EcoSym Final Report

COMP 50CP: Concurrent Programming

Team members: Reema Al-Marzoog, Ben deButts, Nathan Stocking

PROJECT OVERVIEW

We created a simulation of a marine ecosystem. The ecosystem contains marine organisms and simulates the relationships between these organisms. We also simulate a couple of natural processes that occur in such ecosystems.

Minimum Deliverable:

Our minimum deliverable is a simulation of several marine organisms interacting in an enclosed and limited landscape (i.e., on the scale of a small saltwater lake rather than an entire ocean). The interactions will include predator-prey relationships. We will also simulate a limited number of oceanic events.

Maximum Deliverable:

Our maximum deliverable is a simulation of the entire ocean. This includes natural processes such as ocean currents, a large number of organisms and relationships, and unexpected natural occurrences in response to which the ecosystem will react and adjust.

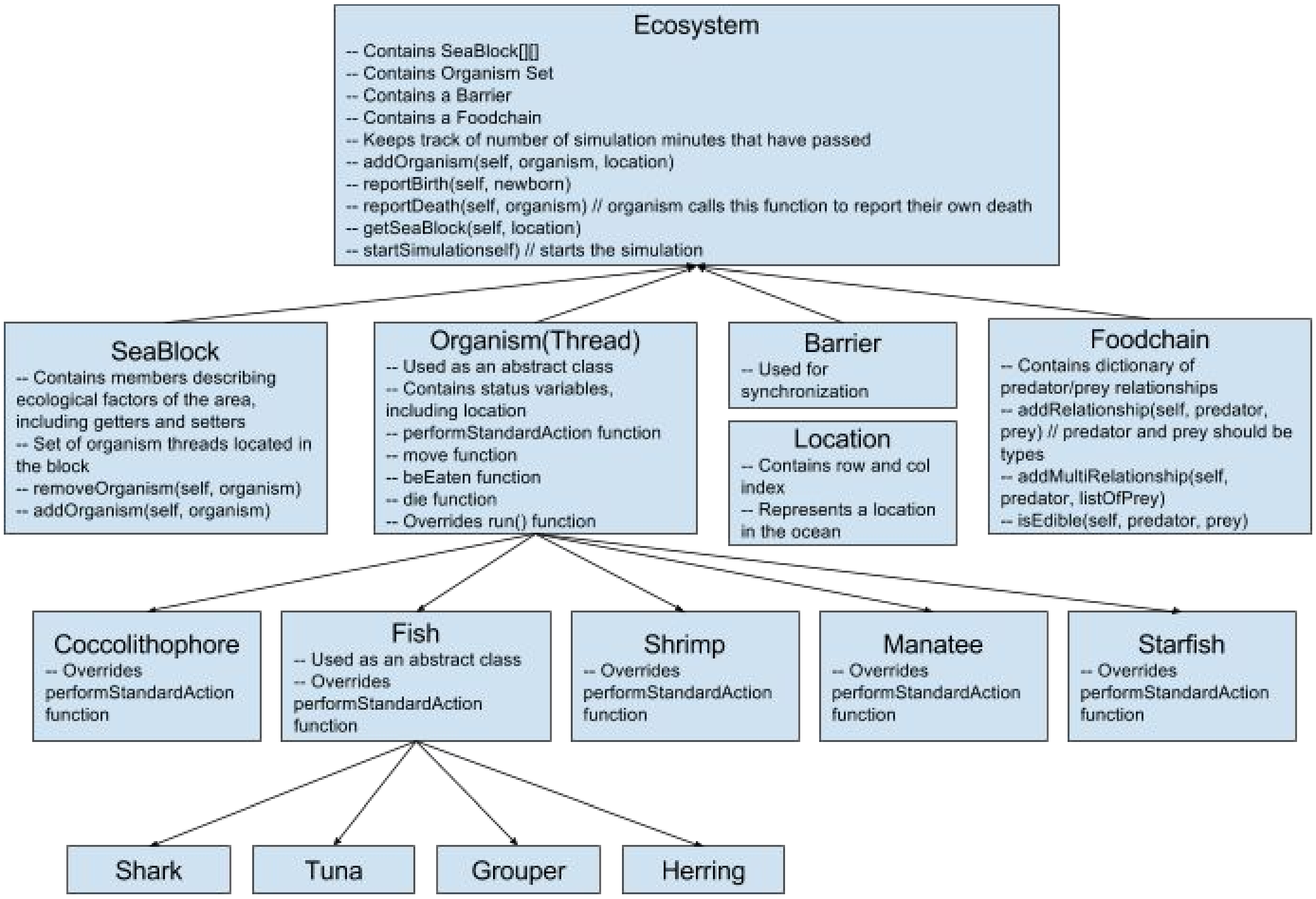
Achieved Deliverable:

We created a simulation of a number of marine organisms, with the potential to quickly add more. The size of the body of water and number of organisms is only limited by the thread limit on the machine running the simulation. Features include predator-prey relationships, reproduction, aging and maturity, movement, and starvation. We also have graphic output for a visualization of the ecosystem.

