《计算机图形学实验》综合实验报告

题目 三维图形渲染

学 号 **20201120283**

姓 名 李志辉

指导教师 钱文华

日 期 **2022.6.19**

目录

三维图形渲染

摘要：图像渲染是将三维的光能传递处理转换为一个二维图像的过程。场景和实体用三维形式表示，更接近于现实世界，便于操纵和变换，而图形渲染又包括光照和纹理贴图等操作。OpenGL 库中包含了多种光照的类型。材质是用光反射率来表示的。其原理是基于人眼的原理，场景中的物体是由光的红绿蓝的分量以及材质的红绿蓝的反射率的乘积后所形成的颜色值。纹理指的是物体表面的花纹。OpenGL 库中也集合了对于物体纹理的映射处理方式，能够十分完整的复现物体表面的真实纹理。

关键词：OpenGL 光照 纹理 图形渲染

1.1 实验背景

在 OpenGL 中，任何事物都在 3D 空间中，而屏幕和窗口却是 2D 像素数组，这导致 OpenGL 的大部分工作都是关于把 3D 坐标转变为 2D 像素。3D 坐标转为 2D 坐标的处理过程是由 OpenGL 的图形渲染管线（Graphics Pipeline）管理的。

图形渲染管线可以被划分为两个主要部分：第一部分把 3D 坐标转换为 2D 坐标，第二部分是把 2D 坐标转变为实际的有颜色的像素。

图形渲染管线可以被划分为几个阶段，每个阶段将会把前一个阶段的输出作为输入。它们具有并行执行的特性，当今大多数显卡都有成千上万的小处理核心，它们在 GPU 上为每一个（渲染管线）阶段运行各自的小程序，从而在图形渲染管线中快速处理你的数据。

1.2 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。

2.1 开发工具

OpenGL，C/C++，Microsoft Visual Studio

2.2 基本模块介绍

2.2.1 光照

```
void init(void)
{
    //材质反光性设置
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };  //镜面反射参数
    GLfloat mat_shininess[] = { 50.0 };               //高光指数
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat white_light[] = { 2.0, 2.0, 2.0, 1.0 };   //灯位置(2,2,2), 最后 1-开关
    GLfloat Light_Model_Ambient[] = { 0.2, 0.2, 0.2, 1.0 }; //环境光参数

    glClearColor(0.0, 0.0, 0.0, 0.0);  //背景色
    glShadeModel(GL_SMOOTH);           //多变性填充模式

    //材质属性
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    //灯光设置
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);   //散射光属性
    glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);  //镜面反射光
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, Light_Model_Ambient);  //环境光参数

    glEnable(GL_LIGHTING);   //开关:使用光
    glEnable(GL_LIGHT0);     //打开 0#灯
    glEnable(GL_DEPTH_TEST); //打开深度测试
}
```

2.2.2 阴影

```
void draw_world() // 绘制一个地板{
    glm::vec4 v1(-3, 0, -3, 1), v2(-3, 0, 3, 1), v3(3, 0, 3, 1), v4(3, 0, -3, 1);//四个
顶点
    glm::mat4 m = glm::translate(glm::vec3(0.5f, 0.5f, 0.5f))
        * glm::scale(glm::vec3(0.5f, 0.5f, 0.5f)); // 需要将裁剪坐标的[-1,+1]缩放到[0,1]
    glm::vec4 t;
    glBegin(GL_POLYGON);
    glNormal3f(0, 1, 0);
    t = m * shadow_mat_p * shadow_mat_v * v1; // 按和生成纹理相同的变换计算纹理坐标
    glTexCoord4fv(&t[0]); glVertex3fv(&v1[0]);
    t = m * shadow_mat_p * shadow_mat_v * v2;
    glTexCoord4fv(&t[0]); glVertex3fv(&v2[0]);
```

```
    t = m * shadow_mat_p * shadow_mat_v * v3;
    glTexCoord4fv(&t[0]); glVertex3fv(&v3[0]);
    t = m * shadow_mat_p * shadow_mat_v * v4;
    glTexCoord4fv(&t[0]); glVertex3fv(&v4[0]);
    glEnd();
}
glm::mat4 shadow_mat_p; // 光源视角的投影矩阵
glm::mat4 shadow_mat_v; // 光源视角的视图矩阵

void tex_init() // 纹理初始化
{
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glGenTextures(1, &tex_shadow);
    glBindTexture(GL_TEXTURE_2D, tex_shadow);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE, GL_LUMINANCE);
}
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_R_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);

}
```

