Lab 9: Ray Tracer extensions

Shadows and Mirrors

3D Computer Graphics

Introduction

No new jar file is provided for this Lab. The idea is that you keep on working on your version of the rendering framework which you obtained after finishing the exercises of Lab 8. However, it is strongly advised that you make a new fresh copy of the project of Lab 8 and rename this copy to 3DCG_Lab9. This will make it easier for you to look again at the work you did in each Lab when you study for the exam later on.

In this Lab session, we will add two main features to our ray tracer: support for shadows and mirror reflections. This requires calculations which slow down the rendering process even more. Your computer/laptop may not able to carry out these calculations in a reasonable time. If this is the case, note that you can reduce the rendering time of your graphical application by reducing the size of the image on the screen. For example, changing the `canvas.width` and `canvas.height` values in the configuration file of your graphical application to 200 and 150, respectively, will reduce the computing time by a factor 16.

Exercise 1

a) Download the `simpleScene2.sdl` file and store it in the `resources` folder.

b) Open this file and study its content. Make sure you understand all commands.

c) Create a new package `apps.app4`.

d) Create a new graphical application (`App4`) in this package.

e) Configure this graphical application as follows:

   – The scene to be rendered is described in the `simpleScene2.sdl` file.

   – The width and height of the canvas are 800 and 600 pixels, respectively.

- The eye of the camera is located at the position $(0, 10, 45)$.
- The camera is aimed at the origin.
- The upwards vector of the camera is in the direction of the y-axis.
- The worldwindow has a width and height of $4/3$ and 1, respectively, and is located 1 unit in front of the camera.

f) Run `App4` and make sure you get the expected image on your screen.

We will use this graphical application to check our support for shadows and mirrors.

Exercise 2

A ray tracer computes the colour of each pixel of the final image by casting a ray through this pixel into the scene, determining the closest intersection point (the hitPoint) and computing the colour of this intersection point based on a shading model. The latter is accomplished by the `shadeHit` method of the `RayTracer` class. Support for shadows and mirrors requires changing the implementation of this method considerably. Instead of implementing all these extensions in the same class, we will use several `RayTracer` classes so that the user can configure whether the scene should be rendered with shadows or mirror reflections turned on or off.

a) Create a `BasicRayTracer` class (in the `raytracer` package) which extends the `RayTracer` class. Create one constructor which takes a `Scene` object and passes it on to the constructor of its superclass.

b) Copy the implementation of the `shadeHit` method of the `RayTracer` class (and possibly other methods of the `RayTracer` class which are called by this implementation) to the `BasicRayTracer` class. (Do not copy the `shade` and `getBestIntersection` methods!)

c) Make the `shadeHit` method in the `RayTracer` class abstract and protected. So remove its implementation and all other methods in this class which are only called by this implementation.

d) Make the `RayTracer` class abstract.

e) Make the instance variable `scene` of the `RayTracer` class protected.

By refactoring our rendering framework in this way, all future extensions of our ray tracer can be implemented as a subclass of the abstract class `RayTracer`. Next, we add code which allows the user to configure which extension of our ray tracer should be used to render the scene. (So far, we have only one subclass `BasicRayTracer` but this will change soon.)

Exercise 3

a) Open the `RayTraceRenderer` class. Note that its instance variable `rayTracer` is initialized with a `RayTracer` object in its constructor. This is not possible anymore as we made the `RayTracer` class abstract. Instead, we need to initialize this instance variable with one of the subclasses of `RayTracer`. We will delegate this work to a factory class.

b) Create a class `RayTracerFactory` in the `renderer.raytracer` package.

c) Add the following method to this class.

```
public static RayTracer createRayTracer(Scene scene,
                                        Properties prop){
  String raytraceMode = prop.getProperty("raytrace.mode");
  if(raytraceMode.equals("basic"))
        return new BasicRayTracer(scene);
  return new BasicRayTracer(scene);
}
```

d) Change the constructor of the `RayTraceRenderer` class so that it makes use of this factory class to initialize its `RayTracer` object.

Note that `RayTraceFactory` currently always returns a `BasicRayTracer` class. In the future, the user can configure this in the configuration file by means of the `raytrace.mode` key.

d) Instead of having to change the configuration file of all existing graphical applications, we will simply change the `default.cfg` file which contains the default configuration settings. Add the following line at the end of this file.

```
raytrace.mode = basic
```

e) Run `App4` again to make sure your refactoring did not break your code.

The `BasicRayTracer` class is currently the only subclass of the abstract `RayTracer` class. It only implements a simple shading model taking into account diffuse and ambient light. Next, we will extend our ray tracer with support for shadows.

