微软对OpenGL的支持只到1.1，1.1以后微软就不再支持了，为什么，因为微软更想发展自家的DirectX。所以如果想使用OpenGL1.1以上的功能或者函数，只能使用OpenGL扩展，这些扩展是一些OpenGL团体或个人开发出来的能Windows上使用的OpenGL1.1以后的一些功能及函数。扩展并没有一个统一的标准，你可以使用glex，glew，glee等

一.判断显卡所支持的扩展:

const GLubyte \*str = glGetString(GL\_EXTENSIONS) ;

cout << str << endl ;

当然，这样查询的只是gl扩展，还可以用gluGetString获得glu扩展。

二.使用glew，包含了OpenGL中许多核心及扩展函数

1．包含glew.h和lib

#include <GL/glew.h>

#include <GL/glut.h>

#progmma comment(lib, "glew.lib")

2.调用glewInit() ;来初始化一下方可使用扩展

三.使用GLUT，演示用的窗口系统和操作系统

Opengl是独立于任何窗口系统或操作系统，因此没有包含打开窗口或者从键盘鼠标读取事件的函数，因此使用Opengl实用工具库GLUT来简化打开窗口，检测输入等任务。

注意：从鼠标事件返回的鼠标位置位于窗口坐标系中，该坐标系的原点在窗口的左上角，而opengl函数是基于视口的左下角，若窗口和视口一致，则必须利用窗口的当前高度减去鼠标回调函数获得的y值

Void drawSquare(int x, int y)

{

Y = window\_y – y ;

…  
}

OpenGL对场景中的图像进行渲染时所执行的主要图形操作：

1.根据几何图元形状，建立物体的数学描述。

（OpenGL的基本图元包括点，直线，多边形和位图，除位图外，均由顶点表示。程序员只能组合这些基本图元来创建复杂的物体）

2. 在三维空间中排列物体，并选择视察复合场景的有利视角

3. 计算所有物体的颜色

（颜色可以由应用程序明确指定，也可以是根据特定的光照条件所确定，或者是通过把纹理贴到物体的表面而获得，或是上述这三种操作的混合产物）

（纹理贴图texture mapping）允许把一幅二维图像应用到一个三维物体之上）

4. 光栅化转换为屏幕上的像素

Opengl状态机：先设置状态或默认状态，再使用

//clear background as black...

//因为在计算机中，保存图片的内存通常被计算机所绘制的前一幅图像所填充，因此在绘制新场景之前一般需要把它清除为某种背景颜色。

glClearColor(0.0f, 0.0f, 0.0f, 0.0f) ;

glClear(GL\_COLOR\_BUFFER\_BIT) ; // clear color buffer

双缓冲：可能硬件或软件实现，即提供了两个完整的颜色缓冲区。当一个缓冲区被显示时，另一个缓冲区正在进行绘图。当一个帧绘制完成之后，两个缓冲区就进行交换。使用双缓冲，每一帧只有在绘制完成后才会被显示，观众永远不会看到不完整的帧。

glutInitDisplayMode(GLUT\_DOUBLE) ;

…

glutSwapBuffers() ;

1. 绘图

glBegin(GL\_TRIANGLES);

glVertex3fv(V0) ;

glVertex3fv(V1) ;

glVertex3fv(V2) ;

glEnd() ;

2.用顶点数组画图，减少函数调用次数

//激活数组，如顶点坐标，表面法线，颜色，颜色索引，纹理坐标等

glEnableClientState(GL\_VERTEX\_ARRAY) ;

//指定数组的数据

glVertexPointer(2, GL\_INT, 0, verticesData) ; //verticesData is the data buffer

//解引用和渲染

//在顶点数组被解引用之前，数组一直保存在客户端，它们的内容很容易进行修改，解引用即提取指针所指向的数据，接着被发送到服务器，然后被发送到图形处理管线进行渲染。

glDrawArrays(GL\_TRIANGLES, 0, sizeof verticesData / sizeof verticesData[0]) ;

3. 用缓冲区对象，允许应用程序显式地把哪些数据存储在显卡里。

//创建缓冲区对象

GLuint buffers[NUM\_BUFFERS] ;

glGenBuffers(NUM\_BUFFERS, buffers) ;

//绑定缓冲区对象，确定它用于存储顶点数据还是索引。

glBindBuffer(GL\_ARRAY\_BUFFER, buffers[0]) ;

//缓冲区对象请求数据存储空间并初始化

glBufferData(GL\_ARRAY\_BUFFER, sizeof(verticesData), verticesData, GL\_STATIC\_DRAW) ;

//更新缓冲区对象的数据值...

