计算机学院实验报告

实验题目： Blinn-Phong		学号：202300130183
日期：2025/4/21	班级：23 级智能班	姓名：宋浩宇

Email：2367651943@qq.com
202300130183@mail.sdu.edu.cn

实验目的：

在这次编程任务中，我们会进一步模拟现代图形技术。我们在代码中添加了 Object Loader(用于加载三维模型), Vertex Shader 与 Fragment Shader，并且支持了纹理映射。

1. 修改函数 rasterize_triangle(const Triangle& t) in rasterizer.cpp: 在此处实现与作业 2 类似的插值算法，实现法向量、颜色、纹理颜色的插值。
2. 修改函数 get_projection_matrix() in main.cpp: 将你自己在之前的实验中实现的投影矩阵填到此处，此时你可以运行 ./Rasterizer output.png normal 来观察法向量实现结果。
3. 修改函数 phong_fragment_shader() in main.cpp: 实现 Blinn-Phong 模型计算 Fragment Color.
4. 修改函数 texture_fragment_shader() in main.cpp: 在实现 Blinn-Phong 的基础上，将纹理颜色视为公式中的 kd，实现 Texture Shading Fragment Shader.
5. 修改函数 bump_fragment_shader() in main.cpp: 在实现 Blinn-Phong 的基础上，仔细阅读该函数中的注释，实现 Bump mapping.
6. 修改函数 displacement_fragment_shader() in main.cpp: 在实现 Bump mapping 的基础上，实现 displacement mapping.

实验环境介绍：

软件环境：
主系统：Windows 11 家庭中文版 23H2 22631.4317
虚拟机软件：Oracle Virtual Box 7.1.6
虚拟机系统：Ubuntu 18.04.2 LTS
编辑器：Visual Studio Code
编译器：gcc 7.3.0
计算框架：Eigen 3.3.7

硬件环境：
CPU：13th Gen Intel(R) Core(TM) i9-13980HX 2.20 GHz
内存：32.0 GB (31.6 GB 可用)
磁盘驱动器：NVMe WD_BLACKSN850X2000GB
显示适配器：NVIDIA GeForce RTX 4080 Laptop GPU

解决问题的主要思路：
首先是前两条任务，我们根据指引找到 GAMES101 的 github 仓库，copy 来需要的代码。
然后我们需要做：
1. 实现实现 Blinn-Phong 模型计算 Fragment Color.
1. phong 模型：
phong 模型需要计算出每个点的漫反射、高光、环境光，给出计算公式：
环境光：（避免图像全黑）

$$I_{\text{ambient}} = k_a \cdot I_{\text{light\_ambient}}$$

漫反射：

$$I_{\text{diffuse}} = k_d \cdot I_{\text{light}} \cdot \max(0, \mathbf{n} \cdot \mathbf{l})$$

- k_d：材质的漫反射系数（与颜色相关）。
- I_{light}：光源强度。
- $\mathbf{n}$：表面法线向量。
- $\mathbf{l}$：光线方向向量（从表面指向光源）。
- $\max(0, \mathbf{n} \cdot \mathbf{l})$：确保只有光线照射到表面时才有贡献。

高光:

$$I_{\text{specular}} = k_s \cdot I_{\text{light}} \cdot \max(0, \mathbf{r} \cdot \mathbf{v})^p$$

- k_s：材质的镜面反射系数（控制高光强度）。
- $\mathbf{r}$：反射光方向向量（由光线方向和法线计算）。
- $\mathbf{v}$：视线方向向量（从表面指向观察者）。
- p：高光锐度（值越大，高光越集中）。
- $\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l}$：反射光方向公式。

2. 实现 Texture Shading Fragment Shader.
这个模型的特点是着色需要考虑颜色,颜色用于漫反射，要理解这个关系，这个模型比起 phong 模型只需把颜色改为当前像素对应的纹理即可。
payload 模型中记录了纹理的坐标 tex_coords 和纹理 Texture 类，若想获取纹理值，只需把纹理坐标传到获取纹理的函数即可。
3. 实现 Bump mapping.
bump 是一种通过修改表面法线来模拟凹凸效果的技术，而不会实际改变几何体的顶点位置。它主要依赖于法线贴图（normal map），通过切线空间中的法线变化来影响光照计算。
使用灰度(Grayscale)图和简单的光影技巧在对象的表面人为地制造这种质感，而不是真的在其表面扣出一个个的凸起和裂痕。一般来说，凹凸贴图是只有 8 位(8-bit)色的灰度图。也就是说它只有 256 种不同的灰度。凹凸贴图中的值就是告诉三维软件两件事：凹或凸。当值接近 50%灰度时，物体表面几乎不会有什么细节变化。当灰度值变亮(白)，表面细节呈现为凸出，当灰度值变暗(黑)，表面细节呈现为凹入。
4. 实现 displacement mapping.

通过实际位移顶点位置（基于高度图）生成几何细节，再重新计算法线。

实验步骤与实验结果：

先把任务 12 的代码 copy 过来：

投影矩阵：

```
Eigen::Matrix4f get_projection_matrix(float eye_fov, float
aspect_ratio,
                                      float zNear, float zFar)
{
  // Students will implement this function

  Eigen::Matrix4f projection = Eigen::Matrix4f::Identity();
  // TODO: Implement this function
  // Create the projection matrix for the given parameters.
  // Then return it.
  float t = std::tan(eye_fov / 2);
  projection << 1 / (t * aspect_ratio), 0, 0, 0, 0, 1 / t, 0, 0,
0, 0,
      (zNear + zFar) / (zNear - zFar), (2 * zNear * zFar) / (zNear
- zFar), 0,
      0, -1, 0;
  return projection;
}
```

