CUPIF Loïc
PICHARD Tristan

Int-2
2022

PIXEL TRACER

Algorithmics and Data Structures

I) Introduction

The aim of this project is to understand how the representation of a digital image works. A digital image can be either a raster image or a vector image. A raster image is a set of pixels in a matrix. This makes it possible to modify the image pixel by pixel and also to create colour shades and texture effects. However, this type of digital image is very large and when you zoom in it is very pixelated. To overcome this problem the second type, vector images, represent the images from a mathematical point of view. This means that no quality is lost when zooming in, but with this type of image you can only make geometric shapes, which has an impact on the final rendering of the image.

For our project, we are going to base ourselves on vector images. The goal is to enter several geometric shapes to create an image. To do this we will use the C language. In order to carry out this project, we will have to create structures in order to organise the code in an optimal way. Through this project we will reinforce our learning of structures but also of arrays. We will also work with pointers, which is a very common notion for structures and arrays.

II) Presentation of our project

Our image creation project offers the user several geometric shapes to create his image. The available shapes are :

- Point
- Square
- Circle
- Rectangle
- Polygon with n side

The user will be able to add any of the above figures to his image at any size. At the beginning, the user will have to define the area he wants to work on and then he can start drawing. A prompt (>>>) allows the user to enter commands.

Here are all the possible commands:

- square x y length
- rectangle x y width height
- point x y
- line x1 y1 x2 y2
- circle x y radius
- square x y length
- polygon number of vertexes
- exit
- list
- delete id
- help
- erase
- plot
- clear

III) Technical presentation of the project

Notre projet comporte 6 fichiers .c, 3 fichiers .h et un makefile, nommés ainsi :

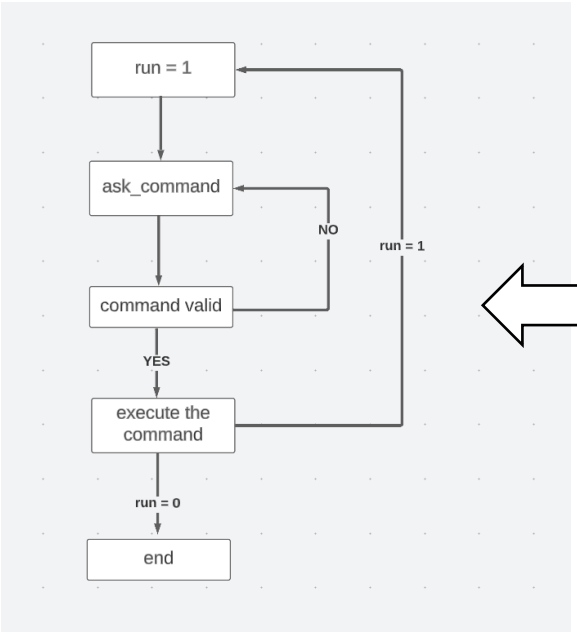
- project.c
- command.c
- shape.c
- manage_shape.c
- id.c
- area.c
- area.h
- project.h
- id.h
- Makefile

Faire différend fichier permet de pouvoir mieux organiser le code et savoir où trouver les fonctions qu'on cherche. Cela permet aussi de mieux comprendre le code et d'éviter de se perdre dans un seul et même fichier. Nous allons vous présenter chaque fichier du projet et vous détailler les fonctions importantes qui les compose.

Project.c

This file is the main file. It contains the int main() function which is the start function. In this function, the user is asked for the size of the grid and the area is created. Then we initialise the area with the function clear_area(area) and display it with print_area(area). Then, we launch the menu(area) function which takes care of the management of user commands. This function is a while loop that allows the program to run continuously. If this loop is stopped, the program is also stopped.

