# Ray tracing (graphics)

In [3D computer graphics](https://en.wikipedia.org/wiki/3D_computer_graphics), **RayTracing** is a technique for modeling [light transport](https://en.wikipedia.org/wiki/Light_transport_theory) for use in a wide variety of [rendering](https://en.wikipedia.org/wiki/Rendering_(computer_graphics)) algorithms for generating [digital images](https://en.wikipedia.org/wiki/Digital_image).

I try to wright my simple RayTracing, which can image simple objects with some effects like defusing, specular, reflection, refraction and so on.

I will use Java for this project.

## **Step 1 Project structure**

### **Geometry**

The main primitive that we will use is Vec3D.

Simple structure consist of three Double values.

Package Geometry consists of primitives like Ray, Sphere etc.

All of them extends GeometryObject class.

At the same time, this package includes GeometryMath class.

This static class processes all of the above primitives.

### **Graphics**

This package includes all element of graphical objects.

The main class is GraphicsObject that includes GeometryObject and Material.

At the same time, this package includes LightObject.

### **Rendering**

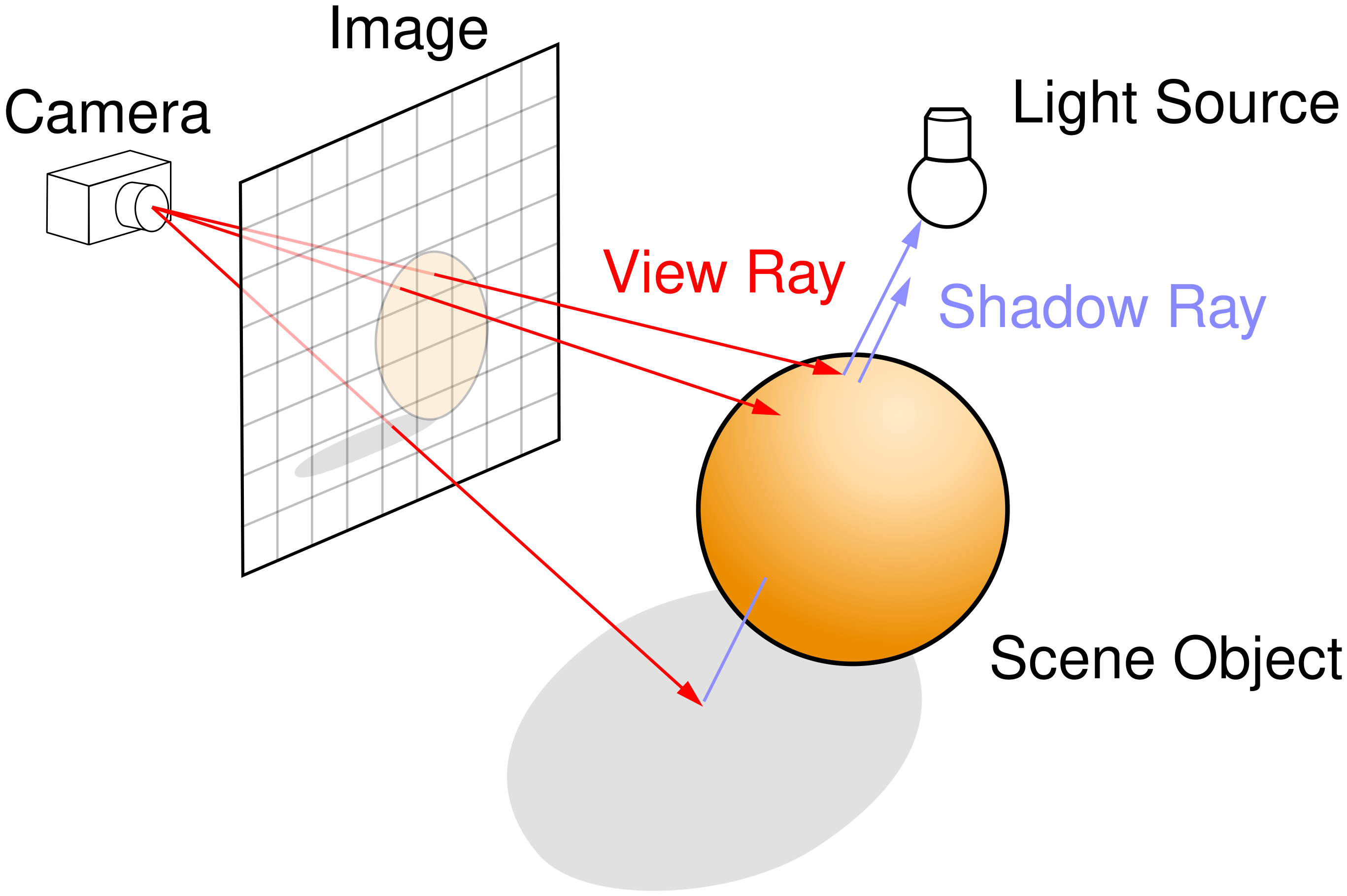
Using that package we rendering our scene. It include Camera, Scene and ScreenPanel that displays our images or cinematics.

