专业：数字媒体技术
姓名：杨锐
学号：3180101941
日期：2020/05/20
地点：家

浙江大学实验报告

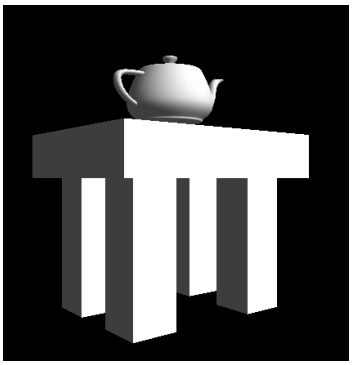
课程名称：计算机图形学 指导老师：唐敏 成绩：_____
实验名称：OpenGL 纹理 实验类型：基础实验 同组学生姓名：_____

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：

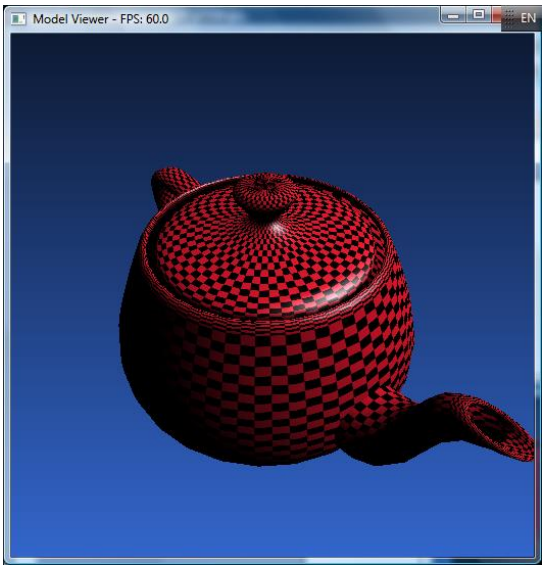
