实验内容

１．根据图形学坐标变换的原理，利用OpenGL函数对图形界面中所加载的几何对象进行交互的平移、旋转、缩放等操作。✅

２．利用OpenGL定义光源及几何对象材料。✅

３．加载多个几何对象，对比有无消隐处理的结果。✅

４．采用C/C++ 、OpenGL编写程序（参考本次及前几次实验所提供的程序代码及建立Project的过程说明。）。✅

５．选作：

i.对光照的颜色和位置以及几何对象的材料进行交互修改✅

ii.利用OpenGL的功能给几何对象粘贴纹理✅

iii.用MFC实现交互界面

实现方法

1. 整体架构

该实验基于OpenGL和GLUT库构建，用于展示和交互不同几何对象，允许用户对这些对象进行平移、旋转、缩放等操作。整体架构包括：

窗口初始化和事件处理：使用GLUT创建窗口，设置绘图上下文，并处理鼠标、键盘事件。

OpenGL渲染管线：实现几何对象的绘制、变换和材质、光照特性的控制。

菜单系统：提供交互式菜单，允许用户选择几何对象、改变光源颜色和位置，修改材质属性等。

1. 实现的功能简介

具体实现功能和操作方法如下：

1. **菜单选择**：

**“Change Color”**：改变光源的颜色；

**“Apply Random Materail”**：随机生成和应用材质属性到几何对象上。

**“Add Texture”**：应用纹理贴图。

**“Remove Texture”**：取消应用纹理贴图。

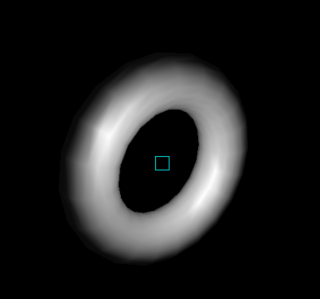
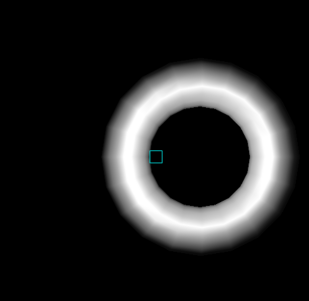
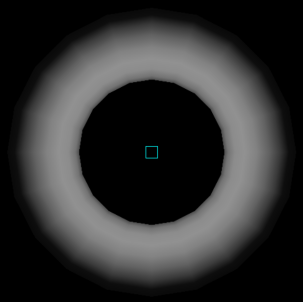
**“Torus”、“Tetrahedron”、“Dodecahedron”**、

**“cosahedron”、“Teapot”、“Quad”**：绘制不同几何对象。

**“DeptTest”**：绘制两个深度不同的物体进行深度测试。

**“Quit”**：退出应用。

1. **鼠标中键**按下拖动实现几何对象的旋转，**鼠标左键**按下拖动实现光照的旋转（光源的移动）。
2. **键盘**wasd实现几何对象的平移，‘+’‘-’实现几何对象缩放。
3. 几何对象交互操作

对窗口中绘制的对象实现平移、旋转、缩放的操作，具体实现方法如下：

* 1. 实现原理

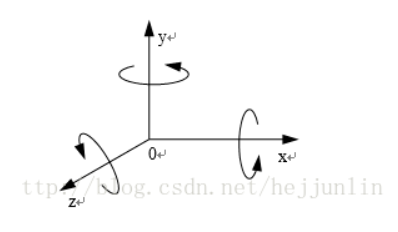
本次实现的Display函数的大致框架如下。



glPushMatrix和glPopMatrix函数的作用是通常在进行一系列变换之前和之后分别调用，以便在进行变换时能够随时返回到这个变换之前的状态，在这个区块中进行的变换不会影响到其他区块。

在display函数中，有两个这样的区块。我们先忽略橙色方框区域的小区块（绘制灯光的部分），单独看蓝色方框区域的大区块。

在进行矩阵变换前，OpenGL中的坐标系示意图如下。屏幕中心为原点(0, 0, 0)，且是始终不变的。面对屏幕，右边是x正轴，上面是y正轴，屏幕指向自己的为z正轴。旋转时正方向遵守右手法则（大拇指指向某轴的正方向，其他四指的方向为转动的正方向）。



OpenGL的运作方式类似于状态机，先定义要绘制物体的状态再进行绘制。此处我们操作的矩阵为即将要绘制物体的坐标矩阵，我们在这里称这个矩阵为M。

在蓝色区块中，首先调用了glTranslatef(0.0, 0.0, -5.0)来将M向屏幕远处（z轴负方向）移动了5个单位，以便能拉远距离看清绘制的物体。接下来如果不对物体另外做任何操作，可以直接调用绘制函数绘制物体。

由于我们要实现平移、旋转和缩放的操作，在glTranslatef(0.0, 0.0, -5.0)之后又添加了四条对M进行矩阵变换的语句：

glScalef(scaleValue, scaleValue, scaleValue);//缩放

glTranslatef(translateX, translateY, 0.0);//平移

glRotatef((GLdouble)rotationV, 1.0f, 0.0f, 0.0f);// 垂直方向旋转

glRotatef((GLdouble)rotationH, 0.0f, 1.0f, 0.0f);// 水平方向旋转

完成这一系列的变换后，矩阵M就算定义好了，便可以开始屏幕中物体的绘制。另外，此处需要注意的是需要先进行缩放和平移的操作，再进行旋转的操作（矩阵操作中，要想达到特定效果，关键顺序不可颠倒）。

如果先进行旋转操作，后面的平移操作就会出现逻辑混乱的情况。（例如：先进行了上下颠倒的180°旋转后，向上平移就会变成向下平移）

矩阵变换中的参数如下：

int rotationV = 0;

int rotationH = 0;

static float translateX;

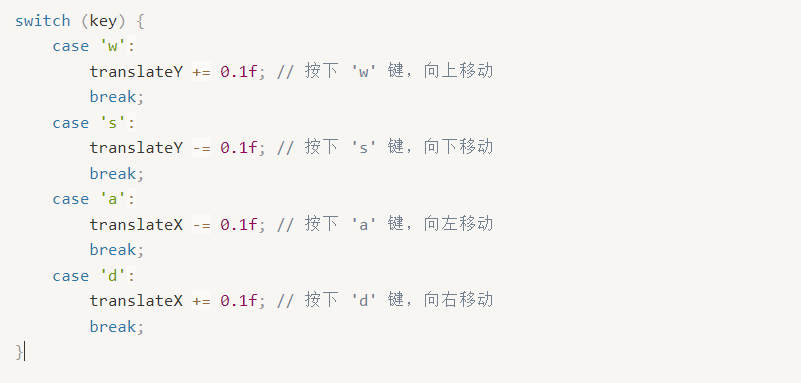
static float translateY;

static float scaleValue = 1.0f;

