钱立坤 15310116001
charles-qian@outlook.com

孙皓 15310116005
sh9339@163.com

C 语言程序设计报告

2017.06.18

目录

一、程序概况

本次程序的内容是借助 Libsx 库，创建一个可操作的图形化人机交互界面和一个机器人，并使得机器人在一个方形区域进行移动，完成用户输入的一系列指令。

指令包括使机器人前进、机器人后退、机器人左转、机器人右转、机器人定位、机器人抬笔、机器人落笔、上色、显示机器人、隐藏机器人、**加宽线条（自创）**。

本程序一共包含 13 个文件，3 个.c 源代码文件（window.c callback.c robot.c）、2 个.h 头文件（callback.h robot.h）和一个可供编译使用的 makefile 文件。因此，具有 3 个对应的.o 对象文件（window.o callback.o robot.o）和 1 个可执行文件（DrawRobot）。此外，我们针对本程序的“文件读取”功能，自行设计了 3 个测试文件（TOUR、PAVILLON、BATEAU）供您检验。

如图 1：

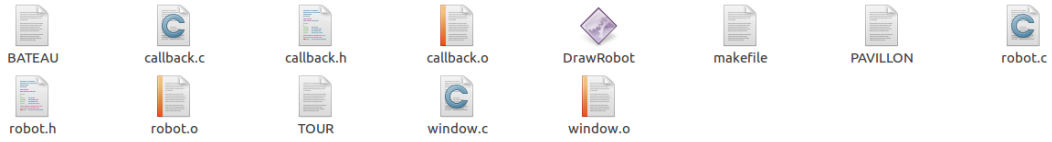


图 1

用户启动程序后，可以选择是直接读取文件实现机器人的绘图工作（FILE 环境下）或者是选择输入具体指令实现机器人绘图工作（programme 环境下），默认环境是用户需要自行输入指令来对机器人进行操作，即 programme 环境下。

（1）在 programme 环境下，用户可以在指令框中输入所需指令，并单击 enter 按钮执行或单击 quit 关闭程序窗口。

如图 2：

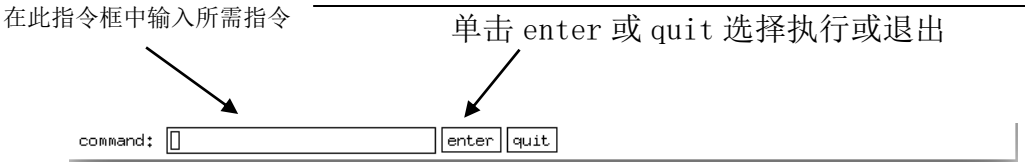


图 2

（2）在 FILE 环境下，用户单击 FILE 按钮，会看到弹出的窗口（图 3），用户按提示在相应位置输入文本，文本即文件的名字，然后点击 OK 按钮，该操作可以读取文本文件中的指令并且执行。若点击 Annuler 按钮，则退出文件读取功能，恢复到 programme 环境下。

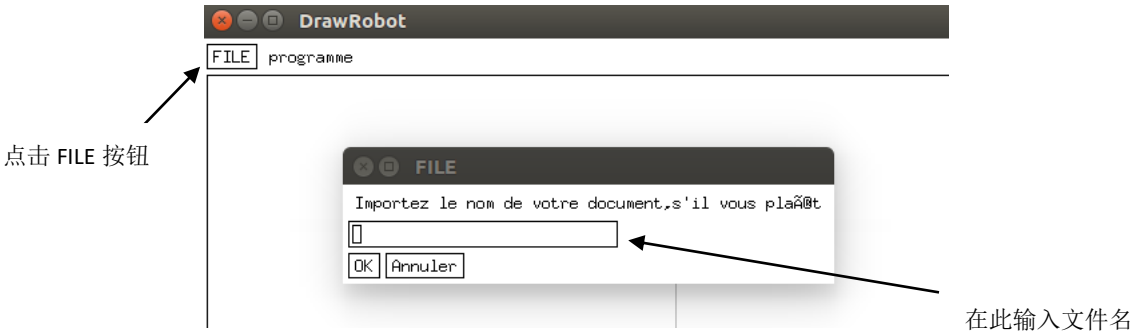


图 3

本程序自带三个测试性的文本文件，名字为：TOUR、PAVILLON、BATEAU，您可以选择用这三个文件测试我们的程序，当然您也可以参考我们测试文件的格式自行编写文本文件来实现本程序的文件读取功能。

二、程序设计

本程序运用 libsx 库中的函数构建图形化界面并进行相关操作，通过宏定义结构体来确定画轨迹所在的点以及颜色，运用指针来读取判断并执行输入的内容，并利用指针的间接引用来改变结构体的值，通过迭代以及循环将点记录下来以便于机器人的隐藏和显示。

程序由三个源代码文件组成，分别用于构成图形化界面（window.c），分析命令语句并画图（callback.c）以及结构体的初始化（robot.c）。

在整个程序中，宏定义的结构体是核心，也是本程序的设计点。结构体共有两部分功能，我们可以将它抽象成“画笔”和“历史”两部分。

首先，在主函数中定义并且初始化之后，结构体就起到了一个“画笔”的作用。通过 OK 按钮对结构体指针进行回调，这个“画笔”就被调入到了 callback 函数中去，任何一个指令旨在改变结构体的内容，从而达到指定的要求。比如说，AVANCE 指令，就是把结构体中的 x 和 y 这两个元素，在指定的角度上所加上一个数字进行更改，从而实现按照既定方向前进一定的距离的操作。

其次，为了在每一步都绘制出来画笔的位置（即机器人），结构体需要包含一个记录所画的点的“历史”的功能。每执行一步，不管是画轨迹或者不画轨迹，结构体都会将这步操作的始末位置、线的颜色（不画线颜色为‘X’）以及画笔的粗细记录下来。从而在每执行一步操作时，需要将之前的屏幕清空后，再次，按照原来的颜色、粗细及位置画上之前的轨迹和新位置的机器人。

另外，通过最后的整体检查和进一步分析发现，结构体完全可以由一个链表所构成，这样一来既可以大大减少结构体中变量的个数，又可以节省许多空间，还可以使操作不受次数限制。但是由于发现时间较晚，需要改动的话必须重新改写大部分程序，因此没有足够的时间更改，这是本次程序有待改善的地方。

三、程序文件介绍

3.1 robot.h

此文件包含了有关机器人的**结构体的定义**和**初始化函数的声明**,应用在所有的.c 源代码文件中。

3.1.1 头文件

robot.h 文件包含的头文件，除去常用的 stdlib.h 库和 stdio.h 库之外，又包含了可视化窗口所必需的 libsx.h 文件。

3.1.2 宏定义

此文件中有三个宏定义,前两个是绘画窗口的高度(HEIGHT)和宽度(WIDTH)。第三个宏定义是次数（FOIS），它代表了指令输入的最大次数。由于没有引用链表，此程序所执行的次数限制在一个长度为 FOIS 的 char 型数组中。

3.1.3 location 结构体

此结构体是本函数最主要的结构体，相当于整个绘画程序的画笔，里面包含了多个元素。

int x、int y 元素是当前画笔的位置坐标。

double angle 元素是当前画笔的角度。

char color 元素是当前画笔的颜色。

int leve_ou_baisse 是判断当前画笔是否抬起来的元素。在函数中，通过看该元素的值是否为真来判断是否落笔。

int cache_ou_montre 是判断当前机器人是否显示的元素。在函数中，通过看该元素的值是否为真来判断机器人是否显示。

int i 元素是记录机器人移动次数的变量，控制数组元素的移动次数。

int history[2][FOIS]元素记录机器人移动的点位置的横纵坐标

int LineWidth[FOIS]元素记录机器人移动的线的宽度。

char Color[FOIS]元素记录下机器人移动所画轨迹的颜色（如果画笔没有画线只是移动，令颜色为字符‘X’）

3.1.4 initialiser 函数

此处用于初始化函数的声明。

3.2 calback.h

此头文件包括**数据**和**小部件的宏定义**及相关**返回函数的声明**。

3.2.1 头文件

调用了之前的 robot.h 头文件。由于需要在计算角度时调用三角函数公式，因此调用了 math.h 库。

3.2.2 宏定义

LONGUEUR 长度定义为可输入文本框的长度，因此也是指令输入的最大长度。

PI 是圆周率的宏定义。

3.2.3 函数声明

分别声明了四个回调函数，用于从窗口到回调文件的连接。

3.2.4 变量声明

小部件 COMMAND 是一个全局变量，它可以在窗口函数中初始化然后直接运用于回调函数，从而获得输入的指令信息。

3.3 window.c

此文件用于主要**图形化界面的制作**。

3.3.1 头文件

调用了两个头文件 callback.h 和 robot.h

3.3.2 main 函数

main 函数定义了结构体并且调用 initialiser 函数进行结构体的初始化。调用 init_display 函数和 MainLoop 函数启动主循环。

3.3.3 init_display 函数

此函数主要负责构建图形化界面。使用的主要形参为在主函数中定义的指向结构体的指针 d，在绘图区 AREA 创建时以及在点击按钮 OK、FILE 时，将此指针传递给相应的回调函数中。

