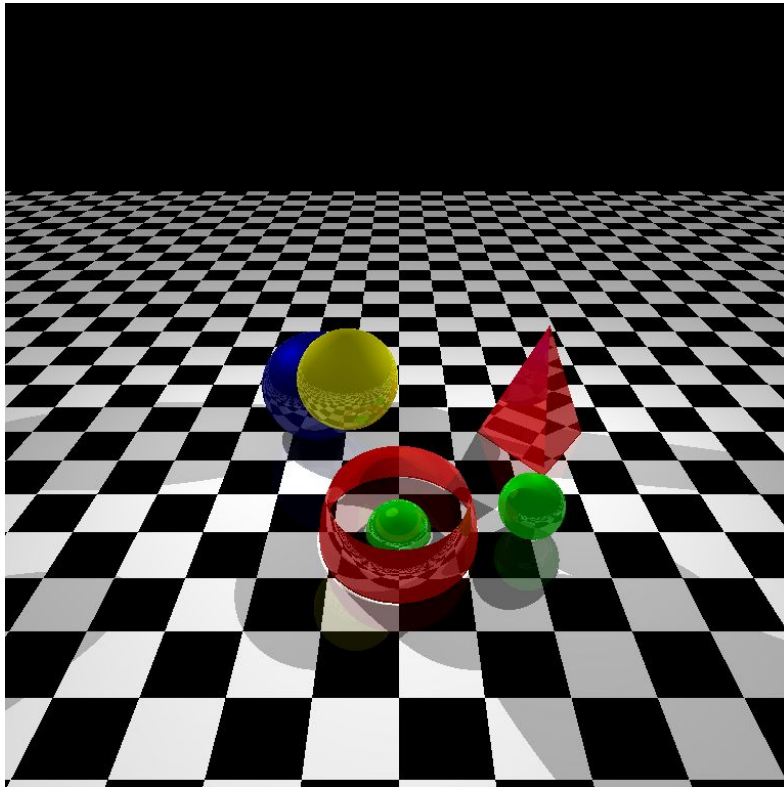


Assignment III: Ray Tracing :)

Submission Deadline: 12th Week Friday 11.55am 26/05/2017 (No excuses)

In this assignment you will have to generate realistic image for common shapes like the picture below.



Please check the attached **OpenGL.exe** and **scene.txt** file for better understanding the mechanism.

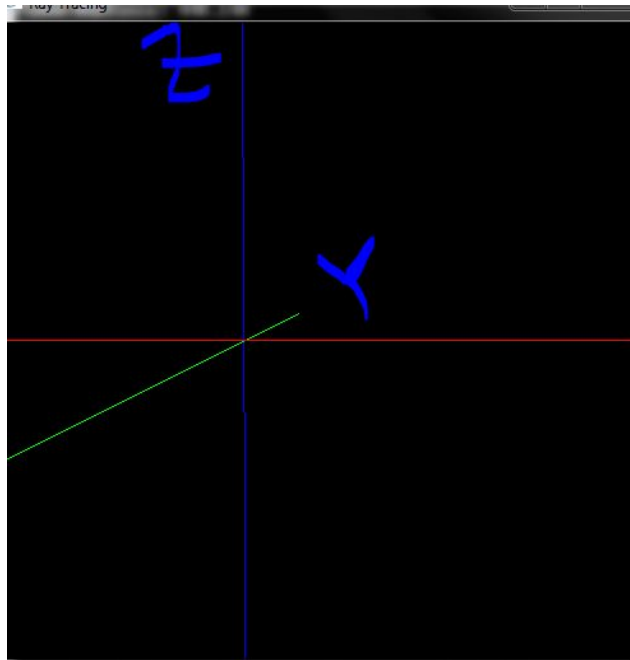
<https://drive.google.com/open?id=0B-SybtAwa8vcRG5xeVVpYW8yQIE>

The scene.txt file contains the configuration and also the explanation of each values. [Read that at first].

