COSC363 Computer Graphics
Lab08: Recursive Ray Tracing

Aim:

In this lab, you will extend the ray tracer developed in lab-07 to generate shadows and reflections using secondary rays at the points of intersection. You will also create functions for rendering planar surfaces.

I. RayTracer.cpp from Lab07

1. In Lab07, we implemented a basic ray tracer for a scene containing a set of spheres, including ambient, diffuse, and specular illumination. To add shadows to the scene, we require the creation of a shadow ray from the point of intersection towards the light source and the computation of the closest point of intersection of this ray with objects in the scene. (Fig. 1.). Add the following statements to the `trace()` function:

```
Ray shadow(ray.xpt, lightVector);
shadow.closestPt(sceneObjects);
```

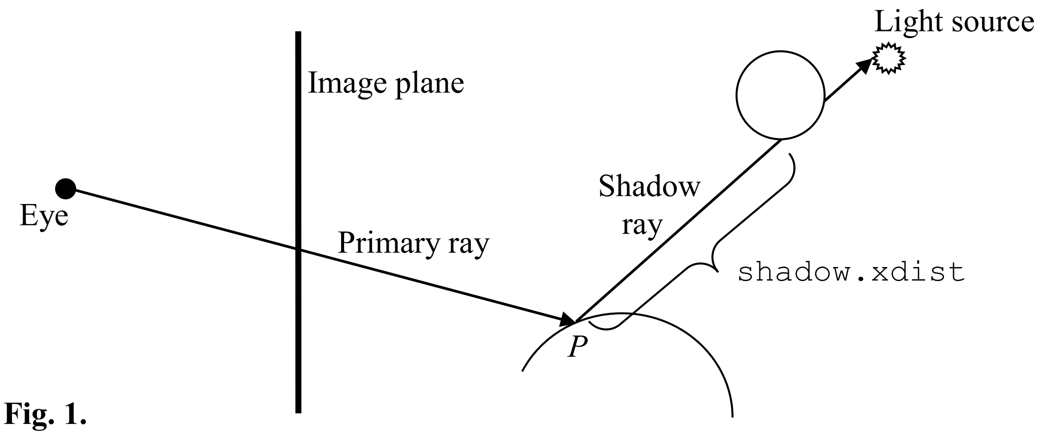

Fig. 1.

A point P is in shadow if either of the following conditions is satisfied:

- $\boldsymbol{l.n} \leq 0$ or,
- the shadow ray intersects an object (`shadow.xindex>-1`) and the distance from P to the closest point of intersection along the shadow ray is smaller than the distance to the light source (`shadow.xdist < lightDist`)

The `trace()` function should return the ambient illumination value at *P* (`ambientCol * materialCol`) if any of the above conditions is satisfied, otherwise it should return the sum of ambient, diffuse and specular components as previously implemented in last week's lab. The output of the ray tracer will now contain shadow regions (Fig. 2).

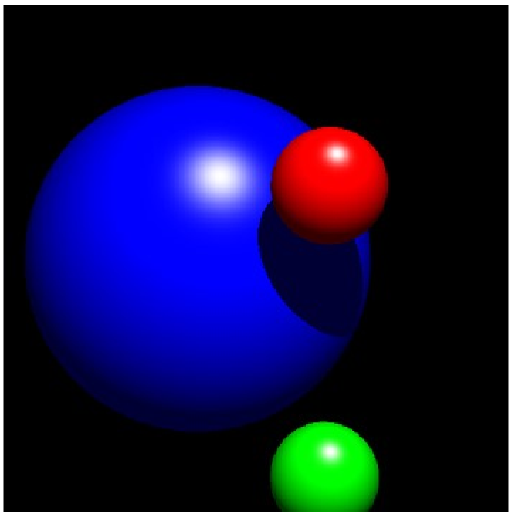
Fig. 2.

2. The generation of reflections and refractions of rays from a surface requires a recursive algorithm that accumulates colour values along secondary rays (Slide 13). We define a variable `glm::vec3 colorSum`, and use it to store the accumulated colour values. If the point of intersection on the primary ray is in shadow as outlined in the previous section, we initialize this variable with `ambientCol * materialCol` instead of returning the value. If the point is not in shadow, the sum of ambient, diffuse and specular compoenents is stored in `colorSum`.