//glMapBuffer()返回一个指向缓冲区对象的指针，可以对缓冲区对象中写入新值或读取数据。

//处理结束后用glUnmapBuffer()完成对数据的更新

GLfloat\* bufferData = (GLfloat\*) glMapBuffer(GL\_ARRAY\_BUFFER, GL\_READ\_WRITE) ;

doSomething(bufferData) ;

glUnmapBuffer(GL\_ARRAY\_BUFFER) ;

//释放缓冲区对象的资源

glDeleteBuffers(NUM\_BUFFERS, buffers) ;

立即模式(immediate mode):

只要程序执行到定义图元的语句，该图元就被发送服务器进行显示，与它有关的住处不会在系统中保留。要想在清屏之后重新显示图元或都在一次交互之后在新的位置绘制图元，程序必须重新确定图元，然后还要再次让这些信息经过显示处理过程。因此对于交互性很强的应用程序中的复杂对象，这个过程可能会使大量的数据从客户流向服务器。

保留模式（retained mode):

对象只需定义一次，然后把它的描述放到显示列表里。显示列表存放在服务器中，从客户向服务器发送一个简单的函数调用就可以重新显示它。这种方法的明显优点是减少了网络中的数据流量。这是因为显示处理器有一个有限的指令集，其中的大多数指令是为了在显示器上绘制图元。主机对用户程序进行编译后生成一个指令列表，然后把这个指令列表发送到显示处理器，显示处理器再把指令列表作为显示文件(display file)或者显示列表(display list)存储在显示存储器中。

void myInit()

{

//保存当前的属性和矩阵值

glPushAttrib(GL\_ALL\_ATTRIB\_BITS) ;

glPushMatrix() ;

#define BOX 1

//将显示列表发送到服务器，但不执行

glNewList(BOX, GL\_COMPILE) ;

glBegin(GL\_POLYGON) ;

glColor3f(1.0f, 0.0f, 0.0f) ;

glVertex2f(0.2f, 0.2f) ;

glVertex2f(0.8f, 0.2f) ;

glVertex2f(0.8f, 0.8f ) ;

glVertex2f(0.2f, 0.8f) ;

glEnd() ;

glEndList() ;

glPopAttrib() ;

glPopMatrix() ;

}

void display(void)

{

//执行指定的显示列表

glCallList(BOX) ;

}

类似照相机拍照原理：

1. 把照相机固定在三角架上，并让它对准场景（视图变换，修改观察点的位置和方向，OpenGL中一般采用默认的视觉坐标系）
2. 对场景进行安排，使各个物体在照片中的位置是我们所希望的（模型变换,对物体进行旋转，移动和缩放）
3. 选择照相机镜头，并调整放大倍数（投影变换）
4. 确定最终照片的大小，比如放大或缩小，位伸（视口变换）

注意：代码顺序：视图变换必须在模型变换之前，在绘图之前的任何时候执行投影变换和视口变换。管线顺序：OpenGL对物体顶点执行的顺序，顶点->模型视图矩阵(视觉坐标)->投影矩阵(裁剪坐标)->透视除法(规范化设备坐标)->视口变换(窗口坐标)

注意：我们可以不必通过移动照相机（使用视图变换）来观察这个立方体，而是可以移动这个立方体（使用模型变换），视图变换和模型变换的这种双重性质就是我们需要同时考虑这两种类型的变换的原理。把这两种变换割裂开来是没有意义的，因此把模型和视图变换组合成模型视图矩阵

注意：在计算机图形学中强调的是对象定义与照相机定义之间的独立性。在OpenGL中，初始的世界标架和照相机标架相同（初始的模型－视图矩阵是单位阵）；照相机位于原点，并指向z轴的负向；OpenGL也指定了默认的视景体，它是一个中心在原点的边长为2的立方体（缺省的投影矩阵是单位阵）；默认的投影是正交投影。

