浙江大学实验报告

课程名称：____计算机图形学___ 指导老师：_____成绩：_____
实验名称：____OpenGL 纹理_____ 实验类型：_____基础实验_____同组学生姓名：_____

装

订

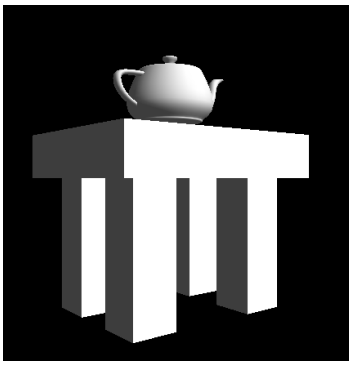
线

一、实验目的和要求

在 OpenGL 消隐和光照实验的基础上，通过实现实验内容，掌握 OpenGL 中纹理的使用，并验证课程中关于纹理的内容。

二、实验内容和原理

使用 Visual Studio C++编译已有项目工程。



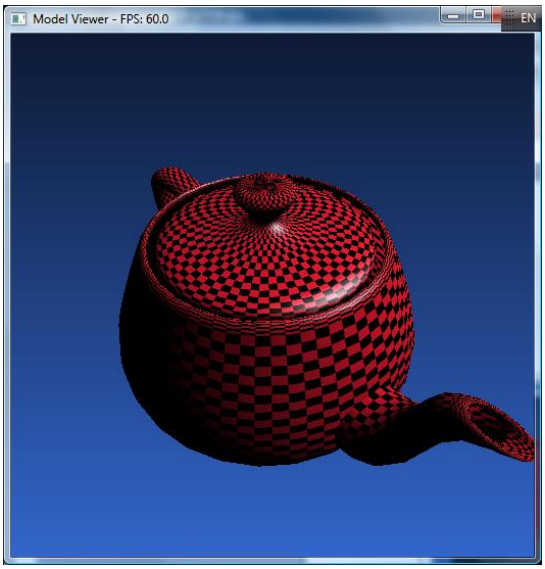
模型尺寸不做具体要求。要求修改代码达到以下要求：

1. 通过设置纹理，使得茶壶纹理为：

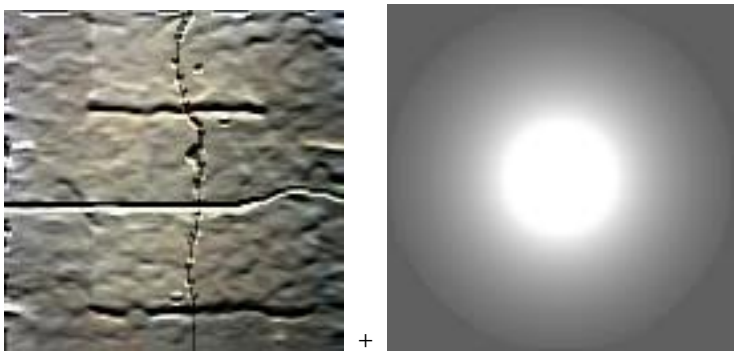
2. 使得桌子纹理为：

3. 对茶壶实现纹理和光照效果的混合

4. 自己用代码产生一张纹理，并贴在茶壶表面，效果类似：



5. 在桌面上实现两张纹理的叠合效果(附加项，完成可加分)：



提示：glSolidCube()并不会为多边形指定纹理坐标，因此需要自己重写一个有纹理坐标的方块函数。另外，此实验需要用到 #include <windows.h>

三、主要仪器设备

Visual Studio C++
glut.zip
模板工程

四、实验原理及过程分析

1.读取纹理图片

BMP 文件的数据按照从文件头开始的先后顺序分为四个部分：bmp 文件头(bmp file header)：提供文件的格式、大小等信息；位图信息头(bitmap information)：提供图像数据的尺寸、位平面数、压缩方式、颜色索引等信息；调色板(color palette)：索引与其对应的颜色的映射表；位图数据(bitmap data)：图像数据。我们将位图的前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 形式储存。

```
typedef struct tagBITMAPFILEHEADER {
    WORD        bfType;
    DWORD bfSize;
    WORD        bfReserved1;
    WORD        bfReserved2;
    DWORD bfOffBits;
} BITMAPFILEHEADER;
```

