计算机图形学课程作业

Assignment3

Lighting and Texturing

姓 名: 李 昊 伟
学 号: 20337056
班 号: 20 级计科一班

计算机图形学
(秋季, 2022)

中山大学
计算机学院
SYSUCSE

2022 年 11 月 9 日

目 录

1 概述

本次作业报告主要是对 4 个 task 的完成过程的介绍以及知识点的解析。作业程序代码可访问本人 github 代码仓库：https://github.com/distancemay5/learn_cg。实现效果以视频的形式保存在作业提交的压缩包。

2 作业任务解答

2.1 Task1+Task4 最邻近采样和双线性过滤采样

2.1.1 问题分析

一般来说，纹理所指的对象是图片，一张图片就是一张纹理；贴图指的是映射关系，即“如何将纹理像素映射到 uv 坐标上；材质描述了渲染所需的数据集合，通常可以包括基础颜色、镜面反射颜色、自发光颜色、光泽度等数值参数，以及多张贴图和要贴的纹理，还有渲染时所用的 shader 程序。

这里将纹理像素映射到 uv 坐标上已经被框架代码实现好了，不需要我们思考三维模型上的点与纹理坐标的点。也就是说，纹理坐标是给定的。已经得到了 uv 坐标，我们只需要把它读出来，决定这个 uv 坐标对应的颜色就行了。

首先第一步，就是将纹理坐标 uv 从 $[0,1]\times[0,1]$ 映射到 $[0,width-1]\times[0,height-1]$。这个过程很简单，只需要对 (u , v) 做一个简单的仿射变换就可以了。

可是，当我们这样得到坐标值的时候。这个值可能不是一个”像素”值，也就是不是一个整数。那么该如何确定这个值取哪一个像素的值呢?

第一种思路就是把这个值四舍五入，得到的事实上是与这个浮点坐标最近的整数”像素“值。这就是最近邻采样。另外一种思路就是采用双线性插值，这就是双线性过滤采样。

所谓双线性过滤采样，其实也非常好理解。需要考虑的点有 a: (u 下取整, v 下取整) ，b: (u 下取整，v 上取整) ，c: (u 上取整，v 下取整) ，d: (u 上取整，v 上取整) 。我们先以 a,b 为一组，c,d 为一组，以 v 上取整和 v 下取整同 v 的距离为权重，可以线性插值得到两个点 tmp1 , tmp2。这两个点 v 值相同。再将这两个临时点以 u 上取整和 u 下取整同 u 的距离为权重，插值得到最终点。这就是双线性插值过滤。