2.2.3 纹理

```
void Draw_Leg();
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
    filePtr = fopen(filename, "rb");
    if (filePtr == NULL) {
        printf("file not open\n");
        return NULL;
    }
        fread(&bitmapFileHeader, sizeof(BITMAPFILEHEADER), 1, filePtr);
    if (bitmapFileHeader.bfType != BITMAP_ID) {
        fprintf(stderr, "Error in LoadBitmapFile: the file is not a bitmap file\n");
        return NULL;
    }
```

```
    fread(bitmapInfoHeader, sizeof(BITMAPINFOHEADER), 1, filePtr);
    fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK_SET);
    bitmapImage = new unsigned char[bitmapInfoHeader->biSizeImage];
    if (!bitmapImage) {
        fprintf(stderr, "Error in LoadBitmapFile: memory error\n");
        return NULL;
    }
    fread(bitmapImage, 1, bitmapInfoHeader->biSizeImage, filePtr);
    if (bitmapImage == NULL) {
        fprintf(stderr, "Error in LoadBitmapFile: memory error\n");
        return NULL;
    }
    for (imageIdx = 0; imageIdx < bitmapInfoHeader->biSizeImage; imageIdx += 3) {
        tempRGB = bitmapImage[imageIdx];
        bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];
        bitmapImage[imageIdx + 2] = tempRGB;
    }
    fclose(filePtr);
    return bitmapImage;
}
void generateTex(){
    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;
        }
    }
}
```

3.1 关键算法理论

3.1.1 光照

运行 OpenGL 程序在屏幕上显示的最终颜色，受场景中光线的特性以及物体反射和吸收光的属性（即材质）影响。光线可能来自特定的位置与方向，也可能是散布在整个场景中（环境光）；而物体表面能够吸收，反射光线，有些物体本身还能够发射光线，物体的这些属性被称为材质。在 OpenGL 中，物体的材质属性通过反射不同方向的环境光，漫反射光，镜面光的 RGB 颜色来表示的。光照计算就是将发射光，泛射光，漫反射光以及镜面高光四个成分分别计算，然后再累加起来。

3.1.2 阴影

绘制阴影，需要用到深度纹理，即从光源角度看模型并绘制一张纹理图，纹理图的颜色代表了模型上的点离光源的深度，只有离光源较近的点才会绘制到深度纹理图中，被遮挡的点不会被绘制到深度纹理图中。判断地平面中的点是否处于阴影中，需要将此点变换到光源坐标系中，然后计算变换后的点离光源的深度，判断此深度是否大于纹理图中相应位置的深度，如果大于说明此点在阴影中。

3.1.3 纹理

在 OpenGL 中，我们通常将纹理中的像素将按照纹理坐标进行编址，纹理坐标系是一个空间直角坐标系，横轴为 S 轴，纵轴为 T 轴，垂直于屏幕的坐标轴为 R 轴。在我们的 2D 纹理中，由于没有 R 轴，我们也可以将横轴称为 U 纵轴称为 V 轴，也就是我们所说的 UV 坐标系。但和 OpenGL 坐标系所不同的是：纹理坐标系的（0,0）点位于纹理的左下角，而（1,1）点位于纹理的右上角。

