实验 四　　基于着色器的纹理映射和光照渲染

一、 实验内容

在实验三的基础上，增加如下功能：

- ✓ 绘制一个由细分四面体得到的三维镂垫（或三维迷宫）及其适当的场景
- ✓ 在场景中至少添加三个光源：一个平行光、一个点光源、一个聚光灯；
- ✓ 给三维镂垫表面（或三维迷宫）添加适当的纹理；
- ✓ 可通过鼠标或键盘的按键（包括特殊健、组合健等）、菜单、滑动条等工具来实现光源、材质属性参数的设置，也包括对光源位置的控制等。

二、 实验环境

Window8 操作系统、VS2013、OpenGL 库、Intel HD Graphics 4000 图形卡

注：运行的话可能需要用 VS2013，并且配置一下 GLAUX 库

三、 实验方案与程序设计

✧ 目标任务

1.绘制梵高画廊（贴各种不同的梵高画到迷宫墙壁）
2.顶点法向量的求解，光照模型的实现
3.纹理、光源、材质三者颜色值和 alpha 值的混合

✧ 所需知识

纹理坐标的映射/顶点法向量的定义/基本光照模型/材质表面参数

✧ 基本说明

1. 在本次实验中，载入多个纹理后，在显示回调函数中通过控制 glBindVertexArray()、glDrawArray()和 glBindTexture()来实现不同顶点集和不同纹理的对应关系，从而完成梵高画廊。

2.顶点法向量的定义是所有包含该顶点的面的法向量之和，因此在本次实验中，考虑到实现的难易度，采用面法向量来近似顶点法向量，求解出来的光照效果符合现实。

3.三种光源的实现，前提是必须先实现散射光、环境光和镜面光，然后再根据不同光源的特性，考虑距离、方向或者是范围，与材质、纹理混合后，得到最终的效果。

✧ 设计方法

采用面向对象的设计思路，定义 Camera 类、Maze 类、Texture 类以及 Light、Material 结构体，通过 Main 函数进行迷宫显示、碰撞检测、摄像机控制和各种光照、材质的交互操作，具体的光源控制在着色器中通过修改内置变量值来实现。Camera 类和 Maze 类如实验三，新增加的类和结构体如下：

纹理类：

```
class GLTexture{
public:
    unsigned int texture[1];
    //使用纹理
    void Use();
    void BuildColorTexture(unsigned char r, unsigned char g, unsigned char b);
    //载入纹理
    void LoadBMP(char *name);
    GLTexture();
};
```

光照结构体：

```
struct gl_LightParameters {
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec4 position;
    vec4 halfVector;
    vec3 spotDirection;
    float spotExponent;
    float spotCutoff;
    float spotCosCutoff;
    float constantAttenuation;
    float linearAttenuation;
    float quadraticAttenuation;
}Light;
```

材质结构体：

```
struct gl_MaterialParameters {
    vec4 emission;
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    float* shininess;
}Material;
```

✧ 基本步骤

在实验三已实现的迷宫和相机漫游的基础上，增加纹理坐标数组 texcoords，在递归生成迷宫顶点坐标时，同时生成与该顶点对应的纹理坐标存入 texcoords，然后通过 VAO、glDrawArray 方式传递给 GPU 进行纹理映射。光照和纹理可以类似固定流水线那样进行定义，但是采用着色器将丢失处理步骤，因此需要在着色器通过内置结构体 glLightSource、glFrontMaterial 和 glBackMaterial 中访问 OpenGL 设置的参数，从而计算距离、夹角实现光照模型。应该注意的是，并不通过传递 uniform 变量的形式来改变光源，而是通过设置光源结构体来设置光源，举个例子说，当 glLightSource.position.w = 0 时，

表示方向光; glLightSource.position.w = 1&&glLightSource.spotDirection = 0 时，表示点光源；glLightSource.position.w = 1&&glLightSource.spotDirection != 0 时，表现聚光灯。

✧ 模块说明

纹理映射

在片元着色器定义 `uniform sampler2D tex`，这个变量 tex 包含我们将会使用的纹理单元，通过 texture2D 函数我们可以得到一个纹素（texel），这是一个纹理图片中的像素。函数参数分别为 simpler2D 以及纹理坐标：texel = texture2D(texture,texCoord)

组合纹理与片断

OpenGL 允许我们通过多种方式将纹理颜色和片断颜色联合到一起。RGBA 可用的联合方式：

GL_REPLACE	C = Ct	A = At
GL_MODULATE	C = Ct*Cf	A = At*Af
GL_DECAL	C = Cf * (1 - At) + Ct * At	A = Af

