CS 7490, Spring 2016

Homework 1A: Basic Ray Tracing

Due: 11:55pm, Monday, January 25, 2016

Objective

The goal of this project is to write a basic ray tracing renderer. Later we will add to this, so you should take care to organize your data structures carefully. This project is divided into two parts, 1A and 1B, and this web page just describes part 1A. Your program should be able to read scene data from a file according to a defined scene description language. From this, your program will then render an image of the scene and write out the image to a file. In this part of the project we will use just one kind of geometric primitive, the sphere.

Scene Description Language

- **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.

- **point_light x y z r g b**
  Point light source at position (x,y,z) and its color (r, g, b). Your code should allow at least 10 light sources. For now, these lights do not need to cast shadows.

- **diffuse Cd_r Cd_g Cd_b Ca_r Ca_g Ca_b**
  This command describes the color of a diffuse surface, and this reflectance should be given to the objects that follow the command in the scene description, such as spheres and triangles. The first three values are the diffuse coefficients (red, green, blue), followed by ambient coefficients.

- **sphere radius x y z**
  Create a sphere with its center initially at (x, y, z) with the given radius. The sphere's surface color should be determined by the most recently issued **diffuse** command.

- **write filename[.png]**
  Ray-traces the scene and saves the image to a PNG image file.

Note on color specification: Each of the red, green, and blue components range from 0.0 to 1.0.

Extra Commands

Some of you may have created a ray tracer before that is similar to this one in Processing. If so, you should first make sure that all of the above commands work correctly. Then, you should add the following commands. You should only implement one of the two primitives below, either **sphere2** or **cylinder**, which ever one you prefer.

- **fisheye angle**
  Specifies the field of view (in degrees) for a fisheye lens. When this command is given, your renderer will use a fisheye lens instead of the standard perspective projection to create its images. Consult Paul Bourke's web page for details. You may implement either a hemispherical or angular fisheye lens.

- **shiny Cd_r Cd_g Cd_b Ca_r Ca_g Ca_b Cs_r Cs_g Cs_b Exp K_{refl} K_{trans} Index**
  This command describes the color of a shiny surface, and this reflectance should be given to the objects that follow the command in the scene description, such as spheres and triangles. In addition to the parameters that it shares with the Diffuse command, it also includes the color and the exponent for a specular highlight. Moreover, it specifies a reflective coefficient and a refractive coefficient. This means such surfaces should spawn both reflective and refractive rays. The **Index** is the index of refraction of the surface.

- **sphere2 radius x y z**
  Create a sphere with a different intersection routine than you used in your earlier ray tracer. Which of your sphere implementations is faster?

- **cylinder radius x z ymin ymax**
  Create a cylinder that has its axis parallel to the y-axis. Your cylinder should have end caps.

- **refine off/on**
  When the **refine** flag is turned on, your program should render the ray traced scene using progressive refinement. This is a way of showing a crude version of the scene fast, and then adding more and more details until the final image is ready. For the first rendered version, you should render every 8th ray horizontally and vertically. Instead of just coloring a single pixel with such a ray, you will instead draw a solid colored 8x8 square. This method will give you a blocky version of your image. Then render every 4th ray, showing the results as 4x4 pixel blocks. Keep doing this until the final rendering pass shows the fully rendered scene, and uses a different ray to color each pixel.

Code Provided

All the provided source code that you will need for this assignment can be downloaded here: ray_tracer.zip. A interpreter routine is provided so that you do not have to write a parser for the scene description language. Here are the details on this routine:

- **interpreter**

This routine takes the current active CLI file(which you can switch between through numeric keys) and parses it into tokens. Those tokens are organized into arrays. For example, the input "background r g b" will be put into token[0], token[1], token[2], and token[3]. You can access token[2] if you want to get the green value. Based on each token in token[0], your program should do different kinds of stuff discribed in the previous section. By default, three tokens has been handled: "rect", "color", "read" and "write." "rect" token does basic rectangle drawing. "color" changes the color to be filled for the later drawing calls. "write" dumps the current frame into an image. "read" caused the interpreter to read commands from another .cli file, which is useful for describing a scene in multiple sub-files.

