9. homework assignment; JAVA, Academic year 2018/2019; FER

Napravite prazan Maven projekt, kao u 1. zadaći: u Eclipsovom workspace direktoriju napravite direktorij `hw09-0000000000` (zamijenite nule Vašim JMBAG-om) te u njemu oformite Mavenov projekt `hr.fer.zemris.java.jmbag0000000000:hw09-0000000000` (zamijenite nule Vašim JMBAG-om) i dodajte ovisnost prema biblioteci `junit`. Importajte projekt u Eclipse. Sada možete nastaviti s rješavanjem zadataka. Potrebne biblioteke koje se spominju smjestite u direktorij `lib` unutar projekta (napravite ga), i zatim podesite `pom.xml` tako da koristi te biblioteke. Pri tome ćete JAR-arhive koje sadrže bytekod importati u Vaš lokalni maven-repozitorij; uputa o točnim koordinatama dana je kroz daljnji tekst. Direktorij `lib` NE uploadate u okviru predaje Vaše zadaće.

Problem 1.

Radimo vrlo jednostavnu biblioteku koja nudi potporu za rad s nekoliko različitih matematičkih objekata: vektori, kompleksni brojevi te polinomi.

Napravite paket `hr.fer.zemris.math` i u njega smjestite modele vektora, kompleksnih brojeva i polinoma.

Napravite razred `Vector3` koji modelira neizmjenjiv trokomponentni vektor (sve operacije nad njime vraćaju nove objekte ovog tipa koji predstavljaju rezultat operacije). Javno sučelje ovog razreda specificirano je u nastavku.

```
public Vector3(double x, double y, double z) {…}  // konstruktor

public double norm() {...}  // norma vektora (“duljina”)

public Vector3 normalized() {...}  // normalizirani vektor

public Vector3 add(Vector3 other) {...}  // zbrajanje

public Vector3 sub(Vector3 other) {...}  // oduzimanje: ja minus drugi

public double dot(Vector3 other) {...}  // skalarni produkt

public Vector3 cross(Vector3 other) {...}  // vektorski produkt: ja puta on

public Vector3 scale(double s) {...}  // skaliranje zadanim faktorom

public double cosAngle(Vector3 other) {...}  // kosinus kuta između mene i njega

public double getX() {...}  // prva komponenta vektora

public double getY() {...}  // druga komponenta vektora

public double getZ() {...}  // treća komponenta vektora

public double[] toArray() {...}  // pretvorba u polje s 3 elementa

public String toString() {...}  // pretvorba u string
```

Primjer ispitnog programa:

```java
public static void main(String[] args) {
	Vector3 i = new Vector3(1,0,0);
	Vector3 j = new Vector3(0,1,0);
	Vector3 k = i.cross(j);

	Vector3 l = k.add(j).scale(5);

	Vector3 m = l.normalized();

	System.out.println(i);
	System.out.println(j);
	System.out.println(k);
	System.out.println(l);
	System.out.println(l.norm());
	System.out.println(m);
	System.out.println(l.dot(j));
	System.out.println(i.add(new Vector3(0,1,0)).cosAngle(l));
}
```

Očekivani ispis:

```
(1.000000, 0.000000, 0.000000)
(0.000000, 1.000000, 0.000000)
(0.000000, 0.000000, 1.000000)
(0.000000, 5.000000, 5.000000)
7.0710678118654755
(0.000000, 0.707107, 0.707107)
5.0
0.4999999999999999
```

U nastavku ćemo napraviti još model kompleksnog broja, te dva modela polinoma koji su zadani nad kompleksnim brojevima, i čiji su koeficijenti kompleksni brojevi. Sva tri modela moraju stvarati neizmjenjive objekte.

Napravite razred `Complex` koji modelira kompleksni broj, prema predlošku u nastavku.

```java
public class Complex {

	...

	public static final Complex ZERO = new Complex(0,0);
	public static final Complex ONE = new Complex(1,0);
	public static final Complex ONE_NEG = new Complex(-1,0);
	public static final Complex IM = new Complex(0,1);
	public static final Complex IM_NEG = new Complex(0,-1);

	public Complex() {...}

	public Complex(double re, double im) {...}

	// returns module of complex number
	public double module() {...}

	// returns this*c
	public Complex multiply(Complex c) {...}

	// returns this/c
	public Complex divide(Complex c) {...}

	// returns this+c
	public Complex add(Complex c) {...}

	// returns this-c
	public Complex sub(Complex c) {...}

	// returns -this
	public Complex negate() {...}

	// returns this^n, n is non-negative integer
	public Complex power(int n) {...}

	// returns n-th root of this, n is positive integer
	public List<Complex> root(int n) {...}

	@Override
	public String toString() {...}
}
```

Napravite razred `ComplexRootedPolynomial` koji modelira polinom nad kompleksim brojevima, prema predlošku u nastavku. Radi se o polinomu $f(z)$ oblika $z_0*(z-z_1)*(z-z_2)*...*(z-z_n)$, gdje su z_1 do z_n njegove nultočke a z_0 konstanta (sve njih zadaje korisnik kroz konstruktor). Primjetite, radi se o polinomu n-tog stupnja (kada biste izmnožili zagrade). Svi z_i zadaju se kao kompleksni brojevi, a i sam z je kompleksan broj. Metoda *apply* prima neki konkretan z i računa koju vrijednost ima polinom u toj točki.

```
public class ComplexRootedPolynomial {

	// ...

	// constructor
	public ComplexRootedPolynomial(Complex constant, Complex ... roots) {...}

	// computes polynomial value at given point z
	public Complex apply(Complex z) {...}

	// converts this representation to ComplexPolynomial type
	public ComplexPolynomial toComplexPolynom() {...}

	@Override
	public String toString() {...}

	// finds index of closest root for given complex number z that is within
	// treshold; if there is no such root, returns -1
	// first root has index 0, second index 1, etc
	public int indexOfClosestRootFor(Complex z, double treshold) {...}
}
```

