REAL-TIME SUPERIMPOSING A PHOTO-REALISTIC VIRTUAL OBJECT ONTO A REAL MOVIE

写実的な仮想物体を実動画中に
リアルタイムに埋め込む。

by

Morihiro Hayashida

林田 守広

A Senior Thesis
卒業論文

Submitted to

the Department of Information Science

the Faculty of Science

the University of Tokyo

on February 14, 2000

in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Thesis Supervisor: Katsushi Ikeuchi　池内 克史
Professor of Information Science

ABSTRACT

When people superimpose a virtual object painted by CG on the photograph of taking the place where there are no object, they often make up shadows, and put it on a suitable place by their sense of perception. However, we can't obtain truly the real image by using its method. Therefore, in order to remove these ambiguity, our paper describes a method of making shadows by using a fish-eye image, and of keeping geometrical order, which have been already suggested.

But, they have often superimposed onto only static images because it also costs lots of time to render a virtual object by using Radiosity method in order to obtain the real image. Accordingly, in this paper, by applying their theories to static images, we'll propose and implement the method to superimpose onto a real movie, moreover to keep a rendered virtual object high quality.

論文要旨

何も無い場所を撮った写真にＣＧで描いた仮想物体を埋め込むのに、人はしばしば見た目による手作業で陰影づけや配置を行ってきた。しかしそれでは本当にリアルな画像は得られない。そこで、これら人手による曖昧さを排除しより写実的な画像を得るために、すでに提案されている、魚眼画像を用いた陰影づけの手法と仮想物体の幾何的整合性をつける方法について述べる。

しかしこれらの手法は、仮想物体にリアル感を出すためにレンダリングにラジオシティ法などの時間のかかる方法を用いていることもあって、もっぱら静止画像についてのみ埋め込みが行われてきた。そこで本論文では、静止画像についての理論を応用し、仮想物体のレンダリング画像は高品質に保ちつつも、リアルタイムに実動画中に埋め込む方法について提案し実装する。

Acknowledgements

I would like to thank Prof.Ikeuchi,I.Sato,Y.Sato,T.Oishi and S.Sato for useful advises and discussions.And I thank people of Computer Vision Lab., Sato Lab. and Sakauchi Lab. for their equipments etc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We can see advertisements and commercials of including both CG and an actual scene. For example, remember the advertisements of houses or buildings. Those images include a beautiful drawn house or so in full of verdure. It looks like very good. However, they have been usually drawn by intuition, and lacking in reality because the drawing method is not based on the physical laws. In this paper, first in order to obtain photo-realistic images, we describe the consistency of geometry and illumination to follow the physical laws. Next, we propose the method of rendering on realtime which we derived by applying these theories. We use a famous free software "Radiance" [1] for rendering virtual objects, which software renders with the ray-tracing and radiosity method, and the rendered images are photo-realistic. But, we can't obtain the superimposed image on realtime by using this software. Nevertheless, we describe the theory to superimpose a virtual object onto a real movie, and implement this. We maybe could say it a kind of virtual reality. Finally, we show some results, and discuss the conclusion and the future work.

Chapter 2

Theory and Method to Superimpose

In this chapter, we explain the theories and methods to superimpose a virtual object onto a real static image. First, we explain Camera Model proposed by Tsai [2], one of the methods to put a virtual object correctly on the photograph. Next, we explain how to render real shadows [3].

2.1 Consistency of Geometry

First, in order to put a virtual object, we must define a three-dimensional coordinate system in the real world. Then, we can put the object on an appropriate place of its coordinate system. However, because the image which we are going to superimpose is two-dimensional coordinates, we need to know a transformation from the three dimensional real world coordinates to the two dimensional image coordinates.

We usually regard the transformation by camera as a projective transformation.

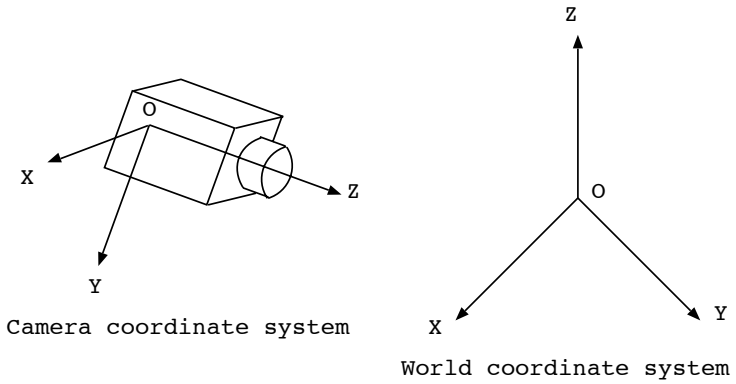

Figure 2.1: World and Camera coordinate system

The rendering soft "Radiance" also uses a perspective view, or the projective transformation. Therefore, we use Tsai's camera model to adopt the projective transformation. In addition, Tsai's model has been considered in respect of (1) radial distortion, (2) displacement of the image center, and (3) mismatching between camera and frame grabber horizontal scan rates, which cause image distortion. Owing to this, we can calibrate with high accuracy.

2.1.1 Tsai's camera model

Tsai's camera model supposes simply an ideal pinhole camera. This is a rectangular box with a convex lens. Its pinhole camera transforms from three to two dimensional coordinate system like other cameras.

Firstly, the three dimensional coordinates of a point $\vec{P}$, denoted $\vec{P_w}$, are mapped to three dimensional camera coordinates $\vec{P_c}$ by a rotation R about the world origin followed by a translation T (Figure 2.1). The three dimensional camera coordinate system is centered at the camera's optical center (where optical rays converge in the absense of distortion). Viewed in the direction the camera is pointing, the

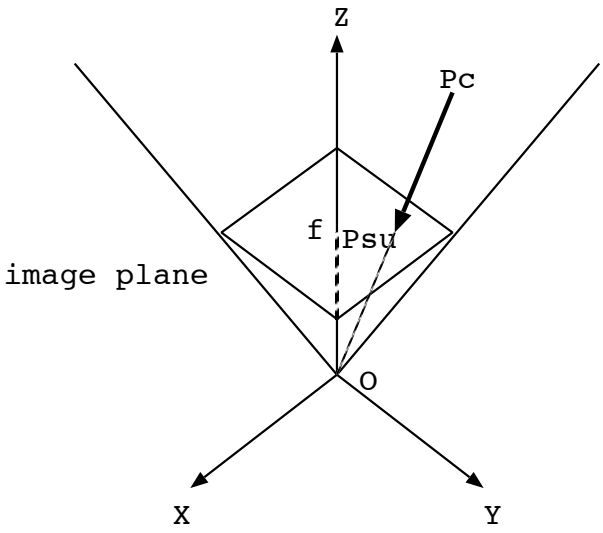

Figure 2.2: Projection from Camera coordinate system to Camera sensor plane

camera coordinate system's $+X$ axis points right, its $+Y$ axis points down, and its $+Z$ axis lies along the optical axis. The transformation from world to camera coordinates is

$$\vec{P_c} = R\vec{P_w} + T \tag{2.1}$$

Secondly, the point $\vec{P_c}$ is projected onto the camera sensor. The focal length is f and the projected point, in an ideal undistorted sensor coordinate system (Figure 2.2), has coordinates $\vec{P_{su}}$ given by

$$P_{su}^x = fP_c^x/P_c^z \tag{2.2}$$

$$P_{su}^y = fP_c^y/P_c^z \tag{2.3}$$

Thirdly, the projected point is subjected to radial lens distortion, mapping it from undistorted sensor coordinate $\vec{P_{su}}$ to distorted sensor coordinates $\vec{P_{sd}}$. This distortion is modeled by a single factor proportional to the square of the distance

from the optical axis to the projected point. This factor is parameterized by κ_1:

$$P_{sd}^x = P_{su}^x(1 + \kappa_1|\vec{P_{su}}|^2) \tag{2.4}$$

$$P_{sd}^y = P_{su}^y(1 + \kappa_1|\vec{P_{su}}|^2) \tag{2.5}$$

Finally, the point is mapped from distorted sensor coordinates $\vec{P_{sd}}$ to distorted image coordinates $\vec{P_{id}}$. This mapping is determined by the image coordinates of the image center, $\vec{C}$, the size of image pixels in sensor coordinates, $\vec{D}$, and a factor s accounting for mismatching between sensor and frame grabber horizontal scan rates. Unlike the sensor and camera, the image's coordinate system has its origin at the lower left corner of the image, its $+X$ axis points right, and its $+Y$ axis points up.

$$P_{id}^x = C^x + sP_{sd}^x/D^x \tag{2.6}$$

$$P_{id}^y = C^y - P_{sd}^y/D^y \tag{2.7}$$

The extrinsic camera parameters (R, T) among those camera parameters (R, T, f, κ_1, s) are necessary for rendering a virtual object. on the other hand, the intrinsic camera parameters (f, κ_1, s) are provided when we decide a camera to use. And $\vec{P_{id}}$ is obtained from a taken image. Furthermore, we can provide $\vec{P_w}$. Thus, we can get R, T by using Tsai's camera model.

