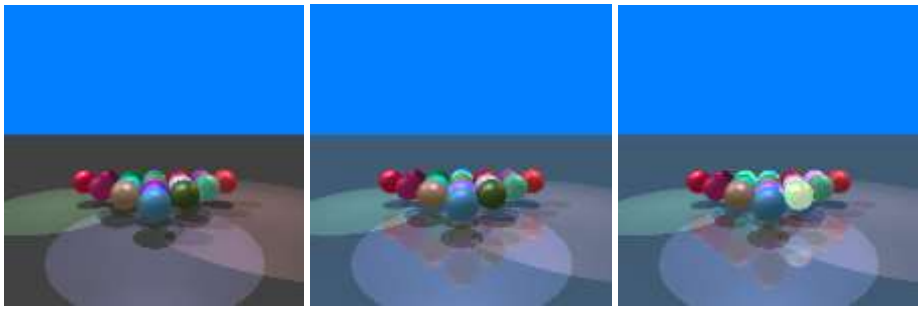
Exercise 3 – Ray Tracing
Due 24th of May 2020 - 23:55



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 scan-line rendering methods, but at a greater computational cost.

The objective of this exercise is to implement a ray 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).**

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:

- Pinhole Camera (15 points)
- Background
  - Plain color background (5 points)
- Display geometric primitives in space:
  - Spheres - (10 points)
  - Domes - (5 Points **bonus**)
- Basic lighting
  - Ambient light (5 points)
  - Directional light (10 points)
  - Spot light with cutoff angle (15 points) - it is based on a point light which is already implemented for you.
  - Simple materials (ambient, diffuse, specular...) (15 points)
- Basic hard shadows (10 points)
- Reflecting surfaces (15 points)
- Refracting surfaces (5 point **bonus**)
- Super sampling (5 points **bonus**)

Challenge: you will get a bonus (up to 15 points) for implementing advanced extra functionality (like supporting domes or super sampling) 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.**

Environment features:

- A parser for a simple scene definition language. (base implementation is supplied)
  Note: you will not have to implement the **json** 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 collection of scenes makers appeared on the class Scenes, you can make your own scenes there. We will elaborate on this later.
- A simple GUI (supplied).
- Render the scene to an image file (export functionality is supplied).

Scene Definition

The 3D scene your ray tracer will be rendering will be defined in a scene definition text file (**json**). The scene definition contains all of the parameters required to render the scene and the objects in it.

Usage

GUI Mode

Your implementation must use the accompanying GUI!

The GUI for the application is composed of a main window (very similar to Ex1) which displays the rendered image. All operations are accessed via window buttons.

The user selects a scene by clicking *"Brows scene…"* button. This file has to be a valid .json file containing a valid scene structure. Then the user can select output image size by entering the dimensions within the right fields. Finally the user can render the scene by clicking the "Render scene" button. This will pop up, eventually, a window with the rendered image.

You can also create new scenes, using the Scenes creator. To do so, you first need to create a scene in the **Scenes** class. Each scene you create has the name scene# (case sensitive) where # is the scene number. The scene is merely public static object in Scenes class. For example, you can create a scene named **scene5** as follow:

```
public static Scene scene5() {
        Scene theSceneYouWantToCreate = new Scene();
        // Initialize theSceneYouWantToCreate  by adding objects, lighting etc..
        return theSceneYouWantToCreate };
```

Now in order to render the scene, you change the scene number in the GUI to **5** (as this is the scene number you just created), and click on the “Create Scene” button. This will essentially create your scene for you, and clicking on Render Scene button will render the scene you created.

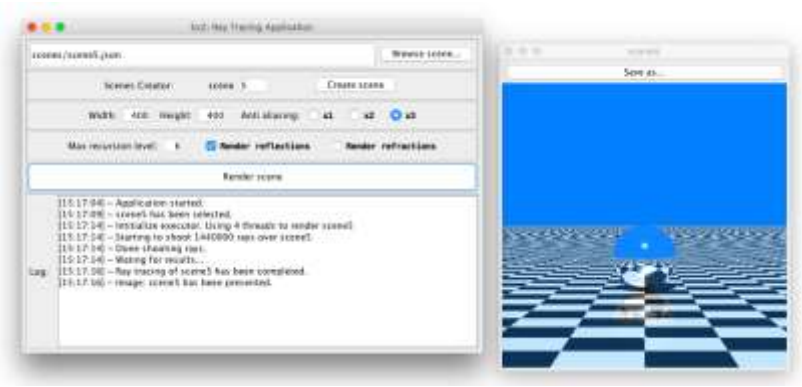


Figure 1 The GUI on start-up

Geometry (package edu.cg.algebra)

Vectors (Vec class)
An object which represents a vector in a 3D space, also represents colors.

Points (Point class)
A class which represents a point in a 3D affine space. As you know, although both vector and point can be represented by the same three coordinates, there is a big difference between them.

Ray (Ray class)
A class that represents a parametric representation of a line. This class contains two fields: A source point - which describes the beginning of the ray, and a normalized vector – which describes the direction of the ray.