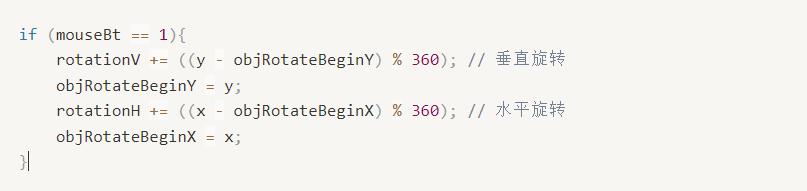
它们都为全局变量，在交互的时候可以进行用户自定义更改。

* 1. 交互实现

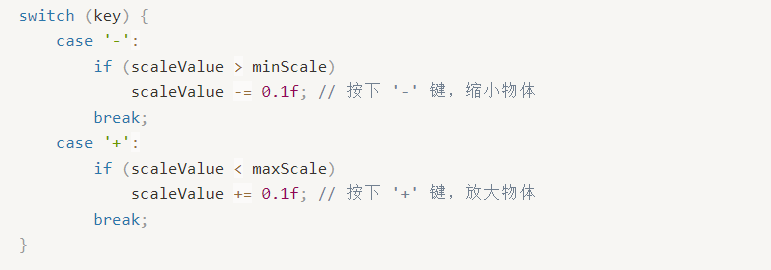
平移：使用键盘控制，按下键盘上的 w、s、a、d 可以使几何对象上下左右移动。在键盘事件处理函数 keyboard中，捕捉键盘输入控制translateX和translateY变量的值。



旋转：在鼠标事件处理函数 motion中实现。通过按下并拖动鼠标中键实现。objRotateBeginX和objRotateBeginY为鼠标上个状态的x和y坐标值，通过比对前后鼠标的坐标差实现旋转参数rotationV和rotationH的改变。



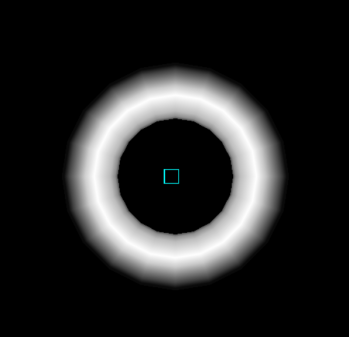
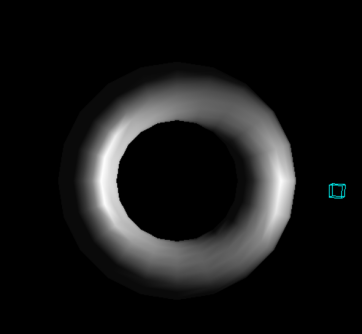
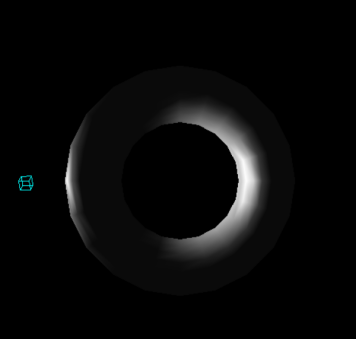
缩放：按下"+"和"-"键实现，按下"+"键可以放大几何对象，按下"-"键可以缩小几何对象。（因为"+"键有时候发生系统冲突，当"+"键不起作用的时候按下“shift”和 "+"键即可）控制scaleValue的值改变。此外设置最小和最大缩放值minScale = 0.5和fmaxScale = 2.0f控制可缩放的范围。



这些操作通过改变变量的值，影响了几何对象在渲染时的位置、大小和方向，从而实现了交互操作的效果。这些变量会在 display() 函数中影响矩阵变换和绘制不同的几何对象。

1. 光源的设置和位置移动

实现了使用OpenGL内置的光源功能，并用鼠标交互改变位置。

* 1. 实现原理

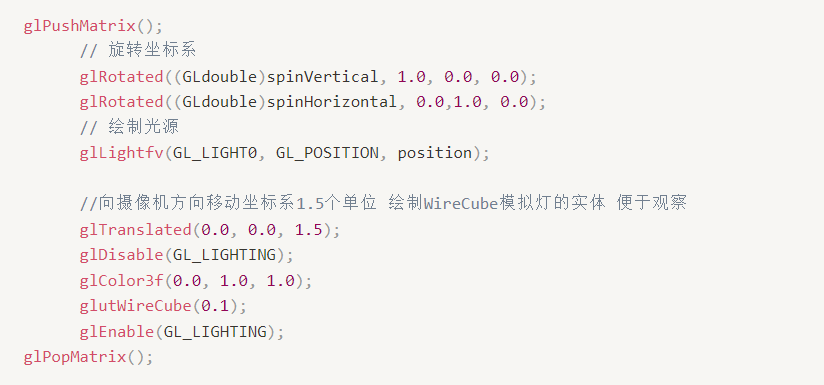
在启用光照后，OpenGL会考虑光照对物体表面的影响，根据设置的光源、材质等参数计算物体表面的颜色和亮度。

在init函数中调用glEnable(GL\_LIGHTING)启用光照功能，调用glEnable(GL\_LIGHT0)启用编号为0的光源。

接下来我们就可以进行绘制光源的操作了，在display中，橙色区域的区块便是对光源进行绘制的语句。



具体如下。



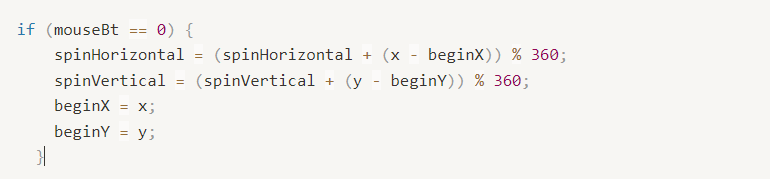
首先我们对即将要绘制光源的坐标系进行旋转，其中的参数spinVertical和spinHorizontal为设置的全局变量，方便在交互操作中进行更改。

glLightfv(GL\_LIGHT0, GL\_POSITION, position)设置光源 GL\_LIGHT0 的位置为 position 所表示的位置，position 是一个长度为4的数组，用来表示光源的位置。其中前三个分量表示光源在三维空间中的位置（x、y、z坐标），最后一个分量表示光源是点光源还是方向光源（0 表示方向光源，1 表示点光源）。设置GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 }。表示光源初始位置为物体的位置再向摄像机方向（z轴正方向）平移1.5个单位，在旋转的时候会围绕着物体在半径为1.5的球体上旋转。