bmp 文件头

```
typedef struct tagBITMAPINFOHEADER {
    DWORD  biSize;
    LONG   biWidth;
    LONG   biHeight;
    WORD   biPlanes;
    WORD   biBitCount;
    DWORD  biCompression;
    DWORD  biSizeImage;
    LONG   biXPelsPerMeter;
    LONG   biYPelsPerMeter;
    DWORD  biClrUsed;
    DWORD  biClrImportant;
} BITMAPINFOHEADER;
```

位图信息图

在读取纹理图片时，需要用到 fopen_s 函数，fopen_s 函数比 fopen 函数多了溢出检测，更安全一些。使用方法如下：

```
errno_t err;
err = fopen_s(&filePtr, filename, "rb");
```

读取纹理图片的代码如下：

```
// 读纹理图片
unsigned char *loadBitmapFile(const 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
    // 需要用到 fopen_s
    errno_t err;
    err = fopen_s(&filePtr, 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
```

```
51.     fclose(filePtr);
52.     return bitmapImage;
53. }
```

2.加载纹理及纹理混合

读入图片后我们用 bitmapData 指针指向读入图片的信息，开始纹理加载，此处分为两种情况，其中一种情况为显示单独的图片，另一种情况为显示纹理混合的图片。如果需要显示纹理混合的图片则需要将两张图片的像素进行简单的运算以达到混合的目的，如果只需要加载单独的图片，则直接用 bitmapData 读入相应图片的信息即可。

在进行加载纹理的操作时，我们用到了 void glBindTexture(GLenum target, Gluint texture)函数，void glTexParameteri(Glenum target,Glenum pname,Glint param) 函数和 void glTexImage2D(Glenum target, Glint level, Glint internalformat, Glsizei width, Glsizei height, Glint border, Glenum format, Glenum type, const Glvoid * pixels)函数。

其中 glBindTexture()函数用来将一个命名的纹理绑定到一个纹理目标上，其中参数 target 指明了纹理要绑定到的目标，texture 指明了一张纹理的名字。这里我们用到的是 glBindTexture(GL_TEXTURE_2D, texture[i])，即这张纹理被绑定成为一张二维纹理。

glTexParameteri（）函数用来指定指定当前纹理的放大/缩小过滤方式，其中参数 target 用来指定目标纹理，pname 用来指定单值纹理参数的符号名称，param 用来指定 pname 的值，这里我们用到的是 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST) 和 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST); 其中 GL_TEXTURE_MAG_FILTER 代表放大过滤，即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，GL_TEXTURE_MIN_FILTER 代表缩小过滤，即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。GL_NEAREST 表示取 4 个坐标上最接近待映射纹理像素的颜色。

glTexImage2D（）函数用来定义材质图片，纹理映射一个指定的纹理图像的每一部分到相应的图元中。调用带 GL_TEXTURE_2D 参数的 glEnable 和 glDisable 函数来启用和禁止二维材质贴图，默认中二维材质贴图是禁用的。其中参数 target 指定目标纹理，这个值必须是 GL_TEXTURE_2D；level 表示执行细节级别。0 是最基本的图像级别，1 表示第 N 级贴图细化级别。Internalformat 指定纹理中的颜色组件，这个取值和后面的 format 取值必须相同；width 表示纹理图像的宽度，必须是 2 的 n 次方；纹理图片至少要支持 64 个材质元素的宽度；height 表示纹理图像的高度，必须是 2 的 m 次方；纹理图片至少要支持 64 个材质元素的高度；border 指定边框的宽度，必须为 0；Format 表示像素数据的颜色格式，必须和 internalformat 取值必须相同；type 指定像素数据的数据类型。 Pixels 指定内存中指向图像数据的指针。