These classes contain some algebraic operations we've discussed on class.

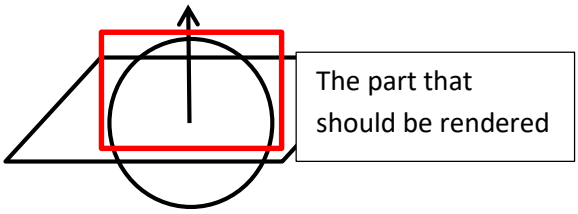
Note: *edu.cg.algebra contains more classes. Explore them and learn their functionality.*

Geometric primitives (package edu.cg.scene.objects)
You need to support up to 2 primitives (spheres and domes) - From these 2 primitives you can create a great variety of objects. The intersection of a line with a sphere box was described in the recitations, and the intersection with a sphere was shown in the written assignment.

Plain (This is already implemented for you)
A plain is defined by four coefficients: a, b, c and d. Those are come from the implicit representation of a plain: $ax + by + cz + d = 0$. The normal of each intersection point on the plain should create a sharp angle with the opposite direction of the ray (use the dot product to check that). The normalize vector of (a, b, c) is not necessarily the right one, you need to check the normalized vector of (-a, -b, -c) too.

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.

Dome
A combination of a sphere intersected with a plain. The normal to the plain (represented by it's a, b and c coefficients) describes the orientation of the plain relative to the sphere. Only the part of the sphere that is founded above the plain and the disc formed by the intersection should be rendered. Note that for simplicity, we will assume that the plain passes through the sphere center.



Background

The background of the rendered scene is a flat color. You should return the background color when no intersection occurred.

Camera

The camera is a simple pinhole camera as described in the lecture and the recitation slides. You need to implement the camera as given in the documentation. The camera is used to determine the **center point of the pixel** (x, y) on the image plain.

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. Directional – A light source like the sun, which lies at infinity and has a direction.
3. Cutoff Spotlight – 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.

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 are expressed 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 (shadow rays). 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 (a close intersection can be caused by inaccuracy lack of floating point representation. Use the Ops.epsilon value to check that). 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, refraction parameters and a shininess parameter (the power of $V \cdot R$). Look at the class Material at the package edu.cg.scene.objects. We have implemented different types of material for you so you can check your implementation easily.

Reflection

This is where ray-casting becomes ray-tracing. If a material has reflectance (reflectionIntensity 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 recitation 2. 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 * reflectionIntensity) and add it the color sum of this ray as explained in class. Make sure that the vector representing the ray's direction is normalized to make calculations simple.

Refraction (bonus)

Look into Snell's law: http://en.wikipedia.org/wiki/Snell's_law
Refractive index table: http://en.wikipedia.org/wiki/List_of_refractive_indices

Super Sampling (bonus)

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 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 chosen in the menu (x1, x2, x3) that tells how many vertical and horizontal rays are casted per pixel ($antiAliasingFactor = 1$ means no super sampling, $antiAliasingFactor = 2$ means 2x2=4 rays per pixel etc.)

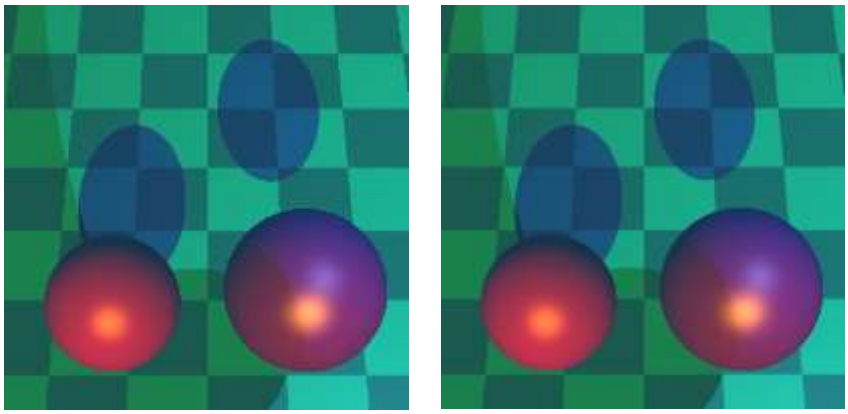


Figure 2 On the left an image rendered without super sampling, on the right with super sampling.

Notes

Getting started

You should import the project to eclipse. Note the given gson-2.8.2.jar file; if the project doesn't compile, import that jar file. Then, in the class Scene (at the package edu.cg.scene), you should implement the method:

```
private Vec calcColor(Ray ray, int recusionLevel)
```

This method simulates the ray tracing algorithm.

Note that if you need add some fields to this class; mark them as **transient** so the json parser will ignore them.

For example:

```
private transient Object aField;
```

You can initialize those fields at the function:

```
private void initSomeFields(int imgWidth, int imgHeight, Logger logger)
```

Look at the classes at the packages:

- edu.cg.scene.objects
- edu.cg.scene.lightSources

There are some implementations you should make; look for the TODO key words.

