COSC363 Computer Graphics

Lab07: Ray Tracing

Aim:

This lab aims to provide an introduction to the fundamental concepts and methods used in ray tracing.

I. RayTracer.cpp

Ray tracing is an advanced computer graphics rendering paradigm using which a variety of effects such as complex shadows, realistic reflections, refractions through glass etc., can be generated with the help of a global illumination model. In this lab, we will explore the basic structure of a ray tracer and create a simple scene consisting of a set of spheres. In next week's lab we will add planar surfaces, reflections and textures to the generated scene.

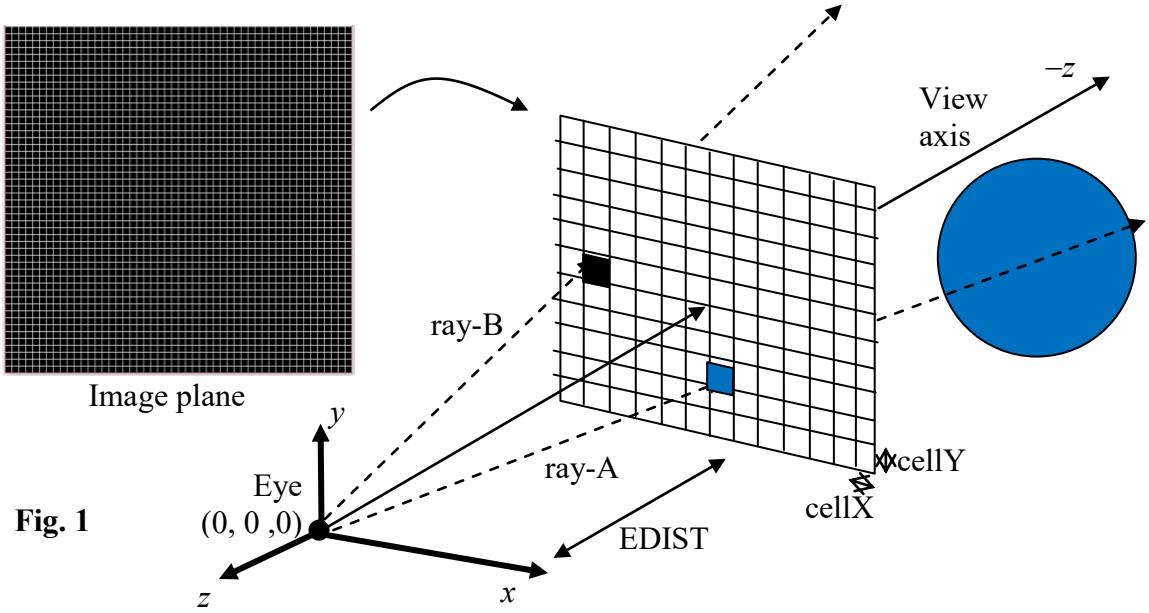


Fig. 1

A ray tracing algorithm uses a simple camera model (Fig. 1) with the origin specified as the eye position, and the view axis along the $-z$ direction. The image plane consists of a regular grid of cells.

At the start of the program, a few constant variables are initialized:

`WIDTH, HEIGHT`: Width, height of the image plane in world units
`EDIST`: The distance of the image plane from the camera/origin.
`NUMDIV`: The number of cells (subdivisions of the image plane) along x and y directions.
`MAX_STEPS`: The number of levels (depth) of recursion (to be used next week)
`XMIN, XMAX, YMIN, YMAX`: The boundary values of the image plane defined in terms of `WIDTH` and `HEIGHT`, such that the view axis passes through its centre.

Let us now turn our attention to the **`display()`** function. Here you will find the code that draws each cell as a quad. The cell width (`cellX`) and the cell height (`cellY`) are calculated using the constants defined above. A primary ray is generated from the origin (eye) through the centre of each cell (Fig. 2).

