Ray Tracer I: Ray Casting

Due date: 11:59pm April 12, 1999

1 Overview

In this assignment you will implement the camera and several primitive objects for a ray tracer. We will not do recursive ray tracing yet, just ray casting.

The input language of the ray tracer will be similar to a subset of the geometry file format of POV raytracer.

2 What your program should do

Your program should read a file specified on the command line and create an image of the scene described in the file using ray casting.

In addition to the name of the file, on the command line you should be able to specify the size of the output image.

3 File format description

The input file format is similar (but slightly different from) the POV-Ray file format. The file in this format is a sequence of object descriptions. You should implement the following POV-Ray objects: `box, sphere, cone, cylinder, quartic`.

Comments. Two slashes are used for single line comments in the file. Anything on a line after a double slash // is ignored by the ray tracer. For example:

```
 // This line is ignored
sphere { <10, 10, 10>, 1 } // a sphere of radius 1 centered at <10, 10, 10>
```

Note that a 3D vector or point is specified using angular brackets, and the components are separated by commas.

Objects. The basic syntax of an object is a keyword describing its type, some floats, vectors or other parameters which further define its location and/or shape and some optional object modifiers such as pigment or transformations. In this assignment the only modifiers that we consider are transformations.

Sphere. The syntax for a sphere is:

```
sphere { RADIUS }
```

where `RADIUS` is a float value specifying the radius. The sphere is assumed to be centered at zero. You can also add translations, rotations, and scaling to the sphere. For example, this is a sphere of radius 10 centered at 25 on the Y axis:

```
sphere { 1.0
  scale 10
  translate <0,25,0>
}
```

Transformations can be also applied to all other objects (see details on transformations below).

Box. A simple box can be defined by listing two corners of the box like this:

```
box { <CORNER1>, <CORNER2> }
```

Where `<CORNER1>` and `<CORNER2>` are vectors defining the x,y,z coordinates of opposite corners of the box. For example:

```
box { <0, 0, 0>, <1, 1, 1> }
```

Note that all boxes are defined with their faces parallel to the coordinate axes. They may later be rotated to any orientation using a `rotate` parameter.

Each element of CORNER1 should always be less than the corresponding element in CORNER2.

Cone. A finite length cone or a frustum (a cone with the point cut off) may be defined by

```
cone { HEIGHT, RADIUS1, RADIUS2 }
```

where HEIGHT is the height of the cone and RADIUS1 and RADIUS2 are float values for the radius of the ends of the cone For example: The center of the base of the cone is assumed to be at the origin with the symmetry axis pointing along the Y axis. Example:

```
cone { 3,2,0}
```

is a cone 3 units tall pointing up the y axis. The base has a radius of 2. The other end has a radius of 0 which means it comes to a sharp point. If neither radius is zero then the results look like a tapered cylinder or a cone with the point cut off.

The ends of a cone are closed by flat planes which are parallel to each other and perpendicular to the axis of the cone.

Cylinder. A finite length cylinder with parallel end caps may be defined by

```
cylinder { HEIGHT, RADIUS }
```

As the cone, the cylinder's base is centered at the origin and and the Y axis is its axis. For example,

```
cylinder { 3, 2}
```

is a cylinder 3 with a radius of 2.

The ends of a cylinder are closed by flat planes which are parallel to each other and perpendicular to the length of the cylinder.

Polygon. A polygon with N vertices is defined by:

```
polygon { <VERTEX1>, <VERTEX2>, ..., <VERTEXN> }
```

where <VERTEXi> is a vector defining the x,y,z coordinates of each corner of the triangle.

Quadric. Quadric surfaces can produce shapes like ellipsoids, spheres, cones, cylinders, paraboloids (dish shapes), and hyperboloids (saddle or hourglass shapes). A quadric is defined in POV-Ray by:

```
quadric { <A,B,C>, <D,E,F>, <G,H,I>, J }
```

where A through J are float expressions.

This defines a surface of x,y,z points which satisfy the equation:

$$Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz + J = 0$$

Different values of $A, B, C, \ldots J$ will give different shapes. So, if you take any three dimensional point and use its x, y, and z coordinates in the above equation, the answer will be 0 if the point is on the surface of the object. The answer will be negative if the point is inside the object and positive if the point is outside the object. Here are some examples:

- $x^2 + y^2 + z^2 - 1 = 0$ Sphere;
- $x^2 + y^2 - 1 = 0$ Infinitely long cylinder along the Z axis;
- $x^2 + y^2 - z^2 = 0$ Infinitely long cone along the Z axis.

Camera. The camera definition describes the position, field of view and properties of the camera viewing the scene. If you do not specify a camera then a default camera is used. The camera is defined by the following parameters: `location, look_at, fov, up, ratio` (this is different from the standard POV-Ray definition).

