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](mailto: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. 2. 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）    漫反射：    高光:     1. 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。   1. 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。   1. 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:  phong  Normal.png:  normal  Texture.png:  texture  Bump.png:  bump  Displacement.png:  displacement  然后是双线性插值绘制纹理的结果：  IMG_256  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::Vector3f(vertex\_shader\_payload)> vert\_shader)  {      vertex\_shader = vert\_shader;  }  void rst::rasterizer::set\_fragment\_shader(std::function<Eigen::Vector3f(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].Position.X,mesh.Vertices[i+j].Position.Y,mesh.Vertices[i+j].Position.Z,1.0));                  t->setNormal(j,Vector3f(mesh.Vertices[i+j].Normal.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： | | |