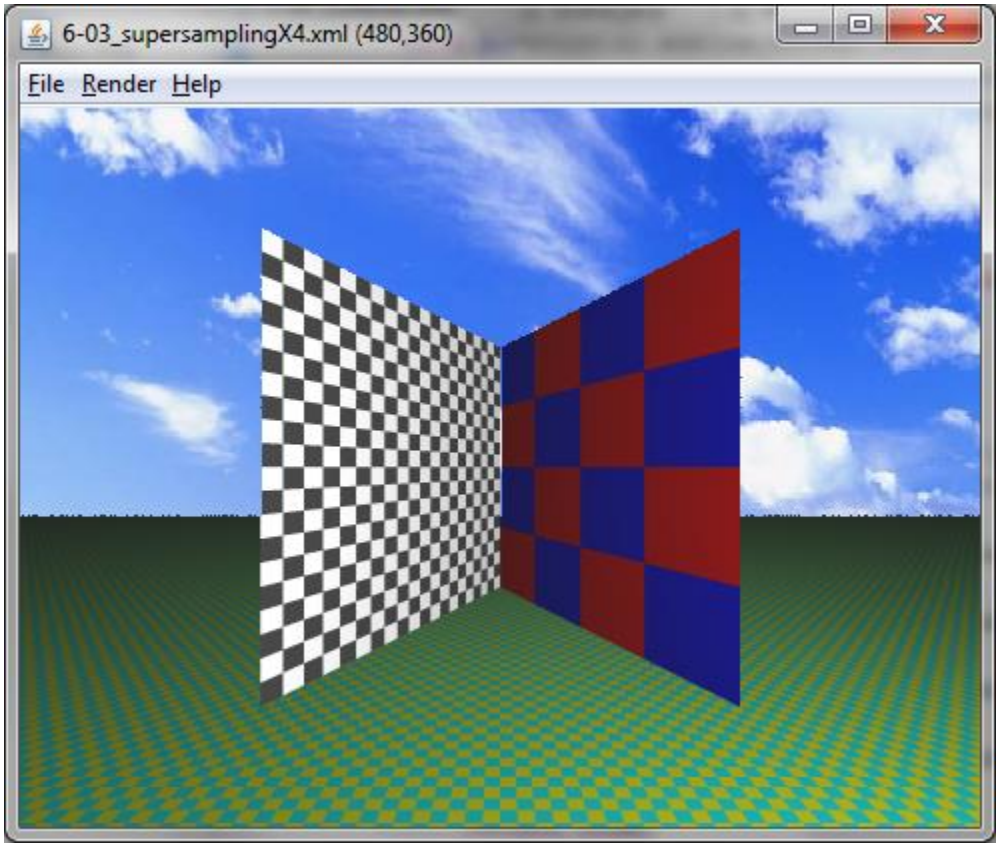
Exercise 3 – Ray Tracing



Contents

Overview- Ray Tracing Exercise

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 better than the typical scaling rendering methods, but at a greater computational cost.

The objective of this exercise is to implement a ray casting/tracing engine. 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**.

Input: **XML files, which you construct, containing:**

- parameters of the scene: geometric objects and their properties (geometry, position and material), lights, camera parameters
- ray tracing algorithm coefficients (like super resolution parameters, recursion coefficient, acceleration method if you chose to implement it)
- properties you like to add to your scene description

Output: **Rendered image of the scene from the point of view of the camera**

Environment features Supplied by Us

- **XML parser** for input XML files. This is simply a way to get the data for the scene. Basic implementation is supplied. **There's an explanation for it here.** You will have to understand it and add your definition for scene features in the XML files, use the forum, your friends etc.
- A simple **GUI** which will call your code for rendering
- **IRenderer, IInitable** interface class (will be explained later)
- **RayTracer, Scene**, **Ray, Point3D, Vec, Light** envelopes for classes which you should implement (will be explained later)

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.

Go through the examples before you start and see the XML syntax and conventions. Most of it is in the XML syntax document we provided. The rest you can figure out easily from the output images and the files.

*The last example of the house is in written a little differently. Trimesh is set of triangles. Understand the geometry of the scene, you can rewrite the XML in a different format, like combine 2 triangles into a rectangle, according to your code implementation.