三角形光栅化：

```

// Screen space rasterization
void rst::rasterizer::rasterize_triangle(
    const Triangle& t, const std::array<Eigen::Vector3f, 3>&
view_pos)
{
  // TODO : Find out the bounding box of current triangle.
  float aabb_minx = 0;
```

```
  float aabb_miny = 0;
  float aabb_maxx = 0;
  float aabb_maxy = 0;
  for (size_t i = 0; i < 3; i++)
  {
    const Vector4f& p = t.v[i];
    if (i == 0)
    {
      aabb_minx = aabb_maxx = p.x();
      aabb_miny = aabb_maxy = p.y();
      continue;
    }
    aabb_minx = p.x() < aabb_minx ? p.x() : aabb_minx;
    aabb_miny = p.y() < aabb_miny ? p.y() : aabb_miny;
    aabb_maxx = p.x() > aabb_maxx ? p.x() : aabb_maxx;
    aabb_maxy = p.y() > aabb_maxy ? p.y() : aabb_maxy;
  }
  // iterate through the pixel and find if the current pixel is
inside the
  // triangle
  auto v = t.v;
  for (int x = (int)aabb_minx; x < aabb_maxx; x++)
  {
    for (int y = (int)aabb_miny; y < aabb_maxy; y++)
    {
      if (!insideTriangle(x, y, t.v))
        continue;
      // TODO: Inside your rasterization loop:
      //    * v[i].w() is the vertex view space depth value z.
      //    * Z is interpolated view space depth for the current
pixel
      //    * zp is depth between zNear and zFar, used for z-buffer
      auto [alpha, beta, gamma] = computeBarycentric2D(x, y, t.v);
      float Z = 1.0 / (alpha / v[0].w() + beta / v[1].w() + gamma
/ v[2].w());
      float zp = alpha * v[0].z() / v[0].w() + beta * v[1].z() /
v[1].w() +
                 gamma * v[2].z() / v[2].w();
      zp *= Z;
      int buf_index = get_index(x, y);
      if (zp >= depth_buf[buf_index])
        continue;
      depth_buf[buf_index] = zp;
      // TODO: Interpolate the attributes:
```

```
      // auto interpolated_color
      // auto interpolated_normal
      // auto interpolated_texcoords
      // auto interpolated_shadingcoords
      auto interpolated_color = interpolate(alpha, beta, gamma,
t.color[0],
                                          t.color[1],
t.color[2], 1);
      auto interpolated_normal = interpolate(alpha, beta, gamma,
t.normal[0],
                                           t.normal[1],
t.normal[2], 1);
      auto interpolated_texcoords =
          interpolate(alpha, beta, gamma, t.tex_coords[0],
t.tex_coords[1],
                      t.tex_coords[2], 1);
      auto interpolated_viewpos = interpolate(alpha, beta, gamma,
view_pos[0],
                                            view_pos[1],
view_pos[2], 1);
      fragment_shader_payload payload(
          interpolated_color, interpolated_normal.normalized(),
          interpolated_texcoords, texture ? &*texture : nullptr);
      payload.view_pos = interpolated_viewpos;
      auto pixel_color = fragment_shader(payload);
      set_pixel(Vector2i(x, y), pixel_color);
    }
  }
  // Use: fragment_shader_payload payload( interpolated_color,
  // interpolated_normal.normalized(), interpolated_texcoords,
texture ?
  // &*texture : nullptr); Use: payload.view_pos =
interpolated_shadingcoords;
  // Use: Instead of passing the triangle's color directly to the
frame buffer,
  // pass the color to the shaders first to get the final color;
Use: auto
  // pixel_color = fragment_shader(payload);
}
```

然后是实现 Blinn-Phong 模型计算 Fragment Color.

```
Eigen::Vector3f phong_fragment_shader(const
fragment_shader_payload& payload)
{
  Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
```

```
  Eigen::Vector3f kd = payload.color;
  Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937, 0.7937);

  auto l1 = light{{20, 20, 20}, {500, 500, 500}};
  auto l2 = light{{-20, 20, 0}, {500, 500, 500}};
  std::vector<light> lights = {l1, l2};
  Eigen::Vector3f amb_light_intensity{10, 10, 10};
  Eigen::Vector3f eye_pos{0, 0, 10};
  float p = 150;
  Eigen::Vector3f color = payload.color;
  Eigen::Vector3f point = payload.view_pos;
  Eigen::Vector3f normal = payload.normal;
  Eigen::Vector3f result_color = {0, 0, 0};
  for (auto& light : lights)
  {
    // TODO: For each light source in the code, calculate what the
*ambient*,
    // *diffuse*, and *specular* components are. Then, accumulate
that result on
    // the *result_color* object.
  }
  return result_color * 255.f;
}
```

然后是实现 Texture Shading Fragment Shader.

```
Eigen::Vector3f texture_fragment_shader(const
fragment_shader_payload& payload)
{
  Eigen::Vector3f return_color = {0, 0, 0};
  if (payload.texture)
  {
    // TODO: Get the texture value at the texture coordinates of
the current
    // fragment
  }
  Eigen::Vector3f texture_color;
  texture_color << return_color.x(), return_color.y(),
return_color.z();

  Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
  Eigen::Vector3f kd = texture_color / 255.f;
  Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937, 0.7937);
  auto l1 = light{{20, 20, 20}, {500, 500, 500}};
  auto l2 = light{{-20, 20, 0}, {500, 500, 500}};
  std::vector<light> lights = {l1, l2};
```

```
  Eigen::Vector3f amb_light_intensity{10, 10, 10};
  Eigen::Vector3f eye_pos{0, 0, 10};
  float p = 150;
  Eigen::Vector3f color = texture_color;
  Eigen::Vector3f point = payload.view_pos;
  Eigen::Vector3f normal = payload.normal;
  Eigen::Vector3f result_color = {0, 0, 0};
  for (auto& light : lights)
  {
    // TODO: For each light source in the code, calculate what the
*ambient*,
    // *diffuse*, and *specular* components are. Then, accumulate
that result on
    // the *result_color* object.
  }
  return result_color * 255.f;
}
```

然后是实现 Bump mapping.

```
Eigen::Vector3f bump_fragment_shader(const
fragment_shader_payload& payload)
{

  Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
  Eigen::Vector3f kd = payload.color;
  Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937, 0.7937);
  auto l1 = light{{20, 20, 20}, {500, 500, 500}};
  auto l2 = light{{-20, 20, 0}, {500, 500, 500}};
  std::vector<light> lights = {l1, l2};
  Eigen::Vector3f amb_light_intensity{10, 10, 10};
  Eigen::Vector3f eye_pos{0, 0, 10};
  float p = 150;
  Eigen::Vector3f color = payload.color;
  Eigen::Vector3f point = payload.view_pos;
  Eigen::Vector3f normal = payload.normal;
  float kh = 0.2, kn = 0.1;
  // TODO: Implement bump mapping here
  // Let n = normal = (x, y, z)
  // Vector t =
(x*y/sqrt(x*x+z*z),sqrt(x*x+z*z),z*y/sqrt(x*x+z*z))
  // Vector b = n cross product t
  // Matrix TBN = [t b n]
  // dU = kh * kn * (h(u+1/w,v)-h(u,v))
  // dV = kh * kn * (h(u,v+1/h)-h(u,v))
  // Vector ln = (-dU, -dV, 1)
```

```
  // Normal n = normalize(TBN * ln)
  Eigen::Vector3f result_color = {0, 0, 0};
  result_color = normal;
  return result_color * 255.f;
}
```

