C++ programming course: project

Deliverables

Deadline for the project is January, 21st 2013. Please send by email to your instructor, TA and the course lecturer (instructor and TA of the exercise class that you belong to) a compressed archive (zip, tar.gz, ...) containing your project (source code, build files, makefile etc) and your report. Please send only one email per group.

Preliminaries

This year the course requires the completion of a project (it will count for 25% of the final grade). The project has to be realized in C++ and we strongly suggest teamwork. Teams can consist of a maximum of 4 members.

The goal of the project is to realize an implementation of the snake game, which is described below in this document. Deliverables for the project are:

- The source code corresponding to your implementation
- A report describing your work: description of the architecture of your program, architecture and design choices, data-structures used and reasons behind their use, how you decomposed the problem functionally and solved each part, etc

For grading the project we will first consider whether or not your program is correct with respect to the specifications given below. Almost as importantly as correctness, we will also consider how your program is architectured and designed, if its logic is sound. We will look at your choices of data-structures, whether they are appropriate or not. We will consider how readable your code is and if it can be easily maintained by others. We will look at whether you are using comments adequately, whether your naming conventions (variables, functions, classes, files) are consistent through the project. The list of points that we will take into account when evaluating your solution is given here.

In addition to the description given below, we are also providing the following:

- Binaries for solaris 11 and OS X, corresponding to a possible implementation of the snake program
- Two examples of levels: "level01.txt" and "level02.txt"

The Snake game.

The game

The snake game is an old video game that was already present, for example, in old versions of MS DOS. In this game, a snake evolves in a world delimited by walls and tries to eat food on its way. Each time it eats food, its length increases and the game-score also increases. The ultimate goal is to have the highest possible score. The game ends when the snake collides with something: either a wall or its own body. In the game, the snake is piloted by a human player with the keyboard, but in this project we will consider a simple AI for driving the snake.

The rules

The version of the snake game that you need to implement should follow the rules below:

- The snake moves by extending its head in the current direction (speed) it is moving and pulling its tail accordingly
- When the snake eats one piece of food, its length increases by one unit
- A new piece of food is randomly placed on the board every time the snake eats one
- The snake wins if it eats 5 pieces of food
- The snake looses if:
    - It crashes into a wall
    - It crashes into itself
    - It goes outside of the board

Additionally, please assume that during the game the speed of the snake can only change in direction but not in magnitude. At the beginning, the initial speed should have unit norm and stay with unit norm during the course of the game.

The AI algorithm

In contrary to the original game where the snake is controlled by a player, we will use a simple AI to pilot the snake. The snake is driven by the following algorithm:

- Try to move the snake according to its current speed
- If this move brings the snake to an invalid position OR with a given probability of 10% then:
    - Try to make the snake turn left or right
    - If it can turn left and can not turn right, then turn left
    - If it can turn right and can not turn left, then turn right

- If it can turn left and can turn right, then select randomly (with a probability of 50%) one of the two directions
- If it can not turn left and can not turn right, then continue with the current speed.

This algorithm corresponds to a random search. It is primitive (and very slow). It can be improved in several ways but it is not the goal of this project.

In order to decide if you can accept a decision with a given probability, e.g. 10%, you can use the following code:

```
// ----- BEGIN SAMPLE CODE -----
#include <cstdlib>
#include <cassert>

// Generate a random floating point number in [0,1] using an uniform distribution
double GenerateRandomUnitDouble() {
    double uniform_random = static_cast<double>(rand()); // double in [0, RAND_MAX]
    return uniform_random / RAND_MAX;
}

// Accept a decision with a given probability in [0,1]
bool RandomAccept(double probability) {
    assert(probability>=0.0 && probability<=1.0);

    double unit_sample = GenerateRandomUnitDouble();
    return (unit_sample < probability);
}
// ----- END SAMPLE CODE -----
```

Functional units

- Game The game unit needs to keep track of the snake, the AI and the world. It needs to provide functions for loading new world map from files, initialize a new game, display intermediate states.

- Snake The snake unit needs to encapsulate the snake representation (i.e. its current position and its speed) and functions to control its moves.

- AI The AI unit needs to contain function to find the next move that the snake should take following the AI algorithm given in section 2.3 above.

- World The world unit is in charge of holding a representation for the world in which the snake evolves and functions to manipulate it.

File format for the levels

Levels corresponding to worlds where the snake evolves should be loaded from a file. Either as a command line argument to your program or through some simple text user interface. A level should contain the following information:

- The width and height of the world
- The initial position of the snake (at first the snake should be one unit long) and its initial speed. A speed is a vector described by its components in the x and y direction. We shall assume that the initial speed vector is always of unit norm and corresponds to one of the following directions: +x, or -x, or +y, or -y. I.e. it has always one component equal to 0 and the other equal to +/- 1.
- A series of symbols (also called tiles) representing the world. Each symbol can correspond to either: an empty space, a wall, food or a part of the snake

The file format to be used is as follow:

- First two integers specifying the width then height respectively
- Followed by two integers specifying the initial velocity: velocity along x, then velocity along y
- Followed by a list of chars that correspond to the tiles in the game.

An example of input file is:

```
15 11
1 0
###############
#*           *#
#      #      #
#      #      #
#      #      #
#      #      #
#      #      #
#             #
#         o   #
#*           *#
###############
```

The tiles used in the file format are described below:

- ' ': represents an empty tile
- '#': represents a wall
- '*': represents food

- 'o': represents a tile occupied by the snake

Output

Your implementation should writes on the standard output at each time step the current state of the world, i.e. the walls making the world, the current position of the snake, the position of the pieces of food and the total number of pieces of food eaten so far. A copy of the output obtained during the execution of the provided binary is shown below:

```
// ----- BEGIN OF SCREENSHOT -----

Food eaten: 0
###############
#*           *#
#      #      #
#      #      #
#      #o     #
#      #      #
#      #      #
#             #
#             #
#*           *#
###############

// ------ END OF SCREENSHOT -----
```

The simplest way is to simply write on standard output the world state after each move of the snake. If you want to achieve an effect similar to the provided binary, you can force the screen to be cleared by using the shell command `clear` (or `cls` for MS Windows shell). To call a shell command within a C++ program, you can use the function `int system(const char* arg)` from the libc. Example of usage:

```
// for clear the standard output where the C++ program writes:
system("clear");
```