Exercise 2 – Ray Tracing (Part 1)

Submission: Sunday, May 4th 2014

In this exercise you will implement a basic ray tracer. A ray tracer shoots rays from the observer's eye (the camera) through a screen and into a scene which contains one or more surfaces. it calculates the rays intersection with the surfaces, finds the nearest intersection and calculates the color of the surface according to its material and lighting conditions.

You are required to implement a ray tracer with the following features. Please read the complete document carefully before you start coding!

For part 1, you only need to have basic rendering features: diffuse and specular shading, without reflection, transparency or shadows. These features are grayed out in this document but you will need to implement them for part 2 of the exercise. Some examples are given of scenes without the advanced features so you can make sure your scenes look exactly the same.

Surfaces

- **Spheres**. Each sphere is defined by the position of its center and its radius.

- **Infinite planes**. Each plane is defined by its normal N and an offset c along the normal. A point on the P plane will satisfy the formula $P \cdot N = c$.

- **Ellipsoids**. Each ellipsoid is defined by the position of its center, and a matrix which defines the transformation needed to transform the ellipsoid back into a sphere.

* Those are the three simplest surfaces to compute intersections with. If you want, you can later add the option to render other objects such as: boxes, triangles, rectangles, cylinders etc. However it is not required.

Materials

To compute the color of a surface, you have to take into considerations the object's material. If the material is reflective, you need to shoot reflection rays to find objects which would be reflected. If the material is transparent or semi-transparent, you need to shoot rays through the surface to find objects that are behind it.

Each rendered surface (sphere or plane) is associated with a material in the scene file. Several surfaces can be associated with the same material.

Each material have the following attributes:

- **Diffuse color (RGB)**. This is the "regular" color of a surface. This value is multiplied by the light received by the object to find the base color of the object.

- **Specular color (RGB)**. Specularity is the reflection of a light source. The specular color of a surface defines the intensity and color of that reflection. Materials in real life often have different specular color than diffuse color. For example, a polished wooden desk would have a brown diffuse color but white specular color.

- **Phong specularity coefficient (floating point number)**. We will use the Phong shading model. This coefficient controls the type of specularity of a surface. A high value (around 100) renders small and sharp specular reflections, for shiny surfaces such as metal, and a low value (around 1 or so) renders wide and soft specular reflections, for materials such as clay or stone.

- **Reflection color (RGB)**. Reflections on the surface are multiplied by this value. For example a red metal pipe might reflect its surrounding but the reflection will have a red tint.

- **Transparency (floating point number)**. This value will be 0 when the material is opaque (not transparent at all) and 1 when the material is completely transparent. Note that a surface can be completely transparent and still show reflections of the surfaces which surround it (the reflections are independent of the object's transparency).

Lights

To light the scene we will use point lights. Point lights are located at a certain point and shoot light equally in all directions around that point. You will also have to implement soft shadows, as explained later.

Each light will have the following attributes:

- **Position (XYZ)**. A vector specifying the position of the point light.

- **Color (RGB)**. The color of the light. This color multiplies the surface diffuse and specular colors when the surface is lit by this light.

- **Specular intensity (floating point number)**. Sometimes we would like a light to produce only diffuse lighting, or only partial specular lighting. If this number is 1, the light intensity will be the same for the diffuse component and the specular component computation. Otherwise, the light color will be multiplied by the specular intensity for the specular component computation.

- **Shadow intensity (floating point number)**. Usually when lighting a scene there will be a key light which is casting a strong shadow, and some other lights (like fill light or rim light) which does not cast shadows. Sometimes we would also like a light to partially go through surfaces and light their background as well, to reduce the necessary number of lights to light a scene.

The light received by a surface which is hidden from the light is multiplied by (1-shadow intensity).

In other words, If this number is 1, the light casts full shadows so no light will be received by the obscured surface (from this particular light source). If this number is 0, the light casts no shadows at all (the lighting is the same whether the surface is obscured from the light or not). For values in the range between 0 and 1, the light will cast a partial shadow. Note that in this case the shadow intensity is not accumulated; it doesn't matter how many objects are in the way between the light source and the surface, it is either obscured with partial shadow or it is fully lit.

- **Light Radius (floating point number)**. This value is used to compute soft shadows, see description below.

Camera and General Settings

The camera is defined by the following attributes. There will be only one camera in each scene file.

- **Position (XYZ)**. A vector specifying the position of the camera.

- **Look-at point (XYZ)**. A vector specifying the point the camera is looking at. When defining a scene, it is simpler to work with look-at points than specifying a direction. In your code you have to compute the direction of the camera from the look-at point and the camera position.

- **Up vector (XYZ).** The up vector of the camera defines the direction the camera is looking up at. For convenience, this vector is not necessary perpendicular to the direction vector of the camera. In your code you have to fix the up vector so it is perpendicular to the direction vector you computed from the look-at point - this will require some thinking ;)

- **Screen distance (floating point number)**. The distance of the camera's screen from the camera. This value defines the focal point of the camera, and controls the direction of the ray that will go through each pixel.

