

# Building a Raytracer

Lars Werne, Hugh Leather

Welcome to the one and only UoE Programming Club tutorial on how to build a ray tracer in Java!

First of all, you might be wondering: What is a ray tracer and what does it do?  
Well, it traces rays.

Thank you, that was the tutorial, see you next week!

In all seriousness though, ray tracing is a technique used to create images using a computer. The path of light is simulated using pixels. You then add a bunch of virtual objects and see how your ray tracer simulates the encounter of light and objects. This proves the “problem of visibility”: Given a 3D environment and a camera, which objects should be visible?

The resulting reflections and shadows can lead to a very realistic image, like this one:



But let's not get ahead of ourselves.  
Our first ray tracer will create images that look a bit like this one:

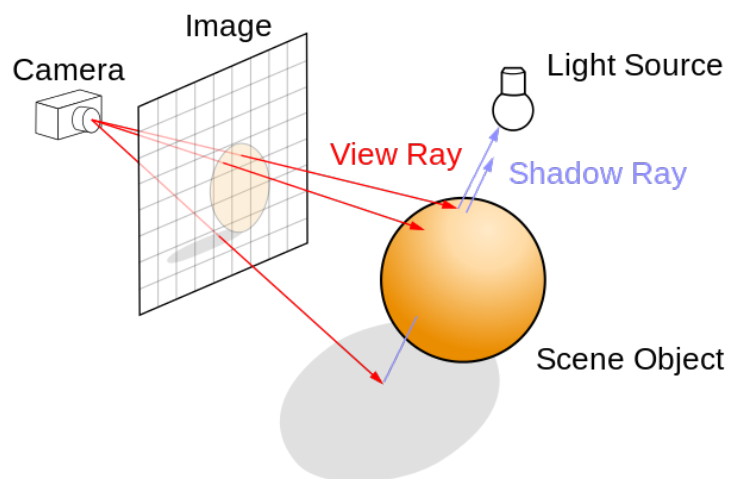


This may all still sound very vague, but we will slowly take you through the process of creating a basic ray tracer, which you can then try to improve, using and refining the techniques you will have learned.

[How to download Eclipse]

For this project, you may use whatever IDE (Integrated Development Environment) you like. If you are a first-year student, doing Java in your second semester, you will be required to use Eclipse in the exam. Therefore, that is the IDE I will be showcasing.

This is a sketch of the situation we would like to simulate:



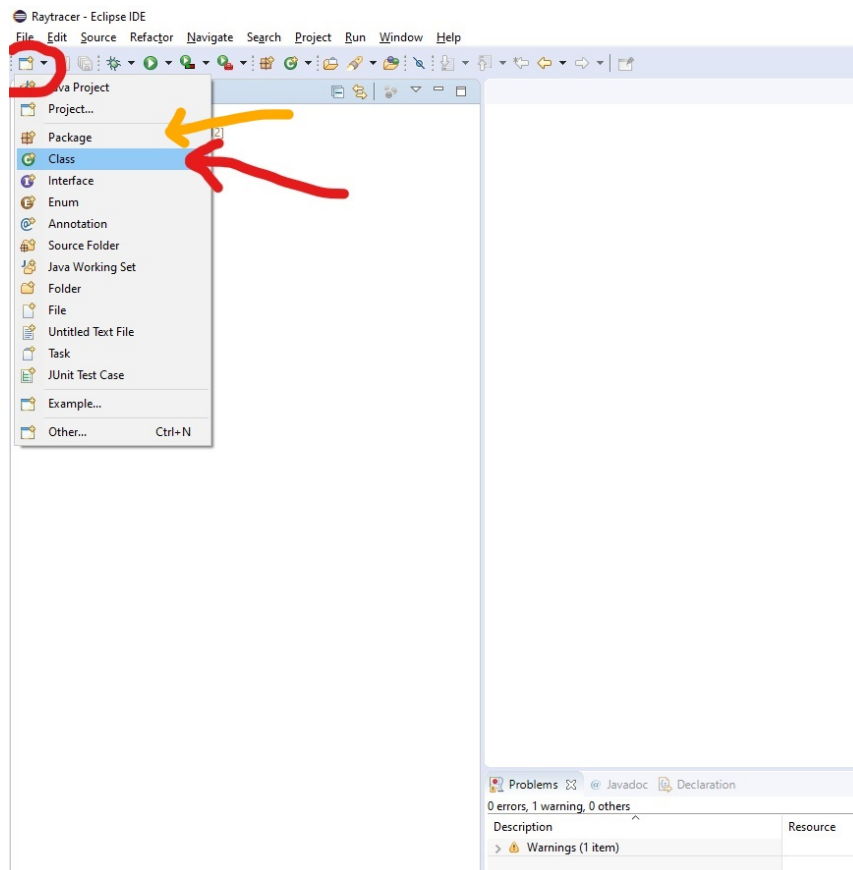
Before we can actually use this technique to create images, we will need ways to simulate Scene Objects, Light Sources and Rays. In order to create the according classes, we will first need to implement more basic objects like vectors and planes. The final, and most complex, step will be to make all these things visible. But we won't worry about that step just yet, and start at the beginning.

Unsurprisingly, we will need some maths to make this work. Don't be too worried about understanding every single mathematical detail, though you should try to comprehend how we implement the mathematical formulae in Java.

We will save our implementations of the formulae we need in a special class, which we will call Utils.java.

Having opened the src-folder of our Raytracer project we just created, select “Package” in the drop-down menu in the upper left corner of your screen. Let’s call it “raytracer” and create it. Next, click on your new package and select “Class” from the same drop-down menu we used before. Make sure it’s located in the package “raytracer”, give it the name “Utils”.

Packages are a convention in object-oriented programming. Among other things, packages are useful because classes that are part of the same package can be allowed to access each other’s “protected” variables/functions, which is not possible from outside of the package.



Now that we created our first class, we want to start filling it with life. But before we can do so, we need one more class that we must implement first.

The rays of light our program is all about can be represented as lines in Cartesian space. Therefore, much of the maths we are going to use is related to vectors.

As a first step, we will thus come up with a representation of vectors, in the form of our very own Vector class.

***Create a class called Vector.java in our raytracer package.***

The vectors we would like to implement can be represented by three real numbers, an x-, y- and a z-coordinate.

To represent real numbers, we could use either one of the primitive data types double or float.

The difference between them is that double variables/fields take up more space than their float counterparts (in fact, twice the amount of space). In turn, using double values allows for bigger numbers (more digits before the decimal point) and higher precision (more digits after the decimal point).

Because we are not too worried about space at this point, we decide to roll with double variables.

