LifeForm Simulation

Problem 1 - Simulate Life

You are to complete the simulation infrastructure that you get from us. You are also to invent at least one new LifeForm (although you are encouraged to invent several) and simulate the LifeForm's existance (or evolution) of that LifeForm.

We will evaluate your output in two phases. The bulk of the score (90 out of 100 total, to be exact) will be determined by running your simulation in isolation. Your LifeForm must be capable of sustaining a population of individuals that eat, move around, reproduce and die. Reproduction can be sexual or assexual (assexual is much easier, (hint)) and your species need not evolve (although, evolution is fun). In the second phase, we will run a simulation using all of those species that are properly designed to be compatible with the overall simulation strategy. In other words, we'll build our own region and LifeForm base class, and compile all of your objects along with it. Species that become extinct quickly will receive no points, but species that succeed in dominating the simulation (have the largest populations) will get up to 10 points.

Events

You should use the make_procedure function to create function objects as described in class. Events can then be dynamically allocated. The Event constructor takes a time and a procedure. The procedure will be called that many time units into the future (the time is relative, not absolute). Creating an event automatically puts the event in the central event queue. **ALWAYS CALL new TO CREATE EVENTS**. Events automatically delete themselves when they occur. If you delete an event before it occurs it automatically removes itself from the event queue. Don't accidentally delete the same event twice! (more on this later).

Note that if an object is destroyed (e.g., eaten), it is necessary to remove all of the pending events for that object (it shouldn't continue to move). The LifeForm::die() member function automatically deletes all of the events associated with a LifeForm. The die function is given to you (in LifeForm-Craig.cc).

Regions of space

You will use the QuadTree<LifeForm*> data structure to represent space. One additional method has been added to the QuadTree class specifically to support Project 2. This method `update_position` takes two arguments; the old object position and the new object position. The update_position method moves an object from one location to another in the QuadTree. Zero, one or two callbacks may be invoked as the object is moved. There must be an object in the tree at the old position (or else the QuadTree will crash).

How to move around

LifeForms can set their speed and their course. In the virtual world, they move continuously. However, in the simulation, they ``move'' by experiencing movement events. The QuadTree decomposes the virtual universe (which is two-dimensional) into small ``regions''. There will be at most one LifeForm in any region at any time. A LifeForm should experience a movement event whenever it crosses a boundary between two regions. You can find out when your object will next cross a boundary by asking the quadtree. The `QuadTree::distance_to_edge` function will tell you how many units of distance you can travel along some course until you'll reach the edge of the boundary.

To implement movement, provide five methods for LifeForms.

- LifeForm::update_position(void) - computes the new position, charge for energy consumed. To compute the new position, you'll need to know how long it has been since the last time you'd called update_position on this object (how much delta time has passed). You'll need a data member called update_time. Each time update_position is called, use Event::now() to see how much time has passed since update_time. Calculate the new position by multiplying your speed by the delta time. Don't forget to set update_time equal to Event::now() when you're done.
- LifeForm::border_cross(void) - This is your movement event handler function. It calls update_position and then schedules the next movement event. You'll probably want to keep a pointer to the movement event as part of your LifeForm base class (that way, you can cancel this event, see below).
- LifeForm::region_resize(void) - this function will be a callback from the QuadTree. When another object is created nearby, your object needs to determine the next possible time that you could collide. The QuadTree knows when objects are inserted inside it, so it can invoke this callback function on your objects. What you will want to do is have region_resize cancel any pending border crossing events, update your position, and schedule the next movement event (since the region has changed size, you will encounter the boundary at a different time than than before.)
- set_course and set_speed - These functions should cancel any pending border_cross event, update the current position of the object and then schedule a new border_cross event based on the new course and speed. Note that an object that is stationary will never cross a border.

Encounters

If two creatures ever get within one unit of distance of each other, they have an enounter. You only need to check for encounters when an object crosses a border. Since there's only one object per region, you can't possibly encounter anyone until you cross a border and enter their region.

Actually, that's not 100% accurate. Objects could, in principle, travel very close to the edge of a region, get very close to other objects, and then turn away at the last (micro)second. YOU DO NOT NEED TO HANDLE THIS CASE. It's very unlikely that objects will get within one unit of other objects without crossing a border, and so we'll simply ignore that possibility completely.

You should check for an encounter when an object moves from one region to another. That guarantees that if an object moves directly towards a stationary object, it must eventually encounter that object. Similarly, if two objects move towards each other, they must eventually encounter each other. The test program will look for these cases.

Collisions occur only between two LifeForms. When a LifeForm moves to a new region, it collides with the closest LifeForm that is within 1.0 space units. The test program will not check for collisions between three or more LifeForms. However, your simulator must not crash when this happens.

When a collision occurs, simulator must invoke the *encounter* method on each of the colliding life forms. The *encounter* method is pure virtual (abstract) and must be implemented by every derived LifeForm class (Craig, Algae). Be careful, the derived LifeForm can do arbitrarily sophisticated things inside its *encounter* method. That includes calling *perceive*, or *set_course*, or even *reproduce*.

