Ray Tracer

Due date: April 26, 2012

1 Overview

In this assignment you will implement the camera and several primitive objects for a ray tracer, and a basic ray tracing algorithm. The input language of the ray tracer will be a subset of the geometry file format of POV-Ray raytracer.

2 What your program should do

Your program should read a file and create an image of the scene described in the file. You can store the image in a texture and display it using the OpenGL code provided on the page (this is not required however – you can also write a command-line utility that simply writes an image file to the disk). You can write out images using TGA code provided for the texture-mapping assignment, but using screen capture is also ok.

3 File format description

The input file format is a subset of the POV-Ray file format. When in doubt, consult the POV-Ray manual. Installing POV-Ray on your computer is strongly recommended. The file in POV-Ray format is a sequence of object descriptions. You should implement the following POV-Ray objects: `box, sphere, polygon and cylinder`.

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 { <0,0,0>, 1  } // a sphere of radius 1 centered at the origin
```

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: finishes, pigments and transformations.

Sphere. The syntax for a sphere is:

```
sphere { <CENTER>, RADIUS }
```

where `<CENTER>` is a 3d point, and `RADIUS` is a real value specifying the radius. 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 { <0,0,0>, 1.0
  scale 10
  translate <0,25,0>
}
```

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

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

```
cylinder {
  <POINT1>, <POINT2>, RADIUS
}
```

where `<POINT1>` and `<POINT2>` are the centers of the end caps, and `RADIUS` is the cylinder radius.

```
cylinder { <0,0,0>, <0,3,0>, 2}
```

is a cylinder of height 3 with a radius of 2, and base centered at the origin in the XZ plane.

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 (extra credit). A polygon with N vertices is defined by:

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

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

Camera. The camera definition describes a camera used for viewing the scene. The camera is defined by the following parameters: `location, look_at, angle, up, right`. The `location` is simply the X, Y, Z coordinates of the camera. The default `location` is ¡0,0,0¿. The `lookat` method of positioning the camera have been discussed at a previous lecture in the context of OpenGL. In addition to the location ("eye" in the `gluLookAt` function call), we specify the the `up` vector and the `look_at` point (the "center" argument of `gluLookAt`). When the `look_at` point is specified, the `up` vector should be forced to be perpendicular to the viewing direction, that is, the component of `up` parallel to the viewing direction should be subtracted from it and it should be normalized to 1. By default, the `look_at` point is 1 unit in Z direction from the location. The `angle` keyword followed by a float expression specifies the (horizontal) viewing angle in degrees. The `right` vector plays a limited role: we use the ratio of the length of this vector to the length of the `up` vector to determine the aspect ratio of the image. The default values are:

```
up <0,1,0>
right <1.33333,0,0>
```

The main use of the camera model in ray tracing is to compute pixel rays. For the simplified model that we use, formulas are included in the lecture notes. Warning: POV-Ray has a relatively complicated camera model with many interdependent parameters; we are using a small subset. 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 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 points along the X axis.

In addition to the standard parameters, a camera may have a transformation attached to it; the parser reads it, if there is one, but you are not required to implement it.

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 in any order. The argument of `rotate` is a vector with three components, which are angles of rotation around x, y and z axes in degrees. Example

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

The `scale` is a nonuniform scale and its argument is a vector of three scale factors in x,y and z directions. The `translate` takes the translation vector as an argument. The parser reads all transformations and converts them to 4 by 4 matrices.

It is possible to transform each object before rendering, and this is the most efficient approach. However, deriving the necessary transformations of the object parameters may be tricky. A simpler approach is to transform the ray to the coordiante system in which the object is defined. This means that each time we intersect a ray with an object that has a transformation attached to it, we need to apply the *inverse* of the transformation to the ray. Thus, it makes sense to store both the transformation matrix and its inverse together with the object. A routine for inverting matrices is provided. To transform the ray $p + vt$, starting at p and with direction vector v, apply transformations to $[p_x, p_y, p_z, 1]$ and $[v_x, v_y, v_z, 0]$. Do not forget to transform the intersection and the normal back to the original system!

Light sources. Light sources have no visible shape of their own. We will use only one type of light sources: point lights. The syntax for a light source is:

```
light_source { <X, Y, Z> color rgb<R, G, B> }
```

Where X, Y and Z are the coordinates of the location and R, G, B are the components of the color in the range 0 to 1. For example,

```
light_source { <3, 5, -6> color rgb<0.5, 0.5, 0.5> }
```

is a 50% white light at X=3, Y=5, Z=-6.

Pigment. The color or pattern of colors for an object is defined by a pigment statement. For example,

```
sphere { <0,0,0>, 1
  pigment {color rgb<1.0, 0.0, 0.0> }
}
```

defines a red sphere. The color you define is the way you want it to look if fully illuminated. The parameter is called `pigment` because we are defining the basic color the object actually is rather than how it looks.

We will implement only the simplest pigment type, which uses the `color` statement to specify the pigment. As in the case of light sources, the color statement has the form `color rgb<R, G, B>`. In addition, the color statement may have the form `color rgbf<R, G, B, F>`, where `F` stands for "filter", and determines how transparent the object is; the actual transparency for each color component is determined by the product of the filter value and the color component value. E.g. `<1,0,0,0>` means an object that allows only the red light through. The default filter value is 0, which means that the surface of the object is not transparent. Transparency implementation is extra-credit, you can ignore it.

