CS61B, Fall 2008Project #2: Spatial Database (Version 4.2, revised 10/24/2009)P. N. Hilfinger

Due: 2 November 2009

1 Background

For this second project, we consider the problem of organizing a collection of data about objects in space. Imagine that you are given a large collection of things—people, say—each of which has some location at any given time. We might then imagine making *queries* about this collection, such as "Where is so-and-so?" or "Who is currently within 100 yards of location such-and-such?" or "What pairs of people are within 50 yards of each other?" If, in addition, we give each of these objects a velocity, their relationships will change from moment to moment, as will the answers to these and other queries.

Any one of the sample queries above could be answered by simply searching all objects or (in the last case) all pairs of objects. However, if we want to our system to handle large collections of data, it would be nice to narrow down the set of objects or pairs we must consider. In the case of geographical data, one way to do this is a data structure known as a *quadtree,* described in §6.3 of *Data Structures (Into Java)* (for three-dimensional data, there is an analogous structure known as an *octtree.*) This is a recursive structure that divides the set of data into four quadrants by position, and then repeatedly subdivides the quadrants as necessary to get down to subdivisions that contain just one (or at least some small number) of items. With this, it is relatively easy to answer "who is within distance d of point x." Yes, we're jumping ahead a little here, but the data-structuring idea is not that hard.

In this project, we'll consider just such a structure for representing a large set of moving billiard-ball like objects, answering the queries listed above about them, and tracking their movements as they bounce off each other and off a set of walls surrounding them. Your solution will consist not just of a main program (implementing a simple textual command processor), but also of a specific library data structure that can be used by other main programs. That is, you'll be fulfilling an *Application Programming Interface (API).* We'll be testing *both* your main program and API implementation separately. Therefore, if you violate the boundary between these two—making your command processor depend on some detail of your data structure other than the public interface, or making your data structure assume certain things about the command processor that uses it—your program will fail our tests (it might not even compile).

2 Running the program

The main program you will write is called `track`. You invoke it from a command line like this (things in square braces are optional):

```
java track [ --debug=N ] [ INPUTFILE [ OUTPUTFILE ] ]
```

This runs the program taking input from the file *INPUTFILE,* or, if this is defaulted, from the standard input, writing the results to OUTPUTFILE, or if that is defaulted, to the standard output. The `--debug` switch, if present, turns on any diagnostic printing you want to do (N indicates what level of output you want; interpret it as you choose). We will test that your program does not blow up if we use it, but will ignore the output and won't dictate what N means. We suggest that if you want to put in statements to print things while debugging your program, you arrange for the printing to happen *only* when this switch is present. Your program is wrong if it produces extraneous debugging output with the switch absent.

3 Commands

The `track` program should accept commands in free format, separated by semicolons or ends of lines. Other whitespace is ignored except to separate words and numbers. When input is from the standard input, print a prompt ('`>␣`') at the beginning and after each newline (except maybe the last). Floating-point numbers (approximations to real numbers) called for in these commands should assume at least the precision and range of the Java type `double`. The commands are as follows:

(empty) An empty (blank) command does nothing. These occur whenever the user leaves a blank line or redundantly ends a line with a semicolon, since both semicolons and ends of lines separate commands.

# *Comment* A comment, ending at the end of this line. Comments are ignored.

bounds x_{low} y_{low} x_{high} y_{high} Set the positions of the four walls that enclose the objects to be tracked by specifying the lower-left and upper-right corners of a rectangle aligned with the x and y coordinate axes. Initially, the bounds are $(0, 0)$ and $(0, 0)$ (a box with no area). For simplicity, the walls may only be moved outward.

add *ID* x y v_x v_y Add a new object whose center is initially at position (x, y) and moving with velocity (v_x, v_y) (so that, after a time Δt has passed, its center has moved by $v_x \Delta t$ in the x direction, and $v_y \Delta t$ in the y direction). Label this object with the non-negative integer *ID*. It is an error to add an object that is outside the walls or closer than the current radius to any wall. It is an error to enter two objects with the same *ID*, or two objects that are closer than twice the current radius from each other.

rad r Set the radii of all the objects being tracked to $r \geq 0$. Initially, all have "infinite" radius, so that they will not fit in any set of walls. Thus, you must use this command before doing any adds. For simplicity, it is illegal to increase the radius.

load *filename* Read commands from *filename,* executing them as if they had been written in place of the `load` command itself. A file name is any sequence of non-whitespace characters. Commands and comments may not be split between files; a command or comment may not start in file *filename* and then end in the current command file.

write *filename* Write the current state of the system to the file named *filename* as a sequence of a `bounds` and a `rad` command, followed by `add` commands. Print each command on a separate line and output `add` commands in order of increasing *ID.* The idea is that reading such a file would reproduce the state of the system.

near x y d Print the positions and velocities of all objects whose centers are within distance d of (x, y). Print two points per line (except possibly the last line) in the format *ID:*(x, y, v_x, v_y), separated by white space, ordered by ascending *ID*. Print four significant digits for each number (`%g` format).

near x $*$ d Print the positions and velocities of all objects whose x coordinate is within d units of x, using the same format as the first variety of the `near` command, above.

near $*$ y d Print the positions and velocities of all objects whose y coordinate is within d units of y, using the same format as above.

closer-than d Print all pairs of distinct objects whose centers are within distance d of each other in the format:

$$ID_1\texttt{:(}x_1, y_1, v_{x1}, v_{y1}\texttt{)}\ \ ID_2\texttt{:(}x_2, y_2, v_{x1}, v_{y1}\texttt{)}$$

where $ID_1 < ID_2$, one pair per line on the standard output. Print each pair exactly once, in increasing order by ID_1 and, for pairs with the same ID_1, in increasing order by ID_2. As for the 'near' command, print four significant digits for each number using `%g` format.

simulate t Where $t \geq 0$ is a floating-point number. This performs a simulation to determine where all the objects will be in t seconds, accounting for all collisions of objects with walls and each other and updating their positions and velocities accordingly. (For you physics majors, the collisions are elastic.) This command does not print anything.

quit Exit the program with no further output. Do exactly the same thing when you reach the end of all input to the program.

help Prints a helpful summary of the possible commands.

The user may abbreviate any command name with any unique prefix of that name. Thus, `c` or `cl` are legal in lieu of `closer-than`.

4 Algorithms

The real problems come with the `closer-than` and `simulate` commands. Naive approaches to these problems involve checking all pairs of objects for their separation (`closer-than`) or to see whether they collide (`simulate`), $\Omega(N^2)$ operations, where N is the number of objects. We're going to try to do better. First, we'll use a quadtree data structure to find nearby objects quickly.

Then, we'll handle simulation by breaking the total time t into shorter periods so that we only have to consider collisions between a limited number of nearby objects during any period. That cuts the time for computing collisions from N^2 to KN for some constant K. That is, for simulation we'll use the following strategy:

1. Repeat until $t = 0$:

   (a) Find v_m, the maximum value of $|\vec{v}_i|$, where $\vec{v}_i$ is the velocity of particle i.

(b) From v_m compute a time interval, $\Delta t \leq t$, during which no object moves more than some fixed distance—let's say D, (you might, for example, choose $D = 2r$, where r is the radius of each object). That is, $\Delta t = \min(D/v_m, t)$.

(c) Now we know that during the period of time Δt, any object may collide *only* with objects whose centers are $\leq 2(D + r)$ away. (Imagine two objects moving toward each other along the x-axis, both at v_m. They will collide when they are $2r$ apart, and each moves $\leq D$ in Δt seconds, which adds up to $2(D + r)$.)

(d) So we find all pairs of objects that are within $2(D + r)$ and check *only* these pairs to see if they collide.

(e) We also check for collisions with walls for objects that are $\leq D + r$ from a wall.

(f) Find the shortest time interval, t_c to a collision, amongst all of these.

(g) Now we move all objects to their new positions $\min(\Delta t, t_c)$ time units from now and decrement t by this same amount.

(h) For each pair of objects that now collide, calculate their new velocities after "bouncing."

As for figuring out when objects collide and how their velocities change when they collide (that is, how they "bounce"), we're providing a "physics" package with these algorithms. Of course, those of you studying mechanics and vector algebra might just want to figure them out for yourselves, but that's up to you.

5 The API

The template files for this project include an interface called `util.Set2D`, and an implementation of it contained in `util.QuadTree`, and a utility class, `util.Debugging` to help control debugging output. The only things in the package `util` that your other files (`track.java` and any other classes you write) are allowed to use are the public methods and constructors defined in these classes. You may not expand this set with additional public or protected members (only private or package private ones). Do *not* try to circumvent this restriction, since we will be testing your main program and your `util.QuadTree` class separately using our own implementations, and they will fail if you violate the interface in any way. Both `util` and `tracker` also contain testing classes. Be sure *not* to make your implementation depend on these files either.

