深圳大学考试答题纸

(以论文、报告等形式考核专用)

二〇 一九 ～二〇 二〇 学年度第 一 学期

课程编号 150061003 课序号 01 课程名称 计算机图形学 主讲教师 胡瑞珍 评分

学 号 2017153017 姓名 钟辉 专业年级 2017 级网络工程 01 班

教师评语:

题目: 虚拟场景建模

一、 场景设计和显示

1. 构建层次结构：

   i. 如图 1 所示为使用层次结构进行生成的机器人，该机器人使用的是如图 2 所示的层次结构进行生成机器人的；其中，一共分为 3 层的结构进行生成；第一层是机器人的躯体，因为其他的结构都是会与它有所关联，让需要整个身体进行前行时就需要躯干带动着手臂、腿以及头部进行运动；第二层包含了头部、上臂以及上腿，第二层的结构都是在躯干的基础上进行运动的；因此，在使用运动的矩阵是都是需要变换到躯干的适当位置；

图 1：利用层次结构生成的机器人

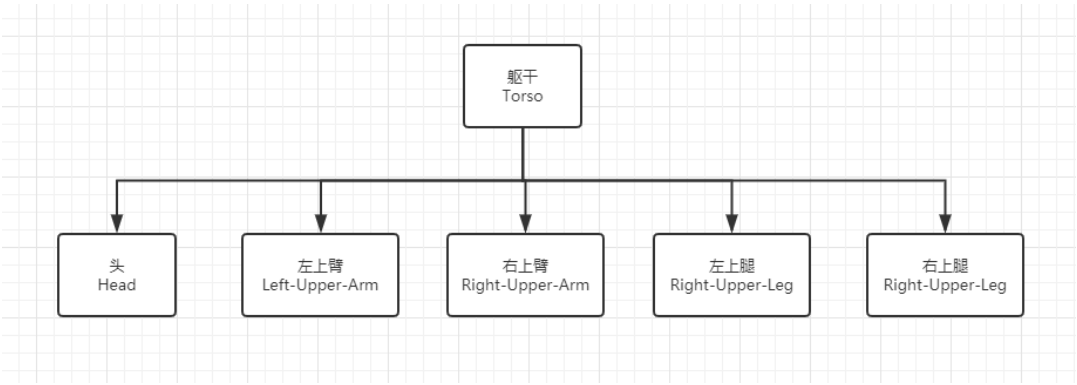

图 2：生成机器人的层次结构

   ii. 如图 3 所示为，生成层级结构的机器人所需要的参数；其中，enum 包含的是对于机器人在部件上的名称，例如躯干、头部、左上臂等等；也进行宏定义了所有对于机器人的尺寸大小的参数，其中对于 degree 是检测机器人在进行运动时手臂或者腿的最大摆幅，以防止出现在手臂或者腿脱离身体的情况（即避免机器人在行走过程中被肢解）；

```
//机器人参数
enum robot
{
    TORSO,
    HEAD,
    LEFT_UPPER_ARM,
    RIGHT_UPPER_ARM,
    LEFT_UPPER_LEG,
    RIGHT_UPPER_LEG,
    NUM_COMPONENTS
};
//机器人的身体部件参数
#define HEAD_HEIGHT 0.20
#define HEAD_WIDTH 0.20
#define HEAD_DEPTH 0.14

#define TORSO_HEIGHT 0.30
#define TORSO_WIDTH 0.20
#define TORSO_DEPTH 0.10

#define UPPER_ARM_HEIGHT 0.30
#define UPPER_ARM_WIDTH 0.10
#define UPPER_ARM_DEPTH 0.10

#define UPPER_LEG_HEIGHT 0.35
#define UPPER_LEG_WIDTH 0.10
#define UPPER_LEG_DEPTH 0.10

float degree = 0; //运动时的手臂或者脚的摆幅
```

图 3：生成层级结构的机器人的参数配置

iii. 如图 4 所示为生成机器人躯干、手臂的思维导图，首先将正方体或者长方体，即图符的 8 个顶点先产生好、写好 6 个面的法线信息、写好纹理坐标的所有阵列，之后便将顶点索引以及纹理索引、法线索引对应好即可生成一个正方体或者长方体，与读取的 obj 文件时的顺序一样，之后由自己撰写出一个生成正方体图符的 obj 文件；

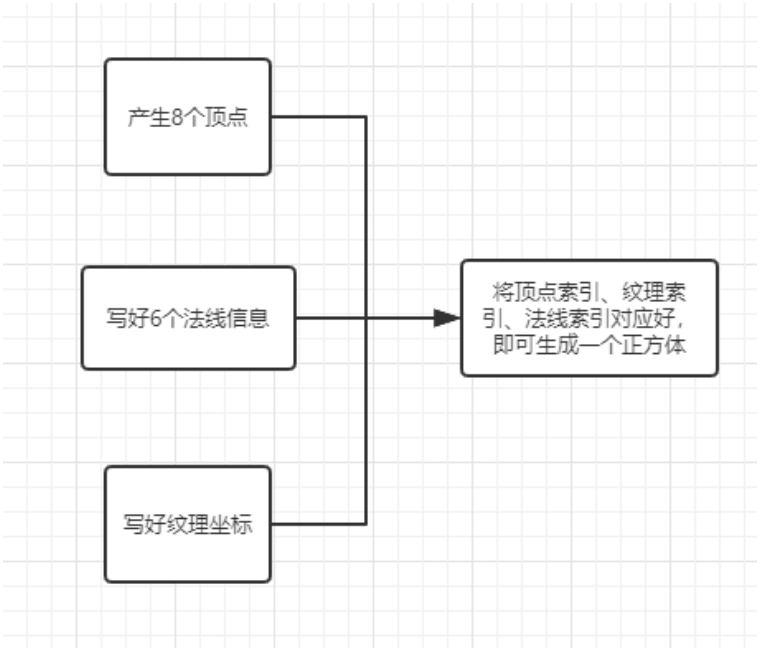

图 4：产生机器人躯体、手臂以及腿的思维导图

iv. 如图 5 所示为使用函数产生的头部方块；

图 5：产生的头部方块

v. 如图 6 所示为对于机器人的每一个部件都一一进行生成，并按照给定的机器人参数进行生成机器人的各个部件大小；

```
void createRobot()
{
    for (int i = TORSO; i < NUM_COMPONENTS; i++)
    {
        robot[i] = new My_Mesh;

        robot[i]->load_obj("texture/robot/box.obj", mat4(1.0));
        robot[i]->setShadow(true);
        robot[i]->setfox(false);
        my_meshs.push_back(robot[i]);
        mp_->add_mesh(robot[i]);
    }
    robot[TORSO]->set_texture_file("texture/steve/torso.png");
    robot[HEAD]->set_texture_file("texture/steve/head.png");
    robot[LEFT_UPPER_ARM]->set_texture_file("texture/steve/arm.png");
    robot[RIGHT_UPPER_ARM]->set_texture_file("texture/steve/arm.png");
    robot[LEFT_UPPER_LEG]->set_texture_file("texture/steve/leg.png");
    robot[RIGHT_UPPER_LEG]->set_texture_file("texture/steve/leg.png");
    robot[TORSO]->set_scale(TORSO_WIDTH, TORSO_HEIGHT, TORSO_DEPTH);
    robot[HEAD]->set_scale(HEAD_WIDTH, HEAD_HEIGHT, HEAD_DEPTH);
    robot[LEFT_UPPER_ARM]->set_scale(UPPER_ARM_WIDTH, UPPER_ARM_HEIGHT, UPPER_ARM_DEPTH);
    robot[RIGHT_UPPER_ARM]->set_scale(UPPER_ARM_WIDTH, UPPER_ARM_HEIGHT, UPPER_ARM_DEPTH);
    robot[LEFT_UPPER_LEG]->set_scale(UPPER_LEG_WIDTH, UPPER_LEG_HEIGHT, UPPER_LEG_DEPTH);
    robot[RIGHT_UPPER_LEG]->set_scale(UPPER_LEG_WIDTH, UPPER_LEG_HEIGHT, UPPER_LEG_DEPTH);
}
```