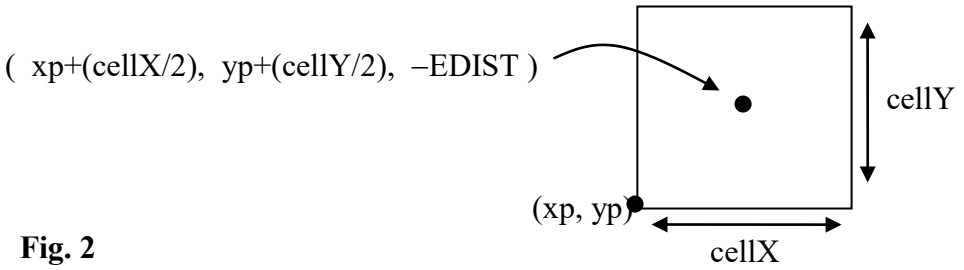


Fig. 2

A ray object `ray` is constructed using its source point and direction. For a primary ray, the construction of a ray object is done as follows:

```
Ray  ray = Ray(eye, dir);
```

The nested for-loop in the `display()` function traverses each cell of the image plane and generates a primary ray through that cell. This colour value obtained by tracing a ray is used to draw the corresponding cell. If the ray hits a blue sphere, the cell colour will be blue. If the ray does not hit any object in the scene, the cell colour will be black (see Fig. 1).

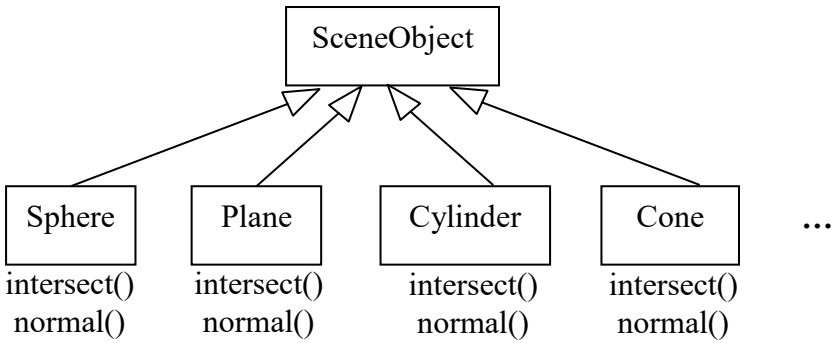
The heart of a ray tracing application is the **`trace()`** function. In the given program, it is included in its simplest form. The function takes a ray as input and calls the function `closestPt()` which computes the closest point of intersection of the ray with the scene objects. If the closest point of intersection is found, `ray.index` will contain the index of the object on which the point lies. If the ray does not intersect any object, this variable will have a value –1. The `trace()` function returns the colour of this object.

The header and implementation files of the following classes are given:

SceneObject

This is an abstract class that represents objects in the scene. This provides a generic type for different types of scene objects such as spheres, planes, cylinders etc. Using this common type, all scene objects can be stored in a common container (e.g., a list) and processed in a sequence using an iterator. Each object type such as "Sphere", "Plane" etc. is defined as a subclass of this class (Fig. 3). The `SceneObject` class stores common properties of objects such as material colour, reflectivity, refractivity, specularity, shininess, coefficient of reflection, refractive index, etc. It also has setter/getter/query methods for accessing these attributes. The subclasses (e.g. `Sphere`) inherit these properties, and also implement geometry specific functions for computing normal vectors and intersection points.

Fig. 3



Sphere

The "Sphere" class is a subclass of "SceneObject" 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);
```

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 "SceneObject" 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. E.g. A green sphere with radius 20 units at (0, 5, -100) is created as below:

```
SceneObject *s = new Sphere(glm::vec3(0, 5, -100), 20);
s->setColor(glm::vec3(0, 1, 0));
```

Ray

A ray can be represented using its source point p_0 and direction d. The source and the direction of a `ray` can be retrieved using the public member variables, `ray.p0` and `ray.dir` respectively.

The only function in the Ray class is `closestPt()` which takes a list of scene objects as inputs and updates the values of the following member variables:
`ray.hit`: The closest point of intersection on the ray
`ray.index`: The index of the object on which the closest point of intersection lies.
`ray.dist`: The distance of the closest point of intersection from the source of the ray.

1. Compile and run the program "RayTracer.cpp". You will get a blank screen! The scene currently does not contain any objects. Inside the `initialize()` function, uncomment the three statements: the first statement creates a pointer to a sphere with centre at (–5, 0, –90) and radius 15 units, the second statement sets the sphere's material colour to blue, and the third statement adds the sphere object's reference to the list of scene objects.

Note that the image plane is positioned at z = –EDIST = –40, and therefore all scene objects must have a z value less than –40. The program now generates the display given in Fig.4. Change the value of NUMDIV to 500. You will notice that the program takes a longer time to run due to the generation of a much larger number of rays, producing the output shown in Fig. 5.

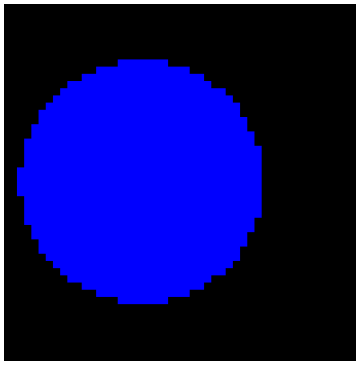


Fig. 4

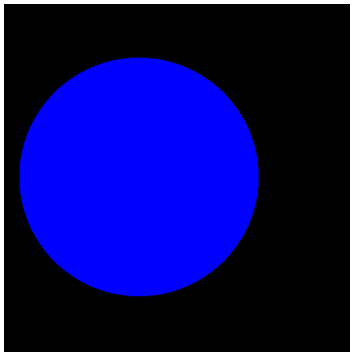


Fig. 5

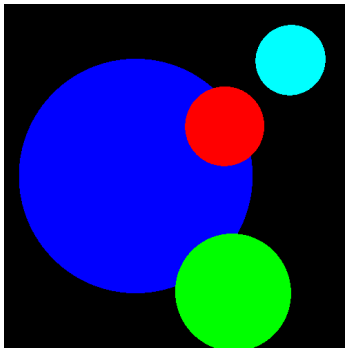


Fig. 6

Create a few more spheres, all located on the correct side of the image plane ($z <$ –EDIST), and add them to the array of scene objects (Fig. 6). The first sphere added to the array will have index 0, the next sphere index 1 and so on. The basic version of the ray tracer that uses only the material colours of objects producing outputs as shown in Fig. 6, is called a ray casting method.

Fig. 7 shows the configuration of the four spheres used as an example in this note.

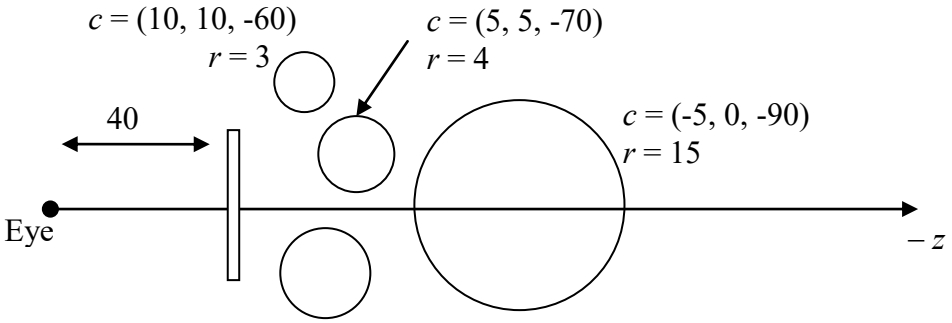


Fig. 7

2. We will now modify the `trace()` function to return the colour values computing using the Phong's illumination model. For this, we require the position of a light source, which is already defined in the function. The lighting computation is done by the `lighting()` function of the `SceneObject` class. The function takes three parameters: the light's position, the view vector and the point of intersection where the colour value needs to be computed.