DESIGN

The project primarily uses object polymorphism and separate threads to represent organisms or groups thereof. There is a central ecosystem class which contains the shared state of the ocean and stores data and functions allowing organisms to read and affect changes to the ocean.

Class Diagram:


Ecosystem
-- Contains SeaBlock[][]
-- Contains Organism Set
-- Contains a Barrier
-- Contains a Foodchain
-- Keeps track of number of simulation minutes that have passed
-- addOrganism(self, organism, location)
-- reportBirth(self, newborn)
-- reportDeath(self, organism) // organism calls this function to report their own death
-- getSeaBlock(self, location)
-- startSimulationself) // starts the simulation
SeaBlock
-- Contains members describing ecological factors of the area, including getters and setters
-- Set of organism threads located in the block
-- removeOrganism(self, organism)
-- addOrganism(self, organism)
Organism(Thread)
-- Used as an abstract class
-- Contains status variables, including location
-- performStandardAction function
-- move function
-- beEaten function
-- die function
-- Overrides run() function
Barrier
-- Used for synchronization
Location
-- Contains row and col index
-- Represents a location in the ocean
Foodchain
-- Contains dictionary of predator/prey relationships
-- addRelationship(self, predator, prey) // predator and prey should be types
-- addMultiRelationship(self, predator, listOfPrey)
-- isEdible(self, predator, prey)
Coccolithophore
-- Overrides performStandardAction function
Fish
-- Used as an abstract class
-- Overrides performStandardAction function
Shrimp
-- Overrides performStandardAction function
Manatee
-- Overrides performStandardAction function
Starfish
-- Overrides performStandardAction function
Shark
Tuna
Grouper
Herring

<u>Ecosystem Class:</u>
This class is used to store and change the state of the shared ocean and the organisms in it. Only functions from this class need be called by the end-user, as organism-specific functions will be called from the organisms themselves or by one of ecosystem's functions.

Details:
- When instantiated, caller chooses dimensions of the body of water and how many ticks of time to run the simulation for
- Contains:
  - Ocean 2D array, containing the data of the shared ocean as explained below
  - A set of all organism threads for using all at once, (e.g. for printing all data from them, eliminating all organisms in the ocean, etc.)
  - Barrier used for synchronization between ticks
  - A food chain, which defines predator-prey relationships
- Functions used by end-user or main:
  - startSimulation()

    - loadOrganisms() -- set the number of instances of each Organism type to populate the ocean with. Should be called before the simulation has started.
- Functions used by other modules (and sometimes by the ecosystem itself):
    - Function allowing retrieval of an ocean block given location of that block
    - Function to move an organism, given its current location and destination location, which cleanly moves the organism without leaving residue of its previous location.
    - Function to report that an organism has died
    - Function to report that an organism has been born
    - Function to get an organism’s neighboring organisms
    - Function that tells an organism whether a potential prey is edible
- Functions used internally by the Ecosystem class:
    - Function to add an organism to the ocean, used when initializing the ocean
    - Function to remove an organism from the set of organisms when an organism dies
    - Function to create the food chain
    - Control loop
    - Printing functions for printing stats about the simulation
    - Other maintenance functions

<u>The Ocean 2D Array:</u>
This array stores the environment of the ecosystem. It consists of many blocks, approximately representing one square meter of ocean, and stores information about that area. Each block is an instance of a SeaBlock object, which stores and changes statistics of that specific block and facilitates finding organisms in the block. See below for the description of this class.

<u>SeaBlock Class:</u>
This class represents a small, discrete chunk of the ocean. It stores data about the environment of the block and organisms in the block. Organisms use the block's data to govern their actions.

Contains:
- Members describing various ecological factors of the area, including water temperature, salinity, sunlight, oxygenation, and currents.
- Functions allowing getting and setting of the aforementioned factors.
- A set of all organism threads currently located in the block.
- A function to retrieve a list of the organism threads currently in the block.
- Functions to add and remove organisms from the block

<u>Organism Class:</u>
This class is not intended to be instantiated -- it is used as an abstract class. All organisms in the ocean should inherit from this class. This class does the work of setting up a thread for the organism and implements some default function behaviors, which subclasses can override if necessary.