We could enable changing around the coordinates of an existing vector, but this might lead to problems at a later stage. For example, remember how a line in Mathematics can be represented as a position vector, plus a direction vector times a variable. Neither the position vector, nor the direction vector should have their values changed, unless we want the line to change with them. To avoid any potential problems like this one, we thus decide to make the coordinates of our vectors immutable.

Adding the keyword “final” to the variable declaration makes sure that the coordinates of our vector cannot be changed after the vector has been created.

***In our Vector class, declare global, immutable fields of type double, called x, y and z.***

If we did this correctly, we will receive an error message:

The blank final field <variable\_name> may not have been initialized.

Final fields can only be assigned a value once, but not assigning any value to them at all would make them quite useless as well.

For our purposes, a vector can not assist without having 3 coordinates. Thus, we want to assign values to the 3 coordinates of a vector as soon as it is created.

We can achieve this, using a special kind of function, a constructor. Like all functions, constructors can take parameters – values of certain types, which the constructor can then work with.

When called, a constructor creates an object of the class the constructor is in. Generally, constructors can be used to make sure an object has certain properties as soon as it is created.

In our case, we can make sure that any vector has 3 coordinate values by passing these values to the Vector constructor on creation of a Vector. The constructor can then assign these values to the according variables.

***Write a constructor for our Vector class, which takes 3 double values as parameters, and then assigns these vectors to our coordinate fields.***

If we did this correctly, the errors we got earlier should now have disappeared.

Private/Public components.

Private components of a class can only be accessed from within that class.

Public components can be accessed from all other classes as well.

There are certain “shades of visibility” in between those two extremes, but we won’t worry about them now.

It is good style to consider carefully, which components of your classes should be accessible from outside the class, and which should be kept hidden. Often, variables are kept hidden in order to protect relevant data. If we still want other classes to be able to see/change the values of these private variables, we can add get-/set-methods to the class which the variables belong to. These can then be made public, enabling other classes to call these methods.

Since our coordinate fields are immutable, we don’t need to worry about set methods, which are generally used to change the values of private fields.

**Make our coordinate fields private, and add 3 public get-methods of return type double, each of which should return the value of one coordinate value.**

What other information might we want to know about one of our vectors?

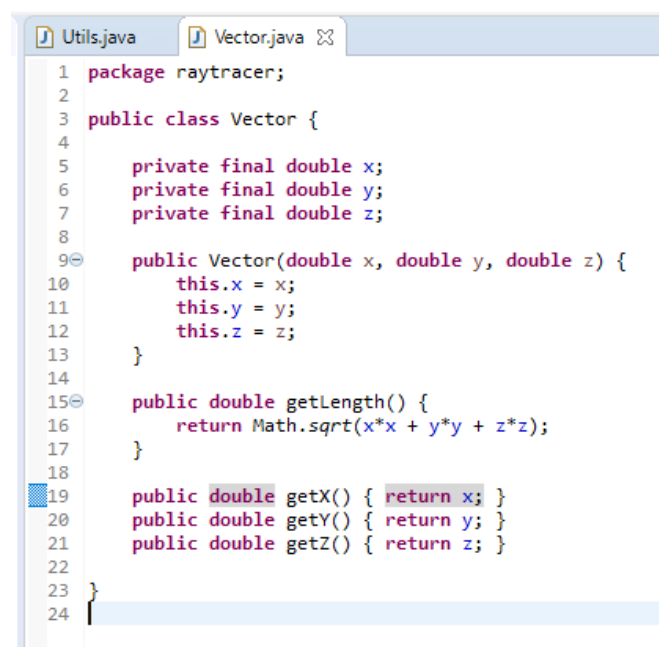
Often, we may be interested in the length of one of our vectors.

Remember that, applying Pythagoras' Theorem, the length of a 3-dimensional vector (x, y, z) equals the square root of the sum of the squares of the individual coordinates, or in short:

**$\text{length}(x, y, z) = \sqrt{x^2 + y^2 + z^2}$**

**Add a function of return type double to our Vector class, which returns the length of a vector.**

(Hint: In Java, the library function `Math.sqrt(int x)` returns the square root of x as a double value)



```
1 package raytracer;
2
3 public class Vector {
4
5     private final double x;
6     private final double y;
7     private final double z;
8
9     public Vector(double x, double y, double z) {
10         this.x = x;
11         this.y = y;
12         this.z = z;
13     }
14
15     public double getLength() {
16         return Math.sqrt(x*x + y*y + z*z);
17     }
18
19     public double getX() { return x; }
20     public double getY() { return y; }
21     public double getZ() { return z; }
22
23 }
24
```

Your Vector class should now look approximately like this.

We can think of a subtle problem with this piece of code, and a decision we have to make.

When we do arithmetic with vectors, we might require the length of a vector quite often. Computing it once doesn't take very long but doing it thousands of times over the run time of our program would be pretty expensive.

How could we solve this issue?

The easiest thing would be to simply store the length of a vector as one of its variables. Then, we would no longer have to compute the length of a vector every time we want to use it. We could compute the length as a part of our constructor. But then again, there may be many cases where we are never interested in the length of a vector, and computing it even just once would be a waste of time.

So, what's a good compromise?

A sensible thing to do is compute the length of a vector when we call the `getLength()` function for the first time. At that point, we can store the length as one of the vector's variables. If we then call the `getLength()` function again, we can simply return the already computed value.

How do we let our function know whether we already computed the length of a vector?  
And what value should the length have before we compute it?

A way to answer these questions is to add a variable of type `Double` to our `Vector` class, that we use to store the length of the vector. Note that I capitalized “`Double`”, meaning that the length should be an object of the class `Double`, which is one of the classes the Java Development Kit provides us with. Objects of the class `Double` are not the same as primitive variables of type `double`. All other primitive data types, like `int` (`Integer`) or `boolean` (`Boolean`) have these counterpart classes. The main difference is that the `Double` class carries methods and fields which we can use. The reason we decide to use it here, however, is that we can assign the value **null** to objects of the `Double` class.

(Pointers are a topic beyond the scope of this tutorial. Essentially, they are values of addresses where a specific piece of data is stored. Assigning a null pointer to an object lets the computer know that the according object does not exist yet, and that there is thus no part of memory to be pointed at.)

We can now assign the value `null` to the length field of a vector, whenever it is created. In our `getLength()` function, we can check if length is still `null`, which would mean that we haven’t computed it yet. If it has not been computed, we compute it and assign the computed value to our length field before returning it. If it has already been computed, we simply return the value of the length field.