The *encounter* method returns either `EAT` or `IGNORE`. Since there are two LifeForms colliding, there are four cases; IGNORE/IGNORE, EAT/IGNORE, IGNORE/EAT and EAT/EAT. Provided at least one LifeForm attempts to eat, you should generate a random number and compare the result to eat_success_chance(eater->energy, eatee->energy). More on this later.

Creation of LifeForms

NOTE: LifeForms cannot be created spontaneously and just expect to be simulated. There are two ways that LifeForms can be created. First, they can be created by the LifeForm::create_life method during initialization. create_life reads from standard input a set of lines with the format ``Species #''. For example, if you type ``Craig 10'' followed by ``Algae 100'' and then hit ctrl-D, the simulator will create 10 Craig objects and 100 Algae objects, insert all of these objects are random places within the QuadTree, and then begin the simulation.

The other way that objects can be created is by calling reproduce. Only objects that are alive can be used to reproduce other objects. Your simulator must check that non-alive objects do not successfully reproduce new objects.

The proper procedure for reproduction is illustrated in the Craig::spawn method. First, `new` is used to create a new LifeForm derived object. Then, reproduce in invoked on an existing object with the new object as the argument. The Reproduce method must insert

the new object into the QuadTree and begin simulating it (see below for further game rules).

NOTE: Since only living objects can invoke set_course, set_speed, reproduce, perceive, , and since objects are not ``alive'' until they've been inserted into the QuadTree, we have the very important result that LifeForm objects must not call any of these functions from inside their constructors. Once again, Craig::Craig illustrates the recommended behavior. The constructor schedules an event to occur as soon as possible (zero time in the future). Since this event will not happen until after the current processing (initialization or reproduction - whatever it was that caused us to call `new Craig`) is over, we know that the Craig object will be inserted into the tree before it begins running *hunt* for the first time.

HINTS:

- is_alive - Objects have a nasty habit of dying at inconvenient times. For example, object A moves next to object B. The movement of A causes the resize_callback to be invoked for B. When the resize_callback is run, B's new position is calculated (see update_position above). B may now be dead. Yup, B may have just run out of energy and is officially dead before A can eat him. There are no corpses in this game, B must be removed and A must not encounter B. Depending on how you implement your simulator you may or may not have trouble with this specific case. However, sooner or later, you'll encounter a tricky situation with objects dying at inconvenient times.

  The is_alive flag is included so that you can indicate dead objects that might not yet have been deleted. Check is_alive at appropriate times to make sure you don't have zombies in your game.

- die/_die - To further help with the problem of inconveniently timed deaths, the actual desctruction of LifeForms is performed by the *_die* method. You should never invoke _die directly! Instead, invoke *die* (without the underscore). The die method clears is_alive and then schedules an event for _die to be invoked as soon as possible. NOTE: _die deletes this (destroying the LifeForm forever).
- Update position may need to move the object from one position in the QuadTree to another position in the QuadTree. As objects are moved in the QuadTree, their region_resize callbacks are invoked. The region_resize callback is very likely going to want to update the object's position. If you think about this long enough, you can easily see the possibility of a cycle forming, and two objects repeatedly updating their positions and causing each others callbacks to be invoked, etc.

  To avoid this cycle, and to generally increase the robustness of the simulator, we strongly recommend that update_position check to see if the amount of time that has passed is greater than 0.001 time units. Objects don't move very far in 0.001 time units, and so there's no point in recalculating their position. More importantly, there'll be no reason to even try moving them in the QuadTree.

- When calculating the time until the next border crossing event (something you need to do from inside set_course, set_speed, region_resize and border_cross), fudge. Specifically add Point::tolerance time units to the time. That way, when the time is up, the LifeForm will not only have reached the edge of the boundary, but it is certain to have crossed the edge. If you make the mistake of scheduling the event for when the object is exactly on the edge, you will run the risk of the object not actually leaving its region. Due to floating-point roundoff errors, the object could be 1×10^{-15} space units from the edge (or so). Effectively, your program could go into an infinite loop, as the object keeps getting closer and closer to the edge without ever really crossing it.

Rules of the Game

Every LifeForm must provide the following methods:

- Action encounter(const ObjInfo&)
- String species_name(void) const
- void draw(void) const

LifeForm Species Names

There are two functions, player_name and species_name. Both of these functions are virtual. Species_name is pure virtual. By default, player_name returns the species name. Each derived class must implement a player name that is unique. I recommend some variation of your lrc login name. The player name is used during the contest to select the winning LifeForms among all students in the class. If you create more than one LifeForm, make sure each LifeForm has a different player name. For example, if you had two LifeForms and your lrc login was ``astud'', then you could have one LifeForm with the player name ``astudHunter'' and the other LifeForm with the player name ``astudGatherer''.

The player_name is used only by the contest and only to print out which LifeForms are surviving (and which have become extinct). The species_name is used when objects collide with or perceive each other. The species_name can be anything you'd like. We recommend that you use your player_name for your species_name. To facilitate this, you may include a ``:'' in your name. The contest ignores the : and anything that comes after it, but, the full name is available to anyone who perceives you. For example, my life form might use ``Craig:HELP I'M BEING EATEN'' as a species name. If one of my other Craig LifeForms were to perceive an object with this name, it could immediately rush to help.