1. 通过设置纹理，使得茶壶纹理为：

装 订 线

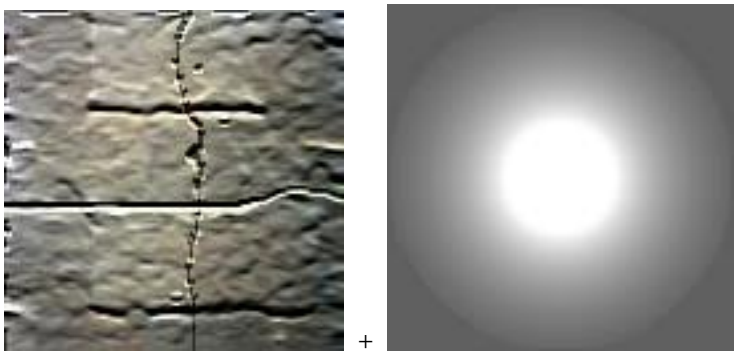
2. 使得桌子纹理为：

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)：

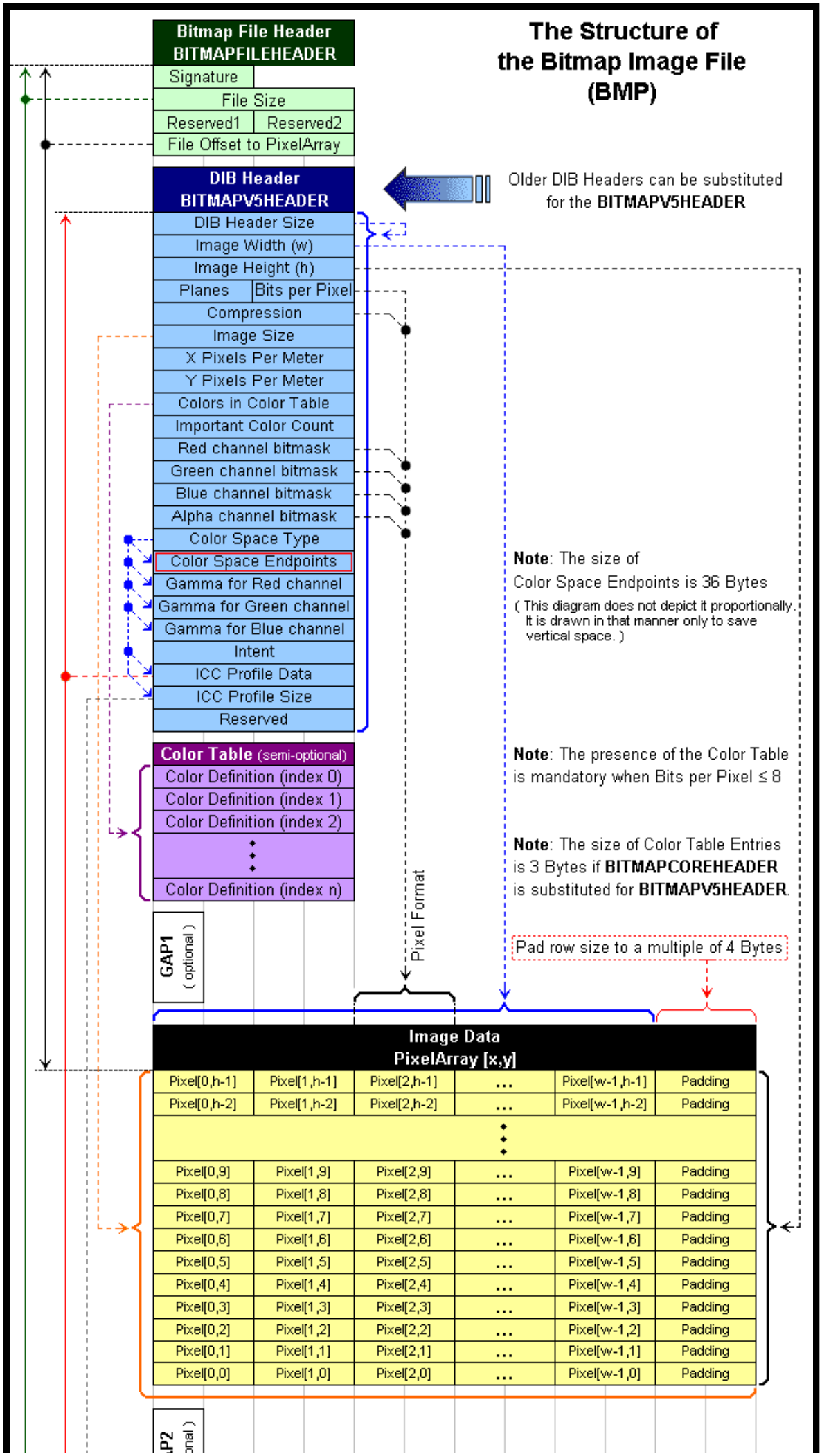


提示：glSolidCube()并不会为多边形指定纹理坐标，因此需要自己重写一个有纹理坐标的方块函数。另外，此实验需要用到 #include <windows.h>

