**《计算机图形学实验》综合实验报告**

**题目 茶壶的光照变换和纹理贴图**

**学 号 20201050422**

**姓 名 陈柱宏**

**指导教师 钱文华**

**日 期 2022年6月20日**

**摘要**

随着计算机技术的发展，对所辐射的各个领域的应用、完善和生产生 活中的所带来的便利做分析和调研。图形图像技术创造出了一系列新的工作方式，在我国社会上的很多领域中也已经开始使用这项技术。但是之前我们的茶壶光是黑白灰，太单调了，现在我们来给点颜色。为了能够给茶壶贴上纹理，首先我们得能够读取图片里的颜色值。计算机图形学——光照模型，光线照到物体表面发生三种情况：反射、投射、吸收转换为热能。 只有反射光和投射光能够刺激眼睛产生颜色，因而物体表面的反射光和投射光决定了物体呈现的颜色。使用OpenGL实现三维图形渲染，设计一个茶壶的三维图形，渲染过程加入纹理、光照等效果，采用了光照明模型、纹理贴图等算法。运用计算机图形学理论、算法、技术等要求设计并实现程序。

**关键词：**OpenGL；三维图形；茶壶

目录

**[1.](#_Toc105597052)****[实验背景](#_Toc105597052)** [1](#_Toc105597052)

**[2.](#_Toc105597053)****[实验内容](#_Toc105597053)** [1](#_Toc105597053)

**[3.](#_Toc105597054)****[开发工具，程序设计及实现目的](#_Toc105597054)** [1](#_Toc105597054)

**[4.](#_Toc105597055)****[关键算法理论](#_Toc105597055)** [2](#_Toc105597055)

**[5.](#_Toc105597056)****[实验结果及小结](#_Toc105597056)** [6](#_Toc105597056)

**[参考文献：](#_Toc105597057)** [8](#_Toc105597057)

**[附录：](#_Toc105597058)** [9](#_Toc105597058)

1. **实验背景**

在上世纪五十年代，计算机图形处理技术就开始被人们所使用 了，在这么多年对计算机进行深度探究的过程中，图形处理方面计 算机技术也变得越来越成熟，在图形图像及美学方面的社会需求也 推动了信息技术的发展。随着技术进一步的优化，人们在搜集信息 和处理信息方面也逐渐开始使用图形图像处理技术，在计算机技术 进一步完善的过程中，人们在图形图像处理技术方面取得了重大的 突破——相关技术更加符合当代社会发展和审美的需要，在众多领 域之中加强对图形图像处理技术的使用和研发，推动了我国当前的 生产与发展[1]。对于计算机图形学从业者，掌握变换****(transform)****是极其重要的。 掌握了transform，你就可以摆放物体 (objects)、灯光 (lights)和相机 (cameras)的位置，变换 (reshape)它们的形状，并使 (animate)它们动起来。 懂了transform，你就可以确保所有的计算是在同一个坐标系 (the same coordinate system)中进行，而不是错误地使用坐标系。 你还可以以不同的方式，把物体 (objects)投影到平面上。首先第一种就是我们游戏中常用的贴图纹理，我们开始用第三方软件做我们的需要的模型外观；我们也叫图像纹理：其实就是个贴图，里面存放二维纹理图案，那么上色的过程就是通过映射把纹理图案中的像素一一指定给三位物体的表面；纹理映射（Texture Mapping）：我们有了纹理，那么我们就可以把模型的表面通过纹理给覆盖上去，这样的话我们做模型的人就不要那么辛苦的去构造模型和材质细节，节省了大量人力和时间。如果我们站在像素级别角度，我们只是需要解析这个纹理粗糙的多边形对应和像素来替代详细的几何造型。我们通过数字化技术把纹理图案覆盖或者映射或者投射到三位物体的表面，给物体增加表面细节的过程叫纹理映射。纹理映射的思想：我们需要寻找一种纹理空间（U,V）坐标到三维物体曲面（s,t）坐标之间的映射关系，将纹理空间对应的坐标上对应的彩色参数值映射给曲面上，从而给曲面得到彩色的纹理细节。

1. **实验内容**

利用Visual C++, OpenGL, Java等工具，实现三维图形渲染，自定义三维图形，三维图形不能仅仅是简单的茶壶、球体、圆柱体、圆锥体等图形，渲染过程须加入纹理、色彩、光照、阴影、透明等效果，可采用光线跟踪、光照明模型、纹理贴图、纹理映射等算法。

