CS61B, Fall 2004 **Project #2: Geographical Database** **P. N. Hilfinger**

Due: Wednesday, 10 November 2004 at 2400

1 Background

For this second project, we consider the problem of organizing a collection of data about objects in space (well, two-dimensional space to keep things simple). 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 constant 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 dimensionless moving objects (*particles* is the generic name), and answering the queries listed above about 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 ('>␣', as for Project #1) at the beginning and after each command and comment. The commands are as follows:

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

load *filename* Load data from file *filename* containing the positions and velocities of points. A file name is any sequence of non-whitespace characters. Any previous set of points is

first discarded. Each line of the file *filename* should contain four numbers (floating-point format) separated by blanks:

$$x \quad y \quad v_x \quad v_y$$

This denotes a particle at position (x, y), moving at v_x units per second in the x direction and v_y units per second in the y direction.

write *filename* Write the current state of all particles in the file named *filename* in the same format as used for **load**.

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

within d Print all pairs of points that are within distance d of each other in the format

(x_1, y_1) (x_2, y_2)

one pair per line on the standard output.

step N δ ϵ Where $N \geq 0$ is an integer, $\delta, \epsilon > 0$ are real numbers. This performs a simulation. The following is repeated N times:

1. Using the current positions and velocities of the particles, and assuming they continue to move at those velocities, update their positions to a time δ seconds in the future.
2. Then, *interact* all pairs of particles that are within a distance ϵ of each other, using the `interact` method we have supplied in the skeleton. This changes just their velocities. When a particle interacts with several other particles, just interact them in pairs in any order. (The term "*interact*" in this particular case happens to mean "bounce off of.")

This command does not print anything.

quit Exit the program with no further output. Do exactly the same thing on end-of-file.

3 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` and any other classes you write in the anonymous package) are allowed to use are the public methods defined in `util.Set2D`, 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.

4 Your Task

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

```
cp -r ~cs61b/hw/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). The makefile we provide has a convenient target for running such tests. 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 and update your makefile to run them.

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 makefile is set up to compile everything on the command `gmake` and to run all your tests on the command `gmake check`. The makefile provided in our skeleton files is already set up to do this. Be sure to keep it up to date.
- 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.
- We will 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.

5 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.