Assignment 2: Raytracer

Computer Graphics – CS 4300/5310

Due: *February 19th, 11:59pm*

You may work in pairs (i.e. groups of two) on this assignment.

Educational Objectives

- Understand ray tracing
- Implement Lambertian diffuse and Blinn-Phong specular shading
- Become comfortable with thinking in 3D

Assignment Description

For this homework, you will be implementing a raytracer along the lines of that described in Chapter 4 of the course textbook. All input data will come from a text file; the format of this file is described below. You are required to produce a single, static image of the scene – there is no need to implement interactive viewpoint navigation.

You will be reading in a scene from a file. We will test your raytracer will several different scene files. **You must make it easy for us to change what file we run your raytracer with**. This may be done by making the filename a command line argument to your program, or by using a file chooser. Processing has a built-in file chooser dialog: *selectInput()*. If you are using Processing to render your image, make sure that you enable *noLoop()* in the draw method and save your render to a file at the end of the draw method using the *save()* function.

I strongly, strongly recommend starting this assignment **early** and testing it incrementally. **Very incrementally.** Probably 90% of all bugs in raytracers manifest themselves as rendering a black screen, and it is very difficult to debug. It is hard to know if the problem is in your camera or your ray-sphere intersection test, for example. If your camera is facing the wrong way, then it's not pointing at the scene. If the ray-sphere intersection test isn't working, then it won't find any objects in the scene. Both bugs result in a black screen.

You want to be sure that the feature you have just implemented is what caused the problem. Start small, test frequently, and add features slowly. ***Develop incrementally.***

Input File Format

The input file for this assignment is an ASCII text file. We promise to only ever give your raytracer well-formed input; while testing for errors in the input is always a good idea, it is not necessary for this assignment. If you encounter a line in the file for a feature you did not implement or that you do not understand, simply throw it away and move on to the next line. This is a computer graphics class, not a text parsing class; you should not be spending massive amounts of time perfecting your parser so that it throws away bad input safely and gracefully.

Each line has a syntax determined by its first set of characters. The meaning, or *semantics*, of each line type is described in detail below. All entries in a line are whitespace-delimited.

All values in the input file except for filename, vertex indices, specular highlight exponent, image resolution parameters, and the recursion depth parameter are floating point numbers. Color values (r, g, b) are guaranteed to be on [0, 1]. All floating point values are written to at most 4 decimal places. It is possible that a value will have a + or – sign in front of it; you should be able to handle this.

Line types marked with an * are optional for undergraduate students but mandatory for graduate students. Line types marked with an ** are optional for all students. See the "raytracer features and assignment grading" section below for more information about assignment requirements for undergraduate and graduate students.

Line type	**Start**	**Rest**
Comment	##	Any remaining text
Vertex	vv	x y z dx dy dz
Ambient Material	am	r g b
Diffuse Material	dm	r g b
Specular Material	sm	r g b n
Transmissive Material**	tm	r g b ior
Sphere	ss	i
Triangle*	ts	i j k
Plane	ps	i
Point Light	pl	i r g b
Directional Light	dl	i r g b
Spot Light*	sl	i r g b
Ambient Light	al	r g b
Camera	cc	i
Image Resolution	ir	w h
Output Image	out	filename
Background	back	r g b
Recursion Depth	rdepth	n

When vertex lines are read in, they should be stored in an array in the order in which they are received.

Raytracer Features and Assignment Grading
You may choose to work in pairs for this assignment. Undergraduate and graduate students have different requirements. If a pair contains a graduate student (even if the other partner is an undergraduate student), the assignment will be graded according to graduate student criteria.

All students are **required** to implement the following features:

- Parsing the file as described above
- Sphere and plane primitives
- Lambertian (diffuse) and Phong (specular) shading
- Point lights, directional lights, and ambient lights
- Parameterizable perspective camera and image resolution
- Parameterizable background color
- Shadows

Graduate students are **required** to implement the following additional features:

- Triangle primitives (worth 5 points if implemented by undergraduate students)
- Reflection (worth 10 points if implemented by undergraduate students)
- Spot lights (worth 5 points if implemented by undergraduate students)

The following features are worth additional credit, to a maximum score of 130% on this assignment:

- Additional primitives, e.g. cones or cylinders (5 points per primitive, 10 points max)

- Constructive solid geometry: union, difference, and intersection (15 points)[1]
- A UI that shows the scene being raytraced pixel by pixel (5 points)
- Refraction (10 points)
- An acceleration structure, e.g. bounding volumes, octree, BSP tree (5 - 10 points)

If you have other ideas for extensions to the raytracer and would like to know how many points they are worth, ask!