根据设定的参数进行设置机器人的大小

图 6：根据部位的尺寸进行规划部位的大小

2. 地板的构建与显示：

i. 如图 7 所示为使用与产生机器人部件相同的方法进行产生地面，只不过产生的平面而非立方体；因为，地面都是重复的情况，则产生多块相同的面块一起组成地板；

```
void My_Mesh::generate_floor(float height, float x, float y, float dx, float dy, int num)
{ //地面
    this->clear_data();
    this->m_center_ = point3f(0, 0, 0);
    unsigned int k = 0;
    int n = 0;
    float xl = dx - x;
    float yl = dy - y;
    for (float xx = x; xx <= dx; xx += 1)
    {  //构建多个矩形来构成地面
        for (float yy = y; yy <= dy; yy += 1)
        {
            // 矩形的顶点
            vec3 points[6] = {
                vec3(xx - 1, height, yy + 1),
                vec3(xx + 1, height, yy + 1),
                vec3(xx + 1, height, yy - 1),
                vec3(xx - 1, height, yy - 1),
                vec3(xx - 1, height, yy + 1),
                vec3(xx + 1, height, yy + 1),
            };
```

通过循环进行生成多个举行来构成地面

图 7：产生地面的函数

ii. 如图 8 所示，我调用写好的产生地面你的函数进行生成多块面片组成地板；（其中，将地板向下平移 0.01 的大小的目的是为了后面产生的阴影更加的美观一些）；

```
My_Mesh *floor = new My_Mesh;
floor->generate_floor(-0.0001, -50, -50, 50, 50, 5);
floor->set_texture_file("texture/wood.jpg");
floor->set_translate(0, -0.01, 0);
floor->set_theta(0, 0, 0);
floor->setShadow(false);
floor->setfox(false);
my_meshs.push_back(floor);
mp_->add_mesh(floor);
```

图 8：产生地面的构建

iii. 如图 9 所示为使用上述方法进行产生的地面情况；

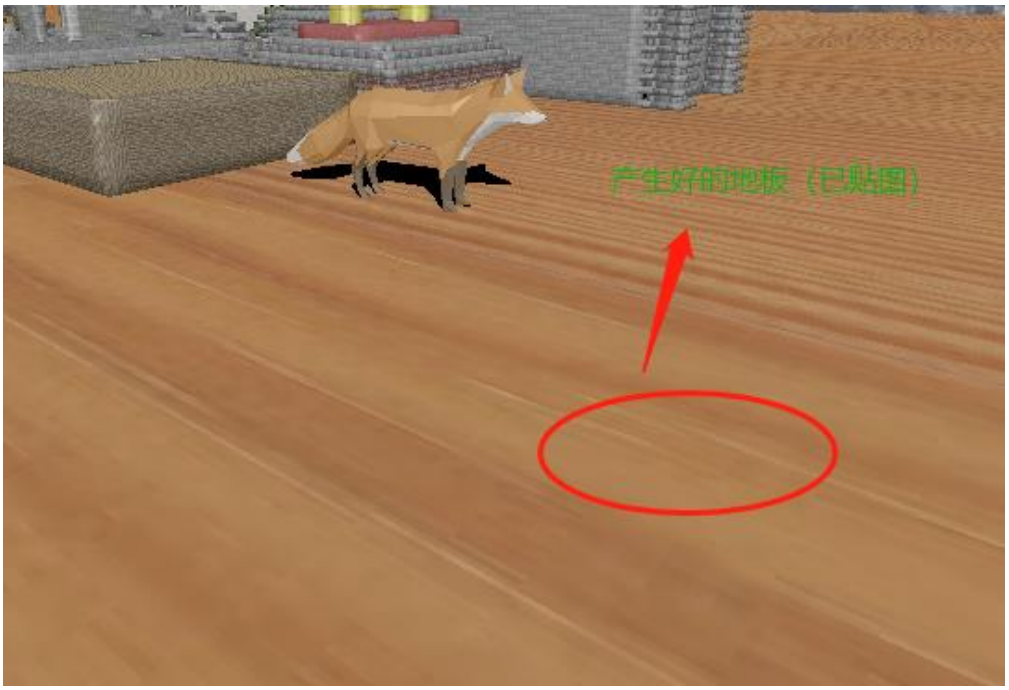

图 9：产生好的地面

3. 其他场景的构建：

i. Obj 的读取：如图 10 所示为对于 obj 文件的读取，方法同实验四的类似将文件打包成流对象，之后读取我们需要的顶点、法线、纹理以及它们对应的索引信息；在此次实验中，我还在读取文件时添加了一个 tranform 的四维变换矩阵，方便我设置读取 obj 文件得到的对象的大小；

```
while (getline(ifs, s))
{
    if (s.length() < 2)
        continue;
    if (s[0] == 'v' && s[1] == 't')
    { //纹理坐标
        istringstream is(s);
        std::string st;
        is >> st >> x >> y >> z;
        m_vt_list_.push_back(x);
        m_vt_list_.push_back(y);
        m_vt_list_.push_back(z);
    }
    else if (s[0] == 'v' && s[1] == 'n')
    { //法线坐标 以及颜色
        istringstream is(s);
        std::string st;
        is >> st >> x >> y >> z;
        m_normals_.push_back(x);
        m_normals_.push_back(y);
        m_normals_.push_back(z);
        normal_to_color(x, y, z, r, g, b);
        m_color_list_.push_back(r);
        m_color_list_.push_back(g);
        m_color_list_.push_back(b);
    }
```

进行读取
文件中的信息

图 10：obj 文件的读取

ii. 如图 11 所示，使用读取 obj 文件的方法进行产生了房子等建筑物，通过读取对应的 obj 文件后产生；

```
My_Mesh *building1 = new My_Mesh;
building1->load_obj("texture/building.obj", Scale(0.2, 0.2, 0.2) * mat4(1));
building1->set_texture_file("texture/scene.png");
building1->set_translate(0, 0, 0);
building1->set_theta(0, 0, 0);
building1->setShadow(true);
building1->setfox(false);
my_meshs.push_back(building1);
mp_->add_mesh(building1);
```

