Snake - General information

This game runs in your browser (client side), no server is needed. To start the game simply open [snake.html].

Game rules

File structure

The game

- [snake.html] and [snake.css] are the main game components, running the actual game.
  - [game.js] is the starting point, creating the board and adding the snakes to the game.
  - In [snake.html], the starters are defined (see Add a new AI).
- Helper classes are implemented in [snake.js] and [board.js].
- In sub folder *Team* you can find the snake AIs, one *js* files per team.

Add a new AI

What do you have to do to add your snake AI:

1) Create your on *js* file in sub folder *Teams*
2) Add your new *js* to the [snake.html] into the array *SnakeAIs*. Each list entry has to hold an object with properties *color* (a string which can be intepreted by canvas), *team* (as string with your team name) and a *moveFunction* (a reference to your AI function executed on each step).
3) Implement the AI function: your function needs to take the *snake* object as argument. It does not have to return any value. Inside the function your are not allowed to manipulate the snake state, but you have to call methode *turn* exactly zero or one time.

Tips on implementing the AI

With this repo, there comes a very simple exampe AI, see [Teams/team_example.js].

The actual functions called by the game are:

- team_example_runToFood(snake)
- team_example_tryToSurvive(snake)

To encapsulate the methods, all methods are part of a helper class *AI_Example*. This is not necessary, but avoids functions with the same name as used by other teams. Similar, if you want to save data (e.g. to store something over multiple steps, such as making decision and remembering that for the next step), you can store the data on the snake object itself. As an example, the two methods *setData* and *getData* are part of the example class.

Short summary of the example methods

1) **runToFood**

   - First it obtains the pixel (positions) for each possible move, i.e. move straight, turn left and turn right.
   - For each move, it checks, if this is actually possible using the *checkCollision* methods of the snake.
     - If there is no possible move, well, then we are screwed up (just return and face death in the eye).
   - Then, with helper method *shortestWayToFood*, we find the move, which brings us closer to the food target.
     - This method returns the global direction, so up (“U”), down (“D”), left (“L”) and right (“R”).
   - We then convert the global direction into a command for the snake using *weWantToGo*, which is acutally an array.
     - Example 1: we are going right, but want to go up: then the command will be “turn left” (“L”).
     - Example 2: we are going right, but want to go left: then the command will be “turn left” (“L”) or “turn right” (“R”), since for are u-turn we can go either way.
   - With the list of command we want to execute, we remove those commands which are not possible (see second step).
   - In the end, we execute one of the remaining command selected randomly. If the selected command is “S” (going straight), we do nothing, else we use snake’s methode *turn(command)*.

2) **tryToSurvive**

   - We start very similar to the other methode: find all possible moves without collisions.
     - Since we want to prefer the straight command (to not roll up the snake like a snai), we add the straight command multiple times (as often as the board is wide, see *dim.w*).
   - Then, we select one command by chance. In case of “S”, do nothing. In case of “L” or “R” execute method *turn* accordingly.

- One special case: there is condition check for food distance and snake health.
  - Since snake will starve in this game and the movement is completely random, the snake will starve.
  - To avoid starvation, the snake changes its behaviour when passing food (with distance of 20 pixel or less) or when healt is below 20%.
  - In that case, we simple swith to the first method *runToFood*.
  - Setting the run mode with *setData* is just for demonstration purposes and not used anywhere.

API useful for your API

In general all data is accessable via the *snake* object handed over to the AI function.

- *snake.game* give you the *Game* object.
- *snake.game.board* give you access to the *Board* object.

In the following, if something is returned (or taken) as **position**, an object with structure *{x: number, y: number}* is meant. A **direction** means the global direction, one of up (“U”), left (“L”), down (“D”) or right (“R”). A **command** could be one of the following: turn left (“L”) or turn right (“R”). If you want to go straiht, do not give a command. **pixel** refers to the stats of given coordinates on the board, which can be *0* for a free pixel, *1* for a pixel with food or a *snake object* when a pixel is occupied by a snake.

Some functions help to find the correct information:

- *snake.getPos() : position* returns the position of the snake (head) as position.
- *snake.getScore() : number* returns the score of the snake, basically just the snake length.
- *snake.getStepCount() : number* returns the total number steps a snake has taken.
- *snake.getHealth() : number* returns the health of the snake in percent. If the health runs out, the snake starves. The health completely refills when eating. Each step reduces the snake’s health.
- *snake.getDir() : direction* returns the global direction of the snake.
- *snake.turn(command)* is the main functions you call from your API. Once per step, you can decide to turn once left or right.
- *snake.turnDir(command) : direction* gives you the global direction if you would turn. Example: if your snake is going in direction “R”, then the method *snake.turnDir(“L”)* will return “U”).
- *snake.nextPosInDir(direction) : position* gives you the theoretical new position if the step is executed. If no direction is given, the current snake direction is assumed. This function is helpful together with *snake.turnDir(command)* to get the next position if turned into a certain direction.

- *snake.checkCollision(position) : bool* returns true if the given position would cause a collision (including with board boundaries or snakes).
- *snake.game.board.getFoodPos() : position* returns the position of the food item on the board.
- *snake.game.board.getPixel(x: number, y: number) : pixel* returns the content of the given coordinates. The return values can be *0* for a free pixel, *1* for the food or an *snake object*. If the return value is a snake object can be checked with the next method:
- *snake.game.board.isSnake(pixel)* : bool: returns true, if the given pixel is a snake object.

All other methods or properties should not be called or modified by the AI (cheating).