We have organized the template code to put all the command processing in a package called `tracker`, so that `track.java` consists of little more than a call to the public instance method `tracker.Main.main`. Again, leave this part of the interface untouched, leave `track.java` pretty much as it is, and don't move anything out of the `tracker` package into the anonymous package. This will allow us to test your `tracker` package separately from your `util` package (and incidentally, give you plenty of experience with packages).

Similarly, although you are free to not use the `ucb.proj2.Physics` package we provide, do *not* copy our `Physics.java` class into your directory in order to avoid having to learn about how to deal with external packages! Ask, if necessary, about how to use such a package (how to "add it to your class path").

6 Your Task

The directory `~cs61b/code/proj2` contains skeleton files for this project. They are also available via Subversion (under URL `$STAFFREPOS/proj2`).

Please read *General Guidelines for Programming Projects* (see the main lab web page). To submit your result, use the usual procedures. Before doing so, you can use `pretest proj2` on the instructional machines to do a simple sanity check. You will turn in nothing on paper.

Be sure to include tests of your program (yes, that is part of the grade), by extending the `tracker.Testing` and `util.Testing` methods in the skeleton. Our skeleton directory contains a couple of trivial tests, but *these do not constitute an adequate set of tests!* Make up your tests ahead of time.

The input to your program will come from fallible humans. Therefore, part of the problem is dealing gracefully with errors. When the user makes a syntax error, your program should not simply halt and catch fire, but should give some sort of message including the word "error" (in either case) on its first line and then try to get back to a usable state. However, for this project, we are not going to be fussy. As long as you detect and report the *first* error, your program will be judged to be correct, and any output after the first error message will be ignored. As for Project 1, your choice of recovery is less important than being sure that your program *does* recover gracefully. Your project should exit with status code 1 (which happens when you call `System.exit(1)`, if the command-line arguments are wrong. Do so also if the user made any error entering commands to the program, but instead of exiting immediately on such errors, remember that they happened and generate a status code of 1 when your program eventually does exit. Error-free execution, on the other hand, should result in a status code of 0, as is conventional.

Be sure to include documentation. This consists of a user's manual explaining how to use your program, and a brief internals document describing overall program structure.

Our testing of your projects (but not our grading!) will be automated. The testing program will be finicky, so be sure that:

- Your main function is in a class called `track.`. Your quadtree implementation must be in class `util.QuadTree` and must implement `util.Set2D`. The skeleton is already set up this way.
- Again, don't modify the API.
- The main procedures in classes `tracker.Testing` and `util.Testing` should run your tests. We have set up the skeleton file to show you how you test even package-private methods in your JUnit tests.
- Observe the exit-code conventions.

We will eventually be providing our own version of the main program and our own implementation of the API. You can use these to test the two parts of your program.

7 Advice

As before, don't make the problem any harder than it already is. If you find handling the command syntax to be difficult, you are probably making life difficult for yourself unnecessarily: talk to us about it. For writing files, there are `java.io.FileWriter` and `java.io.PrintWriter`.

It's important to have *something* working as soon as possible. You'll prevent really serious trouble by doing so. I suggest the following order to getting things working:

1. Write the user documentation.
2. Write some initial test cases.
3. Get the printing of prompts, handling of comments, the 'quit' command, and the end of input to work.
4. Implement the rest of the commands (yes, even though you don't have `Set2D` completely implemented, you *can* write this.) This should not be difficult; if you find that it is, please see us immediately, so that we can see if *we* have inadvertently made things hard! You'll be able to test it against our own `util` package (we'll give you details of how).
5. Now figure out (and document) how you are going to represent a quadtree, which you will use for finding points.
6. Implement the insertion of points into your quadtree.
7. Implement the iterator that lets you sequence through all points in your `Set2D`.
8. Now implement finding all particles within a given distance of a given point. To find all particles within a distance d of (x, y), first find all particles whose coordinates are in the range $(x \pm d, y \pm d)$, which gives you a superset of what you want. Next, check all of these points to see which fall within distance d.
9. Now it should be easy to find all pairs of points that are closer together than d.
10. Finally, tackle the `simulate` method.

As you go through this process, keep adding JUnit tests of the features or methods you add, using them to test your program.

Please don't move code out of the `tracker` or `util` packages in order to avoid dealing with packages. We will be testing that your `tracker` package works with our `util` package. In order for this to work, you must leave `track.java` pretty much as it is (basically just a call to `tracker.Main.main`).