表中 Ct 和 At 表示纹理的颜色和 alpha 值，Cf 和 Af 表示片断（fragment）的颜色和 alpha 值，C 和 A 表示最终的颜色和 alpha 值。本次实验采用 GL_MODULATE 方式进行混合。

```
//载入纹理
void CMaze::TextureInit(){
  texture[0].LoadBMP("Data/20.bmp");
  texture[1].LoadBMP("Data/12.bmp");
  texture[2].LoadBMP("Data/37.bmp");
  texture[3].LoadBMP("Data/1.bmp");
  texture[4].LoadBMP("Data/2.bmp");
  texture[5].LoadBMP("Data/3.bmp");
  texture[6].LoadBMP("Data/4.bmp");
  texture[7].LoadBMP("Data/5.bmp");
  texture[8].LoadBMP("Data/6.bmp");
  texture[9].LoadBMP("Data/7.bmp");
  glActiveTexture(GL_TEXTURE0);
}

//使用纹理
void GLTexture::Use()
{
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D, texture[0]);
}

//将墙壁顶点和纹理坐标相关联
void CMaze::Dw(int x, int y, int p) {
  int w = p;
```

```
  CloseIt(x, y);
  do{
    int x2 = 0;
    int y2 = 0;
    if (Neighbor(x, y, w, &x2, &y2)){
        if (OnOpen(x2, y2)) {
            Dw(x2, y2, (w + 3) % 4);
        }
        else {
            if ((w + 1) % 4 == p)
            {
                return;
            }
        }
    }
    else {
        float fx;
        float fy;
        if (Diagnal(x, y, w, &x2, &y2) && OnOpen(x2, y2)) {
            Dw(x2, y2, (w + 2) % 4);
        }
        texcoordX = (texcoordX<0.5) ? 1.0f : 0.0f;
        fx = (float)x + ((w == 1 || w == 2) ? 1.0f : 0.0f);
        fy = (float)y + ((w == 0 || w == 1) ? 1.0f : 0.0f);
        wall_texcoords[NumVerticesW] = vec2(texcoordX, 0.0f);
        wall_points[NumVerticesW++] = vec4(fx, fy, 0.0f, 1.0f);
        wall_texcoords[NumVerticesW] = vec2(texcoordX, 1.0f);
        wall_points[NumVerticesW++] = vec4(fx, fy, 1.0f, 1.0f);
    }
    w++; w %= 4;
  } while (w != p);
  return;
}

//将地板顶点和纹理坐标相关联
void CMaze::DrawSkyGround() {
  for (int y = 0; y < MAZE_HEIGHT; y++) {
    for (int x = 0; x < MAZE_WIDTH; x++) {
        if (!Wall(x, y)) {
            ground_texcoords[NumVerticesG] = vec2(0.0f, 0.0f);
            ground_normals[NumVerticesG] = vec3(0.0f, 0.0f, 1.0f);
            ground_points[NumVerticesG++] = vec4(x + 1.0f, y + 0.0f, 0.0f, 1.0f);

            ground_texcoords[NumVerticesG] = vec2(0.0f, 1.0f);
            ground_normals[NumVerticesG] = vec3(0.0f, 0.0f, 1.0f);
```

```
            ground_points[NumVerticesG++] = vec4(x + 1.0f, y + 1.0f, 0.0f, 1.0f);

            ground_texcoords[NumVerticesG] = vec2(1.0f, 0.0f);
            ground_normals[NumVerticesG] = vec3(0.0f, 0.0f, 1.0f);
            ground_points[NumVerticesG++] = vec4(x + 0.0f, y + 0.0f, 0.0f, 1.0f);

            ground_texcoords[NumVerticesG] = vec2(1.0f, 1.0f);
            ground_normals[NumVerticesG] = vec3(0.0f, 0.0f, 1.0f);
            ground_points[NumVerticesG++] = vec4(x + 0.0f, y + 1.0f, 0.0f, 1.0f);
        }
    }
  }
}

//梵高画廊的绘制
void CMaze::Maze3D(){
  glBindVertexArray(vao[0]);
  GLint PerNum = NumVerticesW / 10 + 3;
  for (int i = 0; i < 10; i++){
      texture[i].Use();
      glDrawArrays(GL_TRIANGLE_STRIP, i*PerNum, (i + 1)*PerNum);
  }
  texture[0].Use();
  glBindVertexArray(vao[1]);
  glDrawArrays(GL_TRIANGLE_STRIP, 0, NumVerticesG);
}
```

光照和材质

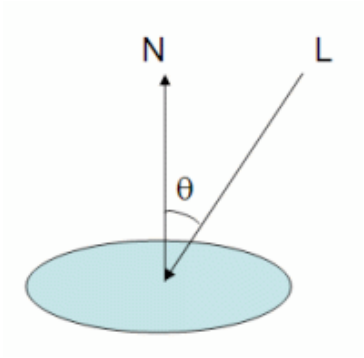
- 光照模型的实现，首先要计算出各顶点的法向量，而顶点法向量计算较为复杂，因此用面法向量来近似顶点法向量，将生成三角形面的顶点进行叉乘，此时需注意顶点相减的顺序会影响法向量是朝正面还是反面。具体求解方式如下：

```
// 墙壁面法向量的计算
for (int i = 0; i < NumVerticesW; i++){
    if ((i + 2) != NumVerticesW){
      Temp_Normals[NumFaces++] = cross(wall_points[i + 1] - wall_points[i],
          wall_points[i + 2] - wall_points[i + 1]);
    }
}
for (int i = 0; i < NumFaces; i++){
  wall_normals[3 * i] = Temp_Normals[i];
  wall_normals[3 * i + 1] = Temp_Normals[i];
  wall_normals[3 * i + 2] = Temp_Normals[i];
}
```

