Lab 5:
Ray tracing polygonal meshes
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 4. However, it is strongly advised that you make a new fresh copy of the project of Lab 4 and rename this copy to 3DCG_Lab5. This will make it easier for you to look again at the work you did in each Lab when you study for the exam in a few months.

The main goal of this Lab is to render a polygonal mesh with our ray tracer. But we will start with a few extensions and changes to our rendering framework which will allow to render 3D objects in another colour than red.

Exercise 1

a) Create a new `material` package in the `src` folder.

b) Create a new class `Material` in this package. Give it a public instance variable `colour` of the type `Colour`. (If you feel uncomfortable making instance variables public, you can also make them private and provide getter and setter methods.)

c) Add two constructors to the `Material` class: a default constructor which sets the colour to red and another, a copy constructor, which takes a `Material` object as parameter from which it copies all values (currently only one colour).

Next, we give every 3D object in the scene an object of this `Material` class so that we can easily retrieve its colour.

a) Open the `Shape` class in the `geomobj` package and add a public instance variable `mtrl` of the type `Material`.

b) Initialize this instance variable in the default constructor of the `Shape` class with the default constructor of the `Material` class.

Note that both the Square and Sphere class have a Material object now as they extend from the Shape class.

Exercise 2

a) Open the RayTracer class in the renderer/raytracer package and have a look at the shade method which you implemented in Lab 4.

In the current implementation, the best Intersection object is determined and if this object contains at least one hitPoint, the colour red is returned. Of course, instead of always returning a red colour, we want to make use of the Material class created in the previous Exercise. Do we have this information at our disposal in the shade method? In principle, yes. The first HitInfo object (of the best Intersection object) contains an instance variable hitShape which we can ask for its Material object. However, there are two reasons why we are not going to do this.

1. If our ray tracer would always take the Material object of the Shape class, it would mean that each shape would be rendered in one particular colour. Although this may seem a good choice for simple shapes like spheres and squares, this is certainly not the case for a polygonal mesh. For example, if your polygonal mesh represent a barn, it is very unlikely you want to render the walls and the roof in the same colour.
2. It will become clear in a few Lab sessions that our ray tracer does not need all the data which will be stored in the hitShape later on. It only needs to know the Material object of the Shape class. Following the principle of least knowledge we will adapt our code so that a HitInfo class only contains the hitMaterial object and not the entire hitShape.

Note that this code change also solves the first issue. Indeed, the

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

method of the Mesh class — which you will implement in a later exercise — can create a HitInfo object containing the Material object of the face which was hit by the given ray.

b) Open the HitInfo class in the renderer package. Change the instance variable hitShape into hitMaterial (of type Material) and adapt the constructor accordingly.

c) Rename the method getBestHitShape into getBestHitMaterial and change its implementation accordingly.

d) Adapt the Square and Sphere classes so that they only provide their Material object — and not themselves — to an object of the HitInfo class.

Now that the HitInfo objects created by the Square and Sphere class contain their colour, we have to make sure our ray tracer uses this information to render these shapes.

Exercise 3

a) Open the RayTracer class again and replace the line of code which returns the red colour by

```
return shadeHit(ray, best);
```

with best being the best Intersection object.

b) Implement this method in the RayTracer class by returning the colour stored in the Material object of the best hit. (Note that we do not use the ray provided as argument to this method but we will do so later on.)

Ok, let's take a step back and look at what we have done so far.

- We added support for the colour of a 3D object by adding a Material object to the Shape class.
- We adapted the two currently supported shapes (Square and Sphere) so that they return their colour to our ray tracer.
- We changed our ray tracer so that it uses the provided colour information to render these shapes.

What is the final missing link? Well, our ray tracer can only render a 3D object in the right colour if the Material object of this 3D object contains the right data. But how are this data set?

In the previous Lab, we introduced .sdl files to describe the background colour of the scene as well as the 3D objects in the scene. It seems a natural choice to add the material properties of these 3D objects to these file as well. This is the topic of the following exercise.

