Assignment 4: Lighting

Computer Graphics Teaching Stuff, Sun Yat-Sen University

- Due Date: 11月1号晚上12点之前，提交到zhangzk3@mail2.sysu.edu.cn邮箱。

> 在上一次作业，我们实现了光栅化算法和z-buffering算法，将一个给定的三角形离散化到屏幕的像素上，从而将三角形绘制出来，并考虑距离前后的遮挡关系。如果回顾上一次作业的代码框架，你就会发现我们在光栅化的时候，是直接设置像素的颜色值，没有对输出像素的颜色值做任何复杂的操作，故而绘制出来的三角形是纯色的。在学习了传统的光照算法之后，我们将在这次作业实现Phong光照模型和Blinn-Phong光照模型，给这些像素上色，使之呈现更为真实的渲染效果！注意：我们允许同学们相互讨论，但严禁直接抄袭。

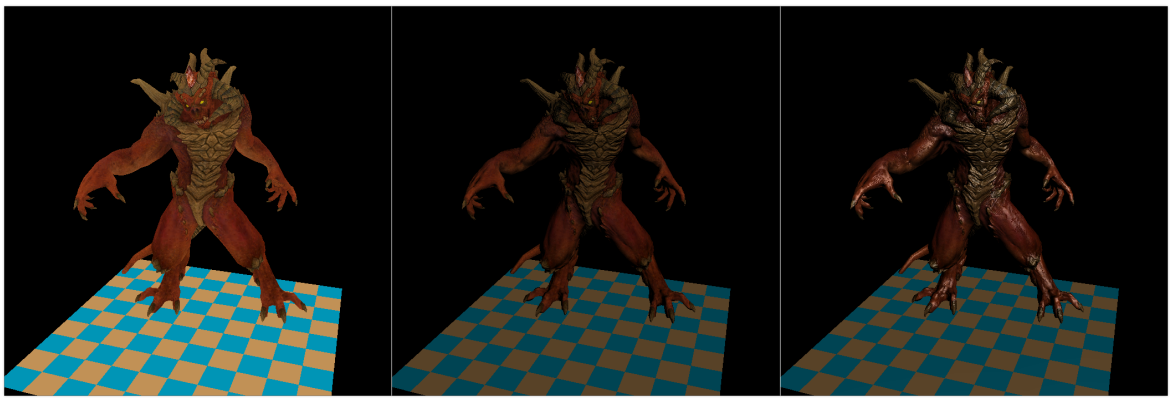


图1 最左边图没有任何光照，跟后面两张图区别很明显

1、总览

本次作业实现的是一个场景的光照算法，该场景由一个暗黑破坏神3中的怪物模型和一个简单的地面构成，框架代码已经构建好了渲染管线的基本流程（即顶点着色器、光栅化和Z-buffering和片元着色器，但片元着色器没有实现任何光照算法），**需要你在片元着色器中实现相关的光照计算，你主要修改的文件就是`main.cpp`文件，主要修改的函数是`fragment()`和`main()`**。你要实现的功能有如下四点：

- 实现Phong光照模型的漫反射光照
- 实现Phong光照模型的镜面反射光照
- 实现Blinn-Phong光照模型的镜面反射光照
- 实现视点绕着y轴旋转

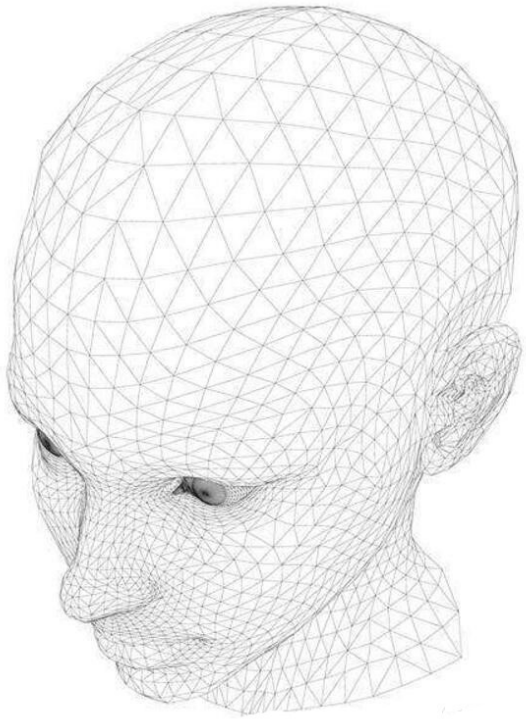
在默认提供的场景中，仅有一个光源——平行光源，默认光的颜色（或者说辐射率）为$(255, 255, 255)$，即白光。平行光源的方向为$(1, 1, 1)$。平行光不考虑距离的衰减，因此可以移除掉相关的光强衰减系数部分。总的来说，你需要实现的Phong光照模型公式如下：

