Allegro Tutorials

Introduction

Tout d'abord, je tiens à préciser qu'une connaissance correcte du C/C++ est exigée et que Allegro doit être correctement installé (on reviendra sur ce point dans un autre tutorial).

- Durant tous mes tutoriaux, les prototypes des fonctions propres à Allegro seront affichés **en caractères gras.**
- **/*Les commentaires seront en bleu-vert très caractéristique */**
- Les types en marron(int, float, …), y compris les types spécifiques d'Allegro (BITMAP, DATAFILE, ...).
- Les structures de contrôle en bleu (if, else, while, {, }, …)
- Les nombre en rouge (0 , 1, 324, 4, …)
- Les « include » et « define » en vert.
- "Les chaînes de caractère en gris"

1 . A quoi sert donc Allegro?

Plongeons directement dans le sujet : Allegro, qu'est-ce que c'est ? C'est une librairie qui fournie tout ce qui est nécessaire pour programmer un jeu vidéo. En clair, Allegro vous apporte une solution pour gérer l'affichage, le son, le clavier, la souris, les timers, … bref, tout ce dont vous avez besoin ! A l'origine, Allegro a été conçu par Shawn Hargreaves, pour Atari ST, puis rapidement pour DOS. Les premières versions de la bibliothèque datent de début 1996 : elle n'est plus tout jeune ! Rapidement, les programmeurs (contributeurs) d'Allegro ont orienté leur programmation vers une librairie multi-plateforme. Maintenant, vous pouvez utiliser Allegro sous DOS (DJGPP, Watcom), Windows (MSVC, Mingw32, Cygwin, Borland), Linux (console), Unix (X), BeOS, QNX, MacOS (MPW) . Vous l'aurez compris, la grande force d'Allegro réside dans le fait qu'elle est supportée par un grand nombre d'OS. Concrètement, vous pouvez porter et compiler vos programmes sous n'importe quel compilateur (mentionné au-dessus) sans changer une seule ligne de code. En quelque sorte, Allegro sélectionnera tout seul les bons drivers selon l'OS. Par exemple, un programme compilé sous Windows utilise l'accélération de DirectDraw, tandis que sous Linux vous pouvez profiter des drivers X11 : encore un petit bonus donc, vous pourrez profiter de l'accélération matérielle de votre carte vidéo en 2D, ce qui n'est pas négligeable. . De même pour DirectSound et DirectInput pour Windows. En revanche, la 3D n'est pas le point fort de cette librairie : aucune accélération matérielle ne sera fournie : pas de Direct3D donc ! Rassurez-vous, en revanche l'OpenGL est très bien géré, grâce à un add-on d'Allegro, AllegroGL. Signalons aussi que la librairie est gratuite et libre, donc les codes sources sont disponibles !

2.Allegro : Exemple d'application

Un jeu qui utilise Allegro peut donc être une application pour Windows, dans ce cas, DirectX 7.0 (minimum) devra être installé pour que le jeu marche. Allegro peut-être dans une DLL -alleg40.dll- ou peut être aussi statique, au contraire de la DLL : le code d'Allegro utilisé se retrouve tout simplement dans le fichier exécutable du jeu (c'est systématiquement le cas pour DOS). Je vous rassure, pour les autres OS, vous pouvez aussi choisir une compilation statique ou dynamique de vos programmes, donc ne vous en faîtes pas.

Pour donner exemple, vous avez à votre disposition la démo officielle (dans le répertoire allegro\démo) qui regroupe en un seul programme quelques caractéristiques intéressantes de la librairie.

N'oubliez pas de linker la librairie lors de la compilation de vos programmes, sinon, vous aurez une flopée d'erreur (reportez-vous au fichier d'aide disponibles pour chaque compilateur !)

N'oubliez pas non plus que vous avez à votre disposition toute une panoplie d'exemples très pratiques. C'est d'ailleurs grâce à eux que j'ai appris à me servir « proprement » d'Allegro. Toutes les descriptions des fonctions sont disponibles dans le fichier d'aide généré lors de l'installation d'Allegro. Elles sont très bien expliquées et c'est l'occasion de découvrir toutes les fonctions d'Allegro. (/docs/html/allegro.html)

Enfin, si vraiment vous tenez à découvrir le véritable potentiel de cette librairie, vous pouvez vous rendre sur le dépôt officiel des jeux utilisant Allegro : www.allegro.cc

3. Les bases d'Allegro : un premier programme

Commençons un petit programme basique qui nous servira d'exemple. Tout d'abord, il faut inclure le header de la librairie, le nom de ce header est « **allegro.h** ». Attention, pour l'instant, il n'est pas question de **WinMain** ni de <windows.h>, il faut oublier tout ce qui se rapporte à un seul OS.

```
/* On inclut le header de la librairie */
#include <allegro.h>

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

Très important, avant de faire quoi que soit avec la librairie, il faut appeler la fonction d'initialisation :

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

Parfait ! Allegro est initialisé ! Et maintenant, si on installait le clavier et la souris ? Ca serait bien plus pratique…

```
    /* Initialise le clavier */
    install_keyboard() ;
```

Si la fonction a bien initialisé le clavier, elle retourne 0, sinon, elle retourne un nombre négatif. On peut considérer que ce n'est pas la peine de vérifier le resultat, tant les chances d'échec sont minimes…