Void reshape(int w, int h)

{

//视口变换，在默认情况下，视口被设置为占据打开窗口的整个像素矩形即(0,0,w,h),使用视口变换，可以选择一个更小的绘图区域。例如我们可以对窗口进行划分，在同一个窗口中显示分割屏幕的效果，以显示多个视图。

glViewport(0, 0, (GLsizei) w/2, (GLsizei)h/2) ;

//可以选择视景体，默认视景体是glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0)

glMatrixMode(GL\_PROJECTION) ;

glLoadIdentity() ;

//透视投影，创建一个表示透视视图平截头体的视景体。

glFrushtum(-1.0, 1.0, -1.0, 1.0, 1.5, 20.0) ;

//透视投影，创建一个表示对称透视视图平截头体的视景体

//gluPerspective(60.0, 1.0, 1.5, 20.0) ;

//正投影,

glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0) ;

//gluOrtho(-1.0, 1.0, -1.0, 1.0) ;

//设为默认模型视图变换

glMatrixMode(GL\_MODELVIEW) ;

}

Void display(void)

{

//视图变换，记住视图变换函数必须在调用任何模型变换函数之前，以确保首先作用于物体的是模型变换.默认情况下照相机位于原点，指向Z轴的负方向，朝上向量为(0,1,0).

glLoadIdentity() ;

gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0) ;

//模型变换，顶点v的变换是(NM)v

glTranslatef(-20., 0.0, 0.0) ; //N

glRotatef(45.0, 1.0, 1.0, 1.0) ; //M,以逆时针方向绕着从原点到点(1,1,1)的直线旋转45

draw\_triangle() ;

}

第四章：几何对象和变换（用模视变换矩阵把对象的顶点从对象坐标系变换到眼坐标系）

模视变换：模型坐标 -> 世界坐标 -> 眼坐标

初始的模视矩阵是单位矩阵，所以此时的对象标架和眼标架是重合的，并且照相机正对的方向是z轴的负方向。因此如果不改变模视变换矩阵，那么我们使用的标架是眼标架。大多数情况下把照相机看做是固定的，其化标架则相对于照相机移动。因此模视变换矩阵

A= [1 0 0 0

0 1 0 0

0 0 1 -d

0 0 0 1]

把对象标架下的点(x,y,z)变换成照相机下的点（x,y,z,-d)

一般的三维旋转，必须指定如下三个实体：不动点P，旋转角度theta和旋转轴。以原点为不动点的旋转可以分解为三个绕坐标轴的旋转。虽然最后得到的旋转矩阵是相同的，但绕三个坐标轴旋转的顺序不是唯一的。

顶点 -> CTM -> 顶点

CTM (current transformation matrix) = 模视变换矩阵 + 投影变换矩阵

//模视变换矩阵通常是一个仿射变换矩阵，只有12个自由度。投影矩阵不是仿射矩阵，有16个自由度

//用OpenGL实现具有任意不动点的旋转

//注意OpenGL修改CTM是右乘，所以以反方向顺序，从而为左乘

//实现45旋转，其旋转轴的方向向量从原点指向点(1,2,3),并且(4,5,6)是不动点。

glMatrixMode(GL\_MODELVIEW) ;

glPushMatrix() ;

glLoadIdentity() ;

glTranslatef(4.0, 5.0, 6.0) ;

glRotated(45.0, 1.0, 2.0, 3.0) ;

glTranslatef(-4.0, -5.0, -6.0) ;

...

glPopMatrix() ;

最终CTM = T(4,5,6)R(45,1,2,3)T(-4,-5,-6)，然后处理后的顶点q = Cp

2.变换矩阵的加载

glLoadMatrixf(myarray) ; //注意myarray是把矩阵的各列连起来组成的一维数组。

如:

GLfloat m[4][4] ;

GLfloat myarray[16] ;

for (int i = 0 ; i <3 ; ++ i)

for (int j = 0 ; j <3 ; ++ j)

myarray[4\*j + i] = m[i][j] ;

3.对当前矩阵右乘一个用户定义的矩阵

glMultMatrixf(myarray) ;