Procedure:

Task 1: Control Over Scene

1. First, it is important to make sure that your camera rotation code from assignment1 is fully working.
 - a) So check it again and make sure you are able to navigate in any position freely.
 - b) Set your eye, look, up such that you are looking at x-y-z like the following (preferable for testing)
 - c) `cam.set(Point3(0, -200, 10), Point3(0, 0, 0), Vector3(0, 0, 1))` something like this
 - d) `#define your Window_width, Window_height, 500x500` for test case



e)

Task 2: Creating Environment

1. In your main function refer to a function `loadTestData()`

```
Void loadTestData(){ }
```

```
main(){  
    loadTestData();  
}
```

```

        //others
    }

```

We will customize the different shape and configuration here for test purposes. Later you have to create a function loadActualData() which will read from the scene.txt file.

2. Create a separate header file/src file with preferable name. Here we will create most of the classes. (You can do everything in same file, but better approach is to module your codes for simplicity). Say here the filename is FILE2, and we have main codes in MAIN_FILE

In FILE2 we will create a Base Class Object with following methods and properties initially. Later you should add and refactor

```

Object{
    Vector3 reference_point;
    Double height, width, length;
    Int Shine;
    Double color[3];
    Double co_efficients[4];

    Object(){ }
    Virtual void draw(){}
    Void setColor
    Void setShine
    Void setCoEfficients
}

```

And a derived class

```

Sphere: Object{
    Sphere(Center, Radius){
        reference_point=Center;
        length=Radius;
    }

    Void draw(){
        //write codes for drawing sphere
    }

}

```

3. a) In MAIN_FILE Keep two vectors one for objects and another one for lights and **make it accessible to FILE2 too. (just use extern)**

```
Vector <Object>  objects;  
Vector <Vector3> lights; // or you could just typedef Vector3 to Light for  
clarity
```

b) In your loadTestData() function

```
Object *temp;  
  
temp=new Sphere(Center, Radius); // Center(0,0,10), Radius 10  
temp->setColor(1,0,0)  
temp->setCoEfficients(0.4,0.2,0.2,0.2)  
temp->setShine(1)  
  
objects.push_back(temp);  
  
Vector3 light1(-50,50,50);  
lights.push_back(light1);
```

c) in you display method where

- i) Loop over the objects and call draw method
- ii) Loop over the lights object and draw Point for each light souce to visualize position

d) Test it.

4. Create a derived class Floor

```
Floor: Object{  
    Floor(FloorWidth, TileWidth){  
        reference_point=(-FloorWidth/2, -FloorWidth/2,0);  
        length=TileWidth;  
    }  
  
    Void draw(){
```

```

        //write codes for drawing blackand white floor
    }
}

```

Now write your draw methods such that it creates a checkerboard of black and white with alternating color on each tileWidth.

