Ray Tracing exercise

TAU, Computer Graphics, 0368.3014, semester A 2009/2010

Overview

The objective of this exercise is to implement a ray casting/tracing engine.
Ray tracing is a way to visualize 3D models.
A ray tracer shoots rays from the observers eye through a screen and into a scene of objects. it calculates the rays intersection with the objects, finds the nearest intersection and calculates the color of the surface according to its material and lighting conditions.

The feature set you are required to implement in your ray tracer is as follows (by order from easy to hard):

- Background (2 points)
  - Plain color background
  - Background image
- Control the camera and screen (5 points)
  - simple pinhole camera
- Display geometric primitives in space: (20 points)
  - rectangular planes
  - circular planes (discs)
  - spheres
  - boxes (cubes)
  - Cylinders (tubes)
  - Triangular meshes (ply2, off formats)
- Render Surfaces:
  - Simple materials (ambient, diffuse, specular...) (9 points)
    - According to the lighting equation studied in class.
  - Basic "checkers" pattern (7 points)
  - Image Textures (8 points)
- Basic lighting (13 points)
  - parallel directional light
  - omni-direction point light
- Basic hard (sharp) shadows (9 points)
- pixel super sampling (5 points)
- Reflection - reflecting surfaces. mirrors and shiny objects (8 points)
- Advanced Lighting
  - One of the following (7 points)
    - Area light simulated by multiple point lights
    - Monte-carlo Lighting
  - Hemispherical Lighting (7 points)

Environment features:

- A parser for a simple scene definition language. (base implementation will be supplied)
- A simple GUI (will be written for you)
- Render the scene to an image file (in the GUI you're going to get).

Additional features which will grant you a bonus:

- Acceleration

    - Uniform grid speedup
    - Octree or BSP for speedup
- Monte-carlo path tracing

Scene definition language

The 3D scene your ray tracer will be rendering will be defined in a scene definition text file. The scene definition contains all of the parameters required to render the scene and the objects in it.
The specific language used in the definition file is defined in detail in appendix A.
In case the parser encounters an object or a parameter it does not recognize it needs to output a warning about it and continue.
A sample scene definition file can be found here.And in the examples

Features

Background

The background is either a flat color or an image scaled to the dimensions of the canvas. feel free to use ImageData.scaledTo() in order to fit the image to the canvas size.

Camera

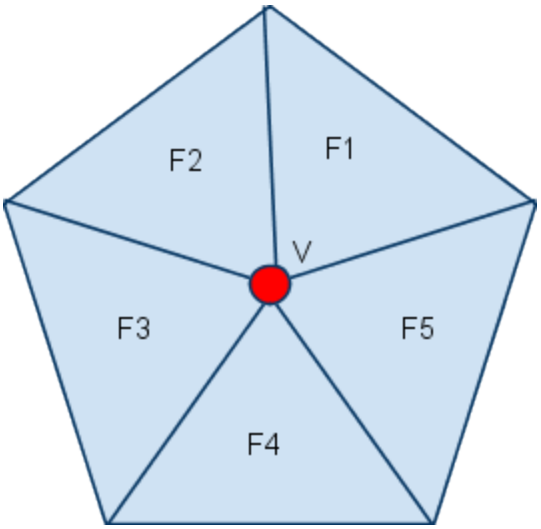
The camera is a simple pinhole camera described in the lecture slides. see the appendix for further notes.

Geometric primitives

Intersection calculations for a plane and a sphere are described in the lecture notes. more explanations about intersections can be found in these links.
plane:
http://www.blackpawn.com/texts/pointinpoly/default.html
notice that this page discusses a triangle and we need a rectangle and a circle. you will need to perform the necessary modifications.
sphere:
http://www.devmaster.net/wiki/Ray-sphere_intersection
cylinder (uncapped):
http://www.gamedev.net/community/forums/topic.asp?topic_id=467789

