Tuomo Nappa 899143, Computer Science 2020, 27.04.2022

General description

I made a 3d-visualizer that renders a scene from a file using a ray tracing algorithm. The scene can have spheres and walls of different colors. The objects can be smooth, mirror-like or light sources. The user can move and rotate the viewpoint freely and the program will render the scene from the given point of view.

User interface

The user can select the file to render and move and turn the view with commands through standard input stream. The program will output the rendered view in a different window using Java Swing library. All text-based output is printed to standard output stream.

The commands are:

- help: open help dialog
- render <filename>: renders scene from the file
- moveto <x> <y> <z>: move viewer to the given coordinates
- move <x> <y> <z>: move viewer to the direction given by the coordinates
- turnto <x> <y> <z>: turn viewer to the direction of the coordinates
- turn <x> <y> <z>: turn viewer by the given amount
- quit: quit the application

Program structure

The FileIO object handles all the IO functions of the program such as reading and parsing files and user input. I made this a separate object to separate all the IO functions in to one place.

The Scene class has all the information about the Viewer and the Objects in the Scene. The Scene class also has the methods for rendering the image. I decided to have the render methods here because this class has all the necessary data for the calculations.

The Viewer class has information about the viewers position and the facing and it also computes the first LightRays corresponding to each pixel in the final image.

The Objects have information about their position, color and other characteristics. They also have methods for computing the normal of the solid at a given point and the intersection with a LightRay.

The MyVector class represents a 3d vector and it has methods for all the vector calculations that are needed in the program. I made this class to make vector calculations easier.

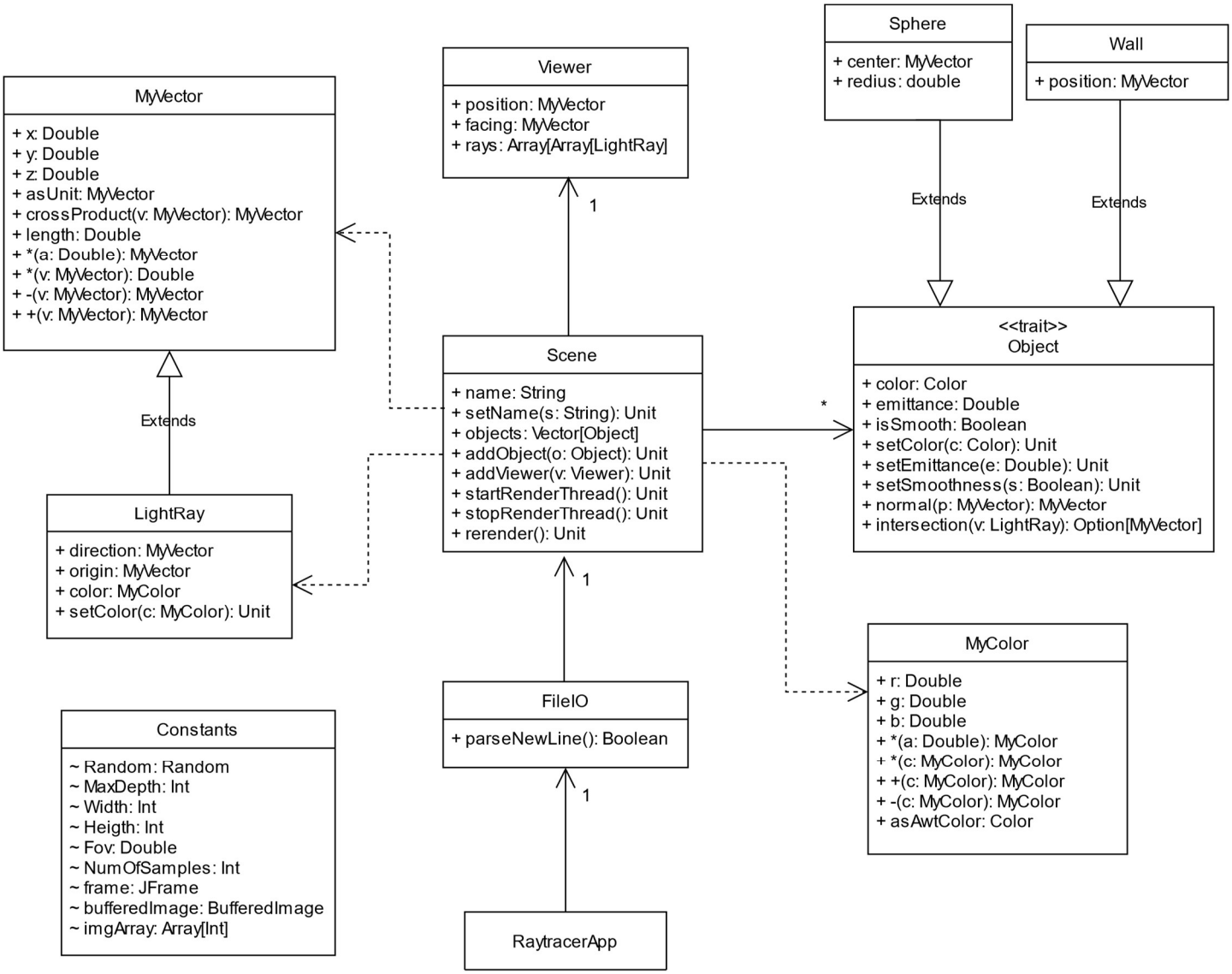
The LightRay class represents a light ray, and it has a direction, an origin and a color. It extends MyVector so it can be treated like a vector.

