实验三

实验目的

- 理解三维模型进行贴图，光照，纹理，阴影的过程
- 修改完成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
- 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
- 修改函数 phong_fragment_shader() in main.cpp: 实现 Blinn-Phong 模型计算 Fragment Color
- 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
- 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.
- 修改函数 displacement_fragment_shader() in main.cpp: 在实现 Bump mapping 的基础上,实现 displacement mapping
- 提高部分：
  - 尝试更多模型: 找到其他可用的.obj 文件,提交渲染结果并把模型保存在 /models 目录下。
  - 双线性纹理插值: 使用双线性插值进行纹理采样, 在 Texture类中实现一个新方法 Vector3f getColorBilinear(float u, float v) 并通过 fragment shader 调用它。为了使双线性插值的效果更加明显,你应该考虑选择更小的纹理图。请同时提交纹理插值与双线性纹理插值的结果,并进行比较。

实验过程

1. 修改完成rasterize_triangle(const Triangle& t) in rasterizer.cpp

   回忆之前在实验二中写的rasterize_triangle函数，只实现了深度缓存值的计算，这里我们需要完成对法向量、颜色、纹理颜色与底纹颜色 (Shading Colors) 进行插值，借用的都是顶点值对内部某点进行插值，并且参数还是基于重心坐标系下算出的alpha, beta, gamma，插值函数已经为我们写好了，所以直接调用就好。

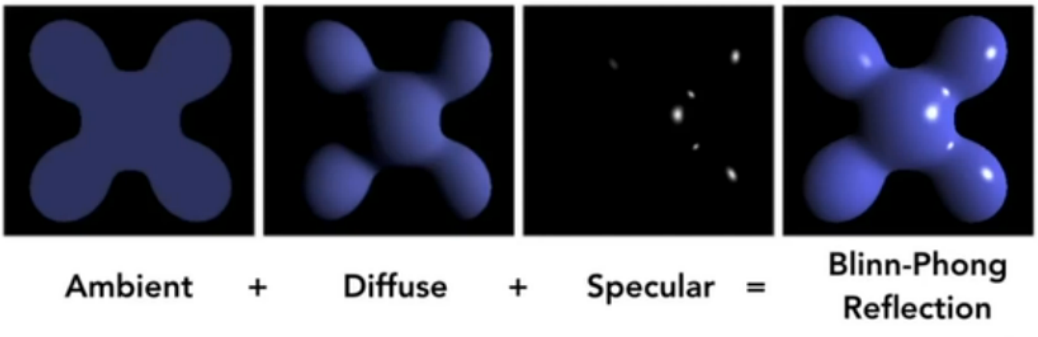
```
void rst::rasterizer::rasterize_triangle(const Triangle &t, const
std::array<Eigen::Vector3f, 3> &view_pos)
{
    // TODO : Find out the bounding box of current triangle.
    auto v = t.toVector4();
    int xmin = std::min(std::min(v[0].x(), v[1].x()), v[2].x());
    int xmax = std::max(std::max(v[0].x(), v[1].x()), v[2].x());
    int ymin = std::min(std::min(v[0].y(), v[1].y()), v[2].y());
    int ymax = std::max(std::max(v[0].y(), v[1].y()), v[2].y());
    for(int x = xmin; x<=xmax; x++)
    {
        for(int y = ymin; y<=ymax; y++)
        {
            if(insideTriangle(x+0.5, y+0.5, t.v))
            {
                float alpha, beta, gamma;
```

```
                std::tie(alpha, beta, gamma) = computeBarycentric2D(x+0.5,
y+0.5, t.v); // 函数返回值是一个tuple元组, 可以使用std::tie接收
                // 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
                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; // zp是z-buffer中存的深度值
                // TODO: Interpolate the attributes:
                // auto interpolated_color
                // auto interpolated_normal
                // auto interpolated_texcoords
                // auto interpolated_shadingcoords

                // 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);
                if(zp < depth_buf[get_index(x, y)])
                {
                    depth_buf[get_index(x, y)] = zp;
                    float weight = 1; // 这里权重是什么作用
                    Vector3f interpolated_color = interpolate(alpha, beta,
gamma, t.color[0], t.color[1], t.color[2], weight);
                    Vector3f interpolated_normal = interpolate(alpha, beta,
gamma, t.normal[0], t.normal[1], t.normal[2], weight);
                    Vector2f interpolated_texcoords = interpolate(alpha,
beta, gamma, t.tex_coords[0], t.tex_coords[1], t.tex_coords[2], weight);
                    Vector3f interpolated_shadingcoords = interpolate(alpha,
beta, gamma, view_pos[0], view_pos[1], view_pos[2],weight); // 阴影和视点有关所
以用视点插值
                    fragment_shader_payload payload( interpolated_color,
interpolated_normal.normalized(), interpolated_texcoords, texture ? &*texture
: nullptr); // 构造函数
                    payload.view_pos = interpolated_shadingcoords;
                    Vector3f pixel_color = fragment_shader(payload);
                    Vector2i point = {x, y};
                    // set_pixel(point, pixel_color);
                    set_pixel(point, pixel_color);
                }
            }
        }
    }
}
```

