COSC363 Computer Graphics
Lab07: Ray Tracing

Aim:

This lab aims to provide a good understanding of the fundamental principles, data structures and methods used for ray tracing. In this lab, you will develop a ray tracing program that will display a set of spheres with Phong lighting and shadows. The program will be extended further next week with the inclusion of planar objects and reflections.

I. RayTracer.cpp

1. The program contains the skeleton code for a basic ray tracer. It uses four classes "Color", "Object", "Sphere", and "Vector". The header and implementation files for each of these classes are also provided. Please go through each of these files to get a clear understanding of the structure of the classes and the methods implemented therein. A brief description of the classes follows:

   Vector

   The "Vector" class contains the constructors and functions for defining a 3D point or a vector (*x*, *y*, *z*), and to perform various vector operations. A vector or a point can be defined using any of the following ways:

```
Vector v;  //This is equivalent to Vector(0,0,0);
  or
Vector v(10, 6, -3);
  or
Vector v = Vector(-5, 0 ,0);
```

   The operators *, *=, /, /=, +, +=, –, and –= are all provided. eg:

```
Vector v(5,0,2); //A new vector with x=5, y=0, z=2
v *= 2;          //v now equals (10, 0, 4)
Vector v2 = v*2; //v is not changed,
                 //v2 = Vector(20, 0, 8)
Vector light = Vector(3, 100, 2);
Vector point = Vector(1, 1, 1);
Vector u = light – point;
```

   Functions for length, normalisation, dot product, and cross product are also included. eg:

```
Vector v(15, 6, -2);
u.normalise();
v.normalise();
float dotProd = u.dot(v);    //or v.dot(u)
float dist = u.length();   //Get the magnitude of u
```

Object

This is an abstract class that represents objects in the scene. This provides a generic type for the objects so that they can be stored in a common container (vector, list etc.) and processed in a sequence using an iterator. Each object type such as "Sphere", "Plane" etc. must be defined as subclass of this class, and override geometry specific functions "intersect()" and "normal()".

Sphere

The "Sphere" class is a subclass of "Object" and implements the methods intersect() and normal() for a sphere. A sphere can be defined using any of the following ways:

```
Sphere s;   //This is a unit sphere at the origin
 or
Sphere s(centre, radius, color);
 or
Sphere s = Sphere(centre, radius, color);
```

We can also create pointers to sphere objects as shown in the example below. These pointers could then be stored as pointers of the generic type "Object" which would then exhibit polymorphic behavior when functions like "intersect()" are called on the objects. Use the indirection operator (->) to invoke functions using pointers to objects.

```
Object *s = new Sphere(centre, radius, color);
s->setColor(Color::GREEN);   //not s.setColor(..)
```

The intersect() function of the sphere class returns the value of the parameter t on the ray at the point of intersection (if it exists) with the sphere, otherwise returns a value –1.0. The intersection parameter t is computed as follows:

If $c = (x_0, y_0, z_0)$ denotes the centre, and 'R' the radius, the equation of the sphere is given by

$(p-c)^2 = R^2$, or equivalently $(p-c)\bullet(p-c) = R^2$,

where $p = (x, y, z)$ is any point on the sphere.

The input to the intersect() function is a ray defined using its source p_0 and unit direction d. Any point on the ray can be represented by the parametric equation $p = p_0 + t\,d \quad (t > 0)$.

At the point of intersection of the ray and the sphere,

$(p_0 + t\,d - c)\bullet(p_0 + t\,d - c) = R^2$.

