Image Based BRDF Measurement

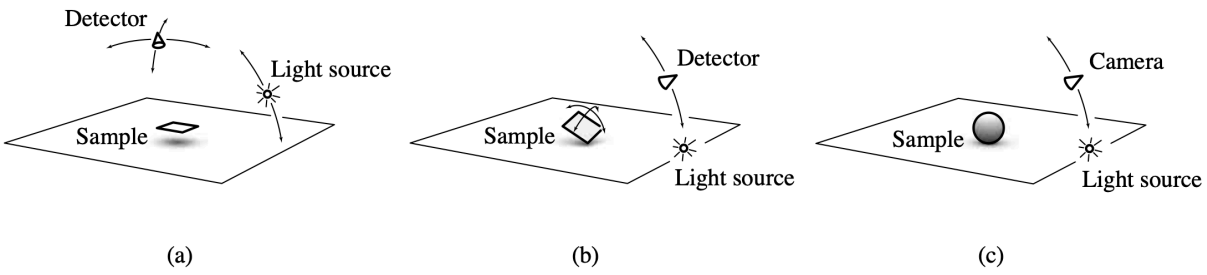
Chen Liang Qingyi He Zhenghao Xu

1. Background

The basic idea of our project originates from Image-Based BRDF Measurement Including Human Skin published in 1999. In this paper, a new image-based process for measuring the bidirectional reflectance of homogeneous surfaces rapidly, completely, and accurately is presented. This paper mainly eliminates two constraints on traditional BRDF measurement techniques.

Firstly, the constraint on geometry no longer exits. With disparity map or other 3D reconstruction techniques, the image-based measurement can measure single material of any geometry. And material of a sphere or other implicitly or explicitly represented geometry can be measured even without 3D reconstruction techniques. However, the traditional machine-based BRDF measurement can only measure material on a single point from different angles.

Secondly, the contraint on hardware disappears. Traditional BRDF measurement techniques depends on hardware called gonioreflectometer which is too delicate and hard to operate. Gonioreflectometer measures θ_i, ϕ_i, θ_e, ϕ_e one by one, consuming 3 to 4 hours to get the whole BRDF. However, the image-based technique "hides" two dimensions in the image. Only with rotations of light source and detectors each on one single dimension, whole BRDF can be measured within minutes. About isotropic material, we can just move the detector on a single dimension to measure the whole BRDF.



(a) (b) (c)

2. Difficulties Conquered

As you can see, the theory behind is rather simple. At first we did not realize when it comes to practice, it will be unbelievably exhausting. We did not find satisfying code on the internet, which means we had to build both our hardware and software from scratch. The main of difficulty of building software is how to calibrate and reconstruct 3D coordinates from 2D images precisely. And the main difficulty of building hardware is how to choose a darkroom, a point light source and isotropic sphere properly. Moreover, software and hardware must work together to make the error down to an acceptable range. In the following I gave a sketch of difficulties that we have conquered through the process.

- Hardware
    - An Ideal Point Light Source
    - An Ideal Darkroom with No Reflection
    - Ideal Isotropic Spheres with No Light Tansmitting
    - A Small Enough Chessboard for Calibration
    - A Camera with Little Distortion and Wide FOV
    - A Car Carrying the Camera to Move around A Circle Track
    - Place the Objects Properly
- Software
    - Calibrate Camera Track
    - Calibrate Light Source
    - Calibrate Sphere
    - Rendering
- Data Analysis
    - Local Polynomial Fitting
    - Physical Representation of BRDF

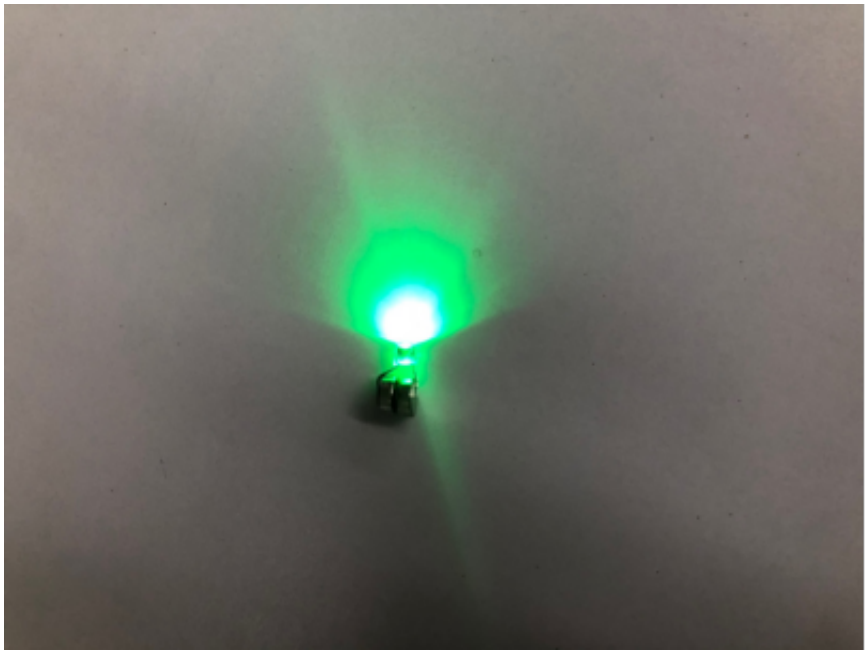
3. Hardware Setup

- Darkroom

  For measuring the BRDF, the ideal situation is which exactly eliminates the influence of ambient light. Although it is impossible to reach the perfect, we chose to get a black paper box and painted it with black ink. To provide enough space for camera to move, this box is sized 50cm*50cm*50cm(it proved that it is still too small to operate).