为了更直观地表示光源，在相同的位置绘制一个线框正方体（wireCube），在绘制这个正方体的时候停用光源，使改正方体不受光源的影响。

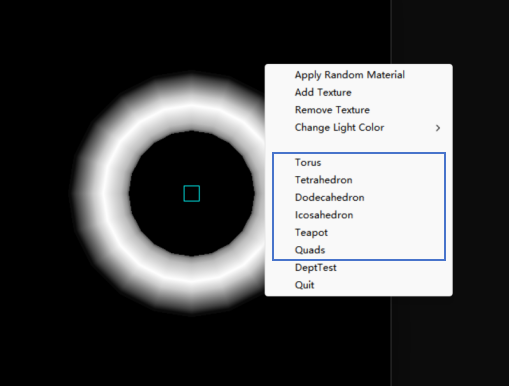
* 1. 交互操作的实现

和上面物体的旋转一样，通过鼠标移动的捕捉改变旋转变量。

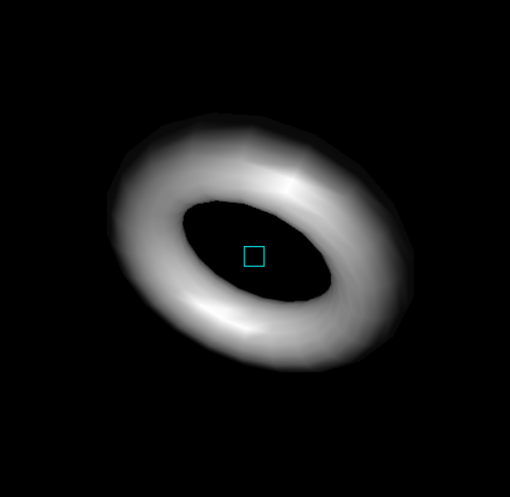
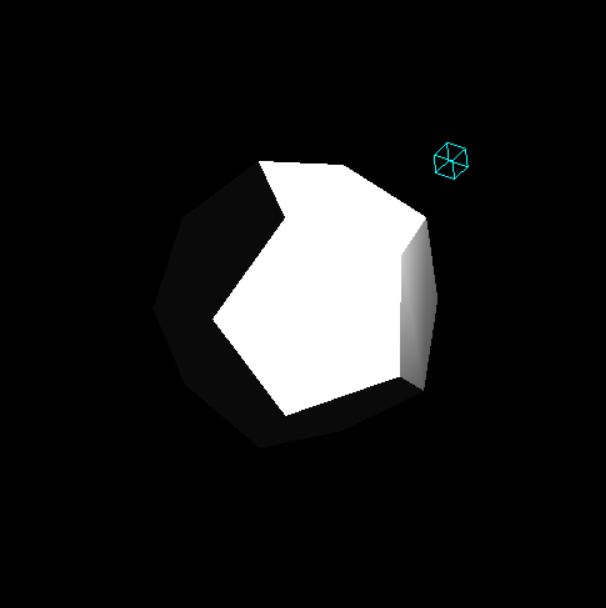


1. 几何对象加载

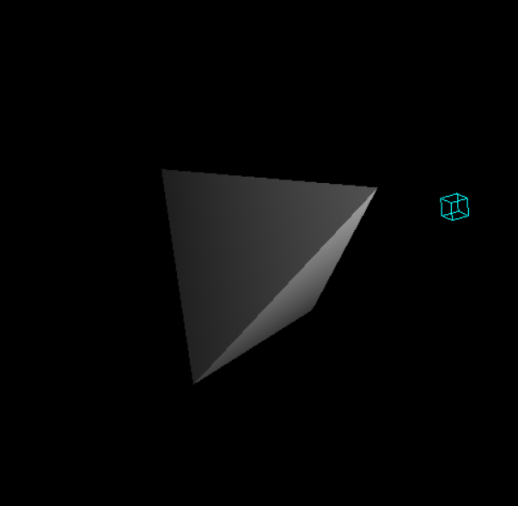
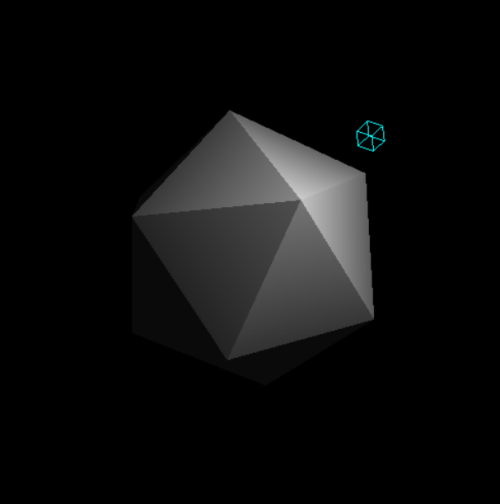
通过菜单选择不同几何对象进行加载。



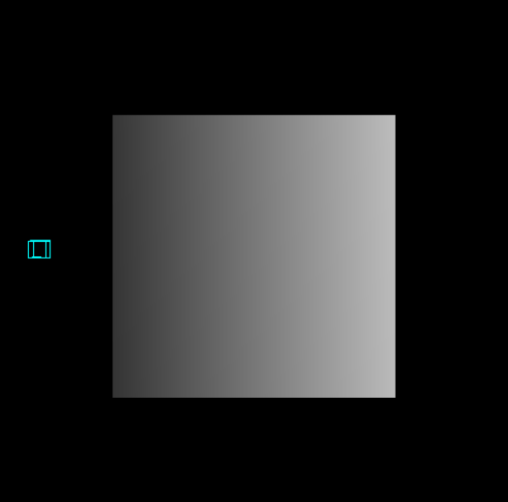
在 display() 函数中的 switch (obj) 语句下，根据变量 obj 的值不同选择不同的几何对象进行绘制。