### **Structures**

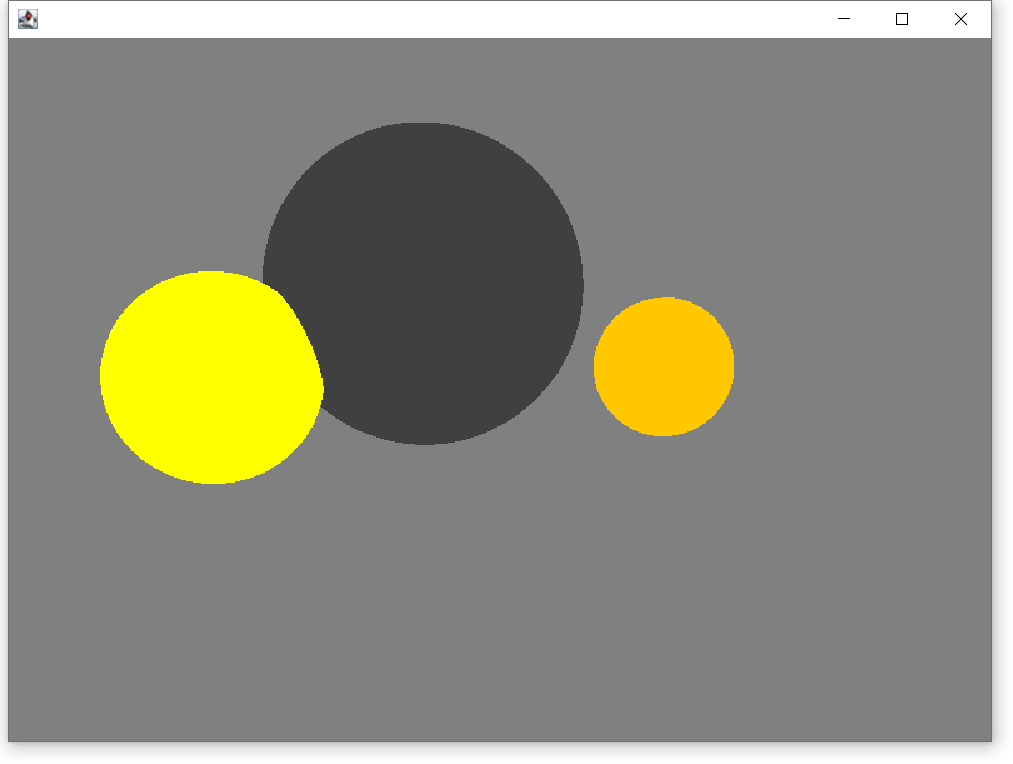
Includes some auxiliary structures like Pair.

## **Step 2 Simple RayTracing**

Well the main idea of RayTracing is to cast rays in 3D. It looks like RayTracing, but in 3D.