Triangular Meshes

Can be regarded as a collection of triangular planes stitched together with the same material. In order to achieve smooth shading, the normal of each vertex should be calculated as the average of its neighboring facet's normals. I.e. :



$$N(V) = \frac{1}{|V \in F|} \sum_{F}^{V \in F} N(F)$$

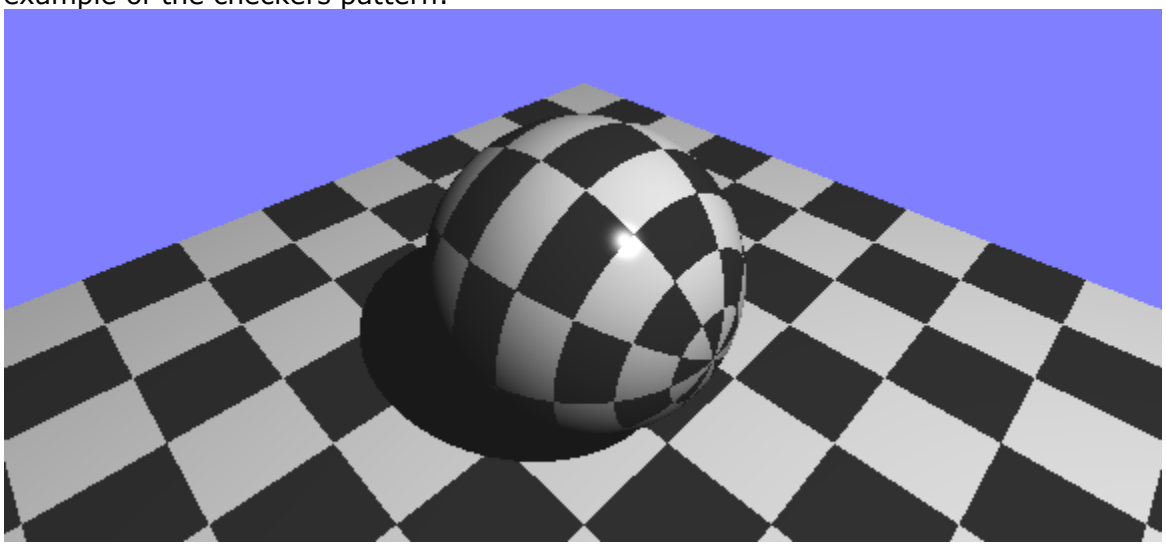
Intersection calculations should be performed for each triangle separately. Note that without proper accelerations it would take forever to render high-polygon meshes.

Materials

You need to implement the lighting formula from the lecture slides. The material of a surface can be one of 3 possible options:

- Flat material with ambient, diffuse, specular reflections, emission parameter and a shininess parameter (the power of V.R). the first 4 parameters are RGB colors.
- Checkers material - a checkers pattern over the surface with alternating colors. the diffuse color is taken from the two diffuse colors defined for the material.
- Texture material - the diffuse color is taken from an image file.

An example of the checkers pattern:



Parametrization

In order to implement the last two materials you will need to define a **parametrization** over the surface rendered.
Say a ray intersects a surface at 3D point p=(x,y,z) which is on the surface. A parametrization takes this point and transforms it into a 2D point e=(u,v) that can be used to access a flat texture.
For a rectangular plane, the parametrization is trivial. use the method described in the link above to derive baricentric coordinates.

For a sphere, the parametrization is derived from the concept of Spherical Coordinates. Essentially for a point p=(x,y,z), calculate phi and theta, normalize them to the appropriate use them as (u,v).
A good way to see if you got this right is using a sphere with a gradient texture such as the one in reference usphere_tex_2.

The parametrization of the Disc and Cylinder are similar in concept to the method used for the sphere. You should base your parametrization on two vectors, the first vector is the normal of the disc or the direction vector of the cylinder
the second vector needs to be some vector which is orthogonal to the first vector. A simple way to find such a vector (there are infinitly many) is to arbitrary pick an axis (for instance the X axis) and taking its cross product with the first vector. Pay attention to end cases of this procedure.
Once you have these two vectors, the parametrization is linear in the first and angular to the second.

