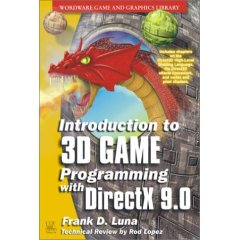
# 《DirectX 9 3D游戏设计入门》



《Introduction to 3D Game Programming with DirectX 9》

by Frank Luna  ISBN:1556229135

Wordware Publishing © 2003 (388 pages)

This text provides an introduction to programming interactive 3D computer graphics using DirectX 9.0, with an emphasis on game development.

随书代码下载地址：<http://www.moon-labs.com/ml_book_samples.htm>

翻译：

01章 – 12章 **翁云兵** Email：[WengYB@126.com](mailto:WengYB@126.com)

13章 – 19章 **天兵** Email：[zzprogram@21cn.com](mailto:zzprogram@21cn.com)

首先感谢[游戏开发资源网](http://www.gameres.com/)，让我发现了**翁云兵**老兄翻译的前12章，从此让我有信心从事D3D游戏的开发，再次感谢**翁云兵，**还有在我学习中给予帮助的网友们，对初学者的帮助与指导。出于同样的目的，我将整理后的文档再次发到网上，希望能帮助更多热爱游戏开发的朋友。

本教程由**天兵**收集、翻译并整理，最初是为方便自己学习，非专业翻译，如有不准确的地方请谅解。欢迎提出翻译错误的章节，请联系：[zzprogram@21cn.com](mailto:zzprogram@21cn.com)。

本书没有取得作者授权，此翻译版本纯属个人爱好，不得用于传播以及其他商业目的，只供爱好者参考使用，希望用户下载完毕后24小时之内自觉删除！如果喜欢请购买正版支持！

[《DirectX 9 3D游戏设计入门》 1](#_Toc138011431)

[第一部分 必备的数学知识 (Mathematical Prerequisites) 10](#_Toc138011432)

[目标 10](#_Toc138011433)

[三维空间中的向量 11](#_Toc138011434)

[向量相等 13](#_Toc138011435)

[计算向量大小（向量的模） 14](#_Toc138011436)

[标准化向量 14](#_Toc138011437)

[向量相加 15](#_Toc138011438)

[向量相减 16](#_Toc138011439)

[标量与向量的乘积 16](#_Toc138011440)

[点积 17](#_Toc138011441)

[叉积 17](#_Toc138011442)

[矩阵 18](#_Toc138011443)

[相等、数乘矩阵以及相加 19](#_Toc138011444)

[乘法 19](#_Toc138011445)

[单位矩阵 20](#_Toc138011446)

[逆转 21](#_Toc138011447)

[矩阵的转置 21](#_Toc138011448)

[D3DX 矩阵 22](#_Toc138011449)

[基本变换 24](#_Toc138011450)

[矩阵平移 25](#_Toc138011451)

[矩阵旋转 26](#_Toc138011452)

[矩阵缩放 28](#_Toc138011453)

[综合变换 29](#_Toc138011454)

[一些向量变换函数 30](#_Toc138011455)

[平面 31](#_Toc138011456)

[D3DX平面 31](#_Toc138011457)

[点和平面的空间关系 32](#_Toc138011458)

[创建平面 32](#_Toc138011459)

[标准化平面 33](#_Toc138011460)

[变换平面 34](#_Toc138011461)

[点到平面上最近的点 34](#_Toc138011462)

[射线（可选的） 35](#_Toc138011463)

[射线 35](#_Toc138011464)

[线/面相交 35](#_Toc138011465)

[摘要（略） 36](#_Toc138011466)

[第二部分 Direct3D基础 (Direct3D Fundamentals) 37](#_Toc138011467)

[第一章 初始化Direct3D (Direct3D Initialization) 37](#_Toc138011468)

[目标 37](#_Toc138011469)

[1.1 Direct3D概述 37](#_Toc138011470)

[1.1.1 REF设备 38](#_Toc138011471)

[1.1.2 D3DDEVTYPE 38](#_Toc138011472)

[1.2 COM 38](#_Toc138011473)

[1.3 一些准备工作 38](#_Toc138011474)

[1.3.1 表面 39](#_Toc138011475)

[1.3.2 Multisampling 40](#_Toc138011476)

[1.3.3像素格式 41](#_Toc138011477)

[1.3.4 内存池 41](#_Toc138011478)

[1.3.5 交换链和页面切换 41](#_Toc138011479)

[1.3.6 深度缓冲 42](#_Toc138011480)

[1.3.7 顶点处理 43](#_Toc138011481)

[1.3.8 设备能力 43](#_Toc138011482)

[1.4 初始化Direct3D 44](#_Toc138011483)

[1.4.1获得IDirect3D9接口 44](#_Toc138011484)

[1.4.2 检测硬件顶点处理 45](#_Toc138011485)

[1.4.3 填充D3DPRESENT\_PARAMETERS结构 46](#_Toc138011486)

[1.4.4 创建IDirect3DDevice9对象 47](#_Toc138011487)

[1.5 初始化Direct3D实例 48](#_Toc138011488)

[1.5.1 d3dUtility.h/cpp 49](#_Toc138011489)

[1.5.2 实例框架 50](#_Toc138011490)

[1.5.3 D3D Init实例 51](#_Toc138011491)

[1. 6摘要（略） 53](#_Toc138011492)

[第二章 渲染管线 (The Rendering Pipeline) 54](#_Toc138011493)

[目标 54](#_Toc138011494)

[2.1表现模型 54](#_Toc138011495)

[2.1.1 顶点格式 55](#_Toc138011496)

[2.1.2 三角形 56](#_Toc138011497)

[2.1.3 索引 57](#_Toc138011498)

[2.2虚拟照相机 57](#_Toc138011499)

[2.3 渲染管线 58](#_Toc138011500)

[2.3.1自身坐标系（Local Space） 59](#_Toc138011501)

[2.3.2世界坐标系（World Space） 59](#_Toc138011502)

[2.3.3视图坐标系（View Space） 60](#_Toc138011503)

[2.3.4背面拣选（Backface Culling） 61](#_Toc138011504)

[2.3.5光源（Lighting） 62](#_Toc138011505)

[2.3.6裁剪（Clipping） 63](#_Toc138011506)

[2.3.7投影（Projection） 63](#_Toc138011507)

[2.3.8视口变换（Viewport Transform） 65](#_Toc138011508)

[2.3.9光栅化（Rasterization） 66](#_Toc138011509)

[2.4 摘要(略) 66](#_Toc138011510)

[第三章 在Direct3D中绘制 (Drawing in Direct3D) 67](#_Toc138011511)

[目标 67](#_Toc138011512)

[3.1顶点/索引缓存 67](#_Toc138011513)

[3.1.1创建一个顶点和索引缓存 67](#_Toc138011514)

[3.1.2 访问缓冲内存 69](#_Toc138011515)

[3.1.3 找回顶点和索引缓存信息 70](#_Toc138011516)

[3.2 渲染状态 70](#_Toc138011517)

[3.3 绘制准备 71](#_Toc138011518)

[3.4用顶点/索引缓存绘制 71](#_Toc138011519)

[3.4.1 IDirect3DDevice9::DrawPrimitive 72](#_Toc138011520)

[3.4.2 IDirect3DDevice9::DrawIndexedPrimitive 72](#_Toc138011521)

[3.4.3 开始/结束场景 73](#_Toc138011522)

[3.5 D3DX几何物体 73](#_Toc138011523)

[3.6 实例程序：三角形、立方体、茶壶、D3DXCreate\* 75](#_Toc138011524)

[3.7 摘要（略） 79](#_Toc138011525)

[第四章 色彩 (Color) 80](#_Toc138011526)

[目标 80](#_Toc138011527)

[4.1 颜色表示法 80](#_Toc138011528)

[4.2 顶点颜色 82](#_Toc138011529)

[4.3 着色处理 83](#_Toc138011530)

[4.4 实例程序：彩色三角形 83](#_Toc138011531)

[4.5 摘要(略) 85](#_Toc138011532)

[第五章 灯光 (Lighting) 86](#_Toc138011533)

[目标 86](#_Toc138011534)

[5.1灯光的组成 86](#_Toc138011535)

[5.2材质 87](#_Toc138011536)

[5.3顶点法线 88](#_Toc138011537)

[5.4光源 90](#_Toc138011538)

[5.5实例程序：灯光 93](#_Toc138011539)

[5.6附加实例 95](#_Toc138011540)

[5.7摘要(略) 96](#_Toc138011541)

[第六章 纹理 (Texturing) 97](#_Toc138011542)

[目标 97](#_Toc138011543)

[6.1 纹理坐标 97](#_Toc138011544)

[6.2创建并赋予材质 99](#_Toc138011545)

[6.3过滤器 99](#_Toc138011546)

[6.4 Mipmaps 100](#_Toc138011547)

[6.4.1 Mipmaps过滤器 101](#_Toc138011548)

[6.4.2 Direct3D中使用Mipmaps 101](#_Toc138011549)

[6.5 寻址模式 101](#_Toc138011550)

[6.6实例程序：有纹理的方块 103](#_Toc138011551)

[5. 7摘要(略) 105](#_Toc138011552)

[第七章 混合 (Blending) 106](#_Toc138011553)

[目标 106](#_Toc138011554)

[7.1混合因素 106](#_Toc138011555)

[7.2混合要素 108](#_Toc138011556)

[7.3透明度 108](#_Toc138011557)

[7.3.1Alpha通道 108](#_Toc138011558)

[7.3.2指定Alpha资源 109](#_Toc138011559)

[7.4使用DirectX纹理工具创建Alpha通道 109](#_Toc138011560)

[7.5实例程序：透明度 111](#_Toc138011561)

[7.6摘要(略) 113](#_Toc138011562)

[第八章 模版 (Stenciling) 114](#_Toc138011563)

[目标 115](#_Toc138011564)

[8.1使用模版缓存 115](#_Toc138011565)

[8.1.1请求一个模版缓存 115](#_Toc138011566)

[8.1.2模版测试 115](#_Toc138011567)

[8.1.3控制模版测试 116](#_Toc138011568)

[8.1.3.1模版参考值（Reference Value） 116](#_Toc138011569)

[8.1.3.2模版掩码 116](#_Toc138011570)

[8.1.3.3模版值（Stencil Value） 116](#_Toc138011571)

[8.1.3.4比较运算 116](#_Toc138011572)

[8.1.3更新模版缓存 117](#_Toc138011573)

[8.1.4模版写掩码 118](#_Toc138011574)

[8.2实例程序：镜子 118](#_Toc138011575)

[8.2.1反射数学 118](#_Toc138011576)

[8.2.2镜面实现流程 120](#_Toc138011577)

[8.2.3代码和解释 121](#_Toc138011578)

[8.2.3.1第一部分 121](#_Toc138011579)

[8.2.3.2第二部分 122](#_Toc138011580)

[8.2.3.3第三部分 122](#_Toc138011581)

[8.2.3.4第四部分 123](#_Toc138011582)

[8.2.3.5第五部分 123](#_Toc138011583)

[8.3实例程序：平面阴影 124](#_Toc138011584)

[8.3.1平行光阴影 125](#_Toc138011585)

[8.3.2点光源阴影 126](#_Toc138011586)

[8.3.3阴影矩阵 126](#_Toc138011587)

[8.3.4用模版缓存防止双倍混合 127](#_Toc138011588)

[8.3.5代码和解释 128](#_Toc138011589)

[8.4摘要(略) 129](#_Toc138011590)

[第三部分 实用的Direct3D (Applied Direct3D) 130](#_Toc138011591)

[第九章 字体 (Fonts) 130](#_Toc138011592)

[目标 130](#_Toc138011593)

[9.1 ID3DXFont 130](#_Toc138011594)

[9.1.1创建一个ID3DXFont 130](#_Toc138011595)

[9.1.2绘制文本 131](#_Toc138011596)

[9.1.3计算每秒的渲染帧数 131](#_Toc138011597)

[9.2 CD3DFont 132](#_Toc138011598)

[9.2.1创建一个CD3DFont 132](#_Toc138011599)

[9.2.2绘制文本 133](#_Toc138011600)

[9.2.3 清除 133](#_Toc138011601)

[9.3 D3DXCreateText 133](#_Toc138011602)

[9.4摘要(略) 135](#_Toc138011603)

[第十章 网格模型I (Meshes Part I) 136](#_Toc138011604)

[目标 136](#_Toc138011605)

[10.1 几何信息 136](#_Toc138011606)

[10.2 子集和属性缓存 137](#_Toc138011607)

[10.3 绘制 138](#_Toc138011608)

[10.4 优化 138](#_Toc138011609)

[10.5 属性表 140](#_Toc138011610)

[10.6 邻接信息 141](#_Toc138011611)

[10.7 复制 142](#_Toc138011612)

[10.8 创建一个Mesh（D3DXCreateMeshFVF） 143](#_Toc138011613)

[10.9 实例程序：创建和渲染Mesh 144](#_Toc138011614)

[10.10 摘要(略) 148](#_Toc138011615)

[第十一章 网格模型II (Building a Flexible Camera Class) 149](#_Toc138011616)

[目标 149](#_Toc138011617)

[11.1 ID3DXBuffer 149](#_Toc138011618)

[11.2 X文件 150](#_Toc138011619)

[11.2.1读取X文件 150](#_Toc138011620)

[11.2.2 X文件的材质 151](#_Toc138011621)

[11.2.3 实例程序：X文件 151](#_Toc138011622)

[11.2.4 产生顶点法线 154](#_Toc138011623)

[11.3渐进网格（Progressive Meshes） 155](#_Toc138011624)

[11.3.1 产生一个渐进网格 156](#_Toc138011625)

[11.3.2 顶点品质权重 156](#_Toc138011626)

[11.3.3 ID3DXPMesh方法 157](#_Toc138011627)

[11.3.4实例程序：渐进网格 158](#_Toc138011628)

[11.4 界线容积（Bounding Volumes） 160](#_Toc138011629)

[11.4.1一些新的特殊常量 162](#_Toc138011630)

[11.4.2界线容积类型 162](#_Toc138011631)

[11.4.3实例程序：界线容积 163](#_Toc138011632)

[11.5 摘要(略) 165](#_Toc138011633)

[第十二章 创建灵活的摄像机类 (Building a Flexible Camera Class) 166](#_Toc138011634)

[目标 166](#_Toc138011635)

[12.1 摄像机设计 166](#_Toc138011636)

[12.2 执行详细资料 168](#_Toc138011637)

[12.2.1计算视图矩阵 168](#_Toc138011638)

[12.2.1.1 第一部分：平移 168](#_Toc138011639)

[12.2.1.2 第二部分：旋转 169](#_Toc138011640)

[12.2.1.3 将两部分合并 169](#_Toc138011641)

[12.2.2围绕任意轴旋转 170](#_Toc138011642)

[12.2.3 Pitch、Yaw和Roll 171](#_Toc138011643)

[12.2.4 Walking、Strafing和Flying 173](#_Toc138011644)

[12.3实例程序：摄像机 174](#_Toc138011645)

[12.4 摘要 176](#_Toc138011646)

[第十三章 地形渲染基础 (Basic Terrain Rendering) 177](#_Toc138011647)

[目标 177](#_Toc138011648)

[13.1 Heightmaps（高度图） 177](#_Toc138011649)

[13.1.1 创建高度图（Heightmap） 178](#_Toc138011650)

[13.1.2 读取RAW文件 179](#_Toc138011651)

[13.1.3 访问与修改Heightmap 180](#_Toc138011652)

[13.2 生成地形几何数据 180](#_Toc138011653)

[13.2.1 计算顶点 182](#_Toc138011654)

[13.2.2 计算索引-定义三角形 184](#_Toc138011655)

[13.3 纹理 186](#_Toc138011656)

[13.3.1 程序上的处理方法 186](#_Toc138011657)

[13.4 光照 189](#_Toc138011658)

[13.4.1概览(OVERVIEW) 189](#_Toc138011659)

[13.4.2 计算方格的阴影（Shade） 190](#_Toc138011660)

[13.4.3 地形阴影（Shading） 191](#_Toc138011661)

[13.5 在地形上“行走” 192](#_Toc138011662)

[13.6 例子程序: Terrain 194](#_Toc138011663)

[13.7 一些改进 196](#_Toc138011664)

[13.8 摘要 197](#_Toc138011665)

[第十四章 粒子系统 (Particle Systems) 198](#_Toc138011666)

[目标： 198](#_Toc138011667)

[14.1 粒子和点精灵（Point Sprite） 198](#_Toc138011668)

[14.1.1 结构的格式 198](#_Toc138011669)

[14.1.2点精灵（Point Sprite）渲染状态 199](#_Toc138011670)

[14.1.3 粒子和他们的属性 200](#_Toc138011671)

[14.2 粒子系统的组成 201](#_Toc138011672)

[14.2.1 绘制粒子系统 205](#_Toc138011673)

[14.2.2 随机 209](#_Toc138011674)

[14.3 具体的粒子系统：雪、火、粒子枪 209](#_Toc138011675)

[14.3.1 例子程序：雪 210](#_Toc138011676)

[14.3.2 例子程序：火 212](#_Toc138011677)

[14.3.3 例子程序：粒子枪 214](#_Toc138011678)

[14.4 摘要 216](#_Toc138011679)

[第十五章 选取 (Picking) 217](#_Toc138011680)

[概览(OVERVIEW) 217](#_Toc138011681)

[目标 218](#_Toc138011682)

[15.1 屏幕到投影窗口的转换 218](#_Toc138011683)

[15.2 计算射线 220](#_Toc138011684)

[15.3 变换射线 220](#_Toc138011685)

[15.4 射线－对象 交点 221](#_Toc138011686)

[15.5 例子程序：选取 223](#_Toc138011687)

[15.6 摘要 224](#_Toc138011688)

[第四部分 着色器和特效 (Shaders and Effects) 225](#_Toc138011689)

[第十六章 高级着色器语言入门 (Introduction to the High-Level Shading Language) 225](#_Toc138011690)

[目标 225](#_Toc138011691)

[16.1 编写HLSL 着色器 226](#_Toc138011692)

[16.1.1 全局变量 227](#_Toc138011693)

[16.1.2 输入和输出结构 227](#_Toc138011694)

[16.1.3 函数的入口点 228](#_Toc138011695)

[16.2 编译HLSL 着色器 229](#_Toc138011696)

[16.2.1 常量表 229](#_Toc138011697)

[16.3 变量类型 233](#_Toc138011698)

[16.3.2 向量类型 234](#_Toc138011699)

[16.3.3 矩阵类型 235](#_Toc138011700)

[16.3.4 数组 236](#_Toc138011701)

[16.3.5 结构 236](#_Toc138011702)

[16.3.6 typedef关键字 237](#_Toc138011703)

[16.4关键字、语句和强制转换 238](#_Toc138011704)

[16.4.1 关键字 238](#_Toc138011705)

[16.4.2 基本程序流程 238](#_Toc138011706)

[16.4.3 强制转换（casting） 239](#_Toc138011707)

[16.5 操作符 239](#_Toc138011708)

[16.6 用户定义函数 241](#_Toc138011709)

[16.7内建函数 242](#_Toc138011710)

[16.8 摘要 244](#_Toc138011711)

[第十七章 顶点着色器入门 (Introduction to Vertex Shaders) 245](#_Toc138011712)

[概览 245](#_Toc138011713)

[目标 246](#_Toc138011714)

[17.1顶点声明 246](#_Toc138011715)

[17.1.1 描述顶点声明 246](#_Toc138011716)

[17.1.2 创建顶点声明 248](#_Toc138011717)

[17.1.3 使用一个顶点声明 249](#_Toc138011718)

[17.2顶点数据用途 249](#_Toc138011719)

[17.3使用顶点着色器的步骤 250](#_Toc138011720)

[17.3.1 编写并编译顶点着色器 251](#_Toc138011721)

[17.3.2 创建顶点着色器 251](#_Toc138011722)

[17.3.3 建立顶点着色器 251](#_Toc138011723)

[17.3.4 销毁顶点着色器 252](#_Toc138011724)

[17.4样例应用程序：散射光照 252](#_Toc138011725)

[17.5 卡通渲染 257](#_Toc138011726)

[17.5.1 卡通着色 258](#_Toc138011727)

[17.5.2 卡通着色的顶点着色器代码 259](#_Toc138011728)

[17.5.3轮廓勾勒 260](#_Toc138011729)

[17.5.4 轮廓边顶点着色器代码 263](#_Toc138011730)

[17.6 摘要 265](#_Toc138011731)

[第十八章 像素着色器入门 (Introduction to Pixel Shaders) 266](#_Toc138011732)

[目标 266](#_Toc138011733)

[18.1多纹理化概览 266](#_Toc138011734)

[18.1.1 允许多个纹理 268](#_Toc138011735)

[18.1.2 多纹理坐标 269](#_Toc138011736)

[18.2像素着色器输入和输出 269](#_Toc138011737)

[18.3使用像素着色器的步骤 270](#_Toc138011738)

[18.3.1 编写并编译像素着色器 270](#_Toc138011739)

[18.3.2 创建像素着色器 271](#_Toc138011740)

[18.3.3 建立像素着色器 271](#_Toc138011741)

[18.3.4 销毁像素着色器 271](#_Toc138011742)

[18.4 HLSL采样器对象 272](#_Toc138011743)

[18.5 例子程序：Multitexturing in a Pixel Shader 273](#_Toc138011744)

[18.6 摘要 280](#_Toc138011745)

[第十九章 效果架构 (The Effects Framework) 281](#_Toc138011746)

[概览 281](#_Toc138011747)

[目标 281](#_Toc138011748)

[19.1 技术与传递（Techniques and Passes） 281](#_Toc138011749)

[19.2 更多HLSL内置对象（ More HLSL Intrinsic Objects） 282](#_Toc138011750)

[19.2.1 纹理对象 283](#_Toc138011751)

[19.2.2 采样器对象与采样器状态 283](#_Toc138011752)

[19.2.3 顶点与像素着色器对象（Vertex and Pixel Shader Objects） 283](#_Toc138011753)

[19.2.4 字符串 284](#_Toc138011754)

[19.2.5 注解 (Annotations) 285](#_Toc138011755)

[19.3 效果文件的设备状态（ Device States in an Effect File） 285](#_Toc138011756)

[19.4 创建效果 286](#_Toc138011757)

[19.5 设置系数（Setting Constants） 287](#_Toc138011758)

[19.6 使用效果 289](#_Toc138011759)

[19.6.1 获得效果句柄（ Obtaining a Handle to an Effect） 289](#_Toc138011760)

[19.6.2 激活一个效果（ Activating an Effect） 289](#_Toc138011761)

[19.6.3 启动效果 290](#_Toc138011762)

[19.6.4 设置当前的渲染传递（Setting the Current Rendering Pass） 290](#_Toc138011763)

[19.6.5 结束效果（Ending an Effect） 290](#_Toc138011764)

[19.6.6 例子 290](#_Toc138011765)

[19.7 例子程序: Lighting and Texturing in an Effect File 291](#_Toc138011766)

[19.8例子程序: Fog Effect 296](#_Toc138011767)

[19.9例子程序: Cartoon Effect 298](#_Toc138011768)

[19.10 效果编辑（EffectEdit） 299](#_Toc138011769)

[19.11摘要 300](#_Toc138011770)

[略 300](#_Toc138011771)

[相关文章： 301](#_Toc138011772)

[AGP内存 301](#_Toc138011773)

[Direct3D中实现图元的鼠标拾取 303](#_Toc138011774)

[1、什么是拾取，拾取能做什么？ 303](#_Toc138011775)

[2、拾取操作的步骤和实现 303](#_Toc138011776)

[3、结束及声明 309](#_Toc138011777)

[4、参考文献 310](#_Toc138011778)

[克莱姆（Cramer）法则 311](#_Toc138011779)

[一、线性方程组 311](#_Toc138011780)

[三、齐次线性方程组 313](#_Toc138011781)

[四、例子 314](#_Toc138011782)

# 第一部分 必备的数学知识 (Mathematical Prerequisites)

在这最开始的一部分中我们将介绍本书所要用到的数学知识。我们讨论的主题是向量，矩阵和相应的变换，当然还有一些有关面和线的内容。最开始阅读时这部分是可选的。

本教程对这些知识的讨论是很有限的，因此对于不同数学知识背景的读者来说都容易阅读。对于想了解更多更全的这方面信息的读者，请查看有关线性代数的书籍。当然已经学习过线性代数的读者也可将它作为有必要的复习内容来阅读。（这里推荐你看看《线性代数与空间解析几何》）

除此之外，我们还将展示D3DX类中相关的数学模型和执行特殊变换的函数。

## 目标

* 学习向量以及它们的3D计算机图形程序
* 学习矩阵以及学会使用它们来变换3D图形
* 学习怎样模拟面和线以及它们的3D图形程序
* 熟悉用于3D数学运算的D3DX库中包含的类和程序的子集

## 三维空间中的向量

几何学中，我们用有向线段表示向量，如图1。向量的两个属性是他的长度和他的顶点所指的方向。因此，可以用向量来模拟既有大小又有方向的物理模型。例如，以后我们要实现的粒子系统。我们用向量来模拟粒子的速度和加速度。在3D计算机图形学中我们用向量不仅仅模拟方向。例如我们常常想知道光线的照射方向，以及在3D世界中的摄象机。向量为在3维空间中表示方向的提供了方便。

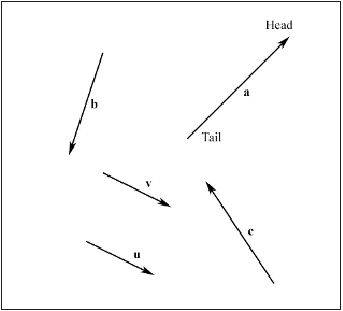


图1

向量与位置无关。有同样长度和方向的两个向量是相等的，即使他们在不同的位置。观察彼此平行的两个向量，例如在图1中u和v是相等的。

我们继续学习左手坐标系。图2显示的是左手坐标系和右手坐标系。两者不同的是Z轴的方向。在左手坐标系中Z轴是向书的里面去的，而右手坐标系是向书的外边去的。

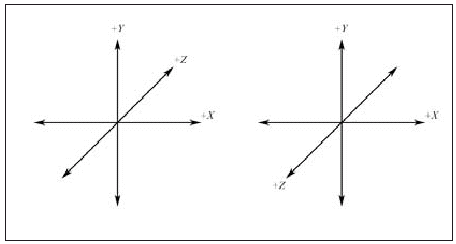


图2

因为向量的位置不能改变它的性质，我们可以把所有向量平移使他们的尾部和坐标系的原点重合。因此，当一个向量在标准位置我们能通过头点来描述向量。图3显示的是图1中的向量在标准位置的样子。

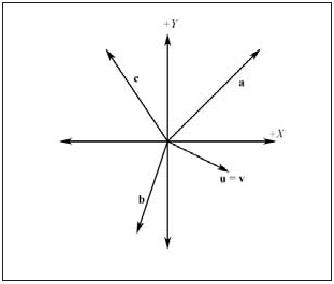


图3

我们通常用小写字母表示一个向量，但有时也用大写字母。如2、3和4维向量分别是：

**u** = (*ux*, *uy*),

**N** = (*Nx*, *Ny*, *Nz*),

**c** = (*cx*, *cy*, *cz*, *cw*)。

我们现在介绍4个特殊的3D向量，就象图4显示的。首先是都由含有0的零向量；它被表示成加粗的**0** = (0, 0, 0)。接下来3个特殊的向量标准基向量。它们被叫做**i**, **j**和**k**向量，分别沿着坐标系的x轴,y轴和z轴，并且有1的单位长：**i** = (1, 0, 0), **j** = (0, 1, 0), and **k** = (0, 0, 1)。

注意：只有1个单位长度的向量叫做单位向量（模长为1的向量）。

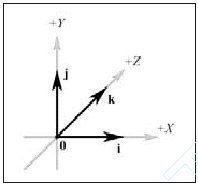


图4

在D3DX库中，我们能用D3DXVECTOR3类表示3维空间中的向量。它的定义是：

|  |
| --- |
| typedef struct D3DXVECTOR3 : public D3DVECTOR  {  public:  D3DXVECTOR3() {};  D3DXVECTOR3( CONST FLOAT \* );  D3DXVECTOR3( CONST D3DVECTOR& );  D3DXVECTOR3( CONST D3DXFLOAT16 \* );  D3DXVECTOR3( FLOAT x, FLOAT y, FLOAT z );  // casting  operator FLOAT\* ();  operator CONST FLOAT\* () const;  // assignment operators  D3DXVECTOR3& operator += ( CONST D3DXVECTOR3& );  D3DXVECTOR3& operator -= ( CONST D3DXVECTOR3& );  D3DXVECTOR3& operator \*= ( FLOAT );  D3DXVECTOR3& operator /= ( FLOAT );  // unary operators  D3DXVECTOR3 operator + () const;  D3DXVECTOR3 operator - () const;  // binary operators  D3DXVECTOR3 operator + ( CONST D3DXVECTOR3& ) const;  D3DXVECTOR3 operator - ( CONST D3DXVECTOR3& ) const;  D3DXVECTOR3 operator \* ( FLOAT ) const;  D3DXVECTOR3 operator / ( FLOAT ) const;  friend D3DXVECTOR3 operator \* ( FLOAT, CONST struct D3DXVECTOR3& );  BOOL operator == ( CONST D3DXVECTOR3& ) const;  BOOL operator != ( CONST D3DXVECTOR3& ) const;  } D3DXVECTOR3, \*LPD3DXVECTOR3; |

注意D3DXVECTOR3是从D3DVECTOR继承的。它的定义是：

|  |
| --- |
| typedef struct \_D3DVECTOR {  float x, y, z;  } D3DVECTOR; |

向量有它们自己的算法，就象你在D3DXVECTOR3定义中看到的数学运算。现在你不需要知道它们怎么使用。以后介绍这些向量运算以及一些有用的函数和关于向量的，重要的详细资料。

**注意**：在3D图形程序中，虽然我们主要关心3D向量，但有时也会用到2D和4D向量。在D3DX库中提供了D3DXVECTOR2和D3DXVECTOR4类来分别表现2D和4D向量。不同维数的向量有着和3D向量一样的性质，也就是它们描述大小和方向，仅仅是在不同的维数中。所有这些向量的数学运算对于不同维数向量都有效只是有一个除外，就是向量积。这些运算我们可通过论述3D向量扩展到2D, 4D甚至n维向量。

## 向量相等

几何学上，有同样方向和长度的两个向量相等。数学上，我们说有同样维数和分量的向量相等。例如：如果*ux* = *vx*, *uy* = *vy*, 且 *uz* = *vz*.那么(*ux*, *uy*, *uz*) = (*vx*, *vy*, *vz*)。在代码中我们能够用“==”判断两个向量相等。

|  |
| --- |
| D3DXVECTOR u(1.0f, 0.0f, 1.0f);  D3DXVECTOR v(0.0f, 1.0f, 0.0f);  if( u == v ) return true; |

同样的，我们也能用“！=”判断两个向量不相等。

|  |
| --- |
| if( u != v ) return true; |

注意：当比较浮点数时，必须注意。因为浮点数不是精确的，我们认为相等的两个浮点数是有细微差别的；因此，我们测试它们近似相等。我们定义一个常数EPSILON，把它当作非常小的“buffer”。假如两个数和EPSILON相差很小我们说它们近似相等。换句话说，EPSILON让浮点数有一定的精度。接下来的实例函数是怎样用EPSILON比较两个浮点数相等。

|  |
| --- |
| bool Equals(float lhs, float rhs)  {  // if lhs == rhs their difference should be zero  return fabs(lhs - rhs) < EPSILON ? true : false;  } |

当我们用D3DXVECTOR3类时不必担心，因为它已经帮我们处理了，但是在一般情况下适当注意比较两个浮点数是很重要的。

## 计算向量大小（向量的模）

几何学上，向量的大小是有向线段的长度。知道向量的分量，利用下面的公式我们就能计算出向量的大小。



‖**u**‖表示向量**u**的长度。例如：计算向量**u** = (1, 2, 3)和**v** = (1, 1)的大小。

根据公式（1），我们得到：





我们利用D3DX库中下面的函数便能计算向量的大小。

FLOAT D3DXVec3Length( // Returns the magnitude.

CONST D3DXVECTOR3\* pV // The vector to compute the length of.

);

|  |
| --- |
| D3DXVECTOR3 v(1.0f, 2.0f, 3.0f);  float magnitude = D3DXVec3Length( &v ); // = sqrt(14) |

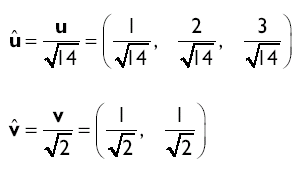
## 标准化向量

标准化向量是让向量的大小等于1，即被叫作单位向量。我们能利用向量大小以及各个分量把一个向量标准化，就象这样：



我们这样表示单位向量**û**。如：标准化向量**u** = (1, 2, 3) 和 **v** = (1, 1)。

解答方法：根据(2)和(3)我们得到‖**u**‖=√14 和 ‖**v**‖=√2,因此：



我们利用D3DX库中下面的函数能标准化向量。

|  |
| --- |
| D3DXVECTOR3 \*D3DXVec3Normalize(  D3DXVECTOR3\* pOut, // Result.  CONST D3DXVECTOR3\* pV // The vector to normalize.  ); |

注意：这个函数返回一个指针，因此它可以作为一个参数传递给另一个函数。大数情况下，除非另作说明，D3DX数学函数返回的结果是一个指针。但不是所有函数都这样。

## 向量相加

我们能够通过分别把两个向量的各个分量相加得到向量之和，注意在相加之前必须保证它们有相同的维数。



图5显示的是几何学上的向量相加。

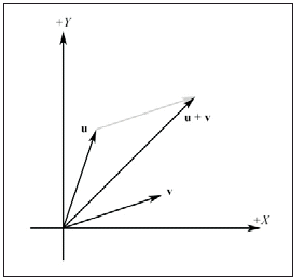


图5

两个向量相加的代码，我们使用重载的加法操作符：

|  |
| --- |
| D3DXVECTOR3 u(2.0f, 0.0f, 1.0f);  D3DXVECTOR3 v(0.0f, -1.0f, 5.0f);  // (2.0 + 0.0, 0.0 + (-1.0), 1.0 + 5.0)  D3DXVECTOR3 sum = u + v; // = (2.0f, -1.0f, 6.0f) |

## 向量相减

和加法类似，通过分别把两个向量的各个分量相减得到向量之差。再次重声两个向量必须是相同维数。



图6显示的是几何学上的向量相减。

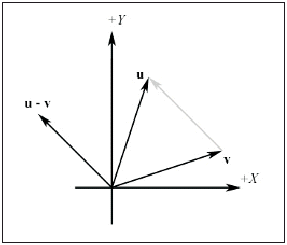


图6

两个向量相减的代码，我们使用重载的减法操作符：

|  |
| --- |
| D3DXVECTOR3 u(2.0f, 0.0f, 1.0f);  D3DXVECTOR3 v(0.0f, -1.0f, 5.0f);  D3DXVECTOR3 difference = u - v; // = (2.0f, 1.0f, -4.0f) |

图6显示，向量减法得到一个从v向量终点到u向量终点的向量。假如我们解释**u**和**v**的分量，我们能用向量相减找到从一个点到另一个点的向量。这是非常方便的操作，因为我们常常想找到从一个点到另一个点的方向向量。

## 标量与向量的乘积

我们能用一个标量与向量相乘，就象名字暗示的一样，向量按比例变化。这种运算不会改变向量的方向，除非标量是负数，这种情况向量方向相反。



D3DXVECTOR3类提供了向量与标量乘法的操作符。

|  |
| --- |
| D3DXVECTOR3 u(1.0f, 1.0f, -1.0f);  D3DXVECTOR3 scaledVec = u \* 10.0f; // = (10.0f, 10.0f, -10.0f) |

## 点积

数学上定义点积是两个向量的乘积。按下面等式计算：



上面的等式不能很明显的体现几何上的意义。利用余弦定律，我们能够发现它们的关系。

**u · v =|u||v|**cosθ，表示两个向量的点积是它们的模和夹角的余弦之积。因此，如果**u** 和**v**都是单位向量，那么**u · v**就是它们夹角的余弦。

一些点积中有用的特性

■　假如**u · v** = 0，那么**u**⊥**v**。

■　假如**u · v** > 0，那么两个向量的角度θ小于90度。

■　假如**u · v** < 0，那么两个向量的角度θ大于90度。

我们使用下面的D3DX函数计算两个向量的点积：

FLOAT D3DXVec3Dot( // Returns the result.

CONST D3DXVECTOR3\* pV1, // Left sided operand.

CONST D3DXVECTOR3\* pV2 // Right sided operand.

);

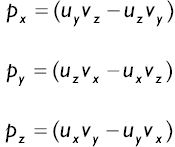
|  |
| --- |
| D3DXVECTOR3 u(1.0f, -1.0f, 0.0f);  D3DXVECTOR3 v(3.0f, 2.0f, 1.0f);  // 1.0\*3.0 + -1.0\*2.0 + 0.0\*1.0  // = 3.0 + -2.0  float dot = D3DXVec3Dot( &u, &v ); // = 1.0 |

## 叉积

第二种乘法在向量数学中叫叉积。不象点积，结果值是一个标量，叉积的结果值是另一个向量。通过把两个向量**u**和**v**相乘得到另一的向量**p**，向量**p**垂直于**u**和**v**。也就是说向量**p**垂直于**u**并且垂直于**u**。

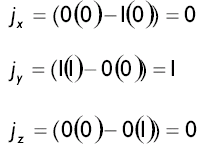
十字相乘就象这样计算：





如：发现**j** = **k**×**i** = (0, 0, 1)×(1, 0, 0) 并且**j**同时垂直于**k**和**i**.

解答：



因此，**j** = (0, 1, 0)。回忆一下上节的标题“叉积”，是说如果**u · v** = 0，那么**u**⊥**v**。。同样的如果**j** · **k** = 0并且**j** · **i** = 0那么我们便能知道**j**既垂直于**k**又垂直于**i**的。

我们使用下面的D3DX函数计算两个向量的叉积：

|  |
| --- |
| D3DXVECTOR3 \*D3DXVec3Cross(  D3DXVECTOR3\* pOut, // Result.  CONST D3DXVECTOR3\* pV1, // Left sided operand.  CONST D3DXVECTOR3\* pV2 // Right sided operand.  ); |

从图7很明显，向量-**p**与**u**和**v**也都相互垂直。我们执行叉积的命令，确定得到的的结果不管是**p**或者-**p**。换句话说，**u×v** = -(**v**×**u**)。这说明叉积是不可交换的。你能通过左手法则确定叉积返回的向量。按照第一个向量指向第二个向量弯曲你的左手，这时拇指所指的方向就是向量所指的方向。

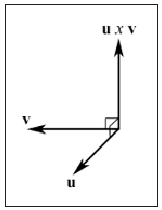
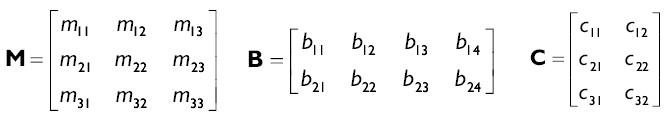


图7

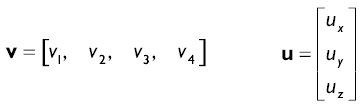
## 矩阵

在这一部分我们关注的焦点是数学中的矩阵。它们在3D图形学中的应用将在下一部分讲解。

一个*m×n*的矩阵是由m行和n列的数字组成的矩阵列。行和列的数字就是这个矩阵的维数。我们通过写在下方的数字识别矩阵清单，数字中的第一个表示行第二个表示列。例如下边的**M**是3×3矩阵，**B**是2×4矩阵, **C**是3×2矩阵。



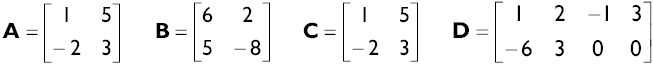
我们使用加粗的大写字母表示矩阵。有时一个矩阵只包含一行或者一列。我们用行矩阵和列矩阵这个特殊的名称来称呼。例如下边就是行和列矩阵：



当使用行或列矩阵时，我们只用一个下标，有时我们还用字母表示。

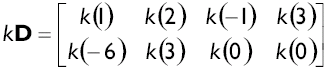
## 相等、数乘矩阵以及相加

这部分我们将用到下边4个矩阵：

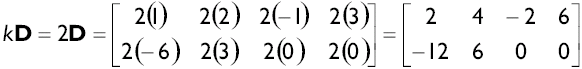


■假如两个矩阵维数和成员都相同那么它们就相等。例如，**A = C** 因为**A**和**C**有同样的维数并且他们的成员都相等。**A**≠**B**同时**A**≠**D**因为他们的成员或者维数是不相同的。

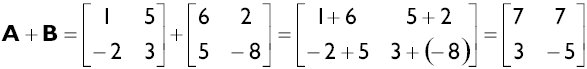
■我们能通过标量与矩阵的每个成员相乘得到标量与矩阵相乘。如矩阵D与k相乘：



假如*k* = 2,那么：



■当两个矩阵的维数相同时才能把它们相加。和是把两个矩阵相应的成员相加得到。如：



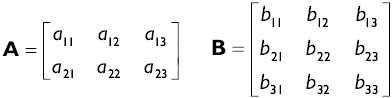
■矩阵有加法当然也就有减法，前提是有相同的维数。矩阵减法如图所示：



## 乘法

矩阵相乘在3D计算机图形学中是非常重要的运算。通过矩阵相乘，我们能变换向量并且，将不同向量转换到一起。变换是[下一节](#_单位矩阵)的内容。

为了得到矩阵之积**AB**，**A**的列数必须等于**B**的行数。假如这个条件不满足，就不能相乘。考虑下边两个矩阵，**A** 和 **B**，分别是2×3 和 3×3，如：

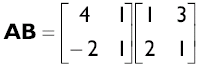


我们看乘积**AB**是可以计算的，因为**A**的列数等于**B**的行数。注意乘积**BA**,它们是不能计算的，因为**B**的列数不等于**A**的行数。由此说明：一般情况下矩阵乘法不满足乘法交换律（也就是, **AB**≠**BA**）。我们说“一般不可交换”因为有一些矩阵乘法的实例还是可以的。

知道了矩阵乘法的计算方法，现在我们就能给出精确的定义：假如**A**是一个*m*×*n*的矩阵，**B**是一个*n*×*p*的矩阵，那么它们之积**AB**可计算并且是一个*m*×*p* 的矩阵**C**, **C**的成员*ij* 等于**A**的第i个与**B**的第j个相乘：



例如，求解：

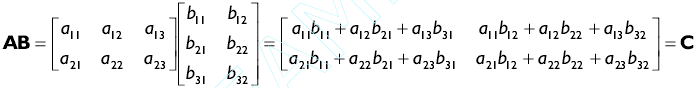


我们检查知道乘法是可计算的，因为**A**的列数等于**B**的行数。也知道计算的结果是一个2×2的矩阵。根据公式（4），我们得到：



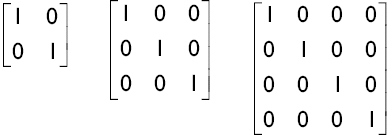
作为练习，检查**AB**≠**BA。**

更一般的例子：



## 单位矩阵

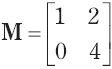
有一种特殊矩阵叫做单位矩阵。单位矩阵是除了对角（左上到右下）以外所有成员都是0，对角都是1的方矩阵。例如，下边是2×2, 3×3, 和 4×4的单位矩阵：



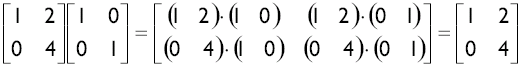
单位矩阵有如下特性：

MI = IM=M

即，用单位矩阵乘以矩阵不会改变矩阵。此外，这是一个特例：用单位矩阵进行乘法运算满足乘法交换律。单位矩阵可以看作矩阵运算中的数字“1”。



例如：验证2×2矩阵**M**与单位矩阵相乘得到的结果是**M**。

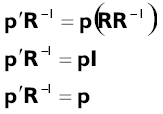


## 逆转

下面列举了关于逆矩阵的重要信息。

* 只有正方形的矩阵（方阵）才能求逆，因此当我们说矩阵求逆，那么它就是方矩阵。
* *n*×*n*矩阵**M**的逆矩阵是一个*n*×*n*矩阵表示为**M**–1
* 不是每个方矩阵都有逆矩阵
* 矩阵和他的逆矩阵相乘得到一个单位矩阵：**M M**–1 = **M**–1**M** = **I。**注意当我们进行这样的操作时矩阵是可交换的。

逆矩阵用来解决与其他矩阵相等是非常有用的。例如，考虑等式**p**’= **pR** 并且假设我们知道**p**’和**R**想求**p**。首先找到**R**–1，一旦求得**R**–1，我们便能求出**p**，就象这样：



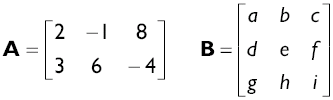
求逆矩阵的方法已经超出了本书的范围，但是这在任何一本线性代数书上都有讲解。在“[基本变换](#_基本变换)”一节我们给出一个特定矩阵的逆矩阵。在“[D3DX 矩阵](#_D3DX_矩阵)”部分我们将学习一个为我们求逆矩阵的D3DX函数。

我们介绍几个有用的推论：(**AB**) –1 = **B**–1 **A**–1。这个性质前提是：假定**A**和**B**都能求逆并且它们都是有相同维数的方矩阵。

## 矩阵的转置

矩阵的转置是相互交换矩阵的行和列。因而，*m*×*n*的矩阵的转置是一个*n*×*m*的矩阵。我们把矩阵**M**的转置记作**M***T*。

例如：求下面两个矩阵的转置：



重声一下，转置是交换矩阵的行和列。

因此：



## D3DX 矩阵

当设计Direct3D应用程序时，使用4×4矩阵和1×4行矩阵（向量）是有代表性的。注意使用这两种矩阵意味着可以进行下列定义的矩阵乘法。

* + **向量-矩阵乘法**。即，假如1×4的单行矩阵**V，**和4×4的矩阵**T**，那么积**VT**可计算并且返回的结果是一个1×4的单行矩阵（向量）。
* **矩阵-矩阵乘法**。即，假如4×4的矩阵**T，**和4×4的矩阵**R**，那么积**TR**和**RT**可计算并且两者返回的结果都是一个4×4的矩阵。注意因为矩阵乘法不满足交换律所以**TR**和**RT**不一定相等。

在D3DX中表示1×4的行矩阵（向量），我们用D3DXVECTOR3和D3DXVECTOR4向量类。当然D3DXVECTOR3只有3个成员，不是4个。然而，第4个成员缺省是1或0（在下一部分有更多信息）。

在D3DX中表示4×4的矩阵，我们用D3DXMATRIX类，定义如下：

|  |
| --- |
| typedef struct D3DXMATRIX : public D3DMATRIX {  public:  D3DXMATRIX() {};  D3DXMATRIX( CONST FLOAT \* );  D3DXMATRIX( CONST D3DMATRIX& );  D3DXMATRIX( FLOAT \_11, FLOAT \_12, FLOAT \_13, FLOAT \_14,  FLOAT \_21, FLOAT \_22, FLOAT \_23, FLOAT \_24,  FLOAT \_31, FLOAT \_32, FLOAT \_33, FLOAT \_34,  FLOAT \_41, FLOAT \_42, FLOAT \_43, FLOAT \_44 );  // access grants  FLOAT& operator () ( UINT Row, UINT Col );  FLOAT operator () ( UINT Row, UINT Col ) const;  // casting operators  operator FLOAT\* ();  operator CONST FLOAT\* () const;  // assignment operators  D3DXMATRIX& operator \*= ( CONST D3DXMATRIX& );  D3DXMATRIX& operator += ( CONST D3DXMATRIX& );  D3DXMATRIX& operator -= ( CONST D3DXMATRIX& );  D3DXMATRIX& operator \*= ( FLOAT );  D3DXMATRIX& operator /= ( FLOAT );  // unary operators  D3DXMATRIX operator + () const;  D3DXMATRIX operator - () const;  // binary operators  D3DXMATRIX operator \* ( CONST D3DXMATRIX& ) const;  D3DXMATRIX operator + ( CONST D3DXMATRIX& ) const;  D3DXMATRIX operator - ( CONST D3DXMATRIX& ) const;  D3DXMATRIX operator \* ( FLOAT ) const;  D3DXMATRIX operator / ( FLOAT ) const;  friend D3DXMATRIX operator \* ( FLOAT, CONST D3DXMATRIX& );  BOOL operator == ( CONST D3DXMATRIX& ) const;  BOOL operator != ( CONST D3DXMATRIX& ) const;  } D3DXMATRIX, \*LPD3DXMATRIX; |

D3DXMATRIX类是从单数结构D3DMATRIX继承的复数形式。D3DMATRIX的定义是：

|  |
| --- |
| typedef struct \_D3DMATRIX {  union {  struct {  float \_11, \_12, \_13, \_14;  float \_21, \_22, \_23, \_24;  float \_31, \_32, \_33, \_34;  float \_41, \_42, \_43, \_44;  };  float m[4][4];  };  } D3DMATRIX; |

观察D3DXMATRIX类发现有很多有用的运算符，比如对矩阵检测相等，矩阵相加和矩阵相减，标量与矩阵相乘，铸造（casting），以及非常重要的两个D3DXMATRIXs相乘。因为矩阵相乘是非常重要的，我们给出一段实例代码：

|  |
| --- |
| D3DXMATRIX A(…); // initialize A  D3DXMATRIX B(…); // initialize B  D3DXMATRIX C = A \* B; // C = AB |

D3DXMATRIX类另一个重要的运算符是小括号，它允许我们非常方便的为矩阵成员赋值。注意当使用小括号时我们的下标就象C语言数组下标一样是从0开始的。例如，为一个矩阵的*ij* = 11 赋值，我们写成：

|  |
| --- |
| D3DXMATRIX M;  M(0, 0) = 5.0f; // Set entry ij = 11 to 5.0f. |

D3DX库也提供下列有用的函数：将D3DXMATRIX转化为单位矩阵，转置D3DXMATRIX矩阵以及求逆矩阵。

D3DXMATRIX \*D3DXMatrixIdentity(

D3DXMATRIX \*pout // 将矩阵转换为单位矩阵

);

|  |
| --- |
| D3DXMATRIX M;  D3DXMatrixIdentity( &M ); // M = 单位矩阵 |

D3DXMATRIX \*D3DXMatrixTranspose(

D3DXMATRIX \*pOut, // 输出的转置矩阵

CONST D3DXMATRIX \*pM // 原矩阵

);

|  |
| --- |
| D3DXMATRIX A(...); // 初始化矩阵A  D3DXMATRIX B;  D3DXMatrixTranspose( &B, &A ); // B = 输出的转置矩阵 |

假如我们将不能求逆的矩阵用求逆函数，那么函数将会返回null.同样的，这本书我们忽视第二个参数，并且总是把它设置为0。

D3DXMATRIX \*D3DXMatrixInverse(

D3DXMATRIX \*pOut, // 输出的逆矩阵

FLOAT \*pDeterminant, // 除非是必需的，一般设为0

CONST D3DXMATRIX \*pM // 原矩阵

);

|  |
| --- |
| D3DXMATRIX A(...); // 初始化矩阵  D3DXMATRIX B;  D3DXMatrixInverse( &B, 0, &A ); // B = A的逆矩阵 |

## 基本变换

当用Direct3D编程时，我们使用4×4矩阵来进行矩阵变换。用它的原因是：我们设置一个4×4矩阵**X**是为了更精确的描述矩阵变换。同样我们设置一个相匹配的点或者把向量的分量放置到一个1×4的行矩阵**V**中。**VX**的乘积返回一个新的向量**V**’。例如：让**X**沿着x轴平移10个单位同时**V** = [2, 6, –3, 1]，乘积**VX** = **V**’= [12, 6, –3, 1]。

有一些东西需要阐明。我们使用4×4矩阵是因为这样的大小能表现我们需要的所有变换。最初看来一个3×3的好象更适合3D。然而这里有很多种我们喜欢用的变换是不能用一个3×3的矩阵来表示的，比如平移、投影、反射。我们使用向量-矩阵相乘来工作，因此我们至少要通过一个矩阵乘法来完成相应的变化。增大到4×4的矩阵后，它允许我们用一个矩阵描述更多的变换，并且向量-矩阵乘法是可行的。

我们说过把一个相匹配的点或者一个向量的成员放置到一个1×4的行矩阵中。但是点和向量是3D的！为什么我们要用一个1×4的行矩阵呢？我们必需把3D点/向量增大为4D的单行矩阵，是为了符合向量与矩阵的乘法定义，而1×3的单行矩阵和4×4的矩阵相乘是不允许的。

那么，我们怎么使用第四个成员（我们用*w*来表示）呢？当我们把一个点放置到一个1×4的行矩阵中时，我们设置*w*为1。允许对点进行适当的平移。因为向量和位置无关，所以向量的平移没有被定义，如果试图这样做会返回一个无意义的向量。为了防止对向量进行平移，当在把一个向量放置到一个1×4行矩阵中时我们把*w*设置为0。例如：

把点**p** = (*p*1, *p*2, *p*3)放置到一个单行矩阵中就象这样：

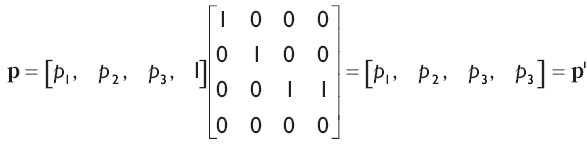
**[*p*1, *p*2, *p*3, 1]**，

同样把向量**v** = (*v*1, *v*2, *v*3) 放置到一个单行矩阵中就象这样：

**[*v*1, *v*2, *v*3, 0]**。

**注意：**我们设置*w* = 1是为了让点可以被恰当的移动，同样我们设置*w* = 0是为了防止向量被平移。当我们检查矩阵实际平移时这是一个非常清晰的模型。

有时一个矩阵变换时我们改变向量成员*w*的值，即*w≠*0 且 *w*≠1。考虑下边例子：



因为*p*3≠0 且 *p*3≠1。

我们注意*w* =*p*3。当*w≠*0 且 *w*≠1时，我们说我们有一个向量在**同类空间**中，与3维空间中的向量是相对的。我们能通过把向量的每个分量与*w*相除将**同类空间**中的向量映射到3维空间中来。例如把同类空间中向量(*x, y, z, w*) 映射到3维空间中的向量**x**，我们这样做：



在3D图形程序设计中，从齐次坐标空间映射到3D空间时使用投影透视。

## 矩阵平移

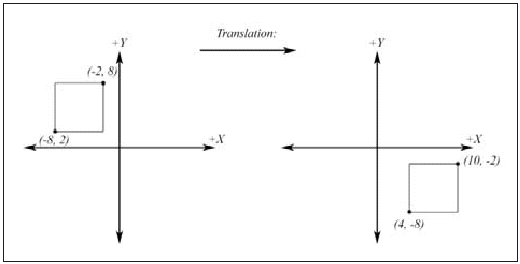
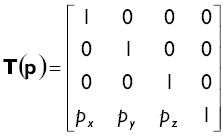


图8

我们能通过与下面的矩阵相乘把向量(*x, y, z,* 1)沿x轴移动*px*个单位，沿y轴移动*py* 个单位，沿z轴移动*pz*个单位：

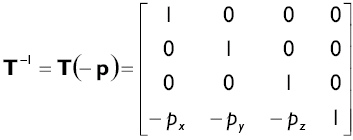


将矩阵平移的D3DX函数是：

|  |
| --- |
| D3DXMATRIX \*D3DXMatrixTranslation(  D3DXMATRIX\* pOut, // 返回平移后的矩阵.  FLOAT x, // x轴移动的单位  FLOAT y, // y轴移动的单位  FLOAT z // z轴移动的单位  ); |

练习：让**T**(**p**)做为一个平移变换矩阵，**v** = [*v*1, *v*2, *v*3, 0]是也个任意向量。验证**vT**(**p**) = **v**（即，假如*w* = 0，验证通过平移不会改变向量）。

平移矩阵求逆只需要简单的将向量**p**取反即可：



## 矩阵旋转

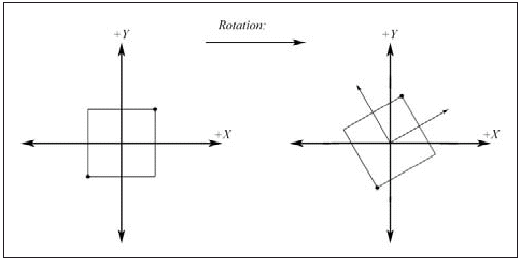
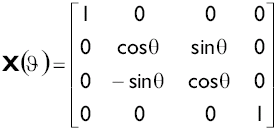


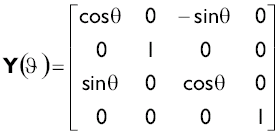
图9

我们能用下面的矩阵把一个向量围绕x,y 和z轴旋转δ弧度。注意：当我们俯视绕轴原点时，角度是指顺时针方向的角度。



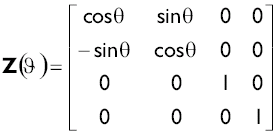
将矩阵饶着x轴旋转的D3DX函数是：

|  |
| --- |
| D3DXMATRIX \*D3DXMatrix**RotationX**(  D3DXMATRIX\* pOut, // 返回旋转后的矩阵  FLOAT Angle // Angle是旋转的弧度  ); |



将矩阵饶着y轴旋转的D3DX函数是：

|  |
| --- |
| D3DXMATRIX \*D3DXMatrix**RotationY**(  D3DXMATRIX\* pOut, // 返回旋转后的矩阵  FLOAT Angle // Angle是旋转的弧度  ); |



将矩阵饶着z轴旋转的D3DX函数是：

|  |
| --- |
| D3DXMATRIX \*D3DXMatrix**RotationZ**(  D3DXMATRIX\* pOut, // 返回旋转后的矩阵  FLOAT Angle // Angle是旋转的弧度  ); |

旋转矩阵**R**的逆矩阵等于它的转置矩阵：**R**T= **R**-1。这样的矩阵我们说它们是互相垂直的。

## 矩阵缩放

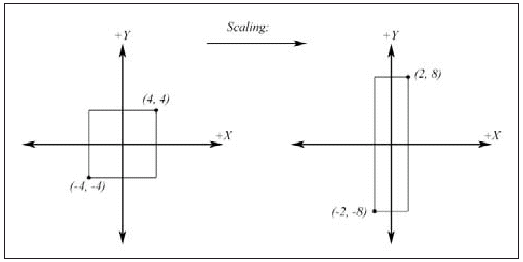
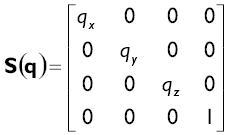


图10

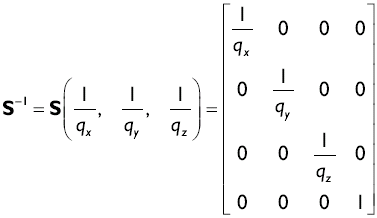
我们能通过与下面的矩阵相乘把向量沿x轴缩放*qx*个单位，沿y轴缩放*qy* 个单位，沿z轴缩放*qz*个单位：



将矩阵缩放的D3DX函数是：

|  |
| --- |
| D3DXMATRIX \*D3DXMatrix**Scaling**(  D3DXMATRIX\* pOut, // 返回缩放后的矩阵  FLOAT sx, // x轴缩放的比例  FLOAT sy, // y轴缩放的比例  FLOAT sz // z轴缩放的比例.  ); |

缩放矩阵求逆只需要将每个缩放因子取倒即可：

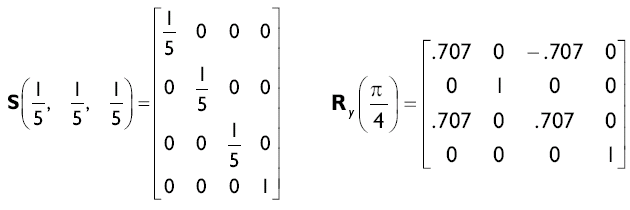


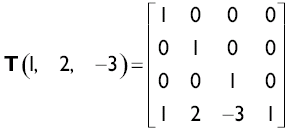
## 综合变换

常常我们要对一个向量进行一系列的变换。比如，我们可能先缩放一个向量，然后旋转它，最后把它平移到指定的位置。

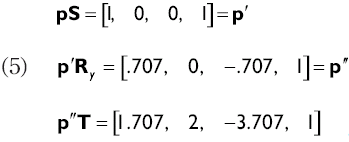
例如：先把向量**p** = [5, 0, 0, 1] 在所有轴上缩小为原来的1/5，然后沿着y轴旋转π/4，最后把它在x轴上移动1个单位，在y轴上移动2个单位，在z轴上移动3个单位。

解答：注意我们必须完成缩放，沿y轴旋转，以及移动。我们设缩放、旋转、移动的变换矩阵分别是**S**, **R***y*, **T**，如下：

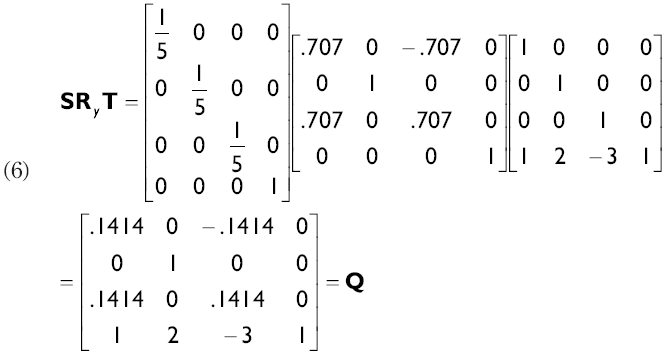




应用缩放，旋转，以及平移一系列变换，我们得到：



我们能用矩阵乘法把几个变换矩阵转换成一个矩阵，它是非常有益的矩阵。比如，重新考虑这部分开始的例子。通过使用矩阵相乘把3个变换矩阵合成一个矩阵。注意我们必须按实际应用的顺序来进行矩阵相乘。



那么 **pQ** = [1.707, 2, –3.707, 1]。

联合变换有提高效率的能力。假如我们需要对一组数量巨大的向量（在3D图形任务中是很普遍的）进行同样的缩放，旋转以及移动变换。替换这一系列的变换，即就象等式(5)中对每一个向量的做法，我们能把所有3个变换转换到一个矩阵中，即就象在等式(6)中的做法。这样我们只需要对每一个向量进行一次乘法就可以实现3种变换。这就减少了大量的向量-矩阵乘法操作。

## 一些向量变换函数

D3DX库分别提供了下边两个对点和向量的变换函数。D3DXVec3TransformCoord函数变换点同时设置向量第4个成员为1（用于变换点向量）。D3DXVec3TransformNormal函数变换向量并且设置第4个成员为0（用于变换方向向量）。

D3DXVECTOR3 \*D3DXVec3**TransformCoord**(

D3DXVECTOR3\* pOut, // 返回的点向量

CONST D3DXVECTOR3\* pV, // 点向量

CONST D3DXMATRIX\* pM // 变换矩阵

);

|  |
| --- |
| D3DXMATRIX T(...); // 初始化矩阵  D3DXVECTOR3 p(...); // 初始化点  D3DXVec3TransformCoord( &p, &p, &T); // 变换一个点 |

D3DXVECTOR3 \*WINAPI D3DXVec3**TransformNormal**(

D3DXVECTOR3 \*pOut, //返回的方向向量

CONST D3DXVECTOR3 \*pV, // 方向向量

CONST D3DXMATRIX \*pM //变换矩阵

);

|  |
| --- |
| D3DXMATRIX T(...); // 初始化变换矩阵  D3DXVECTOR3 v(...); // 初始化方向向量  D3DXVec3TransformNormal( &v, &v, &T); // 变换方向向量 |

注意：D3DX库也提供D3DXVec3TransformCoordArray和D3DXVec3TransformNormalArray来分别变换一个点数组和向量数组

## 平面

一个平面能通过一个向量**n**和平面上的一个点**p**0来描述。这个向量**n**垂直于平面，它被称为此平面的法向量（如图11）。

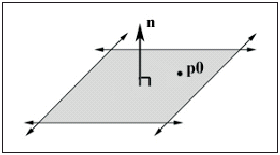


图11

在图12中我们能够发现平面上任意一点**p**都满足如下等式。即：假如**p**、**p**0都是平面上一点，那么向量（**p** - **p**0）垂直于平面的法向量。

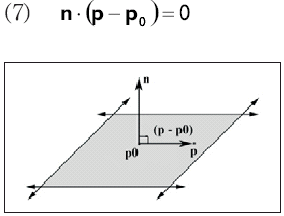


图12

当我们通过法向量**n**和平面上一个已知点来描述一个平面时，等式（7）又被写成这样：



这时*d* = –**n**·**p**0。

## D3DX平面

在代码中描述一个平面：仅仅需要一个法向量**n**和常数**d**就可以了。因此我们就使用一个4D向量（我们记录成(**n**, *d*)）来实现它。D3DX库中用如下的结构来定义一个平面：

|  |
| --- |
| typedef struct D3DXPLANE  {  #ifdef \_\_cplusplus  public:  D3DXPLANE() {}  D3DXPLANE( CONST FLOAT\* );  D3DXPLANE( CONST D3DXFLOAT16\* );  D3DXPLANE( FLOAT a, FLOAT b, FLOAT c, FLOAT d );  // casting  operator FLOAT\* ();  operator CONST FLOAT\* () const;  // unary operators  D3DXPLANE operator + () const;  D3DXPLANE operator - () const;  // binary operators  BOOL operator == ( CONST D3DXPLANE& ) const;  BOOL operator != ( CONST D3DXPLANE& ) const;  #endif //\_\_cplusplus  FLOAT a, b, c, d;  } D3DXPLANE, \*LPD3DXPLANE; |

对照等式（8）可知：这里a, b和c是平面法向量**n**的成员，d就是那个常数。

## 点和平面的空间关系

我们判定点和平面的关系主要是利用等式(8)来实现。例如，假设平面(**n**, *d*)，我们能判定点**p**和平面的关系**：**

假如**n**·**p** + *d* = 0，那么点**p**与平面共面。

假如**n**·**p** + *d* >0，那么点**p**在平面的前面且在平面的正半空间里。

假如**n**·**p** + *d* <0，那么点**p**在平面的背面且在平面的负半空间里。

下边的D3DX函数就是利用**n**·**p** + *d* 来判定点和平面的关系的函数：

FLOAT D3DX**PlaneDotCoord**(

CONST D3DXPLANE \*pP, // 平面

CONST D3DXVECTOR3 \*pV // 点

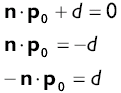
);

|  |
| --- |
| // 测试点相对于平面的位置  D3DXPLANE p(0.0f, 1.0f, 0.0f, 0.0f);  D3DXVECTOR3 v(3.0f, 5.0f, 2.0f);  float x = D3DXPlaneDotCoord( &p, &v );  if( x approximately equals 0.0f ) // v在平面.上  if( x > 0 ) // v在正半空间  if( x < 0 ) // v在负半空间 |

## 创建平面

我们能通过两种方法创建平面。

**第一种方法**，直接用指定法线和点创建平面。假设法线**n**和在平面上的已知点**p**0,我们就能求出*d：*



D3DX库提供如下函数来完成创建平面的任务：

|  |
| --- |
| D3DXPLANE \*D3DX**PlaneFromPointNormal**(  D3DXPLANE\* pOut, // Result.  CONST D3DXVECTOR3\* pPoint, // Point on the plane.  CONST D3DXVECTOR3\* pNormal // The normal of the plane.  ); |

**第二种方法**，我们能通过在平面上的3个点创立一个平面。

假如有点**p**0, **p**1, **p**2，那么我们就能得到平面上的两个向量：



因此我们能通过把平面上的两个向量进行十字相乘得到平面的法线。回忆左手坐标系。



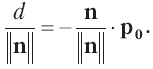
那么–(**n**·**p**0) = *d*.

D3DX库提供如下函数来完成通过同一平面上的3个点确定一个平面：

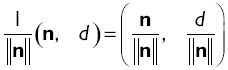
|  |
| --- |
| D3DXPLANE \*D3DX**PlaneFromPoints**(  D3DXPLANE\* pOut, // Result.  CONST D3DXVECTOR3\* pV1, // Point 1 on the plane.  CONST D3DXVECTOR3\* pV2, // Point 2 on the plane.  CONST D3DXVECTOR3\* pV3 // Point 3 on the plane.  ); |

## 标准化平面

有时我们可能想标准化一个平面的法向量，即标准化平面。初一想，好象我们只需象标准化其他向量一样标准化平面的法向量就可以了。但是回忆在等式**n**·**p** + *d* = 0中的*d* = –**n**·**p**0。我们明白法向量的长度将影响常数*d。*因此，假如我们标准化法向量，我们必须重新计算*d*.注意



因此，我们有下边公式来标准化平面(**n**, *d*)的法向量：



我们能用下面的D3DX函数来标准化一个平面：

|  |
| --- |
| D3DXPLANE \*D3DX**PlaneNormalize**(  D3DXPLANE \*pOut, // Resulting normalized plane.  CONST D3DXPLANE \*pP // Input plane.  ); |

## 变换平面

我们能够通过如下处理来变换一个面（**n**, *d*），就象一个4D向量通过乘以它渴望得到变换的变换矩阵的逆矩阵一样来达到变换目的。（哎，好难说清楚，还是看例子吧。）注意平面的法向量必须首先被标准化。

我们能用下面的D3DX函数来完成操作：

|  |
| --- |
| D3DXPLANE \*D3DX**PlaneTransform**(  D3DXPLANE \*pOut, // Result  CONST D3DXPLANE \*pP, // Input plane.  CONST D3DXMATRIX \*pM // Transformation matrix.  ); |

示例代码：

|  |
| --- |
| D3DXMATRIX T(...); // Init. T to a desired transformation.  D3DXMATRIX inverseOfT;  D3DXMATRIX inverseTransposeOfT;  D3DXMatrixInverse( &inverseOfT, 0, &T );  D3DXMatrixTranspose( &inverseTransposeOfT, &inverseOfT );  D3DXPLANE p(...); // Init. Plane.  D3DXPlaneNormalize( &p, &p ); // make sure normal is normalized.  D3DXPlaneTransform( &p, &p, &inverseTransposeOfT ); |

## 点到平面上最近的点

假如我们在空间中有一个点**p**并且想找到在平面( **n**, *d*)上的与**p**最接近一个点**q**。注意如果平面的法向量是单位长度，这将简化问题。

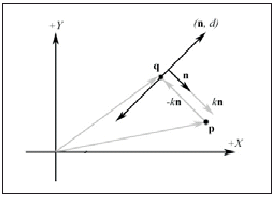


图13

从图13我们能看出**q** = **p** + (–*k*\_**n**)，k是有符号之分的从点**p**到平面的距离，也就是点**p**和**q**之间的有向距离。假如平面的法向量**n**是单位长度，那么**n**·**p** + *d* 就是从平面到点**p**有向距离.

