Allegro Tutorials

Introduction

First, I must tell you that a general knowledge of C/C++ is required, and that Allegro must be correctly installed (we'll look at this point in another tutorial.)

- Throughout all my tutorials, function prototypes belonging to allegro will be written **in bold characters**.
- **/*Comments will be in a characteristic blue-green*/**
- Types are in maroon(int, float, …), as well as types specific toAllegro (BITMAP, DATAFILE, ...).
- Control structures in blue (if, else, while, {, }, …)
- Numbers in red (0 , 1, 324, 4, …)
- "include" and "define" in green.
- "Character strings in gray"

1 . What is Allegro?

Let's get directly to the subject: "What is Allegro?" It's a library that furnishes everything you need to program a video game. Allegro gives you a solution for managing the screen, sound, keyboard, mouse, timers, in short, everything you need. Allegro was originally written by Shawn Hargreaves for the Atari ST, and soon after ported to DOS. The first versions of the library are dated at the start of 1996, so the library isn't all that young! Programmers (contributors) to Allegro quickly developed it into a multi-platform library. Today you can use Allegro under DOS (DJGPP, Wacom), Windows (MSVC, Wingw32, Cygwin, Borland), Linux (console) Unix (X), BeOS, QNX, and MacOS (MPW) . You can see that the great power of Allegro comes from the fact that it is supported by a large number of operating systems. Specifically, you can port and compile your programmes under any type of compiler (mentioned above) without changing a single line of code. In each case, Allegro will select only the drivers appropriate for that OS. For example, a program compiled under Windows uses DirectDraw accelaration, while under Linux you can use the X11 drivers: again a small bonus, you can take advantage of the graphic acceleration from your video card in 2D, and that s not a negligible difference. This is the same for DirectSound and DirectInput in Windows. On the other hand, 3D isn't the strong point of this library: it won't take advantage of the acceleration provided by Direct3D. Be assured, however, that OpenGL is very well supported, thanks to AllegroGL, an Allegro add-on. We also note that this library is free (gratis) and free (as in free speech), since the source code is available!

2.Allegro : A Sample Application

A game that uses Allegro can be a Windows application. In this case, DirectX 7.0 (minimum) should be installed for the game to work. Allegro can be used as a DLL, `alleg40.dll`, or it can be statically linked: the code Allegro uses will be copied into the game's executable file (this is always the case with DOS). I can assure you, for other OS, you can choose static or dynamic compilation for your programs as you wish.
As an example, you have at your disposal the official demo (in the `allegro/demo` directory) that combines many interesting characteristics of the library into a single program.

Don't forget to link the library after compiling the program; otherwise you'll get an error (see the help file for your compiler!)

Don't forget that you have at your disposal a panoply of very practical examples. It's thanks to these that I have learned to use allegro "properly". All the descriptions of Allegro functions are available in the general help file after you install Allegro. They're very well explained and will let you discover all of Allegro's functions (`/docs/html/allegro.html`)

Finally, if you truly want to discover the amazing potential of this library, you can go to the official repository for games that use Allegro: www.allegro.cc

3. Basic Allegro: a first program

Let's begin with a small, simple program that will serve as an example. First of all, you must include the library header, whose name is "**allegro.h**". Note, right now, that there's no mention of **WinMain** or <windows.h>; you have to forget all those things that pertain to only a single OS.

```
/* This includes the library header */
#include <allegro.h>

/* And now our main function begins!! */
int main()   {
```

It's very important to call the initialization function before doing anything else with the library:

```
/* General initialization function */
allegro_init();
```

Perfect! Allegro is initialized! And now, if you want to use the keyboard and at would be very practical...

```
/* Initialize the keyboard */
install_keyboard();
```

If the function is able to initialize the keyboard, it returns 0, otherwise it returns a negative number. You can consider that it's not worth the trouble to verify this result, since the chances of it failing are minimal...

```
/* Initialize the mouse */
install_mouse();
```

Now things get more interesting: if the function has failed, it returns -1; if it succeeds, it returns the number of buttons on your mouse that Allegro can manage. It is thus important to test this result because not all DOS users will have a working mouse installed. You can thus write:

```
/* If the function fails, then… */
if (install_mouse() == -1) {
    /* Display an error message */
    allegro_message("Error! %s", allegro_error);

    /* And exit the program! */
    return -1;
}
/* Otherwise, you are sure you have a mouse! */
```

But, you ask, "What is this **allegro_message**? And **allegro_error**?"
Here's the prototype for **allegro_message**:

void allegro_message(char *msg, ...);

This function uses the same format as the **printf** function. It is thus very convenient to use. Note: This function should be used only in non-graphic display modes! Clearly, you can't use it unless you haven't yet set a video mode or if you've explicitly changed to text mode. For example, here, you haven't yet initialized the video mode, so you can use the function without any problem.In OS where you have a text mode in a "console", like DOS or Unix, the message will dispaly normally in the console. For Windows (or for BeOS), this will produce a dialogue box appropriate for that OS with an "OK" button at the bottom. The dialog's title will be the name of the program that's running. This functionis therefore very practical for signaling errors independent of the OS.

extern char allegro_error[ALLEGRO_ERROR_SIZE];

This is a character string used by various Allegro functions, such as **set_gfx_mode** our **install_mouse**. It serves to report errors that can occur during initialization. If the user wants to know the nature of the error, the string **allegro_error** contains the descriptoin of the problem: there is nothing more to display (with **allegro_message** for example). **ALLEGRO_ERROR_SIZE** is the length of the character string.
Very good; we now have the mouse and keyboard. It's easy, yes? Only, it would be

better to go into graphic mode... First of all, we have to define the color video mode; that is, to say if each pixel should be encoded with 8, 15, 16, 24, or 32 bits. The higher the number of bits, the greater the available color palette. For example, with 16 bits you have 2^{16} = 65536 colors; with 24 bits 2^{24} = 16777216 colors. Eight bit color is a special case; let's put it aside for the moment.

```
/*  Initialize the color graphic mode. 16 bits is a good value;
it's enough to display our blank screen. It has the advatange of
being very common, and thus easily supported by video cards */
set_color_depth(16);
```

Now, set the graphics mode itself. To do this, you use the command:

```
set_gfx_mode(GFX_AUTODETECT, 640, 480, 0, 0);
```

But wait, how on earth do you use this function? Here's its prototype:

int set_gfx_mode(int card, int width, int height, int v_width, int v_height);

First, the easy part. The function returns **0** if it succeeds, otherwise it returns a negative number.

"int card" isn't what it seems like; it's the index of the graphic mode that you want to use. Here, then, are the different values that you can set: (the #define is part of **allegro.h**):

- **-GFX_AUTODETECT** . This lets Allegro choose the best driver. It will set window mode if the resolution isn't available to full screen and if the OS supports it, of course.
- **-GFX_AUTODETECT_FULLSCREEN** . Forces Allegro to choose a full-screen mode.
- **-GFX_AUTODETECT_WINDOWED** . Forces Allegro to choose a window mode.
- **-GFX_TEXT** . This is very useful for returning to text mode. In this case, you can use a 0 for the screen dimensions (this is just for readability).

There are naturally other values, but they are more specific to each OS (and thus you should avoid using them). We'll lok at this later, but for now let's stick to the basics. The values **width** and **height** represent the size of the graphic screen created. You can find the dimension of the screen by using the macros **SCREEN_W** (width) and **SCREEN_H** (height), which this function initializes. At the moment, don't worry about **v_width** and **v_height** (you can use zero for both these values). As every respectable program does,
it must check to see that there was no error! Here are the necessary tests:

```
if (set_gfx_mode(GFX_AUTODETECT, 640, 480, 0, 0) != 0) {
    /* If you have followed my explanations well, you know
    that allegro_message is used only in text mode.
    For this reason, we use GFX_TEXT, to ensure that we
    return to text mode… */
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);    /* here's the command for
text mode */
                allegro_message("Unable to initialize video mode!\n%s\n",
                        allegro_error);
                return 1; // don't forget to exit…
        }

        // Here, all is well with SCREEN_W = 640 and SCREEN_H = 480
```

Voilà! We've finished initializing our little program! We could add sound and joystick control, but that's the subject of later chapters. Now, let's make the program stop when you press the ESC key. Thus, add the following few lines of code...

```
        while (!key[KEY_ESC]) { /* while the [ESC] key hasn't been pressed... */
                /* Erase the screen. We'll discuss this function in
                the next section that's dedciated to display*/
                clear_bitmap(screen);
        }
```

Here's an important aspect of the library: handling keypresses from the user.

```
      extern volatile char key[KEY_MAX];
```

It's very simple:all the keys in Allegro are grouped in this table (of size **KEY_MAX**). It is a table where each key has its own index number. If you want to see the entire list of keys, open the file allegro/keyboard.h; everything is defined there. Normally this table represents the physical state of the actual keyboard. That is to say, whether it is pressed or not. Thus, this table is meant to be read-only. It is "modified" only by keyboard interrupts. You can certainly simulate a keypress, but that's another story... Here are other examples of use (don't put these into your program; it won't work there! That's because **printf** doesn't work in graphic mode):

```
        if (key[KEY_ENTER]) {
                printf("The ENTER key was pressed!\n");
        }
        if (!key[KEY_SPACE]) {
                printf("The space bar was  *NOT* pressed!\n");
        }
```

Don't forget to exit your program:

```
        return 0; // and this exits the program
        }
        END_OF_MAIN();
```

You will have noticed that Windows applications have their entry point as **WinMain** and not **main**. Because of this the macro **END_OF_MAIN()** is required: it let you use the **main** function no matter what your system. Place this directly after the end of your **main** function. Don't worry; you won't get any warning or error messages when you compile it!

That's it for the basic initialization of Allegro. If you've copied and understood this example completely, it should have no problems. Of course, it's very basic; it doesn't do anything except display a black screen (if there was no initialization error) and wait for you to press [ESC] to quit. Now let's discuss an essential part of any video game: display on the screen!