Exercise 4

a) Open the `simpleScene.sdl` file in the `resources` folder. Add two lines so that its content is identical to the following lines

```
  background 1 1 1
  colour 0 1 0
  square
  colour 0 0 1
  sphere
```

It should be clear that we want to render a scene with a green square and a blue sphere both centered at the origin.

It is important to realize that all material properties specified in a .sdl file are valid until the next command is given to change their value. This means that if one wants to render 100 red objects and 50 green objects, it suffices to specifiy the red colour once, list the 100 objects, specify the green colour once and finally list the 50 remaining objects in the .sdl file. In other words, if an objects such as a square or sphere is specified in an .sdl file, it is rendered with the current colour state.

Now that we have adapted our scene description language, we also have to adapt the classes which are responsible for parsing the .sdl files.

b) Open the enum `Token` in the `scene` package and add `COLOUR` as one of the tokens our parser should recognize.

c) Open the `SceneFactory` class and add a local variable `currMtrl` in the `createScene` method immediately after declaring the variable `objects`. Use the default constructor of the `Material` class to initialize `currMtrl`. This variable will keep track of the current material state while parsing the .sdl file.

d) The heart of the `createScene` method currently looks like

```
if(token.equals(Token.BACKGROUND.toString())){
  // set background colour
} else {
  // add object
}
```

In orde to support material properties, you need to change this into

```
if(token.equals(Token.BACKGROUND.toString())){
  // set background colour
} else if(!processMaterial(token, scanner, currMtrl)){
  // add object
}
```

If the token given to the processMaterial method is a material property (currently only colour), this method will use the given Scanner object to read the value(s) of this token, store it in the given Material object (currMtrl) and return true. If the token given to the processMaterial method is not a material property, false is returned.

e) Implement the processMaterial method in the SceneFactory class.

Finally, we need to make sure that when our parser encounters a shape token that this object gets a copy of the current material object.

f) Give the object currMtrl as third argument to the createShape method call in the createScene method.

g) Adapt the implementation of the createShape method in the createScene method so that it sets the given Material object to the created shape just before it returns this shape.

High time to check your work!

h) Run App1.java which renders the scene described in SimpleScene.sdl and check that you get the expected result.

So far, your ray tracer can render two primitive objects. The aim of the remaining exercises is to add support for rendering the polygonal meshes introduced in Lab 2. The main work will be the implementation of the algorithm to find the hit points between a ray and a polygonal mesh. This is the subject of Exercise 5.

Exercise 5

Your current project should already contain your Mesh (en Face) class which you implemented in Lab 2. First, we will connect this class to our rendering framework.

a) Open the Mesh class and let this class extend the Shape class. In this way, the Mesh class automatically inherits a Material object. Remember that the abstract Shape class implements the GeomObj interface without implementing the only method intersection of this interface. This method should be implemented in the Mesh class to allow polygonal meshes to be rendered by our ray tracer.

b) Implement the `intersection` method in the `Mesh` class. Use the information provided in the slides of this Lab.

In order to test your implementation, we need a new graphical application ...

Exercise 6

a) Make a copy of the `apps.app1` package and rename it to `apps.app2`. Rename the two files in this new package as well (`app2.java` and `app2.cfg`).

b) Make sure that `app2.java` uses the correct configuration file.

c) Open `app2.cfg` and change the value of the key `scene.file` to "resources/buckyball.sdl". This .sdl file does not exist yet, so let's create it.

d) Create a new "buckyball.sdl" file in the `resources` folder. This file should contain three lines which specify the scene properties: a black background, yellow as drawing colour, and one 3D object. The latter should be done with the line

```
mesh resources/buckyball.txt
```

This line says that we want to draw a mesh whose data can be found in the file "buckyball.txt".

e) Download the file "buckyball.txt" and put it in the `resources` folder.

f) Briefly study the content of this file. How many faces does this object have? And how many vertices? Do all faces have the same number of vertices?

Do we have all pieces in place to test your support for ray tracing polygonal meshes? Not yet. Note that the third line in the "bucketball.sdl" file contains a new token "mesh" which our current rendering framework does not recognize yet.

g) Make the necessary changes to the enum `Token` and the `createShape` method of the `SceneFactory` class.

h) Finally .... run `App2.java` and make sure you get the expected result.

To be continued ...