实验作业说明

实现原理及效果

此处就按照CG1\CG2\CG3的顺序来说明了

基本代码框架

- 首先在此列举一些固定的需要写的函数以及结构

```
//基本结构
int main(){
    glutInit(&argc, argv);
    glutInitWindowSize(640, 640);
    glutInitWindowPosition(50, 50);
    int mode = GLUT_RGB | GLUT_SINGLE;
    glutInitDisplayMode(mode);
    glutCreateWindow("homework"); //创建窗口
    glutDisplayFunc(&display);
    glutMainLoop();// 调用函数来绘制
    return 0;
}
void display(){
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();//初始化
    glFlush();
}
```

物体旋转

- 使用glut的自带函数监听按键，通过对于全局变量angle的加减，和display函数里的旋转，完成对于整体图形的旋转。答题代码结构如下所示：

```
GLfloat xRot_triangle = 0.0f;
GLfloat yRot_triangle = 0.0f;

void SpecialKeys(int key, int x, int y) {
    if (key == GLUT_KEY_UP) {
        xRot_triangle += 5.0f;//通过调整等号右边的数字的大小，控制每点一次按键，模型旋转的角度。
    }
    ...
    glutPostRedisplay();
}
int main(){
    ...
        glutSpecialFunc(SpecialKeys);
    ...
}
void display(){
    glRotatef(xRot_triangle, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot_triangle, 0.0f, 1.0f, 0.0f);
    ...
}
```

CG1

原理

- CG1完成的是去除两个面的立方体，并使其旋转，将一些2D的图形放在了中间
- **基本图形的绘制**

  由于采用的固定流水线，所以图形的绘制较为简单，只需要用opengl的固定模板即可，例如

  ```
  glBegin(GL_QUADS)
  glColor3fv(vertex[index[i][j]]);
  glVertex3fv(vertex[index[i][j]]);
  glEnd();

  glBegin(GL_POLYGON);
  glBegin(GL_TRIANGLES);
  ```

