Direct3D 起步

本节提供对 Microsoft® Direct3D®应用程序编程接口（API）中三维图形功能的介绍。可以在这里找到有关*图形流水线*的概述，以及可以快速运行，帮助开发者了解 Direct3D 基本功能的教程。

Direct3D体系结构

三维坐标系与几何学

Direct3D对象

设备

资源

状态

顶点声明

顶点格式

几何体

渲染

Direct3D 体系结构

本节包含了有关 Microsoft® Direct3D®部件和其它 Microsoft DirectX®部件、操作系统、及系统硬件之间关系的信息，讨论了以下主题。

Direct3D体系结构概述

硬件抽象层

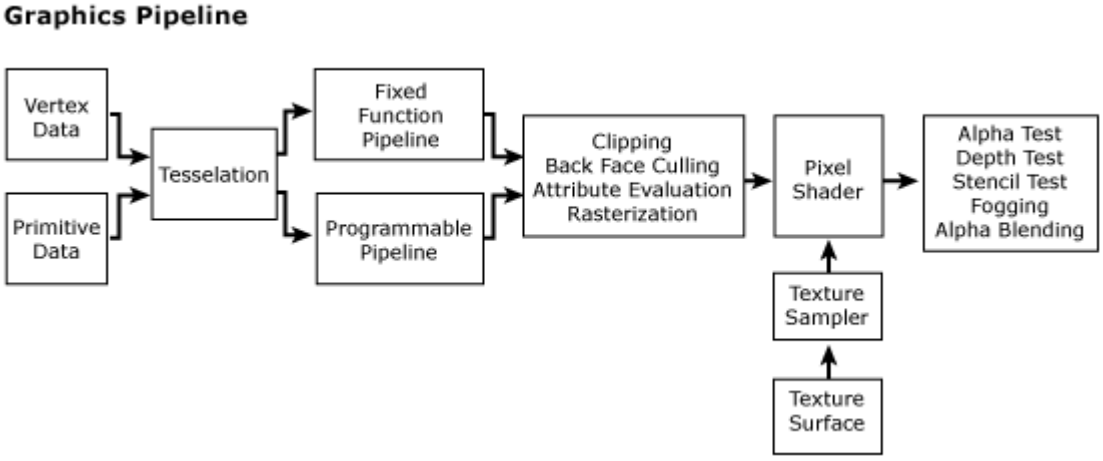
系统集成

可编程顶点着色器体系结构

可编程像素着色器体系结构

Direct3D 体系结构概述

这是一幅图形流水线的图示。下面介绍了每一块的功能，以及在哪里可以找到更多信息的链接。

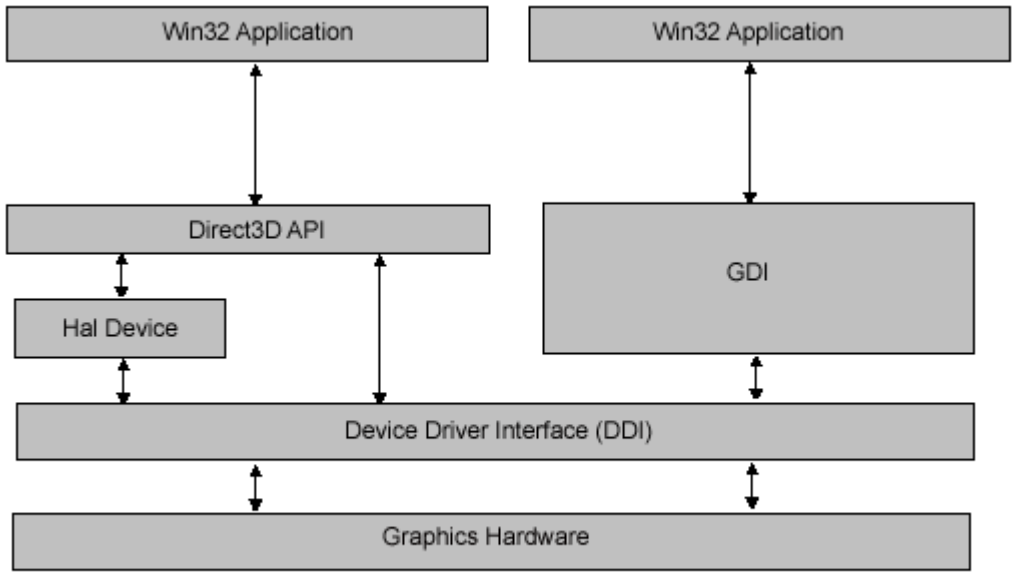


有关Microsoft® Direct3D®可编程部分的体系结构的更多信息，请参阅可编程顶点着色器体系结构和可编程像素着色器体系结构。

硬件抽象层

Microsoft® Direct3D®通过硬件抽象层（HAL）提供设备无关性。HAL 是一个设备相关的接口，由设备制造商提供，Direct3D 使用 HAL 与显示硬件协同工作。应用程序从不直接与 HAL 打交道。相反，通过 HAL 提供的基础，Direct3D 暴露了一组统一的接口和方法，应用程序用这些接口和方法绘制/显示图形。在 Microsoft Windows® XP、Microsoft Windows NT®和 Windows 2000 下，设备制造商用 32 位代码实现 HAL。而在 Windows 98 和 Windows Millennium Edition (Windows Me) 下，则混合使用 16 位和 32 位代码。HAL 可以是显示驱动程序的一部分，或者是一个单独的动态

链接库（DLL），该 DLL 通过驱动程序开发人员定义的私有接口与显示驱动程序进行通信。
Direct3D HAL 由芯片制造商、板卡制造商或原始设备制造商（OEM）实现。HAL 仅实现与设备相关的代码并且不做任何模拟。如果硬件不能完成某项功能，则 HAL 不将其声明为硬件的能力。另外，HAL 不检验参数，Direct3D 在调用 HAL 之前执行这项操作。
在 Microsoft DirectX® 9.0 中， HAL 可以有三种不同的顶点处理模式：软件顶点处理、硬件顶点处理、以及在同一设备上的混合顶点处理。纯设备模式是 HAL 设备的一个变体。纯设备类型只支持硬件顶点处理，且只允许应用程序查询设备状态中很小的一个子集。另外，纯设备仅在具有某一最低能力级的适配器上可用。
系统集成
下图显示了 Microsoft® Direct3D®、Microsoft Windows®图形设备接口（GDI）、硬件抽象层（HAL）及硬件之间的关系。

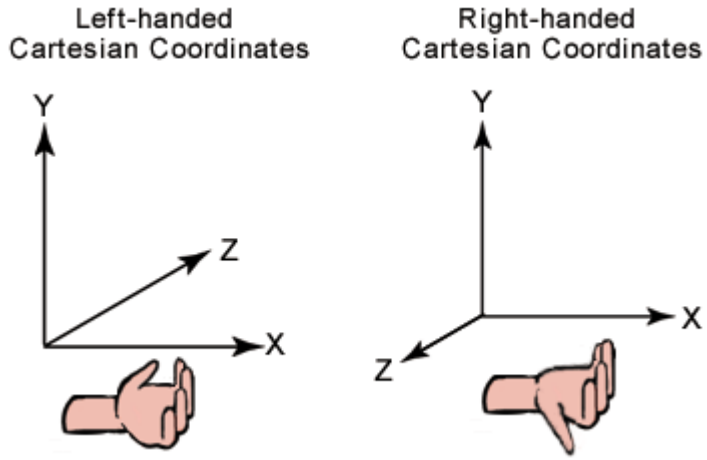


如上图所示，Direct3D 应用程序位于 GDI 应用程序旁边，它们都可以通过图形卡的设备驱动程序访问图形硬件。与 GDI 不同的是，当选择了 HAL 设备时，Direct3D 可以利用硬件特性。基于图形卡支持的特性集，HAL 设备提供了硬件加速。为了在运行时检查设备是否能执行某项操作，Direct3D 提供了相应的方法。
有关Direct3D所支持设备的更多信息，请参阅设备类型。
可编程顶点着色器体系结构
有关顶点着色器寄存器的更多信息，请参阅Registers - vs 1 1。
有关顶点着色器参考章节的更多信息，请参阅Vertex Shader 1_1。
可编程像素着色器体系结构
有关像素着色器寄存器的更多信息，请参阅Registers - ps 1 X。
有关像素着色器参考章节的更多信息，请参阅Pixel Shader 1_X。

三维坐标系与几何学

编写 Microsoft® Direct3D®应用程序需要熟悉三维几何学原理。本节介绍创建三维场景所需的最重要的几何概念。本节涉及到以下主题。
三维坐标系

三维图元
表面和顶点法向
三角形光栅化法则
矩形
三角形插值
向量、顶点和四元数
这些主题给读者提供了一个对Direct3D应用程序所涉及到的基本概念的高层描述。更多有关这些主题的信息，请参阅更多的信息。
三维坐标系
通常三维图形应用程序使用两种笛卡尔坐标系：左手系和右手系。在这两种坐标系中，正 x 轴指向右面，正 y 轴指向上面。通过沿正 x 轴方向到正 y 轴方向握拳，大姆指的指向就是相应坐标系统的正 z 轴的指向。下图显示了这两种坐标系统。

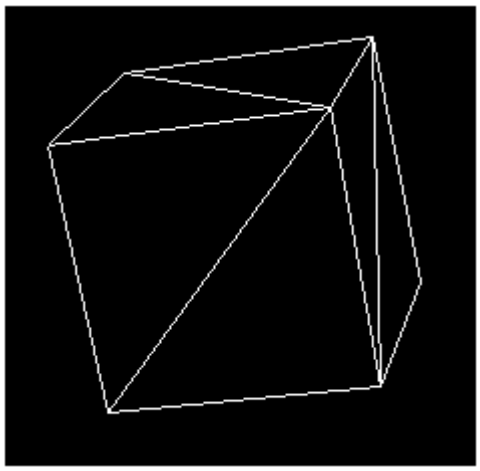


Microsoft® Direct3D®使用左手坐标系。如果正在移植基于右手坐标系的应用程序，必须将传给 Direct3D 的数据做两点改变。
颠倒三角形顶点的顺序，这样系统会从正面以顺时针的方向遍历它们。换句话说，如果顶点是 v0，v1，v2，那么以 v0，v2，v1 的顺序传给 Direct3D。
用观察矩阵对世界空间中的z值取反。要做到这一点，将表示观察矩阵的D3DMATRIX结构的_31、_32、_33 和_34 成员的符号取反。
要得到等同于右手系的效果，可以使用D3DXMatrixPerspectiveRH和D3DXMatrixOrthoRH函数定义投影矩阵。但是，要小心使用D3DXMatrixLookAtRH函数，并相应地颠倒背面剔除的顺序及放置立方体贴图。
虽然左手坐标系和右手坐标系是最为常用的系统，但在三维软件中还使用许多其它坐标系。例如，对三维建模应用程序而言，使用 y 轴指向或背向观察者的坐标系统并非罕见。在这种情况下，任意轴（x，y 或 z）的正半轴指向观察者的被定义为右手系。任意轴（x，y 或 z）的正半轴背向观察者的被定义为左手系。如果正在移植一个基于左手系进行建模的应用程序，z 轴向上，那么除了前面的步骤外，还必须旋转所有的顶点数据（译注：如果原来的坐标系为正 x 轴向里，正 y 轴向左，正 z 轴向上，那么传给 Direct3D 的顶点的 x 值对应原来的 y 值，y 值对应原来的 z 值，z 值对应原来的 x 值，亦即旋转顶点数据）。
对三维坐标系统中定义的三维物体执行的最基本操作是变换、旋转和缩放。可以合并这些基本变换以创建一个新的变换矩阵。细节请参阅三维变换。
即使合并相同的变换操作，不同的合并顺序得到的结果是不可交换的——矩阵相乘的顺序很重要。

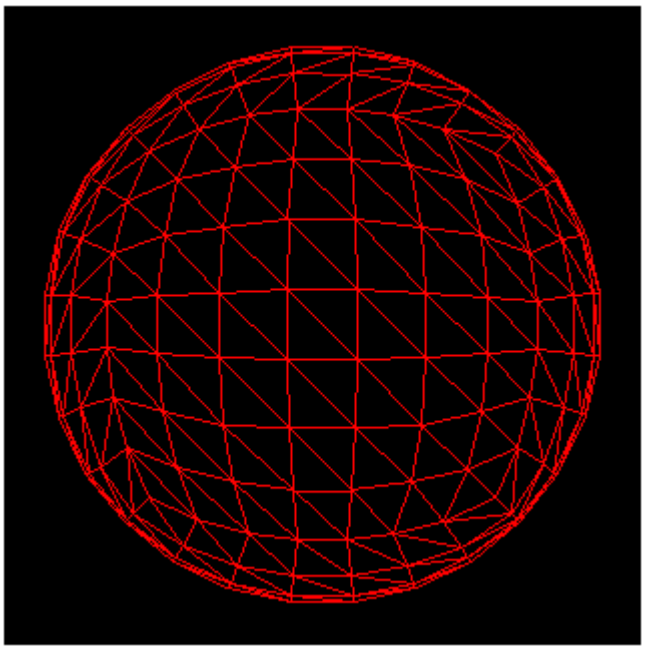
三维图元

三维图元是组成单个三维实体的顶点集合。三维坐标系统中最简单的图元是点的集合，称为点表。通常三维图元是多边形。一个多边形是由至少三个顶点描绘的三维形体。最简单的多边形是三角形。Microsoft® Direct3D®使用三角形组成大多数多边形，因为三角形的三个顶点一定是共面的。应用程序可以用三角形组合成大而复杂的多边形及网格（mesh）。

下图显示了一个立方体。立方体的每个面由两个三角形组成。整个三角形的集合构成了一个立方体图元。可以将纹理和材质应用于图元的表面使它们看起来像是实心的。

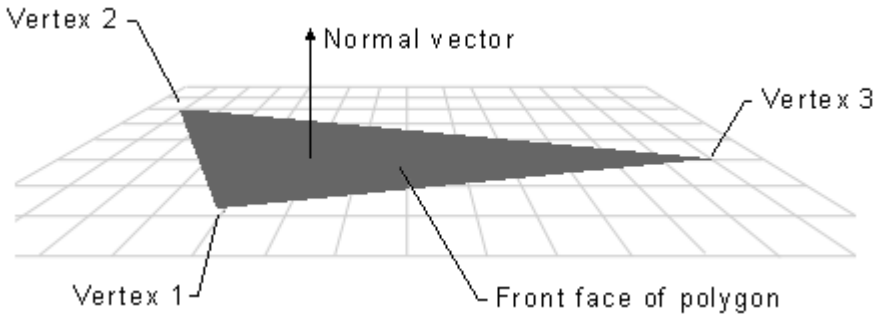


可以使用三角形创建具有光滑曲面的图元。下图显示了如何用三角形模拟一个球体。应用了材质后，渲染得到的球体看起来是弯曲的。如果使用高洛德着色，结果更是如此。更多信息请参阅高洛德着色。



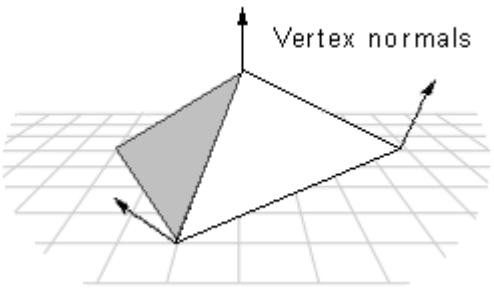
表面和顶点法向量

网格中的每个面有一个垂直的法向量。该向量的方向由定义顶点的顺序及坐标系统是左手系还是右手系决定。表面法向量从表面上指向正向面那一侧，如果把表面水平放置，正向面朝上，背向面朝下，那么表面法向量为垂直于表面从下方指向上方。在 Microsoft® Direct3D®中，只有面的正向是可视的。一个正向面是顶点按照顺时针顺序定义的面。



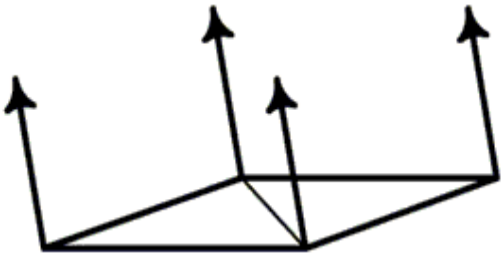
任何不是正向面的面都是背向面。由于Direct3D不总是渲染背向面，因此背向面要被剔除。如果想要渲染背向面的话，可以改变剔除模式。更多信息请参阅剔除状态。

Direct3D 在计算高洛德着色、光照和纹理效果时使用顶点法向。



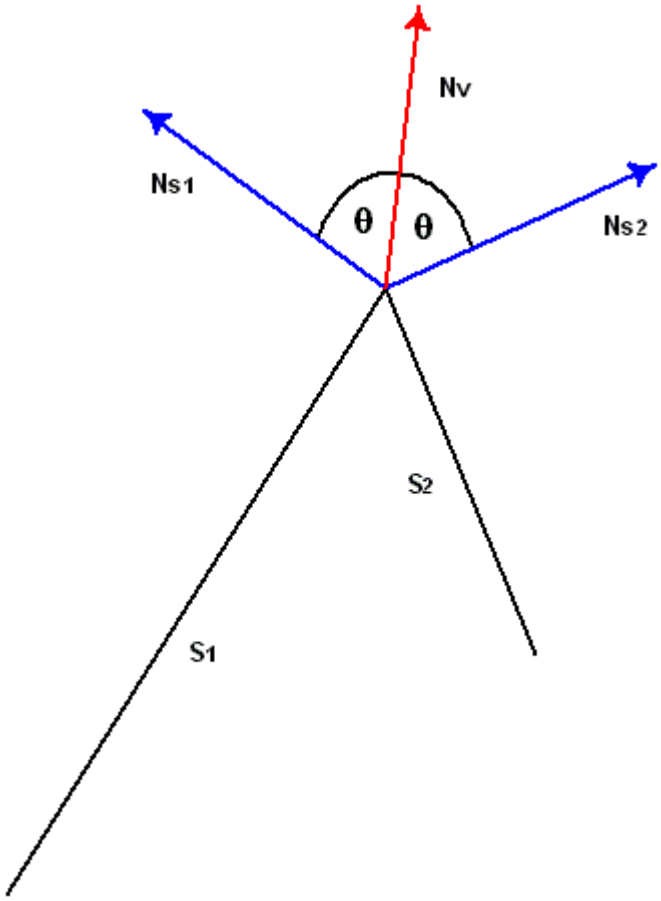
Direct3D 使用顶点法向计算光源和表面间的夹角，对多边形进行高洛德着色。Direct3D 计算每个顶点的颜色和亮度值，并对图元表面所覆盖的所有像素点进行插值。Direct3D 使用夹角计算光强度，夹角越大，表面得到的光照就越少。

如果正在创建的物体是平直的，可将顶点法向设为与表面垂直，如下图所示。该图定义了一个由两个三角形组成的平直表面。

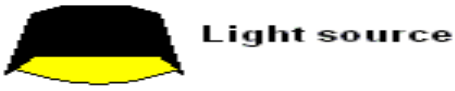


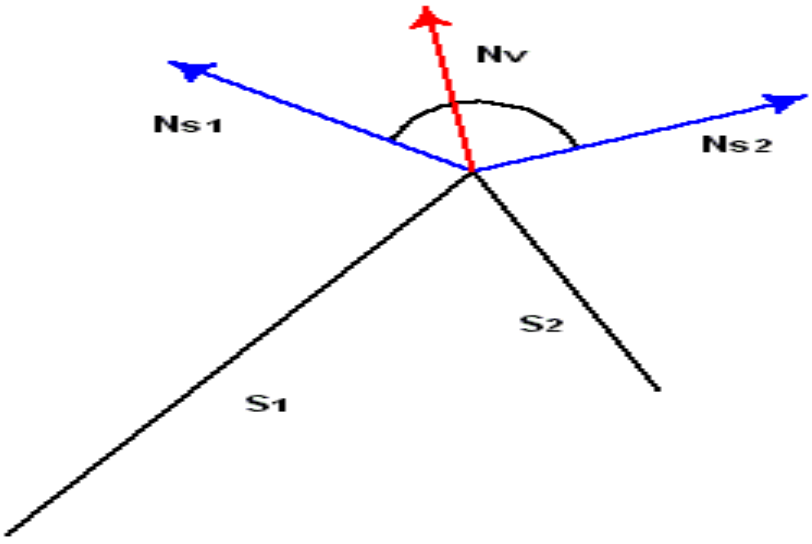
但是，更可能的情况是物体由三角形带（triangle strips）组成且三角形不共面。要对整个三角形带的三角形平滑着色的一个简单方法是首先计算与顶点相关联的每个多边形表面的表面法向量。可以这样计算顶点法向，使顶点法向与顶点所属的每个表面的法向的夹角相等。但是，对复杂图元来说这种方法可能不够有效。

这种方法如下图所示。图中有两个表面，S1 与 S2，它们的邻边在上方。S1 与 S2 的法向量用蓝色显示。顶点的法向量用红色显示。顶点法向量与 S1 表面法向的夹角和顶点法向量与 S2 表面法向的夹角相同。当对这两个表面进行光照计算和高洛德着色时，得到结果是中间的边被平滑着色，看起来像是弧形的（而不是有棱角的）。

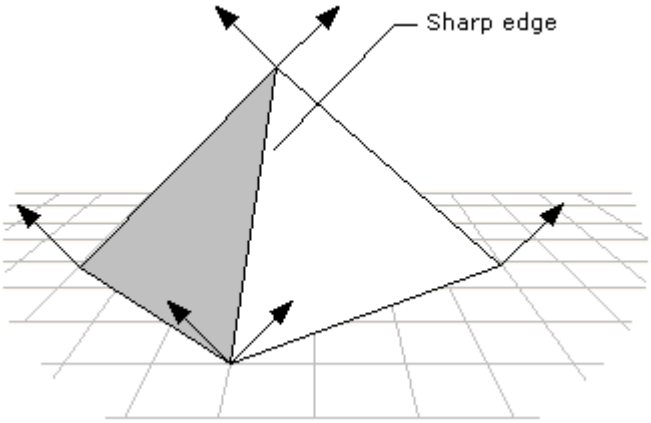


如果顶点法向偏向与它相关联的某个面，那么会导致那个面上的点光强度的增加或减少。下图显示了一个例子。这些面的邻边依然朝上。顶点法向倾向 S1，与顶点法向与表面法向有相同的夹角相比，这使顶点法向与光源间的夹角变小。





可以用高洛德着色在三维场景中显示一些有清晰边缘的物体。要达到这个目的，只要在需要产生清晰边缘的表面交线处，把表面法向复制给交线处顶点的法向，如下图所示。



如果使用 DrawPrimitive 方法渲染场景，要将有锋利边缘的物体定义为三角形表，而非三角形带。当将物体定义为三角形带时，Direct3D 会将它作为由多个三角形组成的单个多边形处理。高洛德着色被同时应用于多边形每个表面的内部和表面之间。结果产生表面之间平滑着色的物体。因为三角形表由一系列不相连的三角形面组成，所以 Direct3D 对多边形每个面的内部使用高洛德着色。但是，没有在表面之间应用高洛德着色。如果三角形表的两个或更多的三角形是相邻的，那么在它们之间看起来会有一条锋利边缘。

另一种可选的方法是在渲染具有锋利边缘的物体时改变到平面着色模式。这在计算上是最有效的方法，但它可能导致场景中的物体不如用高洛德着色渲染的物体真实。

三角形光栅化法则

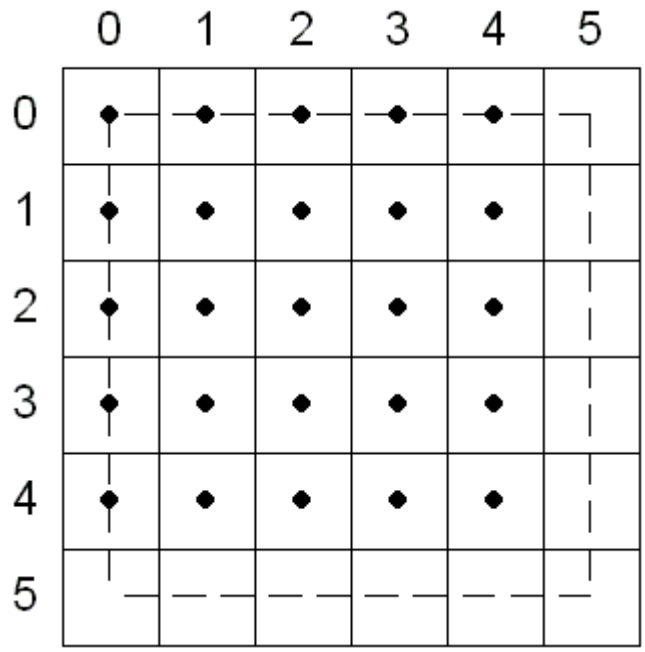
顶点指定的点经常不能精确地对应到屏幕上的像素。此时，Microsoft® Direct3D®使用三角形光栅化法则决定对于给定三角形使用哪个像素。

三角形光栅化法则
点、线光栅化法则
点精灵光栅化法则

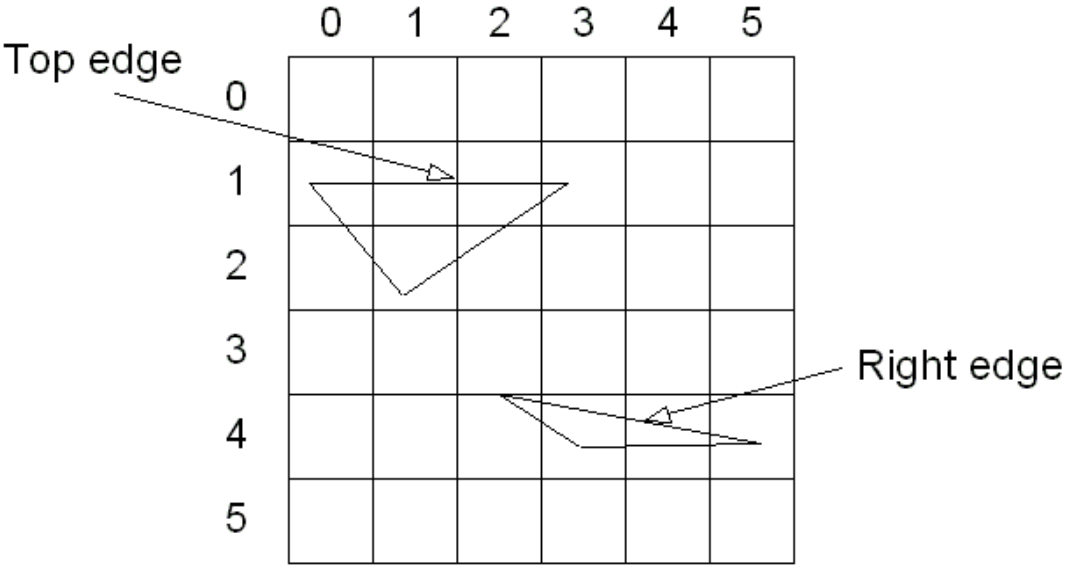
三角形光栅化法则

Direct3D 在填充几何图形时使用左上填充约定(top-left filling convention)。这与 Microsoft Windows®的图形设备接口（GUI）和 OpenGL 中的矩形使用的约定相同。Direct3D 中，像素的中心是决定点。如果中心在三角形内，那么该像素就是三角形的一部分。像素中心用整数坐标表示。这里描述的 Direct3D 使用的三角形光栅化法则不一定适用于所有可用的硬件。测试可以发现这些法则的实现间的细微变化。

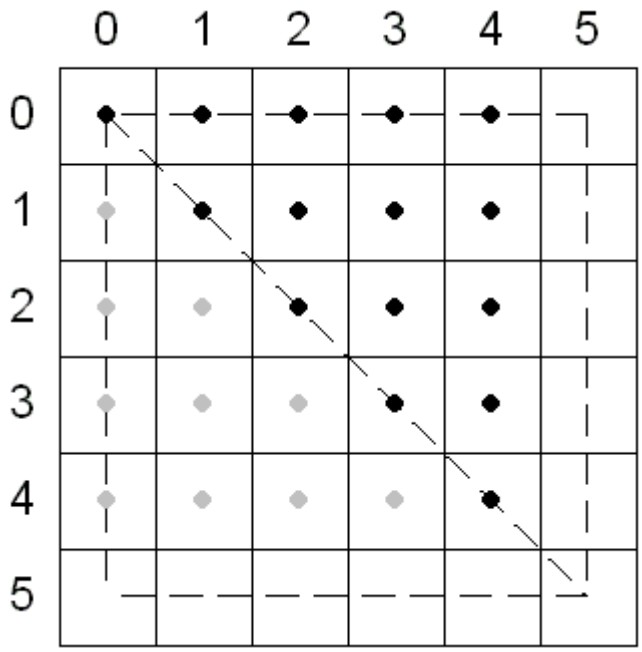
下图显示了一个左上角为（0，0），右下角为（5，5）的矩形。正如大家想象的那样，此矩形填充 25 个像素。矩形的宽度由 right 减 left 定义。高度由 bottom 减 top 定义。



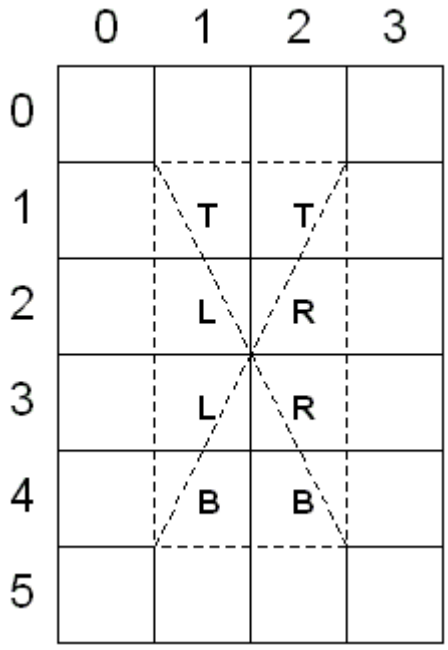
在左上填充约定中，*上*表示水平 span 在垂直方向上的位置，*左*表示 span 中的像素在水平方向上的位置。一条边除非是水平的，否则不可能是顶边——一般来说，大多数三角形只有左边或右边。



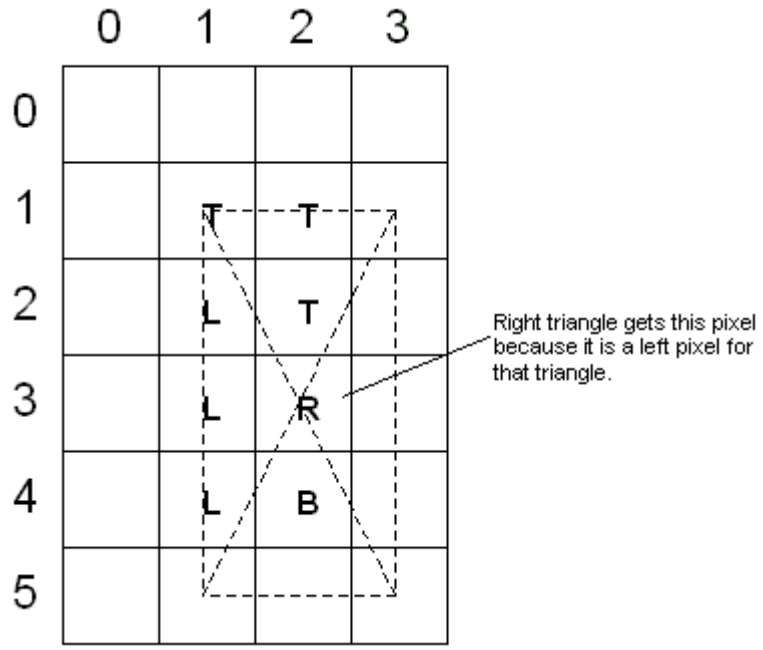
左上填充约定确定当一个三角形穿过像素的中心时 Direct3D 采取的动作。下图显示了两个三角形，一个在（0，0），（5，0）和（5，5），另一个在（0，5），（0，0）和（5，5）。在这种情况下第一个三角形得到 15 个像素（显示为黑色），而第二个得到 10 个像素（显示为灰色），因为公用边是第一个三角形的左边。



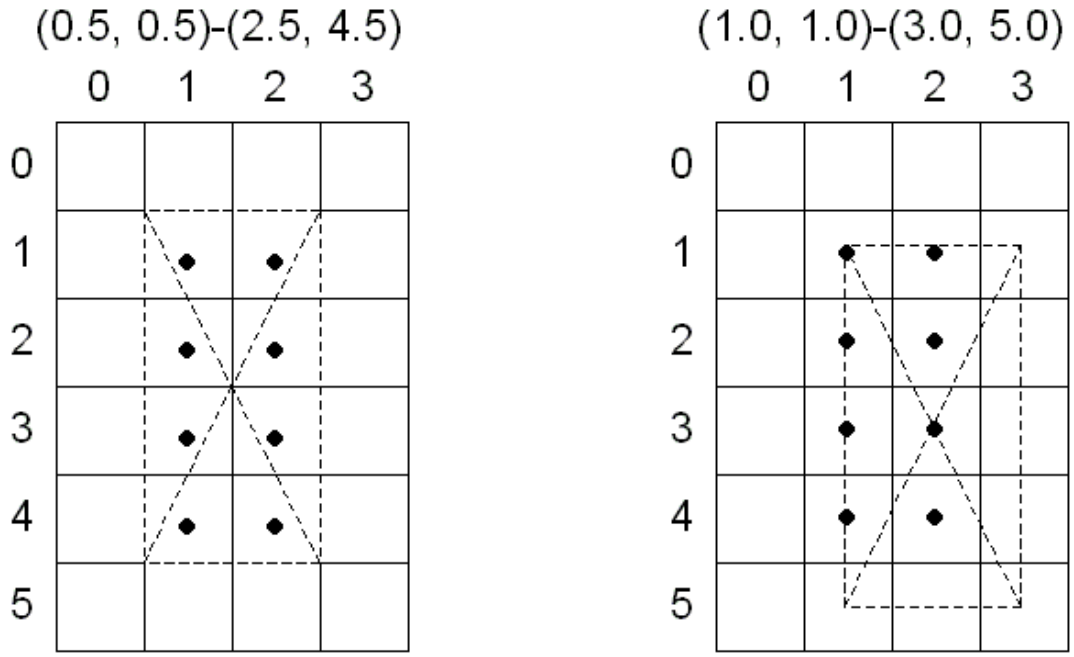
如果应用程序定义一个左上角为（0.5，0.5），右下角为（2.5，4.5）的矩形，那么这个矩形的中心在（1.5，2.5）。当 Direct3D 光栅化器 tessellate 这个矩形时，每个像素的中心都毫无异义地分别位于四个三角形中，此时就不需要左上填充约定。下图显示了这种情况。矩形内的像素根据在 Direct3D 中被哪个三角形包含做了相应的标注。



如果将上例中的矩形移动，使之左上角为（1.0，1.0），右下角为（3.0，5.0），中心为（2.0，3.0），那么 Direct3D 使用左上角填充约定。这个矩形中大多数的像素跨越两个或更多的三角形的边界，如下图所示。



这两个矩形会影响到相同的像素。



点、线光栅化法则

点和点精灵一样，都被渲染为与屏幕边缘对齐的四边形，因此它们使用与多边形同样的渲染法则。非抗锯齿线段的渲染法则与 GDI 使用的法则完全相同。

更多有关抗锯齿线段的渲染，请参阅ID3DXLine。

点精灵光栅化法则

对点精灵和patch图元的渲染，就好像先把图元tessellate成三角形，然后将得到的三角形进行光栅化。更多信息，请参阅点精灵。

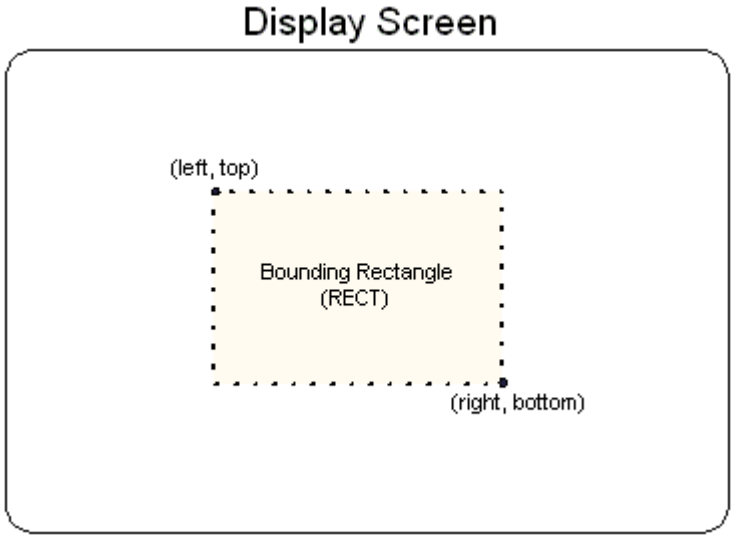
矩形

贯穿 Microsoft® Direct3D®和 Microsoft Windows®编程，都是用术语包围矩形来讨论屏幕上的物体。由于包围矩形的边总是与屏幕的边平行，因此矩形可以用两个点描述，左上角和右下角。当在屏幕上进行位块传输（Blit = Bit block transfer）或命中检测时，大多数应用程序使用

RECT 结构保存包围矩形的信息。
C++中，RECT 结构有如下定义。

```
typedef struct tagRECT {
    LONG    left;    // 这是左上角的 x 坐标。
    LONG    top;     // 这是左上角的 y 坐标。
    LONG    right;   // 这是右下角的 x 坐标。
    LONG    bottom;  // 这是右下角的 y 坐标。
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;
```

在上例中，left 和 top 成员是包围矩形左上角的 x-和 y-坐标。类似地，right 和 bottom 成员组成右下角的坐标。下图直观地显示了这些值。



为了效率、一致性及易用性， Direct3D 所有的 presentation 函数都使用矩形。
三角形插值对象（interpolants）
在渲染时，流水线会贯穿每个三角形的表面进行顶点数据插值。有五种可能的数据类型可以进行插值。顶点数据可以是各种类型的数据，包括（但不限于）：漫反射色、镜面反射色、漫反射阿尔法（三角形透明度）、镜面反射阿尔法、雾因子（固定功能流水线从镜面反射的阿尔法分量中取得，可编程顶点流水线则从雾寄存器中取得）。顶点数据通过顶点声明定义。
对一些顶点数据的插值取决于当前的着色模式，如下表所示。

着色模式	描述
平面	在平面着色模式下只对雾因子进行插值。对所有其它的插值对象，整个面都使用三角形第一个顶点的颜色。
高洛德	在所有三个顶点间进行线性插值。

根据不同的颜色模型，对漫反射色和镜面反射色的处理是不同的。在 RGB 颜色模型中，系统在插值时使用红、绿和蓝颜色分量。
颜色的阿尔法成员作为单独的插值对象对待，因为设备驱动程序可以以两种不同的方法实现透明：使用纹理混合或使用点画法（stippling）。
可以用D3DCAPS9结构的ShadeCaps成员确定设备驱动程序支持何种插值。
向量、顶点和四元数
贯穿 Microsoft® Direct3D®，顶点用于描述位置和方向。图元中的每个顶点由指定其位置的向量、颜色、纹理坐标和指定其方向的法向量描述。
四元数给三元素向量的[*x, y, z*]值增加了第四个元素。用于三维旋转的方法，除了典型的矩阵

以外，四元数是另一种选择。四元数表示三维空间中的一根轴及围绕该轴的一个旋转。例如，一个四元数可能表示轴(1,1,2)和 1 度的旋转。四元数包含了有价值的信息，但它们真正的威力源自可对它们执行的两种操作：合成和插值。
对四元数进行插值与合成它们类似。两个四元数的合成如下表示：

$$Q = q_1 \circ q_2$$

将两个四元数的合成应用于几何体意味着“把几何体绕 axis2 轴旋转 rotation2 角度，然后绕 axis1 轴旋转 rotation1 角度”。在这种情况下，Q 表示绕单根轴的旋转，该旋转是先后将 q2 和 q1 应用于几何体的结果。
使用四元数，应用程序可以计算出一条从一根轴和一个方向到另一根轴和另一个方向的平滑、合理的路径。因此，在 q1 和 q2 间插值提供了一个从一个方向变化到另一个方向的简单方法。
当同时使用合成与插值时，四元数提供了一个看似复杂而实际简单的操作几何体的方法。例如，设想我们希望把一个几何体旋转到某个给定方向。我们已经知道希望将它绕 axis2 轴旋转 r2 度，然后绕 axis1 轴旋转 r1 度，但是我们不知道最终的四元数。通过使用合成，我们可以在几何体上合成两个旋转并得到最终单个的四元数。然后，我们可以在原始四元数和合成的四元数间进行插值，得到两者之间的平滑转换。
Direct3D扩展（D3DX）工具库包含了帮助用户使用四元数的函数。例如，
D3DXQuaternionRotationAxis函数给一个定义旋转轴的向量增加一个旋转值，并在由D3DXQUTERNION结构定义的四元数中返回结果。另外，D3DXQuaternionMultiply函数合成四元数，D3DXQuaternionSlerp函数在两个四元数间进行球面线性插值（spherical linear interpolation）。
Direct3D 应用程序可以使用下列函数简化对四元数的使用。
D3DXQuaternionBaryCentric
D3DXQuaternionConjugate
D3DXQuaternionDot
D3DXQuaternionExp
D3DXQuaternionIdentity
D3DXQuaternionInverse
D3DXQuaternionIsIdentity
D3DXQuaternionLength
D3DXQuaternionLengthSq
D3DXQuaternionLn
D3DXQuaternionMultiply
D3DXQuaternionNormalize
D3DXQuaternionRotationAxis
D3DXQuaternionRotationMatrix
D3DXQuaternionRotationYawPitchRoll
D3DXQuaternionSlerp
D3DXQuaternionSquad
D3DXQuaternionToAxisAngle
Direct3D 应用程序可以使用下列函数简化对三成员向量的使用。
D3DXVec3Add
D3DXVec3BaryCentric
D3DXVec3CatmullRom

D3DXVec3Cross
D3DXVec3Dot
D3DXVec3Hermite
D3DXVec3Length
D3DXVec3LengthSq
D3DXVec3Lerp
D3DXVec3Maximize
D3DXVec3Minimize
D3DXVec3Normalize
D3DXVec3Project
D3DXVec3Scale
D3DXVec3Subtract
D3DXVec3Transform
D3DXVec3TransformCoord
D3DXVec3TransformNormal
D3DXVec3Unproject
D3DX工具库提供的数学函数中包含了许多辅助函数，可以简化对二成员和四成员向量的使用。

Direct3D 对象

Microsoft® Direct3D®是通过组件对象模型（COM）对象和接口实现的。C++应用程序可以直接访问这些接口和对象，而 Microsoft Visual Basic®应用程序则与一层被称为 Microsoft DirectX® for Visual Basic Classes 的代码进行交互，它们为 Visual Basic 应用程序组织数据并传送给 DirectX 运行库（run time）。

Direct3D 对象是应用程序第一个创建并最后一个释放的对象。可以通过 Direct3D 对象访问一些函数，用它们枚举并取得 Direct3D 设备的能力，这允许应用程序选择设备而无需创建它们。

当一个C++应用程序启动时,它必须取得一个指向IDirect3D9接口的指针以使用Direct3D的功能。以下示例代码显示了如何用Direct3DCreate9函数取得一个指向Direct3D接口的指针。

```
LPDIRECT3D9 g_pD3D = NULL;

if( NULL == (g_pD3D = Direct3DCreate9(D3D_SDK_VERSION)))
    return E_FAIL;
```

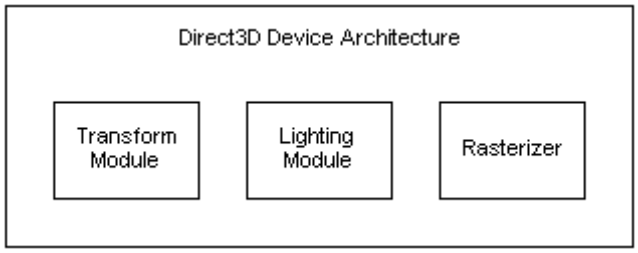
要从Direct3DDevice对象取得创建它的Direct3D对象，可以用IDirect3DDevice9::GetDirect3D方法。

设备

Microsoft® Direct3D®设备是 Direct3D 的渲染部件，它封装并储存渲染状态。此外，Direct3D 设备还执行变换和光照操作以及把图像光栅化到表面上。

设备类型
创建设备
选择设备
丢失的设备
检测硬件支持
处理顶点数据
设备支持的图元类型

从结构上讲，Direct3D 设备包含一个变换模块、一个光照模块和一个光栅化模块，如下图所示。



当前 Direct3D 支持两种主要类型的 Direct3D 设备：一种是硬件抽象层（HAL）设备，HAL 设备具有硬件光栅化加速，并可用软硬件顶点处理进行着色操作；另一种是参考设备。

可以认为这些设备是两个单独的驱动程序。软件和参考设备由软件驱动程序表示，而 HAL 设备由硬件驱动程序表示。这些设备最通常的用法是使用 HAL 设备发行应用程序，而用参考设备做特性测试。这些参考设备由第三方提供用于模拟特定的设备——例如，尚未发布的还在开发中的硬件。

应用程序创建的设备必须与应用程序所运行的图形硬件的能力相对应。Direct3D 以两种方式提供对渲染的支持，一种是访问安装在计算机中的三维硬件，另一种是用软件模拟三维硬件的能力。因此，对于硬件访问和软件模拟，Direct3D 都提供了相应的设备。

硬件加速设备提供了比软件设备好得多的性能。HAL 设备类型在所有支持 Direct3D 的图形适配器上都可用。在大多数情况下，应用程序将具有硬件加速的计算机作为目标平台，并依赖软件模拟以适应低端计算机。

除了参考设备，软件设备并不总是支持与硬件设备相同的特性。应用程序应该总是查询设备能力以确定该设备支持的特性。

因为 Microsoft DirectX® 9.0 提供的软件设备和参考设备的行为与 HAL 设备完全相同，所以针对 HAL 设备开发的应用程序代码也可以用在软件设备或参考设备上而无需修改。注意，虽然 Direct3D 提供的软件设备或参考设备的行为与 HAL 设备完全相同，但是设备的能力肯定会不同，某特定的软件设备可能仅实现一个较小的能力集。

行为

Direct3D允许应用程序指定设备的行为和设备的类型。IDirect3D9::CreateDevice方法允许用一个或多个行为标志的组合控制Direct3D设备的整体行为。这些标志指定让Direct3D运行库维护哪些以及不维护哪些行为，同时设备类型指定使用哪个驱动程序。虽然有些设备行为标志的组合是无效的，但是在各种不同类型的设备上还是有可能会用到所有的设备行为标志的。例如，在一个用D3DCREATE_PUREDEVICE标志创建的设备上指定D3DDEVTYPE_SW是有效的。

设备类型

HAL 设备

最主要的设备类型是硬件抽象层（HAL）设备，它支持硬件光栅化加俸腿碛布　艹愦　怼H绻τ贸绦蛟谝煌悼δ沧傲酥C諱 icrosoft® Direct3D®的显示适配器的计算机上运行，那么应用程序应该用它进行三维操作。Direct3D HAL 设备用硬件实现变换、光照及光栅化模块的部分或全部。应用程序不直接访问三维加速卡，它们调用 Direct3D 的函数和方法，而 Direct3D 通过 HAL 访问硬件。如果应用程序在支持 HAL 的计算机上运行，那么通过使用 HAL 设备它将获得最佳的性能。

要在C++程序中创建一个HAL设备，应该调用IDirect3D9::CreateDevice方法，并将D3DDEVTYPE_HAL常数作为设备类型传入。

注意 硬件设备不能渲染到 8 位渲染目标表面。

参考设备

Direct3D 支持另一种称为参考设备或参考光栅化器的设备类型。与软件设备不同的是，参考光栅化器支持所有 Direct3D 特性。因为这些特性的实现是为了精确而不是为了速度，并且是用软件实现，所以结果不会很快。虽然参考光栅化器尽可能地使用了特殊的 CPU 指令，但它不是为正

式零售的应用程序准备的。应该仅把参考光栅化器用于特性测试或演示用途。
要在 C++程序中创建一个参考设备，应该调用 IDirect3D9::CreateDevice 方法，并将 D3DDEVTYPE_REF 常数作为设备类型传入。
创建设备
注意由 Microsoft® Direct3D®对象创建的所有渲染设备共享相同的物理资源。虽然应用程序可以从单个 Direct3D 对象创建多个渲染设备，但因为它们共享相同的硬件，所以会导致严重的性能下降。
要在C++应用程序中创建一个Direct3D设备，应用程序必须先创建一个Direct3D对象，如Direct3D对象中所述。
首先，初始化用于创建Direct3D设备的D3DPRESENT_PARAMETERS结构的值。以下示例代码指定了一个窗口应用程序，其后缓存（back buffer）只有在垂直回扫（VSYNC）时才翻转到前缓存（front buffer）。

```
LPDIRECT3DDEVICE9 pDevice = NULL;

D3DPRESENT_PARAMETERS d3dpp;

ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed   = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_COPY;
```

下一步，创建Direct3D设备。以下IDirect3D9::CreateDevice调用指定了默认的适配器，硬件抽象层（HAL）设备，以及软件顶点处理。

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                  &d3dpp, &d3dDevice ) ) )
    return E_FAIL;
```

注意，对创建、释放或重置设备的调用应该仅发生在焦点窗口的窗口处理函数所在的线程中。
创建设备后，应该设置它的状态。
选择设备
要检测硬件支持的 Microsoft® Direct3D®设备类型，应用程序可以对硬件进行查询。本节包含了枚举显示适配器和选择 Direct3D 设备所涉及到的主要任务的信息。
要选择适当的 Direct3D 设备，应用程序必须执行一系列任务。注意以下步骤是全屏应用程序使用的。大多数情况下，窗口应用程序可以跳过其中的多数步骤。
首先，应用程序必须枚举系统中的显示适配器。一个适配器是一块物理硬件。注意图形卡可能包含一个以上适配器，如双头显示的情况。与多显示器无关的应用程序可以忽略这一步，并在第 2 步时将D3DADAPTER_DEFAULT参数传给IDirect3D9::EnumAdapterModes方法。
对每一个适配器，应用程序通过调用 IDirect3D9::EnumAdapterModes 枚举该适配器支持的显示模式。
如果需要，应用程序可以通过调用IDirect3D9::CheckDeviceType检查每一个被枚举的显示模式是否具备硬件加速，如以下示例代码所示。注意这仅是IDirect3D9::CheckDeviceType的用法之一，更多细节请参阅检测硬件支持。

```
D3DPRESENT_PARAMETERS Params;
// 初始化 D3DPRESENT_PARAMETERS 成员的值。

Params.BackBufferFormat = D3DFMT_X1R5G5B5;
```

```
if(FAILED(m_pD3D->CheckDeviceType(Device.m_uAdapter,
                                  Device.m_DevType,
                                  Params.BackBufferFormat, Params.BackBufferFormat,
                                  FALSE)))
        return E_FAIL;
```

应用程序通过调用IDirect3D9::GetDeviceCaps方法检查此适配器上的设备对希望使用的功能的支持度。此方法会过滤掉不支持所需的功能的设备。对经由IDirect3D9::CheckDeviceType验证过所有显示模式的设备，可以保证IDirect3D9::GetDeviceCaps返回的设备能力是不变的。

若设备支持被枚举的显示模式，则设备总是可以渲染到这些格式的表面。若应用程序需要渲染到不同格式的表面，则可以调用IDirect3D9::CheckDeviceFormat。若设备可以渲染到这种格式的表面，则可以保证所有IDirect3D9::GetDeviceCaps返回的能力都是可用的。

最后，应用程序可以使用IDirect3D9::CheckDeviceMultiSampleType方法检测设备是否支持多重取样技术，如全屏抗锯齿。

完成以上步骤后，应用程序应该有一个可用的显示模式列表。最后一步是验证有足够的设备可存取内存可供使用，以容纳所需数量的缓存和抗锯齿所需的内存。这个测试是必需的，因为不同显示模式和多重取样的组合所消耗的内存不通过验证是无法预测的。此外，有些显示适配器的体系结构可能无法保证设备可存取内存的数量保持不变。这意味着当切换到全屏模式时，应用程序应该有能力报告视频内存（此后简称为显存）用尽的错误。一般来说，应用程序应该在提供给用户的模式列表中移除全屏模式，或者应该通过减少后缓存的数量或使用不太复杂的多重取样技术以试图消耗较少的内存。

窗口应用程序执行一系列类似的操作。

检测被窗口的客户区覆盖的桌面矩形。

枚举适配器，找到覆盖该客户区的相应适配器。如果客户区被一个以上适配器拥有，那么应用程序可以选择单独处理每个适配器，或仅处理单个适配器，并由 Direct3D 在 presentation 时把像素从一个设备上传送到另一设备上。应用程序也可以忽略以上两步并使用 D3DADAPTER_DEFAULT。注意当窗口被放置在第二个显示器上时，运行速度可能会比较慢。

应用程序应该调用IDirect3D9::CheckDeviceType检测在桌面模式下设备是否支持渲染到指定格式的后缓存。IDirect3D9::GetAdapterDisplayMode可用于检测桌面显示格式，如以下示例代码所示。

```
D3DPRESENT_PARAMETERS Params;
// 初始化 D3DPRESENT_PARAMETERS 成员的值。

// 使用当前的显示模式。
D3DDISPLAYMODE mode;

if(FAILED(m_pD3D->GetAdapterDisplayMode(Device.m_uAdapter , &mode)))
        return E_FAIL;

Params.BackBufferFormat = mode.Format;

if(FAILED(m_pD3D->CheckDeviceType(Device.m_uAdapter, Device.m_DevType,
Params.BackBufferFormat, Params.BackBufferFormat, FALSE)))
       return E_FAIL;
```

丢失的设备

一个Microsoft® Direct3D®可以处于操作状态或丢失状态。操作状态是设备的正常状态，设备按预期运行并present所有渲染结果。当事件发生时，如全屏应用程序失去键盘输入焦点，设备就转变到丢失状态，这会导致渲染无法进行。丢失状态表现为所有渲染操作的悄然失败，这意味着即使渲染操作失败所有的渲染方法仍可以返回成功码。在这种情况下，

IDirect3DDevice9::Present返回错误码D3DERR_DEVICELOST。

Direct3D有意没有对可能导致设备丢失的所有情况进行详细说明。一些典型的例子包括窗口失去焦点，例如用户按下了ALT+TAB或弹出了一个系统对话框。设备也会因为电源管理事件而丢失，或者另一个应用程序进行全屏操作。另外，任何对IDirect3DDevice9::Reset调用的失败会把设备置为丢失状态。

注意 可以保证所有继承自 IUnknown 的方法在设备丢失后仍能正常工作。设备丢失后，每个函数一般有三种可能：

调用失败，返回值为 D3DERR_DEVICELOST – 这意味着应用程序必须发现设备已经丢失，从而知道一些事情没有按照预期进行。

悄然失败，返回值为 S_OK 或其它值 – 若函数调用悄然失败，则应用程序一般无法区分出“调用成功”或“悄然失败”。

函数返回一个返回值。

对丢失的设备作出响应

设备在被重置后，应该重新创建资源（包括显存资源）。如果设备丢失了，那么应用程序应该查询设备状态，看是否可以将之恢复回操作状态。如果不行，那么就等到设备可以被恢复为止。

如果设备可以被恢复，那么应用程序应该销毁所有显存资源和交换链，并准备恢复。然后，应用程序调用IDirect3DDevice9::Reset方法。Reset方法是当设备丢失时唯一有效的方法，并且是应用程序可用来把设备从丢失状态恢复到操作状态的唯一方法。除非应用程序释放所有在D3DPOOL_DEFAULT中分配的资源，包括用IDirect3DDevice9::CreateRenderTarget和IDirect3DDevice9::CreateDepthSstencilSurface方法创建的资源，否则Reset将会失败。

Direct3D中大部分被频繁调用的方法不返回任何关于设备是否已丢失的信息。应用程序可以继续调用渲染方法，如IDirect3DDevice9::DrawPrimitive，而不会收到设备丢失的通知。在Direct3D内部，这些操作被抛弃，直到设备被重置为操作状态为止。

通过查询IDirect3DDevice9::TestCooperativeLevel方法的返回值，应用程序可以决定在遇到设备丢失时如何处理。

锁定操作

为了确保在设备丢失后锁定操作仍会成功，Direct3D 在内部做了很多工作，但是，它并不保证在锁定操作时显存资源中的数据是正确的，Direct3D 只保证不返回错误代码。这使得在编写应用程序时，不必关心在锁定操作时设备丢失与否。

资源

资源会消耗显存。因为丢失的设备与适配器拥有的显存的连接被切断，所以当设备丢失时不可能保证显存的分配。因此，在这种情况下，所有用于创建资源的方法都被实现为成功返回 D3D_OK，而实际上只分配假的系统内存。因为在调整设备的大小之前任何显存资源必须被销毁，所有不会存在显存被过度分配的问题。这些假的表面使锁定和复制操作看起来运行正常，直到应用程序调用 IDirect3DDevice9::Present 并发现设备已经丢失。

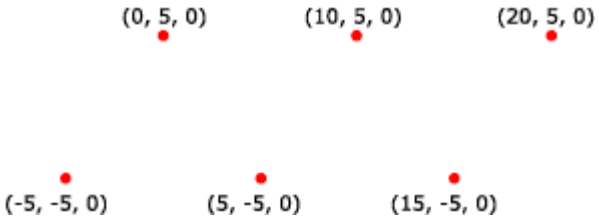
在可以把设备从丢失状态重置为操作状态前，所有显存必须被释放。这意味着应用程序应该释放所有用IDirect3DDevice9::CreateAdditionalSwapChain创建的交换链和所有放在D3DPOOL_DEFAULT内存类型中的资源。应用程序无需释放在D3DPOOL_MANAGED或D3DPOOL_SYSTEMMEM内存类型中的资源。其余状态数据在改变到操作状态时会被自动销毁（译注：

如后缓存）。
Direct3D 鼓励在开发应用程序时用同一部分代码对设备丢失做出响应。这部分代码和启动应用程序时用于初始化设备的那部分代码即使不完全相同，也应该差不多。
取回的数据
Direct3D允许应用程序通过IDirect3DDevice9::ValidateDevice对照单次渲染用硬件验证纹理和渲染状态。这种方法一般在应用程序初始化时调用，如果设备已经丢失，那么该方法将返回D3DERR_DEVICELOST。
Direct3D 也允许应用程序把生成的（译注：即渲染得到的）或以前写入到显存中的图像从显存资源中复制到非易失性的（nonvolatile）内存资源中。因为这类传输中的源图像可能在任何时候丢失，所以当设备丢失时 Direct3D 允许这类操作失败。
关于异步查询，如果使用了FLUSH标志，那么IDirect3DQuery9::GetData将返回D3DERR_DEVICELOST，这是为了告诉应用程序IDirect3DQuery9::GetData方法根本不会返回S_OK，。
当设备丢失时，因为不存在主表面，所以复制操作IDirect3DDevice9::GetFrontBufferData会失败，返回值为D3DERR_DEVICELOST。当设备丢失时，IDirect3DDevice9::CreateAdditionalSwapChain也会因为无法创建后缓存而失败，并返回D3DERR_DEVICELOST。注意除了IDirect3DDevice9::Present，IDirect3DDevice9::TestCooperativeLevel和IDirect3DDevice9::Reset方法之外，D3DERR_DEVICELOST返回值只可能出现在以上两种情况中。
可编程着色器
在Microsoft DirectX® 9.0 中，Vertex Shader 1 1和Pixel Shader 1 X在Reset后不需要被重置，它们会被记住。在DirectX的前一个版本中，设备丢失后需要重新创建着色器。
检测硬件支持
Microsoft® Direct3D®为检测硬件支持提供了以下函数。
IDirect3D9::CheckDeviceFormat
用于验证某个表面格式是否可被用作纹理，某个表面格式是否可同时被用作纹理和渲染目标，或某个表面格式是否可被用作深度/模板缓存。另外，这个方法可用于验证对深度缓存格式的支持和对深度/模板缓存格式的支持。
IDirect3D9::CheckDeviceType
用于验证设备执行硬件加速的能力，设备创建 presentation 交换链的能力，或设备渲染到当前显示格式的能力。
IDirect3D9::CheckDepthStencilMatch
用于验证是否某个深度/模板缓存格式与某个渲染目标格式相兼容。注意在调用此方法前，应用程序应该同时对深度/模板格式和渲染目标格式调用 IDirect3D9::CheckDeviceFormat。
处理顶点数据
IDirect3DDevice9接口同时支持软件和硬件顶点处理。一般来说，设备的软件和硬件顶点处理能力是不完全一样的。硬件能力是可变的，取决于显示适配器和驱动程序，而软件能力则是固定的。下列标志控制硬件抽象层（HAL）和参考设备的顶点处理状态。
D3DCREATE_SOFTWARE_VERTEXPROCESSING
D3DCREATE_HARDWARE_VERTEXPROCESSING
D3DCREATE_MIXED_VERTEXPROCESSING
当调用IDirect3D9::CreateDevice时，要指定以上顶点处理状态标志中的一个。混合模式标志允许设备同时执行软件和硬件顶点处理。在任意时刻，只能给某个设备设置一个顶点处理标志。注意当创建一个纯设备（pure device，D3DCREATE_PUREDEVICE）时需要设置

D3DCREATE_HARDWARE_VERTEXPROCESSING标志。
为了避免在一个设备上的双重顶点处理能力，只有硬件顶点处理能力可以在运行的时候查询。软件顶点处理能力是固定的，不能在运行的时候查询。
要确定设备的硬件顶点处理能力，可以参考D3DCAPS9结构的VertexProcessingCaps成员。软件顶点处理支持以下能力。
D3DVTXPCAPS_DIRECTIONALLIGHTS
D3DVTXPCAPS_LOCALVIEWER
D3DVTXPCAPS_MATERIALSOURCE7
D3DVTXPCAPS_POSITIONALLIGHTS
D3DVTXPCAPS_TEXGEN
D3DVTXPCAPS_TWEENING
另外，下表列出了在软件顶点处理模式下设备的 D3DCAPS9 结构成员的值。

成员	软件顶点处理能力
MaxActiveLights	没有限制
MaxUserClipPlanes	6
MaxVertexBlendMatrices	4
MaxStreams	16
MaxVertexIndex	0xFFFFFFFF

软件顶点处理提供了一个可以保证的顶点处理能力集，包括不限数量的光源和对可编程顶点着色器的完全支持。在使用 HAL 设备时，可在任何时间在软件和硬件顶点处理间切换。HAL 设备是唯一同时支持软件和硬件顶点处理的设备类型，但唯一的要求是用于软件顶点处理的顶点缓存必须分配在系统内存中。
注意硬件顶点处理和软件顶点处理的性能是相当的，因此在一个设备类型中同时提供顶点处理的硬件加速和软件仿真是个不错的想法。而光栅化器则不同，主处理器比特定的图形硬件慢得多，因此在一个设备类型中不会同时提供硬件和软件仿真。软件顶点处理是在一个设备类型中 Direct3D 运行库和硬件（驱动程序）功能重复的唯一例子，因此所有其余的设备能力代表了由驱动程序提供的潜在可变的功能。
设备支持的图元类型
Microsoft® Direct3D®设备可以创建并管理以下类型的图元。
点表 (Point List)
线表 (Line List)
线带 (Line Strip)
三角形表 (Triangle List)
三角形带 (Triangle Strip)
三角形扇 (Triangle Fan)
可以在C++应用程序中用IDirect3DDevice9接口提供的任何渲染方法渲染各种类型的图元。
点表
点表是一个顶点的集合，被渲染为孤立的点。应用程序可以将它们用于星空，或多边形表面的虚线。
下图描绘了一个渲染得到的点表。



应用程序可以将材质和纹理应用于点表。材质和纹理的颜色只影响绘制得到的那些点，不会影响各点之间的任何其它地方。

以下示例代码显示了如何为这个点表创建顶点。

```
struct CUSTOMVERTEX
{
    float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
    {-5.0, -5.0, 0.0},
    { 0.0,  5.0, 0.0},
    { 5.0, -5.0, 0.0},
    {10.0,  5.0, 0.0},
    {15.0, -5.0, 0.0},
    {20.0,  5.0, 0.0}
};
```

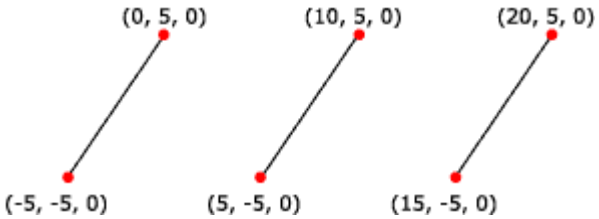
以下示例代码显示了如何使用IDirect3DDevice9::DrawPrimitive渲染这个点表。

```
//
// 这里假定 d3dDevice 是指向 IDirect3DDevice9 接口的有效指针
//
d3dDevice->DrawPrimitive( D3DPT_POINTLIST, 0, 6 );
```

线表

线表是一个孤立线段的集合。线表对诸如给三维场景加入雨雪或大雨这样的任务很有用。

下图描绘了一个渲染得到的线表。



可以将材质和纹理用于线表。材质和纹理的颜色只会影响绘制得到的那些线段，而不会影响各线段之间的任何其它地方。

以下示例代码显示了如何为这个线表创建顶点。

```
struct CUSTOMVERTEX
{
    float x,y,z;
};
```

```
CUSTOMVERTEX Vertices[] =
{
    {-5.0, -5.0, 0.0},
    { 0.0,  5.0, 0.0},
    { 5.0, -5.0, 0.0},
    {10.0,  5.0, 0.0},
    {15.0, -5.0, 0.0},
    {20.0,  5.0, 0.0}
};
```

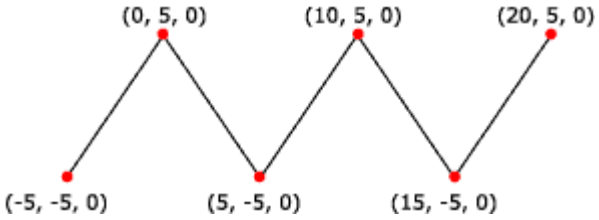
以下示例代码显示了如何使用IDirect3DDevice9::DrawPrimitive渲染这个线表。

