COS126/128: General Computer Science	**Spring 1996**
Problem Set # 10	
Handed out: May 2, 1996	*Due: May 12, 1996*

Summary:

In this assignment you will write a program that simulates the motion of N bodies, mutually affected by gravitational forces, in a two-dimensional space.

As warm-up, you will write a program that uses a simple X windows graphics library interface (provided by us) to simulate a bouncing ball within an X window.

This assignment asks you to use a lot of code that we have written that we don't expect you to understand. You can take the graphics library interface on faith and focus on the functions that calculate the positions of objects. It should be clear where your code goes from the description below; if not, ask!

Part 1: Bouncing Ball (20%)

For this part, you need to add only a few lines of code to the software that we provide. *Do this early so you can get help if you need it.* You need not turn in this part if you get Part 2 working. Part 2 is a similar, but more complex simulation. If you try to skip directly to Part 2, you may find it very difficult to make any progress.

Your starting point is the code `SimLib.c`, `SimLib.h`, and `bounce.c` in the directory `/u/cs126/progs`. `SimLib.c` provides a simple facility that allows you to write programs to do dynamic graphics in an X window, a sample program (`bounce.c`) that you will modify, and an interface (`SimLib.h`) between the two.

There's also a `makefile` that builds `a.out`. `Make` is a UNIX utility that is useful when one is writing a program that is broken up into separate files. It allows one to write down in a file called `makefile` the dependencies among the files comprising the program (in a special language that you can read about if you care to by typing `man make`). Executing the command `make` will cause the pieces of the program to be separately compiled, and then linked as necessary. You can do this assignment without understanding `make` if you follow the directions carefully. (Those of you who take cos 217 will learn all about `make`).

Copy `makefile`, `bounce.c`, `SimLib.c` and `SimLib.h` into an otherwise empty directory, then type the command `make`. This compiles the C files and binds the resulting object files with the X windows library into a single `a.out`. Executing this `a.out` displays a window and shows the track of a ball moving from the center out at an arbitrary angle (watch carefully). The routine `SLdraw` generally controls the situation—it stays in a loop, waiting for events that affect the window, and calling the routine `orbit` that is passed to it. Yes, the argument to `SLdraw` is a function name! This is soemthing new to us. But you don't need to fool with SLdraw, just use it as we have shown you.

Your first task is to fix the code so that the ball doesn't leave the window. The "true coordinates" of the ball are intended to fall in the in interval $[-.5, .5]$. These are converted to "screen coordinates" int the range $[0, 500]$ in the routine `drawball`. Your task is to make them stay in that range, by maintaining the illusion that the ball bounces off the side of the window. This involves only a few lines of code.

Next, add code to remove the tracking "feature" (showing the track of the ball) and just show the ball as it moves along. This involves implementing a standard idiom in graphics: to give the appearance of motion, "erase" the image of the ball at the old position before drawing it at the new position.

Though `ctrl-C` always works, the proper way to cause the window display to disappear and exit cleanly is to type "q" while the display window is active. This and other details about how you interact with the window, such as what happens when you change its size, etc., are handled in `SimLib.c` using the X toolkit. You may wish to browse this code to gain some appreciation for what is involved. You probably won't want to change it: if you do, stand back!

The n-body simulation program involves adding more balls and having them move in more complicated paths than straight lines. You may want to try adding more bouncing balls to this program, or modifying it to take some parameters as input, as a warmup.

Part 2: Simulating N bodies (80%)

Write a program that simulates the motion of N bodies, mutually affected by gravitational forces, in a two-dimensional space. This program will use the same simple X windows graphics interface used in Part 1, the code `SimLib.c` and `SimLib.h` and the `makefile` in the directory `/u/cs126/progs`. You may use the code `/u/cs126/progs/nbody.c` as a starting point: this has the same basic structure as in `bounce.c`, but additionally includes an `initread` function to read the initial characteristics of the N bodies, and an expanded `main` that allows more flexibility in providing data to the program with command line arguments and data input from files. Your task is to implement the function `drawall` that handles the actual calls to the graphics package, and the subroutine `orbit` that recalculates positions. You'll also need to edit your copy of `makefile`: change occurrences of `bounce.c` and `bounce.o` to `nbody.c` and `nbody.o`.