By using the rayIntersect in GeometryMath we can cast our rays in simple level and can get some simple image like this:

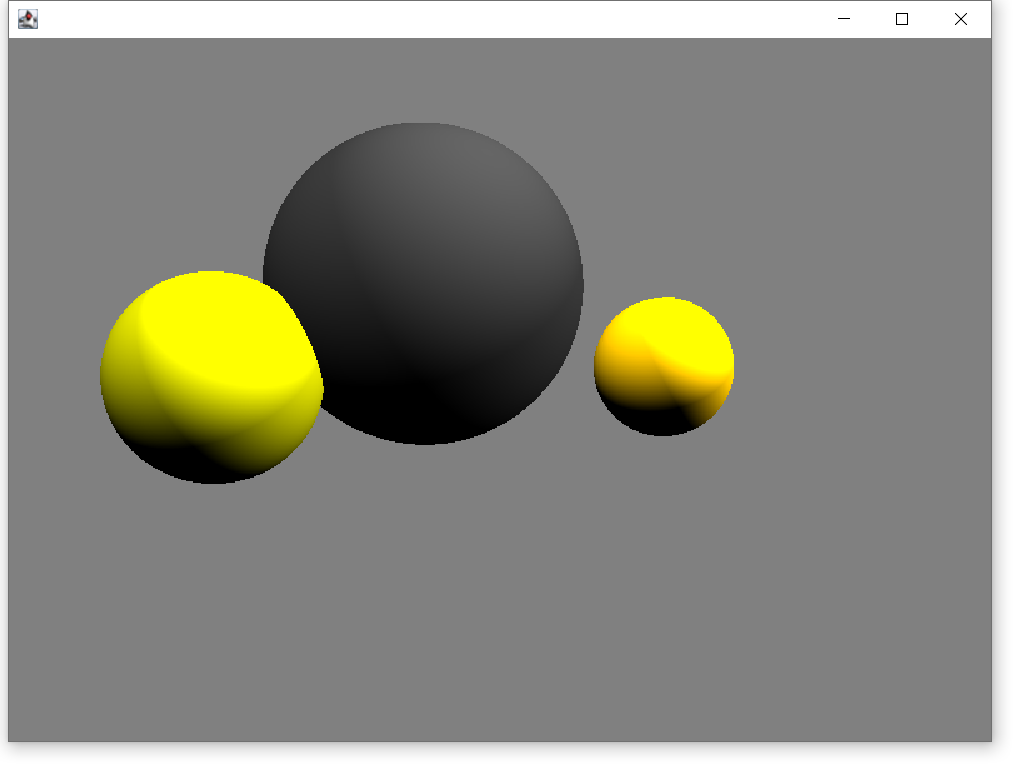


## **Step 3 Defuse**

Well, the first problem solved. Let us add some light.

To count real lighting is a very, very difficult task, so, like everyone, we will deceive the eye by drawing completely non-physical, but as much as possible plausible results. First note: why is it cold in winter and hot in summer? Because the heating of the ground depends on the angle of sunlight. The higher the sun above the horizon, the brighter the surface is. Conversely, the lower the horizon, the weaker the surface. Well, after the sun sets behind the horizon, photons don’t even reach us. In relation to our spheres: here is our beam emitted from the camera (no relation to photons, notice!) crossed with the sphere. How do we understand how the intersection is illuminated? You can just look at the angle between the normal vector at that point and the vector describing the direction of light. The smaller the angle, the better the illuminated surface. To make it even more convenient, you can simply take the scalar production between the normal vector and the light vector. Recall that the scalar production between two vectors a and b equals the product of the norms of the vectors on the cosine angle between the vectors: a\*b = |a| b| cos(alpha(a,b)). If we take unit length vectors, the simplest inner product gives us the intensity of surface illumination.

