作业3 使用GLSL

18340012 陈晨

1 实验题目

要求：绘制一个小球并使用GLSL实现不同着色模型

- **必要:**
  - 实现环境光反射，漫反射，镜面反射
- 比较三种着色模型的效果
  - Flat， Gouraud， Phong
- 比较其他绘制参数变化时，绘制结果的变化
  - 如不同反射类型的光照强度，镜面反射中的指数α等
- 比较小球细分程度不同时，绘制结果的变化

可选:

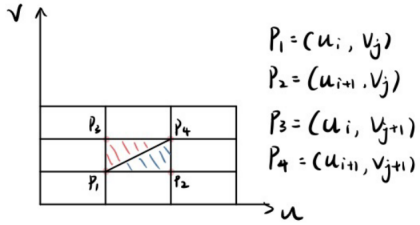
- 对小球进行形变，使用多个光源
- 根据顶点在三维空间中的位置指明颜色
- 根据片元在二位屏幕空间上的法向量变化指明颜色

2 实验过程

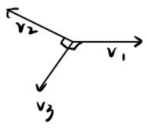
- 绘制小球

  根据球面的数学模型，可以得到角度组(u,v)对应的球面坐标。

  遍历不同的$（u,v）$，得到对应球面坐标，通过画三角形近似球面。易知下面四个点共面。



  除此之外，还需要计算三角形的法向量，用于光照的计算。四个点的法向量均认为是这个面的法向量。



  代码实现如下。由于三角形的画点顺序为顺时针，所以在之后的代码中需要通过`glFrontFace(GL_CW);`注明。

```
#define M_PI       3.14159265358979323846
GLfloat* get_point_on_ball(GLfloat u, GLfloat v, GLfloat r) {
    GLfloat* points = new GLfloat[3];
    points[0] = r * sin(M_PI * u) * cos(2 * M_PI * v);//x
    points[1] = r * sin(M_PI * u) * sin(2 * M_PI * v);//y
    points[2] = r * cos(M_PI * u);//z
    return points;
```

```
}

void create_ball(GLfloat* ball, GLuint u_num, GLuint v_num, GLfloat r) {
    //根据采样数量计算步长
    GLfloat u_step = 1.0f / u_num;
    GLfloat v_step = 1.0f / v_num;
    GLuint offset = 0;
    for (int u = 0; u < u_num; u++) {
        for (int v = 0; v < v_num; v++) {
            GLfloat* point_1 = get_point_on_ball(u * u_step, v * v_step, r);
            GLfloat* point_2 = get_point_on_ball((u + 1) * u_step, v * v_step, r);
            GLfloat* point_3 = get_point_on_ball((u + 1) * u_step, (v + 1) * v_step, r);
            GLfloat* point_4 = get_point_on_ball(u * u_step, (v + 1) * v_step, r);

            //计算法向量
            GLfloat v1[3], v2[3], v3[3];
            v1[0] = point_2[0] - point_1[0];
            v1[1] = point_2[1] - point_1[1];
            v1[2] = point_2[2] - point_1[2];

            v2[0] = point_3[0] - point_2[0];
            v2[1] = point_3[1] - point_2[1];
            v2[2] = point_3[2] - point_2[2];

            v3[0] = (v1[1] * v2[2]) - (v2[1] * v1[2]);
            v3[1] = (v1[2] * v2[0]) - (v2[2] * v1[0]);
            v3[2] = (v1[0] * v2[1]) - (v2[0] * v1[1]);

            // 单位化
            GLfloat a = 1.0 / sqrt(v3[0] * v3[0] + v3[1] * v3[1] + v3[2] * v3[2]);
            for (int i = 0; i < 3; i++) {
                v3[i] *= a;
            }

            GLfloat* points[6] = { point_1, point_4, point_3, point_1, point_3, point_2 };
            for (int i = 0; i < 6; i++) {
                memcpy(ball + offset, points[i], sizeof(GLfloat) * 3);
                offset += 3;
                memcpy(ball + offset, v3, sizeof(GLfloat) * 3);
                offset += 3;
            }
        }
    }
}
```