然后是实现 displacement mapping.

```
Eigen::Vector3f
displacement_fragment_shader(const fragment_shader_payload&
payload)
{

  Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
  Eigen::Vector3f kd = payload.color;
  Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937, 0.7937);
  auto l1 = light{{20, 20, 20}, {500, 500, 500}};
  auto l2 = light{{-20, 20, 0}, {500, 500, 500}};
  std::vector<light> lights = {l1, l2};
  Eigen::Vector3f amb_light_intensity{10, 10, 10};
  Eigen::Vector3f eye_pos{0, 0, 10};
  float p = 150;
  Eigen::Vector3f color = payload.color;
  Eigen::Vector3f point = payload.view_pos;
  Eigen::Vector3f normal = payload.normal;
  float kh = 0.2, kn = 0.1;
  // TODO: Implement displacement mapping here
  // Let n = normal = (x, y, z)
  // Vector t =
(x*y/sqrt(x*x+z*z),sqrt(x*x+z*z),z*y/sqrt(x*x+z*z))
  // Vector b = n cross product t
  // Matrix TBN = [t b n]
  // dU = kh * kn * (h(u+1/w,v)-h(u,v))
  // dV = kh * kn * (h(u,v+1/h)-h(u,v))
  // Vector ln = (-dU, -dV, 1)
  // Position p = p + kn * n * h(u,v)
  // Normal n = normalize(TBN * ln)
  Eigen::Vector3f result_color = {0, 0, 0};
  for (auto& light : lights)
  {
    // TODO: For each light source in the code, calculate what the
*ambient*,
    // *diffuse*, and *specular* components are. Then, accumulate
that result on
    // the *result_color* object.
  }
```

```
    return result_color * 255.f;
}
```
然后为了附加实验我们实现一下双线性插值算法

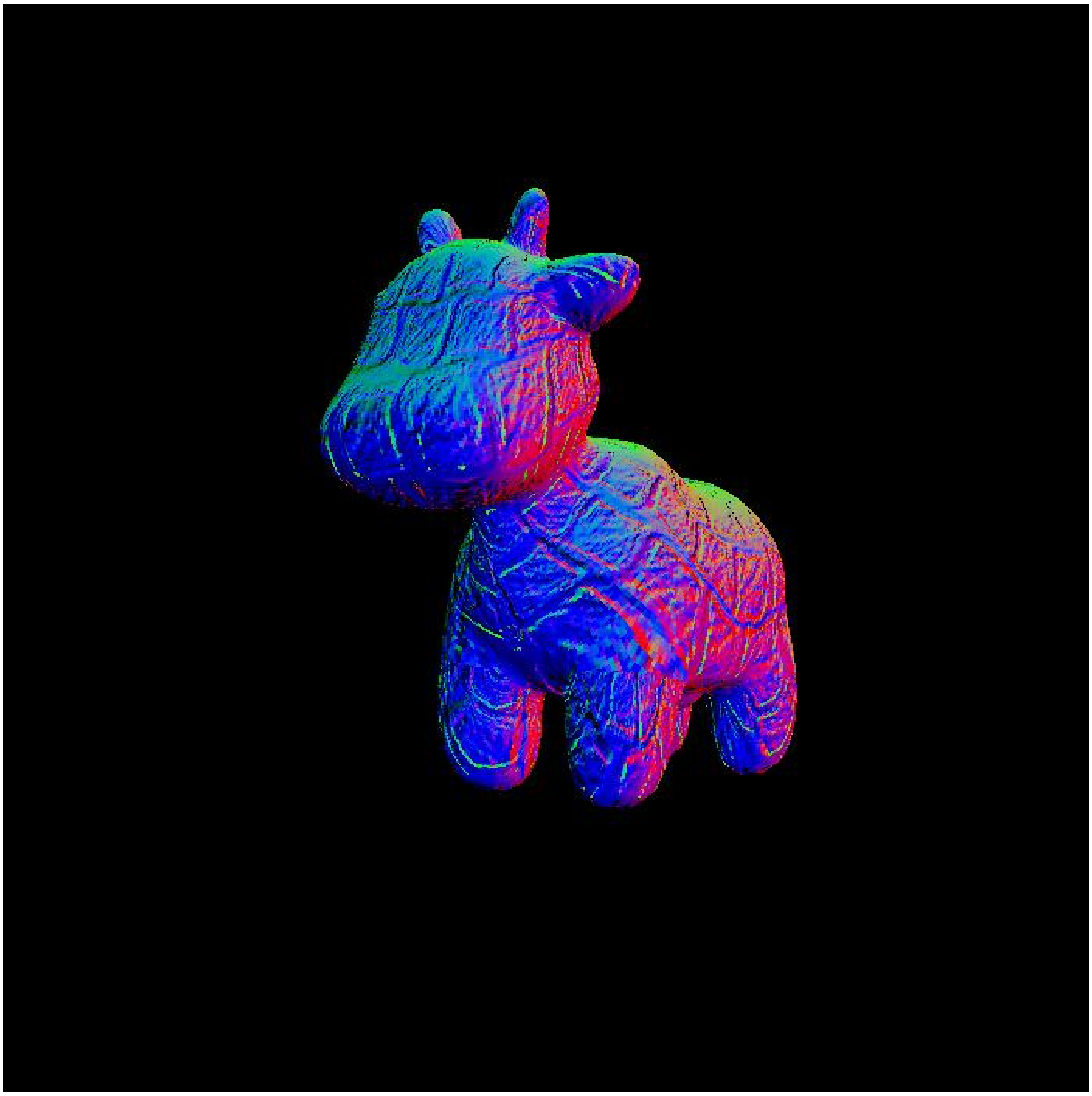
```
    Eigen::Vector3f getColorBilinear(float u, float v)
    {
        auto x = u * width;//像素 x 坐标
        auto y = (1 - v) * height;//像素 y 坐标
        int roundx = static_cast<int>(x);//最近的纹理点 x 坐标
        int roundy = static_cast<int>(y);//最近的纹理点 y 坐标

        //两个插值比例
        float s = x - roundx;
        float t = y - roundy;
        //四个相邻纹理点
        auto p00 = image_data.at<cv::Vec3b>(roundy, roundx);
        auto p01 = image_data.at<cv::Vec3b>(roundy, 
std::min(roundx + 1, width - 1));
        auto p10 = image_data.at<cv::Vec3b>(std::min(roundy + 1, 
height - 1), roundx);
        auto p11 = image_data.at<cv::Vec3b>(std::min(roundy + 1, 
height - 1), std::min(roundx + 1, width - 1));
        //水平方向插值
        auto px = p00 * (1 - s) + p10 * s;
        auto py = p01 * (1 - s) + p11 * s;
        //竖直方向插值
        auto color = px * (1 - t) + py * t;
        return Eigen::Vector3f(color[0], color[1], color[2]);
    }
```
实验结果:

Phong.png:

Normal.png:

Texture.png:

Bump.png:



Displacement.png:

然后是双线性插值绘制纹理的结果：

实验代码：

Rasterizer.cpp

```
//
// Created by goksu on 4/6/19.
//

#include <algorithm>
#include "rasterizer.hpp"
#include <opencv2/opencv.hpp>
#include <math.h>
rst::pos_buf_id rst::rasterizer::load_positions(const
std::vector<Eigen::Vector3f> &positions)
{
    auto id = get_next_id();
    pos_buf.emplace(id, positions);
    return {id};
}
```

```
rst::ind_buf_id rst::rasterizer::load_indices(const
std::vector<Eigen::Vector3i> &indices)
{
    auto id = get_next_id();
    ind_buf.emplace(id, indices);
    return {id};
}
rst::col_buf_id rst::rasterizer::load_colors(const
std::vector<Eigen::Vector3f> &cols)
{
    auto id = get_next_id();
    col_buf.emplace(id, cols);
    return {id};
}
rst::col_buf_id rst::rasterizer::load_normals(const
std::vector<Eigen::Vector3f>& normals)
{
    auto id = get_next_id();
    nor_buf.emplace(id, normals);
    normal_id = id;
    return {id};
}
// Bresenham's line drawing algorithm
void rst::rasterizer::draw_line(Eigen::Vector3f begin,
Eigen::Vector3f end)
{
    auto x1 = begin.x();
    auto y1 = begin.y();
    auto x2 = end.x();
    auto y2 = end.y();
    Eigen::Vector3f line_color = {255, 255, 255};
    int x,y,dx,dy,dx1,dy1,px,py,xe,ye,i;
    dx=x2-x1;
    dy=y2-y1;
    dx1=fabs(dx);
    dy1=fabs(dy);
    px=2*dy1-dx1;
    py=2*dx1-dy1;
    if(dy1<=dx1)
    {
        if(dx>=0)
        {
            x=x1;
            y=y1;
```

```
            xe=x2;
        }
        else
        {
            x=x2;
            y=y2;
            xe=x1;
        }
        Eigen::Vector2i point = Eigen::Vector2i(x, y);
        set_pixel(point,line_color);
        for(i=0;x<xe;i++)
        {
            x=x+1;
            if(px<0)
            {
                px=px+2*dy1;
            }
            else
            {
                if((dx<0 && dy<0) || (dx>0 && dy>0))
                {
                    y=y+1;
                }
                else
                {
                    y=y-1;
                }
                px=px+2*(dy1-dx1);
            }
//            delay(0);
            Eigen::Vector2i point = Eigen::Vector2i(x, y);
            set_pixel(point,line_color);
        }
    }
    else
    {
        if(dy>=0)
        {
            x=x1;
            y=y1;
            ye=y2;
        }
        else
        {
```

```
            x=x2;
            y=y2;
            ye=y1;
        }
        Eigen::Vector2i point = Eigen::Vector2i(x, y);
        set_pixel(point,line_color);
        for(i=0;y<ye;i++)
        {
            y=y+1;
            if(py<=0)
            {
                py=py+2*dx1;
            }
            else
            {
                if((dx<0 && dy<0) || (dx>0 && dy>0))
                {
                    x=x+1;
                }
                else
                {
                    x=x-1;
                }
                py=py+2*(dx1-dy1);
            }
//             delay(0);
            Eigen::Vector2i point = Eigen::Vector2i(x, y);
            set_pixel(point,line_color);
        }
    }
}
auto to_vec4(const Eigen::Vector3f& v3, float w = 1.0f)
{
    return Vector4f(v3.x(), v3.y(), v3.z(), w);
}
static bool insideTriangle(int x, int y, const Vector4f* _v){
    Vector3f v[3];
    for(int i=0;i<3;i++)
        v[i] = {_v[i].x(),_v[i].y(), 1.0};
    Vector3f f0,f1,f2;
    f0 = v[1].cross(v[0]);
    f1 = v[2].cross(v[1]);
    f2 = v[0].cross(v[2]);
    Vector3f p(x,y,1.);
```

```
    if((p.dot(f0)*f0.dot(v[2])>0) &&
(p.dot(f1)*f1.dot(v[0])>0) && (p.dot(f2)*f2.dot(v[1])>0))
        return true;
    return false;
}
static std::tuple<float, float, float>
computeBarycentric2D(float x, float y, const Vector4f* v){
    float c1 = (x*(v[1].y() - v[2].y()) + (v[2].x() - v[1].x())*y
+ v[1].x()*v[2].y() - v[2].x()*v[1].y()) / (v[0].x()*(v[1].y()
- v[2].y()) + (v[2].x() - v[1].x())*v[0].y() + v[1].x()*v[2].y()
- v[2].x()*v[1].y());
    float c2 = (x*(v[2].y() - v[0].y()) + (v[0].x() - v[2].x())*y
+ v[2].x()*v[0].y() - v[0].x()*v[2].y()) / (v[1].x()*(v[2].y()
- v[0].y()) + (v[0].x() - v[2].x())*v[1].y() + v[2].x()*v[0].y()
- v[0].x()*v[2].y());
    float c3 = (x*(v[0].y() - v[1].y()) + (v[1].x() - v[0].x())*y
+ v[0].x()*v[1].y() - v[1].x()*v[0].y()) / (v[2].x()*(v[0].y()
- v[1].y()) + (v[1].x() - v[0].x())*v[2].y() + v[0].x()*v[1].y()
- v[1].x()*v[0].y());
    return {c1,c2,c3};
}
void rst::rasterizer::draw(std::vector<Triangle *>
&TriangleList) {
    float f1 = (50 - 0.1) / 2.0;
    float f2 = (50 + 0.1) / 2.0;
    Eigen::Matrix4f mvp = projection * view * model;
    for (const auto& t:TriangleList)
    {
        Triangle newtri = *t;
        std::array<Eigen::Vector4f, 3> mm {
                (view * model * t->v[0]),
                (view * model * t->v[1]),
                (view * model * t->v[2])
        };
        std::array<Eigen::Vector3f, 3> viewspace_pos;
        std::transform(mm.begin(), mm.end(),
viewspace_pos.begin(), [](auto& v) {
            return v.template head<3>();
        });
        Eigen::Vector4f v[] = {
                mvp * t->v[0],
                mvp * t->v[1],
                mvp * t->v[2]
        };
```

```
        //Homogeneous division
        for (auto& vec : v) {
            vec.x()/=vec.w();
            vec.y()/=vec.w();
            vec.z()/=vec.w();
        }
        Eigen::Matrix4f inv_trans = (view *
model).inverse().transpose();
        Eigen::Vector4f n[] = {
                inv_trans * to_vec4(t->normal[0], 0.0f),
                inv_trans * to_vec4(t->normal[1], 0.0f),
                inv_trans * to_vec4(t->normal[2], 0.0f)
        };
        //Viewport transformation
        for (auto & vert : v)
        {
            vert.x() = 0.5*width*(vert.x()+1.0);
            vert.y() = 0.5*height*(vert.y()+1.0);