4. Display

4.1 General Presentation

Now let's talk about display, without a doubt the most important part of the library. To do this, we will display a white disc with a diameter of 50 pixels which will be move towards the right side of the screen. It will be exactly at the vertical center of the screen. The user will have to touch [ENTER] to display the disc. We'll modify the loop to that waits to see if the [ESC] key has been pressed:

```
/* declare our variable that represents the position of the circle.
If you're programming in C, place it at the beginning of the
main function. */
double circle_pos_x = 0;


/* Wait patiently for the user to press ENTER to begin
to display and move the circle. */
while (!key[KEY_ENTER])
     /* do nothing */ ;

/* while ESC has not been pressed and we are still on screen */
while (!key[KEY_ESC] && circle_pos_x - 50 < SCREEN_W) {

     /* erase the screen */
     clear_bitmap(screen);

     /* Display the "filled" circle in white (color 255,255,255) */
     circlefill(screen, (int)circle_pos_x, SCREEN_H/2,
          50, makecol(255,255,255));
     circle_pos_x += 0.1; // Move the disc to the right 0,1 pixel
}
```

Don't worry; we'll explain all of this. The **circlefill** function serves to display the disc on the screen.

```
void circlefill(BITMAP *bmp, int x, int y, int radius, int color);
```

x and **y** represent the coordinates of the center of the circle on the screen. All the coordinates are given with respect to the upper left corner of the screen. **radius** represents the radius in pixels. As you may have noticed, there's a new data type here, the **BITMAP**. We'll discuss that soon.

```
extern BITMAP *screen;
```

Just remember that **screen** is a global variable that points to a **BITMAP** (an image area) that represents video memory. It points directly to the screen and the size of that bitmap is thus **SCREEN_W * SCREEN_H**.
bmp is thus the bitmap upon which we will draw our disc. It's the destination of the drawing function.**color** is the color, as its name indicates. The best way to specify a color is to use a utility function:

```
int makecol(int r, int g, int b);
```

This function lets you define a color independent of the color mode. You can call this function in 8, 15, 16, 24, and 32 bit per pixel mode. **r** represents the red component, **g** the green component, and **b** the blue component. This is also called an RGB color format. These components can have a value from 0 to 255 inclusive. makecol(255,255,255) specifies the color white in the current color mode.

Now, if you start the program, you'll see a horrible blinking of the screen. You won't see a disc, but rather a succession of black and white lines. This is absolutely not the deisred effect! Well then, where's the problem? It's very simple. To program a game correctly, one must *never* write directly to the screen unless [se fiche totalment] the result, but that's not the case here. Instead, we must use a new method of display, called **double buffering**. The principle is very simple: instead of drawing to the screen, one draws to a bitmap that has been [prealablement] placed in RAM. Then, at the end of our [boucle], we copy the contents of the bitmap (also called a *buffer*) to the screen in one fell swoop. This is not the method that gives the most spectacular results, but it will always give a much cleaner [ce sera deja nettement mieux] display.

This is a portion of the definition of a **BITMAP** (you can see the whole definition in the library headers):

```
typedef struct BITMAP {
    int w, h;       /* la taille du bitmap en pixel (w largeur, h hauteur) */
};
```

There are other variables in the BITMAP type, but you won't need them. Thus, to find the size of the screen, you can look at `screen->w`, which represents the width of the screen and `screen->h`, which represents the height. Don't forget that Allegro works only with pointers to BITMAP. (BITMAP *) We declare our video buffer like this:

```
BITMAP* buffer; /* This is the variable that points to the buffer! */
```

Now we have to create a bitmap that will have the same dimensions as the screen. You must do this **after** you initialize the video mode.

```
buffer = create_bitmap(SCREEN_W, SCREEN_H);
```

BITMAP* create_bitmap(int width, int height);

This function creates a bitmap of the given **width** and **height**. It returns a pointer to the bitmap that was created. Normally this image is crated in RAM and isn't completely empty (black) You have to clear it up from residual bits before using it. To clear a bitmap to black, use this function:

void clear_bitmap(BITMAP *bitmap);

This function can be accelerated, and when it is, the result will be an incredible gain in performance.

Here, more specifically:

```
clear_bitmap(buffer); /* Very easy… */
```

Now let us rewrite the wait loop:

```
/* while ESC has not been pressed and we are still on screen */
while (!key[KEY_ESC] && circle_pos_x - 50 < SCREEN_W) {
    /* Begin by clearing the buffer */
    clear_bitmap(buffer);

    /* Display the "filled" circle in white (color 255,255,255) */
    circlefill(buffer, (int)circle_pos_x, SCREEN_H/2,
         50, makecol(255,255,255));
    circle_pos_x += 0.1; //  move the circle to the right 0.1 pixel

    /* Now copy our buffer's content to the screen */
    blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);
}
```

We have replaced **screen** with **buffer** in the appropriate functions. It remains only to find out what the function **blit** does… This function lets you copy a part of a source bitmap to a part of a destination bitmap.

void blit(BITMAP *source, BITMAP *dest, int source_x,
int source_y, int dest_x, int dest_y, int width, int height);

this is a function that begins to take some interesting parameters! **source** is clearly the source bitmap (in this case **buffer**), **dest** is the destination bitmap (**screen** in this case). **source_x** and **source_y** represent the coordinates of the origin in the bitmap to be copied. **dest_x** and **dest_y** represent the starting coordinates to which the bitmap should be copied in the destination. Finally, **width** and **height** represent the dimensions of the area that you want to copy.

In this example, we want to cpie the entire buffer to the screen. For this reason, we begin to draw at coordinates (0, 0) and draw the entire buffer (SCREEN_W and SCREEN_H). Run the new program; you'll be surprised! No more rebellious flashing! Things look good... On the other hand, the speed of movement of the disc is dangerously close to being too slow! This is to be expected, because the computer irequires a lot of time to move all those bits an ddraw them. Drawing a small disc on the screen is very much quicker than copying an entire buffer to the screen, so ilet's change the line for the disc's displacement by:

```
++circle_pos_x;  /* Move the disk 1 pixel at a time */
```

The disk's speed depends on the speed of your computer (meaning that display is a very limiting factor in the speed of a program). Even with a super-powerful computer, you can't attain astronomical speeds (in terms of images per second) bedcause you have to wait for the video card to finish drawing each image! In the next sections, we'll discuss *real* time, that is to say, that your game runs at the same speed, no matter what the power of the computer on which it runs.

4.2 Displaying Text

One of the most "brilliant" features of Allegro is its display of text to the screen in a very simple fashion! Ineed, in graphic mode, there's absolutely no question about using **printf** or other functions of that type. Allero furnishes us with a panoply of functions that let you do these task automatically. Let's once more modify the main loop of our small program to display the screen's resolution and the position of our small white disc:

```
/* while ESC has not been pressed and we are still on screen */
while (!key[KEY_ESC] && circle_pos_x - 50 < SCREEN_W) {

    /* Start by clearing the buffer */
    clear_bitmap(buffer);

    /* Display a character string at coordinates 0,0 using a
    blue-violet color (150, 150, 255) */
    textout(buffer, font, "I'm writing text!", 0, 0, makecol(150, 150, 255));

    /* Display a white circle (color 255, 255, 255) in the buffer */
    circlefill(buffer, (int)circle_pos_x, SCREEN_H/2,
          50, makecol(255,255,255));
```

```
        circle_pos_x += 0.1; /* Move the circle right 0.1 pixel */

        /* Copy the buffer contents to the screen */
        blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);
    }
```

You use the **textout** function to display text:

```
void textout(BITMAP *bmp, const FONT *f, const char *str,
        int x, int y, int color);
```

bmp represents the bitmap on which the text will be displayed. **str** is the character string to display, **x** and **y** are the coordinates of the upper left point where the text will be displayed (0 and 0 assure us that the text will be at the corner of the screen). We need only introduce the idea of a **FONT**. As with Windows, Allegro can display many kinds of typefaces. You just have to specify a pointer to the desired font, which must be of type **FONT**. To load fonts, you must reference a database. This will be described in later chapters. Luckily, you can use the default BIOS font, expressed thus:

```
extern FONT *font;
```

By the way, you can modify this pointer to point to any other font, but tat’s not very useful at the moment, because we don’t know how to load fonts yet.

Now you can display the size of the screen on the screen. Just add this:

```
#include <string.h> /* include the standard ANSI C header. */
```

You’ll need a new variable:

```
/* Here’s a small buffer that can contain 256 characters */
char str_buf[256];
```

And then we change the call to **textout:**

```
/* First copy the screen size into str_buf */
sprintf(str_buf, "Here is the screen size: %d*%d",
    SCREEN_W, SCREEN_H);
/* Replace our new string… */
textout(buffer, font, str_buf, 0, 0, makecol(150, 150, 255));
```

Admire the result! You will see at the upper left of the screen: “Here is the screen size: 640*480”. But the creators of Allegro thought of everything. To save one line, they let you use textout with the syntax of printf. You can rewrite that portion of the code as:

```
 /* Change the color for this occasion… */
textprintf(buffer, font, 0, 0, makecol(100, 255, 100),
         "Screen size: %d*%d", SCREEN_W, SCREEN_H);
```

```
void textprintf(BITMAP *bmp, const FONT *f, int x, y, color,
        const char *fmt, ...);
```

Notice the prototype. You can now display formatted text, just as you do with printf. The function is similar to textout, so we don't have to review those details here. One final note: using textprintf allows you to take out the line "#include <string.h>", don't forget to also take out the declaration of the character buffer, which is no longer used.

Why not display the circle's position on the screen? And make the color change depending upon the position...

```
/* Here's the new command, a bit longer to type... */
textprintf(buffer, font, 0, 10,
     makecol(circle_pos_x / SCREEN_W * 255,
     circle_pos_x / SCREEN_W * 255,
     circle_pos_x / SCREEN_W * 255),
     "Disc x: %d", (int)circle_pos_x);
```

Thus, the character string will display with increasing brightness, which creates an interesting visual effect. In the last fifty pixels, the string "disappears." In fact, this is because the variable circle_pos_x has taken a value greater than 255. The program cuts this off to zero, which is a very dark color. However, to continue, there are other functions which are similar to **textprintf**.

```
void textprintf_centre(BITMAP *bmp, const FONT *f, int x, y, color,
        const char *fmt, ...);
```

This function does exactly the same thing as **textprintf**, except that it interprets **x** and **y** as the center of the character string rather than the upper left corner.

```
void textprintf_right(BITMAP *bmp, const FONT *f, int x, y, color,
        const char *fmt, ...);
```

Same thing, except that x and y are the coordinates of the upper right corner.

```
void textprintf_justify(BITMAP *bmp, const FONT *f, int x1, int x2,
        int y,int diff, int color, const char *fmt, ...);
```

Here, the text is justified between the **x1** and **x2** coordinates. If the letters overlap, the function defaults to the standard **textprintf** function.

There. Now you know all the Allegro functions based on **textprintf**. There are also some utility functions that can be useful in certain cases. For example:Il existe quelques fonctions annexes qui peuvent être bien pratiques dans certains cas. Par exemple:

int text_length(const FONT *f, const char *str);

This function returns the lengnth in pixels of the character string **str** using the font **f**. You can use this to find out if a string will fit on the screen or not.

int text_height(const FONT *f)

You don't need to specify a string here, because you're only interested in the font height. You need only give the font **f** to retrieve the height of the font in pixels.

We will now use the **textprintf_centre** function in our little example program, replacing the line that controls the display of the screen dimensions by this:

```
  /* Display the dimensions at the exact center of the screen */
textprintf_centre(buffer, font, SCREEN_W / 2, SCREEN_H / 2,
  makecol(100, 255, 100), "Screen size: %d*%d",
  SCREEN_W, SCREEN_H);
```

Run the program... If you've followed our instructions correctly, the circle should display above the text. This is logical because you draw the circle **after** drawing the text, so the disk is superimposed. Now, move the line that draws the text so that it follows the line that draws the circle.

