Programming Project 7

EE312 Spring 2014

Due 4/4/2013 at end of the day

FIVE POINTS

General: This project is a C++ project, and it includes some really cheesy graphics. As such, unless you want to write your own graphics, the project has to be completed in Windows and compiled against the Microsoft graphics libraries. Fortunately, there's a Visual Studio project file already set up for you that will take care of all this.

Your Mission: Simulate Life. As usual, the functions that you will write are in Project7.cpp. There's quite a bit of (gnarly) code in main.cpp. Unlike most of our projects, this time the code in main.cpp may be too confusing to figure out. We're using Microsoft Foundation Classes to provide the (primitive) graphics in this program. As a consequence, we have to use classes (from C++) and deal with the "event-driven" nature of modern window systems. The end result may be impenetrable. The bottom line is that every 100 milliseconds a timer goes off (courtesy of Windows). That timer causes the function *CFrame::OnTimer* to be called. This function, in turn, calls the main simulation function *TimeStep*. When *TimeStep* is called, the simulator should update the position, direction and energy of all the life forms in the (simulated) universe. After another 100ms have passed, the timer will go off again and *TimeStep* will be called to update the world again. This process continues until the graphics output window is closed. As the simulator runs, the function *CMyFrame::OnPaint* is used to draw graphics on the screen. This function relies on two global variables, a two-dimensional array called *world* and a *vector* called *bug_list*. Your simulator will have to update these variables correctly each time step. When we grade your program, we'll look at the value of these variables to determine your score.

Rules of the Simulation: The purpose of this simulation is to study the implications of Darwin's theory of evolution. To this end, we'll create a virtual world (200x200 units in size) and simulate a population of roughly 100 "bugs". The bugs are sort of like protozoa, and they swim around trying to eat algae. If a bug does not eat enough algae, it will eventually die (of starvation). If a bug eats enough algae it will reproduce. The birth and death of bugs is the key to our simulation. Bugs with good genes will survive and multiply. Bugs with bad genes will die off. *Et Voila!* Evolution.

Each time step, every bug moves one position forward in its current direction. There are eight directions for this project, the four "manhattan" directions (left, right, up and down) plus the four diagonals. The functions *newX* and *newY* are provided to you (in Project9.cpp) to standardize the directions. Given the current x-coordinate and the current direction of a bug, *newX* returns the x-coordinate of the new position the bug would have after moving one step in that direction. The function *newY* does the same thing for the y-coordinate. As I was saying, each time step, every bug moves one

position forward in its current direction. The new position the bug moves into may be empty, in which case the bug simply moves into this position or occupied. If the new position is occupied by *FOOD* (i.e., algae), the bug should eat the food. Eating food causes the bug to gain strength. Specifically, each bug has a *health* data member, and this *health* should be incremented by *EAT_HEALTH* whenever the bug eats food. If the new position is occupied by another bug, then nothing happens. Both bugs are allowed to occupy the same place. While that may not be very realistic, it does not change the way in which a population evolves, and it does make the simulation much simpler.

Regardless of whether the bug ate anything or not, after moving the bug's health is decremented by the value *MOVE_HEALTH*. If the bug's health ever gets to zero, the bug will die (if it gets to *REPRODUCE_HEALTH*, the bug will give birth to an offspring, more on this later). All of these constants (*EAT_HEALTH*, *MOVE_HEALTH*, etc.) are in the file *Bugs.h*.

Then the bug decides (randomly) whether or not to change direction. The bug's genes determine the probability of turning. Since there are eight possible directions, there are eight possible choices the bug can make. Choice 0 is to go straight. Choice 1 is to turn right 45 degrees, choice two is to turn right 90 degrees, and so on until we get to choice 7 which is to turn right 315 degrees (i.e., same as turning left 45 degrees). With the way we model directions, if your current direction is direction 3 (heading down and to the right diagonally) and you've decided to take choice 2 (turn right 90 degrees), then your new direction is direction 5 (3 + 2 == 5). Don't forget to deal with the wrap arounds. For example, if you're currently headed in direction 6 and you select choice 4, then your new direction should be direction 2.

Genes: Each bug has a set of genes which describe the relative probabilities of turning. We store those genes in an array of eight integers for each bug. Simply put, the genes of a bug can be assigned to the eight possible choices in any way. So, a bug with 16 genes might have 5 genes allocated for going straight (choice 0), 3 genes allocated for choice 1, 3 genes allocated for choice 2 four genes allocated for choice 6 and one gene allocated for choice 7, with zero genes allocate for the remaining choices. That means that this bug has a 5/16 chance of going straight, a 3/16 chance of choice 1 (turning right 45 degrees), also a 3/16 chance for turning right 90 degrees, a 4/16 chance of turning left 90 degrees and a 1/16 chance of turning left 45 degrees. Each time step, the bug will randomly update its current direction according to these probabilities.

Birth and Death: When a bug dies, it is simply removed from the simulation. When a bug gives birth, we mutate the child's genes slightly. Specifically, the child inherits the parent's genes except that one gene has been moved to a new position. So, if the parent's genes were as above (listed here in order): {5, 3, 3, 0, 0, 0, 4, 1} then the child's genes would be almost exactly the same except one position in the array would be incremented and one would be decremented (by one). Obviously you can't have a negative number of genes allocated to anything, so the smallest possible entry is zero. With the example above, the child's genes might be {5, 2, 3, 0, 0, 0, 4, 2} (moved one gene from choice 1 to choice 7).

