Computer Graphics Tick 1

Introduction to Ray Tracing

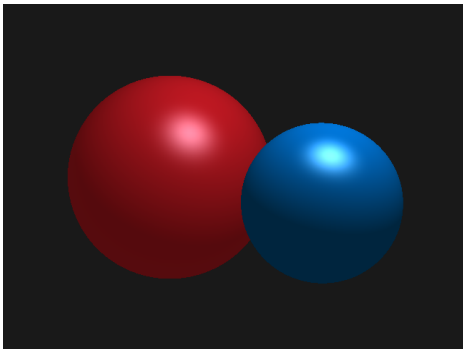


Figure 1: The image you will create in this exercise.

1 Introduction

In this exercise you will write code for a simple ray tracer. Ray tracing is a method for rendering a 2D image from a 3D scene.

In real life, light sources (e.g. the sun) cast rays of light that bounce off 3D objects and into our eyes. These light rays eventually enter our eyeballs and form a 2D image on our retina. Ray tracing models this phenomenon efficiently by tracing light rays *backwards* through a 3D scene. Each pixel in the 2D image plane casts a ray into the scene through a camera. If a ray intersects with an object in the scene, we colour its pixel according to the object's colour and the scene illumination.

2 Getting started

Download archive *tick1.zip* from Moodle area "Tick 1". After extracting you will see the following files:

```
tick1
├─ src/gfx/tick1
│  ├─ Camera.java
│  ├─ PointLight.java
│  ├─ Ray.java
│  ├─ RaycastHit.java
│  ├─ Renderer.java
│  ├─ Scene.java
│  ├─ SceneLoader.java
│  ├─ Sphere.java
│  ├─ Tick1.java
│  └─ Vector3.java
└─ basic_scene.xml
```

The files you will modify and submit for this exercise have been highlighted. Note that much of the code has already been written for you. You should not modify the other files, though please look to see how they work.

Note that the package names used in this and other ticks do not adhere to the Java specification; normally package names should contain the full URL of the institution, for example uk.ac.cam.cl.gfxintro.crsid. We use shorter package names to simplify the code for the automatic tester.

The main method for this exercise is in Tick1.java. It takes two arguments:

`-i, --input` : the XML scene file to render.

`-o, --output` : the image file to write out.

Compile the source code, and run it with the following arguments:

```
--input basic_scene.xml --output output.png
```

As our 3D scene is currently empty, and our ray tracer un-implemented, this will output a blank image output.png.

2.1 Compiling and running the code

The instruction below is for compiling and running the code form the command line. Refer to the document *Working with IDEs* for an instruction for an IDE.

To compile, change the current directory to tick1, then run:

```
javac -d ./out src/gfx/tick3/*.java
```

The -d option specifies where to put the compiled classes. It is a good practice not to mix the sources with compiled code. You will need to create this directory before compiling: mkdir out.

You can start the program with the following command:

```
java -classpath ./out gfx.tick3.Tick3 --input basic_scene.xml --
    output output.png
```

Vector3

You have been provided with a Vector3 class that you will use in this exercise. It has three fields x, y, and z that represent the components of a vector in 3D space. Vector3 also contains many useful methods that let you add, subtract, and multiply vectors together. We also represent colour as a Vector3, with x, y, z corresponding to r, g, b pixel values. You should make sure you are familiar with this class before continuing.

3 Building a scene

A scene is a description of the virtual world that we want to render. Throughout the graphics practicals you will use an XML-based scene format. For this first exercise, scene files can contain the following elements:

`sphere` – 3D spheres specified by a position, radius, and colour.

`ambient-light` – a light colour that provides background illumination.

`point-light` – a light source that has a position, colour, and intensity.

Parameters are specified by setting attributes for each element. For example, colour is set with `colour="#FF0000"` using hex colour codes.

You can preview scene files by dragging and dropping them into the web-based previewer: `http://www.cl.cam.ac.uk/teaching/1617/Graphics/previewer/previewer.html`. You will need a modern browser. This is a handy way to check your scene files have been specified correctly. Note that both the ray-tracer and the previewer use the left-handed coordinate system, as shown in Figure 2.

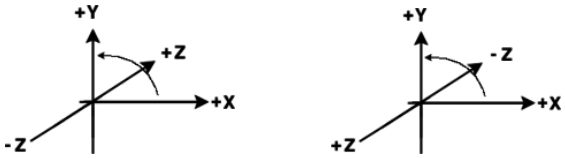


Figure 2: Left: left-handed coordinate system. Right: right-handed coordinate system. The ray-tracer you work on in this tick uses left-handed coordinate system, in which objects further away from the camera have larger z coordinate values. Note that right-handed coordinate system is used OpenGL, which you will work with in Tick 3.

