本。对于展示统计表和游戏菜单是很有用的。

12.1 路径

源文件	(*SDK root*)\Samples\Managed\Direct3D\Text3D
可执行文件	(*SDK root*)\Samples\Managed\Direct3D\Bin\x86\csText3D.exe

12.2 编程须知

例子使用了 Font类来展示 3-D场景里的 2-D文本。这个类使用了Microsoft® Windows®

Graphics Device Interface (GDI)来加载一个字体并且输出每个字母到位图。那些位图依次地被用来创建纹理。
当 **DrawText** 函数被调用，一个顶点 buffer 缓冲器由多边形所填充这些多边形纹理使用了前面描述所创建的的字体纹理。这个多边形也许被绘制成一个 2-D 覆盖图，这也许对打印统计是有用的。
这个例子也建立了一个文本字符串的 3-D 网格并且渲染之。

第2篇 DirectSound

Microsoft® DirectSound® 提供了一个系统来捕捉来自输入设备的声音和播放那些贯穿各种各样的使用先进的 3-D 位置特效的回放设备的声音，并且过滤回声、扭曲、反射和其它特效。下面的标题被介绍：

Playing Sounds	这部分是通过 buffer 缓冲器播放 WAV 声音的引导。
3-D Sound	使用 DirectSound，你能在空间定位声音和应用多普勒移位来移动声音。3-D 特效被应用于个别的 DirectSound buffer 缓冲器。全局参数被设置在称为听众的对象上。
Using Effects	DirectSound 通过 Microsoft DirectX® 媒体对象(DMOs)提供了特效处理支持。
Capturing Waveforms	DirectSound 允许你捕捉来自麦克风或其它输入到声卡，为了立即回放或在文件贮存。数据能被捕捉在脉冲编码调制(PCM)或压缩格式。
Optimizing Performance	这部分提供某些多彩的技巧来改善那些直接在 DirectSound buffer 缓冲器播放它们的音频数据程序的执行效率。

更多信息参看 Microsoft.DirectX.DirectSound 托管代码参考文档。

第1章 声音播放

这个部分是通过 Microsoft® DirectSound® buffer 缓冲器播放 WAV 声音的导学。
相关主题
使用特效

第1节 回放概览

DirectSound设备对象设备对象描绘一个声音回放的 Device，并且通常用来管理设备和建立声音的buffer缓冲器。
很多应用程序为同样的声音设备，能建立 Device 对象。当输入焦点在应用程序间变化时，

音频输出自动从一个应用程序流转换到其它的流。结果，应用程序不必重复的播放或停止它们的buffer缓冲，当这种输入焦点变化的时候。
主 buffer 缓冲器保存读者将要听到的音频。次要声音 buffer 缓冲器各自包含一个单一的声音或音频流。DirectSound 自动建立主 buffer 缓冲器，但是次要 buffer 缓冲器的建立却是应用程序的职责。当声音们在次要 buffer 缓冲器被播放，DirectSound 在主 buffer 缓冲器混合它们，并且把它们发送到输出设备。只有可获得的处理时间限制 DirectSound 能混合的 buffer 缓冲器数量。
buffer缓冲器能被 Buffer类也能被 SecondaryBuffer类描绘。很多应用程序只使用继承自 Buffer类的 SecondaryBuffer类。可是一个描绘为主buffer缓冲器的对象必须是作为一个 Buffer对象来声名。
一个短音 可以将自己全部 加载到一个 buffer 缓冲器，并且通过简单的调用能在任何时间被播放。较长的声音不得不作为流。 通过轮流检测播放指针的位置，或在播放游标到达确定点后请求通告的方式，一个应用程序可以探知有更多的数据流向 buffer 缓冲器 的时候。托管版的 DirectX 提供了一个方法，仅仅来加载非流的静态的 buffer 缓冲器从一个 WAV 文件直接到一个次要 buffer 缓冲器；但是对于流 buffer 缓冲器，你有责任解析文件和拷贝数据到 buffer 缓冲器。

第2节　设备

在应用程序中第 1 步执行DirectSound，将建立一个描绘声音设备的 Device 对象。
本节描绘了你的程序能怎样列举可获得的声音设备，建立 Device对象，使用对象的方法们来设置cooperative合作级别，重新得到设备的功能，建立声音buffer缓冲器，设置系统的扬声器配置，精简硬件内存。

Enumeration of Sound Devices
Creating the Device Object
Cooperative Levels
Device Capabilities
Speaker Configuration
Compacting Hardware Memory

2.1 枚举声音设备

作为一个简单的应用程序，它将要播放声音贯穿于用户首选指定的回放设备，你不需要枚举可获得设备。要知道更多信息，看 Creating the Device Object 。
如果你在寻找一个特殊种类的设备，想提供用户一个设备选择，或者需要 2 个以上设备的协作，你必须枚举这些系统可获得的设备们。
枚举服务的 3 个目的：

汇报有什么硬件被获得。
为每个设备提供 GUID 。
能让你为每个被枚举的设备建立一个临时 Device 对象，以便于你能查看设备的能力。

要枚举设备，例如一个 DevicesCollection对象,使用它的方法和属性来查询获得的设备们。得到设备描述：设备的GUID，能被从结构体的属性来获得的每个设备的模块名。

下面的 C# 代码找回可获得的设备。同样的程序过程通过 CaptureDevicesCollection 取代 DevicesCollection 能被用来鉴别可获得的捕获的设备们。

```
[C#]
using Microsoft.DirectX.DirectSound;

public class wfEnum : System.Windows.Forms.Form

private DevicesCollection myDevices = null;

private struct myDeviceDescription

{

    public DeviceInformation info;

    public override string ToString()

    {

        return info.Description;

    }

        public myDeviceDescription(DeviceInformation di)

        {

            info = di;

        }

}

public wfEnum()

{

    // Retrieve the available DirectSound devices

    myDevices = new DevicesCollection();


    foreach (DeviceInformation dev in myDevices)

    {

        myDeviceDescription dd = new myDeviceDescription(dev);


        // Use DevicesCollection and DeviceInformation to query for devices.

    }
```

第一个被枚举的设备总是被称为主声音设备。这个设备被用户在控制面板中设置为首选的重

放设备。它被独立地枚举，使应用程序很容易的加上"主声音设备"到一个列表，在当前用户有选择设备机会的时候。主设备也用它的名字和 GUID 来被枚举。

2.2 建立设备对象

下面的 C# 范例展示了怎么为默认设备创建一个 Device设备对象

```
[C#]
private Device dsDevice = null;

dsDevice = new Device();
```

下面的 C# 代码建立了一个来自 GUID 的设备，它是被用 deviceGuid 变量描述的设备。这个 GUID 能被设备们的枚举获得。

```
[C#]
dsDevice = new Device(deviceGuid);
```

你也可以用下面的值之一，来指定一个默认设备。

值	描述
DSoundHelper.DefaultPlaybackDevice	这个默认的系统音频设备。这是和通过无参数的 Device构造器同样建立的设备。
DSoundHelper.DefaultVoicePlaybackDevice	这个默认的语音通信设备。比如 USB 接口的扩音器，典型的是次要设备。

如果没有声音设备，或者VxD虚拟设备驱动器，对象不能被创建。如果声音设备是在使用标准 Microsoft Win32? waveform-audio 函数的程序下控制。参看 Capturing Waveforms。

2.3 合作级别

因为 windows 是多任务环境，在任何可能时刻，超过 1 个的程序能和驱动设备一起工作。通过使用 cooperative 合作级，DirectX 可以确定每个程序不用错误的方法、不在错误的时间的访问设备。每个 DirectSound 程序有合作级别决定许可访问的设备的范围。
在建立一个 Device 对象之后，在你播放声音之前，你必须使用 Device.SetCooperativeLevel方法为要使用的设备设置合作级。
DirectSound为声音设备定义 3 个合作级别，用 CooperativeLevel枚举值来描述。

2.3.1 标准合作级别

在标准合作级别，程序不能设置主 buffer 缓冲器的格式，写到主 buffer 缓冲器，或者压缩该设备的板载内存。所有的程序在这个 cooperative 合作级别使用的主 buffer 缓冲器格式为 22 kHz，立体声，和 8 bit 采样，以便于设备能在应用程序间尽可能平滑的转换。

2.3.2 优先合作级别

当使用一个 DirectSound 设备的优先合作级别时，程序有优先权来用硬件资源，例如硬件混和器，并且能设置主 buffer 缓冲器的格式和压缩该设备的板载内存。
游戏程序能在几乎所有的环境使用优先合作级别。当允许程序控制频率采样和深度位数时，这个级别给了很充沛的行为。优先合作级别也允许来自其它应用程序的音频，例如 IP 电话，来和游戏音频一同被收听。

2.3.3 写优先合作级别

最高合作级别是写优先合作级别。当使用一个有合作级别的 DirectSound 设备，你的程序已经直接访问到主声音 buffer 缓冲器。在这种模式，程序必须直接写操作到主 buffer。当发生这种状况，次要 buffer 不能被播放。
为了获得直接的写操作访问主buffer缓冲器的音频采样，一个程序必须被设置成写优先合作级别。如果程序不被设置到这个级别，接下来的所有关于 Buffer.Write或者 Buffer.Read 方法的调用将会失败。
当你的程序将被设置成写优先合作级别，并且得到前台，所有的为其它应用程序工作的次要 buffer 缓冲器们都被被停止和标记为丢失。当你的程序转移到后台，又再次转移到前台，它的主 buffer 被标记成丢失并且必须被保存。
你不能设置写优先合作级别，如果一个DirectSound设备不在当前的用户系统。要确定是否是这种场合，检查 Caps.EmulateDriver 属性，它在 Device.Caps 中。如果 EmulateDriver设为true，则这个DirectSound设备正在被设备仿效，并且写优先合作级别不能被设置。

2.4 设备功能

DirectSound 能让你的程序重新获得声音设备的硬件功能。很多程序不需要这么做，因为 DirectSound 自动得益于任何可获得的硬件加速器。可是，高执行效率程序能使用信息来伸缩它们的声响需求到可获得的硬件上。举例：如果硬件混合器是可获得的，那么比起它不可获得的情况，一个程序可能选择播放更多个声音。
在建立 Device对象以后，你的程序能重新得到一些声音设备能力，它们来自 Device.Caps 属性。这个 Caps结构体包含关于执行效率和声音设备资源的信息，包括每个类型的最大资源和当前可以获得的资源们。
如果你的程序伸缩于硬件能力，在每个buffer缓冲器安置之后，你应该检查这个 Caps属性，来决定是否有足够的资源来建立下一个buffer缓冲器。

2.5 扬声器配置

DirectSound 使用扬声器配置，那就是说，相对于听众的扬声器位置，使用户的声音系统 3-D 特效最优化。
在Win9x，2k以及后续的操作系统，扬声器配置能被通过控制面板的使用来设置。一个程序能重新获得来自于 Device.SpeakerConfig属性的值。虽然这个属性能被设置，但是应用程序也不该做这些，因为它作为全局设置它会影响到其它的用户和应用程序。

2.6 压缩硬件内存

只要你的应用程序有最少优先的cooperative合作级别，就能使用 Device.Compact方法来移动任何数据，从板载的系统内存中到邻近的阻塞中，来安排出最多部分的空闲内存供获得。这个方法不被用在多数的现代声卡，这些声卡使用系统内存。

第3节　缓存

DirectSound 的 buffer 缓冲器对象控制着发送波形数据，从源头到目的地。对于大多数 buffer 缓冲器，目的地是个被称为"主 buffer 缓冲器"的混合引擎。从"主 buffer 缓冲器"，数据流到转换数字采样为声音波形的硬件。
关于使用DirectSound 的buffer缓冲器的信息包括以下标题。关于捕获buffer缓冲器的信息，参看 Capturing Waveforms 。

3.1 缓冲器初步

当 DirectSound 被初始化时，为了混合声音和发送它们到输出设备，会自动建立和管理一个主 buffer 缓冲器。
你的程序必须建立至少一个次要声音buffer缓冲器来贮存和播放个别的声音。更多的怎么去做这点，请参看 Creating Secondary Buffers 。
一个次要buffer缓冲器有一个波形格式，这个格式必须和数据格式匹配。这个格式在WaveFormat结构体中被描绘。一个次要buffer缓冲器的格式不能被改变。声音的不同格式能在不同的次要buffer缓冲器中被播放，并且自动混合到主buffer缓冲器中的共有格式。
一个次要 buffer 缓冲器存在于应用程序的整个生命，或者当它不再被需要时，它能被销毁。它能包含一个单独的声音，这个声音将能够被重复播放，比如游戏的音效，或者它能时而被新数据填满。程序能播放存贮在次要 buffer 缓冲器里的一个声音，这被视为一个事件，或作为连续演奏的循环音响。次要 buffer 缓冲器也能使用流数据，这些情况中声音文件包含更多的数据，相比方便的存贮在内存中的那些。
buffer缓冲器既能被硬件也能被软件定位。硬件buffer缓冲器被混合是通过声卡的处理器，软件buffer缓冲器被混合是通过CPU。软件buffer缓冲器总是在系统内存中的；软件buffer数据能在系统内存中，也能不在，如果程序需要这样，并且资源们是可以在板载内存利用的。更多信息参看 Dynamic Voice Management 和 Hardware Acceleration on ISA and PCI Cards
你从不同的次要 buffer 缓冲器混合声音，是通过简单的同时播放它们。任何数目的 buffer 缓冲器能在某一个时刻被播放，这决定于可利用的处理器能力的限制。
通常的，在主 buffer 缓冲器，你根本不是必须关注你自己；DirectSound 在场景后台管理这些。如果你的程序将要执行它特有的混合，DirectSound 将让你直接写到主 buffer 缓冲器。如果你做了这些，你也不能使用次要 buffer 缓冲器。

3.2 建立次要缓冲器

要建立次要buffer缓冲器，要使用 SecondaryBuffer的 5 个重载的构造器之一，通过构造

器这将实例化一个 Device对象。适当场合，也通过构造一个 Stream源数据流对象，或者一个 String文件命名对象。
buffer缓冲器对象通过建立它们的 Device对象所拥有。当 Device对象被发布，所有的buffer缓冲器被这个对象建立，也将被发布，并且不能被引用。
DirectSound分配硬件资源给第一个能利用这些硬件的buffer缓冲器。因为硬件buffer缓冲器是被声卡的处理器混合的，它们在程序执行效率上有非常少的冲突。要指定buffer缓冲器位置，是通过 BufferDescription对象来做下面的构造器。

3.3 缓冲器描绘选项

为了描绘buffer缓冲器的特性和位置，一个实例化的 BufferDescription对象可以通过SecondaryBuffer构造器完成。
如果你想指定buffer缓冲器的位置，以取代由DirectSound决定它属于哪里，既可以设置BufferDescription.LocateInHardware 属性，也可以设置BufferDescription.LocateInSoftware属性。如果 LocateInHardware被设置，并且有不够的硬件资源，这个buffer缓冲器的创建请求则失败。
为了利用DirectSound的语音管理特色，当建立buffer缓冲器的时候，要设置BufferDescription.DeferLocation属性。这个属性推迟给buffer缓冲器的资源分配，直到它被播放。更多信息，参看 Dynamic Voice Management
你能确定现有的buffer缓冲器的位置，通过获得 Buffer.Caps属性（它继承于SecondaryBuffer属性）。既检查 LocateInHardware属性，也检查 LocateInSoftware属性里的 BufferCaps结构体。两者之一总是被指定的。
设置 BufferDescription.StaticBuffer让DirectSound知道buffer缓冲器应该将被建立在板载的硬件内存，如果可能。buffer缓冲器的建立不会失败，如果硬件buffer缓冲器是不可用的。硬件静态buffer缓冲器将只被短促的声音使用，这些声音将被重复的回放。对于大多数现代声卡，这个属性没有影响，因为它们的buffer缓冲器使用系统内存。
参看 Hardware Acceleration on ISA and PCI Cards.

3.4 缓冲器控制选项

当建立一个声音buffer缓冲器，你的程序必须需要为buffer缓冲器指定一些控制选项。这是用来处理 BufferDescription类，它能包含一个或更多的后继控制属性，这些控制属性来自于 BufferCaps结构体。

属性	描述
Control3D	Buffer supports 3-D control. Cannot be combined with ControlPan. buffer缓冲器支持 3D控制。不能被 ControlPan组合。
ControlFrequency	Buffer supports frequency control. buffer 缓冲器支持频率控制。
ControlEffects	Buffer supports effects processing.

	buffer 缓冲器支持效果处理。
ControlPan	Buffer supports pan control. Cannot be combined with Control3D. buffer缓冲器支持面板控制。不能被 Control3D组合。
ControlPositionNotify	Buffer supports notifications of cursor position. buffer 缓冲器支持公告游标位置。
ControlVolume	Buffer supports volume control. buffer 缓冲器支持音量控制。

要在所有声卡上获得最佳性能，你的程序应该指定，它将要使用的唯一一种控制选项。DirectSound使用控制选项来决定是否硬件资源能够被分配到声音buffer缓冲器。例如：一个设备可能支持硬件buffer缓冲器，但是对于那些buffer缓冲器，不提供面板控制这种控制选项。在这种例子，DirectSound将只使用硬件加速，如果 ControlPan属性不被指定的话。

如果你的程序尝试使用一种控制选项，这种选项是buffer缓冲器缺乏支持的，那么这个方法会调用失败。例如：如果你尝试设置 Buffer.Volume属性，除非在buffer缓冲器被建立时，这个 ControlVolume属性被指定，一个异常会发生。

参看 Playback Controls

3.5 缓冲器的 3D 运算法则

当你建立一个使用了 BufferCaps.Control3D属性的次要buffer缓冲器，并且它是软件实现的，你能指定一个运算法则来模拟声音的 3D空间位置。默认，没有首要关系转换函数（HRTF）处理过程被执行，并且声音相对于听众的位置的确定，只是通过面板和音量。对于buffer缓冲器，你能要求 2 级HRTF。

参看 BufferDescription.Guid3DAlgorithm.