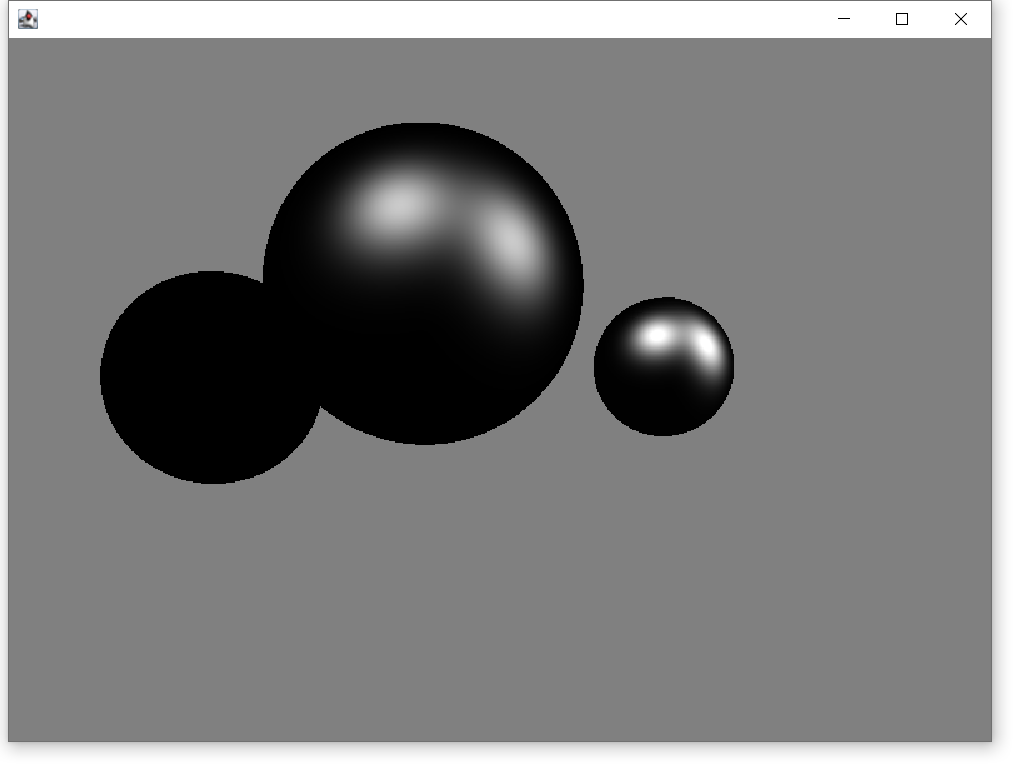
**Step 4 Specular**



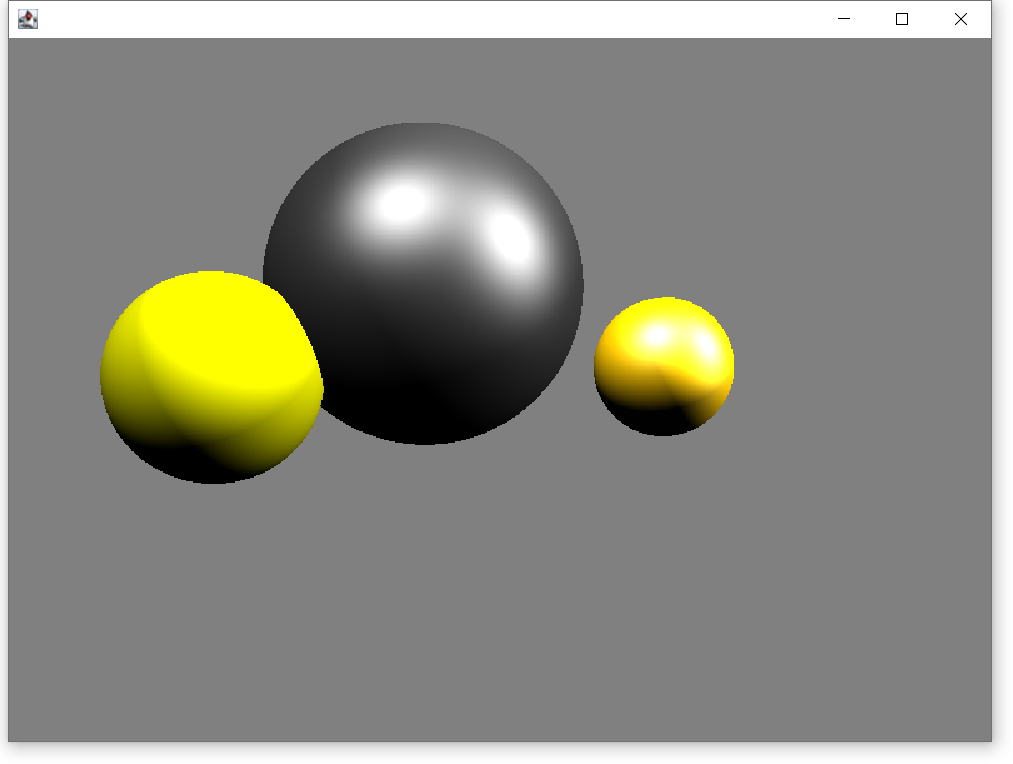
Nice, matte objects display well, but what about glossy ones.

