8. homework assignment; JAVA, Academic year 2014/2015; FER

This homework has two problems.

Problem 1.

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:

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

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:

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

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 clear now 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. 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+i0 because z7 was closer to z_6=-0.9991+i0 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.

So, 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(remin, remax, immin, immax);
    zn = c
    iter = 0;
    iterate {
      zn1 = zn – f(zn)/f'(zn)
      iter++;
    } while(|zn1-zn|>convergenceTreshold && iter<maxIter);
    index = findClosestRootIndex(zn1, rootTreshold);
    if(index==-1) { data[offset++]=0; } else { data[offset++]=index; }
  }
}
```

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

Important: you will need to declate in pom.xml the dependency on `hr.fer.zemris.java.fractals:fractal-viewer:1.0` (the same one that we used during lectures).

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

Details

The Maven coordinates for your solution of this problem are: groupId **`hr.fer.zemris.java.studentVASJMBAG.hw08`**, **artifactId `newton`.**

In order to complete this problem, you are required to write following:

- **immutable** model of complex number denoted `Complex`,
- **immutable** model of root-based complex polynomial denoted `ComplexRootedPolynomial`,
- **immutable** model of coefficient-based complex polynomial denoted `ComplexPolynomial`.

Here are the skeletons for these classes:

```
public static 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() {...}

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

```java
public static class ComplexRootedPolynomial {

	// ...

	// constructor
	public ComplexRootedPolynomial(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
	public int indexOfClosestRootFor(Complex z, double treshold) {...}
}
```

```java
public static 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() {...}
}
```

Given these classes, the core of iterative loop could be written as:

```
Complex numerator = polynomial.apply(zn);
Complex denominator = derived.apply(zn);
Complex fraction = numerator.divide(denominator);
Complex zn1 = zn.sub(fraction);
module = zn1.sub(zn).module();
zn = zn1;
```