Note that for Meshes we don't require parametrization.

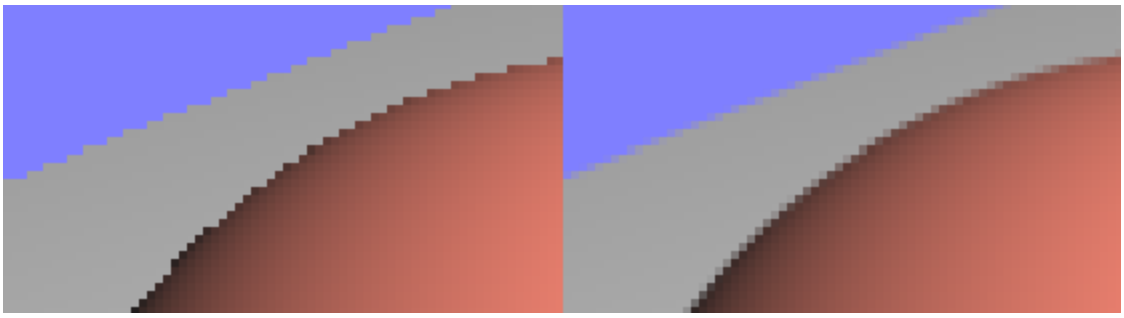
Basic lights and shadows

For basic lighting you need to implement two of the light sources discussed in class. A directed light and point light. Notice that for the point light the user can specify attenuation. Every light source has its own intensity (color) and there can be multiple light sources in a scene.

Shadows are caused where objects obscure a light source. In the equation they come to effect in the $\mathbf{S_L}$ term. to know if a point 'p' in space (usually on a surface) lays in a shadow of a light source you need to shoot a ray from p in the direction of the light and check if it hits something. If it does, make sure that it really hits it before reaching the light source and that the object hit is not actually the object the ray emanated from.
Some common mistakes may cause spurious shadows to appear. Make sure you understand the vector math involved and all the edge-cases.

Super Sampling

if you only shoot one ray from each pixel of the canvas the image you get will contain alias artifacts (jugged edges). A simple way to avoid this is super sampling. With super sampling you shoot several rays from each pixels, for each such ray you receive the color it's supposed to show. Then you average all these colors to receive the final color of the pixel.
In your implementation you will divide every pixel to grids of 2x2 or 3x3 etc' of sub pixels and shoot a ray from every such sub pixel.
Another explanation of super sampling can be found in this page:
http://everything2.com/e2node/supersampling



Reflection

This is where ray-casting becomes ray-tracing. if a material has reflectance (K_S) of greater than 0 than a recursive ray needs to be shot from its surface in the direction of the reflected ray.
Say the ray from the camera arrives at point 'p' of the surface at direction V (as in slide 35). using the normal N at point p, you need to calculate vector R_V (not R) which is the reflected vector of V. this can be done using simple vector math and is similar to the calculation done when calculating specular lighting. Using R_V, you need to recursively shoot a ray again,this time from point p. Once the calculation of this ray returns you multiply the color returned by the reflectance factor K_S and add it the the color sum of this ray.

Advanced Lighting

An area light is a light source that has an area greater that 0 and hence creates soft shadows.
The simplest way of implementing area light is simulating it using a grid of point lights.
In this exercise the area light you are required to implement is a rectangular uniform grid area light. Inside this rectangle is a grid of NxN point lights, each with intensity of I_L/N^2. the total intensity of the light source will still be I_L.
Notice that using such an area light means that for every rendered point you now need to shoot N^2 rays, one ray in the direction of each and every one of the point light source it is made of. This is instead of a single ray you normally need to shoot at a simple light source. Predictably, this will slow the rendering down significantly.

Monte-Carlo Lighting

In Monte-Carlo lighting the light origin is not a single point but a spherical cloud of possible points. For each ray hit you should randomly select numerous candidate points in the sphere as light sources and average their individual effect. The further away the candidate point is from the center of the sphere the weaker it's influence is.

