1 Project description

You will create a multithread application to simulate life of autonomous worms. All the worms will be placed within the same environment and will interact with each other. The goal is to simulate an ecosystem with diverse types of individuals living inside.

TODO: picture of worms / screenshot of app?

All worms will live on a ractangular ground, where hitting an edge results in appearing on the other side. Similar to the *Snake* game available on old Nokia phones. In other words, the environment is a torus projection on a 2D screen.

TODO: picture of scrossing the edge / series of pictures showing edge crossing.

The design of the core functionality with two types of worms has been prepared, together with an implementation with a few missing functionalities. You will need to implement them to make the application run and to satisfy predefined test cases and later you will prepare your own design for the extensions of the application.

1.1 Worms movements

Board is a square grid where all the worms are placed. Each worm is entirely within one of the fields (i.e. $(1, 1)$) and has its own identifier which is visible in debug mode. A worm may choose any adjacent field to move, but if on the desired field is already a worm placed then it will be eaten by the one taking the step. It means that when two worms are in adjacent fields and both deside to eat each other then the winner is the one whose thread will execute this first.

1.2 Interactions between worms

There will be two types of worms (you will make additional types in *extended functionality*):

- *Lazy Worm (black)*
- *Hunter Worm (red)*

1.2.1 ***Lazy Worm (black)***

This worm randomly travels around. Complete random movements don't look realistic, so it has some direction he's currently moving (i.e. left which means going directly to the left border of the application window) and with probability 75% it keeps that direction. It turns left (relatively to the current direction) in 12.5% times and right (relative) with 12.5% chance.

1.2.2 *Hunter Worm (red)*

This worm choses a goal on the board and moves towards it. Of course possible targets are constantly moving around, so the goal coordinates need to be updated. Before each step he is locating the desired item on the board, updates the coordinates he's heading to and then performs a move towards it. If the goal (i.e. other worm) disappeared (was eaten) then a new goal is chosen which is closest in distance. When nothing is left on the board he choses to reach a random field on the board.

2 Technical specification

2.1 Sources

Project package contains a directory with source codes, makefile and tests. It should follow best practices of a multithread application and a style guidelines.

Source files with main logic are available in `sources/`, all tests are within `tests/`. Your first modifications will need to be made in `sources/worm.cpp` and `sources/board.cpp`.

2.2 Debugging

Tha application has debug mode which draws all the worm ids on the board instead of plain circles. You may turn it on by changing the very last parameter of call function `createAndRun` to `true` within `main.cpp` file. You may also want to modify other parameters to change the size of the board (cureently it is 20x20 fields).

3 Basic functionality

List of files that need to be modified for the whole application to run:

- `sources/worm.cpp`
- `sources/board.cpp`

List of functions to implement:

- `move` function that based on `currDir_` which is the current worm direction (`LEFT, RIGHT, UP` or `DOWN`) performs one step towards it by modifying appropriate class variables,
- `addWorm` function that should add new worm into the internal board at given coordinates, give it an appropriate id (see `nextId_`) and run the thread (*hint:* `emplace` method will be useful),
- find appropriate function and implement joining the threads.

List of basic functionality needed to add:

- **synchronization** – currently no synchronization is implemented, so threads (worms) may create race conditions on the board (they kill each other and update their location),
- **error handling / exceptions** – currently all the errors are printed to the standard output using `std::cout` and some are not handled at all (like giving incorrect board size which should definitely greater than 0).

4 Extended functionality

When finished with basic functionality you will need to extend your application to become interesting to watch. It will engage you to shape your code in a way that fits the core design, but extension's design will be entirely up to you.

Extensions (detailed explanation in a separate section of this document):

- Implement bonuses (small items like cherries) that appear randomly on the board,
- Save function that allows to dump current state into a file (and load it during the next run),
- (big) Implement behaviour schema for a worm, basically artificial intelligence,

4.1 Bonuses

Once in a while on the board may appear additional item – bonus – which will be desired by the worms. It doesn't need to actually give any points or anything, but worm's logic may prioritize to collect it and make the whole ,,show" more interesting.

Notice: This task may require to get familiar with `graphics.*` and *gtkmm* library for drawing. It would be sufficient to draw circles of a different color (i.e. blue) to make it distinguishable from other objects.

4.2 File dump

The whole board may be dumped into a file (together with worms locations) and retrieved during next run. Since it is not possible to dump exact threads and their contexts it is enough to create a new worm of a correct type within a place of a dumped one.

4.3 Artificial intelligence

Provided worms have very limited ,,thinking" – it would be interesting to see actual ,,smart" worms who use some clever ideas of your own to make this battle spectacular.

Good luck!