

**XXXX实验报告**

**小组成员：**

陈志炯 11331040

李嘉良 11331166

黄永智 11331141

**项目简介：**

**开发环境：**

**实现效果：**

**项目内容（下面列出来这些是得分点）：**

**1、Illumination model (光照模型)：**

即模拟物体表面的光照物理现象的数学模型，又分为简单光照模型和全局光照模型两种。

简单光照模型亦称局部光照模型，其假定物体是不透明的，只考虑光源的直接照射，而将光在物体之间的传播效果笼统地模拟为环境光。

在简单光照模型中，Phong光照模型是真实感图形学中提出的第一个有影响的光照模型。它模拟物体表面对光的反射作用，反射作用分为物体间作用用环境光(Ambient Light)、漫反射(Diffuse Reflection)、镜面反射(SpecularReflection)，光源为点光源。

Phong模型存在的不足有：显示出的物体象塑料无质感变化、没有考虑物体间相互反射光、镜面反射颜色与材质无关、镜面反射大入射角失真现象。

简单光照模型不考虑周围环境对当前景物表面的光照影响，忽略了光在环境景物之间的传递，很难表现自然界复杂场景的高质量真实感图形。为了增加图形的真实感，必须考虑环境的漫射、镜面反射和规则投射对景物表面产生的整体照明效果。可以处理物体之间光照的相互作用的模型称为整体光照模型。

**2、Texture (材质)**

材质简单的说就是物体看起来是什么质地。材质可以看成是材料和质感的结合。在渲染程式中，它是表面各可视属性的结合，这些可视属性是指表面的色彩、纹理、光滑度、透明度、反射率、折射率、发光度等。

它用材料对光的红、绿、蓝三原色的反射率来近似定义材料的颜色。像光源一样，材料颜色也分成环境、漫反射和镜面反射成分，它们决定了材料对环境光、漫反射光和镜面反射光的反射程度。在进行光照计算时，材料对环境光的反射率与每个进入光源的环境光结合，对漫反射光的反射率与每个进入光源的漫反射光结合，对镜面光的反射率与每个进入光源的镜面反射光结合。对环境光与漫反射光的反射程度决定了材料的颜色，并且它们很相似。对镜面反射光的反射率通常是白色或灰色（即对镜面反射光中红、绿、蓝的反射率相同）。镜面反射高光最亮的地方将变成具有光源镜面光强度的颜色。

**3、Display list (显示列表)**

OpenGL显示列表是由一组预先存储起来的留待以后调用的OpenGL函数语句组成的，当调用这张显示列表时就依次执行表中所列出的函数语句。前面内容所举出的例子都是瞬时给出函数命令，则OpenGL瞬时执行相应的命令，这种绘图方式叫做立即或瞬时方式（immediate mode）。

OpenGL显示列表的设计能优化程序运行性能，尤其是网络性能。它被设计成命令高速缓存，而不是动态数据库缓存。也就是说，一旦建立了显示列表，就不能修改它。因为若显示列表可以被修改，则显示列表的搜索、内存管理的执行等开销会降低性能。

采用显示列表方式绘图一般要比瞬时方式快，尤其是显示列表方式可以大量地提高网络性能，即当通过网络发出绘图命令时，由于显示列表驻留在服务器中，因而使网络的负担减轻到最小。另外，在单用户的机器上，显示列表同样可以提高效率。因为一旦显示列表被处理成适合于图形硬件的格式，则不同的OpenGL实现对命令的优化程度也不同。

**4、Image manipulation (图像操作)**

（这什么鬼东西去死吧，翻译就是图像处理= =简直无情）

**5、Texture mapping (纹理映射)**

纹理映射是将纹理空间中的纹理像素映射到屏幕空间中的像素的过程。

在三维图形中，纹理映射的方法运用得最广，尤其描述具有真实感的物体。比如绘制一面砖墙，就可以使用一幅具有真实感的图像或者照片作为纹理贴到一个矩形上，这样，一面逼真的砖墙就画好了。如果不用纹理映射的方法，这墙上的每一块砖都要作为一个独立的多边形来绘制。另外，纹理映射能够保证在变换多边形时，多边形上的纹理也会随之变化。例如，用透视投影模式观察墙面时，离视点远的墙壁的砖块的尺寸就会缩小，而离视点近的就会大些，这些是符合视觉规律的。此外，纹理映射也被用在其他一些领域。如飞行仿真中常把一大片植被的图像映射到一些大多边形上用以表示地面，或者用大理石、木材等自然物质的图像作为纹理映射到多边形上表示相应的物体。