glutSolidTorus() 圆环体 glutSolidDodecahedron() 十二面体

glutSolidTetrahedron() 实心四面体 glutSolidIcosahedron()实心二十面体

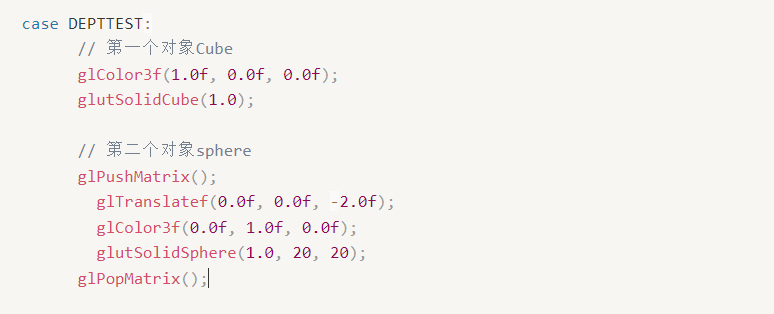
 

glutSolidTeapot()茶壶 定义顶点绘制四边形

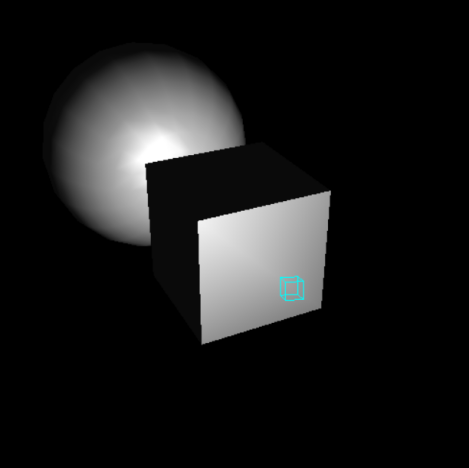
1. 消隐处理：

OpenGL提供了深度缓冲区（Z-buffer）来处理物体的深度顺序，使得远处的物体不会遮挡近处的物体。

在菜单栏的DeptTest选项下，可以选择绘制实心球体和实心正方体两个物体。



我们用这两个物体和表示光源的WireCube来做深度测试。如下为正常启用深度测试的情况。



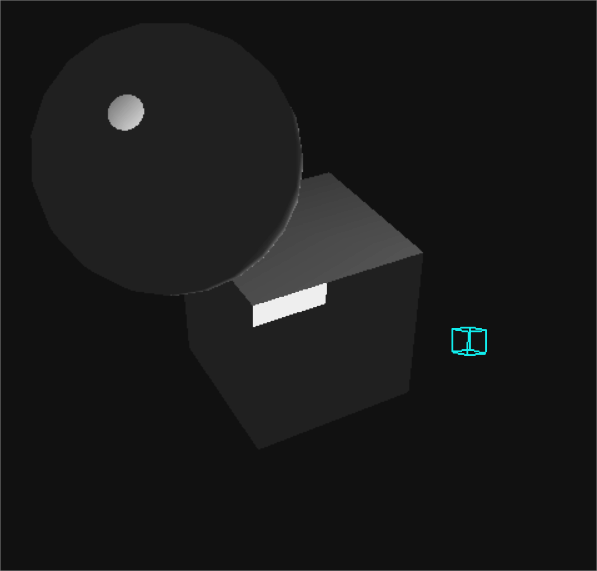
在init函数中调用：

glDepthFunc(GL\_LESS)；

glEnable(GL\_DEPTH\_TEST);

来启用深度测试。启用后OpenGL会在绘制像素时，检查每个像素的深度值，并将其与深度缓冲区中已经存在的深度值进行比较。如果像素的深度值小于深度缓冲区中的对应位置的深度值，那么这个像素就会被绘制到屏幕上，并更新深度缓冲区。使用 GL\_LESS 表示深度测试函数为“小于比较”。即，当前像素的深度值小于深度缓冲区中的值时才会通过深度测试，即离观察者更近的像素会被绘制，而距离观察者更远的像素会被深度测试丢弃。

关闭深度测试后，渲染结果如下。可以看到后绘制的物体会出现屏幕的前方。且单个物体的渲染也出现了问题，实体的物体看起来像是空心的，正方体后方的面没有被遮挡直接可见。

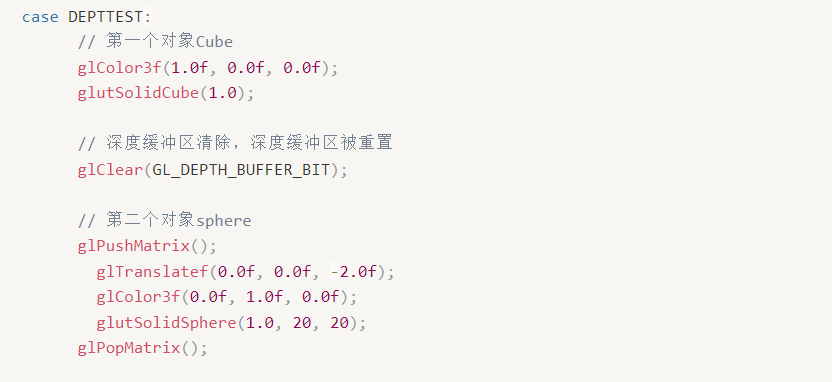


上图中球体的位置应该在正方体后方，却被渲染到了其前方。

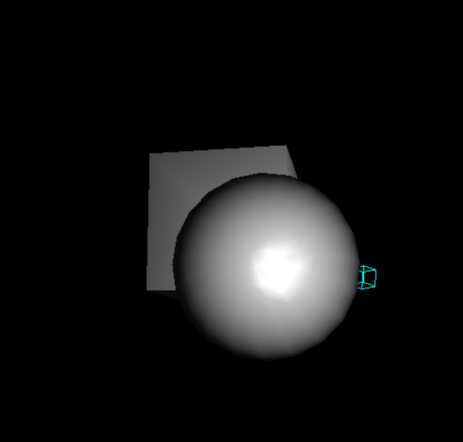
我们再用另一种方式测试。在绘制完正方体后调用：

glClear(GL\_DEPTH\_BUFFER\_BIT)

