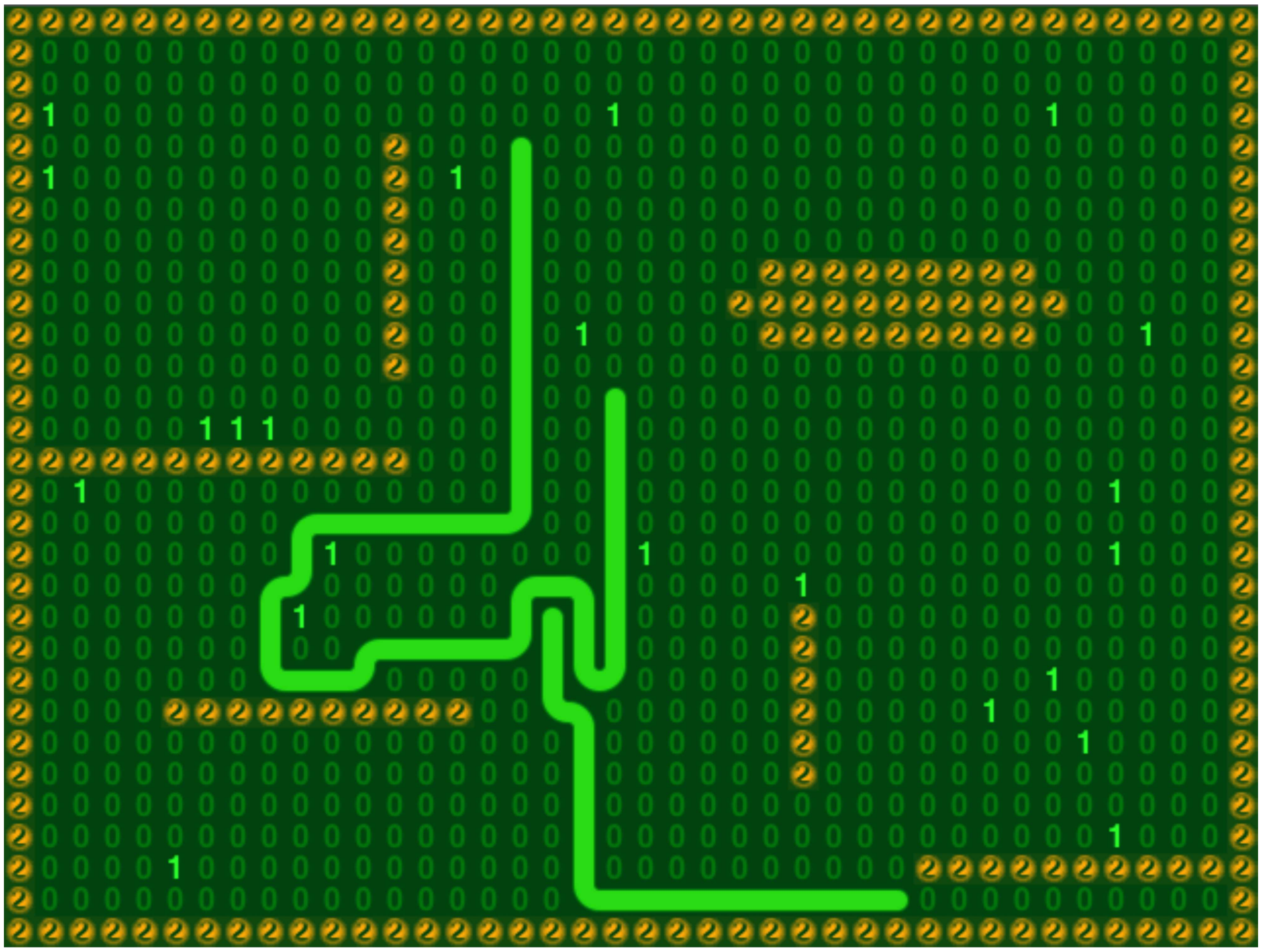
Machine Problem 5: Snake Game

"Snake" is a classic arcade game. From Wikipedia:

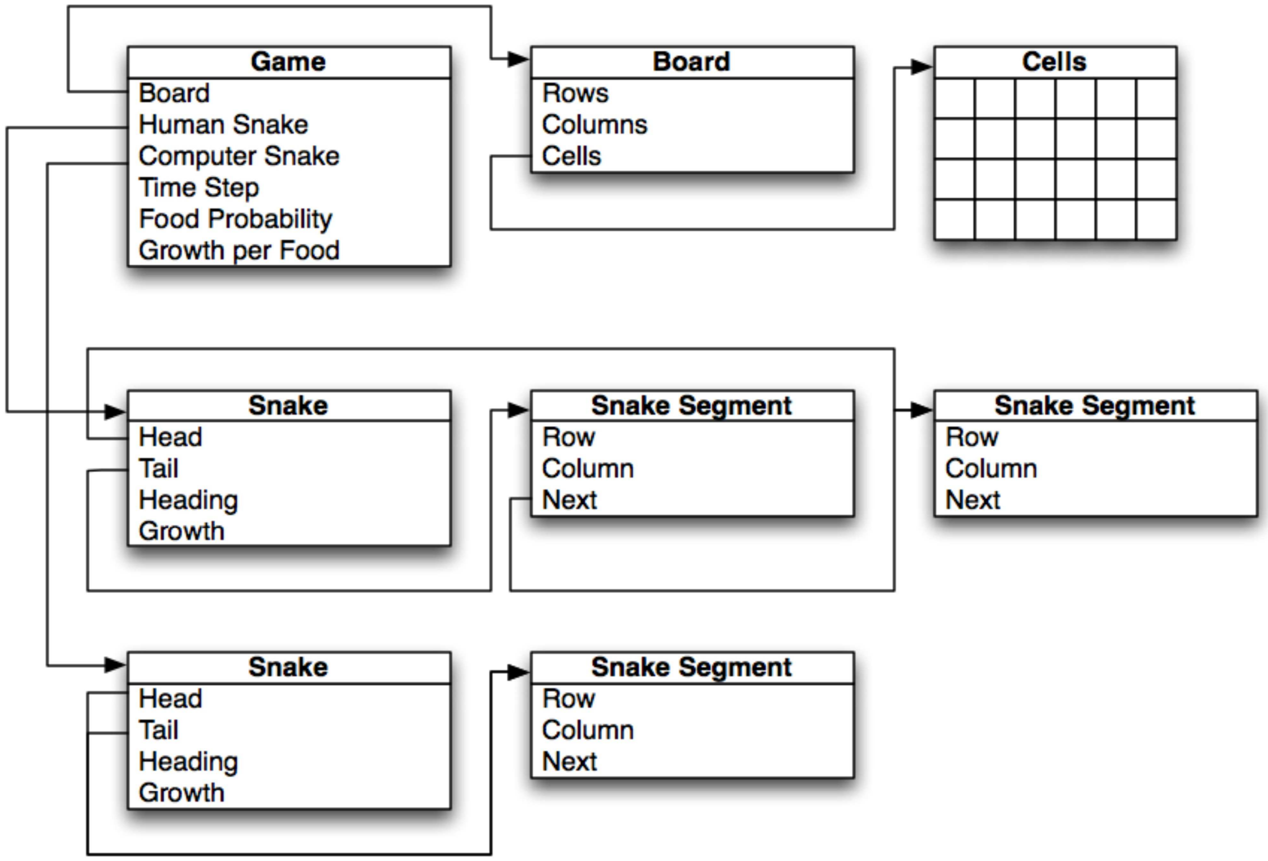
> The player controls a long, thin creature, resembling a snake, which roams around on a bordered plane, picking up food (or some other item), trying to avoid hitting its own tail or the "walls" that surround the playing area. Each time the snake eats a piece of food, its tail grows longer, making the game increasingly difficult. The user controls the direction of the snake's head (up, down, left, or right), and the snake's body follows. The player cannot stop the snake from moving while the game is in progress, and cannot make the snake go in reverse.