Hemispherical Lighting

This lighting model relies exclusively on the hit surface's normal. The scene description defines two colors - one for skies and one for the ground. The resulting color for each hit is

a linear interpolation between these colors based on the dot product between the (normalized) normal and the "up" vector (which points to the skies).

Bonus: Intersections Accelerations

Choose between implementing either Octree or BSP tree for speeding up the rendering process.
When rendering without any acceleration, every intersection check iterates over all the objects in the scene to find if a ray intersects any of them. The idea behind the acceleration structures is to divide space in such a way that allows you iterate over only a subset of the objects in the scene. This allows rendering of large scenes in reasonable times.

Bonus: Monte-Carlo path tracing

Render your scene with global illumination using a Monte-Carlo method. Upon ray hit recursively trace random rays in the reflecting hemisphere. When recursion reaches a predefined limit return the light intensity at that point and accumulate all results. Pay close attention at how you combine the results as not to create a bias towards noisy results. Note that this should be an alternative to ray tracing.

Sample Scenes

To exemplify the workings of your ray tracer you need to compose two scenes containing multiple objects which demonstrate the array of features implemented. You need to supply the textual definition file, any texture files needed and a screen capture of the rendered image.
The scenes need to be original and creative. Bonus points will be given to the best scenes submitted.

Reference scenes

In this section you will find sample scenes and the way they are supposed to render in your ray tracer. Your implementation may vary from the supplied image in little details but in general the scene should look the same.

basic camera positioning:

reference scenes batch no. 1

primitive surfaces

reference scenes batch no. 2

Please notice that the position and orientation of the texture is highly dependent of the parametrization you implement.
It is OK if in your rendering the textures are positioned differently than how they are positioned in these examples.
It is not ok however if the textures are doubled or partially or multiply shown in some way. A difference in the frequency of the texture is not ok.

complex scenes and shadows
reference scenes batch no. 3

Area Light, reflections examples
reference scenes batch no. 4

Soon - meshes and advanced lighting

Implementation

The preferred implementation language is Java. if you really want can also implement in C/C++.
We do not not supply a skeleton GUI and parser in C/C++, you will have to write these on your own. It is not OK to submit a command line implementation.

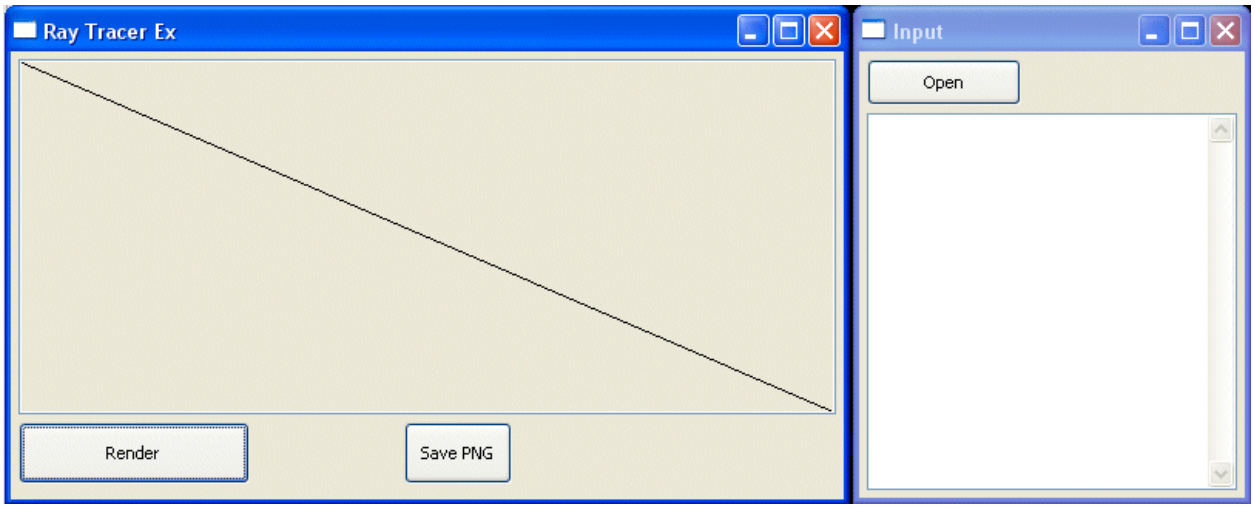
Think before you code
before you start coding you should think carefully about the structure and design of your system. Take the time to define interfaces, classes and interactions between objects. Implementing a ray tracer poses an array of opportunities for creating good OOP design. make sure you take advantage of them.

