Lab 9 : Writing Games with Allegro 5

CSH : Initiation au C et au shell
Première année

The goal of this lab is to teach you how to write a simple game using the allegro library. We use the version 5.0 of the library (which is still alpha at the time of writing). Please visit the project website at `www.liballeg.org` more information and downloads (the soft should be already installed at ESIAL).

In particular, the API documentation is available from : `http://docs.liballeg.org/`

▶ Partie 1: Project setup

Before actually coding your game, you want to setup your project. As most library, Allegro requires some specific function calls to get initialized. The following application template is provided to ease your work ; we will put some flesh on the bones later. Most of that template should be self-explanatory, but that's perfectly OK if you don't understand each line, we will explain the obscure ones later on.

Simple Allegro 5 game template

```c
#include <allegro5/allegro5.h>        /* Load the allegro headers */
#include <allegro5/a5_color.h>
#include <stdio.h>                    /* Load some common headers */

struct {                              /* This structure contains every globals related to the display */
  int FPS;
  ALLEGRO_EVENT_QUEUE *queue;
  ALLEGRO_COLOR background;
} ui;

void draw(void) {                     /* Draws all you want on the screen (still empty) */
  al_set_target_bitmap(al_get_backbuffer());            /* we use double buffering */
  al_clear_to_color(ui.background);
  /* More drawing goes here */
  al_flip_display();                                    /* More on double buffering */
}
void run(void) {
  ALLEGRO_EVENT event;
  bool need_draw = true;

  while (1) {
    if (need_draw && al_event_queue_is_empty(ui.queue)) { /* don't redraw until the queue is empty */
      draw();
      need_draw = false;
    }

    al_wait_for_event(ui.queue, &event);                  /* block until an event arrives */

    switch (event.type) {                                   /* handle the event */
    case ALLEGRO_EVENT_DISPLAY_CLOSE:                       /* window closed */
      return;

    case ALLEGRO_EVENT_KEY_DOWN:                            /* self-explanatory */
      if (event.keyboard.keycode == ALLEGRO_KEY_ESCAPE)
        return;
      break;

    case ALLEGRO_EVENT_TIMER:                               /* The timer elapsed */
      need_draw = true;
      break;

    }
  }
}
int main(void) {
  ALLEGRO_DISPLAY *display;
  ALLEGRO_TIMER *timer;

  if (!al_init()) {                   /* initialize Allegro */
    printf("Could not initialize Allegro.\n");
    return 1;
  }
  al_install_keyboard();              /* Hey allegro, we'll use those peripheral */
  al_install_mouse();

  display = al_create_display(640, 640);
  if (!display) {
    printf("Error creating display.\n");
    return 1;
  }
  al_set_window_title("your game");

  ui.background = al_color_name("white");
  ui.FPS = 60;
  timer = al_install_timer(1.0 / ui.FPS); /* make sure that a timer will elapse often enough for the FPS */

  ui.queue = al_create_event_queue();     /* setup the event queue */
  al_register_event_source(ui.queue, (void *)al_get_keyboard());
  al_register_event_source(ui.queue, (void *)al_get_mouse());
  al_register_event_source(ui.queue, (void *)display);
  al_register_event_source(ui.queue, (void *)timer);

  al_start_timer(timer);

  run(); /* run my application */

  al_destroy_event_queue(ui.queue);

  return 0;
}
```

★ Exercice 1: Setting up your game

▷ **Foreword :** If you want to do the lab at home, you need to download the Allegro library and install it. We use the version 5 although it was not released yet. Grab from `www.liballeg.org` the higher available 4.9.x version, and install it. I tested the v4.9.11. With this version, I had to add the following to the linking command. I hope that this won't be mandatory anymore when you test it.

```
 -la5_color-4.9.11 -la5_font-4.9.11 -la5_primitives-4.9.11 -la5_ttf-4.9.11
```

▷ **Question 1:** Get the template from the class web page[1], and save it on your account for example under `mygame.c`. Read it througfully.

