Report

Contents:

1. About the code

There are eleven **.cpp files** with their respective **headers** as well as three **text files** that contain the information about the different life forms and a **makefile**.
Below, there is a list of the different classes and other files with short descriptions and if applicable their functions excluding constructors, destructors, getters, and setters.

a) Makefile

The **makefile** is used to quickly compile all the classes. It creates the file "World" from all the other files.

b) Constants

Constants stores different modes that are used for testing purposes, like different debug modes that show additional information to what is displayed during a normal run, fast modes that are used to quickly advance the simulation, or different display modes that allow quicker printing.
Also additional variables that influence the dynamics of the game, but are not included in the **text files**, can be set in this class. These include for example the amount of food a newborn creature receives from its mother, the age a creature has to reach in order to become pregnant, or the characters and colors used to display the creatures.
Constants was created to have quick access to the variables which is useful when testing the program or when trying to figure out what values work best for a long lasting simulation.

c) ConsumerI and ConsumerII

These classes inherit from **Creature**. They contain a constructor that is used whenever a new Consumer is placed in the world.

d) Creature
This class inherits from **Life**. The constructor sets the values that are received from the **text files** and the values that are stored in **Constants**. There are also *Getter and Setter methods* for some of the variables that are needed in other classes as well as functions to regulate a creature's pregnancy and functions to test the current state of the creature.
Functions:

incrementPregnantTime
incrementTimeWithoutFood
incrementLifeTime
setPregnant
changePosition
computeNewCoordinates
checkWeatherFoodFound
isPregnant
isReadyForPregnant
isHungry

e) Life
Life stores information that is relevant for all of the life forms and has *Getter and Setter methods* that are used to determine the life time or position of a life form or place it in the world and determine if it is able to move around.
Functions:

incrementLifeTime
isWalkable
setWalkable

f) Map
This is the array that contains the different life forms in the form of **MapItems**. It is initialized with the dimensions that are entered by the user and there are functions that can fill it with life forms or remove them in case of death. Additionally, this class takes care of updating the smell emissions and printing the current state of the simulation. It tracks the number of life forms that is used in order to determine if the simulation should continue to run.
Functions:

removeMonster
insertMonster
deleteMonster
print
print_detail
updateEmission
test
setfirstLinesToPrint
printSeparatorS
printSeparator

g) MapItem
MapItems are placed inside the array that is constructed in **Map**. They are used to identify what life form inhabits a certain cell in case there even is any on it and they keep track of the different smell emissions on the map.
Functions:
cell_empty

h) Utils
In **Utils**, there are many useful functions that don't necessarily belong into any of the classes. There are mathematical functions that were programmed to better suit the needs of the simulation than the pre-programmed functions or functions that were not in any of the

libraries we imported, definitions that differentiate between operating systems, input and output functions, conversion functions, and custom error handling functions. The last are used to quickly determine where an error was created or help the user if there is a problem with the input.

Functions:

length
sign
sleepd
is_file
modulo
wait_for_keypressed
clear_keyboard_buffer
clear_screen
exit_error
getRandomNumber
addCoordinates
subCoordinates
isEqualCoordinates
getggT
getkgV
convertInt

i) Values

Values is used to read the **text files** and store the information so that other classes can use it. It parses the files and returns errors if there is a problem that causes the program not to function properly. Also there are some functions that can print the values in case the user wants to check if the input was parsed correctly. Those are usually not used by any of the classes and they have to be manually called in the code if they need to be used since they are not part of any of the debug options.

Functions:

readFile
print
writeV
writeCI
writeCII
testAllValues
checkValues

j) Vegetal

This class inherits from **Life**. Like the **Consumers**, the constructor creates an object with the values from the **text files**.

k) World

World is the main class that contains the most important functions of the game.
In its constructor it takes values entered by the user and creates the **map** with the specified amount of **creatures** and **vegetals**. Afterwards it starts the simulation which runs until either the **consumers** die or the *maximal amount of steps* entered by the user is reached.
Vegetals and **consumers** are placed in the world by using a function that randomly chooses a free spot. Each turn a *step* is performed and the **map** is printed to show the new state of the simulation.
There are functions that can place new life forms on the map or delete them and at the beginning of a turn each consumer is set *walkable* which enables them to make a move.
Since the world is toroidal, there is a function that uses modulo in order to keep the coordinates valid.
Every time a *step* is taken, it is decided whether a life form is dead or alive, it is checked where a creature can move if it is eligible for movement, whether it needs to eat, run away from predators, give birth, mate, or do nothing.

