January 2023 CSE 410 - Computer Graphics Sessional
Offline – 3
Assignment on Ray Tracing

Introduction:

In this assignment, you have to generate realistic images for a few geometric shapes using ray tracing with appropriate illumination techniques.

Input description:

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

Objects:

There are four types of objects:

- An infinite checkerboard
- Pyramids
- Spheres
- Cubes

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.

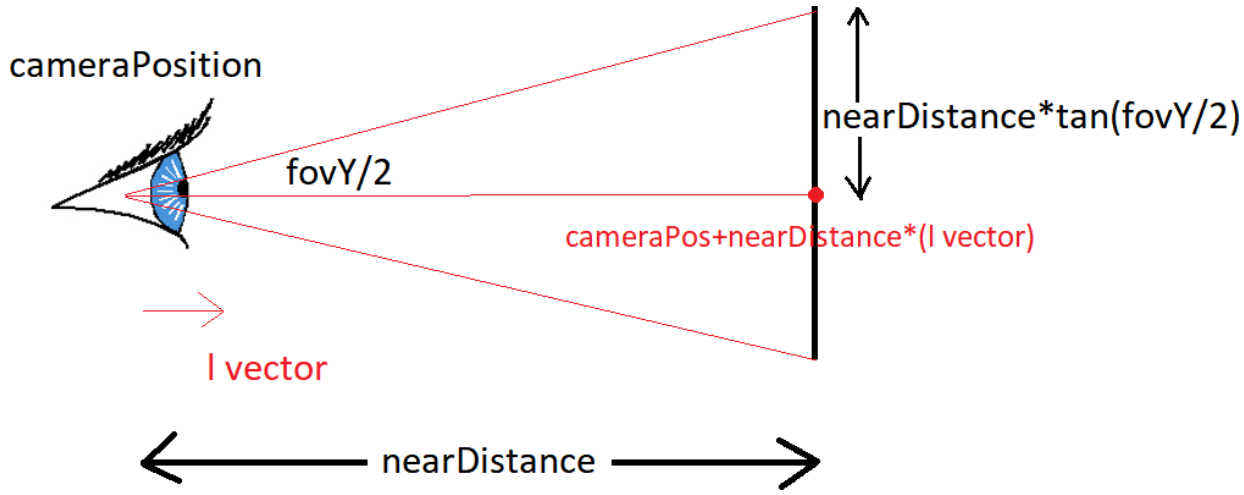
Part Two: Implement the Ray Tracer

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

Ray Generation:

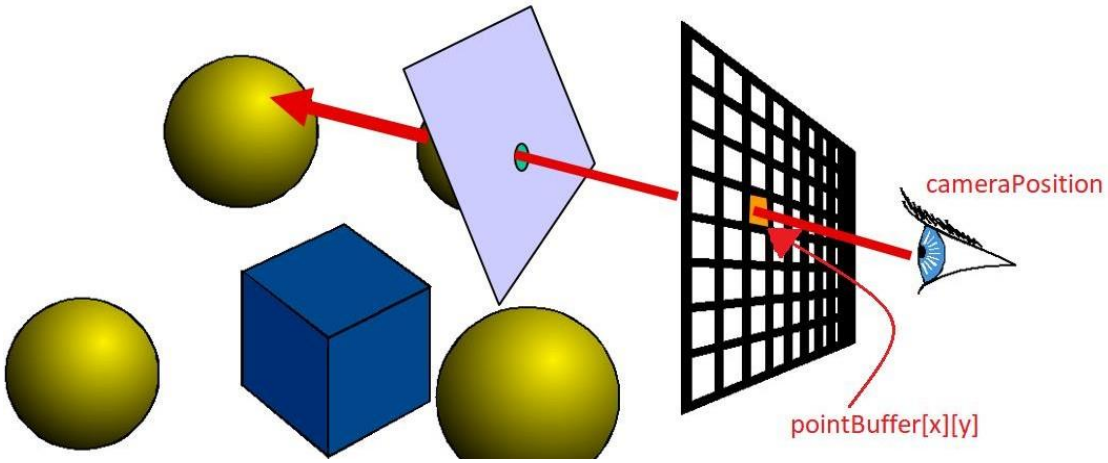
When the key is pressed, the position of the camera and its l, r, u vectors will define the image. This is why