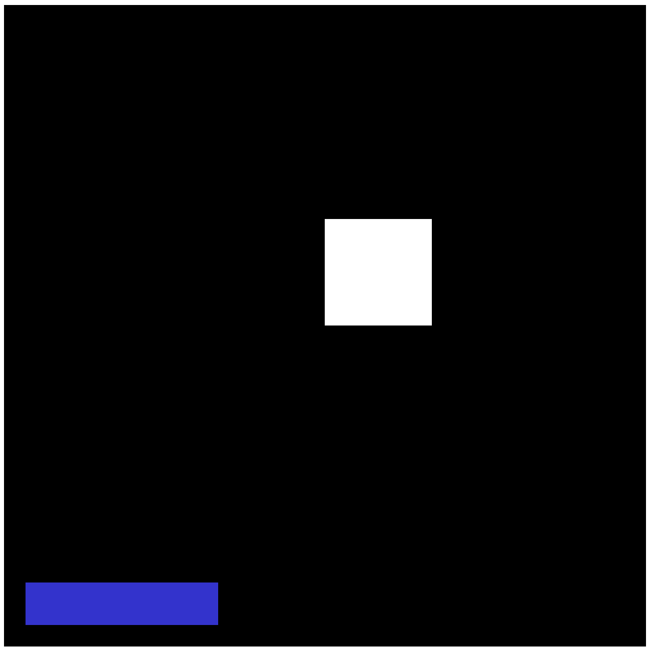
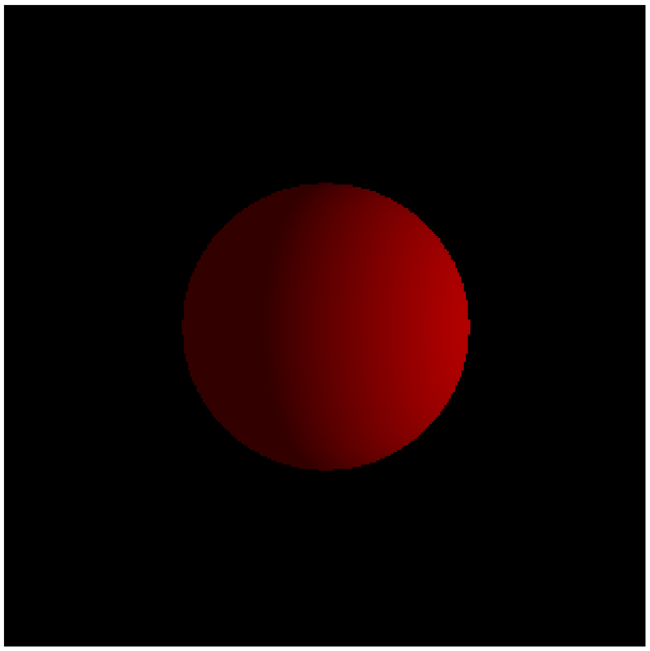
You should feel free to use some of Processing's pre-defined data types, including PVector. Here is a good link about PVector: PVector