   Replace the function call `obj->getColor(..)` with `obj->lighting(..)`. The direction of the view vector (the vector from the point of intersection to the eye position) is given by `-ray.dir` (verify this!). The point of intersection is given by `ray.hit`. You should now get an output similar to that given in Fig. 8.

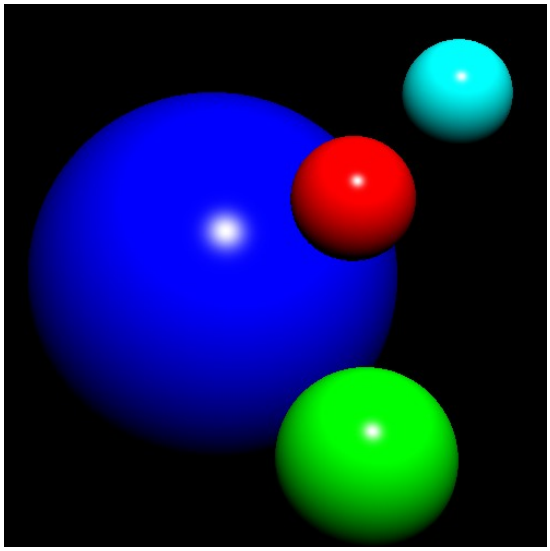


Fig. 8.

3. Try altering the properties of the spheres in the `intitalize()` function. For example, specular reflections on a sphere can be suppressed using

   ```
   sphere->setSpecularity(false);
   ```

   The default value for "shininess" is 50. Try reducing it to 5 for a sphere using

   ```
   sphere->setShininess(5);
   ```

   (Note the variable `sphere` above should be replaced with a variable name you have used in the program. e.g. `sphere3`)

4. Let us now add shadows to the scene. Include the following statements in the `trace()` function, after the statement containing the function call `obj->lighting()`.

   ```
   glm::vec3 lightVec = lightPos - ray.hit;
   Ray shadowRay(ray.hit, lightVec);
   ```

   The statements create a shadow ray at the point of intersection with a direction given by `lightVec` which is a vector from the point of intersection to the light source. Find the closest point of intersection on the shadow ray:

   ```
   shadowRay.closestPt(sceneObjects);
   ```

   If the shadow ray hits an object (`shadowRay.index > -1`) and the distance to the point of intersection on this object is smaller than the distance to the light source (`shadowRay.dist < lightDist`), reset the colour value at the point of intersection to the ambient colour:

   ```
   color = 0.2f * obj->getColor();   //0.2 = ambient scale factor
   ```

(Note: The distance to the light source `lightDist` can be computed as the length of the light vector `lightVec` using `glm::length()` function)

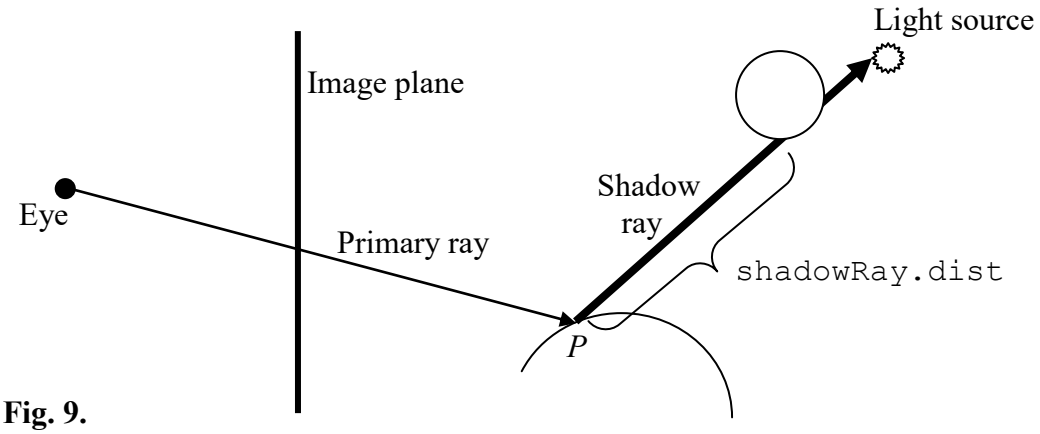


Fig. 9.

You should now get an output similar to the one attached below, containing shadows.

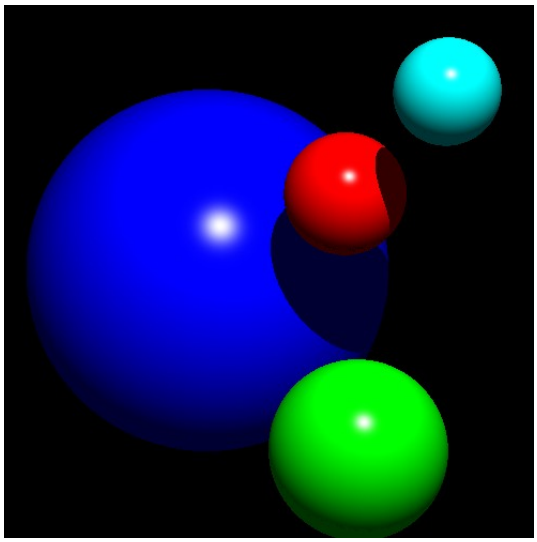


Fig. 10

5. Please go through the code in `Sphere.cpp` and familiarize with the methods used for computing normal vectors and points of intersection. Please compare the implementations with the equations given on Slide 28.

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

II. Quiz-07

The quiz will remain open until **5pm, 8-May-2020**.