插值完之后，还需要将制全部赋给类fragment_shader_payload下的对象payload（使用构造函数的方法），然后先将payload传给fragment_shader函数，计算出最终的颜色，最后将颜色传给像素点。

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
3. 修改函数 phong_fragment_shader() in main.cpp

   课堂上讲的Blinn-Phong光照模型包含镜面反射，漫反射，环境光三个部分，计算公式：



$$
\begin{aligned}
L &= L_a + L_d + L_s \\
&= k_a\, I_a + k_d\, (I/r^2)\max(0, \mathbf{n}\cdot\mathbf{l}) + k_s\, (I/r^2)\max(0, \mathbf{n}\cdot\mathbf{h})^p
\end{aligned}
$$

```
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}; // 视点方向

    float p = 150;

    Eigen::Vector3f color = payload.color; // 渲染点颜色
    Eigen::Vector3f point = payload.view_pos; // 渲染点位置
    Eigen::Vector3f normal = payload.normal; // 渲染点法向量

    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.
        Vector3f I = (light.position - point).normalized(); // 光源位置到渲染点
颜色, normalized()表示单位化
        Vector3f r = light.position - point;
        Vector3f Ld =
kd.cwiseProduct(light.intensity/r.dot(r))*std::max((float)0,
normal.normalized().dot(I)); // cwiseProduct()函数作用是实现两个向量对应位置直接相
乘
        Vector3f v = (eye_pos - point).normalized();
```

```
        Vector3f h = (v + I).normalized();
        Vector3f Ls =
ks.cwiseProduct(light.intensity/r.dot(r))*std::pow(std::max((float)0,
normal.normalized().dot(h)), p);
        Vector3f La = ka.cwiseProduct(amb_light_intensity);
        result_color += (Ld + Ls + La);
    }

    return result_color * 255.f;
}
```

代码的关键在于理解公式各项和代码变量的对应关系。

4. 修改函数 texture_fragment_shader() in main.cpp

   在Bling-Phong模型的基础上，将漫反射改变成纹理颜色。

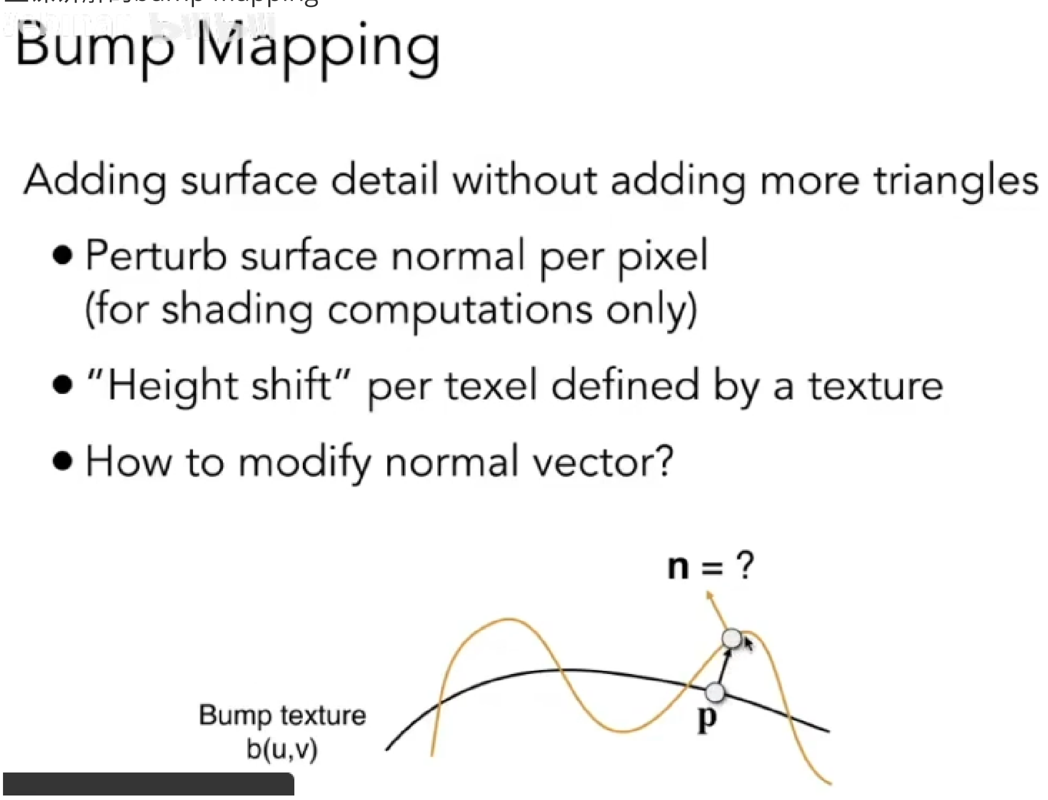
```
Eigen::Vector3f texture_fragment_shader(const fragment_shader_payload&
payload)
{
    Eigen::Vector3f return_color = {0, 0, 0};
    if (payload.texture)
    {
        // TODO: Get the texture value at the texture coordinates of the
current fragment
        return_color = payload.texture->getColor(payload.tex_coords.x(),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

    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.
```

```
        Vector3f I = (light.position - point).normalized(); // 光源位置到渲染点颜色, normalized()表示单位化
        Vector3f r = light.position - point;
        Vector3f Ld =
kd.cwiseProduct(light.intensity/r.dot(r))*std::max((float)0,
normal.normalized().dot(I)); // cwiseProduct()函数作用是实现两个向量对应位置直接相乘
        Vector3f v = (eye_pos - point).normalized();
        Vector3f h = (v + I).normalized();
        Vector3f Ls =
ks.cwiseProduct(light.intensity/r.dot(r))*std::pow(std::max((float)0,
normal.normalized().dot(h)), p);
        Vector3f La = ka.cwiseProduct(amb_light_intensity);
        result_color += (Ld + Ls + La);
    }

    return result_color * 255.f;
}
```

纹理颜色的获取借助纹理的坐标和提供的接口getColor()函数。

5. 修改函数 bump_fragment_shader() in main.cpp

上课讲解的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)
    Vector3f n =normal;
    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 = n.cross(t);
    Matrix3f TBN;
    TBN.col(0) = t;
    TBN.col(1) = b;
    TBN.col(2) = n;

    auto u = payload.tex_coords.x();
    auto v = payload.tex_coords.y();
    auto h = payload.texture->height;
    auto w = payload.texture->width;

    auto dU = kh * kn * (payload.texture->getColor(u+1.0/w,v).norm()-
payload.texture->getColor(u,v).norm());
    auto dV = kh * kn * (payload.texture->getColor(u,v+1.0/h).norm()-
payload.texture->getColor(u,v).norm());

    Vector3f ln = {-dU, -dV, 1.0f};
    normal = TBN * ln; // 矩阵乘法

    Eigen::Vector3f result_color = {0, 0, 0};
    result_color = normal;

    return result_color * 255.f;
}
```

根据官方论坛的

修改框架和书写公式即可。

6. 修改函数 displacement_fragment_shader() in main.cpp

```
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)
    Vector3f n =normal;
    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 = n.cross(t);
    Matrix3f TBN;
    TBN.col(0) = t;
    TBN.col(1) = b;
    TBN.col(2) = n;

    auto u = payload.tex_coords.x();
    auto v = payload.tex_coords.y();
    auto h = payload.texture->height;
    auto w = payload.texture->width;

    auto dU = kh * kn * (payload.texture->getColor(u+1.0/w,v).norm()-
payload.texture->getColor(u,v).norm());
    auto dV = kh * kn * (payload.texture->getColor(u,v+1.0/h).norm()-
payload.texture->getColor(u,v).norm());
    point = point + kn * n * payload.texture->getColor(u, v).norm();
```

```
    Vector3f ln = {-dU, -dV, 1.0f};

    normal = TBN * ln; // 矩阵乘法

    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.
        Vector3f I = (light.position - point).normalized(); // 光源位置到渲染点
颜色, normalized()表示单位化
        Vector3f r = light.position - point;
        Vector3f Ld =
kd.cwiseProduct(light.intensity/r.dot(r))*std::max((float)0,
normal.normalized().dot(I)); // cwiseProduct()函数作用是实现两个向量对应位置直接相
乘
        Vector3f v = (eye_pos - point).normalized();
        Vector3f h = (v + I).normalized();
        Vector3f Ls =
ks.cwiseProduct(light.intensity/r.dot(r))*std::pow(std::max((float)0,
normal.normalized().dot(h)), p);
        Vector3f La = ka.cwiseProduct(amb_light_intensity);
        result_color += (Ld + Ls + La);
    }

    return result_color * 255.f;
}
```

displacement增加了光线，展现bump模型的实际效果。

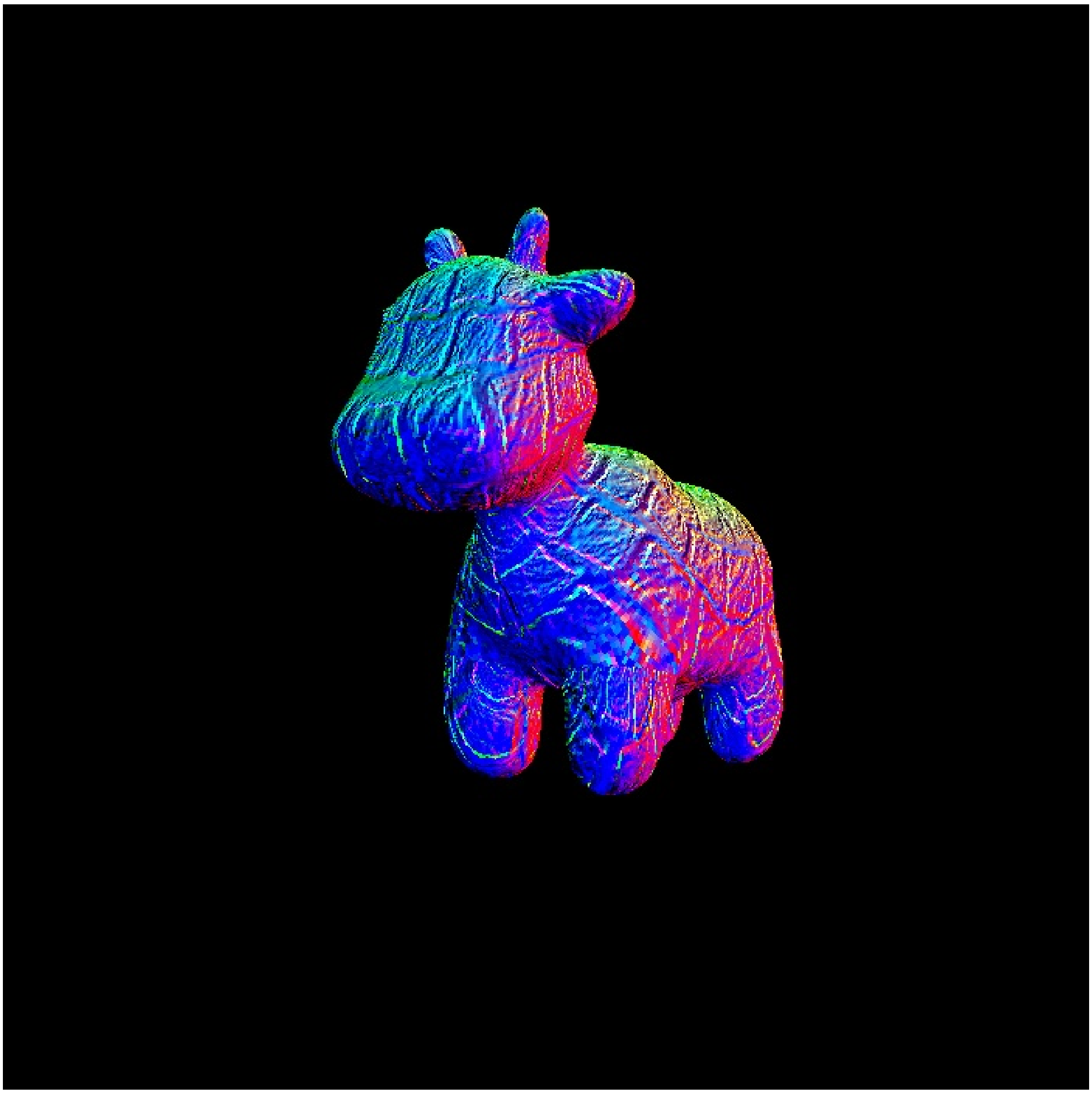
7. 执行代码：

```
mkdir build
cd build
cmake ..
make -j8
./Rasterizer output1.png texture
./Rasterizer output2.png normal
./Rasterizer output3.png phong
./Rasterizer output4.png bump
./Rasterizer output5.png displacement
```

当修改代码之后,都需要重新 make 才能看到新的结果

编译成功

```
cs18@games101vm:/mnt/hgfs/GAMES101/3/Assignment3/Code/build$ make -j8
Scanning dependencies of target Rasterizer
[ 20%] Building CXX object CMakeFiles/Rasterizer.dir/main.cpp.o
[ 40%] Linking CXX executable Rasterizer
[100%] Built target Rasterizer
```

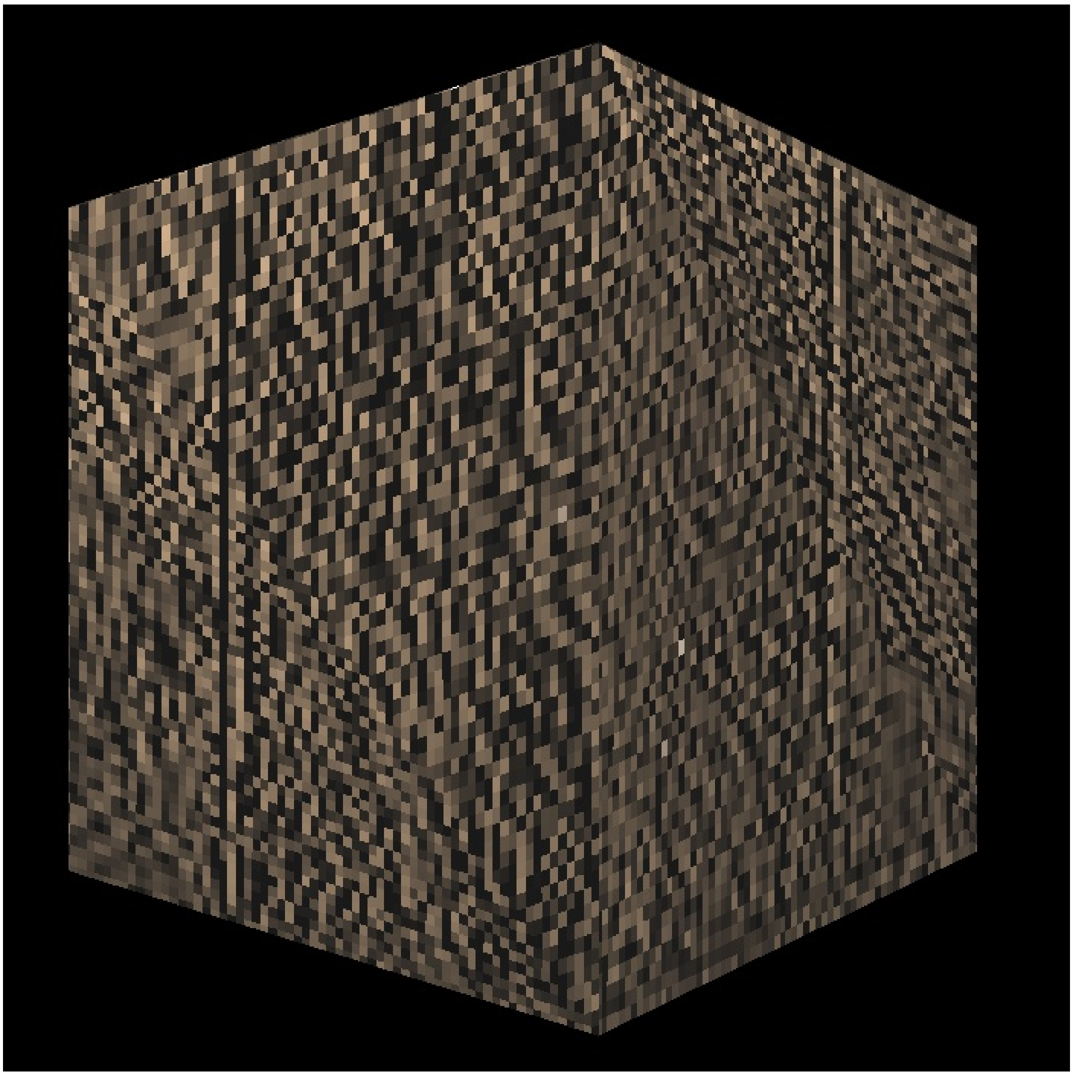


8. **提高部分**
   - 尝试更多模型: 找到其他可用的.obj 文件,提交渲染结果并把模型保存在 /models 目录下。

     根据model目录下的rock文件夹，修改main.cpp的文件存取路径：

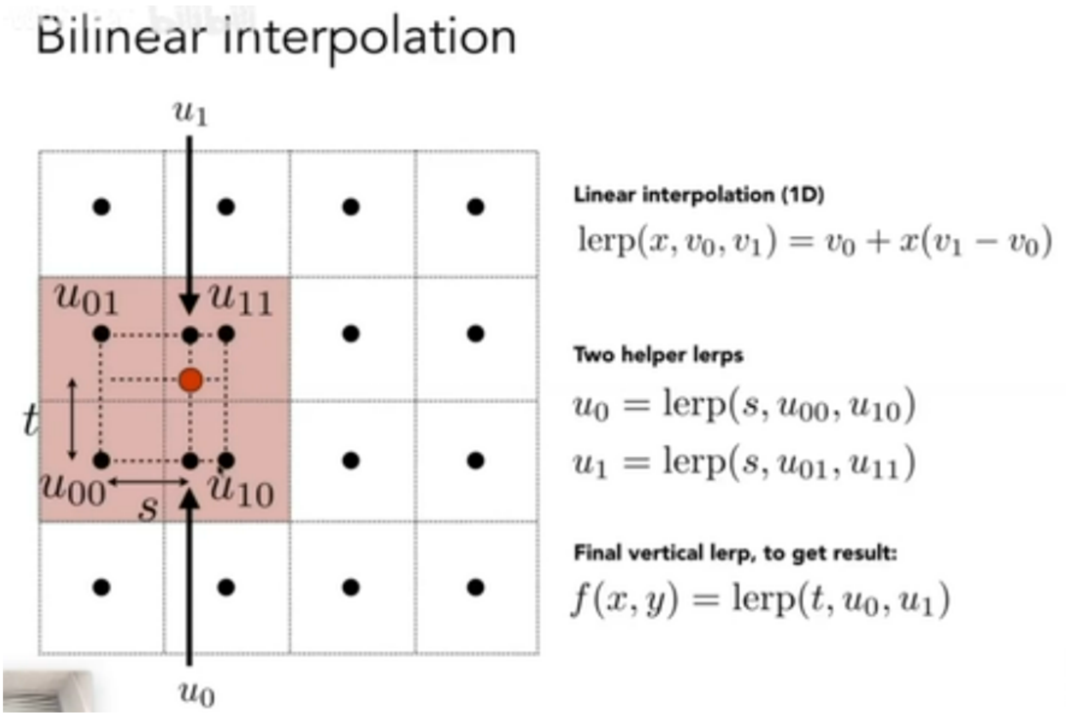
```
//std::string obj_path = "../models/spot/";
std::string obj_path = "../models/cube/";

