Raytracer Part 2

In this project you will keep refactoring and extending the raytracer with the ability to draw planes, dynamically specify the camera position and loading scenes from files.

1) Refactor: `Primitive`

Currently our scene consist only of a single sphere. To support other types of shapes introduce a pure virtual base class that will be the parent of all shapes. Create a new directory `scene/primitive`, move `Sphere` into this directory and create the class `Primitive`. Refactor `Sphere` to inherit from `Primitive`. Make the method `intersections` from `Sphere` a pure virtual function in `Primitive`.

2) Add: `Plane`

Now that you have a base class for different shapes, you can add a second shape for planes. Planes are infinite and can be specified with a single point on the plane and the normal vector of that plane.

> *Example:* Recall, that the normal vector of a plane is perpendicular to that plane. Hence, if you wanted to construct a plane, that is horizontal, the normal vector for this plane is vertical. Such a vector could be [x=0;y=1;z=0].

Define a class `Plane` in `scene/primitive`. The constructor of `Plane` takes a point on the plane and the normal vector as arguments. However, if we want to calculate intersections for the plane, we have to transform the coordinates first.

This algorithm to compute intersections requires the normalized normal vector (e.g. the normal vector scaled to length=1) and an offset `d`. `d` is the distance from the origin (0;0;0) to that point of the plane, that is nearest to the origin. This distance can be computed like this:

```
Plane(plane_point, plane_normal):
    this.normal = normalize plane_normal
    plane_vector = vector from origin to plane_point
    this.distance = -(plane_vector dot this.normal)
```

Now we can calculate intersections with this formula:

```
Plane::intersections(ray):
    vd = ray.direction dot this.normal
    if vd = 0: ray is parallel to plane -> no intersection
    distance = -(this.normal dot (vector from origin to ray.start) + this.distance) / vd
    return distance if greater than 0
```

3) Fixing image upside-down

The code from the first assignment contained a small error, that the image would be displayed as upside down. This will be visible if you try testing the new plane primitive. The cause is, that the raytracer emits the picture from top to bottom and thus should count down the y-coordinate. To fix this, revert the loop in `Raytracer`, that increments to y-pixel from 0 to resolution-1 to decrement from resolution-1 to 0 instead.

4) Refactor: `Intersection`

The function `Primitive::intersections` returns the intersections between rays and primitives as doubles. These doubles indicate the distance from the start point of the ray to the point of intersection on the surface of the primitive. To calculate lights and shadows later, we not only need the distance, but also the exact point and primitive for each intersection.

Create a struct `Intersection` in `scene` that holds:

- a pointer to the intersected primitive,
- the distance from the start point to the intersection and
- the exact point of intersection.

Compute the point of intersection: `ray.start + ray.direction * distance`.

Change the function `Primitve::intersections` to return a vector of intersections instead of doubles and propagate this change to `Sphere` and `Plane`.

5) Refactor: `Camera`

The camera is hardcoded at a fixed location. We would like to move the camera regardless of the contents of the scene.

Define a class `Camera` in the folder `scene`, which will control the position and direction of the camera and the size of the viewport. To calculate rays, the camera needs to store four variables:

- `camera_position`: the position of the camera, currently `(0;0;5)`.
- `view_vector`: the vector from the camera position to the viewport center, currently `[0;0;-5]`.
- `up_vector`: the up vector of the viewport, currently `[0;1;0]`.
- `right_vector`: the right vector of the viewport, currently `[1;0;0]`

To make it easier to specify the viewport, we define the constructor of `Camera` with these four arguments:

- a point for the camera position,
- a point for the viewport center,
- two doubles for the viewport height and viewport width.

You can calculate the view, up and right vectors like this:

```
view_vector = vector from camera_position to viewport_center
right_vector = scale (view_vector cross Vector{0,1,0}) to (viewport_width / 2)
up_vector = scale (right_vector cross view_vector_) to (viewport_height / 2)
```

A function to calculate the cross product between two vectors will be useful here and later. Recall, that the cross product of two vectors `a` and `b` returns a vector that is perpendicular to both `a` and `b`.

Now you can define a function on the camera, that returns a ray provided with viewport coordinates. You calculate rays with this formula:

```
horizontal = right_vector * viewport_horizontal_coordinate
vertical = up_vector_ * viewport_vertical_coordinate
direction = view_vector + horizontal + vertical
ray = camera position, direction
```

Refactor the raytracer to use your new `Camera` class.

6) Loading scenes from files

Now that we can add spheres and planes, and specify the camera position, we would like to be able to load scenes from files. Define a class `Scene` in the `scene` directory. A scene stores the camera and a list of primitives.

A scene files consists of a number of definitions with one definition per line (without the comments):

```
camera [x y z] [x y z] w h      // required, exactly one
sphere [x y z] r                // optional, zero or more
plane [x y z] <x y z>           // optional, zero or more
```

Define a constructor in `Scene` with one parameter for the filename of the scene file. You can use `std::ifstream f{path}` to read the file. If you want to read a single line, you can use `line = std::getline(f)`. You can then parse the line, by storing it in an `istringstream ss{line}`. Useful methods are:

- `ss.peek()`: returns the next character
- `ss.ignore()`: skips the next character
- `ss >> value`: reads the next value from the stream like cin.

You can use everything from the standard library that you think is useful or define additional classes.

Define a function `find_intersection` that takes a `Ray` as parameter and returns the closest intersection over all objects in the scene. To account for the case, that not all rays intersect an object, you can return `std::optional<Intersection>`.

Now refactor the class `Raytracer` to use `Scene` instead of using a hardcoded `Camera` and `Sphere`. Update your main, to look like this:

```
int main(int, char *argv[]) {
    Scene scene{argv[1]};
    int resolution = static_cast<int>(strtol(argv[2], nullptr, 10));
    Raytracer raytracer{move(scene)};
    PGMOut().save(raytracer.to_raster(resolution));
    return 0;
}
```

7) Adapt brightness

Our old brightness calculation does not work well with planes. As we still do not have light sources, use this formula in the meanwhile:

```
int brightness;
if (intersection.has_value()) {
    double distance{intersection->distance};
    distance = distance < 1 ? 1 : distance;
    brightness = static_cast<int>(1 / sqrt(distance) * 255);
} else {
    brightness = 0;
}
```

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
├── image
│   ├── PGMOut.cpp
│   └── PGMOut.h
├── main.cpp
├── Makefile
├── scene
│   ├── Camera.cpp
│   ├── Camera.h
│   ├── primitive
│   │   ├── Intersection.cpp
│   │   ├── Intersection.h
│   │   ├── Plane.cpp
│   │   ├── Plane.h
│   │   ├── Primitive.h
│   │   ├── Sphere.cpp
│   │   │
│    │    └── Sphere.h
│    ├── Scene.cpp
│    └── Scene.h
└── tracer
     ├── Raytracer.cpp
     └── Raytracer.h
```