Re-run the program. You will see this time that the text lies "above" the circle, but it seems to be written on some strange black box! Let me reassure you right away that this is perfectly normal. Let's modify this right away. To do this, here are the new lines to write:

```
/* Set the mode for text display to transparent */
text_mode(-1);

/* Display the dimensions at the exact center of the screen */
textprintf_centre(buffer, font, SCREEN_W / 2, SCREEN_H / 2,
   makecol(100, 255, 100), "Screen size: %d*%d",
   SCREEN_W, SCREEN_H);
```

The result is truly perfect: no more black framework! But how can this be? All displayed text is composed of two parts: the foreground, which is the text itself, and the background, which represents a box enclosing the displayed character string. By default, this box is black, and it is for this reason that you didn't see it (have I managed to [magouillé mon coup pas vrai?]), but on the contrary, you can see the text clearly if it's drawn on a color other than the color of the "box." Let's look at the **text_mode** function:

```
int text_mode(int mode);
```

This function is very easy to use. **mode** is the clor of the box on which the text is displayed. You can create it with **makecol**, as you have seen. For example, to display on a white box:

```
text_mode(makecol( 255, 255, 255));
```

There's one small exception. If you don't want to draw the "box," that is, you want to have the effect of transparency, you must use the color **-1**. Important: once you use **text_mode** to set a mode, that modeis in effect for all subsequent text display! Thus, you must redefine the background color each time you need to change it. For example, in our program, from the second occurrence of the loop onwards, all text will be drawn with a transparent background!

You now have all the tools in hand to display text on the screen from your programs. Do not hestitate to try all the functions; nothing beats practice and personal experience to thoroughly learn how to use them.

We've drawn a circle...wouldn't it be better to display a true **image** (i.e., a **sprite**)?

4.3 Displayingsprites

Simply put, it is quite possible to load an image and display it anywhere on the screen. This is the very basis of a 2D game. A **sprite** is nothing more than a **bitmap**, like the **buffer** that serves as the display, except that this time, you must initialize its content. Normally, one should not modify a **sprite** directly after having loaded it into memory, and it shouldn't be drawn on the screen without a specific display mode (like transparency, or using a solid color background). Select a nice image of medium size (for example 320 by 200), or resize it so that it will fit on the screen! The image doesn't **have** to be saved as eight bits per pixel; we'll talk about that later. Now we will once again modify our program. We'll replace the circle with a sprite. First, one must load the image and rename the variable **circle_pos_x** (for greater readability). We presume that your **sprite** is in a file namd "sprite.bmp" and that is in the same directory as your executable file.

```
/* Declare the variable that represents the position of the image.
If you are programming in C, plae this at the beginning of the main  function.
*/
double sprite_pos_x = 0;

/* Now declare the sprite as a pointer to BITMAP. Note: This is really a
pointer! Don't forget the asterisk. */
BITMAP* sprite1;

/* Put these lines after the initialization of the video mode! */
sprite1 = load_bitmap("sprite.bmp", NULL);

/* Now, one should verify that the bitmap load worked! */
if (!sprite1) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("Error! Unable to load the image file!");
    return1;
}
```

Why do you place the image loading code after the video mode initialization? So that the image can be loaded in the correct (initalized) video mode! If you start the program now, nothing will have changed. If your image didn't load correctly, check the spelling of the file name string, and make sure that the file is really in the correct directory. There's nothing remarkable about the **load_bitmap** function:

BITMAP* load_bitmap(const char *filename, RGB *pal);

This is truly an important function! First, it returns NULL in case of an error; if not, it returns a pointer to a **BITMAP** which has been created from the given **filename** string.As you already know, Allegro works in an eight bit per pixel color mode, which is special because it requires the use of a color palette. Let's consider the **pal** parameter for a moment. [Sachez] in order to load a **truecolor** image (that is, 15, 16, 24, or 32 bits per pixel), pass NULL as a value for **pal**. The pointer returned by this function points to a block of memory that has been allocated for the image. Don't forget to free this space before leaving the program. (We'll see how in a moment.) The **load_bitmap** function supports BMP, LBM, PCX, and TGA files. These different types are determined by the filename extension. Sprites have one great peculiarity: they posses a **mask color**. When you deisplay a sprite, it is always displayed in a perfectly rectangular area. Let's imagine for a moment that you want to display a red marble above an image representing the surface of a lawn. Of course, your sprite for the marble will contain a red marble centered in the sprite. But around it? As the sprite is a rectangle, it will be filled with, for example, black. You start the game and OMG! It seems as if the marble isn't displayed correctly on the lawn. It's displayed in a rectangle (or square) that's all black! You have to get rid of the "box" surrounding the marble. This is exactly the same problem we had with the text display (you remember it, I hope?) This is where our mask color comes to our aid. In **truecolor** mode, all the pixels containing the color (255, 0, 255) will be totally ignored! Replace the black surrounding the marble with this color (bright purple) and you'll get the desired result. It's as simple as that.

Now we will once more modify the infinite loop controlled by the ESC key. Here's the new version:

```
/* Set the mode for text display to transparent */
text_mode(-1);

while(!key[KEY_ESC]) { /* While the [ESC] key hasn't been pressed... */

    /* Start by clearing the buffer */
    clear_bitmap(buffer);

    /* Draw the sprite at the vertical center of the screen */
    draw_sprite(buffer, sprite1, (int)sprite_pos_x – sprite1->w/2,
        SCREEN_H/2 - sprite1->h/2 );

    /* Display the dimensions at the exact center of the screen */
    textprintf_centre(buffer, font, SCREEN_W / 2, SCREEN_H / 2,
        makecol(100, 255, 100), "Screen size: %d*%d", SCREEN_W,
        SCREEN_H);

    /* Display the sprite coordinates */
    textprintf(buffer, font, 0, 10,
        makecol(sprite_pos_x / SCREEN_W * 255,
        sprite_pos_x / SCREEN_W * 255,
        sprite_pos_x / SCREEN_W * 255),
        "Sprite_x: %d", (int)sprite_pos_x);
    /* Move the sprite to the right 1 pixel */
    ++sprite_pos_x ;

    /* Check if the sprite is off the screen; if so, exit the program */
    if (sprite_pos_x - sprite1->w / 2 >= SCREEN_W) {
        break;
    }

    /* Now copy our buffer's content to the screen */
    blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);
}
```

Run our new program... If the program is slow, adjust the value that serves to move the sprite to the right. There's nothing really new there; it's just the time to make a point about the program [en cours]. I have placed the **text_mode** function before the loop because the program works entirely in transparent mode. Thus,it's inefficient to call the function every time you go through the loop. Always remmeber that what you draw into the buffer first is in the "back," and the order of objects is determined by the order in which the program draws them. Given that our sprite is of type **BITMAP**, you can easily dtermine its size, which is very practical when you want to display the sprite centered:

```
        /* sprite width */
        sprite1->w;
        /* sprite height */
        sprite1->h;
```

If you don't understand yet why the sprite is centered, draw a diagram and label all the values you know (screen size, sprite size) and it will become much clearer. Let's take a closer look at the function that displays the famous sprite!

```
    void draw_sprite(BITMAP* bmp, BITMAP* sprite, int x, int y);
```

It's quite basic. **bmp** represents the destination (the image where the sprite will be drawn). **sprite** is the pointer to the sprite (the name is quite appropriate). **x** and **y** are simply the coordinates of the upper left corner where the sprite will be drawn. This function will ignore all pixels that have the mask color. This function is similar to "blit(sprite, bmp, 0, 0, x, y, sprite->w, sprite->h)", except that **blit** copies all the pixels. These drawing functions can be accelerated greatly if the video driver allows it. The **draw_sprite** is the basis for all other sprite display functions, as we will see.

There's another way to draw a sprite. For example, do you want to draw the sprite with a vertical flip (a vertical mirror effect) or a horizontal flip, or maybe both at the same time?

```
    void draw_sprite_v_flip(BITMAP *bmp, BITMAP *sprite, int x, int y);
    void draw_sprite_h_flip(BITMAP *bmp, BITMAP *sprite, int x, int y);
    void draw_sprite_vh_flip(BITMAP *bmp, BITMAP *sprite, int x, int y);
```

These work exactly like **draw_sprite**, except that the functions reverse the image vertically, horizontally, and in both directions. The images are exact “mirror” images; it’s not a simple rotation. The rotation function exists as well, but it works much more slowly. Here it is:

void rotate_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y, fixed angle);

Notice that, as you might have suspected, this function needs an extra argument: the angle. Only this time, it’s not a an **int** or **float**, but a **fixed**. **fixed** is a new type defined by Allegro. It’s not important to know all about it right now; we’ll see the details later. Simply note that this type is encoded in 32 bits. The first 16 bits are the integer part, and the last 16 bits are for the decimal part. Actually, this type is an **int** managed so that it can hold fractions. The values can vary from -32768 to 32767, but what we really need to know is how to convert between common types and the **fixed** type.

fixed itofix(int x);
Converts an **int** to **fixed**. It’s the same as writing x<<16.

int fixtoi(fixed x);
The inverse; converts **fixed** to **int**.

fixed ftofix(float x);
Converts a **float** to **fixed**.

float fixtof(fixed x);
And finally, as you may have guessed, conversion of **fixed** to **float**.

Voilà; now you know the esentials of this new type that is used by the **rotate_sprite** function. We must remind you that the angle is in the range 0 to 256. 256 represents a complete rotation, 64 a fourth of a circle, and so on… This function, like **draw_sprite**, will automatically skip pixles marked by the mask color. It is thus very practical for games where there are always objects superimposed on other objects, even though it is relatively slow...