// Load .obj File
// bool loadout =
Loader.LoadFile("../models/spot/spot_triangulated_good.obj");
bool loadout = Loader.LoadFile("../models/cube/cube.obj");

auto texture_path = "hmap.jpg";
// auto texture_path = "wall.tif";

// texture_path = "spot_texture.png";
texture_path = "wall.tif";
```

     编译执行./Rasterizer output7 texture后结果如下：



- 双线性纹理插值:

  双线性纹理插值的原理是在像素点周围取四个子像素点，根据与像素点的距离进行u方向和v方向两个方向的插值，图示如下：



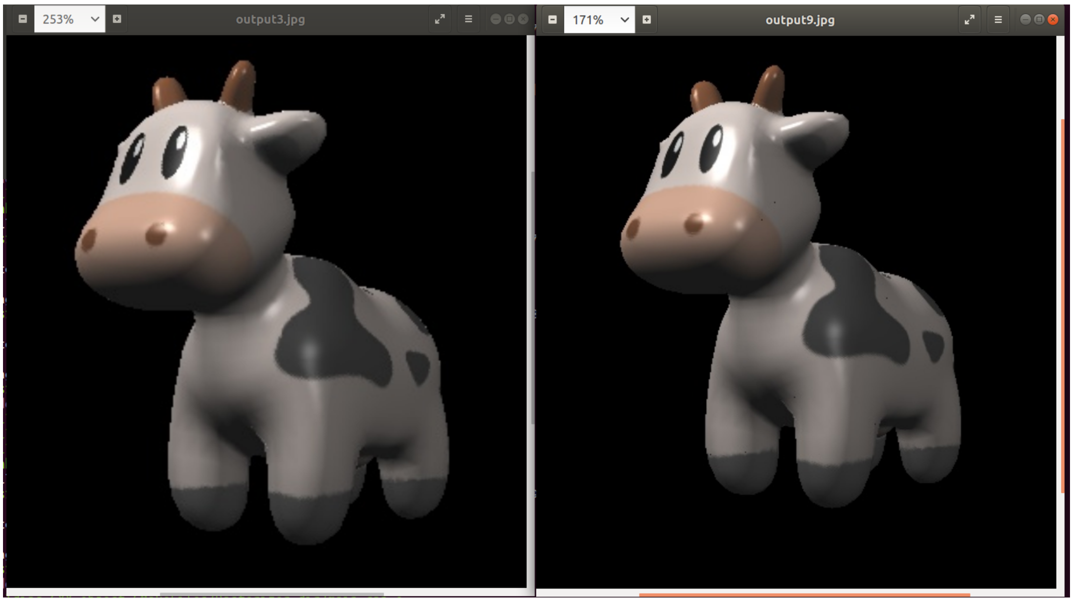

```
//提高部分
Eigen::Vector3f getColorBilinear(float u, float v)
{
```

```
        // 限制u,v制为0~1之间，不然会出现指针越界
        if(u<0) u=0;
        if(v<0) v=0;
        if(u>1) u=1;
        if(v>1) v=1;

        auto u_img = u * width;
        auto v_img = (1 - v) * height;
        //左下角
        auto u_img00 = std::floor(u_img);
        auto v_img00 = std::floor(v_img);
        //左上角
        auto u_img01 = std::floor(u_img);
        auto v_img01 = std::ceil(v_img);
        //右上角
        auto u_img11 = std::ceil(u_img);
        auto v_img11 = std::ceil(v_img);
        //右下角
        auto u_img10 = std::ceil(u_img);
        auto v_img10 = std::floor(v_img);