**Add a length field of type `Double`, initializing it as `null` on creation of the vector (e.g. in the constructor). Then change the `getLength()` function, so that the length of a vector only has to be computed at most once during the lifetime of a vector.**

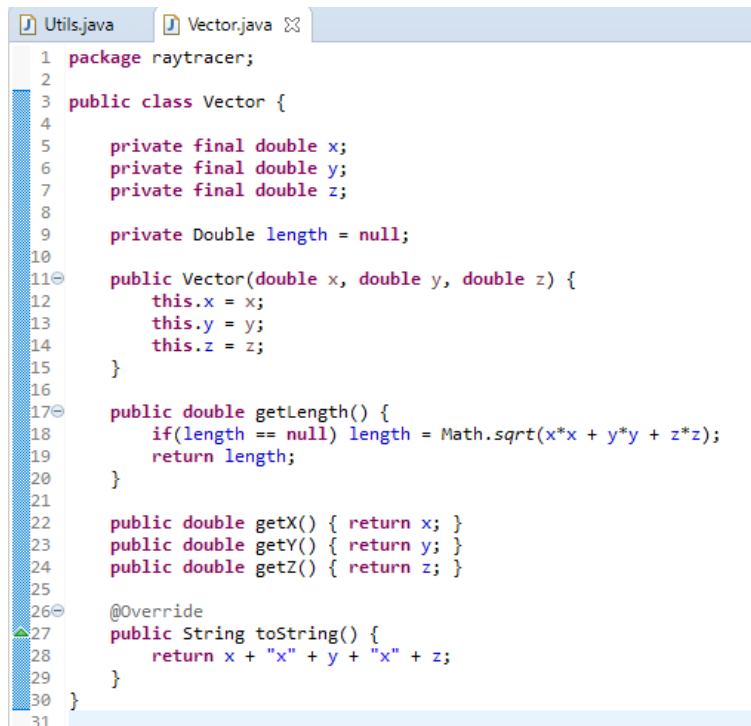
*(Can you think of any other ways this could have been done?)*

Lastly, we might want a convenient way to print the coordinates of a vector to our console. Therefore, we would like to implement a `toString()` method, which returns a `String` that represents the according vector object. The exact way this result looks isn’t important.

One representation we could choose is:  $x + “x” + y + “x” + z$ ,  
which would e.g. result in “3x5x8” for vector `[3, 5, 8]`.

**Add a `toString()` function to our `Vector` class, which returns a `String` representation of the vector object.**

Your Vector class should now look more or less like this:



```
1 package raytracer;
2
3 public class Vector {
4
5     private final double x;
6     private final double y;
7     private final double z;
8
9     private Double length = null;
10
11     public Vector(double x, double y, double z) {
12         this.x = x;
13         this.y = y;
14         this.z = z;
15     }
16
17     public double getLength() {
18         if(length == null) length = Math.sqrt(x*x + y*y + z*z);
19         return length;
20     }
21
22     public double getX() { return x; }
23     public double getY() { return y; }
24     public double getZ() { return z; }
25
26     @Override
27     public String toString() {
28         return x + "x" + y + "y" + z;
29     }
30 }
31
```

### **Inheritance + toString()**

Maybe you're confused about the "@Override" in line 26.

This line is in fact not necessary, the code would work just fine without it.

Typing "@Override" before a method is declared tells the compiler that we want to overwrite a method which our class inherits from a parent class.

(Inheritance in Java is another deep topic we won't cover in too much detail in this tutorial. If you're new to Object-oriented Programming, you should have a look at it.)

Every class we create in Java is going to be a subclass of the library class "Object" and will inherit the methods it contains. One of these methods is the "toString" method.

If v is an object of the Vector class, writing

```
System.out.println( v.toString () );
```

would have printed a String representation of our vector object. It just wouldn't have been a very useful one. ("Vector@", followed by the hexadecimal representation of the hash code of the object)

Therefore, we override the method to allow us to ask for a more useful representation.

After defining the method, writing

```
System.out.println( v.toString () );
```

would print the representation we defined instead.

But why did we write @Override? It has the following effect: If the function we define after the Override command was not a function already inherited from a superclass, the compiler would give us an error, and we would be unable to run the program. This might not seem very useful, but it

prevents us from thinking we are overwriting an inherited function when we are in fact not, e.g. because of a typo.

### **Implementing the maths**

So now we have a representation of a vector in Java. Remember that we created a Utils.java class, intending to fill it with arithmetic operations we would like to be able to perform.

Now, let us add methods to our Utils class that allow us to perform maths on vectors.

All methods we want to add should be static. That is because having an object of the Utils class wouldn't make much sense. How, and why would we differentiate between two Utils objects? We just don't, because we only ever need the Utils class and the methods it provides us with.

Static methods are essentially methods which belong to the class itself, rather than an object of the class.

Let us list some of the most elementary operations we can perform on vectors, and how they are defined:

- Vector addition:  $[x_1, y_1, z_1] + [x_2, y_2, z_2] = [x_1+x_2, y_1+y_2, z_1+z_2]$
- Vector subtraction:  $[x_1, y_1, z_1] - [x_2, y_2, z_2] = [x_1-x_2, y_1-y_2, z_1-z_2]$
- Scalar product:  $c [x, y, z] = [cx, cy, cz]$
  
- Dot product:  $[x_1, y_1, z_1] \cdot [x_2, y_2, z_2] = x_1x_2 + y_1y_2 + z_1z_2$

Add static methods to the Utils class which allows us to perform these elemental operations on vectors. Note that some of them should return a vector, others a scalar, i.e. a real number.

We can now use these operations to define more complex methods.

**Add a static method which computes the inner angle between two Vectors and returns it.**

$$\cos(\alpha) = (\mathbf{a} \cdot \mathbf{b}) / (|\mathbf{a}| |\mathbf{b}|)$$

*If you use this way of finding the angle, which you have probably seen before, you'll have to compute the inverse cosine of a number. Fortunately, the library class Math provides us with many useful methods for this sort of calculation.*

*Have a look at <https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html> to find a function that may help you. When working in Java, the Java documentation can become one of your best friends if you use it in a clever way.*



```

1 package raytracer;
2
3 public class Utils {
4
5     public static double dot(Vector a, Vector b) {
6         return a.getX()*b.getX() + a.getY()*b.getY() + a.getZ()*b.getZ();
7     }
8
9     public static Vector scalar(Vector a, double x) {
10        return new Vector(a.getX() * x, a.getY() * x, a.getZ() * x);
11    }
12
13    public static Vector subtract(Vector a, Vector b) {
14        return new Vector(a.getX() - b.getX(), a.getY() - b.getY(), a.getZ() - b.getZ());
15    }
16
17    public static Vector add(Vector a, Vector b) {
18        return new Vector(a.getX() + b.getX(), a.getY() + b.getY(), a.getZ() + b.getZ());
19    }
20
21    public static double getInnerAngle(Vector v1, Vector v2) {
22        double cosAngle = Utils.dot(v1, v2) / (v1.getLength() * v2.getLength());
23        return Math.acos(cosAngle);
24    }
25 }
26
27

```

For now, there's one more method we need:

We would like to be able to normalize a vector. What that means is that we want to take a non-zero vector and return the vector with length 1 which points in the same direction.

