Ray Tracing exercise

TAU, Computer Graphics, 0368.3014, semester B

Go to the Updates and FAQ Page

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
  - 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)
- Render Surfaces:
  - Simple materials (ambient, diffuse, specular...) (10 points)
    - According to the lighting equation studied in class.
  - Basic "checkers" pattern (7 points)
  - Image Textures (8 points)
- Basic lighting (15 points)
  - parallel directional light
  - omni-direction point light
- Basic hard (sharp) shadows (10 points)
- pixel super sampling (5 points)
- Reflection - reflecting surfaces. mirrors and shiny objects (10 points)
- Area light - soft shadows. (10 points)
  - Simulated by multiple point lights

- 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
- Geometric Primitives - Add a torus primitive.

- Lighting - A more sophisticated and faster area light

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 your 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.

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 slide 35 of lecture notes 4a. see the appendix for further notes.

Geometric primitives

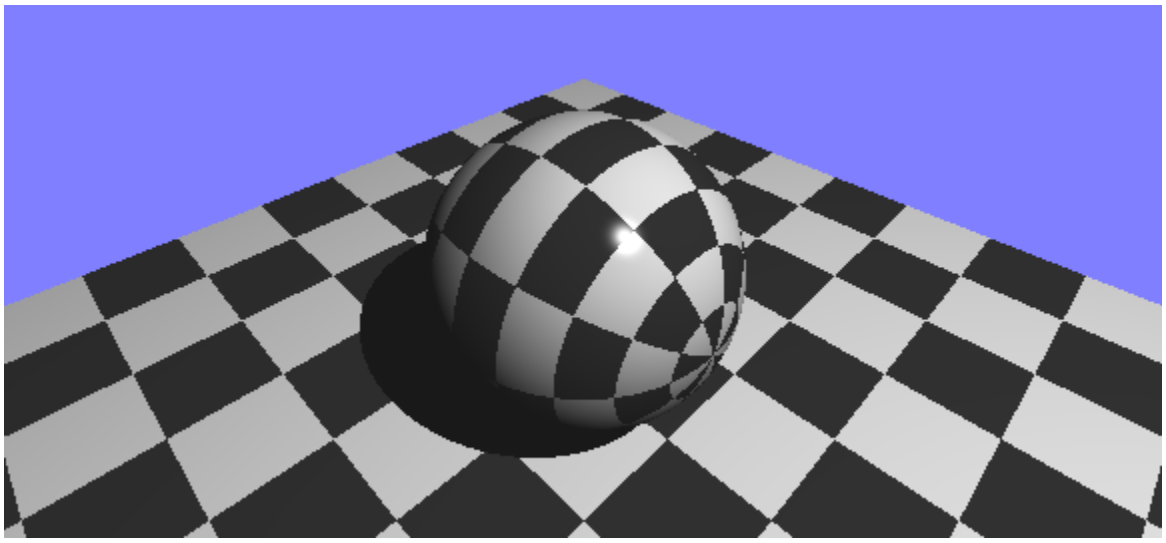
Intersection calculations for a plane and a sphere are described in the lecture notes. more explanations about intersections can be found in these links.
plane:
http://www.blackpawn.com/texts/pointinpoly/default.html
(notice that this page discusses a triangle and we need a rectangle or a circle. you 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

Materials

You need to implement the lighting formula from slide 36 of lecture notes 4b. The material of a surface can be one of 3 possible materials:

- Flat material with ambient, diffuse, specular reflections, emission parameter and a shininess parameter (the power of V.R). the first 4 parameters are essentially colors. Note that each parameters consists of red, green and blue values.
- 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 checkers rendering:



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.

For a sphere, the parametrization is as follows (in pseudo-java):

```
given 'p' on the surface of a sphere centered at 'center' with radius 'radius':
    rp = p - center;
    double v = acos(rp.z/m_radius);
    double u = acos(rp.x/(radius * sin(v)));
    if (rp.y < 0)
        u = -u;
    if (rp.z < 0)
        v = v + PI;
    return new Vector2(u / (2*PI), v / PI);
```

This is a fixed version of the method from this page. Notice that this parametrization can return negative u and v values. your texturing code needs to handle this case.
From this parametrization it is easy to deduce the parametrization of a disc (replace v with the distance from the center) and of a cylinder (replace v with rp.y)
Update: this is not accurate. please read the FAQ question about it.