Let's amuse ourselves one more time by modifying the program to illustrate these features. We will rotate the image at the same time that it moves, and the angle will very as a function of the distance from the starting point. You just have to change the line that draws our sprite:

```
/* Draw the sprite at the vertical center of the screen and rotate */
rotate_sprite(buffer, sprite1,
    (int)sprite_pos_x – sprite1->w/2,
    SCREEN_H/2 - sprite1->h/2, ftofix(sprite_pos_x));
```

Admire the result.... well,at any rate, be pleased with it. If you find that the sprite moves too quickly, change **sprite_pos_x**. This allows the rotation value to go beyond the value 256, 257 becoming 1, 258 becoming 2, etc… This is to tell you what is really happening. [Le tout est de savoir ce qu'on fait.] In any case, I can use this to present Allegro's macro **ABS(x)**. This macro replaces the value **x** by its absolute value, no matter what its type! Thus we could have written our program to create a rotation in the other direction (only for the first revolution):

```
/* Draw the sprite at the vertical center of the screen and rotate */
rotate_sprite(buffer, sprite1
    (int)sprite_pos_x- sprite1->w/2,
    SCREEN_H/2 - sprite1->h/2,
    ftofix(ABS(256 - sprite_pos_x)));
```

One last point, and it's extremely important: before exiting your program, you absolutely *must* free the memory allocated by **load_bitmap()**. Because this function copies the bitmap contents into RAM, you must free that memory space. To do this, use this function:

void destroy_bitmap(BITMAP *bmp);

Thus, the end of the preceding program is easy to write:

```
    /* Free the sprite's RAM */
    destroy_bitmap(sprite1);

    /* Free the buffer's RAM */
    destroy_bitmap(buffer);

    /* End of the main function */
    return 0;
}
END_OF_MAIN();
```

4.4 Different methods for display...

It has been quite some time ago that you asked me to talk about different methods of display, and now let's talk about it! Let me assure you that Allegro is so simple to use that managing the different display modes doesn't require any particular proficiency. As I have already mentioned, the method I've showed you is nothing more or less than **double buffering**. Let's now investiagte more [répendu] mode which also gives better performance: page flipping. We will also look at triple buffering, which, if done correctly, gives excellent results. You have to test all three methods to see which one is the most effective for your program. Never lose sight, however, of the fact that your game's performance will depend on the hardware configuration of your computer. Thus, one method might be more effective on one computer but less so on another, and vice-versa. My advice is to leave the choice to the person playing the game. I've always done it this way and I think it's a good solution to make everyone happy. Very well; enough of this chatter; let's get down to serious business! je pense que c'est une très bonne solution pour contenter tout le monde. Bon, assez bavardé, on va passer enfin aux choses sérieuses ! First, we will initialize page flipping, which, as you will see, is not at all complicated:

4.4.1 Page Flipping

```
/* Declare the two buffers we will need, and a buffer that will point
to the other two. */
BITMAP* page1, page2, buffer;
int current_page = 1;

set_color_depth(16);

/* As usual, initialize graphic mode */
if (set_gfx_mode(GFX_AUTODETECT_FULLSCREEN, 640, 480, 0, 0)  != 0)
{
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    return 1;
}

/* Now initialize page flipping... */
page1 = create_video_bitmap(SCREEN_W, SCREEN_H);
page2 = create_video_bitmap(SCREEN_W, SCREEN_H);

/* Verify that BOTH pages were created correctly */
if ((!page1) || (!page2)) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message ("Insufficient memory for page flipping!");
    return 1;
}
```

And that does it for initialization. Now, the explanation. At the beginning, as usual, we have to set the video mode; that's required. It goes without saying that we also need to verify the result... Page flipping gets its name from the two video buffers placed directly into video memory, as opposed to the mode we already saw that creates a single buffer in RAM. Creating two pages in vdieo memory has the benefit of giving acceleration by a direct display of video memory into the same memory! [vers cette même mémoire!] Remember that the variable **screen** points to an area of video memory! We must now use a special function to create a bitmap in video memory:

```
BITMAP* create_video_bitmap(int width, int height);
```

This function creates a bitmap of the given **width** and **height**. It returns a pointer to the bitmap that was created. Normally, this image in **video memory** isn't completely empty (blank); it contains whatever was left over in memory. It is thus necessary to erase it before using it. [Vous voyez, je me suis pas foulé, j'ai recopié ce que j'avais déjà écrit pour la fonction] As you can see, I've just copied and pasted what I wrote for the function **create_bitmap**, so the modification is simple... However, video memory is not a bed of roses. Even though recent video cards have ample memory, there are still lots of video cards with limited capacity. Moreover, some OS don't correctly detect the amount of available video memory with certain video cards—for example BeOS with a GeForce2MX—which makes this a very delicate procedure to use. Let me explain: two buffers that are 1024 by 768 by 32 bits take 3 megabytes of video memory... Yes,you can always try to copy the maximum number of graphics for your game into video memory with the goal of getting maximum speed, but [sachez] that this can give rise to big incompatibilities. Moreover, I don't believe that the results are always guaranteed; that is to say that the game could be slower on certain old configurations. In fact, it is necessary for a video card to support video memory to video memory copy in hardware; if not, page flipping is totally ineffective for more than blitting sprites. To sumarize, you have to be sure that two buffers have been created in video memory if you want to do page flipping. I think it's useless to try to copy what's in one buffer into the other [de copier quoi que ce soit d'autre dedans], unless you have good, and well-understood, reasons for doing so.

Page flipping is based on a very simple principle: you draw one on page, display that page, then draw on the other and display it, and so on... In effect, you alternate the drawing of the game between two pages; that's all! Here's how we write the example of the circle that moves from left to right, using page flipping:

```
while (key[KEY_ESC]) {

    clear_bitmap(buffer);

    circlefill(buffer, (int)circle_pos_x, SCREEN_H/2, 50,
          makecol(255,255,255));
     /* Move the circle to the right "0.1" pixel */
    circle_pos_x += 0.1;

    /* This is the function that displays our bitmap which is
    placed in video memory */
    show_video_bitmap(buffer);

    /* Take care of alternating the two buffers */
    if (current_page == 1) {
```

```
                current_page = 2;
                buffer = page2;
            }
            else {
                current_page = 1;
                buffer = page1;
            }
        }
```

And it's done! Add the two lines to destroy the two bitmaps, and, of course END_OF_MAIN();. Run the program. If you have a configuration that fits the bill, you should obtain a result that's cleaner and less "jumpy" than the first result example that used double buffering! There's only one new function here, and here's exactly what it does:

```
int show_video_bitmap(BITMAP *bitmap);
```

To spoil the suspense all at once, this function is very well named: it tries to do a "page flip" of **screen** to the **bitmap** passed as a parameter. This bitmap must be exactly the size of the screen and should have been created by **create_video_bitmap**; otherwise, you'll have trouble with scaling. As usual, this function returns zero if successful, non-zero if not. As you see, it's very simple. Please note that **show_video_bitmap(buffer)** is the equivalent of **blit (buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H)** for page flipping with the aforementioned restrictions. This is a good place to talk abot the problem of vertical synchornization... Ah, I see that you are confused by this; let me explain.

Ordinarily, the video card displays its contents at a particular address--which is what the global variable **screen** contains-- at a particular frequency. For example, let's say you copy a completely red screen to the **screen** bitmap. If your screen is set to 75HZ, your video card will send the contents of the **screen** address to the display exactly 75 times each second. (By the way, the higher this "refresh rate," the easier it is for your eyes.) This is where the complications occur. If the program doesn't wait for the video card to finish drawing its image on the screen, the contents of the **screen** bitmap will change while the drawing takes place—your gamewill certainly try to display its images continuously. This phenomenon happens all the time, and it explains why you the image seems to "tear" slightly whie scrolling. In the example of the moving circle, it's very noticeable. However, the **show_video_bitmap** function waits paitently for the video card to finish drawing before changing the contents of the video memory that is about to be displayed! You can test this later: count the number of images per second that you display some part of your game: it will be the same as the screen's refresh rate! This means that the computer has to be powerful, because if it isn't, your program will be overwhelmed because it can't keep up with the "cadence" imposed by the monitor's refresh rate. Coming back to double buffering, the function that serves to copy the buffer contents to the screen will draw to video memory without regard to the refresh rate.

[se moque de s'avoir où en est le dessin et modifie la mémore vidéo sans aucun scrupule.] However, there is a function to correct this:

```
void vsync(void);
```

This function waits until the image is completely drawn on the monitor. At the end of the drawing, the electron beam moves from the lower right of the screen to its beginning position (the upper left). During this period, the video card will not send anything to the screen since it can't do anything while the electron beam moves back to draw the next image. This is what we are interested in:

For simple double buffering, try putting a **vsync()** just before calling the function that "blit"s the final image to the screen. Again, as always, if your compute ris fast, this lets you obtain as many FPS (Frames Per Second, or images per second) as the refresh rate of your screen. If the computer is too slow, same thing, performance will suffer. This is why I recommend that you always allow peopleto select different display modes in your game.

That concludes the discussion about double buffering and page flipping. Experiment with existing Allegro programs and games that let you choosen different display methods (like the official demo). Again, don't hesitate to use page flipping; it is often very effective and speedy.

4.4.2 Triple Buffering

One might ask what purpose there is to triple buffering, given that page flipping works so well. You might think you're attending a course about "proliferation of technical terms to mystify users." Well, in a rough sense, triple buffering is the same thing as page flipping, except that you are using three video memory pages instead of two. Clearly, this eats up more video resources, but if managed well, it provides excellent results. Quite luckily, Allegro has code that tests if your video card will support this mode. Now that the generalities are out of the way, let's talk about details, OK?

```
/* Declare three page buffers and a pointer that will point to one of the other
two. */
BITMAP* page1, page2, page3, buffer;
int current_page = 1;

set_color_depth(16);

/* As usual, start by initializing the video mode */
if (set_gfx_mode(GFX_AUTODETECT_FULLSCREEN,
640, 480, 0, 0)  != 0) {
      set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
      return 1;
}
```

```
if (!(gfx_capabilities & GFX_CAN_TRIPLE_BUFFER))
    enable_triple_buffer();

if (!(gfx_capabilities & GFX_CAN_TRIPLE_BUFFER)) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("Triple buffering not supported!");
    return 1;
}

/* Now initialize the three buffers... */
page1 = create_video_bitmap(SCREEN_W, SCREEN_H);
page2 = create_video_bitmap(SCREEN_W, SCREEN_H);
page3 = create_video_bitmap(SCREEN_W, SCREEN_H);

/* Verify that all THREE pages were created OK */
if ((!page1) || (!page2) || (!page3)) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message ("Insufficient memory for triple buffering!");
    return 1;
}
```

If this all seems clear and obscure at the same time, that's perfectly normal. First, you have to make sure that the video card supports triple buffering. This is done with a global variable named **gfx_capabilities** :

int gfx_capabilities;

This variable is used with many **flags**. Each flag describes if the video card has a certain capability or not according to whether the designated bit is on or not (if it's 1, the capability exists, otherwise it's zero.) Chaque flag décrit si la carte vidéo est capable ou non d'effectuer certaines taches, suivant si le bit qu'il désigne est armé ou non (bon, si c'est 1 c'est que c'est bon, sinon, c'est 0). I won't go over every bit here because there are lots of them, and it would be useless to try to cover them all (see allegro/docs/html/allegro.html for the full story). However, here are some of themost useful ones:

- GFX_HW_VRAM_BLIT : tells if the video card can significantly speed up drawing by means of video memory. If so, bingo—you can experience the joys of page flipping.

- GFX_CAN_TRIPLE_BUFFER : this is the flag we're interested in right now. If it's on, that's perfect; if not, there's one last chance: the **enable_triple_buffer** function.

int enable_triple_buffer(void);

I'm sorry to spoil the mood, but this function doesn't work under certain conditions. It appears to work under DOS, but under Windows I've never gotten it to work. sous DOS mais sous Windows, je ne l'ai jamais vu marcher... In any event, it tells if triple-buffering can be enabled; it returns zero if triple buffering has been established successfully.That’s all you have to know.

