COSC363 Computer Graphics

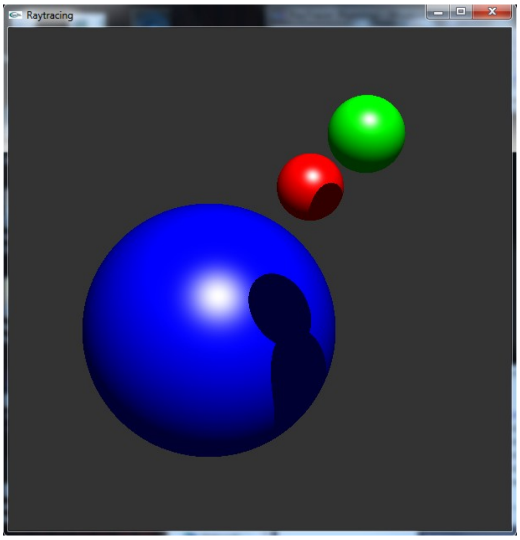
Lab08: Recursive Ray Tracing

Aim:

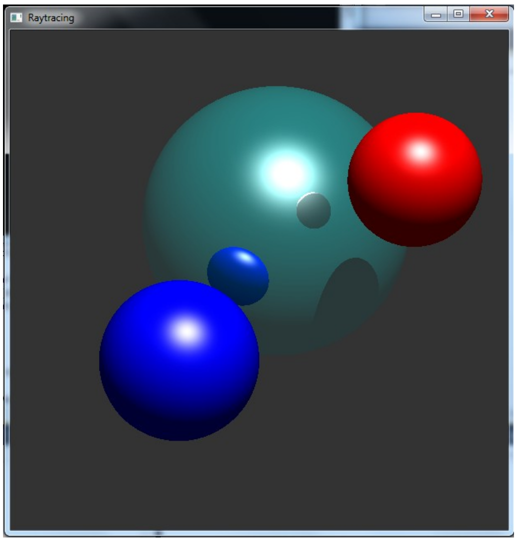
In this lab, you will implement a recursive ray tracer to simulate reflections from surfaces. You will also create functions for rendering planar surfaces.

I. RayTracer.cpp from Lab07

1. In Lab07, you implemented a basic ray tracer for a scene containing a set of spheres, including diffuse, specular reflections and shadows generated by a light source (Fig. (a)).



(a)



(b)

The `trace()` function returned the following colour value computed at the closest point of intersection on the primary ray, provided the point was not in shadow:

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

Instead of returning the colour value, we will now use it to initialize the sum of all colours obtained at that point including those from secondary rays (reflection, refraction etc.):

```
Color  colorSum = col.phongLight(backgroundCol, lDotn, spec);
```

2. Let the first sphere (with index 0) be reflective. We extend the `trace()` function as follows: We check if the point of intersection is on a reflective sphere. If so, we recursively call the `trace()` function to get the colour value along the reflected ray and combine it with `colorSum`. The parameter `step` of the `trace` function defines the recursion depth. We have to make sure that this does not exceed the pre-specified maximum value `MAX_STEPS`:

```
//Generate reflection ray
if(q.index == 0   && step < MAX_STEPS)
{
  ...
  Color reflectionCol = trace(q.point, reflectionVector, step+1);
  colorSum.combineColor(reflectionCol, reflCoeff);
}
```

The origin of the reflected ray (Fig. c) is the point of intersection (`q.point`), and has a direction $\boldsymbol{r} = 2(\boldsymbol{n}\bullet\boldsymbol{v})\boldsymbol{n} - \boldsymbol{v}$. Remember to normalize the direction $\boldsymbol{r}$. Note also that the `Vector` class supports scalar post-multiplication of a vector, so the above computation must be implemented as $((\boldsymbol{n}*2)*(\boldsymbol{n}\bullet\boldsymbol{v})) - \boldsymbol{v}$.

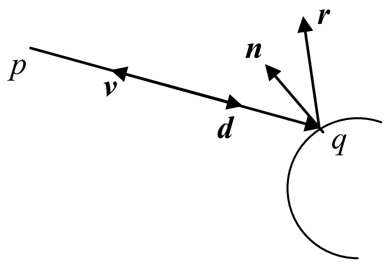


Fig. (c)

The colour I_r (`reflectionCol`) obtained along the reflection ray is combined with the colour I_A computed at the surface to produce the colour $I_A + \rho_r I_r$ where ρ_r is the coefficient of reflection (`reflCoeff`). ρ_r usually has a value between 0.5 and 0.8. The combined colour value can now be returned by `trace()`. An output of the ray tracer with a reflecting sphere is shown in Fig. (b).

