CS 7490, Spring 2016

Homework 1B: More Basic Ray Tracing

Due: 11:55pm, Friday, February 5, 2016

Objective

The goal of this project is to extend your basic ray tracer to handle polygons, shadows, and a matrix stack for transformations.

New Commands for Scene Description Language

- **point_light x y z r g b**
  As before, your ray tracer should include point light sources. From now on, however, you should modify your shading routine so that each light source correctly casts shadows.
- **begin**
  Begin the definition of a polygon. Should be followed by "vertex" commands, and the polygon definition is terminated by an "end".
- **vertex x y z**
  One vertex of a polygon. For this project, all of the provided polygons will be triangles. This means you can assume that there will be exactly three "vertex" commands between a "begin" and "end". The coordinates of this vertex should be affected by the current transformation matrix.
- **end**
  End the definition of a polygon.
- **sphere radius x y z**
  Create a sphere with its center initially at (x, y, z) with the given radius. The final center and radius of the sphere should be affected by the current transformation matrix. Don't worry about rotations or non-uniform scaling. We will address these transformations of the sphere in a later assignment.
- **push**
  Duplicate the current transformation matrix and push it on top of the matrix stack.
- **pop**
  Pop the top matrix off the matrix stack and discard it.
- **translate x y z**
  Creat a translation matrix and multiply the current transformation matrix by this translation

matrix.

- **scale x y z**
  Creat a scale matrix and multiply the current transformation matrix by this scaling matrix.

- **rotate angle x y z**
  Creat a rotation matrix by the given angle IN DEGREES around the axis (x, y, z) and multiply the current transformation matrix by this rotation matrix.

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.

- **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.

- **orthographic width height**
  Change the projection to orthographic. The primary rays that you shoot will all be parallel to one another. The center pixel in the window will still point down the -z axis. Other rays will have their origin on the z=0 plane and be in the range of [-width/2, width/2] in x and in [-height/2, height/2] in the y direction.

- **spotlight x y z dx dy dz angle_inner angle_outer r g b**
  Create a spotlight. In addition to the position of the spotlight, the command specifies the direction in which the light is pointing and an inner and outer angle. If a point is inside the cone of the inner angle, it is fully lit. If it is between the inner and outer angle, it is partially lit. If it is outside of the outer angle, it is not lit by this light source. Note that the angle to a given point can be calculated based on the dot product between the (normalized) spotlight direction and a (normalized) vector from the light to the point in question.

- **plane a b c d**
  Create the geometry for an infinite plane. The function that defines the plane is f(x,y,z) = ax + by + cz + d = 0. Thus the values a, b, c specify a normal vector to the plane. As with all geometric objects, the reflective properties of the plane are given from the most recent material command (e.g. diffuse, shiny).

Code Provided

All the provided source code and example scene files that you will need for this assignment can be downloaded here: ray_tracer_p1b.zip.

You should feel free to use some of Processing's pre-defined data types, including PVector and PMatrix3D. You can also make use of the Processing routines such as "translate", "scale", "rotate", "pushMatrix", "popMatrix", and "getMatrix". This will help you add the matrix commands to your ray tracer with less effort. For example, the following command reads the matrix from Processing:

```
  PMatrix3D mat = (PMatrix3D) getMatrix();
```

You will need to use the P3D environment to use the Processing matrix stack. You set this while using the size() command, usually in setup(). Unfortunately, when using this environment, Processing sets the current transformation matrix to a value that is not the identity, for the purpose of drawing 2D objects. If you want to use Processing's matrices, this means you will have to save this 2D drawing matrix, reset the global matrix to the identity (using resetMatrix()), and then re-introduce the 2D drawing matrix each time you draw a pixel on the screen. The alternative to this is to write your own matrix routines, which is fine as well. Here are some helpful links about PVector and PMatrix3D:

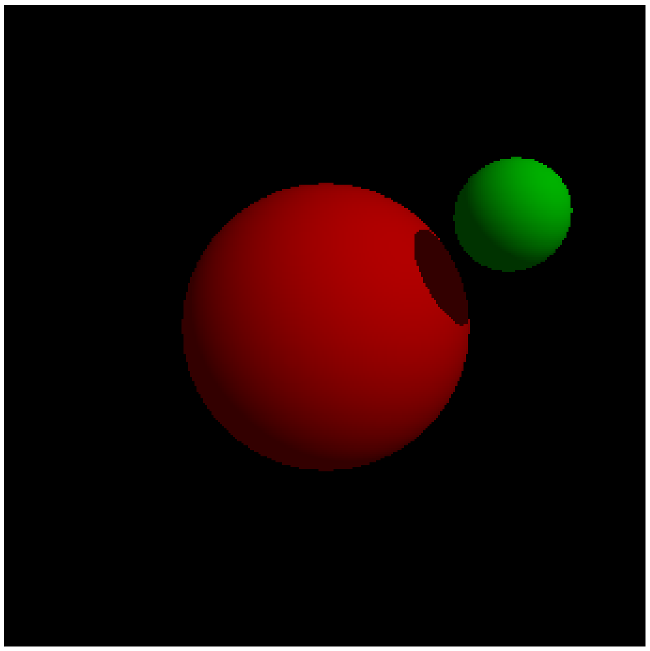
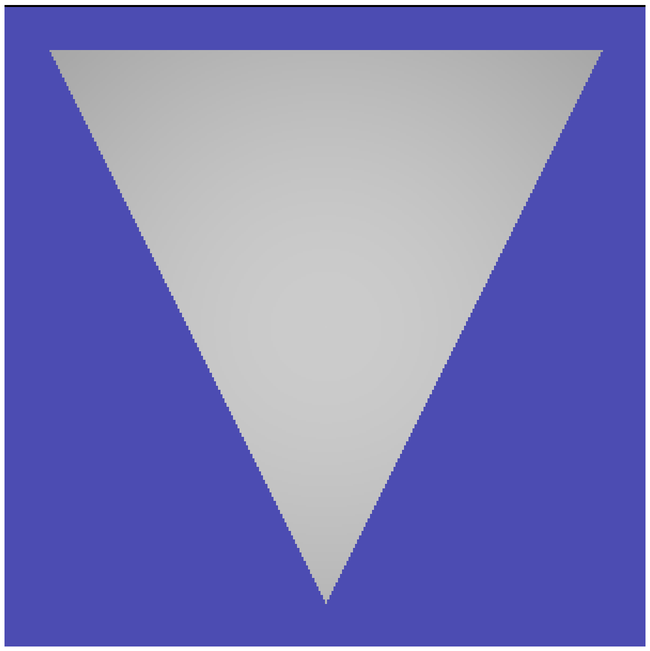
PVector

