CS 465, Fall 2003, HW 3b

Andy Scukanec

November 9, 2003

1 Introduction

In this assignment, your task will be to modify a raytracer so that it can handle basic transformations on objects, and so that it can handle groups of objects. Although you will be building functionality into a ray tracer, this assignment will focus more heavily on the math underlying the transformations than on the actual ray tracing algorithm itself.

2 Requirements

As before, your ray tracer will read in an input file and generate an image based on the ray tracing algorithm covered in class. The final ray tracer should be able to scale, rotate (about an arbitrary axis), and translate any of the surfaces in the scene. The Transformation class will be the key component of this new functionality. There are some other places in which changes must be made though, in order to use the transformations.

Each Surface is now given a Transformation. The parser will call the appropriate methods as it encounters Transformations associated with a particular Surface. It is up to you to implement the methods that the parser will be calling. All of these methods should be very short after writing the Transformation class.

Finally, you are asked to implement Groups, a new kind of Surface which is really just a collection of Surfaces. The interesting thing here, is that objects can be grouped, transformed as a group, and then put into more groups. This creates a potentially very expressive way of describing a scene.

The full list of changes is:

- Implement the following methods in Transformation:
  - `void leftProduct(Transformation input, Transformation output)` This method should compute the result of `input*this` and store the result in `output`. `output` will never be `null`, but it might be equal to `this`.
  - `void rightProduct(Transformation input, Transformation output)` This method should compute the result of `this*input` and store the result in `output`. `output` will never be `null`, but it might be equal to `this`.
  - `void transformVector(Vector3d input, Vector3d output)` This method should compute the result of transforming the *direction* reprsented by `input` and store the result in `output`. `output` will never be null, but it might be equal to `input`.

  - `void transformPoint(Vector3d input, Vector3d output)` This method should compute the result of transforming the *location* reprsented by `input` and store the result in `output`. `output` will never be null, but it might be equal to `input`.
  - `void transposeTransformVector(Vector3d input, Vector3d output)` This method should compute the result of transforming the *direction* reprsented by `input` by the transpose of this transformation, and store the result in `output`. `output` will never be null, but it might be equal to `input`.
  - `Transformation newRotation(Vector3d axis, double angle, Transformation output)` This method should create a rotation of `angle` degrees about `axis`, and return it. If `output` is not `null`, the resulting transformation should be stored in `output` too.

- Write the `scaleBy(Vector3d scale)`, `translateBy(Vector3d translate)`, and `rotateBy(Vector3d axis, double angle)` methods in `Surface`. These methods should simply modify the existing transformation for the current surface by creating the appropriate transformation, and then left multiplying it onto the current transformation.

- Write the `Surface.intersects(Ray incoming, IntersectionRecord record)` method. This method should take the incoming ray, transform it into object space, then call the abstract method `Surface.intersectsUntransformed(Ray incoming, IntersectionRecord record)`. After that, the method should take the intersection location and normal, and transform them from object space into world space.

- Write the `Group.intersects(Ray incoming, IntersectionRecord record, Scene scene)` method. Intersecting with a group is the same as intersecting with all of the surfaces within the group. This method should look a lot like your `Scene.trace(Ray)` method, but without the shading calculations.

3 Framework

As before, we will be giving you a framework within which you can write your solution code. The framework we will give you contains the solution code for homework 3b. We recommend that you do this project starting from our framework, although in the spirit of having your own raytracer, you are welcome to build from your own code base. We will be posting a test scene later that should help to find some rather subtle bugs in your raytracer. If you are going to use your own code, please check your raytracer against this scene!

3.1 ViewingFrame

There are a few updates to the ViewingFrame class. You can now turn the preview render on or off, and a few troublesome lines of code were removed. Those odd crashes on startup shouldn't occur any more.

3.2 Cube

We've given you a new surface to work with - cube. The cube has no parameters other than a material. The basic cube is centered around the origin, and its sides are 2 units by 2 units large. Changing the cube's size, location, etc. is only possible through transformations!

3.3 Parser

The parser has changed quite a lot, although it is fully backwards compatible. The parser can now handle comments, which are lines that start with the '#' character, it is much more forgiving on whitespace, and the errors presented to you should be a lot more clear.