## 射线（可选的）

设想在游戏中的一个玩家，正用他的枪射击敌人。我们怎么判断子弹是否从一个位置击中另一个位置的目标？一个方法是用一条射线模拟子弹，用一个球体模型模拟敌人。（球体模型只是一个球体，它紧紧的围绕一个物体，从而粗略地表示它的大小。球体模型将在第11章中做更详细的介绍。）那么通过计算我们就能够判定是否射中球体。在这部分我们学习射线的数学模型。

## 射线

一条射线能用一个起点和方向来描述。射线的参数方程是：



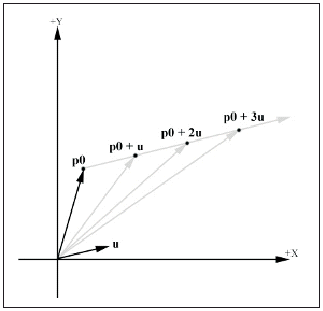


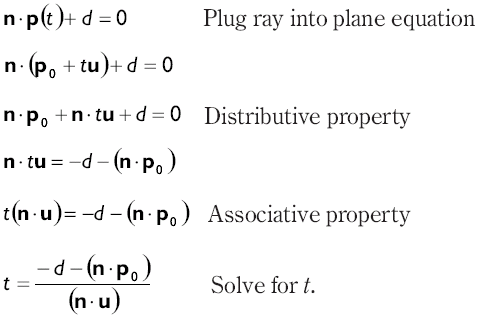
图14

**p**0 是射线的起点，**u**是射线的方向，*t*是参数。通过赋予不同的*t*值，我们能计算出在射线上不同的点。要描述一条射线，参数*t*范围就必须在[0, ∞)之间。实际上，假如我们让*t*∈(–∞, ∞),那么我们就能得到一条3维空间直线。

## 线/面相交

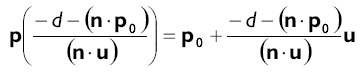
假设一条射线**p**(*t*) = **p**0 + *t***u** 和 一个平面**n**·**p** + *d* = 0，我们想知道射线是否与平面相交，以及相交的交点信息（如果相交的话）。照这样做，我们把射线代入平面方程并且求满足平面方程的参数*t，*解答出来的参数就是相交的点。

把等式（9）代入平面方程：



假如*t* 不在[0, ∞)之间，那么射线与平面不相交。

假如*t* 在[0, ∞)之间，那么射线与平面相交。且把参数代入射线方程就能找到交点：



## 摘要（略）

第一部分完

# 第二部分 Direct3D基础 (Direct3D Fundamentals)

# 第一章 初始化Direct3D (Direct3D Initialization)

以前Direct3D的初始化一直是一项单调乏味的工作。幸运的是8.0版本简化了初始化模式并且DX9.0也使用和它相同的模式。然而，在这个过程中仍需要程序员熟知图形学的基础知识和D3D的基本类型，本章的前几节将讲述这方面的内容。在余下的部分里将解释初始化的过程。

## 目标

* 学习D3D怎样与图形硬件相互作用
* 弄懂COM在D3D中所扮演的角色
* 学习基础图形学知识，如2D图片是如何存储的、页面切换和深度缓冲
* 学习如何初始化D3D
* 熟悉本书例程中的一些常用的结构

## 1.1 Direct3D概述

Direct3D是一种低层图形API，它能让我们利用3D硬件加速来渲染3D世界。我们可以把Direct3D看作是应用程序和图形设备之间的中介。例如通知图形设备清空屏幕，应用程序将调用Direct3D的IDirect3DDevice9::Clear方法。图1.1显示了应用程序、Direct3D和图形设备之间的关系。

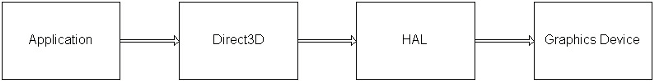


图1.1

图1.1中Direct3D所表示的是Direct3D中已定义的，供程序员使用的Direct3D接口和函数的集合。这些接口和函数代表了当前版本的Direct3D所支持的全部特性。注意：仅仅因为Direct3D支持某种特性，并不意味着你所使用的图形硬件（显卡）也能支持它。

如图1.1所示，在Direct3D和图形设备之间有一层中介——叫做硬件抽象层（HAL，Hardware Abstraction Layer）。Direct3D不能直接作用于图形设备，因为现在市面上的显卡种类实在是太多了并且每种显卡都有不同的性能和处理事件的方式。例如，两种不同的显卡实现清屏的方式也可能是不同的。因此，Direct3D要求设备制造商实现HAL。HAL是一组指示设备执行某种操作的特殊设备代码的集合。用这种方法，Direct3D避免了必须去了解某个设备的特殊细节，使它能够独立于硬件设备。

设备制造商在HAL中实现他们的产品所支持的所有特性。HAL将不会实现那些Direct3D支持但硬件产品不支持的特性。调用一个HAL中没有实现的Direct3D的函数将会出错，除非它是顶点处理操作，因为这个功能可以由软件模拟来实现。因此当使用某些仅由市面上少数显卡所支持的高级特性时，必须检测一下设备是否支持。（设备的功能将在1.3.8节中讲解）

### 1.1.1 REF设备

你也许想把一些你的设备不支持的Direct3D函数写入程序。为了达到这个目的，Direct3D提供了REF设备,它用软件模拟了所有的Direct3D API。这允许你写并测试那些你的显卡不支持的Direct3D特性的代码。例如在本书的第四部分，某些人的显卡可能会不支持顶点和像素着色器。如果你的显卡不支持着色器，你仍然能够使用RE设备测试示例代码。懂得RE设备仅仅是为了发展，这是很重要的。它只会和DirectX SDK一起被装载，而不会发布给最终用户。 另外，RE设备实在是太慢了，除了测试以外它没有任何利用价值。

### 1.1.2 D3DDEVTYPE

在代码中，我们用D3DDEVTYPE\_HAL来定义HAL设备，它是D3DDEVTYPE枚举类型的一个成员。同样的，REF设备则由D3DDEVTYPE\_REF来定义，它也属于D3DDEVTYPE枚举类型。记住这些类型很重要，因为在创建设备的时候我们需要指定我们将要使用的类型。

## 1.2 COM

组件对象模型（COM, Component Object Model）是一种能使DirectX独立于编程语言和具有向下兼容性的技术。我们通常把COM对象作为一个接口，你可以把它当作达到某种目的的C++类来使用它。当使用C++写DirectX程序的时候，COM的大部分细节对我们来说是透明。但是有一件事，我们必须知道，那就是我们通过某个特殊的COM接口的函数或指针获得了另一个COM接口指针，而不是通过C++的新关键字来创建它。当我们使用完某个接口后，调用它的Release方法比直接Delete它更好。COM对象具有它们自己的内存管理。

对COM来说还有很多细节可以了解，但是掌握这些细节对于我们有效的使用DirectX不是必须的。

注意：COM接口都具有前缀大写字母“I”，例如表示一个表面的COM接口叫做IDirect3DSurface9。

## 1.3 一些准备工作

Direct3D的初始化过程要求我们对图形学基础知识和Direct3D类型有一定了解。本节将介绍这些知识和类型，以确保下一节能把焦点集中在讨论Direct3D的初始化上。

### 1.3.1 表面

表面是一个像素点阵，在Direct3D中主要用来存储2D图形数据。图1.2指明了表面的一些成分。由图可以看出表面数据就像一个矩阵，像素数据实际上存储在线性数组里面。

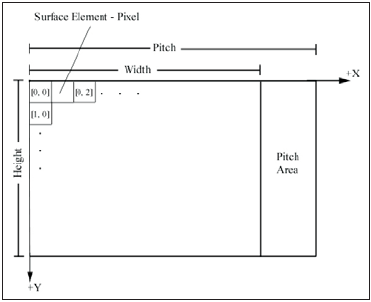


图1.2

表面的Width和Height是按像素计算的。Pitch以字节为单位。而且Pitch有可能比Width大且依赖于低层硬件，所以不能单纯的认为Pitch = Width \* sizeof (pixelFormat)。

在代码中，我们可以使用IDirect3DSurface9接口来描述表面。这个接口提供若干方法来直接读写表面数据并且还有一个方法用来返回表面信息。IDirect3DSurface9中最重要的方法是：

* LockRect——使用这个方法，我们将获得一个指向表面内存的指针，然后，通过一系列指针运算，我们可以对表面上任一个像素点进行读、写操作。
* UnlockRect——当你调用了LockRect和完成了对表面内存的访问后，你必须调用这个方法给表面解锁。
* GetDesc——这个方法将通过填充D3DSURFACE\_DESC结构来返回表面的描述信息。

最初锁定表面和改写每一像素看来稍微有点迷茫。下面的代码表示锁定表面并将每一像素染成红色：

|  |
| --- |
| // 假定\_surface是一个指向IDirect3DSurface9接口的指针  // 假定每个像素为：32-bit的像素格式  // 取得表面描述  D3DSURFACE\_DESC surfaceDesc;  \_surface->GetDesc(&surfaceDesc);  // 取得被锁定表面的像素数据的指针  D3DLOCKED\_RECT lockedRect;  \_surface->LockRect(  &lockedRect, // 指向被锁定表面的数据  0, // 0表示锁定全部表面  0); // 0表示没有指定锁定标记  // 遍例表面上的每个像素，将它们设为红色  DWORD\* imageData = (DWORD\*)lockedRect.pBits;  for(int i = 0; i < surfaceDesc.Height; i++)  {  for(int j = 0; j < surfaceDesc.Width; j++)  {  // 取得纹理索引, 注意我们用pitch 除以4是因为pitch的单位是像素，  // 并且一个DWORD类型占为4 bytes空间  int index = i \* lockedRect.Pitch / 4 + j;  imageData[index] = 0xffff0000; // 每个像素设为红色  }  }  \_surface->UnlockRect(); |

程序中D3DLOCKED\_RECT结构的定义如下：

|  |
| --- |
| typedef struct \_D3DLOCKED\_RECT {  INT Pitch; // 表面深度  void \*pBits; // 指向表面开始处的内存  } D3DLOCKED\_RECT; |

在这里有一些关于表面锁定代码的一些说明。32-bit像素格式这个设定很重要，我们把bits转换成DWORD**s**。这让我们能把每一个DWORD视为表示一个像素。同样我们暂时不用去关心为什么0xffff0000表示红色,关于颜色的说明将在第四章讲到。

### 1.3.2 Multisampling

由于使用像素矩阵来表示图像，在显示时会出现锯齿状，Multisampling就是使其变得平滑的技术。它的一种最普通的用法即为——全屏抗锯齿（看图1.3）。

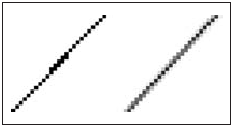


图1.3

D3DMULTISAMPLE\_TYPE枚举类型使我们可以指定全屏抗锯齿的质量等级：

* D3DMULTISAMPLE\_NONE——不使用全屏抗锯齿。
* D3DMULTISAMPLE\_1\_SAMPLE…D3DMULTISAPLE\_16\_SAMPLE——设定1~16级的等级。

本书的示例程序中没有使用全屏抗锯齿的功能，因为它大大的降低了程序运行速度。如果你实在很想使用它的话，要记住使用IDirect3D9::CheckDeviceMultisampleType来检测你的显卡是否支持。

### 1.3.3像素格式

当我们创建一个表面或纹理时，经常需要指定这些Direct3D资源的像素格式。它是由D3DFORMAT枚举类型的一个成员来定义的。这里例举一部分：

* D3DFMT\_R8G8B8——表示一个24位像素，从左开始，8位分配给红色，8位分配给绿色，8位分配给蓝色。
* D3DFMT\_X8R8G8B8——表示一个32位像素，从左开始，8位不用，8位分配给红色，8位分配给绿色，8位分配给蓝色。
* D3DFMT\_A8R8G8B8——表示一个32位像素，从左开始，8位为ALPHA通道，8位分配给红色，8位分配给绿色，8位分配给蓝色。
* D3DFMT\_A16B16G16R16F——表示一个64位浮点像素，从左开始，16位为ALPHA通道，16位分配给蓝色，16位分配给绿色，16位分配给红色。
* D3DFMT\_A32B32G32R32F——表示一个128位浮点像素，从左开始，32位为ALPHA通道，32位分配给蓝色，32位分配给绿色，32位分配给红色。

想了解全部的像素格式请查看SDK文档中的D3DFORMAT部分。

注意：这前三种格式（D3DFMT\_R8G8B8、D3DFMT\_X8R8G8B8、D3DFMT\_A8R8G8B8）是最常用并为大部分显卡所支持。但浮点像素格式或其它一些类型的支持并不是很广泛，在使用它们前请先检测你的显卡，看是否支持。

### 1.3.4 内存池

表面和其它一些Direct3D资源被放在多种内存池中。内存池的种类由D3DPOOL枚举类型的一个成员来指定。可用到的内存池有下列几种：

* D3DPOOL\_DEFAULT——表示Direct3D将根据资源的类型和用途把它们放在最合适的地方。这有可能是显存、[AGP内存](#_AGP内存)或者系统内存中。值得注意的是，这种内存池中的资源必须要在IDirect3DDevice9::Reset被调用之前消毁掉，并且再次使用时必须重新初始化。
* D3DPOOL\_MANAGED——资源将由Direct3D管理并且按设备的需要来指定放在显存还是放在AGP内存中。当应用程序访问和改变资源时它先把这些资源拷贝到系统内存中，当需要时Direct3D会自动把它们拷贝到显存里。
* D3DPOOL\_SYSTEMMEM——指定资源放在系统内存中。
* D3DPOOL\_SCRATCH——指定资源放在系统内存中，它与D3DPOOL\_SYSTEMMEM不同之处在于使用这些资源不必受图形设备的限制。因此，参数使图形设备不能访问该内存池的资源，但资源可以相互拷贝。

### 1.3.5 交换链和页面切换

Direct3D通常创建2~3个表面组成一个集合，即为交换链，通常由IDirect3DSwapChain接口来表示。我们不必去了解它更详细的细节。我们也很少去管理它，通常Direct3D会自己去管理。所以我们只要大概的了解一下它就可以了。

交换链以及页面切换技巧被用在使两帧动画之间过度更平滑。图1.4展示的是一个有两个绘制表面的交换链。

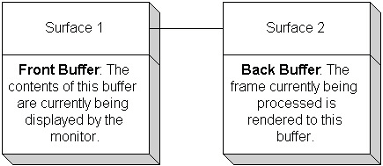


图1.4

如图1.4，在Front Buffer中的表面将用来在屏幕上显示。显示器不能即时显示Front Buffer中表示的图像；通常情况下，它是每六十分之一秒刷新显示一次，即刷新率为60赫兹。应用程序的帧率经常与监视器的刷新率不同步（比如应用程序的渲染帧速度可能比显示器的刷新速度快）。然而，我们不能在显示器显示完成当前帧之前就更新有下一帧动画的Front Buffer内容，但是我们又不想让程序停止渲染而去等待显示器显示。因此，我们渲染另一个屏幕表面Back Buffer。当监视器将Front Buffer显示出来后，Front Buffer就被放到交换链的末端，即变成图中的Back Buffer，而Back Buffer就会变成交换链中的Front Buffer。这个过程就叫做presenting。图1.5表示了交换的整个过程。

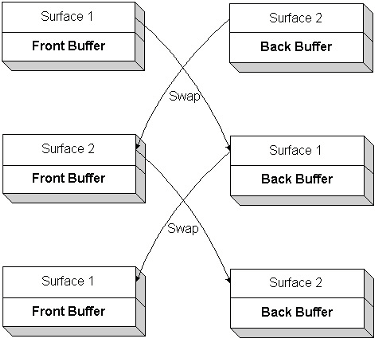


图1.5

因此，我们绘图代码的结构就会像下面这样：

1. Render to back buffer
2. Present the back buffer
3. Goto (1)

### 1.3.6 深度缓冲

深度缓冲也是一个表面，但它不是用来存储图像数据的，而是用来记录像素的深度信息。它将确定哪一个像素最后被绘制出来。所以，如果要绘制640\*480分辨率的图片，那么就会有640\*480个深度值。

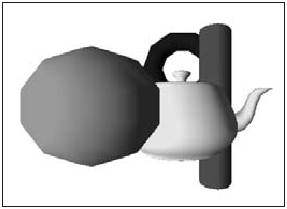


图1.6

图1.6展示了一个简单的场景，在这个场景里，一个物体把将另一个物体的一部分遮住了。为了使Direct3D能确定物体的前后关系并正确的绘制出来，我们使用一种深度缓冲，又叫做z-buffering的技术。

深度缓冲为每一个像素计算深度值，并进行深度测试。通过深度测试，我们可以比较出哪个像素离照相机更近，并将它画出来。这样就可以只绘制最靠近照相机的像素，被遮住的像素就不会被画出来。

深度缓冲的格式决定着深度测试的精确性。一个24位的深度缓冲比16位的深度缓冲更精确。通常，应用程序在24位深度缓冲下就能工作的很好，但是Direct3D也同时支持32位的深度缓冲。

* D3DFMT\_D32——表示32位深度缓冲
* D3DFMT\_D24S8——表示24位深度缓冲并保留8位模版缓冲（stencil buffer）
* D3DFMT\_D24X8——表示24位深度缓冲
* D3DFMT\_D24X4S4——表示24位深度缓冲并保留4位模版缓冲
* D3DFMT\_D16——表示16位深度缓冲

注意：关于模版缓冲的问题将在第八章详细说明。

### 1.3.7 顶点处理

顶点是3D图形学的基础，它能够通过两种不同的方法被处理，一种是软件方式（software vertex processing），一种是硬件方式（hardware vertex processing），前者总是被支持且永远可用，后者必须要显卡硬件支持顶点处理才可用。

使用硬件顶点处理总是首选，因为它比软件方式更快，而且不占用CPU资源，这意味CPU至少可以有更多的空闲时间进行别的计算。

注意：如果一块显卡支持硬件顶点处理的话，也就是说它也支持硬件几何转换和光源计算。

### 1.3.8 设备能力

Direct3D支持的每一项特性都对应于D3DCAPS9结构的一个数据成员。初始化一个D3DCAPS9实例应该以你的设备实际支持特性为基础。因此，在我们的应用程序里，我们能够通过检测D3DCAPS9结构中相对应的某一成员来检测设备是否支持这一特性。

下面将举例说明，假设我们想要检测显卡是否支持硬件顶点处理（换句话说，就是显卡是否支持硬件几何转换和光源计算）。通过查阅SDK中的D3DCAPS9结构，可以得知数据成员D3DCAPS9::DevCaps中的D3DDEVCAPS\_HWTRANSFORMANDLIGHT位表示硬件是否支持硬件顶点处理即硬件几何变换和光源计算。程序如下：

|  |
| --- |
| bool supportsHardwareVertexProcessing;  // 如果为真，意味着硬件设备支持它  if( caps.DevCaps & D3DDEVCAPS\_HWTRANSFORMANDLIGHT )  {  // 支持  supportsHardwareVertexProcessing = true;  }  else  {  // 不支持  hardwareSupportsVertexProcessing = false;  } |

注意：DevCaps即为“device capabilities”

　　　下一节将学习怎样根据硬件的实际情况来初始化D3DCAPS9

我们建议你阅读SDK中关于D3DCAPS9的结构，它完整的列出了Direct3D支持的特性。

## 1.4 初始化Direct3D

下面几点说明怎样初始化Direct3D。根据下边的步骤你能初始化Direct3D：

1. 获得一个IDirect3D9接口指针。这个接口用于获得物理设备的信息和创建一个IDirect3DDevice9接口，它是一个代表我们显示3D图形的物理设备的C++对象。
2. 检查设备能力（D3DCAPS9），搞清楚主显卡是否支持硬件顶点处理。我们需要知道假如它能支持，我们就能创建IDirect3DDevice9接口。
3. 初始化一个D3DPRESENT\_PARAMETERS结构实例，这个结构包含了许多数据成员允许我们指定将要创建的IDirect3DDevice9接口的特性。
4. 创建一个基于已经初始化好的D3DPRESENT\_PARAMETERS结构的IDirect3DDevice9对象。它是一个代表我们显示3D图形的物理设备的C++对象。

请注意，本书使用主显示设备绘制3D图形，如果你的机子只有一块显卡，那它就是主显示设备。如果你有多个显卡，那么你当前使用的显卡将会成为主显示设备（如：用来显示Windows桌面的显卡）。

### 1.4.1获得IDirect3D9接口

Direct3D的初始化是从获得一个IDirect3D9接口指针开始的。使用一个专门的Direct3D函数来完成这个工作是非常容易的，代码如下：

|  |
| --- |
| IDirect3D9\* \_d3d9;  \_d3d9 = Direct3DCreate9(D3D\_SDK\_VERSION); |

Direct3DCreate9的唯一一个参数总是D3D\_SDK\_VERSION，这可以保证应用程序通过正确的头文件被生成。如果函数调用失败，那么它将返回一个空指针。

IDirect3D9对象通常有两个用途：设备列举和创建IDirect3DDevice9对象。设备列举即为查明系统中显示设备的技术特性，显示模式、格式，以及其它每一种显卡各自支持的特性。创建代表物理设备的IDirect3DDevice9对象，我们需要利用这个物理设备的显示模式结构和格式来创建它。为了找到一个工作配置，我们必须使用IDirect3D9的列举方法。

然而，设备列举实在太慢了，为了使Direct3D运行得尽可能快，我们通常不使用这个测试，除了下一节所谈到的一项测试。为了安全跳过它，我们可以选择总是被所有显卡都支持的“安全”配置。

### 1.4.2 检测硬件顶点处理

当我们创建一个IDirect3DDevice9对象来表示主显示设备时，必须要设定其顶点处理的类型。如果可以的话，当然要选用硬件顶点处理，但是由于并非所有显卡都支持硬件顶点处理，因此我们必须首先检查显卡是否支持。

首先我们要根据主显示设备的技术特性来初始化D3DCAPS9实例。可以使用如下方法：

|  |
| --- |
| HRESULT IDirect3D9::GetDeviceCaps(  UINT Adapter,  D3DDEVTYPE DeviceType,  D3DCAPS9 \*pCaps  ); |

* Adapter——指定要获得哪个显示适配器的特性
* DeviceType——指定设备类型（硬件设备（D3DDEVTYPE\_HAL），软件设备（D3DDEVTYPE\_REF））
* PCaps——返回一个已初始化的D3DCAPS9结构

然后，我们就可以象1.3.8部分那样检测显卡的能力了。下面就是代码片段：

|  |
| --- |
| // 填充主显示设备的能力（D3DCAPS9结构）  D3DCAPS9 caps;  d3d9->GetDeviceCaps(  D3DADAPTER\_DEFAULT, // 主显示设备  deviceType, // 设备类型，一般是D3DDEVTYPE\_HAL.  &caps); // 返回填充后的D3DCAPS9 结构，包含主显示设备的能力  // 是否可以使用硬件顶点处理?  int vp = 0;  if( caps.DevCaps & D3DDEVCAPS\_HWTRANSFORMANDLIGHT )  {  // 是，支持硬件顶点处理  vp = D3DCREATE\_HARDWARE\_VERTEXPROCESSING;  }  else  {  // 不，只能用软件顶点处理  vp = D3DCREATE\_SOFTWARE\_VERTEXPROCESSING;  } |

观察代码，我们使用变量vp来存储顶点处理类型。这是因为在稍后创建IDirect3DDevice9对象时要求指定其顶点处理的类型。

注意：标识符D3DCREATE\_HARDWARE\_VERTEXPROCESSING和D3DCREATE\_SOFTWARE\_VERTEXPROCESSING是预定义的值，它们分别代表硬件顶点处理和软件顶点处理。

技巧：若我们开发有一些新的，高级特性的程序，在使用前我们总是先检查硬件是否支持这些特性。

注意：如果一个应用程序在你的机子上不能运行，说明它用到的一些特性可能你的显卡并不支持，可以试试把设备类型换成REF。

### 1.4.3 填充D3DPRESENT\_PARAMETERS结构

初始化过程的下一步是填充一个D3DPRESENT\_PARAMETERS结构的实例。这个结构用于设定我们将要创建的IDirect3DDevice9对象的一些特性，它的定义如下：

|  |
| --- |
| typedef struct \_D3DPRESENT\_PARAMETERS\_ {  UINT BackBufferWidth;  UINT BackBufferHeight;  D3DFORMAT BackBufferFormat;  UINT BackBufferCount;  D3DMULTISAMPLE\_TYPE MultiSampleType;  DWORD MultiSampleQuality;  D3DSWAPEFFECT SwapEffect;  HWND hDeviceWindow;  BOOL Windowed;  BOOL EnableAutoDepthStencil;  D3DFORMAT AutoDepthStencilFormat;  DWORD Flags;  UINT FullScreen\_RefreshRateInHz;  UINT PresentationInterval;  } D3DPRESENT\_PARAMETERS; |

下面介绍其比较重要的数据成员，至于更详细的信息，请查阅SDK：

BackBufferWidth——后备缓冲表面的宽度（以像素为单位）

BackBufferHeight——后备缓冲表面的高度（以像素为单位）

BackBufferFormat——后备缓冲表面的像素格式（如：32位像素格式为D3DFMT——A8R8G8B8）

BackBufferCount——后备缓冲表面的数量，通常设为“1”，即只有一个后备表面

MultiSampleType——全屏抗锯齿的类型，详情请看SDK

MultiSampleQuality——全屏抗锯齿的质量等级，详情看SDK

SwapEffect——指定表面在交换链中是如何被交换的，取D3DSWAPEFFECT枚举类型中的一个成员。其中D3DSWAPEFFECT\_DISCARD是最有效的

hDeviceWindow——与设备相关的窗口句柄，你想在哪个窗口绘制就写那个窗口的句柄

Windowed——BOOL型，设为true则为窗口模式，false则为全屏模式

EnableAutoDepthStencil——设为true，D3D将自动创建深度/模版缓冲

AutoDepthStencilFormat——深度/模版缓冲的格式

Flags——一些附加特性，设为0或D3DPRESENTFLAG类型的一个成员。下列两个最常用的标志

全部的标志请查阅SDK：

D3DPRESENTFLAG\_LOCKABLE\_BACKBUFFER——设定后备表面能够被锁定，这会降低应用程序的性能

D3DPRESENTFLAG\_DISCARD\_DEPTHSTENCIL——深度/模版缓冲在调用IDirect3DDevice9::present方法后将被删除，这有利于提升程序性能

FullScreen\_RefreshRateInHz——刷新率，设定D3DPRESENT\_RATE\_DEFAULT使用默认刷新率

PresentationInterval——属于D3DPRESENT成员，又有两个常用标志，其余请查SDK：

D3DPRESENT\_INTERVAL\_IMMEDIATE——立即交换

D3DPRESENT\_INTERVAL\_DEFAULT——D3D选择交换速度，通常等于刷新率

填充示例如下：

|  |
| --- |
| D3DPRESENT\_PARAMETERS d3dpp;  d3dpp.BackBufferWidth = 800;  d3dpp.BackBufferHeight = 600;  d3dpp.BackBufferFormat = D3DFMT\_A8R8G8B8; //像素格式  d3dpp.BackBufferCount = 1;  d3dpp.MultiSampleType = D3DMULTISAMPLE\_NONE;  d3dpp.MultiSampleQuality = 0;  d3dpp.SwapEffect = D3DSWAPEFFECT\_DISCARD;  d3dpp.hDeviceWindow = hwnd;  d3dpp.Windowed = false; // fullscreen  d3dpp.EnableAutoDepthStencil = true;  d3dpp.AutoDepthStencilFormat = D3DFMT\_D24S8; // depth format  d3dpp.Flags = 0;  d3dpp.FullScreen\_RefreshRateInHz = D3DPRESENT\_RATE\_DEFAULT;  d3dpp.PresentationInterval = D3DPRESENT\_INTERVAL\_IMMEDIATE; |

### 1.4.4 创建IDirect3DDevice9对象

在填充完了D3DPRESENT\_PARAMETERS结构后，我们就可以用下面的方法创建一个IDirect3DDevice9对象了：

|  |
| --- |
| HRESULT IDirect3D9::CreateDevice(  UINT Adapter,  D3DDEVTYPE DeviceType,  HWND hFocusWindow,  DWORD BehaviorFlags,  D3DPRESENT\_PARAMETERS \*pPresentationParameters,  IDirect3DDevice9\*\* ppReturnedDeviceInterface  ); |

* Adapter——指定对象要表示的物理显示设备
* DeviceType——设备类型，前面说过
* hFocusWindow——同我们在前面d3dpp.hDeviceWindow的相同
* BehaviorFlags——设定为D3DCREATE\_SOFTWARE\_VERTEXPROCESSING或者D3DCREATE\_HARDWARE\_VERTEXPROCESSING
* pPresentationParameters——指定一个已经初始化好的D3DPRESENT\_PARAMETERS实例
* ppReturnedDeviceInterface——返回创建的设备

例子：

|  |
| --- |
| IDirect3DDevice9\* device = 0;  hr = d3d9->CreateDevice(  D3DADAPTER\_DEFAULT, // primary adapter  D3DDEVTYPE\_HAL, // device type  hwnd, // window associated with device  D3DCREATE\_HARDWARE\_VERTEXPROCESSING, // vertex processing type  &d3dpp, // present parameters  &device); // returned created device  if( FAILED(hr) )  {  ::MessageBox(0, "CreateDevice() - FAILED", 0, 0);  return 0;  } |

## 1.5 初始化Direct3D实例

在本章的例程中，初始化了一个Direct3D应用程序并用黑色填充显示窗口（如图1.7）。

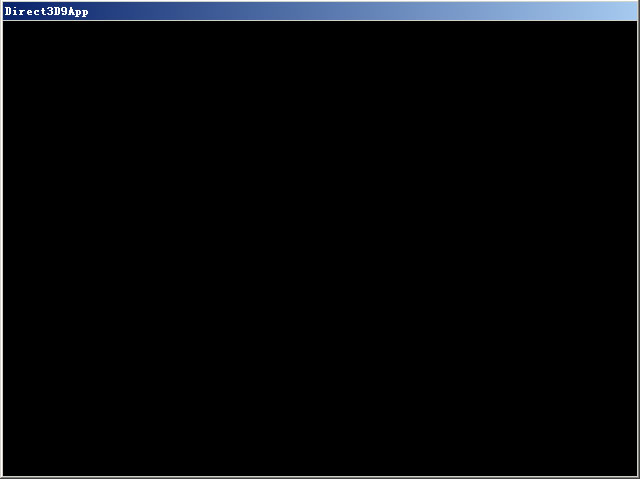


图1.7

本书所有的应用程序都包含了d3dUtility.h和d3dUtility.cpp这两个文件，它们所包含的函数实现了所有Direct3D应用程序都要去做的一些常见的功能。例如：创建一个窗口、初始化Direct3D、进入程序的消息循环等。将这些功能封装在函数中能使示例程序更加突出该章的主题。另外，在我们学习本书的过程中还会在这两个文件中加上一些通用的代码。

### 1.5.1 d3dUtility.h/cpp

在开始本章的例程之前，让我们先熟悉一下d3dUtility.h/cpp所提供的函数。d3dUtility.h如下：

|  |
| --- |
| // 包含主要的Direct3DX头文件，这将包含我们需要的另外的Direct3D头文件  #include <d3dx9.h>  namespace d3d  {  bool InitD3D(  HINSTANCE hInstance, // [in] 应用程序实例  int width, int height, // [in] Back buffer尺寸  bool windowed, // [in] 是否全屏  D3DDEVTYPE deviceType, // [in] HAL 或 REF  IDirect3DDevice9\*\* device); // [out] 创建的设备  int EnterMsgLoop(  bool (\*ptr\_display)(float timeDelta));  LRESULT CALLBACK WndProc(  HWND hwnd,  UINT msg,  WPARAM wParam,  LPARAM lParam);  template<class T> void Release(T t)  {  if( t )  {  t->Release();  t = 0;  }  }  template<class T> void Delete(T t)  {  if( t )  {  delete t;  t = 0;  }  }  } |

InitD3D——初始化一个应用程序主窗口并进行Direct3D的初始化。如果成功，则输出IDirect3DDevice9接口指针。从它的参数我们可以发现，我们能够设置窗口的大小和以窗口模式运行还是全屏模式运行。要知道它实现的细节，请看示例代码。

EnterMsgLoop——这个函数封装了应用程序的消息循环。它需要输入一个显示函数的函数指针，显示函数为程序中绘制图形的代码块，这样做是为了使显示函数能够在空闲的时候被调用并显示场景，它的实现如下：

|  |
| --- |
| int d3d::EnterMsgLoop( bool (\*ptr\_display)(float timeDelta) )  {  MSG msg;  ::ZeroMemory(&msg, sizeof(MSG));  static float lastTime = (float)timeGetTime();    while(msg.message != WM\_QUIT)  {  if(::PeekMessage(&msg, 0, 0, 0, PM\_REMOVE))  {  ::TranslateMessage(&msg);  ::DispatchMessage(&msg);  }  else  {  float currTime = (float)timeGetTime();  float timeDelta = (currTime - lastTime)\*0.001f;  ptr\_display(timeDelta); // call display function  lastTime = currTime;  }  }  return msg.wParam;  } |

与“time”有关的代码用于计算每次调用显示函数的时间间隔，即是每帧的时间。

Release——这个模版函数能方便的释放COM接口并将它们的值设为NULL

Delete——这个模版函数能方便的删除一个对象并将指向其的指针设为NULL

WndProc——应用程序主窗口的回调函数

### 1.5.2 实例框架

通过实例框架，我们形成了一种通用的方法去构造本书的示例程序。每一个例程都含有三个函数的实现，当然这不包括回调函数和WinMain主函数。这三个函数用特定的代码实现特定的功能。这三个函数是：

* bool Setup()——在这个函数里，我们将准备一切该程序需要用到的东西，包括资源的分配，检查设备能力，设置应用程序的状态
* void Clearup()——这个函数将释放Setup()中分配的资源，如分配的内存。
* bool Display(float timeDelta)——这个函数包含所有与我们绘图和显示有关的代码。参数timeDelta为每一帧的间隔时间，用来控制每秒的帧数。

### 1.5.3 D3D Init实例

这个示例程序将创建并初始化一个Direct3D应用程序，并用黑色填充屏幕。注意，我们使用了通用函数简化了初始化过程。

首先，我们要包含d3dUtility.h头文件，并为设备声明一个全局变量：

|  |
| --- |
| #include "d3dUtility.h"  IDirect3DDevice9\* Device = 0; |

然后实现我们的框架函数：

|  |
| --- |
| bool Setup()  {  return true;  }  void Cleanup()  {  } |

在这个程序中，我们不需要使用任何资源或触发任何事件，所以这两个函数都为空。

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  Device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER,  0x00000000, 1.0f, 0);  Device->Present(0, 0, 0, 0); // 页面切换  }  return true;  } |

Display方法调用了IDirect3DDevice::Clear方法，分别用黑色和1.0填充后备表面和深度/模版缓冲。如果应用程序不停止的话，我们会一直执行这个操作。IDirect3DDevice::Clear声明如下：

|  |
| --- |
| HRESULT IDirect3DDevice9::Clear(  DWORD Count,  const D3DRECT\* pRects,  DWORD Flags,  D3DCOLOR Color,  float Z,  DWORD Stencil  ); |

* Count——pRects组中的矩形的个数
* pRects——将要清除的屏幕矩形的数组，这使我们可以清除屏幕的某一部分
* Flags——指定在哪些表面上执行清除表面的操作

D3DCLEAR\_TARGET——目的表面，通常为后备表面

D3DCLEAR\_ZBUFFER——深度缓冲

D3DCLEAR\_STENCIL——模版缓冲

* Color——使用什么颜色填充清除的表面
* Z——设置深度缓冲的值
* Stencil——设置模版缓冲的值

屏幕被填充后，要调用IDirecte3DDevice9::Present方法进行后备表面的交换。

Windows 回调函数为一组事件集，即，我们可按ESC键让程序退出。

|  |
| --- |
| LRESULT CALLBACK d3d::WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)  {  switch( msg )  {  case WM\_DESTROY:  ::PostQuitMessage(0);  break;  case WM\_KEYDOWN:  if( wParam == VK\_ESCAPE )  ::DestroyWindow(hwnd);  break;  }  return ::DefWindowProc(hwnd, msg, wParam, lParam);  } |

最后，WinMain按如下步骤运行：

1. 初始化主显示窗口和Direct3D
2. 调用Setup进行程序的准备工作
3. 使用Display函数作为参数进入消息循环
4. 清除应用程序最后释放IDirecte3DDevice9对象

|  |
| --- |
| int WINAPI WinMain(HINSTANCE hinstance,  HINSTANCE prevInstance,  PSTR cmdLine,  int showCmd)  {  if(!d3d::InitD3D(hinstance, 800, 600, true, D3DDEVTYPE\_HAL, &Device))  {  ::MessageBox(0, "InitD3D() - FAILED", 0, 0);  return 0;  }  if(!Setup())  {  ::MessageBox(0, "Setup() - FAILED", 0, 0);  return 0;  }  d3d::EnterMsgLoop( Display );  Cleanup();  Device->Release();  return 0;  } |

就像你所看到的，我们的例程的模板结构是相当简洁的：有效的操作窗口函数、 Direct3D的初始化过程。这本书中的大部分程序，都是通过执行Setup, Cleanup, 和Display这三个函数来实现。

**注意**：不要忘了在你的工程中加入d3d9.lib、d3dx9.lib、winmm.lib 这三个库！

## 6摘要（略）

# 第二章 渲染管线 (The Rendering Pipeline)

本章的主题是渲染管线。它是用来创建为3D世界进行几何描述的2D图形并设定一个虚拟照相机确定这个世界中哪一部分将被透视投影到屏幕上。



图2.1

## 目标

* 要弄清楚我们怎样在Direct3D中表示3D物体
* 学习怎样模拟虚拟照相机
* 弄懂渲染管线——这个过程是用几何学来表现3D场景和用它来产生2D图象。

## 2.1表现模型

一个场景是多个物体或模型的集合。一个物体可以用三角形网格（triangle mesh）来近似表示，如图2.2所示。由三角形网格建立一个物体，我们称之为建模。3D世界中最基本的图元就是三角形，但是Direct3D也支持点图元和线图元但我们都不常用到。不过在学到第14章的粒子系统的时候，将会用到点图元。

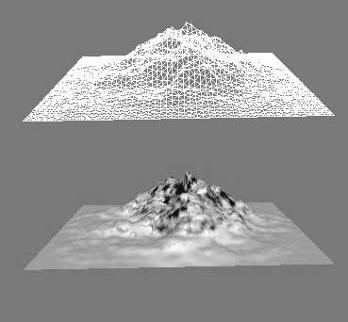


图2.2

一个多边形的两边相交的点叫做顶点。为了描述一个三角形，我们通常指定三个点的位置来对应三角形的三个顶点（如图2.3），这样我们就能够很明确的表示出这个三角形了。



图2.3

### 2.1.1 顶点格式

我们以前定义的点在数学上来说是正确的，但是当我们在Direct3D环境中使用它的时候就会觉得很不完善。这是因为在Direct3D中的顶点包含了许多附加的属性，而不再单纯的只有空间位置的信息了。例如：一个顶点可以有颜色和法线向量属性（这两个属性分别在第四章和第五章介绍）。Direct3D让我们可以灵活的构造自己的顶点格式。换句话说，我们可以自己定义顶点的成员。

为了创建一个自定义的顶点结构，我们首先要创建一个包含能存放我们选择的顶点数据的结构。例如，下面我们举出两种不同顶点数据类型的例子，一种包含了位置和颜色信息，第二种则包含了位置，法线向量，纹理坐标信息（“纹理”见第六章）。

|  |
| --- |
| struct ColorVertex  {  float \_x, \_y, \_z; // 位置  DWORD \_color; // 颜色  };  struct NormalTexVertex  {  float \_x, \_y, \_z; // 位置  float \_nx, \_ny, \_nz; // 法线向量  float \_u, \_v; // 纹理坐标  }; |

一旦我们有了完整的顶点格式，我们就要使用灵活顶点格式（FVF）的组合标志来描述它。例如第一个顶点结构，我们要使用如下的顶点格式：

|  |
| --- |
| #define FVF\_COLOR (D3DFVF\_XYZ | D3DFVF\_DIFFUSE) |

上面的顶点结构表明它包含位置和颜色属性。

而第二种结构则要使用：

|  |
| --- |
| #define FVF\_NORMAL\_TEX (D3DFVF\_XYZ | D3DFVF\_NORMAL | D3DFVF\_TEX1) |

上面的顶点结构表明它包含了位置，法线向量，纹理坐标的属性（这些常量是D3D内置的）。

有一点要注意，你的标志的顺序必须要和你的顶点结构的顺序一一对应。如果想知道所有的D3DFVF标志，请查阅SDK文档。

### 2.1.2 三角形

三角形是构建3D物体的基本图形。为了构造物体，我们创建了三角形列表（triangle list）来描述物体的形状和轮廓。三角形列包含了我们将要画的每一个三角形的数据信息。例如为了构造一个矩形，我们把它分成两个三角形，如图2.4所示，最后指定每个三角形的顶点。

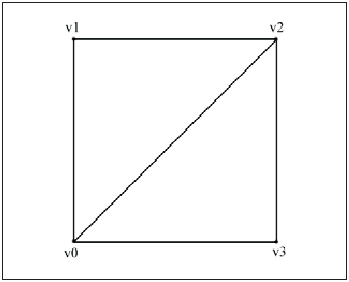
****

图2.4

|  |
| --- |
| Vertex rect[6] = {v0, v1, v2, // 三角形0  v0, v2, v3}; // 三角形1 |

注意：指定三角形顶点的顺序是很重要的，将会按一定顺序环绕排列，这会在2.3.4节学习相关的内容。

### 2.1.3 索引

3D物体中的三角形经常会有许多共用顶点。如图2.4所表示的矩形。虽然现在仅有两个点被重复使用，但是当要表现一个更精细更复杂的模型的时候，重复的顶点数将会变得很大。例如图2.5所示的立方体，仅有八个顶点，但是当用三角形列表示它的时候，所有的点都被重复使用。

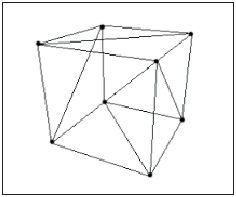


图2.5

为了解决这个问题，我们引入索引（indices）这个概念。它的工作方式是：我们创建一个顶点列表和一个索引列表（index list）。顶点列表包含所有不重复的顶点，索引列中则用顶点列中定义的值来表示每一个三角形的构造方式。回到那个矩形的示例上来，它的顶点列表的构造方式如下：

|  |
| --- |
| Vertex vertexList[4] = {v0, v1, v2, v3}; |

索引列表则定义顶点列中的顶点是如何构造这两个三角形的：

|  |
| --- |
| WORD indexList[6] = {0, 1, 2, //三角形0  0, 2, 3}; //三角形1 |

也就是说，用顶点列表中的0（vertexList[0]）、1（vertexList[1]）和2（vertexList[2]）顶点构成三角形0；用顶点列表中的0（vertexList[0]）、2（vertexList[2]）和3（vertexList[3]）顶点构成三角形1。

## 2.2虚拟照相机

照相机确定3D世界中的哪部分是可见的，因而需要将哪部分转换为2D图形。在3D世界中照相机被放置和定向，并且定义其可视体，图2.6展示了我们的照相机模型。

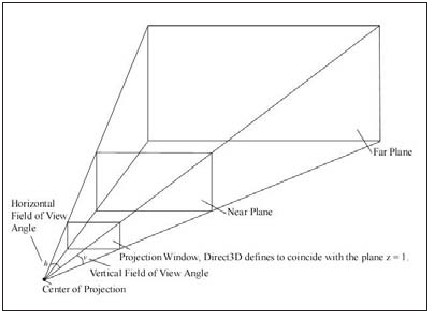


图2.6

可视体是由可视角度和前裁剪面（Near Plane）与后裁剪面（Far Plane）定义的一个平截头体。之所以要选择平截头体构造可视体，是因为我们的显示器都是矩形的。在可视体中不能被看见的物体都会被删除，删除这种数据的过程就叫做“裁剪”。

投影窗口（Projection Window）是可视体内的3D几何图形投影生成的用来显示3D场景的2D图像的2D区域。重要的是要知道，我们使用min=(-1,-1)和max=(1,1)来定义投影窗口的大小。

为了简化本书接下来的部分绘制，我们使前裁剪面与投影窗口在同一平面上。并且，注意Direct3D中定义的投影平面（即投影窗口所在的平面）是Z = 1的平面。

## 2.3 渲染管线

一旦我们描述几何学上的3D场景和设置了虚拟照相机，我们要把这个场景转换成2D图象显示在显示器上。这一系列必须完成的操作就叫做渲染管线。图2.7展示了一个简化的渲染管线，随后将详细解释图中的每一部分。

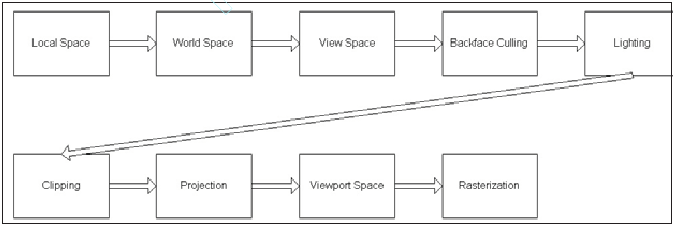


图2.7

渲染管线中的许多级都是从一个坐标系到另一个坐标的几何变换。这些变换都通过矩阵变换来实现。Direct3D为我们进行变换计算并且如果显卡支持硬件变换的话那就更有利了。使用Direct3D进行矩阵变换，我们唯一要做的事就是提供从一个系统变换到另一个系统的变换矩阵就可以了。我们使用IDirect3DDevice9::SetTranform方法提供变换矩阵。它输入一个表示变换类型的参数和一个变换矩阵。如图2.7所示，为了进行一个从自身坐标系到世界坐标系的变换，我们可以这样写：

|  |
| --- |
| Device->SetTransform(D3DTS\_WORLD, &worldMatrix); |

在下面的小节我们会了解到这个方法的更多细节。

### 2.3.1自身坐标系（Local Space）

自身坐标系又叫做建模空间，这是我们定义物体的三角形列的坐标系。自身坐标系简化了建模的过程。在物体自己的坐标系中建模比在世界坐标系中直接建模更容易。例如，在自身坐标系中建模不像在世界坐标系中要考虑本物体相对于其他物体的位置、大小、方向关系。

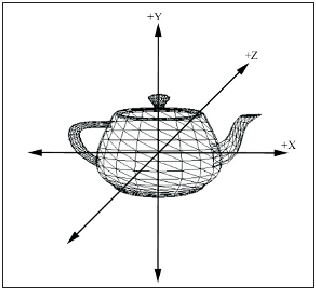


图2.8

### 2.3.2世界坐标系（World Space）

一旦我们构造了各种模型，它们都在自己的自身坐标系中，但是我们需要把它们都放到同一个世界坐标系中。物体从自身坐标系到世界坐标系中的换叫做世界变换。世界变换通常是用平移、旋转、缩放操作来设置模型在世界坐标系中的位置、大小、方向。世界变换就是通过各物体在世界坐标系中的位置、大小和方向等相互之间的关系来建立所有物体。

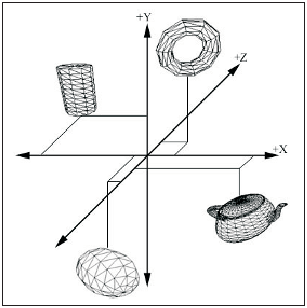


图2.9

世界变换由一个矩阵表示，并且在Direct3D中调用IDirect3DDevice9::SetTransform方法设置它，记住将转换类型设为D3DTS\_WORLD。例如我们要在世界坐标系中放置一个立方体定位在（-3，2，6）和一个球体定位在（5，0，-2），我们可以这样写程序：

|  |
| --- |
| //创建立方体的世界矩阵（一个平移矩阵）  D3DXMATRIX cubeWorldMatrix;  D3DXMatrixTranslation(&cubeWorldMatrix, -3.0f, 2.0f, 6.0f);  //创建球体的世界矩阵（一个平移矩阵）  D3DXMATRIX sphereWorldMatrix;  D3DXMatrixTranslation(&sphereWorldMatrix, 5.0f, 0.0f, -2.0f);  // 变换立方体，然后绘制它  Device->SetTransform(D3DTS\_WORLD, &cubeWorldMatrix);  drawCube(); // draw the cube  // 因为球体使用一个不同的世界变换，我们必须更改世界矩阵为球体的～，  // 如果不更改，球体将绘制在上一个世界矩阵的位置上（立方体的世界矩阵）  Device->SetTransform(D3DTS\_WORLD, &sphereWorldMatrix);  drawSphere(); // 绘制球体 |

这是个非常简单的实例，没有用到矩阵的旋转和缩放。但是一般很多物体都需要进行这些变换，不过这个例子也还是展示了世界变换是怎样进行的。

### 2.3.3视图坐标系（View Space）

世界坐标系中的几何图与照相机是相对于世界坐标系而定义的，如图2.10所示。然而在世界坐标系中当照相机是任意放置和定向时，投影和其它一些操作会变得困难或低效。为了使事情变得更简单，我们将照相机平移变换到世界坐标系的源点并把它的方向旋转至朝向Z轴的正方向，当然，世界坐标系中的所有物体都将随着照相机的变换而做相同的变换。这个变换就叫做视图坐标系变换（view space transformation）。

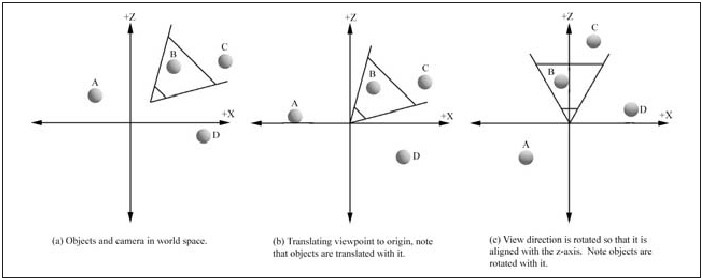


图2.10

视图坐标的变换矩阵可以通过如下的D3DX函数计算得到：

|  |
| --- |
| D3DXMATRIX \*D3DXMatrixLookAtLH(  D3DXMATRIX\* pOut, // 指向返回的视图矩阵  CONST D3DXVECTOR3\* pEye, // 照相机在世界坐标系的位置  CONST D3DXVECTOR3\* pAt, // 照相机在世界坐标系的目标点  CONST D3DXVECTOR3\* pUp // 世界坐标系的上方向(0, 1, 0)  ); |

pEye参数指定照相机在世界坐标系中的位置，pAt参数指定照相机所观察的世界坐标系中的一个目标点，pUp参数指定3D世界中的上方向，通常设Y轴正方向为上方向，即取值为（0，1，0）。

例如：假设我们要把照相机放在点（5，3，-10），并且目标点为世界坐标系的中点（0，0，0），我们可以这样获得视图坐标系变换矩阵：

|  |
| --- |
| D3DXVECTOR3 position(5.0f, 3.0f, –10.0f);  D3DXVECTOR3 targetPoint(0.0f, 0.0f, 0.0f);  D3DXVECTOR3 worldUp(0.0f, 1.0f, 0.0f);  D3DXMATRIX V;  D3DXMatrixLookAtLH(&V, &position, &targetPoint, &worldUp); |

视图坐标系变换也是通过IDirect3DDevice9::SetTransform来实现的，只是要将变换类型设为D3DTS\_VIEW，如下所示：

|  |
| --- |
| Device->SetTransform(D3DTS\_VIEW, &V); |

### 2.3.4背面拣选（Backface Culling）

一个多边形有两个表面，我们将一个标为正面，一个为背面。通常，后表面总是不可见的，这是因为场景中大多数物体是密封的。例如盒子、圆柱体、箱子、characters等，并且我们也不能把照相机放入物体的内部。因此照相机永不可能看到多边形的背面。这是很重要的，如果我们能看背面，那么背面拣选就不可能工作。

图2.11表示了一个物体在视图坐标系中的正面。一个多边形的边都是面向照相机叫正面多边形，而一个多边形的边都背对照相机叫背面多边形。

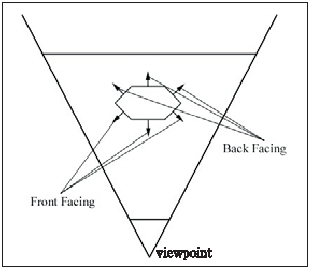


图2.11

由图2.11可知，正面多边形挡住了在它后面的背面多边形，Direct3D将通过拣选（即删除多余的处理过程）背面多边形来提高效率，这种方法就叫背面拣选。图2.12展示了背面拣选之后的多边形，从照相机的观察点来看，仍将绘制相同的场景到后备表面，那些被遮住的部分无论如何都永远不会被看见的。

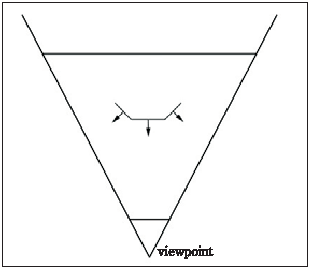


图2.12

当然，为了完成这项工作，Direct3D需要知道哪个多边形是正面，哪个是背面。Direct3D中默认顶点以顺时针方向（在观察坐标系中）形成的三角形为正面，以逆时针方向形成的三角形为背面。

如果我们不想使用默认的拣选状态，我们可以通过改变D3DRS\_CULLMODE来改变渲染状态：

|  |
| --- |
| Device->SetRenderState(D3DRS\_CULLMODE, Value); |

Value可以是如下一个值：

* D3DCULL\_NONE——完全不使用背面拣选
* D3DCULL\_CW——拣选顺时针方向环绕的三角形
* D3DCULL\_CCW——拣选逆时针方向环绕的三角形，这是默认值。

### 2.3.5光源（Lighting）

光源定义在世界坐标系中然后被变换到视图坐标系中。视图坐标系中光源给物体施加的光照大大增加了场景中物体的真实性，至于光照的相关函数的细节将会在第五章学习。在本书的第四部分，我们将使用可编程管线实现自己的光照。

### 2.3.6裁剪（Clipping）

我们拣选那些超出了可视体范围的几何图形的过程就叫做裁剪。这会出现三种情况：

* 完全包含——三角形完全在可视体内，这会保持不变，并进入下一级
* 完全在外——三角形完全在可视体外部，这将被拣选
* 部分在内（部分在外）——三角形一部分在可视体内，一部分在可视体外，则三角形将被分成两部分，可视体内的部分被保留，可视体之外的则被拣选

图2.13展示了上面三种情况：

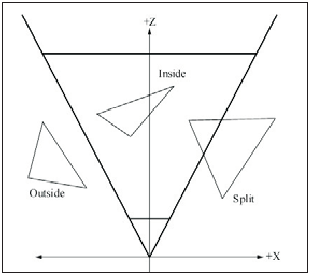


图2.13

### 2.3.7投影（Projection）

视图坐标系的主要任务就是将3D场景转化为2D图像表示。这种从n维转换成n-1维的过程就叫做投影。投影的方法有很多种，但是我们只对一种特殊的投影感兴趣，那就是透视投影。因为透视投影可以使离照相机越远的物体投影到屏幕上后就越小，这可以使我们把3D场景更真实的转化为2D图像。图2.14展示了一个3D空间中的点是如何通过透视投影到投影窗口上去的。

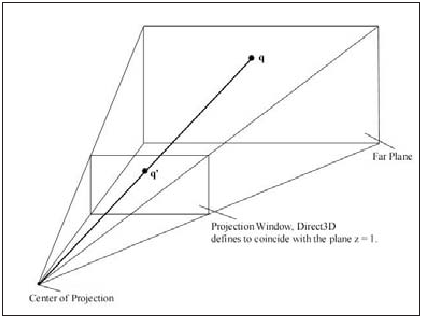


图2.14

投影变换的实质就是定义可视体**，**并将可视体内的几何图形投影到投影窗口上去。投影矩阵的计算太复杂了，这里我们不会给出推导过程，而是使用如下的Direct3D函数通过给出平截头体的参数来求出投影矩阵。

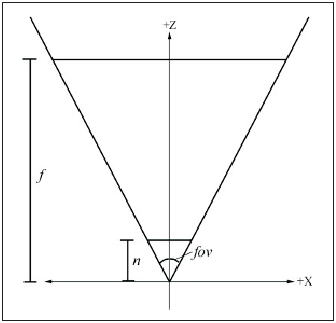


图2.15

|  |
| --- |
| D3DXMATRIX \*D3DXMatrixPerspectiveFovLH(  D3DXMATRIX\* pOut, // 返回的投影矩阵  FLOAT fovY, // 用弧度表示的视野角度vertical field of view angle in radians  FLOAT Aspect, // 宽高比  FLOAT zn, // 前裁剪面距离  FLOAT zf // 后裁剪面距离  ); |

（fovY定义镜头垂直观察范围，以弧度为单位。对于这个参数，下面是我的理解：如果定义为D3DX\_PI/4（90度角），那么就是表示以摄像机的观察方向为平分线，上方45度角和下方45度角就是摄像机所能看到的垂直范围了。嗯，可以想象一下自己的眼睛，如果可以把自己眼睛的fovY值设为D3DX\_PI/2（180度角），那么我们就可以不用抬头就看得见头顶的东西了。如果设为D3DX\_PI的话。。。我先编译一下试试（building…）。哈哈，结果啥也看不见。很难想象如果自己能同时看到所有方向的物体，那么将是一个怎样的画面啊）

Aspect参数为投影平面的宽高比例值，由于最后都为转换到屏幕上，所以这个比例一般设为屏幕分辨率的宽和高的比值（见2.3.8节）。如果投影窗口是个正方形，而我们的显示屏一般都是长方形的，这样转换后就会引起拉伸变形。



我们还是通过调用IDirect3DDevice9::SetTranform方法来进行投影变换，当然，要把第一个投影类型的参数设为D3DTS\_PROJECTION。下面的例子基于一个90度视角、前裁剪面距离为1、后裁剪面距离为1000的平截头体创建投影矩阵：

|  |
| --- |
| D3DXMATRIX proj;  D3DXMatrixPerspectiveFovLH(  &proj, PI \* 0.5f, (float)width / (float)height, 1.0, 1000.0f);  Device->SetTransform(D3DTS\_PROJECTION, &proj); |

### 2.3.8视口变换（Viewport Transform）

视口变换主要是转换投影窗口到显示屏幕上。通常一个游戏的视口就是整个显示屏，但是当我们以窗口模式运行的时候，也有可能只占屏幕的一部分或在客户区内。视口矩形是由它所在窗口的坐标系来描述的，如图2.16。

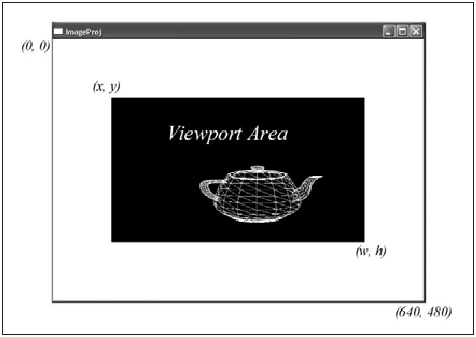


图2.16

在Direct3D中，视口矩形通过D3DVIEWPORT9结构来表示。它的定义如下：

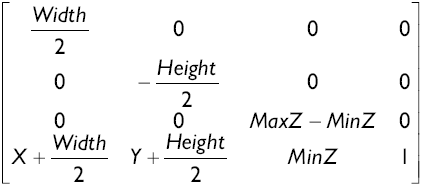
|  |
| --- |
| typedef struct \_D3DVIEWPORT9 {  DWORD X;  DWORD Y;  DWORD Width;  DWORD Height;  DWORD MinZ;  DWORD MaxZ;  } D3DVIEWPORT9; |

前四个参数定义了视口矩形与其所在窗口的关系。MinZ成员指定最小深度缓冲值，MaxZ指定最大深度缓冲值。Direct3D使用的深度缓冲的范围是0~1，所以如果不想做什么特殊效果的话，将它们分别设成相应的值就可以了。

一旦我们填充完D3DVIEWPORT9结构后，就可以如下设视口：

|  |
| --- |
| D3DVIEWPORT9 vp{ 0, 0, 640, 480, 0, 1 };  Device->SetViewport(&vp); |

这样，Direct3D就会自动为我们处理视口变换。现在还是给出视口变换矩阵作为参考：



### 2.3.9光栅化（Rasterization）

在把三角形每个顶点转换到屏幕上以后，我们就画了一个2D三角形。光栅化是计算需要显示的每个三角形中每个点颜色值（如图2.17）。

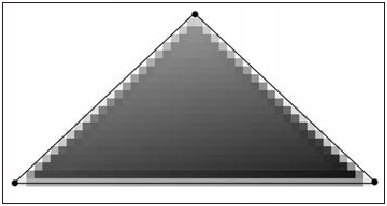


图2.17

光栅化过程是非常繁重的计算，它应该通过硬件图形处理来完成。它的处理结果就是把2D图象显示在显示器上。

## 2.4 摘要(略)

# 第三章 在Direct3D中绘制 (Drawing in Direct3D)

在上一章中我们学习了创建和渲染场景的概念。这一章中我们将这些东西用于实践，同时学习怎样在Direct3D中画一些几何物体。本章中所讲的有些Direct3D接口和方法很重要，因为它们的使用会贯穿全书。

## 目标

* 要弄清楚在Direct3D中怎样存储顶点和索引。
* 怎样使用渲染状态来改变渲染结果
* 学习怎样渲染场景
* 学习怎样用D3DXCreate\*函数创建更多的复杂的3D形体

## 3.1顶点/索引缓存

顶点和索引缓存有相似的接口并且共享相似的方法；因此我们把它们合在一起讲解。一个顶点缓存是一块连续的存储了顶点数据的内存。同样的，一个索引缓存是一块连续的存储了索引数据的内存。我们使用顶点和索引缓存保存我们的数据是因为它们能被放置在显存中。渲染显存中的数据要比渲染系统内存中的数据快的多。

在代码中，一个顶点缓存是通过IDirect3DVertexBuffer9接口来定义的。类似的，一个索引缓存是通过IDirect3DIndexBuffer9接口来定义。

### 3.1.1创建一个顶点和索引缓存

我们能使用下面两个方法创建一个顶点缓存和索引缓存：

|  |
| --- |
| HRESULT IDirect3DDevice9::CreateVertexBuffer(  UINT Length,  DWORD Usage,  DWORD FVF,  D3DPOOL Pool  IDirect3DVertexBuffer9\*\* ppVertexBuffer,  HANDLE\* pSharedHandle  );  HRESULT IDirect3DDevice9::CreateIndexBuffer(  UINT Length,  DWORD Usage,  D3DFORMAT Format,  D3DPOOL Pool,  IDirect3DIndexBuffer9\*\* ppIndexBuffer,  HANDLE\* pSharedHandle  ); |

这两个方法大部分参数是相同的，因此我们一起介绍它们。

* Length —— 分配给缓存的字节大小。假如想得到一个能存储8个顶点的顶点缓存，那么我们就要在顶点结构中设置这个参数为 8 \* sizeof ( Vertex ) 。
* Usage —— 指定关于怎样使用缓存的额外信息。这个值可以是0，没有标记，或者是下面标记的一个或多个的组合：
  + D3DUSAGE\_DYNAMIC——设置这个参数可以使缓存是动态的。在下一页说明静态和动态缓存。
  + D3DUSAGE\_POINTS——这个参数指定缓存存储原始点。原始点将在第14章粒子系统中介绍。这个参数仅仅用在顶点缓冲中。
  + D3DUSAGE\_SOFTWAREPROCESSING——使用软件顶点处理
  + D3DUSAGE\_WRITEONLY——指定应用程序只能写缓存。它允许驱动程序分配最适合的内存地址作为写缓存。注意如果从创建好的这种缓存中读数据，将会返回错误信息。
* FVF —— 存储在缓存中的顶点格式
* Pool —— 缓存放置在哪一个内存池中
* ppVertexBuffer ——返回创建好的顶点缓存的指针。
* pSharedHandle ——没有使用；设置为0。
* Format ——指定索引的大小；使用D3DFMT\_INDEX16设置16位索引，使用D3DFMT\_INDEX32设置32位索引。注意并非所有设备都支持32位索引；请检查设备能力。
* ppIndexBuffer ——返回创建好的索引缓存的指针。

注意：不使用D3DUSAGE\_DYNAMIC参数创建的缓存被叫做静态缓存。静态缓存通常被放置在显存中，在其中的数据能被很有效的处理。然而，对于静态缓存，从中读取和写入数据是很慢的，因为访问显存是很慢的。因为这个原因我们用静态缓存存储静态数据（不需要被经常改变的数据）。对于静态缓存地形和建筑物是很好的后选例子，因为在应用程序中他们通常不需要被改变。静态缓存应该在应用程序初始话的时候就被填充好，而不是在运行时才做。

注意：使用D3DUSAGE\_DYNAMIC参数创建的缓存被叫做动态缓存。动态缓存通常被放在AGP内存中，这种内存中的数据能被很快的更新。处理动态缓存中的数据不会比处理静态缓存中的数据快，因为这些数据必须在渲染前被转移到显存中，动态缓存的好处是它们能够被稍微快点地被更新（比CPU写快）。因此，假如你需要经常更新缓存中的数据，那么你就应该使用动态缓存。对于动态缓存粒子系统是很好的一个应用，因为它们是动态的，并且他们通常每一帧都会被更新。

注意：在程序中读取显存和AGP内存都是非常慢的。因此，假如你在运行时需要读取你的几何物体，最好的方案是指定一块系统内存，都在其中拷贝并且读取数据。

下边是创建一个静态顶点缓存的例子，该缓存能存储8个顶点。

|  |
| --- |
| IDirect3DVertexBuffer9\* vb;  device->CreateVertexBuffer(  8 \* sizeof( Vertex ),  0,  D3DFVF\_XYZ,  D3DPOOL\_MANAGED,  &vb,  0); |

### 3.1.2 访问缓冲内存

为了访问一个顶点/索引缓存，我们需要得到一个指针。我们通过一个指针获得缓存数据必须使用Lock方法。当我们访问完缓存后必须对它解锁。一旦有一个指向内存的指针，我们就能对它进行读写。

|  |
| --- |
| HRESULT IDirect3DVertexBuffer9::Lock(  UINT OffsetToLock,  UINT SizeToLock,  BYTE\*\* ppbData,  DWORD Flags  );  HRESULT IDirect3DIndexBuffer9::Lock(  UINT OffsetToLock,  UINT SizeToLock,  BYTE\*\* ppbData,  DWORD Flags  ); |

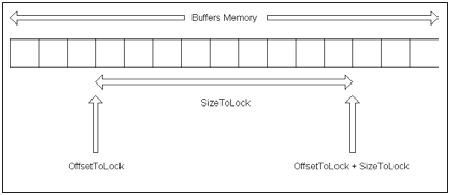


图3.1

这两个方法的参数都是完全相同的。

* OffsetToLock —— 偏移量，以字节为单位，从缓存开始位置到锁定开始位置的距离。如图3.1。
* SizeToLock —— 锁定的字节数。
* ppbData —— 一个指向锁定内存开始位置的指针。
* Flags —— 标记描述怎样锁定内存。它可能是0或者是下面参数中的1个或多个的组合：
  + D3DLOCK\_DISCARD——这个参数仅仅会在动态缓存时被使用。它指示硬件丢弃缓存并返回一个指向新分配的缓存的指针。这是很有用，因为当我们存取一个新分配的缓存时它允许硬件继续从丢弃的缓存渲染。这防止了硬件延迟。
  + D3DLOCK\_NOOVERWRITE——这个参数仅仅会在动态缓存时被使用。它声明你将向缓存中添加数据。即，你不能向已经渲染的内存中写数据。这是有好处的因为他允许你在添加新数据到缓存的同时让硬件继续渲染。
  + D3DLOCK\_READONLY——这个参数声明你锁定的缓存只能从中读取数据而不能写数据。这允许一些内在的优化。

用参数D3DLOCK\_DISCARD和D3DLOCK\_NOOVERWRITE的地址实际上就是缓存的一部分被使用（正在渲染）时它被锁定。假如情况允许这些标记被使用，当在锁定时他们防止渲染停止。

下边的例子展示了通常怎样使用Lock方法。注意当我们使用完以后要调用Unlock方法。

|  |
| --- |
| Vertex\* vertices;  \_vb->Lock(0, 0, (void\*\*)&vertices, 0); // 锁定整个缓存  vertices[0] = Vertex(-1.0f, 0.0f, 2.0f); // 向缓存里写顶点  vertices[1] = Vertex( 0.0f, 1.0f, 2.0f);  vertices[2] = Vertex( 1.0f, 0.0f, 2.0f);  \_vb->Unlock(); // 当你访问完缓存时，解锁缓存 |

### 3.1.3 找回顶点和索引缓存信息

有时我们需要得到顶点/索引缓存信息。下面的例子示范了用于获得这些信息的方法：

|  |
| --- |
| D3DVERTEXBUFFER\_DESC vbDescription;  \_vertexBuffer->GetDesc(&vbDescription); // 取得顶点缓存信息  D3DINDEXBUFFER\_DESC ibDescription;  \_indexBuffer->GetDesc(&ibDescription); //取得索引缓存信息 |

D3DVERTEXBUFFER\_DESC和D3DINDEXBUFFER\_DESC结构的定义如下：

|  |
| --- |
| typedef struct \_D3DVERTEXBUFFER\_DESC {  D3DFORMAT Format;  D3DRESOURCETYPE Type;  DWORD Usage;  D3DPOOL Pool;  UINT Size;  DWORD FVF;  } D3DVERTEXBUFFER\_DESC;  typedef struct \_D3DINDEXBUFFER\_DESC {  D3DFORMAT Format;  D3DRESOURCETYPE Type;  DWORD Usage;  D3DPOOL Pool;  UINT Size;  } D3DINDEXBUFFER\_DESC; |

## 3.2 渲染状态

Direct3D提供了多种渲染状态，它影响几何物体怎样被渲染。渲染状态有默认值，因此假如你的应用程序需要不同于默认设置的渲染时，你仅仅改变它即可。一种渲染效果会一直起作用，直到你下一次改变渲染状态为止。为了设置一个渲染状态，我们使用下面的方法：

|  |
| --- |
| HRESULT IDirect3DDevice9::SetRenderState(  D3DRENDERSTATETYPE State, // 更改的渲染状态  DWORD Value // 新的状态值  ); |

例如，在这一章的例子中我们将使用线框模式渲染我们的物体。因此，我们设置如下的渲染状态：

|  |
| --- |
| \_device->SetRenderState(D3DRS\_FILLMODE, D3DFILL\_WIREFRAME); |

注意：查看DirectX SDK中关于D3DRENDERSTATETYPE的信息。其中详细介绍了所有的渲染状态。

## 3.3 绘制准备

一旦我们创建好一个顶点缓存以及一个索引缓存（可选的）后，我们就为渲染其中的内容准备得差不多了，但是在渲染前我们还有3个步骤必须先做。

1. **设置资源流**。设置资源流与一个顶点缓存挂钩，此流就是一个流入渲染管线的几何信息的流。

下面的方法是用于设置一个资源流：

|  |
| --- |
| HRESULT IDirect3DDevice9::SetStreamSource(  UINT StreamNumber,  IDirect3DVertexBuffer9\* pStreamData,  UINT OffsetInBytes,  UINT Stride  ); |

* StreamNumber——确定我们的顶点缓存与哪一个资源流挂钩。在这本书中我们不使用多重流；因此我们总是使用0号流。
* pStreamData——一个指向我们想与流挂钩的那个顶点缓存的指针。
* OffsetInBytes——相对流开始处的偏移量。以字节为单位，它指定被填入渲染管线的顶点数据的开始位置。通过检查D3DCAPS9结构中的D3DDEVCAPS2\_STREAMOFFSET标志，假如你的设备支持，那么这个参数就有一些非0值。
* Stride——我们在顶点缓存中操作的每个部分的流的字节大小。

例如，假设vb是一个已经填充了顶点信息的顶点缓存：

|  |
| --- |
| \_device->SetStreamSource( 0, vb, 0, sizeof( Vertex ) ); |

1. **设置顶点格式**。在这里我们指定后面用来绘图调用的顶点的顶点格式。

|  |
| --- |
| \_device->SetFVF( D3DFVF\_XYZ | D3DFVF\_DIFFUSE | D3DFVF\_TEX1 ); |

1. **设置索引缓存**。假如我们使用了索引缓存，我们必须设置后面要用于绘制操作的索引缓存。每次我们只能使用一个索引缓存；因此假如你需要用一个不同的索引缓存绘制一个物体时，你必须转换到另一个上。下面的代码设置一个索引缓存：

|  |
| --- |
| \_device->SetIndices( \_ib ); // 传递一个索引缓存指针的拷贝 |

## 3.4用顶点/索引缓存绘制

在我们创建好顶点/索引缓存以及做好准备工作以后，我们就能绘制我们的几何物体了。这是通过使用DrawPrimitive或者DrawIndexedPrimitive传送几何信息到达渲染管线。这些方法从顶点流中获得顶点信息以及从索引缓存中获得索引信息。

### 3.4.1 IDirect3DDevice9::DrawPrimitive

这个方法不使用索引信息绘制图元。

|  |
| --- |
| HRESULT IDirect3DDevice9::DrawPrimitive(  D3DPRIMITIVETYPE PrimitiveType,  UINT StartVertex,  UINT PrimitiveCount  ); |

* PrimitiveType——我们绘制的图元类型。比如，我们能绘制点和线以及三角形。以后我们使用三角形，用D3DPT\_TRIANGLELIST参数。
* StartVertex——索引到在顶点流中的一个元素。设置渲染顶点中的开始点。这个参数给予我们一定的机动性，可以绘制一个顶点缓存中的某部分。
* PrimitiveCount——绘制图元的个数。

例子：

|  |
| --- |
| // 绘制4个三角形  \_device->DrawPrimitive( D3DPT\_TRIANGLELIST, 0, 4); |

### 3.4.2 IDirect3DDevice9::DrawIndexedPrimitive

这个方法使用索引信息来绘制图元。

|  |
| --- |
| HRESULT IDirect3DDevice9::DrawIndexedPrimitive(  D3DPRIMITIVETYPE Type,  INT BaseVertexIndex,  UINT MinIndex,  UINT NumVertices,  UINT StartIndex,  UINT PrimitiveCount  ); |

* Type——我们绘制的图元类型。比如，我们能绘制点和线以及三角形。以后我们使用三角形，用D3DPT\_TRIANGLELIST参数。
* BaseVertexIndex——一个基本数字，在调用中用它去加上索引。参看下面的说明。
* MinIndex——将被引用的最小索引值。
* NumVertices——在此调用中将被引用的顶点数。
* StartIndex——索引到索引缓存中的某个位置，它标记开始渲染的开始索引点。
* PrimitiveCount——绘制图元的个数。

例子：

|  |
| --- |
| \_device->DrawIndexedPrimitive(D3DPT\_TRIANGLELIST, 0, 0, 8, 0, 12); |

注意：BaseVertexIndex参数需要一些特别的解释。在解释过程中将会用到的图3.2。

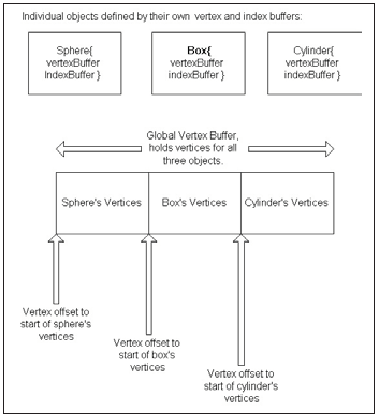


图3.2

在索引缓存中定位顶点相应的也就在顶点缓存中定位了。然而，假设我们想将球，盒子，圆柱体的顶点放置到一个公共的顶点缓存中。对于每一个物体，我们将不得不再计算在公共顶点缓存中的索引。这个新的索引值是通过与一个偏移量相加得到。注意这个偏移量是标准的顶点，而不是字节。

我们需要计算物体在公共顶点缓存中的索引值。Direct3D允许我们通过设置BaseVertexIndex参数得到一个顶点偏移量，随后Direct3D就能利用顶点自身的索引重新计算新的索引。

### 3.4.3 开始/结束场景

最后一点就是所有绘制方法都必须在IDirect3DDevice9::BeginScene和IDirect3DDevice9::EndScene方法之间被调用。例如，我们将这样写：

|  |
| --- |
| \_device->BeginScene();  // 绘制场景  \_device->DrawPrimitive(...);  \_device->EndScene(); |

## 3.5 D3DX几何物体

通过在代码中建造每个三角形来建造3D物体是一件非常枯燥的事。幸运的是，D3DX库已经为我们提供了一些方法来产生简单3D物体的网格数据。

D3DX库提供如下6种网格生成函数。

* D3DXCreateBox
* D3DXCreateSphere
* D3DXCreateCylinder
* D3DXCreateTeapot
* D3DXCreatePolygon
* D3DXCreateTorus

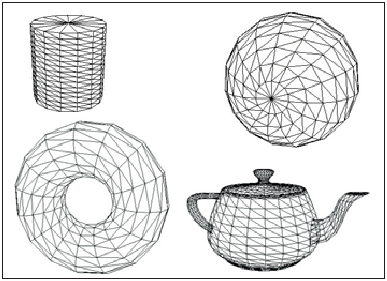


图3.3

这6种函数的使用都很类似，并且使用D3DX网格数据结构ID3DXMesh就象使用ID3DXBuffer接口一样。这些接口回在第10章和11章中讲解。现在，我们忽视它们的详细信息，只需简单使用它们即可。

|  |
| --- |
| HRESULT D3DXCreateTeapot(  LPDIRECT3DDEVICE9 pDevice, // 与mesh关联的设备  LPD3DXMESH\* ppMesh, // 返回的mesh  LPD3DXBUFFER\* ppAdjacency // 现在设成0  ); |

一个使用D3DXCreateTeapot函数的例子：

|  |
| --- |
| ID3DXMesh\* mesh = 0;  D3DXCreateTeapot(\_device, &mesh, 0); |

一旦生成了网格数据，我们就能使用ID3DXMesh::DrawSubset方法绘制图形了。这个方法有一个参数，它用来识别网格的一个子集。这个网格是通过上面的D3DXCreate\*函数中的一个子集创建的，因此可以给这个参数指定0值。一个渲染网格的例子：

|  |
| --- |
| \_device->BeginScene();  mesh->DrawSubset(0);  \_device->EndScene(); |

使用了网格以后，必须释放（release）它：

|  |
| --- |
| mesh->Release();  \_mesh = 0; |

## 3.6 实例程序：三角形、立方体、茶壶、D3DXCreate\*

这里有4个例子。

* 三角形——这是非常简单的应用程序，它示范了在线框模式下怎样创建并渲染一个三角形。
* 立方体——只比三角形稍微复杂一点，这个程序渲染一个线框立方体。
* 茶壶——这个程序使用D3DXCreateTeapot函数创建并渲染一个纺纱茶壶。
* D3DXCreate——这个程序创建并渲染几种不同的能够使用D3DXCreate\*函数创建的3D物体。

让我们简单讨论一下创建立方体的例子。通过对它的学习你自己就能很快地理解其他例子。

这个简单的绘制和渲染立方体的程序的运行结果如图3.4。

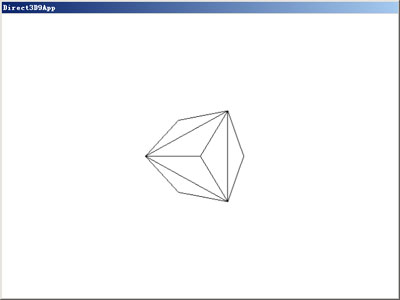


图3.4

首先我们定义下边两个全局变量来保存立方体的顶点和索引数据：

|  |
| --- |
| IDirect3DVertexBuffer9\* VB = 0;  IDirect3DIndexBuffer9\* IB = 0; |

下一步，我们定义两个全局常量，由它们来指定我们的屏幕大小：

|  |
| --- |
| const int Width = 800;  const int Height = 600; |

接下来定义我们的顶点结构以及结构中顶点的格式。在这个例子中顶点结构只保存顶点的位置信息：

|  |
| --- |
| struct Vertex  {  Vertex(){}  Vertex(float x, float y, float z)  {  \_x = x; \_y = y; \_z = z;  }  float \_x, \_y, \_z;  static const DWORD FVF;  };  const DWORD Vertex::FVF = D3DFVF\_XYZ; |

让我们把它迁移到框架程序（见1.53节）上。Setup函数创建顶点和索引缓存，锁定它们，把构成立方体的顶点写入顶点缓存，以及把定义立方体的三角形的索引写入索引缓存。然后把摄象机向后移动几个单位以便我们能够看见在世界坐标系中原点处被渲染的立方体。

|  |
| --- |
| bool Setup()  {  // 创建顶点、索引缓存  Device->CreateVertexBuffer(  8 \* sizeof(Vertex),  D3DUSAGE\_WRITEONLY,  Vertex::FVF,  D3DPOOL\_MANAGED,  &VB,  0);  Device->CreateIndexBuffer(  36 \* sizeof(WORD),  D3DUSAGE\_WRITEONLY,  D3DFMT\_INDEX16,  D3DPOOL\_MANAGED,  &IB,  0);  // 向立方体的顶点缓存填充数据  Vertex\* vertices;  VB->Lock(0, 0, (void\*\*)&vertices, 0);  // vertices of a unit cube  vertices[0] = Vertex(-1.0f, -1.0f, -1.0f);  vertices[1] = Vertex(-1.0f, 1.0f, -1.0f);  vertices[2] = Vertex( 1.0f, 1.0f, -1.0f);  vertices[3] = Vertex( 1.0f, -1.0f, -1.0f);  vertices[4] = Vertex(-1.0f, -1.0f, 1.0f);  vertices[5] = Vertex(-1.0f, 1.0f, 1.0f);  vertices[6] = Vertex( 1.0f, 1.0f, 1.0f);  vertices[7] = Vertex( 1.0f, -1.0f, 1.0f);  VB->Unlock();  // 定义立方体的三角形  WORD\* indices = 0;  IB->Lock(0, 0, (void\*\*)&indices, 0);  // 前面  indices[0] = 0; indices[1] = 1; indices[2] = 2;  indices[3] = 0; indices[4] = 2; indices[5] = 3;  // 背面  indices[6] = 4; indices[7] = 6; indices[8] = 5;  indices[9] = 4; indices[10] = 7; indices[11] = 6;  // 左面  indices[12] = 4; indices[13] = 5; indices[14] = 1;  indices[15] = 4; indices[16] = 1; indices[17] = 0;  // 右面  indices[18] = 3; indices[19] = 2; indices[20] = 6;  indices[21] = 3; indices[22] = 6; indices[23] = 7;  // 顶部  indices[24] = 1; indices[25] = 5; indices[26] = 6;  indices[27] = 1; indices[28] = 6; indices[29] = 2;  // 底部  indices[30] = 4; indices[31] = 0; indices[32] = 3;  indices[33] = 4; indices[34] = 3; indices[35] = 7;  IB->Unlock();  // 照相机位置（视图矩阵）  D3DXVECTOR3 position(0.0f, 0.0f, -5.0f);  D3DXVECTOR3 target(0.0f, 0.0f, 0.0f);  D3DXVECTOR3 up(0.0f, 1.0f, 0.0f);  D3DXMATRIX V;  D3DXMatrixLookAtLH(&V, &position, &target, &up);  Device->SetTransform(D3DTS\_VIEW, &V);  // 投影矩阵  D3DXMATRIX proj;  D3DXMatrixPerspectiveFovLH(  &proj,  D3DX\_PI \* 0.5f, // 90 - degree  (float)Width / (float)Height,  1.0f,  1000.0f);  Device->SetTransform(D3DTS\_PROJECTION, &proj);  // 渲染状态（填充模式：框架填充）  Device->SetRenderState(D3DRS\_FILLMODE, D3DFILL\_WIREFRAME);  return true;  } |

Display方法有两个任务；它必须更新场景并且紧接着渲染它。既然想旋转立方体，那么我们将对每一帧增加一个角度使立方体能在这一帧旋转。对于这每一帧，立方体将被旋转一个很小的角度，这样我们看起来旋转就会更平滑。接着我们使用IDirect3DDevice9::DrawIndexedPrimitive方法来绘制立方体。

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  // 旋转立方体  D3DXMATRIX Rx, Ry;  // x轴旋转45弧度  D3DXMatrixRotationX(&Rx, 3.14f / 4.0f);  // 每一帧中增加y轴的弧度  static float y = 0.0f;  D3DXMatrixRotationY(&Ry, y);  y += timeDelta;  //当y轴旋转2周时，重新回到0弧度  if( y >= 6.28f )  y = 0.0f;  // 结合x轴与y轴的旋转矩阵  D3DXMATRIX p = Rx \* Ry;  Device->SetTransform(D3DTS\_WORLD, &p);  // 清空目标缓存和深度缓存（用0xffffffff, 1.0f）  Device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER,  0xffffffff, 1.0f, 0);    Device->BeginScene(); // 开始绘制场景  Device->SetStreamSource(0, VB, 0, sizeof(Vertex)); // 设置资源流  Device->SetIndices(IB); // 设置索引缓存  Device->SetFVF(Vertex::FVF); // 设置顶点格式  // 利用索引缓存绘制  Device->DrawIndexedPrimitive(D3DPT\_TRIANGLELIST, 0, 0, 8, 0, 12);  Device->EndScene(); // 结束绘制场景  Device->Present(0, 0, 0, 0); // 翻转表面  }  return true;  } |

最后，我们释放使用过的所有内存。这意味着释放顶点和索引缓存接口：

|  |
| --- |
| void Cleanup()  {  d3d::Release<IDirect3DVertexBuffer9\*>(VB);  d3d::Release<IDirect3DIndexBuffer9\*>(IB);  } |

## 3.7 摘要（略）

# 第四章 色彩 (Color)

在上一章中我们用线框模式渲染了场景中的物体。这一章我们将学习怎样渲染有颜色的物体。

## 目标

* 学习在Direct3D中怎样描述颜色
* 弄懂怎样给三角形赋予颜色

## 颜色表示法

在Direct3D中，颜色是使用RGB三部分来描述的。也就是说，我们要分别指定红、绿和蓝三种颜色的值。混合这三个颜色决定最终的颜色。利用这三种颜色我们能够表现数万种颜色。

我们使用两种不同的结构来存储RGB数据。这第一种是D3DCOLOR，它实际上一个DWORD即32位。在D3DCOLOR类型中的这些位按照8-bit被分为4个部分，每一部分存储的是该色的亮度值。如图4.1所示。

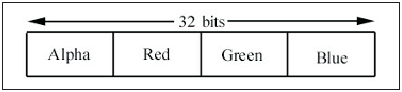


图4.1

每种颜色占用内存的一个字节，各颜色亮度值的取值范围是0-255。这个值越接近0就越暗，越接近255就越亮。

注意：现在不要管alpha部分；它被用在alpha混合中——在第7章中会讲解。

指定其中的每一部分并且把它放到D3DCOLOR中适当的位置需要使用到一些位操作。Direct3D为我们提供了一个完成这个任务的宏D3DCOLOR\_ARGB.它使用包含每种颜色以及alpha位一共4个参数。每一个参数的取值必须在0-255之间，如：

|  |
| --- |
| D3DCOLOR brightRed = D3DCOLOR\_ARGB(255, 255, 0, 0);  D3DCOLOR someColor = D3DCOLOR\_ARGB(255, 144, 87, 201); |

另外，我们也能使用D3DCOLOR\_XRGB宏，它与刚才的宏类似只不过不必指定alpha部分；不过我们最好还是把alpha指定为0xff（255）。

|  |
| --- |
| #define D3DCOLOR\_XRGB(r,g,b) D3DCOLOR\_ARGB(0xff,r,g,b) |

在Direct3D中另外一种存储颜色的结构是D3DCOLORVALUE。在这个结构中，我们分别使用一个浮点数来表示每一部分的亮度值。其取值范围是0-1，0表示没有亮度，1表示最大亮度。

|  |
| --- |
| typedef struct \_D3DCOLORVALUE {  float r; // the red component, range 0.0-1.0  float g; // the green component, range 0.0-1.0  float b; // the blue component, range 0.0-1.0  float a; // the alpha component, range 0.0-1.0  } D3DCOLORVALUE; |

