Computer Graphics Tick 2

Shadows and Reflections

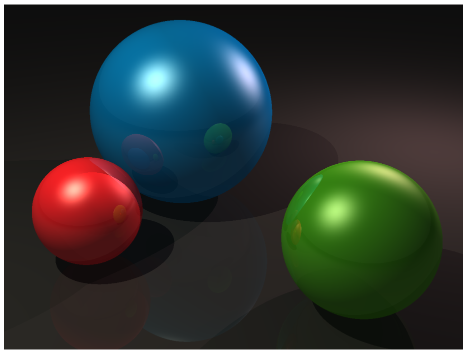


Figure 1: The image you will create in this exercise.

1 Introduction

In this exercise you will extend the ray tracer to handle a new shape and additional effects. The new shape you will render is a plane with which you can represent the ground. The additional effects you will implement are multiple light sources, hard shadows, and reflective surfaces.

2 Getting started

Download archive *tick2.zip* from Moodle area "Tick 2". After extracting you will see the following files:

```
tick2
├─ gfx/tick2
│  ├─ Camera.java
│  ├─ Plane.java
│  ├─ PointLight.java
│  ├─ Ray.java
│  ├─ RaycastHit.java
│  ├─ Renderer.java
│  ├─ Scene.java
│  ├─ SceneLoader.java
│  ├─ SceneObject.java
│  ├─ Sphere.java
│  ├─ Tick2.java
│  └─ Vector3.java
└─ tick2.xml
```

The files you will modify and submit for this exercise have been highlighted. You will modify Plane.java to implement ray-plane intersection, and you will modify Renderer.java to shade with multiple lights, shadows, and reflections.

Tick2.java contains the main method. Along with the `--input` and `--output` arguments from the last tick, it can also take a `--bounces` argument that specifies the number of times rays can bounce when calculating reflections.

3 Intersecting with planes

We often want to render planes in graphics as they can simulate the ground or walls in a scene. The Tick 2 scene file `tick2.xml` now can contain plane elements specified like so:

```
<plane x="0.0" y="0" z="2" nx="1" ny="0" nz="0" colour="#222222"/>
```

Where the plane is specified by a point (x, y, z) and normal vector (n_x, n_y, n_z).

Your task is to implement Plane.intersectionWith() so we can render planes as well as spheres.

Ray-plane intersection formula

Given:

- A ray defined as $P(s) = O + sD$.
- A plane defined as $(P - Q) \cdot N = 0$, where N is the normal to the plane, Q is a point on the plane and "$\cdot$" is a dot product.

We can find where the ray intersects the plane by plugging the ray equation into the plane equation, and solving for resulting ray parameter s.

$$(O + sD - Q) \cdot N = 0$$
$$s = \frac{(Q - O) \cdot N}{D \cdot N}$$

If $s < 0$, the plane is behind the ray origin so there is no intersection.

Ray-plane intersection code

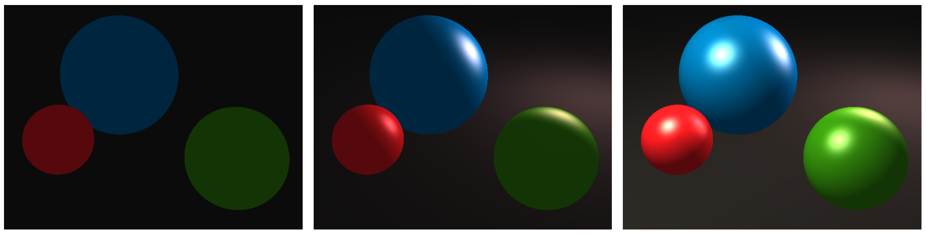
The current ray-intersection code in Plane.java looks like this:

```
public RaycastHit intersectionWith(Ray ray) {
    Vector3 O = ray.getOrigin();
    Vector3 D = ray.getDirection();
    Vector3 Q = this.point;
    Vector3 N = this.normal;

    TODO: Calculate ray parameter s at intersection

    TODO: If s < 0, return empty RaycastHit, otherwise return
    RaycastHit describing point of intersection

    return new RaycastHit();
}
```

Your task is to modify this code to calculate ray parameter s. If s is not negative, return a RaycastHit corresponding to the point of intersection:

```
    return new RaycastHit(this, s, ray.evaluate(s), N);
```

Your renderer should now be able to render planes.



Ambient light only | One point light | Two point lights

Figure 2: The effect of multiple lights: here is the Tick 2 scene shaded with zero, one, and two point light sources (from left to right).