```
//
// 这里假定 d3dDevice 是指向 IDirect3DDevice9 接口的有效指针
//
d3dDevice->DrawPrimitive( D3DPT_LINELIST, 0, 3 );
```

线带

线带是由相互连接的线段组成的图元。应用程序可以将线段带用于创建不封闭的多边形，封闭多边形是最后一个顶点与第一个顶点通过一条线段相连的多边形。如果应用程序使用基于线段带的多边形，那么不能保证顶点是共面的。

下图描绘了一个渲染得到的线段带。



以下示例代码显示了如何为这个线段带创建顶点。

```
struct CUSTOMVERTEX
{
    float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
    {-5.0, -5.0, 0.0},
    { 0.0,  5.0, 0.0},
    { 5.0, -5.0, 0.0},
    {10.0,  5.0, 0.0},
    {15.0, -5.0, 0.0},
    {20.0,  5.0, 0.0}
};
```

以下示例代码显示了如何使用IDirect3DDevice9::DrawPrimitive渲染这个线段带。

```
//
// 这里假定 d3dDevice 是指向 IDirect3DDevice9 接口的有效指针
```

```
//
d3dDevice->DrawPrimitive( D3DPT_LINESTRIP, 0, 5 );
```

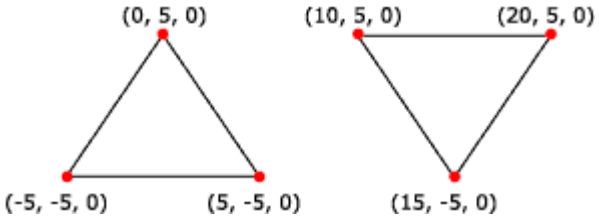
三角形表

三角形表是一个孤立三角形的集合。它们可能也可能不彼此相邻。三角形表必须至少有三个顶点，全部顶点的数量必须是三的倍数。

可以将三角形表用于创建由不相连的部分组成的物体。例如，在三维游戏中创建一个能量场的一种方法就是指定一个很大的三角形表，它由许多小且不相连的三角形组成的。然后将看起来发光的材质和纹理应用于三角形表。墙上的每个三角形看起来都发光。穿过三角形之间的缝隙，可以看到墙后面的一部分，这就和玩家看一个能量场时所应该看到的一样。

三角形表在创建具有锋利边缘并使用高洛德着色算法进行着色的物体时同样有用。请参阅面和顶点法向量。

下图描绘了一个渲染得到的三角形表。



以下示例代码显示了如何为这个三角形表创建顶点。

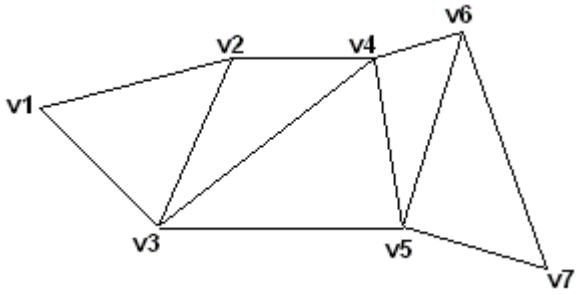
```
struct CUSTOMVERTEX
{
    float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
    {-5.0, -5.0, 0.0},
    { 0.0,  5.0, 0.0},
    { 5.0, -5.0, 0.0},
    {10.0,  5.0, 0.0},
    {15.0, -5.0, 0.0},
    {20.0,  5.0, 0.0}

};
```

以下示例代码显示了如何使用IDirect3DDevice9::DrawPrimitive渲染这个三角形表。

```
//
// 这里假定 d3dDevice 是指向 IDirect3DDevice9 接口的有效指针
//
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 2 );
```

三角形带

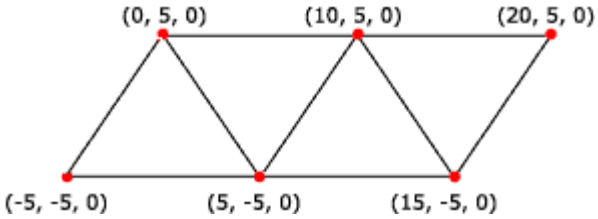
三角形带是一系列相连的三角形。因为三角形是相连的，所以应用程序无需为每个三角形重复指定所有三个顶点。例如，定义以下三角形带只需七个顶点。



系统用顶点 v1，v2 和 v3 画第一个三角形，v2，v4 和 v3 画第二个三角形，v3，v4 和 v5 画第三个，v4，v6 和 v5 画第四个，依次类推。注意第二和第四个三角形的顶点顺序是颠倒的，为了确保所有三角形都按顺时针的顺序绘制，这是必需的。

三维场景中的大多数物体都由三角形带组成，这是因为用三角形带指定复杂的物体，可以有效地节省内存和处理时间。

下图描绘了一个渲染得到的三角形带。



以下示例代码显示了如何为这个三角形带创建顶点。

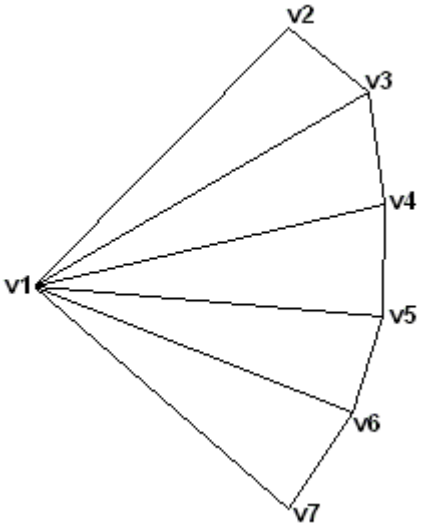
```
struct CUSTOMVERTEX
{
    float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
    {-5.0, -5.0, 0.0},
    { 0.0,  5.0, 0.0},
    { 5.0, -5.0, 0.0},
    {10.0,  5.0, 0.0},
    {15.0, -5.0, 0.0},
    {20.0,  5.0, 0.0}
};
```

以下示例代码显示了如何使用IDirect3DDevice9::DrawPrimitive渲染这个三角形带。

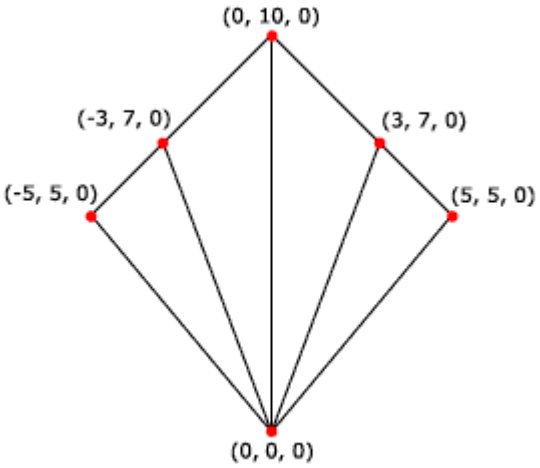
```
//
// 这里假定 d3dDevice 是指向 IDirect3DDevice9 接口的有效指针
//
d3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 4);
```

三角形扇

除了所有三角形共享同一个顶点以外，三角形扇与三角形带相似。如下图所示。



系统使用顶点 v2，v3 和 v1 画第一个三角形，v3，v4 和 v1 画第二个三角形，v4，v5 和 v1 画第三个三角形，依次类推。当启用平面着色时，系统使用第一个顶点的颜色对三角形进行着色。下图描绘了一个渲染得到的三角形扇。



以下示例代码显示了如何为这个三角形扇创建顶点。

```
struct CUSTOMVERTEX
{
    float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
    { 0.0,  0.0,  0.0},
    {-5.0,  5.0,  0.0},
    {-3.0,   7.0,  0.0},
    { 0.0,  10.0,  0.0},
    { 3.0,   7.0,  0.0},
    { 5.0,   5.0,  0.0},
};
```

以下示例代码显示了如何使用IDirect3DDevice9::DrawPrimitive渲染这个三角形扇。

```
//
// 这里假定 d3dDevice 是指向 IDirect3DDevice9 接口的有效指针
//
d3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 4 );
```

资源

资源是用于渲染场景的纹理和缓存。应用程序需要创建、载入、复制和使用资源。本节简要介绍资源及应用程序使用资源所需的步骤和方法。更多有关特定资源类型的信息，请参阅纹理，顶点缓存和索引缓存。

所有资源，包括几何体资源IDirect3DIndexBuffer9和IDirect3DVertexBuffer9都继承自IDirect3DResource9接口。纹理资源，IDirect3DCubeTexture9，IDirect3DTexture9和IDirect3DVolumeTexture9都继承自IDirect3DBaseTexture9接口。

信息被分为以下主题：

资源属性

操控资源

锁定资源

资源间的关系

管理资源

由应用程序管理的资源及分配策略

资源属性

所有资源共享以下属性。

用途。资源的用法，例如，作为纹理还是作为渲染目标。

格式。数据的格式，例如，二维表面的像素格式。

池（Pool）。分配资源的内存类型。

类型。资源的类型，例如，顶点缓存或渲染目标。

对资源的使用是强制的。若应用程序要将资源用于某一操作，则必须在资源创建时说明该操作。关于为资源定义的用途常数表，请参阅D3DUSAGE。

D3DUSAGE_RTPATCHES，*D3DUSAGE_NPATCHES* 和 *D3DUSAGE_POINTS* 常数告诉驱动程序这些缓存将分别被用于 triangular patchs，grid patchs，N-patchs，或点精灵。提供这些标志是为了以防万一硬件不使用主机处理就无法执行这些操作，因此驱动程序希望把这些表面分配在系统内存中，让 CPU 可以访问它们。如果驱动程序可以完全用硬件执行这些操作，那么它可以将这些表面分配在加速图形接口（AGP）内存中，这样既避免了在主机保存一份复本，又提高了性能，一举两得。注意这些标志提供的信息并不是绝对必需的。驱动程序可以检测正在对数据执行这类操作，并在随后几帧把缓存移回系统内存。

有关用途标志及它们与特定资源如何关联的细节，请参阅每个资源创建方法的参考页。

有关资源的表面格式的信息，请参阅D3DFORMAT枚举类型。

存放资源缓存的那种内存被称为池。池的值由D3DPOOL枚举类型定义。对于单个资源包含的不同对象（也就是说，mipmap中的mip levels），池不能混合，并且一旦为某资源选择了池，池就不能再被改变了。

资源类型是在运行的时候，当应用程序调用诸如IDirect3DDevice9::CreateCubeTexture之类的资源创建方法时隐式设置的。资源类型由D3DRESOURCETYPE枚举类型定义。应用程序可以在运行的时候查询这些类型，但是，最好大多数场景都无需进行运行时类型检查。

操控资源

为了渲染场景，应用程序需要操控资源。首先，应用程序用以下方法创建纹理资源。

IDirect3DDevice9::CreateCubeTexture

IDirect3DDevice9::CreateTexture

IDirect3DDevice9::CreateVolumeTexture

由纹理创建方法返回的纹理对象是表面或立体纹理的容器，这些容器一般被称为缓存。资源拥有的缓存在继承资源的用途、格式和池的同时也有自己的类型。更多信息请参阅资源属性。

为了载入图片，应用程序应该调用以下方法，得到对所包含的表面的访问权。更多细节请参阅锁定资源。

IDirect3DCubeTexture9::LockRect

IDirect3DTexture9::LockRect

IDirect3DVolumeTexture9::LockBox

lock 方法接收的参数指出了所包含的表面——例如，纹理的 mipmap sub-level 或立方体纹理的表面——并返回指向像素的指针。一般的应用程序从不直接使用表面对象。

另外，应用程序用以下方法创建面向几何体的资源。

IDirect3DDevice9::CreateIndexBuffer

IDirect3DDevice9::CreateVertexBuffer

应用程序通过调用以下方法锁定并填充缓存资源。

IDirect3DIndexBuffer9::Lock

IDirect3DVertexBuffer9::Lock

如果应用程序允许Microsoft® Direct3D®运行库管理这些资源，那么资源创建过程就到此为止。否则，应用程序还要通过调用IDirect3DDevice9::UpdateTexture方法，负责把系统资源提升为设备可访问的资源，这样硬件加速器就可以使用它们。

要呈现(present)从资源渲染得到的图像，应用程序还需要颜色和深度/模板缓存。对于一般的应用程序，颜色缓存属于设备的交换链，交换链是一个后缓存表面的集合，由设备隐式地创建。深度/模板缓存表面可以被隐式创建，或使用IDirect3DDevice9::CreateDepthStencilSurface方法显式创建。应用程序通过调用IDirect3DDevice9::SetRenderTarget方法将设备与它的深度和颜色缓存关联起来。

更多有关呈现(presenting)最终图像的细节，请参阅呈现场景(Presenting a Scene)。

锁定资源

锁定资源意味着允许 CPU 对它的存储器进行访问，Direct3D 为资源定义了以下锁定方法。

D3DLOCK_DISCARD

D3DLOCK_READONLY

D3DLOCK_NOOVERWRITE

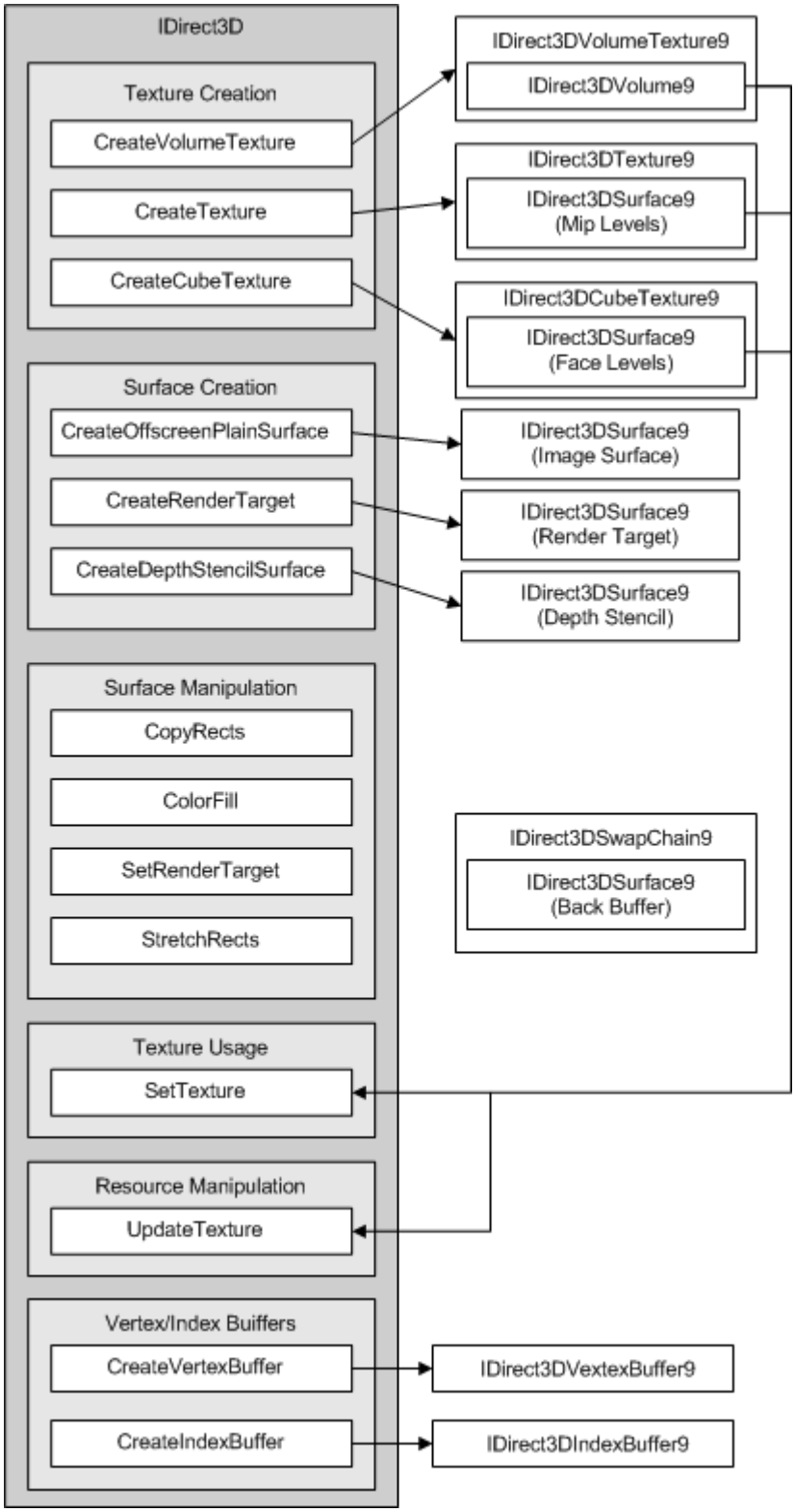
D3DLOCK_NOSYSLOCK

D3DLOCK_NO_DIRTY_UPDATE

有关锁定标志及它们如何与特定资源关联的细节，请参阅每个资源锁定方法的参考页。应用程序开发者应该注意 D3DLOCK_DISCARD，D3DLOCK_READONLY 和 D3DLOCK_NOOVERWRITE 标志只是提示。运行库不检查应用程序是否遵循由这些标志指定的功能。可以想象如果应用程序先指定 D3DLOCK_READONLY 但然后写入资源，那么肯定会导致未定义的结果。一般来说，不正当地使用锁定标志，包括锁定用途标志，将无法保证今后释放操作会成功并可能导致严重的性能下降。

一个锁定操作后应该跟随一个解锁操作。例如，在锁定一个纹理后，应用程序随后通过给锁定的纹理解锁以结束对它的访问。除了允许处理器的访问外，在锁定期间，任何涉及到该资源的其它操作被序列化（译注：可能是在锁定期间，把涉及到该表面的其它操作都记录下来，但并不真正执行，等到解锁后再执行被记录下来但并未执行的操作）。对资源的锁定只允许有一次，即使是

不重叠的区域，只要一个表面的锁定操作仍未完成，硬件加速器就不能对该表面进行操作。
每种资源接口都有锁定该资源所包含的缓存的方法。每种纹理资源还可以锁定它的一部分。二维资源（表面）允许锁定子矩形(sub-rectangle)，立体纹理资源允许锁定子卷或子盒（sub volume or boxes）。每种锁定方法返回一个结构，它包含了指向存放资源的存储器的指针和表示行或位面数据之间的距离值（译注：亦即 pitch），该距离值取决于资源的类型。更多细节，请参阅资源接口的方法列表。返回的指针总是指向被锁定子区域的左上角。
当使用索引或顶点缓存时，可以进行多次锁定调用，但是，必须确保锁定调用的数量与解锁调用的数量相同。
因为 Microsoft DirectX®的压缩纹理格式将像素编码为 4×4 的块进行存储，所以只能在 4×4 的边界进行锁定操作。
资源间的关系
下图描绘了在句法上可能对特定资源及其所含内容进行的操作。更多有关如何使用资源的信息，请参阅管理资源。



从左到右的箭头表示目标类型由箭头指向的方法创建,而从右到左的箭头表示这些类型的资源可以作为参数传入由箭头指向的方法。

管理资源

资源管理是将资源从系统内存提升到设备可访问存储器及从设备可访问存储器中抛弃的过程。Microsoft® Direct3D®运行库有自己的基于最近最少使用（least-recently-used）优先级技术的管理算法。当Direct3D检测到在一帧中——在IDirect3DDevice9::BeginScene和

IDirect3DDevice9::EndScene调用之间——设备可访问内存无法同时存储所有资源时，它就切换到最近最多使用（most-recently-used）优先级技术。
在创建时使用D3DPOOL_MANAGED标志指定一个由系统管理的资源。由系统管理的资源在设备的丢失状态和操作状态间的转换中持续存在。通过调用IDirect3DDevice9::Reset设备可以被重置，并且这类资源可以继续正常运作而无需重新载入图片。但是，如果设备必须被销毁和重建，那么所有用D3DPOOL_MANAGED创建的资源也必须被重建。
在创建时使用 D3DPOOL_DEFAULT 标志指定把资源放在默认的池中。在默认的池中的资源在设备从丢失状态到操作状态的转换过程中不持续存在，这些资源必须在调用 Reset 之前释放，然后重建。更多有关设备的丢失状态的信息，请参阅丢失的设备。
注意不是所有的类型和用途都支持资源管理。例如，用 D3DUSAGE_RENDERTARGET 标志创建的对象不支持资源管理。另外，不建议对需要频繁改变其内容的对象使用资源管理。例如，在某些硬件上对一个每帧都需改变的顶点缓存进行自动管理会严重降低性能。但是，对纹理资源来说这不是一个问题。
由应用程序管理的资源及分配策略
由系统管理的顶点缓存或索引缓存在创建时不能通过指定 D3DUSAGE_DYNAMIC 标志被声明为动态的，这就使得对顶点缓存内容的每一次修改都需要一次额外的复制操作。动态顶点缓存用于渲染动态几何体，数据从 BSP 树（binary space partition tree）或其它用于可见判别的数据结构引入。为了完成这个任务，可以预先为想要使用的格式分配缓存。这些资源然后通过应用程序中的资源管理器分配出去以满足应用程序的需要。因为应用程序同时使用为数不多的不同的顶点跨度（vertex stride），并且只有不同的跨度才需要不同的顶点缓存，所以动态顶点缓存的总数很小。当用这种方式管理动态资源时，需要保证对资源的频繁需求不会严重降低应用程序的性能。当使用同时由 Direct3D 和由应用程序管理的资源时，应该在创建任何由 Direct3D 管理的资源之前，把由应用程序管理的资源分配在 D3DPOOL_DEFAULT 内存中。这使 Direct3D 的内存管理器可以保持对可用内存的精确计数。

状态
Microsoft® Direct3D®设备是一个状态机。应用程序设置光照、渲染和变换模块的状态，然后在渲染时传递数据给它们。
本节描述图形流水线用到的所有不同类型的状态。
渲染状态
取样器状态
纹理层状态
状态块
渲染状态
设备渲染状态控制 Microsoft® Direct3D®设备光栅化模块的行为，它们通过改变渲染状态的属性，使用何种类型的着色算法，雾属性和其它光栅化器操作来达到这个目的。
C++应用程序通过调用IDirect3DDevice9::SetRenderState方法控制渲染状态的属性。
D3DRENDERSTATETYPE枚举类型指定所有可能的渲染状态，应用程序把一个枚举类型值作为第一个参数传递给IDirect3DDevice9::SetRenderState方法。
固定功能顶点处理由 IDirect3DDevice9::SetRenderState 方法和以下设备渲染状态控制。这些控制中的大多数在使用可编程顶点着色器时没有任何作用。
D3DRS_SPECULARENABLE
D3DRS_FOGSTART
D3DRS_FOGEND

D3DRS_FOGDENSITY
D3DRS_RANGEFOGENABLE
D3DRS_LIGHTING
D3DRS_AMBIENT
D3DRS_FOGVERTEXMODE
D3DRS_COLORVERTEX
D3DRS_LOCALVIEWER
D3DRS_NORMALIZENORMALS
D3DRS_DIFFUSEMATERIALSOURCE
D3DRS_SPECULARMATERIALSOURCE
D3DRS_AMBIENTMATERIALSOURCE
D3DRS_EMISSIVEMATERIALSOURCE
D3DRS_VERTEXBLEND
另外，固定功能顶点处理流水线使用以下方法设置变换、材质和光照。
IDirect3DDevice9::SetTransform
IDirect3DDevice9::SetMaterial
IDirect3DDevice9::SetLight
IDirect3DDevice9::LightEnable
注意D3DRS_SPECULARENABLE 控制像素流水线中镜面反射色的加法。D3DRS_FOGSTART，D3DRS_FOGEND 和 D3DRS_FOGDENSITY 控制如何计算雾的起点、终点和像素雾的密度。
更多的信息包含在以下主题中。
概述
阿尔法混合状态
阿尔法测试状态
环境光状态
抗锯齿状态
剔除状态
深度缓存状态
雾状态
光照状态
轮廓和填充状态
每顶点颜色状态
图元裁剪状态
着色状态
模板缓存状态
纹理环绕状态
阿尔法混合状态
一个颜色的阿尔法值控制它的透明度。启用阿尔法混合允许把一个表面上的颜色、材质和纹理根据透明度混合到另一个表面上。
更多信息请参阅阿尔法纹理混合和纹理混合。
C++应用程序使用D3DRS_ALPHABLENDENABLE渲染状态启用阿尔法透明混合。Microsoft® Direct3D® API允许多种类型的阿尔法混合。但是，重要的是要注意用户的三维硬件可能不完全支持所有Direct3D提供的混合状态。
已完成的阿尔法混合的类型取决于 D3DRS_SRCBLEND 和 D3DRS_DESTBLEND 渲染状态。源和目的混

合状态须成对使用。以下示例代码显示了如何将源混合状态设置为 D3DBLEND_SRCCOLOR 并将目的混合状态设置为 D3DBLEND_INVSRCCOLOR。

```
// 本例假设 d3dDevice 为指向 IDirect3DDevice9 接口的有效指针。

// 设置源混合状态。
d3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);

// 设置目的混合状态。
// 设置目的混合状态。
d3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCCOLOR);
```

改变源和目的混合状态可以使物体在雾很浓或灰尘很多的环境中看起来像发光物体。例如，若应用程序在雾很浓的环境中建模了火焰，能量场，离子束或类似的发光体，则可把源和目的混合状态都设置为 D3DBLEND_ONE。

阿尔法混合的另一种应用是控制三维场景中的光照，也称为光照贴图。将源混合状态设置为 D3DBLEND_ZERO 并将目的混合状态设置为 D3DBLEND_SRCALPHA，会根据源的阿尔法信息使场景变暗。源图元被用作光照贴图，对帧缓存中的内容进行缩放，并在适当的时候使之变暗，这就是单色光照贴图。

应用程序可以生成彩色光照贴图，只要把源阿尔法混合状态设置为 D3DBLEND_ZERO，并把目的混合状态设置为 D3DBLEND_SRCCOLOR。

阿尔法测试状态

C++应用程序可以用阿尔法测试控制何时把像素被写入渲染目标表面。通过设置 D3DRS_ALPHATESTENABLE 渲染状态，应用程序让当前的 Direct3D 设备根据阿尔法测试函数测试每个像素。如果测试成功，那么就把像素写入表面。如果不成功，那么 Direct3D 就忽略该像素。应用程序通过 D3DRS_ALPHAFUNC 渲染状态选择阿尔法测试函数。应用程序可以通过 D3DRS_ALPHAREF 渲染状态设置一个参考阿尔法值用来和所有像素进行比较。

阿尔法测试常用于在光栅化几乎透明的物体时提高性能。如果正被光栅化的颜色数据比给定像素更不透明（D3DPCMPCAP_GREATEREQUAL），那么该像素就被写入。否则，光栅化器就完全忽略该像素，这样就节省了将两个颜色混合所需要的处理。以下示例代码检查当前设备是否支持一个给定的比较函数，若支持，则设置比较函数的参数，用来在渲染时提高性能。

```
// 本示例代码假设 pCaps 为一 D3DCAPS9 结构，
// 被先前的一个 IDirect3D9::GetDeviceCaps 调用填充。

if (pCaps.AlphaCmpCaps & D3DPCMPCAPS_GREATEREQUAL)
{
    dev->SetRenderState(D3DRS_ALPHAREF, (DWORD)0x00000001);
    dev->SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
    dev->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);
}

// 如果不支持比较，那么就照常渲染。唯一的缺点是没有性能上的提升。
```

并不是所有的硬件都支持全部阿尔法测试特性。可以通过调用IDirect3D9::GetDeviceCaps方法检查设备能力，取回设备能力后，根据希望使用的比较函数检查相关D3DCAPS9结构的AlphaCmpCaps成员。如果AlphaCmpCaps成员只包含D3DPCMPCAPS_ALWAYS能力或D3DPCMPCAPS_NEVER能力，那么驱动程序不支持阿尔法测试。

环境光状态

环境光是周围的光，从各个方向照射而来。

有关Microsoft® Direct3D®如何使用环境光的信息，请参阅直射光与环境光，和与光照相关的数学。

C++应用程序调用IDirect3DDevice9::SetRenderState方法设置环境光的颜色，并将D3DRS_AMBIENT枚举类型值作为第一个参数传入。第二个参数是颜色值，默认值为零。

```
// 本例假设 d3dDevice 为指向 IDirect3DDevice9 接口的有效指针。

// 设置环境光

d3dDevice->SetRenderState(D3DRS_AMBIENT, 0x00202020);
```

抗锯齿状态

抗锯齿是使屏幕上的线和边缘看起来更为平滑的一种方法。Microsoft® Direct3D®支持两种抗锯齿方法：边缘抗锯齿和全屏抗锯齿。

有关这些技术的更多细节，请参阅抗锯齿。

默认情况下，Direct3D不执行抗锯齿。边缘抗锯齿需要渲染第二遍，要启用边缘抗锯齿，应该把D3DRS_EDGEANTIALIAS渲染状态设置为TRUE，要禁用边缘抗锯齿，应该把D3DRS_EDGEANTIALIAS设置为FALSE。

要启用全屏抗锯齿，应该把 D3DRS_MULTISAMPLEANTIALIAS 渲染状态设置为 TRUE。要禁用全屏抗锯齿，应该把 D3DRS_MULTISAMPLEANTIALIAS 设置为 FALSE。

剔除状态

Direct3D 渲染图元时会剔除背向用户的图元。

C++应用程序使用D3DRS_CULLMODE渲染状态设置剔除模式，它可以被设置为D3DCULL枚举类型的成员。默认情况下，Direct3D把顶点逆时针排列的面作为背向面剔除。

以下示例代码描述了设置剔除模式的过程，这里把顶点顺时针排列的面作为背向面剔除。

```
// 本例假设 d3dDevice 为指向 IDirect3DDevice9 接口的有效指针。
// 设置剔除状态。
d3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
```

深度缓存状态

深度缓存是消除隐藏线和隐藏面的一种方法。默认的情况下，Direct3D 不使用深度缓存。

有关深度缓存的概念，请参阅深度缓存。

C++应用程序用D3DRS_ZENABLE渲染状态更新深度缓存的状态，用D3DZBUFFERTYPE枚举类型成员指定新的状态值。

若应用程序要阻止Direct3D写入深度缓存，则可在调用IDirect3DDevice9::SetRenderState时用D3DRS_ZWRITEENABLE枚举类型作为第一个参数，并将第二个参数指定为D3DZB_FALSE。

以下示例代码显示了如何将深度缓存状态设置为启用 z 缓存。

```
// 本例假设 d3dDevice 为指向 IDirect3DDevice9 接口的有效指针。
// 启用 z 缓存
d3dDevice->SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
```

应用程序也可以使用D3DRS_ZFUNC渲染状态控制Direct3D用于深度缓存的比较函数。

Z偏移是把一个表面显示在另一个表面之前的一种方法，即使它们的深度值相同。可以将此技术用于多种效果。一个常用的例子是在墙上渲染影子，影子和墙具有相同的深度值，但是应用程序希望把影子显示在墙上，只要给影子赋一个z偏移值就可以让Direct3D正确地显示它们（请参阅D3DRS_ZBIAS）。

雾状态

雾效果可以赋予三维场景更强的真实感。雾效果不仅可以用于模拟雾，也能减少远处场景的清晰度。这反映了真实世界中的情况，物体离用户越远，它们的细节就越模糊。

有关如何在应用程序中使用雾的更多信息，请参阅雾。

C++应用程序通过设备渲染状态控制雾。D3DRENDERSTATETYPE枚举类型包括许多状态，控制应用程序使用的是像素（查找表）雾还是顶点雾，雾是何颜色，系统使用的雾的公式，及公式的参数。

可以通过将D3DRS_FOGENABLE渲染状态设置为TRUE启用雾。雾的颜色可以使用D3DRS_FOGCOLOR渲染状态设置为指定颜色，雾的颜色的阿尔法分量被忽略。

D3DRS_FOGTABLEMODE和D3DRS_FOGVERTEXMODE渲染状态控制雾计算使用的公式，它们也间接控制使用何种类型的雾。这两个渲染状态都可以被设置为D3FOGMODE枚举类型的成员。将任何一个渲染状态设置为D3DFOG_NONE相应地禁用像素雾或顶点雾。如果两个渲染状态都被设置为有效的模式，则系统只启用像素雾。

D3DRS_FOGSTART和D3DRS_FOGEND渲染状态控制D3DFOG_LINEAR模式下雾公式的参数。

D3DRS_FOGDENSITY渲染状态控制指数雾模式下雾的密度。

更多信息请参阅雾的参数。

光照状态

使用 Microsoft® Direct3D®几何流水线的应用程序可以启用或禁用光照计算。只有包含顶点法向的顶点才能正确地计算光照，不含法向的顶点在所有光照计算中将使用零点积，因此没有法向的顶点得不到光照。

更多信息请参阅光照的数学。

应用程序通过将D3DRS_LIGHTING渲染状态设置为TRUE启用Direct3D光照，这是默认的设置，应用程序通过将该渲染状态设置为FALSE禁用Direct3D光照。

光照渲染状态与可以对顶点缓存中的顶点执行的光照计算完全无关。在进行顶点处理时IDirect3DDevice9::ProcessVertices方法接收自己的标志用于控制光照计算。

轮廓和填充状态

没有纹理的图元会使用它们的材质指定的颜色进行渲染，或者如果为顶点指定了颜色，就使用顶点色。可以将D3DRS_FILLMODE渲染状态指定为D3DFILLMODE枚举类型的值以选择填充图元的方法。

要启用抖动，应用程序必须把D3DRS_DITHERENABLE枚举类型值作为第一个参数传给IDirect3DDevice9::SetRenderState，并把第二个参数设置为TRUE，把第二个参数设置为FALSE则禁用抖动。

有时，画一条线中的最后一个像素可能导致与周围图元的重叠。可以使用D3DRS_LASTPIXEL枚举值控制这种情况。但是，未经深思熟虑最好不要改变这个设置。在有些情况下，禁止渲染最后一个像素可能会导致图元之间的缝隙。

通过设置适当的画线模式可以画物体的轮廓。默认的画线状态是画实线。更多信息请参阅Direct3D扩展（D3DX）线段绘制渲染状态。

每顶点颜色状态

当使用弹性顶点格式（FVF）编码时，顶点可以同时包含顶点颜色和顶点法向的信息。默认情况下， Microsoft® Direct3D®在计算光照时使用这些信息。要设置是否把顶点颜色用于光照计算，应该调用IDirect3DDevice9::SetRenderState方法，把D3DRS_COLORVERTEX作为第一个参数，把第二个参数设置为FALSE禁用顶点颜色光照，或TRUE启用之。

若每顶点颜色被启用，则应用程序可以配置系统从何处取得源顶点颜色信息。

D3DRS_AMBIENTMATERIALSOURCE，D3DRS_DIFFUSEMATERIALSOURCE，D3DRS_EMISSIVEMATERIALSOURCE和D3DRS_SPECULARMATERIALSOURCE渲染状态控制相应环境反射色、漫反射色、放射色和镜面反射颜色成员的来源。每个状态可以被设置为

D3DMATERIALCOLORSOURCE枚举类型的成员，该枚举类型定义了一些常数，通知系统是使用当前材质的颜色、顶点的漫反射颜色还是顶点的镜面反射颜色作为指定颜色成员的来源。

图元裁剪状态

如果图元的一部分位于视区外，那么Microsoft® Direct3D®可以对此类图元进行裁剪。在C++应用程序中，Direct3D裁剪由D3DRS_CLIPPING渲染状态控制。可以将该渲染状态设置为TRUE（默认值）启用图元裁剪，或将之设置为FALSE禁用Direct3D裁剪服务。

着色状态

Direct3D同时支持平面和高洛德着色，默认情况下使用高洛德着色。要控制当前的着色模式，C++应用程序可以给D3DRS_SHADEMODE渲染状态指定一个D3DSHADEMODE枚举类型值。

以下 C++示例代码显示了把着色状态设置为平面着色模式的过程。

```
// 本例假设 d3dDevice 为指向 IDirect3DDevice9 接口的有效指针。
// 设置着色模式。
d3dDevice->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);
```

模板缓存状态

应用程序使用模板缓存决定是否把一个像素写入到渲染目标表面。

细节请参阅模板缓存技术。

C++应用程序通过调用IDirect3DDevice9::SetRenderState方法启用或禁用模板缓存，应该将D3DRS_STENCILENABLE作为第一个参数的值，并相应地把第二个参数设置为TRUE或FALSE启用或禁用模板缓存。

通过调用IDirect3DDevice9::SetRenderState，可以设置Microsoft® Direct3D®执行模板测试时使用的比较函数，只需将第一个参数的值设置为D3DRS_STENCILFUNC，并把D3DCMPFUNC枚举类型值作为第二个参数的值传入。

模板参考值是在模板缓存中的值，模板函数用它进行测试。默认情况下，模板参考值为零。应用程序可以调用IDirect3DDevice9::SetRenderState设置它的值，只需将D3DRS_STENCILREF作为第一个参数传入，并把第二个参数的值设置为新的参考值。

Direct3D在对每个像素执行模板测试前，会对模板参考值和模板掩码值进行按位与操作，并把得到的结果用模板比较函数与模板缓存的内容进行比较。应用程序可以调用IDirect3DDevice9::SetRenderState设置模板掩码值，只需将D3DRS_STENCILMASK作为第一个参数传入，并把第二个参数设置为新的模板掩码值。

要设置模板测试失败时Direct3D采取的行动，可以调用IDirect3DDevice9::SetRenderState并将D3DRS_STENCILFAIL作为第一个参数传入，第二个参数必须被设为D3DSTENCILCAPS枚举类型的成员。

应用程序也可以控制当模板测试通过但是z缓存测试失败时Direct3D如何响应，只需调用IDirect3DDevice9::SetRenderState，将D3DRS_STENCILZFAIL作为第一个参数传递，并把第二个参数设为D3DSTENCILCAPS枚举类型的成员。

另外，应用程序还可以控制当模板测试和z缓存测试都通过时Direct3D做什么，只需调用IDirect3DDevice9::SetRenderState并将D3DRS_STENCILPASS作为第一个参数，并把第二个参数设为D3DSTENCILCAPS枚举类型的成员。

写掩码可以用于修改将要写入模板缓存的值。要设置模板缓存写掩码，只需调用IDirect3DDevice9::SetRenderState，并将D3DRS_STENCILWRITEMASK作为第一个参数传递，并把第二个参数设为写掩码的值。

纹理环绕状态

D3DRS_WRAP0到D3DRS_WRAP7渲染状态启用或禁用设备的多重纹理级联中各种纹理的u和v环绕。可

以把这些渲染状态设置为标志D3DWRAPCOORD_0，D3DWRAPCOORD_1，D3DWRAPCOORD_2 和 D3DWARPCOORD_3 的组合，相应地启用纹理在第一、第二、第三和第四个方向上的环绕，若使用零值，则禁用所有环绕。默认情况下，所有纹理的所有方向上的纹理环绕都被禁用。
有关的概念综述请参阅纹理环绕。
取样器状态
取样状态控制诸如过滤、tiling 及寻址等与取样有关的操作。
取样状态
SetSamplerState 设置取样器状态（包括 tessellator 中对位移贴图进行取样的状态），为了能够在编译时检测出正在移植 Microsoft® DirectX® 8.x 的应用程序，这些状态已经被重新命名并以 D3DSAMP_前缀开头。这些状态包括：
D3DSAMP_ADDRESSU, D3DSAMP_ADDRESSV, D3DSAMP_ADDRESSW
D3DSAMP_BORDERCOLOR
D3DSAMP_MAGFILTER, D3DSAMP_MINFILTER, D3DSAMP_MIPFILTER
D3DSAMP_MIPMAPLODBIAS
D3DSAMP_MAXMIPLEVEL
D3DSAMP_MAXANISOTROPY
在DirectX 9.0 中，使用像素着色器 2.0 版时，虽然纹理坐标的数量仍被限制为 8 个，但每一趟最多可以支持 16 个纹理表面。有些纹理层状态与表面相关，有些与坐标集相关，有些与顶点处理相关，而有些则与像素处理相关。IDirect3DDevice9::SetSamplerState方法可以指定纹理过滤、平铺、截取、MIPLOD等状态，最多可以有 16 个取样器。
C++应用程序通过调用IDirect3DDevice9::SetSamplerState方法控制与纹理有关的取样状态。
D3DSAMPLERSTATETYPE枚举类型用于指定取样状态。
相关主题
纹理层状态
纹理层状态
纹理层状态控制纹理坐标的生成及纹理坐标的状态，如环绕模式。
C++应用程序通过调用IDirect3DDevice9::SetTextureStageState方法控制与纹理有关的状态。
D3DTEXTURESTAGESTATETYPE枚举类型定义了所有与纹理有关的渲染状态，应用程序应该将 D3DTEXTURESTAGESTATETYPE枚举类型值作为第一个参数传递给 IDirect3DDevice9::SetTextureStageState方法。
应用程序可以通过调用IDirect3DDevice9::SetTexture方法设置某一层的纹理。
SetTextureStageState
SetTextureStageState 现在可以设置以下状态。
固定功能顶点处理状态。这些状态控制对纹理坐标的操控：D3DTSS_TEXTURETRANSFORMFLAGS 和 D3DTSS_TEXCOORDINDEX。最多可以设置八个（因为 Direct3D 总是支持八组纹理坐标）
固定功能像素着色器状态（以前的 TextureStageState）。D3DTSS_COLOROP, D3DTSS_ALPHAOP, D3DTSS_COLORARG0, D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHAARG0, D3DTSS_ALPHAARG1, D3DTSS_ALPHAARG2, D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT01, D3DTSS_BUMPENVMAT10, D3DTSS_BUMPENVMAT11, D3DTSS_BUMPENVLSCALE, D3DTSS_BUMPENVLOFFSET, and D3DTSS_RESULTARG。这些状态最多可有 MaxTextureBlendStages 组可供设置。
D3DTSS_TEXCOORDINDEX 是一个固定功能顶点处理状态。如果使用可编程顶点着色器，那么这个状态被忽略。
应用程序可以使用的纹理取样器的数量由像素着色器的版本决定。

固定功能像素着色器：MaxTextureBlendStages/MaxSimultaneousTextures 个纹理取样器
ps.1.1 到 ps.1.3：4 个纹理取样器。
ps.1.4：6 个纹理取样器。
ps.2.0：16 个纹理取样器。
支持 Microsoft DirectX® 9.0 位移贴图的设备将支持一个额外的取样器（D3DDMAPSAMPLER），它在 tesselator 对位移贴图进行取样。
有关更多纹理混合的信息，请参阅纹理混合。
状态块
一个状态块是一组设备状态、渲染状态、光照和材质的参数、变换状态、纹理层状态和当前纹理信息。状态块是设备当前状态的一个记录，或者是被显式地记录下来的。可以通过单独的一次调用把记录应用于设备。渲染设备可以优化设备状态块以加速应用程序所要求的一般顺序的状态改变。另外，设备状态块也可以使改变设备状态更为方便。
C++应用程序可以在调用IDirect3DDevice9::EndStateBlock方法结束记录一个状态块，以及调用IDirect3DDevice9::CreateStateBlock方法保存一组预定义的设备状态数据时，得到一个状态块句柄。
更多信息包含在以下主题中。
概述
创建预定义的状态块
记录状态块
创建预定义的状态块
IDirect3DDevice9::CreateStateBlock方法创建一个新的状态块，它包含全部的设备状态或是仅与顶点或像素处理相关的设备状态。IDirect3DDevice9::CreateStateBlock方法接收两个参数，第一个参数代表要在新的状态块中记录的状态信息的类型，第二个参数是返回句柄的地址，用于接收当调用成功时返回的有效状态块的句柄。
第一个参数为D3DSTATEBLOCKTYPE枚举类型，可以从以下三种选择：
保存顶点和像素状态。
只保存顶点状态。
只保存像素状态。
要了解可以保存的状态的全部列表，请参阅D3DSTATEBLOCKTYPE。
一定要检查 IDirect3DDevice9::CreateStateBlock 方法的错误代码，如果该方法失败，很可能是显示模式改变了。为了从此类失败中恢复，应用程序应该重新创建表面，然后重新创建状态块。
记录状态块
IDirect3DDevice9接口提供IDirect3DDevice9::BeginStateBlock方法，在应用程序设置设备状态时，该方法会把它们记录到一个状态块中。该方法使系统开始把设备状态的改变记录到一个状态块中，而不是将它们（新的设备状态）应用于设备。可被记录的状态列表请参阅IDirect3DDevice9::BeginStateBlock。
当应用程序完成对状态块的记录后，应该调用IDirect3DDevice9::EndStateBlock方法通知系统以结束记录。该方法返回一个状态块的句柄，应用程序应该将接收状态块句柄的变量的地址作为*pToken*参数传入。应用程序可以根据需要用这个句柄应用该状态块，记录新的状态数据到状态块中，以及在不再需要该状态块时将之删除。
一定要检查 IDirect3DDevice9::EndStateBlock 方法的错误代码，如果该方法失败，很可能是因为显示模式改变了。应该合理地设计应用程序，使之能够从此类失败中恢复，只要重新创建表面并再次记录状态块就可以达到这个目的。

顶点声明

从概念上讲，顶点声明是对顶点直接内存访问（DMA）以及图形流水线的 tessellator 引擎进行编程的一种方法。顶点声明简要地表示了数据的布局及 tessellator 操作。为了解决 Microsoft® DirectX® 8.x 中顶点声明的复杂性和可扩展性，9.0 版引入了用来表示顶点数据流的新格式。

顶点着色器和顶点声明不再是在 CreateVertexShader 的时候绑定在一起。对着色器的验证已经被分成两部分，一部分在顶点着色器创建时执行，另一部分在 DrawPrimitive 时执行。顶点着色器和顶点声明都由相应的对象表示。

Decls 不再用 DWORD 流表示。它们现在用一个 D3DVERTEXELEMENT9 结构的数组表示。数组中的每个元素描述一个顶点元素。

另外，为了解决API的可用性问题，9.0 版增加了一个与SetVertexDeclaration调用等价的SetFVF调用。这是一个有用的函数，当调用这个函数时，新的FVF会取代当前的顶点声明，反之亦然。如果驱动程序是DirectX 8.0 之前的版本（NumStream为 0），那么对于那些不能被转换成弹性顶点格式（FVF）的顶点声明，SetVertexDeclaration可能会失败并返回错误码。SetFVF既能用于固定功能顶点流水线，又能用于可编程顶点流水线。在内部，系统会根据把FVF映射到DirectX 9.0 中的Decl中定义的规则，把FVF码转换为顶点着色器声明。在编写顶点着色器函数中的DCL命令时，应该紧记这一点。因为这个转换的关系，所有后面的讨论将仅限于顶点声明。

D3DVERTEXELEMENT9 结构

以下是 D3DVERTEXELEMENT9 结构中的域以及说明。

Stream（数据流） - 数据流的编号，当前域会从该编号的数据流中读取。

Offset（偏移量） - 偏移量，表示当前域从哪里开始读取，以字节为单位。

Type（类型） - 输入数据的类型，以及数据在传入顶点着色器的寄存器时如何转换格式，也就是，float1,float2,short2n 等等。输出的类型由方法决定，有些方法有隐式的输入类型。

Method（方法） - 将由 tessellator（或任何程序化的几何函数）对指定的输入执行的任何预定义操作。

D3DDECLMETHOD_DEFAULT: 当使用 tessellator 时，这个元素被插值或复制到顶点处理器的输入寄存器中。这个操作的输入可以是任何类型。这个操作的输出类型与输入相同。

D3DDECLMETHOD_PARTIALU: 计算 rectangular patch（R-patch）上某一点在 U 方向上的正切值。这个操作的输入类型可以是 D3DDECLTYPE_FLOAT[43]，D3DDECLTYPE_D3DCOLOR，D3DDECLTYPE_UBYTE4，或 D3DDECLTYPE_SHORT4。这个操作的输出类型总是 D3DDECLTYPE_FLOAT3。

D3DDECLMETHOD_PARTIALV: 计算 R-patch 上某一点在 V 方向上的正切值。这个操作的输入类型可以是 D3DDECLTYPE_FLOAT[43]，D3DDECLTYPE_D3DCOLOR，D3DDECLTYPE_UBYTE4，D3DDECLTYPE_SHORT4。这个操作的输出类型总是 D3DDECLTYPE_FLOAT3。

D3DDECLMETHOD_CROSSUV: 通过求两个正切值的叉积计算 rect/tri patches（RT-patch）上某一点的法向。这个操作的输入类型可以是 D3DDECLTYPE_FLOAT[43]，D3DDECLTYPE_D3DCOLOR，D3DDECLTYPE_UBYTE4，或 D3DDECLTYPE_SHORT4。这个操作的输出类型总是 D3DDECLTYPE_FLOAT3。

D3DDECLMETHOD_UV: 复制 RT-patch 上某一点的 U,V 值，产生一个二维浮点数。这个操作的输入类型必须被设为 D3DDECLTYPE_UNUSED。这个操作的输出类型总是 D3DDECLTYPE_FLOAT2。输入数据流和偏移量也没有用到（但必须被设为 0）。

D3DDECLMETHOD_LOOKUP: 查找一个位移贴图。输入类型可以是 D3DDECLTYPE_FLOAT[234]。只有.x 和.y 成员被用于纹理贴图查找。这个操作的输出类型总是 D3DDECLTYPE_FLOAT4。只有设备支持位移贴图时才能使用这个方法。这个方法只能和 D3DDECLUSAGE_SAMPLE 一起使用。

D3DDECLMETHOD_LOOKUPPRESAMPLED: 查找一个预取样的位移贴图。输入类型、流索引值和流偏移量都没有用到（类型必须被设为 D3DDECLTYPE_UNUSED，数据流和偏移量必须被设为 0）。这个操作的输出类型总是 D3DDECLTYPE_FLOAT4。只有设备支持位移贴图时才能使用这个方法。这个方

法只能和 D3DDECLUSAGE_SAMPLE 一起使用。

Semantic Type（语义类型） - 元素的用途。例如，是否是一个法向量？这对 N-patches 很有用，并能极大地增强各种数据布局和顶点着色器间的互操作性。TEXCOORDS 语义可被用作用户定义的域（Microsoft® Direct3D®没有为这些域定义现成的语义）。一般来说语义是一种把顶点声明和顶点着色器绑定在一起的机制，但是在某种情况下，它们有特殊的解释。例如，N-patch tessellator 用一个含有 NORMAL 和 POSITION 语义的元素来设置 tessellation。

以下这些都是允许的。

语义，用途索引	特殊解释
(D3DDECLUSAGE_POSITION, 0)	固定功能着色器和 N-patch 中未经变换的位置。
(D3DDECLUSAGE_POSITION, 1)	固定功能着色器中用于蒙皮的未经变换的位置。
(D3DDECLUSAGE_BLENDWEIGHT, 0)	在固定功能顶点着色器中的混合加权。
(D3DDECLUSAGE_BLENDINDICES, 0)	固定功能顶点着色器中 indexed paletted skinning 的矩阵索引
(D3DDECLUSAGE_NORMAL, 0)	固定功能顶点着色器和 N-patch tessellator 中的顶点法向
(D3DDECLUSAGE_NORMAL, 1)	固定功能顶点着色器中用于蒙皮的顶点法向。
(D3DDECLUSAGE_PSIZE, 0)	为实现 point sprite 功能，给光栅化器的设置引擎使用的点的大小属性，用来把一个点扩展成一个四边形。
(D3DDECLUSAGE_TEXCOORD, n)	固定功能顶点着色器和 3.0 版以前的像素着色器中的纹理坐标。可以用来传递用户定义的数据。
(D3DDECLUSAGE_TANGENT, n)	正切值。没有特别的解释。
(D3DDECLUSAGE_BINORMAL, n)	副法线（Binormal）。没有特别的解释。
(D3DDECLUSAGE_TESSFACTOR, 0)	在 tessellation 单元中使用的 tessellation 因子，用来控制 tessellation 比率。
(D3DDECLUSAGE_POSITIONT, 0)	变换后的位置。当设置了包含该语义的声明时，顶点处理会被跳过。
(D3DDECLUSAGE_COLOR, 0)	固定功能顶点着色器和 3.0 版以前的像素着色器中的漫反射色。
(D3DDECLUSAGE_COLOR, 1)	固定功能顶点着色器和 3.0 版以前的像素着色器中的镜面反射色。
(D3DDECLUSAGE_FOG, 0)	当设置了 3.0 版以前的像素着色器时，在像素着色器处理之后使用的雾混合值。
(D3DDECLUSAGE_DEPTH, n)	深度。没有特别的解释。
(D3DDECLUSAGE_SAMPLE, n)	查找得到的位移值。只能和 D3DDECLMETHOD_LOOKUP 或 D3DDECLMETHOD_LOOKUPPRESAPLED 一起使用。

有一个特殊的顶点元素值被视为顶点着色器声明的结束。这个值是#define D3DDECL_END() {0xFF,0,0,0,0,0}。

UsageIndex 与 Usage 连用用来指定一个顶点元素的语义。现在用户可以指定 Usage 为 D3DDECLUSAGE_POSITION 以及 UsageIndex = 1，而不是像 Microsoft DirectX® 8.x 中那样使用

D3DVSDE_POSITION2。
一个 Decl 指定 tessellator 引擎（或者万一没有使用 tessellator 引擎的话，就直接是顶点处理引擎，）的输入和输出，顶点元素中的类型域指定输入到 decl 的数据类型。输出类型由方法隐式决定。
顶点着色器中有如下形式的前缀{semantic, semantic index, register number}。运行库会把声明的语义与着色器的语义信息进行匹配,这样就把输入寄存器和数据流中的字节偏移量配成一对。
把 FVF 映射到 DirectX 9.0 的 Decl

FVF	数据类型	用途	用途索引
D3DFVF_XYZ	D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_POSITION	0
D3DFVF_XYZRHW	D3DDECLTYPE_FLOAT4	D3DDECLUSAGE_POSITIONT	0
D3DFVF_XYZW	D3DDECLTYPE_FLOAT4	D3DDECLUSAGE_POSITION	0
D3DFVF_XYZB5 and D3DFVF_LASTBETA_UBYTE4	D3DVSDT_FLOAT3	D3DDECLUSAGE_POSITION	0
	D3DVSDT_FLOAT4	D3DDECLUSAGE_BLENDWEIGHT	
	D3DVSDT_UBYTE4	D3DDECLUSAGE_BLENDINDICES	
D3DFVF_XYZB5 and D3DFVF_LASTBETA_D3DCOLOR	D3DVSDT_FLOAT3	D3DDECLUSAGE_POSITION	0
	D3DVSDT_FLOAT4	D3DDECLUSAGE_BLENDWEIGHT	
	D3DVSDT_D3DCOLOR	D3DDECLUSAGE_BLENDINDICES	
D3DFVF_XYZB5	D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_POSITION	0
	D3DDECLTYPE_FLOAT4	D3DDECLUSAGE_BLENDWEIGHT	
	D3DDECLTYPE_FLOAT1	D3DDECLUSAGE_BLENDINDICES	
D3DFVF_XYZBn (n=1..4)	D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_POSITION	0
	D3DDECLTYPE_FLOATn	D3DDECLUSAGE_BLENDWEIGHT	
D3DFVF_XYZBn (n=1..4) and D3DFVF_LASTBETA_UBYTE4	D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_POSITION	0
	D3DDECLTYPE_FLOAT(n-1)	D3DDECLUSAGE_BLENDWEIGHT	
	D3DDECLTYPE_UBYTE4	D3DDECLUSAGE_BLENDINDICES	
D3DFVF_XYZBn (n=1..4) and D3DFVF_LASTBETA_D3DCOLOR	D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_POSITION	0
	D3DDECLTYPE_FLOAT(n-1)	D3DDECLUSAGE_BLENDWEIGHT	

	D3DDECLTYPE_D3DCOLOR	D3DDECLUSAGE_BLENDINDICES	
D3DFVF_NORMAL	D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_NORMAL	0
D3DFVF_PSIZE	D3DDECLTYPE_FLOAT1	D3DDECLUSAGE_PSIZE	0
D3DFVF_DIFFUSE	D3DDECLTYPE_D3DCOLOR	D3DDECLUSAGE_COLOR	0
D3DFVF_SPECULAR	D3DDECLTYPE_D3DCOLOR	D3DDECLUSAGE_COLOR	1
D3DFVF_TEXCOORDSIZEm(n)	D3DDECLTYPE_FLOATm	D3DDECLUSAGE_TEXCOORD	n

使用了 D3DDECLUSAGE_POSITIONT 的顶点声明

(D3DUSAGE_POSITIONT, 0)顶点元素的存在用来告诉设备输入的顶点数据已经经过了顶点处理（和 FVF 中设置了 D3DFVF_XYZRHW 位相似）。在绘制时，如果当前设置的顶点声明中的某一元素具有(D3DUSAGE_POSITIONT, 0)语义，那么整个顶点处理会被略过（就和 FVF 中设置了 D3DFVF_XYZRHW 位相似）。

使用(D3DDECLUSAGE_POSITIONT, 0)的顶点声明有一些限制：

此类声明中只能使用数据流零。

顶点元素必须以偏移量递增的方式存储。

数据流中的偏移量必须对齐到 DWORD。

一对相同的（用途，用途索引）只能出现一次。

只能使用 D3DDECLMETHOD_DEFAULT 方法。

其余顶点元素不能具有（D3DDECLUSAGE_POSITION, 0）语义。

此外，此类声明还有一些与驱动程序版本有关的限制。这些限制的存在是因为 Direct3D 直接把此类声明传送给驱动程序而没有做任何转换。

DirectX 9.0 之前的驱动程序

输入的声明必须可以被翻译成一个有效的 FVF（顶点元素及其数据类型具有相同的顺序）。

纹理坐标间不允许有间隔。这意味着如果有一个(D3DDECLUSAGE_TEXCOORD, n)顶点元素，那么同时也应该有一个(D3DDECLUSAGE_TEXCOORD, n-1)顶点元素。

不支持 3.0 版本像素着色器的 DirectX 9.0 驱动程序

输入的声明必须可以被翻译成一个有效的 FVF（顶点元素及其数据类型具有相同的顺序）。

纹理坐标间不允许有间隔。

支持 3.0 版本像素着色器的 DirectX 9.0 驱动程序

允许更通用的声明。

顶点元素的顺序可以任意排列，可以使用任何数据类型。

如果设备设置了 D3DDEVCAPS2_VERTEXELEMENTSCANSHARESTREAMOFFSET 能力位，那么多个顶点元素可以在共享相同的数据流偏移量的同时具有不同的类型。

未使用 D3DDECLUSAGE_POSITIONT 的顶点声明

运行库会在创建声明时进行验证。以下是一些通用的规则，用来判断声明的合法性。

数据流中所有的顶点元素必须是连续的，并按偏移量排序。

数据流偏移量必须对齐到 DWORD。

一对相同的（用途，用途索引）只能出现一次。

如果设置了 D3DDEVCAPS2_VERTEXELEMENTSCANSHARESTREAMOFFSET 能力位，那么顶点元素可以：

多个顶点元素可以共享数据流中相同的偏移量。

它们可以是不同的类型，从而有不同的大小。

它们可以任意地重叠。例如，一个元素在数据流中的起始位置可以同时位于另一个元素的中间。

顶点元素的数据流偏移量的顺序可以是任意的。

顶点元素的数量不能大于 64。

UsageIndex 应该在[0-15]范围内。
用于 DrawPrimitive API 的声明，不应该包含除了 D3DDECLMETHOD_DEFAULT, D3DDECLMETHOD_LOOKUPPRESAMPLED 或 D3DDECLMETHOD_LOOKUP 之外的顶点元素。
包含 D3DDECLMETHOD_LOOKUP 或 LOOKUPPRESAMPLED 的顶点声明，应该只用于可编程顶点流水线。
用于 DrawRectPatch/DrawTriPatch API 的声明，不能包含 D3DDECLMETHOD_LOOKUPPRESAMPLED 或 D3DDECLMETHOD_LOOKUP 顶点元素。
声明只能有一个使用 D3DDECLMETHOD_LOOKUP 或 D3DDECLMETHOD_LOOKUPPRESAMPLED 方法的元素。
使用了 D3DDECLMETHOD_LOOKUP 或 D3DDECLMETHOD_LOOKUPPRESAMPLED 的声明不应该使用 D3DDECLMETHOD_DEFAULT 之外的元素，因为位移贴图只对 N-patches 进行。
使用了 D3DDECLMETHOD_LOOKUP 或 D3DDECLMETHOD_LOOKUPPRESAMPLED 的顶点元素只能和 (D3DDECLUSAGE_SAMPLE, n)一起使用，反之亦然。
如果一个使用了 D3DDECLMETHOD_LOOKUP 方法的顶点元素的数据流索引和偏移量与现有的顶点元素相同，那么该顶点元素也应该具有相同的数据类型。
使用了 D3DDECLMETHOD_LOOKUP 方法的顶点元素应该具有 D3DDECLTYPE_FLOAT2/3/4 数据类型。
使用了 D3DDECLMETHOD_CROSSUV, D3DDECLMETHOD_PARTIALU 和 D3DDECLMETHOD_PARTIALV 的顶点元素应该具有兼容的数据类型。
使用了 D3DDECLMETHOD_UV 或 D3DDECLMETHOD_LOOKUPPRESAMPLED 的顶点元素的类型必须是 D3DDECLTYPE_UNUSED，数据流索引和偏移量必须为 0。
使用了 D3DDECLMETHOD_UV, D3DDECLMETHOD_PARTIALU 和 D3DDECLMETHOD_PARTIALV 方法的声明只能用于 DrawRectPath。
用途 D3DDECLUSAGE_TESSFACTOR 只能和数据类型 D3DDECLTYPE_FLOAT1 及用途索引 0 一起使用。
把一个声明用于 tessellation（DrawRectPatch, DrawTriPatch, N-patches）时，数据类型必须小于或等于 D3DDECLTYPE_SHORT4。
如果声明包含的方法需要特定的设备能力（例如，位移贴图，RT-patches），那么只有当设备支持这些能力时才能创建。
用于绘制点和线的顶点声明，不能使用除了 D3DDECLMETHOD_DEFAULT 之外的方法。
可以创建哪些声明还取决于驱动程序的能力。
用于可编程顶点流水线
在绘制时 Direct3D 会在当前顶点声明和当前顶点着色器函数中查找相同的“用途 - 用途索引”组合。如果找到，那么着色器函数 DCL 中的寄存器会被用作顶点元素的目的寄存器。
如果当前顶点声明中的某个顶点元素的用途未能在当前顶点着色器中找到，那么该顶点元素就被忽略。
当使用版本 2.0 以下的顶点着色器时，在着色器代码中用到的所有语义，必须存在于在绘制时与之绑定的声明中。当使用版本 2.0 及以上的顶点着色器时，不存在这种限制，这使应用程序可以把同一个顶点着色器和不同的顶点声明一起使用。当顶点着色器根据静态条件读取输入数据时，这是有用的。因为这个原因而未被初始化的顶点着色器寄存器会包含未定义的值。
当在一个 DirectX 8.x 驱动程序上使用硬件顶点处理时，还有其它的限制：
顶点元素不能重叠或共享相同的偏移量。
只能使用 DirectX 8.x 驱动程序能够理解的数据类型。
如果提供的声明无法被转换为 DirectX 8.x 风格的声明，那么 CreateVertexDeclaration 可能会失败。对混合模式设备来说，这种失败会发生在 Draw*时刻，因为只有这时候才能知道着色器是被用于硬件还是软件顶点处理。
用于固定功能顶点流水线
只能使用符合以下规则的声明：

顶点元素之间不能有间隔（固定功能流水线使用了 DirectX 8.x 声明，而 SKIP 在 DirectX 8.x 声明中是不允许的）
必须指定(D3DDECLUSAGE_POSITION, 0)语义，并且具有 D3DDECLTYPE_FLOAT3 数据类型。
使用了 D3DDECLMETHOD_UV 方法的顶点元素必须指定 D3DDECLUSAGE_TEXCOORD 或 D3DDECLUSAGE_BLENDWEIGHT 用途。
使用了 D3DDECLMETHOD_PARTIALU, D3DDECLMETHOD_PARTIALV 或 D3DDECLMETHOD_CROSSUV 的顶点元素只能和这些用途一起使用：D3DDECLUSAGE_POSITION, D3DDECLUSAGE_NORMAL, D3DDECLUSAGE_BLENDWIEGHT 或 D3DDECLUSAGE_TEXCOORD，并且输入类型必须是 D3DDECLTYPE_FLOAT3。
当声明和 DirectX 8.x 驱动程序的硬件顶点处理一起使用时，Direct3D 运行库会根据以下规则把它转换为 DirectX 8.x 风格的声明。
顶点元素不能共享数据流中相同的偏移量，并且不能重叠。
数据类型必须小于或等于 D3DDECLTYPE_SHORT4。
只允许以下这些方法：D3DDECLMETHOD_DEFAULT, D3DDECLMETHOD_CROSSUV, D3DDECLMETHOD_UV。
根据把DirectX 9.0 声明映射到DirectX 8.x声明把Usage和UsageIndex转换到寄存器值。
把DirectX 9.0 声明映射到DirectX 8.x声明列出了哪些DirectX 9.0 语义可以被转换到DirectX 8.x风格的声明。
如果声明中有n个顶点元素，其中元素 0 - m (m < n)可以映射到FVF（把DirectX 9.0 声明映射到FVF位中描述的元素），但元素m+1 不可以，那么：
如果 m+2 到 n-1 之间的任何一个顶点元素可以映射到 FVF/dx8decl，那么声明就是无效的。
根据把DirectX 9.0 声明映射到FVF位，对元素 0 到m进行转换（由DirectX 8.0 运行库和旧的驱动程序，以及DirectX 9.0 驱动程序执行）。m+1, m+2 直到n-1 被映射到连续的texcoord(k), texcoord(k+1)，k为元素 0 - m之间的texcoord（的数量）。
映射后的texcoord的数据类型会是float[1234]，取代当前元素原来的数据类型（但大小是相同的）。因此即使现有的数据类型没有被包括在把DirectX 9.0 声明映射到FVF位中，也没有关系。
如果 k 到达 8，那么声明对 FVF/dx8decl 来说就是无效的。
如果存在任何到 texcoord 的映射，那么除了 DEFAULT 方法，整个声明不能包含使用（纹理）生成方法的元素，否则声明对 FVF/dx8decl 来说就是无效的。
把 DirectX 9.0 声明映射到 DirectX 8.x 声明

DirectX 9.0 用途	DirectX 9.0 用途索引	DirectX 8.0
D3DDECLUSAGE_POSITION	0	D3DVSDE_POSITION
D3DDECLUSAGE_POSITION	1	D3DVSDE_POSITION2
D3DDECLUSAGE_NORMAL	0	D3DVSDE_NORMAL
D3DDECLUSAGE_NORMAL	1	D3DVSDE_NORMAL2
D3DDECLUSAGE_BLENDWEIGHT	0	D3DVSDE_BLENDWEIGHT
D3DDECLUSAGE_BLENDINDICES	0	D3DVSDE_BLENDINDICES
D3DDECLUSAGE_PSIZE	0	D3DVSDE_PSIZE
D3DDECLUSAGE_COLOR	0	D3DVSDE_DIFFUSE
D3DDECLUSAGE_COLOR	1	D3DVSDE_SPECULAR
D3DDECLUSAGE_TEXCOORD	n	D3DVSDE_TEXCOORDn

当声明和 DirectX 7.0 驱动程序上的硬件顶点处理一起使用时，Direct3D 运行库会根据以下规则把它转换为 FVF。

只能使用数据流 0（显然可以从 MaxStreams 设备能力中看出）。
顶点元素的顺序应该和 FVF 位的顺序相同。
不允许纹理坐标间有间隔。
对所有DirectX 8.0 之前的驱动程序来说，任何没有在把DirectX 9.0 声明映射到FVF位中描述的顶点元素不能被转换为有效的FVF码。因此无法在这些驱动程序中使用。
如果设备没有设置以下这些能力位：D3DPTEXTURECAPS_PROJECTED 或 D3DPTEXTURECAPS_CUBEMAP，那么使用了 D3DDECLUSAGE_TEXCOORD 的顶点元素只能用 D3DDECLTYPE_FLOAT2。
把 DirectX 9.0 声明映射到 FVF 位

数据类型	用途	用途索引	FVF
D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_POSITION	0	D3DFVF_XYZ
D3DDECLTYPE_FLOAT4	D3DDECLUSAGE_POSITIONT	0	D3DFVF_XYZRHW
D3DDECLTYPE_FLOATn	D3DDECLUSAGE_BLENDWEIGHT	0	D3DFVF_XYZBn
D3DDECLTYPE_UBYTE4	D3DDECLUSAGE_BLENDINDICES	0	D3DFVF_XYZB (nWeights+1)
D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_NORMAL	0	D3DFVF_NORMAL
D3DDECLTYPE_FLOAT1	D3DDECLUSAGE_PSIZE	0	D3DFVF_PSIZE
D3DDECLTYPE_D3DCOLOR	D3DDECLUSAGE_COLOR	0	D3DFVF_DIFFUSE
D3DDECLTYPE_D3DCOLOR	D3DDECLUSAGE_COLOR	1	D3DFVF_SPECULAR
D3DDECLTYPE_FLOATm	D3DDECLUSAGE_TEXCOORD	n	D3DFVF_TEXCOORDSIZEm(n)
D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_POSITION	1	N/A
D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_NORMAL	1	N/A

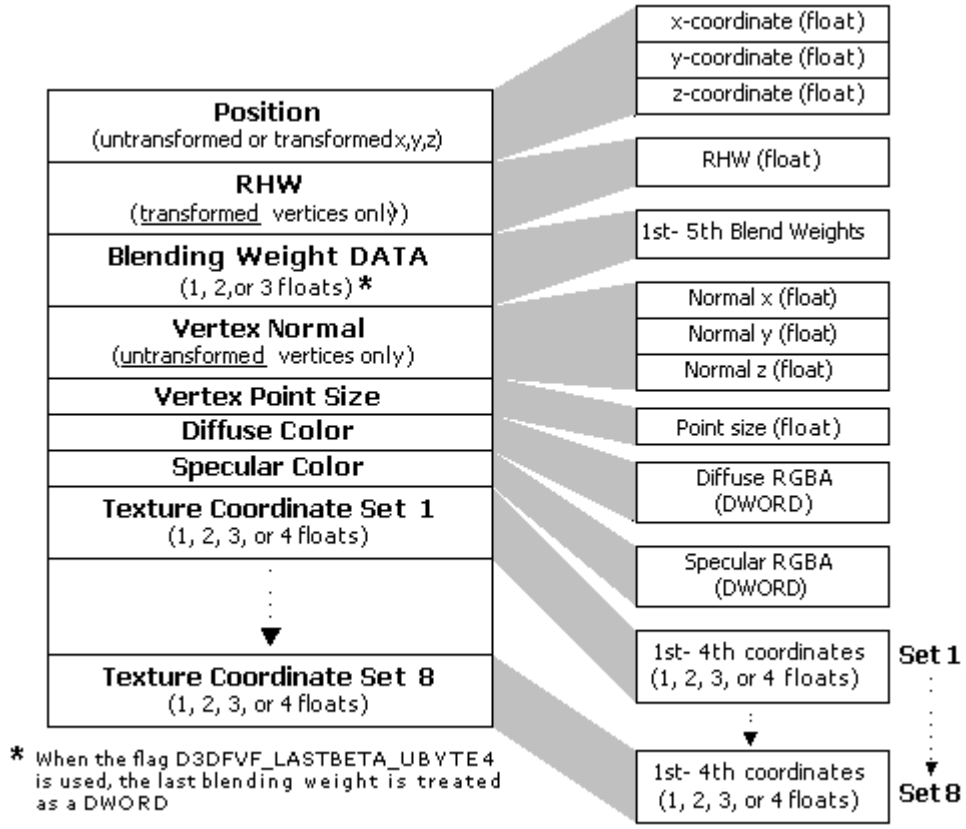
用于 ProcessVertices 的顶点声明
顶点声明可以被用来描述 ProcessVertices 的输出。此类声明应该遵循以下规则。
只能而且必须使用数据流 0。
顶点元素不能共享相同的偏移量或相互重叠。
只允许使用 D3DDECLMETHOD_DEFAULT 方法。
只能使用 D3DDECLTYPE_FLOATn 或 D3DDECLTYPE_D3DCOLOR 数据类型。
使用(D3DDECLUSAGE_POSITION, 0)或(D3DDECLUSAGE_POSITIONT, 0)的顶点元素不是必需的。
使用(D3DDECLUSAGE_POSITION, 0)和(D3DDECLUSAGE_POSITIONT, 0)的顶点元素不能在同一个声明中出现。
当使用此类声明时，当前顶点着色器的必须是 3.0 及以上版本。

顶点格式
弹性顶点格式（FVF）码描述了交叉存储在单个数据流中顶点的内容。它通常说明了要被固定功能顶点处理流水线处理的数据。
Microsoft® Direct3D®应用程序能以几种不同的方式定义建模的顶点。对弹性顶点定义（也叫弹性顶点格式或弹性顶点格式码）的支持使用应用程序只使用必需的顶点成员成为可能，这样就消除了无用的成员。通过只使用必需的顶点成员，应用程序可以节省内存并最小化渲染建模需要的处理带宽。应用程序通过使用D3DFVF的组合来定义顶点格式。
FVF 规范包含了点的大小的格式，由 D3DFVF_PSIZE 指定。对非 TL 顶点而言，这个大小表示的是摄像机空间中的单位，对 TL 顶点而言，则是设备空间中的单位。
IDirect3DDevice9接口给C++应用程序提供了一些方法，可以接收这些标志的组合，并使用它们决定如何渲染图元。基本上，这些标志告诉系统应用程序使用哪些顶点成员——位置，顶点混合

加权值，法向，颜色，纹理坐标的数量和格式——并且间接地说明要使用渲染流水线中的哪些部分。另外，某个特定顶点格式标志的存在与否会通知系统哪些顶点成员域存在于内存中，而哪些被省略了。

要测定设备的限制，可以查询设备是否具有D3DFVFCAPS_DONOTSTRIPLELEMENTS和D3DVFVCAPS_TEXCOORDCOUNTMASK弹性顶点格式标志。更多信息，请参阅D3DCAPS9结构的FVFCaps成员

系统对如何安排顶点有一个重要规定，那就是数据出现的顺序。下图描绘了所有可能的顶点成员在内存中规定的顺序，以及它们相应的数据类型。



纹理坐标可以被声明为不同的格式，这使纹理可以用最少一个坐标或最多四个坐标（二维投影纹理坐标）进行寻址。更多信息，请参阅纹理坐标格式。应该用D3DFVF_TEXCOORDSIZEn系列宏创建应用程序使用的顶点格式中的纹理坐标格式。

没有哪个应用程序会使用每个成员——齐次坐标 W 的倒数（RHW）和顶点法向这两个域是互斥的。大多数应用程序也不会使用所有八组纹理坐标，但 Direct3D 具有这种能力。对于应用程序如何使用标志有一些限制。例如，不能同时使用 D3DFVF_XYZ 和 D3DFVF_XYZRHW 标志，同时使用意味着应用程序描述的是一个既未经变换又经过变换的顶点位置。

要使用索引顶点混合，D3DFVF_LASTBETA_UBYTE4 标志应该出现在FVF的最后。这个标志的出现表示第五个混合加权值将被解释为DWORD类型而非浮点数。更多信息，请参阅索引顶点混合。

以下示例代码显示了使用和不使用 D3DFVF_LASTBETA_UBYTE4 标志的 FVF 码之间的区别。下面定义的 FVF 码没有使用 D3DFVF_LASTBETA_UBYTE4 标志。当使用四个混合索引值时，会用到 D3DFVF_XYZB3 标志，这是因为应用程序使用的第四个值总是等于（1 - 前三个的和）。