Here is a flowchart that represents the function int menu(Area* area)




```c
int menu(Area *area)
{
    int run = 1;
    while (run)
    {
        fflush( Stream: stdin);
        Command *cmd = create_commande();
        read_from_stdin(cmd);
        if (verif_command(cmd) == 1){
            run = read_exec_command(cmd, area);
        }
        else{
            printf( Format: "Invalid command\n");
        }
        free_cmd(cmd);
    }

    for (ui i = 0; i < area->height; i++)
    {
        free( Block: area->mat[i]);
    }
    free( Block: area->mat);
    return 0;
}
```

Command.c

This file handles commands entered by the user. To do this, the file consists of 6 different functions. The first function Command* create_command() is a function that creates the command structure. It will allocate the necessary memory, initialise the counter to zero and allocate memory for the parameter list. It will return a pointer to this created structure.

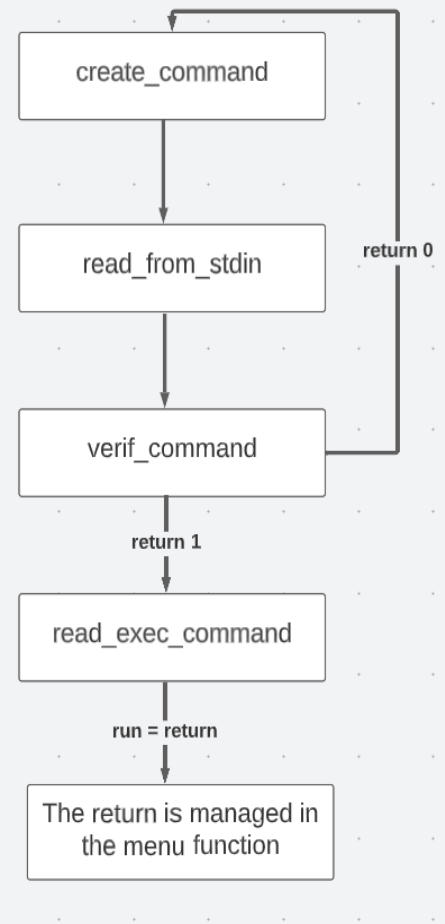
The second function void add_int_params(Command* cmd, int p) adds the given parameter to the current command list.

The third function void free_cmd(Command* cmd) frees the space allocated for the command. It is used when the user's command has been processed.

The fourth function void read_from_stdin(Command* cmd) asks the user to enter a command. It will also process it and use the second function to sort the parameters entered in the cmd structure.

The fifth function int verif_command(Command* cmd) checks whether the command entered by the user is correct. It returns 1 if it is correct, 0 otherwise. It checks whether there are too many or too few parameters. For example, if the user wants to add a point and he enters this command: >>>point 1, the function will return 0 because a parameter is missing. This function is made with if, else if to process commands on a case-by-case basis.

The sixth function int read_exec_command(Command* cmd, Area* area) executes the command entered by the user. Like the previous function, it deals with each command individually. It's return 0 if the command entered is exit because this allows to stop the while loop in the function int menu(Area* area)