▷ **Question 2:** To compile it, use the following commands :

```
gcc `allegro5-config --cflags` -c mygame.c -Wall -Werror -g
gcc mygame.o `allegro5-config --libs` -o mygame -Wall -Werror -g
```

The allegro5-config script is used to get automatically the right compiler options to compile against allegro. When provided the –cflags argument, it gives the preprocessor include path and other compilation options, while the –libs argument is to be used during the linking.

Note that we added some options to the gcc compilation line (namely, -Wall -Werror -g). They ask the compiler to be a bit more picky on enforcing some good practices. From my experience, they are not optional since it is ways too easy to write poor C code, misbehaving in some way. `-Wall` ask gcc to produce more warnings (despite what the name seems to imply, the list is incomplete and some more warnings can be turned on). `-Werror` requests gcc to handle warnings as if they were errors. That way, it's impossible to oversee any warning since the compilation process will fail until you solve them. `-g` requests gcc to add some more information in the produced binary so that the debugging tools produce more informative messages about your source code.

▷ **Question 3:** Run the resulting program. It displays an empty window that get closed when pressing the Esc key. Not quite exciting yet, actually. But we are now ready to begin coding.

▶ Partie 2: The contamination game

We will now implement a simple yet complete game. It consists of a grid board, where each cell is of a given color. The goal is to contaminate every cell of the board. At the beginning of the game, you own the top left cell. At each game turn, you choose a color ; you earn every cell of that color which is adjacent to one of your cells. A stock implementation of the game is provided on the class web page so that you can play a bit with the concept.

▷ **Question 1:** Copy the template introduced previously under ***contamine.c***. It will constitute the main file of your game. Write a Makefile to automate its compilation.

★ Exercice 1: First things first : the board model.

First of all, we will design the data structure which will contain every information about the board. Even if C is not object-oriented, it is a good idea to encapsulate this in a given structure to gain some advantages of the OOP. It will allow us to have more than one board at the same time (useful for example for AI design).

Likewise, we really should apply the Model-View-Controler (MVC) pattern by isolating as much as possible the core game logic (the model) from the display logic (the view). This is for several reasons. First, it allows to easily change the view, for example if you want to test SDL (or even ncurse) instead of allegro one day. Then, it allows to have more than one view of the same data (we'll come back on this later). And finally, MVC is just mandatory nowadays if you want to do a serious user interface.

▷ **Question 1:** Create the following files : ***board.c*** and ***board.h***. Do not forget to protect the content of your board.h against multiple inclusion using pre-processor macros. Update your makefile to produce the relevant .o files when one of the source file gets updated, and then combine them in your binary.

[1] `http://www.loria.fr/~quinson/teach-prog.html`

▷ **Question 2:** Add the following to your header file, and implement the corresponding elements in your source file :

```
typedef struct board *board_t;
board_t board_new(int size_x, int size_y, int color_amount);
void board_free(board_t b);
int board_get(board_t b,int x, int y);
```

Remark : `board_t` is a pointer to the structure. From my experience, this helps enforcing the OOP illusion.

▷ **Question 3: Storing the content.** Each cell of the board has to store a single integer : the color of the cell. You are completely free of how you represent this in your structure in `struct board`. I personally used a vector, and came up with the following macro associating the x,y coordinate on the board to a given position in the vector. The good thing of this macro is that it can be used both to retrieve the content of the cell, and to change it (as exemplified below). The structure content should be private and thus placed directly in the ***board.c*** file.

```
#define cell(board,x,y) ((board)->grid[(board)->size_y*(x) + (y)])
[...]
  color = cell(board,x,y);     /* example of use of the macro were we retrieve the content */
  cell(board,x,y) = new_color; /* the macro can also be used to change the cell content */
```

▷ **Question 4: Initializing the cell's color.** Do so in your constructor using `rand()%color_amount` to get a random number between 0 and `color_amount`. You also need a call to `srand()` in your initialization process (for example `srand(time(NULL))` in the `main()`) to make sure that the pseudo-number generators don't always give the same sequence.

