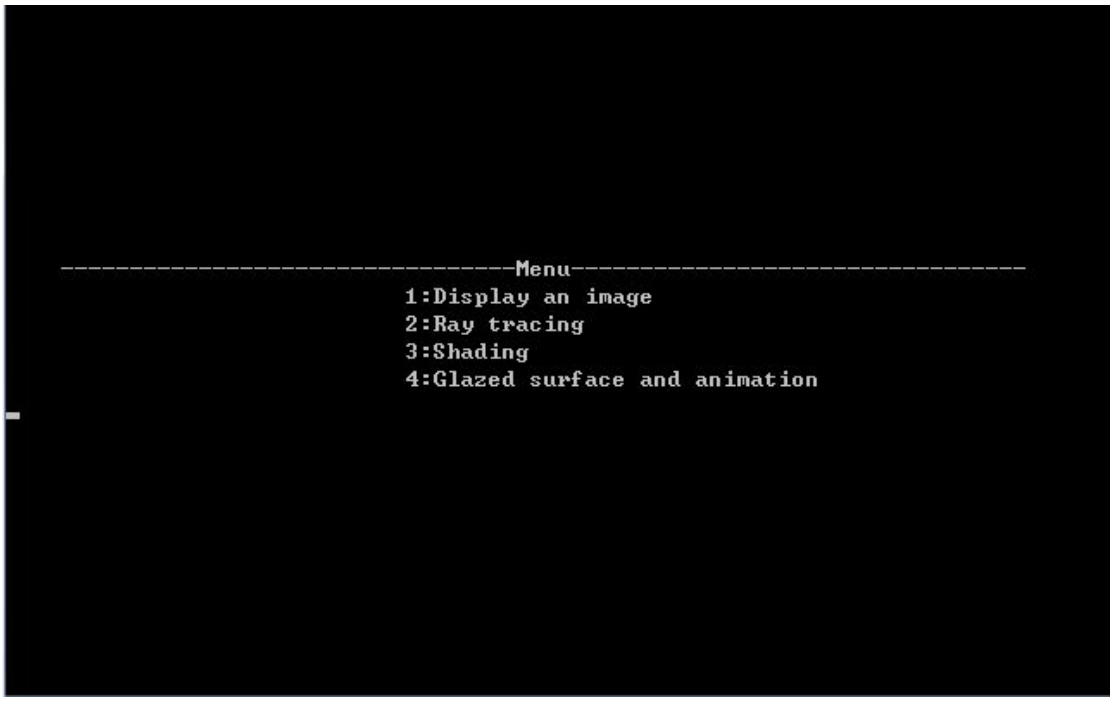
CAP 5705 - Homework 1
First-Hit Ray Tracer

I. Introduction

The project renders a scene composed of two spheres and a tetrahedron, adding shading, shadow, reflection and a simple animation. And each step of the project can be realized separately, by choosing from the main menu. The main menu is as below(Figure 1-1).

The environment of the homework is Windows7+Visual-Studio2010+Glut, the code and the makefile are included in the zip file. To make sure they can work well, I have tried to run it on computers in the lab.



```
---------------------------------Menu---------------------------------
                         1:Display an image
                         2:Ray tracing
                         3:Shading
                         4:Glazed surface and animation
```

Figure 1-1

II. Display an Image

It's the first step to be familiar the whole environment of OpenGL, Glut, and then get known to basic functions. Especially **glDrawPixels (width, height, format, type, data)** -- write a block of pixels to the frame buffer.

