Introduction to Graphics: Tick 1

Ray Tracing

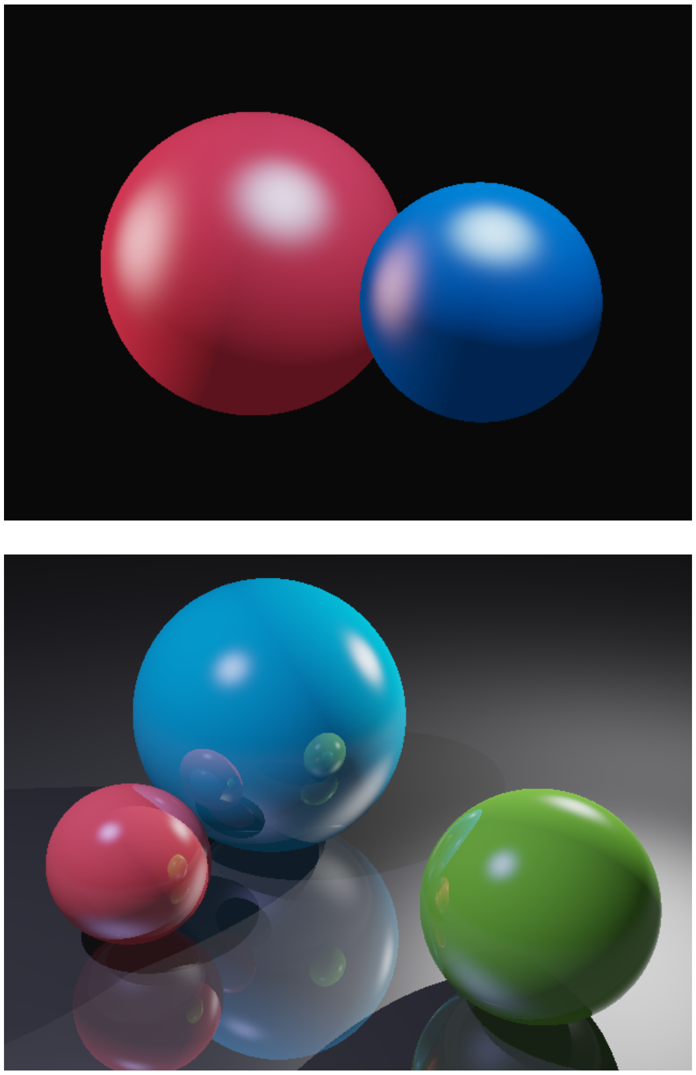


Figure 1: *the images you will create in this exercise*

Contents

Getting started **3**

1 Part 1 **5**

1.1 Scene setup 5

1.2 Intersecting with spheres 7

1.3 Shading 9

Checkpoint **11**

2 Part 2 **12**

2.1 Intersecting with planes 12

2.2 Shadows 13

2.3 Reflection 15

3 Rendering competition (optional) **17**

4 Submission **17**

Background

In this tick you will write code for a simple ray tracer, which will render a 2D image from a supplied 3D scene.

In real life, light sources (e.g. the sun) cast rays of light that reflect off objects and into our eyes, forming a 2D image on our retinas. Ray tracing models this by tracing light rays *backwards* through a 3D scene — rather than rays originating from a light source and then reflecting off an object into a camera, they are traced from camera to object to light source.

We therefore model our camera as casting a ray backwards into the 3D scene through each pixel of our 2D image. These rays will interact with objects in the scene, and we can use these interactions to colour the pixels of our 2D image by examining the object's colour and illumination at the point of intersection.

This tick is designed to be completed over the course of about ten days. We strongly recommend you adhere to this schedule, and use the supplied unit test to monitor your progress. If you would like to complete the starred tick too, you may wish to complete the main tick a little faster to give yourself time to work on the starred tick afterwards. Help sessions will be available if you get stuck.

Getting started

Download tick1.zip from the Moodle area *1A Introduction to Graphics → Tick 1*. After extracting this to a suitable directory, you should find the following files:

```
tick1
├─ src/uk/ac/cam/cl/gfxintro/crsid/tick1
│  ├─ BumpySphere.java
│  ├─ Camera.java
│  ├─ ColorRGB.java
│  ├─ Plane.java
│  ├─ PointLight.java
│  ├─ Ray.java
│  ├─ RaycastHit.java
│  ├─ Renderer.java
│  ├─ Scene.java
│  ├─ SceneLoader.java
│  ├─ SceneObject.java
│  ├─ Sphere.java
│  ├─ Tick1.java
│  ├─ UnitTest.java
│  └─ Vector3.java
├─ bumpmap.png
├─ bumpmap-test.png
├─ tick1.pdf
├─ tick1star.pdf
├─ test1.xml
├─ test1_reference.png
├─ test2.xml
└─ test2_reference.png
```

Please ensure you replace `crsid` with your CRSid, and modify package names accordingly.

The files you will be required to modify and submit for this tick have been highlighted. Note that much of the code has already been written for you. You should not modify this code, though you will need to understand it – familiarise yourself with the other files, and be prepared to answer questions from your ticker. The files `BumpySphere.java`, `bumpmap.png`, `bumpmap-test.png`, and `tick1star.pdf` are for the starred tick – you do not need to look at these files unless you are completing the starred tick.

The `main()` method for this exercise is in `Tick1.java`. It takes three optional arguments:

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

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

`-b, --bounces` : this argument will become relevant when we consider reflective surfaces later on in the tick.

Compiling and running the code

The instructions below are for compiling and running the code from the command line. Please refer to the document *Working with IDEs* for instructions if you are working with an IDE.

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

```
mkdir out
javac -d ./out src/uk/ac/cam/cl/gfxintro/crsid/tick1/*.java
```

where you should replace `crsid` with your CRSid.

Here we create the directory `out`, then use the `-d` option to specify that the compiled classes should be placed in the newly created directory. It is generally good practice not to mix sources files with compiled code.

You can run the program with the following command:

```
java -classpath ./out uk.ac.cam.cl.gfxintro.crsid.tick1.Tick1
↪  --input test1.xml --output output.png
```

where you should replace `crsid` with your CRSid.

Compile the source code, and run it with the arguments shown above. As our 3D scene is currently empty and we have not implemented our ray tracer, this will output a blank image to the file `output.png`.

Vector3 and ColorRGB

You have been provided with a Vector3 class and a ColorRGB class to use in this exercise. Vector3 contains three fields (`x`, `y`, and `z`) representing the components of a vector in 3D space, and useful methods that allow to you add, subtract, multiply, and scale the vectors.

ColorRGB is based heavily off Vector3, and has three corresponding fields (`r`, `g`, and `b`) representing the three colour components red, green, and blue. You should familiarise yourself with both of these classes before continuing.

1 Part 1

1.1 Scene setup

A scene is a 3D description of the virtual world that we want to render – it contains information on the objects and lights in the virtual world, and their positions and colours. Our ray-tracer will take the scene as input, and then trace rays through the scene to determine the colours 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` – spheres specified by a position, radius, and colour.

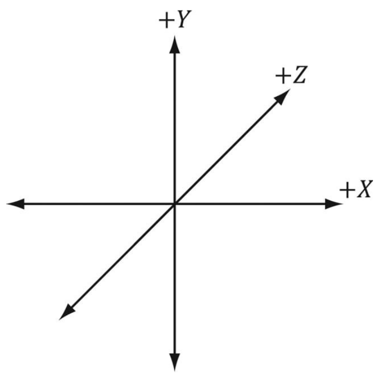
`plane` – planes specified by a point on the plane, normal vector, and colour.

`ambient-light` – background illumination, specified by colour only.

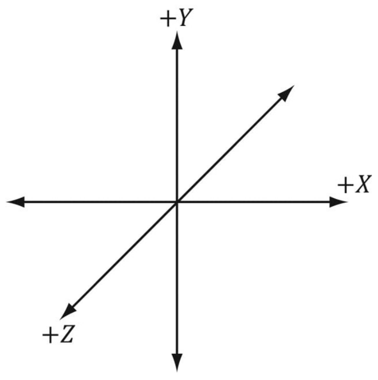
`point-light` – a light source that has a position, colour, and intensity.

We can specify parameters by setting attributes for each element. For example, colour is set with `<... colour="#FF0000"/>` using hexadecimal RGB colour codes.

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



(a) *left-handed coordinate system*



(b) *right-handed coordinate system*

Figure 2: *we use a left-handed coordinate system, in which objects with more positive* z *coordinates are further away from the camera at the origin. In Tick 2 you will encounter OpenGL, which uses a right-handed system.*

Example scene

Here is a basic example scene:

```
<scene>
  <ambient-light colour="#FFFFFF" intensity="0.1"/>
  <point-light x="3" y="3" z="3" colour="#FFFFFF" intensity="100"/>
  <sphere x="0" y="0" z="6" radius="1" colour="#FF0000"/>
  <plane x="0" y="1" z="2" nx="0" ny="0" nz="-1" colour="#4AAA0A"/>
</scene>
```

This scene contains a dark grey ambient light, a white point light at $(3, 3, 3)$ with intensity 100, a unit-radius red sphere at $(0, 0, 6)$, and a green plane perpendicular to the plane of the image, passing through $(0, 1, 2)$.

The elements sphere and plane can contain additional attributes, which control material properies: kD – diffuse factor, kS — specular factor, alphaS — specular “roughness” parameter and reflectivity, which specifies material reflectivity between 0 and 1.

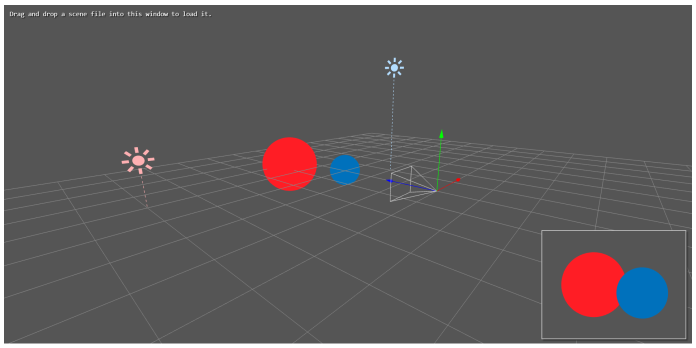


Figure 3: *the online scene previewer – the camera is the white wireframe pyramid positioned at the origin, and the image it views is in the bottom-right panel.*

Define a new scene

Your first task is to build a scene according to a specification. Create a new scene file in the `tick1` folder called `basic_scene.xml`, and include the following:

- An ambient light with colour `#FFFFFF` and intensity 0.05.
- Two point light sources of intensity 80 at $(1, 3, 2)$ and $(-5, 1, 4)$, with colours `#B3DDFF` and `#FFB0B2` respectively.
- A sphere at $(0.55, -0.16, 3.5)$ with radius 0.5 and colour `#0030BC`.
- A sphere at $(-0.55, 0, 5)$ with radius 0.9 and colour `#FF1122`.

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

1.2 Intersecting with spheres

Our ray-tracer will cast rays through each image pixel from the camera – this has been implemented for you in the Camera and Renderer classes. Our camera is positioned at the origin, pointing along the positive z-axis.

The template code for the renderer outputs blank images of colour `BACKGROUND_COLOUR`, as we have not implemented the code for finding the intersection of a cast ray with an object in the scene. Your task in this section is to implement the `Sphere.intersectionWith(Ray ray)` method to handle these intersections correctly.

Theory

Given:

- A ray $\mathbf{p}$, defined by $\mathbf{p} = \mathbf{o} + s\mathbf{d}$, where $\mathbf{o}$ is origin vector of the ray, $\mathbf{d}$ is the direction vector of the ray, and $s \geq 0$ is some non-negative distance travelled by the ray;
- A sphere defined by $\mathbf{q}$, such that $(\mathbf{q} - \mathbf{c}) \cdot (\mathbf{q} - \mathbf{c}) - r^2 = 0$ with centre $\mathbf{c}$ and radius r;[1]

we can determine if the ray intersects with the sphere by substituting the ray equation into the sphere equation, and solving the resulting quadratic equation for ray parameter s (***Equation 1***). We can then substitute s back into the ray equation to determine the point of intersection.

$$\begin{aligned}
\Big((\mathbf{o} + s\mathbf{d}) - \mathbf{c}\Big) \cdot \Big((\mathbf{o} + s\mathbf{d}) - \mathbf{c}\Big) - r^2 &= 0 \\
\Big((\mathbf{o} - \mathbf{c}) + s\mathbf{d}\Big) \cdot \Big((\mathbf{o} - \mathbf{c}) + s\mathbf{d}\Big) - r^2 &= 0 \\
s^2 \underbrace{(\mathbf{d} \cdot \mathbf{d})}_{a} + s \underbrace{\big(2\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})\big)}_{b} + \underbrace{(\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2}_{c} &= 0
\end{aligned} \tag{1}$$

We can see that this takes the form of a quadratic equation in s (i.e. $as^2 + bs + c = 0$, where the coefficients a, b, and c are as shown above). Take care when implementing this – if the discriminant ($b^2 - 4ac$) is negative then solutions for s will be imaginary, which will mean no intersection in our real interpretation. In most cases, you will find two solutions for s – make sure you use one closest to the ray origin, and therefore closest to the camera.

Implementation

The Sphere class contains an `Sphere.intersectionWith(Ray ray)` method, which you must now implement. We require it to determine if the Ray object `ray` interesects with the Sphere object **`this`**, returning a RaycastHit object. If there is no intersection, the RaycastHit object will be empty, otherwise it will contain information on the intersection point.

You should therefore modify this method in `Sphere.java` to implement the following algorithm:

1. Determine if the ray intersects with the sphere using the discriminant – if there is no intersection, return an empty RaycastHit.

[1] Note: The "$\cdot$" symbol means the scalar, or "dot" product of two vectors. The scalar product of a vector with itself gives the squared magnitude of that vector: $(\mathbf{q} - \mathbf{c}) \cdot (\mathbf{q} - \mathbf{c}) = |\mathbf{q} - \mathbf{c}|^2$.

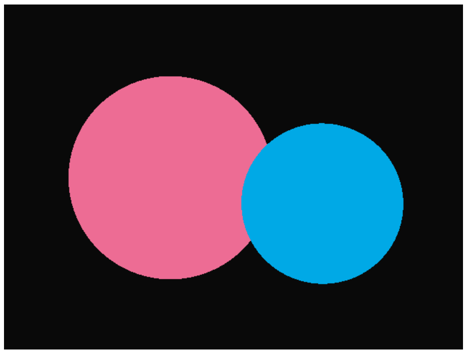


Figure 4: *the rendered image once ray-sphere intersection has been implemented.*

2. Compute the two solutions for s using the quadratic formula – if neither are positive, intersection occurs behind the camera, and so we return an empty RaycastHit.
3. Otherwise return a RaycastHit corresponding to the solution closest to the camera.

We can create an empty RaycastHit by using a constructor with no arguments: `RaycastHit empty = new RaycastHit();`. Otherwise, the RaycastHit references the object hit (in this case the Sphere), the distance the ray travelled before intersection, the location of the intersection, and the normal to the object at the point of intersection. Examine the Ray, RaycastHit, and Sphere classes closely to determine the relevant methods to calculate these.

Using the scene we built previously, compile and run your program – your output should resemble ***Figure 4***. As you can see, the blue sphere correctly occludes the red one – we must now implement an illumination model to shade the spheres.

1.3 Shading

Our ray-tracer currently renders our objects without any illumination, colouring each pixel according to the object colour only. We will now implement the Phong illumination model in the `Renderer.illuminate()` method to correctly shade our objects with ambient light and any point light sources present in the scene.