**Add a static method `norm()` to the `Vector` class, which returns the normalized version of the vector object.**

### Ray.java

Remember that we want to simulate rays. Formally, what is a ray?

A ray, also called a half-line, can be described by a direction vector, and a point of origin in space. Given our vector implementation, implementing a ray will thus be very straight-forward.

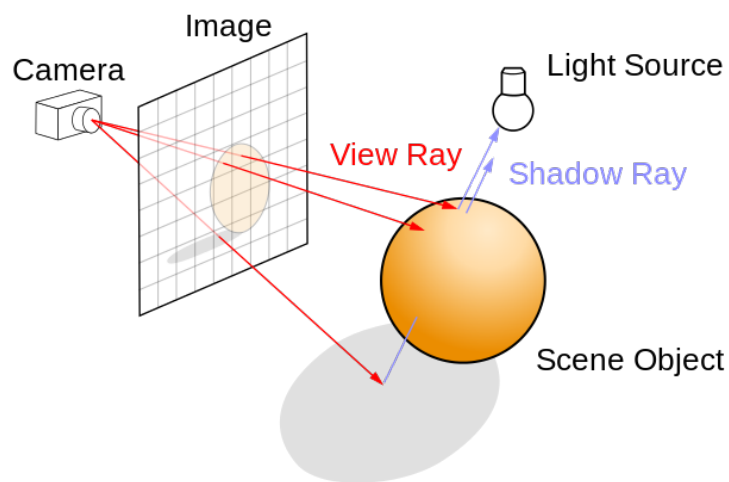
**Create a class `Ray.java`, with two variables: An origin- and a direction vector, both of which should be represented by `Vector` objects. Then, add a constructor, accepting two `Vector` arguments and assigning them to the according variable. The direction vector should be normalized.**

```

1 package raytracer;
2
3 public class Ray {
4
5     public Vector dir;
6     public Vector origin;
7
8     public Ray(Vector direction, Vector origin) {
9         dir = Utils.norm(direction);
10        this.origin = origin;
11    }
12 }

```

This very short class should look approximately like this.



Have another look at this sketch of what a ray tracer does.

We have now found ways of simulating rays in Java. That's a crucial part of our code, since we want to simulate the interaction of rays and other objects. We just don't know how to implement the objects yet. So that's what we'll take care of next.

### SceneObject.java + Colour.java

We will want to describe different kinds of objects. But as objects, they should all share some properties. Therefore, we create an abstract superclass called "SceneObject".

Abstract classes in Java are classes we can never have instances of. Does that mean we will never be able to create objects? No! We will later create subclasses of the class SceneObject that inherit its methods and fields.

If you're not sure what this means, have a look at this example:

Let's say we want to simulate all animals on planet Earth. We could have an abstract class "Mammal", with non-abstract subclasses like "Dog", "Cat", "Hippopotamus".

We could then create instances of "Dog" or "Cat", both of which inherit certain properties from "Mammal". But we could never create an object that is of the class "Mammal" but of none of its subclasses, since every mammal needs some further specification of what type of animal it actually is.

### **Create an abstract class SceneObject.java**

Now what properties should every object in our scenes have?

First of all, it certainly needs to have some position. The way we compute the position of an object might depend on what type of object it is.

Therefore, we won't define how it is computed just yet.

Instead, we define an abstract function with return type Vector, called getPosition().

Abstract functions do not have a function body, that means we have no idea yet what the function is supposed to do, just that we want it to return a Vector object.

Every non-abstract subclass of SceneObject will have to implement getPosition(). Those

implementations will then contain a function body, meaning that what the function is supposed to do will actually be well-defined.

**Add an abstract function `getPosition()` with return type `Vector` to your `SceneObject` class.**

Also, every one of our scene objects will be of a certain type. Of what type an object is will decide how we will deal with it in our ray tracer. I.e. a light object will be handled differently than a solid sphere.

What types are there? Well, that is up to us. We will start with only 2: Light sources, and spheres.

We can use an enum to store what different types there are. In Java, enums are used to define collections of constants. Here, the constants are supposed to be the different values the type of one of our `SceneObjects` can have.

Add an enum called `Type`, which contains constants for the possible types of scene objects that we want to be able to simulate. These constants should be words that describe the type.

Finally, each one of our objects should have a colour, otherwise we wouldn't know how to draw it. There are library classes that describe colours. But we might as well create our own.

A colour can be described by a red, a green and a blue portion. The purest red can be described by  $R=255, G=0, B=0$ . Green is described by  $R=0, G=255, B=0$  and blue by  $R=0, G=0, B=255$ . For our purposes, all colours can be described by an R-value, a G-value and a B-value, depending on how dominant the according portion is. (Have a look at "Additive Colour Mixing" if you want to, but you don't have to be an expert on this.)

To make our final product more easily usable, we can define some example colours in this class, once we added an appropriate constructor. For the colour red, we could e.g. write  
`public static final Colour RED = new Colour (255, 0, 0);`  
as one of our variable declarations.

**Create a class `Colour.java`. Add integer variables called `R`, `G`, `B` and add a constructor that takes 3 arguments and assigns them to the aforementioned fields. Define some static example colours, so that we don't have to manually call our constructor whenever we want a basic colour.**

**Finally, add a `toString()` method, that returns the RGB code of a colour as a string.**

Now that we know how to represent colours in our code, we can add a colour variable to our `SceneObject` class. Colour and Type should be initialized whenever a `SceneObject` is created. Therefore, we will need a constructor. Remember that `SceneObject` is abstract. Its constructor will thus describe things that will happen every time some kind of `SceneObject` is created. We will see how to call this "super-constructor" in the constructor of a subclass.