- **Screen width (floating point number)**. The width of the camera's screen. Together with the screen distance, this value controls the viewing angle of the camera. When the screen is wider or the distance is shorter, the camera's viewing angle is larger. The screen height is computed using the requested aspect ratio of the image and this number. A screen distance roughly equal to the diagonal of the screen size produces a "normal" camera lens, similar to the camera in your phone.

There are also some general settings that are used to describe a scene.

- **Background color (RGB)**. This color is used when a ray does not hit any surface. Note that reflection rays and transparency rays also use this background color when they do not hit another surface.

- **Number of shadow rays (small integer, 1-10)**. This number controls the amount of rays which are used for every light to compute soft shadows. Note that shadow rays are emitted from a squared area N*N, so if this number is for example 5 there will be 25 rays emitted for each shadow calculation.

- **Maximum recursion level (integer, usually around 10)**. Since reflective surfaces can reflect other reflective surfaces and so on, each time the reflection color is computed there is a recursion. This number limits the recursion level, so after this amount of recursion the returned color would be the background color (as if the tenth reflected surface does not exist).

Some Tips

This section will give a general overview of ray tracing as a reminder to what you saw in class. The general process of ray tracing is shooting a ray through each pixel in the image. For each pixel, you should discover the location of the pixel on the camera's screen (using camera parameters), and construct a ray from the camera through that pixel.

Next you check the intersection of the ray with all surfaces in the scene (you can add optimizations such as BSP trees if you wish but they are not mandatory), and find the nearest intersection of the ray. This is the surface that will be seen in the image.

To compute the color of the surface, you go over each light in the scene and add the value it induces on the surface. To find out whether the light hits the surface or not, you shoot rays from the light towards the surface and find whether the ray intersects any other surfaces before the required surface - if so, the surface is occluded from the light and the light does not affect it (or partially affects it because of the shadow intensity parameter). To produce soft shadows, as explained below, you will shoot several rays from the proximity of the light to the surface and find out how many of them hit the required surface.

For part 1, you only need to check that the normal of the surface is facing the light. Make sure you are not adding "negative light" to the object when it is facing the other way.

A short reminder about additive colors: The output color of the surface is the **sum** of its diffuse color, its specular color and its reflective color.

If a surface is transparent, objects behind the surface should be ray traced to discover the color (by shooting the same ray from the camera and checking what is the next surface the ray hits). The following formula will then be used:

```
output color = (background color) * transparency
             + (diffuse + specular) * (1 - transparency)
             + (reflection color)
```

In other words, the reflection color is added **after** the transparency of the object (otherwise we could not have strong reflections on transparent objects which is very common in real life - think of the a soap bubble)

Some other tips

- Write a vector class to handle all the necessary vector math. You will need to perform a lot of dot products, cross products, vector additions and multiplications, and even matrix multiplication.
- Use cross products to generate vectors which are perpendicular to a plane. To find a plane that is perpendicular to a vector, find a perpendicular vector (this should be easy!) and then find the third vector by using a cross product. You can also use the cross products to fix the up vector of the camera.
- Rendering ellipsoids: To render an ellipsoid you need to transform the space such that the ellipsoid is transformed into a sphere. You can then compute the intersection between the transformed ray and a sphere. Note that if the ray hits the sphere you need to transform the hit position you found (or distance from the camera) back into the original space.

Computing Soft Shadows

There are several methods to compute soft shadows. We will implement one method which produces nice results but is somewhat computationally heavy. The submitted solution should follow the method here.

In general, to generate shadows on a surface we will send a ray from the light source to the surface and check whether the ray hits other objects prior to that surface. This will produce a hard shadow, as every pixel would either be completely lit or completely shadowed. In reality lights do not emanate from points, because the light source has some area to it.

To generate soft shadows, we will send several shadow rays from the light source to a point on the surface. The light intensity that hits the surface from this light source will be multiplied by the number of rays that hit the surface divided by the total number of rays we sent. For example, if we send 25 rays from the light source and 5 of them hit the surface at the given point, the surface will 20% illuminated at that point. If the number of shadow rays parameter is 1 only one ray will be cast and the shadows will be hard.

The sent rays should simulate a light which has a certain area. Each light is defined with a light radius (see light parameters). To simulate a light with a radius we use the following process:

1. Find a plane which is perpendicular to the ray.

2. Define a rectangle on that plane, centered at the light source and as wide as the defined light radius.

3. Divide the rectangle into a grid of N*N cells, where N is the number of shadow rays from the scene parameters.

4. If we shoot a ray from the center of each cell, we will get banding artifacts (you can test it and see for yourself). To avoid banding we select a random point in each cell (by uniformly sampling x value and y value using rnd.nextDouble() function), and shoot the ray from the selected random point to the light.

5. Aggregate the values of all rays that were cast and count how many of them hit the required point on the surface.

Note that this process is done for every point on the surface that we look at (or every camera ray, reflection ray or transparency ray that we shoot).