```
    /* initialise la souris */
    install_mouse() ;
```

Là, ca devient plus intéressant : si la fonction a échouée, elle retourne -1, sinon, elle retourne le

nombre de boutons de votre souris que Allegro peut gérer. Il est donc important d'effectuer le test, tous les utilsateurs de DOS n'ont pas forcément un driver de souris qui marche… On pourra donc écrire :

```
/* Si la fonction retourne un échec, alors… */
if(install_mouse() == -1) {
      /* On affiche le message d'erreur */
      allegro_message("Erreur ! %s", allegro_error) ;

      /* Et on quitte le programme ! */
      return-1 ;
}
/* Maintenant, on est sur que la souris marche ici ! */
```

Mais, vous allez me dire : qu'est-ce que **allegro_message** ? Et **allegro_error** ?
Voici le prototype de **allegro_message** :

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

Cette fonction utilise le même format de sortie que la fonction **printf**. Elle est donc très commode à utiliser. Attention, cette fonction doit être utilisée uniquement sur des modes d'affichage non graphiques ! En clair, vous ne pouvez l'utiliser que si vous n'avez pas encore initialisé le mode vidéo ou si vous êtes passé explicitement en mode texte. Par exemple, ici, on a pas encore initialisé le mode vidéo, on peut donc lancer la fonction sans problème. Sur les OS qui ont un mode texte en « console », comme DOS ou Unix, le message s'affichera normalement dans la console. Pour Windows (ou pour BeOS), cela affichera une boite de dialogue propre à l'OS, avec un bouton « OK » en bas. Elle aura pour titre le nom de l'exécutable de votre programme. Cette fonction est donc très pratique pour signaler des erreurs indépendamment de l'OS.

extern char allegro_error[ALLEGRO_ERROR_SIZE];

C'est la chaîne de caractère utilisée par quelques fonctions d'allegro, comme **set_gfx_mode** ou **install_mouse**. Elle sert à reporter les erreurs qui ont pu survenir lors de l'initialisation. Si l'utilisateur veut savoir la nature de l'erreur, la chaîne **allegro_error** contient la description du problème : il n'y a plus qu'à l'afficher (avec **allegro_message** par exemple). ALLEGRO_ERROR_SIZE, pour la petite histoire, c'est la taille de la chaîne de caractère, tout simplement.

Bon, nous avons la souris et le clavier ! C'est génial n'est-ce pas ? Seulement, ca serait mieux de passer en mode graphique… Tout d'abord, il faut définir son mode vidéo de couleur, c'est à dire que chaque pixel sera codé par 8, 15, 16, 24 ou 32 bits. Plus le nombre de bits est élevé, plus la palette de couleur disponible est large. Par exemple, pour 16 bits, vous avez droit à 2^16 = 65536 couleurs, pour 24 bits : 2^24 = 16777216 couleurs. Le cas des couleurs 8 bits est plus particulier, on le laisse de côté pour l'instant.

```
/*  On initialise le mode graphique de couleur. On se contente du mode 16 bits, ça
sera amplement suffisant pour afficher notre écran  noir… Il présente l'avantage d'être très
répandu donc facilement supportable par les cartes vidéo */
set_color_depth(16) ;
```

Maintenant, on va passer au mode graphique proprement dit. Pour cela, on va utiliser la commande :

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

Mais attention, comment diable utiliser cette fonction ? Voici son prototype :

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

On va commencer par le plus facile. La fonction retourne **0** en cas de succès, sinon, elle retourne un nombre négatif.
« int card » n'est pas très parlant, c'est l'index du mode graphique que l'on veut utiliser. Voici donc, les différentes valeurs que l'on devrait rentrer (les #define sont quelque part dans **allegro.h**) :

-GFX_AUTODETECT . On ne s'occupe de rien, on laisse Allegro prendre le meilleur driver. Il se mettra en mode fenêtré si la résolution n'est pas disponible en plein écran et si l'OS le supporte, bien sûr.
-GFX_AUTODETECT_FULLSCREEN . On force Allegro à choisir un pilote en plein écran.
-GFX_AUTODETECT_WINDOWED . On force Allegro à choisir un pilote en mode fenêtré.
-GFX_TEXT . C'est très utile pour retourner en mode texte, dans ce cas, on peut mettre 0 pour les dimensions de l'écran (c'est juste pour la lisibilité).

Il existe naturellement d'autres valeurs, mais elles sont plus spécifiques à chaque OS (on devrait donc éviter de les utiliser), on y reviendra plus tard, il s'agit pour l'instant d'ingurgiter les bases. Les valeurs **width** et **height** représentent la taille de l'écran graphique crée. Vous pouvez vérifier les dimensions de l'écran grâce aux macros **SCREEN_W** (largeur) et **SCREEN_H** (hauteur) initialisées par la fonction. Pour l'instant, on ne s'occupe pas de **v_width** et de **v_height** (on pourra donc passer la valeur zéro). Comme tout programme qui se respecte, il faut vérifier s'il n'y a pas eu d'erreur ! On va donc rajouter les tests nécessaires :

```
if(set_gfx_mode(GFX_AUTODETECT, 640, 480, 0, 0) != 0) {
    /* Comme vous avez bien suivi mes explications, vous savez que allegro_message s'utilise
    uniquement en mode texte, c'est pour cela que l'on utilise GFX_TEXT, pour être sur de repasser en mode texte… */
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);  /* voilà la commande pour le mode texte */
    allegro_message("Impossible d'initialiser le mode vidéo !\n%s\n", allegro_error);
    return 1; //et on oublie pas de quitter…
}
// Ici, tout va bien SCREEN_W = 640 et SCREEN_H = 480
```

Voilà ! On a fini d'initialiser notre petit programme ! On peut aussi rajouter du son et gérer le joystick, ca sera le sujet des prochains chapitres. Maintenant, on va faire en sorte que le programme s'interrompt dès que l'on appuie sur la touche [ESC]. On va donc rajouter à la suite quelques lignes…