General Hints
- Write a good 3D vector class to handle 3d coordinates and vectors (and RGB colors)
- One of the first things you need to do is map the canvas coordinates you receive from the renderer to the scene coordinate system. you do this using the camera parameters.
- Before plotting a pixel, make sure that it's color does not exceed the range of 0-255 for every color channel.

Write incrementally. We suggest the following implementation milestones:
- Camera on the Z axis, scene includes a single rectangle at the origin (0,0,0)
- same as above, only with the rectangle at an arbitrary point and in different orientations.
- move the camera around, rotate it, change distance and size.
- basic lighting - point and directed light
- add the sphere primitive
- shadows
- full materials implementations
- parametrization
- all other features...

Startup implementation
These files are an initial framework for you implementing the GUI and parser.
Feel free to change them anyway you like.
You will need to have **SWT** installed for these to work.

RayTracer.java
A bare-boned GUI you for use as a starting point. The GUI has two windows, one is the rendering window, containing the canvas which eventually holds the rendered image.
The other is a text edit box where you can write the scene definition or load it from a file



Parser.java
A parser for the scene definition syntax. You should inherit from this class, overriding its methods in order to create a scene from the definition text.

Download from this link

Submission

- Submission is in **Pairs** (unless approved by us).
- **Before submission be sure you check Updates for stuff you may have missed.**
- You need to make the submission by 23:59 of the submission date.
- On submission date, send a zip file containing all of the following items to the email address listed below
    - Complete source code
    - A compiled JAR file. The JAR file needs to have its extension renamed so it won't be filtered by the email server.
    - Documentation: A 1-3 pages document (doc/docx/pdf) explaining the structure of your code, bonuses implemented, and anything else needed to make it work.
    - The examples you created, including the text file, all the textures needed and the rendered result.
- After sending the submission email make sure you receive confirmation that your submission was received. A confirmation will be sent to you within 24 hours.

Submit your work to chen.goldberg+TAU_CG@gmail.com

Appendix A - Scene definition Language

The scene definition language aims to define objects such as "scene", "camera", "sphere", "rectangle" etc'. An object definition starts with a line containing the type of the object followed by a colon. Following the object's name are lines which define parameters which govern the object looks, From basic parameters such "center" and "radius" to other more complex parameters. For a given object some parameters are compulsory and some are

optional and receive some default value.
A parameter value can be in one of 3 forms:

- a string, for example texture file name
- a single floating point number, defining some scalar quantity
- a series of 3 floating point numbers separated by spaces, defining a vector or a coordinate of a point in 3d space or a color. the order of the numbers is (x, y, z) or (red, green, blue) for colors.

A few more notes:

- The scale for each value of a color is from 0 (no color) to 1 (most color) for display it needs to be stretched back to 0-255.
- When coordinates in space are specified they are given in the scene coordinate system. the scene coordinate system is a right-hand coordinate system. the units that will be used in most examples are in the range of [-2, 2] around the origin.
- Every parameter defined below has its type in brackets right after its name.
- parameters that are obligatory are marked in **red**
- parameters that are of type "vector" must be normalized to be of length 1 by your code. (they may not be of length 1 in the examples for ease of writing)
- Notice the default values of optional parameters. You will need to hard-code these values as defaults in your code.
- External files (e.g. textures, meshes) **must be located in the same directory of the loaded scene file.** That is the scene file should not contain either relative or absolute paths - just a filename.