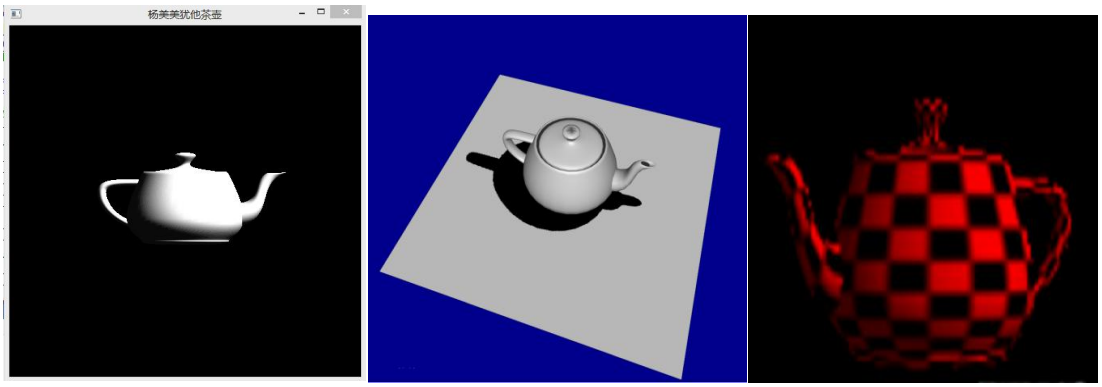
通过纹理坐标获取像素颜色信息的过程称为采样，而采样的结果会根据纹理参数设置的不同而千差万别。OpenGL 中设置纹理参数的 API 接口为 glTextureParameter，我们所有的纹理参数都由这个接口设置，下面我们介绍几种常用的纹理参数的配置。

3.2 程序实现步骤

在本次大作业过程中，我按照如下顺序完成了三维图形渲染：

4、实验结果

4.1 光照　　4.2 阴影　　4.3 纹理



5、心得体会

经过一学期计算机图形学课程的学习，我初次了解到计算机图形学的相关内容，也对 OpenGL 的一些简单操作有了了解。在本学期的课程中，我学到了一些经典的算法，比如 Bresenham 算法、Cohen-Sutherland 裁剪算法以及 Z-buffer 消隐算法等等，同时，也学会了二维观察、三维观察的一般过程和二维三维图形几何变换，这为本次的期末大作业奠定了基础。

在本次期末大作业中，我以茶壶为载体，实现了光照、阴影和纹理贴图等图形渲染操作。在此过程中，我要从零开始研究这些操作的实现原理以及相关算法，我发现对于相同的问题，每个人的做法都不大一致，在这些做法中，我取百家之所长，再加入一些自己的理解，真正的学会了这一项技术。在网上了解到，这些操作只是图形渲染中最基本的一些操作，其他一些更难的操作需要在今后的学习生涯中逐渐了解并掌握。不能否认的是，经过本次大作业的完成，我的进步是十分巨大的，眼界也更加开阔。

在代码实现过程中，我也不可避免地遇到了一些困难，比如程序无法正常运行、程序缺少必要的头文件等问题，经过与同学们的探讨以及在网上查找解决方案，都可以勉强解决并顺利运行程序，从而达到想实现的效果。

这种遇到难题解决难题的过程虽然复杂曲折，但在真正解决了之后，我也确实有不少的收获。比如在实验中经常出现的 C4996 fopen：This function or variable may be unsafe 以及 c++未定义标识符等报错，在今后的编程生涯中一定会频繁出现，而通过本次的综合大作业，这些问题对我来说已经了然于心。因