```
while(!key[KEY_ESC]) { /* tant que la touche [ESC] n'est pas appuyée… */
    /* On efface l'écran. On reviendra sur cette fonction dans la prochaine section dédiée à l'affichage */
    clear_bitmap(screen) ;
}
```

Voici un aspect important de la librairie, la gestion des touches appuyées par l'utilisateur.

extern volatile char key[KEY_MAX];

C'est très simple : toutes les touches d'Allegro sont regroupées dans ce tableau (de taille KEY_MAX, vous l'avez compris). C'est un tableau où chaque touche a son index. Pour connaître la liste de toutes les touches, ouvrez le fichier « allegro/keyboard.h », tout y est défini dedans. Normalement, ce tableau représente l'état physique de la touche actuelle. C'est à dire si elle est pressée ou pas : donc, ce tableau n'est censé être fait que pour lire. Il est « modifié » tout seul grâce aux intéruptions clavier. Vous pouvez bien sur simuler l'appui sur une touche mais c'est une autre histoire… Voici d'autres exemple d'utilisation (ne les recopiez pas dans le programme, ca serait inutile ici! En effet, printf ne marche plus en mode graphique):

```
if(key[KEY_ENTER])
    printf("La touche entrée est appuyée !\n") ;
if(!key[KEY_SPACE])
    printf("La barre d'espace n'est *PAS* appuyée !\n") ;
```

N'oublions pas de quitter le programme bien sûr :

```
return 0; //et on quitte le programme
}
END_OF_MAIN() ;
```

Vous aurez remarqué que les applications Windows ont leur point d'entrée qui est **WinMain** et non pas **main**. C'est pour cela que la macro **END_OF_MAIN()** est nécessaire : elle permet d'utiliser la fonction main quelque soit le système. Placez-là directement après la fin de votre fonction **main**. Vous n'aurez aucun avertissement ni aucune erreur de la compilation, ne vous en faites pas !

Voilà pour les bases de l'initialisation d'Allegro. Normalement, si vous avez recopié et compris le programme d'exemple au fur et à mesure, il ne devrait plus y avoir de problèmes ! Bien sûr, il est très basique, il ne fait qu'afficher un écran noir (s'il n'y a pas eu une erreur d'initialisation avant) et attends qu'on appuie sur la touche [ESC] pour quitter. On va maintenant s'intéresser à une partie essentielle du jeu vidéo : l'affichage sur l'écran !

4. Affichage

4.1 Présentation générale

Nous allons maintenant passer à la partie affichage, sans doute la partie la plus importante. Pour cela, on va afficher un disque blanc de diamètre 50 pixels qui va se déplacer tout seul vers la droite de l'écran. Il sera exactement à la moitié de la hauteur de l'écran. L'utilisateur devra appuyer sur la touche [ENTER] pour lancer le disque. Modifions notre boucle d'attente qui vérifie que la touche [ESC] n'est pas appuyée :

```
/* on déclare notre variable qui représente la position du cercle. Si vous programmez en C, placez là au début de la fonction main */
double circle_pos_x = 0;


/* On attend patiemment que l'utilisateur appuie sur [ENTER] pour commencer à afficher et à bouger le disque */
```

```
        while(!key[KEY_ENTER])
            ;

        while(!key[KEY_ESC]) { //tant que la touche [ESC] n'est pas appuyée…

            /* On commence par effacer l'écran */
            clear_bitmap(screen) ;

            /* On dessine le cercle « plein » en blanc (couleur 255,255,255) */
            circlefill(screen, (int)circle_pos_x, SCREEN_H/2, 50, makecol(255,255,255));
            circle_pos_x += 0.1 ; // On décale le disque vers la droite de 0,1 pixel