Examples for scene definition files can be found in the Reference scenes section above.

`scene:`
defines global parameters about the scene and its rendering.

- **`background-col`** - (rgb) color of the background. default = (0,0,0)
- **`background-tex`** - (string) texture file for the background. default = null. Just a filename (see note above)
- **`ambient-light`** - (rgb) intensity of the ambient light in the scene. $\mathbf{I_{AL}}$ from slide 36 in the lecture notes. default = (0,0,0)
- **`super-samp-width`** - (number) controls how fine is the super sampling grid. If this value is N then for every pixel an NxN grid should be sampled, producing N*N sample points, for every pixel, which are averaged. default = 1.
  This number is truncated to an integer.
- **`use-acceleration`** - (number) should be 0 or 1. Enable (1) or disable (0) the Octree or BSP tree acceleration. When this parameter is 1 in a complex scene there should be a significant increase in performance. default = 0.
- **`mc-path-trace`** - (number) should be 0 or 1. Enable (1) or disable (0) the Monte-Carlo path tracing rendering
- **`mc-path-trace-rec`** - (number) the maximum level of recursion allowed
- **`mc-path-trace-num`** - (number) the number of random rays to track

`camera:`

- **`eye`** - (3d coord) the position of the camera pinhole. rays originate from this point. **P** point from the slides
- **`direction*`** - (vector) the explicit direction the camera is pointed at. **towards** vector from the slides.
- **`look-at*`** - (3d coord) this point along with the eye point can implicitly set the camera direction.

- **`up-direction`** - (vector) the "up" direction of the camera. **up** vector from the slides
- **`screen-dist`** - (number) the distance of the screen from the eye, along the direction vector. **d** from the slides
- **`screen-width`** - (number) the width of the screen, (in scene coordinates of course) this in effect controls the opening angle of the camera. *xfov* from the slides. default = 2.0

A few notes you should about the camera:

- You need to either specify a `direction` or a `look-at` point but not both. specifying a look-at point implicitly set the direction and if you set the direction there is no need to specify a look-at point. The direction can be calculated using the eye and look-at points.
- the direction and the up vector of the camera need to be orthogonal to each other (90 degrees between them) however the vectors specified in the file may be Not orthogonal. you need to pre-process these vectors to make them orthogonal. This can always be done if they are not co-linear. (hint- use a cross product to find the right vector and another cross product to find the real up direction)
- only the screen width is specified. the screen height (*yfov*) needs to be deduced according to the aspect-ratio of the canvas.

Lights

There can be multiple sources of light in a scene. each with its own intensity color, each emitting light and causing shadows.

All light objects can take the following parameter:

- **`color`** - (rgb) the intensity of the light. $\mathbf{I_L}$ from slide 36. default = (1,1,1) - white.

`light-directed:`

A directed light originates in infinity and heads in a single direction

- **`direction`** - (vector) the direction the light is coming from.

`light-point:`

A light emitted from a single point

- **`pos`** - (3d coord) the point where the light emanates from.
- **`attenuation`** - (3 numbers- kc,kl,kq) the attenuation of the light as described in slide 11 of the lecture notes. default = (1,0,0) - no attenuation, only constant factor.

`light-area:`

An Area light is a rectangle which emits light in every direction. it can be simulated by a grid of point lights. see above and the slides for explanation.

- **`p0`**, **`p1`**, **`p2`** - (3d coord) 3 points defining the rectangle. see `rectangle` surface definition.
- **`grid-width`** - (number) controls the density of the grid of point lights. If this is N then the grid of point lights will be a grid of NxN point lights, each of intensity I_L/N^2. The number should be truncated to an integer.

`light-area-mc:`

An area light emitted from a spherical cloud of probable points

- **`pos`** - (3d coord) the center of the sphere

- **`radius`** - (number) The radius of the sphere. Default = 2.
- **`candidates`** - (number) The number of points sampled each time. Default = 9.