3.4 robot.c

此文件用于画笔机器人数组的初始化。

3.4.1 头文件

头文件调用 robot.h 文件

3.4.2 initialiser 初始化函数

此函数用于初始化结构体，可以将里面的值初始化为最初值：将坐标设置为用户界面的（0,0）点，初始方向为 x 轴正方向，初始画笔颜色为黑、粗细设置为 1（原始粗细），抬笔且显示机器人，并且初始化结构体中的三个数组以实现清空历史记录的目的。

3.5 callback.c

此文件是整个程序的核心，程序的绝大部分操作都写在此文件中。

3.5.1 头文件

调用 callback.h 和 robot.h 两个头文件

3.5.2 file 函数

文件函数。创建读取文件操作时的图形化界面，并且调用 file_ok 函数。

3.5.3 file_ok 函数

运行文件读取功能的函数。文件环境下窗口的 OK 按钮对应的回调函数，主要负责检测所输入的文件名称是否正确以及按行读取文件中的指令并进行相应操作。

3.5.4 area 函数

区域函数。AREA 绘图区的回调函数，负责最初绘图区构建时建立坐标轴以及机器人的绘制。

3.5.5 Drawline 函数

画轨迹函数。这是我们自己定义的单独的画线函数，与库中的 DrawLine 函数不同。因为画轨迹时轨迹需要带颜色，而库中的 DrawLine 函数并不能实现轨迹带颜色的功能，所以为了方便画轨迹定义了该函数。

在该函数中，通过第一个位置的参数传递，来给定含有颜色信息的结构体，从而画出具有指定颜色的轨迹。

3.5.6 Nettoie 函数

清空函数所引用的子函数。主要负责清空画板并重新画坐标系，不清空历史记录，不初始化结构体。

3.5.7 nettoie 函数

清空函数。功能比 Nettoie 更多，函数体中调用了 Nettoie 函数清空画图区域，添加坐标系，最重要的地方在于将画笔机器人初始化。

3.5.8 error_format 函数

纠错函数。当检测到输入格式、内容或命令出现错误时，会引用此函数，并根据错误的不同类型弹出不同的提示窗口来反馈给用户。

3.5.9 avance 函数

前进函数。判断输入的语句是否有格式问题，如果没有，在落笔的情况下，绘制一条按照当前角度、颜色和线条粗细前进输入距离的线，同时移动机器人到所画的端点。

3.5.10 recule 函数

后退函数。判断输入的语句是否有格式问题，如果没有，在落笔的情况下，绘制一条按照当前角度的反方向、当前颜色和线条粗细前进输入距离的线，同时移动机器人到所画的端点。

3.5.11 gauche 函数

左转函数。使当前绘制方向按照逆时针旋转所输入的角度，同时相应地改变机器人方向。

3.5.12 droite 函数

右转函数。使当前绘制方向按照顺时针旋转所输入的角度，同时相应地改变机器人方向。

3.5.13 leve_crayon 函数

提笔函数。使机器人提笔，在此条件下，机器人的轨迹被隐藏，不能显示在屏幕上。

3.5.14 baisse_crayon 函数

落笔函数。使机器人落笔，在此条件下，机器人的轨迹会显示在屏幕上。

3.5.15 couleur 函数

颜色函数。改变画笔颜色和机器人颜色。

3.5.16 allera 函数

位移函数。使画笔机器人移动到用户输入的位置。

3.5.17 cache 函数

隐藏函数。隐藏机器人，使机器人不能出现在屏幕上。

3.5.18 montre 函数

显示函数。显示机器人，使机器人显示在屏幕上。

3.5.19 robot 函数

机器人函数。利用该函数绘制机器人。为了体现出机器人移动的效果，此函数负责清空屏幕并且将数组中记录的历史轨迹重新绘制出来。在这之后，再根据显示机器人与否判断是否需要绘制机器人。

3.5.20 ligne_large 函数

线条粗细函数。设置所画轨迹和机器人线条的粗细。

3.5.21 ok 函数

利用该函数调用 valide_draw 子函数执行相应的操作。

3.5.22 d_accord 函数

利用该函数关闭当前窗口。

3.5.23 valide_darw 函数

判断指令是否有效同时绘图函数。该函数读取并检测指令的正确性，并调用相应的函数来执行操作。我们将其单独列出来而没有放到 ok 回调函数的原因是我们要在文件的部分再次用到它，所以单独将它列出来便于我们的调用。

3.5.24 TEXT 变量

文件变量。作为一个文件内部的全局变量，让 TEXT 小部件可以跨越多个文件使用。

3.6 Makefile

描述了整个程序所有文件的编译顺序、编译规则。有独特的书写格式、关键字、函数。

3.7 TOUR

测试文件之一。该文件中的指令经过读取和执行，会在可执行文件的绘图区域中绘制出西电观光塔。

3.8 PAVILLON

测试文件之一。该文件中的指令经过读取和执行，会在可执行文件的绘图区域中绘制出法国国旗。

3.9 BATEAU

测试文件之一。该文件中的指令经过读取和执行，会在可执行文件的绘图区域中绘制出帆船。

四、程序运行过程

4.1 图形化界面的建立以及输入信息的转换

4.1.1. 图形化界面的建立

本部分的程序代码源自 window.c 文件。

在主函数中定义一个类型为 location、名为 d 的结构体，是整个程序的开始。经过 initialiser 函数的初始化之后，将此结构体传入到图形化界面的绘制函数中去。其中，location 类型的结构体定义在 robot.c 文件中已经利用 typedef 语句完成，initialiser 函数的定义在 robot.h 文件中也已经完成。

如图 4：

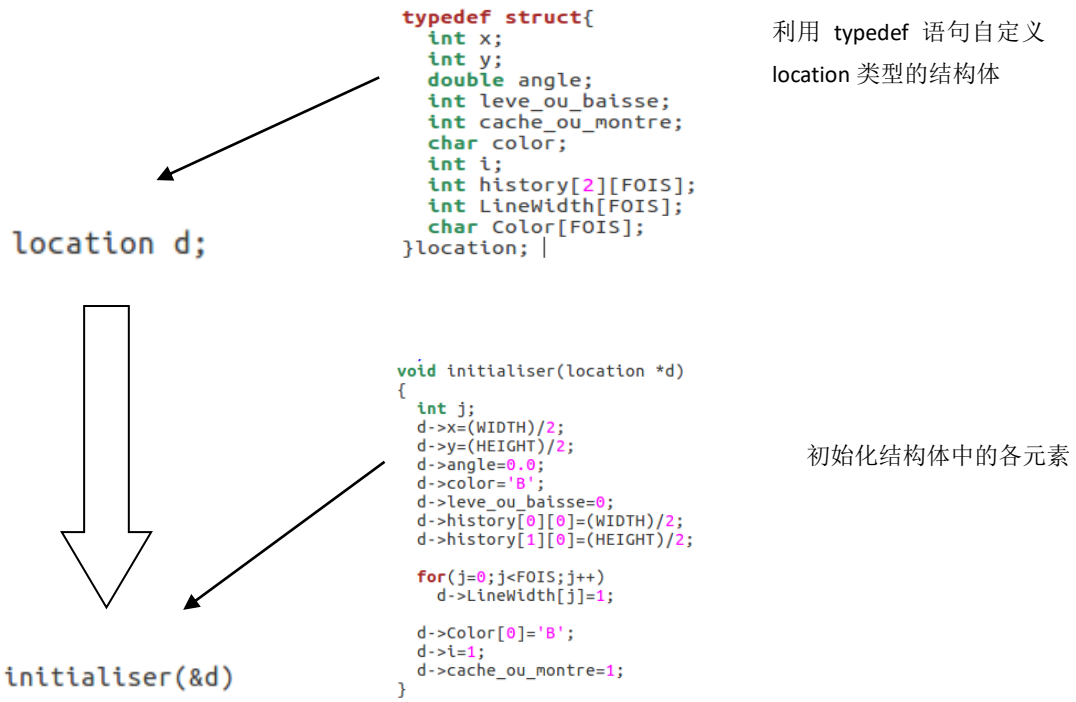


图 4

绘制函数用到五个局部的小部件和一个全局的小部件，分别为 FILE, PROG, AREA, COMTEXT, OK 和 COMMAND。

MakeDrawArea、MakeButton、MakeLabel 函数分别为制作绘图区、制作按钮、制作文本标签的函数。

多次调用 SetWidgetPos 函数，将这六个小部件按照要求排列，从而达到所需要的视觉效果（此处效果图中的坐标系是后期利用调用了其它函数实现的，在此处暂且不提）

