实验报告 3

姓名：蒋卓洋
学号：59119125

1、实现

（1）函数位置：rasterizer.cpp

- 函数实现：

① rasterize_triangle()

```
//Screen space rasterization
void rst::rasterizer::rasterize_triangle(const Triangle& t, const std::array<Eigen::Vector3f, 3>& view_pos)
{
    // TODO: From your HW3, get the triangle rasterization code.
    // 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;

    // 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);

    ////////////////Solution/////////////////
    ////Name:JiangZhuoyang
```

```
////StudentID:58119125
////FinishDate:21/10/23

 //1.COnstruct the bounding box with for value, the value is difined by the 4 extremums in to directions
//(1)Get bound
auto v = t.toVector4();
 float bound_L = std::min(v[0][0], std::min(v[1][0],v[2][0]));//Left bound: bounded by the minimum of x-coordinate of three points of triangle
float bound_R = std::max(v[0][0], std::max(v[1][0],v[2][0]));//Right bound: bounded by the maximum of x-coordinate of three points of triangle
 float bound_T = std::min(v[0][1], std::min(v[1][1],v[2][1]));//Top bound: bounded by the minimum of y-coordinate of three points of triangle
 float bound_B = std::max(v[0][1], std::max(v[1][1],v[2][1]));//Bottom bound: bounded by the maximum of y-coordinate of three points of triangle
//(2)Nomalize to integer for iteration
bound_L = (int)std::floor(bound_L);//round down the left bound
bound_R = (int)std::ceil(bound_R); //round up the right bound
bound_T = (int)std::floor(bound_T);//round down the top bound
bound_B = (int)std::ceil(bound_B);  //round up the bottom bound
 //2.Iterate through the pixel in the bound box and find if the current pixel is inside the triangle
for(int x = bound_L; x <= bound_R; x++){
    for(int y = bound_T; y <= bound_B; y++){
//(1)Judge if the current pixel is inside the triangle

        if(insideTriangle(x+0.5, y+0.5, t.v)){
//(2)Depth interpolate:
            //A.define the min depth, innitialize it with infinite.
            float depth_min = FLT_MAX;
            //B.calculate the Barycentric coordinate weight
            auto tuple = computeBarycentric2D(x, y, t.v);
            float alpha, beta, gamma;
            std::tie(alpha, beta, gamma) = tuple; // Debug the given method
            //C.interpolate the depth with Barycentric coordinate weight:
            //a.normalization: satisfies "alpha + beta + gamma = 1"??????????????????????
            float w_reciprocal = 1.0/(alpha / v[0].w() + beta / v[1].w() + gamma / v[2].w());
            //b.
             float z_interpolated = alpha * v[0].z() / v[0].w() + beta * v[1].z() / v[1].w() + gamma * v[2].z() / v[2].w();
            z_interpolated *= w_reciprocal;
            //D.get the min depth
            depth_min = std::min(depth_min,z_interpolated);
```

```
    //(3)Interpolate the attributes under the premise of depth
            if(depth_min < depth_buf[get_index(x,y)]){//the current point is more shallow
            //A.Renew the z-buffer with current point
            depth_buf[get_index(x,y)] = depth_min;

            //B.INTERPOLATE!!
            //a.color
             auto interpolated_color = alpha * t.color[0] + beta * t.color[1] + gamma * t.color[2];
            //b.normal
            auto interpolated_normal = alpha * t.normal[0] + beta * t.normal[1] + gamma * t.normal[2];
            //c.texture
            auto interpolated_texcoords = alpha * t.tex_coords[0] + beta * t.tex_coords[1] + gamma * t.tex_coords[2];
            //d.shadingcoords
             auto interpolated_shadingcoords = alpha * view_pos[0] + beta * view_pos[1] + gamma * view_pos[2];

            //C.Interpolate Result transportation
             fragment_shader_payload payload( interpolated_color, interpolated_normal.normalized(), interpolated_texcoords, texture ? &*texture : nullptr);
            payload.view_pos = interpolated_shadingcoords;
            auto pixel_color = fragment_shader(payload);

            //D.Instead of passing the triangle's color directly to the frame buffer, pass the color to the shaders first to get the final color;
               set_pixel(Vector2i(x,y), pixel_color);
            }

         }
      }
   }

}
```

（2）函数位置：main.cpp

• 函数实现：

① get_projection_matrix()

```
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.
    /////////////////Solution//////////////////
    ////Name:JiangZhuoyang
    ////StudentID:58119125
    ////FinishDate:21/9/30
    //1.Definition
    Eigen::Matrix4f perspective = Eigen::Matrix4f::Identity();//perspective projection
    Eigen::Matrix4f persp_to_ortho = Eigen::Matrix4f::Identity();//turn the perspective projection to orthographic projection
    Eigen::Matrix4f orthographic = Eigen::Matrix4f::Identity();//orthographic projection

    //2.Construction
    //2.1.P -> O
    persp_to_ortho << zNear,0,0,0,
                    0,zNear,0,0,
                    0,0,zNear+zFar,-zNear*zFar,
                    0,0,1,0;

    //2.2.Orthographic
    //(1)Get edges: implicitly, do the first translation with the use of eye fov and aspect ration directly.
    float yTop = -zNear * tan( (eye_fov/2) * MY_PI / 180 );
    float yBottom = (-1) * yTop;
    float xRight = yTop * aspect_ratio;
    float xLeft = (-1) * xRight;
    //(2)Orthographic translate:
    Eigen::Matrix4f ortho_trans = Eigen::Matrix4f::Identity();
    ortho_trans << 1, 0, 0, -(xRight + xLeft) / 2, //(1,0,0,0)
                0, 1, 0, -(yTop + yBottom) / 2, //(0,1,0,0)
                0, 0, 1, -(zNear + zFar) / 2,
                0, 0, 0, 1;
```

```
    //(3)Orthographic scale:
    Eigen::Matrix4f ortho_scale = Eigen::Matrix4f::Identity();
    ortho_scale << 2/(xRight - xLeft), 0, 0, 0, //(1,0,0,0)
                0, 2/(yTop - yBottom), 0, 0, //(0,1,0,0)
                0, 0, 2/(zNear - zFar), 0,
                0, 0, 0, 1;
    //(3)Orthographic Matrix:
    orthographic = ortho_scale * ortho_trans;

    //2.3.Perspective:
    perspective = orthographic * persp_to_ortho;

    //3.projection:
    projection = perspective;
    ////////////////////////////////////////////
    return projection;

}
```

② texture_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
        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;//fir specular light

    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.

        /////////////////Solution//////////////////
        ////Name:JiangZhuoyang
        ////StudentID:58119125
        ////FinishDate:21/10/22

        //1.Get all the vectors we need to use
        //(1).Light direction
        Eigen::Vector3f light_dir = light.position - point;
        //(2).Viewer direction
        Eigen::Vector3f viewer_dir = eye_pos - point;
        //(3).Surface normal:has had
        //(4).Half vector
        Eigen::Vector3f h = (light_dir + viewer_dir).normalized();

        //2.Difine light distance to represent the light fallof
        float r_2 = light_dir.dot(light_dir);

        //3.Calculate all 3 kings of light
        //(1).Ambient:
        Eigen::Vector3f La = ka.cwiseProduct(amb_light_intensity);
        //(2).Diffuse:
        Eigen::Vector3f Ld = kd.cwiseProduct(light.intensity / r_2) * std::max(0.0f, normal.normalized().dot(light_dir.normalized()));
        //(3).Specular:
        Eigen::Vector3f Ls = ks.cwiseProduct(light.intensity / r_2) * std::pow(std::max(0.0f, normal.normalized().dot(h)) , p);

        //4.get result of Blinn-Phong Model
        result_color += (La + Ld + Ls);
```

```
        /////////////////////////////////////////////

    }

    return result_color * 255.f;
}
```

③ phong_fragment_shader()

```
Eigen::Vector3f phong_fragment_shader(const fragment_shader_payload& payload)
{
    //Three coefficient
    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.

        /////////////////Solution//////////////////
        ////Name:JiangZhuoyang
        ////StudentID:58119125
        ////FinishDate:21/10/22

        //1.Get all the vectors we need to use
        //(1).Light direction
```

```
        Eigen::Vector3f light_dir = (light.position - point).normalized();
        //(2).Viewer direction
        Eigen::Vector3f viewer_dir = (eye_pos - point).normalized();
        //(3).Surface normal:has had
        //(4).Half vector
        Eigen::Vector3f h = (light_dir+viewer_dir).normalized();

        //2.Difine light distance to represent the light fallof
        float r_2 = (light.position - point).dot(light.position - point);

        //3.Calculate all 3 kings of light
        //(1).Ambient:
        Eigen::Vector3f La = ka.cwiseProduct(amb_light_intensity);
        //(2).Diffuse:
        Eigen::Vector3f Ld = kd.cwiseProduct((light.intensity / r_2) * std::max(0.0f, normal.dot(light_dir)));
        //(3).Specular:
        Eigen::Vector3f Ls = ks.cwiseProduct((light.intensity / r_2) * std::pow(std::max(0.0f, normal.dot(h)) , p));

        //4.get result of Blinn-Phong Model
        result_color += (La + Ld + Ls);
        ////////////////////////////////////////////
    }

    return result_color * 255.f;
}

```

④ bump_fragment_shader()

```
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)
/////////////////Solution///////////////////
////Name:JiangZhuoyang
////StudentID:58119125
////FinishDate:21/10/23
//Follow the cue
float x = normal.x(), y = normal.y(), z = normal.z();
Eigen::Vector3f t = Vector3f( x*y / std::sqrt(x*x + y*y), std::sqrt(x*x + z*z), y*z / std::sqrt(x*x + z*z) );
Eigen::Vector3f b = normal.cross(t);
Eigen::Matrix3f TBN;
TBN << t,b,normal;
float u = payload.tex_coords.x();
float v = payload.tex_coords.y();
float w = payload.texture->width;
float h = payload.texture->height;
float dU = kh * kn *( payload.texture->getColor( (u+1.0f)/w, v ).norm() - payload.texture->getColor(u,v).norm() );
float dV = kh * kn *( payload.texture->getColor( u, (v+1.0f)/h ).norm() - payload.texture->getColor(u,v).norm() );
```

```
    Eigen::Vector3f Ln{-dU,-dV,1};

    normal = (TBN *Ln).normalized();

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

    return result_color * 255.f;
}
```

2、结果

实验结果如下：

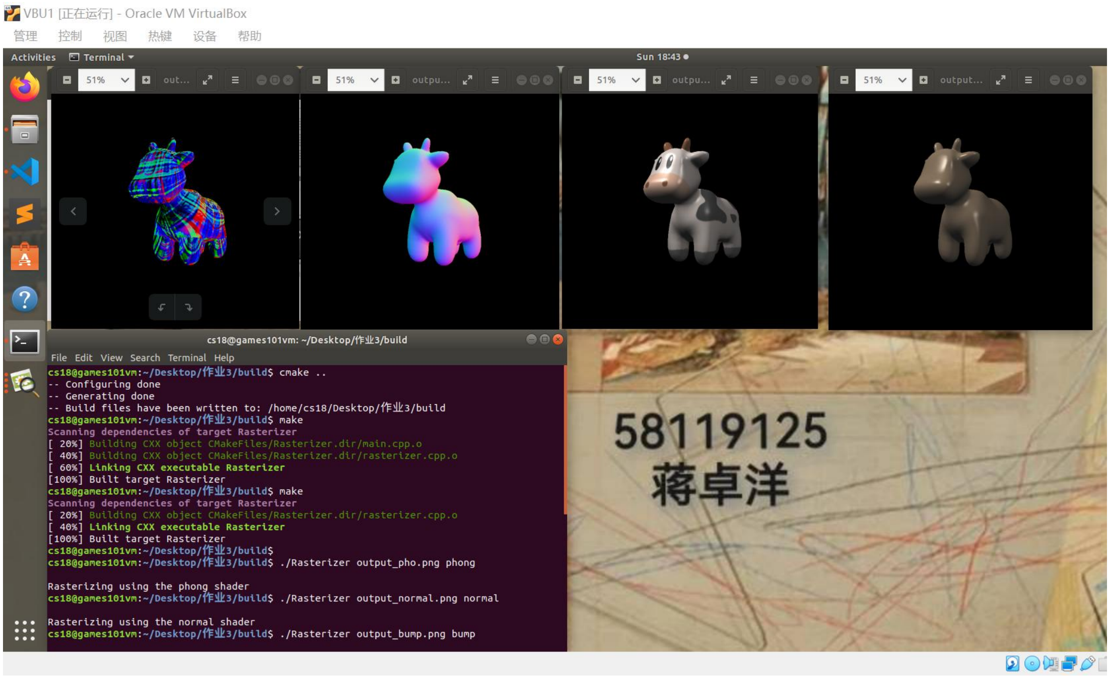

图 1.作业编译三结果

图 2.作业三结果--phong

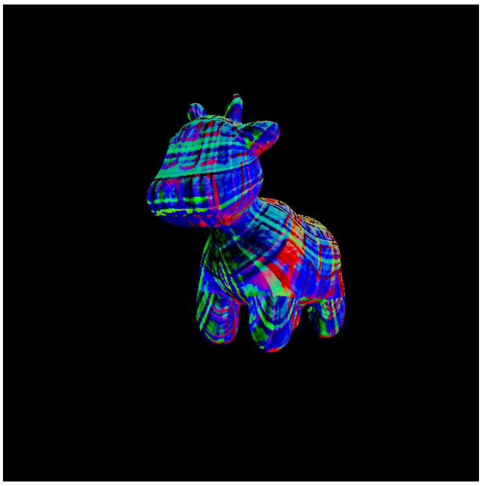

图 3.作业三结果--bump

图 4.作业三结果--normal

图 5.作业三结果--texture