Lab 6: Shading

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 5. However, it is strongly advised that you make a new fresh copy of the project of Lab 5 and rename this copy to 3DCG_Lab6. 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 add a shading model which will determine the surface colour of a 3D model based on the light(s) in the scene. Adding a shading model to our ray tracer will significantly improve the photorealism of our images by depicting depth perception.

Exercise 1

First, we will add a `Light` class to our rendering framework to represent the light(s) in our scene.

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

b) Create a new class `Light` in this package. Give it two public instance variables:

   - `pos` of the type `Point`.
   - `colour` of the type `Colour`

   These instance variables represent the position and colour of the light source, respectively.

c) Add a constructor with two parameters which allow to initialize the instance variables of the `Light` object.

As light sources are considered part of the scene, they are stored in the `Scene` class.

d) Open the `Scene` class.

e) Add a private ArrayList of `Light` objects, called `lights` immediately after the ArrayList of `GeomObj` objects.

f) Initialize `lights` in the constructor and add a `getLights` and `setLights` method similar as was done for `objects`.

Exercise 2

In Lab 5, we added a `Material` class containing one colour to represent the colour of a 3D object. As our shading model will compute the colour of a point on the surface of a 3D object based on the ambient and diffuse colour of this 3D object, we need to adapt the `Material` class.

a) Open the `Material` class in the `material` package and rename the public instance variable `colour` into `diffuse`.

b) Add a second public instance variable `ambient` of type `Colour`. Initialize `ambient` to $(0.2, 0, 0)$ in the default constructor and make the necessary changes to the copy constructor.

Next, we need to add these light and material properties to our graphical applications.

Exercise 3

a) Open the `App1.cfg` of our first graphical application. It renders the scene described in `simpleScene.sdl`.

b) Open the file `simpleScene.sdl` and add the line

```
light -10 10 10  1 1 1
```

after the first line.This line specifies that there is one light source in the scene located at $(-10, 10, 10)$ and which has a colour $(1, 1, 1)$.

c) Replace the colour specification for the square by

```
diffuse 0 1 0
ambient 0 0.2 0
```

and for the sphere by

```
diffuse 0 0 1
ambient 0 0 0.2
```

c) Open the `App2.cfg` of our second graphical application. It renders the scene described in `buckyball.sdl`.

d) Open the file `buckyball.sdl`, add the line

```
light -10 10 10  0.8 0.8 0.8
```

after the first line and replace the colour specification for the mesh by

```
diffuse 1 1 0
ambient 0.2 0.2 0
```

Now that we have added the light and material properties to our sdl files, we also need to update the `SceneFactory` class and the `Token` enum so that these data are properly parsed and stored.

Exercise 4

a) Open the `Token` enum, remove the `COLOUR` token and add three new tokens: `LIGHT`, `DIFFUSE` and `AMBIENT`.

b) Open the `SceneFactory` class and add a local variable `lights` in the `createScene` method immediately after declaring the variable `objects`. This variable is an ArrayList of `Light` objects which should be declared and initialized in one line.

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

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

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

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

d) Add a line of code so that the ArrayList of `Light` objects is properly set to the `Scene` object before this object is returned by the `createScene` method.

The light properties will be parsed and stored properly but we still need to do the same for the new material properties.

e) Adapt the method `processMaterial` so that it recognizes the two new tokens which replace the old `COLOUR` token.

Exercise 5

The light and material properties are added to the sdl files, properly parsed and stored in the right classes. The only task left is to adapt our ray tracer so that it makes use of this information to render the 3D objects in the scene.

a) Open the `RayTracer` class in the `renderer.raytracer` package.

Remember that we added the method `shadeHit` to this class (see Lab 5). Its current implementation simply returns the colour of the 3D object without taking any light source into account.

b) Change the implementation of the `shadeHit` method in the `RayTracer` class so that it returns the colour computed by the shading model discussed in the slides of this Lab.

Time to check your work!

Exercise 6

a) Run `App1`. If your implementation is correct, you get a nicely shaded blue sphere and a green square on your screen.

b) Move the camera forward so that you can clearly see the smooth shading of the sphere. Do not close the window.

c) Change the ambient colour of the sphere to $(0, 0, 0.5)$. How will this influence the image?

d) Verify your answer to the previous question by running `App1` again. Compare this new image with the previously rendered image.

e) Run `App2`. You should get a nicely shaded yellow bucketball on your screen. Do not close the window.

f) Take a closer look at the shading by moving the camera forward. Is your system able to animate the camera in realtime?

g) Add a second light source at position $(10, -10, 10)$ with colour $(0.2, 0.2, 0.2)$ to the scene. How will this influence the image?

h) Verify your answer to the previous question by running `App2` again. Compare this new image with the previously rendered image.