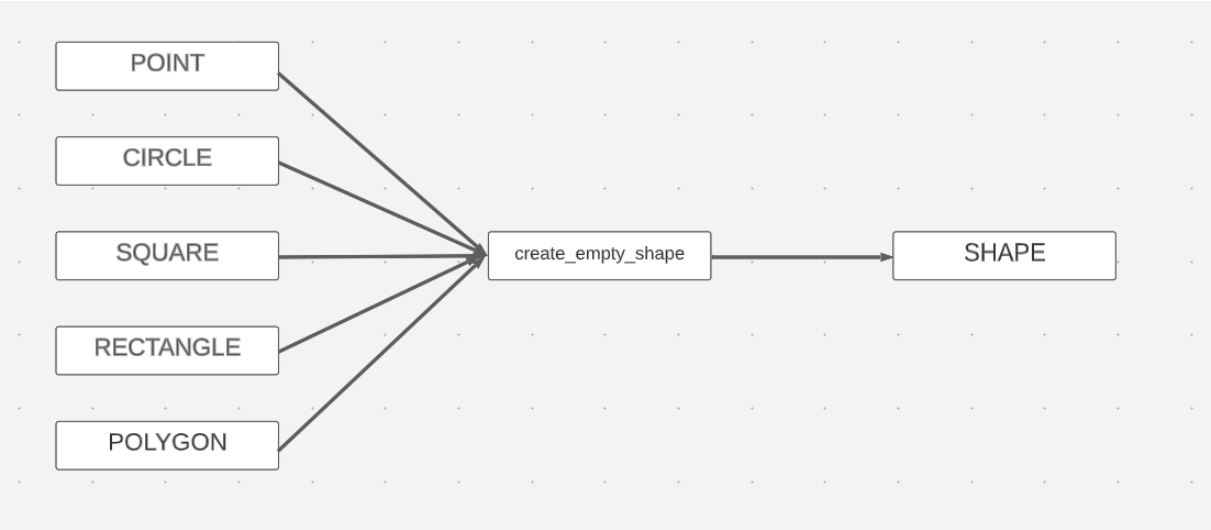


Shape.c

This file contains all the functions to create a figure. It is thanks to these functions that we can change any type of shape into a single type which is shape. This means that a point becomes a shape, for example, and thus we can create functions that take the shape type into account and can therefore be universal. This avoids creating repetitive functions for every possible shape type.

The Shape function Shape *create_empty_shape(SHAPE_TYPE shape_type) is the most important one in this file. Indeed, this function creates the pointer that we store in the list of shapes of the area. In this function, we also create the shape identity, and we say what type is this shape (point, square, ...). We also notice that in all the functions after we use this function. This function is therefore important because it is a transfer function that allows to link any figure to the area.

The other functions allow to make the links with the create_empty_shape function on a case-by-case basis.



Manage_shape.c

The file is used to create and display different types of shape. We have two types of functions, void functions, and functions with a type. The functions with a type are used to create a shape (point, circle, square, …) and the void functions are used to display information about the shape type.

Here, on this image, we create a line with the function Line* create_line(Point *p_1, p_2). This function returns a pointer of type Line to the space we are allocating. Then the function void print_line(Line * line), which takes a pointer of type Line in parameters, displays the information relative to this pointer.

```
Line* create_line(Point *p_1, Point *p_2)
{
    Line* line = (Line*) malloc( Size: sizeof(Line));
    line->p1 = p_1;
    line->p2 = p_2;
    return line;
}

void print_line(Line *line)
{
    printf( Format: "LINE ->\n\tPOINT 1 -> x : %d | y : %d\n\tPOINT 2 -> x : %d | y
}
```

Id.c

This file is used to generate an identifier for each shape created. There is only one function and one variable. The variable is initialized to 0 and each time the function is called, we add 1 to the variable. This makes it possible to create an infinite number of different identifiers for the shapes.

```
Shape *create_empty_shape(SHAPE_TYPE shape_type)
{
    Shape *shp = (Shape *) malloc( Size: sizeof(Shape));
    shp->ptrShape = NULL;
    shp->id = get_next_id();
    shp->shape_type = shape_type;
    return shp;
}
```

We call the function in this one which is in **shape.c**. This makes that with each shape created we call the function so that each shape has a different identifier.

Area.c

In this file, we will find all the functions that have an impact on the area. This file contains 12 different functions.

```
Area* create_area(unsigned int width, unsigned int height)
{
    Area* area = (Area*) malloc( Size: sizeof(Area));
    area->width = width;
    area->height = height;
    area->shapes = (Shape**) malloc( Size: SHAPE_MAX * sizeof(Shape*));
    area->nb_shape = 0;
    area->mat = (BOOL**) malloc( Size: width * sizeof(BOOL*));
    for (ui i = 0; i < width; i++)
    {
        area->mat[i] = (BOOL*) malloc( Size: height * sizeof(BOOL));
    }
    return area;
}
```

The main function is Area* create_area(unsigned int width, unsigned int height). In this function, we create an area from an Area structure. We give it a height and width which we pass as parameters. This function is very important because it creates all the variables which will store the information of the area.

We can find 5 functions that are directly linked to this structure created in the previous function. All these functions have the word area in their name, which makes them easy to find. Among these functions, we can find, for example, void print_area(Area* area) which displays on the screen the area matrix with ".