The project realizes it in display1(). The window size is 500*500, the colors of the pixel is decided by the position, they are separated from the middle of the figure. The code(Code 2-1) and the figure(Figure 2-1) are listed below.

```
void myDisplay1(void)
{ /*Just display an image*/
	int x,y;
	for (x=0;x<500;x++)
		for(y=0;y<500;y++)
		{
			if(x<=250&&y<=250)
			{
				image[y][x][0] =0.3;
				image[y][x][1] =0.3;
				image[y][x][2] =0;
			}
			else if(x<=250&&y>=250)
			{
				image[y][x][0] =0;
				image[y][x][1] =0.3;
				image[y][x][2] =0.3;
			}
			else if(x>=250&&y<=250)
			{
				image[y][x][0] =0.3;
				image[y][x][1] =0;
				image[y][x][2] =0.3;
			}
			else
			{
				image[y][x][0] =0;
				image[y][x][1] =0;
				image[y][x][2] =0;
			}
		}
	glClearColor(0,0,1,0);
	glClear(GL_COLOR_BUFFER_BIT);
	glDrawPixels(width,height,GL_RGB,GL_FLOAT,image);
	glutSwapBuffers();
}
```

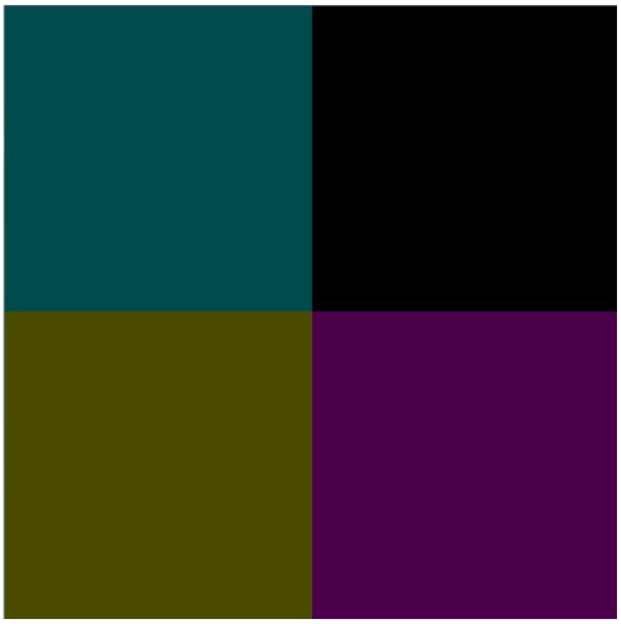


Figure 2-1

Code 2-1

III. Ray Tracing

Rendering is a process that takes as its input a set of objects and produces as its output an array of pixels, and ray tracing is an image-order algorithm for making renderings of 3D scenes. The ray tracing of the spheres and the tetrahedron are different. The project realizes them separately. However, they use the same LookAt and LightDir vectors. **LookAt=getUnitVector(0,0.3,0); LightDir=getUnitVector(-0.5,-0.5,0.1)**

1. Ray-Sphere Intersection

Given a ray $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$ and an implicit surface $f(\mathbf{p}) = 0$, I use $f(\mathbf{p}(t)) = 0$ to get the intersection point. Then cames out the following equation(Equation 3-1).

$$t = \frac{-\mathbf{d}\cdot(\mathbf{e}-\mathbf{c}) \pm \sqrt{(\mathbf{d}\cdot(\mathbf{e}-\mathbf{c}))^2 - (\mathbf{d}\cdot\mathbf{d})\left((\mathbf{e}-\mathbf{c})\cdot(\mathbf{e}-\mathbf{c}) - R^2\right)}}{(\mathbf{d}\cdot\mathbf{d})}.$$