2.1.2 Calibration method

In order to obtain the extrinsic camera parameters, we use the calibration board which has latticed points (Figure 2.3, and see also Figure **??**). The center of these points is the origin, Z axis stands vertical to the plane of the calibration board. Thus, because we can get the correspondence of three dimensional coordinates in the real scene ($\vec{P_w}$) and two dimensional coordinates in the taken image ($\vec{P_{id}}$), we'll

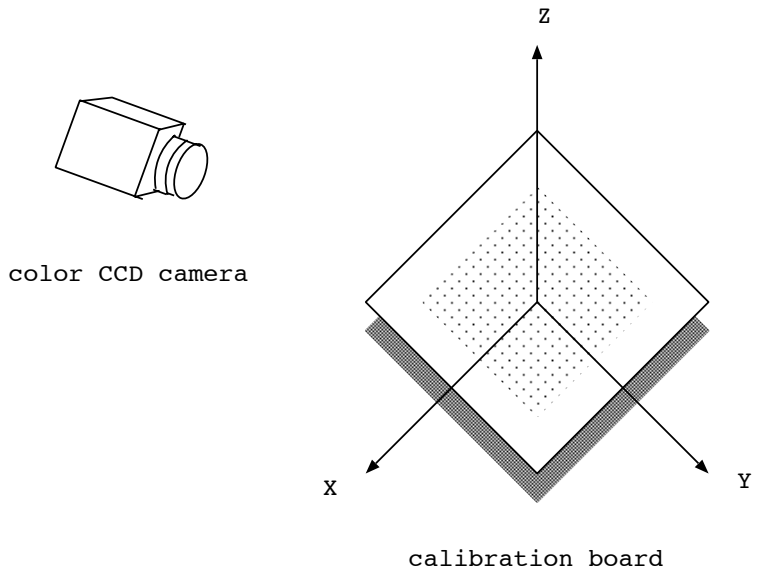

Figure 2.3: Image acquisition system

obtain the camera parameters (R, T) by using Tsai's camera model, and render the virtual object on its geometry.

2.2 Consistency of Illumination

In this section, we explain how to generate shadows on a virtual object. The shadows are important because they provide a sense of existense on its object.

2.2.1 Light source environment

Under the complicated light source environment, we need to think not only direct light sources like a fluorescent light, but also indirect light sources like reflected lights from a wall. In this paper, we consider a hemisphere surface light source to deal with any light sources from omnidirection. And then, we derive the total

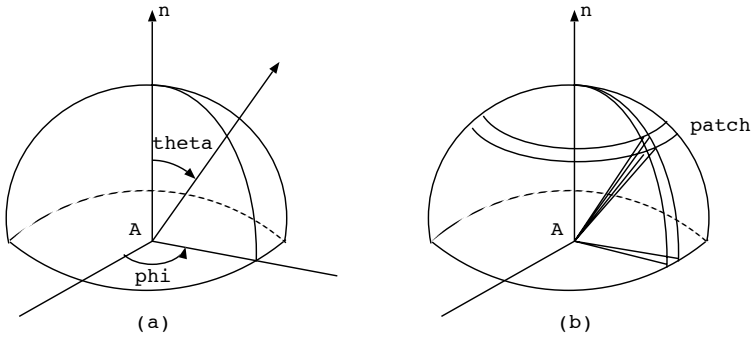

Figure 2.4: (a)The direction of incident and emmited light rays. (b)Infinitesimal patch of an extended light source.

irradiance at a point A from the radiance map. First, consider an infinitesimal patch of the extended light source, of size $\partial\theta_i$ in polar angle and $\partial\phi_i$ in azimuth (Figure 2.4). Seen from the center point A, this patch subtends a solid angle $\partial\omega = \sin\theta_i \partial\theta_i \partial\phi_i$. If we let $L_0(\theta_i, \phi_i)$ be the radiance per unit solid angle coming from the direction (θ_i, ϕ_i), then the radiance from the patch under consideration is $L_0(\theta_i, \phi_i) \sin\theta_i \partial\theta_i \partial\phi_i$,[4], and the total irradiance E_0 of the hemisphere surface is

$$E_0 = \int_{-\pi}^{\pi} \int_{0}^{\frac{\pi}{2}} L_0(\theta_i, \phi_i) \sin\theta_i \partial\theta_i \partial\phi_i \tag{2.8}$$

As the total irradiance light sources, it is natural to use the fisheye image like Figure 2.5. However, we can't compute the total irradiance E_0 in the formula 2.8 because its formura include the double integral. Therefore, we approximate the double integral by discrete sampling over entire hemisphere. We use nodes of a geodesic dome[5](Figure 2.6). For each node, the radiance of a corresponding point in the real scene is used as the radiance of the node. Consider a ray from a point on a virtual object surface to the node. A color at an intersection of the ray

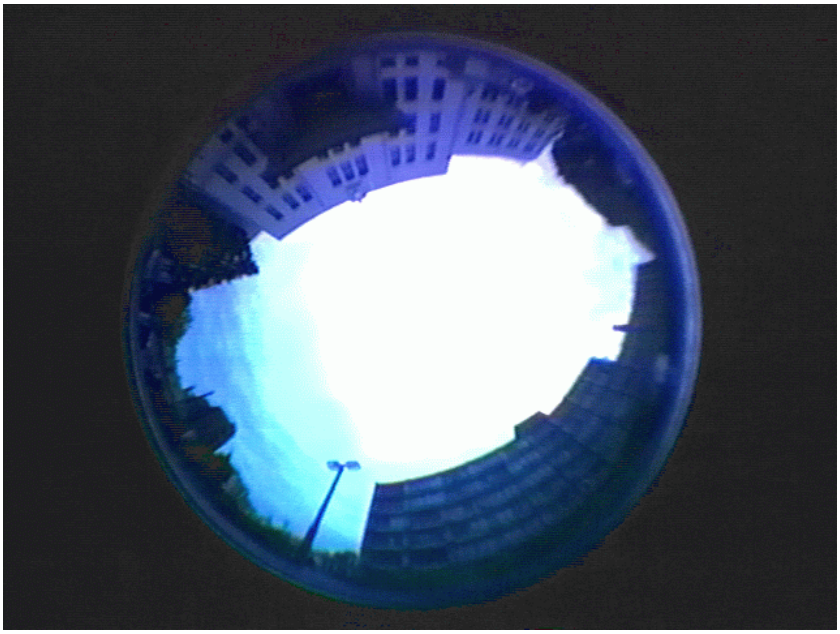

Figure 2.5: Fisheye image

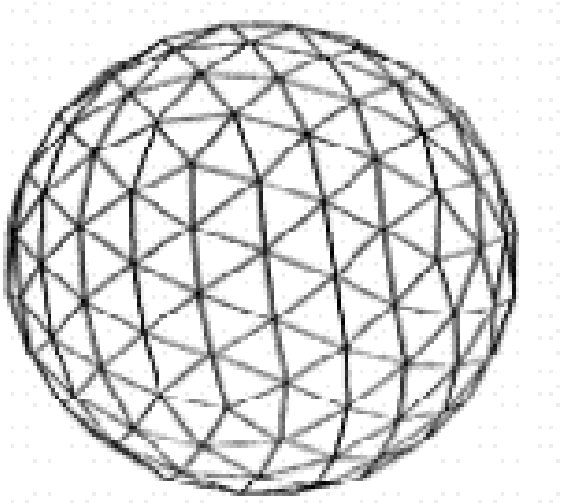

Figure 2.6: Geodesic Dome

and the hemisphere surface (also the fisheye image) is obtained as the radiance. The nodes of a geodesic dome are uniformly distributed over the surface of a sphere. Therefore, by using N nodes of a geodesic dome in a northern hemisphere as a sampling direction, the double integral in the formula 2.8 can be approximated as a sampling at equal solid angle $\partial\omega = 2\pi/N$. The number of the nodes N can be adjusted by changing the sampling frequency of a geodesic dome.