            vert.z() = vert.z() * f1 + f2;
        }
        for (int i = 0; i < 3; ++i)
        {
            //screen space coordinates
            newtri.setVertex(i, v[i]);
        }
        for (int i = 0; i < 3; ++i)
        {
            //view space normal
            newtri.setNormal(i, n[i].head<3>());
        }
        newtri.setColor(0, 148,121.0,92.0);
        newtri.setColor(1, 148,121.0,92.0);
        newtri.setColor(2, 148,121.0,92.0);
        // Also pass view space vertice position
        rasterize_triangle(newtri, viewspace_pos);
    }
}
static Eigen::Vector3f interpolate(float alpha, float beta,
float gamma, const Eigen::Vector3f& vert1, const
Eigen::Vector3f& vert2, const Eigen::Vector3f& vert3, float
weight)
{
    return (alpha * vert1 + beta * vert2 + gamma * vert3) / weight;
}
```

```
static Eigen::Vector2f interpolate(float alpha, float beta,
float gamma, const Eigen::Vector2f& vert1, const
Eigen::Vector2f& vert2, const Eigen::Vector2f& vert3, float
weight)
{
    auto u = (alpha * vert1[0] + beta * vert2[0] + gamma *
vert3[0]);
    auto v = (alpha * vert1[1] + beta * vert2[1] + gamma *
vert3[1]);
    u /= weight;
    v /= weight;
    return Eigen::Vector2f(u, v);
}
//Screen space rasterization
void rst::rasterizer::rasterize_triangle(const Triangle& t,
const std::array<Eigen::Vector3f, 3>& view_pos)
{
    // TODO : Find out the bounding box of current triangle.
    float aabb_minx = 0;
    float aabb_miny = 0;
    float aabb_maxx = 0;
    float aabb_maxy = 0;
    for (size_t i = 0; i < 3; i++)
    {
        const Vector4f& p = t.v[i];
        if(i == 0)
        {
            aabb_minx = aabb_maxx = p.x();
            aabb_miny = aabb_maxy = p.y();
            continue;
        }
        aabb_minx = p.x() < aabb_minx ? p.x() : aabb_minx;
        aabb_miny = p.y() < aabb_miny ? p.y() : aabb_miny;
        aabb_maxx = p.x() > aabb_maxx ? p.x() : aabb_maxx;
        aabb_maxy = p.y() > aabb_maxy ? p.y() : aabb_maxy;
    }

    // iterate through the pixel and find if the current pixel
is inside the triangle
    auto v = t.v;
    for(int x = (int)aabb_minx;x < aabb_maxx;x++)
    {
        for (int y = (int)aabb_miny; y < aabb_maxy; y++)
        {
```

```
            if(!insideTriangle(x,y,t.v)) continue;
            // TODO: Inside your rasterization loop:
            //    * v[i].w() is the vertex view space depth value
z.
            //    * Z is interpolated view space depth for the
current pixel
            //    * zp is depth between zNear and zFar, used for
z-buffer
            auto[alpha, beta, gamma] = computeBarycentric2D(x,
y, t.v);
            float Z = 1.0 / (alpha / v[0].w() + beta / v[1].w()
+ gamma / v[2].w());
            float zp = alpha * v[0].z() / v[0].w() + beta *
v[1].z() / v[1].w() + gamma * v[2].z() / v[2].w();
            zp *= Z;
            int buf_index = get_index(x,y);
            if(zp >= depth_buf[buf_index]) continue;
            depth_buf[buf_index] = zp;
            // TODO: Interpolate the attributes:
            // auto interpolated_color
            // auto interpolated_normal
            // auto interpolated_texcoords
            // auto interpolated_shadingcoords
            auto interpolated_color =
interpolate(alpha,beta,gamma,t.color[0],t.color[1],t.color[2]
,1);
            auto interpolated_normal =
interpolate(alpha,beta,gamma,t.normal[0],t.normal[1],t.normal
[2],1);
            auto interpolated_texcoords
=  interpolate(alpha,beta,gamma,t.tex_coords[0],t.tex_coords[
1],t.tex_coords[2],1);
            auto interpolated_viewpos =
interpolate(alpha,beta,gamma,view_pos[0],view_pos[1],view_pos
[2],1);
            fragment_shader_payload
payload( interpolated_color, interpolated_normal.normalized(),
interpolated_texcoords, texture ? &*texture : nullptr);
            payload.view_pos = interpolated_viewpos;
            auto pixel_color = fragment_shader(payload);
            set_pixel(Vector2i(x,y),pixel_color);
        }
    }
    // Use: fragment_shader_payload
```

```
payload( interpolated_color, interpolated_normal.normalized(),
interpolated_texcoords, texture ? &*texture : nullptr);
    // Use: payload.view_pos = interpolated_shadingcoords;
    // Use: Instead of passing the triangle's color directly to
the frame buffer, pass the color to the shaders first to get the
final color;
    // Use: auto pixel_color = fragment_shader(payload);

}
void rst::rasterizer::set_model(const Eigen::Matrix4f& m)
{
    model = m;
}
void rst::rasterizer::set_view(const Eigen::Matrix4f& v)
{
    view = v;
}
void rst::rasterizer::set_projection(const Eigen::Matrix4f& p)
{
    projection = p;
}
void rst::rasterizer::clear(rst::Buffers buff)
{
    if ((buff & rst::Buffers::Color) == rst::Buffers::Color)
    {
        std::fill(frame_buf.begin(), frame_buf.end(),
Eigen::Vector3f{0, 0, 0});
    }
    if ((buff & rst::Buffers::Depth) == rst::Buffers::Depth)
    {
        std::fill(depth_buf.begin(), depth_buf.end(),
std::numeric_limits<float>::infinity());
    }
}
rst::rasterizer::rasterizer(int w, int h) : width(w), height(h)
{
    frame_buf.resize(w * h);
    depth_buf.resize(w * h);
    texture = std::nullopt;
}
int rst::rasterizer::get_index(int x, int y)
{
    return (height-1-y)*width + x;
}
```

```
void rst::rasterizer::set_pixel(const Vector2i &point, const
Eigen::Vector3f &color)
{
    //old index: auto ind = point.y() + point.x() * width;
    int ind = (height-1-point.y())*width + point.x();
    frame_buf[ind] = color;
}
void
rst::rasterizer::set_vertex_shader(std::function<Eigen::Vecto
r3f(vertex_shader_payload)> vert_shader)
{
    vertex_shader = vert_shader;
}
void
rst::rasterizer::set_fragment_shader(std::function<Eigen::Vec
tor3f(fragment_shader_payload)> frag_shader)
{
    fragment_shader = frag_shader;
}
```

