6. homework assignment; JAVA, Academic year 2011/2012; FER

As usual, please see the last page. I mean it! You are back? OK. Here we have two problems for you to solve.

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:

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

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:

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

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:

$$z_{n+1}=z_n-\frac{f(z_n)}{f'(z_n)}=x_n-\frac{z^4-1}{4z^3} \quad \text{with} \quad x_0=c \quad .$$

We will generate iterations until we reach a predefined number of iterations (for example 16) or until $|z_{n+1}-z_n|$ 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+*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.

So, we will proceed just as with Mandelbrot fractal:

```
for(y in y_min to y_max) {
  for(x in x_min to x_max) {
    c = map_to_complex_plain(re_min, re_max, im_min, im_max);
    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 must download the newest version of dodatak06.jar (from 2012-04-29).

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

Details

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:

```java
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 brojnik = polynomial.apply(zn);
Complex nazivnik = derived.apply(zn);
Complex razlomak = brojnik.divide(nazivnik);
Complex zn1 = zn.sub(razlomak);
module = zn1.sub(zn).module();
zn = zn1;
```

Write a main program `hr.fer.zemris.java.hw06.part1.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.hw06.part1.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 diveded 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.

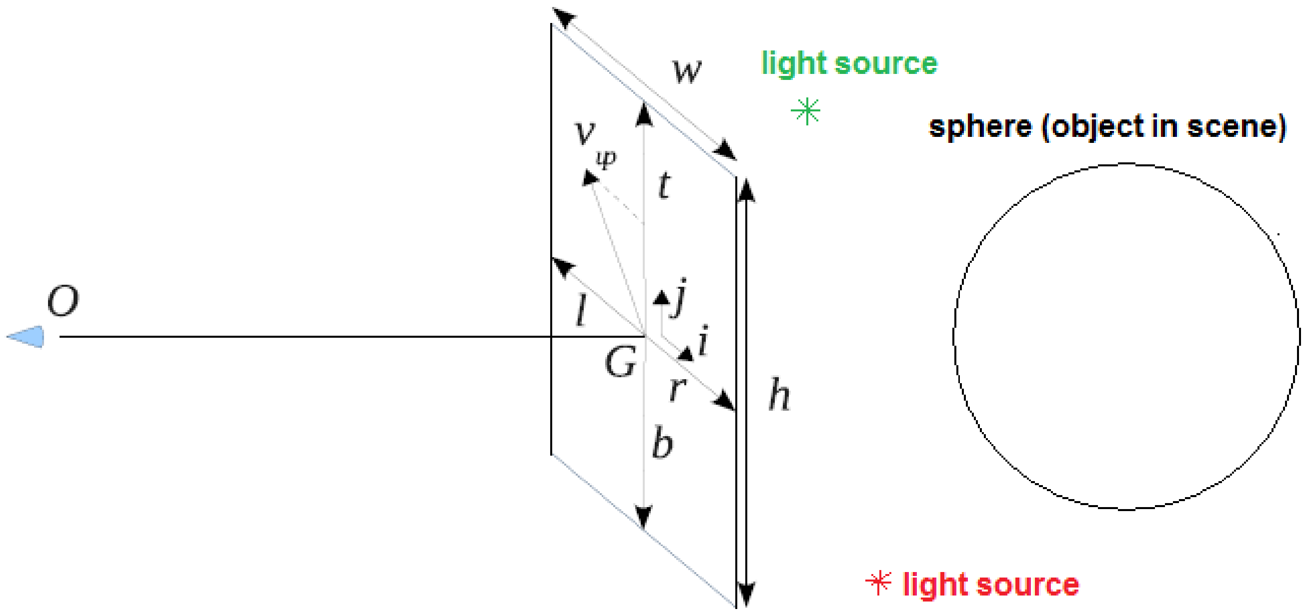
Problem 2.

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

I have already prepared a lot of code for you: please download `hw06-ray01.jar` and `hw06-ray02.jar` from repository in Ferko. To better understand this, you are also advised to download:

http://java.zemris.fer.hr/nastava/irg/knjiga-0.1.2011-02-16.pdf

and read section 8.2 (Phong model, pages 179 to 181) and section 9.2 (Ray-casting algorithm). To render an image using ray-casting algorithm, you start by defining which object are present in 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 eye position (O) and view point (G). In that plane you will create a 2D coordinate system, so you will have x-axis (as indicated by vector *i* on image) and y-axis (as indicated by vector *j* on image). If you only start with eye-position and view point, your y-axis can be arbitrarily placed in this plane. To help us fix y-axis, it is customary to specify another vector: view-up vector that does not have to lay in plane but it must not be co-linear with G-O vector. If this holds, that exists projection of view-up vector into plane: normalized version of this projection will become our vector *j* and hence determine the orientation of y-axis.