Using the discrete sampling, the formula 2.8 can be approximated as

$$E = \sum_{i=0}^{N} \frac{2\pi}{N} L(\theta_i, \phi_i) \tag{2.9}$$

Note that a radiance does not depend on the distance between a viewpoint and a light source. Therefore, the distance from the point on the virtual object to the real scene does not affect the radiance. Also, we assume that the real scene reflects or emits light as a perfect Lambertian plane. In other words, at each surface point in the scene, light energy is emitted equally in all directions. Owing to this assumption, we do not consider directional light sources such as spotlights in our method. However, by combining multiple omnidirectional images taken from different locations, our method could be extended to model changes of a radiance, depending on viewing directions. Next, we are going to describe the geodesic dome.

Geodesic dome

In this section, we describe a method to generate a node of a geodesic dome. By dividing a regular twenty surface box, we can increase the number of nodes N (Figure 2.7). First, a regular twenty surface box has twelve nodes. Then, we divide each triangle surface of the twenty surface box. In this figure, the number of division (f) is two. (1) Divide $\angle BOC$ into f angles, and name the intersection

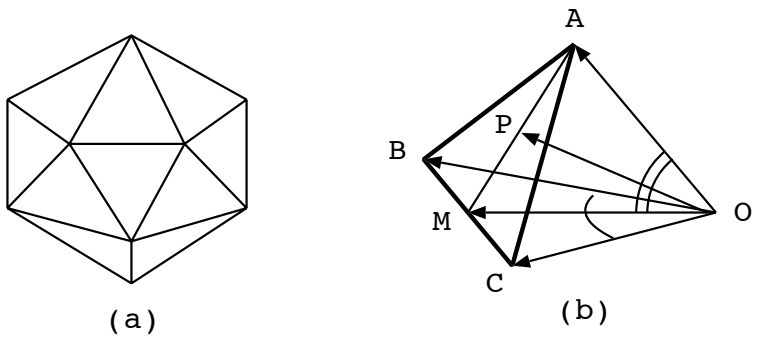

Figure 2.7: (a)Regular 20 surface box, (b)Division of a triangle surface

point M between the side BC and one of the lines of the divided angle. (2) Furthermore, divide $\angle AOM$ into f angles. (3) Finally, extend the vector $\vec{OP}$ so that the point P lies on the sphere. In this paper, we use only up half of the geodesic dome nodes.

2.2.2 Obtaining color values from a fisheye image

A CCD camera with a fisheye lens is used to take omnidirectional images of the real scene. The fisheye lens[1] which we use has characters as follows:

- Field of view) diagonal:180 degrees, horizontal:180 degrees, vertical:180 degrees
- Method of projection) equally distance projection transformation

This fisheye lens project an incoming ray to its lens onto an imaging plane as

$$r = f\theta \tag{2.10}$$

[1] Fisheye lens made by FIT corporation, FI-19

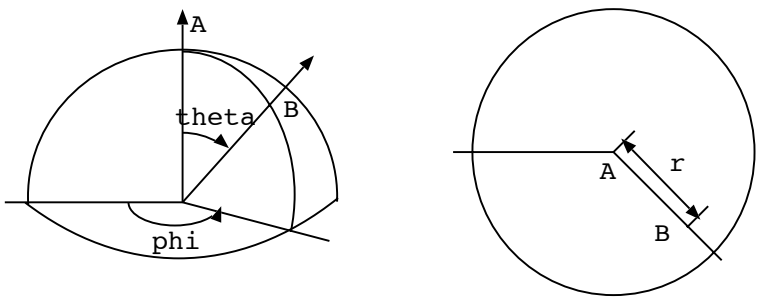

Figure 2.8: Omnidirectional image acquisition system

where f is the focal length of the lens, r is the distance between the image center and the projection of the ray, and θ is an incident angle of the ray (Figure 2.8). Using this projection model, the incident angle of the ray is given as $\theta = r/f$, where r is determined from the image coordinates of the point corresponding to the ray. For instance, if a direct light source appears as a point in the omnidirectional image, the direction from the camera projection center to the light source is determined from the image coordinates of the point in the image. Besides, we can know the value of f by using the fact that the field of view $\theta = \pi/2$ when the image height r is maximum.

Through the formula 2.10, if the direction (θ_i, ϕ_i) of each light source is provided, the fisheye image coordinates (i_x, i_y) corresponding to the light source is as follows:

$$r = f\theta_i \tag{2.11}$$

$$i_x = r\cos\phi_i + c_x \tag{2.12}$$

$$i_y = r\sin\phi_i + c_y \tag{2.13}$$

where (c_x, c_y) is the center coordinates of the fisheye image. The bright on (i_x, i_y) is used when we render the virtual object. The bright of each direction becomes

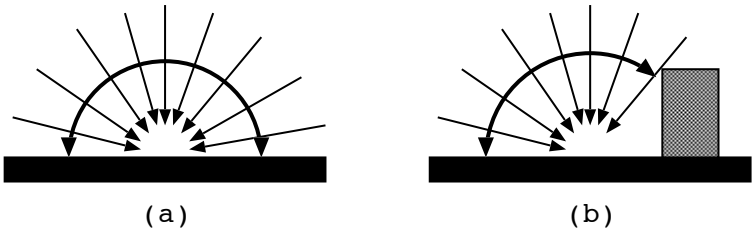

Figure 2.9: Total irradiance:(a) without occluding object (b) with occluding object

the pixel value of the fisheye image on (i_x, i_y).

Besides, we use bilinear combination in order to present better reality. The values (i_x, i_y) are floating points, but the location of the fisheye image (i_x, i_y) is integer. Therefore, we approximate the location from the color values of four integer coordinates near the (i_x, i_y) by using the bilinear combination. However, in order to carry out in high speed, it may be not necessary.

2.2.3 Generating virtual object's shadows

We calculate how much dark the pixels become when the virtual object is put. In figure 2.9 (a), there is no virtual objects, and the omnidirectional lights of the real scene shine the center point on the virtual ground. In contrast, in figure (b), the virtual object occlude the lights. Through the formula 2.9, the total irradiance with the occluding object, E_s, is as the follow:

$$E_s = \sum_{i=0}^{N} \frac{2\pi}{N} L(\theta_i, \phi_i) S(\theta_i, \phi_i) \tag{2.14}$$

$$S(\theta_i, \phi_i) = \begin{cases} 0 & (when\ L(\theta_i, \phi_i)\ is\ occluded) \\ 1 & (otherwise) \end{cases}$$

Because the bright of the pixel on the virtual ground becomes darker E_s from E when the virtual object is placed, the bright of the pixel on the real background

image P becomes also darker as much. Thus, the bright of the pixel on the real background image when the virtual object is placed, P', is by taking a ratio as the follow:

$$P' = P\frac{E_s}{E} \tag{2.15}$$

After all, under the light sources of the real scene in Section 2.2.1, we render the only virtual ground, and render the virtual object and ground. Then, we can apply the formula 2.15 at the image coordinates of the shadows.

