Raytracer Part 3

In this assignment, we will take a slightly more advanced version of the raytracer and make the necessary changes to augment it with multithreading. To make the speedup with multithreading more visible we provide a version of the raytracer that includes everything from the previous homeworks and an advanced light model with a single light souce, shadows and reflections.

1) ArrayRaster

The rasterization (calculating a brightness value for each pixel of the image) is currently part of the raytracer class as a nested loop. In this nested loop, the viewport coordinates x and y are calculated, creates a ray, traces it and adds it to the vector raster. Move the part of the loop body into a new function, that takes two viewport coordinates (in the interval [-1;1]) and traces a ray. The result a relative brightness value between 0.0 (black) and 1.0 (white):

```
double trace_primary(double x, double y);
```

We want to remove the rest of the rasterization from the Raytracer class. Create a class `raster/ArrayRaster.h` with a constructor that takes the width and height of the picture. Add a method `render` that takes a `Raytracer&` as parameter. The definition of the `render` method will contain the nested loop of the original `Raytracer::to_raster` method, which traces all rays, scales the brightness values to 0-255 and stores them in a vector. Add methods for accessing a brightness value in the raster and two methods to get the width and height of the raster.

Now, add a virtual base class `raster/Raster.h` to the Raytracer that defines all four methods of the ArrayRaster (e.g. `render`, `at`, `width`, `height`) as pure virtual functions. Modify the main method to use the `ArrayRaster`. `PGMOut` also has to be modified, as it is now passed a `Raster*`.

If you implemented everything corretly, your Raytracer should still emit exactly the same pictures.

2) ParallelRaster

Now to make the raytracer multithreaded, add a new class `raster/ParallelRaster.h`, that inherits from `Raster`. Add a constructor that takes the width, height and number of threads. Instead of statically assigning each thread a segment of the picture, the picture is segmented in its rows. Every thread retrieves in a loop a row that has not been computed yet. You must ensure with appropriate synchronization mechanisms, that no row is computed twice. Modify your main method to use the ParallelRaster with 4 threads. The pictures generated by the Raytracer should be the same as before, but the Raytracer should be faster now.

Submission notes

Make sure that every row is computed exactly once (think of printing to `cerr`). Unlike in Java, `volatile` variables are in C++ not a sufficient synchronization mechanism. This goal of this submission is the correct usage of synchronization mechanisms. A solution that tries to cleverly avoid that is not acceptable. The submission is to be uploaded to gradescope, where a basic output test is performed. We will grade the submission by hand however to ensure correctness in your threading implementation.

Project structure

If you followed these instructions, your project should look similar to this:

```
.
├── geometry
│   ├── Point.cpp
│   ├── Point.h
│   ├── Ray.cpp
│   ├── Ray.h
│   ├── Vector.cpp
│   └── Vector.h
├── main.cpp
├── Makefile
├── output
│   ├── PGMOut.cpp
│   └── PGMOut.h
├── raster
│   ├── ArrayRaster.cpp
│   ├── ArrayRaster.h
│   ├── ParallelRaster.cpp
│   ├── ParallelRaster.h
│   ├── Raster.cpp
│   ├── Raster.h
│   ├── Viewport.cpp
│   └── Viewport.h
├── scene
│   ├── Camera.cpp
│   ├── Camera.h
│   ├── primitive
│   │   ├── Intersection.cpp
│   │   ├── Intersection.h
│   │   ├── Plane.cpp
│   │   ├── Plane.h
│   │   ├── Primitive.cpp
│   │   ├── Primitive.h
│   │   ├── Sphere.cpp
│   │   ├── Sphere.h
│   │   └── Surface.h
│   ├── Scene.cpp
│   └── Scene.h
└── tracer
    ├── LightModel.cpp
    ├── LightModel.h
    ├── Raytracer.cpp
    └── Raytracer.h
```