```
#define D3DFVF_BLENDVERTEX (D3DFVF_XYZB3|D3DFVF_NORMAL|D3DFVF_TEX1)

struct BLENDVERTEX
```

```
{
    D3DXVECTOR3 v;        // 被作为顶点着色器中的 v0 引用
    FLOAT       blend1;   // 被作为顶点着色器中的 v1.x 引用
    FLOAT       blend2;   // 被作为顶点着色器中的 v1.y 引用
    FLOAT       blend3;   // 被作为顶点着色器中的 v1.z 引用
                          // v1.w = 1.0 - (v1.x + v1.y + v1.z)
    D3DXVECTOR3 n;        // 被作为顶点着色器中的 v3 引用
    FLOAT       tu, tv;   // 被作为顶点着色器中的 v7 引用
};
```

下面定义的 FVF 码使用了 D3DFVF_LASTBETA_UBYTE4 标志。

```
#define D3DFVF_BLENDVERTEX (D3DFVF_XYZB4 | D3DFVF_LASTBETA_UBYTE4
|D3DFVF_NORMAL|D3DFVF_TEX1)

struct BLENDVERTEX
{
    D3DXVECTOR3 v;        // 被作为顶点着色器中的 v0 引用
    FLOAT       blend1;   // 被作为顶点着色器中的 v1.x 引用
    FLOAT       blend2;   // 被作为顶点着色器中的 v1.y 引用
    FLOAT       blend3;   // 被作为顶点着色器中的 v1.z 引用
                          // v1.w = 1.0 - (v1.x + v1.y + v1.z)
    DWORD       indices;  // 被作为顶点着色器中的 v2.xyzw 引用
    D3DXVECTOR3 n;        // 被作为顶点着色器中的 v3 引用
    FLOAT       tu, tv;   // 被作为顶点着色器中的 v7 引用
};
```

几何体

Microsoft® Direct3D®支持几种处理几何体的高级抽象。它们被分为以下主题。

顶点缓存

顶点缓存，由IDirect3DVertexBuffer9接口表示，是包含顶点数据的内存缓冲区。顶点缓存可以包含任何顶点类型——经过变换的或未经变换的，经过光照处理或未经光照处理的——它们可以使用IDirect3DDevice9接口的渲染方法进行渲染。应用程序可以在顶点缓存中处理顶点，执行诸如变换、光照或生成裁剪标志的操作。变换操作总是被执行。

顶点缓存的灵活性使之成为重用经过变换的几何体的完美中转站。可以创建单个顶点缓存，在其中执行变换、光照和裁剪操作，并根据需要在场景中多次渲染该建模，即使在这之间渲染状态变了，无需重新进行变换。这在渲染使用多重纹理的建模时非常有用：几何体只被变换一次，然后可以根据需要渲染各个部分，如果有必要，期间可以改变纹理。顶点被处理后渲染状态的改变会在下一次顶点被处理时生效。

描述

顶点缓存由它的能力描述：是否只能存在于系统内存中，是否只能用于写操作，以及所能包含的顶点的类型和数量。所有这些特性都保存在D3DVERTEXBUFFER_DESC结构中。

为了表明是一个顶点缓存，Format 成员要被设为 D3DFMT_VERTEXDATA。Type 成员表示顶点缓存的资源类型。Usage 成员包含了综合能力标志。D3DUSAGE_SOFTWAREPROCESSING 标志表示顶点缓存被用于软件顶点处理。D3DUSAGE_WRITEONLY 标志表示顶点缓存只用于写操作。这使驱动程序可以自由地将顶点数据放到最合适的内存位置并进行快速处理和渲染。如果没有使用 D3DUSAGE_WRITEONLY 标志，驱动程序不太可能将数据放在读操作效率很低的地方。这牺牲了一些处理时间和渲染速度。如果没有指定该标志，那么系统会假定应用程序将对顶点缓存中的数据执行读写操作。

Pool 成员指定为顶点缓存分配内存的类型。D3DPOOL_SYSTEMMEM 标志指示系统在系统内存中创建顶点缓存。

Size成员存储了顶点缓存数据的大小，以字节为单位。FVF成员包含了D3DFVF的组合，用来识别缓存包含顶点的类型。

内存池及用途

应用程序可以使用IDirect3DDevice9::CreateVertexBuffer方法创建顶点缓存，该方法接收池（内存类型）和用途参数。IDirect3DDevice9::CreateVertexBuffer也可以用一个指定的FVF码创建顶点缓存，以用于固定功能顶点处理或作为顶点处理的输出。更多细节，请参阅FVF顶点缓存。

当在设备上启用了混合模式或软件顶点处理（D3DCREATE_MIXED_VERTEXPROCESSING / D3DCREATE_SOFTWARE_VERTEXPROCESSING）时，可以设置 D3DUSAGE_SOFTWAREPROCESSING 标志。在混合模式中，要进行软件顶点处理的缓存必须设置 D3DUSAGE_SOFTWAREPRECESSING 标志，但在使用硬件顶点处理（D3DCREATE_HARDWARE_VERTEXPROCESSING）时，为了得到尽可能好的性能，不应设置该标志。但是，当一个缓存要同时进行硬件和软件顶点处理时，设置 D3DUSAGE_SOFTWAREPROCESSING 是唯一的选择。和软件设备一样，混合设备也可以使用 D3DUSAGE_SOFTWAREPROCESSING。

即使顶点处理是用硬件完成的，通过指定 D3DPOOL_SYSTEMMEM，把顶点和索引缓存强制分配到系统内存中也是可以的。当驱动程序将这些缓存放在 AGP 内存中时，这种方法可以避免大量的页面锁定内存。

本节介绍了一些概念，这些概念是理解顶点缓存以及在 Microsoft® Direct3D®应用程序中使用顶点缓存所必需了解的。信息被分为以下部分。

创建顶点缓存

应用程序应该通过调用IDirect3DDevice9::CreateVertexBuffer方法创建一个顶点缓存对象，该方法接收五个参数。第一个参数指定缓存的长度，以字节为单位。可以用sizeof操作符确定某一顶点格式的大小。考虑以下自定义顶点格式。

```
struct CUSTOMVERTEX {
        FLOAT x, y, z;
        FLOAT rhw;
        DWORD color;
        FLOAT tu, tv;   // 纹理坐标
```

```
};

// 自定义 FVF，描述了自定义顶点的结构

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW | D3DFVF_DIFFUSE | D3DFVF_TEX1)
```

要创建一个可以保存四个 CUSTOMVERTEX 结构的顶点缓存，应该把 *Length* 参数指定为 [4*sizeof(CUSTOMVERTEX)]。

第二个参数是一组用途控制标志。该参数的值特别用来决定对那些位于视区外的顶点，顶点缓存是否能包含裁剪信息——以裁剪标志的形式。要创建一个不能包含裁剪标志的顶点缓存，应该在 *Usage* 参数中包含 D3DUSAGE_DONOTCLIP 标志。只有在应用程序指明顶点缓存将包含经过变换的顶点——*FVF* 参数包含 D3DFVF_XYZRHW 标志——的同时，D3DUSAGE_DONOTCLIP 标志才有用。如果应用程序指明缓存将包含未经变换的顶点（D3DFVF_XYZ 标志），那么 IDirect3DDevice9::CreateVertexBuffer 方法会忽略 D3DUSAGE_DONOTCLIP 标志。裁剪标志会占用额外的内存，这使可裁剪的顶点缓存比不能包含裁剪标志的顶点缓存略大。因为这些资源是在创建顶点缓存时分配的，所以必须提前要求可裁剪的顶点缓存。

第三个参数，*FVF*，是描述顶点缓存的顶点格式的D3DFVF组合。如果把这个参数指定为 0，那么顶点缓存就是一个非FVF顶点缓存。更多信息，请参阅FVF顶点缓存。第四个参数描述用来存放顶点缓存的内存类型。

IDirect3DDevice9::CreateVertexBuffer接收的最后一个参数是变量的地址，如果调用成功，它会把一个指向新的IDirect3DVertexBuffer9顶点缓存对象接口的指针填入该变量。

注意 如果应用程序在创建顶点缓存时不支持裁剪标志，那么就不能生成裁剪标志。

以下 C++示例代码显示了如何在代码中创建一个顶点缓存。

```
    // 根据本示例的目的，d3dDevice 变量是一个由 Direct3DDevice 对象暴露的
    // IDirect3DDevice9 接口的地址，g_pVB 是一个 LPDIRECT3DVERTEXBUFFER9 类型的变量。

// 自定义顶点类型
struct CUSTOMVERTEX {
        FLOAT x, y, z;
        FLOAT rhw;
        DWORD color;
        FLOAT tu, tv;   // 纹理坐标
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW | D3DFVF_DIFFUSE | D3DFVF_TEX1)

    // 创建一个可裁剪的顶点缓存。在默认的内存池中分配足够的内存
    // 以保存三个 CUSTOMVERTEX 结构。

    if( FAILED( d3dDevice->CreateVertexBuffer( 3*sizeof(CUSTOMVERTEX),
                                               0 /* 用途 */, D3DFVF_CUSTOMVERTEX,
                                               D3DPOOL_DEFAULT, &g_pVB ) ) )
        return E_FAIL;
```

存取顶点缓存的内容

顶点缓存对象使应用程序能直接存取为顶点数据分配的内存。应用程序可以通过调用

IDirect3DVertexBuffer9::Lock方法得到一个指向顶点缓存内存的指针，然后根据需要存取内存并将新的顶点数据填入缓存，或读取已经包含的数据。IDirect3DVertexBuffer9::Lock方法接收四个参数。第一个参数*OffsetToLock*，是顶点数据中的偏移量。第二个参数是顶点数据的大小，以字节为单位。第三个参数*ppbData*，是一个字节指针的地址，如果调用成功的话，该指针将会指向顶点数据。

最后一个参数*Flags*，告诉系统如何锁定内存。应用程序应该根据顶点数据将被存取的方式给*Flags*参数指定相应的常量。应该确保D3DUSAGE的值与D3DLOCK的值相对应。例如，如果正在创建一个只写的顶点缓存，那么试图通过指定D3DLOCK_READONLY标志去读取数据就没有意义。根据要求的存取类型，合理地使用这些标志允许驱动程序锁定内存并提供最佳的性能。

完成写入或读取顶点数据后，应该调用IDirect3DVertexBuffer9::Unlock方法，如以下示例代码所示。

```
// 本示例代码假定 g_pVB 为 LPDIRECT3DVERTEXBUFFER9 类型的变量，
// 并且 g_Vertices 已经用顶点数据做了适当的初始化。

// 要填充顶点缓存，必须锁定缓存以取得对顶点的存取权。
// 这种机制是必要的，因为顶点缓存可能在设备内存中。
VOID* pVertices;

if( FAILED( g_pVB->Lock( 0,                 // 从缓存的开始处填充。
                         sizeof(g_Vertices), // 要载入数据的大小。
                         (BYTE**)&pVertices, // 返回的顶点数据。
                         0 ) ) )             // 设置默认的锁定标志位。
    return E_FAIL;

memcpy( pVertices, g_Vertices, sizeof(g_Vertices) );
g_pVB->Unlock();
```

注意 如果应用程序用 D3DUSAGE_WRITEONLY 标志创建一个顶点缓存，那么不要用 D3DLOCK_READONLY 锁定标志。如果应用程序只从顶点缓存中读取数据，那么应该使用 D3DLOCK_READONLY 标志。如果对内存的存取是只读的，那么包含这个标志使 Microsoft® Direct3D®能够优化它的内部处理并提高效率。

有关把D3DLOCK_DISCARD或D3DLOCK_NOOVERWRITE标志作为IDirect3DVertexBuffer9::Lock方法的*Flags*参数的更多信息，请参阅使用动态顶点和索引缓存。

因为C++应用程序直接存取为顶点缓存分配的内存，所以要确保应用程序正确地存取分配的内存。否则，存在渲染无效内存的风险。应用程序在从分配的缓存中的一个顶点移到另一个时，应该使用顶点格式的步距（stride）。顶点缓存内存是一个在弹性顶点格式中指定的简单的顶点数组。对于应用程序定义的任何顶点格式结构，都应使用步距。应用程序可以通过在运行的时候检查顶点缓存描述中包含的D3DFVF来计算步距。下表显示了每个顶点成员（vertex component）的大小。

顶点格式标志	大小
D3DFVF_DIFFUSE	sizeof(DWORD)
D3DFVF_NORMAL	sizeof(float) × 3
D3DFVF_SPECULAR	sizeof(DWORD)

D3DFVF_TEX*n*　　sizeof(float) × *n* × t

D3DFVF_XYZ　　sizeof(float) × 3

D3DFVF_XYZRHW　sizeof(float) × 4

顶点格式中出现的纹理坐标的数量由 D3DFVF_TEX*n* 标志描述（*n* 为从 0 到 8 的值）。要计算某个数量的纹理坐标所需要的内存，应该将纹理坐标组的数量乘以每组纹理坐标的大小，每组纹理坐标的大小可以是一到四个浮点数。

要存取某个顶点，应该使用整个顶点的步距增大或减小内存指针。

取得顶点缓存描述

应用程序可以通过调用IDirect3DVertexBuffer9::GetDesc方法取得关于一个顶点缓存的信息。这个方法会把关于顶点缓存的信息填入D3DVERTEXBUFFER_DESC结构的成员。

用顶点缓存进行渲染

用顶点缓存渲染顶点数据需要几个步骤。首先，应用程序需要通过调用 IDirect3DDevice9::SetStreamSource方法设置数据流的源，如下示例代码所示。

```
d3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
```

IDirect3DDevice9::SetStreamSource 方法的第一个参数告诉Microsoft® Direct3D®设备数据流的源。第二个参数是要绑定到数据流的顶点缓存。第三个参数是成员的大小，以字节为单位。在上面的示例代码中，使用了 CUSTOMVERTEX 的大小作为成员的大小。

下一步是通过调用IDirect3DDevice9::SetVertexShader方法告诉Direct3D使用哪个顶点着色器。以下示例代码给顶点着色器设置了一个弹性顶点格式（FVF）码。这告诉Direct3D正在处理的顶点类型。

```
d3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
```

在设置数据流的源和顶点着色器之后，任何绘制方法将使用设定的顶点缓存。以下示例代码显示了如何用IDirect3DDevice9::DrawPrimitive方法用顶点缓存渲染顶点。

```
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
```

IDirect3DDevice9::DrawPrimitive 方法接收的第二个参数是顶点缓存中要载入的第一个顶点的索引值。

FVF 顶点缓存

把IDirect3DDevice9::CreateVertexBuffer方法的弹性顶点格式（FVF）参数设置为非零值，但必须是有效的FVF码，表示缓存的内容会通过FVF码描述。用一个FVF码创建的顶点缓存被称为FVF顶点缓存。一些IDirect3DDevice9的方法或用法需要FVF顶点缓存，而另一些需要非FVF顶点缓存。IDirect3DDevice9::ProcessVertices方法需要FVF顶点缓存作为目标顶点缓存参数。

FVF 顶点缓存可以被绑定到任意一个编号的源数据流。但是，不允许把 FVF 顶点缓存用作可编程顶点着色器的输入。

FVF 顶点缓存中存在 D3DFVF_XYZRHW 成员表示缓存中的顶点已经处理过了。用作 IDirect3DDevice9::ProcessVertices 目标顶点缓存的顶点缓存必须是处理后的。用作固定功能着色器的输入的顶点缓存既可以是处理前的也可以是处理后的。如果顶点缓存是处理后的，那么着色器实际上被略过，数据被直接送到图元裁剪和设置模块。

FVF 顶点缓存可用于顶点着色器。同样，顶点流可以表示和非 FVF 顶点缓存相同的顶点格式。它们不一定要用于从单独的顶点缓存中输入数据。新的顶点流的额外灵活性使需要分散保存它们数据的应用程序工作得更好，但这不是必须的。如果应用程序可以预先维持交叉存取的数据，那么这会导致性能的提升。如果应用程序只会在每次渲染调用前交叉存取数据，那么它应该启用 API 或硬件，用多数据流完成这类操作。

就顶点性能而言，最重要的是要使用 32 字节顶点，并维持好的高速缓存排序。

固定功能顶点处理

在固定功能顶点流水线中，处理顶点缓存中的顶点会应用设备当前的变换矩阵。另外也可以有选择地执行诸如光照、生成裁剪标志及更新区域（extents）之类的操作。在使用固定功能顶点处理时，对目标顶点缓存中的元素的修改由D3DPV_DONOTCOPYDATA标志控制。这个标志仅用于固定功能顶点处理。IDirect3DDevice9接口暴露了用于处理顶点的IDirect3DDevice9::ProcessVertices接口。通过调用IDirect3DDevice9::ProcessVertices方法，应用程序可以将顶点从一个顶点着色器处理为一个输入数据的集合，并生成一个单独的交叉存取的顶点缓存到目标顶点缓存。该方法接收五个（译注：应该是六个）参数，分别描述要处理的顶点的位置和数量，目标顶点缓存，及处理选项。在调用该方法后，目标缓存会包含经过处理的顶点数据。

第一个参数*SrcStartIndex*，为要载入的第一个顶点的索引值，同时也指定了该方法应该从哪里开始处理顶点。第二个参数*DestIndex*，为一个索引值，指定把顶点放在目标缓存中的何处。第三个参数*VertexCount*，为要处理并放在目标缓存中的顶点的全部数量。应该把第四个参数*pDestBuffer*设为IDirect3DVertexBuffer9顶点缓存对象接口的地址，该顶点缓存对象用来存放处理后的源顶点。

最后一个参数*Flags*，决定该方法的特殊处理选项。可以把这个参数设为 0 进行默认的顶点处理，或设为D3DPV_DONOTCOPYDATA以便在某种情况下对处理进行优化。当应用程序把*Flags*设为 0 时，目标顶点缓存的顶点格式中不受顶点处理影响的顶点成员仍然会从顶点着色器中复制或设为 0。但是，当使用了D3DPV_DONOTCOPYDATA时，IDirect3DDevice0::ProcessVertices不会覆盖目标缓存中的颜色和纹理坐标信息，除非数据是由Microsoft® Direct3D®生成的。漫反射色在光照被启用时产生，也就是说，D3DRS_LIGHTING被设为TRUE。镜面反射色在光照被启用且镜面反射也被启用时产生，也就是说，D3DRS_SPECULARENABLE和D3DRS_LIGHTING被设为TRUE。镜面反射色也在雾被启用时产生。纹理坐标在纹理变换和纹理生成被启用时产生。

IDirect3DDevice9::ProcessVertices使用当前的渲染状态决定应该进行哪些顶点处理。

目标顶点缓存的 FVF 用途设置

IDirect3DDevice9::ProcessVertices方法需要对目标顶点缓存的D3DFVF进行特殊设置。弹性顶点格式（FVF）用途设置与当前顶点处理的设置必须不能相矛盾。

对于固定功能顶点处理，IDirect3DDevice9::ProcessVertices 需要以下 FVF 设置。

位置类型总是 D3DFVF_XYZRHW；因此，D3DFVF_XYZ 和 D3DFVF_XYZB1 到 D3DFVF_XYZB5 是无效的。

必须没有设置 D3DFVF_NORMAL，D3DFVF_RESERVED0，和 D3DFVF_RESERVED2 标志。

若以下条件的或操作返回真，则 D3DFVF_DIFFUSE 标志必须被设置。

光照被启用，也就是说，D3DRS_LIGHTING为真。

光照被禁用，同时输入顶点流中存在漫反射色，并且没有设置 D3DPV_DONOTCOPYDATA 标志。

若以下条件的或操作返回真，则 D3DFVF_SPECULAR 标志必须被设置。

光照被启用且镜面反射色被启用，也就是说，D3DRS_SPECULARENABLE为真。

光照被禁用，同时输入顶点流中存在镜面反射色，并且没有设置 D3DPV_DONOTCOPYDATA 标志。

顶点雾被启用，也就是说，D3DRS_FOGVERTEXMODE没有被设置为D3DFOG_NONE。

另外，纹理坐标的数量必须按以下方式设置。

如果在所有激活的纹理层中，纹理变换和纹理生成被禁用，且没有设置 D3DPV_DONOTCOPYDATA 标志，那么输出纹理坐标的数量和类型必须与输入纹理坐标相对应。如果设置了 D3DPV_DONOTCOPYDATA 标志，并且纹理变换和纹理生成被禁用，那么输出纹理坐标被忽略。

如果在任何激活的纹理层中，纹理变换和纹理生成被启用，那么输出顶点可能需要包含比输入顶点更多的纹理坐标组。这是由纹理生成产生的纹理坐标的增生造成的，或者是因为纹理变换。注意在IDirect3DDevice9::DrawPrimitives调用期间会发生一个类似的纹理坐标增生，但这对应用程序程序员而言是不可见的。在这种情况下，Direct3D会生成一组新的纹理坐标。系统通过遍历

纹理层，分析纹理生成、纹理变换和纹理坐标索引来决定是否该层需要唯一的纹理坐标组。每当需要一组新的纹理坐标时，系统会按照递增的顺序进行分配。注意，虽然可以通过使用D3DTSS_TEXCOORDINDEX共享未经变换的纹理坐标减少对纹理坐标组的需求，但最大需求，也是典型需求，为每层一组纹理坐标。

因此，对每一纹理层来说，如果有一个纹理被绑定到该层且以下任一条件为真，那么会产生一组新的纹理坐标。

该层启用了纹理生成。

该层启用了纹理变换。

未经变换的输入纹理坐标第一次通过 D3DTSS_TEXCOORDINDEX 被引用。

当 Direct3D 生成纹理坐标时，要求应用程序执行以下操作。

使用一个有合适的 FVF 用途的目标顶点缓存。

根据处理后纹理坐标的布局重新调整纹理层的D3DTSS_TEXCOORDINDEX。注意对D3DTSS_TEXCOORDINDEX设定的重新调整在经过处理的顶点缓存被用于随后的IDirect3DDevice9::DrawPrimitive和IDirect3DDevice9::DrawIndexedPrimitive调用时发生。

最后，纹理坐标的维度（从 D3DFVF_TEX0 到 D3DFVF_TEX8）必须按以下方式设置。

对每一组纹理坐标来说，如果纹理变换和纹理生成被禁用，那么输出纹理坐标的维度必须与输入相对应。如果纹理变换被启用，那么输出的维度必须与 D3DTTFF_COUNT1，D3DTTFF_COUNT2，D3DTTFF_COUNT3，D3DTTFF_COUNT4 设定所定义的数量相对应。如果纹理变换被禁用且纹理生成被启用，那么输出的维度必须与纹理生成模式的设定相对应，当前所有模式都产生三个浮点数。

当 IDirect3DDevice9::ProcessVertices 因为与目标顶点缓存的 FVF 码不兼容而失败时，Direct3D 会把希望得到的 FVF 码打印到调试输出（仅限于调试版）。

可编程顶点处理

当使用可编程顶点着色器时，对目标顶点缓存中元素的更新由着色器程序控制。当应用程序写入目标顶点寄存器之一时，目标顶点缓存中每个顶点对应的元素被更新。目标顶点缓存中未经应用程序写入的元素不会被修改。若应用程序试图写入目标顶点缓存中不存在的元素，则IDirect3DDevice9::ProcessVertices方法会失败。

设备类型与顶点处理的要求

顶点处理操作的性能，包括变换和光照，很大程度上取决于顶点缓存在内存中的位置以及正在使用的渲染设备的类型。在创建顶点缓存时，应用程序控制它们的内存分配。如果设置了 D3DPOOL_SYSTEMMEM 内存标志，那么顶点缓存会被创建在系统内存中。如果使用了 D3DPOOL_DEFAULT 内存标志，那么驱动程序会决定最好把顶点缓存分配在哪里，这通常被称为驱动程序最优（driver-optimal）内存。驱动程序最优内存可以是本地显存，非本地显存，或系统内存。

在调用IDirect3DDevice9::CreateVertexBuffer时设置*D3DUSAGE_SOFTWAREPROCESSING*常量会启用对顶点缓存数据的软件顶点处理。在混合模式下，这个标志是软件顶点处理必需的。这个标志不能用于硬件顶点处理模式。用于软件顶点处理的顶点缓存包括以下几种：

IDirect3DDevice9::ProcessVertices的所有输入流。

IDirect3DDevice9::DrawPrimitive和IDirect3DDevice9::DrawIndexedPrimitive的所有输入流。

应用程序用来决定顶点缓存所在的位置——系统内存或驱动程序最优内存——的原因与纹理相同。硬件顶点处理，包括变换和光照，最好与分配在驱动程序最优内存中顶点缓存一起使用，而软件顶点处理则最好与分配在系统内存中的顶点缓存一起使用。对纹理来说，分配在驱动程序最优内存中的纹理最好与硬件光栅化一起使用，而软件光栅化则最好与系统内存纹理一起使用。

注意Microsoft DirectX® 9.0 的Microsoft® Direct3D®由IDirect3DDevice9::ProcessVertices

支持独立的顶点处理，而不需要渲染任何图元。这个独立的顶点处理总是以软件方式在主处理器上执行。由于这个原因，用IDirect3DDevice9::SetStreamSource设置的源顶点缓存必须是用D3DUSAGE_SOFTWAREPROCESSING标志创建的。在使用软件顶点处理时，IDirect3DDevice9::ProcessVertices提供的功能与IDirect3DDevice9::DrawPrimitive和IDirect3DDevice9::DrawIndexedPrimitive方法提供的功能完全一样。

如果应用执行自己的顶点处理并传递经过变换、光照和裁剪的顶点给渲染方法，那么应用程序可以直接把顶点写入到分配在驱动程序最优内存中的顶点缓存。这项技术避免了以后一个冗余的复制操作。注意如果应用程序要从顶点缓存中读回数据，那么这项技术不会很有效，因为从主机到驱动程序最优内存的读操作可能非常慢。因此，如果应用程序需要在处理过程中读取数据或不规律地写入数据，那么系统内存顶点缓存是一个更好的选择。

在使用 Direct3D 的顶点处理特性时（通过传递未经变换的顶点到顶点缓存渲染方法），根据设备类型和设备创建标志，处理既可用硬件也可用软件执行。实际上在所有情况下，为得到最佳性能，最好在 D3DPOOL_DEFAULT 池中分配顶点缓存。当设备使用硬件顶点处理时，根据 D3DUSAGE_DYNAMIC 和 D3DUSAGE_WRITEONLY 标志，还可以进行许多其它的优化。更多有关使用这些标志的信息，请参阅 IDirect3DDevice9::CreateVertexBuffer。

索引缓存

索引缓存，由IDirect3DIndexBuffer9接口表示，是包含索引数据的内存缓存。索引数据，或索引，是顶点缓存中的整型偏移量，在用IDirect3DDevice9::DrawIndexedPrimitive方法渲染图元时使用。

顶点缓存和索引缓存是不相关的概念，也就是说，用索引表示的图元可以与顶点缓存一起或不一起使用，或者顶点缓存可以与用索引表示的图元一起或不一起使用。

索引缓存由它的能力表示：是否只能存在于系统内存中，是否只能用于写操作，及类型和能包含的索引的数量。这些特性包含在D3DINDEXBUFFER_DESC结构中。

索引缓存描述告诉应用程序一个已有的缓存是如何创建的。只要应用程序给系统提供一个空的描述结构，系统会在结构中填入先前创建的索引缓存的能力。有关此任务的更多信息，请参阅存取索引缓存的内容。

Format 成员描述索引缓存数据的表面格式。

Type 标识索引缓存的资源类型。

Usage 结构成员包含了综合的能力标志。D3DUSAGE_SOFTWAREPROCESSING 标志表示索引缓存要被用于软件顶点处理。Usage 中存在 D3DUSAGE_WRITEONLY 标志表示索引缓存只能用于写操作。这使驱动程序可以自由地将索引数据放在最佳的内存位置以进行快速处理和渲染。如果没有使用 D3DUSAGE_WRITEONLY 标志，驱动程序不太可能把数据放在一个读操作的效率很低的地方。这牺牲了一些处理时间和渲染速度。如果没有指定该标志，那么系统会假定应用程序对索引缓存中的数据执行读写操作。

Pool 指定为索引缓存分配的内存的类型。D3DPOOL_SYSTEMMEM 标志指示系统在系统内存中创建索引缓存。

Size 成员存储索引缓存的大小，以字节为单位。

更多信息包含在以下主题中。

创建索引缓存

存取索引缓存的内容

用索引缓存进行渲染

创建索引缓存

调用IDirect3DDevice9::CreateIndexBuffer方法创建一个索引缓存，该方法接收五个参数。第一个参数指定索引缓存的长度，以字节为单位。

第二个参数是一组用途控制标志。其中，它的值决定由索引引用的顶点是否能包含裁剪信息。要提高性能，应该在不需要裁剪时指定 D3DUSAGE_DONOTCLIP 标志。
当在设备上启用了混合模式或软件顶点处理（D3DCREATE_MIXED_VERTEXPROCESSING / D3DCREATE_SOFTWARE_VERTEXPROCESSING）时，可以设置 D3DUSAGE_SOFTWAREPROCESSING 标志。在混合模式中，要进行软件顶点处理的缓存必须设置 D3DUSAGE_SOFTWAREPRECESSING 标志，但在使用硬件顶点处理（D3DCREATE_HARDWARE_VERTEXPROCESSING）时，为了得到尽可能好的性能，不应设置该标志。但是，当一个缓存要同时进行硬件和软件顶点处理时，设置 D3DUSAGE_SOFTWAREPROCESSING 是唯一的选择。和软件设备一样，混合设备也可以使用 D3DUSAGE_SOFTWAREPROCESSING。
即使顶点处理是用硬件完成的，通过指定 D3DPOOL_SYSTEMMEM，把顶点和索引缓存强制分配到系统内存中也是可以的。当驱动程序将这些缓存放在 AGP 内存中时，这种方法可以避免大量的页面锁定内存。
第三个参数是D3DFORMAT枚举类型的成员，D3DFMT_INDEX16 或D3DFMT_INDEX32，它用来指定每个索引的大小。
第四个参数是D3DPOOL枚举类型的成员，告诉系统把新的索引缓存放在内存中的何处。
IDirect3DDevice9::CreateIndexBuffer接收的最后一个参数是变量的地址，如果调用成功的话，它会把一个指向新的IDirect3DIndexBuffer9索引缓存对象的指针填入该变量。
以下 C++示例代码显示了如何在代码中创建一个索引缓存。

```
/*
 * 根据本示例的目的，d3dDevice 变量是一个由 Direct3DDevice 对象暴露的
 * IDirect3DDevice9 接口的地址，g_IB 是 LPDIRECT3DINDEXBUFFER9 类型的变量。
 */

if( FAILED( d3dDevice->CreateIndexBuffer( 16384 *sizeof(WORD),
                                          D3DUSAGE_WRITEONLY, D3DFMT_INDEX16,
                                          D3DPOOL_DEFAULT, &g_IB ) ) )
    return E_FAIL;
```

索引处理的要求
索引处理操作的性能很大程度上取决于索引缓存位于内存的何处以及正在使用的渲染设备的类型。在创建索引缓存时，应用程序控制它们的内存分配。如果设置了 D3DPOOL_SYSTEMMEM 内存标志，那么索引缓存会被创建在系统内存中。如果设置了 D3DPOOL_DEFAULT 内存标志，那么设备驱动程序会决定最好把索引缓存的内存分配在哪里，这通常被称为驱动程序最优内存。驱动程序最优内存可以是本地显存，非本地显存，或系统内存。
在调用 IDirect3DDevice9::CreateIndexBuffer 方法时设置 D3DUSAGE_SOFTWAREPROCESSING 行为标志说明索引要被用于软件顶点处理。在混合模式顶点处理中，在使用软件顶点处理时必需要设这个标志。
应用程序可以直接把索引写入到位于驱动程序最优内存中的索引缓存。这项技术避免了以后一个冗余的复制操作。如果应用程序要从索引缓存中读回数据的话，那么这项技术不会很有效，因为从主机到驱动程序最优内存的读操作可能非常慢。因此，如果应用程序需要在处理过程中读取数据或不规律地写入缓存，那么系统内存索引缓存是一个更好的选择。
注意除非不想消耗显存，或当驱动程序把顶点或索引缓存放在 AGP 内存中时不想增加大量的页面锁定内存，否则应该总是使用 D3DPOOL_DEFAULT 标志。
存取索引缓存的内容

索引缓存对象使应用程序能直接存取为索引数据分配的内存。应用程序可以通过调用IDirect3DIndexBuffer9::Lock方法得到一个指向索引缓存的内存指针，然后根据需要存取内存并填入新的索引数据或读取其中包含的任何数据。IDirect3DIndexBuffer9::Lock方法接收四个参数。第一个参数*OffsetToLock*，是索引数据中的偏移量。第二个参数*SizeToLock*，是索引数据的大小，以字节为单位。第三个参数*ppbData*，是一个字节指针的地址，如果调用成功的话，会在该地址中填入指向索引数据的指针。

最后一个参数 *Flags*，告诉系统怎样锁定内存。应用程序可以用它表示应用程序如何存取缓存中的数据。应该根据应用程序存取索引数据的方法指定 *Flags* 参数的常量。这允许驱动程序根据给定的存取类型，锁定内存并提供最佳性能。如果应用程序只从索引缓存中读取数据，那么应该使用 D3DLOCK_READONLY 标志。假设对内存的存取是只读的，包含这个标志使 Microsoft® Direct3D® 能够优化它的内部处理以提高效率。

在填入或读取索引数据后，应该调用IDirect3DIndexBuffer9::Unlock方法，如以下示例代码所示。

```
// 本示例代码假设 IB 是一个 LPDIRECT3DINDEXBUFFER9 类型的变量
// 并且 g_Indices 已经用索引值进行了适当地初始化。

// 要填入索引缓存，必须先锁定缓存取得对索引的存取权，
// 这个机制是必要的因为索引缓存可能在设备内存中。

VOID* pIndices;

if( FAILED( IB->Lock( 0,                 // 从缓存开始处填充
                      sizeof(g_Indices), // 要载入数据的大小
                      (BYTE**)&pIndices, // 返回的索引数据
                      0 ) ) )            // 默认的锁定标志
    return E_FAIL;

memcpy( pIndices, g_Indices, sizeof(g_Indices) );
IB->Unlock();
```

注意

如果应用程序用 D3DUSAGE_WRITEONLY 标志创建一个索引缓存，那么不应该用 D3DLOCK_READONLY 锁定标志。如果应用程序只从索引缓存的内存中读取数据，那么应该用 D3DLOCK_READONLY 标志。假设对内存的存取为只读，包含这个标志使 Direct3D 能够优化它的内部处理以提高效率。

有关在IDirect3DIndexBuffer9::Lock方法的*Flags*参数中使用D3DLOCK_DISCARD或D3DLOCK_NOOVERWRITE标志的信息，请参阅使用动态顶点和索引缓存。

因为 C++程序直接存取分配给索引缓存的内存，所以要确保应用程序正确地存取内存。否则，存在渲染无效内存的风险。在应用程序中从缓存中的一个索引移动到另一个时，应该使用索引格式的步距。

取得索引缓存的描述

通过调用IDirect3DIndexBuffer9::GetDesc方法可以取得关于一个索引缓存的信息。这个方法把有关索引缓存的信息填入D3DINDEXBUFFER_DESC结构的成员。

用索引缓存进行渲染

用索引缓存渲染索引数据需要几个步骤。首先，应用程序需要通过调用IDirect3DDevice9::SetStreamSource方法设置数据流的源。

```
d3dDevice->SetStreamSource( 0, VB, 0, sizeof(CUSTOMVERTEX) );
```

SetStreamSource 的第一个参数告诉 Microsoft® Direct3D®设备数据流的源。第二个参数是要绑定到数据流的顶点缓存。第三个参数是成员的大小，以字节为单位。在上面的示例代码中，CUSTOMVERTEX 的大小被用作成员的大小。

下一步是调用IDirect3DDevice9::SetIndices方法设置索引数据的源。

```
d3dDevice->SetIndices( IB );
```

IDirect3DDevice9::SestIndices 方法接收的第一个参数是要设置的索引缓存的地址。第二个参数是在顶点流中的起始点。

在设置数据流和索引的源后，应该用IDirect3DDevice9::DrawIndexedPrimitive方法渲染使用了索引缓存中的索引的顶点。

```
d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 0, dwVertices, 0, dwIndices / 3);
```

IDirect3DDevice9::DrawIndexedPrimitive 接收的第三个参数 *MinIndex* 是本次调用中用到的最小顶点索引。第四个参数 *NumVertices* 是本次调用过程中用到的索引的数量，从 *BaseVertexIndex* + *MinIndex* 开始。第五个参数 *StartIndex* 是在索引数组中读取索引的起始位置。

IDirect3DDevice9::DrawIndexedPrimitive 接收的最后一个参数 *PrimitiveCount* 是要渲染的图元的数量。这个参数是一个图元数量和图元类型的函数。上面的示例代码使用了三角形，因此要渲染的图元数量是索引数量除以三。

渲染

应用程序使用 DrawPrimitive 系列方法渲染三维场景。以下主题讨论用 DrawPrimitive 渲染。

着色

呈现场景

渲染图元

深度缓存

雾

阿尔法混合

D3DX线段绘制

着色

本节描述 Microsoft® Direct3D®中用于控制三维多边形着色的技术。

概述

着色模式

着色模式的比较

设置着色模式

着色模式

用于渲染多边形的着色模式完全影响到渲染结果。着色模式决定在多边形表面上任意一点上颜色的强度和光照。Microsoft® Direct3D®支持两种着色模式。

平面着色

高洛德着色

平面着色

在平面着色模式中，当 Direct3D 渲染流水线渲染一个多边形时，使用第一个顶点的材质颜色作为整个多边形的颜色。用平面着色渲染的三维物体有明显的边缘，如果它们不共面的话。

下图显示了一个用平面着色模式渲染的茶壶。我们可以清楚地看到每个多边形的轮廓。平面着色是计算量最小的着色模式。

高洛德着色

当 Direct3D 使用高洛德着色渲染一个多边形时，它根据顶点法向和光照参数为每个顶点计算颜色。然后，贯穿多边形的表面对颜色进行插值。插值是线性的。例如，如果顶点 1 的颜色的红色分量为 0.8，顶点 2 的颜色的红色分量为 0.4，如果使用高洛德着色模式和 RGB 颜色模型，那么 Direct3D 光照模块会把红色分量 0.6 赋给位于这两个顶点连线中点处的像素。

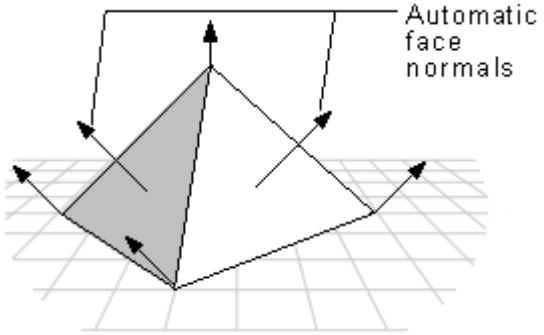
下图显示了高洛德着色。茶壶由许多平直的三角形组成。但是，高洛德着色使物体的表面看起来是弯曲和光滑的。

高洛德着色也可以用于显示有清晰边缘的物体。

更多信息请参阅表面和顶点法向量。

着色模式的比较

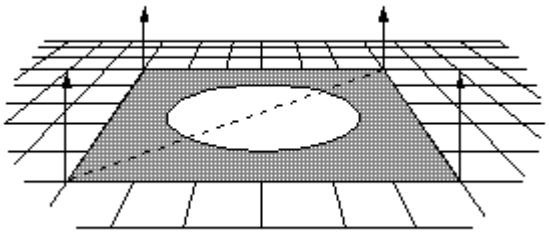
在平面着色模式中，下面显示的金字塔在相邻表面间会有清晰的边缘。但是在高洛德着色模式中，用于着色的值是贯穿边缘插值得到的，最终看起来是一个弯曲的表面。



高洛德着色使经过光照的平直表面看起来比平面着色更真实。平面着色模式中的表面具有相同的颜色，但高洛德着色能使光线更准确地照射在表面上。如果附近有点光源，那么这种效果特别明显。

高洛德着色使在平面着色中明显可见的边缘变得光滑。但是，它会产生马克带（marc band），这是在相邻多边形间没有平滑混合的颜色带或光带。应用程序可以通过增加物体中多边形的数量，或增大屏幕分辨率，或增加颜色深度来减少马克带效应。

高洛德着色会漏掉一些细节。下图显示了这样一个例子，一个光照点完全被包含在多边形表面内。



在这种情况下，在顶点间进行插值的高洛德着色将会完全漏掉光照点，表面的渲染就好像光照点不存在一样。

设置着色模式

Microsoft® Direct3D®允许在任一时刻选择一种着色模式。默认情况下选择高洛德着色。C++应用程序可以通过调用IDirect3DDevice9::SetRenderState方法改变着色模式。应该将*State*参数设为D3DRS_SHADEMODE。*Value*参数必须被设为D3DSHADEMODE枚举类型的成员。以下示例代码显示了如何将Direct3D应用程序当前的着色模式设置为平面或高洛德着色模式。

```
// 设置为平面着色。
// 本示例代码假设 pDev 是一个指向 IDirect3DDevice9 接口的有效指针。
hr = pDev->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);
if(FAILED(hr))
{
    // 这里是错误处理代码。
}

// 设置为高洛德着色。这是 Direct3D 的默认设置。
hr = pDev->SetRenderState(D3DRS_SHADEMODE,
                          D3DSHADE_GOURAUD);
if(FAILED(hr))
{
    // 这里是错误处理代码。
}
```

呈现场景

本节介绍呈现（presentation）应用程序编程接口（API），并讨论了将一个场景呈现在显示设备上的相关问题。

信息被分为以下主题。

概述

呈现场景入门

窗口模式下的多个视区

多显示器操作

操作深度缓存

存取前颜色缓存

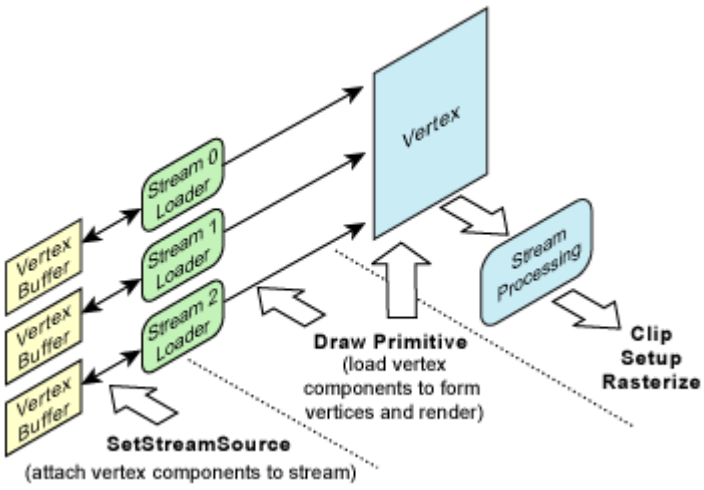
呈现场景入门

呈现 API 是控制设备状态的一组方法，这些状态会影响用户在显示器上所看到的。这些方法包括设置显示模式和在每一帧用来把图像呈现给用户的方法等。

IDirect3DDevice9::Present

IDirect3DDevice9::Reset
IDirect3DDevice9::GetGammaRamp
IDirect3DDevice9::SetGammaRamp
IDirect3DDevice9::GetRasterStatus
要理解呈现 API，熟悉以下术语是必需的。
前缓存。是一块矩形内存，由图形适配器解释并显示在显示器或其它输出设备上。
后缓存。是一个包含可被切换为前缓存的内容的表面。
交换链。是一些可连续被呈现到前缓存的后缓存组成的集合。一般来说，一个全屏交换链用 flipping 设备驱动程序接口（DDI）呈现后续图像，一个窗口交换链用 blitting 设备驱动程序接口呈现图像。
因为Microsoft® DirectX® 9.0 的Microsoft® Direct3D®有一个交换链作为设备的属性，所以每个设备总是至少有一个交换链。IDirect3DDevice9接口有一组方法操作隐式交换链，这些方法是交换链自身接口的复制品。应用程序可以创建附加的交换链，但是，对一般的单窗口或全屏应用程序而言这并不是必需的。
DirectX 9.0 的Direct3D API不直接暴露前缓存。因此，应用程序不能锁定或渲染到前缓存。细节请参阅存取前颜色缓存。
注意　DirectX 7.0提供了许多需要一起调用的呈现API。IDirectDraw7::SetCooperativeLevel，IDirectDraw7::SetDisplayMode 和 IDirectDraw7:CreateSurface 序列就是很好的例子。另外，IDirectDrawSurface7::Flip 和 IDirectDrawSurface7::Blt 方法通知将渲染完的帧数据传送到显示器。DirectX 9.0 将这些不同组的 API 合并为两个主要的方法：Reset 和 Present。Reset 归入了 SetcooperativeLevel，SetDisplayMode，CreateSurface 和 Flip 的一些参数。Present 归入了 Flip 和作为呈现 API 使用的 Blt。
对IDirect3D9::CreateDevice的调用表示隐式地重置设备。DirectX 9.0 API没有主表面的概念，应用程序无法创建一个代表主表面的对象。主表面被视为设备的内部属性。
与交换链相关的伽马（gamma）值通过 IDirect3DDevice9::GetGammaRamp 和 IDirect3DDevice9::SetGammaRamp 方法进行操作。
窗口模式下的多个视区
除了Direct3DDevice对象拥有并操纵的交换链外，应用程序可以调用 IDirect3DDevice9::CreateAdditionalSwapChain方法创建附加的交换链，这样就可以从同一设备呈现到不同的视区。
一般情况下，应用程序为每个视区创建一个交换链，并将每个交换链与特定的视区相关联。应用程序将图像渲染到每个交换链的后缓存中，然后调用IDirect3DDevice9::Present方法单独呈现它们。注意同一时刻每个适配器只能有一个交换链是全屏的。
多显示器操作
当设备成功地重置（IDirect3DDevice9::Reset）或在全屏模式下创建（IDirect3D9::CreateDevice）时，创建该设备的Microsoft® Direct3D®对象将被标记为拥有系统中所有的适配器。这种状态被称为独占模式，且Direct3D对象拥有独占模式。独占模式意味着由其它Direct3D对象创建的设备既不能进行全屏操作也不能分配显存。另外，当一个Direct3D对象进入独占模式时，除了当前进入全屏模式以外的所有设备都被置为丢失状态。细节请参阅丢失的设备。
进入独占模式时，Direct3D 对象会被告知设备将要使用的焦点窗口。当 Direct3D 设备拥有的最后一个全屏设备被重置为窗口模式或销毁时，独占模式即被解除。
当一个 Direct3D 对象拥有独占模式时，可以把设备分为两类。第一类设备具有以下特点：
它们由创建全屏设备的 Direct3D 对象创建。

它们具有和全屏设备相同的焦点窗口。
它们和任何全屏设备代表不同的适配器。
这类设备在被重置或创建方面没有任何限制，并且它们不会被置为丢失状态。这类设备甚至可以进入全屏模式。
不属于第一类的设备——由另一个 Direct3D 对象创建，用不同的焦点窗口创建，以及在一个已经存在全屏设备的适配器上创建——不能被重置，并且在独占模式退出之前一直保持丢失状态。因此，多显示器应用程序可以把几个设备置为全屏模式，不过条件是这些设备是同一个 Direct3D 对象在不同的适配器上创建的，并且共享同一个焦点窗口。
操作深度缓存
深度缓存与设备相关联。在设置渲染对象时，应用程序需要移动深度缓存。
IDirect3DDevice9::GetDepthStencilSurface和IDirect3DDevice9::SetRenderTarget方法用于操作深度缓存。
存取前颜色缓存
通过IDirect3DDevice9::GetFrontBuffer方法可以存取前缓存。该方法是取得反走样场景的截图的唯一方法。
渲染图元
以下主题介绍IDirect3DDevice9::DrawPrimitive等渲染方法，并提供关于在应用程序中使用它们的信息。
顶点数据流
设置数据流的源
用顶点和索引缓存渲染
用用户内存指针渲染
顶点数据流
Microsoft® Direct3D®的渲染接口包含了许多方法，这些方法用存储在一个或多个数据缓存中的顶点数据渲染图元。顶点数据由顶点元素组合而成的顶点成员组成。顶点元素是顶点的最小单位，代表诸如位置、法向和颜色的实体。
顶点成员是一个或多个顶点元素连续存储（每个顶点交叉）在单个内存缓存中。一个完整的顶点由一个或多个成员组成，每个成员位于单独的内存缓存中。要渲染一个图元，需要读取多个顶点成员并将它们组合，这样得到的完整的顶点就可以进行顶点处理。下图显示了使用顶点成员渲染图元的过程。



渲染图元有两个步骤。首先，设置一个或多个顶点成员数据流；然后，调用IDirect3DDevice9::DrawPrimitive方法用那些数据流渲染。对位于这些成员数据流中的顶点元

素的识别由顶点着色器指定。更多细节，请参阅顶点着色器。

IDirect3DDevice9::DrawPrimitive 方法指定一个顶点数据流中的偏移量，这样就可以在每次调用时渲染一组顶点数据中的一个连续图元的子集。这使应用程序在用同一个顶点缓存渲染其中不同的图元组时，可以改变渲染状态。

Direct3D同时支持使用索引和不使用索引的绘制方法。更多信息，请参阅用顶点和索引缓存渲染。

设置数据流的源

IDirect3DDevice9::SetStreamSource方法将一个顶点缓存绑定到一个设备数据流，在顶点数据和多个数据流端口中的一个之间创建关联，这些端口给图元处理函数提供数据。对流数据的实际引用直到调用诸如IDirect3DDevice9::DrawPrimitive之类的绘制方法时才发生。

数据流被定义为由成员数据组成的统一数组，其中每个成员由一个或多个表示单个实体的元素组成，如位置、法向、颜色等等。*Stride* 参数指定成员的大小，以字节为单位。

用顶点和索引缓存渲染

Microsoft® Direct3D®同时支持使用索引和不使用索引的绘制方法。使用索引的方法（以后简称为索引方法）给所有顶点成员使用单独一组索引。顶点数据以顶点缓存的形式呈现给 Direct3D 应用编程接口（API），给索引绘制方法使用的索引数据以索引缓存的形式呈现。更多信息，请参阅以下参考主题。

IDirect3DDevice9::DrawPrimitive

IDirect3DDevice9::DrawIndexedPrimitive

这些方法绘制当前输入数据流的数据中的图元。更多有关将索引缓存设置为当前索引数组的细节，请参阅IDirect3DDevice9::SetIndices方法。

用用户内存指针渲染

第二组渲染接口支持直接来自用户内存的顶点和索引数据。这些接口只支持单独的数据流。更多信息，请参阅以下参考主题。

IDirect3DDevice9::DrawPrimitiveUP

IDirect3DDevice9::DrawIndexedPrimitiveUP

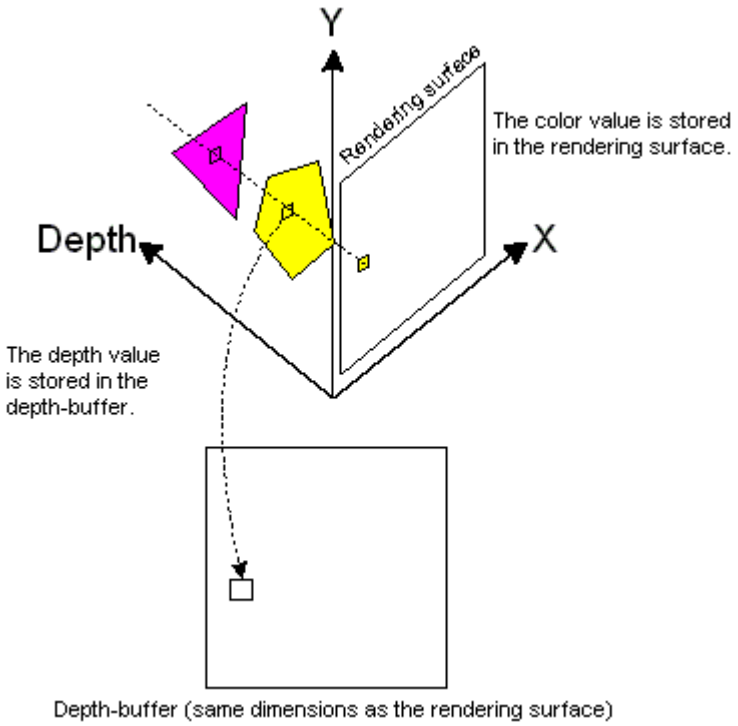
这些方法用用户内存指针渲染，而不是顶点和索引缓存。

深度缓存

深度缓存，常被称为 z 缓存或 w 缓存，是设备的一个属性，用来存储深度信息，给 Microsoft® Direct3D®使用。当 Direct3D 将三维场景渲染到一个目标表面时，它可以使用与深度缓存表面相关的内存作为工作空间来决定经过光栅化的多边形的像素如何相互遮蔽。Direct3D 使用一个屏外表面作为写入最后颜色的目的地。与渲染目标表面相关联的深度缓存被用于存储深度信息，告诉 Direct3D 场景中每个可见像素有多深。

光栅化三维场景时，如果启用了深度缓存，那么渲染表面上的每个点都要被测试。深度缓存中的值可以是一个点的 z 坐标值或它的齐次坐标的 w 值——从该点在投影空间中的位置（x,y,z,w）得到。使用 z 值的深度缓存通常被称为 z 缓存，使用 w 值的被称为 w 缓存。两种深度缓存各有利弊，这在以后讨论。

在测试开始时，深度缓存中的值被设为场景中最大可能达到的值。渲染表面的颜色值被设为背景颜色值或者背景纹理在该点处的颜色值。场景中的每个多边形都要被测试，看它有没有与渲染表面的当前坐标（x,y）相交。如果相交，那么当前点的深度值——z 缓存的 z 坐标，或 w 缓存中的 w 坐标——被测试，看它是不是小于存储在深度缓存中的深度值。如果多边形在该点处的的深度值更小，那么它将被存储到深度缓存中，并且多边形在该点处的颜色值将被写入到渲染表面的当前点。如果多边形在该点处的深度值更大，那么（渲染）列表中的下一个多边形将被测试。这个过程如下图所示。



注意虽然大多数应用程序不使用这个特性，但是应用程序可以改变Direct3D使用的比较函数，这个比较函数被用来决定哪个值将被放在深度缓存及随后的渲染表面中。要使用这个特性，应该改变D3DRS_ZFUNC渲染状态的值。

市面上所有的三维加速卡都支持 z 缓存，这使 z 缓存成为当今最普遍的深度缓存类型。类型但无论如何普遍，z 缓存有它的缺点。因为涉及到的数学计算，z 缓存中生成的 z 值在 z 缓存的范围内（一般来说从 0.0 到 1.0，闭区间）不是平均分布的。特别是远裁剪平面和近裁剪平面的比值严重影响到 z 值的不平均分布。如果远/近裁剪平面的比值为 100，那么深度缓存中 90%的范围被用在场景中最开始 10%的深度范围。一个一般的室外场景的娱乐或视觉模拟应用程序经常需要远/近裁剪平面的比值在 1000 到 10000 之间。如果比值是 1000，那么 98%的范围被用在场景中最开始 2%的深度范围，更高的比值会导致分布情况变得更糟。这会导致远处的物体的隐藏面遗留物，尤其是使用 16 位深度缓存时，而这也是被广泛支持的位深度。

另一方面，基于 w 的深度缓存，相比 z 缓存更为平均地分布在远近裁剪平面之间。最大的好处在于远/近裁剪平面的比值不再是问题。这使应用程序可以支持很大的最大范围，同时在视点附近保持相当的精确度。基于 w 的深度缓存并非完美，有时会出现近处物体的隐藏面遗留物。w 缓存方法的另一个缺点与硬件支持相关：w 缓存没有 z 缓存那样广泛的硬件支持。

Z缓存在渲染过程中需要一些开销。使用z缓存时，可以使用多种方法进行优化。更多细节，请参阅Z缓存的性能。

注意对深度值的解释取决于三维渲染器。

本节提供有关使用深度缓存进行隐藏线和隐藏面消除的信息。

查询深度缓存支持

创建深度缓存

启用深度缓存

取得深度缓存

清除深度缓存

改变深度缓存的写权限

改变深度缓存的比较函数

使用Z偏移

查询深度缓存支持

与任何特性一样，应用程序使用的驱动程序可能不支持所有类型的深度缓存。应该总是检查驱动程序的能力。虽然大多数驱动程序支持基于 z 的深度缓存，但并不都支持基于 w 的深度缓存。如果应用程序试图启用不支持的类型，那么驱动程序不会失败，它们会退而使用另一种缓存类型，或者有时完全禁用深度缓存，这将导致渲染出来的场景有极多的深度排序遗留物。

应用程序可以在创建 Direct3D 设备前，通过 Microsoft® Direct3D®查询应用程序将要使用的显示设备，以检查对深度缓存的常规支持。如果 Direct3D 对象报告设备支持深度缓存，那么从该 Direct3D 对象创建的硬件设备支持 z 缓存。

要查询对深度缓存的支持，可以使用IDirect3D9::CheckDeviceFormat方法，如以下示例代码所示。

