EECS3431, Fall 2023, Assignment 3

RAY-TRACING

Collaboration: **In groups of maximum two**. If you discuss this assignment with others you should submit their names along with the assignment material.

For this assignment, you will be building a Ray Tracer using C/C++, or Java. The system only needs to handle the rendering of ellipsoids, with a fixed camera situated at the origin in a right handed coordinate system, looking down the negative z-axis. Local illumination, reflections, and shadows will also need to be implemented.

The program should take a single argument, which is the name of the file to be parsed. Make sure your executable has the name "raytracer.exe" or "raytracer.java", and that we can run it as in the following example:

```
> raytracer.{exe,java}  testCase1.txt
```

We will use a script to generate the outputs for the set of posted test cases. You will get zero marks if we cannot compile your program, or if we cannot run this script because your project does not implement the required specifications.

INPUT FILE

The content and syntax of the file is as follows:

Content:

a. The near plane**, left**, right**, top**, and bottom**
b. The resolution of the image nColumns* X nRows*
c. The position** and scaling** (non-uniform), color***, K_a***, K_d***, K_s***, K_r*** and the specular exponent n* of a sphere
d. The position** and intensity*** of a point light source
e. The background color***
f. The scene's ambient intensity***
g. The output file name (you should limit this to 20 characters **with no spaces**)

** int* *** float* **** float between 0 and 1*

Syntax:

```
NEAR <n>
LEFT <l>
RIGHT <r>
BOTTOM <b>
T OP <t>
RES <x> <y>
SPHERE <name> <pos x> <pos y> <pos z> <scl x> <scl y> <scl z> <r> <g> <b> <Ka> <Kd> <Ks> <Kr> <n>
… // up to 14 additional sphere specifications
```

LIGHT <name> <pos x> <pos y> <pos z> <I_r> <I_g> <I_b>
… // up to 9 additional light specifications
BACK <r> <g > <b>
AMBIENT <I_r> <I_g> <I_b>
OUTPUT <name>

All names should be limited to 20 characters, with **no spaces**. All fields are separated by spaces. There will be no angle brackets in the input file. The ones above are used to indicate the fields.

MARKING SCHEME:

- [2] Coding Style (i.e. well designed clean commented code)
- [2] x 11 For each of the given test cases.
- There will be no partial marks given if your program fails to parse the input file or if it does not produce the correct output.
- **Make sure you submit all the required files so we can compile and build your program. If there are missing libraries you will get zero marks.**
- **We will use a script to generate the outputs for a set of test cases. You will get zero marks if we cannot run this script because your program does not adhere to given requirements. (Repeated for emphasis).**
- **[-1 Penalty if not included] Submit a readme.txt that states which cases work properly, and anything that requires clarification.**

INSTRUCTIONS AND CLARIFICATIONS:

- ***Start working on it early. You will not have time to do it at the last minute.***
- Submit your assignment via submit (as a zipped project file). . Your submission should include ALL of the code necessary to compile and run the program, and should not contain any additional functionality besides what is described below. It should not contain any xxGL API calls. You may use xxGL for displaying the results during your debugging.
- You may use the vector and matrix libraries from here https://github.com/g-truc/glm , or any other such libraries. However, you have to code the ray-sphere intersection yourself.
- On the website, you will find two pieces of code. One inverts a 4x4 matrix, and the other writes a char buffer to a ppm image, which is the expected output of this program.
- The code that inverts a 4x4 matrix expects two 4x4 matrices to be passed in as arguments. The first matrix will be inverted and the result will be stored in the second matrix. Both matrices are row order, so you have M[row][column].
- You may use the STL string and vector classes.
- The assignment must be done from scratch. You can only use code provided by the instructor or the TA as specified in this document, and the libraries allowed. If in doubt, ASK!
- Make sure that your parse routine does not crash based on where the EOF character is.
- Given a reasonable resolution (400x400), your program should take no more than five seconds (probably less than that) to run when compiled in "release" mode.
- A sphere at position (0,0,0), with scaling parameters (1,1,1) should be centered at (0,0,0) with radius 1.
- If the eye-ray is constructed using the convention described in class, then when intersecting it with an object, the closest object is the one with minimum hit time greater than 1. A hit time between 0 and 1 falls between the eye and the near plane, and hence is not a part of the visible view volume.

- When creating rays from the closest hit point on an object, you need to start them at t = 0.000001, to avoid false intersections due to numerical errors. In other words you may not want to consider intersections at t=0 precicely.
- For the keys and results we use the following convention: Rays from the eye that hit nothing return the color of the background, while reflected rays that hit nothing return black (i.e. (0,0,0)).
- The template code for saving your image to disk uses the ppm image format. If you do not already have a program that can read images of this type, you can download, GIMP, IfranView from the web (https://www.irfanview.com/) or Xnviewgb. These are excellent and free programs for viewing images, and can amongst others read ppm files. The default image viewers of most modern operating systems also seem to work.
- The "NEAR" value is an absolute value and represents the distance along the negative z-axis.
- Your code may need to handle hollow spheres, which are "cut" open by the near plane.
- Your code may need to be able to handle lights inside spheres.
- You will be using the following **local illumination** model:
  - PIXEL_COLOR[c] = K_a*I_a[c]*O[c] +
    for each point light (p) { K_d*I_p[c]*(N dot L)*O[c]+K_s*I_p[c]*(R dot V)^n } +
    K_r*(Color returned from reflection ray)
  - O is the object color (<r> <g> <b>)
  - [c] means that the variable has three different color component, so the value may vary depending on whether the red, green, or blue color chanel is being calculated
  - The other components of this equation are explained in the lecture notes.
- You should not spawn more than three reflection rays for each pixel, i.e. stop the recursion after 3 bounces.
- When summing over all lights, it is possible that the value of a channel of the PIXEL_COLOR goes above 1. In this case, the simplest solution is to clamp the value of that channel to 1. Do not forget to scale by 255 before creating the ppm image using the given "`save_imageP6()`" function in ppm.cpp. "`save_imageP3()`" is provided for debugging reasons because it produces a text file that is human readable. The P6 version produces a binary file. Your final program must produce the binary version (P6).
- Make sure you submit all the required files so we can compile and build your program. If there are missing libraries you will get zero marks. On linux we should be able to compile by just typing make "make" and on windows with MS VS -> Build.
- Make sure that your Visual Studio project is designed to create a "Console Application" or "Command Line Tool".
- Make sure that everything works on a standard windows 10 machine.
- Make sure you "clean" your project before zipping it and submitting it, to remove all the large auxiliary files that VS creates.
- Make sure you check regularly the forum and the announcements in case the grading or submission instructions change.

Submission (ssh/terminal or via websubmit) - DUE: Dec 15, 2023, 11:59pm

submit 3431 a3 <firstname_lastname>_a3bundle.zip
or via websubmit: https://webapp.eecs.yorku.ca/submit/