CG Assignment2 Ray Tracing

Yongxi Huang – 515030910469

Environment: VS2015+OpenGL (glut), glut is only used to draw pixels in the window.

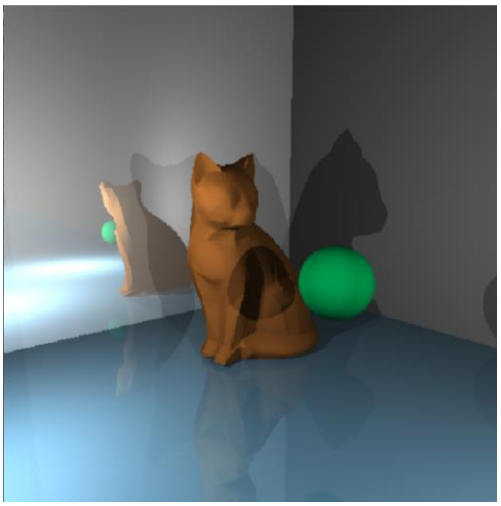
1 How to Run

Main function is in file “ques_h2.cpp”. You can add any object to the scene using `scene.push(&object);` and add any light using `lights.push(&light1);` . The scene will be rendered automatically.

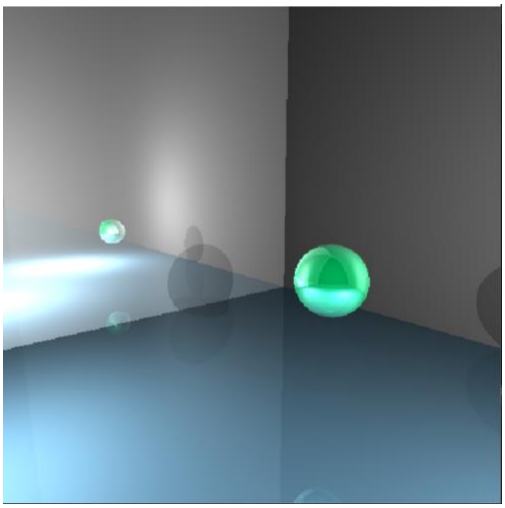
As the rendering of ply model is too time consuming, I have comment the code rendering ply model. So if you run the program directly, you will get result as shown in **fig1(b)**. If you want to render the ply model, please uncomment the code at line 159~168 in “ques_h2.cpp”.

I also comment all the code about the animation rendering. You can uncomment them in “ques_h2.cpp” if you want to try animation rendering.

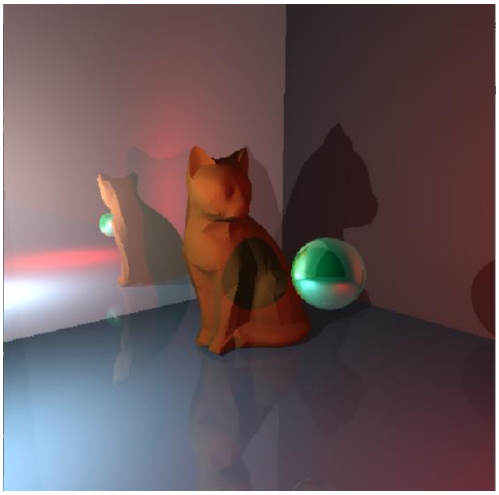
2 Result



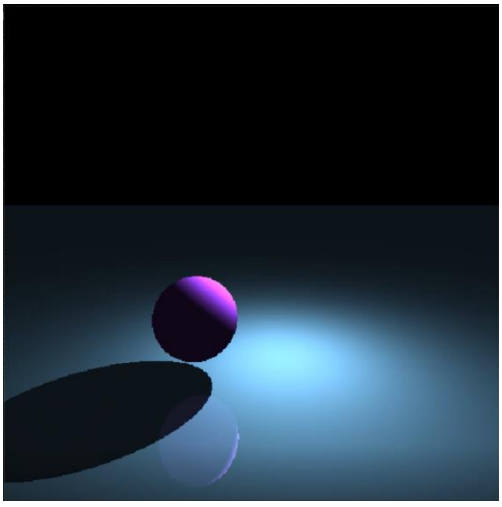
(a)



(b)



(c)



(d)

fig1. Results

(a) Here I draw one ply model, one sphere, three planes and two point lights. The ambient light is set as 0.2. And set the left and bottom plane to be reflective.
(b) Here I remove the ply model, and set the sphere to be reflective.
(c) Change the color of one point light.
(d) A screenshot of animation. The whole video can is in “animation.mp4”.