Exercise 7

In this exercise we will create a third graphical application which will reveal a bug in our rendering framework.

a) Download the "cylinder1.txt" en "cylinder2.txt" files and put them in the `resources` folder.

b) Make a copy of the `apps.app2` package and rename it to `apps.app3`. Rename the two files in this new package as well (`app3.java` and `app3.cfg`).

c) Make sure that `app3.java` uses the correct configuration file.

d) Open `app3.cfg` and change the value of the key `scene.file` to "resources/cylinder.sdl". This .sdl file does not exist yet, so let's create it.

e) Create a new "cylinder.sdl" file in the `resources` folder. This file should specify a scene with

   - a black background,
   - one light source at position $(-5, -5, 2)$ and colour $(0.8, 0.8, 0.8)$,
   - one polygonal mesh described in the file "cylinder1.txt" (in the `resources` folder) which should have a green colour (diffuse: $(0, 1, 0)$ and ambient: $(0, 0.2, 0)$).

The file "cylinder1.txt" contains the polygonal mesh data of a **generic cylinder**. A generic cylinder is a cylinder whose axis coincides with the z-axis and that has a circular cross section of radius 1 and extends in z from 0 to 1.

f) Which side of the cylinder will you see on the screen when you run `App3`?

g) Verify your result by running `App3`.

h) Change the eye and the look point of the camera to $(0, -5, 0.5)$ and $(0, 0, 0.5)$, respectively and change the up vector to $(0, 0, 1)$. How will these changes influence the image on your screen?

i) Verify your result by running `App3` again. Do not close the window.

j) Change the `cylinder.sdl` file so that the scene contains the mesh described in "cylinder2.txt".

k) Run `App3` again and compare the resulting image with the previously rendered image.

Both "cylinder1.txt" and "cylinder2.txt" contain a polygonal mesh approximation for a generic cylinder. But it is clear that our ray tracer does not render the polygonal mesh stored in "cylinder2.txt" correctly. Both polygonal meshes have the same vertices and faces but they have different normal vectors. The normal vectors associated with the vertices of each face in "cylinder1.txt" are equal to the face plane normal while the normal vectors associated with the vertices of some faces in "cylinder2.txt" are perpendicular to the true surface and hence, not equal to the face plane normal.

Note that "buckyball.txt" also contains a polygonal mesh in which the normal vectors associated with the vertices of each face are equal to the face plane normal. So it seems that our ray tracer does not render polygonal meshes correctly if normal vectors are associated with the vertices of (some) faces which are not equal to the face plane normal.

l) Can you guess where this bug comes from?

Exercise 8

In Lab 5, we implemented the `intersection` method in the `Mesh` class to allow polygonal meshes to be rendered by our ray tracer. Remember that we took the normal vector associated with one vertex instead of the normal vector to the face plane to compute the intersection between a given ray and the face plane. This is only valid when the normal vectors associated with the vertices of this face are all equal to the normal vector to the face plane. This is the case if the face is an exact representation of (a part of) the surface of the 3D object. However, when the normal vectors associated with the vertices

of a face are not equal to each other, the face is only an approximation to (a part of) the surface of the 3D object. In this case, one needs to compute the face plane normal and use this in the `intersection` method of the `Mesh` class.

a) Open the `Face` class and add two public instance variables:
   - a boolean `oneNormal` indicating whether all vertices of this face have the same normal,
   - a `Vector facePlaneNormal` storing the coordinates of the normal vector to the face plane.

b) Adapt the `readFile` method of the `Mesh` class so that it checks the indices of the normal vectors associated with the vertices of each face. If they are the same, the instance variable `oneNormal` of the face should be set to true and the `facePlaneNormal` should store the normal vector of one of the vertices. If they are not the same, the instance variable `oneNormal` of the face should be set to false and the `facePlaneNormal` should store the normal vector to the face plane. See Exercise 3 of Lab 1 for a method to compute this `facePlaneNormal`.

c) Adapt the `intersection` method of the `Mesh` class so that it uses this `facePlaneNormal` when computing the intersection between a ray and a face.

d) Run `App3` again and make sure that "cylinder2.txt" is now rendered correctly.

Notice that the polygonal mesh used to approximate the generic cylinder is clearly visible in the rendered image.

The normal vectors associated with the vertices of each face in "cylinder1.txt" are equal to the face plane normal. This indicates that the polygonal mesh is an exact representation of the 3D object. In this case, the faceted appearance of the object in the final image is what we want. However, the vertices of each face in "cylinder2.txt" are perpendicular to the true surface and hence, are not always equal to the face plane normal. This indicates that the polygonal mesh is an approximation to a curved surface, and so the faceted appearance is not the desired result. The last exercise of this Lab will solve this issue.

Exercise 9

a) Adapt the `intersection` method of the `Mesh` class so that it computes the hitNormal correctly. If a ray hits a face which has only one normal,

the normal vector of this face should be used as hitNormal (flat shading). If a ray hits a face which has more than one normal vector associated with it, the hitNormal should be computed by interpolating the normal vectors associated with its vertices. The latter is called Phong shading which is an example of smooth shading. See the slides of this Lab for more information.

b) Run `App3` again and make sure that "cylinder2.txt" is shown as a smooth cylinder on your screen.