Simulation Engine

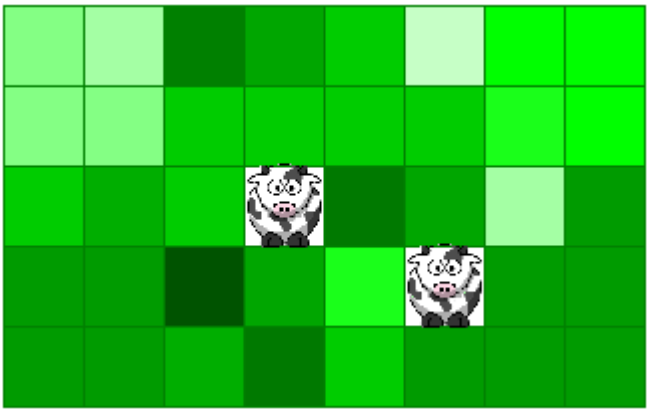
*(**UML** class diagram + working **C++** implementation required!)*

I. Requirements

R1. Introduction

There is a fantasy Green World in which there are **cows** which graze grass on a **Grass Field**. From a cow's perspective, the field is just a (rectangular) grid with a fixed size and certain amount of grass in each grid cell.

In the picture below, dark-colored cell depicts a rich cell (high grass quantity) and lighter colors depict poor cell (little or no grass in it). ***Note:*** *This coloring scheme is just for illustrative purposes and is not mandatory. The user interface may be* ***Console*** *or* ***Graphical*** *UI, see* ***R7.***



Note 2: Feel free to use maps (2D arrays) of your own. Make sure grass is smoothly distributed (no sharp changes). You can check out ***bivariate multimodal*** distribution just as an example.

R2. Cow's life

For now, cows are born in a Farm. The grass field has one Farm that is outside the grass land itself. Currently, a Farm produces 1 cow each ***CBRI*** number of simulation steps (***"simsteps"***). ***CBRI*** (*Cow Birth Rate Interval*) is set when the Farm is being constructed (For orientation, try with 20). Assume it stays the same for the entire Farm's existence.

The newborn cows are put into a random cell of the grass field.

Each cow has ***age*** that starts from 0 and is incremented each ***simstep***.

Each cow has some ***energy*** level. Energy is ***incremented*** when a cow grazes – by the *quantity* of the eaten grass. Energy is ***decremented*** when a cow moves – by ***10%*** of cow's ***age*** each ***simstep***. A newborn cow has 0 energy.

Cows live at most ***LE simsteps***, (***LE*** – Life Expectancy - is a constant that needs adjustment, try with 200 at first), but they can die earlier if their ***energy*** is depleted entirely.

R3. Grazing process.

While grazing, cows do the following:

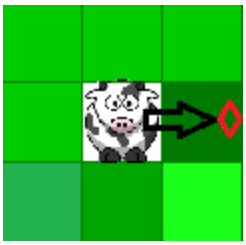
1. Eat ***some*** grass from their current cell. This diminishes the grass quantity in the cell! Each cow tries to graze as much as possible, but the maximum quantity of grass it can eat in a single step is ***(age + 3)***.
2. Look around at nearby cells and ***decide*** their next move (select next cell). Cows always move exactly 1 cell at a time, horizontally, vertically or diagonally. The ***decision making*** is described in ***R5***
3. Move to the selected cell, and so on.

R4. There can be ***multiple cows in a cell*** at any given moment. However some of the cows might eat less than expected food from that cell, since other cows might have eaten most of the grass there first.

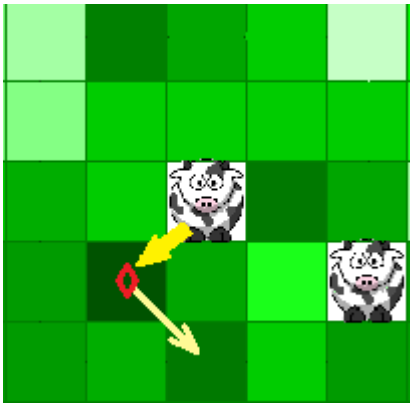
R5. Cow decision making.