2.2.4 Superimposing a virtual object onto a real static image

We must decide what type each pixel on the superimposed image represents: a virtual object, its shadow, or backgroud. To decide this, we use a mask image such as Figure 2.11. This is the rendered image by using only ambient light in which the transparent object such as windows is dealed as an opaque object. The red pixels in the mask image are the virtual object zone, and the other pixels are shadows or background. We can decide shadows or background by comparing between the only virtual ground image such as Figure 2.12 and the virtual object and ground such as Figure 2.13. When the pixel value on the object and ground image is smaller than that on the only ground image, the pixel is decided as 'shadow', otherwise 'background'. Thus, by taking the background pixel when the pixel is 'background', by taking the sum over the virtual object and ground images when the pixel is 'object', and by multipling the ratio between two pixels to the background pixel when the pixel is 'shadow', we can superimpose the virtual object onto the real scene. Figure 2.15 is an example of superimposing Figure 2.13 onto Figure 2.14 by using fisheye image of Figure 2.5.

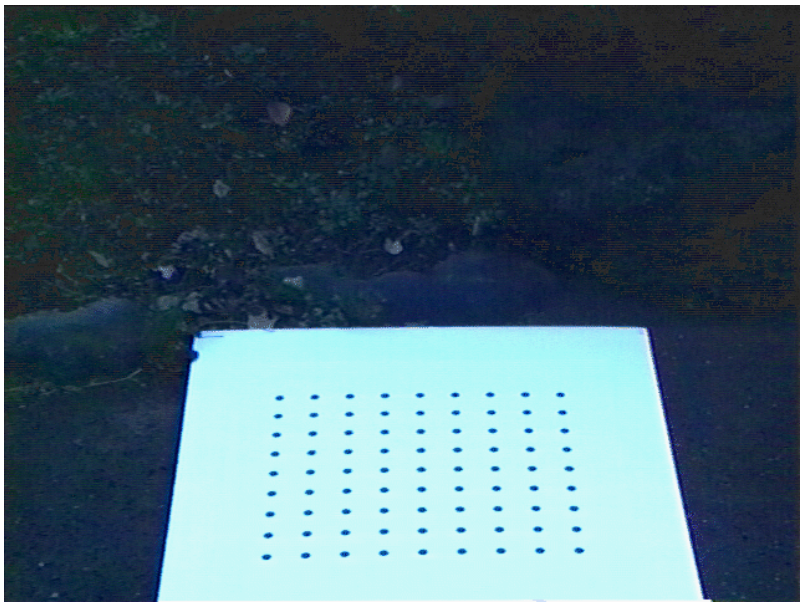

Figure 2.10: Calibration board in a real scene

Figure 2.11: Mask image (a house)

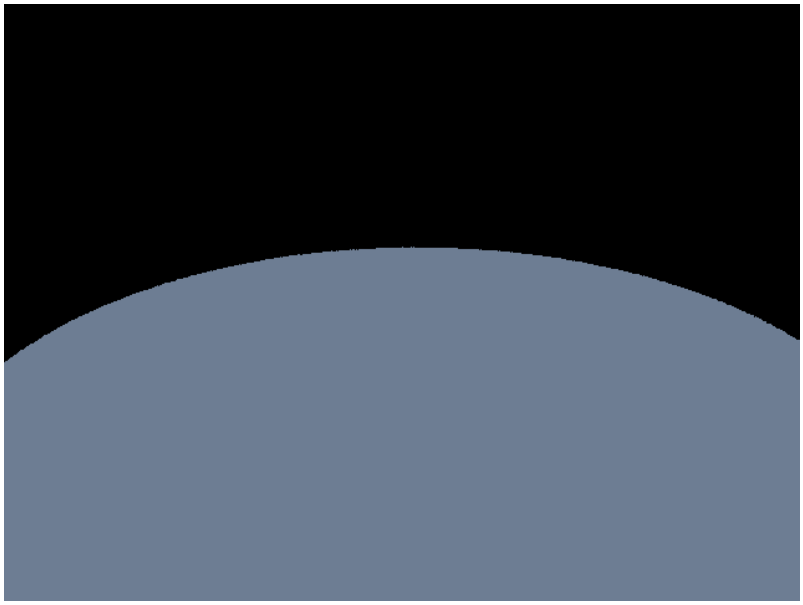

Figure 2.12: Only virtual ground

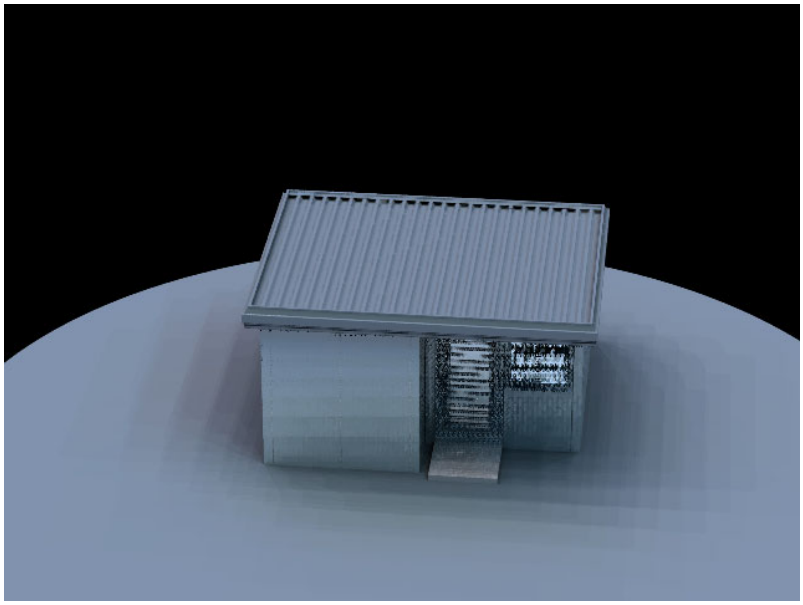

Figure 2.13: Virtual object and ground

Figure 2.14: Background image

Figure 2.15: Superimposed image

Chapter 3

Realtime Rendering

In this chapter, we consider to superimpose a virtual object onto a movie on realtime by applying the theories of the previous chapters. First, we describe the new method to realize realtime rendering. Finally, we implement it.

3.1 Theory

The fundamental idea is the same as the previous theories. The different point is to change the order of rendering a virtual object. In the previous chapter, we render a virtual object after obtaining colors of the light sources from a fisheye image. However, to the contrary, consider to complete the image by obtaining the colors from the fisheye image after having rendered a virtual object. In other words, we can generate N images by rendering a virtual object under each light source instead of generating a image by rendering under N light sources of the geodesic dome. Then, we can compound of these N images into the superimposed image.

3.1.1 Linear combination

First, we render a virtual object under each directional light source which has an unit bright (Example. Figure 3.1 and Figure 3.2). Next, we obtain the colors (R,G,B value) of the corresponding coordinates on the fisheye image with the light source direction. Then, these color values are multiplied to all the pixels on the rendered image under an unit light source. Finally, we total these images to a completed image.

Therefore, we can cost huge time to render a virtual object under each unit light. And, we can improve the quality of the superimposed image very much enough to wish. Thus, we must prepare these N images to superimpose.

3.2 Implementation

3.2.1 Parallel superimposing

The main calculation is to multiply and to add at each pixel. Thereupon, we consider to distribute pixels of the image to each processer. However, It is necessary to arrange the number of pixels at each processer because the amount of calculating is different at each pixel. We distribute every pixels by the amount of dealing with data such as the color RGB values, or the type ('shadow','virtual ground', or 'ground and object' See Section 2.2.4) of the pixel. This method is superior to distributing every pixels by the number of dealing with the pixels.

3.2.2 High-speed superimposing

Next, we consider not to calculate the linear combination about weaker light source than a threshold value. When N is enough large, although the threshold value is

Figure 3.1: Only a virtual ground under an unit bright light source

Figure 3.2: Virtual ground and object under an unit bright light source

some large, the shodaws of the virtual object become natural. Moreover, we can shorten the elapsed time to calculate.

Chapter 4

Experimental Results

In this chapter, we show the results. First, we explain the devices which we used on this time. Then, we show the serial superimposed images.

4.1 Setting devices

4.1.1 3CCD video camera with a fisheye lens

Look at Figure 4.1. There are a yellow bus, a chair, and trees on a plane. On the center of the plane, there is a camera with a fisheye lens. This camera connects to a composer which takes up to four video inputs, and outputs a video stream. In trouble, this camera reverses a fisheye image vertically. However, in our program, it is considered.