10% of your assignment grade will be determined by how readable and well-organized your code is. Comment it well! Partial credit can **only** be assigned to well-commented code.

Vertices
Each line starting in vv defines a 3D vertex in world frame. The first three numbers are the vertex location [x, y, z], and the last three numbers are a direction vector at that location [dx, dy, dz], which has different meanings in different situations, as described below.

Your program must create a *vertex array* corresponding to all the vertices specified in the input, in order. If there are *n* vertices in the input, this array will have *n* entries, indexed from *0* to *n - 1*. In a language like Java, an easy way to implement this is to store each vertex as you read it from the input into a List l (typically ArrayList is used in a situation like this, but LinkedList will work also). Then after the entire input has been processed, convert the list to an array by calling *l.toArray().*

Lights
Lights are defined in the input file as they were in class. Point lights have a point and intensity; assume that the attenuation constants for point lights and spot lights are: a=0, b=0, c=1. All lights have their intensity specified in RGB. All RGB colors are floating point colors specified on [0, 1].

Point lights are positioned at the [x, y, z] coordinate in its specified vertex. The vector portion is ignored. Directional lights have the direction [dx, dy, dz] from its specified vertex. The point portion is ignored. Spot lights have position and direction as specified in the vertex. Ambient light has no position or direction, it simply exists.

It makes sense to store all lights in some sort of data structure; a list would be a good choice.

Camera
There is only one camera in the scene; if there are multiple cameras in the scene file, use the last one specified. The camera is at the position [x, y, z] from its associated vertex, and its view direction is specified by [dx, dy, dz] from its vertex. The up vector is always [0, 1, 0]. The right vector can be calculated as the cross-product of the view direction and the up vector. Assume a 60 degree field of view. If the camera is not specified, assume that it is at [0, 0, 0] and is looking down the negative z axis.

Materials
All materials have ambient, diffuse, specular, and transmissive properties. You are not required to implement transmissive properties for the assignment – this constitutes extra credit (i.e. implementing refraction). Ambient and diffuse properties are specified as [r, g, b] colors. Specular material properties are an [r, g, b] color and an n shininess constant. Transmissive material properties are an [r, g, b] color and an index of refraction for the material (ior). All RGB colors are floating point colors specified on [0, 1].

---

[1] If you choose to implement CSG, you must update your scene file to parse the geometry. You may do this in a manner of your choosing; make sure you describe in your README what format you used.

Material properties apply to all primitives specified after them, until the material property changes. The default material is:

Ambient:	0.2 0.2 0.2
Diffuse:	1 1 1
Specular:	1 1 1 64
Transmissive:	0 0 0 1

Primitives
Sphere primitives are defined by a point [x, y, z] and a radius defined as the norm of [dx, dy, dz]. Plane primitives are defined by a vector that is normal to the surface: [dx, dy, dz] and a depth, the z component of [x, y, z] ([x, y, z] is guaranteed to be on the plane). Triangle primitives are defined by three points in clockwise order: [x1, y1, z1], [x2, y2, z2], [x3, y3, z3]. Each of these points has a normal vector [dx, dy, dz] associated with it; these normal vectors will all be the same as each other (i.e. no curved triangles). However, the normal vectors are not guaranteed to be normalized.

Other Instructions
If implementing reflection and/or refraction, make sure that you obey the maximum recursion depth parameter from the scene file. The image resolution should be taken from the scene file, as should the name of the file you will save.

Incremental Testing
If you do not add features slowly and test your code incrementally, your code will be impossible for **anyone** to debug. At this point in your computer science career, you should be doing the majority of the debugging work for your code yourself. Course staff are happy to help as much as possible, but nobody is going to be able to solve a mess of code that you cannot help us through yourself.