Screenshot of the digital themed snake game

For this MP, you will be implementing a digital themed version of "snake". Two snakes play the game, a human snake controlled by the keyboard, and a computer snake controlled by a recursive search algorithm. The game is played on a 2-dimensional grid of cells, where each cell is either 0, 1, or 2. Snakes can pass through 0s freely, and eat 1s for food. Since the snakes only understand binary, they are very afraid of 2s and cannot pass through them.

Background

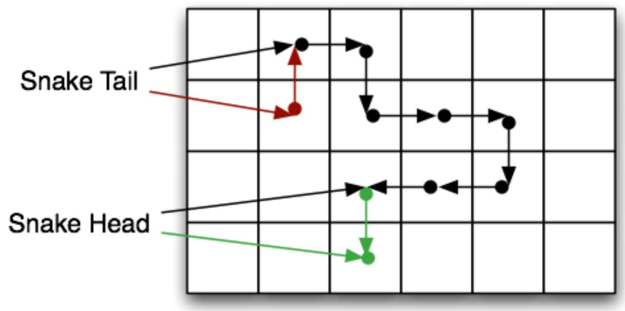
The game is modeled with a hierarchy of data structures, as shown in the figure below. Each block in the figure represents a region of dynamically allocated memory. Building the in-memory game model is done by starting with the data structures at the bottom of the hierarchy, and building upwards, making sure to initialize pointers in the higher level data structures to point to the appropriate lower level ones.



Game data structure and substructures. In this example, the game has a 2-segment human snake, and a 1-segment computer snake.

Snake data structure

The snake data structure is similar to a singly-linked list, except that **the snake head is the linked-list tail and the snake tail is the linked-list head**. The figure below shows how the snake segments, by being associated with a particular row and column on the board, model a snake. Each segment has a next pointer which points in the direction the segments move. These next pointers allow the head and tail of the snake to be advanced easily.



Snake data structure. Red nodes and pointers represent the state of the snake before the tail moves. Green nodes and pointers represent the new state of the snake after the head moves.

Computer AI

The computer AI is based on a recursive search algorithm which finds the closest food and follows a path to it. At the core of this algorithm is a function which determines the distance to the nearest food. This function is like a recursive maze search algorithm, where each recursive step is to search all of the cells adjacent to a given cell. In order to determine distance to food, however, this algorithm must also mark entries in a distance map with the distance traveled thus far. Also, in order to avoid wasteful searching, the algorithm must also consider any cell with a distance less than or equal to the distance traveled to be a barrier, because searching it would result in the same or worse results.

2	3	4	5	6	7
1	16	15	X	X	8
0	17	14	X	X	9
19	18	13	12	11	10

Distance map after first 19 iterations (all pushes of recursive function onto runtime stack)

2	3	4	5	6	7
1	16	15	X	X	8
0	**15**	14	X	X	9
15	**14**	13	12	11	10

Some refinement after searching through the deep branch we started with.

2	3	4	5	6	7
1	**2**	**3**	X	X	8
0	**1**	**2**	X	X	**7**
1	**2**	**3**	**4**	**5**	**6**

Distance map after completion of the search; 5 is the closest distance to food.

Implementation

Your task is to implement all functions marked `Implement me!` in *file.c*, *game.c*, and *ai.c* To maximize partial credit awards, we recommend that you follow this order of implementation:

MP5.1 due Wednesday 4/20

- (30 points) Implement `create_game` in *file.c* to get board loads working. You'll need to implement a few helper functions in game.c to simplify this function implementation. This includes `create_board` and `board_cell`. Run the program to see the boards that you're loading.
- (5 points) Implement `destroy_game` in *file.c*. You'll need to implement a few helper functions in game.c to simplify this function implementation. This includes `destroy_board` and `destroy_snake`. Make sure the file tests pass with valgrind.
- (5 points) Implement `randomly_add_food` in *game.c*. Run the game, and verify that food is sporadically added.

