```
/*            PURPOSE : Simple framework for ray-tracing

        PREREQUISITES : matrix.h
 */

#include <X11/Xlib.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"

#define INFINITE_PLANE   0
#define PLANE            1
#define SPHERE           2

#define EPSILON          0.00001
#define INFTY            999999.9
#define N_OBJECTS        1024
#define MAX_INTENSITY    255.0

#define Ex               12.0
#define Ey               12.0
#define Ez               3.0

#define Gx               0.0
#define Gy               0.0
#define Gz               0.0

#define UPx              0.0
#define UPy              0.0
#define UPz              1.0

#define Lx               0.0
#define Ly               5.0
#define Lz               25.0

#define Near             1.0
#define Far              25.0

#define THETA            45.0
#define ASPECT           1.5

#define H                640
#define M_PI             3.14159265358979323846264338327950288419716939937510582097494459230781640628

typedef struct {
    int width, height ;
} window_t ;

typedef struct {
    dmatrix_t UP ;
    dmatrix_t E ;
    dmatrix_t G ;
    dmatrix_t u, v, n ;
} camera_t ;

typedef struct {
    double r, g, b ;
} color_t ;

typedef struct {
    int type ;
    double (*intersection_function)(dmatrix_t *,dmatrix_t *) ;
    dmatrix_t M, Minv ;
    color_t specular_color, diffuse_color, ambient_color ;
    double reflectivity, specular_coeff, diffuse_coeff, ambient_coeff, f ;
} object_t ;
```

```
typedef struct {
    dmatrix_t position ;
    color_t color ;
    color_t intensity ;
} light_t ;

object_t object[N_OBJECTS] ;
int nobjects = 0 ;

Display *InitX(Display *d, Window *w, int *s, window_t *Window) {

    d = XOpenDisplay(NULL) ;
    if(d == NULL) {
        printf("Cannot open display\n") ;
        exit(1) ;
    }
    *s = DefaultScreen(d) ;
    *w = XCreateSimpleWindow(d,RootWindow(d,*s),0,0,Window->width,Window->height,1,BlackPixel
(d,*s),WhitePixel(d, *s)) ;
    Atom delWindow = XInternAtom(d,"WM_DELETE_WINDOW",0) ;
    XSetWMProtocols(d,*w,&delWindow,1) ;
    XSelectInput(d,*w,ExposureMask | KeyPressMask) ;
    XMapWindow(d,*w) ;
    return(d) ;
}

void SetCurrentColorX(Display *d, GC *gc, unsigned int r, unsigned int g, unsigned int b) {

    XSetForeground(d,*gc,r << 16 | g << 8 | b) ;
}

void SetPixelX(Display *d, Window w, int s, int i, int j) {

    XDrawPoint(d,w,DefaultGC(d,s),i,j) ;
}

void QuitX(Display *d, Window w) {

    XDestroyWindow(d,w) ;
    XCloseDisplay(d) ;
}

light_t *build_light(light_t *light, dmatrix_t *position, color_t color, color_t intensity) {

    dmat_alloc(&light->position,4,1) ;

    light->position = *position ;
    light->color.r = color.r ;
    light->color.g = color.g ;
    light->color.b = color.b ;
    light->intensity.r = intensity.r ;
    light->intensity.g = intensity.g ;
    light->intensity.b = intensity.b ;
    return light ;
}

window_t *build_window(window_t *Window, int height, float aspect) {

    Window->height = height ;
    Window->width =  aspect*height ;

    return(Window) ;
}

camera_t *build_camera(camera_t *Camera, window_t *Window) {
```

```
    dmat_alloc(&Camera->E,4,1) ;

    Camera->E.m[1][1] = Ex ;
    Camera->E.m[2][1] = Ey ;
    Camera->E.m[3][1] = Ez ;
    Camera->E.m[4][1] = 1.0 ;

    dmat_alloc(&Camera->G,4,1) ;

    Camera->G.m[1][1] = Gx ;
    Camera->G.m[2][1] = Gy ;
    Camera->G.m[3][1] = Gz ;
    Camera->G.m[4][1] = 1.0 ;

    dmat_alloc(&Camera->n,4,1) ;
    Camera->n = *dmat_normalize(dmat_sub(&Camera->E,&Camera->G)) ;
    Camera->n.l = 3 ;

    dmat_alloc(&Camera->UP,4,1) ;

    Camera->UP.l = 3 ;

    Camera->UP.m[1][1] = UPx ;
    Camera->UP.m[2][1] = UPy ;
    Camera->UP.m[3][1] = UPz ;
    Camera->UP.m[4][1] = 1.0 ;

    dmat_alloc(&Camera->u,4,1) ;

