
Final Work

PROGRAMMATION ORIENTE OBJETS

4 FÉVRIER 2022

GASTON MAZZEI

GASTONMAZZEI95@GMAIL.COM
WEB : GASTONMAZZEI.GITHUB.IO

JULIEN DUPRAT
JULIEN.D.A@LIVE.FR

This is the final project for Patrick Amar’s course “Object Oriented Programming” at Paris-Saclay University during T2 2021. The code and this presentation will remain available at “<https://github.com/GastonMazzei/MazeGame>” unless requested the contrary.

Table des matières

1	Labyrinthe	2
1.1	<code>__width</code> and <code>__height</code>	2
1.2	<code>destroyGardienByIndex(int ix)</code>	2
1.3	<code>is_lab</code>	2
1.4	<code>__data</code> and <code>data</code>	3
1.5	<code>__guards</code>	3
1.6	constructor	4
2	Gardien	5
2.1	Generic dynamic private attributes	5
2.2	Behaviour-defining dynamic private attributes	5
2.3	Macro-defined private attributes	6
2.4	<code>check_availability</code>	6
2.5	<code>update_chasseur_visibility</code>	6
2.6	<code>update_gardien_accuracy</code>	7
2.7	Movement	8
2.8	Public access to lifesigns	8
2.9	<code>update</code>	8
2.10	Fire-related methods	9
3	Chasseur	10
3.1	Ending the game after finding the treasure	10
3.2	Movement-fueled screen messages	10
3.3	Killing Gardiens	10

Throughout this report, the different features of the implementation will be covered while each of the three classes are briefly explained. In order to promote brevity, only the most relevant methods and attributes will be described.

1 Labyrinthe

The class “Labyrinthe” inherits from the class “Environnement” and handles maze’s operations. As such it decyphers a layout file, provides storage, and exposes manipulation operations. The constructor takes as input a pointer to a char array that is the name of the “.txt” file that contains the layout of the maze. The most interesting part is the constructor whose sub-section is at the bottom of this section.

1.1 `__width` and `__height`

The labyrinth is a rectangle (2D), and it’s width and height are private attributes. Their type is integer because they represent a number of elements for (respectively) the width/height in each direction. Users of this class are offered read-only access to them via the following associated public methods.

The `width()` and `height()` methods are reversed for a good reason : The `__data` segment is a char 2d array so it makes more sense to represent it as an array of lines more than an array of columns. Furthermore we don’t benefit more from either representation in the algorithms since we have to parse either way.

In particular, getters return the opposites (i.e. “width” returns “`__height`”). This is to provide users (i.e. public methods’ consumers) with an intuitive representation of data.

```
1 // This are the sections relevant to the width and height
2 // inside the Labyrinthe Class declaration
3 private:
4     int         _width = -1;
5     int         _height = -1;
6 public:
7     int width (int flag = 1) { return (flag == 1)? _width:_height; };
8     int height (int flag = 1) { return (flag == 1)? _height:_width; };
```

1.2 `destroyGardienByIndex(int ix)`

The public method called `destroyGardienByIndex(int ix)` effectively disables the Gardien at the “`__guards`” array at index “ix”.

```
1 // This are the sections relevant to destroying a Gardien
2 // based on it's index inside the Labyrinthe Class declaration
3 public:
4     void destroyGardienByIndex(int i);
```

This is the function that users of the class are required to call when a Gardien’s lifesigns drop below 0, and it’s purpose is to try to make all the Gardien-Chasseur communication not be direct but be mediated by the Labyrinthe class itself.

1.3 `is_lab`

As the name indicates, the public function called `is_lab` returns a boolean indicating if a given char is or not part of the maze (or is at least candidate to be). It’s used by the class constructor, but it was deemed appropriate to also expose it to users “just in case” they find a good use to it.

```

1 // This is a Class Method that explicitly checks if a character
2 // is a word, inside the Labyrinthe Class declaration
3 public:
4     bool is_lab (char c);

```

1.4 __data and data

The private attribute called `__data` is the Labyrinth itself, i.e. a memory block of size $width \times length$ indicating in each position if there is either a wall, a box, free space, the chasseur, a gardien or the bounty itself.

Users of this class have access to reading and writing memory values of this block by requesting them by index via the public method called `data`. Perhaps the most notable example of it's usage is in the moving function of the objects Gardien and Chasseur, where they call it to check if a new position is empty to verify it is a valid movement. This procedure is illustrated in Fig. 1.

```

1 // This are the sections relevant to the block of memory
2 // that stores the labyrinth structure, and the
3 // public interface exposed to users
4 private:
5     char **_data;
6 public:
7     char **data() { return _data; };
8     char data (int i, int j);
9     void set_data (int i, int j, char value);

```



FIGURE 1 – The public Labyrinthe method “data(dx,dy)” allows Gardiens to check if a particular displacement is possible, i.e. if the requested cell is empty or not.

1.5 __guards

`__guards` is a public attribute inherited from the base class from which the Labyrinthe derives from (i.e. “Environnement”). It consists of an array of pointers to objects of type “Mover”, which is the class from which both Gardiens and the Chasseur derive from.

The Labyrinthe Constructor defines this attribute as a block of allocated memory after having counted how many Gardiens are required based on the input text file. Its first element will point to the Chasseur, and the rest of the elements will be the Gardiens by order of apparition increasing in X and then in Y. The public nature of `__guards` is taken advantage of by several methods of the Gardiens and Chasseur, for example to consult the position of each other.

```

1 // This are the relevant sections of the base class declarations
2 // that our main class Labyrinthe inherits from
3 class Environnement {

```

```

4 public:
5     // ...etc...
6     Mover** _guards;
7     // ...etc...
8 };
9
10 // As Labyrinthe publicly inherits from Environnement,
11 // the attribute type Mover** called _guards is still public
12 class Labyrinthe : public Environnement {
13     // ...etc...
14 };

```

1.6 constructor

The constructor reads the layout 'txt' file and either initializes an instance of the Labyrinthe class with appropriate values or crashes/stops/handles if the layout does not meet the format specifications.

```

1 Labyrinthe::Labyrinthe (char* filename) {
2     std::ifstream lab_file;
3     lab_file.open(filename, std::ifstream::in);
4
5     // ...
6
7     find_measurements(&lab_file, line);
8
9     // ...
10
11     // Create walls and objects :-)
12     walls_create ();
13     objects_create ();
14 }

```

The function called “*find_measurements(ifstream *lab_file, char *line)*” finds every size and number of entities to spawn, e.g. the number of boxes, the number of posters. In order to do this, it first isolates the maze from the layout and writes it in the memory addresses reserved for the attribute called “*_data*”. Finally, the memory block can be parsed in order to set the correct value for walls and objects, which is precisely what the last two called functions do : “*walls_create*” and “*objects_create*”.

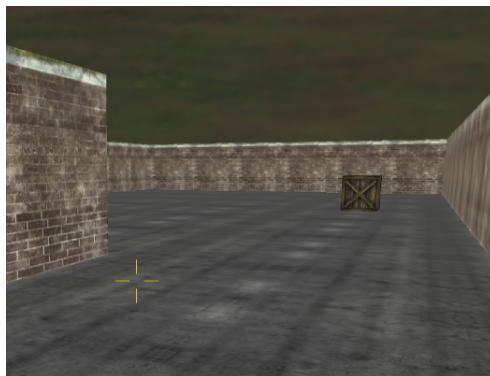


FIGURE 2 – A section of the rendered Labyrinth. The user experience indicates the purpose has been fulfilled : it is not infrequent getting lost in unknown maps.

2 Gardien

The following is the code for the class “Gardien”, which inherits from the class “Mover”. The constructor is unique and requires as input a pointer to a char and a pointer to a Labyrinthe object, the class from the previous section. As the Gardiens have behaviours that depend on randomness, the default constructor has been modified to set the random seed using the current time. Specific methods and attributes will be defined in the following subsections.

2.1 Generic dynamic private attributes

The Gardien class has several private attributes that are “dynamic”, i.e. they are changed over time. For example, the attribute called “_counter” is increased by one every time the public class method called “update” is called.

Other dynamic private attributes are related to the movement : “x” and “y” moving trends are the $\{-1, 1\}$ direction in each axis that the Gardien will take during the random movement, and the distance and which the Chasseur is along with its azimuthal angle in relation to the Gardien itself are stored (and frequently updated) in the attributes “chasseur_distance” and “chasseurTheta” respectively.

```
1 // This are the sections relevant to the NAME
2 // inside the Gardien Class declaration
3 private:
4     int _counter = 0;
5     double chasseur_distance = 0;
6     double chasseurTheta = 0;
7     int x_moving_trend = 1;
8     int y_moving_trend = -1;
```

2.2 Behaviour-defining dynamic private attributes

There is a set of private dynamic attributes in the class Gardien that are not of a numeric type but rather define the current behaviour. They are of type boolean, and they offer ways to parametrize the inner state inside the class methods.

They are as follows :

- “isChasseurVisible” indicates if the Chasseur is visible or not, i.e. if it exists a ray that can travel from Chasseur to the current Gardien without encountering any obstacle.
- “stayStill” indicates if the Gardien’s current behaviour should be to stay still : it is used when the Gardien is visible to implement a mechanism that consists of the Gardien not approaching the Chasseur beyond a minimum radius, macro defined.
- “angry” is a binary parameter that basically determines if the Gardien should be chasing the Chasseur or not.
- “dummy” indicates if the current Gardien has been killed and is lying dead, i.e. is a dummy, or not.

```
1 // This are the sections relevant to the Behaviour-defining
2 // dynamic private attributes
3 // inside the Gardien Class declaration
4 private:
5     bool isChasseurVisible = false;
6     bool stayStill = false;
7     bool dummy = true;
8     bool angry = false;
```

2.3 Macro-defined private attributes

A few private attributes are constants defined via macros. They are the Gardien's lifesigns, the Gardien's accuracy, it's probability of changing direction when moving randomly, the eyesight and minimum Chasseur approach radius in units of Labyrinth blocks, as well as the frequency with which it updates its step and the frequency with which it fires. This altogether defines the behaviour, that can be set up during compilation, and thus is part of the machine code of the program.

```
1 // -----Labyrinthe.h-----
2 // Macros are defined here
3 #define GARDIEN_FIRING_FREQUENCY 50
4 #define GARDIEN_WALKING_FREQUENCY 2
5 #define EYESIGHT_REACHES_HERE 50;
6 #define DONT_GET_CLOSER_THAN 3;
7 #define GARDIEN_PROBABILITY_CHANGE_DIRECTION 0.03
8 #define GARDIEN_LIFE 50
9 #define GARDIEN_ACCURACY 1.0
10
11
12 // -----Gardien.h-----
13 #include "Labyrinthe.h"
14 // This are the sections relevant to the
15 // Macro-defined private attributes
16 // inside the Gardien Class declaration
17 private:
18     int firing_frequency = GARDIEN_FIRING_FREQUENCY;
19     int walking_frequency = GARDIEN_WALKING_FREQUENCY;
20     int eyesight_reaches_here = EYESIGHT_REACHES_HERE;
21     int dont_get_closer_than = DONT_GET_CLOSER_THAN;
22     double lifesigns = GARDIEN_LIFE;
23     double accuracy = GARDIEN_ACCURACY;
24     double probability_change_direction = (double) 1 -
25         (double) GARDIEN_PROBABILITY_CHANGE_DIRECTION;
```

2.4 *check_availability*

Gardien's private method called "check_availability" takes a point (x, y) and returns a value indicating if it can be reached by a ray of light from Gardien's current position or not. Instead of returning a boolean it returns an integer, so that it can be accumulated by the caller as the binary decision about if the Chasseur is visible or not is not made using one ray of light but a collection of them using a threshold over the accumulated value.

```
1 // This are the sections relevant to the check_availability
2 // inside the Gardien Class declaration
3 private:
4     int check_availability(int ix_x, int ix_y);
```

2.5 *update_chasseur_visibility*

Gardien's private method called "update_chasseur_visibility" serves the purpose of updating private attributes that define the Gardien's state : it computes the ray of light that joins Gardien and Chasseur and, if it's path is not impeded by anything, it sets up the Gardien's state as angry with the Chasseur

visible. In the process, the attributes related to the distance from Gardien to Chasseur and the azimuthal angle between them are also updated.

```

1 // This are the sections relevant to the update_chasseur_visibility
2 // inside the Gardien Class declaration
3 private:
4     void update_chasseur_visibility(void);

```

2.6 update_gardien_accuracy

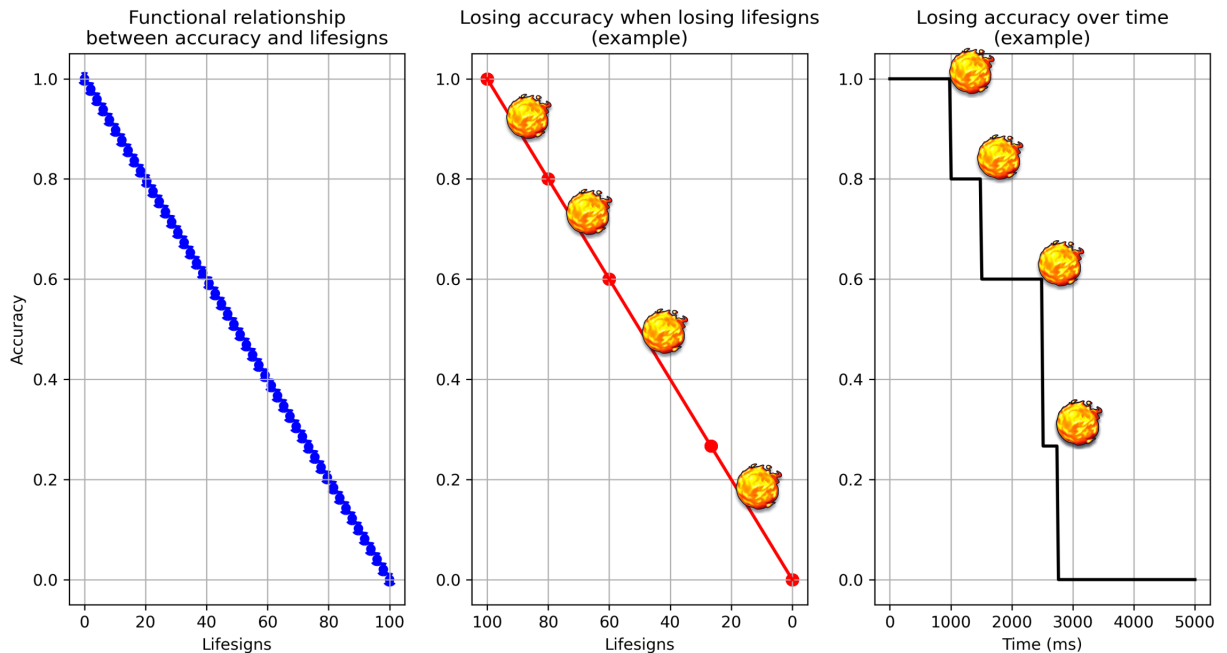
The private Gardien method called “update_gardien_accuracy” updates the Gardien’s fireball firing accuracy based on the lifesigns levels at that time. The function implemented is the one in Eq. 1. The function along with an example of a session can be visualized at Fig. 3. A similar dependency occurs for the frequency at which the Gardien fires, the details are visible at the implementation of “update_gardien_firingFrequency”.

$$accuracy = \left(\frac{current_lifesigns}{GARDIEN_INITIAL_LIFE} \right) \times GARDIEN_INITIAL_ACCURACY \quad (1)$$

```

1 // This are the sections relevant to the update_gardien_accuracy
2 // inside the Gardien Class declaration
3 private:
4     void update_gardien_accuracy(void);

```




Where  represent having been hit by a fireball

FIGURE 3 – (left) Functional relationship between the accuracy and the lifesigns. Gardiens are continuously updating their accuracy value as a function of their current lifesigns. (center, right) Example of a session. As time progresses (horizontal axis, right) the lifesigns decrease (horizontal axis, center) due to received fireball damage (icon : fireball). The automatic update leads to varying levels of accuracy (vertical axis).

2.7 Movement

The private methods called “*move_towards_chasseur*” and “*move_randomly*” define the two types of movement.

When the Chasseur is visible¹, the Gardien’s inner state will be “angry”. If the Gardien is angry, it tries to move towards the Chasseur. If not, it moves randomly.

When it moves randomly it uses the macro-defined constant attribute type double called “*probability_change_direction*” to draw a uniformly-sampled random number and effectively change or not its random direction by using the constant attribute as a threshold. If not, it continues moving in a straight line. In any case, if it reaches a wall it will reverse it’s direction by 180 degrees.

On the contrary, if the Gardien is angry it will try to move towards the Chasseur. There is a macro-defined constant attribute of type int that is called “*dont_get_closer_than*”, which effectively sets the maximum proximity it can be between the Gardien and the Chasseur (in unit cells). Gardiens won’t approach further, but in any case they will update their orientation to point towards Chasseur.

```
1 // This are the sections relevant to the
2 // move_towards_chasseur and move_randomly methods
3 // inside the Gardien Class declaration
4 private:
5     void move_towards_chasseur();
6     void move_randomly();
```

2.8 Public access to lifesigns

Gardiens offer a public endpoint to interact with their lifesigns. In particular they can be (1) brought to life via “*give_life*”, (2) killed via “*remove_life*”, (3) their lifesign values can be accessed via “*get_lifesigns*”, and (4) their status (dead or alive) can be retrieved via “*get_life*”.

```
1 // This are the sections relevant to the public access
2 // to lifesigns inside the Gardien Class declaration
3 public:
4     void give_life(){dummy=false;}
5     void remove_life(){dummy=true;}
6     bool get_life(){return dummy;}
7     double get_lifesigns();
```

2.9 update

The public update method can be interpreted as the central processing unit of the Gardien object : it loops through all the required processes : if Gardien is alive then it does nothing, if not it recomputes accuracy and fireball firing frequency according to the current level of lifesigns, if Chasseur is visible it attempts to walk towards it while firing a fireball, or else it moves randomly.

```
1 // This are the sections relevant to the update
2 // inside the Gardien Class declaration
3 public:
4     void update (void);
```

1. i.e. a ray of light can travel between the Gardien and the Chasseur, albeit only if the distance is not too big. There is a threshold that models the Gardien’s eyesight, so that he cannot distinguish the Chasseur from other Gardiens if the distance is largen than this value.

2.10 Fire-related methods

There are three fire-related methods. “*fire*” provides access to firing a fireball. It attempts to do so at an ideal direction, i.e. the Chasseur’s direction, and it can be randomly perturbed to fire in another direction (i.e. to miss) in dependence with the accuracy. It is called by the central processing method, i.e. “Update”, if the Gardien is angry and the Chasseur is visible. “*process_fireball*” controls the explosion of the fireball, and whenever it explodes it computes the distance to the Chasseur to discount the lifesigns if required. Finally “*process_fireball_external*” provides access to the update of the lifesigns according to an external fireball. It is used by Chasseur whenever a fireball is thrown : it iterates through all Gardiens calling their external fireball processor methods.

```
1 // This are the sections relevant to the fire-related methods,
2 // there are three of them.
3 // inside the Gardien Class declaration
4 public:
5     void fire (int angle_vertical);
6     bool process_fireball (float dx, float dy);
7     bool process_fireball_external (float dx, float dy);
```

3 Chasseur

The following is the code for the class “Chasseur”, which inherits from the class “Mover”. The only constructor available is one that takes as input a pointer to a char and a pointer to a Labyrinthe object, the class from the first section. Some behaviours and class properties will be mentioned in the following subsections ; for the complete code please refer to “Chasseur.cc”.

3.1 Ending the game after finding the treasure

The implementation of the ending of the game after finding the treasure is inside the displacement feasibility calculation. The boolean-returning function called “*move*” uses a private auxiliary function called “*move_aux*”, so the instruction to terminate was added for the case the proposed displacement is towards the treasure. The success message is visible in (E) at Fig. 4.

3.2 Movement-fueled screen messages

During the game, several screen messages are visible. Instead of implementing the screen printing on the “Update” method, it was deemed more amusing to implement a **movement-fueled counter**.

The messages appear and disappear on the screen based on the value of an integer counter in relation to constant class attributes that control the frequency. As the printing module is attached to the “*move__aux*” function, the counter only increases when the user moves the Chasseur.

This allows the screen messages to be updated only when the Chasseur moves, thus giving the sensation that the screen information is movement-powered.

4.

3.3 Killing Gardiens

The most probable result of an interaction between Gardiens and the Chasseur is the death of the former. This is so because in this implementation the Gardiens have lifesigns ≈ 10 times smaller those of the Chasseur. As part of the method “*process_fireball*”, the Chasseur updates the Gardiens lifesigns. If one’s lifesigns are found to have reached zero, the Labyrinthe class Gardien killer is called (i.e. “*destroyGardienByIndex*”. For the users, the result of this is that (1) the Gardien will stop throwing fireballs at them and they will (2) lie on the ground. Examples of this interaction are available in Fig. 4.

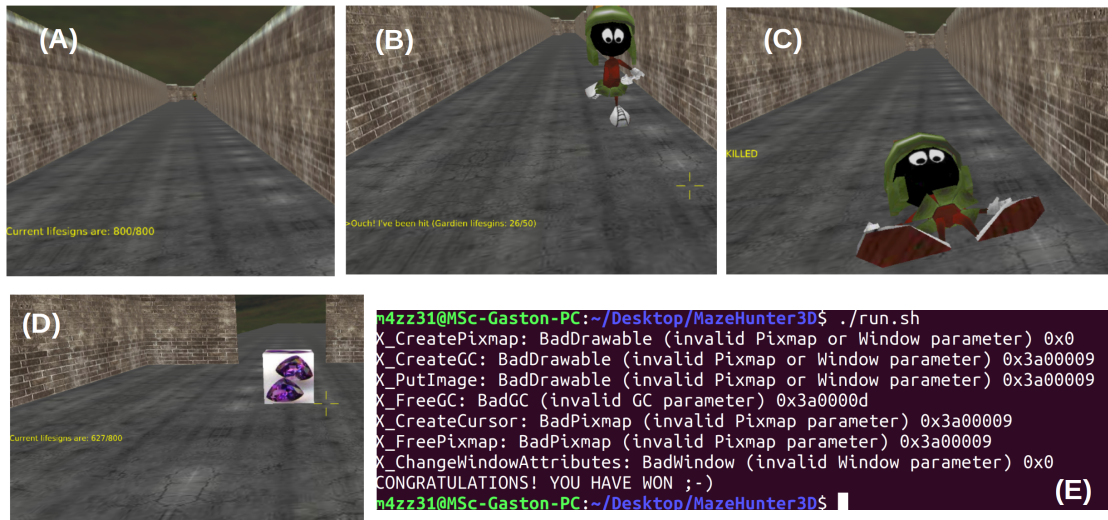


FIGURE 4 – Storyboard showing relevant scenes from a possible game. (A) The Chasseur is spawn, and lifesigns start being visible after the movement is started. (B) Melvin the Marsian appears. The Chasseur throws him a fireball and his lifesigns are displayed on screen. (C) After killing him, a message appears indicating so. (D) The display has defaulted to the Chasseur’s lifesigns, and the treasure has been found. (E) After finding the treasure, session terminates with a success message.