Details:

- Inherits from threading.Thread
- Contains:
    - Location of the organism as an instance of the Location class. Location is an optional argument that can be passed to the constructor; if none is give, a random location will be selected.
    - wasEaten variable, used to determine if the organism was eaten and should die
    - directionXImpact and directionYImpact, which are used to determine the X and Y directions the organism should move in. They default to 0.
    - movementImpact, which is an organism's speed. It defaults to 0.
    - A reference to the ecosystem
    - survivalProbability, which indicates an organism's probability of surviving an attack on a scale of 0 to 1. A higher survivalProbability indicates a higher chance of surviving.
- Functions:
    - A run function, which overrides the run function in threading.Thread and contains the work that gets carried out when someone (in this case, the ecosystem) starts a thread. Handles cases when a thread should stop running, synchronization at the barrier, and calls performStandardAction.
    - A performStandardAction function that allows the thread to proceed with its life. This function contains the central logic of the organism. It represents what an organism does in one time tick of the simulation. The default behavior is to do nothing, so most subclasses will override this function
    - A move function that contains mobility logic. The default behavior is to move in a direction determined by the directionXImpact and directionYImpact at a speed determined by the movementImpact
    - A beEaten function that handles predation of the organism. This function is performed under a lock and may cause the death of an organism depending on their survivalProbability. It's overriden by certain Organism subclasses like Coccolithophores, which need to decrease their population instead of dying completely.
    - A die function that handles the death of the organism. The die function notifies the ecosystem of the organism's death and stops the thread from running.
- Most organisms override performStandardAction and other default functions, and many also contain other functions not defined in the Organism class, e.g. reproduce().

Fish:

- Used as an abstract class
- Inherits from Organism
- Defines behavior common to many fish. for example, handles basic logic for reaching maturity and reproduction.

- Organisms that inherit from fish must define their own reproduce() method, which is in the Fish class when conditions are right to reproduce

Foodchain:
- Essentially a dictionary where the keys represent predators and the values are a list of prey
- Any type (e.g. string, int, Organism instance) can technically be used as keys and in the value list, but we use types (e.g. <type ‘shark.Shark’> might be a key and its associated value might be [<type ‘tuna.Tuna’>, <type ‘herring.Herring’>]. This allows for the ecosystem to easily add organisms to the food chain and check if one organism can eat another.

Location Class:
Used to represent a location in the ocean. Stores a row and column index.

Barrier:
- Based off of the reusable barrier in Allen Downey’s *Little Book of Semaphores*.
- Organisms call barrier.wait() when they are done with their actions for a tick of time to wait until it is safe to proceed.
- barrier.wait() is shorthand for barrier.phase1() and barrier.phase2()
- The ecosystem calls barrier.phase1() to wait for all organisms to be done and blocked, waiting at the barrier. It then performs maintenance and prints the simulation. It calls barrier.phase2() when it is done to unblock all the organisms and allow them to carry out the next time tick.

REFLECTION ON DESIGN

Hard to choose a “best decision” we made. From some options:

1) Using a barrier class to synchronize all organisms
2) Using Python so the simulation was nicely object-oriented
3) Trajectory of project goals, what problems we chose to tackle and when

Exploring the first as the somewhat subjective “best”: in programming our simulation and threads to collectively wait at a barrier we were able to securely deal with tricky shared variable maneuvers like organisms consuming each other and removing themselves from the ecosystem. This also gave the entire simulation a very lifelike “heartbeat” in which organisms were all permitted only one “standardAction” per tick of the program. Additionally, a barrier allowed easy and accurate prints of the simulation’s state at any given time for both debugging and visualizing what was going on.

Next time we might consider using a language that isn’t as slow and can deal with more threads running at once--our program in Python was seriously limited in number of threads and hit a limit disappointingly early.

ANALYSIS OF OUTCOME

Our minimum deliverable was essentially achieved. We did include “some oceanic events” as within the scope of our minimum deliverable and we didn’t make that the focus of our work. However, some events, for example, day vs night-time effects could be easily coded and their foundations are already present (within our Seablock class is a sunlight value that plankton [Coccolithophores] essentially consume). Our maximum deliverable was essentially limitless so when asked: “where did you land between min and max deliverable?” our response is probably: farther than we expected but still nowhere near max. We ran into several interesting conflicts that delayed the expansion of the simulation. We also spent more time strengthening the foundations of our program (and implementing a nice visualization of it) rather than expanding the scope to include more and trickier mechanics (like tides, current, 3D, hurricanes, human pollution, etc…). Our focus on getting the fundamentals working well allows for rapid scaling and increasing complexity if we were to continue working on this project.