  此处就可以指定四边形的顶点颜色以及坐标，绘制一个四边形，立方体就是由六个（去掉两个是四个）四边形拼接起来即可

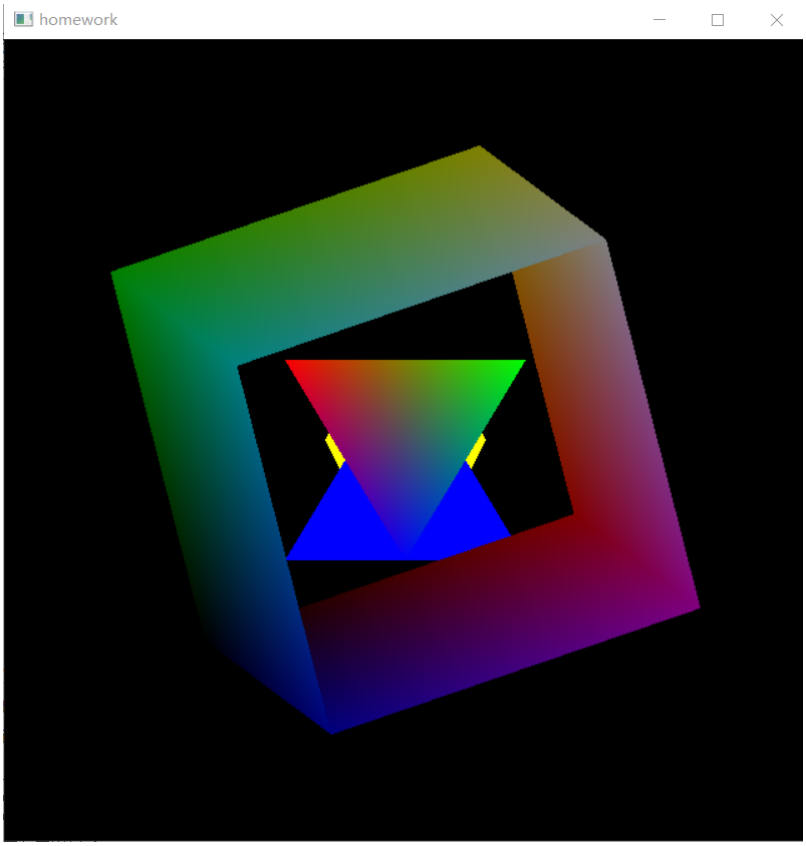
- **旋转效果**的实现如下：

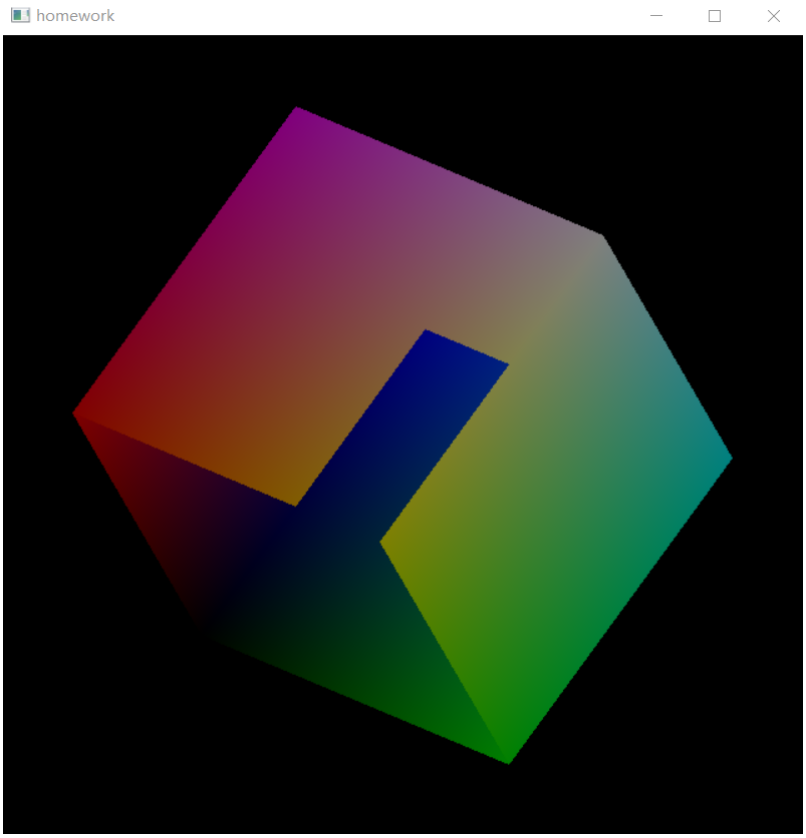
  ```
  void myAnim(void)
  {
      angle += 0.01f;
      if (angle > 360) {
          angle = 0.0f;
      }
      display();
  }
  int main(int argc, char** argv) {
      ...
      glutIdleFunc(&myAnim);  // 不断调用显示函数，达到动画效果
      ...
  }
  int display(){
      ...
      glRotatef(angle, 1, 1, 1);//沿着3轴旋转一次
      ...
  }
  ```

  创建了自己的myAnim函数，对全局变量angle随着调用不断进行修改，在display函数中按照angle的角度不断旋转，main函数也要使用特定函数来调用。

效果

-



- 

此处可以看到2D图形和立方体的遮挡关系，立方体的透视，立方体的旋转

CG2

原理

- 实现的是使用递归细分绘制的球面
- 此处在display函数里使用了新学习的函数操作：

```
    // 设置逆时针排列的点围成的平面为正面
    glFrontFace(GL_CCW);
    // 设置不绘制背面，节省算力同时不会出现背面覆盖正面的情况
    glCullFace(GL_BACK);
    glEnable(GL_CULL_FACE);
    // 设置背景为白色
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    // 加载单位阵
    glLoadIdentity();
```