- Point Light Source
  - LED Light Bulb

    It is really a good point light source, unless it works with PCB board and a good battery and provides enough light. Since we didn't think about designing a board to control such a *simple* equipment, we finally found out how difficult it is to use a LED bulb with a bare button battery.



- Bicycle Front Light

  It is really a good choice(although my teammates doesn't think so), at least it is bright enough and easy to turn on/off though the strength seems to be fixed and is too much larger than a single point. We abandoned it due to its large size.

- iPhone Flashlight

  It is fairly a good choice since it looks like a point and will never run out of power. The four light levels also allows us to reach a better video quality.

- Camera
  - SLR with Shutter Line

    At the very beginning, we thought it would be a good idea to use it because we can control it without opening the box. However, it is just too big.

- iPhone with RCam/Manual Cam

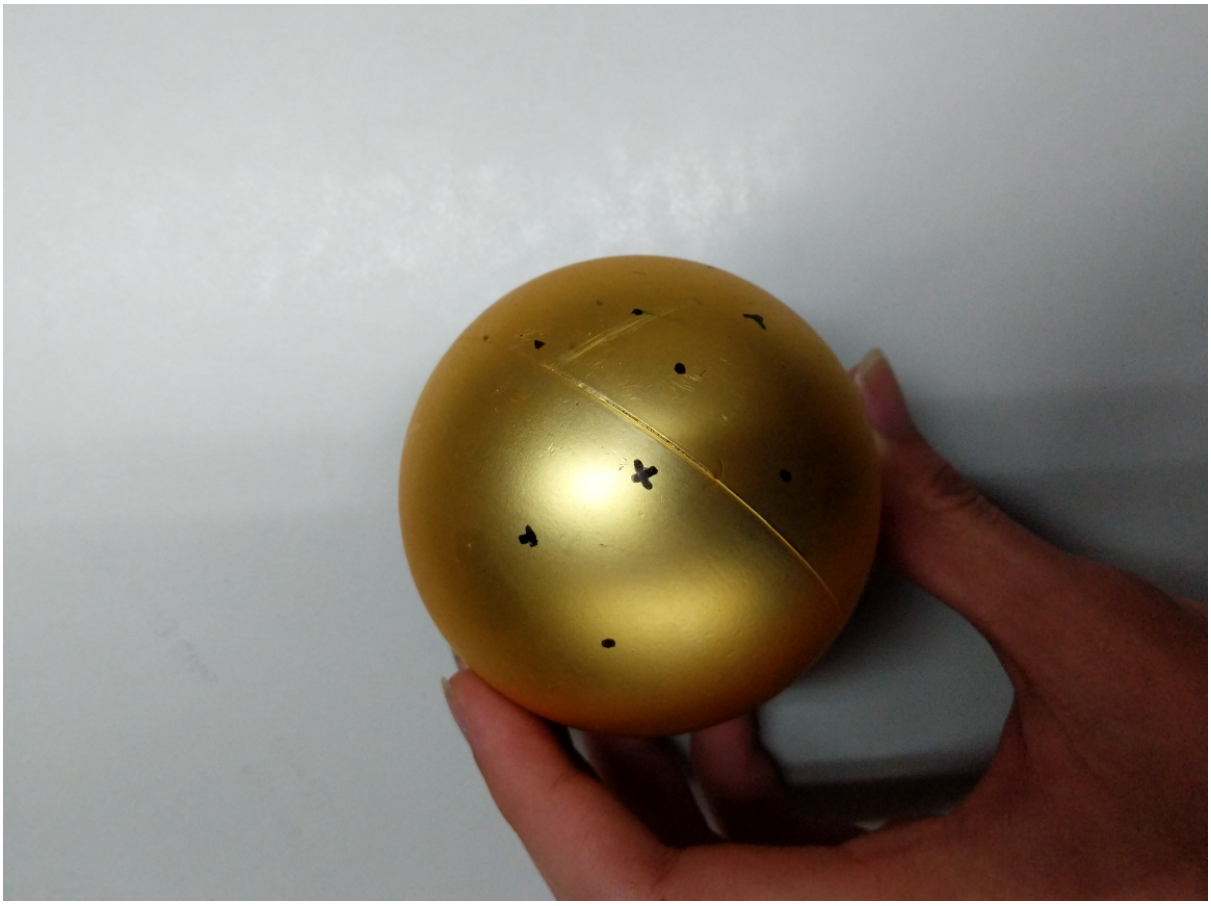
  It seems ridiculous to do this precise task with such a daily equipment, but those apps really gave us a chance to capture information we need in this confined space while the result seems fairly good. The app also provides all the parameters like exposure and focus which helps us to check our computing results.

- Carrier

  The orbit of car is almost a circle, while it cost not too much compared with the circular sliders.

- Sphere

We use the Christmas ball which is seen as isotropic. And we manually marked lots of features on it.