Napravite razred `ComplexPolynomial` koji modelira polinom nad kompleksim brojevima, prema predlošku u nastavku. Radi se o polinomu $f(z)$ oblika $z_n*z^n+z^{n-1}*z_{n-1}+...+z_2*z^2+z_1*z+z_0$, gdje su z_0 do z_n koeficijenti koji pišu uz odgovarajuće potencije od z (i zadaje ih korisnik kroz konstruktor). Primjetite, radi se o polinomu n-tog stupnja (što još zovemo red – engl. *polinom order*). Svi koeficijenti zadaju se kao kompleksni brojevi, a i sam z je kompleksan broj. Metoda *apply* prima neki konkretan z i računa koju vrijednost ima polinom u toj točki. Redoslijed faktora predanih u konstruktoru s lijeva na desno se tumači kao z_0, z_1, z_2, ...

```
public class ComplexPolynomial {

	// ...

	// constructor
	public ComplexPolynomial(Complex ...factors) {...}

	// returns order of this polynom; eg. For (7+2i)z^3+2z^2+5z+1 returns 3
	public short order() {...}

	// computes a new polynomial this*p
	public ComplexPolynomial multiply(ComplexPolynomial p) {...}

	// computes first derivative of this polynomial; for example, for
	//  (7+2i)z^3+2z^2+5z+1 returns (21+6i)z^2+4z+5
	public ComplexPolynomial derive() {...}

	// computes polynomial value at given point z
	public Complex apply(Complex z) {...}

	@Override
	public String toString() {...}
}
```

Evo u nastavku jednostavnog primjera.

```
ComplexRootedPolynomial crp = new ComplexRootedPolynomial(
      new Complex(2,0), Complex.ONE, Complex.ONE_NEG, Complex.IM, Complex.IM_NEG
);
ComplexPolynomial cp = crp.toComplexPolynom();
System.out.println(crp);
System.out.println(cp);
System.out.println(cp.derive());
```

Daje okvirni ispis:

```
(2.0+i0.0)*(z-(1.0+i0.0))*(z-(-1.0+i0.0))*(z-(0.0+i1.0))*(z-(0.0-i1.0))
(2.0+i0.0)*z^4+(0.0+i0.0)*z^3+(0.0+i0.0)*z^2+(0.0+i0.0)*z^1+(-2.0+i0.0)
(8.0+i0.0)*z^3+(0.0+i0.0)*z^2+(0.0+i0.0)*z^1+(0.0+i0.0)
```

Problem 2.

We will consider another kind of fractal images: fractals derived from Newton-Raphson iteration. As you are surely aware, for about three-hundred years we know that each function that is *k*-times differentiable around a given point x_0 can be approximated by a *k*-th order Taylor-polynomial:

$$f(x_0+\varepsilon)=f(x_0)+f'(x_0)\varepsilon+\frac{1}{2!}f''(x_0)\varepsilon^2+\frac{1}{3!}f'''(x_0)\varepsilon^3+\dots$$

So let x_1 be that point somewhere around the x_0:

$$x_1=x_0+\varepsilon$$

Substituting it into previously given formula we obtain:

$$f(x_1)=f(x_0)+f'(x_0)(x_1-x_0)+\frac{1}{2!}f''(x_0)(x_1-x_0)^2+\frac{1}{3!}f'''(x_0)(x_1-x_0)^3+\dots$$

For approximation of function *f* we will restrict our self on linear approximation, so we can write:

$$f(x_1)\approx f(x_0)+f'(x_0)(x_1-x_0)$$

Now, let us assume that we are interested in finding x_1 for which our function is equal to zero, i.e. we are looking for x_1 for which $f(x_1)=0$. Plugging this into above approximation, we obtain:

$$0=f(x_0)+f'(x_0)(x_1-x_0)$$

and from there:

$$x_1=x_0-\frac{f(x_0)}{f'(x_0)}$$

However, since we used the approximation of *f*, it is quite possible that $f(x_1)$ is not actually equal to zero; however, we hope that $f(x_1)$ will be closer to zero than it was $f(x_0)$. So, if that is true, we can iteratively apply this expression to obtain better and better values for *x* for which $f(x)=0$. So, we will use iterative expression:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

which is known as Newton-Raphson iteration.

For this homework we will consider complex polynomial functions. For example, lets consider the complex polynomial whose roots are +1, -1, *i* and -*i*:

$$f(z) = (z-1)(z+1)(z-i)(z+i) = z^4 - 1$$

After deriving we obtain:

$$f'(z) = 4z^3$$

It is easy to see that our function *f* becomes 0 for four distinct complex numbers *z*. However, we will pretend that we don't know those roots. Instead, we will start from some initial complex point *c* and plug it into our iterative expression:

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)} = x_n - \frac{z^4 - 1}{4z^3} \quad \text{with} \quad z_0 = c \quad .$$

We will generate iterations until we reach a predefined number of iterations (for example 16) or until module $|z_{n+1}-z_n|$ becomes adequately small (for example, convergence threshold 1E-3). Once stopped, we will find the closest function root for final point z_n, and color the point *c* based on index of that root (let root indexes start from 1). However, if we stopped on a z_n that is further than predefined threshold from all roots, we will color the point *c* with a color associated with index 0.

For example, if the function roots are +1, -1, *i* and -*i*, if acceptable root-distance is 0.002, if convergence threshold equals 0.001 and if we stopped iterating after z_7=-0.9995+*i*0 because z7 was closer to z_6=-0.9991+*i*0 then convergence threshold, we will determine that z_7 is closest to second function root (first is +1, second is -1, third is +i, fourth is -i) and that z_7 is within predetermined root-distance (0.002) to -1, so we will color pixel *c* based on color associated with index 2. Since `ComplexRootedPolynomial.indexOfClosestRootFor` returns 0-based indexes, in pseudocode below we make coloring based on the returned index value incremented by 1.

