# Developer's Documentation

The snake game is developed using SDL 1.2 in C language. The functions used do not actually create and object, there is no "object snake" that walks and eats apple. The logic to make the game work is in the positions of the drawings. Every element is a drawing using SDL functions to show in the screen, or the *SDL*_Surface. What is used is the position of these drawings to make the game work. When the snake walks from a to b, there are 3 actions involved, which are: Erasing the snake from the a position, giving the b position for the snake and redrawing the snake in the b position. Whenever the snake eats an apple, it happens because the drawing of the snake is the same and the drawing of the apple, this collision is detected based of the x and y coordinates of the elements and then the program reacts. With that in mind, it is known that many functions will be related to the position of the drawings and erasing and redrawing elements. The program is divided in three sections: the MACRO definitions, the FUNCTIONS and the main. The sections also have subsections that will be explained in the future. Starting with the MACRO:

## MACRO DEFINITIONS

Those definitions were made because these are values frequently used throughout the program in different functions and occasions. In addition to that, it is easier to understand and develop when it is not needed to memorize the values but instead use the words for it. The first ones are:

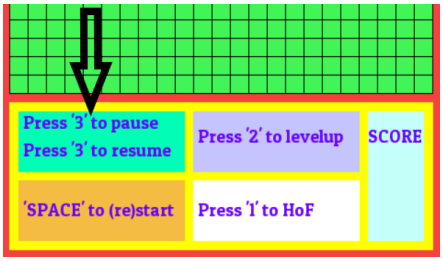
```
1.  //GameSpace specifications
2.  #define GAME_SPACE 460
3.  #define CELL_SIZE 20
4.  #define BORDER 10
5.
6.  //Window specifications
7.  #define WINDOW_WIDTH 480
8.  #define WINDOW_HEIGTH 640
9.
10. //Snake
11. #define SNAKE_RADIUS 9
12. #define SNAKE_VELOCITY 20
```

These are specifications of the size of the game, since SDL is opened in a new window, you must tell the specifications of the window. GAME_SPACE is the size of the game grid and CELL_SIZE is the size of each cell of the grid. The Snake's definitions are the size of the radius of snake and its velocity. Every value is in pixel.

```
1.  //Moving specs
2.  #define LEFT 1
3.  #define DOWN 2
4.  #define RIGHT 3
5.  #define UP 4
6.  #define LEVEL_up 25
7.  #define LEVEL 100
8.
9.  //Bottom of Screen specs
10. #define BTN_HEIGHT 60
11. #define BTN_WIDTH 180
12.
13. //Colors
```

```
#define BACKGROUND_COLOR 0x42F456FF
#define GRID_COLOR 0x000000FF
#define BLACK 0x000000FF
#define APPLE_COLOR 0xF442D7FF
#define SNAKE_COLOR 0xF44242FF
#define BOTTOM_COLOR 0XFFFF00FF
#define PAUSE_BTN 0x00FFB6FF
#define RESET_BTN 0xF4BC42FF
#define LEVEL_BTN 0xC4C4FFFF
#define SCORE_BTN 0xC4FFFAFF
#define HOF_BTN   0xFFFFFFFF
```

After that, we have the directions definitions, that set each direction one value. The level will be used in the TIMER, an SDL subsystem that will be explained later. The BTN definitions are the size of the rectangles in the bottom of the game as seen below. And the colors definitions used in the SDL functions in the RGBA format.




The most important definitions of this sections are related to the snake and the apple.

```
    /*Snake segment*/
typedef struct Snake_elem{
    int x; int y;
    int vx; int vy;
    struct Snake_elem *next;
    int dir;
}Snake_elem;

//Apple
#define APPLE_RADIUS 7
typedef struct{
    int x; int y;
}Apple;
```

The snake segment *"Snake_elem"* is the key to the program. The snake is a linked list that stores: Its position in x and y; its velocity in x (vx) and y (vy); the pointer to the next segment of the snake and its direction. With this linked list, we can access each element and every element will be linked to the head, that is the main element of the snake. The *"Apple"* is a struct that stores its position in x and y and there is its radius.

These definitions are very important because they will be used frequently in the functions that will come.

## Functions

The functions are responsible to make the program works. Basically, every action that happens in the MAIN code, calls a function. They are also implemented in different situations so one function is used more than once. Inside the functions, there are 4 sections: The GAME section, responsible to initialize the game itself, in this case is most of them are related to SDL; The HALL OF FAME section, that manages the scores in the game and does file handling; The SNAKE section

is responsible for every snake action, that is moving, creating new segment, deleting snake, checking if it touched the wall, resetting the snake and initializing it; and there is the APPLE section that creates the apples. Each of them will be explained separately.

### Game Section

The first function we have is *draw_field*, a function only called once to draw the field of the game. It receives as parameter and SDL subsystem that is the screen and just modify it since is passed a pointer to it.

```
1.  //Function that draws the field of the game
2.  void draw_field(SDL_Surface *field){
3.      boxColor(field,0,0, WINDOW_WIDTH, WINDOW_HEIGTH, SNAKE_COLOR);
4.      //draw the the game field
5.      boxColor(field, BORDER, 0, WINDOW_WIDTH-
    BORDER, GAME_SPACE, BACKGROUND_COLOR);
6.      //draw the lines of the field
7.      int i;
8.      for( i = 0; i <= GAME_SPACE; i=i+CELL_SIZE){
9.          hlineColor(field, BORDER, GAME_SPACE+BORDER, i, GRID_COLOR);
10.         vlineColor(field, i + BORDER, 0, GAME_SPACE, GRID_COLOR);
11.     }
12. }
```

The *boxColor* function is a SDL function that draws a filled rectangle. The for loop is used to draw the lines of the field using two SDL functions, the *hlineColor* and *vlineColor*. The next function is the *draw_bottom*. It receives the SDL_surface object and modify it to draw the bottom of the game where there are the instructions (look like buttons) and the score. Since one block of code is repeated many times, one will be explained, and the others may just change the color or position of the drawing.

```
1.  void draw_bottom(SDL_Surface *field, TTF_Font *font){
2.      //Text general info
3.      SDL_Surface *textimg;
4.      SDL_Color text_img_color = {106, 0, 255};
5.      SDL_Rect where_to = { 0, 0, 0, 0 };
6.      //Bigger bottom box
7.      boxColor(field, BORDER, GAME_SPACE+BORDER, WINDOW_WIDTH-
    BORDER, WINDOW_HEIGTH-BORDER, BOTTOM_COLOR);
8.      //Buttons
9.          /*pause button*/
10.     boxColor(field, 2*BORDER, GAME_SPACE+2*BORDER, 2*BORDER+BTN_WIDTH, GAME_SP
    ACE+2*BORDER+BTN_HEIGHT+5, PAUSE_BTN);
11.         //pause btn text
12.     textimg = TTF_RenderUTF8_Blended(font, "Press '3' to pause", text_img_colo
    r);
13.     where_to.x = 2*BORDER+5;
14.     where_to.y = GAME_SPACE+2*BORDER;
15.     SDL_BlitSurface(textimg, NULL, field, &where_to);
16.     textimg = TTF_RenderUTF8_Blended(font, "Press '3' to resume", text_img_col
    or);
17.     where_to.y = GAME_SPACE+5*BORDER;
18.     SDL_BlitSurface(textimg, NULL, field, &where_to);
```

First, we receive the screen and a *TTF_FONT* element. This element is needed to write different text in SDL. This font is initialized in the main. After that, the initializations are the Surface that will receive the text, the *SDL_Color* that the text will have and the rectangle where the text will be put. So, the *boxColor* function is called to draw a little box where the text will be placed, then *textimg* receive the *TTF_RenderUTF8_Blended* return. This function draws our text with the font we initialized in the main, what is going to be written and the color of it. Then, the position of

the *SDL_Rect* is defined, this element is important in the next function. The most important function of this section is the *SDL_BlitSurface* that copies the *textimg* surface in the main *field*, the screen of the user, in the *SDL_Rect position*. After the image is copied, the *field* has the text already. This process is repeated for all the bottom elements with small changes in its positions, colors and texts. Afterwards, the *textimg* surface is freed.

Following, there is a very similar function to the shown above, that is *score_Update*. This function is called whenever the snake eats an apple and it updates the score shown. It receives the score as an int, convert it to string and then using the same technique above, we copy the *textimg* in the user's screen.

The last function of this section is responsible to make everything work in the program.

```
1.  //this function will be called by mytimer
2.  //generates a user event and places it into the event queue (push)
3.  Uint32 mytimer(Uint32 ms, void *param) {
4.      SDL_Event ev;
5.      ev.type = SDL_USEREVENT;
6.      SDL_PushEvent(&ev);
7.      return ms;   /* wait this long till next call */
8.  }
```

This function generates events in each interval, so the program does not have to wait until the next user event is presented. It is crucial because the snake moves every event created by this timer. It is an SDL function and every time it generates and event, the game will move on. One of its parameters, *Uint32 ms*, is the interval between each event. This parameter is used to set the level of the game. The snake moves faster if the interval is lower and slower if the interval is big.

### Hall of Fame

The Hall of Fame is where the file handling is managed. The Hall of fame is initialized in the Main function. There are only 2 functions in this section, to show the Hall of fame and to insert a new score. To insert the file, the file in the main is received and opened with the *fopen* function. The program asks for the user to input a name and the score is recorded in the file. To show the hall of fame, it pauses the game in the switch case and only opens the file to print all the information on the screen.

```
1.  //Function to show hall of fame in prompt
2.  void show_Hall_of_Fame(FILE *file){
3.      //Creates a char data that will receive the string read
4.      char data[50];
5.      file = fopen("Hall_of_Fame", "r");
6.      //show if file could not be opened
7.      if(file==NULL)
8.          fprintf(stderr, "The Hall of Fame file could not be shown!\n");
9.      //Graphic division of what is the hall of fame
10.     printf("/-------------------------------------------------------------------
    -------/\n");
11.     printf("/-------------------HALL OF FAME---------------------------------
    -------/\n");
12.     printf("/-------------------------------------------------------------------
    -------/\n");
13.
14.     //While it reads anything, print the information
15.      while(fgets(data, 50, file) != NULL)
16.          printf("%s", data);
17.
18.     fclose(file);
```

```
}
//Function to insert the new score in the Hall of Fame
void insert_Hall_of_fame(FILE *file, int score){
    char name[10];

    file = fopen("Hall_of_Fame", "a");

    gets(name);
    printf("-----------\n");
    fprintf(file, "%s got score of: %d\n", name, score);

    fclose(file);
}
```

## Snake Section

This section contains all the functions that involves the snake. Firstly, we have the function that initializes the snake with the starting positions, speed and direction. This function is *Snake_elem *Initialize*. It returns a *Snake_elem* pointing to the head of the snake.

```
Snake_elem *initialize(){
    Snake_elem *newel = (Snake_elem *)malloc(sizeof(Snake_elem));
    newel->x = BORDER+10+( 11 * 20 );
    newel->y = 10+( 11 * 20 );
    newel->vx = SNAKE_VELOCITY;
    newel->vy = 0;
    newel->dir = RIGHT;
    newel->next = NULL;
    return newel;
}
```

Secondly there is **add_snake*. We call this function when the snake eats an apple, receiving the head of the snake as parameter, since it is a linked list, we need to allocate memory dynamically to store the new element. We traverse the list to the end, if there are more elements, and then we add the new element to the bottom of the list.

```
Snake_elem *add_snake(Snake_elem *head){
    //Dynamic memory is used to create a new segment of the snake
    Snake_elem *newel = (Snake_elem *)malloc(sizeof(Snake_elem));
    //Auxiliary variable p created to traverse the list
    Snake_elem *p = head;
    //goes to the end of the list to add the new segment
    while(p->next != NULL)p = p->next;
    p->next = newel;
    newel->next = NULL;
    return head;
}
```

The next function is responsible to change the snake's head direction. *Set_head_dir* receives the head of the snake and change its direction based on what was pressed. This function is called whenever an ARROW KEY is pressed.

Following, There are two *bool* functions that detects if the game was lost, that is, if the snake has touched the wall or itself, those are: *touch_wall* and *collision_with_snake*. In the first one, it is checked if the snake's head is inside the boundaries of the game, if it is, false is returned, if it is not, then true is returned. In the second function, an auxiliary *snake_elem *p* is created and set to the *->next* attribute of the snake's head, that is, the next element. To check if it has

touched itself, it must be checked if the head has touched any other element, that is the *->next*. In this case, we traverse the list with the auxiliary element p checking every list element. The first function:

```
1.  bool touch_wall(Snake_elem *cobra, SDL_TimerID id){
2.      //auxiliary variable to traverse the list
3.      Snake_elem *p = cobra;
4.      //Check if the snake's head touches the wall, if so, the game is lost and
    reseted to the beginning
5.      if(p->x < BORDER || p->x > GAME_SPACE+BORDER || p->y < 0 || p-
    >y > GAME_SPACE){
6.          SDL_RemoveTimer(id);
7.          SDL_Delay(1000);
8.          return true;
9.      }
10.     return false;
11. }
```

The second one:

```
1.  bool collision_with_snake(Snake_elem *cobra){
2.      Snake_elem *p = cobra;
3.      bool collision = false;
4.      //element p is always the next segment after the head
5.      //If the head's position is equal to any segment position, the game is los
    t
6.      p = cobra->next;
7.
8.      while(p != NULL){
9.          if(cobra->x == p->x && cobra->y == p->y){
10.             collision = true;
11.         }
12.         p = p->next;
13.     }
14.
15.     return collision;
16. }
```

Then, there is also a function used when the game is reset. The *Snake_elem *reset_snake* receives the screen of the user and the snake's head. Firstly, it erases the snake from the previous position and the dispose the list starting from the element after the snake's head. To dispose the list, another the function *dispose_list* is called. It receives the next element and calls another function *pop_front* that will free every element from the starting one given in the *dispose_list* function. The function to reset the snake:

```
1.  Snake_elem *reset_snake(SDL_Surface *field, Snake_elem *cobra){
2.      Snake_elem *p = cobra;
3.      /*clear the snake from previous position*/
4.      while(p != NULL){
5.              filledCircleColor(field, p ->x, p -
    >y, SNAKE_RADIUS, BACKGROUND_COLOR);
6.              p = p->next;
7.          }
8.      p = cobra;
9.      /*Delete only from the 2 segment, the head of the snake is preserved*/
10.     dispose_list(p->next);
11.     /*Set snake to starting position*/
12.     cobra = initialize();
13.     return cobra;
14. }
```

And the functions to dispose the snake's elements:

```
1.  Snake_elem *pop_front(Snake_elem *head){
2.      if (head != NULL) /* not empty */
3.      {
4.          Snake_elem *p = head;
5.          head = head->next;
6.          free(p);
7.      }
8.      return head;
9.  }
10. void dispose_list(Snake_elem *head){
11.     while (head != NULL)
12.         head = pop_front(head);
13. }
```

The last functions are the most important of the section, they are the ones the moves the snake. Moving the snake consists in: erasing the previous snake's position, calculating the new position based on the direction of the snake and its velocities. The *move_snake* function receives the surface used and the snake's head. An auxiliary element *p* is created, this element is set to the snake's head and it traverse the whole list (if there is more than one element) erasing its position. Then, another auxiliary element is created, the *tmp*. *Tmp* points to the snake's head, and p is set to the next position. To make the snake move, each element does not need to change its velocities or direction, it needs to be on the previous element position, that is, the snake's head position will be passed to the second element, and the third element's position will receive the second element's. In this case, it is needed to update the position of each element from the back of the list to the front. The last element's position will receive the previous and so on. To make this happen, the *Reverse* function is recursive. When *p* reaches the last position, the *tmp* element will point to the previous one, so it is copied until the head of the snake. The head of the snake moves accordingly to the keys pressed and to its velocities that are constantly changing.

```
1.  void Reverse(Snake_elem *p, Snake_elem *tmp){
2.      // Base case
3.      if (p == NULL)
4.         return;
5.
6.      // print the list after head node
7.      Reverse(p->next, tmp->next);
8.
9.      // After everything else is printed, print head
10.     p->x = tmp->x;
11.     p->y = tmp->y;
12. }
13. void move_snake(SDL_Surface *field, Snake_elem *cobra){
14.     //First, we erase all the snake without changing its position
15.     Snake_elem *p = cobra;
16.
17.     while(p != NULL){
18.         filledCircleColor(field, p ->x, p -
    >y, SNAKE_RADIUS, BACKGROUND_COLOR);
19.         p = p->next;
20.     }
21.     //Passes the position of the segment in the front to the segment in behind

22.     Snake_elem *tmp = cobra;
23.     p = cobra;
24.     p = p->next;
25.
```

```
    Reverse(p, tmp);

        /*calculates new position of the snake's head*/
    cobra ->x += cobra ->vx;
    cobra ->y += cobra ->vy;
        /*draw snake in new position*/

    p=cobra;
    while(p != NULL){
        filledCircleColor(field, p ->x, p ->y, SNAKE_RADIUS, SNAKE_COLOR);
        p = p->next;
    }
}
```

### Apple section

The only function in this section places apples on the field. But it cannot be anywhere because it can be placed in one snake element and since everything in the game is a drawing, the snake will "eat" the apple but the program will not detect it, because only the snake's head can eat the apples. In this case, one position of the field will be blank but there will be an apple there. To solve this problem, generate a random position of the apple and compare to each element of the snake. If it is different from every snake's element, then an apple is placed on the field.

```
void place_apple(SDL_Surface *field, Apple *maca, Snake_elem *cobra){
    Snake_elem *p = cobra;
    //Generate random position for apple and then place it on the board
    (*maca).x = rand() % 23;
    (*maca).y = rand() % 23;

    //Check if the position of the apple is equal to any segment of the snake

    //if yes, a new position for the apple is placed
    while( p != NULL){
        if((p->x)/20 == (*maca).x+1 && ((p->y)+10)/20 == (*maca).y+1){
        (*maca).x = rand() % 23;
        (*maca).y = rand() % 23;
        }
        p = p->next;
    }
    filledCircleColor(field,((*maca).x * 20)+20, ((*maca).y*20)+10, APPLE_RADI
US, APPLE_COLOR);
}
```

## Main Section

Firstly, we initialize some variables that will check the conditions of the game and some SDL objects. The *SDL_Screen* is the window opened where the game runs, *SDL_Event* and *SDL_TimerID* are objects responsible to create the time in the game, the *SDL_Event* is the event created by the timer function explained before. Then, we initialize the snake's head dynamically and set the starting specifications. The level variable contains the level of the game. *Apple_check* is a Boolean responsible to verifying if there is a apple in the field or not, when it is false there isn't and true because there is. *tWall and tSnake* are Booleans that verify if the snake touched the wall or itself. The *pause* Boolean pauses the game. There is also the score integer that stores

the score and the initialization of the Hall of Fame file. And last but not least, the Boolean *quit* that is responsible for closing the game.

```
1.  //Creating SDL surface
2.      SDL_Surface *screen;
3.
4.      //Initialize the snake
5.      Snake_elem *cobra_head = NULL;
6.      Snake_elem *newel = (Snake_elem *)malloc(sizeof(Snake_elem));
7.      newel->x = BORDER+10+( 11 * 20 );
8.      newel->y = 10+( 11 * 20 );
9.      newel->vx = SNAKE_VELOCITY;
10.     newel->vy = 0;
11.     newel->dir = RIGHT;
12.     newel->next = NULL;
13.     cobra_head = newel;
14.
15.         /*Level*/
16.     int level;
17.     level = LEVEL;
18.
19.         /*Apple*/
20.     Apple maca;
21.     bool apple_check;
22.     apple_check = false;
23.     //
24.     bool game = true;
25.     //Variable that checks if the snake touched the wall
26.     bool tWall;
27.     //Variable that check if the snake touched itself
28.     bool tSnake = false;
29.     //Variable to keep the game loop
30.     bool quit = false;
31.     //Variable to pause the game
32.     bool pause = false;
33.     //Score
34.     int score = 0;
35.     //Hall of Fame
36.     FILE *hof;
37.
38.     //Creating SDL event and Timer to make the snake move
39.     SDL_Event event;
40.     SDL_TimerID id;
```

After that, SDL functions are used to the variables initialized. *TTF_Font* is initialized and receives the “PatuaOne_regular.ttf” font and the size of it. To initialize this function, the file ttf must be in the project’s file. Following, screen receives the screen in the given format based on the macro definitions made in the beginning of the program.

```
1.  TTF_Font *font;
2.  TTF_Init();
3.  font = TTF_OpenFont("PatuaOne-Regular.ttf", 19);
4.  if (!font) {
5.      fprintf(stderr, "The font could not be opened! %s\n", TTF_GetError());
6.      exit(1);
7.  }
8.
9.  //Initializing SDL and open window
10. SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER);
11. screen = SDL_SetVideoMode(WINDOW_WIDTH, WINDOW_HEIGTH, 32, SDL_ANYFORMAT);
12. if(!screen){
13.     fprintf(stderr, "The window could not be opened!\n");
14.     exit(1);
```

```
15. }
16. SDL_WM_SetCaption("Snake2711", "NULL");
```

Then, the functions in the game section are called to draw the field and the bottom section of the game.

The while loop will run until the *quit* is not changed, the *SDL_WaitEvent* function will receive the events generated by the user or by the timer created. The switch afterwards is to decide the program response base on what event is has receive. If no user event is entered, then the *SDL_USEREVENT* case is active. Firstly, it checks if there is an apple or not in the field according to the *apple_check* variable and if needed the *place_apple* function is called. Secondly, the *move_snake* function is called and the snake moves in the game. *tWall* and *tSnake* receive their respectives functions, *touch_wall* and *collision_with_snake*. If any of them is true, the Hall of Fame is changed, the snake resets and the timer is removed. The timer is removed many times to pause the game and the program will only wait for the keyboard instructions. Then, it checks if the snake ate an apple or not in the new position. The last function is a SDL function to update the screen with the new drawing made.

```
1.  if(apple_check==false){
2.      place_apple(screen, &maca, cobra_head);
3.      apple_check = true;
4.  }
5.  //Verify if the next move is possible
6.  //It it isn't, the timer is removed and the game is paused
7.  //Calls the function to draw snake
8.  move_snake(screen, cobra_head);
9.  //Check if snake touched the wall
10. tWall = touch_wall(cobra_head, id);
11. //Check if snake touched itself
12. tSnake = collision_with_snake(cobra_head);
13. if(tWall == true || tSnake == true){
14.     insert_Hall_of_fame(hof, score);
15.     cobra_head = reset_snake(screen, cobra_head);
16.     game = false;
17.     SDL_RemoveTimer(id);
18. }
19. //Now with the new position, it is checked if it has eaten an apple or not
20. if( (cobra_head->x)/20 == maca.x+1 && ((cobra_head-
    >y)+10)/20 == (maca.y)+1){
21.     cobra_head = add_snake(cobra_head);
22.     score = score + 1;
23.     score_Update(screen,font,score);
24.     apple_check = false;
25. }
26. SDL_Flip(screen);break;
```

Following the next case of event, there are the events generated by the player using the keyboard. The ESCAPE key changes the Boolean *quit* and exits the while loop closing the game. The NUMBER 1 key will call the Hall of Fame functions and pause the game, when pressed again the game is resumed. The NUMBER 2 key will reset the game and decrease the level of one *LEVEL_UP* unit. This is done by removing the timer existing and creating a new one with the new level value. The NUMBER 3 case, pauses the game and set the *pause* Boolean, it removes the timer and set true to pause, and creates the timer and set pause to resume the game. The SPACE case will reset the snake and the score calling functions explained before. The arrow keys call the function to change the snake's head direction with the direction of the key pressed and defined in the macro definitions section.

```
case SDL_KEYDOWN:
    switch(event.key.keysym.sym){
    //When ESC button is pressed the program exits
    case SDLK_ESCAPE:
        quit = true;
        SDL_RemoveTimer(id);break;
    //Hall of Fame button
    case SDLK_1:
        if(pause == false){
            SDL_RemoveTimer(id);
            show_Hall_of_Fame(hof);
            pause = true;
        }else{
            id = SDL_AddTimer(level, mytimer, NULL);
            pause = false;
        }break;
    //level button
    //Only 4 levels based on the timer
    //When pressed, the game resets with the new level
    case SDLK_2:
        score = 0;
        score_Update(screen, font, score);
        if(level >= 50){
            level = level - LEVEL_up;
            cobra_head = reset_snake(screen, cobra_head);
            SDL_Flip(screen);
            SDL_RemoveTimer(id);
            game = true;
        } else {
            level = LEVEL;
        }break;
    //Pause button
    //When pressed again, the game resumes
    case SDLK_3:
        if(pause == false){
            SDL_RemoveTimer(id);
            pause = true;
        }else{
            id = SDL_AddTimer(level, mytimer, NULL);
            pause = false;
        }break;
    //Restart button
    case SDLK_SPACE:
        if(game == false){
            cobra_head = reset_snake(screen, cobra_head);
            score = 0;
            score_Update(screen, font, score);
            game = true;break;
        }
        id = SDL_AddTimer(level, mytimer, NULL);
        game = false;break;
    /*------------------------------*/
    /*-----------MOVE KEYS----------*/
    /*------------------------------*/
    /* w key for going up */
    case SDLK_UP:
        set_head_dir(cobra_head, UP);break;
    /* a key for going left*/
    case SDLK_LEFT:
        set_head_dir(cobra_head, LEFT);break;
    /* s key for going down*/
    case SDLK_DOWN:
        set_head_dir(cobra_head, DOWN);break;
    /* d key for going right*/
    case SDLK_RIGHT:
        set_head_dir(cobra_head, RIGHT);break;
```

```
67.     }
68.     break;
```

After the while loop, the timer created is removed, the font created is closed, SDL is also closed and then the program ends