   Next, we specify the index of the object that should be made reflective, and set the maximum limit (MAX_STEPS) for recursive ray tracing. In the code given below, we assume that the reflective sphere has index 0.

```
if(ray.xindex == 0   && step < MAX_STEPS)
{
    glm::vec3 reflectedDir = glm::reflect(ray.dir, normalVector);
    Ray reflectedRay(ray.xpt, reflectedDir);
    glm::vec3 reflectedCol = trace(reflectedRay, step+1);  //Recursion!
    colorSum = colorSum + (0.8f*reflectedCol);
 }
```

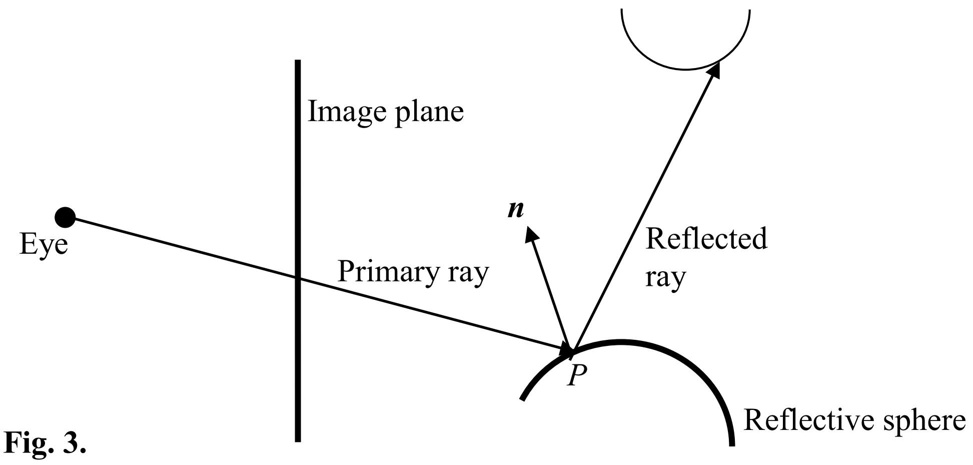

Fig. 3.

The first line inside the if-block above computes the direction of the reflected ray using the GLM function `reflect()` which takes the direction of the incident ray and the normal vector at the point as inputs. Note that this direction vector need not be normalized as it will have a unit length since both the incident ray's direction and the normal vector are unit vectors. The second line defines the reflected ray using its source (the point of intersection on the object) and the direction. The third line recursively invokes the `trace()` function using the reflected ray. The colour values are accumulated in the fourth line, where the coefficient of reflection is specified as 0.8. The trace() function should return this value (`colorSum`). A sample output with reflections is shown in Fig. 4.

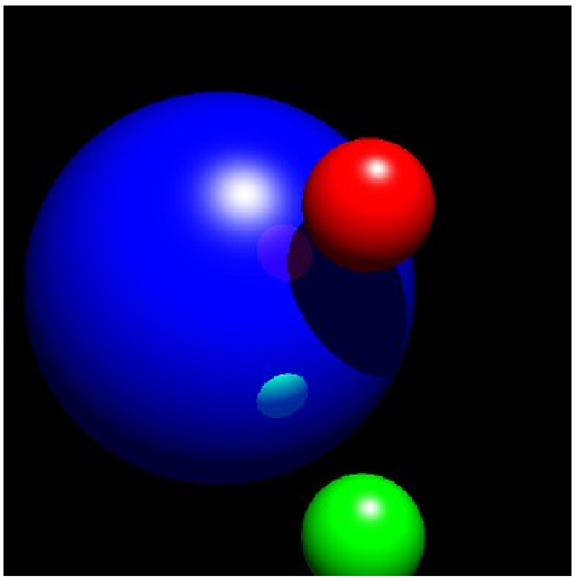

Fig. 4.

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 header file `Plane.h` and the implementation file

`Plane.cpp` are provided. You will need to complete a set of functions in the implementation file (see below).

Plane.cpp

The `Plane` class is a subclass of `SceneObject`, and has a constructor that takes five parameters: the four vertices and a colour value. Being a subclass of `SceneObject`, the `Plane` class must provide implementations for the functions `intersect()` and `normal()`. The surface normal vector ***n*** of the plane (Fig. 5) can be computed as $(B - A)\times(D-A)$. 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(p)`) as specified in the `SceneObject` class.