The MyColor class represents colors as multipliers for red, green and blue. It also has methods for multiplying, adding and subtracting colors. I decided to make this class to make it easier to do

calculations such as multiplying two colors and to allow color values to exceed the maximum value of java.awt.Color.

The Constants object stores all the constants that are used through out the program. I made this object so it would be easy to reference constants from everywhere and to have the constants all be in a single place.

I made the MyIterator class to be used in the FileIO class with lines read from a file. I needed an iterator that could return its current value without side effects.



Algorithms

The image is rendered using backwards path tracing[1]. I first calculate the rays corresponding to each pixel of the viewport. Then I follow each ray until it hits an object. The intersection point is calculated by the line-spere[2] or line-plane[3] intersection formula. Then it is checked if the object hit is a light source or not. If it is, the color of the ray is multiplied by the color and the emittance of the object and the resulting color is returned. If it isn't a light source, a new light ray is sent according to

bidirectional reflectance distribution function[4]. If the surface is smooth, a new light ray is sent in a random direction in the hemisphere of the normal of the surface. The random direction vector is chosen by the formula $\frac{1}{\sqrt{x^2+y^2+z^2}}\begin{bmatrix}1\\1\\1\end{bmatrix}$, where x, y, and z are gaussian distributed random numbers[5]. If the vector has any component in the opposite direction of the normal, its dot product with the normal is negative. These vectors are just flipped around so that they are in the correct hemisphere. If the surface is mirror-like, a new ray is sent according to Snell's law[6]. The color of the new ray is calculated by multiplying the color of the incoming ray with the color of the object. This is continued until a light source is hit or the maximum number of bounces is reached. In which case the color black is returned. This calculation is done multiple times for each pixel and the colors of each pixel are summed together. Then the value of each pixel is divided by the number of samples to get the average of those samples and the given color is rendered on screen.

I decided to use the above mentioned way of rendering an image because it is simple, and it produces good-enough-looking images. I could have also just calculated the first ray collisions and then looked for an intersection with a light source from that point. That would have probably also looked pretty good, and the image would have been less grainy but then everything that doesn't have straight line of sight with a light source would have been completely black. I could have also used bidirectional path tracing[7] which would have also reduced the graininess, but it would have also been more complicated. I decided not to add specular highlights[8] for smooth surfaces because it would have made the program more complicated. Supersampling[9] could have also been used to reduce the graininess but I didn't have enough time to implement it.

Data structures

I used Arrays in the program when the size of the collection stays constant because they are easy and efficient to use. If I couldn't use arrays, I mostly used immutable Vectors because they are the default in Scala, and I am more used to using immutable data structures. I also made my own iterator MyIterator which works exactly like a regular iterator, but it also has a method for getting its current value without side effects. I made it because it made the code in FileIO clearer and more modular.

Files

The application constructs the scene from a text file. The file must start with *scene <name>*. The name field can also be left empty. The file must also define a viewer for the scene. A viewer is defined like so:

@viewer
    p <x> <y> <z>
    f <x> <y> <z>

Where p is the position of the viewer and f is the facing of it. There can be any amount of white space around different attributes and the attributes can be in any order.