绘制场景中的建筑物

图 11：产生房子等建筑物

iii. 如图 12 所示为读取 obj 文件产生的建筑物场景，包括房子等；

图 12：读取 obj 产生的建筑物场景

iv. 如图 13 所示为通过读取 obj 文件产生的狐狸对象；

图 13：读取 obj 文件产生的狐狸

v. 对于天空盒子的产生同产生地板的方法类似，只是我采用写好 obj 文件产生正方形的面片，之后进行平移、旋转以及缩放到合适的位置进行调整出天空盒子的效果；如图 14 展示位产生盒子的前平面的方法进行产生；

```
//生成天空盒子
void createSkybox()
{
    My_Mesh *front = new My_Mesh;
    front->load_obj("texture/skybox/wall.obj", Scale(R, R, R) * mat4(1.0)); //生成盒子的前平面
    front->set_texture_file("texture/skybox/frozen_ft.tga");                //指定纹理图像文件
    front->set_translate(0, 100, R);                                        //平移
    front->set_theta(0, 90, 0);                                             //旋转轴
    front->set_theta_step(0.0, 0.0, 0.0);                                   //旋转速度
    front->setShadow(false);                                                //阴影
    front->setfox(false);                                                   //狐狸与否
    my_meshs.push_back(front);
    mp_->add_mesh(front);
```

图 14：产生天空盒子的方法

vi. 如图 15 为产生的天空盒子的效果图，可见产生的天空距离远的话，就会给人一种真实感；

图 15：产生的天空盒子效果图

二、 添加纹理

1. 如图 16 所示为加载纹理的函数实现，其中都是讲纹理文件转换成像素级别进行生成纹理对象，之后再对于图片的一些处理，例如插值；

```
void Mesh_Painter::load_texture_STBImage(std::string file_name, GLuint &m_texName)
{
    int width, height, channels = 0;
    unsigned char *pixels = NULL;
    stbi_set_flip_vertically_on_load(true);
    pixels = stbi_load(file_name.c_str(), &width, &height, &channels, 0);

    // 调整行对齐格式
    if (width * channels % 4 != 0)
        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    GLenum format = GL_RGB;
    // 设置通道格式
    switch (channels)
    {
    case 1:format = GL_RED;break;
    case 3:format = GL_RGB;break;
    case 4:format = GL_RGBA;break;
    default:format = GL_RGB;break;
    }

    // 绑定纹理对象
    glBindTexture(GL_TEXTURE_2D, m_texName);

    // 指定纹理的放大，缩小滤波，使用线性方式，即当图片放大的时候插值方式
    // 将图片的rgb数据上传给opengl
    glTexImage2D(GL_TEXTURE_2D, 0,format,
        width,height,0,format,
        GL_UNSIGNED_BYTE,pixels
    );

    // 指定插值方法
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

    // 恢复初始对齐格式
    glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
    // 释放图形内存
    stbi_image_free(pixels);
};
```

设置像素读取，后绑定纹理对象

图 16：加载纹理的函数实现

2. 如图 17 所示，在对于生成好面片信息之后，还需要讲顶点坐标、法线、颜色以及纹理坐标信息根据索引映射到 shader 上；

```
void Mesh_Painter::update_vertex_buffer()
{
    this->vao_all.clear();
    this->buffer_all.clear();
    this->vPosition_all.clear();
    this->vColor_all.clear();
    this->vTexCoord_all.clear();
    this->vNormal_all.clear();

    for (unsigned int m_i = 0; m_i < this->m_my_meshes_.size(); m_i++)
    {
        //在这里添加代码实现顶点坐标，法线，颜色，纹理坐标到shader的映射
        int num_face = this->m_my_meshes_[m_i]->num_faces();
        int num_vertex = this->m_my_meshes_[m_i]->num_vertices();
        const VertexList &vertex_list = this->m_my_meshes_[m_i]->get_vertices();
        const NormalList &normal_list = this->m_my_meshes_[m_i]->get_normals();
        const FaceList &face_list = this->m_my_meshes_[m_i]->get_faces();
        const STLVectorf &color_list = this->m_my_meshes_[m_i]->get_colors();
        const VtList &vt_list = this->m_my_meshes_[m_i]->get_vts();
```

图 17：对于纹理在 shader 上的映射实现

如图 18 所示，根据顶点索引将顶点映射到 shader 中，根据需要的顶点索引进行将顶点映射到 shader 中，之后在进行绘制；类似的还需要将法线、颜色以及纹理根据对应的索引进行映射到 shader 中，以方便 GPU 根据存储的数据已经映射关系进行绘制图片信息；

```
vec3 *points = new vec3[num_face * 3];

for (int i = 0; i < num_face; i++)
{
    int index = face_list[9 * i];
    points[3 * i] = vec3(
        (vertex_list[index * 3 + 0]),
        (vertex_list[index * 3 + 1]),
        (vertex_list[index * 3 + 2]));

    index = face_list[9 * i + 3];
    points[3 * i + 1] = vec3(
        (vertex_list[index * 3 + 0]),
        (vertex_list[index * 3 + 1]),
        (vertex_list[index * 3 + 2]));

    index = face_list[9 * i + 6];
    points[3 * i + 2] = vec3(
        (vertex_list[index * 3 + 0]),
        (vertex_list[index * 3 + 1]),
        (vertex_list[index * 3 + 2]));
}
GLintptr offset = 0;
glBufferSubData(GL_ARRAY_BUFFER, offset, sizeof(vec3) * num_face * 3, points);
//顶点坐标传给shader
offset += sizeof(vec3) * num_face * 3;
delete[] points;
```

图 18：将顶点传给 shader

3. 如图 19 所示将调整好映射关系的顶点阵列、缓冲信息、位置信息、颜色信息、纹理信息以及法线信息给保存起来，保存的方式是存储在特定的存储容器中（需要注意的是容易需要能够容纳较多的信息，因为顶点以及缓冲等信息具有非常多的数据）；

```
this->vao_all.push_back(vao);
this->buffer_all.push_back(buffer);
this->vPosition_all.push_back(vPosition);
this->vColor_all.push_back(vColor);
this->vTexCoord_all.push_back(vTexCoord);
this->vNormal_all.push_back(vNormal);
```

图 19：将调整好映射关系的顶点阵列、缓冲等信息给保存起来；

4. 如图 20 所示为在设置纹理坐标时需要注意的重要的一步，就是需要将生成的顶点阵列一一对应的绑定起来，以及需要使用 buffer 进行缓冲；不然，就会出现后面绘制的物体会将之前未绑定 VAO 信息的顶点阵列信息给覆盖掉，并且还有很大可能会出现错误；

```
// Create a vertex array object
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```