    Camera->u = *dmat_normalize(dcross_product(&Camera->UP,&Camera->n)) ;
    Camera->v = *dmat_normalize(dcross_product(&Camera->n,&Camera->u)) ;

    return(Camera) ;
}

dmatrix_t *intersection_coordinates(dmatrix_t *e, dmatrix_t *direction, double t) {

    dmatrix_t *intersection ;

    intersection = (dmatrix_t *)malloc(sizeof(dmatrix_t)) ;
    dmat_alloc(intersection,4,1) ;

    intersection->m[1][1] = e->m[1][1] + direction->m[1][1]*t ;
    intersection->m[2][1] = e->m[2][1] + direction->m[2][1]*t ;
    intersection->m[3][1] = e->m[3][1] + direction->m[3][1]*t ;
    intersection->m[4][1] = 1.0 ;

    return intersection ;
}

double infinite_plane_intersection(dmatrix_t *e, dmatrix_t *d) {

    if (d->m[3][1] >= 0.0) return -1.0 ; else return -1.0*e->m[3][1]/d->m[3][1] ;
}

double plane_intersection(dmatrix_t *e, dmatrix_t *d) {

    double t ;
    dmatrix_t *intersection ;

    if (d->m[3][1] >= 0.0) {
        return -1.0 ;

    }
    else {
        t = -1.0*e->m[3][1]/d->m[3][1] ;
        intersection = intersection_coordinates(e,d,t) ;
```

```
        if ((fabs(intersection->m[1][1]) < 1.0) && (fabs(intersection->m[2][1]) < 1.0)) {
            delete_dmatrix(intersection) ;
            return t ;
        }
        else {
            delete_dmatrix(intersection) ;
            return -1.0 ;
        }
    }
}

double sphere_intersection(dmatrix_t *e, dmatrix_t *d) {

    double a = ddot_product(from_homogeneous(d),from_homogeneous(d)) ;
    double b = ddot_product(from_homogeneous(e),from_homogeneous(d)) ;
    double c = ddot_product(from_homogeneous(e),from_homogeneous(e)) - 1.0 ;

    double discriminant = b*b - a*c ;

    if (discriminant < 0.0) {
        return -1.0 ;
    }
    else {
        if (discriminant < EPSILON) {
            return -b/a ;
        }
        else {
            double t1 = -b/a - sqrtl(discriminant)/a ;
            double t2 = -b/a + sqrtl(discriminant)/a ;
            if (t1 < t2) {
                if (t1 > EPSILON) {
                    return t1 ;
                }
                else {
                    return -1.0 ;
                }
            }
            else {
                return t2 ;
            }
        }
    }
}

dmatrix_t *normal_to_surface(object_t *object, dmatrix_t *intersection) {

    dmatrix_t *normal ;

    normal = (dmatrix_t *)malloc(sizeof(dmatrix_t)) ;
    dmat_alloc(normal,4,1) ;

    switch ((*object).type) {

        case PLANE          :   normal->m[1][1] = 0.0 ;
                                normal->m[2][1] = 0.0 ;
                                normal->m[3][1] = 1.0 ;
                                break ;

        case INFINITE_PLANE :   normal->m[1][1] = 0.0 ;
                                normal->m[2][1] = 0.0 ;
                                normal->m[3][1] = 1.0 ;
                                normal->m[4][1] = 0.0 ;
                                break ;

        case SPHERE         :   normal->m[1][1] = intersection->m[1][1] ;
                                normal->m[2][1] = intersection->m[2][1] ;
                                normal->m[3][1] = intersection->m[3][1] ;
```

```
                                normal->m[4][1] = 0.0 ;
                                break ;

        default: printf("No such object type\n") ;

    }
    return normal ;
}

int find_min_hit_time(double t0[N_OBJECTS]) {

    double min_t = INFTY ;
    int position = -1 ;

    for(int i = 0 ; i < nobjects ; i++) {
        if (t0[i] != -1.0) {
            if (t0[i] < min_t) {
                min_t = t0[i] ;
                position = i ;
            }
        }
    }
    return position ;
}

dmatrix_t *ray_direction(camera_t *Camera, window_t *Window, double height, double width, double i, double
j) {

    dmatrix_t *d ;

    d = (dmatrix_t *)malloc(sizeof(dmatrix_t)) ;
    dmat_alloc(d,4,1) ;

    d->m[1][1] = -1.0*Near*Camera->n.m[1][1] +
    width*(2.0*i/Window->width - 1.0)*Camera->u.m[1][1] +
    height*(2.0*j/Window->height - 1.0)*Camera->v.m[1][1] ;

    d->m[2][1] = -1.0*Near*Camera->n.m[2][1] +
    width*(2.0*i/Window->width - 1.0)*Camera->u.m[2][1] +
    height*(2.0*j/Window->height - 1.0)*Camera->v.m[2][1] ;

    d->m[3][1] = -1.0*Near*Camera->n.m[3][1] +
    width*(2.0*i/Window->width - 1.0)*Camera->u.m[3][1] +
    height*(2.0*j/Window->height - 1.0)*Camera->v.m[3][1] ;

    d->m[4][1] = 0.0 ;

    return(d) ;
}

dmatrix_t *vector_to_light_source(dmatrix_t *intersection, dmatrix_t *light_position) {

    dmatrix_t *s, *sn ;

    s = dmat_sub(light_position,intersection) ;
    sn = dmat_normalize(s) ;
    delete_dmatrix(s) ;

    return sn ;
}

dmatrix_t *vector_to_center_of_projection(dmatrix_t *intersection, dmatrix_t *e) {

    dmatrix_t *v, *vn ;

    v = dmat_sub(e,intersection) ;
    vn = dmat_normalize(v) ;
```

```
    delete_dmatrix(v) ;

    return vn ;
}

dmatrix_t *vector_to_specular_reflection(dmatrix_t *N, dmatrix_t *S) {

    dmatrix_t *r, *rn ;

    r = (dmatrix_t *)malloc(sizeof(dmatrix_t)) ;
    dmat_alloc(r,4,1) ;

    double sn = 2.0*ddot_product(N,S) ;

    r->m[1][1] = -1.0*S->m[1][1] + sn*N->m[1][1] ;
    r->m[2][1] = -1.0*S->m[2][1] + sn*N->m[2][1] ;
    r->m[3][1] = -1.0*S->m[3][1] + sn*N->m[3][1] ;
    r->m[4][1] = 0.0 ;

    rn = dmat_normalize(r) ;
    delete_dmatrix(r) ;

    return rn ;
}

color_t color_init(double r, double g, double b) {

    color_t s ;

    s.r = r ;
    s.g = g ;
    s.b = b ;

    return s ;
}

color_t color_mult(double a, color_t c) {

    color_t s ;

    s.r = a*c.r ;
    s.g = a*c.g ;
    s.b = a*c.b ;

    return s ;
}

color_t color_add(color_t c1, color_t c2) {

    color_t s ;

    s.r = c1.r + c2.r ;
    s.g = c1.g + c2.g ;
    s.b = c1.b + c2.b ;

    return s ;
}

color_t shade(light_t *light, object_t *object, dmatrix_t *e, dmatrix_t *d, color_t background) {
    //set the initial color to black

    color_t color;
    color.r=0;
    color.b=0;
    color.g=0;

    //get hit times
```

```
    double hit_times[nobjects];
    for (int i=0; i < nobjects; i++){
            if (object[i].type==2)
{
                hit_times[i] = sphere_intersection(dmat_mult(&object[i].Minv, e),dmat_mult(&object
[i].Minv, d));
                }else if (object[i].type==1){
                        hit_times[i] = plane_intersection(dmat_mult(&object[i].Minv, e),dmat_mult(&object
[i].Minv, d));
                }else if (object[i].type==0){
                        hit_times[i] = infinite_plane_intersection(dmat_mult(&object[i].Minv, e),dmat_mult
(&object[i].Minv, d));
                }

        }

        //get the min hit time
        int this_obj = find_min_hit_time(hit_times);
        double t = -1.0;
        if (this_obj!=-1){
        t = hit_times[this_obj];
        }

    //if there is an intersection
    if(t!=-1.0){
                //get some points and vectors that we will need
                dmatrix_t *intersection = intersection_coordinates(dmat_mult(&object[this_obj].Minv,
e),dmat_mult(&object[this_obj].Minv, d),t);
                dmatrix_t *surface_normal = normal_to_surface(&object[this_obj], intersection);
                dmatrix_t *light_position = &light->position;
                dmatrix_t *vector_to_light = vector_to_light_source(intersection, light_position);

                //determine if pixel is engulfed in shadow *nore that this isnt working properly*

                int covered=0;
                double cur_hit_time;
                for (int j=0; j < nobjects; j++){
                                if (object[j].type==2)
{
                                cur_hit_time = sphere_intersection(intersection, vector_to_light);
                                if (cur_hit_time>0 && cur_hit_time<1){