Back to the initialization code. You first test the bit to see if triple buffering is active. If not, use the function and test again. If everything went well, you can go on to allocate the three video pages and verify that they have been created sucessfully. You really have to do this, because if you try for 1600 by 1200, it takes up the memory all the same. Finally, the triple buffer initialization is finished. This connects to the display of the circle: (le coup classique) :

```
while (key[KEY_ESC]) {

    clear_bitmap(buffer);

    circlefill(buffer, (int)circle_pos_x, SCREEN_H/2,
       50, makecol(255,255,255));
     /* Move the circle to the right "0.1" pixel  */
    circle_pos_x += 0.1;

    /* This is the function that displays our bitmap which is
    placed in video memory*/
    do {
    } while (poll_scroll());

    /* Post a request */
    request_video_bitmap(buffer);

    /* Take care of alternating the three buffers */
    if (current_page == 1) {
       current_page = 2;
       buffer = page2;
    }
    else  if (current_page == 2) {
       current_page = 3;
       buffer = page3;
    }
    else {
       current_page = 1;
       buffer = page1;
    }
}
```

This is a good thing indeed. [Bien, voilà une bonne chose de faîte] ! To prove it, the general principle is the same for changing the pages; let’s explain the new functions:

int request_video_bitmap(BITMAP *bitmap);
This function should be used only with triple buffering. It requests a “page flip” on the **bitmap** passed as a parameter. Specifically, as opposed to other functions, this takes effect [rend la main] immediately.

int poll_scroll(void);
This function is also used only with triple buffering. Simply put, it verifies that the previous function has completed its task. It returns zero if it is possible to call **request_video_bitmap** and returns some non-zero value if not.

En clair, on dessine sur une page. Ensuite, on demande à ce que la page soit affichée avec la fonction **request_video_bitmap**, enfin, on dessine sur une autre page, et on attend que la première se soit complètement affichée. Dès que c'est fait, on fait encore un **request_video_bitmap** sur la seconde et on embraye le dessin sur une troisième page, pendant que la première disons soit "prète à nouveau".

4.4.3 Combiner plusieurs méthodes d'affichage dans un même programme

La je sors le grand jeu, puisque nous allons faire cohabiter les trois modes d'affichage dans un même et unique programme ! En fait, je vais faire trois fonctions ici. La première initialisera le mode graphique en fonction de la méthode choisie. La deuxième sera appelée avant de dessiner le jeu sur le buffer. Et la dernière sera utilisée pour coller le buffer à l'écran. Vous n'aurez donc plus qu'à créer la fonction de dessin, qui affichera sur un buffer, independamment de la méthode choisie, ce qui est très intéressant !

Mais tout d'abors, définissons quelques **defines** et quelques variables externes pour nous faciliter la vie. Et bien sur on crée les prototypes des trois fonctions. Rien ne vous empèche d'ailleurs de créer un fichier exprès pour l'affichage en déclarant les BITMAPS* comme locaux à ce fichier. C'est d'ailleurs plus cohérent mais faîtes comme vous voulez du moment que ça marche.

```
#define DOUBLE_BUFFERING      1
#define PAGE_FLIPPING         2
#define TRIPLE_BUFFERING            3

BITMAP* page1, page2, page3, buffer;
int draw_method, current_page;

/* La fonction qui copie le buffer sur l'écran */
void buffer_onto_screen(void);

/* La fonction qui prépare le buffer avant le dessin */
```

```
void prepare_drawing(void);

/* La fonction qui initialise le mode vidéo */
int init_video(int draw_method, int size_x, int size_y, int color_depth, int
WINDOWED_MODE);
```

Allez, on commence par le plus gros morceau : **init_video**. Ce n'est pas du tout une fonction d'Allegro officielle puisque ça va être la notre donc je ne la présente pas comme les autres. Reprenons, cette fonction prend en paramètre la méthode d'affichage **draw_method**, qui a donc le droit de valoir **DOUBLE_BUFFERING**, **PAGE_FLIPPING** ou **TRIPLE_BUFFERING**. On lui passe aussi la résolution de l'écran choisi (**size_x** * **size_y**), la profondeur de la palette de couleur **color_depth** qui peut donc valoir 8, 15, 16, 24 ou 32. Et en enfin **WINDOWED_MODE** est le paramètre qui détermine si l'on veut ou non utilser un mode fenetre ou un mode plein écran. Cette variable a le droit de valoir TRUE ou FALSE (ce sont deux noms definis dans allegro.h). Notez que j'aurais pu définir deux defines par exemple FULLSCREEN et WINDOWED et envoyer un ou l'autre à la fonction... Mais bon, il faut bien faire un choix et sur le coup, j'avoue que je suis plutot feignant... Allez, maintenant que vous savez exactement son rôle, vous allez l'écrire ! Heh, rassurez-vous, je suis là pour vous la donner :

```
int init_video(int config_draw_method, int size_x, int size_y, int
color_depth, int WINDOWED_MODE) {
        int gfx_mode;

        if (WINDOWED_MODE == TRUE) {
                gfx_mode = GFX_AUTODETECT_WINDOWED;
                color_depth = desktop_color_depth();

                /* Si votre jeu est en true color mais que le bureau est en 256
couleurs, forcez le 16 bits quand même*/
                if (color_depth < 16)
                        color_depth = 16;
        }
        else
                gfx_mode = GFX_AUTODETECT_FULLSCREEN;

        /* on peut appliquer tranquillement le color_depth maintenant */
        set_color_depth(color_depth);
```

Alors, on va faire une petite pause ici... Il faut que je vous explique le coup du **desktop_color_depth();**

intdesktop_color_depth();

Cette fonction retourne la profondeur de la palette de couleur utilisée par le bureau actuel d'où est lancé le programme. Alors bien sur, cette fonction n'est interessante que si elle est appelée à partir d'une fonction en mode fenetré. Pourquoi donc? Et bien, tout simplement parce qu'une application tournera beaucoup plus vite si elle utilise elle même la même palette que celle du bureau. Sinon, des conversions supplémentaires vont se mettre en place pour convertir l'affichage de la fenêtre par rapport à celui du bureau d'où la perte de temps. Et croyez moi, c'est bien plus rapide de faire une fenêtre en 32 bits plutot que en 16 sur un bureau lui-même 32 bits! Cette fonction retourne 0 si elle n'est pas capable de déterminer le resultat ou tout simplement s'il n'a pas lieu d'exister (exemple sous DOS).

On enchaîne avec la suite. Mais avant de continuer, je voulais vous signaler que l'on va faire une fonction intelligente : si elle voit que ce n'est pas possible de faire du page flipping ou du triple buffering, elle va se rabattre automatiquement vers le double buffering, qui lui est quasiment sûr de marcher à tous les coups.

```
		if (set_gfx_mode(gfx_mode, resol_x, resol_y, 0, 0) != 0) {
			set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
			allegro_message("%s", allegro_error);
			return FALSE;  /* FALSE comme erreur ici bien sur */
		}

		/* Ici, pas de surprise, on retrouve ce que l'on avait vu tout à l'heure,
enfin un peu plus haut */

		if (config_draw_method == TRIPLE_BUFFERING)
			enable_triple_buffer();

		if (gfx_capabilities & GFX_CAN_TRIPLE_BUFFER &&
config_draw_method == TRIPLE_BUFFERING)
			draw_method = TRIPLE_BUFFER;
		else if (!(gfx_capabilities & GFX_CAN_TRIPLE_BUFFER) &&
config_draw_method == TRIPLE_BUFFERING)
			draw_method = DOUBLE_BUFFERING;
		else
			draw_method = config_draw_method;

		/* Si on a passé tous les tests pour le triple buffering... */
		/* Tout est déjà connu et sans surprise par ici... */

		if (draw_method == TRIPLE_BUFFERING) {
			page1 = create_video_bitmap(SCREEN_W, SCREEN_H);
			page2 = create_video_bitmap(SCREEN_W, SCREEN_H);
			page3 = create_video_bitmap(SCREEN_W, SCREEN_H);
			if ((!page1) || (!page2) || (!page3))
				draw_method = DOUBLE_BUFFERING; /* on passe au
mode par défaut */
		}

		if (draw_method == PAGE_FLIPPING) {
```

```
                        page1 = create_video_bitmap(SCREEN_W, SCREEN_H);
                        page2 = create_video_bitmap(SCREEN_W, SCREEN_H);
                        if ((!page1) || (!page2))
                                draw_method = DOUBLE_BUFFERING; /* on passe au
mode par défaut */
                }

                if (draw_method == DOUBLE_BUFFER)  {
                        buffer = create_bitmap(SCREEN_W, SCREEN_H);
                        if(!buffer) {
                                allegro_message("Impossible de créer un buffer de
(%d*%d)", SCREEN_W, SCREEN_H);
                                return FALSE;
                        }
                }

                /* Tout va bien ici */
                return TRUE;
        }
```

Alors, je vous préviens tout de suite, il faut faire attention avec cette fonction : en effet, si jamais l'initialisation du triple buffering échoue à la création du troisième buffer vidéo, vous allez continuer avec du double buffering et avec deux pages de mémoires vidéo allouées qui ne seront jamais libérées. Je vous présente vraiment cette fonction pour vous montrer le principe. Vous devriez avertir l'utilisateur en lui précisant de changer le mode vidéo au plus vite par exemple. Ce n'est pas très dur à faire, c'est juste l'histoire de quelques **allegro_message**. Je vous apprendrais plus tard comment on peut très bien se sortir de ce genre genre de situations en utilisant des **"config files"**. Cette fonction retourne donc TRUE si tout s'est bien passé, sinon FALSE. Voilà comment on pourrait l'appeler par exemple depuis la partie initialisation de votre programme :

```
        if (init_video(PAGE_FLIPPING, 800, 600, 16, TRUE) == FALSE) {
                /* Erreur à gérer ici -> il faut quitter */
                return 0;
        }
        /* Et ici, tout va bien ! */
```

Et voilà, la partie d'initialisation est terminée... C'était vraiment pas très dûr n'est-ce pas? Alors, maintenant, on va s'intéresser à la fonction qui prépare le buffer à être déssiné ! On va appeler cette fonction **prepare_drawing**, et on décide qu'elle ne recoie pas d'argument et ne renvoie rien.

```
        void prepare_drawing(void) {

                if (draw_method == TRIPLE_BUFFER) {
```

```
                if (current_page == 0) {
                        buffer = page2;
                        current_page = 1;
                }
                else if (current_page == 1) {
                        buffer = page3;
                        current_page = 2;
                }
                else {
                        buffer = page1;
                        current_page = 0;
                }
        }
        else if (draw_method == PAGE_FLIPPING) {
                if (current_page == 2) {
                        buffer = page1;
                        current_page = 1;
                }
                else {
                        buffer = page2;
                        current_page = 2;
                }
        }
        if (draw_method != DOUBLE_BUFFER)
                acquire_bitmap(buffer);

        return;
}
```