四、操作方法和实验步骤

1.茶壶纹理的设置

在加载纹理之前，我们需要先读取bmp图片的像素数据，因此需要了解bmp图片内部数据的排列方式：


The Structure of the Bitmap Image File (BMP)
Bitmap File Header
BITMAPFILEHEADER
Signature
File Size
Reserved1
Reserved2
File Offset to PixelArray
DIB Header
BITMAPV5HEADER
DIB Header Size
Image Width (w)
Image Height (h)
Planes
Bits per Pixel
Compression
Image Size
X Pixels Per Meter
Y Pixels Per Meter
Colors in Color Table
Important Color Count
Red channel bitmask
Green channel bitmask
Blue channel bitmask
Alpha channel bitmask
Color Space Type
Color Space Endpoints
Gamma for Red channel
Gamma for Green channel
Gamma for Blue channel
Intent
ICC Profile Data
ICC Profile Size
Reserved
Older DIB Headers can be substituted for the BITMAPV5HEADER
Note: The size of Color Space Endpoints is 36 Bytes
(This diagram does not depict it proportionally. It is drawn in that manner only to save vertical space.)
Color Table (semi-optional)
Color Definition (index 0)
Color Definition (index 1)
Color Definition (index 2)
Color Definition (index n)
Note: The presence of the Color Table is mandatory when Bits per Pixel ≤ 8
Note: The size of Color Table Entries is 3 Bytes if BITMAPCOREHEADER is substituted for BITMAPV5HEADER.
GAP1 (optional)
Pixel Format
Pad row size to a multiple of 4 Bytes
Image Data
PixelArray [x,y]
Pixel[0,h-1] Pixel[1,h-1] Pixel[2,h-1] ... Pixel[w-1,h-1] Padding
Pixel[0,h-2] Pixel[1,h-2] Pixel[2,h-2] ... Pixel[w-1,h-2] Padding
Pixel[0,9] Pixel[1,9] Pixel[2,9] ... Pixel[w-1,9] Padding
Pixel[0,8] Pixel[1,8] Pixel[2,8] ... Pixel[w-1,8] Padding
Pixel[0,7] Pixel[1,7] Pixel[2,7] ... Pixel[w-1,7] Padding
Pixel[0,6] Pixel[1,6] Pixel[2,6] ... Pixel[w-1,6] Padding
Pixel[0,5] Pixel[1,5] Pixel[2,5] ... Pixel[w-1,5] Padding
Pixel[0,4] Pixel[1,4] Pixel[2,4] ... Pixel[w-1,4] Padding
Pixel[0,3] Pixel[1,3] Pixel[2,3] ... Pixel[w-1,3] Padding
Pixel[0,2] Pixel[1,2] Pixel[2,2] ... Pixel[w-1,2] Padding
Pixel[0,1] Pixel[1,1] Pixel[2,1] ... Pixel[w-1,1] Padding
Pixel[0,0] Pixel[1,0] Pixel[2,0] ... Pixel[w-1,0] Padding

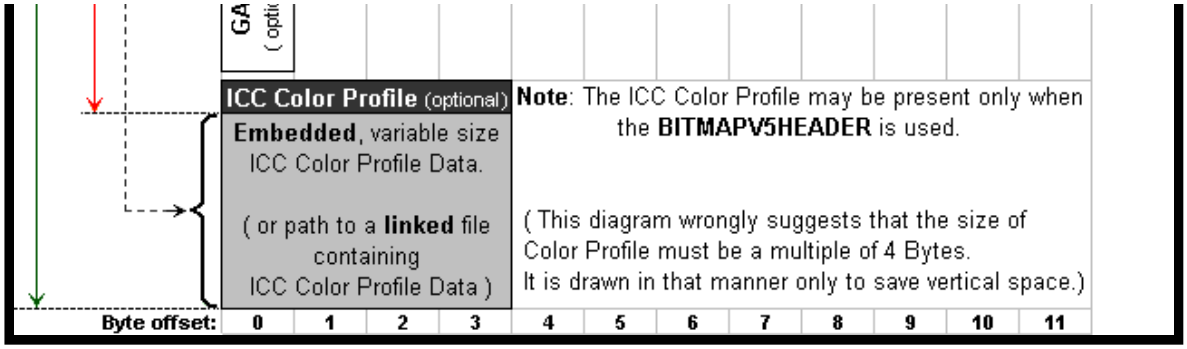


Diagram 1 (From Wikipedia)

简单来说一个bmp文件至少由3部分组成，按顺序如下:*FileHeader*, *InfoHeader*, *PixelData*, 根据图片的位深度和分辨率还可能有:*ColorTable*, *BitMasks*, *Gap*等部分.对于此次实验，我们主要利用到*InfoHeader*和*PixelData*.由于*<Windows.h>*头文件已经为我们预先定义了具体的数据结构，我们依次读取*FileHeader* 和*InfoHeader*，通过*FileHeader*内的*FIleOffset*（指当前位置距*PixelArray*的偏移量）读取像素数据到数组中。