第五章观察

观察过程包括两个步聚：

1. 使用模视变换矩阵把对象的顶点表示从对象标架变换到眼标架。
2. 指定想要的投影类型（平行投影或透视投影）和视见体（或称为裁剪体，是场景中我们希望成像的那一部分），投影类型和视见体确定一个投影变换矩阵。

经典平行投影（正投影，轴投影和斜投影）和经典透视投影（一点，两点，三点透视），从应用程序编程人员的角度来看，不同的平行投影之间没有显著的差异，应用程序编程人员只需指定一种投影类型（平行投影或透视投影）并对描述照相机的一组参数进行设置，三种透视投影只不过是一般透视投影的特殊情况。因为计算机图形学中的观察建立在虚拟照相机模型的基础上，所以应该能够生成任何一种经典投影图。但经典投影图都依赖于对象，观察者和投影线之间的特定位置关系，而在计算机图形学中，我们强调对象定义和照相机参数设置这两者之间的独立性。Opengl不能判断一个透视投影是一点，两点还是三点透视，因为这需要知道对象和照相机之间的位置关系。

从流水线结构的角度来看，观察包括两个基本步聚。第一步，我们必须把照相机放到指定的位置并使其具有指定的方向。这一步同模视变换来完成。顶点经过模视变换之后，就得到在照相机标架下的表示。第二步，应用投影变换，这一步要对顶点应用指定的投影（正投影或透视投影），并把指定裁剪体变换成规范化的裁剪体。

Opengl的默认照相机位于对象标架的原点，方向指向Z轴的负方向。该照相机生成的是正交投影图。默认的视见体是中心位于原点并且边长为2的立方体。默认的投影平面为z=0，投影方向沿着Z轴。因为默认的模视变换矩阵和投影变换矩阵是单位矩阵，所以CTM是单位矩阵。

如何在OpenGL中定位照相机？

第一种方法通过应用一系列旋转和平移来改变模视变换矩阵，从而间接指定照相机的方位。如：

//将照相机位于原点变换点(1, 1, 1) ，从成生成等轴测投影图。

glMatrixMode(GL\_MODELVIEW) ;

glLoadIdentity() ;

glTranslatef(0, 0, -d) ;

glRotatef(35.26, 1, 0, 0) ;

glRotatef(45, 0, 1, 0) ;

第二种方法指定观察坐标系+观察参考点,照相机位于点e为视点，是在建模标架下指定，照相机的方向指向另一个点a，该点称为参考点，再对照相机指定想要的观察正向。

如：

glMatrixMode(GL\_MODELVIEW) ;

glLoadIdentity() ;

gluLookAt(eyex, eyey, eyez, atx, aty, atz, upx, upy, upz) ;

注意，我们常把标准的旋转，平移和缩放作为定义对象的一部分。同时认为gluLookAt是在定位照相机，尽管这些变换都会改变模视变换矩阵。

定义投影类型和视见体？

注意：在OpenGL中，定位照相机的函数（如gluLookAt）是改变模视变换矩阵，并且其参数是用建模坐标给出的，而如下函数是改变投影变换函数，且函数参数是在观察坐标给出。

透视投影的三种方式：

1.

glMatrixMode(GL\_PROJECTION) ;

glLoadIdentity() ;

glFrustum（left, right, bottom, top, near, far） ;

注意：near and far分别是从COP（观察标架的原点）到前裁剪面和后裁剪面的距离，都平行于平面z=0。因为照相机的方向指向z轴的负方向，所以近裁剪面的方程z=-near,远裁剪面的方程为z=-far。

2.

//利用视域指定视见体

glMatrixMode(GL\_PROJECTION) ;

glLoadIdentity() ;

gluPerspective(fovy, aspect, near, far) ;

3.