From the above equation, we get

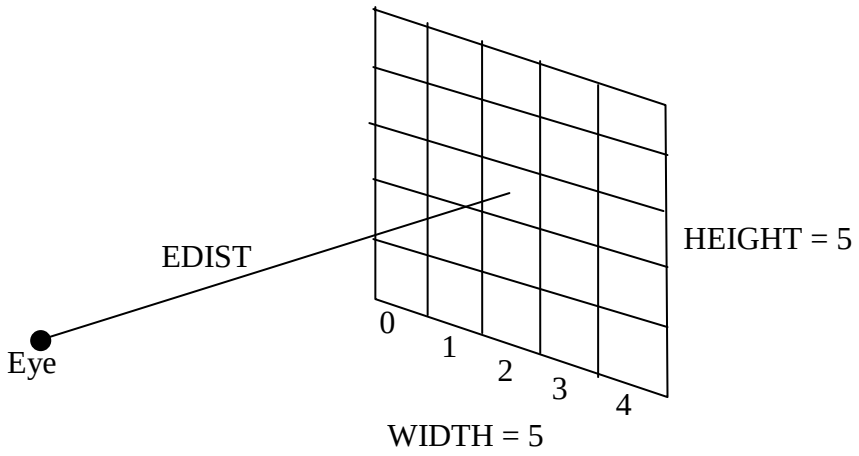
$t = -b \pm \sqrt{b^2 - c}$ where $b = (p_0 - c) \bullet d$, and $c = (p_0 - c)\bullet(p_0 - c) - R^2$.

Color

This class is useful for defining algebraic operations on color such as scaling, addition and multiplication. It also has a function to compute Phong lighting using ambient colour, diffuse term ($\boldsymbol{l}\bullet\boldsymbol{n}$) and the specular term $(\boldsymbol{r}\bullet\boldsymbol{v})^f$.

RayTracer.cpp

At the beginning of the program, you will find a number of constants used for setting up the display window. "WIDTH" and "HEIGHT" define the window's width and height in world units. The parameter "EDIST" specifies the distance between the eye position and the image plane. PPU specifies the number of pixels per unit distance on the image plane. In the following example, if PPU is 20, the image would be of size 100 x 100 pixels. The display function draws each pixel as a square region and fills it with the colour value obtained from the ray through that pixel. This is the only place (apart from GL and GLUT initialization) where OpenGL functions are used.



A ray can be represented using its source point p_0 and direction d. Remember to normalize the vector d. The parametric representation of this ray is $p_0 + td$. Any point on the ray can then be given by a value of t. Since d is a unit vector, t directly gives the distance of the point from the source p_0.

The heart of a ray tracing application is the `trace()` function. In its simplest form, it calls the `closestPt()` function and if a valid intersection point is found, returns the colour at that intersection point. Otherwise it returns the background colour. This process is called ray casting (Slide [8]-8).

PointBundle

This is a structure defined inside RayTracer.cpp. It represents a collection of information related to the closest point of intersection on a ray. The function `closestPt()` uses a ray as input, and returns a value of this type. eg:

```
PointBundle q = closestPt(p, d);    //ray = (p, d)
```

In the above example `q.index` gives the index of the object on which the point of intersection lies (-1 if there is no intersection), `q.point` gives the closest point of intersection (if it exists), and `q.dist` the distance of the point of intersection from the source of the ray. The x, y, z coordinates of the point of intersection can be obtained as `q.point.x, q.point.y, q.point.z.`

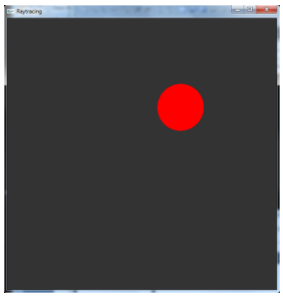
2. Compile and run the program "RayTracer.cpp". You will get a blank screen! This is because no object is defined in the scene. Inside the initialize() function, define a pointer to a sphere object as shown below.

```
Sphere *sphere1 = new Sphere(Vector(5, 6, -70), 3.0, Color::RED);
```

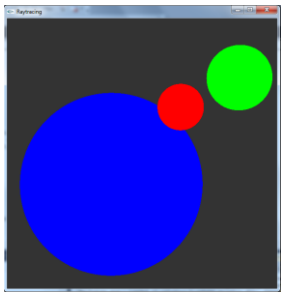
and add it to the array `sceneObjects` (an STL container of type `vector`):

```
sceneObjects.push_back(sphere1);
```

Note that the centre of the sphere is (5, 6, -70) which is along the view axis (-z axis), and is located beyond the image plane. The image plane is at z = –EDIST = –40. The program now generates the display given in Fig.(a).