We will proceed just as with Mandelbrot fractal:

```
for(y in ymin to ymax) {
  for(x in xmin to xmax) {
    c = map_to_complex_plain(x, y, xmin, xmax, ymin, ymax, remin, remax, immin, immax);
    zn = c;
    iter = 0;
    iterate {
      znold = zn;
      zn = zn – f(zn)/f'(zn);
      iter++;
    } while(|zn-znold|>convergenceTreshold && iter<maxIter);
    index = findClosestRootIndex(zn, rootTreshold);
    data[offset++]=index+1;
  }
}
```

We use *data[]* array same way as we did for Mandelbrot fractal and the GUI component will handle the rest; the only difference here is that content of *data[]* array does not represent the speed of divergence but instead holds the indexes of roots in which observed complex point *c* has converged or 0 if no convergence to a root occurred. Another difference is that the upper limit to *data[i]* is number of roots, so we won't call observer with:

```
observer.acceptResult(data, (short)(m), requestNo);
```

but instead with:

```
observer.acceptResult(data, (short)(polynom.order()+1), requestNo);
```

If you completed this correct, for our first example with roots +1, -1, +i and -i you will get the following picture:

In order to solve this, install in your local maven repository jar `fractal-viewer-1.0.jar` under `hr.fer.zemris.java.fractals:fractal-viewer:1.0`, and add it as dependency to your `pom.xml`. Javadoc is available as separate jar.

More verbose introduction to fractals based on Newton-Raphson iteration can be found at: http://www.chiark.greenend.org.uk/~sgtatham/newton/

Details

Given the classes you developed in problem 1, the core of iteration loop can be written as:

```
Complex numerator = polynomial.apply(zn);
Complex denominator = derived.apply(zn);
Complex znold = zn;
Complex fraction = numerator.divide(denominator);
Complex zn = zn.sub(fraction);
module = znold.sub(zn).module();
```

Write a main program `hr.fer.zemris.java.fractals.Newton`. The program must ask user to enter roots as given below (observe the syntax used), and then it must start fractal viewer and display the fractal. In order to run this successfully, you will have to add classpath configuration argument in command line when starting `java`.

```
C:\somepath> java hr.fer.zemris.java.fractals.Newton
Welcome to Newton-Raphson iteration-based fractal viewer.
Please enter at least two roots, one root per line. Enter 'done' when done.
Root 1> 1
Root 2> -1 + i0
Root 3> i
Root 4> 0 - i1
Root 5> done
Image of fractal will appear shortly. Thank you.
```

(user inputs are shown in red)

General syntax for complex numbers is of form *a+ib* or *a-ib* where parts that are zero can be dropped, but not both (empty string is not legal complex number); for example, zero can be given as 0, i0, 0+i0, 0-i0. If there is '*i*' present but no *b* is given, you must assume that *b=1*.

The implementation of `IFractalProducer` that you will supply must use parallelization to speed up the rendering. The range of *y*-s must be divided into 8 * `numberOfAvailableProcessors` jobs. For running your jobs you must use `ExecutorService` based on `FixedThreadPool`, and you must collect your jobs by calling `get()` on provided `Future` objects. Do not create new `ExecutorService` for each call of method `produce`. Instead, create it in producer's constructor. Use a variant of `FixedThreadPool` which allows you to specify a custom `ThreadFactory` as last argument. Implement a `DaemonicThreadFactory` that produces threads which have daemon flag set to `true` and pass an instance of this factory to the `newFixedThreadPool`; this way, your program won't hang once the GUI is closed.

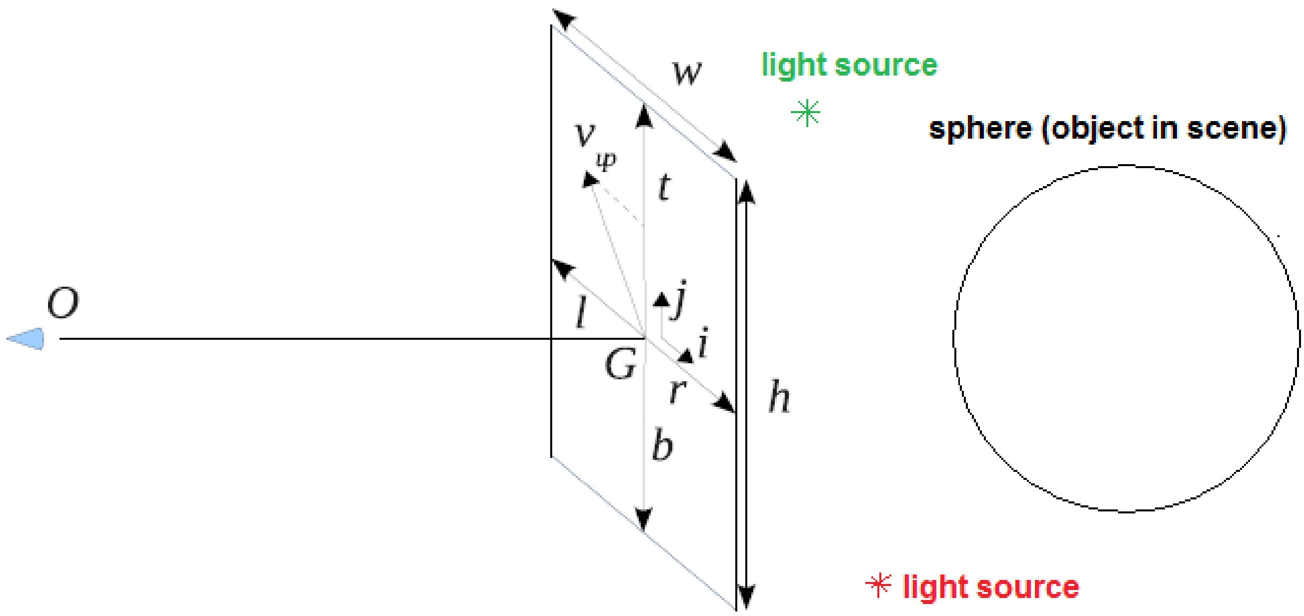
Problem 3.