Add,

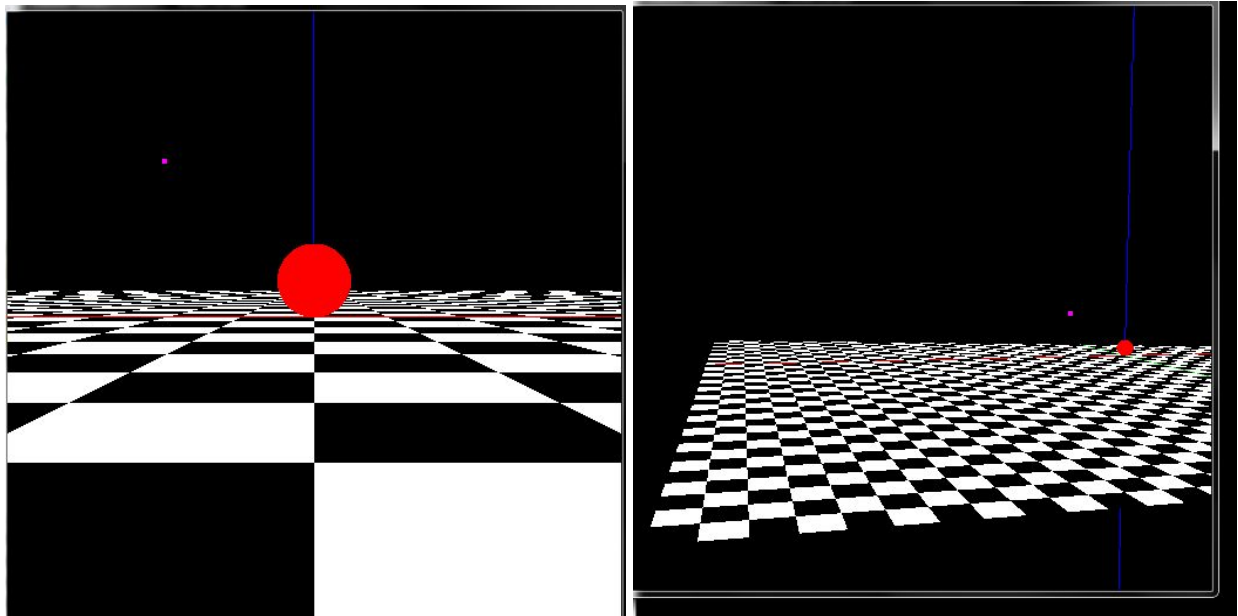
```

temp=new Floor(1000, 20);
temp->setCoEfficients(0.4,0.2,0.2,0.2)
temp->setShine(1)
objects.push_back(temp)

```

Test it;

Check it should look Something like the followings



Task 3: Hidden Surface Removal

1. Create a method Capture() in MAIN_FILE which will be called when you press 0
2. In loadTestData() set global variable values for image_width, 768 for test case

3. In FILE2 create a class

```
Ray{
    Vector3 start;
    Vector3 dir;

    //write appropriate constructor
}
```

4 Now the pseudocode for intersecting checkings are

a) capture():

```
Initialize bitmap_image of image_widthximage_width to black

plane_distance= (window_height/2)/tan(VIEW_ANGLE/2)

//here l, r, u direction of camera depends on your implementation so
use +/- correctly
topleft= eye - l*plane_distance-r*WINDOW_WIDTH/2+u*WINDOW_HEIGHT/2);

du=window_width/image_width;
dv=window_height/image_height;

For i=1:image_width
    For j=1:image_width
        corner=Find corner point for i, j th pixel using eye
similar to topleft above
        Create a Ray using (eye, (corner-eye)) //always normalize
direction

        nearest=-1;

        For each object k
            t =object[k]->intersect(ray, dummyColorAt, 0)
```

```

        //dummyColorAt is the color array where pixel value
        will be stored in return time. As this is only for nearest
        object detection dummy should be sufficient. Level is 0
        here

        if(t<=0) continue;

        Update t, nearest if t<t_min
    End

    if(nearest!=-1)
        t =object[nearest]->intersect(ray, colorAt, 1)

        //in this case we know nearest object so level should
be set to 1

        //we will deal with this later

        Update_image_i_j pixel value
    end

End
End

save_image

```

b) In Object base class create a virtual method intersect

```

Virtual double intersect(Ray *r, double *current_color, int level){

    Return -1;

}

```

c) Now in Sphere Derived Class override this function

In this function you have to calculate the sphere ray intersection:

Please refer to your ray_casting Slide Page no 31, 32

Here you will find necessary calculation for calculating t

Ray-Sphere Intersection

- Quadratic: $at^2 + bt + c = 0$
 - $a = 1$ (remember, $\|R_d\| = 1$)
 - $b = 2R_d \cdot R_o$
 - $c = R_o \cdot R_o - r^2$
- with discriminant $d = \sqrt{b^2 - 4ac}$
- and solutions $t_{\pm} = \frac{-b \pm d}{2a}$

A= dot(ray->dir, ray->dir)

B= from equation

C= from equation

D=B^2-4ac

If D<0 return -1;

Otherwise Calculate t1, t2

Update current_color=color // for the time being testing purpose

Return the minimum t

d) Now test it, Make sure everything working

If everything works then you should see an image with only a circle in it

If it does not work, then in capture function

Set i, j to a specific pixel and check for intersection

Or in intersect function, Set custom Ray->start(0,100,10) and Ray->direction (0, 1, 0)

So for a sphere centered at 0,0,10 with radius 10.