                                        covered = 1;
                                }
                                }else if (object[j].type==1){
                                        cur_hit_time = plane_intersection(intersection, vector_to_light);
                                if (cur_hit_time>0 && cur_hit_time<1){
                                        covered = 1;
                                }
                                }else if (object[j].type==0){
                                        cur_hit_time = sphere_intersection(intersection, vector_to_light);
                                if (cur_hit_time>0 && cur_hit_time<1){
                                        covered = 1;
                                }
                                }
                        }
                        if (covered!=1)
{

                        //calculate Diffuse compenent
                        double diffuse_val = ddot_product(vector_to_light,surface_normal)/(dmat_norm
(vector_to_light)*dmat_norm(surface_normal));
                        if (diffuse_val<0){
                                diffuse_val = 0;
                        }
                        double diff_r = light->intensity.r*object[this_obj].diffuse_color.r*object
[this_obj].diffuse_coeff*diffuse_val;
                        double diff_b = light->intensity.b*object[this_obj].diffuse_color.b*object
```

```
[this_obj].diffuse_coeff*diffuse_val;
                        double diff_g = light->intensity.g*object[this_obj].diffuse_color.g*object
[this_obj].diffuse_coeff*diffuse_val;
                        //add diffuse component
                        color.r = color.r + diff_r;
                        color.b = color.b + diff_b;
                        color.g = color.g + diff_g;

                        //calculate specular component
                        dmatrix_t *r = vector_to_specular_reflection(surface_normal, vector_to_light);
                        dmatrix_t *v = vector_to_center_of_projection(intersection, e);
                        double spec_val = pow((ddot_product(r,v))/(dmat_norm(r)*dmat_norm(v)),object
[this_obj].f);
                        delete_dmatrix(r);
                        delete_dmatrix(v);
                        if (spec_val<0){
                                spec_val=0;
                        }
                        double spec_r = light->intensity.r*object[this_obj].specular_color.r*object
[this_obj].specular_coeff*spec_val;
                        double spec_b = light->intensity.b*object[this_obj].specular_color.b*object
[this_obj].specular_coeff*spec_val;
                        double spec_g = light->intensity.g*object[this_obj].specular_color.g*object
[this_obj].specular_coeff*spec_val;
                        //add specular components
                        color.r = color.r + spec_r;
                        color.b = color.b + spec_b;
                        color.g = color.g + spec_g;
                }
                //calculate ambient component and add it to the color of this pixel
                color.r = color.r + light->intensity.r*object[this_obj].ambient_color.r*object
[this_obj].ambient_coeff;
                color.b = color.b + light->intensity.b*object[this_obj].ambient_color.b*object
[this_obj].ambient_coeff;
                color.g = color.g + light->intensity.g*object[this_obj].ambient_color.g*object
[this_obj].ambient_coeff;
                //clean up unused matrices

                delete_dmatrix(intersection);
                delete_dmatrix(surface_normal);
                //delete_dmatrix(light_position);
                delete_dmatrix(vector_to_light);
            }else{
                color=background;
            }
        //multiply by max intensity