通过加载，或者通过对初始单位阵应用旋转平移和缩放变换来直接生成投影变换矩阵。

平行投影：

//默认的投影变换矩阵相当于如下：

glMatrixMode(GL\_PROJECTION) ;

glLoadIdentity() ;

glOrtho(-1, 1, -1, 1, -1, 1) ;

注：照相机位于原点，方向指向z轴的负方向，因此近裁剪面位于z=1.0，在照相机的后方；远裁剪面位于z=-1，在照相机的前方。

投影规范化：先把对象变形，使得视见体变形为规范视见体，规范视见体是中心位于原点，边长为2的立方体。优点是相同的绘制流水线可以既支持透视投影又支持平行投影。只需加载合适的规范化矩阵。

扩展：

隐藏面消除：

如果一个立方体的各面不透明，那么只能看到它的三个面向前方的面，从基本的观察模型的角度来看，只有这些面被看到是因为它们挡住了投影线，使投影线不能到达任何其他的面。不过，从计算机图形学的角度来看，立方体的所有6个面都已经被定义，并且这个6个面都经过了绘制流水线的处理。因此，图形系统必须关心它所显示的是哪个面。因此需要隐藏面消除算法或可见表面算法，OpenGL采用Z缓存算法。

从应用程序编程人员角度，调用如下的函数来使用隐藏面消除功能：

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB | GLUT\_DEPTH) ;

glEnable(GL\_DEPTH\_TEST) ;

…

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT) ;

以上隐藏面消除功能要处理所有面，即6个面都要经过光删化。如果对象是凸对象，比如像立方体，法线方向背离观察者的面总是不可见的，可以采用背面剔除功能，从而只有3个面经过光删化。所以一般在使用z缓存算法时同时开户背面剔除功能

glEnable(GL\_CULL) ;

流水线中的仿射变换单元只需对端点进行变换，在光栅化阶段根据端点的处理结果就可以生成内部的点。

第五章 光照

光源 + 光照模型 + 材料

OpenGL在模拟光照时，假定光可分解为红绿蓝成分，因此光源的特征是由它所发射的红绿蓝数量决定。表面材料的特征是由它向各个不同方向所反射的红绿蓝入射光的百分比决定。

OpenGL的光照模型把光分成4种成分：环境光+散射光+镜面光+发射光

环境光ambient light: 在环境中进行了充分的散射，无法分辨其方向的光。似乎是来自于所有的方向。房间里的逆光包含了非常多的环境光成分。户外的探照灯所包含的环境光成分非常少。场景中可能存在一些基本的环境光叫全局环境光

散射光diffuse light: 来自某个方向，与顶点到光源的方向和顶点的法向方向的点积成正比，与眼睛在哪个位置无关，散射光看上去总是一样亮。

镜面光specular light:来自特定的方向。具有光泽的金属或塑料具有非常高的镜面成分。粉笔和地毯几乎不存在镜面成分。

OpenGL光照模型根据材料所反射的红绿蓝比例来模拟颜色。决定材料对红绿蓝的反射率。

材料还可能具有一种发射颜色emissive color，模拟那些源自某个物体的光。在整体场景中，发射颜色并没有被作为一种额外的光照。

物体的法线向量决定它相对于光源的方向，对于物体的每个顶点，OpenGL使用法线判断这个顶点从每个光源所接收的光线数量。

顶点的颜色 = 顶点处的材料发射颜色

+ 全局环境光（在顶点处根据材料环境颜色属性进行缩放）

+ 经过适当衰减的来自所有光源的环境、散射和镜面光成分

= emissionmaterial

+ ambientlignt model \* ambientmaterial

+ \* (聚光灯效果)I \*

\* [ambientlight \* ambientmaterial + max(L.N, 0) \*diffuselight \* diffusematerial

+ max(S.N, 0)shininess \* specularlight \* specularmaterial]i

如果 光照模型颜色控制为GL\_SEPARATE\_SPECULAR\_COLOR, 那么每个顶点都将产生主颜色和辅助颜色

主颜色 = 顶点的材料发射颜色

+ 根据顶点处的材料环境颜色属性进行缩放后的全局环境光

+ 来自所有光源并进行了适当衰减的环境光和散射光成分

= emissionmaterial

+ ambientlignt model \* ambientmaterial

+ \* (聚光灯效果)I \*

\* [ambientlight \* ambientmaterial + max(L.N, 0) \*diffuselight \* diffusematerial]i

辅助颜色 = 来自所有光源并进行了适当衰减的镜面光成分

= \* (聚光灯效果)I \*

\* [max(S.N, 0)shininess \* specularlight \* specularmaterial]i

在进行纹理贴图时，只有主颜色与纹理颜色进行混合，在纹理操作之后，辅助颜色被添加到主颜色和纹理颜色混合所产生的颜色之中。

注意：L是从顶点到光源(GL\_POSITION)的单位向量, N是顶点的单位法线向量。S是L和顶点到观察点的向量的和规范化。

光源：

//光源的环境强度，散射强度和镜面强度

GLfloat light\_ambient[] = {0, 0, 0, 1} ;