Et c'est tout bon pour la préparation au dessin d'une nouvelle image! Comme vous pouvez le constater, il n'y a rien de bien nouveau en fait, car tout a déjà était vu précédemment. Nouveauté cependant, j'espère que vous l'avez vue, c'est la fonction **acquire_bitmap**! Ne perdons pas de temps et allons découvrir à quoi elle sert :

void acquire_bitmap(BITMAP *bmp);
Cette fonction vérouille le bitmap vidéo **bmp** avant de dessiner dessus. Elle ne marche donc pas sur les bitmaps créés avec **create_bitmap**, juste ceux créés avec **create_video_bitmap**. De plus, elle ne concerne que certaines plateformes uniquement. Par exemple, Windows doit l'utiliser alors que DOS non. Alors, là vous devez penser : mais pourquoi n'en a-t-il pas parlé avant ? Ca a l'air super important ! Oui, en effet, ça l'est, mais laissez moi me défendre : à chaque fois que vous utilisiez **blit** auparavant, ou une fonction de dessin quelconque, sur le buffer, cette fonction était en fait appelée tout seule! Alors on pourrait se demander où est l'intérêt de la chose ? C'est très simple, vérouiller une surface DirectDraw est très lent, donc, plutot que de vérouiller / dévérouiller cinquante fois pour afficher cinquante dessins à l'écran -l'astuce s'impose toute seule-, on vérouille le buffer avant de dessiner quoi que ce soit dessus, et on le

dévérouille une fois tout le dessin terminé ! C'est précisement ce que nous faisons ici. Lorsqu'on prépare le dessin, on intervertit les buffers vidéos et on en profite pour vérouiller le buffer courant ! Attention toutefois, pensez bien à déverouiller le bitmap à la fin de votre dessin, car les programmes DirectX ne peuvent plus recevoir de signal en entrée, c'est à dire tout ce qui n'est pas graphique du genre les timers, le clavier, la souris et Cie... tant que le bitmap est vérouillé. Allez, puisqu'on y est, on peut passer à la fonction associée :

void release_bitmap(BITMAP *bmp);

Relache **bmp** qui normalement a été vérouillé par la fonction **acquire_bitmap**. Si par hasard vous avez vérouillé un bitmap plusieurs fois de suite, il faut le dévérouillé autant de fois !

Dernier petit effort, la fonction qui affiche le buffer complètement dessiné sur l'écran : on la nomme **buffer_onto_screen**. Elle ne prend et ne donne rien, tout comme la précédente.

```
void buffer_onto_screen(void) {
    if (draw_method != DOUBLE_BUFFER)
        release_bitmap(buffer);

    if (draw_method == TRIPLE_BUFFER) {
        /* Il faut être sur que le dernier flip a bien eu lieu */
        do {
        } while (poll_scroll());

         /* On demande à  ce que le buffer soit affiché */
        request_video_bitmap(buffer);
    }
    else if (draw_method == PAGE_FLIPPING)
        show_video_bitmap(buffer);

    else if (draw_method == DOUBLE_BUFFER)
        blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);

    return;
}
```

Si vous ne trouvez pas ça difficile ou que vous n'êtes pas surpris de la tête de la fonction, alors c'est bon signe ! Eh bien, c'est avec une émotion certaine que je viens de terminer la première chose que je voulais faire apparaître dans mon tutorial. mais comme vous pu le constater, il a largement dévié pour devenir le plus généraliste possible à propos d'Allegro, et j'espère que vous n'allez pas vous en plaindre ! Pour fêter tout ça, on regarde un petit exemple d'application :

```
    if (init_video(PAGE_FLIPPING, 800, 600, 16, TRUE) == FALSE) {
        /* Erreur à gérer ici -> il faut quitter */
        return 0;
    }
    /* Et ici, tout va bien ! */

    circle_pos_x = 0;

    while (key[KEY_ESC]) {

        prepare_drawing();

        /* Ici commence votre section d'affichage perso, je remets des
exemples déjà vus pour voir... */
        clear_bitmap(buffer);

        circlefill(buffer, (int)circle_pos_x, SCREEN_H/2, 50,
makecol(255,255,255));

         /* On décale le disque vers la droite de "0,1" pixel */
        circle_pos_x += 0.1;

        /* Ici, c'est la fonction qui affiche notre bitmap qui est placé en
mémoire vidéo */
        buffer_onto_screen();

    }
```

La grande classe, vous choisissez votre mode d'affichage de façon complètement indépendante de la partie qui s'occupe du dessin proprement dit! Vous ne vous occupez que de la variable **buffer**, qui pointe directement vers le buffer qui nous interesse. n'hesitez pas à faire beaucoup de tests, en changeant les modes et la résolution, et bien sûr en incluant votre propre séquence de dessin là où il le faut!

5 . Le mode d’affichage 8 bits.

5.1 Introduction

Nous avons vu combien il était facile d’utiliser des modes **truecolor** avec Allegro, c’est à dire 15, 16, 24 ou 32 bits par pixel. Facile car le code nécessaire pour afficher des **sprites** est indépendant pour chaque mode de couleur vidéo. Mais si vous ne comptez pas utiliser beaucoup de couleurs dans votre programme, et si vous recherchez avant tout un grand gain de vitesse, le mode 8 bits est fait pour vous! Il n’est pas beaucoup plus compliqué à utiliser. Il fait en revanche intervenir une notion supplémentaire: la notion de

palette de couleur.

Comment donc se présente cette palette, me direz-vous? Et bien, une palette est tout simplement composée de 256 couleurs, chaque couleur étant codée par les trois paramètres RGB décrits avec le prototype de la fonction **makecol()** . Le **sprite** fait ensuite référence à l'index d'une couleur. Comme il y a 256 couleurs dans la palette, le sprite est bien codé en 8 bits par pixels car 2^8 = 256. Concrètement, si l'index **2** de la palette contient la couleur (255,255,255), c'est à dire le blanc, tout pixel ayant **2** comme valeur sera en fait blanc! Encore mieux donc, on peut mettre n'importe quelle couleur dans la palette. On peut donc créer par exemple une palette contenant uniquement des dégradés de gris! Faisons maintenant le lien avec Allegro: une image enregistrée au format 8bpp sera automatiquement enregistrée avec sa propre palette. Vous vous souvenez de la fonction **load_bitmap?** Comment ça non? Et bien voici son prototype à nouveau:

BITMAP *load_bitmap(const char *filename, RGB *pal);

Cette fois-ci, nous n'allons pas passer NULL pour la valeur de **pal**, mais on va passer comme argument un pointeur vers une palette. Une palette est de type **RGB**, on va la déclarer en tant que pointeur ainsi:

```
/* Déclaration de la palette */
RGB* ma_palette;
```

Une palette est donc en fait un simple tableau contenant 256 éléments RGB. Le fait de passer ce pointeur en argument de la fonction va mettre la palette du sprite dans **ma_palette!** Ainsi, pour utiliser la palette du sprite "sprite.bmp" (n'oubliez pas qu'il doit s'agir d'une image enregistrée en 8 bits!), on tapera la commande:

```
BITMAP* sprite1;

/* On charge le bitmap, ma_palette contient la palette du sprite! */
sprite1 = load_bitmap("sprite.bmp", ma_palette);
```

Il faut maintenant dire au programme qu'on veut utiliser **ma_palette** en tant que palette courante. Oui, je dis bien palette courante car un programme ne peut utiliser qu'une seule palette à la fois pour dessiner. Inutile d'essayer d'alterner différentes palettes pendant l'affichage, le resultat est inexploitable, à moins bien sûr d'utiliser des techniques spécifiques… Mais il vaut mieux passer en mode truecolor si on a vraiment besion de plus de 256 couleurs. Vous pouvez toutefois essayer pour vous en rendre compte. Pour définir la palette courante à utiliser, on utilise la fonction:

void set_palette(const PALETTE p);

preprésente un tableau de 256 RGB. On peut donc passer directement **ma_palette** en paramètre pour cette fonction.

Si l'on récapitule notre début de programme, on a le code suivant:

```
/* On inclut le header de la librairie */
```

```
#include <allegro.h>

/* Et on commence notre fonction main! */
int main()   {

            /* Déclaration de la palette */
            RGB* ma_palette;

            /* Du sprite */
            BITMAP* sprite1;

            /* Du buffer vidéo */
            BITMAP* buffer;

            /* Fonction d'initalisation générale */
            allegro_init();

            /* Initialise le clavier */
            install_keyboard();

            /*  On initialise le mode graphique de couleur. Cette fois-ci 8 bits!*/
            set_color_depth(8);

            if(set_gfx_mode(GFX_AUTODETECT, 640, 480, 0, 0)!= 0) {
                 set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
                 allegro_message("Impossible d'initialiser le mode vidéo!\n%s\n",
allegro_error);
                 return1; //et on oublie pas de quitter…
            }

            /* On charge le bitmap, ma_palette contient la palette du sprite! */
            sprite1 = load_bitmap("sprite.bmp", ma_palette);

            /* Et maintenant, on définit la palette à utiliser */
            set_palette(ma_palette);
```

Vous pouvez à présent utiliser les fonctions classiques pour afficher vos bitmaps telles que **blit**, **draw_sprite** et tous ses dérivés. Je dois quand même vous signaler quelques restictions au niveau de **makecol**. Cette fonction était très pratique pour définir une coleur en mode truecolor, mais elle s'avère plus délicate à utiliser en mode 8 bits. Elle marchera mais vous aurez des resultats plus ou moins probants selon la palette utilisée. Le mieux est d'entrer directement **l'index de la couleur** au lieu d'appeler **makecol**. Par exemple, si vous voulez faire un rectangle "plein", vous n'avez qu'à utiliser la fonction suivante (c'est l'occasion de découvrir une nouvelle fonction...):

void rectfill(BITMAP* bmp, int x1, int y1, int x2, int y2, int color);

Cette fonction dessine donc un rectangle “plein”, entre les points de coordonée (**x1**,**y1**) et (**x2**,**y2**). Petite précision utile : il n'est pas nécéssaire que le point 1 soit au dessus et à gauche du point 2. Essayez vous même, on intervertir les deux points sans changer le resultat visuel. Alors, revenons à notre problème de couleur... En mode truecolor, si vous voulez un rectangle bleu sur votre écran:

```
rectfill(buffer, 25, 5, 1, 1, makecol(0,0,255));
```

Mais maintenant, si votre palette contient la couleur noir à l'index 58, faites simplement:

```
rectfill(buffer, 25, 5, 1, 1, 58);
```

Ca sera de plus plus rapide que de faire un appel à **makecol**.

Le mode 8 bits présente un autre avantage : Pour effectuer des fondus sur l'écran, on peut toucher que la palette courante sans toucher aux sprites eux-même! On obtient ainsi de super effets sans se fatiguer. On se fatiguera encore moins que prévu car Allegro fournit de quoi faire des effets sympathiques!

void fade_out(int speed);

Cettefonction va décolorer l'écran progressivement jusqu'à obtenir une image complétement noire. Le paramètre **speed** prend des valeurs allant de 1 (le plus lent) jusqu'à 64 (instantané). N'hesitez pas à l'utiliser car elle rend les jeux tout de suite plus “professionnels”.

