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 shadows and reflections to the generated scene.

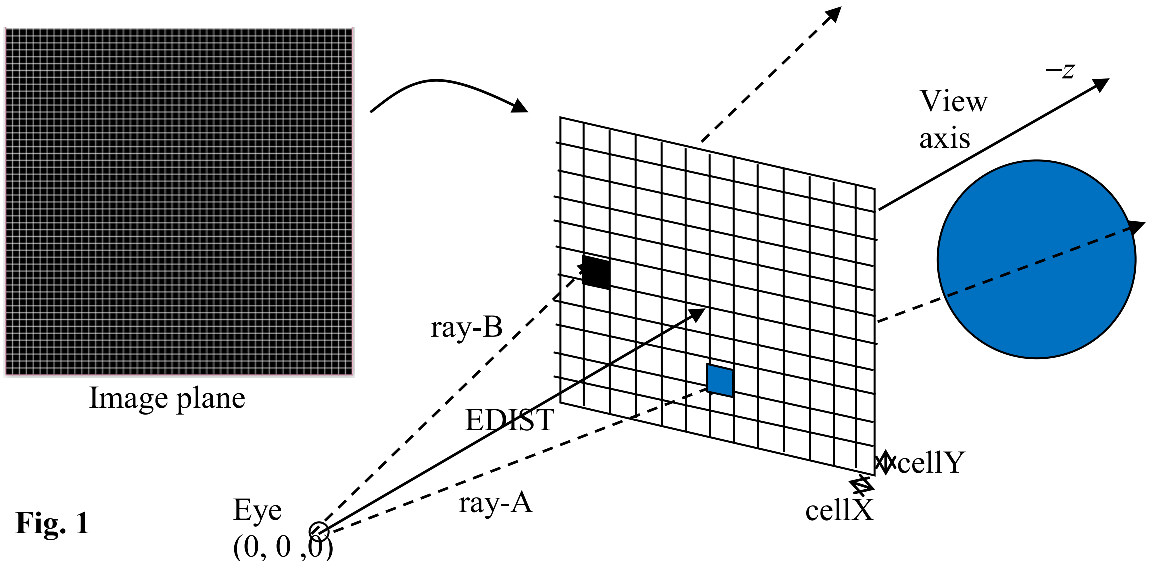

Fig. 1

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 values are defined:
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 such that the view axis passes through it 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 ray is generated from the origin through the centre of each cell (Fig. 2).

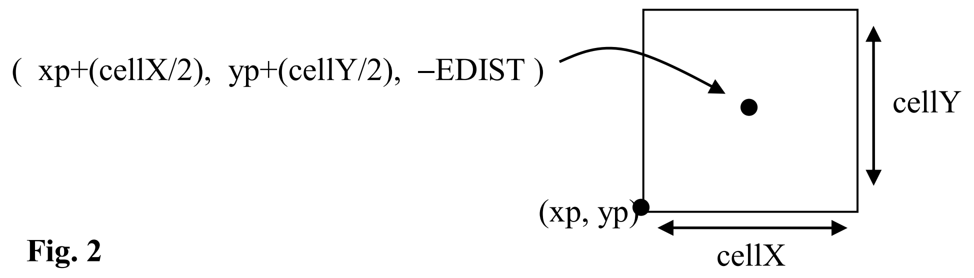

Fig. 2

The `trace()` function returns a colour value along a ray. This value 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. The ray class has the following member variables: (i) `xpt`: The closest point of intersection, if found. (ii) `xindex`: This is the index of the scene object on which the ray intersection point was found. If the ray does not intersect any object, this variable will have a value -1 (iii) `xdist:` The distance of the point of intersection from the origin of the ray. The `trace()` function returns the background colour if `xindex` has a value `-1,` otherwise returns the objects colour.

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 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 "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, 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 "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.

```
SceneObject *s = new Sphere(centre, radius, color);
s->setColor(newColor);   //not s.setColor(..)
```

Ray

A ray can be represented using its source point p_0 and direction d. After creating a ray object, always remember to normalize the ray direction using the normalize() function. See code in the display() function for examples of ray creation and normalization.

1. Compile and run the program "RayTracer.cpp". You will get a blank screen! This is because the scene currently does not contain any objects. Inside the initialize() function, uncomment the two statements: the first statement creates a pointer to a blue sphere with centre at (-5, -5, -90) and radius 15 units, and the second statement adds this pointer 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.3. 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. 4.