Finish. The finish statement, which defines other parameters of the lighting model. In general, the finish statement contains a number of items, that control ambient, diffuse and specular reflection. A complete finish statement looks like this:

```
finish {
  ambient 0.2
  diffuse 0.7
  phong 0.5
  phong_size 25
  metallic 1
  reflection 0.8
}
```

Any item can be omitted. When it is the case, it is assigned a default value. The default values are listed below:

```
finish {
  ambient 0.1
  diffuse 0.6
  phong 0.0
  phong_size 40
  metallic 0
  reflection 0.0
}
```

Thus, `finish {}` is exactly equivalent to the expanded statement above. The `ambient` component has effect only if there is a non-zero ambient light, which is specified using a special statement like this:

```
global_settings {ambient_light COLOR}
```

where COLOR is of the form `rgb <r,g,b>` as for a light source. The default is no ambient light.

Refraction (extra credit). For closed transparent objects (spheres, cones, cylinders, boxes) one can define the index of refraction. The keyword used to define it is `ior`, which as to be inside a special object modifier `interior`.

```
sphere{ 1
  pigment{ rgbf<1.0,1.0,1.0,0.5> }
  interior {ior 1.5}
}
```

The default ior value of 1.0 will give no refraction. The index of refraction for air is 1.0, water is 1.33, glass is 1.5, and diamond is 2.4. For the index of refraction to have any effect, the object should be transparent, that is, the color in the pigment statement should have a "filter" component different from 0.0.

Complete lighting formula. The complete formula is written for one component (red); the formulas for the other two components are obtained by replacing red with green or blue. Assume that for a given point there are M visible lights, with colors r_i, g_i, b_i, $i = 1 \ldots M$. Assume that the pigment for the object was given by `pigment { color rgbf <`m_r`,`m_g`,`m_b`,`m_f`>}`, and the finish is

```
finish {
   ambient k_amb
   diffuse k_diff
   phong k_spec
   phong_size p
   reflection k_refl
}
```

Further, assume that the direction ti the i-th light source is L_i, the direction to the eye (or, in the case of a recursive ray, the incoming direction) is V, the normal is N, and the reflected direction is R.

The total red intensity at the point is given by

$$(1-m_f)k_{amb}m_r r_{amb} + \sum_{i=1}^{M}(1-m_f)r_i\Big(k_{diff}m_r(N\cdot L_i) + k_{spec}\{m_r \text{ or } r_i\}(R\cdot L_i)^p\Big) + k_{refl}r_{refl} + m_f m_r r_{trans}$$

In the formula above r_{refl} and r_{trans} are the red components of the recursively traced reflected and refracted rays. Note that for a transparent object ($m_f \neq 0$) the ambient component and the diffuse component are attenuated and the specular and reflected components are not. This is a peculiar POV-Ray feature and this may be not true in other ray tracers. In addition for the the specular component is scaled by either m_r (the material red) if `metallic` keyword is present in the finish, or by r_i (the light red) if it is not.

4 Ray tracing

Implement the basic ray tracing algorithm described in class; for each point, shoot rays to all light sources, the reflected ray if `reflection` is not zero) and the refracted ray (if the "filter" component of the pigment is not zero). Refraction implementation is extra credit, and is not required.

The recursive spawning of rays is terminated when the user-specified maximal depth is reached, or when the ray weight is below a threshold. The weight is computed as follows: for pixel rays (the rays starting at the camera) it is set to 1. Each reflected or refracted ray is assigned a weight equal to the product of its current weight with the coefficient with which its contribution enters the summation in the lighting equation.

5 What to turn in

Create test images from the file that is provided on the Web page.

6 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. These suggestions are for C++ programming. However, similar style can be maintained in C code.

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. Unless you already have it, write a small library of standard vector functions (addition, multiplication by a constant, dot product, cross product etc.) using the code provided with the parser as the starting point.

Parsing the geometry file. To simplify your task, a parser is provided. Currently, the parser simply prints out the parameters of the objects that are parsed. You need to add code to it to create the objects. The parser uses a standard approach: there is a function `getToken`, that skips the comments and returns the next token in the input file. A token is either a reserved keyword, a floating point number, a comma, or one of the symbols `{`, `}`, <, >. 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, vectors etc. Only simple error processing is done: if there is a syntax error in the file, the parser prints a message and aborts.

In general, each object type (sphere, cylinder, polygon) should correspond 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.

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 `double`s for all fields except `color`. In the geometric calculations, double precision is preferable, although the geometry itself can be described by `float`s. In lighting calculations single precision is sufficient. Feel free to use your 3D vector classes for the fields instead of `double[3]`.

Start with implementing a single simple object (e.g. spheres) and all other parts of the raytracer.

Dummy lighting. It is advisable not to debug the initial code with lighting and recursive ray tracing computations turned on. For simple pseudo-lighting, if the ray did not hit any object, use a background color (e.g. blue), otherwise, use a function of coordinates of the intersection point to set the ray color. The simplest choice would be simply Z-coordinate with appropriate scaling, so that the resulting value is 255 for the points closest to the camera and 0 for the most distant points, but you can be more creative, provided that your choice allows to distinguish the shapes in the picture. Another simple approach is to assign a color to each shape.