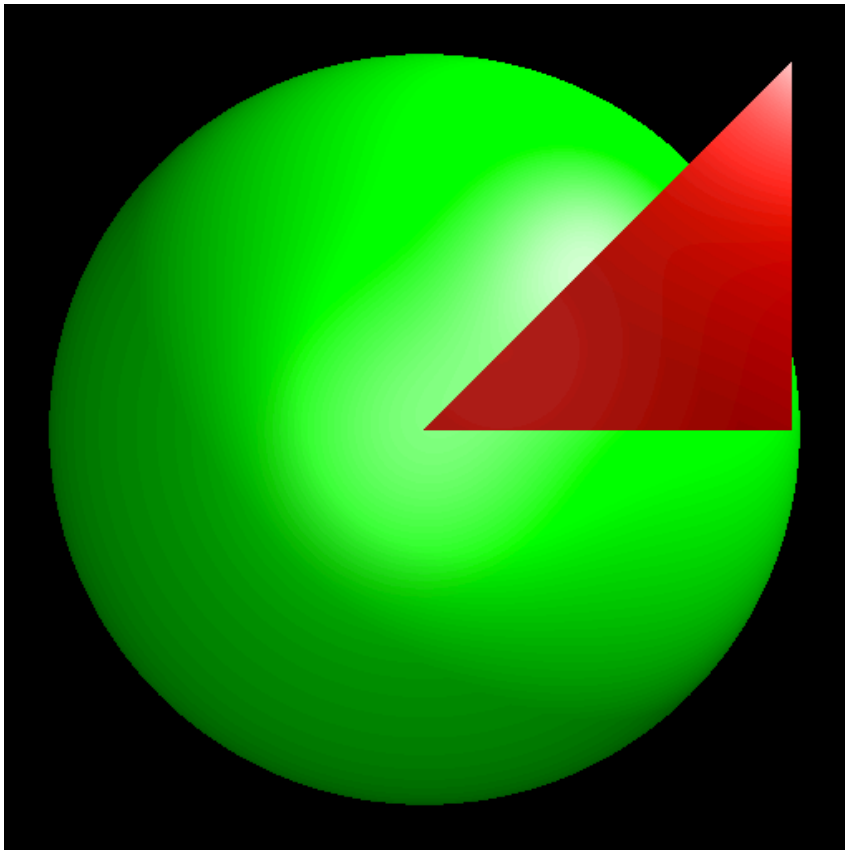
The following order of feature implementation is recommended, but not required. If you follow this implementation script it will make it easier for course staff to understand how far you have got in building your raytracer when you request help.

1. Implement your parser. Store primitives and lights in arrays or another appropriate data structure. Store materials in a data structure and build the references between primitives and materials.
2. Be able to save **anything** to an image of the appropriate width and height. Test that your image saving code is working.
3. Create a camera with a 60 degree view frustum that is looking straight at the scene. Make sure your camera vectors are unit vectors.
4. Implement ray/sphere intersection, if you hit the sphere color set it to white, else color it the background color.
5. Set your sphere to the ambient material color.
6. Implement ambient lighting.
7. Implement Lambertian shading for a single sphere and a single light in the scene.
8. Implement Blinn-Phong shading for a single sphere and a single light in the scene.
9. Introduce new light types, still for just a single sphere. Test the lights incrementally.
10. Implement planes.
11. Implement shadows.
12. [Grad] Implement triangle primitives.
13. [Grad] Implement spot lights.
14. [Grad] Implement reflection.
15. Any extra credit.

Examples

Here is an example scene file and the associated scene produced by the sample code. [Note the image has been scaled to fit on the page.]

```
## sample input file
al 0.1 0.1 0.1
output sample
ir 512 512
rdepth 2
back 0 0 0
cc 4
pl 5 0.8 0.8 0.8
dl 6 0.5 0.5 0.5
vv 0 0 -10 -1 -1 3
am 1 0 0
vv 10 0 -10 1 -1 3
dm 1 0 0
vv 0 0 -100 0 50 0
sm 1 1 1 16
ts 0 1 3
am 0 1 0
dm 0 1 0
ss 2
vv 10 10 -10 1 1 3
vv 0 0 10 0 0 -1
vv 100 100 0 0 0 0
vv 0 0 0 0 0 -1
```



Submission Instructions

A zip file containing the following information must be uploaded to Blackboard:

- Your well-documented code
- A Windows or Linux executable
- Instructions for how to run your program and choose a scene file
- Three example output images and associated scene files
- A list of features you implemented on top of the required features
- A README listing the number of late days you wish to use, providing the names of both team members (or state if you worked alone), and whether you are an undergraduate or graduate student team

Emailed assignments will **not** be accepted.