**Add a colour variable to our `SceneObject` class. Then add a constructor that initializes the colour and type fields when called. Finally, add a `toString` method that gives a String representation of a `SceneObject`.**

```

1 package raytracer;
2 public class Colour {
3
4     public static final Colour GREY = new Colour(100,100,100);
5     public static final Colour WHITE = new Colour(255,255,255);
6     public static final Colour RED = new Colour(255,0,0);
7     public static final Colour GREEN = new Colour(0,255,0);
8     public static final Colour BLUE = new Colour(0,0,255);
9     public static final Colour BLACK = new Colour(0,0,0);
10
11     public int R;
12     public int G;
13     public int B;
14
15     public Colour(int r, int g, int b) {
16         if(r > 255 || r < 0 || g > 255 || g < 0 || b > 255 || b < 0)
17             throw new RuntimeException("Colour channel out of range");
18
19         R = r;
20         G = g;
21         B = b;
22     }
23
24     @Override
25     public String toString() {
26         return R + " " + G + " " + B;
27     }
28 }

```

```

1 package raytracer;
2 public abstract class SceneObject {
3
4     public enum Type { POINTLIGHT, SPHERE };
5
6     public Colour colour;
7     public Type type;
8
9     public SceneObject(Colour c, Type t) {
10         colour = c;
11         type = t;
12     }
13
14     public abstract Vector getPosition();
15
16     @Override
17     public String toString() {
18         /*StringBuilder bld = new StringBuilder()
19         .append("Colour: ").append(colour).append("\n")
20         .append("type: ").append(type.name());
21         return bld.toString(); */
22
23         String s = "Colour: " + colour + "\n";
24         s = s + "Type: " + type.name();
25         return s;
26     }
27 }
28
29 }
30

```

If you know how StringBuilders work, we can use one here. But we can also not be fancy.

### LightObject.java

One of the type of objects we will want to include in our scenes are light sources. Not only will it

inherit a colour and a type from its superclass, it will also have a position which can be described as a vector.

**Create a LightObject class, which is supposed to be a sub class of SceneObject. Add a variable of type Vector to describe the position of the object.**

Now we can add a constructor to this class. Remember that the class SceneObject already has a constructor. Calling the constructor of a superclass is done using a function call following the – template

super(vartype\_1 varname\_1, vartype\_2 varname\_2, ... , vartype\_n varname\_n);

where the types of the parameters of course depend on the constructor we want to call.

This should be done inside the body of the LightObject constructor.

Add a constructor to the LightObject class. It should call the constructor of the SceneObject class and initialize the position field. Also override SceneObject's toString() and getPosition() methods.

```
1 package raytracer;
2 public class LightObject extends SceneObject {
3
4     public Vector position;
5
6     public LightObject(Vector p, Colour c, Type t) {
7         super(c, t);
8         position = p;
9     }
10
11     @Override
12     public String toString() {
13         return super.toString() + "\npos: " + position;
14     }
15
16     @Override
17     public Vector getPosition() {
18         return position;
19     }
20 }
21
```

This class contains all the information we need about a light source.  
LightObjects will serve as the origin of rays of light in our implementation.

### OrbObject.java & BoundingSphere.java

Of course, Light Objects are not the only kind of object we need to consider. Ideally, we want to be able to model any type of solid object in our code and simulate its interaction with incoming rays of light. For simplicity's sake, we will restrict ourselves for now, and **only model solid, round objects**. We can call these orbs, or Orb Objects.

Before implementing the OrbObject class, we shall consider the class **BoundingSphere**. The bounding sphere is the “**shell**”/outer layer of an orb.

After everything we have done so far, you shouldn't have too much trouble creating a class that contains all the information we need to represent a sphere as a geometrical object in 3 dimensions. A (hollow) sphere is defined as the set of all points with a certain distance (the radius) to a certain point, which we can model using a position vector.

Create a class that models a hollow sphere. Include variables for the values that define the sphere, a suitable constructor, and a toString() method.

```
public class BoundingSphere {  
  
    public Vector centre;  
    public double radius;  
  
    public BoundingSphere(Vector centre, double radius) {  
        this.centre = centre;  
        this.radius = radius;  
    }  
  
    @Override  
    public String toString() {  
        StringBuilder bld = new StringBuilder();  
        bld.append("Centre: ").append(centre).append(" ")  
            .append("radius: ").append(radius);  
        return bld.toString();  
    }  
}
```

Now that we can represent a sphere, adding the OrbObject class is straight-forward. Instead of a position variable, like in the case of a LightObject, we need a field of type BoundingSphere that describes the outer border of our 3-dimensional object. The position of the OrbObject, which must be a single point, can be given by the center of its BoundingSphere.

**Create a subclass of SceneObject that models an Orb.**  
**An orb is given by its BoundingSphere, Colour and Type.**  
**Add a suitable constructor and a toString method.**

```
1 package raytracer;  
2 public class OrbObject extends SceneObject {  
3  
4     public BoundingSphere bs;  
5  
6     public OrbObject(BoundingSphere bs, Colour c, Type t) {  
7         super(c, t);  
8         this.bs = bs;  
9     }  
10  
11     @Override  
12     public String toString() {  
13         return super.toString() + "\nBS: " + bs;  
14     }  
15  
16     @Override  
17     public Vector getPosition() {  
18         return bs.centre;  
19     }  
20  
21 }
```

### JRGBFrameBuffer.java

So far, we have been implementing basic geometric objects, and the mathematical machinery we need to properly deal with them. And now? Now we are almost done!

The last step is the most complex step, however. We need to find a way to simulate the interactions between the objects we have created so far. And we need a way to create an image for our scene, to display said interactions on screen.

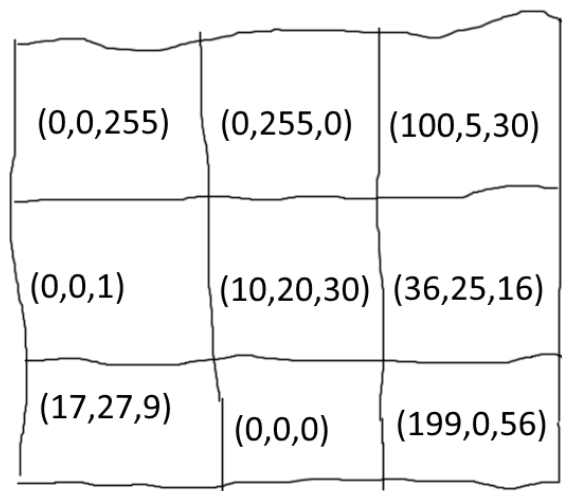
That's what we will do next. We will try to find the simplest representation of an image we can find.

We will do so by introducing the class `JRGBFrameBuffer.java`.

For our purposes, a Frame Buffer is a part of memory that contains a mapping from each pixel of our image to the colour of that pixel. Once we have this mapping, we can write this mapping to a .ppm file. Opening this file, e.g. using GIMP, will then show us our image.

Since we represent colours as 3-tuples of integer values from 0 to 255, we can store our entire buffer as a 1-dimensional array of byte. The primitive data type byte in Java represents integers which can be represented using no more than 8 bits, i.e. numbers  $x$  from 0 to  $2^8 - 1 = 255$ .

Suppose for example we want to represent a grid of pixels, each of which is given by a colour. This will look as follows:



(0,0,255)	(0,255,0)	(100,5,30)
(0,0,1)	(10,20,30)	(36,25,16)
(17,27,9)	(0,0,0)	(199,0,56)

This map can be represented as the 1-dimensional array

`[0, 0, 255, 0, 255, 0, 100, 5, 30, 0, 0, 1, 10, 20, 30, 36, 25, 16, 17, 27, 9, 0, 0, 0, 199, 0, 56]`.