The less the angle between the direction of sight and the direction of reflected light. In addition, the corners, of course, we will count through the scalar, just like before.

Using this idea, we can give this resolute:



Now let us group Simple RayTracing, Defuse and Specular into one image and we can give this resolute: 



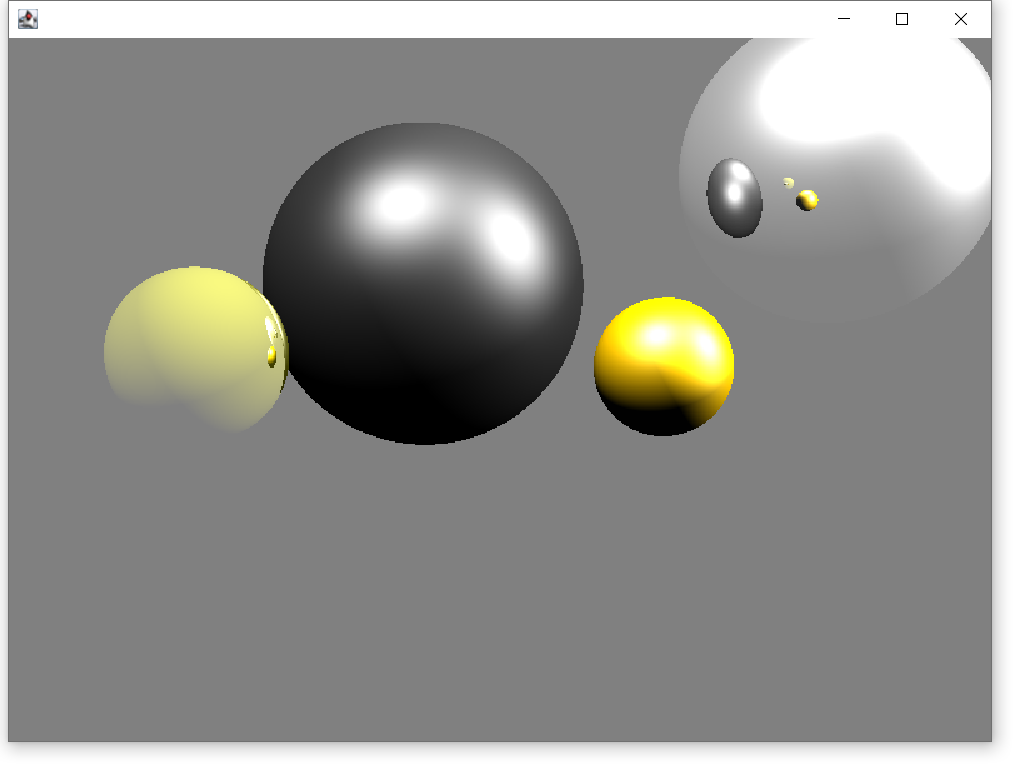
## **Step 5 Reflection**

Huu we did it.