GLfloat light\_diffuse[] = {1, 1, 1, 1} ;

GLfloat light\_specular[] = {1, 1, 1, 1} ;

glLightfv(GL\_LIGHT0, GL\_AMBIENT, light\_ambient) ;

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, light\_diffuse) ;

glLightfv(GL\_LIGHT0, GL\_SPECULAR, light\_specular) ;

//光源位置。(可以把光源放在无限远处，模拟太阳光，或靠近场景如壁灯)

GLfloat light\_position[] = {1, 1, 1, 0} ;

glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position) ;

//聚光灯方向，聚光指数，聚光灯的切角。(产生狭窄的聚集光束或角度较广的光束)

GLfloat spot\_direction[] = {-1, -1, 0} ;

glLightfv(GL\_LIGHT0, GL\_SPOT\_CUTOFF, 45.0) ;

glLightfv(GL\_LIGHT0, GL\_SPOT\_DIRECITON, spot\_direction) ;

glLightfv(GL\_LIGHT0, GL\_SPOT\_EXPONENT, 2.0) ;

//常量，线性，二次衰减因子（无线限处如方向性光源，）

glLightfv(GL\_LIGHT0, GL\_CONSTANT\_ATTENUATION, 1.5) ;

glLightfv(GL\_LIGHT0, GL\_LINEAR\_ATTENUATION, 0.5) ;

glLightfv(GL\_LIGHT0, GL\_QUADRATIC\_ATTENUATION, 0.2) ;

光照模型：

//全局环境光强度

GLfloat lmodel\_ambient[] = {0.2, 0.2, 0.2, 1.0} ;

glLightModelfv(GL\_LIGHT\_MODEL\_AMBIENT, lmodel\_ambient) ;

//观察点的位置是位于场景还是位于无限远处

(一个特定顶点上的亮点的强度计算取决于这个顶点的法线，这个顶点和光源的方向以及这个顶点和观察点的距离)

glLightModeli(GL\_LIGHT\_MODEL\_LOCAL\_VIEWER, GL\_TRUE) ;

//物体的正面和背面是否应该执行不同的光照计算(所有的多边形都要执行光照计算，不管它们是正面还是背面，但是，由于我们在思维中所设置的光照条件一般适用于正面的多边形，因此背面的多边形所接受的光照一般并不正确)

glLightModeli(GL\_LIGHT\_MODEL\_TWO\_SIDE, GL\_FLASE) ;

//镜面颜色是否应该从环境和散射颜色中分离出来，并在纹理操作之后再应用

在这种模式下，光照为每个顶点产生两种颜色：一种是主颜色，它由所有非镜面光照颜色组成，另一种是辅助颜色，它是所有镜面光照颜色的总和。在进行纹理贴图时，只有主颜色与纹理颜色进行组合，在纹理贴图后，辅助颜色被添加到主颜色和纹理颜色的最终组合颜色。

glLightModeli(GL\_LIGHT\_MODEL\_COLOR\_CONTROL, GL\_SEPARATE\_SPECULAR\_COLOR);

材料属性：

从概念上说，绝大多数材料属性类似于我们在创建光源时所设置的属性。

//材料的环境颜色，散射颜色和镜面颜色

GLfloat mat\_ambient[] = {0.7, 0.7, 0.7, 1.0} ;

GLfloat mat\_diffuse[] = {0.1, 0.5, 0.8, 1.0} ;

GLfloat mat\_specular[] = {1.0, 1.0, 1.0, 1.0} ;

glMaterialfv(GL\_FRONT, GL\_AMBIENT, mat\_ambient) ;

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_diffuse) ;

glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_specular) ;

//镜面指数，表示光泽度

GLfloat no\_shininess[] = {0.0} ;

glMaterialfv(GL\_FRONT, GL\_SHININESS, no\_shininess) ;