- 光照模型的实现，除了计算法向量外，还要注意散射光、环境光和镜面光的计算，适

用于三种光源。具体的理论如下：

散射光的计算：



$$I_o = L_d * M_d * \cos(\theta)$$

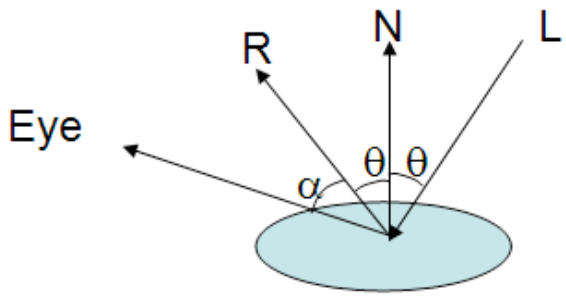
L：光源的散射光分量
M：材质的散射光属性

环境光的计算：

$$I_a = G_a * M_a + L_a * M_a$$

G：全局环境光
L：光照环境光
M：材质

镜面光的计算：

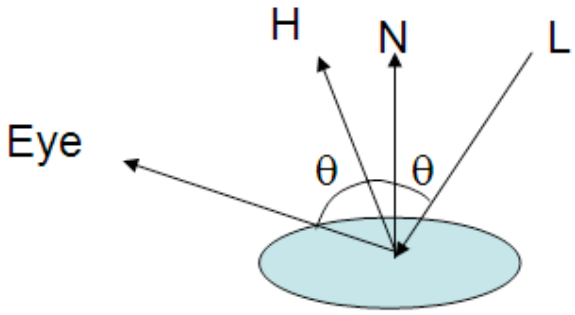


L 是入射光，R 是反射光,N 是法向量,Eye 是眼坐标。光照的强度跟 Eye 和 R 的夹角的余弦成比例，即 alpha。如果 Eye 和 R 重合，光照最强，Eye 不断远离 R，光照不断衰退，衰退的速率跟一个参数有关，即 shininess。shininess 的取值范围是 0-128。

镜面成分的计算公式：

$$Spec = (R \cdot Eye)^s * L_s * M_s$$

计算镜面光照还有一种更简单快速的方式，如下图所示：



向量 H 是 L 和 Eye 的中间位置，在这种方式下，光照强度跟 H 和 N 的夹角余弦成比例。

H（可直接得到）的计算公式是：

$$H = Eye - L$$

镜面成分的计算公式：

$$Spec = (N \cdot H)^s * L_s * M_s$$

- 光照模型的实现，还要注意各类型光源的特性，包括距离、方向和范围的计算，下面列出点光照和聚光灯的理论基础：

点光照

在定向光源中，对于每个顶点，光的方向是一致的，但是对于点光源却不是这样。因此在顶点处理器中需要计算光的方向。点光源还需要考虑光的衰减，光源的贡献要乘以衰减系数，衰减系数的方程是：

$$att = \frac{1}{k_0 + k_1 d + k_2 d^2}$$

其中 d 是光源到顶点的距离，k0,k1,k2 是可以设置的。因为衰减不是线性变化的，因此在片元处理器里不能用插值的方式来计算衰减系数。正确的方法是在顶点处理器里计算距离 d,在片元处理器里根据 d 的插值来计算衰减系数。因此在片元处理器中计算颜色的公式是：

$$color = ambientGlobal + att(ambient + diffuse + specular)$$

由上式可以看出环境光被分为了全局环境光和光源的环境光分别处理，其中全局环境光

是不需要进行衰减处理的。值得注意的是，可以参考最前面光源的数据结构中的 constantAttenuation 、linearAttenuation 、quadraticAttenuation 分别代表了前面公式中的 k0、k1、k2

聚光灯

聚光灯是特殊的点光源，聚光灯是有方向的 *spotDirection* 用来得到方向即圆锥体的轴心。聚光灯还有一个椎体顶角，即圆锥体轴线和母线之间的夹角，可以用 *spotCosCutoff* 得到这个夹角的余弦值。光强会随着远离轴线而衰减，可以用 *spotExponent* 得到这个衰减值。

需要注意的是散射光、镜面光和环境光必须在圆锥体内才会起作用。而全局环境光不受影响，实现方法是取得光源和顶点连线，得到这个连线和圆锥体的轴线之间的夹角的余弦值，这个余弦值必须大于 *spotCosCutoff*。

