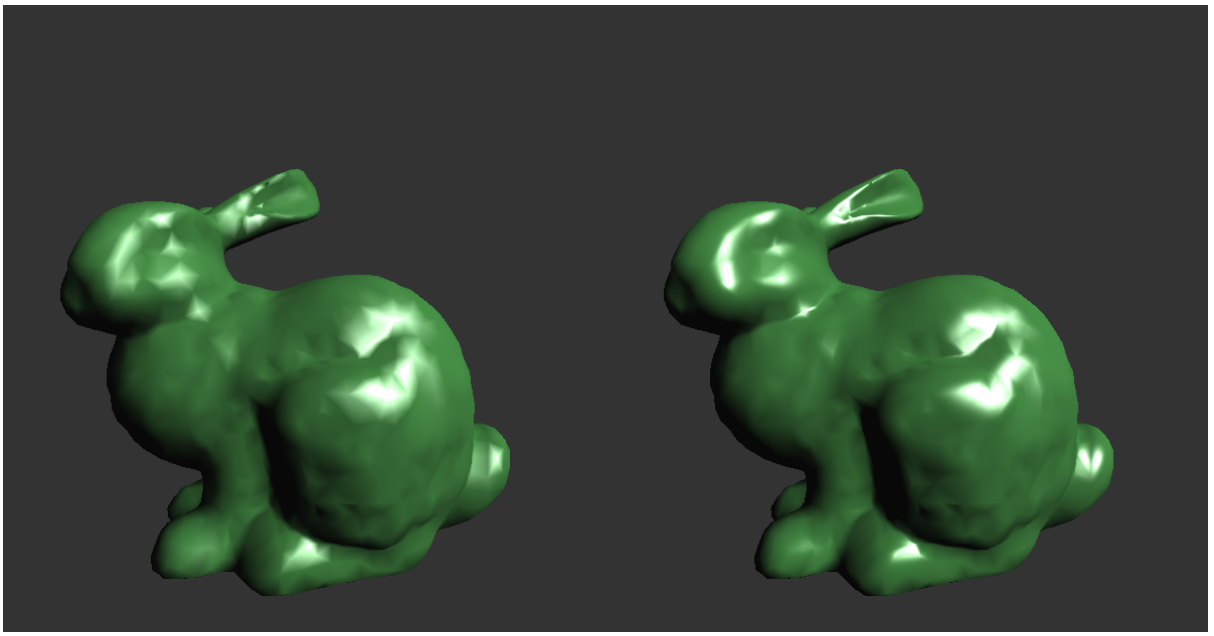
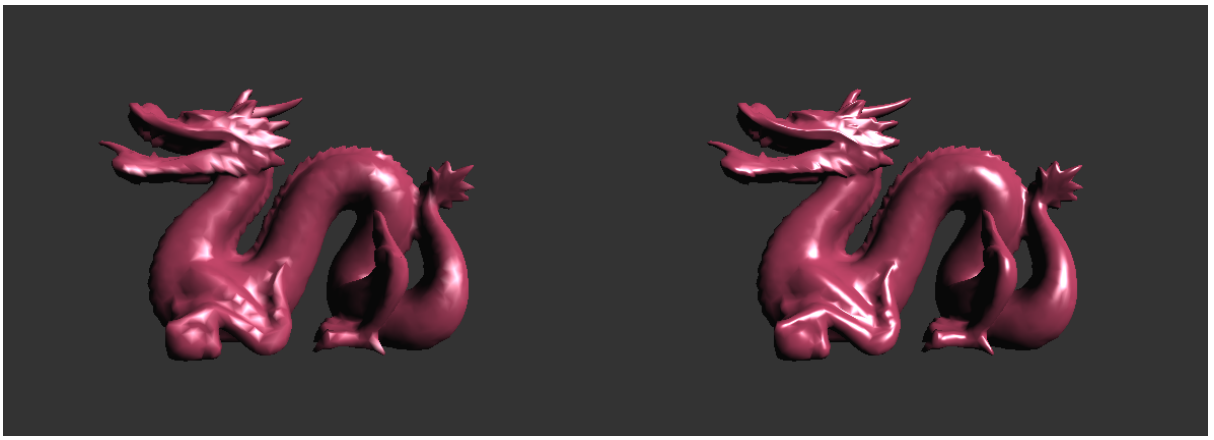
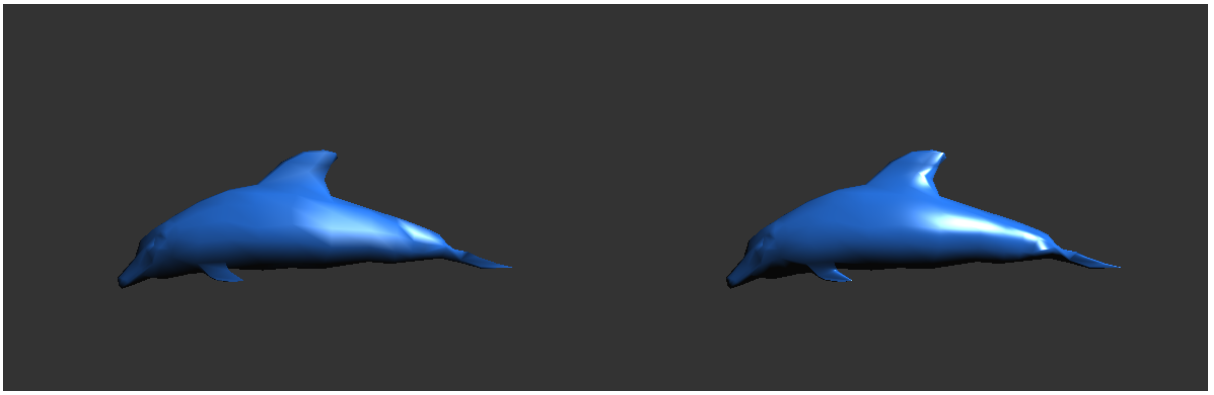
HW2_Report