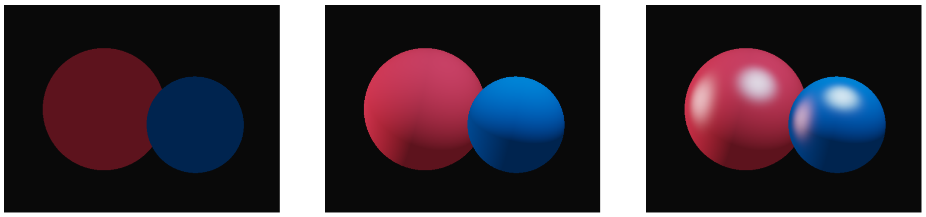

(a) *ambient illumination only* (b) *ambient and diffuse illumination* (c) *ambient, diffuse, and specular illumination*

Figure 5: *The scene shaded with the Phong illumination model. Note that the darker lines at the boundary of the diffuse reflection, where the cosine term become negative, are illusionary. They are caused by the Mach band illusion.*

Theory

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

$$\mathbf{I}_{\mathcal{P}} = \underbrace{\mathbf{C}_{diff}\, I_a}_{\text{Ambient}} + \sum_{i=1}^{m} \underbrace{\mathbf{C}_{diff}\, k_d\, I_i \max\big(0, \hat{\mathbf{n}} \cdot \hat{\mathbf{l}}_{\mathbf{i}}\big)}_{\text{Diffuse}} + \sum_{i=1}^{m} \underbrace{\mathbf{C}_{spec}\, k_s\, I_i \max\big(0, \hat{\mathbf{r}}_{\mathbf{i}} \cdot \hat{\mathbf{v}}\big)^{\alpha}}_{\text{Specular}}, \tag{2}$$

where $\mathbf{I}_{\mathcal{P}}$ is the resulting colour of the pixel, and we sum over the m point lights in the scene (i.e. adding diffuse and specular components from each light source). Note that the hat represents a unit vector (e.g. $\hat{\mathbf{n}}$). We represent colour in the equation as a 3D vector, with red, blue, and green components – this mirrors the implementation of the ColorRGB class. Bold font symbols are used for vectors, plain font symbols for scalars. The other terms are described below – see also ***Figure 5*** for a breakdown of the shading components.

Ambient lighting approximates the indirect illumination in the scene. $\mathbf{C}_{diff}$ is the diffuse colour of the surface (as specified by the object in the scene), and I_a is the ambient illumination intensity of the scene.

Diffuse lighting is based on the observation that the intensity of light reflected from a diffuse surface depends on the angle between the surface and incoming ray. k_d is the surface's diffuse coefficient, I_i the illumination intensity (or RGB colour) from the point light i, $\hat{\mathbf{n}}$ the unit surface normal, and $\hat{\mathbf{l}}_{\mathbf{i}}$ the (unit) direction to the point light i from the surface point.

Specular lighting adds mirror-like reflection, giving shiny highlights. The highlights are the strongest in the direction of reflected ray. The colour of the specular reflection is determined by the colour of the light, not the colour of the object — a green light will give green highlights for example. k_s is the surface's specular coefficient, $\hat{\mathbf{r}}_{\mathbf{i}}$ is the reflection of $\hat{\mathbf{l}}_{\mathbf{i}}$ in $\hat{\mathbf{n}}$, $\hat{\mathbf{v}}$ is the view vector (pointing to either the camera or the point of reflection), and α is the specular exponent, which controls how shiny the surface is (the larger the exponent, the narrower and more intense the highlight).

Note that $\hat{\mathbf{n}}$, $\hat{\mathbf{l}}$, $\hat{\mathbf{r}}$, and $\hat{\mathbf{v}}$ should all be unit vectors. We have set k_d and k_s to constants in this exercise.

Implementation

The code you are given in the Renderer class contains a `Renderer.illuminate(Scene scene, SceneObject sceneObject, Vector3 P, Vector3 N, Vector3 O)` method, which you must now implement. It will determine the colour of the SceneObject object `sceneObject` found in our `scene` at the point given by position vector `P`, at which the normal vector is `N`. This colour is then returned as a ColorRGB object.