```
// 以下示例代码假设 pCaps 是一个指向 D3DCAPS9 结构的有效指针。
if( FAILED( m_pD3D->CheckDeviceFormat( pCaps->AdapterOrdinal,
                                       pCaps->DeviceType,
                                       AdapterFormat,
                                       D3DUSAGE_DEPTHSTENCIL,
                                       D3DRTYPE_SURFACE,
                                       D3DFMT_D16 ) ) )
    return E_FAIL;
```

IDirect3D9::CheckDeviceFormat 允许应用程序根据设备的能力选择要创建的设备。在这种情况下，不支持 16 位深度缓存的设备被抛弃。

另外，可以使用IDirect3D9::CheckDepthStencilMatch方法检测某一被支持的深度/模板缓存的格式与当前显示模式中的渲染目标是否兼容，特别是深度缓存的格式必须与渲染目标的格式相同时。

以下示例代码显示了使用 IDirect3D9::CheckDepthStencilMatch 方法检测深度缓存/模板缓存与渲染目标之间的兼容性。

```
// 抛弃不能创建渲染目标为 RTFormat，同时后缓存为 RTFormat，
// 深度缓存/模板缓存至少为 8 位的设备。
if( FAILED( m_pD3D->CheckDepthStencilMatch( pCaps->AdapterOrdinal,
                                            pCaps->DeviceType,
                                            AdapterFormat,
                                            RTFormat,
                                            D3DFMT_D24S8 ) ) )
    return E_FAIL;
```

当知道驱动程序支持深度缓存时，可以检验对 w 缓存的支持。虽然所有软件光栅化器都支持深度缓存，但是只有参照光栅化器支持 w 缓存，而参考光栅化器不适合用于现实应用程序。无论应用程序使用何种类型的设备，都应该在试图启用基于 w 的深度缓存之前检验对 w 缓存的支持。

在创建设备之后，调用IDirect3DDevice9::GetDeviceCaps方法，并传入一个经过初始化的D3DCAPS9结构。

在调用之后，*LineCaps* 成员包含了驱动程序对渲染图元的支持的信息。

如果结构的 *RasterCaps* 成员包含 D3DPRASTERCAPS_WBUFFER 标志，那么对那种类型的图元，驱动程序支持基于 w 的深度缓存。

创建深度缓存

深度缓存是设备的一项属性。要创建一个由Microsoft® Direct3D®管理的深度缓存，应该如以下

示例代码所示设置D3DPRESENT_PARAMETERS结构的相应成员。

```
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed               = TRUE;
d3dpp.SwapEffect             = D3DSWAPEFFECT_COPY;
d3dpp.EnableAutoDepthStencil = TRUE;
d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
```

通过把 *EnableAutoDepthStencil* 成员设置为 TRUE，应用程序指示 Direct3D 为它管理深度缓存。注意 *AutoDepthStencilFormat* 必须被设置为一个有效的深度缓存格式。如果设备支持，那么可以用 D3DFMT_D16 标志指定一个 16 位深度缓存。

以下对IDirect3D9::CreateDevice方法的调用创建一个设备，该设备随后创建一个深度缓存。

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                  &d3dpp, &d3dDevice ) ) )
return E_FAIL;
```

深度缓存被自动设为设备的渲染目标。当设备被重置时，深度缓存被自动销毁然后用新的大小重建。

要创建一个新的深度缓存表面，应该使用IDirect3DDevice9::CreateDepthStencilSurface方法。

要给设备设置一个新的深度缓存表面，应该使用IDirect3DDevice9::SetRenderTarget方法。

要在应用程序中使用深度缓存，需要启用深度缓存。更多细节，请参阅启用深度缓存。

启用深度缓存

在如创建深度缓存中所描述的那样创建完深度缓存之后，应该通过调用IDirect3DDevice9::SetRenderState方法启用深度缓存。要启用深度缓存，应该设置D3DRS_ZENALBE渲染状态。使用D3DZBUFFERTYPE枚举类型的D3DZB_TRUE成员（或TRUE）启用z缓存，D3DZB_USEW启用w缓存，或D3DZB_FALSE（或FALSE）禁用深度缓存。

注意要使用w缓存，即使不使用Microsoft® Direct3D®变换流水线，应用程序也必须设置合适的投影矩阵。关于如何提供合适的投影矩阵的信息，请参阅W友好投影矩阵。什么是投影变换？中描述的投影矩阵是合适的。

取得深度缓存

以下示例代码显示了如何使用IDirect3DDevice9::GetDepthStencilSurface方法取得设备拥有的深度缓存。

```
LPDIRECT3DSURFACE9 pZBuffer;

m_d3dDevice->GetDepthStencilSurface( &pZBuffer );
```

清除深度缓存

许多C++程序在渲染新的一帧前清除深度缓存。可以通过调用IDirect3DDevice9::Clear方法，将*Flags*参数指定为D3DCLEAR_ZBUFFER让Microsoft® Direct3D®清除深度缓存。

IDirect3DDevice9::Clear方法允许给*Z*参数指定任意的深度值。

改变深度缓存的写权限

默认情况下，允许 Microsoft® Direct3D®系统写入深度缓存。虽然大多数应用程序允许写入深度缓存，但是也可以通过不允许 Direct3D 系统写入深度缓存以实现某种特殊效果。

C++应用程序可以通过调用IDirect3DDevice9::SetRenderState方法，将*Stage*参数设置为D3DRS_ZWRITEENABLE，*Value*参数设置为 0，以禁止写入深度缓存。

改变深度缓存的比较函数

默认情况下，在对一个渲染目标进行深度测试时，如果一个点的深度值（z或w）小于深度缓存中的值，那么Microsoft® Direct3D®系统会相应地更新渲染目标表面。C++应用程序可以通过调用IDirect3DDevice9::SetRenderState方法，将*State*参数设为D3DRS_ZFUNC，以改变系统比较深度值的方式。*Value*参数应该被设为D3DCMPFUNC枚举类型值。

使用 Z 偏移

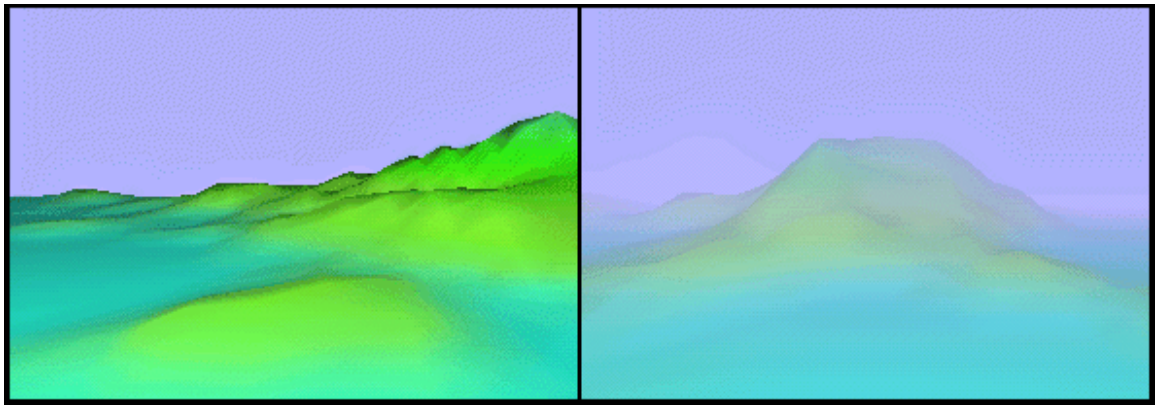
可以通过给多边形添加 z 偏移，使三维场景中共面的多边形看起来不共面。这是一种通常用来保证场景中的影子正确显示的技术。例如，墙上的影子很可能和墙有相同的深度值。如果先渲染墙后渲染影子，那么影子可能会看不见，或者可能看见深度遗留物。如果希望得到相反的效果，可以交换渲染共面物体的顺序，但是仍有可能有深度遗留物。

当渲染一组共面的多边形时，C++应用程序可以通过给系统使用的z值添加一个偏移来保证共面的多边形正确渲染。要给一组多边形添加一个偏移，应该在渲染它们之前调用IDirect3DDevice9::SetRenderState方法，将*State*参数设为D3DRS_DEPTHBIAS，将*Value*参数设为从 0 到 16 闭区间中的值。在与其它共面的多边形一起显示时，更高的z偏移值增加了正在渲染的多边形可见的可能性。

雾

给三维场景添加雾可以增强真实感，创造环境氛围，并减少有时由于远处几何体进入视线时造成的残留物。Microsoft® Direct3D® 支持两种类型的雾——像素雾和顶点雾——每种都有各自的特性和编程接口。

本质上讲，雾是通过根据场景中物体的深度或与视点间的距离，将场景中物体的颜色与选定的雾的颜色进行混合实现的。如果物体越来越远，它们的本色和雾的颜色混合的越多，造成物体愈加被场景中悬浮的微粒模糊的假象。下图显示了一个没有使用雾渲染的场景，和一个类似的使用雾渲染的场景。



在这幅图中，左边的场景有明显的地平线，在此之后就没有东西可见了，即便在现实世界中会是可见的。右边的场景通过使用与背景色相同颜色的雾使地平线变得模糊，使多边形淡出到远处。通过将离散的雾效果和有创意的场景设计相结合，应用程序可以添加气氛或使场景中物体的颜色变得柔和。

Direct3D 提供两种方法将雾添加到场景中，顶点雾和像素雾，由雾效果如何作用而命名。细节请参阅像素雾和顶点雾。简言之，像素雾——也被称为查找表雾——在设备驱动程序中实现，而顶点雾则在 Direct3D 光照引擎中实现。

注意不论应用程序使用何种类型的雾——像素雾或顶点雾，为了确保雾效果被正确应用，应用程序必须提供相容的透视矩阵。这个限制甚至对不使用Direct3D变换和光照引擎的应用程序同样适用。有关如何提供相应矩阵的更多细节，请参阅W友好投影矩阵。

以下主题介绍了雾并介绍了如何在 Direct3D 应用程序中使用各种雾特性的信息。

雾的公式

雾的参数
雾混合
雾的颜色
顶点雾
像素雾
雾混合由渲染状态控制，不是可编程像素流水线的一部分。
阿尔法混合
阿尔法混合用于显示含有透明或半透明像素的图像。除了红、绿、兰颜色通道外，阿尔法位图中的每个像素还有一个被称为阿尔法通道的透明成员。一般的阿尔法通道包含的位数与颜色通道相同。例如，一个 8 位阿尔法通道可以表示 256 级透明度，从 0（像素完全透明）到 255（像素完全可见）。下面显示了可以用阿尔法混合创建的一些特效。
可以定义包含或不含阿尔法的颜色。不含阿尔法的颜色是 RGB，包含阿尔法的颜色是 RGBA。顶点数据，材质数据和纹理数据可用于产生物体的透明度。也可以使用帧缓存产生透明效果。
顶点阿尔法
材质阿尔法
纹理阿尔法
帧缓存阿尔法
渲染目标阿尔法
演示阿尔法的示例包括：
广告牌
云、烟、和水汽尾迹
火、闪光和爆炸
顶点阿尔法
阿尔法数据可以在顶点数据中提供。要启用顶点阿尔法，应该将 D3DRENDERSTATE_DIFFUSEMATERIALSOURCE 设置为 D3DMCS_COLOR1，这样 Microsoft® Direct3D® 运行库就会从顶点的漫反射色而不是从材质颜色中取得漫反射色的值。

```
m_pd3dDevice->SetRenderState( D3DRENDERSTATE_DIFFUSEMATERIALSOURE,
                              D3DMCS_COLOR1 );
```

然后，在漫反射色中提供阿尔法值。AddAlphaToASphere 函数将阿尔法值添加到球体的顶点中。下面这个示例显示了如何将阿尔法信息提供给该函数。

```
AddAlphaToASphere( m_pObstacleVertices, 12,
          D3DRGBA(light.dcvDiffuse.r, light.dcvDiffuse.g,
                  light.dcvDiffuse.b, vAlpha ));
```

这是函数看起来的样子。

```
void AddAlphaToASphere(D3DLVERTEX* pVertices, DWORD dwNumRings, D3DCOLOR lightcolor)
{
 WORD x, y;
 // rings around
    for( y=0; y < dwNumRings; y++ )
        for( x=0; x < (dwNumRings*2)+1; x++ )
            (pVertices++)->color = lightcolor;

 // top and bottom
    (pVertices++)->color = lightcolor;
```

```
    (pVertices++)->color = lightcolor;
}
```

AddAlphaToASphere 函数只是简单地修改类型为 D3DLVERTEX 的每个顶点的颜色成员，使之包含阿尔法信息。

D3DLVERTEX 看起来像这样。

```
// 经过光照的顶点
typedef struct {
 D3DVALUE x, y, z;
 DWORD dwReserved;
 D3DCOLOR color, specular;
 D3DVALUE tu, tv;
} D3DLVERTEX, *LPD3DLVERTEX;
```

绘制球体，

```
  // 绘制经过光照的球体。
 m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, D3DFVF_LVERTEX,
                                m_pObstacleVertices, m_dwNumObstacleVertices,
                                m_pObstacleIndices,  m_dwNumObstacleIndices, 0 );
```

会得到一个使用顶点阿尔法的半透明球体。

材质阿尔法

阿尔法也可以在材质中提供。要启用材质阿尔法，应该将 D3DRENDERSTATE_DIFFUSEMATERIALSOURCE 设置为 D3DMCS_MATERIAL，这样 Microsoft® Direct3D® 运行库就会从材质颜色而不是从顶点的漫反射色元素中取得漫反射色的值。

```
   m_pd3dDevice->SetRenderState( D3DRENDERSTATE_DIFFUSEMATERIALSOURCE,
D3DMCS_MATERIAL );
```

要产生透明度，须用阿尔法值初始化材质。

```
mAlpha = 0.5f;
D3DUtil_InitMaterial( m_mtrl,
light.dcvDiffuse.r, light.dcvDiffuse.g, light.dcvDiffuse.b, mAlpha );
```

然后用材质阿尔法绘制一个半透明的球体。

```
// 绘制经过光照的球体。
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, D3DFVF_LVERTEX,
                        m_pObstacleVertices, m_dwNumObstacleVertices,
                        m_pObstacleIndices,  m_dwNumObstacleIndices, 0 );
```

纹理阿尔法

阿尔法也可以在纹理中提供。首先，必须创建纹理。然后，必须将之设置到一个纹理层，并选择相应的操作符和操作数。接着，就可以绘制有透明度的图元了。

```
// 创建一个阿尔法纹理
CreateEmptyTexture( "gradient",
128,128,0,D3DTEXTR_32BITSPERPIXEL|D3DTEXTR_CREATEWITHALPHA);

LoadGradient( "gradient" );
```

这些函数的实体完成这项工作。*CreateEmptyTexture* 通过创建一个空的纹理设置所需的一切。

```
HRESULT LoadGradient( TCHAR* strName )
{
```

```
    TextureContainer* ptcTexture = FindTexture( strName );

    // 载入一个梯度图
    return ptcTexture->LoadGradient32( strName );
}
```

一旦执行了查找纹理，就调用 *LoadGradient32* 将阿尔法通道载入。

```
HRESULT TextureContainer::LoadGradient32( TCHAR* strPathname )
{
 int  yGrad, xGrad;
 // w x h set in createempty
    m_pRGBAData = new DWORD[m_dwWidth*m_dwHeight];

    if( m_pRGBAData == NULL )
    {
        return E_FAIL;
    }

    for( DWORD y=0; y < m_dwHeight; y++ )
    {
        DWORD dwOffset = y*m_dwWidth;
        yGrad = (int)(((float)y/(float)m_dwWidth) * 255.0f);

        for( DWORD x=0; x < m_dwWidth; x )
        {
            xGrad = (int)(((float)x/(float)m_dwWidth) * 255.0f);

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

            m_pRGBAData[dwOffset+x] = (r<<24L)+(g<<16L)+(b<<8L)+(a);
            x++;
        }
    }

    return S_OK;
}
```

阿尔法值根据当前像素在纹理中的相对位置计算得到。

下一步，将纹理指定到一个纹理层并设置纹理层。

```
// 指定纹理
m_pd3dDevice->SetTexture( 0, D3DTextr_GetSurface("gradient") );

// 纹理层状态
```

```
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );

m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );

m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE );

m_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_MODULATE );

// 绘制一个经过光照的球
m_pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, D3DFVF_LVERTEX,
                                    m_pObstacleVertices, m_dwNumObstacleVertices,
                                    m_pObstacleIndices,  m_dwNumObstacleIndices, 0 );
```

这样就得到了一个透明的，带纹理的球。

帧缓存阿尔法

帧缓存阿尔法和顶点阿尔法、材质阿尔法和纹理阿尔法不太一样。顶点、材质和纹理的阿尔法设置仅用于当前图元的透明度值，它们对其它图元不产生任何效果。帧缓存阿尔法混合控制当前图元如何与帧缓存中现存的像素结合并产生最终像素的颜色。

执行阿尔法混合时会把两个颜色结合在一起：一个源色和一个目标色。源色来自透明物体，目标色是在像素位置已经存在的颜色。目标颜色是渲染一些位于透明物体之后的其它物体的结果，也就是说，它们可以透过透明物体被看见。阿尔法混合使用一个公式控制源和目标物体的比例。

最终颜色 = 物体颜色 * 源混合因子 + 像素颜色 * 目标混合因子

*物体颜色*是正在当前像素位置渲染的图元的颜色。*像素颜色*是帧缓存中当前像素位置的颜色。

所有可使用的阿尔法混合因子如下表所列。

混合模式因子	描述
D3DBLEND_ZERO	(0, 0, 0, 0)
D3DBLEND_ONE	(1, 1, 1, 1)
D3DBLEND_SRCCOLOR	(Rs, Gs, Bs, As)
D3DBLEND_INVSRCCOLOR	(1-Rs, 1-Gs, 1-Bs, 1-As)
D3DBLEND_SRCALPHA	(As, As, As, As)
D3DBLEND_INVSRCALPHA	(1-As, 1-As, 1-As, 1-As)
D3DBLEND_DESTALPHA	(Ad, Ad, Ad, Ad)
D3DBLEND_INVDESTALPHA	(1-Ad, 1-Ad, 1-Ad, 1-Ad)
D3DBLEND_DESTCOLOR	(Rd, Gd, Bd, Ad)
D3DBLEND_INVDESTCOLOR	(1-Rd, 1-Gd, 1-Bd, 1-Ad)
D3DBLEND_SRCALPHASAT	(f, f, f, 1); f = min(As, 1-Ad)
D3DBLEND_BOTHSRCALPHA	Microsoft® DirectX® 6.0 及后续版本已不再使用。可以通过将源混合因子和目标混合因子分别设为 D3DBLEND_SRCALPHA 和 D3DBLEND_INVSRCALPHA 达到相同的效果。

D3DBLEND_BOTHINVSRCALPHA	Microsoft® DirectX® 6.0 及后续版本已不再使用。可以通过将源混合因子和目标混合因子分别设为 D3DBLEND_INVSRCALPHA 和 D3DBLEND_SRCALPHA 达到相同的效果。
D3DBLEND_BLENDFACTOR	使用从 D3DRS_BLENDFACTOR 设定中得到的 color.r，color.g，color.b 和 color.a。
D3DBLEND_INVBLENDFACTOR	使用从 D3DRS_BLENDFACTOR 设定中得到的 1-color.r，1-color.g，1-color.b 和 1-color.a。

Microsoft Direct3D®使用 D3DRS_ALPHABLENDENABLE 渲染状态启用阿尔法透明度混合。根据 D3DRS_SRCBLEND 和 D3DRS_DESTBLEND 渲染状态决定要进行的阿尔法混合的类型。源和目标混合状态必须成对使用。以下示例代码将源混合状态设为 D3DBLEND_SRCCOLOR，将目标混合状态设为 D3DBLEND_INVSRCCOLOR。

```
// 本代码段假设 lpD3DDevice 为指向设备的有效指针。

// 启用阿尔法混合。
lpD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE,
                            TRUE);

// 设置源混合状态。
lpD3DDevice->SetRenderState(D3DRS_SRCBLEND,
                            D3DBLEND_SRCCOLOR);

// 设置目标混合状态。
lpD3DDevice->SetRenderState(D3DRS_DESTBLEND,
                            D3DBLEND_INVSRCCOLOR);
```

这段代码在源色（正在当前位置渲染的图元的颜色）和目标色（帧缓存中当前位置的颜色）间进行性线插值。结果有点像带色的玻璃，目标物体中的一些颜色像是穿过源物体，而其余的像是被吸收了。

这些混合因子中的许多都需要纹理中的阿尔法值才能正确地起作用。因此，在使用顶点或材质阿尔法时，应用程序受到限制而无法使用一些能产生有趣结果的模式。

渲染目标阿尔法

帧缓存混合器现在可以单独混合渲染目标的阿尔法通道和颜色通道。一个新的渲染状态 D3DRS_SEPARATEALPHABLENDENABLE 用来启用这个控制。

如果 D3DRS_SEPARATEALPHABLENDENABLE 被设为 FALSE（默认值），那么渲染目标的阿尔法通道使用的混合因子和操作与颜色通道使用的相同。驱动程序需要设置 D3DPMISCCAPS_SEPARATEALPHABLEND 能力位以表示它可以支持渲染目标的阿尔法混合。为了告诉流水线需要阿尔法混合，一定要启用 D3DRS_ALPHABLEND。

为了控制渲染目标混合器中阿尔法通道的因子，定义了两个新的渲染状态，如下所示：

D3DRS_SRCBLENDALPHA

D3DRS_DESTBLENDALPHA

和D3DRS_SRCBLEND和D3DRS_DESTBLEND相似，这两个渲染状态可以被设为D3DBLEND枚举类型值。对源和目的混合操作的设定可以按多种不同的方式组合，这取决于D3DCAPS9的*SrcBlendCaps*和*DestBlendCaps*成员的值。

阿尔法混合根据以下公式进行：

renderTargetAlpha = (alphain* srcBlendOp) BlendOp (alphart* destBlendOp)

此处：
alphain 为输入的阿尔法值。
srcBlendOp 为 D3DBLEND 定义的混合因子之一。
BlendOp 为 D3DBLEND 定义的混合因子之一。
alphart 为渲染目标中的阿尔法值。
destBlendOp 为 D3DBLEND 定义的混合因子之一。
renderTargetAlpha 为混合得到的最终的阿尔法值。
线段绘制
这个版本的 Direct3D 已经在单像素宽度的带锯齿线段（即非抗锯齿线段）中加入了对抗锯齿线段的支持。
要启用线段抗锯齿，需将D3DRS_ANTIALIASEDLINEENABLE渲染状态设置为TRUE。为得到最佳性能，它的默认值为FALSE。这既适用于以线框模式绘制的三角形，也适用于线段图元类型。当渲染到多重取样渲染目标时，这个渲染状态被忽略，所有的线段都按带锯齿线段渲染。（要在多重取样渲染目标中渲染抗锯齿线段，应用程序可以使用Direct3D扩充库（D3DX）对抗锯齿线段渲染的支持，扩展库会生成带纹理的多边形。）请参阅D3DX线段绘制。
对线段样式（line pattern）的支持已经移除，也不再支持 D3DRS_LINEPATTERN 渲染状态。
API 的改变
这是一个新的渲染状态和新的能力位。

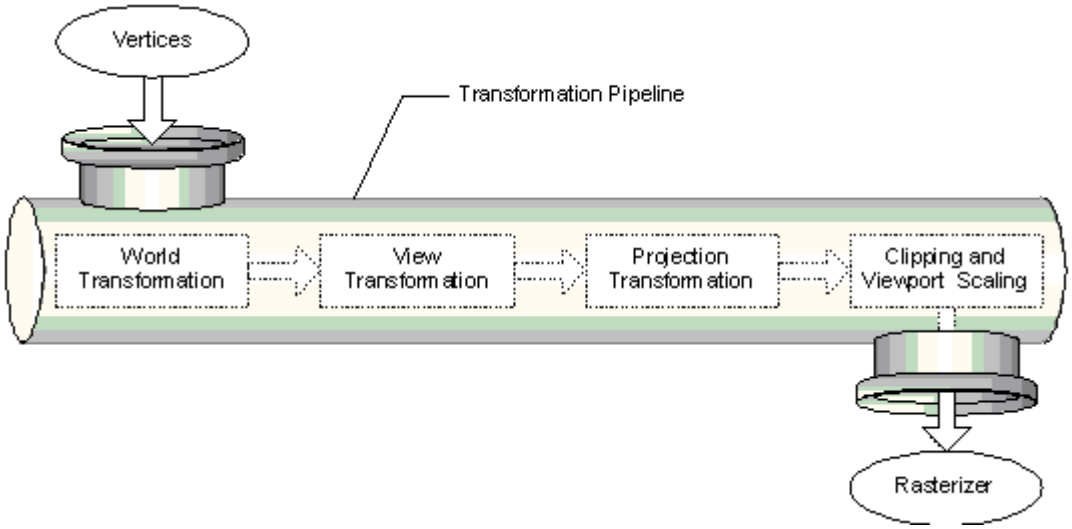
```
// 新的渲染状态
D3DRS_ANTIALIASEDLINEENABLE
// 可以为 TRUE (非零) 或 FALSE (零)

// 新的能力位
D3DLINECAPS_ANTIALIAS
```

有一些渲染状态也不再被支持。

```
// 已删除: D3DRS_LINEPATTERN
// 已删除: 结构 D3DLINEPATTERN
// 已删除: D3DPMISCCAPS_LINEPATTERNREP
```

固定功能流水线
Microsoft® Direct3D®中把几何体送经固定功能几何流水线进行处理的那部分是变换引擎。它取得三维模型和观察者在世界空间中的位置，把顶点投影到屏幕上，并根据视区对顶点进行裁剪。变换引擎也对每个顶点执行光照计算，得到漫反射色和镜面反射色。
顶点和像素处理
顶点格式
变换
视区和裁剪
光照和材质
纹理
顶点和像素处理
Microsoft® Direct3D®中把几何体送经固定功能几何流水线进行处理的那部分是变换引擎。它取得三维模型和观察者在世界空间中的位置，把顶点投影到屏幕上，并根据视区对顶点进行裁剪。变换引擎也对每个顶点执行光照计算，得到漫反射色和镜面反射色。

几何流水线接收顶点作为输入。变换引擎将三个变换——世界、观察和投影变换——应用于顶点，对结果进行裁剪，将所有数据传递给光栅化器。下图描绘了这些步骤的顺序。



在流水线的开始处，没有进行任何变换，因此所有建模的顶点是相对于局部坐标系——这是一个局部原点和方向——声明的。这个坐标系通常被称为建模空间，单个的坐标被称为建模坐标。

几何流水线的第一步将建模顶点从局部坐标系变换到场景中所有物体都使用的一个坐标系。重定位顶点的过程被称为世界变换。新的坐标系通常被称为世界空间，世界空间中的顶点用世界坐标表示。

下一步，描述三维世界的顶点根据摄像机进行重定位。也就是说，应用程序在场景中选择一个视点，世界空间中的坐标被重定位并围绕摄像机的视线旋转，从世界空间转变到摄像机空间。这就是视变换。

下一步是投影变换。在流水线的这一部分，为了营造场景的立体感，通常要根据物体与观察者之间的距离对它们进行缩放，使得近的物体显得比远处的物体大一些，等等。为了简单，本文档将经过变换的顶点所在的空间称为投影空间。一些图形学书籍可能将投影空间称为后透视齐次空间（post-perspective homogeneous space）。并不是所有的投影变换都会对场景中物体的大小进行缩放，这样的投影有时被称为仿射或正交投影。

在流水线的最后一部分，任何在场景中不可见的顶点被移除，这样光栅化器就不必花时间去为一些不可见的东西计算颜色及进行明暗处理，这个过程被称为裁剪。经过裁剪后，剩下的顶点根据视区的参数进行缩放并转换到屏幕坐标。最后得到的顶点——在屏幕上看见的经过光栅化的场景——存在于屏幕空间中。

传统FVF格式

传统 FVF 格式

为了使用传统的弹性顶点格式（FVF）的格式，应该在调用IDirect3DDevice9::SetFVF方法时用一个FVF码作为参数，如以下示例代码所示。

```
g_d3dDevice->SetFVF( CUSTOM_FVF );
```

以下主题涵盖了传统 FVF 格式的类型。

Vertex传统类型

LVertex传统类型

TLVertex传统类型

Vertex 传统类型

本主题说明了初始化并使用包含位置、法向和纹理坐标的顶点所需的步骤。

第一步是定义一个自定义顶点类型和 FVF 码，如以下示例代码所示。

```
struct Vertex
{
    FLOAT x, y, z;
    FLOAT nx, ny, nz;
    FLOAT tu, tv;
};

const DWORD VertexFVF = ( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1 );
```
下一步通过调用IDirect3DDevice9::CreateVertexBuffer方法创建一个有足够空间包含四个顶点的顶点缓存，如以下示例代码所示。
```
g_d3dDevice->CreateVertexBuffer(
                4*sizeof(Vertex), VertexFVF,
                D3DUSAGE_WRITEONLY,
                D3DPOOL_DEFAULT, &pBigSquareVB);
```
下一步是设置每个顶点的值，如以下示例代码所示。
```
Vertex * v;
pBigSquareVB->Lock( 0, 0, (BYTE**)&v, 0 );

v[0].x  = 0.0f;  v[0].y  = 10.0;  v[0].z  = 10.0f;
v[0].nx = 0.0f;  v[0].ny = 1.0f;  v[0].nz = 0.0f;
v[0].tu = 0.0f;  v[0].tv = 0.0f;

v[1].x  = 0.0f;  v[1].y  = 0.0f;  v[1].z  = 10.0f;
v[1].nx = 0.0f;  v[1].ny = 1.0f;  v[1].nz = 0.0f;
v[1].tu = 0.0f;  v[1].tv = 0.0f;

v[2].x  = 10.0f; v[2].y  = 10.0f; v[2].z  = 10.0f;
v[2].nx = 0.0f;  v[2].ny = 1.0f;  v[2].nz = 0.0f;
v[2].tu = 0.0f;  v[2].tv = 0.0f;

v[3].x  = 0.0f; v[3].y  = 10.0f;  v[3].z = 10.0f;
v[3].nx = 0.0f; v[3].ny = 1.0f;   v[3].nz = 0.0f;
v[3].tu = 0.0f; v[3].tv = 0.0f;

pBigSquareVB->Unlock();
```
至此顶点缓存已经初始化结束，可以准备渲染了。以下示例代码显示了如何使用传统的弹性顶点格式（FVF）绘制一个正方形。
```
g_d3dDevice->SetFVF( VertexFVF );
g_d3dDevice->SetStreamSource( 0, pBigSquareVB, 4*sizeof(Vertex) );
g_d3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0 ,2);
```
把一个FVF传递给IDirect3DDevice9::SetFVF会指定一个传统的FVF，顶点数据在数据流 0 中。
LVertex 传统类型
本主题说明了初始化并使用包含位置、漫反射色、镜面反射色和纹理坐标的顶点所需的步骤。
第一步是定义自定义顶点类型和弹性顶点格式（FVF），如以下示例代码所示。

```
struct LVertex
{
    FLOAT    x, y, z;
    D3DCOLOR specular, diffuse;
    FLOAT    tu, tv;
};

const DWORD VertexFVF = (D3DFVF_XYZ | D3DFVF_DIFFUSE |
                         D3DFVF_SPECULAR | D3DFVF_TEX1 );
```

下一步通过调用IDirect3DDevice9::CreateVertexBuffer创建一个有足够空间包含四个顶点的顶点缓存，如以下示例代码所示。

```
g_d3dDevice->CreateVertexBuffer( 4*sizeof(LVertex), VertexFVF,
                 D3DUSAGE_WRITEONLY, D3DPOOL_DEFAULT, &pBigSquareVB);
```

下一步是设置每个顶点的值，如以下示例代码所示。

```
LVertex * v;
pBigSquareVB->Lock( 0, 0, (BYTE**)&v, 0 );

v[0].x  = 0.0f;  v[0].y  = 10.0;  v[0].z  = 10.0f;
v[0].diffuse  = 0xffff0000;
v[0].specular = 0xff00ff00;
v[0].tu = 0.0f;  v[0].tv = 0.0f;

v[1].x  = 0.0f;  v[1].y  = 0.0f;  v[1].z  = 10.0f;
v[1].diffuse  = 0xff00ff00;
v[1].specular = 0xff00ffff;
v[1].tu = 0.0f;  v[1].tv = 0.0f;

v[2].x  = 10.0f; v[2].y  = 10.0f; v[2].z  = 10.0f;
v[2].diffuse  = 0xffff00ff;
v[2].specular = 0xff000000;
v[2].tu = 0.0f;  v[2].tv = 0.0f;

v[3].x  = 0.0f; v[3].y  = 10.0f;  v[3].z = 10.0f;
v[3].diffuse  = 0xffffff00;
v[3].specular = 0xffff0000;
v[3].tu = 0.0f; v[3].tv = 0.0f;

pBigSquareVB->Unlock();
```

至此顶点缓存已经初始化完成，可以准备渲染了。以下示例代码显示了如何使用传统 FVF 绘制一个正方形。

```
g_d3dDevice->SetFVF( VertexFVF );
g_d3dDevice->SetStreamSource( 0, pBigSquareVB, 4*sizeof(LVertex) );
g_d3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0 ,2);
```

把一个FVF传递给IDirect3DDevice9::SetFVF方法会指定一个传统FVF，并且数据流 0 是唯一有效

的数据流。
TLVertex 传统类型
本主题说明了初始化并使用包含经过变换的位置、漫反射色、镜面反射色和纹理坐标的顶点所需的步骤。
第一步是定义自定义顶点类型和弹性顶点格式（FVF），如以下示例代码所示。

```
struct TLVertex
{
    FLOAT    x, y, z, rhw;
    D3DCOLOR specular, diffuse;
    FLOAT    tu, tv;
};

const DWORD VertexFVF = (D3DFVF_XYZRHW | D3DFVF_DIFFUSE |
                         D3DFVF_SPECULAR | D3DFVF_TEX1 );
```

下一步通过调用IDirect3DDevice9::CreateVertexBuffer方法创建一个有足够空间包含四个顶点的顶点缓存，如以下示例代码所示。

```
g_d3dDevice->CreateVertexBuffer( 4*sizeof(TLVertex), VertexFVF,
                 D3DUSAGE_WRITEONLY, D3DPOOL_DEFAULT, &pBigSquareVB);
```

下一步设置每个顶点的值，如以下示例代码所示。

```
TLVertex * v;
pBigSquareVB->Lock( 0, 0, (BYTE**)&v, 0 );

v[0].x  = 0.0f;  v[0].y  = 10.0;  v[0].z  = 10.0f; v[0].rhw = 1.0f;
v[0].diffuse  = 0xffff0000;
v[0].specular = 0xff00ff00;
v[0].tu = 0.0f;  v[0].tv = 0.0f;

v[1].x  = 0.0f;  v[1].y  = 0.0f;  v[1].z  = 10.0f; v[1].rhw = 1.0f;
v[1].diffuse  = 0xff00ff00;
v[1].specular = 0xff00ffff;
v[1].tu = 0.0f;  v[1].tv = 0.0f;

v[2].x  = 10.0f; v[2].y  = 10.0f; v[2].z  = 10.0f; v[2].rhw = 1.0f;
v[2].diffuse  = 0xffff00ff;
v[2].specular = 0xff000000;
v[2].tu = 0.0f;  v[2].tv = 0.0f;

v[3].x  = 0.0f; v[3].y  = 10.0f;  v[3].z = 10.0f; v[3].rhw = 1.0f;
v[3].diffuse  = 0xffffff00;
v[3].specular = 0xffff0000;
v[3].tu = 0.0f; v[3].tv = 0.0f;

pBigSquareVB->Unlock();
```

至此顶点缓存已经初始化完成，可以准备渲染了。以下示例代码显示了如何使用传统 FVF 绘制一

个正方形。

```
g_d3dDevice->SetFVF( VertexFVF );
g_d3dDevice->SetStreamSource( 0, pBigSquareVB, 4*sizeof(TLVertex) );
g_d3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0 ,2);
```

把一个FVF传递给IDirect3DDevice9::SetFVF会指定一个传统的FVF，顶点数据在数据流 0 中。

变换

变换用于将几何体从一个坐标系转换到另一个坐标系。最常用的变换都是用矩阵完成的。矩阵是保存变换值并将之应用于数据的基本工具。以下主题介绍矩阵，并解释如何用它们生成世界、观察和投影变换。

矩阵

世界变换

从世界坐标转换到观察坐标

观察变换

从观察坐标转换到投影坐标

投影变换

从投影坐标转换到屏幕坐标

矩阵

Microsoft® Direct3D®用矩阵进行三维变换。本节解释如何用矩阵创建三维变换，描述变换的一些通常用法，以及如何合并矩阵生成单个包含多重变换的矩阵的细节。信息被分为以下主题。

概述

三维变换

矩阵串接

旋转

平移和缩放

三维变换

在使用三维图形的应用程序中，可以用变换做以下事情：

描述一个物体相对于另一个物体的位置。

旋转并改变物体的大小。

改变观察的位置、方向和视角。

可以用一个 4 x 4 矩阵将任意点（x，y，z）变换为另一个点（x’，y’，z’）。

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

对（x，y，z）和矩阵执行以下操作产生点（x’，y’，z’）。

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

最常见的变换是平移、旋转和缩放。可以将产生这些效果的矩阵合并成单个矩阵，这样就可以一次计算多种变换。例如，可以构造单个矩阵，对一系列的点进行平移和旋转。更多信息，请参阅

矩阵串接。

矩阵以行列顺序书写。一个沿每根轴均匀缩放顶点的矩阵，也称为统一缩放，用如下数学符号表示。

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

在C++应用程序中，Microsoft® Direct3D®使用D3DMATRIX结构，将矩阵声明为一个二维数组。以下示例代码显示了如何初始化一个D3DMATRIX结构，使之成为一个统一缩放矩阵。

```
// 本例中，s 为浮点类型的变量

D3DMATRIX scale = {
    s,              0.0f,           0.0f,           0.0f,
    0.0f,           s,              0.0f,           0.0f,
    0.0f,           0.0f,           s,              0.0f,
    0.0f,           0.0f,           0.0f,           1.0f
};
```

平移和缩放

平移

以下变换将点（x，y，z）平移到一个新的点（x’，y’，z’）。

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

可以在 C++应用程序中手工创建一个平移矩阵。以下示例代码显示了的一个函数的源码，该函数创建一个矩阵用于平移顶点。

```
D3DXMATRIX Translate(const float dx, const float dy, const float dz) {
    D3DXMATRIX ret;

    D3DXMatrixIdentity(&ret);  // 由 Direct3DX 实现
    ret(3, 0) = dx;
    ret(3, 1) = dy;
    ret(3, 2) = dz;
    return ret;
}    // 平移结束
```

为了方便，Direct3DX工具库提供了D3DXMatrixTranslation函数。

缩放

以下变换用指定值缩放点（x，y，z）的 x-，y-和 z-方向，产生新的点（x’，y’，z’）。

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

旋转

这里描述的变换是基于左手坐标系的，也许和别处见到的变换矩阵不同。更多信息，请参阅三维坐标系。

以下变换将点（x，y，z）围绕 x 轴旋转，产生新的点（x'，y'，z'）。

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

以下变换将点围绕 y 轴旋转。

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

以下变换将点围绕 z 轴旋转。

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

在这些示例矩阵中，希腊字母（θ）表示旋转的角度，以弧度为单位。当沿着旋转轴朝原点看时，角度以顺时针方向计量。

C++应用程序可以用Direct3D扩展（D3DX）工具库提供的D3DXMatrixRotationX，D3DXMatrixRotationX和D3DXMatrixRotationX函数创建旋转矩阵。以下示例是D3DXMatrixRotationX函数的源码。

```
D3DXMATRIX* WINAPI D3DXMatrixRotationX
    ( D3DXMATRIX *pOut, float angle )
{
#if DBG
    if(!pOut)
        return NULL;
#endif

    float sin, cos;
    sincosf(angle, &sin, &cos);  // 计算角度的正弦和余弦值。

    pOut->_11 = 1.0f; pOut->_12 =  0.0f;    pOut->_13 = 0.0f; pOut->_14 = 0.0f;
    pOut->_21 = 0.0f; pOut->_22 =  cos;     pOut->_23 = sin;  pOut->_24 = 0.0f;
```

```
    pOut->_31 = 0.0f; pOut->_32 = -sin;    pOut->_33 = cos;  pOut->_34 = 0.0f;
    pOut->_41 = 0.0f; pOut->_42 =  0.0f;   pOut->_43 = 0.0f; pOut->_44 = 1.0f;

    return pOut;
}
D3DXMATRIX* WINAPI D3DXMatrixRotationX
    ( D3DXMATRIX *pOut, float angle )
{
#if DBG
    if(!pOut)
        return NULL;
#endif

    float sin, cos;
    sincosf(angle, &sin, &cos);  // Determine sin and cos of angle.

    pOut->_11 = 1.0f; pOut->_12 =  0.0f;   pOut->_13 = 0.0f; pOut->_14 = 0.0f;
    pOut->_21 = 0.0f; pOut->_22 =  cos;    pOut->_23 = sin;  pOut->_24 = 0.0f;
    pOut->_31 = 0.0f; pOut->_32 = -sin;    pOut->_33 = cos;  pOut->_34 = 0.0f;
    pOut->_41 = 0.0f; pOut->_42 =  0.0f;   pOut->_43 = 0.0f; pOut->_44 = 1.0f;

    return pOut;
}
```

矩阵串接

使用矩阵的一个优势就是可以通过把两个以上的矩阵相乘,将它们的效果合并在一起。这意味着，要先旋转一个建模然后把它平移到某个位置，无需使用两个矩阵，只要把旋转矩阵和平移矩阵相乘，产生一个包含了所有效果的合成矩阵。这个过程被称为矩阵串接，可以写成以下公式。

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

在这个公式中，C 是将被创建的合成矩阵，M1 到 Mn 是矩阵 C 包含的单个矩阵。虽然大多数情况下，只需要串接两三个矩阵，但实际上并没有数量上的限制。

可以使用D3DXMatrixMultiply函数进行矩阵乘法。

进行矩阵乘法时先后次序是至关重要的。前面的公式反映了矩阵串接从左到右的规则。也就是说，用来创建合成矩阵的每个矩阵产生的直观效果会按从左到右的次序出现。下面显示了一个典型的世界变换矩阵。想象一下给一个旋转飞行的碟子创建世界变换矩阵。应用程序也许想让飞行的碟子绕它的中心——建模空间中的 y 轴——旋转，然后把它平移到场景中的另一个位置。要实现这样的效果，首先创建一个旋转矩阵，然后将它与平移矩阵相乘，如以下公式所示。

$$W = R_y \cdot T_w$$

在这个公式中，Ry 是绕 y 轴的旋转矩阵，Tw 是平移到世界坐标中某个位置的矩阵。

矩阵相乘的顺序很重要，因为矩阵乘法是不可交换的，这和两个标量相乘不同。将矩阵以相反的顺序相乘会产生这样的直观效果：先把飞行中的碟子平移到世界空间中的某个位置，然后将它围绕世界坐标的原点旋转。

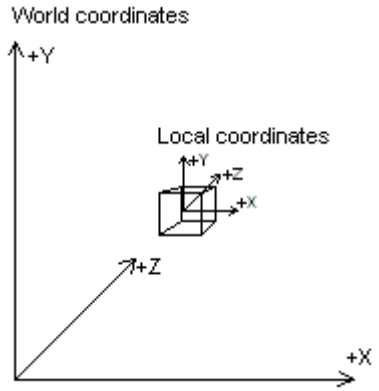
无论创建何种类型的矩阵，都要记住从左到右的规则，这样才能保证得到想要的效果。

世界变换

对世界变换的讨论介绍了基本概念，并提供了如何在 Microsoft® Direct3D®应用程序中设置世界变换矩阵的细节。

什么是世界变换

世界变换将坐标从建模空间,在这个空间中的顶点相对于建模的局部原点定义,转变到世界空间,在这个空间中的顶点相对于场景中所有物体共有的原点定义。本质上，世界变换将一个建模放到世界中，并由此而得名。下图描绘了世界坐标系统和建模的局部坐标系统间的关系。



世界变换可以包含任意数量的平移、旋转和缩放的合并。有关对变换的数学讨论，请参阅三维变换。

设置世界矩阵

同任何其它变换一样，通过将一系列变换矩阵串接成单个矩阵，应用程序可以创建包含这些矩阵全部效果的世界矩阵。最简单的情况下，建模在世界的原点并且它的局部坐标轴与世界空间的方向相同，这时世界矩阵就是单位矩阵。更通常的情况下，世界矩阵是一系列矩阵的合成，包含一个平移矩阵，并且根据需要可能有一个以上的旋转矩阵。

以下示例，来自一个用 C++编写的假想三维建模类，使用 Direct3D 扩展（D3DX）工具库提供的帮助函数创建了一个世界矩阵，这个世界矩阵包含了三个旋转矩阵，用于调整三维建模的方向，以及一个平移矩阵，用来根据建模在世界空间中的相对坐标重新确定它的位置。

```
/*
 * 根据本示例的目的，假设以下变量都是有效的并经过初始化。
*
 * 变量 m_xPos，m_yPos，m_zPos 包含了建模在世界坐标中的位置。
*
 * 变量 m_fPitch，m_fYaw 和 m_fRoll 为浮点数，包含了建模的方向，
 * 用 pitch，yaw 和 roll 旋转角表示，以弧度为单位。
 */

void C3DModel::MakeWorldMatrix( D3DXMATRIX* pMatWorld )
{
    D3DXMATRIX MatTemp;  // 用于旋转的临时矩阵
    D3DXMATRIX MatRot;   // 最终的旋转矩阵，应用于 pMatWorld.

    // 使用从左到右的矩阵串接顺序，在旋转之前对物体在世界空间中
    // 的位置进行平移。
```

```
    D3DXMatrixTranslation(pMatWorld, m_xPos, m_yPos, m_zPos);
    D3DXMatrixIdentity(&MatRot);

    // 现在将方向变量应用于世界矩阵
    if(m_fPitch || m_fYaw || m_fRoll) {
        // 产生并合成旋转矩阵。
        D3DXMatrixRotationX(&MatTemp, m_fPitch);          // Pitch
        D3DXMatrixMultiply(&MatRot, &MatRot, &MatTemp);
        D3DXMatrixRotationY(&MatTemp, m_fYaw);            // Yaw
        D3DXMatrixMultiply(&MatRot, &MatRot, &MatTemp);
        D3DXMatrixRotationZ(&MatTemp, m_fRoll);           // Roll
        D3DXMatrixMultiply(&MatRot, &MatRot, &MatTemp);

        // 应用旋转矩阵，得到最后的世界矩阵。
        D3DXMatrixMultiply(pMatWorld, &MatRot, pMatWorld);
    }
}
```

当准备好世界变换矩阵后，应该调用IDirect3DDevice9::SetTransform方法设置它，并把第一个参数指定D3DTS_WORLD宏。

注意Direct3D 使用应用程序设置的世界和观察矩阵配置许多内部数据结构。应用程序每次设置新的世界或观察矩阵时，系统都要重新计算相关的内部数据结构。频繁地设置这些矩阵——例如，每帧上千次——是计算量很大的。通过将世界矩阵和观察矩阵串接成一个世界/观察矩阵，并将该矩阵设置为世界矩阵，然后将观察矩阵设置为单位矩阵，应用程序可以将所需的计算次数减到最少。最好保存一份单独的世界矩阵和观察矩阵的副本在高速缓存中，这样就可以根据需要修改、串接及重置世界矩阵。为清晰起见，本文档中的 Direct3D 示例很少使用这项优化。

观察变换

本节介绍观察变换的基本概念，并提供有关如何在 Microsoft® Direct3D®应用程序中设置观察矩阵的细节。信息被分为以下主题。

什么是观察变换？

设置观察矩阵

什么是观察变换？

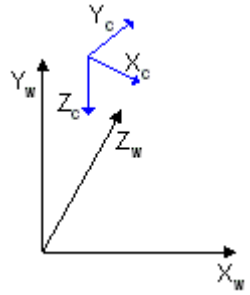
观察变换根据观察者在世界空间中的位置，把顶点变换到摄像机空间。在摄像机空间中，摄像机，或观察者，位于原点，朝 sz 轴的正向看去。再次提醒一下，因为 Direct3D 使用左手坐标系，所以 z 轴的正向是朝着场景的。观察矩阵根据摄像机的位置——摄像机空间的原点——和方向重定位世界中的所有物体。

有两方法可以创建观察矩阵。所有情况下，摄像机在世界空间中的逻辑位置和方向会被用作起始点来创建观察矩阵，得到的观察矩阵会被应用于场景中的三维建模。观察矩阵平移并旋转物体，将它们放入摄像机空间中，摄像机位于原点。创建观察矩阵的一种方法是把平移矩阵和围绕每根坐标轴旋转的旋转矩阵合并。这种方法使用了以下通用矩阵公式。

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

在这个公式中，*V*是要创建的观察矩阵，*T*是在世界中重定位物体的平移矩阵，*R*x 到 *R*z 分别是绕 x 轴，y 轴和 z 轴旋转物体的旋转矩阵。平移和旋转矩阵基于摄像机在世界空间中逻辑位置和方向。因此，如果摄像机在世界中的逻辑位置是<10，20，100>，那么平移矩阵的目的是沿 x 轴

移动物体-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摄像机的方向，根据摄像机空间的坐标轴与世界空间的坐标轴间的夹角决定。例如，如果前面提到的摄像机是垂直向下放的，那么它的 z 轴与世界空间的 z 轴有 90 度夹角，如下图所示。



旋转矩阵将角度相同但方向相反的旋转量应用于场景中的建模。这个摄像机的观察矩阵包含了一个绕 x 轴-90 度的旋转。旋转矩阵与平移矩阵合并生成观察矩阵，观察矩阵调整物体在场景中的位置和方向，使它们的顶部朝着摄像机，看起来就好像摄像机在建模的上方一样。

设置观察矩阵

D3DXMatrixLookAtLH和D3DXMatrixLookAtRH辅助函数根据摄像机的位置和被观察点创建一个观察矩阵。

以下示例代码创建了一个用于右手系的观察矩阵。

```
D3DXMATRIX out;
D3DXVECTOR3 eye(2,3,3);
D3DXVECTOR3 at(0,0,0);
D3DXVECTOR3 up(0,1,0);
D3DXMatrixLookAtRH(&out, &eye, &at, &up);
```

Direct3D 使用应用程序设置的世界矩阵和观察矩阵配置许多内部数据结构。每次应用程序设置一个新的世界矩阵或观察矩阵，系统都要重新计算相关的内部数据结构。频繁地设置这些矩阵——例如，每帧 20,000 次——计算量非常大。通过将世界矩阵和观察矩阵串接成一个世界/观察矩阵，并将之设置为世界矩阵，然后将观察矩阵设为单位矩阵，应用程序可以将所需的计算量减到最小。最好保存一份单独的世界矩阵和观察矩阵的副本在高速缓存中，这样就可以根据需要修改、串接及重置世界矩阵。为清晰起见，Direct3D 示例很少使用这项优化。

投影变换

可以认为投影变换是控制摄像机的内部参数，这选择摄像机的镜头有些相似。这是三种类型的变换中最为复杂的。对投影变换的讨论被分为以下主题。

什么是投影变换？

设置投影矩阵

W友好投影矩阵

什么是投影变换？

典型的投影变换就是一个缩放和透视投影。投影变换将视棱锥转变为一个立方体。因为视棱锥的近端比远端小，所以这就产生了离摄像机近的物体被放大的效果，这就是透视如何被应用于场景的。

在视棱锥中，摄像机与观察变换空间的原点之间的距离被定义为*D*，因此投影矩阵看起来是这样：

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

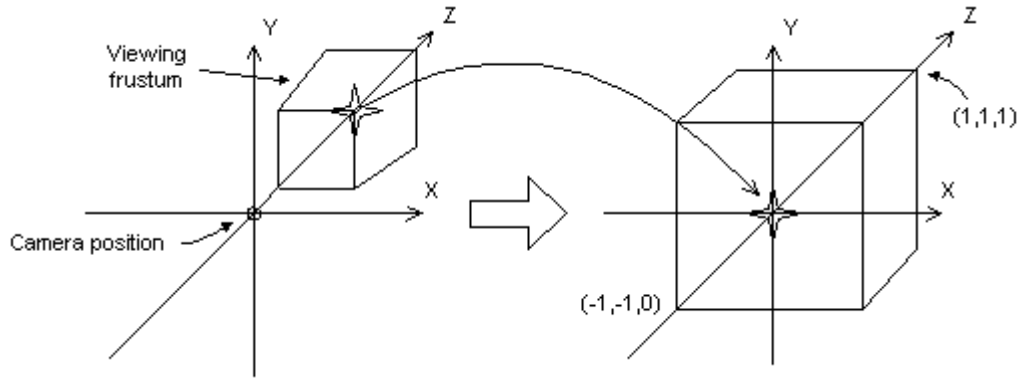
通过在 z 方向平移-*D*，观察矩阵将摄像机平移到原点。平移矩阵如下所示：

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

将平移矩阵与投影矩阵相乘（T*P），得到合成的投影矩阵。如下：

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

下图描绘了透视投影如何将视棱锥转变到新的坐标空间。注意棱锥变成了立方体，同时原点从场景的右上角移到了中心。（译注：应该是从前裁剪平面的中心移到了原点）



在透视变换中，x 和 y 方向的边界值是-1 和 1。Z 方向的边界值分别是，0 对应于前平面，1 对应于后平面。

这个矩阵根据指定的从摄像机到近裁剪平面的距离，平移并缩放物体，但没有考虑视角（*fov*），并且用它为远处物体产生的 z 值可能几乎相同，这使深度比较变得困难。以下矩阵解决了这些问题，并根据视区的纵横比调整顶点，这使它成为透视投影的一个很好的选择。

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

在这个矩阵中，Z_n 是近裁剪平面的 z 值。变量 *w*，*h* 和 *Q* 有以下含义。注意 *fov*w 和 *fov*h 表示视区在水平和垂直方向上的视角，以弧度为单位。

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

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

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

对应用程序而言，使用视角的角度定义 x 和 y 的比例系数可能不如使用视区在水平和垂直方向上的大小（在摄像机空间中）方便。可以用数学推导，得出下面两个使用视区大小计算 *w* 和 *h* 的公式，它们与前面的公式是等价的。

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

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

在这两个公式中，*Z*n 表示近裁剪平面的位置，*V*w 和 *V*h 变量表示视区在摄像机空间的宽和高。
对于C++应用程序而言，这两个大小直接对应于D3DVIEWPORT9结构的Width和Height成员。（译注：虽然直接对应，但是不等价的，因为*V*w和*V*h位于摄像机空间，而Width和Height位于屏幕空间）
无论决定使用什么公式，非常重要的一点是要尽可能将 *Z*n 设得大，因为接近摄像机的 z 值变化不大。这使得用 16 位 z 缓存的深度比较变得有点复杂。
同世界变换和观察变换一样，应用程序调用IDirect3DDevice9::SetTransform方法设置投影矩阵。
设置投影矩阵
以下ProjectionMatrix示例函数设置了前后裁剪平面，以及在水平和垂直方向上视角的角度。此处的代码与什么是投影矩阵？ 主题中讨论的方法相似。视角应该小于弧度 π 。

```
D3DXMATRIX
ProjectionMatrix(const float near_plane, // 到近裁剪平面的距离
                 const float far_plane,  // 到远裁剪平面的距离
                 const float fov_horiz,  // 水平视角，用弧度表示
                 const float fov_vert)   // 垂直视角，用弧度表示
{
    float    h, w, Q;

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

    D3DXMATRIX ret;
    ZeroMemory(&ret, sizeof(ret));

    ret(0, 0) = w;
    ret(1, 1) = h;
    ret(2, 2) = Q;
    ret(3, 2) = -Q*near_plane;
```

```
    ret(2, 3) = 1;
    return ret;
}   // End of ProjectionMatrix
```

在创建矩阵之后，调用IDirect3DDevice9::SetTransform方法设置投影矩阵，要将第一个参数设为D3DTS_PROJECTION。

Direct3D 扩展（D3DX）工具库提供了以下函数，帮助应用程序设置投影矩阵。

D3DXMatrixPerspectiveLH

D3DXMatrixPerspectiveRH

D3DXMatrixPerspectiveFovLH

D3DXMatrixPerspectiveFovRH

D3DXMatrixPerspectiveOffCenterLH

D3DXMatrixPerspectiveOffCenterRH

W 友好投影矩阵

对于已经用世界、观察和投影矩阵变换过的顶点，Microsoft® Direct3D®使用顶点的W成员进行深度缓存中基于深度的计算或计算雾效果。类似这样的计算要求应用程序归一化投影矩阵，使生成的W与世界空间中的Z相同。简而言之，如果应用程序的投影矩阵包含的（3，4）系数不为 1，那么为了生成适用的矩阵，应用程序必须用（3，4）系数的倒数缩放所有的系数。如果应用程序不提供符合这样要求的矩阵，那么会造成雾效果和深度缓存不正确。什么是投影矩阵？ 中推荐的矩阵符合基于w的计算的要求。

下图显示了不符合要求的投影矩阵，以及对同一个矩阵进行缩放，这样就可以启用基于视点的雾。

Non-compliant

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

Compliant

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

我们假设在前面的矩阵中，所有变量都不为零。更多有关相对于视点的雾的信息，请参阅基于视点的深度与基于Z的深度的比较。更多有关基于w的深度缓存，请参阅深度缓存。

注意Direct3D 在基于 w 的深度计算中使用当前设置的投影矩阵。因此，即使应用程序不使用 Direct3D 变换流水线，但是为了使用基于 w 的特性，应用程序必须设置一个符合要求的投影矩阵。

视区和裁剪

从概念上讲，视区是一个二维矩形，三维场景被投影到这个矩形中。在 Microsoft® Direct3D® 中，这个矩形以 Direct3D 表面内的坐标的形式存在，该表面被系统用作渲染目标。投影变换把顶点转换到视区所使用的坐标系统。

应用程序用 Direct3D 中的视区指定以下特性。

屏幕空间中的视区，渲染将被限制在该区域内。

渲染目标表面的深度值范围（通常是 0.0 到 1.0），在这之间的场景会被渲染。

本节讨论裁剪，即几何流水线的最后一步。讨论被分为以下主题。

视区矩形

视棱锥

裁剪体

视区缩放

视区的使用

Direct3D 通过给设备设置一系列的视区参数来实现裁剪。

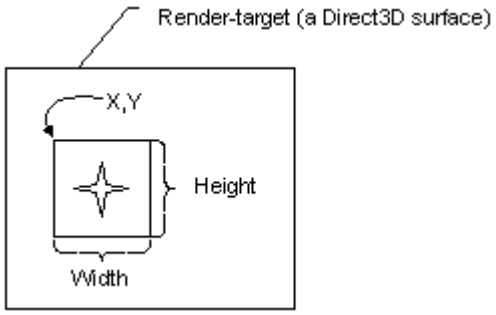
视区矩形

C++应用程序通过使用D3DVIEWPORT9结构定义视区矩形。D3DVIEWPORT9 结构要配合以下IDirect3DDevice9接口暴露的视区操作方法一起使用。

IDirect3DDevice9::GetViewport

IDirect3DDevice9::SetViewport

D3DVIEWPORT9 结构包含的四个成员——X，Y，Width 和 Height——定义了渲染目标表面中的一块区域，场景将被渲染在该区域中。这些值对应于目标矩形，或视区矩形，如下图所示。



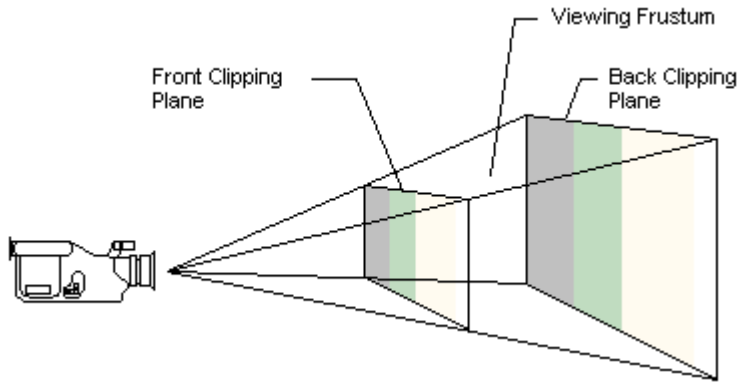
应用程序给 D3DVIEWPORT9 结构的 X，Y，Width 和 Height 成员指定的值是屏幕坐标，相对于渲染目标表面的左上角。该结构定义了两个附加成员（MinZ 和 MaxZ），指出在这个深度范围内的场景会被渲染。

Microsoft® Direct3D®假设视区裁剪体在X方向上从-1.0 到 1.0，在Y方向上从 1.0 到-1.0。过去，这些是应用程序最常使用的设定。在投影变换过程中，应用程序可以在裁剪前调整纵横比。这项任务被涵盖在投影变换一节中。

注意D3DVIEWPORT9 结构的 MinZ 和 MaxZ 成员指出在这个深度范围内的场景会被渲染，它们不用于裁剪。大多数应用程序把这两个成员设置为 0.0 和 1.0，使系统能渲染深度缓存中全部范围的深度值。在一些情况下，应用程序可以通过使用其它的深度范围实现特殊效果。例如，要在游戏中显示提示（heads-up display），应用程序可以将两个值都设为 0.0，强制系统渲染场景中最前面的物体，或者也可以将它们都设为 1.0，渲染总是在最后面的物体。

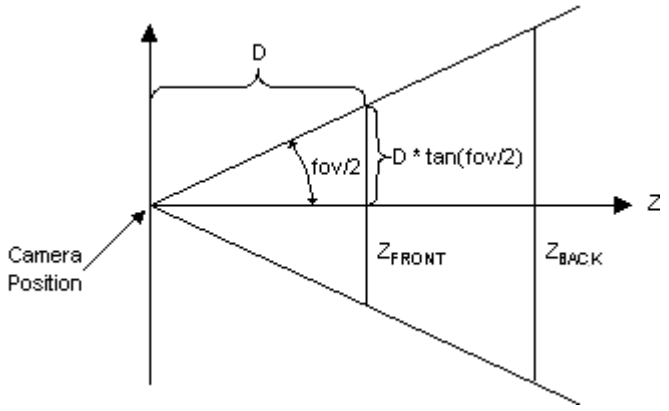
视棱锥

视棱锥是场景中的三维区域，相对于视区的摄像机。区域的形状会影响到建模如何从摄像机空间投影到屏幕。最普通的投影类型，透视投影，是造成离摄像机近的物体显得比远处物体大的原因。对于透视变换，视棱锥可以被想象成一个金字塔，摄像机被放在顶端。这个金字塔分别被前后裁剪平面截取。金字塔中部位于前后裁剪平面之间的区域就是视棱锥。物体只有在这个区域中时才是可见的。



想象我们站在一个黑暗的房间里，从一个方的窗口中看出去，这就是一个直观的视棱锥。在这个类比中，近裁剪平面就是窗口，远裁剪平面是最终妨碍我们视线的任何东西——街上的高楼，远处的山，或什么也不是。我们可以看见从窗口处开始，到任何妨碍我们视线的东西处结束，位于截棱锥内的一切，除此之外我们什么也看不见。

视棱锥由 *fov*（视角）和视点在 z 轴方向上到前后裁剪平面的距离定义。



在这幅图中，变量*D*是从摄像机到空间的原点的距离，该空间由几何流水线的上一步——视变换——定义。这就是用于限定视棱锥区域的边界。有关如何用变量*D*构建一个投影矩阵的信息，请参阅什么是投影矩阵？。

裁剪体

投影矩阵的结果决定了投影空间中的裁剪体。Microsoft® Direct3D®将裁剪体定义为以下公式：

$$-W_c < X_c \le W_c$$
$$-W_c < Y_c \le W_c$$
$$0 < Z_c \le W_c$$

在前面的公式中，*X*，*Y*，*Z*和 *W*表示经过投影变换的顶点坐标。如果裁剪被启用（默认值），任何 x-，y-，或 z-成员在这些范围之外的顶点将被裁剪掉。

除了顶点缓存，应用程序通过D3DRS_CLIPPING渲染状态启用或禁用裁剪。顶点缓存的裁剪信息在处理过程中产生，更多信息请参阅固定功能顶点处理和可编程顶点处理。

Direct3D不裁剪顶点缓存中经过变换的顶点，除非顶点是由IDirect3DDevice9::ProcessVertices得到的。如果应用程序自己进行变换并需要Direct3D进行裁剪，那么不应该使用顶点缓存。在这种情况下，应用程序遍历顶点数据进行变换，Direct3D 再次遍历顶点数据进行裁剪，然后驱动程序渲染顶点数据，这是效率很低的。因此，如果应用程序自己变换顶点数据那么它同时也应该裁剪顶点数据。

当设备需要对收到的经过变换且经过光照的顶点（TL 顶点）进行裁剪时，为了执行裁剪操作，Direct3D 会根据顶点的 rhw 和视区信息，把顶点变换回裁剪空间。然后执行裁剪操作。并不是所有设备都能执行这种裁剪 TL 顶点所需的反向变换操作。

D3DPMISCCAPS_CLIPTLVERTS 设备能力标志指出设备是否能裁剪 TL 顶点。如果没有设置这个能力标志，那么应用程序应该负责对准备送到设备进行渲染的 TL 顶点进行裁剪。在软件顶点处理模式（无论设备自身是在软件顶点处理模式下创建，还是用 D3DRS_SOFTWAREVERTEXPROCESSING 渲染状态作为混合模式顶点处理设备创建然后切换到软件顶点处理模式）下，设备总是能裁剪 TL 顶点。