109065539 韓承翰

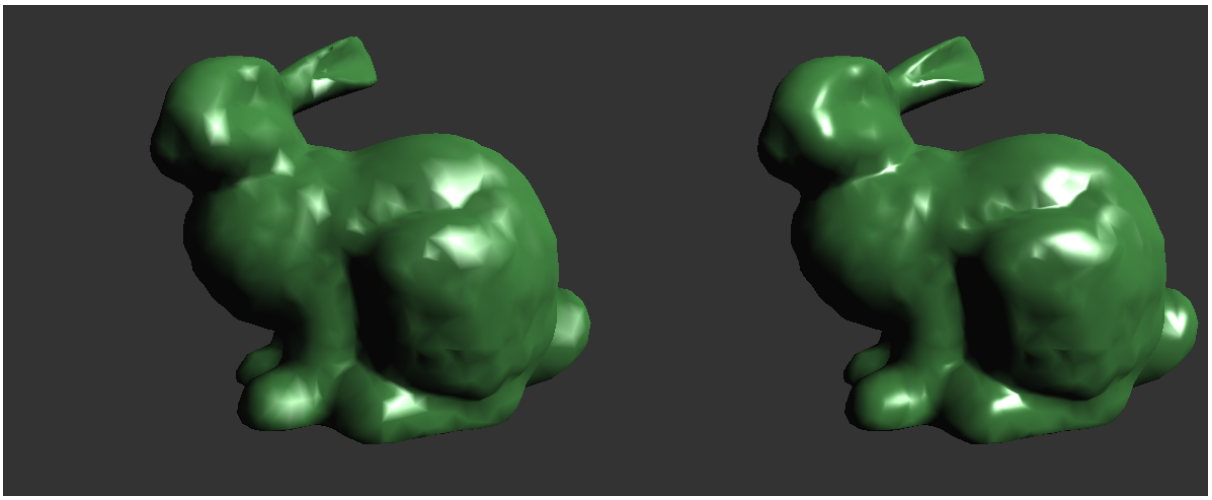
Demo:

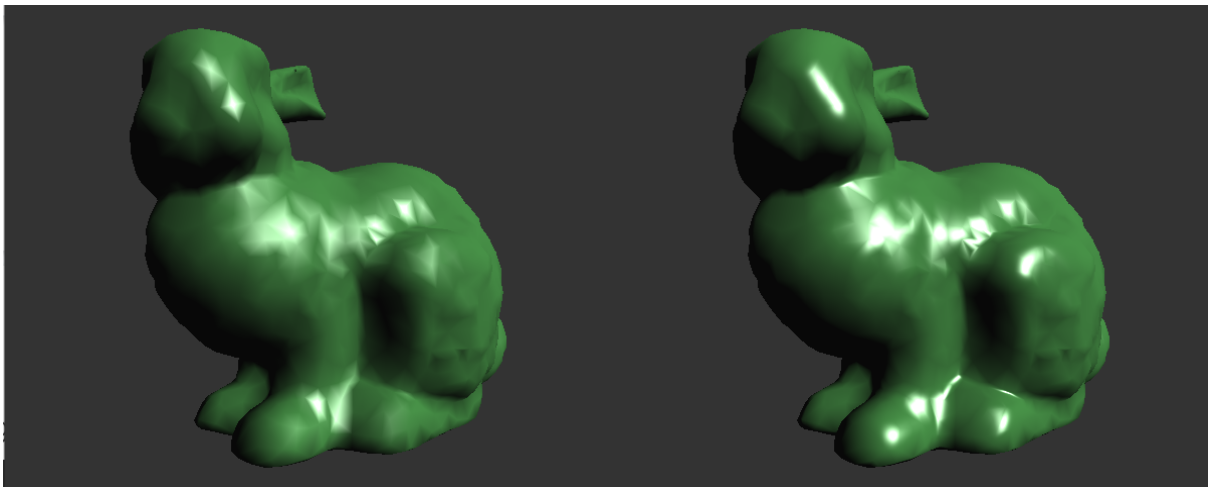


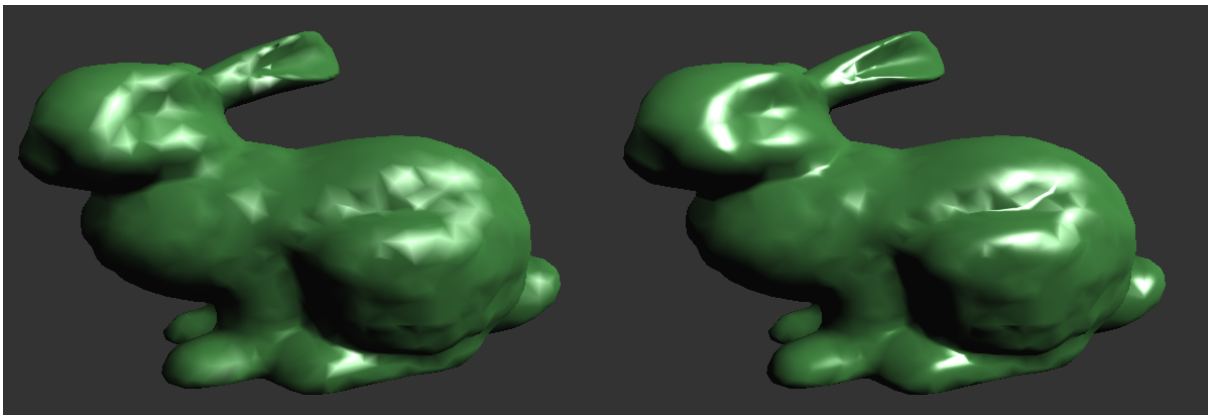




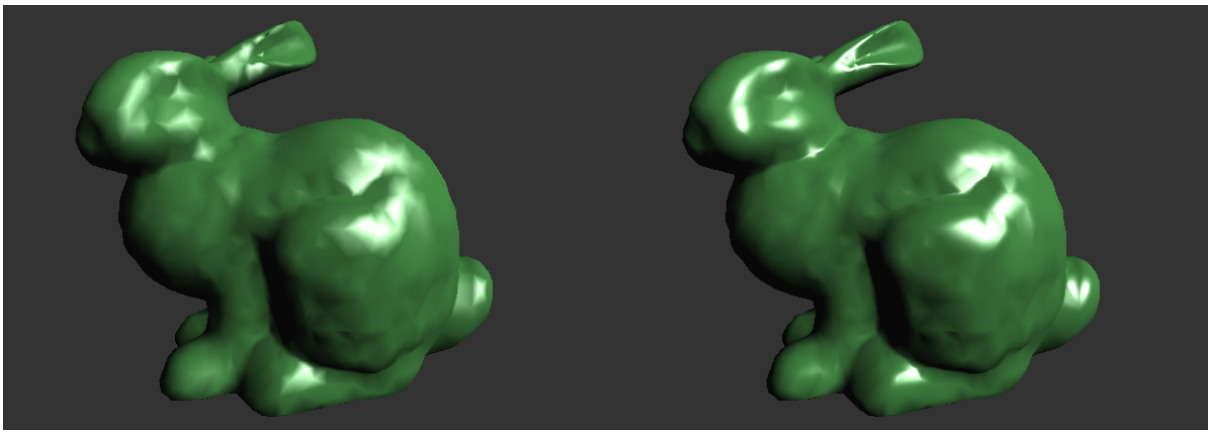
Translation, Rotation, Scaling models:

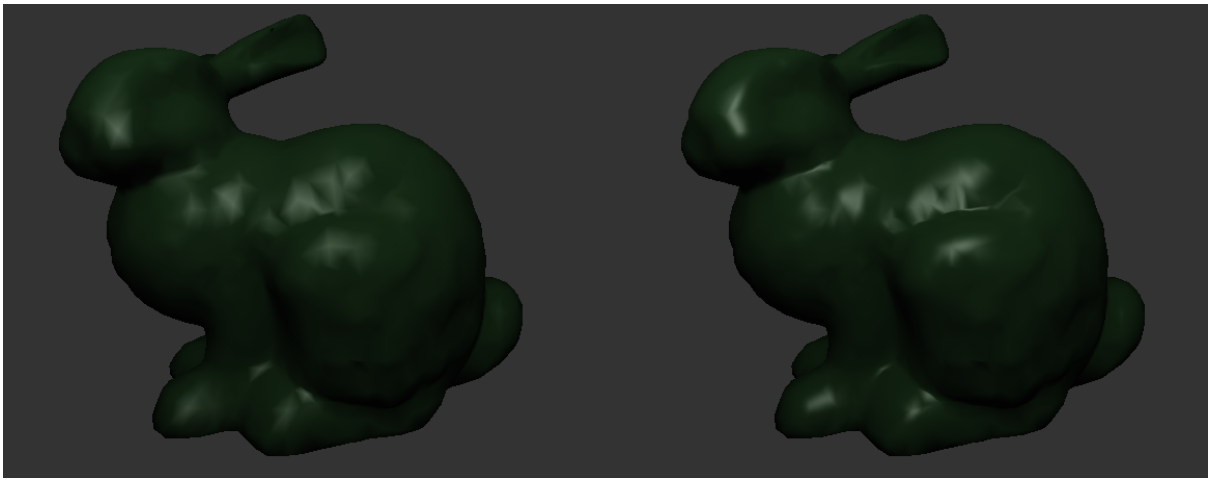


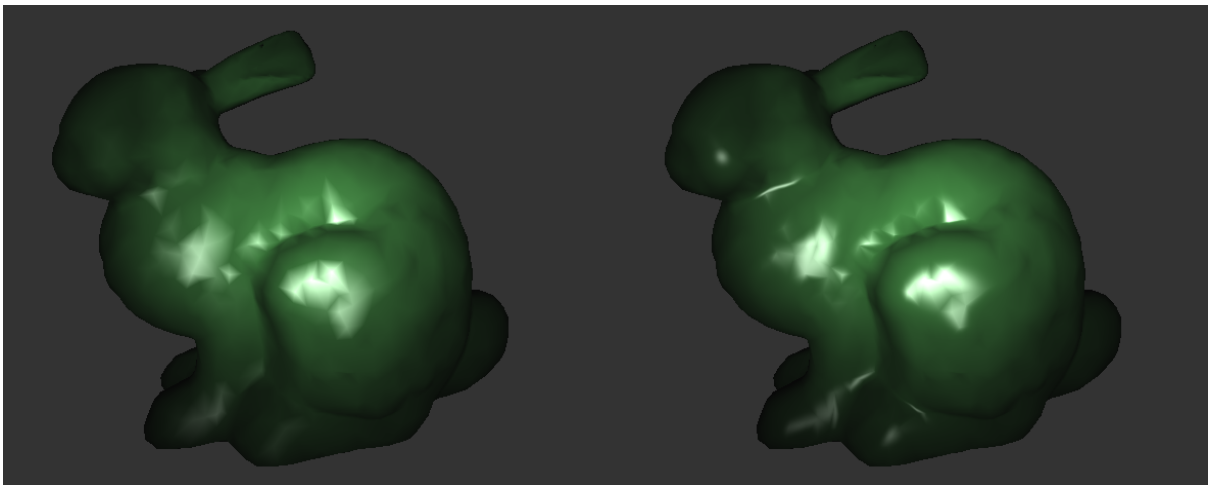




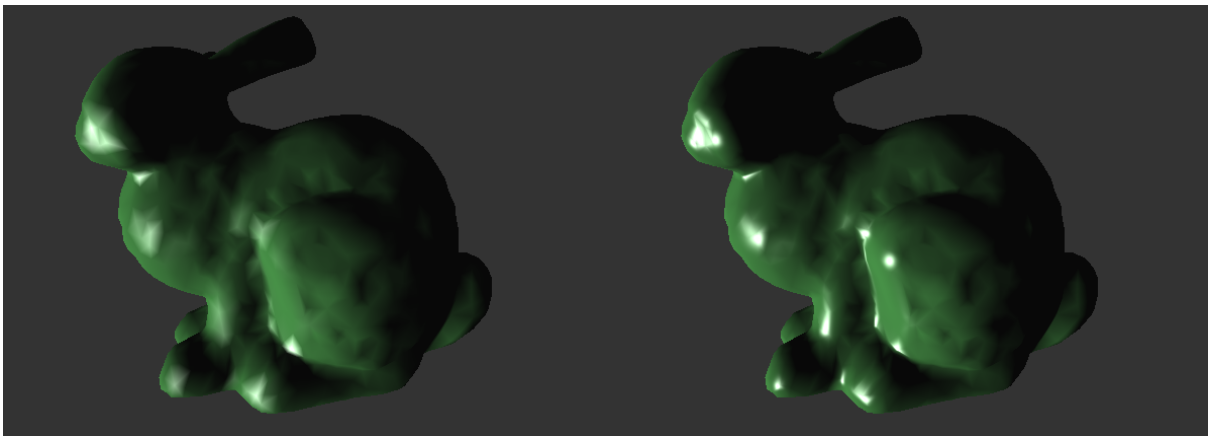
Directional/Point/Spot light:

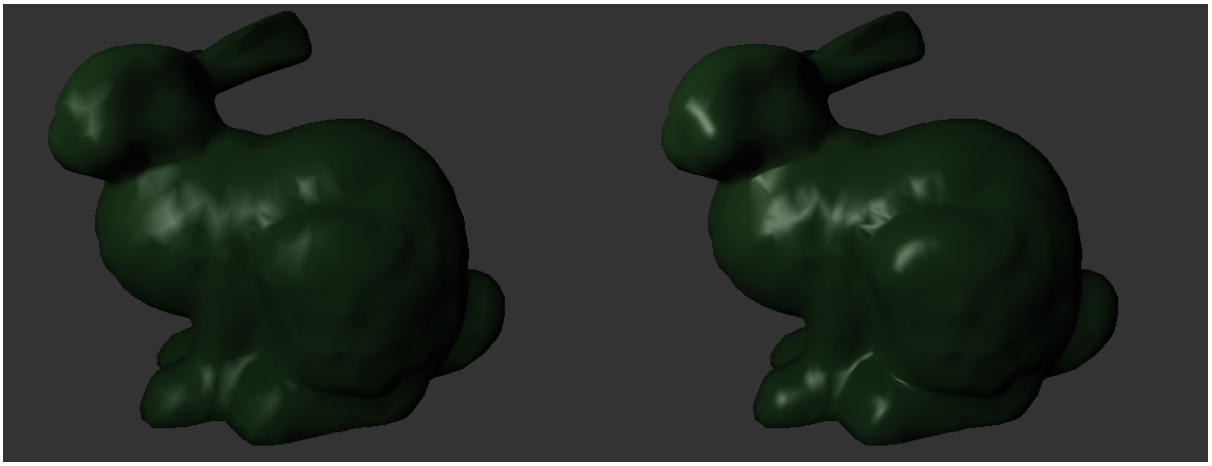


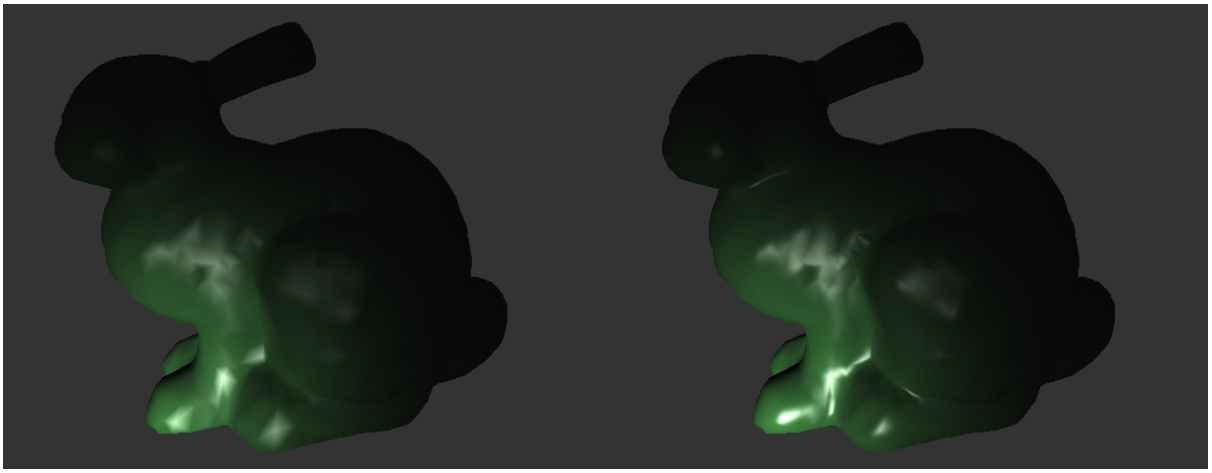




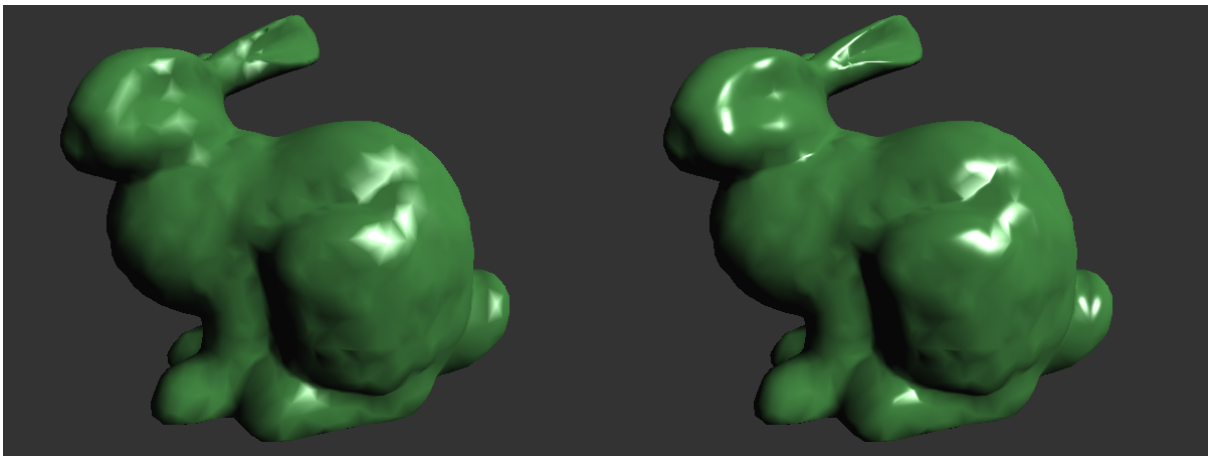
Light Editing Mode:

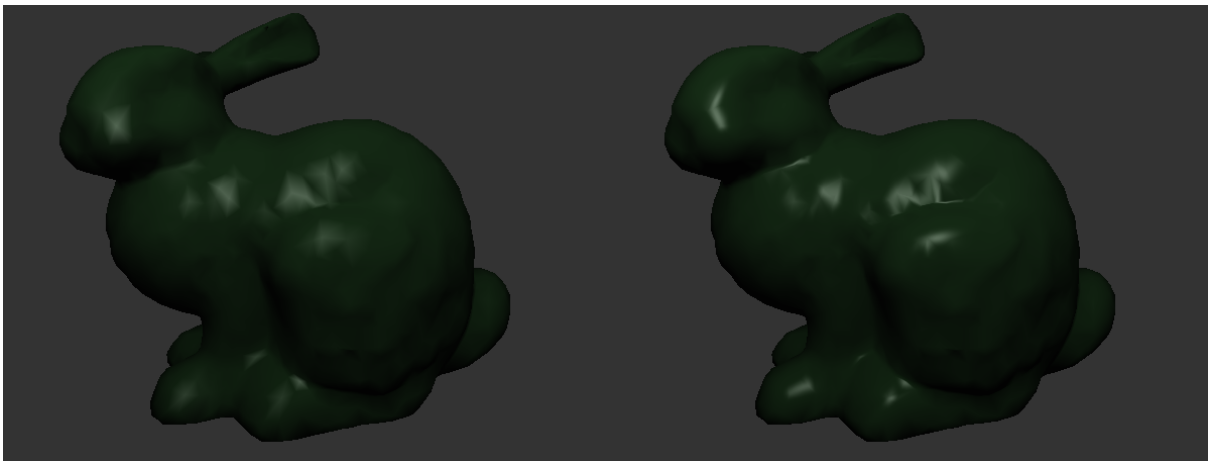


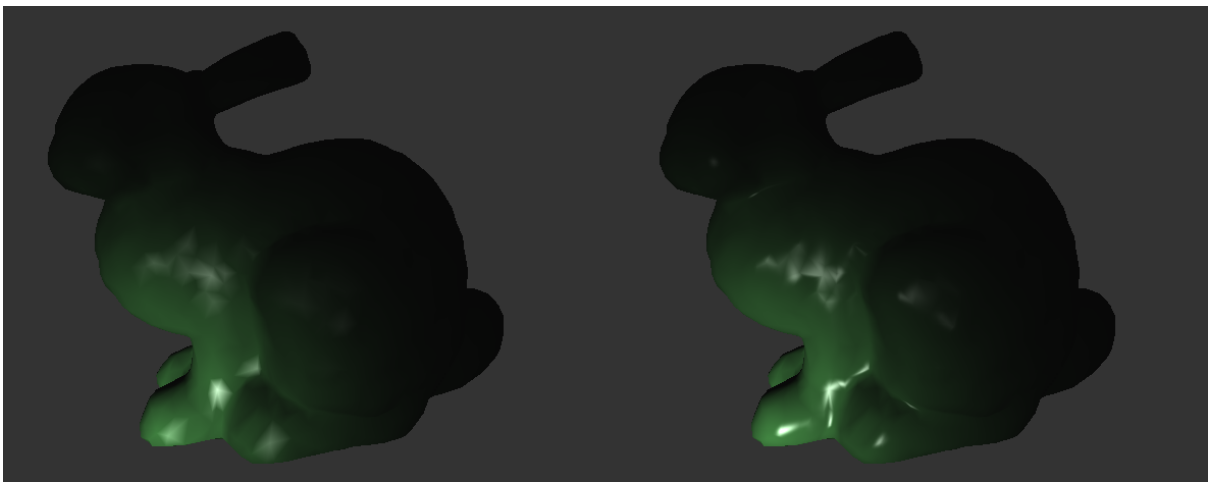




Shininess Editing Mode:







Code:

main.cpp

- 新增變數:
  - 紀錄目前螢幕長寬

```
// <SELF ADD> Record window size
GLfloat current_window_width = WINDOW_WIDTH;
GLfloat current_window_height = WINDOW_HEIGHT;
```

  - 模式

```
// <SELF ADD>
enum ModeType
{
    None = -1,
    Transformation = 0,
    Lighting = 1,
};
```

```
// <SELF ADD>
enum LightMode
{
    LightSwitch = 0,
    LightEdit = 1,
    ShininessEdit = 2
};

// <SELF ADD>
enum LightType
{
    Directional = 0,
    Point = 1,
    Spot = 2,
};
```

  - 初始模式

```
// <SELF ADD>
ModeType cur_mode_type = None;
ProjMode cur_proj_mode = Orthogonal;
TransMode cur_trans_mode = GeoTranslation;
LightMode cur_light_mode = LightSwitch;
LightType cur_light_type = Directional;
```

- 新增傳到shader的變數

```
// <SELF ADD>
GLint iLocMVP;
GLint iLocMV;
GLint iLocKa;
GLint iLocKd;
GLint iLocKs;
GLint iLocDir;
GLint iLocPos;
GLint iLocMode;
GLint iLocDiffuse;
GLint iLocSpecular;
GLint iLocAmbient;
GLint iLocShininess;
GLint iLocAngle;
GLint iLocshademode;
```

- 光的參數與類型

```
// <SELF ADD>
struct light {
    Vector3 position;
    Vector3 direction;
    Vector3 diffuse;
    Vector3 specular;
    GLint angle;
};

// <SELF ADD>
struct light_types {
    light directional;
    light point;
    light spot;
};
light_types Light;
```

- 修改function:
  - phong_model(): 由renderScene呼叫，將改變的參數傳至shader中

```
// <SELF ADD> compute phong model
void phong_model(int material_index, Matrix4 V, int shade_mode)
{
    PhongMaterial phong_material = models[cur_idx].shapes[material_index].material;
    Vector3 ka = phong_material.Ka;
    Vector3 kd = phong_material.Kd;
    Vector3 ks = phong_material.Ks;

    GLfloat Ka[3] = { ka[0], ka[1], ka[2] };
    GLfloat Kd[3] = { kd[0], kd[1], kd[2] };
    GLfloat Ks[3] = { ks[0], ks[1], ks[2] };

    light light_temp;
    switch(cur_light_type) {
        case Directional:
            light_temp = Light.directional;
            break;
        case Point:
            light_temp = Light.point;
            break;
        case Spot:
            light_temp = Light.spot;
            break;
    }

    GLfloat light_diffuse[3] = { light_temp.diffuse.x, light_temp.diffuse.y, light_temp.diffuse.z };
    GLfloat light_specular[3] = { light_temp.specular.x, light_temp.specular.y, light_temp.specular.z };
    GLfloat light_ambient[3] = { ambient.x, ambient.y, ambient.z };
    GLint light_angle = light_temp.angle;

    //change light position to view space
    Vector4 light_pos4 = Vector4(light_temp.position.x, light_temp.position.y, light_temp.position.z, 0) * V;
    GLfloat light_position[3] = { light_pos4.x, light_pos4.y, light_pos4.z };

    //change light direction to view space
    Vector4 light_dir4 = Vector4(light_temp.direction.x, light_temp.direction.y, light_temp.direction.z, 0) * V;
    GLfloat light_direction[3] = { light_dir4.x, light_dir4.y, light_dir4.z };

    // send to shader
    glUniform3fv(iLocKa, 1, Ka);
    glUniform3fv(iLocKd, 1, Kd);
    glUniform3fv(iLocKs, 1, Ks);
    glUniform3fv(iLocDir, 1, light_direction);
    glUniform3fv(iLocPos, 1, light_position);
    glUniform3fv(iLocDiffuse, 1, light_diffuse);
    glUniform3fv(iLocSpecular, 1, light_specular);
    glUniform3fv(iLocAmbient, 1, light_ambient);
    glUniform1i(iLocMode, cur_light_type);
    glUniform1i(iLocAngle, light_angle);
    glUniform1i(iLocShininess, shininess);
    glUniform1i(iLocshademode, shade_mode);
}
```