        color.r = color.r * MAX_INTENSITY;
    color.b = color.b * MAX_INTENSITY;
        color.g = color.g * MAX_INTENSITY;

    return color;
}

object_t *build_object(int object_type, dmatrix_t *M, color_t ambient_color, color_t diffuse_color,
color_t specular_color, double ambient_coeff, double diffuse_coeff, double specular_coeff, double f,
double reflectivity) {

    object_t *object ;

    object = malloc(sizeof(*object));

    object->type = object_type ;
    object->M = *M ;

    dmat_alloc(&object->Minv,4,4) ;
    object->Minv = *dmat_inverse(&object->M) ;
```

```
    object->reflectivity = reflectivity ;

    object->specular_color.r = specular_color.r ;
    object->specular_color.g = specular_color.g ;
    object->specular_color.b = specular_color.b ;
    object->specular_coeff = specular_coeff ;
    object->f = f ;

    object->diffuse_color.r = diffuse_color.r ;
    object->diffuse_color.g = diffuse_color.g ;
    object->diffuse_color.b = diffuse_color.b ;
    object->diffuse_coeff = diffuse_coeff ;

    object->ambient_color.r = ambient_color.r ;
    object->ambient_color.g = ambient_color.g ;
    object->ambient_color.b = ambient_color.b ;
    object->ambient_coeff = ambient_coeff ;

    switch (object_type) {

        case SPHERE         :   object->intersection_function = &sphere_intersection ;
                                break ;

        case PLANE          :   object->intersection_function = &plane_intersection ;
                                break ;

        case INFINITE_PLANE :   object->intersection_function = &infinite_plane_intersection ;
                                break ;
    }

    nobjects++ ;
    return(object) ;
}

int main() {

    Display *d ;
    Window w ;
    XEvent e ;

    int i, j, s ;

    camera_t Camera ;
    window_t Window ;
    light_t light ;
    dmatrix_t M, light_position ;
    color_t pixel, background, light_intensity, light_color, ambient_color, diffuse_color,
specular_color ;
    double height, width, aspect, ambient_coeff, diffuse_coeff, specular_coeff, f, reflectivity ;

    /* Set the background color */

    background.r = 0.1 ;
    background.g = 0.1 ;
    background.b = 0.1 ;

    /* Set up light position, intensity, and color */

    dmat_alloc(&light_position,4,1) ;

    light_position.m[1][1] = Lx ;
    light_position.m[2][1] = Ly ;
    light_position.m[3][1] = Lz ;
    light_position.m[4][1] = 1.0 ;

    light_intensity.r = 1.0 ;
    light_intensity.g = 1.0 ;
```

```
    light_intensity.b = 1.0 ;

    light_color.r = 1.0 ;
    light_color.g = 1.0 ;
    light_color.b = 1.0 ;

    light = *build_light(&light,&light_position,light_color,light_intensity) ;

    /* Build display window and synthetic camera */

    Window = *build_window(&Window,H,ASPECT) ;
    Camera = *build_camera(&Camera,&Window) ;

    /* Create sphere 1 */
    dmat_alloc(&M,4,4) ;
    M = *dmat_identity(&M) ;
    reflectivity = 0.2 ;

    //transformation
    //tranlations
    M.m[1][4] = 4.0 ;
    M.m[2][4] = 4.0 ;
    M.m[3][4] = 1.0 ;

    specular_color.r = 1.0 ;
    specular_color.g = 1.0 ;
    specular_color.b = 1.0 ;
    specular_coeff = 0.3 ;
    f = 10.0 ;

    diffuse_color.r = 0.0 ;
    diffuse_color.g = 1.0 ;
    diffuse_color.b = 0.0 ;
    diffuse_coeff = 0.6 ;

    ambient_color.r = 0.0 ;
    ambient_color.g = 1.0 ;
    ambient_color.b = 0.0 ;
    ambient_coeff = 0.1 ;


    object[nobjects] = *build_object
(SPHERE,&M,ambient_color,diffuse_color,specular_color,ambient_coeff,diffuse_coeff,specular_coeff,f,reflectivit


    /* Create sphere 2 */
    dmat_alloc(&M,4,4) ;
    M = *dmat_identity(&M) ;
    reflectivity = 0.2 ;

    //transformation
    //tranlations
    M.m[1][4] = 4.0 ;
    M.m[2][4] = -4.0 ;
    M.m[3][4] = 1.0 ;
    //scaling
    M.m[1][1] = 1.5 ;
    M.m[2][2] = 1.5 ;