Equation 3-1

In this equation, **d** stands for view direction, **e** is the position of the pixel,and **c** is the position of the sphere center. And the project realizes ray-sphere intersection in **Vector GetInterPoint_Sphere(Vector pixel , Vector dir , Sphere s)**. The code(Code 3-1) is listed below.

```
Vector GetInterPoint_Sphere(Vector pixel,Vector dir,Sphere s)
{
        ......//parameters initializing
        temp1=multiply(dir,subtract(pixel,s.O));//d*(e-c)
        temp1=temp1*temp1;
        temp2=multiply(dir,dir)*((multiply(subtract(pixel,s.O),subtract(pixel,s.O))-s.R*s.R));//(d`d)((
e-c)(e-c)-R2)
        if((temp1-temp2)<0)
                IntSphere=false;
        else
        {
                temp1=sqrt(temp1-temp2);
                temp2=-(multiply(dir,subtract(pixel,s.O))); //-d(e-c)
                t1=(temp2+temp1)/multiply(dir,dir);
                t2=(temp2-temp1)/multiply(dir,dir); //equation3-1
                t=fabs(t1)>fabs(t2)?t2:t1;
                result.x=pixel.x+t*dir.x;//e+td
                result.y=pixel.y+t*dir.y;
                result.z=pixel.z+t*dir.z;
        }
        return result;
}
```

Code 3-1

2. Ray-Tetrahedron Intersection

A tetrahedron is made up of 4 triangles, so the problem can be divided into 4 separate ray-triangle intersection problems.

The problem of ray-triangle intersection point can be solved by 3 steps:

- Intersection point is on the ray: R(t) = p + t d
- Intersection point is on a plane: (x - a) . n = 0

• Intersection point is inside all three edges

The equation(Equation 3-2) comes from the first 2 steps:

$$(\boldsymbol{p}+t\boldsymbol{d}-\boldsymbol{a})\cdot\boldsymbol{n}=0$$

$$t=\frac{(\boldsymbol{a}-\boldsymbol{p})\cdot\boldsymbol{n}}{\boldsymbol{d}\cdot\boldsymbol{n}}$$

Equation 3-2

In this equation, **d** stands for view direction, **p** is the position of the pixel,and **a** is the position of vertex a. The project realizes ray-triangle intersection in **Vector getNormal_Triangle (int Triangle_num).**

The 3rd step can be solved by Equation 3-3:

$$((\boldsymbol{b}-\boldsymbol{a})\times(\boldsymbol{x}-\boldsymbol{a}))\cdot\boldsymbol{n}>0$$
$$((\boldsymbol{c}-\boldsymbol{b})\times(\boldsymbol{x}-\boldsymbol{b}))\cdot\boldsymbol{n}>0$$
$$((\boldsymbol{a}-\boldsymbol{c})\times(\boldsymbol{x}-\boldsymbol{c}))\cdot\boldsymbol{n}>0$$

Equation 3-3

a,b,c stand for the 3 vertexes In this equation. And the project realizes ray-triangle intersection in **Vector GetInterPoint_Triangle(Vector pixel,Vector dir,Triangle t).**

The ray-tetrahedron intersection is realized in **Vector GetInterPoint_Tetrahedron (Vector pixel,Vector dir,Tetrahedron t)**. It does ray tracing to each triangle in order, and finally decide which point the pixel traces to. The code(Code 3-2) and the figure (Figure 3-1) are listed below.

```
Vector GetInterPoint_Triangle(Vector pixel,Vector dir,Triangle t)
{
    ......//parameters initializing
    t1=multiply(subtract(t.ver1,pixel),t.normal)/multiply(dir,t.normal); //equation 3-2
    Vector ab=subtract(t.ver2,t.ver1);
    Vector bc=subtract(t.ver3,t.ver2);
    Vector ca=subtract(t.ver1,t.ver3);
    temp.x=pixel.x+t1*dir.x; //p+td
    temp.y=pixel.y+t1*dir.y;
    temp.z=pixel.z+t1*dir.z;
    if(multiply(GetCrossproduct(ab,subtract(temp,t.ver1)),t.normal)>0)
        if(multiply(GetCrossproduct(bc,subtract(temp,t.ver2)),t.normal)>0)
            if(multiply(GetCrossproduct(ca,subtract(temp,t.ver3)),t.normal)>0) //equation 3-3
            {
                IntTriangle=true;
                result=temp;
            }
    return result;
}
```