Your Implementation, the bug_list: You are required to use a *vector* named *bug_list* to contain all of the bugs currently alive in the simulation. For the most part, you can treat this *vector* just like an array. There are a couple of really nice features.

- The *bug_list* can grow or shrink the number of elements (i.e., number of bugs). You are required to grow the *bug_list* whenever a new bug is born, and shrink it whenever a bug dies.
- Using "`bug_list.size()`" you can find out the current length of the *bug_list*. This expression evaluates to an **int** and you can use it anywhere you could use an **int** variable. Note that the "." and the "()" are required.
- Using "`bug_list.pop_back()`" you can reduce the size of the *bug_list* by one. The last bug in the *vector* is removed. You don't need to call malloc or free, the implementation of *pop_back* will take care of the memory management for you.
- Using "`bug_list.push_back(b)`" you can add the new bug *b* to the bug list. The size of the *vector* is increased by one, and the new bug is added to the end. You can access this bug with "`bug_list[bug_list.size() - 1]`"

Note that in this program, you will not need to call *malloc* or *free* (ever). You will almost certainly not need to use pointers either. Think of the *vector* as being an array with three funky functions (those shown above) and you'll do fine. Remember, with structs, we treat a struct variable just like we do an integer. So:

```
Bug b; // declared a new Bug variable, like "int x"
b.health = 42; // set the data members
b.x = 86; b.y = 13;
bug_list[10] = b; // no more exciting than v[10] = 17 for int array v
```

Your Implementation, the world: The graphics are based on the values stored in a 2-dimensional array called *world*. Each element in the array represents a position in the virtual world we simulate. The *world* array holds **int** values. If the value is non-negative (positive or zero), then there is a bug in that position. For example, if *world[10][8]* is the value 3, then bug #3 is position (10,8) in the world (x = 10, and y = 8). If the value in *world* is negative then the square is either empty or has food (no bug). We use special variables (constants) to describe the negative values. *EMPTY* is the value when a position is empty, and *FOOD* is the value when a position has food in it.

What to do: You have to finish the implementation of a *timeStep*. During a time step, each bug moves one space, and may eat or turn. Dead bugs are removed from the bug_list and healthy bugs give birth to new bugs.

moveBugs: For each bug in bug_list do the following:

1. calculate the new position for the bug
2. check the new position to see if there's food there. If there's food, increment the bug's health by EAT_HEALTH.
3. calculate a new direction for the bug:

   a. generate a random number between 0 and GENE_TOTAL with "`i = rand() % GENE_TOTAL;`"
   b. determine which choice the bug has made. The easiest way to do that is to recognize that if genes[0] is 5 then there is a 5 out of GENE_TOTAL chance of choice 0 (go straight). So, if your random number is smaller than genes[0] the bug has picked choice 0. If your random number is larger than genes[0] but smaller than genes[0] + genes[1] then the bug has picked choice 1 and so on.
   c. Update the direction based upon the choice made
4. Update the *x* and *y* coordinates for the bug (the data members named *x* and *y*.
5. Update the world[][], change the old position to EMPTY and set the new position equal to the bug's number (it's index in the bug_list array).
6. Subtract MOVE_HEALTH from the bug
7. (done for you) Add one to the bug's age and update total_age (total_age should be set to the total age of all bugs). Also update total_gen.

KillDeadBugs: For each bug in bug_list check to see if the bug's health is zero. If the bug has zero health, it is dead and should be removed from the world and the bug list as follows. Set world[i][j] to EMPTY where i and j are the current position of the bug. Finally remove the bug from the bug_list. You can do this in constant time by swapping the bug with the last position and then using bug_list.pop_back. Make sure you don't skip over any bugs in the bug list (test your code for the case that both bug_list[1] and bug_list[2] are dead (have zero health) and bug_list[3] is not dead).

ReproduceBugs: For each bug in the bug_list check to see if the bug's health is greater than or equal to *REPRODUCE_HEALTH*. Each such bug should give birth to a new bug as follows:

1. Create a new bug. This is easy, just declare a variable of type Bug.
2. Make the child bug a copy of the parent (also easy, use *child = parent).*
3. Update the energy of both parent and child, give ½ the parent's energy to each.
4. Update the genes of the child. Chose one non-zero element in the genes[] array to decrement and one element to increment (you can increment a zero, just don't make any negative values in the array).
5. (done for you) Update the statistics for total_straight, total_right, total_left and total_back to be the total number of genes assigned to each.
6. Add the child bug to the end of the bug_list using push_back.

JUST LOOK FOR "TODO" in Project7.cpp if you're confused about what you actually need to do.

That's All: If you get timeStep to work, just run the program. It should open a window and start displaying the bugs (hopefully they're moving around). You can run the program in the debugger, but you will not be able to see the graphics when you single-step the program. On the right side of the window is a graph. Resize the window large enough (vertically) so that the statistics summary appears and the graph will start working. The graph shows the population. The red line is "stable" population (100

bugs). If the black line is to the right, then the bugs are overpopulated, to the left, underpopulated. The graph shows time on the vertical axis, with the distant past at the top of the graph and the present at the bottom. You can tell if your program is working if the population oscillates back and forth around 100, and if the genes begin to evolve. You should very quickly see the "back" genes disappearing (back percentage goes to 1% or 0%). The straight genes should increase to somewhere between 65% and 75%, but that can take several hours depending upon the value of GENE_TOTAL – the larger the value of GENE_TOTAL (which must be a multiple of 8) the slower the bugs will evolve). Good luck, have fun!