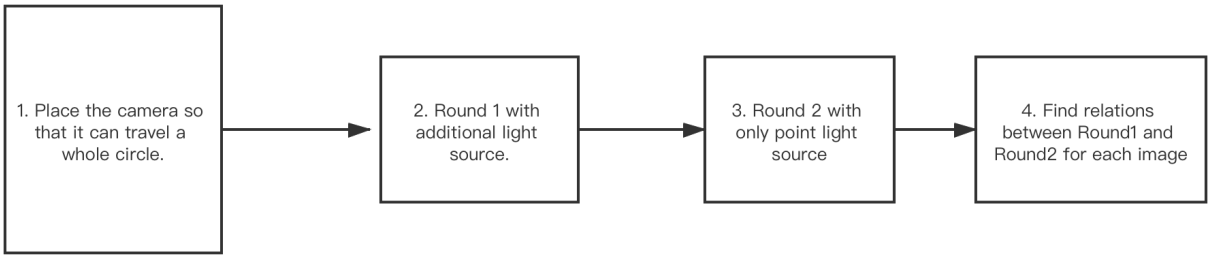


- Fix the Object

We used a naive but useful way to get the center of the box, used needle and string to tie the Christmas ball on it. Black sponge is attached to it to avoid it swinging and absorb the shock.

4. Hardware Pipeline

With proper hardware built, the hardware pipeline is easy to run following the below diagram:



To make the chessboard and features on the sphere easier to be found for the calibration process, we do 2 round. Round 1 is with additional light source and Round 2 is only with the point light source. We can see Round 1 as calibration round and Round 2 as measurement round. We have to find the relation between the two rounds for each image manually. We can use the relative position of the sphere, chessboard and the poingt light source to make the relation-finding process easier.

Here is an example:

Image260 in Round2(feature is easy to be found):