2.1.2 代码实现

```
glm::vec4 TRTexture2DSampler::textureSampling_nearest(const TRTexture2D
    &texture, glm::vec2 uv)
{
  unsigned char r = 255, g = 255, b = 255, a = 255;
  {
    float u = uv.x;
    float v = uv.y;
    int width = texture.getWidth();
    int height = texture.getHeight();
    int x = (int)(u * width - 1.0f);
    int y = (int)(v * height - 1.0f);
    texture.readPixel(x, y, r, g, b, a);
  }

  constexpr float denom = 1.0f / 255.0f;
  return glm::vec4(r, g, b, a) * denom;
}


glm::vec4 TRTexture2DSampler::textureSampling_nearest(const TRTexture2D
    &texture, glm::vec2 uv)
{
  unsigned char r = 255, g = 255, b = 255, a = 255;
  {
    float u = uv.x;
    float v = uv.y;
    int width = texture.getWidth();
    int height = texture.getHeight();
    int x = (int)(u * width - 1.0f);
    int y = (int)(v * height - 1.0f);
    texture.readPixel(x, y, r, g, b, a);
  }

  constexpr float denom = 1.0f / 255.0f;
  return glm::vec4(r, g, b, a) * denom;
}


glm::vec4 TRTexture2DSampler::textureSampling_bilinear(const TRTexture2D
    &texture, glm::vec2 uv)
{
  //Note: Delete this line when you try to implement Task 4.
  unsigned char r_0 = 255, g_0 = 255, b_0 = 255, a_0 = 255;
  unsigned char r_1 = 255, g_1 = 255, b_1 = 255, a_1 = 255;
  unsigned char r_2 = 255, g_2 = 255, b_2 = 255, a_2 = 255;
  unsigned char r_3 = 255, g_3 = 255, b_3 = 255, a_3 = 255;
  float frac1 = 0;
  float frac2 = 0;
  {
    float u = uv.x;
    float v = uv.y;
    int width = texture.getWidth();
```

```
    int height = texture.getHeight();
    int x = (int)(u * width - 1.0f);
    int y = (int)(v * height - 1.0f);
    frac1 = u * width - 1.0f - x;
    frac2 = v * height - 1.0f - y;
    // x = x < 0 ? width + x : x;
    // y = y < 0 ? height + y : y;
    texture.readPixel(x, y, r_0, g_0, b_0, a_0);
    texture.readPixel(x+1, y, r_1, g_1, b_1, a_1);
    texture.readPixel(x, y+1, r_2, g_2, b_2, a_2);
    texture.readPixel(x+1, y+1, r_3, g_3, b_3, a_3);
  }
  glm::vec4 tmp0 = { r_0 * (1 - frac1) + r_1 * frac1 ,  g_0 * (1 - frac1)
      + g_1 * frac1  , b_0 * (1 - frac1) + b_1 * frac1 , a_0 * (1 -
      frac1) + a_1 * frac1 };
  glm::vec4 tmp1 = { r_2 * (1 - frac1) + r_3 * frac1 ,  g_2 * (1 - frac1)
      + g_3 * frac1  , b_2 * (1 - frac1) + b_3 * frac1 , a_2 * (1 -
      frac1) + a_3 * frac1 };
  glm::vec4 result = { tmp0.x * (1 - frac2) + tmp1.x * frac2 , tmp0.y *
      (1 - frac2) + tmp1.y * frac2  , tmp0.z * (1 - frac2) + tmp1.z *
      frac2  , tmp0.w * (1 - frac2) + tmp1.w * frac2 };
  constexpr float denom = 1.0f / 255.0f;
  return result * denom;
}
```