6 . La souris

There’s no need to divide this chapter into a sub-chapter, because using the mouse is so simple! We’ve already seen at the beginning of this tutorial how to initialize the mouse. Now, let’s see how to use it. But to begin, I would like to refresh your memory be repeating the code for mouse initialization here:

```
/* If the function fails, then… */
if (install_mouse() == -1) {
     /* Display an error message */
     allegro_message("Error! %s", allegro_error);

     /* And exit the program! */
     return -1;
}
/* Otherwise, you are sure you have a mouse! */
```

And it’s done. install_mouse is an Allegro function which returns the number of available mouse buttons. That is to say, most likely it will return 3. Other mice will return

2. Of course, the most unfortunate case is if you dn't have a mouse at all, so you'll receive a -1. Never forget to do this test; it doesn't take a lot of code and it can truly simplify your life in case of problems. I will concede that, nowadays, it's very rare to not have a mouse, but who can ever tell? A DOS user, for example, could have a system without a mouse driver installed... Checking for the mouse is a good habit to make: systematically verify the values that initialization functions return. But enough of this chatter; let's see how to use this mouse. It's easier than it seems: you need only a few exteral variables:

```
extern volatile int mouse_x;
extern volatile int mouse_y;
extern volatile int mouse_z;
extern volatile int mouse_b;
```

This isn't terribly confusing: mouse_x represents the current horizontal coordinate of your mouse; mouse_y the current vertical coordinate. Remember that these are relative to the upper left corner of your screen. Thus, the pixel at that corner has coordinates (0,0). You can find out the location of the mouse pointer at any time; it's at coordinates (mouse_x, mouse_y). Thus, it has a range from (0,0) to (SCREEN_W, SCREEN_H). Ah, but you ask, how can I refresh the values of these variables? Just as with the keyboard, you don't have to; Allegro does it all by itself. Strictly speaking, you don't have to do anything to be sure that your program knows the place where the user (that is, the player) has positioned the mouse. In regard to the variable mouse_z, that represents the position of the mouse scroll wheel, if there is one. When you call init_mouse(), mouse_z is set to 0 by default. If you scroll with the mouse, the value will be incremented or decremented. The mouse_b variable represents the other essential aspect of the mouse: this variable reflects the state of the mouse buttons. Thanks to this variable, you'll know if the user is in the process of clicking the left, right, or middle button. Bit 0 is the left button, bit 1 is the right button, and bit 2 is the middle button. You will soon see all of this in practice: creating a program that initializes the mouse, the video mode, loads a sprite into memory and creates a buffer the size of the screen. Once this is displayed, you shouldn't have any further programs. Now we will rewrite the main loop, using the mouse instead of the keyboard. For the sprite, try to select a small image for the mouse pointer; it will be cleaner than working with a simple image that represents a wall!

```
while(!(mouse_b & 2)) { /* While the right button isn't pressed… */

    /* Start by clearing the buffer */
    clear_bitmap(buffer);

    /* If left button, draw the sprite at the mouse coordinates */
    if(mouse_b & 1)
        draw_sprite(buffer, sprite, mouse_x, mouse_y);
    /* Now, we need to copy our buffer contents to the screen */
    blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);
}
```

After this loop, dont' forget to free your **sprite** et le **buffer**. This is the moment of truth: compile and run the program. If you click the left button, your sprite should appear at the coordinates of your mouse! On the other hand, if you don't click, you'll just see a black screen. It's easy to quit the program; just click the right button. Simple, isn't it? There's another method to display the mouse automatically. Personally, though, I won't describe it because I find it to be "useless." Finally, let's say that the method I've shown you to display the mouse gives you copmlete control and is more transparent than the other method. In effect, the fact of displaying within the main loop [avec tout le reste] lets you do whatever you want as simply as possible: You need only change the **sprite** to another sprite to display a different cursor. If you feel the need to offset the mouse to the left or right, nothing stops you from doing this:

```
    draw_sprite(buffer, sprite, mouse_x + 50, mouse_y - 35);
```

It's also very easy to mask the cursor when you want to. You can create animated cursors more easily this way... You'll see how to do this easily later on, specifically by using timers. There are other useful functions that can save you effort for certain types of games:

void get_mouse_mickeys(int* mickeyx, int* mickeyy);

This function measusres the distance that the mouse has moved since the last call to the same function. That is, to say, if the user tries to move too far to the right off the screen, this function can detect such a movement. Note that because it detects a displacement, not a position, it can be very practical for games that need constant mouse movement. You pass two variables as parameters; the function will modify them to contain the new values.

void set_mouse_range(int x1, int y1, int x2, int y2);

Very simply, this function serves to define the range of movement for the mouse. By default, if you don't call this function, you are restricted to this area: (0,0)-(SCREEN_W-1, SCREEN_H-1). You pass as parameters the coordinates of the upper left and lower right of the mouse zone.

void position_mouse(int x, int y);

This function is useful if you want to re-center or place the mouse precisely where you want it. Nothing special here; just pass the new coordinates and the mouse is there.

I think you now know everything essential for working with your mouse. There are some other interesting functions; take a look at the documentation if you want to see more possibilities. But you should be able to realize that these other interfaces to the mouse may not be as “easy” as the one I’ve shown you...

8. Timers

8.1 Introduction - Theory (simplified)

There are some absolutely indispensable utilities, and when the time arrives that you decide one day to make a decent game, you simply can’t *not* use them...Why? Put simply, it seems that up to now, the speed of execution of all the examples (or games?) that you have been able to program depend directly on the speed of the processor and graphics card on which they run! Evidently, this seems to be the first really problematic aspect. How can one handle this? In fact, you should think about it this way: how can we be sure that the speed of the final result will be the same on all machines? Clearly, you can’t depend on anything whatsoever belonging to the computer itself; that’s certain death. And programmers have for a long time have found the answer: it’s sufficient to use as a base something that all machines have and that is sufficiently reliable for what you want to do: the CPU’s internal clock. This is terribly oversimplified, but sufficient to understand a little how it all works. I am going to talk about things that are less hardware oriented, but nevertheless quite important for us.

You’ve all noticed something until now: a game is more fluid if it has a greater fps (Frames Per Second). You’ve also noticed that the FPS are higher if you set up a few things. Really, it’s quite simple: up until now the reasoning has been as follows: If the computer is faster, the game is faster. One could note that by proceeding thus, one obtains a frame rate

remarquer qu'en procédant ainsi, on obtient un framerate (= taux de fps) à peu près dépendant de ce qu'on veut afficher. Si par exemple on affiche une trentaine de sprites sur l'écran, le fait de passer à une centaine de sprites va à coup sûr couler le framerate. Mais comme la vitesse du jeu est directement proportionnelle au framerate, on aura une vitesse plus faible! Alors là, j'en entends d'ici quelques uns s'étonner de ce que je viens de dire. C'est pourtant la stricte vérité! Je m'en défends donc à l'instant. Voilà la structure d'un jeu telle que l'on a fait jusqu'à présent en pseudo code maison:

```
Début de la boucle principale
    1) On récupère les données d'entrées (joystick, clavier, souris, ...)
    2) On incrémente les variables relatives au jeu (positions, vitesses, actions diverses...)
    3) On affiche le beau resultat!
Fin de la boucle principale
```

Si une de ces trois étapes n'a pas une durée constante : le jeu va se dérouler à une vitesse non constante! Il est clair que ce qui fait perdre le plus de temps au programme, c'est bel et bien l'affichage! Et de très loin encore... C'est à dire que les temps d'execution des étapes [1) et 2)] seront négligeables devant 3). Donc si 3) doit afficher 50 sprites à un moment et le double à un autre, le temps d'execution de toute la boucle va varier du simple au double! A un moment les éléments du jeu bougeront deux fois moins vite... J'espère que vous avez saisi l'idée... Passons maintenant à la solution...

On va donc créer un compteur qui va s'incrementer toutes les millisecondes (par exemple). Ici, il s'intitule **compteur**.

Cette fonction s'appele "toute seule" toutes les millisecondes. Elle ne fait ni plus ni moins que ça :

```
{
    On incrémente compteur;
}
```

Maintenant, dans notre boucle principal on va retrouver:

```
Début de la boucle principale
    SI compteur > 0
        1) On pique les données d'entrées (joystick, clavier, souris, ...)
        2) On incrémente les variables relatives au jeu (positions, vitesses, actions diverses...)
        - On décrémente compteur
    SINON
        3) On affiche le beau resultat!

Fin de la boucle principale
```

Alors là, je vous sens perplexes... Pourquoi est-ce que diable ce truc marcherait? Imaginez-vous un peu... **compteur** vaut 0, on rentre dans la boucle principale... On affiche le jeu dans ses valeurs initiales. Mettons qu'il faille exactement 20ms pour les afficher : **compteur** vaut maintenant 20. Puisqu'il est positif, on va effectuer à la suite 1) et 2) exactement 20 fois (puisque le temps mis pour les executer est considéré ici comme négligeable). Si maintenant d'aventure 40ms sont nécessaires pour afficher le jeu : eh bien on va effectuer 40 fois de suite 1) et 2). Pour faire plus clair : s'il se trouve que c'est deux fois plus long d'afficher le jeu, on va déplacer les éléments deux fois plus! Si l'on compte bien : en 1 seconde on va effectuer 1000 fois 1) et 2), et ce quelque soit le temps mis pour afficher les images! On a atteint notre but! Bon, si vous n'avez pas compris, prenez papier-crayon et représentez-vous calmement ce qui se passe dans la boucle à partir des valeurs initiales...

Il existe toutefois un petit risque... Si le temps nécéssaire pour réaliser 1) et 2) dépasse la milliseconde, on se retrouve dans une impasse : les images ne pourront jamais s'afficher. En effet : si compteur vaut 0, et qu'il vaut 2 à la sortie de 1) et 2), il repasse à 1 juste avant de refaire une boucle (une ligne décrémente le compteur). La condition compteur>0 est toujours vraie, et le jeu ne peut jamais s'afficher.

Si par contre le phénomène n'arrive qu'une seule fois, l'utilisateur final ne vera rien ou presque : au pire un petit saut dans les déplacements. Le tout est de bien calibrer la période l'incrémentation du compteur pour l'utilisation qu'on compte en faire.

On a vu que ça marchait bien si le temps mis pour réaliser 1) et 2) était négligeable... Maintenant que faire si 1) et 2) varient de façon non négligeables? Est-ce que ça marcherait aussi? Eh bien oui, vous allez le voir grace à un exemple (là encore, prenez un papier et un crayon!)

- Mettons [1) puis 2)] nécessite 0,9 ms et 3) 20ms
  On commence, on affiche, et **compteur** vaut 20.
  Ensuite, **compteur** vaut 20.9 puis 19.9, 20.8, 19.8, ..., 1.0 puis 0 -> on s'arrete!
  On va effectuer exactement 200 fois [1) puis 2)]
  Donc, le temps réalisé pour afficher une image est de 200*0,9 + 20 = 200ms! (5 FPS)
  Et là encore, on réalise [1) puis 2)] 1000 fois par seconde...

Et là tout d'un coup votre programme doit gérer de gros calculs et tout devient différent:

- Mettons [1) puis 2)] nécessite 0,5 ms et 3) 20ms
  On commence, on affiche, et **compteur** vaut 20.
  Ensuite, **compteur** vaut 20.5 puis 19.5, 20, 19.5, ..., 1.0 puis 0 -> on s'arrete!
  On va effectuer exactement 40 fois [1) puis 2)]
  Donc, le temps réalisé pour afficher une image est de 40*0,5 + 20 = 40ms! (25 FPS)
  Et là encore, on réalise [1) puis 2)] 1000 fois par seconde...

Oui en effet ça sent le copier/coller douteux mais ça marche bel et bien! Maintenant vous l'avez vu, cette méthode marche quel que soient les situations, exeptée bien sûr celle que je vous ai décrite un peu plus haut. Conclusion : 2) s'execute à vitesse moyenne constante, et ce quelque soit le temps mis pour l'affichage ou le temps mis pour le calcul de 2). Plus l'ordinateur sera rapide, et plus il affichera d'images, mais ce ne sera jamais au dépend du taux de raffraichissement des données du jeu.

Bon, la théorie est là, vous savez comment faire. Maintenant on ve se pencher sur des questions plus pratiques.

8.2 Pratique

Alors, il va falloir faire des déclarations que vous n'avez pas encore vues... Elles sont spécifiques à Allegro et assurent sa compatibilité multi-plateforme. Je me doute que vous devez prendre la nouvelle de la façon : "ah zut, c'est pas aussi simple qu'il ne le prétend! Il m'a bien eu!". Rassurez-vous je vais juste vous expliquer ce dont vous avez besoin et vous allez voir, ça va très bien se passer...

Alors, le mystère réside dans la fameuse fonction qui s'appele toutes les millisecondes... Il va falloir la déclarer ainsi:

```
/* on déclare la variable */
volatile int game_time;

/* Et le timer pour ajuster la vitesse du jeu */

void game_timer(void) {
    game_time++;
}
END_OF_FUNCTION(game_timer);
```

END_OF_FUNCTION() est une macro au même titre que **END_OF_MAIN()**. Le problème vient du fait que sous DOS et dans certains autres environements, la mémoire peut être virtuelle. Elle peut donc se retrouver écrite sur le disque. Vous vous imaginez la catastrophe : le programme écrit sur le disque la fonction **game_timer** et nécessite une dizaine de millisecondes pour l'executer! Tout est faussé, et le timer devient complètement inéfficace... Il convient donc de forcer l'écriture de la fonction en mémoire vive. Pour cela on la déclare comme indiqué pour automatiser la chose. De plus, lors de l'initialisation du timer, il faudra penser à bloquer la fonction en mémoire grace à la macro **LOCK_FUNCTION()**. De même il faut être sûr de bloquer **game_time** en mémoire grâce à **LOCK_VARIABLE()**. Ce qui nous donne lors de l'initialisation:

```
LOCK_FUNCTION(game_timer);
LOCK_VARIABLE(game_time);
```

Ca y est, vous avez initialisé votre compteur! C'était pas vraiment difficile n'est-ce pas? Maintenant reste la dernière étape : lancer le compteur que l'on a crée. Pour cela, une simple fonction:

int install_int(void (*proc)(), int speed);
Cette fonction va en fait installer notre timer. On ne peut pas crée autant de timers que l'on veut. Si cette fonction renvoie un nombre négatif, c'est que la création a échoué! En paramètre nous avons bien sur la fonction en question et puis la vitesse **speed** exprimée en millisecondes. Si vous n'avez pas lancé **install_timer()** auparavant, elle le fera toute seule (cette fonction initialise les routines reliées aux timers). Inutile donc de vous embeter avec ça. De plus, lors de la fermeture du programme, le timer est détruit, donc pas la peine non plus de le détruire à la main. Pour notre exemple, l'initialisation complète se ferait ainsi :

```
install_int(game_timer, 1); /* 1 ms de temps de résolution */
```

```
        game_time = 0;

        LOCK_VARIABLE(game_time);
        LOCK_FUNCTION(game_timer);
```

Eh bien je crois que vous savez tout maintenant! Essayez d'inclure vous même la structure que je vous ai décrite au paragraphe d'introduction avec le système qui déplace tout seul un sprite sur l'écan en fonction de l'appui sur les touches fléchées... Votre programme tournera en temps réél! Si vous séchez, c'est à dire si une étape vous manque, regardez la solution complète ci-dessous. Ne la regardez que si vous bloquez, c'est de loin la meilleure façon d'assimiler!

8.3 Un programme complet en temps réél... Avec un compteur de FPS!

```
#include <allegro.h>

/* Les incréments pour les déplacements */
#define INCREMENT_X 0.25
#define INCREMENT_Y 0.25

/* la variable qui controle le temps réel */
volatile int game_time;
/* La variable qui compte les secondes de jeu depuis le début */
volatile unsigned long sec_counter;
/* La variable provisoire qui sert à detecter si on a changé de seconde (on peut faire la même chose avec un flag) */
volatile int sec_counter;

/* un nouveau type : VECTOR */
typedef struct{
        double x,y;
}VECTOR;

/* Prototypes de fonctions */
/* Cette fonction initie les timers. Elle retourne 0 si tout s'est bien passé */
int init_timer(void);


/* le timer pour ajuster la vitesse du jeu */
void game_timer(void) {
  game_time++;
}
END_OF_FUNCTION(game_timer);
```

```
/* le timer qui donne le nombre de secondes du jeu */
void game_sec_timer(void) {
		sec_counter++;
}
END_OF_FUNCTION(game_sec_timer);

int init_timer(void) {
		install_timer();

		/* Résolution : 1 ms */
		if (install_int(game_timer, 1) < 0)
			return 1;

		/* Résolution : 1 s */
		if (install_int(game_sec_timer, 1000) < 0)
			return 1;

		sec_counter = 0;
		game_time = 0;

		LOCK_VARIABLE(game_time);
		LOCK_VARIABLE(sec_counter);
		LOCK_FUNCTION(game_timer);
		LOCK_FUNCTION(game_sec_timer);

		return 0;
}


int main() {

		/* Le compteur qui change à chaque dessin */
		int current_fps = 0;
		/* Celui qui change chaque seconde */
		int fps = 0;

		/* un ptit "flag" pour nous indiquer si l'utilisateur veut quitter */
		int user_wants_to_quit = FALSE;

		BITMAP* le_sprite;
		/* on va faire du simple double buffering ici */
		BITMAP* buffer;

		/* Sert uniquement pour repérer quand on change de seconde */
		unsigned long last_sec_counter = 0;
		/* La position du sprite est repérée par un vecteur */
```

```
        VECTOR sprite_position;


        allegro_init();
        install_keyboard();
        if (init_timer() != 0) {
            allegro_message("Erreur lors de l'initialisation des timers!");
            return 1;
        }

        set_color_depth(16);
        if (set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640,480, 0, 0) !=
0) {
            allegro_message("Impossible d'initialiser le mode vidéo!");
            return 1;
        }

        /* On charge le sprite en mémoire!
        Le bitmap est "ship105.bmp" à la racine du jeu. On ne prend pas la
palette car
        on n'est pas en mode de couleur 8 bits */
        le_sprite = load_bitmap("ship105.bmp", NULL);
        if (!le_sprite) {
            set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
            allegro_message ("Erreur : imposible de charger le bitmap!");
            return 1;
        }

        buffer = create_bitmap(SCREEN_W, SCREEN_H);
        if (!buffer) {
            set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
            allegro_message ("Erreur : imposible de créer le buffer!");
            return 1;
        }

        /* On initialise les variables */
        sprite_position.x = SCREEN_W / 2;
        sprite_position.y = SCREEN_H / 2;


        /* Boucle principale du jeu */
        while (user_wants_to_quit == FALSE) {

            while (game_time > 0) {
                /* Partie 1) et 2): les entrées puis l'update des positions
                Ici c'est très simple alors on peut tout regrouper. Méfiez vous,
si vous gérez
                des AI plus tards par exemple, les deux parties devront etre
séparées */
                if (key[KEY_LEFT])
                        sprite_position.x -= INCREMENT_X;
                if (key[KEY_RIGHT])
                        sprite_position.x += INCREMENT_X;
                if (key[KEY_UP])
                        sprite_position.y -= INCREMENT_Y;
                if (key[KEY_DOWN])
                        sprite_position.y += INCREMENT_Y;
                if (key[KEY_ESC])
                        user_wants_to_quit = TRUE;

                --game_time;
            }

            if (sec_counter != last_sec_counter) {
                fps = current_fps;
                last_sec_counter = sec_counter;
                current_fps = 0;
            }
            current_fps++;

            /* Maintenant on déssine */
            /* acquire_bitmap bitmap est inutile ici : on ne dessine qu'un sprite!
*/
            clear_bitmap(buffer);

            draw_sprite(buffer, le_sprite, sprite_position.x, sprite_position.y);
            text_mode(-1);
            textprintf_centre(buffer,     font,     SCREEN_W     /     2,     0,
makecol(195,125,255), "FPS : %d", fps);

            blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);

        }

        destroy_bitmap(le_sprite);
        destroy_bitmap(buffer);

        return 0;
}
END_OF_MAIN();
```

9. En attendant la prochaine version…

Voilà! C'est la fin de cette version "publique". Bientôt viendrons quelques nouveautés comme (dans l'ordre) :

- Le son (indispensable pour un jeu digne de ce nom).
- Les datafiles (super pratiques et super importants).

De toute façon, vous pouvez toujours vous reporter au fichier "allegro.htm" pour en savoir plus long sur le sujet. Avec ce que vous savez maintenant, vous devriez être tout à fait capable d'écrire des jeux simples.

Version en Rich Text Format / PDF uniquement pour l'instant, à cause des trop nombreux problèmes de mise en forme que j'ai rencontrés avec l'HTML...
A signaler quelques bugs de mise en forme en PDF (ceci est du à la mauvaise lecture du RTF par OpenOffice)

Si vous avez des critiques ou des précisions quelconques à apporter:
E-Mail: itmfr@yahoo.fr

Vous pouvez peut-être trouver une version plus récente de ce tutorial sur:
http://iteme.free.fr

10. Petit historique

Le 24 Août 2004 : - OpenOffice m'a bousillé pas mal d'espaces... Et puis j'ai rajouté la déclaration d'une variable dans le 8. que j'avais oubliée!
Le 28 juillet 2004 : - J'ai écrit toute la partie 8 sur les timers, j'ai remis en forme les parties de code.
- Première version PDF générée par OpenOffice (quelques bugs de mises en forme à signaler hélas...)

Au 28 Juillet 2004,
Tutorial écrit par Emeric Poupon (ITM) .