- OpenGL 设置和着色器的具体实现如下：

光照材质初始化

```
void InitLights(){
   //光照结构体初始化
  Light.position = vec4(8.0, 2.0, 2.0, 1.0);
  Light.diffuse = vec4(1.0, 1.0, 1.0, 1.0);
  Light.ambient = vec4(0.2, 0.2, 0.2, 1.0);
  Light.specular = vec4(0.0, 0.0, 1.0, 1.0);
  Light.spotDirection = vec3(-1.0f, -1.0f, 0.0f);
  Light.spotExponent = 8.0f;
  Light.spotCutoff = 60.0f;
  Light.constantAttenuation = 0.001f;
  Light.linearAttenuation = 0.001f;
  Light.quadraticAttenuation = 0.001f;
 //材质结构体初始化
  Material.shininess = new float[]{ 100 };
  Material.diffuse = vec4(1.0, 1.0, 1.0, 1.0);
  Material.ambient = vec4(0.0, 1.0, 1.0, 1.0);
  Material.specular = vec4(1.0, 1.0, 0.0, 1.0);
 //设置光的成分
  glLightfv(GL_LIGHT0, GL_SPECULAR, Light.specular);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, Light.diffuse);
  glLightfv(GL_LIGHT0, GL_AMBIENT, Light.ambient);
 //设置光的位置或方向
  glLightfv(GL_LIGHT0, GL_POSITION, Light.position);
  glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, Light.constantAttenuation);
  glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, Light.linearAttenuation);
  glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, Light.quadraticAttenuation);
 //设置光的聚光特性
  glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, Light. spotDirection);
```

```
  glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, Light.spotExponent);
  glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, Light.spotCutoff);
 //设置正反面材质
  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, Material.diffuse);
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, Material.ambient);
  glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, Material.specular);
  glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, Material.shininess);
}
```

顶点着色器

```
in vec3 vNormal;
in vec4 vPosition;
in vec2 vTexCoord;
out float mode;
out float dist;
out vec2 texCoord;
out vec3 lightDir,normal,halfVector;
out vec4 diffuse,ambient,ambientGlobal;

void main()
{
   texCoord = vTexCoord;
   gl_Position = gl_ModelViewProjectionMatrix * vPosition;
   normal = normalize(gl_NormalMatrix * vNormal);
   halfVector = normalize(gl_LightSource[0].halfVector.xyz);
   diffuse =    gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
   ambient =    gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
   ambientGlobal = gl_LightModel.ambient * gl_FrontMaterial.ambient;
   //方向光
   if(gl_LightSource[0].position.w ==0){
         lightDir = normalize(vec3(gl_LightSource[0].position));
         mode=1;
   }
   //点光源
   else if(gl_LightSource[0].spotDirection.x==0 && gl_LightSource[0].spotDirection.y==0
    &&gl_LightSource[0].spotDirection.z==0){
         vec3 dic = vec3(gl_LightSource[0].position-gl_Position);
         lightDir = normalize(dic);
         dist = length(dic);
         mode=2;
    }
   //聚光灯
    else{
          vec3 dic = vec3(gl_LightSource[0].position-gl_Position);
          lightDir = normalize(dic);
```

```
            dist = length(dic);
            mode=3;
        }
}
```

片元着色器

```
in float mode;
in float dist;
in vec2 texCoord;
in vec3 lightDir,normal,halfVector;
in vec4 diffuse,ambient,ambientGlobal;
uniform sampler2D texture;

//函数声明
void DirectionLight();       //方向光
void PointLight();           //点光源
void SpotLight();            //聚光灯
void main()
{
   if(mode==1){
        DirectionLight();
   }
   else if(mode==2){
        PointLight();
   }
   else if(mode==3){
        SpotLight();
    }
}

//方向光
void DirectionLight(){

   vec3 ct,cf;
   vec3 n,halfV;
   vec4 texel;
   float NdotL,NdotHV;
   float at,af;

   texel = texture2D(texture,texCoord);
   ct = texel.rgb;
   at = texel.a;
   cf = ambientGlobal.rgb;
   af = diffuse.a;

   n = normalize(normal);
```

```
    NdotL = max(dot(n,lightDir),0.0);
    if (NdotL > 0.0) {
        cf += NdotL * diffuse.rgb + ambient.rgb;

        halfV = normalize(halfVector);
        NdotHV = max(dot(n,halfV),0.0);
        cf += gl_FrontMaterial.specular.rgb * gl_LightSource[0].specular.rgb *
            pow(NdotHV,gl_FrontMaterial.shininess);
    }
    gl_FragColor = vec4(ct * cf, at * af);
}
//点光源
void PointLight(){

    vec3 ct,cf;
    vec3 n,halfV;
    vec4 texel;
    float NdotL,NdotHV;
    float at,af;
    float att;

    texel = texture2D(texture,texCoord);
    ct = texel.rgb;
    at = texel.a;
    cf = ambientGlobal.rgb;
    af = diffuse.a;

    n = normalize(normal);
    NdotL = max(dot(n,normalize(lightDir)),0.0);
    if (NdotL > 0.0) {
        att = 1.0 / (gl_LightSource[0].constantAttenuation +gl_LightSource[0]
          .linearAttenuation * dist + gl_LightSource[0].quadraticAttenuation * dist * dist);
        cf += att * (NdotL * diffuse.rgb + ambient.rgb);

        halfV = normalize(halfVector);
        NdotHV = max(dot(n,halfV),0.0);
        cf += att * gl_FrontMaterial.specular.rgb * gl_LightSource[0].specular.rgb *
            pow(NdotHV,gl_FrontMaterial.shininess);
    }
    gl_FragColor = vec4(ct * cf, at * af);
}
```

```
//聚光灯
void SpotLight(){

    vec3 ct,cf;
    vec3 n,halfV;
    vec4 texel;
    float NdotL,NdotHV;
    float at,af;
    float att,spotEffect;

    texel = texture2D(texture,texCoord);
    ct = texel.rgb;
    at = texel.a;
    cf = ambientGlobal.rgb;
    af = diffuse.a;

    n = normalize(normal);
    NdotL = max(dot(n,normalize(lightDir)),0.0);
    if (NdotL > 0.0){
       spotEffect = dot(normalize(gl_LightSource[0].spotDirection), normalize(-lightDir));
       if(spotEffect > gl_LightSource[0].spotCosCutoff)
       {
          spotEffect = pow(spotEffect, gl_LightSource[0].spotExponent);
          att = spotEffect / (gl_LightSource[0].constantAttenuation +
               gl_LightSource[0].linearAttenuation * dist +
               gl_LightSource[0].quadraticAttenuation * dist * dist);
          cf += att * (NdotL * diffuse.rgb + ambient.rgb);

            halfV = normalize(halfVector);
            NdotHV = max(dot(n,halfV),0.0);
            cf += att * gl_FrontMaterial.specular.rgb *
            gl_LightSource[0].specular.rgb *pow(NdotHV,gl_FrontMaterial.shininess);
        }
     }
     gl_FragColor = vec4(ct * cf, at * af);
}
```