In order to calculate what move a creature should make, the smell function is used. Once it is determined by means of the *computeScore* function what action a creature should take, it moves to the cell that was determined to be the best and interacts accordingly. There are different functions that help with the interaction including functions used to impregnate a creature, functions used to scan the surrounding fields for food, the function that is used to give birth or the function used to kill off a creature once it dies.
The *timePassed* functions change the variables that increase or decrease every turn.
There also is the main method that takes the parameters and checks them for validity before creating the world and setting the values in Values.
Functions:

getRandomFreePosition
initializeCreature
print
performOneStep
run
testfree
timePassed
timePassed
modulo
smellAndGetBestDestination
creatureMustDie
cell_is_empty
isAConsumerI
isAConsumerI
isAConsumerII
isAConsumerII
isAVegetal
isACreature
createNewVegetal
createNewConsumerI
createNewConsumerII
setAllConsumersWalkableAndInteractable
normCoordinateToWorld
getAFreePositionAroundme
impregnate
giveBirthToABaby
interact
computeScore
findMatingPartner
findFood
cellIsFood

l) Text Documents
Consumer1.txt, **Consumer2.txt**, and **Vegetal.txt** contain the values that are used to initialize the life forms as instructed. Additionally to the values, they also contain the name of the values in order to make changing it easier. They are parsed by **Values**.

2. How to install and launch

Use *make* to compile the program with the **makefile** that is located in the same folder as the other files. This will generate a file named World that can be executed with the parameters "*height*", *"width"*, *"maximal number of steps"*, "*number of consumer 1 at the beginning*", *"number of consumer 2 at the beginning"*, *"path to vegetal.txt"*, *"path to consumer1.txt"*, and "*path to consumer2.txt"*. If the parameters that were entered are incomplete or if any of the parameters does not fit the expected value, there will be an error message.

3. Additional features

The life forms are displayed with different colors in order to make it easier to distinguish them. This works on most terminals under Linux.
Also, instead of clearing the screen before re-printing the map, the cursor is moved to the top-left corner of the screen and each line is replaced individually to prevent the screen from constantly blinking. Thus, the cursor may flash across the screen which can be stopped by disabling it. This was not done in the code, because if the program was shut down incorrectly,

the cursor would still be disabled which may cause confusion for a user that is not aware that the cursor had been disabled in the first place.
Instead of ending the simulation when both kinds of consumer are extinct, it now ends whenever either of them dies out. That was done, because whenever all of the consumer1 die, the consumer2 are quick to follow due to starvation and whenever the consumer2 die, the consumer1 have no natural enemies anymore. In either case, continuing the simulation does not make much sense anymore.
There is an option to run the simulation indefinitely – assuming that none of the creatures go extinct – by entering *"-1"* as the *maximum number of steps*.
In order to control the population of the creatures, there is a slightly more elaborate system for a creature's pregnancy. A creature has to be off a certain age in order to mate with another creature and the pregnancy lasts for a certain amount of time. When a baby is due, the cells surrounding the mother are searched and if a free cell is found, the baby is born and splits the food with its mother. This prevents overpopulation due to uncontrolled reproduction, because if the baby is born with a full stomach, the population is able to survive without food.
If there is no empty field close to the mother, the baby dies due to overpopulation. This may also cause the mother to look for a field that is further away from the herd in order to give birth, if she does not have any other needs at the time. Additionally, during the mating process only one of the participants is impregnated, while the other is set to not be eligible for another move during that turn.
Whenever a creature is pregnant, it does not need to look for a mate. Thus its only purpose becomes finding food or fleeing from enemies if applicable. Especially with consumer2 that will lead to trouble as a pregnant consumer2 can and will hunt down large amounts of consumer1 without even being particularly hungry. Thus, there is a hunger threshold that will stop a creature from eating whenever it is full. There are variables that determine the percentage that the hunger has to reach before a creature will start looking for something to eat.
Additionally the score calculation did not account for a high score on an empty field due to the field being surrounded by life forms that emit a favorable smell. Therefore, the creatures behavior has been modified to react to such situations and act accordingly by searching the cells surrounding the empty field with the highest score. If a creature is about to die from hunger, the food score also gets an additional boost. Also, there is an additional line of code that makes sure that a creature does not accidentally walk onto a cell that is already being occupied by another creature.
The number of vegetals that grow on the map is automatically adjusted to the number of consumer1 that are placed on the map during initialization.
Since the simulation was programmed using different operating systems, it will differentiate between Linux and Windows.

