CS61B, Spring 2006 **Project #2: Spatial Database** **P. N. Hilfinger**

Due: 7 April 2006

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.

2 Commands

The main program you will write, called **track**, should accept commands in free format, meaning that whitespace is ignored except to separate words and numbers. Ends of lines terminate comments, but should otherwise be treated as whitespace. Print a prompt ('`>␣`') at the beginning and after each command and comment. Use the type `double` to represent approximate real numbers, as used in positions, lengths, times, and velocities. The commands are as follows:

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

bounds $x_{\mathbf{low}}$ $y_{\mathbf{low}}$ $x_{\mathbf{high}}$ $y_{\mathbf{high}}$ Set the positions of 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. You may assume that the largest *ID* is not supposed to be much larger than the total number of objects stored in the set.

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.

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*.

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

near x $*$ d Print the positions of all objects whose x coordinate is within d units of x, using the same format as the first variety of the ¡code¿near¡/code¿ command, above.

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

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

ID_1`:(`x_1, y_1`)` ID_2`:(`x_2, y_2`)`

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 on end-of-file.

3 Algorithms

The real problems come with the `within` and `simulate` commands. Naive approaches to these problems involve checking all pairs of objects for their separation (`within`) or to see whether they collide (`simulate`), $\Omega(N^2)$ problems, where N is the number of objects. We're going to try to do better. First, we'll use 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. Be careful how you do this. To do this, you should create a new quadtree, add the moved objects to it (i.e., at their new coordinates), and replace the old quadtree with the new one. (We have to do this rather than simply moving the objects within the old tree to avoid certain pathological cases).
   (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'll provide 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.

4 The API

The template files for this project include an interface called `util.Set2D`, and an implementation of it called `util.QuadTree`. 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 defined in `util.Set2D` and `util.Set2DIterator`, which you may not change, and the constructor for `util.QuadTree`. 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.

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").

5 Your Task

The directory `~cs61b/code/proj2` will contain skeleton files for this project. Copy them into a fresh directory as a starting point. Use the command

```
cp -r ~cs61b/code/proj2 mydir
```

to create a new directory called `mydir` containing copies of all our files (with the right protections).

Please read *General Guidelines for Programming Projects* (see the "homework" page on the class web site). To submit your result, use the command '`submit proj2`'. You will turn in nothing on paper.

Be sure to include tests of your program (yes, that is part of the grade) in the form of a JUnit test in class `FullTest`. 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.

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 must be 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 file FullTest.java should run your JUnit tests. We have set up the skeleton file to show you how you test even package-private methods in your JUnit tests.
- 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.

6 Advice

As before, don't make the problem any harder than it already is. Again, `Scanners` are useful for reading commands. 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.