1. **开发工具，程序设计及实现目的**

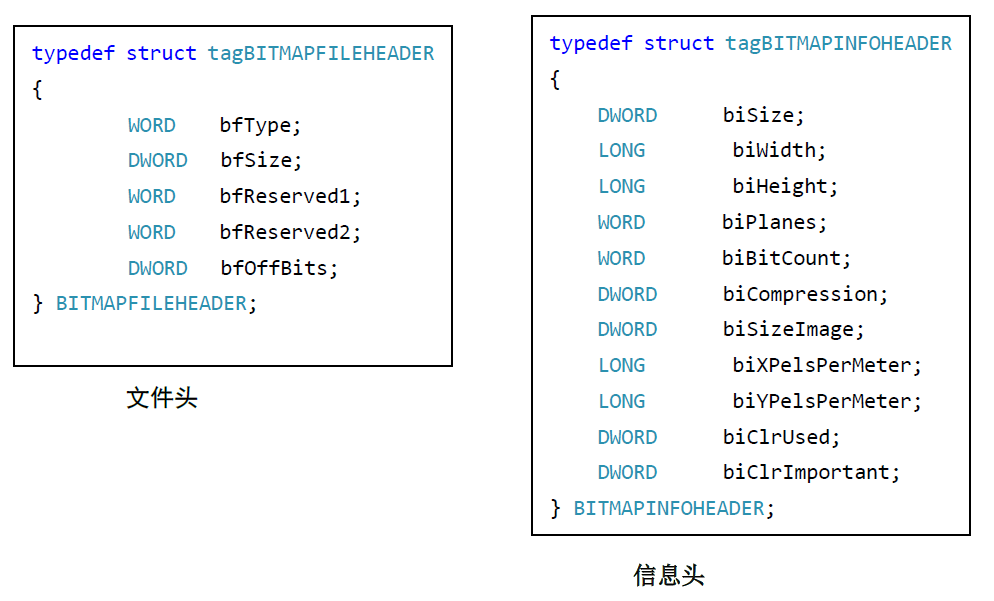
开发工具：Visual C++, OpenGL等工具；

**4.****关键算法理论**

1. **图像纹理贴图**

**读纹理图片：**

利用C标准库中的fopen，以二进制只读形式加载图像。filePtr = fopen( filename , "rb" ); 如果把位图看作数据流的话，我们会看到它的前几位存储了关于位图的一些信息，如大小、类型、地址、色深、压缩等信息，我们把这几位信息映射到程序中位图文件头结构体以及信息头结构体中。



映射完成后，再把剩下的位图具体数据存储起来，然后交换像素信息保证最后以RGB形式存储。

**加载纹理：**

在已经读入图片后，我们用bitmapData指针指向图片信息，然后开始纹理加载；首先调用如下函数进行纹理与纹理存储对象的绑定：glBindTexture(GL\_TEXTURE\_2D, texture);接下来指定过滤方式：//指定当前纹理的放大/缩小过滤方式

glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_MAG\_FILTER,GL\_NEAREST) ；

glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_MIN\_FILTER,GL\_NEAREST)；

我们注意到这里使用了两个参数：GL\_TEXTURE\_MAG\_FILTER和GL\_TEXTURE\_MIN\_FILTER。前者代表了待映射纹理像素少于一个纹理单元的像素时，以怎样的方式映射；后者代表了待映射纹理像素多于一个纹理单元的像素时，以怎样的方式映射。第三个参数我们使用了GL\_NEAREST和GL\_LINEAR两种，前者取4个坐标上最接近待映射纹理像素的颜色，后者取坐标上带映射纹理接近像素的平均值。（曼哈顿距离）

调用以下函数可以指定纹理图片的重复。

glTexParameteri( GL\_TEXTURE\_2D , GL\_TEXTURE\_WRAP\_S , GL\_REPEAT ); //S 方向重复

glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_WRAP\_T,GL\_REPEAT);//T方向重复

**纹理贴图：**

加载好纹理并做好预处理后，我们可以把纹理和物体结合了，只要把绘制物体的函数放在中间部分即可：

glEnable( GL\_TEXTURE\_2D );

glBindTexture(GL\_TEXTURE\_2D, texture[index]); //选择纹理texture[index]

glPushMatrix();

……

glPopMatrix();

glDisable(GL\_TEXTURE\_2D);//关闭纹理texture[index]

对于立方体，本次实验需要自己实现具体代码，纹理映射需要指出边界，所以在绘制立方体每个面时，我们都设定它的四个边界点：

glBegin(GL\_QUADS);

for (i = 0; i < 6; i++) {

for (j = 0; j < 4; j++) {

glTexCoord2iv(dir[j]);//设定边界点

glVertex3fv(point[i][j]);

}

}

glEnd();

注意到实验中存在光照，所以我们有必要指定纹理和光照是什么关系，在这里我们用了如下两个参数，一个保证纹理和光照混合，一个单纯绘制纹理而不考虑光照。

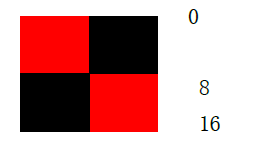
glTexEnvf( GL\_TEXTURE\_ENV , GL\_TEXTURE\_ENV\_MODE , GL\_MODULATE ); // 设置纹理受光照影响

glTexEnvf(GL\_TEXTURE\_ENV,GL\_TEXTURE\_ENV\_MODE,GL\_DECAL);//设置纹理不受光照影响

我们在绘制不同对象时调用不同的函数，这样就能实现不同的纹理与光照结合的效果。

**自定义贴图：**

如果需要自己生成纹理贴图，我们不再停留在图形世界，而是把目光投向图像世界，也就是说，我们需要考虑生成的不是点线面，而是一个个像素点，这意味着我们最好不再使用像glrectf这样的函数来绘制。和RGB图像一样，我们建立一个三维数组来存储图像信息，直接赋值每个像素点即可。待生成纹理的最小单位:



当然我们的纹理图片可以是这个最小单位的若干倍。在本次实验中，设长宽都为16个像素点。选16这个数字是有目的的。我们可以发现这样的规律：

1.0-7 这8个数字的二进制表示中，右数第4位为0

2.8-15这8个数字的二进制表示中，右数第4位为1

所以我们得到这样的结论：x&8 == 1，坐标大于8，x&8 == 0坐标小于8。因而，(x&8)^(y&8) = 1时，填充黑色，(x&8)^(y&8) = 0时，填充红色（反过来也可以），通过位运算，我们只用六行就完成了这个纹理，具体见代码[2]。

1. **图像光照变换：**

OpenGL可以设置至少8种光源，它们的标号为GL\_LIGHT0,GL\_LIGHT1,GL\_LIGHT2……。在这里我们使用了两种光源，一种是环境光，另一种是聚光灯。在设置光照时，我们需要考虑这样三种光：环境反射光、镜面反射光、漫反射光。在Phong光照模型中，就是通过这三种分量的取值来模拟真实光照的。其中，环境反射光是光源多次反射后的光，可以理解为背景光，镜面反射和漫反射反映了物体表面的粗糙/光滑程度，两者的和是一定的[3]。在OpenGL中，如果我们想要使用光源，需要输入以下语句开启光照模式：

glEnable( GL\_LIGHTING ); // 开启光照模式

对于一个光源，我们首先需要确定它的位置，这个光源可以是点光源，也可以是平行光源，我们写如下函数，如果是点光源，第四个参数为1，前三个参数代表光源位置；如果是平行光源，第四个参数是0，前三个参数表示方向。其中light\_pos是一个数组。在本次试验中，我们采用的是点光源。

glLightfv(GL\_LIGHT0,GL\_POSITION, light\_pos);//设置第0号光源的光照位置

同时，设置漫反射、镜面反射、环境反射的颜色参数：

glLightfv(GL\_LIGHT0,GL\_SPECULAR, color);//设置镜面反射光照

glLightfv(GL\_LIGHT0,GL\_DIFFUSE, color); //设置漫射光成分

glLightfv(GL\_LIGHT0,GL\_AMBIENT, color); //设置第0号光源多次反射后的光照颜色（环境光颜色）

在本次实验中，我们把环境光设为指定的颜色（在这里，我们选择了白色和绿色），而镜面反射光和漫反射光都设置为白色。使用如下函数，开启灯光：

glEnable(GL\_LIGHT0);//开启第0号光源

聚光灯我们另外使用GL\_LIGHT1光源。除了基本的三种光，还包含了其他属性，分别是裁剪角度、光源方向和聚集度：

glLightf(GL\_LIGHT1,GL\_SPOT\_CUTOFF, spotangle); //裁减角度

glLightfv(GL\_LIGHT1,GL\_SPOT\_DIRECTION, lightDir); //光源方向

glLightf(GL\_LIGHT1,GL\_SPOT\_EXPONENT, 2.); //聚集度

对于聚光灯，我们设置和环境光一样。在设置完了基本的光照后，我们来关注物体的材质，这里的材质并不是指我们理解意义上的材质，而是物体在不同光照下反射的颜色。我们知道，物体的显示效果和光照有着很大的关联，比如在黑暗中，我们看到的物体就是黑色的。我们所看到的物体颜色是由灯光以及物体反射颜色共同决定的。

glMaterialfv(GL\_FRONT\_AND\_BACK,GL\_SPECULAR, color);

glMaterialfv(GL\_FRONT\_AND\_BACK,GL\_DIFFUSE, color);

glMaterialfv(GL\_FRONT\_AND\_BACK,GL\_AMBIENT, color);

其中，第一个参数代表了操作对象的正面还是反面，参数包括GL\_FRONT、GL\_BACK、GL\_FRONT\_AND\_BACK；第二个参数代表了材质的颜色是在什么光照下表现的；第三个参数是光照下材质的颜色。对于茶壶，我们设置镜面指数为50，镜面反射为0.6f, 0.6f, 0.6f,漫反射为0.85f, 0.65f, 0.2f。对于桌面和腿，我把镜面反射和漫反射的颜色设为相同的指定颜色。