交互操作

除了实验三定义的交互方式外，在本次实验中增设了光源类型选择、光源位置和光照方向设置。光源类型选择主要是通过设置 Light.position 和 Light.spotDirection 来进行控制。当处于点光源和聚光灯模式时，1 到 6 数字键可以设置光源位置；当处于方向光模式时，1 到 6 数字键可以设置光源方向。具体实现如下：

```
//键盘回调
void SpecialKeys(int key, int x, int y)
{
 // 是否按下 UP 箭头键
 if (key == GLUT_KEY_UP) {
     // 移动摄像机
     camera.MoveCamera(kSpeed);
 }
 // 是否按下 DOWN 键
 if (key == GLUT_KEY_DOWN) {
     // 移动摄像机
     camera.MoveCamera(-kSpeed);
 }
 // 是否按下 LEFT 键
 if (key == GLUT_KEY_LEFT) {
     // 旋转摄像机
     camera.RotateCamera(-kAngle);
 }
 // 是否按下 RIGHT 键
 if (key == GLUT_KEY_RIGHT) {
     // 旋转摄像机
     camera.RotateCamera(kAngle);
 }
 if (key == GLUT_KEY_ALT_L) {
     // 3D
     Is3D = true;
 }
 if (key == GLUT_KEY_ALT_R) {
     // 2D
     Is3D = false;
 }
 if (key == GLUT_KEY_PAGE_UP) {
     // 加速
     kSpeed *= 2.0f;
 }
 if (key == GLUT_KEY_PAGE_DOWN) {
     // 减速
     kSpeed /= 2.0f;
```

```
  }
  if (key == GLUT_KEY_F1) {
      // 平行光
      Light.position[3] = 0.0f;
  }
  if (key == GLUT_KEY_F2) {
      // 点光源
      Light.position[3] = 1.0f;
      Light.spotDirection = 0.0f;
  }
  if (key == GLUT_KEY_F3) {
      // 聚光灯
      Light.position[3] = 1.0f;
      Light.spotDirection = -1.0f;
  }
  glutPostRedisplay();
}
//光照位置/方向
void Keyboard(unsigned char key, int x, int y)
{
  switch (key)
  {
  case '1':
      Light.position[0] += 3.0f;
      break;
  case '2':
      Light.position[0] -= 3.0f;
      break;
  case '3':
      Light.position[1] += 3.0f;
      break;
  case '4':
      Light.position[1] -= 3.0f;
      break;
  case '5':
      Light.position[2] += 0.5f;
      break;
  case '6':
      Light.position[2] -= 0.5f;
      break;
  default:break;
  }
  glutPostRedisplay();
}
```

四、 实验结果与分析（包括功能说明、操作说明）

- ✧ 按住键盘方向键可实现相机的前进、后退、左看和右看。
- ✧ 移动鼠标可从八个方向实现相机视线的调整，包括左看、右看、上看和下看。
- ✧ 点击键盘 F1 键选择方向光模式，F2 键选择点光源模式，F3 选择聚光灯模式。
- ✧ 点击键盘 1 到 6 键可以更新光源的位置或者光照的方向（XYZ 轴）。
- ✧ 点击键盘的 PageUp、PageDown 可以进行漫游加减速。
- ✧ 点击键盘的 ALT+L 选择三维显示，ALT+R 选择二维显示。

