Shading for Ray Tracing

Computer Graphics: Project 3B

1 - Objective

The goal of this project is to complete your ray tracing renderer. In this project you will expand your Ray Tracer to detect intersections between rays and vertically-oriented cylinders. You will also expand your shading function to include ambient and specular shading, as well as cast shadows (via shadow rays) and reflection (via recursive ray casting).

2 - Deadline

This project should be submitted on T-Square by 11:55PM on Tuesday, March 27, 2018.

3 - Process

3.1 Source Code for this Project

You will base your updated program on the code that you wrote for project P3A.

3.2 Project description

You have four main tasks for this second part of the ray tracing project:

1. Detect ray/cylinder collisions
2. Expand your shading function to include ambient and specular light
3. Cast shadow rays to determine when a light should not contribute to the color
4. Implement reflection by recursively casting new rays into the scene inside of your shading function

You can accomplish these goals however you see fit. A good approach would be to use object oriented programming practices and create objects for each of the major scene components (scene, light, surface material, ray, sphere, cylinder, hit information). Global lists of scene objects could be stored to allow for easy access. Even if you got everything working correctly for part 3A, you may find that it is better to re-arrange your code to prepare for the second part of this assignment.

You will need to work out the details of how to ray trace a cylinder. You need to intersect your ray with both the main (round) body of the cylinder and also the disks at either end. Intersecting with the round body is similar to intersecting a ray with a sphere, except that there is no y^2 term. Intersecting the ray with a disk on the end is similar to a ray intersecting a polygon: first intersect the ray with the plane containing the disk (should be an equation such as y = ymin). Then, see if the point of intersection is inside the disk. Scene i5.cli contains two cylinders, and is probably the most helpful for this part of the assignment. Note that other scenes use the tops of cylinders as ground planes.

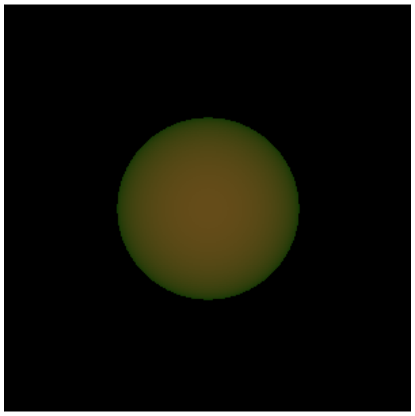
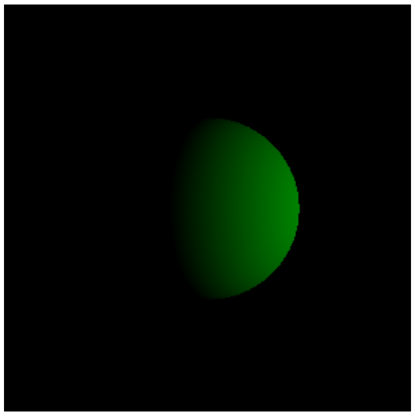
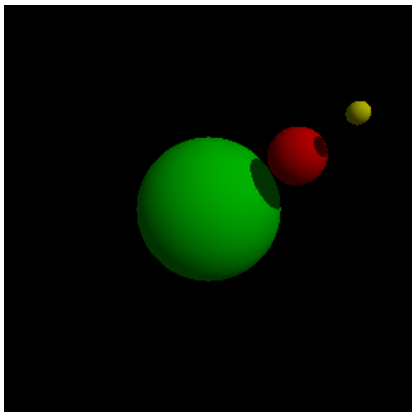
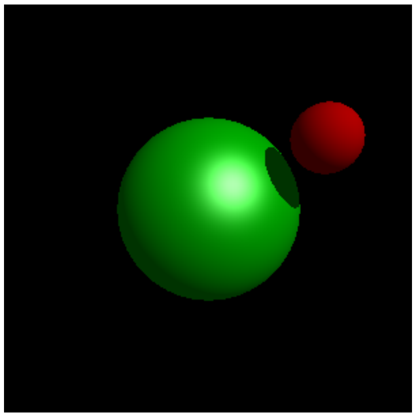
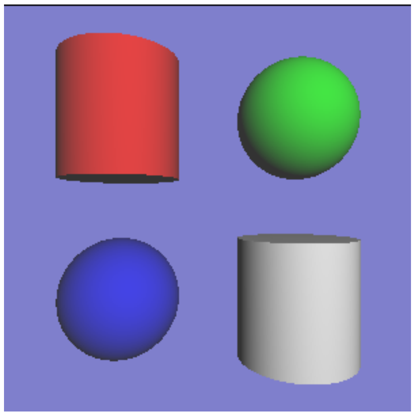
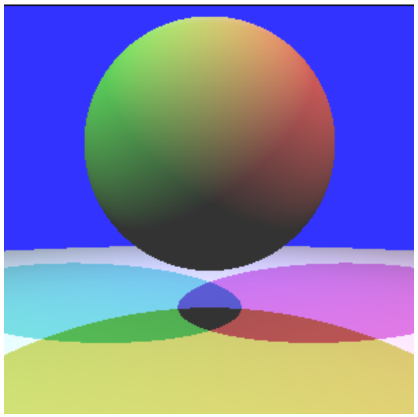
Note that your code will need to determine whether or not a ray hits an object in three different cases: eye rays, shadow rays, and reflection rays. We strongly urge you to write a **single** routine that helps you to accomplish these separate tasks. You could call such a routine RayIntersectScene. You would pass to this routine a ray, and then have this routine return information about whether the ray hit anything. The routine should test the given ray for intersection with each of the visible objects in the scene, that is, all spheres and all cylinders. It should determine the closest such hit, creating a Hit data structure that specifies information about this nearest hit, and finally returning this Hit object. Inside the Hit object would be all the information that you will need to know about what the ray hit, including the ray "t" value, the 3D point of intersection, the surface normal, and a reference to the surface material that was hit (e.g. surface colors and so on). If your ray misses all objects, the RayIntersectScene routine could return a null pointer, or your Hit object could include a boolean that says whether or not the ray hit anything.