```
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER,
             sizeof(vec3) * num_face * 3 + sizeof(vec3) * num_face * 3
             + sizeof(vec3) * num_face * 3 + sizeof(vec2) * num_face * 3,
             NULL, GL_STATIC_DRAW);
```

图 20：在设定纹理坐标时重要的一步

5. 如图 21 所示为添加纹理前后的狐狸观察结果，可见添加纹理之后狐狸始会变好看很多的；

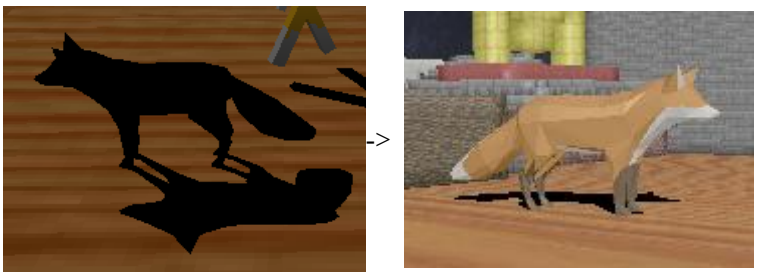

图 21：添加纹理前后的狐狸

三、 添加光照、材质、阴影效果

1. 添加光照：如图 22 所示，需要有光照的前提是拥有一个光源；因此，设置光源的位置是必不可少的信息；我设置的默认光源的位置是在坐标在（400，400，400）的位置处，其中默认的窗口大小是 800 乘于 800 的；

```
//光照的参数
vec3 lightPos = vec3(400, 400, 400);
```

图 22：设定光源的位置

2. 添加材质：如图 23 所示，对于添加材质需要的材质反射率进行固定化设置、之后计算 phong 光照模型中设计的四个向量并进行归一化；之后，计算漫反射分量以及高光系数；结合最终，将漫反射分量、镜面反射分量以及环境光综合起来即得到了 phong 光照模型；即对于光照物体添加了材质了的；

```
if (is_shadow!=0){
    fColor = vec4(0, 0, 0, 1);
    //fNormal = vec4(0, 1, 0, 1);
}
else{
//  给定材质反射率
vec3 ambiColor = vec3(0.2, 0.2, 0.2);
vec3 diffColor = vec3(0.5, 0.5, 0.5);
vec3 specColor = vec3(0.3, 0.3, 0.3);
// 计算四个归一化的向量 N,V,L,R
vec3 N_norm = normalize(N);
vec3 L_norm = normalize(lightPos-V);
vec3 V_norm = normalize(-V);
vec3 R_norm = reflect(-L_norm,N_norm);
//  高光系数
float lambertian = clamp(dot(L_norm, N_norm), 0.0, 1.0);
float specular = clamp(dot(R_norm, V_norm), 0.0, 1.0);
float shininess = 10.0;
//  计算最终每个面片的颜色
fColor = vec4(ambiColor + diffColor * lambertian + specColor * pow(specular, shininess), 1.0)
fColor = fColor * texture2D( texture, texCoord );
fNormal = vec4(N,0);
}
```


图 23：对于反射率、高光系数等参数进行添加材质

3. 如图 24 所示为添加材质之后的结构，可见添加材质之后被光照的物体更加接近于现实的物体；

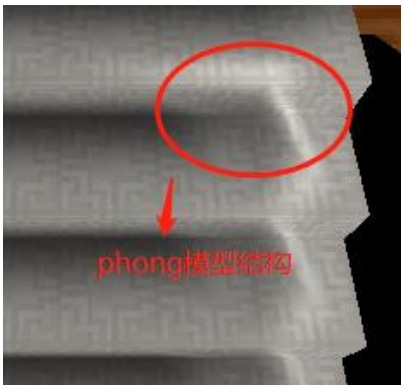

图 24：添加了材质之后的结构

4. 添加阴影：

i. 如图 25 所示，首先添加透视投影的矩阵方便进行相机进行观察，即实现近大远小的效果；之后，计算投影矩阵，根据需要投影到那个平面进行计算；

```
mat4 projMatrix = Perspective(90, 1, 0.00001, 100);
float lx = lightPos.x;
float ly = lightPos.y;
float lz = lightPos.z;
//计算投影矩阵
mat4 shadowProjMatrix(
    -ly, 0.0, 0.0, 0.0,
    lx, 0.0, lz, 1.0,
    0.0, 0.0, -ly, 0.0,
    0.0, 0.0, 0.0, -ly);
```

透视投影矩阵

阴影矩阵

图 25：对于透视投影矩阵以及投影的矩阵

ii. 如图 26 所示，对于需要进行变换的矩阵都传递给着色器进行变换；对于物体旋转、缩放、平移的变换对应的时 modelMatrix 矩阵，即物体含有旋转、缩放或者平移的几何变换时修改该矩阵之后再传递给着色器，后由着色器进行绘制出变化即可；对于相机矩阵以及必须要传递的透视投影的矩阵一定需要传递给着色器进行绘制，之后对于需要绘制阴影的物体则是将阴影矩阵加入到模式变换矩阵 modelMatrix 中，再传递给着色器进行绘制；其中，最后将需要进行绘制阴影的物体进行绘制出对应的阴影；

```
this->m_my_meshes_[i]->get_scale(x, y, z);
vec3 scale(x, y, z);
mat4 modelMatrix =
    this->m_my_meshes_[i]->getTransform() * Translate(translate) * RotateZ(Theta[2]) * RotateY(Theta[1]) * RotateX(Theta[0]) * Scale(scale);

glUniformMatrix4fv(viewMatrix_all[i], 1, GL_TRUE, viewMatrix);
glUniformMatrix4fv(projMatrix_all[i], 1, GL_TRUE, projMatrix);
glUniform3fv(lightPos_all[i], 1, lightPos);
if (this->m_my_meshes_[i]->isShadow())
{ // 将投影矩阵以及模式矩阵传给着色器
    glUniformMatrix4fv(modelMatrix_all[i], 1, GL_TRUE, shadowProjMatrix * modelMatrix);
    glUniform1i(is_shadow_all[i], 1);
    glDrawArrays(GL_TRIANGLES, 0, this->m_my_meshes_[i]->num_faces() * 3);
}
glUniformMatrix4fv(modelMatrix_all[i], 1, GL_TRUE, modelMatrix);
glUniform1i(is_shadow_all[i], 0);
glDrawArrays(GL_TRIANGLES, 0, this->m_my_meshes_[i]->num_faces() * 3);
```

物体需要进行的变换矩阵

将相机矩阵以及透视投影矩阵传给着色器，如需要绘制阴影还将阴影矩阵变换加入

将模式矩阵传给着色器，并将需要绘制阴影的物体进行绘制

图 26：将需要进行的变换都传给着色器进行变换

iii. 如图 27 跟 28 所示为对于建筑物中的树进行绘制阴影，可见阴影绘制成功；

```
My_Mesh *building1 = new My_Mesh;
building1->load_obj("texture/building.obj", Scale(0.2, 0.2, 0.2) * mat4(1));
building1->set_texture_file("texture/scene.png");
building1->set_translate(0, 0, 0);
building1->set_theta(0, 0, 0);
building1->setShadow(true);
building1->setfox(false);
my_meshs.push_back(building1);
mp_->add_mesh(building1);
```

对于建筑物选择了绘制阴影

图 27：对于建筑物选择绘制阴影

图 28：对于建筑物中树进行绘制出阴影

四、 用户交互实现视角切换完成对场景的任意角度浏览

1. 相机的设置：

   i. 如图 29 所示，在 mat.h 中设置全景观察相机的矩阵设置，设置的相机观察矩阵根据在实验 3.1 中推到的矩阵进行设置，具体的矩阵如图 30 所示；