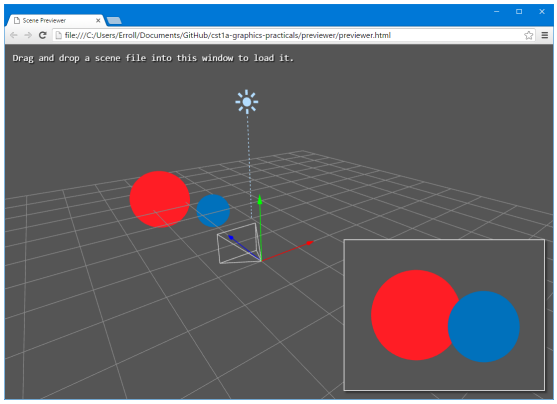


Figure 3: You can drag and drop scene files into the online scene previewer. The camera is the white wireframe pyramid positioned at the origin. The bottom right panel shows what the camera will render.

Example scene

Here is a basic example scene:

```
<scene>
    <ambient-light colour="#333333"/>
    <point-light x="3" y="3" z="3" colour="#FFFFFF" intensity
        ="100"/>
    <sphere x="0" y="0" z="6" radius="1" colour="#FF0000"/>
</scene>
```

This scene contains a dark grey ambient light, a white point light at $(3, 3, 3)$ with intensity 100, and a unit-radius red sphere at $(0, 0, 6)$.

Define a new scene

Your task is to build a scene according to our specification. Create a new scene in the `tick1.xml` file to include the following:

- An ambient light with colour `#555555`.
- A point light source at $(1, 3, 2)$ with colour `#B3DDFF` and intensity 120.
- A sphere at $(0.55, -0.16, 3.5)$ with radius 0.5 and colour `#0071BC`.
- A sphere at $(-0.55, 0, 5)$ with radius 0.9 and colour `#FF1D25`.

Use the online scene previewer to check that your scene is correct. It should look like the scene shown in Figure 3.

4 Intersecting with spheres

Ray tracers cast rays through each image pixel. The formation and casting of these rays has been written for you in the Camera and Renderer classes. The camera is positioned at the origin, pointing along the positive z-axis.

At this stage, the renderer is outputting blank images with colour BACKGROUND_COLOUR as it does not include code for intersecting rays with spheres. Your task is to modify the Sphere.intersectionWith() method to handle these intersections correctly.

Ray-sphere intersection formula

Given:

- A ray defined as $P(s) = O + sD$, where O is a 3×1 vector with the origin, D is the direction, and $s \geq 0$ is some non-negative (scalar) distance travelled by the ray.
- A sphere $(P - C) \cdot (P - C) - r^2 = 0$ with centre C and radius r. The "$\cdot$" symbol is a dot product of two vectors. Note that a dot-product of a vector with itself gives a squared length of that vector: $(P-C)\cdot(P-C) = |P-C|^2$.

We can find where (and if) the ray intersects the sphere by plugging the ray equation into the sphere equation, and solving the resulting quadratic equation for ray parameter s.

$$((O + sD) - C) \cdot ((O + sD) - C) - r^2 = 0 \tag{1}$$

$$((O - C) + sD) \cdot ((O - C) + sD) - r^2 = 0 \tag{2}$$

$$s^2 \underbrace{(D \cdot D)}_{a} + s \underbrace{(2D \cdot (O - C))}_{b} + \underbrace{(O - C) \cdot (O - C) - r^2}_{c} = 0 \tag{3}$$

Where a, b, and c are quadratic equation constants. Take care: if the discriminant is negative, solutions to s will be imaginary, signifying no intersection. In general you will get two solutions for s – make sure you use the closest one to the ray origin. Consult the lecture notes on how to do this if you get stuck.