另外，我们能够使用D3DXCOLOR结构，就象D3DCOLORVALUE包含同样的数据成员一样。同时提供有用的构造函数和重载操作符，这将让颜色处理更容易。D3DXCOLOR的定义如下：

|  |
| --- |
| typedef struct D3DXCOLOR  {  #ifdef \_\_cplusplus  public:  D3DXCOLOR() {}  D3DXCOLOR( DWORD argb );  D3DXCOLOR( CONST FLOAT \* );  D3DXCOLOR( CONST D3DXFLOAT16 \* );  D3DXCOLOR( CONST D3DCOLORVALUE& );  D3DXCOLOR( FLOAT r, FLOAT g, FLOAT b, FLOAT a );  // casting  operator DWORD () const;  operator FLOAT\* ();  operator CONST FLOAT\* () const;  operator D3DCOLORVALUE\* ();  operator CONST D3DCOLORVALUE\* () const;  operator D3DCOLORVALUE& ();  operator CONST D3DCOLORVALUE& () const;  // assignment operators  D3DXCOLOR& operator += ( CONST D3DXCOLOR& );  D3DXCOLOR& operator -= ( CONST D3DXCOLOR& );  D3DXCOLOR& operator \*= ( FLOAT );  D3DXCOLOR& operator /= ( FLOAT );  // unary operators  D3DXCOLOR operator + () const;  D3DXCOLOR operator - () const;  // binary operators  D3DXCOLOR operator + ( CONST D3DXCOLOR& ) const;  D3DXCOLOR operator - ( CONST D3DXCOLOR& ) const;  D3DXCOLOR operator \* ( FLOAT ) const;  D3DXCOLOR operator / ( FLOAT ) const;  friend D3DXCOLOR operator \* (FLOAT, CONST D3DXCOLOR& );  BOOL operator == ( CONST D3DXCOLOR& ) const;  BOOL operator != ( CONST D3DXCOLOR& ) const;  #endif //\_\_cplusplus  FLOAT r, g, b, a;  } D3DXCOLOR, \*LPD3DXCOLOR; |

注意：D3DCOLORVALUE和D3DXCOLOR结构都有4个浮点数成员。这使我们的颜色处理符号能象4D向量一样。颜色向量能被加，减以及缩放。另一方面点积和叉积不能用于颜色向量，但是颜色成员相乘是可以的。因此在D3DXCOLOR类中执行的乘法就是成员相乘。它的定义如下：



现在使用下面全局颜色常量更新我们的d3dUtility.h文件：

|  |
| --- |
| namespace d3d  {  .  .  .  const D3DXCOLOR WHITE( D3DCOLOR\_XRGB(255, 255, 255) );  const D3DXCOLOR BLACK( D3DCOLOR\_XRGB( 0, 0, 0) );  const D3DXCOLOR RED( D3DCOLOR\_XRGB(255, 0, 0) );  const D3DXCOLOR GREEN( D3DCOLOR\_XRGB( 0, 255, 0) );  const D3DXCOLOR BLUE( D3DCOLOR\_XRGB( 0, 0, 255) );  const D3DXCOLOR YELLOW( D3DCOLOR\_XRGB(255, 255, 0) );  const D3DXCOLOR CYAN( D3DCOLOR\_XRGB( 0, 255, 255) );  const D3DXCOLOR MAGENTA( D3DCOLOR\_XRGB(255, 0, 255) );  } |

## 4.2 顶点颜色

图元的颜色是由构成它的顶点的颜色决定的。因此，我们必须把一个颜色成员加入到我们的顶点数据结构中。注意D3DCOLORVALUE类型不能用在这里，因为Direct3D希望用一个32位的值来描述顶点的颜色。（通过使用顶点着色器我们能为顶点颜色使用4D颜色向量，它能提供一个128位的颜色，但是对于我们现在的水平来说那太超前了。顶点着色器将在17章中介绍。）

|  |
| --- |
| struct ColorVertex  {  float \_x, \_y, \_z;  D3DCOLOR \_color;  static const DWORD FVF;  }  const DWORD ColorVertex::FVF = D3DFVF\_XYZ | D3DFVF\_DIFFUSE; |

## 4.3 着色处理

着色处理发生在光栅化和指定图元上的顶点颜色怎样被计算成像素颜色之间。目前这里有2种着色处理模式可用：平面着色（flat shading）和高洛德着色（Gouraud shading）。

平面着色，图元像素的颜色是均匀的，且就是指定图元第一个顶点的颜色。因此一旦三角形的第一个顶点被指定成红色，那么它的其他三个顶点也将会是红色。通过使用平面着色来为第二和第三个顶点着色。

|  |
| --- |
| ColorVertex t[3];  t[0].\_color = D3DCOLOR\_XRGB(255, 0, 0);  t[1].\_color = D3DCOLOR\_XRGB(0, 255, 0);  t[2].\_color = D3DCOLOR\_XRGB(0, 0, 255); |

平面着色使物体呈现是斑驳的，因为没有从一个颜色到另一个颜色的平滑过渡。一个更好的着色模式叫做高洛德着色（也被叫做平滑着色）。高洛德着色，图元表面的颜色是由每个顶点通过线性插值来赋予。图4.2显示了分别使用平面着色和高洛德着色处理的红色三角形。

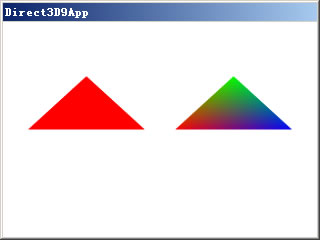


图4.2

就象Direct3D中很多东西一样，着色处理模式是受Direct3D设置状态决定的。

|  |
| --- |
| // set flat shading  Device->SetRenderState(D3DRS\_SHADEMODE, D3DSHADE\_FLAT);  // set Gouraud shading  Device->SetRenderState(D3DRS\_SHADEMODE, D3DSHADE\_GOURAUD); |

## 4.4 实例程序：彩色三角形

这个实例程序展示了分别使用本章中的平面着色和高洛德着色处理的三角形。渲染出的图片如图4.2所示。首先我们定义如下的全局变量：

|  |
| --- |
| D3DXMATRIX World;  IDirect3DVertexBuffer9\* Triangle = 0; |

我们包含一个D3DXMATRIX，它将存储我们将要绘制的三角形在世界坐标中的变换信息。Triangle变量是存储三角形顶点数据的顶点缓存。注意，我们只需要存储一个三角形，因为我们能用它在世界坐标系中不同位置绘制若干次。

Setup方法创建顶点缓存同时填充上带颜色信息的三角形顶点数据。三角形的第一个顶点填充为全亮度红色（255）第二个填充全亮度绿色（255），第三个填充全亮度蓝色（255）。最后，在这个例子中我们屏蔽掉灯光。值得注意的是该例子使用的是一个新的ColorVertex结构，就象在4.2节中说明的一样。

|  |
| --- |
| bool Setup()  {  // create vertex buffer  Device->CreateVertexBuffer(  3 \* sizeof(ColorVertex),  D3DUSAGE\_WRITEONLY,  ColorVertex::FVF,  D3DPOOL\_MANAGED,  &Triangle,  0);    // fill the buffers with the triangle data  ColorVertex\* v;  Triangle->Lock(0, 0, (void\*\*)&v, 0);  v[0] = ColorVertex(-1.0f, 0.0f, 2.0f, D3DCOLOR\_XRGB(255, 0, 0));  v[1] = ColorVertex( 0.0f, 1.0f, 2.0f, D3DCOLOR\_XRGB( 0, 255, 0));  v[2] = ColorVertex( 1.0f, 0.0f, 2.0f, D3DCOLOR\_XRGB( 0, 0, 255));  Triangle->Unlock();  // set projection matrix  D3DXMATRIX proj;  D3DXMatrixPerspectiveFovLH(  &proj,  D3DX\_PI \* 0.5f, // 90 - degree  (float)Width / (float)Height,  1.0f,  1000.0f);  Device->SetTransform(D3DTS\_PROJECTION, &proj);  // set the render states  Device->SetRenderState(D3DRS\_LIGHTING, false);  return true;  } |

Display函数使用不同的着色模式在两个不同的地方分别绘制2个Triangle。每个三角形的位置由世界矩阵World来决定。

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  Device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER, 0xffffffff, 1.0f, 0);  Device->BeginScene();  Device->SetFVF(ColorVertex::FVF);  Device->SetStreamSource(0, Triangle, 0, sizeof(ColorVertex));  // draw the triangle to the left with flat shading  D3DXMatrixTranslation(&World, -1.25f, 0.0f, 0.0f);  Device->SetTransform(D3DTS\_WORLD, &World);  Device->SetRenderState(D3DRS\_SHADEMODE, D3DSHADE\_FLAT);  Device->DrawPrimitive(D3DPT\_TRIANGLELIST, 0, 1);  // draw the triangle to the right with gouraud shading  D3DXMatrixTranslation(&World, 1.25f, 0.0f, 0.0f);  Device->SetTransform(D3DTS\_WORLD, &World);  Device->SetRenderState(D3DRS\_SHADEMODE, D3DSHADE\_GOURAUD);  Device->DrawPrimitive(D3DPT\_TRIANGLELIST, 0, 1);  Device->EndScene();  Device->Present(0, 0, 0, 0);  }  return true;  } |

## 4.5 摘要(略)

# 第五章 灯光 (Lighting)

为了提高场景的真实性，我们可以为其加入灯光。灯光也能帮助表现物体的立体感以及物体的体积。当使用灯光时，我们不再自己指定顶点的颜色；Direct3D中每个顶点都通过灯光引擎来计算顶点颜色，该计算是基于定义的灯光资源，材质以及灯光资源关心的表面方向。通过灯光模型计算顶点颜色会得到更真实的场景。

## 目标

* 学习Direct3D支持的灯光资源，以及它们照射出的灯光类型。
* 弄懂怎样定义灯光去影响其照射的表面。
* 找出怎样算术描述三角形的方向以便我们能够确定灯光照射到三角形的角度。

## 5.1灯光的组成

在Direct3D灯光模型中，灯光是通过灯光资源的三个成员之一来照射的，即有三种灯光。

* 环境光（Ambient Light）——这种类型的灯光将被其他所有表面反射且被用在照亮整个场景。例如，物体的各部分都被照亮，对于一个角度，甚至穿过不在光源直接照射的地方他们都能被照亮。环境光的使用是粗略的，便宜的，它模仿反射光。
* 漫反射（Diffuse Reflection）——这种灯光按照特殊方向传播。当它照射到一个表面，它将在所有方向上均匀的反射。因为漫射光在所有方向上都均匀的反射，被反射的光线将到达眼睛而与观察点无关，因此我们不必为观察者考虑。因而，漫射光仅仅需要考虑灯光方向和表面的姿态。这种灯光将成为你的资源中照射的普通灯光。
* 镜面反射（Specular Reflection）——这种灯光按照特殊方向传播。当它照射到一个表面时，它严格地按照一个方向反射。这将产生一个明亮的光泽，它能在某角度被看见。因为这种灯光在一个方向反射。明显的观察点，必须考虑灯光的方向和表面姿态，且必须按照镜面灯光等式来考虑。镜面灯光被用在物体上产生高光的地方，这种光泽只有在灯光照射在磨光的表面上才会产生。

镜面光比其他灯光类型要求更多的计算；因此，Direct3D提供了一个开关选择。实际上，它默认是被关闭的；要使用镜面光你必须设置D3DRS\_SPECULARENABLE渲染状态。

|  |
| --- |
| Device->SetRenderState(D3DRS\_SPECULARENABLE, true); |

每一种灯光都是通过D3DCOLORVALUE结构或者描述灯光颜色的D3DXCOLOR来描绘的。这里有几个灯光颜色的例子：

|  |
| --- |
| D3DXCOLOR redAmbient(1.0f, 0.0f, 0.0f, 1.0f);  D3DXCOLOR blueDiffuse(0.0f, 0.0f, 1.0f, 1.0f);  D3DXCOLOR whiteSpecular(1.0f, 1.0f, 1.0f, 1.0f); |

注意：在D3DXCOLOR类中的alpha值用在描述灯光颜色时是被忽略的。

## 5.2材质

在现实世界中我们看到的物体颜色将由物体反射回来的灯光颜色来决定。比如，一个红色的球是红色的，因为它吸收所有的灯光颜色除了红色光。红色光是被球反射回来进入我们眼睛的，因此我们看到的球是红色的。Direct3D通过我们定义的物体材质来模拟这些所有的现象。材质允许我们定义表面反射灯光的百分比。在代码中通过D3DMATERIAL9结构描述一个材质。

|  |
| --- |
| typedef struct \_D3DMATERIAL9 {  D3DCOLORVALUE Diffuse, Ambient, Specular, Emissive;  float Power;  } D3DMATERIAL9; |

* Diffuse——指定此表面反射的漫射光数量。
* Ambient——指定此表面反射的环境光数量。
* Specular——指定此表面反射的镜面光数量
* Emissive——这个是被用来给表面添加颜色，它使得物体看起来就象是它自己发出的光一样。
* Power——指定锐利的镜面高光；它的值是高光的锐利值。

举例，想得到一个红色的球。我们将定义球的材质来只反射红光吸收其他颜色的所有光：

|  |
| --- |
| D3DMATERIAL9 red;  ::ZeroMemory(&red, sizeof(red));  red.Diffuse = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); // red  red.Ambient = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); // red  red.Specular = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f); // red  red.Emissive = D3DXCOLOR(0.0f, 0.0f, 0.0f, 1.0f); // no emission  red.Power = 5.0f; |

这里我们设置绿色和蓝色的值为0，这表明材质反射0%此颜色的光。我们设置红色为1，表示材质反射100%的红光。注意，我们能够控制每种灯光反射的颜色（环境、漫射和镜面光）。

同样假如我们定义一个只发出蓝色光的光源，对球的光照将失败因为蓝色光将被全部吸收而没有红光被反射。当物体吸收了所有光以后，物体看起来就为黑色。同样的，当物体反射100%的红、绿和蓝光，物体就将呈现为白色。

因为手工填充一个材质结构将是乏味的工作，我们添加下列有用的函数和全局材质常数到d3dUtility.h/cpp文件中：

|  |
| --- |
| D3DMATERIAL9 d3d::InitMtrl(D3DXCOLOR a, D3DXCOLOR d,  D3DXCOLOR s, D3DXCOLOR e, float p)  {  D3DMATERIAL9 mtrl;  mtrl.Ambient = a;  mtrl.Diffuse = d;  mtrl.Specular = s;  mtrl.Emissive = e;  mtrl.Power = p;  return mtrl;  }  namespace d3d  {  .  .  .  D3DMATERIAL9 InitMtrl(D3DXCOLOR a, D3DXCOLOR d, D3DXCOLOR s, D3DXCOLOR e, float p);  const D3DMATERIAL9 WHITE\_MTRL = InitMtrl(WHITE, WHITE, WHITE, BLACK, 8.0f);  const D3DMATERIAL9 RED\_MTRL = InitMtrl(RED, RED, RED, BLACK, 8.0f);  const D3DMATERIAL9 GREEN\_MTRL = InitMtrl(GREEN, GREEN, GREEN, BLACK, 8.0f);  const D3DMATERIAL9 BLUE\_MTRL = InitMtrl(BLUE, BLUE, BLUE, BLACK, 8.0f);  const D3DMATERIAL9 YELLOW\_MTRL = InitMtrl(YELLOW, YELLOW, YELLOW, BLACK, 8.0f);  } |

顶点结构没有材质属性；一个通用的材质必须被设置。设置它我们使用IDirect3DDevice9::SetMaterial(CONST D3DMATERIAL9\*pMaterial)方法。

假设我们想渲染几个不同材质的物体；我们将按照如下的写法去做：

|  |
| --- |
| D3DMATERIAL9 blueMaterial, redMaterial;  ...// set up material structures  Device->SetMaterial(&blueMaterial);  drawSphere(); // blue sphere  Device->SetMaterial(&redMaterial);  drawSphere(); // red sphere |

## 5.3顶点法线

面法线（*face normal*）是描述多边形表面方向的一个向量（如图5.1）。

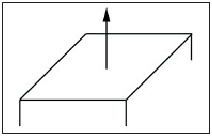


图5.1

顶点法线（*Vertex normals*）也是基于同样的概念，但是我们与其指定每个多边形的法线，还不如为每个顶点指定（如图5.2）。

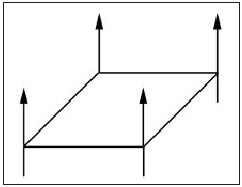


图5.2

Direct3D需要知道顶点法线以便它能够确定灯光照射到物体表面的角度，并且一旦计算了每个顶点的灯光，Direct3D需要知道每个顶点的表面方向。注意顶点法线不一定和面法线相同。球体/环形物就是很好的实物例子，它们的顶点法线和三角形法线是不相同的（如图5.3）。

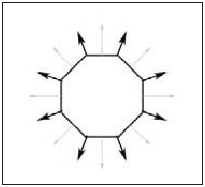


图5.3

为了描述顶点的顶点法线，我们必须更新原来的顶点结构：：

|  |
| --- |
| struct Vertex  {  float \_x, \_y, \_z;  float \_nx, \_ny, \_nz;  static const DWORD FVF;  }  const DWORD Vertex::FVF = D3DFVF\_XYZ | D3DFVF\_NORMAL; |

注意，我们已经将上一章中使用的颜色成分去除了。这是因为我们将使用灯光来计算顶点的颜色。

作为一个简单的物体比如立方体和球体，我们能够通过观察看见顶点法线。对于更多复杂的网格，我们需要一个更多的机械方法。假设一个由p0,p1,p2构成的三角形，我们需要计算每个顶点的法线n0,n1,n2。

简单的步骤，我们列举它是为了找到由三个点构成的三角形的面法线，同时使用面法线作为顶点法线。首先计算三角形上的两个向量：



那么面法线是：



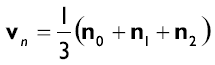
每个顶点的法线和面法线是相等的：



下面是一个C函数，它通过三角形的三个顶点计算三角形的面法线。注意这个函数的三个顶点是按照顺时针方向指定的。假如不是这样，那么法线方向将是相反的。

|  |
| --- |
| void ComputeNormal(D3DXVECTOR3\* p0,  D3DXVECTOR3\* p1,  D3DXVECTOR3\* p2,  D3DXVECTOR3\* out)  {  D3DXVECTOR3 u = \*p1 - \*p0;  D3DXVECTOR3 v = \*p2 - \*p0;  D3DXVec3Cross(out, &u, &v);  D3DXVec3Normalize(out, out);  } |

当用三角形近似表示曲面时，使用面法线作为顶点法线不能表现一个平滑的结果。一个更好的方法是找到顶点法线的平均法线。为了找到顶点v的顶点法线vn，我们找到网格模型中所有三角形的面法线记为顶点v。vn是通过计算他们的平均面法线得到的。这里有一个例子，假设有3个三角形它们的面法线分别是n0，n1，n2，指定为顶点v。那么vn的平均法线就是：



通过改变“舞台”，把顶点法线变为non-normal,这是有可能的。因此这样最好是安全的且在通过D3DRS\_NORMALIZENORMALS设置渲染状态来改变“舞台”后，Direct3D从新规格化所有法线。

|  |
| --- |
| Device->SetRenderState(D3DRS\_NORMALIZENORMALS, true); |

## 5.4光源

Direct3D支持三种类型的光源。

* 点光源——这种光源在世界坐标中有一个位置且向所有方向上都照射光线。

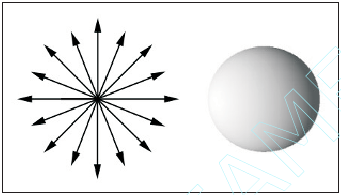


图5.4

* 方向光源——这种光源没有位置但是向指定方向发出平行光线。

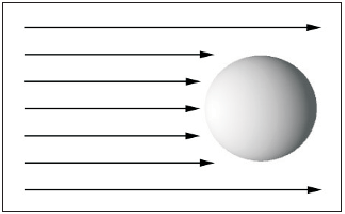


图5.5

* 聚光灯——这种类型的光源和手电筒的光类似；它有位置并且发出的光在指定方向上按照圆锥形照射。这个圆锥形有两个角度，θ和φ。角度θ描述内圆锥，φ描述外圆锥。

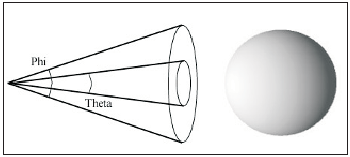


图5.6

在代码中一个灯光资源是通过D3DLIGHT9结构来表现的。

|  |
| --- |
| typedef struct \_D3DLIGHT9 {  D3DLIGHTTYPE Type;  D3DCOLORVALUE Diffuse;  D3DCOLORVALUE Specular;  D3DCOLORVALUE Ambient;  D3DVECTOR Position;  D3DVECTOR Direction;  float Range;  float Falloff;  float Attenuation0;  float Attenuation1;  float Attenuation2;  float Theta;  float Phi;  } D3DLIGHT9; |

* Type——定义灯光类型，我们能够使用下面三种类型之一：D3DLIGHT\_POINT, D3DLIGHT\_SPOT, D3DLIGHT\_DIRECTIONAL
* Diffuse——此光源发出的漫射光颜色。
* Specular——此光源发出的镜面光颜色。
* Ambient——此光源发出的环境光颜色。
* Position——用一个向量来描述的光源世界坐标位置。这个值对于灯光的方向是无意义的。
* Direction——用一个向量来描述的光源世界坐标照射方向。这个值不能用在点光源上。
* Range——灯光能够传播的最大范围。这个值不能比大。且不能用于方向光源。
* Falloff——这个值只能用在聚光灯上。它定义灯光在从内圆锥到外圆锥之间的强度衰减。它的值通常设置为1.0f。
* Attenuation0, Attenuation1, Attenuation2——这些衰减变量被用来定义灯光强度的传播距离衰减。它们只被用于点光源和聚光灯上。Attenuation0定义恒定衰减，Attenuation1定义线性衰减，Attenuation2定义二次衰减。适当的使用这个公式，D是代表到光源的距离，A0,A1,A2与Attenuation0，1，2相匹配。



* Theta——只用于聚光灯；指定内圆锥的角度，单位是弧度。
* Phi——只用于聚光灯；指定外圆锥的角度，单位是弧度。

就象初始化D3DMATERIAL9结构一样，初始化D3DLIGHT9结构是一件单调乏味的工作。我们添加下面的函数到d3dUtility.h/cpp文件中用于初始化简单灯光。

|  |
| --- |
| namespace d3d  {  … …  D3DLIGHT9 InitDirectionalLight(D3DXVECTOR3\* direction, D3DXCOLOR\* color);  D3DLIGHT9 InitPointLight(D3DXVECTOR3\* position, D3DXCOLOR\* color);  D3DLIGHT9 InitSpotLight(D3DXVECTOR3\* position, D3DXVECTOR3\* direction, D3DXCOLOR\* color);  } |

使用这些函数是非常简单的。我们现在只是演示怎样使用InitDirectionalLight。其他的也很类似：

|  |
| --- |
| D3DLIGHT9 d3d::InitDirectionalLight(D3DXVECTOR3\* direction, D3DXCOLOR\* color)  {  D3DLIGHT9 light;  ::ZeroMemory(&light, sizeof(light));  light.Type = D3DLIGHT\_DIRECTIONAL;  light.Ambient = \*color \* 0.4f;  light.Diffuse = \*color;  light.Specular = \*color \* 0.6f;  light.Direction = \*direction;  return light;  } |

然后创建一个方向光源，它沿着x轴正方向照射白色灯光。我们按照下面的方法来做：

|  |
| --- |
| D3DXVECTOR3 dir(1.0f, 0.0f, 0.0f);  D3DXCOLOR c = d3d::WHITE;  D3DLIGHT9 dirLight = d3d::InitDirectionalLight(&dir, &c); |

在把D3DLIGHT9初始化好以后，我们需要用Direct3D内在支持的灯光来注册。就象这样做：

|  |
| --- |
| Device->SetLight(  0, // element in the light list to set, range is 0-maxlights  &light);// address of the D3DLIGHT9 structure to set |

一旦灯光注册了，我们就能使用下面的列举的例子来开或关灯光了：

|  |
| --- |
| Device->LightEnable(  0, // the element in the light list to enable/disable  true); // true = enable, false = disable |

## 5.5实例程序：灯光

这一章的例子是创建如图5.7所显示的场景。它示范了怎样指定顶点法线，怎样创建材质，以及怎样创建和使用一个方向灯光。注意在这个示例程序中我们不会使用在文件d3dUtility.h/cpp中的材质和灯光函数。因为我们想展示怎样手动来做这些设置。

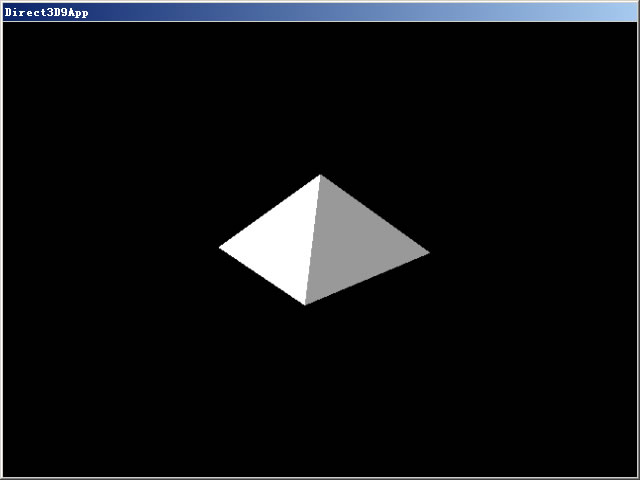


图5.7

给场景增加灯光的步骤是：

1. 允许使用灯光。
2. 为每个物体创建材质并且在渲染相应物体前应将材质附予物体。
3. 创建一个或多个光源，设置它们，把它们设为可用。
4. 将其他附加光源设为可用，比如镜面高光。

首先我们初始化一个全局顶点缓存用他来存储“金字塔”的顶点：

|  |
| --- |
| IDirect3DVertexBuffer9\* Pyramid = 0; |

Setup函数包含本章的所有代码，因此我们忽略其他函数。它执行刚才讨论的步骤来给场景加入灯光。Setup方法首先允许使用灯光，当然这不是必须的因为默认设置就是允许使用灯光的。

|  |
| --- |
| bool Setup()  {  Device->SetRenderState(D3DRS\_LIGHTING, true); |

下一步，我们创建顶点缓存，锁定，并且把“金字塔”的三角形顶点放入其中。顶点法线是利用5.3节中的运算法则预先计算好的。注意三角形共享顶点，但它们的法线不能共享；因此对这个物体使用索引列表并不是最有利的。例如，所有三角形都共享顶点（0,1,0）；然而，对每个三角形，它们的顶点法线是不相同的。

|  |
| --- |
| Device->CreateVertexBuffer(  12 \* sizeof(Vertex),  D3DUSAGE\_WRITEONLY,  Vertex::FVF,  D3DPOOL\_MANAGED,  &Pyramid,  0);  // fill the vertex buffer with pyramid data  Vertex\* v;  Pyramid->Lock(0, 0, (void\*\*)&v, 0);  // front face  v[0] = Vertex(-1.0f, 0.0f, -1.0f, 0.0f, 0.707f, -0.707f);  v[1] = Vertex( 0.0f, 1.0f, 0.0f, 0.0f, 0.707f, -0.707f);  v[2] = Vertex( 1.0f, 0.0f, -1.0f, 0.0f, 0.707f, -0.707f);    // left face  v[3] = Vertex(-1.0f, 0.0f, 1.0f, -0.707f, 0.707f, 0.0f);  v[4] = Vertex( 0.0f, 1.0f, 0.0f, -0.707f, 0.707f, 0.0f);  v[5] = Vertex(-1.0f, 0.0f, -1.0f, -0.707f, 0.707f, 0.0f);  // right face  v[6] = Vertex( 1.0f, 0.0f, -1.0f, 0.707f, 0.707f, 0.0f);  v[7] = Vertex( 0.0f, 1.0f, 0.0f, 0.707f, 0.707f, 0.0f);  v[8] = Vertex( 1.0f, 0.0f, 1.0f, 0.707f, 0.707f, 0.0f);  // back face  v[9] = Vertex( 1.0f, 0.0f, 1.0f, 0.0f, 0.707f, 0.707f);  v[10] = Vertex( 0.0f, 1.0f, 0.0f, 0.0f, 0.707f, 0.707f);  v[11] = Vertex(-1.0f, 0.0f, 1.0f, 0.0f, 0.707f, 0.707f);  Pyramid->Unlock(); |

为物体产生了顶点数据以后，我们描述利用灯光表现各自材质的物体间是怎样相互影响的。在这个例子中，“金字塔”反射出白光，自身不发光，且会产生一些高光。

|  |
| --- |
| D3DMATERIAL9 mtrl;  mtrl.Ambient = d3d::WHITE;  mtrl.Diffuse = d3d::WHITE;  mtrl.Specular = d3d::WHITE;  mtrl.Emissive = d3d::BLACK;  mtrl.Power = 5.0f;  Device->SetMaterial(&mtrl); |

接着，我们创建一个方向光并将其设为可用。方向光是沿着x轴的正方向照射的。灯光照射最强的白色漫射光（dir.Diffuse = WHITE），较弱的白色镜面光（dir.Specular = WHITE \* 0.3f）以及一个中等强度的白色环境光（dir.Ambient = WHITE \*0.6f）。

|  |
| --- |
| D3DLIGHT9 dir;  ::ZeroMemory(&dir, sizeof(dir));  dir.Type = D3DLIGHT\_DIRECTIONAL;  dir.Diffuse = d3d::WHITE;  dir.Specular = d3d::WHITE \* 0.3f;  dir.Ambient = d3d::WHITE \* 0.6f;  dir.Direction = D3DXVECTOR3(1.0f, 0.0f, 0.0f);  Device->SetLight(0, &dir);  Device->LightEnable(0, true); |

最后，我们设置状态使法线从新规格化且把镜面高光设置为可用。

|  |
| --- |
| Device->SetRenderState(D3DRS\_NORMALIZENORMALS, true);  Device->SetRenderState(D3DRS\_SPECULARENABLE, true);  // ... code to set up the view matrix and projection matrix  // omitted  return true;  } |

## 5.6附加实例

这一章中还有三个附加的例子。它们使用D3DXCreate\*函数来创建组成场景的3D物体。D3DXCreate\*函数创建的顶点数据是D3DFVF\_XYZ | D3DFVF\_NORMAL格式。在增加的函数中为我们的网格模型的每个顶点计算了顶点法线。这些实例演示了怎样使用方向光，点光源，以及聚光灯。图5.8显示的是方向光实例中的一个场景图。

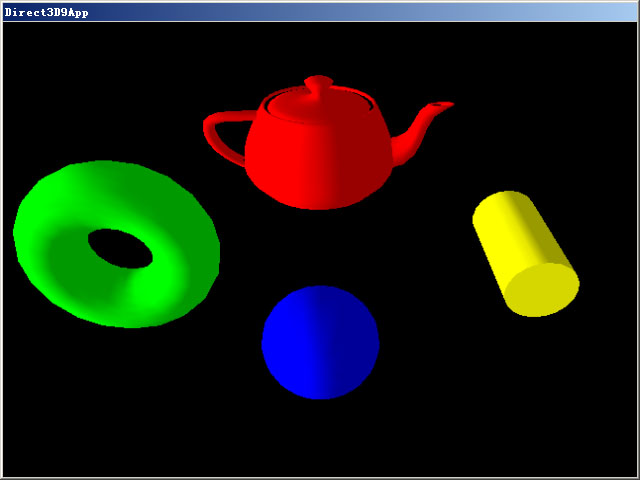


图5.8

## 5.7摘要(略)

# 第六章 纹理 (Texturing)

纹理映射是一种允许我们为三角形赋予图象数据的技术；这让我们能够更细腻更真实地表现我们的场景。例如，我们能够创建一个立方体并且通过对它的每个面创建一个纹理来把它变成一个木箱（如图6.1）。

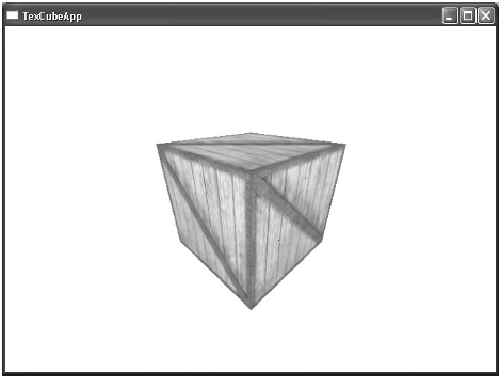


图6.1

在Direct3D中一个纹理是通过IDirect3DTexture9接口来表现的。一个纹理是一个类似像素矩阵的表面它能够被映射到三角形上。

## 目标

* 学习怎样指定纹理到三角形上。
* 弄懂怎样创建一纹理。
* 学习怎样通过过滤纹理来创建一个更光滑的图象。

## 6.1 纹理坐标

Direct3D使用一个纹理坐标系统，它是由用水平方向的u轴和竖直方向v轴构成。由u,v坐标决定纹理上的元素，它被叫做*texel*。注意v轴是向下的（如图6.2）。



图6.2

同样，注意规格化的坐标间隔，[0，1]，它被使用是因为它给Direct3D一个固定的范围用于在不同尺寸的纹理上工作。

对每一个3D三角形，我们都希望在给它贴图的纹理上定义一个用相应的三角形。（如图6.3）。

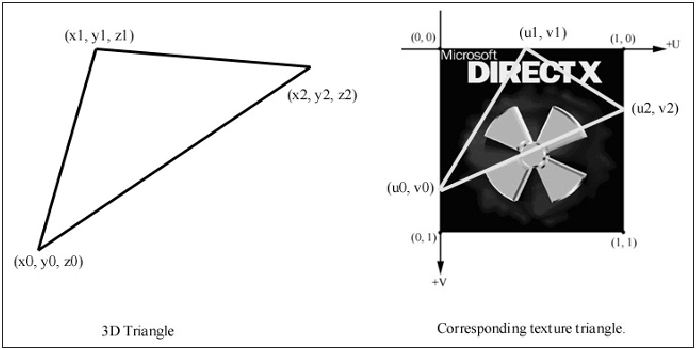


图6.3

我们再一次修改原来的顶点结构，添加一个用于在纹理上定位的纹理坐标。

|  |
| --- |
| struct Vertex  {  float \_x, \_y, \_z;  float \_nx, \_ny, \_nz;  float \_u, \_v; // texture coordinates  static const DWORD FVF;  };  const DWORD Vertex::FVF = D3DFVF\_XYZ | D3DFVF\_NORMAL | D3DFVF\_TEX1; |

我们在顶点格式中添加了一个D3DFVF\_TEX1，它是说我们的顶点结构中包含了一个纹理坐标。

现在每个三角形都通过顶点的三个对象来建立，同时也通过纹理坐标定义了一个相应的纹理三角形。

## 6.2创建并赋予材质

纹理数据通常是从存储在磁盘中的图片文件中读取的，且被读进IDirect3DTexture9对象中。我们能够使用下面的D3DX函数完成这项工作：

|  |
| --- |
| HRESULT D3DXCreateTextureFromFile(  LPDIRECT3DDEVICE9 pDevice, // device to create the texture  LPCSTR pSrcFile, // filename of image to load  LPDIRECT3DTEXTURE9\* ppTexture // ptr to receive the created texture  ); |

这个函数能够读取下面图片格式中的任意一种：BMP,DDS,DIB,JPG,PNG,TGA。

例如，用一个名为stonewall.bmp的图片创建一个纹理，我们将按照下面的例子来写：

|  |
| --- |
| IDirect3Dtexture9\* \_stonewall;  D3DXCreateTextureFromFile(\_device, "stonewall.bmp", &\_stonewall); |

设置当前纹理，我们使用下面的方法：

|  |
| --- |
| HRESULT IDirect3DDevice9::SetTexture(  DWORD Stage, // A value in the range 0-7 identifying the texture  // stage – see note on Texture Stages  IDirect3DBaseTexture9\* pTexture // ptr to the texture to set  ); |

例子：

|  |
| --- |
| Device->SetTexture(0, \_stonewall); |

注意：在Direct3D中，你能够设置八个纹理，它们能够组合起来创建更多细节的图象。这又被叫做多重纹理。在本书的第四部分以前我们不会使用多重纹理；因此现在我们总是设置stage为0。

为了销毁一个纹理，我们设置pTexture为0。例如，假如不想用一个纹理来渲染物体，那么我们就这样写：

|  |
| --- |
| Device->SetTexture(0, 0);  renderObjectWithoutTexture(); |

假如场景中有使用不同纹理的三角形，我们就必须添加与下面类似的一些代码：

|  |
| --- |
| Device->SetTexture(0, \_tex0);  drawTrisUsingTex0();  Device->SetTexture(0, \_tex1);  drawTrisUsingTex1(); |

## 6.3过滤器

就象以前提及的，纹理被映射到屏幕中的三角形上。通常纹理三角形和屏幕三角形是不一样大的。当纹理三角形比屏幕三角形小时，纹理三角形会被适当放大。当纹理三角形比屏幕三角形大时，纹理三角形会被适当缩小。这两种情况，变形都将会出现。过滤（*Filtering*）是一种Direct3D用它来帮助这些变形变的平滑的技术。

Direct3D提供了三种不同的过滤器；每种都提供了一个不同的品质级别。越好的品质越慢，因此你必须在品质与速度之间取得一个平衡。纹理过滤器是用IDirect3DDevice9::SetSamplerState方法来设置的。

* **Nearest point sampling**——这是默认的过滤方法且返回最差的效果，但是它的计算是最快的。下面的代码就是设置**Nearest point sampling**作为缩小放大的过滤器：

|  |
| --- |
| Device->SetSamplerState(0, D3DSAMP\_MAGFILTER, D3DTEXF\_POINT);  Device->SetSamplerState(0, D3DSAMP\_MINFILTER, D3DTEXF\_POINT); |

* **Linear filtering**——这种过滤产生还算比较好的效果，在今天的硬件上处理它还是非常快的。它是被推荐使用的。下面的代码就是设置**Linear filtering**作为缩小放大的过滤器。

|  |
| --- |
| Device->SetSamplerState(0, D3DSAMP\_MAGFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(0, D3DSAMP\_MINFILTER, D3DTEXF\_LINEAR); |

* **Anisotropic filtering**——这种过滤产生最好的效果，但是处理时间也是最长的。下面的代码就是设置**Anisotropic filtering**作为缩小放大的过滤器。

|  |
| --- |
| Device->SetSamplerState(0, D3DSAMP\_MAGFILTER, D3DTEXF\_ANISOTROPIC);  Device->SetSamplerState(0, D3DSAMP\_MINFILTER, D3DTEXF\_ANISOTROPIC); |

当使用**Anisotropic filtering**时，我们必须设置D3DSAMP\_MAXANISOTROPY等级，它决定处理的质量。该值越高处理的效果越好。检查D3DCAPS9结构确认你的显卡是否支持此功能。下面的代码设置该值为4：

|  |
| --- |
| Device->SetSamplerState(0, D3DSAMP\_MAXANISOTROPY, 4); |

## 6.4 Mipmaps

就象6.3节所说的，在屏幕上的三角形和纹理三角形通常是不一样大的。为了使这个大小差异变小，我们为纹理创建*mipmaps*链。也就是说将一个纹理创建成连续的变小的纹理，但是对它们等级进行定制过滤，因此对我们来说保存细节是很重要的（如图6.4）。

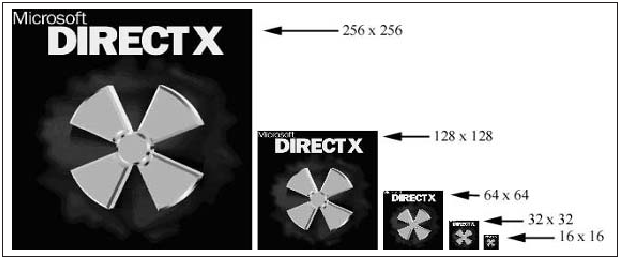


图6.4

### 6.4.1 Mipmaps过滤器

mipmap过滤器是被用来控制Direct3D使用mipmaps的。设置mipmap过滤器，你可以这样写：

|  |
| --- |
| Device->SetSamplerState(0, D3DSAMP\_MIPFILTER, Filter); |

在Filter处你能用下面三个选项中的一个：

* D3DTEXF\_NONE——不使用mipmap。
* D3DTEXF\_POINT——通过使用这个过滤器，Direct3D将选择与屏幕三角形大小最接近的mipmap等级。一旦等级选定了，Direct3D就将按照指定的过滤器进行缩小和放大过滤。
* D3DTEXF\_LINEAR­­——通过使用这个过滤器，Direct3D将选择两个最接近的mipmap等级，缩小和放大过滤每个等级，然后线性联合计算它们两个等级来得到最终的颜色值。

### 6.4.2 Direct3D中使用Mipmaps

在Direct3D中使用Mipmaps是很简单的。假如你的显卡支持Mipmaps，那么使用D3DXCreateTextureFromFile将为你产生一个Mipmap链。Direct3D自动选择与屏幕三角形最匹配的Mipmap。因此Mipmap有非常广泛的应用，且它能被自动设置。

## 6.5 寻址模式

以前，我们规定纹理坐标必须指定在[0，1]之间。从技术上来说这是不正确的；他们能够超出这个范围。纹理坐标也可以在[0，1]的范围之外，它通过Direct3D的寻址模式来定义。这里有四种寻址模式：环绕纹理寻址模式、边框颜色纹理寻址模式、截取纹理寻址模式、镜像纹理寻址模式，这里分别给出了它们的示意图6.5，6.6，6.7，6.8。

图6.5（环绕） 图6.6（边框）

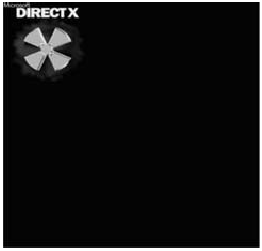
 

图6.7（截取） 图6.8（镜像）

在这些图片中，纹理坐标通过（0,0）（0,3）（3,0）（3,3）顶点来定义。在u轴和v轴上方块又被分成子块放进3×3的矩阵中。假如，你想让纹理按5×5的方格来平铺，你就应该指定环绕纹理寻址模式并且纹理坐标因该设置为（0,0）（0,5）（5,0）（5,5）。

下面的代码片段列举的是怎样设置这四种寻址模式：

|  |
| --- |
| // set wrap address mode  if( ::GetAsyncKeyState('W') & 0x8000f )  {  Device->SetSamplerState(0, D3DSAMP\_ADDRESSU, D3DTADDRESS\_WRAP);  Device->SetSamplerState(0, D3DSAMP\_ADDRESSV, D3DTADDRESS\_WRAP);  }  // set border color address mode  if( ::GetAsyncKeyState('B') & 0x8000f )  {  Device->SetSamplerState(0, D3DSAMP\_ADDRESSU, D3DTADDRESS\_BORDER);  Device->SetSamplerState(0, D3DSAMP\_ADDRESSV, D3DTADDRESS\_BORDER);  Device->SetSamplerState(0, D3DSAMP\_BORDERCOLOR, 0x000000ff);  }  // set clamp address mode  if( ::GetAsyncKeyState('C') & 0x8000f )  {  Device->SetSamplerState(0, D3DSAMP\_ADDRESSU, D3DTADDRESS\_CLAMP);  Device->SetSamplerState(0, D3DSAMP\_ADDRESSV, D3DTADDRESS\_CLAMP);  }  // set mirror address mode  if( ::GetAsyncKeyState('M') & 0x8000f )  {  Device->SetSamplerState(0, D3DSAMP\_ADDRESSU, D3DTADDRESS\_MIRROR);  Device->SetSamplerState(0, D3DSAMP\_ADDRESSV, D3DTADDRESS\_MIRROR);  } |

## 6.6实例程序：有纹理的方块

这个例子是怎样为方块加上纹理以及设置一个纹理过滤器（如图6.9）。假如你的显卡支持，通过D3DXCreateTextureFromFile函数一个mipmap链将被自动创建。



图6.9

注意：还提供了其他两个例子大家就自己看看了。

为一个场景增加纹理的必要步骤是：

1. 用纹理坐标指定的，创建物体的顶点。
2. 用D3DXCreateTextureFromFile函数读取一个纹理到IDirect3DTexture9接口中。
3. 设置缩小倍数，放大倍数以及mipmap过滤器。
4. 在你绘制一个物体前，用IDirect3DDevice9::SetTexture设置与物体关联的纹理。

我们先定义几个全局变量；一个是顶点缓存，它存储方块的顶点。另外一个是我们为方块映射的纹理：

|  |
| --- |
| IDirect3DVertexBuffer9\* Quad = 0;  IDirect3DTexture9\* Tex = 0; |

Setup程序是很容易读懂的；我们用已经定义了纹理坐标的两个三角形创建一个方块。然后把文件dx5\_logo.bmp读进IDirect3DTexture9接口中。接着使用SetTexture方法赋予纹理。最后设置缩小和放大过滤器进行线性过滤，我们也可以设置mipmap过滤器来进行D3DTEXF\_POINT:

|  |
| --- |
| bool Setup()  {  //  // Create the quad vertex buffer and fill it with the  // quad geoemtry.  //  Device->CreateVertexBuffer(  6 \* sizeof(Vertex),  D3DUSAGE\_WRITEONLY,  Vertex::FVF,  D3DPOOL\_MANAGED,  &Quad,  0);  Vertex\* v;  Quad->Lock(0, 0, (void\*\*)&v, 0);  // quad built from two triangles, note texture coordinates:  v[0] = Vertex(-1.0f, -1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f);  v[1] = Vertex(-1.0f, 1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f);  v[2] = Vertex( 1.0f, 1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f);  v[3] = Vertex(-1.0f, -1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f);  v[4] = Vertex( 1.0f, 1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f);  v[5] = Vertex( 1.0f, -1.0f, 1.25f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f);  Quad->Unlock();  //  // Create the texture and set filters.  //  D3DXCreateTextureFromFile(  Device,  "dx5\_logo.bmp",  &Tex);  Device->SetTexture(0, Tex);  Device->SetSamplerState(0, D3DSAMP\_MAGFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(0, D3DSAMP\_MINFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(0, D3DSAMP\_MIPFILTER, D3DTEXF\_POINT);  //  // Don't use lighting for this sample.  //  Device->SetRenderState(D3DRS\_LIGHTING, false);  //  // Set the projection matrix.  //  D3DXMATRIX proj;  D3DXMatrixPerspectiveFovLH(  &proj,  D3DX\_PI \* 0.5f, // 90 - degree  (float)Width / (float)Height,  1.0f,  1000.0f);  Device->SetTransform(D3DTS\_PROJECTION, &proj);  return true;  } |

我们现在可以渲染方块了，且通常已经为它赋予了纹理：

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  Device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER, 0xffffffff, 1.0f, 0);  Device->BeginScene();  Device->SetStreamSource(0, Quad, 0, sizeof(Vertex));  Device->SetFVF(Vertex::FVF);  Device->DrawPrimitive(D3DPT\_TRIANGLELIST, 0, 2);  Device->EndScene();  Device->Present(0, 0, 0, 0);  }  return true;  } |

## 7摘要(略)

# 第七章 混合 (Blending)

在这一章里我们介绍一种叫做混合（blending）的技术，它允许我们混合像素，我们通常用已经光栅化的像素光栅化同一位置的像素。换句话说就是我们在图元上混合图元。这种技术允许我们完成多种特效。

## 目标

* 弄懂怎样混合以及怎样使用它。
* 学习Direct3D支持的不同类型的混合方式。
* 弄懂alpha混合能够被用来控制图元的透明度。

## 7.1混合因素

观察图7.1，我们将一个红色的茶壶绘制在一个木质背景上。

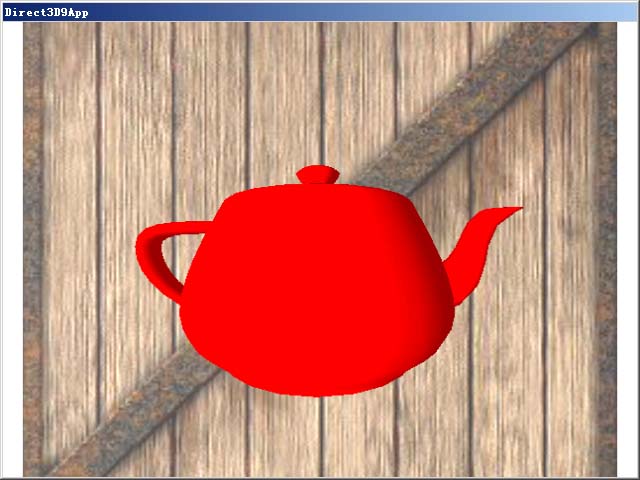


图7.1

假设想让茶壶有一个透明度，以便我们能够透过茶壶看见背景（如图7.2）。



图7.2

我们怎样才能实现这个效果呢？我们只需要在木箱子上光栅化茶壶三角形，我们需要结合像素颜色，就象通过茶壶显示木箱那样来计算茶壶的像素颜色。结合像素值的意思就是用以前写过的目标像素值去估算源像素值这被叫做混合。注意混合的效果不仅仅象是玻璃透明一样。我们有很多选项来指定颜色是怎样被混合的，就象7.2部分中看到的一样。

这是很重要的，认识三角形普遍利用以前写入后缓存中的像素来与之混合来光栅化。在示例图片中，木箱图片首先被画出来且它的像素在后缓存中。我们然后绘制茶壶，以便用木箱的像素来混合茶壶的像素。因此，当使用混合时，下面的规则将被遵循：

规则：首先不使用混合绘制物体。然后根据物体离摄象机的距离使用混合对物体拣选；这是非常有效的处理，假如物体是在视图坐标中，那么你能够利用z分量简单地拣选。最后使用从后到前的顺序混合绘制物体。

下面的公式是用来混合两个像素值的：



上面的所有变量都是一个4D颜色向量（r,g,b,a），并且符号是表示分量相乘。

* *OutputPixel*——混合后的像素结果。
* *SourcePixel*——通常被计算的像素，它是利用在后缓存中的像素来被混合的。
* *SourceBlendFactor*——在[0，1]范围内的一个值。它指定源像素在混合中的百分比。
* *DestPixel*——在后缓存中的像素。
* *DestBlendFactor*——在[0，1]范围内的一个值。它指定目的像素在混合中的百分比。

源和目的混合要素使我们能够按照多种途径改变原始源和目的像素，允许实现不同的效果。7.2节列举了能够被使用的预先确定的值。

混合默认是被关闭的；你能够通过设置D3DRS\_ALPHABLENDENABLE渲染状态为true来开启它：

|  |
| --- |
| Device->SetRenderState(D3DRS\_ALPHABLENDENABLE, true); |

## 7.2混合要素

通过设置不同的源和目的要素，我们能够创造很多不同的混合效果。通过实验，使用不同的组合看看它们到底能实现什么效果。你能够通过设置D3DRS\_SRCBLEND和D3DRS\_DESTBLEND渲染状态来分别设置源混合要素和目的混合要素。例如我们可以这样写：

|  |
| --- |
| Device->SetRenderState(D3DRS\_SRCBLEND, Source);  Device->SetRenderState(D3DRS\_DESTBLEND, Destination); |

这里Source和Destination能够使用下面混合要素中的一个：

* D3DBLEND\_ZERO——*blendFactor*=(0, 0, 0, 0)
* D3DBLEND\_ONE——*blendFactor*=(1, 1, 1, 1)
* D3DBLEND\_SRCCOLOR——*blendFactor*=(rs, gs, bs, as)
* D3DBLEND\_INVSRCCOLOR——*blendFactor*=(1-rs, 1-gs, 1-bs, 1-as)
* D3DBLEND\_SRCALPHA——*blendFactor*=(as, as, as, as)
* D3DBLEND\_INVSRCALPHA——*blendFactor*=(1-as, 1-as, 1-as, 1-as)
* D3DBLEND\_DESTALPHA——*blendFactor*=(ad, ad, ad, ad)
* D3DBLEND\_INVDESTALPHA——*blendFactor*=(1-ad, 1-ad, 1-ad, 1-ad)
* D3DBLEND\_DESTCOLOR——*blendFactor*=(rd, gd, bd, ad)
* D3DBLEND\_INVDESTCOLOR——*blendFactor*=(1-rd, 1-gd, 1-bd, 1-ad)
* D3DBLEND\_SRCALPHASAT——*blendFactor*=(f, f, f, 1) ,  *f*=min(*as*, 1 – *ad*)
* D3DBLEND\_BOTHINVSRCALPHA——这种混合模式设置源混合要素为（1-as, 1-as, 1-as, 1-as,）以及目的混合要素为（as,as,as,as）。这种混合模式仅对D3DRS\_SRCBLEND有效。

源和目的混合要素的默认值分别是D3DBLEND\_SRCALPHA和D3DBLEND\_INVSRCALPHA。

## 7.3透明度

在以前的章节中我们忽略了颜色顶点和材质中的alpha部分，那是因为当时它并不是必须的。现在它首先被用在混合中。

Alpha部分主要是用来指定像素的透明等级。我们为每个像素的alpha部分保留8位，alpha的有效值在[0,255]范围内，[0,255]代表不透明度[0%,100%]。因此，像素的alpha为0时，表示完全透明，像素的alpha为128时，表示50%透明，像素的alpha为255时，表示完全不透明。

为了让alpha部分描述像素的透明等级，我们必须设置源混合要素为D3DBLEND\_SRCALPHA以及目的混合要素为D3DBLEND\_INVSRCALPHA。这些值碰巧也是被默认设置的。

### 7.3.1Alpha通道

代替使用Alpha部分来计算遮影，我们能够从纹理的alpha通道中得到alpha信息。Alpha通道是额外的设置位，用它来保存每一个点的alpha值。当一个纹理被映射到一个图元上时，在alpha通道中的alpha信息也被映射，并且它们利用alpha信息为每个像素赋予纹理。图7.3显示了一个带8位alpha通道的图片。

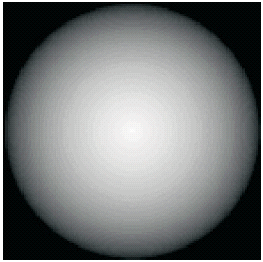


图7.3

图7.4显示的是一个利用alpha通道指定透明度来渲染的一个纹理方块。

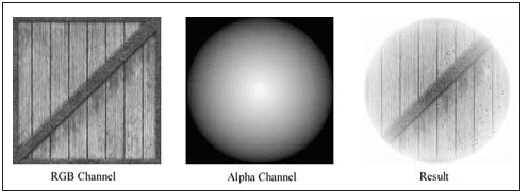


图7.4

### 7.3.2指定Alpha资源

默认情况下，假如设置一个有alpha通道的纹理，alpha值从在alpha通道中获得。假如没有alpha通道，那么alpha值是通过顶点颜色获得。然而，你能够通过下面的渲染状态来指定使用哪一个资源：

|  |
| --- |
| // compute alpha from diffuse colors during shading  Device->SetTextureStageState(0, D3DTSS\_ALPHAARG1, D3DTA\_DIFFUSE);  Device->SetTextureStageState(0, D3DTSS\_ALPHAOP, D3DTOP\_SELECTARG1);  // take alpha from alpha channel  Device->SetTextureStageState(0, D3DTSS\_ALPHAARG1, D3DTA\_TEXTURE);  Device->SetTextureStageState(0, D3DTSS\_ALPHAOP, D3DTOP\_SELECTARG1); |

## 7.4使用DirectX纹理工具创建Alpha通道

绝大多数普通图象文件格式没有存储alpha信息。在这一部分我们给你演示怎样使用DirectX纹理工具来创建一个带alpha通道的DDS文件。DDS文件是一个为DirectX应用程序和纹理设置的图象格式。DDS文件能够利用D3DXCreateTextureFromFile被读进纹理中，就象bmp和jpg文件一样。DirectX纹理工具被放在你的DXSDK目录下的\Bin\DXUtils文件夹下（我是放在C:\Program Files\Microsoft DirectX 9.0 SDK (February 2005)\Utilities\Bin\x86下的，文件名是DxTex.exe）。

打开DirectX纹理工具，并且把本章中示例文件夹下的crate.jpg文件用工具打开。木箱被自动的按照24位RGB纹理被读取。它包含8位红色，8位绿色，以及8位蓝色。我们需要将该纹理增加为32位ARGB纹理，增加的是额外的8位alpha通道。从菜单中选择Format，选择Change Surface Format。一个象图7.5的对话框将被弹出。选择A8R8G8B8格式点击OK。

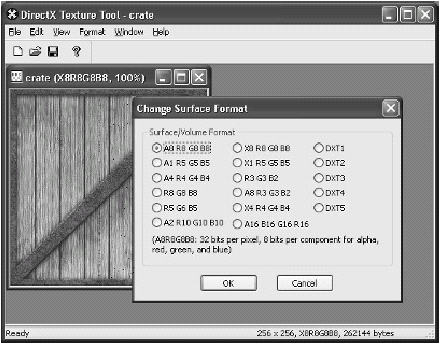


图7.5

它创建了一个32位颜色深度的图象，它的每个象素都有8位alpha通道，8位红色，8位绿色，8位蓝色。我们下一步是向alpha通道中写入数据。我们将图7.3中的8位灰色图片信息写进alpha通道中。选择菜单中的File，选择Open Onto Alpha Channel Of This Texture。一个对话框将弹出让你选择包含你想要写入alpha通道中数据信息的图片。选择alphachannel.bmp文件。图7.6显示的是程序已经插入了alpha通道数据。

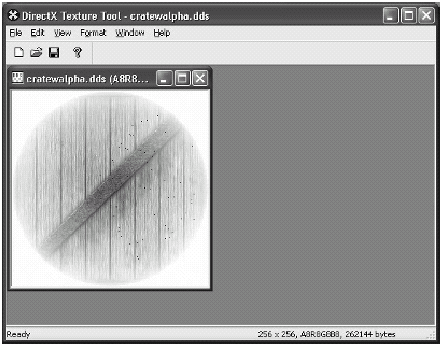


图7.6

现在用你选择的文件名存储纹理；我们使用cratewalpha.dds文件名。

## 7.5实例程序：透明度

这个实例程序是在一个木箱背景上绘制一个透明的茶壶，就象图7.2所显示的一样。在这个例子中alpha值是从材质中得到。应用程序允许我们通过按A或S键来增加/减少alpha的值。

使用混合的必要步骤是：

1. 设置混合要素D3DRS\_SRCBLEND 和 D3DRS\_DESTBLEND。
2. 假如你使用alpha部分，指定资源（材质或alpha通道）。
3. 允许alpha混合渲染状态。

对于这个例子，我们定义下面的全局变量：

|  |
| --- |
| ID3DXMesh\* Teapot = 0; // the teapot  D3DMATERIAL9 TeapotMtrl; // the teapot’s material  IDirect3DVertexBuffer9\* BkGndQuad = 0; // background quad - crate  IDirect3DTexture9\* BkGndTex = 0; // crate texture  D3DMATERIAL9 BkGndMtrl; // background material |

Setup方法设置很多东西；我们省略了很多与本章无关的代码。关心混合，Setup方法指定alpha值的获取资源。在这个例子中，我们通过材质指定alpha值。注意我们设置茶壶的材质alpha部分为0.5，也就是说茶壶将按照50%的透明度被渲染。我们在这里也要设置混合要素。要注意的是在这个方法中我们不能将alpha混合设置为启用。理由是alpha混合要进行额外的处理并且应该仅在需要用时才被使用。举例，在这个例子中只有茶壶需要用允许alpha混合来被渲染——而方块不需要。因此，我们在Display函数中启用alpha混合。

|  |
| --- |
| bool Setup()  {  TeapotMtrl = d3d::RED\_MTRL;  TeapotMtrl.Diffuse.a = 0.5f; // set alpha to 50% opacity  BkGndMtrl = d3d::WHITE\_MTRL;  D3DXCreateTeapot(Device, &Teapot, 0);    ...// Create background quad snipped  ...// Light and texture setup snipped    // use alpha in material's diffuse component for alpha  Device->SetTextureStageState(0, D3DTSS\_ALPHAARG1, D3DTA\_DIFFUSE);  Device->SetTextureStageState(0, D3DTSS\_ALPHAOP, D3DTOP\_SELECTARG1);  // set blending factors so that alpha  // component determines transparency  Device->SetRenderState(D3DRS\_SRCBLEND, D3DBLEND\_SRCALPHA);  Device->SetRenderState(D3DRS\_DESTBLEND, D3DBLEND\_INVSRCALPHA);    ...// view/projection matrix setup snipped  return true;  } |

在Display函数中，我们检测假如A或S键被按下那么就通过增加或减少材质的alpha值来反馈。注意这个方法要保证alpha值不会超出[0,1]的范围。我们然后渲染背景。最后，我们启用alpha混合，利用alpha混合来渲染茶壶，关闭alpha混合。

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  //  // Update  //  // increase/decrease alpha via keyboard input  if( ::GetAsyncKeyState('A') & 0x8000f )  TeapotMtrl.Diffuse.a += 0.01f;  if( ::GetAsyncKeyState('S') & 0x8000f )  TeapotMtrl.Diffuse.a -= 0.01f;  // force alpha to [0, 1] interval  if(TeapotMtrl.Diffuse.a > 1.0f)  TeapotMtrl.Diffuse.a = 1.0f;  if(TeapotMtrl.Diffuse.a < 0.0f)  TeapotMtrl.Diffuse.a = 0.0f;  //  // Render  //  Device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER,  0xffffffff, 1.0f, 0);  Device->BeginScene();  // Draw the background  D3DXMATRIX W;  D3DXMatrixIdentity(&W);  Device->SetTransform(D3DTS\_WORLD, &W);  Device->SetFVF(Vertex::FVF);  Device->SetStreamSource(0, BkGndQuad, 0, sizeof(Vertex));  Device->SetMaterial(&BkGndMtrl);  Device->SetTexture(0, BkGndTex);  Device->DrawPrimitive(D3DPT\_TRIANGLELIST, 0, 2);  // Draw the teapot  Device->SetRenderState(D3DRS\_ALPHABLENDENABLE, true);  D3DXMatrixScaling(&W, 1.5f, 1.5f, 1.5f);  Device->SetTransform(D3DTS\_WORLD, &W);  Device->SetMaterial(&TeapotMtrl);  Device->SetTexture(0, 0);  Teapot->DrawSubset(0);  Device->SetRenderState(D3DRS\_ALPHABLENDENABLE, false);  Device->EndScene();  Device->Present(0, 0, 0, 0);  }  return true;  } |

注意：在本章中有另一个使用纹理通道来演示alpha混合的例子texAlpha。与上边的代码不同之处仅仅在于得到alpha值是从纹理而不是从材质。

|  |
| --- |
| // use alpha channel in texture for alpha  Device->SetTextureStageState(0, D3DTSS\_ALPHAARG1, D3DTA\_TEXTURE);  Device->SetTextureStageState(0, D3DTSS\_ALPHAOP, D3DTOP\_SELECTARG1); |

这个应用程序读取的是一个在7.4节中用DX Tex Tool工具创建的带有alpha通道的DDS文件。

## 7.6摘要(略)

# 第八章 模版 (Stenciling)

这一章将带我们学习模版缓存，同时这也将结束第二部分的学习。模版缓存是一个远离屏幕的缓存，我们能够用它来完成一些特效。模版缓存与后缓存和深度缓存有相同的定义，因此在模版缓存中的*ijth*像素与后缓存和深度缓存中的*ijth*像素是相协调的。就象名字所说，模版缓存就象一个模版它允许我们印刷渲染后缓存的某个部分。

举例，当要实现一个镜子时，我们只需要简单地反射一个物体细节到镜子平面上；然而，我们仅仅想只绘制镜子里的反射结果。我们能用模版缓存来印制渲染它。图8.1清楚的显示了这一点。

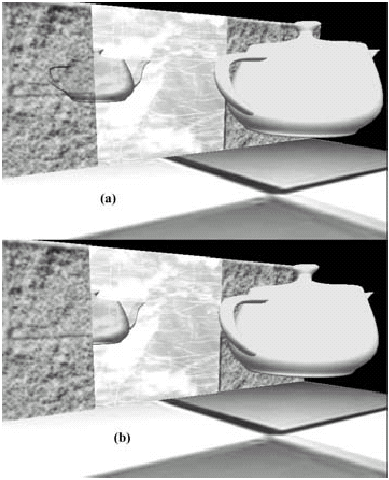


图8.1

模版缓存是Direct3D中的一小部分，它是通过一个简单的表面而被约束的。就象混合，这个简单的表面提供了可变的强大的设置能力。有效地学习使用模版缓存最好的方法是通过学习实际的应用程序。一旦你学懂了一点应用程序中的模版缓存，你将会得到一个更好的用于你自己需要特效的主意。

正因为这个原因，这一章我们特别安排学习两个使用模版缓存的应用程序。

## 目标

* 理解模版缓存是怎样工作的，怎样创建一个模版缓存以及怎样控制它。
* 学习怎样实现一个镜面效果，使用模版缓存来防止绘制反射到不在镜子表面上的物体。
* 利用模版缓存怎样渲染阴影和防止“双倍混合”。

## 8.1使用模版缓存

为了使用模版缓存，我们在初始化Direct3D时必须首先请求一个，然后必须启用它。我们在8.1.1中讲述怎样请求一个模版缓存。为了启用模版缓存，我们必须设置D3DRS\_STENCILENABLE渲染状态并且指定它为true（关闭它即可指定为false）。下面的代码是启用和关闭模版缓存的代码：

|  |
| --- |
| Device->SetRenderState(D3DRS\_STENCILENABLE, true);  ... // do stencil work  Device->SetRenderState(D3DRS\_STENCILENABLE, false); |

我们可以使用IDirect3DDevice9::Clear方法来清除模版缓存并让其拥有默认值。回忆一下，同样的方法被用在清除后缓存和深度缓存中。

|  |
| --- |
| Device->Clear(0, 0,  D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER | D3DCLEAR\_STENCIL,  0xff000000, 1.0f, 0 ); |

注意我们已经添加了D3DCLEAR\_STENCIL到第三个参数中，它表示我们想把模版缓存和目标（后缓存）以及深度缓存一起清除。有6种值可以用来指定清除后的模版缓存；在这个例子中我们将它清除为0。

### 8.1.1请求一个模版缓存

在我们创建深度缓存的同时一个模版缓存能够被创建。当指定深度缓存格式的时候，我们同时指定模版缓存的格式。这样，模版缓存和深度缓存分享同一个离屏表面缓存，但是每个像素被指定到各自缓存内存片段中。下面列出了3种深度/模版缓存的格式：

* D3DFMT\_D24S8—这种格式是说创建一个32位深度/模版缓存，其中24位为深度缓存，8位为模版缓存。
* D3DFMT\_D24X4S4—这种格式是说创建一个32位深度/模版缓存，其中24位为深度缓存，4位为模版缓存，还有4位留着不用。
* D3DFMT\_D15S1—这种格式是说创建一个16位深度/模版缓存，其中15位为深度缓存，1位为模版缓存。

注意，还有一些格式没有分配任何位给模版缓存。例如，D3DFMT\_D32格式是说只创建一个32位深度缓存。

同样，不同硬件对模版缓存的支持也是不同的。例如有些显卡就不支持8位模版缓存。

### 8.1.2模版测试

如前所述，我们能够使用模版缓存来阻止渲染后缓存中的某些部分。阻止特殊像素被写是通过模版测试（*stencil test*）来决定的，这是通过下面的表达式来完成的：

|  |
| --- |
| (ref & mask) ComparisonOperation (value & mask) |

模版测试是对每个像素进行的，假设模版是被允许。将有两个操作：

* 左手边操作数（*LHS*=*ref&mask*）
* 右手边操作数（*RHS*=*value&mask*）

模版测试比较LHS和RHS，通过比较运算来指定。全部的运算都得到一个布尔值（true/false）。假如测试的结果是true，那么我们把像素写入后缓存。假如测试的结果是false,我们就阻止像素被写入后缓存。当然，如果像素不能被写入后缓存，那么它也不能被写入深度缓存。

### 8.1.3控制模版测试

Direct3D允许我们控制变量用于模版测试。换句话说，我们可以指定参考值（stencil reference）和掩码(mask value)，以便进行比较运算。虽然我们不能明确地设定模版值（stencil value）,但是我们能够控制写入模版缓存的值。

### 8.1.3.1模版参考值（Reference Value）

模版参考值ref的默认值为0，但是我们能够通过设置D3DRS\_STENCILREF渲染状态来改变它。例如，下面的代码就是设置模版参考值为1：

|  |
| --- |
| Device->SetRenderState(D3DRS\_STENCILREF, 0x1); |

注意我们往往使用16进制，因为这让它看起来比整数更容易象一个位队列，并且当我们做位操作时这样看起来更有用，比如相加。

### 8.1.3.2模版掩码

模版掩码值mask是被用来掩饰（隐藏）在ref和value变量中的位。它的默认值是0xffffffff，也就是没有掩饰任何位。我们能够通过设置D3DRS\_STENCILMASK渲染状态来改变它。下面的例子就是掩饰高16位：

|  |
| --- |
| Device->SetRenderState(D3DRS\_STENCILMASK, 0x0000ffff); |

### 8.1.3.3模版值（Stencil Value）

作为以前的规定，在模版缓存中我们进行模版测试的当前像素。例如，假如我们对*ijth*像素进行模版测试，那么该值将被写入*ijth*模版缓存。我们不能明确地设置个别模版值，但是可以清除模版缓存。我们能够使用模版渲染状态来控制将什么写入模版缓存。

### 8.1.3.4比较运算

我们能够通过设置D3DRS\_STENCILFUNC渲染状态来设置比较运算。这个比较运算能够被D3DCMPFUNC的任何成员类型列举：

|  |
| --- |
| typedef enum \_D3DCMPFUNC {  D3DCMP\_NEVER = 1,  D3DCMP\_LESS = 2,  D3DCMP\_EQUAL = 3,  D3DCMP\_LESSEQUAL = 4,  D3DCMP\_GREATER = 5,  D3DCMP\_NOTEQUAL = 6,  D3DCMP\_GREATEREQUAL = 7,  D3DCMP\_ALWAYS = 8,  D3DCMP\_FORCE\_DWORD = 0x7fffffff  } D3DCMPFUNC; |

* D3DCMP\_NEVER——模版测试永不成功。
* D3DCMP\_LESS——假如*LHS < RHS*，那么模版测试成功。
* D3DCMP\_EQUAL——假如*LHS = RHS*，那么模版测试成功。
* D3DCMP\_LESSEQUAL——假如*LHS <= RHS*，那么模版测试成功。
* D3DCMP\_GREATER——假如*LHS > RHS*，那么模版测试成功。
* D3DCMP\_NOTEQUAL——假如*LHS <> RHS*，那么模版测试成功。
* D3DCMP\_GREATEREQUAL——假如*LHS >= RHS*，那么模版测试成功。
* D3DCMP\_ALWAYS——模版测试总是成功。

### 8.1.3更新模版缓存

除了决定是否写或阻止一个特殊像素被写入后缓存以外，我们能够定义模版缓存基于三种可能的案例怎样被更新：

* 对于*ijth*像素模版测试失败。我们能够定义怎样更新在模版缓存中的*ijth，*通过设置D3DRS\_STENCILFAIL渲染状态来适应这种情形：

|  |
| --- |
| Device->SetRenderState(D3DRS\_STENCILFAIL, StencilOperation); |

* 对于*ijth*像素深度测试失败。我们能够定义怎样更新在模版缓存中的*ijth，*通过设置D3DRS\_STENCILZFAIL渲染状态来适应这种情形：

|  |
| --- |
| Device->SetRenderState(D3DRS\_STENCILZFAIL, StencilOperation); |

* 对于*ijth*像素模版测试和深度测试都成功。我们能够定义怎样更新在模版缓存中的*ijth，*通过设置D3DRS\_STENCILPASS渲染状态来适应这种情形：

|  |
| --- |
| Device->SetRenderState(D3DRS\_STENCILPASS, StencilOperation); |

其中StencilOperation能够是下面预先定义的常数：

* D3DSTENCILOP\_KEEP——指定不改变模版缓存。
* D3DSTENCILOP\_ZERO——指定设置模版缓存入口为0。
* D3DSTENCILOP\_REPLACE——指定用模版参考值（reference value）来替换模版缓存入口。
* D3DSTENCILOP\_INCRSAT——指定增加模版缓存入口。假如增加的值超过了允许的最大值，我们就设置它为最大值。
* D3DSTENCILOP\_DECRSAT——指定减少模版缓存入口。假如减少后的值小于了0，我们就设置它0。
* D3DSTENCILOP\_INVERT——指定按位取反模版缓存入口。
* D3DSTENCILOP\_INCR——指定增加模版缓存入口。假如增加的值超过了允许的最大值，我们就设置它为0。
* D3DSTENCILOP\_DECR——指定减少模版缓存入口。假如减少后的值小于了0，我们就设置它为允许的最大值。

### 8.1.4模版写掩码

除了已经提及的模版渲染状态之外，我们能够设置一个写掩码（write mask）它将掩饰我们写进模版缓存的任何值的位。我们能够通过D3DRS\_STENCILWRITEMASK渲染状态来设置写掩码。它的默认值是0xffffffff。下面的例子是掩饰高16位：

|  |
| --- |
| Device->SetRenderState(D3DRS\_STENCILWRITEMASK, 0x0000ffff); |

## 8.2实例程序：镜子

在自然界中的很多表面象镜子一样允许我们通过它的反射来看物体。这一部分讲了我们怎样用3D应用程序来模拟镜子。注意为了简单我们只模拟平面镜。举点例子，一辆擦亮的小汽车能够反射；然而，小车的车身是光滑的，圆的，不是一个平面。我们渲染反射是这些，象光滑的大理石地板、挂在墙上的镜子。换句话说就是在一个平面的镜子。

实现镜子的程序需要我们解决两个问题。第一，我们必须学习沿着一个面怎样反射一个物体以便能够正确地绘制反射结果。第二，我们必须只能在一个镜子范围内显示反射结果。即，我们必须掩饰一个表面作为一个镜子，且只渲染那些在镜子里物体。图8.1就是说的这个内容。

第一个问题只需要用一些几何向量就可以简单解决。我们能够利用模版缓存解决第二个问题。下两小节分别介绍怎样解决这两个问题。第三小节把它们柔和在一起并且介绍一下本章的第一个应用程序实例代码——镜子。

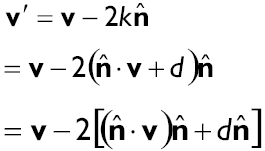
### 8.2.1反射数学

我们现在演示怎样计算点V=（Vx, Vy, Vz）被平面n\*p+d=0反射的点V’=（V’x, V’y, V’z）。图8.2贯穿整个讨论。

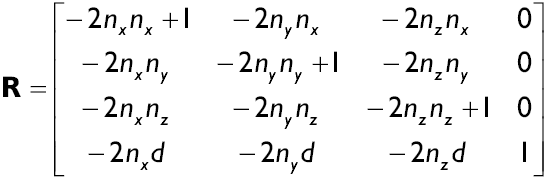


图8.2

根据**Part I**中的“平面”部分，我们能够知道q=v-kn,这里k是有符号的从v到平面的距离。下面是v相对与平面（n，d）的反射推导：



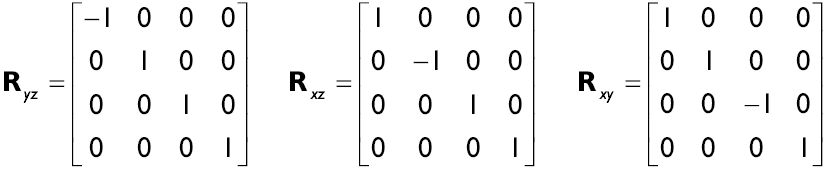
我们用下面的矩阵来实现从v到v’的转换：



在D3DX库中用下面的函数来创建反射矩阵R。

|  |
| --- |
| D3DXMATRIX \*D3DXMatrixReflect(  D3DXMATRIX \*pOut, // The resulting reflection matrix.  CONST D3DXPLANE \*pPlane // The plane to reflect about.  ); |

一旦我们说到反射变换的话题，就让我们看看其他3种特殊的反射变换。它们是关于三个坐标平面的反射—yz平面，xz平面，和xy平面—它们分别通过下面三个矩阵来表现：



通过yz平面反射一个点，我们只需要简单的将x分量取反就可以了。同样的，通过xz平面反射一个点，我们只需要简单的将y分量取反。通过xy平面反射一个点，我们只需要简单的将z分量取反。这种反射是非常容易理解的。

### 8.2.2镜面实现流程

当实现一个镜面，一个物体假如在一面镜子前那么它就会被反射。然而，我们不想测试空间假如一个物体在一面镜子前，要做它是非常复杂的。因此，为了简化事情，我们总是反射物体并且无限制地渲染它。但是这样就有一个象本章开头的图8.1一样的问题。即，物体反射被渲染到了没有镜子的表面。我们能够用模版缓存来解决这个问题，因为模版缓存允许我们阻止渲染在后缓存中的特定区域。因此，我们使用模版缓存来阻止渲染被反射的不在镜子里的茶壶。下面的步骤简要的说明了怎样实现：

1. 正常渲染所有的场景——地板，墙，镜子和茶壶——不包含反射的茶壶。注意这一步没有修改模版缓存。
2. 清除模版缓存为0。图8.3显示了后缓存和模版缓存。

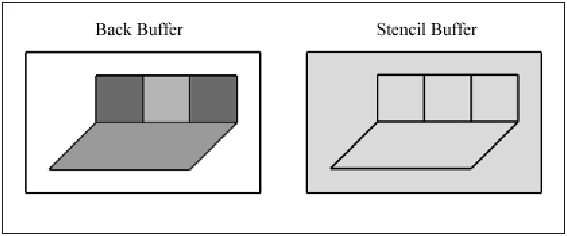


图8.3

1. 渲染只有镜子部分的图元到模版缓存中。设置模版测试总是成功，并且假如测试成功就指定模版缓存入口为1。我们仅仅渲染镜子，在模版缓存中的所有像素都将为0，除了镜子部分为1以外。图8.4显示了更新以后的模版缓存。也就是说，我们在模版缓存中对镜子像素做了标记。

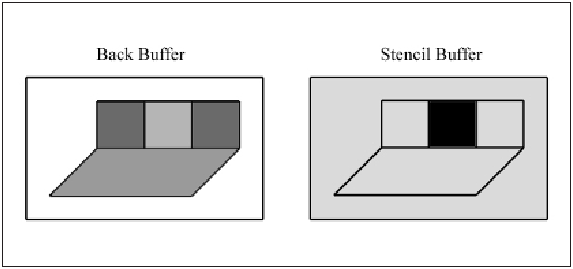


图8.4

1. 现在我们渲染被反射的茶壶到后缓存和模版缓存中。但是假如模版测试通过，我们就只渲染后缓存。假如在模版缓存中的值为1，那么我们设置模版测试通过。这样，茶壶就仅仅被渲染到模版缓存为1的地方了。因为只有镜子对应的模版缓存值为1，所以反射的茶壶就只能被渲染到镜子里。

### 8.2.3代码和解释

这个例子的相关代码在RenderMirror函数中，它首先渲染镜子图元到模版缓存，然后渲染那些能被渲染到镜子里的反射茶壶。我们现在一行一行的分析RenderMirror函数的代码，并解释为什么要这么做。