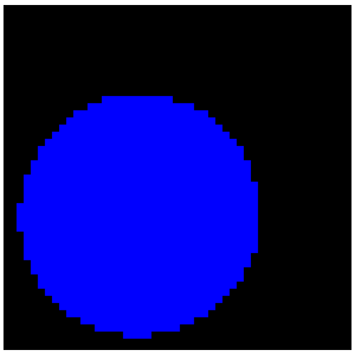
Fig. 3.

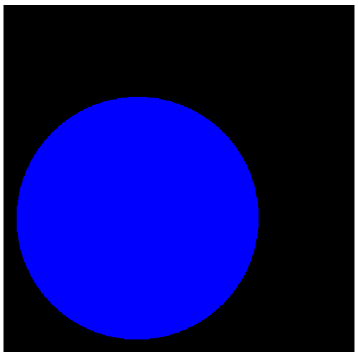
Fig. 4.

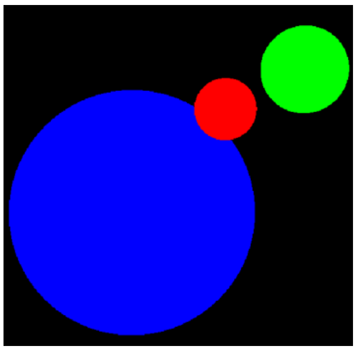
Fig. 5

   Create a few more spheres like this, all located on the correct side of the image plane (z < –EDIST), and add them to the array of scene objects (Fig. 5). The first sphere added to the array will have index 0, the next sphere index 1 and so on.

2. 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 position of a light source, which is already defined in the function. We also require the normal vector on the sphere at the point of intersection, which can be obtained as

```
glm::vec3 normalVector = sceneObjects[ray.xindex]->normal(ray.xpt);
```

   The light vector (the vector from the point of intersection towards the light source) can be defined as

```
glm::vec3 lightVector = light - ray.xpt;
```

Convert the light vector to a unit vector using the normalize() function of the GLM library, and compute the dot product of the vector and the normal vector. This gives the term ***l.n.*** Use the variable name `lDotn` for this term. If the value of this variable is negative, the function should return only the ambient component (=ambientCol*materialCol), otherwise it should return the sum of ambient and diffuse colours (`ambientCol*materialCol + lDotn*materialCol`). Note that the product u*v where both u, v are vectors of type glm::vec3, represents component-wise multiplication of the vectors u, v.
The output of the program is shown in Fig. 6. Note that a ray traced output gives an accurate rendering of the object's shape because the method uses exact mathematical equations and not polygonal representations.

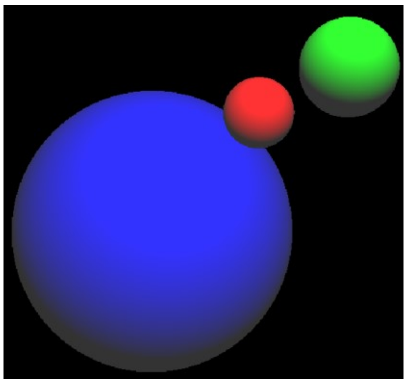
Fig. 6.

3. For computing specular reflections, we require the reflection vector ***r***: The GLM library has the function `reflect()` which could be used for this purpose:

```
glm::vec3 reflVector = glm::reflect(-lightVector, normalVector);
```

The first parameter of the reflect() function is the incident light's direction (unit vector from light source to the point of intersection), which is a vector in the direction opposite to `lightVector`.

Compute the specular term $(\boldsymbol{r}.\boldsymbol{v})^f$, where f is the Phong's constant (shininess). If $\boldsymbol{r}.\boldsymbol{v} < 0$, set the specular colour to 0, otherwise the specular reflection is given by $(\boldsymbol{r}.\boldsymbol{v})^f$ (1, 1, 1). Add this colour value to the ambient and diffuse components computed earlier. You should now get an output similar to that shown in Fig. 7.

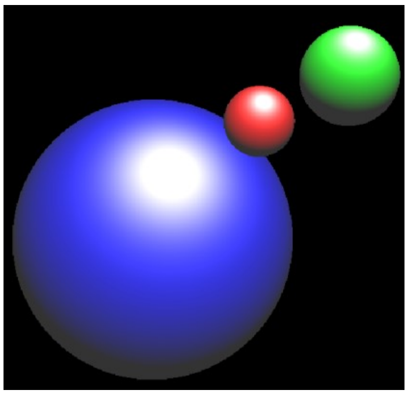
Fig. 7.

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

II. Quiz-07

The quiz will remain open until **5pm, 10-May-2019**.