Functionality 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 (5 Points **bonus**)
- Display geometric primitives in space:
  - Spheres (5 Points)
  - Discs (5pt)
  - Convex polygons (10 Points)
  - Triangular meshes (**bonus**)
- Basic lighting
  - Ambient light (5 points)
  - Directional 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- you can render any polygonal shape from convex polygons, you can load obj files of cool shapes and render them, as they are sets of triangles!

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 extremely short explanation of what you did in the attached submitted document. Note that for each extra feature you implement you need to supply a scene definition that demonstrates it.

Code Design

Before getting to the code, plan your design. Implementing a ray tracer poses an array of opportunities for creating good OOP design.

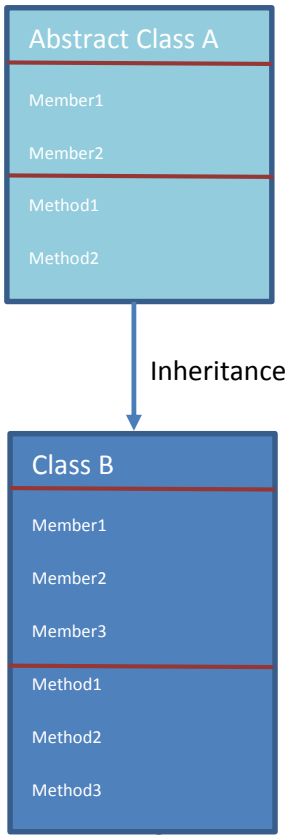
First, understand what is given in the scene and what do you need to calculate. **Don't define features in the XML that can be calculated, the XML should contain minimal data for constructing the scene and rendering it!**

Then define classes with their properties, design inheritance scheme, use interfaces, abstract classes, divide responsibility for different functionalities etc. as was explained in class.

Documentation

This is mainly for you! Write a short documentation which describes your design:

1. Total structure description of your design and a short explanation for each class responsibility. Add a description of algorithm flow through your design (which methods are called from which classes) when you calculate the color of a pixel. Start from the RayTracer class renderLine method.
2. Draw a schematic class diagram. The diagram should contain blocks for classes with methods and members. Make a different shape/color for abstract classes and interfaces, draw arrows between classes that have inheritance/implementation of interface relations. Here is a basic example:



Coding Guidelines

- **Write functions** for different functionalities even if they will be used only once. Long functions are hard to follow, deal with and debug. **Divide and conquer!**
- **Check each component separately:** Instead of writing tones of code and then debugging all this mess all together, make sanity checks for small functions and components. This checks include simple input, like small arrays or easy primitives aligned to axis and so on, where you know which output to expect.
- **Code documentation**: Write input output description for each method and a description comment for each class member

Submission

Submission is in pairs.

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 (PNG format).
- Compiled runnable JAR file named "ex3.jar" (no folder).
- A pdf describing your design as mentioned in this document. If you implemented bonus features, add a chapter about it in this doc (no folder).

Getting started

Using Our Code Envelope and Design Tips

Using Application - Running Modes

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.

The implementation won't render unless you have loaded a scene file. Remember that before you press 'render' in the GUI.

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.

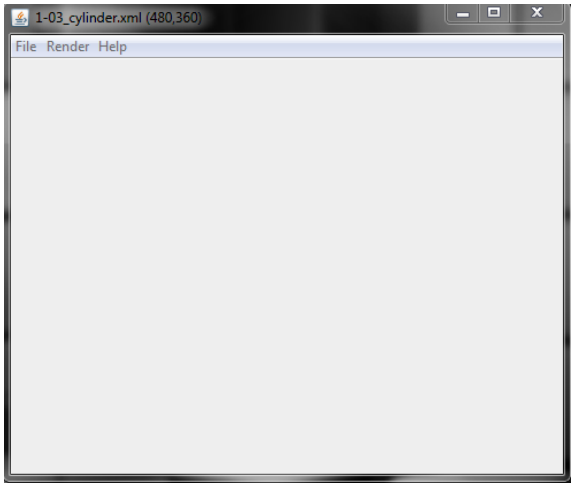


Figure 1 The GUI on start-up

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.

XML Parser

You don't need to worry about reading from the XML into the objects variable because this functionality is already supplied to you. Read the following explanation and then load an XML file in debug mode and check what the class **SceneDescriptor** contains. See also document about XML syntax and scene definition language.