假如你想使用8.2.2部分的步骤实现代码，注意我们从第3步开始，因为对模版缓存来说1和2步已经没有什么事做了。同样我们通过这个解释来讨论通过镜子渲染的信息。

注意我们将分成几个部分来讨论它。

### 8.2.3.1第一部分

我们通过允许模版缓存和设置渲染状态来开始：

|  |
| --- |
| void RenderMirror()  {  Device->SetRenderState(D3DRS\_STENCILENABLE, true);  Device->SetRenderState(D3DRS\_STENCILFUNC, D3DCMP\_ALWAYS);  Device->SetRenderState(D3DRS\_STENCILREF, 0x1);  Device->SetRenderState(D3DRS\_STENCILMASK, 0xffffffff);  Device->SetRenderState(D3DRS\_STENCILWRITEMASK,0xffffffff);  Device->SetRenderState(D3DRS\_STENCILZFAIL, D3DSTENCILOP\_KEEP);  Device->SetRenderState(D3DRS\_STENCILFAIL, D3DSTENCILOP\_KEEP);  Device->SetRenderState(D3DRS\_STENCILPASS, D3DSTENCILOP\_REPLACE); |

这是非常容易理解的。我们设置模版比较运算为D3DCMP\_ALWAYS,这就是说让所有模版测试都通过。

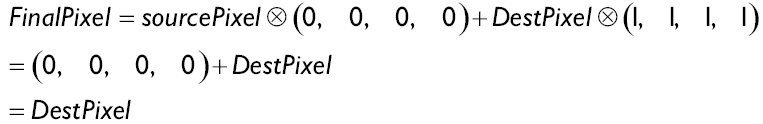
假如深度测试失败了，我们指定D3DSTENCILOP\_KEEP，它表明不更新模版缓存入口。即，我们保存当前值。这样做的原因是假如深度测试失败了，那么就意味着像素被“模糊”了。我们不想渲染被“模糊”的反射像素。

同样假如模版测试失败了，我们也指定D3DSTENCILOP\_KEEP。但是在这里这样做不是必须的，因为我们指定的是D3DCMP\_ALWAYS，当然这样的测试也就永远不会失败。然而，我们只改变比较运算的一位，那么设置模版失败渲染状态是必须的。我们现在就这样做。

假如深度测试和模版测试都通过了，我们就指定D3DSTENCILOP\_REPLACE，更新模版缓存入口，设置模版参考值为0x1。

### 8.2.3.2第二部分

这下一步阻止渲染镜子代码，除了模版缓存。我们通过设置D3DRS\_ZWRITEENABLE并指定为false来阻止写深度缓存。我们能够防止更新后缓存，混合和设置源混合要素为D3DBLEND\_ZERO目的混合要素为D3DBLEND\_ONE。将这些混合要素代入混合等式，我们得到后缓存是不会改变的：



|  |
| --- |
| // disable writes to the depth and back buffers  Device->SetRenderState(D3DRS\_ZWRITEENABLE, false);  Device->SetRenderState(D3DRS\_ALPHABLENDENABLE, true);  Device->SetRenderState(D3DRS\_SRCBLEND, D3DBLEND\_ZERO);  Device->SetRenderState(D3DRS\_DESTBLEND, D3DBLEND\_ONE);  // draw the mirror to the stencil buffer  Device->SetStreamSource(0, VB, 0, sizeof(Vertex));  Device->SetFVF(Vertex::FVF);  Device->SetMaterial(&MirrorMtrl);  Device->SetTexture(0, MirrorTex);  D3DXMATRIX I;  D3DXMatrixIdentity(&I);  Device->SetTransform(D3DTS\_WORLD, &I);  Device->DrawPrimitive(D3DPT\_TRIANGLELIST, 18, 2);  // re-enable depth writes  Device->SetRenderState(D3DRS\_ZWRITEENABLE, true); |

### 8.2.3.3第三部分

在模版缓存中，符合镜子可视像素的为0x1，因此对已经渲染的镜子区域做记号。我们现在准备渲染被反射的茶壶。回忆一下，我们仅仅想渲染镜子范围内的反射像素。我们现在可以很容易的做到了，因为在模版缓存中这些像素已经被做了记号。

我们设置下面的渲染状态：

|  |
| --- |
| Device->SetRenderState(D3DRS\_STENCILFUNC, D3DCMP\_EQUAL);  Device->SetRenderState(D3DRS\_STENCILPASS, D3DSTENCILOP\_KEEP); |

用一个新的比较运算设置，我们进行下面的模版测试：

|  |
| --- |
| (ref & mask == (value & mask)  (0x1 & 0xffffffff) == (value & 0xffffffff)  (0x1)== (value & 0xffffffff) |

这说明了只有当value=0x1时模版测试才成功。因为在模版缓存中只有镜子相应位置的值才是0x1，若我们渲染这些地方那么测试将会成功。因此，被反射的茶壶只会在镜子里绘制而不会在镜子以外的表面上绘制。

注意我们已经将渲染状态由D3DRS\_STENCILPASS变为了D3DSTENCILOP\_KEEP，简单的说就是假如测试通过那么就保存模版缓存的值。因此，在下一步的渲染中，我们不改变模版缓存的值。我们仅仅使用模版缓存来对镜子相应位置的像素做标记。

### 8.2.3.4第四部分

RenderMirror函数的下一部分就是计算在场景中反射位置的矩阵：

|  |
| --- |
| // position reflection  D3DXMATRIX W, T, R;  D3DXPLANE plane(0.0f, 0.0f, 1.0f, 0.0f); // xy plane  D3DXMatrixReflect(&R, &plane);  D3DXMatrixTranslation(&T,  TeapotPosition.x,  TeapotPosition.y,  TeapotPosition.z);  W = T \* R; |

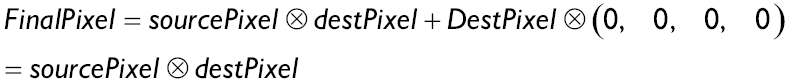
注意我们首先确定没有反射的茶壶位置，然后就通过xy平面来反射。这种变换规则是通过矩阵相乘来指定的。

### 8.2.3.5第五部分

我们已经为渲染反射茶壶做好了准备。然而，假如我们现在就渲染它，它是不会被显示的。为什么呢？因为被反射的茶壶的深度比镜子的深度大，因此镜子的图元将把被反射茶壶的图元弄模糊。为了避免这种情况，我们清除深度缓存：

|  |
| --- |
| Device->Clear(0, 0, D3DCLEAR\_ZBUFFER, 0, 1.0f, 0); |

并不是所有问题都解决了。假如我们简单的清除深度缓存，被反射的茶壶会被绘制到镜子的前面，物体看起来就不对了。我们想做的是清除深度缓存并且要混合被反射的茶壶和镜子。这样，被反射的茶壶看起来就象在镜子里了。我们能够通过下面的混合等式来混合被反射的茶壶和镜子：



因为原像素（sourcePixel）来自被反射的茶壶，目的像素（DestPixel）来自镜子，我们能够通过这个等式明白它们是怎么被混合到一起的。我们有如下的代码：

|  |
| --- |
| Device->SetRenderState(D3DRS\_SRCBLEND, D3DBLEND\_DESTCOLOR);  Device->SetRenderState(D3DRS\_DESTBLEND, D3DBLEND\_ZERO); |

最后，我们准备绘制被反射的茶壶：

|  |
| --- |
| Device->SetTransform(D3DTS\_WORLD, &W);  Device->SetMaterial(&TeapotMtrl);  Device->SetTexture(0, 0);  Device->SetRenderState(D3DRS\_CULLMODE, D3DCULL\_CW);  Teapot->DrawSubset(0); |

回顾一下8.2.3.4部分的W，它能够正确的将被反射的茶壶变换到场景中恰当的位置。同样，我们也要改变背面拣选模式。必须这样做的原因是当一个物体被反射以后，它的正面和背面将会被交换。因此为了改变这种情况，我们必须改变背面拣选模式。

|  |
| --- |
| Device->SetRenderState(D3DRS\_ALPHABLENDENABLE, false);  Device->SetRenderState( D3DRS\_STENCILENABLE, false);  Device->SetRenderState(D3DRS\_CULLMODE, D3DCULL\_CCW);  } // end RenderMirror() |

## 8.3实例程序：平面阴影

在场景中被灯光照射的地方会产生阴影，这将使场景变的更真实。在这一部分我们将演示怎样实现平面阴影，即在平面上的阴影（如图8.5）。



图8.5

注意这种阴影是“快砍”，虽然它们增强了场景效果，但是这并不是现实中的阴影。阴影值是一个高级的概念，要深入研究它已经超出了本书的范围。然而，特别值得提及的是在DirectX SDK中有一个示例程序演示了阴影值。

为了实现平面阴影，我们首先必须找到物体投射到平面上的阴影并进行几何建模以便我们能够渲染它。用一些3D数学就能很容易的实现它。我们然后用50%透明度的黑色材质来渲染描述阴影的多边形。渲染阴影时可能出现“双倍混合”，我们将用一小部分进行解释。我们使用模版缓存来防止双倍混合发生。

### 8.3.1平行光阴影

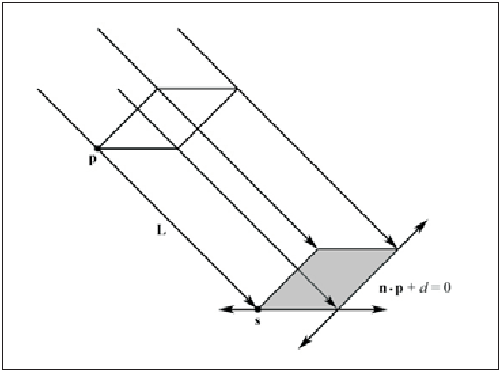
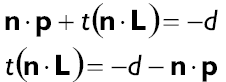
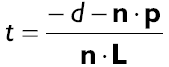


图8.6

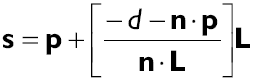
图8.6显示了物体在平行光照射下得到的阴影。光线是从平行光源放射出的，它的方向是L,通过顶点p得到r（t） = p + tL。光线r（t）和平面n \* p + d = 0 相交得到 s 。交点s 通过射线和平面相交测试是非常容易得到的：

把r(t)带进平面等式

求解t



那么：



### 8.3.2点光源阴影

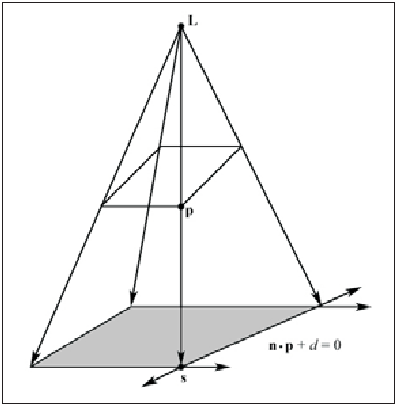


图8.7

图8.7显示了物体在点光源照射下得到的阴影。点光源的位置是L。光线通过顶点p，则得到 r(t) = p + t ( p – L )。光线r（t）和平面n \* p + d = 0 相交得到 s 。用8.3.1同样的方法我们可以得到s。

**注意：**在点光源和平行光中的L是不同的。对于点光源，我们用L来表示点光源的位置。而对于平行光，我们则是用L来表示平行光的照射方向。

### 8.3.3阴影矩阵

注意图8.6中所示的平行光，影子本质上是把物体按照灯光照射方向平行地投射到平面n\*p+d=0之上。同样的，图8.7中所示的点光源，影子本质上是把物体按照透视画法从光源投射到平面n\*p+d=0之上。

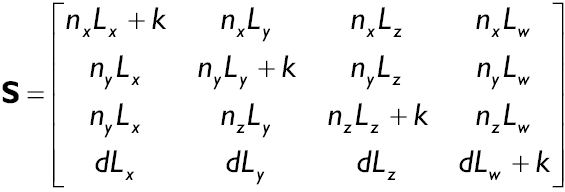
我们能够使用一个矩阵来表示从一个顶点p变换到平面n\*p=d=0上的s的变化。而且，我们能够用同一个矩阵来表现正交投影和透视投影。

我们用一个4D向量（nx, ny, nz, d）来表示将要用于投射阴影平面的平面等式中的各个系数。让4D向量L=（Lx, Ly, Lz, Lw）来表示平行光的照射方向或点光源的位置。我们用w来区别：

1. 假如w＝０，那么L表示平行光的照射方向。
2. 假如w＝1 ，那么L表示点光源的位置。

规格化的平面是非常不逊的，我们让k＝（nx, ny, nz, d）\*（Lx, Ly, Lz, Lw）= nxLx+nyLy+nzLz+dLw

那么我们就可得到表示点p到点s的变换矩阵，即阴影矩阵：



因为在其他地方已经被推导出来了，对于我们来说推导它并没有重大的意义，在这里我们就不再演示推导怎样得到这个矩阵的过程了。但是对与感兴趣的读者可以自己到网上查找相应的信息。

　　在D3DX库中已经给我们提供了一个建立阴影矩阵的函数。其中当w＝０时表示平行光，当w＝１时表示点光源：

|  |
| --- |
| D3DXMATRIX \*D3DXMatrixShadow(  D3DXMATRIX \*pOut,  CONST D3DXVECTOR4 \*pLight, // L  CONST D3DXPLANE \*pPlane // plane to cast shadow onto  ); |

### 8.3.4用模版缓存防止双倍混合

几何学上，当我们将一个物体投影到一个平面上时，很可能会有两个或者更多的投影三角形被重叠到一起。若我们就这样渲染，那么有重叠三角形的地方就会被多次混合以至这些地方将会变得更黑。图8.8就是这种情况。

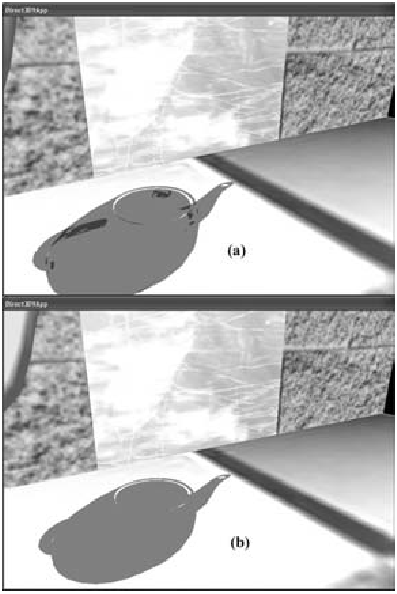


图8.8

我们能够使用模版缓存来解决这个问题。我们设置模版测试为允许像素第一次被渲染。即，当把影子像素渲染到后缓存时，我们同时在模版缓存中做好标记。然后，如果试图把像素向一个已经渲染过的地方写，那么模版测试将会失败。这样，我们就防止了重复写像素也就是防止了双倍混合的发生。

### 8.3.5代码和解释

下面的代码就是讲解影子例子。本例的相关代码都在RenderShadow函数中。注意我们假设模版缓存都已经被清除为０了。

首先设置模版渲染状态。将模版比较运算设为D3DCMP\_EQUAL且将D3DRS\_STENCILREF渲染状态设置为0x0，因此假如在模版缓存中相应的值为0x0，那么就指定渲染阴影到后缓存中。

因为模版缓存是被清除为0x0的，所以我们第一次将影子像素写入的时候总是正确的；不过因为我们设置D3DRS\_STENCILPASS为D3DSTENCILOP\_INCR,假如你试图将已经写过的像素写入的话，这个测试将会失败。在第一次写入的时候模版像素已经被写成了0x1，因此假如你再一次写入，模版测试将会失败。因此，我们避免了重复写像素，也避免了双倍混合。

|  |
| --- |
| void RenderShadow()  {  Device->SetRenderState(D3DRS\_STENCILENABLE, true);  Device->SetRenderState(D3DRS\_STENCILFUNC, D3DCMP\_EQUAL);  Device->SetRenderState(D3DRS\_STENCILREF, 0x0);  Device->SetRenderState(D3DRS\_STENCILMASK, 0xffffffff);  Device->SetRenderState(D3DRS\_STENCILWRITEMASK, 0xffffffff);  Device->SetRenderState(D3DRS\_STENCILZFAIL, D3DSTENCILOP\_KEEP);  Device->SetRenderState(D3DRS\_STENCILFAIL, D3DSTENCILOP\_KEEP);  Device->SetRenderState(D3DRS\_STENCILPASS, D3DSTENCILOP\_INCR); |

下一步，我们计算阴影变换并将它放置到场景中适当的位置。

|  |
| --- |
| // compute the transformation to flatten the teapot into a shadow.  D3DXVECTOR4 lightDirection(0.707f, -0.707f, 0.707f, 0.0f);  D3DXPLANE groundPlane(0.0f, -1.0f, 0.0f, 0.0f);  D3DXMATRIX S;  D3DXMatrixShadow(&S, &lightDirection, &groundPlane);  D3DXMATRIX T;  D3DXMatrixTranslation(&T, TeapotPosition.x, TeapotPosition.y,  TeapotPosition.z);  D3DXMATRIX W = T \* S;  Device->SetTransform(D3DTS\_WORLD, &W); |

最后，我们设置一个50%透明度的黑色材质，关闭深度测试，渲染阴影，然后开启深度缓存同时关闭alpha混合和模版测试。我们关闭深度缓存来防止z-fighting，它是当两个不同的表面在深度缓存中有同样的深度值时出现的虚拟物体；深度缓存不知道那一个是在前面，此时就会产生讨厌的闪动。因为阴影和地板是在同一个平面上，z-fighting很可能就会出现。通过先渲染地板然后用深度测试屏蔽阴影，这样我们就能够保证阴影将绘制在地面只之上。

|  |
| --- |
| Device->SetRenderState(D3DRS\_ALPHABLENDENABLE, true);  Device->SetRenderState(D3DRS\_SRCBLEND, D3DBLEND\_SRCALPHA);  Device->SetRenderState(D3DRS\_DESTBLEND, D3DBLEND\_INVSRCALPHA);  D3DMATERIAL9 mtrl = d3d::InitMtrl(d3d::BLACK, d3d::BLACK,  d3d::BLACK, d3d::BLACK, 0.0f);  mtrl.Diffuse.a = 0.5f; // 50% transparency.  // Disable depth buffer so that z-fighting doesn't occur when we  // render the shadow on top of the floor.  Device->SetRenderState(D3DRS\_ZENABLE, false);  Device->SetMaterial(&mtrl);  Device->SetTexture(0, 0);  Teapot->DrawSubset(0);  Device->SetRenderState(D3DRS\_ZENABLE, true);  Device->SetRenderState(D3DRS\_ALPHABLENDENABLE, false);  Device->SetRenderState(D3DRS\_STENCILENABLE, false);  }//end RenderShadow() |

## 8.4摘要(略)

# 第三部分 实用的Direct3D (Applied Direct3D)

# 第九章 字体 (Fonts)

在游戏中，我们常常需要显示一写文本信息给用户。这一章讨论三种在Direct3D中能够产生和输出文字的方法。每一种方法都有一个对应的实例程序。

## 目标

* 学习怎样使用ID3DXFont接口来渲染文字。
* 学习怎样使用CD3DFont类来渲染文字。
* 学习怎样计算渲染帧率。
* 学习怎样使用D3DXCreateText函数来创建和渲染3D文字。

## 9.1 ID3DXFont

在D3DX库中提供了一个ID3DXFont接口，它能被用于在Direct3D应用程序中绘制文字。这个接口是使用GDI来绘制文字的，因此我们能够使用这个接口来执行这个操作。无论如何，因为ID3DXFont使用的是GUI，所以它能够联合字体句柄和格式化字体。

### 9.1.1创建一个ID3DXFont

我们能够使用D3DXCreateFontIndirect函数来创建一个ID3DXFont接口。

|  |
| --- |
| HRESULT D3DXCreateFontIndirect(  LPDIRECT3DDEVICE9 pDevice, // device to be associated with the font  CONST LOGFONT\* pLogFont, // LOGFONT structure describing the font  LPD3DXFONT\* ppFont // return the created font  ); |

下面的代码片段显示了怎样使用这个函数：

|  |
| --- |
| LOGFONT lf;  ZeroMemory(&lf, sizeof(LOGFONT));  lf.lfHeight = 25; // in logical units  lf.lfWidth = 12; // in logical units  lf.lfWeight = 500; // boldness, range 0(light) - 1000(bold)  lf.lfItalic = false;  lf.lfUnderline = false;  lf.lfStrikeOut = false;  lf.lfCharSet = DEFAULT\_CHARSET;  strcpy(lf.lfFaceName, "Times New Roman"); // font style  ID3DXFont\* font = 0;  D3DXCreateFontIndirect(Device, &lf, &font); |

我们必须填充一个LOGFONT结构来描述想创建的字体类型。

注意：你也能够使用D3DXCreateFont函数来获得一个ID3DXFont接口指针。

### 9.1.2绘制文本

一旦我们获得了ID3DXFont接口指针，绘制文本就是很简单的事情了，我们只要调用ID3DXFont::DrawText方法就可以实现了。

|  |
| --- |
| INT ID3DXFont::DrawText(  LPCSTR pString,  INT Count,  LPRECT pRect,  DWORD Format,  D3DCOLOR Color  ); |

* pString — 指向要绘制的文字。
* Count — 字符串中特征字符的数量。假如字符是以null结束的字符串则可将其指定为-1。
* pRect — 指向一个RECT结构，它定义一个文字被绘制在屏幕上的范围。
* Format — 可选参数，指定文字怎样被格式化；要获得更详细的信息请查看SDK文档。
* Color — 文字的颜色。

例子：

|  |
| --- |
| Font->DrawText(  "Hello World", // String to draw.  -1, // Null terminating string.  &rect, // Rectangle to draw the string in.  DT\_TOP | DT\_LEFT, // Draw in top-left corner of rect.  0xff000000  ); // Black. |

### 9.1.3计算每秒的渲染帧数

这一章的ID3DXFont和Cfont例子是计算和显示每秒渲染的帧数（FPS）。这一部分说明怎样计算FPS。

首先，我们定义如下三个全局变量：

|  |
| --- |
| DWORD FrameCnt; // The number of frames that have occurred.  float TimeElapsed; // The time that has elapsed so far.  float FPS; // The frames rendered per second. |

我们计算每一秒的FPS；它给我们一个很好的平均。另外，在同一秒中内只保存一个FPS，这给了我们足够时间来读取它，在它再一次改变之前。

因此每一帧我们增加FrameCnt并且把从上一帧到现在流逝的时间写进TimeElapsed：

|  |
| --- |
| FrameCnt++;  TimeElapsed += timeDelta; |

这里timeDelta是两帧之间的时间。

在一秒种结束以后，我们能够用下面的公式来计算FPS：

|  |
| --- |
| FPS = (float)FrameCnt / TimeElapsed; |

我们从新设置FrameCnt和TimeElapsed为计算下一秒的FPS做准备。下面就是合在一起的代码：

|  |
| --- |
| void CalcFPS(float timeDelta)  {  FrameCnt++;  TimeElapsed += timeDelta;  if(TimeElapsed >= 1.0f)  {  FPS = (float)FrameCnt / TimeElapsed;  TimeElapsed = 0.0f;  FrameCnt = 0;  }  } |

## 9.2 CD3DFont

DirectX SDK给我们提供了一些很有用的代码，它们在你的DXSDK目录下的\Samples\C++\Commond下。CD3DFont类代码是使用纹理三角形和Direct3D。因为CD3DFont使用Direct3D代替GDI来渲染， 这比ID3DXFont快的多。然而，CD3DFont不能够联合字体句柄和格式化ID3DXFont。假如你追求速度和只需要一些简单的字体，CD3DFont类就能满足你的要求了。

使用CD3DFont类，你需要添加下列文件到你的程序中：d3dfont.h, d3dfont.cpp, d3dutil.h, d3dutil.cpp, dxutil.h和dxutil.cpp。这些文件可以在刚才所说目录下的Include和Src目录下。

### 9.2.1创建一个CD3DFont

为了创建一个CD3DFont实例，我们只需要简单地象一般的C++对象那样实例化就可以了；下面是它的构造原型：

|  |
| --- |
| CD3DFont(const TCHAR\* strFontName, DWORD dwHeight, DWORD dwFlags=0L); |

* strFontName — 以null结束的字符串，它指定字体类型。
* dwHeight — 字体的高度。
* dwFlags — 可选参数；你能设置该参数为0或者用下面参数；D3DFONT\_BOLD, D3DFONT\_ITALIC, D3DFONT\_ZENABLE。

实例化一个CD3DFont对象以后，我们必须调用下面的方法来初始化字体：

|  |
| --- |
| Font = new CD3DFont("Times New Roman", 16, 0); // instantiate  Font->InitDeviceObjects( Device );  Font->RestoreDeviceObjects(); |

### 9.2.2绘制文本

现在我们已经创建和初始化了一个CD3DFont对象，这已经为绘制文字做好了准备。绘制文字是使用下面的方法：

|  |
| --- |
| HRESULT CD3DFont::DrawText(FLOAT x, FLOAT y, DWORD dwColor,  const TCHAR\* strText, DWORD dwFlags=0L); |

* x — 文字在屏幕上开始绘制的x坐标。
* y —文字在屏幕上开始绘制的y坐标。
* dwColor — 文字的颜色。
* strText — 要绘制的文字。
* dwFlags — 可选参数；你能设置该参数为0或者用下面参数；D3DFONT\_CENTERED, D3DFONT\_TWOSIDED, D3DFONT\_FILTERED。

例子：

|  |
| --- |
| Font->DrawText(20, 20, 0xff000000, “Hello, World”); |

### 9.2.3 清除

在删除一个CD3DFont对象之前，我们必须首先调用一些清除程序，就象下面列举的代码片段：

|  |
| --- |
| Font->InvalidateDeviceObjects();  Font->DeleteDeviceObjects();  delete Font; |

## 9.3 D3DXCreateText

最后的函数是被用来创建一个3D 文字网格。图9.1显示了本章FontMes3D实例渲染的3D文字网格。



图9.1

该函数的原型是：

|  |
| --- |
| HRESULT D3DXCreateText(  LPDIRECT3DDEVICE9 pDevice,  HDC hDC,  LPCTSTR pText,  FLOAT Deviation,  FLOAT Extrusion,  LPD3DXMESH\* ppMesh,  LPD3DXBUFFER\* ppAdjacency,  LPGLYPHMETRICSFLOAT pGlyphMetrics  ); |

这个函数如果调用成功则返回D3D\_OK。

* pDevice — 和mesh关联的device。
* hDC — 我们将要用来产生mesh的包含描述字体的设备环境句柄。
* pText — 指向以null结束的字符串的指针，此字符串是用来指定创建什么文字mesh。
* Deviation — 字型轮廓段数间距。该值必须大于等于0。当它为0时，段数等于字体原始设计单位（该值越接近0，那么字体就越光滑）。
* Extrusion — 文字在z轴方向的深度。
* ppMesh — 返回创建的mesh。
* ppAdjacency — 返回创建mesh的相关信息。假如你不需要它可以将其指定为null。
* pGlyphMetrics — 一个指向LPGLYPHMETRICSFLOAT结构数组的指针，它包含了字型米数据。假如你不关心此数据，你可以把它设置为0。

下面的示例代码展示的是使用这个函数来创建一个文字3D 网格模型。

|  |
| --- |
| // Obtain a handle to a device context.  HDC hdc = CreateCompatibleDC( 0 );  // Fill out a LOGFONT structure that describes the font’s properties.  LOGFONT lf;  ZeroMemory(&lf, sizeof(LOGFONT));  lf.lfHeight = 25; // in logical units  lf.lfWidth = 12; // in logical units  lf.lfWeight = 500; // boldness, range 0(light) - 1000(bold)  lf.lfItalic = false;  lf.lfUnderline = false;  lf.lfStrikeOut = false;  lf.lfCharSet = DEFAULT\_CHARSET;  strcpy(lf.lfFaceName, "Times New Roman"); // font style  // Create a font and select that font with the device context.  HFONT hFont;  HFONT hFontOld;  hFont = CreateFontIndirect(&lf);  hFontOld = (HFONT)SelectObject(hdc, hFont);  // Create the 3D mesh of text.  ID3DXMesh\* Text = 0;  D3DXCreateText(\_device, hdc, "Direct3D", 0.001f, 0.4f, &Text, 0, 0);  // Reselect the old font, and free resources.  SelectObject(hdc, hFontOld);  DeleteObject( hFont );  DeleteDC( hdc ); |

现在你便能简单地调用mesh的DrawSubset方法来渲染一个3D文字：

|  |
| --- |
| Text->DrawSubset(0); |

## 9.4摘要(略)

# 第十章 网格模型I (Meshes Part I)

我们已经在D3DXCreate\*程序中使用ID3DXMesh接口工作过了；在这一章我们更详细地介绍这个接口。这一章主要介绍ID3DXMesh接口的数据和方法。

注意ID3DXMesh接口的主要功能继承自ID3DXBaseMesh父接口。了解这些是很重要的，其它一些mesh接口如ID3DXPMesh也是继承自ID3DXBaseMesh。因此在这章中讲的知识也适用于其它的mesh类型。

## 目标

* 学习ID3DXMesh对象的内在数据组织。
* 学习怎样创建一个ID3DXMesh对象。
* 学习怎样优化一个ID3DXMesh对象。
* 学习怎样渲染一个ID3DXMesh对象。

## 10.1 几何信息

ID3DXBaseMesh接口包含一个用来存储网格顶点的顶点缓存和一个用来定义这些顶点怎样连接在一起组成网格三角形的索引缓存。我们能够通过使用下面的方法来得到这些缓存的指针：

|  |
| --- |
| HRESULT ID3DXMesh::GetVertexBuffer(LPDIRECT3DVERTEXBUFFER9\* ppVB);  HRESULT ID3DXMesh::GetIndexBuffer(LPDIRECT3DINDEXBUFFER9\* ppIB); |

这里有一些使用这些方法的例子：

|  |
| --- |
| IDirect3DVertexBuffer9\* vb = 0;  Mesh->GetVertexBuffer( &vb );  IDirect3DIndexBuffer9\* ib = 0;  Mesh->GetIndexBuffer( &ib ); |

假如想锁定这些缓存来读写数据，那么我们能够使用下面的方法。注意这些方法锁定整个顶点/索引缓存。

|  |
| --- |
| HRESULT ID3DXMesh::LockVertexBuffer(DWORD Flags, BYTE\*\* ppData);  HRESULT ID3DXMesh::LockIndexBuffer(DWORD Flags, BYTE\*\* ppData); |

Flags参数描述怎样锁定它。这些Flags参数在第三章中我们介绍过。ppData是函数返回的指向锁定内存的指针的地址。

当然在你锁定以后一定要记得解锁：

|  |
| --- |
| HRESULT ID3DXMesh::UnlockVertexBuffer();  HRESULT ID3DXMesh::UnlockIndexBuffer(); |

下面是另外一些与mesh几何结构有关的ID3DXMesh接口方法：

* DWORD GetFVF() — 返回顶点的格式
* DWORD GetNumVertices() — 返回顶点缓存中的顶点数
* DWORD GetNumBytesPerVertex() — 返回一个顶点所占的字节数
* DWORD GetNumFaces() — 返回在mesh中的面（三角形）数

## 10.2 子集和属性缓存

一个mesh由一个或数个子集组成。一个子集（subset）是在mesh中的使用相同属性渲染的一组三角形。这里的属性是指材质，纹理和渲染状态。图10.1显示了一座房子mesh可能被分成的几个子集。

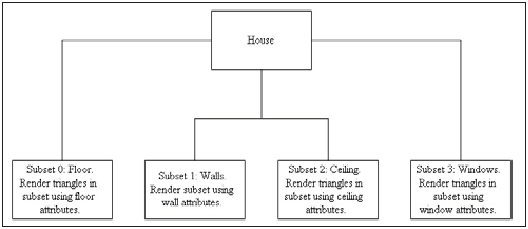


图10.1

我们通过给每个子集指定一个唯一非负整数来标识子集。这个值可以是存储在一个DWORD中的任意数值。例如，在图10.1中我们用0，1，2和3来标识子集。

在mesh中的每个三角形都与一个属性ID相关联，表示该三角形属于该子集。例如，图10.1中组成地板的三角形具有属性ID0，它表示这些三角形属于子集0。同样，组成墙的三角形具有属性ID1，它表示这些三角形属于子集1。

三角形的属性ID存储在mesh的属性缓存中，它是一个DWORD数组。因为每个面对应属性缓存中的一项，所以属性缓存中的项目数等于mesh中的面的个数。属性缓存中的项目和索引缓存中定义的三角形一一对应。即，属性缓存中的第i项和索引缓存中的第i个三角形相对应。三角形i由下面三个索引缓存中的索引项定义：

A = i \* 3

B = i \* 3 + 1

C = i \* 3 + 2

图10.2显示了这个对应关系：

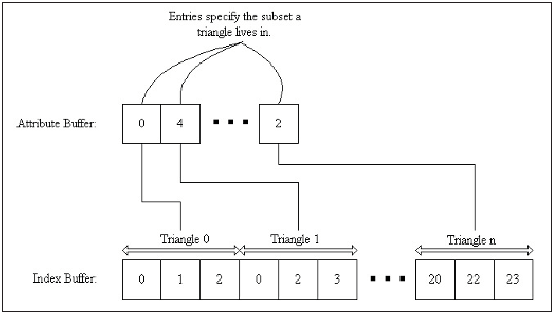


图10.2

我们可以锁定属性缓存，就象下面的代码片段：

|  |
| --- |
| DWORD\* buffer = 0;  Mesh->LockAttributeBuffer(lockingFlags, &buffer);  // Read or write to attribute buffer...  Mesh->UnlockAttributeBuffer(); |

## 10.3 绘制

ID3DXMesh接口提供了DrawSubset（DWORD AttribId）方法来绘制AttribId指示的子集中的各个三角形。例如，要绘制子集0中的所有三角形，我们将这样写：

|  |
| --- |
| Mesh->DrawSubset(0); |

为了绘制整个mesh，我们必须绘制mesh的所有子集。这是非常方便的用0，1，2，…，n-1来标识子集，这里的n是子集的总数。且有一个相对应的材质和纹理数组，即子集i与材质和纹理数组的第i项对应。这就使我们能够简单的用循环来渲染mesh：

|  |
| --- |
| for(int i = 0; i < numSubsets; i++)  {  Device->SetMaterial( mtrls[i] );  Device->SetTexture( 0, textures[i] );  Mesh->DrawSubset(i);  } |

## 10.4 优化

Mesh的顶点和索引能够被重组以便能更有效的渲染mesh。当我们这样做时，我们说我们优化了一个mesh。我们可以使用下面的方法来进行优化：

|  |
| --- |
| HRESULT ID3DXMesh::OptimizeInplace(  DWORD Flags,  CONST DWORD\* pAdjacencyIn,  DWORD\* pAdjacencyOut,  DWORD\* pFaceRemap,  LPD3DXBUFFER\* ppVertexRemap  ); |

* Flags — 表示执行什么类型的优化方法。它可以是下面的一个或几个的组合：
  + D3DXMESHOPT\_COMPACT — 从mesh中移除没有用的顶点和索引项。
  + D3DXMESHOPT\_ATTRSORT — 根据属性给三角形排序并调整属性表，这将使DrawSubset执行更有效（参见10.5节）。
  + D3DXMESHOPT\_VERTEXCACHE — 增加顶点缓存的命中率。
  + D3DXMESHOPT\_STRIPREORDER — 重组顶点索引使三角带尽可能的长。
  + D3DXMESHOPT\_IGNOREVERTS — 只优化索引信息；忽略顶点信息。

注意：D3DXMESHOPT\_VERTEXCACHE和D3DXMESHOPT\_STRIPREORDER不能同时使用。

* pAdjacencyIn — 指向没有优化的mesh的邻接数组。
* pAdjacencyOut — 指向一个DWORD数组，它被用来填充优化好了的mesh邻接信息。该数组必须有ID3DXMesh::GetNumFaces() \* 3个元素。如果不需要该信息，可以将其设置为0。
* pFaceRemap —指向一个DWORD数组，它被用来填充面重影射信息。该数组必须不小于ID3DXMesh::GetNumFaces()。当一个mesh被优化时，由索引缓存定义的面可能被移动；也就是说，在pFaceRemap中的第i项表示第i个原始面被移动到的面索引值。如果不需要该信息，可以将其设置为0。
* ppVertexRemap — 指向ID3DXBuffer指针的地址（参见11.1节），它被用来填充顶点重影射信息。这个缓存应该包含ID3DXMesh::GetNumVertices()个顶点。当一个mesh被优化后，顶点可能被移动。顶点重影射信息用来说明原来的顶点被移动到新位置；也就是说，在ppVertexRemap中的第i项表示原来的第i个顶点的新位置。如果不需要该信息，可以将其设置为0。

例子：

|  |
| --- |
| // Get the adjacency info of the non-optimized mesh.  DWORD adjacencyInfo[Mesh->GetNumFaces() \* 3];  Mesh->GenerateAdjacency(0.0f, adjacencyInfo);  // Array to hold optimized adjacency info.  DWORD optimizedAdjacencyInfo[Mesh->GetNumFaces() \* 3];  Mesh->OptimizeInplace(  D3DXMESHOPT\_ATTRSORT |  D3DXMESHOPT\_COMPACT |  D3DXMESHOPT\_VERTEXCACHE,  adjacencyInfo,  optimizedAdjacencyInfo,  0,  0); |

一个更简单的方法是Optimize方法，它输出一个优化的mesh，而不是在原来mesh的基础上进行优化：

|  |
| --- |
| HRESULT ID3DXMesh::Optimize(  DWORD Flags,  CONST DWORD\* pAdjacencyIn,  DWORD\* pAdjacencyOut,  DWORD\* pFaceRemap,  LPD3DXBUFFER\* ppVertexRemap,  LPD3DXMESH\* ppOptMesh // the optimized mesh to be output  ); |

## 10.5 属性表

当一个mesh被使用D3DXMESHOPT\_ATTRSORT参数来优化后，mesh的几何信息将按照属性进行排序，这样各个子集的顶点/索引将组成连续的块（如图10.3）。

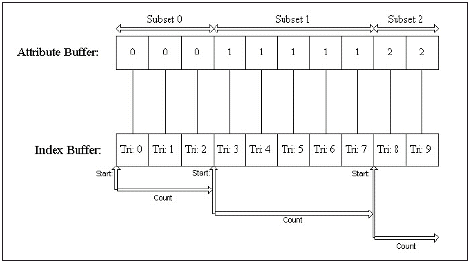


图10.3

除了进行几何信息的排序外，D3DXMESHOPT\_ATTRSORT优化项还将创建一个属性表。该表是D3DXATTRIBUTERANGE结构的一个数组。在属性表中的每一项对应mesh的一个子集并指示顶点/索引缓存中的一个连续连续内存块，这个子集的几何信息就包含在这个块中。D3DXATTRIBUTERANGE结构的定义如下：

|  |
| --- |
| typedef struct \_D3DXATTRIBUTERANGE {  DWORD AttribId;  DWORD FaceStart;  DWORD FaceCount;  DWORD VertexStart;  DWORD VertexCount;  } D3DXATTRIBUTERANGE; |

* AttribId — 子集的ID。
* FaceStart — 该子集的面的起始值，FaceStart\*3就是起始三角形在索引缓存中的序号。
* FaceCount — 在子集中的面（三角形）数。
* VertexStart — 该子集的起始顶点在顶点缓存中的序号。
* VertexCount — 在子集中的顶点数。

我们能够很容易的明白D3DXATTRIBUTERANGE结构的各个成员，如图10.3。在图10.3中mesh的属性表有三项——它们和各个子集一一对应。

建立了属性表以后，渲染一个子集就很容易了。仅仅查一下属性表就能找出自己的几何信息。注意如果没有属性表，每渲染一个子集就需要对属性缓存进行一次线性搜索来找出子集包含的几何信息。

可以使用下面的方法来访问mesh的属性表：

|  |
| --- |
| HRESULT ID3DXMesh::GetAttributeTable(  D3DXATTRIBUTERANGE\* pAttribTable,  DWORD\* pAttribTableSize  ); |

这个方法能够做两件事情：它可以返回属性表的属性数，也可以用属性数据来填充一个D3DXATTRIBUTERANGE结构数组。

要得到属性表的元素个数，可以就将第一个参数设置为0：

|  |
| --- |
| DWORD numSubsets = 0;  Mesh->GetAttributeTable(0, &numSubsets); |

一旦我们知道了属性表的元素个数，我们就能够通过写属性表来填充一个D3DXATTRIBUTERANGE结构数组：

|  |
| --- |
| D3DXATTRIBUTERANGE table = new D3DXATTRIBUTERANGE [numSubsets];  Mesh->GetAttributeTable( table, &numSubsets ); |

我们能够使用ID3DXMesh::SetAttributeTable方法来直接设置属性表。下面的代码就是设置一个有12个子集的属性表：

|  |
| --- |
| D3DXATTRIBUTERANGE attributeTable[12];  // ...fill attributeTable array with data  Mesh->SetAttributeTable( attributeTable, 12); |

## 10.6 邻接信息

对于mesh的某些操作，如优化，有必要了解的是三角形之间的邻接信息。Mesh的邻接数组存储了这些信息。

邻接数组是一个DWORD数组，其中的每一项对应了mesh中的一个三角形。例如，第i项对应的三角形由以下三个索引值定义：

*A* = *i* 3

*B* = *i* 3 + 1

*C* = *i* 3 + 2

注意，使用ULONG\_MAX = 4294967295表示该边没有邻接三角形。我们也可以用-1来表示，因为-1转换成DWORD就是ULONG\_MAX。回想一下，DWORD就是一个unsigned32-bit整数。

因为每个三角形都有三条边，所以他就有三个邻接三角形（如图10.4）。

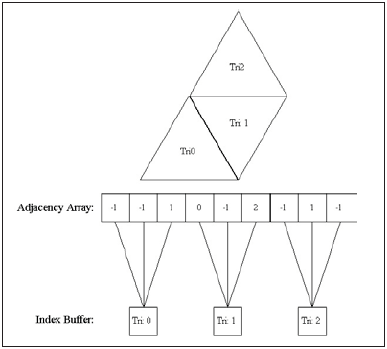


图10.4

因此，邻接数组必须有三项（ID3DXBaseMesh::GetNumFaces()\*3）—— 在mesh中每个三角形都可能有三个邻接三角形。

很多D3Dxmesh创造函数都能输出邻接信息，但我们也可以使用下面的方法：

|  |
| --- |
| HRESULT ID3DXMesh::GenerateAdjacency(  FLOAT fEpsilon,  DWORD\* pAdjacency  ); |

* fEpsilon — 指示当两个点距离有多近时，可以认为是一个点。当两点间的距离小于epsilon时，可认为它们是同一个点。
* pAdjacency — 一个指向填充了邻接信息的DWORD数组指针。

例子：

|  |
| --- |
| DWORD adjacencyInfo[Mesh->GetNumFaces() \* 3];  Mesh->GenerateAdjacency(0.001f, adjacencyInfo); |

## 10.7 复制

有时我们需要将一个mesh中的数据拷贝到另一个之中。我们可以使用ID3DXBaseMesh::CloneMeshFVF方法。

|  |
| --- |
| HRESULT ID3DXMesh::CloneMeshFVF(  DWORD Options,  DWORD FVF,  LPDIRECT3DDEVICE9 pDevice,  LPD3DXMESH\* ppCloneMesh  ); |

* Options — 用来创建mesh的一个或多个创建标志。要了解所有标志信息请查看sdk文档。现在列出一部分：
* D3DXMESH\_32BIT — mesh使用32位索引。
* D3DXMESH\_MANAGED — mesh数据将被放在受控的内存中。
* D3DXMESH\_WRITEONLY — mesh数据只能执行写操作，不能执行读操作。
* D3DXMESH\_DYNAMIC — mesh缓存将是动态的。
* FVF — 创建复制mesh的灵活顶点格式。
* pDevice — 与复制mesh有关的设备。
* ppCloneMesh — 输出复制的mesh。

注意这个方法允许指定与原mesh不同的options和FVF。例如我们有顶点格式为D3DFVF\_XYZ的mesh，现在想复制一个顶点格式为D3DFVF\_XYZ|D3DFVF\_NORMAL的mesh。我们可以这样写：

|  |
| --- |
| // 假设\_mesh和device是有效的  ID3DXMesh\* clone = 0;  Mesh->CloneMeshFVF(  Mesh->GetOptions(), // 使用与源模型同样的选项  D3DFVF\_XYZ | D3DFVF\_NORMAL,// 指定克隆的FVF  Device,  &clone  ); |

## 10.8 创建一个Mesh（D3DXCreateMeshFVF）

我们可以使用D3DXCreate\*函数来创建mesh物体。然而，我们也可以使用 D3DXCreateMeshFVF函数来创建一个空mesh。所谓空mesh是指我们已经指定了顶点数和面数，函数D3DXCreateMeshFVF也分配了适当大小的内存给顶点、顶点索引、属性缓冲区。有了这些缓冲区后，就可以手动填写上下文数据了（需要分别向顶点缓存，索引缓存、属性缓存提供顶点、索引、属性数据）。

我们使用D3DXCreateMeshFVF函数来创建空mesh：

|  |
| --- |
| HRESULT D3DXCreateMeshFVF(  DWORD NumFaces,  DWORD NumVertices,  DWORD Options,  DWORD FVF,  LPDIRECT3DDEVICE9 pDevice,  LPD3DXMESH\* ppMesh  ); |

* NumFaces — mesh将拥有的面数。该值必须大于0。
* NumVertices — mesh将拥有的顶点数。该值必须大于0。
* Options —用来创建mesh的一个或多个创建标志。要了解所有标志信息请查看sdk文档。现在列出一部分：
  + D3DXMESH\_32BIT — mesh使用32位索引。
  + D3DXMESH\_MANAGED — mesh数据将被放在受控的内存中。
  + D3DXMESH\_WRITEONLY — mesh数据只能执行写操作，不能执行读操作。
  + D3DXMESH\_DYNAMIC — mesh缓存将是动态的。
* FVF — mesh的顶点格式。
* pDevice — 与mesh相关的设备。
* ppMesh — 输出创建好的mesh。

下一节将给出实例程序，它演示了用这个函数怎样创建一个mesh以及手动填充mesh的数据内容。

另外，你也可以使用D3DXCreateMesh来创建空mesh。它的原型是：

|  |
| --- |
| HRESULT D3DXCreateMesh(  DWORD NumFaces,  DWORD NumVertices,  DWORD Options,  CONST LPD3DVERTEXELEMENT9\* pDeclaration,  LPDIRECT3DDEVICE9 pDevice,  LPD3DXMESH\* ppMesh  ); |

这些参数和D3DXCreateMeshFVF的参数是非常相似的，除了第四个。作为替代指定的FVF，我们指定一个D3DVERTEXELEMENT9结构，它描述了顶点格式。

|  |
| --- |
| HRESULT D3DXDeclaratorFromFVF(  DWORD FVF, // input format  D3DVERTEXELEMENT9 Declaration[MAX\_FVF\_DECL\_SIZE]//output format  ); |

注意：D3DVERTEXELEMENT9将在第17章中讨论。

这个函数通过输入一个FVF返回一个D3DVERTEXELEMENT9结构的数组。注意MAX\_FVF\_DECL\_SIZE的定义如下：

|  |
| --- |
| typedef enum {  MAX\_FVF\_DECL\_SIZE = 18  } MAX\_FVF\_DECL\_SIZE; |

## 10.9 实例程序：创建和渲染Mesh

这一章的实例程序是渲染一个立方体（如图10.5）



图10.5

它演示了这一章中的大部分功能，包括如下一些操作：

* 创建一个空mesh。
* 用一个立方体几何信息来填充mesh。
* 根据mesh的每个面指定子集。
* 产生mesh的邻接信息。
* 优化mesh。
* 绘制mesh。

注意，我们忽略一些无关的代码来讨论本例。你能在叫做D3DXCreateMeshFVF的例子中找到全部的代码。

另外，为了更容易调试和研究mesh的构成，我们执行如下的函数来将内在内容放进文件中：

|  |
| --- |
| void dumpVertices(std::ofstream& outFile, ID3DXMesh\* mesh);  void dumpIndices(std::ofstream& outFile, ID3DXMesh\* mesh);  void dumpAttributeBuffer(std::ofstream& outFile, ID3DXMesh\* mesh);  void dumpAdjacencyBuffer(std::ofstream& outFile, ID3DXMesh\* mesh);  void dumpAttributeTable(std::ofstream& outFile, ID3DXMesh\* mesh); |

这些函数的名字就显示了它们的功能。执行这些函数是非常简单的，我们在这里讨论时忽略它们（可以看程序的原代码）。在这一节我们只展示一个dumpAttributeTable函数。

我们首先来浏览一下该例子，看看如下的一些全局变量：

|  |
| --- |
| ID3DXMesh\* Mesh = 0;  const DWORD NumSubsets = 3;  IDirect3DTexture9\* Textures[3] = {0, 0, 0};// texture for each subset  std::ofstream OutFile; // used to dump mesh data to file |

这里我们定义了一个mesh对象的指针，我们以后要创建的。我们也定义了mesh拥有的子集数——三。在这个例子中，每个子集都用一个不同的纹理来渲染；纹理数组包含每个子集的纹理，如第i个纹理对应mesh的第i个子集。最后，Outfile变量被用来把mesh的内容输出为一个文本文件。

这个例子的大部分工作是在setup函数中进行。我们首先创建一个空的mesh：

|  |
| --- |
| bool Setup()  {  HRESULT hr = 0;  hr = D3DXCreateMeshFVF(  12,  24,  D3DXMESH\_MANAGED,  Vertex::FVF,  Device,  &Mesh); |

这里我们分配一个有12个面和24个顶点的mesh，这是描述一个盒子所必须的。

这样的话，mesh是空的，因此我们需要将组成盒子的顶点和索引分别写入顶点缓存和索引缓存。锁定顶点/索引缓存并手动写入数据这是很容易的：

|  |
| --- |
| // Fill in vertices of a box  Vertex\* v = 0;  Mesh->LockVertexBuffer(0, (void\*\*)&v);  // fill in the front face vertex data  v[0] = Vertex(-1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f);  v[1] = Vertex(-1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f);  .  .  .  v[22] = Vertex( 1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f);  v[23] = Vertex( 1.0f, -1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f);  Mesh->UnlockVertexBuffer();  // Define the triangles of the box  WORD\* i = 0;  Mesh->LockIndexBuffer(0, (void\*\*)&i);  // fill in the front face index data  i[0] = 0; i[1] = 1; i[2] = 2;  i[3] = 0; i[4] = 2; i[5] = 3;  .  .  .  // fill in the right face index data  i[30] = 20; i[31] = 21; i[32] = 22;  i[33] = 20; i[34] = 22; i[35] = 23;  Mesh->UnlockIndexBuffer(); |

一旦mesh的几何信息被写入，我们必须指定每个三角形在哪个子集中。回想一下属性缓存就是存储的在mesh中每个三角形所属的子集信息。在这个例子中，我们指定索引缓存中的前四个三角形子集为0，接着的四个三角形子集为1，最后四个三角形子集为2。代码如下：

|  |
| --- |
| DWORD\* attributeBuffer = 0;  Mesh->LockAttributeBuffer(0, &attributeBuffer);  for(int a = 0; a < 4; a++) // triangles 1-4  attributeBuffer[a] = 0; // subset 0  for(int b = 4; b < 8; b++) // triangles 5-8  attributeBuffer[b] = 1; // subset 1  for(int c = 8; c < 12; c++) // triangles 9-12  attributeBuffer[c] = 2; // subset 2  Mesh->UnlockAttributeBuffer(); |

现在我们已经创建了一个包含有效数据的mesh。在这一小部分我们将渲染mesh，不过首先还是先将其优化一下。注意虽然这对于一个盒子mesh来说，优化mesh数据没有真正的效果，但是我们还是用ID3DXMesh接口方法来实践一下。为了优化一个mesh，我们首先需要计算mesh的邻接信息：

|  |
| --- |
| std::vector<DWORD> adjacencyBuffer(Mesh->GetNumFaces() \* 3);  Mesh->GenerateAdjacency(0.0f, &adjacencyBuffer[0]); |

然后我们就能够优化mesh了：

|  |
| --- |
| hr = Mesh->OptimizeInplace(  D3DXMESHOPT\_ATTRSORT |  D3DXMESHOPT\_COMPACT |  D3DXMESHOPT\_VERTEXCACHE,  &adjacencyBuffer[0],  0, 0, 0); |

设置好了mesh以后，我们就为渲染它做好了准备。不过在setup函数中还有最后一个问题，也就是在前面我们说的将mesh的内在数据内容写入文件的函数。这能够检查mesh的数据，它能帮助我们调试和学习mesh的结构。

|  |
| --- |
| OutFile.open("Mesh Dump.txt");  dumpVertices(OutFile, Mesh);  dumpIndices(OutFile, Mesh);  dumpAttributeTable(OutFile, Mesh);  dumpAttributeBuffer(OutFile, Mesh);  dumpAdjacencyBuffer(OutFile, Mesh);  OutFile.close();  ...Texturing loading, setting render states, etc., snipped  return true;  } // end Setup() |

例如，dumpAttributeTable函数将属性表的数据写入文件。它的具体实现如下：

|  |
| --- |
| void dumpAttributeTable(std::ofstream& outFile, ID3DXMesh\* mesh)  {  outFile << "Attribute Table:" << std::endl;  outFile << "----------------" << std::endl << std::endl;  // number of entries in the attribute table  DWORD numEntries = 0;  mesh->GetAttributeTable(0, &numEntries);  std::vector<D3DXATTRIBUTERANGE> table(numEntries);  mesh->GetAttributeTable(&table[0], &numEntries);  for(int i = 0; i < numEntries; i++)  {  outFile << "Entry " << i << std::endl;  outFile << "------" << std::endl;  outFile << "Subset ID: " << table[i].AttribId << std::endl;  outFile << "Face Start: " << table[i].FaceStart << std::endl;  outFile << "Face Count: " << table[i].FaceCount << std::endl;  outFile << "Vertex Start: " << table[i].VertexStart << std::endl;  outFile << "Vertex Count: " << table[i].VertexCount << std::endl;  outFile << std::endl;  }  outFile << std::endl << std::endl;  } |

下面的文本文件来自于通过dumpAttributeTable函数得到的mesh Dump.txt文件。

|  |
| --- |
| Attribute Table:  ----------------  Entry 0  ------------  Subset ID: 0  Face Start: 0  Face Count: 4  Vertex Start: 0  Vertex Count: 8  Entry 1  ------------  Subset ID: 1  Face Start: 4  Face Count: 4  Vertex Start: 8  Vertex Count: 8  Entry 2  ------------  Subset ID: 2  Face Start: 8  Face Count: 4  Vertex Start: 16  Vertex Count: 8 |

我们能够了解到我们为mesh所指定的相匹配的数据——有三个子集且每个子集有4个三角形。建议你去看看本例子Dump.txt的完整信息。该文件在本示例文件目录下。

最后，我们使用下面的代码就能够非常容易地渲染mesh了；我们只需要循环每个子集，设置相关联的纹理然后在绘制子集即可。这是非常容易的，因为我们已经为每个子集指定的下标如0，1，2，…，n-1，这里的n就是子集的个数。

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  //...update frame code snipped  Device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER,0x00000000, 1.0f, 0);  Device->BeginScene();  for(int i = 0; i < NumSubsets; i++)  {  Device->SetTexture( 0, Textures[i] );  Mesh->DrawSubset( i );  }  Device->EndScene();  Device->Present(0, 0, 0, 0);  }  return true;  } |

## 10.10 摘要(略)

# 第十一章 网格模型II (Building a Flexible Camera Class)

在这一章我们继续学习mesh相关的接口和结构，以及D3DX库提供的函数。在上一章的基础上，我们能够学习到更多有趣的技巧，比如读取和渲染一个存储在硬盘上的复杂3D模型，并且通过渐进网格接口来控制我们网格细节的等级。

## 目标

* 学习怎样从一个Xfile中读取数据来形成一个ID3DXMesh对象。
* 更进一步理解使用渐进网格的好处以及学习怎样用渐进网格接口ID3DXPMesh。
* 学习关于界线容器，为什么它们是有用的，以及怎样用D3DX函数来创建它们。

## 11.1 ID3DXBuffer

对ID3DXBuffer接口的一些参考是在上一章，这里我们不会详细讲解。在D3DX库中到处都能看见这个接口，因此大概介绍一下该接口还是很有必要的。

ID3DXBuffer接口是一个很普通的数据结构， D3DX用它将数据存储到邻接内存块中。它只有两个方法：

* LPVOID GetBufferPointer()——返回一个指向开始数据的指针。
* DWORD GetBufferSize()——返回在缓存中的字节大小。

为了保持结构特性，它使用一个空指针。也就是说它让我们知道被存储的数据的类型。例如，D3DXLoadMeshFromX使用一个ID3DXBuffer来返回mesh的邻接信息。因为邻接信息是被存储在DWORD数组中的，所以当我们希望使用缓存中的邻接信息时，我们不得不将缓存转换为DWORD数组。

例如：

|  |
| --- |
| DWORD\* info =(DWORD\*)adjacencyInfo->GetBufferPointer();  D3DXMATERIAL\* mtrls = (D3DXMATERIAL\*)mtrlBuffer->GetBufferPointer(); |

因为ID3DXBuffer是一个COM对象，当你使用完以后就必须释放它以防止内存泄漏：

|  |
| --- |
| adjacencyInfo->Release();  mtrlBuffer->Release(); |

我们能够使用下面的方法来创建一个空的ID3DXBuffer：

|  |
| --- |
| HRESULT D3DXCreateBuffer(  DWORD NumBytes, // Size of the buffer, in bytes.  LPD3DXBUFFER \*ppBuffer // Returns the created buffer.  ); |

下面的例子是创建一个能包含4个整数的缓存：

|  |
| --- |
| ID3DXBuffer\* buffer = 0;  D3DXCreateBuffer( 4 \* sizeof(int), &buffer ); |

## 11.2 X文件

迄今为止，我们已经使用过了简单的几何物体，如球体，圆柱体，立方体等，它们都是用D3DXCreate\*函数来创建的。假如你想通过手工指定顶点来创建你自己的3D物体，你能，不用怀疑，不过这是非常枯燥乏味的事情。为了减轻建造3D物体数据的工作，专门的应用程序已经被开发出来了，我们把它们叫做3D建模工具。它们允许我们在一个虚拟的拥有丰富工具的交互环境下建造复杂的真实的mesh，在这建造这些模型都是非常容易的。例如在游戏开发中常用到的有3DSMax（www.discreet.com）,LightWave 3D（www.newtek.com）,以及Maya（[www.aliaswavefront.com](http://www.aliaswavefront.com)）。

这些工具，当然能够输出创建好的mesh数据到文件中。因此，我们也能够写一个文件来提取在我们的3D应用程序中要用到的mesh数据。这的确是一种可行的解决办法。不过，还存在一个更方便的解决方案。它是一种叫做X文件的特殊mesh文件格式（扩展名为.X）。很多3D建模软件都能输出这种格式，当然这里存在一个将其他流行的mesh文件转换为X文件的过程。是什么使X文件这么便利呢？因为它是DirectX定义的格式，并且D3DX库很容易地支持X文件。D3DX库提供了读和写X文件的函数。因此，如果我们使用这种格式就避免了还要自己写程序文件来读/写模型文件了。

注意：你能够下载DirectX9 SDK Extra——你能从MSDN（www.msdn.microsoft.com）上得到一些已经开发好的针对3DMax,LightWave,Maya软件导出.X文件的Direct3D工具包。

### 11.2.1读取X文件

我们使用下面的函数来读取存储在X文件中的mesh数据。注意这个方法创建一个ID3DXMesh对象，且从X文件中读取几何信息数据填入其中。

|  |
| --- |
| HRESULT D3DXLoadMeshFromX(  LPCSTR pFilename,  DWORD Options,  LPDIRECT3DDEVICE9 pDevice,  LPD3DXBUFFER \*ppAdjacency,  LPD3DXBUFFER \*ppMaterials,  LPD3DXBUFFER\* ppEffectInstances,  PDWORD pNumMaterials,  LPD3DXMESH \*ppMesh  ); |

* pFilename — 读取的X文件的文件名。
* Options — 用来创建mesh的一个或多个创建标志。要了解所有标志信息请查看sdk文档。现在列出一部分：
* D3DXMESH\_32BIT — mesh使用32位索引。
* D3DXMESH\_MANAGED — mesh数据将被放在受控的内存中。
* D3DXMESH\_WRITEONLY — mesh数据只能执行写操作，不能执行读操作。
* D3DXMESH\_DYNAMIC — mesh缓存将是动态的。
* pDevice — 与复制mesh有关的设备。
* ppAdjacency — 返回一个ID3DXBuffer包含一个DWORD数组，描述mesh的邻接信息。
* ppMaterials — 返回一个ID3DXBuffer包含一个D3DXMATERIAL结构的数组，存储了mesh的材质数据。我们在下一节介绍mesh材质。
* ppEffectInstances — 返回一个ID3DXBuffer包含一个D3DXEFFECTINSTANCE结构的数组。我们现在通过指定0值来忽略这个参数。
* pNumMaterials — 返回mesh的材质数。
* ppMesh — 返回填充了X文件几何信息的ID3DXMesh对象。

### 11.2.2 X文件的材质

D3DXLoadMeshFromX的第七个参数返回的是mesh包含的材质数，第五个参数返回的是包含着材质数据的一个D3DXMATERIAL结构数组。D3DXMATERIAL结构的定义如下：

|  |
| --- |
| typedef struct D3DXMATERIAL {  D3DMATERIAL9 MatD3D;  LPSTR pTextureFilename;  } D3DXMATERIAL; |

这是一个简单的结构；它包含一个基本的D3DMATERAIL9结构和一个用来指定与之相关联的纹理文件名的一个以null结束的字符串指针。一个X文件是不能插入纹理数据的；它只能插入文件名。因此，在使用D3DXLoadMeshFromX读取一个X文件以后，我们还必须从纹理文件中读取纹理数据。我们将在下一节中说明怎样具体实现。

D3DXLoadMeshFromX函数读取X文件数据以便在返回的D3DXMATERIAL数组中的第i项与第i个子集相对应。因此，子集是使用0，1，2，…，n-1标记的，n是子集和材质的数目。这也就允许使用简单的循环来渲染mesh了。

### 11.2.3 实例程序：X文件

我们现在演示本章中的第一个实例（X文件）的相关代码。该例子调用一个叫做bigship1.x的x文件，你可以在DirectX SDK下的media文件夹下找到它。完整原代码可以在相应的文件中找到。图11.1是该实例的一个截图。



图11.1

该实例使用下面的全局变量：

|  |
| --- |
| ID3DXMesh\* Mesh = 0;  std::vector<D3DMATERIAL9> Mtrls(0);  std::vector<IDirect3DTexture9\*> Textures(0); |

这里有一个ID3DXMesh对象，它被用来存储从X文件中读取的mesh数据。也有一个材质vector和纹理vector,我们用它们来分别存储mesh的材质和纹理。

我们首先在Setup函数中操作。首先，我们读取X文件：

|  |
| --- |
| bool Setup()  {  HRESULT hr = 0;  //  // Load the XFile data.  //  ID3DXBuffer\* adjBuffer = 0;  ID3DXBuffer\* mtrlBuffer = 0;  DWORD numMtrls = 0;  hr = D3DXLoadMeshFromX(  "bigship1.x",  D3DXMESH\_MANAGED,  Device,  &adjBuffer,  &mtrlBuffer,  0,  &numMtrls,  &Mesh);  if(FAILED(hr))  {  ::MessageBox(0, "D3DXLoadMeshFromX() - FAILED", 0, 0);  return false;  } |

读取完X文件数据以后，我们必须遍历D3DXMATERIAL数组来读取mesh中所使用的所有纹理：

|  |
| --- |
| //  // Extract the materials, and load textures.  //  if( mtrlBuffer != 0 && numMtrls != 0 )  {  D3DXMATERIAL\* mtrls = (D3DXMATERIAL\*)mtrlBuffer->GetBufferPointer();  for(int i = 0; i < numMtrls; i++)  {  // the MatD3D property doesn't have an ambient value set  // when its loaded, so set it now:  mtrls[i].MatD3D.Ambient = mtrls[i].MatD3D.Diffuse;  // save the ith material  Mtrls.push\_back( mtrls[i].MatD3D );  // check if the ith material has an associative texture  if( mtrls[i].pTextureFilename != 0 )  {  // yes, load the texture for the ith subset  IDirect3DTexture9\* tex = 0;  D3DXCreateTextureFromFile(  Device,  mtrls[i].pTextureFilename,  &tex);  // save the loaded texture  Textures.push\_back( tex );  }  else  {  // no texture for the ith subset  Textures.push\_back( 0 );  }  }  }  d3d::Release<ID3DXBuffer\*>(mtrlBuffer); // done w/ buffer  .  . // Snipped irrelevant code to this chapter (e.g., setting up lights,  . // view and projection matrices, etc.)  .  return true;  } // end Setup() |

在Display函数中我们让mesh在每一帧中都旋转一个小角度。我们使用简单的循环，Mesh便能够被渲染了：

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  //  // Update: Rotate the mesh.  //  static float y = 0.0f;  D3DXMATRIX yRot;  D3DXMatrixRotationY(&yRot, y);  y += timeDelta;  if( y >= 6.28f )  y = 0.0f;  D3DXMATRIX World = yRot;  Device->SetTransform(D3DTS\_WORLD, &World);  //  // Render  //  Device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER, 0xffffffff, 1.0f, 0);  Device->BeginScene();  for(int i = 0; i < Mtrls.size(); i++)  {  Device->SetMaterial( &Mtrls[i] );  Device->SetTexture(0, Textures[i]);  Mesh->DrawSubset(i);  }  Device->EndScene();  Device->Present(0, 0, 0, 0);  }  return true;  } |

### 11.2.4 产生顶点法线

一个X文件不包含顶点法线数据，这是很有可能的。假如是这种情况，那么手动计算顶点法线以便我们能够使用灯光这是很有必要的。在第5章中我们简要的介绍了一下怎么做。然而，现在我们知道了ID3DXMesh接口和它的父接口ID3DXBaseMesh，我们能够使用下面的函数来产生任何mesh的顶点法线：

|  |
| --- |
| HRESULT D3DXComputeNormals(  LPD3DXBASEMESH pMesh, // Mesh to compute normals of.  const DWORD \*pAdjacency // Input adjacency info.  ); |

这个函数通过使用平均法线的方法来产生顶点法线。假如有邻接信息，那么重复的顶点是被忽略的。假如没有邻接信息，那么重复的顶点也会被重复计算。了解这些是很重要的，我们检查pMash必须有一个包含D3DFVF\_NORMAL标记的顶点格式。

注意假如X文件不包含顶点法线数据，那么通过D3DXLoadMeshFromX创建的ID3DXMesh对象在它的顶点格式中没有指定的D3DFVF\_NORMAL标记。因此，在我们能够使用D3DXComputeNormals之前，我们必须复制mesh并且为其指定包含D3DFVF\_NORMAL的顶点格式。下面就是相应的代码：

|  |
| --- |
| // does the mesh have a D3DFVF\_NORMAL in its vertex format?  if ( !(pMesh->GetFVF() & D3DFVF\_NORMAL) )  {  // no, so clone a new mesh and add D3DFVF\_NORMAL to its format:  ID3DXMesh\* pTempMesh = 0;  pMesh->CloneMeshFVF(  D3DXMESH\_MANAGED,  pMesh->GetFVF() | D3DFVF\_NORMAL, // add it here  Device,  &pTempMesh );  // compute the normals:  D3DXComputeNormals( pTempMesh, 0 );  pMesh->Release(); // get rid of the old mesh  pMesh = pTempMesh; // save the new mesh with normals  } |

## 11.3渐进网格（Progressive Meshes）

渐进网格，它通过ID3DXPMesh接口来表现，允许我们通过简化边缩减转换（*edge collapse transformations*，ECT）来简化mesh。每执行一次ECT就移除一个顶点和一或2个面。因为每个ECT是可逆的（它的逆过程叫顶点分裂），我们能够逆转简化过程并且恢复mesh为它的原始状态。当然，我们不可能得到比原始情况还要精细的网格。我们仅仅只能简化然后恢复简化操作。图11.2显示了同一个mesh的三种不同精细级别（levels of detail，LOD）：高，中，低。



图11.2

渐进网格和mipmaps纹理非常相似。当使用纹理时，我们已经注意到在一个小或远的图元上使用高分辨率的纹理简直就是浪费。对于mesh也是同样的道理；一个小或远的mesh不需要太多三角形，多了也是浪费。因此，我们不会花费渲染高三角形模型的时间来渲染一个只需要表现小的低三角形模型。

我们可以使用渐进网格来根据模型距离摄象机的距离来调整模型的LOD。也就是说，当距离减少时，我们增加mesh的细节，当距离增加时我们减少mesh的细节。

注意我们还没有讨论渐进网格是怎样被实现的；这里我们只讲解怎样使用ID3DXPMesh接口。对此感兴趣的读者可以到渐进网格的原始页面Hoppe上查看。Hoppe的网址：http://research.microsoft.com/~hoppe/。

### 11.3.1 产生一个渐进网格

我们能够使用下面的函数来创建一个ID3DXPMesh对象：

|  |
| --- |
| HRESULT D3DXGeneratePMesh(  LPD3DXMESH pMesh,  CONST DWORD \*pAdjacency,  CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,  CONST FLOAT \*pVertexWeights,  DWORD MinValue,  DWORD Options,  LPD3DXPMESH \*ppPMesh  ); |

* pMesh— 输入原始mesh，它包含了我们想要生成的渐进网格的mesh数据。
* pAdjacency — 指向一个包含pMesh邻接信息的DWORD数组。
* pVertexAttributeWeights — 指向一个D3DXATTRIBUTEWEIGHTS数组，它的大小是pMesh->GetNumVertices（）。它的第i项与pMesh中的第i个顶点相对应并且指定的是它的品质权重。品质权重被用来确定一个顶点被删除的可能性大小。你能够将此参数设置为null，对于每个顶点一个默认的顶点品质权重将被设置。在11.3.2节中有关于顶点品质权重和D3DXATTRIBUTEWEIGHTS结构的更多信息。
* pVertexWeights — 指向一个float数组，它的大小是pMesh->GetNumVertices（），它的第i项与pMesh中的第i个顶点相对应并且指定的是它的顶点权重。顶点权重越高被删除的可能性越小。你能够将此参数设置为null，对于每个顶点一个默认的顶点品质权重1.0将被设置。
* MinValue — 我们想要简化到的最小顶点或面数。注意该值是必须的，而且与顶点/品质权重有关，最终可能达不到该值。
* Options — 只能取D3DXMESHSIMP枚举类型中的一个值：
* D3DXMESHSIMP\_VERTEX — 指定在上一个参数MinValue中提到的数为顶点数。
* D3DXMESHSIMP\_FACE —指定在上一个参数MinValue中提到的数为面数。
* ppPMesh — 返回生成好的渐进网格。

### 11.3.2 顶点品质权重

|  |
| --- |
| typedef struct \_D3DXATTRIBUTEWEIGHTS {  FLOAT Position;  FLOAT Boundary;  FLOAT Normal;  FLOAT Diffuse;  FLOAT Specular;  FLOAT Texcoord[8];  FLOAT Tangent;  FLOAT Binormal;  } D3DXATTRIBUTEWEIGHTS; |

顶点权重结构允许我们为每个顶点属性指定一个权值。0.0表示该属性没有权重。顶点属性的权重越高在简化过程中被移除的可能性越小。默认的权值如下：

|  |
| --- |
| D3DXATTRIBUTEWEIGHTS AttributeWeights;  AttributeWeights.Position = 1.0;  AttributeWeights.Boundary = 1.0;  AttributeWeights.Normal = 1.0;  AttributeWeights.Diffuse = 0.0;  AttributeWeights.Specular = 0.0;  AttributeWeights.Tex[8] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}; |

默认的权值是被推荐的，除非你的应用程序有一个重要的理由而不使用它。

### 11.3.3 ID3DXPMesh方法

ID3DXPMesh接口是继承自ID3DXBaseMesh接口。因此它拥有以前所学习过的ID3DXMesh的所有函数，下面是一些额外的方法：

DWORD GetMaxFaces(VOID)——返回渐进网格能够被设置的最大面数。

DWORD GetMaxVertices(VOID)——返回渐进网格能够被设置的最大顶点数。

DWORD GetMinFaces(VOID)——返回渐进网格能够被设置的最小面数。

DWORD GetMinVertices(VOID)——返回渐进网格能够被设置的最小顶点数。

HRESULT SetNumFaces(DWORD Faces)——这个方法允许我们设置面的个数，以便让mesh简化/复杂化。例如，假设mesh目前有50个面，我们现在想将它简化到30个面；我们将写成：

|  |
| --- |
| pmesh->SetNumFaces(30); |

注意调整后的面数可能并不是我们设定的面数。假如面数小于了GetMinFaces（），那么面数将为GetMinFaces（）。同样的，假如面数大于了GetMaxFaces（），那么面数将为GetMaxFaces（）。

HRESULT SetNumVertices(DWORD Vertices)——这个方法允许我们设置顶点的个数，以便让mesh简化/复杂化。例如，假设mesh目前有20个顶点，我们现在想将它增加到40个；我们将写成：

|  |
| --- |
| pmesh->SetNumVertices(40); |

注意调整后的顶点数可能并不是我们设定的数。假如顶点数小于了GetMinVertices（），那么顶点数将为GetMinVertices（）。同样的，假如顶点数大于了GetMaxVertices（），那么顶点数将为GetMaxVertices（）。

HRESULT TrimByFaces(

DWORD NewFacesMin,

DWORD NewFacesMax,

DWORD \*rgiFaceRemap, // Face remap info.

DWORD \*rgiVertRemap // Vertex remap info.

);

这个方法允许我们设置新的最小和最大面数，分别通过NewFacesMin和NewFacesMax指定。注意新的最小和最大值必须在现有最小和最大面数之间；也就是说，必须在[GetMinFaces（），GetMaxFaces（）]之中。该函数也返回面和顶点的重影射信息。重影射信息参见10.4节。

HRESULT TrimByVertices(

DWORD NewVerticesMin,

DWORD NewVerticesMax,

DWORD \*rgiFaceRemap, // Face remap info.

DWORD \*rgiVertRemap // Vertex remap info.