//材料的发射颜色

GLfloat mat\_emission[] = {0.3, 0.2, 0.2, 0.0} ;

glMaterialfv(GL\_FRONT, GL\_EMISSION, mat\_emission) ;

//开启光照，光源

glEnable(GL\_LIGHTING) ;

glEnable(GL\_LIGHT0) ;

第六章 混合、抗锯齿、雾和多边形偏移

混合的目的使场景看上去像是半透明的，做法是指定一个混合函数，把源颜色和目标颜色进行混合。如果启用了混合，alpha值常常用来把被处理片断和颜色值与已经存储在帧缓冲区中的像素颜色进行组合。通过alpha混合来创建半透明的片断，仍然保留一些原先存储的颜色值，颜色混合是诸如透明化，数字合成，油漆这类技巧的核心。

//选择源混合因子S和目标混合因子D

glBlendFunc(GL\_ONE, GL\_ONE) ;

//启用混合功能

glEnable(Gl\_BLEND) ;

//使用混合方程式组合像素如Cs\*S + Cd\*D ;

glBlendEquation(GL\_FUNC\_ADD) ;

抗锯齿：OpenGL根据片断覆盖像素方块的面积来计算覆盖比例。在RGBA模式下，OpenGL将片断的覆盖比例与alpha值相乘，并将乘积作为alpha值把片断与帧缓冲区中的对应像素进行混合。

可以采用多重采样技巧，每个片断不再只有一种颜色，一个深度值和一组纹理坐标，而是根据子像素样本的数量，具有多种颜色，多个深度值和多组纹理坐标。多重采样特别适合对多边形的边缘进行抗锯齿处理。

如果多重采样被启用，并且存在一个多重采样缓冲区，那么点直线和多边形所产生的片断一般都进行了抗锯齿处理。

//获取一个支持多重采样功能的窗口

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB | GLUT\_MULTISAMPLE) ;

//启用多重采样功能

glEnable(GL\_MULTISAMPLE) ;

雾：计算机图像有时候由于过于清晰和锐利，反而显篮球不太逼真，抗锯齿处理使物体的边缘显得更为平滑，增加了逼真感。欲使整幅图像变得更加逼真，可以添加雾效果，使远处的物体的看上去逐渐变得模糊，主要用来描述大气效果，常用于模拟模糊、薄雾、烟或污染。雾在本质上是一种视觉模拟应用。用于模拟具有有限可视性的场合。

雾是在执行矩形变换，光照和纹理之后才被应用的，因此对经过变换的，带光照和经过纹理贴图的物体产生影响。

RGBA模式下经雾处理的颜色：C = f \* Ci + (1-f)\*Cf

其中Ci表示源片断的RGBA值，Cf雾颜色，f通过雾方程式计算的值

//选择雾方程式,同时设置雾方程式参数

glFogi(GL\_FOG\_MODE, GL\_LINEAR) ;

glFogf(GL\_FOG\_START, 1.0) ;

glFogf(GL\_FOG\_END, 5.0);

//雾的颜色Cf

GLfloat fogColor[4] = {0.5, 0.5, 0.5, 1.0} ;

glFogfv(GL\_FOG\_COLOR, fogColor) ;

多边形偏移：绘制实心物体，并使用那个物体相同的顶点绘制在实心物体上的一个线框轮廓，将会产生丑陋的人工痕迹。这是由于直线和多边形的光栅化方式并不完全相同，因此OpenGL为直线上的像素所产生的深度值和为多边形边缘上的像素所产生的深度值通常并不相同，即使是在同一个顶点，它们的值也可能不同，着重显示的直线可能受到与它重叠的多边形的影响而变得忽浓忽淡，从而产生了非常不好的视觉效果。

多边形偏移增加一个适当的偏移值，把重合的z值适当分开一些，使着重显示的直线与多边形的边缘清晰地分离开来。为了漂亮地渲染边缘着重显示的实心物体，避免产生不良的视觉效果，我们可以向实心物体添加一个正值（把它推向远处），或者向线框添加一个负值（把它拉到近处）。偏移量的大小取决于许多因素，包括每个多边形的深度斜率以及线框直线的宽度。

//glPolygonOffset(0.0, 1.0) ; 很小的常数偏移量，

glPolygonOffset(1.0, 1.0) ; //明显的深度区别

glEnable(GL\_POLYGON\_OFFSET\_FILL) ;