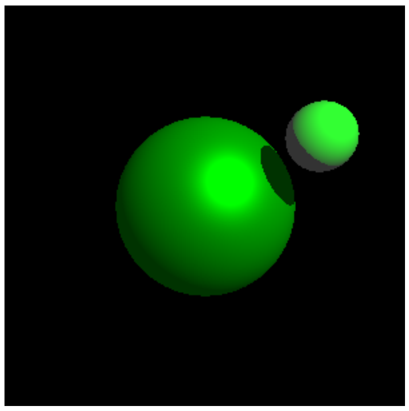
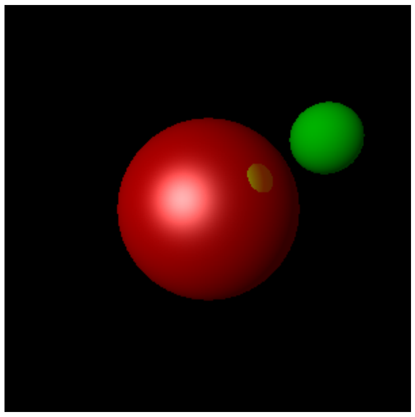
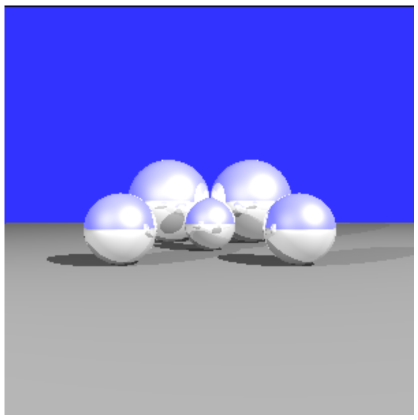
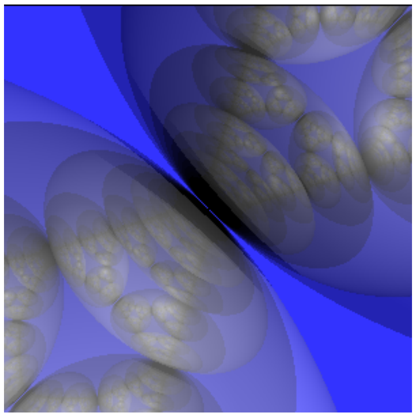
For shadow rays, all you will need to know is whether the ray hit any objects. For eye and reflection rays, you will need to take the Hit information that RayIntersectScene returned and pass it to a shading function that will determine the color for the given ray. In the case of reflections, the shading function will need to call itself recursively in order to get the color for the reflection ray. Such an approach would be easy to test and clean to implement.

No matter how you implement the Ray Tracer, your results should appear like the examples below.