梵高画廊如下截图：

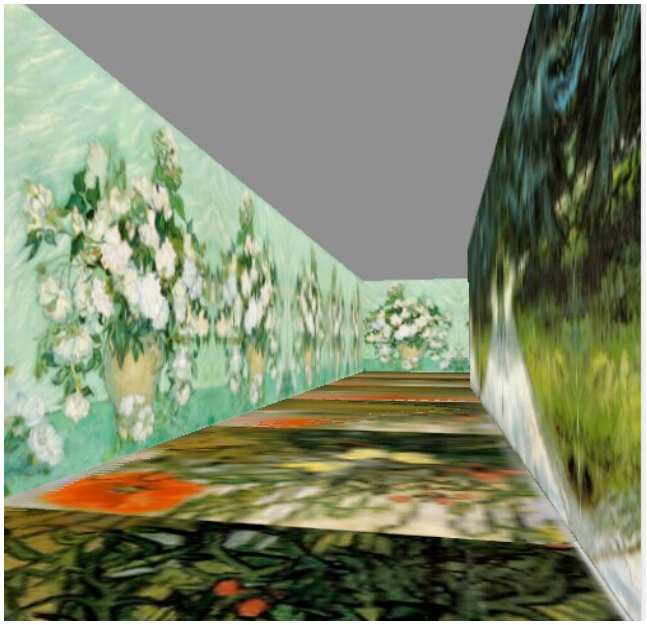


图 1-未设置光源

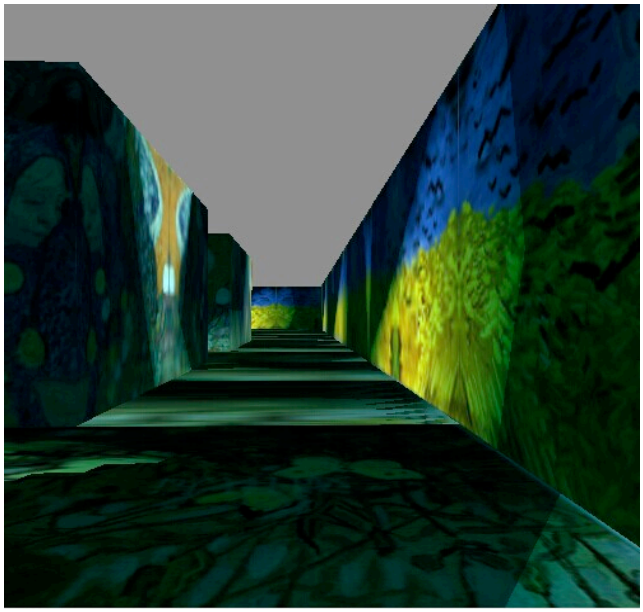


图 2-方向光

不管相机如何移动，墙壁跟地板的光照图案和阴影都不会发生变化

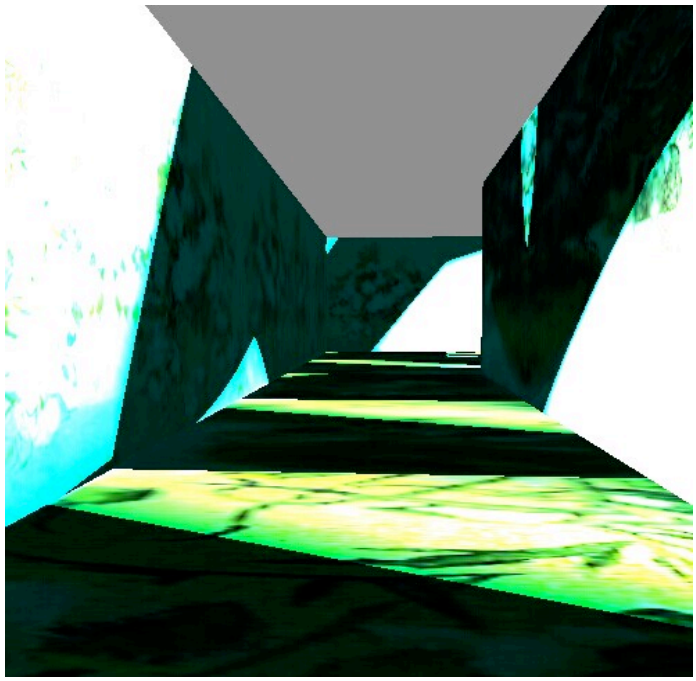


图 3-点光源（1）

相机在远处观察墙壁和地板

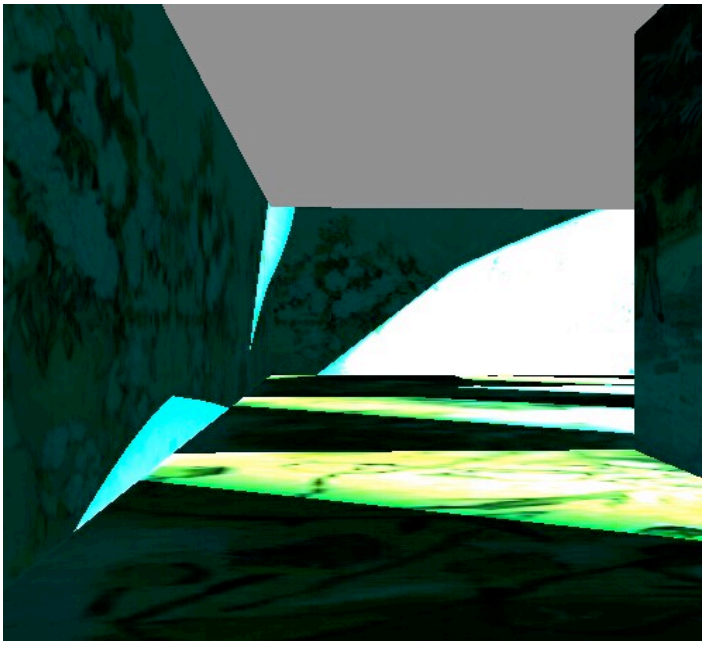


图 4-点光源（2）

相机在近处观察，墙上和地板的图案随着距离靠近微小变化

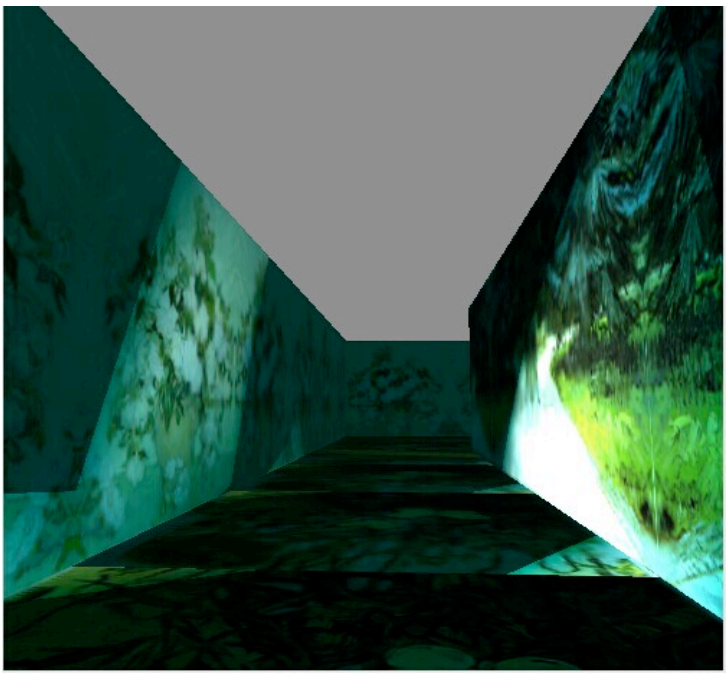


图 5-聚光灯（1）

相机在远处观察墙壁和地板，远处无光照图案

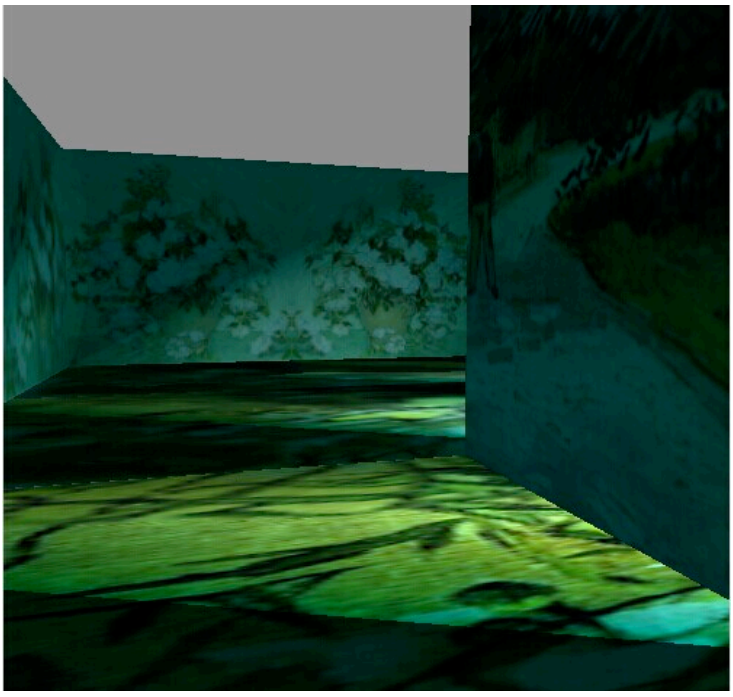


图 6-聚光灯（2）

相机在近处观察，墙上和地板出现光照图案