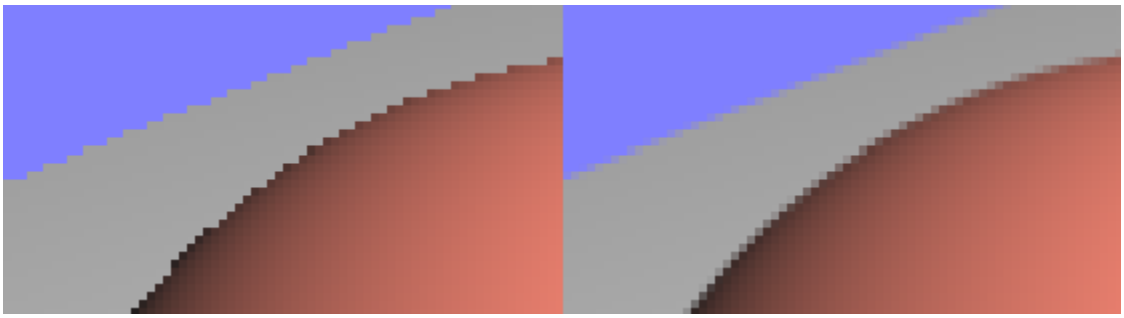
Basic lights and shadows

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 you can specify attenuation.
A light source can have 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 to effect in

the $\mathbf{S_L}$ term. to know if a point 'p' in space (usually on a surface) lays in a shadow from a light source you need to shoot a ray from p to the direction of the light and see if it hits something. If it does, make sure that it really hit it before reaching the light source.

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 32). using the normal N at point p, you need to calculate vector R_V which is the reflective vector for V. this can be done using simple vector math and is similar to the calculation done when calculating the specular lighting. Using RV, 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.

Area Light

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 simply be simulating it using a grid of point lights.
In this exercise the area light you are required to implement is a rectangular 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 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. This will predictably slow the rendering significantly.

Sample Scenes

For the presentation of your ray tracer you need to compose one scene containing multiple objects which demonstrate the array of features implemented. You need to supply the textual file, any texture files needed and a screen shot of the rendered image.
The best images submitted are going to be posted to the course site for all to admire :)

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 (another link)

primitive surfaces

reference scenes batch no. 2 (another link)

NOTICE! read the README file in this zip regarding texturing.

complex scenes and shadows

reference scenes batch no. 3 (another link)

Area Light examples

reference scenes batch no. 4 (another link)

Some Reflections

reference scenes batch no. 5 (another link)

more Will be added soon

Implementation

The preferred implementation language is Java, however you can also implement in C/C++ if you so want.

Note: We will not supply a skeleton file and parser in C/C++, you'll have to write these on your own.

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

- 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 that using the camera parameters.
- invest some time in writing a good vector class to handle 3d coordinates and vectors (and RGB colors)
- Before plotting a pixel, make sure that it's color does not exceed the allowed range of 0-255 for every color. the sooner in the rendering you do this the better.

We suggest the following implementation milestones:

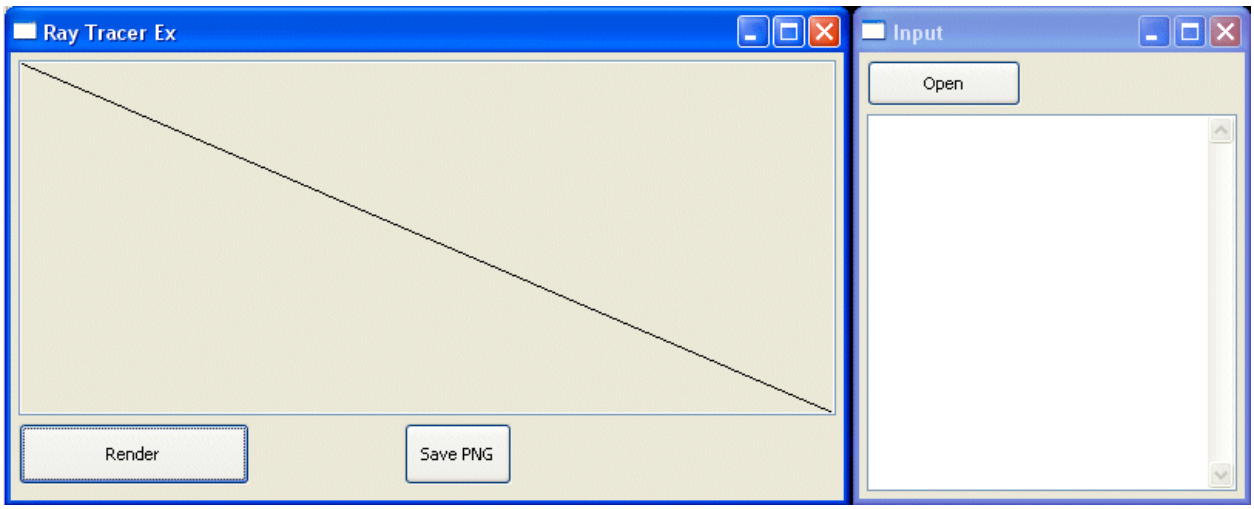
- Camera on the Z axis, scene includes a single rectangle at the origin (0,0,0)
- same as above, only with the rectangle at an arbitrary point and in different orientations.
- move the camera around, rotate it.
- basic lighting - point and directed light
- add the sphere primitive
- shadows
- full materials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