视区缩放

用于视区的D3DVIEWPORT9结构的X，Y，Width和Height成员定义了视区在渲染目标表面中的位置和大小。这些值是屏幕坐标，相对于表面的左上角。

Microsoft® Direct3D®使用视区的位置和大小缩放顶点使渲染后的场景正好在渲染目标表面的相应位置。在内部，Direct3D 会把这些值插入到应用于每个顶点的矩阵中。

$$\begin{bmatrix} dwWidth/2 & 0 & 0 & 0 \\ 0 & -dwHeight/2 & 0 & 0 \\ 0 & 0 & dvMaxZ-dvMinZ & 0 \\ dwX+dwWidth/2 & dwHeight/2+dwY & dvMinz & 1 \end{bmatrix}$$

这个矩阵根据视区的大小和希望的深度范围缩放顶点，并把它们转换到渲染目标表面的相应位置。矩阵同时翻转了 y-坐标，使屏幕原点在左上角，y 值向下增长。在应用这个矩阵之后，顶点仍然在齐次空间中——也就是说，它们仍然以[x,y,z,w]顶点的形式存在——并且它们在被送到光栅化器之前必须被转换到非齐次坐标。

注意 视区缩放矩阵合并了 D3DVIEWPORT9 结构的 MinZ 和 MaxZ 成员，使缩放后的顶点在深度范围[MinZ, MaxZ]内。这和以前版本的 Microsoft DirectX®具有不同的语义，以前这些成员曾被用于裁剪。

注意 更多信息，请参阅视区矩形和裁剪体。应用程序一般将MinZ和MaxZ设为 0.0 和 1.0，使系统渲染整个深度范围。应用程序可以把两个值都设为 0.0，强制渲染所有最前面的物体，或都设为 1.0,渲染所有最后面的物体。

视区的使用

本节提供有关使用视区的细节，信息被分为以下主题。

设置视区裁剪体

清除视区

手工变换顶点

设置视区裁剪体

要给一个渲染设备配置视区参数的唯一要求是设置视区的裁剪体。要完成这个任务，首先初始化裁剪体和渲染目标表面，并给它们设置裁剪值。视区一般被设置成渲染目标表面的整个区域，但这不是必需的。

C++应用程序可以对D3DVIEWPORT9结构的成员使用以下设置。

```
D3DVIEWPORT9 viewData = { 0, 0, width, height, 0.0f, 1.0f };
```

给D3DVIEWPORT9 结构赋值之后，通过调用IDirect3DDevice9::SetViewport方法会把视区参数应用于设备。以下示例代码显示了这个调用。

```
HRESULT hr;

hr = pd3dDevice->SetViewport(&viewData);
if(FAILED(hr))
    return hr;
```

如果调用成功，视区参数就设置完成并会在下一次渲染方法被调用时生效。要改变视区的参数，只要更改 D3DVIEWPORT9 结构的值并再次调用 IDirect3DDevice9::SetViewport 方法。

注意D3DVIEWPORT9 结构的 MinZ 和 MaxZ 成员指出在这个深度范围内的场景将被渲染，它们不用于裁剪。大多数应用程序将这两个成员设置为 0.0 和 1.0，使用系统渲染深度缓存中全部范围的深度值。在一些情况下，应用程序可以通过使用其它的深度范围实现特殊效果。例如，要在游戏中显示提示，应用程序可以把两个值都设为 0.0，强制系统渲染场景中最前面的物体，或者也可以把它们都设为 1.0，渲染总是在最后面的物体。

清除视区

清除视区会重置渲染目标表面中视区矩形内的内容，如果指定的话，还可以重置深度缓存和模板

缓存表面中相应矩形区域中的内容。一般来说，为了保证图形和其它数据（渲染目标表面和深度/模板缓存表面）都会接受新的渲染物体而不会显示遗留物，应用程序要在渲染新的一帧前清除视区。

C++开发人员可以用IDirect3DDevice9接口提供的IDirect3DDevice9::Clear方法清除视区。该方法接收一个或多个定义表面中需要被清除区域的矩形。万一正在渲染的场景包含穿越整个视区矩形的运动——例如，在一个第一人称视角的游戏中——应用程序可能希望每帧清除整个视区。在这种情况下，应用程序将*Count*参数设为 1，将*pRect*参数设为覆盖整个视区的矩形的地址。如果这样更方便，应用程序也可以将*pRect*参数设为NULL，将*Count*参数设为 0，表示要清除整个视区矩形。

IDirect3DDevice9::Clear 方法很灵活，它提供了对清除深度缓存中的模板缓存的支持。*Flags* 参数接收三个标志，确定怎样清除渲染目标及相关的深度缓存和模板缓存。如果 *Flags* 参数包含 D3DCLEAR_TARGET 标志，该方法用 *Color* 参数（不是材质颜色）中指定的 RGBA 颜色值清除视区。如果 *Flags* 参数包含 D3DCLEAR_ZBUFFER 标志，该方法用 *Z* 参数中指定的深度值清除深度缓存，0.0 是最近的距离，1.0 是最远的距离。如果 *Flags* 参数包含 D3DCLEAR_STENCIL 标志，该方法用 *Stencil* 参数中指定的值清除模板缓存。应用程序可以使用从 0 到 2 的 *n* 次方减 1 范围内的整数，*n* 是模板缓存的位数。

注意Microsoft DirectX® 5.0 允许背景材质具有关联的纹理，这可以将视区清除为一张纹理而不是一个单纯的颜色。这项特性很少使用，实际上效率也不高。从 DirectX® 6.0 开始及后续的接口不再接受纹理句柄，这意味着应用程序不能再用一张纹理清除视区。更确切地讲，应用程序现在必须手工绘制背景。因此，很少需要清除渲染目标表面上的视区。只要应用程序清除了深度缓存，渲染目标表面上的所有像素都将被覆盖。

在一些情况下，应用程序可能希望只渲染到渲染目标和深度缓存表面的一小部分。清除方法同样允许应用程序在单个调用中清除表面中的多个区域。要完成这个任务，将 *Count* 参数设置为希望清除的矩形区域的数量，并把 *pRects* 参数设为矩形数组中第一个矩形的地址。

手工变换顶点

在Microsoft® Direct3D®应用程序中，可以使用三种顶点。更多有关顶点格式的细节请参阅顶点格式。

应用程序可以用顶点缓存从简单顶点类型转到复杂顶点类型。顶点缓存是为了进行高速渲染，用来有效地存储并处理大批量顶点的对象，并经过优化可以充分利用特定处理器的特性。要用IDirect3DDevice9::ProcessVertices方法执行顶点变换。IDirect3DDevice9::ProcessVertices只接收未经变换的顶点，也可以有选择地对顶点执行光照计算和裁剪。光照计算在应用程序调用IDirect3DDevice9::ProcessVertices方法时执行，但裁剪在渲染时执行。

在处理完顶点之后，应用程序可以使用特殊的渲染方法渲染顶点，或者也可以通过锁定顶点缓存直接存取它们。更多有关使用顶点缓存的信息，请参阅顶点缓存。

光与材质

光用于照亮场景中的物体。当光照被启用时，Microsoft® Direct3D®根据下列组合计算每个顶点的颜色。

当前材质的颜色及相关纹理贴图中的纹理像素（texels）。

若已给出顶点的漫反射色和镜面反射色，则使用。

场景中光源产生的光的颜色和强度，或场景中环境光的级别。

当使用 Direct3D 光照和材质时，应用程序允许 Direct3D 代为处理照明的各种细节。如果需要的话，高级用户可以执行自己的光照计算。

应用程序如何使用光照和材质使渲染得到的场景有很大区别。材质定义了光在表面上如何反射。

直射光和环境光级别定义了反射的光。如果启用了光照，应用程序渲染场景时必须使用材质。对于渲染场景而言，光不是必须的，但是如果渲染的场景没有光，那么会有许多细节不可见。渲染一个未经光照的场景至多也只能得到场景中物体的轮廓，而这对大多数用途而言是不够的。

以下主题包含了更多信息。

Direct3D 光照模型与自然光的比较

在自然界，当光从光源发出后，在到达用户的眼睛之前，已经经过了成千上万个物体的反射。每经过一次反射，一部分光被表面吸收，一部分被散射到各个方向，而剩余的则被反射到其它表面或用户的眼睛里。这个过程一直持续，直到光衰减为零或到达用户眼中。

显然，要完美地模拟自然光所需的计算量太大以致于无法用在实时三维图形中。因此，考虑到速度，Microsoft® Direct3D®光照模型对自然光的工作方式进行模拟。Direct3D 用红、绿、蓝三原色描述光，并将它们合成产生最终的颜色。

在Direct3D中，当光从表面反射时，光的颜色与表面本身以某种数学方式相互作用，并产生最终显示在屏幕上的颜色。有关Direct3D使用的具体算法，请参阅与光照相关的数学。

Direct3D 光照模型将光归纳为两类：环境光和直射光。每种光具有不同的属性，并以不同的方式与表面材质相互作用。环境光是经过多次散射的光以至于它的方向和来源都无法确定：它给各处提供一个较低级别的光强。摄像师使用的非直射光是环境光的一个很好的例子。Direct3D 中的环境光，和自然界中的一样，没有实际的方向和光源，只有颜色和光强。事实上，环境光的级别完全独立于场景中的任何发光物体。环境光不参与镜面反射。

直射光是场景中的光源产生的光，它总是具有颜色和强度，并沿特定的方向传播。直射光与表面材质相互作用产生镜面反射高光（highlight），它的方向用作各种着色算法（包括高洛德着色算法）中的一个因子。当直射光反射时，它不影响场景中的环境光级别。场景中产生直射光的光源具有不同的特征，这些特征影响到光如何照亮场景。更多信息，请参阅光照与材质。

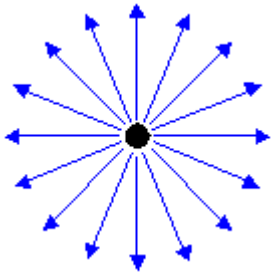
另外，多边形的材质具有一些属性，这些属性影响多边形如何反射它接收的光线。应用程序可以设置一个专门的反射系数，用于描述材质如何反射环境光，应用程序还可以设置另一个反射系数以决定材质的镜面反射和漫反射。更多信息请参阅材质。

光与材质的颜色值

Microsoft® Direct3D®用四个成员——红、绿、蓝和阿尔法——描述颜色，并将它们合成，产生最终颜色。Direct3D为C++程序定义了D3DCOLORVALUE结构，该结构包含了所有成员，每个成员是一个浮点数，一般在 0.0 到 1.0 范围内，闭区间。虽然光照和材质使用相同的结构描述颜色，但对结构中的值的使用略有不同。

光源的颜色值表示它发出的某种光成分的数量。因为光不包含阿尔法成员，所以只有颜色的红、绿和蓝成员是有用的。可以把三种成员想象成投影电视的红、绿、蓝镜头。每个镜头可能被关掉（相应成员的值为 0.0），或是位于范围内的某个值。镜头中照射出的颜色组合成光的最终颜色。如 R: 1.0, G: 1.0, B: 1.0 组合成白光，而 R: 0.0, G: 0.0, B: 0.0 则完全不发光。也可以产生只发出某个成员颜色的光，这样可以得到纯红、纯绿或纯蓝光，或者也可以将它们组合，得到黄色或紫色。甚至可以将颜色的成员设为负值，这样就产生了实际上将光从场景中移除的“暗光（dark light）”，得到场景变暗的效果。或者，也可以将成员设为大于 1.0 的值，产生特别亮

的光。
另一方面，材质的颜色值表示用该材质渲染的表面对某种光成分的反射度。颜色成员为R: 1.0, G: 1.0, B: 1.0, A: 1.0 的材质会反射所有的入射光。同样，成员为R: 0.0, G: 1.0, B: 0.0, A: 1.0 的材质会反射所有入射的绿光。具有多重反射系数值（译注：漫反射、镜面反射、等等）的材质可以创建不同类型的效果，更多信息，请参阅材质属性。
环境光的颜色值和用于直射光的光源及材质的颜色值是不一样的。更多信息，请参阅直射光与环境光的比较。
直射光与环境光的比较
虽然直射光与环境光都用于照亮场景中的物体，但它们互不相关，且具有非常不同的效果，并要按照完全不同的方法使用。
直射光就是直接照射的光。直射光总有方向和颜色，并且用作着色算法的一个因子，如高洛德着色算法。不同类型的光以不同的方式发出直射光，产生特殊的衰减效果。通过调用IDirect3DDevice9::SetLight方法，可以设置一组直射光参数。
环境光存在于场景中的任何地方。可以认为它是充满整个场景的光的强度，与物体及它们在场景中的位置无关。环境光没有位置和方向，只有颜色和强度。每个光源会增加场景中总的环境光级别。可以调用IDirect3DDevice9::SetRenderState方法设置环境光的级别，将*State*参数指定为D3DRS_AMBIENT，将*Value*参数设置为希望的RGBA颜色值。
环境反射色采用RGBA值的形式表示，其中每个成分为从 0 到 255 之间的整数值。这与Microsoft® Direct3D®中大多数的颜色值有所不同。更多信息，请参阅光源和材质的颜色值。
可以用D3DCOLOR_RGBA宏生成RGBA值。红、绿、蓝成分组合成最终的环境反射色值。阿尔法成分控制颜色的透明度。当使用硬件加速或RGB模拟设备时，阿尔法成分被忽略。
光的属性
光的属性描述了光源的类型和颜色。取决于正在使用的光源的类型，光源可以拥有表示衰减和范围的属性，或表示聚光灯效果的属性。但是，并不是所有类型的光源都使用所有的属性。
Microsoft® Direct3D®用D3DLIGHT9结构存放所有类型光源的属性。本节包含关于所有光的属性的信息。信息被划分为以下部分。
光的类型
光的颜色
有颜色的顶点（Color Vertices）
光源的位置、范围和衰减
光的方向
光的类型
光的类型属性定义了正在使用的光源是什么类型。光的属性由D3DLIGHT9结构的Type成员的值设置，在C++程序中该成员是一个D3DLIGHTTYPE枚举类型。Microsoft® Direct3D®中有三种类型的光——点光源、聚光灯和平行光。每种类型以不同的方式照亮场景中的物体，所需的计算量也不同。
点光源
场景中的点光源具有颜色和位置，但没有确定的方向。点光源向各个方向发出的光相等，如下图所示。



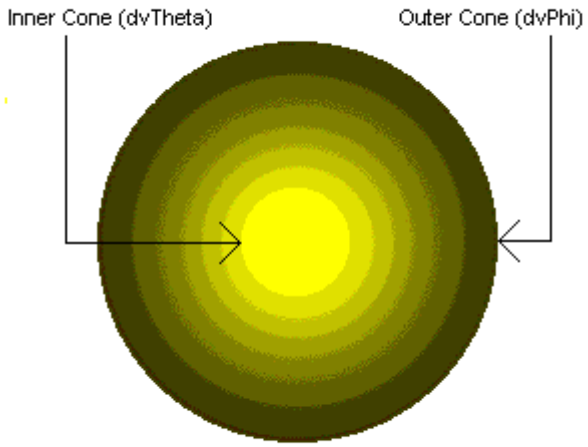
灯泡是点光源的一个很好的例子。点光源受衰减和范围的影响，并基于每个顶点对网格进行照明。在计算光照的过程中，Direct3D 使用点光源在世界坐标中的位置和当前顶点的坐标得到光的方向向量，以及光传播的距离。两者连同顶点法向一起，用于计算光在表面照明中所起的作用。

平行光

平行光只有颜色和方向，没有位置。平行光发出平行的光，这意味着所有平行光产生的光在场景中以相同的方向传播。可以认为平行光是位于无限远处的光源，如太阳。平行光不受衰减和范围的影响，因此应用程序指定的方向和颜色是 Direct3D 计算顶点颜色时要考虑的唯二因子。因为照明因子的数量少，所以平行光是可用的计算量最小的光。

聚光灯

聚光灯具有颜色、位置和发出光的方向。聚光灯发出的光由一个比较亮的内圆锥和一个较大的外圆锥组成，光强由内而外逐渐减小，如下图所示。

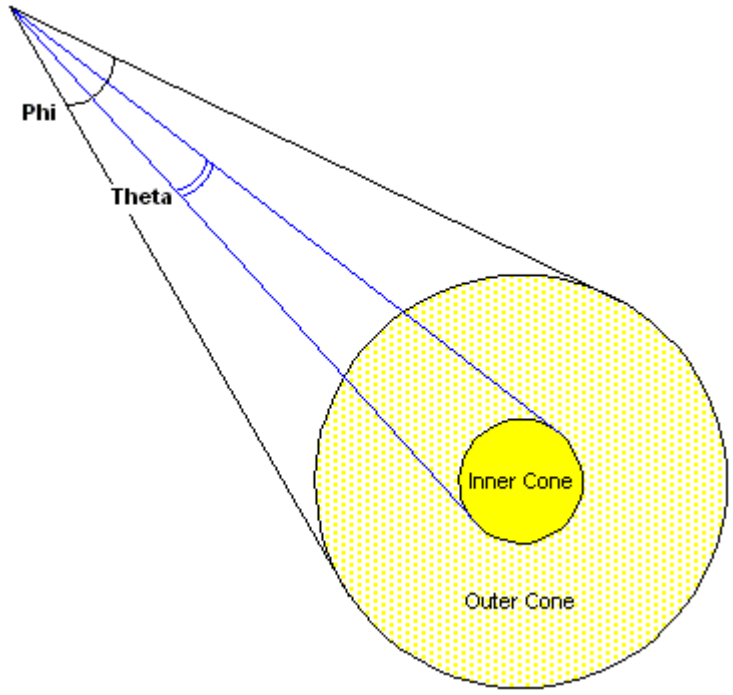


聚光灯受辐射（falloff）、衰减和范围的影响。这些因子同光到每个顶点的距离一起，参与计算场景中物体的光照效果。由于需要对每个顶点计算这些效果，因此这使得聚光灯成为 Direct3D 所有类型的光源中最为耗时的。

C++程序中，D3DLIGHT9 结构包含了三个仅用于聚光灯的成员。这些成员——Falloff、Theta、和 Phi——控制聚光灯内外锥的大小，以及光如何在两者之间减弱。

Theta 值为聚光灯内锥的角度，以弧度为单位，Phi 值为聚光灯外锥的角度。Falloff 值控制光强如何从内锥的外侧向外锥的内侧减弱。大多数应用程序将 Falloff 设为 1.0，使光在两个圆锥间平滑地减弱，但也可以根据需要设成其它值。

下图显示了这些成员的值之间的关系，以及它们如何决定聚光灯的内外锥。



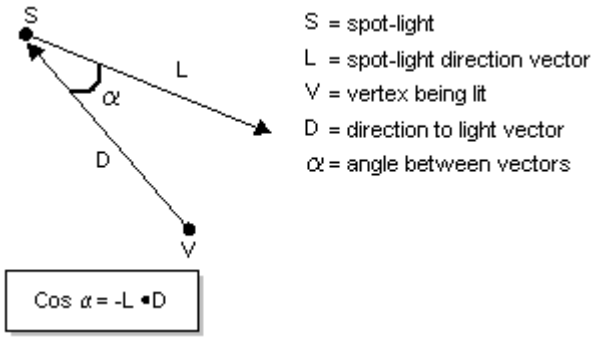
聚光灯模型

这里包含了有关聚光灯如何工作的更多信息。

聚光灯模型

聚光灯发出圆锥形的光，圆锥分为两部分：较亮的内锥和外锥。内锥中的光最亮，而外锥以外则没有光，在内外锥之间光强逐渐衰减。这种衰减一般被称为辐射。

一个顶点接收到的光的数量（译注：此后简称为光量）取决于顶点在内锥或外锥中的位置。Microsoft® Direct3D®计算聚光灯的方向向量(*L*)和从顶点到聚光灯的向量(*D*)的点积。这个值等于两个向量夹角的余弦值，并作为顶点位置的一个指标，可与聚光灯的圆锥角度进行比较以确定顶点位于内锥或外锥的何处。下面提供了这两个向量之间关系的表示图。

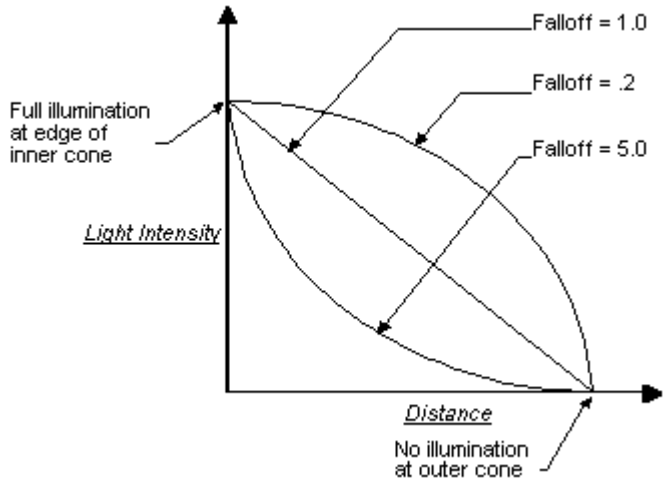


系统将这个值与聚光灯的内锥和外锥角度的余弦值进行比较。在聚光灯的D3DLIGHT9结构中，Theta和Phi成员表示内锥和外锥的整个圆锥的角度。因为衰减和顶点与照明中心（亦即圆锥的中线）的夹角有关，所以在计算余弦值时Direct3D会将这两个圆锥的角度除以二。

如果向量 *L* 和 *D* 的点积小于等于外锥角度的余弦值，那么顶点位于外锥以外，不接收任何光。如果 *L* 和 *D* 的点积大于内锥角度的余弦值，那么顶点位于内锥里面，接收最多的光量，此时仍需考虑光随距离的衰减。如果顶点位于这两个区域之间的某处，那么 Direct3D 使用以下公式计算顶点的辐射值。

$$I_f = \left(\frac{Cos\alpha - Cos\phi}{Cos\theta - Cos\phi} \right)^p$$

在这个公式中，*If* 为辐射后的光强，对正在计算光照的顶点来说，α 为向量 *L* 和 *D* 之间的夹角，F 为二分之一外锥角度的余弦值，φ 为二分之一内锥角度，为聚光灯在 D3DLIGHT9 结构中的辐射属性 Falloff。这个公式产生一个从 0.0 到 1.0 之间的值，用这个值对光强进行缩放就产生了辐射的效果。作为从顶点到聚光灯的距离的因子，衰减同时也被使用。*p* 值对应 D3DLIGHT9 结构的 Falloff 成员，控制辐射曲线的形状。下图显示了不同的 Falloff 值如何作用于辐射曲线。



在实际光照中，不同的 Falloff 值产生的效果是很敏感的，并且如果使用除了 1.0 之外的值作为 Falloff 值描述辐射曲线，那么还会导致些许性能下降。为此，这个值一般被设为 1.0。

光的颜色

Microsoft® Direct3D®中光源发出三种颜色，这三种颜色单独用于系统的光照计算：漫反射色、环境反射色、和镜面反射色。每种颜色在Direct3D光照模型的协助下，与当前材质中的对应部分相互作用，产生用于渲染的最终颜色。漫反射色与当前材质的漫反射系数属性相互作用，镜面反射色与材质的镜面反射反射系数属性相互作用，依次类推。有关Direct3D如何应用这些颜色的细节，请参阅与光照相关的数学。

在C++应用程序中，D3DLIGHT9结构包含了三个与这些颜色——漫反射色、环境反射色、和镜面反射色——相对应的成员，每个成员都是一个D3DCOLORVALUE结构，定义了发出的颜色。

对系统计算量影响最大的颜色类型是漫反射色。最常用的漫反射色是白色(R:1.0 G:1.0 B:1.0)，但是应用程序可以根据需要创建其它颜色以达到想要的效果。例如，应用程序可以为火炉使用红光，或者为处于“通行”状态的红绿灯使用绿光。

一般来说，应用程序将光的颜色成员设为从 0.0 到 1.0 之间的值，闭区间，但这不是必需的。例如，应用程序可以将所有成员设为 2.0，创建出一个“比白色更亮”的光源。当应用程序使用的衰减值不为常数时，这种设定尤其有用。

注意，虽然Direct3D使用RGBA值表示光的颜色，但是并没有使用颜色的阿尔法成员。更多信息，请参阅光和材质和颜色值。

有颜色的顶点（Color Vertices）

通常用材质颜色进行光照计算。但是，应用程序可以指定用顶点的漫反射色或镜面反射色覆盖材质颜色——放射色（emissive），环境反射色、漫反射色、和镜面反射色。这可以通过调用 IDirect3DDevice9::SetRenderState方法并用下表所列的值设置设备的状态变量来完成。

设备状态变量	含义	类型	默认值

D3DRS_AMBIENTMATERIALSOURCE	定义从何处得到环境光的材质色。	D3DMATERIALCOLORSOURCE	D3DMCS_MATERIAL
D3DRS_DIFFUSEMATERIALSOURCE	定义从何处得到漫反射材质色。	D3DMATERIALCOLORSOURCE	D3DMCS_COLOR1
D3DRS_SPECULARMATERIALSOURCE	定义从何处得到镜面反射材质色。	D3DMATERIALCOLORSOURCE	D3DMCS_COLOR2
D3DRS_EMISSIVEMATERIALSOURCE	定义从何处得到 emissive 材质色。	D3DMATERIALCOLORSOURCE	D3DMCS_MATERIAL
D3DRS_COLORVERTEX	禁用或启用顶点色。	BOOL	TRUE

阿尔法/透明度值总是从漫反射色的阿尔法通道中得到。

雾因子的值总是从镜面反射色的阿尔法通道中得到。

D3DMATERIALCOLORSOURCE 可以为以下值：

D3DMCS_MATERIAL - 用材质颜色作为材质颜色的来源。

D3DMCS_COLOR1 - 用顶点的漫反射色作为材质颜色的来源。

D3DMCS_COLOR2 - 用顶点的镜面反射色作为材质颜色的来源。

光的位置、范围和衰减

位置、范围和衰减属性定义了光源在世界空间中的位置和光源发出的光如何随距离而变化。同C++应用程序中使用的所有其它光的属性一样，这些属性被包含在光源的D3DLIGHT9结构中。

位置

光源的位置用D3DVECTOR结构表示，它对应于D3DLIGHT9 结构中的Position成员。约定x，y和z坐标在世界空间中。平行光是唯一不使用位置属性的光源类型。

范围

光源的范围属性决定在世界空间中的距离，场景中与光源相距超出这个距离的网格不再接收该光源发出的光。Range 成员是一个浮点数，表示光源在世界空间中的最大范围。平行光不使用范围属性。

衰减

衰减控制光强如何朝范围属性指定的最大范围逐渐减弱。D3DLIGHT9 结构中有三个成员用于表示衰减：Attenuation0、Attenuation1、和Attenuation2。这些成员为从 0.0 到无穷大的浮点数，用于控制光强的衰减。一些应用程序将Attenuation1 成员设为 1.0，而其余的设为 0.0，得到光强的变化为 1/*D*，这里*D*为从光源到顶点之间的距离。光强在光源处最大，在光的最大范围处减弱到 1/(*Light Range*)。一般来说，应用程序将Attenuation0 设为 0.0，将Attenuation1 设为一个常数，将Attenuation2 设为 0.0。（译注：请参阅与光照相关的数学）

为得到更复杂的衰减效果，应用程序可以将衰减值结合起来使用。或者，为创建更奇怪的衰减效果，应用程序也可以将它们设为正常范围外的值。但是，负的衰减值是不允许的。

光的方向

光的方向属性决定光源发出的光在世界空间中传播的方向。只有平行光和聚光灯使用方向，方向用一个向量表示。

C++应用程序在D3DLIGHT结构的Direction成员中设置光的方向。Direction成员为D3DVECTOR类型。方向向量用始于逻辑原点的一段距离描述，与光源在场景中的位置无关。因此，一个笔直指向场景内——沿正z轴——的聚光灯，无论它的位置在哪里，它的方向向量都是<0,0,1>。类似地，通过使用方向为<0,-1,0>的平行光，应用程序可以模拟直射到场景中的阳光。显然，应用程序不一定要创建沿坐标轴照射的光，可以混合搭配这些值，创建出沿某个更有意义的角度照射的光。

注意虽然应用程序无需归一化光的方向向量，但是要保证该向量长度不为零。换句话说，不要将<0,0,0>用作方向向量。

光的使用

设置和取得光的属性

在C++应用程序中，可以通过先准备一个D3DLIGHT9结构，然后调用IDirect3DDevice9::SetLight方法设置光的属性。IDirect3DDevice9::SetLight方法接收一个索引值，告诉设备将这组光属性存放在它的内部属性列表的何处，及定义这些属性的D3DLIGHT9 结构的地址。要更新光的照明属性，应用程序可以根据需要用新的属性信息调用IDirect3DDevice9::SetLight。

每次应用程序用一个索引值调用IDirect3DDevice9::SetLight，但该索引值从未设置过属性时，系统会分配内存以容纳新的一组光属性。应用程序可以设置许多光源，但是任一时刻只能启用其中的一个子集（译注：子集的大小由硬件能力决定）。要确定设备支持的可激活的光源，应用程序可以在取得设备能力后检查D3DCAPS9结构的MaxActiveLights成员。如果应用不再需要使用某个光源，可以将它禁用或用新的一组光属性将它覆盖。

以下 C++示例代码准备并设置了一个白色点光源的属性，它发出的光不会随距离而衰减。

```
// 假设 d3dDevice 为一个指向 IDirect3DDevice9 接口的有效指针。
D3DLIGHT9 d3dLight;
HRESULT   hr;

// 初始化结构。
ZeroMemory(&d3dLight, sizeof(d3dLight));

// 设置一个白色的点光源。
d3dLight.Type = D3DLIGHT_POINT;
d3dLight.Diffuse.r  = 1.0f;
d3dLight.Diffuse.g  = 1.0f;
d3dLight.Diffuse.b  = 1.0f;
d3dLight.Ambient.r  = 1.0f;
d3dLight.Ambient.g  = 1.0f;
d3dLight.Ambient.b  = 1.0f;
d3dLight.Specular.r = 1.0f;
d3dLight.Specular.g = 1.0f;
d3dLight.Specular.b = 1.0f;

// 将它放在场景中的高处，位于用户的后面。
// 记住，这些坐标是在世界空间中的，因此用户也可以在世界空间中的任何位置。
// 根据本示例代码的目的，假设用户位于世界空间的原点。
d3dLight.Position.x = 0.0f;
d3dLight.Position.y = 1000.0f;
d3dLight.Position.z = -100.0f;

// 不随距离衰减。
d3dLight.Attenuation0 = 1.0f;
d3dLight.Range        = 1000.0f;
```

```
// 设置第一个光源的属性。
hr = d3dDevice->SetLight(0, &d3dLight);
if (SUCCEEDED(hr))
    // 成功后的处理
else
    // 失败后的处理
```

应用程序可以在任何时候再次调用 IDirect3DDevice9::SetLight 方法以更新一组光属性。只需指定要更新的那组光属性的索引值和包含新属性的 D3DLIGHT9 结构的地址。

注意给设备分配一组光属性并不意味着启用对应的光源。要对设备调用IDirect3DDevice9::LightEnable方法才能启用光源。

在C++程序中，通过对设备调用IDirect3DDevice9::GetLight方法，应用程序可以取得一个现存光源的所有属性。

以下示例代码描述了这个过程。

```
// 假设 d3dDevice 为一个指向 IDirect3DDevice9 接口的有效指针。
HRESULT hr;
D3DLIGHT9 light;

// 取得第一个光源的属性信息。
hr = pd3dDevice->GetLight(0, &light);
if (SUCCEEDED(hr))
    // 成功后的处理
else
    // 失败后的处理
```

如果应用程序提供的索引值超出了分配给设备的光源的范围,那么IDirect3DDevice9::GetLight方法将会失败，返回D3DERR_INVALIDCALL错误码。

启用和禁用光源

当应用程序给场景中的光源指定了光属性时，就可以对该设备调用IDirect3DDevice9::LightEnable 方法激活这个光源。默认情况下新的光源是禁用的。

IDirect3DDevice9::LightEnable 方法接收两个参数。将第一个参数设为要用改变的光源的索引值，从零开始计数，将第二个参数设为 TRUE 启用该光源或 FALSE 禁用该光源。

以下示例代码描述了用这种方法启用设备的光源属性列表中的第一个光源。

```
// 假设 d3dDevice 为一个指向 IDirect3DDevice9 接口的有效指针。
HRESULT hr;

hr = pd3dDevice->LightEnable(0, TRUE);
if (SUCCEEDED(hr))
    // 成功后的处理
else
    // Handle failure
```

要确定设备支持的最多可激活的光源的数量，应用程序可以在取得设备能力时，检查 D3DCAPS9 结构的 MaxActiveLights 成员。

如果应用程序启用或禁用一个光源，但没有调用 IDirect3DDevice9::SetLight 设置过该光源的属性，那么 IDirect3DDevice9::LightEnable 方法会先用下表所列的属性创建一个光源，然后再启用或禁用该光源。

成员	默认值
Type	D3DLIGHT_DIRECTIONAL
Diffuse	(R:1, G:1, B:1, A:0)
Specular	(R:0, G:0, B:0, A:0)
Ambient	(R:0, G:0, B:0, A:0)
Position	(0, 0, 0)
Direction	(0, 0, 1)
Range	0
Falloff	0
Attenuation0	0
Attenuation1	0
Attenuation2	0
Theta	0
Phi	0

与光照相关的数学

Microsoft® Direct3D®光照模型涵盖了环境光、漫反射光、镜面反射光和放射光，这足以解决绝大部分的光照情况。我们将场景中光的总和称为全局照明（*global illumination*），并使用以下公式计算：

全局照明 = 环境光 + 漫反射光 + 镜面反射光 + 放射光

环境光是恒定的光照。它在所有方向上不发生变化，对物体中所有像素产生的作用也完全相同。它计算起来很快，但得到的物体看起来是平面的，没有真实感。要了解Direct3D如何计算环境光，请参阅环境光。

漫反射光取决于光的方向和表面的法向。由于光的方向和表面法向量的变化，因此漫反射光会随物体的表面而变化。因为漫反射光随着每个顶点而变化，所以需要更长的时间进行计算，但是使用漫反射光带来的好处是它使物体呈现出明暗变化和三维深度。要了解Direct3D如何计算漫反射光，请参阅漫反射光。

镜面反射光代表了当光线照射到物体表面时反射回摄像机形成的明亮的镜面反射高光。它比漫反射光更强，但在物体表面也衰减得更快。计算镜面反射光需要比计算漫反射光更长的时间，但是使用镜面反射光带来的好处是它给表面增添了重要的细节。要了解Direct3D如何计算镜面反射光，请参阅镜面反射光。

放射光是物体发出的光，例如光晕（glow）。要了解Direct3D如何计算放射光，请参阅放射光。通过在三维场景中使用这些类型的光，可以得到真实的光照效果。要得到更为真实的光照效果，应用程序可以添加更多的光源，但是，这增加了渲染场景的时间。要达到（游戏）设计师想要的所有效果，一些游戏使用了超出一般的 CPU 计算能力。在这种情况下，一般通过在使用纹理贴图的同时，使用光照贴图和环境贴图给场景加入光照的效果，这样就可以把光照所需的计算量减到最少。

光照计算在摄像机空间进行。要了解如何计算光照变换，请参阅摄像机空间的变换。经过优化的光照计算可以在建模空间进行，但需满足以下特殊条件：法向量已经归一化（D3DRS_NORMALIZENORMALS为TRUE），不需要进行顶点混合，变换矩阵为正交的，等等。

环境光、漫反射光和镜面反射光会受到给定光源的衰减和聚光灯因子的影响。为环境光、放射光和漫反射光成分计算的颜色值被保存在输出顶点的漫反射色中。漫反射和镜面反射公式都包含了

衰减和聚光灯因子属性。更多信息，请参阅衰减和聚光灯因子。

环境光

环境光为场景提供了一种恒定不变的光。环境光对所有物体的顶点的照明效果相同，因为它与其余光照因子无关，如顶点法向、光的方向、光的位置、范围或衰减等。环境光是最快的一种类型，但它提供的真实感最少。Microsoft® Direct3D®包含了一个全局的环境光属性，应用程序可以直接使用而无需创建任何光源。另外，应用程序也可以指定某个光源提供环境光。场景中环境光的计算由以下公式描述。

Ambient Lighting = Ca*[Ga + sum(Lai*Attenuationi*SpotFactori)]

环境光的公式不完全正确。应该是 Ambient Lighting = Ca*[Ga + sum(Lai)*Atti*Spoti]，这里 Att 和 Spot 为第 i 个光源的衰减和聚光灯因子。

参数在下表中定义。

参数	默认值	类型	描述
Ca	(0,0,0,0)	D3DCOLORVALUE	材质的环境反射色。
Ga	(0,0,0,0)	D3DCOLORVALUE	全局的环境反射色。
sum	N/A	N/A	所有光源产生的环境光的总和。
Lai	(0,0,0,0)	D3DVECTOR	第 i 个光源产生的环境反射色。
Atteni	(0,0,0,0)	D3DCOLORVALUE	第i个光源的衰减因子。请参阅衰减和聚光灯因子。
Spoti	(0,0,0,0)	D3DVECTOR	第i个光源的聚光灯因子。请参阅衰减和聚光灯因子。

Ca 的值可以是：

顶点颜色 1，如果 AMBIENTMATERIALSOURCE = D3DMCS_COLOR1，并且顶点声明中给出了第一个顶点的颜色。

顶点颜色 2，如果 AMBIENTMATERIALSOURCE = D3DMCS_COLOR2，并且顶点声明中给出了第二个顶点的颜色。

材质的环境反射色。

注意如果使用了任何一种 AMBIENTMATERIALSOURCE，但是没有提供顶点颜色，那么系统会使用材质的环境反射色。

要使用材质的环境反射色，按以下示例代码使用 SetMaterial 方法。

Ga 为全局的环境反射色，通过 SetRenderState(D3DRENDERSTATE_AMBIENT)设置。Direct3D 场景中只有一个全局环境反射色，它与其余 Direct3D 光源无关。

Lai 为场景中第 i 个光源的环境反射色。每个 Direct3D 光源都有一组属性，其中一个就是环境反射色。符号 sum(Lai)表示场景中所有环境反射色的总和。

示例

在本例中，通过计算场景的环境光的颜色和材质的环境反射色得到物体的颜色。代码如下所示。

```
#define GRAY_COLOR            0x00bfbfbf

// 创建材质
D3DMATERIAL9 mtrl;
ZeroMemory( &mtrl, sizeof(mtrl) );
mtrl.Ambient.r = 0.75f;
mtrl.Ambient.g = 0.0f;
mtrl.Ambient.b = 0.0f;
mtrl.Ambient.a = 0.0f;
m_pd3dDevice->SetMaterial( &mtrl );
```

m_pd3dDevice->SetRenderState(D3DRS_AMBIENT, GRAY_COLOR);
根据公式，得到的物体顶点的颜色是材质颜色和光的颜色的合成。
下面两张图片显示了材质颜色，为灰色，和光的颜色，为红色。


渲染得到的场景如下所示。场景中唯一的物体是一个球体。环境光用相同的颜色对物体的所有顶点进行光照计算，它不依赖于顶点法向和光的方向。因此，球体看起来像是二维的圆，因为物体的表面没有明暗变化。

要使物体看起来更真实，除了环境光之外，需要再添加漫反射光或镜面反射光。
漫反射光
在根据任何衰减效果调整完光的强度之后，光照引擎用给定的顶点法向与入射光方向之间的夹角，计算剩余的光中有多少会从顶点反射。对于平行光，光照引擎略过这一步，因为平行光不随距离而衰减。系统会考虑两种反射类型，漫反射和镜面反射，并使用不同的公式计算每种类型各反射多少光。在计算完反射光的数量后，Microsoft® Direct3D®把得到的新值应用于当前材质的漫反射和镜面反射反射系数属性。最终的颜色值是漫反射色和镜面反射色成员，会被光栅化器用于计算高洛德着色和镜面反射高光。
漫反射光由以下公式描述。
Diffuse Lighting = sum[Cd*Ld*(N.Ldir)*Atten*Spot]

参数	默认值	类型	描述
sum	N/A	N/A	每个光源的漫反射色成分的总和。
Cd	(0,0,0,0)	D3DCOLORVALUE	漫反射色。
Ld	(0,0,0,0)	D3DCOLORVALUE	光源的漫反射色。
N	N/A	D3DVECTOR	顶点法向。
Ldir	N/A	D3DVECTOR	从顶点到光源的方向向量。
Atten	N/A	FLOAT	光的衰减因子。请参阅衰减和聚光灯因子。
Spot	N/A	FLOAT	聚光灯因子。请参阅衰减和聚光灯因子。

Cd 的值可以是：
顶点颜色 1，如果 DIFFUSEMATERIALSOURCE = D3DMCS_COLOR1，并且顶点声明中给出了第一个顶点的颜色。
顶点颜色 2，如果 DIFFUSEMATERIALSOURCE = D3DMCS_COLOR2，并且顶点声明中给出了第二个顶

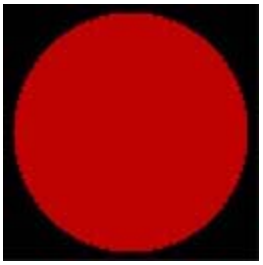
点的颜色。
材质的环境反射色。
注意如果使用了任何一种 DIFFUSEMATERIALSOURCE，但是没有提供顶点颜色，那么系统会使用材质的漫反射色。
要计算衰减(Atten)或聚光灯(Spot)属性，请参阅衰减和聚光灯因子。
在所有光源都经过单独处理和插值后，漫反射成员被截取到从 0 到 255 之间。最终的漫反射光的颜色值是环境光、漫反射光和放射光的颜色值的组合。

示例

在本例中，通过计算光源的漫反射色和材质的漫反射色得到物体的颜色。代码如下所示。

```
D3DMATERIAL9 mtrl;
ZeroMemory( &mtrl, sizeof(mtrl) );

D3DLIGHT9 light;
ZeroMemory( &light, sizeof(light) );
light.Type = D3DLIGHT_DIRECTIONAL;

D3DXVECTOR3 vecDir;
vecDir = D3DXVECTOR3(0.5f, 0.0f, -0.5f);
D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDir );

// 设置平行光的漫反射色
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
light.Diffuse.a = 1.0f;
m_pd3dDevice->SetLight( 0, &light );
m_pd3dDevice->LightEnable( 0, TRUE );

// 如果使用了材质，那么必须使用 SetRenderState。
// 顶点颜色 = 光的漫反射色 * 材质的漫反射色
mtrl.Diffuse.r = 0.75f;
mtrl.Diffuse.g = 0.0f;
mtrl.Diffuse.b = 0.0f;
mtrl.Diffuse.a = 0.0f;
m_pd3dDevice->SetMaterial( &mtrl );
m_pd3dDevice->SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE, D3DMCS_MATERIAL);
```

根据公式，得到的物体顶点的颜色是材质颜色和光的颜色的组合。
下面两张图片显示了材质颜色，为灰色，和光的颜色，为红色。

渲染得到的场景如下所示。场景中唯一的物体是一个球体。漫反射光照计算取得材质和光的漫反射色，用光的方向和顶点法向的点积作为它们之间的夹角修正得到的颜色。因此，球体的后面变得较黑，因为那部分球面背向光源。

将漫反射光和前例产生的环境光结合起来，会使物体的整个表面呈现出明暗效果。环境光使整个表面变亮，而漫反射光则有助于展现出物体的三维形状。

漫反射光比环境光需要更多的计算，因为它依赖于顶点的法向和光的方向。我们可以看一下在三维场景中的几何体，漫反射光产生了比环境光更为真实的光照效果。我们还可以使用镜面反射高光来达到更为真实的效果。

镜面反射光

建立一个镜面反射模型需要系统不仅知道光传播的方向，还要知道从顶点到视点的方向。

Direct3D 光照系统使用了一个经过简化的 Phong（译注：冯）镜面反射模型，该简化模型使用一个中间向量（原文：halfway vector，译注：某些图形学书籍译为半角矢量）计算镜面反射光强的近似值。

默认的光照状态不计算镜面反射高光。要启用镜面反射高光，需要将D3DRS_SPECULARENABLE设置为TRUE。

镜面反射光公式

镜面反射光由以下公式描述。

Specular Lighting = Cs*sum[Ls*(N.H)P*Atten*Spot]

下表描述了所有变量，其类型及范围。

参数	默认值	类型	描述
Cs	(0,0,0,0)	D3DCOLORVALUE	镜面反射色。
sum	N/A	N/A	每个光源的镜面反射成员的总和。
N	N/A	D3DVECTOR	顶点法向。
H	N/A	D3DVECTOR	中间向量。请参阅中间向量相关的部分。
P	0.0	FLOAT	镜面反射反射指数。范围从 0 到正无穷大。
Ls	(0,0,0,0)	D3DCOLORVALUE	光的镜面反射色。
Atten	N/A	FLOAT	光的衰减值。请参阅衰减和聚光灯因子。
Spot	N/A	FLOAT	聚光灯因子。请参阅衰减和聚光灯因子。

Cs 的值可以是：
顶点颜色 1，如果 SPECULARMATERIALSOURCE = D3DMCS_COLOR1，并且顶点声明中给出了第一个顶点的颜色。
顶点颜色 2，如果 SPECULARMATERIALSOURCE = D3DMCS_COLOR2，并且顶点声明中给出了第二个顶点的颜色。
材质的环境反射色。
注意 如果使用了任何一种 SPECULARMATERIALSOURCE，但是没有提供顶点颜色，那么系统会使用材质的镜面反射色。
在所有光源都经过单独处理和插值后，镜面反射成员被截取到从 0 到 255 之间。

中间向量

中间向量(H)位于两个向量之间：从顶点到光源的向量，和从顶点到摄像机位置的向量。
Microsoft® Direct3D®提供了两种方法来计算中间向量。当D3DRS_LOCALVIEWER被设为TRUE时，系统使用摄像机的位置和顶点的位置，与光源的方向向量来计算中间向量。以下公式描述了这种方法。

```
H = norm(norm(Cp - Vp) + Ldir)
```

参数	默认值	类型	描述
Cp	N/A	D3DVECTOR	摄像机的位置。
Vp	N/A	D3DVECTOR	顶点的位置。
Ldir	N/A	D3DVECTOR	从顶点位置到光源位置的向量。

用这种方法计算中间向量的计算量会非常大。另一种可选的方法是将D3DRS_LOCALVIEWER设置为FALSE，告诉系统将视点当作在z轴无限远处进行计算。下面的公式反映了这种方法。

```
H = norm((0,0,1) + Ldir)
```

这种设定计算量不很大，但很不精确，因此它最适合用在使用正交投影的应用程序中。

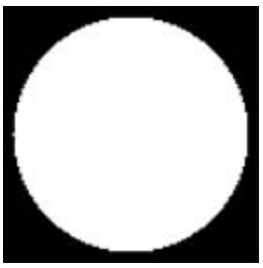
示例

在本例中，通过计算光源的镜面反射色和材质的镜面反射色得到物体的颜色。代码如下所示。

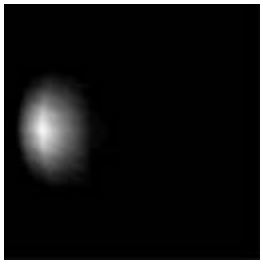
```
D3DMATERIAL9 mtrl;
ZeroMemory( &mtrl, sizeof(mtrl) );

D3DLIGHT9 light;
ZeroMemory( &light, sizeof(light) );
light.Type = D3DLIGHT_DIRECTIONAL;

D3DXVECTOR3 vecDir;
vecDir = D3DXVECTOR3(0.5f, 0.0f, -0.5f);
D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDir );

light.Specular.r = 1.0f;
light.Specular.g = 1.0f;
light.Specular.b = 1.0f;
light.Specular.a = 1.0f;

light.Range = 1000;
```

```
light.Falloff = 0;
light.Attenuation0 = 1;
light.Attenuation1 = 0;
light.Attenuation2 = 0;
m_pd3dDevice->SetLight( 0, &light );
m_pd3dDevice->LightEnable( 0, TRUE );
m_pd3dDevice->SetRenderState( D3DRS_SPECULARENABLE, TRUE );

mtrl.Specular.r = 1.0f;
mtrl.Specular.g = 1.0f;
mtrl.Specular.b = 1.0f;
mtrl.Specular.a = 1.0f;
mtrl.Power = 20;
m_pd3dDevice->SetMaterial( &mtrl );
m_pd3dDevice->SetRenderState(D3DRS_SPECULARMATERIALSOURCE, D3DMCS_MATERIAL);
```

根据公式，得到的物体顶点的颜色是材质颜色和光的颜色的组合。
以下两张图片显示了材质颜色，为灰色，和光的颜色，为白色。

得到的镜面反射高光如下所示。



将镜面反射高光与前例产生的环境光和漫反射光结合起来，会得到以下图像。当所有三种类型的光一起使用时，渲染得到的球体明显看起来更像是真实物体。

镜面反射光需要比漫反射光更多的计算量。镜面反射光一般用于以可见的方式提供有关表面材质的信息。镜面反射高光会随着表面材质的不同在大小和颜色上有所不同。
放射光

放射光由一个单独的参数表示。

Emissive Lighting = Ce

这里：

参数 默认值 类型 描述

Ce (0,0,0,0) D3DCOLORVALUE Emissive color.

Ce 的值可以是：

顶点颜色 1，如果 EMISSIVEMATERIALSOURCE = D3DMCS_COLOR1，并且顶点声明中给出了第一个顶点的颜色。

顶点颜色 2，如果 EMISSIVEMATERIALSOURCE = D3DMCS_COLOR2，并且顶点声明中给出了第二个顶点的颜色。

材质的放射色。

注意如果使用了任何一种 EMSSIVEMATERIALSOURCE，但是没有提供顶点颜色，那么系统会使用材质的放射色。

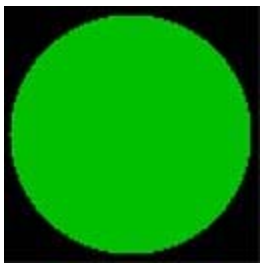
示例

在本例中，通过计算场景的环境光的颜色和材质的环境反射色得到物体的颜色。代码如下所示。

```
// 创建材质
D3DMATERIAL9 mtrl;
ZeroMemory( &mtrl, sizeof(mtrl) );
mtrl.Emissive.r = 0.0f;
mtrl.Emissive.g = 0.75f;
mtrl.Emissive.b = 0.0f;
mtrl.Emissive.a = 0.0f;
m_pd3dDevice->SetMaterial( &mtrl );
m_pd3dDevice->SetRenderState(D3DRS_EMISSIVEMATERIALSOURCE, D3DMCS_MATERIAL);
```

根据公式，得到的物体顶点的颜色为材质的颜色。

下图显示了材质颜色，为绿色。放射光用相同的颜色对物体的所有顶点进行光照计算，它不依赖于顶点法向或光的方向。因此，球体看起来像是二维的圆，因为物体表面没有明暗变化。



下图显示了放射光与前例中其余三种类型的光混合后得到的结果。球体的右边是绿色的放射光和红色的环境光的混合。球体的左边，绿色的放射光与红色的环境光和漫反射光混合产生了红色的颜色梯度。中间是白色的镜面反射高光，随着镜面反射高光向周围迅速衰减，产生了一个黄色的圆环，其余区域只有环境光、漫反射光和放射光混合成黄色。

摄像机空间的变换

摄像机空间中的顶点通过用世界视矩阵（world view matrix）对顶点进行变换得到。

V = V * wvMatrix

摄像机空间中的顶点法向通过用世界视矩阵的转置逆矩阵对物体的法向进行变换得到。世界视矩阵可能是也可能不是正交的。

N = N * (wvMatrix-1)T

求逆矩阵和转置矩阵的操作在一个 4x4 矩阵上进行。乘法操作将法向和得到的 4x4 矩阵中的 3x3 那部分相乘。

如果渲染状态 D3DRENDERSTATE_NORMALIZENORMALS 被设置为 TURE，那么在变换到摄像机空间后，顶点法向量被归一化，如下所示。

N = norm(N)

摄像机空间中光源的位置通过用视矩阵对光源位置进行变换得到。

Lp = Lp * vMatrix

摄像机空间中平行光的方向通过将光源的方向与视矩阵相乘、归一化、并对结果取反得到。

Ldir = -norm(Ldir * wvMatrix)

对 D3DLIGHT_POINT 和 D3DLIGHT_SPOT 类型的光源，光的方向用以下公式计算：

Ldir = norm(V * Lp)，这里的参数由下表定义。

参数	默认值	类型	描述
Ldir	N/A	D3DVECTOR	从顶点到光源的方向向量
V	N/A	D3DVECTOR	摄像机空间中顶点的位置
wvMatrix	Identity	D3DMATRIX	包含世界和视变换和合成矩阵
N	N/A	D3DVECTOR	顶点法向
Lp	N/A	D3DVECTOR	摄像机空间中光源的位置
vMatrix	Identity	D3DMATRIX	包含视变换的矩阵

衰减和聚光灯因子

漫反射光和镜面反射光成员的全局照明公式中包含了描述光的衰减和聚光灯因子的属性。这些属性在下面描述。

衰减因子

光的衰减因子取决于光的类型和从光到顶点所在位置的距离。要计算衰减因子，可以使用以下三个公式之一。

Atten = 1/(att0i + att1i * d + att2i * d2)

这里：

参数	默认值	类型	描述
att0i	0.0	FLOAT	线性衰减因子。范围从 0 到正无穷大

att1i 0.0 FLOAT 平方衰减因子。范围从 0 到正无穷大

att2i 0.0 FLOAT 指数衰减因子。范围从 0 到正无穷大。

d N/A FLOAT 从顶点位置到光源位置的距离

Atten = 1, 如果光源为平行光。

Atten = 0, 如果从光源到顶点的距离超出了光的范围。

att0、att1、和att2 值由D3DLIGHT9结构的*Attenuation0*、*Attenuation1*、和*Attenuation2* 成员给出。

从光源到顶点位置的距离总是正的。

d = | Ldir |

这里：

参数 默认值 类型 描述

Ldir N/A D3DVECTOR 从顶点位置到光源位置的方向向量

如果 *d* 大于光的范围，也就是 D3DLIGHT9 结构的 *Range* 成员，Microsoft® Direct3D®不再进行更进一步的衰减计算，也不应用任何光照效果于该顶点。

衰减常数作为公式中的一个系数——应用程序仅需调整它们的值就可以产生不同的衰减曲线。应用程序可以将 *Attenuation1* 设为 1.0，创建一个不随距离衰减但仍受距离限制的光源，或者应用程序也可以试验不同的值以达到不同的衰减效果。

位于最大范围处，光的衰减因子不是 0.0。为防止光在最大范围处突然出现，应用程序可以增加光的范围。或者，应用程序也可以设置衰减常数使衰减因子在光的最大范围处接近 0.0。作为光到达一个顶点所经过的距离的因子，衰减值与光的颜色的红、绿和蓝成分相乘以缩放光强。

聚光灯因子

$$spot_i = \begin{cases} 1 & \text{for non- spotlights or if } rho_i > \cos\left(\frac{theta_i}{2}\right) \\ 0 & \text{if } rho_i \le \cos\left(\frac{phi_i}{2}\right) \\ \left[\frac{rho_i - \cos\left(\frac{phi_i}{2}\right)}{\cos\left(\frac{theta_i}{2}\right) - \cos\left(\frac{phi_i}{2}\right)}\right]^{falloff} & \text{otherwise} \end{cases}$$

参数 默认值 类型 描述

rhoi N/A FLOAT 聚光灯 i 的余弦值（角度）

phii 0.0 FLOAT 聚光灯 i 的外锥角度，以弧度为单位。范围为：[theta, pi)

thetai 0.0 FLOAT 聚光灯 i 的内锥角度，以弧度为单位。范围为：[0, pi)

falloff 0.0 FLOAT 辐射因子。范围为：（负无穷，正无穷）

这里：

rho = norm(Ldcs).norm(Ldir)

并且：

参数 默认值 类型 描述

Ldcs N/A D3DVECTOR 光在摄像机空间中的方向取反

Ldir N/A　　D3DVECTOR 从顶点位置到光源位置的方向向量

在计算光的衰减后，Direct3D还会考虑：聚光灯效果，如果适用的话，光从表面反射的角度，用于计算顶点的漫反射和镜面反射成员的当前材质的反射系数。更多信息，请参阅聚光灯模型。

材质

材质描述在三维场景中的多边形如何反射光或如何发光。本质上，材质是一组属性集合，告诉Microsoft® Direct3D®有关正在渲染的多边形的下列信息。

材质如何反射环境光和漫反射光

材质和镜面反射高光看起来是什么样

多边形发光时看起来是什么样

用C++写的Direct3D应用程序用D3DMATERIAL9结构描述材质属性。更多信息，请参阅材质属性。

设置材质属性

Direct3D 渲染设备在同一时刻可以使用一组材质属性进行渲染。

C++应用程序通过先准备一个D3DMATERIAL9 结构，然后调用IDirect3DDevice9::SetMaterial方法设置给系统使用的材质属性。

要准备使用一个 D3DMATERIAL9 结构，需根据想要创建的渲染效果设置该结构的属性信息。以下示例代码为一个具有高强度白色镜面反射高光的紫色材质设立了相应的 D3DMATERIAL9 结构。

```
D3DMATERIAL9 mat;

// 设置漫反射反射系数的 RGBA 值。
mat.Diffuse.r = 0.5f;
mat.Diffuse.g = 0.0f;
mat.Diffuse.b = 0.5f;
mat.Diffuse.a = 1.0f;

// 设置环境光反射系数的 RGBA 值
mat.Ambient.r = 0.5f;
mat.Ambient.g = 0.0f;
mat.Ambient.b = 0.5f;
mat.Ambient.a = 1.0f;

// 设置镜面反射高光的颜色和强度。
mat.Specular.r = 1.0f;
mat.Specular.g = 1.0f;
mat.Specular.b = 1.0f;
mat.Specular.a = 1.0f;
mat.Power = 50.0f;

// 设置放射光的 RGBA 颜色值。
mat.Emissive.r = 0.0f;
mat.Emissive.g = 0.0f;
mat.Emissive.b = 0.0f;
mat.Emissive.a = 0.0f;
```

在准备 D3DMATERIAL9 结构之后，应用程序通过调用 IDirect3DDevice9::SetMaterial 方法将材质属性应用于渲染设备。这个方法接收一个准备好的 D3DMATERIAL9 结构的地址作为它唯一的参

数。要更新设备的材质属性，应用程序可以根据需要用新的属性调用IDirect3DDevice9::SetMaterial。以下示例代码显示了这个过程。

```
// 本示例代码使用本节前面定义的 mat 变量保存的材质属性。
// 假设 pd3dDev 为一个指向 IDirect3DDevice9 接口的有效指针。
HRESULT hr;
hr = pd3dDev->SetMaterial(&mat);
if(FAILED(hr))
{
    // 错误处理的代码放在这里。
}
```

当应用程序创建 Direct3D 设备时，当前材质被自动设为如下表所示的默认值。

成员	值
Diffuse	(R:1, G:1, B:1, A:0)
Specular	(R:0, G:0, B:0, A:0)
Ambient	(R:0, G:0, B:0, A:0)
Emissive	(R:0, G:0, B:0, A:0)
Power	(0.0)

取得材质属性

通过对设备调用IDirect3DDevice9::GetMaterial方法，应用程序可以取得当前设备正在使用的材质属性。与IDirect3DDevice9::SetMaterial方法不同的是，IDirect3DDevice9::GetMaterial无需准备工作。IDirect3DDevice9::GetMaterial方法接收一个D3DMATERIAL9 结构的地址，将描述当前材质属性的信息填入该结构，然后返回。

```
// 在这个例子中，假设 pd3dDev 变量为一个指向 IDirect3DDevice9 接口的有效指针。
HRESULT hr;
D3DMATERIAL9 mat;

hr = pd3dDev->GetMaterial(&mat);
if(FAILED(hr))
{
    // 错误处理的代码放在这里。
}
```

更多关于材质属性的信息，请参阅：

材质属性

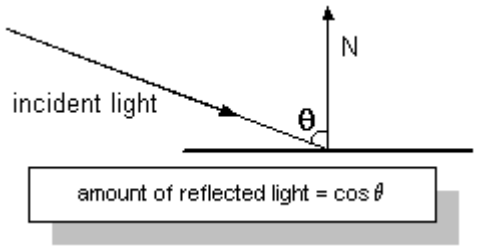
注意 如果应用程序不指定渲染使用的材质属性，系统会使用默认的材质。默认的材质反射所有漫反射光——也就是白色——但没有环境反射和镜面反射，也没有放射色。

材质属性

材质属性详细描述了材质的漫反射系数、环境反射系数、发光度和镜面反射高光的特性。Microsoft® Direct3D®使用D3DMATERIAL9结构保存所有材质属性信息。材质属性会影响Direct3D对使用该材质的多边形做光栅化操作得到的颜色。除了镜面反射属性，其余每个属性都用一个RGBA颜色描述，表示该材质对某一给定类型的光的红、绿和蓝成分的反射度，以及一个阿尔法混合因子——RGBA颜色的阿尔法成员。材质的镜面反射属性用两部分描述：颜色和幂指数。更多信息，请参阅光和材质的颜色值。

漫反射和环境反射

D3DMATERIAL9 结构的 Diffuse 和 Ambient 成员描述了材质如何反射场景中的环境光和漫反射光。因为大多数场景包含的漫反射光比环境光要多，所以在决定最终颜色的过程中漫反射所起的作用最大。另外，因为漫反射具有方向性，漫反射光的入射角会影响到整个反射光的强度。当光的入射方向与顶点法向平行时，漫反射最强。随着入射方向与顶点法向之间夹角的增大，漫反射效果逐渐减少。反射光的数量是入射光与顶点法向之间夹角的余弦值，如下所示。



环境反射和环境光一样，没有方向性。环境反射对被渲染物体最终的颜色影响较小，但它确实会影响最终的颜色，最为明显的就是当材质很少甚至不反射漫反射光时。材质的环境反射受场景中的环境光的影响，场景的环境光通过调用IDirect3DDevice9::SetRenderState方法，并用D3DRS_AMBIENT作为参数设置。

漫反射和环境反射一起用来决定物体反射的颜色，而且通常是相同的值。例如，要渲染一个蓝色水晶物体，应用程序可以创建一个只反射漫反射光和环境光中蓝色成分的材质。当水晶放在有白光的地方，它看起来会是蓝色的。但是，在一个只有红光的地方，同一个水晶看起来就会是黑色的，因为它的材质不反射红光。

放射

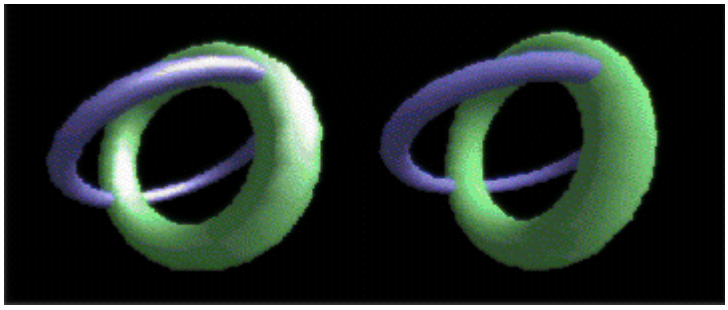
材质可以用来使被渲染的物体看起来像是自身发光的。D3DMATERIAL9 结构的 Emissive 成员就是用来描述物体发出的光的颜色和透明度的。放射会影响物体的颜色，也可以使暗的材质变亮并部分呈现出所发光的颜色。

应用程序可以使用材质的发光属性给一个物体增加发光的感觉，这不会导致因为给场景增加光源而带来的计算开销。在蓝色水晶的例子中，如果希望水晶显得亮，但又不照射到场景中的其它物体上，发光属性就很有用。记住，具有发光属性的材质不发射可以被场景中其它物体反射的光。要得到可以被其它物体反射的光，需要在场景中另外增加一个光源。

镜面反射

镜面反射在物体表面产生高光，这使它们看起来像是有光泽的。D3DMATERIAL9 结构包含了两个描述镜面反射高光和整体亮度的成员。应用程序可以通过 Specular 成员将镜面反射高光设为想要的 RGBA 颜色值——最常用的颜色是白色或淡灰色。应用程序可以设置 Power 成员的值控制镜面反射效果的剧烈程度。

镜面反射高光可以产生生动的效果。还是以蓝色水晶为例：较大的 Power 值会产生更为明亮的镜面反射高光，使水晶看起来光芒四射。较小的值增大了产生效果的区域，造成不太明亮的反射，这使水晶看起来像灰色的。要使物体的表面真正是粗糙的，只需将 Power 成员设为零，并将 Specular 成员设为黑色。为了根据自己的需要产生真实的反射，可以试验不同的反射系数。下图描绘了两个相同的建模。左边使用了 Power 为 10 的镜面反射，而右边没有镜面反射。



纹理

纹理是增强计算机生成的三维图像的真实感的有力工具。Microsoft® Direct3D®支持广泛的纹理特性，并使开发人员可以很方便地使用高级纹理技术。

本节讲述如何使用纹理。

纹理的基本概念

纹理坐标

纹理过滤

纹理资源

纹理环绕

纹理混合

表面

以下主题将更详细地介绍另外的纹理功能。

Mipmap的自动生成

自动纹理管理

压缩纹理资源

使用纹理时需要考虑的硬件问题

立体纹理资源

要提高性能，可以考虑使用动态纹理。动态纹理在每一帧都可以被锁定，写入及解锁。更多信息请参阅使用动态纹理。

纹理的基本概念

早期计算机生成的三维图像看起来往往像是发亮的塑料，虽然这在当时也是比较先进的，但是它们缺乏各种纹路——如磨损、裂痕、指纹和污渍等，而这些纹路会增加三维物体的真实感。近年来，纹理已经在开发人员中得到普及并作为增强计算机生成的三维图像的真实感的工具。

词语“纹理”在日常使用中表示物体的光滑度或粗糙度，但是在计算机图形学中，纹理指的是一张表示物体表面细节的位图。

因为 Direct3D 中所有纹理都是位图，所以可以把任何位图贴到 Direct3D 图元的表面。例如，应用程序可以创建物体并使它们的表面看起来有木纹的样式。可以把草、泥土和岩石等纹理贴在构成山的图元的表面，这样就能得到看起来很真实的山坡。应用程序也可以用纹理创建其它的效果，如：路边的路标，悬崖边的岩层，或是地面上的大理石。