2.1.3 实现结果

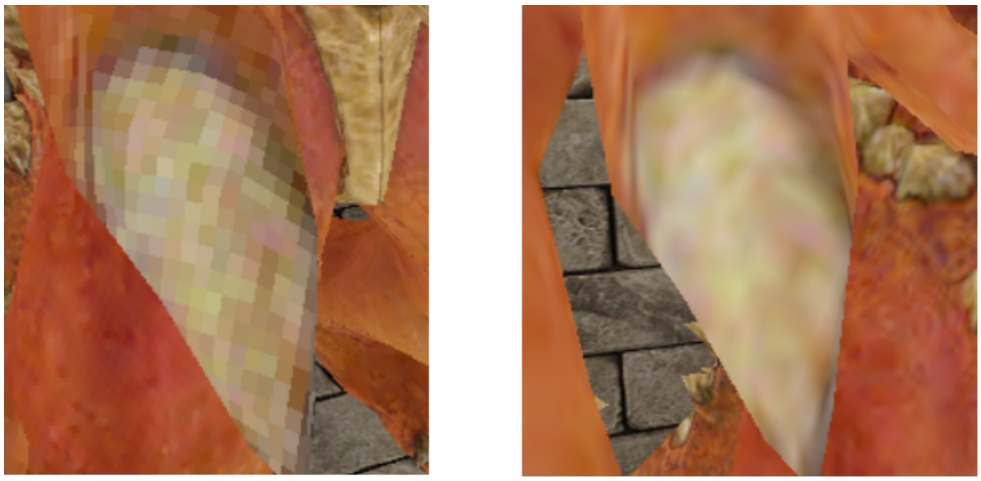


图 1: 最近邻插值与双线性插值过滤对比

可见，最近邻插值可以看出明显的像素块，而双线性插值过滤显得更加平滑自然。

2.2 一般光照模型的模拟思路

在点光源光照模型当中，一般把物体对光线的反射模拟为三个部分。当然物体的亮度也与同点光源的距离有关，我们把它作为一个系数，最后再乘，最后再考虑。

第一部分是环境光照。这个值对于任何一个物体来说都是平等的。是由物体本身对于光的吸收情况 (也就是物体本身的 RGB 值) 和光线颜色 (光的 RGB 值) 共同完全决定的 (直接相乘)。

第二部分是漫反射。这个值与光线对平面的入射角有关，但是向四面八方反射是各向同性的，也就是完全一样的。这个值由 Lambert’s Cosine Law 计算。直观上来看，这个规律描述的是这样一个现象：当入射光线与平面法线越接近，漫反射强度越大。但是从四面八方观察到的漫反射强度，是一样的。

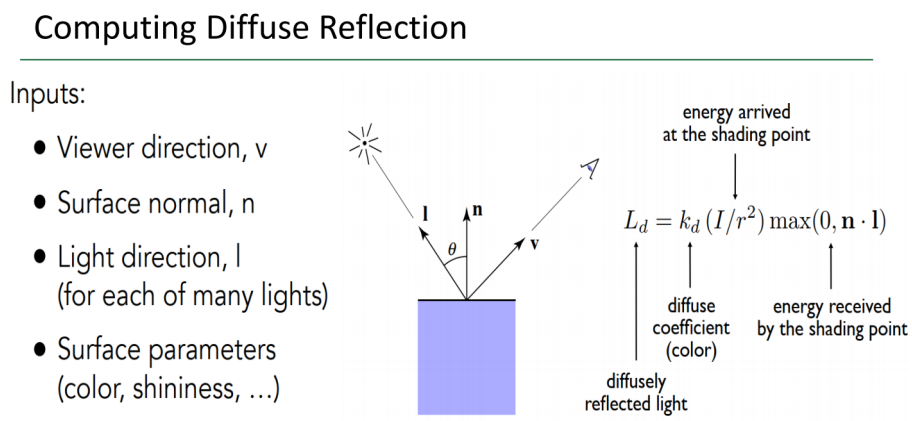


图 2: 兰伯特余弦定理

第三部分是镜面反射，这个值不仅与光线对平面的入射角有关，还与观察的角度有关。描述计算它的经典模型有两个，phong 模型和 blinn-phong 模型。在下两部分我将展示其内容与实现以及实现效果。

随着光线传播距离的增长逐渐削减光的强度通常叫做衰减 (Attenuation)。随距离减少光强度的一种方式是使用一个线性方程。这样的方程能够随着距离的增长线性地减少光的强度，从而让远处的物体更暗。然而，这样的线性方程通常会看起来比较假。在现实世界中，灯在近处通常会非常亮，但随着距离的增加光源的亮度一开始会下降非常快，但在远处时剩余的光强度就会下降的非常缓慢了。所以，我们需要一个不同的公式来减少光的

强度。

$$F_{att} = \frac{1.0}{K_c + K_l * d + K_q * d^2}$$

图 3: 衰减系数计算公式

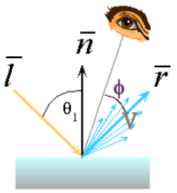
上图的公式加入了常数项和二次项。常数项通常保持为 1.0，它的主要作用是保证分母永远不会比 1 小，否则的话在某些距离上它反而会增加强度，这肯定不是我们想要的效果。由于二次项的存在，光线会在大部分时候以线性的方式衰退，直到距离变得足够大，让二次项超过一次项，光的强度会以更快的速度下降。这样的结果就是，光在近距离时亮度很高，但随着距离变远亮度迅速降低，最后会以更慢的速度减少亮度。