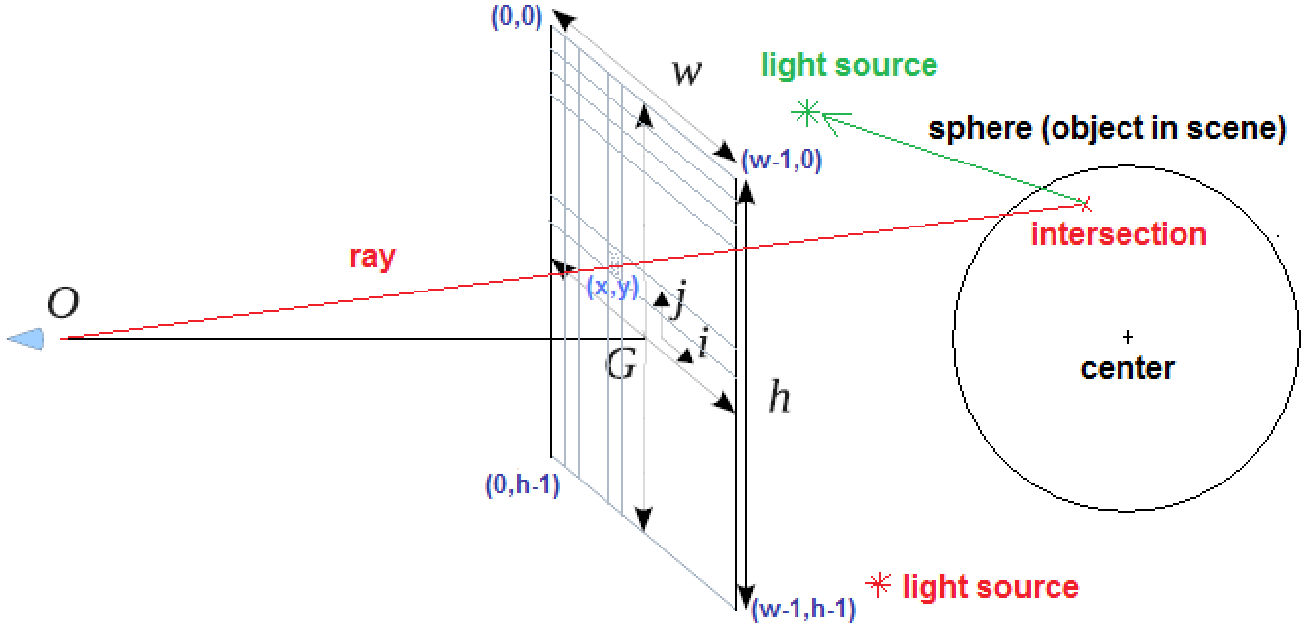
Lets start calculating. Let: $\vec{OG}=\frac{\vec{G}-\vec{O}}{\|\vec{G}-\vec{O}\|}$, i.e. it is normalized vector from $\vec{O}$ to $\vec{G}$; let $\vec{VUV}$ be normalized version of view-up vector. Then we can obtain vector $\vec{j}'$ as follows:

$\vec{j}'=\vec{VUV}-\vec{OG}(\vec{OG}\cdot\vec{VUV})$ where $\vec{OG}\cdot\vec{VUV}$ is scalar product. Define its normalized version to be: $\vec{j}=\frac{\vec{j}'}{\|\vec{j}'\|}$. Now we can calculate vector $\vec{i}'$ that will determine orientation of x-axis as cross product: $\vec{i}'=\vec{OG}\times\vec{j}$ and its normalized version $\vec{i}=\frac{\vec{i}'}{\|\vec{i}'\|}$.

Once we determined where the plane is and what are the vectors determining our x-axis (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. We will asume it to be 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*.

You have class `Point3D` at your disposal 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 on next picture.



We will define (0,0) to be upper left point of our rectangular part of plane; x-axis will be oriented just as vector $\vec{i}$ is, and y-axis will be oriented opposite from vector $\vec{j}$. We can obtain 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 position of pixel (x,y) in our 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 its simple: we define a ray of light which starts at $\vec{O}$ and passes through $\vec{point}_{xy}$ .We calculate 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 be visible for our pixel (x,y). If no intersection is found, pixel will be rendered black (r=g=b=0). However, if an intersection is found, we must determine what color to use. If multiple intersections are found, we must chose the closest one to eye-position since that is what we will see. For coloring we will use Phongs model which assumes that there is one or more light sources present in scene. In our example there are two light sources. Each light source is specified with intensities of r, g and b components it radiates.

Here is the pseudo code:

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 scene (in front of observer)
  if no *S*, color (*x*,*y*) with rgb(0,0,0) else use rbg(determineColorFor(*S*))

The procedure determineColorFor(*S*) is given by 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 if 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.tecaj_06.rays` in your homework and add class `Sphere`:

```
package hr.fer.zemris.java.tecaj_06.rays;

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) {
		...
	}
}
```

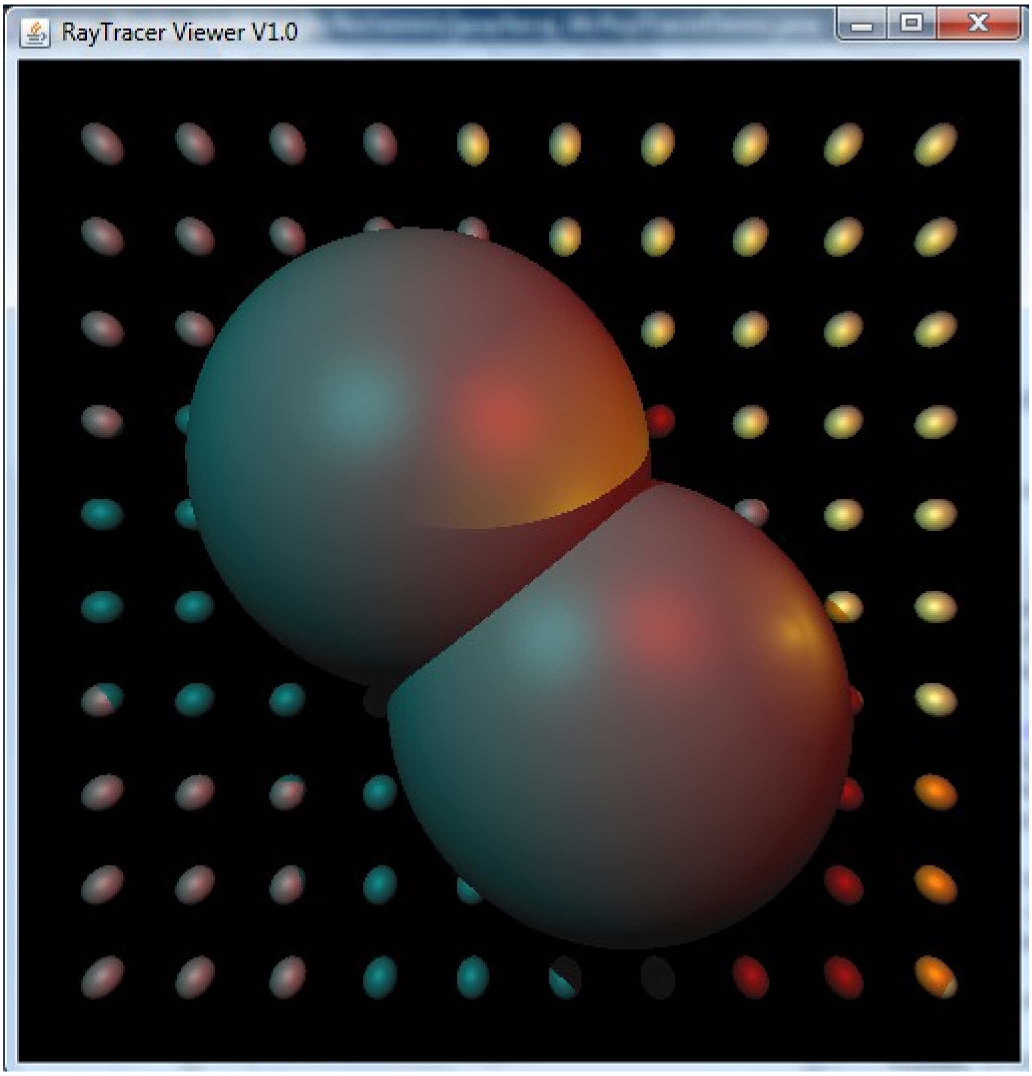
and implement all that is missing. Until you do that, eclipse will report errors that RayTracerViewer references a class `Sphere` that does not exists. Coeficients *kd** determine object parameters for diffuse component and *kr** for reflective components.

Write a main program `hr.fer.zemris.java.hw06.part2.RayCaster`. The basic structure of it 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];
```

```java
				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! If you do this OK, you will get 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.hw06.part2.RayCasterParallel` which parallelizes the calculation using Fork-Join framework and `RecursiveAction`.

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). Document your code!

In order to solve this homework, create a blank Eclipse Java Project and write your code inside. Once you are done, export project as a ZIP archive and upload this archive on Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted.