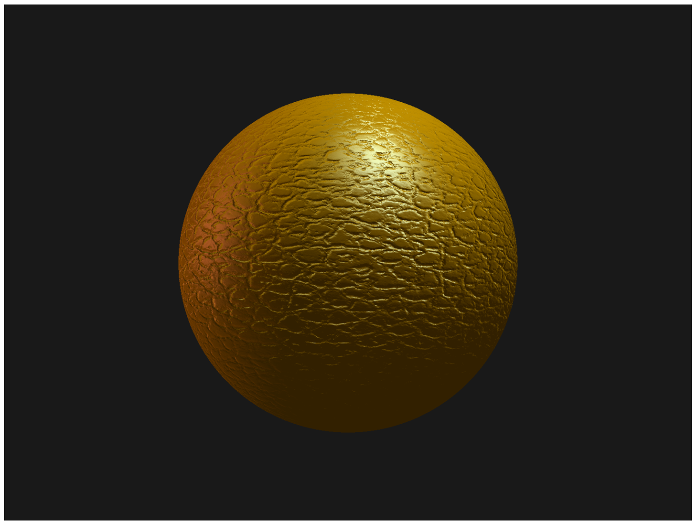
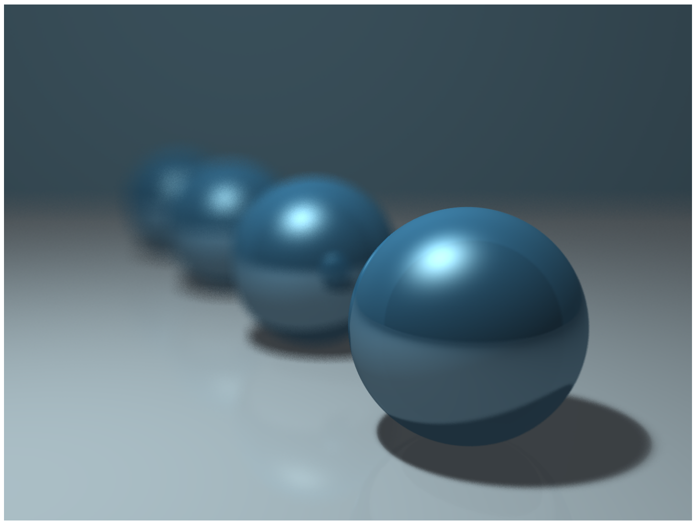
Introduction to Graphics: Tick 1*

Further Ray Tracing



(a) *bump-mapping*



(b) *distributed ray tracing*

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

Contents

Background

In this starred tick you will extend your ray-tracer further, to include more sophisticated effects such as bump mapping, or depth of field and soft shadows. Do not attempt this tick until you are satisfied with your implementation of the main tick.

Bump mapping is a process by which we can give a surface the appearance of bumpiness without changing geometry, simply by modifying the surface normals we use to calculate shading. Depth of field can give our renderings the appearance of depth, making them more realistic – our images will have a focal distance, at which objects will appear sharp and clear, but blurry elsewhere. Soft shadows make the shadows cast on our scene more realistic, by giving them soft, faded edges.

Depth of field and soft shadows are both achieved by a particular type of ray tracing known as distributed ray tracing. Rather than tracing single rays to/from a single point, we trace many rays distributed over an area, and average the contributions of each. For example, for depth of field we model our camera as having a finite aperture, and trace rays originating from randomly distributed points across the aperture (rather than the pinpoint camera we were using previously). For soft shadows, we trace multiple rays from the light, randomly distributed across a small sphere – thus we model our lights as spheres rather than infintesimal points.

You should only attempt the starred tick after completing the main tick and passing the automated tester. You will only need to attempt one of the options to gain a starred tick, though you may of course attempt more.

Getting started

Copy across your implementation of Tick 1 into a new package `uk.ac.cam.cl.gfxintro.crsid.tick1star`, where you should replace `crsid` with your CRSid. For this starred tick, you will need to modify the `BumpySphere.java` or the `Renderer.java` file.

In order to gain a starred tick, you need only complete one of the options below. You are welcome to complete both if you have the time, but you will not gain any extra credit for this.

1 Option 1: bump-mapping

Bump-mapping is a simple and powerful tool to make our surfaces seem more realistic. So far all our renderings have resulted in perfectly smooth surfaces, which resemble plastic. Bump-mapping, as the name suggests, allows us to make our surface appear bumpy, giving it some depth according to a grayscale image we refer to as a bump-map. We do this by modifying the surface normal, which in turn modifies how we shade the pixel (recall the surface normal is used to calculate diffuse and specular reflection in the Phong reflection model). Note that while this technique will make the surface appear bumpy, we are only modifying the surface normal – the edges of the object will still appear smooth. This technique is therefore limited in how much bumpiness we can add to a surface. Other techniques exist for mapping a surface to cause it to be truly bumpy, such as isosurface generation (see ***Figure 2***).

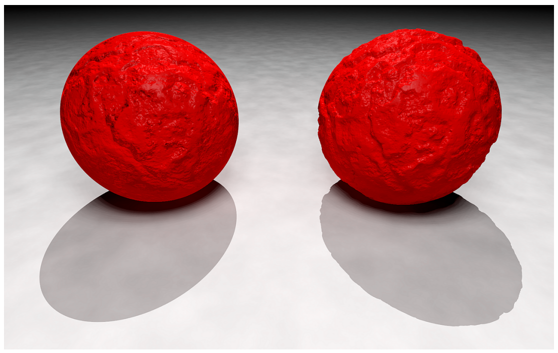


Figure 2: *the methods of bump-mapping (left) and isosurface generation (right). Note how bump-mapping does not affect the edge of the shape, but isosurface generation does.*

Your task in this section is to implement the `BumpySphere.getNormalAt(...)`

method so that it correctly uses the bump-map to return a modified surface normal.

BumpySphere

You have been provided with a BumpySphere class, which is designed to encapsulate a bump-mapped sphere – this class extends the Sphere class. So far the constructor reads in the image `bumpmap.png`, and uses it to create a heightmap of the surface. Later on we will override the `Vector3 getNormalAt(...)` method to implement our bump-mapping.

We can specify a bump-mapped sphere in our scene file as follows:

```
<bumpy-sphere x="0.0" y="0" z="3.5" radius="0.7" colour="#FAA000"
↪  bump-map="bumpmap.png"/>
```

This is the same as for an ordinary sphere, with the additional `<... bump-map="..." />` attribute, which specifies the name of the image from which we will derive our surface heightmap (located in the root directory of our program).

Theory

Given:

- A 2D bump-map parametrised by coordinates (u, v) where $0 \leq u, v \leq 1$;
- A sphere similarly parametrised by coordinates (u, v) where every point on the sphere is uniquely defined by a coordinate (u, v) (we will parametrise our sphere using spherical polar coordinates);

we can map the bump-map to the surface of the sphere in a one-to-one (bijective) relationship, and perturb the surface normal using ***Equation 1***:

$$\hat{\mathbf{n}}' = \text{normalize}\left(\hat{\mathbf{n}} + \underbrace{B_u(\hat{\mathbf{n}} \times \hat{\mathbf{P}}_v)}_{u \text{ direction}} + \underbrace{B_v(\hat{\mathbf{n}} \times \hat{\mathbf{P}}_u)}_{v \text{ direction}}\right), \tag{1}$$

where we are bump-mapping the surface normal of the surface defined by $\mathbf{P}$ (which can be mapped to coordinates (u, v)) – $\hat{\mathbf{n}}$ is the unit surface normal at a certain point on the sphere, B_u and B_v are scalar coefficients representing the amount we would like to perturb the normal in the u and v directions respectively at that point, and $\hat{\mathbf{P}}_u$ and $\hat{\mathbf{P}}_v$ are the unit tangent vectors in the directions of u and v at the point. To find these vectors, you can first define functions $F_x(u, v)$, $F_y(u, v)$, $F_z(u, v)$ mapping from spherical coordinates (u, v) into catesian coordinates (x, y, z) of the point on a sphere with a unit radius. Vector $\hat{\mathbf{P}}_u$ can be then computed as partial derivates at the point (u, v):

$$\hat{\mathbf{P}}_u(u, v) = \begin{bmatrix} \frac{\partial F_x}{\partial u}(u, v) & \frac{\partial F_y}{\partial u}(u, v) & \frac{\partial F_z}{\partial u}(u, v) \end{bmatrix} \tag{2}$$

The gradient above describes the direction of change in (x, y, z) with the change of u. Vector $\hat{\mathbf{P}}_v$ can be computed in a similiar manner. We use "$\times$" to denote the vector (cross) product. See ***Figure 3*** for a diagram.

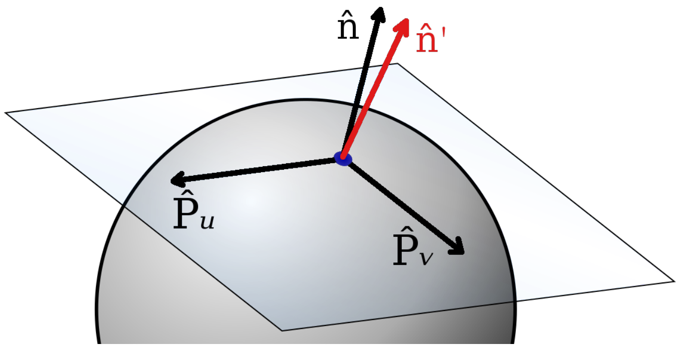


Figure 3: *a tangent plane to a sphere at the point in blue. The vectors* $\hat{\mathbf{n}}$*,* $\hat{\mathbf{P}}_u$*, and* $\hat{\mathbf{P}}_v$ *are shown in black, while* $\hat{\mathbf{n}}'$ *is in red.*

If we examine this equation closely, we can see that it effectively takes the original surface normal and perturbs it in two directions:

- in the direction of $(\hat{\mathbf{n}} \times \hat{\mathbf{P}}_v)$ scaled by a factor B_u
- in the direction of $(\hat{\mathbf{n}} \times \hat{\mathbf{P}}_u)$ scaled by a factor B_v

These two perturbations are in fact in the u and v directions respectively, as $\hat{\mathbf{P}}_u$, $\hat{\mathbf{P}}_v$, and $\hat{\mathbf{n}}$ are orthogonal to each other if we use an orthogonal coordinate system (such as spherical polar coordinates).

The scaling factors B_u and B_v are determined by our bump-map – a steeper slope (i.e. a bigger bump) in our bump-map should give a larger coefficent, resulting in greater perturbation. As our bump-map is effectively a heightmap, with darker pixels corresponding to lower height, we must find its gradient. Thus we can see that B_u and B_v are effectively the gradients $\frac{\partial B}{\partial u}$ and $\frac{\partial B}{\partial v}$ respectively, where $B(u, v)$ is our scalar bump-map.

In order to succesfully bump-map our sphere, we will need to:

1. Parametrise the sphere in (u, v) coordinates using spherical polar coordinates.
2. Determine the vectors $\hat{\mathbf{P}}_v$ and $\hat{\mathbf{P}}_v$ for a given point on the sphere.
3. Determine the scalar coefficients B_u and B_v using the bump-map and the calculated (u, v) coordinates.

Implementation

In the BumpySphere class, you must modify the `BumpySphere.getNormalAt(Vector3 position)` method to return the bump-mapped surface normal.

You should therefore do the following:

1. Determine the surface normal for a smooth sphere, and the (u, v) coordinates corresponding to the current `position`.
2. Calculate $\hat{\mathbf{P}}_u$ and $\hat{\mathbf{P}}_v$ according to how you have parametrised your sphere surface.
3. Calculate B_u and B_v for your current position – this is most easily done by taking your current pixel (specified by (u, v) coordinates) and subtracting from it the pixels below and to the right of it for v and u respectively.
4. Use the equation given above to return the new unit surface normal.

Using the scene file below, and the `bumpmap.png` image provided, you should obtain an image similar to ***Figure 1a*** (colours may differ slightly). Please generate the output image `screenshot.png` from this scene file for submission.

```
<scene>
  <ambient-light colour="#050505"/>
  <point-light x="1" y="3" z="2" colour="#B3DDFF" intensity="100"/>
  <point-light x="-5" y="1" z="4" colour="#FFB0B2"
  ↪  intensity="100"/>
  <bumpy-sphere x="0.0" y="0" z="3.5" radius="0.7" colour="#F45B00"
  ↪  bump-map="bumpmap.png"/>
</scene>
```

Testing

Please test your solution thoroughly – use the `bumpmap-test.png` file to ensure that you have correctly bump-mapped your sphere. The most common mistake is to invert the bump-map – the test map should give you raised letter “H”s, and lowered letter “L”s.

2 Option 2: distributed ray tracing

We will now extend our ray-tracer to render soft shadows and depth of field, using the method of distributed ray-tracing. Rather than originating our rays from a single point, we will cast many further rays, spreading their origins over some defined area by sampling from a probability distribution. We can then interpolate values and average contributions for these cast rays, and so create more subtle effects at the expense of some rendering time.

2.1 Soft shadows

Theory

Until now, we have treated our light sources as point lights, resulting in hard shadows with a clearly defined edge. We would like to have softer, more realistic shadows that have faded edges – in real life lights are not point sources. In order to do this, we can distribute our light sources over a sphere, to give spherical light sources rather than point light sources.

In the main tick, we spawned a shadow ray to each point light for each camera ray that we traced into the scene, and thus determined if the point was in shadow from that light or not. Now instead we can spawn multiple shadow rays, tracing them all over our spherical light source randomly (for simplicity – more advanced effects will trace a larger proportion of rays to the centre of the light, as this gives a more realistic effect. We will then use the ratio of the number of shadow rays that reach the light to the number that are occluded by another object in the scene to determine how much the point is in shadow from that light. Note that this will in fact give slightly different images each time we render a scene – by introducing a random variation, our ray-tracer is no longer deterministic.

Implementation

You will find it useful to have the following constants defined:

```
// Distributed shadow ray constants
private final int SHADOW_RAY_COUNT = 10; // no. of spawned
↪  shadowRays
private final double LIGHT_SIZE = 0.4; // size of spherical light
↪  source
```

You will need to modify your `Renderer.illuminate(...)` method to implement the following:

For each light:

1. Cast `SHADOW_RAY_COUNT` shadow rays from the point you are illuminating and determine if each is in shadow from the light
   - Each shadow ray is a ray from the point you are illuminating to a random location within the spherical light source. You may find the method `Vector3.randomInsideUnitSphere()` useful for creating random vectors of less than unit length, and the constant `LIGHT_SIZE` useful for setting the size of your spherical light.
   - Determine if each shadow ray is in shadow (i.e. is occluded from the light by another object in the scene) as we did before.
2. Determine the proportion of shadow the light casts on the point you are illuminating by using your shadow rays.
3. Scale the diffuse and specular components of the light by the proportion of shadow.

2.2 Depth of field

Theory

Until now, we have treated our camera as a pinhole aperture, resulting in perfect focus for all distances. In reality cameras have finite apertures, which result in a focal length – objects at this focal length appear sharp and in focus, and objects nearer/further away appear blurred and out of focus.

In order to achieve this depth of field effect, we will distribute our camera rays over an aperture. ***Figure 4*** shows the difference between our old pinhole camera and our new aperture one – we now trace multiple rays distributed across the aperture, but each ray intersects the focal plane at the same point the original ray did. We can therefore see that objects at the focal plane will have no blur (as all distributed rays hit the same point in the focal plane), whereas objects away from the plane will be blurred (as the distributed rays diverge and we average their contributions).

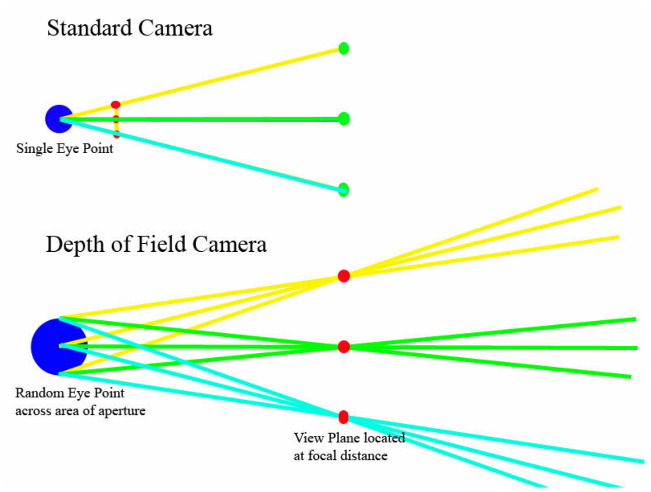

Standard Camera
Single Eye Point
Depth of Field Camera
Random Eye Point across area of aperture
View Plane located at focal distance

Figure 4: *basic point-camera ray-tracing (top) compared to depth of field distributed ray-tracing (bottom). Note the rays intersect the focal plane at the same point, but in distibuted ray-tracing their origins are spread over an aperture.*

Implementation

You will find it useful to have the following constants defined:

```
// Distributed depth-of-field constants
private final int DOF_RAY_COUNT = 50; // no. of spawned DoF rays
private final double DOF_FOCAL_PLANE = 3.51805; // focal length of
↪  camera
private final double DOF_AMOUNT = 0.2; // amount of DoF effect
```

You will need to modify your `Renderer.render(...)` method to implement the following:

For each pixel:

1. Determine the point the original, single ray would have intersected the focal plane (situated at length `DOF_FOCAL_PLANE` from the camera in the z direction).

2. Cast `DOF_RAY_COUNT` rays randomly from our camera aperture through this intersection point.

   - Use an square aperture in the xy-plane, of dimension `DOF_AMOUNT`. You may find the `Math.random()` function useful for generating a random number in the range $[-1, 1]$.

    - Calculate the direction of each new ray, and so create each ray from its own new origin, passing through the focal plane at the intersection point.

3. Trace each ray through the scene to determine the colour it gives, and average all the contributions.

4. Colour the pixel with this averaged colour, using the same BufferedImage methods as before.

You should now be able to render images with soft shadows and depth of field – note that your renderings will be far slower now, as you are casting many times as many rays per pixel, then many times as many shadow rays per point. How much slower do we expect our renderings to be?

Using the scene file below, you should obtain an image similar to ***Figure 1b*** (colours may differ slightly). Please generate the output image `screenshot.png` from this scene file for submission.

```
<scene>
<ambient-light colour="#050505"/>
<point-light x="-1.5" y="4.5" z="2" colour="#e1f2ef"
↪  intensity="450"/>
<sphere x="0.5" y="-0.26398" z="3.51805" radius="0.5"
↪  colour="#051a34"/>
<sphere x="-0.2" y="0.01504" z="4.99187" radius="0.5"
↪  colour="#051a34"/>
<sphere x="-0.9" y="0.29406" z="6.46569" radius="0.5"
↪  colour="#051a34"/>
<sphere x="-1.6" y="0.57308" z="7.93951" radius="0.5"
↪  colour="#051a34"/>
<plane x="0.0" y="-0.47623" z="5.085" nx="0" ny="0.9825"
↪  nz="-0.186" colour="#82898f"/>
<plane x="0.0" y="0" z="12" nx="0" ny="0" nz="-1"
↪  colour="#516e7e"/>
</scene>
```

3 Submission

Once you're happy with your code, go to the Tick 1* submission area on Moodle. If you have chosen the bump-mapping option, please submit `BumpySphere.java`. If you have chosen the distributed ray-tracing option, please submit `Renderer.java`. Both options require you to submit `screenshot.png`, the output file you have generated from the relevant scene file. Your tick and generated image will have to pass visual inspection at a ticking session – your ticker will then award you a starred tick.

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.