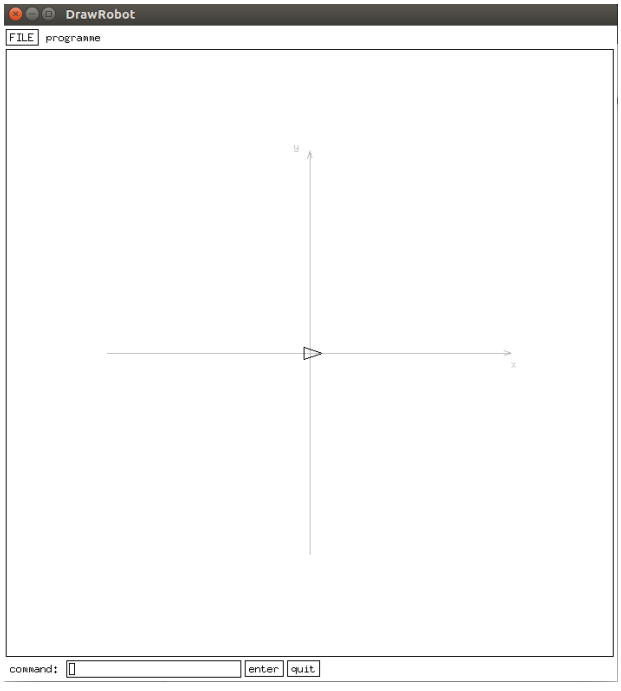
如图 5：

```
Widget FILE,QUIT,AREA,COMTEXT,OK,PROG;

AREA=MakeDrawArea(WIDTH,HEIGHT,area,d);
QUIT=MakeButton("quit",d_accord,NULL);
FILE=MakeButton("FILE",file,d);
COMTEXT=MakeLabel("command:");
PROG=MakeLabel("programme");
COMMAND=MakeStringEntry(NULL,LONGUEUR,NULL,NULL);
OK=MakeButton("enter",ok,d);

SetWidgetPos(PROG,PLACE_RIGHT,FILE,NO_CARE,NULL);
SetWidgetPos(AREA,PLACE_UNDER,FILE,NO_CARE,NULL);
SetWidgetPos(COMTEXT,PLACE_UNDER,AREA,NO_CARE,NULL);
SetWidgetPos(COMMAND,PLACE_UNDER,AREA,PLACE_RIGHT,COMTEXT);
SetWidgetPos(OK,PLACE_UNDER,AREA,PLACE_RIGHT,COMMAND);
SetWidgetPos(QUIT,PLACE_RIGHT,OK,PLACE_UNDER,AREA);
```

小部件的建立和摆放



运行成功的效果图

图 5

4.1.2 输入信息的转换

本部分的代码源自 callback.c 文件。

在用户输入完指令之后，点击 OK 按钮启动回调函数，同时将指针 d 传入 callback.c 文件中的 ok 函数中，此时便开始了对画笔机器人的赋值和应用.

ok 函数。先定义了三个指针，cd、q 和 p。

（1）cd 一开始先指向用户输入的指令的首位置，定义了 location 型指针 l 并指向画笔机器人之后，把 cd 和 l 一同传递给 valide_draw 函数。

如图 6：

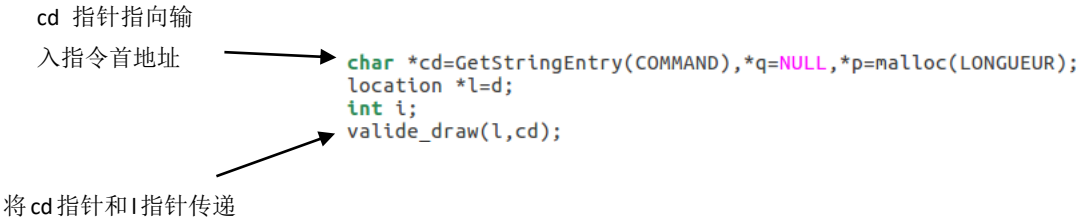


图 6

（2）利用指针 q 和 p 来对已经写有具体指令的指令框进行初始化——做清空处理用 p 开辟一个指令框长度的动态数组之后，利用 p 来初始化。在这之前应该**让 q 来先指向 p 所指向的位置**，以防止初始化结束后，p 指针指向不明确，从而出现无法 free 的情况。把 q 所指向的地方 free 掉。这样就实现了每输入条指令之后将该段内存空间释放，可以接着输入下一条指令的操作。

如图 7：

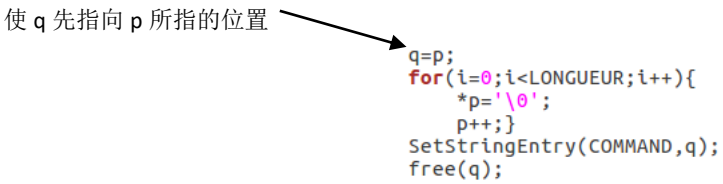


图 7

（3）对于 cd 指针和指向画笔结构体的指针传递给 valide_draw 函数之后，就结束了把用户输入的指令转成 C 语言可以直接使用的信息的过程。

4.2 输入信息的检测和分配

4.2.1 输入信息的检测

本部分的程序代码源自 callback.c。

cd 指针和 l 指针传递给 valide_draw 函数之后，此函数便对 cd 指针所指向的字符串进行分析：

（1）首先让 a 指针指向 cd，即用户输入的指令；b 指针指向 NULL。根据指针 a 判断，如果首位字符为不为大写字母则提示错误 1，并要求重新输入。否则，则使 cd 指针进入循环，直到遇到空格或者字符串结束符为止。

如图 8：

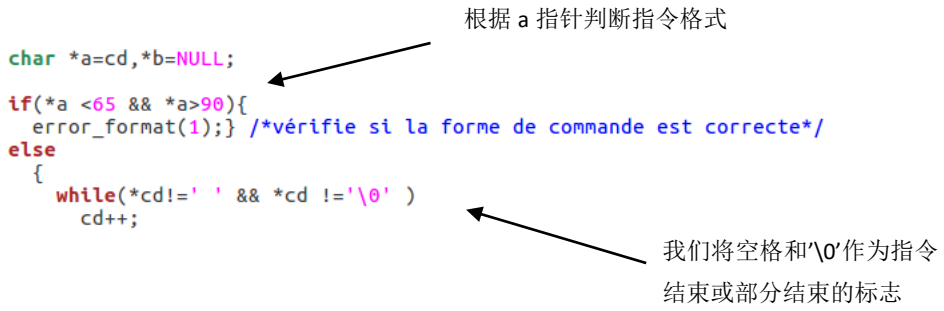


图 8

（2）如果遇到的是结束符的话，令 b 也指向这个结束符。如果是空格的话，**令此位置赋值为结束符**（以便于判断前半句指令是否是正确的输入）令 b 指向 cd 的下一位。

此时，cd 所指向的字符串已经从所输入指令的空格处**被结束符完全断成两部分**（如果可以的话），a 所指向第一部分，b 指向第二部分。

如图 9：

图 9

4.2.2 输入信息的分配

本部分的程序代码源自 callback.c。

接下来的 if 和 else if 语句是判断 a 所指向的具体是哪一部分，在此我们借助了 strcmp 比较函数。

如果有某一条符合的指令后，就成功的调用对应的函数，并将指向画笔结构体的指针作为实参传递给了这个函数。完成了对输入信息的检测以及合理的调用对应的函数的过程。

如图 10：

```
if(strcmp(a,"AVANCE")==0)
  avance(b,l);
else if(strcmp(a,"RECULE")==0)
  recule(b,l);
else if(strcmp(a,"GAUCHE")==0)
  gauche(b,l);
else if(strcmp(a,"DROITE")==0)
  droite(b,l);
else if(strcmp(a,"LEVE_CRAYON")==0)
  leve_crayon(b,l);
else if(strcmp(a,"BAISSE_CRAYON")==0)
  baisse_crayon(b,l);
else if(strcmp(a,"COULEUR")==0)
  couleur(b,l);
else if(strcmp(a,"NETTOIE")==0)
  nettoie(b,l);
else if(strcmp(a,"ALLERA")==0)
  allera(b,l);
else if(strcmp(a,"CACHE")==0)
  cache(b,l);
else if(strcmp(a,"MONTRE")==0)
  montre(b,l);
else if(strcmp(a,"LIGNE_LARGE")==0)
  ligne_large(b,l); /*choisi la fonction correspondante*/
else
  error_format(2);
```

图 10

4.3 输入信息内容的判断回馈

本部分的程序代码源自 callback.c。

信息内容的判断回馈只用到了 error_format 函数。

首先建立一个新的含有一个标签语句和一个按钮的窗口，通过标签上显示的语句来提示输入错误的内容。点击按钮确认来关闭窗口。

此函数只有一个整型的形参，通过形参的不同来分辨错误的情况。

预设的可能出现的错误的情况有：

情况 1：输入命令的格式错误。显示内容：请按照正确的格式输入指令。

如图 11：

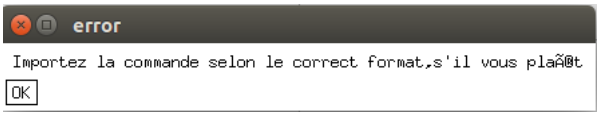


图 11

情况 2：命令大小写错误。显示内容：请写正确的命令，命令要大写。

如图 12：

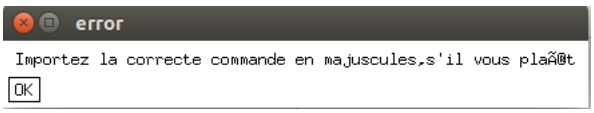


图 12

情况 3：需要后接数字的指令后，没加空格。显示内容：在指令的后面，您必须输入一个空格加数字。

如图 13：

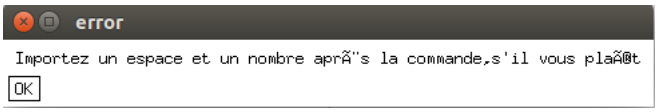


图 13

情况 4：不需要接数字的指令后，输入数字。显示内容：此命令语句后面不能加数字。

如图 14：

图 14

情况 5：ALLERA 的书写格式错误。显示内容：请按照如下方式书写：ALLERA x y

如图 15：

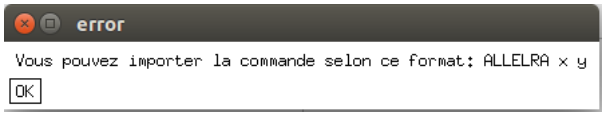


图 15

情况 6：颜色输入错误。显示内容：请输入如下颜色：RED YELLOW BLACK WHITE GREEN

如图 16：

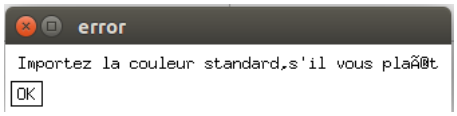


图 16

情况 7：指令超过边界。显示内容：超出边界请重新输入。

如图 17：

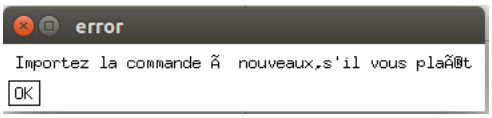


图 17

4.4 绘图区控制与机器人的生成

本部分的程序代码源自 callback.c。

调用函数 MakeDrawArea 完成绘图区的产生，同时调用绘图区的回调函数 area 完成坐标系和机器人的绘制。

robot 函数是整个控制绘图区的核心。任何一个更改绘图区的指令，除了清空指令，都需要经过此函数的处理。

robot 函数的形参只有 location 类型的结构体 l，即在主函数中声明的画笔机器人。下面来对此函数做详细介绍：

（1）定义整型变量 j 且将结构体中的 i（**初始化为 1**）赋值给 j

（2）调用 Nettoie 函数做屏幕清空（不改变结构体内容）

（3）判断 j 是否等于 1（**第一次调用 robot 函数时，j==1**）

（4）若不等于 1 则说明做过机器人位移变换（**即之前已经调用过 robot 函数**）

（5）在此情况下，可以**从第一个点开始，将之前所有**颜色不为‘X’的轨迹、设置完粗细之后给画出来。

以上流程见图 18：

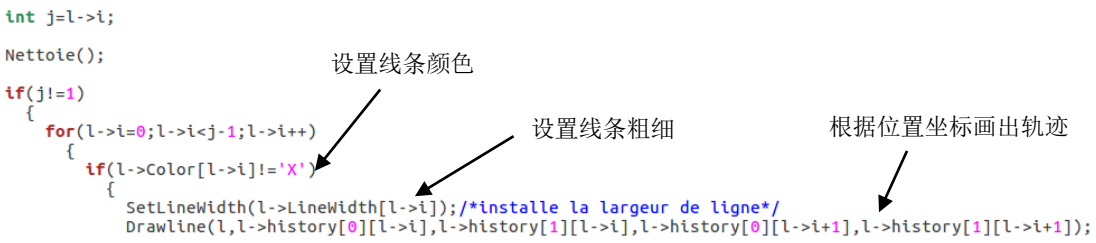


图 18

（6）如果等于 1 则说明机器人未做过位移变换，是刚刚开始启动程序，因此不需要画轨迹。

以上程序是实现一次画轨迹的过程。

PS：设置线宽为当前线宽后画出线来。在此用到了我们自己设置的画线函数，此函数引用了一个结构体变量，它可以检测结构体中的 Color 数组，并且按此数组所对应的颜色设置前景色。