Of course we might as well choose to use a 2-dimensional array, which would be more intuitive. Saying that, it would make it ever so slightly trickier to write our buffer to a file, as we will soon see.

This means that we store 3 integers per pixel, one for its red, one for its blue, and one for its green value.

This number 3 is what we call the number of channels of our image format. If we weren't using the RGB format, we might have a different number of channels (e.g. 4 in the Cyan-Magenta-Yellow-Black Format).

We can store this number 3 as a constant called `NUM_CHANNELS` in our class.

**Create a class `JRGBFrameBuffer.java` and add a constructor that initializes height and width variables, followed by a byte array of the appropriate size. Also introduce getters for the relevant fields.**

```

1 package raytracer;
2+ import java.io.FileOutputStream;
3
4
5 public class JRGBFrameBuffer implements Cloneable {
6     public static final int CHANNEL_NUM = 3;
7
8     private int width;
9     private int height;
10    private int buffSize;
11
12    /** Treat this as unsigned byte by interpreting with: b & 0xFF */
13    private byte[] buffer;
14
15-   public JRGBFrameBuffer(int width, int height) {
16       this.width = width;
17       this.height = height;
18       buffSize = width * height * CHANNEL_NUM;
19       buffer = new byte[buffSize];
20   }
21
22   public int getWidth() { return width; }
23   public int getHeight() { return height; }
24   public int getSize() { return buffSize; }
25
26   -

```

Obviously, we also need get- and set-methods for the colours in our grid.

E.g. setting a colour in our grid would consist of setting the 3 channels it consists of.

Therefore, we first need methods to get/set a single channel in our grid, being given x- and y-coordinates, the number of the relevant channel (0 for R, 1 for G, 2 for B) and, in the case of the set method, the value we would like to insert into our grid.

**Introduce methods to get/set the value of a single channel at a pair of x-/y-coordinates.**

**Ideally, throw an appropriate exception if the method receives invalid parameters (i.e. a channel number that is larger than 2, or a negative x-coordinate).**

Once we have the getChannel and setChannel methods, defining the methods getColour and setColour is straight-forward. The setColour method should take x- and y-coordinates, as well as a colour, while the getColour method should take a pair of coordinates and return a Colour object.

**Add appropriate methods getColour and setColour to your class.**



```

30 public int getChannel(int x, int y, int channel) {
31     if (channel > CHANNEL_NUM - 1 || channel < 0)
32         throw new IllegalArgumentException("Given channel must be between 0 and " + (CHANNEL_NUM - 1) + ": " + channel);
33     if (x < 0 || x >= width)
34         throw new IllegalArgumentException("Given x coordinate out of bounds: " + x);
35     if (y < 0 || y >= height)
36         throw new IllegalArgumentException("Given y coordinate out of bounds: " + y);
37
38     int idx = (y * width + x) * CHANNEL_NUM + channel;
39
40     return buffer[idx] & 0xFF;
41 }
42
43 public void setChannel(int x, int y, int channel, int value) {
44     if (channel > CHANNEL_NUM - 1 || channel < 0)
45         throw new IllegalArgumentException("Given channel must be between 0 and " + (CHANNEL_NUM - 1) + ": " + channel);
46     if (x < 0 || x >= width)
47         throw new IllegalArgumentException("Given x coordinate out of bounds: " + x);
48     if (y < 0 || y >= height)
49         throw new IllegalArgumentException("Given y coordinate out of bounds: " + y);
50
51     int idx = (y * width + x) * CHANNEL_NUM + channel;
52
53     buffer[idx] = (byte) value;
54 }
55
56 public void setColour(int x, int y, Colour c) {
57     setChannel(x, y, 0, c.R);
58     setChannel(x, y, 1, c.G);
59     setChannel(x, y, 2, c.B);
60 }
61
62 public Colour getColour(int x, int y) {
63     return new Colour((byte) getChannel(x, y, 0),
64         (byte) getChannel(x, y, 1), (byte) getChannel(x, y, 2));
65 }

```

Finally, we need a method to write our buffer to a file.

To do so, we can use a technique that you may not have seen before if you only just started coding, but one that is quite intuitive and often useful.

In Java, we can create objects of a class called `FileOutputStream`. Its constructor takes a `String` which is supposed to be the name of the file we wish to write to.

Calling the `write()` method of the object then writes the parameter we pass to the method to the file.

What's amazing is that this parameter might actually be a byte array, like our buffer!

You can check the documentation of `FileOutputStream` online to see how this works.

Note: When we use the method `System.out.println()`, we write text to the standard output stream, the content of which is displayed in our console. So even if you have never really heard of streams before, we have all been using them since writing our first "Hello World" program.

**Add a function `writeToFile` to your class, that takes the name of a file and prints our current buffer to the indicated file.**

**Before printing the contents of the buffer, you will have to print 3 lines of text to the file:**

- The "magic number" of the PPM format, P6.
- The width and height of the image.
- The maximum value a channel in our file can have.

**Because we want to print bytes to the file, you'll want to make use of the method `toBytes` that the class `String` implements.**

**(Have a look at <http://netpbm.sourceforge.net/doc/ppm.html> to see the exact format of a PPM file.)**

```

public void writeToFile(String filename) throws IOException {
    FileOutputStream out = new FileOutputStream(filename);
    out.write(("P6\n" + getWidth() + " " + getHeight() + "\n255\n").getBytes()); // write header
    out.write(buffer);
    out.close();
}

```

### RayTracer.java

The good news: We are only missing a single class now!  
It is however slightly complex. But we'll get there.

What do we want our "main" Raytracer class to do?

We want to create a scene, simulate the interaction of our objects and our light rays and "paint" the resulting image to a file, using the resources we have created thus far.

Let's start filling in the main method with everything we know we will need.