3.6 填充和播放静态缓冲器

自身完全包含声音的次要 buffer 缓冲器，被称作为静态 buffer 缓冲器。虽然对于不同的声音，可能会重新使用同一个 buffer 缓冲器；不过数据写到静态 buffer 缓冲器只有一次，这算是个特色。

静态 buffer 缓冲器被建立和管理，就像流 buffer 缓冲器那样。唯一的不同就在于它们被使用的方法：静态 buffer 缓冲器被一次填充，然后播放；但是流 buffer 缓冲器是经常的刷新播放时的数据。

通过在buffer描绘中，设置 BufferCaps.StaticBuffer属性来创建一个静态buffer缓冲器不是必要的。这个属性请求声卡上的内存分配，这在大多数现代声卡上是不可利用的。静态buffer缓冲器能存在于系统内存，既能被 LocateInHardware属性，也能被 LocateInHardware属性建立。更多信息参看 Hardware Acceleration on ISA and PCI Cards

当你使用 SecondaryBuffer(String,BufferDescription,Device) 或

SecondaryBuffer(Stream,BufferDescription,Device)构造器建立次要buffer缓冲器的时候，这个buffer缓冲器被自动给予正确的格式和尺寸，并且被数据填充。如果你使用SecondaryBuffer(String,BufferDescription,Device)或SecondaryBuffer(Stream,BufferDescription,Device)构造器，你就有责任设置buffer缓冲器的格式和尺寸，并且你必须通过 Buffer.Write方法（此方法被 SecondaryBuffer继承）来写数据到buffer缓冲器。
要播放buffer缓冲器，调用 Buffer.Play 方法。
接下来的 C# 范例建立了一个来自于文件的 buffer 缓冲器，并且播放之。这个 buffer 缓冲器支持音量控制 ，面板和频率控制。假定文件名是个 WAV 文件的有效路径，并且设备是个 Device 对象。

```
[C#]
SecondaryBuffer secBuff = null;

BufferDescription desc = new BufferDescription();

desc.Flags = BufferCaps.ControlPan | BufferCaps.ControlVolume |
BufferCaps.ControlFrequency;

secBuff = new SecondaryBuffer(FileName, desc, device);

secBuff.Play(0, 0);
```

在这个范例，因为 BufferPlayFlags标识没有被设置，当到达结尾，这个buffer缓冲器自动停止。通过使用 Buffer.Stop方法，你也可以提前停止它，或者终止一个正在循环的buffer缓冲器。

3.7 使用流缓冲器

流 buffer 缓冲器播放一个不适合立刻整个装在 buffer 缓冲器里的长声音。当 buffer 缓冲器播放时，旧数据被周期性的替代为新数据。播放一个声音流有下面的过程。
1. 调用 SecondaryBuffer(Stream,BufferDescription,Device)来建立一个有着正确的波形格式和便利的尺寸的buffer缓冲器。buffer缓冲器的大小，以容纳 1 到 2 秒的数据为宜；更小尺寸的buffer缓冲器能被使用，但是它们不得不被更频繁的刷新，这样会效率低下。
2. 设置公告位置，以便于你的程序知道刷新一部分buffer缓冲器的时刻。例如，你能中途设置公告，对于buffer缓冲器。当播放游标到达中点，第一部分buffer缓冲器被刷新；当游标到达结尾，第二部分buffer缓冲器被刷新。还有种解决方案可备选，在buffer缓冲器播放时，你能票选播放游标，但是这么做比起公告的解决方案降低了效率。更多信息，参看 Play and Write Cursors
3. 通过使用 Buffer.Write 方法，加载含有数据的整个 buffer 缓冲器。
4. 调用 Buffer.Play方法（被 SecondaryBuffer继承），指定 BufferStatus.Looping属性。
5. 当播放游标到达你想刷新数据的第一点，再次调用 Write方法，写新数据从buffer缓冲器的起始直到播放游标。保存最后一个字节被写入的位置。
6. 每次调用 Write方法，播放游标会到达一个刷新点。 把数据写到buffer缓冲器介于上个保存位置和播放游标之间的那一部分。注意，这个循环是自动处理的：如果buffer缓冲器

长度是 10,000 bytes，并且你写 2000 bytes 到偏移量 9000 的位置，那么最后的 1000 bytes 被写到buffer缓冲器的前面。
7．当所有数据已经被写到buffer缓冲器，在最后有效的字节设置一个公告位置，或者票选这个位置。当游标到达数据的末尾，调用 Buffer.Stop 。
绝不要写到整个 buffer 缓冲器，当它在播放时。每次只刷新 buffer 缓冲器的一个部分。不要写到 buffer 缓冲器中介于播放游标和写入游标之间的那一个部分。

3.8 回放控制

要重新找回并且设置buffer缓冲器被播放的音量，你的程序能使用 Buffer.Volume属性（所有的属性都由 SecondaryBuffer所继承）。在主buffer缓冲器设置音量，会改变声卡的波形音频的音量。音量是以百分贝来度量的，并且是一个负数。因为分贝的衡量不是线形的，有效的静音也许发生在-10,000 的最小音量被到达的很久以前。
这个 Buffer.Frequency属性控制着音量每秒的采样播放频率。设置这个属性从 0 到重置频率回原始值，是在 WaveFormat 结构体被指定时，这个结构体描述着buffer缓冲器的格式。你不能改变主buffer缓冲器的频率。
这个 Buffer.Pan属性控制着声音的左右位置。这个面板效果是通过消弱一个声道来达到的。当声音在中心时（一个 Pan 值为 0），两个声道是全部音量。
为了使用这些控制中任何的一个，当创建buffer缓冲器时，你必须设置适当的标识。参看 Buffer Control Options 。

3.9 播放游标和写入游标

在 buffer 缓冲器中，DirectSound 维持着两种指示器：播放游标和写入游标。这些位置是 buffer 缓冲器中的字节偏移量。
这个 Buffer.Play 方法总是于buffer缓冲器的播放游标起始播放。当buffer缓冲器被建立，这个游标位置被设置为 0。当一个声音被播放，这个游标发生移动，并且总是指向将被输出数据的下一个字节。当buffer缓冲器被中止，游标保持在数据的下一个字节。
写入游标是一个点，这点之后对于写入数据到 buffer 缓冲器是安全的。在当前播放位置和当前写入位置之间的区块已经被用于播放，改变它是不安全的。
你可以可视化这个 buffer 缓冲器为时钟外形，其数据写入按顺时针方向。播放位置和写入位置就像两根针，它们以相同的速度在表面圆周运动，写入位置总是在播放位置前面保留一点超前量。如果播放位置指向到 1 刻度，并且写入位置指向到 2 刻度，只有写数据到 2 刻度之后才是安全的。在 1 刻度和 2 刻度之间的数据，可能已经通过 DirectSound 处于回放队列，并且不应该被接触。
写入位置和播放位置一起移动，不是和数据写到 buffer 缓冲器一起。如果你是流数据，你有责任维持你自己的指示器在里，来指出在哪里下一数据块应该被写入。
一个程序能重新得到播放游标和写入游标，从 PlayPosition属性和 WritePosition属性。你也能设置 PlayPosition属性，它会间接改变 WritePosition属性。
要确保播放游标尽可能正确地被报告，当建立次要buffer缓冲器时，总是指定 BufferCaps.CanGetCurrentPosition 属性。

3.10 播放缓冲器公告

音频流的情况，你也许想要你的程序将被公告，在播放游标到达buffer缓冲器里的某一点的时候，或者回放被停止的时候。通过使用 Notify.SetNotificationPositions方法，你能设置任何数目的点，在buffer缓冲器里，在那里事件们将被触发。当buffer缓冲器被播放时，你不能做这些。
要建立公告，做以下步骤：
1. 为每个公告位置建立一个 AutoResetEvent。
2. 通过 SecondaryBuffer对象到 Notify构造器，来获取这个 Notify对象。
3. 建立一个 BufferPositionNotify结构体的数组，每一个元素对应每个公告位置。设置 Offset属性到你想要被公告的位置的字节偏移量。设置 EventNotifyHandle属性到你在第一步建立的事件之一的 AutoResetEvent.Handle 。
4. 调用 Notify.SetNotificationPositions方法，经过 BufferPositionNotify结构体们的数组。
你现在能够播放buffer缓冲器在单独的线程，并且使用 WaitHandle.WaitAny来等待公告。

3.11 混频音响

混频使自动被做的，当你同时播放多个次要 buffer 缓冲器时。
在早期的驱动模式下，DirectSound 混频器产生最好的声音质量，如果你程序的所有声音使用同一种 WAV 格式，并且硬件输出格式是匹配于声音格式的情况下。如果是这种情况，混频器不需要执行任何格式转换。
你的应用程序能改变硬件输出格式，通过建立主buffer缓冲器对象和设置 Buffer.Format属性。这个主buffer缓冲器对象只是为了控制目的；建立它和获得主buffer缓冲器的写入访问权限不一样，并且你不需要 CooperativeLevel.WritePrimary 合作级别。可是，你需要 CooperativeLevel.Priority 合作级别。DirectSound将会恢复硬件格式到指定格式，这个格式由每次得到输入焦点的应用程序的最后调用所决定。
你必须设置主 buffer 缓冲器的格式，在建立任何次要 buffer 缓冲器之前。
在微软视窗驱动模式（WDM）驱动下，设置主buffer缓冲器格式是无效的。这个格式是由内核混频器决定的。更多信息参看 DirectSound Driver Models。

3.12 丢弃和恢复缓冲器

对于一个声音 buffer 缓冲器的内存，在特定情形能被丢弃：例如，当 buffer 缓冲器们在声卡内存中被定位，并且另一个应用程序得到了硬件资源的控制权。当一个有 write-primary 的 cooperative 合作级别的应用程序移到前台，丢弃也能发生在这种情况；在这种情况，DirectSound 令所有其他的声音 buffer 缓冲器们丢弃，以便于前台应用程序能直接写入到主 buffer 缓冲器。
当一个应用程序尝试播放或者写入数据到一个丢弃的buffer缓冲器时，一个异常被凸现。当那个导致丢弃的应用程序从它原来的write-primary cooperative合作级别降低，或者这个程序移动到后台的时候；其它的应用程序能通过调用 Buffer.Restore方法，尝试再分配buffer缓冲器内存。如果成功，这个方法恢复buffer缓冲器记忆，和buffer缓冲器的所有其他设置，比如音量和面板设置。可是，一个被恢复的buffer缓冲器也许不能包含正确有效的

声音数据，所以自己的应用程序应该重写到buffer缓冲器的数据。

第4节　使用 WAV 数据

对于一个声音 buffer 缓冲器的内存，在特定情形能被丢弃：例如，当 buffer 缓冲器们在声卡内存中被定位，并且另一个应用程序得到了硬件资源的控制权。当一个有 write-primary 的 cooperative 合作级别的应用程序移到前台，丢弃也能发生在这种情况；在这种情况，DirectSound 令所有其他的声音 buffer 缓冲器们丢弃，以便于前台应用程序能直接写入到主 buffer 缓冲器。
当一个应用程序尝试播放或者写入数据到一个丢弃的buffer缓冲器时，一个异常被凸现。当那个导致丢弃的应用程序从它原来的write-primary cooperative合作级别降低，或者这个程序移动到后台的时候；其它的应用程序能通过调用 Buffer.Restore方法，尝试再分配buffer缓冲器内存。如果成功，这个方法恢复buffer缓冲器记忆，和buffer缓冲器的所有其他设置，比如音量和面板设置。可是，一个被恢复的buffer缓冲器也许不能包含正确有效的声音数据，所以自己的应用程序应该重写到buffer缓冲器的数据。

第2章　3-D 声音

将被应用到单独的 DirectSound 缓存中。全局参数被设置在一个叫做收听者的对象上。下面主题覆盖了某些 3-D 声音常见应用。

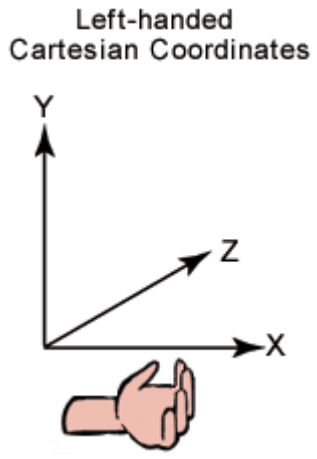
3-D空间坐标系
声音位置的感知

关于如何在程序中使用 3-D 声音的信息可以在下面部分找到。

3-D缓存
3-D收听者

第1节　3-D 空间坐标系

声音源的位置、速度和方向和听众在 3-D 空间中通过笛卡儿坐标系（具有 3 个轴上的值：x-轴，y-轴，z-轴）来表达。这些轴与和程序建立的视点相关。x-轴上的值从左到右增长，y-轴从下往上，z-轴从近到远。



DirectSound 使用如上所示的左手坐标系。如果你左手的手指指向 x-轴正方向并弯曲它们指向 y 轴正方向，你的拇指将指向 z-轴正反向。
按照惯例，向量通过包含在括号中并由逗号分隔的三个值所表达，顺序依次为（x，y，z）。在DirectSound中，三个轴向上的位置、速度或方向坐标用一个 Vector3 结构体来表达。位置上的值默认用米制表达。如果你的程序不使用米制作为衡量 3-D图形的测量单位，你可以设置一个距离因子。它是一个浮点型值，代表了程序指定的每距离单位折算的米制。比如你的程序使用英尺作为距离单位，程序可以指定一个 0.3048 的距离因子，这是每英尺换算成的米数。更多信息参看 Distance Factor。
速度上，向量描述了每秒沿每个轴向移动单位刻度的速度。默认单位还是米制，但这可以通过程序改变。
方向上的值单位是任意的并彼此相关。如果 3-D 世界的基础视点是面向北方，与地面水平，和收听者的方向是（1，0，0）。这就是说 x-轴方向是正值，另 2 个轴向保持中值。然后收听者面向正东方。如果向量是（1，0，1），收听者方向是朝向东北方。如果是（-1，0，1），收听者朝向西北方。因为在一个向量中的值不是绝对值，最后的范例可以同样的用许多其他方式来表达，诸如（-5，0，5）或（-0.25，0，0.25）。
你可以看见 2-D 空间中的向量是如何通过在一页图形页面上绘制来工作的。让值从页面的底部往顶部，从左到右增长。一条从（0，0）到（1，1）的直线具有与从（0，0）到（5，5）绘制的直线相同的方向或指向。然而第二条线指明了 1 个更大的距离或速度。3-D 向量就是以相同方式工作的，并具有 1 个额外的轴。

第2节　声音位置的感知

在真实世界中，对空间中声音位置的理解是受许多因素影响的。这些因素不全都是声学方面的；最重要的之一是视线。声音本身的因素包含以下内容：

Overall loudness。总响度。当声源远离收听者时，声源的被感知音量将以一个固定速率衰减。这种现象被称为高低频规律性衰减。

Interaural intensity difference。耳间声强差。比如，来自于收听者右边的 1 个声音在右边听上去要比在左边听上去更响。

Interaural time difference。耳间时差。比如，来自于收听者右边的 1 个声音到达右耳的时间要比到达左耳的时间稍微提前一点。这种差异大约在 1 毫秒。

Muffling。声音抑制。耳朵的轮廓和方向确保了来自于收听者后方的声音要比来自于前方的声音稍微压抑一点。另外，如果 1 个声音来自于右边，到达左耳的声音将会穿过收听者的头部块并因为左耳的方向而受到抑制。

Effect of the pinnae。背脊特效。耳垂的背脊改变了声音从不同方向到达的程度和时序，从而将声源位置的精细线索提示给我们的大脑。这种效果后面的数学原理被称为头部相关转移函数（HRTF）。

如果用户在控制面板中已经选择了 1 个环绕声扬声器配置的话，Microsoft DirectSound 将在 2 只扬声器、耳机或者通过Microsoft Windows Driver Model（WDM）驱动程序在多声道系统上建立虚拟 3-D特效。更多信息参看 BufferDescription。

第3节　3-D 缓存

3-D环境中的每个声音源都是通过一个 Buffer3D类的对象来表达的。该类仅被

BufferCaps.Control3D属性建立的声音缓存所支持。
当使用DirectSound的 3-D功能时，程序必须提供单声道音源。如果你试着用 Control3D 属性集建立 1 个缓存并用 1 个以上的声道来建立 1 个WAV格式的话就会产生 1 个异常。

3.1 获取 Buffer3D 对象

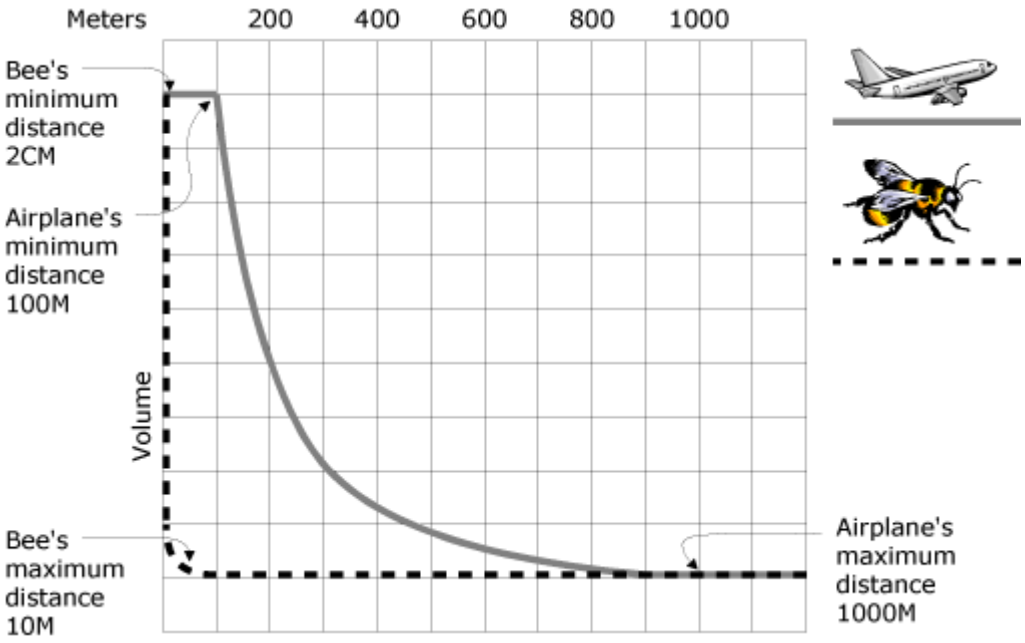
通过传递 1 个 SecondaryBuffer对象到 Buffer3D构造函数来建立 1 个 Buffer3D对象。BufferDescription.Control3D属性必须被设置为true。
参看 Buffer3D(SecondaryBuffer)

3.2 3-D 缓存的批量参数

程序可以单独或成批的找回或设置 1 个 3-D 声音缓存的参数。要设置单独的值，你的程序可以使用可用 Buffer3D 属性。然而程序经常在声音源移动时，必须立即设置或找回某些值，改变速度等等。你可以通过使用 Buffer3D 完成这些所有参数属性。
参数改变也可以通过延缓排列然后依次运行全部参数的方式来变得更有效率。更多信息参看 Deferred Settings。

3.3 最小和最大距离

当 1 个收听者接近 1 个声音源时，声音变大；音量在距离减半的时候加倍。然而超过了 1 个确定点后，音量继续加大就不是很有实际作用了。这就是声音源的最小距离。
在程序必须为不同声音的绝对音量级别之间差异作出补偿时，最小距离尤其有用。虽然 1 个喷气飞机发动机比 1 只蜜蜂吵的多，因为特殊理由这些声音必须被记录为相似大小的绝对音量。程序也许为喷气飞机发动机使用了 1 个 100 米的最小距离，为 1 只蜜蜂使用了 2 厘米的最小距离。通过这些设置，喷气发动机必须在收听者 200 米外减半，但蜜蜂在收听者 4 厘米外必须减半。
声音缓存的默认最小距离定义为 1 个单位或在默认距离因子上的 1 米。除非你改变该值，声音在距离收听者 1 米外为全部音量，在 2 米外减半，在 4 米外减为四分之一等等。对大多数声音来说你将可能想要设置一个更大的最小距离，使得声音不会随它的离开而快速衰减。
声音源的最大距离是 1 个超过该值以后音量不再降低的距离。DirectSound 3-D缓存的默认最大距离是 10 亿，这意味着大多数情况下衰减计算将在声音移出听觉范围外继续。为了避免在VxD虚拟设备驱动下的软件缓存上进行不必要处理，程序应该在建立缓存时，设置一个合理的最大距离并包含 Mute3DAtMaximumDistance属性。
最大距离常用于防止声音变得不可听。比如，如果你已经为 1 个声音在 100 米处设置了最小距离，该声音也许在 1000 米或更近处就变得不可听。通过设置在 800 米处的最大距离，你可以确保声音总是具有至少最大音量的八分之一而不管距离远近。既然这样，你就不应该设置 Mute3DAtMaximumDistance属性。
下图描绘了最小和最大距离是如何影响 1 个喷气飞机发动机和 1 只蜜蜂的声音音量随距离增长的。



默认的，距离值是用米制表达的，参看 Distance Factor。
要为所有声音缓存中的音量调节距离效果，你可以改变 Rolloff Factor。

参看

Buffer3D.AllParameters属性
Buffer3D.MaxDistance 属性
Buffer3D.MinDistance 属性

3.4 处理模式

声音缓存具有 3 种处理模式：正常，头部相关，关闭。
正常模式下，声音源是在世界空间中被安排绝对位置和朝向的。这是默认的模式，被用于不移动的、通过收听者转动的声音源。
头部相关模式下，声音源的 3-D 属性是和当前位置、速度和收听者的方向全部相关的。当收听者移动和转动时，缓存自动在世界空间中重新定位。头部相关模式可以用于收听者自己脚步等等之类的声音。然而，大多数头部相关声音并不一定必须全部是 3-D 声音。
关闭模式下，3-D 声音处理被禁用并且声音似乎源于收听者头部中央。

参看

Buffer3D.AllParameters 属性
Buffer3D.Mode 属性

3.5 缓存位置和速度

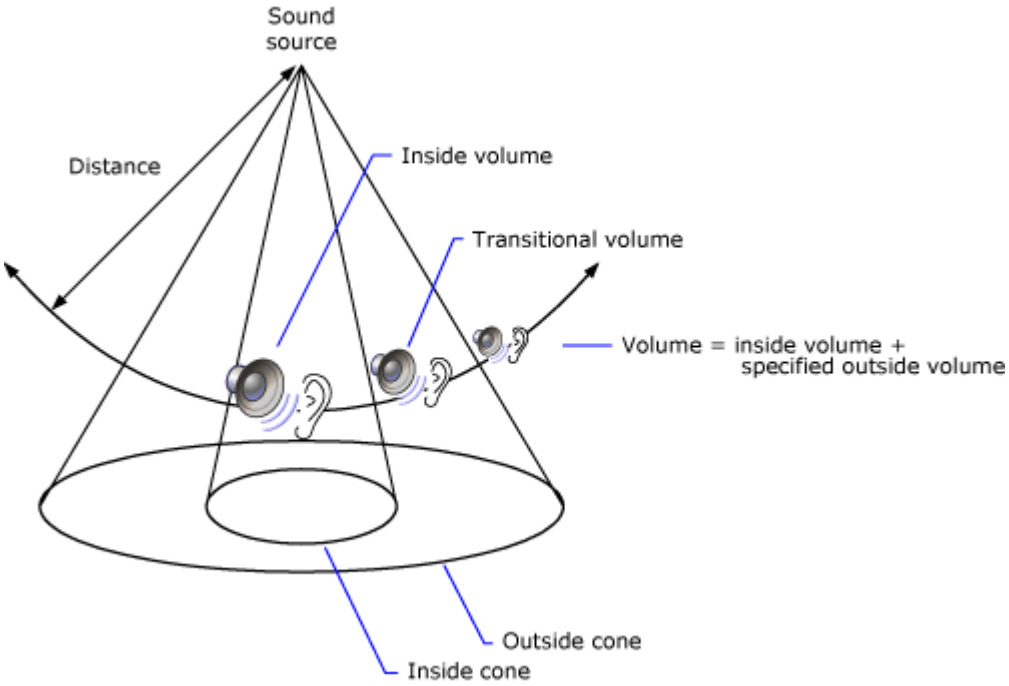
当声音源移动时，它以程序指定的值作为位置和速度。
位置以沿 3 个轴的每个轴方向上距离单位来衡量，相对于世界空间或收听者，这取决于处理模式。
速度以沿 1 个向量的 3 个轴的每个轴方向上距离单位/秒来衡量。默认距离单位是米制。速度通过 DirectSound 仅用于计算多普勒转换的效果。

参看

Buffer3D.AllParameters 属性
Buffer3D.Position 属性
Buffer3D.Velocity 属性
Distance Factor
Doppler Factor
Processing Mode

3.6 声音锥体

没有方向的声音具有和在所有方向上 1 个给定距离相同的振幅。一个有方向的声音在方向的正面是最高的。描绘有方向的声音高低的模型叫做声音锥体，它在声音的发源点具有定位顶点。声音锥体由 1 个内部圆锥和 1 个外部圆锥构成。外部锥角总是必须和内部锥角相等或更大。
在内部圆锥内的任意角度，在考虑到缓存的基本音量、距离收听者的远近和收听者方向等等之后，声音的音量就和没有圆锥一样。
在外部圆锥的任何角度，正常音量通过程序被 1 个因子削弱。外部圆锥音量是以百分制的分贝来表达的，并且是一个负值。因为它代表了从默认音量 0 的削减。
在内部和外部圆锥之间是从内部音量向外部音量转换的区域。音量随角度增加而减弱。
下图描绘了声音锥体的概念。



每个 3-D 声音缓存具有 1 个声音锥体，但 1 个缓存默认表现为全方向声音源，因为外部音量没有削减，内部和外部锥角都是 360 度。除非程序改变了这些值，声音是不会有明显方向的。
正确的设计声音圆锥可以给你的程序加入动态效果。比如你可以在房间的中央定位 1 个声音源，设置它的方向朝向走廊中的开放的门。然后设置内部圆锥的角度使得它扩展到门口的宽度，让外部圆锥更宽一点，并设置外部圆锥音量削弱。沿着走廊移动的收听者仅首先在靠近门口时听见声音，然后声音随着收听者移动至开放的门前时变得最高。
参看

Buffer3D.ConeAngles 属性
Buffer3D.ConeOrientation 属性
Buffer3D.ConeOutsideVolume 属性

第4节　3-D 收听者

和真实世界中一样，在虚拟 3-D 环境中，声音以点接收的方式存在。在 Microsoft DirectSound 程序中的 3-D 特效不仅受声音位置、方向和速率值的影响，还受到虚拟收听者位置、方向和速率的影响。
默认情况下，收听者固定在所有向量的零点，方向为鼻子朝向 z-轴正向，头顶朝向 y-轴正向。通过获取 1 个代表收听者的对象，程序可以改变所有这些值在虚拟空间中反映出移动过程和用户的"面向"。收听者对象还控制常用的声学环境参数，诸如多普勒转换的数量和音量随距离衰减的速率。
该部分描述了你的程序如何获取 1 个收听者并管理全局 3-D 声学参数的。

4.1 获取 3-D 收听者

通过使用 Listener3D对象来设置并找回全局声学参数，该对象由主缓存中获得。在 1 个程序中仅有 1 个收听者。
下列Microsoft Visual Basic函数建立了 1 个代表主缓存的 Buffer对象，主缓存由 Control3D属性建立。然后函数建立并返回 1 个 Listener3D对象。

```
[Visual Basic .NET]
Private Function getListener(ByVal dev As Device) As Listener3D

    Dim primaryBuffer As Buffer

    Dim desc As New BufferDescription()

    Dim caps As New BufferCaps()

    desc.PrimaryBuffer = True

    desc.Flags = caps.PrimaryBuffer OR caps.Control3D

    primaryBuffer = New Buffer(desc, dev)

    Dim listener = New Listener3D(primaryBuffer)

    Return listener

End Function
```

4.2 3-D 收听者的批量参数

程序可以单独的或者成批的找回或设置 1 个收听者的参数。要设置单独的值，你的程序可以使用可用的 Listener3D属性。然而当声源移动、改变速率等时候，程序经常必须 1 次设置或找回多个值。你可以使用 Listener3D.AllParameters属性完成这个操作。

参数改变也可以通过延缓排列它们然后 1 次全部执行的方式变得更有效率。更多信息参看 Deferred Settings。

4.3 延缓设置

每个对 3-D声音缓存和收听者设置的改变导致花费CPU周期进行混音。要使改变 3-D设置的性能影响降至最低，可以在 Listener3D对象和所有 Buffer3D对象上设置 Deferred属性。当该属性设置为true时，直到你调用 Listener3D.CommitDeferredSettings都不会有其他属性发生变化。

注意：通过立即设置安放 CommitDeferredSettings设置。比如，如果你将 Listener3D.Deferred设置为**true**并将收听者速率设置为(1.0, 0.0, 0.0)，然后你将 Deferred设置为**false**并将速率设置为(2.0, 0.0, 0.0)，速率立即变为(2.0, 0.0, 0.0)，当 CommitDeferredSettings调用时将不再改变。

4.4 距离因子

距离因子是向量单位中的米数。默认距离因子是 1.0。如果 1 个缓存的速率是(2.0, 0.0, 0.0)，声源就被认为是沿着 x-轴正向移动 2 米/秒。使用不同刻度单位 3-D 图形向量的程序也许想要相应的改变距离因子。

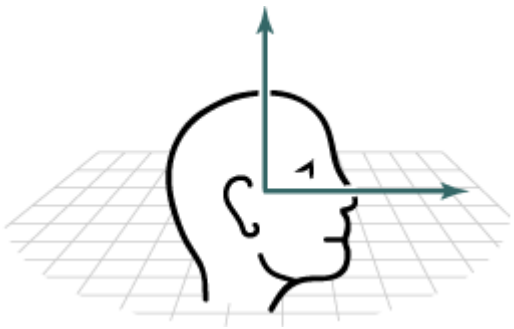
假设比如你程序中的基本刻度单位是英尺或 0.3048 米。你将 Listener3D.DistanceFactor属性设置为 0.3048。从那时起，诸如位置和速率之类的属性就将以英尺来度量。

距离因子主要通过改变实际速率影响多普勒转换，该速率代表 *n* 单位/秒。它不直接影响高低评规律性衰减，因为随距离衰减速率是基于未指明单位的最小距离的。如果你为 1 个给定声音在 2 个单位处设置了最小距离，音量将在 4 个单位距离处减半，无论那些声音是以英尺、米还是以其他任何方法度量的。

参看 Minimum and Maximum Distances

4.5 收听者方向

收听者方向通过 2 个向量（顶部向量和前部向量）之间的关系来定义，这 2 个向量在收听者的头部中央共享 1 个原点。顶部向量穿出头部指向上方，前部向量穿出收听者面部指向前方与顶部向量成直角，如下图所示。



默认前部向量是(0.0, 0.0, 1.0)，顶部向量是(0.0, 1.0, 0.0)。2 个向量总是必须互成直角。如果必须的话，DirectSound 将会调整前部向量使得它与顶部向量成直角。

参看 Listener3D.Orientation 属性

4.6 收听者位置和速率

当收听者移动时，它将变成程序为它的位置和速率所指定的值。
位置沿着 3 个轴的每一个方向通过距离单位来度量，相对于世界空间或收听者，这取决于处理模式。
速率沿着向量的 3 个轴的每一个方向通过距离单位/秒来度量，默认距离单位是米。要计算多普勒转换的话，DirectSound 就将声源的速率和收听者的速率组合起来。
参看
Listener3D.AllParameters 属性
Listener3D.Position 属性
Listener3D.Velocity 属性
Distance Factor
Doppler Factor

4.7 多普勒因子

DirectSound 基于 3-D 声音缓存相对于收听者的速率来为它们建立多普勒转换效果。
了在你的程序中具有真实的多普勒转换效果，你必须计算任何对象移动的速度，无论它是 1 个声源还是 1 个收听者，并设置合适的速率。在 1 个特定情形下为了建立特殊效果，你可以随意扩大或缩小这个值。你还可以通过改变 Listener3D.DopplerFactor属性从整体上增加或减少多普勒转换。
多普勒因子范围从 0.0(DSoundHelper.MinDopplerFactor) 到 10.0(DSoundHelper.MaxDopplerFactor)。值 0 意味着没有多普勒转换应用到声音上。其他每个值代表了 1 个真实世界多普勒转换的倍数。换句话说，值 1(DSoundHelper.DefaultDopplerFactor) 意味着真实世界中体验到的多普勒转换被应用到声音上，值 2 意味着 2 倍真实世界的多普勒转换，以此类推。
参看
Buffer3D.Velocity 属性
Listener3D.Velocity 属性

4.8 高低频规律性衰减因子

高低频规律性衰减是应用到声音上的衰减总计，它基于收听者离开声源的距离。DirectSound可以忽略、扩大高低频规律性衰减或给它以在真实世界中同样的效果，这取决于 Listener3D.RolloffFactor属性的值。
高低频规律性衰减范围从 0.0(DSoundHelper.MinRolloffFactor) 到 10.0(DSoundHelper.MaxRolloffFactor)。值 0 意味着声音没有应用高低频规律性衰减。其他每个值代表了真实世界中高低频规律性衰减的倍数。换句话说，值 1(DSoundHelper.DefaultRolloffFactor) 意味着真实世界中体验到的高低频规律性衰减被应用到声音上，值 2 意味着 2 倍真实世界的高低频规律性衰减，以此类推。
要为 1 个单独声音缓存改变距离效果，你可以为缓存设置最小距离。更多信息参看 Minimum and Maximum Distances。

第3章　使用特效

Microsoft® DirectSound® 通过 Microsoft DirectX® Media Objects (DMOs)提供了声音特效处理的支持。
所有的标准 DMOs 除了波反射，能处理有 1 或 2 个声道、8-bit 或 16-bit 脉冲数码调制(PCM)、任何被 DirectSound 支持的采样频率的数据。

第1节　在缓存上设置特效

一个以 BufferCaps.ControlEffects属性被建立的次要buffer缓冲器能设置任何数目的特效。特效可能不能平稳的工作在恰好小的buffer缓冲器。并且Microsoft® DirectSound®不允许有这种具有"支持少于 150(BufferSize.FxMin)毫秒数据"的特效能力的buffer缓冲器创建。
要执行一个在buffer缓冲器上的特效，使用 SecondaryBuffer.SetEffects方法。这个方法一个 EffectDescription结构体数组，这个结构体是描述特效的。你也可以使用 SetEffects方法来移除特效，经由空引用代替数组。在任一种情况，buffer缓冲器必须不在被播放。特效能用硬件（这就是说，通过声卡驱动执行）或用软件示例。更多信息，参看EffectDescription。
下面的 VB.net 范例函数在一个 buffer 缓冲器上设置标准回声效果。

```
[Visual Basic .NET]
Private Sub SetEcho(ByVal b As SecondaryBuffer)

    Dim fx As EffectDescription()

    ReDim fx(0)

    fx(0).GuidEffectClass = DSoundHelper.StandardEchoGuid

    Try

        b.SetEffects(fx)

    Catch e As ControlUnavailableException

        Debug.WriteLine("BufferDescription does not have ControlEffects set.")

    End Try

End Sub
```

第2节　特效参数

要设置或者重新得到声音特效们的那些参数，首先你必须从那个包含特效的 buffer 缓冲器，获得适当的特效对象。
下面类中的对象们给你有权访问 Microsoft® DirectX® Media Objects (DMOs)的参数，这是由 DirectX 提供的。

ChorusEffect class
CompressorEffect class
DistortionEffect class
EchoEffect class
FlangerEffect class
GargleEffect class
Interactive3DLevel2ReverbEffect class
ParamEqEffect class
WavesReverbEffect class

要重新得到一个特效对象，使用 SecondaryBuffer.GetEffects方法。这个方法能重新得到一个单一的对象或者是一个对象数组。

接下来的VB.net代码重新得到一个 EchoEffect对象，这个对象描绘了第一个设置在buffer缓冲器上的特效，并且在这个特效上设置两个参数。假定fxBuffer是一个SecondaryBuffer 对象。

```
[Visual Basic .NET]
Dim echoFx As EchoEffect

Dim echoParams As EffectsEcho

echoFx = fxBuffer.GetEffects(0)

echoParams = echoFx.AllParameters

echoParams.LeftDelay = 500

echoParams.RightDelay = 500

echoFx.AllParameters = echoParams
```

当 buffer 缓冲器被播放时，特效参数们能被改变。

为了效率，Microsoft DirectSound®预处理在buffer缓冲器里的 100 毫秒的声音数据，开始于播放游标，在 SecondaryBuffer.Play方法被调用前。这能发生在调用以下任何方法之后。

SecondaryBuffer.SetCurrentPosition
SecondaryBuffer.SetEffects
SecondaryBuffer.Stop
SecondaryBuffer.Write

如果你调用这些方法的任何一种，并且之后改变特效参数，新的参数们将不被审理，直到被预处理的数据已经被播放。为了避免这个情形，做以下的一条：

写数据到 buffer 缓冲器在特效参数已被改变之后，而不是之前。

在buffer缓冲器调用 Stop 或者 SetCurrentPosition方法，来强制预处理在你已经设置参数们之后。

第3节　标准特效

这个专题是介绍 Microsoft® DirectSound®提供的声音特效。涵盖了下列效果：

3.1 Chorus 合声

合声是被一种语音倍增特效，它是这样建立的：对原声稍微延迟附和并稍微调整回声的延迟。
这个特效通过 ChorusEffect类来描绘。它的参数被包含在 EffectsChorus结构体里。

3.2 Compression 压缩

压缩是对某一振幅上信号的波动的缩减。
这个特效通过 CompressorEffect类来描绘。它的参数被包含在 EffectsCompressor结构体里。

3.3 Distortion 扭曲失真

扭曲失真是这样完成的：通过将和声加入信号这样一种方式，当水平面增加时，波形的顶部变成方型或锯齿型。
这个特效通过 DistortionEffect类来描绘。它的参数被包含在 EffectsDistortion结构体里。

3.4 Echo 回声

回声效果导致声音在 1 个固定时延之后被重复，通常在音量减小处。当重复音被填入混音器中时，它们就再次被重复。
这个特效通过 EchoEffect类来描述。它的参数包含在 EffectsEcho结构体里。

3.5 Environmental Reverberation 环境回响

DirectX 依照由交互式音频特别兴趣小组出版的交互式 3-D 音频-第 2 级别（I3DL2）规范支持环境回响。
Microsoft DirectX®环境回响效果是对 I3DL2 规范中的收听者属性的 1 种执行。源属性不被支持。
到达收听者的声音具有 3 个临时成分。

direct path 直接路径。它是从声源直接到收听者的声音信号，不会越过任何表面。因此仅有 1 个直接路径信号。

Early reflections 早期反射。它是在 1 个或 2 个反射离开诸如墙体、地板、天花板之后到达收听者的声音信号。如果 1 个信号是声音仅撞击了 1 面墙体后向前到达收听者的结果，它就被称为 1 个首次反射。如果它在到达收听者之前弹离了 2 面墙体，就被称为 1 个二次反射。人类通常可以感知到仅仅单独的首次或二次反射。

Late reverberation 或者简写为 reverb 后期回响。由低次序的反射组成，通常是 1 个强度逐渐减小的一连串密集回声。

早期反射和后期回响的组合有时称为 room effect。
回响属性包含以下内容：

Attenuation 早期反射和后期回响的衰减。

Rolloff factor 高低频规律性衰减因子，或是反射信号随距离开始衰减的比率。直

接路径的高低频规律性衰减因子由 DirectSound 收听者管理。

Reflections delay 反射延时，是直接路径信号到达和首次早期反射到达之间的时间间隔。

Reverb delay 回响延时，是首次早期反射和后期回响开始时刻之间的时间间隔。

Decay time 衰减时间，是后期回响开始时刻和其强度降低了 60 分贝时刻之间的时间间隔。

Diffusion 扩散，与后期回响中每秒回声数量成比例。取决于执行，密度可以随回响衰减而改变。在 DirectX 中，程序可以通过设置执行过程所允许范围的百分比来控制该属性。

Density 密度，与后期回响中每赫兹共振数量成比例。低密度产生空洞的声音，就像那些声音是在小房间内发生的。在 DirectX 中，程序可以通过设置执行过程所允许范围的百分比来控制该属性。

回响属性由 1 个 Interactive3DLevel2ReverbEffect类表示，环境参数包含在 1 个 EffectsInteractive3DLevel2Reverb结构体中。

DirectSound支持许多组默认参数或预设置，它们描述了从山脉到下水道各种范围声音环境的回响属性。大多数程序可以通过使用 Interactive3DLevel2ReverbEffect.Preset属性简单的从这些环境中选择一个。

要设置并获取返回的自定义属性，使用 Interactive3DLevel2ReverbEffect.AllParameters属性。

3.6 Flange 镶边

镶边也叫作镶边器，是在原始信号和它的回声之间的延迟回声效果，它的回声非常短并随时间变化。结果它不时被提及为 1 种扫除声音。术语镶边和通过抓取磁带轮来改变速度的原理下相似。

这个效果由 FlangerEffect类描述。它的参数包含在 EffectsFlanger结构体中。

3.7 Gargle 漱口音

漱口音效果调整信号的振幅。

这个效果由 GargleEffect类描述。它的参数包含在 EffectsGargle结构体中。

3.8 Parametric Equalizer 参量均衡器

1 个参量均衡器放大或缩小某固定频率的信号。

可以通过在相同缓存上设置效果的多个实例来应用不同程度的参量均衡器效果。这样，程序可以具有类似于 1 个硬件均衡器的音调控制。

这个效果由 ParamEqEffect类描述。它的参数包含在 EffectsParamEq结构体中。

3.9 Waves Reverberation 声波回响

声波回响效果打算用于音乐。声波回响 DirectX Media Object（DMO）是基于 Wavees MaxxVerb 技术的，这被授权给 Microsoft 。

这个效果由 WavesReverbEffect类描述。它的参数包含在 EffectsWavesReverb结构体中。

第4章　捕获波形

为了立即回放或者在文件里储存，Microsoft® DirectSound® 允许你从一个麦克风或其它输入到声卡的设备，捕获声音。数据能被捕获为脉冲编码调制(PCM)或者压缩格式。Capture对象被用来查询声音捕捉设备的能力和建立用来捕捉声音样本的buffer缓冲器。CaptureBuffer对象描绘一个用来接收来自输入设备的数据的buffer缓冲器。这个buffer缓冲器概念上理解为圆形的：当输入到达buffer缓冲器末尾，它自动从开头重新开始。CaptureBuffer方法允许你重新得到buffer缓冲器属性，开始和停止音频的捕捉，以及重新得到数据。在那些支持捕捉特效的操作系统上，它也提供重新得到特效对象的方法和确定特效的状态。

第1节　捕获设备的枚举

如果你在寻找一个特殊种类的设备，想要提供用户一个设备的选择机会，或者需要和 2 个以上的设备一同工作，你必须枚举系统中可用的设备。因为一个应用程序将要从用户首选的捕捉设备里捕捉声音，你不需要枚举可利用的设备。更多信息，参看 Creating the Capture Object 。

枚举适应三个目的：

汇报什么硬件是可以利用的。

为每个设备提供一个全局唯一标识(GUID)。

允许你为每个设备在被枚举时，建立一个临时 Capture对象，以便于你能检查设备的功能。

要枚举设备，首先要使用 CaptureDevicesCollection，来创建一个捕获设备。

第2节　建立捕获对象

如果你的应用程序将要捕获和记录声音，你将建立和使用一个 Capture对象。

下面的C#范例展示如何为默认设备，初始化一个 Capture对象。

```
[C#]
private Capture dsCapture = null;

dsCapture = new Capture();
```

下面的 C#范例建立了一个来自 GUID 的设备，被 deviceGuid 变量所描绘。这个 GUID 能被设备枚举所获得。

```
[C#]
dsCapture = new Capture(deviceGuid);
```

你也能使用下面的值之一来指定一个默认设备。

值	描述
DSoundHelper.DefaultCaptureDevice	默认的系统捕获设备。这是和通过 Capture的无参数构造器所创建的设备是一样的。
DSoundHelper.DefaultVoiceCaptureDevice	默认的语音捕获设备。这是典型的次要设备，例如 USB 接头的麦克风。

如果没有捕获设备，或者 VxD 虚拟设备驱动下，如果声音设备在使用标准 Win32® 波形音频函数的应用程序的控制下，这个对象不能被创建。
如果你的应用程序将要捕获声音，又要播放它们，你应该建立 Device对象和 Capture对象两者。
如果你想回放对象和捕获对象共存，你应该建立并且初始化 Device对象，在创建、初始化 Capture对象之前。
一些音频设备不被默认配置为全双工音频。如果你的应用程序在建立并且初始化 Device对象和 Capture对象两者时有问题，你应该忠告用户检查音频设备属性，以确定全双工被激活。

第3节　捕获设备功能

一个捕获设备的功能被用 CaptureCaps结构体来描绘，这个结构体包含设备支持的通道的数目，以及那些被支持的标准音频格式的组合体，它和用在Win32 波形音频函数里的WAVEINCAPS结构体是等价的。
在非WDM(Microsoft Windows® Driver Model)驱动，一个捕获设备在一个时刻只能被一个应用程序使用。通过设置 CaptureCaps属性，每个应用程序能为每个捕获buffer缓冲器设置它自己的格式。

第4节　建立 1 个捕获缓存

通过建立一个 CaptureBuffer对象，建立一个捕捉buffer缓冲器，来储存声音数据。
捕捉buffer缓冲器能被使用 CaptureBuffer(IntPtr,Capture)构造器直接指定，或者一个新的捕捉buffer缓冲器能用 CaptureBuffer(CaptureBufferDescription,Capture)构造器建立，这个构造器传入的 CaptureBufferDescription结构体是详细说明要得到的buffer缓冲器的特性。这个结构体的 Format格式成员是一个 WaveFormat结构体，它必须使用你想要的WAV格式来初始化。
注意如果你的应用程序是使用 Microsoft® DirectSound® 回放又使用 DirectSound 捕捉，当捕捉 buffer 缓冲器的格式和主 buffer 缓冲器不一样时，捕捉 buffer 缓冲器的建立会失败。原因是某些声卡只有一个专一时钟，不能支持捕捉和回放在两个不同的频率。

第5节　捕获缓存信息

捕获buffer缓冲器的属性从 CaptureBuffer.Caps属性获得。

CaptureBufferCaps结构体描述buffer缓冲器的尺寸，它是否支持特效，并且它是否支持当它被创建时被指定或必须使用Microsoft® Win32® wave mapper指定的格式。
当buffer缓冲器被建立时，要得到关于buffer缓冲器里的数据的格式的信息，应使用使用CaptureBuffer.Format属性。
要发现一个捕获buffer缓冲器当前在做什么，检查 CaptureBuffer属性。如果这个buffer缓冲器被捕获，Capturing属性会被设置。如果 Looping在上次调用 CaptureBuffer.Start方法时就被设置，这种情况常常发生，Looping就被设置。
CaptureBuffer.GetCurrentPosition 方法返回在buffer缓冲器中的读取游标和捕获游标的偏移量。读取游标是在那些已经被完全捕获进buffer缓冲器的数据的这一点末尾。捕获游标是在那些当前被拷贝进buffer缓冲器数据块的末尾。你不能安全地得到数据在读取游标和捕获游标之间。

第6节　捕获缓存通知

当捕获音频时，你可能想你的应用程序将被公告何时读去游标到达buffer缓冲器中一个特定的点，或者buffer缓冲器被停止的时刻。通过使用 Notify.SetNotificationPositions方法，你能设置任何在事件将被通告的buffer缓冲器中的点的数目。当buffer缓冲器被捕获，你就不能做这个了。
要建立公告，作以下步骤：

1. 为每个公告位置，建立一个 AutoResetEvent 。
2. 经由 CaptureBuffer对象的 Notify构造器获得 Notify通告对象。
3. 建立一个 BufferPositionNotify结构体数组，每个元素对应一个公告位置。设置 Offset成员，这是到你想被公告的位置的字节偏移量。设置 EventNotifyHandle成员到你在step 1 建立的事件之一的 AutoResetEvent.Handle 。
4. 调用 Notify.SetNotificationPositions，经由 BufferPositionNotify结构体的数组。

你现在能启动buffer缓冲器在一个独立的线程，并且使用 WaitHandle.WaitAny来等待公告。

第7节　捕获缓存特效

有两个捕获 buffer 缓冲器特效作为 kernel 核心模式执行在 Windows® XP 及其的后续版本上，是可用的。

回声消除(AEC)
噪声抑制

这些特效只能在和一个 FullDuplex全双工对象的联合时被使用。
CaptureBuffer 构造器获取一个预定义buffer缓冲器在内存中的定位，或者一个CaptureBufferDescription结构体，这个结构体包含了一个描绘你所需要的特效的CaptureEffectDescription结构体。
对于每个 CaptureEffectDescription结构体，你必须指定属性 LocateInHardware或者指定 LocateInSoftware属性。大多数情况，指定 LocateInSoftware。如果特效在指定的位置不可用，buffer缓冲器创建会失败。
特效在 CaptureEffectDescription.GuidEffectsClass属性被鉴别。要获得Microsoft 软

件执行，就要指定 DSoundHelper.CaptureEffectsMsAcousticEchoCancellation 或 DSoundHelper.CaptureEffectsMsNoiseSuppression 。接下来要使用硬件或者软件执行，获得类和来自制造厂商的GUID实例。
在buffer缓冲器被创建之后，你能通过调用 CaptureBuffer.GetEffectsStatus方法得到关于特效的信息。
AEC 主要影响站点之间使用语音通信的应用程序。没有回声消除，来自站点 A 的麦克风的信号是从站点 B 的扬声器输出的，通过站点 B 的麦克风获得，并且在站点 A 重播，可能导致反馈循环。AEC 是这样克服这个问题的：监视所获得的信号，通过强调室内环境来调整它，然后将它从流出信号中删除。
AEC 有下列局限：

如果声学环境改变，适应滤波器要 2 秒以上的时间来调节。
任何晚回响发生在原信号之后超过 128 毫秒，没有从流出信号中删除。
在一个时刻只有一个捕获 buffer 缓冲器能使用 AEC。
如果下一个应用程序在捕获中，AEC 不能被建立。

通过使用 CaptureBuffer.GetObjectInPath，从捕获buffer缓冲器对象得到特效对象。
使用 CaptureAcousticEchoCancellationEffect对象，来设置并且得到AEC在那些包含AEC特效的捕获buffer缓冲器上的参数。你也能允许噪声填充，那意味着当没有新数据在流出信号时，彻底阻止静音。
噪声抑制只能被应用于也允许AEC的场合。噪声抑制这种捕获特效删除持续的背景噪声，例如鼓风机噪声。使用 CaptureNoiseSuppressEffect对象来设置和得到参数。

第8节　使用捕获缓存

一旦你已经建立一个 CaptureBuffer对象，你将需要控制读写声音数据的过程。捕捉一个声音有以下几步：

1. 通过调用 CaptureBuffer.Start(**true**)启动buffer缓冲器，这个方法设置此捕捉buffer缓冲器到循环模式。
2. 等待直到你要求的数据总数是可获得的。至于一种决定何时捕捉游标到达一个特定点的方法，参看 Capture Buffer Notification。作为选择，你能票选读取游标通过使用 CaptureBuffer.GetCurrentPosition 。
3. 当足够的数据是可利用的，通过调用 CaptureBuffer.Read方法从buffer缓冲器读取数据。
4. 重复steps 2 and 3 直到你准备停止捕捉数据。然后调用 CaptureBuffer.Stop方法。

第5章　优化性能

这个部分提供一些多彩的要点，为了提高那些直接播放 Microsoft® DirectSound® buffer 缓冲器里的音频数据的应用程序的执行性能。

第1节　匹配缓存格式

这个部分提供一些多彩的要点，为了提高那些直接播放 Microsoft® DirectSound® buffer 缓冲器里的音频数据的应用程序的执行性能。

第2节　使用硬件混合

大多数声卡支持某些水准的硬件混频，如果声卡有 Microsoft® DirectSound®驱动的话。下面几点将允许你能进行大多数硬件混频：

在运行时，使用 Device.Caps属性来决定什么格式被声音硬加速设备所支持，并且只使用那些可能的格式。

为那些你使用最多的声音，优先建立声音 buffer 缓冲器们。能被进行硬件混频的 buffer 缓冲器的数目是有限制的。

要强制一个 buffer 缓冲器被用硬件建立，设置 BufferDescription.LocateInHardware属性。如果你这么做，并且用于硬件混频的资源不可使用，则buffer缓冲器的创建将会失败。

使用语音管理来许可DirectSound 分配buffer缓冲器们到硬件资源，当硬件资源可以利用时，并且为了释放硬件资源，强制终止较不重要的buffer缓冲器。参看 Dynamic Voice Management 。

第3节　动态声音管理

选项数目可用于在硬件或软件设备中处理多语音。很多声卡维持它们自己的次要 buffer 缓冲器，并且处理 3-D 特效和它们的混合。这些硬件 buffer 缓冲器，顾名思义，通常居住于系统内存而不是声卡自身。但因为它们被设备混合，它们比软件的 buffer 缓冲器对系统处理器的要求要小很多。因此尽可能的分配硬件给 buffer 缓冲器更有效率，尤其是 3-D buffer 缓冲器。

默认情况，Microsoft® DirectSound® 只要能够，就分配 buffer 缓冲器到硬件。可是它能建立的硬件 buffer 缓冲器数目仅仅和这个设备某一时刻播放的缓冲器数目一样多。硬件 buffer 缓冲器数目因此受限于可以用的硬件语音。DirectSound 在一个 buffer 缓冲器被建立时分配一个硬件语音，并且发布它只在这个 buffer 缓冲器被销毁时；即使当 buffer 缓冲器不在播放的时候，这个语音也不释放。如果一个应用程序创建很多 buffer 缓冲器，机会是那些将要结束的一部分 buffer 缓冲器是软件的 buffer 缓冲器，它们将被 CPU 管理和混合而不是声卡。

要延迟硬件混合和 3-D 特效的资源的分配到buffer缓冲器被播放的时刻，设置 DeferLocation 属性在 BufferDescription 对象，这个对象已被传递给 Buffer(BufferDescription,Device)的buffer缓冲器的创建方法。当你在buffer缓冲器上调用 SecondaryBuffer.Play方法和 SecondaryBuffer.AcquireResources方法的时候，尽可能放置buffer缓冲器在硬件，否则在软件。

当调用 Play方法，你能通过调用下面的一个 BufferPlayFlags标识符，来尝试为buffer缓冲器释放硬件语音，或者前两个标识之一和第三个标识的组合体。DirectSound 然后查找一个在播放的buffer缓冲器，这个buffer缓冲器适应于终止标识指定的。

标识	动作
TerminateByTime	选择比其它候选 buffer 缓冲器播放更长久的那个 buffer 缓冲器。
TerminateByDistance	选择来自听众中最快的 3-D buffer 缓冲器。
TerminateByPriority	当在调用 Play方法中设置的时候，选择候选buffer缓冲器中最低优先权的那个。如果它和前两个之一的标识符组合，则另一个标识符只在解决捆绑时被使用。

如果发现一个合适的buffer缓冲器，DirectSound 终止它并且分配正在使用的语音到这个刚被播放的buffer缓冲器。如果没有适合终止的buffer缓冲器，这个刚被播放的buffer缓冲器备用软件播放，除非 LocateInHardware标识被指定，在指定这个标识的情况下 Play调用会失败。

更多关于怎样延迟buffer缓冲器和处理何时被播放的信息，参看 Buffer.Play 。

当建立buffer缓冲器时，不要使用 StaticBuffer属性和 DeferLocation属性的组合。当StaticBuffer被指定，buffer缓冲器被放置在可以利用的板载内存里，这对应于一些PCI卡的情况。如果你建立了具有 DeferLocation属性的这样一个buffer缓冲器，这个buffer缓冲器内存不能被拷贝到声卡，直到这个buffer缓冲器作为第一次播放，这能导致无法接受的延迟。更多信息，参看 Hardware Acceleration on ISA and PCI Cards。

第4节　在 ISA 和 PCI 声卡上的硬件加速

在过去几年里，声卡已经从 ISA 总线发展到 PCI 总线。这个设计变化根本改变了 Microsoft® DirectSound® 音频文件加速卡之路。另外，DirectSound 已经引进了语音管理器，它能允许应用程序使用有限的硬件加速资源来制造更多的特效。

这个部分论述一些需要考虑的事项在分配 buffer 缓冲器上和在两种声卡上使用语音管理器。

4.1 ISA 卡上的 DirectSound 缓存（略）

4.2 PCI 卡上的 DirectSound 缓存

现代音频卡被设计成连接到 PCI 总线。随着总线的发展，来到了在系统和音频卡间可利用带宽的高速增长期。现在有音频芯片直接到达系统内存来重新获得音频数据给硬件混合，是很实际的。结果，不再值得有板载内存给 buffer 缓冲器。

在 PCI 卡，静态 buffer 缓冲器和流 buffer 缓冲器有着同样的效率。这个BufferDescription.StaticBuffer属性不影响buffer缓冲器的位置，因为所有的buffer缓冲器被定位在系统内存中。

大多数声卡在 Device.Caps属性汇报它们有静态buffer缓冲器，即使它们没有继承在卡上的RAM。这些声卡汇报的静态buffer缓冲器和流buffer缓冲器通常是相同的buffer缓冲器。

4.3 在 ISA 和 PCI 卡上的语音管理器

当建立一个buffer缓冲器时，通过使用 BufferDescription.DeferLocation ，你能延迟分配buffer缓冲器到硬件混频或软件混频内存，直到它们被播放的时刻。更多信息，参看

Dynamic Voice Management。
应用程序的延迟定位的特效的执行性能依靠于用户系统中 DirectSound 加速的种类。
考虑到 PCI 加速器的情况。当一个被延迟的 buffer 缓冲器被播放，DirectSound 首先决定是否有可以利用的硬件来播放声音。假定有，DirectSound 然后确保硬件有权限访问声音数据，那些数据保留它的位置。最后 DirectSound 告诉硬件开始播放。这个过程是既有效率又快速的。
DirectSound 语音管理器分配硬件混和资源，而不是内存。对于 PCI 声卡，buffer 缓冲器占用同样的内存，在它被分配的前后，不管是被分配给硬件混频器还是软件混频器。
如果用户有一个基于 ISA 的加速卡，这个情形就大不相同。这种情况，得到音频数据到一个硬件混频的 buffer 缓冲器的动作是很慢的，因为数据需要通过 ISA 总线，并且在能够被混频之前加载到声卡上的内存。对于被延迟的 buffer 缓冲器，这样会引进一个不能接受的延迟，这个延迟在 play 方法调用时和声音启动的时候之间被造成。因为这个原因，语音管理器不被推荐使用在基于 ISA 的 DirectSound 加速器。
要在ISA设备上，阻止你的程序使用语音管理器是很容易的事。除非BufferDescription.StaticBuffer属性被设置，buffer缓冲器将不被分配给声卡板载的内存。因此，只要你不组合 StaticBuffer 属性和 DeferLocation属性，就没有在播放时buffer缓冲器将被分配给声卡板载的内存的危险。
使用 StaticBuffer属性，通过它本身确保buffer缓冲器将会利用任何ISA卡上存在的内存，但是这不影响PCI声卡。在缺省其它属性的情况，DirectSound 将尝试放置buffer缓冲器与硬件控制下而不是仍然在系统内存中。使用 DeferLocation属性，通过它本身不影响ISA卡，既然buffer缓冲器总是被放置在系统内存中，并且不通过硬件管理。
接下来的表格概述了 StaticBuffer属性和 DeferLocation属性在由ISA设备和PCI设备建立的buffer缓冲器的影响。

声卡	BufferDescription 属性	内存	硬件加速
ISA	StaticBuffer	如果硬件可利用。	是的，如果在硬件内存中。
PCI	StaticBuffer	系统。	是的。如果硬件语音可利用；属性设置没有影响。
ISA	DeferLocation	如果硬件可利用，如果BufferCaps.StaticBuffer被设置。	是的，如果在硬件内存中，但是潜伏于首次播放
PCI	DeferLocation	系统。	是的，如果硬件语音可利用。

要在ISA卡上利用加速器，而没有运行到上述介绍的被延迟分配的危险，首先检查设备的功能。如果声卡汇报超过 0 个静态buffer缓冲器，并且没有流buffer缓冲器，它很可能是一个老式的有板载内存的ISA卡。然后你能使用 StaticBuffer属性，但不要使用DeferLocation属性。如果声卡汇报超过 0 个流buffer缓冲器，它很可能是一款较新式的基于PCI的加速卡。在这种情况，你应该想到用 DeferLocation属性，而不是 StaticBuffer，来最大极限的使用硬件语音。

第5节　控制改变最小化

当你改变面板、音量或者次要 buffer 缓冲器上的频率时，执行性能就被影响了。要防止声

音输出的中断，Microsoft® DirectSound® 混频器必须提前 20 至 100 毫秒混合，甚至提前更多。无论何时你改变控制，混频器不得不犯洪它的"提前混合 buffer 缓冲器"，并且重新混合被改变的声音。
有一个好办法来最小化你发送的控制改变数目。试着减少为了日常事务而调用的频率，这些日常事务使用 SecondaryBuffer.Volume，.Pan和.Frequency。例如，如果你有一个日常事务是和动画桢同步地移动一个声音从左扬声器到右扬声器，试着只设置 Pan每 2 或 3 桢。
3-D 控制改变（方向、位置、速率、多普勒因素等等）也导致DirectSound 重新混合它的"提前混合buffer缓冲器"。可是，你能聚合一个 3-D 控制改变的数目，并且促成一个单一重新混合，参看 3-D Listeners。

第6节　3-D 缓存的 CPU 考虑

软件模拟的 3-D buffer 缓冲器是强烈计算时间的。你应该考虑到这点，当决定什么时候和怎样在你的应用程序中使用 3-D buffer 缓冲器的时候。
尽你可能的几乎不使用 3-D音效，并且不要在那些将不能从特效中真正受益的声音上使用 3-D。设计你的程序，以便于容易的在每个声音上的允许或不允许 3-D特效。你能设置 Buffer3D.Mode属性到 Mode3D.Disable来在任何一个 3-D 声音buffer缓冲器上不允许 3-D处理。
你也能确保 3-D处理在那些离听众太遥远而不能被听到的buffer缓冲器上被暂停。参看 Minimum and Maximum Distances。

第7节　DirectSound 驱动模型

在虚拟设备驱动(VxD)模式下，所有的 Microsoft® DirectSound® 的混频被在 Dsound.vxd 这个虚拟设备驱动里做。Dsound.vxd 也提供近似访问到实际的直接内存存取(DMA) 式的 buffer 缓冲器，这种 DMA 式 buffer 缓冲器 声卡用来从主机 CPU 收到数据。这是和 DirectSound 的主 buffer 缓冲器一样的。一个 DirectSound 应用程序能设置明确的主 buffer 缓冲器属性，比如采样频率和 bit 深度，从而改变了硬件自身的属性。
在微软视窗驱动模式下(WDM)，DirectSound 不能直接访问到声音硬件，除了硬件加速的 buffer 缓冲器。相应的，DirectSound 和核心混频器或者 Kmixer 一起工作。Kmixer 的工作是转换多个音频流格式为一个共有的格式，一起混合它们并且发送结果给硬件。凭感觉说,它做了 Dsound.vxd 要做的。一个主要的不同是 Dsound.vxd 只混合 DirectSound 的 buffer 缓冲器数据，但是 Kmixer 混合所有的视窗音频流数据，包括来自于使用 Microsoft Win32®**waveOut** 函数的数据。DirectSound 和波形音频输出设备不能同时都被打开这个规则，在 WDM 驱动系统不是正确的。
Kmixer 和音频硬件的关系是特别重要的。Kmixer 只是系统的软件，这个软件能指定硬件的 DMA buffer 缓冲器格式。它选择格式是基于那些被要求混合的声音。它设置输出格式给那些被要求混合的声音的高质量格式，或者到硬件支持的最接近的格式。
这有一个非常重要的暗示：DirectSound 不能设置的硬件的 DMA buffer 缓冲器格式。对你的程序而言，这意味着硬件格式是基于你实际播放的数据。如果你播放一个 44 kHz 的文件，Kmixer 将混合所有数据到 44 kHz，并且确保硬件运行在 44 kHz。
作为一个程序开发者，你不能选择使用的驱动模式。那是完全由声卡的类型、windows 版

本，和使用者已经安装的特殊驱动决定的。因为那个原因，在你测试你的程序时，你覆盖所有的可能性是重要的。DirectSound 可能在被 Dsound.vxd 或者 Kmixer 使用，并且你应该确定你程序的行为和执行性能都是可以接受的。

第3篇 DirectX 其它组件

第1章 DirectInput

第1节 枚举 DirectInput 设备

这部分提供了关于Microsoft DirectX应用程序界面（API）的Microsoft DirectInput组件信息。DirectInput API常用于从键盘、鼠标、操纵杆或其他游戏控制器处理数据。更多信息参看 Microsoft.DirectX.DirectInput托管代码参考文档。
为了捕获设备输入的信息使用DirectInput，你不得不知道你想要捕获的device设备来自哪里，和是否它是当前附在用户PC机的。通过枚举 DeviceInstance对象来使用托管类的不同的静态方法，你能做到这个。

1.1 枚举所有的 Available DirectInput 设备

你应该枚举，当你正在查找一个特殊种类的 Devices 设备，或者想提供用户一种设备选择，或需要和 2 个以上设备一起工作时。
枚举服务的 3 个目的

汇报获得的 DirectInput devices 。
为每个 DirectInput devices 提供一个 GUID。
允许你建立一个 DeviceInstance 对象给每个 DirectInput 设备，当它被枚举，以至于你能检查这个设备的类型和功能。

你能枚举所有的可获得的 DirectInput 设备在用户的 PC 机通过使用托管类的 Devices 属性。下面的 C# 代码通常用来组装 1 个 TreeView 控件联系所有的可获得 DirectInput 设备。

```
[C#]
private void PopulateAllDevices(TreeView tvDevices)

{

    //Add "All Devices" node to TreeView
```

```
    TreeNode allNode = new TreeNode("All Devices");
    tvDevices.Nodes.Add(allNode);

    //Populate All devices
    foreach(DeviceInstance di in Manager.Devices)
    {

        //Get Device name
        TreeNode nameNode = new TreeNode(di.InstanceName);

        //Is device attached?
        TreeNode attachedNode = new TreeNode(
            "Attached = " +
            Manager.GetDeviceAttached(di.ProductGuid));

        //Get device Guid
        TreeNode guidNode = new TreeNode(
            "Guid = " + di.InstanceGuid);

        //Add nodes
        nameNode.Nodes.Add(attachedNode);
        nameNode.Nodes.Add(guidNode);
        allNode.Nodes.Add(nameNode);
    }

}
```

1.2 枚举特殊类型的设备

为了枚举例如键盘这样的特殊种类设备，你能使用托管类的 GetDevices方法。GetDevices是个允许你重新得到特定类型的 DeviceInstance对象集合的静态方法，这些特定类型比如

键盘。为了过滤这个集合，它有个参数给你能通过 1 个标识或 1 个连立标识。这种方法，你能重新得到 1 个只附上键盘的列表。

下面的 C# 代码用来组装 1 个 TreeView 控件来附上所有的键盘。

```
[C#]
private void PopulateKeyboards(TreeView tvDevices)

{

    //Add "Keyboard Devices" node to TreeView

    TreeNode keyboardNode = new TreeNode("Keyboard Devices");

    tvInputDevices.Nodes.Add(keyboardNode);


    //Populate Attached Keyboards

    foreach(DeviceInstance di in

Manager.GetDevices(DeviceType.Keyboard,EnumDevicesFlags.AttachedOnly))

    {


        //Get device name

        TreeNode nameNode = new TreeNode(di.InstanceName);

        TreeNode guidNode = new TreeNode(

            "Guid = " + di.InstanceGuid);


        //Add nodes

        nameNode.Nodes.Add(guidNode);

        keyboardNode.Nodes.Add(nameNode);

    }


}
```

1.3 枚举特定类的设备

当枚举指示的设备在用户的 PC 机，你很可能不想只是受限于一个鼠标指针。一些笔记本用户使用轨迹板或其它类型的指针设备。

下面 C# 举例怎么使用 GetDevices 方法来枚举 DeviceClass ，以实现组装 1 个

TreeView 控件，这上面附着你的 PC 机所有的屏幕指针设备，例如鼠标、轨迹板。

```
[C#]
private void PopulatePointers(TreeView tvDevices)

{

    //Add "Pointer Devices" node to TreeView

    TreeNode pointerNode = new TreeNode("Pointer Devices");

    tvInputDevices.Nodes.Add(pointerNode);


    //Populate Attached Mouse/Pointing Devices

    foreach(DeviceInstance di in

Manager.GetDevices(DeviceClass.Pointer,EnumDevicesFlags.AttachedOnly))

    {


        //Get device name

        TreeNode nameNode = new TreeNode(di.InstanceName);

        nameNode.Tag = di;

        TreeNode guidNode = new TreeNode(

            "Guid = " + di.InstanceGuid);


        //Add nodes

        nameNode.Nodes.Add(guidNode);

        pointerNode.Nodes.Add(nameNode);

    }


}
```

注意：这个代码也通常用来枚举所有类型的手柄，只是通过使用 GetDevices方法中的DeviceClass.GameControl值而已。

1.4 枚举所有力反馈设备

有时你也许只是想枚举那些有力反馈能力的设备。因为许多不同类型的设备使用力反馈，你也许必须寻找任何 1 种你要支持的设备。
下面的 C#代码举例说明怎么样枚举所有那些支持力反馈的设备，并且组装 1 个有那些设备的 TreeView 控件。

```
[C#]
private void PopulateForceFeedback(TreeView tvDevices)

{

    //Add "ForceFeedback Devices" node to TreeView

    TreeNode forcefeedbackNode = new TreeNode("ForceFeedback Devices");

    tvInputDevices.Nodes.Add(forcefeedbackNode);


    //Populate ForceFeedback Devices

    foreach(DeviceInstance di in

Manager.GetDevices( DeviceClass.All,EnumDevicesFlags.ForceFeedback))

    {

        //Get device name

        TreeNode nameNode = new TreeNode(di.InstanceName);

        nameNode.Tag = di;

        TreeNode guidNode = new TreeNode("Guid = " + di.InstanceGuid);


        //Add nodes

        nameNode.Nodes.Add(guidNode);

        forcefeedbackNode.Nodes.Add(nameNode);

    }

}
```

第2节　从 DirectSound 设备中进行捕获

使用DirectInput捕获设备输入可以用 3 个简单步骤来完成。首先建立一个 Device对象，它代表了你想捕获的输入设备；建立Device对象之后你要做好所有必要的配置，诸如调用

SetCooperativeLevel方法；一旦完成了设备的配置，你就可以开始检查程序中设备的状态。

2.1 建立 DirectInput Device 对象

和 Enumerating DirectInput Devices中讨论的一样，有很多方法可以找出当前有哪些输入设备与用户计算机相关联。通常你将必须从多于 1 个的设备中捕获输入。
下面的 C# 范例 代码描述了如何建立 3 个不同的 Device 对象：一个代表用户计算机默认键盘，一个代表默认鼠标，一个代表操纵杆。

```
[C#]
private void InitDevices()

{

    //create keyboard device.

    keyboard = new Device(SystemGuid.Keyboard);

    if(keyboard == null) throw new Exception("No keyboard found.");


    //create mouse device.

    mouse = new Device(SystemGuid.Mouse);

    if(mouse == null)

    {

        throw new Exception("No mouse found.");

    }


    //create joystick device.

    foreach(

        DeviceInstance di in

        Manager.GetDevices(

            DeviceClass.GameControl,

            EnumDevicesFlags.AttachedOnly))

    {

        joystick = new Device(di.InstanceGuid);

        break;
```

```
    }


    if(joystick == null)
    {
        //Throw exception if joystick not found.
        throw new Exception("No joystick found.");
    }
}
```

2.2 配置 DirectInput Device 对象

因为你捕获的输入设备来自于一个共享系统资源，所以你必须配置它们与其他系统资源合作。你可以调用你建立的 Device 对象的 SetCooperativeLevel 方法并组合 CooperativeLevelFlags枚举的标识符来完成这个配置。下列表格描述了设置cooperative level时使用的 CooperativeLevelFlags标识符。

标识	描述
CooperativeLevelFlags.Background	这个设备可以用于后台并在任何时候被获取，即使程序窗体没有激活。
CooperativeLevelFlags.Foreground	这个设备仅在程序窗体处于前台活动时使用。当程序窗体失去焦点，该设备将不可再获得。
CooperativeLevelFlags.Exclusive	这个设备需要独占访问。其它程序不能获得这个设备。安全起见，这个标识符不能和诸如键盘和鼠标之类特定设备上的后台标识符协同使用。
CooperativeLevelFlags.NonExclusive	这个设备可以被不需要独占访问的其他程序所共享。
CooperativeLevelFlags.NoWindowsKey	这个标识符禁用视窗（WIN）键。

另一个配置任务是设置操纵杆的轴范围。它允许你对操纵杆轴上的全部移动距离进行数值范围设定。即使你使用的操纵杆不支持所设定的分辨率范围，DirectInput 也会将操纵杆的分辨率转化到你指定的范围中。

下面 C# 范例 代码示范了如何对前例中建立的设备进行配置。

```
[C#]
//Set mouse axis mode absolute.

mouse.Properties.AxisModeAbsolute = true;



//Set joystick axis ranges.

foreach(DeviceObjectInstance doi in joystick.Objects)
```

```
{
    if((doi.ObjectId & (int)DeviceObjectTypeFlags.Axis) != 0)
    {
        joystick.Properties.SetRange(
            ParameterHow.ById,
            doi.ObjectId,
            new InputRange(-5000,5000));
    }
}

//Set joystick axis mode absolute.
joystick.Properties.AxisModeAbsolute = true;

//set cooperative level.
keyboard.SetCooperativeLevel(
    this,
    CooperativeLevelFlags.NonExclusive |
    CooperativeLevelFlags.Background);

mouse.SetCooperativeLevel(
    this,
    CooperativeLevelFlags.NonExclusive |
    CooperativeLevelFlags.Background);

joystick.SetCooperativeLevel(
    this,
    CooperativeLevelFlags.NonExclusive |
    CooperativeLevelFlags.Background);
```

```
//Acquire devices for capturing.

keyboard.Acquire();

mouse.Acquire();

joystick.Acquire();
```

2.3 从 DirectInput Device 对象进行捕获

通常你想在程序的一个特定点捕获 device 对象的状态。
下面的 C#范例 代码包含了从前例建立的设备中捕获设备输入和 更新 标签控制的方法。
程序可以在你需要捕获 DirectInput 设备输入的任何时候调用它们 。

```
[C#]
private void UpdateKeyboard()

{

    string info = "Keyboard: ";


    //Capture pressed keys.

    foreach(Key k in keyboard.GetPressedKeys())

    {

        info += k.ToString() + " ";

    }

    lbKeyboard.Text = info;

}


private void UpdateMouse()

{

    string info = "Mouse: ";


    //Get Mouse State.

    MouseState state = mouse.CurrentMouseState;


    //Capture Position.
```

```
    info += "X:" + state.X + " ";
    info += "Y:" + state.Y + " ";
    info += "Z:" + state.Z + " ";

    //Capture Buttons.
    byte[] buttons = state.GetMouseButtons();
    for(int i = 0; i < buttons.Length; i++)
    {
        if(buttons[i] != 0)
        {
            info += "Button:" + i + " ";
        }
    }

    lbMouse.Text = info;
}

private void UpdateJoystick()
{
    string info = "Joystick: ";

    //Get Mouse State.
    JoystickState state = joystick.CurrentJoystickState;

    //Capture Position.
    info += "X:" + state.X + " ";
    info += "Y:" + state.Y + " ";
    info += "Z:" + state.Z + " ";
```

```
    //Capture Buttons.
    byte[] buttons = state.GetButtons();
    for(int i = 0; i < buttons.Length; i++)
    {
        if(buttons[i] != 0)
        {
            info += "Button:" + i + " ";
        }
    }

    lbJoystick.Text = info;
}
```

第3节　使用力反馈

力反馈允许你发送些消息和属性到用户的 DirectInput 设备，来得到像物理力量的转播回馈。这仅仅能应用于支持力反馈的设备。

3.1 力反馈的类型

有 2 种方法回馈物理力量到 input 设备。1 种是下载进入到设备的力反馈效果文件。一个力反馈效果文件包含了一个效果集合，一旦下载，它能被在任何的时候播放。
另一种是分配一个被设备支持的特殊效果。这些效果有参数能被配置和分派给设备的轴。下面是那些被 DirectX 9.0 托管版本包括的效果的列表。

ForceFeedbackGuid.ConstantForce
ForceFeedbackGuid.CustomForce
ForceFeedbackGuid.Damper
ForceFeedbackGuid.Friction
ForceFeedbackGuid.Inertia
ForceFeedbackGuid.RampForce
ForceFeedbackGuid.SawtoothDown
ForceFeedbackGuid.SawtoothUp
ForceFeedbackGuid.Sine
ForceFeedbackGuid.Spring
ForceFeedbackGuid.Square
ForceFeedbackGuid.Triangle

3.2 建立和配置一个力反馈设备

为了使用力反馈，你必须首先为具有力反馈功能的设备建立一个 Device 对象。下面的 C# 代码建立了一个来自于 1st 可获得的力反馈游戏控制器的 Device 对象。

```
[C#]
//create force feedback capable joystick device.

foreach(DeviceInstance di in Manager.GetDevices(

    DeviceClass.GameControl,

    EnumDevicesFlags.AttachedOnly | EnumDevicesFlags.ForceFeedback))

{

    joystick = new Device(di.InstanceGuid);

    break;

}

if (joystick == null)

{

    //Throw exception if joystick not found.

    throw new Exception("No joystick that supports forced feedback found.");

}
```

一旦 Device对象被建立，它需要被合理的配置来和力反馈工作。为了力反馈工作得正确，你必须设置cooperative合作级别为exclusive独占。你也需要建立一个轴的数组来使用力反馈。
下面的 C# 代码阐明怎么配置前面建立好了的 Device 对象。

```
[C#]
//set cooperative level.

joystick.SetCooperativeLevel(

    this,

    CooperativeLevelFlags.Exclusive | CooperativeLevelFlags.Background);


//Set axis mode absolute.

joystick.Properties.AxisModeAbsolute = true;


//Acquire joystick for capturing.
```

```
joystick.Acquire();

//Configure axes
int[] axis = null;
foreach(DeviceObjectInstance doi in joystick.Objects)
{

    //Set axes ranges.
    if((doi.ObjectId & (int)DeviceObjectTypeFlags.Axis) != 0)
    {
        joystick.Properties.SetRange(
        ParameterHow.ById,
        doi.ObjectId,
        new InputRange(-5000,5000));
    }

    int[] temp;

    // Get info about first two FF axii on the device
    if ((doi.Flags & (int)ObjectInstanceFlags.Actuator) != 0)
    {
        if (axis != null)
        {
            temp = new int[axis.Length + 1];
            axis.CopyTo(temp,0);
            axis = temp;
        }
        else
        {
```

```
            axis = new int[1];
        }


        // Store the offset of each axis.
        axis[axis.Length - 1] = doi.Offset;
        if (axis.Length == 2)
        {
            break;
        }
    }
}
```

3.3 加载一个力反馈效果文件

下面的 C# 代码表明怎么样下载一个 力反馈效果文件到一个在先前范例中被建立和配置的 Device 对象。

```
[C#]
//Load force feedback effect file

string path =

    @"C:\Program Files\Microsoft DirectX 9.0 SDK (December
2004)\Samples\Managed\DirectInput\ReadFFE\guna.ffe";



EffectList el = null;

el = joystick.GetEffects(path,FileEffectsFlags.ModifyIfNeeded);

foreach(FileEffect fe in el)

{

    EffectObject feo = new EffectObject(

        fe.EffectGuid,

        fe.EffectStruct,

        joystick);

    try

    {
```

```
        feo.Download();

    }

    catch(Exception ex)

    {

        throw new Exception("Could not download force feedback effect file.",
ex);

    }

    fxList.Add(feo);


}
```

一旦上面的力反馈效果文件被下载到一个 Device 对象，它能被下面的 C# 范例播放。

```
[C#]
//Play gun force feedback file effect

foreach(EffectObject eo in fxList)

{

    eo.Start(1,EffectStartFlags.NoDownload);

}

button0pressed = true;
```

3.4 给设备分配一个预先效果

如果你想反馈一个物理力量，比如到 Device设备对象轴的一个恒力，你能通过分配一个预先效果来做到。下面的 C# 首先检查这个 Device 设备对象是否支持这个效果；然后建立这些效果，给这些效果设置参数，然后应用它们到这个 Device 对象的轴。

```
[C#]
//See if joystick supports ConstantForce and set it.

EffectObject eo = null;

Effect e;


foreach (EffectInformation ei in joystick.GetEffects(EffectType.All))

{

    //If the joystick supports ConstantForce, then apply it.
```

```
    if (DInputHelper.GetTypeCode(ei.EffectType)
        == (int)EffectType.ConstantForce)
    {
        // Fill in some generic values for the effect.
        e = new Effect();
        e.SetDirection(new int[axis.Length]);
        e.SetAxes(new int[axis.Length]);
        e.ConditionStruct =   new Condition[axis.Length];

        e.EffectType = EffectType.ConstantForce;
        e.Duration = (int)DI.Infinite;
        e.Gain = 10000;
        e.SamplePeriod = 0;
        e.TriggerButton = (int)Microsoft.DirectX.DirectInput.Button.NoTrigger;
        e.TriggerRepeatInterval = (int)DI.Infinite;
        e.Flags = EffectFlags.ObjectOffsets | EffectFlags.Cartesian;
        e.SetAxes(axis);

        // Create the effect, using the passed in guid.
        eo = new EffectObject(ei.EffectGuid, e, joystick);

    }
}

if (eo == null)
{
    throw new Exception("ConstantForce is not supported.");
}
```

第2章　Audio Video PlayBack

第1节　音频/视频回放

音视频回放API提供了基本的回放和简单控制音视频文件。更多信息参看Microsoft.DirectX.AudioVideoPlayback 托管代码的引用文档。
使用 Video类来播放视频文件，包括那些含有音频的视频文件。使用 Audio类来播放只含有音频的文件。你也能使用 Audio类来控制这个你播放的视频文件的音频属性。这个SeekPositionFlags枚举控制了寻找操作，并且 StateFlags有标识来指示是否这个媒体文件正在运行、暂停或者停止。
这个Audio类主要为了简单的回放情形被设计，否则使用Video类。你也能使用Microsoft® DirectSound® 来播放音频文件，它在音频回放方面给了你强大得多的控制。

1.1 播放一个视频文件

要播放视频文件，一开始要通过创建一个 Video类的实例。你能在 Video构造器指定文件名，就像接下来的C#代码范例，否则调用文件名的 Open 方法 。

```
[C#]
using Microsoft.DirectX.AudioVideoPlayback;

public class MyVideoPlayer : System.Windows.Forms.Form

{

    /* ... */

    private void OpenFile()

    {

        try

        {

            Video ourVideo = new Video("C:\\Example.avi");

            /* ... */

        }

    }

    /* ... */

}
```

这个 Video对象抛出一个异常，如果你试着打开一个不包含视频的文件。下一步，在你的应用程序中制定一个父窗体来支持 Video对象的视频窗口，如下：

```
[C#]
ourVideo.Owner = this;   // 'this' refers to the application's Form object.
```

通过调用 Play、 Pause和 Stop方法来控制回放。例如：下面的事件处理器停止了回放。

```
[C#]
private void mnuStop_Click(object sender, System.EventArgs e)

{

    if (ourVideo != null)

    {

        ourVideo.Stop();

        }

}
```

要设置回放窗口的尺寸，设置 Size属性，它将得到一个 System.Drawing.Size对象，如下：

```
[C#]
ourVideo.Size = new Size(480, 320);
```

你能通过检查 DefaultSize属性得到本地视频尺寸。如果视频文件包含音频，这个 Video.Audio属性返回一个 Audio对象。你能使用这个对象来设置音量或者音频的立体声平衡。如果文件不包含音频，设置这些属性导致一个异常。使用一个try模块包围这段代码，如下：

```
[C#]
try

{

    Video.Audio.Volume = 100;

}
```

1.2 播放一个音频文件

一个 Audio对象是类似于 Video对象的，但是它支持的属性是关系到音频的，比如 Volume和 Balance。要播放一个音频文件，就要在 Audio构造器中指定文件名，就像接下来的C#代码，否则调用文件名的 Open方法。

```
[C#]
Audio ourAudio = new Audio("C:\MyAudioFile.wav");
```

第2节　音频/视频回放错误代码

下面是发生在 Microsoft.DirectX.AudioVideoPlayback中的 1 个可能错误代码列表。

声频 视频 错误代码表

错误代码	错误字符串	描述
-2147220992	VFW_E_INVALIDMEDIATYPE	The specified media type is invalid.
-2147220991	VFW_E_INVALIDSUBTYPE	The specified media subtype is invalid.
-2147220990	VFW_E_NEED_OWNER	This object can only be created as an aggregated object.
-2147220989	VFW_E_ENUM_OUT_OF_SYNC	The state of the enumerated object has changed and is now inconsistent with the state of the enumerator. Discard any data obtained from previous calls to the enumerator and then update the enumerator by calling the enumerator's Reset method.
-2147220988	VFW_E_ALREADY_CONNECTED	At least one of the pins involved in the operation is already connected.
-2147220987	VFW_E_FILTER_ACTIVE	This operation cannot be performed because the filter is active.
-2147220986	VFW_E_NO_TYPES	One of the specified pins supports no media types.
-2147220985	VFW_E_NO_ACCEPTABLE_TYPES	There is no common media type between these pins.
-2147220984	VFW_E_INVALID_DIRECTION	Two pins of the same direction cannot be

		connected.
-2147220983	VFW_E_NOT_CONNECTED	The operation cannot be performed because the pins are not connected.
-2147220982	VFW_E_NO_ALLOCATOR	No sample buffer allocator is available.
-2147220981	VFW_E_RUNTIME_ERROR	A run-time error occurred.
-2147220980	VFW_E_BUFFER_NOTSET	No buffer space has been set.
-2147220979	VFW_E_BUFFER_OVERFLOW	The buffer is not sufficiently large.
-2147220978	VFW_E_BADALIGN	An invalid alignment was specified.
-2147220977	VFW_E_ALREADY_COMMITTED	The allocator was not committed.
-2147220976	VFW_E_BUFFERS_OUTSTANDING	One or more buffers are still active.
-2147220975	VFW_E_NOT_COMMITTED	Cannot allocate a sample when the allocator is not active.
-2147220974	VFW_E_SIZENOTSET	Cannot allocate memory because no size has been set.
-2147220973	VFW_E_NO_CLOCK	Cannot lock for synchronization because no clock has been defined.
-2147220972	VFW_E_NO_SINK	Quality messages could not be sent because no quality sink has been defined.
-2147220971	VFW_E_NO_INTERFACE	A required interface has not been implemented.
-2147220970	VFW_E_NOT_FOUND	An object or name was not found.
-2147220969	VFW_E_CANNOT_CONNECT	No combination of intermediate

		filters could be found to make the connection.
-2147220968	VFW_E_CANNOT_RENDER	No combination of filters could be found to render the stream.
-2147220967	VFW_E_CHANGING_FORMAT	Could not change formats dynamically.
-2147220966	VFW_E_NO_COLOR_KEY_SET	No color key has been set.
-2147220965	VFW_E_NOT_OVERLAY_CONNECTION	Transport.
-2147220964	VFW_E_NOT_SAMPLE_CONNECTION	Current pin connection is not using the correct transport.
-2147220963	VFW_E_PALETTE_SET	Setting a color key would conflict with the palette already set.
-2147220962	VFW_E_COLOR_KEY_SET	Setting a palette would conflict with the color key already set.
-2147220961	VFW_E_NO_COLOR_KEY_FOUND	No matching color key is available.
-2147220960	VFW_E_NO_PALETTE_AVAILABLE	No palette is available.
-2147220959	VFW_E_NO_DISPLAY_PALETTE	Display does not use a palette.
-2147220958	VFW_E_TOO_MANY_COLORS	Too many colors for the current display settings.
-2147220957	VFW_E_STATE_CHANGED	The state changed while waiting to process the sample.
-2147220956	VFW_E_NOT_STOPPED	The operation could not be performed because the filter is not stopped.
-2147220955	VFW_E_NOT_PAUSED	The operation could not be performed because the filter is not paused.

-2147220954	VFW_E_NOT_RUNNING	The operation could not be performed because the filter is not running.
-2147220953	VFW_E_WRONG_STATE	The operation could not be performed because the filter is in the wrong state.
-2147220952	VFW_E_START_TIME_AFTER_END	The sample start time is after the sample end time.
-2147220951	VFW_E_INVALID_RECT	The supplied rectangle is invalid.
-2147220950	VFW_E_TYPE_NOT_ACCEPTED	This pin cannot use the supplied media type.
-2147220949	VFW_E_SAMPLE_REJECTED	This sample cannot be rendered.
-2147220948	VFW_E_SAMPLE_REJECTED_EOS	This sample cannot be rendered because the end of the stream has been reached.
-2147220947	VFW_E_DUPLICATE_NAME	An attempt to add a filter with a duplicate name failed.
-2147220946	VFW_E_TIMEOUT	A time-out has expired.
-2147220945	VFW_E_INVALID_FILE_FORMAT	The file format is invalid.
-2147220944	VFW_E_ENUM_OUT_OF_RANGE	The list has already been exhausted.
-2147220943	VFW_E_CIRCULAR_GRAPH	The filter graph is circular.
-2147220942	VFW_E_NOT_ALLOWED_TO_SAVE	Updates are not allowed in this state.
-2147220941	VFW_E_TIME_ALREADY_PASSED	An attempt was made to queue a command for a time in the past.
-2147220940	VFW_E_ALREADY_CANCELLED	The queued command was

		already canceled.
-2147220939	VFW_E_CORRUPT_GRAPH_FILE	Cannot render the file because it is corrupt.
-2147220938	VFW_E_ADVISE_ALREADY_SET	Advise link already exists.
-2147220936	VFW_E_NO_MODEX_AVAILABLE	No full-screen modes are available.
-2147220935	VFW_E_NO_ADVISE_SET	This Advise cannot be canceled because it was not successfully set.
-2147220934	VFW_E_NO_FULLSCREEN	Full-screen mode is not available.
-2147220933	VFW_E_IN_FULLSCREEN_MODE	Window methods while in full-screen mode.
-2147220928	VFW_E_UNKNOWN_FILE_TYPE	The media type of this file is not recognized.
-2147220927	VFW_E_CANNOT_LOAD_SOURCE_FILTER	The source filter for this file could not be loaded.
-2147220925	VFW_E_FILE_TOO_SHORT	A file appeared to be incomplete.
-2147220924	VFW_E_INVALID_FILE_VERSION	The file's version number is invalid.
-2147220921	VFW_E_INVALID_CLSID	This file is corrupt: it contains an invalid class identifier.
-2147220920	VFW_E_INVALID_MEDIA_TYPE	This file is corrupt: it contains an invalid media type.
-2147220919	VFW_E_SAMPLE_TIME_NOT_SET	No time stamp has been set for this sample.
-2147220911	VFW_E_MEDIA_TIME_NOT_SET	No media time was set for this sample.
-2147220910	VFW_E_NO_TIME_FORMAT_SET	No media time format was selected.