            /* On vérifie si le disque sort de l'écran, dans ce cas, on sort de la boucle donc  on
quitte le programme */
            if(circle_pos_x - 50 >= SCREEN_W)
                break ;
        }
```

Ne vous en faites pas, on va éclaircir tout ça ! **circlefill** est la fonction qui nous sert à afficher sur l'écran un disque.

void circlefill(BITMAP *bmp, int x, int y, int radius, int color);
xet **y** représentent les coordonnées du centre du disque à l'écran. Toutes les coordonnées sont données par rapport au coin supérieur gauche de l'écran. **radius** représente tout simplement le rayon en pixel. Comme vous le constatez, il y a un nouveau type présenté ici, il s'agit de **BITMAP**. Ne vous en faites pas, on en parlera juste après.

extern BITMAP *screen;
Retenez juste que **screen** est une variable globale qui pointe vers un **BITMAP** (une zone image) qui représente la mémoire vidéo, donc qui pointe directement vers l'écran, la taille de ce bitmap est donc SCREEN_W * SCREEN_H.
bmpest donc le bitmap dans lequel on va dessiner notre disque. C'est la destination de la fonction de dessin. **color** est la couleur, comme son nom l'indique. Le mieux est d'employer une petite fonction pour la définir.

int makecol(int r, int g, int b);
Cette fonction permet de définir une couleur indépendamment du mode couleur utilisé. On peut appeler cette fonction dans des modes de fonctionnement de 8, 15, 16, 24 et 32 bits/pixel. **r** représente la composante rouge, **g** la composante verte, **b** la composante bleue. On dit aussi un format de couleur **RGB**. Ces composantes peuvent prendre comme valeur de 0 à 255 compris. makecol(255,255,255) renvoie donc le code de la couleur blanche dans le mode de couleur courant.

Maintenant, si vous lancez le programme, vous vous apercevrez qu'il y a d'horribles scintillements à l'écran. En effet, on ne voit pas un disque uni mais une succession de ligne noires et blanches. Ce n'est pas vraiment l'effet désiré ! Alors, ou est le problème ? C'est très simple, pour programmer correctement un jeu, on ne doit **jamais** écrire directement sur l'écran, sauf si on se fiche totalement du résultat, mais ce n'est pas notre cas. On va donc utiliser une nouvelle méthode d'affichage, le **double buffering** ! Le principe est très basique : au lieu de dessiner sur l'écran, on va dessiner dans un **bitmap** que l'on aura

préalablement placé en RAM. Ensuite, à la fin de notre boucle, on va copier d'un coup tout le contenu de ce bitmap (on parle de **buffer**) dans l'écran. Ce n'est pas la méthode qui donne les résultats les plus spectaculaires, mais ce sera déjà nettement mieux ! On va définir une bonne fois pour toute a quoi ressemble ce type **BITMAP** (vous pouvez retrouver la déclaration dans les headers de la librairie) :

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

Il existe d'autres variables dans le type BITMAP, mais on n'en aura pas besoin. Ainsi, pour connaître la taille de l'écran, il existe une autre méthode, screen->w représente la taille en largeur, screen->h en hauteur. N'oubliez jamais qu'Allegro ne travaille qu'avec des pointeurs vers **BITMAP** (BITMAP*). On va donc déclarer notre buffer vidéo :

```
BITMAP* buffer ; /* C'est la variable qui pointe vers le buffer vidéo ! */
```

Maintenant, il va falloir créer un bitmap qui aura les mêmes dimension que l'écran. Il est donc nécessaire de le créer **après** avoir initialisé le mode vidéo.

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

BITMAP* create_bitmap(int width, int height);
Cette fonction crée un bitmap de largeur **width** et de hauteur **height**. Elle retourne un pointeur vers le bitmap crée. Normalement, cette image crée dans la RAM n'est pas complètement vide (noire), il doit y rester des résidus. Il faut donc la vider avant de l'utiliser. Pour vider un bitmap à la couleur 0 (noir), on utilise la fonction:

void clear_bitmap(BITMAP *bitmap);
Cette fonction peut être accélérée matériellement, le résultat sera un incroyable gain de performance.

Ici, plus concrètement :

```
clear_bitmap(buffer) ; /* Tout simplement… */
```

On va donc reécrire une nouvelle fois notre boucle d'attente :

```
while(!key[KEY_ESC]) { //tant que la touche [ESC] n'est pas appuyée…

    /* On commence par effacer le buffer */
    clear_bitmap(buffer) ;

    /* On dessine le cercle « plein » en blanc (couleur 255,255,255)  dans le buffer */
    circlefill(buffer, (int)circle_pos_x, SCREEN_H/2, 50, makecol(255,255,255));

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

    /* On vérifie si le disque sort de l'écran, dans ce cas, on sort de la boucle donc  on
quitte le programme */
```

```
        if(circle_pos_x – 50 >= SCREEN_W)
            break ;

        /* Maintenant, il faut copier le contenu de notre buffer dans l'écran */
        blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H) ;
    }
```

On a donc remplacé **screen** par **buffer** dans les fonctions nécessaires. Il ne reste plus que la fonction **blit** à découvrir… Cette fonction permet de copier une zone d'un bitmap source vers une zone du bitmap de destination.

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

Voilà une fonction qui commence à prendre quelques paramètres intéressant ! **source** est bien entendu le bitmap source (ici **buffer**), **dest** est le bitmap de destination (ici **screen**). **source_x** et **source_y** représentent les coordonnées de l'origine dans le bitmap que l'on devra copier. **dest_x** et **dest_y** représentent quand à eux les coordonnées à partir desquelles on commence à dessiner dans le bitmap de destination. Enfin, **width** et **height** représentent les dimensions de ce que l'on veut copier.

Dans notre exemple, on veut recopier l'intégralité du buffer dans l'écran. C'est pour cela que l'on commence à dessiner aux coordonnées (0,0) et que l'on dessine tout le buffer (SCREEN_W et SCREEN_H). Lancez le nouveau programme, vous serez surpris ! Plus de scintillements rebelles ! Les choses se présentent bien… En revanche la vitesse de déplacement du disque risque d'être beaucoup plus lente ! C'est tout à fait normal, car sachez que l'ordinateur perd la plupart de son temps à afficher. Afficher un petit disque à l'écran est bien plus rapide que de copier tout un buffer sur l'écran, il est donc préférable de changer la ligne de déplacement par :

```
++circle_pos_x;  /* On décale le disque vers la droite de 1 pixel cette fois */
```

La vitesse du disque dépend ici directement de la vitesse de votre ordinateur (bien que l'affichage soit un facteur très limitatif de la vitesse du programme). Même sur un ordinateur surpuissant, vous ne pourrez atteindre des vitesses astronomiques (en terme d'images par seconde) car il faut attendre à chaque image que la carte vidéo aie fini de dessiner ! Rassurez vous, on s'occupera dans les prochaines sections du **temps réel**, c'est à dire le fait que votre jeu tourne à la même vitesse quelque soit la puissance de l'ordinateur sur lequel il tourne.

4.2 Affichage de textes

Une des fonctionnalités « génialissimes » d'Allegro est l'affichage de texte à l'écran, et ce, de façon très simple ! En effet, en mode graphique, il n'est absolument plus question d'utiliser **printf** et les autres fonctions du même type. Allegro nous fourni une panoplie de fonctions permettant d'effectuer ces taches de façon automatique. Modifions une fois encore la boucle principale de notre petit programme pour qu'il affiche la résolution actuelle de l'écran et la position de notre petit disque blanc :