- VBO, VAO的初始化

  顶点数据（三维坐标和法向量）存储在VBO中。通过glGenBuffers函数生成VBO的id并通过glBindBuffer绑定，便创建了VBO。然后调用glBufferData函数，把通过create_ball函数得到的数据复制到当前绑定的缓冲中。

VAO存储了顶点属性相关的信息。通过glGenVertexArrays函数生成VAO的id，通过glBindVertexArray进行绑定，以生成VAO。

最后链接顶点属性。由于顶点数据为三维坐标和法向量交替存储，所以一个顶点包含了6个GLfloat数据。通过glVertexAttribPointer和glEnableVertexAttribArray指定坐标和法向量两个属性的位置。

```
void MyGLWidget::init_vbo(int u_num_, int v_num_) {
    u_num = u_num_, v_num = v_num_;
    //创建球的数组
    GLfloat* ball = new GLfloat[6 * 6 * u_num * v_num];
    create_ball(ball, u_num, v_num, 1.0f);

    //创建VBO，VAO
    GLuint VBO, VAO;
    glGenBuffers(1, &VBO);
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 6 * 6 * u_num * v_num, ball, GL_STATIC_DRAW);

    //链接顶点属性
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (void*)0);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (void*)(3 * sizeof(GLfloat)));
    glEnableVertexAttribArray(1);
}
```

- 着色器的初始化

顶点着色器和片段着色器分别存储在两个文件中。着色器需要通过glCreateShader函数得到对应id，然后通过glShaderSource指定从文件读出的着色器的源码，并且通过glCompileShader进行编译。然后创建着色器程序，将两个着色器附加到着色器程序之上，链接并启用。之后画图便会调用这个程序中的着色器了。

由于展示过程中，投影矩阵不发生改变，所以在这里对传入着色器的投影矩阵进行初始化。

```
void MyGLWidget::init_shaders(const char* v_path, const char* f_path) {
    //读取着色器
    string v_code, f_code;
    ifstream v_shader_file, f_shader_file;
    v_shader_file.open(v_path);
    f_shader_file.open(f_path);
    stringstream v_shader_stream, f_shader_stream;
    v_shader_stream << v_shader_file.rdbuf();
    f_shader_stream << f_shader_file.rdbuf();
    v_shader_file.close();
    f_shader_file.close();
    v_code = v_shader_stream.str();
    f_code = f_shader_stream.str();
    const char* v_shader_source = v_code.c_str();
    const char* f_shader_source = f_code.c_str();

    // 顶点着色器
```

```
    GLuint v_shader;
    v_shader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(v_shader, 1, &v_shader_source, NULL);
    glCompileShader(v_shader);

    //片段着色器
    GLuint f_shader;
    f_shader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(f_shader, 1, &f_shader_source, NULL);
    glCompileShader(f_shader);

    //着色器程序
    shader_program = glCreateProgram();
    glAttachShader(shader_program, v_shader);
    glAttachShader(shader_program, f_shader);
    glLinkProgram(shader_program);
    glDeleteShader(v_shader);
    glDeleteShader(f_shader);
    glUseProgram(shader_program);

    //获取并设置投影矩阵
    GLfloat projection_matrix[16];
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0f, (GLfloat)width() / (GLfloat)height(), 0.1f,
200.0f);
    glGetFloatv(GL_PROJECTION_MATRIX, projection_matrix);
    glUniformMatrix4fv(glGetUniformLocation(shader_program,
"projection"), 1, GL_FALSE, projection_matrix);
}
```