```
inline
mat4 LookAt( const vec4& eye, const vec4& at, const vec4& up )
{
    vec4 n = normalize(eye - at);
    vec3 uu = normalize(cross(up, n));
    vec4 u = vec4(uu.x, uu.y, uu.z, 0.0);
    vec3 vv = normalize(cross(n,u));
    vec4 v = vec4(vv.x, vv.y, vv.z, 0.0);
    vec4 t = vec4(0.0, 0.0, 0.0, 1.0);
    mat4 c = mat4(u, v, n, t);
    return c * Translate( -eye );
}
```

设定全景观察相机矩阵

图 29：设定全景观察相机的矩阵

$$\begin{bmatrix} u_x & u_y & u_z & -xu_x - yu_y - zu_z \\ v_x & v_y & v_z & -xv_x - yv_y - zv_z \\ n_x & n_y & n_z & -xn_x - yn_y - zn_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

图 30：相机观察矩阵

   ii. 如图 31 所示为进行设置相机位置信息的情况；其中，首先利用 get_translate 函数获取相机与运动物体的相对位置信息；之后，只用球坐标信息进行设置相机的位置；即需要使用相机位置与仰角、水平角之间的关系进行设置相机；其中，当观察的运动物体是机器人是狐狸时，相机时通过运动的物体的相对位置进行设置，机器人的第一人称则不是；

```
float x2, y2, z2, xl, yl, zl;
float tx, ty, tz; //当前夹角
//att = 0时,运动物体为机器人, att = 1时,运动物体为狐狸
//att = 2时机器人的第一视角
if (att == 0)
{   //获取机器人躯干的位置信息
    robot[TORSO]->get_translate(x2, y2, z2);
    //cout<<x2<<y2<<z2;
}
else if (att == 1)
{   //获取狐狸的位置信息
    fox->get_translate(x2, y2, z2);
}
else if (att == 2)
{   //同样获取机器人躯干的信息
    robot[TORSO]->get_translate(x2, y2, z2);
}
//设置相机的位置
vec4 up = vec4(0, 1, 0, 0);
if (att == 1 || att == 0)
{   //全景相机的x、y、z，看的位置就是运动的物体的位置
    float x = radius * cos(upAngle * M_PI / 180.0) * sin(rotateAngle * M_PI / 180.0) + x2;
    float y = radius * sin(upAngle * M_PI / 180.0) + y2;
    float z = radius * cos(upAngle * M_PI / 180.0) * cos(rotateAngle * M_PI / 180.0) + z2;
    eye = vec4(x, y, z, 1);
    at = vec4(x2, y2, z2, 0);
}
else
{ //第一视角时的相机参数，将相机置于机器人的适当位置进行观察
    eye = vec4(x2, y2 + 0.5, z2, 1);
    at = vec4(x2 + fr * sin(rotateAngle * M_PI / 180.0), y2 + 0.4,
              z2 + fr * cos(rotateAngle * M_PI / 180.0), 0);
}
```

根据与运动物体的相对位置进行设置相机位置

设置第一人称的相机位置

图 31：设置相机的位置

iii. 对于机器人第一视角坐标的求法；因为在第一人称观察时，实现方向与相机的位置的 Y 值是相同的；因此，只需要考虑俯视图下 Z-X 平面的坐标关系；其中，转换为第一人称时，原机器人的坐标就成为了现在相机的观察位置；但是，对于视点方向则不同；视点方向则还需要根据如图 32 所示的方法进行求解视点的 X 值跟 Z 值，Y 值同相机的相同；

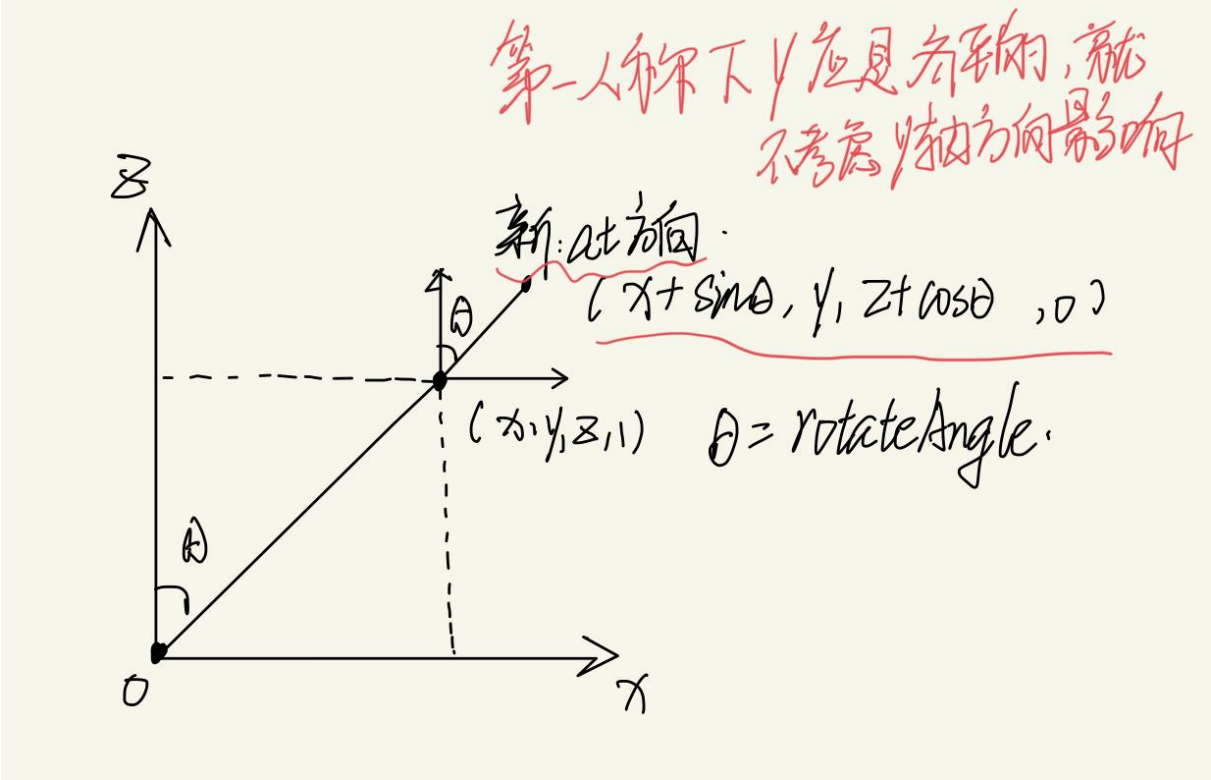

图 32：第一人称相机的设置

iv. 如图 33 所示为将相机变换矩阵传递给绘制物体的函数中，以便在绘制物体时与后面需要绘制的阴影矩阵一同传递给着色器进行处理；也就时达到了对于传递给 GPU 的数据只有一组点数据以及变换的矩阵，能够在选择变换矩阵与一大堆顶点数据中节省了 GPU 的损耗，进一步加大了效率；