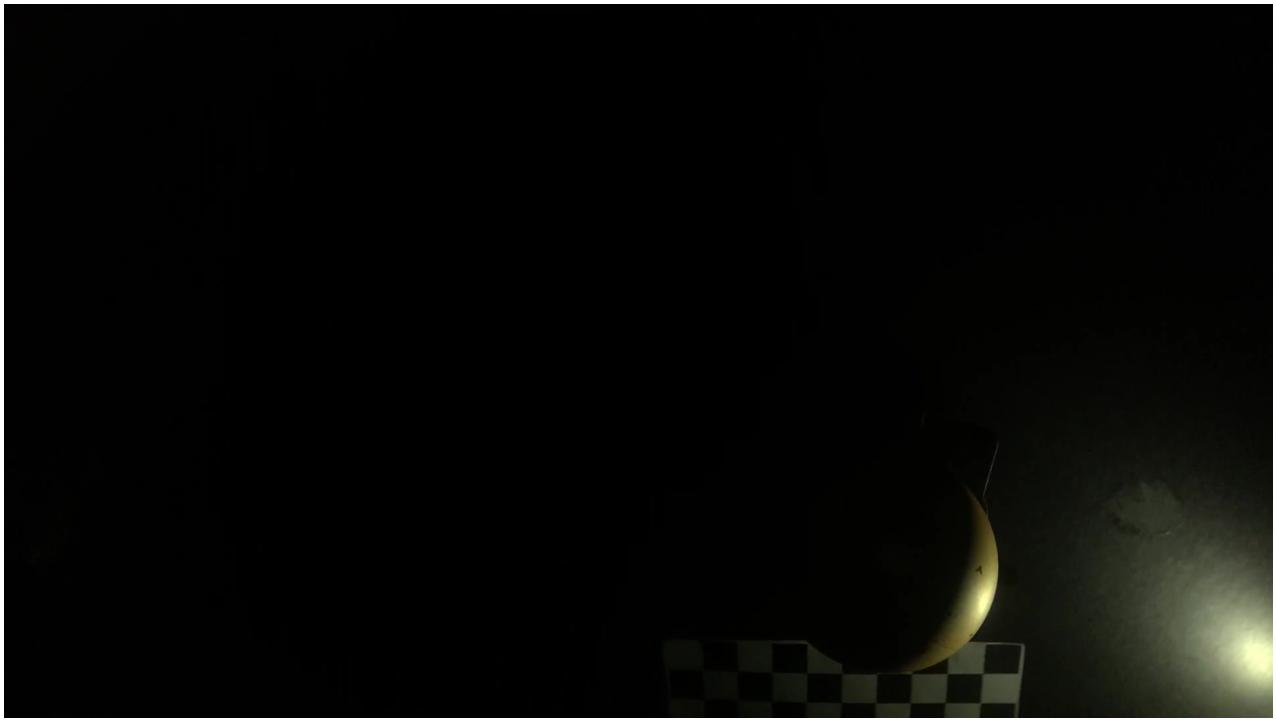


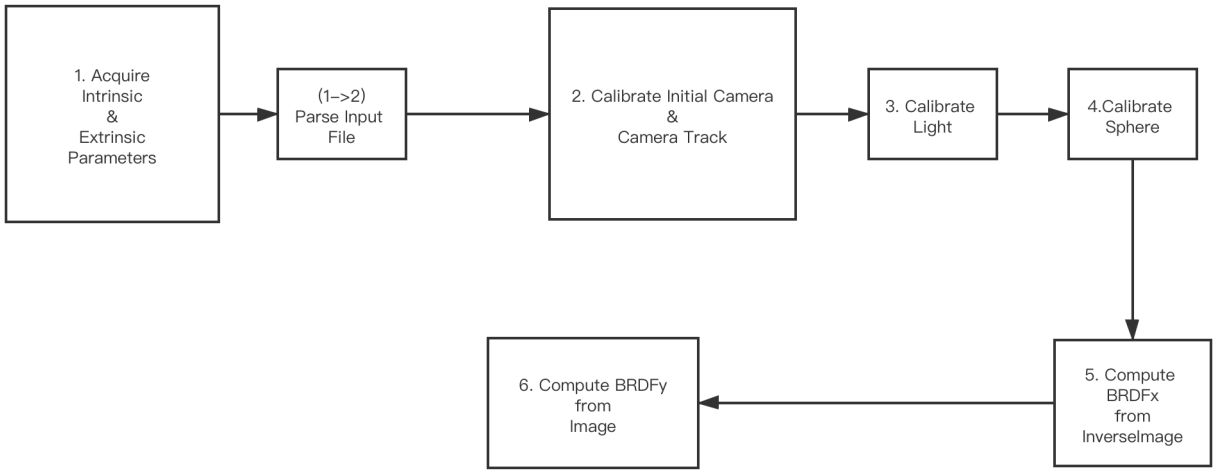
Image223 in Round1(chessboard is easy to be found):

We assert that the two images above were taken in the same position according to the relative postion between the sphere and the chessboard.

Keep in mind that in Round1 and Round2, the intrinsic parameters and the three axises of the camera(direction, up and horizontal) are not allowed to be change.

5. Software Pipeline

The software Pipeline is shown below:



The parsed input file is written in the following format:

```
//pic_num,width,height
200 192 108
//exposure_time
0.125
//focal_length,cx,cy
134 96 96
//r_and_t_of_initial_camera
2.2427522580004164e-01 2.5908934894269053e-01 -1.1371962271610156e+00
-2.7168094753128395e+02 6.0357124988317878e+01 1.2121804657508103e+03
//light_color
1 1 1
//t1_and_t2_of_cameras_to_calibrate_track
-2.7581488738424542e+02 1.3108436630791925e+02 1.2379757529355043e+03
-2.7018959265217359e+02 2.0221008236008169e+02 1.2701167905045952e+03
//pixel_coordinates_of_light
46 74 56 102
//cx,cy
96 96 96 96
//derivations_from_initial_photo
-15 -10
//pixel_coordinates_of_sphere
148 86 146 96 148 85 147 86
143 80 151 85 142 79 142 80
//cx,cy
96 96 96 96
//derivations_from_initial_photo
48 72
```

With the information of the input file, BRDF can be calculated. BRDF is stored as:

```
map<Vec4f,Vec3f> BRDF;
```

Vec4f is $(\theta_i, \phi_i, \theta_e, \phi_e)$, and Vec3f is RGB.

The first step of the software pipeline is to acquire intrinsic parameters of the camera with the compiled calibration software used in Assignment 1. In this step, we can use as many photos as possible to get parameters (f_x, f_y, c_x, c_y) precise enough. Keep in mind that the quadruple here is under pixel coordinate. Code below is to convert it to floating point numbers within (0,1) and compute FOV of the camera:

```
_cx/=width;
_cy/=width;
_cx-=0.5;
_cy-=0.5;
fovx=2*atan(width/2/focus_length_x);
fovy=2*atan(height/2/focus_length_y);
```

The second step is to acquire 3 groups of extrinsic parameters to calibrate the camera track. As long as we know the coordinates of three different camera positions, which is the opposite of the translation vectors, the equatino of the camera track can be computed:

```
void cal_axisAndTrackC(Vec3f ex1,Vec3f ex2,int dev)
    {
      Vec3f pos1=ex1,pos2=ex2;
      Vec3f a=pos1-camera_pos,b=pos2-camera_pos;
      Vec3f::Cross3(axis,a,b);
      axis.Normalize();
      Vec3f d1,d2;
      Vec3f::Cross3(d1,a,axis);
      Vec3f::Cross3(d2,b,axis);
      d1.Normalize();
      d2.Normalize();
      Vec3f o1=(1.0f/2)*(pos1+camera_pos),o2=(1.0f/2)*(pos2+camera_pos);
      Vec3f o=o1-o2;
      float t=(o.x()*d2.y()-o.y()*d2.x())/(d2.x()*d1.y()-d1.x()*d2.y()),tu=
(o.z()*d2.y()-o.y()*d2.z())/(d2.z()*d1.y()-d1.z()*d2.y());
      trackC=o1+t*d1;
      Vec3f trackRVec=trackC-camera_pos;
      trackR=trackRVec.Length();
      Vec3f dist=camera_pos-ex1;
      velocity=dist.Length()/exposure_time/dev1;
      cout<<trackR<<endl<<trackC;
    }
```

As shown in the above code, the center of the track is trackC, and the radius of the track is trackR. And the axis is exactly the normal vector of the track plane. The center of the track is on the axis. Knowing the axis and the exposure time, the position of the camera on each image can be computed.