- 画图

由于展示过程中会对球进行旋转，所以声明static的变量angle，作为旋转的角度，程序每次调用update函数时，angle会进行更新。因此，每次传入着色器的modelview矩阵也会变化，赋值过程放在draw函数中，而并非init_shader函数中。

```
void MyGLWidget::draw(){
    //获取并设置modelview矩阵
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glPushMatrix();

    static float angle = 0.0f;

    GLfloat modelview_matrix[16];
    glTranslatef(0.0f, 0.0f, -5.0f);
    glRotatef(angle, 1.0f, 1.0f, 0.0f);
    glGetFloatv(GL_MODELVIEW_MATRIX, modelview_matrix);
    glUniformMatrix4fv(glGetUniformLocation(shader_program,
"modelview"), 1, GL_FALSE, modelview_matrix);

    glDrawArrays(GL_TRIANGLES, 0, 6 * u_num * v_num);

    angle += 1.0f;
    if (abs(angle - 360.0f) <= 1e-1)angle = 0.0f;

```

```
    glPopMatrix();
}
void MyGLWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    glFrontFace(GL_CW);

    draw();
}
```

- 实现光照

  首先，声明光照颜色，物体颜色，光照位置，观察位置。

```
vec3 object_color = vec3(0.6f, 0.4f, 0.8f);
vec3 light_color = vec3(1.0f, 1.0f, 1.0f);

vec3 light_pos = vec3(-1.0f, 1.0f, 0.0f);
vec3 view_pos = vec3(0.0f, 0.0f, 0.0f);
```

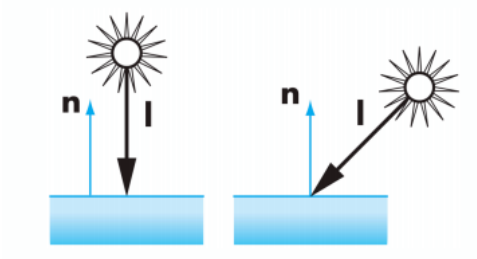
  - 环境光

    环境光反射强度$I_\alpha = k_\alpha L_\alpha$。其中$k_\alpha$为环境光反射系数，$L_\alpha$为环境光强度。

```
float ambient_rate = 0.3;
vec3 ambient = ambient_rate * light_color;
```

  - 漫反射光

    漫反射强度$I_d=k_dL_d(n·l)$。其中n为照射点处法向量，l为光照向量（从照射点到光源）。若两个向量的夹角越小，说明光更加直射，则产生的反射光越强。由于只考虑夹角，不考虑距离，因此这里对于n,l都需要进行标准化。



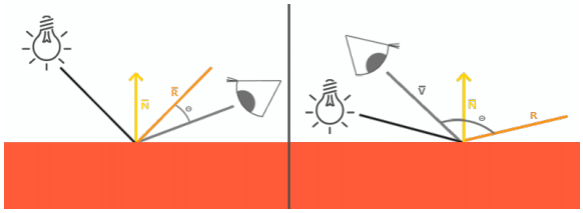

```
float diffuse_rate = 1.0;
vec3 light_dir = normalize(light_pos - frag_pos);
float diff = max(dot(frag_normal, light_dir), 0.0);
vec3 diffuse = diffuse_rate * diff * light_color;
```

    由于法向量在变换中会变形，因此需要对法向量进行处理：

```
frag_normal = normalize(mat3(transpose(inverse(modelview))) *
normal);
```

  - 镜面反射光

镜面反射光强度$I_s=k_sL_s·\max((r·v)^\alpha,0)$。其中$r$为反射方向，$v$为观察方向，$\alpha$为高光系数。由于反射方向与观察方向的点积可能为负，此时不存在镜面反射，因此要保证最终的强度大于等于0。



```
float specular_rate = 0.5;
vec3 view_dir = normalize(view_pos - frag_pos);
vec3 reflect_dir = reflect(-light_dir, frag_normal);
float spec = pow(max(dot(view_dir, reflect_dir), 0.0), 64);
vec3 specular = specular_rate * spec * light_color;
```

最后，将三个光照的分量相加，并乘以物体颜色，就得到了最终的颜色。

```
vec3 result = (ambient + diffuse + specular) * object_color;
frag_color = vec4(result, 1.0);
```

