Lab 3: 3D Viewing

3D Computer Graphics

Introduction

Import the archive file `3DCG_Lab3.jar` into Eclipse by selecting

```
 File > Import... > General > Existing Projects into Workspace
             > Next > Select archive file > Finish
```

If you have set up JOGL correctly in Lab 1, you can simply add the user library jogl-2.0 to this project as follows: right-mouse click on the project's name in the Package Explorer window and select

```
Build Path > Add Libraries ... > User Library > jogl-2.0.
```

Exercise 1

First, we solve the compile errors in the loaded project.

a) Copy your `Face` and `Mesh` class from Lab 2 to the package `geomobj/mesh` in the current project.

b) Copy your `Point` and `Vector` class from Lab 2 to the package `util` in the current project.

c) Solve the remaining compile errors by adding the correct import statements.

This project contains one graphical application with `app1.cfg` as configuration file. Open `app1.cfg` in the `apps.app1` package. You will notice that the this configuration file consists of key/value pairs. It is very easy to read (and edit) this file. It describes

- the name of the file containing the polygonal mesh which will be rendered by this graphical application,
- the properties of the camera (location and orientation).

It is clear from the first line of this configuration file that the first application will render the wineglass.

The purpose of the `apps` package is to collect the data of all graphical applications. The `App` class in the `apps` package is the heart of every graphical application. By putting all the common features in one class, it becomes very straightforward to create new graphical applications. Convince yourself of this statement by looking at the provided example: the class `App1` in the package `apps.app1`. It simply calls the constructor of the `App` class and passes the appropriate configuration file.

When looking at the constructor of the `App` class, one notices that it uses an `AppPropertiesLoader` class — which you do not have to study for now — to read in the configuration file of a particular graphical application and to get an object of the `java.util.Properties` class (`prop`) which stores all the key/value pairs mentioned in the configuration file. This `Properties` object will be passed on to all classes which require the configuration settings. For example, on the sixth line of code in the constructor of the `App` class, a new `Camera` object is created by calling its constructor and passing this `Properties` object. By looking at the constructor of the `Camera` class in the `renderer` package, one notices that the relevant configuration settings are read out of the `Properties` object and stored.

d) Run `App1` and make sure you get the wineglass nicely centered on your screen.

The image also shows a small yellowish sphere which is added automatically (by the `App` class). It shows the location of the light source in the scene and serves as visual guidance.

Exercise 2

a) Change the value of `camera.eye.z` to 4 in the configuration file of the first application. How will this influence the image on the screen? Run `App1` again to verify your answer.

b) Set the value of `camera.eye.z` to 6 again and change the value of `camera.look.x` to 1.5. What result do you expect? Verify your answer by running `App1` again.

c) Set the value of `camera.look.x` back to 0 and change the value of `camera.up.y` to -1. What result do you expect? Verify your answer by running `App1` again.

d) Set the value of `camera.up.y` back to 1.

Now that we have experimented a little with manually setting the position and orientation of the camera, we will start to implement the features which will allow to animate the camera.

Exercise 3

Look again at the `Camera` class. It contains instance variables u, v and n, three unit vectors which together, represent a coordinate system attached to the camera. Note, however, that these vectors are not initialized at present.

a) Initialize u, v and n in the constructor of the `Camera` class based on the given `eye`, `look` and `up`.

Subsequently, we implement forward and backward camera movement.

b) Add two public methods to the `Camera` class which have no parameters and return void. Name these methods `forward` and `backward` and implement both of them. (One line of code should suffice for each implementation!)

We will add some more functionality in the next two exercises which will allow us to test these methods.

Exercise 4

We want our graphical applications to respond to key actions: when the user presses the “f” or “b” key, the camera should move forward or backward, respectively. Hereto, we need a class which listens to such key events.

a) Create a new class `UserEventMediator` in the `ui` package. This class should extend the `KeyAdapter` class which Java provides to listen to keyboard events.

b) Add a private instance variable `camera` to the `UserEventMediator` class. This variable will hold the `Camera` object which will be animated by keyboard events.

c) Create a constructor which takes a `Camera` object as parameter. This `Camera` object should be used to initialize the instance variable `camera`.

d) Override the method `keyPressed` and give it the following implementation:

```
if (e.getKeyChar() == ’f’){
  camera.forward();
} else if (e.getKeyChar() == ’b’){
  camera.backward();
}
```

It should be clear that this implementation makes sure that the camera moves in the expected way when the user presses the “f” or “b” key.

Exercise 5

Next, we need to make sure our graphical applications make use of this `UserEventMediator` class to listen to keyboard events.

a) Open the `App` class. Add some code to the constructor of this class immediately below the point where a GLEventListener is added to the canvas:

   – Create a new `UserEventMediator` object.
   – Register this object as a key listener of the canvas.

b) Run `App1` again. Press the “f” and “b” keys and check whether the camera moves forward and backward as expected. Make sure it works before you continue.

At this point, we have basic support for moving the camera forward and backward. But we want more: we want to extend this functionality so that the camera can be rotated as well. It turns out that 3D rotations can be carried out in an elegant way by using quaternions. This is the subject of the remaining exercises of this lab.

Exercise 6

First, we add support for quaternions to our rendering framework. Look at the slides of this Lab for the appropriate formulas.

a) Create a new class `Quaternion` in the `util` package.

b) Add four public instance variables a, b, c and d which define the quaternion.

c) Add a constructor which has four floating point values as parameters.

d) Add a constructor which creates a pure imaginary quaternion based on the `Point` object given as parameter.

e) Add a constructor which creates a rotation quaternion based on two given parameters: an angle specified in degrees and a `Vector` object r indicating the direction of the rotation axis. This constructor may assume r has unit length.

f) Add a method `mult` which computes and returns the product of this quaternion with the quaternion given as parameter to this method.

g) Add a method `conjugate` which computes and returns the conjugate of this quaternion.

Exercise 7

a) Add a method

```
public void up(){
  // todo: implement
}
```

to the `Camera` class and provide an implementation which rotates the camera by an angle 0.5 in upwards direction. Note that this method should update both the `eye`, v and n as explained in the slides of this Lab.

b) Change the `keyPressed` method in the `UserEventMediator` class so that the camera will rotate upwards by pressing the up arrow key.

c) Run `App1` again and test your implementation by using the up arrow key.

Exercise 8

a) Provide methods `down`, `left` and `right` which rotate the camera respectively downwards, to the left and to the right.

b) Choose respectively the down arrow key, the left arrow key and the right arrow key to carry out these camera movements.

c) Run `App1` again and test your implementations.

Exercise 9

In this last exercise, you will combine your self-created polyhedron of Lab 2 with the camera animation implemented in this Lab.

a) Add your `polyhedron.txt` file of Lab 2 to the `resources` map of your current project.

b) Create a new package `app2` in the `apps` package.

c) Copy the two files in the package `app1` to the package `app2`.

d) Rename both files in package `app2` to make clear they are part of the second application.

e) Change the class `App2` so that the correct configuration file is given to the `App` class.

f) Change the `app2.cfg` file so that the correct 3D object file is used.

g) Run `App2` and play with the “f”, “b”, and arrow keys to test your work.

Good luck!