来使得深度缓冲区重置，此时再绘制球体。



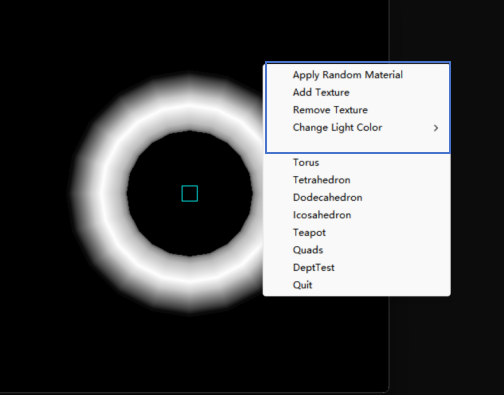
得到的结果如下图。



可以看到球体被渲染到所有物体的前面。这是因为清除深度缓冲意味着OpenGL会忘记之前绘制的正方体的深度信息，所有新的物体会被当作在旧的物体的前面。

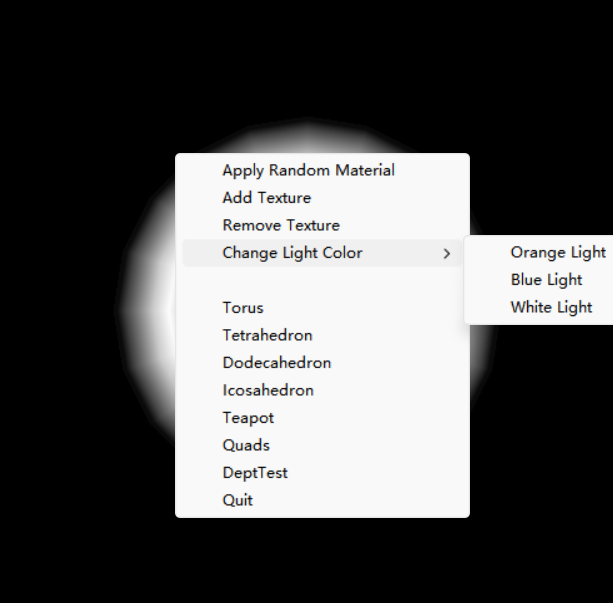
1. 光照、材质交互修改

通过菜单选项和鼠标交互，可以动态改变光源颜色，以及随机生成并应用新的材质属性到几何对象上。



* 1. 光照颜色

通过菜单选择“Change Light Color”来改变光源的颜色。根据选择的不同菜单项，设置了GL\_LIGHT0 的 GL\_DIFFUSE（漫反射光）属性为不同的颜色值，从而改变了光源照射到物体上的颜色。在这里设置了三种可选的灯光颜色（橙色、蓝色、白色）。



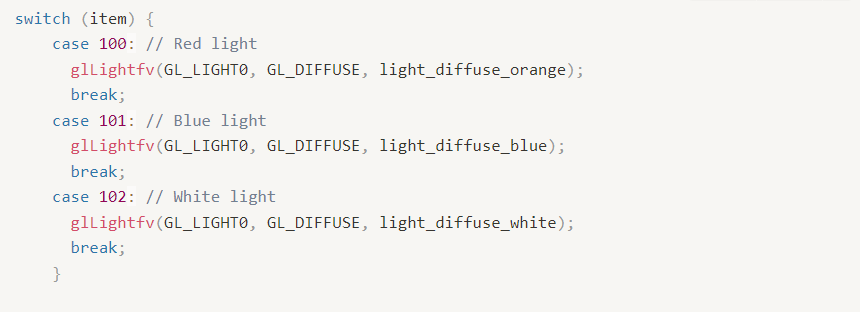
以glLightfv(GL\_LIGHT0, GL\_DIFFUSE, light\_diffuse\_orange)这行代码为例，说明设置光源颜色的方法。

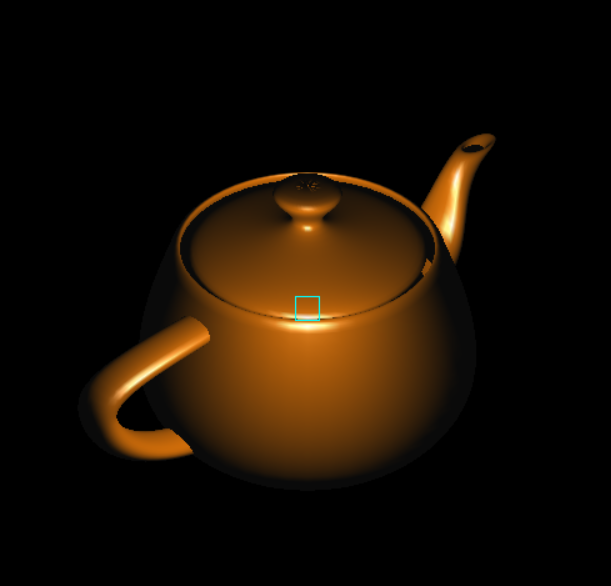
GL\_LIGHT0 是光源的编号，代表要修改的光源，这里是修改第一个光源的属性，也是该程序的场景中唯一启用的光源编号。

GL\_DIFFUSE 表示设置漫反射光属性。

light\_diffuse\_orange包含了橙色的 RGBA 颜色值。

该函数改变了光源的漫反射光属性，使光源发出的漫反射光是橙色的。光源 GL\_LIGHT0 发出的光线在与物体表面相交时会产生橙色的漫反射效果。





* 1. 物体材质

设置全局变量obj\_ambient、obj\_diffuse、obj\_specular 和 obj\_shininess。

在选择“Apply Random Material”菜单栏后，随机生成物体的环境光、漫反射、镜面反射颜色值以及高光指数，通过修改 obj\_ambient、obj\_diffuse、obj\_specular 和 obj\_shininess 这些属性参数，影响了物体的材质属性，进而改变了物体表面的光照反射和高光效果。

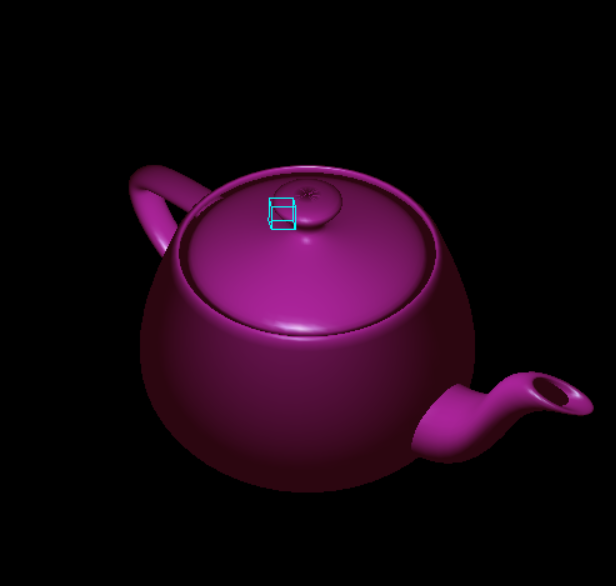
在绘制物体之前指定材料的参数：

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

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

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

glMaterialfv(GL\_FRONT, GL\_SHININESS, &obj\_shininess);