- 坐标处理

  在处理光照时，光照的位置不随物体位置变换而变换，因此在光照的计算时，物体（照射点）的位置是变换后的位置，着色器需要对此进行计算。

```
frag_pos = vec3(modelview * vec4(pos, 1.0));
```

而物体的点最终还要经过投影，因此最终输出的坐标需要左乘投影矩阵：

```
gl_Position = projection * modelview * vec4(pos, 1.0);
```

- 不同的着色模型

  本次实验中，着色模型均通过一个顶点着色器和一个片段着色器实现。

  - flat

    flat着色模型中，每个多边形只需要应用光照模型计算一次颜色，因此光照的计算在顶点着色器中实现即可。由于不需要进行颜色的插值计算，所以需要通过flat进行声明。flat采用面的法向量。

```
//顶点着色器
#version 330 core
layout (location = 0) in vec3 pos;
layout (location = 1) in vec3 normal;

uniform mat4 modelview;
uniform mat4 projection;

flat out vec4 color;

void main()
{
    gl_Position = projection * modelview * vec4(pos, 1.0);

```

```
    vec3 frag_pos = vec3(modelview * vec4(pos, 1.0));
    vec3 frag_normal = normalize(mat3(transpose(inverse(modelview))) * normal);

    ...//光照计算
    vec3 result = (ambient + diffuse + specular) * object_color;
    color = vec4(result, 1.0);
}

//片段着色器
#version 330 core
out vec4 frag_color;
flat in vec4 color;

void main()
{
    frag_color = color;
}
```

- gourand

  gourand着色模型是基于插值的方法。它根据每个顶点的法向量，算出光照之后，对于每个面根据顶点进行颜色的插值。因此，它的光照计算依旧可以在顶点着色器中实现。当不声明flat时，opengl会自动认为是颜色插值。除此之外，gouraud采用顶点的法向量，而此次实验画的球是以原点为球心的，因此球面某点的三维坐标与其法向量数值上相等。其它部分的实现与flat相同。

```
frag_normal = normalize(mat3(transpose(inverse(modelview))) * pos);
```

- phong

  phong着色模型需要计算内部的顶点法向量，因此将光照计算放在片段着色器中。

```
//顶点着色器
#version 330 core
layout (location = 0) in vec3 pos;
layout (location = 1) in vec3 normal;

uniform mat4 modelview;
uniform mat4 projection;

out vec3 frag_pos;
out vec3 frag_normal;

void main()
{
    gl_Position = projection * modelview * vec4(pos, 1.0);

    frag_pos = vec3(modelview * vec4(pos, 1.0));
    frag_normal = normalize(mat3(transpose(inverse(modelview))) * pos);
}

//片段着色器
#version 330 core

out vec4 frag_color;
```

```
24
25  in vec3 frag_pos;
26  in vec3 frag_normal;
27
28  void main()
29  {
30      ...//光照计算
31      vec3 result = (ambient + diffuse + specular) * object_color;
32      frag_color = vec4(result, 1.0);
33  }
```

- 根据顶点在三维空间中的位置指明颜色

  这里根据球的起始坐标给对应的颜色赋值，因此只需将球面的变换前的三维坐标标准化，作为RGB值，传给片段着色器作为输出颜色即可。

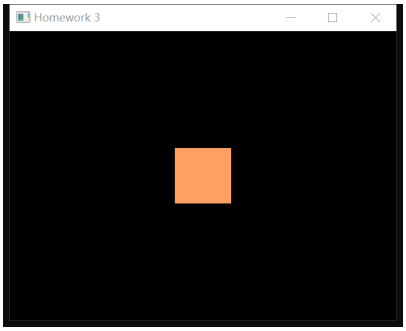
```
1   //顶点着色器
2   #version 330 core
3   layout (location = 0) in vec3 pos;
4   layout (location = 1) in vec3 normal;
5
6   uniform mat4 modelview;
7   uniform mat4 projection;
8
9   out vec4 color;
10
11  void main()
12  {
13      gl_Position = projection * modelview * vec4(pos, 1.0);
14      vec3 n_pos = normalize(pos);
15      color = vec4(n_pos, 1.0f);
16  }
17
18  //片段着色器
19  #version 330 core
20
21  out vec4 frag_color;
22  in vec4 color;
23
24  void main()
25  {
26      frag_color = color;
27  }
```