);

这个方法允许我们设置新的最小和最大顶点数，分别通过NewVerticesMin和NewVerticesMax指定。注意新的最小和最大值必须在现有最小和最大顶点数之间；也就是说，必须在[GetMinVertices（），GetMaxVertices（）]之中。该函数也返回面和顶点的重影射信息。重影射信息参见10.4节。

### 11.3.4实例程序：渐进网格

渐进网格例子与X文件例子很相似，除了实际上我们创建和渲染的是一个渐进网格，通过ID3DXPMesh接口来表现。我们允许用户通过键盘输入进行交互式地改变渐进网格。你能通过按A键来增加mesh的面数，按S键来减少mesh的面数。

在这个例子中使用的全局变量和X文件例子中的是一样的，不过我们增加了一个用来存储渐进网格的变量：

|  |
| --- |
| ID3DXMesh\* SourceMesh = 0;  ID3DXPMesh\* PMesh = 0; // progressive mesh  std::vector<D3DMATERIAL9> Mtrls(0);  std::vector<IDirect3DTexture9\*> Textures(0); |

回想一下，为了得到一个渐进网格我们必须输入一个包含了数据信息的源mesh。因此，我们首先读取一个X文件数据到ID3DXMesh对象SourceMesh之中，然后再产生渐进网格：

|  |
| --- |
| bool Setup()  {  HRESULT hr = 0;  // ...Load XFile data into SourceMesh snipped.  //  // ...Extracting materials and textures snipped. |

因为这一部分代码和X文件例子中的是完全一样的，在这里我们就把它省略了。一但有了源mesh，我们就能够象下面一样来生成渐进网格了：

|  |
| --- |
| //  // Generate the progressive mesh.  //  hr = D3DXGeneratePMesh(  SourceMesh,  (DWORD\*)adjBuffer->GetBufferPointer(), // adjacency  0, // default vertex attribute weights  0, // default vertex weights  1, // simplify as low as possible  D3DXMESHSIMP\_FACE, // simplify by face count  &PMesh);  d3d::Release<ID3DXMesh\*>(SourceMesh); // done w/ source mesh  d3d::Release<ID3DXBuffer\*>(adjBuffer); // done w/ buffer  if(FAILED(hr))  {  ::MessageBox(0, "D3DXGeneratePMesh() - FAILED", 0, 0);  return false;  } |

注意，因为顶点/品质权值的缘故，很难将Mesh简化到只有一个面，但是，如果将其指定为1，则可以将Mesh简化到最低。

在这一点上，渐进网格已经被产生了，但是假如你现在就渲染它，它将以最简化的方式来渲染。以为我们想开始渲染最高精度的mesh，所以我们设置它为：

|  |
| --- |
| // set to original detail  DWORD maxFaces = PMesh->GetMaxFaces();  PMesh->SetNumFaces(maxFaces); |

在Display函数中，我们测试A键和S键并将结果输入。

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  //  // Update: Mesh resolution.  //  // Get the current number of faces the pmesh has.  int numFaces = PMesh->GetNumFaces();  // Add a face, note the SetNumFaces() will automatically  // clamp the specified value if it goes out of bounds.  if( ::GetAsyncKeyState('A') & 0x8000f )  {  // Sometimes we must add more than one face to invert  // an edge collapse transformation  PMesh->SetNumFaces( numFaces + 1 );  if( PMesh->GetNumFaces() == numFaces )  PMesh->SetNumFaces( numFaces + 2 );  }  // Remove a face, note the SetNumFaces() will automatically  // clamp the specified value if it goes out of bounds.  if( ::GetAsyncKeyState('S') & 0x8000f )  PMesh->SetNumFaces( numFaces - 1 ); |

这是很简单的，但是要注意当增加面时我们有时必须增加两个面来完成ECT。

最后，我们就能象渲染ID3DXMesh对象一样来渲染ID3DXPMesh对象。另外，为了更加直观的观察网格的三角形数的变化情况，使用黄色材质在线框模式（Wireframe Mode）下渲染Mesh的三角形。

|  |
| --- |
| Device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER, 0xffffffff, 1.0f, 0);  Device->BeginScene();  for(int i = 0; i < Mtrls.size(); i++)  {  // draw pmesh  Device->SetMaterial( &Mtrls[i] );  Device->SetTexture(0, Textures[i]);  PMesh->DrawSubset(i);  // draw wireframe outline  Device->SetMaterial(&d3d::YELLOW\_MTRL);  Device->SetRenderState(D3DRS\_FILLMODE, D3DFILL\_WIREFRAME);  PMesh->DrawSubset(i);  Device->SetRenderState(D3DRS\_FILLMODE, D3DFILL\_SOLID);  }  Device->EndScene();  Device->Present(0, 0, 0, 0);  }  return true;  } |



图11.3

## 11.4 界线容积（Bounding Volumes）

有时我们需要计算mesh的界线容积（边界范围）。常用的有两种类型：立方体和球。也有使用其它方法的，如圆柱体，椭球体，菱形体，胶囊形。图11.4演示了对同一个mesh分别使用立方体和球体类型。这一节我们只讨论立方体和球体两种边界形式。

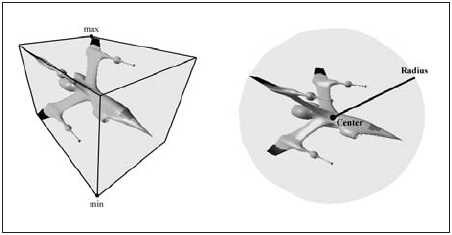


图11.4

边界盒/球常常被用来加速可见性测试，碰撞检测等。例如，假如一个mesh的边界盒/球不可见，那么我们就说mesh不可见。一个盒/球可见性测试是比分别测试mesh中的每个三角形要廉价的多。对于一个碰撞检测例子，如果一枚导弹点火起飞，我们需要检测它是否击中了同一场景中的目标。由于这些物体都是由大量三角形构成，我们可以依次检测每个对象的每个三角形，来测试导弹（可以用射线数学模型）是否碰撞到了这些三角形。这个方法需要进行多次的射线/三角形交点的运算。一个更好的方法是使用边界盒或边界球，计算射线与场景中的每个对象的边界盒/边界球的交点。如果射线与对象的边界范围相交，可以认为该对象被击中了。这是一个公平的近似方法，如果需要更高的精度，可以用边界范围法先去除那些明显不会相撞的对象，然后用更精确地方法检测很可能相撞的对象。如果边界范围检测发现相撞，则该对象就很有可能相撞。

D3DX库提供了计算mesh的边界盒和边界球的函数。这些函数使用顶点数组作为输入计算边界盒/球。这些函数本来就是设计的很灵活的，它们可以使用各种顶点格式：

|  |
| --- |
| HRESULT D3DXComputeBoundingSphere(  LPD3DXVECTOR3 pFirstPosition,  DWORD NumVertices,  DWORD dwStride,  D3DXVECTOR3\* pCenter,  FLOAT\* pRadius  ); |

* pFirstPosition——指向在顶点数组中第一个顶点的向量，描述顶点位置。
* NumVertices——在顶点数组中的的顶点数。
* dwStride——每个顶点的字节大小。这是很需要的，因为顶点结构可能有一些额外信息如法向量和纹理坐标，这些信息对计算边界又没有用，函数需要知道应该跳过多少字节来得到下一个顶点的位置。
* pCenter——返回边界球的中心。
* pRadius——返回边界球的半径。

|  |
| --- |
| HRESULT D3DXComputeBoundingBox(  LPD3DXVECTOR3 pFirstPosition,  DWORD NumVertices,  DWORD dwStride,  D3DXVECTOR3\* pMin,  D3DXVECTOR3\* pMax  ); |

前三个参数和D3DXComputeBoundingSphere的前三个参数是完全一样的。最后两个参数分别用来返回边界盒的最小和最大点。

### 11.4.1一些新的特殊常量

让我来介绍两个常量，它们在本书中是经常要用到的。我们把它们添加到d3d名称空间中：

|  |
| --- |
| namespace d3d  {  ...  const float INFINITY = FLT\_MAX;  const float EPSILON = 0.001f; |

常量INFINITY是用来表示一个浮点数所能存储的最大数。因为我们找不到一个比FLT\_MAX还要大的浮点数，我们可以将它视为无穷大。常量EPSILON是一个很小的值，我们这样定义它，凡是比它小的数就视为0。这也是很有必要的，因为得到的浮点是不精确的，一个被读作0的数可能有一点点小偏差。因此，让它和0比较相等肯定会失败。我们因此可以通过把该值与0的差值与EPSILON比较来确定是否相等：

|  |
| --- |
| bool Equals(float lhs, float rhs)  {  // if lhs == rhs their difference should be zero  return fabs(lhs - rhs) < EPSILON ? true : false;  } |

### 11.4.2界线容积类型

为了更容易的使用边界盒和边界球，我们将它们分别封装到两个类中。现在在d3d名称空间中定义类：

|  |
| --- |
| struct BoundingBox  {  BoundingBox();  bool isPointInside(D3DXVECTOR3& p);  D3DXVECTOR3 \_min;  D3DXVECTOR3 \_max;  };  struct BoundingSphere  {  BoundingSphere();  D3DXVECTOR3 \_center;  float \_radius;  };  d3d::BoundingBox::BoundingBox()  {  // infinite small bounding box  \_min.x = d3d::INFINITY;  \_min.y = d3d::INFINITY;  \_min.z = d3d::INFINITY;  \_max.x = -d3d::INFINITY;  \_max.y = -d3d::INFINITY;  \_max.z = -d3d::INFINITY;  }  bool d3d::BoundingBox::isPointInside(D3DXVECTOR3& p)  {  // is the point inside the bounding box?  if(p.x >= \_min.x && p.y >= \_min.y && p.z >= \_min.z &&  p.x <= \_max.x && p.y <= \_max.y && p.z <= \_max.z)  {  return true;  }  else  {  return false;  }  }  d3d::BoundingSphere::BoundingSphere()  {  \_radius = 0.0f;  } |

### 11.4.3实例程序：界线容积

在这一章中被叫做界线容积的实例程序主要是演示使用D3DXComputeBoundingSphere和D3DXComputeBoundingBox。程序读取一个X文件并且计算该mesh的边界球。它创建两个ID3DXMesh对象，一个用来作为边界球模型一个用来作为边界盒模型。X文件生成的mesh被渲染，其中的边界球或边界盒不可见（如图11.5）。你能够通过敲空格键来再边界球和边界盒之间切换。



图11.5

这个例子是非常简单的，我们列出你要学习的代码。我们实现的两个函数是用来计算网格的边界球和边界盒的：

|  |
| --- |
| bool ComputeBoundingSphere(  ID3DXMesh\* mesh, // mesh to compute bounding sphere for  d3d::BoundingSphere\* sphere) // return bounding sphere  {  HRESULT hr = 0;  BYTE\* v = 0;  mesh->LockVertexBuffer(0, (void\*\*)&v);  hr = D3DXComputeBoundingSphere(  (D3DXVECTOR3\*)v,  mesh->GetNumVertices(),  D3DXGetFVFVertexSize(mesh->GetFVF()),  &sphere->\_center,  &sphere->\_radius);  mesh->UnlockVertexBuffer();  if( FAILED(hr) )  return false;  return true;  }  bool ComputeBoundingBox(  ID3DXMesh\* mesh, // mesh to compute bounding box for  d3d::BoundingBox\* box) // return bounding box  {  HRESULT hr = 0;  BYTE\* v = 0;  mesh->LockVertexBuffer(0, (void\*\*)&v);  hr = D3DXComputeBoundingBox(  (D3DXVECTOR3\*)v,  mesh->GetNumVertices(),  D3DXGetFVFVertexSize(mesh->GetFVF()),  &box->\_min,  &box->\_max);  mesh->UnlockVertexBuffer();  if( FAILED(hr) )  return false;  return true;  } |

注意，类型转换(D3DXVECTOR3\*)v假定顶点位置成员是被存储在我们所使用的顶点结构的开始位置。同样要注意我们能够使用D3DXGetFVFVertexSize函数来得到顶点结构的大小。

## 11.5 摘要(略)

# 第十二章 创建灵活的摄像机类 (Building a Flexible Camera Class)

迄今，我们已经使用过D3DXMatrixLookAtLH函数来计算视图空间变换矩阵。这个函数对于在固定位置布置和对准摄像机是非常好用的，不过它的用户接口对于要响应用户输入来实现摄像机移动就不那么好用了。这就激发我们用我们自己的方法来解决。在这一章我们展示了怎样实现一个Camera类，它使我们能够比D3DXMatrixLookAtLH函数更好地操作摄像机，并且可以用来作为飞行模拟摄像机和第一人称视角摄像机。

## 目标

* 学习怎样实现一个灵活的摄像机类，它可以用作飞行模拟摄像机和第一人称视角摄像机。

## 12.1 摄像机设计

我们定义一个相对于世界坐标系的位置和摄像机的方向，这里使用四个摄像机向量：*right vector* , *up vector*, *look vector* 以及 *position vector*, 如图12.1所示。这些向量用来为摄像机定义一个坐标系来描述在世界坐标中的对应关系。因为 right ，up 和 look 向量定义了摄像机在世界中的方向，我们有时把它们三个向量一起称为方向向量（*orientation vectors*）。方向向量必须被标准化。假如彼此互相垂直且都是单位长度，那么我们就称它们是正交标准化向量。我们做这些限制是因为等一会儿我们要将方向向量插入到一个行矩阵中。因为行向量是正交标准化的，所以该矩阵也就是直交矩阵。回忆一下，直交矩阵有一个特性就是它的逆矩阵等于它的转置矩阵。这在等一下的12.2.1.2节中是很有用的。

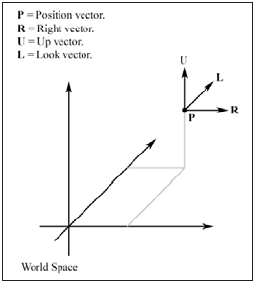


图12.1

有了这四个向量来描述摄像机，我们的摄像机就能够按照下面六种方式变化了：

* + 围绕right向量旋转（pitch倾斜）
  + 围绕up向量旋转（yaw 偏航）
  + 围绕look向量旋转（roll 滚转）
  + 沿着right向量平移（strafe）
  + 沿着up向量飞行（fly）
  + 沿着look向量移动（move）

通过这六种操作，我们能够沿着三个轴移动以及饶着三个轴旋转，这给了我们一个六度的自由。下面的Camera类定义了我们要的描述数据以及想要的方法：

|  |
| --- |
| class Camera  {  public:  enum CameraType { LANDOBJECT, AIRCRAFT };  Camera();  Camera(CameraType cameraType);  ~Camera();  void strafe(float units); // left/right  void fly(float units); // up/down  void walk(float units); // forward/backward  void pitch(float angle); // rotate on right vector  void yaw(float angle); // rotate on up vector  void roll(float angle); // rotate on look vector  void getViewMatrix(D3DXMATRIX\* V);  void setCameraType(CameraType cameraType);  void getPosition(D3DXVECTOR3\* pos);  void setPosition(D3DXVECTOR3\* pos);  void getRight(D3DXVECTOR3\* right);  void getUp(D3DXVECTOR3\* up);  void getLook(D3DXVECTOR3\* look);  private:  CameraType \_cameraType;  D3DXVECTOR3 \_right;  D3DXVECTOR3 \_up;  D3DXVECTOR3 \_look;  D3DXVECTOR3 \_pos;  }; |

在类中我们定义了一个还没有讨论的CameraType枚举类型。目前，我们的摄像机支持两种摄像机模式，LANDOBJECT模式和AIRCRAFT模式。AIRCRAFT模式允许我们在空间中完全自由的移动。不过，在有些游戏中，比如第一人称设计游戏，人是不能飞的；因此我们必须限制它在某些轴上的运动。指定为LANDOBJECT模式的摄像机就限制了这些，你可以在下一部分看见。

## 12.2 执行详细资料

### 12.2.1计算视图矩阵

我们现在演示怎样根据摄像机向量来计算视图矩阵变换的。让 **p** = (*px*, *py*, *pz*), **r** = (*rx*, *ry*, *rz*), **u** = (*ux*, *uy*, *uz*) 以及 **d** = (*dx*, *dy*, *dz*) 分别表示 position, right, up 以及 look 向量。

回忆第二章我们所说的，视图空间变换是指在世界坐标系中进行几何变换以便将照相机平移变换到坐标系的源点并把它的方向旋转至朝向Z轴的正方向（如图12.2）。



图12.2

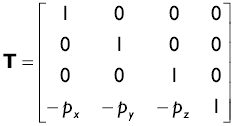
因此，我们希望有一个象这样的变换矩阵**V** ：

* **pV** = (0, 0, 0)—矩阵**V**能将摄像机移动到原点。
* **rV** = (1, 0, 0)—矩阵**V**能将摄像机的right向量与世界坐标系中的x轴对齐。
* **uV** = (0, 1, 0)—矩阵**V**能将摄像机的up向量与世界坐标系中的y轴对齐。
* **dV** = (0, 0, 1)—矩阵**V**能将摄像机的look向量与世界坐标系中的z轴对齐。

我们能将变换任务分为两个部分：1）平移部分，将摄像机的位置移动到原点；2）旋转部分，将摄像机的方向向量与世界坐标系的轴对齐。

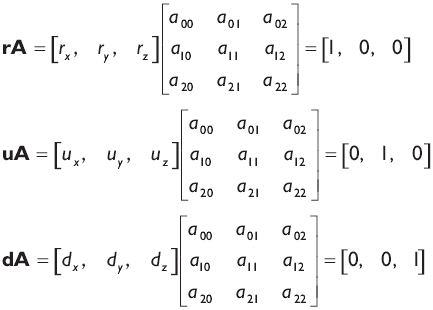
### 12.2.1.1 第一部分：平移

平移只需要利用 –**p** 就可简单地将 **p** 移动到原点，因为 **p**–**p=0**。因此我们能够用下面的矩阵来描述视图变换中的平移部分：



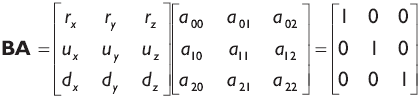
### 12.2.1.2 第二部分：旋转

矫正摄像机的三个方向向量使其与世界坐标系的轴对齐需要更多的工作。我们需要一个3\*3的旋转矩阵**A** ，它能将right，up和look分别与x-，y-以及z轴对齐。这个矩阵将满足如下三个等式：

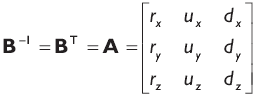


注意：我们在这里使用3\*3矩阵来工作是因为现在不需要额外的信息来表现旋转。等一下我们将它增加到常用的4\*4矩阵。

因为这三个等式都有一个相同系数矩阵**A** ，所以我们能够把它们合在一起。我们把它们从新写到一起来：

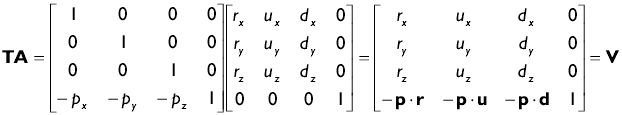


求**A**有很多方法，但是我们知道**A**是**B**逆矩阵因为**BA = BB-1 = I**。因为B 是一个直交矩阵（它的行向量是正交标准化的），我们知道它的逆矩阵就是它的转置矩阵。因此，将方向向量和世界坐标系中的坐标轴对齐的变换如下：



### 12.2.1.3 将两部分合并

最后，将**A**增加为4\*4矩阵，同时将平移部分合并到旋转部分形成的视图变换矩阵**V**：



我们在Camera::getViewMatrix方法中建立这个矩阵：

|  |
| --- |
| void Camera::getViewMatrix(D3DXMATRIX\* V)  {  // Keep camera's axes orthogonal to eachother  D3DXVec3Normalize(&\_look, &\_look);  D3DXVec3Cross(&\_up, &\_look, &\_right);  D3DXVec3Normalize(&\_up, &\_up);  D3DXVec3Cross(&\_right, &\_up, &\_look);  D3DXVec3Normalize(&\_right, &\_right);  // Build the view matrix:  float x = -D3DXVec3Dot(&\_right, &\_pos);  float y = -D3DXVec3Dot(&\_up, &\_pos);  float z = -D3DXVec3Dot(&\_look, &\_pos);  (\*V)(0,0) = \_right.x; (\*V)(0, 1) = \_up.x; (\*V)(0, 2) = \_look.x; (\*V)(0, 3) = 0.0f;  (\*V)(1,0) = \_right.y; (\*V)(1, 1) = \_up.y; (\*V)(1, 2) = \_look.y; (\*V)(1, 3) = 0.0f;  (\*V)(2,0) = \_right.z; (\*V)(2, 1) = \_up.z; (\*V)(2, 2) = \_look.z; (\*V)(2, 3) = 0.0f;  (\*V)(3,0) = x; (\*V)(3, 1) = y; (\*V)(3, 2) = z; (\*V)(3, 3) = 1.0f;  } |

你可能想知道方法中前面几行代码是干什么的。在几次旋转后，摄像机的方向向量可能变的不相互垂直了。因此，每当该函数被调用时，我们根据look向量从新计算up和right向量，使它们保持相互垂直。新的up向量是这样计算的up = look × right。 接着新的right向量是这样计算的right = up × look。

### 12.2.2围绕任意轴旋转

为了实现我们的摄像机旋转方法，我们需要能够绕着任意轴旋转。D3DX库提供下面的函数来解决这个问题：

|  |
| --- |
| D3DXMATRIX \*D3DXMatrixRotationAxis(  D3DXMATRIX \*pOut, // returns rotation matrix  CONST D3DXVECTOR3 \*pV, // axis to rotate around  FLOAT Angle // angle, in radians, to rotate  ); |



图12.3

例如，假如我们想绕向量（0.707, 0.707, 0）轴旋转π/2角度。我们可以这样写：

|  |
| --- |
| D3DXMATRIX R;  D3DXVECTOR3 axis(0.707f, 0.707f, 0.0f);  D3DXMatrixRotationAxis(&R, &axis, D3DX\_PI / 2.0f); |

D3DXMatrixRotationAxis的变换矩阵的来源你可以在Eric Lengyel的 *Mathematics for 3D Game Programming &Computer Graphics*中找到。

### 12.2.3 Pitch、Yaw和Roll

因为方向向量描述了摄像机相对于世界坐标系的方向，我们必须考虑在使用倾斜（pitch）、偏航（yaw）和滚转（roll）时及时更新方向向量。这其实也是非常简单的。图12.4，12.5，12.6分别显示了摄像机的倾斜、偏航和滚转操作。

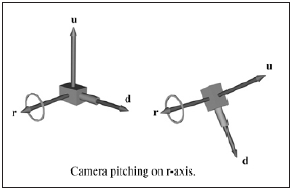


图12.4

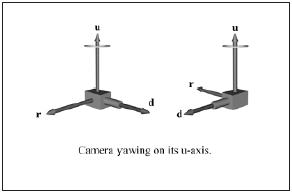


图12.5

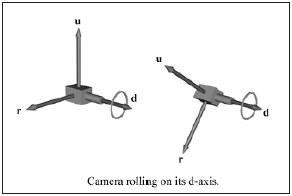


图12.6

当倾斜（pitch）时，我们需要将up和look向量绕着right向量旋转一定角度。同样的，当偏航（yaw）时，我们需要将look和right向量绕着up向量旋转一定角度。最后，当滚转（roll）时，我们需要将up和right向量绕着look向量旋转一定角度。

我们现在明白了为什么D3DXMatrixRotationAxis函数是非常必要的，因为这三个向量中的任何一个都可能围绕世界坐标系中的任意轴旋转。

对于倾斜（pitch）、偏航（yaw）和滚转（roll）的执行我们已经讨论了。然而，对于LANDOBJECT模式就有一些限制。我们在偏航（yaw）方法中只围绕y轴旋转，我们完全屏蔽滚转（roll）。当然你可以根据你的程序需要来改变Camera类。我们这里只是一个示例而已。

倾斜（pitch）、偏航（yaw）和滚转（roll）方法代码的具体实现如下：

|  |
| --- |
| void Camera::pitch(float angle)  {  D3DXMATRIX T;  D3DXMatrixRotationAxis(&T, &\_right, angle);  // rotate \_up and \_look around \_right vector  D3DXVec3TransformCoord(&\_up,&\_up, &T);  D3DXVec3TransformCoord(&\_look,&\_look, &T);  }  void Camera::yaw(float angle)  {  D3DXMATRIX T;  // rotate around world y (0, 1, 0) always for land object  if( \_cameraType == LANDOBJECT )  D3DXMatrixRotationY(&T, angle);  // rotate around own up vector for aircraft  if( \_cameraType == AIRCRAFT )  D3DXMatrixRotationAxis(&T, &\_up, angle);  // rotate \_right and \_look around \_up or y-axis  D3DXVec3TransformCoord(&\_right,&\_right, &T);  D3DXVec3TransformCoord(&\_look,&\_look, &T);  }  void Camera::roll(float angle)  {  // only roll for aircraft type  if( \_cameraType == AIRCRAFT )  {  D3DXMATRIX T;  D3DXMatrixRotationAxis(&T, &\_look, angle);  // rotate \_up and \_right around \_look vector  D3DXVec3TransformCoord(&\_right,&\_right, &T);  D3DXVec3TransformCoord(&\_up,&\_up, &T);  }  } |

### 12.2.4 Walking、Strafing和Flying

当提到walking时，我们的意思是在我们观察的方向上移动位置（也就是说，沿着look向量）。Strafing是说在我们观察方向的左右移动，也就是沿着right向量移动。最后，我们说flying就是沿着up向量移动。为了沿着这些轴移动，我们只需要简单地加一个向量就可以了（如图12.7）。

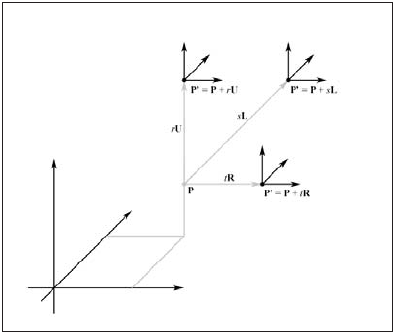


图12.7

就象旋转一样，我们需要对移动作一些限制。例如，LANDOBJECT不允许飞起来。因此我们把移动限制在xz平面。然而，因为LANDOBJECT能够允许爬楼梯和登山，所以，我们设置Camera::setPosition方法，它允许你手动设置你的摄像机位置来达到你的高度和位置。

移动（walk）、平移（strafe）和飞行（fly）方法代码的具体实现如下：

|  |
| --- |
| void Camera::walk(float units)  {  // move only on xz plane for land object  if( \_cameraType == LANDOBJECT )  \_pos += D3DXVECTOR3(\_look.x, 0.0f, \_look.z) \* units;  if( \_cameraType == AIRCRAFT )  \_pos += \_look \* units;  }  void Camera::strafe(float units)  {  // move only on xz plane for land object  if( \_cameraType == LANDOBJECT )  \_pos += D3DXVECTOR3(\_right.x, 0.0f, \_right.z) \* units;  if( \_cameraType == AIRCRAFT )  \_pos += \_right \* units;  }  void Camera::fly(float units)  {  // move only on y-axis for land object  if( \_cameraType == LANDOBJECT )  \_pos.y += units;  if( \_cameraType == AIRCRAFT )  \_pos += \_up \* units;  } |

## 12.3实例程序：摄像机

这一章的实例程序是创建和渲染一个如图12.8所示的场景。你能够通过键盘输入在场景中自由地飞行。下面是相应键盘设置：

* W/S—向前/向后移动
* A/D—向左/向右平移
* R/F—向上/向下飞行
* Up/Down方向键—倾斜
* Left/Right方向键—偏航
* N/M—滚转



图12.8

例子的执行是非常简单的，因为所有工作都包含在摄像机类中了，这些我们都已经讨论过了。我们在Display函数中获得键盘的输入。记住，我们在全局域中实例化了一个摄像机类对象TheCamera。同样注意我们使用时间变化量来控制移动摄像机；这可以排除帧速度的影响而稳定地移动。

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  //  // Update: Update the camera.  //  if( ::GetAsyncKeyState('W') & 0x8000f )  TheCamera.walk(4.0f \* timeDelta);  if( ::GetAsyncKeyState('S') & 0x8000f )  TheCamera.walk(-4.0f \* timeDelta);  if( ::GetAsyncKeyState('A') & 0x8000f )  TheCamera.strafe(-4.0f \* timeDelta);  if( ::GetAsyncKeyState('D') & 0x8000f )  TheCamera.strafe(4.0f \* timeDelta);  if( ::GetAsyncKeyState('R') & 0x8000f )  TheCamera.fly(4.0f \* timeDelta);  if( ::GetAsyncKeyState('F') & 0x8000f )  TheCamera.fly(-4.0f \* timeDelta);  if( ::GetAsyncKeyState(VK\_UP) & 0x8000f )  TheCamera.pitch(1.0f \* timeDelta);  if( ::GetAsyncKeyState(VK\_DOWN) & 0x8000f )  TheCamera.pitch(-1.0f \* timeDelta);  if( ::GetAsyncKeyState(VK\_LEFT) & 0x8000f )  TheCamera.yaw(-1.0f \* timeDelta);  if( ::GetAsyncKeyState(VK\_RIGHT) & 0x8000f )  TheCamera.yaw(1.0f \* timeDelta);  if( ::GetAsyncKeyState('N') & 0x8000f )  TheCamera.roll(1.0f \* timeDelta);  if( ::GetAsyncKeyState('M') & 0x8000f )  TheCamera.roll(-1.0f \* timeDelta);  // Update the view matrix representing the cameras  // new position/orientation.  D3DXMATRIX V;  TheCamera.getViewMatrix(&V);  Device->SetTransform(D3DTS\_VIEW, &V);  //  // Render  //  Device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER, 0x00000000, 1.0f, 0);  Device->BeginScene();  d3d::DrawBasicScene(Device, 1.0f);  Device->EndScene();  Device->Present(0, 0, 0, 0);  }  return true;  } |

注意：我们已经用一个新的函数DrawBasicScene更新了d3d名称空间。这个函数绘制了如图12.8的场景。我们已经将其添加进了d3d名称空间，这是因为对于建立一个基本的场景它是一个非常方便的函数。以后的例子我们就可以集中精力在例子代码中而不需要关注这些不相关的绘制场景的代码了。它是在d3dUtility.h中被声明的：

|  |
| --- |
| // Function references "desert.bmp" internally. This file must  // be in the working directory.  bool DrawBasicScene(  IDirect3DDevice9\* device,// Pass in 0 for cleanup.  float scale); // uniform scale |

如果该函数不能任何东西也就是什么都显示不出来，你就需要看看相应的代码了。你可以在本章的代码中找到它。注意这个函数需要调用一张desert.bmp图片用作纹理。当然该文件也可以在同一个文件夹下找到。

## 12.4 摘要

我们以四个向量来描述在世界坐标系中照相机的位置和方向：right、up、look、position向量，藉由这个描述, 我们能轻易的实现一个自由的六角度照相机，为游戏中的模拟飞行器、第一人称视角的游戏玩家提供了一个灵活的照相机接口。

# 第十三章 地形渲染基础 (Basic Terrain Rendering)

实际上，地形网格不比三角形网格复杂，图13.1.(a)所示，网络的每个顶点指定了高度，格子模型用这种方式显示从山脉到河流的平滑过渡。图13.1 (b)，模拟自然地形。当然，我们可以用漂亮的纹理表现沙石地，绿色的山丘。图13.1.(c)雪山效果。

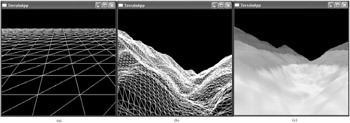


图 13.1: (a) 三角网格. (b) 平滑高度过渡的三角网格. (c) 光和纹理，我们在这一章节中写的例子的一个屏幕截图。

这一章的内容是实现一个Terrain（地形）类。这个类的功能很强。我们的意思是，它只是储存整个地形的顶点/索引数据，然后渲染它。因为如果游戏需要一个小的地形，那么它能够在现代图形卡支持的硬件顶点处理下工作。然而实际上，游戏需要大量的地形，你必须对细节做某种（级别）程度的捡选，因为模型需要大量的几何数据，这样大的地形对于再强大的处理方法也是无法处理的。

## 目标

* 学习怎样生成地形的高度信息，它能使山丘、河流等地带的平滑的过度，模拟自然界的地形。
* 了解怎样生成地形的顶点和三角形数据。
* 学习使用地形的纹理和光照.
* 找到控制地形上照相机位置的方法，以便模拟在地形上走动。

## 13.1 Heightmaps（高度图）

我们使用高度图去描述地形上的山丘、河流。高度图是一个数组，数组中的每个成员指定地形顶点描述中的高度信息。我们经常把高度图想像成一个矩阵，因为每个元素都一一对应于每个地形网格中的顶点。

当我们保存高度图到磁盘上时，我们通常为高度图的每个元素分配1个byte的内存，所以高度的范围是0..255，0..255的范围对于地形的高度之间保持平滑过渡是足够用的。但为了在我们的程序中匹配3D世界中的物体，可能需要的范围在0..255以外。例如，我们在3D世界中的测量单位是英尺，那么0..255的范围对于表现任何有趣的东西是不够的。因此，当我们读取数据进应用程序时，给每个高度元素分配一个整型数（或浮点型），它允许我们很好的缩放0..255范围之外的任何大小的物品。

高度图图形表示法这一是灰度图(grayscale map)。较黑的值表示地形中较低的地方，较白的值表现地形中较高的地方。



图13.2: 高度图的灰度图表示

### 13.1.1 创建高度图（Heightmap）

高度图不是用程序生成就是用图像编辑器生成，比如：Adobe Photoshop。使用图像编辑器大概是最容易的方法了。当你想生成地形时，可以交互式的可视化的创建。你可以利用图像编辑器的功能，比如：过滤器，创建一个有趣的高度图，图13.3显示了一个用Adobe Photoshop图像编辑器的工具创建的金字塔形的高度图。注意：当创建图像时我们指定一个**灰度图类型**。



图13.3 一幅用Adobe Photoshop创建的灰度图

一但你画完了你的高度图，你必须将它保存为一个8bit的RAW文件。RAW文件只图像的逐个字节。我们的应用程序可以非常容易的读这样的图像。你的软件可能告诉你保存的RAW文件是有文件头的还是没有文件头的。

**注意：**用RAW格式保存高度信息不是必须的；你可以用符合你需要的任何格式。RAW格式是我们能使用的的格式之一。我决定使用RAW格式是因为很多流行的图像编辑器支持导出这种格式，而且应用程序读取RAW文件的数据非常简单。这章中有使用8-bit RAW文件的例子。

### 13.1.2 读取RAW文件

RAW文件与一段连续的bit内存块没什么分别。我们能用很简单的方法读取这段内存块，注意：变量\_heightmap是Terrain类的一个成员，定义如下 ：

|  |
| --- |
| std::vector<int> \_heightmap;  bool Terrain::readRawFile(std::string fileName)  {  // A height for each vertex  std::vector<BYTE> in( numVertices );  std::ifstream inFile(fileName.c\_str(), std::ios\_base::binary);  if( inFile == 0 )  return false;  inFile.read(  (char\*)&in[0], // buffer  in.size());// number of bytes to read into buffer  inFile.close();  // copy BYTE vector to int vector  \_heightmap.resize( \_numVertices );  for(int i = 0; i < in.size(); i++)  \_heightmap[i] = in[i];  return true;  } |

我们COPY一个bytes向量到一个整形向量，这样做我们能够缩放 [0,255]以外的高度。这个方法唯一限制是：RAW文件必须读入至少与地形的顶点数一样多的高度信息。因此，如果你读取一个256x256 的RAW文件，你的地形也必须包含256x256个顶点。

### 13.1.3 访问与修改Heightmap

Terrain类提供以下2个方法访问和修改Heightmap的入口。

|  |
| --- |
| int Terrain::getHeightmapEntry(int row, int col)  {  return \_heightmap[row \* \_numVertsPerRow + col];  }  void Terrain::setHeightmapEntry(int row, int col, int value)  {  \_heightmap[row \* \_numVertsPerRow + col] = value;  } |

这些方法允许我们以行和列来访问入口，并且隐藏方法。当使用它去描述矩阵时，我们必须将一个线性数组编入索引。

（These methods allow us to refer to an entry by row and column and hide the way we must index a linear array when using it to describe a matrix）

## 13.2 生成地形几何数据

图13.4显示Terrain类的一些属性、词汇和我们提到的一些关键点。我们定义地形的大小，指定每行、每列顶点的数量，和单元的间隔。传递这些值到Terrain类的构造器中。另外，也传递地形所关联的设备，一个包含高度图数据的字符串文件名，一个用来缩放高度图成员的高度缩放值。

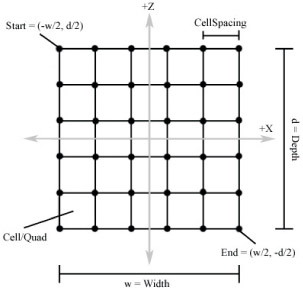


图13.4：三角形网络的属性，延着方格线上的点是地形的顶点。

|  |
| --- |
| class Terrain  {  public:  Terrain(  IDirect3DDevice9\* device,  std::string heightmapFileName,  int numVertsPerRow,  int numVertsPerCol,  int cellSpacing, // space between cells  float heightScale); // value to scale heights by  ... methods snipped  private:  ...device/vertex buffer etc snipped  int \_numVertsPerRow;  int \_numVertsPerCol;  int \_cellSpacing;  int \_numCellsPerRow;  int \_numCellsPerCol;  int \_width;  int \_depth;  int \_numVertices;  int \_numTriangles;  float \_heightScale;  }; |

Terrain类定义的全部的源代码，实在是太多了，无法在这里全部包含进来。根据传递给构造器的值，我们能够计算Terrain类的其他变量：

|  |
| --- |
| \_numCellsPerRow = \_numVertsPerRow - 1;  \_numCellsPerCol = \_numVertsPerCol - 1;  \_width = \_numCellsPerRow \* \_cellSpacing;  \_depth = \_numCellsPerCol \* \_cellSpacing;  \_numVertices = \_numVertsPerRow \* \_numVertsPerCol;  \_numTriangles = \_numCellsPerRow \* \_numCellsPerCol \* 2; |

Terrain类定义的顶点结构：

|  |
| --- |
| struct TerrainVertex  {  TerrainVertex(){}  TerrainVertex(float x, float y, float z, float u, float v)  {  \_x = x; \_y = y; \_z = z; \_u = u; \_v = v;  }  float \_x, \_y, \_z;  float \_u, \_v;  static const DWORD FVF;  }; |

注意：TerrainVertex是Terrain类内部的一个嵌套类（译者：看样子是结构啊？），之所以这么做，是因为它在Terrain类外部基本没有什么用处。

### 13.2.1 计算顶点

在图13.4中，计算三角形网格上的顶点，我们只是在开始产生顶点的地方，一行一行的生成顶点数据，直到结束为止。单元格的顶点与顶点之间有一块空白区域，这会让我们取得x、z坐标，但y坐标是什么呢？得到y坐标很容易，当读取高度图数据结构时会找到对应的入口。

注意：这个操作使用一个巨大的顶点缓存去保存所有地形上的所有顶点。这可能会引起硬件局限性的问题。例如：一个原始计数界限的最大值和3D设备设定的最大的顶点索引界限。检查MaxPrimitiveCount和D3DCAPS9结构的MaxVertexlndex成员，查看你的设备的限定值，在13.7节讨论，使用顶点缓存时存在问题和解决方法。

计算纹理坐标，看图13.5，给我们一个简单的设定，允许我们用(u, v)纹理坐标去对应地形顶点坐标。

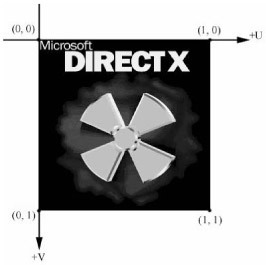
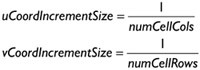


图13.5：地形顶点与纹理顶点之间一一对应。

**u = j uCoordIncrementSize**

**v = i vCoordIncrementSize**

And where:



最后，用代码生成顶点：

|  |
| --- |
| bool Terrain::computeVertices()  {  HRESULT hr = 0;  hr = \_device->CreateVertexBuffer(  \_numVertices \* sizeof(TerrainVertex),  D3DUSAGE\_WRITEONLY,  TerrainVertex::FVF,  D3DPOOL\_MANAGED,  &\_vb,  0);  if(FAILED(hr))  return false;  // 对应第一个生成的顶点坐标  int startX = -\_width / 2;  int startZ = \_depth / 2;  // 对应最后一个生成的顶点坐标  int endX = \_width / 2;  int endZ = -\_depth / 2;  // compute the increment size of the texture coordinates  // from one vertex to the next.  float uCoordIncrementSize = 1.0f / (float)\_numCellsPerRow;  float vCoordIncrementSize = 1.0f / (float)\_numCellsPerCol;  TerrainVertex\* v = 0;  \_vb->Lock(0, 0, (void\*\*)&v, 0);  int i = 0;  for(int z = startZ; z >= endZ; z -= \_cellSpacing)  {  int j = 0;  for(int x = startX; x <= endX; x += \_cellSpacing)  {  // compute the correct index into the vertex buffer  // and heightmap based on where we are in the nested  // loop.  int index = i \* \_numVertsPerRow + j;  v[index] = TerrainVertex(  (float)x,  (float)\_heightmap[index],  (float)z,  (float)j \* uCoordIncrementSize,  (float)i \* vCoordIncrementSize);  j++; // next column  }  i++; // next row  }  \_vb->Unlock();  return true;  } |

### 13.2.2 计算索引-定义三角形

计算三角形网格的索引，只需要循环访问每一个格子，从左上到右下，如图13.4，并且计算组成格子的2个三角形。

这里的技巧是：提取出计算第ij格子的2个三角形的公式。用图13.6去推导公式，找到第ij的格子：

**∆ABC = {i · numVertsPerRow + j i·numVertsPerRow + j + 1 (i + 1). numVertsPerRow + j}**

**∆CBD = {(i + 1) numVertsPerRow + j i·numVertsPerRow + j + 1 (i·l) numVertsPerRow + j + 1}**

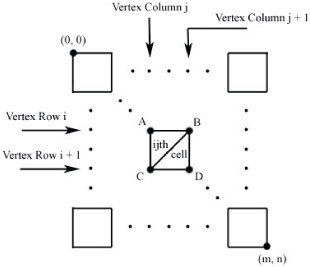


图13.6 方格的顶点

代码生成索引：

|  |
| --- |
| bool Terrain::computeIndices()  {  HRESULT hr = 0;  hr = \_device->CreateIndexBuffer(  \_numTriangles \* 3 \* sizeof(WORD), // 每个三角形有3个索引  D3DUSAGE\_WRITEONLY,  D3DFMT\_INDEX16,  D3DPOOL\_MANAGED,  &\_ib,  0);  if(FAILED(hr))  return false;  WORD\* indices = 0;  \_ib->Lock(0, 0, (void\*\*)&indices, 0);  // 将组成一个方格的2个三角形的一组6个索引的开始位置编入索引  int baseIndex = 0;  // 从头到尾计算每一个格子中的三角形  for(int i = 0; i < \_numCellsPerCol; i++) //行循环  {  for(int j = 0; j < \_numCellsPerRow; j++) //列循环  {  indices[baseIndex] = i \* \_numVertsPerRow + j;  indices[baseIndex + 1] = i \* \_numVertsPerRow +  j + 1;  indices[baseIndex + 2] = (i+1) \* \_numVertsPerRow + j;  indices[baseIndex + 3] = (i+1) \* \_numVertsPerRow + j;  indices[baseIndex + 4] = i \* \_numVertsPerRow +  j + 1;  indices[baseIndex + 5] = (i+1) \* \_numVertsPerRow +  j + 1;  // next quad  baseIndex += 6;  }  }  \_ib->Unlock();  return true;  }  ; |

## 13.3 纹理

Terrain类提供2个方法去处理地形的纹理。最简单的方法是简单地读取一个已经制作好的纹理文件并使用它，下面的方法使用Terrain类实现将一个文件读取纹理到\_tex成员中，然后指向一个IDirect3DTexture9接口的指针。关键是，在地形渲染之前先用Terrain: :draw方法设置\_tex。

到目前为止，书中的实现对于读者来说还是比较容易的。

|  |
| --- |
| bool Terrain::loadTexture(std::string fileName)  {  HRESULT hr = 0;  hr = D3DXCreateTextureFromFile(  \_device,  fileName.c\_str(),  &\_tex);  if(FAILED(hr))  return false;  return true;  } |

### 13.3.1 程序上的处理方法

一个可选择的方法是用程序计算地形的纹理，就是说，我们创建一个空纹理，根据定义的参数用代码计算每一个部分的颜色，在例子中，参数是地形的高度。

我们用Terrain::genTexture方法用程序去生成纹理，首先用D3DXCreateTexture方法创建一个空的纹理，锁定高度级别（top level，纹理图的一个成员，有多个级别），不断的循环每一个texel（图素）并给它上色，texel的颜色取决于与方格对应的高度（近似高度）。我们的想法是：地形中较低的地方是沙滩色，中间的地方像是绿色的小山丘，较高的地方颜色好像雪山。我们定义的高度是方格中左上角的近似高度。

一旦每个texel都有了颜色，我们想让每一个texel变暗或是变亮，这基于光打在格子中对应的texel上的角度，由Terrain::lightTerrain方法实现。（Once we have a color for each texel, we want to darken or brighten each texel based on the angle at which sunlight (modeled by a directional light) strikes the cell to which the texel corresponds. This is done in the Terrain::lightTerrain method）

Terrain::genTexture方法通过计算lower mipmap级别的texels来得出结论，它是通过D3DXFilterTexture函数实现。用代码生成纹理：

|  |
| --- |
| bool Terrain::genTexture(D3DXVECTOR3\* directionToLight)  {  // Method fills the top surface of a texture procedurally. Then  // lights the top surface. Finally, it fills the other mipmap  // surfaces based on the top surface data using  // D3DXFilterTexture.  HRESULT hr = 0;  // texel for each quad cell  int texWidth = \_numCellsPerRow;  int texHeight = \_numCellsPerCol;  // create an empty texture  hr = D3DXCreateTexture(  \_device,  texWidth, texHeight, // dimensions  0, // create a complete mipmap chain  0, // usage - none  D3DFMT\_X8R8G8B8, // 32-bit XRGB format  D3DPOOL\_MANAGED, // memory pool  &\_tex);  if(FAILED(hr))  return false;  D3DSURFACE DESC textureDesc;  \_tex->GetLevelDesc(0 /\*level\*/, &textureDesc);  // make sure we got the requested format because our code  // that fills the texture is hard coded to a 32-bit pixel depth.  if( textureDesc.Format != D3DFMT\_X8R8G8B8 )  return false;  D3DLOCKED\_RECT lockedRect;  \_tex->LockRect(0/\*lock top surface\*/, &lockedRect,  0 /\* lock entire tex\*/, 0/\*flags\*/);  // fill the texture  DWORD\* imageData = (DWORD\*)lockedRect.pBits;  for(int i = 0; i < texHeight; i++)  {  for(int j = 0; j < texWidth; j++)  {  D3DXCOLOR c;  // get height of upper-left vertex of quad.  float height = (float)getHeightmapEntry(i, j)/\_heightScale;  // set the color of the texel based on the height  // of the quad it corresponds to.  if( (height) < 42.5f ) c = d3d::BEACH SAND;  else if( (height) < 85.0f ) c = d3d::LIGHT YELLOW GREEN;  else if( (height) < 127.5f ) c = d3d::PUREGREEN;  else if( (height) < 170.0f ) c = d3d::DARK YELLOW GREEN;  else if( (height) < 212.5f ) c = d3d::DARKBROWN;  else c = d3d::WHITE;  // fill locked data, note we divide the pitch by four  // because the pitch is given in bytes and there are  // 4 bytes per DWORD.  imageData[i \* lockedRect.Pitch / 4 + j] = (D3DCOLOR)c;  }  }  \_tex->UnlockRect(0);  // light the terrain  if(!lightTerrain(directionToLight))  {  ::MessageBox(0, "lightTerrain() - FAILED", 0, 0);  return false;  }  // fill mipmaps  hr = D3DXFilterTexture(  \_tex,// texture to fill mipmap levels  0, // default palette  0, // use top level as source for lower levels  D3DX\_DEFAULT); // default filter  if (FAILED (hr))  {  ::MessageBox(0, "D3DXFilterTexture() - FAILED", 0, 0);  return false;  }  return true;  } |

注意：颜色常量BEACH\_SAND等定义在d3dUtility.h.文件中。

## 13.4 光照

Terrain::genTexture方法会调用Terrain::lightTerrain，顾名思义，光照使地形更接近于现实。当我们已经计算完地形纹理以后，我们只需要计算阴影系数（shade factor），使一个定义了光源的地形区域变亮或变暗。在这一节中，我们检验这样一个技巧，你会惊讶于为什么我们照亮地图却没有让Direct3D来做。我们自己来计算有三个好处：

* 内存中不必保存顶点法线。
* 因为纹理是静态的，所以不能随意的移动光源。虽然我们可以重新计算光源，但因此采用Direct3D实时的照亮地形是很耗时的。
* 我们获得了一些数学上的经验，熟悉了一些基本的光照概念，并且是用Direct3D函数实践的。

### 13.4.1概览(OVERVIEW)

光照是计算地形阴影（shade）的一个最基本的技巧之一，一般认为的光是漫射光（*diffuse lighting*），我们定义一个平行光源，指定光的方向，延着光线的相反方向是散发平行光的光源。因此，如果我们想让光线从空中笔直落下，那么*lightRaysDirection =* (0, -1, 0)，按相反的方向：*directionToLight* = (0, 1, 0)。**注意：创建光照向量要使用单位向量。**

**注意：**虽然指定方向的光是从光源发射出来的，这么说更直接一点，指定方向的光在计算上要比漫谢光更合得来。

对于地形中的每个方格，我们计算光的向量与方格的面法线之间的角度。

在图13.7中我们看到，当角度变得比较大时，方格的面离光源越来越远，接收的光越少。反过来说，角度变小，方格的面则离光源越来越近，相应的会接收更多的光。注意：一旦光向量与法线角度大于90度，表面就接收不到光。

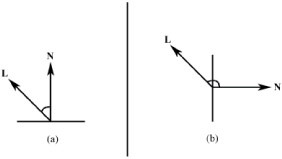


图13.7 光向量与平面法线的关系，我们能够创建一个阴影（shading） 标量，用0..1之间的范围来表示表面能接收到光的多少。使用阴影标量，角度大则标量接近于0。当颜色与一个阴影标量接近0的值相乘时，得到的结果是：颜色变暗。相反，乘以一个阴影标量的值接近1的值时，颜色则接近于原始亮度。

### 13.4.2 计算方格的阴影（Shade）

光源的方向是一个单位向量，为了计算光源方向与面法线间的夹角，首先需要找到面法线，这是叉积的一小部分应用，但首先必须在方格里找到二个共面的非0并且不平行的向量。看图 13.8有两个这样的向量：

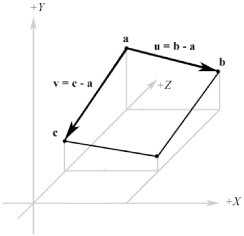


图13.8: 计算在同一方格中的共面的二个向量

**u = (cellSpacing, by - ay, 0)**

**v = (0, cy, -ay, -cellSpacing)**

关于u和v，方格的法线N = u × v，当然要把N标准化：

无标题

找到光线与法线的夹角，回忆一下点积，是二个3维空间中的单位向量组成的夹角的余弦。

无标题

它的标量是在-1..1的范围，因为-1..0的sin值符合夹角角度且大于90度，在图13.7中接受不到光照，如果它在-1..0之间那么夹角是0度。

|  |
| --- |
| float cosine = D3DXVec3Dot(&n, directionToLight);  if(cosine < 0.0f)  cosine = 0.0f; |

现在s的夹角大于90度，s的阴影标量将在0..1之间。因为光线与法线的角度从0增加到90度时，s的值将从1到降到0。这是我们想要的结果，具体讲解请看13.4.1节。

给指定的格子计算阴影系数用Terrain::computeShade方法，它需要参数：行和列来确定方格，还有平行方向光的光源。

|  |
| --- |
| float Terrain::computeShade(int cellRow, int cellCol,  D3DXVECTOR3\* directionToLight)  {  // 取得方格中三个顶点的高度（从高度图中）  float heightA = getHeightmapEntry(cellRow, cellCol);  float heightB = getHeightmapEntry(cellRow, cellCol+1);  float heightC = getHeightmapEntry(cellRow+1, cellCol);  // 创建方格中的二个顶点  D3DXVECTOR3 u( cellSpacing, heightB - heightA, 0.0f);  D3DXVECTOR3 v(0.0f, heightC - heightA, - cellSpacing);  //用方格中的二个向量的叉积找到面法线  D3DXVECTOR3 n;  D3DXVec3Cross(&n, &u, &v);  D3DXVec3Normalize(&n, &n);  float cosine = D3DXVec3Dot(&n, directionToLight);  if(cosine < 0.0f)  cosine = 0.0f;  return cosine;  } |

### 13.4.3 地形阴影（Shading）

一旦知道了如何给指定的方格加阴影，我们就能给地形上所有的方格加阴影。只要遍例每一个方格，计算方格的阴影值，并测量方格对应的texel颜色。光照少则方格会变暗。下面一段代码展示了Terrain::lightTerrain方法的重要部分：

|  |
| --- |
| DWORD\* imageData = (DWORD\*)lockedRect.pBits;  for(int i = 0; i < textureDesc.Height; i++)  {  for(int j = 0; j < textureDesc.Width; j++)  {  int index = i \* lockedRect.Pitch / 4 + j;  // get current color of cell  D3DXCOLOR c( imageData[index] );  // shade current cell  c \*= computeShade(i, j, lightDirection);;  // save shaded color  imageData[index] = (D3DCOLOR)c;  }  } |

## 13.5 在地形上“行走”

构造了一个地形以后，我们想要有移动照相机的能力，以便模拟在地形上行走的效果。我们需要调整照相机的高度，这依赖于地形部分的知识，好的，我们继续往下看。我们首先需要找到照相机所在的方格的位置，并给出x轴和z轴坐标，Terrain::getHeight函数能做到这些，它能提供x轴、y轴坐标参数，返回照相机需要被设置在地形上的高度值，现在看实现部分。

|  |
| --- |
| float Terrain::getHeight(float x, float z)  {  // Translate on xz-plane by the transformation that takes  // the terrain START point to the origin.  x = ((float) width / 2.0f) + x;  z = ((float) depth / 2.0f) - z;  // Scale down by the transformation that makes the  // cellspacing equal to one. This is given by  // 1 / cellspacing since cellspacing \* 1 / cellspacing = 1.  x /= (float) cellSpacing;  z /= (float)\_cellSpacing; |

我们首先转换地形的起始点为原点，然后，我们按反方向去测量空间变量（we scale by the inverse of the cell spacing variable），设置单元空间间隔为1。我们切换到一个新的参考框架，z轴正方向是向下的。当然，没有代码转换参考框架，但现在我们知道+z是向下的。图13.9显示了这些步骤：

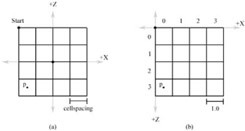


图13.9：地形网格在转换前的第一个点，转换后为原点。单元格的空间为1，转换z轴。

我们看到我们转换的坐标系统与矩阵的行和列相对应，也就是说左上为原点，列数的增加向右，行数的增加向下。因此，在图13.9中我们知道了单元格的空间是1，通过以下的方法我们马上就能得到单元格行和列：

**float col = ::floorf(x);**

**float row = ::floorf(z);**

换句话说，在x轴部分列是整数，z轴部分行也是整数。回忆floor(t)函数，。

现在我们将取得方格的四个顶点的高度。

// A B

// \*--\*

// | / |

// \*--\*

// C D

|  |
| --- |
| float A = getHeightmapEntry(row, col);  float B = getHeightmapEntry(row, col+1);  float C = getHeightmapEntry(row+1, col);  float D = getHeightmapEntry(row+1, col+1); |

现在我们知道了方格的四个顶点的高度，我们需要找到照相机所在的位置的方格的高度，因为一个方格可能同时向几个方向倾斜，这可能会稍微难一点，见图 13.10:

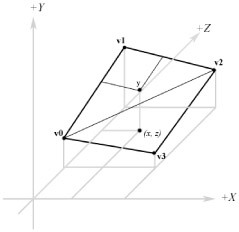


图13.10: 照相机所在的位置的方格的高度

为了找到高度，我们需要知道我们在方格中的哪个三角形里。方格是由二个三角形渲染成的，找到我们所在的三角形，我们要取得我们所在的方格并且转换它，它的左上点是原点。

自从用行和列来描述我们所在的方格左上顶点的位置以来，我们必须转换列x轴与行z轴，转换x、z坐标：

**float dx = x - col;**

**float dz = z - row;**

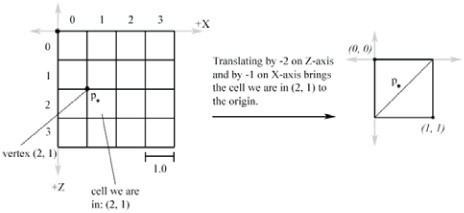


图13.11: 我们所在的方格在转换前与转换后，左上顶点变成了原点。

.现在解释当我们在方格中的上三角形时如何找到高度，这和在下三角形是相似的。马上你会看到这两种情况的代码。在上三角形时，构造2个向量：**u** = (*cellSpacing, B* *-A,* 0) and **v** = (0, *C* - *A, - cellSpacing*)，三角形的边上并且在矢量**q** = (*qx, A*, *qz*)终点点开始的地方，如图13.12(a)。

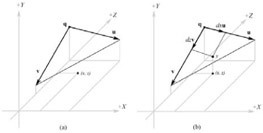


图13.12 (a) 计算三角形的邻边和对边这两个向量。 (b)使用线性差值创建高度

注意：我们只关心改变的高度值，我们只修改y值，忽视其他部分，因此，Height=sum *A + dxuy + dzvy*

以下是Terrian::getHeight函数的实现代码：

（Note that since we are only concerned about the interpolated height value, we can just interpolate the y-components and ignore the other components. Thus, the height is obtained by the sum *A + dxuy + dzvy.*）

|  |
| --- |
| if(dz < 1.0f - dx) // upper triangle ABC  {  float uy = B - A; // A->B  float vy = C - A; // A->C  height = A + d3d::Lerp(0.0f, uy, dx) +  d3d::Lerp(0.0f, vy, dz) ;  }  else // lower triangle DCB  {  float uy = C - D; // D->C  float vy = B - D; // D->B  height = D + d3d::Lerp(0.0f, uy, 1.0f - dx) +  d3d::Lerp(0.0f, vy, 1.0f - dz);  } return height;  } |

Lerp函数是一个沿着一维直线的基本线性插值算法，实现如下：

|  |
| --- |
| float d3d::Lerp(float a, float b, float t)  {  return a - (a\*t) + (b\*t);  } |

## 13.6 例子程序: Terrain

这章的例子是用一个包含高度信息的RAW文件创建一个地形，纹理和光源。用方向键在地形上行走。注意，下列函数中不相关的代码被省略了，被省略的代码用(...)表示，依赖你的硬件，这个例子可能运行得很慢，请偿试运行一个小地形。

首先，增加全局变量：地形、照相机、每秒帧数。

|  |
| --- |
| Terrain\* TheTerrain = 0;  Camera TheCamera(Camera::LANDOBJECT);  FPSCounter\* FPS = 0; |

下面是框架函数：

|  |
| --- |
| bool Setup()  {  D3DXVECTOR3 lightDirection(0.0f, -1.0f, 0.0f);  TheTerrain = new Terrain(Device, "coastMountain256.raw",  256, 256, 10, 1.0f);  TheTerrain->genTexture();  TheTerrain->lightTerrain(&directionToLight);  ...  return true;  }  void Cleanup()  {  d3d::Delete<Terrain\*>(TheTerrain);  d3d::Delete<FPSCounter\*>(FPS);  }  bool Display(float timeDelta)  {  if( Device )  {  // Update the scene:  ...[snipped input checking]  // Walking on the terrain: Adjust camera's height so we  // are standing 5 units above the cell point we are  // standing on.  D3DXVECTOR3 pos;  TheCamera.getPosition(&pos);  float height = TheTerrain->getHeight( pos.x, pos.z );  pos.y = height + 5.0f;  TheCamera.setPosition(&pos);  D3DXMATRIX V;  TheCamera.getViewMatrix(&V);  Device->SetTransform(D3DTS VIEW, &V);  // Draw the scene:  Device->Clear(0, 0, D3DCLEAR TARGET | D3DCLEAR ZBUFFER,  0xff000000, 1.0f, 0);  Device->BeginScene();  D3DXMATRIX I;  D3DXMatrixIdentity(&I);  if( TheTerrain )  TheTerrain->draw(&I, false);  if( FPS )  FPS->render(0xffffffff, timeDelta);  Device->EndScene();  Device->Present(0, 0, 0, 0);  }  return true;  } |

## 13.7 一些改进

Terrain读取顶点数据到一个很大的缓存，在多重的顶点缓存中划分地形结构，在速度和可测量性方面都十分有利。为我们提出一个问题：顶点缓存最大支持多大？回答是，这依赖于你的硬件。所以你必须先检测。

将地图划分为许多小的顶点缓存是重要的练习，然后将类似矩阵的数据结构编入索引，并且管理数据，这不需要引入新的概念。我们不必详细讨论它。简单的说，你基本上站在地形中一个我们叫做“blocks”的矩阵上,每个block是地形的一个矩形区域。另外，每个block区域（在它自己的顶点索引缓存中）的下方包含地形中的几何信息，为了画它在地形中的位置。

另外，你可以读取地形到一个很大的ID3DXMesh接口。使用D3D函数D3DXSplitMesh划分地形为许多小的Mesh, 以下是D3DXSplitMesh函数原型：

|  |
| --- |
| void D3DXSplitMesh(  const LPD3DXMESH pMeshIn,  const DWORD \*pAdjacencyIn,  const DWORD MaxSize,  const DWORD Options,  DWORD \*pMeshesOut,  LPD3DXBUFFER \*ppMeshArrayOut,  LPD3DXBUFFER \*ppAdjacencyArrayOut,  LPD3DXBUFFER \*ppFaceRemapArrayOut,  LPD3DXBUFFER \*ppVertRemapArrayOut  ); |

这个函数将一个源Mesh划分多个小的Mesh,，pMeshIn参数是一个指针，指向想划分的Mesh，pAdjacencyIn指向一个邻接数组，MaxSize参数指定作为结果返回的最大顶点数，为返回的Meshe使用指定的创建标记，pMeshesOut参数返回ppMeshArrayOut数组中的Mesh数量，最后3个参数是可选的（可以指定为null），返回邻接信息的数组。

## 13.8 摘要

* 我们能用三角形网格和不同的高度值来模拟地形，创建山丘、河流。
* Heightmap数据包含地形顶点的高度值。
* 我们能通过程序使用磁盘上的图像文件生成地形上的纹理。
* 我们能照亮地形，通过计算阴影系数来使每个格子变亮或变暗，阴影系数是由光照在格子上的角度决定的。
* 使照相机在地形上走动，我们需要找到我们站立的三角形。我们计算三角形上的邻边和对边这两个向量，高度是通过…（线性插值在这些向量中每个使用x、z对应的单位向量，以左高顶点为原点为参数。）找到的。

（The height is then found by linearly interpolating on each of these vectors using the x- and z-coordinates in a normalized cell with an upper-left vertex at the origin as parameters.）

# 第十四章 粒子系统 (Particle Systems)

许多自然现象是由很多小的小颗粒组成的，它们有相似的行为。（例如，雪花落下，闪烁的火焰，冲出枪管的“子弹”），粒子系统用来模拟这种现象。

## 目标：

* 学习我们给定的粒子属性，如何描述D3D中的粒子。
* 设计一个灵活的粒子基系统的基类，包括一般的粒子系统都有的属性和方法。
* 模拟3个具体的粒子系统，雪、爆炸、粒子枪。

## 14.1 粒子和点精灵（Point Sprite）

粒子是一个很小的对象，它通常用来模拟数学中的一个点。点元是用来显示粒子的很好的方案。可是点元被光栅化成一个简单的像素。这没给我们多少灵活性，因为我们想有各种大小不同的粒子，并且把整个纹理平滑映射到这些粒子上。在Direct3D 8.0,以前，因为点元方法的局限性而完全不使用他们。代替的方法是，程序员将使用公告板去显示粒子，一个板是一个方格，世界矩阵用它来确定方向，使它总是朝向照相机。

Direct3D 8.0引入一个特殊的点元叫**点精灵**，多数时候被应用在粒子系统中。与一般的点元不同的是，点精灵有纹理映射并能改变大小。与公告板不同的是，能用一个简单的点描述一个点精灵，节省内存和处理时间，因为我们只是必须保存和处理一个点，而公告板则是四个。

### 14.1.1 结构的格式

我们使用下面的顶点结构来描述粒子的位置和颜色：

|  |
| --- |
| struct Particle  {  D3DXVECTOR3 \_position;  D3DCOLOR \_color;  static const DWORD FVF;  };  const DWORD Particle::FVF = D3DFVF\_XYZ | D3DFVF\_DIFFUSE; |

这个结构只保存粒子的位置和颜色，这取决于你程序的需要，你能够用同样的结构去保存一套纹理坐标，我们在下一节讨论给点精灵赋予纹理。

增加一个浮点变量给Particle结构去指定粒子的大小是可能的。我们必须增加一个D3DFVF\_PSIZE标记给我们的灵活的顶点格式，以反映这个变化。每个粒子维护自己的大小很有用，因为它允许我们以具体情况指定并改变粒子的大小。可是，大多数的图形卡不支持控制粒子的大小，因此我们不使用它。（检查D3DFVFCAPS\_PSIZE在D3 DCAPS9结构的FVFCaps成员）代替的方法是：用渲染状态（render states）去控制粒子的大小，就像你很快看到的，有尺寸成员的顶点结构的例子：

|  |
| --- |
| strict Particle  {  D3DXVECTOR3 \_position;  D3DCOLOR \_color;  float \_size;  static const DWORD FVF;  };  const DWORD Particle::FVF = D3DFVF XYZ | D3DFVF DIFFUSE |  D3DFVF\_PSIZE; |

注意：通过vertex shader，能够获取每个粒子的大小，即使你的硬件不支持D3DFVFCAPS\_PSIZE。Vertex shaders的内容在本书的第IV部分。

### 14.1.2点精灵（Point Sprite）渲染状态

点精灵的行为大部分由渲染状态（render states）来控制，现在让我们来看一下这些渲染状态：

**D3DRS\_POINTSPRITEENABLE**—A Boolean value. The default value is false.

True表示将当前的纹理全部映射到点精灵上。

False 表示用指定的纹理坐标映射到点精灵的点（图素）上。

\_device->SetRenderState(D3DRS\_POINTSPRITEENABLE, true);

**D3DRS\_POINTSCALEENABLE**—A Boolean value. The default value is false.

True表示用视图空间单位来解释点的大小。视图空间单位的3D空间点在照相机中，点精灵将会自动缩放，这取决到它有多远, 像其他对象一样，离照相机近的粒子比离照相机远的粒子要小。

False 表示点的大小将用屏幕空间单位来解释。屏幕空间单位是屏幕上的像素单位。. 因此如果你指定false, 例如, 设置点精灵的尺寸为3, 则点精灵在屏幕区域中的尺寸为3×3像素。.

\_device->SetRenderState(D3DRS\_POINTSCALEENABLE, true);

**D3DRS\_POINTSIZE**—表示点精灵的尺寸. 这个值可以任意指定视图空间或屏幕空间的点精灵的尺寸, 取决于D3DRS\_POINTSCALEENABLE 状态如何设置. 下面的代码段设置点的尺寸为2.5个单位。:

\_device->SetRenderState( D3DRS\_POINTSIZE, d3d::FtoDw(2.5f) );

**d3d::FtoDw** 是我们新加进 d3dUtility.h/cpp 文件中的一个函数，它将float型转换为 DWORD型。 我们必须这么做是因为所有的IDirect3DDevice9::SetRenderState 都要一个 DWORD 型的值而不是float型。

DWORD d3d::FtoDw(float f)

{

return \*((DWORD\*)&f);

}

**D3DRS\_POINTSIZE\_MIN**—表示点精灵的最小尺寸。例子，将设置最小值为0.2：

\_device->SetRenderState(D3DRS\_POINTSIZE\_MIN, d3d::FtoDw(0.2f));

**D3DRS\_POINTSIZE\_MAX**—表示点精灵的最大尺寸。例子，将设置最大值为5.0:

\_device->SetRenderState(D3DRS\_POINTSIZE\_MAX, d3d::FtoDw(5.0f));

**D3DRS\_POINTSCALE\_A**, **D3DRS\_POINTSCALE\_B**, **D3DRS\_POINTSCALE\_C**—这3个常量表示如何根据距离控制点精灵的尺寸—这个距离是点精灵到照相机的距离。

D3D用以下的公式去计算点精灵的最终尺寸，这取决于距离和这3个常量。

无标题

其中：

*FinalSize：*距离计算后，点精灵的最后尺寸。

*ViewportHeight：*视口的高度。

*Size：*分别为D3DRS\_POINTSCALE\_A, D3DRS\_POINTSCALE\_B, and D3DRS\_POINTSCALE\_C值。

*D：*在视图空间中点精灵与照相机的距离。因为照相机被放置在视图空间中的原点，这个值是：无标题，也是点精灵所在的位置。

下面代码设置点精灵的距离常量，因此远处的点精灵将变小。

|  |
| --- |
| \_device->SetRenderState(D3DRS\_POINTSCALE\_A, d3d::FtoDw(0.0f));  \_device->SetRenderState(D3DRS\_POINTSCALE\_B, d3d::FtoDw(0.0f));  \_device->SetRenderState(D3DRS\_POINTSCALE\_C, d3d::FtoDw(1.0f)); |

### 14.1.3 粒子和他们的属性

一个粒子系统是由除了位置、颜色以外的更多的属性组成，例如，一个粒子有某些速度。然而，这些额外的属性对于渲染粒子来说不是必须的。因此，我们在单独的结构中保存渲染粒子所必须的数据和属性。当我们创建、显示或更新粒子时，我们使用属性来工作。当我们准备渲染时，我们从Particle（粒子）结构中COPY位置和颜色。

对于我们模拟的具体粒子系统，粒子的属性也是不同的。因此我们能够归纳一些通用的属性，一面的结构例子中包含一些通用的属性，大多数系统用不上这么多，一些系统需要的属性这里可能还没有。

|  |
| --- |
| struct Attribute  {  D3DXVECTOR3 \_position;  D3DXVECTOR3 \_velocity;  D3DXVECTOR3 \_acceleration;  float \_lifeTime;  float \_age;  D3DXCOLOR \_color;  D3DXCOLOR \_colorFade;  bool \_isAlive;  }; |

\_position—粒子在世界空间中的位置

\_velocity—粒子的速度，每秒多少个单位。

\_acceleration—粒子的加速度, 每秒多少个单位。

\_lifeTime—粒子的生命周期. 例如,当一个时间段后，我们可以杀死一个激光柱的粒子.

\_age—粒子的当前年龄。

\_color—粒子的颜色。

\_colorFade—粒子随时间的变化而褪去的颜色。

\_isAlive—True 表示粒子活着;false 表示粒子死了。

## 14.2 粒子系统的组成

粒子系统是粒子的集合，用来保存和显示这些粒子。粒子系统维护所有粒子的全部属性，影响系统中的所有粒子：粒子的尺寸，起始的位置及应用在粒子上的纹理等。粒子系统的方法负责更新、显示、杀死和创建粒子。

虽然不同的具体（与抽象是相对的）粒子系统有不同的行为，我们归纳并找到一些所有的粒子系统共有的基本属性，我们把这些公共的属性放到一个抽象的Psystem基类，它是我们所有的具体粒子系统的父类，现在让我们看一下Psystem类：

|  |
| --- |
| class PSystem  {  public:  PSystem();  virtual ~PSystem();  virtual bool init(IDirect3DDevice9\* device, char\* texFileName);  virtual void reset();  virtual void resetParticle(Attribute\* attribute) = 0;  virtual void addParticle();  virtual void update(float timeDelta) = 0;  virtual void preRender();  virtual void render();  virtual void postRender();  bool isEmpty();  bool isDead();  protected:  virtual void removeDeadParticles();  protected:  IDirect3DDevice9\* \_device;  D3DXVECTOR3 \_origin;  d3d::BoundingBox \_boundingBox;  float \_emitRate;  float \_size;  IDirect3DTexture9\* \_tex;  IDirect3DVertexBuffer9\* \_vb;  std::list<Attribute> \_particles;  int \_maxParticles;  DWORD \_vbSize;  DWORD \_vbOffset;  DWORD \_vbBatchSize;  }; |

一些数据成员：

* **\_origin**—粒子系统的原点， 这是粒子系统产生时的位置。
* **\_boundingBox**—创建粒子系统使用的边界盒，用于限制粒子的活动范围。例如，假如我们让雪系统只落在一个围绕高山的峰顶的体积内； 我们会定义一个包括这个体积的边界盒, 出界的粒子将会被杀死。
* **\_emitRate**—新增加到系统中的粒子的速度。 通常的标准是每秒。
* **\_size**—系统中所有粒子的尺寸。
* **\_particles**—系统中粒子属性的一个列表。 我们用这个列表创建，释放及更新粒子。 当我们准备画粒子时, 我们COPY列表节点的一部分到顶点缓存并画粒子。 当我们COPY另外一批时绘制这批粒子，然后重复这一过程直到绘制完所有粒子。 这有点太简单了，我们将在section 14.2.1节详细的解释绘制的过程。
* **\_maxParticles**—在给定的时间内，系统中允许的粒子最大数。例如, 如果创建粒子的速度比释放快的话, 随着时间的增长粒子的数量将会是巨大的，这个成员将避免出现这样的问题。
* **\_vbSize**—在给定的时间内顶点缓存中能够保存的粒子的数量，这个值与实际的粒子系统中的粒子数量无关。

**注意：**member \_vbOffset和\_vbBatchSize数据成员在渲染粒子系统时使用，我们在稍后的section 14.2.1节讨论。

方法：

* **PSystem/ ~PSystem**—用来初始化默认值的构造器/用来释放设备接口的析构器 (vertex buffer, texture)。
* **init**—这个方法做与设备无关的初始化工作, 比如创建用来保存点精灵的顶点缓存或创建纹理。 顶点缓存的创建包括一些标记，现在我们都已经讨论过了，但还没有用:

|  |
| --- |
| hr = device->CreateVertexBuffer(  \_vbSize \* sizeof(Particle),  D3DUSAGE DYNAMIC | D3DUSAGE POINTS | D3DUSAGE WRITEONLY,  Particle::FVF,  D3DPOOL\_DEFAULT,  &\_vb,  0) ; |

* + 注意： 我们使用动态的顶点缓存（D3DUSAGE DYNAMIC）。 因为我们需要在每帧中更新我们的粒子,意思是我们将会去存取顶点缓存的内存，回想一下，访问一个静态的顶点缓存慢得不可接受， 所以我们使用动态的顶点缓存。
  + 查看我们用过的 **D3DUSAGE\_POINTS** 标记,它说明顶点缓存将保存点精灵。
  + 顶点缓存的尺寸是由**\_vbSize**预先确定的，而且与系统中粒子的数量无关。 也就是说, **\_vbSize** 将小于等于系统中粒子的数量。 这是因为渲染粒子系统是一批一批的，不是一次渲染全部。 我们将在section 14.2.1节中解释渲染过程。
  + 我们使用默认的内存池(pool)代替通常使用的托管内存池，因为动态顶点缓存不能用在托管内存池中。
* **reset**—这个方法重新设置系统中每个粒子的属性:

|  |
| --- |
| void PSystem::reset()  {  std::list<Attribute>::iterator i;  for(i = \_particles.begin(); i != \_particles.end(); i++)  {  resetParticle( &(\*i) );  }  } |

* **resetParticle**—这个方法重新设置粒子的属性。如何重设粒子的属性，这依赖于具体粒子系统的特性。因此我们定义这个方法为虚拟的，等待子类去实现。
* **addParticle**—这个方法用来在系统中增加一个粒子。在增加它到粒子列表之前，使用resetParticle 方法先初始化粒子:

|  |
| --- |
| void PSystem::addParticle()  {  Attribute attribute;  resetParticle(&attribute);  \_particles.push\_back(attribute);  }  void PSystem::addParticle() |

* **update—**这个方法更新系统中所有的粒子。因为这个的方法的执行取决于具体粒子系统的特性, 因此我们定义这个方法为抽象的，等待子类去实现。
* **render**—这个方法用来显示系统中所有的粒子。 执行起来很复杂，我们将在14.2.1 节讨论。
* **preRender**—用它来初始化渲染状态， 在渲染前设置。 因为系统与系统之间是不同的,所以我们定义它为虚拟的。 默认将执行下列代码:

|  |
| --- |
| void PSystem::preRender()  {  \_device->SetRenderState(D3DRS\_LIGHTING, false);  \_device->SetRenderState(D3DRS\_POINTSPRITEENABLE, true);  \_device->SetRenderState(D3DRS\_POINTSCALEENABLE, true);  \_device->SetRenderState(D3DRS\_POINTSIZE, d3d::FtoDw( size));  \_device->SetRenderState(D3DRS\_POINTSIZE MIN, d3d::FtoDw(0.0f));  // control the size of the particle relative to distance  \_device->SetRenderState(D3DRS\_POINTSCALE A, d3d::FtoDw(0.0f));  \_device->SetRenderState(D3DRS\_POINTSCALE B, d3d::FtoDw(0.0f));  \_device->SetRenderState(D3DRS\_POINTSCALE C, d3d::FtoDw(1.0f));  // use alpha from texture  \_device->SetTextureStageState(0, D3DTSS\_ALPHAARG1, D3DTA\_  TEXTURE);  \_device->SetTextureStageState(0, D3DTSS\_ALPHAOP, D3DTOP\_  SELECTARG1);  \_device->SetRenderState(D3DRS\_ALPHABLENDENABLE, true);  \_device->SetRenderState(D3DRS\_SRCBLEND, D3DBLEND\_SRCALPHA);  \_device->SetRenderState(D3DRS\_DESTBLEND, D3DBLEND\_INVSRCALPHA);  } |

* 注意：我们使用alpha混合渲染，以便设置纹理的alpha通道，来设置纹理像素的透明度。 用它产生多种效果；一种特殊的情况是：获得象纹理那样的非矩形的粒子。例如， 获得一个圆形“雪球形”的粒子, 我们使用一个简单的带有alpha通道的纹理， 它看上去是背景为黑色的带有白色圆形的样子。因此，显示出来时只是一个白圆，这比白色的矩形纹理要好。
* **postRender**—用它去保存所有渲染状态，它是一个特殊的粒子系统可能有的设置。因为系统与系统间是不同的,所以我们定义它为虚拟的。默认将执行下列代码:

|  |
| --- |
| void PSystem::postRender()  {  \_device->SetRenderState(D3DRS\_LIGHTING, true);  \_device->SetRenderState(D3DRS\_POINTSPRITEENABLE, false);  \_device->SetRenderState(D3DRS\_POINTSCALEENABLE, false);  \_device->SetRenderState(D3DRS\_ALPHABLENDENABLE, false);  } |

* **isEmpty**—如果为True 则在当前的系统中没有粒子， 否则为false.
* **isDead**—如果为True 则系统中的所有粒子都是死的，否则为false。 注意： 系统中所有粒子状态为idDead时并不意味着isEmpty. 空意思着系统中没有粒子。 Dead的意思是系统中有粒子，但都是死的。.
* **removeDeadParticles**—搜索属\_particle性表，从表中杀死并删除粒子。

### 14.2.1 绘制粒子系统

因为粒子系统是动态的，在每一个帧中我们需要更新系统中的粒子，对于渲染粒子系统的一种直观但效率低下的方法如下:

* 创建一个足够大的顶点缓存保存最大数量的粒子。

每一帧里执行：

1. 更新所有粒子。
2. COPY所有活着的粒子到顶点缓存。
3. 绘制顶点缓存。

这个方法正确，不过不是最有效率的。第一，顶点缓冲必须足够大以保存系统中所有粒子。但是非常重要的是，当我们从列表拷贝所有粒子到顶点缓冲（步骤B）时，显卡却什么也不做。举个例子，假设我们系统有10,000个粒子，首先我们需要一个能容纳10,000个粒子的顶点缓冲，这是一个很大的内存。另外显卡将停着什么也不做直到列表中的10,000个粒子拷到顶点缓冲，并且我们调用DrawPrimitive。这个特定情况是CPU与显卡不同时工作的一个很好的例子。

更好的办法（SDK中点精灵例程中用到的方法）就象这样：

提示：这是一个简单的描述，但它说明了这一思想。它假定我们总是有500个粒子以填充一个缓存片段，但是这是不可能发生的，因为我们经常杀死并创建粒子，所以从一帧到另一帧粒子数量是变化的。举个例子，假设我们只剩下200个粒子要在当前帧拷贝并渲染。因为200个粒子不能填充整个缓存片段，我们用代码处理这个特定情形。这个特定情形只有在最后的缓存片段中才会出现，因为如果不是最后的片断，就意味着必然有500个粒子将被移到下一缓存片段。

创建一个合适尺寸的顶点缓存（能够保存2000个粒子），然后我们划分顶点缓存为几个小的块，就像这个例子，我们设置每个缓存片断的尺寸为500个粒子。

无标题

* 然后创建一个全局变量 *i* = 0 ，用来记录片段。

每一帧里执行:

* 1. 更新所有粒子。
  2. 直到所有粒子渲染完毕。:
     1. 如果顶点缓存没有满：
        1. 用D3DLOCK\_NOOVERWRITE标记锁定缓存片段i
        2. COPY 500个粒子到片段i
     2. 如果顶点缓存满了：
        1. 从起始的地方开始顶点缓冲: i=0
        2. 用D3DLOCK\_NOOVERWRITE标记锁定缓存段i
        3. COPY 500个粒子到片段i
     3. 渲染片段*i.*
     4. 下一片段： *i*+ +

**备注：**顶点缓存是动态的， 因此我们能利用动态锁定标记D3DLOCK\_NOOVERWRITE 和 D3DLOCK\_DISCARD。这两个标记允许我们锁定顶点缓存的某一部分。当顶点缓存中的其他部分被渲染时，它是不能渲染的。例如，假如我们正在使用D3DLOCK\_NOOVERWRITE标记渲染片段0时， 当渲染片段0的时候我们能锁定并填充片段1。这样可以防止渲染的延迟。

这个方法更有效率。首先，我们减少顶点缓存的尺寸；然后， CPU与显卡在协调的工作。也就是说，当我们绘制一小批粒子时(graphics card work)，同时拷贝另一小批粒子到顶点缓存 (CPU work)。这个动作是连续执行的，直到所有的粒子都被渲染完毕，就像你了解的一样， 显卡在全部顶点缓存被填充的时候是不用处于空闲状态的。

我们现在将注意力转向这一个渲染方案的实现，为了方便使用这个粒子系统的渲染方案, 我们使用 PSystem 类中的下列数据成员:

* **\_vbSize**—在给定时间内我们的顶点缓存能够保存的粒子数量。这个值与实际的粒子系统中的粒子数无关。
* **\_vbOffset**—这个变量是在顶点缓存中的偏移，在顶点缓存里我们将用它开始COPY下一批粒子，例如，如果第一批在缓存中是0到499，偏移到第二批COPY的开始处将是500。
* **\_vbBatchSize**—定义一批缓存中的粒子数量。

我们现在介绍渲染方法的代码：

|  |
| --- |
| void PSystem::render()  {  if( !\_particles.empty() )  {  // set render states  preRender();  \_device->SetTexture(0, \_tex);  \_device->SetFVF(Particle::FVF);  \_device->SetStreamSource(0, \_vb, 0, sizeof(Particle));  // start at beginning if we're at the end of the vb  if(\_vbOffset >= \_vbSize)  \_vbOffset = 0;  Particle\*v =0;  \_vb->Lock(  \_vbOffset \* sizeof( Particle ),  \_vbBatchSize \* sizeof( Particle ),  (void\*\*)&v,  \_vbOffset ? D3DLOCK\_NOOVERWRITE : D3DLOCK\_DISCARD);  DWORD numParticlesInBatch = 0;  //  // Until all particles have been rendered.  //  std::list<Attribute>::iterator i;  for(i = \_particles.begin(); i != \_particles.end(); i++)  {  if( i->\_isAlive )  {  //  // Copy a batch of the living particles to the  // next vertex buffer segment  //  v->\_position = i->\_position;  v->\_color = (D3DCOLOR)i->\_color;  v++; // next element;  numParticlesInBatch++; //increase batch counter  // is this batch full?  if(numParticlesInBatch == \_vbBatchSize)  {  //  // Draw the last batch of particles that was  // copied to the vertex buffer.  //  \_vb->Unlock();  \_device->DrawPrimitive(  D3DPT\_POINTLIST,  \_vbOffset,  \_vbBatchSize);  //  // While that batch is drawing, start filling the  // next batch with particles.  //  // move the offset to the start of the next batch  \_vbOffset += \_vbBatchSize;  // don't offset into memory thats outside the vb's  // range. If we're at the end, start at the beginning.  if(\_vbOffset >= \_vbSize)  \_vbOffset = 0;  \_vb->Lock(  \_vbOffset \* sizeof( Particle ),  \_vbBatchSize \* sizeof( Particle ),  (void\*\*)&v,  \_vbOffset ? D3DLOCK\_NOOVERWRITE :  D3DLOCK\_DISCARD);  numParticlesInBatch = 0; // reset for new batch  }//end if  }//end if  }//end for  \_vb->Unlock();  // it's possible that the LAST batch being filled never  // got rendered because the condition  // (numParticlesInBatch == \_vbBatchSize) would not have  // been satisfied. We draw the last partially filled batch now.  if( numParticlesInBatch )  {  \_device->DrawPrimitive(  D3DPT\_POINTLIST,  \_vbOffset,  numParticlesInBatch);  }  // next block  \_vbOffset += \_vbBatchSize;  postRender();  }//end if  }// end render() |