```
mp_->draw_meshes(viewMatrix);
```

图 33：之后将相机变换矩阵传递给绘制物体的函数中，进一步传递给着色器

2. 相机交互：

i. 如图 34 所示为在对于相机位置的调整的参数改变的键盘相应事件，其中例如按下键盘的“W/w”键时，相机与 Y 轴与 X-Z 平面的夹角 upAngle 就会增加，即实现了相机相对于物体向上移动了的感觉；

```
case 'W' :
case 'w' :
    if (upAngle + 1 < 90) upAngle += 1; break;
case 'S':
case 's':
    if (upAngle - 1 > 0)upAngle -= 1;break;
case 'A':
case 'a':rotateAngle -= 1;break;
case 'D':
case 'd':rotateAngle += 1;break;
case 'Q':
case 'q':
if (radius > 1.5) radius -= 0.2;break;
case 'E':
case 'e':radius += 0.2; break;
```

图 34：在键盘相应事件中的对于相机的改变

ii. 如图 35 所示，使用数字键盘的 0、1、2 键进行切换视角位置，可选择全景视角下观察机器人或者狐狸，也可选择机器人的第一视角下观察世界；

```
case '1': //切换视角
    att = 1;    //全景视角，观察机器人
    break;
case '0':   //全景视角，观察狐狸
    att = 0;
    break;
case '2':   //机器人的第一视角
    att = 2;
    break;
```

图 35：切换相机的视角位置

iii. 如图 36 所示为按下键盘的空格键时，会将相机的参数设置为最初设置的所有相机参数（视角除外，视角确定为机器人的第一视角）；

```
case ' ':
    // 初始化全景相机参数
    radius = 6.80;
    rotateAngle = -92;
    upAngle = 43.0;
    at = vec4(0, 0, 0, 1);
    eye = vec4(0, 0, 0, 1);
    att = 2;
```

图 36：按下空格键时便是初始化相=相机参数为最初设置的

3. 对于相机变换的效果：

对于相机的运动：

i. 如图 37 所示为按下键盘的“A/a”键时，相机会沿着 X 轴负方向进行水平平移，即会减小 X 轴与 Z 轴的夹角 rotateAngle；

图 37：按下键盘的“A/a”键时的情况

ii. 如图 38 所示为按下键盘的“D/d”键时，相机会朝着 X 轴的正方向进行水平平移，即会增加 X 轴与 Z 轴的夹角 rotateAngle；

图 38：按下键盘的“D/d”键时的情况

iii. 如图 39 所示为按下键盘的“W/w”键时，相机会朝着 Y 轴的正方向进行竖直平移，即会增加 Y 轴与 X-Z 平面的夹角 upAngle；

图 39：按下键盘的“W/w”键时的情况

iv. 如图 40 所示为按下键盘的“S/s”键时，相机会朝着 Y 轴的负方向进行竖直平移，即会减小 Y 轴与 X-Z 平面的夹角 upAngle；

图 40：按下键盘的“S/s”键时的情况

调整相机与物体的距离：

i. 如图 41 所示为按下键盘的“Q/q”键时，相机与物体的距离会变小；

图 41：按下键盘的“Q/q”键时的情况

ii. 如图 42 所示为按下键盘的“E/e”时，相机与物体的距离会增加；

图 42：按下键盘的“E/e”键时的情况

iii. 如图 43 所示为按下键盘的空格键时，相机的参数会恢复到程序起初运行时的参数(视角除外，视角转变为机器人的第一视角)；

图 43：按下键盘的空格键时的情况

按键对于的视角切换：

i. 如图 44 所示为按下键盘的数字“1”键时，视角的视点会由机器人转变到狐狸那里；

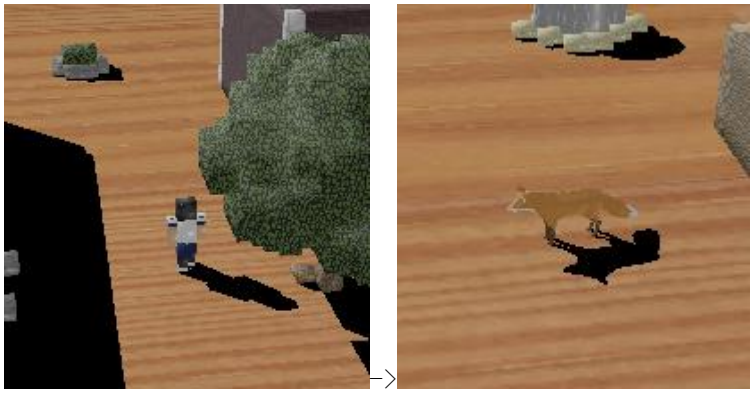

图 44：按下键盘的数字“1”的情况

ii. 如图 45 为按下键盘的数字“2”键时，机器人的视角会由第三人称变为第一人称；

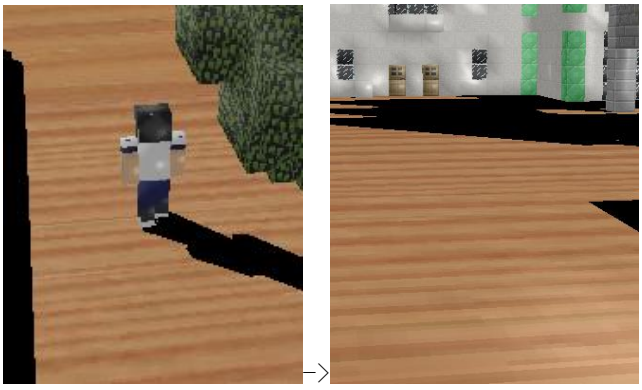

图 45：按下键盘的数字“2”时的情况

iii. 如图 46 为按下键盘的数字“0”键时，视角的视点会由其他地方转换到机器人身上；

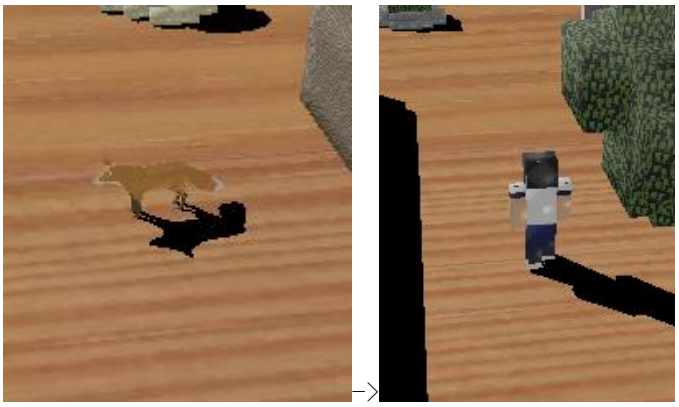

图 46：按下键盘的数字“0”的情况

五、 通过交互控制物体

1. 通过键盘控制不同的物体：

i. 如图 47 所示，对于使用键盘控制不同的物体，通过一个全局变量来检测当前控制的物体到底是哪一个，即再此次程序中通过 att 的值进行控制当前控制的物体究竟是狐狸还是机器人；att = 0 时，使用的是机器人（默认进行控制机器人），反之则是进行控制狐狸进行运动（视点都是在指定的运动物体）；

Att = 2 时，使用的是机器人的第一视角进行观察周围世界；