▷ **Question 5: Instantiating your class.** Create a new global variable of type board_t in `contamine.c`; initialize it at the beginning of the `run()` method and destroy it at the end of the same method.

Test your work with a few `printf()`.

★ **Exercice 2: The View.**

Conceptually, the view component is where all displaying logic goes. In Java, you would create a new ContamineView object for this; the C equivalent is to create the two files ***board_ui.h*** and ***board_ui.c***. Naturally, we put the public declaration of our "object" in the former one while the implementation go in the second one.

▷ **Question 1: The view template.** Create ***board_ui.h*** and ***board_ui.c***. Create an empty `struct board_ui` structure in your c file, and add the following declaration in the right file : `typedef struct board_ui *board_ui_t;` Also create a constructor and destructor method (respectively `board_ui_new` and `board_ui_free`). The constructor do not take any argument so far since there is no field in our object. Modify the makefile accordingly.

Create a global variable of type board_ui_t in your contamine.c file and create a new instance right after the board is created. Use the `al_get_clipping_rectangle()` function to get the dimension of the windows. This function is documented on the API website. Likewise, destroy your view before the board gets destroyed.

▷ **Question 2: Linking the board_ui_t object to the represented board_t.**

To display our board, we need to draw one little filled rectangle per cell, each of the relevant color. In order to retrieve the cell content, we naturally need to save a reference to the board in the view object. This first field is of type `board_t`. Add it to the structure representing the `board_ui_t` object, and pass its value as argument to your constructor.

▷ **Question 3: Deciding the colors to use.** We now need a mapping between cell contents and the relevant color. For example, we could decide that if the cell content is 0, we want to draw the box in red. If the content is 1, the box is orange, and so on. Each color being represented as a `ALLEGRO_COLOR` variable, we need a `ALLEGRO_COLOR *` field named `colors`. It points to a vector of colors. But in order to allocate this array in our constructor, we need to retrieve the amount of color from somewhere. This information clearly belongs to the model, and there is little reason of adding an explicit argument to the view constructor.

Remember that the board constructor had that information as argument, and you used it to initialize the content of your board. Add a field to the `board_t` object storing that information into the board object, and write an accessor `board_get_colors_amount(board)` to retrieve it. Once this modification to the model component is done, you can allocate the colors field of your view like the following. You also need to free it in your destructor.

```
res->colors = malloc(sizeof(ALLEGRO_COLOR)*board_get_colors_amount(board));
```

To complete the constructor, we then need to initialize each field of the colors vector. Since Allegro allows you to create colors from their name, you want to add a static list of color names in your program such as the following. Make sure that it is long enough to fit your gaming taste. Then, add a loop in the constructor of `board_ui_t`, calling the function `ALLEGRO_COLOR al_color_name(char *name)` to create each cell with one of the names of the list. There is no need to free the colors in the constructor because they are structure, not pointers to structure. Some more color names can be retrieved here : `http://alleg.sourceforge.net/a5docs/refman/color.html`

```
char *color_name[8] = {"red","orange","yellow","green","cyan","blue","magenta","violet"};
```

▷ **Question 4: Finishing the constructor.** It will become handy later to only use a region of the board to draw the board. This allows to draw some status text below the gaming area. For that, we add 4 floats parameters to the constructor, specifying the x,y position of the top left corner as well as the area width and height. Store these fields in your object.

We should also precompute the cell dimensions once for all, and store them in the view object. We simply proceed by evenly divide the available space by the amount of cells. So, cell_width is for example the region's width divided by the amount of rows in the board. You may need to store this row count in your model, and add a accessor if you didn't already. Compute the cell height in a similar way.

▷ **Question 5: Writing the actual drawing function.** Everything is now in place and writing this function will be quite easy with a bit of arithmetics. You need two embedded loops to traverse each cell. For each cell (cx,cy), call the function **al_draw_filled_rectangle(x1,y1,x2,y2, color)**, where (x1,y1) is the top left corner and (x2,y2) the bottom right one. x1 is given by the following formula : $dx + cx * cell\_width$ while x2 is given by $dx + (cx + 1) * cell\_width$. Y values are computed similarly. The color is given by retrieving the cell content from the board and using the corresponding element of the `colors` field from the view.