1. **SceneXMLParser** is the class that reads the XML and fills the **SceneDescriptor** class with the data from the file.
2. **SceneDescriptor** holds all **attributes** of all objects read from the XML (objects variable). This class is the input to your **RayTracer**.
   This calss has 3 members **sceneAttribtes** (for algorithmic properties and basic scene general properties as you wish), **cameraAttributes** and a **List of objects** that will hold lights, surfaces and all other objects you decide to put in the XML. See the XML document we provided for more details. These objects have their attributes and you will probably have a class representing each object that will contain these properties in your design.
3. Classes that are initialized from the XML (like lights, geometric primitives etc.) should implement the **IInitable interface**, which we provided. You should add an **init** procedure to an object and set its parameters as was read from the XML :

```
public class AAA implements IInitable{
      SomeType property1;

      public void init(Map<String, String> attributes) {
                  if (attributes.containsKey("property1")) {
                        this. property1= new SomeType(attributes.get("property1"));
                  }

      …}

…}
```

IRenerer, RayTracer- Rendering *Interface* and Rendering classes

You should fill the **RayTracer** class that will implement the **IRenderer** interface we provided. A stub class was already created for you at ex3.render.raytrace.RayTracer.
On **init**, IRenderer receives by parameter the **SceneDescriptor** object which is filled by the xml parser. Here it can initialize the your scene class (see next). You have some suggested code that is commented.
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.

Camera Class

Implement the Camera class. Basic envelope is provided. Add and modify as you wish.

Your Own Scene Class

You should construct your own scene data structure. We gave you an envelope with some commented suggested code, do as you wish. This class should be optimized for efficient rendering.
Don't confuse with the SceneDescriptor which we provided. SceneDescriptor class is just for getting the data from the XML

Other components

The recommended approach is to create a class for each scene object (light, sphere, material, etc.) and access them from a Scene class. It makes sense to have a Light super class for omni-directional lights and spotlights, a surface super class for geometric primitives. We provided you with a suggested **Light** class envelope. Modify as you wish.

Math package

This package has 3 classes for mathematical entities- Ray, Point3D and Vec. Replace Vec with the code you implemented in Ex2. Implement methods you need for each entity. You can find more details in the Theory document.

Step by Step Guidance and Recommended Milestones

Design before you start coding! See Code Design Part above. 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...

Following are some practical suggestions that could make the exercise easier for you. **There are many other ways to approach it as well, and you don't necessarily have to follow these directions.**

1. Go through the Ray Casting and Tracing code in the lecture and understand it. Really, understand deeply. Only when you're done, you can begin the project.
2. Take a look at our envelope design, understand the building blocks. The flow **starts in MainFrame.render()**
3. Go over it to understand how the XML is read. This function is called after you press 'render' in the GUI.
4. The rest of important events happen in the RayTracer class (which you should implement, along with other classes that RayTracer uses).
5. Write an initial design, according to our guidelines. No coding, just a scheme document for a design! Theory and Tips document can help you understand which entities you need.
6. Understand the algorithm flow through your design- which class calls to which class when calculating color for a single pixel.
7. RayTracer.renderLine() is the function that renders things to the window. A row at a time from top to bottom. Inside it, you should loop over all x-pixels in that row. If you still don't understand how it works, try to put [canvas.setRGB(x, line, 0xffffff);] inside with x=0 first to see what happens. Then iterate over all possible x values.
8. Implement Ray casting in your code following the lecture. Implement the Ray class using Point3D. If Ray doesn’t intersect with an object return null in the function castRay()
9. Implement the method constructRayThroughPixel() in Camera class.
10. Render a sphere as said in the milestones.
11. Play with camera position.
12. Implement background - it should be drawn at pixels that returned null at castRay(). More details in the Theory and Tips document.
13. Create an omni light class, and initialize it from XML. Make it work as explained in class.

14. Implement the other lighting types. You probably would want a Light superclass for some of them (not necessarily all).
15. Implement all material properties given in the Scene Definition Language document.
16. Add support for shadows and reflection by recursively shooting additional rays.
17. Go through the exercise definition and make sure you haven't forgotten anything.

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!