MP5.2 due Wednesday 4/27

- (20 points) Assume that all snakes go north, ignore food, and implement the linked-list style `append_snake_head` and `remove_snake_tail` functions, as well as `update_snake_head`, and `update_snake_tail`. Observe snakes moving north and crashing.
- (20 points) Rescind the aforementioned assumption and take food into account, and get the human controlled snake working.
- (10 points) Implement the computer AI by completing the functions in *ai.c*, observe the computer controlled snake chasing food.

The maximum score for this MP is 100 points, so the remaining points will be given for style:

- (10 points) Code conforms to ECE 190 coding conventions

Submission

MP5.1

To submit your code for MP5.1, run `make test` to confirm that your code compiles and that functions that you think are completed are passing the tests. Then run in the *src* directory:

```
handin --MP 5.1 file.c game.c
```

MP5.2

To submit your code for MP5.2, run `make test` to confirm that your code compiles and that the functions that you think are completed are passing the tests. Then run in the *src* directory:

```
handing --MP 5.2 file.c game.c ai.c
```

Grading

Your functionality grade for this MP (90 points) will be determined by automated tests run on your MP5.2 submission. This means that fixing bugs in your MP5.1 code before the MP5.2 submission will be to your advantage. Your MP5.1 submission will not be graded directly, but it will have an effect on your grade in the following situations:

- Late submission will result in your functionality scores for the corresponding MP5.1 parts being reduced according to the ECE 190 late policy.

- Submitting incomplete or no work for MP5.1 will result in half credit for **all** parts that you were supposed to have done by the MP5.1 checkpoint.

Building and testing

Simply run `make` to build the game, which will produce the game executable in the *bin* directory. The command line usage for the game executable is:

```
snake <level_file>
```

You can use one of the example level files in the *levels* directory for `<level_file>`

Automated test programs are provided to help you check your code. To run all tests, simply run `make test`. To run a particular test, such as *test_file*, run in the *tests* directory:

```
make test_file
./test_file
```

Valgrind and gdb can be used with these test programs to help you track down bugs.

Project Layout

Makefile
: Defines rules for building and testing the project. For most tasks, this is the Makefile to use

Makefile.inc
: Build configuration file; defines compiler program and flags.

src/

- **Makefile**
  - Defines rules for building the program
- **main.c**
  - Contains entry point and "glue code" to connect the pieces of the program together. You should not need to modify this file.
- **game.c, game.h**
  - Defines the game data structures, and functions for creating, updating, and destroying game elements
- **view.c, view.h**
  - Implements the game view windows, associated functions that draw the game elements, and keyboard input handling. You should not need to modify these files.
- **file.c, file.h**
  - Implements snake game loading from a level file.
- **ai.c, ai.h**
  - Implements the computer snake's move selection algorithm

bin/

- **snake**
  - Your program

levels/

README.txt
Format specification for the level files
random.txt, small.txt, walls.txt, maze.txt, simple.txt, spiral.txt
Example level files

graphics/
0.bmp, 1.bmp, 2.bmp, blob.bmp, e.bmp, ew.bmp, n.bmp, ne.bmp, ns.bmp, nw.bmp, s.bmp, se.bmp, sw.bmp, w.bmp
Sprite images used by the game to draw the contents of the view window

tests/
Makefile
Defines rules for building the tests
test_utils.c, test_utils.h
Utility library used by the test modules. You should not need to modify these files.
test_file.c
Tests for the file module
test_game.c
Tests for the game module
test_ai.c
Tests for the AI module