You should get two intersecting point 0,-10,10 and 0,10,10

If works add more sphere and test it, it should work as hidden surface removal procedure

Task 4: Illumination

a)

Now That Your hidden surface is working. The next step is to add some lighting. If you look closely, the purpose of **level variable in intersecting method is to determine the nearest object.** So no color computation actually necessary here.

So after computation of intersecting t in do a simple check like following

```
If (level==0){  
    Return t;  
}
```

b) Now if level is not 0 (here 1) then add some lighting codes, and regroup functions like following

So skeleton of your function should look like this

```
double intersect(Ray *r, double *current_color, int level){  
    t= getIntersectingT(Ray *r) //perform computation of intersection here  
  
    If t<=0 return -1  
    if(level=0)return t;  
  
    intersectionPoint = r->start+ r->direction*t;  
  
    colorAt=getColorAt(intersectionPoint)  
    // generally this function should return single color but for  
    checkerboard like plane color depends on intersectionPoint  
  
    setColorAt(current_color, colorAt)  
  
    Return t;  
}
```

At setColorAt function multiply each colorAt color value with ambient coefficient

```
current_color=colorAt*co_efficient[AMBIENT];
```

Because AMBIENT means how normally illuminated an object is

Test it. So see Object getting dimmer.

c) After the above step, add the following codes after setColorAt()

```
normal=getNormal(intersectionPoint);  
reflection=getReflection(ray, intersectionPoint);
```

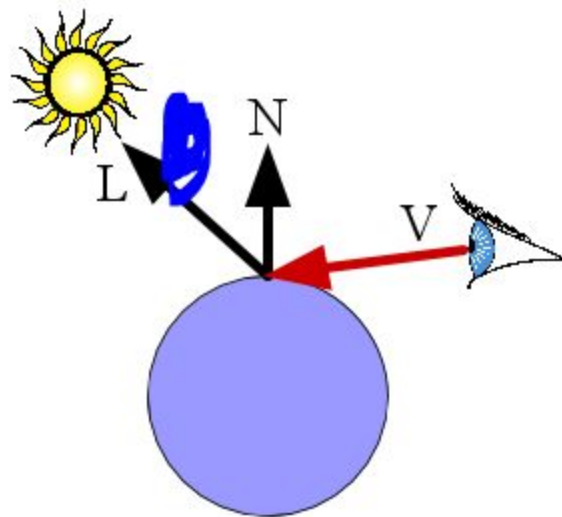
**to calculate reflection from incident ray at intersectionPoint check the formula

$\text{reflection} = 2 (\text{ray} \rightarrow \text{direction} \cdot \text{normal}) \text{normal} - \text{ray} \rightarrow \text{direction}$ // may be different for you
Normalize it

http://asawicki.info/news_1301_reflect_and_refract_functions.html

d) Now you have to check whether the intersecting point is obscured by any objects from the light

Because if light source is obscured by an object then no impact of light will be applicable on the intersecting pixel



So after the above part do the following

For each light source

```
Construct L ray like in the picture  
direction= (lightSource-intersectionPoint) //normalize it  
start= intersection + direction*1 //1 is for taking slightly above the  
point so it doesn't again intersect with same object due to precision
```

```
Ray L(start, direction)
```

For each object now check whether this L ray obscured by any object or not.