▷ **Question 6: Gluing the view in position.** In order to get your game display working, you simply need to add a call to `board_ui_draw(board_ui) ;` to the `draw()` method of ***contamine.c*** (provided that you called your drawing method `board_ui_draw` and your board variable `board_ui` – your mileage may vary).

★ **Exercice 3: The game logic.**

▷ **Question 1: Retrieving the clicked cell.** Mouse clicks can be retrieved by adding the following code to the `run()` method :

```
case ALLEGRO_EVENT_MOUSE_BUTTON_DOWN:
  if (event.mouse.button == 1)
    printf("clic at position %f %f\n", event.mouse.x, event.mouse.y);
  break;
```

▷ **Question 2: Passing clicks to the view.** You should add a function in the view to handle such clicks. Its prototype is `void board_ui_click(board_ui_t bui, int ui_x, int ui_y)`. It does nothing if the click is out of its painting area. In other cases, it compute which cell got clicked (`cell_x=(ui_x-dx)/cell_width`). Use a `printf` to display the computed coordinate, and test it. Make sure that it works even if the painting area does not cover the whole windows.

▷ **Question 3: Passing clicks to the model.** Add a `void board_click(board_t b,int x, int y)` function to the model, and call it from `board_ui_click()`. That's it, you should get the click events in the right format in your model. It's time to implement the actual game logic.

▷ **Question 4: Adding players to the board.** Our game opposes, 2 players trying to conquer the board. We thus need to mark cells belonging to a given player. Several solutions are possible, such as creating a new matrix of boolean per player indicating whether or not it belongs to the given player. We could also create a `cell_t` type containing its color, and the player it belongs to. Then, we need to change the matrix representing the board from a matrix of integer (coding the color) to a matrix of `cell_t`.

The solution I advise you is to code the fact that a cell belongs to player 1 by storing -1 as a color in this cell. Player 2 is denoted by -2 as a color. Since the view wants the `board_get_cell` method to return the color to display, we will add to new integers, representing the current color of each player. This method thus becomes :

```
int board_get_cell(board_t b,int x, int y) {
  int val = cell(b,x,y);
  if (val==-1)
    return b->player_color[0];
  if (val==-2)
    return b->player_color[1];
  return val;
}
```

We should also add another field to the board representing the player which plays next.

▷ **Question 5: Game logic core.** When a player clicks on a given color, he conquers every cells of this color being adjacent to the cells he already own. Computing which cell should be conquered turns out to be a quite challenging task. We should find the border of the already owned zone, and find any cells of the given color neighboring that zone. Moreover, the process is iterative : if a recently conquered cell is neighboring another cell of the right color, it should be conquered too.

For example, in the situation depicted on the right, where the player already owns the (0,0) cell and plays green, the following cells must be conquered :
– (1,0) because it is a neighboring of the previously owned (0,0)
– (0,1) for the same reason
– (0,2) because it is a neighbor of just conquered (0,1)
– (1,2) because it is a neighbor of just conquered (0,2)
– (2,1) because it is a neighbor of just conquered (1,0) and (1,2)

🟥 0,0	🟩 1,0	🟨 2,0	🟨 3,0
🟩 0,1	🟨 1,1	🟩 2,1	🟨 3,1
🟩 0,2	🟩 1,2	🟨 2,2	🟨 3,2
🟨 0,3	🟨 1,3	🟨 2,3	🟨 3,3

The first approach to solve this problem consists in traversing the cell array, and for each of them being of the searched color (green in our case), conquer it if they are neighboring an already owned cell. This process must naturally be called several times, but how many ? For that, we can get inspired by the bubble sort, which sorts a bit the array until there is nothing left to sort. In our context, it would become : convert every cell you can until you failed to convert any. The corresponding pseudo-code is shown on the right.

Pseudo-code of the iterative version

```
int we_converted_something;
do {
  we_converted_something = 0;
  foreach cell c {
    foreach (n in neighbors of c) {
      if (n is already owned) {
        convert(c);
        we_converted_something = 1;
      }
    }
  }
} while (we_converted_something);
```

When implementing this, the main difficulty is to iterate over all neighbors. We could use the brute force, and handle each case manually. This is what is done in the left solution below. This is naturally very laborious, and you should avoid such code duplication by all means. A much better solution (shown on the right) is to have a vector of deltas to apply on the position to get each neighbor, and to apply each of them in a for loop. Since there is no “point” type in C and since we are too lazy to declare one just for that iteration, we store the x and y components of the deltas in separate vectors.

Brute force iteration over the neighbors

```
if (cell(x-1,y-1) is owned) {
  convert(x,y);
  we_converted_something = 1;
} else if (cell(x,y-1) is owned) {
  convert(x,y);
  we_converted_something = 1;
} else if (cell(x+1,y-1) is owned) {
  convert(x,y);
  we_converted_something = 1;
} else if (cell(x+1,y-1) is owned) {
[...] /* More code duplication here */
```

Better iteration over the neighbors

```
int neighbor_x[ ] = {-1,-1,-1,   0,0,   1,1,1};
int neighbor_y[ ] = {-1, 0, 1,  -1,1,  -1,0,1};

for (i=0;i<8;i++) {
  int new_x = x + neighbor_x[i];
  int new_y = y + neighbor_y[i];
  if (cell(new_x,new_y) is owned) {
    convert(x,y);
    we_converted_something = 1;
  }
}
```

You now have every elements to write that solution. But heck, this solution is as inelegant as bubble sort, and promises the same poor performance to solve than bubble sort too. Can’t we do better by thinking a bit more ? Another way yo solve our problem is to start from the player initial position, and grow in every direction by considering its neighboring cells. For each considered cell, if it’s already owned, we grow further from there. If it’s of the searched color, we conquer it and grow further.

So we are basically designing a recursive solution. The main issue is to avoid to consider the same cell twice, to cut the infinite loop. Indeed, from the initial setup depicted above, from (0,0), we will grow to (1,0). How to not grow back to (0,0) when considering the neighbors of (1,0) ? We could try growing only down or right, but this won’t work since we want our algorithm to follow every paths, even the ones drawing a G or H. The solution is to allocate a new boolean matrix `seen` where each cell indicates whether we already traversed this cell or not. The pseudo code becomes :

Pseudo-code of the recursive solution

```
void look_border(board_t b, int x,int y, int color, int player) {
  foreach neighbor (new_x,new_y) of cell (x,y) {
    if ((new_x,new_y) is not out of the board) and (not already seen) {
      mark (new_x,new_y) as seen
      if (new_x,new_y) is owned by the player {
        look_border(b,new_x,new_y,color,player)
      } else if (new_x,new_y) is of searched color {
        mark (new_x,new_y) as owned by the player
        look_border(b,new_x,new_y,color,player)
} } } }
```

You can now write the core logic of your game using either the iterative and the recursive solution, or even both in order to compare their respective performance.
Trick : Even if not utterly complicated, the code you have to write now can fail in subtle ways if you get the indices wrong. You thus need to add some `printf`s to get what's going on. It is also very handy to display differently the cells owned by a given player.

For that, I changed my view component to allocate 2 more colors than expected (I took black and white), and I changed my `board_get_cell` so that when in debug mode, it returns `color_amount` for cells owned by player 1 and `color_amount+1` for player 2.

Displaying zones owned by each player

```
int board_get_cell(board_t b,int x, int y) {
  int debug = 1; // Boolean to set to 0 in release mode
  int val = cell(b,x,y);
  if (val==-1)
    return debug?b->color_amount  :b->player_color[0];
  if (val==-2)
    return debug?b->color_amount+1:b->player_color[1];
  return val;
}
```

▷ **Question 6: Finishing the game logic.** Your game should now be almost playable and it is time to make it completely playable. For example, make sure that each player plays one after the other by changing the `player_turn` field of your model when one player just clicked. Also, it is a good idea to act as if the players would have clicked on their own color before the game starts. Ie, if the player's initial position is red, we should give him every red cells adjacent to that position.