Now let add reflection, because it is simple.

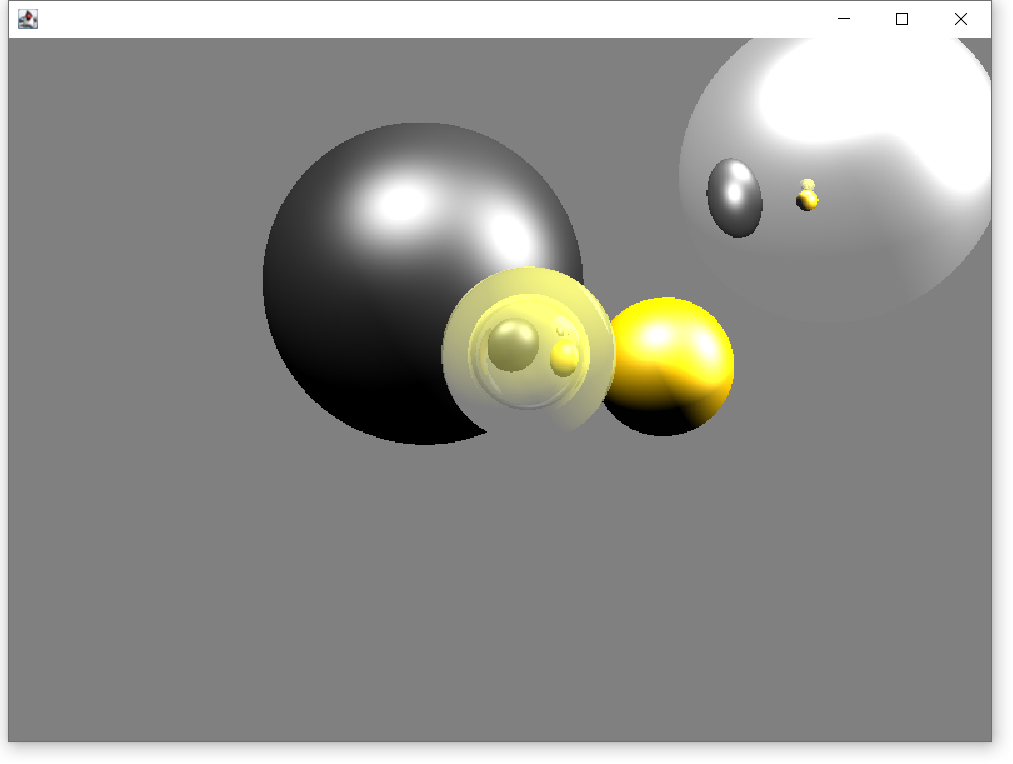
Everything that you need is to cast reflection ray and all.

Adding a simple pair line of code we can get this beautiful resolute:



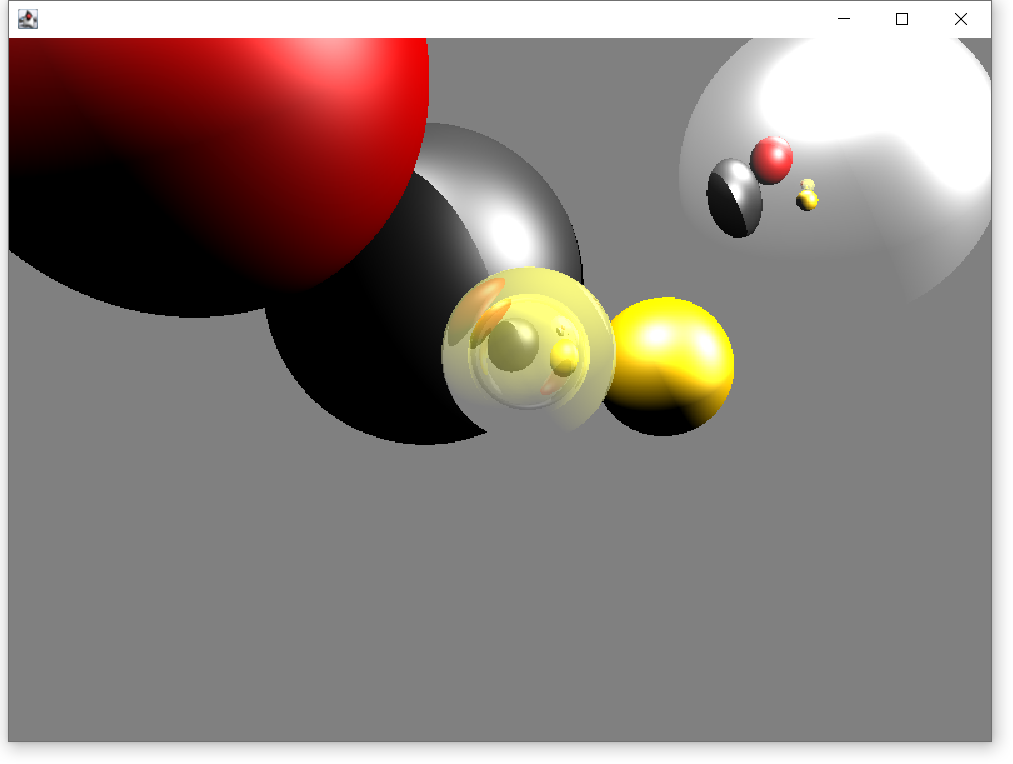
## **Step 6 Refraction**

Using the method, that we have used to display reflection, we can display refraction, by the Snail’s formula. In addition, using them we have new beautiful resolute:



## **Step 7 Shadow**

The idea of shadows is simple. Everything that we need to do, is to check that our light ray intersect any other objects. In addition, the resolute:



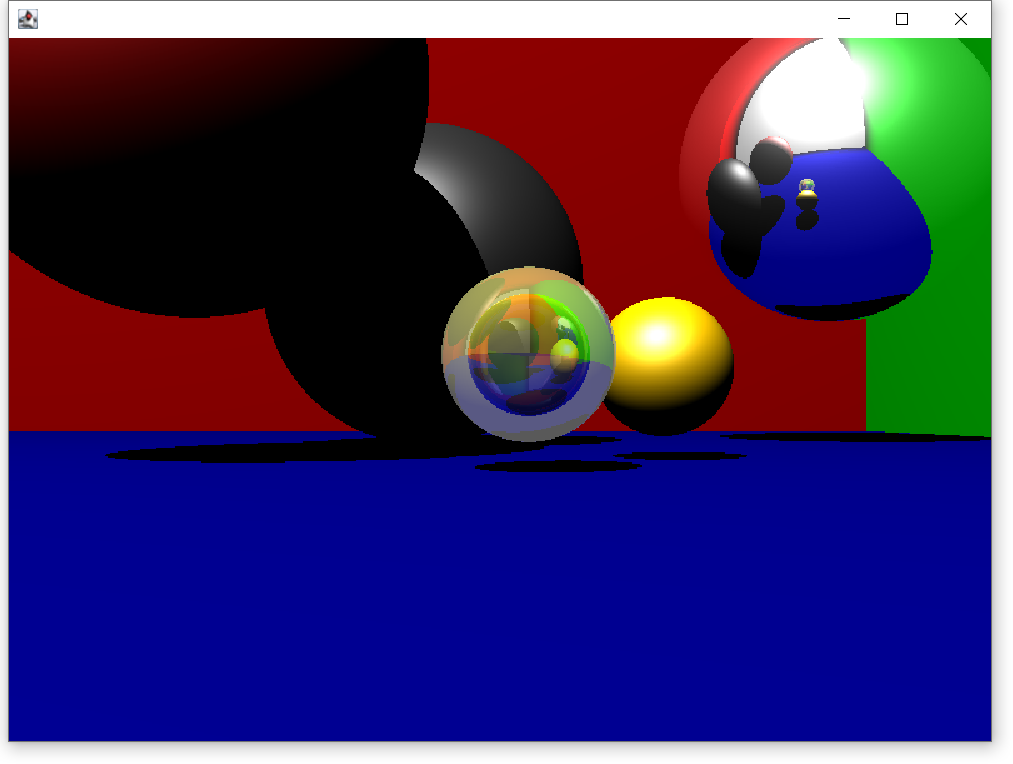
## **Step 8 Plane**

Well we did Simple RayTracing, and know let us add some new primitives.