$$L = k_d max(0, n \cdot l) + k_s max(0, n \cdot h)^p$$

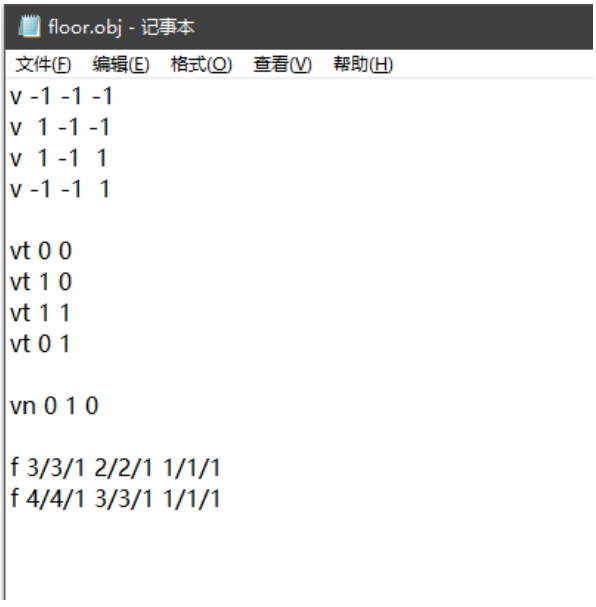
其中k_d和k_s分别是漫反射系数和镜面反射系数，用于调整漫反射光和镜面反射光的占比权重。n是表面的法线向量，l是光照方向，p是镜面反射的高光系数，用于调整镜面反射效果。而上式中的h是视线方向还是half-vector取决于是Phong光照模型和Blinn-Phong光照模型。默认情况下，框架代码中$k_d = 0.8$、$k_s = 0.4$，而p**建议同学们在实现的时候取**16。由于Phong光照模型中的环境光对视觉效果

影响极其微小，这里就不考虑进来了。除此之外，公式中的方向向量一定要归一化为单位向量！

除了光照模型之外，这里还值得一提的就是图形学中一种表示复杂形状物体的方式——网格模型。在前面两次作业中，我们处理的是最基本几何体——三角形，对于三角形、四边形这些简单的几何体，我们直接给出相应的顶点就能描述其形状。但真实世界的物体通常是复杂、不规范的，而且他们往往没有显示的数学公式去表达。为此，一种简单又方便的网格模型诞生了。对于复杂的物体表面，我们用很多个三角形（或者四边形等其他简单的几何体）拼接成几何的复杂表面形状，用有限多个三角形去近似地逼近复杂几何体，如下图所示的人脸模型。这个不难理解，三角形越多则相应的网格模型精度越高，这被称为高模（反之则是低模）。



网格模型基本上是通过设计师相关人员手动制作，当然也有一些通过程序生成。模型的基本单元就是一个面片，三角网格的基本单元就是三角形。三角形的三个顶点除了存储空间位置之外，通常还要给出各个顶点的法线向量、uv纹理坐标等等。美术人员制作好模型之后将按照一定的格式存储到文件当中，程序需要的时候再从文件中读取出来。目前一种简单、清晰的网格模型文件就是`.obj`后缀的模型文件（这里不是指c++编译得到的中间文件），这种模型的格式非常容易解析，你不妨以记事本的方式打开`code/obj/floor.obj`，就可以看到如下的内容：



`v`是vertex的简写，以`v`开头后面的数字是一个三维顶点位置；`vt`是纹理坐标，后面紧跟着的是二维的纹理坐标；`vn`则是vertex normal的缩写，后面跟着的数字是一个三维法线向量。而以`f`开头的则代表该行存储了一个网格面片，在这里就是三角形面片，每个面片由前面的顶点、uv坐标和法线向量组合而成，例如`3/3/1`表示该面片的顶点是`v`开头的第三行的顶点，纹理坐标是`vt`开头的第三行，法线向量是`vn`开头的第一行。**通过这种方式我们就组装了网格模型，在渲染程序中我们读取网格模型的所有三角形，将每个三角形经过顶点着色器处理然后光栅化，最后执行正确的z-buffering，就能绘制出三维的物体形状。**关于网格模型的介绍就到这里，除了`.obj`模型，还有其他各种各样的格式的模型文件，这在相关的游戏引擎中能经常看得到。