### 14.2.2 随机

这有一个随机的粒子系统。例如，如果我们模拟雪花，不能让所有雪花以完全相同的方式落下。我们要让它们按相似的方式落下而不是完全相同的方式。为了使粒子系统的随机功能更简单，我们增加了下列两个函数到d3dUtility.h/cpp文件。

第一个函数在[lowBound, highBound]区间内随机的返回一个Float类型值：

|  |
| --- |
| float d3d::GetRandomFloat(float lowBound, float highBound)  {  if( lowBound >= highBound ) // bad input  return lowBound;  // get random float in [0, 1] interval  float f = (rand() % 10000) \* 0.0001f;  // return float in [lowBound, highBound] interval.  return (f \* (highBound - lowBound)) + lowBound;  } |

第二个函数在边界盒的范围内，输出一个随机的向量。

|  |
| --- |
| void d3d::GetRandomVector(  D3DXVECTOR3\* out,  D3DXVECTOR3\* min,  D3DXVECTOR3\* max)  {  out->x = GetRandomFloat(min->x, max->x);  out->y = GetRandomFloat(min->y, max->y);  out->z = GetRandomFloat(min->z, max->z);  } |

注意：记得用srand()去seed随机数生成器。

## 14.3 具体的粒子系统：雪、火、粒子枪

现在让我们用Psystem类开始一个具体的粒子系统，为了说明用意，这些系统的设计很简单，没有用到Psystem类所提供的所有灵活性。我们实现雪、火、粒子枪系统。这些系统的名字基本上概括了他们的模型。雪系统模拟下落的雪花，火系统模拟看上去像火焰的爆炸，粒子枪系统从照相机位置向对面发射出粒子（用键盘）。

注意：照例，用全部的工程代码来说明这些系统，你能够在本章找到这些文件。

### 14.3.1 例子程序：雪



图14.2 雪系统例子的屏幕截图

雪系统类定义如下：

|  |
| --- |
| class Snow : public PSystem  {  public:  Snow(d3d::BoundingBox\* boundingBox, int numParticles);  void resetParticle(Attribute\* attribute);  void update(float timeDelta);  }; |

备注：因为父类做了大部分的工作，所以雪系统的接口非常简单。事实上，我们在这一节中实现的这三个粒子系统，接口简单并相对容易实现。

构造器提供一个点给边界盒结构，边界盒是粒子系统的成员。边界盒描述雪花在哪个范围内（体积范围）下落，如果雪花出了边界盒，它将被杀死并再生。这样，雪系统始终能保存有同样数量的激粒子，构造器的实现：

|  |
| --- |
| Snow::Snow(d3d::BoundingBox\* boundingBox, int numParticles)  {  \_boundingBox = \*boundingBox;  \_size = 0.8f;  \_vbSize = 2048;  \_vbOffset = 0;  \_vbBatchSize = 512;  for(int i = 0; i < numParticles; i++)  addParticle();  } |

同样注意：我们指定顶点缓存的尺寸，每一批的尺寸和开始的偏移。

ResetParticle方法创建一个雪花，在x、z轴随机的位置并在边界盒的范围内。设置y轴高度为边界盒的顶部。如果给雪花一个速度，以便让雪花下落时稍稍向左倾斜。雪花是白色的。

|  |
| --- |
| void Snow::resetParticle(Attribute\* attribute)  {  attribute->\_isAlive = true;  // get random x, z coordinate for the position of the snowflake.  d3d::GetRandomVector(  &attribute->\_position,  &\_boundingBox.\_min,  &\_boundingBox.\_max);  // no randomness for height (y-coordinate). Snowflake  // always starts at the top of bounding box.  attribute->\_position.y = \_boundingBox.\_max.y;  // snowflakes fall downward and slightly to the left  attribute->\_velocity.x = d3d::GetRandomFloat(0.0f, 1.0f)\*-3.0f;  attribute->\_velocity.y = d3d::GetRandomFloat(0.0f, 1.0f)\*-10.0f;  attribute->\_velocity.z = 0.0f;  // white snowflake  attribute->\_color = d3d::WHITE;  } |

Update方法更新粒子和粒子间的位置，并且测试粒子是否在系统的边界盒之外，如果它已经跳出边界盒，就再重新创建。

|  |
| --- |
| void Snow::update(float timeDelta)  {  std::list<Attribute>::iterator i;  for(i = \_particles.begin(); i != \_particles.end(); i++)  {  i->\_position += i->\_velocity \* timeDelta;  // is the point outside bounds?  if( \_boundingBox.isPointInside( i->\_position ) == false )  {  // nope so kill it, but we want to recycle dead  // particles, so respawn it instead.  resetParticle( &(\*i) );  }  }  } |

### 14.3.2 例子程序：火

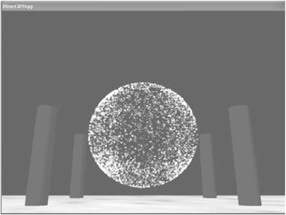


图14.3 火粒子系统例子的屏幕截图

火系统类定义如下：

|  |
| --- |
| class Firework : public PSystem  {  public:  Firework(D3DXVECTOR3\* origin, int numParticles);  void resetParticle(Attribute\* attribute);  void update(float timeDelta); void preRender();  void postRender();  }; |

构造器需要提供一个点作为粒子系统中的原点，和系统中的粒子数，原点是火焰爆发的那个点。

ResetParticle方法在原点位置初始化粒子系统，并在边界球内创建一个随机的速度，粒子系统中的每个例子有一个随机的颜色，我们定义粒子只能存活2秒。

|  |
| --- |
| void Firework::resetParticle(Attribute\* attribute)  {  attribute->\_isAlive = true;  attribute->\_position = \_origin;  D3DXVECTOR3 min = D3DXVECTOR3(-1.0f, -1.0f, -1.0f);  D3DXVECTOR3 max = D3DXVECTOR3( 1.0f, 1.0f, 1.0f);  d3d::GetRandomVector(  &attribute->\_velocity,  &min,  &max);  // normalize to make spherical  D3DXVec3Normalize(  &attribute->\_velocity,  &attribute->\_velocity);  attribute->\_velocity \*= 100.0f;  attribute->\_color = D3DXCOLOR(  d3d::GetRandomFloat(0.0f, 1.0f),  d3d::GetRandomFloat(0.0f, 1.0f),  d3d::GetRandomFloat(0.0f, 1.0f),  1.0f);  attribute->\_age = 0.0f;  attribute->\_lifeTime = 2.0f; // lives for 2 seconds  } |

Update方法更新每个粒子的位置，并在粒子超出自己的生活周期时杀死它。注意：这个系统不能移除死掉的粒子，这么做是因为我们想产生一个新的火焰的时候，我们只要简单的重新设置已经存在的死了的火焰系统就可以了。这样为我们不必频繁的去产生和释放粒子。

|  |
| --- |
| void Firework::update(float timeDelta)  {  std::list<Attribute>::iterator i;  for(i = \_particles.begin(); i != \_particles.end(); i++)  {  // only update living particles  if( i->\_isAlive )  {  i->\_position += i->\_velocity \* timeDelta;  i->\_age += timeDelta;  if(i->\_age > i->\_lifeTime) // kill  i->\_isAlive = false;  }  }  } |

当渲染时，火系统使用不同的方法渲染像素。进一步讲，它不写深度缓存，我们可以简单的改变混合像素，通过重写PSystem::preRender方法和PSystem::postRender方法，下面是重写的实现：

|  |
| --- |
| void Firework::update(float timeDelta)  {  std::list<Attribute>::iterator i;  void Firework::preRender()  {  PSystem::preRender();  \_device->SetRenderState(D3DRS\_SRCBLEND, D3DBLEND\_ONE);  \_device->SetRenderState(D3DRS\_DESTBLEND, D3DBLEND\_ONE);  // read, but don't write particles to z-buffer  \_device->SetRenderState(D3DRS\_ZWRITEENABLE, false);  }  void Firework::postRender()  {  PSystem::postRender();  \_device->SetRenderState(D3DRS\_ZWRITEENABLE, true);  } |

注意：这两个方法调用父类版本，这样，我们仍能重新使用父类的一些功能，做一些小的改变就变成了火焰系统。

### 14.3.3 例子程序：粒子枪

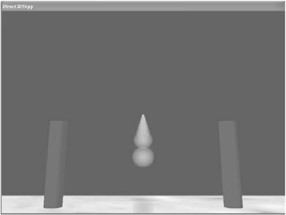


图14.4 激光枪系统的截图

下面是粒子枪系统的定义：

|  |
| --- |
| class ParticleGun : public PSystem  {  public:  ParticleGun(Camera\* camera);  void resetParticle(Attribute\* attribute);  void update(float timeDelta);  private:  Camera\* \_camera;  }; |

构造器需要提供一个照相机的位置点，这是因为系统需要知道照相机的位置及朝向，以决定在哪创建一个粒子。

ResetParticle方法设置粒子的位置为当前照相机的位置，并且设置方向上的速度，在照像机视角的100个单位。这样，子弹将射向我们正在看的方向，粒子颜色为绿色。

|  |
| --- |
| void ParticleGun::resetParticle(Attribute\* attribute)  {  attribute->\_isAlive = true;  D3DXVECTOR3 cameraPos;  \_camera->getPosition(&cameraPos);  D3DXVECTOR3 cameraDir;  \_camera->getLook(&cameraDir);  // change to camera position  attribute->\_position = cameraPos;  attribute->\_position.y -= 1.0f; // slightly below camera so it's  // like we're carrying gun  // travels in the direction the camera is looking  attribute->\_velocity = cameraDir \* 100.0f;  // green  attribute->\_color = D3DXCOLOR(0.0f, 1.0f, 0.0f, 1.0f);  attribute->\_age = 0.0f;  attribute->\_lifeTime = 1.0f; // lives for 1 seconds  } |

Update方法更新粒子的位置，并且杀死超过其生命周期的粒子，然后，我们搜索粒子列表删除已经死了的粒子。

|  |
| --- |
| {  std::list<Attribute>::iterator i;  for(i = \_particles.begin(); i != \_particles.end(); i++)  {  i->\_position += i->\_velocity \* timeDelta;  i->\_age += timeDelta;  if(i->\_age > i->\_lifeTime) // kill  i->\_isAlive = false;  }  removeDeadParticles();  } |

## 14.4 摘要

* 用点精灵来显示一个粒子是方便且灵活的，它可能改变粒子尺寸、给粒子赋予纹理。此外，能够使用简单的顶点（vertex）来描述它们。
* 粒子系统维护一个粒子的集合，并负责创建、释放、更新和显示粒子。
* 还有一些其他粒子系统的概念，是你能够实现的：烟，火箭的轨迹，喷泉/河水车效果，火，光，爆炸，和雨。

# 第十五章 选取 (Picking)

## 概览(OVERVIEW)

如果用户点击了屏幕上的点 **s** = (*x, y*)。 从图15.1 我们能看到用户选取了茶壶。 无论如何，应用程序无法根据给定的**s**点就立即确定茶壶是被选取。所以，我们必须拿出计算这个动作技巧来，叫做**选取技巧**。

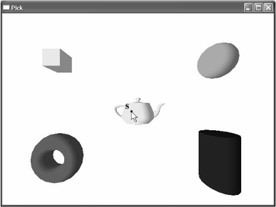


图15.1 用户正在选择茶壶

我们知道一些知识：关于茶壶和它的关联点**s**，茶壶投影在围绕**s**点的区域，更准确的说是：它投影到投影窗口上围绕**p**点的区域，与它对应的屏幕点是**s**。因为这个问题依赖于3D物体与它的投影之间的关系，我们看图15.2就可以了解。

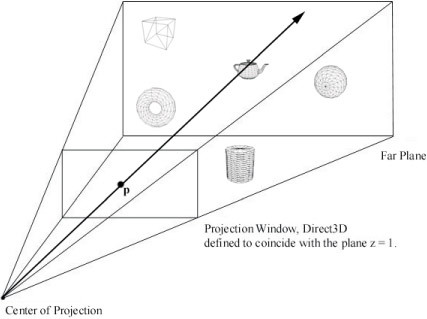


图15.2放射线穿过点**p**点将会相交于围绕**p**点投影的对象。 注意：在投影窗口上的点 **p**与荧屏上被按下了点**s**相关联。

图15.2我们看到如果我们发射一条选取射线，从原点发出，经过点**p**，会与围绕**p**点投影的对象相交，即茶壶。所以一旦我们计算选取射线，我们可以遍例场景中的每个对象并测试，看射线是否与它相交。与射线相交的对象即是用户选择的对象，在这个例子中用户选取的对象是茶壶。

上面的例子讲解了点**s**与茶壶的关系。通常我们任意点击屏幕上的点，我们遍例场景中的每个对象，如果对象与射线相交，那么这个对象就是用户选取的对象。例如，图15.1中，如果用户没有点击5个对象中的一个，而是点击了白色的背景区域，射线将不能相交任何对象。因此，结论是：如果射线没有与场景中的任何对象相交，则用户没有点击任何一个对象，其它的我们不关心。

“选取”适用于所有种类的游戏和3D程序。例如，玩家通过用鼠标点击来影响3D世界中的不同对象，玩家可能点击向敌人射击，或点击拾取物品。好的程序会适当做出反应，程序需要知道哪个对象被选取（是敌人还是物品），和在3D空间中的位置（开枪会击中哪？或玩家将要移动到哪去拾取物品？）。选取回答了我们这些问题。

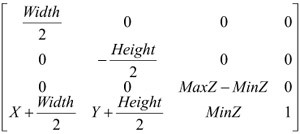
## 目标

学习如何计算选取算法并了解它是如何工作的，我们将选取分解成四步：

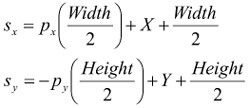
1. 给一个屏幕点**s**，找出它在投影窗口上相交的点，即**p**。
2. 计算射线，它是从原点出发并经过点**p**。
3. 转换射线与模型到同一空间。
4. 测试与射线相交的对象，相交的对象即是屏幕上点击的对象。

## 15.1 屏幕到投影窗口的转换

首先，转换屏幕点到投影窗口，视口变换矩阵是：

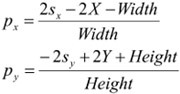


根据屏幕上的点**s** = (*sx, sy*)，通过视口转换，得到在投影窗口上的点**p** = (*px, py, pz*)：

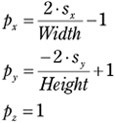


回忆一下2D图形部分：视口转换后z轴是不用保存的，而被保存在z缓存中。

给出屏幕点**s**，我们要找到点**p**，使用下列公式：

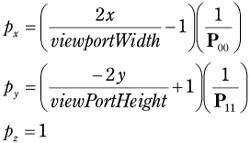


假定视口成员x和y都是0，通常我们能进一步得到：



因为前面的定义，投影窗口就是z=1的平面，所以*pz* = 1。

可是我们还什么都没做，投影矩阵缩放投影窗口上的点，来模拟不同的视角。为了返回缩放前的点值，我们必须用与缩放相反的操作来转换点。**P**是投影矩阵，因为**P**00 和 **P**11转换距阵缩放点的x和y坐标，我们得到：



## 15.2 计算射线

回忆一下，射线能够描述参数方程：**p**(*t*) = **p**0 + *t***u。**其中**p**0是射线的起点，用来描述它的位置，**u**是向量，用来描述它的方向。

如图15.2，我们知道射线的起点总是视图空间的原点，所以**p**0 = (0, 0, 0)，如果**p**是射线穿过投影窗口上的点，方向向量**u**给出：**u** = **p** - **p***0* = (*px,* *py,* 1) - (0, 0, 0) = **p**。

下面的方法用来计算选取射线（从屏幕空间点击的点所对应的**视图空间的点x、y坐标**）：

|  |
| --- |
| d3d::Ray CalcPickingRay(int x, int y)  {  float px = 0.0f;  float py = 0.0f;  D3DVIEWPORT9 vp;  Device->GetViewport(&vp);  D3DXMATRIX proj;  Device->GetTransform(D3DTS\_PROJECTION, &proj);  px = ((( 2.0f\*x) / vp.Width) - 1.0f) / proj(0, 0);  py = (((-2.0f\*y) / vp.Height) + 1.0f) / proj(1, 1);  d3d::Ray ray;  ray.\_origin = D3DXVECTOR3(0.0f, 0.0f, 0.0f);  ray.\_direction = D3DXVECTOR3(px, py, 1.0f);  return ray;  }  where Ray is defined as:  struct Ray  {  D3DXVECTOR3 \_origin;  D3DXVECTOR3 \_direction;  }; |

我们更新d3dUtility.h文件，在d3d命名空间中加入选取射线**Ray**。

## 15.3 变换射线

上一节讲到，选取射线的计算被描述在视图空间，为了完成射线的相交的测试，射线和对象必须在同一个坐标系统。**通常转换射线到世界空间**（甚至对象在本地空间）要好于将所有对象转换到视图空间。

我们能够将一个变换矩阵转换为一条原点为**p**0，方向为**u**的射线**r**(*t*) = **p**0 + *t***u**，注意：原点转换为一个点，方向转换为一个向量，在本章的选取例子中，下列函数转换一条射线：

|  |
| --- |
| void TransformRay(d3d::Ray\* ray, D3DXMATRIX\* T)  {  // transform the ray's origin, w = 1.  D3DXVec3TransformCoord(  &ray->\_origin,  &ray->\_origin,  T);  // transform the ray's direction, w = 0.  D3DXVec3TransformNormal(  &ray->\_direction,  &ray->\_direction,  T);  // normalize the direction  D3DXVec3Normalize(&ray->\_direction, &ray->\_direction);  } |

D3DXVec3TransformCoord和D3DXVec3TransformNormal接受一个Ray类型参数（包含二个3D向量成员）。 D3DXVec3TransformCoord函数中，射线的原点（\_origin）向量的第四部分***w*** = 1。相反，函数D3DXVec3TransformNormal中，射线的方向（\_direction）向量的第四部分***w*** = 0。

这样，当我们向世界空间转换时，能够用D3DXVec3TransformCoord转换一个点，用D3DXVec3TransformNormal转换一个向量。

## 15.4 射线－对象 交点

我们将射线和对象转换到同一坐标系统后，准备测试哪个对象与射线相交。因为我们将对象描述为三角形组成的网络，下面详细说明这种方法。遍例场景中每个对象的三角形列表并测试，如果射线相交于一个三角形，它就与三角形所在的对象相交。

然而，通过遍例场景中的每个三角形来实现射线相交在计算上会增加时间，一种比较快的方法，虽然准确性会差一点。它将每个对象围成一个近似的球形（边界球），这样我们就能通过遍例每个边界球来测试射线相交。用边界球来描述相交的对象。

注意：射线可能相交多个对象，然而离照相机近的对象会被选取。因为近距离对象遮挡了后面的对象。

给出一个边界球的圆心**c**和半径**r**，使用下列恒等式能够测试点**p**是否在边界球上：

||**p-c**||-*r* = 0

如果恒等式满足，则点**p**在边界球上。如图15.3

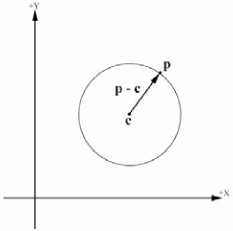


图15.3 向量p到c的长度表示为：**||**p - c**||**，如果等于半径则表示点p在边界球上。注意：我们使用边界球是为了方便，但这将扩展出三个种情况。

假定射线**p**(*t*) = **p**0 + *t***u**相交于边界球，我们将射线代入球的恒等式中，使参数t满足了球的恒等式，给出了满足相交点的参数。

将射线**p**(*t*) = **p**0 + *t***u**代入球的恒等式：

**||p**(*t*) - **c||** - ***r*** = 0 🡪 **||p**0 + *t***u** - **c||** - ***r*** = 0

通过以上推导，我们得到二次方程：

*At*2 + *Bt* + *C* = 0

其中*A* = **u** · **u**, *B* = 2(**u** · (**p**0 - **c**))，而*C* = (**p**0 - **c**) . (**p**0 - **c**) – ***r*** *2。*

如果**u**是标准化的，那么*A* = 1。

因为**u**是标准化的，我们解*t*0 和 *t*1：

无标题

图15.4显示可能返回的*t*0 和 *t*1，并显示了一些返回值的几何意义：

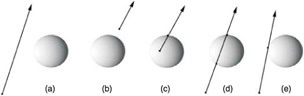


图15.4 (a)射线从球边上擦过；(b)射线在球前；(c)射线在球的内部；(d)射线相交于球；(e)射线是球的切线。

下列方法射线通过并与边界球相交，返回true；射线错过边界球，返回false。

|  |
| --- |
| bool PickApp::raySphereIntersectionTest(Ray\* ray,  BoundingSphere\* sphere)  {  D3DXVECTOR3 v = ray->\_origin - sphere->\_center;  float b = 2.0f \* D3DXVec3Dot(&ray->\_direction, &v);  float c = D3DXVec3Dot(&v, &v) - (sphere->\_radius \* sphere->  \_radius);  // find the discriminant  float discriminant = (b \* b) - (4.0f \* c);  // test for imaginary number  if( discriminant < 0.0f )  return false;  discriminant = sqrtf(discriminant);  float s0 = (-b + discriminant) / 2.0f;  float s1 = (-b - discriminant) / 2.0f;  // if a solution is >= 0, then we intersected the sphere  if( s0 >= 0.0f || s1 >= 0.0f )  return true;  return false;  } |

当然，我们已经准备了一个边界球，为了便于理解我们再次显示它的定义：

|  |
| --- |
| bool PickApp::raySphereIntersectionTest(Ray\* ray,  BoundingSphere\* sphere)  {  struct BoundingSphere  {  BoundingSphere();  D3DXVECTOR3 \_center;  float \_radius;  }; |

## 15.5 例子程序：选取

图15.5显示了本章例子程序的屏幕截图，茶壶绕着屏幕移动，你可以用鼠标试着点击它。如果你点击到茶壶的边界球上，一个消息框将弹出，表示你点中了。我们通过测试WM\_LBUTTONDOWN消息来处理鼠标点击事件：

|  |
| --- |
| case WM\_LBUTTONDOWN:  // compute the ray in view space given the clicked screen point  d3d::Ray ray = CalcPickingRay(LOWORD(lParam), HIWORD(lParam));  // transform the ray to world space  D3DXMATRIX view;  Device->GetTransform(D3DTS\_VIEW, &view);  D3DXMATRIX viewInverse;  D3DXMatrixInverse(&viewInverse, 0, &view);  TransformRay(&ray, &viewInverse);  // test for a hit  if( RaySphereIntTest(&ray, &BSphere) )  ::MessageBox(0, "Hit!", "HIT", 0);  break; |

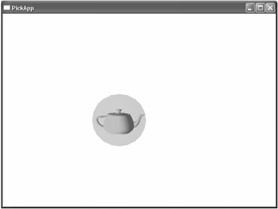


图15.5 这章例子程序的屏幕截图

## 15.6 摘要

* 选取技巧通过鼠标点击，来确定与3D对象对应的屏幕上显示的2D投影对象。
* 选取线是一个射线，源自视图空间的原点，穿过投影窗口上的点关联到屏幕上点击的点。
* 我们能够变换一个射线**r**(*t*) = **p**0 + *t***u**，通过变换原点**p**0和通过矩阵变换得到方向**u**。注意：原点变换自一个（w=1）的点，方向是（w=0）的向量。
* 测试射线与对象相交，我们能测试射线是否相交于组成对象的三角形，或测试射线是否相交于围绕对象的一个体积，比如边界球。

# 第四部分 着色器和特效 (Shaders and Effects)

# 第十六章 高级着色器语言入门 (Introduction to the High-Level Shading Language)

在这一章里我们描述高级着色器语言（High-Level Shading Language ，简称HLSL），在下三章里我们用它去编写顶点和像素着色器。简单的说，在我们写的程序里顶点和像素是很小的对象，它们由GPU来执行，是固定功能管线的一部分。用我们自己写的着色器程序替换一部分固定功能管线，在绘制效果上我们获得很大的灵活性。我们不再局限于预定义的"固定"操作。

为了编写着色器程序，我们需要一种语言。 在DirectX 8.x,中，着色器是用低级着色器汇编语言编写的。幸运的是，我们不必再用汇编语言来写着色器了，DirectX 9支持一种高级着色器语言来xyna写。用HLSL在汇编语言来写着色器程序与使用高级语言有同样的优势，像C++，它超越了汇编语言，即：

* 增加生产力—用高级语言比用低级语言写程序更快、更容易。 我们可以花费更多的时间关注于算法而不是代码。
* 增加可读性—用高级语言写的程序更易读，这意味着用高级语言编程更易于调试和维护。
* 大多数情况下，编译器产生的汇编代码比手写有效率。
* 使用HLSL 编译器，我们可以编译我们的代码到任何可用shader版本，使用汇编语言我们将不得不为每个需要的版本移植代码。

HLSL 同C和C++语法很类似, 所以缩短了学习曲线。

最后，如果你的显卡不支持顶点和像素着色器的话，为了执行着色器的例子程序你将需要转换REF设备。使用REF设备意味着着色器例子运行的会很慢，但它至少能显示结果，让我们去检查是否代码可以被执行。

提示：顶点shaders可以用软件来模拟 ―― D3DCREATE\_SOFTWARE\_VERTEX-PROCESSING。16.1

## 目标

* 学习如何定、编译一个HLSL 着色器程序。
* 学习如何将程序中的数据传送到着色器程序。
* 熟悉语法、类型，和HLSL的内建函数。

## 16.1 编写HLSL 着色器

我们可以在程序源文件中用长字符串直接编写HLSL 着色器代码，然而更方便、更模块化的方法是把它与程序代码分离出来。因此，我们在记事本中编写着色器并保存成一般的ASCII文本文件，然后可以用D3DXCompileShaderFromFile函数(section 16.2.2)来编译它们。

作为介绍，下面是用HLSL编写的一个简单的顶点着色器，用记事本生成并保存成文本文件“Transform.txt”。全部工程都在标题为Transform的目录下，顶点着色器用组合视图和投影矩阵转换顶点，并设置顶点漫射光为蓝色。

注意：这是一个顶点着色器的例子，不必关心顶点着色器做了什么，这是下一章包含的内容，现在的目标是熟悉HLSL编程的语法和格式。

|  |
| --- |
| // File: transform.txt  // Author: Frank D. Luna (C) All Rights Reserved  // System: AMD Athlon 1800+ XP, 512 DDR, Geforce 3, Windows XP,  // MSVC++ 7.0  // Desc: 顶点着色器用组合视图和投影矩阵转换顶点，并设置顶点漫射光为蓝色.  //全局变量  //用来保存视图和投影的组合矩阵，在程序中初始化变量  matrix ViewProjMatrix;  // 初始化颜色变量（蓝色）  vector Blue = {0.0f, 0.0f, 1.0f, 1.0f};  // 结构  // Input结构用来描述输入到着色器的顶点，这个Input顶点只包含一个位置成员  struct VS\_INPUT  {  vector position : POSITION;  };  //Output结构用来描述从着色器输出的顶点，这个Output顶点包含位置和颜色成员  struct VS\_OUTPUT  {  vector position : POSITION;  vector diffuse : COLOR;  };  //主入口点，这个main函数接收一个Input顶点的拷贝作为参数，返回一个Output顶点的拷贝  VS\_OUTPUT Main(VS\_INPUT input)  {  // 将output结构所有成员初始化  VS\_OUTPUT output = (VS\_OUTPUT)0;  // 将位置变换到投影空间  output.position = mul(input.position, ViewProjMatrix);  // 设置顶点颜色  output.diffuse = Blue;  //Output the projected and colored vertex.  return output;  } |

### 16.1.1 全局变量

首先是2个全局变量：

matrix ViewProjMatrix;

vector Blue = {0.0f, 0.0f, 1.0f, 1.0f};

第1个变量ViewProjMatrix是矩阵类型，它是一个在HLSL 内创建的4×4的矩阵类型。这个变量保存视图与投影的组合矩阵，它描述两者的变换。使用这种方法我们只要做一个向量和矩阵的乘法（而不是二个）。注意，在着色器源代码的任何地方都没有初始化这个变量，因为它是我们在应用程序的源代码里设置的，而不是在着色器中。从应用程序向着色器程序通讯是常用的操作，例子在16.2.1节。

第二个变量Blue是built-in（内建）类型的4D向量，我们简单的将它初始化成蓝色，它是个RGBA的颜色向量。

### 16.1.2 输入和输出结构

在全局变量定义之后，定义2个特殊的结构，我们调用输入和输出结构。对于顶点着色器而言，这些结构定义了顶点的数据，分别是：

|  |
| --- |
| struct VS\_INPUT  {  vector position : POSITION;  };  struct VS\_OUTPUT  {  vector position : POSITION;  vector diffuse : COLOR;  }; |

注意：给像素着色器的结构定义输入和输出像素数据。

在例子中，INPUT 顶点着色器只包含位置成员（POSITION），OUTPUT顶点着色器包含位置和颜色成员（POSITION and COLOR）。

特殊的冒号是一种语义，用于是声明变量。这与vertex结构中的自由顶点格式（FVF）相似。例如，在VS\_INPUT中有成员：vector position : POSITION;

": COLOR"是说顶点的漫射光是用VS\_OUTPUT结构的COLOR成员来说明的。在下一章中对于顶点和像素着色器中向量的标识符用法，我们将会有更多的讨论。

注意：从底层来说，着色器变量的语义和语法同硬件寄存器是相关联的。即，input变量与input寄存器关联，output变量与output寄存器关联。例如，VS\_INPUT中的position成员与顶点input的position寄存器相关联。同样，diffuse与顶点的output 的color寄存器关联。

### 16.1.3 函数的入口点

在C++程序中，每个HLSL程序有一个入口点。在我们的着色器例子中，我们调用入口点函数main。然而名字不是强制的。入口点函数名可以是任何有效的函数名，入口点函数必须有一个input结构参数，它通过input顶点进入着色器。入口点函数必须返回一个output结构实例，在着色器中使用output操作顶点。

|  |
| --- |
| VS\_OUTPUT Main(VS\_INPUT input)  { |

注意：实际上，使用input、output结构不是强制的。例如，有时你将会看到使用类似下面的语法，特别是在像素着色器中：

|  |
| --- |
| float4 Main(in float2 base : TEXCOORD0,  in float2 spot : TEXCOORD1,  in float2 text : TEXCOORD2) : COLOR  {  ...  } |

例子中，输入到着色器中的参数是3个纹理坐标。着色器输出（返回）一个颜色，COLOR语句在函数的声明以后。这种定义是类似于：

|  |
| --- |
| struct INPUT  {  float2 base : TEXCOORD0;  float2 spot : TEXCOORD1;  float2 text : TEXCOORD2;  };  struct OUTPUT  {  float4 c : COLOR;  };  OUTPUT Main(INPUT input)  {  ...  } |

输入点函数负责根据给定的input顶点计算output顶点。例子中的着色器简单的变换input顶点到视图空间和投影空间，设置顶点颜色为蓝色，并返回结果顶点。首先我们定义VS\_OUTPUT的实例并初始化所有成员为0。

|  |
| --- |
| VS\_OUTPUT output = (VS\_OUTPUT)0; // zero out all members |

然后着色器变换input顶点位置用ViewProjMatrix变量，使用mul 函数。它是一个built-in（内建）函数，实现向量与矩阵相乘，或矩阵与矩阵相乘。我们保存结果变换的向量（在output实例的position成员中）。

|  |
| --- |
| // 变换后为投影空间的位置  output.position = mul(input.position, ViewProjMatrix); |

然后设置output的成员diffuse的颜色为蓝色：

|  |
| --- |
| // 设置顶点颜色  output.diffuse = Blue; |

最后返回结果向量：

|  |
| --- |
| return output;  } |

## 16.2 编译HLSL 着色器

### 16.2.1 常量表

每个着色器有一个常量表，用来保存它的变量。D3DX库通过ID3DXConstantTable接口，提供给应用程序访问着色器的常量表。通过这个接口我们能够在应用程序中设置着色器源代码中的变量。

我们现在描述一个被节选了的ID3DXConstantTable接口的方法列表的实现，全部的列表请查阅Direct3D文档。

#### 16.2.1.1 取得常量句柄

为了在应用程序中设置着色器中的一个特定变量，需要有一种方法去引用它，我们能够在应用程序中用D3DXHANDLE引用一个在着色器中的变量，下面的方法返回一个着色器中的变量的D3DXHANDLE，使用时，需要传递一个变量的名字作为参数：

|  |
| --- |
| D3DXHANDLE ID3DXConstantTable::GetConstantByName(  D3DXHANDLE hConstant, // scope of constant  LPCSTR pName // name of constant  ); |

* Hconstant——我们要取得的父结构中变量句柄的D3DXHANDLE标识。例如，如果我们想获得一个特定数据结构中单一数据成员的句柄，我们可以传递结构实例的句柄。如果我们获得一个顶级变量的句柄，给这个参数设为0。
* PName——我们想获得的句柄的着色器代码中的变量的名字。

例如，如果在着色器中变量的名字为ViewProjMatrix，并且这是顶级变量，我们这么写：

|  |
| --- |
| // 取得着色器中ViewProjMatrix变量的句柄  D3DXHANDLE h0;  h0 = ConstTable->GetConstantByName(0, "ViewProjMatrix"); |

#### 16.2.1.2 设置常量

一旦应用程序有了一个D3DXHANDLE，要引用着色器代码中的具体变量，我们可以在应用程序中使用ID3DXConstantTable::SetXXX方法设置变量。如果我们想设置一个向量数组类型的变量，方法名是SetVectorArray。

ID3DXConstantTable::SetXXX的一般语法是：

|  |
| --- |
| HRESULT ID3DXConstantTable::SetXXX(  LPDIRECT3DDEVICE9 pDevice,  D3DXHANDLE hConstant,  XXX value  ); |

* PDevice：常量表所关联的设备的指针。
* HConstant：我们正在设置的变量句柄的引用。
* Value：我们要把变量设置成的值，XXX是我们设置的要替换的变量类型名，对于有些类型（bool, int, float），传递变量值的COPY，另外一些类型（vectors, matrices, structures），传递值的指针。

下面列表描述了我们能用ID3DXConstantTable接口设置的类型列表。这里假定我们有一个有效的设备，和一个有效句柄。

**SetBool**—Used to set a Boolean value. Sample call:

|  |
| --- |
| bool b = true;  ConstTable->SetBool(Device, handle, b); |

**SetBoolArray**—Used to set a Boolean array. Sample call:

|  |
| --- |
| bool b[3] = {true, false, true};  ConstTable->SetBoolArray(Device, handle, b, 3); |

**SetFloat**—Used to set a float. Sample call:

|  |
| --- |
| float f = 3.14f;  ConstTable->SetFloat(Device, handle, f); |

**SetFloatArray**—Used to set a float array. Sample call:

|  |
| --- |
| float f[2] = {1.0f, 2.0f};  ConstTable->SetFloatArray(Device, handle, f, 2); |

**SetInt**—Used to set an integer. Sample call:

|  |
| --- |
| int x = 4;  ConstTable->SetInt(Device, handle, x); |

**SetIntArray**—Used to set an integer array. Sample call:

|  |
| --- |
| int x[4] = {1, 2, 3, 4};  ConstTable->SetIntArray(Device, handle, x, 4); |

**SetMatrix**—Used to set a 4 × 4 matrix. Sample call:

|  |
| --- |
| D3DXMATRIX M(…);  ConstTable->SetMatrix(Device, handle, &M); |

**SetMatrixArray**—Used to set a 4 × 4 matrix array. Sample call:

|  |
| --- |
| D3DXMATRIX M[4];  // ...Initialize matrices  ConstTable->SetMatrixArray(Device, handle, M, 4); |

**SetMatrixPointerArray**—Used to set an array of 4 × 4 matrix pointers. Sample call:

|  |
| --- |
| D3DXMATRIX\* M[4];  // ...Allocate and initialize matrix pointers  ConstTable->SetMatrixPointerArray(Device, handle, M, 4); |

**SetMatrixTranspose**—Used to set a transposed 4 × 4 matrix. Sample call:

|  |
| --- |
| D3DXMATRIX M(…);  D3DXMatrixTranspose(&M, &M);  ConstTable->SetMatrixTranspose(Device, handle, &M); |

**SetMatrixTransposeArray**—Used to set an array of 4 × 4 transposed matrices. Sample call:

|  |
| --- |
| D3DXMATRIX M[4];  // ...Initialize matrices and transpose them.  ConstTable->SetMatrixTransposeArray(Device, handle, M, 4); |

**SetMatrixTransposePointerArray**—Used to set an array of pointers to 4 × 4 transposed matrices. Sample call:

|  |
| --- |
| D3DXMATRIX\* M[4];  // ...Allocate,initialize matrix pointers and transpose them.  ConstTable->SetMatrixTransposePointerArray(Device, handle, M, 4); |

**SetVector**—Used to set a variable of type D3DXVECTOR4. Sample call:

|  |
| --- |
| D3DXVECTOR4 v(1.0f, 2.0f, 3.0f, 4.0f);  ConstTable->SetVector(Device, handle, &v); |

**SetVectorArray**—Used to set a variable that is a vector array. Sample call:

|  |
| --- |
| D3DXVECTOR4 v[3];  // ...Initialize vectors  ConstTable->SetVectorArray(Device, handle, v, 3); |

**SetValue**—Used to set an arbitrarily sized type, such as a structure. In the sample call, we use SetValue to set a D3DXMATRIX:

|  |
| --- |
| D3DXMATRIX M(…);  ConstTable->SetValue(Device, handle, (void\*)&M, sizeof(M)); |

#### 16.2.1.3 设置常量默认值

下一个方法就是设置常量的默认值，这些默认值在声明时初始化。这个方法应该在应用程序建立（setup）期间被一次性调用（called once）。

|  |
| --- |
| HRESULT ID3DXConstantTable::SetDefaults(  LPDIRECT3DDEVICE9 pDevice  ); |

pDevice——关联到常量表的设备的指针。

#### 16.2.2 编译HLSL着色器

我们可以编译一个着色器——用我们已保存的着色器的文本文件——使用下列函数：

|  |
| --- |
| HRESULT D3DXCompileShaderFromFile(  LPCSTR pSrcFile,  CONST D3DXMACRO\* pDefines,  LPD3DXINCLUDE pInclude,  LPCSTR pFunctionName,  LPCSTR pTarget,  DWORD Flags,  LPD3DXBUFFER\* ppShader,  LPD3DXBUFFER\* ppErrorMsgs,  LPD3DXCONSTANTTABLE\* ppConstantTable  ); |

* pSrcFile——要编译的包含着色器源代码的文本文件的文件名
* pDefines——参数可选，本书中指定为空。
* pInclude——ID3DXInclude接口指针。这个接口被设计成由应用程序实现，所以我们可以重载默认include的行为。通常，默认行为就可以了，而且我们可以通过将其指定为空忽略此参数。
* pFunctionName——指定入口点函数名的字符串。例如，如果着色器的入口点函数叫做Main，我们可以给此参数传递“Main”。
* pTarget——指定要编译成的HLSL着色器源文件的版本的字符串。有效的顶点着色器版本是：vs\_1\_1, vs\_2\_0, vs\_2\_sw。有效的像素着色器版本是2.0，我们可以给此参数传递vs\_2\_0。

备注：有编译不同版本着色器的能力，是HLSL与汇编语言比的主要优势。用HLSL我们只需为需要的目标简单的重新编译，便可快速移植着色器到不同的版本。使用汇编，我们可能需要手动移植代码。

* Flags——可选的编译标记，指定为0标识没有标记。有效的选项是：
* D3DXSHADER\_DEBUG——通知编译器写入调试信息
* D3DXSHADER\_SKIPVALIDATION——通知编译器不要做任何代码检查。此项仅用于你已知着色器能够工作时
* D3DXSHADER\_SKIPOPTIMIZATION——通知编译器不要执行任何代码优化。实践中，这个选项应该仅用于调试，因为这种情况下你不希望编译器以任何方式修改代码。
* ppShader——返回已编译的着色器代码的ID3DXBuffer指针。这个已编译过的着色器代码将作为另一个实际创建顶点/像素着色器函数的参数
* ppErrorMsgs——返回包含错误码和错误消息字符串的ID3DXBuffer指针
* ppConstantTable——返回包含此着色器常量表数据的ID3DXConstantTable指针

这里是一个调用D3DXCompileShaderFromFile的例子：

|  |
| --- |
| // Compile shader  ID3DXConstantTable\* TransformConstantTable = 0;  ID3DXBuffer\* shader = 0;  ID3DXBuffer\* errorBuffer = 0;  hr = D3DXCompileShaderFromFile(  "transform.txt", // shader filename  0,  0,  "Main", // entry point function name  "vs 2 0", // shader version to compile to  D3DXSHADER\_DEBUG, // debug compile  &shader,  &errorBuffer,  &TransformConstantTable);  // output any error messages  if( errorBuffer )  {  ::MessageBox(0, (char\*)errorBuffer->GetBufferPointer(), 0, 0);  d3d::Release<ID3DXBuffer\*>(errorBuffer);  }  if (FAILED (hr))  {  ::MessageBox(0, "D3DXCreateEffectFromFile() - FAILED", 0, 0);  return false;  } |

## 16.3 变量类型

注意：除了下列各小节中描述的类型外，HLSL还有一些内建的对象类型（如：纹理对象）。但是，由于这些对象类型主要用于效果框架，我们将对其延迟到第19章讨论。  
  
**16.3.1 数值类型**

HLSL支持下列数值类型（scalar type）：

* bool—True or false value. Note that HLSL provides the true and false keywords.
* int—32-bit signed integer
* half—16-bit floating-point number
* float—32-bit floating-point number
* double—64-bit floating-point number

注意：一些平台不支持int, half, and double类型，这时我们使用 float类型模拟。

### 16.3.2 向量类型

HLSL有下列内建的向量类型（vector type）：

* vector——各分量为float类型的4D向量
* vector<T, n>——一个n维向量，其每个分量都为T类型。n维必须在1到4之间。这里是一个2D double向量的例子：

|  |
| --- |
| vector<double, 2> vec2; |

我们可以使用数组下标的语法访问向量的一个分量。例如，要设置向量vec的第i个分量，我们可以写成：

|  |
| --- |
| vec[i] = 2.0f; |

此外，我们可以像访问结构的成员一样访问向量vec的一个分量，使用已定义的分量名x，y，z，w，r，g，b和a。

|  |
| --- |
| vec.x = vec.r = 1.0f;  vec.y = vec.g = 2.0f;  vec.z = vec.b = 3.0f; |

vec.w = vec.a = 4.0f;

名称为r，g，b和a的分量分别对应x，y，z和w的分量。当使用向量来表示颜色时，RGBA符号是更适合的，因为它加强了向量所表示的颜色。

作为选择，我们可以使用其它一些预定义类型，分别用来代表2D，3D和4D向量的类型：

|  |
| --- |
| float2 vec2;  float3 vec3;  float4 vec4; |

考虑向量u = (ux, uy, uz, uw)，假设我们要拷贝u的所有分量到一个像v = (ux, uy, uy, uw)这样的向量v。最直接的方法可能是逐个从u往v拷贝每个分量。但不管怎样，HLSL提供了一种特殊的语法做这些无序的拷贝，它叫做“鸡尾酒”（swizzles）：

|  |
| --- |
| vector u = {l.0f, 2.0f, 3.0f, 4.0f};  vector v = {0.0f, 0.0f, 5.0f, 6.0f};  v = u.xyyw; // v = {1.0f, 2.0f, 2.0f, 4.0f} |

拷贝数组时，我们不必拷贝每个分量。例如，我们可以仅拷贝x和y分量，代码段举例如下：

|  |
| --- |
| vector u = {1.0f, 2.0f, 3.0f, 4.0f};  vector v = {0.0f, 0.0f, 5.0f, 6.0f};  v.xy = u; // v = {l.0f, 2.0f, 5.0f, 6.0f} |

### 16.3.3 矩阵类型

HLSL有下列内建矩阵类型：

* matrix——一个4×4矩阵，其各项类型为float
* matrix<T, m, n>——一个m×n矩阵，其每个成员为类型T。矩阵维数m和n必须在1至4之间。

这里是一个2×2整型矩阵的例子：

|  |
| --- |
| matrix<int, 2, 2> m2x2; |

作为选择，我们可以定义一个m×n矩阵，其m和n在1至4之间，使用下列语法：

floatmxn matmxn;

实例：

|  |
| --- |
| float2x2 mat2x2;  float3x3 mat3x3;  float4x4 mat4x4;  float2x4 mat2x4; |

注意：类型不必是float类型——我们可以使用其它类型。举例来说，我们可以用整型，写成这样：

|  |
| --- |
| int2x2 i2x2;  int2x2 i3x3;  int2x2 i2x4; |

我们可以用二维数组的下标语法访问矩阵中的项。例如，要设置矩阵M的第i，j个项，我们可以写成：

|  |
| --- |
| M[i] [j] = value; |

此外，我们可以像访问结构的成员那样访问矩阵M的项。下列条目已定义：

以1为基数的：

|  |
| --- |
| M.\_11 = M.\_12 = M.\_13 = M.\_14 = 0.0f;  M.\_21 = M.\_22 = M.\_23 = M.\_24 = 0.0f;  M.\_31 = M.\_32 = M.\_33 = M.\_34 = 0.0f;  M.\_41 = M.\_42 = M.\_43 = M.\_44 = 0.0f; |

以0为基数的：

|  |
| --- |
| M.\_m00 = M.\_m01 = M.\_m02 = M.\_m03 = 0.0f;  M.\_m10 = M.\_m11 = M.\_m12 = M.\_m13 = 0.0f;  M.\_m20 = M.\_m21 = M.\_m22 = M.\_m23 = 0.0f;  M.\_m30 = M.\_m31 = M.\_m32 = M.\_m33 = 0.0f; |

有时，我们想要访问矩阵中一个特定的行。我们可以用一维数组的下标语法来做。例如，要引用矩阵M中第i行的向量，我们可以写：

|  |
| --- |
| vector ithRow = M[i]; // get the ith row vector in M |

注意：可以使用两种语法在HLSL中初始化变量：

|  |
| --- |
| vector u = {0.6f, 0.3f, 1.0f, 1.0f};  vector v = {1.0f, 5.0f, 0.2f, 1.0f}; |

也可以，等价的，使用构造风格的语法：

|  |
| --- |
| vector u = vector(0.6f, 0.3f, 1.0f, 1.0f);  vector v = vector(1.0f, 5.0f, 0.2f, 1.0f); |

其它一些例子:

|  |
| --- |
| float2x2 f2x2 = float2x2(1.0f, 2.0f, 3.0f, 4.0f);  int2x2 m = {1, 2, 3, 4};  int n = int(5);  int a = {5};  float3 x = float3(0, 0, 0); |

### 16.3.4 数组

我们可以用类似C++的语法声明特定类型的一个数组。例如：

|  |
| --- |
| float M[4][4];  half p[4];  vector v[12]; |

### 16.3.5 结构

结构的定义和在C++里一样。但是，HLSL里的结构不能有成员函数。这是一个HLSL里的结构的例子：

|  |
| --- |
| struct MyStruct  {  matrix T;  vector n;  float f;  int x;  bool b;  };  MyStruct s; // instantiate  s.f = 5.0f; // member access |

### 16.3.6 typedef关键字

HLSL的typedef关键字功能和C++里的完全一样。例如，我们可以给类型vector<float, 3>用下面的语法命名：

|  |
| --- |
| typedef vector<float, 3> point; |

然后，不用写成：

|  |
| --- |
| vector<float, 3> myPoint; |

……我们只需这样写：

|  |
| --- |
| point myPoint; |

这里是另外两个例子，它展示了如何对常量和数组类型使用typedef关键字：

|  |
| --- |
| typedef const float CFLOAT;  typedef float point2[2]; |

1.3.7 变量前缀

下列关键字可以做变量声明的前缀：

* static——如果带static关键字前缀，那它是全局变量。就表示它不是暴露于着色器之外的。换句话说，它是着色器局部的。如果一个局部变量以static关键字为前缀，它就和C++中static局部变量有相同的行为。也就是说，该变量在函数首次执行时被一次性初始化，然后在所有函数调用中维持其值。如果变量没有被初始化，它就自动初始化为0。static int x = 5;
* uniform——如果变量以uniform关键字为前缀，就意味着此变量在着色器外面被初始化，比如被C++应用程序初始化，然后再输入进着色器。
* extern——如果变量以extern关键字为前缀，就意味着该变量可在着色器外被访问，比如被C++应用程序。仅全局变量可以以extern关键字为前缀。不是static的全局变量默认就是extern。
* shared——如果变量以shared关键字为前缀，就提示效果框架（参见19章）：变量将在多个效果间被共享。仅全局变量可以以shared为前缀。
* volatile——如果变量以volatile关键字为前缀，就提示效果框架（参见19章）：变量将被时常修改。仅全局变量可以以volatile为前缀。
* const——HLSL中的const关键字和C++里的意思一样。也就是说，如果变量以const为前缀，那此变量就是常量，并且不能被改变。const float pi = 3.14f;

## 16.4关键字、语句和强制转换

### 16.4.1 关键字

为便于参考，这里给出一个HLSL定义的关键字列表：

asm bool compile const decl do

double else extern false float for

half if in inline inout int

matrix out pass pixelshader return sampler

shared static string struct technique texture

true typedef uniform vector vertexshader void

volatile while

下面的集合显示了被保留并且未使用但是将来可能成为关键字的标识符：

auto break case catch char class

const\_cast continue default delete dynamic cast enum

explicit friend goto long mutable namespace

new operator private protected public register

reinterpret\_cast short signed sizeof static\_cast switch

template this throw try typename union

unsigned using virtual

### 16.4.2 基本程序流程

HLSL支持很多与C++相似的选择、重复、和一般程序流程语句。这些语句的语法和C++极为相似。

return语句：

return (expression);

if和if…else语句：

if( condition )

{

statement(s);

}

if( condition )

{

statement(s);

}

else

{

statement(s);

}

for语句：

for(initial; condition; increment)

{

statement(s);

}

while语句：

while( condition )

{

statement(s);

}

do…while语句：

do

{

statement(s);

}while( condition );

### 16.4.3 强制转换（casting）

HLSL支持一种非常自由的强制转换设计。HLSL中强制转换的语法和C程序语言中的一样。例如要把float转换到matrix，我们写：

float f = 5.0f;

matrix m = (matrix)f;

从本书的例子中，你就能推断出这个转换语法的意思。但是，如果想要得到更详细的受支持的转换的信息，那么在DirectX SDK里，Content（内容）标签页下，看DirectX Graphics\Reference\Shader Reference\High Level Shading Language\Type就可以了。

## 16.5 操作符

HLSL支持很多类似C++的操作符。除了很少一些底下注释的例外以外，他们的用法和C++里的完全一样。下表列出了HLSL的操作符：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [] |  | > | < | < = | > = |
| ! = | = = | ! | && |  | ?: |
| + | + = | - | - = | \* | \*= |
| / | /= | % | %= | + + | -- |
| = | () | ' |  |  |  |

虽然操作符的行为和C++很相似，但是也有一些差异。第一，求模%运算符对整型和浮点型都起作用。为了使用求模操作符，左边的值和右边的值都必须有相同的正负号（如：左边和右边必须都是正或者负）。

第二，要注意HLSL操作是以每个分量为基础的。这是由于实际上向量和矩阵是语言内建的，并且这些类型是由若干个分量组成。通过将这些操作施加在分量级别之上，我们可以像使用数值类型一样完成诸如向量/矩阵的加法，减法和相等测试这些操作（），见下例：

注意：操作符的行为正如对数值操作一样（也就是说，按一般C++的方式）。

vector u = {1.0f, 0.0f, -3.0f, 1.0f};

vector v = {-4.0f, 2.0f, 1.0f, 0.0f};

// adds corresponding components

vector sum = u + v; // sum = (-3.0f, 2.0f, -2.0f, 1.0f)

增量一个向量就是增量其每个分量：

// before increment: sum = (-3.0f, 2.0f, -2.0f, 1.0f)

sum++; // after increment: sum = (-2.0f, 3.0f, -1.0f, 2.0f)

向量相乘也是按分量的：

vector u = {1.0f, 0.0f, -3.0f, 1.0f};

vector v = {-4.0f, 2.0f, 1.0f, 0.0f};

// multiply corresponding components

vector sum = u \* v; // product = (-4.0f, 0.0f, -3.0f, 0.0f)

比较操作也是按分量进行的，并且返回一个每个分量都为bool类型的向量或者数组。作为结果的“bool”向量包含了每个分量比较的结果。例如：

vector u = { 1.0f, 0.0f, -3.0f, 1.0f};

vector v = {-4.0f, 0.0f, 1.0f, 1.0f};

vector b = (u == v); // b = (false, true, false, true)

最后，我们以讨论二元操作的变量提升（promotion）作为结束：

* 对于二元操作，如果（操作符的）左边和右边维数不同，则维数较少的一边提升（强制转换）到具有和维数较大的一边相同的维数。例如，如果x的类型为float，而y的类型为float3，在表达式(x + y)中变量x被提升到float3，并且计算出来的表达式的值的类型也为float3。提升使用已定义的转换完成。注意，若转换未定义则提升也是未定义的。例如，我们不能转换float2到float3，因为没有定义这个转换。
* 对于二元操作，如果左边和右边类型不同，那么较低精度的类型（the lower type resolution）被提升（强制转换）到具有同类型的较高精度的类型（the higher type resolution）。例如，如果x类型为int，y类型为half，则表达式(x + y)中的变量x被提升到half，并且计算出来的表达式的值的类型也为half。

## 16.6 用户定义函数

HLSL中的函数有下例属性：

* 函数使用类似C++的语法
* 参数总是按值传递
* 递归不被支持
* 函数总是inline的

此外，函数还加上了一些用于其上的额外的关键字。例如，考虑一个写在HLSL中的下面这个函数：

|  |
| --- |
| bool foo(in const bool b, // input bool  out int r1, // output int  inout float r2) // input/output float  {  if( b ) // test input value  {  r1 = 5; // output a value through r1  }  else  {  r1 = 1; // output a value through r1  }  // since r2 is inout we can use it as an input  // value and also output a value through it  r2 = r2 \* r2 \* r2;  return true;  } |

函数几乎和C++函数是一样的，除了in，out和inout关键字：

* in——指定型参（argument，特指传递给实参的变量）应该在函数开始前被拷贝给实参。传入参数不必强制指定，因为实参默认是in的。例如，下面两段是等价的：

|  |
| --- |
| float square(in float x)  {  return x \* x;  } |

也可以不强制指定in：

|  |
| --- |
| float square(float x)  {  return x \* x;  } |

* out——指定实参应该在函数返回时被拷贝给型参。这样可以通过参数返回值。out关键字是必须的，因为HLSL不允许传递一个引用或一个指针。我们要注意：如果实参标记为out，在函数开始前，型参就不拷贝给实参。换句话说，out实参仅可以被用于输出数据——它不能用于输入。

|  |
| --- |
| void square(in float x, out float y)  {  y = x \* x;  } |

这里，我们输入了要被乘方的数x，并且通过参数y返回了x的乘方。

* inout——这是一个指示实参既用于输入又用于输出的快捷方法。如果要使用实参同时用作输入和输出，就指定inout。

|  |
| --- |
| void square(inout float x)  {  x = x \* x;  } |

这里，我们输入了要被乘方的数x，同时又通过x返回了的x的乘方。

## 16.7内建函数

HLSL有一个丰富的内建函数的集合，它们对3D图形来说非常有用。下表是一个删减了的列表。在下两章中，我们会使用这些函数中的一些进行实践。而现在，熟悉它们就够了。

注意：要得到更多的参考，可以参看DirectX文档中内建HLSL函数的完整列表，在Content页下，然后到DirectX Graphics\Reference\Shader Reference\High Level Shader Language\Intrinsic Functions。

译者注：以下表格中，//<variable>//表示变量variable的模（例如向量的绝对值）。

|  |  |
| --- | --- |
| 函数 | 描述 |
| abs(x) | 返回 |x| |
| ceil(x) | 返回 ≥ *x* 的最小整数 |
| clamp(x, a, b) | clamp(x, a, b) |
| clamp(x, a, b) | clamp(x, a, b) |
| cross(u, v) | 返回 u × v（叉积） |
| degrees(x) | 转换 x 从弧度到角度 |
| determinant(M) | 返回矩阵M的行列式det(M) |
| distance(u, v) | 返回u点和v点之间的距离||v - u|| |
| dot(u, v) | 返回 u · v（点积） |
| floor(x) | 返回 ≤ *x* 的最大整数 |
| length(v) | 返回 ||v|| |
| lerp(u, v, t) | 在u和v之间线性插值，根据参数 *t* ∊ [0, 1 ] |
| log(x) | 返回 ln(x) |
| log10(x) | 返回 log10(x) |
| log2(x) | 返回 log2(x) |
| max(x, y) | 如果*x ≥ y*，则返回 x；否则返回 *y* |
| min(x, y) | 如果 x *≤ y*，返回x；否则返回 *y* |
| mul(M, N) | 返回矩阵乘积 MN. 注意：矩阵乘积必须是已定义的. 如果M是一个向量，它被作为一个行向量，则向量－矩阵（vector-matrix）乘法是已定义的。类似的,如果N 是一个向量，他被作为一个列向量，则矩阵－向量（matrix-vector）乘法是已定义的 |
| normalize(v) | 返回 v/∥v∥ |
| pow(b, n) | 返回 *bn* |
| radians(x) | 转换 x 从 角度 到 弧度 |
| reflect(v, n) | 给定向量v和表面法线n，计算其反射向量 |
| refract(v,n, eta) | 给定向量v、表面法线n和两种材质的两个索引的比率eta，计算其折射向量. 翻看一下物理书中Snell的规则或者在互联网上搜索一下关于refraction（反射）的信息 |
| rsqrt(x) | 返回x的平方根的倒数 |
| saturate(x) | 返回clamp(x, 0.0, 1.0) |
| sin(x) | 返回x的正弦,其中x单位为弧度 |
| 返回x的正弦,其中x单位为弧度 | 返回x的正弦和余弦，其中x单位为弧度 |
| sqrt(x) | 返回x的平方根 |
| tan(x) | 返回x的正切,其中 x 单位为弧度 |
| transpose(M) | 返回M*T*的转置 |

大多数函数已经重载以使其可以对所有内建类型有意义。例如，abs对所有数值类型有意义，所以它为所有这些数值类型进行了重载。又例如，叉积的叉乘仅对3D向量有意义，所以它对所有类型的3D向量（比如：int，float，double的3D向量）进行了重载。另一方面，线性插值——lerp，对于数值、2D、3D和4D向量有意义，因此重载了这些类型。

注意：如果你传递进去一个非数值类型到一个（要求）数值类型的函数，也就是一个仅能对数值类型进行操作的函数（比如：cos(x)），那么这个函数会对传进去的每个分量进行操作。例如，你写：

|  |
| --- |
| floats v = float3 (0.0f, 0.0f, 0.0f);  v = cos(v); |

那么函数将会对每个分量进行操作：v=(cos(x),cos(y),cos(z))。

下例展示了这些固有的函数可能被调用的方式：

|  |
| --- |
| float x = sin(1.0f); // sine of 1.0f radian.  float y = sqrt(4.0f); // square root of 4.  vector u = {1.0f, 2.0f, -3.0f, 0.0f};  vector v = {3.0f, -1.0f, 0.0f, 2.0f};  float s = dot(u, v); // compute dot product of u and v.  float3 i = {1.0f, 0.0f, 0.0f};  float3 j = {0.0f, 1.0f, 0.0f};  float3 k = cross(i, j); // compute cross product of i and j.  matrix<float, 2, 2> M = {1.0f, 2.0f, 3.0f, 4.0f};  matrix<float, 2, 2> T = transpose(M); // compute transpose |

## 16.8 摘要

* 我们在ASCII文本文件中编写了HLSL程序，并且在我们的应用程序中使用D3DXCompileShaderFromFile函数编译了它们。
* ID3DXConstantTable接口允许我们在应用程序中对着色器程序中的变量进行设置。这种通信是必须的，因为被着色器使用的变量可以按一帧一帧的变化而改变。例如，如果应用程序中的视图矩阵发生了改变，我们需要使用新的视图矩阵更新着色器的视图矩阵变量。我们可以用ID3DXConstantTable完成这种更新。
* 对于每个着色器，我们必须定义一个输入和一个输出结构，这些结构分别描述了着色器中输入和输出数据的格式。
* 每个着色器有一个入口点函数，它有一个输入结构参数用于传递数据进着色器。此外，每个着色器返回一个输出结构的实例，它用于从着色器输出数据。

# 第十七章 顶点着色器入门 (Introduction to Vertex Shaders)

## 概览

顶点着色器（vertex shader）是一个在图形卡的GPU上执行的程序，它替换了固定功能管线（fixed function pipeline）中的变换（transformation）和光照（lighting）阶段。（这不是百分之百的正确，因为顶点着色器可以被Direct3D运行时（Direct3D runtime）以软件模拟，如果硬件不支持顶点着色器的话）。图16.1说明了管线中顶点着色器替换的部件。

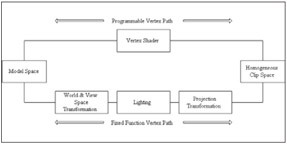


图16.1：顶点着色器替换固定功能管线的光照和变形阶段

从图16.1，我们知道，顶点以局部坐标输入到顶点着色器，并且必须输出齐次剪裁空间的有颜色的顶点。（为了保持简单，本书中我们没有深入研究投影变换的细节。但是经投影矩阵变换顶点后的空间称作齐次剪裁空间（homogeneous clip space）。因此，要把一个顶点从局部空间变换到齐次坐标空间，我们必须应用下列变换序列：世界变换（world transformation），视图变换（view transformation）和投影变换（projection transformation），它们分别由世界矩阵，视图矩阵和投影矩阵来完成。）对于点元（point primitive），顶点着色器也被用于操作每个顶点的顶点大小。

由于顶点着色器是我们（在HLSL中）写的一个自定义程序，因此我们在图形效果方面获得了我们能够达到的极大的自由性。我们不再受限于Direct3D的固定光照算法。此外，应用程序操纵顶点位置的能力也有了多样性，例如：cloth simulation，粒子系统的点大小操纵，还有顶点混合/morphing。此外，我们的顶点数据结构更自由了，并且可以在可编程管线中包含比在固定功能管线中多得多的数据。

顶点着色器仍然是相对新的特性，并且许多图形卡不支持它们，特别是随DirectX 9发布的较新版本的顶点着色器。通过检查D3DCAPS9结构的VertexShaderVersion成员，可以测试顶点着色器的版本。下列代码段展示了这一点：

|  |
| --- |
| // If the device's supported version is less than version 2.0  if( caps.VertexShaderVersion < D3DVS VERSION(2, 0) )  // Then vertex shader version 2.0 is not supported on this device. |

我们看到D3D\_VERSION的两个参数分别接收主和次版本号。现在，D3DXCompileShaderFromFile函数支持顶点着色器版本1.1和2.0。

## 目标

* 学习如何在可编程管线中定义顶点结构的分量
* 了解顶点分量的不同用法
* 学习如何创建、设置和销毁一个顶点着色器
* 学习如何使用顶点着色器实现卡通动画渲染效果

## 17.1顶点声明

到现在为止，我们已经使用自由顶点格式（flexible vertex format，FVF）来描述顶点结构中的各分量。但是，在可编程管线中，顶点数据包含的数据比用FVF所能表达的多很多。因此，我们通常使用更具表达性并且更强大的顶点声明（vertex declaration）。

注意：如果FVF能够描述我们的顶点格式 我们仍然可以在可编程管线中使用它。不管用何种方法，只是为了方便，同样FVF会在内部被转换为一个顶点声明。

### 17.1.1 描述顶点声明

我们将一个顶点声明描述为一个D3DVERTEXELEMENT9结构的数组。D3DVERTEXELEMENT9数组中的每个成员描述了一个顶点的分量。所以，如果你的顶点结构有三个分量（例如：位置、法线、颜色），那么其相应的顶点声明将描述3个D3DVERTEXELEMENT9结构的数组。这个D3DVERTEXELEMENT9结构定义如下：

|  |
| --- |
| typedef struct \_D3DVERTEXELEMENT9 {  BYTE Stream;  BYTE Offset;  BYTE Type;  BYTE Method;  BYTE Usage;  BYTE UsageIndex;  } D3DVERTEXELEMENT9; |

* Stream——指定关联到顶点分量的流
* Offset——偏移，按字节，相对于顶点结构成员的顶点分量的开始。例如，如果顶点结构是：

|  |
| --- |
| struct Vertex  {  D3DXVECTOR3 pos;  D3DXVECTOR3 normal;  }; |

……pos分量的偏移是0，因为它是第一个分量；normal分量的偏移是12，因为sizeof(pos) == 12。换句话说，normal分量以Vertex的第12个字节为开始。

* Type——指定数据类型。它可以是D3DDECLTYPE枚举类型的任意成员；完整列表请参见文档。常用类型如下：
  + D3DDECLTYPE\_FLOAT1——浮点数值
  + D3DDECLTYPE\_FLOAT2——2D浮点向量
  + D3DDECLTYPE\_FLOAT3——3D浮点向量
  + D3DDECLTYPE\_FLOAT4——4D浮点向量
  + D3DDECLTYPE\_D3DCOLOR—D3DCOLOR类型，它扩展为RGBA浮点颜色向量(r g b a)，其每一分量都是归一化到区间[0, 1]了的。
    - Method——指定网格化方法。我们认为这个参数是高级的，因此我们使用默认值，标识为D3DDECLMETHOD\_DEFAULT.。
    - Usage——指定已计划的对顶点分量的使用。例如，它是否准备用于一个位置向量、法线向量、纹理坐标等？有效的用途标识符（usage identifier）是D3DDECLUSAGE枚举类型的：

|  |
| --- |
| typedef enum \_D3DDECLUSAGE {  D3DDECLUSAGE\_POSITION = 0, // Position.  D3DDECLUSAGE\_BLENDWEIGHTS = 1, // Blending weights.  D3DDECLUSAGE\_BLENDINDICES = 2, // Blending indices.  D3DDECLUSAGE\_NORMAL = 3, // Normal vector.  D3DDECLUSAGE\_PSIZE = 4, // Vertex point size.  D3DDECLUSAGE\_TEXCOORD = 5, // Texture coordinates.  D3DDECLUSAGE\_TANGENT = 6, // Tangent vector.  D3DDECLUSAGE\_BINORMAL = 7, // Binormal vector.  D3DDECLUSAGE\_TESSFACTOR = 8, // Tessellation factor.  D3DDECLUSAGE\_POSITIONT = 9, // Transformed position.  D3DDECLUSAGE\_COLOR = 10, // Color.  D3DDECLUSAGE\_FOG = 11, // Fog blend value.  D3DDECLUSAGE\_DEPTH = 12, // Depth value.  D3DDECLUSAGE\_SAMPLE = 13 // Sampler data.  } D3DDECLUSAGE; |

D3DDECLUSAGE\_PSIZE类型用于指定一个顶点的点的大小。它用于点精灵，因此我们可以基于每个顶点控制其大小。一个D3DDECLUSAGE\_POSITION成员的顶点声明意味着这个顶点已经被变换，它通知图形卡不要把这个顶点送到顶点处理阶段（变形和光照）。

注意：这些中的少数用途类型（usage type）未在本书中提及，例如BLENDWEIGHTS, BLENDINDICES, TANGENT, BINORMAL, 和TESSFACTOR

* UsageIndex——用于标识多个相同用途的顶点分量。这个用途索引是位于区间[0, 15]间的一个整数。例如，假设我们有三个用途为D3DDECLUSAGE\_NORMAL的顶点分量。我们可以为第一个指定用途索引为0，为第二个指定用途索引为1，并且为第三个指定用途索引为2。按这种方式，我们可以通过其用途索引标识每个特定的法线。

顶点描述声明的例子：假设我们想要描述的顶点格式由位置向量和三个法线向量组成。顶点声明可以指定如下：

|  |
| --- |
| D3DVERTEXELEMENT9 decl[] =  {  {0, 0, D3DDECLTYPE\_FLOAT3, D3DDECLMETHOD\_DEFAULT,  D3DDECLUSAGE\_POSITION, 0},  {0, 12, D3DDECLTYPE\_FLOAT3, D3DDECLMETHOD\_DEFAULT,  D3DDECLUSAGE\_NORMAL, 0},  {0, 24, D3DDECLTYPE\_FLOAT3, D3DDECLMETHOD\_DEFAULT,  D3DDECLUSAGE\_NORMAL, 1},  {0, 36, D3DDECLTYPE\_FLOAT3, D3DDECLMETHOD\_DEFAULT,  D3DDECLUSAGE\_NORMAL, 2},  D3DDECL\_END()  }; |

D3DDECL\_END宏用于初始化D3DVERTEXELEMENT9数组的最后一个顶点元素。同样的，注意法线向量的用途索引标签。

### 17.1.2 创建顶点声明

一旦你描述了一个顶点声明为D3DVERTEXELEMENT9数组，我们就可以使用下面的方法获得一个IDirect3DVertexDeclaration9接口指针：

|  |
| --- |
| HRESULT IDirect3DDevice9::CreateVertexDeclaration(  CONST D3DVERTEXELEMENT9\* pVertexElements,  IDirect3DVertexDeclaration9\*\* ppDecl  ); |

* pVertexElements——D3DVERTEXELEMENT9结构数组，它描述我们想要创建的顶点声明。
* ppDecl——用于返回创建的IDirect3DVertexDeclaration9接口指针

例子调用，其中decl是一个D3DVERTEXELEMENT9数组：

|  |
| --- |
| IDirect3DVertexDeclaration9\* \_decl = 0;  hr = \_device->CreateVertexDeclaration(decl, &\_decl); |

### 17.1.3 使用一个顶点声明

回忆一下：自由顶点格式是一个方便的特性并且在内部转换成了顶点声明。因此，当直接使用顶点声明，我们不再需要调用：Device->SetFVF( fvf );

相反，我们调用：Device->SetVertexDeclaration( \_decl );

其中，\_decl是一个IDirect3DVertexDeclaration9接口指针。

## 17.2顶点数据用途

考虑这个顶点声明：

|  |
| --- |
| D3DVERTEXELEMENT9 decl[] =  {  {0, 0, D3DDECLTYPE\_FLOAT3, D3DDECLMETHOD\_DEFAULT,  D3DDECLUSAGE\_POSITION, 0},  {0, 12, D3DDECLTYPE\_FLOAT3, D3DDECLMETHOD\_DEFAULT,  D3DDECLUSAGE\_NORMAL, 0},  {0, 24, D3DDECLTYPE\_FLOAT3, D3DDECLMETHOD\_DEFAULT,  D3DDECLUSAGE\_NORMAL, 1},  {0, 36, D3DDECLTYPE\_FLOAT3, D3DDECLMETHOD\_DEFAULT,  D3DDECLUSAGE\_NORMAL, 2},  D3DDECL\_END()  }; |

我们需要一种方式，来定义一个顶点声明的元素到顶点着色器的Input结构的数据成员的映射。我们在Input结构中通过指定每个数据成员的语义（: usage-type [usage-index]）定义这个映射。语义通过元素的用途类型和用途索引标识顶点声明中的一个元素。由数据成员的语义标识的顶点元素是得以映射到数据成员的元素。例如，对应于前面的顶点声明的输入结构是：

|  |
| --- |
| struct VS\_INPUT  {  vector position : POSITION;  vector normal : NORMAL0;  vector faceNormal1 : NORMAL1;  vector faceNormal2 : NORMAL2;  }; |

注意：如果我们遗漏了用途索引，就意味着用途索引为零。例如，POSITION和POSITION0是同一样东西。

这里[decl](#_17.2顶点数据用途)中的元素0，由用途POSITION和用途索引0标识，它映射到position。decl中的元素1，由用途NORMAL和用途索引0标识，它映射到normal。decl中的元素2，由NORMAL和用途索引1标识，它映射到faceNormal1。decl中的元素3，由用途NORMAL和用途索引2标识，它映射到faceNormal2。

受支持的顶点着色器输入用途（input usage）是：

* POSITION [n]——位置
* BLENDWEIGHTS [n]——混合权重
* BLENDINDICES [n]——混合索引
* NORMAL [n]——法线向量
* PSIZE[n]——顶点大小
* DIFFUSE [n]——散射颜色
* SPECULAR [n]——镜面颜色
* TEXCOORD [n]——纹理坐标

其中，n是一个位于区间[0, 15]的可选整数。

注意：再重复一遍，这些用途类型中的少数未在本书中提及，如：BLENDWEIGHTS, TANGENT, BINORMAL, BLENDINDICES, 和TESSFACTOR。

此外，对于输出结构，我们必须指定每个成员是用来做什么的。例如，数据成员应该被作为位置向量、颜色、纹理坐标等对待吗？图形卡没主意，除非你强制的告诉它。这也需要通过语法的语义来完成：

|  |
| --- |
| struct VS\_OUTPUT  {  vector position : POSITION;  vector diffuse : COLOR0;  vector specular : COLOR1;  }; |

受支持的顶点着色器输出用途是：

* POSITION—位置
* PSIZE—顶点大小
* FOG—雾混合值
* COLOR [n]—顶点颜色。注意：可以有多个顶点颜色被输出，并且这些颜色可以被混合在一起以产生最终的颜色。
* TEXCOORD [n]—顶点纹理坐标。注意：多个顶点纹理坐标可以被输出。

其中，n是一个位于区间[0, 15]的可选整数。

## 17.3使用顶点着色器的步骤

下面的列表概括了创建和使用顶点着色器的必须步骤：

1. 编写并编译顶点着色器
2. 创建一个IDirect3DVertexShader9接口以引用已编译的着色器代码上的顶点着色器。
3. 用IDirect3DDevice9:: SetVertexShader方法使用这个顶点着色器。

当然，在我们做完这些之后，我们还得销毁这个顶点着色器。下面的各小节将更详细的迈入这些步骤。

### 17.3.1 编写并编译顶点着色器

首先，我们必须编写一个顶点着色器程序。在本书中的HLSL一章中，我们已经编写了我们的着色器（译者注：参见我翻译的译文第一章中各节）。一旦着色器代码写好之后，我们就使用D3DXCompileShaderFromFile函数编译这个着色器，如16.2.2节所述。回忆一下，这个函数返回一个ID3DXBuffer指针，它包含已编译的着色器代码。

### 17.3.2 创建顶点着色器

一旦我们拥有了编译好的着色器代码，我们就能够获得一个IDirect3DVertexShader9接口的指针，它代表一个顶点着色器——通过使用下面的方法：

|  |
| --- |
| HRESULT IDirect3DDevice9::CreateVertexShader(  const DWORD \*pFunction,  IDirect3DVertexShader9\*\* ppShader  ); |

pFunction——已编译着色器代码的指针

ppShader——返回一个IDirect3DVertexShader9接口的指针

例如，假设变量shader是一个包含已编译的，着色器代码的ID3DXBuffer指针。然后要获得一个IDirect3DVertexShader9接口，我们可以写：

|  |
| --- |
| IDirect3DVertexShader9\* ToonShader = 0;  hr = Device->CreateVertexShader(  (DWORD\*)shader->GetBufferPointer(),  &ToonShader); |

注意：重申一遍，D3DXCompileShaderFromFile是一个函数，它将返回已编译着色器的代码（shader）。

### 17.3.3 建立顶点着色器

在我们获得了一个代表我们的顶点着色器的IDirect3DVertexShader9接口的指针之后，我们就能够使用下面的方法使用它：

|  |
| --- |
| HRESULT IDirect3DDevice9::SetVertexShader(  IDirect3DVertexShader9\* pShader  ); |

这个方法仅接受一个参数，我们在其中传递一个想要使用的顶点着色器的指针。要使用这个我们在[17.3.2节](#_17.3.2_创建顶点着色器)创建的着色器，我们可以写：Device->SetVertexShader(ToonShader);

### 17.3.4 销毁顶点着色器

和所有的Direc3D接口一样，要清除他们，我们就必须在用完它们之后调用其的Release方法。仍然以我们在17.3.2节创建的顶点着色器为例，我们写：

d3d::Release<IDirect3DVertexShader9\*>(ToonShader);

## 17.4样例应用程序：散射光照

作为创建并使用顶点着色器的热身，我们写一个顶点着色器，它用一个方向（平行）光对每个顶点进行标准的散射光照。简而言之，散射光照根据顶点法线和光线向量（它的点朝向光源方向）的角度计算顶点接收到的光线的数量。角度越小，则顶点接收到的光线就越多；而角度越大，则顶点接收到的光线就越少。如果角度大于等于90度，顶点就接收不到光线了。

我们以检阅着色器代码作为开始：

|  |
| --- |
| // File: diffuse.txt  // Desc: Vertex shader that does diffuse lighting.  // Global variables we use to hold the view matrix, projection matrix,  // ambient material, diffuse material, and the light vector that  // describes the direction to the light source. These variables are  // initialized from the application.  matrix ViewMatrix;  matrix ViewProjMatrix;  vector AmbientMtrl;  vector DiffuseMtrl;  vector LightDirection;  // 环境光强度，漫射光强度  // 这些变量定义在着色器代码中  vector DiffuseLightIntensity = {0.0f, 0.0f, 1.0f, 1.0f};  vector AmbientLightIntensity = {0.0f, 0.0f, 0.2f, 1.0f};  // Input and Output structures.  struct VS\_INPUT  {  vector position : POSITION;  vector normal : NORMAL;  };  struct VS\_OUTPUT  {  vector position : POSITION;  vector diffuse : COLOR;  };  //Main  VS\_OUTPUT Main(VS\_INPUT input)  {  // zero out all members of the output instance.  VS\_OUTPUT output = (VS\_OUTPUT)0;  // 变换位置到齐次坐标空间，保存到output.position成员中  output.position = mul(input.position, ViewProjMatrix);  // 变换光和法线到视图空间，设置w分量为0，是因为变换的向量不是点  LightDirection.w = 0.0f;  input.normal.w = 0.0f;  LightDirection = mul(LightDirection, ViewMatrix);  input.normal = mul(input.normal, ViewMatrix);  // 计算光与法线夹角的余弦  float s = dot(LightDirection, input.normal);  // 回忆一下，如果法线和光的夹角大于90度，则表面接收不到光。  if( s < 0.0f )  s = 0.0f;  // 环境光反射是执行一个叉积（环境材质向量与环境光强度向量），  // 漫射光反射是执行一个叉积（漫射材质向量与漫射光强度向量，  // 更进一步讲，我们测量着色器的颜色，基于顶点从光源处接收到多少光  //环境光和漫射光综合起来，决定一个顶点的最终颜色  output.diffuse = (AmbientMtrl \* AmbientLightIntensity) +  ((DiffuseMtrl \* DiffuseLightIntensity) \* s);  return output;  } |