- 1 - Create an image variable with appropriate width and height (use our class `JRGBFrameBuffer`)
- 2 -- Declare and initialize a String variable with our intended file name (e.g. "image.ppm")
- 3 - Create our "scene" - An array containing our scene objects, e.g. 2 orb objects and a light object

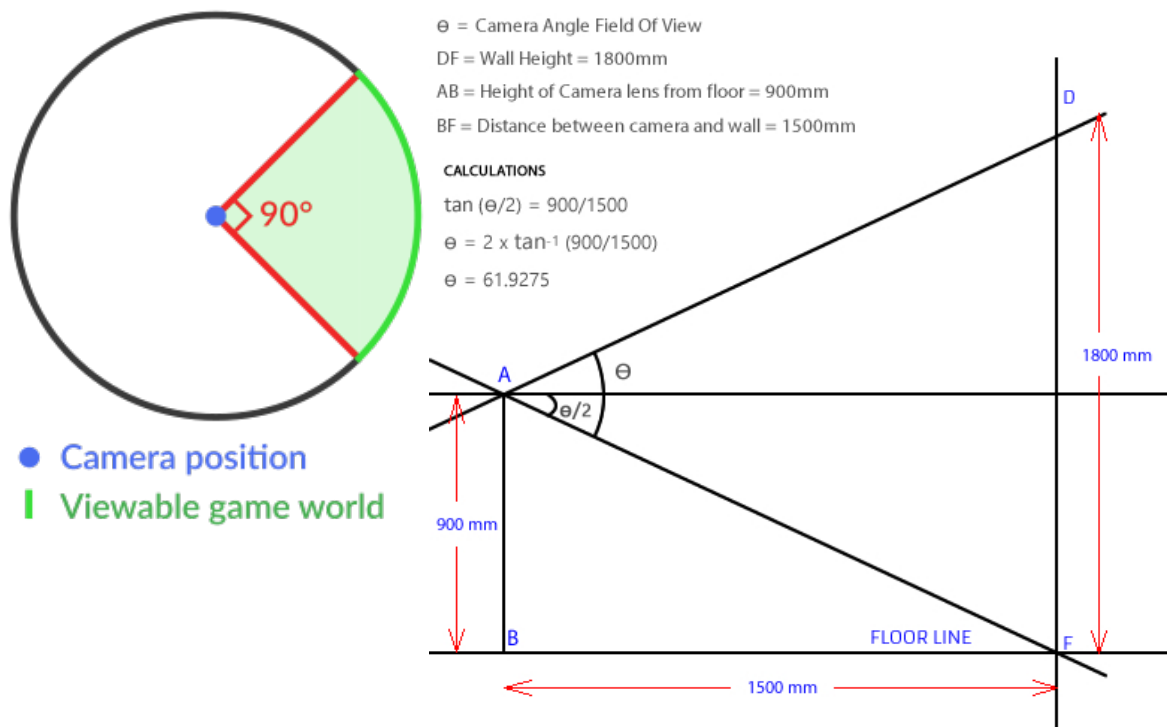
Then, we will need several variables to help us compute the interaction of rays and objects

- 4 - We keep the inverse of the width and height ( $1/\text{width}$  ,  $1/\text{height}$ ) as double variables

We need to know how wide our field of view is supposed to be.

- 5 - Our Width:Height ratio is determined by the width and height of our image. We call the according double variable "aspectratio".

- 6 - How wide our view of our scene is, depends on the angle between the left and right end of our field of view (see image). We keep this value as an integer. Take e.g. 45 degrees.



7 - Knowing the ratio between the height of our FOV (field of view) and the distance between our camera and our "canvas" is crucial when painting our scene - afterall, we need to know how big everything needs to look.

The image above teaches you the formula for this ratio and why it works.

Since Java's Tan method takes its angle parameter to be in radians and not in degrees, we need to multiply our angle variable by Pi and divide it by 180, to convert it from degrees to radians.

Our scene takes place in a Euclidean coordinate system. We define its origin to be the position of our "camera".

8 - Create a Vector element called camera, corresponding to position (0,0,0)

```

8 public class RayTracer {
9
10 public static void main(String[] args) {
11
12     final String imgOutName = "image.ppm";
13     final int imageWidth = 1000;
14     final int imageHeight = 800;
15     final int MAX_DEPTH = 3;
16     JRGBFramebuffer image = new JRGBFramebuffer(imageWidth, imageHeight);
17
18     SceneObject[] scene = {
19         new LightObject(new Vector(-50, 50, 250), Colour.WHITE, SceneObject.Type.POINTLIGHT),
20         new LightObject(new Vector(-50, -50, 250), Colour.WHITE, SceneObject.Type.POINTLIGHT),
21         new OrbObject(new BoundingSphere(new Vector(0, -10, 300), 20), Colour.RED, SceneObject.Type.SP),
22         new OrbObject(new BoundingSphere(new Vector(50, 20, 300), 30), Colour.BLUE, SceneObject.Type.SP);
23     };
24
25     System.out.println("Tracing image: " + imageWidth + "x" + imageHeight + " ...");
26
27     double invWidth = 1 / (double) imageWidth;
28     double invHeight = 1 / (double) imageHeight;
29     double angle = 30;
30     double angleRad = angle * Math.PI / 180.0;
31     double aspectratio = imageWidth / (double) imageHeight;
32     double fovToDistance = Math.tan(0.5 * angleRad);
33
34     Vector camera = new Vector(0, 0, 0);
35
36

```

Before we can finish our main method however, we need to take care of the remaining computational aspects.

**9 – Add an intersect method to your Utils class, returning a Double object that denotes the closest intersection distance between a Ray and the BoundingSphere of an orb object, given to the method as parameters.**

(Note: The maths needed to make this work are a bit tricky. You can follow the formulas given by [https://en.wikipedia.org/wiki/Line%E2%80%93sphere\\_intersection](https://en.wikipedia.org/wiki/Line%E2%80%93sphere_intersection) )

```

/**
 * Returns the closest intersection distance of r and s
 * or null if there is none.
 */
public static Double intersect(Ray r, BoundingSphere s) {
    Double res;

    double a = dot(r.dir, r.dir);
    double b = dot(scalar(r.dir, 2), subtract(r.origin, scentre));
    double c = dot(subtract(r.origin, scentre), subtract(r.origin, scentre)) - Math.pow(s.radius, 2);

    double sqrtTerm = b*b - 4*a*c;

    if(sqrtTerm < 0) res = null;
    else if(sqrtTerm == 0) {
        res = (-1 * b) / (2*a);
        if(Math.abs(res) < EPSILON) res = null;
    } else {
        sqrtTerm = Math.sqrt(sqrtTerm);
        double i1 = (-1 * b + sqrtTerm) / (2*a);
        double i2 = (-1 * b - sqrtTerm) / (2*a);

        if(Math.abs(i1) < EPSILON) res = i2;
        else if(Math.abs(i2) < EPSILON) res = i1;
        else res = Math.min(i1, i2);
    }

    return res;
}

```

To capture the intersection of a ray and an object, we can create a helper class: You could define it in a separate file, but since we will only use it inside of the Raytracer class, we might as well turn it into a nested class.

Beginners should generally avoid doing this but if used correctly, they can make programs simpler and more concise.

Note: This Intersection class should be static. Otherwise we could not create instances of it without creating an instance of (the “outer class”) Raytracer first. And we don’t want to do that, we would like to just run the class’s main method. Only nested (i.e. “inner”) classes can be static.