（7）再把 j 赋值给结构体中的 i（**j 起到了一个保留原来的 i 值的作用**）

（8）再次设置线宽以便于画出当前机器人

（9）画机器人之前检测是 cache_montre 变量是否为 1

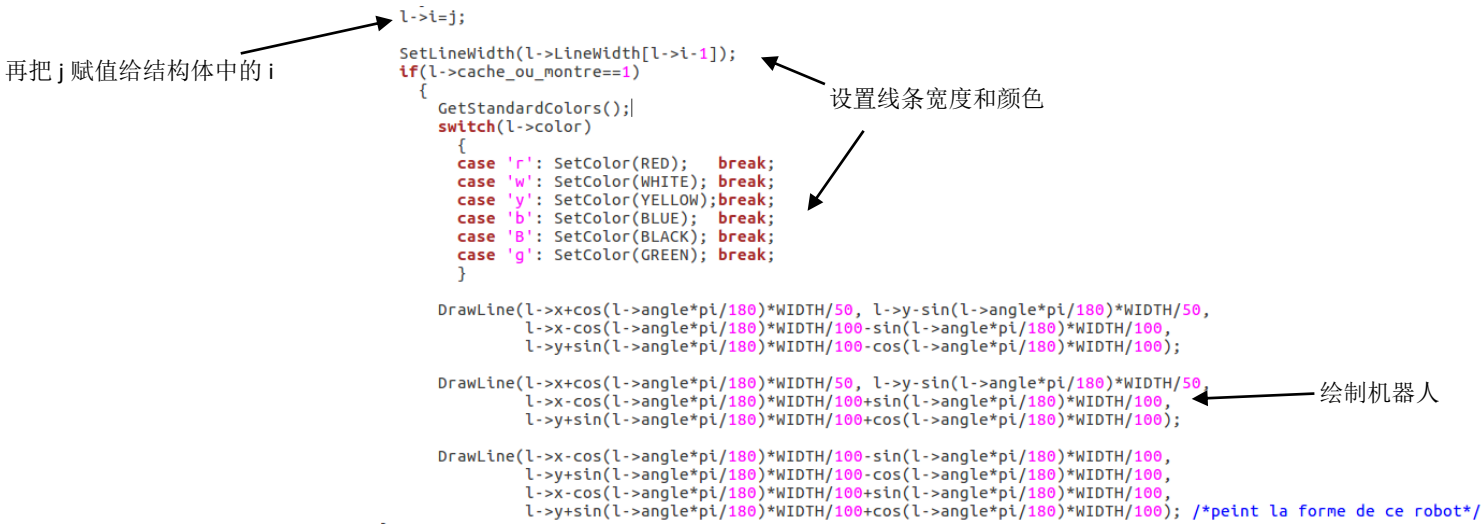
（10）如果为 1，则画出机器人

（11）机器人画之前，首先要检测并设置当前色，才能保证画笔颜色与画出的线条颜色相同

（12）然后以三角形的重心为参考点，分别算出三个顶点的位置，用 DrawLine 函数将其画出。

PS：我们设计的机器人的形状为一个等腰不等边三角形，顶点角所指的方向即机器人的方向

以上流程见图 19：




```
l->i=j;

SetLineWidth(l->LineWidth[l->i-1]);
if(l->cache_ou_montre==1)
  {
    GetStandardColors();
    switch(l->color)
      {
      case 'r': SetColor(RED);   break;
      case 'w': SetColor(WHITE); break;
      case 'y': SetColor(YELLOW);break;
      case 'b': SetColor(BLUE);  break;
      case 'B': SetColor(BLACK); break;
      case 'g': SetColor(GREEN); break;
      }

    DrawLine(l->x+cos(l->angle*pi/180)*WIDTH/50, l->y-sin(l->angle*pi/180)*WIDTH/50,
             l->x-cos(l->angle*pi/180)*WIDTH/100-sin(l->angle*pi/180)*WIDTH/100,
             l->y+sin(l->angle*pi/180)*WIDTH/100-cos(l->angle*pi/180)*WIDTH/100);

    DrawLine(l->x+cos(l->angle*pi/180)*WIDTH/50, l->y-sin(l->angle*pi/180)*WIDTH/50,
             l->x-cos(l->angle*pi/180)*WIDTH/100+sin(l->angle*pi/180)*WIDTH/100,
             l->y+sin(l->angle*pi/180)*WIDTH/100+cos(l->angle*pi/180)*WIDTH/100);

    DrawLine(l->x-cos(l->angle*pi/180)*WIDTH/100-sin(l->angle*pi/180)*WIDTH/100,
             l->y+sin(l->angle*pi/180)*WIDTH/100-cos(l->angle*pi/180)*WIDTH/100,
             l->x-cos(l->angle*pi/180)*WIDTH/100+sin(l->angle*pi/180)*WIDTH/100,
             l->y+sin(l->angle*pi/180)*WIDTH/100+cos(l->angle*pi/180)*WIDTH/100); /*peint la forme de ce robot*/
```

图 19

到此就实现了每移动一次的清屏与重新按照历史绘制所有的线的过程，从而也实现了机器人的移动也顺带实现了机器人的显示与隐藏。

4.5 机器人前进、后退和定位

本部分的程序代码源自 callback.c。

在以上功能完成的基础上，机器人的位移，即绘制轨迹的操作（如果是落笔状态的话），就很容易实现了。其本质就是改变结构体的中的数字，然后调用 robot 函数进行绘制。

（1）机器人的前进过程分为两部分：**检测部分**和**执行部分**。

检测部分又分为两部分：

第一部分为指令的改错。如果 b 指针所指向的字符串的首位不为数字的话，则调用纠错函数的情况 3，以及以后的需要输入数字的指令，在没输入数字时的错误处理方式和此处相同，后面的部分不再赘述。

如图 20：

```
if(*b>57 || *b<48)
  error_format(3);
```

图 20

第二部分为边界检测。如果位移之后的坐标超出了边界，则调用改错函数的情况 7。在接下来的后退和定位操作中，也与此相同。

代码如图 21：

```
if((strtod(b,NULL)*cos(l->angle*pi/180) + l->x > WIDTH || strtod(b,NULL)*cos(l->angle*pi/180) + l->x < 0)
   || (-strtod(b,NULL)*sin(l->angle*pi/180) + l->y > HEIGHT ||-strtod(b,NULL)*sin(l->angle*pi/180) + l->y < 0))

  error_format(7); /*vérifie si la trace est hors de portée*/
```

图 21

在检测无误之后，便开始执行对应操作：

如果落笔，将当前颜色赋值给颜色数组。

如果抬笔，情况是**机器人前进而未画线**。这时需要注意，要**把颜色赋值给位移之后的位置所对应的颜色数组**，而**位移之前的位置所对应的颜色数组需要赋值为‘X’，以代表此处是颜色为空的只位移不画线的情况。**

如图 22：

```
if(l->leve_ou_baisse==1)
  l->Color[l->i-1]=l->color;
else
  {
    mid=l->Color[l->i-1];
    l->Color[l->i-1]='X';
    l->Color[l->i]=mid;
  }
```

落笔情况，颜色赋给当前位置对应的颜色数组

抬笔情况，颜色赋给位移后位置对应的颜色数组

图 22

读取 b 所指的字符串并且转换成 double 型，乘上对应的三角函数之后与原来的坐标值进行求和运算，得到新的坐标值。注意，**此处有坐标的转换**。因为**系统默认的坐标原点的位置是在左上角**，且 **y 轴正方向指下**，而**习惯的是将坐标原点置于中心并且 y 轴正方向指上**，所以每次涉及到坐标的情况，要做坐标的转换。

如图 23：

```
l->x = strtod(b,NULL)*cos(l->angle*pi/180) + l->x;
l->y =-strtod(b,NULL)*sin(l->angle*pi/180) + l->y;
```

图 23

将新的点写在 history 数组中以**记录新的点**，因为此函数并不改变线宽，所以将线宽数组上一个位置的线宽赋值给新的位置。

使结构体中的 i 自加，以便于下一个点的记录。

最后，调用 robot 函数，实现画线。

以上流程见图 24：

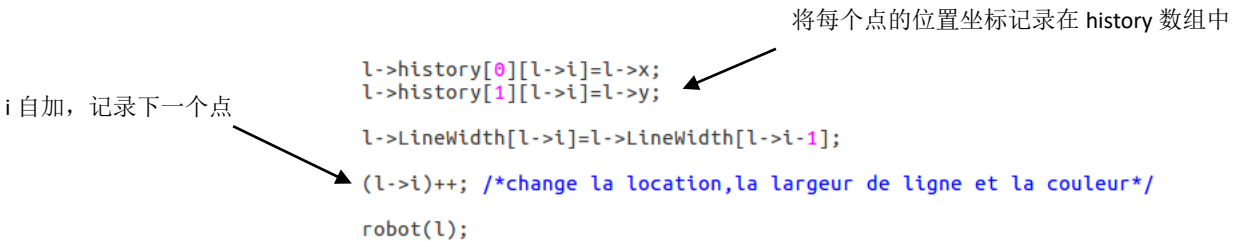


图 24

机器人前进实际效果图（抬笔前进 100 个单位）：

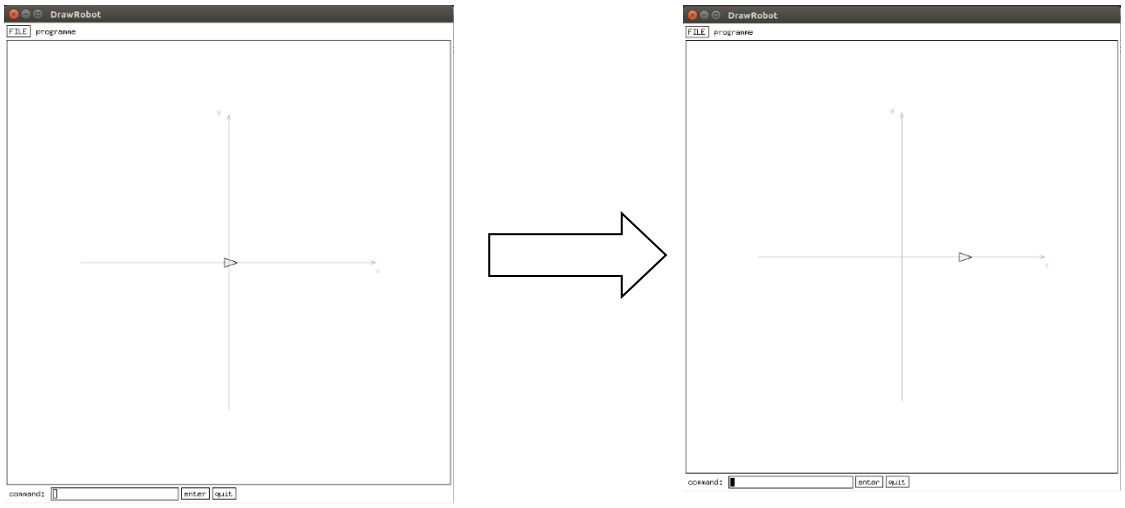


图 25　　图 26

（2）后退函数和 avance 相同，只是在坐标转换时，新坐标的位置要变换一

下正负号。

如图 27：

注意 strtod 函数前的符号发生了改变

```
l->x =-strtod(b,NULL)*cos(l->angle*pi/180) + l->x;
l->y =strtod(b,NULL)*sin(l->angle*pi/180) + l->y;
```