纹理映射是真实感图像制作的一个重要部分，运用它可以方便的制作出极具真实感的图形而不必花过多时间来考虑物体的表面细节。然而纹理加载的过程可能会影响程序运行速度，当纹理图像非常大时，这种情况尤为明显。如何妥善的管理纹理，减少不必要的开销，是系统优化时必须考虑的一个问题。还好，OpenGL提供了纹理对象管理技术来解决上述问题。与显示列表一样，纹理对象通过一个单独的数字来标识。这允许OpenGL硬件能够在内存中保存多个纹理，而不是每次使用的时候再加载它们，从而减少了运算量，提高了速度。

**5、Display text (显示文字，中文/英文，立体/平面)**

（嗯这个没什么好解释的）

**6、Anti aliasing (反走样)**

在光栅图形显示器上绘制非水平且非垂直的直线或多边形边界时，或多或少会呈现锯齿状或台阶状外观。这是因为直线、多边形、色彩边界等是连续的，而光栅则是由离散的点组成，在光栅显示设备上表现直线、多边形等，必须在离散位置采样。由于采样不充分重建后造成的信息失真，就叫走样(aliasing)。而用于减少或消除这种效果的技术，就称为反走样(antialiasing)。

计算机生成图像时通常存在三种走样现象中的两种：锯齿形边以及图形细节或纹理绘制失真。第三种现象出现在显示非常微小对象的场合。

基本上反走样方法可分为两类。第一类是提高分辨率 即增加采样点(提高采样频率)。然而，CRT光栅扫描设备显示非常精细光栅的能力是有限的，因此人们通常是在较高分辨率上对光栅进行计算，然后采用某种平均算法(滤除高频分量)得到较低分辨率的象素的属性，并显示在分辨率较低的显示器上 。这种方法称为超采样或后置滤波。另一类反走样是把像素作为一个有限区域，对区域采样来调整像素的亮度，以光顺边界来减小锯齿现象。这种方法等价于图像的前置滤波。

**7、Particle system (粒子系统，渲染雨、雪、雾等)**

粒子系统表示三维计算机图形学中模拟一些特定的模糊现象的技术，而这些现象用其它传统的渲染技术难以实现真实感。经常使用粒子系统模拟的现象有火、爆炸、烟、水流、火花、落叶、云、雾、雪、尘、流星尾迹或者像发光轨迹这样的抽象视觉效果等等。

通常粒子系统在三维空间中的位置与运动是由发射器控制的。发射器主要由一组粒子行为参数以及在三维空间中的位置所表示。粒子行为参数可以包括粒子生成速度（即单位时间粒子生成的数目）、粒子初始速度向量（例如什么时候向什么方向运动）、粒子寿命（经过多长时间粒子湮灭）、粒子颜色、在粒子生命周期中的变化以及其它参数等等。使用大概值而不是绝对值的模糊参数占据全部或者绝大部分是很正常的，一些参数定义了中心值以及允许的变化。

典型的粒子系统更新循环可以划分为两个不同的阶段：参数更新/模拟阶段以及渲染阶段。每个循环执行每一帧动画。

在模拟阶段，根据生成速度以及更新间隔计算新粒子的数目，每个粒子根据发射器的位置及给定的生成区域在特定的三维空间位置生成，并且根据发射器的参数初始化每个粒子的速度、颜色、生命周期等等参数。然后检查每个粒子是否已经超出了生命周期，一旦超出就将这些粒子剔出模拟过程，否则就根据物理模拟更改粒子的位置与特性，这些物理模拟可能象将速度加到当前位置或者调整速度抵消摩擦这样简单，也可能象将外力考虑进取计算正确的物理抛射轨迹那样复杂。另外，经常需要检查与特殊三维物体的碰撞以使粒子从障碍物弹回。由于粒子之间的碰撞计算量很大并且对于大多数模拟来说没有必要，所以很少使用粒子之间的碰撞。