**10 – Create an inner class Intersection, capturing the properties of the intersection of a ray and a scene object. You need to add variables for the object the ray intersects with, and for the distance from the source of the ray at which the intersection occurs.**

More than one scene object may lie on the path of a light ray. Assuming the object is not transparent, only the intersection with the lowest intersection distance will matter. Therefore, we need a way of comparing two intersections.

**11 – Add a method compareTo to the Intersection class, which takes an Intersection “o” as a parameter and returns  
-1 if the Intersection has a lower distance than o,  
0 if the Intersections have equal distances,  
1 otherwise.**

This way of implementing a comparison operation corresponds to the Java standard. Therefore, we can say that Intersection should implement the interface Comparable<Intersection>, which simply denotes that Intersection must have a compareTo method that takes another Intersection as its parameter and returns an integer. This will be insanely useful shortly.

Now, we want to be able to find all intersections of a ray with objects of our scene. Intersections between a ray and a POINTLIGHT object don’t really make sense, so we ignore these. If we encounter an unexpected type of object, we throw an exception. That might e.g. happen if we add new types of objects to our code and don’t remember accounting for them in our getIntersects method. We use the type variable of our scene objects for easy type checking.

**12 – Add a method getIntersects to the Raytracer class, taking a Ray and an array of SceneObjects as inputs and returning a list of all Intersections of the Ray, sorted from smallest to largest intersection distance.**

(Note: Once we have created an unsorted collection of intersections, we can use the sort method of the Java library class Collections. Because of the way we implemented our compareTo method, it sorts the intersections just right.)

For the intersection of two spheres, we have already created a method in the Utils class.

```

private static List<Intersection> getIntersects(Ray r, SceneObject[] scene) {
    List<Intersection> inters = new ArrayList<Intersection>();
    for(SceneObject obj : scene) {
        Double dist = null;
        switch(obj.type) {
            case POINTLIGHT: // no intersections with point light sources
                continue;
            case SPHERE:
                dist = Utils.intersect(r, ((OrbObject)obj).bs);
                break;
            default:
                throw new RuntimeException("Unknown object type: " + obj.type.name());
        }

        if(dist != null && dist > 0) {
            inters.add(new Intersection(dist, obj));
        }
    }

    Collections.sort(inters);
    return inters;
}

```

Raytracing is all about light. And to make things more interesting, we can work with colours. Where 2 RGB colours mix, we want to blend them. Given two Colours, our blendColours method should return the “mean” of the two inputs.

### 13 - Add a method blendColours(Colour c1, Colour c2), which returns the mean of the two inputs.

But where does the magic happen? Well, we have two methods to go.

What a raytracer does is simulate the interactions of a light ray with the objects in our scene, to create an image. We call this process “rendering”.

Our render method is given a Ray and the list of our SceneObjects as inputs. It should return the colour that is being produced. (The input ray will be a ray starting at the camera, corresponding to our line of sight)

First, we compute the intersections of the ray and the objects, and then pick out the intersection with the lowest intersection distance.

If the according object is a pointlight source, we return the colour of the light.

If the list of intersections is empty, the ray is invisible and we return null.

Otherwise, the ray hits a solid object and things get complicated. We call our final method: renderDiffuse.

### 14 - Add a method render(Ray r, SceneObject[] scene), which returns the Colour that we (positioned at the origin of the input ray) see. If the ray hits a solid object, call the renderDiffuse method, which is the last method we need to implement and takes the according ray and intersection, as well as our list of scene objects as inputs.

In renderDiffuse, first compute the position of the intersection. Since the direction vector of our ray definitely has length 1, this position is the vector sum of the ray’s origin and its direction vector times the intersection distance.

We declare a Colour variable which we want to return at the end of the method body. By default, it gets the value null. # Shouldn’t it be black instead?

Then, for every point light object in our scene:

Compute the vector from the source to the intersection position, then create a Ray according to this light ray. If the ray intersects with some object, we update our result by blending it with the colour black. This just makes the colour darker, corresponding to the light ray being swallowed by another object, before reaching the intersection of our initial ray and the object. This is not exact, but yields decent results for now. In a scene without light objects, everything will be dark.

If this new light ray is not swallowed however, the intersection position is illuminated and things get interesting:

We compute a normalized vector `surfNormal`, pointing from the intersection position to the center of the corresponding object.

We then compute a vector corresponding to the direction of the reflection of our initial ray, bouncing off the object it hit. The formula for this `reflecV` is the initial direction vector, minus the scalar product of `surfNormal` and twice the dot product of the direction vector and `surfNormal`. This is difficult to make mathematically precise, but corresponds to the ray being reflected more strongly/weakly depending on the angle at which it hits the object.

We save the angle between `reflecV` and the light vector stemming from the point light source.

Based on said angle, we compute the intensity of the effect the light source has on the colour being produced. The angle lies in the range  $[0, \pi]$ . We divide it by  $(-1)$  and add 1 to it, resulting to an intensity in the range  $[0, 1]$ . The smaller the angle between our initial ("sight") ray and the light ray, the larger the effect the light ray has on the colour we observe.

We then define two Colour variables:

The colour of the object, and the colour of the light source, the R, G and B values of which we multiply by the intensity we computed earlier.

We then blend these colours, before blending the outcome of this with our preliminary result variable. We update the result variable by setting it to this new mixture.

Once this has been done for all light sources, we return the result variable.

**15 - Add a method `renderDiffuse`, taking a ("sight" ray, an intersection and the list of our scene objects as inputs, and returns the colour that is being produced by the interaction of the sight ray, the object, and all light objects in our scene.**

Awesome! And now let's wrap this all up by finishing our main method. Then we are done and can run our program to produce more or less beautiful images!

After initializing the camera vector in our main method, we initialize our image: an `JRGBFrameBuffer` object with the width and height of the image we would like to produce.

Then we need to produce our "sight rays". (or "primary - " / "camera - " / "eye rays")

We need to loop over all pixels of our image, generate a ray for each pixel, cast it into the scene and call our "render" function to look for possible interactions with our scene objects.

(How these primary rays are computed is fairly complex, have a look at this article if you want to understand how it works: <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays>).

After calling the render method, we set the colour at the current pixel to that colour, or to grey if the result was null.

Having looped over all pixels, there is just one thing left to do: Write the image to a file, using our writeToFile method in JRGBFrameBuffer. And that's it. If everything went well, we should be able to produce simple images of spheres using raytracing now.

**16 - Wrap up the main method, producing „sight rays“ / “primary rays“ for every pixel of our image. Then write the image to a file. Congratulations! You have completed this tutorial. Now you can play around with the program, debug it if you have to, or maybe even extend its functionality.**

You could think about how the renderDiffuse method could be more sophisticated, or how interactions with non-spherical objects could be implemented.

But for now: Good job, and have a nice day!