Ray-sphere intersection code

The code you are given in the Sphere class is as follows:

```
public RaycastHit intersectionWith(Ray ray) {

    Vector3 O = ray.getOrigin();
    Vector3 D = ray.getDirection();
    Vector3 C = position;
    double r = radius;

    // Determine quadratic equation constants
    double a = D.dot(D);
    double b = 2 * D.dot(O.subtract(C));
    double c = (O.subtract(C)).dot(O.subtract(C))
               - Math.pow(r,2);

    TODO: Determine if ray and sphere intersect

    TODO: Work out point of intersection

    TODO: Return a RaycastHit that includes the object, ray
    distance, point, and normal vector

    // Currently returning empty RaycastHit for no intersection
    return new RaycastHit();
}
```

You should modify the above code to implement the following algorithm:

1. Calculate the discriminant as $\sqrt{b^2 - 4ac}$.
2. If the discriminant < 0, return an empty RaycastHit for no intersection.
3. Compute the two solutions for s, s_1 and s_2 using the quadratic formula.
4. If $s_1 > 0$ and $s_1 < s_2$, return an RaycastHit at distance s_1.
5. Else, if $s_2 > 0$ and $s_2 < s_1$, return an RaycastHit at distance s_2.
6. Finally, if neither s_1 or s_2 are positive (the sphere is behind the ray), return an empty RaycastHit for no intersection.

To indicate that there is no intersection, simply return an empty RaycastHit:

```
return new RaycastHit();
```

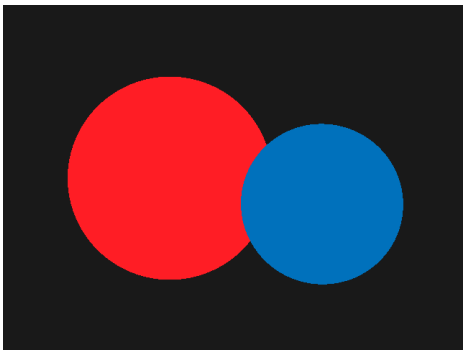


Figure 4: The scene once ray-sphere intersection has been implemented.

However, once you have found an intersection, return a RaycastHit that references the object hit (this sphere), the distance the ray travelled, the point hit, and the normal of the object at that point:

```
Vector3 point = ray.evaluateAt(s);
Vector3 normal = this.getNormalAt(point);
return new RaycastHit(this, s, point, normal);
```

Now, compile and run your program with the scene you have built previously. Figure 4 shows the desired output. Next, we will implement an illumination model to shade the spheres.

5 Shading the spheres

At this point, your ray tracer is rendering the spheres without any illumination, colouring the pixels according to the sphere's colour only.

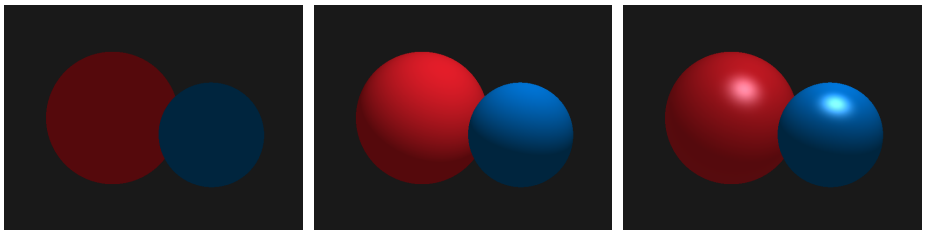
You will now implement the Phong illumination model in the Renderer.illuminate() method to correctly shade them with the ambient and point light source.

The Phong illumination model

The Phong illumination model shades a point on an object's surface with three lighting components which are added together:

$$\mathcal{P}(c) = \underbrace{C_{diff}(c)\, I_a(c)}_{\text{Ambient}} + \underbrace{C_{diff}(c) k_d\, I(c) \max(0, N \cdot L)}_{\text{Diffuse}} + \underbrace{C_{spec}(c) k_s\, I(c) \max(0, R \cdot V)^n}_{\text{Specular}} \tag{4}$$

$\mathcal{P}(c)$ is the value of the colour channel c (red, green or blue) for the pixel. The other terms are described below.



(a) Ambient illumination (b) +Diffuse lighting (c) +Specular highlights

Figure 5: The scene shaded with the Phong illumination model. (a) shows the background ambient illumination, (b) adds the Lambertian term to include diffuse illumination, and (c) finally adds specular highlights.

Ambient lighting represents background illumination that is omnipresent in the scene. C_{diff} is the surface's diffuse colour (set by the scene file), and I_a the scene's ambient lighting.

Diffuse lighting is based on the observation that the amount of energy from a light source falling on a surface depends on the angle between the surface and the light. k_d is the surface's diffuse coefficient, I the illumination from the point light at the surface (that decreases over distance), N the surface normal, and L the direction to the point light from the surface.

Specular lighting adds shiny highlights. The idea is that highlights are strongest when V and L are symmetrically positioned across the surface normal. R is the reflection of L in N, V the direction to the camera, and n the specular exponent that controls how shiny the surface is.

Figure 5 shows the contribution of each illumination component in turn. Note that N, L, R, and V should all be unit vectors. k_d and k_s have been set to constants in this exercise.