There are also 6 functions with the word write in their name. These functions allow you to display the figures in the matrix numerically. Its functions take two arguments, the first is the area in order to have access to the matrix and take the structure relative to the shape in order to have the necessary information to be able to create the shape. Its functions will change the necessary 0's to 1's in the matrix so that when displayed the user can see the figure.

Area.h

This file is used to implement the **area.c** and **command.c** functions in the project. We can therefore divide this file into two parts. In the first part we will find all the functions and structures of **area.c** and in the second the functions and structures of **command.c**.

We define two variables. The first one SHAPE_MAX which is used to contain the maximum number of possible shapes, here 100, and the type of BOOL which is equivalent to an integer.

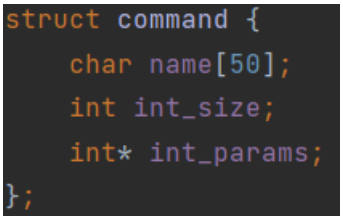
```
// definition of the struc
struct area {
    unsigned int width;
    unsigned int height;
    BOOL** mat;
    Shape** shapes;
    int nb_shape;
};
```

We find the area structure which contains 5 parameters. The width and height of the area, the integer matrix which acts as a numerical grid, a list which will contain all the shape pointers and a variable which will store the number of shapes in the list.

And after we generalize all the function contain in **area.c**.

In the second part of the file, we do the same but with the **command.c** file.

The command structure contains 3 parameters. The parameter name contains the name of the parameter (exit, point, clear), int_params is a list that will contain all the parameters of the command, like for example the coordinates of a point and int_size contains the number of parameters in the list.



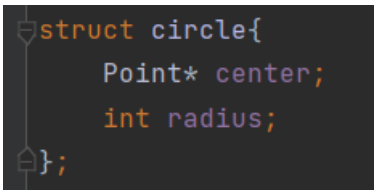
And after we implement all the function in **command.c** in the project.

Project.h

This file is very important because it implements many things in the project. Firstly, it includes all the libraries we need to run the project. Secondly, it defines a SHAPE_TYPE which can be of different types. This allows the shape structure that we will see next to contain different shapes (point, circle, etc...).

Next, we create all the structures that we are going to use. We create Point, Line, Circle, Square, Rectangle, Polygon, Shape. All these structures become types in the project. We notice that all structures, except Shape, are in the type enum of SHAPE_TYPE.

On the left is an example of a structure which contains all the parameters of a shape. Here it is the Circle structure which contains a point variable (a point structure, which contains two coordinates) which represent the centre of the circle and the radius variable which contains the radius of the circle.



Next we can see the two functions in **project.c**. This allows them to be listed in the project even though they are not used in any other function.

Then we can see that we implement all the functions of **manages_shape.c** and **shape.c** in order to use them in the project. The explanations of these functions can be found in their respective sections.

Id.h

File which allows the generalisation of the identifier generating function. This makes it possible to have the function in the whole project.