**5.实验结果及小结**

实验结果：

纹理实验：

纹理1：

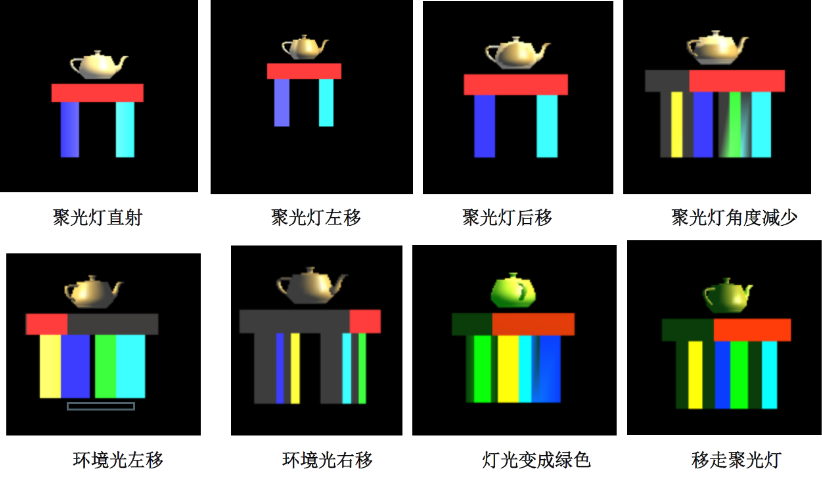


纹理2：



**光照实验：**

按下a图像整体左移，按下d图像图像整体右移，按下w图像整体上移，按下s图像整体下移，按下z图像整体前移，按下c图像整体后移，按下j环境光左移，按下l环境光右移，按下i环境光上移，按下k环境光下移，按下n环境光前移，按下m环境光后移，按下f聚光灯左移，按下h聚光灯右移，按下t聚光灯上移，按下g聚光灯下移，按下v聚光灯后移，按下b聚光灯前移，按下x聚光灯角度变大，按下y聚光灯角度变小，按下q退出程序：



**实验总结**

通过这次的期末实验我对计算机图形学真实感图形有了更加深入的了解，虽然实验过程中碰到了各种各样的问题，例如有些函数如fopen（）居然会报错，上网查阅相关资料得知是环境配置的问题需要升级一下环境才行，另外，目前光照实验中桌子腿的效果不太真实，但是实验要求中也没有要求了多种颜色，所以没有做过多优化。可以更改一下数据设置，让桌子腿的效果更加真实，纹理贴图实验中我觉得最大的难点是将图片导入进去，一开始我没注意到图片要.BMP格式的才能导入并使用所以绕了一些弯子；在解决各种各样实验问题的过程中我的编程水平有了很大提高，对计算机图形学的关照变换和纹理贴图这种实用性非常强的方法有了很大兴趣，希望我在今后的时间里再接再厉，努力提高自己的水平。

**参考文献：**

1. 赵靖.计算机图形学与图形图像处理技术的应用[J].电子技术与软件工程,2021(10):113-114.
2. 吴峥,肖晗.基于C++平台的计算机图形学算法实现[J].电子制作,2013(09):81.DOI:10.16589/j.cnki.cn11-3571/tn.2013.09.027.
3. 肖源源,王子牛.基于VC++的OpenGL纹理映射技术的研究与实现[J].贵州大学学报(自然科学版),2008(02):158-160.

**附录：**

纹理和贴图实验代码及注释：

#include <stdlib.h>

#include <windows.h>

#include <stdio.h>

#include "GL/glut.h"

#define BITMAP\_ID 0x4D42

#define Height 16

#define Width 16

GLubyte image[Height][Width][3]; // 图像数据

float fTranslate;

float fRotate;

float fScale = 1.0f; // set inital scale value to 1.0f

int status = 0;

int status2 = 1;

bool bPersp = false;

bool bAnim = false;

bool bWire = false;

int wHeight = 0;

int wWidth = 0;

GLuint texture[3];

void Draw\_Leg();

// 纹理标示符数组，保存两个纹理的标示符

// 描述: 通过指针，返回filename 指定的bitmap文件中数据。

// 同时也返回bitmap信息头.（不支持-bit位图）

//读纹理图片

unsigned char\* LoadBitmapFile(char\* filename, BITMAPINFOHEADER\* bitmapInfoHeader)

{

FILE\* filePtr; // 文件指针

BITMAPFILEHEADER bitmapFileHeader; // bitmap文件头

unsigned char\* bitmapImage; // bitmap图像数据

int imageIdx = 0; // 图像位置索引

unsigned char tempRGB; // 交换变量

// 以“二进制+读”模式打开文件filename

filePtr = fopen(filename, "rb");

if (filePtr == NULL) {

printf("file not open\n");

return NULL;

}

// 读入bitmap文件图

fread(&bitmapFileHeader, sizeof(BITMAPFILEHEADER), 1, filePtr);

// 验证是否为bitmap文件

if (bitmapFileHeader.bfType != BITMAP\_ID) {

fprintf(stderr, "Error in LoadBitmapFile: the file is not a bitmap file\n");

return NULL;

}

// 读入bitmap信息头

fread(bitmapInfoHeader, sizeof(BITMAPINFOHEADER), 1, filePtr);

// 将文件指针移至bitmap数据

fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK\_SET);

// 为装载图像数据创建足够的内存

bitmapImage = new unsigned char[bitmapInfoHeader->biSizeImage];

// 验证内存是否创建成功

if (!bitmapImage) {

fprintf(stderr, "Error in LoadBitmapFile: memory error\n");

return NULL;

}

// 读入bitmap图像数据

fread(bitmapImage, 1, bitmapInfoHeader->biSizeImage, filePtr);

