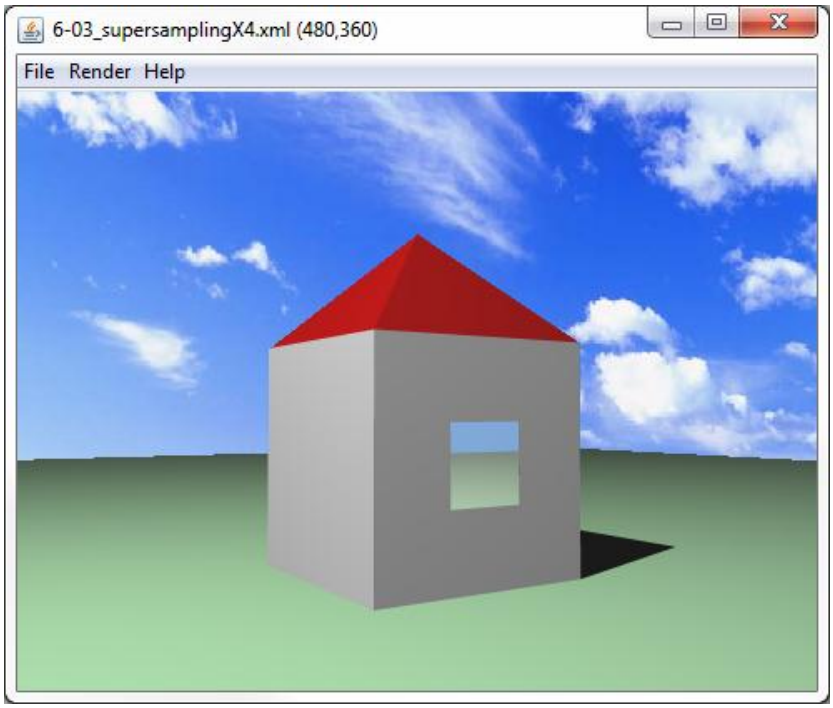
Exercise 3 – Ray Tracing



Overview

The concept of ray tracing: a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost.

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 observer's eye through a screen and into a scene of objects. It calculates the ray's intersection with the objects, finds the nearest intersection and calculates the color of the surface according to its material and lighting conditions. **(This is the way you should think about it – this will help your implementation).**

Requirements

Read this entire explanation before starting. Understand the slides taught in class especially Phong illumination model – this will not be explained in this document!

The feature set you are required to implement in your ray tracer is as follows:

- Background
  - Plain color background (5 Points)
  - Background image (5 Points)
- Control the camera and screen
  - Simple pinhole camera (10 Points)
  - Super sampling (10 Points)
- Display geometric primitives in space:
  - Spheres (5 Points)
  - Triangle meshes (10 Points)
- Basic lighting
  - Ambient light (5 points)
  - Omni-direction point light (5 Points)
  - Spot light (10 Points)
  - Self-emittance (5 Points)
  - Simple materials (ambient, diffuse, specular...) (5 Points)
- Basic hard shadows (10 Points)
- Reflecting surfaces (15 Points)
- Acceleration
  - Hierarchical bounding boxes (10 Points **bonus**)
- Additional points for designing your own **amazing** scene.

Challenge: you will get a bonus (up to 15 points) for implementing advanced extra functionality, like ray refraction or texture mapping (presenting texture from an image file over spheres or triangular meshes), and whatever other advanced method you invent/learn. You're totally on your own here – no explanations or support will be given from us. Wikipedia is your friend. If you choose to go for it, you should add an explanation of what you did in the attached submitted document.

Environment features:

- A parser for a simple scene definition language. (base implementation will be supplied) **Note:** you will not have to implement the XML parser but will have to understand it and work with it – use the forum, your friends etc. This is simply a way to get the data for the scene.
- A simple GUI (will be given to you)
- Render the scene to an image file (in the GUI you're going to get).

Scene Definition

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 the appendix.

Usage

GUI Mode

Your implementation must use the accompanying GUI!

The GUI for the application is composed of a main window which displays the rendered image. All operations are accessed via menu items or keyboard shortcuts (highly recommended).

The user selects a scene from *"File->Select Scene…"* (CTRL + T) This file has to be a valid XML file containing a valid scene structure (see below for more). Then the user can select canvas size by resizing the main window. Finally the user can render the scene to the canvas size using *"Render->Quick Render"* (CTRL + Q). This will cause the selected scene file to be read, processed and rendered.

Note that the file is read and then loaded into the memory this means the file will be read at the point of rendering. Therefore you can edit the file externally, save it, then go back to the application and perform render.