3 Principle

3.1 Ray Tracing

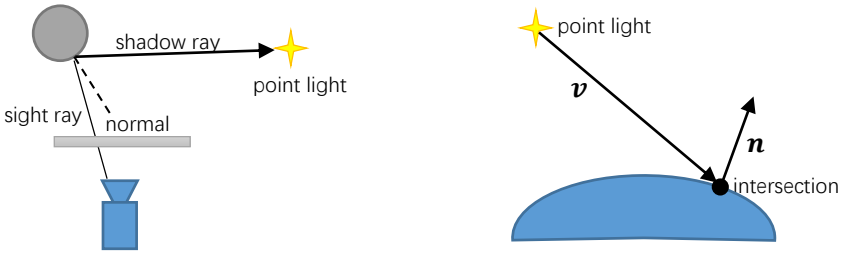


fig2. Ray Tracing

As shown in **fig2(a)**. Several sight rays are generated from viewpoint. If they are intersected with any object, we can compute the color of the pixel from which the sight ray is generated. As for shadow, we will create a ‘shadow ray’ from the intersection. If the shadow reaches any object before light, it means the intersection is in shadow. Otherwise we will compute the color at the intersection. Here we compute the color of the intersection in following way:

$$
\begin{aligned}
&\boldsymbol{n}: normal\ \ of\ \ the\ \ intersection\\
&\boldsymbol{pos}: position\ \ of\ \ the\ \ intersection\\
&\boldsymbol{origin}: origin\ \ of\ \ the\ \ light\\
&\boldsymbol{v} = \boldsymbol{pos} - \boldsymbol{origin}\\
&\boldsymbol{if}\ \ \boldsymbol{v.dir} \cdot \boldsymbol{n} > \boldsymbol{0}: we\ \ cannot\ \ see\ \ the\ \ object\\
&\quad \boldsymbol{return}\ \ \boldsymbol{black}\\
&\boldsymbol{else}: we\ \ can\ \ see\ \ the\ \ object\\
&\quad \boldsymbol{color} = \boldsymbol{light.intensity} * |\boldsymbol{v.dir} \cdot \boldsymbol{n}| \frac{\boldsymbol{1}}{\boldsymbol{v^2}} * \boldsymbol{object.color}\\
&\quad \boldsymbol{return}\ \ \boldsymbol{color}
\end{aligned}
$$

This is the way to compute the color when the light is point light. Here $\boldsymbol{v}$ is the ‘light ray’ which is from light origin to intersection. $\boldsymbol{v.dir}$ is the direction of light. $|\boldsymbol{v.dir} \cdot \boldsymbol{normal}|$ can be seen as a fade factor which means the object with a larger inclination angle will be darker. And $\boldsymbol{v^2}$ is the square of the distance between intersection and point light which means the brightness of the intersection will fade if the light is far from it. The way to computer color is shown in **fig2(b)**.

However, if the light is ambient light, we can get the color of intersection directly by $\boldsymbol{object.color} * \boldsymbol{light.color}$.

3.2 Reflecting

If the object is reflective, we can implement reflecting by create a reflect ray from intersection and do ray tracing recursively.

3.3 Rendering Pipeline

$for\ \boldsymbol{pixel}\ in\ window$:
    $generate\ \boldsymbol{ray}\ from\ pixel$
    $rayTracing$:
    $for\ \boldsymbol{object}\ in\ scene$:
        $conpute\ \boldsymbol{intersection}\ of\ \boldsymbol{ray}\ and\ \boldsymbol{object}$
    $\boldsymbol{inter} = the\ \boldsymbol{intersection}\ that\ is\ closest\ to\ comera$
    $for\ \boldsymbol{light}\ in\ all\ lights$
        $conpute\ \boldsymbol{color}$
        $merge\ \boldsymbol{color}$
    $if\ reflective$:
        $do\ rayTracing\ again\ using\ reflected\ ray$
    $draw\ \boldsymbol{pixel}\ in\ window\ using\ \boldsymbol{color}$

4 Implement

4.1 class GVector3

Implementation of 3D vector. It contains 3 coordinates (x, y, z). I overload several operators such as '=', '+', '-', '*', '/' and implement the dot multiply and cross multiply. GVector3 is defined in file 'gvector.h' and implemented in 'gvector.cpp'.

It is the most basic class in my project and is widely used in the whole project.