4 Multiple light sources

In the previous exercise, scenes contained only a single point light. In graphics, we want to simulate multiple light sources. Figure 2 shows the Tick 2 scene shaded with varying numbers of lights (without shadows for now).

To handle multiple lights we modify the equation for calculating pixel's colour to sum over a set of n lights:

$$\mathcal{P} = \underbrace{C_{\mathit{diff}}\, I_a}_{\text{Ambient}} + \sum_{i=0}^{n} \underbrace{C_{\mathit{diff}}\, k_d\, I_i \max(0, N \cdot L_i)}_{\text{Diffuse}} + \sum_{i=0}^{n} \underbrace{C_{\mathit{spec}} k_s\, I_i \max(0, R_i \cdot V)^n}_{\text{Specular}}$$

I_i, L_i, and R_i are the intensity, light vector, and reflection vector for the i^{th} light source. Like before, we have a single ambient light term. However, we now sum over lights in a scene, adding the contribution of each light source to the pixel $\mathcal{P}$. We do this because light is additive.

Implementing multiple light sources

You will modify Renderer.illuminate() to add a diffuse and specular component for each light source.

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

    Vector3 colourToReturn = new Vector3(0);

    ...

    // Add ambient light term to start with
    colourToReturn = colourToReturn.add(I_a.scale(C_diff));

    // Loop over each point light source
    List<PointLight> pointLights = scene.getPointLights();
    for (int i = 0; i < pointLights.size(); i++) {

        PointLight light = pointLights.get(i);
        double distanceToLight = light.getPosition().subtract(P).
            magnitude();
        Vector3 I = light.getIlluminationAt(distanceToLight);

        TODO: Calculate L, V, R, NdotL, and RdotV

        TODO: Calculate diffuse and specular terms

        TODO: Add diffuse and specular terms to colourToReturn
    }

    return colourToReturn;
}
```

The loop over the light sources in the scene has already been implemented for you. Your task is to compute diffuse and specular illumination terms for each light in the same way you did for Tick 1, and accumulate them in the colourToReturn variable.

5 Shadows

Shadows are an important visual cue so we can tell where objects are in relation to each other. Implementing shadows in a ray tracer is simple: when calculating the illumination of a point by a certain light source, we must first check if that

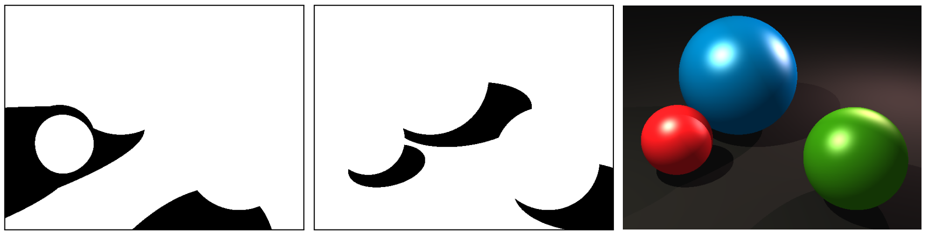

(a) Shadows cast by light 1 (b) Shadows cast by light 2 (c) The scene with shadows

Figure 3: The black pixels in (3a) and (3b) correspond to pixels in shadow for the corresponding light source. (3c) shows how shadows provide an important visual cue: we can now see the spheres and plane are touching.

point can "see" the light source. If it cannot, then we do not accumulate that light's diffuse and specular contributions for that point.

5.1 Implementing shadows

You will modify Renderer.illuminate() once more to consider shadows before adding the diffuse and specular component for each light source.

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

    for (int i = 0; i < pointLights.size(); i++) {

        FROM BEFORE: Calculate diffuse and specular terms

        // Check if point P is in shadow from that light or not
        Ray shadowRay = new Ray(P.add(L.scale(EPSILON)), L);
        boolean inShadow = false;

        TODO: Cast the shadowRay with findClosestIntersection to
        determine if P is in shadow or not, and set inShadow

        TODO: if not inShadow, add diffuse and specular to
        colourToReturn
    }

    return colourToReturn;
}
```