Note that any additional fields in these classes should be marked as **transient** as well.

Recommended Milestones

Design Before you start coding! You should think carefully about the structure and design of your system. Write incrementally. We suggest the following implementation milestones:

- Create a scene with only a plain surface (which is already implemented), use it and make sure you can fully render it.
- Add basic lighting support
- Implement full material properties
- Add more objects to the scene...

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. We also provide a reference executable that you can use to check your own scenes, but it should be clear that in case of contradiction this document supersedes the reference app's results.

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.

- A color can be seen as a vector that points from black (0,0,0) to a point in the RGB space (where (1,1,1) is white). Use the Vec class to handle vectors and RGB. Use the method toColor inside Vec class to convert the vector into color object (note that this method will clip values so that the final values is in the range [0,1], thus the vector (2.0,2.0,3.3) will be clipped into (1.0,1.0,1.0) and the color returned will be white).
- Make sure you work with normalized vectors as this will ease the calculations (reflection, refraction etc).
- When you find the intersection point (the t value in a ray), then you need to make sure that t is positive. Since negative values indicate that the intersection point is behind the source point of the ray, and $t = 0$ indicates the source point is the intersection point.

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 : 'Ex03 - Bart Cohen-Simpson 34567890 Darth Vader-Levi 12345678.zip'

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 .json files and rendered results in (PNG format) - **BE CREATIVE**.
- (only if you implement the bonus) A short and concise document where you can briefly discuss your implementation choices and bonus features (no folder).

Appendix A – FAQ

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 can I debug my ray tracer? 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. Use the logger object for that.

Another way, which is more manual but accurate – say you found there is a problem at (344,205). You can start by writing something like this:

```
if (x == 344 && y == 205)
      logger.log("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 .json files?

A: **Do not edit .json files**, they are complicated. Use the class Scenes to edit and create your own scenes. There are 7 examples of scenes there. Each function in this class should have the form:

```
public static Scene scene<n>()
```

Where <n> is a positive integer without leading zeroes. These functions will eventually build your scenes and create the suitable .json files. Look at the examples in the class Scenes at the package edu.cg.

Appendix B - Classes

A few more notes:

The scale for each value of a color is from 0 (no color) to 1 (most color). Look at the method toColor() at the class Vec.

Scene

This element defines global parameters about the scene and its rendering.

- Name – (String) the scene's name.
- backgroundColor - (Vec) color of the background. default = (0,0,0)
- maxRecursionLevel – (number) limits the number of recursive rays (recursion depth) when calculating reflections. Default = 1.
- ambient- (Vec) intensity of the ambient light in the scene. I_{AL} from the lecture notes. default = (0,0,0)
- renderRefarctions – (boolean) to indicate if refraction rendering is activated. Default = false.
- renderReflections– (boolean) to indicate if reflection rendering is activated. Default = false.
- camera – (PinholeCamera) an object that represents a pinhole camera (look at the documentation of the PinholeCamera class as you need to implement this).
- lightSources – (List<Light>) a list of the light sources participating the scene.
- surfaces – (List<Surface>) a list of surfaces participating the scene. Each surface contains Material properties and a shape like a sphere or a plain.
- antiAliasingFactor [bonus] - (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.

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:

- intensity - (Vec) the color of the light. I_0 from the slides. Default = (1,1,1) - white.

DirectionalLight extends Light: A light emitted from infinity with parallel rays

- direction - (Vec) the direction in which the light shines at.

PointLight extends Light: A light emitted from a single point in all directions **(Implemented)**

- position - (Point) 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, 0.1, 0.01).

Cutoff Spotlight extends PointLight: A light emitted from a single point in a specific direction.

- direction - (Vec) the main direction in which the light shines at.
- Cutoff angle - the cutoff angle of the spotlight.
- NOTE: look at the implementation of the Point light before actually implementing the cutoff spotlight. **You may** implement the Cutoff Spotlight from scratch and not extending it from the given PointLight class.

Surfaces

Every surface has a material and a shape. In this exercise we model materials as just a flat color. The following parameters may occur on every type of material:

- Ka – (Vec) the ambient part of the material (K_A). Default value = (0.1, 0.1, 0.1)
- Kd - (Vec) the diffuse part(s) of a flat material (K_D). You can choose in your ray tracer how to use it. Default Kd = (1, 1, 1).
- Ks - (rgb) the specular (and the reflectance coefficient) part of the material (K_S) default = (0.7, 0.7, 0.7)
- shininess - (int) the power of the $(V \cdot R)$ in the formula (n). Default = 10
- reflectionIntensity - (double) the reflectance decay coefficient of the material. Default = 0.3.
- isTransparent – (boolean) an indicator for transparency. Default = false.
- refractionIntensity – (double) the refraction decay coefficient of the material. Default = 0 – no refraction.
- refractionIndex – (double) the refraction index of the material appeared in Snell's law. Default = 1.5.

Shapes

Plain **(Implemented)**:
a, b, c, d – (doubles) the plain implicit form coefficients.

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

- center - (Point)
- radius - (double)

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!