If it is obscured that means light falls onto the intersection point so you have update current_color,

Calculate lambert value,

Calculate phong value

//check the illumination slide for formula

Now update each pixel value of Current_Color by following

current_color+=source_factor*lambert*co_efficient[DIFFIUSE]*color

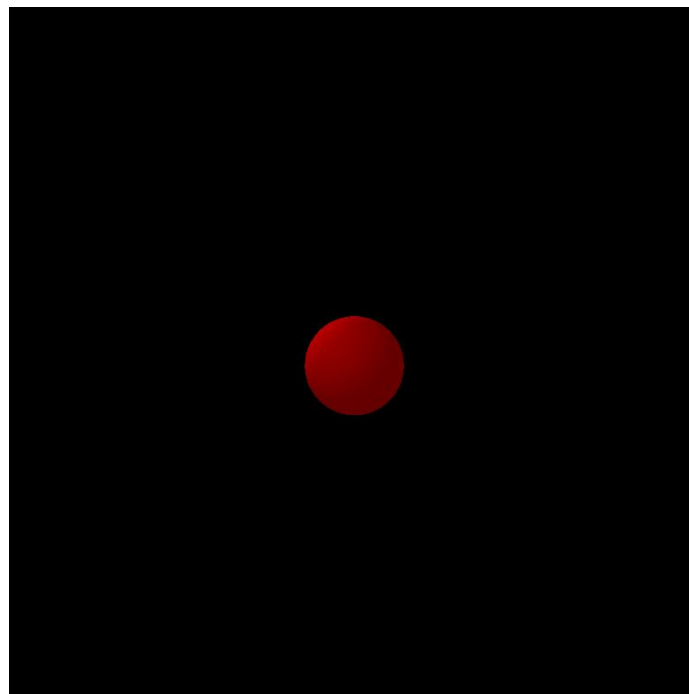
At

current_color+=source_factor*pow(phong,shine)*co_efficient[SPECULAR]*colorAt

end

end

**** IF YOU COMPLETE THE ABOVE PART by then you have completed the illumination PART
:D So your output should look like following



Task 5: Reflection

- a) Reflection means using the reflected ray you do the same as before and how many times you reflect is your Recursion_Level
- b) So in MAIN FILE declare global variable recursion_level, make it available in FILE2 via extern operator and in loadTestData set it as 3 or 4
- c) Now after the above code do the following

```
if(level<recursion_level)
```

```
    start=intersectionPoint+reflection*1 //slight up to avoid own  
intersection
```

```
    Ray reflectionRay(start, reflection)
```

```
    Like capture method, find the nearest intersecting object, using  
intersect function
```

```
    If found
```

```
        objects[nearest]->intersect(reflectionRay, reflected_color,  
level+1);
```

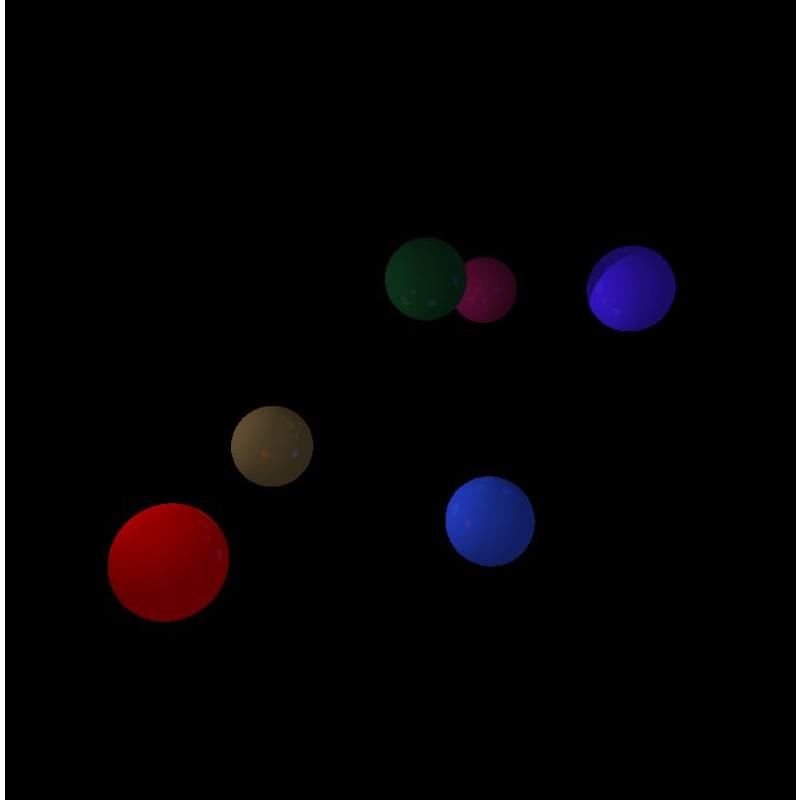
```
        //update currennt_color using reflected_color  
        current_color+=reflected_color*co_efficient[REFLECTION];
```

```
    End
```

```
End
```

```
Check whether all current_color pixel value is within 1 or 0 if not set  
it
```