-2147220909	VFW_E_MONO_AUDIO_HW	Cannot change balance because audio device is monaural only.
-2147220907	VFW_E_NO_DECOMPRESSOR	Cannot play back the video stream: could not find a suitable decompressor.
-2147220906	VFW_E_NO_AUDIO_HARDWARE	Cannot play back the audio stream: no audio hardware is available, or the hardware is not supported.
-2147220903	VFW_E_RPZA	Cannot play back the video stream: format 'RPZA' is not supported.
-2147220901	VFW_E_PROCESSOR_NOT_SUITABLE	Microsoft® DirectShow® cannot play MPEG movies on this processor.
-2147220900	VFW_E_UNSUPPORTED_AUDIO	Cannot play back the audio stream: the audio format is not supported.
-2147220899	VFW_E_UNSUPPORTED_VIDEO	Cannot play back the video stream: the video format is not supported.
-2147220898	VFW_E_MPEG_NOT_CONSTRAINED	DirectShow cannot play this video stream because it falls outside the constrained standard.
-2147220897	VFW_E_NOT_IN_GRAPH	Cannot perform the requested function on an object that is not in the filter graph.
-2147220895	VFW_E_NO_TIME_FORMAT	Cannot access the time format on an object.
-21472208	VFW_E_READ_ONLY	Could not make

94		the connection because the stream is read-only and the filter alters the data.
-2147220892	VFW_E_BUFFER_UNDERFLOW	The buffer is not full enough.
-2147220891	VFW_E_UNSUPPORTED_STREAM	Cannot play back the file: the format is not supported.
-2147220890	VFW_E_NO_TRANSPORT	Pins cannot connect because they don't support the same transport.
-2147220887	VFW_E_BAD_VIDEOCD	The Video CD can't be read correctly by the device or the data is corrupt.
-2147220880	VFW_S_NO_STOP_TIME	The sample had a start time but not a stop time. In this case, the stop time that is returned is set to the start time plus one.
-2147220879	VFW_E_OUT_OF_VIDEO_MEMORY	There is not enough video memory at this display resolution and number of colors. Reducing resolution might help.
-2147220878	VFW_E_VP_NEGOTIATION_FAILED	The video port connection negotiation process has failed.
-2147220877	VFW_E_DDRAW_CAPS_NOT_SUITABLE	Either Microsoft DirectDraw® has not been installed or the video card capabilities are not suitable. Make sure the display is not in 16-color mode.

-2147220876	VFW_E_NO_VP_HARDWARE	No video port hardware is available, or the hardware is not responding.
-2147220875	VFW_E_NO_CAPTURE_HARDWARE	No capture hardware is available, or the hardware is not responding.
-2147220874	VFW_E_DVD_OPERATION_INHIBITED	This user operation is inhibited by digital video disc (DVD) content at this time.
-2147220873	VFW_E_DVD_INVALIDDOMAIN	This operation is not permitted in the current domain.
-2147220872	VFW_E_DVD_NO_BUTTON	Requested button is not available.
-2147220871	VFW_E_DVD_GRAPHNOTREADY	DVD-Video playback graph has not been built yet.
-2147220870	VFW_E_DVD_RENDERFAIL	DVD-Video playback graph building failed.
-2147220869	VFW_E_DVD_DECNOTENOUGH	DVD-Video playback graph could not be built due to insufficient decoders.
-2147220853	VFW_E_DVD_NOT_IN_KARAOKE_MODE	The DVD Navigator is not in karaoke mode.
-2147220850	VFW_E_FRAME_STEP_UNSUPPORTED	Frame stepping is not supported.
-2147220845	VFW_E_PIN_ALREADY_BLOCKED_ON_THIS_THREAD	Pin is already blocked on the calling thread.
-2147220844	VFW_E_PIN_ALREADY_BLOCKED	Pin is already blocked on another thread.
-2147220843	VFW_E_CERTIFICATION_FAILURE	Use of this filter is restricted by a software key. The application must unlock the filter.

-2147220842	VFW_E_VMR_NOT_IN_MIXER_MODE	The Video Mixing Renderer (VMR) is not in mixing mode.
-2147220841	VFW_E_VMR_NO_AP_SUPPLIED	The application has not yet provided the VMR filter with a valid allocator-presenter object.
-2147220840	VFW_E_VMR_NO_DEINTERLACE_HW	The VMR could not find any deinterlacing hardware on the current display device.
-2147220839	VFW_E_VMR_NO_PROCAMP_HW	The VMR could not find any hardware that supports ProcAmp controls on the current display device.
-2147220838	VFW_E_DVD_VMR9_INCOMPATIBLEDEC	The hardware decoder uses video port extensions (VPE), which are not compatible with the VMR-9 filter.
-2147220494	VFW_E_BAD_KEY	A registry entry is corrupt.

第4篇 我如何上手？

这个篇章的文档展示了怎么完成通常遇到的 Microsoft® DirectX®开发任务。
下面的表格列出了使用 C#代码范例的 DirectX 特性区域来说可用的任务。

处理 1 个设备 (Direct3D)

Check for Shader Support	例子展示了怎样检查支持着色器的硬件设备。
Create an Additional Swap Chain	例子示范了使用 SwapChain类建立一个额外的swap chain。额外的交换链表对于多视口的支持是有用的；例如 1 个单一窗体被分为 4 个子窗体，每个子窗体是同一场景的不同视角。
Maintain Floating	当一个 Device对象被创建时，中间语言运行时CLR将浮点单元

Point Precision after Device Creation	(FPU)改变为单精度以维持更好的性能。要维持CLR默认的双精度FPU，应和下面范例代码中一样，在建立 1 个 Device对象时使用CreateFlags.FpuPreserve标识。
Record and Apply StateBlocks	例子示范了如何使用 state blocks 设置并获取 1 个设备状态的返回值。

渲染 1 个 3-D 场景(Direct3D)

Draw A Sprite	例子示范了如何绘制一个动画精灵。
Generate a Scene	例子示范了如何开始场景生成并绘制造型。
Render Mesh with Texture using HLSL	例子示范了如何使用高级着色器语言（HLSL）渲染 1 个网格纹理。
Set Up a Projection Matrix	例子示范了如何设立投影变换矩阵，该矩阵将 3-D 照相机或视点空间坐标变换为 2-D 屏幕坐标。
Set Up a View Matrix	例子示范了如何初始化视点变换矩阵，该矩阵将世界坐标变换为照相机或视点空间。
Using an Effect	例子示范了如何使用 1 个从文件中载入特效的技术。

使用网格、纹理和特效(Direct3D)

Check For HDR Texture Support	例子示范了如何检查 1 个设备以决定是否支持 1 个指定纹理格式。
Clean a Mesh	例子示范了如何为简化工作准备清理 1 个网格。
Clone a Mesh	例子示范了如何将 1 个网格克隆加入到 1 个原来不存在法线、纹理坐标、色彩、重量等等特性的空间之中。
Compute a Bounding Sphere from a Mesh	例子示范了如何使用 Geometry.ComputeBoundingSphere 方法产生 1 个围绕 3-D对象的简单弹球。
Copy a Texture	例子示范了如何复制 1 个纹理。
Create a Cube Texture	例子示范了如何建立 1 个立方体纹理。
Create a Mesh Object	This example shows how to create a Mesh object. 例子示范了如何建立 1 个 Mesh对象。
Save a Screenshot	例子示范了如何将 1 个屏幕快照保存为 1 个文件。
Simplify a Mesh	例子示范了如何简化 1 个网格。

使用顶点和索引缓存

Create a Vertex Declaration	例子示范了如何建立 1 个顶点声明。
Read and Write VertexBuffer and IndexBuffer Data With GraphicsStreams	例子示范了如何使用 GraphicsStream对象填充并从 1 个 VertexBuffer 和 IndexBuffer对象中获取返回值。
Read and Write VertexBuffer Data With Arrays	例子示范了如何使用数组填充并从 1 个 VertexBuffer 和 IndexBuffer对象中获取返回

	值。

使用声音特效

Add Effects to a SecondaryBuffer Object	这个C#例子示范了如何将特效对象加入到 1 个SecondaryBuffer对象中。
Use Effect Parameters	这个C#例子示范了如何从 1 个 SecondaryBuffer对象中使用 1 个特效对象的参数。

第1章　处理 1 个设备

第1节　检查着色器支持

这个例子展示了怎样检查硬件设备支持着色器。
要决定硬件设备是否支持着色器，Microsoft® Direct3D®允许应用程序检查着色器版本。
要检查着色器支持：

1. 通过使用 Manager.GetDeviceCaps方法，获得设备的功能。
2. 经过前面方法的调用使用获得的对象的功能，通过调用 Caps.VertexShaderVersion方法检查着色器版本。

```
[C#]



public bool CheckShaderSupport()

{

    Version v1_1 = new Version(1,1); // check version is at least shader 1.1


    // retrieve the device caps

    Caps caps = Manager.GetDeviceCaps(0, DeviceType.Hardware);


    // check the supported shader version

    if ((caps.VertexShaderVersion >= v1_1) && (caps.PixelShaderVersion >=
v1_1))

    {

        return true;

    }
```

```
    return false;
}
```

第2节　建立 1 个额外的交换链表

这个示范展示了使用 SwapChain类，怎样创建一个额外的交换链swap chain。 额外的交换链在支持多视口是有用的；例如，一个单一窗体被瓜分为 4 个子窗体，每一个子窗体是同一个场景的不同视图。
在托管代码版里，每一个 Device被用一个默认的交换链建立的，它作为隐式交换链而为人所知的。使用 SwapChain类允许为渲染操作创建额外的交换链。
要建立一个新的交换链：

1. 创建一个 PresentParameters类的实例，或者使用一个现有的实例，并且设置陈述属性来评估你对于交换链的需要。
2. 然后使用 SwapChain类的构造器，创建一个交换链对象。

下面的C#范例，device假定是一个被渲染的 Device设备。

```
[C#]



    // Create a swap chain using an existing instance of PresentParameters.
    SwapChain sc = new SwapChain(device, presentParams);
```

第3节　在设备建立后维持浮点精度

当一个 Device对象被创建，中间语言运行时CLR将会改变浮点单元(FPU)为单精度，以维持较好的执行性能。要维护CLR默认的双精度FPU，当创建像下面的范例代码一样的一个Device对象时，要使用 CreateFlags.FpuPreserve 标识。

```
[C#]
Device device = null;                    // Create rendering device
        PresentParameters presentParams = new PresentParameters();


        device = new Device(0, DeviceType.Hardware, this,
                            CreateFlags.SoftwareVertexProcessing |
                            CreateFlags.FpuPreserve, presentParams);
```

第4节　记录和应用 1 个 StateBlock

这个范例展示了怎样使用 state blocks 状态块来设置和重新得到一个设备的状态。

一个 StateBlock实例能被用来返回一个设备到默认状态。它也能被储存作为一个类的成员，这个类需要快速恢复一个想要的渲染状态。一个 StateBlock是很有用的，通常用于这种场合：一旦一个设备被建立，就要保存默认设备状态。当需要在渲染期间，它允许快速恢复默认状态。

要记录和应用一个 state block 状态块：

1. 调用 Device.BeginStateBlock方法来开始记录设备状态。
2. 告诉设备状态和值以供记录。
3. 调用 Device.EndStateBlock方法来停止正在记录的设备状态，并且返回一个包含记录状态的 StateBlock实例。
4. 通过使用前先获得的 StateBlock实例的 StateBlock.Capture方法调用，捕获设备的当前状态。
5. 做一些改变设备状态的处理。
6. 通过调用 StateBlock.Apply，恢复设备的被保存状态。

下面的C#代码，device被假定是正在渲染的 Device设备。

```
[C#]


// begin recording device state

device.BeginStateBlock();


// tell device the states and values to record

device.RenderState.FogStart = 0.3f;

device.RenderState.FogEnd = 100.0f;

device.RenderState.FogColor = Color.Gray;


// stop recording device state

StateBlock sb = device.EndStateBlock();


// save device states recorded above

sb.Capture;


// change device states

device.RenderState.FogStart = 2000.0f;

device.RenderState.FogEnd = 35000.0f;
```

```
device.RenderState.FogColor = Color.Yellow;


// render scene w/changed states
device.DrawPrimitives();


// restore device's saved states
sb.Apply();


// render scene w/saved states
device.DrawPrimitives();
```

第2章　渲染 1 个 3-D 场景

第1节　绘制 1 个动画精灵

例子示范了怎样绘制一个动画精灵。
要绘制一个动画精灵：

1. 调用 Sprite.Begin方法为绘制动画精灵的设备做准备。
2. 调用 Sprite.Draw2D渲染动画精灵。
3. 调用 Sprite.End通告这批动画精灵的完成。

下面的C#代码，device被假定为正在渲染的 Device。texture变量是一个被加载的 Texture对象。

```
    [C#]


sprite.Begin(SpriteFlags.None);


sprite.Draw2D(texture, Rectangle.Empty, Rectangle.Empty,
              new Point(5.0f, 5.0f), Color.White);
sprite.End();
```

第2节　产生 1 个场景

例子展示了怎样开始产生场景和绘制造型。

SetStreamSource方法绑定了一个顶点buffer 缓冲器到一个设备数据流，来建立一个关于顶点数据和几个数据流端口之一的联合，这些端口流入原始的处理函数。这个方法的参数是数据流的数目，VertexBuffer对象的name，和流顶点步幅。
下面的C#代码，device被假定为正在渲染的 Device 。 vBuffer是一个用CustomVertex.PositionNormal数据填充的顶点buffer 缓冲器。

```
  [C#]



device.BeginScene();


device.SetStreamSource(0, vBuffer, 0);

device.VertexFormat = CustomVertex.PositionNormal.Format;

device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);


device.EndScene();
```

第3节　使用 HLSL 渲染 1 个具有纹理的网格

范例示范了使用高级着色器语言(HLSL)，怎样渲染一个网格纹理。
下面的C#代码，假定有以下假设：

1. effect是一个有效的HLSL Effect，它有技术设置。
2. 代码发生在 Effect.BeginPass调用和 Effect.EndPass调用之间。
3. mesh是一个有效的 Mesh。
4. meshTextures 是一个有效的网格纹理数组。

```
  [C#]



effect.SetValue("WorldViewProjection", worldMatrix);


// Iterate through each subset and render with its texture

for (int m = 0; m < meshTextures.Length; ++m)

{

    effect.SetValue("SceneTexture", meshTextures[m]));

    effect.CommitChanges();
```

```
    mesh.DrawSubset(m);
}
```

第4节　设置 1 个投影矩阵

例子示范了如何设置投影变换矩阵，该矩阵将 3-D 照相机或视点空间坐标变换为 2-D 屏幕坐标。
看下面的C#范例，Projection投影变换矩阵被设置等价于左手 PerspectiveFovLH矩阵。PerspectiveFovLH的一些要点如下：

1. 观察区域的弧度：Pi/4。
2. 屏幕高宽比，或者说视野空间的高比上宽：1，因为是正方形窗体。
3. 近剪辑平面距离：1 个单位。
4. 远剪辑平面距离：100 个单位。

```
   [C#]



using Microsoft.DirectX;


Direct3D.Device device = null;  // Create rendering device.


// For the projection matrix, you set up a perspective transform (which
//    transforms geometry from 3-D view space to 2-D viewport space, with
//    a perspective divide making objects smaller in the distance). To build
//    a perspective transform, you need the field of view (1/4 pi is common),
//    the aspect ratio, and the near and far clipping planes (which define
//    the distances at which geometry should no longer be rendered).


device.Transform.Projection = Matrix.PerspectiveFovLH(
                                  (float)Math.PI / 4, 1.0f, 1.0f, 100.0f );
```

第5节　设置 1 个视点矩阵

例子示范了怎么初始化视图变换矩阵，它转化世界坐标系为照相机或视图空间。
下面的C#代码，来自 LookAtLH方法的参数是由 Vector3 结构体的矢量组成，它建立了一个左手矩阵。View视图变换矩阵被设置等价于矩阵。

3 个输入矢量分别描绘了以下：

1. 着眼点：[0, 3, -5]。
2. 照相机注视目标：原点[0, 0, 0]。
3. 当前世界的向上的方向：通常[0, 1, 0]。

```
  [C#]



using Microsoft.DirectX.Direct3D;


Device device = null;   // Create rendering device.


// Set up the view matrix. A view matrix can be defined given an eye point,
//    a point to view, and a direction for which way is up. Here, you set
//    the eye five units back along the z-axis and up three units, view the
//    origin, and define "up" to be in the y-direction.


device.Transform.View = Microsoft.DirectX.Matrix.LookAtLH(
                            new Vector3(0.0f, 3.0f, -5.0f),
                            new Vector3(0.0f, 0.0f, 0.0f),
                            new Vector3(0.0f, 1.0f, 0.0f));
```

第6节　使用 1 个特效

例子示范了怎样使用一个从文件加载特效的技巧。
给你的渲染方法加特效代码，例如OnRender ()，位置在调用 Device.BeginScene方法和 Device.EndScene方法之间。

1. 创建或者获得访问一个高级着色器语言(HLSL)的文件(.fx)的权限。
2. 使用 Effect.FromFile，加载这个HLSL 文件。
3. 设置 Effect.Technique来决定特效的表现手法
4. 使用 Effect.Begin来获得特效经历的数目。
5. 创建一个循环来重述所有的特效经历。渲染一个发生在一个调用 Effect.BeginPass 方法和 Effect.EndPass方法之间循环的特效。循环自身嵌套在调用 Effect.Begin 方法和 Effect.End方法之间。
6. 调用 Effect.BeginPass，Device.DrawPrimitives，和 Effect.EndPass，渲染每一个在循环中的经历。

下面的C#代码，device被假定为正在渲染的 Device。

[C#]

```
public void OnRender ()
{
    .
    .
    .

    // Load the effect from file.
    Effect effect = Effect.FromFile(device, "shadercode.fx", null,
                                    ShaderFlags.None, null);
    // Set the technique.
    effect.Technique = "ShaderTechnique";

    // Note: Effect.Begin returns the number of
    // passes required to render the effect.
    int passes = effect.Begin(0);

    // Loop through all of the effect's passes.
    for (int i = 0; i < passes; i++)
    {
        // Set a shader constant
        effect.SetValue("WorldMatrix", worldMatrix);

        // Set state for the current effect pass.
        effect.BeginPass(i);

        // Render some primitives.
        device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
```

```
        // End the effect pass
        effect.EndPass();
    }

    // Must call Effect.End to signal the end of the technique.
    effect.End();

    .
    .
    .
}
```