另外，Direct3D支持更高级的纹理技术，如纹理混合（包含或不含透明度）和光照贴图。更多信息请参阅纹理混合和用纹理实现光照贴图。

如果应用程序创建一个 HAL 设备或软件设备，那么可以使用 8、16、24 或是 32 位纹理。

以下主题包含了更多的信息。

纹理寻址模式

无效纹理区域

纹理调色板

纹理寻址模式

Microsoft® Direct3D®应用程序可以把纹理坐标值赋给任何图元的任何顶点。更多细节，请参阅纹理坐标。一般来说，应用程序赋给顶点的u、v纹理坐标值在 0.0 到 1.0 范围内，闭区间。但是，通过把纹理坐标值赋为此范围外的值，应用程序可以创建某些特殊纹理效果。

通过设置纹理寻址模式，应用程序可以控制当纹理坐标位于范围[0.0, 1.0]外时希望 Direct3D 执行何种操作。例如，应用程序可以设置寻址模式，使纹理平铺于图元表面。下面的主题包含了更多的细节。

Direct3D使应用程序可以进行纹理环绕，很重要的一点是要注意把纹理寻址模式设为D3DTADDRESS_WRAP与进行纹理环绕并不相同。把纹理寻址模式设为D3DTADDRESS_WRAP会使源纹理的多个复本被贴到当前图元的表面，而启用纹理环绕则会改变系统对贴有纹理的多边形进行光栅化的方式。更多细节，请参阅纹理环绕。

启用纹理环绕实际上使位于[0.0, 1.0]范围之外的纹理坐标无效，在这种情况下，对无效的纹理坐标进行光栅化操作将导致未定义的结果。当启用纹理环绕时，不会使用纹理寻址模式，同时应用程序应该注意不要给出小于 0.0 或大于 1.0 的纹理坐标。

设置寻址模式

应用程序可以通过调用IDirect3DDevice9::SetSamplerState方法设置每个纹理层的纹理寻址模式，只需把纹理层的标识作为第一个参数，并把第二个参数设置为D3DSAMP_ADDRESSU，D3DSAMP_ADDRESSV或D3DSAMP_ADDRESSW，就可以分别改变u，v或w寻址模式。

IDirect3DDevice9::SetSamplerState方法的第三个参数决定要设置的模式，这是一个D3DTEXTUREADDRESS枚举类型值。要取得某一纹理层当前的纹理寻址模式，只需调用IDirect3DDevice9::GetSamplerState方法，把第二个参数设置为D3DTEXTURESTAGESTATETYPE枚举类型值，并把第三个参数设置为用于保存返回的寻址模式的变量的地址。

设备限制

虽然系统允许纹理坐标位于闭区间 0.0 到 1.0 之外，但硬件限制通常会决定纹理坐标究竟可以超出该范围多少。当应用程序取得设备能力时，渲染设备通过D3DCAPS9结构的MaxTextureRepeat成员表明这个限制，这个成员的值描述了设备允许的纹理坐标的全部范围。例如，若这个值为 128，则输入的纹理坐标必须保持在从-128.0 到+128.0 之间，若输入顶点的纹理坐标位于这个范围外，则纹理坐标是无效的。对自动生成的纹理坐标和由纹理坐标变换得到的纹理坐标也有同样的限制。

对 MaxTextureRepeat 的解释同时还受 D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE 能力位的影响。若设置了这个能力位，则 D3DCAPS9 结构的 MaxTextureRepeat 成员的值正如前面描述的一样。但是，若没有设置该能力位，则纹理循环的限制还取决于纹理坐标寻址的那个纹理的大小。在这种情况下，MaxTextureRepeat 必须除以当前 mipmap 中最大纹理的大小。例如，若纹理大小为 32，而 MaxTextureRepeat 的值为 512，则实际有效的纹理坐标的范围是 512/32 = 16，因此传给该设备的纹理坐标必须在范围-16.0 到+16.0 之间。

以下主题包含了更多有关纹理寻址的信息：

环绕纹理寻址模式

镜像纹理寻址模式

截取纹理寻址模式

边框颜色纹理寻址模式

环绕纹理寻址模式

环绕纹理寻址模式由D3DTEXTUREADDRESS枚举类型的D3DTADDRESS_WRAP成员表示，它会使 Microsoft® Direct3D®在纹理坐标的整数边界重复使用该纹理。例如，设想应用程序创建了一个方的图元并把纹理坐标指定为(0.0,0.0)，(0.0,3.0)，(3.0,3.0)和(3.0,0.0)，把纹理寻址模式

设置为D3DTADDRESS_WRAP会使纹理在u和v方向都重复三次。

这种纹理寻址模式的效果与镜像纹理寻址模式有些相似，却又明显不同。更多信息，请参阅镜像纹理寻址模式。

镜像纹理寻址模式

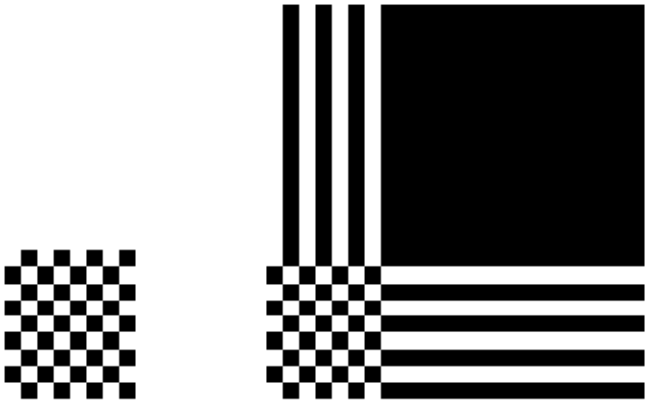
镜像纹理寻址模式由D3DTEXTUREADDRESS枚举类型的D3DTADDRESS_MIRROR成员表示，它会使Microsoft® Direct3D®在纹理坐标的整数边界先对纹理进行镜像然后再重复使用。例如，设想应用程序创建了一个方的图元并把纹理坐标指定为(0.0,0.0)，(0.0,3.0)，(3.0,3.0)和(3.0,0.0)，把纹理寻址模式设置为D3DTADDRESS_MIRROR会使纹理在u和v方向都重复三次，每一行和每一列的纹理都是相邻行和列的纹理的镜像。

这种纹理寻址模式的效果与环绕纹理寻址模式有些相似，却又明显不同。更多信息，请参阅环绕纹理寻址模式。

截取纹理寻址模式

截取纹理寻址模式由D3DTEXTUREADDRESS枚举类型的D3DTADDRESS_CLAMP成员表示，它会使Microsoft® Direct3D®把纹理坐标截取到[0.0, 1.0]范围内，也就是说，这种模式只应用纹理一次，然后就重复使用纹理边缘处像素的颜色。例如，设想应用程序创建了一个方的图元并把纹理坐标指定为(0.0,0.0)，(0.0,3.0)，(3.0,3.0)和(3.0,0.0)，把纹理寻址模式设置为D3DTADDRESS_CLAMP会使纹理只被应用一次，列的顶端和行的末端处像素的颜色被相应地延伸至图元的顶端和右边。

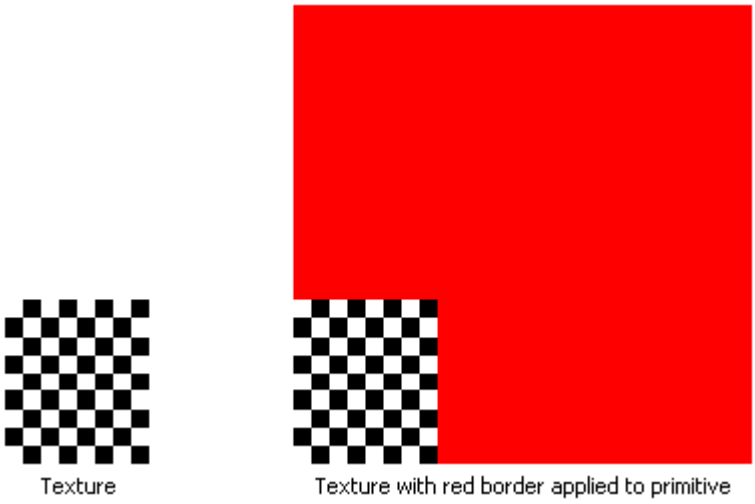
下图描绘了截取寻址模式。



边框颜色纹理寻址模式

边框颜色纹理寻址模式由D3DTEXTUREADDRESS枚举类型的D3DTADDRESS_BORDER成员表示，该寻址模式会使Microsoft® Direct3D®对于位于[0.0, 1.0]范围之外的纹理坐标使用一个被称为边框颜色的指定颜色。

下图描绘了边框颜色纹理寻址模式，这里应用程序指定红色为纹理的边框颜色。



应用程序可以通过调用IDirect3DDevice9::SetSamplerState方法设置边框颜色。在调用时要把第一个参数设为想要设置的纹理层的标识，把第二个参数设为D3DSAMP_BORDERCOLOR纹理层状态值，并把第三个参数设为以RGBA形式表示的新的边框颜色。

无效纹理区域

通过给纹理指定无效区域，应用程序可以对需要复制纹理的哪些子集进行优化，只有那些被标记为无效的区域才会被IDirect3DDevice9::UpdateTexture方法更新。当创建纹理时，整个纹理被标记为无效的。只有以下几种操作可以改变纹理的无效状态。

给一个纹理添加一个无效区域。

锁定纹理中的一些区域。此操作会把被锁定的区域添加到无效区域中，如果应用程序明确知道哪些是真正的无效区域，那么也可以关闭对无效区域的自动更新。

将纹理作为目标表面进行更新的话会把整个纹理标记为无效的。

对纹理调用 IDirect3DDevice9::UpdateTexture 方法会清除该纹理的所有无效区域。

为了得到设备上下文（device context）而调用IDirect3DDevice9::GetDC。

对于 mipmap 纹理而言，无效区域被设在最高一级的纹理上，为了最小化对 mipmap 纹理中每一级的纹理更新所需复制的字节数，IDirect3DDevice9::UpdateTexture 方法可以扩展无效区域并沿

mipmap 链更新子纹理。注意子级中无效区域的纹理坐标被向上舍入，也就是说，它们的小数部分被向上取整到纹理中最近的像素。

因为每种类型的纹理具有不同类型的无效区域,所以每种类型的纹理都有相应的方法表示无效区域。二维纹理使用矩形，立体纹理使用立方体。

IDirect3DCubeTexture9::AddDirtyRect

IDirect3DTexture9::AddDirtyRect

IDirect3DVolumeTexture9::AddDirtyBox

把以上方法的 *pDirtyRect* 或 *pDirtyBox* 参数设置为 NULL 会扩大无效区域并使之覆盖整个纹理。每种锁定方法都有D3DLOCK_NO_DIRTY_UPDATE标志，使用这个标志可以防止对纹理无效区域的改变。更多信息，请参阅锁定资源。

如果在锁定操作时可以得到已改变区域的完整集合，那么应用程序应该使用 D3DLOCK_NO_DIRTY_UPDATE 标志。注意，对纹理一个的子级的锁定或复制操作（也就是说，未对纹理的最高一级进行锁定或复制操作）不会更新该纹理的无效区域。当应用程序锁定了纹理的子级而没有锁定纹理的最高一级时，它同样有责任对无效区域进行更新。

纹理调色板

Microsoft DirectX® 9.0 中的Microsoft® Direct3D®通过一组与IDirect3DDevice9对象相关联的 256 色调色板支持调色板纹理（paletted texture）。通过调用 IDirect3DDevice9::SetCurrentTexturePalette方法可以设置当前调色板。当前调色板用于对所有已激活的纹理层中的所有调色板纹理进行颜色转换。IDirect3DDevice9::SetPaletteEntries 方法可以更新调色板中的全部 256 个颜色项。每个颜色项都是一个用D3DFMT_A8R8G8B8 格式表示的PALETTEENTRY结构，默认值为 0xFFFFFFFF。

IDirect3DDevice9 的调色板包含了一个阿尔法通道。若设备设置了 D3DPTEXTURECAPS_ALPHAPALETTE 能力位，则表示该设备支持调色板阿尔法，并可以使用该阿尔法通道。当纹理格式不含阿尔法通道时，就使用调色板阿尔法通道。若设备不支持调色板阿尔法，同时纹理格式也不含阿尔法通道，则使用 0xFF 作为阿尔法值。

系统中最多可以有 65,536 个调色板。因为调色板占用的内存资源与应用程序引用到的最大的调色板编号成正比，所以最好使用从零开始且连续的编号。

纹理坐标

大多数纹理，如位图，都是一个存放颜色值的二维数组，但立方体环境贴图除外，具体细节请参阅立方体环境贴图。数组中的每个颜色值被称为texel。每个texel在纹理中有唯一的地址，可以认为这个地址是行和列的编号，它们分别被标记为u和v。

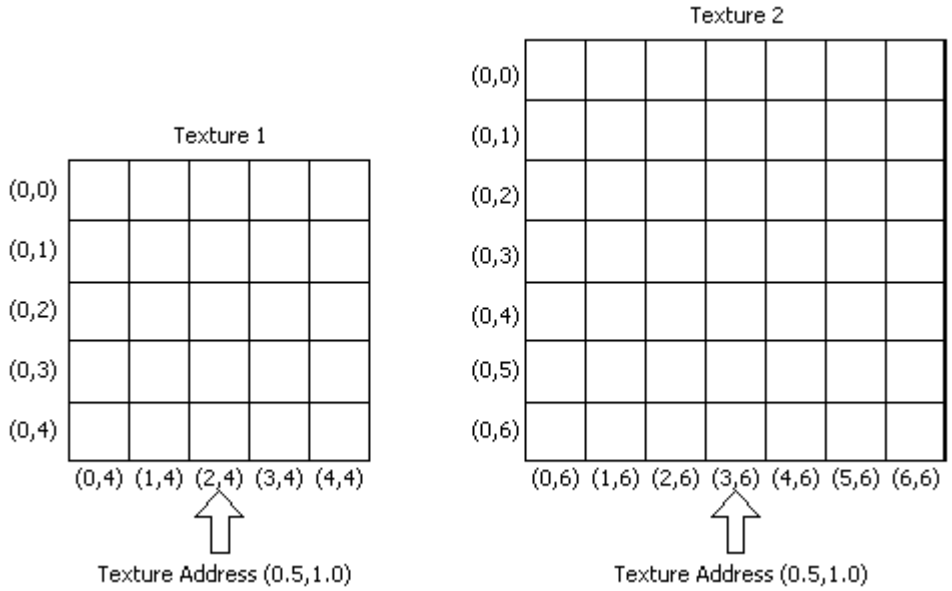
纹理坐标位于纹理空间中，也就是说，它们相对于纹理中的位置(0,0)点。当把纹理贴到三维空间中图元的表面时，纹理的 texel 必须先被映射到对象坐标系，然后再变换到屏幕坐标系，或像素的位置。

将 Texel 映射到屏幕空间

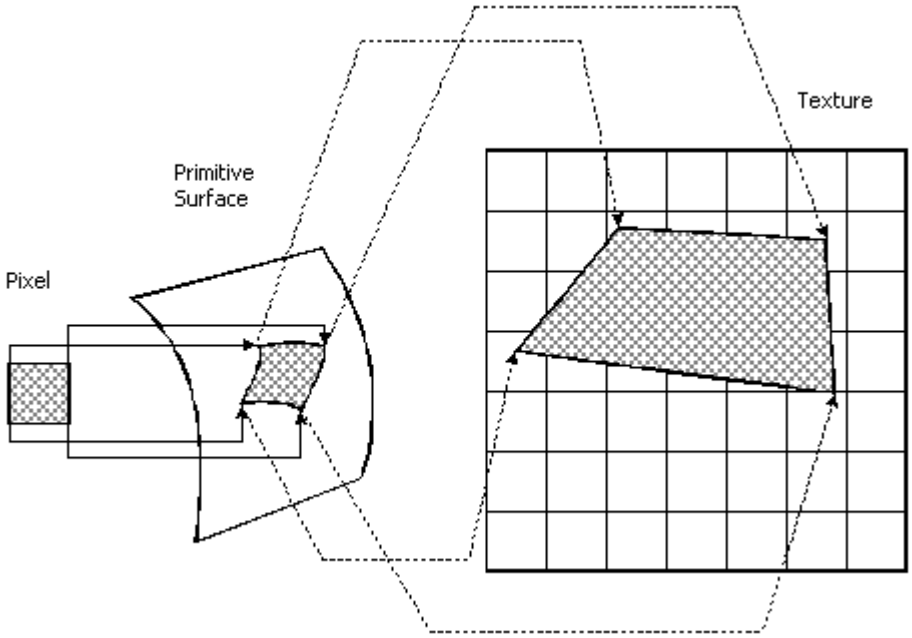
Microsoft® Direct3D®直接把纹理中的texel映射到屏幕空间，这样就省略了中间步骤并极大地提高了效率。这个映射的过程实际上是一个反向映射，也就是说，系统根据每个像素在屏幕空间中的位置计算该像素在纹理空间中相应的texel的位置，然后对位于该点或该点附近的纹理颜色进行取样。取样的过程被称为纹理过滤。更多信息请参阅纹理过滤。

纹理中每个texel的位置可以用它的texel坐标表示。但是为了把texel贴到图元表面，Direct3D需要所有的纹理中的texel具有相同的地址范围，所以Direct3D使用了一种通用的寻址方法。在这种寻址方法中，所有texel的地址都在闭区间 0.0 到 1.0 内。Direct3D用u，v值表示纹理坐标，这和用x，y坐标表示二维笛卡尔坐标系非常相似。从技术上讲，系统事实上可以处理 0.0 到 1.0 范围外的纹理坐标，系统根据应用程序设置的纹理寻址模式来进行此类处理。更多信息，请参阅

纹理寻址模式。
采用这种方法的结果是相同的纹理地址在不同的纹理中会映射到不同的 texel 坐标。在下图中，正在使用的纹理地址是(0.5,1.0)。但是，因为纹理的大小不同，所以该纹理地址映射到不同的 texel。左边纹理的大小为 5x5，纹理地址(0.5,1.0)映射到 texel (2,4)。右边纹理的大小为 7x7，纹理地址(0.5,1.0)映射到 texel (3,6)。

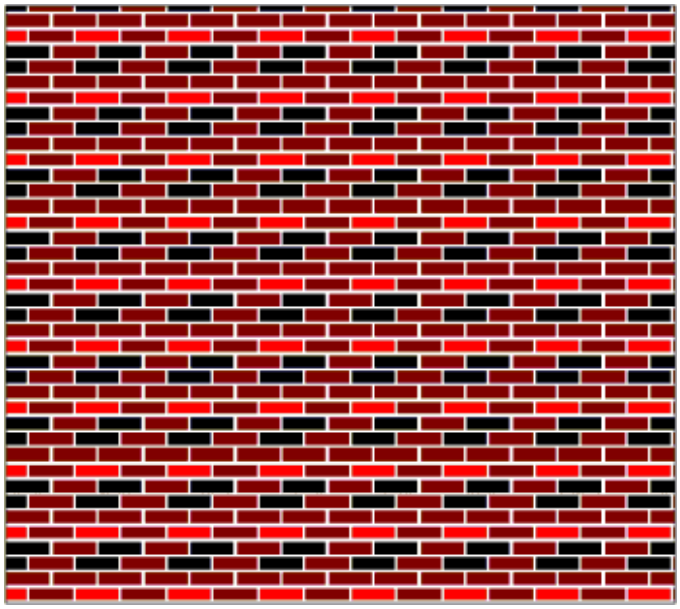


下图描述了一个经简化的texel映射过程。这个示例显然非常简单，要了解更多细节信息，请参阅直接把Texel映射到像素。



本例中，图的左边显示了一个被理想化为正方形色块的像素。像素四角的地址被映射到对象空间中的三维图元，因为三维场景中图元的形状及观察角度的不同，像素的形状常常会被扭曲。像素四角所对应图元表面的区域然后被映射到纹理空间，这个映射过程会再次扭曲像素的形状。像素的最终颜色根据像素映射的区域所覆盖的 texel 计算得到。通过设置纹理过滤方法，应用程序可以控制让 Direct3D 使用何种方法计算像素颜色。更多信息，请参阅纹理过滤。

应用程序可以直接给顶点指定纹理坐标，这使应用程序可以控制把纹理的哪些部分贴到图元上。例如，设想应用程序创建了一个与下面的纹理大小完全相同的图元，本例中，如果应用程序希望把整张纹理贴到整面墙上，那么应用程序给图元的顶点指定的纹理坐标就应该是(0.0,0.0)，(1.0,0.0)，(1.0,1.0)，和 (0.0,1.0)。



如果应用程序决定把墙的高度减半，应用程序仍可以把整张贴图贴到稍小的墙上，但这会挤压纹理并使之扭曲，或者应用程序也可以重新指定纹理坐标，使 Direct3D 使用纹理的下半部分。

如果应用程序为了把纹理贴到稍小的墙上而决定挤压或拉伸纹理，那么应用程序使用的纹理过滤方法会影响最终图像的质量。更多信息，请参阅纹理过滤。

如果应用程序决定重新指定纹理坐标并使 Direct3D 使用纹理的下半部分，那么在本例中应用程序给图元的顶点指定的纹理坐标应该是(0.0,0.5)，(1.0,0.5)，(1.0,1.0)，和 (0.0,1.0)，这样 Direct3D 就会把纹理的下半部分贴到墙上。

顶点的纹理坐标有可能大于 1.0，如果应用程序给顶点指定的纹理坐标不在闭区间 0.0 到 1.0 范围内，那么应用程序还应该设置纹理寻址模式。更多信息，请参阅纹理寻址模式。

纹理坐标和纹理层

纹理坐标通过纹理层与纹理联系在一起。纹理通过SetTexture(stageIndex, pTextre) 被设定到某一纹理层。请参阅IDirect3DDevice9::SetTexture。

一个弹性顶点格式码最多可以定义八组纹理坐标，纹理坐标数据由用户在顶点数据中提供，数据通过索引值 0 到 7 来引用。最多可以有八个纹理混合层，一张纹理通过 SetTexture(stageIndex, pTexture)与某一纹理层联系在一起。

完成以上操作后，任意一组纹理坐标可以被任意一纹理层使用。每一组纹理坐标通过 SetTextureStageState(stageIndex, D3DTSS_TEXCOORDINDEX, textureCoordinateIndex)与某一纹理层联系在一起。请参阅IDirect3DDevice9::SetTextureStageState。通过这种方法，可以设置纹理混合层使它们使用任意一张纹理和任意一组纹理坐标。多个纹理层可以使用同一张纹理，或同一组纹理坐标。

以下主题包含了更多的信息。

直接把Texel映射到像素

纹理坐标的格式

纹理坐标的处理

直接把 Texel 映射到像素

应用程序经常需要把纹理贴到几何体上并使 texel 直接映射到屏幕上的像素。例如，以一个需要在纹理中显示文本的应用程序为例，为了使纹理中的文本能清晰地显示，应用程序需要以某种方式确保映射到几何体上的 texel 不受纹理过滤的影响。如果无法保证这一点，那么得到的图像通常会是模糊的，如果纹理过滤方法为最近点取样，那么可能会产生粗糙边缘。

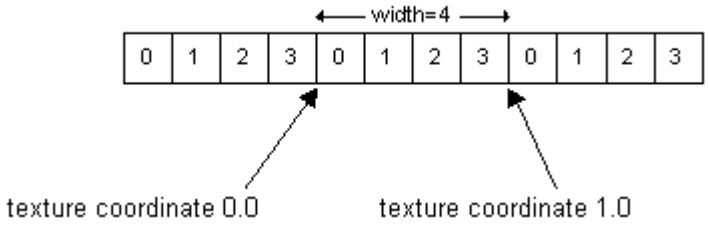
为了统一像素和纹理取样，并同时支持图像和纹理过滤，Microsoft® Direct3D®的像素和纹理取样规则经过了精心定义，但这也使得把纹理中的 texel 直接映射到屏幕上的像素成为一个相当有意义却又艰难的挑战。要战胜这个挑战，需要透彻地理解 Direct3D 如何把用浮点数表示的纹理坐标映射到光栅化器使用的整数像素坐标。

为了把用浮点数表示的纹理坐标映射到 texel 地址，Direct3D 执行下面的计算。

$$T_x = (u \times M_x) - 0.5$$
$$T_y = (v \times M_y) - 0.5$$

在这些公式中，T_x 和 T_y 为水平/垂直方向的输出 texel 坐标，u 和 v 为顶点提供的水平/垂直方向的纹理坐标。M_x 和 M_y 元素表示当前 mipmap 级水平/垂直方向的 texel 的数量。本节剩余部分将主要讨论在水平方向上从 texel 到像素的映射，垂直方向上的映射与水平方向完全相同。

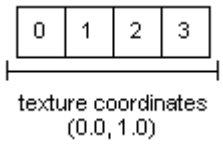
把纹理坐标 0.0 和 1.0 代入以上公式，会使纹理坐标 0.0 映射到本次纹理迭代的第一个 texel 和上次纹理迭代的最后一个 texel 的中间，纹理坐标 1.0 会被映射到本次纹理迭代的最后一个 texel 和下次纹理迭代的第一个 texel 的中间。对于一个宽度为 4 的迭代纹理，在 mipmap 的第 0 级，下图显示了系统把坐标 0.0 和 1.0 映射到哪里。



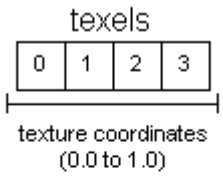
理解了这种映射方式，应用程序只需给几何体的屏幕空间坐标加上一个偏移量，就可以强制系统把每个 texel 映射到相应的像素。例如，要绘制一个四边形并使前面的纹理中的每个 texel 一一映射到屏幕上唯一的像素，应用程序必须使几何体坐标覆盖所有像素，并使每个 texel 的中心正好对应每个像素的中心，这样得到的结果就是应用程序常要追求的一对一映射。

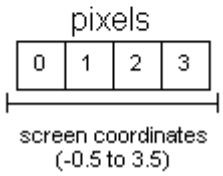
为了把宽度为 4 的纹理映射到像素坐标 0 到 3，可以用两个三角形绘制一个四边形，三角形在屏幕空间的坐标为-0.5 到 3.5，纹理坐标为 0.0 到 1.0。以屏幕空间坐标为 0.0 的像素为例，因为 0.0 与第一个顶点，位于屏幕空间-0.5，相距半个像素，而总的宽度为 4.0，迭代后的纹理坐标为 0.125（译注：每个像素的宽度为 0.25=1/4，半个像素的宽度为 0.125），然后用纹理的大小，此处为 4，对纹理坐标进行缩放，得到的结果坐标为 0.5。再减去偏移量 0.5 即得到纹理地址为 0.0，该地址完全对应于贴图中的第一个 texel。

再概括一下，纹理坐标覆盖纹理贴图的两边，并平均分布在它们之间。下图显示了这种映射，纹理的宽度为 4。



系统以相同的方法归一化像素坐标和纹理坐标，因此，如果顶点与要渲染的像素重叠，且顶点使用的纹理坐标为 0.0 和 1.0，那么像素和顶点就可以对齐。如果它们的大小相同且排列整齐，那 texel 和像素就可以完全对应，如下图所示。





纹理坐标的格式

Microsoft® Direct3D®中的纹理坐标可以包含一个、两个、三个或四个用浮点数表示的元素，用来寻址不同大小的纹理。一维纹理——纹理表面的大小为 1xn个texel——通过一个纹理坐标寻址。最常见的情况是二维纹理，通过两个被称为*u*和*v*的纹理坐标寻址。Direct3D支持两种三维纹理，*立方体环境贴图*和*立体贴图*。立方体环境贴图不是真正三维的，但使用一个三元素向量寻址。更多细节，请参阅立方体环境贴图。

如顶点格式中所述，应用程序把纹理坐标编码在顶点格式中。顶点格式可以包含多组纹理坐标。可以用FVF码D3DFVF_TEX0 到D3DFVF_TEX8 描述不包含纹理坐标或最多可包含八组纹理坐标的顶点格式。

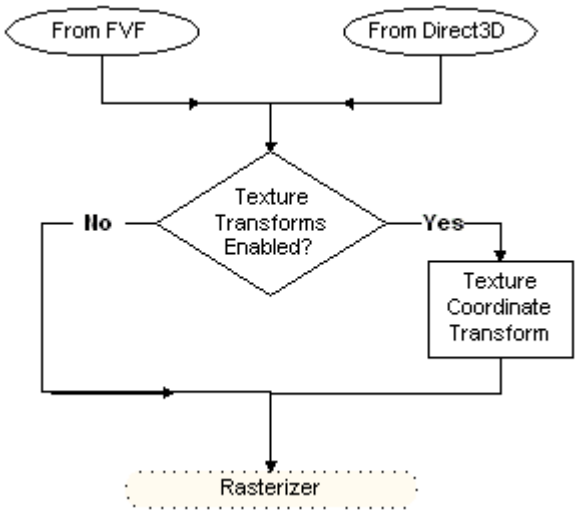
每组纹理坐标可以有一到四个元素。D3DFVF_TEXTUREFORMAT1 到D3DFVF_TEXTUREFORMAT4 标志用来描述一组纹理坐标中元素的个数，但这些标志不能直接使用，D3DFVF_TEXCOORDSIZEn系列宏使用这些标志创建位掩码，用来在顶点格式中描述某组纹理坐标中元素的个数。这些宏带一个参数，表示要设置元素个数的那组纹理坐标的索引值。以下示例代码显示了如何使用这些宏。

```
// 这个顶点格式包含两组纹理坐标。第一组（索引为 0）包含 2 个元素，
// 第二组包含一个元素。顶点格式描述应该是：
//     dwFVF = D3DFVF_XYZ  | D3DFVF_NORMAL | D3DFVF_DIFFUSE | D3DFVF_TEX2 |
//             D3DFVF_TEXCOORDSIZE2(0) | D3DFVF_TEXCOORDSIZE1(1);
//
typedef struct CVF
{
    D3DVECTOR position;
    D3DVECTOR normal;
    D3DCOLOR  diffuse;
    float     u, v;    // 第一组纹理坐标，二维
    float     t;       // 第二组纹理坐标，一维
} CustomVertexFormat;
```

注意除了立方体贴图和立体贴图外，光栅化器无法使用两个以上的元素寻址纹理。应用程序最多可以提供三个元素给一组纹理坐标，但只有当纹理是立方体贴图或立体贴图，或使用了D3DTTFF_PROJECTED纹理变换标志时多余的元素才会被用到。D3DTTFF_PROJECTED标志会使光栅化器把前两个元素除以第三个（或第*n*个）元素。更多信息，请参阅纹理坐标变换。

纹理坐标的处理

下图显示了纹理坐标从数据源，经过处理然后到达光栅化器的流程。



系统可以从两个数据源得到纹理坐标。对于某一层纹理，应用程序可以使用包含在顶点格式（D3DFVF_TEX1 到D3DFVF_TEX8）中的纹理坐标，也可以使用Microsoft® Direct3D®自动生成的纹理坐标。对于后一种情况，请参阅自动生成的纹理坐标。如果当前纹理层的D3DTSS_TEXTURETRANSFORMFLAGS纹理层状态为D3DTTFF_DISABLE（默认值），那么系统不会对输入的纹理坐标进行变换。如果D3DTSS_TEXTURETRANSFORMFLAGS为任何其它值，那么系统会用该纹理层的变换矩阵对该输入纹理坐标进行变换。

D3DTEXTURETRANSFORMFLAGS枚举类型定义了D3DTSS_TEXTURETRANSFORMFLAGS纹理层状态的有效值。除了D3DTTFF_DISABLE标志不进行纹理坐标变换外，该枚举类型值用于设定系统传送至光栅化器的输出坐标的数量。D3DTTFF_COUNT1 到D3DTTFF_COUNT4 标志告诉系统把输出纹理坐标中的一个、两个、三个或四个元素传送至光栅化器。

D3DTTFF_PROJECTED 标志有些特别：它告诉系统纹理坐标将用于经过投影的纹理。把 D3DTTFF_PROJECTED 标志与 D3DTEXTURETRANSFORMFLAGS 的其它成员一起使用可以告诉系统在光栅化操作前把所有元素除以最后一个元素。例如，当显式使用三元素纹理坐标，或纹理变换产生三元素纹理坐标时，应用程序可以同时使用 D3DTTFF_COUNT3 和 D3DTTFF_PROJECTED 标志，这会使光栅化器把前两个元素除以最后一个元素，并得到寻址二维纹理所需的二维纹理坐标。

注意除了立方体贴图和立体贴图外，光栅化器无法使用多于两个元素的纹理坐标。如果应用程序提供的元素比寻址当前纹理层所需的多，那么多余的元素会被忽略。当把二维纹理坐标用于一维纹理时也是这样。

以下主题包含了更多的信息。

自动生成的纹理坐标

纹理坐标变换

特效

自动生成的纹理坐标

系统可以使用经过变换的摄像机空间中的位置或顶点的法向量作为纹理坐标，也可以计算用于寻址立方体贴图的三元素向量。与应用程序在顶点数据中明确给出的纹理坐标一样，应用程序可以使用自动生成的纹理坐标作为纹理变换的输入。

通过无需在顶点格式中显式地给出纹理坐标，自动生成的纹理坐标可以显著降低几何数据所需的带宽。在许多情况下，系统生成的纹理坐标可以和纹理变换一起使用以生成特效。当然这只是一种特殊用途，大多数情况下应用程序还是要用显式给出的纹理坐标。

设定自动生成的纹理坐标

C++应用程序用D3DTSS_TEXCOORDINDEX纹理层状态（来自D3DTEXTURESTAGESTATETYPE）控制系统

如何产生纹理坐标。
一般来说，这个状态告诉系统使用编码在顶点格式中的某组特定的纹理坐标。当应用程序给这个状态指定的值包含 D3DTSS_TCI_CAMERASPACENORMAL，D3DTSS_TCI_CAMERASPACEPOSITION，或 D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR 标志时，系统执行的操作会完全不同。如果使用了这些标志中的任意一个，那么纹理层会忽略在顶点格式中的给出的纹理坐标，并优先使用系统生成的坐标。下表列出了每个标志的意义。
D3DTSS_TCI_CAMERASPACENORMAL
使用变换到摄像机空间的顶点法向作为输入纹理坐标。
D3DTSS_TCI_CAMERASPACEPOSITION
使用变换到摄像机空间的顶点位置作为输入纹理坐标。
D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR
使用变换到摄像机空间的反射向量作为输入纹理坐标。反射向量根据输入顶点位置和法向量计算得到。
前面的这些标志是互斥的。如果应用程序指定了其中一个标志，那么应用程序还可以指定一个索引值，系统用这个索引值决定纹理环绕模式。
以下示例代码显示了如何在 C++应用程序中使用这些标志。

```
/*
 * 在本例中，d3dDevice 变量为指向 IDirect3DDevice9 接口的有效指针。
 *
 * 在当前纹理层使用顶点位置（摄像机空间）作为输入纹理坐标，
 * 纹理环绕模式在 D3DRENDERSTATE_WRAP1 渲染状态中设置。
 */
d3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
                                 D3DTSS_TCI_CAMERASPACEPOSITION | 1 );
```

自动生成的纹理坐标在作为纹理坐标变换的输入，或使应用程序无需计算用于寻址立方体环境贴图的三元素向量时最为有用。
球形贴图使用一张预计算的（在建模时）纹理贴图，该贴图包含了一个反光的球体反射的整个环境。Microsoft® Direct3D®有一个纹理坐标自动生成特性，使用了 D3DTSS_TCI_CAMERASPACENORMAL渲染状态，该特性会取得变换到摄像机空间的顶点法向，并对它进行纹理变换生成纹理坐标。更多信息请参阅Sphere Map示例。
相关主题
纹理坐标变换
立体环境贴图
纹理坐标变换
Microsoft® Direct3D®设备可以用一个 4x4 矩阵对顶点的纹理坐标进行变换。系统使用相同的方法对纹理坐标和几何体进行变换。任何变换（缩放、旋转、投影、shear 或这些变换的组合）可以用一个 4x4 矩阵完成。
注意Direct3D 不改变经过变换和光照处理的顶点，因此，使用经过变换和光照处理的顶点的应用程序无法让 Direct3D 变换顶点的纹理坐标。
支持硬件变换和光照的设备（TnLHAL 设备）也会对纹理坐标变换进行硬件加速。如果设备不支持硬件变换和光照，那么 Direct3D 会使用几何流水线中与平台相关的优化进行纹理坐标变换。
纹理坐标变换非常有用，它在生成特效的同时避免了对几何体纹理坐标的直接修改。应用程序可以使用简单的平移或旋转矩阵给物体表面的纹理生成动画效果，也可以对 Direct3D 自动生成的纹理坐标进行变换，这样可以简化并可能加速如投影纹理和动态光照贴图等高级特效。另外，在

多层纹理中，应用程序也可以用纹理坐标变换重复使用某一组纹理坐标并将之用于多种用途。
设置及取得纹理坐标变换的信息
和应用程序用于变换几何体的矩阵一样，应用程序可以通过IDirect3DDevice9::SetTransform和IDirect3DDevice9::GetTransform方法设置和取得纹理坐标变换的信息。在调用这些方法时，用D3DTRANSFORMSTATETYPE枚举类型的成员D3DTS_TEXTURE0 到D3DTS_TEXTURE7 标识纹理层 0 到 7。
以下示例代码设置了一个矩阵，对纹理层 0 的纹理坐标进行变换。

```
// 本例中，假设 d3dDevice 变量为指向 IDirect3DDevice9 接口的有效指针。

D3DMATRIX matTrans = D3DXMatrixIdentity( NULL );

// 为希望的变换设置矩阵。
d3dDevice->SetTransform( D3DTS_TEXTURE0, &matTrans );
```

启用纹理坐标变换
D3DTSS_TEXTURETRANSFORMFLAGS纹理层状态控制对纹理坐标的变换。这个纹理层状态的值由D3DTEXTURETRANSFORMFLAGS枚举类型定义。
当 D3DTSS_TEXTURETRANSFORMFLAGS 为 D3DTTFF_DISABLE（默认值）时，纹理坐标变换被禁用。
假设纹理层 0 的纹理坐标变换已启用，以下代码将之禁用。

```
// 本例中，假设 d3dDevice 变量为指向 IDirect3DDevice9 接口的有效指针。

d3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
                                 D3DTTFF_DISABLE );
```

D3DTEXTURETRANSFORMFLAGS 定义的其它值用于启用纹理坐标变换，并控制把结果纹理坐标中的几个元素传送到光栅化器。以下为示例代码。

```
// 本例中，假设 d3dDevice 变量为指向 IDirect3DDevice9 接口的有效指针。

d3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
                                 D3DTTFF_COUNT2 );
```

D3DTTFF_COUNT2 值告诉系统对纹理层 0 进行变换，然后把得到的纹理坐标的前两个元素传送给光栅化器。
D3DTTFF_PROJECTED 纹理变换标志表示用于投影纹理的坐标。当这个标志被设置时，光栅化器会把传入的元素除以最后一个元素。以下为示例代码。

```
// 本例中，假设 d3dDevice 变量为指向 IDirect3DDevice9 接口的有效指针。

d3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
                                 D3DTTFF_COUNT3 | D3DTTFF_PROJECTED );
```

本例告诉系统传送三个纹理坐标元素到光栅化器。光栅化器把前两个元素除以第三个元素，得到寻址纹理所需的二维纹理坐标。
特效
本主题包含了可以用纹理坐标处理实现的特效。
给建模上的纹理产生动画效果（通过变换或旋转）
把纹理坐标作为建模在摄像机空间中的位置的线性函数创建
用立方体环境贴图实现环境贴图
实现投影纹理
给建模表面的纹理生成动画效果（通过变换或旋转）

在顶点格式中定义一组二维纹理坐标。

```
// 使用单纹理，二维纹理坐标。这个位掩码会根据需要
// 被扩展为包含位置，法向和颜色信息。
DWORD dwFVFTex = D3FVF_TEX1 | D3DFVF_TEXCOORDSIZE2(0);
```

设定光栅化器，使之使用二维纹理坐标。

```
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2);
```

定义并设置合适的纹理坐标变换矩阵

```
// M 为要设置的 D3DMATRIX，用于在 U 和 V 方向对纹理坐标进行变换。
//      1   0   0   0
//      0   1   0   0
//      du  dv  1   0 (du 和 dv 每帧都会改变)
//      0   0   0   1

D3DMATRIX M = D3DXMatrixIdentity(); // 在 d3dutil.h 中声明
M._31 = du;
M._32 = dv;
```

把纹理坐标作为建模在摄像机空间中的位置的线性函数创建

用 D3DTSS_TCI_CAMERASPACEPOSITION 标志告诉系统使用顶点在摄像机空间中的位置作为纹理变换的输入。

```
// 为了节省带宽，输入顶点没有纹理坐标。三个纹理坐标用顶点在
// 摄像机空间中的位置(x, y, z)产生。
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_CAMERASPACEPOSITION);
```

告诉光栅化器使用二维纹理坐标。

```
// 使用了两个输出纹理坐标。
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2);
```

定义并设置生成线性函数的矩阵。

```
// 把纹理坐标作为线性函数创建，这样的话：
//      u = Ux*x + Uy*y + Uz*z + Uw
//      v = Vx*x + Vy*y + Vz*z + Vw
// 这种情况下矩阵 M 为：
//      Ux  Vx  0   0
//      Uy  Vy  0   0
//      Uz  Vz  0   0
//      Uw  Vw  0   0

SetTransform(D3DTS_TEXTURE0, &M);
```

用立方体贴图实现环境贴图

用 D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR 标志告诉系统自动产生纹理坐标，并用作立方体贴图的反射向量。

```
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR);
```

告诉光栅化器使用三元素的纹理坐标。

```
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT3);
```

实现投影纹理

使用 D3DTSS_TCI_CAMERASPACEPOSITION 标志告诉系统用顶点在摄像机空间中的位置作为纹理变换矩阵的输入。

```
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_CAMERASPACEPOSITION);
```

创建纹理变换矩阵并执行变换，这超出了本文档的范围，计算机图形工业中有许多文章讨论这个主题。

告诉光栅化器使用三元素投影纹理坐标。

```
// 使用了两个输出纹理坐标。

SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTF_PROJECTED |
D3DTTFF_COUNT3);
```

纹理过滤

Microsoft® Direct3D®在渲染图元时，要把三维图元映射到二维屏幕上。如果图元贴有纹理，那么 Direct3D 必须用该纹理给图元在二维屏幕上对应的每个像素产生一个颜色。对于每个像素，Direct3D 必须从纹理获得一个颜色值，这个过程被称为纹理过滤。

在执行纹理过滤操作时，正在使用的纹理一般会被放大或缩小，换句话说，就是纹理被贴到比它大或比它小的图元上。纹理放大会使多个像素映射到一个 texel，得到的图像可能会有马赛克。纹理缩小会使一个像素映射到多个 texel，得到的图像可能会模糊不清或有锯齿。要解决这些问题，必须在计算像素的颜色时对 texel 的颜色进行一些混合操作。

Direct3D 把复杂的纹理过滤过程简化了，它给应用程序提供了三种类型的纹理过滤——线性过滤、各向异性过滤和 mipmap 过滤。如果应用程序不选择以上三种纹理过滤，那么 Direct3D 使用一种被称为最近点取样的技术。

每种类型的纹理过滤都各有优缺点。例如，线性纹理过滤可能会在最终图像中产生锯齿边缘或马赛克，但它是三种纹理过滤方法中计算量最小的。用 mipmap 纹理过滤通常可以得到最好的效果，尤其是和各向异性过滤一起使用的时候，但是在 Direct3D 支持的纹理过滤技术中，它对内存的需求也最大。

使用纹理接口指针的应用程序应该调用IDirect3DDevice9::SetSamplerState方法设置当前的纹理过滤方法。第一个参数为从 0 到 7 的整数，表示要设置纹理过滤方法的纹理层的索引值。第二个参数为*D3DSAMP_MAGFILTER*，*D3DSAMP_MINFILTER*或*D3DSAMP_MIPFILTER*，分别表示放大、缩小和mipmap过滤。第三个参数为D3DTEXTUREFILTERTYPE枚举类型值，为要设置的纹理过滤方法。

本节介绍了 Direct3D 支持的纹理过滤方法，并被划为以下主题。

最近点取样

线性纹理过滤

各向异性纹理过滤

用Mipmap进行纹理过滤

注意 虽然D3DRENDERSTATETYPE枚举类型中定义的纹理过滤渲染状态已经被纹理层状态取代，但是如果应用程序试图使用这些渲染状态的话，IDirect3DDevice9 与IDirect3DDevice2 不同的是，IDirect3DDevice9::SetRenderState方法不会失败。相反，系统会把这些渲染状态映射到多重纹理的第一层，也就是索引值为 0 的那层。应用程序不应该把老的渲染状态和它们对应的纹理层状态混在一起，这样可能会产生无法预知的结果。

最近点取样

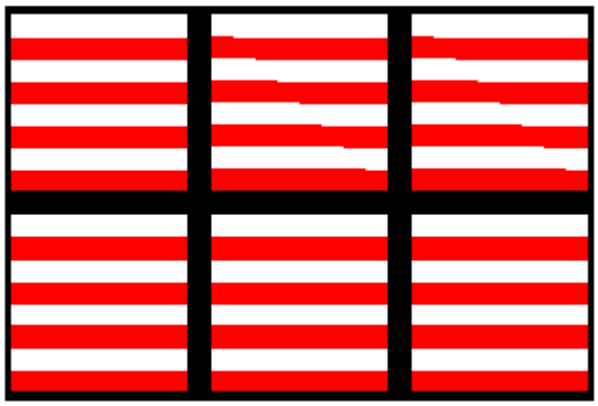
应用程序不一定要使用纹理过滤。应用程序可以让 Microsoft® Direct3D®先计算纹理地址（通常都不是整数），然后使用离该值最近的整数地址处的 texel 的颜色，这个过程被称为*最近点取样*。如果纹理的大小与图元在屏幕上的大小相近的话，那么这将是快速且有效的纹理处理方法，但如果不是这样的话，得到的图像可能会有马赛克、锯齿或模糊不清。

C++应用程序可以调用IDirect3DDevice9::SetSamplerState方法选择最近点取样，只需把第三个参数设置为D3DTEXF_POINT即可。

应用程序在使用最近点取样时应该小心，因为当在两个 texel 间的边界处进行纹理取样时，这种方法有时会产生图形残留物。当使用最近点取样时，系统要么取样一个 texel，要么取样另一个，这样当纹理地址从一个 texel 移向下一个 texel 时，取样得到的 texel 会突然改变。这种效果会导致在最终显示的纹理中出现不希望的图形残留物。当使用线性过滤时，取样得到的 texel 是根据所有邻近的 texel 计算得到的，当纹理地址在邻近的 texel 间移动时，线性过滤会根据当前纹理地址与邻近 texel 间的位置关系，把相邻 texel 混合。

当把非常小的纹理贴到非常大的多边形表面时可以看到这种效果：这个操作通常被称为*纹理放大*（*magnification*）。例如，当使用的纹理看起来像西洋跳棋的棋盘时，最近点取样会得到一个非常大的棋盘，格子之间有清晰的边缘，相比之下，线性纹理过滤得到的图像中棋盘的颜色会沿着多边形逐渐改变。

大多数情况下，为了得到最好的效果，应用程序应该尽量避免使用最近点取样。当今的大多数硬件都为线性过滤做了优化，所以应用程序不必担心因此导致的性能下降。如果应用程序想要的效果一定要用最近点取样——比如用纹理显示可读的文本——那么应用程序应该极度小心，避免在 texel 边界取样，因为那样会导致不想得到的效果。下图显示了这些图形残留物可能的样子。



注意这组图片中右上角的两个方块与其余的不同，可以在它们的对角线上看到明显的偏移。要避免此类图形残留物，开发人员必须熟悉 Direct3D 在最近点取样时使用的纹理取样规则。Direct3D 把闭区间[0.0, 1.0]范围内的浮点纹理坐标映射到 texel 空间中的整数地址[－0.5, *n* － 0.5]范围内，这里 *n* 为给定纹理的大小。得到的纹理地址被舍入到最近的整数，这种映射方法在 texel 边界会产生取样误差。

举个简单的例子，设想应用程序用 D3DTADDRESS_WRAP 纹理寻址模式渲染多边形。根据 Direct3D 使用的映射方法，对于宽度为 4 个 texel 的纹理，纹理在 *u* 方向上的映射如下。

注意这张图中的纹理坐标 0.0 和 1.0，正好在 texel 之间的边界。根据 Direct3D 的映射方法，纹理坐标的范围为[－0.5, 4 － 0.5]，这里 4 为纹理的宽度。在这个例子中，纹理坐标为 1.0 处取样得到的 texel 是 texel 0。但是，如果纹理坐标只比 1.0 稍微小一点，那么取样得到的就会是 texel *n* 而不是 texel 0。

这就意味着用正好等于 0.0 和 1.0 的纹理坐标去放大较小的纹理，并把它贴到整齐排列在屏幕上的三角形时，如果过滤方法为最近点取样方法，那么得到的像素可能会在 texel 之间的边界进行

取样。在纹理坐标计算过程中的任何误差，无论多小，都可能在渲染得到的图像上与 texel 边界对应的部分出现图形残留物。

要完全精确地执行从浮点纹理坐标到整数 texel 地址的映射是很难的，也很耗时，并且通常是不必要的。大多数硬件实现在给三角形内的每个像素计算纹理坐标时使用迭代法。迭代法趋向于隐藏误差，因为误差在迭代过程中被均匀地累积起来。

Direct3D 参考光栅化器在给每个像素计算纹理地址时使用直接赋值法。直接赋值法与迭代法的不同之处在于这种方法产生的误差分布更随机。因为参考光栅化器不进行完全精确的计算，所以这种方法会导致在边界处的取样误差更容易被察觉。

最好的方法是只在必要的时候使用最近点取样。如果应用程序必须使用这种方法，那么最好把纹理坐标稍微偏移一些使之离开边界位置，这样就可以避免残留物的产生。

线性纹理过滤

Microsoft® Direct3D®使用一种被称为双线性过滤的线性纹理过滤方法。和最近点取样一样，双线性过滤首先计算一个texel地址，这通常都不会是整数，然后找到离该地址最近的整数地址。另外，Direct3D渲染模块还会根据最近取样点上、下、左、右的texel计算它们的加权平均值。

可以调用IDirect3DDevice9::SetSamplerState方法选择双线性过滤，只需把第三个参数设为D3DTEXF_LINEAR即可。

各向异性纹理过滤

因为三维物体表面与屏幕间的夹角而造成的纹理扭曲被称为各向异性。当把一个各向异性的图元所对应的像素映射到 texel 时，像素的形状会被扭曲。Microsoft® Direct3D®根据像素被反向映射到纹理空间中的伸长率——也就是长度除以宽度——计量屏幕上像素的各向异性属性。

为了提高渲染质量,应用程序可以把各向异性过滤与线性纹理过滤或mipmap纹理过滤结合在一起使用。应用程序可以调用IDirect3DDevice9::SetSamplerState方法启用各向异性纹理过滤，只需把第三个参数设为D3DTEXF_ANISOTROPIC即可。

应用程序必须同时把degree of anisotropy设为大于一的值。可以调用IDirect3DDevice9::SetSamplerState方法设置这个值。第一个参数为 0 到 7 的纹理层索引值，把第二个参数设为*D3DSAMP_MAXANISOTROPY*，第三个参数为要设置的degree of anisotropy（译注：原文为degree of isotropy）的值。

应用程序只需把degree of anisotropy（译注：原文为degree of isotropy）设为一即可禁用各向异性过滤。要确定degree of anisotropy的允许范围，可以检查D3DCAPS9结构的*MaxAnisotropy*成员。

用 Mipmap 进行纹理过滤

Mipmap 是一系列纹理，每一张纹理都表示同一幅图像，但是分辨率逐渐变低。Mipmap 中的每张图像，或每一级，都比前一级小一半。Mipmap 不必是正方形的。

较高分辨率的 mipmap 图像用于离用户近的物体，而较低分辨率的图像则用于远处的物体。使用 Mipmap 在消耗更多内存的情况下，提高了渲染得到的纹理的质量。

Microsoft® Direct3D®用一连串从属表面表示 mipmap。分辨率最高的纹理在链的最前端，下一级 mipmap 是它的从属表面。依次，每一级 mipmap 的从属表面是它在 mipmap 中的下一级，一直到 mipmap 中分辨率最低的那级。

下图显示了这样的例子。该纹理表示在一个三维第一人称游戏中的一个容器上的标记。创建 mipmap 时，最高分辨率的纹理是 mipmap 中的第一个，mipmap 中每个随后的纹理的宽度和高度都是原来的一半，在这个例子中，最高分辨率的 mipmap 为 256x256。下一级纹理为 128x128，最后一级纹理为 64x64。

这个标记有一个最远可见距离。如果用户离标记很远，那么游戏就用 mipmap 链中最小的纹理显示，在这个例子中就是 64x64 的纹理。

随着用户移动视点并离标记越来越近，游戏会逐渐使用 mipmap 链中更高分辨率的纹理。下图中纹理的分辨率为 128x128。

当视点离标记的距离为所允许的最近距离时，游戏就使用最高分辨率的纹理。

对于纹理而言，这是一种更有效的模拟透视的方法，与在不同的分辨率下用单张纹理进行渲染相比，在不同分辨率下使用多张纹理会更快。

Direct3D 可以确定 mipmap 链中哪一级纹理的分辨率与当前需要的最为接近，并把像素映射到那一级纹理的 texel 空间。如果最终图像需要的分辨率位于 mipmap 链中两级纹理的分辨率之间，Direct3D 会取得这两级 mipmap 中的 texel 并把它们的颜色值混合在一起。

要使用mipmap，应用程序必须创建一个mipmap链。应用程序只需把mipmap链设为当前纹理可以使用mipmap。更多信息，请参阅纹理混合。

下一步，应用程序必须设置 Direct3D 用于取样 texel 的纹理过滤方法。Mipmap 过滤最快的方法就是让 Direct3D 选择最近的 texel，D3DTEXF_POINT 枚举类型值就是用来选择这种方法的。如果应用程序使用 D3DTEXF_LINEAR 枚举类型值，那么 Direct3D 可以产生更好的过滤效果，这会使 Direct3D 选择最近的那级 mipmap，然后根据当前像素在那一级 mipmap 中所映射的 texel 及其附近的 texel 计算加权平均值。

Mipmap 纹理用于减少渲染三维场景所需的时间，同时提高了场景的真实感。但是，mipmap 通常需要大量的内存。

创建 mipmap 链

以下示例代码显示了应用程序如何调用IDirect3DDevice9::CreateTexture方法创建一条五级 mipmap链：256x256, 128x128, 64x64, 32x32, 及 16x16。

```
// 本例假设变量 d3dDevice 为指向 IDirect3DDevice9 接口的有效指针。

IDirect3DTexture9 * pMipMap;
d3dDevice->CreateTexture(256, 256, 5, 0, D3DFMT_R8G8B8, D3DPOOL_MANAGED, &pMipMap);
```

IDirect3DDevice9::CreateTexture的前两个参数为最高一级的纹理的宽和高。第三个参数为mipmap纹理的级数，如果应用程序把它设为零，那么Direct3D会创建一系列表面，每个都是前一个的一半，直到大小为 1x1 为止。第四个参数指定该资源的用途，本例中，零表示不为该资源指定特殊的用途。第五个参数指定纹理的表面格式，该参数为D3DFORMAT枚举类型值。第六个参数为D3DPOOL枚举类型值，表示在把创建的资源放在哪种类型的内存中，除非应用程序使用动态纹理，否则建议使用D3DPOOL_MANAGED。最后一个参数为指向IDirect3DTexture9接口指针的地址。注意Mipmap 链中的每个表面的大小都是前一个表面的一半。如果最高一级 mipmap 的大小为 256x128，那么第二级的 mipmap 就是 128x64，第三级为 64x32，依次类推，直到 1x1（译注：最后几级分别为：4x2,，2x1，1x1）。应用程序在 *Levels* 中要求的 mipmap 级不能使链中的任何 mipmap 的宽和高小于 1。举个简单的例子，如果最高一级的 mipmap 表面为 4x2，那么 *Levels* 的最大允许值就是三，第三层的大小为 1x1。如果 *Levels* 的值大于 3，那么会导致第二级 mipmap 的高度值出现小数，而这是不允许的。（译注：原文表述不够准确，应该是 log2（max（width, height））＜ 0）

选择并显示 mipmap

可以调用IDirect3DDevice9::SetTexture方法把mipmap纹理设置为当前纹理中的第一个纹理,更多信息请参阅纹理混合。

应用程序在选择mipmap纹理后，必须用D3DTEXTUREFILTERTYPE枚举类型值设置 *D3DSAMP_MIPFILTER*取样器状态。之后Direct3D就可以自动执行mipmap纹理过滤。以下示例代码显了如何启用mipmap纹理过滤。

```
d3dDevice->SetTexture(0, pMipMap);
d3dDevice->SetTextureStageState(0, D3DSAMP_MIPFILTER, D3DTEXF_POINT);
```

应用程序也可以手工遍历mipmap链，只需调用IDirect3DTexture9::GetSurfaceLevel方法，并指定要取得的mipmap级即可。以下示例代码从mipmap链的最高一级遍历到最低一级。

```
IDirect3DSurface9 * pSurfaceLevel;
for (int iLevel = 0; iLevel < pMipMap->GetLevelCount(); iLevel++)
{
    pMipMap->GetSurfaceLevel(iLevel, &pSurfaceLevel);
    //Process this level.
    pSurfaceLevel->Release();
}
```

为了把位图数据载入mipmap链中的每个表面，应用程序需要手工遍历mipmap链，一般来说这是遍历mipmap链的唯一原因。应用程序可以通过调用IDirect3DBaseTexture9::GetLevelCount取得mipmap的级数。

纹理资源

纹理资源在IDirect3DTexture9接口中实现。要得到一个指向纹理接口的指针，应该调用IDirect3DDevice9::CreateTexture方法或以下Direct3D扩展（D3DX）函数。

D3DXCreateTexture

D3DXCreateTextureFromFile

D3DXCreateTextureFromFileEx

D3DXCreateTextureFromFileInMemory

D3DXCreateTextureFromFileInMemoryEx
D3DXCreateTextureFromResource
D3DXCreateTextureFromResourceEx
以下示例代码调用 D3DXCreateTextureFromFile 从文件 Tiger.bmp 中载入一张纹理。

```
// 以下示例代码假设 D3dDevice 为指向 IDirect3DDevice9 接口的有效指针。

LPDIRECT3DTEXTURE9 pTexture;

D3DXCreateTextureFromFile( d3dDevice, "tiger.bmp", &pTexture);
```

D3DXCreateTextureFromFile的第一个参数为指向IDirect3DDevice9接口的指针。第二个参数为文件名，告诉Direct3D从哪个文件载入纹理。第三个参数为指向IDirect3DTexture9 接口的指针的地址，表示创建得到的纹理对象。

用纹理资源进行渲染

Direct3D通过纹理层的概念支持多重纹理混合，每个纹理层包含一张纹理以及可以在这张纹理上执行的操作。纹理层中的纹理组成了一个当前纹理的集合。更多信息请参阅纹理混合。每张纹理的状态被封装在对应的纹理层中。

C++应用程序必须调用IDirect3DDevice9::SetTextureStageState方法设置每张纹理的状态。第一个参数为从 0 到 7 的纹理层索引值，第二个参数为D3DTEXTURESTAGESTATETYPE枚举类型值，最后一个参数为要设置的纹理层状态值。

在使用纹理接口指针进行渲染时，应用程序最多可以把八张纹理混合。应用程序通过调用IDirect3DDevice9::SetTexture方法设置当前纹理。Direct3D会先把所有当前纹理混合，然后再贴到正在渲染的图元的表面。

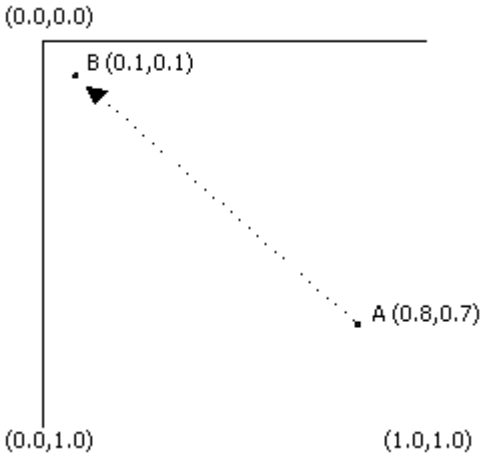
注意因为 IDirect3DDevice9::SetTexture 方法会增加正在设置的纹理表面的参考计数（reference count），所以当不再需要该纹理时，应用程序应该把相应纹理层的纹理设置为 NULL。如果应用程序不这样做，那么表面就不会被释放，并造成内存泄漏。

应用程序可以调用IDirect3DDevice9::SetRenderState方法设置当前纹理的纹理环绕状态，只需把第一个参数设为从D3DRS_WRAP0 到D3DRS_WRAP7 的值，并把第二个参数设为D3DWRAPCOORD_0，D3DWRAPCOORD1，D3DWRAPCOORD2，及D3DWRAPCOORD3 标志的组合，这样就可以启用在u, v, 或w方向上的纹理环绕。

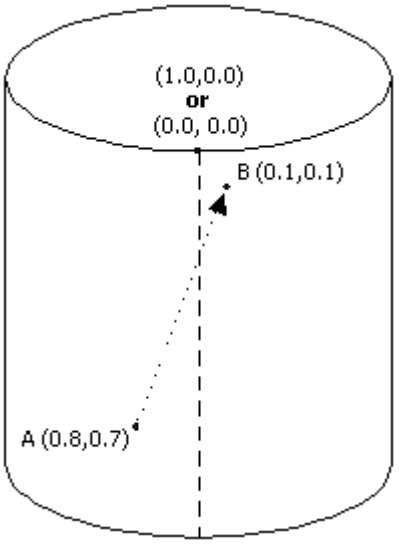
应用程序还可以设置纹理透视和纹理过滤状态。请参阅纹理过滤。

纹理环绕

简而言之，纹理环绕用来改变 Microsoft® Direct3D®用每个顶点的纹理坐标对贴有纹理的多边形进行光栅化的基本方法。在光栅化一个多边形时，为了确定多边形所覆盖的每个像素所需使用的 texel，系统要在每个多边形顶点的纹理坐标之间进行插值。一般来说，系统把纹理当做二维平面，为了在纹理中的点 A 和点 B 之间插值得到新的 texel，系统会取两点间的最短路径。如果点 A 表示的 u, v 坐标为(0.8, 0.1)，点 B 表示的 u, v 坐标为(0.1,0.1)，那么插值的路径会如下图所示。

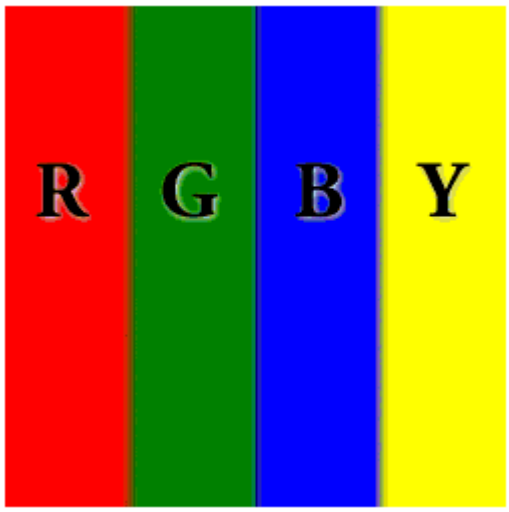


注意这张图中点 A 和点 B 间的最短路径大致穿越纹理的中央。启用 u 方向或 v 方向上的纹理坐标环绕会相应改变 Direct3D 在 u 方向或 v 方向上对纹理坐标间最短路径的理解。纹理环绕的定义使光栅化器在获取纹理坐标间的最短路径时，假设纹理坐标 0.0 和 1.0 是等价的。最后这一点是技巧所在：可以这样想象，在某一方向上启用纹理环绕会使系统把纹理当成在圆柱体表面环绕的纹理进行处理。例如，考虑下图。

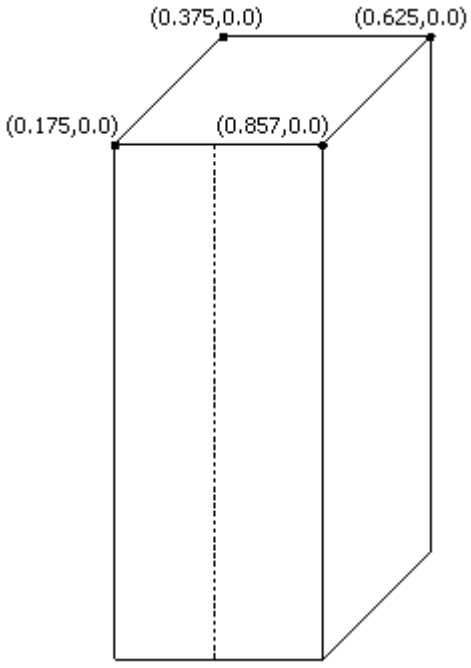


这幅图显示了在 u 方向上的环绕是怎样影响系统对纹理坐标间的插值的。在一张普通的，或没有环绕的纹理上使用与前例中相同的点，我们会发现点 A 和点 B 间的最短路径不再穿越纹理的中央，而是穿越纹理的边框，也就是纹理坐标 0.0 和 1.0 重合的地方。在 v 方向上的环绕与此类似，唯一的不同在于圆柱体是平躺的。同时在 u 方向和 v 方向上进行环绕比较复杂，在这种情况下，可以把纹理想象成是一个立体圆环。