Phong illumination code

The code you are given in the Renderer class is as follows:

```
private Vector3 illuminate(Scene scene, Sphere object, Vector3 P,
    Vector3 N) {

    PointLight light = scene.getPointLight();
    double distanceToLight = light.getPosition().subtract(P).
        magnitude();

    Vector3 I_a = scene.getAmbientLighting();
    Vector3 I = light.getIlluminationAt(distanceToLight);

    Vector3 C_diff = object.getColour();     // Diffuse colour
    Vector3 C_spec = new Vector3(1);         // Specular colour

    // Object's Phong coefficients
    double k_d = object.getPhong_kD();
    double k_s = object.getPhong_kS();
    double n = object.getPhong_n();

    TODO: Calculate L, V, and R

    TODO: Calculate NdotL and RdotV

    TODO: Calculate Vector3 ambient, diffuse, and specular terms

    TODO: return ambient+diffuse+specular

    return object.getColour();
}
```

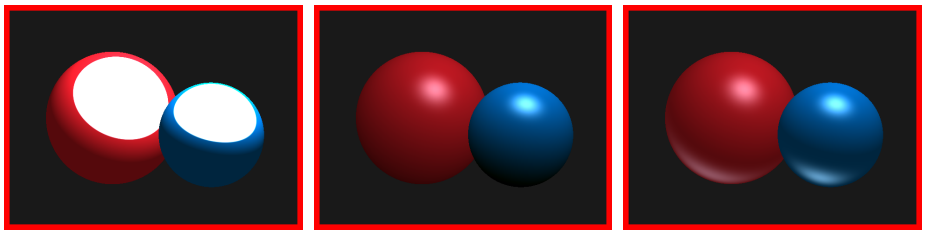
Your task is to modify the code to return the Phong illumination at point P rather than just the object's colour. Note that many of the parameters have already been calculated for you.

Colours are represented as Vector3, with the three components corresponding the red, green, and blue components of the colour.

You should implement the following:

1. Calculate L, V, and R. Remember that the camera is positioned at the origin. You can use Vector3.reflectIn() to reflect vectors.[1]
2. Calculate the $N \cdot L$ factor for diffuse, and $R \cdot V$ for specular.

[1] Note: Vector3.reflectIn() computes the *mirror-like* reflection of a vector, not the *bounce-like* reflection off an object off a surface.



(a) Make sure vectors L and V are normalized.

(b) Make sure $N \cdot L$ is non-negative: `max(0, NdotL)`.

(c) Make sure $R \cdot V$ is non-negative: `max(0, RdotV)`.

Figure 6: Some mistakes you might make: (a) is too bright because direction vectors are not normalized. In (b) the bottom of the blue sphere is too dark because you are not using `max(0, NdotL)`. (c) has incorrect secondary highlights from not using `max(0, RdotV)`.

3. Calculate the three lighting terms L_{Amb}, L_{Diff}, and L_{Spec}. Take care that each is represented by a Vector3.
4. Finally return their sum $L_{\text{Amb}} + L_{\text{Diff}} + L_{\text{Spec}}$.

Now your renderer should produce the same image as Figure 1. Figure 6 shows some common mistakes you might make, and explains how to avoid them.

Next time, you'll extend your ray tracer to handle planes, multiple light sources, reflection, and shadows.

6 Submission

Once you're happy with your tick's output, go to Moodle → *Tick 1* → *Tick 1 Submission*, switch to *Submission* tab, drag and drop two files: Renderer.java and Sphere.java and click *Submit*. There is no need to put anything in the *Comments* field. Then, you can switch to the tab *Submission view* and hit *Evaluate*.

If your code generates correct results, you should see the message *"Congratulations! Your code passed the tester."*. Note that this is a provisional mark and you may still need to have and interview to get credit for your tick.

If your code does not compile or fails to produce correct results, you will see an error message or a report from running the tests. Your code will be tested on the same scene as the one you put in (tick1.xml) and one addition test scene that is kept hidden. The tester will award a full mark only when correct images are produced for both scenes.

Note that you can use *Edit* tab to make small changes to your code and run evaluation again. But do not use this option for debugging or completing a larger portion of work. Note that you cannot see images generated by your program when you click *Evaluate* link.

There is a small chance that your program generates correct results but the tester reports it fails the tests. If you have checked your code thoroughly and suspect that this could be the problem with the tester, please let us know on *Help forum for Graphics 1a* in Moodle or by e-mail `rkm38@cam.ac.uk`. Please do not post your code on the open forum. If your code works correctly, you will be awarded a full mark in the ticking session.