2.3 Task2 Phong 光照模型

2.3.1 问题分析

Phong 模型是比较直观的，也就是说衡量观察角度同反射光线的夹角。如果这个夹角大，观察到的光强就弱，如果这个夹角小，这个光强就比较强。当然，这个反射光强也与物体本身与镜面多么接近有关系。如下图：

$$I_{specular} = k_s I_{light} (\bar{v} \cdot \bar{r})^{n_{shiny}}$$



- V is the unit vector towards the viewer
- r is the ideal reflectance direction
- K_s: specular component
- I_{light}: incoming light intensity
- n_{shiny}: purely empirical constant, varies rate of falloff(材质发光常数，值越大，表面越接近镜面，高光面积越小。)

图 4: phong 光照模型

当然，实现过程中我们需要一个函数快速计算出基于入射角等于反射角得到的反射光线向量。这个函数由 glm 库提供：glm::reflect(normal , lightDir)。其中 normal 是标准法向量，而 lightDir 是入射光向量。

glm::reflect(...)

genType glm::reflect(const genType &I, const genType &N)

Documentation from code comments

For the incident vector I and surface orientation N, returns the reflection direction : result = I - 2.0 * dot(N, I) * N.

Type parameters:

genType Floating-point vector types.

See also: GLSL reflect man page

See also: GLSL 4.20.8 specification, section 8.5 Geometric Functions

图 5: glm::reflect

2.3.2 代码实现

```
glm::vec3 r = glm::reflect(normal , lightDir);
float cof = glm::pow(glm::dot(r, viewDir) , m_shininess);
specular = glm::vec3(light.lightColor.x * spe_color.x, light.lightColor.y *
    spe_color.y, light.lightColor.z * spe_color.z) * cof;
```

2.3.3 实现效果

图 6: 从左到右依次是环境光、漫反射光、镜面反射光

综合效果见附件视频。

2.4 Task3 Blinn-Phong 光照模型

2.4.1 问题分析

phong 使用视线与入射光线反射向量的夹角。这样，因为视线与光反射向量夹角大于 90 度时取 0，因此有明显的边缘。blinn-phong 使用 halfway（视线与入射光线的中间方向）与 normal 的夹角，解决了这一问题。

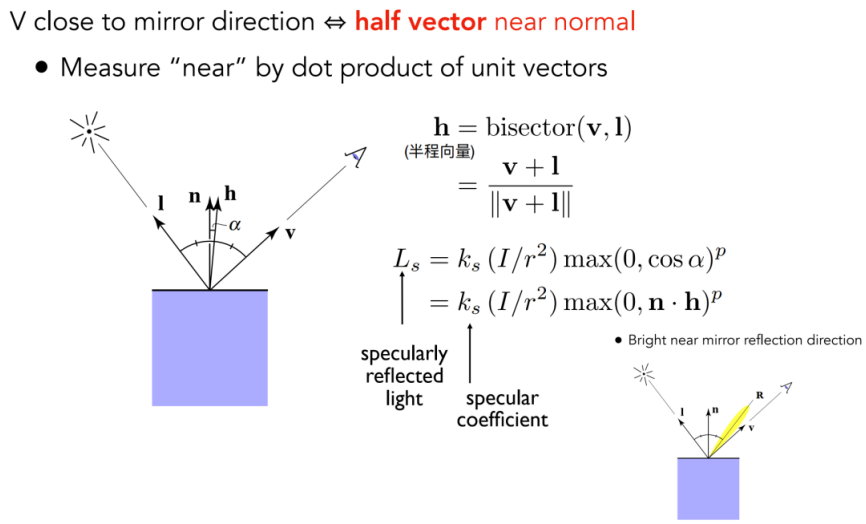


图 7: blinn-phong 光照模型

2.4.2 代码实现

```
glm::vec3 halfwayDir = glm::normalize(lightDir + viewDir);
float cof = glm::pow(glm::max(glm::dot(normal, halfwayDir), 0.0f),
    m_shininess);
specular = glm::vec3(light.lightColor.x * spe_color.x, light.lightColor.y *
    spe_color.y, light.lightColor.z * spe_color.z) * cof;
```