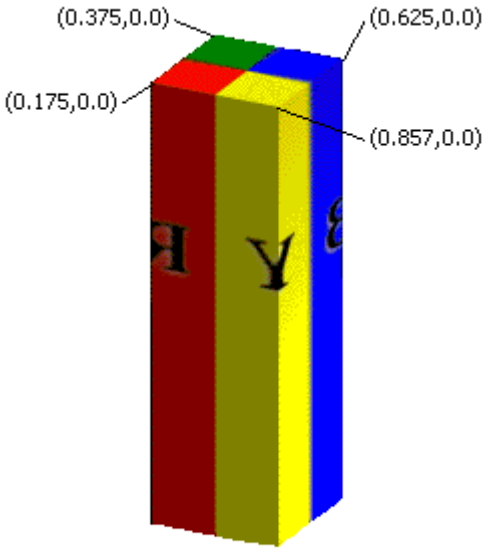
纹理环绕最通常的应用是在使用环境贴图时。通常，使用环境贴图的物体会显得比较反光，并反射出物体周围的场景。为了便于讨论，我们想象一个房间，房间有四面墙，每面墙上写着字母 R, G, B, Y，分别对应颜色红，绿，蓝，黄。这样一个简单的房间使用的环境贴图可能如下图所示。



设想房屋的屋顶由一根完全反光的柱子支撑，柱子有四面。要把环境贴图贴到柱子上很简单，但是要让柱子看起来像是反射墙上的字母和颜色就没有那么容易了。下图显示了用线框模式绘制的柱子，并在顶部的顶点处列出了相应的纹理坐标。环绕将发生在纹理的接缝处，在下图中用虚线表示。



如果在 u 方向上启用纹理环绕，那么柱子会正确地显示出环境贴图上的颜色和字母，并且在纹理前面的接缝处，光栅化器会假设 u 坐标 0.0 和 1.0 是等价的并正确地选择纹理坐标间的最短路径。贴上纹理后的柱子如下图所示。



如果没有启用纹理环绕，那么光栅化器就不会产生可信的反射图像。相反，柱子前面的部分会包含经过挤压的位于 u 坐标 0.175 到 0.875 之间的 texel，因为这些 texel 穿越纹理的中央。这样环绕效果被破坏了。

使用纹理环绕

要启用纹理环绕，应该调用IDirect3DDevice9::SetRenderState方法，如以下示例代码所示。

```
d3dDevice->SetRenderState(D3DRS_WRAP0, D3DWRAPCOORD_0);
```

IDirect3DDevice9::SetRenderState 的第一个参数是要设置的渲染状态，应该设为从 D3DRS_WRAP0 到 D3DRS_WRAP7 的枚举类型值，表示要设置哪一个纹理层的环绕状态。把第二个参数设为从 D3DWRAPCOORD_0 到 D3DWRAPCOORD_3 的标志可以在对应的方向上启用纹理环绕，也可以组合使用这些标志在多个方向上启用纹理环绕。如果应用程序忽略其中某个标志，那么在相应的方向上的纹理环绕就被禁用，要禁用某组纹理坐标在所有方向上的纹理环绕，只需把第二个参数设为 0。

不要把纹理环绕与名字相近的纹理寻址模式相混淆。纹理环绕在对纹理进行寻址之前进行。一定要保证用于纹理环绕的数据不包含[0.0, 1.0]范围外的纹理坐标，因为那样会导致不希望的结果。有关纹理寻址的更多信息，请参阅纹理寻址模式。

位移贴图的环绕

位移贴图由 tessellation 引擎解释，因为无法为 tessellation 引擎指定环绕模式，所以不能对位移贴图进行纹理环绕。应用程序可以用一组顶点，强制进行在任何方向上的环绕。应用程序也可以指定只进行简单的线性插值。

纹理混合

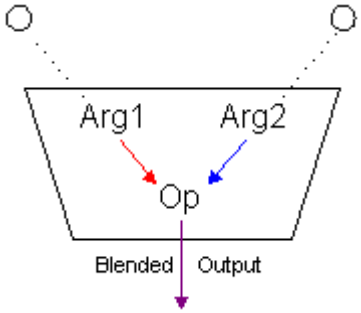
Microsoft® Direct3D®最多可以在一趟渲染过程中把八张纹理混合并贴到图元上。使用多重纹理可以极大地提高 Direct3D 应用程序的执行速度。应用程序可以用多重纹理混合在一趟渲染过程中产生纹理、影子、镜面反射光、漫反射光，以及其它特效。

要使用纹理混合，应用程序必须先检查硬件是否支持纹理混合。这个信息包含在D3DCAPAS9结构的TextureCaps成员中。有关如何查询硬件的纹理混合能力的细节，请参阅IDirect3DDevice9::GetDeviceCaps。

纹理层和纹理混合级联

通过使用纹理层，Direct3D 支持在一趟渲染过程中完成多重纹理混合。一个纹理层有两个输入，

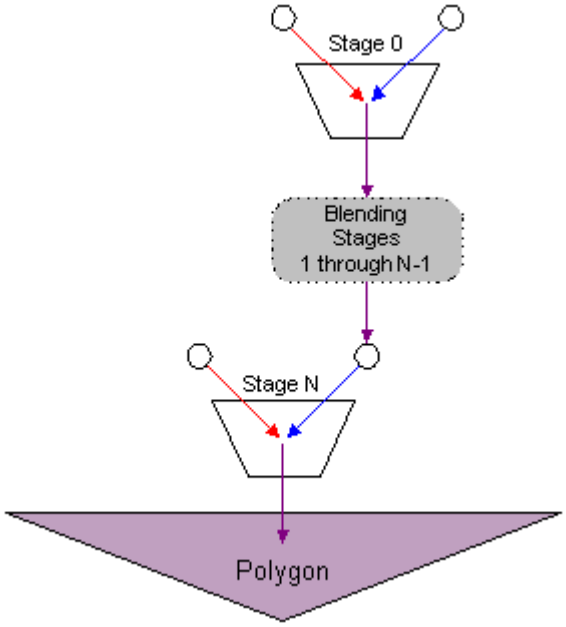
并对它们执行一个混合操作，然后把结果用于进一步的处理或用于光栅化。可以把纹理层想象为如下图所示。



如上图所示，纹理层用一个指定的操作符把两个输入混合。常用的操作符包括对输入参数的颜色或阿尔法的简单调制或相加，但实际上Direct3D支持几十种混合操作。纹理层的输入可以是与该层关联的纹理，迭代后的颜色或阿尔法（在进行高洛德着色的过程中迭代得到），指定的颜色或阿尔法，或前一个纹理层的结果。更多信息，请参阅纹理混合操作和输入。

注意Direct3D 区分颜色混合和阿尔法混合。应用程序分别设置要对颜色和阿尔法执行的混合操作及相应的输入，并且这些设置互不影响。

多重混合层的参数和操作的组合定义了一种简单的基于流程的混合语言。每一层的结果流入下一层，依次类推。这个概念，也就是混合的结果在层与层之间流动，并最终被用来对多边形进行光栅化操作，通常被称为*纹理混合级联*。下图显示了各独立的纹理层如何组成纹理混合级联。



设备中的每个纹理层都有一个从零开始的索引值。虽然应用程序应该总是检查设备的能力以确定当前设备支持几层纹理，但是 Direct3D 最多允许有八个混合层。第一层的索引值为 0，第二层的索引值为 1，依次类推，直到索引值 7。系统根据索引值的增长混合各纹理层。

最好只使用需要的那些混合层，默认情况下没有用到的混合层被禁用。因此，如果应用程序只使用前两个混合层，那么只需设置层 0 和层 1 的操作符和参数。系统会对这两层执行混合操作，并忽略其余被禁用的层。

有关性能的注意事项　如果应用程序在不同的情况下使用不同数量的纹理层——比如对一些物体使用四层纹理，而对其它物体只使用两层纹理——那么应用程序无需显式地禁止所有以前使用过的层。一种选择是对未曾用到的纹理层的第一层，禁用其颜色操作符，这样该纹理层及其后的纹理层将不会被用到。另一种选择是设置第一层纹理（层 0）的颜色操作符，禁用所有纹理贴图。

第三种选择是当纹理层的 D3DTSS_COLORARG1 等于 D3DTA_TEXTURE 时，只要该层的纹理指针为空，该层及其后的纹理层都不会被处理。

以下主题包含了更多信息。

纹理混合操作及参数

应用程序把混合层与当前纹理集合中的每张纹理相联系。Microsoft® Direct3D®按照顺序对每个混合层求值，从集合中的第一张纹理开始，至第八张结束。

Direct3D把当前纹理集合中每张纹理的信息应用于与之相联系的混合层。通过调用 IDirect3DDevice9::SetTextureStageState，应用程序可以控制使用纹理层中的哪些信息。应用程序可以分别设置对颜色和阿尔法通道的操作，每个操作有两个参数。用D3DTSS_COLOROP纹理层状态指定要对颜色通道执行的操作，用D3DTSS_ALPHAOP纹理层状态指定要对阿尔法通道执行的操作，这两个纹理层状态都是D3DTEXTUREOP枚举类型值。

纹理混合的参数使用D3DTEXTURESTAGESTATETYPE枚举类型的D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHAARG1 和D3DTSS_ALPHAARG2 成员表示。对应的参数值由D3DTA指定。

注意通过把某一层的颜色操作设置为 D3DTOP_DISABLE，应用程序可以禁用该纹理层及纹理混合级联中所有的后续层。禁用颜色操作会同时禁用阿尔法操作。当颜色操作被启用时，阿尔法操作无法被禁用。当颜色混合被启用时，把阿尔法操作设置为 D3DTOP_DISABLE 会导致不确定的结果。

要测定一个设备支持的纹理混合操作，请查询D3DCAPS9结构的TextureCaps成员。

设置当前纹理

Microsoft® Direct3D®维护着一个当前纹理列表，最多可以有八张。Direct3D 会把这些纹理混合到要渲染的图元上。只有作为纹理接口指针创建的纹理可以被用于当前纹理集合。

应用程序可以调用IDirect3DDevice9::SetTexture方法把纹理设置到当前纹理集合中。第一个参数必须是闭区间 0 到 7 之间的数字，第二个参数是纹理接口指针。

以下 C++示例代码显示了如何把一张纹理加入到当前纹理集合中。

```
// 本示例代码假设变量 lpd3dDev 为指向 IDirect3DDevice9 接口的有效指针，
// 且 pTexture 为指向 IDirect3DBaseTexture9 接口的有效指针。

// 设置第三层纹理
d3dDevice->SetTexture(2, pTexture);
```

注意软件设备不支持同时把一张纹理指定到一个以上的纹理层。

创建混合层

一个混合层是一个纹理操作及相应参数的集合，它定义了怎样混合纹理。C++应用程序在创建混合层时调用IDirect3DDevice9::SetTextureStageState方法。第一次调用指定要执行的操作，另外两次调用指定参数，Direct3D将用这两个参数执行指定的操作。以下示例代码描述了如何创建一个混合层。

```
// 本示例代码假设 lpD3DDev 为指向 IDirect3DDevice9 接口的有效指针。

// 设置要对第一个纹理进行的操作
```

```
d3dDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_ADD);

// 将参数 1 设置为纹理颜色
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);

// 将参数 2 设置为迭代后的漫反射色。
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
```

纹理中的texel数据包含颜色和阿尔法值。应用程序可以在一个混合层中分别定义颜色和阿尔法操作。颜色操作和阿尔法操作分别有自己的参数。更多细节，请参阅D3DTEXTURESTAGESTATETYPE。虽然下面的宏并不是 Microsoft® Direct3D®应用程序编程接口（API）的一部分，但是应用程序可以用它们简化创建纹理混合层所需的代码。

```
#define SetTextureColorStage( dev, i, arg1, op, arg2 )      \
   dev->SetTextureStageState( i, D3DTSS_COLOROP, op);       \
   dev->SetTextureStageState( i, D3DTSS_COLORARG1, arg1 ); \
   dev->SetTextureStageState( i, D3DTSS_COLORARG2, arg2 );

#define SetTextureAlphaStage( dev, i, arg1, op, arg2 )      \
   dev->SetTextureStageState( i, D3DTSS_ALPHAOP, op);       \
   dev->SetTextureStageState( i, D3DTSS_ALPHAARG1, arg1 );  \
   dev->SetTextureStageState( i  D3DTSS_ALPHAARG2, arg2 );
```

阿尔法纹理混合

Microsoft® Direct3D®在渲染一个图元时会根据图元的材质、或图元的顶点颜色，及光照信息为该图元产生一个颜色。更多细节，请参阅光照与材质。如果应用程序启用纹理混合，那么Direct3D必须把经过处理的多边形上的像素颜色与已经储存在帧缓存中的像素颜色进行混合。Direct3D使用以下公式计算图元的（渲染得到的）图像中每个像素的最终颜色。

FinalColor = TexelColor × SourceBlendFactor + PixelColor × DestBlendFactor

在这个公式中，*FinalColor*为输出到目标渲染表面的最终像素颜色。*TexelColor*为经过纹理过滤的输入颜色值，对应当前像素。有关Direct3D如何把texel映射到像素的细节，请参阅纹理过滤。*SourceBlendFactor*为一个经过计算的值，Direct3D用它计算输入颜色值在最终颜色中所占的百分比。*PixelColor*为当前存储在图元的图像中像素的当前颜色。*DestBlendFactor*为当前像素颜色在最终渲染得到的像素中所占的百分比。*SourceBlendFactor*和*DestBlendFactor*的值在闭区间 0.0 到 1.0 范围内。

正如我们从以上的公式中所看到的，如果 *SourceBlendFactor* 为 D3DBLEND_ONE 且 *DestBlendFactor* 为 D3DBLEND_ZERO，那么最终渲染得到的像素是不透明的。如果 *SourceBlendFactor* 为 D3DBLEND_ZERO 且 *DestBlendFactor* 为 D3DBLEND_ONE，那么得到的像素是完全透明的。如果应用程序把这些因子设置为为任何其它值，那么最终渲染得到的像素会具有一定的透明度。

经过纹理过滤后，每个像素的颜色值包含红、绿和蓝三种颜色值。默认情况下，Direct3D 使用 D3DBLEND_SRCALPHA 作为 *SourceBlendFactor*，使用 D3DBLEND_INVSRCALPHA 作为 *DestBlendFactor*。因此，通过设置纹理中的阿尔法值，应用程序可以控制处理后的像素的透明度。

C++应用程序可以用D3DRS_SRCBLEND和D3DRS_DESTBLEND渲染状态控制这些因子，只需调用IDirect3DDevice9::SetRenderState方法，把这两个渲染状态其中之一作为第一个参数传入。第

二个参数必须是D3DBLEND枚举类型成员。
多趟纹理混合
通过在多趟渲染的过程中将多个纹理贴到一个图元的表面，Microsoft® Direct3D®应用程序可以实现许多特效，这通常被称为多趟（multipass）纹理混合。多趟纹理混合的一个典型用途就是通过把几个不同纹理上的颜色混合，模拟复杂的光照和着色模型的效果。这种应用被称为光照贴图。更多信息，请参阅用纹理实现光照贴图。
注意一些设备可以在一趟渲染过程中将多张纹理贴到图元表面。细节请参阅纹理混合。
如果用户的硬件不支持多重纹理混合，应用程序可以使用多趟纹理混合以达到同样的视觉效果。但是，与使用多重纹理混合相比，应用程序将无法保持相同的帧速率。
要进行多趟纹理混合，C++应用程序应该执行以下操作。
调用IDirect3DDevice9::SetTexture方法给纹理层 0 设置一张纹理。
调用IDirect3DDevice9::SetTextureStageState方法设置相应的颜色和阿尔法混合操作及参数。默认的设定就很适合用于多趟纹理混合。
渲染场景中相应的物体。
将下一张纹理指定到纹理层 0。
根据需要设置D3DRS_SRCBLEND和D3DRS_DESTBLEND渲染状态以调整源和目的混合因子。系统根据这些参数把新的纹理和已经存在于渲染目标表面中的像素进行混合。
根据所需纹理的数量，重复步骤 3，4，5。
用纹理实现光照贴图
对于想要真实地渲染一个三维场景的应用程序来说，必须要考虑光源会对渲染得到的场景产生的效果。虽然诸如平面着色和高洛德着色之类的技术在这方面也是有用的工具，但它们可能无法满足应用程序的要求。Microsoft® Direct3D®支持多趟和多重纹理混合。与仅使用着色技术相比，这些能力使应用程序能够渲染更具真实感的场景。通过使用一张或多张光照贴图，应用程序可以把光影的范围映射到图元上。
光照贴图是包含三维场景中光照信息的一张纹理或一组纹理。应用程序可以把光照信息存放在光照贴图的阿尔法值中，颜色值中，或以上两者中。
如果应用程序用多趟纹理混合实现光照贴图，应用程序应该在第一趟渲染时把光照贴图贴到图元上，在第二次渲染时使用基本纹理。镜面反射光照贴图是个例外，在这种情况下，要先渲染基本纹理，再添加光照贴图。
多重纹理混合使应用程序能一次同时渲染光照贴图和基本纹理。如果用户的硬件支持多重纹理混合，应用程序应该在进行光照贴图时利用这一特性，这将极大地提高应用程序的性能。
如果使用光照贴图，Direct3D 应用程序可以在渲染图元时得到多种光照效果。应用程序不仅可以在场景中使用单色光和有色光，还可以添加诸如镜面反射高光和漫反射光之类的细节。
以下主题介绍了用 Direct3D 纹理混合实现光照贴图的信息。
单色光照贴图
有色光照贴图
镜面反射光照贴图
漫反射光照贴图
单色光照贴图
一些老的三维加速卡不支持使用目标像素的阿尔法值进行纹理混合，更多信息请参阅阿尔法纹理混合。一般来说这些加速卡也不支持多重纹理混合，如果应用程序在此类适配器上运行，那么可以用多趟纹理混合进行单色光照贴图。
要进行单色光照贴图，应用程序应该把光照信息存放在光照贴图的阿尔法数据中。应用程序使用 Microsoft® Direct3D®的纹理过滤功能把图元的图像中的每个像素映射到光照贴图中的相应

texel。应用程序应该把源混合因子设为相应 texel 的阿尔法值。
以下 C++示例代码描述了应用程序如何把一张纹理用作单色光照贴图。

```
// 本例假设 d3dDevice 为指向 IDirect3DDevice9 接口的有效指针，
// 且 lptexLightMap 为指向包含单色光照贴图数据的纹理的有效指针。

// 把光照贴图设置为当前纹理。
d3dDevice->SetTexture(0, lptexLightMap);

// 设置颜色操作。
d3dDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);

// 设置颜色操作的第一个参数。
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG1,
        D3DTA_TEXTURE | D3DTA_ALPHAREPLICATE);
```

因为不支持目标阿尔法混合的适配器一般来说也不支持多重纹理混合，这个示例把光照贴图设为第一张纹理，而这在所有三维加速卡上都是可用的。示例代码先设置纹理混合层的颜色操作，让纹理数据与图元已有的颜色进行混合，然后选择第一张纹理和图元已有的颜色作为输入数据。
有色光照贴图
如果应用程序使用有色光照贴图，那么通常会渲染得到更具真实感的三维场景。一张有色光照贴图使用 RGB 数据存放光照信息。
以下 C++示例代码显示了如何用 RGB 颜色数据进行光照贴图。

```
// 本例假设 d3dDevice 为指向 IDirect3DDevice9 接口的有效指针，
// 且 lptexLightMap 为指向包含单色光照贴图数据的纹理的有效指针。

// 把光照贴图设为第一张纹理。
d3dDevice->SetTexture(0, lptexLightMap);

d3dDevice->SetTextureStageState( 0,D3DTSS_COLOROP, D3DTOP_MODULATE );
d3dDevice->SetTextureStageState( 0,D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState( 0,D3DTSS_COLORARG2, D3DTA_DIFFUSE );
```

这个示例先把光照贴图设为第一张纹理，然后设置第一个混合层的状态，对输入数据进行调制（相乘），并把第一张纹理和图元的当前颜色用作调制操作的参数。
镜面反射光照贴图
在对发亮的物体——那些使用了高反射度材质的物体——进行光照计算时会产生镜面反射高光。在一些情况下，由光照模块产生的镜面反射高光不够精确，为了产生更吸引人的镜面反射高光，许多 Microsoft® Direct3D®应用程序会给图元使用镜面反射光照贴图。
要进行镜面反射光照贴图，只需把镜面反射光照贴图与图元的纹理相加，然后再和 RGB 光照贴图进行调制（与结果相乘）操作。
以下 C++示例代码描述了这个过程。

```
// 本例假设 d3dDevice 为指向 IDirect3DDevice9 接口的有效指针。
// lptexBaseTexture 为指向纹理的有效指针。
// lptexSpecLightMap 为指向包含 RGB 镜面反射光照贴图数据的纹理的有效指针。
// lptexLightMap 为指向包含 RGB 光照贴图数据的纹理的有效指针。
```

```
// 设置基本纹理。
d3dDevice->SetTexture(0, lptexBaseTexture );

// 设置要对基本纹理执行的操作及参数。
d3dDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE );
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// 设置镜面反射光照贴图。
d3dDevice->SetTexture(1, lptexSpecLightMap);

// 设置要对镜面反射光照贴图执行的操作及参数。
d3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADD );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );

// 设置 RGB 光照贴图。
d3dDevice->SetTexture(2, lptexLightMap);

// 设置要对 RGB 光照贴图执行的操作及参数。
d3dDevice->SetTextureStageState(2,D3DTSS_COLOROP, D3DTOP_MODULATE);
d3dDevice->SetTextureStageState(2,D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(2,D3DTSS_COLORARG2, D3DTA_CURRENT );
```

漫反射光照贴图

不光滑的表面在被光源照射时会显示漫反射光。漫反射光的亮度取决于表面到光源的距离及表面法向与光源的方向向量间的夹角。通过光照计算（译注：由光照模块执行）模拟漫反射光只能得到一般的效果。

应用程序可以用光照贴图模拟更为复杂的漫反射光照效果，只需在基本纹理的基础上再加一张漫反射光照贴图即可，如以下 C++示例代码所示。

```
// 本例假设 d3dDevice 为指向 IDirect3DDevice9 接口的有效指针。
// lptexBaseTexture 为指向纹理的有效指针。
// lptexLightMap 为指向包含 RGB 光照贴图数据的纹理的有效指针。

// 设置基本纹理。
d3dDevice->SetTexture(0,lptexBaseTexture );

// 设置要对基本纹理执行的操作及参数。
d3dDevice->SetTextureStageState(0,D3DTSS_COLOROP, D3DTOP_MODULATE );
d3dDevice->SetTextureStageState(0,D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(0,D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// 设置漫反射光照贴图。
d3dDevice->SetTexture(1,lptexDiffuseLightMap );
```

```
// 设置混合层 。
d3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
```

表面

表面是显存的线性表示，虽然可以存在于系统内存中，但更通常都存在于显卡的显存中。

IDirect3DSurface9接口包含了表面对象。

可以通过调用下列方法得到一个 IDirect3DSurface9 接口。

IDirect3DCubeTexture9::GetCubeMapSurface

IDirect3DDevice9::CreateDepthStencilSurface

IDirect3DDevice9::CreateRenderTarget

IDirect3DDevice9::GetBackBuffer

IDirect3DDevice9::GetDepthStencilSurface

IDirect3DDevice9::GetFrontBufferData

IDirect3DDevice9::GetRenderTarget

IDirect3DSwapChain9::GetBackBuffer

IDirect3DTexture9::GetSurfaceLevel

IDirect3DSurface9 接口允许应用程序通过IDirect3DDevice9::UpdateSurface方法间接访问内存。该方法允许应用程序从一个IDirect3DSurface9 接口复制一块矩形区域的像素到另一个IDirect3DSurface9 接口。表面接口也提供了直接访问显存的方法。例如，应用程序可以用IDirect3DSurface9::LockRect方法锁定显存的一块矩形区域。很重要的一点是在完成对锁定的矩形区域的操作后，要调用IDirect3DSurface9::UnlockRect方法。

表面格式用于描述如何解释表面内存中的像素数据。Microsoft® Direct3D®使用D3DSURFACE_DESC结构的D3DFORMAT成员描述表面格式。应用程序可以通过调用IDirect3DSurface9::GetDesc方法取得一个现有的表面的格式。

以下主题包含了更多信息。

宽度与Pitch的比较

翻转表面

页面翻转和后缓存

复制到表面

复制表面

直接访问表面内存

私有表面数据

Gamma控制

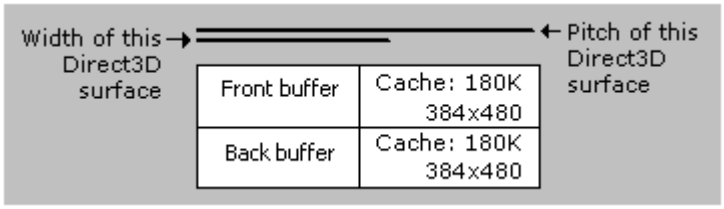
宽度与 Pitch 的比较

虽然对术语宽度和 pitch 的使用经常不很正式，但它们有着非常重要并且完全不同的意义。因此，开发人员应该理解它们的意义，以及如何解释 Microsoft® Direct3D®用于描述它们的值。

Direct3D使用D3DSURFACE_DESC结构保存描述表面的信息。其中，这个结构包含了表面的大小，以及这些大小在内存中是如何表示的。结构使用Height和Width成员描述表面的逻辑大小，这两个成员都以像素为单位。因此，对于一个 640x480 表面来说，无论它是 8 位表面或 24 位RGB表面，它的Height和Width值都是相同的。

当应用程序使用IDirect3DSurface9::LockRect方法锁定一个表面时，该方法会填写一个D3DLOCKED_RECT结构，这个结构包含了表面的pitch值及一个指向被锁定数据的指针。Pitch成员的值描述了表面在内存中的pitch，也被称为跨度。Pitch是两个内存地址间以字节为单位的距离，

两个内存地址分别表示一个位图某一行的起始地址以及下一行的起始地址。因为pitch是以字节为单位而非以像素为单位，所以一个 640x480x8 表面的pitch值与另一个大小相同但像素格式不同的表面的pitch值会大不相同。另外，pitch值有时反映出被Direct3D保留并用作高速缓存的字节数，因此简单地认为pitch就是宽度乘以每个像素所占的字节数是不保险的。如下图所示，对宽度和pitch之间的区别做一个直观的比较会更清楚。



在这张图中，前后缓存都是 640x480x8，高速缓存为 384x480x8。

当直接访问表面时要小心，不要访问为表面分配的内存以外的地方，更不要访问任何为高速缓存目的而保留的内存。另外，当应用程序锁定一个表面的一部分时，应用程序必须保持在锁定表面时指定的矩形区域内。不按这些指导方针行事将会导致无法预料的结果。当直接渲染到表面内存时，应该总是使用由 IDirect3DSurface9::LockRect 方法返回的 pitch 值。不要认为 pitch 仅取决于显示模式。如果应用程序在一些显示适配器上运行良好，但在另一些适配器上显示不正确的话，很可能就是 pitch 的问题。

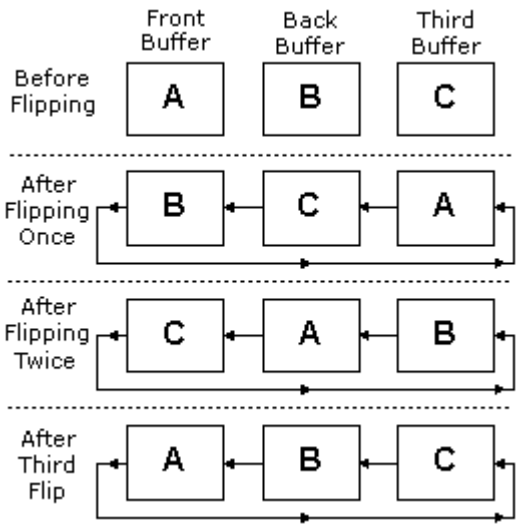
更多信息，请参阅直接访问表面内存。

翻转表面

Microsoft® Direct3D®应用程序一般通过这种方式显示动画序列，即：先在后缓存中生成动画的各帧，然后按顺序把这些帧显示出来。后缓存是属于交换链的一部分。一个交换链是一系列后缓存，这些后缓存会一个接一个被“翻转”到屏幕上。这种方法可以用来在内存中渲染一个场景，当渲染完成后随即把场景翻转到屏幕上。这避免了画面撕裂的现象，并能生成更为平滑的动画。

在Direct3D中创建的每个设备至少有一个交换链。当应用程序初始化第一个Direct3D设备时，应用程序要设置D3DPRESENT_PARAMETER结构的BackBufferCount成员，告诉Direct3D交换链需要包含的后缓存的数量。随后对IDirect3DDevice9::CreateDevice的调用会创建Direct3D设备及相应的交换链。

当应用程序使用IDirect3DDevice9::Present方法要求一个翻转操作时，指向前缓存的指针和指向后缓存的指针被交换。翻转是通过切换显示设备用来引用内存的指针完成的，而不是复制表面的内存。当翻转链包含一个前缓存和一个以上的后缓存时，指针的切换以循环的方式进行，如下图所示。



通过调用IDirect3DDevice9::CreateAdditionalSwapChain，应用程序可以为设备创建附加的交换链。应用程序可以为每个视区创建一个交换链并将每个交换链与某个特定窗口相关联。应用程序在每个交换链的后缓存中渲染图像，然后分别显示它们。

IDirect3DDevice9::CreateAdditionalSwapChain的两个参数分别为一个指向D3DPRESENT_PARAMETER结构的指针和一个指向IDirect3DSwapChain9接口的指针。应用程序可以使用IDirect3DSwapChain9::Present显示位于前缓存之后的那个后缓存的内容。注意一个设备只能有一个全屏交换链。

应用程序可以通过调用IDirect3DDevice9::GetBackBuffer或IDirect3DSwapChain9::GetBackBuffer方法取得对某个后缓存的访问权，这两个方法会返回一个指向IDirect3DSurface9接口的指针，代表被返回的后缓存表面。注意对这两个方法的调用会增加IDirect3DDevice9接口的内部引用计数，因此当完成对表面的操作后要记得调用IUnknown::Release，否则会导致内存泄漏。

记住，Direct3D 通过交换交换链内指向表面内存的指针翻转表面，而不是交换表面本身。这意味着应用程序总是在下次将被显示的那个后缓存上进行渲染。

很重要的一点是要注意由显卡驱动程序执行的“翻转操作”和一个用 D3DSWAPCHAIN_FLIP 标志创建的交换链执行的“present”操作间的区别。

按照惯例，术语“翻转”表示改变显卡用来产生输出信号的视频内存地址的范围，这样就导致原先隐藏着的后缓存的内容将被显示。在 Microsoft DirectX® 9.0 中，这个术语更经常地是被用来描述把任何用 D3DSWAPEFFECT_FLIP 标志创建的交换链中的后缓存显示出来。

而当交换链为全屏模式时，“present”操作几乎总是通过翻转操作实现，当交换链为窗口模式时，“present”操作必然通过复制操作实现。此外，显卡驱动程序可能会根据 D3DSWAPEFFECT_DISCARD 和 D3DSWAPEFFECT_COPY 标志，用翻转实现全屏交换链的 present 操作。以上讨论适用于常用的情况，也就是用 D3DSWAPEFFECT_FLIP 标志创建的全屏交换链。

有关窗口和全屏交换链的各种不同交换效果的讨论，请参阅D3DSWAPEFFECT。

页面翻转和后缓存

页面翻转是多媒体、动画和游戏软件中的关键，它和动画师用一叠纸产生动画的方法相似。在每张纸上，动画师对图片稍做改变，因此当动画师在页与页之间快速地翻动时，图片看起来就像是动了。

软件中的页面翻转与这个过程相似。Microsoft® Direct3D®通过交换链实现页面翻转功能，而交换链是设备的一个属性。开始时，应用程序先设置一系列 Direct3D 缓存，这些缓存会以和动画师相同的翻页方法翻转到屏幕。第一个缓存被称为前颜色缓存，它之后的缓存被称为后缓存。应

用程序可以先写入到后缓存，然后翻转颜色缓存，这样后缓存就显示在屏幕上。当系统显示图像时，应用程序又可以写入到后缓存。这个过程可以一直持续，这样应用程序就可以高效地生成活动的图像。

Direct3D 使建立一个页面翻转机制非常容易——从一个双缓存机制（一个前颜色缓存和一个后缓存）到使用额外后缓存的更为复杂的机制。

复制到表面

当使用IDirect3DDevice9::UpdateSurface时，要传入源表面中的一个矩形，或者用NULL表示整个表面，应用程序还需要传入目标表面中的一个点，源图像的左上角将被复制到这个位置。该方法不支持裁剪，除非源矩形和对应的目标矩形分别完全被包含在源和目标表面内，否则操作将会失败。这个方法不支持阿尔法混合、color key，及格式转换。注意目标表面和源表面不能是同一个表面。

其它有关UpdateSurface的使用限制，请参阅IDirect3DDevice9::UpdateSurface。

C++/C 应用程序可以用下列方法把图像复制到一个 Microsoft® Direct3D®表面。

D3DXLoadSurfaceFromFile

D3DXLoadSurfaceFromFileInMemory

D3DXLoadSurfaceFromMemory

D3DXLoadSurfaceFromResource

D3DXLoadSurfaceFromSurface

IDirect3DDevice9::UpdateSurface

相关主题

IDirect3DDevice9::StretchRect

复制表面

术语blit是“位块传输（bit block transfer）”的缩写，表示把数据块从内存中的一处传输到另一处的过程。作为在每帧——IDirect3DDevice9::Present方法背后的面向复制的机制——中移动大块矩形中的像素的主要机制，blitting设备驱动程序接口（DDI）仍在继续使用。blit操作中对纹理数据的传输由IDirect3DDevice9::UpdateTexture方法执行。在DirectX 9.0 中，纹理数据也可以用IDirect3DDevice9::UpdateSurface方法复制,该方法复制像素的一个矩形子集。注意DirectX 9.0 提供了Direct3D扩展（D3DX）函数，这使应用程序可以从文件载入纹理，进行颜色转换，及调整纹理的大小。有关更多可供使用的函数的信息，请参阅与纹理相关的函数。

相关主题

IDirect3DDevice9::StretchRect

直接访问表面内存

通过使用IDirect3DSurface9::LockRect方法，应用程序可以直接访问表面内存。在调用这个方法时，*pRect*参数为指向RECT结构的指针，描述要直接访问表面上的哪一部分。如果要锁定整个表面，只需把*pRect*设为NULL即可。同时，应用程序可以指定一个只覆盖表面的一部分的RECT。如果提供两个不相交迭的矩形，那么两个线程或进程可以同时锁定一个表面中的多个矩形。注意一个多重取样的（multisample）后缓存不能被锁定。

IDirect3DSurface9::LockRect方法会填写一个D3DLOCKED_RECT结构，该结构中包含了访问表面内存所需的全部信息。该结构包含了pitch信息，及一个指向被锁定的数据的指针。当应用程序完成对表面内存的访问后，应该调用IDirect3DSurface9::UnlockRect方法将表面解锁。

应用程序在锁定了一个表面后，可以直接对其中的内容进行操作。下面给出了一些提示，说明如何避免在使用直接渲染表面内存（directly rendering surface memory）时遇到的一些问题。

绝对不要认为pitch是一个常数，应该总是检查IDirect3DSurface9::LockRect方法返回的pitch

信息。Pitch可能会因为各种原因而不同，包括表面内存所在的位置，显卡的类型，甚至是Microsoft® Direct3D®驱动程序的版本。更多信息请参阅宽度与pitch的比较。
应用程序应该保证只对未锁定的表面进行复制操作，如果是锁定的表面，那么 Direct3D 的复制操作将会失败。
当一个表面被锁定时，应用程序应该限制对它进行的操作。
在复制数据时，应用程序应该总是保证和显存对齐。Microsoft Windows® 98 使用了一个页故障处理器，Vflatd.386，它为使用内存单元切换（bank-switched memory）的显卡实现一个虚拟的平面帧缓存。该处理器允许此类显示设备以线性方式把帧缓存提供给 Direct3D。当复制与显存不对齐的数据时，如果复制的数据跨越内存单元，那么可能会导致系统挂起。
如果表面隶属于D3DPOOL_DEFAULT内存池中的资源，那么它可能无法被锁定，除非它是动态纹理或是用IDirect3DDevice9::CreateOffscreenPlainSurface创建的表面。后缓存表面可以通过IDirect3DDevice9::GetBackBuffer和IDirect3DSwapChain::GetBackBuffer方法访问，只有在创建交换链时，D3DPRESENT_PARAMETER结构的Flags成员包含了D3DPRESENT_LOCKABLE_BACKBUFFER标志的情况下，它们才可以被锁定。
私有表面数据
应用程序可以在表面中存储任何类型的应用程序特有的数据。例如，在一个游戏中，一个表示地图的表面可以包含有关地形的数据。
一个表面可以有一个以上的私有数据缓存。每个缓存用一个 GUID 标识，该 GUID 由应用程序在把数据连接到表面时提供。
要存储私有表面数据，应该使用SetPrivateData，并传入源缓存，数据的大小，及应用程序为数据定义的GUID。或者，源数据也可以以COM对象的形式存在，这种情况下，应用程序只需传入对象的IUnknown接口指针，并设置D3DSPD_IUNKNOWNPOINTER标志。
SetPrivateData 会为数据分配一块内部缓存并把数据复制到其中。应用程序可以安全地释放源缓存或对象。当 FreePrivateData 被调用时，内部缓存或对接口的引用也会被释放。当释放一个表面时，系统会自动执行这个操作。
要得到表面的私有数据，应用程序必须先分配一块大小合适的缓存，然后调用 GetPrivateData 方法，并把原先赋给数据的 GUID 传入。应用程序有责任释放任何用于这块缓存的动态内存。如果数据是 COM 对象，那么该方法会取得 IUnknown 指针。
如果应用程序不知道应该分配多大的缓存，可以先把 pSizeOfData 设为零并调用 GetPrivateData，如果调用失败并返回 D3DERR_MOREDATA，那么该方法会在 pSizeOfData 中返回所需的字节数。
Gamma 控制
Gamma 控制允许应用程序改变系统如何显示表面的内容，同时不会影响表面本身的内容。可以认为这些控制是很简单的过滤器，在把表面数据显示在屏幕上之前，Microsoft® Direct3D®会对这些数据进行过滤。
Gamma 控制是交换链的一个属性。有了 Gamma 控制，动态改变如何把表面的红、绿和蓝色深映射到系统最终显示的实际色深就成为了可能。通过设置 Gamma level，应用程序可以使用户的屏幕闪现不同的颜色——当用户控制的角色被击中时为红色，当角色捡起了新的物品时为绿色，等等——同时不必为了达到相同的效果而把新的图像复制到帧缓存中去。
由于在 Microsoft DirectX® 9.0 中，交换链是设备的一个属性，因此每个 Direct3D 设备都至少有一条交换链（隐式交换链）。正因为 gamma ramp 是交换链的一个属性，所以当交换链处于窗口模式下时，gamma ramp 也可以使用。Gamma ramp 会立刻生效，不存在等待 VSYNC 的操作。
IDirect3DDevice9::SetGammaRamp和IDirect3DDevice9::GetGammaRamp方法允许应用程序在把表面中的像素送到数模转换器（DAC）进行显示之前对ramp levels进行操作，这会影响到表面中

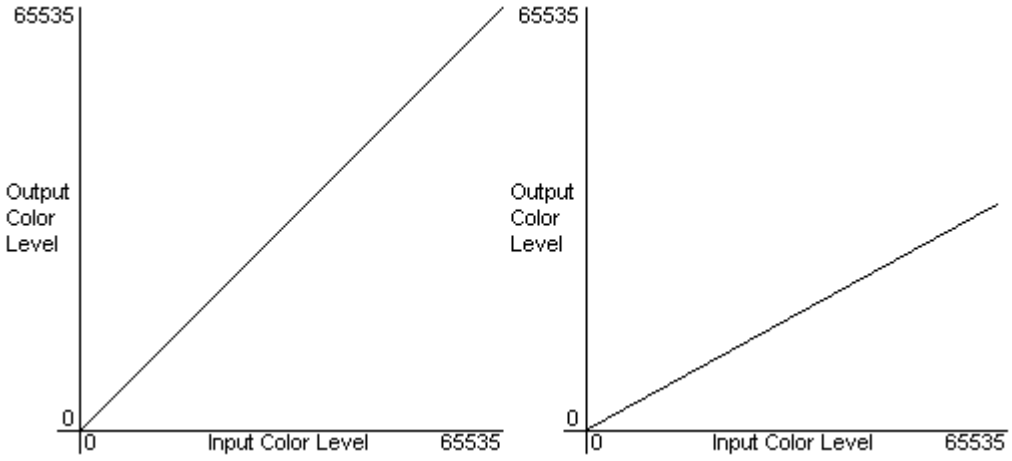
的像素的红、绿和蓝色分量。

Gamma Ramp Levels

在 Direct3D 中，术语 *gamma ramp* 指的是一个组数值，这些数值用于把帧缓存中所有像素的某一颜色分量——红、绿、蓝——的 level 映射到被 DAC 接收并用于显示的新的色深。

以下是 gamma ramp 的工作方式：Direct3D 从帧缓存中得到一个像素并分别计算每个红、绿和蓝颜色分量。每个分量由一个位于 0 到 65535 之间的值表示。Direct3D 用这个原始值作为索引，在一个有 256 个元素的数组（即 ramp）中查找，数组中每个元素包含一个值，用来替换原始值。Direct3D 对帧缓存中每个像素的每个颜色分量进行这个查找并替换的过程，从而改变了所有最终显示在屏幕上的像素的颜色。

通过画图很容易就能把 ramp 值直观地表示出来。下面两张图中左图显示了一个完全不改变颜色的 ramp，右图显示的 ramp 会给它所作用于的颜色分量加上负偏移。



左图中数组元素包含的值与它们的索引值相同——索引为 0 的元素的值为 0，索引为 255 的元素的值为 65535。这是默认的 ramp 类型，它不会在显示输入值之前改变它们。右图显示的 ramp 有较大变化，第一个元素的值为 0，最后一个元素的值为 32768，第一个和最后一个元素之间的元素的值在 0 到 32768 之间均匀分布。得到的效果就是使用这个 ramp 的颜色分量在显示器上会显得比较暗。Direct3D 并没有限制必须使用线性映射，如果需要，应用程序可以指定任意的映射方式。应用程序甚至可以把所有元素都设为零，以把某一颜色分量从显示器上完全消除。

设置及取得 Gamma Ramp Levels

Gamma ramp levels是Direct3D用于把帧缓存中的颜色分量映射到将被显示新的level的快速查找表。应用程序可以通过调用IDirect3DDevice9::SetGammaRamp和IDirect3DDevice9::GetGammaRamp方法设置和取得主表面的ramp levels。

IDirect3DDevice9::SetGammaRamp接收两个参数，第一个参数为D3DSGR_CALIBRATE或D3DSGR_NO_CALIBRATION，第二个参数*pRamp*为一指向D3DGAMMARAMP结构的指针。D3DGAMMARAMP结构包含了三个有 256 个元素的WORD数组，每个数组用来存放红、绿和蓝gamma ramp。

IDirect3DDevice9::GetGammaRamp接收一个参数，为一指向D3DGAMMARAMP的指针，当前的gamma ramp将被填写到该指针指向的结构中。

应用程序可以把 IDirect3DDevice9::SetGammaRamp 的第一个参数设为 DDSGR_CALIBRATE，这样在设置新的 gamma levels 时就会调用校正器。由于校正 gamma ramp 会增加一些开销，因此最好不要频繁地调用。无论显卡或显示器是何类型，设置一个校正过的 gamma ramp 会给用户提供完全一致的 gamma 值。

并非所有系统都支持gamma校正，要测定设备是否支持gamma校正，应该调用IDirect3DDevice9::GetDeviceCaps，然后检查由该方法返回的D3DCAPS9结构的Caps2 成员。如

果设置了D3DCAPS2_CANCALIBRATEGAMMA能力标志，那么设备就支持gamma校正。
在设置新的 ramp levels 时，谨记应用程序在数组中设置的 levels 只有当应用程序运行于全屏独占模式时才会被使用。一旦应用程序切换到正常模式，ramp levels 就会被忽略，并在应用程序恢复到全屏模式时重新生效。
即使设备在当前 presentation 模式（全屏或窗口）下不支持 gamma ramps，也不会返回错误码。应用程序可以检查 D3DCAPS9 结构的 Caps2 成员是否设置了 D3DCAPS2_FULLSCREENGAMMA 和 D3DCAPS2_CANCALIBRATEGAMMA 能力位，以确定设备的能力及是否安装了校正器。
Mipmap 的自动生成
Mipmap 的自动生成在创建纹理时利用了硬件过滤，这使得这项功能对应用程序而言是完全透明的。自动生成对 mipmap 渲染目标尤其有用，因为它们位于显存中，而这种情况下，软件过滤的效率很低。
要自动生成 mipmap，应该在创建纹理时设置D3DUSAGE_AUTOGENMIPMAP标志。后续的sublevel的生成对应用程序来说都是完全透明的。在某些情况下，某些硬件的自动mipmap生成可能会占用许多时间，这时应用程序可以适时地使用IDirect3DBaseTexture9::GenerateMipSubLevels，示意驱动程序生成相应数量的sublevels。
IDirect3DBaseTexture9::SetAutoGenFilterType用来控制自动生成时的过滤质量。改变过滤类型会导致mipmap的sublevels被标记为无效的并需要重新生成。
IDirect3DBaseTexture9::GetAutoGenFilterType用来取得当前的过滤类型。在创建纹理时使用的默认过滤类型是D3DTEXF_LINEAR。如果驱动程序不支持线性过滤，那么过滤类型将被设为D3DTEXF_POINT。
如果纹理不是用D3DUSAGE_AUTOGENMIPMAP标志创建的，那么调用这些方法不会产生任何效果，也不会返回任何错误。除了D3DTEXF_NONE外，驱动程序支持的所有可用于常规纹理过滤的过滤类型都可用于自动生成。对每种资源类型而言，驱动程序应该支持它在相应的纹理、立方体纹理和立体纹理过滤能力信息中提供的过滤类型。
当源纹理是自动生成的mipmap，而目标纹理不是时，IDirect3DDevice9::UpdateTexture是非法的，调用会失败。如果源纹理不是自动生成的mipmap且目标纹理是自动生成的mipmap，那么只有目标纹理中的最高的相匹配的那层被更新，该层的sublevels会重新被生成，而源纹理的sublevels将会被忽略。与此类似，如果源纹理和目标纹理都是自动生成的mipmap，那么只有目标纹理中的最高的相匹配的那层被更新，该层的sublevels会被重新生成，而源纹理的sublevels将被忽略。
在创建一个自动生成的 mipmap 时，*Levels* 参数必须被设置为零或一。
要检查设备对自动mipmap生成的支持，应该检查设备是否设置了D3DCAPS2_CANAUTOGENMIPMAP能力位。如果是，那么应该用D3DUSAGE_AUTOGENMIPMAP作为参数调用IDirect3D9::CheckDeviceFormat方法。如果返回值为D3D_OK，那么可以保证mipmap是自动生成的。如果返回值是D3DOK_NOAUTOGEN,这意味着对创建纹理的调用会成功,但不会生成任何mipmap。
要知道设备支持哪些过滤类型，应该检查D3DCAPS9结构的*TextureFilterCaps*、*CubeTextureFilterCaps*（译注：及*VolumeTextureFilterCaps*）成员所包含的能力位。
最后，要注意D3DUSAGE_AUTOGENMIPMAP只是一个提示，在创建纹理或调用IDirect3D9::CheckDeviceFormat的过程中指定这个标志不会在任何类型的设备驱动程序接口（DDI）上引起错误。
相关主题
D3DUSAGE
D3DCAPS2
D3DTEXTUREFILTERTYPE

自动纹理管理

纹理管理是确定在某一特定时刻需要用哪些纹理进行渲染，并确保这些纹理已经被载入显存的过程。同任何算法一样，不同的纹理管理机制在复杂度上会有所不同，但任何纹理管理机制都会涉及到以下一些关键任务。

跟踪可用显存的数量。

计算哪些纹理需要被用于渲染，而哪些不需要。

确定哪些现存（于显存中）的纹理资源可以重新载入其它纹理图像，以及哪些表面应该被销毁并被新的纹理资源代替。

为了保证纹理载入具有最佳的性能，Microsoft® Direct3D®内建了对纹理管理的支持。由 Direct3D 管理的纹理资源被称为*由系统管理的资源*。

纹理管理器用时间戳对纹理进行跟踪，时间戳记录了纹理最后被使用的时间。管理器然后用最近最少使用（least-recently-used）算法确定哪些纹理应该被移除。在准备把两张纹理从显存中移除时，纹理的优先级用来仲裁。如果两张纹理具有相同的优先级，那么最近最少使用的那张纹理会被移除。如果两张纹理具有相同的时间戳，那么优先级较低的那张纹理会先被移除。

应用程序可以在创建纹理表面时要求自动纹理管理。要在C++应用程序中得到一个由系统管理的纹理，应该调用IDirect3DDevice9::CreateTexture创建纹理资源，并把*Pool*参数指定为 D3DPOOL_MANAGED。Direct3D不允许应用程序指定要在何处创建纹理。在创建由系统管理的纹理时，应用程序不能使用D3DPOOL_DEFAULT或D3DPOOL_SYSTEMMEM标志。创建完由系统管理的纹理后，应用程序可以调用IDirect3DDevice9::SetTexture方法把纹理设到渲染设备的纹理级联中。

应用程序可以通过调用IDirect3DDevice9::SetPriority方法给由系统管理的纹理设置优先级。Direct3D 根据需要自动把纹理载入显存。系统可能会根据非本地视频内存的可用性或其它因素把由系统管理的纹理放在本地或非本地视频内存中作为高速缓存。系统不会把由系统管理的纹理所用的缓存的位置和大小告诉应用程序，而且对使用自动纹理管理而言也无需了解该信息。如果应用程序使用的纹理超过了显存所能容纳的数量，那么 Direct3D 会把旧的纹理从显存中移除以给新的纹理腾出空间。如果应用程序再次用到被移除的纹理，那么系统会用原始的系统内存纹理表面把纹理重新载入到显存的高速缓存中。虽然重新载入纹理是必须的，但它同时降低了应用程序的性能。

通过更新或锁定纹理资源，应用程序可以动态地修改纹理位于系统内存中的原件。当系统检测到一个无效表面时——在更新操作完成后，或当表面被解锁时——纹理管理器会自动地更新纹理位于显存中的复本。由此导致的性能下降与重新载入一个被移除的纹理相似。

当进入游戏中新的一关时，应用程序可能需要清空显存中所有由系统管理的纹理，此时应该调用 IDirect3DDevice9::EvictManagedResources。

有关资源管理的更多信息，请参阅管理资源。

压缩纹理资源

纹理贴图是画在三维物体上的数字化图像，用来添加可视细节。它们在光栅化时被贴到物体表面，这个过程会消耗大量的系统带宽和内存。为了减少纹理所消耗内存的数量，Microsoft® Direct3D®支持对纹理表面的压缩。一些 Direct3D 设备本身就支持压缩纹理表面。在这些设备上，只要应用程序创建了压缩表面并将数据载入其中，该表面就可以和其它任何纹理表面一样，在 Direct3D 中使用。在把压缩纹理贴到三维物体表面时，Direct3D 会进行解压。

存储效率和纹理压缩

所有纹理压缩的格式都是二的乘方。虽然这并不表示纹理一定要是方的，但确实表示 X 和 Y 都是二的乘方。例如，如果一个纹理原来是 512×128，那下一级 mipmap 应该是 256×64，依次类推，每一级都以两倍递减。到最低两级，纹理被过滤成 16×2 和 8×1，因为压缩块总是一个 4×4 的 texel 块，所以这里会浪费一些数据位。块中没有用到的部分被填满。虽然在最低几级会浪费

一些数据位，但总体的收获还是显著的。理论上最差的情况是，一个 2K×1 的纹理。这里，每一块只用到一行像素，其余的都没有用到。

在单个纹理内的混用不同格式

需要特别注意的是任何单个的纹理必须指明它的数据——每组 16 个 texel——是以 64 位还是以 128 位存储的。如果是 64 位块——也就是说，纹理用了 DXT1 格式，那么在同一纹理内以块为单位，混用不透明和一位阿尔法格式是可以的。换句话说，对每个由 16 个 texel 组成的块，对 color_00 和 color_1 两个无符号整数的比较是单独进行的。

一旦使用了 128 位块，整个纹理的阿尔法通道必须被指定为直接模式（DXT2 和 DXT3 格式）或插值模式（DXT4 和 DXT5 格式）。和颜色一样，一旦选择了插值模式，就可以以块为单位，混合使用八位或六位插值阿尔法。对 alpha_0 和 alpha_1 大小的比较仍然是以块为单位进行的。

对用于三维建模的纹理，Direct3D 提供了压缩表面的服务。本节提供了有关创建压缩纹理表面及操控表面中的数据的信息。

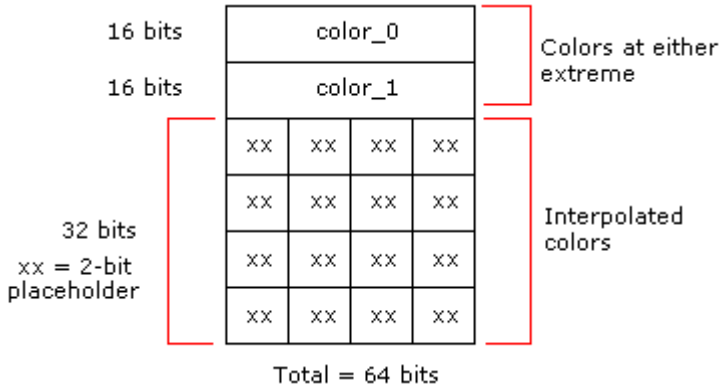
信息被分为以下主题。

不透明和一位阿尔法纹理
带阿尔法通道的纹理
压缩纹理格式
使用压缩纹理

不透明和一位阿尔法纹理

DXT1 纹理格式用于不透明的或只有一个透明色的纹理。

每个不透明或一位阿尔法块保存了两个 16 位颜色值（RGB 5:6:5 格式）和一个 4x4 的位图，位图中的每个像素占用 2 位。这样 16 个 texel 一共占用 64 位，或每个像素占用四位。在位图块中，每个 texel 占用 2 位，可以选择四个颜色，其中两个直接存储在经过编码的数据中，另两个则通过线性插值从存储的颜色值导出。下图显示了这种布局。



可以通过对存储在块中的两个 16 位颜色值进行比较来区分一位阿尔法格式和不透明格式。两个 16 位颜色值被当作无符号整数。如果第一个颜色值大于第二个，那么就暗示这一块只定义了不透明 texel。这意味着有四个颜色可以用来表示 texel。在四色编码中，有两个是导出的颜色，四个颜色值在 RGB 颜色空间中均匀分布。这种格式和 RGB 5:6:5 格式相似。否则（第一个颜色值小于等于第二个），就是一位阿尔法格式，一位阿尔法格式可以使用三个颜色，第四个颜色被保留，用来表示透明的 texel。

在三色编码中，有一个导出的颜色，第四个 2 位编码被保留，用来表示透明的 texel（阿尔法信息）。这种格式与 RGBA 5:5:5:1 格式相似，RGBA 5:5:5:1 格式的最后一位被用来编码阿尔法掩码。

以下示例代码描述了用来决定当前块是使用了三色编码还是四色编码的算法。

```
if (color_0 > color_1)
```

```
{
    // 四色编码块：导出另两个颜色。
    // 00 = color_0, 01 = color_1, 10 = color_2, 11 = color_3
    // 这些二位编码对应于存储在 64 位块中的二位位域。
    color_2 = (2 * color_0 + color_1 + 1) / 3;
    color_3 = (color_0 + 2 * color_1 + 1) / 3;
}
else
{
    // 三色编码：导出一个颜色。
    // 00 = color_0, 01 = color_1,  10 = color_2,
    // 11 = transparent.
    // 这些二位编码对应于存储在 64 位块中的二位位域。
     color_2 = (color_0 + color_1) / 2;
     color_3 = transparent;

}
```

应用程序在进行混合操作之前，最好把透明像素的 RGBA 成员设为零。

下面这些表显示了八字节块的内存布局，这里假设第一个索引值对应 y 坐标，第二个索引值对应 x 坐标。例如，Texel[1][2]指的是纹理贴图中位于(x,y) = (2,1)处的像素。

下表显示了八字节（64 位）块的内存布局。

字地址 16 位字

0　　Color_0

1　　Color_1

2　　位图数据 Word_0

3　　位图数据 Word_1

Color_0 和 Color_1 为位于两端的颜色，它们的布局如下所示。

位　　颜色

4:0（最低位）蓝色分量

10:5　　绿色分量

15:11

可编程流水线

Microsoft DirectX® 9.0 中的 Microsoft® Direct3D®使用了程序化的模块来指定顶点变换、光照流水线、像素/纹理混合流水线的行为。

把顶点着色器集成到几何流水线中

把像素着色器集成到图形流水线中

可编程数据流模型

顶点着色器

像素着色器

使用基于程序化的模块来指定硬件的行为有许多好处。

首先，程序化的模型使得用更为通用的语句来指定常用的操作成为可能。与可编程 API 不同，固定功能 API 必须随着所支持操作的增长，定义新的模式，标志，等等。此外，随着硬件能力的加强——更多的颜色，更多的纹理，更多的顶点数据流，等等——由输入数据导致操作符空间的增长也变得复杂。另一方面，可编程模型能够以更为直接的方式进行甚至简单的操作，例如把适当的颜色和纹理放到光照模块中合适的地方。开发人员无需搜索所有可能的模式，只要学习计算机体系结构并指定想要执行的算法即可。

例如，可编程流水线可以支持以下众所周知的特性。

基本几何变换

简单光照模型

表面蒙皮所需的顶点混合

顶点内插值（tweening）

纹理变换

纹理生成

环境贴图

其次，程序化的模型为开发新的操作提供了一个简单的机制。当前 API 不支持许多开发人员需要的操作。大多数情况下，这并不是由于硬件能力的限制造成的，相反是由于 API 的限制造成的。一般情况下，比起试图超越设计者的意图，强制扩充固定功能 API 以实现相同功能来说，用程序化的模型执行这些操作会更简单，速度也更快。

开发人员普遍希望实现的新特性包括以下这些：

Matrix Palette Skinning。用于每个 mesh 含 8 至 10 个骨架的角色动画。

Anisotropic Lighting(各向异性光照)。这种光照当前只能把纹理用作查找表实现。

Membrane Shaders。用于汽球、皮肤、等等(1/cos(eye DOT normal))的着色器。

Kubelka-Munk Shaders。对穿过表面的折射光线做了考虑的着色器。

Procedural Geometry。把 mesh 与程序生成的几何体（球体）合成，以模拟皮肤下肌肉的运动。

Displacement Mapping(位移贴图)。用波形样式或可以拼接/重复的山脊对 mesh 进行修改。

第三，程序化的模型提供了可扩充性和可发展性。硬件能力正在高速地发展，程序化的表示可以适应 API，因为它们很容易扩充。通过以下手段，可以很容易地不断地添加新的特性和能力。

增加新的指令

增加新的数据输入

给流水线的固定功能部分或可编程部分增加新的能力

在对复杂事物的描述方法中，代码具有最好的扩充性。此外，对于添加到可编程着色器中的新特性，Direct3D 内部所需改动的代码量也非常小。

第四，程序化的模型更容易被掌握。比起硬件，软件开发人员更理解编程。一个真正能够满足软件开发人员需要的 API 应该把硬件能力映射到代码的形式。

第五，程序化的模型跟随了具有照片真实感的渲染领域所经历的足迹。在具有照片真实感的高端渲染领域，使用可编程着色器的传统已经有许多年了。一般来说，这个领域不受性能的影响，因此对渲染技术而言，可编程着色器代表了无妥协的最终目标。

最后，程序化模型允许直接映射到硬件。当前大多数三维硬件，至少在顶点处理阶段，实际应该是可编程的。API 提供的可编程性使用应用程序的指令可以直接映射到这部分硬件。这使用开发人员可以根据需要管理硬件资源。要用有限数量的寄存器和可以运行的指令，去实现一个能启用所有特性并且特性之间不相互影响的固定功能流水线是很难的。如果开发人员打开了太多特性，而这些特性需要共享同一资源，那么这些特性可能都会以意想不到的方式停止运作。通过让应用程序开发人员与硬件直接对话，从而使此类限制对 API 变得透明，可编程 API 消除了这个问题，这也遵循了 DirectX 的传统。

把顶点着色器集成到几何流水线中

可编程顶点着色器在操作状态时会取代 Microsoft® Direct3D®的几何流水线中的变换和光照模块。实际上，有关变换和光照的状态都被忽略。但是，如果禁用顶点着色器并重新使用固定功能处理，那么所有当前状态设置会起作用。

任何对 high-order 图元的 tessellation 操作必须在顶点着色器执行之前完成。对那些在着色器处理后执行表面 tessellation 的硬件实现来说，必须采用某种方式使之对应用程序不可见。因为在着色器之前一般来说没有提供语义信息，所以系统使用了一个特殊的 token 来确定输入流中的哪个成员表示基位置，所有其它成员都相对于该成员进行插值。Direct3D 不支持无法插值的数据通道。

在输出时，顶点着色器必须产生齐次裁剪空间中的顶点位置。其它可以产生的数据包括纹理坐标，颜色，雾因子等等。

标准图形流水线会处理着色器输出的顶点，包括以下操作。

图元组装

根据视棱锥和用户裁剪平面进行裁剪

齐次除法

视区缩放

背向面和视区剔除

设置三角形

光栅化

Microsoft DirectX® 9.0 的顶点着色器和固定功能流水线的裁剪空间是相同的。更多细节，请参阅裁剪体。

可编程几何流水线是 Direct3D 应用程序编程接口（API）中的一种模式。当启用时，它会取代顶点流水线。当禁用时，API 就切换回固定功能顶点处理。顶点着色器的执行不会影响 Direct3D 的内部状态，同样着色器也不能使用 Direct3D 的任何状态。

应该用IDirect3DDevice9::CreateVertexShader创建一个顶点着色器，并在进行绘制之前调用IDirect3DDevice9::SetVertexShader设置可编程着色器。

把像素着色器集成到图形流水线中

像素处理由像素着色器对每个像素单独执行。像素着色器可以单独工作，也可以和顶点着色器及数据流协同工作。开发人员不能凭空对像素着色器进行编程，它们要依赖于上游的数据成员。

下面是像素流水线中的操作顺序：

设置三角形

像素着色器（取代固定格式的多重纹理）

对颜色、纹理坐标等等进行迭代，

对纹理进行取样

对纹理/颜色进行混合

雾混合

阿尔法、模板、深度测试

帧缓存混合

像素着色器的输入来自顶点着色器的输出。寄存器 v0 和 v1 包含了顶点颜色，它们来自顶点着色器的输出寄存器 oD0 和 oD1。颜色层中的纹理由诸如 tex t0 之类的像素着色器指令引用，系统会根据顶点着色器的输出寄存器中对应的纹理坐标（如 oT0）对纹理进行取样。像素着色器使用颜色和阿尔法混合指令以及纹理寻址指令对这些输入进行操控并计算出结果。像素着色器计算得到的结果是寄存器 r0 的内容或输出的像素颜色。着色器完成处理后会把处理结果送到雾处理阶段和渲染目标混合器做进一步的处理。顶点着色器的输出提供了像素着色器的输入。

ps_3_0 着色器模型和 ps_1_X/ps_2_0 有些不同的概念。
在 3_0 版像素着色器中，最终的雾混合应该由像素着色器执行。因此，像素流水线中的雾混合阶段被禁用，像素着色器的输出被送到像素流水线中的阿尔法/模板/深度测试阶段。
3_0 版本的顶点着色器不再支持颜色（oDn）和纹理（oTn）寄存器。现在使用的是输出（on）寄存器，它们的含义由顶点着色器中的声明指令定义。更多有关vs_3_0 中的寄存器和声明的信息，请参阅Registers - vs 3 0和dcl usage。
可编程数据流模型
本节讲述可用于可编程数据流模型的着色器。
顶点颜色着色器
单纹理着色器
多重纹理着色器
数据流的使用
Microsoft® DirectX® 8.0 引入了数据流的概念，用来把数据绑定到着色器使用的输入寄存器。一个数据流是一个成员数据的数组，每个成员由一个或多个元素构成，这些元素代表单个实体，如位置、法向、颜色等等。数据流使图形芯片能并行地从多个顶点缓存执行直接内存访问（DMA）操作，同时也降低了多重纹理的开销。可以这样理解数据流：
一个顶点由 n 个数据流组成。
一个数据流由 m 个元素组成。
一个元素是[位置、颜色、法向、纹理坐标]。
IDirect3DDevice9::SetStreamSource方法把一个顶点缓存绑定到一个设备数据流，这样就在顶点数据和一个顶点数据流端口之间建立了联系，有多个数据流端口用来给图元处理函数输入数据。对数据流中的数据的真正引用只有在调用诸如IDirect3DDevice9::DrawPrimitive之类的绘制方法时才发生。
从输入顶点元素到可编程顶点着色器使用的顶点输入寄存器的映射是在着色器声明中定义的，但是输入顶点元素并没有专门的语义来描述它们的使用。对输入顶点元素的解释通过着色器指令进行编程。顶点着色器函数由一个指令数组定义，这些指令会应用于每个顶点。顶点输出寄存器用着色器函数中的指令显式地写入。
本节的讨论较少关注从元素到寄存器的语义映射，而更侧重于“为什么要使用数据流？”和“数据流可以解决什么问题？”这些问题。数据流的最大好处是消除了原来和多重纹理有关的顶点数据的开销。在引入数据流之前，为了处理单纹理和多重纹理的情况，用户要么复制两份顶点数据，每份顶点数据中都没有用不到的数据；要么在一份顶点数据中包含所有的数据元素，但其中一部分数据除了多重纹理的情况以外不会被用到。
这里是一个使用两份顶点数据的示例，一份用于单纹理，另一份用于多重纹理。

```
struct CUSTOMVERTEX_TEX1
{
    FLOAT x, y, z;      // 未经变换的顶点位置
    DWORD diffColor;    // 顶点的漫反射色
    DWORD specColor;    // 顶点的镜面反射色
    float tu_1, tv_1;   // 单纹理的纹理坐标
};

struct CUSTOMVERTEX_TEX2
{
    FLOAT x, y, z;      // 未经变换的顶点位置
```

```
    DWORD diffColor;    // 顶点的漫反射色
    DWORD specColor;    // 顶点的镜面反射色
    float tu_2, tv_2;   // 多重纹理的纹理坐标
};
```
另一种方法是在一个顶点元素中包含全部两组纹理坐标。
```
struct CUSTOMVERTEX_TEX2
{
    FLOAT x, y, z;      // 未经变换的顶点位置
    DWORD diffColor;    // 顶点的漫反射色
    DWORD specColor;    // 顶点的镜面反射色
    float tu_1, tv_1;   // 单纹理的纹理坐标
    float tu_2, tv_2;   // 多重纹理的纹理坐标
};
```
如果使用这份顶点数据，那么只要在内存中保存一份顶点和颜色数据，代价是在渲染过程中保存了全部两组纹理坐标，甚至在单纹理的情况下也是如此。
现在这其中的权衡已经很清楚了，数据流为这种左右为难的情况提供了一种极好的解决方案。这里提供了一套顶点定义，用来支持三个数据流：一个数据流包含位置和颜色，一个数据流包含第一组纹理坐标，另一个数据流包含第二组纹理坐标。
```
// 多数据流顶点
// 数据流 0, 位置, 漫反射色, 镜面反射色
struct POSCOLORVERTEX
{
    FLOAT x, y, z;
    DWORD diffColor, specColor;
};
#define D3DFVF_POSCOLORVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_SPECULAR)

// 数据流 1, 纹理坐标组 0
struct TEXC0VERTEX
{
    FLOAT tu1, tv1;
};
#define D3DFVF_TEXC0VERTEX (D3DFVF_TEX1)

// 数据流 2, 纹理坐标组 1
struct TEXC1VERTEX
{
    FLOAT tu2, tv2;
};
#define D3DFVF_TEXC1VERTEX (D3DFVF_TEX0)
```
顶点定义为：
```
// 多重纹理 - 多重数据流

D3DVERTEXELEMENT9 dwDecl3[] =
```

```
{
    { 0, 0,  D3DDECLTYPE_FLOAT3,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
    { 0, 12, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,  0 },
    { 0, 28, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,  1 },
    { 1, 0, D3DDECLTYPE_FLOAT2,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0 },
    { 2, 0, D3DDECLTYPE_FLOAT2,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0 },
    D3DDECL_END()
};
```