`light-hemisphere:`
A light emitted from the skies and ground around.

- **`ground-color`** - (rgb) the intensity of light emitted from the lower hemishpere. default = (1,1,1). Note that this replaces the inherited color parameter.
- **`sky-color`** - (rgb) the intensity of light emitted from the upper hemishpere.default = (0,0,0)
- **`up`** -(vector) the direction of the skies. default = (0,1,0).

Surfaces

"Physical" surfaces that is rendered in the scene.

Every surface has a material. the material can be either a flat color material or a textured one. The following parameters may occur on every type of surface:

- **`mtl-type`** - (string) the type of the material. can be one of the following: "`flat`", "`checkers`" or "`texture`", without the double quotes. default = "`flat`".
- **`mtl-diffuse`** - (rgb) the diffuse part of a flat material (**K_D**) default = (0.7, 0.7, 0.7)
- **`mtl-specular`** - (rgb) the specular part of the material (**K_S**) default = (1, 1, 1)
- **`mtl-ambient`** - (rgb) the ambient part of the material (**K_A**) default = (0.1, 0.1, 0.1)
- **`mtl-emission`** - (rgb) the emission part of the material (**I_E**) default = (0, 0, 0)
- **`mtl-shininess`** - (number) the power of the (V.R) in the formula in slide 36 (**n**) default = 100
- **`checkers-size`** - (number) if the type is "`checkers`" then this controls the size of a single square in the checkers pattern. The checkers pattern is made of interleaving color squares. The diffuse color of the surface is one of `checkers-diffuse1` or `checkers-diffuse2` according to the pattern. default = 0.1 for a 10x10 checkers board.
- **`checkers-diffuse1`** - (rgb) the first checkers diffuse intensity. default = (1, 1, 1) - white
- **`checkers-diffuse2`** - (rgb) the second checkers diffuse intensity. default = (0.1, 0.1, 0.1) - dark gray.
- **`texture`** - (string) if the type is "`texture`" then this is the name of the texture file. you only need to support PNG files. Just filename. default = null.
- **`reflectance`** - (number) the reflectance coefficient of the material. **K_T** from slide 44. default = 0. - no reflectance.

  Note: Slide 44 and 42 in this lighting presentation mention both transparency and reflectance. In the exercise you only need to implement **reflectance.**
  The constant K_T you get from the definition file replaces the K_S from the slides which is multiplied by I_R, the reflection color.

  In other words, the equation you need to implement is this:

$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L + K_T I_R$$

  Where K_T is the constant you get from the **`'reflectance'`** parameter and I_R is the reflection color.

If K_T is 0, there is no reflection.
You are not required to implement refraction/transparency.
Notice that this is a scalar coefficient and not an rgb value.

Note: When using textures or checkers material only the diffuse intensity is affected. the ambient, specular, emission and shininess of the material are set exactly as with flat color using the `mtl-xxx` parameters.

rectangle:

A rectangle is square plane defined by 3 points - p0, p1, p2. the rectangle is drawn between the two vectors (p1-p0), (p2-p0) when they originate from point p0. notice that the angles doesn't need to be 90 degrees so this is actually a parallelogram, not a rectangle.

- **p0**, **p1**, **p2** - (3d-coord) the points in space. The points must not be co-linear.

disc:

A disc is a circular plane defined by a point, a normal and a radius. the plane originates in the center point, is orthogonal to the normal and spans the area according to the radius.

- **center** - (3d coord)
- **normal** - (vector)
- **radius** - (number)

sphere:

A sphere is defined by a center point and a radius around that center point.

- **center** - (3d coord)
- **radius** - (number)

box:

a box is a 3D hexahedron with pairs of parallel planes. The angles don't have to be 90 degrees. essentially, a box is made out of 6 rectangles. At any time only 3 of them face the camera (possible optimization). a box is defined by 4 points - p0, p1, p2, p3 which make 3 vectors: (p1-p0), (p2-p0), (p3-p0). these vectors define the extents of the box.