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.

```
C:\somepath> java -cp bin 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.

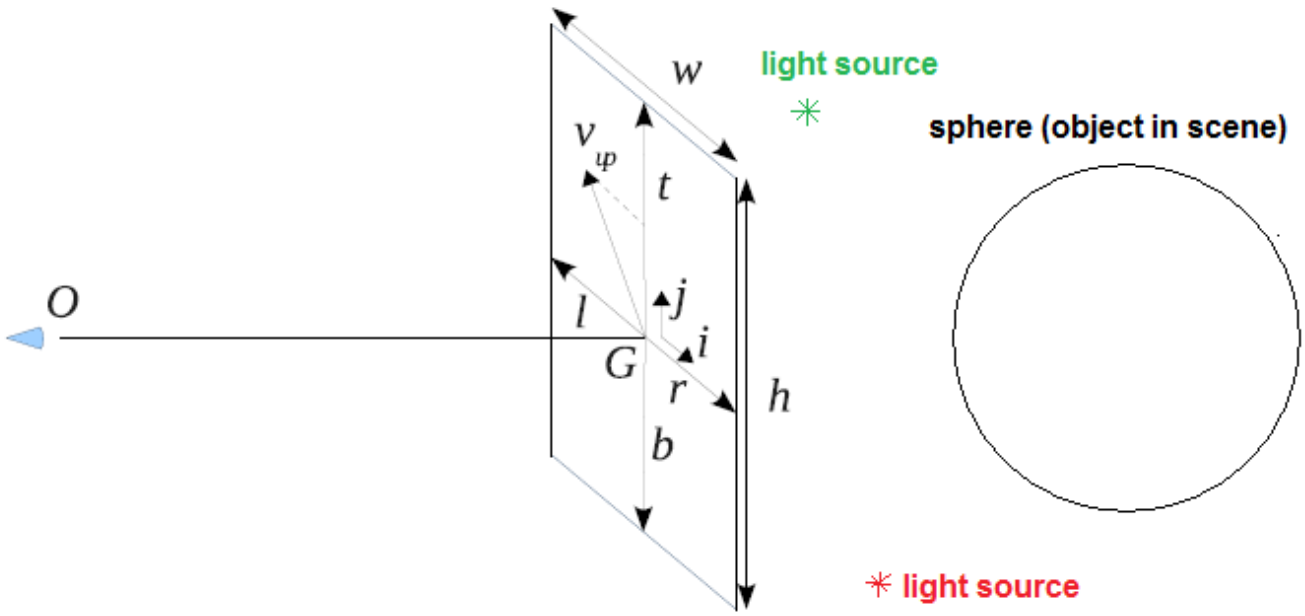
Problem 2.

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. The Maven coordinates for your solution of this problem are: groupId **`hr.fer.zemris.java.studentVASJMBAG.hw08`**, **artifactId `raycaster`.**

I have already prepared a lot of code for you: please declare in **`pom.xml`** dependency on `hr.fer.zemris.java.raytracer:raytracer:1.0`. The sources are also available. To better understand this, you are also advised to download the book available at:

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

(version knjiga-0.1.2014-02-07.pdf) and read section 9.2 (Phong model, pages 225 to 228) and section 10.2 (Ray-casting algorithm, pages 235 to 238). 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.

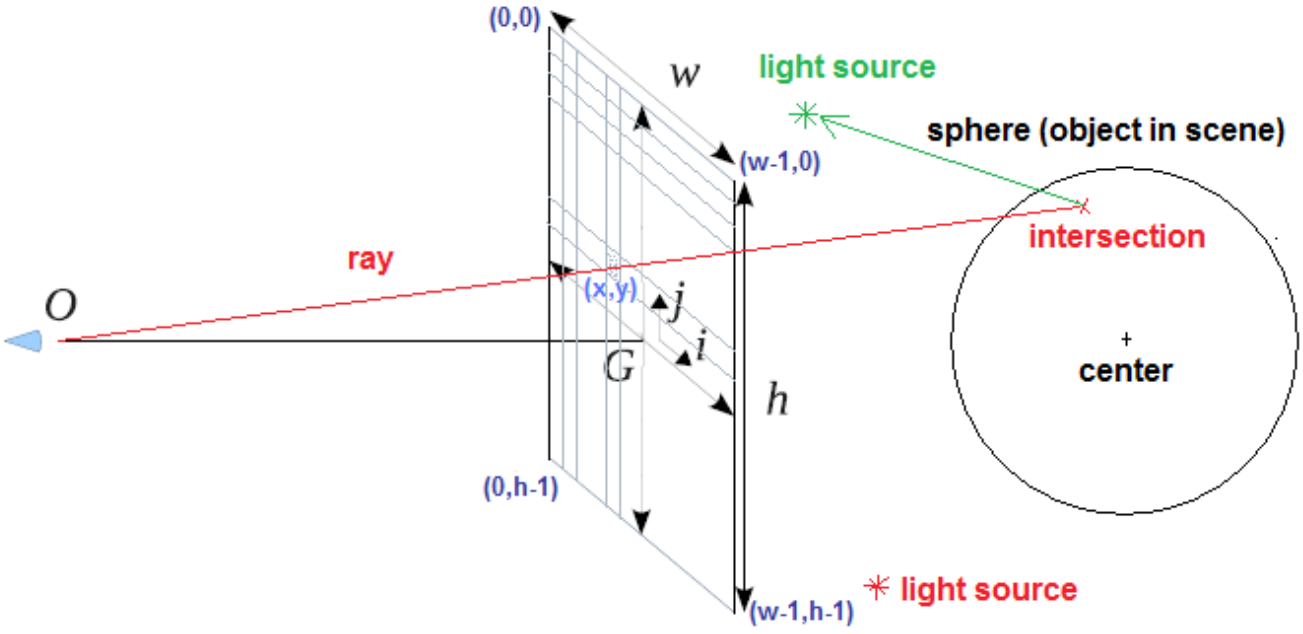
Lets start calculating. Let: $\vec{OG} = \frac{\vec{G}-\vec{O}}{\|\vec{G}-\vec{O}\|}$, i.e. it is the normalized vector from $\vec{O}$ to $\vec{G}$; let $\vec{VUV}$ be normalized version of the view-up vector. Then we can obtain the $\vec{j}'$ vector as follows:

$\vec{j}' = V\vec{U}V - \vec{OG}(\vec{OG} \cdot V\vec{U}V)$ where $\vec{OG} \cdot V\vec{U}V$ 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 archive `hw06-ray01.jar` 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 Phongs 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`. Create package `hr.fer.zemris.java.raytracer.model` in your homework and add class `Sphere`:

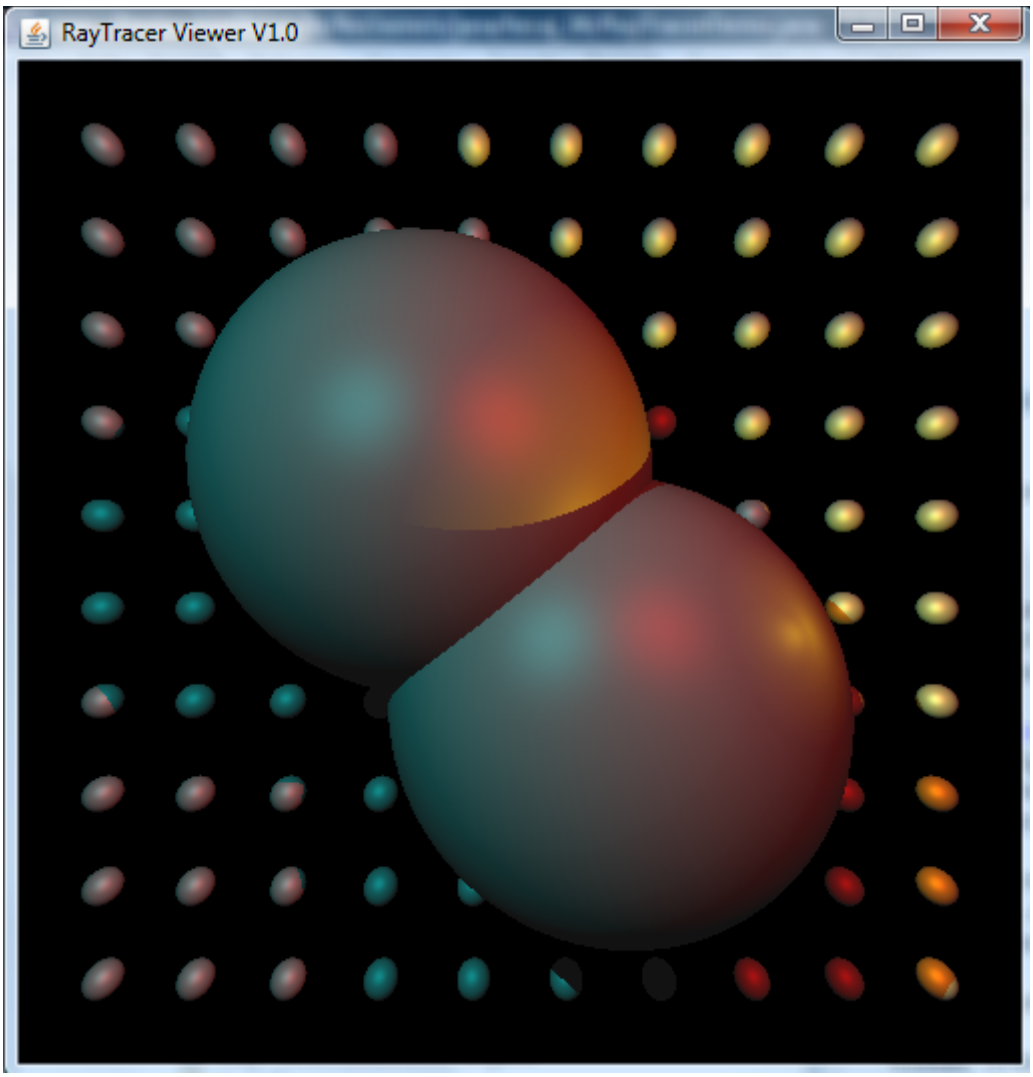
```
package hr.fer.zemris.java.raytracer.model;

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.

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

```
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`.

Important notes

You must create a single ZIP archive containing all projects which you have created as part of this homework (each in its own folder), and than upload this single ZIP. ZIP arhive must have name `HW08-`*yourJMBAG*`.zip`.

All of the classes should have appropriate javadoc.

You are expected to write tests for at least complex numbers and polynomials (from problem 1) and utility methods for finding intersection of ray with spheres.

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 which is not part of Java standard edition (Java SE) unless explicitly allowed or provided by me. You can use Java Collection Framework classes and its derivatives. Document your code!

Upload final ZIP archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is May 9nd 2015. at 07:00 AM.