1. 纹理贴图：

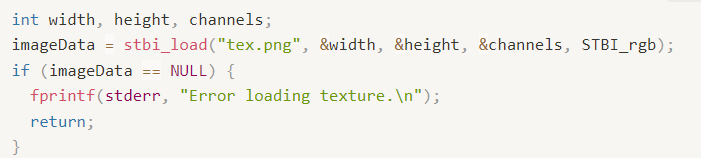
提供了添加和删除纹理的选项，并在渲染阶段启用纹理贴图。加载纹理图像并使用GLUT库的函数将纹理映射到几何对象表面。

* 1. 加载纹理图

项目中添加了外部库stb\_image，引入头文件stb\_image.h后，因为前面定义了指定的宏，可以直接当成cpp文件加载该文件中的函数使用。

#define STB\_IMAGE\_IMPLEMENTATION

#include "stb\_image.h"



使用 stbi\_load 函数从文件中加载纹理图像。"tex.png" 是要加载的纹理图像的文件路径。width、height、channels 用于存储图像的宽度、高度和通道数。STBI\_rgb 指定了加载图像的颜色通道格式。如果加载纹理失败，会打印错误消息并返回。

* 1. 创建Texture

glGenTextures(1, &texture)创建一个纹理对象，并将其 ID 存储在 texture 变量中。glBindTexture(GL\_TEXTURE\_2D, texture)将刚刚创建的纹理对象绑定到当前的纹理单元上。

* 1. 绑定纹理图到Texture并调节texture参数

使用 gluBuild2DMipmaps 将加载的图像数据生成 Mipmap。

设置了纹理的过滤方式：

glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR); 设置纹理的缩小过滤方式为三线性过滤。

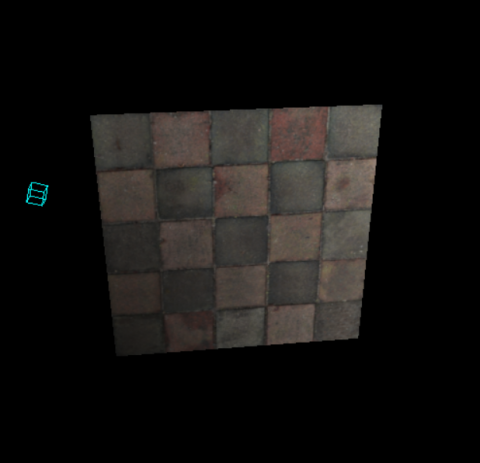
glTexParameteri(GL\_TEXTURE\_2D,

GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR); 设置纹理的放大过滤方式为线性过滤。

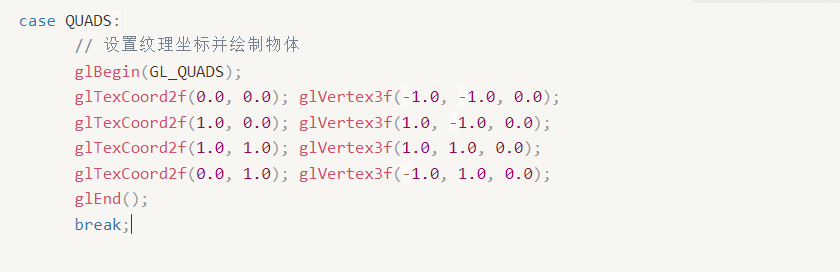
* 1. 纹理映射绘制在物体上

这里纹理映射采用了两种方式。

第一种是应用在Quad和Teapot上的，纹理相对物体的位置不变。

对Quad，应用自定义的纹理映射，通过为几何图元的顶点指定纹理坐标来实现。



glTexCoord2f(0.0, 0.0) 设置了纹理坐标为 (0.0, 0.0)，即纹理的左下角。

glVertex3f(-1.0, -1.0, 0.0) 设置了顶点的三维空间坐标为 (-1.0, -1.0, 0.0)。

它们共同完成绘制一个位于三维空间中 (-1.0, -1.0, 0.0) 位置的顶点，并将纹理的左下角贴到这个顶点上。其他三个顶点也类似地指定了纹理坐标和顶点位置，最终构成了一个四边形，并根据设置的纹理坐标，将纹理图像映射到这个四边形上。

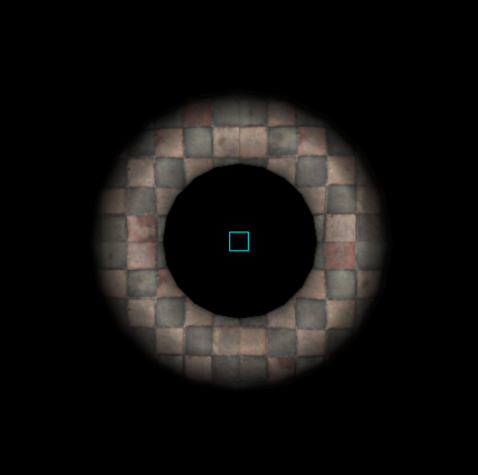
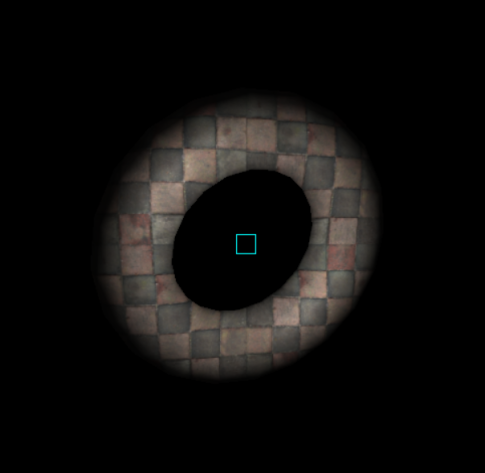
对于Teapot，glut给了已有的纹理映射方式，只需要启用纹理并绘制即可。

第二种是应用在其他物体上的，纹理相对物体的位置改变。

这种映射方式为自动纹理映射，使用纹理坐标自动生成时，OpenGL会自动为顶点生成纹理坐标，而不需要手动设置。不同的自动生成方式可以根据对象的几何特征来确定纹理坐标，比如球体、圆环等可以利用其表面的特性自动生成纹理坐标。