此，这种解决问题的能力对于我今后来说十分重要。

经过本次实验，我的进步不仅仅在于计算机图形学的相关知识，更重要的是自主学习以及解决问题的能力，希望在今后的编程生涯中可以继续努力，不断进步。

6、参考文献

[1] Donald Hearn、M. Pauline Baker、Warren R. Carithers 著，蔡士杰，杨若瑜 译《计算机图形学第四版》。

[2] 百度百科——关于三维图形渲染

[3] CSDN——阴影、光照、纹理等操作原理及算法

7、附录

光照源代码

```
#include <GL/glew.h>
#include <GL/freeglut.h>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#pragma comment(lib,"glew32.lib")
using namespace std;
GLfloat lightPosition[3] = { 30.0,30.0,30.0 };
GLfloat ambient[4] = { 0.0 , 0.0 , 1.0 , 1.0 };
GLfloat lightcolor[4] = { 1.0 , 1.0 , 1.0 , 1.0 };
GLfloat eyeposition[3] = { 0.0 , 10.0 , 30.0 };
GLfloat Ns = 30;
GLfloat attenuation = 0.01;
GLfloat objectSize = 15.0;
GLuint programHandle;
GLuint vShader, fShader;
char* textFileRead(const char* fn){
    FILE* fp;
    char* content = NULL;
    int count = 0;
    if (fn != NULL){
        fp = fopen(fn, "rt");
        if (fp != NULL){
            fseek(fp, 0, SEEK_END);
            count = ftell(fp);
```

```
            rewind(fp);
            if (count > 0){
                content = (char*)malloc(sizeof(char) * (count + 1));
                count = fread(content, sizeof(char), count, fp);
                content[count] = '\0';
            }
            fclose(fp);
        }
    }
    return content;
}
void initShader(const char* VShaderFile, const char* FShaderFile){
    vShader = glCreateShader(GL_VERTEX_SHADER);
    if (0 == vShader){
        cerr << "ERROR : Create vertex shader failed" << endl;
        exit(1);
    }
    const GLchar* vShaderCode = textFileRead(VShaderFile);
    const GLchar* vCodeArray[1] = { vShaderCode };
    glShaderSource(vShader, 1, vCodeArray, NULL);
    glCompileShader(vShader);
    GLint compileResult;
    glGetShaderiv(vShader, GL_COMPILE_STATUS, &compileResult);
    if (GL_FALSE == compileResult){
        GLint logLen;
        glGetShaderiv(vShader, GL_INFO_LOG_LENGTH, &logLen);
        if (logLen > 0)
        {
            char* log = (char*)malloc(logLen);
            GLsizei written;
            glGetShaderInfoLog(vShader, logLen, &written, log);
            cerr << "vertex shader compile log : " << endl;
            cerr << log << endl;
            free(log);//释放空间
        }
    }
    fShader = glCreateShader(GL_FRAGMENT_SHADER);
    if (0 == fShader){
        cerr << "ERROR : Create fragment shader failed" << endl;
        exit(1);
    }
    const GLchar* fShaderCode = textFileRead(FShaderFile);
    const GLchar* fCodeArray[1] = { fShaderCode };
    glShaderSource(fShader, 1, fCodeArray, NULL);
```

```
    glCompileShader(fShader);
    glGetShaderiv(fShader, GL_COMPILE_STATUS, &compileResult);
    if (GL_FALSE == compileResult){
        GLint logLen;
        if (logLen > 0){
            char* log = (char*)malloc(logLen);
            GLsizei written;
            glGetShaderInfoLog(fShader, logLen, &written, log);
            cerr << "fragment shader compile log : " << endl;
            cerr << log << endl;
            free(log);//释放空间
        }
    }
    if (GL_FALSE == linkStatus){
        cerr << "ERROR : link shader program failed" << endl;
        GLint logLen;
        glGetProgramiv(programHandle, GL_INFO_LOG_LENGTH,
            &logLen);
        if (logLen > 0){
            char* log = (char*)malloc(logLen);
            GLsizei written;
            glGetProgramInfoLog(programHandle, logLen,
                &written, log);
            cerr << "Program log : " << endl;
            cerr << log << endl;
        }
    }
}
void init(){
    GLenum err = glewInit();
    if (GLEW_OK != err){
        cout << "Error initializing GLEW: " << glewGetErrorString(err) << endl;
    }
    glEnable(GL_DEPTH_TEST);
    initShader("VertexShader.txt", "FragmentShader.txt");
    glClearColor(0.0, 0.0, 0.0, 0.0);
}
void Reshape(int w, int h){
    glViewport(0, 0, (GLsizei)w, (GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(90, 1, 0.1, 1000.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
```

```
    gluLookAt(eyeposition[0], eyeposition[1], eyeposition[2], 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
void display(){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glUseProgram(programHandle);
    glUniform1f(glGetUniformLocation(programHandle, "Ns"), Ns);
    glUniform1f(glGetUniformLocation(programHandle, "attenuation"), attenuation);
    glutSolidTeapot(objectSize);
    //glutSolidSphere(objectSize-3, 100, 100);
    glutSwapBuffers();
}
void SpecialKey(GLint key, GLint x, GLint y){
    if (key == GLUT_KEY_UP){
        //do something
    }
    display();
}
int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("李志辉 20201120283");
    init();
    glutReshapeFunc(Reshape);
    glutDisplayFunc(display);
    glutSpecialFunc(SpecialKey);
    glutMainLoop();
    return 0;
}
```