3.3 Result Images

Below are the correct images for the ten given scene files (i1.cli through i10.cli). Note that the colors of some spheres are different than those shown in the part A solutions, due to ambient, specular, or reflection contributions. For example, the sphere in i1.cli now has a greenish-yellow color due to the ambient component of its surface material.

3.4 Example Scene Files

The scene files for this project are exactly the same .cli files as for part A.

3.5 Your Code

You should modify your project 3A code in any way you see fit to solve 3B. Visit "py.processing.org/reference/" for more information on Processing. You are NOT allowed to use most of the built in processing/openGL functions in this project. The exceptions are that you can use PVector and the standard math functions. Be aware, however, that some students have reported that the PVector operations are slow. When in doubt about this, ask the instructor. As with project 3A, you must set the color of each pixel manually, such as by using rect() or set().

3.6 Scene Description Language

Each scene is described in a .cli file using the grammar described below. These files are contained in the **/data** folder. The suffix "cli" stands for "command language interpreter".

fov angle

Specifies the field of view (in degrees) for a perspective projection. The viewer's eye position is assumed to be at the origin and to be looking down the negative z-axis (giving us a right-handed coordinate system). The y-axis points up.

background r g b

Background color. If a ray misses all the objects in the scene, the pixel should be given this color.

light **r g b x y z**

Point light source at position (x,y,z) and its color (r, g, b). Your code should allow for at least 10 light sources. For this second part of the assignment, you will cause these lights to cast shadows.

surface **Car Cag Cab Cdr Cdg Cdb Csr Csg Csb P Krefl**

This command describes the reflectance properties of a surface, and this reflectance should be given to the objects that follow the command in the scene description, such as spheres and cylinders. The first three values are the ambient coefficients (red, green, blue), followed by diffuse and specular coefficients. Next comes the specular power P (the Phong exponent), which says how shiny the highlight of the surface should be. The final value is the reflection coefficient (0 = no reflection, 1 = perfect mirror).

Usually, 0 <= Cd,Ca,Cs,Krefl <= 1. When Krefl is larger than zero, this indicates that you will need to create reflection rays for this surface.

sphere **x y z radius**

A sphere with its center at (x, y, z) and the given radius.

cylinder **radius x z ymin ymax**

A vertical cylinder of a given radius. The axis of the cylinder is parallel to the y-axis, so all of your cylinders will be oriented up-and-down. The position of the cylinder's axis is given by the (x,z) values. The lower and upper caps of the cylinders are at ymin and ymax.

write [filename].png

Ray-traces the scene and saves the image to a PNG image file.

3.7 Debugging Suggestions

If you have a bug in your ray tracer, you will want to print out some values to the screen (hit position, surface normal, light vector, etc.) We strongly suggest you should only print out information for ONE PIXEL, and not for all of the pixels on the screen. You could do this by creating a Boolean variable "debug_flag". In the loop that creates the rays from the eye, only set this Boolean to "true" for one pixel on the screen. Then make sure all of your debug print statements only print when debug_flag is set to "True".

It is sometimes useful to use the mouse to select a particular pixel to debug with. You can determine the coordinates of a given pixel in your image by clicking in the image with the mouse. You can do this by defining a routine MousePressed() that will be called every time a mouse button is clicked. Processing will call this named routine if you define it. You can then print the built-in variables mouseX and mouseY to show what pixel was clicked on. Then you can use these numbers to set debug_flag to "true" just for this one pixel. Don't forget that a zero value for mouseY indicates that the cursor is at the **top** of the screen, not the bottom.

Keep in mind that all of the scene description files (i1.cli to i10.cli) are text files that you can view and modify. It may be helpful for you to look at or modify one of these files to help you figure out what is going on with your ray tracing code.

3.8 Authorship Rules

The code that you turn in should be entirely your own. You are allowed to talk to other members of the class and to the Professor and the TA about general implementation of the assignment. It is also fine to seek the help of others for general Processing/Java programming questions. You may not, however, use code that anyone other than yourself has written. Code that is explicitly not allowed includes code taken from the Web, from books, or from any source other than yourself. The only exception to this is that you may include code that was specifically provided for this assignment or the previous one (Project 3A and 3B). You should not show your code to other students. Feel free to seek the help of the Professor and the TA's for suggestions about debugging your code.

3.9 Submission

In order to run the source code, it must be in a folder named after the main file (ending in .pyde). All other source code files (ending in .py) must also be in this folder. When submitting any assignment, leave your code in this folder, compress into a zip file (not rar or tar, please!) and submit via T-square.