★ Exercice 4: Improving the gaming experience.

The game is now playable, but not very friendly yet. We need to display which player's turn it is, compute their score and determine when the game is over.

For that, I changed my view

▷ **Question 1: Getting ready to display text.** Before we can write any string on the window, we should tell Allegro that we intend to. First, add `al_init_font_addon() ;` in the main, after initializing the keyboard and mouse. Add the following field to your ui global structure : `ALLEGRO_FONT *font ;`

Finally, add the following chunk to the init function. Don't forget to copy the ***font.tga*** file in the current directory. Recompile your code and test that it still works (no change expected yet).

```
ui.font = al_load_ttf_font("font.tga", 0, 0);
if (!ui.font) {
  printf("font.tga not found.\n");
  exit(1);
}
```

▷ **Question 2: Displaying which player's turn it is.** Add an accessor to this information in your model. Then, modify the draw function to display that string. Choosing the used color to display text is rather complicated in Allegro. You should adapt the following chunk to fit your needs. We save the currently used blender (ie, filter on colors put on screen by drawing operations) so that it gets all black, we draw and then we restore the previous blender. `al_draw_textf` is used just like printf, with 4 more arguments specifying where to draw the string and how.

```
ALLEGRO_STATE state;
al_store_state(&state, ALLEGRO_STATE_BLENDER);
al_set_blender(ALLEGRO_ALPHA, ALLEGRO_INVERSE_ALPHA, al_color_name("black"));
al_draw_textf(ui.font, x,y, ALLEGRO_ALIGN_CENTRE,"Player %d: it's your turn",board_get_player(board));
al_restore_state(&state);
```

▷ **Question 3: Displaying each player's score.** We simply take the percentage owned by a player as its score. Change your model to store the amount of cells owned by a player and update it when he plays. Then add an accessor computing the percentage it represents. Finally use it in the draw function to show that value at the right location.

▷ **Question 4: Detecting end of games.** Add a method in the model returning whether free cells remain or not. Use it in draw() to change the centered text when the game is over. Use it in the main loop so that the program quits when we click anywhere after game over.

▷ **Question 5: Welcome screen.** Every game ought to have a nice welcome screen allowing to choose between the options and start the game. The stock implementation we provide is not very nice in that domain, and you should definitely do better in your own game.

▷ **Question 6: Final polishing.** Use `valgrind` to detect any memleaks of your program and fix them. Of course, you cannot fix the errors from the allegro library itself, even if valgrind report some. When you're done, play a bit with your game, you deserve it. But don't forget to fix any bugs you'll find.

★ **Exercice 5: Adding an AI.**

The game is now completely playable, but you don't always have a friend to play with. We will now add the possibility to play against the computer. Naturally we want to have several difficulties level, implemented by separate AIs.

▷ **Question 1: The AI template.** Each AI will be implemented as a function taking one board as argument and returning the color it wants to play next. Declare the following new type in ***board.h*** `typedef int (*ai_play_fun_t)(board_t b) ;`

Every variable of type `ai_play_fun_t` is a pointer to a function taking a board as argument and returning an integer. To call the corresponding function, you simply need to dereference the pointer.

```
ai_play_fun_t myfun;
int played_color = (*myfun)(board);
board_play(board, played_color);
```

1. Add a model field `ai_players` being an array of two such variables, and initialize them to NULL in the constructor.
2. Add a `int board_player_is_AI(board_t b)` function to the model indicating whether the next player to play (stored in the model) is an AI or not. You can retrieve that by testing whether `ai_players[player]` is differing from NULL.
3. Add a `int board_play_AI(board_t b)` function asking the next AI to play which color it wants to play, and apply it. If the next player is not an AI, printf an error message and do nothing. You may need to move the playing logic from `board_click(board,x,y)` where you probably put it into a newly created `board_play(board,color)`.
4. Change your main loop so that if the next player to play is an AI, then it gets a chance to do so.