阴影源代码

```
#include <stdlib.h>
#include "glut.h"
float fRotate = 0.0f; //旋转因子（茶壶和桌子）
float fScale = 1.0f; //缩放因子
float tRotate = 0.0f; //旋转因子（茶壶）
bool bPersp = false; //是否为透视投影 （vs 正投影）
bool bAnim = false; // 茶壶和桌子是否旋转
bool bWire = false; // 绘制模式是否为线形 （vs 填充）
bool isRotate = false; //茶壶是否旋转
int wHeight = 0;
```

```
int wWidth = 0;
void Draw_Leg(){
    glScalef(1, 1, 3);
    glutSolidCube(1.0);
}
void Draw_Scene(float place[]){
    glutSolidTeapot(1);
    glPopMatrix();
    t = m * shadow_mat_p * shadow_mat_v * v1; // 按和生成纹理相同的变换计算纹理坐标
    glTexCoord4fv(&t[0]); glVertex3fv(&v1[0]);
    t = m * shadow_mat_p * shadow_mat_v * v2;
    glTexCoord4fv(&t[0]); glVertex3fv(&v2[0]);
    t = m * shadow_mat_p * shadow_mat_v * v3;
    glTexCoord4fv(&t[0]); glVertex3fv(&v3[0]);
    t = m * shadow_mat_p * shadow_mat_v * v4;
    glTexCoord4fv(&t[0]); glVertex3fv(&v4[0]);
    glEnd();
}
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, whRatio, 1, 100); //透视投影
    }
    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）
    }
    height = width = min(height, width);
    wHeight = height;
    wWidth = width;
    updateView(wHeight, wWidth); //更新视角
}
void idle(){
    glutPostRedisplay();//调用当前绘制函数
}
float eye[] = { 0, 0, 8 };
float center[] = { 0, 0, 0 };
```

```
float place[] = { 0, 0, 5 };
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);
    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
    }
}
int main(int argc, char* argv[])
{
    glutInit(&argc, argv);//对 glut 的初始化
    glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
    glutInitWindowSize(480, 480);//设置窗口大小
    int windowHandle = glutCreateWindow("Ex 3");//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;
}
```

纹理源代码

```
#include <stdlib.h>
#include <windows.h>
#include <stdio.h>
#include "glut.h"
#define BITMAP_ID 0x4D42
#define Height 16
#define Width 16
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();
unsigned char* LoadBitmapFile(char* filename, BITMAPINFOHEADER* bitmapInfoHeader){
    FILE* filePtr;       // 文件指针
    unsigned char * bitmapImage;
    Int imageIdx = 0;              // 图像位置索引
    unsigned char tempRGB;       // 交换变量
    filePtr = fopen(filename, "rb");
    if (filePtr == NULL) {
        printf("file not open\n");
        return NULL;
    }
    if (!bitmapImage) {
        fprintf(stderr, "Error in LoadBitmapFile: memory error\n");
        return NULL;
    }
    if (bitmapImage == NULL) {
        fprintf(stderr, "Error in LoadBitmapFile: memory error\n");
        return NULL;
    for (imageIdx = 0; imageIdx < bitmapInfoHeader->biSizeImage; imageIdx += 3) {
        tempRGB = bitmapImage[imageIdx];
        bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];
        bitmapImage[imageIdx + 2] = tempRGB;
    }
    fclose(filePtr);
    return bitmapImage;
}
void generateTex(){
    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;
        }
    }
}
void init(){
    glGenTextures(3, texture);
    texload(1, "Crack.bmp");
    generateTex();
    glBindTexture(GL_TEXTURE_2D, texture[2]);
```

```
}
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;
    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} };
    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]);
    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]

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
    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){
        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();
}
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
        if (bWire) {
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    }
    else {
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    }
int main(int argc, char* argv[]){
    glutInit(&argc, argv);//对 glut 的初始化
    glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
    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;
}
```