图 27

机器人后退实际效果图（抬笔后退 100 个单位）：

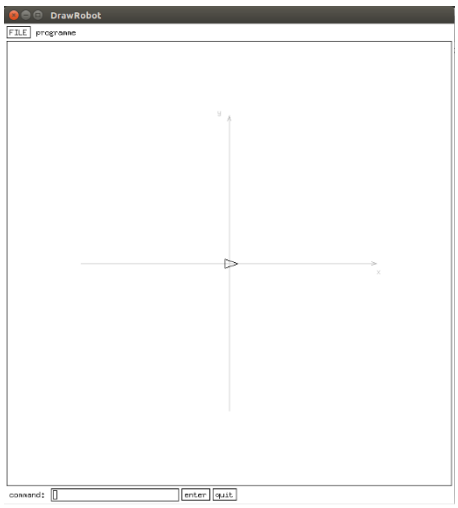


图 28

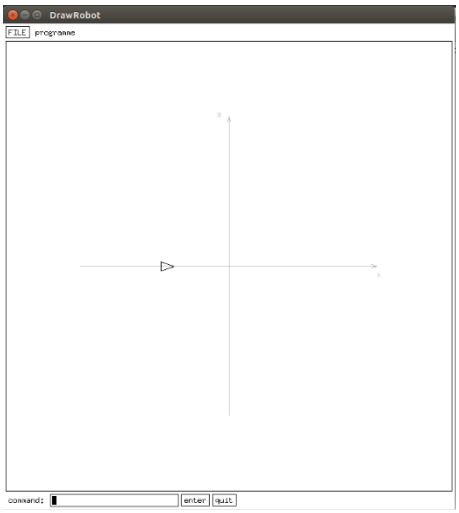


图 29

（3）定位函数由于输入较为特殊，需要在检测语句多加一步。之后将坐标直接赋值为新的坐标其他和 avance 相同。

如图 30、图 31：

检测语句与 avance 和 recule 有区别

```
if((*b>57 || *b<48) && *b!='-')
  error_format(5);
```

图 30

新坐标直接赋值

```
l->x=strtod(a,NULL)+WIDTH/2;
l->y=-strtod(b,NULL)+HEIGHT/2;
```

图 31

机器人定位实际效果图（从（0,0）定位到（100,100））：

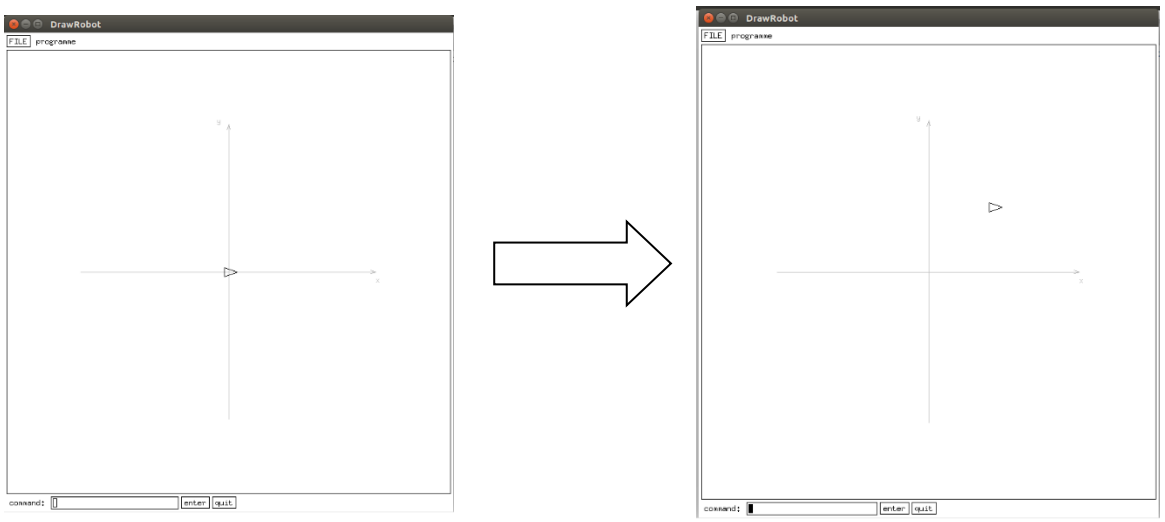


图 32　　图 33

4.6 机器人转向和画笔粗细的调节

4.6.1 机器人转向

本部分的程序代码源自 callback.c。

机器人的转向，检测与上述相同，具体步骤是将结构体中的角度这个变量给按照要求加减输入的数字。我们规定**左转为加，右转为减**。注意该角度是以**弧度制**为单位。

我们以左转为例。

源代码如图 34：

```
if(*b>57 || *b<48)
  error_format(3); /*vérifie si la second partie de commande est un nombre*/
else
  l->angle=l->angle + strtod(b,NULL); /*change l'angle*/
robot(l);
```

图 34

机器人左转实际效果图（在原点左转 90°）：

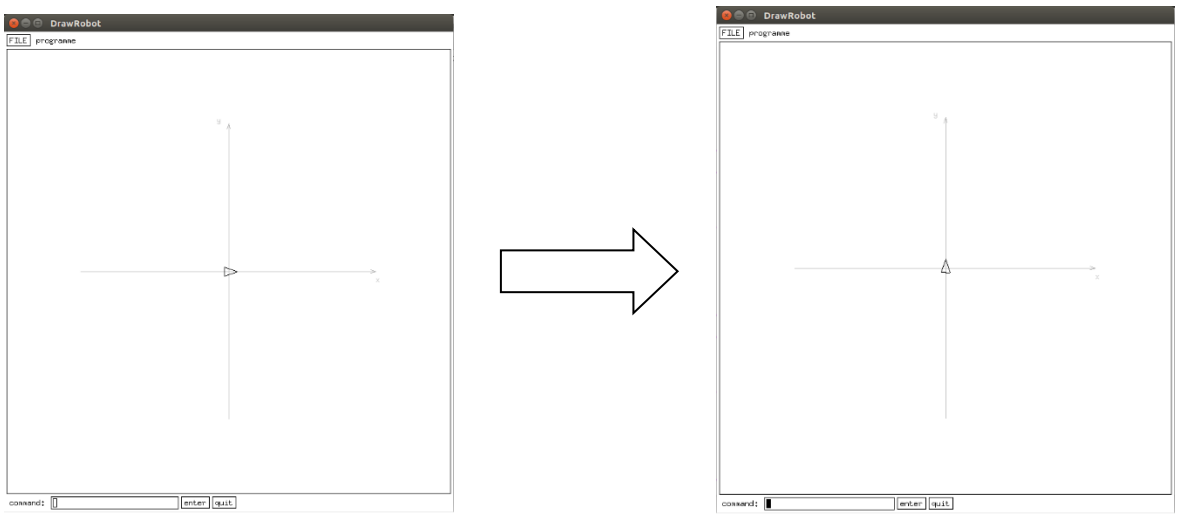


图 35　　图 36

4.6.2 画笔粗细的调节

本部分的程序代码源自 callback.c。

画笔粗细的调节，和转向相同，只是把结构体中的线宽这一变量附上具体的宽度值。

然后调用机器人函数，通过新的机器人的方向不同来表现出方向的改变或者粗细的改变。

如图 37：

```
if(*b>57 || *b<48)
  error_format(3);
else
  {
    l->LineWidth[l->i-1] = strtod(b,NULL);
    robot(l);
```

图 37

画笔粗细调节实际效果图（由默认值 1 调节至 5）：

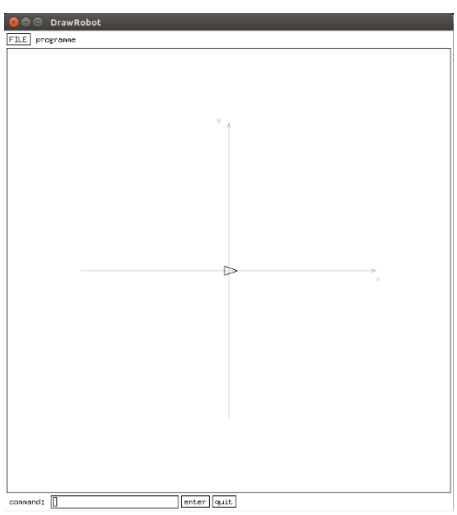


图 38

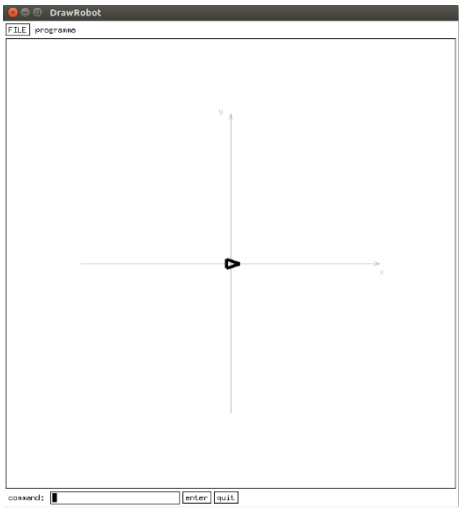


图 39

4.7 机器人抬笔落笔和显示隐藏

本部分的程序代码源自 callback.c。

机器人的这四个操作，因为指令后不需要加任何内容，所以检测用到的是情况 4。具体语句，是改变结构体中的 leve_ou_baisse 或 cache_ou_montre 变量。