4.2 class Ray

Ray can be expressed by $\boldsymbol{e} + t\boldsymbol{d}$, where $\boldsymbol{e}$ is the origin of ray, $\boldsymbol{d}$ is direction and t is distance. So Ray has two member, origin and direction. And I implement a function `GVector3 getPoint(double t);` which can output a position on the ray given distance t.

```
class CRay
{
private:
    GVector3 origin;
    GVector3 direction;
public:
    CRay();
    CRay(const GVector3& o,const GVector3& d);
    ~CRay();
    void setOrigin(const GVector3& o);
    void setDirection(const GVector3& d);
    GVector3 getOrigin();
    GVector3 getDirection();
    GVector3 getPoint(double t);
};
```

4.3 objects

I implement several kinds of objects. All of them is inherited from **BaseObj**.

```
class BaseObj {
public:
    BaseObj();
    ~BaseObj();
    virtual Intersection intersect(Ray& RAY);
    Material *material;
};
```

`material` contains some important information such as the color and reflectivity of the object. Material is defined in 'material.h'.

```
class Material{
private:
    Color color;
    double reflectivity;
public:
    double getRef();
    Color getColor();
    void setRef(const double& _reflectivity);
    void setColor(const Color& _color);
    Material();
    Material(const Color& _color, const double& _reflectivity = 0);
    virtual ~Material();
};
```

Intersection is a structure contains all the information about intersection of ray and object.

```
struct Intersection {
    double distance;
    bool isHit;
    GVector3 position;
    GVector3 normal;
    BaseObj* object;
    Intersection() {
        isHit = 0;
        object = NULL;
    }
    static inline Intersection noHit() { return Intersection(); }
};
```

```
distance:   distance from eye (origin of ray).
isHit:      whether the object is intersected by ray.
position:   position of intersection.
normal:     normal of the object at the intersection.
object:     the object that is intersected by ray.
```

When implement different objects, computing these parameters of intersection is the most important work. In other words, we should rewrite function `virtual Intersection intersect(Ray& RAY)` for every object class. Here I implement three different kinds of object: Plane, Sphere and face.

4.3.1 Plane

Plane can be defined by $\{P|\boldsymbol{n}\cdot(P-P_0)=0\}$, where $\boldsymbol{n}$ is the normal of the plane and P_0 is one point on the plane. So class Plane has two member variables: normal of the plane `GVector3 normal` and distance from origin `double d`.

Given an ray $\boldsymbol{r}$, the way to compute intersection is shown in **fig3**.

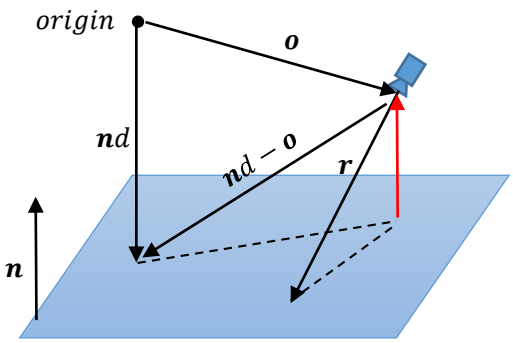


fig3. Plane

d: $distance\ from\ origin\ to\ Plane\ (Note\ that\ d\ is\ negative\ here)$.

$\boldsymbol{n}: normal\ \ of\ \ Plane.$

$\boldsymbol{o}: position\ \ of\ \ camera.$

$\boldsymbol{if}\ \ \boldsymbol{n}d < 0: \ the\ \ ray\ \ is\ \ intersected\ \ with\ \ plane$

$\boldsymbol{There\ \ is}\ \ (\boldsymbol{n}d - \boldsymbol{o}) \cdot \boldsymbol{n} = \boldsymbol{r} \cdot \boldsymbol{n} \cdots\cdots$ (1)

When computing intersection, we know $\boldsymbol{n}$, d, $\boldsymbol{o}$ and $\boldsymbol{r}.\boldsymbol{direction}$. According to (1), we can get $\boldsymbol{r}.\boldsymbol{length}$. Then we can compute parameters of `Intersection` by:

```
Intersection.distance = r.length;
Intersection.position = ray.getPoint(distance);
```

4.3.2 Sphere

Sphere can be defined by $\{P||P - c| = radius\}$, where c is the center of sphere and radius is the radius of sphere. So class Sphere only has two member variables: `GVector3 center` and `double radius`.

Given an ray $\boldsymbol{r}$, the way to compute intersection is shown in **fig4**.

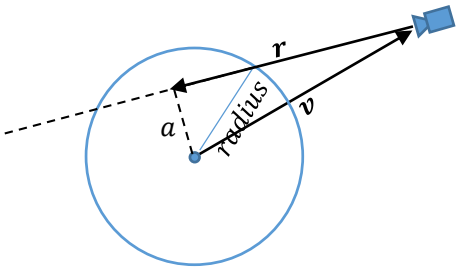


fig4. Sphere

$\boldsymbol{v} = ray.origin - sphere.center$

$r = ray.direction \cdot \boldsymbol{v}.$

$a^2 = \boldsymbol{v}^2 - \boldsymbol{r}^2$

$if\ \ r < 0\ \ and\ \ a < r: \ the\ \ ray\ \ is\ \ intersected\ \ with\ \ s$

$Intersection.distance = |r| - \sqrt[2]{radius^2 - a^2}$

$Intersection.position$ `= ray.getPoint(`$Intersection.distance$`)`

4.3.3 face

face is the basic unit of 3D mesh model. A face can be seen as a triangle in 3D coordinate. So class face contains three member variables `GVector3 v1,GVector3 v2, GVector3 v3`.