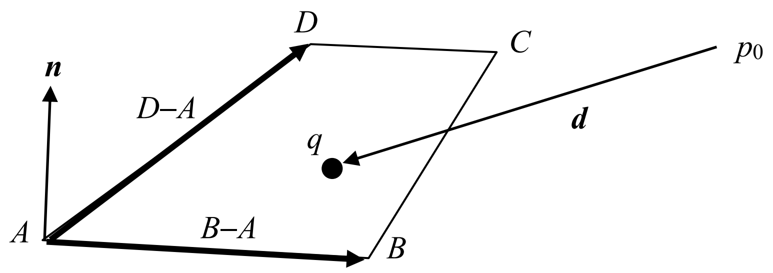

Fig. 5.

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

$$t = \frac{(A - p_0)\cdot \boldsymbol{n}}{\boldsymbol{d}.\boldsymbol{n}} \qquad \text{(Slide 27)}$$

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

3. Complete the `normal()` function in `Plane.cpp`. The class definition includes member variables `a, b, c, d` representing the four vertices of a quadrilateral. Using these variables, compute the vector cross product as given above, and normalize this vector.

In the initialize() function of the ray tracer, create a pointer to a plane object as shown below. The first four parameters define the vertices of the floor plane.

```
Plane *plane = new Plane (glm::vec3(-20., -20, -40),    //Point A
                          glm::vec3(20., -20, -40),     //Point B
                          glm::vec3(20., -20, -200),    //Point C
                          glm::vec3(-20., -20, -200),    //Point D
                              glm::vec3(0.5, 0.5, 0));   //Colour
```

and add this to the list of 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. You should get an output similar to that shown in Fig. 6:

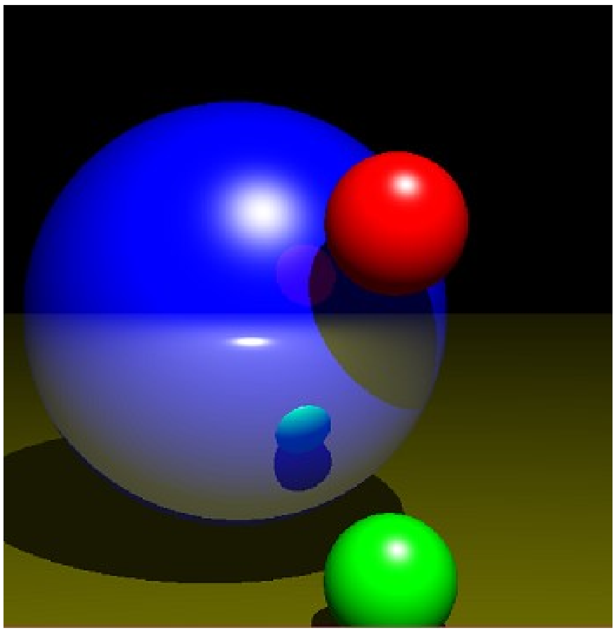
Fig. 6.

4. Please note that the plane shown in the above figure is an infinite plane. You need to complete the `isInside()` function in `Plane.cpp` to check if the point of intersection is within the quadrilateral specified by the four points.

   Let p be the point to be tested. This point is passed to the function as a parameter. From each vertex, define two vectors as below (see Slide 29):

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

   The point p 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 in Fig. 7.

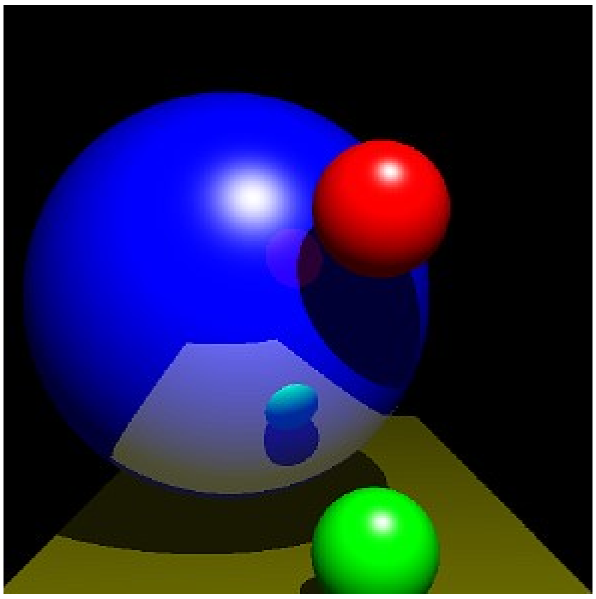
Fig. 7.

II. Quiz-08

The quiz will remain open until **5pm, 17-May-2019**.