Main.cpp

```
#pragma warning( disable : 4305 )
#include <iostream>
#include <opencv2/opencv.hpp>
#include "global.hpp"
#include "rasterizer.hpp"
#include "Triangle.hpp"
#include "Shader.hpp"
#include "Texture.hpp"
#include "OBJ_Loader.h"
Eigen::Matrix4f get_view_matrix(Eigen::Vector3f eye_pos)
{
    Eigen::Matrix4f view = Eigen::Matrix4f::Identity();
    Eigen::Matrix4f translate;
    translate << 1,0,0,-eye_pos[0],
                0,1,0,-eye_pos[1],
                0,0,1,-eye_pos[2],
                0,0,0,1;
    view = translate*view;
    return view;
}
Eigen::Matrix4f get_model_matrix(float angle)
{
    Eigen::Matrix4f rotation;
    angle = angle * MY_PI / 180.f;
```

```
    rotation << cos(angle), 0, sin(angle), 0,
                0, 1, 0, 0,
                -sin(angle), 0, cos(angle), 0,
                0, 0, 0, 1;
    Eigen::Matrix4f scale;
    scale << 2.5, 0, 0, 0,
              0, 2.5, 0, 0,
              0, 0, 2.5, 0,
              0, 0, 0, 1;
    Eigen::Matrix4f translate;
    translate << 1, 0, 0, 0,
            0, 1, 0, 0,
            0, 0, 1, 0,
            0, 0, 0, 1;
    return translate * rotation * scale;
}
Eigen::Matrix4f get_projection_matrix(float eye_fov, float
aspect_ratio, float zNear, float zFar)
{
    Eigen::Matrix4f projection;
    float top = -tan(DEG2RAD(eye_fov/2.0f) * abs(zNear));
    float right = top * aspect_ratio;
    projection << zNear/right,0,0,0,
                  0,zNear/top,0,0,
                  0,0,(zNear+zFar)/(zNear-zFar),(2*zNear*zFar)
/(zFar-zNear),
                  0,0,1,0;
    return projection;
}
Eigen::Vector3f vertex_shader(const vertex_shader_payload&
payload)
{
    return payload.position;
}
Eigen::Vector3f normal_fragment_shader(const
fragment_shader_payload& payload)
{
    Eigen::Vector3f return_color =
(payload.normal.head<3>().normalized() + Eigen::Vector3f(1.0f,
1.0f, 1.0f)) / 2.f;
    Eigen::Vector3f result;
    result << return_color.x() * 255, return_color.y() * 255,
return_color.z() * 255;
    return result;
```

```
}
static Eigen::Vector3f reflect(const Eigen::Vector3f& vec,
const Eigen::Vector3f& axis)
{
    auto costheta = vec.dot(axis);
    return (2 * costheta * axis - vec).normalized();
}
struct light
{
    Eigen::Vector3f position;
    Eigen::Vector3f intensity;
};
Eigen::Vector3f texture_fragment_shader(const
fragment_shader_payload& payload)
{
    Eigen::Vector3f return_color = {0, 0, 0};
    if (payload.texture)
    {
        // TODO: Get the texture value at the texture coordinates
of the current fragment
        return_color =
payload.texture->getColor(payload.tex_coords.x(),payload.tex_
coords.y());
    }
    Eigen::Vector3f texture_color;
    texture_color << return_color.x(), return_color.y(),
return_color.z();
    Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
    Eigen::Vector3f kd = texture_color / 255.f;
    Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937,
0.7937);
    auto l1 = light{{20, 20, 20}, {500, 500, 500}};
    auto l2 = light{{-20, 20, 0}, {500, 500, 500}};
    std::vector<light> lights = {l1, l2};
    Eigen::Vector3f amb_light_intensity{10, 10, 10};
    Eigen::Vector3f eye_pos{0, 0, 10};
    float p = 150;
    Eigen::Vector3f color = texture_color;
    Eigen::Vector3f point = payload.view_pos;
    Eigen::Vector3f normal = payload.normal;
    Eigen::Vector3f result_color = {0, 0, 0};
    Vector3f view_dir = (eye_pos - point).normalized();
    for (auto& light : lights)
    {
```

```
        // TODO: For each light source in the code, calculate what
the *ambient*, *diffuse*, and *specular*
        // components are. Then, accumulate that result on the
*result_color* object.
        float rr =  ( light.position -point).squaredNorm();
        Vector3f diffsue(0,0,0);
        Vector3f specular(0,0,0);
        Vector3f ambient(0,0,0);
        Vector3f light_dir =  (light.position
-point).normalized();

        for (size_t i = 0; i < 3; i++)
        {
            Vector3f h = (view_dir + light_dir).normalized(); //
half
            float intensity = light.intensity[i]/rr;
            diffsue[i] = kd[i] * intensity *
std::max(0.0f,normal.dot(light_dir));
            specular[i] = ks[i] * intensity *
std::pow(std::max(0.0f,normal.dot(h)),p);
            ambient[i] = amb_light_intensity[i] * ka[i];
        }
        result_color += diffsue;
        result_color += specular;
        result_color += ambient;
    }
    return result_color * 255.f;
}
Eigen::Vector3f phong_fragment_shader(const
fragment_shader_payload& payload)
{
    Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
    Eigen::Vector3f kd = payload.color;
    Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937,
0.7937);
    auto l1 = light{{20, 20, 20}, {500, 500, 500}};
    auto l2 = light{{-20, 20, 0}, {500, 500, 500}};
    std::vector<light> lights = {l1, l2};
    Eigen::Vector3f amb_light_intensity{10, 10, 10};
    Eigen::Vector3f eye_pos{0, 0, 10};
    float p = 150;
    Eigen::Vector3f color = payload.color;