Let v_1, v_2, v_3 be three vertices of face. Then all the points on the face can be defined by $\{P \mid P = (1 - \beta - \gamma)v_1 + \beta v_2 + \gamma v_3, \quad 0 < \beta < 1, \quad 0 < \gamma < 1\}$. As show in 4.2, ray can be defined as $\boldsymbol{e} + t\boldsymbol{d}$. We already have $\boldsymbol{e} = ray.origin$ and $\boldsymbol{d} = ray.direction$. There is:

$$\boldsymbol{e} + t\boldsymbol{d} = (1 - \beta - \gamma)v_1 + \beta v_2 + \gamma v_3$$

We can get t, β, γ by solving this equation. Then we can get the parameters of `Intersection` by

$if\ \ 0 < \beta < 1\ \ and\ \ 0 < \gamma < 1\ \ and\ \ t > 0$:
    $the\ \ ray\ \ is\ \ intersected\ \ with\ \ face.$
    $Intersection.distance = t$
    $Intersection.position = ray.getPoint(t)$
$else$:
    $The\ \ ray\ \ is\ \ not\ \ intersected\ \ with\ \ face.$

4.3.4 objUnion

objUnion is a container of objects. Its intersect function will call the intersect function of all the objects in it and return the intersection that is closest to the origin of ray.

4.4 light

I implement two kinds of lights: PointLight and AmbientLight. Both of them are inherited from class Light.

```
class Light {
public:
    Color color;
    bool isShadow;
    double intensity;
    Light();
    virtual ~Light();
    virtual Color intersect(objUnion &scence, Intersection &result);
};
```

The most important function of Light is `virtual Color intersect(objUnion &scence, Intersection &result)` which is used to compute the color of intersection of sight ray and object. For PointLight, it also judges whether the intersection is in shadow.

4.4.1 class PointLight

PointLight has another member variable `GVector3 position` which set the position of light. Its intersect function is as follow.

```
Color PointLight::intersect(objUnion &scence, Intersection &result)
{
    double factor = 0.05;
    GVector3 v = result.position - position;
    GVector3 vDir = v;
    vDir.normalize();
    double DdotV = result.normal.dotMul(vDir);
    if (DdotV >= 0) {
        return Color::black();
    }
    double distance = v.getLength();
    Ray shadowRay = Ray(result.position+(vDir.negate()*factor), v.negate());
    Intersection lightResult = scence.intersect(shadowRay);
    if (lightResult.isHit && (lightResult.distance <= distance)){
        return Color::black();
    }
    else {
        double fadeIntensity = -(intensity / (distance*distance))*DdotV;
        return
((color*fadeIntensity)*(result.object->material->getColor())).saturate();
    }
}
```

As said in section 3.1, it will generate a shadowRay used to judge whether the intersection is in shadow. The final return color is the multiplication of light color and object color.

4.4.2 class AmbientLight

We do not consider fading and shading when compute the color of intersection lighted by AmbientLight. So the intersect function of AmbientLight is really simple.

```
Color AmbientLight::intersect(objUnion &scene, Intersection &result) {
    if (result.isHit)
        return color*(result.object->material->getColor());
    else
        return Color::black();
}
```

4.4.3 lightUnion

lightUnion is a container of lights. Its intersect function will call the intersect function of all the lights in it and merge their returned color together.

4.5 rendering

I write a function rend() in 'render.h' to rendering the whole scene. It is the implementation of rendering pipeline shown in section 3.3.

```
Color rend(Ray& ray, objUnion& scene, lightUnion& lights, bool isRef =
false, int maxReflect = 0) {
    Intersection result = scene.intersect(ray);
    if (result.isHit == 0) {
        return Color::black();
    }
    Color color = lights.intersect(scene, result);
    if (isRef && maxReflect > mini)
    {
        float reflectivity = result.object->material->getRef();
        GVector3 r = result.normal*(-2 *
result.normal.dotMul(ray.getDirection())) + ray.getDirection();
        Ray reflectRay = Ray(result.position, r);
        Color reflectedColor = rend(reflectRay, scene, lights, isRef,
maxReflect - 1);
        color = color.add(reflectedColor.multiply(reflectivity));
    }
    return color;
}
```

Input a sight ray and all the objects and lights in the scene, it will return the color you can get from this sight ray. It will call itself recursively to implement reflecting.