REFLECTION ON DIVISION OF LABOR

With the exception of the graphics program, which was given to one member of the team, the program as a whole was fairly cleanly divided amongst members. We were able to meet enough times to closely describe and code what the structure of the program was so that filling in the separate parts was seamless and easy. After agreeing on the overall scheme, writing subclasses that inherit from the superclasses enabled members to individually contribute without necessarily consulting or requiring the work of other members. The problem also naturally divided itself such that individual members could work on specific features, e.g. one member could work on getting the foundation of movement working and other members could later add that to the organism subclasses they were working on.

BUG REPORT

It’s hard to choose the most difficult bug because there were bugs that were easier to find but more important because they caused deadlock and there were ones that allowed the program to keep running but caused some unwanted behavior with organisms.

An example of the first kind is a bug that occured when we were first implementing organism death. An organism that exit would cause deadlock because we no longer had the correct number of threads reaching the barrier. To fix this, we called barrier.wait() in the dying thread before exiting, but after removing the organism from the ecosystem’s organism list. The reason it was necessary to call it after removing the organism was so that the next time the ecosystem set the barrier, it wouldn’t include the dying thread even if the dying thread was still blocked at barrier.wait(). This was pretty easy to find because it caused deadlock, but fixing it was crucial.

A harder to find, but arguably less important bug occurred in several organisms’ beEaten function. An organism would determine whether or not it has been eaten by selecting either True or False, where the likelihood of choosing True was 1 - self.survivalProbability and the likelihood of choosing False was self.survivalProbability. This gave the organism the chance to “escape.” The issue, however, was that if two predators tried to eat the same prey, True might get picked when the first predator called beEaten, but False might get picked when the second predator called beEaten. Therefore, the second predator would essentially bring the prey back to life. While this didn’t cause deadlock, it’s not the desired behavior. The bug was there for probably a couple of weeks before we noticed. We found it when we were adding a feature in which organisms only decremented their hunger level if the prey they ate didn’t survive. We could have found this issue earlier had we used more specific debugging output for each organism. We also could have done more thorough testing of organism behavior, checking that they acted as expected rather than just that they didn’t cause deadlock.

CODE OVERVIEW

- main.py
    - contains the main function, which prompts the user for input, retrieves it, instantiates an Ecosystem with the appropriate parameters, and starts the simulation
    - imports ecosystem and all of the organisms (used for allowing the user to decide how many of each organism to populate the ocean with)
- ecosystem.py
    - contains the definition of the Ecosystem class
    - imports all of the organisms, seablock, location, foodchain, as well as Lock, Barrier, and with_lock (a helper function) for synchronization.
- seablock.py
    - contains the definition for the SeaBlock class
    - imports Set, with_lock, and Lock
- foodchain.py
    - contains the definition for the Foodchain class
- location.py
    - contains the definition for the Location class
- organism.py
    - contains the definition for the Organism class
    - imports threading, random, location, and helper_functions
- fish.py
    - contains the definition for the Fish class
    - imports random and organism
- barrier.py
    - contains the definition for the Barrier class
- graphic_output.py
    - contains the definition for the graphicsOutput() function, which can be used to create an image of each tick of the simulation
- helper_functions.py
    - contains definitions for two helper functions, with_lock and random_pick
    - with_lock was taken from the class notes

    - random_pick was taken from Python Cookbook, 2nd Edition: http://bit.ly/1QfSwDs
- Files that contain class definitions for our organisms. All of them inherit from either Organism or Fish:
    - coccolithophores.py
    - grouper.py
    - herring.py
    - shark.py
    - shrimp.py
    - starfish.py
    - tuna.py
    - manatee.py

INSTRUCTIONS ON HOW TO RUN CODE

In the terminal, run:

```
python main.py [optional arguments]
```

Then follow the onscreen prompts to choose the number of ticks the simulation should run for, the size of the ocean, and how many of each organism to populate the ocean with.

[optional arguments]
-g f # outputs graphics to files starting frame0.jpg - frameN.jpg
-g s # generates temp files that open as the simulation runs
-g d # outputs a large csv file of all pixel rgb values

Examples:
To launch the simulation with graphics output to file, enter

```
python main.py -g
```

To launch a simulation with automatic graphics output and files, enter

```
python main.py -g fs
```

Once you have launched the simulation, you must answer some questions about the specific simulation parameters you would like to use. The program will prompt you for input and provide examples. Follow all instructions to launch the simulation.

Note:
The prompts will ask you to choose how many minutes of marine life you'd like to simulate. This is because we consider each tick of time to approximate one minute's worth of an organism's actions. In real time, each tick will last for about a second (so, a real world second approximates a minute of simulated marine life).