Implementation Details

The ray tracer should be implemented in Java. To define a scene to render, the input for the ray tracer will be a scene file which is a simple text file that defines every surface, material and light source in the scene, as well as some camera settings and general settings for the render.

You will be provided with some skeleton files to parse the scene file and save an image to the file system. The code should run in command line (**you need to upload a .jar file**) and accept 4 parameters. For example:

```
java RayTracing.RayTracer scenes\Spheres.txt scenes\Spheres.png 500 500
```

The first parameter is the scene file, and the second is the name of the image file to write. Those two are mandatory. The next two parameters are optional and define the image width and height, respectively. Please set the default size to 500x500.

There is no need for optimizations (such as BSP trees etc). Running time is not very important, however if your code takes more than a minute or two to run on any of the scene files you probably have a bug somewhere in the code.

Scene Definition Format

The scenes are defined in text scene files with the following format.
Every line in the file defines a single object in the scene, and starts with a 3 letter code that identifies the object type. After the 3 letter code a list of numeric parameters is given. The parameters can be delimited by any number of white space characters, and are parsed according to **the specific order in which they appear**. Empty lines are discarded, and so are lines which begin with the character "#" which are used for remarks. The provided skeleton files parse the file and provide the list of parameters for each object as an array of strings params[]. You need to parse the

strings in the correct order using functions such as Double.parseDouble() or Integer.parseInt().

The possible objects with their code and list of required parameters are given below. Note that for part 1 you can ignore the grayed parameters but they would still be in the file (so the positions of other parameters do not change):

```
"cam" = camera settings (there will be only one per scene file)
      params[0,1,2] = position (x, y, z) of the camera
      params[3,4,5] = look-at position (x, y, z) of the camera
      params[6,7,8] = up vector (x, y, z) of the camera
      params[9] = screen distance from camera
      params[10] = screen width from camera

"set" = general settings for the scene (once per scene file)
      params[0,1,2] = background color (r, g, b)
      params[3] = root number of shadow rays (N^2 rays will be shot)
      params[4] = maximum number of recursions

"mtl" = defines a new material
      params[0,1,2] = diffuse color (r, g, b)
      params[3,4,5] = specular color (r, g, b)
      params[6,7,8] = reflection color (r, g, b)
      params[9] = phong specularity coefficient (shininess)
      params[10] = transparency value between 0 and 1

"sph" = defines a new sphere
      params[0,1,2] = position of the sphere center (x, y, z)
      params[3] = radius
      params[4] = material index (integer). each defined material gets an
                  automatic material index starting from 1, 2 and so on

"elp" = defines a new ellipsoid
      params[0,1,2] = position of the ellipsoid center (x, y, z)
      params[3,4,5] = first row of transformation matrix
      params[6,7,8] = second row of transformation matrix
      params[9,10,11] = third row of transformation matrix
      params[12] = material index

"pln" = defines a new plane
      params[0,1,2] = normal (x, y, z)
      params[3] = offset
      params[4] = material index

"lgt" = defines a new light
      params[0,1,2] = position of the light (x, y, z)
      params[3,4,5] = light color (r, g, b)
      params[6] = specular intensity
      params[7] = shadow intensity
      params[8] = light width / radius (used for soft shadows)
```

Several scene files will be uploaded along with the images they should produce. Please also perform tests on other scene files that you write yourself. Your code will be tested on several other scene files as well to make sure it works correctly for all cases. Note the following scenes in particular:

- Scenes which are named "Part1_..." have only the basic features and your code should produce the same output as the reference images.
- Notice how the camera parameters affect the rendering in the scenes titled "ManySpheres_Narrow", "ManySpheres_Wide", etc.
- The scenes named "SpheresCompare" and "EllipsoidsCompare" should look exactly the same - if they don't, you probably have a bug the code that renders ellipsoids.

Bonus Creativity Points

After you finish implementing all the features in part 2, you can add extra features.

We want you to have some fun and play around with the endless possibilities of rendering a scene. You can get up to 10 bonus points by implementing something creative. In other words, you can get 100 without implementing it. Implementing it will give you a maximum score of 110 which can compensate for some mistakes in your code if there are any. It should be quite easy and fun to implement once you have all the rest.

The creative rendering should be submitted in a **separate** executable file, with a short readme file that explains how to run it if necessary (if you require extra parameters etc), and what is the expected result. It should be able to run on a regular scene file.

Examples of creative rendering:

- Using the position or normal where the ray hits the surface to enhance the rendered color.
- Adding noise to the geometry, for example render spheres with a randomly varying radius.
- Use non-linear rays, for example using parabolas instead.
- Anything else you can think of!

The creative rendering will be graded according to the amount of extra work (if the change is very small you will not get a lot of extra points) and how interesting the results are.

Submission

The submission deadline for part 1 is Sunday, May 4th 2014.

Submit this exercise in pairs.
Please submit the executable jar file and the source in a zip file named
<id1>_<id2>.zip. If you choose to implement something creative, upload a second zip file named <id1>_<id2>_creative.zip, which includes the executable jar file and the source code of your creative rendering, and a readme file if necessary.

Upload the files to the exercise page in moodle.