2、代码框架

打开`code`目录，你应该能看到如下的几类文件：

- `obj`目录：存储要用到的三角网格模型文件及其相应的纹理图片（`.tga`后缀格式）；
- `geometry.h`和`geometry.cpp`：该代码文件实现了一个简单的线性代数库；
- `model.h`和`model.cpp`：该代码文件实现了`.obj`三角网格模型的加载读取；
- `our_gl.h`和`our_gl.cpp`：该代码文件实现了矩阵变换、光栅化和z-buffering算法；
- `tgaimage.h`和`tgaimage.cpp`：该代码文件实现了`.tga`文件的加载读取；
- `main.cpp`：程序的启动点和主要的循环逻辑；
- `Makefile`：程序编译的格式说明文件，基本上不用管。

先来看`main`函数的主要逻辑，首先一开始就是调用相关的函数进行初始化，主要负责模型加载、创建着色器、创建缓冲区、变换矩阵、光照设定等：

```
    if (2>argc) {
        std::cerr << "Usage: " << argv[0] << " obj/model.obj" << std::endl;
        return 1;
    }

    // load the models from .obj
    for (int m=1; m<argc; m++) {
        models.push_back(new Model(argv[m]));
    }

    // the shader for rendering.
    Shader shader;

    // z-buffer, framebuffer.
    float *zbuffer = new float[width*height];
    TGAImage frame(width, height, TGAImage::RGB);

    // settings of model, camera and projection matrices.
    viewport(width/8, height/8, width*3/4, height*3/4);
    lookat(eye, center, up);
    projection(-1.f/(eye-center).norm());

    // light direction.
    light_dir = proj<3>((Projection*ModelView*embed<4>(light_dir,
0.f))).normalize();
```

然后进入渲染循环，在渲染循环中，我们首先对缓冲区进行清理，避免上一次渲染的残留影响；然后设置视图矩阵、投影矩阵；紧接着对于每一个模型的每一个面片的每一个顶点，调用顶点着色器做相关的处理，经过顶点着色器之后，再调用`triangle`函数进行三角形光栅化，在光栅化函数中产生一系列的片元（可以理解为像素，但它不单单是像素，因为它还存储了其他的几何信息，这些几何信息通过线性插值得到），调用片元着色器计算每个片元的颜色，然后写入缓冲当中：

```
    int key = 0;
    int frame_count = 0;
    while(key != 27)
    {
        // clear the framebuffer before rendering, and so does z-buffer.
        frame.clear();
        for (int i=width*height; i--; zbuffer[i] = -
std::numeric_limits<float>::max());

        // TODO 4: rotate the eye around y-axis.
        {
            // do something to variable "eye" here to achieve rotation.

        }
        lookat(eye, center, up);
        projection(-1.f/(eye-center).norm());

        // rendering per model.
        for (unsigned int m=0;m<models.size(); ++m) {
            // for each triangle of model,
            // call vertex shader, rasterization and fragment shader to render
it.
            for (int i=0; i<models[m]->nfaces(); i++) {
                // vertex shader.
                for (int j=0; j<3; j++) {
                    shader.vertex(m, i, j);
                }
                // rasterization, z-buffering and fragment shader.
                triangle(m, shader.varying_tri, shader, frame, zbuffer);
            }
        }

        // display the rendered image.
        frame.flip_vertically(); // to place the origin in the bottom left
corner of the image
        cv::Mat image(width, width, CV_8UC3, frame.buffer());
        cv::imshow("Lighting", image);

        // key event.
        key = cv::waitKey(10);

        std::cout << "frame count: " << frame_count++ << '\n';
    }
```

最后就是收尾的清理工作，并将最后渲染的结果保存成图片。

```
    frame.write_tga_file("rendered.tga");

    // release the models
    for (unsigned int i = 0; i < models.size();++i) {
        delete models[i];
    }
    std::vector<Model*>().swap(models);
    delete [] zbuffer;
```

整个流程相信大家也不难理解，具体细节大家看源代码。

3、编译

如果使用自己的系统而不是之前提供的虚拟机，本次程序作业中使用到的库为`OpenCV`，请确保该库的配置正确。如果使用我们提供的虚拟机，请在终端(命令行)以此输入以下内容：

```
make                                        // 通过make编译代码
./main obj/diablo3_pose.obj obj/floor.obj // 运行编译生成的可执行文件
```