但是这种方式看起来就像是纹理投影在了物体上一样，当改变物体的属性（平移、旋转、缩放）时，纹理并不会跟着物体一起变化。

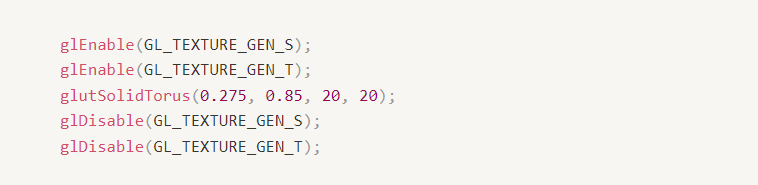
 

以上图中圆环体的纹理贴图绘制过程为例：

glEnable(GL\_TEXTURE\_GEN\_S)和glEnable(GL\_TEXTURE\_GEN\_T) 启用了纹理坐标的自动生成。GL\_TEXTURE\_GEN\_S 和 GL\_TEXTURE\_GEN\_T 分别代表了纹理坐标的 S 和 T 轴（对应于纹理坐标的 x 和 y 轴）。

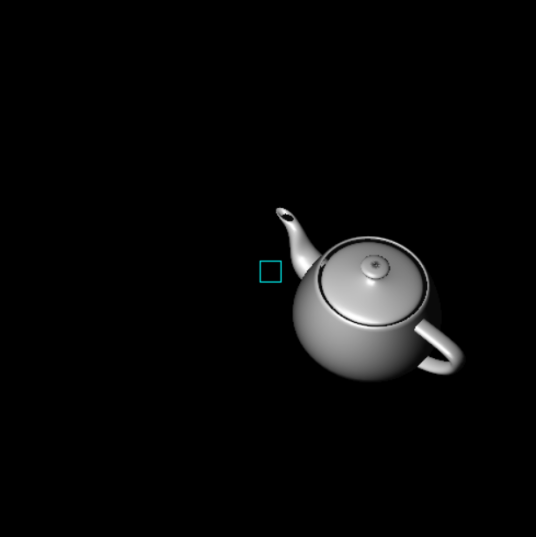
glutSolidTorus(0.275, 0.85, 20, 20) 绘制了一个实心的环面，但不像一般的纹理映射方式那样手动为每个顶点指定纹理坐标。

glDisable(GL\_TEXTURE\_GEN\_S)和glDisable(GL\_TEXTURE\_GEN\_T) 关闭了纹理坐标的自动生成。

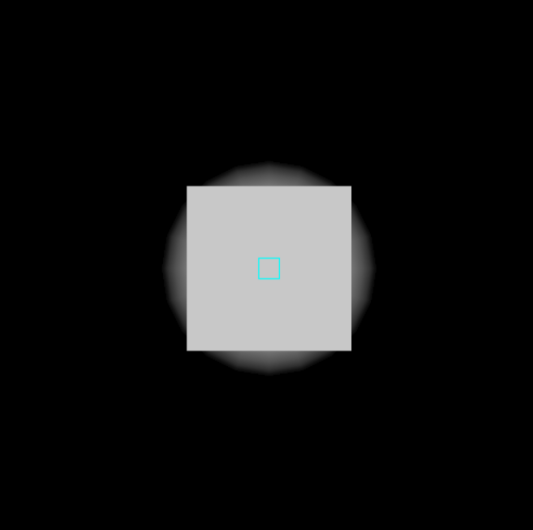
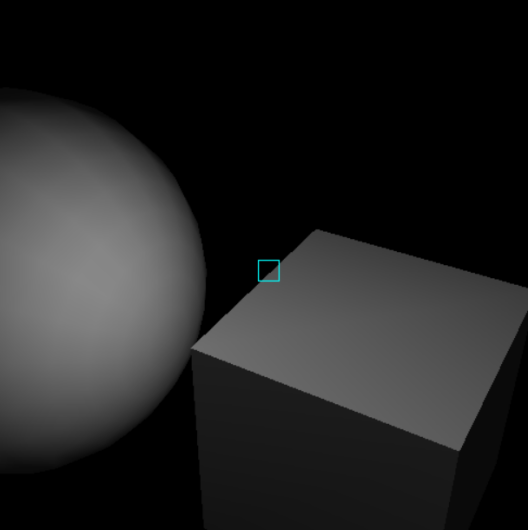


实验结果

1. 物体的平移、旋转和缩放操作

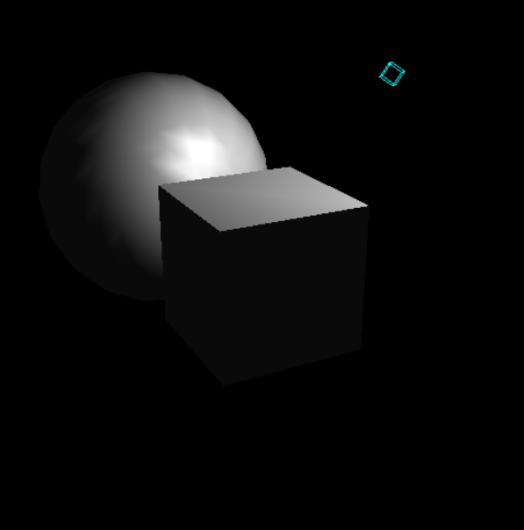
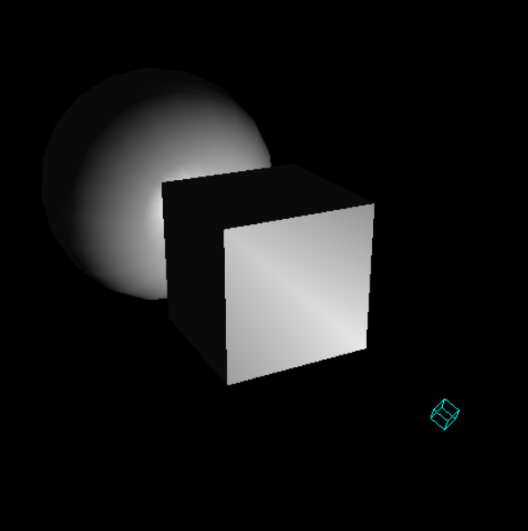
 

左图通过平移旋转和缩小变到右图。

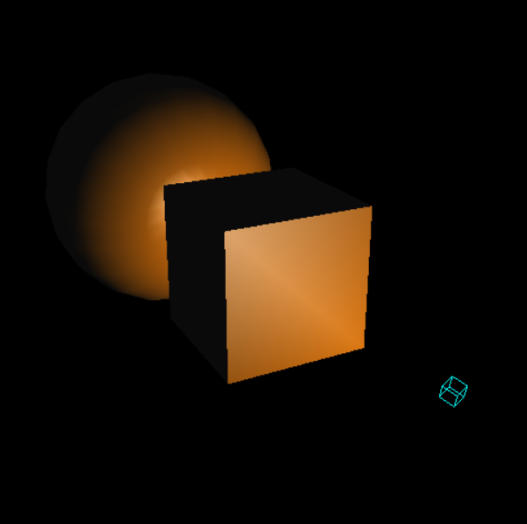
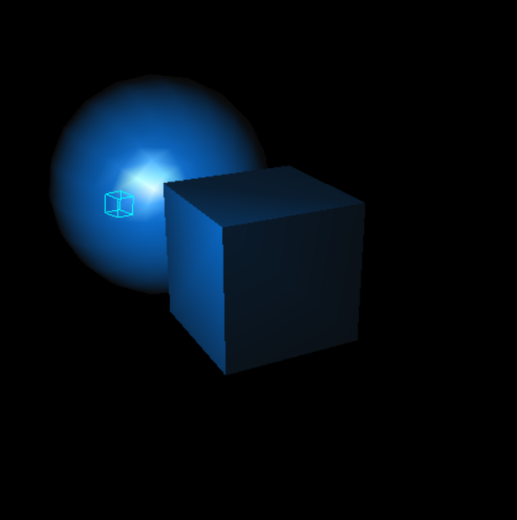
左图通过平移旋转和放大变到右图。

1. 光源的移动

1. 光源颜色改变

分别应用橙色的光和蓝色的光。

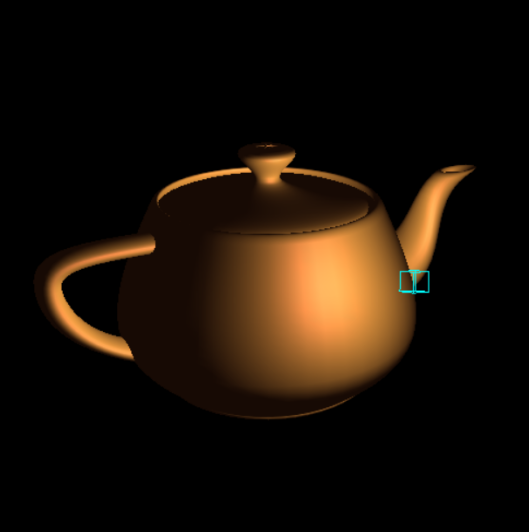
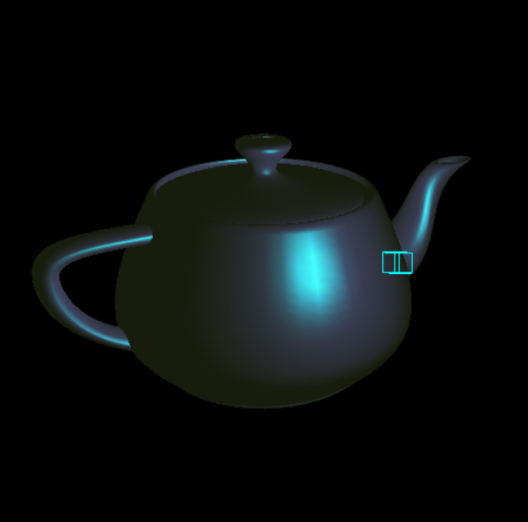
 

1. 物体材质改变

四种随机材质的应用结果图。

可以看到颜色、反射强度、反射区域大小等材质属性的区别。

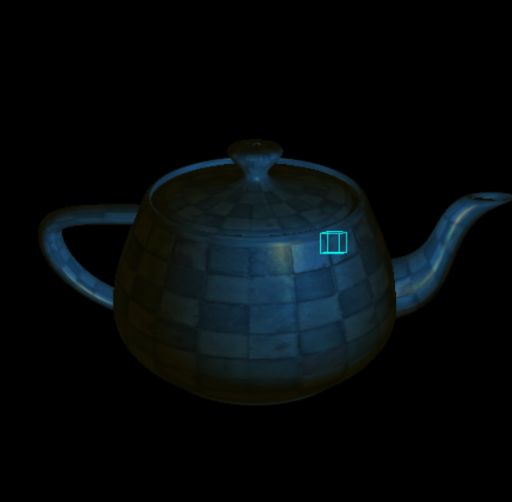
 

同一材质，左图为白光照射，右图为橙色光照射。

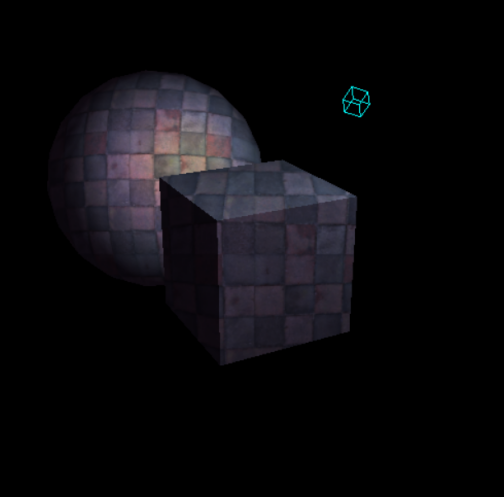
 

1. 纹理贴图

应用随机材质后再贴图，结果如图。

对多个物体应用材质。



结论分析

实验中使用了光照、材质、纹理等功能，允许用户在窗口中绘制不同的几何对象，并通过菜单选择更改光源颜色、应用随机材质、添加/删除纹理等。同时，用户可以通过鼠标和键盘实现对象的旋转、平移和缩放。

在纹理贴图部分，初始化纹理并将其应用于不同的几何图形，通过手动指定顶点和纹理坐标，成功实现了一个矩形的纹理映射，但无法直接用于固定功能管线的 glutSolidCube() 和 glutSolidSphere()。使用 glTexCoord2f() 和 glVertex3f() 可以手动指定顶点和纹理坐标，但这种方式并不适用于固定功能管线的某些绘制函数，如 glutSolidCube() 和 glutSolidSphere()。这些函数是预定义的，无法直接控制每个顶点的坐标和纹理坐标。如果需要更精细的控制，可能需要自行编写函数进行顶点数据的生成和渲染。

现代OpenGL提供了更灵活的方式，如使用顶点缓冲对象（VBO）和着色器，能够更灵活地控制顶点数据和纹理坐标的映射关系，这也是程序未来改进的方向。

总之，本次实验使我了解了OpenGL的一些操作，但也呈现出了在固定功能管线下的一些限制，后续我将学习更多有关现代OpenGL的方式实现更多功能。