The program is to be invoked with a command such as

```
a.out 100 SSfive
```

where `100` is the time step to be used in the simulation and `SSfive` is a file containing one line with five pieces of information for each body: the x and y coordinates; initial values for v_x and v_y (the x and y components of the velocity); and the "mass" of the body. These values are needed in computing the effects of mutual gravitational interactions, as we will see.

First, get `drawall` implemented, and make sure that you understand the basic data structures and how `main` works by plotting the bodies in the file `/u/cs126/data/SSfive`. This is the easy part: don't even put in any "erase" code to remove tracking at this point; it's liable to get in the way of debugging. Note that `initread` keeps a variable `max` that is the largest value of any x or y coordinate.

Next, do the code for updating the coordinates for each display. This involves a "force" calculation and a "velocity and acceleration" calculation, both from elementary physics. We consider these in the next two paragraphs.

The strength of the gravitational force between two bodies is given by the formula $F = M_1 M_2 G/r^2$, where M_1 and M_2 are the masses of the two bodies, r is the distance between them, and G is the gravitational constant 6.67×10^{-8}. You need to calculate these forces for all possible pairs of bodies and add all the forces acting on each body. This is a *vector* addition: the pull of one body towards another is on the line between them. Therefore you will have to compute the x and y components of the forces and add them, to get F_x and F_y components representing the total gravitational pull on each body of all the other bodies.

Next, use $\mathbf{F} = m\mathbf{a}$ to get the x and y components of the acceleration of each body that is implied by these forces over the given time step dt. From these, the velocity and position can be updated: the change in velocity is given by $\mathbf{\Delta v} = \mathbf{a}dt$ and the change in position is given by $\mathbf{\Delta p} = \mathbf{v}dt + \frac{1}{2}\mathbf{a}dt^2$. These are all vector formulae: for example, $\mathbf{p}$ represents the x and y position coordinates, $\mathbf{v}$ represents the x and y coordinates of the velocity, etc.

Part of the challenge of this assignment is in coping with realistic scientific data values, as you will see when examining the data files. Use `double` floating-point values at every opportunity, though this isn't even quite good enough (see below). Most solutions to this problem will have serious roundoff errors in the computation. For example, this leads to all planets eventually spiraling into the sun in `SSfive`. The force, velocity, and acceleration update function only takes thirty or so lines of straightforward code, but the use of "real data" means that you will want to think carefully about what the code is doing, as debugging will be difficult. Can you find a way to fix or at least minimize the the roundoff error problem? (Actually fixing the roundoff error is quite difficult and not expected of you, but you should be aware of the problem.)

To speed things up somewhat, you should fix things so that `orbit` only calls `drawall` every T steps, where T is set to, say, 100 with a `define`. Once you get the orbits plotting in "track" mode, put the erase code in.

After you get the orbits for `SSfive` working, try the data `BINfling`. Note that the small planet gets slung out into the far reaches of the solar system. The simulation may be made correct by tracking it and continuing to scale down, but it normally would be difficult to provide a display that looks good under all conditions for the system.

For an amusing variation, borrow from your experience in Part 1 to keep the bodies in the window by having them bounce off the edges. Though it's dubious that this happens in real-world situations, this makes for a "virtual" universe with fascinating properties, especially for systems such as `BINfling`.

Extra Credit: (10%) Add a slow moving comet to `SSfive`. Try to make it hit one of the planets.

What to submit: You should submit your program for Part 2. Call your source file `nbody.c`. If you use more source files, also submit your `makefile`. Do not submit your program for Part 1 unless you fail to get a solution to Part 2. If you attempt the extra credit, submit it separately.

Due: Sunday, May 12, 1996

Submit electronically by 10:00PM Sunday, May 12. Annotated hardcopy must be handed in by 10AM Monday, May 13. Turn in hardcopy using the COS126 bin in the lobby of the second floor of the Computer Science Building.