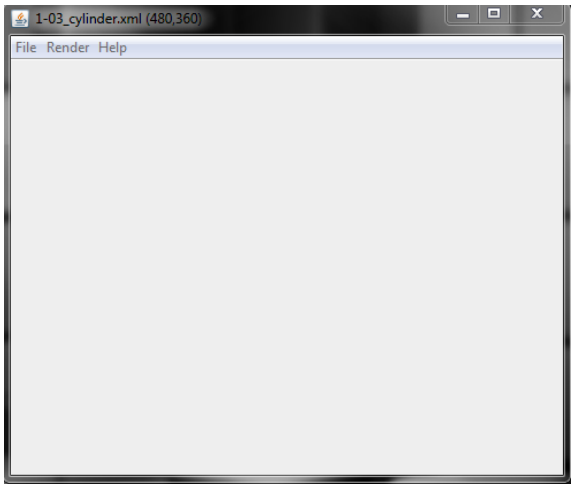


Figure 1 The GUI on start-up

Batch Mode

In addition the application may also receive the scene's filename as an argument causing it to automatically load it on startup: `ex3.jar <input scene filename> <canvas width> <canvas height> <target image filename>` The three first arguments will initialize the scene accordingly. If the last argument is passed then the scene will automatically be rendered on startup, the image saved to the target file and the application will close.

Geometry

Vector Class (Vec)

In ex2 you've implemented this class you should import it to your project under ex3.math.

Point3D Class

You should implement a Point3D class which represents a point. As you know, although both vector and point can be represented by the same three coordinates, there is a big difference between them. Point class should be located in ex3.math.

Your Point3D class should hold 3 coordinates (x,y,z). In addition your point class should be able to perform basic point calculations. *Note: your Point3D Class should not extend your Vec class! but it is ok to have a constructor that receives a vector and vice verse.*

Background

The background of the rendered scene is either a flat color or an image scaled to the dimensions of the canvas.

Camera

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

Geometric primitives

You need to support 2 primitives (spheres and a mesh of triangles). From these 2 primitives you can create a great variety of objects. Intersection calculations for spheres and triangles are described in the lecture notes and recitations.

Sphere

A sphere is defined by a center point and scalar radius. The normal of each point on the sphere's surface is easily computed as the normalized subtraction of the point and the center.

Triangular Mesh

A triangular mesh is just a collection of triangles, which can be interpreted as the surface of a 3d volumetric shape. A mesh class includes a list of triangles and it is your job to place these triangles so that they form a 3D shape. For instance it can describe a box, a pyramid, a parallelepiped or anything you want.

A triangle is defined by 3 points. There is no restriction on how the three points relate but they cannot be linearly aligned. However, each triangle has a front and a back. The front of the triangle is facing outside (in a 3D object) and it is the direction of the triangle's normal. Usually only the front of triangles is rendered.

Lighting

Basic lights and shadows

For basic lighting you need to implement:

1. Ambient lighting – a color that reflects from all surfaces with equal intensity.
2. Omni-direction point light – A point light source that illuminates in all directions, which intensity fades as the distance from the light source increases using the parameters I_0, k_c, k_l, k_q as was explained in class (and its slides).
3. Spot light – a point light source that illuminates in a direction given by a vector D, which intensity fades as the distance from the light source increases using the parameters I_0, k_c, k_l, k_q as was explained in class (and its slides)
4. Self-emittance – each surface can also have a color component that is emitted from within the surface.

Every light source has its own intensity (color) and there can be multiple light sources in a scene. Shadows appear where objects obscure a light source. In the equation they effect in the S_i term. To know if a point p in space (usually on a surface) lies 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 a close intersection with 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.

Materials

You need to implement the lighting formula (Phong) from the lecture slides. The material of a surface should be flat with ambient, diffuse, specular reflections, emission parameter and a shininess parameter (the power of $V \cdot R$). The first 4 parameters are RGB colors.

Super Sampling

If you only shoot one ray from each pixel of the canvas the image you get will contain aliasing artifacts (jagged edges). A simple way to avoid this is with super sampling. With super sampling you shoot several rays through each pixel. 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 the center of every such sub pixel. In fact, the parameter that will control super sampling will be an integer k that tells how many vertical and horizontal rays are casted per pixel ($k = 1$ means no super sampling, $k = 2$ means 2x2=4 rays per pixel etc.)