现在创建顶点声明对象，如下所示：

```
LPDIRECT3DVERTEXDECLARATION9 m_pVertexDeclaration;
g_d3dDevice->CreateVertexDeclaration( dwDecl3, m_pVertexDeclaration );
```

组合的示例

一个数据流，只使用漫反射色

只用漫反射色渲染的顶点声明和数据流设置看起来会如下所示：

```
D3DVERTEXELEMENT9 dwDecl3[] =
{
    { 0, 0,  D3DDECLTYPE_FLOAT3,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
    { 0, 12, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,  0 },
    { 0, 28, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,  1 },
    D3DDECL_END()
};

m_pd3dDevice->SetStreamSource( 0, m_pVBVertexShader0, 0, sizeof(CUSTOMVERTEX) );
m_pd3dDevice->SetStreamSource( 1, NULL, 0, 0);
m_pd3dDevice->SetStreamSource( 2, NULL, 0, 0);
```

两个数据流，使用颜色和纹理

使用单纹理进行渲染的顶点声明和数据流设置看起来会如下所示：

```
D3DVERTEXELEMENT9 dwDecl3[] =
{
    { 0, 0,  D3DDECLTYPE_FLOAT3,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
    { 0, 12, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,  0 },
    { 0, 28, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,  1 },
    { 1, 0, D3DDECLTYPE_FLOAT2,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0 },
    D3DDECL_END()
};

m_pd3dDevice->SetStreamSource( 0, m_pVBPosColor, 0, sizeof(POSCOLORVERTEX) );
m_pd3dDevice->SetStreamSource( 1, m_pVBTexC0, 0, sizeof(TEXC0VERTEX) );
m_pd3dDevice->SetStreamSource( 2, NULL, 0, 0);
```

三个数据流，使用颜色和两张纹理

使用两张纹理进行多重纹理渲染的顶点声明和数据流设置看起来会如下所示。

```
D3DVERTEXELEMENT9 dwDecl3[] =
{
    { 0, 0,  D3DDECLTYPE_FLOAT3,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
```

```
    { 0, 12, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,  0 },
    { 0, 28, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,  1 },
    { 1, 0, D3DDECLTYPE_FLOAT2,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0 },
    { 2, 0, D3DDECLTYPE_FLOAT2,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0 },
    D3DDECL_END()
};

m_pd3dDevice->SetStreamSource( 0, m_pVBPosColor, 0, sizeof(POSCOLORVERTEX) );
m_pd3dDevice->SetStreamSource( 1, m_pVBTexC0, 0, sizeof(TEXC0VERTEX) );
m_pd3dDevice->SetStreamSource( 2, m_pVBTexC1, 0, sizeof(TEXC1VERTEX) );
```

以上所有三种情况，都可以调用以下IDirect3DDevice9::DrawPrimitive。

```
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, NUM_TRIS );
```

这个例子显示了数据流在解决重复的数据/冗余数据在总线上的传输（也就是说，带宽的浪费）问题上的灵活性。

顶点颜色着色器

本主题显示了初始化和使用一个用到了位置和漫反射色的简单顶点着色器的必须步骤。

第一步是声明用来保存位置和颜色的结构，如以下示例代码所示。

```
struct XYZBuffer
{
    FLOAT x, y, z;
};

struct ColBuffer
{
    D3DCOLOR color;
};
```

下一步是创建顶点着色器声明，如以下示例代码所示。

```
D3DVERTEXELEMENT9 decl[] =
{
    { 0, 0, D3DDECLTYPE_FLOAT3,   D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
    { 1, 0, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,    0 },
    D3DDECL_END()
};
```

现在创建顶点声明对象：

```
LPDIRECT3DVERTEXDECLARATION9 m_pVertexDeclaration;
g_d3dDevice->CreateVertexDeclaration( decl, &m_pVertexDeclaration );
```

下一步是调用IDirect3DDevice9::CreateVertexShader方法创建顶点着色器。但首先，必须先对着色器进行汇编。

```
TCHAR               strVertexShaderPath[512];
LPD3DXBUFFER        pCode;
LPDIRECT3DVERTEXSHADER9  m_pVertexShader;

hr = DXUtil_FindMediaFileCb( strVertexShaderPath,
                      sizeof(strVertexShaderPath), _T("ShaderFile.vsh");
```

```
hr = D3DXAssembleShaderFromFile( strVertexShaderPath, NULL, NULL,
                      dwFlags, &pCode, NULL );

g_d3dDevice->CreateVertexShader( (DWORD*)pCode->GetBufferPointer(),
                      &m_pVertexShader );
```

以下示例代码显示了如何设置顶点着色器，设置数据流的源，然后渲染三角形表。

```
g_pd3dDevice->SetVertexDeclaration( m_pVertexDeclaration );
g_d3dDevice->SetVertexShader( m_pVertexShader );
g_d3dDevice->SetStreamSource( 0, xyzbuf, sizeof(XYZBuffer));
g_d3dDevice->SetStreamSource( 1, colbuf, sizeof(ColBuffer));
g_d3dDevice->SetIndices( pIB, 0 );
g_d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0, count / 3 );
```

单纹理着色器

本主题显示了初始化和使用一个用到了位置和一组纹理坐标的简单顶点着色器的必须步骤。

第一步是声明用来保存位置和纹理坐标的结构，如以下示例代码所示。

```
struct XYZBuffer
{
    float x, y, z;
};

struct TEX0Buffer
{
    float tu, tv;
};
```

下一步是创建顶点着色器声明，如以下示例代码所示。

```
D3DVERTEXELEMENT9 decl[] =
{
    { 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
    { 1, 0, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0 },
    D3DDECL_END()
};
```

下一步是调用IDirect3DDevice9::CreateVertexShader方法创建顶点着色器。但首先，必须先对着色器进行汇编。

```
TCHAR               strVertexShaderPath[512];
LPD3DXBUFFER        pCode;
LPDIRECT3DVERTEXSHADER9  m_pVertexShader;

hr = m_pd3dDevice->CreateVertexDeclaration( decl, &m_pVertexDeclaration );

hr = DXUtil_FindMediaFileCb( strVertexShaderPath,
                sizeof(strVertexShaderPath), _T("ShaderFile.vsh");
```

```
hr = D3DXAssembleShaderFromFile( strVertexShaderPath, NULL, NULL,
                        dwFlags, &pCode, NULL );

g_d3dDevice->CreateVertexShader( (DWORD*)pCode->GetBufferPointer(),
                                    &m_pVertexShader );
```

在创建完顶点缓存和顶点着色器后，就可以使用了。以下示例代码显示了如何设置顶点着色器，设置数据流的源，然后用新的顶点着色器绘制三角形表。

```
g_d3dDevice->SetVertexShader( m_pVertexShader );
g_d3dDevice->SetStreamSource( 0, xyzbuf, 0, sizeof(XYZBuffer));
g_d3dDevice->SetStreamSource( 1, tex0buf, 0, sizeof(TEX0Buffer));
g_d3dDevice->SetIndices( pIB, 0 );
g_d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0, count / 3 );
```

多重纹理纹理着色器

本主题显示了初始化和使用一个用到了位置和用于多重纹理的多组纹理坐标的简单顶点着色器的必须步骤。

第一步是声明用来保存位置和纹理坐标的结构，如以下示例代码所示。

```
struct XYZBuffer
{
    float x, y, z;
};

struct Tex0Buffer
{
    float tu, tv;
};

struct Tex1Buffer
{
    float tu2, tv2;
};
```

下一步是创建顶点着色器声明，如以下示例代码所示。

```
D3DVERTEXELEMENT9 decl[] =
{
    { 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
    { 1, 0, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0 },
    { 2, 0, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 1 },
    D3DDECL_END()
};
```

现在创建顶点声明对象：

```
LPDIRECT3DVERTEXDECLARATION9 m_pVertexDeclaration;
g_d3dDevice->CreateVertexDeclaration( decl, &m_pVertexDeclaration );
```

下一步是调用IDirect3DDevice9::CreateVertexShader方法创建顶点着色器。但首先，必须先对着色器进行汇编。

```
TCHAR                   strVertexShaderPath[512];
LPD3DXBUFFER            pCode;
LPDIRECT3DVERTEXSHADER9  m_pVertexShader;

hr = DXUtil_FindMediaFileCb( strVertexShaderPath,
                sizeof(strVertexShaderPath), _T("ShaderFile.vsh"));

hr = D3DXAssembleShaderFromFile( strVertexShaderPath, NULL, NULL,
                        dwFlags, &pCode, NULL );

g_d3dDevice->CreateVertexShader( (DWORD*)pCode->GetBufferPointer(),
                                    &m_pVertexShader );
```

在创建完顶点缓存和顶点着色器后，就可以使用了。以下示例代码显示了如何设置顶点着色器，设置数据流的源，然后用新的顶点着色器绘制三角形表。

```
g_pd3dDevice->SetVertexDeclaration( m_pVertexDeclaration );
g_d3dDevice->SetVertexShader( m_pVertexShader );
g_d3dDevice->SetStreamSource( 0, xyzbuf, 0, sizeof(XYZBuffer) );
g_d3dDevice->SetStreamSource( 1, tex0buf, 0, sizeof(Tex0Buffer) );
g_d3dDevice->SetStreamSource( 2, tex1buf, 0, sizeof(Tex1Buffer) );
g_d3dDevice->SetIndices( pIB, 0 );
g_d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0, count / 3 );
```

顶点着色器

在 Microsoft® DirectX® 8.x 之前，Microsoft® Direct3D®以固定功能流水线的方式运作，把三维几何体渲染到屏幕上的像素。用户设置流水线的属性，以控制 Direct3D 变换、光照和渲染像素的方式。固定功能顶点格式用来确定输入顶点的格式，在编译时定义。一旦定义，用户就无法在运行的时候控制流水线的改变。

创建顶点着色器 - 本节包含的代码实例，用一个顶点着色器将一固定颜色应用于物体的顶点。本示例包含了对所使用的方法的详细描述。

顶点颜色 - 又一示例，显示了更多的代码实例，用来加入纹理，并把顶点颜色和纹理混合。

纹理 - 又一示例，显示了更多的代码实例，用来加入纹理，并把顶点颜色和纹理混合。

带光照的纹理贴图 - 又一示例，显示了更多的代码实例，用来加入纹理，并把顶点颜色和纹理混合。

将顶点着色器与顶点声明分离 - 现在顶点着色器声明与顶点声明已经分开。本节包含了固定功能流水线和顶点着色器流水线的声明之间的映射关系的详细资料。

通过允许在运行的时候进行变换、光照和渲染的功能，可编程着色器将图形流水线带入了一个新的高度。着色器是在运行时定义的，但是当完成以后，用户可以改变要激活哪个着色器，并使用数据流动态地控制着色器。这在渲染像素的方法上，给了用户更高的灵活性。

顶点着色器文件包含顶点着色器指令。顶点着色器可以控制顶点颜色和把纹理应用于顶点的方式。也可以通过使用顶点着色器指令加入光照。着色器指令文件包含的是ASCII文本，因此它是可读的,并且看起来有点象汇编语言。顶点着色器在任何DrawPrimitive或DrawIndexedPrimitive之后被调用。用SetVertexShader指定新的着色器文件可以动态地切换着色器，也可以用数据流输入改变ASCII文本的着色器文件中的指令。Vertex_Shader_1_1中包含了着色器指令的完全列表。

指令集的变化非常快。为避免在使用指令时出现问题，请查阅硬件开发商的网站。或者，也可以使用高级着色器语言，这样就可以得到由Direct3D扩展（D3DX）编译得到的着色器指令。

创建顶点着色器

本示例创建一个顶点着色器，把一固定颜色应用于物体。本示例将会给出着色器文件的内容，以及为了在应用程序中使用该着色器而设置 Microsoft® Direct3D®流水线所需的代码。

如果读者已经知道如何构建并运行 Direct3D 示例，那么可以从本示例中复制代码并粘贴到已有的应用程序中。

本文讨论了以下几个步骤。

第 1 步：声明顶点数据

本示例使用由两个三角形构成的四边形。顶点数据包含(x,y,z)位置和漫反射色。顶点数据在一个全局顶点数组(g_Vertices)中声明。四个顶点以原点为中心。

```
// 声明顶点数据结构。
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
};

// 声明顶点位置和漫反射色。
CUSTOMVERTEX g_Vertices[]=
{
    //  x         y       z
    { -1.0f, -1.0f, 0.0f },
    { +1.0f, -1.0f, 0.0f },
    { +1.0f, +1.0f, 0.0f },
    { -1.0f, +1.0f, 0.0f },
};
```

第 2 步：设计着色器的功能

该着色器将一固定颜色应用于每个顶点。以下为着色器文件 VertexShader.vsh：

```
vs_1_1              // 版本指令
dcl_position v0     // 说明位置数据在寄存器 v0 中
m4x4 oPos, v0, c0   // 用视/投影矩阵变换顶点
mov oD0, c4         // 载入固定颜色
```

该文件包含了三条数据运算指令和一个寄存器定义。着色器文件的第一条指令必须是着色器版本声明。vs 指令用来声明顶点着色器的版本，此处为 1_1。

dcl_position 指令把寄存器 v0 定义为顶点位置数据的源。m4x4 指令用视/投影矩阵对物体的顶点进行变换。矩阵被载入并保存在常量寄存器 c0, c1, c2, c3（如下所示）中。Mov 指令把寄存器 c4 中的固定颜色复制到输出漫反射色寄存器 oD0 中。这导致输出顶点的颜色被改变。

第 3 步：检查对顶点着色器的支持

在使用顶点着色器之前，可以查询驱动程序对顶点着色器的支持。

```
D3DCAPS9 caps;
m_pd3dDevice->GetDeviceCaps(&caps);     // 在使用前初始化 m_pd3dDevice
if( caps.VertexShaderVersion < D3DVS_VERSION(1,1) )
    return E_FAIL;
```

在调用IDirect3DDevice9::GetDeviceCaps后，caps结构会返回硬件可用的能力。应该用D3DVS_VERSION宏测试硬件支持的版本号。如果版本号小于 1_1，那么该调用将会失败。这个方法的结果应该被应用程序用来控制是否使用顶点着色器。

第 4 步：声明顶点着色器数据

顶点声明用来描述顶点数据。

```
// 创建着色器声明。
D3DVERTEXELEMENT9 decl[] =
{
  { 0, 0,  D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
  D3DDECL_END()
};
```

这个声明说明了以下这些：数据来自数据流 0，从数据流起始位置起偏移量为 0 处的位置开始，声明位置数据为三个浮点数(D3DDECLTYPE_FLOAT3)，告诉 tessellator 复制顶点数据(D3DDECLMETHOD_DEFAULT)，定义数据的用途为顶点数据(D3DDECLUSAGE_POSITION)，并说明用途索引为 0。

D3DDECLEND()宏用来结束顶点声明。

第 5 步：创建着色器

下一步汇编并创建着色器。

```
LPDIRECT3DPIXELSHADER9 m_pVertexShader;
TCHAR        strShaderPath[512];
LPD3DXBUFFER pCode;                  // 包含经过汇编的着色器代码的缓存
LPD3DXBUFFER pErrorMsgs;             // 包含错误信息的缓存
DXUtil_FindMediaFileCb( strShaderPath, sizeof(strShaderPath),
                        _T("VertexShader.vsh") );

D3DXAssembleShaderFromFile( strPixelShaderPath, NULL, NULL, 0,
                            &pCode, &pErrorMsgs, NULL );
m_pd3dDevice->CreateVertexShader((DWORD*)pCode->GetBufferPointer(),
                                 &m_pVertexShader)
pCode->Release();
pErrorMsgs->Release()
```

在定位了着色器文件后，D3DXAssembleShaderFromFile读取并验证着色器指令。然后IDirect3DDevice9::CreateVertexShader对指令进行汇编并创建着色器。该方法会返回用于绘制时调用的着色器对象。

*CreateVertexShader*用于创建一个可编程着色器。

第 6 步：渲染输出像素

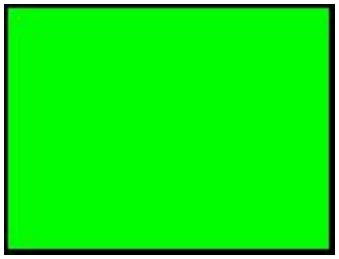
这里是一份代码示例，可以用在渲染循环中用顶点着色器来渲染物体。由于三维场景的变化，渲染循环不断更新顶点着色器常数，并调用IDirect3DDevice9::DrawPrimitive绘制输出顶点。

```
// 用视/投影矩阵更新顶点着色器常数。
```

```
D3DXMATRIX mat, matView, matProj;
D3DXMatrixMultiply( &mat, &matView, &matProj );
D3DXMatrixTranspose( &mat, &mat );
m_pd3dDevice->SetVertexShaderConstantF( 0, (float*)&mat, 4 );

// 声明并定义固定的顶点颜色。
float color[4] = {0,1.0,0,0};
m_pd3dDevice->SetVertexShaderConstantF( 4, (float*)&color, 1 );

// 关闭镜面反射光，因为顶点着色器没有输出镜面反射光。
m_pd3dDevice->SetRenderState( D3DRS_SPECULAR, FALSE );

// 渲染输出。
m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTEX) );
m_pd3dDevice->SetVertexShader( m_pVertexShader );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
```

视和投影矩阵包含了摄像机的位置和方向的数据。因为场景可能在渲染得到的帧之间变化，所以渲染循环中要得到最新的数据，并用这些数据更新着色器的常量寄存器

IDirect3DDevice9::DrawPrimitive照常使用IDirect3DDevice9::SetStreamSource提供的数据渲染输出数据。IDirect3DDevice9::SetVertexShader是用来告诉Direct3D要使用顶点着色器。顶点着色器的结果如下图，它显示了在平面物体上的固定颜色。



顶点颜色

本示例把顶点数据中的顶点颜色应用于物体。顶点数据包含了位置和漫反射色。这些数据如下所示：

```
struct CUSTOMVERTEX_POS_COLOR
{
    float       x, y, z;
    DWORD       diffuseColor;
};

// 创建包含位置和纹理坐标的顶点数据。
CUSTOMVERTEX_POS_COLOR g_Vertices[]=
{
    //  x      y     z      漫反射色
    { -1.0f, 0.25f, 0.0f, 0xffff0000,  }, //  右下 - 红
    {  0.0f, 0.25f, 0.0f, 0xff00ff00,  }, //  左下 - 绿
    {  0.0f, 1.25f, 0.0f, 0xff0000ff,  }, //  左上 - 蓝
```

```
    { -1.0f, 1.25f, 0.0f, 0xffffffff,  }, //  右上 - 白
};
```

顶点着色器声明也需要反映位置和颜色数据。

```
// 创建着色器声明。
D3DVERTEXELEMENT9 decl[] =
{
{ 0, 0,  D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
{ 0, 12, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR, 0 },
D3DDECL_END()
};
```

着色器得到变换矩阵的一种方法是通过常量寄存器，常量寄存器是通过调用 SetVertexShaderConstant 设置的。

```
D3DXMATRIX mat;
D3DXMatrixMultiply( &mat, &m_matView, &m_matProj );
D3DXMatrixTranspose( &mat, &mat );
hr = m_pd3dDevice->SetVertexShaderConstantF( 1, (float*)&mat, 4 );
if(FAILED(hr))
    return hr;
```

该声明声明了一个包含了位置和颜色数据的数据流。颜色数据被指定到寄存器 1。以下是着色器代码。

```
vs_1_1              ; 版本指令
m4x4 oPos, v0, c0   ; 用视/投影矩阵变换顶点
mov oD0, v1         ; 从寄存器 1 载入颜色并赋给漫反射色
```

这个着色器包含三条指令。第一条总是版本指令。第二条指令用来变换顶点。第三条指令用来把顶点寄存器中的颜色复制到输出漫反射色寄存器中。结果是输出顶点使用了顶点的颜色数据。得到的输出看起来如下所示：

纹理

本示例将一张纹理贴图应用于物体。

顶点数据包含了物体的位置和纹理坐标（uv）。这导致了顶点声明的改变。下面还显示了顶点数据。

```
struct CUSTOMVERTEX_POS_TEX1
{
    float       x, y, z;      // 物体位置
    float       tu1, tv1;     // 纹理坐标
};

CUSTOMVERTEX_POS_TEX1 g_Vertices[]=
```

```
{
    //  x       y      z       u1     v1
    { -0.75f, -0.5f, 0.0f, 0.0f, 0.0f },  //  右下 - 红
    {  0.25f, -0.5f, 0.0f, 1.0f, 0.0f },  //  左下 - 绿
    {  0.25f,  0.5f, 0.0f, 1.0f, -1.0f }, //  左上 - 蓝
    { -0.75f,  0.5f, 0.0f, 0.0f, -1.0f }, //  右上 - 白
};

D3DUtil_CreateTexture( m_pd3dDevice, TEXT("earth.bmp"),
                &m_pTexture0, D3DFMT_R5G6B5 );
```

必须先载入纹理的图像。在本例中，文件"earth.bmp"包含了地球的二维纹理贴图，并将被用来给物体着色。

顶点着色器声明需要反映出顶点位置和纹理坐标数据。

```
// 创建着色器声明。
D3DVERTEXELEMENT9 decl[] =
{
{ 0, 0,  D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
{ 0, 12, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0 },
D3DDECL_END()
};
```

该声明声明了一个包含顶点位置和纹理坐标的数据流。

渲染代码告诉 Microsoft® Direct3D®到何处去得到数据流和着色器，因为使用了纹理贴图，所以还要设置纹理层。

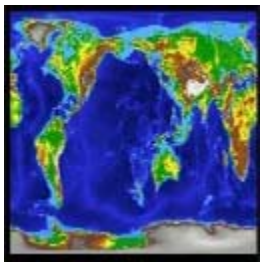
```
m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTEX_POS_TEX1) );
m_pd3dDevice->SetVertexShader( m_pVertexShader );

m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
m_pd3dDevice->SetTexture( 0, m_pTexture0 );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
m_pd3dDevice->SetTexture( 0, NULL );
```

因为使用了一个纹理，所以需要设置纹理层 0 的纹理层状态。上面调用的方法告诉 Direct3D 纹理的 texel 值用来给物体的顶点提供漫反射色。换句话说，二维纹理贴图的使用就象贴花纸一样。

这里是着色器代码。

```
vs_1_1                // 版本指令
dcl_position v0       // 声明位置寄存器
dcl_texcoord v8       // 声明纹理坐标寄存器
def c4, 1, 1, 1, 1    // 初始化常量
m4x4 oPos, v0, c0     // 用视/投影矩阵变换顶点
mov oD0, c4           // 把漫反射色赋给输出颜色寄存器
mov oT0, v8           // 把纹理的颜色赋给纹理寄存器
```

着色器文件包含的这些指令会产生一个贴上了纹理的物体，如下所示。



带光照的纹理贴图

本实例使用了一个顶点着色器，将纹理贴图和光照应用于场景。这里使用的物体是一个球体，示例代码把地球的纹理贴图应用于球体，并用漫反射光照来模拟昼夜。

Shader3 实例给贴上了纹理的物体添加了光照。关于如何载入纹理和设置纹理层状态的信息，请参阅 Shader3。

在示例框架中有关于框架的示例代码的详细说明。读者可以复制这里的示例代码并粘贴到示例框架中去，这样就可以很快得到一个能运行的示例。

顶点着色器的创建

为了包含顶点法向，需要修改 Shader3 实例中的顶点数据。要产生光照，物体必须有顶点法向。修改后的顶点数据的数据结构如下所示。

```
struct CUSTOMVERTEX_POS_NORM_COLOR1_TEX1
{
    float       x, y, z;        // 位置
    float       nx, ny, nz;     // 法向
    DWORD       color1;         // 漫反射色
    float       tu1, tv1;       // 纹理坐标
};
```

着色器声明定义了输入顶点寄存器以及和它们关联的数据。

```
// 创建着色器声明。
D3DVERTEXELEMENT9 decl[] =
{
{ 0, 0,  D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
{ 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL,   0 },
{ 0, 24, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,  0 },
{ 0, 28, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0 },
D3DDECL_END()
};
```

这里声明了一个数据流，其中包含了顶点位置，法向，漫反射色和纹理坐标。

下一步是创建着色器。可以用一个 ASCII 文本字符串创建着色器，也可以从包含相同指令的着色器文件中载入。本示例使用着色器文件。

```
// v7   用于光照的顶点漫反射色
// v8   纹理
// c4   视/投影矩阵
// c12 光的方向

vs_1_1                          // 版本指令
dcl_position v0                 // 声明寄存器数据
```

```
dcl_normal v4                // v0 为位置，v4 为法向
dcl_color0 v7                // v7 为漫反射色
dcl_texcoord0 v8             // v8 为纹理坐标
m4x4 oPos, v0, c4            // 用视/投影矩阵变换顶点
dp3  r0, v4, c12             // 执行世界空间中的光照计算 N dot L
mul  oD0, r0.x , v7          // 根据光强度和经插值的顶点漫反射色计算像素的最终颜色
mov  oT0.xy , v8             // 将纹理坐标得到到输出
```

第一条总是版本指令，最后一条指令把纹理数据复制到输出寄存器 oT0。写完了着色器指令后，就可以用来创建着色器。

```
LPDIRECT3DPIXELSHADER9 m_pVertexShader;
TCHAR        strShaderPath[512];
LPD3DXBUFFER pCode;                  // 包含汇编后的着色器代码的缓存
LPD3DXBUFFER pErrorMsgs;             // 包含错误信息的缓存
DXUtil_FindMediaFileCb( strShaderPath, sizeof(strShaderPath),
                        _T("VertexShader3.vsh") );

D3DXAssembleShaderFromFile( strPixelShaderPath, NULL, NULL, 0,
                            &pCode, &pErrorMsgs, NULL );
m_pd3dDevice->CreateVertexShader((DWORD*)pCode->GetBufferPointer(),
                                 &m_pVertexShader)
pCode->Release();
pErrorMsgs->Release();
```

在定位了文件后，Microsoft® Direct3D®会创建顶点着色器并返回着色器对象。本示例用了一个着色器文件，这样调用一个方法就可以创建着色器。另一种方法是创建一个包含着色器指令的 ASCII 文本字符串。

顶点着色器常量

可以在着色器外定义顶点着色器常量，如以下示例所示。此处，常量用来给着色器提供视/投影矩阵，漫反射光颜色 RGBA 和光的方向向量。

```
float constants[4] = {0, 0.5f, 1.0f, 2.0f};
m_pd3dDevice->SetVertexShaderConstantF( 0, (float*)&constants, 1 );

D3DXMATRIX mat;
D3DXMatrixMultiply( &mat, &m_matView, &m_matProj );
D3DXMatrixTranspose( &mat, &mat );
m_pd3dDevice->SetVertexShaderConstantF( 4, (float*)&mat, 4 );

float color[4] = {1,1,1,1};
m_pd3dDevice->SetVertexShaderConstantF( 8, (float*)&color, 1 );

float lightDir[4] = {-1,0,1,0}; // fatter slice
m_pd3dDevice->SetVertexShaderConstantF( 12, (float*)&lightDir, 1 );
```

也可以在着色器内部用def指令定义常量。

渲染

在写完着色器指令后，要将顶点数据与正确的顶点寄存器连接并初始化常量，然后渲染输出。渲

染代码告诉 Direct3D 到哪里去得到顶点缓存的数据流，并给 Direct3D 提供着色器的句柄。因为使用了纹理，所以还必须设置纹理层以告诉 Direct3D 如何使用纹理数据。

```
// 设置顶点缓存的数据源。
m_pd3dDevice->SetStreamSource(0, m_pVB,
sizeof(CUSTOMVERTEX_POS_NORM_COLOR1_TEX1));
// 设置着色器。
m_pd3dDevice->SetVertexShader( m_pVertexShader );

// 设置用到的纹理和纹理层状态。
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_MODULATE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
m_pd3dDevice->SetTexture( 0, m_pTexture0 );

// 绘制物体。
DWORD dwNumSphereVerts = 2 * m_dwNumSphereRings*(m_dwNumSphereSegments + 1);
m_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, dwNumSphereVerts - 2);
```

下面是输出的图像。

贴了纹理后，球体看起来像是地球。光照在球体表面产生了从亮到暗的变化。

将顶点着色器与顶点声明分离

在 Microsoft® DirectX® 9.0 中，顶点着色器和顶点声明不再是在创建顶点着色器（CreateVertexShader）的时候绑定在一起。对着色器的验证已经被分成两部分，一部分在顶点着色器创建时执行，另一部分在绘制（DrawPrimitive）时执行。

把DirectX 8.x的顶点声明映射到DirectX 9.0 的顶点声明

把FVF码映射到DirectX 9.0 的顶点声明

把DirectX 9.0 的顶点声明映射到DirectX 8.x的顶点声明

把DirectX 9.0 的顶点声明映射到FVF码

顶点着色器和顶点声明都由相应的对象表示。为了使之能与 DirectX 8.x 驱动程序一起工作，Direct3D 进行了一些高速缓存。在“绘制的时候”，运行库会检查是否存在一个组合着色器对象，该对象封装了当前声明和着色器。如果存在，那么运行库会将它送给驱动程序，反之，运行库会为当前着色器和声明的组合创建一个新的组合着色器对象。另外，为了解决 API 的可用性问题，9.0 版增加了一个与 SetVertexDeclaration 调用等价的 SetFVF 调用。这是一个有用的函数，当调用这个函数时，新的 FVF 会取代当前的顶点声明，反之亦然。如果驱动程序是 DirectX 8.0 之前的版本（NumStream 为 0），那么对于那些不能被转换成弹性顶点格式（FVF）的顶点声明，SetVertexDeclaration 可能会失败并返回错误码。

SetVertexDeclaraion 和 SetVertexShader 调用会被状态块记录。设置或取得着色器或声明会导致该对象的引用计数的增加。

FVF 码既可以用于固定功能流水线，又能用于可编程顶点流水线。FVF 要根据转换表，被转换为顶点着色器声明。在写顶点着色器函数中的 DCL 命令时，应该紧记这一点。

软件顶点处理支持 DirectX 9.0 级别的特性，也支持 tessellation，因此在用软件顶点处理进行绘制时，可以使用更多的声明。

对可编程顶点流水线而言：在绘制的时候，Microsoft® Direct3D®会在当前的顶点声明和当前的顶点着色器函数中查找相同的“用途 - 用途索引”组合。如果找到了这样的组合，那么着色器函数中 DCL 声明的寄存器会被用作顶点元素的目标寄存器。

如果 FVF 包含了 XYZRHW 位置类型，那么顶点被认为是经过变换和经过光照处理的，在用于可编程顶点流水线时，不会应用顶点着色器函数。

如果当前顶点声明中的一个顶点元素的用途无法在当前顶点着色器中找到，那么该顶点元素就被忽略。

在 DirectX 8.x 驱动程序（numstrems 不为零，但不支持数据流偏移量）上允许使用声明的一个子集，只能创建那些可被转换为 DirectX 8.x 风格的声明。由于这个原因，如果声明不能被转换，那么 CreateVertexDeclaration 调用可能会失败。对混合模式设备来说，此类失败会发生在 Draw*xxx* 的时候，因为只有这时候才能知道着色器是被用于硬件还是软件顶点处理。表中概括了这类转换。

在 DirectX 8.0 之前的驱动程序（numstreams 为零）上允许使用一个更小的子集，只能使用那些可被转换为 FVF 的声明。如果无法进行转换，那么 CreateVertexDeclaration 可能会失败。对混合模式设备来说，此类失败会发生在 Draw*xxx* 的时候，因为只有这时候才能知道着色器是被用于硬件还是软件顶点处理。表中概括了这类转换。只能使用数据流 0（显然可以从 MaxStreams 设备能力中看出）。

顶点元素的顺序应该和 FVF 码相对应，D3DDECLUSAGE_POSITION 和 D3DDECLUSAGE_NORMAL 的用途索引应该为零。使用混合顶点处理的设备时，如果要切换顶点处理模式，那么无需重置所有的输入顶点（和索引）数据流，顶点声明和顶点函数。不能用 NULL 作为 SetVertexDeclaration 的输入。在使用软件顶点处理时，可编程顶点流水线的着色器代码中用到的用途也应该存在于在绘制的时候与之绑定的声明（或 FVF）中。

符合以下规则的声明可以用来在固定功能流水线（假设不需要 tessellation）中进行渲染。

只使用 D3DDECLMETHOD_DEFAULT。

顶点元素之间不能有间隔（SKIP 在 DirectX 8.x 声明中是不允许的）

指定 POSITION 用途。

为 POSITION 用途指定 D3DDECLTYPE_FLOAT3 数据类型。

下表中没有描述的顶点元素在所有 DirectX 8.0 之前的驱动程序上都无法被转换为有效的 FVF 码，因此无法用于固定功能顶点处理。

把 DirectX 8.x 的顶点声明映射到 DirectX 9.0 的顶点声明

DirectX 8.x	DirectX 9.0 用途	DirectX 9.0 用途索引
D3DVSDE_POSITION	D3DDECLUSAGE_POSITION	0
D3DVSDE_POSITION2	D3DDECLUSAGE_POSITION	1
D3DVSDE_NORMAL	D3DDECLUSAGE_NORMAL	0
D3DVSDE_NORMAL2	D3DDECLUSAGE_NORMAL	1
D3DVSDE_BLENDWEIGHT	D3DDECLUSAGE_BLENDWEIGHT	0
D3DVSDE_BLENDINDICES	D3DDECLUSAGE_BLENDINDICES	0
D3DVSDE_PSIZE	D3DDECLUSAGE_PSIZE	0

D3DVSDE_DIFFUSE	D3DDECLUSAGE_COLOR	0
D3DVSDE_SPECULAR	D3DDECLUSAGE_COLOR	1
D3DVSDE_TEXCOORDn	D3DDECLUSAGE_TEXCOORD	n

把 FVF 码转换为 DirectX 9.0 的顶点声明

FVF	数据类型	用途	用途索引
D3DFVF_XYZ	D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_POSITION	0
D3DFVF_XYZRHW	D3DDECLTYPE_FLOAT4	D3DDECLUSAGE_POSITIONT	0
D3DFVF_XYZW	D3DDECLTYPE_FLOAT4	D3DDECLUSAGE_POSITIONT	0
D3DFVF_XYZB5 and D3DFVF_LASTBETA_UBYTE4	D3DVSDT_FLOAT3, D3DVSDT_FLOAT4, D3DVSDT_UBYTE4	D3DDECLUSAGE_POSITION, D3DDECLUSAGE_BLENDWEIGHT, D3DDECLUSAGE_BLENDINDICES	0
D3DFVF_XYZB5	D3DDECLTYPE_FLOAT3, D3DDECLTYPE_FLOAT4, D3DDECLTYPE_FLOAT1	D3DDECLUSAGE_POSITION, D3DDECLUSAGE_BLENDWEIGHT, D3DDECLUSAGE_BLENDINDICES	0
D3DFVF_XYZBn (n=1..4)	D3DDECLTYPE_FLOAT3 D3DDECLTYPE_FLOATn	D3DDECLUSAGE_POSITION, D3DDECLUSAGE_BLENDWEIGHT	0
D3DFVF_XYZBn (n=1..4) and D3DFVF_LASTBETA_UBYTE4	D3DDECLTYPE_FLOAT3 D3DDECLTYPE_FLOAT(n-1) D3DDECLTYPE_UBYTE4	D3DDECLUSAGE_POSITION, D3DDECLUSAGE_BLENDWEIGHT, D3DDECLUSAGE_BLENDINDICES	0
D3DFVF_NORMAL	D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_NORMAL	0
D3DFVF_PSIZE	D3DDECLTYPE_FLOAT1	D3DDECLUSAGE_PSIZE	0
D3DFVF_DIFFUSE	D3DDECLTYPE_D3DCOLOR	D3DDECLUSAGE_COLOR	0
D3DFVF_SPECULAR	D3DDECLTYPE_D3DCOLOR	D3DDECLUSAGE_COLOR	1
D3DFVF_TEXCOORDSIZEm(n)	D3DDECLTYPE_FLOATm	D3DDECLUSAGE_TEXCOORD	n

把 DirectX 9.0 的顶点声明转换为 DirectX 8.x 的顶点声明

用途	用途索引	DirectX decl
D3DDECLUSAGE_POSITION	0	D3DVSDE_POSITION
D3DDECLUSAGE_POSITION	1	D3DVSDE_POSITION2
D3DDECLUSAGE_BLENDWEIGHT	0	D3DVSDE_BLENDWEIGHT
D3DDECLUSAGE_BLENDINDICES	0	D3DVSDE_BLENDINDICES
D3DDECLUSAGE_NORMAL	0	D3DVSDE_NORMAL
D3DDECLUSAGE_NORMAL	1	D3DVSDE_NORMAL2
D3DDECLUSAGE_PSIZE	0	D3DVSDE_PSIZE
D3DDECLUSAGE_COLOR	0	D3DVSDE_DIFFUSE
D3DDECLUSAGE_COLOR	1	D3DVSDE_SPECULAR
D3DDECLUSAGE_TEXCOORD	n	D3DVSDE_TEXTUREn, n <= 7

把 DirectX 9.0 的顶点声明转换为 FVF 码

在 DirectX 8.0 及以后的驱动程序上，更多的类型可以成功地被转换为有效的声明，因此可以被用于固定功能顶点处理。

如果使用了 DirectX 8.x 驱动程序，那么顶点声明会根据以下规则被转换为 DirectX 8.x 的声明。

如果使用其它的用途组合，那么对 Direct3D 的调用会失败。

对于 D3DVSD_STREAM 中每个新的数据流，会插入一个标记（token）。

顶点元素不能共享数据流中相同的偏移量，并且不能重叠。

数据类型必须小于或等于 D3DDECLTYPE_SHORT4。

对于每个使用了 D3DDECLMETHOD_DEFAULT 方法的顶点元素，会插入 D3DVSD_REG 标记。

Usage 和 UsageIndex 会根据下表被转换为寄存器值。如果使用 DrawRectPatch（RT-patches），那么 tessellator 的输出应该符合前面的规则。

数据类型	用途	用途索引	FVF
D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_POSITION	0	D3DFVF_XYZ
D3DDECLTYPE_FLOATn	D3DDECLUSAGE_BLENDWEIGHT	0	D3DFVF_XYZBn
D3DDECLTYPE_UBYTE4	D3DDECLUSAGE_BLENDINDICES	0	D3DFVF_XYZB (nWeights+1)
D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_NORMAL	0	D3DFVF_NORMAL
D3DDECLTYPE_FLOAT1	D3DDECLUSAGE_PSIZE	0	D3DFVF_PSIZE
D3DDECLTYPE_D3DCOLOR	D3DDECLUSAGE_COLOR	0	D3DFVF_DIFFUSE
D3DDECLTYPE_D3DCOLOR	D3DDECLUSAGE_COLOR	1	D3DFVF_SPECULAR
D3DDECLTYPE_FLOATm	D3DDECLUSAGE_TEXCOORD	n	D3DFVF_TEXCOORDSIZEm(n)
D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_POSITION	1	N/A
D3DDECLTYPE_FLOAT3	D3DDECLUSAGE_NORMAL	1	N/A

像素着色器

在 Microsoft DirectX® 8.0 之前，Microsoft® Direct3D®使用固定功能流水线把三维几何体转换为屏幕上的像素。用户通过设置流水线的属性来控制 Direct3D 进行变换、光照和渲染像素的方式。固定功能顶点格式在编译的时候定义并决定输入顶点的格式，一旦定义，用户在运行的时候就几乎无法控制流水线的改变。

通过允许对顶点的变换、光照和对每个像素的着色等功能进行编程，着色器把图形流水线引入了一个新的高度。像素着色器是一些小程序，在对三角形进行光栅化操作时运行。这在渲染像素的方法上给了用户更高一级的灵活性。

像素着色器包含由ASCII文本组成的像素着色器指令。算术指令可以用来进行漫反射和/或镜面反射光照计算。纹理寻址指令提供了多种读取和应用纹理数据的操作。着色器具有这样的功能，可以给颜色分量设置掩码以及交换颜色分量。着色器的正文看起来有点像汇编语言，它用Direct3D扩展（D3DX）进行汇编，输入可以是文本字符串或是文件。汇编器的输出是一系列操作码，应用程序可以通过IDirect3DDevice9::CreatePixelShader方法把这些操作码提供给Direct3D。像素着色器有几个版本，请参阅着色器参考手册。

创建像素着色器 - 包含了示例代码，用像素着色器对物体的漫反射色进行高洛德插值。本示例包含了对所使用方法的详细描述。

确认对像素着色器的支持 - 提供了更为详细的说明，解释如何检测硬件对像素着色器的支持。

纹理操作的转换 - 提供了一些把纹理操作转换为像素着色器指令的例子。

对纹理的一些考虑 - 详细说明了在像素着色器中被忽略的纹理层状态。

像素着色器示例 - 显示了更多的示例代码，添加纹理并把顶点颜色和纹理进行混合。
调试 - 提供了有关调试的信息。
指令集的变化非常快。为避免在使用指令时出现问题，请查阅硬件开发商的网站。或者，也可以使用高级着色器语言，这样就可以得到由Direct3D扩展（D3DX）编译得到的着色器指令。
创建像素着色器
本示例用像素着色器对一个四边形的漫反射色进行高洛德插值。示例显示了着色器文件的内容以及应用程序中所需的代码。
以下是创建像素着色器所需的步骤：
第 1 步：检查对像素着色器的支持。
第 2 步：声明顶点数据。
第 3 步：设计像素着色器。
第 4 步：创建像素着色器。
第 5 步：渲染输出像素。
如果读者已经知道如何构建并运行 Direct3D 示例，那么可以从本示例中复制代码并粘贴到已有的应用程序中。
第 1 步
要检查对像素着色器的支持，应该使用以下代码。这个例子检查 1.1 版本的像素着色器。

```
D3DCAPS9 caps;
m_pd3dDevice->GetDeviceCaps(&caps);        // 使用 m_pd3dDevice 前要进行初始化
if( caps.PixelShaderVersion < D3DPS_VERSION(1,1) )
    return E_FAIL;
```

caps 结构会返回硬件可用的能力。要用 D3DPS_VERSION 宏检查当前硬件支持的所有着色器版本。如果 caps 返回的版本小于 1.1，那么这个调用会失败。反之，对所有大于或等于 1.1 的版本，调用会成功。如果硬件不支持被测试的着色器版本，那么应用程序将不得不退而使用别的渲染方法（也许可以使用一个较低版本的着色器）。
第 2 步
这个示例使用了一个四边形，由两个三角形组成。每个顶点的数据结构包含了位置和漫反射色数据。D3DFVF_CUSTOMVERTEX 宏定义了与顶点数据相匹配的数据结构。实际的顶点数据在全局数组 g_Vertices 中声明。四个顶点以原点为中心，每个顶点具有不同的漫反射色。

```
// 声明顶点数据结构。
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD diffuseColor;
};

// 声明自定义 FVF 宏。
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE)

// 声明顶点位置和漫反射色数据。
CUSTOMVERTEX g_Vertices[]=
{
//      x       y       z    漫反射色
    { -1.0f, -1.0f, 0.0f, 0xffff0000 },  // 红 - 左下
```

```
    { +1.0f, -1.0f, 0.0f, 0xff00ff00 },  // 绿 - 右下
    { +1.0f, +1.0f, 0.0f, 0xff0000ff },  // 蓝 - 右上
    { -1.0f, +1.0f, 0.0f, 0xffffffff },  // 白 - 左上
};
```

第 3 步

这个着色器把经过高洛德插值的漫反射色数据复制到输出像素。着色器文件 PixelShader.txt 如下所示：

```
ps_1_1          // 版本指令
mov r0,v0       // 把顶点的漫反射色复制到输出寄存器。
```

像素着色器文件的第一条指令声明了像素着色器的版本，此处为 1.1。

第二条指令把颜色寄存器(v0)的内容复制到输出寄存器(r0)。因为在第 1 步中声明的顶点数据已经包含了经过插值的漫反射色，所以颜色寄存器包含了顶点的漫反射色。输出寄存器决定渲染目标使用的像素颜色（因为本例中没有更多的处理，如雾，所以输出寄存器就是最终的像素颜色）。

第 4 步

像素着色器由像素着色器指令创建。本例中，指令被包含在一个单独的文件中。指令也可以被包含在一个文本字符串中。

```
LPD3DXBUFFER pCode;                 // 存放经过汇编的着色器代码的缓存
LPD3DXBUFFER pErrorMsgs;            // 存放错误信息的缓存
TCHAR        strPixelShaderPath[512];// 用来定位着色器文件
DXUtil_FindMediaFileCb( strPixelShaderPath, sizeof(strPixelShaderPath),
                        _T("PixelShader.txt") );
```

这个函数是示例框架使用的一个辅助函数，许多示例都以它为基础。

```
LPDIRECT3DPIXELSHADER9 m_pPixelShader;
D3DXAssembleShaderFromFile( strPixelShaderPath, NULL, NULL, 0,
                            &pCode, &pErrorMsgs, NULL );
m_pd3dDevice->CreatePixelShader( (DWORD*)pCode->GetBufferPointer(),
                                 &m_pPixelShader );
```

着色器创建完成后，指针 *m_pPixelShader* 用来对它进行引用。

第 5 步

除了用像素着色器句柄来设置着色器外，渲染输出像素的过程和使用固定功能流水线类似。

```
// 在本例中关闭关照。它不会对最终像素的颜色产生影响。
// 像素颜色完全由经过插值的顶点颜色决定。
m_pd3dDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTEX) );
m_pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
m_pd3dDevice->SetPixelShader( m_pPixelShader );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
```

源顶点数据由 SetStreamSource 设置。本例中，*SetFVF* 使用在声明顶点数据时定义的 FVF 码告诉 Direct3D 进行固定功能顶点处理。顶点着色器和像素着色器既可以一起使用，也可以分开使用。可以用固定功能流水线代替这两者。*SetPixelShader* 设置像素着色器，而 *DrawPrimitive* 则绘制四边形。

确认对像素着色器的支持

应用程序可以查询D3DCAPS9的成员以确定硬件对像素着色器所涉及的操作的支持程度。下表列出

了与可编程像素处理有关的设备能力。

设备能力	描述
PixelShader1xMaxValue	寄存器中可存储的值的范围为[-PixelShader1xMaxValue, PixelShader1xMaxValue]。这个值只对版本 1.1 到 1.4 有效。
MaxSimultaneousTextures	用于固定功能流水线，纹理取样器的数量为 MaxTextureBlendStages 除以 MaxSimultaneousTextures。用于像素着色器的纹理取样器的数量在接下来的表中显示。
PixelShaderVersion	硬件支持的像素着色器的版本。版本号小于或等于该值的像素着色器被支持。

可用于像素着色器的纹理取样器的数量取决于像素着色器的版本。

像素着色器版本	纹理取样器的数量
ps_1_1 - ps_1_3	4 个纹理取样器
ps_1_4	6 个纹理取样器
ps_2_0 - ps_3_0	16 个纹理取样器
Fixed function pixel shader	MaxTextureBlendStages/MaxSimultaneousTextures 个纹理取样器

PixelShaderVersion 的第一个字节包含次版本号，第二个字节包含主版本号。经过汇编的着色器的第一个标记就是像素着色器的版本。每种硬件实现都会设置该版本号，表示它能完全支持的像素着色器的最高版本。

纹理操作的转换

像素着色器在以下几个方面扩展并一般化了 Microsoft DirectX® 6.0 和 7.0 的多重纹理能力。加入了一组通用读/写寄存器，这样就允许更为灵活的表达式。用 D3DTA_CURRENT 的连续级联需要为每层指定一个单独的结果寄存器参数。

D3DTOP_MODULATE2X 和 D3DTOP_MODULATE4X 纹理操作被分成了单独的修饰符，可用于任何指令。这样就无需单独的 D3DTOP_MODULATE 和 D3DTOP_MODULATE2X 操作了。

乘加操作加入了可选的第三个参数，因此程序化的像素着色器可以执行 arg1 ×arg2 + arg0 的操作。这样就不再需要 D3DTOP_MODULATEALPHA_ADDCOLOR 和 D3DTOP_MODULATECOLOR_ADDALPHA 纹理操作了。

混合操作加入了可选的第三个参数，因此程序化的像素着色器可以用 arg0 作为 arg1 和 arg2 之间的混合比。这样就不再需要 D3DTOP_BLENDDIFFUSEALPHA, D3DTOP_BLENDTEXTUREALPHA, D3DTOP_BLENDFACTORALPHA, D3DTOP_BLENDTEXTUREALPHAPM, 和 D3DTOP_BLENDCURRENTALPHA 纹理操作了。

对纹理寻址的修改操作，如 D3DTOP_BUMPENVMAP，被从颜色和阿尔法操作中分离出来并作为第三种操作类型，专门用于纹理寻址操作。

为了有效地支持这种新增的灵活性，API 的语法从 DWORD 对改成了 ASCII 汇编代码语法。这样就暴露了程序化的像素着色器所提供的功能。

注意在使用像素着色器时，镜面反射加法不专门由一个渲染状态控制，如果需要，这可能由像素着色器实现。但是，雾混合仍然由固定功能流水线执行。

对纹理的一些考虑

像素着色器完全取代了由Microsoft DirectX® 6.0 和 7.0 的多重纹理API提供的像素混合功能，尤其是那些由D3DTSS_COLOROP, D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHAOP, D3DTSS_ALPHAARG1和D3DTSS_ALPHAARG2纹理层状态、相关的参数和修饰符定义的操作。如果设置

了程序化的像素着色器，那么这些状态会被忽略。

纹理层和取样器状态

如果像素着色器正在运行，那么以下纹理层状态仍然会被使用。对这些状态的使用取决于像素着色器的版本，如下表所示。

纹理层状态	版本
D3DTSS_BUMPENVMAT00	1_1 - 1_4
D3DTSS_BUMPENVMAT01	1_1 - 1_4
D3DTSS_BUMPENVMAT10	1_1 - 1_4
D3DTSS_BUMPENVMAT11	1_1 - 1_4
D3DTSS_BUMPENVLSCALE	1_1 - 1_3
D3DTSS_BUMPENVLOFFSET	1_1 - 1_3
D3DTSS_TEXCOORDINDEX	1_1 - 1_3. 仅对固定功能顶点处理有效。
D3DTSS_TEXTURETRANSFORMFLAGS	1_1 - 1_3. 仅对固定功能顶点处理有效。

如果像素着色器正在运行，那么下面的取样器状态仍然会被使用。

取样器状态	版本
D3DSAMP_ADDRESSU	所有版本
D3DSAMP_ADDRESSV	所有版本
D3DSAMP_ADDRESSW	所有版本
D3DSAMP_BORDERCOLOR	所有版本
D3DSAMP_MAGFILTER	所有版本
D3DSAMP_MINFILTER	所有版本
D3DSAMP_MIPFILTER	所有版本
D3DSAMP_MIPMAPLODBIAS	所有版本
D3DSAMP_MAXMIPLEVEL	所有版本
D3DSAMP_MAXANISOTROPY	所有版本
D3DSAMP_SRGBTEXTURE	所有版本
D3DSAMP_ELEMENTINDEX	所有版本
D3DSAMP_DMAPOFFSET	仅顶点着色器（位移贴图）

因为纹理层状态不是像素着色器的一部分，在编译着色器时它们是不可用的，所以驱动程序无法对它们做任何假定。例如，驱动程序不能在那时区分出双线性过滤和三线性过滤。应用程序可以自由地改变这些状态而无需重新生成当前绑定的着色器。

像素着色器和纹理取样

纹理取样和过滤操作由标准纹理层状态中的放大、缩小、mip过滤、以及环绕寻址模式控制。更多信息，请参阅纹理层状态。因为驱动程序在编译时也无法得到这些信息，所以着色器必须能够在这些状态改变后继续运行。应用程序负责设置像素着色器所需的正确类型的纹理（二维贴图，立方体贴图，立体贴图等）。如果设置不正确的纹理类型，那么将会产生不可预料的结果。

在像素着色器之后的处理

其它一些像素操作——如雾混合、模板操作、以及渲染目标混合——在着色器执行以后发生。为了支持本主题描述的新特性，渲染目标混合的语法已经做了更新。

像素着色器的输入

对像素着色器版本 1.1 到 2.0 来说，漫反射色和镜面反射色在给着色器使用之前被截取到范围 0 到 1 之间，因为这是着色器的有效输入范围。
输入到像素着色器的颜色值被认为是经过透视校正的，但并非所有硬件都能保证这一点。寻址处理器根据纹理坐标产生的颜色总是以透视校正的方式进行迭代。但是，在迭代过程中，它们也会被截取到范围 0 到 1 之间。
像素着色器输出
对像素着色器版本 1.1 到 1.4 来说，像素着色器产生的输出是寄存器 r0 的内容。该寄存器的值会在着色器处理结束后被送往雾处理阶段和渲染目标混合器。
对像素着色器版本 2.0 以上来说，输出的颜色值为 oC0 到 oC4。
像素着色器示例
本节包含三个像素着色器示例。每个示例都建立在前一个示例的基础上，并增加一些功能。
应用纹理贴图
把顶点漫反射色和纹理进行混合
用颜色值把两张纹理进行混合
应用纹理贴图
本示例把一张纹理贴图应用于一个四边形。本例和前例的区别在于：
顶点数据结构和 FVF 码包含了纹理坐标。顶点数据包含了 u, v 数据。因为像素的颜色将从纹理贴图得到，所以顶点数据不再需要漫反射色。
用IDirect3DDevice9::SetTexture把纹理连接到纹理层 0。
着色器用 t0 纹理寄存器取代 v0 漫反射色寄存器。
示例代码如下：

```
// 定义顶点数据结构。
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    FLOAT u1, v1;
};

// 定义相应的 FVF 码。
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_TEX|D3DFVF_TEXCOORDSIZE2(0))

// 创建包含位置和纹理坐标的顶点数据。
static CUSTOMVERTEX g_Vertices[]=
{
    //  x      y     z     u1    v1
    { -1.0f, -1.0f, 0.0f,  0, 1, },
    {  1.0f, -1.0f, 0.0f,  1, 1, },
    {  1.0f,  1.0f, 0.0f,  1, 0, },
    { -1.0f,  1.0f, 0.0f,  0, 0, },
    // 为了和 Windows 从上到下的约定一致，v1 被颠倒了。
    // 左上角的纹理坐标为(0,0)。
    // 右下角的纹理坐标为(1,1)。
};
```

```
// 创建纹理。这个文件包含在供下载的 DirectX 9.0 SDK 的媒体文件中。
TCHAR        strTexturePath[512];
DXUtil_FindMediaFileCb( strTexturePath, sizeof(strTexturePath),
                        _T("DX5_Logo.bmp") );

LPDIRECT3DTEXTURE9      m_pTexture0;
D3DUtil_CreateTexture( m_pd3dDevice, strTexturePath,
                       &m_pTexture0, D3DFMT_R5G6B5 );

// Create the pixel shader.
TCHAR        strShaderPath[512];
DXUtil_FindMediaFileCb( strShaderPath, sizeof(strShaderPath),
                        _T("PixelShader2.txt") );
```

这个函数是示例框架使用的一个辅助函数，许多示例都以它为基础。

```
LPD3DXBUFFER pCode;                  // 用来存放经过汇编的着色器代码的缓存
LPD3DXBUFFER pErrorMsgs;             // 用来存放错误信息的缓存
LPDIRECT3DPIXELSHADER9 m_pPixelShader;
D3DXAssembleShaderFromFile( strShaderPath, 0, NULL, &pCode, NULL );
m_pd3dDevice->CreatePixelShader( (DWORD*)pCode->GetBufferPointer(),
                                          &m_pPixelShader );
m_pd3dDevice->SetPixelShader( m_pPixelShader );

// 载入纹理并渲染输出像素。
m_pd3dDevice->SetTexture( 0, m_pTexture0 );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );

// 文件"PixelShader2.txt"的内容
// 把纹理贴图应用于物体的顶点。
ps_1_1      // 文件的第一条必须是版本指令。
tex t0      // 声明纹理寄存器 t0，从纹理层 0 载入。
mov r0, t0  // 把纹理寄存器数据(t0)复制到输出寄存器(r0)。
```

得到的图像如下面所示。

把顶点漫反射色和纹理进行混合

本例把纹理贴图中的颜色与顶点的颜色进行混合。本例与前例的区别如下：

顶点的数据结构，FVF 码和顶点数据包含了漫反射色。

着色器文件用乘法(mul)指令把纹理的颜色和顶点的漫反射色进行混合。

创建纹理和载入纹理的代码是相同的，放在这里是为了保持代码的完整性。

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD color1;
    FLOAT tu1, tv1;
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX1|
                             D3DFVF_TEXCOORDSIZE2(0))
static CUSTOMVERTEX g_Vertices[]=
{
    //  x      y     z     diffuse     u1    v1
    { -1.0f, -1.0f, 0.0f, 0xffff0000, 0, 1, }, // 红
    {  1.0f, -1.0f, 0.0f, 0xff00ff00, 1, 1, }, // 绿
    {  1.0f,  1.0f, 0.0f, 0xff0000ff, 1, 0, }, // 蓝
    { -1.0f,  1.0f, 0.0f, 0xffffffff, 0, 0, }, // 白
    // 为了和 Windows 从上到下的约定一致，v1 被颠倒了。
    // 左上角的纹理坐标为(0,0)。
    // 右下角的纹理坐标为(1,1)。
};

// 创建纹理。这个文件包含在供下载的 DirectX 9.0 SDK 的媒体文件中。
TCHAR        strTexturePath[512];
DXUtil_FindMediaFileCb( strTexturePath, sizeof(strTexturePath),
                        _T("DX5_Logo.bmp") );

LPDIRECT3DTEXTURE9      m_pTexture0;
D3DUtil_CreateTexture( m_pd3dDevice, strTexturePath,
                       &m_pTexture0, D3DFMT_R5G6B5 );

// 创建像素着色器。
TCHAR        strShaderPath[512];
DXUtil_FindMediaFileCb( strShaderPath, sizeof(strShaderPath),
                        _T("PixelShader3.txt") );

LPD3DXBUFFER pCode;                 // 用来存放经过汇编的着色器代码的缓存
LPD3DXBUFFER pErrorMsgs;            // 用来存放错误信息的缓存
LPDIRECT3DPIXELSHADER9 m_pPixelShader;
D3DXAssembleShaderFromFile( strShaderPath, 0, NULL, &pCode, NULL );
m_pd3dDevice->CreatePixelShader( (DWORD*)pCode->GetBufferPointer(),
                                       &m_pPixelShader );
m_pd3dDevice->SetPixelShader( m_pPixelShader );

// 载入纹理并渲染输出像素。
```

```
m_pd3dDevice->SetTexture( 0, m_pTexture0 );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );

// 文件"PixelShader3.txt"的内容
ps_1_1          // 版本指令
tex t0          // 声明纹理寄存器 t0, 从纹理层 0 载入
mul r0, v0, t0  // v0*t0, 然后复制到 r0
```

着色器的输入如下面所示。第一幅图为顶点颜色，第二幅图为纹理贴图。


得到的图像如下面所示，也就是顶点的颜色和纹理贴图混合的结果。

用颜色值把两张纹理进行混合

本例把两张纹理贴图进行混合，顶点的颜色用来决定每张纹理贴图的颜色所占的比例。本例和前例的区别如下：

因为使用了两张纹理，所以顶点的数据结构，FVF 码，以及顶点数据包含了第二组纹理坐标。另外 IDirect3DDevice9::SetTexture 也调用了两次，并设置两个纹理层的状态。

着色器文件声明了两个纹理寄存器并使用线性插值(lrp)指令把两张纹理进行混合。漫反射色的值用来决定两张纹理在输出的颜色中所占的比例。

下面是示例代码。

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD color;
    FLOAT tu1, tv1;
    FLOAT tu2, tv2;    // 第二组纹理坐标
};
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3D_FVF_DIFFUSE|D3DFVF_TEX2|
                             D3DFVF_TEXCOORDSIZE4(0))
static CUSTOMVERTEX g_Vertices[]=
{
    //  x      y     z     color       u1    v1    u2    v2
    { -1.0f, -1.0f, 0.0f, 0xff0000ff, 1.0f, 1.0f, 1.0f, 1.0f },
```

```
    { +1.0f, -1.0f, 0.0f, 0xffff0000, 0.0f, 1.0f, 0.0f, 1.0f },
    { +1.0f, +1.0f, 0.0f, 0xffffff00, 0.0f, 0.0f, 0.0f, 0.0f },
    { -1.0f, +1.0f, 0.0f, 0xffffffff, 1.0f, 0.0f, 1.0f, 0.0f },
};

// 创建纹理。这个文件包含在供下载的 DirectX 9.0 SDK 的媒体文件中。
TCHAR        strTexturePath[512];
LPDIRECT3DTEXTURE9      m_pTexture0, m_pTexture1;

DXUtil_FindMediaFileCb( strTexturePath, sizeof(strTexturePath),
                        _T("DX5_Logo.bmp") );
D3DUtil_CreateTexture( m_pd3dDevice, strTexturePath,
                       &m_pTexture0, D3DFMT_R5G6B5 );

DXUtil_FindMediaFileCb( strTexturePath, sizeof(strTexturePath),
                        _T("snow2.jpg") );
D3DUtil_CreateTexture( m_pd3dDevice, strTexturePath,
                       &m_pTexture0, D3DFMT_R5G6B5 );



// 创建像素着色器。
TCHAR        strShaderPath[512];
DXUtil_FindMediaFileCb( strShaderPath, sizeof(strShaderPath),
                        _T("PixelShader4.txt") );

LPD3DXBUFFER pCode;                  // 用来存放经过汇编的着色器代码的缓存
LPD3DXBUFFER pErrorMsgs;             // 用来存放错误信息的缓存
LPDIRECT3DPIXELSHADER9 m_pPixelShader;
D3DXAssembleShaderFromFile( strShaderPath, 0, NULL, &pCode, NULL );
m_pd3dDevice->CreatePixelShader( (DWORD*)pCode->GetBufferPointer(),
                                       &m_pPixelShader );
m_pd3dDevice->SetPixelShader( m_pPixelShader );

// Load the textures stages.
m_pd3dDevice->SetTexture( 0, m_pTexture0 );
m_pd3dDevice->SetTexture( 1, m_pTexture1 );  // 使用第二层纹理

m_pd3dDevice->SetStreamSource( 0, m_pQuadVB, sizeof(CUSTOMVERTEX) );
m_pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
m_pd3dDevice->SetPixelShader( m_pPixelShader );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );

// 文件"PixelShader4.txt"的内容
ps_1_1              // 像素着色器版本。
```

```
tex t0                  // 纹理寄存器 t0 从第 0 层载入。
tex t1                  // 纹理寄存器 t1 从第 1 层载入。
mov r1, t1              // 把纹理 t1 复制到输出寄存器 r1。
lrp r0, v0, t0, r1  // 用 v0 中指定的比例在 t0 和 r1 间进行线性插值。
```

得到的图像如下：

调试

用 Visual Studio 进行调试

Microsoft® Visual Studio®存在一个扩展以支持对某些类型的顶点着色器进行调试。更多信息请参阅着色器调试器。

其它着色器调试器

一些图形芯片厂商在他们的网站提供着色器调试工具。可以通过查找互联网或阅读下面的文章找到这些工具。读者可以在程序运行的过程中把调试器连上去，并用调试器单步运行着色器。通过设置断点，读者可以一次执行一行着色器代码并观察寄存器状态的变化。有关顶点着色器和调试技巧的更多信息，请参阅Using Vectex Shaders: Part I。

对纹理混合进行调试

另一个示例程序MFC Tex Sample是SDK安装的组成部分。这个MFC程序是学习如何在固定功能流水线中进行多重纹理混合操作的一个不错的方法。

诊断支持

另一个帮助调试 Microsoft® DirectX®问题的选择是使用 DirectX 诊断程序(DXDiag.exe)创建一个计算机的内存转储。在计算机崩溃后运行 DxDiag.exe 就可以得到内存转储，可以通过 More Help 分栏中的 Report 按钮将之发送给 Microsoft，也可以直接将文件发送到 directx@microsoft.com。内存转储文件可以用来追捕并重现问题。

可以在http://msdn.microsoft.com/directx找到更多有关调试的信息。

更多信息，请参阅DirectX诊断工具。