4.1.2 CCD video camra with a normal lens

There is a camera in front of the plane. This camera takes a background image stream. And this camera connects also to the composer.

Figure 4.1: Experimental scene

4.1.3 Handycam video camera

There is a camera in front of the normal lens camera. We can't see it in Figure 4.1. This camera takes over experimental scene; how we move light sources. This camera connects also to the composer.

4.1.4 Video composer and Digital converter

This composer takes four video streams, and outputs a video stream. The size of each input video image is equal to the size of the output image. And, because this output image is analog data, we must convert it to digital data in order to send to a computer. Because of it, there is the converter. In trouble, this converter abridges an input video image vertically. It is also considered in the program.

4.1.5 Parallel computer

This is the machine to calculate the superimposed image. It is ONYX2 which has six IP27 processors running with 195 MHz. Its processor consists of MIPS R10000 CPU and MIPS R10010 FPU. The size of main memory is 2048Mbytes, the size of instruction or data cache is 32Kbytes, and secondary unified cache size is 4Mbytes.

4.1.6 Display

Because this can display 'TrueColor' images, we can output PPM images without compressing color values. Our program outputs a image divided into four images to a X Window as soon as it obtain the superimposed image.

4.2 Results

4.2.1 Some Pictures

Figure 4.2-4.5 and 4.6 are examples of results. The up left image at each figure is the experimental scene. The up right image is a fisheye image which red points show that its points are included to calculate the superimposed image, and which blue points show that its points are not included. The down left image is the image before the program superimpose. To the contrary, the down right image is the image after the program superimposed. Figure 4.6 were mixed with a red lamp. In these fisheye images, the upper red points are secondary light sources from the screen. And, we can see that these points lie at equally density on the fisheye image. Besides, we can see that a virtual cow is superimposed with no incongruity.

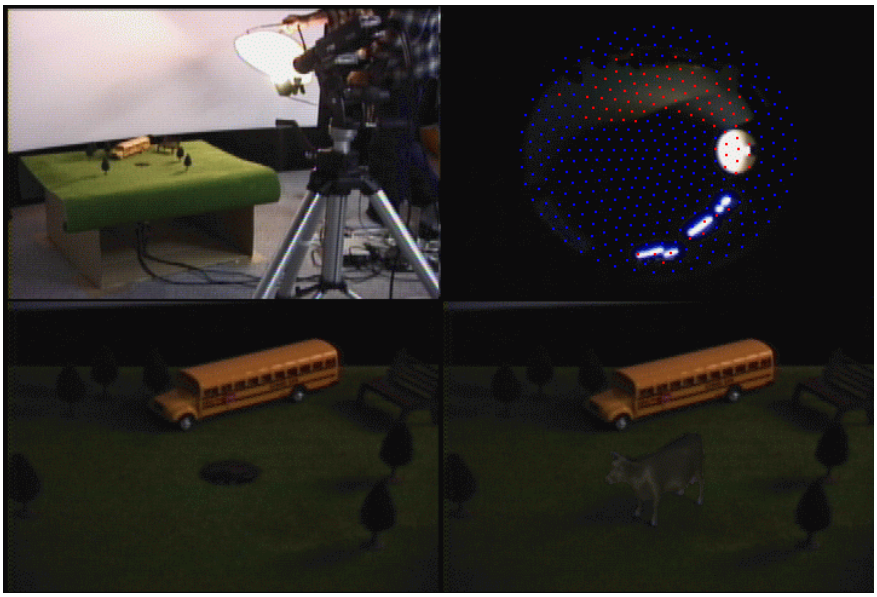

Figure 4.2: First time

The size of these images is the width 720 pixels, and the height 486 pixels. We can't magnify the size because the hardware which we use restricts it to be the constant size, and we can't change by software.

4.2.2 Elapsed rendering time

The rendering time depends on the number of threads, light sources to calculate, and so on. Table 4.1 shows average elapsed time to render a superimposed image when the threshold value is zero, in short, the program calculate all the light sources. The elapsed time under the same precondition rises uneven. One of the reason is the synchronization between the video deamon process of taking the video stream from the hardware and the thread of bringing the video streams to our process. Or, It is considered to be an intervention of OS.