// 确认读入成功

if (bitmapImage == NULL) {

fprintf(stderr, "Error in LoadBitmapFile: memory error\n");

return NULL;

}

//由于bitmap中保存的格式是BGR，下面交换R和B的值，得到RGB格式

for (imageIdx = 0; imageIdx < bitmapInfoHeader->biSizeImage; imageIdx += 3) {

tempRGB = bitmapImage[imageIdx];

bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];

bitmapImage[imageIdx + 2] = tempRGB;

}

// 关闭bitmap图像文件

fclose(filePtr);

return bitmapImage;

}

//加载纹理的函数

void texload(int i, char\* filename)

{

BITMAPINFOHEADER bitmapInfoHeader; // bitmap信息头

unsigned char\* bitmapData; // 纹理数据

bitmapData = LoadBitmapFile(filename, &bitmapInfoHeader);

glBindTexture(GL\_TEXTURE\_2D, texture[i]);

// 指定当前纹理的放大/缩小过滤方式

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexImage2D(GL\_TEXTURE\_2D,

0, //mipmap层次(通常为，表示最上层)

GL\_RGB, //我们希望该纹理有红、绿、蓝数据

bitmapInfoHeader.biWidth, //纹理宽带，必须是n，若有边框+2

bitmapInfoHeader.biHeight, //纹理高度，必须是n，若有边框+2

0, //边框(0=无边框, 1=有边框)

GL\_RGB, //bitmap数据的格式

GL\_UNSIGNED\_BYTE, //每个颜色数据的类型

bitmapData); //bitmap数据指针

}

void generateTex()

{

//生成红黑相间的图像

for (int i = 0; i < Height; i++) {

for (int j = 0; j < Width; j++) {

int x = ((i & 4) ^ (j & 4)) \* 255;

image[i][j][0] = (GLubyte)x;

image[i][j][1] = 0;

image[i][j][2] = 0;

}

}

}

//定义纹理的函数

void init()

{

glGenTextures(3, texture); // 第一参数是需要生成标示符的个数, 第二参数是返回标示符的数组

texload(0, "Monet.bmp");

texload(1, "Crack.bmp");

// texload(3, "Spot.bmp");

//下面生成自定义纹理

generateTex();

glBindTexture(GL\_TEXTURE\_2D, texture[2]);

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1); //设置像素存储模式控制所读取的图像数据的行对齐方式.

glTexImage2D(GL\_TEXTURE\_2D, 0, 3, Width, Height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, image);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);//放大过滤，线性过滤

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);//缩小过滤，线性过滤

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);//S方向重复

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);//T方向重复

}

void drawCube()

{

int i, j;

const GLfloat x1 = -0.5, x2 = 0.5;

const GLfloat y1 = -0.5, y2 = 0.5;

const GLfloat z1 = -0.5, z2 = 0.5;

//指定六个面的四个顶点，每个顶点用3个坐标值表示

GLfloat point[6][4][3] = {

{ { x1,y1,z1 },{ x2,y1,z1 },{ x2,y2,z1 },{ x1,y2,z1 } },

{ { x1,y1,z1 },{ x2,y1,z1 },{ x2,y1,z2 },{ x1,y1,z2 } },

{ { x2,y1,z1 },{ x2,y2,z1 },{ x2,y2,z2 },{ x2,y1,z2 } },

{ { x1,y1,z1 },{ x1,y2,z1 },{ x1,y2,z2 },{ x1,y1,z2 } },

{ { x1,y2,z1 },{ x2,y2,z1 },{ x2,y2,z2 },{ x1,y2,z2 } },

{ { x1,y1,z2 },{ x2,y1,z2 },{ x2,y2,z2 },{ x1,y2,z2 } }

};

int dir[4][2] = { {1,1},{1,0},{0,0},{0,1} };

//设置正方形绘制模式

glBegin(GL\_QUADS);

for (i = 0; i < 6; i++) {

for (j = 0; j < 4; j++) {

glTexCoord2iv(dir[j]);

glVertex3fv(point[i][j]);

}

}

glEnd();

}

void Draw\_Triangle() // This function draws a triangle with RGB colors

{

glEnable(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, texture[status]); //选择纹理texture[status]

glPushMatrix();

glTranslatef(0, 0, 4 + 1);

glRotatef(90, 1, 0, 0);

glTexEnvf(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_MODULATE);//设置纹理受光照影响

glutSolidTeapot(1);

glPopMatrix();

glDisable(GL\_TEXTURE\_2D); //关闭纹理texture[status]

glEnable(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, texture[status2]); //选择纹理texture[status2]

glTexEnvf(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_DECAL);//设置纹理不受光照影响

glPushMatrix();

glTranslatef(0, 0, 3.5);

glScalef(5, 4, 1);

drawCube();

glPopMatrix();

glPushMatrix();

glTranslatef(1.5, 1, 1.5);

Draw\_Leg();

glPopMatrix();

glPushMatrix();

glTranslatef(-1.5, 1, 1.5);

Draw\_Leg();

glPopMatrix();

glPushMatrix();

glTranslatef(1.5, -1, 1.5);

Draw\_Leg();

glPopMatrix();

glPushMatrix();

glTranslatef(-1.5, -1, 1.5);

Draw\_Leg();

glPopMatrix();

glDisable(GL\_TEXTURE\_2D); //关闭纹理texture[status2]

}

void Draw\_Leg()

{

glScalef(1, 1, 3);

drawCube();

}

void updateView(int width, int height)

{

glViewport(0, 0, width, height);//设置视窗大小

glMatrixMode(GL\_PROJECTION);//设置矩阵模式为投影

glLoadIdentity(); //初始化矩阵为单位矩阵

float whRatio = (GLfloat)width / (GLfloat)height; //设置显示比例

if (bPersp) {

gluPerspective(45.0f, whRatio, 0.1f, 100.0f); //透视投影

//glFrustum(-3, 3, -3, 3, 3,100);

}

else {

glOrtho(-3, 3, -3, 3, -100, 100); //正投影

}

glMatrixMode(GL\_MODELVIEW); //设置矩阵模式为模型

}

void reshape(int width, int height)

{

if (height == 0) //如果高度为0

{

height = 1; //让高度为1（避免出现分母为0的现象）

}

wHeight = height;

wWidth = width;

updateView(wHeight, wWidth); //更新视角

}

void idle()

{

glutPostRedisplay();

}

float eye[] = { 0, 0, 8 };

float center[] = { 0, 0, 0 };

void key(unsigned char k, int x, int y)

{

switch (k)

{

case 27:

case 'q': {exit(0); break; }

case 'p': {bPersp = !bPersp; break; }

case ' ': {bAnim = !bAnim; break; }

case 'o': {bWire = !bWire; break; }

case 'a': {

eye[0] -= 0.2f;

center[0] -= 0.2f;

break;

}

case 'd': {

eye[0] += 0.2f;

center[0] += 0.2f;

break;

}

case 'w': {

eye[1] -= 0.2f;

center[1] -= 0.2f;

break;

}

case 's': {

eye[1] += 0.2f;

center[1] += 0.2f;

break;

}

case 'z': {

eye[2] -= 0.2f;

center[2] -= 0.2f;

break;

}

case 'c': {

eye[2] += 0.2f;

center[2] += 0.2f;

break;

}

case 'r': { //切换茶壶纹理

if (status == 0)status = 2;

else if (status == 2)status = 0;

break;

}

case 't': { //切换桌子纹理

/\*.......\*/

}

}

updateView(wHeight, wWidth);//更新视角

}

void redraw()

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);//清除颜色缓存和深度缓存

glLoadIdentity(); //初始化矩阵为单位矩阵

gluLookAt(eye[0], eye[1], eye[2],

center[0], center[1], center[2],

0, 1, 0); // 场景（0，0，0）的视点中心 (0,5,50)，Y轴向上

if (bWire) {

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

//设置多边形绘制模式：正反面，线型

}

else {

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

//设置多边形绘制模式：正反面，填充

}

glEnable(GL\_DEPTH\_TEST);//开启深度测试

glEnable(GL\_LIGHTING); //开启光照模式

GLfloat white[] = { 1.0, 1.0, 1.0, 1.0 };

GLfloat light\_pos[] = { 5,5,5,1 };

glLightfv(GL\_LIGHT0, GL\_POSITION, light\_pos);//光源位置

glLightfv(GL\_LIGHT0, GL\_AMBIENT, white);//定义白色

glEnable(GL\_LIGHT0);//开启第0号光源

glRotatef(fRotate, 0, 1.0f, 0); //旋转

glRotatef(-90, 1, 0, 0);

glScalef(0.2, 0.2, 0.2);//缩放

Draw\_Triangle();//绘制场景

if (bAnim) fRotate += 0.5f;//旋转因子改变

glutSwapBuffers(); //交换缓冲区

}

int main(int argc, char\* argv[])

{

glutInit(&argc, argv);//对glut的初始化

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

//初始化显示模式:RGB颜色模型，深度测试，双缓冲

glutInitWindowSize(480, 480);//设置窗口大小

int windowHandle = glutCreateWindow("Simple GLUT App");//设置窗口标题

glutDisplayFunc(redraw); //注册绘制回调函数

glutReshapeFunc(reshape); //注册重绘回调函数

glutKeyboardFunc(key); //注册按键回调函数

glutIdleFunc(idle);//注册全局回调函数：空闲时调用

init(); //初始化纹理

glutMainLoop(); // glut事件处理循环

return 0;

}

光照变换实验代码及注释;

#include <stdlib.h>

#include "GL/glut.h"

#include <stdio.h>

float fTranslate;

float fRotate;

float fScale = 1.0f;

float color\_b = 1.0f;

bool bPersp = false;

bool bAnim = false;

bool bWire = false;

bool isWhite = true;

int wHeight = 0;

int wWidth = 0;

GLfloat color[] = { 1.0, 1.0, 1.0, 1.0 }; // 定义颜色

float eye[] = { 0, 0, 8 };

float center[] = { 0, 0, 0 };

GLfloat spotangle = 5.0f; //角度

//环境光位置

GLfloat light\_x = 0.0f;

GLfloat light\_y = 0.0f;

GLfloat light\_z = 0.0f;

//聚光灯方向

GLfloat dir\_x = 0.0f;

GLfloat dir\_y = 0.0f;

GLfloat dir\_z = 0.0f;

void Draw\_Leg();

void Draw\_Triangle() // This function draws a triangle with RGB colors

{

GLfloat mat\_specular[] = { 0.6f, 0.6f, 0.6f, 1.0f };

GLfloat mat\_diffuse0[] = { 0.85f, 0.65f, 0.2f, 1.0f };

GLfloat mat\_diffuse1[] = { 1.0f, 0.0f, 0.0f };

GLfloat mat\_diffuse2[] = { 0.0f, 1.0f, 0.0f };

GLfloat mat\_diffuse3[] = { 1.0f, 1.0f, 0.0f };

GLfloat mat\_diffuse4[] = { 0.0f, 1.0f, 1.0f };

GLfloat mat\_diffuse5[] = { 0.0f, 0.0f, 1.0f };

//画茶壶

glPushMatrix();

glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_specular);// 设置多边形正面的镜面反射属性

glMateriali(GL\_FRONT\_AND\_BACK, GL\_SHININESS, 50);// 指定镜面指数

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_diffuse0); //设置多边形正面漫反射属性

glTranslatef(0, 0, 4 + 1);

glRotatef(90, 1, 0, 0);

glutSolidTeapot(1);

glPopMatrix();

//画桌面

glPushMatrix();

glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_diffuse1);// 设置多边形正面的镜面反射属性

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_diffuse1);//设置多边形正面漫反射属性

glTranslatef(0, 0, 3.5);

glScalef(5, 4, 1);

glutSolidCube(1.0);

glPopMatrix();

//画四条腿

glPushMatrix();

glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_diffuse2);// 设置多边形正面的镜面反射属性

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_diffuse2);//设置多边形正面漫反射属性

glTranslatef(1.5, 1, 1.5);

Draw\_Leg();

glPopMatrix();

glPushMatrix();

glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_diffuse3);// 设置多边形正面的镜面反射属性

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_diffuse3);//设置多边形正面漫反射属性

glTranslatef(-1.5, 1, 1.5);

Draw\_Leg();

glPopMatrix();

glPushMatrix();

glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_diffuse4);// 设置多边形正面的镜面反射属性

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, mat\_diffuse4);//设置多边形正面漫反射属性

glTranslatef(1.5, -1, 1.5);

Draw\_Leg();

glPopMatrix();

glPushMatrix();

glMaterialfv(GL\_FRONT\_AND\_BACK, GL\_SPECULAR, mat\_diffuse5);// 设置多边形正面的镜面反射属性

glMaterialfv(GL\_FRONT\_AND\_BACK, GL\_DIFFUSE, mat\_diffuse5);//设置多边形正面漫反射属性

glTranslatef(-1.5, -1, 1.5);

Draw\_Leg();

glPopMatrix();

}

//绘制腿部

void Draw\_Leg()

{

glScalef(1, 1, 3);

glutSolidCube(1.0);

}

void updateView(int width, int height)

{

glViewport(0, 0, width, height);//设置视窗大小

glMatrixMode(GL\_PROJECTION);//设置矩阵模式为投影

glLoadIdentity(); //初始化矩阵为单位矩阵

float whRatio = (GLfloat)width / (GLfloat)height; //设置显示比例

if (bPersp) {

gluPerspective(45.0f, whRatio, 0.1f, 100.0f); //透视投影

//glFrustum(-3, 3, -3, 3, 3,100);

}

else {

glOrtho(-3, 3, -3, 3, -100, 100); //正投影

}

glMatrixMode(GL\_MODELVIEW); //设置矩阵模式为模型

}

void reshape(int width, int height)

{

if (height == 0) //如果高度为0

{

height = 1; //让高度为1（避免出现分母为0的现象）

}

wHeight = height;

wWidth = width;

updateView(wHeight, wWidth); //更新视角

}

void idle()

{

glutPostRedisplay(); //调用当前绘制函数

}

void key(unsigned char k, int x, int y)

{

switch (k)

{

case 27:

case 'q': {exit(0); break; }

case 'p': {bPersp = !bPersp; break; }

case ' ': {bAnim = !bAnim; break; }

case 'o': {bWire = !bWire; break; }

case 'a': { //整体左移

eye[0] += 0.2f;

center[0] += 0.2f;

break;

}

case 'd': {//整体右移

eye[0] -= 0.2f;

center[0] -= 0.2f;

break;

}

case 'w': {//整体上移

eye[1] -= 0.2f;

center[1] -= 0.2f;

break;

}

case 's': {//整体下移

eye[1] += 0.2f;

center[1] += 0.2f;

break;

}

case 'z': {//整体前移

eye[2] -= 0.2f;

center[2] -= 0.2f;

break;

}

case 'c': {//整体后移

eye[2] += 0.2f;

center[2] += 0.2f;

break;

}

case 'j': {//环境光左移

light\_x = light\_x - 0.2f;

break;