    Eigen::Vector3f point = payload.view_pos;
    Eigen::Vector3f normal = payload.normal.normalized();
```

```
    Eigen::Vector3f result_color = {0, 0, 0};
    Vector3f view_dir = (eye_pos - point).normalized();
    for (auto& light : lights)
    {
        // TODO: For each light source in the code, calculate what
the *ambient*, *diffuse*, and *specular*
        // components are. Then, accumulate that result on the
*result_color* object.
        float rr =  ( light.position -point).squaredNorm();
        Vector3f diffsue(0,0,0);
        Vector3f specular(0,0,0);
        Vector3f ambient(0,0,0);
        Vector3f light_dir =  (light.position
-point).normalized();

        for (size_t i = 0; i < 3; i++)
        {
            Vector3f h = (view_dir + light_dir).normalized(); //
half
            float intensity = light.intensity[i]/rr;
            diffsue[i] = kd[i] * intensity *
std::max(0.0f,normal.dot(light_dir));
            specular[i] = ks[i] * intensity *
std::pow(std::max(0.0f,normal.dot(h)),p);
            ambient[i] = amb_light_intensity[i] * ka[i];
        }
        result_color += diffsue;
        result_color += specular;
        result_color += ambient;
    }
    return result_color * 255.f;
}
Eigen::Vector3f displacement_fragment_shader(const
fragment_shader_payload& payload)
{

    Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
    Eigen::Vector3f kd = payload.color;
    Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937,
0.7937);
    auto l1 = light{{20, 20, 20}, {500, 500, 500}};
    auto l2 = light{{-20, 20, 0}, {500, 500, 500}};
    std::vector<light> lights = {l1, l2};
    Eigen::Vector3f amb_light_intensity{10, 10, 10};
```

```
    Eigen::Vector3f eye_pos{0, 0, 10};
    Eigen::Vector3f color = payload.color;
    Eigen::Vector3f point = payload.view_pos;
    Eigen::Vector3f normal = payload.normal;
    float kh = 0.2, kn = 0.1;

    // TODO: Implement displacement mapping here
    // Let n = normal = (x, y, z)
    // Vector t =
(x*y/sqrt(x*x+z*z),sqrt(x*x+z*z),z*y/sqrt(x*x+z*z))
    // Vector b = n cross product t
    // Matrix TBN = [t b n]
    // dU = kh * kn * (h(u+1/w,v)-h(u,v))
    // dV = kh * kn * (h(u,v+1/h)-h(u,v))
    // Vector ln = (-dU, -dV, 1)
    // Position p = p + kn * n * h(u,v)
    // Normal n = normalize(TBN * ln)
    float x = normal.x();
    float y = normal.y();
    float z = normal.z();
    Vector3f
t(x*y/sqrt(x*x+z*z),sqrt(x*x+z*z),z*y/sqrt(x*x+z*z));
    Vector3f b = normal.cross(t);
    Matrix3f TBN;
    TBN.col(0) = t.normalized();
    TBN.col(1) = b.normalized();
    TBN.col(2) = normal;

    int w = payload.texture->width;
    int h = payload.texture->height;
    float u = payload.tex_coords.x();
    float v = payload.tex_coords.y();
    payload.texture->getColor(u,v);
    auto huv = payload.texture->getColor(u,v).norm();

    float dU = kh * kn *
(payload.texture->getColor(u+1.0f/w,v).norm()-huv);
    float dV = kh * kn *
(payload.texture->getColor(u,v+1.0f/h).norm()-huv);
    Vector3f ln(-dU,-dV,1);
    Vector3f n = (TBN * ln).normalized();
    Vector3f p = point +  n * huv * kn;
    Eigen::Vector3f result_color = {0, 0, 0};
    Vector3f view_dir = (eye_pos - p).normalized();
```

```
    for (auto& light : lights)
    {
        // TODO: For each light source in the code, calculate what
the *ambient*, *diffuse*, and *specular*
        // components are. Then, accumulate that result on the
*result_color* object.
        float rr =  ( light.position -p).squaredNorm();
        Vector3f diffsue(0,0,0);
        Vector3f specular(0,0,0);
        Vector3f ambient(0,0,0);
        Vector3f light_dir =  (light.position -p).normalized();

        for (size_t i = 0; i < 3; i++)
        {
            Vector3f h = (view_dir + light_dir).normalized(); //
half
            float intensity = light.intensity[i]/rr;
            diffsue[i] = kd[i] * intensity *
std::max(0.0f,normal.dot(light_dir));
            specular[i] = ks[i] * intensity *
std::pow(std::max(0.0f,normal.dot(h)),150);
            ambient[i] = amb_light_intensity[i] * ka[i];
        }
        result_color += diffsue;
        result_color += specular;
        result_color += ambient;
    }
    return result_color * 255.f;
}
Eigen::Vector3f bump_fragment_shader(const
fragment_shader_payload& payload)
{

    Eigen::Vector3f ka = Eigen::Vector3f(0.005, 0.005, 0.005);
    Eigen::Vector3f kd = payload.color;
    Eigen::Vector3f ks = Eigen::Vector3f(0.7937, 0.7937,
0.7937);
    auto l1 = light{{20, 20, 20}, {500, 500, 500}};
    auto l2 = light{{-20, 20, 0}, {500, 500, 500}};
    std::vector<light> lights = {l1, l2};
    Eigen::Vector3f amb_light_intensity{10, 10, 10};
    Eigen::Vector3f eye_pos{0, 0, 10};
    float p = 150;
    Eigen::Vector3f color = payload.color;
```

```
    Eigen::Vector3f point = payload.view_pos;
    Eigen::Vector3f normal = payload.normal.normalized();
    float kh = 0.2, kn = 0.1;
    // TODO: Implement bump mapping here
    // Let n = normal = (x, y, z)
    // Vector t =