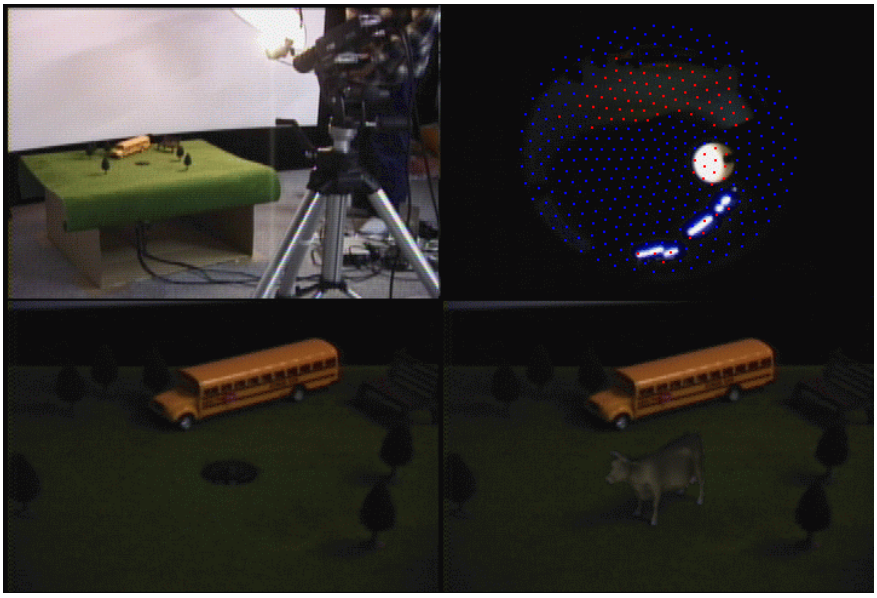

Figure 4.3: Second time

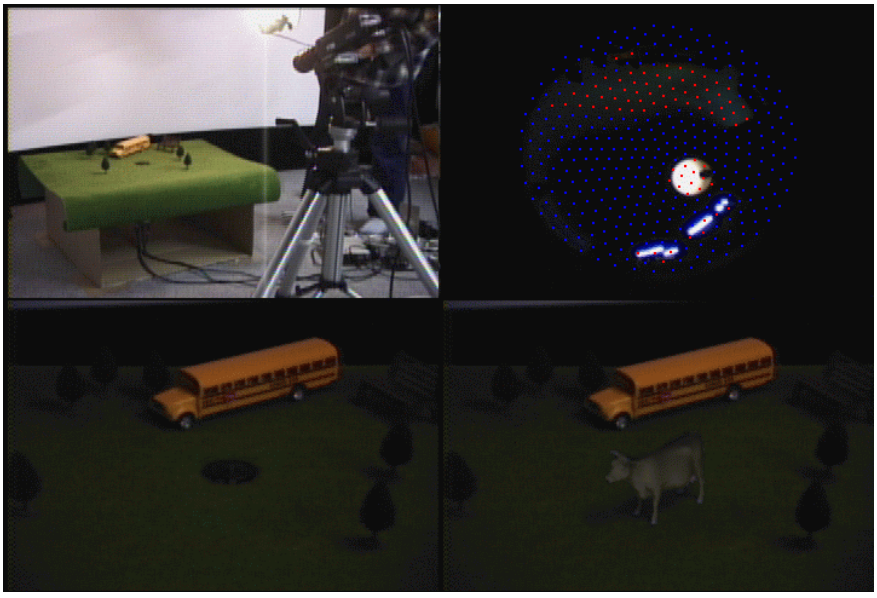

Figure 4.4: Third time

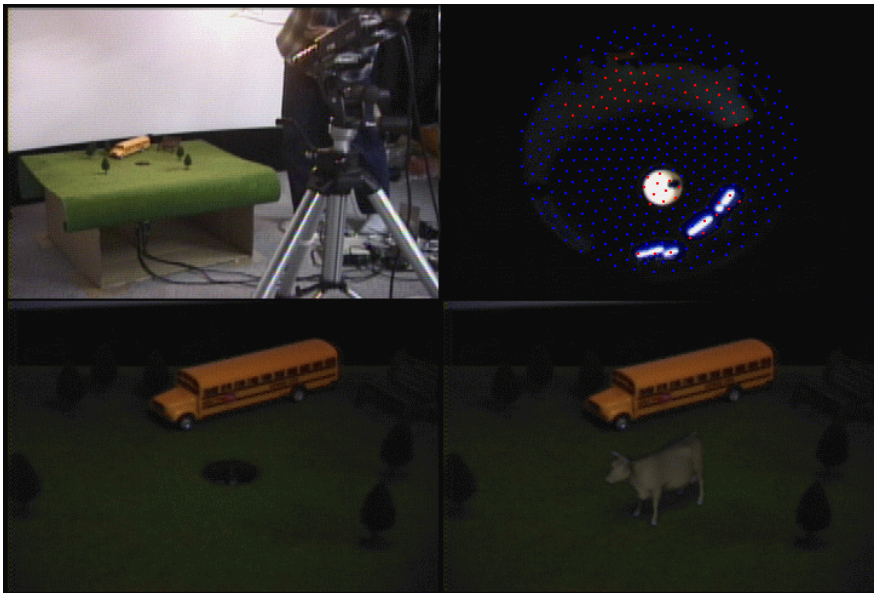

Figure 4.5: Last time

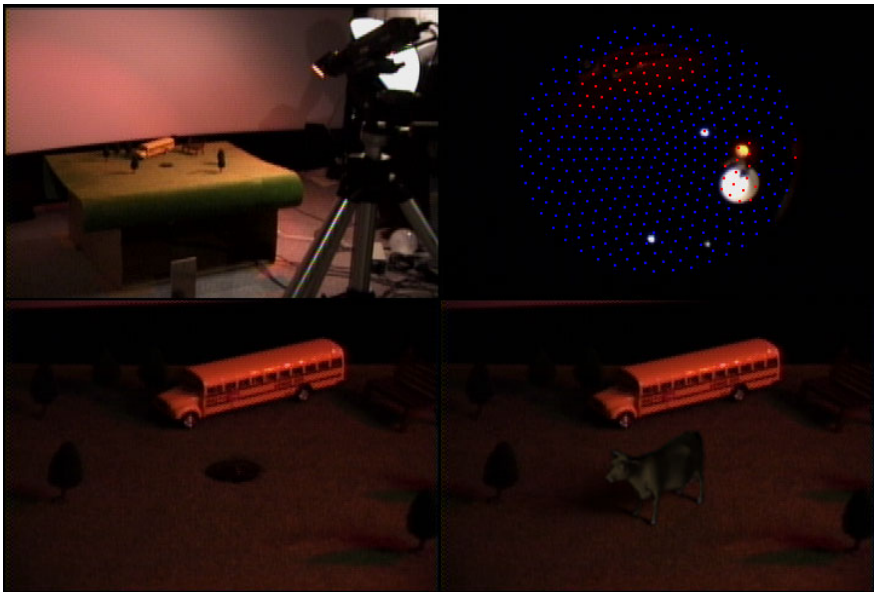

Figure 4.6: Add a red lamp

Table 4.1: Elapsed time when all light sources are computed

light sources	data size (Mbyte)	threads	time (sec)
6	0.61	1	0.09
25	4.9	3	0.2
88	37	8	0.33
191	77	7	0.55
406	200	9	1.3

When we establish a threshold value to the value bigger than zero, the time becomes shorter. For instance, in Figure 4.2-4.6, although the number of all the light sources is 406, the average elapsed time is about 0.4. After all, the program calculate about 180 light sources on the case, and the elapsed time becomes less than half.

Chapter 5

Conclusion

We proposed to superimpose a virtual object onto a video stream on realtime. And then, we think the purpose was achieved sufficiently because our method becomes very much faster than the former method to superimpose onto a static image. However, there are some problems. One of them is that we can't change the view point, nor move the video camera to change the background image. When we want to do so, we must start again from the first step. It costs long time. But, we can move the virtual object like an animation. We only prepare some patterns of rendered images by changing the position of the virtual object.

One of the problems is the sensitivity of CCD video cameras which we used. It is because the brighter points than a color value (also 255) become the value 255. Owing to this, we can't obtain the accurate light source environment, nor generate the accurate superimposed images. Our results were made under lower light source environment.

Appendix A

Main program source code

The below is the main program to superimpose a virtual object onto a video stream in C language.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <unistd.h>

#include "video.h" // taking video frames
```

```
// Input file .msi format is

//    Xsize(int) Ysize(int) Number-of-lights(int)
//    Size-of-this-file(long)
//    Point1(n bytes) Point2 ...

// Point format is

//    Type (unsigned int/1byte)
//    Nothing when Type = PICTURE(see below)
//    R1 G1 B1 R2 G2 B2 ...       when Type = HOUSE
//    Rg1 Gg1 Bg1 Rgh1 Ggh1 Bgh1 Rg2 Gg2 Bg2 Rgh2 Ggh2 Bgh2 ...
//                                when Type = SHADE

#define BUFSIZE (512)
#define COLORMAX (255)
#define DARKRGB (19)   // video image adjust

#define SUNCHR (sizeof(unsigned char))
#define SCHR   (sizeof(char))

// mask[] types
#define PICTURE    (0) // background
#define HOUSE      (1) // virtual object
#define SHADE      (2) // shadow
```

```
int ncpu;         // number of cpu
int pnts;         // number of light
int xsize,ysize,colmax;
int *slp;         // start location of the cpu
double *lpx,*lpy; // light source coordinates on the fisheye image
volatile double *wgtc;     //   RGB values
int oldcnt;
volatile int *elem;
volatile int elemnum;
int darklight;

unsigned char camoutrgb[(CXSIZE*2*CYSIZE*2)*4]; // X Window output
unsigned char fishrgb[CXSIZE*CYSIZE*3]; // fisheye image
double rate;
char *input,*light;
FILE *inputf,*lightf;
unsigned char *mask;  // mask type
unsigned char **hsrgb,**shdrgb; // RGB values of ground and object

volatile int *state; // variable for synchronization

Display *d;  // X Window server id
Window  w,wd;
XVisualInfo xvi;
```

```
GC gc;
XEvent event;

// wait for other threads finishing to operate

void wait(int id)
{
  int i;

  state[id]++;
  for (i=0;i<ncpu;i++) {
    while (state[i]<state[id]);
  }
}

void setfep(char *light) // set each light source (x,y) on the fisheye image
{
  int i;
  double fcx=181; // center X coordinate of fisheye image
  double fcy=119; // center Y coordinate of fisheye image
  double frx=117.5*2/3.14159265; // horizontal focal length
  double fry=105*2/3.14159265; // vertical focal length

  char *buf;
  FILE *lightf;
```

```
buf=(char *)malloc(sizeof(char)*BUFSIZE);


lightf=fopen(light,"r");
if (lightf==NULL) {
  fprintf(stderr,"Can't open %s\n",light);
  exit(0);
}
pnts=0;
while (fgets(buf,BUFSIZE,lightf)!=NULL) {
  pnts++;
}


lpx=(double *)malloc(sizeof(double)*pnts);
lpy=(double *)malloc(sizeof(double)*pnts);
fseek(lightf,(long int)0,SEEK_SET);
i=0;
while (fgets(buf,BUFSIZE,lightf)!=NULL) {
  register double theta=atof(strtok(buf," "));
  register double phai=atof(strtok(NULL," "));
  lpx[i]=-frx*theta*cos(phai)+fcx;   // calculate coordinates
  lpy[i]=fry*theta*sin(phai)+fcy;   //   corresponding to the direction
  i++;
}
fclose(lightf);
```