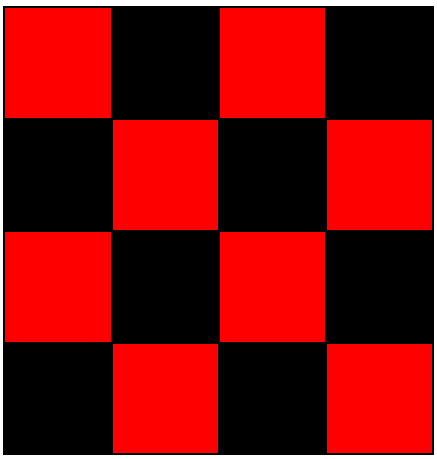
加载纹理的代码如下：

```
1.  // 加载纹理的函数
2.  void texLoad(int I, const char * filename)
3.  {
4.      unsigned char* bitmapData;// 纹理数据
5.      BITMAPINFOHEADER bitmapInfoHeader;// bitmap 信息头
6.
7.      if (filename == “Spot.bmp”) {
8.          BITMAPINFOHEADER bitmapInfoHeader1;// bitmap 信息头
9.          unsigned char* bitmapData1;
10.         unsigned char* bitmapData2;
```

```
        int imageIdx = 0;
        bitmapData1 = loadBitmapFile(filename, &bitmapInfoHeader);
        bitmapData2 = loadBitmapFile("Crack.bmp", &bitmapInfoHeader1);

        bitmapData = new unsigned char[bitmapInfoHeader.biSizeImage];
        for (imageIdx = 0; imageIdx < bitmapInfoHeader.biSizeImage; imageIdx++) {
            bitmapData[imageIdx] = bitmapData1[imageIdx] * 0.5 + bitmapData2[imageIdx] * 0.5;
        }
    }
    else {
        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
}
```

3.自定义贴图

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，设置长宽均为 16 个像素点（16*16），其中分为 4*4 的小块，一共 16 块如图所示：



根据行数和列数赋予不同的像素值，即可得到自定义贴图，所用方法即为判断当前位置是否处于红色格子位置区间从而对其赋值。其中这里用到的 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)和 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)表示制定纹理图片的重复，前者表示在 S 方向重复，后者表示在 T 方向重复。

与此有关的代码为：

```
// 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;
        if ((i < 4 || (i >= 8 && i < 12)) && (j < 4 || (j >= 8 && j < 12)) || (i >= 4 && i < 8 || (i >= 12 && i < 16)) && (j >= 4 && j < 8 || (j >= 12 && j < 16))) x = 0;
        else x = 255;
        image[i][j][0] = (GLubyte)x;
        image[i][j][1] = 0;
        image[i][j][2] = 0;
    }
}
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
```

4.定义纹理。

这里我们用到了 void glGenTextures(GLsizei n,GLuint * textures)函数，产生 n 个纹理 ID 存储在 textures 数组中。其中 n 指定要生成的纹理 ID 的数量，textures 指定存储生成的纹理 ID 的数组。这里我们用到的是 glGenTextures(4, texture)，分别将茶壶的两个纹理和桌子的两个纹理存储在数组中。

与此有关的代码如下：

```
// 定义纹理的函数
void initTexture()
{
    glGenTextures(4, 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

```

```
    // 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;
            if ((i < 4 || (i >= 8 && i < 12)) && (j < 4 || (j >= 8 && j < 12)) || (i >= 4 && i < 8 || (i >= 12 && i < 16)) && (j >= 4 && j < 8 || (j >= 12 && j < 16))) x = 0;
            else x = 255;
            image[i][j][0] = (GLubyte)x;
            image[i][j][1] = 0;
            image[i][j][2] = 0;
        }
    }
    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
}
```

5.纹理贴图

当我们加载好纹理后即可以将物体和纹理结合，将绘制物体的函数放在代码中间即可：

```
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texture[status]); //选择纹理 texture[status]

glPushMatrix();

    ……// 绘制物体的函数

glPopMatrix();

glDisable(GL_TEXTURE_2D);//关闭纹理 texture[status]
```

其中 glBindTexture(GL_TEXTURE_2D, texture[status])函数和 glDisable(GL_TEXTURE_2D 的功能分别为选择纹理 texture[status]和关闭纹理 texture[status]。