```
while(!key[KEY_ESC]) { /* Tant que la touche [ESC] n'est pas appuyée… */

    /* On commence par effacer le buffer */
    clear_bitmap(buffer) ;
```

```
		/* On affiche une chaine de caractère aux coordonnées 0,0 et pour la couleur, on utilisera un bleu - violet vif  (150,150,255) */
		textout(buffer, font, "Je suis en train d'ecrire du texte !", 0, 0, makecol(150, 150, 255));

		/* On dessine le disque en blanc (couleur 255,255,255)  dans le buffer */
		circlefill(buffer, (int)circle_pos_x, SCREEN_H/2, 50, makecol(255,255,255));

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

		/* On vérifie si le disque sort de l'écran, dans ce cas, on sort de la boucle donc  on quitte le programme */
		if(circle_pos_x – 50 >= SCREEN_W)
			break ;

		/* Maintenant, il faut copier le contenu de notre buffer dans l'écran */
		blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H) ;
	}
```

On a utilisé la fonction **textout** pour afficher :

void textout(BITMAP *bmp, const FONT *f, const char *str, int x, int y, int color);
bmpreprésente le bitmap dans lequel on va afficher le texte. **str** sera la chaîne de caractère à afficher, **x** et **y** seront les coordonnées du point supérieur gauche où sera affiché le texte (0 et 0 nous assurent donc de coller le texte dans le coin de l'écran). **color**, comme nous l'avons déjà vu, représente la couleur d'affichage de **str**. Il reste juste à introduire la notion de **FONT**. Comme sous Windows, Allegro est capable d'afficher plusieurs polices de caractères. Il suffit juste de lui spécifier un pointeur vers la police désirée, qui devra être de type **FONT**. Pour charger des **fonts**, il faut passer par le biais de bases de données : ce sera l'objet de futurs chapitres. Heureusement, on peut utiliser la police par défaut du BIOS. Elle est représentée ainsi :

```
extern FONT *font;
```

Cela dit en passant, on peut normalement modifier ce pointeur vers n'importe quelle autre font. Mais ce n'est pas très utile pour l'instant, vu qu'on ne sait pas encore les charger.

Maintenant, nous pouvons afficher la taille de l'écran à l'écran ! Il suffit de rajouter :

```
#include <string.h> /* on inclut le header de la librairie standard du C ANSI. */
```

On aura besoin d'une nouvelle variable :

```
/* Voici un petit buffer pouvant contenir 256 caractères */
char str_buf[256] ;
```

Transformons notre fonction **textout** :

```
/* On copie d'abord la taille de l'écran dans str_buf */
sprintf(str_buf, "Voici la taille de l'ecran : %d*%d", SCREEN_W, SCREEN_H);
```

```
/* Le tour est joué, on a plus qu'à remplacer notre nouvelle chaîne… */
textout(buffer, font, str_buf, 0, 0, makecol(150, 150, 255));
```

Admirez le résultat ! Vous obtenez à l'écran en haut à gauche : « Voici la taille de l'ecran : 640*480 ». Mais les créateurs d'Allegro ont pensé à tout ! Pour vous épargner une ligne, ils ont pensé à utiliser textout avec la syntaxe de printf. On peut donc maintenant simplifier notre programme :

```
/* On va changer de couleur pour l'occasion… */
        textprintf(buffer, font, 0, 0, makecol(100, 255, 100), "Taille de l'ecran : %d*%d",
SCREEN_W, SCREEN_H);
```

void textprintf(BITMAP *bmp, const FONT *f, int x, y, color,const char *fmt, ...);
Voilà le prototype. Vous pouvez maintenant afficher du texte formaté, comme avec printf ! La fonction est similaire à textout, pas besoin de revenir sur les détails donc. Mais du coup, on peut se débarrasser de la ligne « #include <string.h> », n'oubliez pas non plus d'enlever la déclaration de notre buffer de caractères, désormais inutilisé.

Pourquoi ne pas afficher la position du disque sur l'écran ? Mais la couleur pourrait varier en fonction de la position actuelle du disque…