        //获取颜色
        auto color00 = image_data.at<cv::Vec3b>(v_img00, u_img00);
        auto color01 = image_data.at<cv::Vec3b>(v_img01, u_img01);
        auto color10 = image_data.at<cv::Vec3b>(v_img10, u_img10);
        auto color11 = image_data.at<cv::Vec3b>(v_img11, u_img11);

        auto color0 = (v_img - v_img00)*color01 + (v_img01 -
v_img)*color00;
        auto color1 = (v_img - v_img00)*color10 + (v_img01 -
v_img)*color11;
        auto color = (u_img - u_img00)*color1 + (u_img10 -
u_img)*color0;
        return Eigen::Vector3f(color[0], color[1], color[2]);
    }
```

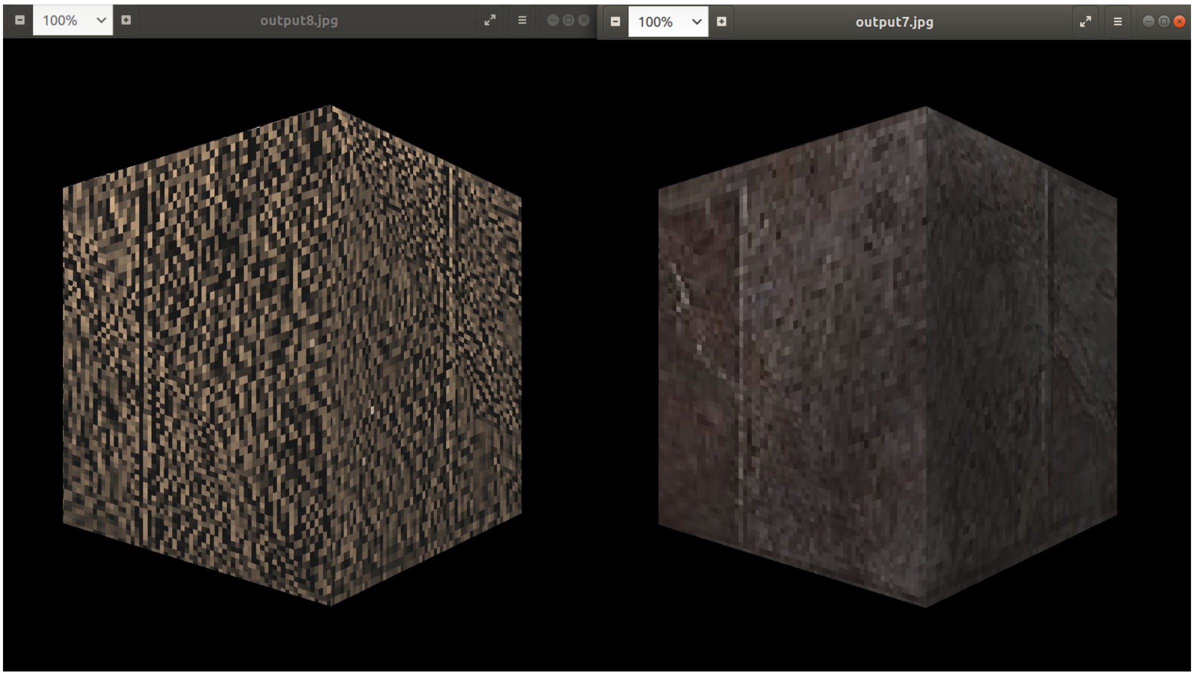
在texture_fragment_shader()函数中修改getColor为getColorBilinear以调用该函数。

```
    if (payload.texture)
    {
        // TODO: Get the texture value at the texture coordinates of the
current fragment
        return_color = payload.texture-
>getColorBilinear(payload.tex_coords.x(), payload.tex_coords.y());
    }
```

执行代码并将纹理插值和双线性纹理插值的结果进行比较：



双线性插值下，右图会更加顺滑，噪点更少，选取rock来比较会更加明显，因为rock的纹理贴图更细粒。



实验总结

实验三的难度明显上升了许多，实验指导书给的信息也很少，需要根据代码框架的注释信息和课堂的内容来写，主要都是公式的书写，代码逻辑也要有个基本的理解，实验框架本身也有问题，需要结合网上论坛修改。在书写Blinn-Phong光照模型因为公式理解的不透彻，结果和预期的颜色和高光总是有差别，经过多次Debug才解决，还学习了双线性插值的原理，并没有想象的复杂，理解原理后代码就可以自己写出来。