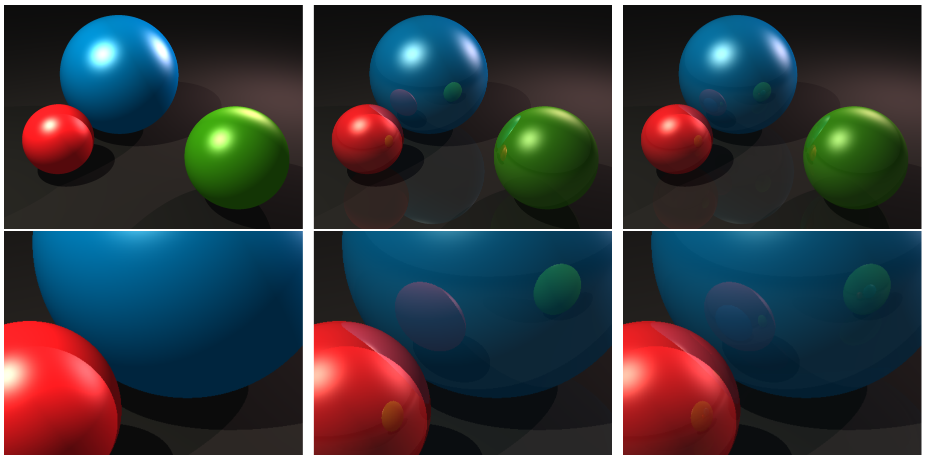


Figure 4: We can render reflections by "bouncing" our traced rays off objects in the scene. To ensure our renderer terminates, we must limit the number of times a ray can bounce. Here is the scene with 0, 1, and 2 bounces.

The idea is to only add the contribution for the i^{th} light source to colourToReturn if the path from P to that light source is clear. This is determined by casting a *shadow ray* from P towards the light source. If the shadow ray hits any object before reaching the light, then point P is in shadow. Otherwise, point P is directly illuminated by the light.

Note the shadow ray has already been created for you. Its starting position has been slightly adjusted by a small *bias* factor EPSILON. This prevents the shadow ray immediately intersecting the same object it started from.

For each point light source, implement the following:

1. Cast the shadow ray using scene.findClosestIntersection(shadowRay) to get a RaycastHit object.
2. If the RaycastHit's distance is less than the distance to the point light, set inShadow to true.
3. If inShadow is false, add the diffuse and specular terms for that light source to colourToReturn, otherwise do nothing.

Your renderer should now render shadows cast by the spheres onto the plane and each other. The output should look like Figure 3c.

6 Reflection

Aside from shadows, another effect that is easy to implement in ray tracing is reflection. Figure 4 shows the result of adding reflections to our scene. Your final task is to implement reflection using recursion in Renderer.trace().

Currently, the trace() method returns the *direct* illumination at a point P. To render reflection, we additionally calculate the *reflected* illumination by "bouncing" the traced ray off the surface.

Implementing reflection

You can see the Renderer.trace() method now includes a parameter bouncesLeft. bouncesLeft corresponds to the number of times the ray will bounce before returning the direct illumination.

```
protected Vector3 trace(Scene scene, Ray ray, int bouncesLeft) {
    ...

    // Calculate direct illumination at that point
    Vector3 directIllum = this.illuminate(scene, object, P, N);

    double reflectivity = object.getReflectivity()

    // If no bounces or reflection, return direct illumination
    if (bouncesLeft == 0 || reflectivity == 0) {
        return directIllum;

    } else {
        TODO: Calculate the direction R of the bounced ray

        TODO: Spawn a reflectedRay with bias

        TODO: Calculate reflectedIllum by tracing reflectedRay

        // Scale direct and reflective illum. to conserve light
        directIllum = directIllum.scale(1.0 - reflectivity);
        reflectedIllum = reflectedIllum.scale(reflectivity);

        return directIllum.add(reflectedIllum);
    }
}
```

The direct illumination at P (directIllum) is calculated using Renderer.illuminate().

For reflection you must implement the following:

1. Calculate the direction of the bounced ray R by reflecting the original ray's direction D in N. Remember, Vector3.reflectIn() computes a mirror-like reflection, so you should negate D before reflecting it. Make sure R and N are unit vectors.
2. Spawn a new ray reflectedRay with origin $(P + \epsilon R)$ and direction R. Make sure to adjust the origin by bias ϵ to prevent immediate self-intersection.
3. Calculate the reflected illumination colour using trace(scene, reflectedRay, bouncesLeft-1). We decrement bouncesLeft to make sure rays don't bounce around the scene forever.

Your renderer should now produce the same image as Figure 1.

Next time you'll explore *rasterization*: an alternative approach for rendering images that is faster than ray tracing, but less realistic.

7 Submission

The instruction for this submission is identical as for Tick 1.

Once you're happy with your tick's output, go to Moodle → *Tick 2* → *Tick 2 Submission*, switch to *Submission* tab, drag and drop two files: Plane.java and Renderer.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 sample scene bundled in the zip file (tick2.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.