▷ **Question 2: The first AI.** The simplest (and dumbest) possible AI only pick randomly the next color to play. You can write this AI in one line.

1. Implement `static int AI_random(board_t b) ;` in ***board.c***
2. Add a `board_select_AI(board_t b,int player,int lvl)` in ***board.c***. If lvl==0, then the given player of the board becomes a random AI.
3. Change your main file to set a random AI in second position, and test how hard it is to defeat it.

▷ **Question 3: Toward better AIs.** The random AI is rather boring. I usually win by 90/10 or more. Do do better, the AI has to *evaluate* how interesting a color is. The easiest way to implement that is to allow the AI to try each color and then pick the most advantaging one. For that, we need a function to copy the current board so that the AI can play one color, see how well it worked and retrieve the state before its try. Implement a `board_t board_copy(board_t b)` function.

▷ **Question 4: A counter AI.** Implement an AI which tries each possible color on a newly created copy of the board, and decides to play the color leading to the higher amount of owned cell. Modify `board_select_AI` so that this new AI gets selected when `lvl==1`. This one is a bit better, I only usually beat it by 60/40 or so.

▷ **Question 5: An AI looking further forward.** The previous AI could be improved if it looked not only what's best at next turn, but also what's the best move now to maximize its score in two turns. Create a new AI doing so by embedding two loops : one on the color of this turn, and one on the color on the turn afterward. At first turn, you need to discard colors bringing no immediate gain or your AI may be unable to finish the game by always picking color 0 when only one color is possible.

▷ **Question 6: Organizing AI matches.** We would like to assess how better this new AI is compared to the previous one. For that, create a new `run_match()` function in ***contamine.c***. It is comparable to `run()`, but creates 100 boards and views, tiling the views on the window.

▷ **Question 7: An AI taking its opponent into account.** The results of the match are not as good as expected. our more complicated AI does not clearly outperform the simpler one. It is maybe because it plays as if it were alone on the board. A play that would have been very interesting at time t+2 may become impossible because of what the opponent chooses at time t+1. Implement an AI looking 2 turns forward, letting its opponent play the most advantageous play for him in between.

▷ **Question 8: Looking further forward (optional).** The AI we devised so far applies an algorithm called MiniMax. You can find more on this here : `http://en.wikipedia.org/wiki/Minimax#Minimax_algorithm_with_alternate_moves`. Implement an AI using this algorithm to search 5 moves forward.

▷ **Question 9: Changing the evaluation function.** Even when minimax is setup to look 5 moves forward, it is still quite easy to beat him. This is because it actually use the wrong evaluation function. Since we ask them to increase the amount of owned cells our AIs are focused on their short-term situation. This turns as under optimal on the long run.

It is interesting to try to maximize what could be called the owned zone perimeter : the amount of free cells neighboring one of our owned cell. Implement that AI (I call it *spider*).

The result is a quite impressive opponent at the beginning : it deploys rather fast over the board. But when the two owned zones join, one could say that it becomes shy : it is very reluctant conquering the free zones within its area. It prefers extending its area by 2 cells far from its initial position than taking 20 closer cells. As a result, when you let spider play against counter, it reaches the middle of the screen rather soon, but looses at the end since counter get into the zone of spider and grabs any free cells in there.

▷ **Going further :** The right evaluation function is still to be found. We should balance between the aggressive *spider* and the defensive *counter* by counting the increase of perimeter, the increase of owned cells, and taking both into account in the evaluation function. Other metrics are to be maximized. A crucial one is the amount of secured cells, ie the cells that the opponent cannot reach anymore. Moreover, we should not only try to improve on the metrics, but also to limit the way the opponent improves.

Another way to improve the AI is to use Minimax to explore more turns in advance. But the complexity of this computation explodes with the game tree. It should be possible to cache some information to speed the process up and allow the exploration of further steps.