Makefile

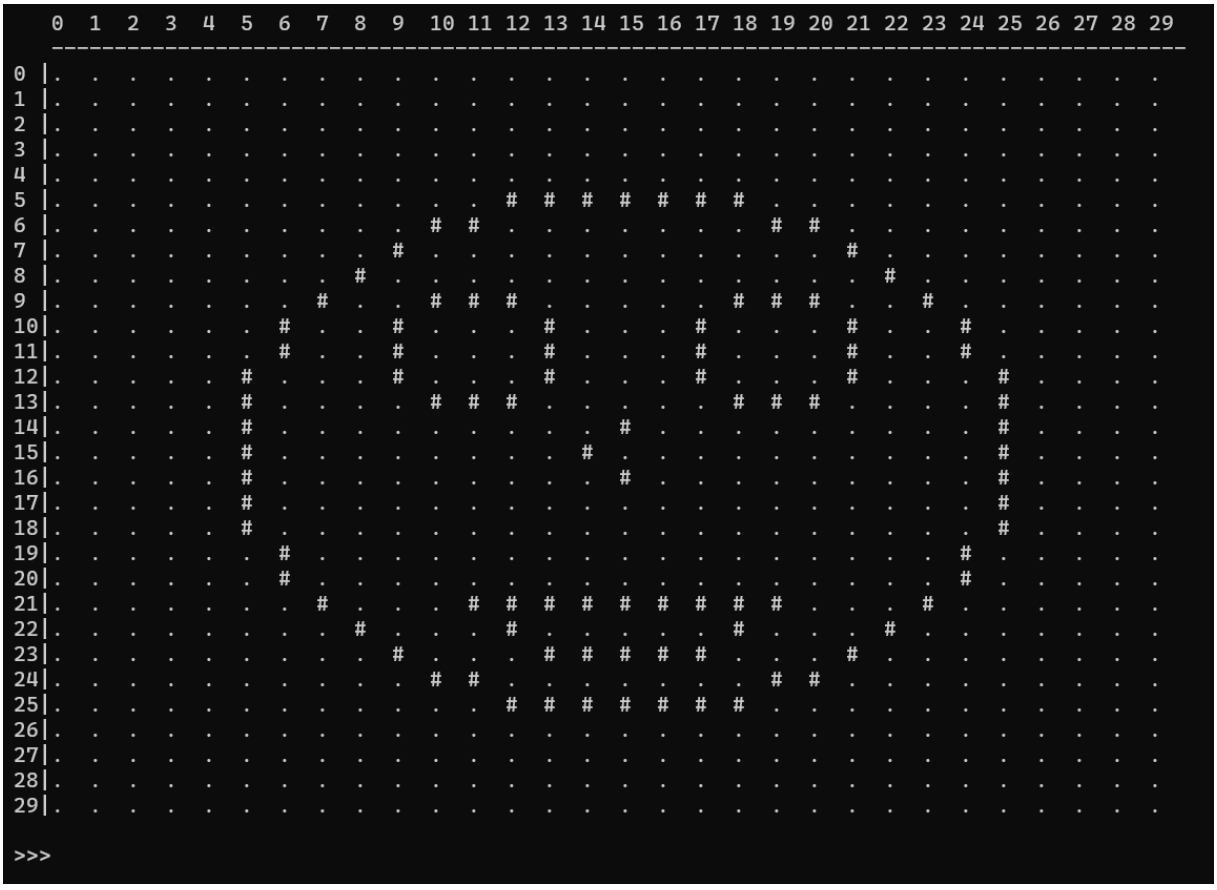
This file is used to compile the project, the first line CC considers the compiler, here gcc. The SRC line considers all the files to compile. In our case it takes all the .c files.

IV) Presentation of the results

The command help :

```
Here are the different commands available:
       - clear: clear the screen
       - exit: exit the program
       - point x y: add a point with coordinate x and y
       - line x1 y1 x2 y2: add a segment connecting two points (x1, y1) and (x2, y2)
       - circle x y radius: add a circle of centre (x, y) and a radius radius
       - square x y length: add a square whose upper left conner is (x, y) and whose side is length
       - rectangle x y width height: add a rectangle whose upper left corner is (x, y), whose width is width and whose height is height
       - polygon number of vertexes: add a polygon with a number of vertexes
       - plot: refresh the screen to display all the geometric shapes in the image
       - list: display a list of all geometric shapes with all their information
       - delete id: delete a shape from its identifier id
       - erase: remove all shapes from an image
       - help: display the list of all the commands with how their works
>>>
```

If the user has trouble with the program, they can enter the help command. This will display all the information needed to make the program work, as you can see in the picture above.



Here is an example of what you can do a cute smiley who is happy in life. To do this enter the following commands:

- Take an area of 30/30
- circle 15 15 10
- line 23 17 23 13
- line 21 11 21 19
- point 22 12
- point 22 18
- circle 11 11 2

- circle 11 19 2
- line 14 15 15 14
- point 16 15
- plot