- RenderScene(): 多傳mv至shader中，與同時印出兩個模型

```
// <SELF ADD>
MV = view_matrix * (T * R * S);

mv[0] = MV[0];  mv[4] = MV[1];   mv[8] = MV[2];    mv[12] = MV[3];
mv[1] = MV[4];  mv[5] = MV[5];   mv[9] = MV[6];    mv[13] = MV[7];
mv[2] = MV[8];  mv[6] = MV[9];   mv[10] = MV[10];  mv[14] = MV[11];
mv[3] = MV[12]; mv[7] = MV[13];  mv[11] = MV[14];  mv[15] = MV[15];

// use uniform to send mvp to vertex shader
glUniformMatrix4fv(iLocMVP, 1, GL_FALSE, mvp);
glUniformMatrix4fv(iLocMV, 1, GL_FALSE, mv);
for (int i = 0; i < models[cur_idx].shapes.size(); i++)
{
    glViewport(0, 0, GLfloat(current_window_width) / 2, current_window_height);
    phong_model(i, view_matrix, 0);
    glBindVertexArray(models[cur_idx].shapes[i].vao);
    glDrawArrays(GL_TRIANGLES, 0, models[cur_idx].shapes[i].vertex_count);

    glViewport(GLfloat(current_window_width) / 2, 0, GLfloat(current_window_width) / 2, current_window_height);
    phong_model(i, view_matrix, 1);
    glBindVertexArray(models[cur_idx].shapes[i].vao);
    glDrawArrays(GL_TRIANGLES, 0, models[cur_idx].shapes[i].vertex_count);

}
```

- setShaders(): 新增傳遞至shader的變數

```
// <SELF ADD>
iLocMVP = glGetUniformLocation(p, "mvp");
iLocMV = glGetUniformLocation(p, "mv");
iLocDir = glGetUniformLocation(p, "light_dir");
iLocPos = glGetUniformLocation(p, "light_pos");
iLocKa = glGetUniformLocation(p, "ka");
iLocKd = glGetUniformLocation(p, "kd");
iLocKs = glGetUniformLocation(p, "ks");
iLocMode = glGetUniformLocation(p, "cur_light_type");
iLocDiffuse = glGetUniformLocation(p, "diffuse");
iLocSpecular = glGetUniformLocation(p, "specular");
iLocAmbient = glGetUniformLocation(p, "ambient");
iLocShininess = glGetUniformLocation(p, "shininess");
iLocAngle = glGetUniformLocation(p, "angle");
iLocshademode = glGetUniformLocation(p, "shade_mode");
```

- initParameters(): 新增light初始化參數

```
// <SELF ADD>
Light.directional.position = Vector3(1.0f, 1.0f, 1.0f);
Light.directional.direction = Vector3(0.0f, 0.0f, 0.0f);
Light.directional.diffuse = Vector3(1.0f, 1.0f, 1.0f);
Light.directional.specular = Vector3(1.0f, 1.0f, 1.0f);
Light.point.position = Vector3(0.0f, 2.0f, 1.0f);
Light.point.diffuse = Vector3(1.0f, 1.0f, 1.0f);
Light.point.specular = Vector3(1.0f, 1.0f, 1.0f);
Light.spot.position = Vector3(0.0f, 0.0f, 2.0f);
Light.spot.direction = Vector3(0.0f, 0.0f, -1.0f);
Light.spot.diffuse = Vector3(1.0f, 1.0f, 1.0f);
Light.spot.specular = Vector3(1.0f, 1.0f, 1.0f);
Light.spot.angle = 30;

shininess = 64;
ambient = Vector3(0.15f, 0.15f, 0.15f);
```

- KeyCallback

```
void KeyCallback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    // [TODO] Call back function for keyboard
    if (action == GLFW_PRESS) {
        switch (key) {
        case GLFW_KEY_Z:
            cur_idx = (cur_idx - 1 + 5) % 5;
            break;
        case GLFW_KEY_X:
            cur_idx = (cur_idx + 1) % 5;
            break;
        case GLFW_KEY_O:
            setOrthogonal();
            break;
        case GLFW_KEY_P:
            setPerspective();
            break;
        case GLFW_KEY_T:
            cur_mode_type = Transformation;
            cur_trans_mode = GeoTranslation;
            break;
        case GLFW_KEY_S:
            cur_mode_type = Transformation;
            cur_trans_mode = GeoScaling;
            break;
        case GLFW_KEY_R:
            cur_mode_type = Transformation;
            cur_trans_mode = GeoRotation;
            break;
        case GLFW_KEY_L:
            cur_mode_type = Lighting;
            cur_light_type = (LightType)((cur_light_type + 1) % 3);
            break;
        case GLFW_KEY_K:
            cur_mode_type = Lighting;
            cur_light_mode = LightEdit;
            break;
        case GLFW_KEY_J:
            cur_mode_type = Lighting;
            cur_light_mode = ShininessEdit;
            break;
        }
    }
}
```