综合实现效果见附件视频。

2.5 Task5 SpotLight 聚光灯光源的实现

2.5.1 问题分析

聚光是位于环境中某个位置的光源，它只朝一个特定方向而不是所有方向照射光线。这样的结果就是只有在聚光方向的特定半径内的物体才会被照亮，其它的物体都会保持黑暗。聚光很好的例子就是路灯或手电筒。

OpenGL 中聚光是用一个世界空间位置、一个方向和一个切光角 (Cutoff Angle) 来表示的，切光角指定了聚光的半径（译注：是圆锥的半径不是距光源距离那个半径）。对于每个片段，我们会计算片段是否位于聚光的切光方向之间（也就是在锥形内），如果是的话，我们就会相应地照亮片段。

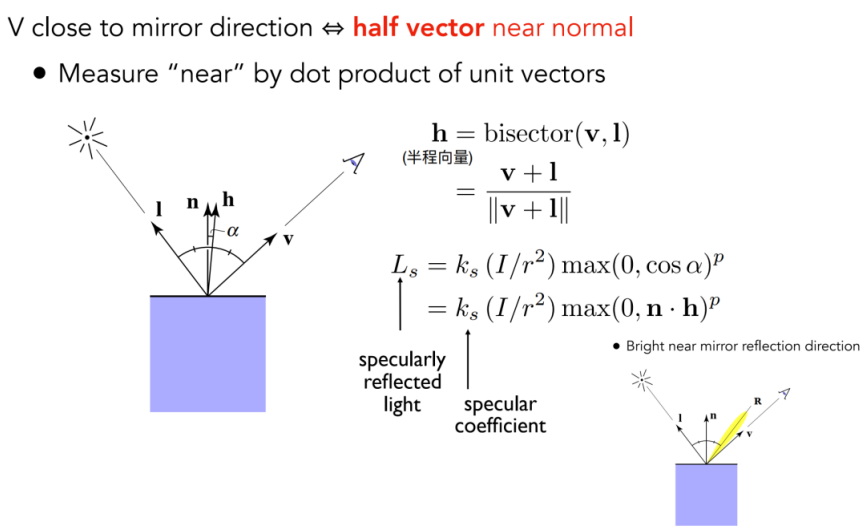


图 8: 聚光灯光源示意图

2.5.2 代码实现

```
  // 定义聚光灯光源
  class TRSpotLight
{
public:
  glm::vec3 lightPos;//Note: world space position of light source
  glm::vec3 attenuation;
  glm::vec3  direction;
  glm::vec3 lightColor;
  float cutOff;
  float outerCutOff;

  TRSpotLight(glm::vec3 pos, glm::vec3 atten, glm::vec3  dir , glm::vec3
      color , float cutOff , float outerCutOff)
    : lightPos(pos), attenuation(atten),direction(dir), lightColor(color)
       , cutOff(cutOff) , outerCutOff(outerCutOff){}
};
  // 设置光源
  float theta = glm::dot(lightDir, normalize(-light.direction));
  float epsilon = light.cutOff - light.outerCutOff;
  float intensity = glm::clamp((theta - light.outerCutOff) / epsilon,
      0.0f, 1.0f);
  if (theta > light.outerCutOff)
  {
fragColor.x += (ambient.x + diffuse.x + specular.x) * attenuation *
    intensity;
fragColor.y += (ambient.y + diffuse.y + specular.y) * attenuation;  *
    intensity;
```

```
        fragColor.z += (ambient.z + diffuse.z + specular.z) * attenuation *
            intensity;
}
```

2.5.3 实现效果

图 9: 聚光灯实现效果