Vector GetInterPoint_Tetrahedron(Vector pixel,Vector dir,Tetrahedron t)

```
{
        ......//parameters initializing
        result_temp=GetInterPoint_Triangle(pixel,dir,t.t1);
        if(IntTriangle==true)
        {       result=result_temp;
                distance_temp=getDistance(pixel,result);
                distance=distance_temp;
                IntTriangle_No=1;         }
        result_temp=GetInterPoint_Triangle(pixel,dir,t.t2);
        if(IntTriangle==true)
        {
                distance_temp=getDistance(pixel,result_temp);
                if(distance_temp<distance)
                {       result=result_temp;
                        distance=distance_temp;
                        IntTriangle_No=2;     }
        }
        result_temp=GetInterPoint_Triangle(pixel,dir,t.t3);
        if(IntTriangle==true)
        {
                distance_temp=getDistance(pixel,result_temp);
                if(distance_temp<distance)
                {       result=result_temp;
                        distance=distance_temp;
                        IntTriangle_No=3;     }
        }
        result_temp=GetInterPoint_Triangle(pixel,dir,t.t4);
        if(IntTriangle==true)
        {
                distance_temp=getDistance(pixel,result_temp);
                if(distance_temp<distance)
                {
                        result=result_temp;
                        distance=distance_temp;
                        IntTriangle_No=4;
                }
        }
        if(IntTriangle_No==0)
                IntTetrahedron=false;
        return result;
}
```

Code 3-2

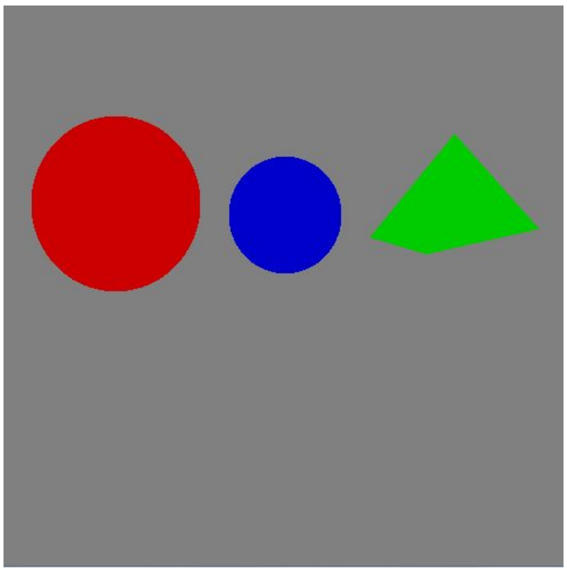


Figure 3-1

IV. Shading

Once the visible surface for a pixel is known, the pixel value is computed by evaluating a shading model. The project uses the ambient, diffuse and specular shading.

Together with the Blinn-Phong model, ambient shading completes the full version of a simple and useful shading model, and the equation(Equation 4-1) is:

$$L = k_a I_a + k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^n.$$

(Equation 4-1)

where L is the pixel color; ka is the surface's ambient coefficient, or "ambient color" , kd is the diffuse coefficient, or the surface color, and ks is the specular coefficient, or the specular color of the surface; Ia is the ambient light intensity, and I is the intensity of the light source. As for the **n**, **h** and **l**, they are shown in the following figures(Figure 4-1 and 4-2), Geometries for Lambertian shading and Blinn-Phong shading. Worth mentioning: $\mathbf{h} = \frac{\mathbf{v}+\mathbf{l}}{\|\mathbf{v}+\mathbf{l}\|}$ here.