- **p0**, **p1**, **p2**, **p3** - (3d-coord) the points in space. No subset of 3 points of these can be co-linear.

cylinder:

A cylinder is a circular tube. It is defined by an origin point where it starts, a direction in which it is headed, a length and a radius.

The cylinder is uncapped. you can add discs to the scene in both ends to close it.

- **start** - (3d-coord) the starting point of the cylinder. The cylinder starts at this point and goes along the length of its direction vector.
- **direction** - (vector) the direction of the cylinder.
- **length** - (number) the length along the direction vector of the cylinder.
- **radius** - (number) the radius of the cylinder.

mesh:

A triangular mesh object. The mesh itself is defined in an external file (see appendix B). The vertex coordinates are translated by the given position parameters (any other transformations should be performed offline on the raw mesh file). Note

- **filename** - (string) mesh filename (just a filename)

- `pos` - (3d coord) translation of the mesh
- `scale` - (number) scaling of the mesh

Appendix B - Mesh file formats

The triangular mesh should be specified in an external file. A typical mesh file is usually composed of two parts: 1) A list of 3d vertices followed by 2) a list of polygons (where each polygon is a list of vertex indices). You'll have to support the parsing of two such formats:

- **OFF** - http://shape.cs.princeton.edu/benchmark/documentation/off_format.html
- **PLY2** - http://www.riken.jp/brict/Yoshizawa/Research/PLYformat/PLYformat.html

You can download low-res mesh files in these formats from the following links:
- http://shape.cs.princeton.edu/benchmark/
- [more to come]

FAQ

Q: What should be done if the definition file defines both a background texture and a background color?

A: use the texture. Unless you are in a reflecting ray. see next Q.

Q: If there are reflections in the scene, a reflected ray may not hit anything and reach the background. If a background image is defined, should the background color of the reflecting ray be taken from the background image?

A: If you reach the background while calculating a reflecting ray, you can use the background color instead of the image. still, the background of the rendered frame should still be the background image defined by `background-tex`.
This is essentially the reason you'll sometimes want to have both background texture and background color defined.

Q: What should I do if the dot products in the light equation are negative?
A: I discussed this in class. essentially **this is up to you. do whatever you think is best.** What you shouldn't be doing is back-face-culling (This was not studied in class so don't worry if you don't know about it yet). Every object defined in the scene needs to be visible if its within the camera frame, even if it is back-facing.

Specifically, reference scenes cylinder_3 and cylinder_4 may look different in your implementation.

Q: What about the attenuation of the point lights in the area light?

A: Use the default attenuation of (Kc,Kl,Kq)=(1,0,0)

Q: How do I make java find the texture files. there is no way to change the current

directory.

A: Use the path from the scene text file you loaded. For example

```
String path = new File(filename).getParent() + File.separator;
```

Q: Can I use javax.vecmath.Vector3d ?
A: You can but you would probably not want to.
javax.vecmath.Vector3d lacks some basic functionality which you will need for this exercise like vector reflection, multiplication and convenient work with RGB values.

Q: What other library classes can I use?
A: You shouldn't need to be using anything other than basic math and vector calculations. all of the external dependencies are taken care of by the supplied wrapper.
If you still think there is something you want to use, send me a message for approval.

Q: How do I calculate R, the reflection vector?
A: This page:
http://www.cs.umbc.edu/~rheingan/435/pages/res/gen-11.Illum-single-page-0.html
contains (among other things) a nice explanation of this.

Q: How can I debug my raytracer? I keep staring at the code but I can't figure out what's wrong.
A: An easy way to start debugging is to find a pixel where you know something is wrong and try to debug the calculation of that specific pixel. Say you found there is a problem at (344, 205), You can start by writing something like this:

```
if (x == 344 && y == 205)
{
    System.out.print("YO!"); // set breakpoint here.
}
```

in your main loop and then setting a breakpoint at that print line. from this point in the execution you can follow what exactly leads to this pixel being the color it is.

Updates

No news good news. Keep coming back!