每个粒子系统都有用于其中每个粒子的特定规则，通常这些规则涉及到粒子生命周期的插值过程。

在更新完成之后，通常每个例子用经过纹理映射的四边形sprite进行渲染，也就是说四边形总是面向观察者。但是，这个过程不是必须的，在一些低分辨率或者处理能力有限的场合粒子可能仅仅渲染成一个像素，在离线渲染中甚至渲染成一个元球，从粒子元球计算出的等值面可以得到相当好的液体表面。另外，也可以用三维网格渲染粒子。

**8、Bessel curve and surface (贝塞尔曲线和曲面)**

贝塞尔曲线，又称贝兹曲线或贝济埃曲线，是应用于二维图形应用程序的数学曲线。一般的矢量图形软件通过它来精确画出曲线，贝兹曲线由线段与节点组成，节点是可拖动的支点，线段像可伸缩的皮筋，我们在绘图工具上看到的钢笔工具就是来做这种矢量曲线的。贝塞尔曲线是计算机图形学中相当重要的参数曲线。

由于用计算机画图大部分时间是操作鼠标来掌握线条的路径，与手绘的感觉和效果有很大的差别。即使是一位精明的画师能轻松绘出各种图形，拿到鼠标想随心所欲的画图也不是一件容易的事。这一点是计算机万万不能代替手工的工作，所以到目前为止人们只能颇感无奈。使用贝塞尔工具画图很大程度上弥补了这一缺憾。

贝塞尔曲线是计算机图形图像造型的基本工具，是图形造型运用得最多的基本线条之一。它通过控制曲线上的四个点（起始点、终止点以及两个相互分离的中间点）来创造、编辑图形。其中起重要作用的是位于曲线中央的控制线。这条线是虚拟的，中间与贝塞尔曲线交叉，两端是控制端点。移动两端的端点时贝塞尔曲线改变曲线的曲率（弯曲的程度）；移动中间点（也就是移动虚拟的控制线）时，贝塞尔曲线在起始点和终止点锁定的情况下做均匀移动。

更高维度的广泛化贝塞尔曲线就称作贝塞尔曲面，其中贝塞尔三角是一种特殊的实例。

**9、Multiple texture (多重纹理)**

多重纹理就是在渲染一个多边形的时候可以用到多张纹理图。把多张纹理图进行一些颜色的操作，可以达到一些效果。但是多重纹理必须是在显卡支持的情况下。但是还有一种多次渲染。可以替代多重纹理，但是这样做的话，速度会很慢。

**10、Bump mapping (凹凸映射)**

真实世界中的物体表面都是不光滑的，所以需要通过凹凸模拟技术来体现真实物体所具有的凹凸起伏和褶皱效果。一般的纹理映射虽然可以添加精致的表面细节，但它无法影响物体表面的光照细节，因此不适合模拟粗糙表面。生成物体凹凸表面的一个简单的方法是对其表面法向量添加扰动，影响光照效果，这种技术又称凹凸映射。同一般的纹理映射一样，凹凸映射也需要一张纹理图，但这张图通常是灰度图，存放的也仅仅是像素的高度信息。实际产生效果的时候是通过计算凹凸图中相邻像素的高度差值来改变表面法向量的值。

**11、Shadow (阴影)**

阴影是由于光源产生的光线被不透明物体遮挡，而产生的黑暗范围。该范围构成二维轮廓，并且该轮廓的大小，形状，位置等会随光源相关属性的改变而改变。

在绘制真实感三维场景时，阴影为场景中各对象之间的空间位置关系提供了许多视觉线索。从物理的角度看，有光源才会产生阴影。如果任何光源都不能照射到空间中的某一点，那么位于那个点的观察者就不能看到任何光源，因此那个点就位于阴影之中。

如果光源是理想点光源，那么生成的阴影为硬阴影(hard shadows)；如果光源是面光源或体光源，那么完全被遮挡的区域称为本影区(umbra)，部分被遮挡的区域称为半影区(penumbra)，所生成的阴影称为软阴影(soft shadows)。

**12、Skeletal animation (骨骼动画)**

当前有两种模型动画的方式：顶点动画和骨骼动画。顶点动画中，每帧动画其实就是模型特定姿态的一个“快照”。通过在帧之间插值的方法，引擎可以得到平滑的动画效果。在骨骼动画中，模型具有互相连接的“骨骼”组成的骨架结构，通过改变骨骼的朝向和位置来为模型生成动画。

骨骼动画比顶点动画要求更高的处理器性能，但同时它也具有更多的优点，骨骼动画可以更容易、更快捷地创建。不同的骨骼动画可以被结合到一起——比如，模型可以转动头部、射击并且同时也在走路。一些引擎可以实时操纵单个骨骼，这样就可以和环境更加准确地进行交互——模型可以俯身并向某个方向观察或射击，或者从地上的某个地方捡起一个东西。多数引擎支持顶点动画，但不是所有的引擎都支持骨骼动画。

一些引擎包含面部动画系统，这种系统使用通过音位（phoneme）和情绪修改面部骨骼集合来表达面部表情和嘴部动作。

**13、Camera roaming (摄像机漫游)**

（完全找不到……）

**14、Sky rendering (天空绘制)**

在本项目中，天空绘制使用了天空盒的方式。

天空盒就是一个包含场景所有元素的长方体盒子，该盒子的六个面附有环境贴图，所有游戏行为均是在盒子中进行，在盒子内看来，附有贴图的盒子既是整体天空环境。

由天空盒的基本原理可以将其实现归纳为以下步骤：

1. 初始化天空盒长方体并将其放置到场景中。

2. 设置纹理分别渲染长方体各个面（如要绘制动态云则需每帧改变纹理UV）。

具体相关代码如下:

1. 初始化天空盒长方体定点信息

a. 定义定点格式

struct SkyboxVertices

{

enum

{

FVF = D3DFVF\_XYZ |D3DFVF\_DIFFUSE| D3DFVF\_TEX1,

};

VECTOR3 vPos;

DWORD nColor;

float fu;

float fv;

};

b. 初始化顶点数据(由于代码相似，只用一个面代码做例子)

m\_Verts[TOP][0].vPos = D3DXVECTOR3( - m\_fSize/2 , + m\_fSize/2 , -m\_fSize/2);m\_Verts[TOP][0].fu = 0.0f;m\_Verts[TOP][0].fv = 0.0f;m\_Verts[TOP][0].nColor = 0xffA283b0;

m\_Verts[TOP][1].vPos = D3DXVECTOR3( + m\_fSize/2 , + m\_fSize/2 , -m\_fSize/2);m\_Verts[TOP][1].fu = 1.0f;m\_Verts[TOP][1].fv = 0.0f;m\_Verts[TOP][1].nColor = 0xffA283b0;

m\_Verts[TOP][2].vPos = D3DXVECTOR3( - m\_fSize/2 , + m\_fSize/2 , +m\_fSize/2);m\_Verts[TOP][2].fu = 0.0f;m\_Verts[TOP][2].fv = 1.0f;m\_Verts[TOP][2].nColor = 0xffA283b0;

m\_Verts[TOP][3].vPos = D3DXVECTOR3( + m\_fSize/2 , + m\_fSize/2 , +m\_fSize/2);m\_Verts[TOP][3].fu = 1.0f;m\_Verts[TOP][3].fv = 1.0f;m\_Verts[TOP][3].nColor = 0xffA283b0;

2. 设置纹理贴图绘制天空盒

pDevice->SetSamplerState( 0, D3DSAMP\_MINFILTER, D3DTEXF\_LINEAR );

pDevice->SetSamplerState( 0, D3DSAMP\_MAGFILTER, D3DTEXF\_LINEAR );

pDevice->SetTexture( 1 , NULL );

// 渲染

if ( m\_Texture[TOP] )

{

pDevice->SetTexture( 0 , m\_Texture[TOP] );

}

pDevice->DrawPrimitiveUP( D3DPT\_TRIANGLESTRIP , 2 , &m\_Verts[TOP] , sizeof(SkyboxVertices) );

**15、Terrain rendering (地形渲染)**

地形渲染是三维图形学中必然不可缺席的一环，在游戏之类的应用中，地形效果的好坏更是直接影响整个产品的最终评价。

一般我们有一张高程图(height map)和一张地形表面纹理(diffuse map)，我们准备一个大型顶点网格（在X-Z平面上），然后以（网格顶点坐标 / 网格尺寸）的方式产生纹理坐标去检索高程图，从而获得该顶点的高度，渲染出来，再直接覆盖式贴上表面纹理，就是一个地形了。这个地形的外形直接取决于高度图的分辨率、网格大小和表面贴图的质量，但是它确实表达了一个“任意地形”的概念，是直至今天地形渲染技术的雏形。

**心得体会：**