You should therefore modify this method in `Renderer.java` to implement the Phong illumination model, rather than just returning the object colour. Be sure to use the methods implemented for you in the Vector3 and ColorRGB classes. You will need to calculate $\hat{\mathbf{l}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{r}}$ (remember that the camera is always positioned at the origin in our ray-tracer).

If you check `render()` method, you will notice that the returned colour is not written directly to the image, but is passed to a tone mapping function. This is because the colour computed using a shading model represents light quantities, which can span the range that is larger than what can be displayed on a monitor. The colour values also need to be encoded for the display (gamma) to be correctly reproduced. You will learn about tone mapping and display encoding in the last lecture.

Now our renderer should produce an image resembling ***Figure 1***. Next, we will extend our ray tracer to handle planes, reflective surfaces, and shadows.

Checkpoint

You are now halfway through this tick – please test your solution thoroughly before proceeding.

Testing our implementation

Our renderer should now be able to read an input scene `.xml` file specifying spheres, ambient light, and point lights and correctly render this to an output image. You can use the unit test found in `UnitTest.java` to check that your implementation behaves as expected – simply compile `UnitTest.java` and run the `main()` method as shown below. This will check you have correctly implemented spheres and shading.

```
javac -d ./out src/uk/ac/cam/cl/gfxintro/crsid/tick1/*.java
java -classpath ./out uk.ac.cam.cl.gfxintro.crsid.tick1.Tick1Tests
```

where you should replace `crsid` with your CRSid.

Ensure that the all tests run successfully before proceeding. If you encounter any problems, please come to a help session. Alternatively let us know on the *Help Forum for Graphics 1A* on Moodle.

2 Part 2

2.1 Intersecting with planes

In this section we will extend our ray tracer implementation to allow it to handle a new type of SceneObject – a Plane. We will modify `Plane.java` to implement the `Plane.intersectionWith(Ray ray)` method, as we did for our Sphere class.

Recall that we represent planes in our scene file as follows:

```
<plane x="0" y="1" z="2" nx="0" ny="0" nz="-1" colour="#4AAA0A"/>
```

where our plane is specified by a point (x, y, z), a normal vector (n_x, n_y, n_z), and a hexadecimal RGB color.

Theory

Given:

- A ray $\mathbf{p}$, defined by $\mathbf{p} = \mathbf{o} + s\mathbf{d}$ as before;
- A plane defined by $\mathbf{q}$, such that $(\mathbf{q} - \mathbf{r}) \cdot \mathbf{n} = 0$, where $\mathbf{r}$ is a point in the plane and $\mathbf{n}$ is the normal vector;

we can determine if the ray intersects with the plane by substituting the ray equation into the plane equation and solving for ray parameter s, as we did for ray-sphere intersections (***Equation 3***).

$$\begin{aligned} 0 &= \Big((\mathbf{o} + s\mathbf{d}) - \mathbf{r}\Big) \cdot \mathbf{n} \\ s &= \frac{(\mathbf{r} - \mathbf{o}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}} \end{aligned} \tag{3}$$

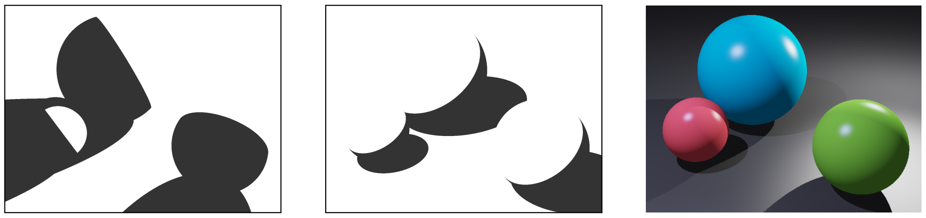
Implementation

You should modify the `Plane.intersectionWith(Ray ray)` method in `Plane.java` to implement the theory shown above, in the same way we did for ray-sphere intersections. As before, return a RaycastHit object, and ensure that any intersections found occur in front of the camera (not behind it).

Our ray tracer should now be able to render planes! Test this before moving on to the next section – you can use the scene provided in `test1.xml`.