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE) ;

第七章 显示列表

显示列表(display list): 是一组存储在一起的OpenGL函数，可以在以后执行。如果需要多次重绘同一个几何图形，或者如果有一些需要多次调用的用于更改状态的函数，把这些函数存储在显示列表中是个很好的思路。使用显示列表，可以一次定义几何图形（状态更改），并在以后多次执行它们。

应用举例：

如绘制三轮车车轮的一种有效方法：把绘制一个轮子的操作存储在显示列表中，并3次执行这个显示列表，每次在执行显示列表之前，根据需要适当地设置模型视图矩阵，以便正确地计算每个轮子的大小和位置。

如当通过网络在另一台远程机器上运行OpenGL程序时，把绘图命令保存在显示列表中具有特别重要的意义。由于显示列表是服务器状态的一部分，保存在服务器中，如果我们把需要重复调用的函数存储在显示列表中，显示可以大大减少通过网络传输的数据量。

有些图形硬件可能会把显示列表存储在专用的内存中，甚至以一种更为优化的形式来存储数据，使之与图形硬件或软件更为兼容。

GLuint theTorus ;

Void init(void)

{

//为显示列表取一个整形索引值标识

theTorus = glGenLists(1) ;

glNewList(theTorus, GL\_COMPILE) ;

glCallList(handlebars) ;

glTranslatef(1.0, 0.0, 0.0) ;

glCallList(wheel) ;

glEndList() ;

}

Void display(void)

{

glCallList(theTorus) ;

}

注意：显示列表里的参数将会进行求值，最终的值是在显示列表被创建时复制到显示列表的。显示列表中的值不能在以后进行修改，当一个函数被存储到显示列表之后，就没有办法将它从这个显示列表中删除，定义了一个显示列表之后，我们就没有办法再在其中插入新函数，我们可以删除整个显示列表并创建一个新显示列表，但不能对显示列表进行编辑。为了优化性能，OpenGL的显示列表更像是命令缓存器，而不是动态数据库，换句话说，当显示列表被创建之后，它就无法进行修改。

显示列表最能够体现优化作用的领域：

1. 矩阵操作
2. 对位图和图像进行光栅化（在编译显示列表时，OpenGL可能会把数据转换为最适合硬件形式）
3. 光源材料属性和光照模型（在复杂的光照条件下绘制场景时，可能需要更改场景中每个物体所使用的材料。设置材料的速度可能非常慢，因为它可能涉及非常复杂的计算，如果把材料定义放在显示列表中，每次切换不同的材料时，就不必重新执行这些计算，因为显示列表所存储的只是最终的计算结果。

存储在显示列表里的是什么？

在创建显示列表时，只有表达式的值存储在显示列表中。如果数组中的值以后发生了更改，显示列表中的值并不会发生变化。

不可以存储在显示列表中执行的OpenGL函数：设置客户机状态的函数以及用于提取状态值的函数（因为它们都有返回值，或者直接返回，或者把值保存在一个传引用的参数中，

如依赖客户机获取信息glGet\*,glIs\*, 或依赖客户状态glFlush(),glFinish()

如修改客户状态 函数glPixelStore(), glSelectBuffer(),

如定义顶点数组函数glVertexPointer(), glColorPointer(), glInterleavedArrays()（因为修改客户状态指针）

如使用像素存储模式的函数glPixelStore()

因为当一个显示列表被创建时，它存储于服务器中，调用程序将不知道如何处理通过网络返回的数据，所以具有返回值的函数都不能存储在显示列表中。

执行多个显示列表：

//定义多个显示列表

Void initStrokedFont(void)

{

GLuint base = glGenLists(128) ;

glListBase(base); //显示列表基址

glNewList(base+’A’, GL\_COMPILE) ; … ; glEndList() ;

glNewList(base+’E’, GL\_COMPILE) ; … ; glEndList() ;

glNewList(base+’S’, GL\_COMPILE) ; … ; glEndList() ;

}

Void printStrokedString(GLbyte\* S)

{

glCallLists(strlen(s), GL\_BYTE, S) ; //显示列表索引={base+S[0], base+S[1],…} ;

}