与此有关的代码为：

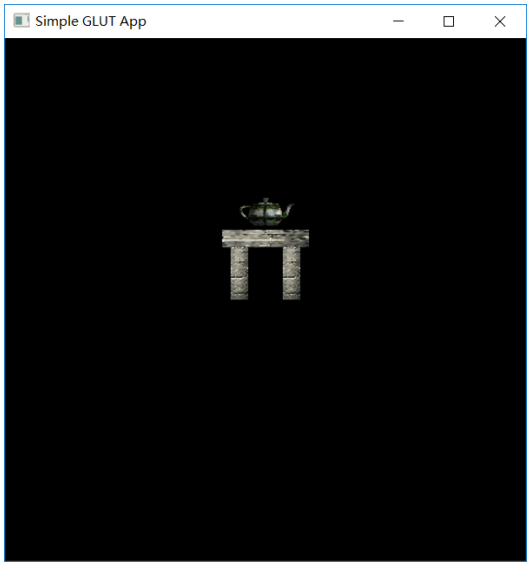
```
void drawTable() // 桌子
{
```

```
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, texture[teapot_tex]);//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[table_tex]);//选择纹理 texture[status]
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);//设置纹理不受光照影响

    ……//绘制桌子和四条腿

    glDisable(GL_TEXTURE_2D); //关闭纹理 texture[status2]
}
```

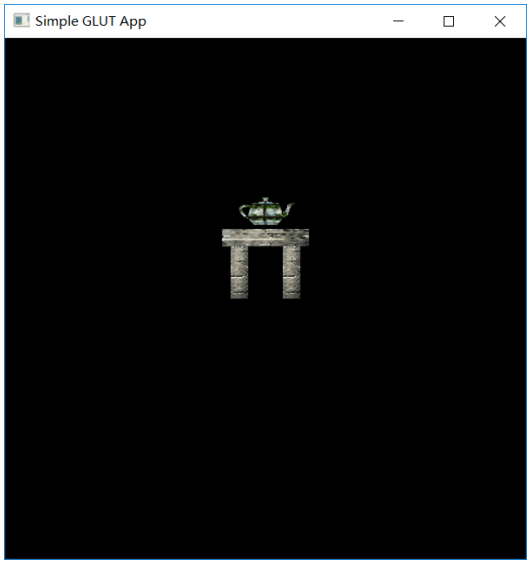
6.光照

在这里我们用glTexEnvf (GLenum target, GLenum pname, GLfloat param)函数指定纹理贴图和材质混合的方式，从而产生特定的绘制效果。我们用的是glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE)设置纹理受光照影响，保证纹理和光照混合，用于茶壶；glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_DECAL)设置纹理不受光照影响，单纯绘制纹理而不考虑光照，用于桌子。

对于茶壶来说，纹理受光照影响和纹理不受光照影响的对比图如下



纹理受光照影响



纹理不受光照影响

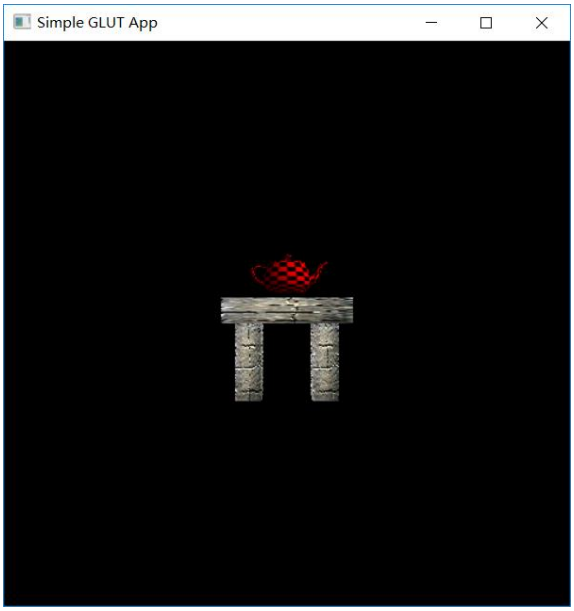
可以看到，当茶壶纹理受光照影响时，明暗变化是比较明显的，不受光照影响时，茶壶表面亮度一致。