*** If you complete the above step then for multiple sphere you output should look like the following



Task 6: Floor , Triangle

Ok now if you complete up to this part, clearly you can now see that the differences between Sphere, FLOOR, Triangle are

`getNormal()`, `getColorAt()`, `getIntersectingT()` these functions

So, You Should make appropriate virtual functions and derived methods for handling these

a) For FLOOR,

- i) the normal will be always 0,0,1
- ii) For t calculation of plane, you can use the equation from slide or other ways.
After finding t, calculate the intersectingPoint
If the point is not within the floor then return -1

iii) At getColorAt(intersection) check on which tile the intersection point belongs and return color accordingly

b) For triangle normal calculation

normal= (b-a) X (c-a)

Intersection formula, you can use slides formula or from the following link

https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm

IF you complete this then you have Basic RayTracing

Task 7: General Quadratic

a) If you check the following Link

<http://tutorial.math.lamar.edu/Classes/CalcIII/QuadricSurfaces.aspx>

general Quadratics have following form

$$Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz + J = 0$$

b) Now if you done everything, then you know

Check Class Lecture for details formula

The only change is getIntersectingT() and getNormal()

i) First, getNormal(intersectionPoint)

Normal vector for the above form will be

$$(dF/dx, dF/dy, dF/dz)$$

So find each of them, substitute x, y, z values of intersectionPoint to get Normal

ii) forGetIntersectionT (Ray *r)

You have $x = x_0 + tx_1$, $y = y_0 + ty_1$, $z = z_0 + tz_1$

So if you substitute these then you will find
Equation of form

$$At^2 + Bt + C = 0$$

Like sphere now you can calculate t_1 and t_2

c) Clipping :

Now from the t_1 and t_2

Calculate intersecting_point1 and intersecting_point2

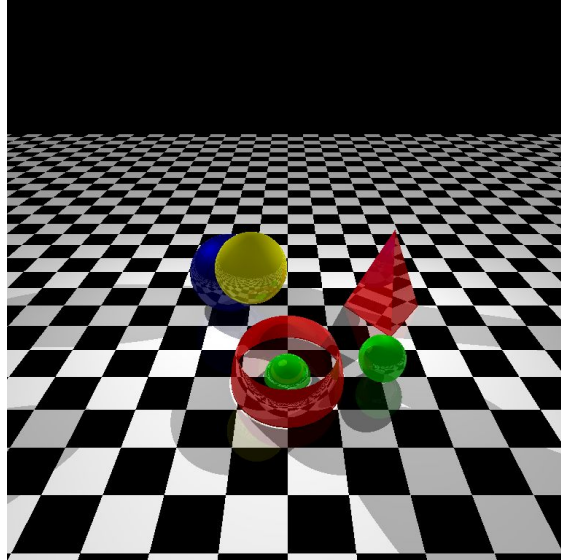
If both point within volume return smallest t

If only one then return that

If none return -1

Task 8 : LoadActualData

- a) load Actual information from scene.txt file
- b) display



Task 9: Clear Memory

- a) Free objects, images, lights and other memories

Do's and Don'ts:

- Do not copy codes, not from your friends, previous years or online. Write it yourself
- Start couple of days earlier because there are many things that can go wrong
- Test your code, and do program incrementally
- Think yourself twice about the problems first then contact me if requires