Objects can be added to the scene like this:

@wall
    n <x> <y> <z>
    p <x> <y> <z>
    c <r> <g> <b>

e <n>
s <boolean>

@sphere
r <n>
p <x> <y> <z>
c <r> <g> <b>
e <n>
s <boolean>

Where n is the normal of the wall, r is the radius of the spere, p is the position, c is the color given in values between 0 and 255, e is the emittance as a non-negative decimal value and s is the smoothness. Values c, e and s are optional.

Comments can be added to the file with # where everything after the symbol on the same line is part of the commented.

Testing

I used unit testing at the start of development when I didn't have any working demo. I did unit tests with scalatest[10] for vector calculations and for calculating intersections and normals. When I got a working demo, I used the demo to see if anything was wrong and I didn't write any more tests. I also took time of the rendering to try to optimize it. I planned on testing the application much better but due to not having enough time, I wasn't able to.

Known bugs and missing features

The user can move the viewer through objects. This could be prevented by checking for collisions along the path to the next position.

The objects have a normal only on one side of their surface, so they only reflect light on the side of the normal.

The methods in FileIO are not fully tested so there will probably be bugs.

Objects that emit light do not reflect light.

3 best side and 3 weaknesses

I really like the pattern matching objects in FileIO and how they made the code so much cleaner.
I also like how modular FileIO came out.
I like that the rendering is done on a separate thread so you can still enter commands while rendering.

I don't like that I didn't get to test FileIO fully.
I don't like that I didn't get the parallel processing working faster than sequential processing.
I don't like how messy the tracePath method came out.

Deviations from the plan, realized process and schedule

The project was completed mainly in the same order as planned. The main difference was that I made the method for calculating mirror-like objects earlier than planned because it was quick to do, and I never did the cube class due to running out of time. I also started working on parallel computing earlier than planned because it was interesting and something I hadn't done before. This

also took a lot of the little time I had for doing this project because I didn't really get any progress with it, and I probably used around 20 hours just reading Java and Scala tutorials and Stackoverflow threads about it. What I also hadn't planned for, was that making tests and testing the application took a lot longer than expected so I was already behind schedule after the first two weeks. What I learned about planning, is that I should schedule a lot more time for testing the application and that I should check what other stuff I have scheduled for the next weeks / months.

Final evaluation

I think the program is well made. The code is mostly clear and understandable, and it is also modular and easily expandable. The program is divided into classes so that each class has a clear function in the program. The object trait also allows for easily adding new objects and FileIO is made so that adding new attributes to files and new commands is easy. The program runs ok, and rendering is done on a separate thread so that the program can still react to commands mid-render. It runs slow because I didn't get the rendering to run well in parallel. I think the problem might have been that the program creates so many objects, like light rays, that are discarded quickly that the garbage collector can't fully keep up with it. This could have maybe been fixed by making light rays and/or vectors mutable so the program wouldn't need to make a new one every time. The program could also be made better by adding specular highlights for smooth objects to make then look more realistic and by using bidirectional path tracing to make the image less grainy when there is less light. The graininess could have also been reduced by implementing supersampling. If I were to start the project again now, I would read some guides and tutorials about making a ray tracer so that I would learn better algorithms for it. I would also probably make the light ray class mutable.

References

[1] https://en.wikipedia.org/wiki/Path_tracing
[2] https://en.wikipedia.org/wiki/Line%E2%80%93sphere_intersection
[3] https://en.wikipedia.org/wiki/Line%E2%80%93plane_intersection
[4] https://en.wikipedia.org/wiki/Bidirectional_reflectance_distribution_function
[5] https://mathworld.wolfram.com/SpherePointPicking.html
[6] https://en.wikipedia.org/wiki/Snell%27s_law
[7] https://en.wikipedia.org/wiki/Path_tracing#Bidirectional_path_tracing
[8] https://en.wikipedia.org/wiki/Specular_highlight
[9] https://en.wikipedia.org/wiki/Supersampling
[10] https://www.scalatest.org/

Also used for code:

https://en.wikipedia.org/wiki/Ray_tracing_(graphics)
https://en.wikipedia.org/wiki/Lambertian_reflectance
https://docs.oracle.com/javase/7/docs/api/java/awt/Color.html
https://docs.scala-lang.org/
https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html
https://www.baeldung.com/scala/
https://stackoverflow.com/questions/42981281/how-to-pattern-match-int-strings