7.按键控制

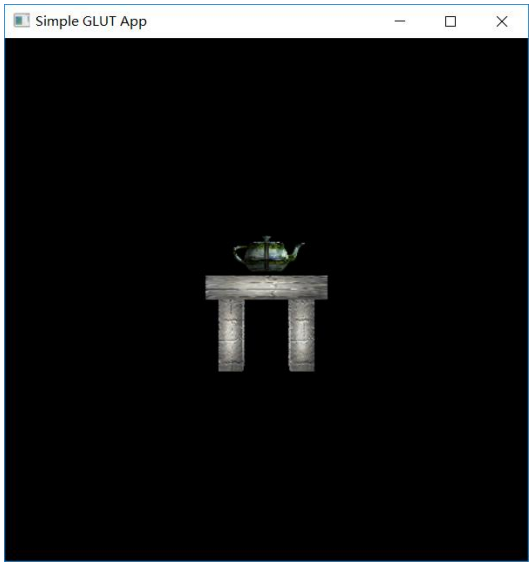
Q：退出
P：切换投影方式（正投影与透视投影）
空格键：启动与暂停旋转（桌子与茶壶一起绕桌子中心轴旋转）
O 切换渲染方式（填充模式与线框模式）
WASDZC：控制相机上下左右前后移动
T：改变茶壶纹理
R：改变桌子纹理（纹理混合与单独纹理）

五、实验结果与分析

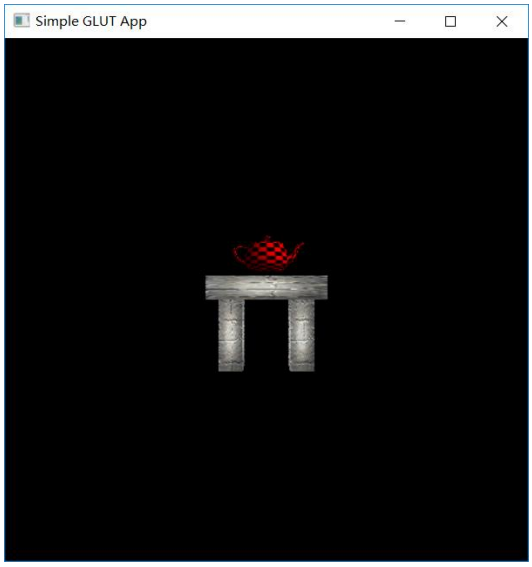
原始贴图



茶壶纹理改变



桌子纹理改变（混合纹理）



茶壶纹理改变，桌子纹理混合、

在本次实验中我掌握了 OpenGL 中纹理的使用，本次实验大致可以分为读纹理图片、加载纹理及纹理混合、自定义贴图、纹理贴图这几个部分，其中读取纹理图片的部分参考资料中给的出 fopen 函数并不能通过编译，我上网搜寻了 fopen_s 函数的使用方法解决了这个问题。通过加载纹理部分掌握了 glBindTexture()函数， glTexParameteri()函数和 glTexImage2D()函数的使用方法。在需要纹理混合时，将两张图片的像素进行简单的运算以达到混合的目的。自定义贴图部分用较短的代码实现了红黑格子样式的贴图，所用方法即为判断当前位置是否处于红色格子位置区间从而对其赋值。在纹理贴图部分，通过 glBindTexture()函数和 glDisable()函数控制纹理的选择和关闭。通过变换 teapot_tex 和 table_tex 的值来执行茶壶和桌子纹理的变换。