```
/* Voici la nouvelle commande, un peu plus longue à taper */
        textprintf(buffer, font, 0, 10, makecol(circle_pos_x / SCREEN_W * 255,
circle_pos_x / SCREEN_W * 255, circle_pos_x / SCREEN_W * 255), "Disque_x : %d",
(int)circle_pos_x);
```

Désormais, la chaîne de caractère s'affichera avec une luminosité progressive, ce qui crée un effet visuel sympathique. Dans les 50 derniers pixels, la chaîne « disparaît » ! En fait, c'est à cause de la variable circle_pos_x qui prend une valeur supérieure à 255. Le programme « repasse » donc l'excédent à 0, d'où, une couleur très sombre. Mais pour aller un peu plus loin, il existe d'autres fonctions tout à fait similaires à **textprintf** :

void textprintf_centre(BITMAP *bmp, const FONT *f, int x, y, color,const char *fmt, ...);
Cette fonction réalise exactement la même chose que **textprintf** sauf qu'elle interprète **x** et **y** comme le centre de la chaîne à afficher plutôt que le coin supérieur gauche.

void textprintf_right(BITMAP *bmp, const FONT *f, int x, y, color,const char *fmt, ...);
De même ici sauf que **x** et **y** seront les coordonnées du coin supérieur droit.

void textprintf_justify(BITMAP *bmp, const FONT *f, int x1, int x2, int y,int diff, int color, const char *fmt, ...);
Ici, on dessine le texte de façon justifiée entre les coordonnées **x1** et **x2**. Si par hasard les lettres se chevauchent, la fonction repasse à la fonction standard **textprintf**.

Voilà, vous connaissez toutes les fonctions d'Allegro basée sur **textprintf**. Il existe quelques fonctions annexes qui peuvent être bien pratiques dans certains cas. Par exemple :

int text_length(const FONT *f, const char *str);
Cette fonction retourne la longueur en pixel de la largeur de la chaîne de caractère **str**, telle qu'elle sera à l'écran avec la police **f**. Ca peut être très pratique pour savoir si une chaîne affichée va sortir de l'écran par

exemple.

int text_height(const FONT *f)

Par contre ici, nul besoin de spécifier une chaîne de caractère puisqu'on s'intéresse qu'à la hauteur. Il suffit donc de donner la police **f** pour récupérer la taille en pixel de la hauteur de la police telle qu'elle sera affichée à l'écran.

Nous allons maintenant utiliser la fonction **textprintf_centre** dans notre petit programme d'exemple, nous allons donc remplacer la ligne qui contrôle l'affichage des dimensions de l'écran par celle-ci :

```
/* On affiche les dimensions exactement au centre de l'écran */
        textprintf_centre(buffer, font, SCREEN_W / 2, SCREEN_H / 2, makecol(100,
255, 100), "Taille de l'ecran : %d*%d", SCREEN_W, SCREEN_H);
```

Lancez le programme… Si vous avez bien suivi mes instructions, le disque devrait s'afficher par dessus le texte. Ce qui est tout à fait logique puisque vous dessinez le disque **après** avoir dessiner le texte. Le disque est donc bien supperposé. Maintenant, déplacez la ligne précédente après celle qui commande l'affichage du disque.

Relancez donc le programme. Vous constatez cette fois que le texte reste bien « par dessus » le disque mais il semble être dans une drôle de boite noire ! Je vous rassure tout de suite, ceci est parfaitement normal. Modifions ça immédiatement… Pour cela, voici les nouvelles lignes à écrire :

```
/* On passe au mode d'affichage du texte par transparence */
text_mode(-1) ;

/* On affiche les dimensions exactement au centre de l'écran */
        textprintf_centre(buffer, font, SCREEN_W / 2, SCREEN_H / 2, makecol(100,
255, 100), "Taille de l'ecran : %d*%d", SCREEN_W, SCREEN_H);
```

Le résultat est vraiment parfait : Plus de cadre noir ! Mais comment cela ce fait-il ? Et bien, tout texte affiché se compose de deux parties : la partie frontale, le texte en lui même, et la partie arrière, qui représente une boite entourant la chaine de caractère affichée. Par défaut, cette boite est noire, c'est pour cela que vous ne vous en êtes pas rendu compte (j'ai bien magouillé mon coup pas vrai ?), mais par contre, on la voit très bien si le texte est affiché sur une couleur autre que la couleur de cette « boite ». Voyons la fonction **text_mode** :

int text_mode(int mode);

Elle est vraiment très simple à utiliser. **mode** est en fait la couleur de la boite sur laquelle le teste sera affiché. On peut donc la créer avec **makecol**, comme nous l'avons vu. Par exemple, pour afficher par dessus une boite blanche :

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

Petite particularité cependant. Pour ne pas dessiner cette « boite », c'est-à-dire pour obtenir un effet de transparence, il faut utiliser la couleur **–1**. Cependant, faites attention, **text_mode** est effective pour toutes les fonctions d'affichage de texte à venir ! Donc, il faut redéfinir la couleur de fond à chaque fois que vous en avez besoin. Par exemple, dans notre programme, dès la deuxième occurrence de la boucle, tous les

textes sont affichés en transparence !

Bon, vous avez maintenant tous les outils en main pour afficher du texte sur l'écran pendant vos programmes. N'hésitez pas à essayer toutes les fonctions, rien ne vaut la pratique et l'expérience personnelle pour assimiler leur fonctionnement.

Nous avons affiché un disque… Ne serait-il pas mieux d'afficher une vraie **image** (on parle alors de **sprite**)?

4.3 Affichage de sprites

Effectivement, il est tout à fait possible de charger une image et de l'afficher n'importe où à l'écran. C'est la base même d'un jeu en 2D. Un **sprite** n'est en fait rien d'autre qu'un **bitmap,** comme notre **buffer** qui nous sert pour l 'affichage, sauf que cette fois-ci, il faut initialiser son contenu. On ne doit normalement pas modifier directement un **sprite** déjà chargé en mémoire, il ne doit être que recopié sur l'écran avec pourquoi pas un mode d'affichage particulier (comme la transparence, ou en utilisant une couleur dominante). Selectionnez donc une belle image de taille moyenne (par exemple 320*200), au cas ou redimensionnez là, au moins pour qu'elle rentre dans l'écran ! L'image ne doit **impérativement** pas être enregistrée en 8 bpp, on étudiera ce cas plus tard. Nous allons donc modifier une fois de plus notre programme ! Nous allons remplacer le disque par un sprite. Tout d'abord, on va charger l'image et renommer la variable **circle_pos_x** (pour plus de lisibilité). On considère que votre **sprite** est nommé « sprite.bmp » et qu'il est placé dans le même répertoire que celui de votre executable.