Location and lookat. Under many circumstances just two vectors in the camera statement are all you need: location and look_at. For example:

```
camera {
  location <3,5,-10>
  look_at  <0,2,1>
}
```

The location is simply the X, Y, Z coordinates of the camera. The default location is ¡0,0,0¿. The `lookat` point was discussed at a previous lecture. By default the camera looks at a point one unit in the -Z direction from the location.

The field of view. The keyword `fov` is used to define the filed of view (in degrees). A reasonable field of view is 30-50 degrees; fields of view wider than 90 degrees result in gross distortion.

The up vector. The keyword `up` is used to define a vector for the vertical direction. Before using the vector, subtract the component parallel to the viewing direction, if any. Also, normalize it to be of unit length. The default up vector points along the Y axis, unless it is the viewing direction. In the latter case, it is points along the X axis.

The aspect ratio is defined using the keyword `ratio`. The default aspect ratio is 1.

Transformations. Three standard types of transformations can be applied in any order to any object: translate, rotate and scale. Any number of transformations can be applied. The argument of `rotate` is a vector (the rotation axis) and a number rotation angle in degrees):

```
rotate <0,1,0> 45 // rotate around Y axis by degrees
```

The `scale` is a uniform scale and takes a single number as an argument. The `translate` takes an vector as an argument.

4 What to turn in

Create a test image from the file that will be provided on the Web page. Also, create 3 images of a 1x1 square in the XY plane for the following camera parameters:

- ```
  camera {
  location 0.5 0.5 1
  fov 90
  }
  ```
- ```
  camera {
  position 2 1 2
  look_at 3 1 3
  fov 90
  ratio 1.5
  }
  ```
- ```
  camera {
  position 2 2 2
  look_at 0 0 0
  up 1 0 0
  fov 90
  }
  ```

5 Implementation Suggestions

General organization. Your ray tracer will consist of two main parts: the rendering engine and a set of modules to manage the parameters of the camera, databases of shapes, light sources and materials. It is convenient to represent each shape with a class derived from a base shape class, and
  ```

use virtual methods for operations like intersecting the object with a ray. You will need to perform a lot of vector arithmetics. Write a small library of standard vector functions (addition, multiplication by a constant, dot product, cross product etc.)

Parsing the geometry file. Use the standard approach: write a function `getToken`, that skips the comments and returns the next token. A token is either an identifier, a floating point number, a comma, or one of the symbols `{`, `}`, `<`, `>`. An identifier consists of letters, numbers, and possibly an underscore, with a starting letter or an underscore. A floating point number consists of a sign, which can be ommitted, a sequence of digits, possibly followed by a decimal dot, possibly followed by another sequence of digits. Numbers and identifiers have to be separated from each other by whitespace characters (space, tab, newline), but need not be separated from the comma or `{`, `}`, `<`, `>`.

The parser calls `getToken`, determines the object type from the first token and calls the function that parses the object definition for each object. These functions, in turn, may call functions for parsing transform definitions. Do not try to do any extensive error processing: if there is a syntax error in the file, print a message and abort.

In general, each object type (sphere, cone, cylinder, polygon, quadric) corresponds to a class in your program. Any object may have a transformation attached; assemble all transformations for each object into a single matrix, and transform the ray into the object's coordinate system when computing intersections. For this, you need to multiply both the direction and the origin of the ray by the *inverse* of the transformation matrix.

Create a data structure to describe rays. Initially, it should contain at least three fields:

```
double[3] orig; // origin of the ray
double[3] dir;  // direction of the ray
float[3] color;
```

Note that we use `doubles` for all fields except `color`. In the geometric calculations, double precision is preferable, although the geometry itself can be described by `floats`. In lighting calculations single precision is sufficient. Feel free to use your 3D vector classes for the fields.

Implement the intersection algorithms for polygons and quadrics described in class. Boxes can be treated as collections of polygons. Recall that the a cone without caps with radii r_1 and r_2 of height h can be defined by the equations $x^2 + z^2 = ((r_2 - r_1)y/h + r_1)^2$, $0 \le y \le h$, and the cylinder of height h and radius r by equations $x^2 + z^2 = r^2$, $0 \le y \le h$. Thus, intersect the rays with the sides of cones and cylinders we use the same approach as for quadrics. The `cylinder` and `cone` also have disks at the ends. Find the algorithm for intresecting a ray with a disk yourself.

Dummy lighting. As we do not have any lighting yet, we should chose a way to define the color of the ray after we found the intersection point with an object. If the ray did not hit any object, use a background color (e.g. blue). Otherwise, choose a function of coordinates of the intersection point to paint the ray. The simplest choice would be simply Z-coordinate with appropriate scaling, but you can be more creative, provided that your choice allows to distinguish the shapes in the picture.