You will write a simplification of a ray-tracer for rendering of 3D scenes; don't worry – it's easy and fun. And also, we won't write a full-blown ray-tracer but only a ray-caster.

Please download from Ferko repository `raytracer-1.0.jar`; javadoc is available as separate jar. Install primary jar in your local maven repository as `hr.fer.zemris.java.raytracer:raytracer:1.0`. To better understand the needed theory, you are advised to download the book available at:

http://java.zemris.fer.hr/nastava/irg/

(version knjiga-0.1.2016-03-02.pdf) and read section 9.2 (Phong model, pages 231 to 236) and section 10.2 (Ray-casting algorithm, pages 241 to 244). To render an image using ray-casting algorithm, you start by defining which object are present in the 3D scene, where are you stationed (eye-position: O), where do you look at (view position: G) and in which direction is "up" (view-up approximation). See next image.



Now imagine that you have constructed a plane perpendicular to vector that connects the eye position (O) and the view point (G). In that plane you will create a 2D coordinate system, so you will have the x-axis (as indicated by vector *i* on the image) and the y-axis (as indicated by vector *j* on the image). If you only start with an eye-position and a view point, your y-axis can be arbitrarily placed in this plane (you could rotate it for any angle). To help us fix the direction of the y-axis, it is customary to specify another vector: the *view-up* vector which does not have to lay in the plane but it also must not be co-linear with G-O vector, so that a projection of this vector onto the plane exists. If this is true, then take a look at the projection of the view-up vector into the plane: we will use the normalized version of this projection to become our *j* vector and hence determine the orientation of y-axis.

Lets start calculating. Let: $\vec{OG}=\frac{\vec{G}-\vec{O}}{\|\vec{G}-\vec{O}\|}$, i.e. it is the normalized vector from $\vec{O}$ to $\vec{G}$; let $\vec{VUV}$ be normalized version of the view-up vector. Then we can obtain the $\vec{j}'$ vector as follows:

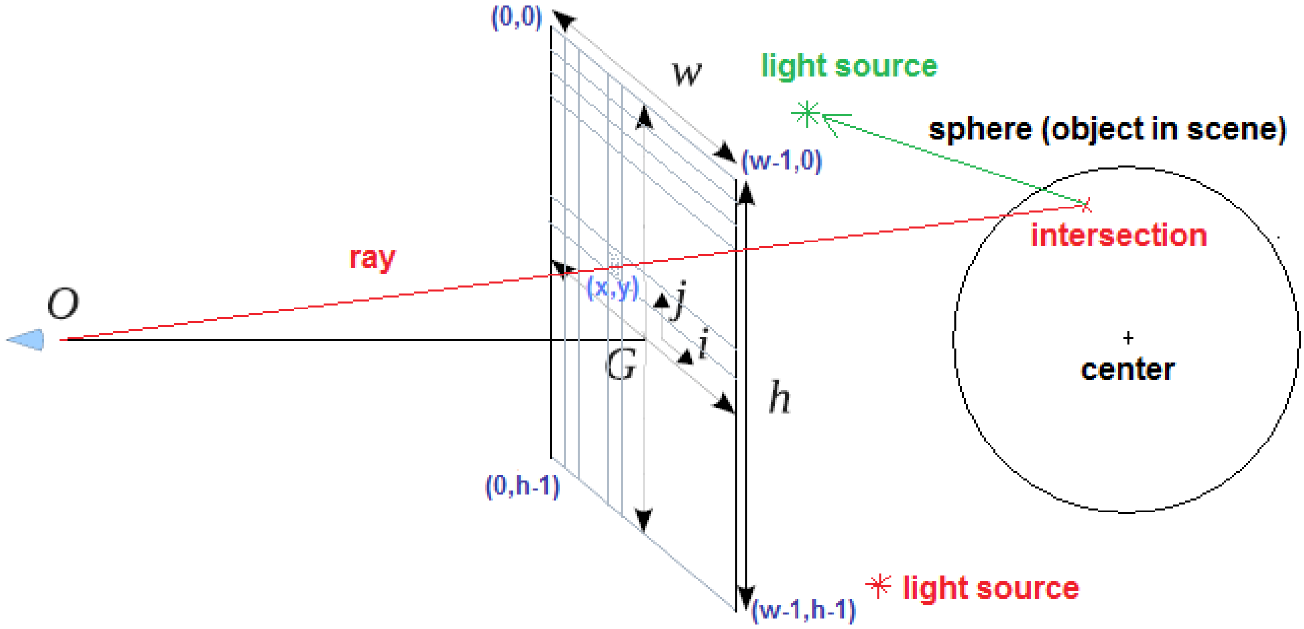
$\vec{j}'=\vec{VUV}-\vec{OG}(\vec{OG}\cdot\vec{VUV})$ where $\vec{OG}\cdot\vec{VUV}$ is a scalar product. Define its normalized version to be:

$\vec{j}=\frac{\vec{j}'}{\|\vec{j}'\|}$. Now we can calculate vector $\vec{i}'$ which will determine the orientation of the x-axis as a cross product: $\vec{i}'=\vec{OG}\times\vec{j}$ and its normalized version $\vec{i}=\frac{\vec{i}'}{\|\vec{i}'\|}$.