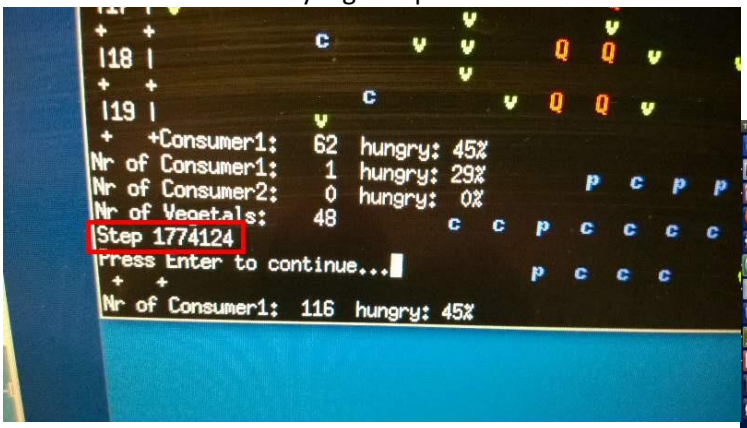
4. Personal contributions

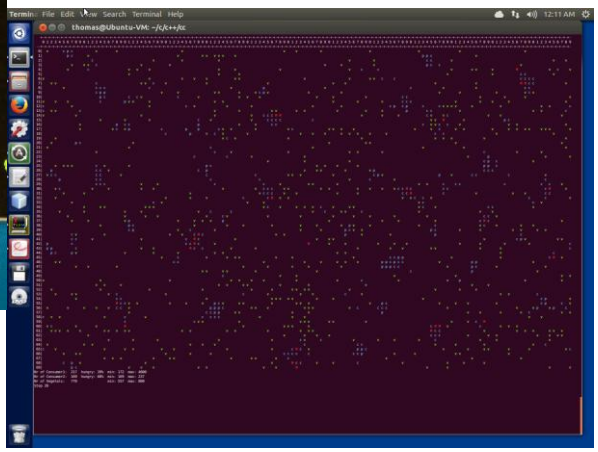
Since the version control system *"git"* was used during the programming process, it was easy to have more than one person work on the same file. Thus, it is hard to tell, who was responsible for what part of the program.
Many functions or classes were written by one person and changed by another while the comments were written by yet someone else.

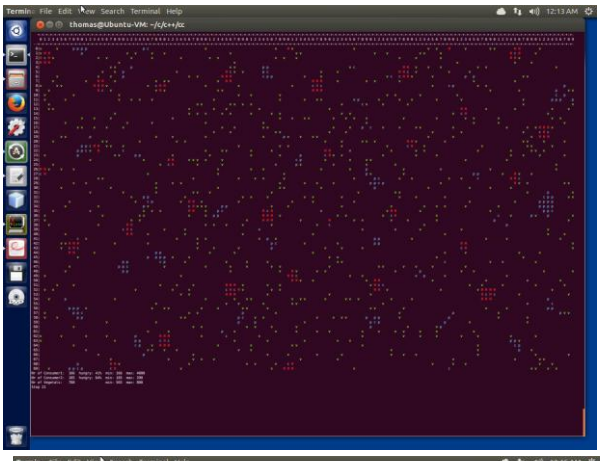
5. Tips for launching the simulation

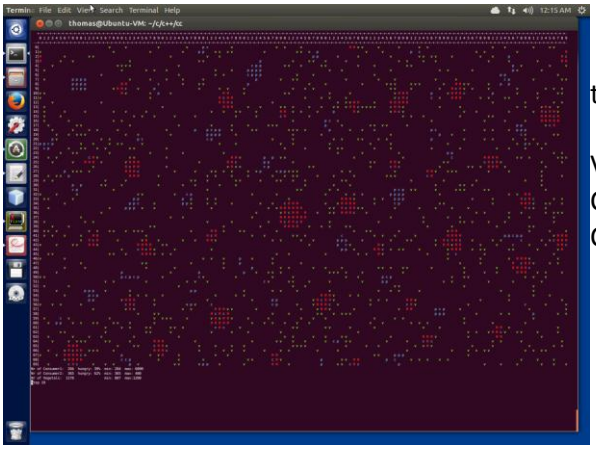
The larger the dimensions of the map are, the more likely the simulation is to run for a long time and a good distribution of Consumer1 to Consumer2 is 3:1.
Sometimes the randomly generated starting positions will turn out more unfortunate than other times, so that the simulation will terminate after a very small amount of steps. That can be fixed by restarting the simulation which will yield a different randomization. However, there is an option in the code that will turn off the randomization in case it is required to get the same results every time for testing purposes.
Finally, there are some pictures of test runs that showed interesting behavior.

This simulation had a very high step count of almost two million steps.









If the map is large enough, it is possible for more than one herd of consumer2 to exist.

Vegetals – green
Consumer1 – blue
Consumer2 – red

This project was programmed by Sarah Dreher, Julius Hülsmann, and Thomas Mattern.