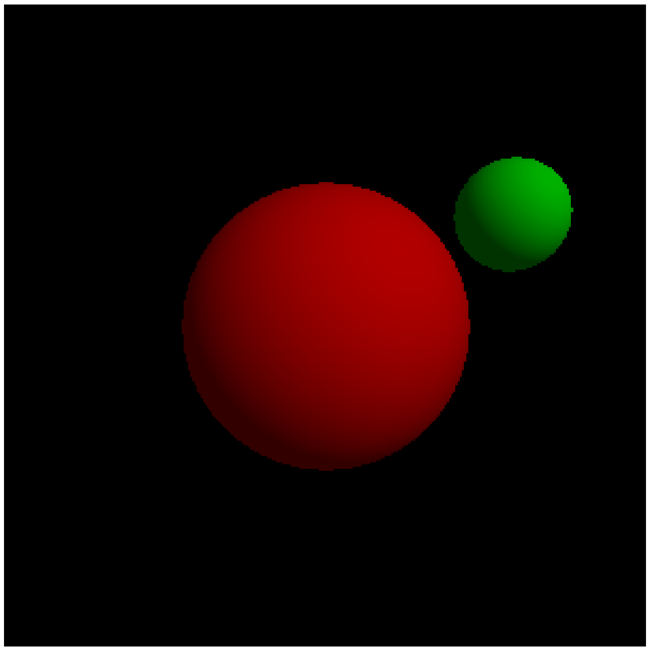
---

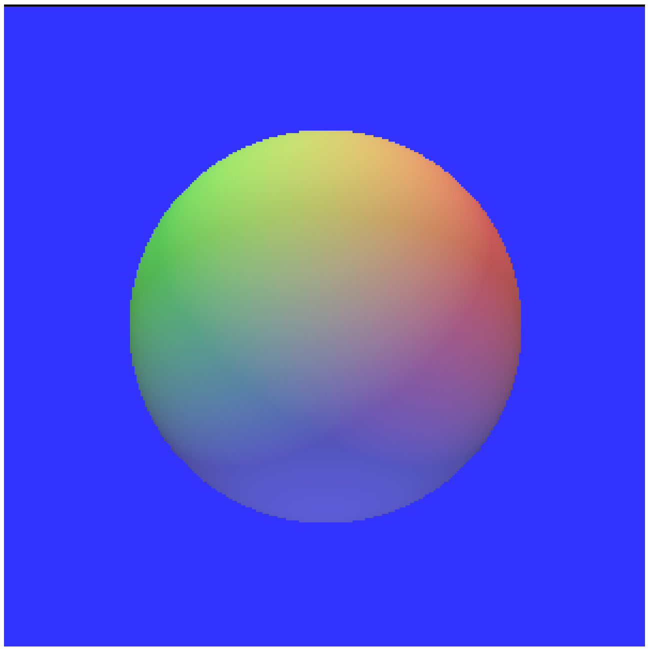
Scene Files

In the directory "**data**" are several test scenes that are described by .cli files. Also in that directory are the images that should be created by these scene files. The file "t01.cli" is perhaps the most simple image, and you might use this scene as a starting point. Pressing keyboard keys 1-9 and 0 calls these test CLI files.
Note: Don't change the directory name or processing won't reconize those files.

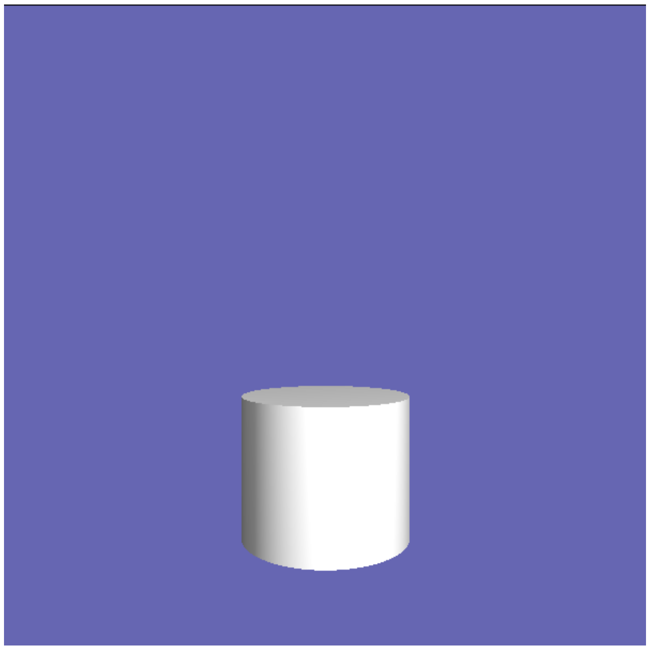
Sample Results:

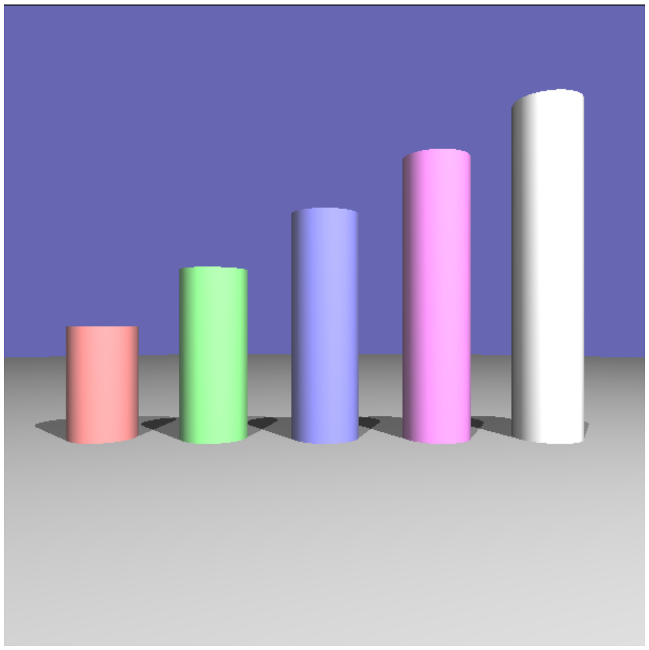



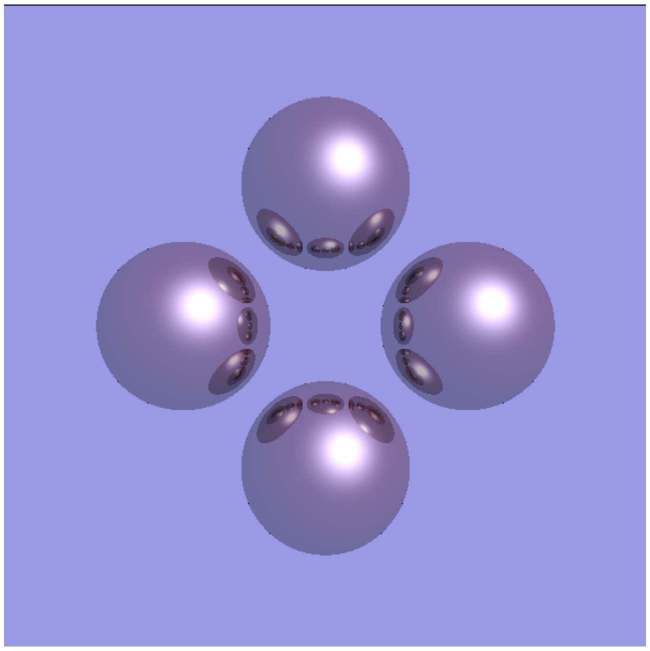
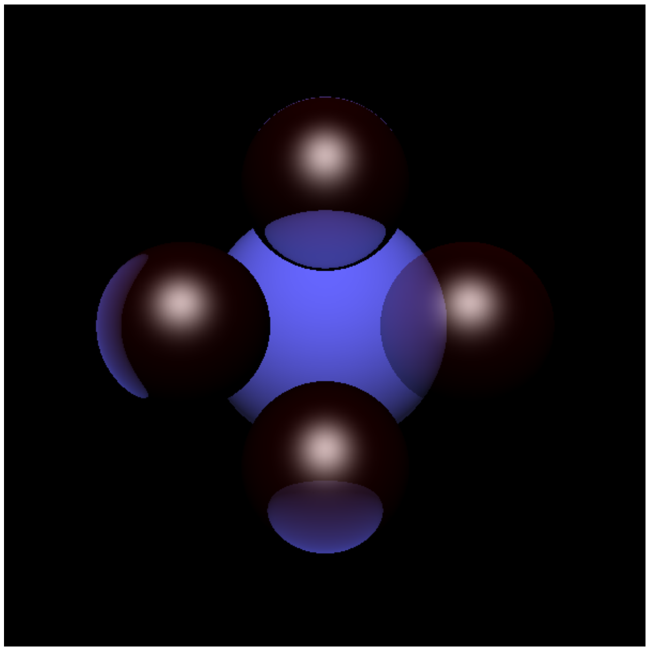




Extra Results:





Suggested Approach

It is probably best to begin by making small changes to the "ray_tracer.pde" program to see how to add new commands to the CLI interpreter. After you feel comfortable with this, then you can make dummy routines for each of the commands that your ray tracer should understand. Then replace each of the dummy routines with real code and you will be done! :-)

Once the shell of the program is ready, I suggest creating the main loop that creates the primary rays (those from the eye). Start by printing out the information about these rays to make sure that the numbers make sense. Once you've got this loop working, then start on the ray/sphere intersection routine. Don't bother shading the sphere correctly at first-- just color it anything other than the background color. Once you've got a round object on the screen, then work on the shading of the sphere.

The best way to write a ray tracer is object-oriented. This is especially true of implementing primitives such as spheres. Later we will be adding more primitives, including polygons. I recommend creating a "primitive" class and make spheres a sub-class of this base class.

Authorship Rules

The code that you turn in must be entirely your own. You are allowed to talk to other members of the class and to the instructor about high-level questions about the assignment. 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 rule is that you should use the parser routines that we provide. You should not show your code to other students. If you need help with the assignment, seek the help of the instructor.

Development Environment

You must use the Processing language which is built on Java. Be sure that you are using Processing version 3.0 or higher. The best resource for Processing language questions is the online or offline Processing language API (found in the "reference" subdirectory of the Processing release).

What To Turn In

Compress the whole folder for this project (not merely the files within the folder) into a zip archive submit them to T-square. The zip archive should be included as an attachment. When unzipped, this should produce a folder containing all of your .pde files and a directory "data".