```
/* on déclare notre variable qui représente la position de l'image. Si vous programmez en C, placez là au début de la fonction main */
double sprite_pos_x = 0;

/* Maintenant, on déclare notre sprite de type pointeur vers BITMAP. Attention, il s'agit bien d'un pointeur ! Faites attention de ne pas oublier « * » */
BITMAP* sprite1 ;

/* Placez ces lignes après l'initialisation du mode vidéo ! */
sprite1 = load_bitmap("sprite.bmp", NULL) ;
/* Maintenant, on doit vérifier si le bitmap chargé est valide !  */
if(!sprite1) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("Erreur ! Impossible de lire le fichier image !");
    return1;
}
```

Pourquoi placer le chargement de l'image après l'initialisation du mode vidéo ? C'est tout simplement que l'image sera chargée en fonction du mode vidéo initialisé ! Si vous lancez le programme maintenant, rien ne devrait être changé. Si vous avez l'erreur du mauvais chargement de l'image, vérifiez l'orthographe de la chaîne de caractère et si l'image est réellement présente dans le bon répertoire. Pas de nouveauté donc à part la fonction **load_bitmap** :

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

C'est vraiment une fonction importante ! Tout d'abord, elle retourne NULL en cas d'erreur, sinon, elle renvoie un pointeur vers **BITMAP**, bitmap qui sera crée à partir de la chaîne **filename**. Comme nous le

savons déjà, Allegro gère les modes de couleurs 8 bpp (Bits Per Pixel). mais ces modes sont particuliers car ils nécessitent l'emploi de palette de couleurs. On se s'occupe donc pas de **pal** pour le moment. Sachez juste que pour charger une image en mode **truecolor** (c'est à dire 15, 16, 24 ou 32 bits), on passe NULL comme valeur pour **pal**. Le pointeur retourné pointe vers une zone de donnée qui a été crée. Il ne faut donc pas oublier de libérer cet espace avant de quitter le programme. On s'en occupera juste après… Pour l'instant, cette fonction supporte les fichiers BMP, LBM, PCX et TGA. Ces différents types sont déterminés par leur extension. Les sprites ont une grande particularité. Ils possèdent ce qu'on apelle une couleur de masque (**mask color**). Lorsque vous affichez un sprite quel qu'il soit, il sera toujours affiché sous une forme parfaitement rectangulaire. Imaginons un instant que vous voulez afficher une bille rouge par dessus une image représentant une surface de gazon. La bille rouge sera donc stockée séparément de la texture du gazon. Certes, votre sprite de bille contiendra une bille rouge centrée dans le sprite. Mais autour ? Comme c'est un rectangle, on mettra par exemple du noir. On lance le jeu et horreur ! On s'aperçoit que la bille n'est pas correctement affichée sur le gazon ! Elle est affichée dans un rectangle (ou un carré) tout noir ! Il faudrait se débarrasser de la « boite » située derrière la bille. C'est exactement le même problème que pour l'affichage du texte (vous vous souvenez j'espère ?). C'est là qu'intervient notre couleur de masque. En mode **truecolor**, tous les pixels contenant la couleur (255, 0, 255) seront totalement ignorés ! Remplacez le noir entourant la bille par cette couleur (rose éclatant) et vous obtiendrez le résultat souhaité. C'est aussi simple que cela.

Maintenant, on va modifier une nouvelle fois notre boucle infinie contrôlée par la touche [ESC]. Voici la nouvelle version :

```
        /* On passe au mode d'affichage du texte par transparence */
        text_mode(-1) ;

        while(!key[KEY_ESC]) { /* Tant que la touche [ESC] n'est pas appuyée… */

            /* On commence par effacer le buffer */
            clear_bitmap(buffer) ;

            /* On affiche le sprite au milieu de l'écran dans la hauteur */
            draw_sprite(buffer, sprite1, (int)sprite_pos_x - sprite1->w/2, SCREEN_H/2 -
sprite1->h/2 );

            /* On affiche les dimensions exactement au centre de l'écran */
            textprintf_centre(buffer, font, SCREEN_W / 2, SCREEN_H / 2, makecol(100, 255,
100), "Taille de l'ecran : %d*%d", SCREEN_W, SCREEN_H);

            /* On affiche les coordonnées du sprite */
            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);

            /* On décale le sprite vers la droite de 1 pixel */
            ++sprite_pos_x ;

            /* On vérifie si le sprite sort de l'écran, dans ce cas, on quitte le programme */
            if(sprite_pos_x - sprite1->w / 2 >= SCREEN_W)
                break ;
```

```
        /* Maintenant, il faut copier le contenu de notre buffer dans l'écran */
        blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H) ;
}
```

Lancez votre nouveau programme… Si le programme est lent, ne vous en faites pas, augmentez juste la valeur qui sert à déplacer le sprite vers la droite. Là non plus rien de bien nouveau, c'est aussi l'occasion de faire le point sur le programme en cours. J'ai placé la fonction **text_mode** avant la boucle car on ne se sert que d'un seul mode, le mode de transparence, inutile donc de la rappeler sans arrêt. N'oubliez jamais que ce que vous affichez en premier sur le buffer sera en arrière plan et que l'ordre des plans est déterminé par l'ordre d'affichage dans le programme. Notre sprite étant de type **BITMAP**, on peut tout à fait récupérer ses dimensions, ce qui est fort pratique pour afficher le sprite de façon centrée.