(a)



(b)

Create a few more spheres like this, all located on the correct side of the image plane, and add them to the array of scene objects (Fig. (b)). The first object added to the array will have index 0, the next object index 1 and so on.

3. We will now modify the `trace()` function so that instead of returning just the object's colour, the function computes the colour using Phong's illumination model. For this, we require the unit normal vector (***n***) at the point of intersection, which can be readily obtained as

```
Vector n = sceneObjects[q.index]->normal(q.point);
```

(Note the polymorphic behavior of the normal() function here).

The light source vector (***l***) is obtained as

```
Vector l = light - q.point;
```

Normalise this vector, and compute the dot product ***l•n***:

```
l.normalise();
lDotn = l.dot(n);   //Note 'l' is the letter el, not the number 1.
```

The material colour is already made available in the function:

```
Color col = sceneObjects[q.index]->getColor();
```

If the dot product is ≤ 0, compute Phong lighting using only the ambient light (background colour) and return this colour:

```
return col.phongLight(backgroundCol, 0.0, 0.0);
```

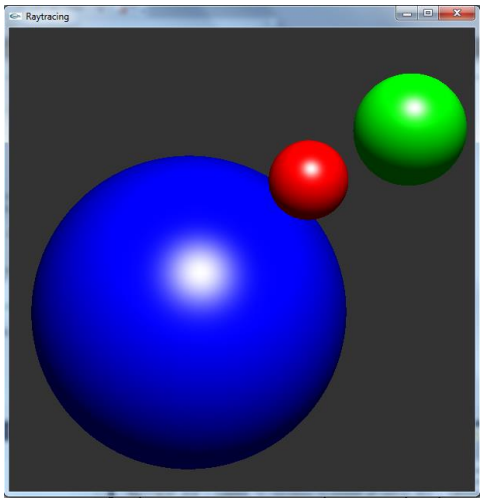
Otherwise compute the specular term as follows, by first calculating the reflection vector ***r***, normalize it, and then computing the ***r•v*** term:

```
Vector r = ((n * 2) * lDotn) - l;    // r = 2(L.n)n – L.  'l' = el
r.normalise();
Vector v(-dir.x, -dir.y, -dir.z);    //View vector;
float rDotv = r.dot(v);
float spec;
if(rDotv < 0) spec = 0.0;
else spec = pow(rDotv, 10);    //Phong exponent  = 10
```

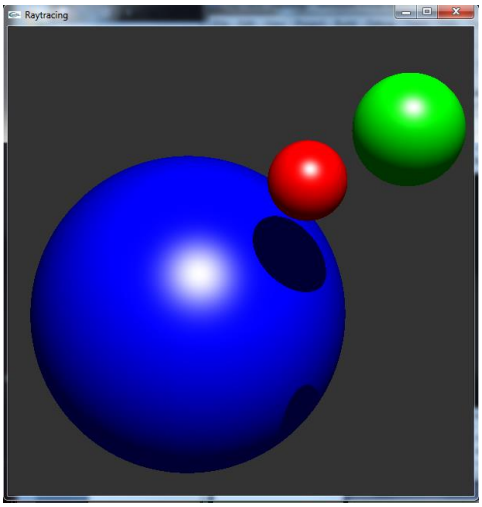
Now we can compute the sum of ambient, diffuse and specular components by calling the function

```
col.phongLight(backgroundCol, lDotn, spec);
```

Return the above colour. The output of the program should look like that given in Fig. (c).



(c)



(d)

4. We can also generate (inside the `trace` function) a shadow ray from the point of intersection. This ray will have `q.point` as the source, and the *unit* light vector as the direction. Call the function `closestPt()` to check if the shadow ray intersects an object. Also verify if the distance to the point of intersection is less than the distance to the light. If so, the point is in shadow, and we return only the contribution of the ambient light (as shown in the previous section). Otherwise we proceed with the computation of the diffuse and specular components as given in the previous section. The output with shadows is shown in Fig. (d).

5. Please save your work. In next week's lab, we will extend the program to include planar surfaces and object reflections (Lab-08).

II. Quiz-07

The quiz will remain open until **5pm, 16-May-2014**.