（1）改变第一个变量，可以影响 avance、recule 和 allera 的函数的画线步骤。

leve_ou_baisse==0 时，即表示抬笔状态，此时体现出来的是机器人轨迹不显示在屏幕上。

leve_ou_baisse==1 时，即表示落笔状态，此时体现出来的是机器人轨迹显示在屏幕上。

下图分别表示抬笔和落笔状态下，机器人从（0,0）前进 100 单位的实际效果：

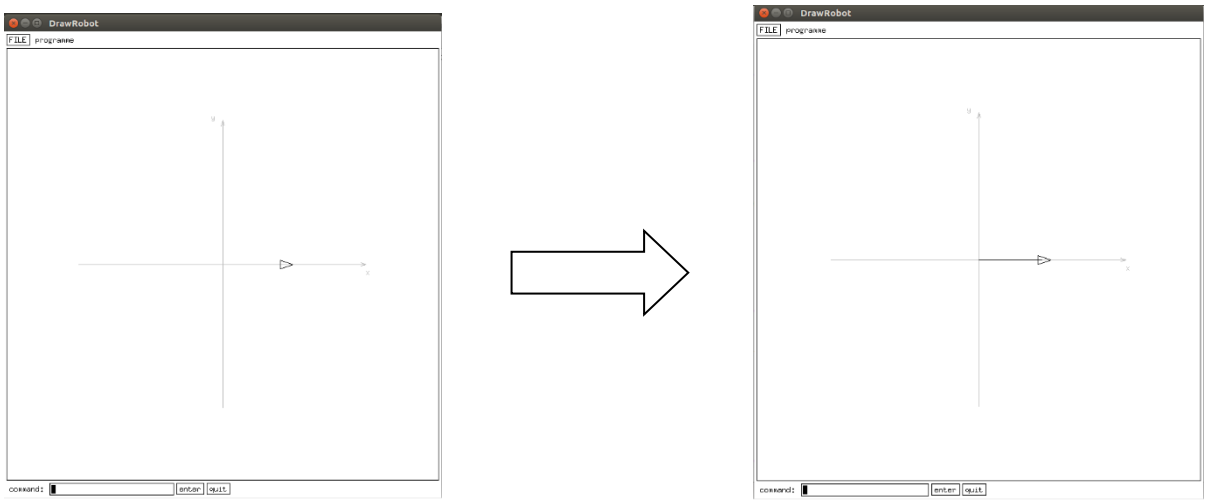


图 40　　图 41

（2）改变第二个变量，可以影响到 robot 函数中的画机器人的部分，体现出来的是是否画出机器人。此处，调用 robot 函数来体现这一步骤。

cache_ou_montre==0 时，即表示隐藏机器人状态，此时体现出来的是机器人不显示在屏幕上。

cache_ou_montre==1 时，即表示显示机器人状态，此时体现出来的是机器人显示在屏幕上。

下图分别表示机器人在隐藏和显示状态下，位于（0,0）的实际效果：

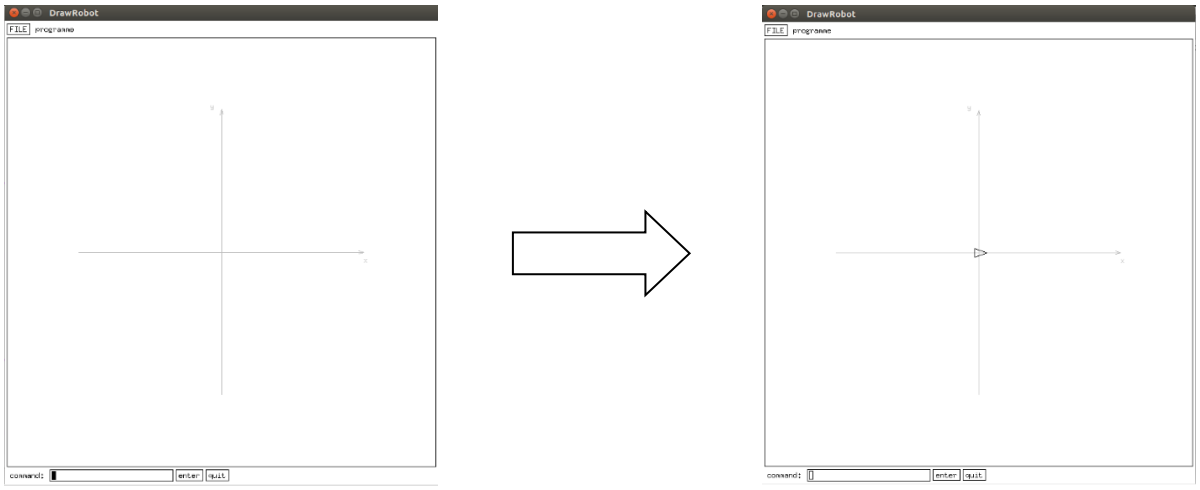


图 42　　图 43

4.8 上色

本部分的程序代码源自 callback.c。

上色操作的源代码也很简单，在检测时，检测后面 b 所指的字符串是否为标准库中的颜色，如果不是的话，出现情况 6。

如果是标准色，将结构体中的 color 变量赋值为相应的颜色。

红色 RED　　为‘r’

绿色 GREEN　为‘g’

蓝色 BLUE　　为‘b’

黄色 YELLOW 为　‘y’

白色 WHITE　为　‘w’

黑色 BLACK　　为　‘B’

然后调用 robot 机器人并且通过新的机器人颜色的不同来表现出颜色的改变。

为了更清晰直观的看到该函数的效果，我们以红色为例，在线条宽度为 3 的情况下，表示机器人在（0,0）处的状态，如图 44：

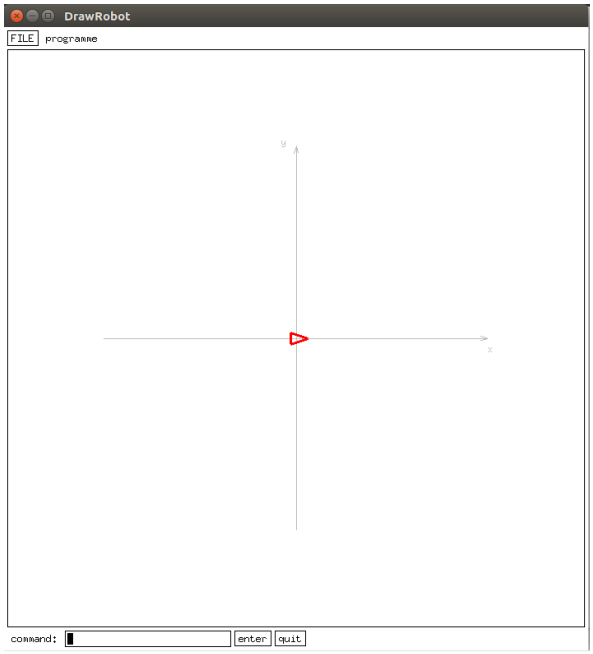


图 44

4.9 清空绘图区并初始化机器人

本部分的程序代码源自 callback.c。

清空操作，因为指令后不需要加任何内容，所以检测用到的是情况 4。具体内容是调用 Nettoie 函数，做表面上的清空，然后调用初始化函数 initialiser 来做结构体的初始化，从而真正实现回到最初的状态。调用 robot 函数来体现这一操作。