同时由于bitmap像素数据是完全颠倒的：像素之间是从下到上，从右到左，像素内是BGR的顺序，因此我们需要预先将每个像素内的RB交换，否则会使得纹理颜色出现较大偏差。而像素排列顺序对结果的影响就是最终纹理产生的效果是颠倒的，呈现中心对称，比如Monet中人物是颠倒的。但由于茶壶和桌子非常对称，而且交换算法较为费时，就没有修改，（最终效果也别有一番风味:)

图像数据读入完成后，就可以进行纹理的创建了。**首先是为纹理创建标识符**（类似于显示列表的数组）

```
unsigned int texture[5];
glGenTextures(5, texture);//该函数生成多个纹理标识符，并将对应ID依次放入数组
```

之后绑定纹理并制定纹理过滤方式

```
    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);
```

纹理过滤时，需要指明两类方式:

*Mag_FILTER*表示当像素比对应纹理上的像素小时,常用方式包括*NEAREST*,点采样，取最近的像素作为插入值, *LINEAR*双线性插值，核心思想是在两个*x*,*y*方向分别进行一次线性插值。

*MIN_FILTER*表示当像素比对应纹理上的像素大时,常用方式除上述外还包括取最近的*mipmap* ，三线性插值，这里不做过多讨论

然后从图像数据生成纹理

```
    glTexImage2D(GL_TEXTURE_2D,
        0,      //mipmap层次(通常为0，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[i]); //bitmap数据指针
    //printf("%d\n", bitmapInfoHeader.biSizeImage);
}
```

因为数据对齐的原因，其中纹理的宽度和高度都必须是2^n

最后绘制时，就非常简单了，类似于光照

```
    glEnable(GL_TEXTURE_2D);//打开纹理
    glBindTexture(GL_TEXTURE_2D, texture[teapot_statue]);//绑定对应纹理
    ...
    glDisable(GL_TEXTURE_2D);//关闭当前纹理
```

2.桌面纹理的绘制

纹理映射的机制是为图形的每个顶点都设置了一个对应的纹理角点，而*glutSolidCube*函数内部并没有进行相应地处理，因此我们需要自己重新定义立方体的绘制函数，并进行相应地绑定。

这里使用实验2所写的单个立方体的绘制函数并稍做修改，然后绑定纹理角点

```
void Draw_cube() {
    //设置立方体的8个顶点坐标
    GLfloat x1 = -0.5, x2 = 0.5;
    GLfloat y1 = -0.5, y2 = 0.5;
    GLfloat z1 = -0.5, z2 = 0.5;
    GLfloat V[8][3]{
        x1 , y1 , z1,
        x1 , y2 , z1,
        x2 , y2 , z1,
        x2 , y1 , z1,
        x2 , y1 , z2,
        x2 , y2 , z2,
        x1 , y2 , z2,
        x1 , y1 , z2,
    };
    //设置立方体的六个面对应顶点
    GLint Planes[6][4]{
        0 , 1 , 2 , 3,
        4 , 5 , 6 , 7,
        2 , 3 , 4 , 5,
        0 , 1 , 6 , 7,
        1 , 2 , 5 , 6,
        0 , 3 , 4 , 7,
    };
    GLint ord[4][2] = { {1, 1}, {1, 0}, {0, 0}, {0, 1 } };
    glBegin(GL_QUADS);
    for (int i = 0; i < 6; ++i)
        for (int j = 0; j < 4; ++j) {
            glTexCoord2iv(ord[j]);
            glVertex3fv(V[Planes[i][j]]);
        }
    glEnd();
}
```

3.光照与纹理混合

我们可以在绘制时调用函数控制纹理是否受光照影响

```
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not受光照影响
```

4.生成纹理

和直接使用bmp图片类似，我们**首先需要获得*ImageData***,由于参考图片是红黑相间，因此像素的GB都为0，R为0或1，具体代码如下:

```
void generate() {
    for (int i = 0; i < TEXH; ++i)
        for (int j = 0; j < TEXW; ++j) {
            tex[i][j][0] = GLubyte(((i < TEXH/2 ) ^ (j < TEXW / 2) )* 255);
            tex[i][j][1] = 0;
            tex[i][j][2] = 0;
        }
}
```

由于图形是中心对称，因此使用*xor*来控制0/1的填充。同时需要注意*TEXH*和*TEXW*都必须是2^n，这里取8。

然后便是和前面绑定纹理的代码完全一样:

```
    generate();
    glBindTexture(GL_TEXTURE_2D, texture[3]);
    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, GL_RGB, TEXW, TEXH, 0, GL_RGB,
GL_UNSIGNED_BYTE, tex);
    //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.
    //glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    //glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

因此我不太理解**在已经制定像素过滤方式的情况下，为什么还需要使用GL_REPEAT在两个方向上进行重复填充?** 如果是因为生成纹理尺寸与原图像尺寸不匹配的原因，那像素过滤已经能够满足要求，再进行repeat是没有必要的。

同时由于*glTexImage2D*要求*width*和*height*满足2^n，因此只要$n >= 2$，那生成的*ImageData*在内存中是已经满足bmp文件的4字节对齐的，不需要再调用glPixelStorei(GL_UNPACK_ALIGNMENT, 1)进行对齐。

经过调试，仅保留像素过滤方式的函数和生成纹理的函数的效果完全一样。影响显示效果的是*TEXW*和*TEXW*，一定范围内越大，黑红块越稀疏，越小，黑红块越密集

5.两张纹理叠合

如果要使用多重纹理的相关函数需要使用*gl_extensions*,考虑到两张图片的尺寸都是$128 \times 128 \times 3$

我采取取像素平均值的方法进行叠合，具体代码如下:

```
    int k = 2;
    bitmapData[k] = new unsigned char[49152];//128*128*3
    for (unsigned int i = 0; i < 49152; i += 3) {
        bitmapData[k][i] = (bitmapData[k1][i] + bitmapData[k2][i])/2;
        bitmapData[k][i+1] = (bitmapData[k1][i+1] + bitmapData[k2][i+1])/2;
        bitmapData[k][i+2] = (bitmapData[k1][i+2] + bitmapData[k2][i+2])/2;
    }
```

将已经载入的bmp图像数据放入数组中，申请一段新的空间，对应位置的像素取两张图片的平均值，再将图像数据生成纹理。

同时设置两个按键控制茶壶和桌面纹理ID的切换:

```
    case 'u': {
        teapot_statue ^= 3;// 0^3 = 3, 3^3 = 2 实现0-3切换
        break;
    }
    case 'i': {
        table_statue ^= 3;// 1^3 = 2, 2^3 = 1 实现1-2切换
        break;
    }
```

五、实验结果与分析

Figure 1 原图

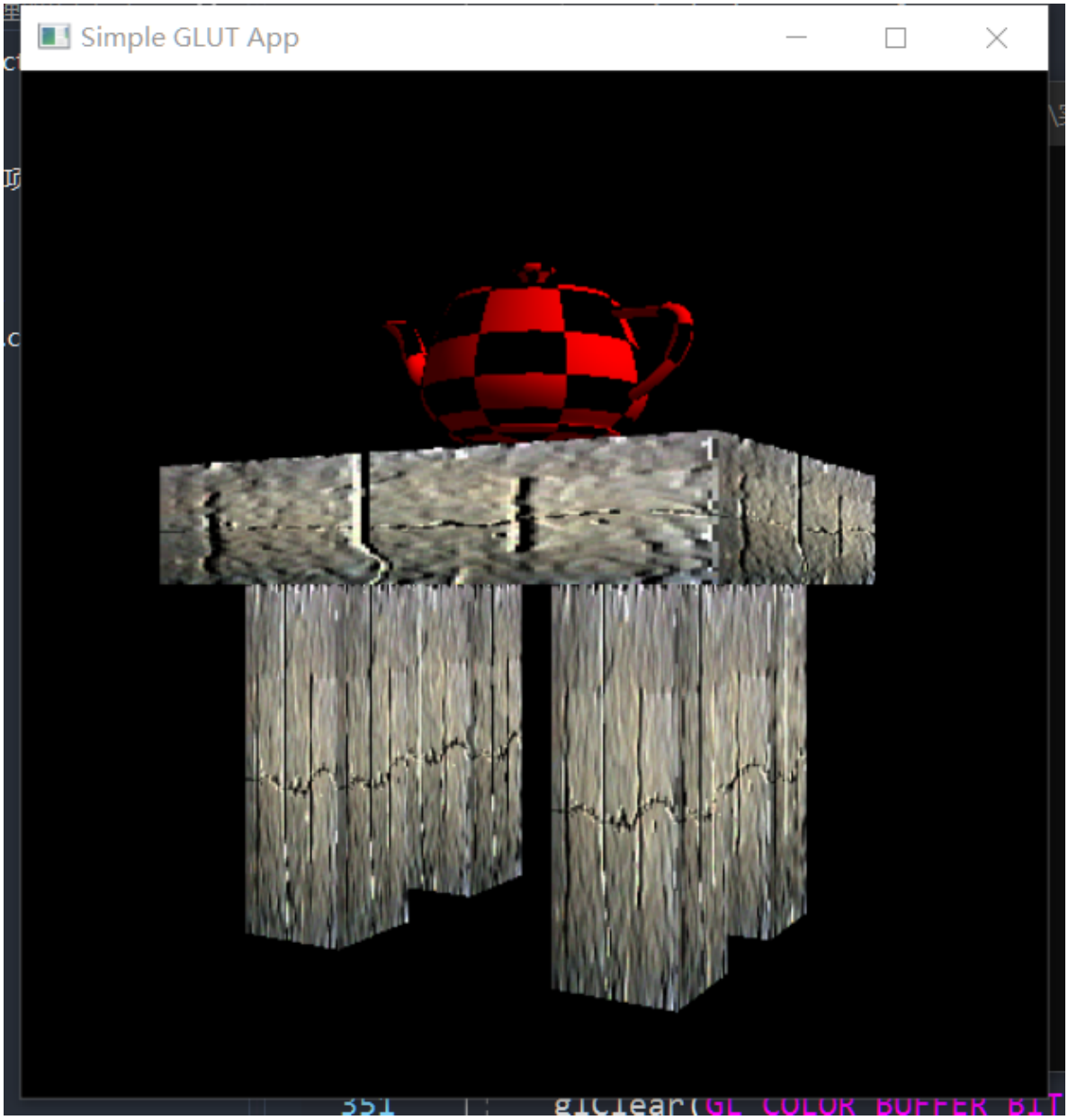


Figure 2 生成纹理

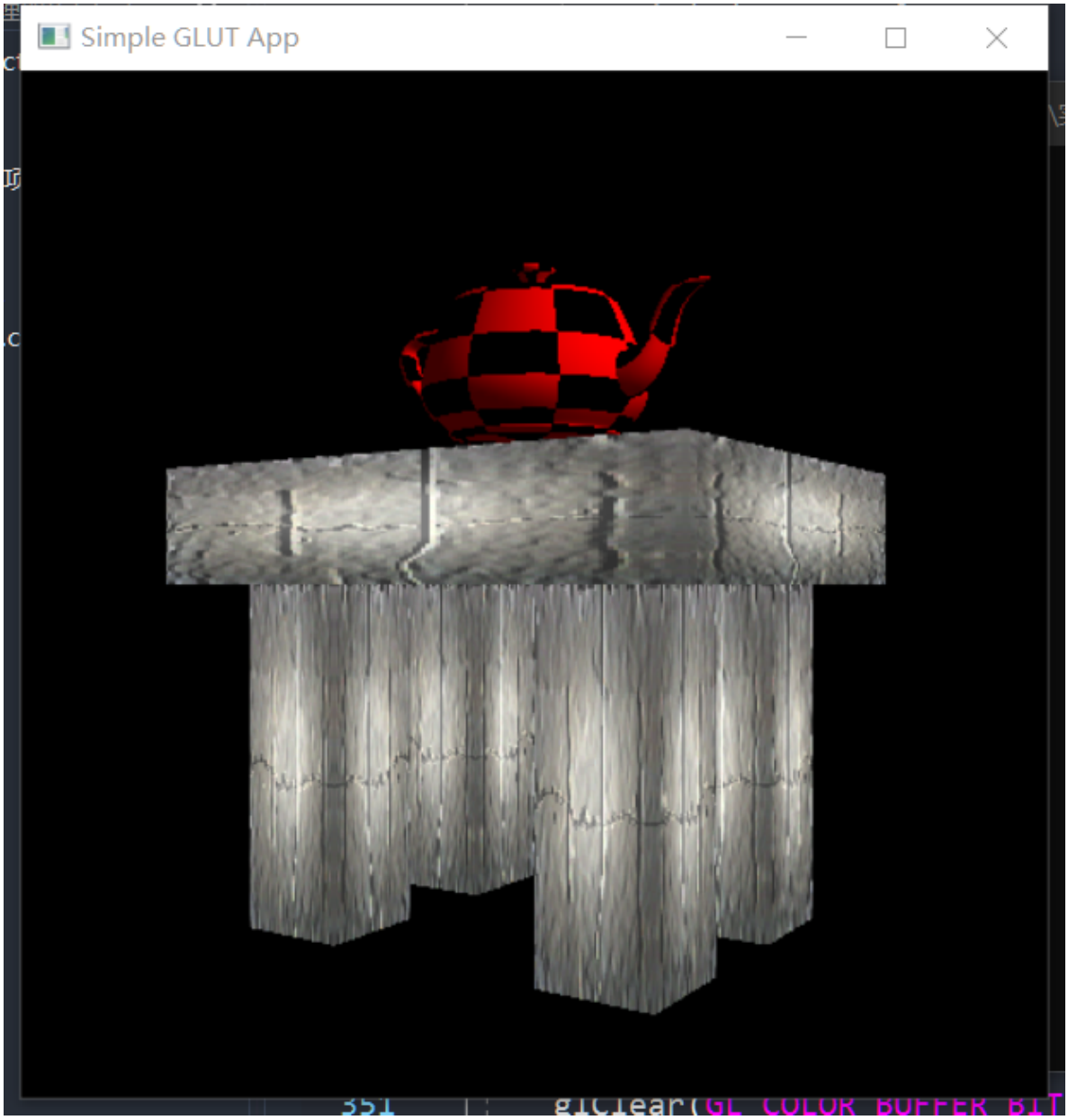


Figure 3 纹理叠合

六、心得

这次实验还是比较有趣的。因为上学期图像信息处理接触过bmp文件的相关处理，加上参考资料里的代码和注释，茶壶纹理部分很快就解决了。桌面虽然要重新绘制cube，不过由于之前实验的代码可用性比较高，因此直接拿过来修改了一些参数效果也很好。在纹理叠合时，一开始参考老师的书，想要使用OpenGL的extensions里的函数*glMultiTexCoord1fARB*,不过看不太懂老师用来初始化的相关函数。想起上学期媒体信号课程里滤波器，加上这个polar.bmp长得也像掩膜的样子:)，边想直接对*ImageData*进行处理算。本来想用中值滤波器的，但一开始试了下直接取平均值，发现效果挺好？可能是因为这两张图片本身数据的色差就不大，像素都是偏灰的，因此结果不会离本身的效果偏差太大。