```
case '1': //切换视角
    att = 1;    //全景视角，观察机器人
    break;
case '0':   //全景视角，观察狐狸
    att = 0;
    break;
case '2':   //机器人的第一视角
    att = 2;
    break;
```

图 47：对于不同物体进行切换

ii. 如图 48 所示，通过 att 变量进行判断当前相机需要跟踪的是哪个物体，即哪个物体是当前运动的物体，因相机跟踪的是运动的物体；或者，转换视角进行观察周围的物体世界；之后，使用图 49 的调整相机的方式进行跟踪或者转换视角进行观察；

```
float x2, y2, z2, x1, y1, z1;
float tx, ty, tz; //当前夹角
//att = 0时,运动物体为机器人, att = 1时,运动物体为狐狸
//att = 2时机器人的第一视角
if (att == 0)
{   //获取机器人躯干的位置信息
    robot[TORSO]->get_translate(x2, y2, z2);
    //cout<<x2<<y2<<z2;
}
else if (att == 1)
{   //获取狐狸的位置信息
    fox->get_translate(x2, y2, z2);
}
else if (att == 2)
{   //同样获取机器人躯干的信息
    robot[TORSO]->get_translate(x2, y2, z2);
}
```

获取运动物体的位置信息

图 48：获取运动物体的位置信息

```
//设置相机的位置
vec4 up = vec4(0, 1, 0, 0);
if (att == 1 || att == 0)
{   //全景相机的x、y、z，看的位置就是运动的物体的位置
    float x = radius * cos(upAngle * M_PI / 180.0) * sin(rotateAngle * M_PI / 180.0) + x2;
    float y = radius * sin(upAngle * M_PI / 180.0) + y2;
    float z = radius * cos(upAngle * M_PI / 180.0) * cos(rotateAngle * M_PI / 180.0) + z2;
    eye = vec4(x, y, z, 1);
    at = vec4(x2, y2, z2, 0);
}
else
{ //第一视角时的相机参数，将相机置于机器人的适当位置进行观察
    eye = vec4(x2, y2 + 0.5, z2, 1);
    at = vec4(x2 + fr * sin(rotateAngle * M_PI / 180.0), y2 + 0.4,
              z2 + fr * cos(rotateAngle * M_PI / 180.0), 0);
}
```

全景视角下观察运动物体

机器人第一视角观察

图 49：调整相机选择指定的方式进行观察

iii. 如图 50 所示为通过加号键或者减号键进行调整物体狐狸的大小；

```
case '+':
    fox->get_scale(x, y, z);
    fox->set_scale(x + 0.01, y + 0.01, z + 0.01);
    break;
case '-':
    fox->get_scale(x, y, z);
    fox->set_scale(x - 0.01, y - 0.01, z - 0.01);
    break;
```

图 50：使用加号键或减号键进行调整狐狸的大小

iv. 如图 51 所示为当 att = 1 时，即对于狐狸进行控制的时候会，按下键盘的上将“↑”键，就对于当前物体-狐狸进行执行前进的操作；其中，对于物体的前进后退都是基于 X-Z 平面上的变化；因此，在 Y 方向上的值都是不需要进行变化的，只需要对 X 跟 Z 轴的坐标进行计算后修改即可；当按下键盘的下键“↓”键时，物体执行后退的操作、按下键盘的左键“←”键时，物体绕着自身的左侧进行转弯的动作、按下键盘的右键“→”键时，物体绕着自身的右侧进行转弯的动作；

```
case 1://控制狐狸进行运动
    switch (key)
    {
    case GLUT_KEY_UP:
        fox->get_translate(x1, y1, z1);
        fox->get_theta(x2, y2, z2);
        fox->set_translate(x1 + sin(y2 * DegreesToRadians) * 0.02, y1, z1 + cos(y2 * DegreesToRadians) * 0.02);
        break;
    case GLUT_KEY_DOWN:
        fox->get_translate(x1, y1, z1);
        fox->get_theta(x2, y2, z2);
        fox->set_translate(x1 - sin(y2 * DegreesToRadians) * 0.02, y1, z1 - cos(y2 * DegreesToRadians) * 0.02);
        break;
    case GLUT_KEY_LEFT:
        fox->get_theta(x2, y2, z2);
        fox->set_theta(x2, y2 + 10, z2);
        break;
    case GLUT_KEY_RIGHT:
        fox->get_theta(x2, y2, z2);
        fox->set_theta(x2, y2 - 10, z2);
        break;
    }
    break;
```

图 51：通过键盘的上下左右键进行控制物体的运动

v. 同样使用上下左右键进行控制物体运动，只是当 att = 2 时观察的视角是处于机器人的第一视角的；