a. We are not taking camera position from input file
b. 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*nearDistance*tan(fovY/2) and width = 2*nearDistance*tan(fovX/2); where fovX and fovY are in angles and fovX = aspecRatio*fovY. Also, the middle point of that screen is midpoint = cameraPosition+(l vector)*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, pointBuffer[x][y] and cameraPosition. The direction vector ofthe ray is pointBuffer[x][y] – cameraPosition.

Ray and object intersections:

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

A key thing to remember is: do not start the rays from cameraPosition. Rather, start them from pointBuffer[x][y]. If you start from camera position, you will have to do extra checks to implement thenear plane. But, if you start the ray from the 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 thecoefficients 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*red, a*greed, a*blue

Diffuse & Specular:

Diffuse component is the result of a light ray coming straight from the light source. Hence, we first checkif any light ray comes from a source S to the intersecting point P. To do this, we generate a ray from P toS, 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 atthe 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();
      distance = distance between intersecting
      point P and source S scaling_factor =
      exp(-distance*distance*S.falloff);
      lambert += toSource.dot(N)*scaling_factor;
      R' = reflected ray at point P (use N here as well) [here,
       original ray is starting from camera, not from source]
      R'.normalize()
      phong += pow( R'.dot(toSource), shininess)*scaling_factor
```

Diffuse component is found as: *d*lambert *red, d*lambert*green, d* lambert*blue*
Specular component is found as: *s*phong *red, s* phong *green, s* phong *blue*

Determining if a light source S illuminates a point P:

Send a ray starting from P to PS direction. If that ray does not intersect anything before reaching thesource, then the source S illuminates P.

For a normal light source, this check will suffice. For a spotlight, we also have to check that the angle does not exceed the cut off angle. For example, say a source S has cut-off angle of 20 degrees. Now, thespotlight source will have a direction vector as well – the direction along which it is casting light. If the vector SP makes an angle of 22 degrees with the direction, then S will not illuminate P.

```
V1 = determine vector SP
V1.normalize()
V2 = direction of the source S
V2.normalize()
angle = acos(v1.dot(v2))
if angle > S.cutoff:
      P is not illuminated
```

Reflection:

You already calculated reflected ray R', right? Now pass that ray as if it is an original ray cast 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].

Bonus:

Pressing <space> enables texture. If you press '0', the "texture_w.bmp" and "texture_b.bmp" will appear on the white and black squares of the checkerboard respectively.
To implement texture, you first load the bmp files into two 2D arrays. 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*red + d*lambert *red) + r*reflected_red*
G(a*green + d*lambert * green) + r*reflected_ green*
B(a*blue + d*lambert * blue) + r*reflected_ blue*

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

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 the range [0,1]. Hence, always check if a color is exceeding 1 or being less than 0.
- The bmp images will contain values in the range [0.255]. So, before writing to image, take care of this. A simple multiplication with 255 should suffice.

Marks Distribution:

- File I/O, Camera control, Memory management, Drawing in OpenGL etc. - 20%
- Ray-object intersection (ray casting) - 35%
- Recursive reflection (ray tracing) - 15%
- Illumination - 25%
- Submission - 5%
- Bonus - 20%

Submission Guidelines:

1. Create a folder having the same name as your 7-digit student id. If your student id is 1805xxx, then the name of the folder will be 1805xxx.
2. Rename all your source files so that they have your student id as prefix (e.g., 1805XXX_Main.cpp, 1805XXX_Header.h etc.).
3. Put the source files in the folder created in step 1 and zip the folder.
4. Upload the zip file (1805XXX.zip) on Moodle.

Submission Deadline:

11th week, Friday, 11:45 PM (No Extension)

Expected date of 11th week, Friday: September 1, 2023.