2.2 Shadows

We will now extend our ray tracer so that it can render shadows. Shadows are an important visual cue – they will make our renderings appear more realistic, and will allow our viewers to tell where objects are in relation to each other (see ***Figure 6***). Our ray tracing method makes shadow implementation simple – when tracing a ray from an object to a point light, we simply check that the path of the ray is clear, and that no other objects intersect with the ray (and so cast a shadow on the object in question). If we determine that a point on the object is in shadow from a particular point light, then we do not include the contribution of that point light in shading the corresponding pixel. Note that pixels in shadow will still be illuminated by the ambient light component.



(a) *shadows cast by the point light at* $(2, 2, 4.5)$ (b) *shadows cast by the point light at* $(-2, 2.5, 1)$ (c) *the resulting scene*

Figure 6: *the scene rendered with shadows.*

Implementation

We must modify the `Renderer.illuminate(...)` method so that it checks for shadows before adding the diffuse and specular components for each point light.

We can achieve this by casting a `shadowRay` from the point we wish to illuminate (`Vector3 P`) to our point light (`PointLight light`), and determining if the `shadowRay` intersects with any other objects before reaching the point light – if it does, then the point we wish to illuminate is in shadow from this particular point light. We can therefore add the following skeleton code to our implementation:

```java
protected ColorRGB illuminate(...) {
  ...
  for(int i = 0; i < pointLights.size(); i++) {
    ...
          // Cast shadow ray
                      Ray shadowRay = new
                       ↪  Ray(P.add(N.scale(EPSILON)), L);

    //TODO: Determine if shadowRay intersects with an
     ↪  object
    //TODO: If it does not, add diffuse/specular components
  }
  return colourToReturn;
}
```

Note that `P` is the point of intersection with the object and `L` is the unit vector pointing to the light source. The origin of the `shadowRay` has been slightly adjusted in the direction of the normal by a bias factor `EPSILON` – this is to prevent the ray immediately intersecting the object from which it originates. Note that are moving the origin of the ray in the direction of the normal (away from the object) rather than in the direction of the light source. This is because if the light source was located behind the object, the adjustment towards it would bring the origin of the ray inside that object.

Your task is to implement the following algorithm:

- Determine if the cast shadowRay interesects with an object before it reaches the pointlight.
  - *If it does*: do not include the specular and diffuse components for this pointlight.
  - *If it does not*: add the specular and diffuse components as before.

You should use the `Scene.findClosestIntersection(Ray ray)` method, and you will also have to check if any returned RaycastHit objects represent an intersection occurring before the `shadowRay` reaches the **`PointLight`** (and not beyond the **`PointLight`**).

Our ray tracer should now be able to render shadows cast by the spheres onto each other and onto the plane. Ensure that your output resembles ***Figure 6c***.

2.3 Reflection

We can extend our ray tracer further to handle reflective surfaces, as shown in ***Figure 7***. Once again, this is easily done due to the nature of ray tracing – we can handle direct and reflected illumination separately, then add them to give the final result, scaling according to the reflectivity of the object. We have already been calculating direct illumination, and reflected illumination can easily be determined by spawning a new reflected ray at each intersection and tracing it through the scene. This can be achieved by recursively calling our `Render.trace(...)` method.

Implementation

We must modify the `Renderer.trace(Scene scene, Ray ray|, int bouncesLeft)` method. We will now use the third parameter (`int bouncesLeft`), which represents the number of bounces for which we will continue to trace the current ray, before returning the direct illumination. We can therefore add the following skleton code to our implementation:

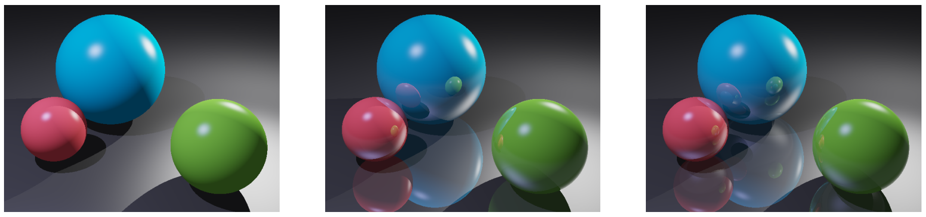
```
protected ColorRGB trace(Scene scene, Ray ray, int bouncesLeft) {
  ...
  // Calculate direct illumination at the point
  ColorRGB directIllumination = this.illuminate(scene, object, P,
  ↪  N, O);

  // Get reflectivity of object
  double reflectivity = object.getReflectivity()

  if (bouncesLeft == 0 || reflectivity == 0) {
    // Base case - if no bounces left or non-reflective surface
    return directIllumination;
  } else { // Recursive case
    ColorRGB reflectedIllumination;

    //TODO: Calculate the direction R of the bounced ray
    //TODO: Spawn a reflectedRay with bias
    //TODO: Calculate reflectedIllumination by tracing reflectedRay

    // Scale direct and reflective illumination to conserve light
    directIllumination = directIllumination.scale(1.0 -
    ↪  reflectivity);
    reflectedIllumination =
    ↪  reflectedIllumination.scale(reflectivity);

    // Return total illumination
    return directIllumination.add(reflectedIllumination);
  }
}
```

Our recursive base case has been completed for you, as has the scaling and summation of direct and reflected illumination. You must implement the following:

1. Calculate the direction of the bounced ray **r** by reflecting the original ray direction in the normal vector. Note that `Vector3.reflectIn(...)` computes a mirror-like reflection and we require a bounce-like reflection, so you should negate the original direction before reflecting it. Ensure too that **r** and **n** are unit vectors.

2. Spawn a new ray `reflectedRay` with origin `P` in the direction of **r**. Make sure you adjust the origin by the bias factor `EPSILON`, as we did for our `shadowRay`. Note that when computing the specular light component (see ***Equation 2***), the new view vector $\hat{\mathbf{v}}$ is the direction to the point of reflection, from the point point of intersection on the new object. The view vector should not point to the camera for reflected rays.

3. Recursively call `Renderer.trace(...)`, decrementing `bouncesLeft` to ensure the recursion terminates and the rays do not bounce around the scene forever.



(a) *no reflection* (b) *one reflection bounce* (c) *two reflection bounces*

Figure 7: *the scene rendered with varying number of reflected ray bounces (i.e. varying the* `-b` *or* `--bounces` *parameter).*

We should now be able to render reflections! Experiment with altering the number of bounces using the third argument on the command line (`-b` or `--bounces`), and ensure your output images resemble those in ***Figure 7***. Your final output on the `test1.xml` file with two reflected ray bounces should resemble ***Figure 1***.

Testing our implementation

You can further test your submission by running the unit test in `UnitTest.java` as before, but with the `-a` or `--all` flag. This will run additional tests to ensure that you have correctly implemented planes, shadows, and reflection. Make sure you have passed all tests before submitting.

3 Rendering competition (optional)

Once you have a fully functional ray tracer, possibly with some Tick* extensions, you can design and render the most stunning scene you can think of. You can submit your rendered scene and the XML file with the scene description to the corresponding submission item in Moodle. The rendering competition is optional and there is no extra credit, but the best scene will be announced and featured on the course web page.

To qualify for the competition, you need to follow the rules:

- The ray tracer used to render the scene can include only the elements covered in Tick 1 and Tick 1* descriptions.
- The only allowed objects are planes, spheres, and cylinders.
- The XML scene description can be manually designed but it can also be generated procedurally.
- The scene can contain up to 200 objects.

4 Submission

Once you're happy with your code, submit `Plane.java`, `Renderer.java`, and `Sphere.java` on Moodle. Moodle will test your scene against the sample scenes bundled in the `.zip` file (`test1.xml` and `test2.xml`), as well as an additional scene. If you code passes all the tests, the automatic tester will award 0.5 mark.

This mark will be changed to 1.0 after a mini-viva on the ticking session (if you are selected for ticking).

If you encounter any problems, please come to one of the help sessions. Alternatively let us know on the *Help Forum for Graphics 1A* on Moodle. Please do not post your code on the forum.