3. A planar surface can be represented by a general linear equation in x, y, z or by a vector equation of the form $(\boldsymbol{p}-\boldsymbol{p}_1)\bullet\boldsymbol{n} = 0$ where $\boldsymbol{n}$ is the plane's normal vector and $\boldsymbol{p}_1$ is any point on the plane. However, these equations represent an infinite plane. For ray tracing applications, it is convenient to define a plane as a quadrilateral with vertices A, B, C, D. The `Plane` class is a subclass of `Object`, and has a constructor that takes five parameters: the four vertices and a colour value. A pointer to a plane object can be created as shown below:

```
Plane *plane = new Plane(Vector(-10, -10, -40), Vector(10, -10, -40),
  Vector(10., -10, -80), Vector(-10., -10, -80), Color(1, 0, 1));
```

It can then be added to sceneObjects:

```
sceneObjects.push_back(plane);
```

The vertices of the plane must be defined in an anti-clockwise sense with respect to the required normal direction. Remember to add the statement `#include "Plane.h"` at the beginning of the program. This header file and the implementation file (Plane.cpp) are provided. You will need to complete a set of functions in the implementation file (see below).

4. **Plane.cpp**

Being a subclass of `Object`, the `Plane` class must provide implementations for the functions `intersect()` and `normal()`. The surface normal vector $\boldsymbol{n}$ of the plane (Fig. d) can be computed as $(B - A)\times(C-A)$. Even though the normal of a plane is independent of the point at which it is computed, we need to use the standard signature of the normal function (`normal(pos)`) as specified in the `Object` class.

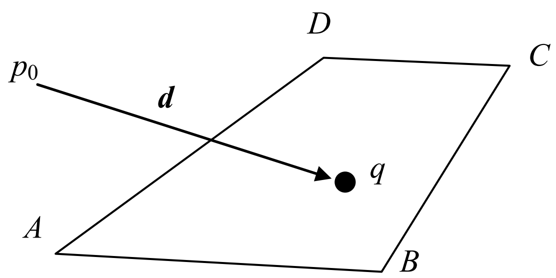


Fig. (d)

The value of the ray parameter t at the point of intersection is obtained as

$$t = \frac{(A - p_0)\cdot \boldsymbol{n}}{\boldsymbol{d.n}}$$

The `intersect()` function containing the above equation has already been implemented in the `Plane` class.

We also need to check if the point of intersection q ($= p_0 + td$) is within the bounds of the quadrilateral. The point inclusion test may be implemented in the `isInside()` function. From each vertex, define two vectors as below:

$\boldsymbol{u}_A = B-A$, $\quad \boldsymbol{v}_A = q-A$
$\boldsymbol{u}_B = C-B$, $\quad \boldsymbol{v}_B = q-B$
$\boldsymbol{u}_C = D-C$, $\quad \boldsymbol{v}_C = q-C$
$\boldsymbol{u}_D = A-D$, $\quad \boldsymbol{v}_D = q-D$

The point q is inside the quad if and only if $(\boldsymbol{u}_A \times \boldsymbol{v}_A)\bullet\boldsymbol{n}$, $(\boldsymbol{u}_B \times \boldsymbol{v}_B)\bullet\boldsymbol{n}$, $(\boldsymbol{u}_C \times \boldsymbol{v}_C)\bullet\boldsymbol{n}$, $(\boldsymbol{u}_D \times \boldsymbol{v}_D)\bullet\boldsymbol{n}$ are all positive.

Return the boolean value `true` only if the point is inside the quad, otherwise return `false`.

With correct implementations of the function definitions in the `Point` class, the program should produce an output similar to the one given below (Fig. e):

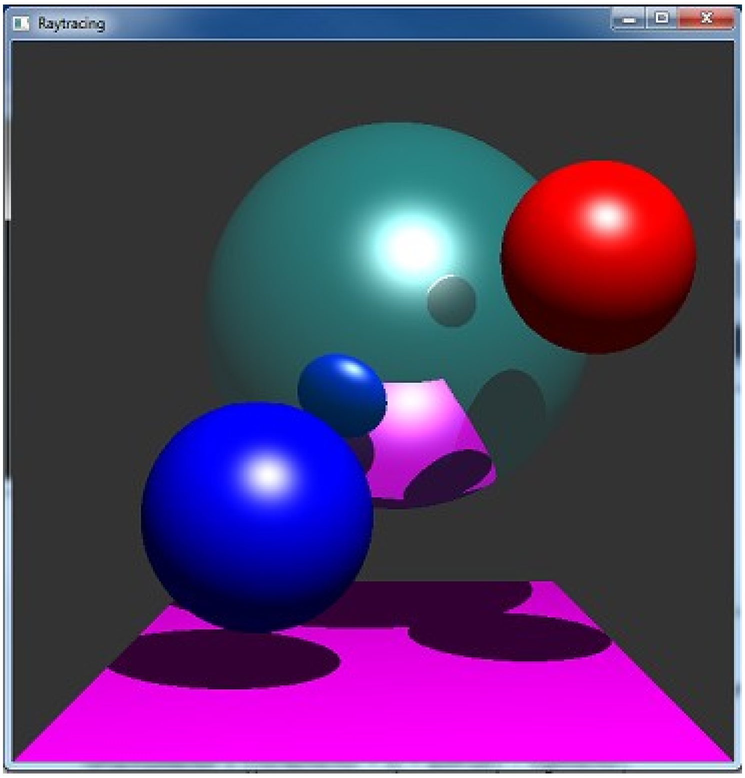


Fig. (e)

II. Quiz-08

The quiz will remain open until **5pm, 23-May-2014**.