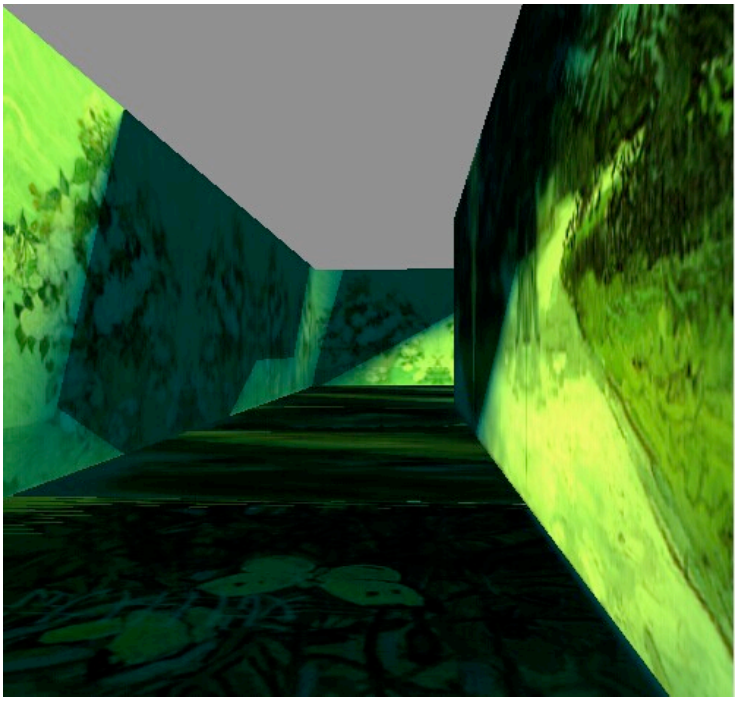


图 7-修改光的颜色属性

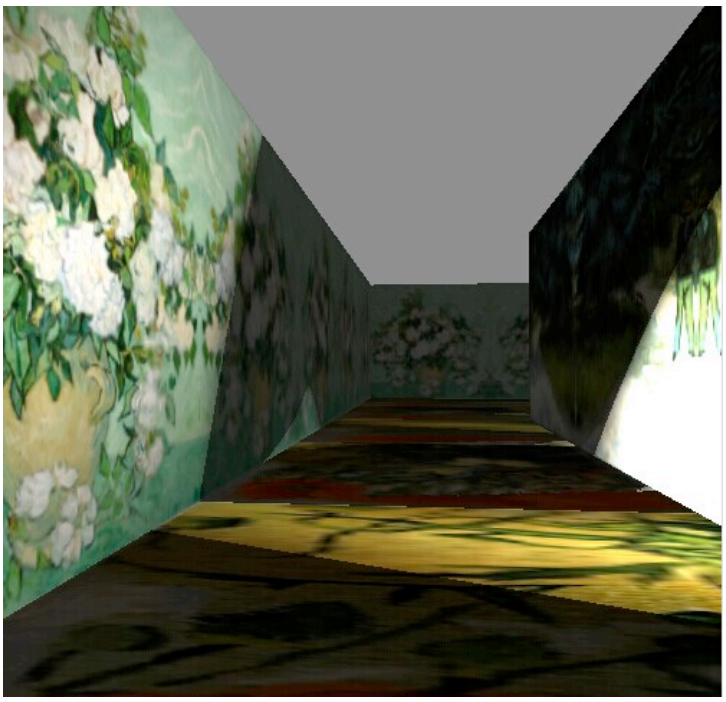


图 8-修改材质属性

五、 实验心得（自我评价）与建议、

- ✧ 在这次实验中，尽管积累了许多实验三取得的经验，但还是碰到了比较多的问题，用 GPU 实现光照模型计算比起 CPU 直接设置参数难度来得大。
- ✧ 在这次实验中，学会了如何设置纹理坐标，按照传递参数的方法同样可以将纹理数据传递给 GPU，通过对顶点获取到的纹理数据进行插值，产生像素纹

理坐标，经由片元着色器进行像素操作。

- ✧ 在这次实验中，顶点法向量的计算较为复杂。一开始的时候采用 GLSL 内置变量 GL_Normal 进行设置，但不管我怎么设置都没有实际光照效果。后来发现是因为没有在 OpenGL 中设置顶点法向量。查阅资料后发现要自己计算顶点法向量，对于迷宫而言，顶点众多，计算起来较为困难，因此我采用面法向量近似顶点法向量，取得的实际效果也不错。
- ✧ 在这次实验中，关于如何控制各光源的显示，我认为可以模仿固定流水线的方式进行处理，即通过 position 和 spotDirection 这两个变量进行选择类型。而不是说定义 Uniform 变量，用 OpenGL 去设置该变量，然后着色器有选择地进行相应类型的光照模型计算。
- ✧ 在这次实验中，对数值计算中的插值方法有了进一步的认识，即只能处理线性变量，对于非线性变量从顶点着色器传递到片元着色器，需要通过一定的处理方式转化成线性变量后再在片元部分进行计算。
- ✧ 在这次实验中，除了实现光照模块、纹理模块外，还有一个很重要的是如何将光照、材质和纹理进行混合，这点处理不好的话严重影响整体的效果。经过查阅大量资料，发现 GLSL 支持三种像素混合方式，我们可以直接处理帧缓存中的颜色缓存。
- ✧ 在这次实验中，得到了实验结果后，我对自己得到的光照结果也有一些疑问，后面也复习了不少物理知识，才敢确信自己的结论。这主要是因为平时没有好好观察过点光源和聚光灯可能会产生的现象引起。