Lab 7: 3D Transformations

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 6. However, it is strongly advised that you make a new fresh copy of the project of Lab 6 and rename this copy to 3DCG_Lab7. 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.

The main goal of this Lab is to add support for applying one transformation (a rotation, scaling or translation) to one object in the scene.

Exercise 1

We start by adapting our first graphical application so that it can be used to test our — yet to be implemented — support for 3D transformations.

a) Open the file `app1.cfg`. The first line of this file tells us that the first application renders the scene described in the `simpleScene.sdl` file.

b) Open this sdl file and make the following changes:

   - Remove the three lines related to rendering the square.
   - Add the line

     ```
     scale 2 1 1
     ```

     just before the instruction to create a `Sphere` object.

   In this way, we restrict ourselves — for now — to applying one transformation (a scaling operation with a factor 2 in the x-direction) to one object (a sphere).

Let's now make all the necessary changes to our rendering framework to render this scene.

Exercise 2

Put the given `Matrix.java` file in your `util` package. This class represents a 4×4-matrix. Note that such a matrix is represented as a two-dimensional array of floats internally.

a) Study the provided implementation of the `Matrix` class.

b) How does the default constructor of the `Matrix` class initialize a matrix?

c) Implement the two unfinished methods in `Matrix.java`.

Exercise 3

a) Make a new package `transfo` in the `src` folder. All classes related to 3D transformations will be put in this package.

b) Put the given `Transfo.java` file in this package. This class represents a geometric transformation.

c) Study the provided implementation of `Transfo.java`. Make sure you know the meaning of its two instance variables.

All 3D transformations supported by our rendering framework will be subclasses of the `Transfo` class. We start by adding support for scaling operations in the next exercise.

Exercise 4

a) Implement a new subclass `ScaleTransfo` of `Transfo`.

b) The `ScaleTransfo` class should only have a constructor which takes three floating point values sx, sy and sz as arguments and correctly initializes its inherited instance variables `mat` and `invMat`.

Our aim is to add support for rotation and translation operations as well, but we will postpone this to a later exercise. First, we will focus on making all the necessary changes to our rendering framework to support scaling operations only. In this way, we can test our code as quickly as possible.

Exercise 5

We developed the basic building blocks to support (scaling) transformations in the previous exercises. We will now integrate them in our rendering framework. First, we adapt the `Shape` class.

a) Add a public instance variable `transfo` to the `Shape` class which holds the transformation which has to be used to resize (or position and/or orient) this shape in the scene.

b) Add a line to the default constructor which initializes this instance variable by making use of the default constructor of the `Transfo` class.

c) Which transformation is initially set to a `Shape` object?

d) Optionally, add a setter to this class to set `transfo`.

Next, we have to make sure that the instance variable `transfo` of each `Shape` object is correctly set based on the information given in the sdl file describing the scene.

Exercise 6

a) Add the token `SCALE` to `Token.java`.

b) Add a private instance variable `currTransfo` (of the type `Transfo`) to the `SceneFactory` class. This instance variable keeps track of the current transformation while scanning the sdl file (similar to `currMtrl` which keeps track of the current material while scanning the sdl file).

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

```
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
}
```

In orde to correctly handle tokens related to 3D transformations, 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)){
  if(!processTransfo(token, scanner)){
    // add object
  }
}
```

If the token given to the `processTransfo` method is a transformation token (currently only `scale`), this method will use the given `Scanner` object to read the value(s) of this token and correctly set the `currTransfo` variable. True is returned in this case. If the token given to the `processTranfo` method is not a transformation token, false is returned.

d) Implement the `processTransfo` method in the `SceneFactory` class.

Finally, we need to make sure that a copy of this current transfo object is set to a shape when our parser encounters a shape token.

e) Just before returning the `Shape` object in the `createShape` method of the `ShapeFactory` class, set its transformation correctly to a copy of `currTransfo`.

At this point, we have added support for parsing scaling data in the sdl file and for storing these data in the `transfo` object of the shape which has to be scaled.

Next, we have to adapt the `intersection` method of all currently supported `Shape` objects (`Sphere`, `Square` and `Mesh`) so that our raytracer will effectively use this scaling data to render a scaled version of the corresponding shape. Again, because we want to be able to run `App1` and test our code as fast as possible, we will focus first on adapting the `intersection` method of the `Sphere` class only.

Exercise 7

Instead of trying to compute the intersection of a ray with a transformed generic object, it was explained in the slides of this Lab that it suffices to intersect the inverse transformed ray with the generic object. In this exercise, you will implement the calculation of the inverse transformed ray.

a) Add and implement the following method in the `Ray` class.