通过这次实验，我掌握了纹理贴图的方法，也完成了附加的纹理混合操作，收获很多。

六、 源代码

```
#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
GLuint texture[4];

float fTranslate;
float fRotate;
float fScale = 1.0f;
bool bPersp = false;
bool bAnim = false;
bool bWire = false;

int wHeight = 0;
int wWidth = 0;

int teapot_tex = 0;
int table_tex = 1;

// 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(const char *filename, BITMAPINFOHEADER *bitmapInfoHeader)
{
```

```
33.     FILE *filePtr;// 文件指针
34.     BITMAPFILEHEADER bitmapFileHeader;// bitmap 文件头
35.     unsigned char* bitmapImage;// bitmap 图像数据
36.     int imageIdx = 0;// 图像位置索引
37.     unsigned char tempRGB;// 交换变量
38.
39.     // 以“二进制+读”模式打开文件 filename
40.     // 需要用到 fopen_s
41.     errno_t err;
42.     err = fopen_s(&filePtr, filename, "rb");
43.     if (filePtr == NULL) {
44.         printf("file not open\n");
45.         return NULL;
46.     }
47.     // 读入 bitmap 文件图
48.     fread(&bitmapFileHeader, sizeof(BITMAPFILEHEADER), 1, filePtr);
49.     // 验证是否为 bitmap 文件
50.     if (bitmapFileHeader.bfType != BITMAP_ID) {
51.         fprintf(stderr, "Error in LoadBitmapFile: the file is not a bitmap file\n");
52.         return NULL;
53.     }
54.     // 读入 bitmap 信息头
55.     fread(bitmapInfoHeader, sizeof(BITMAPINFOHEADER), 1, filePtr);
56.     // 将文件指针移至 bitmap 数据
57.     fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK_SET);
58.     // 为装载图像数据创建足够的内存
59.     bitmapImage = new unsigned char[bitmapInfoHeader->biSizeImage];
60.     // 验证内存是否创建成功
61.     if (!bitmapImage) {
62.         fprintf(stderr, "Error in LoadBitmapFile: memory error\n");
63.         return NULL;
64.     }
65.
66.     // 读入 bitmap 图像数据
67.     fread(bitmapImage, 1, bitmapInfoHeader->biSizeImage, filePtr);
68.     // 确认读入成功
69.     if (bitmapImage == NULL) {
70.         fprintf(stderr, "Error in LoadBitmapFile: memory error\n");
71.         return NULL;
72.     }
73.     // 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
74.     for (imageIdx = 0; imageIdx < bitmapInfoHeader->biSizeImage; imageIdx += 3) {
75.         tempRGB = bitmapImage[imageIdx];
76.         bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];
```

```
77.         bitmapImage[imageIdx + 2] = tempRGB;
78.     }
79.     // 关闭 bitmap 图像文件
80.     fclose(filePtr);
81.     return bitmapImage;
82. }
83.
84. // 加载纹理的函数
85. void texLoad(int i, const char * filename)
86. {
87.     unsigned char* bitmapData;// 纹理数据
88.     BITMAPINFOHEADER bitmapInfoHeader;// bitmap 信息头
89.
90.     if (filename == "Spot.bmp") {
91.         BITMAPINFOHEADER bitmapInfoHeader1;// bitmap 信息头
92.         unsigned char* bitmapData1;
93.         unsigned char* bitmapData2;
94.         int imageIdx = 0;
95.         bitmapData1 = loadBitmapFile(filename, &bitmapInfoHeader);
96.         bitmapData2 = loadBitmapFile("Crack.bmp", &bitmapInfoHeader1);
97.
98.         bitmapData = new unsigned char[bitmapInfoHeader.biSizeImage];
99.         for (imageIdx = 0; imageIdx < bitmapInfoHeader.biSizeImage; imageIdx++) {
100.             bitmapData[imageIdx] = bitmapData1[imageIdx] * 0.5 + bitmapData2[imageIdx] *
    0.5;
101.         }
102.     }
103.     else {
104.         bitmapData = loadBitmapFile(filename, &bitmapInfoHeader);
105.     }
106.
107.     glBindTexture(GL_TEXTURE_2D, texture[i]);
108.     // 指定当前纹理的放大/缩小过滤方式
109.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
110.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
111.
112.     glTexImage2D(GL_TEXTURE_2D,
113.         0,//mipmap 层次(通常为，表示最上层)
114.         GL_RGB,//我们希望该纹理有红、绿、蓝数据
115.         bitmapInfoHeader.biWidth, //纹理宽带，必须是 n，若有边框+2
116.         bitmapInfoHeader.biHeight, //纹理高度，必须是 n，若有边框+2
117.         0, //边框(0=无边框, 1=有边框)