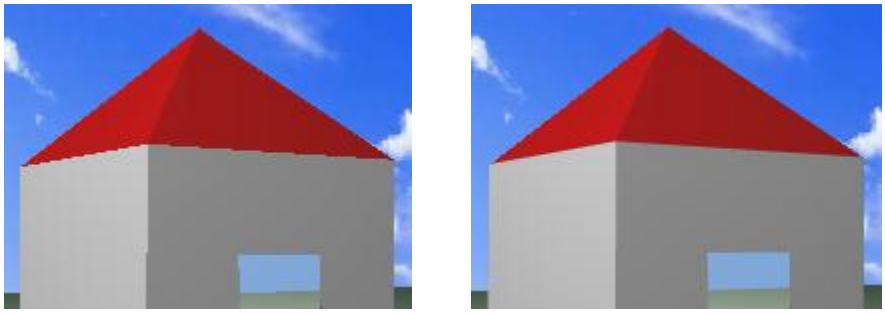


Figure 2 On the left an image rendered without super sampling, on the right with super sampling.

Reflection

This is where ray-casting becomes ray-tracing. If a material has reflectance (K_S) greater than 0, then 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. Using the normal n at point p, you need to calculate vector R_v which is the reflected vector of v. This can be done using simple vector math which was seen in rec2. Using R_V, you 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 color sum of this ray as explained in class.

Acceleration (bonus)

When having many objects in a scene, ray-tracing consumes a great deal of computation time on intersection calculations. Accelerations methods amend this by ruling out intersections prior to the actual intersection test. This prior computation takes a small overhead but on average greatly increases performance (as most intersections are ruled out). One acceleration method that you may implement is the hierarchical bounding boxes method which is addressed in the lecture slides (see also next section). A bounding box for an object is defined as the smallest axis aligned box that contains the object. When rendering, before testing intersection between ray and object, one checks intersection between ray and object's bounding box. If there is no intersection with the bounding box then we know for a fact that there is no intersection with the object. Otherwise we proceed to test for intersection with the object. Note that ray intersection with an axis aligned box can be computed very efficiently. To use this acceleration method one must compute every object's bounding box in preprocessing. Being an axis aligned box, the bounding box can be defined by two points; One point is obtained by taking the minimum over all x,y,z coefficients of the surface points, and the other point is obtained by taking maximum.

Hierarchal Bounding Boxes

Primitives in this ray-tracer are very simple and their intersection computation is not much more expansive than calculating bounding box intersection. Thus a more suitable way for acceleration is to merge groups of bounding boxes together, forming larger bounding boxes and creating a hierarchy. Ruling out intersection with bounding boxes at the top of the hierarchy will be very beneficial. There are many heuristics for forming bounding box hierarchies. Some can be based on fixed space partitioning (quad-tree-like approach) others on clustering methods.

One such method is Hierarchical clustering: For each level in the hierarchy, merge pairs of bounding boxes having the smallest merging error. Merge error can be measured by the amount of empty space introduced by the merge (that is the volume of the parent bounding box minus the volumes of the two children). You should discuss your choice and implementation of the acceleration method in the doc accompanying your submission.

Notes

Getting started

Your renderer class should implement the IRenderer interface. A stub class was already created for you at ex3.render.raytrace.RayTracer. On init, IRenderer receives by parameter the scene description object. You should construct your own scene data structure that is optimized for efficient rendering. The recommended approach is to create a class for each scene object (light, sphere, material, etc) and access them from a Scene class. For reference you should look at the RayCaster initial code given to you at the recitation. After init the GUI may call the rendering method. IRenderer should implement the renderLine method which renders only the pixels at the given line. The reason for this interface is that it allows a convenient way to display the partial rendering in the GUI. This is important as rendering may take a long time.

Recommended Milestones

Design 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. Write incrementally. We suggest the following implementation milestones:

- Place camera on the Z axis, use a single sphere along the negative z-axis as your scene.
- Move the camera around, rotate it, change distance and size - test results.
- Add basic lighting support
- Implement and support all other primitives
- Add support for shadows
- Implement full material properties
- All other features...

Validation

We will provide you with sample scenes for validation 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.

Some general hints:

- There is always an issue with 'double' calculation! Once implementing the exercise, you will get some artifacts that the returning ray will hit the object it already hit multiple time – to overcome this you can use a tolerance value or move the ray just a little ahead before continuing to the next intersection.
- Use the 3D vector class to handle vectors and RGB. Add a 3D point class.
- 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 its color does not exceed the range of 0-255 for every color channel.
- It makes sense to have an Object3D super class for trimeshes and spheres, and a Light super class for omni-directional lights and spotlights.

Submission:

Everything should be submitted inside a single zip file named
<Ex##> - <FirstName1> <FamilyName1> <ID1> <FirstName2> <FamilyName2> <ID2>.zip
For example : 'Ex3 Bart - Cohen-Simpson 34567890 Darth Vader-Levi 12345678'

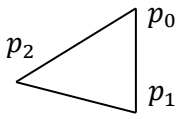
The zip file should include:

- A "proj" folder with all Eclipse project files and Java sources, in the correct directory structure. If you're not using Eclipse then you can submit only the Java source files, but do include everything necessary to run the application.
- A "scene" folder with scenes you created, including description XML files and rendered results in (best in PNG format).
- Compiled runnable JAR file named "ex3.jar" (no folder).
- A short and concise document where you can briefly discuss your implementation choices and bonus features (no folder).