7.1 Dealing with approximation

We haven't talked about floating-point arithmetic. The important point is that it is approximate: it keeps a limited number of digits, and uses binary fractions rather than decimal ones.

As a result, 0.125 is exact, but 0.1 is not. The upshot is that you must be careful about using `==` tests to stop loops or detect collisions. I suggest instead `>=` or `<=` as apppriate.

Contrary to what you'll read in some books, there *are* perfectly good uses for `==` in floating point, such as

```
if (x == 0.0)
    y = 1.0;
else
    y = Math.sin (x) / x;
```

but they probably won't come up in this project.

Because floating-point arithmetic is approximate, we have to take a few precautions. One student encountered this problem:

> "The following call returns 6.5745E-17: ucb.proj2.Physics.collide([0.0730, 3.3607, 1.2, 2.1], [-0.1777, 3.2762, 3.232, 1.132], 0.1323).
>
> I'm not sure whether or not this value is correct, but it is so small that when subtracted from the time the simulation is supposed to run, the value does not change. That is `t-6.5745E-17==t` is true. And so the simulator runs into an infinite loop because t is not decreasing. Is this a correct result from Physics? Or else what should I do about this problem?"

We responded: it is correct, and the looping problem is not caused by the problem you mention. If you compute the new positions of the particles after this period of time, you will find that they are touching to within the precision of floating-point numbers. What should happen is that you call `rebound`, and it adjusts the velocities so that the next call to collide for these particles returns `Infinity`. The fact that this particular round did not advance `t` is irrelevant.

It's OK to call `rebound` when objects are slightly more than `2*radius` apart. There's a right way to do this, but for our purposes, I suggest testing that the squared distance between the two centers is $\leq 4.0 \cdot \delta \cdot \text{radius}^2$, where δ is $1.0 + 2^{-50}$, written `1.0 + 1.0 / (1L << 50)`. (Don't leave out the 'L' after 1.) δ is a 51-bit floating-point value; double carries 52. I suggest testing the square value here (len2) because it avoids an unnecessary square root. Yeah, this is kludgy, but our physics here isn't exactly "physical."

Don't do anything like this:

```
oldTime = t; t = t-collisionTime;
advance things to their new positions oldTime-t from now;
```

for just the reasons you brought up in the original question.

7.2 Considerations for Eclipse and working at home

The skeleton suggests use of the utility class `ucb.util.CommandArgs`, which we've defined for handling command lines. We've also provided a class `ucb.proj2.Physics` for handling the interactions of objects. When compiling and running Java from the command line or in Emacs, you needn't do anything special on the instructional machines. But you'll have to inform Eclipse of where to find the libraries that contains them if you're using Eclipse. They are stored as JAR (Java ARchive) files in `~cs61b/lib/ucb.jar` and `~cs61b/lib/ucb-f2007.jar`. To inform Eclipse:

1. Select your Project 2 and go to the dialog Project-¿Properties.
2. Select "Java Build Path" on the left.
3. Select the "Libraries" tab.
4. Use the "Add External JARS" button for each library.

The Eclipse Project Wizard also gives you a chance to set this up when you first create your project.

Those of you doing the project locally at home can copy these two `.jar` files to your home machine (they are also available in the public svn repository in `staff/lib`). If you are working from home without using Eclipse, you need to add these libraries to your "classpath". On Unix-based systems (GNU/Linux, MacOS X, Cygwin installations on Windows) should add them to the CLASSPATH environment variable (whose value is a list of directories and `.jar` files separated by ':').

If you do your work from home by sshing into the instructional machines, you should have the usual class setup automatically and won't need to copy `.jar` files or change your classpath.

7.3 Using staff versions

We will maintain two JAR (Java Archive Repository) files containing the staff's versions of the packages `tracker` and `util`. You can use these to supply one half of the assignment while you work on the other. We are not supplying source, of course, so when you encounter an error that occurs in one of the staff's classes, you will have to step upwards in the call stack to find where your part most recently called the staff's. Don't overuse the staff packages; you should rely principally on your own unit testing. The staff package, however, can serve as a "sanity check" that you haven't seriously misunderstood the assignment.

The staff version of the `util` package is in `~cs61b/lib/proj2-util.jar` and the `tracker` is in `~cs61b/lib/proj2-tracker.jar`. To use either of these, you must include the appropriate JAR file in your "class path," as described above.