3 实验结果

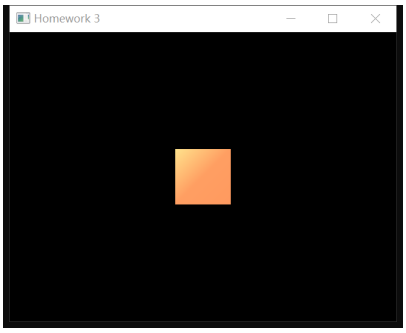
- 比较三种着色模型的效果

  为了更好观察三种着色模型的差异，这里画了一个正方体。其中光照位置在(-1.0f, 1.0f, 0.0f)，观察位置在(0.0f, 0.0f, 0.0f)，图上所示的面x,y属于-0.5f到0.5f，z=-5.0f。

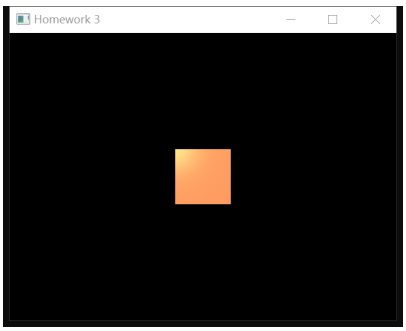
  - Flat



- Gouraud



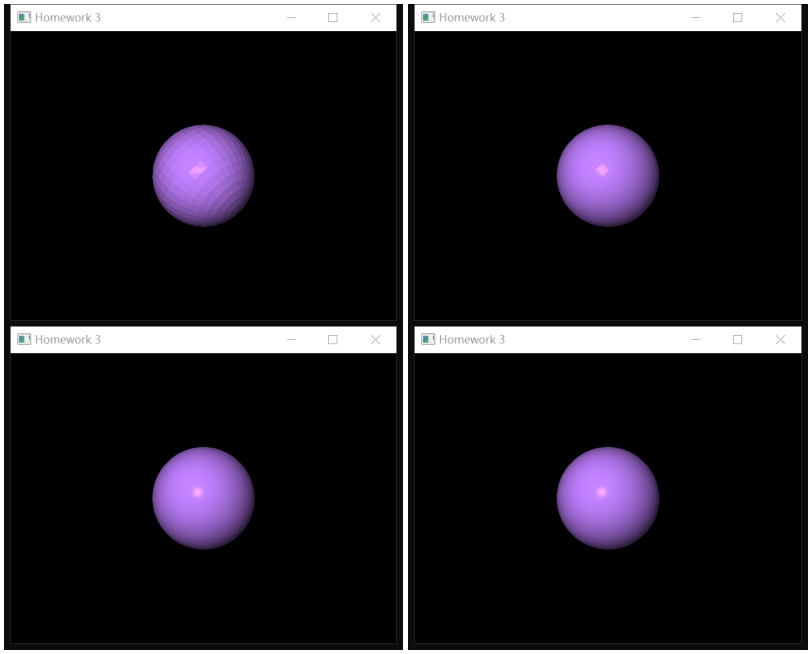
- Phong



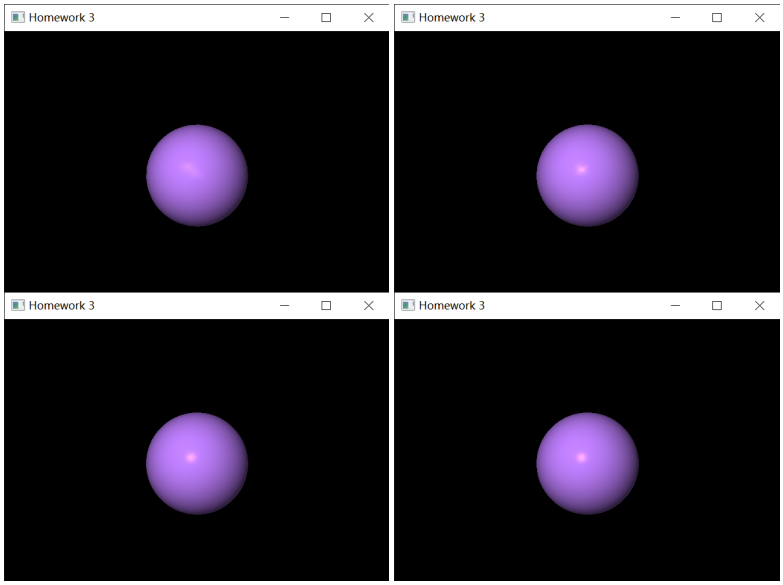
可以观察到，flat着色模型中，图上正方体的一个面都是同一个颜色。gourand着色模型中，浅色的边界更像是正方形的对角线，而phong着色模型中，浅色的边界更像是以正方形左上角为圆心的四分之一圆。