In the above two steps, we can also calculate the three axises of the camera: direction, up and horizontal following the below relation：

$$R_{view} = \begin{bmatrix} x_{\hat{g}\times\hat{t}} & y_{\hat{g}\times\hat{t}} & z_{\hat{g}\times\hat{t}} & 0 \\ x_t & y_t & z_t & 0 \\ x_{-g} & y_{-g} & z_{-g} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$s\,m' = A[R|t]M'$$

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r11 & r12 & r13 & t1 \\ r21 & r22 & r23 & t2 \\ r31 & r32 & r33 & t3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

```
float alpha=Rot.Length();
Rot.Normalize();
Matrix rotate=Matrix::MakeAxisRotation(Rot,alpha);
camera_dir.Set(-rotate.Get(2,0),-rotate.Get(2,1),-rotate.Get(2,2));
camera_up.Set(-rotate.Get(1,0),-rotate.Get(1,1),-rotate.Get(1,2));
camera_pos.Set(-T.x(),-T.y(),-T.z());
camera_dir.Normalize();
camera_up.Normalize();
Vec3f::Cross3(horizontal,camera_dir,camera_up);
```

The third step and the fourth step is to calibrate light source and the sphere. The basic principle of these two steps are the same: Triangulation with Two Cameras. Due to limitaion on space, we place one camera on two positions to substitute the original two-camera-technique. We see light source a single feature under pixel coordinate. And we pre-marked lots of features on the sphere to extract 4 features under piexel coordinate to fix the equation of the sphere. I wrote a function cal_dot_pos(), taking the corresponding feature under two pixel coordinates, and the rotation angles of the pixel coordinates compared to the initial pixel coordinate, to finally caculate the 3D coordinate of the feature.

```
Vec3f cal_dot_pos(int x1,int y1,int x2,int y2,float alpha1,float alpha2)
{
  float pix_x1=-1.0f*(x1+width/2.0f)/width,
        pix_y1=1.0f*(y1+width/2.0f)/width,
        pix_x2=-1.0f*(x2+width/2.0f)/width,
        pix_y2=1.0f*(y2+width/2.0f)/width,ratio=tan(fovx/2)/tan(fovy/2);
  Vec3f orig1=Rotate(camera_pos,alpha1),
        orig2=Rotate(camera_pos,alpha2);
  Vec3f camera_dir1=RotateDir(camera_dir,alpha1),
        camera_dir2=RotateDir(camera_dir,alpha2);
  Vec3f camera_up1=RotateDir(camera_up,alpha1),
        camera_up2=RotateDir(camera_up,alpha2);
  Vec3f horizontal1=RotateDir(horizontal,alpha1),
        horizontal2=RotateDir(horizontal,alpha2);
        camera_dir1.Normalize();
        camera_dir2.Normalize();
        camera_up1.Normalize();
        horizontal1.Normalize();
        camera_up2.Normalize();
        horizontal2.Normalize();
  Vec3f fc1=orig1+camera_dir1*(1.0f/(2*tan(fovx/2))),
        fc2=orig2+camera_dir2*(1.0f/(2*tan(fovx/2)));
  Vec3f c1=fc1+(pix_x1-0.5)*horizontal1+(pix_y1-0.5*ratio)*camera_up1,
        c2=fc2+(pix_x2-0.5)*horizontal2+(pix_y2-0.5*ratio)*camera_up2;
  Vec3f rdir1=c1-orig1,rdir2=c2-orig2;
        rdir1.Normalize();
        rdir2.Normalize();
  Vec3f o=orig1-orig2;
  float t1=(o.x()*rdir2.y()-o.y()*rdir2.x())/(rdir2.x()*rdir1.y()-
rdir1.x()*rdir2.y()),
        t2=(o.y()*rdir2.z()-o.z()*rdir2.y())/(rdir2.y()*rdir1.z()-
rdir1.y()*rdir2.z()),
        t3=(o.x()*rdir2.z()-o.z()*rdir2.x())/(rdir2.x()*rdir1.z()-
rdir1.x()*rdir2.z());
  float t=(t1+t2+t3)/3.0f;
      cout<<"t1:"<<t1<<endl;
      cout<<"t2:"<<t2<<endl;
      cout<<"t3:"<<t3<<endl;
      Vec3f dot=orig1+t*rdir1;
```

```
        return dot;
}
```

You can see that I calculate three parameter "t" in the above code to minimize the error. It can also be used to "debug" the hardware. Parameters t1, t2 and t3 corresponds to projected intersection points on XY, YZ and ZX plane. When the three "t" are close enough, the error caused by the hardware and the measurement of feature pixels is small enough.

The fifth step is with all the information above, the corresponding $BRDF_\chi(\theta_i, \phi_i, \theta_e, \phi_e)$ can be computed. I realized that this process is very similar to the process of rendering using ray casting: we shoot a ray, intersect it with an object, and return information about the intersection point. In the rendering process, the "information about the intersection point" is only RGB. So I created a subclass of Image called InverseImage to calculate and store additional information, which are *"isObject"* showing whether it is an valid pixel containing the sphere, and $BRDF_\chi(\theta_i, \phi_i, \theta_e, \phi_e)$.

```
class InverseImage:public Image{
  public:
    InverseImage(int w,int h):Image(w,h)
    {
      isObject=new bool[w*h];
      BRDF_x=new Vec4f[w*h];
    }
    void SetBool(int x,int y,Hit& h,Ray& r,Light* l)
    {
      if(h.getT()!=inf){
         isObject[y*width+x]=true;
         Vec3f camera_dir=r.getDirection(),
               intersectionPoint=h.getIntersectionPoint();
         Vec3f light_dir,col;
         float dist_to_light;
         l->getIllumination(intersectionPoint,light_dir,col,dist_to_light);
         light_dir*=-1.0f;
         Vec3f xAxis(1,0,0),yAxis(0,1,0),zAxis(0,0,1);
         float camera_phi=camera_dir.Dot3(zAxis)/camera_dir.Length(),
               light_phi=light_dir.Dot3(zAxis)/light_dir.Length();
               camera_dir.Set(camera_dir.x(),camera_dir.y(),0);
               light_dir.Set(light_dir.x(),light_dir.y(),0);
         float camera_theta=camera_dir.Dot3(xAxis)/camera_dir.Length(),
               light_theta=light_dir.Dot3(xAxis)/light_dir.Length();
         BRDF_x[y*width+x].Set(light_theta,light_phi,camera_theta,camera_phi);
      }
    }
    ~InverseImage(){
      if(isObject) delete[] isObject;
      if(BRDF_x) delete[] BRDF_x;
    }
  private:
    bool* isObject;
```

```
    Vec4f* BRDF_x;
};
```

The last step is to extract RGB on taken images as $BRDF_y$. With corresponding $BRDF_x$ recorded in the InverseImage. A scatter diagram of BRDF is acquired.

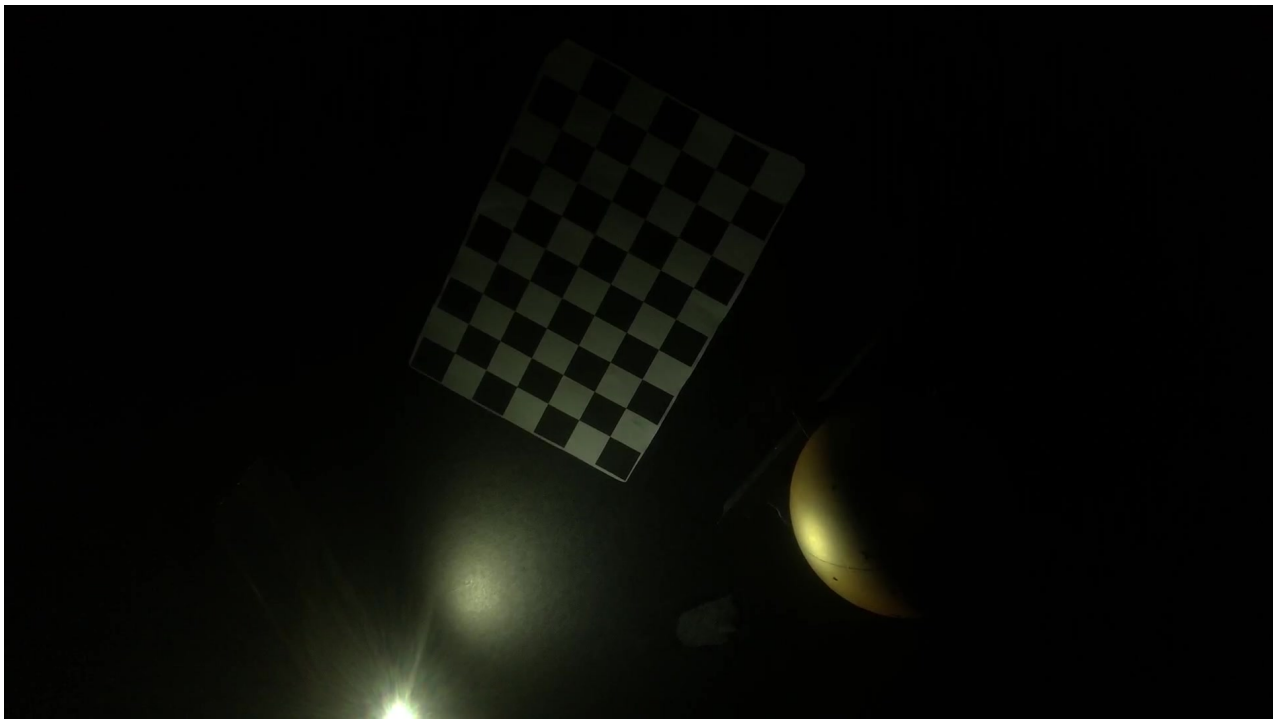
```
for(int i=0;i<pic_num;i++){
   cv::Mat img=cv::imread(img_name[i]);
   for(int y=0;y<img.rows;y++)
   for(int x=0;x<img.cols;x++)
   if(pics[i].IsObject(x,y)){
    float r=img.at<cv::Vec3b>(x,y)[2]/255.0,
          g=img.at<cv::Vec3b>(x,y)[1]/255.0,
          b=img.at<cv::Vec3b>(x,y)[0]/255.0;
    Vec4f BRDF_x=pics[i].getBRDFX(x,y);
    if(BRDF.find(BRDF_x)==BRDF.end()){
      Vec3f BRDF_y(r,g,b);
      BRDF.insert(pair<Vec4f,Vec3f>(BRDF_x,BRDF_y));
    }
   }
}
```

6. Data Analysis

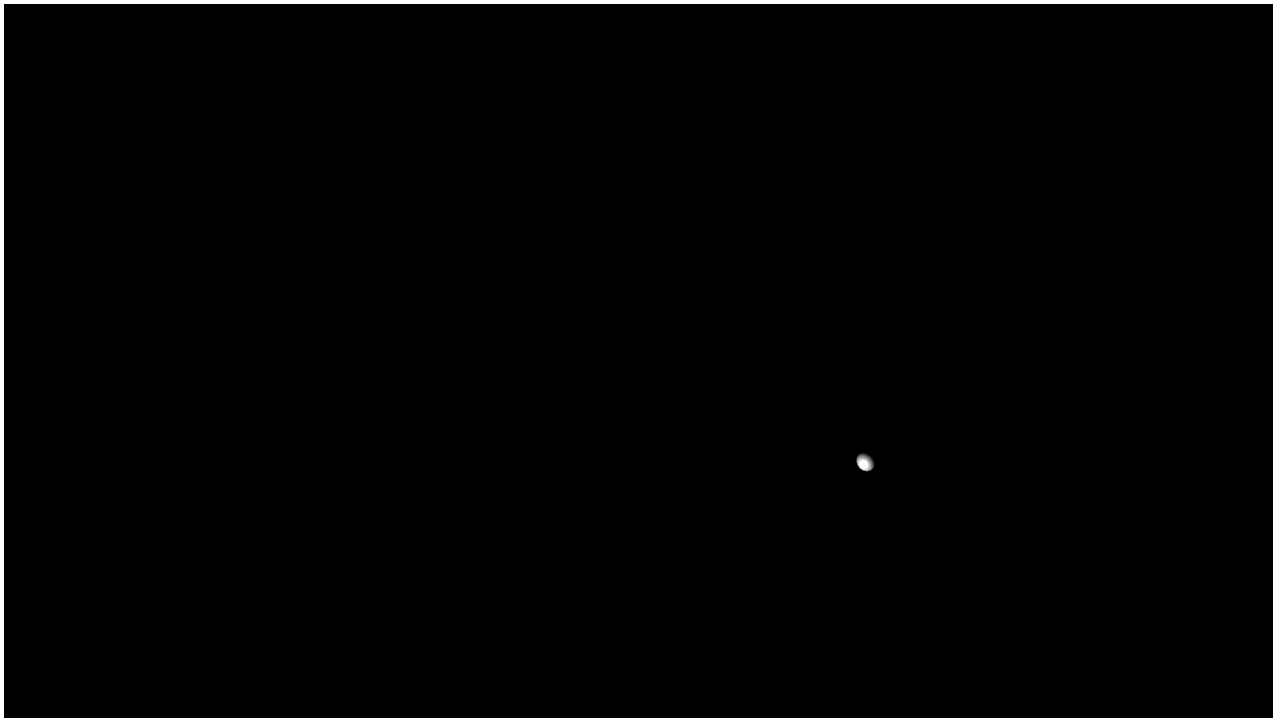
As I realized that the process of calculate $BRDF_X$ is very similar to rendering. I pre-rendering a group of image to see the error. It turned out that the error is relative large.

The initial image is IMAGE_175 of Round 2 and the corresponding pre-rendered image is 0.tag:

IMAGE_175 of Round 2:



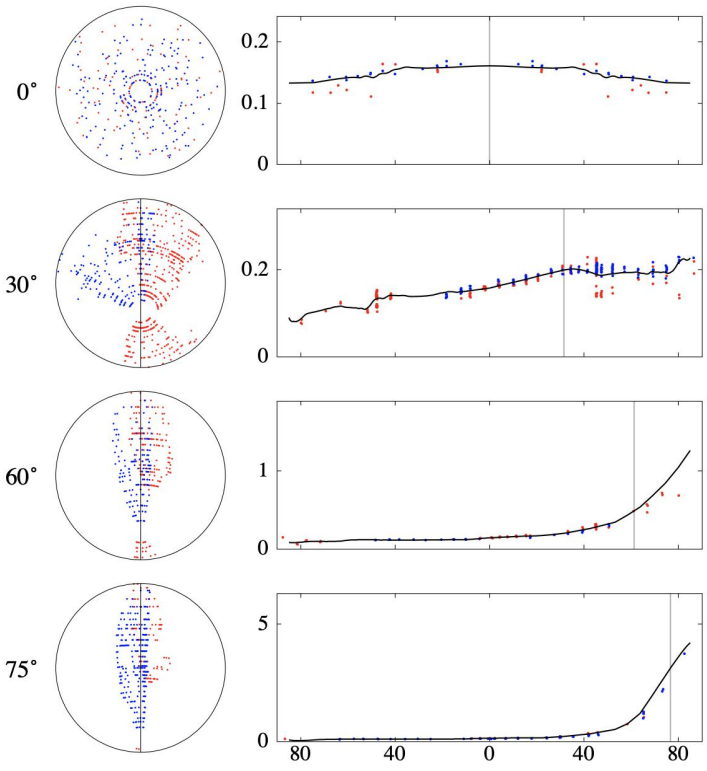
Pre-rendered Image(symmetric with 180 degree rotation):