这次没有**cmake**，已经有现成的Makefile文件，你只需在终端输入`make`即可编译，编译过程的一些warning请直接无视。编译得到的`main.o`就是最终的可执行文件，在命令行输入`./main`即可运行该可执行文件。**但我们的程序还要求输入至少一个参数，该参数指明待渲染的模型文件的相对路径**，上面命令中的`obj/diablo3_pose.obj`和`obj/floor.obj`分别指出了当前代码中`obj`目录下的`diablo3_pose.obj`文件和`floor.obj`，这两个分别是暗黑破坏神三中的怪物模型、简单的地板模型。给出这些模型的路径之后，程序会解析这两个模型文件，从中读取三角形顶点、法线、纹理坐标等相关信息。

执行程序要输入`./main obj/diablo3_pose.obj obj/floor.obj`这个命令有点长，你在终端输入一次之后，下一次运行可以通过按**上**方向键找到上一次的终端命令（再按一次就是上上次的）。

4、作业描述与提交

（1）、作业提交

将PDF报告文件和代码压缩打包提交到zhangzk3@mail2.sysu.edu.cn邮箱，邮件名和压缩包命名格式为：hw4_姓名_学号。

（2）、作业描述

本次作业要求同学们完成的工作如下所示：

Task 1、在`Shader`类中的`fragment`函数中实现漫反射光照的计算逻辑。（30分）

给出的代码框架已经帮你们计算好了法线向量、光照方向、当前片元在世界空间下的位置、视线向量等等，你只需利用这些去计算漫反射系数即可。给出的代码框架没有计算任何的光照，直接输出片元的纹理颜色：

```
    virtual bool fragment(unsigned int index, Vec3f pos, Vec3f bar, TGAColor
&color) {
```

```
        ......
        // use these vectors to finish your assignment.
        // the normal vector of this fragment.
        Vec3f normal = (B*models[index]->normal(uv)).normalize();
        // the lighting direction.
        Vec3f lightDir = light_dir;
        // the position of this fragment in world space.
        Vec3f fragPos = pos;
        // the viewing direction from eye to this fragment.
        Vec3f viewDir = (eye - fragPos).normalize();
        // the texture color of this fragment (sample from texture using uv
coordinate)
        TGAColor fragColor = models[index]->diffuse(uv);

        ......

        return false;
    }
```

实现好漫反射之后，请比较并贴出实现前后的渲染结果差距。

Task 2、在`Shader`类中的`fragment`函数中实现Phong的镜面光照计算逻辑。（30分）

在前面的基础上进一步计算镜面反射系数，**比较并贴出高光系数p取值为4、16、64的渲染结果差距，并分析p取值大小是如何影响高光区域。**

Task 3、在`Shader`类中的`fragment`函数中实现Blinn-Phong的镜面光照计算逻辑。（30分）

在上一步骤的基础上，实现Blinn-Phong的高光，其实只需改一个小小的地方就行了。**比较并贴出Phong高光和Blinn-Phong高光的渲染结果，并回答Blinn-Phong高光为什么要做这个改进?**

Task 4、在`main`函数中实现`eye`绕着`y`旋转的动画。（10分）

相信经过前面的步骤，你已经得到了炫酷的渲染效果，**现在请你实现一个动画。将`eye`绕着`y`轴每帧旋转一定的角度（例如5度）。**（提示：请回顾绕`y`旋转的矩阵，设每帧旋转的为`angle`，实现的旋转函数为`rotate_y`，在渲染循环里面令`eye=rotate_y(angle)`，这样每帧就会旋转一定的角度。注：由于目前实现的算法没有经过任何的多线程优化和GPU硬件优化，因此速度很慢，实现出来的动画看起来卡成ppt也很正常）

Task 5、（选做）在OpenGL中实现传统的光照模型。（50分）

到目前为止，我们提供的代码框架都是在软件层面上模拟现代图形渲染管线的整个流程，没有涉及到任何的图形渲染API（例如OpenGL、DirectX3D和Vulkan）等，也没有涉及到任何的GPU优化，一切都是在软件层面上模拟渲染流程，这对于理解整个渲染管线是非常有帮助的。但在开发游戏、VR和其他一些图形应用程序上，一般都是使用图形API，这些图形API配合GPU底层硬件构建了渲染管线的基本模块（例如光栅化它直接集成在了GPU硬件上，不需要再自己手动去实现），因此深入学习这些API的使用很有必要。本课程的核心在于让同学们理解图形学的核心原理，不教授API的使用，请同学们课后按照

教程去学。**这个选做作业是让同学们按照这个系列的教程去实现OpenGL中的光照计算，相信了解了图形学基本原理之后，学习OpenGL肯定会很快。**