既然我们已经看到了实际的顶点着色器的代码，那么就让我们改变方式来看看应用程序的代码。这个应用程序有下列相关的全局变量：

|  |
| --- |
| IDirect3DVertexShader9\* DiffuseShader = 0;  ID3DXConstantTable\* DiffuseConstTable = 0;  ID3DXMesh\* Teapot = 0;  D3DXHANDLE ViewMatrixHandle = 0;  D3DXHANDLE ViewProjMatrixHandle = 0;  D3DXHANDLE AmbientMtrlHandle = 0;  D3DXHANDLE DiffuseMtrlHandle = 0;  D3DXHANDLE LightDirHandle = 0;  D3DXMATRIX Proj; |

有代表顶点着色器及其常量表的变量，有茶壶网格的变量，接着是一组D3DXHANDLE，其名字描述了他们引用的变量：

Setup函数执行下列任务：

* 创建茶壶网格
* 编译顶点着色器
* 根据已编译代码创建顶点着色器
* 通过常量表获取着色器程序中的几个变量的句柄
* 通过常量表初始化着色器的这几个变量

注意：对于本应用程序，我们的顶点结构不需要任何自由顶点格式没有的额外的分量。因此，在本例中，我们使用一个自由顶点格式来代替顶点声明。回想一下，自由顶点格式描述最终在内部被转换为一个顶点声明。

|  |
| --- |
| bool Setup()  {  HRESULT hr = 0;  // Create geometry:  D3DXCreateTeapot(Device, &Teapot, 0);  // Compile shader  ID3DXBuffer\* shader = 0;  ID3DXBuffer\* errorBuffer = 0;  hr = D3DXCompileShaderFromFile(  "diffuse.txt",  0,  0,  "Main", // entry point function name  "vs\_1\_1",  D3DXSHADER\_DEBUG,  &shader,  &errorBuffer,  &DiffuseConstTable);  // output any error messages  if( errorBuffer )  {  ::MessageBox(0, (char\*)errorBuffer->GetBufferPointer(), 0, 0);  d3d::Release<ID3DXBuffer\*>(errorBuffer);  }  if(FAILED(hr))  {  ::MessageBox(0, "D3DXCompileShaderFromFile() - FAILED", 0, 0);  return false;  }  // Create shader  hr = Device->CreateVertexShader(  (DWORD\*)shader->GetBufferPointer(),  &DiffuseShader);  if(FAILED(hr))  {  ::MessageBox(0, "CreateVertexShader - FAILED", 0, 0);  return false;  }  d3d::Release<ID3DXBuffer\*>(shader);  // Get Handles  LightDirHandle = DiffuseConstTable->GetConstantByName(0, "LightDirection");  ViewMatrixHandle = DiffuseConstTable->GetConstantByName(0, "ViewMatrix");  ViewProjMatrixHandle = DiffuseConstTable->GetConstantByName(0, "ViewProjMatrix");  AmbientMtrlHandle = DiffuseConstTable->GetConstantByName(0, "AmbientMtrl");  DiffuseMtrlHandle = DiffuseConstTable->GetConstantByName(0, "DiffuseMtrl");    // Set shader constants:  // Light direction:  D3DXVECTOR4 directionToLight(-0.57f, 0.57f, -0.57f, 0.0f);  DiffuseConstTable->SetVector(Device, LightDirHandle, &directionToLight);  // Materials:  D3DXVECTOR4 ambientMtrl(0.0f, 0.0f, 1.0f, 1.0f);  D3DXVECTOR4 diffuseMtrl(0.0f, 0.0f, 1.0f, 1.0f);  DiffuseConstTable->SetVector(Device,AmbientMtrlHandle,&ambientMtrl);  DiffuseConstTable->SetVector(Device,DiffuseMtrlHandle,&diffuseMtrl);  DiffuseConstTable->SetDefaults(Device);  // Compute projection matrix.  D3DXMatrixPerspectiveFovLH(  &Proj, D3DX PI \* 0.25f,  (float)Width / (float)Height, 1.0f, 1000.0f);  return true;  } |

Display函数非常简单。它检测用户输入（译者注：这里指的是用户输入的传入着色器程序的变量），并相应的更新视图矩阵。但是，因为我们在着色器中执行这个视图矩阵变换，所以我们还必须更新着色器中的视图矩阵变量。我们用常量表完成这件事情。

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  // Update view matrix code snipped...  D3DXMATRIX V;  D3DXMatrixLookAtLH(&V, &position, &target, &up);  DiffuseConstTable->SetMatrix(Device, ViewMatrixHandle, &V);  D3DXMATRIX ViewProj =V \*Proj;  DiffuseConstTable->SetMatrix(Device, ViewProjMatrixHandle,  &ViewProj);  // Render  Device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER,  0xffffffff, 1.0f, 0);  Device->BeginScene();  Device->SetVertexShader(DiffuseShader);  Teapot->DrawSubset(0);  Device->EndScene();  Device->Present(0, 0, 0, 0);  }  return true;  } |

同样注意，就在DrawSubset调用之前，我们允许了这个我们希望使用的顶点着色器。

清理也需要被完成；我们简单的释放了这个已分配的接口：

|  |
| --- |
| void Cleanup()  {  d3d::Release<ID3DXMesh\*>(Teapot);  d3d::Release<IDirect3DVertexShader9\*>(DiffuseShader);  d3d::Release<ID3DXConstantTable\*>(DiffuseConstTable);  } |

## 17.5 卡通渲染

作为第二个顶点着色器的例子，让我们编写两个顶点着色器，它们以卡通风格的绘画方式对网格着色（shade）并画轮廓（outline）。图17.2展示了这一点：

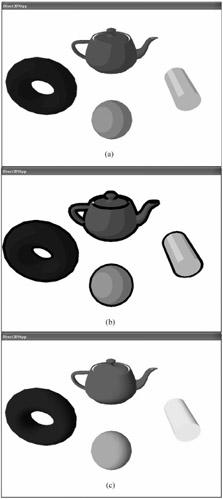


图17.2：（a）使用卡通着色法着色的对象（注意着色间的尖锐过渡）。（b）增强卡通效果，轮廓边（silhouette edge）被勾出。（c）使用标准散射光照着色的对象

注意：卡通渲染是一种特定类型的非写实渲染（non-photorealistic rendering），有时被称作风格化渲染（stylistic rendering）。

虽然卡通渲染不适用于所有游戏，例如激烈的第一人称射击游戏，但是它仍然可以增强一些希望表现卡通感觉类型游戏的气氛。此外，卡通渲染是漂亮的，并易于实现。让我们好好的演示一个顶点着色器。

我们将卡通渲染分为两步：

1. 卡通绘画的特点是：在一个顶点到下一个顶点的强烈转换时，有少量的阴影强度级别；我们看一下这个卡通阴影（cartoon shading）。在图17.2（a）中，我们看到网络着色使用了三种阴影强度（亮、中、暗），而且其间的过渡是不平滑的——不像图17.2（c），其明暗过渡是平滑的。
2. 卡通绘图的主要特点是：在其外框上勾画轮廓，如图17.2（b）所示。

这两个步骤都需要其各自的顶点着色器。

### 17.5.1 卡通着色

要实现卡通着色，我们采用Lander在2000年3月发表在Game Developer Magazine的文章“Shades of Disney: Opaquing a 3D World”中所描述的方法。它像这样工作：我们创建一个带强度级别的灰度纹理，它包含我们需要的不同的着色强度。图17.3显示了我们在样例程序中使用的这个纹理。

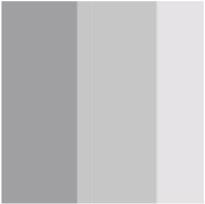


图 17.3：用来保存着色强度的着色纹理。注意观察不连续的着色间过渡和纹理着色强度必须从左到右增加。

然后在顶点着色器中，我们执行标准散射点积运算（standard diffuse calculation dot product）来确定顶点法线N和光线向量L之间角度的余弦，用以确定顶点接收到多少光线：s=L·N

如果s＜0，就表示光线向量和顶点法线之间的角度大于90度，也就表示该表面接收不到光线。因此，如果s＜0，我们就让s＝0。所以s ∈ [0, 1]。

现在，在通常的散射光照模型中，我们使用s来标记颜色向量。这样，顶点颜色的明暗取决于接收到的光照的数量：diffuseColor = s(r, g, b, a)

但是，这将会导致从亮到暗之间平滑的着色。这是与我们期望的卡通着色相反的。我们想要一种在几个不同着色器间突然转换颜色的效果（对卡通渲染来说，在2至4种着色器工作起来还是挺不错的）。

不使用s来标记颜色向量，我们将使用s作为早先提到的强度纹理的u纹理坐标——如图17.3。

注意：标量（scalar）s必定是一个有效的纹理坐标，因为s ∈ [0, 1]，这是通常的纹理坐标区间。

按这种方式，顶点不会被平滑着色，而是间断的。例如，强度纹理可能被分成3种着色，如图17.4所示：

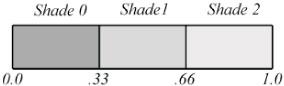


图17.4：那么，s ∈ [0, 0.33]的值使用shader0着色，s ∈ [ 0.33，0.66]的值使用shader1着色，s ∈ [0.66,1]的值使用shader2着色。当然，从这些着色的一种到另一种的过渡是不平滑的，这就赋予了我们期望的效果。

注意：我们还为卡通着色关闭了纹理过滤，因为这种过滤会试图使着色过渡变平滑。这对于我们要求的不连续过渡是多余的。

### 17.5.2 卡通着色的顶点着色器代码

我们现在介绍卡通着色的顶点着色器。这个着色器的主要任务只是根据s=L·N计算并设置纹理坐标。注意观察输出结构，我们已经增加了一个数据成员来存储已被计算过的纹理坐标。同时还需注意，我们仍然输出顶点颜色，虽然我们不修改它，不过当颜色被与强度纹理组合起来的时候，它呈现为被着色的。

|  |
| --- |
| // File: toon.txt  // Desc: Vertex shader that lights geometry so it appears to be  // drawn in a cartoon style.  // Globals  extern matrix WorldViewMatrix;  extern matrix WorldViewProjMatrix;  extern vector Color;  extern vector LightDirection;  static vector Black = {0.0f, 0.0f, 0.0f, 0.0f};  // Structures  struct VS\_INPUT  {  vector position : POSITION;  vector normal : NORMAL;  };  struct VS\_OUTPUT  {  vector position : POSITION;  float2 uvCoords : TEXCOORD;  vector diffuse : COLOR;  };  // Main  VS\_OUTPUT Main(VS\_INPUT input)  {  // zero out each member in output  VS\_OUTPUT output = (VS\_OUTPUT)0;  // transform vertex position to homogenous clip space  output.position = mul(input.position, WorldViewProjMatrix);  // Transform lights and normals to view space. Set w  // components to zero since we're transforming vectors.  // Assume there are no scalings in the world  // matrix as well.  LightDirection.w = 0.0f;  input.normal.w = 0.0f;  LightDirection = mul(LightDirection, WorldViewMatrix);  input.normal = mul(input.normal, WorldViewMatrix);  // Compute the 1D texture coordinate for toon rendering.  float u = dot(LightDirection, input.normal);  // Clamp to zero if u is negative because u  // negative implies the angle between the light  // and normal is greater than 90 degrees. And  // if that is true then the surface receives no light.  if(u < 0.0f)  u = 0.0f;  // Set other tex coord to middle.  float v = 0.5f;  output.uvCoords.x = u;  output.uvCoords.y = v;  // save color  output.diffuse = Color;  return output;  } |

两点注解：

* 我们假设世界矩阵没有执行任何缩放。因为如果它执行，它就会弄乱乘以它的顶点的长度和方向。
* 我们总是设置v纹理坐标为纹理的中点。这意味着我们仅使用纹理中一条单一的线，那就是说我们可以使用1D强度纹理来代替2D的那个纹理。不管怎样，1D和2D纹理都能工作。本例中，我们使用了2D纹理而不是1D纹理，这是没有什么特别的原因的。

### 17.5.3轮廓勾勒

要完成卡通效果，我们还需要勾勒（outline）轮廓边（silhouette edge）。这比卡通着色稍微复杂一点。

#### 17.5.3.1 边的表示法

我们将一个网格的一条边表示为一个四元组（构建自2个三角形）——参见图17.5。

CSDN_Dev_Image_2004-5-132029552

图 17.5：表示边的四元组

我们选择四元组有两个原因：我们可以通过调整四元组的维容易的改变边的厚度，并且我们可以渲染退化的四元组来隐藏某些边，也即非轮廓边。在Direct3D中，我们从两个三角形来构建一个四元组。退化四元组（degenerate quad）是从两个退化三角形构建而来的四元组。退化三角形（degenerate triangle）是一个面积为零的三角形，或者换句话说，是一个三点位于一线上的三角形。如果我们传入一个退化三角形到渲染管线，则该三角形显示为空。这是很有用的，因为如果我们希望隐藏特定三角形，我们可以简单的退化它而不需要实际的从三角形列表（顶点缓冲）移除它。回想一下，我们只需要显示轮廓边——而不是网格的每一条边。

当我们首先创建一条边的时候，我们指定其四个顶点，并使其退化，这意味着边将会被隐藏（渲染时不显示）。

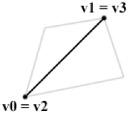


图17.6：由两个三角形共用边描述的退化四元组

注意图17.6中的两个顶点v0和v1，我们设置其顶点法线向量为零向量。然后当我们将边的顶点送入顶点着色器的时候，顶点着色器将会检测顶点是否位于轮廓边上；如果是，则顶点着色器将按顶点法线的方向偏移顶点位置的标量。观察法线向量为零的顶点，它不会被偏移。

因此，我们最终以一个非退化四元组（non-degenerate quad）来表示轮廓边，如图17.7所示。

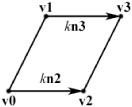


图17.7：位于轮廓边上的顶点v2和v3被按照其各自的顶点法线n2和n3进行偏移。观察顶点v0和v1仍然保持在其固定位置，因为其顶点法线等于零向量，因此对于它们来说没有偏移发生。按这种方式，四元组成功的重新生成来表示轮廓边。

备注：如果我们没有设置顶点v0和v1的顶点法线为零向量，那么那些顶点就同样会被偏移。但是如果偏移描述轮廓边的所有四个顶点，那么我们仅是平移了该退化四元组。通过保持顶点v0和v1固定并仅仅偏移顶点v2和v3，我们重新生成了四元组。

#### 17.5.3.2 轮廓边测试

若两个三角面face0和face1在视图方向上与两个不同方向的面共享同一条边，则该边为轮廓边。也就是说，如果一个面是前面（front facing）而另一个面是后面（back facing），那么这条边就是一条轮廓边。图17.8给出了一个轮廓边和一个非轮廓边的例子。

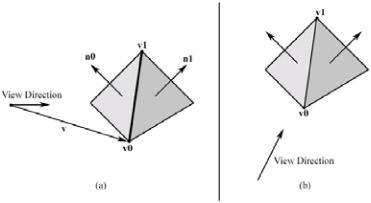


图17.8：在（a）中，由v0 和v1定义的共享边的一个面是前面，而共享边另一个面是背面，因此该边是轮廓边。在（b）中，由v0 和v1定义的这两个共享边面都是前面，因此该边不是轮廓边。

接下来，为了检测一个顶点是否在轮廓边上，我们必须以每个顶点为基础了解face0 和 face1的法线向量。我们的边的顶点数据结构反映如下：

|  |
| --- |
| struct VS\_INPUT  {  vector position : POSITION;  vector normal : NORMAL0;  vector faceNormal1 : NORMAL1;  vector faceNormal2 : NORMAL2;  }; |

前两个分量很直接，但让我们看看两个额外的法线向量，它们是faceNormal1和faceNormal2。这些向量描述了两个三角面的面法线，共享边的顶点位于这两个面的共享边上，这两个面是face0和face1。

实际检测顶点是否在共享边上的数学如下。假设我们在视图空间中，令v为一原点指向检测顶点的向量——图17.8，令n0为face0的面法线且n1为face0的面法线，若下面的不等式为真，则顶点位于轮廓边上：

（1）（v·n0）（v·n1）＜0

若两点积符号相异，则不等式为真，使得不等式左边为负。回想一下点积的性质：两个点积的符号相异，这意味着一个三角面是前面而另一个是后面。

现在，考虑一条边只有一个三角形共享它的情况，如图17.9，其法线将会被存储在faceNormal1中。

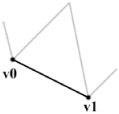


图 17.9：顶点v0和v1定义的边只有一个三角面共享它

我们定义这种边总为轮廓边。要确保顶点着色器将这种边作为轮廓边处理，我们要让faceNormal2 = -faceNormal1。因此，反向的面法线和不等式（1）为真，表示该边为一轮廓边。

#### 17.5.3.3 边的生成

生成网格的边是微不足道的；我们简单的遍历网格的每个三角面并为三角面上每条边计算一个四元组（退化的，如图17.6所示）。

注意：每个三角面有三条边，因为每个三角形有三条边。

对于每条边上的顶点，我们同样需要知道共享边的两个三角面。一个面是边所在的三角形。例如，如果要计算第1个面的一条边，那么第1个面共享该边。共享该边的另一个面可以使用网格的邻接信息找到。

### 17.5.4 轮廓边顶点着色器代码

我们现在呈现渲染轮廓边的顶点着色器代码。这个着色器的主要任务就是确定传入的顶点是否在轮廓边上。如果是，顶点着色器就以一定的值，按顶点法线的方向偏移顶点。

|  |
| --- |
| // File: outline.txt  // Desc: Vertex shader renders silhouette edges.  // Globals  extern matrix WorldViewMatrix;  extern matrix ProjMatrix;  static vector Black = {0.0f, 0.0f, 0.0f, 0.0f};  // Structures  struct VS\_INPUT  {  vector position : POSITION;  vector normal : NORMAL0;  vector faceNormal1 : NORMAL1;  vector faceNormal2 : NORMAL2;  };  struct VS\_OUTPUT  {  vector position : POSITION;  vector diffuse : COLOR;  };  // Main  VS\_OUTPUT Main(VS\_INPUT input)  {  // zero out each member in output  VS\_OUTPUT output = (VS\_OUTPUT)0;  // transform position to view space  input.position = mul(input.position, WorldViewMatrix);  // Compute a vector in the direction of the vertex  // from the eye. Recall the eye is at the origin  // in view space - eye is just camera position.  vector eyeToVertex = input.position;  // transform normals to view space. Set w  // components to zero since we're transforming vectors.  // Assume there are no scalings in the world  // matrix as well.  input.normal.w = 0.0f;  input.faceNormal1.w = 0.0f;  input.faceNormal2.w = 0.0f;  input.normal = mul(input.normal, WorldViewMatrix);  input.faceNormal1 = mul(input.faceNormal1, WorldViewMatrix);  input.faceNormal2 = mul(input.faceNormal2, WorldViewMatrix);  // compute the cosine of the angles between  // the eyeToVertex vector and the face normals.  float dot0 = dot(eyeToVertex, input.faceNormal1);  float dot1 = dot(eyeToVertex, input.faceNormal2);  // if cosines are different signs (positive/negative)  // then we are on a silhouette edge. Do the signs  // differ?  if( (dot0 \* dot1) < 0.0f )  {  // yes, then this vertex is on a silhouette edge,  // offset the vertex position by some scalar in the  // direction of the vertex normal.  input.position += 0.1f \* input.normal;  }  // transform to homogeneous clip space  output.position = mul(input.position, ProjMatrix);  // set outline color  output.diffuse = Black;  return output;  } |

## 17.6 摘要

* 使用顶点着色器，我们可以替换固定功能管线的变换和光照阶段。通过用我们自己的程序（顶点着色器）替换此固定处理，我们可以在图形效果方面获得我们能够达到的极大的自由性。
* 顶点声明用于描述顶点格式。它们和自由顶点格式相似，但是更加自由并允许我们描述FVF不能描述的顶点格式。注意，如果顶点可以用FVF描述，我们仍然可以使用它们；不管怎样，在内部它们被转换为顶点声明。
* 对于输入，用途语义指定了顶点分量如何被从顶点声明映射到HLSL程序中的变量。对于输出，用途语义指定了顶点分量是用来做什么的（例如：位置、颜色、纹理坐标，等等）。

# 第十八章 像素着色器入门 (Introduction to Pixel Shaders)

像素着色器是一个执行在图形卡的GPU上的程序，它运行在对每个像素进行光栅化处理时。（不像顶点着色器，Direct3D不会以软件模拟像素着色器的功能。）它实际上替换了固定功能管线的多纹理化阶段（the multitexturing stage），并赋予我们直接操纵单独的像素和访问每个像素的纹理坐标的能力。这种对像素和纹理坐标的直接访问使我们可以达成各种特效，例如：多纹理化（multitexturing）、每像素光照（per pixel lighting）、景深（depth of field）、云状物模拟（cloud simulation）、焰火模拟（fire simulation）、高级阴影技术（sophisticated shadowing technique）。

图形卡支持的像素着色器的版本可以通过D3DCAPS9结构的PixelShaderVersion成员和D3DPS\_VERSION宏进行检查。下列代码片断展示了这点：

|  |
| --- |
| // If the device's supported version is less than version 2.0  if( caps.PixelShaderVersion < D3DPS\_VERSION(2, 0) )  // Then pixel shader version 2.0 is not supported on this device. |

## 目标

* 获得对多纹理化概念的基本理解
* 学习如何编写、创建并使用像素着色器
* 学习如何使用像素着色器实现多纹理化效果

## 18.1多纹理化概览

多纹理化（Multitexturing）可能是用像素着色器实现的最简单的技巧了。此外，因为像素着色器替换多纹理化阶段，那么接下来我们应该对多纹理化“是什么”和“做什么”有一个最基本的理解。本节介绍多纹理化的简明概览。

当我们一开始讨论纹理化（texturing）的时候（第6章），我们忽略了固定功能管线中对多纹理化的讨论，这有两个原因：第一，多纹理化是有一点棘手的过程，我们考虑到这在当时是一个高级话题；此外，固定功能多纹理化阶段被新的和更强有力的像素着色器替换掉了。因此花时间在已经过时的固定功能纹理化阶段上是无意义的。

多纹理化后面的概念有一点和混合（blending）相关。在第七章中我们了解到：可以将正要被光栅化的像素与之前写入后台缓冲的像素进行混合来达成一种特效。我们延伸这种相同的思想到多纹理化中（multiple texture）。也就是说，我们一次使用几个纹理，然后定义这些纹理如何被混合在一起，以达到一种特殊效果。多纹理化的一个通常的用法是执行光照。作为在顶点处理阶段使用Direct3D的光照模型的替代，我们使用一种叫做“光照图”（light map）的特殊纹理贴图（texture map），它编码（encode）表面是如何被光照的。例如，假设我们希望一盏聚光灯（spotlight）照在一个大木箱上，我们要么可以定义一个D3DLIGHT9结构的聚光灯，要么可以将代表木箱的纹理贴图与代表聚光灯的光照映射混合在一起，如图18.1所示。

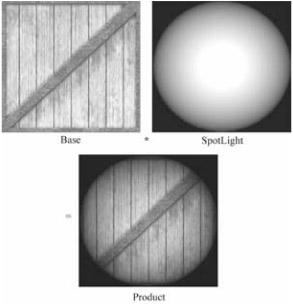


图18.1：使用多纹理化渲染一个通过聚光灯照亮的木箱。这里我们通过将相应的纹理像素（texels）相乘来将这两个纹理组合起来。

注意：用第七章里的混合，结果图像依赖于纹理被混合的方式。在固定功能管线的多纹理化阶段，混合方程式被纹理渲染状态（texture render state）控制。用像素着色器，我们 能够以可编程的方式在代码中写出混合函数的简单表达式。这使我们可以用任何我们想要的方式混合纹理。我们将在讨论本章的例子程序时详细讨论纹理混合。

混合多个纹理（本例中是两个）来照亮木箱比起Direct3D的光照来有两个好处：

* 光照是是预先在聚光灯的光照贴图里计算好的。因此，光照不需要在运行时被计算，这节省了处理时间。当然，只有静态对象和静态灯光的光照可以被预先计算。
* 因为光照图是预先计算好的，我们能够使用比Direct3D的（光照）模型多的多的更加精确的和成熟的光照模型。（更好的光照可以产生更真实的场景。）

备注：多纹理化阶段的典型应用是实现静态对象的完全光照引擎（full lighting engine）。例如，我们可以用一个纹理贴图保存对象的颜色，比如木箱的纹理贴图。然后我们可以用一个散射光照贴图（diffuse light map）保存散射表面着色（diffuse surface shade），一个单独的镜面光照贴图保存镜面表面着色，一个雾状物贴图（fog map）保存覆盖在表面的雾状物的总量，还有可以用一个详细贴图（detail map）保存小的、高访问率的表面的细节。当所有这些纹理被组合起来，只需到这些预先计算的纹理中检索，就可以有效的照亮、着色并且增加细节到场景中去。

注意：聚光灯光照贴图在很基础的光照贴图中是一个价值不高（trivial）的例子。一般的的程序通过给定的场景和光源来生成光照贴图。生成光照贴图超越了本书的范围。有兴趣的读者可以参考Alan Watt和Fabio Policarpo在《3D Games: Real-time Rendering and Software Technology》中描述的光照贴图。

### 18.1.1 允许多个纹理

回忆一下，纹理是用IDirect3DDevice9::SetTexture方法设置，而采样器状态（sampler state）是用IDirect3DDevice9::SetSamplerState方法设置，原型如下：

|  |
| --- |
| HRESULT IDirect3DDevice9::SetTexture(  DWORD Stage, // specifies the texture stage index  IDirect3DBaseTexture9 \*pTexture  );  HRESULT IDirect3DDevice9::SetSamplerState(  DWORD Sampler, // specifies the sampler stage index  D3DSAMPLERSTATETYPE Type,  DWORD Value  ); |

注意：一个特定的采样器阶段索引I联合第i个纹理阶段（texture stage）。即第i个采样器阶段指定采样器状态是第i集（set）纹理。

纹理/采样器阶段索引标识了我们希望设置的纹理/采样器的纹理/采样器阶段。因此，我们可以允许多个纹理并通过使用不同的阶段索引设置其相应的采样器状态。在本书前面的部分中，我们总是指定0，来指示第一个阶段，因为我们一次仅使用一个纹理。所以例如，假设我们要允许三个纹理，我们像这样使用阶段0,1和2：

|  |
| --- |
| // Set first texture and corresponding sampler states.  Device->SetTexture(0, Tex1);  Device->SetSamplerState(0, D3DSAMP\_MAGFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(0, D3DSAMP\_MINFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(0, D3DSAMP\_MIPFILTER, D3DTEXF\_LINEAR);  // Set second texture and corresponding sampler states.  Device->SetTexture(1, Tex2);  Device->SetSamplerState(1, D3DSAMP\_MAGFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(1, D3DSAMP\_MINFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(1, D3DSAMP\_MIPFILTER, D3DTEXF\_LINEAR);  // Set third texture and corresponding sampler states.  Device->SetTexture(2, Tex3);  Device->SetSamplerState(2, D3DSAMP\_MAGFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(2, D3DSAMP\_MINFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(2, D3DSAMP\_MIPFILTER, D3DTEXF\_LINEAR); |

这段代码使用Tex1, Tex2和Tex3，并设置每个纹理的过滤模式。

### 18.1.2 多纹理坐标

回忆一下第六章，对于每个3D三角形，我们应该在纹理上定义一个三角形以映射该3D三角形。我们通过对每个顶点增加纹理坐标完成映射。因此，每三个顶点定义一个三角形，它对应于纹理上的三角形。

因为我们现在使用多纹理，每三个顶点定义一个三角形，我们需要在每个被使用的纹理上定义一个相应的三角形。我们通过给每个顶点增加额外的一套纹理坐标——每个顶点一套，对应于每个使用的纹理。举个例子，如果我们混合三个纹理到一起，那么每个顶点必须有三套纹理坐标以索引到三个使用的纹理。因此，一个包含三个纹理的多纹理化顶点结构看起来可能像这样：

|  |
| --- |
| struct MultiTexVertex  {  MultiTexVertex(float x, float y, float z,  float u0, float v0,  float u1, float v1,  float u2, float v2)  {  \_x = x; \_y = y; \_z = z;  \_u0 = u0; \_v0 = v0;  \_u1 = u1; \_v1 = v1;  \_u2 = u2; \_v2 = v2;  }  float \_x, \_y, \_z;  float \_u0, \_v0; // Texture coordinates for texture at stage 0.  float \_u1, \_v1; // Texture coordinates for texture at stage 1.  float \_u2, \_v2; // Texture coordinates for texture at stage 2.  static const DWORD FVF;  };  const DWORD MultiTexVertex::FVF = D3DFVF\_XYZ | D3DFVF\_TEX3; |

注意，指定自由顶点格式标记D3DFVF\_TEX3表明顶点结构包含3套纹理坐标。固定功能管线支持最多8套纹理坐标。如果多于8套，你必须使用顶点声明和可编程顶点管线。

注意：在新版本像素着色器中，我们可以使用一套纹理坐标集来索引多个纹理，并因此消除了对多个纹理坐标的需要。当然这得假设每个纹理阶段使用相同的纹理坐标。如果每个阶段的纹理坐标不同，则我们仍然需要多纹理坐标。

## 18.2像素着色器输入和输出

有两样东西要输入到像素着色器：颜色和纹理坐标。两样都是以每像素为单位的。

注意：回想一下，顶点颜色是在图元的面（face of primitive）间进行插值的。

每个像素的纹理坐标就是简单的 (u , v) ，它指定了纹理的哪个图素被映射到像素上。在输入到像素着色器前，Direct3D根据顶点颜色和顶点纹理坐标，为每个像素计算颜色和纹理坐标。输入到像素着色器的颜色和纹理坐标的数值依赖于顶点着色器输出的颜色和纹理坐标的数值。例如，如果一个顶点着色器输出了两个颜色和三个纹理坐标，那么Direct3D将会为每个像素计算两个颜色和三个纹理坐标并且把它们把它们输入到像素着色器。我们使用带语意的语法（semantic syntax）映射输入颜色和纹理坐标进我们的着色器程序的变量里。用前面的例子，我们可以这样写：

|  |
| --- |
| struct PS\_INPUT  {  vector c0 : COLOR0;  vector c1 : COLOR1;  float2 t0 : TEXCOORD0;  float2 t1 : TEXCOORD1;  float2 t2 : TEXCOORD2;  }; |

对于输出，像素着色器只输出一个计算过的该像素的颜色值：

|  |
| --- |
| struct PS\_OUTPUT  {  vector finalPixelColor : COLOR0;  }; |

## 18.3使用像素着色器的步骤

下面的列表概述了创建和使用像素着色器的必要步骤：

1. 编写并编译像素着色器
2. 创建一个IDirect3DPixelShader9接口来代表基于已编译代码的像素着色器
3. 用IDirect3DDevice9::SetPixelShader方法允许该像素着色器

当然，用完顶点着色器之后我们必须销毁它。下面几个小节将深入这些步骤。

### 18.3.1 编写并编译像素着色器

我们用与编译顶点着色器一样的方式编译像素着色器。首先，我们必须编写一个像素着色器程序。本书中，我们用HLSL编写我们的着色器。一旦写好着色器代码，我们就可以用D3DXCompileShaderFromFile函数编译该着色器了，如16.2节所述。回忆一下，这个函数返回一个ID3DXBuffer指针，它包含已编译的着色器代码。

注意：因为我们使用的是像素着色器，所以要记得把编译目标改成像素着色器目标（比如：ps\_2\_0），而不是顶点着色器目标（比如：vs\_2\_0）。编译目标通过D3DXCompileShaderFromFile函数的一个参数指定。详见16.2节。

### 18.3.2 创建像素着色器

一旦我们编译了着色器代码，我们就可以获得一个IDirect3DPixelShader的接口指针，它代表一个像素着色器，使用下面的方法：

|  |
| --- |
| HRESULT IDirect3DDevice9::CreatePixelShader(  CONST DWORD \*pFunction,  IDirect3DPixelShader9\*\* ppShader  ); |

pFunction——已编译着色器代码的指针

ppShader——返回一个IDirect3DPixelShader9接口的指针

例如，假设变量shader是一个包含已编译着色器代码的ID3DXBuffer接口指针。那么要获得IDirect3DPixelShader9接口，我们应该写：

|  |
| --- |
| IDirect3DPixelShader9\* MultiTexPS = 0;  hr = Device->CreatePixelShader(  (DWORD\*)shader->GetBufferPointer(),  &MultiTexPS); |

注意：重申一遍，D3DXCompileShaderFromFile是一个可以返回已编译着色器代码（shader）的函数。

### 18.3.3 建立像素着色器

在我们获得一个代表我们的像素着色器的IDirect3DPixelShader9接口的指针之后，我们可以使用下面的方法使用它：

|  |
| --- |
| HRESULT IDirect3DDevice9::SetPixelShader(  IDirect3DPixelShader9\* pShader  ); |

这个方法只接受一个参数，我们通过它传递一个我们希望使用的指向像素着色器的指针。要使用我们在18.3.2节创建的像素着色器，我们应该写：

|  |
| --- |
| Device->SetPixelShader(MultiTexPS); |

### 18.3.4 销毁像素着色器

和其它所有Direct3D接口一样，要清除这些接口，我们必须在使用完毕后调用它们的Release方法。继续使用我们在18.3.2节创建的像素着色器，我们写：

|  |
| --- |
| d3d::Release<IDirect3DPixelShader9\*>(MultiTexPS); |

## 18.4 HLSL采样器对象

在像素着色器中使用HLSL的内建函数tex\*XXXX给纹理采样。

注意：采样时引用纹理上图素的坐标索引和采样器状态来生成像素。

看[16.7节](#_16.7内建函数)详细地解释了这些函数，通常这些函数需要我们做2件事：

* 使用纹理中的索引建立(u, v)纹理坐标。
* 给特定的纹理中编入索引。

将纹理坐标（u, v）输入到像素着色器，在一个指定的HLSL对象中的像素着色器中，我们想编入索引的纹理是在像素着色器中被定义过的，在HLSL中叫作采样器。（The particular texture that we want to index into is identified in the pixel shader by a special HLSL object called a sampler.），我们可以把采样器对象想象成定义纹理和采样器阶段的对象。例如：假如我们使用3张纹理，这意味着我们需要在像素着色器里能够引用3个阶段中的每个一个。在像素着色器中我们这样写：

|  |
| --- |
| sampler FirstTex;  sampler SecondTex;  sampler ThirdTex; |

Direct3D将给每个采样器对象连接一个唯一的纹理级别(stage)，在应用程序中我们找出与采样器对象相关联的阶段，并设置相应的纹理和采样器状态给该阶段。下列代码将举例说明如何在应用程序中设置纹理并把采样器状态设置为FirstTex：

|  |
| --- |
| // 创建  IDirect3DTexture9\* Tex;  D3DXCreateTextureFromFile(Device, "tex.bmp", &Tex);  … …  // 取得常量FirstTex的句柄  FirstTexHandle = MultiTexCT->GetConstantByName(0, "FirstTex");  // 取得常量的描述  D3DXCONSTANT\_DESC FirstTexDesc;  UINT count;  MultiTexCT->GetConstantDesc(FirstTexHandle, &FirstTexDesc, &count);  … …  // 为FirstTex设置纹理和采样器状态. We identify  // the stage FirstTex is associated with from the  // D3DXCONSTANT\_DESC::RegisterIndex member:  Device->SetTexture(FirstTexDesc.RegisterIndex, Tex);  Device->SetSamplerState(FirstTexDesc.RegisterIndex,  D3DSAMP\_MAGFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(FirstTexDesc.RegisterIndex,  D3DSAMP\_MINFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(FirstTexDesc.RegisterIndex,  D3DSAMP\_MIPFILTER, D3DTEXF\_LINEAR); |

注意：作为选择，替换使用采样器类型，你可以使用更多特殊的、强大的类型，如：sampler1D，sampler2D，sampler3D，和samplerCube类型，这些类型更安全并且它们只使用tex\*函数。例如：一个sampler2D对象只使用tex2D\*函数，同样一个sampler3D对象只使用tex3D\*函数。

## 18.5 例子程序：Multitexturing in a Pixel Shader

这章中的例子演示了在像素着色器中使用多纹理，这个例子将纹理一个基于图18.2方格，渲染的目标是一个木箱纹理，一个聚光灯纹理，和一个包含字符串的纹理。这就是例子程序：Pixel Shader。

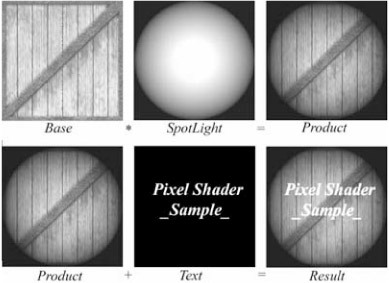


图18.2: 混合纹理. 让我们分别取得木箱纹理上、聚光灯纹理和字符串纹理上相关联的像素颜色：b，s和t，然后定义如何将这些颜色混合： c = b × s + t 。

这个例子可以不使用像素着色器来实现，但实现这个程序是简单直接，它允许我们示范如何写，创建，而且使用像素着色器实现一些特效不必使用那些复杂的算法。

虽然在这个例子中一次只使用3张纹理，检查采样器对象的成员以确定每个像素着色器能够使用的版本，这是值得的。换句话说，我们一次能使用多少纹理这依赖 于使用的像素着色器的版本。

* 像素着色器的版本ps\_1\_1 到 ps\_1\_3支持4个纹理采样器。
* 像素着色器的版本ps\_1\_4支持6个纹理采样器。
* 像素着色器的版本ps\_2\_0到 ps\_3\_0支持16个纹理采样器。

|  |
| --- |
| // File: ps\_multitex.txt  // Desc: Pixel shader that does multitexturing.  // Globals  sampler BaseTex;  sampler SpotLightTex;  sampler StringTex;  // Structures  struct PS\_INPUT  {  float2 base : TEXCOORD0;  float2 spotlight : TEXCOORD1;  float2 text : TEXCOORD2;  };  struct PS\_OUTPUT  {  vector diffuse : COLOR0;  };  // Main  PS\_OUTPUT Main(PS\_INPUT input)  {  // zero out members of output  PS\_OUTPUT output = (PS\_OUTPUT)0;  // sample appropriate textures  vector b = tex2D(BaseTex, input.base);  vector s = tex2D(SpotLightTex, input.spotlight);  vector t = tex2D(StringTex, input.text);  // combine texel colors  vector c =b \*s +t;  // increase the intensity of the pixel slightly  c += 0.1f;  // save the resulting pixel color  output.diffuse = c;  return output;  } |

首先像素着色器定义了3个sampler对象，要渲染的每个纹理，接下来定义是input和output结构。注意：我们没有将任何的颜色值输入到像素着色器中，这是因为我们使用纹理自己的颜色和光照；即BaseTex保存表面的颜色，SpotLightTex是光照图。像素着色器输出只一个简颜色值，指定了我们计算过的这个特定像素的颜色。

Main函数使用tex2D函数采样3 个纹理，即它取得每个纹理的图素，计算映射到的像素，这通常依赖于指定的纹理坐标和采样器对象。然后我们混合图素的颜色用公式：c = b \* s + t。接下来我们让全部的像素变亮一个bit，给每个部分增加0.1f。最后我们保存结果像素颜色并返回它。

现在我们看到了的像素着色器的代码，现在我们改变并考虑应用程序的代码。应用程序有下列相应的全局变量：

|  |
| --- |
| IDirect3DPixelShader9\* MultiTexPS = 0;  ID3DXConstantTable\* MultiTexCT = 0;  IDirect3DVertexBuffer9\* QuadVB = 0;  IDirect3DTexture9\* BaseTex = 0;  IDirect3DTexture9\* SpotLightTex = 0;  IDirect3DTexture9\* StringTex = 0;  D3DXHANDLE BaseTexHandle = 0;  D3DXHANDLE SpotLightTexHandle = 0;  D3DXHANDLE StringTexHandle = 0;  D3DXCONSTANT\_DESC BaseTexDesc;  D3DXCONSTANT\_DESC SpotLightTexDesc;  D3DXCONSTANT\_DESC StringTexDesc; |

多纹理顶点结构的例子如下：

|  |
| --- |
| struct MultiTexVertex  {  MultiTexVertex(float x, float y, float z,  float u0, float v0,  float u1, float v1,  float u2, float v2)  {  \_x = x; \_y = y; \_z = z;  \_u0 = u0; \_v0 = v0;  \_u1 = u1; \_v1 = v1;  \_u2 = u2, \_v2 = v2;  }  float \_x, \_y, \_z;  float \_u0, \_v0;  float \_u1, \_v1;  float \_u2, \_v2;  static const DWORD FVF;  };  const DWORD MultiTexVertex::FVF = D3DFVF\_XYZ | D3DFVF\_TEX3; |

看好，它包含3个纹理坐标系统。

Setup函数执行下列功能：

* 填充方形的顶点缓存
* 编译着像素色器
* 创建像素色器
* 读取纹理
* 设置投影矩阵，不使用光照
* 取得采样器(sampler)对象的句柄
* 取得采样器对象的描述

|  |
| --- |
| bool Setup()  {  HRESULT hr = 0;  // Create quad geometry.  Device->CreateVertexBuffer(  6 \* sizeof(MultiTexVertex),  D3DUSAGE\_WRITEONLY,  MultiTexVertex::FVF,  D3DPOOL\_MANAGED,  &QuadVB,  0);  MultiTexVertex\*v =0;  QuadVB->Lock(0, 0, (void\*\*)&v, 0);  v[0] = MultiTexVertex(-10.0f, -10.0f, 5.0f,  0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f);  v[1] = MultiTexVertex(-10.0f, 10.0f, 5.0f,  0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f);  v[2] = MultiTexVertex( 10.0f, 10.0f, 5.0f,  1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f);  v[3] = MultiTexVertex(-10.0f, -10.0f, 5.0f,  0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f);  v[4] = MultiTexVertex( 10.0f, 10.0f, 5.0f,  1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f);  v[5] = MultiTexVertex( 10.0f, -10.0f, 5.0f,  1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f);  QuadVB->Unlock();  // Compile shader  ID3DXBuffer\* shader = 0;  ID3DXBuffer\* errorBuffer = 0;  hr = D3DXCompileShaderFromFile(  "ps\_multitex.txt",  0,  0,  "Main", // entry point function name  "ps\_1\_1",  D3DXSHADER\_DEBUG,  &shader,  &errorBuffer,  &MultiTexCT);  // output any error messages  if( errorBuffer )  {  ::MessageBox(0, (char\*)errorBuffer->GetBufferPointer(), 0, 0);  d3d::Release<ID3DXBuffer\*>(errorBuffer);  }  if(FAILED(hr))  {  ::MessageBox(0, "D3DXCompileShaderFromFile() - FAILED", 0, 0);  return false;  }  // Create Pixel Shader  hr = Device->CreatePixelShader(  (DWORD\*)shader->GetBufferPointer(),  &MultiTexPS);  if(FAILED(hr))  {  ::MessageBox(0, "CreateVertexShader - FAILED", 0, 0);  return false;  }  d3d::Release<ID3DXBuffer\*>(shader);  // Load textures.  D3DXCreateTextureFromFile(Device, "crate.bmp", &BaseTex);  D3DXCreateTextureFromFile(Device, "spotlight.bmp", &SpotLightTex);  D3DXCreateTextureFromFile(Device, "text.bmp", &StringTex);  // Set projection matrix  D3DXMATRIX P;  D3DXMatrixPerspectiveFovLH(  &P, D3DX\_PI \* 0.25f,  (float)Width / (float)Height, 1.0f, 1000.0f);  Device->SetTransform(D3DTS\_PROJECTION, &P);  // Disable lighting.  Device->SetRenderState(D3DRS\_LIGHTING, false);  // Get handles  BaseTexHandle = MultiTexCT->GetConstantByName(0, "BaseTex");  SpotLightTexHandle = MultiTexCT->GetConstantByName(0, "SpotLightTex");  StringTexHandle = MultiTexCT->GetConstantByName(0, "StringTex");  // Set constant descriptions:  UINT count;  MultiTexCT->GetConstantDesc(  BaseTexHandle,  &BaseTexDesc,  &count);  MultiTexCT->GetConstantDesc(  SpotLightTexHandle,  &SpotLightTexDesc,  &count);  MultiTexCT->GetConstantDesc(  StringTexHandle,  &StringTexDesc,  &count);  MultiTexCT->SetDefaults(Device);  return true;  } |

Display函数设置像素着色器，使用2个纹理，并且在渲染方格前设置他们对应的采样器状态。

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  // ...camera update code snipped  // Render  Device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER,  0xffffffff, 1.0f, 0);  Device->BeginScene();  // set the pixel shader  Device->SetPixelShader(MultiTexPS);  Device->SetFVF(MultiTexVertex::FVF);  Device->SetStreamSource(0, QuadVB, 0, sizeof(MultiTexVertex));  // base tex  Device->SetTexture(BaseTexDesc.RegisterIndex, BaseTex);  Device->SetSamplerState(BaseTexDesc.RegisterIndex,  D3DSAMP\_MAGFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(BaseTexDesc.RegisterIndex,  D3DSAMP\_MINFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(BaseTexDesc.RegisterIndex,  D3DSAMP\_MIPFILTER, D3DTEXF\_LINEAR);  // spotlight tex  Device->SetTexture(SpotLightTexDesc.RegisterIndex, SpotLightTex);  Device->SetSamplerState(SpotLightTexDesc.RegisterIndex,  D3DSAMP\_MAGFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(SpotLightTexDesc.RegisterIndex,  D3DSAMP\_MINFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(SpotLightTexDesc.RegisterIndex,  D3DSAMP\_MIPFILTER, D3DTEXF\_LINEAR);  // string tex  Device->SetTexture( StringTexDesc.RegisterIndex, StringTex);  Device->SetSamplerState(StringTexDesc.RegisterIndex,  D3DSAMP\_MAGFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(StringTexDesc.RegisterIndex,  D3DSAMP\_MINFILTER, D3DTEXF\_LINEAR);  Device->SetSamplerState(StringTexDesc.RegisterIndex,  D3DSAMP\_MIPFILTER, D3DTEXF\_LINEAR);  // draw the quad  Device->DrawPrimitive(D3DPT\_TRIANGLELIST, 0, 2);  Device->EndScene();  Device->Present(0, 0, 0, 0);  }  return true;  } |

当然我们必须想着在Cleanup函数中释放我们自己分配的接口。

|  |
| --- |
| void Cleanup()  {  d3d::Release<IDirect3DVertexBuffer9\*>(QuadVB);  d3d::Release<IDirect3DTexture9\*>(BaseTex);  d3d::Release<IDirect3DTexture9\*>(SpotLightTex);  d3d::Release<IDirect3DTexture9\*>(StringTex);  d3d::Release<IDirect3DPixelShader9\*>(MultiTexPS);  d3d::Release<ID3DXConstantTable\*>(MultiTexCT);  } |

## 18.6 摘要

* 像素着色器取代了固定功能管线的多纹理级别（stage），而且，像素着色器给我们更改单独像素的能力，以任何方式选择和访问纹理数据。因而，使我们能实现很多使用固定功能管线所不能完成的特殊效果。
* 多纹理是一次使用几个纹理，并渲染它们一起创造出一个想要的结果的一个过程。多纹理代表性的用法是用它为静态几何图形实现光引擎。
* HLSL内建的采样器（sampler）对象，标识特定的纹理/采样器级别（stage）。A采样器常用于从像素着色器中引用一个纹理/采样器级别。

注意：一旦你懂得了如何去实现顶点和像素着色器，你需要的一些特效的创意，可以用它们去实现。得到特效创意最好的方法是，学习现有的用顶点和像素着色器实现的特效。《*Direct3D ShaderX: Vertex and Pixel Shader Tips*》和《*Tricks* edited by Wolfgang Engel》这2本书是众多出版物中最好的，像Nvidia和ATI的开发站点：<http://developer.nvidia.com/> 和 <http://ati.com/developer/index.html>。另外我们推荐CG方面：由Randima Fernando 和 Mark J. Kilgard写的《*The Cg Tutorial* by Randima Fernando》，这本书对于使用Cg的3D图形编程是一本相当好的指南，它基本上和Direct3D's HLSL相同。

# 第十九章 效果架构 (The Effects Framework)

## 概览

一个渲染效果一般由以下部分组成：一个顶点和/或像素着色器，一个需要设置的设备状态列表，一个或更多的渲染通道（rendering passes）。此外，有一个能在不同级别的图形硬件上渲染效果的可靠机制通常是值得的（也就是说，有不同的可用的效果版本执行同样的效果或尽可能尝试执行同样的效果）。显然，所有这些必要的任务组合在一起成为一个效果。因此，一个合理的做法（步聚）是，设法将这些任务封装到一个单元中。

Direct3D效果构架提供了这样一个机制：将渲染效果的任务封装到一个效果文件。在效果文件中实现效果有两方面优势。其一，它允许我们不必重编译应用程序就能改变一个效果的执行。这是一种更新效果的过程，不管是修正一个bug，一些简单的加强，或者利用最新的3D硬件特性。第二，它将所有的效果组成部分封装到一个文件。

这一章指导你用必要信息和步骤，编写和创建一个效果文件的。我们注意到效果文件象我们的HLSL程序一样可以写在任何ASCII文件中。

## 目标

         理解一个效果文件的结构和组织

         找到HLSL中的一些额外的对象

         学习如何在效果文件中指定设备状态

         学习如何创建并使用一个效果

         通过学习一些例子程序，取得使用效果框架上的一些经验

## 19.1 技术与传递（Techniques and Passes）

一个效果文件由一个或多个**技术**组成。一个技术是用一个特殊的方法渲染一些特效。所以换句话说，一个效果文件提供了渲染相同特效的一个或多个不同的传递。为什么同样的效果需要几个不同实现呢？是的，一些硬件可能不支持一个效果的一种特定实现。因此，必需在不同硬件上实现相同效果的不同版本。

注意：例如，我们可能实现一种效果的两个版本，一种用着色器实现而一种用固定管线实现。这样，那些有着色器（shader）支持的显卡用户能够利用着色器实现，而那些不支持着色器的用户仍然可以使用固定管线实现。

可以在一个效果文件中实现所有版本的效果，这让我们更完整的封装了所有的效果，也是效果框架的目标之一 ―― 封装（encapsulation）。

每种**技术**包括一次或多次**渲染传递**（passes）。一次渲染传递（*rendering pass*）在特定传递（pass）中封装了设备状态、采样器、和/或用于渲染几何体的着色器。

注意：一个效果不仅限于可编程管线使用。例如，它可以使用固定功能管线控制设备状态，比如灯光、材质以及纹理。

使用多次传递（multiple passes）的理由是，因为对每种特效，是通过使用不同的设备状态、着色器等等，对同样的几何体进行多次渲染来完成的。举例来说，回忆[第8章](#_8.2实例程序：镜子) ，我们不得不在每帧里用不同的设备状态、多次渲染相同的几何体，以达到反射效果。

这个例子，是一个用两种**技术**实现的效果文件的框架，第一种**技术**包括一次传递而每二种**技术**包括两次传递:

|  |
| --- |
| // effect.txt  ...  technique T0  {  // first and only pass for this technique  pass P0  {  ...[specify pass device states, shaders, samplers, etc.]  }  }  technique T1  {  // first pass  pass P0  {  ...[specify pass device states, shaders, samplers, etc.]  }  // second pass  pass P1  {  ...[specify pass device states, shaders, samplers, etc.]  }  } |

## 19.2 更多HLSL内置对象（ More HLSL Intrinsic Objects）

这是一些在HLSL中额外的内建对象类型。我们以前没有过早的提及，是因为它们主要用于效果框架。

### 19.2.1 纹理对象

HLSL内建纹理类型描述了一个IDirect3DTexture9对象。通过使用纹理对象我们可以直接地在效果文件中对特定的采样器阶段结合纹理。纹理对象有下面的可以访问的数据成员：

         type—纹理类型 (例如：2D, 3D)

         format—纹理的像素格式

         width—纹理的宽度（单位像素）

         height—纹理的高度（单位像素）

         depth—纹理的深度（如果是3D纹理，单位像素）

注意：迄今为止我们仅仅使用纹理来存贮图形数据，但是当你学到更高级的技术，你会发现纹理可用来保存任意表格信息。换句话说，纹理仅是数据表，不是必须包含图形数据。例如，在碰撞映射（bump mapping）时我们用到一种叫做法线图的东东（*normal map*），就是一种在每个点上包括了法向量的纹理。

### 19.2.2 采样器对象与采样器状态

我们在[18章](#_18.4_HLSL采样器对象)讨论了采样器对象，然而，效果框架定义了新的关键字：sampler\_state。使用sampler\_state关键字，我们能初始化一个采样器对象（即，直接在效果方件中设置采样器对象的纹理和状态）。下面的例子说明了这点：

|  |
| --- |
| Texture Tex;  sampler SO = sampler\_state  {  Texture = (Tex); // 纹理  // 采样器状态  MinFilter = LINEAR;  MagFilter = LINEAR;  MipFilter = LINEAR;  }; |

这里我们给采样器S0的texture成员关联了纹理 Tex，并给状态成员设置了采样状态。我们直接明了的在效果文件中设置所有信息。

### 19.2.3 顶点与像素着色器对象（Vertex and Pixel Shader Objects）

vertexshader 和 pixelshader是HLSL的内建类型，分别表示**顶点着色器**和**像素着色器**。它们在效果文件中表示特定顶点和/或像素着色器，用于一个特定的**渲染传递**(pass)。vertexshader和/或pixelshader类型在应用程序中用ID3DXEffect::SetVertexShader和ID3DXEffect::SetPixelShader函数分别设置。例如，在效果文件中，让Effect是一个有效的ID3DXEffect对象，让VS是一个有效的IDirect3DVertexShader9对象，以及让VSHandle是一个D3DXHANDLE（是vertexshader 对象的引用）。然后，我们可以通过如下写法初始化VSHandle所引用的顶点着色器：

|  |
| --- |
| Effect->SetVertexShader(VSHandle, VS); |

当在应用程序中设置效果文件中的变量时，多数时候我们使用SetVertexShader 和 SetPixelShader。

做为选择，我们可以直接在效果文件中写顶点和/或像素着色器。当使用一种特定的编译语法时，我们可以设置一个着色器变量。下面的例子展示了如何初始化一个pixelshader类型的变量ps。

|  |
| --- |
| // 定义入口函数  OUTPUT Main(INPUT input){...}    // 编译入口函数  pixelshader ps = compile ps\_2\_0 Main(); |

观察在pixelshader关键字之后的特定的版本名，接下来是着色器入口函数。注意，当用这种方式（style）初始化一个顶点或像素着色器对象时，入口函数必须定义在效果文件中。

最后，我们给一个特定**传递**关联一个着色器，如下：

|  |
| --- |
| // 定义入口函数  OUTPUT Main(INPUT input){...}  // 编译入口函数  vertexshader vs = compile vs\_2\_0 Main();    pass P0  {  // 给这个传递（pass）关联一个着色器（vs）  vertexshader = (vs);  ...  } |

或者更简洁的：

|  |
| --- |
| pass P0  {  // 设置这个传递的顶点着色器，为入口函数" Main()"的顶点着色器  vertexshader = compile vs\_2\_0 Main();  ...  } |

注意：这是一个相当有价值的论述，因此你至少要明白，你能用这样的语法来初始化一个vertexshader 和 pixelshader 类型：

|  |
| --- |
| vertexshader vs = asm { /\*assembly instructions go here \*/ };  pixelshader ps = asm { /\*assembly instructions go here \*/ }; |

如果你用汇编语言来写着色器，你就用这种语法。

### 19.2.4 字符串

最后，这是一个字符串对象，它的用法是这样地：

|  |
| --- |
| string filename = "texName.bmp"; |

尽管没有任何HLSL的内建函数支持字符串类型，但它可以在应用程序中读取。这样，我们能进一步封装效果使用的数据文件，比如纹理文件名和X文件字。

### 19.2.5 注解 (Annotations)

除我们已经描述过的语义符之外，注解可以用在变量上。注解在HLSL中是不使用的，但是它们可以被应用程序通过效果框架访问。它们仅仅服务于一个绑定 “note”的变量，这样应用程序就能够访问这个变量了。为注解加入了<annotation>语法。下面一行举例说明：

|  |
| --- |
| texture tex0 < string name = "tiger.bmp"; >; |

在这个例子中的注解是<string name = "tiger. bmp";>。它关联了一个字符串到变量tex0，即保存纹理数据的文件名。很明显，用相应的文件名注解一个纹理是有益的。

注解可以使用下面函数被重新得到：

|  |
| --- |
| D3DXHANDLE ID3DXEffect::GetAnnotationByName(  D3DXHANDLE hObject,  LPCSTR pName  ); |

pName是我们要操作的注解的名字，而hObject是注解所在的父块句柄，如一个technique、pass或者结构块。一旦我们有了一个注解的句柄，我们就能通过应用ID3DXEffect::GetParameterDesc得到有关它的信息。查看DirectX SDK文档以得到更多详细的内容。

## 19.3 效果文件的设备状态（ Device States in an Effect File）

通常，为了正确执行一个效果，我们必须设置设备的状态，比如渲染状态、纹理状态、材质、灯光和纹理。将全部效果封装进一个文件使它有支持全部效果的能力，效果框架允许我们在效果文件中设置设备状态。设备状态被在渲染的传递部分（pass block）里设置，语法看起来象这样：

|  |
| --- |
| State= Value; |

对于完整的状态的列表，在DirectX SDK文档的索引（index）中查找"states"，或者从SDK的目录（Contents）标签下，查找DirectX Graphics\Reference\Effect Reference\Effect Format\States

考虑FillMode状态。如果你看了一下刚刚提到的SDK中的内容，值与D3DFILLMODE一样，但没有D3DFILL\_前缀。如果我们在SDK文档中查找D3DFILLMODE，我们找到值：D3DFILL\_POINT, D3DFILL\_WIREFRAME, and D3DFILL\_SOLID。因而，对于效果文件我们省略了前缀，并获得下列状态FillMode的有效值：POINT, WIREFRAME, 和 SOLID。例如，你可以在效果文件中这么写-：

|  |
| --- |
| FillMode = WIREFRAME;  FillMode = POINT;  FillMode = SOLID; |

注意： 在后面的小节中我们将在例子程序中设置几个设备状态。多数时候能够通过状态的名字猜到它的用途，但如果你想得到更详细的描述，请查看SDK文档。

## 19.4 创建效果

效果用ID3DXEffect接口表示，我们用下面的D3DX函数创建它：

|  |
| --- |
| HRESULT D3DXCreateEffectFromFile(  LPDIRECT3DDEVICE9 pDevice,  LPCSTR pSrcFile,  CONST D3DXMACRO\* pDefines,  LPD3DXINCLUDE pInclude,  DWORD Flags,  LPD3DXEFFECTPOOL pPool,  LPD3DXEFFECT\* ppEffect,  LPD3DXBUFFER \*ppCompilationErrors  ); |

         pDevice—被创建的ID3DXEffect对象所关联的设备

         pSrcFile—我们要编译的包括效果源代码的文本文件的名字（效果文件名）

         pDefines—这个参数是可选的，在本书中指定为null

         pInclude—ID3DXInclude接口指针。这个接口被设计成由应用程序执行，因而我们可以替换默认行为。通常，默认行为就挺好，我们可以指定null忽略这个参数。

         Flags—编译效果文件中的shader的选项标志，指定0为没有标志。有效选项为：

o        D3DXSHADER\_DEBUG—指示编译器写入调试信息

o        D3DXSHADER\_SKIPVALIDATION—指示编译器不做任何代码检测。这只在你正在用到一个已知正常工作的shader时使用。

o        D3DXSHADER\_SKIPOPTIMIZATION—指示编译器不执行任何优化。实际上这只用于调试时，当你不想让编译器对代码做任何更改时。

         pPool—可选的ID3DXEffectPool接口指针，用于指定效果参数如何共享其它的效果实例。本例中指定null，表示我们不在参数与效果文件之间共享。

         ppEffect—返回一个ID3DXEffect接口指针，表示被创建的效果。

         ppCompilationErrors—返回一个包含错误代码字符串和消息的ID3DXBuffer指针。

这是一个调用D3DXCreateEffectFromFile的例子：

|  |
| --- |
| // 修建效果  ID3DXEffect\* Effect = 0;  ID3DXBuffer\* errorBuffer = 0;  hr = D3DXCreateEffectFromFile(  Device, // 关联的设备  "effect.txt", // 效果源文件  0, // no preprocessor definitions  0, // no ID3DXInclude interface  D3DXSHADER DEBUG, // 编译标记  0, // 不共享参数  &Effect, // 返回创建效果的指针  &errorBuffer); // 返回的错误信息    // 输出错误信息  if( errorBuffer )  {  ::MessageBox(0, (char\*)errorBuffer->GetBufferPointer(), 0, 0);  d3d::Release<ID3DXBuffer\*>(errorBuffer);  }    if (FAILED(hr))  {  ::MessageBox(0, "D3DXCreateEffectFromFile() - FAILED", 0, 0);  return false;  } |

## 19.5 设置系数（Setting Constants）

因为对于顶点和像素着色器，我们需要从程序代码中初始化效果文件中的变量。代替使用常量表，就象我们在顶点和像素着色器中做的那样，ID3DXEffect接口中有内建的设置变量的方法。我们这里不会列出所有的设置不同类型变量的方法，因为要完全列出实在是大多了—请查看DirectX SDK文档以获得完整列表。这里是一个删节的列表：

|  |  |
| --- | --- |
| HRESULT ID3DXEffect::SetFloat(  D3DXHANDLE hParameter,  FLOAT f  ); | Sets a floating-point variable in the effect file identified by hParameter to the value f |
| HRESULT ID3DXEffect::SetMatrix(  D3DXHANDLE hParameter,  CONST D3DXMATRIX\* pMatrix  ); | Sets a matrix variable in the effect file identified by hParameter to the value pointed to by pMatrix |
| HRESULT ID3DXEffect::SetString(  D3DXHANDLE hParameter,  CONST LPCSTR pString  ); | Sets a matrix variable in the effect file identified by hParameter to the value pointed to by pString |
| HRESULT ID3DXEffect::SetTexture(  D3DXHANDLE hParameter,  LPDIRECT3DBASETEXTURE9 pTexture  ); | Sets a texture variable in the effect file identified by hParameter to the value pointed to by pTexture |
| HRESULT ID3DXEffect::SetVector(  D3DXHANDLE hParameter,  CONST D3DXVECTOR4\* pVector  ); | Sets a vector variable in the effect file identified by hParameter to the value pointed to by pVector |
| HRESULT ID3DXEffect::SetVertexShader(  D3DXHANDLE hParameter,  LPDIRECT3DVERTEXSHADER9  pVertexShader  ); | Sets a vertex shader variable in the effect file identified by hParameter to the value pointed to by pVertexShader |
| HRESULT ID3DXEffect::SetPixelShader(  D3DXHANDLE hParameter,  LPDIRECT3DPIXELSHADER9 pPShader  ); | Sets a pixel shader variable in the effect file identified by hParameter to the value pointed to by pPShader |

我们通过下面的方法得到变量（又叫效果参数effect parameters）句柄：

|  |
| --- |
| D3DXHANDLE ID3DXEffect::GetParameterByName(  D3DXHANDLE hParent, // scope of variable - parent structure  LPCSTR pName // name of variable  ); |

它的用法与D3DXConstantTable::GetConstantByName方法一样。即每一个参数是一个D3DXHANDLE，它标识我们想得到的在哪个父结构中的变量句柄。对于没有父结构的全局变量，我们指定null。第二个参数是在效果文件中所显示的变量名。

做为例子，以下显示如何设置效果文件中的一些变量：

|  |
| --- |
| // some data to set  D3DXMATRIX M;  D3DXMatrixIdentity(&M);    D3DXVECTOR4 color(1.0f, 0.0f, 1.0f, 1.0f);    IDirect3DTexture9\* tex = 0;  D3DXCreateTextureFromFile(Device, "shade.bmp", &tex);    // get handles to parameters  D3DXHANDLE MatrixHandle = Effect->GetParameterByName(0, "Matrix");  D3DXHANDLE MtrlHandle = Effect->GetParameterByName(0, "Mtrl");  D3DXHANDLE TexHandle = Effect->GetParameterByName(0, "Tex");    // set parameters  Effect->SetMatrix(MatrixHandle, &M);  Effect->SetVector(MtrlHandle, &color);  Effect->SetTexture(TexHandle, tex); |

注意：对每一个ID3DXEffect::Set\*方法都有相应的ID3DXEffect::Get\*方法用来取得效果文件中的变量值。例如，为得到一个距阵类型的变量，我们可以用这个函数：

|  |
| --- |
| HRESULT ID3DXEffect::GetMatrix(  D3DXHANDLE hParameter,  D3DXMATRIX\* pMatrix  ); |

要取得所有的方法列表，查看DirectX SDK文档。

## 19.6 使用效果

在这一节和它的小节，我们展示一旦一个效果被创建出来后如何使用它。下面步骤概述了全部过程：

1.    得到一个在你想使用的效果文件中的**技术**句柄。

2.    激活想得到的**技术**。

3.    启动当前活动的**技术**。

4.    对每个激活**技术**中的渲染传递，渲染想要的几何体。回想一下，技术可能由几个渲染传递组成，我们必须在每个传递中渲染一次几何体。

5.    结束当前激活的**技术**。

### 19.6.1 获得效果句柄（ Obtaining a Handle to an Effect）

使用**技术**的第一步是获得一个**技术**D3DXHANDLE。可以用这个方法得到一个**技术**句柄：

|  |
| --- |
| D3DXHANDLE ID3DXEffect::GetTechniqueByName(  LPCSTR pName // Name of the technique.  ); |

注意：实际上，一个效果文件包括几个**技术**，每一个都被针对一个特定的硬件能力设计。因此，应用程序通常在系统上运行一些能力测试，然后通过这些测试选择最好的**技术**。看下面小节中的ID3DXEffect::ValidateTechnique。

### 19.6.2 激活一个效果（ Activating an Effect）

一旦得到了想要的**技术**的句柄，我们必须激活这个**技术**。这可以通过下面方法实现：

|  |
| --- |
| HRESULT ID3DXEffect::SetTechnique(  D3DXHANDLE hTechnique // Handle to the technique to set.  ); |

注意：在激活一项**技术**前你可能想用现有设备验证它。也就是说，你也许想确保硬件支持的特色、配置**技术**的使用。你可以用下面的方法：

|  |
| --- |
| HRESULT ID3DXEffect::ValidateTechnique(  D3DXHANDLE hTechnique // Handle to the technique to validate.  ); |

回想一个效果文件可能有几个技术，每个偿试用不同的硬件特色执行一个特定效果，希望最少一个**技术**将在用户系统上执行。对于一个效果，你将遍例每一个**技术**并用ID3DXEffect::ValidateTechnique运行它，因而你能检测哪个技术是被支持的而哪个不被支持，然后进行适当的动作。

### 19.6.3 启动效果

为了使用一个效果渲染几何体，我们必须围绕绘图函数在ID3DXEffect::Begin 和 ID3DXEffect::End**技术**间调用。这些函数就是分别开启和关闭效果。

|  |
| --- |
| HRESULT ID3DXEffect::Begin(  UINT\* pPasses,  DWORD Flags  ); |

         pPasses—返回在当前活动的技术中的传递的数量。

         Flags—下面标志的任何一个：

o        Zero (0)—指定效果保存当前设备状态和着色状态，并在效果结束（这时ID3DXEffect::End被调用）后恢复它们。因为效果文件能够改变状态，对于可以保存启动效果前的状态来说，是很有用的。

o        D3DXFX\_DONOTSAVESTATE—指示效果不保存和恢复设备状态（除shader状态外）。

o        D3DXFX\_DONOTSAVESHADERSTATE—指示效果不保存和恢复shader状态。

### 19.6.4 设置当前的渲染传递（Setting the Current Rendering Pass）

在我们用效果渲染任何几何体前，我们必须指定使用的渲染传递。回想一个**技术**包括一个或多个渲染传递，每一个传递封装了不同的设备状态、采样器、和/或用于这一传递的着色器。渲染传递通过下面方法指定：

|  |
| --- |
| HRESULT ID3DXEffect::Pass(  UINT iPass // Index identifying the pass.  ); |

一个**技术**的渲染传递被用标识为0..*.n-*1的索引，共n个传递。因而，我们能用一个简单的循环遍例每一个传递，并用这一传递渲染几何体。[19.6.6节](#_19.6.6_例子)有一个例子。

### 19.6.5 结束效果（Ending an Effect）

最后，对于每个传递，我们渲染完几何体后，停止并结束效果时使用ID3DXEffect::End方法：

|  |
| --- |
| HRESULT ID3DXEffect::End(VOID); |

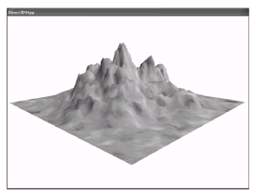
### 19.6.6 例子

下面的代码片断示例了以上的使用一个效果的必要的五个步骤：

|  |
| --- |
| // 有效果文件中  technique T0  {  pass P0  {  ...  }  }  ===============================    // 在应用程序中，取得**技术**句柄  D3DXHANDLE hTech = 0;  hTech = Effect->GetTechniqueByName("TO");    // 激活**技术**  Effect->SetTechnique(hTech );    // 启动激活的**技术**  UINT numPasses = 0;  Effect->Begin(&numPasses, 0);    // 遍例每个传递  for(int i = 0; i < numPasses; i++)  {  // 设置当前传递  Effect->Pass(i);    // 在传递中渲染几何体  Sphere->Draw();  }  // 结束效果  Effect->End(); |

## 19.7 例子程序: Lighting and Texturing in an Effect File

做为热身，让我们创建一个在3D模型中操作灯光和纹理的效果文件。这个例子完全运行于固定功能管线，意味着效果框架不仅限于使用着色器。图19.1展示了使用灯光和纹理例子的屏幕截图。

  
图19.1: 灯光和纹理例子的屏幕截图. 纹理、材质和灯光状态在效果文件中指定。

以下是效果文件的实现：

|  |
| --- |
| // File: light tex.txt  // Desc: 效果文件控制光的设备状态，和纹理一个3D模型    // 全局变量  matrix WorldMatrix;  matrix ViewMatrix;  matrix ProjMatrix;    texture Tex;    // 过滤器    // Associated the texture 'Tex' with the texture stage 'S0'  // corresponds with and also set the sampler states for the sampler  // stage 'S0' corresponds with.  sampler S0 = sampler state  {  Texture = (Tex);  MinFilter = LINEAR;  MagFilter = LINEAR;  MipFilter = LINEAR;  };    // Effect  technique LightAndTexture  {  pass P0  {  // Set misc. render states.  pixelshader = null; // No pixel shader.  vertexshader = null; // No vertex shader.  fvf = XYZ | Normal | Tex1; // Flexible vertex format  Lighting = true; // Enable lighting.  NormalizeNormals = true; // Renormalize normals.  SpecularEnable = false; // Disable specular highlights.    // Set transformation states  WorldTransform[0] = (WorldMatrix);  ViewTransform = (ViewMatrix);  ProjectionTransform = (ProjMatrix);    // Set a light source at light index 0. We fill out all the  // components for light[0] because the Direct3D  // documentation recommends filling out all components  // for best performance.  LightType[0] = Directional;  LightAmbient[0] = {0.2f, 0.2f, 0.2f, 1.0f};  LightDiffuse[0] = {1.0f, 1.0f, 1.0f, 1.0f};  LightSpecular[0] = {0.0f, 0.0f, 0.0f, 1.0f};  LightDirection[0] = {1.0f, -1.0f, 1.0f, 0.0f};  LightPosition[0] = {0.0f, 0.0f, 0.0f, 0.0f};  LightFalloff[0] = 0.0f;  LightRange[0] = 0.0f;  LightTheta[0] = 0.0f;  LightPhi[0] = 0.0f;  LightAttenuation0[0] = 1.0f;  LightAttenuation1[0] = 0.0f;  LightAttenuation2[0] = 0.0f;    // Finally, enable the light:  LightEnable[0] = true;    // Set material components. This is like calling  // IDirect3DDevice9::SetMaterial.  MaterialAmbient = {1.0f, 1.0f, 1.0f, 1.0f};  MaterialDiffuse = {1.0f, 1.0f, 1.0f, 1.0f};  MaterialEmissive = {0.0f, 0.0f, 0.0f, 0.0f};  MaterialPower = 1.0f;  MaterialSpecular = {1.0f, 1.0f, 1.0f, 1.0f};    // Hook up the sampler object 'S0' to sampler stage 0,  // which is given by Sampler[0].  Sampler[0] = (S0);  }  } |

在这个效果文件中我们主要设置设备状态，就象在[19.3节](#_19.3_效果文件的设备状态（_Device)所述。例如，我们直接在效果文件中设置一个光源和一个材质。此外，我们指定转换距阵和纹理及采样器状态。这些状态被指定，然后用LightAndTexture方法和渲染传递P0渲染全部几何体，。

注意：考虑到在一个效果文件中涉及到的的变量，你必须把它们装入圆括号中。举例来说，涉及到距阵变量，你必须这样写：(WorldMatrix), (ViewMatrix), and (ProjMatrix)。不使用圆括号是违法的。

因为大部分必需的和繁琐的工作都在效果文件里做了，比如设置灯光、材质和纹理。应用程序代码就是做一些创建效果和开启效果等简单的事情。例子中有下面一些相关的全局变量：

|  |
| --- |
| ID3DXEffect\* LightTexEffect = 0;  D3DXHANDLE WorldMatrixHandle = 0;  D3DXHANDLE ViewMatrixHandle = 0;  D3DXHANDLE ProjMatrixHandle = 0;  D3DXHANDLE TexHandle = 0;  D3DXHANDLE LightTexTechHandle = 0; |

 这些东西很没劲 —--- 只是一个ID3DXEffect指针和一些句柄。LightTexTechHandle是一个**技术**的句柄，因此在它的名字中有子字符串“Tech”。

RestoreDeviceObjects函数执行三个主要步骤：创建效果，获得作为效果参数的我们要用的**技术**的句柄，并初始化一些效果参数。下面是删节的实现：

|  |
| --- |
| bool Setup()  {  HRESULT hr = 0;  // ...省略了采样器的读取    // 创建效果  ID3DXBuffer\* errorBuffer = 0;  hr = D3DXCreateEffectFromFile(  m\_pd3dDevice ,  "light\_tex.txt",  0, // 没有定义预处理器  0, // 没有ID3DXInclude接口  D3DXSHADER\_DEBUG, // 编译标记  0, // 不共享参数  &m\_LightTexEffect,  &errorBuffer);    // 输出错误信息  if( errorBuffer )  {  ::MessageBox(0, (char\*)errorBuffer->GetBufferPointer(), 0, 0);  SAFE\_RELEASE(errorBuffer);  }    if(FAILED(hr))  {  ::MessageBox(0, "D3DXCreateEffectFromFile() - FAILED", 0, 0);  return false;  }    // 保存经常访问的参数句柄  m\_WorldMatrixHandle = m\_LightTexEffect->GetParameterByName(0, "WorldMatrix");  m\_ViewMatrixHandle = m\_LightTexEffect->GetParameterByName(0, "ViewMatrix");  m\_ProjMatrixHandle = m\_LightTexEffect->GetParameterByName(0, "ProjMatrix");  m\_TexHandle = m\_LightTexEffect->GetParameterByName(0, "Tex");  m\_LightTexTechHandle =  m\_LightTexEffect->GetTechniqueByName("LightAndTexture");    // 设置效果参数  // 设置矩阵  D3DXMATRIX W, P;  D3DXMatrixIdentity(&W);  m\_LightTexEffect->SetMatrix( m\_WorldMatrixHandle, &W);    D3DXMatrixPerspectiveFovLH(  &P,  D3DX\_PI \* 0.25f, // 45 - degree  (float)800.0f / (float)600.0f,  1.0f,  1000.0f);  m\_LightTexEffect->SetMatrix( m\_ProjMatrixHandle, &P);    // Set texture  IDirect3DTexture9\* tex = 0;  D3DXCreateTextureFromFile(m\_pd3dDevice, "Terrain\_3x\_diffcol.jpg", &tex);  LightTexEffect->SetTexture(TexHandle, tex);  d3d::Release<IDirect3DTexture9\*>(tex);    return true;  } |

Disply函数很简单，运行步聚在[19.6 节](#_19.6_使用效果)中简要说明:

|  |
| --- |
| bool Display(float timeDelta)  {  if( Device )  {  // ...[Camera update snipped]  // set the new updated view matrix  LightTexEffect->SetMatrix(ViewMatrixHandle, &V);  // Activate the technique and render  Device->Clear(0, 0, D3DCLEAR TARGET | D3DCLEAR ZBUFFER,  0xffffffff, 1.0f, 0);  Device->BeginScene();  // set the technique to use  LightTexEffect->SetTechnique( LightTexTechHandle );  UINT numPasses = 0;  LightTexEffect->Begin(&numPasses, 0);  for(int i = 0; i < numPasses; i++)  {  LightTexEffect->Pass(i);  for(int j = 0; j < Mtrls.size(); j++)  {  Mesh->DrawSubset(j);  }  }  LightTexEffect->End();  Device->EndScene();  Device->Present(0, 0, 0, 0);  }  return true;  } |

## 19.8例子程序: Fog Effect

非常遗憾，我们没有用一整章篇幅来介绍**Direct3D雾化效果**。雾化效果（以下简称雾）提高了场景的真实性，可以用它来模拟逼真的天气状况。另外，雾可以大大减少长剪裁（far-clip）平面视觉效果。

虽然我们不能给它应有的重视，这里我们还是挤出了一个简要的雾化例程。虽然我们不涉及详细的细节，我们还是展示并解释了Direct3D代码，这是很直接的。

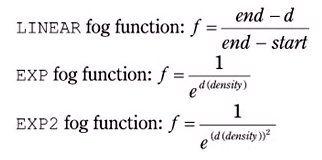
Direct3D雾化是固定功能管线的一部份，受渲染状态限制。下面的效果文件设置顶点雾，以达到必要的雾化状态。

注意：Direct3D也支持像素雾（也叫表格雾table fog），比顶点雾要更精确。

|  |
| --- |
| // File: fog.txt  // Desc: Effect file that handles device states for linear vertex fog.  technique Fog  {  pass P0  {  // Set misc render states.  pixelshader = null;  vertexshader = null;  fvf = XYZ | Normal;  Lighting = true;  NormalizeNormals = true;  SpecularEnable = false;    // Fog states  FogVertexMode = LINEAR; // Linear fog function.  FogStart = 50.0f; // Fog starts 50 units away from viewpoint.  FogEnd = 300.0f; // Fog ends 300 units away from viewpoint.    FogColor = 0x00CCCCCC; // Gray colored fog.  FogEnable = true; // Enable vertex fog.  }  } |

就象你看到的，线性顶点雾能够通过五个简单的渲染状态控制：

         FogVertexMode—使用指定的雾函数用于顶点雾。雾函数指定雾如何根据距离增长，自然界的雾在近视口的地方比较薄并且根据距离增长变得厚起来了。有效的任务类型为LINEAR、EXP、EXP2。这些函数被定义为：



*d* 是到视口的距离(viewpoint.)

注意：如果你用EXP或EXP2雾化函数，你不用设置FogStart 和 FogEnd，因为它们在这些雾函数类型中没被用到。代替的你必须设置雾密度（fog density）渲染状态（如，FogDensity = someFloatType）

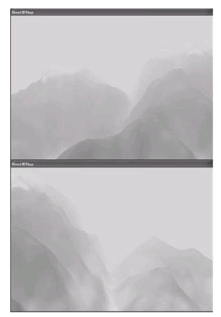
         FogStart—标记了物体将开始雾化的起始深度。

         FogEnd—标记了物体将结束雾化的结束深度。

注意：FogStart 与 FogEnd本质上定义了物体在其中被雾化的深度间隔（从视口）。

         FogColor—一个DWORD 或 D3DCOLOR值，以描述雾的颜色

         FogEnable—指定true以开启顶点雾或false以关闭顶点雾

任何我们用fog.txt效果渲染的几何体将被雾化。通过这种方式，我们可以控制哪一物体得到雾化，而哪些不用雾化。这对只雾化特定区域是很有用的。例如，通常屋外是有雾的，屋里不被雾化。同样的，一定地理部分可能有雾，而另外部分可能没有。图19.2展示了这一小节的调用雾效果的例程的屏幕截图。  
图19.2: 雾化效果例子程序的屏幕截图，在这个例子中我们使用线性雾函数，而且雾化渲染状态在效果文件中指定。

## 19.9例子程序: Cartoon Effect

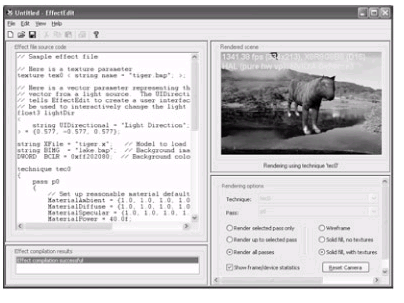
到目前为止的2个效果文件的例子，我们没有使用着色器（shader）。因为着色器在特效中的重要部分，我们想展示一个最精简的例子。例程CartoonEffect执行了在[17章](#_第十七章_顶点着色器入门_(Introduction)中讨论的卡通着色器，但是这次应用效果框架。下面是一个删节版的效果文件：

|  |
| --- |
| // File: tooneffect.txt  // 在效果文件中的卡通着色器  extern matrix WorldMatrix;  extern matrix ViewMatrix;  extern matrix ProjMatrix;  extern vector Color;  extern vector LightDirection;  static vector Black = {0.0f, 0.0f, 0.0f, 0.0f};  extern texture ShadeTex;    struct VS\_INPUT  {  vector position : POSITION;  vector normal : NORMAL;  };  struct VS\_OUTPUT  {  vector position : POSITION;  float2 uvCoords : TEXCOORD;  vector diffuse : COLOR;  };    // Cartoon Shader Function:  VS\_OUTPUT Main(VS\_INPUT input)  {  ...[Implementation omitted for brevity.]  }    sampler ShadeSampler = sampler state  {  Texture = (ShadeTex);  MinFilter = POINT; // no filtering for cartoon shading  MagFilter = POINT;  MipFilter = NONE;  };    technique Toon  {  pass P0  {  // Set P0's vertex shader.  vertexShader = compile vs\_1\_1 Main();  // Hook up the sampler object to sampler stage 0.  Sampler[0] = (ShadeSampler);  }  } |

我们注意到卡通着色器函数被定义在效果文件中，并且我们指定着色器使用一个特定的传递，在传递部分使用语法：vertexShader = compile vs\_1\_1\_Main();。在效果文件中的设备状态象通常一样设置。

## 19.10 效果编辑（EffectEdit）

在结束这章前，我们想提一下在DirectX SDK中的EffectEdit程序。可以在\DXSDK\Samples\C++\Direct3D\Bin文件夹中找到它。图19.3显示了一个屏幕截图。

  
图19.3: 一个在DirectX SDK 中的EffectEdit 程序的屏幕截图

这个EffectEdit程序在测试和书写效果文件时是很有用的。我们推荐您在这个工具上花点时间。

## 19.11摘要

## 略

# 相关文章：

# AGP内存

AGP（Accelerate Graphical Port），加速图形接口。随着显示芯片的发展，PCI总线日益无法满足其需求。英特尔于1996年7月正式推出了AGP接口，它是一种显示卡专用的局部总线。严格的说，AGP不能称为总线，它与PCI总线不同，因为它是点对点连接，即连接控制芯片和AGP显示卡，但在习惯上我们依然称其为AGP总线。AGP接口是基于PCI 2.1 版规范并进行扩充修改而成，工作频率为66MHz。

　　AGP总线直接与主板的北桥芯片相连，且通过该接口让显示芯片与系统主内存直接相连，避免了窄带宽的PCI总线形成的系统瓶颈，增加3D图形数据传输速度，同时在显存不足的情况下还可以调用系统主内存。所以它拥有很高的传输速率，这是PCI等总线无法与其相比拟的。

　　由于采用了数据读写的流水线操作减少了内存等待时间，数据传输速度有了很大提高；具有133MHz及更高的数据传输频率；地址信号与数据信号分离可提高随机内存访问的速度；采用并行操作允许在CPU访问系统RAM的同时AGP显示卡访问AGP内存；显示带宽也不与其它设备共享，从而进一步提高了系统性能。

　　AGP标准在使用32位总线时，有66MHz和133MHz两种工作频率，最高数据传输率为266Mbps和533Mbps，而PCI总线理论上的最大传输率仅为133Mbps。目前最高规格的AGP 8X模式下，数据传输速度达到了2.1GB/s。

　　AGP接口的发展经历了AGP1.0(AGP1X、AGP2X)、AGP2.0(AGP Pro、AGP4X)、AGP3.0(AGP8X)等阶段，其传输速度也从最早的AGP1X的266MB/S的带宽发展到了AGP8X的2.1GB/S。

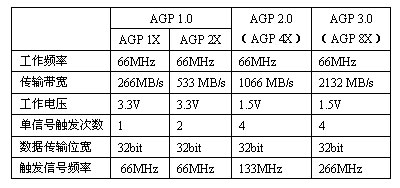
**AGP 1.0（AGP1X、AGP2X）**　 1996年7月AGP 1.0 图形标准问世，分为1X和2X两种模式，数据传输带宽分别达到了266MB/s和533MB/s。这种图形接口规范是在66MHz PCI2.1规范基础上经过扩充和加强而形成的，其工作频率为66MHz，工作电压为3.3v，在一段时间内基本满足了显示设备与系统交换数据的需要。这种规范中的AGP带宽很小，现在已经被淘汰了，只有在前几年的老主板上还见得到。

**AGP2.0(AGP4X)**  
　　显示芯片的飞速发展，图形卡单位时间内所能处理的数据呈几何级数成倍增长，AGP 1.0 图形标准越来越难以满足技术的进步了，由此AGP 2.0便应运而生了。1998年5月份，AGP 2.0 规范正式发布，工作频率依然是66MHz，但工作电压降低到了1.5v，并且增加了4x模式，这样它的数据传输带宽达到了1066MB/sec，数据传输能力大大地增强了。

**AGP Pro**  
　　AGP Pro接口与AGP 2.0同时推出，这是一种为了满足显示设备功耗日益加大的现实而研发的图形接口标准，应用该技术的图形接口主要的特点是比AGP 4x略长一些，其加长部分可容纳更多的电源引脚，使得这种接口可以驱动功耗更大（25-110w）或者处理能力更强大的AGP显卡。这种标准其实是专为高端图形工作站而设计的，完全兼容AGP 4x规范，使得AGP 4x的显卡也可以插在这种插槽中正常使用。AGP Pro在原有AGP插槽的两侧进行延伸，提供额外的电能。它是用来增强，而不是取代现有AGP插槽的功能。根据所能提供能量的不同，可以把AGP Pro细分为AGP Pro110和AGP Pro50。在某些高档台式机主板上也能见到AGP Pro插槽，例如华硕的许多主板。

**AGP 3.0(AGP8X)**  
　　2000年8月，Intel推出AGP3.0规范，工作电压降到0.8V,并增加了8x模式，这样它的数据传输带宽达到了2133MB/sec，数据传输能力相对于AGP 4X成倍增长，能较好的满足当前显示设备的带宽需求。

**AGP接口的模式传输方式**  
　　不同AGP接口的模式传输方式不同。1X模式的AGP，工作频率达到了PCI总线的两倍—66MHz，传输带宽理论上可达到266MB/s。AGP 2X工作频率同样为66MHz，但是它使用了正负沿（一个时钟周期的上升沿和下降沿）触发的工作方式，在这种触发方式中在一个时钟周期的上升沿和下降沿各传送一次数据，从而使得一个工作周期先后被触发两次，使传输带宽达到了加倍的目的，而这种触发信号的工作频率为133MHz，这样AGP 2X的传输带宽就达到了266MB/s×2（触发次数）＝533MB/s的高度。AGP 4X仍使用了这种信号触发方式，只是利用两个触发信号在每个时钟周期的下降沿分别引起两次触发，从而达到了在一个时钟周期中触发4次的目的，这样在理论上它就可以达到266MB/s×2（单信号触发次数）×2（信号个数）＝1066MB/s的带宽了。在AGP 8X规范中，这种触发模式仍然使用，只是触发信号的工作频率变成266MHz，两个信号触发点也变成了每个时钟周期的上升沿，单信号触发次数为4次，这样它在一个时钟周期所能传输的数据就从AGP4X的4倍变成了8倍，理论传输带宽将可达到266MB/s×4（单信号触发次数）×2（信号个数）＝2133MB/s的高度了。



　　目前常用的AGP接口为AGP4X、AGP PRO、AGP通用及AGP8X接口。需要说明的是由于AGP3.0显卡的额定电压为0.8—1.5V，因此不能把AGP8X的显卡插接到AGP1.0规格的插槽中。这就是说AGP8X规格与旧有的AGP1X/2X模式不兼容。而对于AGP4X系统，AGP8X显卡仍旧在其上工作，但仅会以AGP4X模式工作，无法发挥AGP8X的优势。

# Direct3D中实现图元的鼠标拾取

索引：

　　1、什么是拾取，拾取能做什么？

　　2、拾取操作的步骤和实现

　　2.1． 变换并获得通过视点和屏幕上点击点的射线矢量（Dir）

　　 2.1.1 确定鼠标选取点的屏幕坐标

　　 2.1.2 得到Dir在观察坐标空间内的表示

　　 2.1.3 转换Dir到世界坐标空间，并得到观察点在世界坐标系中的坐标

　　2.2 使用射线矢量对场景中的所有三角形图元求交，获得三角形索引值和重心坐标。

　　 2.2.1 D3D扩展函数实现求交

　　 2.2.2射线三角面相交的数学算法

　　 2.2.3 拾取完成根据获得的中心坐标计算我们关心的常见量

　　3、结束及声明

　　4、参考文献

　　补充：重心坐标的概念

　　3D交互图形应用程序中，常常要用鼠标去选择图形，其实现的机制基于鼠标拾取算法。本文主要讲述如何在D3D中实现图元的鼠标拾取。为了讨论简单，本文假定读者理解D3D 坐标变换流程和基本的图形学知识，如果阅读有困难请参考相关资料。

### 1、什么是拾取，拾取能做什么？

首先，拾取操作指当我们在屏幕上用鼠标点击某个图元，应用程序能返回该图元的一个标志和某些相关信息。有图形程序设计经验的人都知道，有这些信息就表示我们有了对该图元的控制权，我们可以删除，可以编辑，可以任意对待该图元，至于你到底想干什么，就是阁下自己的事了^\_^。

### 2、拾取操作的步骤和实现

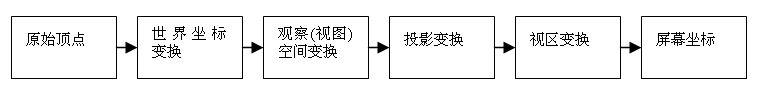
　　拾取算法的思想很简单：得到鼠标点击处的屏幕坐标，通过投影矩阵和观察矩阵把该坐标转换为通过视点和鼠标点击点的一条射入场景的光线，该光线如果与场景模型的三角形相交（本文只处理三角形图元），则获取该相交三角形的信息。本文讲述的方法除可以得到三角形的一个索引号以外还可以得到相交点的重心坐标。

从数学角度来看，我们只要得到射线的**方向矢量**和射线的**发射点**，我们就具备了判断射线与空间三角面是否相交的条件，本文主要讨论如何获得这些条件，并描述了射线与三角面相交判断算法和D3D的通常实现方法。

　　根据拾取操作的处理顺序，大概可以依次分为以下几个步骤

#### 2.1． 变换并获得通过视点和屏幕上点击点的射线矢量（Dir）

详细介绍之前，为了大家方便理解，我们要先简单说一下d3d坐标转换的大概流程，如下图:



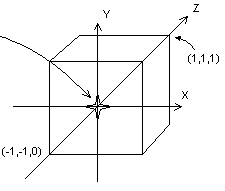
　　 所以我们要通过一系列的反变换，得到我们关心的值在世界坐标中的表示。

　　2.1.1 确定鼠标选取点的屏幕坐标

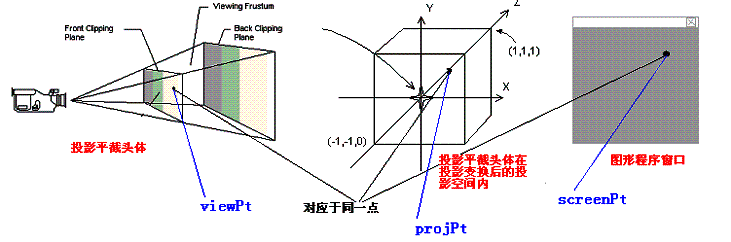
　　这一步是非常简单的Windows给我们提供了API来完成屏幕坐标的获取，使用GetCursorPos获得鼠标指针位置，然后再利用ScreenToClient转换坐标到客户区坐标系(以窗口视区左上角为坐标原点，单位为像素)，设该坐标为（POINT screenPt）。

　　2.1.2 得到Dir在观察坐标空间内的表示

在观察坐标系中，Dir是一条从观察坐标原点出发的射线，所以我们只需要再确定一个该射线经过的点，就可以得到它在观察坐标系中的表示。假设我们要求的射线上的另外一点为该射线与透视投影平截头体近剪切面的交点，针对最普遍的透视投影而言，透视投影平截头体经投影变换后，变成一个1/2立方体（请允许我这么叫^\_^，因为它的大小为一个正方体的一半，x,y方向边长为2，z方向为1）如图：



投影坐标系以近剪切面中心为坐标原点，该立方体从z轴负向看过去与图形程序视区相对应，最终近剪切面（前剪切面）上一点与屏幕坐标之间的对应关系如下图所示：



　　根据比例关系，screenPt与投影空间上的点projPt之间的关系为

　　假设图形程序窗口的宽为screenWidth,高为screenHeight,

　　projPt.x = (screenPt.x-screenWidth/2)/screenWidth\*2; （公式1）

　　projPt.y = (screenPt.y-screenHeight/2)/screenHeight\*2; （公式2）

　　projPt.z =0;（实际该值可任意取，不影响最终结果。为了处理简单，我们取改值为0，表示该点取在近剪切面上）

　　得到projPt后，我们需要做的是把该点坐标从投影空间转换到观察空间(view space),

　　根据透视投影的定义，可假设点(projPt.x，projPt.y，projPt.z)

　　对应的其次坐标为

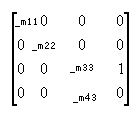
　　(projPt.x\*projPt.w，projPt.y\*projPt.w，projPt.z\*projPt.w，projPt.w)

　　我们可以通过 GetTransform( D3DTS\_PROJECTION, &ProjMatrix)函数获得投影矩阵ProjMatrix,则根据观察空间到投影空间的变换关系则：投影坐标 = 观察坐标×投影矩阵

　　(projPt.x\*projPt.w，projPt.y\*projPt.w，projPt.z\*projPt.w，projPt.w)

　　= (viewPt.x，viewPt.y，viewPt.z, 1)\*pProjMatrx;

　　根据定义和图形学原理

　　ProjMatrix = =

　　所以,

　　(projPt.x\*projPt.w，projPt.y\*projPt.w，projPt.z\*projPt.w，projPt.w)

　　= ( viewPt.x\*ProjMatrix.\_m11,

　　 viewPt.y\*ProjMatrix.\_m22,

　　 viewPt.z\*Q-QZn,

viewPt.z)

　　所以

　　projPt.x\*projPt.w = viewPt.x\*ProjMatrix.\_m11

　　projPt.y\*projPt.w = viewPt.y\*ProjMatrix.\_m22

　　projPt.z\*projPt.w = viewPt.z\*Q-QZn （注意projPt.z = 0）

　　projPt.w = viewPt.z;

　　解得

　　viewPt.x = projPt.x\*Zn/ ProjMatrix.\_m11;

　　viewPt.y = projPt.y\*Zn/ ProjMatrix.\_m22;

　　viewPt.z = Zn;

　　好了，到这里为止我们终于求出了射线与近剪切面交点在观察坐标系中的坐标，现在我们拥有了射线的出发点(0,0,0)和射线方向上另外一点(viewPt.x,viewPt.y,viewPt.z),则该射线的方向矢量在观察空间中的表示可确定为（viewPt.x-0,viewPt.y-0,viewPt.z-0）,化简一下三个分量同除近剪切面z坐标Zn，该方向矢量可写作

　　DIRview = (projPt.x/projMatrix.\_m11,projPt.y/projMatrix.\_m22,1)

　　代入公式1，公式2

　　DIRview.x = (2\*screenPt.x/screenWidth-1)/projMatrix.\_m11;

　　DIRview.y = (2\*screenPt.y/screenHeight-1)/projMatrix.\_m22;

　　DIRview.z = 1;

其中screenWidth和screenHeight可以通过图像显示的backBuffer的目标表面（D3DSURFACE\_DESC）来获得，该表面在程序初始化时由用户创建。

　　2.1.3 转换Dir到世界坐标空间，并得到观察点在世界坐标系中的坐标

　　由于最终的运算要在世界坐标空间中进行，所以我们还需要把矢量DIRview从观察空间转换为世界坐标空间中的矢量DIRworld。

　　因为

　　DIRview = DIRworld\*ViewMatrix;

　　其中ViewMatrix为观察矩阵，在D3D中可以用函数GetTransform( D3DTS\_VIEW, &ViewMatrix )得到。

　　所以DIRworld = DIRview \* inverse\_ViewMatrix,其中inverse\_ViewMatrix为ViewMatrix的逆矩阵。

　　观察点在观察坐标系中坐标为OriginView（0，0，0，1），所以其在世界坐标系中的坐标同样可以利用ViewMatrix矩阵，反变换至世界坐标系中，事实上我们可以很简单的判断出,其在世界坐标系中的表示为:

　　OriginWorld = (inverse\_ViewMatrix.\_41,

　　inverse\_ViewMatrix.\_42,

　　inverse\_ViewMatrix.\_43,

　　1);

　　到这里为止，判断射线与三角面是否相交的条件就完全具备了。

#### 2.2 使用射线矢量对场景中的所有三角形图元求交，获得三角形索引值和重心坐标。

　　这一步骤地实现由两种途径:

　　第一种方法非常简单，利用D3D提供的扩展函数**D3DXIntersect**可以轻松搞定一切。见2.1

　　第二种方法就是我们根据空间解析几何的知识，自己来完成射线三角形的求交算法。一般来讲，应用上用第一种方法就足够了，但是我们如果要深入的话，必须理解相交检测的数学算法，这样才能自由的扩展，面对不同的需求，内容见2.2

　　下面分别讲解两种实现途径：

　　2.2.1 D3D扩展函数实现求交

　　这种方法很简单也很好用，对于应用来说应尽力是用这种方式来实现，毕竟效率比自己写得要高得多。

　　实际上其实没什么好讲的，大概讲一下函数D3DXIntersect吧

　　D3D SDK该函数声明如下

　　HRESULT D3DXIntersect(

　　LPD3DXBASEMESH pMesh,  
　　CONST D3DXVECTOR3 \*pRayPos,  
　　CONST D3DXVECTOR3 \*pRayDir,  
　　BOOL \*pHit,  
　　DWORD \*pFaceIndex,  
　　FLOAT \*pU,  
　　FLOAT \*pV,  
　　FLOAT \*pDist,  
　　LPD3DXBUFFER \*ppAllHits,  
　　DWORD \*pCountOfHits  
　　);  
　　l pMesh指向一个ID3DXBaseMesh的对象，最简单的方式是从.x文件获得，描述了要进行相交检测的三角面元集合的信息，具体规范参阅direct9 SDK  
　　l pRayPos 指向射线发出点  
　　l pRayDir 指向前面我们辛辛苦苦求出的射线方向的向量  
　　l pHit 当检测到相交图元时，指向一个true,不与任何图元相交则为假  
　　l pU 用于返回重心坐标U分量  
　　l pV返回重心坐标V分量  
　　l pDist 返回射线发出点到相交点的长度  
　　注意：以上红色字体部分均指最近的一个返回结果（即\*pDist最小）  
　　l ppAllHits用于如果存在多个相交三角面返回相交的所有结果  
　　l pCountOfHits 返回共有多少个三角形与该射线相交

补充：重心坐标的概念

|  |
| --- |
| 三角形的重心坐标：  P1，P2，P3为空间三角形的三个顶点矢量， (U, V)就称为三角形的重心坐标  在空间三角形平面上的点可以表示为：　**P = P1 + U \* (P2 - P1) + V \* (P3 - P1)**  当0 < U < 1，0 < V < 1，0 < U + V < 1时，这个点P就在这个三角形的内部  收集 |

其中pU和pV用到了**重心坐标的概念**，下面稍作描述

一个三角形有三个顶点，在迪卡尔坐标系中假设表示：

**V1**(x1,y1,z1),

**V2**(x2,y2,z2),

**V3**(x3,y3,z3),

则三角形内任意一点的坐标可以表示为（**pV**为任意点）：

**pV = V1 + U(V2-V1) + V(V3-V1)**

所以已知三个顶点坐标的情况下，任意一点可用坐标(U,V)来表示，其中 参数U控制V2在结果中占多大的权值，参数V控制V3占多大权值，最终V1占多大权值 = 1 - U - V，这种坐标定义方式就叫重心坐标。

　　2..2.2射线三角面相交的数学算法

　　使用d3d扩展函数，毕竟有时不能满足具体需求，掌握了该方法，我们才能够获得最大的控制自由度，任意修改算法。

　　已知条件: 射线源点orginPoint,三角形三个顶点 v1,v2,v3,射线方向 Dir（均以三维坐标向量形式表示）。

　　算法目的: 判断射线与三角形是否相交，如果相交求出交点的重心坐标(U,V)和射线原点到交点的距离T。

我们可先假设射线与三角形相交则交点

(注以下均为向量运算：

dot(X,Y) 点乘；

cross(X，Y) 叉乘；

U，V，T 标量(这三个值为X、Y、Z轴坐标))

　　则：（IntersectPoint为三角形上的交点）

　　IntersectPoint = V1 + U×(V2-V1) + V×(V3-V1) ;

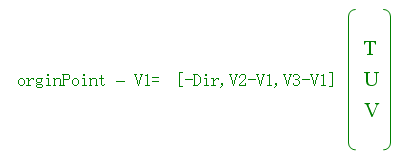
　　IntersectPoint = originPoint + T×Dir；

　　所以：

　　orginPoint + T×Dir = V1 + U×(V2-V1) + V×(V3-V1);

整理得：

orginPoint - V1 = U×(V2-V1) + V×(V3-V1) - T×Dir;



这是一个简单的线性方程组，若有解则行列式［-Dir, V2-V1, V3-V1］不为0。

根据T,U,V的含义当T>0, 0<U<1,0<V<1,0<U+V<1时该交点在三角形内部，解此方程组即可获得我们关心的值,具体解法不再赘述，克莱姆法则就够了（详细见线性代数）:射线原点到相交点的距离T,和交点的中心坐标(U,V)。

下面给出Direct 9 SDK示例程序中的实现代码：

|  |
| --- |
| IntersectTriangle( const D3DXVECTOR3& orig,                     const D3DXVECTOR3& dir, D3DXVECTOR3& v0,                      D3DXVECTOR3& v1, D3DXVECTOR3& v2,                      FLOAT\* t, FLOAT\* u, FLOAT\* v )  {      // 算出两个边的向量      D3DXVECTOR3 edge1 = v1 - v0;      D3DXVECTOR3 edge2 = v2 - v0;    D3DXVECTOR3 pvec;  D3DXVec3Cross( &pvec, &dir, &edge2 );  // 如果det为0，或接近于零则射线与三角面共面或平行，不相交  //此处det就相当于上面的[-Dir, V2-V1, V3-V1]，      FLOAT det = D3DXVec3Dot( &edge1, &pvec );        D3DXVECTOR3 tvec;      if( det > 0 )      {          tvec = orig - v0;      }      else      {          tvec = v0 - orig;          det = -det;      }        if( det < 0.0001f )          return FALSE;        // 计算u并测试是否合法（在三角形内）      \*u = D3DXVec3Dot( &tvec, &pvec );      if( \*u < 0.0f || \*u > det )          return FALSE;        // Prepare to test V parameter      D3DXVECTOR3 qvec;      D3DXVec3Cross( &qvec, &tvec, &edge1 );        //计算u并测试是否合法（在三角形内）      \*v = D3DXVec3Dot( &dir, &qvec );      if( \*v < 0.0f || \*u + \*v > det )          return FALSE;        /\*计算t,并把t,u,v放缩为合法值（注意前面的t,v,u不同于算法描述中的相应量，乘了一个系数det）,注意：由于该步运算需要使用除法，所以放到最后来进行，避免不必要的运算，提高算法效率\*/      \*t = D3DXVec3Dot( &edge2, &qvec );      FLOAT fInvDet = 1.0f / det;      \*t \*= fInvDet;      \*u \*= fInvDet;      \*v \*= fInvDet;        return TRUE;  } |

**2.2.3  拾取完成根据获得的中心坐标计算我们关心的常见量**

根据重心坐标（U,V）,我们可以很容易的算出各种相关量比如纹理坐标和交点的差值颜色，假设以纹理坐标为例设V1,V2,V3的纹理坐标分别为T1(tu1,tv1),T2(tu2,tv2),T3(tu3,tv3)则交点的坐标为

IntersectPointTexture = T1 + U(T2-T1) + V(T3-T1)

### 3、结束及声明

　　Ok, 到这里为止关于拾取的相关知识就介绍完了，小弟第一次写这种文章，不知道有没有把问题说清楚，希望对大家有所帮助，有任何问题可以给我发email: jzhang1@mail.xidian.edu.cn

　　或者到我的网站留言： www.heavysword.com

　　声明：

　　本文写作的目的是为了广大D3D学习者方便学习服务，文中算法为作者参考相关文献总结，作者无意把这些据为自己的成果，所有权原算法提出者所有（参阅参考文献），文中代码为D3d SDK的示例内容，由笔者进行了必要的解释，代码版权归microsoft所有。

### 4、参考文献

　　【1】Microsoft DirectX 9.0 SDK,microsoft

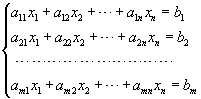
　　【2】fast,Minimun Storage Ray/Triangle Intersection,Tomas Moler,Ben Trumbore

 BY[linklogo](http://www.heavysword.com/)

# 克莱姆（Cramer）法则

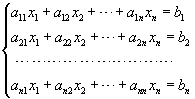
## 一、线性方程组

image004[1]元线性方程组是指形式为：

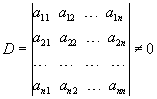
　　　　（1）

的方程组，其中image008[1]代表image010[1]个未知量，image012[1]是方程的个数，image014[1]，image016[1] ; image018[1]称为方程组的系数，image020[1]称为常数项。  
　　线性方程组的一个解是指由image021[1]个数image023[1]组成的有序数组image025[1]， 当image026[1]个未知量image028[1]分别用image029[1]代入后，式（1）中每个等式都成为恒等式。方程组（1）的解的全体称为它的解集合，如果两个线性方程组有相同的解集合，就称它们是同解方程组。  
　　为了求解一个线性方程组，必须讨论以下一些问题：  
　　(1).这个方程组有没有解？  
　　(2).如果这个方程组有解，有多少个解？  
　　(3).在方程组有解时,解之间的关系,并求出全部解。  
　　本节讨论方程的个数与未知量的个数相等(即image031[1])的情形。

**二、克莱姆法则**  
  
　　**定理**1（克莱姆法则）如果线性方程组

　            （2）

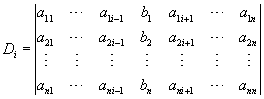
的系数行列式：



那么这个方程组有解，并且解是唯一的，这个解可表示成：

image037[1]　　　　（3）

其中image039[1]是把image041[1]中第image043[1]列换成常数项image045[1]所得的行列式，即

image049[1]。

　　分析：定理一共有3个结论：image051[1]方程组有解；image053[1]解是唯一的；image055[1]解由公式（3）给出。因此证明的步骤是：

　　第一，把image057[1] 代入方程组，验证它确实是解。这样就证明了方程组有解，并且（3）是一个解，即证明了结论image059[1]与image061[1]。

　　第二，证明如果image063[1]是方程组（２）的一个解，那么一定有image065[1]。这就证明了解的唯一性，即证明了结论image067[1]。

　　证明：先回忆行列式的一个性质，设image068[1]阶行列式image070[1]，则有：

image072[1]image074[1]

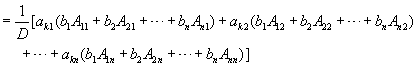
接下来证明定理。首先，证明（3）确实是（2）的解。将行列式image075[1]按第image077[1]列展开得：

image079[1]，

其中image081[1]是行列式image082[1]中元素image084[1]的代数余子式image086[1]。现把image088[1]

代入第image090[1]个方程的左端，得：

image092[1]image094[1]



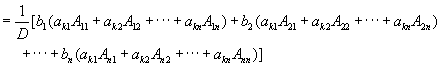


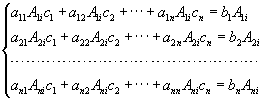
image100[1]

这说明将（3）代入第image102[1]个方程后，得到了一个恒等式，所以（3）是（2）的一个解。

　　其次，设image104[1]是方程组（2）的一个解，那么，将image106[1]代入（2）后，得到image107[1]个恒等式：

　　　　　　（4）

用系数行列式的第image111[1]列的代数余子式image113[1]依次去乘（4）中image114[1]个恒等式，得到：



将此image117[1]个等式相加，得：

image119[1]

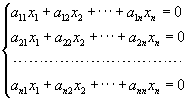
从而有：image121[1]。这就是说，如果image123[1]是方程组（2）的一个解，那么一定有image125[1]，所以方程组只有一个解。

## 三、齐次线性方程组

　　在线性方程组中，有一种特殊的线性方程组，即常数项全为零的方程组，称为齐次线性方程组。显然，齐次线性方程组总是有解的，因为image127[1]就是它的解，这个解称为零解；其他的，即image129[1]不全为零的解（如果还有的话），称为非零解。所以，对于齐次线性方程组，需要讨论的问题，不是有没有解，而是有没有非零解。这个问题与齐次线性方程组解的个数是有密切关系的。如果一个齐次线性方程组只有零解，那么这个方程组就只有唯一解；反之， 如果某个齐次线性方程组有唯一解， 那么由于零解是一个解，所以这个方程组不可能有非零解。

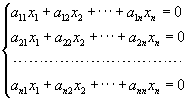
　　对于方程个数与未知量个数相同的齐次线性方程组，应用克莱姆法则，有

　　推论1  如果齐次线性方程组

　　　　（5）

的系数行列式不等于零，那么（5）只有零解。

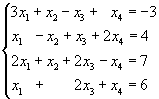
**推论**2　齐次线性方程组



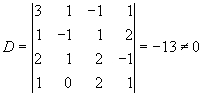
有非零解的必要条件是它的系数行列式等于零。

## 四、例子

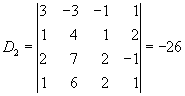
**例1**　解线性方程组image135[1]

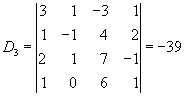
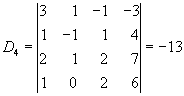


　　解：方程组的系数行列式：



所以根据克莱姆法则，这个线性方程组有唯一解。又因

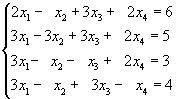
　

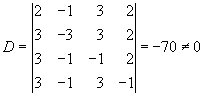
所以这个线性方程组的唯一解为：

image149[1]

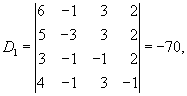
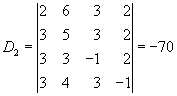
**例2**　解线性方程组

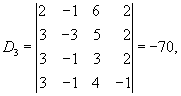
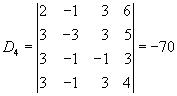


　　解：方程组的系数行列式：



所以根据克莱姆法则，这个线性方程组有唯一解。又因

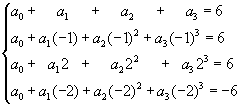
　

所以这个线性方和组的唯一解为：

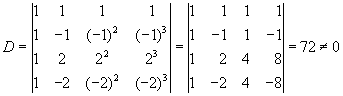
image163[1]

**例3**         已知三次曲线image165[1]在四个点image167[1]处的值分别为：image169[1]，试求其系数image171[1]。

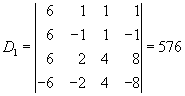
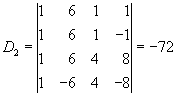
**解：**将三次曲线在4点处的值代入其方程，得到关于image173[1]的线性方程组：

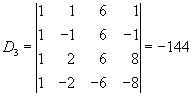
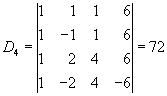


它的系数行列式是范德蒙行列式：



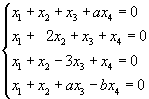
所以根据克莱姆法则，这个线性方程组有唯一解。又因

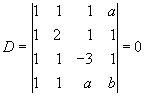
所以image187[1]，即所求的三次曲线方程为image189[1]。

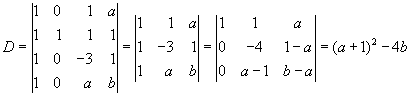
**例4**　如果齐次线性方程组



有非零解，那么image193[1]必须满足什么条件？

　　解：由克莱姆法则知，齐次线性方程组有非零解的必要条件是其系数行列式等于零，因此有



又由：，从而image198[1]必须满足的条件为image200[1]。

**注**用克莱姆法则求解系数行列式不等于零的image201[1]元非齐次线性方程组，需要计算image203[1]个image204[1]阶行列式，它的计算工作量很大。实际上关于数字系数的线性方程组（包括系数行列式等于零及方程个数和未知量个数不相同的线性方程组）的解法，一般都采用后续章节介绍的方法来求解。克莱姆法则主要是在理论上具有重要的意义，特别是它明确地揭示了方程组的解和系数之间的关系。

3D  
三维（three dimension）。客观世界中静止的物体都是三维的，在计算机图形学中常在一定的坐标系中用（x，y，z）坐标系列表示物体。  
  
3D modeling  
3D建模。用三维坐标来描述物体的形状。在各种计算机图形应用领域中有不同的三维建模方法，用不同的算法来描述这些领域中的物体和对象。  
  
3D transformation  
3D变换。在三维空间中把物体的三维坐标从一个位置变换至另一位置，或者从一个坐标系变换至另一坐标系。这是一种对物体的三维坐标（x，y，z）进行数据操作的一种形式。  
  
3D transformation sequence  
3D变换序列。把客观世界中的物体在计算机屏幕上显示，通常需要进行一系列坐标变换，如从物体的相对坐标系变换至计算机屏幕需要经过平移、旋转、视点投影变换等一系列坐标变换。  
  
4D  
四维（four dimension）。在计算机图形学中描述客观世界除了用三维坐标来描述物体的形状外，还用时间t作为第四维来描述过程，通常用（x，y，z，t）表示。  
  
6D  
六维（six dimension）。在计算机图形学三维应用过程中（例如：模拟仿真、虚拟现实应用等）用六个自由度（x，y，z坐标、偏角、倾角、仰角）来描述物体的运动。  
  
a stream  
一段流  
  
AABB  
沿坐标轴的包围盒（axis-aligned bounding boxes）  
  
ABI  
二进制接口  
  
Absorption  
吸收  
  
Accumulation Buffer  
累积缓存。累积缓存同颜色缓存一样也保存颜色数据，但它只保存RGBA颜色数据，而不能保存颜色索引数据（因为在颜色表方式下使用累积缓存其结果不确定）。这个缓存一般用于累积一系列图像，从而形成最后的合成图像。利用这种方法，可以进行场景反走样操作。  
  
action mapping  
动作映射  
  
algorithm  
算法。一般指在用电脑软件解决问题时所用的数学方法或程序实现过程。通常用数学公式或程序框图来描述。  
  
aliasing  
走样。在进行渲染着色的时候，所绘制的图元直接按照所赋给的颜色绘制象素，就会产生锯齿状的边，这就是走样。  
  
alpha blending  
阿尔法混合。一种让图像产生透明感的技术。  
  
alpha channel  
alpha通道  
  
alpha color component  
alpha颜色组件  
  
alpha constant  
alpha常量  
  
alpha value  
阿尔法值。颜色成分中的第四个成分。Alpha值不用于直接显示，一般用于颜色的混合。  
  
Ambient Light  
环境光  
  
analog  
模拟  
  
animated deformation  
动态自由变形  
  
AntiAliasing  
反走样（anti-aliasing）。根据图形单元的象素覆盖区域来确定象素的颜色值，这种渲染绘制技术就是反走样。  
  
APIs  
应用程序编程接口  
  
application specific clipping  
特定裁剪应用。图形单元在视点坐标系的平面上进行。  
  
ARB  
体系结构评审委员会（Architecture Review Board）  
  
Architecture Review Board  
体系结构评审委员会（ARB）  
  
area fill  
区域填充。即用指定的颜色和模式填充多边形或三角形区域围绕区域的线条即为边界。这在计算机图形学中是基本的绘图方式。  
  
articulated motion  
关节运动。一个与其它三维图形相连接的部件的运动。这种运动在动画或模拟中经常用到。  
  
AxDF  
轴变形（axial Deformation）  
  
axial Deformation  
轴变形（AxDF）  
  
axis aligned bounding boxes  
沿坐标轴的包围盒（AABB）  
  
B spline  
B-样条  
  
back face  
背面。多边形的一侧。多边形有正面和背面这二个面，在视窗中只有一个面是可见的，这个可见的面是正面还是背面在多边形投影到视窗后而定，若多边形的边为顺时针方向，其中的一个面可见，若多边形的边为逆时针方向，则另一个面可见，时针方向对应于正面还是反面由编程者而定。  
  
background color  
背景颜色。整个图形的底色。  
  
behavior  
行为  
  
binary file  
二进制文件。以二进制格式存储的文件，它的格式与ASCII格式（文本）相对立，其存储空间相对较小。  
  
bit  
位。二进制数，位作为状态变量只有0或1二个可能的值。二进制数可由一个或多个二进制位组成。  
  
bitmap  
位图。一种在显示内存或常规内存中字节的矩阵存储方式。OpenGL中对位图的显示用glBitmap()函数绘制。  
  
bitplane  
位平面。计算机屏幕上所有象素的字节的某一位组成的矩阵称为一个位平面。位平面用于驱动视频输出，也称为颜色平面。帧缓冲区就是由一组位平面组成的。  
  
Blend  
过渡运算  
  
blending  
混合。把二种颜色减少成为一种颜色，通常混合后的颜色由这二种颜色线性内插得到。  
  
Body  
体。它是面的并集。  
  
Boundary Representation  
边界表示（BRep）。它是几何造型中最成熟、无二义的表示法。  
  
bounding box  
包围盒  
  
bounding volume  
包围盒  
  
bounding volume hierarchy  
包围盒层次  
  
BRDF  
bi-directional reflectance distribution function的缩写。它的概念是，不论你观看的角度如何，让物体表面显现的不同材质看起来真切。  
  
BRep  
边界表示（Boundary Representation）。它是几何造型中最成熟、无二义的表示法。  
  
buffer  
缓冲区。用于临时存储数据和图象的一块内存区域。在计算机图形学中通常指存储色彩的单个成分（如红色、绿色)或单个索引（如颜色索引或模板索引）的一组位面。存储R、G、B、A四个值的缓冲区称为颜色缓冲区。  
  
bump mapping  
凹凸贴图  
  
BYPASSED  
旁路，忽略  
  
CAD  
计算机辅助设计。用计算机软件命令代替绘图工具并用计算机屏幕代替绘图板，而且能够用计算机三维图形显示所设计的工程部件。这种方式大大提高了工程设计者的效率。  
  
CALLBACK  
回调  
  
callback function  
回调函数。在应用程序中处理系统功能的函数。通常在不同系统（如与外部设备交互）间交流时用到回调函数。  
  
CAPP  
计算机辅助工艺过程设计  
  
change with time  
逐时变化  
  
client  
客户。发出OpenGL命令的计算机。这个计算机可以通过网络与其它执行这些命令的计算机相连，或者在同一个计算机上发布和执行命令。  
  
clip coordinates  
裁剪坐标。这是指在投影变换后在透视变换前的坐标系统。视窗体的裁剪在裁剪坐标系下进行，但应用特定裁剪不是在此坐标系下进行。  
  
clipping  
剪 裁。将超出裁剪面所定义的视窗边界的图形部分删除。如果点在外面，就简单地拒绝；如果线或多边形在外面，在外面的部分被删除，并且按照需要补充的新的顶点 来完成裁剪边界内的图形单元。图形单元总是在视窗体的left、right、bottom、top、near、far平面所定义的六个面上裁剪。特定的应 用程序可以在视点坐标系中应用特定的裁剪面。  
  
CMY model  
CMY模型。这是一种以黄色、品红色、青色作为基本色的颜色模式，这种模式主要用于印刷。  
  
collision detection  
碰撞检测。指在运动时检测一个3D物体的边角或面是否与另一个3D模型发生碰撞，从而决定在检测到碰撞后所采取的动作。  
  
Color Buffer  
颜色缓存。颜色缓存通常指的是图形要画入的缓存，其中内容可以是颜色索引，也可以是RGB颜色数据（包含Alpha值也可）。  
  
color cycling  
颜色循环。通过转换调色板的值来产生动画的方法。  
  
color index  
颜色表。又称颜色索引，通过命名而不是通过颜色值来表示颜色的一个值。OpenGL的颜色索引用连续的数值表示（如浮点数），颜色的插值和抖动等操作可以用颜色索引值来计算。但是，保持在帧缓冲区中的颜色索引值总是整数值，浮点数颜色索引总被舍入到最近的整数值。  
  
color index mode  
颜色表模式。如果颜色缓存里存储的是颜色索引值而不是红、绿、蓝和alpha成分，这种OpenGL的颜色存储形式即为颜色表模式。  
  
color interpolation  
颜色插值。通过一个象素的相邻颜色来决定它的颜色，或当两象素的颜色值已知时，通过当前象素与这两个象素的距离来决定这个象素的颜色。  
  
color map  
颜色映射表。颜色索引映射到RGB值所对应的表，它由显示硬件访问。每个颜色索引值从颜色缓存中读出，通过查找颜色映射表转换成RGB值，并送给显示器。  
  
component  
成 分。又叫分量，表示强度或数量的一个数值（如浮点数），通常0表示最小的值或强度，1表示最的值或强度，成分一般在一个范围内解释。如颜色的三个成分 RGB的值（1，1，1）表示为白色。超出范围的成分一般都被归一到标准的成分范围内，如RGB三成分值（1.4，1.5，0.9）在送入缓冲区前被修改 为（1.0，1.0，0.9）。红、绿、蓝、alpha值和深度值总是作为成分而不作为索引。  
  
computer animation  
计算机动画。由计算机产生的许多帧景物图象,并平滑地改变视点或景物中物体的位置，当显示这些图象的速度足够快时就可获得景物的动态效果（如每秒显示24帧图象即为普通电影播放的效果）。  
  
computer graphics  
计 算机图形。用计算机产生的图像。计算机图形学始于六十年代初期，现今的专业计算机图形公司Evens & Sutherland的共同创业者Ivan Sutherland在MIT.攻读博士学位期间为TX-2计算机开发了一个绘制草图的程序。从此以来，计算机图形已发展到如今已有能力产生与照片中捕获 的图像完全没有区别的逼真的图像。  
  
computer graphics simulation  
计算机图形模拟。利用计算机图形来模拟客观世界的现象。近年来发展起来的计算机三维图形处理硬软件使得计算机模拟已涉及于广泛领域，如飞行模拟、汽车模拟、军事训练模拟等。  
  
computer image  
计算机图象。一个由图象单元（象素、象元）组成的矩阵数组,每个图象单元都有其色彩属性，这个带色属性的图象单元数组表示图象的内容。  
  
computer stereo graphics  
计算机立体图形。利用人眼视觉感受客观物体的原理生成二幅具有视差的图象，并且通过某种方式（如红绿图象对或偏振光阀眼镜等）观察就可获得具有深度的立体景物图形。  
  
Construction points  
构造点  
  
Constructive Solid Geometry  
构 造几何法（CSG）。它是三维立体造型技术的一种基本方法。该技术通过布尔运算（交、并、差）将基本体素以一定的顺序来构造出复杂的形体。整个造型过程则 简单地表示为一棵二叉树，我们称之为CSG树。该树的叶结点表示参与构造的一些基本体素，中间结点表示布尔运算，根结点即为所构造的物体。  
  
context  
上下文。[1]指一个完整的OpenGL状态集。需要注意的是，帧缓冲区的内容不是OpenGL状态的一部分，但帧缓冲区的设置则是。  
  
convex  
凸。如果多边形所在平面内的直线与多边形的交点不超过而个，这个多边形就是凸多边形。  
  
convex hull  
凸包  
  
coordinate system  
坐标系。在n维空间中，把n个线性独立的向量固定在一点（原点），点的坐标值通过原点到这个点的矢量距离表明了这个点在空间中的位置。  
  
Core Frequency  
核心频率  
  
Crossbar Memory Controller  
交错型内存控制器  
  
CSG  
构 造几何法（Constructive Solid Geometry）。它是三维立体造型技术的一种基本方法。该技术通过布尔运算（交、并、差）将基本体素以一定的顺序来构造出复杂的形体。整个造型过程则 简单地表示为一棵二叉树，我们称之为CSG树。该树的叶结点表示参与构造的一些基本体素，中间结点表示布尔运算，根结点即为所构造的物体。  
  
culling  
拣选。删除多边形的一个向前的面或向后的面，从而使这个面不被画出，这个过程就是拣选。  
  
current matrix  
当 前矩阵。把一个坐标系中的坐标变换到另一个坐标系中坐标的矩阵。在OpenGL中有三个当前矩阵：模型观察(modelview)矩阵把物体坐标变换到视 点坐标；透视矩阵把视点坐标变换到裁剪坐标；纹理矩阵则变换指定的或自动产生的纹理坐标。每个当前矩阵都是矩阵堆栈的栈顶元素。每个矩阵堆栈都可以用 OpenGL的矩阵操作函数来处理。  
  
current raster position  
当前光栅位置。指当图象光栅化时，用与说明图象位置的窗口坐标。OpenGL中当调用glRasterPos()时，当前的光栅位置和其他当前光栅参数就被修改。  
  
DDA  
数值微分法。  
  
DDR  
double data rate，双倍速数据传输  
  
degree of freedom  
自由度。由虚拟现实人工交互设备(如三维鼠标、空间球、头盔式跟踪器等)提供的输入值的坐标和旋转，即x、y、z值和偏角、仰角、倾角。  
  
demultiplexer  
信号分流器  
  
depth  
深度。一般指方向为从屏幕指向观察者的z轴的窗口坐标。  
  
Depth Buffer  
深 度缓存。深度缓存保存每个象素的深度值。深度通常用视点到物体的距离来度量，这样带有较大深度值的象素就会被带有较小深度值的象素替代，即远处的物体被近 处的物体遮挡住了。深度缓存也称为z-buffer，因为在实际应用中，x、y常度量屏幕上水平与垂直距离，而z常被用来度量眼睛到屏幕的垂直距离。  
  
depth cuing  
深度暗示。根据到视点的距离来决定颜色值的绘制技术。  
  
derived classes  
导出类  
  
developable surface  
可展表面  
  
Diffraction  
衍射  
  
Digital Terrain Model  
数字地面模型（DTM）。它的可视化是地学、仿真等领域的关键技术。  
  
DirectX Video Acceleration  
directx 视频加速  
  
Discreet  
Discreet中国公司 <http://discreet.com.cn/>  
  
displacement mapping  
置换式贴图。这技术借助在平面的多边形上加上一些数据，可以帮材质加上深浅高低的轮廓视觉效果。不妨参考一下这项简单的技术定义：[http://www.cs.berkeley.edu/~jfc/cs184f98/lec32/lec32.html](http://www.cs.berkeley.edu/%7Ejfc/cs184f98/lec32/lec32.html)  
  
display list  
显 示列表。一个OpenGL的命令集的记录表。显示表总是存储在服务器上，在客户 — 服务器环境走，显示表可以减少网络的传输量。显示表的内容可以被预处理，因此执行显示表比按立即方式执行同样一组OpenGL命令更有效。这样的预处理对 于glTextImage()等计算密集的命令特别重要。  
  
dithering  
抖动。一种以牺牲图象的空间分辨率来增加图象颜色的视觉范围的技术。图象中相邻的象素被赋给不同的颜色值。在一定的距离观察时，这些颜色似乎被混合成一个中间颜色。这种技术类似与半色调技术，半色调技术用于黑白出版物中获得灰色的明暗效果。  
  
Dot Product Bump Mapping  
点积凹凸贴图。它所用的方法是对三角形上的每个像素赋予假的法线，因此反射不是按照“真正”的平面三角形计算出来的，而是根据法线图表面的向量计算出来。结果生成了凹凸贴图的效果，让一块并非“真实”凹凸的区域看起来有3D的感觉。  
  
double buffer animation  
双缓存动画。这种方法指当正在应用程序窗口播放当前帧时，程序已在另一缓冲区生成了下一帧图象数据，当前一帧图象显示完毕时，应用程序只要将在另一缓冲区已经生成的图象拷贝到屏幕上即可。这种方式可加快动画的播放速度，但需要较大内存。  
  
double data rate  
双倍速数据传输  
  
downstream  
下级  
  
driver  
驱动(程序)  
  
DTM  
数字地面模型（Digital Terrain Model）。它的可视化是地学、仿真等领域的关键技术。  
  
Dynamic Linking Library  
动态连接库。它是在程序运行时才将库中的函数连接至应用程序，所以是动态的。DLL模块可以通过编程工具经过编译连接建立。  
  
Edge  
边。它是两个邻面(对正则形体而言)、或多个邻面(对非正则形体而言)的交集，边有方向，它由起始顶点和终止顶点来界定。  
  
edge use  
边引用  
  
effect  
效果  
  
Electronic Program Guide  
电子程序向导  
  
element  
元素。一个单独的成分或索引。  
  
embedding  
嵌入  
  
emulation  
冲突  
  
encapsulate  
封装  
  
environment bump mapping  
环境景观贴图  
  
evaluation  
算子运算。从预先指定的Bezier方程中产生坐标顶点和参数的OpenGL的过程。  
  
execute  
执行。[1]当用立即方式调用OpenGL命令或当显示表的一部分被调用时，一个OpenGL的命令即被执行。  
  
extrusion  
拉伸。传统自由曲线曲面设计方法的一种。  
  
eye coordinates  
视点坐标。在模型观察矩阵变换之后和在投影矩阵变换之前的坐标系。光照和应用特定裁剪都在视点坐标系中进行。  
  
face  
面。见背面（back face）。  
  
face use  
面引用  
  
Facial Animation  
面部表情。为3D模型塑造出来的角色赋予人类的面部表情使之给人更真实的感觉。对于游戏来说，面部表情的出现也同时象征着游戏制作开始真正踏出终极目标的一步，使游戏中的人物具有感情。  
  
factor curve  
因子曲线  
  
FDH  
固定方向凸包（fixed direction hull）  
  
feature line  
特征线  
  
features  
特征  
  
FFD  
自由变形（Free-form Deformation）  
  
filter  
过滤器  
  
filter graph  
过滤器图  
  
filter mapper  
过滤器映射  
  
filtering  
过滤。用于消除锯齿状的颜色的方法。  
  
Fins  
翼  
  
fixed direction hull  
固 定方向凸包（FDH）。FDH是一种特殊的凸包（convex hull），它继承了凸包紧密性的优点，同时FDH也可以看作是AABB的扩展，它又具有AABB简单性的特点。FDH是具有许多优秀的特性，很适合用于 复杂环境中的碰撞检测问题的研究，特别是可变形的动态环境中的碰撞检测的研究。  
  
fixed function pipe  
固定功能管线  
  
flat shading  
平面着色。指用单一的、固定的颜色给图元着色，而不是在图元上光滑地插值颜色。  
  
Flexible Vertex Format  
可塑顶点格式（FVF）  
  
fog  
雾。一种模拟自然界的雾、烟等空气效果的渲染技术。这种技术根据到视点的距离逐渐把物体的颜色变淡至背景色。雾也有助于表现物体到视点的距离感，给出一个深度暗示。  
  
font  
字体。通常用于显示文本字符串的一组字符的图形表示。字符可以是罗马字母、数学符号、亚洲的表意文字、埃及的象形文字等。  
  
four matrix skinning  
四组矩阵贴皮  
  
fragment  
片元。片元是由图元的光栅化产生的。每个碎片相应与单独的一个象素并包括颜色、和深度，有时还包括纹理坐标值。  
  
frame  
帧。通常指视窗或整个屏幕的一幅图象。  
  
frame buffer  
帧缓冲区。用于显示图象和文本的视频缓冲区。一般指窗口或上下文的所有缓冲区，有时也包括图形硬件加速器的所有象素存储器。  
  
Free form Deformation  
自由变形（FFD）  
  
front face  
前面。见背面（back face）  
  
frustum  
视景锥台。一个被透视范围包围的视景体。  
  
FVF  
可塑顶点格式（Flexible Vertex Format）  
  
Game Developer Conference  
游戏开发者年会  
  
gamma correction  
gamma修正。一个修正帧缓冲区中的颜色的函数。为了修正眼睛或显示器对颜色的线性反应，需要对颜色的值进行线性变化，这就是gamma修正。  
  
GDC  
游戏开发者年会  
  
geometric model  
几何模型。指描述一个几何体的模型坐标顶点和参数。注意OpenGL不定义几何模型的语法，而是定义渲染绘制几何模型的语法和语义。  
  
geometric modeling  
几何造型  
  
geometric object  
几何对象。就是几何模型（geometric model）  
  
geometric primitive  
几何图元。指点、线或多边形。  
  
geometric processing  
几何处理。计算机图形学中的一些图形处理功能，如平移、旋转、剪裁等。  
  
GLAUX  
OpenGL辅助库（OpenGL utility library）。它包含了一些特殊的函数，例如简单的窗口管理、输入事件处理、某些复杂三维物体绘制等函数。  
  
GLUT  
OpenGL派生库（stands for OpenGL Utility Toolkit）。它是从OpenGL实用库派生出来的可独立于平台系统的程序库，在WWW.OpenGL.ORG里您可以获得更多关于它的信息。  
  
Gourand shading  
Gouraund着色法。计算机图形学中的一种光照模型算法，这种方法计算所得的阴影过渡比较平滑。一般在多边形或线段上光滑地颜色插值。在个顶点上指定颜色，在整个图元内进行线性插值，从而使得色彩的变化变得相对光滑。  
  
grabbing media samples  
媒体样例抓取  
  
graph searching formulation of dynamic programming  
基于动态规划图搜索算法  
  
group  
组。在客户内存中的图象的每个象素是由一、二、三或四个元素的组来表示的。因此，在客户内存图象的上下文中，组和象素是同一件事情。  
  
Guide  
向导  
  
half space  
半空间。一个平面把空间分成二个半空间。  
  
HANDLE  
通用句柄类型  
  
HardSubtract  
差运算  
  
HBRUSH  
标识一个刷子对象  
  
HCURSOR  
标识一个光标对象  
  
HDC  
标识一个设备对象  
  
Head Mounted Display  
即头盔式显示器（HMD）,是虚拟现实的一种外部设备。  
  
heat transfer  
传热学  
  
height surface  
高度曲面  
  
HFONT  
标识一个字体对象  
  
HGLOBAL  
标识一个全局内存对象  
  
HICON  
标识一个图标对象  
  
hiden face  
隐藏面。在三维图形模型中被其它表面所隐藏的表面（曲面或平面）。  
  
hiden face removal  
隐藏面删除。计算机图形学中将三维模型中的所有不可见的平面删除的过程。这种过程有多种算法。  
  
hiden line  
隐藏线。在三维图形模型中被其它图形单元所隐藏的线。  
  
Hierachical Occluding Map  
层次遮挡图（HOM）  
  
Higher Order Surfaces  
高维曲面。它的概念是仅用几个控制点来描述曲面。一个曲线或是弧面仅由几个控制点或是控制向量来定义，而用来描述这曲面形状的函数/公式就称为“Spline”。 有玩Photoshop或CorelDraw的朋友对“Spline”应该很熟悉。  
  
HINSTANCE  
标识一个应用程序模块的一个实例  
  
HLOCAL  
标识一个局部内存对象  
  
HLS model  
HLS模式。一种用色度、亮度、饱和度作为颜色的计算值的颜色表示模式。  
  
HMD  
即头盔式显示器（Head Mounted Display）,是虚拟现实的一种外部设备。  
  
HMENU  
标识一个选单对象  
  
HOM  
层次遮挡图（Hierachical Occluding Map）  
  
homogeneous coordinates  
齐 次坐标。用于表示n维透视空间的点的n+1维分量。透视空间的点可以被认为是欧几里德空间加上一些无穷远处的点。由于每个坐标分量乘以一个非零值都不改变 这些坐标所表示的点，这样的坐标是齐次坐标。齐次坐标主要应用在透视几何计算，因此在场景必须投影到一个窗口上的计算机图形学中也十分有用。  
  
homogenous clip space  
齐次裁剪空间  
  
HPEN  
标识一个笔对象  
  
HWND  
标识一个窗口对象  
  
illumination model  
光照模型。用于计算机生成真实感景物的一种计算机图形学算法。这种算法可真实地模拟自然界中的光影现象。  
  
image  
图象。象素的一个矩阵数组。这个象素矩阵可以在帧缓冲区内或者在客户内存内。  
  
image primitive  
图象图元。指一个位图或一个图象。  
  
image processing  
图象处理。指用一定的算法对图象进行分析、解释、修改等一系列过程。常用的处理过程有图象显示、图象增强、边缘检测、几何处理（拉伸、镜像等）、对比调整、剪裁和粘贴、模式识别、目标识别等。  
  
immdiate mode  
立即方式。[1]直接调用OpenGL命令执行而不是通过调用显示表。在OpenGL中，不存在立即方式位，立即方式中的方式是OpenGL的用法，而不是指OpenGL状态的一个特定的位。  
  
index  
索 引。指作为一个绝对值经内插得到的一个单独的数值，这个值不是在一个指定的范围内的规范化的值（这种值就是成分）。颜色索引就是颜色的名字，显示硬件用颜 色表把它们转换成颜色。索引在超出范围是一般是被屏蔽而不是被取舍。例如，索引值0xf7被写到4位缓冲区（颜色或模板）是屏蔽成0x7。颜色索引和模板 索引总是作为索引对待，从不作为成分对待。  
  
Infrared signature simulation  
红外仿真  
  
input loop  
输入循环  
  
interactive  
交互。计算机程序运行时响应用户的输入。  
  
interactive graphics  
交互式图形。在图形的操作过程中，用户的输入能够参与整个过程，包括生成、移动、修改等。  
  
Interference  
干涉  
  
Intersecdtion  
交运算  
  
Inverse kinetics  
反动力学方程  
  
IRIS GL  
Silicons Graphics公司专有的图形库，在1982到1992年之间开发。OpenGL的设计是以IRIS GL为起点的。  
  
isometric mapping  
等距映射  
  
Isotropic BRDF Based Lighting  
等向BRDF光源。 而BRDF是bi-directional reflectance distribution function的缩写。它的概念是，不论你观看的角度如何，让物体表面显现的不同材质看起来真切。  
  
jaggies  
锯齿。在走样渲染绘制时，多边形的边带有锯齿而不光滑。例如，一条几乎水平的直线被画成一组位于相邻象素行上的水平线，而不是光滑连续的线。  
  
key  
关键  
  
key frame  
关键帧。在计算机动画过程中起连接作用的帧，动画的其它帧在这些帧与帧之间进行内插实现。  
  
Key Frame Animation  
关键帧动画  
  
key frame interpolation  
画面间插补  
  
Key Framing  
关键帧插值  
  
legacy  
传统  
  
light mapping  
光影贴图。目前的游戏最常用的是光影贴图，以光纹理的形式把光源应用于三角形。  
  
Light Speed Memory Architecture  
光速内存架构  
  
Lighting  
光影  
  
line  
线。在两个顶点之间具有有限宽度的长条区域。OpenGL不同于数学意义上的线，它具有有限宽度和长度。线条的每一段就是一条线段。  
  
line pattern  
线模式。线的形式，是连续线或是各种断划线。  
  
line style  
线类型。线的形式，是连续线或是各种断划线。  
  
LOD  
层次细节（Level of Detail）。该技术就是在不影响画面视觉效果的条件下，通过逐次简化景物的表面细节来减少场景的几何复杂性，从而提高绘制算法的效率。  
  
Loop  
环。它是有序、有向边（Edge）组成的封闭边界。环中的边不能相交，相邻两条边共享一个端点。环有方向、内外之分，外环边通常按逆时针方向排序，内环边通常按顺时针方向排序。  
  
loop use  
环引用  
  
luminance  
亮度。一个表面的可见明亮程度。通常是指红、绿、蓝值的加权平均，它给出一个合成的视觉亮度。  
  
mapped  
映射  
  
mapper  
映射  
  
mapping  
映射  
  
matrix  
矩阵。数值的二维数组。OpenGL所有的矩阵都是4\*4矩阵，它在客户内存中以1\*16的一维数组的形式进行计算。  
  
Matrix Palette Skinning  
矩阵调色板变形  
  
Maximum Active Textures Per Pixel per Pass  
每周期每像素最大活动纹理  
  
Memory Interface Width  
内存接口带宽  
  
merit  
指标  
  
model  
模型。指与客观世界中相类似的形体。  
  
modeling  
建模。在三维空间中描绘客观世界中的物体，通常用细小的平面（甚至是三角形）逼近物体表面或者用一些关键点的坐标来精确地构造，这种用三维坐标构造物体表面的方法即为模型建立，简称建模。  
  
modelview matrix  
模型观察矩阵。把点、线、多边形和光栅位置从模型坐标变换到视点坐标的4\*4矩阵。  
  
monitor  
显示器。用于显示帧缓冲区中的图象的硬件设备。  
  
motion blur  
运动模糊。在传统的电影中，由于目标的移动快于摄影的帧速率，会留下物体的痕迹而引起图象的模糊，在计算机图形学中常使用有意识的模仿图象模糊的方法来实现某种特殊的效果。  
  
MSG  
消息结构  
  
multiple resolution spline  
多分辨率样条  
  
muscular vector  
肌肉向量  
  
network  
网络。允许在二个或多个计算机之间进行相互数据传输的连接。  
  
Network Provider  
网络提供者  
  
Neutura Net AI  
神经中枢网络人工智能。这是游戏使用人工智能的突破，它可以使得电脑控制的角色比以前任何时候都要好地模拟玩家行为。  
  
non photorealistic rendering  
非照相真实感图形绘制（NPR）。由于大多数NPR以模拟传统绘画工具的视觉效果为目的，所以NPR包括笔刷设计，笔刷使用以及笔刷与3D物体模型结合三方面内容。  
  
nonconvex  
非凸。如果在多边形的平面上存在一条直线，它与多边形的交点多与二个，则这个多边形为非凸多边形。  
  
normal  
法向量。用于定义平面或曲面的角方向的三成分的平面方程。  
  
normalize  
标准化。法向量的每个分量除以分量平方和的平方根。如果认为法向量是从原点到点（nx'，ny'，nz'）的向量，则这个向量具有单位长度。  
  
NPR  
非照相真实感图形绘制（non photorealistic rendering）。由于大多数NPR以模拟传统绘画工具的视觉效果为目的，所以NPR包括笔刷设计，笔刷使用以及笔刷与3D物体模型结合三方面内容。  
  
Number of Pixel Shading Operations per Pass  
每次每个像素运算的数目  
  
NURBS  
非均匀有理B-样条，这是一种指定参数曲线和曲面的公共方法。  
  
OBB  
带方向的包围盒（oriented bounding box）  
  
object  
模型。在计算机三维图形编程中指组成三维景物的图形模型。它作为一组图元来绘制。  
  
object coordinates  
模型坐标。在任何OpenGL变换之前的坐标系。  
  
Object Modeling Technique  
对象模型技术（OMT）  
  
Octree Space Partitioning  
八 叉树分割算法（OSP）。它通常应用于表示处于3D环境中的固定物件。同时，它亦是很多模型和渲染系统的基础。八叉树分割的重要目的是减少在场景的光线追 踪、碰撞检测、可视性判断及其类似过程中所需要的比较次数。尽管它通常主要应用于静态的环境，但只要稍作修改，它就可以被应用到动态的环境下。八叉树可以 有效地减少在正确显示场景过程中多边形排序所需的时间。同时，它非常适合于那种在空旷的空间内拥有若干体积差别极其悬殊的物件的高性能游戏（比如飞行模拟 游戏）。  
  
OM  
对象模型技术（Object Modeling Technique）  
  
OOP  
即为面向对象的编程（object oriented programming）。这种方法的特点是程序的每个模块都含有数据和操纵这些数据的函数。  
  
Open Inventor  
一个基于OpenGL的三维图形开发工具包，它提供了大量的图形类，从而使三维图形开发十分容易。与OpenGL一样，OpenInventor除了工作站版本外，也有微机版本，目前的微机版本可在Windows NT3.5以上环境下运行。  
  
oriented bounding box  
带方向的包围盒（OBB）  
  
orthographic  
平行投影。非透视投影，它可用于没有透视变化的工程绘图。  
  
OSP  
八 叉树分割算法（Octree Space Partitioning）。它通常应用于表示处于3D环境中的固定物件。同时，它亦是很多模型和渲染系统的基础。八叉树分割的重要目的是减少在场景的光 线追踪、碰撞检测、可视性判断及其类似过程中所需要的比较次数。尽管它通常主要应用于静态的环境，但只要稍作修改，它就可以被应用到动态的环境下。八叉树 可以有效地减少在正确显示场景过程中多边形排序所需的时间。同时，它非常适合于那种在空旷的空间内拥有若干体积差别极其悬殊的物件的高性能游戏（比如飞行 模拟游戏）。  
  
overlay layer  
覆盖层。计算机图形硬件中的一个特殊缓冲区，在这个缓冲区进行图形操作不会影响视频缓冲区中的图形。  
  
PAINTSTRUCT  
绘图结构  
  
pan  
漫游。指计算机图象平滑地左右移动。  
  
parameter  
参数。作为参变量传给OpenGL命令的值。有时通过参照一个OpenGL命令来传递。  
  
parametric surface  
参数曲面  
  
parent class  
父类  
  
Particle System  
粒子系统  
  
pending  
未决的  
  
per pixel lighting  
单独像素光源  
  
Per Vertex Motion Blur  
个别顶点残影。依照移动向量的长度（物体的移动速度），顶点上的alpha值逐渐降低，让它看起来是渐渐的透明模糊化。你不得不承认，这看起来很真实。  
  
perspective division  
透视除法。坐标x，y，z除以w，它在裁剪坐标系中执行。  
  
photorealistic rendering  
照相真实感图形绘制  
  
PIN  
/\* 针脚 \*/  
  
pipeline  
管道线。在计算机三维图形显示过程中的一种图形输出方式，多个管道线常用于同一视窗显示不同图形的三维图形模拟，如模拟汽车驾驶时同时显示汽车前面景物和倒后镜中的景物就需用到多个管道线，SGI的Performer等软件支持这种图形操作。  
  
pixel  
象素。即图形元素，一个象素是在计算机屏幕上可寻址的最小图形单元，其颜色是由红绿蓝的混合色。在帧缓冲区中的所有位面在（x，y）的位组成一个象素（x，y）。在OpenGL的窗口坐标系中，每个象素对应于一个1.0\*1.0的屏幕区域。  
  
Pixel Pipelines  
像素流水线  
  
point  
点。空间中的一个确切的位置，它作为一个具有有限直径的点来绘制。  
  
polygon  
多边形。指一个封闭的由顶点指定的边围起来的多边实体图形。它是计算机图形学中最基本的图形单元。三角形网中的每个三角形，四边形网中的每个四边形都是一个多边形。在OpenGL中的用glRect\*()绘制的矩形也是一个多边形。  
  
polymorphism  
多态性  
  
Potentially Visible Set  
PVS  
  
primitive  
基本图元。指一个点、一条线、一个多边形、一幅位图或图象。注意基本图元不仅仅是一个点、一条线、或一个多边形。  
  
Procedural Deformation  
程序变形。它意谓着在一般正常物体的表面上，如平板，通过特殊的程序改变形状。这种变形可以是动态的（大部分），也可以是静态的。典型的例子是水波，风中飘扬的旗帜，或是（静态的）被射中的金属物体。  
  
Process  
制程  
  
Programmable Pixel Shader  
可编程像素着色引擎  
  
projection matrix  
投影矩阵。把点、线、多边形和光栅位置从视点坐标变换到裁剪坐标的4\*4矩阵。  
  
Purity  
纯度  
  
PVS  
Potentially Visible Set  
  
Radial Edge  
辐射边  
  
radiocity  
光传导渲染法  
  
rasterize  
光栅化。把经过投影变换的点、线、多边形、位图或图象的象素转换成碎片，每个碎片对应于帧缓冲区中的象素，这个过程就是光栅化。光栅化时所有的图元都被光栅化，不仅仅是点、线、和多边形。  
  
ray tracing  
光线跟踪。从光源到光所照的物体，再从这物体反射到其它物体，这样一直跟踪下去直到光源减弱到某一阀值为止。这种通过跟踪光线来模拟光的作用的方法可以逼真地显示三维景物。  
  
real time animation  
实时动画。在计算机图形学中常指二种情形：一种是指在计算机图形软件运行时生成并显示动画序列；二是指在计算机图形软件运行时同时响应外部事件的动画序列。  
  
RECT  
矩形结构  
  
rectangle  
矩形。即平行四边形，是在模型坐标系中对边平行的四边形。在OpenGL中用glRect\*()绘制的多边形总是矩形。  
  
Reflection  
反射  
  
Refraction  
折射  
  
Refraction Effects  
折射效果。一般人第一个想到的，大概就是各种透镜的效果，如这个鱼眼镜的例子。其实，单是一片厚厚的玻璃，或是装水的玻璃杯，也有同样的效果。热空气也能折射光线，赛车中就很典型。只要你顶点运算程序写的漂亮，顶点着色引擎就能够模拟各种折射效果。  
  
refresh buffer  
刷新缓冲区。即由计算机硬件实现显示视频缓冲区的图形内容。  
  
renderer filter  
变换过滤器  
  
rendering  
渲染。也称为着色、真实感图形显示。它指在计算机完成建模后，用光照模型和消隐处理算法在物体表面加入颜色、阴影、亮度、纹理等表面属性，从而使得整个景物模型更加逼真。  
  
revolusion  
旋转。传统自由曲线曲面设计方法的一种。  
  
RGB model  
RGB模式。主要用于计算机图形显示的红绿蓝颜色模式。  
  
RGBA  
指颜色的R，G，B，Alpha值。  
  
RGBA model  
RGBA模式。如果颜色缓冲区内保存的是r，g，b，alpha值，而不是颜色索引，则OpenGL的上下文是RGBA模式。  
  
rotation  
旋转。指计算机图形变换中三维图形单元的偏角、仰角、倾角的变换。  
  
Scalability  
伸缩性。新的多重解析度解决技术。该技术可以使设计者创造出更具说服力的3D世界以及弥补因为高/低配置电脑系统性能不同而导致游戏画面表现不同的差异问题。  
  
Scissor Planes  
剪 裁平面（SP）。这并不算是个为游戏而引进的特色，而是给较复杂的应用程序使用，它将需要硬件支持像素矩形的修剪。这就是它的名称的意义，把物体放在视野 中，然后裁掉看不到的部分。这对于开启多重窗口和剪辑视讯而言有很大的帮助。而这一特性在OpenGL中必须要有某种程度的精确性和硬件支持才能发挥得很 好。OpenGL剪裁和模版印刷功能通常用来停止OpenGL对部分屏幕画面的写入动作。DirectX发展到9.0才支持剪裁平面。  
  
server  
服务器。执行OpenGL命令的计算机。它可能与发布OpenGL命令的计算机不同。  
  
shading  
着色。也称阴影、浓淡，指给三维景物模型加上光照效果、阴影和色彩，使得整个三维景物具有逼真的效果。在Ope  
nGL中一般是在光栅化的过程中，用插值方法获得多边形内部或线段顶点之间的颜色。  
Shadow Mapping|阴影贴图  
  
shape surface  
形状曲面  
  
single buffering  
单缓冲。指没有后台颜色缓冲区的OpenGL上下文是单缓冲的OpenGL上下文。这种方式可以用于动画，但必须注意避免绘制时的闪烁。  
  
single instruction multiple data  
SIMD，单指令多数据  
  
singular matrix  
奇异矩阵。指没有逆的矩阵。这样的矩阵代表一个变换，把至少沿着一条直线的点退化成一个点。  
  
skinning  
蒙皮。传统自由曲线曲面设计方法的一种。  
  
Smart Shading  
智能光源处理  
  
snake  
Snake算法  
  
SoftSubtract  
差运算  
  
solid model  
实体模型。在已经经过隐藏线消隐处理的三维景物模型。  
  
source  
源  
  
SP  
剪 裁平面（Scissor Planes）。这并不算是个为游戏而引进的特色，而是给较复杂的应用程序使用，它将需要硬件支持像素矩形的修剪。这就是它的名称的意义，把物体放在视野 中，然后裁掉看不到的部分。这对于开启多重窗口和剪辑视讯而言有很大的帮助。而这一特性在OpenGL中必须要有某种程度的精确性和硬件支持才能发挥得很 好。OpenGL剪裁和模版印刷功能通常用来停止OpenGL对部分屏幕画面的写入动作。DirectX发展到9.0才支持剪裁平面。  
  
spheres  
球形包围盒  
  
Stencil Buffer  
模 板缓存。模板缓存可以保持屏幕上某些部位的图形不变，而其它部位仍然可以进行图形绘制。比如说，可以通过模板缓存来绘制透过汽车挡风玻璃观看车外景物的画 面。首先，将挡风玻璃的形状存贮到模板缓存中去，然后再绘制整个场景。这样，模板缓存挡住了通过挡风玻璃看不见的任何东西，而车内的仪表及其它物品只需绘 制一次。因此，随着汽车的移动，只有外面的场景在不断地更改。  
  
stipple  
点画式样。指一个一维或二维的二进制图案，它在值为0的地方不产生一个碎片，线的点画式样是一维的并且在线的始点开始应用。多边形的点画式样是二维的并按一个固定的方向在窗口中应用。  
  
Subclass  
子类  
  
Surface  
表面  
  
surface model  
表面法线。在三维图形环境中垂直于景物模型表面的一条直线，通过计算表面法线和入射光线的夹角可得到光照程度。  
  
Surface reconstruction  
曲面重建  
  
sweeping  
扫荡。传统自由曲线曲面设计方法的一种。  
  
tessellation  
[1] 用“Spline”曲面来做出三角形的过程称为镶嵌。  
  
texel  
纹素。类似于texture pixel的合写，指一个纹理元素。纹素可从纹理中得到，并代表应用到碎片上的纹理的颜色。  
  
texture  
纹理。用于修改光栅化产生的碎片的颜色的一维或二维的图象。  
  
texture mapping  
纹理映射。把纹理图象应用到图元的过程。纹理映射通常用于把真实感加到场景中。如把一个建筑表面的图象应用到表示墙的多边形上。  
  
texture matrix  
纹理矩阵。把纹理坐标从指定的坐标变换到用于内插的坐标的4\*4矩阵。  
  
Third parties  
第三方  
  
threshold  
阈值  
  
track  
音轨  
  
Transform  
多边形转换。其根据你的视角，改变组成3D物体的顶点的坐标。  
  
transformation  
变换。一种有限空间的变形。在OpenGL中，变换限制在透视变换中进行，它包括可以用4\*4矩阵表示的任何变换，这样的变换包括旋转、平移、沿坐标轴的（非均匀）比例变化、透视变换、和这些变换的复合变换。  
  
Transmission  
透射  
  
triangle  
三角形。有三条边的多边形。三角形总是非凸的。  
  
triangulation  
三角剖分  
  
two dimensional manifold  
二维流形  
  
Union  
并运算  
  
unorganized points  
散乱数据点  
  
upstream  
上级  
  
variation interpolation  
变分插值  
  
Vertex  
它是3D场景中最基本的单位，携带着大量的描述信息。  
  
Vertex Instructions Per Vertex per Pass  
每次每个顶点的定点指令  
  
Vertex Shader  
顶点着色引擎  
  
vertex use  
点引用  
  
View independent  
视点独立  
  
view volume  
视窗体。在裁剪坐标系中的坐标满足条件：  
  
viewpoint  
视点。视点坐标系和裁剪坐标系的原点，这由OpenGL的上下文而定。如当讨论光照时视点就是视点坐标系的原点，当讨论投影时视点就是裁剪坐标系的原点。对于一个典型的投影矩阵，视点坐标系的原点和裁剪坐标系的原点在同一个位置。  
  
virtual reality  
虚 拟现实（VR）。虚拟现实是一种计算机与人的接口，人可以通过各种感觉设备进入一个三维图形的多媒体的世界，使人有一种身临其境的感受。虚拟现实的关键技 术是三维建模和多媒体、人机交互设备。自从八十年代末以来，虚拟现实已经得到广泛的研究和应用，尤其是在模拟训练、娱乐游戏等方面进展更快。虚拟现实也称 为虚真实、灵境等。  
  
VISC  
科学计算可视化（Visualization in Scientific Computing）  
  
Visualization in Scientific Computing  
科学计算可视化（VISC）  
  
Voice Interaction  
互动声音。这是对于游戏惯用的文字表达沟通方式进行突破的技术 — 游戏将可以在进行过程中整合入玩家的说话声音，而新的说话合成技术则允许电脑同样用语音进行对战中的沟通。  
  
vtable  
虚拟表  
  
Weighed Vertex  
重 力顶点。在过去，每个3D模型都是由许多独立的顶点组成的，而每个顶点位置决定不同的角度连接。一个模型由数百个多边形组成，同时也就有了数百个顶点，一 般来说普通的图像渲染引擎对于它们的处理来说都是同等的。可是在现在，一些游戏中每个角色模型的每一个顶点都将尝试被分配重力。这一做法通常用于对于真实 度要求较高的电影之中，这是因为这种技术可以使得由计算机图形接口处理的图像因为顶点具有重力而使得创造出来的模型人物具有感觉极为真实的皮肤和衣服。  
  
window  
窗口。帧缓冲区的一个子区域，通常是矩形的，其象素具有同样的帧缓冲区的配置。一个OpenGL的上下文一次只绘制到一个窗口中。  
  
windows coordinates  
窗 口坐标。指绘图窗口的坐标系。窗口坐标系中的坐标与象素的坐标的意义是不同的，如在窗口左下角的象素是象素（0，0），而其中心的窗口坐标是（0.5， 0.5，z），窗口坐标系中的坐标是连续的，而象素的位置并不是窗口坐标。窗口坐标系中的坐标除x，y值外，还有一个z值，这个z值代表深度成分。  
  
wireframe  
网格图。用网格线段来进行物体造型的一种三维模型的表示方法。一般网格的线表明了多边形的边。  
  
WNDCLASS  
窗口的类的结构  
  
Writer  
生成器  
  
z buffer depth sorting  
Z缓冲区深度排序。Z缓冲区深度排序是一种隐藏面删除的方法，它用于处理含有多个三维模型的复杂场景。每一个表面的Z值代表了它到视点的距离（深度），把所有场景中的Z值进行排序，从而保证离视点最近的面才被画出。这种方法与人对客观世界的物体的观察感觉方式十分类似。