```
case 2: //第一视角下机器人的运动
    switch (key)
    {
    case GLUT_KEY_UP:
        degree += lor ? 2 : -2;
        if (degree >= 12)
            lor = false;
        else if (degree <= -12)
            lor = true;
        robot[TORSO]->get_translate(x1, y1, z1);
        robot[TORSO]->get_theta(x2, y2, z2);
        robot[TORSO]->set_translate(x1 + sin(y2 * DegreesToRadians) * 0.02,
        y1, z1 + cos(y2 * DegreesToRadians) * 0.02);
        break;
    case GLUT_KEY_DOWN:
        degree += lor ? 2 : -2;
        if (degree >= 12)
            lor = false;
        else if (degree <= -12)
            lor = true;
        robot[TORSO]->get_translate(x1, y1, z1);
        robot[TORSO]->get_theta(x2, y2, z2);
        robot[TORSO]->set_translate(x1 - sin(y2 * DegreesToRadians) * 0.02,
        y1, z1 - cos(y2 * DegreesToRadians) * 0.02);
        break;
    case GLUT_KEY_LEFT:
        robot[TORSO]->get_theta(x2, y2, z2);
        robot[TORSO]->set_theta(x2, y2 + 10, z2);
        rotateAngle += 10;
        break;
    case GLUT_KEY_RIGHT:
        robot[TORSO]->get_theta(x2, y2, z2);
        robot[TORSO]->set_theta(x2, y2 - 10, z2);
        rotateAngle -= 10;
        break;
    }
    break;
}
```

图 52：第一视角下机器人的运动情况

2. 交互结果：

对于物体的运动：

i. 如图 53 所示为按下键盘的右键“→”时，选定的物体会向着其本身的右边转向；

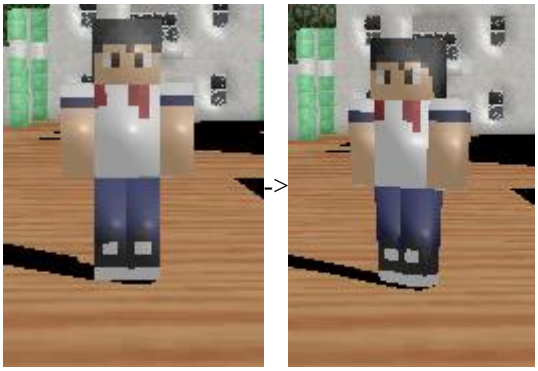

图 53：按下键盘右键的情况

ii. 如图 54 所示为按下键盘的左键“←”时，选定的物体会向着其本身的左边转向；

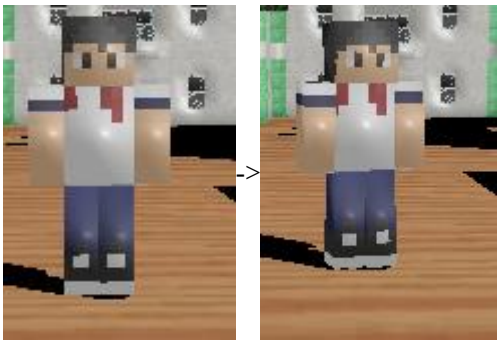

图 54：按下键盘左键的情况

iii. 如图 55 所示为按下键盘的上键“↑”时，选定的物体会向着其本身的前方步行；

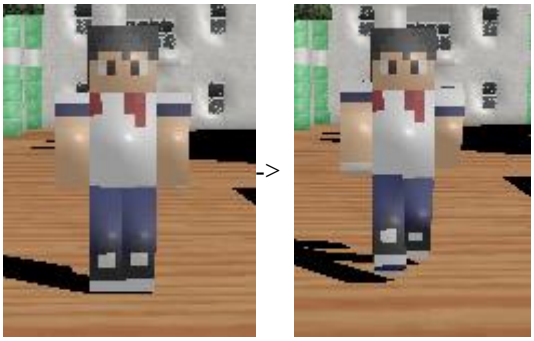

图 55：按下键盘的上键的情况

iv. 如图 56 所示按下键盘的下键“↓”时，选定的物体会向着其本身的后方步行；

图 56：按下键盘下键的情况

对于物体的变化：

i. 如图 57 所示为按下键盘的加号键“＋”时,狐狸进行尺寸进行增大的操作;

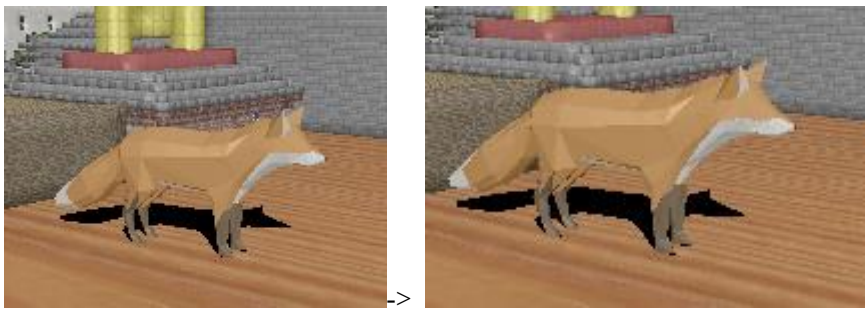

图 57：按下键盘的加号键时狐狸的变化

ii. 如图 16 所示为按下键盘的减号键“-”时,狐狸的尺寸变小的操作;

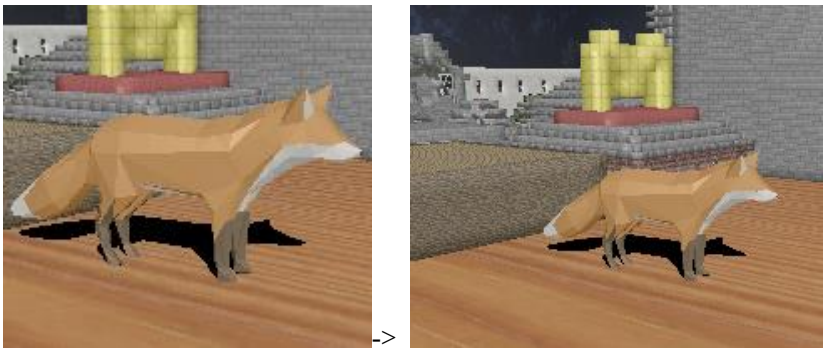

图 58：按下键盘的减号键时狐狸的变化

按键对于物体的视角切换：

iv. 如图 17 所示为按下键盘的数字“1”键时，视角的视点会由机器人转变到狐狸那里；

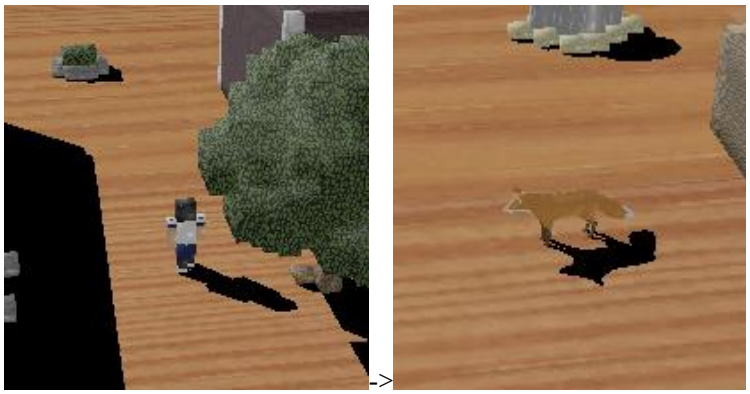

图 59：按下键盘的数字“1”的情况

v. 如图 18 为按下键盘的数字“2”键时，机器人的视角会由第三人称变为第一人称；

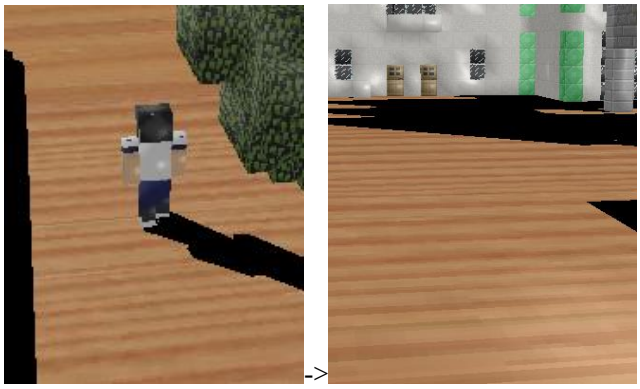

图 60：按下键盘的数字“2”时的情况

vi. 如图 19 为按下键盘的数字“0”键时，视角的视点会由其他地方转换到机器人身上；

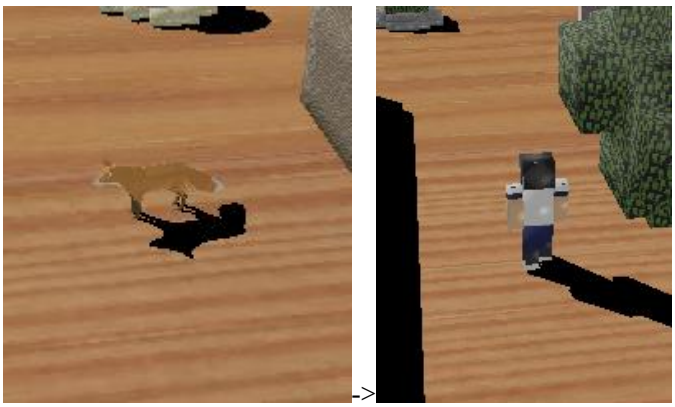

图 61：按下键盘的数字“0”的情况