118.         GL_RGB,//bitmap 数据的格式
119.         GL_UNSIGNED_BYTE, //每个颜色数据的类型
```

```
120.         bitmapData);//bitmap 数据指针
121. }
122.
123.
124. // 定义纹理的函数
125. void initTexture()
126. {
127.     glGenTextures(4, 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
128.     texLoad(0, "Monet.bmp");
129.     texLoad(1, "Crack.bmp");
130.     texLoad(3, "Spot.bmp");
131.
132.     // 下面生成自定义纹理
133.
134.     // 生成红黑相间的图像
135.     for (int i = 0; i < Height; i++) {
136.         for (int j = 0; j < Width; j++) {
137.             int x;
138.             if ((i < 4 || (i >= 8 && i < 12)) && (j < 4 || (j >= 8 && j < 12)) || (i >=
     4 && i < 8 || (i >= 12 && i < 16)) && (j >= 4 && j < 8 || (j >= 12 && j < 16))) x = 0;
139.             else x = 255;
140.             image[i][j][0] = (GLubyte)x;
141.             image[i][j][1] = 0;
142.             image[i][j][2] = 0;
143.         }
144.     }
145.     glBindTexture(GL_TEXTURE_2D, texture[2]);
146.     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.
147.     glTexImage2D(GL_TEXTURE_2D, 0, 3, Width, Height, 0, GL_RGB, GL_UNSIGNED_BYTE, image)
     ;
148.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);//放大过滤,线性过滤
149.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);//缩小过滤,线性过滤
150.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);//S 方向重复
151.     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);//T 方向重复
152. }
153.
154.
155. void drawCube() {
156.     GLfloat vertex[8][3] = { // 长方体的八个顶点
157.         { 0.5, 0.5, 0.5 },
158.         { 0.5, 0.5, -0.5 },
159.         { -0.5, 0.5, -0.5 },
```

```
        { -0.5, 0.5, 0.5 },
        { 0.5, -0.5, 0.5 },
        { 0.5, -0.5, -0.5 },
        { -0.5, -0.5, -0.5 },
        { -0.5, -0.5, 0.5 }
    };

    GLint flat[6][4] = { // 长方体六个面的四个顶点
        { 0, 1, 2, 3 },
        { 4, 5, 6, 7 },
        { 0, 1, 5, 4 },
        { 2, 3, 7, 6 },
        { 3, 0, 4, 7 },
        { 1, 2, 6, 5 }
    };

    GLint textcoor[4][2] = {
        { 1, 1 },
        { 1, 0 },
        { 0, 0 },
        { 0, 1 }
    };

    glBegin(GL_QUADS); // 设置正方形绘制模式
    int i, j;
    for (i = 0; i < 6; i++) {
        for (j = 0; j < 4; j++) {
            glTexCoord2iv(textcoor[j]);
            glVertex3fv(vertex[flat[i][j]]);
        }
    }
    glEnd();
}

void drawLeg()
{
    glScalef(1, 1, 3);
    drawCube();
}

void drawTable() // 桌子
{
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, texture[teapot_tex]);//选择纹理 texture[status]
```

```
204.
205.     glPushMatrix();
206.     glTranslatef(0, 0, 4 + 1);
207.     glRotatef(90, 1, 0, 0);
208.     glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);//设置纹理受光照影响
209.     glutSolidTeapot(1);
210.     glPopMatrix();
211.
212.     glDisable(GL_TEXTURE_2D);//关闭纹理 texture[status]
213.
214.     glEnable(GL_TEXTURE_2D);
215.     glBindTexture(GL_TEXTURE_2D, texture[table_tex]);//选择纹理 texture[status]
216.     glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);//设置纹理不受光照影响
217.
218.     glPushMatrix();
219.     glTranslatef(0, 0, 3.5);
220.     glScalef(5, 4, 1);
221.     drawCube();
222.     glPopMatrix();
223.
224.     glPushMatrix();
225.     glTranslatef(1.5, 1, 1.5);
226.     drawLeg();
227.     glPopMatrix();
228.
229.     glPushMatrix();
230.     glTranslatef(-1.5, 1, 1.5);
231.     drawLeg();
232.     glPopMatrix();
233.
234.     glPushMatrix();
235.     glTranslatef(1.5, -1, 1.5);
236.     drawLeg();
237.     glPopMatrix();
238.
239.     glPushMatrix();
240.     glTranslatef(-1.5, -1, 1.5);
241.     drawLeg();
242.     glPopMatrix();
243.
244.     glDisable(GL_TEXTURE_2D); //关闭纹理 texture[status2]
245. }
246.
247. 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)//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; }