The biggest change under the hood is that when setting the value of some member, the parser actually calls a modifier method. Because of the way in which we do this, the modifier methods *must* keep to a strict naming convention. As an example, lets look at the process by which the Parser sets a sphere's radius.

```
...
Surface Sphere
ref material redPhong
Vector3d center [0 0 0]
double radius 4.3
...
```

When the parser encounters the last line of the Sphere declaration, the Parser will look for a method called `setRadius(double)`. If no method exists, then the Parser will look for a field called `radius`. If it cannot find such a field, then the Parser will throw an error. We made this change so that setting instance members would allow for the programmer to check for erroneous values, take any necessary ancillary actions, etc. For example, if someone tried to set the value of a Sphere's radius to -5.6, then the programmer could reject the new value, throw an error, etc.

There is a new Surface in town called a Group, and its presence requires certain changes in the parser. If the first line of a surface declaration in the scene file starts with the '*' character, the object will be created, named, etc., but it will *not* be added to the Scene's full list of objects. This is so that when writing a scene, you can distinguish between objects that are just being defined for later use, and objects that are simultaneously being defined and put into the scene.

The last change to the parser is its new ability to handle Transformations. Transformations must be formatted as below, where `<type>` is either "scale", "translate", or "rotate". If the "scale" or "translate" transformation are used, then there is only one argument, and it is the vector that describes the scale or translate. The "rotate" transformation has two arguments - the vector that represents the axis about which the rotation will occur, and a number that represents how far around the axis to rotate.

```
Transformation <type> <arg_1> ... <arg_N>
```

Here is a simple example that adds a cube into the scene, rotates it 45 degrees about the axis $[1, 1, 1]^T$, scales it by a factor of 2 in all directions, and finally translates it by 3 units in the +x direction. Please note that the order in which the transformations are listed is the order in which they are applied!

```
...
Surface Cube cube1
ref material green
Transform rotate [1 1 1] 45.0
Transform scale [2 2 2]
Transform translate [3 0 0]
...
```

4 Other Details

4.1 Helpful Hints

Some comments on the various function you will have to write for the Transformation class.

- All of the methods for the Transformation class have an output parameter. If this parameter is not `null`, the result of the operation should be stored in this parameter. If this parameter is `null` and the method has a return value, no changes should be made to it, but rather a new instance should be returned. It is an error to pass `null` as an output parameter to a function that has a `void` return value.
- All of the methods for the Transformation class should be written with the possibility in mind that the output parameter could very well reference `this` object.
- The difference between `transformPoint` and `transformVector` is that `transformPoint` assumes the vector passed in represents point in homogenous coordinates; that its fourth component is 1. `transformVector` assumes the vector passed in represents a direction and that its fourth component is 0.
- `transposeTransformNormal` should transform the incoming vector by the *transpose* of this transformation. Since the input is a direction, the incoming vector should be treated as if its fourth component is 0. This method will be most useful when trying to transform the normal returned to you in an intersection record.
- You might find the `setValues(double[][] newValues)` and `setToIdentity()` methods of the Transformation class useful.
- Inverse is written for you! No need to figure it out on your own!
- Degrees vs. Radians: The Parser will call `Sufrace.rotateBy()` and give it exactly the number found in the file. This number should be in degrees since it is more human readable. The `Transformation` methods will expect angles to be specified in radians. It is up to the `Surface.rotateBy()` method to do the conversion.

4.2 Recap

Implement the following new methods:

- `void leftProduct(Transformation input, Transformation output)`
- `void rightProduct(Transformation input, Transformation output)`
- `void transformVector(Vector3d input, Vector3d output)`
- `void transformPoint(Vector3d input, Vector3d output)`
- `void transposeTransformVector(Vector3d input, Vector3d output)`
- `Transformation newRotation(Vector3d axis, double angle, Transformation output)`
- `Surface.intersects(Ray incoming, IntersectionRecord record, Scene scene)`
- `Surface.scaleBy(Vector3d scale)`

- `Surface.translateBy(Vector3d translate)`
- `Surface.rotateBy(Vector3d axis, double angle)`
- `Group.intersects(Ray incoming, IntersectionRecord record, Scene scene)`

Test thoroughly!

4.3 Submission

You should submit through CMS as with the other coding assignments thus far. Zip up all of your .java files *and* your extra credit scene files as well. Submit them through CMS. Do ***NOT*** submit .class files. JAR files are fine too if you know how to create them.

4.4 FAQ Page

We will be maintaining a FAQ page linked from the assignment page. If you have any questions about how to use the framework code, what we intended something to mean, etc., please check the FAQ page and newsgroups first. If your question is still not answered, then email the course staff at cs465-staff@cs.cornell.edu. We will *not* be posting to notify you that we have updated the FAQ page because we anticipate that this will happen frequently.

If you have any questions regarding the use of lab equipment, a supported IDE, or anything ancillary that does not directly involve this assignment, then please talk to either Andrew Butts or Pet Chean.