}

case 'l': {//环境光右移

light\_x = light\_x + 0.2f;

break;

}

case 'i': {//环境光上移

light\_y = light\_y + 0.2f;

break;

}

case 'k': {//环境光下移

light\_y = light\_y - 0.2f;

break;

}

case 'n': {//环境光前移

light\_z = light\_z + 0.2f;

break;

}

case 'm': {//环境光后移

light\_z = light\_z - 0.2f;

break;

}

case 'r': {//环境光颜色切换

isWhite = !isWhite;

break;

}

case 'f': {//聚光灯左移

dir\_x = dir\_x - 0.05f;

break;

}

case 'h': {//聚光灯右移

dir\_x = dir\_x + 0.05f;

break;

}

case 't': {//聚光灯上移

dir\_y = dir\_y - 0.05f;

break;

}

case 'g': {//聚光灯下移

dir\_y = dir\_y + 0.05f;

break;

}

case 'v': {//聚光灯后移

dir\_z = dir\_z - 0.05f;

break;

}

case 'b': {//聚光灯前移

dir\_z = dir\_z + 0.05f;

break;

}

case 'x': { //聚光灯角度变大

if (spotangle <= 89.0f) {

spotangle = spotangle + 0.2f;

}

break;

}

case 'y': { //聚光灯角度变小

if (spotangle >= 1.0f) {

spotangle = spotangle - 0.2f;

}

break;

}

}

updateView(wHeight, wWidth);

}

void redraw()

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT); //清除颜色缓存和深度缓存

glLoadIdentity(); //初始化矩阵为单位矩阵

gluLookAt(eye[0], eye[1], eye[2],

center[0], center[1], center[2],

0, 1, 0); // 场景（0，0，0）的视点中心 (0,5,50)，Y轴向上

if (bWire) {

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

//设置多边形绘制模式：正反面，线型

}

else {

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

//设置多边形绘制模式：正反面，填充

}

glEnable(GL\_DEPTH\_TEST);//开启深度测试

glEnable(GL\_LIGHTING); //开启光照模式

//GLfloat specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };

GLfloat light\_pos[] = { 5.0 + light\_x ,5.0 + light\_y,5.0 + light\_z,1 }; //定义环境光位置

GLfloat light\_pos1[] = { 0.0f, 5.0f, 0.0f, 1.0f }; //定义聚光灯位置

GLfloat lightDir[] = { 0.0f + dir\_x ,-1.0f + dir\_y ,0.0f + dir\_z }; //角度

GLfloat white[] = { 1.0f, 1.0f, 1.0f, 1.0f }; //定义白色

if (isWhite) {

color[0] = 1.0f, color[1] = 1.0f, color[2] = 1.0f, color[3] = 1.0f;

}

else {

color[0] = 0.0f, color[1] = 1.0f, color[2] = 0.0f, color[3] = 1.0f;

}

glLightfv(GL\_LIGHT0, GL\_POSITION, light\_pos); //设置第0号光源的光照位置

glLightfv(GL\_LIGHT0, GL\_SPECULAR, white); //设置镜面反射光照颜色

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, white); //设置漫射光成分

glLightfv(GL\_LIGHT0, GL\_AMBIENT, color); //设置第0号光源多次反射后的光照颜色（环境光颜色）

glEnable(GL\_LIGHT0); //开启第0号光源

glLightfv(GL\_LIGHT1, GL\_AMBIENT, color); //设置环境光成分

glLightfv(GL\_LIGHT1, GL\_SPECULAR, white); //设置镜面光成分

glLightfv(GL\_LIGHT1, GL\_DIFFUSE, white); //设置漫射光成分

glLightfv(GL\_LIGHT1, GL\_POSITION, light\_pos1);

glLightf(GL\_LIGHT1, GL\_SPOT\_CUTOFF, spotangle); //裁减角度

glLightfv(GL\_LIGHT1, GL\_SPOT\_DIRECTION, lightDir); //光源方向

glLightf(GL\_LIGHT1, GL\_SPOT\_EXPONENT, 2.); //聚集度

glEnable(GL\_LIGHT1);

glRotatef(fRotate, 0, 1.0f, 0); //旋转

glRotatef(-90, 1, 0, 0); //旋转

glScalef(0.2, 0.2, 0.2); //缩放

Draw\_Triangle(); //绘制场景

if (bAnim) fRotate += 0.5f; //旋转因子改变

glutSwapBuffers(); //交换缓冲区

}

int main(int argc, char\* argv[])

{

glutInit(&argc, argv);//对glut的初始化

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

//初始化显示模式:RGB颜色模型，深度测试，双缓冲

glutInitWindowSize(480, 480);//设置窗口大小

int windowHandle = glutCreateWindow("Simple GLUT App");//设置窗口标题

glutDisplayFunc(redraw); //注册绘制回调函数

glutReshapeFunc(reshape); //注册重绘回调函数

glutKeyboardFunc(key); //注册按键回调函数

glutIdleFunc(idle);//注册全局回调函数：空闲时调用

glutMainLoop(); // glut事件处理循环

return 0;

}