Appendix A – FAQ

Q: How do I know if my ray hit the front or back face of a triangle?

A: You should always define triangles in a counter-clockwise way. You could define the triangle on the right as $p_0p_1p_2$ or $p_2p_0p_1$, but not $p_2p_1p_0$. This way, the cross product $p_0p_1 \times p_0p_2$ will always result in a vector that faces the front side of the triangle.



Q: What should I do if a ray (light/viewer's eye) hits the back side of a triangle?

A: You just ignore this intersection. In both cases – if you defined your object meshes properly, then there should be a front facing triangle intersecting the ray closer, and therefore you can discard these intersections and move on to the next ones! For an example, open the scenes folder, got to primitives, and see the rendered tri_front and tri_back scenes,

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: 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.

Q: How do I calculate R, the reflection vector?

A: Have a look at the last slide of rec2. Also, 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.

Q: What can I use to edit XML files?

A: Any text editor (e.g. notpad) is sufficient for the task. However some editors also support syntax highlighting and validation (e.g. notepad++) which might be useful. Some versions of Eclipse also have a fancy built-in XML editor.

Q: Do I need to constantly reload the scene to see changes?

A: No. You should load the scene once and every time you render the scene will be automatically reloaded.

Appendix B – Scene Definition Language

The scene definition language aims to define objects such as "scene", "camera", "sphere", "trimesh" etc. The scene definition language syntax is in XML format which is an extremely popular format for representing documents. In XML, elements are defined between '<' '>' clauses where the first word is the element's name. Each element has a list of attributes which is basically a list of key-value pairs. XML can define a hierarchy of elements. In our case there is only a single root element with multiple direct children. The root element is the scene element which contains all the other elements. A valid scene definition file must have at least one scene element and one camera element. For a given object some attributes are required and some are optional and receive some default value (determined in the application).

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.

Example: <scene background-col="0.5 0.5 1" ambient-light="1 1 1"> <camera eye="1 2 1" look-at="0 0 0" screen-dist="1" up-direction="0 1 0" /> <sphere center="250 250 -200" radius="100" /> <light-point pos="10 10 10" /> </scene>

Scene

This element 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. I_{AL} from 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 N*N 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). When this parameter is 1 in a complex scene there should be a significant increase in performance. default = 0. If you implement this, you should also supply a scene XML to demonstrate that it works.
- max-recursion-level – (number) limits the number of recursive rays (recursion depth) when calculating reflections. Default = 10.

Camera

- eye - (3d coord) the position of the camera pinhole. Rays originate from this point. p_0 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. $dist$ 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 (frustum). $width$ from the slides. default = 2.0

A few notes you should remember about the camera:

- You need to either specify a direction or a look-at point but not both. Specifying a look-at point implicitly sets 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. you don't have to check that programmatically, just be aware
- 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 not be 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 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. I_0 from the slides. default = (1,1,1) - white.

omni-light: A light emitted from a single point in all directions

- pos - (3d coord) the point where the light emanates from.
- attenuation - (3 numbers- k_c, k_l, k_q) the attenuation of the light as described in the slides. Default = (1,0,0) - no attenuation, only constant factor.

spot-light: A light emitted from a single point in a specific direction

- pos - (3d coord) the point where the light emanates from.
- direction - (3d coord) the main direction in which the light shines at.
- attenuation - (3 numbers- k_c, k_l, k_q) the attenuation of the light as described in the slides. Default = (1,0,0) - no attenuation, only constant factor.

Surfaces

Every surface has a material. In this exercise we model materials as just a flat color (you are welcome to do textures for extra bonus points). The following parameters may occur on every type of surface:

- 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 $(V \cdot R)$ in the formula (n). default = 100
- reflectance - (number) the reflectance coefficient of the material. K_S from slides. default = 0 (no reflectance).

sphere

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

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

trimesh

A triangle is a plane fragment defined by 3 points - p0, p1, p2, each consisting of 3 coordinates, totaling in 9 real numbers. A triangle mesh is a list of triangles.

- tri1, tri2, tri3, tri4, …, tri*n* - (9 numbers each) Each attribute starting with the prefix "tri*" defines a triangle in space. The order of coordinates is [p0x p0y p0z p1x p1y p1z p2x p2y p2z]. The points must not be co-linear. The front of the triangle is in the direction of the normal (p1-p0)x(p2-p0).

Helpful links:

http://en.wikipedia.org/wiki/Ray_tracing_(graphics)

http://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node2.html

http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm

Good Luck!