

# CSE 410 – July 2018

## Assignment 4: Ray Tracing

### Introduction:

In this assignment, you will develop a ray tracer image generator.

### Input description:

The input file “description.txt” contains the explanation at the bottom of the file.

### Objects:

There are three types of objects:

- An infinite checkerboard
- Pyramids
- Sphere

### Part One: Develop an OpenGL interface

In this interface, you will draw all the objects according to their positions and colors. In this part, we will not consider any of the ambient, diffuse, specular or reflection coefficients. For the camera, you have to use the camera assignment from Assignment 1 with all possible controls. Implement the camera with  $l$ ,  $r$  and  $u$  vectors, as before. The camera controls must be the same as Assignment 1.

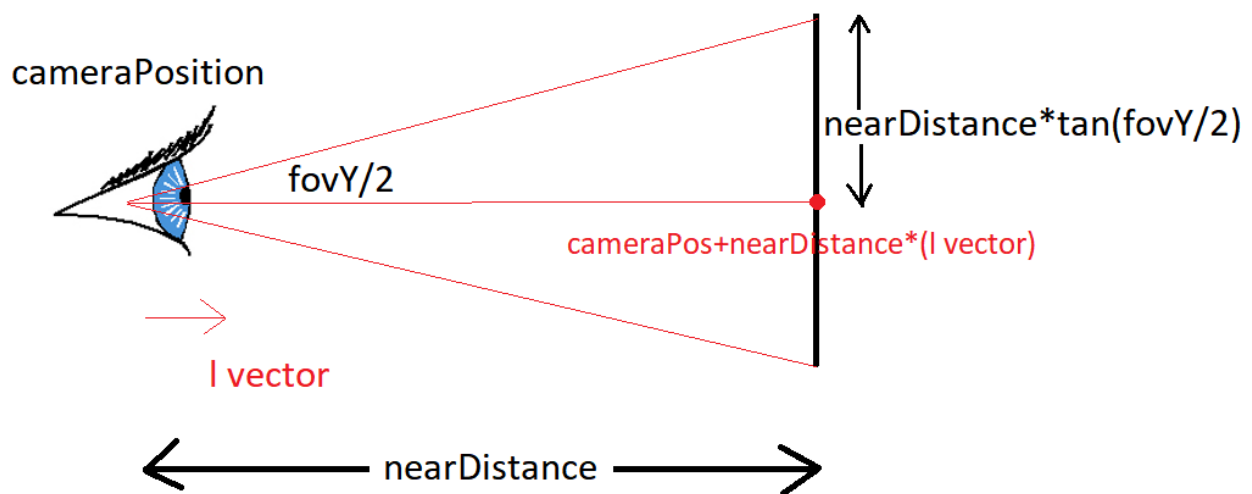
### Part Two: Implement the Ray Tracer

When the key ‘0’ is hit, your program will perform ray tracing and output a bmp image.

## Ray Generation:

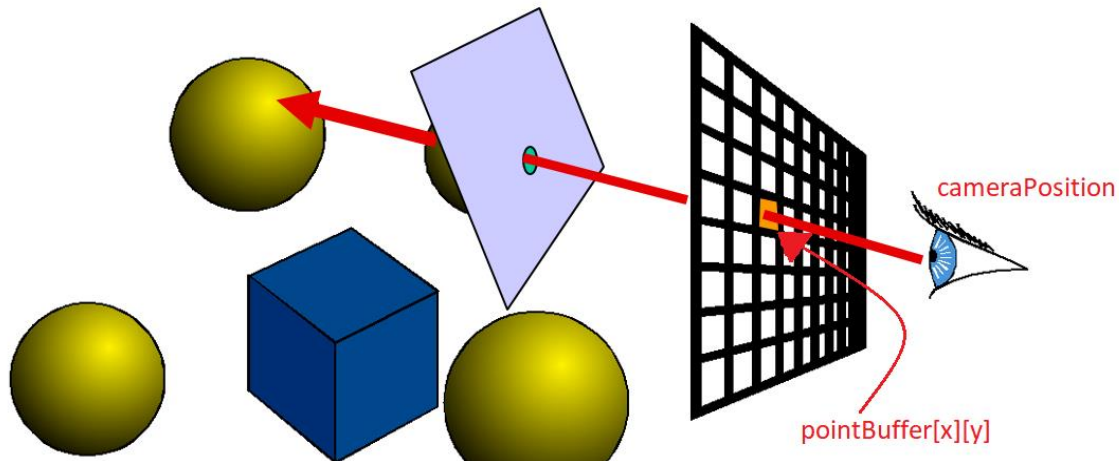
When the key is pressed, the position of the camera and its l, r, u vectors will define the image. This is why

- We are not taking camera position from input file
- It is important that you implement camera using l, r, u



Now, the image that you will see has a dimension screenSize X screenSize. The objects in the OpenGL world will be mapped on a screen with height =  $2 * \text{nearDistance} * \tan(\text{fovY}/2)$  and width =  $2 * \text{nearDistance} * \tan(\text{fovX}/2)$ ; where fovX and fovY are in angles and  $\text{fovX} = \text{aspectRatio} * \text{fovY}$ . Also, the middle point of that screen is midpoint =  $\text{cameraPosition} + (\text{l vector}) * \text{nearDistance}$ .

Now, if you can calculate the midpoint, then you can use the r\_vector and u\_vector to generate various 3D points on the screen. Do this part yourself. Calculate all the 3D points of the OpenGL world that map to a pixel of your image. Store those points in a 2D array. Let us call this array pointBuffer.



Therefore, we have two points on the ray,  $\text{pointBuffer}[x][y]$  and  $\text{cameraPosition}$ . The direction vector of the ray is  $\text{pointBuffer}[x][y] - \text{cameraPosition}$ .

### Ray and object intersections:

Now that we can generate the rays, we have to calculate the intersections of rays with objects. Follow the class lectures and slides for the mathematical formulas for intersections.

A key thing to remember is: do not start the rays from  $\text{cameraPosition}$ . Rather, start them from  $\text{pointBuffer}[x][y]$ . If you start from camera position, you will have to do extra checks to implement the near plane. But, if you start the ray from the  $\text{pointBuffer}[x][y]$ , then you only have to check that the parameter,  $t$  has value  $t > 0$ .

### Color of a point:

When the ray you casted intersects a point on an object, you have to calculate the shade (color) at that point. Let us assume that the intersecting point be  $P$ . On  $P$ , we have to calculate the color for this object.

There are four color components for an object: ambient, diffuse, specular and reflectance. Let the coefficients for these four be  $a$ ,  $d$ ,  $s$ ,  $r$ ; respectively. The input will be given ensuring  $a + d + s + r = 1.0$ .

Now, the object itself has a color specification (that we get from the input file). Let that be red, green, blue. Each of these are floating point values between 0 and 1.

### Ambient:

The ambient color is calculated easily:  $a \cdot \text{red}$ ,  $a \cdot \text{green}$ ,  $a \cdot \text{blue}$

## Diffuse & Specular:

Diffuse component is the result of a light ray coming straight from the light source. Hence, we first check if any light ray comes from a source S to the intersecting point P. To do this, we generate a ray from P to S, and check if it intersects any other object before S. If it does, then the light source S does not make any contribution to the color at the point P.

If no other object intersects the ray before S, then the source S illuminates intersecting point P. So we have to calculate both diffuse and specular components for that source. First, we calculate the normal at the intersecting point. Let the normal be N.

`lambert = phong = 0`

For all sources S:

```
    if S does not illuminate intersecting point P: continue
    vector toSource = PS;
    toSource.normalize();
    N = normal at intersecting point P
    N.normalize();
    lambert += toSource.dot(N);
    R' = reflected ray at point P (use N here as well)
    R'.normalize()
    phong += pow( R'.dot(N), shininess)
```

Diffuse component is found as: `d*lambert *red, d*lambert*green, d* lambert*blue`

Specular component is found as: `d* phong *red, d* phong *green, d* phong *blue`

## Reflection:

You already calculated reflected ray R', right? Now pass that ray as if it is an original ray casted from the camera. After processing it, you will get a color. The reflection portion is calculated as: `r*reflected_red, r*reflected_green, r*reflected_blue`.

Therefore, the color at a point is:

```
a*red + d*lambert *red + d* phong *red + r*reflected_red
a*green + d*lambert * green + d* phong * green + r*reflected_green
a*blue + d*lambert * blue + d* phong * blue + r*reflected_blue
```

Hint: write a recursive function that takes a ray and returns a color by using the above mentioned formulas. The function will also receive an argument: `depthOfRecursion`. Initially you call it with 3/4/5, as specified in input file. Whenever you send a reflected ray, you call the same function with the parameters: `reflectedRay` and `depthOfRecursion-1`. When the depth reaches 0, return black color [0 0 0].

## Careful:

- When sending a reflected ray from intersecting point P, do not start it at P. Rather, advance it a little bit. Otherwise the ray may generate intersection at the same object.
- All calculations in this document assume that the color values are in range [0,1]. Hence, always check if a color is exceeding 1 or being less than 0.
- The bmp images will contain values in range [0..255]. So, before writing to image, take care of this. A simple multiplication with 255 should suffice.

## Deadline:

26 January (Saturday), 11:55 AM.

## Plagiarism:

CSE, BUET stores the source code for all previous year submissions as well. Historically, few students were convicted of copying alumni's code. Be careful.

## Marking:

Submission:	10
OpenGL interface:	5
Checker board:	10
Sphere:	10
Pyramid:	30
Ambient color:	5
Diffuse and specular color:	15
Reflection:	15

## Bonus: Texture (20 marks)

Pressing <space> enables texture. If you press '0', the texture.bmp will appear on the checkerboard.

To implement texture, you first load the bmp file "texture.bmp" into a 2D array. Then, while calculating the color of a point on the infinite checkerboard, you have to determine which pixel of the texture image maps to that point on the checkerboard. Let the colors of the texture image of that pixel be [R G B]. These values will be integers in range [0..255]. You first have to scale them to the interval [0,1]. Then, you calculate the color of that point on the checkerboard as:

$R(a \cdot \text{red} + d \cdot \text{lambert} \cdot \text{red}) + r \cdot \text{reflected\_red}$   
 $G(a \cdot \text{green} + d \cdot \text{lambert} \cdot \text{green}) + r \cdot \text{reflected\_green}$   
 $B(a \cdot \text{blue} + d \cdot \text{lambert} \cdot \text{blue}) + r \cdot \text{reflected\_blue}$

Note that the checker board has no specular component. This is why there are only ambient and diffuse components. The reflected part will not be affected by texture.