- 这里实现了当窗口**放大缩小**的时候图像的比例不会变化，实现方法如下：

```
// 窗口大小自适应函数，使得窗口大小改变时仍保持图形的比例不变
// 有关窗口自适应函数：http://blog.sina.com.cn/s/blog_5497dc110102w8qh.html
void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei)w, (GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (GLfloat)w / (GLfloat)h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(2, 2, 2, 0.0, 0.0, 0.0, -1, -1, 1);
}
int main(){
    glutReshapeFunc(reshape);    //需要在main函数里调用
}
```

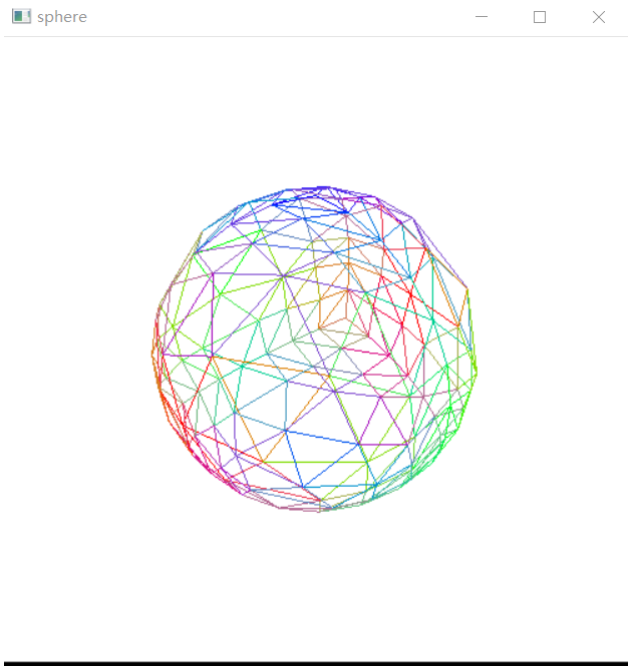
参考了网上的blog，其具体原理就是当放大或者缩小的是时候维持视角的缩放。

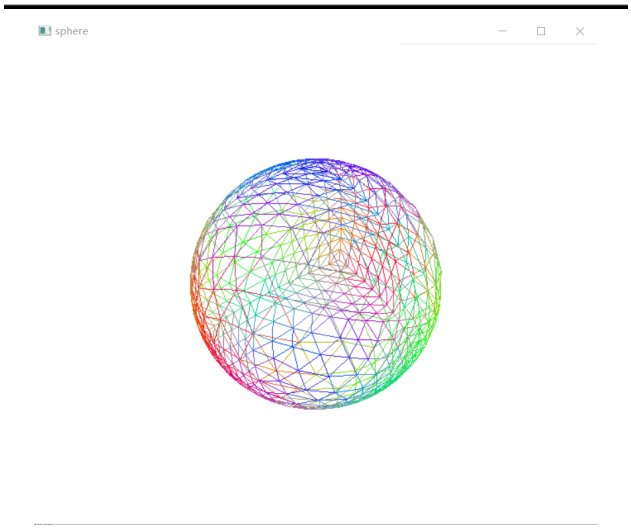
- **递归细分画球面**实际上不难，具体思路就是将一个正四面体的每一个面都向外“推”成一个球面，也就是将一个三角形分成很多三角形，但同时还要保证所有三角形的顶点到达球心的距离都是一样的。这里我地代码架构如下：