```
}

void getcolor() // get colors from fisheye image
{
  int i,j,x,y,elemi,pr,pb;
  double dx,dy,px,py,px0,px1;
  unsigned char *p00,*p01,*p10,*p11;

  while ((readableflag==0)||(count<=oldcnt)); // video synchronization
  // always take a new frame, and not read while the stream is being written
  for (i=0;i<CYSIZE*2;i++) { // read video
    for (j=0;j<CXSIZE*2;j++) {
      register int campb=(int)tmpcamrgb[(i*CXSIZE*2+j)*3]-DARKRGB; // adjust RGB
      register int campg=(int)tmpcamrgb[(i*CXSIZE*2+j)*3+1]-DARKRGB;
      register int campr=(int)tmpcamrgb[(i*CXSIZE*2+j)*3+2]-DARKRGB;

      if (campr<0) {
campr=0;
      }
      if (campg<0) {
campg=0;
      }
      if (campb<0) {
campb=0;
      }
```

```
      if ((i<CYSIZE)&&(j>=CXSIZE)) {
fishrgb[((CYSIZE-i)*CXSIZE+CXSIZE+j)*3]=(unsigned char)campr;
fishrgb[((CYSIZE-i)*CXSIZE+CXSIZE+j)*3+1]=(unsigned char)campg;
fishrgb[((CYSIZE-i)*CXSIZE+CXSIZE+j)*3+2]=(unsigned char)campb;
camoutrgb[((CYSIZE-i)*CXSIZE*2+j)*4]=0;
camoutrgb[((CYSIZE-i)*CXSIZE*2+j)*4+1]=(unsigned char)campr;
camoutrgb[((CYSIZE-i)*CXSIZE*2+j)*4+2]=(unsigned char)campg;
camoutrgb[((CYSIZE-i)*CXSIZE*2+j)*4+3]=(unsigned char)campb;
      } else {
camoutrgb[(i*CXSIZE*2+j)*4]=0;
camoutrgb[(i*CXSIZE*2+j)*4+1]=(unsigned char)campr;
camoutrgb[(i*CXSIZE*2+j)*4+2]=(unsigned char)campg;
camoutrgb[(i*CXSIZE*2+j)*4+3]=(unsigned char)campb;
      }
    }
  }
  oldcnt=count; // so that we can't read old video frame

  p00=(unsigned char *)malloc(SUNCHR*3);
  p01=(unsigned char *)malloc(SUNCHR*3);
  p10=(unsigned char *)malloc(SUNCHR*3);
  p11=(unsigned char *)malloc(SUNCHR*3);

  elemi=0;
  for (i=0;i<pnts;i++) {
```

```
px=lpx[i];
py=lpy[i];
if (px<0) {
  px=0.5;
}
if (px>CXSIZE) {
  px=CXSIZE-0.5;
}
if (py<0) {
  py=0.5;
}
if (py>CYSIZE) {
  py=CYSIZE-0.5;
}

dx=px-(double)floor(px);
dy=py-(double)floor(py);
x=floor(px);
y=floor(py);

p00[0]=fishrgb[(y*CXSIZE+x)*3];
p00[1]=fishrgb[(y*CXSIZE+x)*3+1];
p00[2]=fishrgb[(y*CXSIZE+x)*3+2];
p10[0]=fishrgb[(y*CXSIZE+x+1)*3];
p10[1]=fishrgb[(y*CXSIZE+x+1)*3+1];
```

```
    p10[2]=fishrgb[(y*CXSIZE+x+1)*3+2];
    p01[0]=fishrgb[((y+1)*CXSIZE+x)*3];
    p01[1]=fishrgb[((y+1)*CXSIZE+x)*3+1];
    p01[2]=fishrgb[((y+1)*CXSIZE+x)*3+2];
    p11[0]=fishrgb[((y+1)*CXSIZE+x+1)*3];
    p11[1]=fishrgb[((y+1)*CXSIZE+x+1)*3+1];
    p11[2]=fishrgb[((y+1)*CXSIZE+x+1)*3+2];

    px0=p00[0]+dx*(p10[0]-p00[0]);
    px1=p01[0]+dx*(p11[0]-p01[0]);
    wgtc[i*3]=(px0+dy*(px1-px0));   // bilinear combination

    px0=p00[1]+dx*(p10[1]-p00[1]);
    px1=p01[1]+dx*(p11[1]-p01[1]);
    wgtc[i*3+1]=(px0+dy*(px1-px0)); // bilinear combination

    px0=p00[2]+dx*(p10[2]-p00[2]);
    px1=p01[2]+dx*(p11[2]-p01[2]);
    wgtc[i*3+2]=(px0+dy*(px1-px0)); // bilinear combination

    pr=0;
    pb=255;
    if ((wgtc[i*3]>=darklight)||(wgtc[i*3+1]>=darklight)
||(wgtc[i*3+2]>=darklight)) {
      elem[elemi]=i;  // consider the light brighter than 'darklight'
```

```
      elemi++;
      pr=255;
      pb=0;
    }
    camoutrgb[(CXSIZE+y*CXSIZE*2+x)*4+1]=pr;  // putting a mark for demo
    camoutrgb[(CXSIZE+y*CXSIZE*2+x)*4+2]=0;
    camoutrgb[(CXSIZE+y*CXSIZE*2+x)*4+3]=pb;
    camoutrgb[(CXSIZE+y*CXSIZE*2+x+1)*4+1]=pr;
    camoutrgb[(CXSIZE+y*CXSIZE*2+x+1)*4+2]=0;
    camoutrgb[(CXSIZE+y*CXSIZE*2+x+1)*4+3]=pb;
    camoutrgb[(CXSIZE+(y+1)*CXSIZE*2+x)*4+1]=pr;
    camoutrgb[(CXSIZE+(y+1)*CXSIZE*2+x)*4+2]=0;
    camoutrgb[(CXSIZE+(y+1)*CXSIZE*2+x)*4+3]=pb;
    camoutrgb[(CXSIZE+(y+1)*CXSIZE*2+x+1)*4+1]=pr;
    camoutrgb[(CXSIZE+(y+1)*CXSIZE*2+x+1)*4+2]=0;
    camoutrgb[(CXSIZE+(y+1)*CXSIZE*2+x+1)*4+3]=pb;
  }

  elemnum=elemi; // the number of the calculating lights
  free(p00);
  free(p01);
  free(p10);
  free(p11);

}
```

```
void superimpose(void *arg) // main loop
{
  int i,j;
  int thrid=(int)arg;

  if (thrid==0) {
    getcolor();
  }

  wait(thrid);

  // Linear Combination

  for (i=slp[thrid];i<slp[thrid+1];i++) {
    register int pici=(CYSIZE+i/xsize)*CXSIZE*2+(i%xsize);
    register int outi=(CYSIZE+i/xsize)*CXSIZE*2+CXSIZE+(i%xsize);
    if (mask[i]==PICTURE) { // when the pixel is 'background'
      camoutrgb[outi*4+1]=camoutrgb[pici*4+1];
      camoutrgb[outi*4+2]=camoutrgb[pici*4+2];
      camoutrgb[outi*4+3]=camoutrgb[pici*4+3];
    } else {
      if (mask[i]==HOUSE) { // when the pixel is 'virtual object'
register double sr,sg,sb;
sr=0;
```

```
sg=0;
sb=0;
for (j=0;j<elemnum;j++) {
  register int jx=elem[j]*3;
  sr += hsrgb[i][jx]*wgtc[jx];
  sg += hsrgb[i][jx+1]*wgtc[jx+1];
  sb += hsrgb[i][jx+2]*wgtc[jx+2];
}
sr=(int)(0.5+sr*rate);
sg=(int)(0.5+sg*rate);
sb=(int)(0.5+sb*rate);
if (sr>COLORMAX) {
  sr=COLORMAX;
}
if (sg>COLORMAX) {
  sg=COLORMAX;
}
if (sb>COLORMAX) {
  sb=COLORMAX;
}
camoutrgb[outi*4+1]=(int)sr;
camoutrgb[outi*4+2]=(int)sg;
camoutrgb[outi*4+3]=(int)sb;
      } else {
if (mask[i]==SHADE) { // when the pixel is 'shadow'
```

```
register double sr,sg,sb,ssr,ssg,ssb;
sr=0;
sg=0;
sb=0;
ssr=0;
ssg=0;
ssb=0;
for (j=0;j<elemnum;j++) {
  register int jx=elem[j]*3;
  sr += shdrgb[i][jx*2]*wgtc[jx];
  sg += shdrgb[i][jx*2+1]*wgtc[jx+1];
  sb += shdrgb[i][jx*2+2]*wgtc[jx+2];
  ssr += shdrgb[i][jx*2+3]*wgtc[jx];
  ssg += shdrgb[i][jx*2+4]*wgtc[jx+1];
  ssb += shdrgb[i][jx*2+5]*wgtc[jx+2];
}
if (sr==0.0) {
  camoutrgb[outi*4+1]=0;
} else {
  camoutrgb[outi*4+1]=camoutrgb[pici*4+1]*ssr/sr;
}
if (sg==0.0) {
  camoutrgb[outi*4+2]=0;
} else {
  camoutrgb[outi*4+2]=camoutrgb[pici*4+2]*ssg/sg;
```

```
  }
  if (sb==0.0) {
    camoutrgb[outi*4+3]=0;
  } else {
    camoutrgb[outi*4+3]=camoutrgb[pici*4+3]*ssb/sb;
  }
}
      }
    }
  }

  wait(thrid);

  if (thrid==0) {  // Output to X server
    XImage         *dximage;

    dximage=XCreateImage(d,xvi.visual,xvi.depth,ZPixmap,0,(char *)camoutrgb,
 CXSIZE*2,CYSIZE*2,32,0);
    if (!dximage) {
      fprintf(stderr,"can't create XImage\n");
      exit(0);
    }

    XPutImage(d,wd,gc,dximage,0,0,0,0,xsize*2,ysize*2);
    XFlush(d);
```

```
    fprintf(stdout,"Elapsed time = %lf sec\n",interval());
    // intercal() mesuring the time to superimpose
  }

  pthread_exit(0);
}

main(int argc,char **argv)
{
  int i,best,numxvi,xs,ys,amt,cpui;
  long msisize,cbs;
  Colormap       cm;
  XSetWindowAttributes   xswa;
  XVisualInfo *pxvi;

  pthread_t *thr,vthr;

  // arguments

  if (argc!=7) {
    fprintf(stderr,"Usage: %s <input.msi> <light-angles> <light-rate> <number-cpu> <skip-r
    exit(0);
  }

  input=(char *)malloc(SCHR*strlen(argv[1]));
```

```
light=(char *)malloc(SCHR*strlen(argv[2]));

strcpy(input,argv[1]);
strcpy(light,argv[2]);
rate=atof(argv[3]);
ncpu=atoi(argv[4]);
skip_rate=atoi(argv[5]);
darklight=atoi(argv[6]);

thr=(pthread_t *)malloc(sizeof(pthread_t)*ncpu);
state=(int *)malloc(sizeof(int)*ncpu);
slp=(int *)malloc(sizeof(int)*(ncpu+1));

// transform the angle to (x,y) on the fisheye image

setfep(light);

wgtc=(double *)malloc(sizeof(double)*pnts*3); // light colors of fisheye
elem=(int *)malloc(sizeof(int)*pnts); // the number of this

xsize=CXSIZE;  // background image width
ysize=CYSIZE;  // background image height

// read .msi file
```

```
inputf=fopen(input,"rb");
if (inputf==NULL) {
  fprintf(stderr,"Couldn't open %s\n",input);
  exit(0);
}
fread(&xs,sizeof(int),1,inputf);
fread(&ys,sizeof(int),1,inputf);
fread(&amt,sizeof(int),1,inputf);
fread(&msisize,sizeof(long),1,inputf);

if ((xsize!=xs)||(ysize!=ys)) {
  fprintf(stderr,"Not equal Xsize Ysize %d %d (videod) != %d %d (%s)\n",
  xsize,ysize,xs,ys,input);
  exit(0);
}
if (amt!=pnts) {
  fprintf(stderr,"Not equal the number of lights %d(%s)!=%d(%s)\n",
  pnts,light,amt,input);
  exit(0);
}
mask=(unsigned char *)malloc(sizeof(unsigned char)*xsize*ysize);
hsrgb=(unsigned char **)malloc(sizeof(unsigned char *)*xsize*ysize);
shdrgb=(unsigned char **)malloc(sizeof(unsigned char *)*xsize*ysize);
slp[0]=0;
cbs=0;
```

```
  cpui=1;
  for (i=0;i<xsize*ysize;i++) { // put into variables,
                                //  and distribute to each threads
    fread(&mask[i],sizeof(unsigned char),1,inputf);
    cbs++;
    if (mask[i]==HOUSE) {
      hsrgb[i]=(unsigned char *)malloc(sizeof(unsigned char)*pnts*3);
      fread(hsrgb[i],sizeof(unsigned char)*pnts*3,1,inputf);
      cbs += pnts*3;
    } else {
      if (mask[i]==SHADE) {
shdrgb[i]=(unsigned char *)malloc(sizeof(unsigned char)*pnts*6);
fread(shdrgb[i],sizeof(unsigned char)*pnts*6,1,inputf);
cbs += pnts*6;
      } else {
if (mask[i]!=PICTURE) {
  fprintf(stderr,"%d is not valid mask type\n",mask[i]);
  exit(0);
}
      }
    }
    if ((double)cbs>(double)msisize/(double)ncpu*(double)cpui) {
      slp[cpui]=i+1;
      cpui++;
    }
```

```
}


slp[ncpu]=xsize*ysize;
fclose(inputf);


// X Window Set up


d = XOpenDisplay(NULL);


xvi.class=TrueColor;  // Use 'TrueColor'
xvi.screen=0;
pxvi=XGetVisualInfo(d,VisualClassMask|VisualScreenMask,&xvi,&numxvi);
for (i=0, best = -1; i<numxvi; i++) {
  if (pxvi[i].depth == 24) {
    best = i;
  }
}
if (best==-1) {
  fprintf(stderr,"This display can't have TrueColor\n");
  exit(0);
}
xvi.visual=pxvi[best].visual;
xvi.depth=pxvi[best].depth;
xvi.colormap_size=pxvi[best].colormap_size;
xvi.screen=pxvi[best].screen;
```

```
cm=XCreateColormap(d,RootWindow(d,0),xvi.visual,AllocNone);

XFree((char *)pxvi);

xswa.background_pixel = WhitePixel(d,0);
xswa.border_pixel     = BlackPixel(d,0);
xswa.colormap         = cm;

wd=XCreateWindow(d,RootWindow(d,0),16,16,xsize*2,ysize*2,0,xvi.depth,
                 InputOutput,xvi.visual,
                 CWBackPixel|CWBorderPixel|CWColormap,&xswa);
gc=XCreateGC(d,wd,0,0);

XMapWindow(d,wd);
XFlush(d);

readableflag=0; // initialize
count=0;
oldcnt=0;
// thread for geting video frames
pthread_create(&vthr,NULL,(void *)&getvideo,(void *)NULL);

while (1) {
  for (i=0;i<ncpu;i++) {
    state[i]=0; // initialize
```

```
    }
    for (i=0;i<ncpu;i++) { // parallel processing
      pthread_create(&thr[i],NULL,(void *)&superimpose,(void *)i);
    }
    for (i=0;i<ncpu;i++) {
      pthread_join(thr[i],NULL);
    }
  }
  pthread_join(vthr,NULL);
  getchar();
}
```

References

[1] G.J. Ward and L.B. Lab: The RADIANCE Lighting Simulation and Rendering System, Comput.Gr.Proceedings, Annual Conference Series, 1994.

[2] R.Tsai: A versatile camera calibration tachnique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses, IEEE J. Robotics and Automation, pp. 323-344, 1987.

[3] I.Sato,Yoichi Sato, and K.Ikeuchi: Acquiring a Radiance Distribution to Superimpose Virtual Objecs onto a Real Scene, IEEE Trans. visualization and computer graphics, Vol. 5, No. 1, Jan-Mar 1999.

[4] B.K.P.Horn: Robot Vision, The MIT Press, Cambridge, MA, 1986.

[5] D.H. Ballard and C.M. Brown: Computer Vision, Englewood Cliffs, N.J.:Prentice Hall,1982.