第3章　使用网格、纹理和特效

第1节　检查 HDR 纹理支持

例子示范了怎么样检查一个设备来决定是否一个细节纹理格式被支持。
接下来的C#代码，device被假定为正在渲染的 Device设备，并且adapterFormat变量被假定是一个有效的 Format枚举成员。

```
[C#]



// check support for a Format.A16B16R16F render target
if (!Manager.CheckDeviceFormat(0, DeviceType.Hardware, adapterFormat,
                               Usage.RenderTarget,
ResourceType.CubeTexture,
                               Format.A16B16G16R16F))
    return true;
else
    return false;
```

第2节　清除 1 个网格

例子展示了怎么样在简化准备中清除一个网格。
下面的C#程序，Mesh.Clean方法清除网格，通过加入另一个与原来 2 个三角形扇共享的相同顶点来清除网格。顶点分裂导致 2 个新的三角形加入到多边形上，它们可以位于连接 2 个新顶点的线段的任意一边。

```
[C#]


GraphicsStream adj = null;


// Load the mesh from the specified file
Mesh pMesh = Mesh.FromFile("tiger.x", MeshFlags.Managed, device, out adj);


// perform simple cleaning operation on the mesh
Mesh pTempMesh = Mesh.Clean(CleanType.Optimization, pMesh, adj, adj);
pMesh.Dispose();
pMesh = pTempMesh;
```

第3节　克隆 1 个网格

例子展示了如何将 1 个带有法线、纹理坐标、色彩、重量等等特性的网格克隆（复制）添加到空间中。
下面的C#代码，在网格被从一个文件加载之后，Mesh.Clone方法被调用。调用pMesh.Clone从pMesh获得网格标识，并且为支持的 VertexFormats和顶点法线加标识。原始 pMesh 网格对象被重载以包含这些额外的值。注意 pMesh 在设置到最新的临时网格对象之前被释放。
这个例子中，device被假定为正在渲染的 Device。

```
[C#]


using Microsoft.DirectX.Direct3D;


pMesh = Mesh.FromFile("a_mesh_file.x", MeshFlags.Managed, device);
```

```
if ((pMesh.VertexFormat & VertexFormats.Normal) == 0)
{
    Mesh pTempMesh = pMesh.Clone(pMesh.Options.Value,
                                 pMesh.VertexFormat |
                                 VertexFormats.Normal, device);
    pTempMesh.ComputeNormals();
    pMesh.Dispose();
    pMesh = pTempMesh;
}
```

第4节　由 1 个网格计算 1 个跳球

例子展示了使用 Geometry.ComputeBoundingSphere方法，怎样产生一个围绕 3-D对象的 1 个简单跳动的球体 。一个跳球很可能使用了 3-D图形；例如，帮助测试一个 3-D对象是否和另一个相交。
接下来的C#代码，从网格对象的顶点数据建立了一个顶点buffer 缓冲器。这个新顶点buffer 缓冲器然后被锁定，以便于 Geometry 运算法则能被计算。ComputeBoundingSphere的输出是从中心到网格对象的最远端的半径。这个Mesh对象被假定是一个用 Mesh.FromFile方法有效加载的网格。

```
[C#]


float objectRadius = 0.0f;
Vector3 objectCenter = new Vector3();


using (VertexBuffer vb = Mesh.VertexBuffer)
{
    GraphicsStream vertexData = vb.Lock(0, 0, LockFlags.None);
    objectRadius = Geometry.ComputeBoundingSphere(vertexData,
                                                  mesh.NumberVertices,
                                                  mesh.VertexFormat,
                                                  out objectCenter);
```

```
    vb.Unlock();
}
```

第5节　复制 1 个网格

例子示范了怎么样拷贝一个纹理。
一个纹理被从文件中加载和复制，并且它的所有 AliceBlue 色被用透明的黑色替代。从右顶角计 10 像素边长的正方形区域被拷贝。
下面的C#代码，device被假定为正在渲染的 Device，并且FlagColorsInverted.bmp被假定在当前的路径下。

```
[C#]



using Microsoft.DirectX.Direct3D;

using System.Drawing;


Surface s = null;


PaletteEntry[] pal = new PaletteEntry[256];


// Set up the surface for our surface loader.

s = device.CreateOffscreenPlainSurface(32, 32, Format.A8R8G8B8,
        Pool.Default);


Surface k = TextureLoader.FromFile(device,

                        "FlagColorsInverted.bmp").GetSurfaceLevel(0);


SurfaceLoader.FromSurface(s, out pal, new Rectangle(0,0,9,9), k, out pal,

                new Rectangle(0,0,9,9), Filter.Box, Color.AliceBlue.ToArgb());
```

第6节　建立 1 个立方体纹理

例子示范了怎样创建一个立方体纹理。
在硬件被检查，以确保立方体纹理被支持之后，一个立方体纹理才被创建。这个新的立方体

纹理有长度为 16 像素的边缘，并且没有 Usage值。所有的立方纹理子集在每个表面被缩小至 1*1 像素，这个过程是自动产生的，并且这些子集被提供了对MIP映射立方纹理的硬件支持。
下面的C#代码，device被假定是正在渲染的 Device。开发者也应该检查以确保硬件支持 *X8R8G8B8* 格式。

```
[C#]



//Check cube texture requirements.

CubeTextureRequirements tr = new CubeTextureRequirements();


tr.Format = Format.A8R8G8B8;


TextureLoader.CheckCubeTextureRequirements(device, 0,
                                           Pool.Managed, out tr);


//Create texture.

CubeTexture cube = new CubeTexture( device, 16, 0, 0, Format.X8R8G8B8,
Pool.Managed );
```

第7节　建立 1 个网格对象

例子展示了怎样创建一个 Mesh网格对象。
下面的C#代码，device被假定是正在渲染的 Device。

```
[C#]



int numberVerts = 8;

short[] indices = {

    0,1,2, // Front Face

    1,3,2, // Front Face

    4,5,6, // Back Face

    6,5,7, // Back Face
```

```
    0,5,4, // Top Face
    0,2,5, // Top Face
    1,6,7, // Bottom Face
    1,7,3, // Bottom Face
    0,6,1, // Left Face
    4,6,0, // Left Face
    2,3,7, // Right Face
    5,2,7 // Right Face
};

Mesh mesh = new Mesh(numberVerts * 3, numberVerts, MeshFlags.Managed,
                     CustomVertex.PositionColored.Format, device);

using(VertexBuffer vb = mesh.VertexBuffer)
{
    GraphicsStream data = vb.Lock(0, 0, LockFlags.None);

    data.Write(new CustomVertex.PositionColored(-1.0f, 1.0f, 1.0f,
0x00ff00ff));
    data.Write(new CustomVertex.PositionColored(-1.0f, -1.0f, 1.0f,
0x00ff00ff));
    data.Write(new CustomVertex.PositionColored(1.0f, 1.0f, 1.0f,
0x00ff00ff));
    data.Write(new CustomVertex.PositionColored(1.0f, -1.0f, 1.0f,
0x00ff00ff));
    data.Write(new CustomVertex.PositionColored(-1.0f, 1.0f, -1.0f,
0x00ff00ff));
    data.Write(new CustomVertex.PositionColored(1.0f, 1.0f, -1.0f,
0x00ff00ff));
    data.Write(new CustomVertex.PositionColored(-1.0f, -1.0f, -1.0f,
0x00ff00ff));
    data.Write(new CustomVertex.PositionColored(1.0f, -1.0f, -1.0f,
0x00ff00ff));
```

```
    vb.Unlock();
}

using (IndexBuffer ib = mesh.IndexBuffer)
{
    ib.SetData(indices, 0, LockFlags.None);
}
```

第8节　保存 1 个屏幕快照

例子示范了怎么保存一个屏幕截图到一个文件。
后台 buffer 缓冲器被重新得到，并且作为一个位图图片被保存。
下面的C#代码，device被假定为正在渲染的 Device。在渲染完成后，代码被调用。

```
  [C#]

using Microsoft.DirectX.Direct3D;

Surface backbuffer = device.GetBackBuffer(0, 0, BackBufferType.Mono);
SurfaceLoader.Save("Screenshot.bmp", ImageFileFormat.Bmp, backbuffer);
backbuffer.Dispose();
```

第9节　简化 1 个网格

例子示范了怎样简化一个网格。
通过 25 个顶点，网格被简化。
下面的C#代码，网格被假定是一个已经合理加载和清除的 Mesh实例，adjacency是一个包含网格邻接数据的 GraphicsStream实例，device是一个正在渲染的 Device。

```
[C#]

SimplificationMesh simplifiedMesh = new SimplificationMesh(mesh,
adjacency);
```

```
simplifiedMesh.ReduceVertices(mesh.NumberVertices - 25);


mesh.Dispose();
mesh = simplifiedMesh.Clone(simplifiedMesh.Options.Value,
                            simplifiedMesh.VertexFormat, device);
simplifiedMesh.Dispose();
```

相关主题

Clean a Mesh

第4章　使用顶点和索引缓存

第1节　建立 1 个顶点声明

例子示范了怎样创建一个顶点声明。

[C#]

正如接下来的C#代码，首先声明一个 VertexElement数组来控制定点着色器声明。这个声明的数组必须以 VertexElement.VertexDeclarationEnd作为最后一个元素来结尾（这个顶点元素数组的尺寸将比实际顶点元素个数多一个）。

使用前面建立的 Device和 VertexElement数组来创建 1 个 VertexDeclaration实例。

每个元素的偏移参数是来自声明起点的元素积累的偏移量。例如，第二个元素是偏移了 12 bytes，因为 3 个 floats 型数每个占用 4 bytes，即(3 * sizeof(float) = 12)。

```
// Create the vertex element array.
VertexElement[] elements = new VertexElement[]
{
    new VertexElement(0, 0, DeclarationType.Float3,
                            DeclarationMethod.Default,
                            DeclarationUsage.Position, 0),

    new VertexElement(0, 12, DeclarationType.Float3,
                             DeclarationMethod.Default,
                             DeclarationUsage.Normal, 0),
```

```
    new VertexElement(0, 24, DeclarationType.Float2,
                              DeclarationMethod.Default,
                              DeclarationUsage.TextureCoordinate, 0),


    VertexElement.VertexDeclarationEnd
};


// Use the vertex element array to create a vertex declaration.
VertexDeclaration decl = new VertexDeclaration(device, elements);
```

第2节　通过图形流读写顶点缓存和索引缓存数据

例子示范了怎样使用 GraphicsStream对象来填充，以及从一个 VertexBuffer对象和IndexBuffer对象重新得到数据。
在接下来的C#代码，一个 VertexBuffer对象使用一个可变形顶点格式(FVF)类型来被创建，这个FVF类型是由**PositionNormalTexVertex** 结构体定义的。使用它的偏移量和bytes 尺寸锁定顶点buffer 缓冲器的数据。一个来源于流的 GraphicsStream图形流对象被返回，并被用于一个相似的fashon中。范例代码中，**Write** 方法通常从一个顶点数据的数组拷贝数据到流。
范例的第二部分代码展示了使用非安全数据访问 GraphicsStream的使用。内部数据指针GraphicsStream.InternalData属性，返回一个指向顶点buffer 缓冲器数据的空指针。范例代码中，数据被抛入到一个**PositionNormalTexVertex** 结构体的数组，这个结构体让数据有更好的可读性。

```
[C#]
using System;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;


public struct PositionNormalTexVertex
{
        public Vector3 Position;
        public Vector3 Normal;
        public   float Tu0, Tv0;
```

```
        public static readonly VertexFormats FVF = VertexFormats.Position |
VertexFormats.Texture1;

}

public class Example

{

        public unsafe void   GraphicsStreamReadWrite()

        {

                //Create a vertex buffer in the managed pool

                VertexBuffer vb = new
VertexBuffer(typeof(PositionNormalTexVertex), 100, device, Usage.None,
PositionNormalTexVertex.FVF, Pool.Managed);



                //First, fill an array of PositionNormalTexVertex elements with
data.

                PositionNormalTexVertex[] vertices = new
PositionNormalTexVertex[50];

                for(int i=0; i<50; i++)

                {

                        //fill the vertices with some data...

                        vertices[i].Position = new Vector3(3f,4f,5f);

                }



                //The size of the verticies are 32-bytes each (float3 (12) +
float3 (12) + float(4) + float(4))

                //To lock 50 verticies, the size of the lock would be 1600 (32 *
50)

                GraphicsStream vbData =   vb.Lock(0,1600,
LockFlags.None);



                //copy the vertex data into the vertex buffer

                vbData.Write(vertices);



                //Unlock the VB
```

```
                vb.Unlock();



                //This time, lock the entire VertexBuffer
                vbData =   vb.Lock(0, 3200, LockFlags.None);


                //Cast the InternalDataPointer (a void pointer) to an array of
verticies
                PositionNormalTexVertex* vbArray =
(PositionNormalTexVertex*) vbData.InternalDataPointer;


                for(int i=0; i<100; i++)
                {
                        //perform some operations on the data
                        vbArray[i].Tu0 = i;
                        vbArray[i].Tv0 = vbArray[i].Tu0 * 2;


                        Console.WriteLine(vbArray[i].Tv0.ToString());
                }


                //Unlock the buffer
                vb.Unlock();
                vb.Dispose();
        }
}
```

相关主题

Read and Write VertexBuffer Data With Arrays

第3节　通过数组读写顶点缓存数据

例子示范了怎么使用数组来填充，以及从一个 VertexBuffer对象和 IndexBuffer对象重新得到数据。

在接下来的C#代码，一个 VertexBuffer对象使用一个可变形顶点格式(FVF)类型来被创建，这个FVF类型是由**PositionNormalTexVertex** 结构体定义的。范例的第一部分代码展示了 Lock的使用：用来锁住一个任意数目的顶点和用数据填充它们。范例的第二部分代码展示了怎样锁定一个完整的 VertexBuffer，以供读取和从数组中重新得到数据。一个相似的过程能被用来从几乎所有的Microsoft® Direct3D®资源类型写入和读取。
对于填充和从Direct3D 资源读取数据，使用 Arrays不是最有效的方法；它需要额外的内存和时钟周期来拷贝非托管资源到托管数组。可是，它获得了非安全访问的可读性代码。

```
[C#]

using System;

using Microsoft.DirectX;

using Microsoft.DirectX.Direct3D;



public struct PositionNormalTexVertex

{

        public Vector3 Position;

        public Vector3 Normal;

        public float Tu0, Tv0;

        public static readonly VertexFormats FVF = VertexFormats.Position |
VertexFormats.Texture1;

}



public class Example

{

        public void ArrayBasedReadWrite()

        {

                //Create a vertex buffer in the managed pool

                VertexBuffer vb = new
VertexBuffer(typeof(PositionNormalTexVertex), 100, device, Usage.None,
PositionNormalTex1Vertex.FVF, Pool.Managed);



                //Fill an array of the appropriate type with the VB data using
Lock()
```

```
                PositionNormalTexVertex[] vbData =
(PositionNormalTexVertex[]) vb.Lock(0, typeof(PositionNormalTexVertex),
LockFlags.None, 50);

                for(int i=0; i<50; i++)

                {

                        //set your vertices to something...

                        vbData[i].Position = new Vector3(2f,2f,2f);

                        vbData[i].Normal = new Vector3(1f,0f,0f);

                        vbData[i].Tu0 = i;

                        vbData[i].Tv0 = i;

                }

                //Unlock the vb before you can use it elsewhere

                vb.Unlock();



                //This lock overload simply locks the entire VB -- setting
ReadOnly can improve perf when reading a vertexbuffer

                vbData = (PositionNormalTexVertex[]) vb.Lock(0,
LockFlags.ReadOnly);

                for(int i=0; i<100; i++)

                {

                        //read some vertex data

                        Console.WriteLine("Vertex " + i + "Tu: " +
vbData[i].Tu0 + " , Tv: " + vbData[i].Tv0);

                }



                //Unlock the buffer

                vb.Unlock();




                vb.Dispose();

        }

}
```

相关主题

Read and Write VertexBuffer and IndexBuffer Data With GraphicsStreams

第5章　使用声音特效

第1节　向 1 个 SecondaryBuffer 对象加入特效

下面的C#代码示范了怎么样为一个 SecondaryBuffer对象，加上一些特效对象。

```
   [C#]



//Create and setup the sound device.
Device dev = new Device();
dev.SetCooperativeLevel(this,CooperativeLevel.Normal);

//Create and setup the buffer description.
BufferDescription buffer_desc = new BufferDescription();
buffer_desc.ControlEffects = true; //this has to be true to use effects.
buffer_desc.GlobalFocus = true; //play sound even if application loses focus.

//Create and setup the buffer for playing the sound.
SecondaryBuffer buffer = new SecondaryBuffer(
          @"C:\WINDOWS\Media\ding.wav",
          buffer_desc,
          dev);

//Create an array of effects descriptions,
//set the effect objects to echo and chorus and
//set it in the buffer.
EffectDescription[] effects = new EffectDescription[2];
```

```
effects[0].GuidEffectClass = DSoundHelper.StandardEchoGuid;

effects[1].GuidEffectClass = DSoundHelper.StandardChorusGuid;

buffer.SetEffects(effects);


//Play Buffer.

buffer.Play(0,BufferPlayFlags.Default);
```

相关主题

Use Effect Parameters

第2节　使用特效参数

这个C#代码示范了怎么样使用来自于 SecondaryBuffer对象的一个效果对象的参数。
假设代码片断中的buffer缓冲器对象是一个 SecondaryBuffer次要缓冲器，它的代码来自于 Add Effects to a SecondaryBuffer Object，这篇向次要缓冲器加入特效的文章。

```
[C#]




//Retrieve the effects object and

//the effect param sturctures and edit parameters.

EchoEffect echo = (EchoEffect)buffer.GetEffects(0);

EffectsEcho echo_params = echo.AllParameters;


echo_params.LeftDelay = 250.0f;

echo_params.RightDelay = 100.0f;

echo_params.Feedback = 85.0f;

echo_params.PanDelay = 1;

echo_params.WetDryMix = 50.0f;


ChorusEffect chorus = (ChorusEffect)buffer.GetEffects(1);

EffectsChorus chorus_params = chorus.AllParameters;
```

```
chorus_params.Delay = 15.0f;
chorus_params.Depth = ChorusEffect.DepthMax;
chorus_params.Phase = ChorusEffect.PhaseNegative90;
chorus_params.Waveform = ChorusEffect.WaveSin;
chorus_params.WetDryMix = 50.0f;

//Set the new parameters and play the buffer.
echo.AllParameters = echo_params;
chorus.AllParameters = chorus_params;
buffer.Play(0,BufferPlayFlags.Default);
```

相关主题

Add Effects to a SecondaryBuffer Object