- scroll_callback

```
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    // [TODO] scroll up positive, otherwise it would be negtive
    const float TICK = 0.01;
    if (cur_mode_type == Transformation) {
        switch (cur_trans_mode) {
        case GeoTranslation:
            models[cur_idx].position += Vector3(0, 0, TICK * yoffset);
            break;
        case GeoScaling:
            models[cur_idx].scale += Vector3(0, 0, TICK * yoffset);
            break;
        case GeoRotation:
            models[cur_idx].rotation += Vector3(0, 0, TICK * yoffset);
            break;
        case None:
            break;
        }
    }
    else if (cur_mode_type == Lighting) {
        if (cur_light_mode == LightEdit) {
            switch (cur_light_type) {
            case Directional:
                Light.directional.diffuse += Vector3(TICK * yoffset, TICK * yoffset, TICK * yoffset);
                break;
            case Point:
                Light.point.diffuse += Vector3(TICK * yoffset, TICK * yoffset, TICK * yoffset);
                break;
            case Spot:
                Light.spot.angle += yoffset;
                break;
            }
        }
        else if(cur_light_mode == ShininessEdit) {
            shininess += yoffset;
        }
    }
}
```

- cursor_pos_callback

```
static void cursor_pos_callback(GLFWwindow* window, double xpos, double ypos)
{
    // [TODO] cursor position callback function
    const float TICK = 0.01;

    if (mouse_pressed) {
        if (cur_mode_type == Transformation) {
            switch (cur_trans_mode) {
            case GeoTranslation:
                models[cur_idx].position +=
                    Vector3(TICK * (xpos - starting_press_x), -TICK * (ypos - starting_press_y), 0);
                break;
            case GeoScaling:
                models[cur_idx].scale -=
                    Vector3(TICK * (xpos - starting_press_x), TICK * (ypos - starting_press_y), 0);
                break;
            case GeoRotation:
                models[cur_idx].rotation -=
                    Vector3(TICK * (ypos - starting_press_y), TICK * (xpos - starting_press_x), 0);
                break;
            case None:
                break;
            }
        }
        else if (cur_mode_type == Lighting) {
            if (cur_light_mode == LightEdit) {
                switch (cur_light_type) {
                case Directional:
                    Light.directional.position +=
                        Vector3(TICK * (xpos - starting_press_x), -TICK * (ypos - starting_press_y), 0);
                    break;
                case Point:
                    Light.point.position +=
                        Vector3(TICK * (xpos - starting_press_x), -TICK * (ypos - starting_press_y), 0);
                    break;
                case Spot:
                    Light.spot.position +=
                        Vector3(TICK * (xpos - starting_press_x), -TICK * (ypos - starting_press_y), 0);
                    break;
                }
            }
        }
    }
    starting_press_x = int(xpos);
    starting_press_y = int(ypos);
}
```

shader.fs

- 定義常數

```
# define PI 3.14159265358979323846
# define Directional 0
# define Point 1
# define Spot 2
```

- 新增傳進的參數

```
uniform vec3 ka;
uniform vec3 kd;
uniform vec3 ks;
uniform vec3 light_dir;
uniform vec3 light_pos;
uniform vec3 diffuse;
uniform vec3 specular;
uniform vec3 ambient;
uniform int cur_light_type;
uniform int shininess;
uniform int angle;
uniform int shade_mode;
```

- Ambient Light

```
//---Ambient Light---
vec3 Ia = ambient;
```

- Diffuse Reflection

```
//---Diffuse Reflection---
vec3 Id = diffuse;
vec3 N = normalize(vertex_normal); //normalize normal vector
vec3 V = -normalize(vertex_pos); //normalize viewpoint vector
vec3 L; //normalize light vector

if (cur_light_type == Directional)
    L = normalize(light_pos);
else if (cur_light_type == Point || cur_light_type == Spot)
    L = normalize(light_pos - vertex_pos);

float diffuse_cos = max(dot(N, L), 0.0); // ppt p.20
```

- Specular Highlight(包含halfway vector)

```
//---Specular Highlight---
vec3 Is = specular;

//half way vector
vec3 H = normalize( L + V);
float spec = pow(max(dot(H, N), 0.0), shininess); //specular reflection ppt p.27
```

- Attenuation

```
//---Attenuation---
float attenuation_p;
float attenuation_s;
if (cur_light_type == Point){
    float distance = length(light_pos - vertex_pos);
    attenuation_p = 1.0 / (0.01 + 0.8 * distance + 0.1 * distance * distance);
}
else if (cur_light_type == Spot){
    float distance = length(light_pos - vertex_pos);
    attenuation_s = 1.0 / (0.05 + 0.3 * distance + 0.6 * distance * distance);
}
```

- 最後再總和起來印出模型

```
//---result---
if (cur_light_type == Directional){
    result = Ia * ka + Id * kd * diffuse_cos  + Is * ks * spec;
}
else if (cur_light_type == Point){
    result = attenuation_p * (Ia * ka + Id * kd * diffuse_cos +  Is * ks * spec);
}
else if (cur_light_type == Spot){
    float theta = dot(normalize(vertex_pos - light_pos), normalize(light_dir));
    if (theta <= cos(angle * PI / 180)){
        result = Ia * ka;
    }
    else if (theta > cos(angle * PI / 180)){
        float spot_effect = pow(max(theta, 0), 50); //ppt p.49
        result = Ia * ka + attenuation_s * spot_effect * (Id * kd * diffuse_cos + Is * ks * spec);
    }
}

if (shade_mode == 0) FragColor = vec4(vertex_color, 1.0f);
if (shade_mode == 1) FragColor = vec4(result, 1.0f);
```

shader.vs

與shader.fs幾乎相同，僅差在先經過normalize

```
//---Normalize---
vertex_pos = (mv * vec4(aPos, 1.0f)).xyz;
vertex_normal = (transpose(inverse(mv)) * vec4(aNormal, 0.0f)).xyz;
```