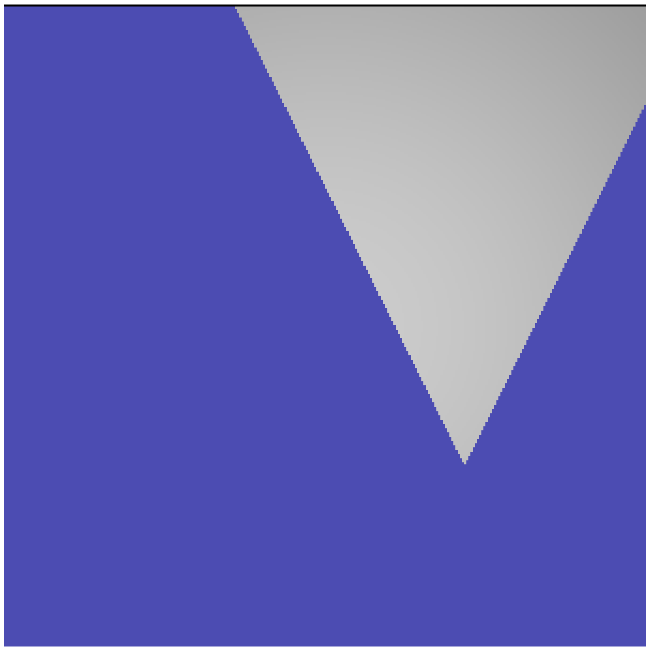
PMatrix3D

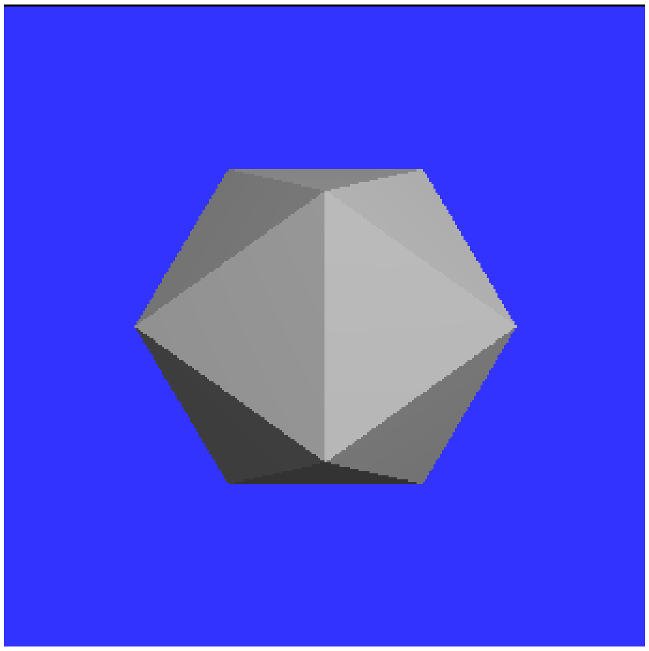
Scene Files

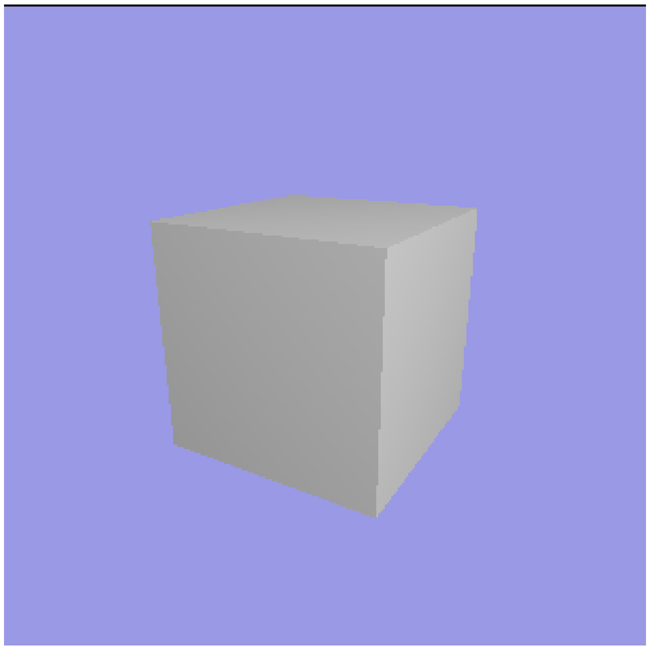
In the directory "**data**" are several test scenes that are described by .cli files. Also in that directory are the images that should be created by these scene files. The file "t01.cli" is perhaps the most simple image, and you might use this scene as a starting point. Pressing keyboard keys 1-9 and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

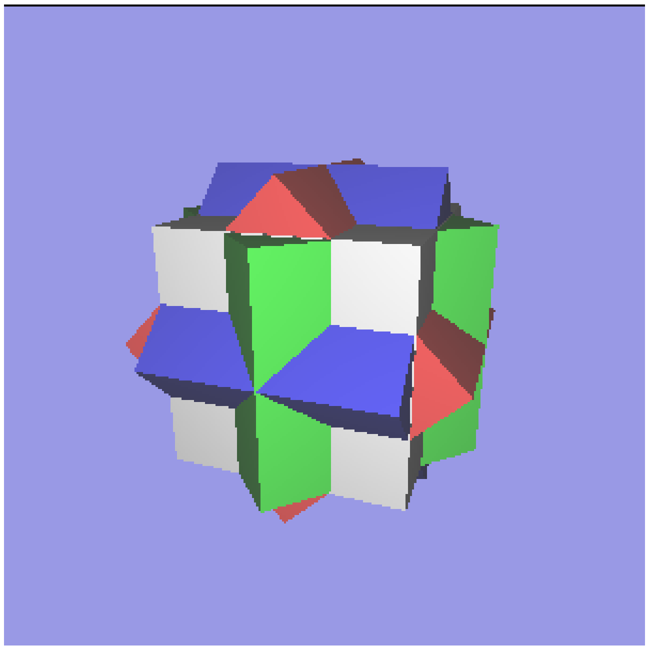
Sample Results:

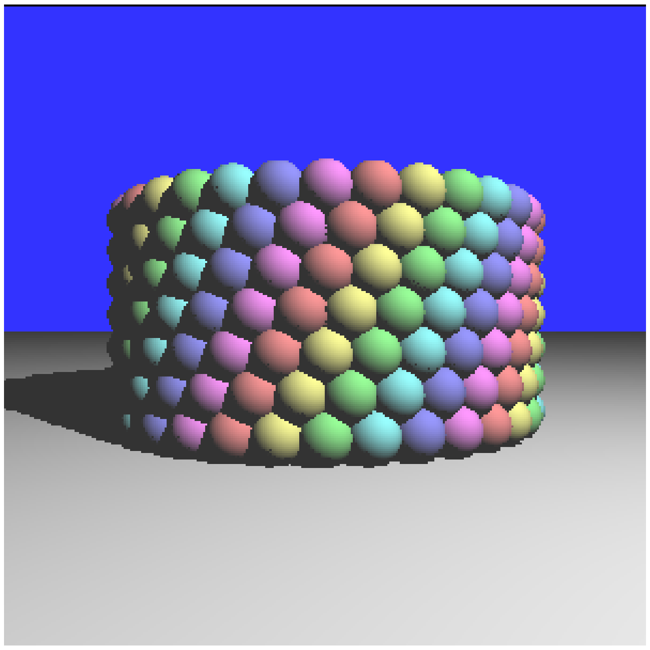













Suggested Approach

The best way to write a ray tracer is object-oriented. This is especially true of implementing primitives such as spheres and polygons. We recommend creating a "primitive" class and make spheres and triangles sub-classes of this base class. This will allow you to create a single list of all of the geometric objects in your scene. That way, you won't need to loop through a new set of objects each time you add a new geometric primitive to your ray tracer.

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. The only exception to this is that you are encouraged to make use of the provided source code that includes the scene description parser. Code that is explicitly **not** allowed includes code taken from the Web, from books, or from any source other than yourself. 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".