Part I : Shadows

Exercise 4

In order to add support for shadows, we need a way to determine whether a point on a 3D object is in shadow or not. This can be accomplished by casting a new ray from this point to the light source and determining whether this ray intersects a shape in the scene. This type of rays are called *shadow feelers*. A shadow feeler is always initialized as follows:

- the start point is the point for which one wants to know whether it is in shadow or not,
- the direction is the vector from the start point to the position of the light source.

Note that the `GeomObj` interface contains the method

```
public Intersection intersection(Ray ray);
```

to compute the closest intersection point between a `Shape` object and the given `Ray` object. Note that this method cannot be used for shadow feelers because we are only interested in intersection points which have a t-value between 0 and 1. Furthermore, we don't need all the intersection data given by this method. We only need to know whether there is an intersection or not. Therefore, we need a new method to determine whether a shadow feeler intersects a `Shape` object.

a) Add the following method to the `GeomObj` interface.

   ```
   public boolean hit(Ray ray);
   ```

   It is assumed that the `Ray` object given to this method is a shadow feeler which is initialized as described above.

b) Implement this method in all shapes which implement the `GeomObj` interface. Take the following hints into account:

   – The implementation of the `hit` and `intersection` method are very similar, so you can simply copy your implementation of the `intersection` method as starting point for your implementation of the `hit` method.

   – The `hit` method does not need to return data about the closest intersection so as soon as you know there is an intersection or not, return the correct boolean value instead of computing these intersection data (hitPoint, hitNormal, etc.).

   – The t-value should not only be larger than zero but also smaller than 1!

Exercise 5

In this exercise, we will create a new subclass `RayTracerWithShadow` of the `RayTracer` class which adds support for shadows.

a) Copy the `BasicRayTracer` class and rename it to `RayTracerWithShadow`.

b) The `shadeHit` method of the `RayTracerWithShadow` class should be implemented like this:

```
protected Colour shadeHit(Ray ray, Intersection best) {
  Colour colour = new Colour();
  for all lights in the scene{
    add the ambient component to the colour
    add the diffuse component to the colour
  }
  return colour;
}
```

Refactor this code so that it adds supports for shadows. Use the following pseudocode as a starting point.

```
protected Colour shadeHit(Ray ray, Intersection inter) {
  Colour colour = new Colour();
  Create a shadow feeler and set its start point
  for all lights in the scene{
    add the ambient component to the colour
    compute and set the direction of the shadow feeler
    if(not in shadow){
      add the diffuse component to the colour
    }
  }
  return colour;
}
```

The test "not in shadow" can be implemented by using a separate private method

```
private boolean isInShadow(Ray ray) {
  // todo: implement
}
```

which determines whether the given shadow feeler intersects one of the shapes in the scene. (See the slides of this Lab for more information.)

Exercise 6

Next, we add the `RayTracerWithShadow` class to the `RayTracerFactory` class.

a) Your current implementation of the `createRayTracer` method of the `RayTracerFactory` class only returns a `BasicRayTracer` object. Change this implementation to:

```
public static RayTracer createRayTracer(Scene scene,
                                        Properties prop){
    String raytraceMode = prop.getProperty("raytrace.mode");
    if(raytraceMode.equals("shadow"))
```

```
        return new RayTracerWithShadow(scene);
  return new BasicRayTracer(scene);
}
```

b) Add one line to `app4.cfg` so that you can check whether your shadow support works.

c) Run `App4.java`. Do you get the expected result? Explain.

Part II : Mirrors

The aim of the following exercises is to extend our ray tracer so that it can render 3D objects with mirror-like material properties. We will create a new subclass of the abstract `RayTracer` class which implements this new feature.

Exercise 7

a) Create a class `RayTracerWithReflection` in the `renderer.raytracer` package. Initially, its implementation is identical to the implementation of the `RayTracerWithShadow` class.

b) Adapt the constructor of the `RayTracerWithReflection` class so that it gets a second parameter as shown below.

```
public RayTracerWithReflection(Scene scene,
                               Properties prop) {
  super(scene);
}
```

(We will make use of this `Properties` object soon.)

c) Adapt the `RayTracerFactory` class so that it returns the appropriate object if the `raytrace.mode` property equals “reflection”.

Exercise 8

We will model the extent to which a 3D object has mirror-like behaviour as a material property `reflectivity` which has a value between 0 and 1. A `reflectivity` of 0 means no mirror-like behaviour while a `reflectivity` of 1 means the maximum degree of mirror-like behaviour.

a) Add a public floating point value `reflectivity` as instance variable to the `Material` class.

b) Adapt both constructors of the `Material` class accordingly. The default value of the `reflectivity` variable should be zero (no mirror-like behaviour).