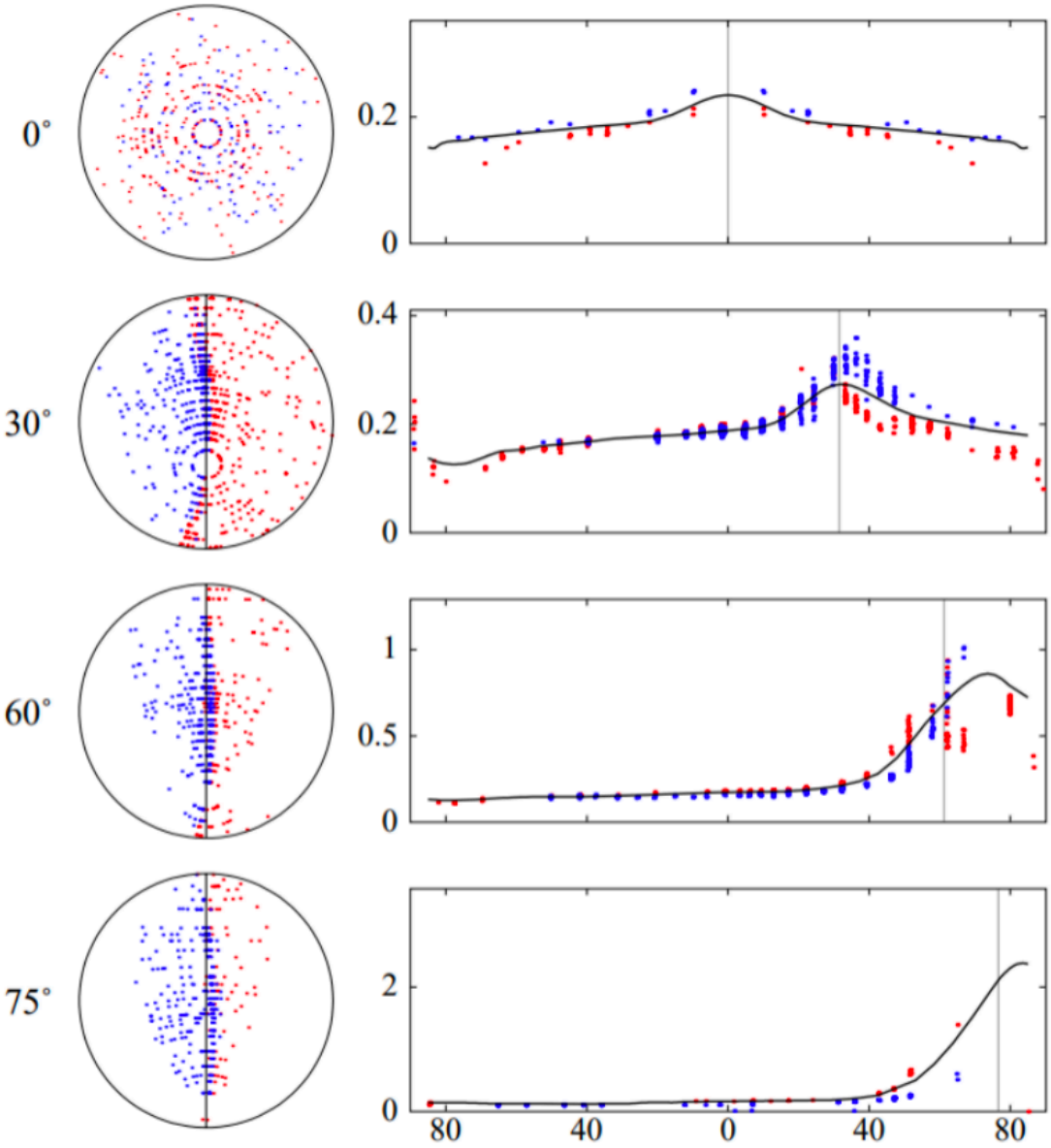


Fortunately, we can discover the relative postion of camera, light source, and sphere is rather accurately with small error. How do we judge it? We use photoshop to find that derivation of the center of the spheres in two photos is just (52,34) pixels. (52/1920,34/1080)=(0.03,0.03), which is quite small. However, the error of the radius of the sphere is quite large, up to 453 pixels, where the error is 453/1920=0.24. The huge error on the sphere originates from the coordinates of 4 feature pixels. In the relative dark or over-exposure evironment, it is extremely hard to find the features manually.

Another way to ensure that small enough relative position error is by analyzing the specular light. We can see the direction and the area of the specular light is very similar in the two pictures.

We have no time for fitting, here, we attach the BRDF gained by local polynomial fitting in the original paper:





Our scatter point graph is in another file called final_data.txt.

7. Thoughts

This is a really struggle but fruitful process!!! We can't believe that finally we make it!!! I still remember on the presentation day, standing in front of the class, admitting that we failed. However, what the professor and the TA said greatly inspired us!!! At the end, it turnd out that as said by professor Wu, we failed because the lack of time. （老师的鼓励和助教的和蔼极大地激励了我们，谢谢你们！！！）Given enough time, we really got some meaningful data! Thanks a lot for these course! Building something from scratch is exactly painful but coooooool!!!!!!