```
public Ray getInvTransfoRay(Matrix invMat){
  // todo
}
```

This method should return a new `Ray` object which is the inverse transformed ray of itself. The inverse transformation which has to be applied is given as a parameter to this method.

Exercise 8

In Lab 4, you implemented the `intersection` method in the `Sphere` class. More specifically, you implemented the algorithm to compute the hitpoint(s) of a generic sphere and the given ray. Note, however, that the `Sphere` class extends the `Shape` class which has a `Transfo` object. Your current implementation of the `intersection` method completely ignores this `Transfo` object, and hence, ignores the transformation instructions given in the sdl file. You should change the implementation of the `intersection` method in the `Sphere` class so that it computes the intersection between the given ray and the transformed generic sphere.

But again, instead of trying to compute the intersection of a ray with a transformed generic object, it suffices to intersect an inverse transformed ray with the generic object.

This means you can keep most of your implementation of the `intersection` method but instead of applying the calculations on the given `Ray` object, you should apply them on the inverse transformed ray.

a) Create the inverse transformed ray of the given `Ray` object at the beginning of the `intersection` method of the `Sphere` class. (Use your solution to Exercise 7.)

b) Refactor your implementation of this `intersection` method so that the calculations are performed on this inverse transformed ray (and the generic sphere). Look at the slides of this Lab for more information.

Note that the hitNormal should also change. Indeed, you created a hitNormal to the generic sphere in Lab 4, but when you take transformations into account, it should be the normal vector to the transformed generic sphere.

c) Refactor your code so that it correctly computes the hitNormal corresponding to the hitpoints of a ray and a transformed generic sphere. More information can be found in the slides of this Lab.

Exercise 9

High time to check your solutions to the previous exercises.

a) Run `App1` and make sure you get a scaled version of a sphere as expected.

b) Change the scaling instruction in the sdl file to

```
scale 0.5 3 1
```

How will this influence the rendered image?

c) Run `App1` and check whether you get the expected result.

Exercise 10

a) Add and implement a new class `TranslateTransfo` which extends the `Transfo` class and represents a 3D translation. This class should only have a constructor which takes three floating point values tx, ty and tz as arguments and correctly initializes its inherited instance variables `mat` and `invMat`.

b) Similarly, add and implement a new class `RotateTransfo` which extends the `Transfo` class and represents a 3D rotation. This class should only have a constructor which has four floating point values defining the rotation (angle, x, y and z) as arguments. The angle is specified in degrees and x, y and z are the coordinates of the vector indicating the rotation axis. Note that this vector should be normalized. As it cannot be expected that the user will always provide a normalized vector, you should normalize these values in the constructor of the `RotateTransfo` class.

c) Make all the necessary changes to the rendering framework to support these two new transformations, similar as you have done for scaling transformations.

d) Test your work by adapting the transformation instruction in the file `simpleScene.sdl` and running `App1` again.

Exercise 11

Currently, your support for rotation, scaling and translation operations is only applied to `Sphere` objects.

a) Change the `intersection` method of the `Square` class — similar to your changes made to the `intersection` method in the `Sphere` class — so that you can also apply 3D transformations to `Square` objects. Make sure you understand the current implementation of the `intersection` method of the `Square` class before making any changes.

b) Change the `intersection` method of the `Mesh` class — similar to your changes made to the `intersection` method in the `Sphere` class — so that you can also apply 3D transformations to `Mesh` objects.

c) Test your work by adapting the shape instruction in the `simpleScene.sdl` file and running `App1` again.

After finishing the above exercises, your rendering framework supports applying one transformation (a scaling, rotation or translation) to one object (a sphere, square or mesh) in the scene. Next week, we will extend this idea so that we can apply multiple transformations to several objects in the scene.

To be continued ...