    M.m[3][3] = 1.5 ;

    specular_color.r = 1.0 ;
    specular_color.g = 1.0 ;
    specular_color.b = 1.0 ;
    specular_coeff = 0.3 ;
    f = 10.0 ;

    diffuse_color.r = 1.0 ;
```

```
    diffuse_color.g = 0.0 ;
    diffuse_color.b = 0.0 ;
    diffuse_coeff = 0.6 ;

    ambient_color.r = 1.0 ;
    ambient_color.g = 0.0 ;
    ambient_color.b = 0.0 ;
    ambient_coeff = 0.1 ;


   object[nobjects] = *build_object
(SPHERE,&M,ambient_color,diffuse_color,specular_color,ambient_coeff,diffuse_coeff,specular_coeff,f,reflectivit

    /* Create sphere 3 */
    dmat_alloc(&M,4,4) ;
    M = *dmat_identity(&M) ;
    reflectivity = 0.2 ;

    //transformation
    //tranlations
    M.m[1][4] = -4.0 ;
    M.m[2][4] = -4.0 ;
    M.m[3][4] = 1.0 ;

    specular_color.r = 1.0 ;
    specular_color.g = 1.0 ;
    specular_color.b = 1.0 ;
    specular_coeff = 0.3 ;
    f = 10.0 ;

    diffuse_color.r = 0.5 ;
    diffuse_color.g = 0.0 ;
    diffuse_color.b = 0.5 ;
    diffuse_coeff = 0.6 ;

    ambient_color.r = 0.5 ;
    ambient_color.g = 0.0 ;
    ambient_color.b = 0.5 ;
    ambient_coeff = 0.1 ;


   object[nobjects] = *build_object
(SPHERE,&M,ambient_color,diffuse_color,specular_color,ambient_coeff,diffuse_coeff,specular_coeff,f,reflectivit

    /* Create sphere 4 */
    dmat_alloc(&M,4,4) ;
    M = *dmat_identity(&M) ;
    reflectivity = 0.2 ;

    //transformationS
    //tranlations
    M.m[1][4] = -4.0 ;
    M.m[2][4] = 4.0 ;
    M.m[3][4] = 1.0 ;

    specular_color.r = 1.0 ;
    specular_color.g = 1.0 ;
    specular_color.b = 1.0 ;
    specular_coeff = 0.3 ;
    f = 10.0 ;

    diffuse_color.r = 0.0 ;
    diffuse_color.g = 0.30 ;
    diffuse_color.b = 0.70 ;
    diffuse_coeff = 0.6 ;

    ambient_color.r = 0.0;
```

```
    ambient_color.g = 0.30 ;
    ambient_color.b = 0.70 ;
    ambient_coeff =
0.1 ;

    object[nobjects] = *build_object
(SPHERE,&M,ambient_color,diffuse_color,specular_color,ambient_coeff,diffuse_coeff,specular_coeff,f,reflectivit


            dmat_alloc(&M,4,4) ;
            M = *dmat_identity(&M) ;

            M.m[1][4] = 0 ;
            M.m[2][4] = 0 ;
            M.m[3][4] = -2.0 ;

            M.m[1][1] = 16.0;
            M.m[2][2] = 16.0;

            reflectivity = 0.5 ;

            specular_color.r = 1.0 ;
            specular_color.g = 1.0;
            specular_color.b = 1.0;
            specular_coeff = 0.3 ;
            f = 10.0 ;

            diffuse_color.r = 0.75;
            diffuse_color.g = 0.75;
            diffuse_color.b = 0.75;
            diffuse_coeff = 0.6 ;

            ambient_color.r = 0.75 ;
            ambient_color.g = 0.75;
            ambient_color.b = 0.75;
            ambient_coeff = 0.1 ;

            object[nobjects] = *build_object
(PLANE,&M,ambient_color,diffuse_color,specular_color,ambient_coeff,diffuse_coeff,specular_coeff,f,reflectivity


    //set up x11 things
    aspect = ASPECT ;
    height = Near*tan(M_PI/180.0*THETA/2.0) ;
    width = height*aspect ;

    dmatrix_t *direction ;

    d = InitX(d,&w,&s,&Window) ;
    XNextEvent(d, &e) ;

    while (1) {
        XNextEvent(d,&e) ;
        if (e.type == Expose) {

            for (i = 0 ; i < Window.width ; i++) {
                for (j = 0  ; j < Window.height ; j++) {

                    direction = ray_direction(&Camera,&Window,height,width,(double)i,(double)j) ;
                    pixel = shade(&light,object,&Camera.E,direction,background) ;
                    SetCurrentColorX(d,&(DefaultGC(d,s)),(int)pixel.r,(int)pixel.g,(int)pixel.b) ;
                    SetPixelX(d,w,s,i,Window.height - (j + 1)) ;
                    delete_dmatrix(direction) ;
                }
            }
        }
        if (e.type == KeyPress)
```

```
            break ;
        if (e.type == ClientMessage)
            break ;
    }
    QuitX(d,w) ;
}
```