这是由于gourand着色模型中，图中正方形的颜色是由四个点确定的，而phong着色模型中，在面的内部还进行了光照计算。

对于作用在球面的情况，根据不同的细分程度（u_num = 20, 40, 80, 160, v_num = 40, 80, 160, 320），得到的结果如下：

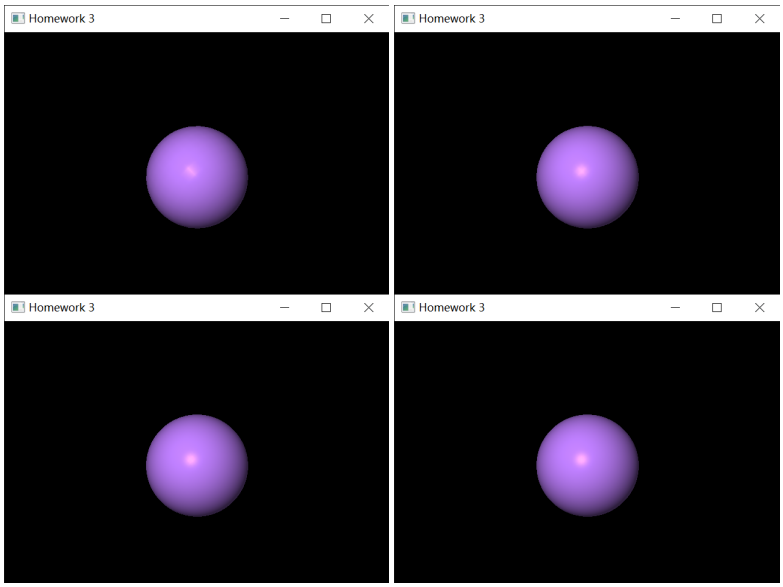
- Flat



- Gouraud



- Phong

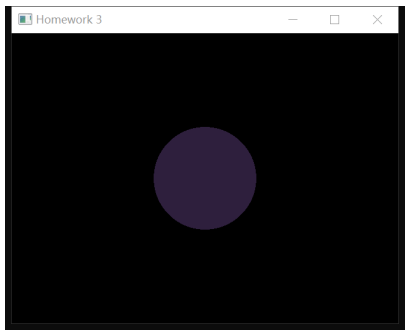


可以观察到，在球面不同的细分程度之下，三种着色方式的差距随着细分程度增加而减小。其中Flat在细分程度较小的情况下，棱角更为明显，而Gouraud和Phong的球面棱角没有Flat明显，主要差异体现在镜面反射上。在细分程度较小时，Gourand的镜面反射光棱角会更为明显。