To keep things interesting, students are allowed to lie about their names in the species_name (but not in their player_name). For example. You can set your species_name to ``Algae''. Other LifeForms will see you and probably come right to you (hoping to eat you). Perhaps you can eat them first!

Perception

The perceive routine can be invoked by any derived LifeForm class at any time. Only living objects can be perceived by other objects. There is no carrion in this game. Percieve has three params in Params.h that must be used. max_perceive_range and min_perceive_range define the maximum and minimum radii for the perception circle. If a LifeForm tries to perceive with a larger or smaller radius, you should set the radius to the maximum (or minimum), and then invoke the perceive code as normally. Each time an object attempts to perceive, it should be charged the perceive_cost. Note that perceive_cost is a function of the radius.

The perceive function should return a list of ObjInfos. Each ObjInfo includes the name, bearing, speed, health and distance to the object. One ObjInfo should be placed into the list for every object within the prescribed radius.

Eating

Objects eat each other according to their desire (indicated by the return value of *encounter*) and their eat_success_chance (defined in Params.h). If object A wants to eat object B, then the simulator should generate a random number and compare it to the eat_success_chance(A, B). If the random number is less than the eat success chance, then A gets to eat B.

Watch out, B could also be trying to eat A at the same time. Your simulator must allow stationary (or slow) objects to eat other objects (and vice-versa). Your simulator must not allow two objects to both succefully eat eath other. Params.h includes a variable called encounter_strategy which explains how to break the tie. To make the implementations uniform, we have set the tie-breaking rule so that the faster object gets to eat the slower object in the tie-break situlation. Note: there's a tie only if both A and B generate random numbers (they must each generate their own number) that are less than their eat_success_chances. In this case only do you use encounter_strategy to break the tie.

Once you've figured out who got to eat whom, you have to award the victor the spoils. The rules for this are for the eater to gain the energy of the eatee, but with two caveats.

1. The energy is awarded to the eatee after exactly digestion_time time units have passed (create an event).
2. The energy awarded is reduced by the eat_efficiency multiplier.

Since eat_efficiency is less than 1, it is usually not a good idea to eat your own young.

Aging

Objects must eventually die if they don't eat. This is true even if they act like rocks and don't move and don't perceive. The LifeForm base class should ensure that the age_penalty is subtracted from a LifeForm every age_frequency time units since it was created. Note that it is not acceptable to apply the age_penalty to all LifeForms at the same time. Not all LifeForms are born at the same time (some are children of other LifeForms) and so they shouldn't be penalized at the same time.

Reproduction

When a LifeForm reproduces (see above), the energy from the parent LifeForm is divided in half. This amount of energy is given to both the parent and the child. Then, the fractional reproduce_cost is subtracted from both the parent and the child. For example, a LifeForm with 100 energy that reproduces will produce a child that has 50 * (1.0 - reproduce_cost) energy. The parent will have the same energy as the child.

As a special rule, do not allow an object to reproduce faster than min_reproduce_time. This rule is included to prevent the ugly strategy of LifeForms reproducing themselves to death to avoid being eaten (inside encounter call reproduce 1000 times). If a LifeForm tries to call reproduce twice within min_reproduce_time time units, the child should be deleted and no energy penalty should be applied to the parent.

Problem 2 - Create Life

Create a subclass of LifeForm that can be simulated. The rules are that each LifeForm must behave independently. You should not allow a LifeForm to know anything other than what it can determine using the LifeForm base operations. For example, a life form does not know the absolute value of its energy. It does not know its position. It does not know anything about other objects except what it has learned through perceive. The file Params.h includes some (adjustable) parameters. You should try and design LifeForms that can adapt to different parameters, rather than hard-coding your strategy based on the current parameter values. **That means, no global variable or static data members for classes derived from LifeForm**.

Your implementation of the LifeForm base class should be sufficient to correctly simulate Algae, Craig and Rock classes. Rocks don't move and don't eat, so they should age and die. Craig's run around and eat Algae, and might even turn carnivorous if there are other LifeForms to eat as well. In addition, you should create your own derived life form object. You must implement `species_name` so that the first several characters are the same as your LRC login. After your LRC login, you can add a ':' character (optional). If you have a ':' (that allows me to find the end of your login name) then you can add as many additional characters as you'd like. Remember, the species name is revealed in the ObjInfo class (when you are perceived or encountered), thus you can use your species name to allow your life forms to communicate. For example, I could dynamically change the species name of some of my objects from ``Craig:everything's OK'' to ``Craig:I'm about to die, eat me!'' to ``Craig:there's a big algae patch to the northwest''.

Design your LifeForm class to outlive the Craig objects. When everyone has turned in the assignment, I'll run a big simulation with everyone's objects. 10% of your score for this assignment will be based upon the outcome of this simulation. I will probably tweak the game parameters somewhat (Params.h), so you're encouraged to design LifeForms that will adapt to the world, rather than spending a lot of time optimizing for a specific set of parameters.