Different cows might have different ***planning methods*** of how to select their next cell. Cows sometimes change their *planning method* (see ***R6***), but usually not on every move.

1. A cow may plan its movement just by looking around at the closest cells (within *radius 1*), then select the cell with the highest grass quantity and move to it (*"simple planning"*). On the picture below, a red diamond marks the selected cell.



2. An improved planning type might be to look further around with *radius 2* and choose the first cell in the best *2-step route* (with ***maximum sum*** of grass quantity $\boldsymbol{q = q_1 + q_2}$). This is called *"simple-2 planning"*
   ***Note**: Of course, the next step might turn out differently, because of the new neighboring cells the cow discovers.*



3. * Another planning type is to choose by ***gradient***. The cow analyses all possible ***radius-2 paths***. For each path the difference between the two quantities is calculated:
   $\boldsymbol{gradient = q_2 - q_1}$
   Then the cow selects the next cell on the path with maximum ***gradient***.

R6. Changing the planning method

1. Cows are born with *"simple planning"*. When they grow up (say, at age 50) they change their planning to *"simple-2 planning"*.
2. * If its ***energy*** drops below the necessary energy for performing ***3*** steps, the cow desperately changes the ***planning method*** to a randomly chosen one.

R7. User Interface

The user interface can be either ***Console*** or ***GUI***.

* Insure your design (and code) against changes in UI type.

The User Interface shall allow at least the following options to the User:

1. ***Configure Simulation:*** User inputs at least ***CBRI and LE constants*** (optionally also the other "magic numbers" from requirements)
2. ***Generate or Load maps.*** Alternatively, at least there shall be 1 hard-coded map (entered as 2D array), but in that case distribute the grass

carefully (not arbitrary distributed, but form dense islands with less grass between them and smooth transitions)

3. ***Run N steps*** (N entered by the user)
4. ***Display current statistics****:*
    a. Each cow age, energy and current planning method
    b. Grass field state (grass quantities, cows as symbols on their current locations)

II. Expected changes

While deciding in which part of the design more flexibility is needed, keep in mind the following expectations:

1. Customers may add requirement for support of multiple Worlds soon!
2. Multiple (separated) grass fields (in each World!) will probably be requested, too.
3. Other species of animals may be added in the future. Probably other grazing animals (i.e. sheep) behaving similarly, but also other types i.e. a bear, rabbit or mouse which have different types of food and completely different possible planning methods
4. Other types of land can be expected, i.e. forest, desert, etc. (and maybe even water?)
5. Transfer of animals between land parcels might also be requested. Some animals may migrate, others can be sold within the same world.
6. Grass can grow on land according to some laws, among which regular presence of cows (and other animals) play important role.
7. ****** Instead of special farms, animal ***breeding*** can be expected as requirement from the customers at some point. Check ***Clone / Prototype*** design patterns and think about passing combination of current ***planning methods*** of the parents to the offspring, instead of always starting with “simple planning” (see ***Genetic Algorithms***). Just as an idea: offspring may receive randomly one of parent's planning method, or have combination of its parents' method parameters (i.e. radius of search).

III. Guidelines

- Of course, it is best if you achieve fully designed and working program. However, if your time is not enough, then it is better to have simpler, but stronger Object-Oriented design and working implementation, than more classes and interfaces with questionable relations and non-working program.

Saying this, at least 3 patterns must be applied (correctly!) even in simplest solutions.

- Use iterative approach during design until you reach best object-oriented qualities and even working “proof of concept” implementation. Optimize ***after*** that.
- Technical correctness will also be evaluated (no crashes, no memory leaks, stable C++ fundamentals, etc.)