Startup implementation

These files are an initial frame work for your implementation.
feel free to change these files anyway you like.

RayTracer.java

A bare-boned GUI you can use as a starting point. the GUI has two windows, one is the rendering window which contains the canvas which eventually holds your rendered image. The other is a text edit box where you can write the scene definition.



Parser.java

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

Download from:
http://www.cs.tau.ac.il/courses/Computer-Graphics/08b/Exercises/Ex2/rayTraceBare.zip

Submission

As before submission is in Pairs.
In the due date you need to submit the following:

- Full source code, packaged in a zip file
- Executables or JAR file
- For each sample scene you prepared:
    - A textual definition file, texture files
    - A saved image file of the scene rendered
- A short document (2-4 pages) explaining:
    - The structure and design of your code
    - How to operate your application
    - Anything else needed to make it work. Please supply

Please make sure you read the FAQ and Updates page frequently and before your submission.

Appendix A.

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 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:

- Notice that 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.
- for every parameter defined below you can see its type.
- parameters that are obligatory are marked in **red**
- parameters that are of type "vector" must be normalized to be of length 1. (they will usually not be of length 1 in the examples for ease of writing)
- notice the default values of the parameters. you will need to use these values as defaults in your implementation.

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 of the background. default = null.
- **`ambient-light`** - (rgb) intensity of the ambient light in the scene. $\mathbf{I_A}$ from slide 36 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

`camera:`

- **`eye`** - (3d coord) the position of the camera pinhole. rays originate from this point.
- **`direction*`** - (vector) the explicit direction the camera is pointed at
- **`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
- **`screen-dist`** - (number) the distance of the screen from the eye, along the direction vector.
- **`screen-width`** - (number) the width of the screen, (in scene coordinates of course) this in effect controls the angle of the camera. default = 2.0

a few notes you should about the camera:

- You need to either specify a `direction` or a `look-at` point. 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 direction 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 left direction and another cross product to find the real up direction)
- only the screen width is specified. the screen height needs to be deduced according to the aspect-ratio of the canvas.

Lights

Sources of light. there can be multiple sources of light in a scene. each with its own 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 of the directed light.

`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 slide 11 of the lecture notes. default = (1,0,0)

light-area:

An Area light is a rectangle which emits light in every direction. it is simulated by a grid of point lights. see above 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.

Surfaces

A surface 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 ($\mathbf{K_D}$) default = (0.8, 0.8, 0.8)
- **mtl-specular** - (rgb) the specular part of the material ($\mathbf{K_S}$) default = (1, 1, 1)
- **mtl-ambient** - (rgb) the ambient part of the material ($\mathbf{K_A}$) default = (0.1, 0.1, 0.1)
- **mtl-emission** - (rgb) the emission part of the material ($\mathbf{I_E}$) 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 material is one of `checkers-diffuse1` and `checkers-diffuse2` according to the pattern. default = 0.1.
- **checkers-diffuse1** - (rgb) the first checkers diffuse intensity. default = (1, 1, 1)
- **checkers-diffuse2** - (rgb) the second checkers diffuse intensity. default = (0.1, 0.1, 0.1)
- **texture** - (string) if the type is "`texture`" then this is the name of the texture file. you only need to support PNG files. default = null.
- **reflectance** - (number) the reflectance coefficient of the material. This is the second $\mathbf{K_S}$ from slide 39, the one that is multiplied with $\mathbf{I_R}$. default = 0.

  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 don't need to be 90 degrees.

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

Note: The relative order of the vectors defines the normal of the plane and hence to which direction it faces. for example the rectangle defined by:

```
rectangle:
```

```
p0=-2 0 -2
p1=-2 0 2
p2=2 0 -2
```

Will be facing UP (positive y) and will be inverted from the rectangle defined by:

```
rectangle:
p0=-2 0 -2
p1=2 0 -2
p2=-2 0 2
```

which will be facing DOWN (negative y).

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 face the camera. 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, to which direction 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.