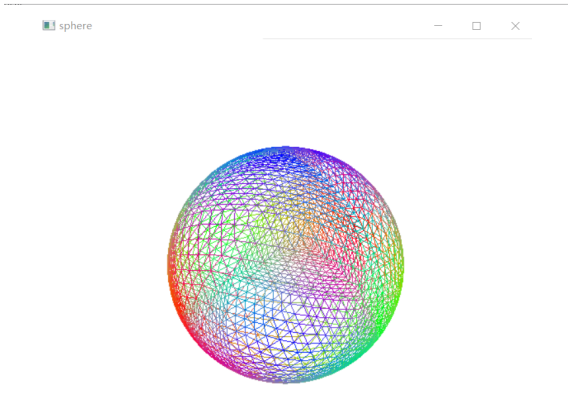
```
//将顶点正则化
void normalize(GLfloat* v)
{
    GLfloat d = sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
    v[0] /= d; v[1] /= d; v[2] /= d;
}
//实现一个三角形分成四个三角形
void divide_triangle(GLfloat* a, GLfloat* b, GLfloat* c, int depth)
{
    if (depth > 0) {
        GLfloat ab[3], ac[3], bc[3];
        for (unsigned int i = 0; i < 3; i++)
            ab[i] = a[i] + b[i];
        normalize(ab);
        ....
        divide_triangle(a, ab, ac, depth - 1);
        divide_triangle(b, bc, ab, depth - 1);
        divide_triangle(c, ac, bc, depth - 1);
        divide_triangle(ab, bc, ac, depth - 1);
    }
}
```

效果

- 由于我设置了全局变量Depth，来控制递归的层数，所以此处我分别列举Depth为345时的递归球面：







可以看到随着递归数的升高，越来越接近一个球面了

CG3

原理

- 在这里主要实现了以下几个功能：开启光照，设置至少两个光源，透视投影，物体的阴影效果，半透明效果，学号姓名图片的纹理映射，立方体环境映射
- 首先我学习了关于**光照**，透视投影等的控制函数：

```
void init()
{
    // 启用抗锯齿（使线平滑）
    glEnable(GL_BLEND);
    glEnable(GL_LINE_SMOOTH);
    glHint(GL_LINE_SMOOTH_HINT, GL_FASTEST);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    // 设置材质和光照的信息
    // 有关光照与材质：https://blog.csdn.net/timidsmile/article/details/7017197
    GLfloat mat_ambient[4] = { 1.0f, 1.0f, 1.0f, 0.0f };
    GLfloat mat_diffuse[4] = { 1.0f, 1.0f, 1.0f, 0.0f };
    GLfloat mat_specular[4] = { 1.0f, 1.0f, 1.0f, 1.0f };   //冯模型
    GLfloat mat_shininess[4] = { 100.0f };
    GLfloat light_position_0[4] = { 0.0f, -6.0f, 0.0f, 1.0f }; //光源位置设置
    GLfloat light_position_1[4] = { 0.0f, 0.0f, 6.0f, 1.0f };
    // 设置正向面的材质和光源的光照
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position_0);
    // 设置颜色材料，使光照模式下仍然可以显示原本的颜色
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
    // 启用平滑着色功能
    glShadeModel(GL_SMOOTH);
    // 启用光照功能
    glEnable(GL_LIGHTING);
    // 启用0号光源
    glEnable(GL_LIGHT0);
}
```

- 物体的**阴影效果**我则是使用了基本的矩阵变换，将正方体变为了一个二维正方形，并且平移一下，其具体实现函数如下所示：

  其阴影颜色的控制我则是放在了正方体的绘制中传递了Flag进行特判

```
void cube_shadow(GLfloat x, GLfloat y, GLfloat z, GLfloat size)
{
    GLfloat m[16] = { 0.0f };
    m[0] = m[5] = m[10] = 1.0f;
    m[11] = -1.0f / z;
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glTranslatef(x, y, z);
    glMultMatrixf(m);
    glTranslatef(-x, -y, -z);
    color_cube(size, true);
    glPopMatrix();
}
```

- 物体的**半透明效果**实现起来则是较为容易，只需要调用现成的函数进行开启即可：

```
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glDisable(GL_BLEND);
```

其中glBlendFunc的使用可以看这篇blog：https://blog.csdn.net/u012209626/article/details/45750559