Firstly, add Plane.

It is easy using linear algebra.

The resolute:



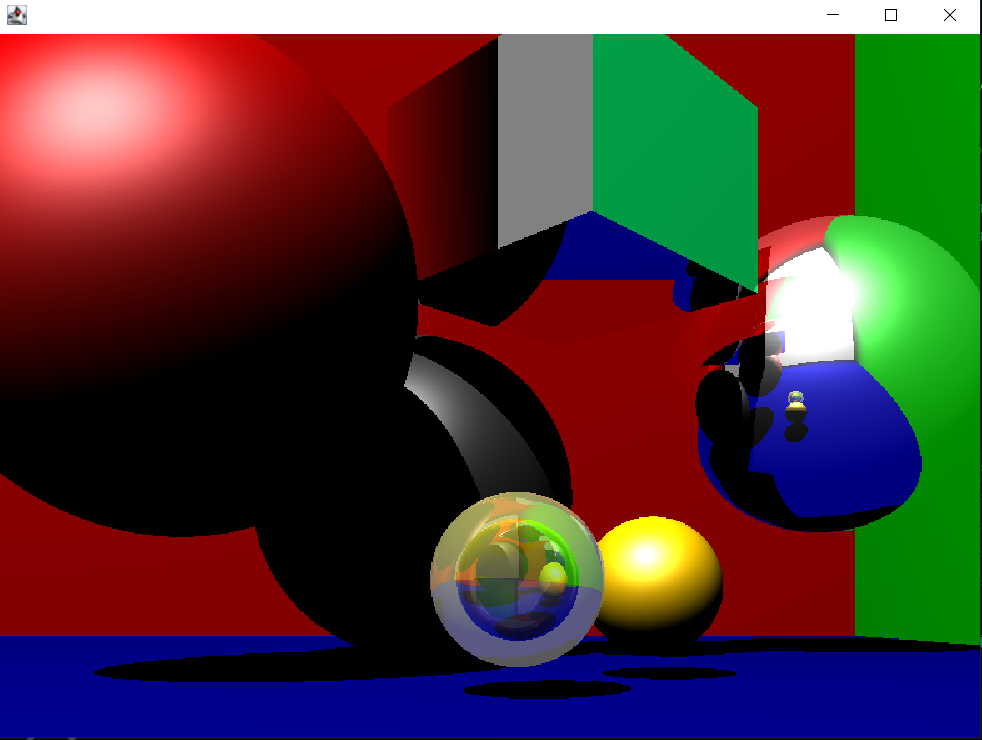
**Step 9 Polygons**

Now let us add polygon-using plane as base.

## Step 10 Models

Now when we have Polygons, it is easy to add Models.

Moreover, the resolute:



Conclusion:

I have wanted to implement this project for a long time. Finally decided. I did not seek to "fit the code on a business card" or write a full-fledged graphics engine. My goal was to show in practice the work of the Retrace technology and its capabilities. I have not implemented some of the functionality I wanted, like recording short cinematics or displaying textures, because now, I do not see the expediency in this. Perhaps if there is a large number of people who want to see these functions, I will develop this project. You can also look at my GitHub; there is my old library of games in the organizations, which has a game based on RayCasting technology. This technology, although useless now, is very interesting to study.

# Literature:

[Internet] / https://habr.com/ru/post/436790/

https://en.wikipedia.org/wiki/Ray\_tracing\_(graphics)

https://web.archive.org/web/20190520020303/http://www.few.vu.nl/~kielmann/theses/avdploeg.pdf