Once we have determined where the plane is and what are the vectors determining our x-asis (i.e. $\vec{i}$) and y-axis (i.e. $\vec{j}$), we have to decide which part of this plane will be mapped to our screen so that we can determine where inthis plaine each screen-pixel is located (where is position (0,0), (0,1), etc.). We will assume it to be a rectangle going left from $\vec{G}$ (i.e. in direction $-\vec{i}$) for *l*, going right from $\vec{G}$ (i.e. in direction $\vec{i}$) for *r*, going up from $\vec{G}$ (i.e. in direction $\vec{j}$) for *t*, and finally going down from $\vec{G}$ (i.e. in direction $-\vec{j}$) for *b*. To simplify things further, lets assume that $l=r=\frac{horizontal}{2}$ and $t=b=\frac{vertical}{2}$ where we introduced two parameters: *horizontal* and *vertical*.

In provided libraries I have already prepared the class `Point3D` with implemented methods for calculation of scalar products, cross-products, vector normalization etc. so use it.

Now we will define final screen coordinate system, as shown in the next image.



We will define (0,0) to be the upper left point of our rectangular part of the plane; the x-axis will be oriented just as $\vec{i}$ vector is, and the y-axis will be oriented opposite from $\vec{j}$ vector. We can obtain the 3D coordinates of our upper-left corner as follows:

$$\vec{corner}=\vec{G}-\frac{horizontal}{2}\cdot\vec{i}+\frac{vertical}{2}\cdot\vec{j}$$

Now for each *x* from 0 to *w*-1 and for each *y* from 0 to *h*-1 we can calculate the 3D position of the screen-pixel (*x*,*y*) in the plane as follows:

$$\vec{point}_{xy}=\vec{corner}+\frac{x}{w-1}\cdot horizontal\cdot\vec{i}-\frac{y}{h-1}\cdot vertical\cdot\vec{j}$$

And now it is simple: we define a ray of light which starts at $\vec{O}$ and passes through $\vec{point}_{xy}$ .Then we check if this ray which is specified by starting point $\vec{O}$ and normalized directional vector

$\vec{d}=\frac{\vec{point}_{xy}-\vec{O}}{\|\vec{point}_{xy}-\vec{O}\|}$ has any intersections with objects in scene! If an intersection is found, then that is exactly what will determine the color of screen-pixel (x,y). If no intersection is found, the pixel will be rendered black (r=g=b=0). However, if an intersection is found, we must determine the color of the pixel. If multiple intersections are found, we must chose the closest one to eye-position since that is what the human observer will see. For coloring we will use Phong's model which assumes that there is one or more point-light-sources present in scene. In our example there are two light sources (one green and one red in the previous image). Each light source is specified with intensities of r, g and b components it radiates.

Here is the pseudo code for the above described procedure:

```
for each pixel (x,y)
  calculate ray r from eye-position to pixel_xy
  determine closest intersection S of ray r and any object in the scene (in front of observer)
  if no S exists, color (x,y) with rgb(0,0,0) else use rbg(determineColorFor(S))
```

The procedure `determineColorFor(S)` is given by the following pseudocode:

```
set color = rgb(15,15,15) // i.e. ambient component
for each light source ls
  define ray r' from ls.position to S
  find closest intersection S' of r' and any objects in scene
  if S' exists and is closer to ls.position than S, skip this light source (it is obscured by that object!)
  else color += diffuse component + reflective component
```

Details

Go through sources of `IRayTracerProducer`, `IRayTracerResultObserver`, `GraphicalObject`, `LightSource`, `Scene`, `Point3D`, `Ray` and `RayIntersection` (separate jar is available on Ferko; do not add this into your project!!!). Create package `hr.fer.zemris.java.raytracer.model` in your homework and add class `Sphere`:

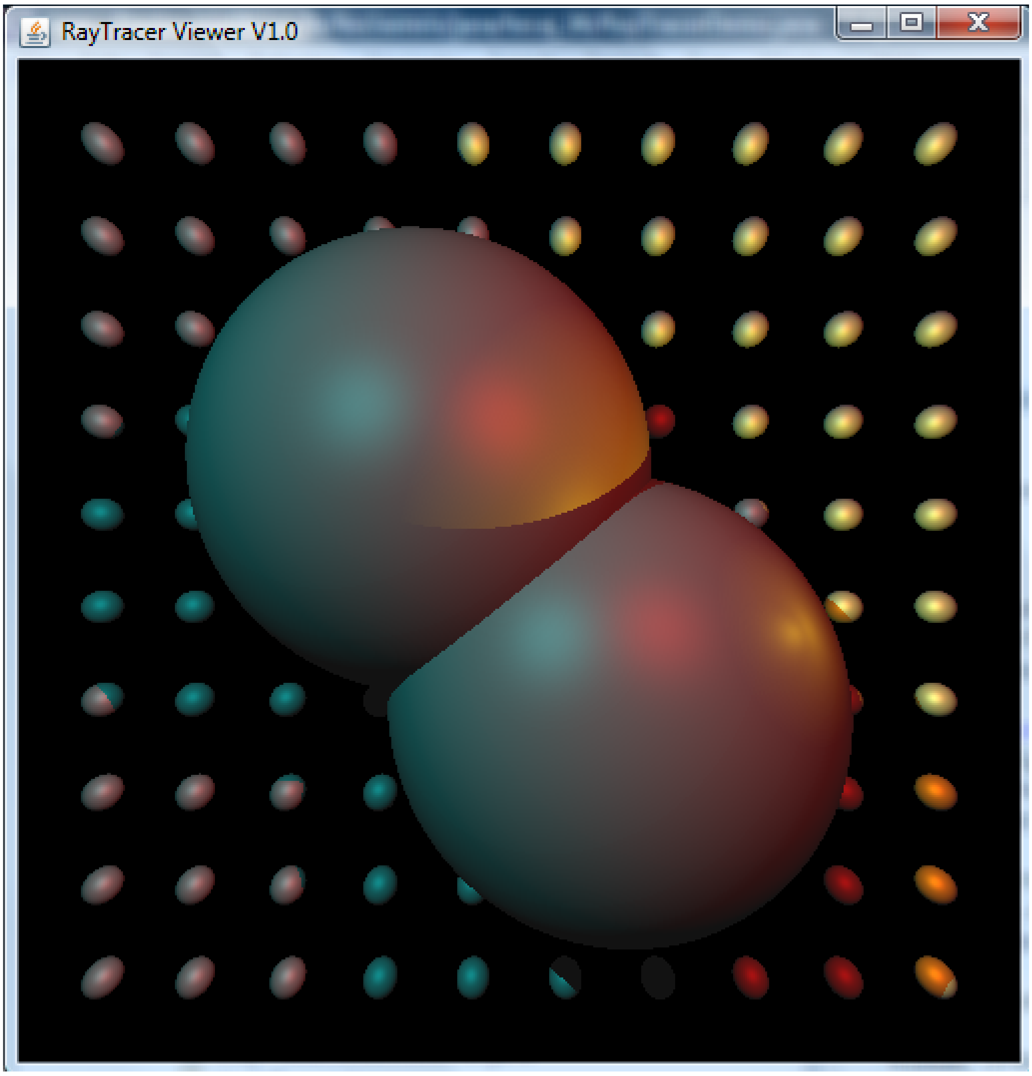
```java
package hr.fer.zemris.java.raytracer.model;

public class Sphere extends GraphicalObject {

	...

	public Sphere(Point3D center, double radius, double kdr, double kdg,
		double kdb, double krr, double krg, double krb, double krn) {
		...
	}

	public RayIntersection findClosestRayIntersection(Ray ray) {
		...
	}
}
```

and implement all that is missing. Until you do that, the method which is used to build the default scene will not work (`RayTracerViewer.createPredefinedScene()`). Coeficients $kd*$ determine the object parameters for diffuse component and $kr*$ for reflective components; krn is shininess factor (n).

Write a main program `hr.fer.zemris.java.raytracer.RayCaster`. The basic structure of the program should look like this:

```java
public static void main(String[] args) {
      RayTracerViewer.show(getIRayTracerProducer(),
      new Point3D(10,0,0),
      new Point3D(0,0,0),
      new Point3D(0,0,10),
      20, 20);
}

private static IRayTracerProducer getIRayTracerProducer() {
      return new IRayTracerProducer() {

            @Override
            public void produce(Point3D eye, Point3D view, Point3D viewUp,
                  double horizontal, double vertical, int width, int height,
                  long requestNo, IRayTracerResultObserver observer) {

                  System.out.println("Započinjem izračune...");
                  short[] red = new short[width*height];
                  short[] green = new short[width*height];
                  short[] blue = new short[width*height];

                  Point3D zAxis = ...
                  Point3D yAxis = ...
                  Point3D xAxis = ...

                  Point3D screenCorner = ...

                  Scene scene = RayTracerViewer.createPredefinedScene();

                  short[] rgb = new short[3];
                  int offset = 0;
                  for(int y = 0; y < height; y++) {
                        for(int x = 0; x < width; x++) {
                              Point3D screenPoint = ...
                              Ray ray = Ray.fromPoints(eye, screenPoint);

                              tracer(scene, ray, rgb);

                              red[offset] = rgb[0] > 255 ? 255 : rgb[0];
                              green[offset] = rgb[1] > 255 ? 255 : rgb[1];
                              blue[offset] = rgb[2] > 255 ? 255 : rgb[2];

                              offset++;
                        }
                  }

                  System.out.println("Izračuni gotovi...");
                  observer.acceptResult(red, green, blue, requestNo);
                  System.out.println("Dojava gotova...");
            }
      };
}
```

Fill the missing parts! If you do this OK, you will get the following image.



Now if this goes OK, please observe that calculation of color for each pixel is independent from other pixels. Using this knowledge write a main program `hr.fer.zemris.java.raytracer.RayCasterParallel` which parallelizes the calculation using *Fork-Join* framework and `RecursiveAction`.

See help provided in Croatian below.

Once done, copy your solution as new class `hr.fer.zemris.java.raytracer.RayCasterParallel2` and then modify it as shown on the next page. In this class use *`createPredefinedScene2()`* instead of *`createPredefinedScene()`*. The interface `IRayTracerAnimator` represents object which can provide a temporal information to GUI showing the scene. The GUI will first ask this object how often it wants the scene to be redrawn. Then it will call the object to inform it about amount of time elapsed since the last update call, and then ask for information on the current user position, where is user looking-at and where is "up"-direction. Based on the information provided, a new rendering will be scheduled and the result shown. By repeating this procedure periodically, we can create a simple animation of user rotating around the scene and from time to time going up-down. Please be aware that you can not pick redraw interval to be arbitrary small. For example, if you wish to redraw scene each 20 ms, but then take 100 ms to render the scene, you want be able to obtain 1/20ms = 50 frames per second, but only 1/100ms = 10 fps. If your rendering is faster then specified period (e.g. all calculations can be done in 130 ms, and you specify redraw period of 200 ms), GUI will automatically wait for the remaining time to pass before scheduling new rendering request.

```java
public class RayCasterParallel2 {

	public static void main(String[] args) {
		RayTracerViewer.show(
			getIRayTracerProducer(), getIRayTracerAnimator(), 30, 30
		);
	}

	private static IRayTracerAnimator getIRayTracerAnimator() {
		return new IRayTracerAnimator() {
			long time;

			@Override
			public void update(long deltaTime) {
				time += deltaTime;
			}

			@Override
			public Point3D getViewUp() { // fixed in time
				return new Point3D(0,0,10);
			}

			@Override
			public Point3D getView() { // fixed in time
				return new Point3D(-2,0,-0.5);
			}

			@Override
			public long getTargetTimeFrameDuration() {
				return 150; // redraw scene each 150 milliseconds
			}

			@Override
			public Point3D getEye() { // changes in time
				double t = (double)time / 10000 * 2 * Math.PI;
				double t2 = (double)time / 5000 * 2 * Math.PI;
				double x = 50*Math.cos(t);
				double y = 50*Math.sin(t);
				double z = 30*Math.sin(t2);
				return new Point3D(x,y,z);
			}
		};
	}

	private static IRayTracerProducer getIRayTracerProducer() {
		return new IRayTracerProducer() {

			@Override
			public void produce(Point3D eye, Point3D view, Point3D viewUp,
					double horizontal, double vertical, int width,
					int height, long requestNo,
					IRayTracerResultObserver observer,
					AtomicBoolean cancel) {

				// your parallel implementation goes here
				// ...
			}
		};
	}

}
```

Pomoć pri rješavanju ovog zadatka

U nastavku je dan ispis izračunatih vrijednosti za nekoliko slučajeva. U metodi `main` gdje pozivate `RayTracerViewer.show(…)` zadajete očište, gledište te *view-up* vektor; pretpostavka je da su oni zadani kao u uputi. Pri pozivu metode `produce` tada će vrijediti:

```
Parametri koje je dobila metoda
===============================
eye: (10.000000, 0.000000, 0.000000)
view: (0.000000, 0.000000, 0.000000)
viewUp: (0.000000, 0.000000, 10.000000)
width: 500
height: 500
horizontal: 20.0
vertical: 20.0

Izračunato
===============================
X-vektor: (0.000000, 1.000000, -0.000000)
Y-vektor: (0.000000, 0.000000, 1.000000)
Z-vektor: (-1.000000, 0.000000, 0.000000)
Screen-corner: (0.000000, -10.000000, 10.000000)
```

Slijedi ispis zraka za odabrane točke ekrana (uzeo sam x=0, w/3, w/2, 2w/3, w-1 "puta" y=0, h/3, h/2, 2h/3, h-1):

```
Informacije za točku x=0, y=0
Screen-point: (0.000000, -10.000000, 10.000000)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.577350, -0.577350, 0.577350)
RGB =[0,0,0]

Informacije za točku x=166, y=0
Screen-point: (0.000000, -3.346693, 10.000000)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.688102, -0.230287, 0.688102)
RGB =[0,0,0]

Informacije za točku x=250, y=0
Screen-point: (0.000000, 0.020040, 10.000000)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.707106, 0.001417, 0.707106)
RGB =[0,0,0]

Informacije za točku x=333, y=0
Screen-point: (0.000000, 3.346693, 10.000000)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.688102, 0.230287, 0.688102)
RGB =[0,0,0]

Informacije za točku x=499, y=0
Screen-point: (0.000000, 10.000000, 10.000000)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.577350, 0.577350, 0.577350)
RGB =[0,0,0]

Informacije za točku x=0, y=166
Screen-point: (0.000000, -10.000000, 3.346693)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.688102, -0.688102, 0.230287)
RGB =[0,0,0]

Informacije za točku x=166, y=166
Screen-point: (0.000000, -3.346693, 3.346693)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.903874, -0.302499, 0.302499)
RGB =[78,123,123]
```