如图 45：

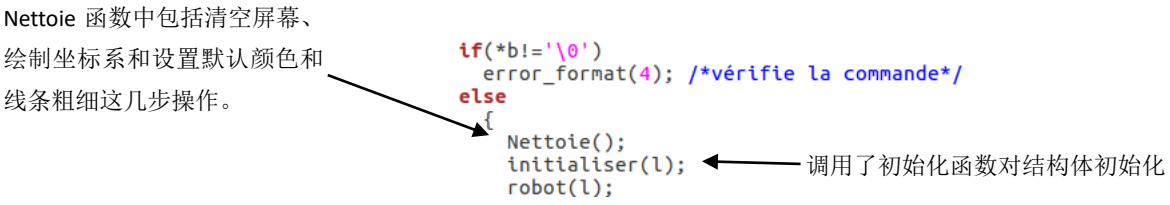


图 45

执行完清空操作的实际效果与程序刚开始运行时状态一致，在此略去配图。

4.10 文件指令的输入与执行

本部分的程序代码源自 callback.c。

程序概况中已经交代过，程序主界面上有一个名为“FILE”的按钮，用户点击该按钮，即进行文件读取操作。

整个文件指令的输入过程用主要到了两个关键的文件函数：**fopen** 和 **fgets** 函数。

首先调用 fopen 函数检测输入的文件名是否正确（程序所在文件夹内是否存在所读取的文件）。

如图 46：

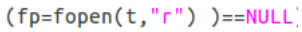


图 46

如果文件名字错误，弹出窗口提示文件不存在并且清空输入框。详细操作和

错误指令操作相同，不再赘述。

实际运行过程中的报错窗口如图 47：

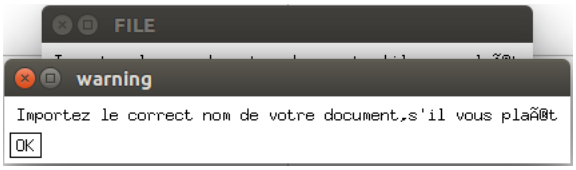


图 47

如果文件名字正确，启动 while 循环。

fgets 函数的工作原理是按行来读取文件中的信息并且返回到函数第一个位置的指针中。我们借助这个指针，进行第一步操作：检测指针 T 所指的字符串中的换行符，**将换行符换为字符串结束符**，以便符合之前程序的输入格式上的要求。

如图 48：

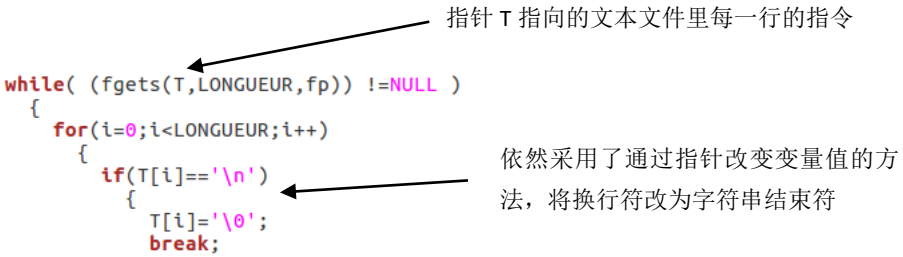


图 48

这一步操作结束之后，指针 T 所指向的这一行指令在正确的情况下就与在 programme 环境下用户自行输入的指令完全相同了。我们在此基础上调用 valide_draw 函数即可分情况进行上述绘图操作。之后，当未检测到 EOF 文字流终止符之前，重复绘画操作，直至文本结束。以上就是文件中指令的提取、输入、分析和执行。

我们以 TOUR 测试文件为例更具体的展示一下文件读取功能。

下图是 TOUR 测试文件的指令：

```
ALLERA -210 -100
BAISSE_CRAYON
LIGNE_LARGE 3
AVANCE 40
GAUCHE 70
AVANCE 75
DROITE 150
AVANCE 100
GAUCHE 150
AVANCE 70
DROITE 145
AVANCE 40
GAUCHE 75
AVANCE 190
GAUCHE 90
AVANCE 180
GAUCHE 90
AVANCE 16
DROITE 90
AVANCE 30
DROITE 90
AVANCE 16
GAUCHE 90
AVANCE 40
DROITE 90
AVANCE 20
GAUCHE 90
AVANCE 36
DROITE 90
AVANCE 12
DROITE 90
AVANCE 14
GAUCHE 90
AVANCE 14
DROITE 90
AVANCE 273
GAUCHE 90
AVANCE 60
LEVE_CRAYON
ALLERA 125 188
GAUCHE 90
LIGNE_LARGE 2
BAISSE_CRAYON
AVANCE 60
```

图 49

我们在指令框内输入它的名称：

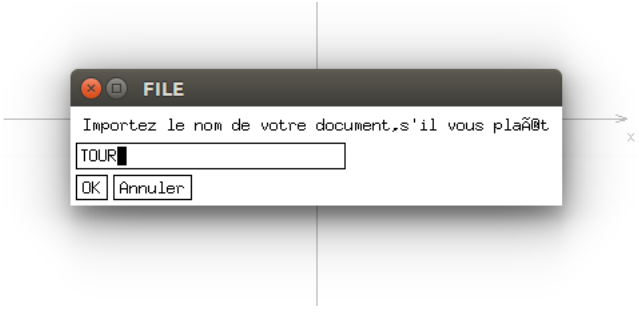


图 50

TOUR 文件便导入到程序中，并根据指令绘制出了相应的图形。

如图 51：

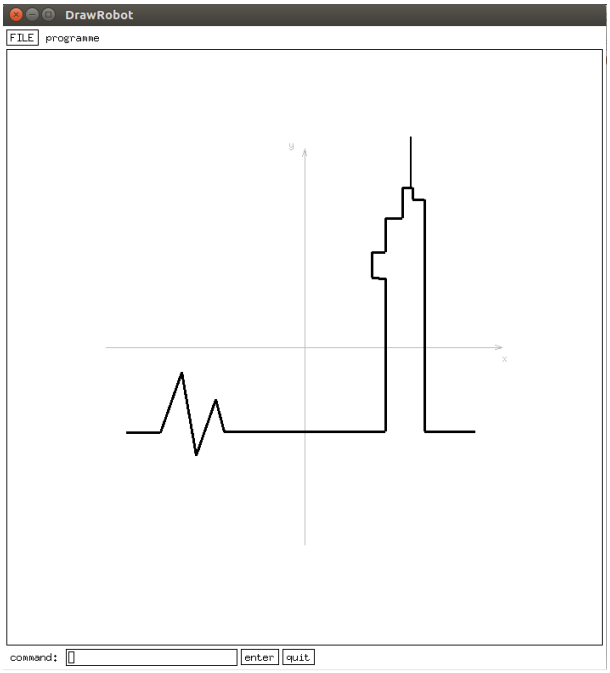


图 51

五、程序的使用说明

1．您输入的所有指令必须大写。

2．机器人定位、机器人前进、机器人后退、机器人左转、机器人右转、设置线条粗细共计六个指令，您需要在语句输入完毕后，输入空格，然后再输入数字以执行该条指令。

其中，机器人定位的指令您需要在输入两次空格和数字。

3．上色指令需要您在语句输入完毕后，输入空格，再输入标准色库（红、绿、蓝、黄、白、黑）中的任意颜色的大写名称以完成指令。

4．抬笔、落笔、隐藏机器人、显示机器人、清空绘图区共计五个指令，您的语句输入完毕即代表指令输入完毕。

5．我们定义的绘图区为 700*700 单位的正方形区域，请您在执行操作时不要超出该区域。

6．我们定义的指令长度为 200 单位，请您的指令长度不要超出 200 单位。

7．我们定义本程序最多连续执行 100 条指令，即在 programme 环境下，您在指令框中最多连续输入 100 条指令完成操作，在 FILE 环境下，您自行编写的文件指令不能超过 100 行。

8．您在使用文件读取功能时，设置的文件名也要全部为大写字母。

六、程序总结

1．程序内容

本次程序的内容是借助 Libsx 库，创建一个可操作的图形化人机交互界面和一个机器人，在 programme 环境和 FILE 环境下，使得机器人在一个方形区域进行移动，完成用户输入的一系列指令。

指令包括：——机器人前进

——机器人后退

——机器人左转

——机器人右转

——机器人定位

——机器人抬笔

——机器人落笔

——上色

——显示机器人

——隐藏机器人

——加宽线条（自创）

2．编写过程中遇到的问题

（1） 差错控制，必须要尽可能的考虑到所有的情况，我们在编写该部分时花费了很多时间和精力

（2） 如何实现机器人的隐藏和轨迹隐藏，这是本次程序编写过程中遇到的最大的问题，经过多次不同方法的尝试，我们最终决定借助历史数组来实现这一功能。

3.仍需改进的地方

（1） 后期检查中发现本程序中定义的结构体完全可以用链表替代，但限于时间紧张，没能完成修改。

（2） 程序的绘图区域、最大指令长度和最大指令条数的数值都不够大，程序使用有一定的局限性。

（3） 机器人的形式过于简单。

（4） 机器人的左转和右转功能中，角度存在误差。