(x*y/sqrt(x*x+z*z),sqrt(x*x+z*z),z*y/sqrt(x*x+z*z))
    // Vector b = n cross product t
    // Matrix TBN = [t b n]
    // dU = kh * kn * (h(u+1/w,v)-h(u,v))
    // dV = kh * kn * (h(u,v+1/h)-h(u,v))
    // Vector ln = (-dU, -dV, 1)
    // Normal n = normalize(TBN * ln)
    float x = normal.x();
    float y = normal.y();
    float z = normal.z();
    Vector3f
t(x*y/sqrt(x*x+z*z),sqrt(x*x+z*z),z*y/sqrt(x*x+z*z));
    Vector3f b = normal.cross(t);
    Matrix3f TBN;
    TBN.col(0) = t;
    TBN.col(1) = b;
    TBN.col(2) = normal;

    int w = payload.texture->width;
    int h = payload.texture->height;
    float u = payload.tex_coords.x();
    float v = payload.tex_coords.y();
    payload.texture->getColor(u,v);
    auto huv = payload.texture->getColor(u,v).norm();

    float dU = kh * kn *
(payload.texture->getColor(u+1.0f/w,v).norm()-huv);
    float dV = kh * kn *
(payload.texture->getColor(u,v+1.0f/h).norm()-huv);
    Vector3f ln(-dU,-dV,1);
    Vector3f n = (TBN * ln).normalized();
    Eigen::Vector3f result_color = n;
    return result_color * 255.f;
}
int main(int argc, const char** argv)
{
    std::vector<Triangle*> TriangleList;
    float angle = 140.0;
```

```
    bool command_line = false;
    std::string filename = "output.png";
    objl::Loader Loader;
    std::string obj_path = "./models/spot/";
    // Load .obj File
    bool loadout =
Loader.LoadFile("./models/spot/spot_triangulated_good.obj");
    for(auto mesh:Loader.LoadedMeshes)
    {
        for(int i=0;i<mesh.Vertices.size();i+=3)
        {
            Triangle* t = new Triangle();
            for(int j=0;j<3;j++)
            {
                t->setVertex(j,Vector4f(mesh.Vertices[i+j].Po
sition.X,mesh.Vertices[i+j].Position.Y,mesh.Vertices[i+j].Pos
ition.Z,1.0));
                t->setNormal(j,Vector3f(mesh.Vertices[i+j].No
rmal.X,mesh.Vertices[i+j].Normal.Y,mesh.Vertices[i+j].Normal.
Z));
                t->setTexCoord(j,Vector2f(mesh.Vertices[i+j].
TextureCoordinate.X,
mesh.Vertices[i+j].TextureCoordinate.Y));
            }
            TriangleList.push_back(t);
        }
    }
    rst::rasterizer r(700, 700);
    auto texture_path = "hmap.jpg";
    r.set_texture(Texture(obj_path + texture_path));
    std::function<Eigen::Vector3f(fragment_shader_payload)>
active_shader = phong_fragment_shader;
    if (argc >= 2)
    {
        command_line = true;
        filename = std::string(argv[1]);
        if (argc == 3 && std::string(argv[2]) == "texture")
        {
            std::cout << "Rasterizing using the texture
shader\n";
            active_shader = texture_fragment_shader;
            texture_path = "spot_texture.png";
            r.set_texture(Texture(obj_path + texture_path));
        }
```

```
        else if (argc == 3 && std::string(argv[2]) == "normal")
        {
            std::cout << "Rasterizing using the normal shader\n";
            active_shader = normal_fragment_shader;
        }
        else if (argc == 3 && std::string(argv[2]) == "phong")
        {
            std::cout << "Rasterizing using the phong shader\n";
            active_shader = phong_fragment_shader;
        }
        else if (argc == 3 && std::string(argv[2]) == "bump")
        {
            std::cout << "Rasterizing using the bump shader\n";
            active_shader = bump_fragment_shader;
        }
        else if (argc == 3 && std::string(argv[2]) ==
"displacement")
        {
            std::cout << "Rasterizing using the bump shader\n";
            active_shader = displacement_fragment_shader;
        }
    }
    Eigen::Vector3f eye_pos = {0,0,10};
    r.set_vertex_shader(vertex_shader);
    r.set_fragment_shader(active_shader);
    int key = 0;
    int frame_count = 0;
    if (command_line)
    {
        r.clear(rst::Buffers::Color | rst::Buffers::Depth);
        r.set_model(get_model_matrix(angle));
        r.set_view(get_view_matrix(eye_pos));
        r.set_projection(get_projection_matrix(45.0, 1, 0.1,
50));
        r.draw(TriangleList);
        cv::Mat image(700, 700, CV_32FC3,
r.frame_buffer().data());
        image.convertTo(image, CV_8UC3, 1.0f);
        cv::cvtColor(image, image, cv::COLOR_RGB2BGR);
        cv::imwrite(filename, image);
        return 0;
    }
    while(key != 'q')
    {
```

```
        r.clear(rst::Buffers::Color | rst::Buffers::Depth);
        r.set_model(get_model_matrix(angle));
        r.set_view(get_view_matrix(eye_pos));
        r.set_projection(get_projection_matrix(45.0, 1, 0.1,
50));
        //r.draw(pos_id, ind_id, col_id,
rst::Primitive::Triangle);
        r.draw(TriangleList);
        cv::Mat image(700, 700, CV_32FC3,
r.frame_buffer().data());
        image.convertTo(image, CV_8UC3, 1.0f);
        cv::cvtColor(image, image, cv::COLOR_RGB2BGR);
        cv::imshow("image", image);
        cv::imwrite(filename, image);
        key = cv::waitKey(0);
        if (key == 'a' )
        {
            angle -= 0.1;
        }
        else if (key == 'd')
        {
            angle += 0.1;
        }
    }
    return 0;
}
```

实验中存在的问题及解决：