It should be easy to specify the `reflectivity` of a 3D object in the sdl file similar to how this is done for the other material properties.

c) Open the file which describes the scene rendered by `App4`.

d) Add a line

```
reflectivity 0.7
```

before drawing the square and a line

```
reflectivity 0.1
```

before drawing the sphere. These extra two lines indicate a ground plane (square) with high mirror-like behaviour and a sphere with limited mirror-like behaviour.

Finally, our rendering framework needs to parse this new `reflectivity` token in the sdl file and process it appropriately.

e) Adapt the `SceneFactory` class so that a `reflectivity` token in an sdl file is correctly processed, similar to the way this was done for the other material properties.

At this point, our rendering framework correctly stores the `reflectivity` property (specified in the sdl file) in the `Material` object of each `Shape`. Next, we need to change the implementation of our ray tracer to use this information to correctly render 3D objects with mirror-like behaviour.

Exercise 9

Our ray tracer `RayTracerWithReflection` currently computes the colour of a hitPoint based on a shading model which takes the sum of different components: the ambient component and (in case the hitPoint is not in shadow) the diffuse component. We will add support for 3D objects with mirror-like behaviour by adding an extra component to the shading model.

This reflected light component is computed by casting a new ray in the mirror reflection direction and (again) computing the colour of the first hitPoint of this new ray based on the shading model. If this hitPoint belongs to an object which is shiny enough (meaning a `reflectivity` larger than or equal to 0.1), computing the colour of this hitPoint results in yet another ray to be cast in the mirror reflection direction at this hitPoint.

It is clear that we need a stop criterion to stop this recursive lighting calculations. One easy stop criterion is to introduce a maximum recursion depth. This also requires a way to keep track of the current recursion depth when casting a new ray.

a) Add a public int called `recursionDepth` as instance variable to the `Ray` class.

b) Make sure this instance variable is always initialized with a value equal to 1.

c) Add the following line

```
raytrace.reflection.maxRecursionDepth = 2
```

at the end of the `app4.cfg`.

Our ray tracer supporting mirror-like behaviour should take this new parameter into account.

d) Add a private instance variable `maxRecursionDepth` (an integer) to the `RayTracerWithReflection` class.

e) Initialize this instance variable in the constructor by reading the appropriate property from the given `Properties` object.

Exercise 10

As stated before, our ray tracer `RayTracerWithReflection` will add support for 3D objects with mirror-like behaviour by adding an extra component to the shading model. This can be done by changing the implementation of its `shadeHit` method according to the following pseudocode

```
protected Colour shadeHit(Ray ray, Intersection best) {
  Colour colour = new Colour();
  Create a shadow feeler and set its start point
  for all lights in the scene{
    add the ambient component to the colour
    compute and set the direction of the shadow feeler
    if(not in shadow){
      add the diffuse component to the colour
    }
  }
  if(ray.recursionDepth <= maxRecursionDepth and
          hitObject shiny enough){
    Ray reflRay = computeReflectedRay(ray, best);
    colour.add(the colour of reflRay multiplied by
          the reflectivity of hitObject);
  }
  return colour;
}
```

Take the following hints into account when implementing this pseudocode:

- We consider a hitObject shiny enough as soon as its material property `reflectivity` is larger than or equal to 0.1.
- The method `computeReflectedRay` should compute a `Ray` object with the hitPoint as start point and the mirror reflection direction as direction. Do not forget to increase the recursionDepth value of this reflected ray with one compared to the given `Ray` object to be able to keep track of the recursion depth.
- The colour of `reflRay` can be computed by using a method which you have implemented before. It is the method which computes the first intersection point of a ray (in this case the reflected ray) with all the 3D objects in the scene and returns the colour of this intersection point.
- Note that in the pseudocode above, the colour of `reflRay` is multiplied by the `reflectivity` parameter of the hitObject to reduce the intensity of the colour of `reflRay` in case of 3D objects with only little mirror-like behaviour.

Take a look at the slides of this Lab for more information.

Exercise 11

At this point, you are ready to test your support for mirror-like behaviour.

a) Configure `App4` so that mirror-like reflections are turned on.

b) Run `App4`. Do you get the expected result? Explain.

Play around to check your implementation. A few suggestions are given below.

c) Animate the camera and make sure everything works as expected.

d) Reduce the reflectivity of the floor to 0.2 and run `App4` again.

e) Render the buckyball instead of the sphere.

f) Stretch the buckyball by a factor 6 (instead of 4) along the y-axis.