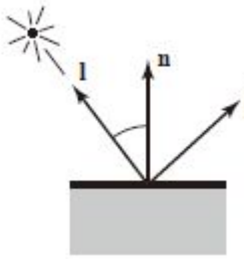


Figure 4-1

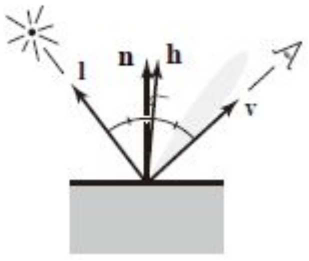


Figure 4-2

The project realizes it in **Color get_pixel_color(Vector ligdir,Vector viewdir,Vector normal,Light I),** The code(Code 4-1) and the figure (Figure 4-3) are listed below.

```
Color get_pixel_color(Vector ligdir,Vector viewdir,Vector normal,Light I)
{
        .....//parameters initializing
        h=getUnit(Add(viewdir,ligdir));
        mul_nl=multiply(ligdir,normal);
        mul_nh=multiply(normal,h);
        max_nl=mul_nl>0?mul_nl:0;
        max_nh=mul_nh>0?mul_nh:0;
L.red=I.ka.red*I.Ia.red+I.kd.red*I.Ib.red*max_nl+I.ks.red*I.Ib.red*pow(max_nh,1000);// eq 4-1
L.green=I.ka.green*I.Ia.green+I.kd.green*I.Ib.green*max_nl+I.ks.green*I.Ib.green*pow(max_nh,
1000);
L.blue=I.ka.blue*I.Ia.blue+I.kd.blue*I.Ib.blue*max_nl+I.ks.blue*I.Ib.blue*pow(max_nh,1000);
        return L;
}
```

Code 4-1

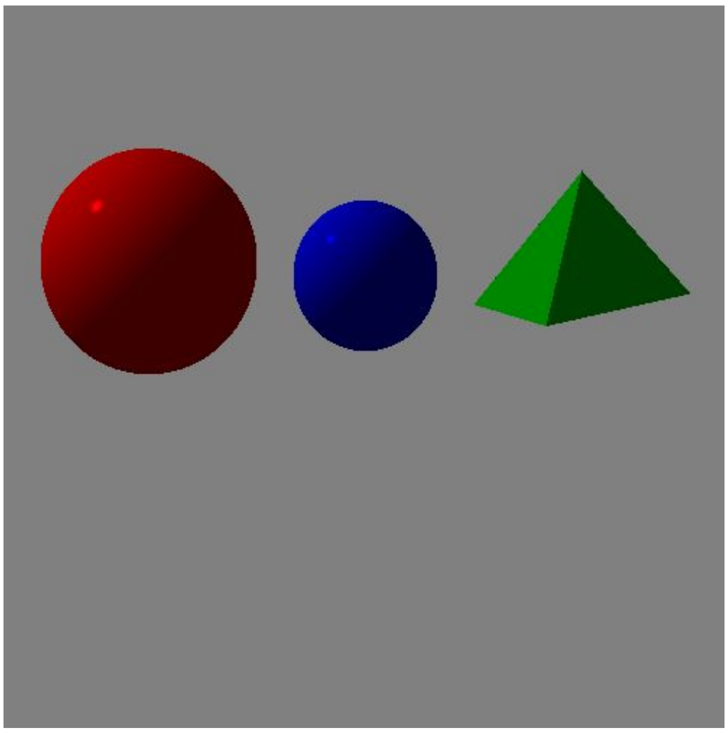


Figure 4-2

V. Glazed Surface

To render a glazed surface, there must be two elements: shadow and reflection.

1. Shadow

As is shown in Figure 5-1, shadows can be added very easily. light comes from direction **l**, pixel is tracing at a point p on a surface being shaded. Tracing in direction **l**, the point is in shadow if it can be traced to a point of the object(like point **q**), otherwise the light is not blocked(as **p**).

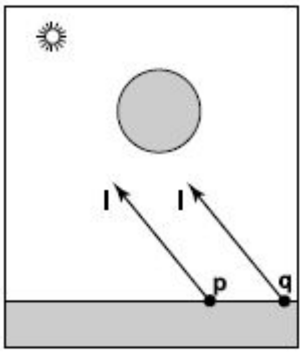


Figure 5-1

The shadow is implemented as the following code section(Code 5-1). The figures are listed as 6-1 and 6-2.

2. Reflection

It is straightforward to add specular reflection. The key observation is shown in Figure 5-2. Where a point is traced from pixel **e** sees what is in direction **r** as seen from the surface. Vector **r** is found using a variant of the Phong lighting reflection

$$\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}.$$ Equation 5-1.

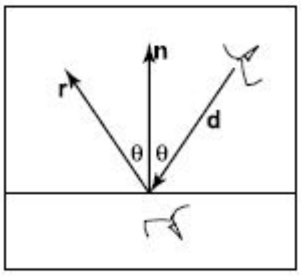


Figure 5-2

The reflection is also implemented as the following code section(Code 5-1). And the shadow is always shown over the reflection in reality, so the project judges about the reflection first. The figures are listed as 6-1 and 6-2.

```
for (x=0;x<500;x++)
    for(y=0;y<500;y++)
    {
        .....//parameters initializing
        interpoint0=GetInterPoint_Surface(pixel,LookAt,sur);/*Draw the surface*/
        if(IntSurface==false)//Background
        {
            image[y][x][0] =0.8;
            image[y][x][1] =0.8;
            image[y][x][2] =0.8;
        }
        else
        {
            image[y][x][0] =0.5;//color of the surface
            image[y][x][1] =0.5;
            image[y][x][2] =0.5;
            pro_dir=get_Projection(LookAt,sur.normal);
            pro_dir.x=-pro_dir.x;
            pro_dir.y=-pro_dir.y;
            pro_dir.z=-pro_dir.z;
            interpoint1=GetInterPoint_Sphere(interpoint0,pro_dir,sphere1);/sphere1 refl
            if(IntSphere==true)
            {
                normal=getNormal_Sphere(interpoint1,sphere1);
                pixelcolor=get_pixel_color(LigDir,LookAt,normal,light1);
                image[y][x][0] =pixelcolor.red*sphere1.c.red*0.5;
                image[y][x][1] =pixelcolor.green*sphere1.c.green*0.5;
                image[y][x][2] =pixelcolor.blue*sphere1.c.blue*0.5;
            }
            interpoint2=GetInterPoint_Sphere(interpoint0,pro_dir,sphere2);//sphere2 refl
            if(IntSphere==true)
            {
                normal=getNormal_Sphere(interpoint2,sphere2);
                pixelcolor=get_pixel_color(LigDir,LookAt,normal,light1);
                image[y][x][0] =pixelcolor.red*sphere2.c.red*0.5;
                image[y][x][1] =pixelcolor.green*sphere2.c.green*0.5;
                image[y][x][2] =pixelcolor.blue*sphere2.c.blue*0.5;
            }
            interpoint3=GetInterPoint_Tetrahedron(interpoint0,pro_dir,T);
            if(IntTetrahedron==true)//tetrahedron refl
            {
                normal=getNormal_Triangle(IntTriangle_No);
                pixelcolor=get_pixel_color(LigDir,LookAt,normal,light1);
                image[y][x][0] =pixelcolor.red*T.c.red*0.5;
                image[y][x][1] =pixelcolor.green*T.c.green*0.5;
                image[y][x][2] =pixelcolor.blue*T.c.blue*0.5;
            }
            ligtemp.x=-LigDir.x;
            ligtemp.y=-LigDir.y;
            ligtemp.z=-LigDir.z;
            interpoint1=GetInterPoint_Sphere(interpoint0,ligtemp,sphere1);
            if(IntSphere==true)//in shadow of sphere1
            {
                image[y][x][0]= light1.ka.red*light1.Ia.red;//ka*Ia
                image[y][x][1]= light1.ka.green*light1.Ia.green;
                image[y][x][2]= light1.ka.blue*light1.Ia.blue;
            }
            interpoint2=GetInterPoint_Sphere(interpoint0,LigDir,sphere2);
            if(IntSphere==true)//in shadow of sphere2
            {
                image[y][x][0]= light1.ka.red*light1.Ia.red;//ka*Ia
                image[y][x][1]= light1.ka.green*light1.Ia.green;
                image[y][x][2]= light1.ka.blue*light1.Ia.blue;
            }
            interpoint3=GetInterPoint_Tetrahedron(interpoint0,ligtemp,T);
            if(IntTetrahedron==true)//in shadow of tetrahedron
            {
                image[y][x][0]= light1.ka.red*light1.Ia.red;//ka*Ia
                image[y][x][1]= light1.ka.green*light1.Ia.green;
                image[y][x][2]= light1.ka.blue*light1.Ia.blue;
            }
        }
```

Code 5-1

V. Animation

The animation is implemented just by adding **glutPostRedisplay()**, and then keeps the pixel screen changing continuously. It is worth to mention that to make the view comfortable and easy for watching, the project keeps the screen been vertical with the LookAt vector while rotating. The code and the figures are as the following (Code 6-1 and figure 6-1 6-2).

```
LookAt.x=sin(alpha);
LookAt.y=0.3;
LookAt.z=cos(alpha);
LookAt=getUnit(LookAt);
for (x=0;x<500;x++)
    for(y=0;y<500;y++)
    {
        d=10;
        x_d=double(x);
        y_d=double(y);
        pixel.x=(x_d-250)/250*cos(alpha)+sin(alpha);//keep the screen vertical
        pixel.y=(y_d-250)/250;
        pixel.z=cos(alpha)-(x_d-250)/250*sin(alpha);//pixel's location
        .....//other funtions
     }
alpha=alpha+0.3;
if(alpha>6.28)
    alpha=0;
glutPostRedisplay();
```

Code 6-1

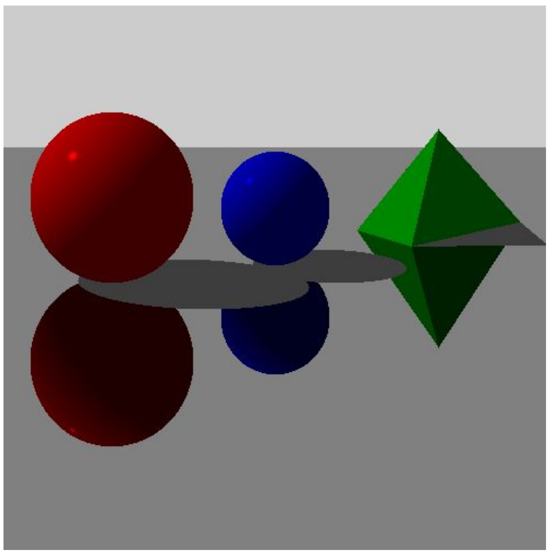


Figure 6-1

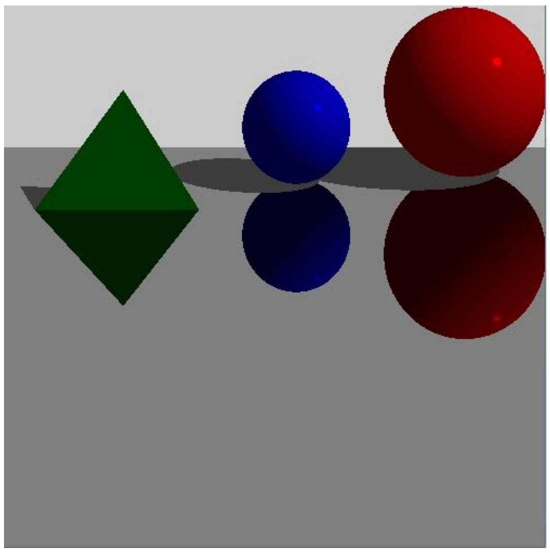


Figure 6-2