```

```
292.     case ' ': {bAnim = !bAnim; break; }
293.     case 'o': {bWire = !bWire; break; }
294.
295.     case 'a': { // 左移
296.         eye[0] += 0.2f;
297.         center[0] += 0.2f;
298.         break;
299.     }
300.     case 'd': { // 右移
301.         eye[0] -= 0.2f;
302.         center[0] -= 0.2f;
303.         break;
304.     }
305.     case 'w': { // 上移
306.         eye[1] -= 0.2f;
307.         center[1] -= 0.2f;
308.         break;
309.     }
310.     case 's': { // 下移
311.         eye[1] += 0.2f;
312.         center[1] += 0.2f;
313.         break;
314.     }
315.     case 'z': { // 前移
316.         eye[2] -= 0.2f;
317.         center[2] -= 0.2f;
318.         break;
319.     }
320.     case 'c': { // 后移
321.         eye[2] += 0.2f;
322.         center[2] += 0.2f;
323.         break;
324.     }
325.     case 't': { // 改变茶壶纹理
326.         if (teapot_tex == 0) teapot_tex = 2;
327.         else if(teapot_tex == 2) teapot_tex = 0;
328.         break;
329.     }
330.     case 'r': { // 改变桌子纹理
331.         if (table_tex == 1) table_tex = 3;
332.         else if (table_tex == 3) table_tex = 1;
333.         break;
334.     }
335.     }
```

```
336.
337.     updateView(wHeight, wWidth);
338. }
339.
340.
341. void redraw()
342. {
343.     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);//清除颜色缓存和深度缓存
344.     glLoadIdentity();//初始化矩阵为单位矩阵
345.
346.     gluLookAt(eye[0], eye[1], eye[2],
347.         center[0], center[1], center[2],
348.         0, 1, 0);// 场景（0，0，0）的视点中心 (0,5,50)，Y 轴向上
349.
350.     if (bWire) {
351.         glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
352.         //设置多边形绘制模式：正反面，线型
353.     }
354.     else {
355.         glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
356.         //设置多边形绘制模式：正反面，填充
357.     }
358.
359.     glEnable(GL_DEPTH_TEST);//开启深度测试
360.     glEnable(GL_LIGHTING);//开启光照模式
361.
362.     GLfloat white[] = { 1.0, 1.0, 1.0, 1.0 };
363.     GLfloat light_pos[] = { 5,5,5,1 };
364.
365.     glLightfv(GL_LIGHT0, GL_POSITION, light_pos);//光源位置
366.     glLightfv(GL_LIGHT0, GL_AMBIENT, white);//定义白色
367.     glEnable(GL_LIGHT0);//开启第 0 号光源
368.
369.     glRotatef(fRotate, 0, 1.0f, 0);//旋转
370.     glRotatef(-90, 1, 0, 0);
371.     glScalef(0.2, 0.2, 0.2);//缩放
372.     drawTable();//绘制场景
373.
374.     if (bAnim) fRotate += 0.5f;//旋转因子改变
375.     glutSwapBuffers(); //交换缓冲区
376. }
377.
378. int main(int argc, char *argv[])
379. {
```

```
380.     glutInit(&argc, argv);
381.     glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
382.     glutInitWindowSize(480, 480);
383.     int windowHandle = glutCreateWindow("Simple GLUT App");
384.
385.     glutDisplayFunc(redraw);
386.     glutReshapeFunc(reshape);
387.     glutKeyboardFunc(key);
388.     glutIdleFunc(idle);
389.     initTexture();
390.     glutMainLoop();
391.     return 0;
392. }
```