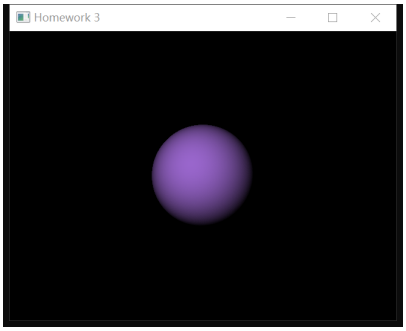
- 比较其他绘制参数变化时，绘制结果的变化

  这里均以phong着色模型为例。

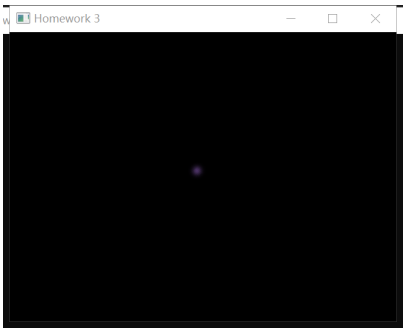
  - 不同反射类型的光照强度
    - 仅有环境光时（反射系数0.3）



    - 仅有漫反射光时（反射系数1）

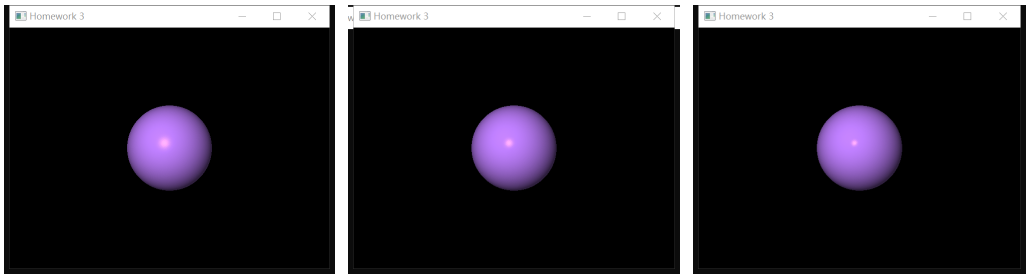


- 仅有镜面反射光时（反射系数0.5）



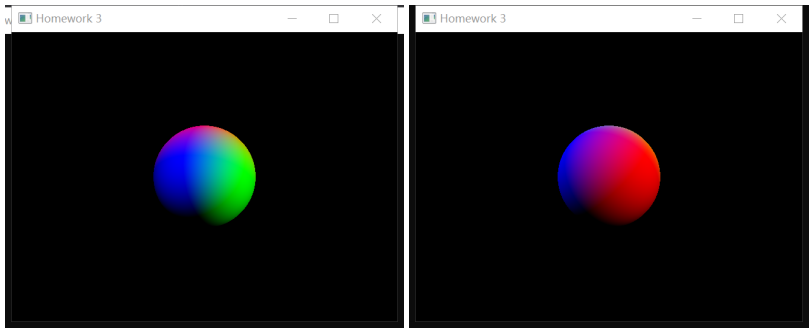
可以知道，三个反射光分量的效果分别如上所示，当调整不同反射类型的光照强度时，对应位置和效果发生变化。

- 镜面反射中的指数α(32,64,128的结果)



随着α的增加，镜面反射，即图中最亮的白点的范围逐渐缩小，白点的中心逐渐变量，周围扩散的比例减小。

- 附加：根据顶点在三维空间中的位置指明颜色



4 实验感想

本次实验主要是通过GLSL实现着色器，进而实现着色模型。光照原理和着色模型的原理虽然较好理解，但是在openGL中通过不熟悉的GLSL实现，确实是有一些困难。在研究flat和guorand的时候，一开始尝试PPT上写的glShadeMode(GL_FLAT)和glShadeMode(GL_SMOOTH)，却发现画出来的图像没有区别，后来搜索发现，glShadeMode()在采用着色器+VBO的情况之下是失效的，此时应该通过添加flat的关键字去实现glShadeMode(GL_FLAT)的功能。