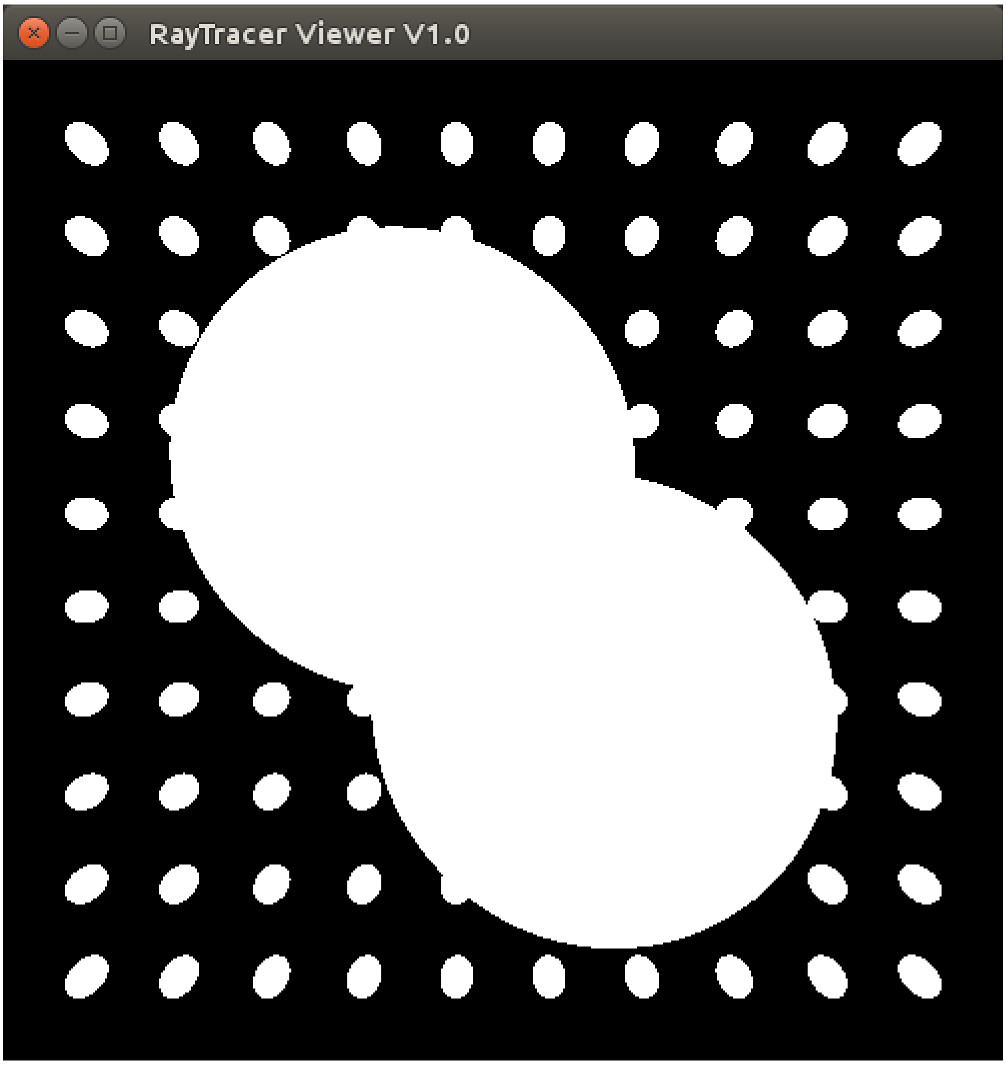
```
Informacije za točku x=250, y=166
Screen-point: (0.000000, 0.020040, 3.346693)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.948301, 0.001900, 0.317367)
RGB =[153,72,57]

Informacije za točku x=333, y=166
Screen-point: (0.000000, 3.346693, 3.346693)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.903874, 0.302499, 0.302499)
RGB =[0,0,0]

Informacije za točku x=499, y=166
Screen-point: (0.000000, 10.000000, 3.346693)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.688102, 0.688102, 0.230287)
RGB =[0,0,0]

Informacije za točku x=0, y=250
Screen-point: (0.000000, -10.000000, -0.020040)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.707106, -0.707106,
-0.001417)
RGB =[0,0,0]

Informacije za točku x=166, y=250
Screen-point: (0.000000, -3.346693, -0.020040)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.948301, -0.317367,
-0.001900)
RGB =[49,57,57]

Informacije za točku x=250, y=250
Screen-point: (0.000000, 0.020040, -0.020040)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.999996, 0.002004, -0.002004)
RGB =[76,33,33]

Informacije za točku x=333, y=250
Screen-point: (0.000000, 3.346693, -0.020040)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.948301, 0.317367, -0.001900)
RGB =[115,69,69]

Informacije za točku x=499, y=250
Screen-point: (0.000000, 10.000000, -0.020040)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.707106, 0.707106, -0.001417)
RGB =[0,0,0]

Informacije za točku x=0, y=333
Screen-point: (0.000000, -10.000000, -3.346693)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.688102, -0.688102,
-0.230287)
RGB =[0,0,0]

Informacije za točku x=166, y=333
Screen-point: (0.000000, -3.346693, -3.346693)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.903874, -0.302499,
-0.302499)
RGB =[0,0,0]

Informacije za točku x=250, y=333
Screen-point: (0.000000, 0.020040, -3.346693)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.948301, 0.001900, -0.317367)
RGB =[62,80,80]
```

```
Informacije za točku x=333, y=333
Screen-point: (0.000000, 3.346693, -3.346693)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.903874, 0.302499, -0.302499)
RGB =[92,61,61]

Informacije za točku x=499, y=333
Screen-point: (0.000000, 10.000000, -3.346693)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.688102, 0.688102, -0.230287)
RGB =[0,0,0]

Informacije za točku x=0, y=499
Screen-point: (0.000000, -10.000000, -10.000000)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.577350, -0.577350,
-0.577350)
RGB =[0,0,0]

Informacije za točku x=166, y=499
Screen-point: (0.000000, -3.346693, -10.000000)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.688102, -0.230287,
-0.688102)
RGB =[0,0,0]

Informacije za točku x=250, y=499
Screen-point: (0.000000, 0.020040, -10.000000)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.707106, 0.001417, -0.707106)
RGB =[0,0,0]

Informacije za točku x=333, y=499
Screen-point: (0.000000, 3.346693, -10.000000)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.688102, 0.230287, -0.688102)
RGB =[0,0,0]

Informacije za točku x=499, y=499
Screen-point: (0.000000, 10.000000, -10.000000)
Ray: start=(10.000000, 0.000000, 0.000000), direction=(-0.577350, 0.577350, -0.577350)
RGB =[0,0,0]
```

Najjednostavnija implementacija metode `tracer` je ona koja provjerava siječe li se zraka s bilo kojih objektom. Ako da, piksel boja bijelo, inače ga ostavlja crno:

```java
protected static void tracer(Scene scene, Ray ray, short[] rgb) {
	rgb[0] = 0;
	rgb[1] = 0;
	rgb[2] = 0;

	RayIntersection closest = findClosestIntersection(scene, ray);

	if(closest==null) {
		return;
	}

	rgb[0] = 255;
	rgb[1] = 255;
	rgb[2] = 255;
}
```

Uz ovakvu implementaciju, dobit ćete sliku kao u nastavku.



Jednom kad Vam to radi, krenite na stvarno bojanje.

Prilikom bojanja, za svaki izvor uzet ćete zraku koja kreće iz izvora i ide prema pronađenom sjecištu. Potom ćete pogledati siječe li se ta zraka s čime. Očekivano je da je odgovor potvrdan: zraka se siječe barem sa objektom prema kojem ste je usmjerili. Ako postoji sjecište s nekim bližim objektom, onda taj zaklanja izvor i izvor se zanemaruje (nema doprinosa promatranoj točki jer ne osvjetljava sjecište). Ovdje trebate paziti samo na jedan sitan implementacijski detalj koji se može pojaviti zbog numeričkih nepreciznosti pri izračunu: kad uspoređujete udaljenosti (tip double), uvijek uzmite u obzir određene tolerancije.

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries for this homework (whether it is yours old code or someones else), except the libraries mentioned in this homework which are available on Ferko. You can use Java Collection Framework and other parts of Java covered by lectures; if unsure – e-mail me. Document your code!

If you need any help, consultations are at standard times (Monday, Tuesday, Wednesday).

All source files must be written using UTF-8 encoding. All classes, methods and fields (public, private or otherwise) must have appropriate javadoc.

There are no mandatory junit tests in this homework.
You are encouraged to write them.

When your **complete** homework is done, pack it in zip archive with name `hw09-0000000000.zip` (replace zeros with your JMBAG). Upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is May 9th 2019. at 07:00 AM.