- **纹理映射**的实现实在是万分艰难，最终我找到了FreeImage库来进行图片的提取，但是在这个过程中我也了解了bmp格式的头信息的结构。纹理映射的基本逻辑就是，首先加载一个纹理，与一个纹理号进行绑定，之后便使用纹理绑定函数与图形绑定即可，具体实现如下：

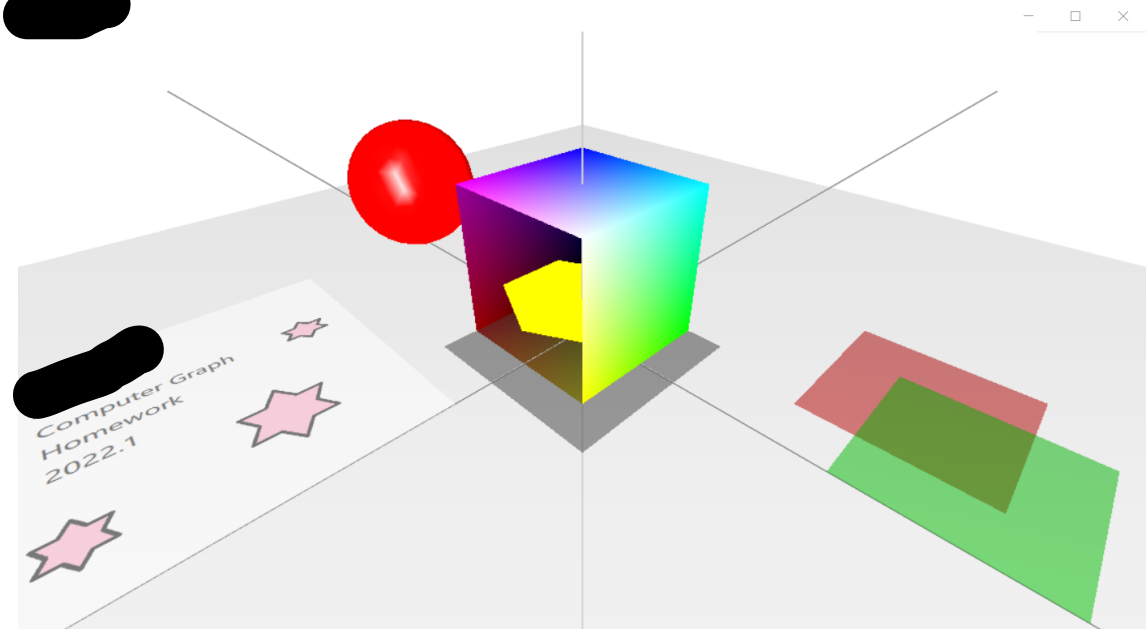
```
/* 函数load_texture
* 读取一个BMP文件作为纹理
* 如果失败，返回0，如果成功，返回纹理编号
*/
GLuint load_texture(const char* file_name)
{
    FreeImage_Initialise(TRUE);
    //加载图片
    FIBITMAP* JPEG = FreeImage_Load(FIF_BMP, file_name, 0);
    //获取影像的宽高，都以像素为单位
    width = FreeImage_GetWidth(JPEG);
    height = FreeImage_GetHeight(JPEG);
    pixels = (GLubyte*)FreeImage_GetBits(JPEG);
    // 分配一个新的纹理编号
    glGenTextures(1, &texture_ID);
    if (texture_ID == 0)
    {
        //free(pixels);
        FreeImage_Unload(JPEG);
        return 0;
    }
    // 绑定新的纹理，载入纹理并设置纹理参数
    // 在绑定前，先获得原来绑定的纹理编号，以便在最后进行恢复
    GLint lastTextureID = last_texture_ID;
    glGetIntegerv(GL_TEXTURE_BINDING_2D, &lastTextureID);
    glBindTexture(GL_TEXTURE_2D, texture_ID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
        GL_BGR_EXT, GL_UNSIGNED_BYTE, pixels);
    glBindTexture(GL_TEXTURE_2D, lastTextureID);  //恢复之前的纹理绑定
    //free(pixels);
    FreeImage_Unload(JPEG);
    return texture_ID;
}
void namecard(void)
{
    glBindTexture(GL_TEXTURE_2D, texname);
    glBegin(GL_QUADS);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 0.0f, 1.0f); //绑定
    ...
    glEnd();
}
void display(){
    glEnable(GL_TEXTURE_2D);    // 启用纹理
    ...
    glDisable(GL_TEXTURE_2D);   // 关闭纹理
}
```

- **环境映射**实际上就是一个纹理映射，只不过需要按照角度进行纹理采样，并进行一些矩阵运算，完成反射

效果

- [redacted]



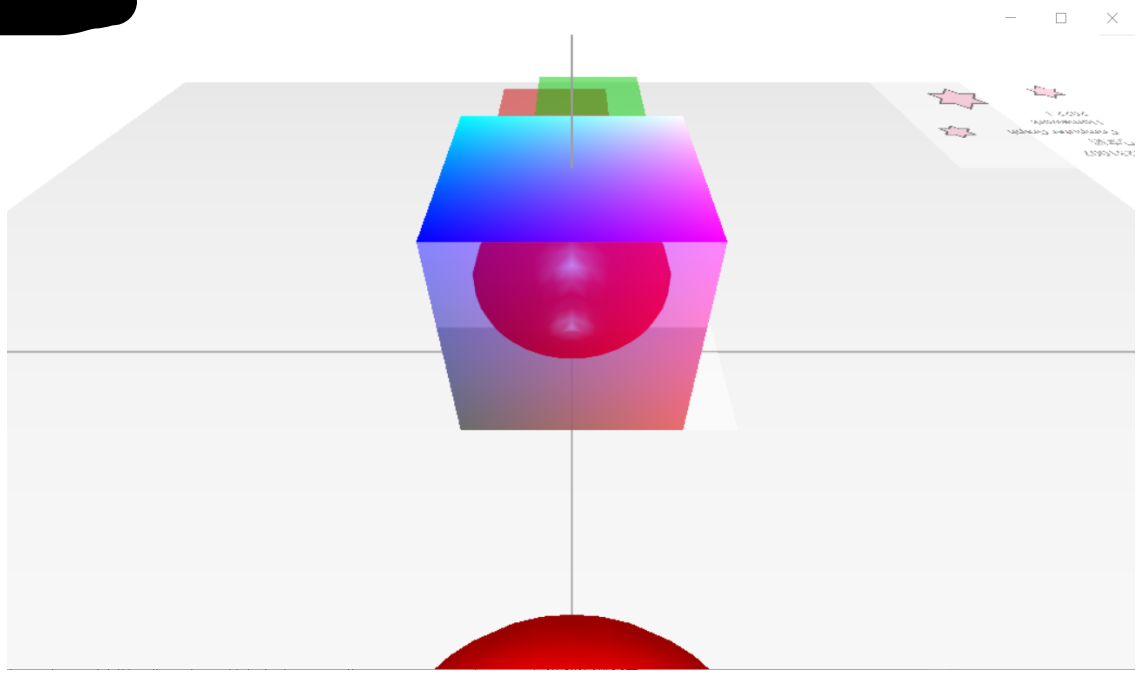
这张图片显示了CG3的全貌，左下角是一个印着我的姓名学号的纹理图片，绑定在了一个正方形上；

右侧的是连个半透明的不同颜色的正方形，可以看到由于透视，其相交区域的颜色是叠加的；

总共设置了两个光源，一个在图片的后方，一个在前方，前方导致的球的光泽可以被看到；后方生成的阴影在正方体的下方；

正方体的中间有二维图形，其左侧有一个球；

-



将视角移到正方体的侧面，我们可以看到由于环境映射，球的反射的像清晰可见。