```
/* Largeur du sprite */
sprite1->w;
/* Hauteur du sprite */
sprite1->h;
```

Si vous ne comprenez toujours pas pourquoi le sprite est centré, faites un schéma et reportez les valeurs que vous connaissez (la taille de l'écran, du sprite), ca ira beaucoup mieux. Voyons tout de même la fonction qui nous sert à afficher notre fameux sprite !

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

Vous voyez qu'elle est assez basique. **bmp** représente la destination (l'image dans laquelle le sprite sera affiché). **sprite** est le pointeur vers le sprite (ca porte bien son nom). **x** et **y** sont simplement les coordonnées du coin supérieur gauche où le sprite sera affiché. Cette fonction va passer tous les pixels marqués par la couleur de masque. Cette fonction est à peu près la même que « blit(sprite, bmp, 0, 0, x, y, sprite->w, sprite->h) », à part que **blit** copie tous les pixels sans exceptions. Ces fonctions de dessin peuvent être accélérée matériellement si le driver vidéo le permet. La fonction **draw_sprite** est la base de toutes les autres fonctions de d'affichage de sprite, comme nous allons le voir.

Il existe d'autre façon de dessiner un sprite. Vous voulez par exemple qu'il soit affiché avec un **flip vertical** (c'est un effet de miroir vertical) ? Et même un **flip horizontal** ? Ou pourquoi pas les deux en même temps ?

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);

Elles marchent exactement comme **draw_sprite**, sauf que les fonctions renversent l'image verticalement, horizontalement, ou les deux à la fois. Les images sont exactement des images « miroir ». Ce n'est pas comme si on effectuait une simple rotation de l'image. La fonction de rotation existe aussi, mais reste beaucoup plus lente. La voici :

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

Tiens donc, comme on pouvait s'en douter, cette fonction a besoin d'un argument supplémentaire, l'angle. Sauf que cette fois, le type n'est pas **int** ou **float** mais **fixed**. **fixed** est une nouveau type que définit Allegro. Ce n'est pas très important de tout savoir à son propos, on y reviendra plus en détails plus tard. Sachez simplement que ce type est codé sur 32 bits. Le premier mot pour la partie entière et le dernier

pour la partie flottante. En clair, c'est un **int** géré de telle sorte qu'on puisse y placer des nombres à virgules. Ses valeurs peuvent donc varier entre –32768 à 32767. On va plutôt s'intéresser à la conversion entre les types communs et le type **fixed**.

fixed itofix(int x);
Convertit un **int** en **fixed**. C'est la même chose qu'écrire x<<16.

int fixtoi(fixed x);
A l'inverse, convertit un **fixed** en **int**.

fixed ftofix(float x);
Convertit un **float** en **fixed**.

float fixtof(fixed x);
Et enfin, comme on aurait pu s'en douter, convertit un **fixed** en **float**.

Voilà, vous savez maintenant vraiment l'essentiel de ce nouveau type pour manipuler aisément la fonction **rotate_sprite**. N'oublions pas de dire que l'angle doit être compris entre 0 et 256. 256 représente alors une rotation complète, 64 un quart de cercle, et ainsi de suite… Cette fonction, tout comme **draw_sprite** passera automatiquement les pixels marqués par la couleur de masque, elle est donc très pratique pour les jeux ou les superpositions sont omniprésentes, bien qu'elle soit relativement lente…

On peut s'amuser une fois de plus à modifier le programme pour illustrer les nouveautés ! On va donc faire en sorte d'effectuer une rotation à l'image en même temps qu'elle se déplace, et l'angle variera en fonction de la distance parcourue depuis le point de départ. On a donc juste besoin de changer la ligne qui commande l'affichage de notre sprite :

```
/* On affiche le sprite au milieu de l'écran dans la hauteur, et on effectue une rotation
*/
        rotate_sprite(buffer, sprite1, (int)sprite_pos_x - sprite1->w/2, SCREEN_H/2 -
sprite1->h/2, ftofix(sprite_pos_x));
```

Admirez le résultat… Ca vous plaît au moins? Si vous trouvez que le sprite avance trop vite, modifiez directement **sprite_pos_x**. On peut donc se permettre de dépasser la valeur 256, 257 devenant 1, 258 devenant 2, etc… Le tout est de savoir ce qu'on fait. Par ailleurs, je profite de l'occasion pour vous présenter la macro **ABS(x)** d'Allegro. Cette macro permet de remplacer le nombre **x** par sa valeur absolue, et ce, quelque soit son type ! On aurait très bien pu écrire dans notre programme pour effectuer une rotation dans l'autre sens (valable pour le premier tour seulement) :

```
/* On affiche le sprite au milieu de l'écran dans la hauteur, et on effectue une rotation
*/
        rotate_sprite(buffer, sprite1, (int)sprite_pos_x - sprite1->w/2, SCREEN_H/2 -
sprite1->h/2, ftofix(ABS(256 - sprite_pos_x)));
```

Un dernier point très important : avant de quitter votre programme, vous devez absolument libérer la mémoire allouée par **load_bitmap()**. En effet, cette fonction copie le contenu du bitmap en RAM, il faut donc libérer cet espace mémoire. Pour cela, une seule fonction à utiliser, il s'agit de :

void destroy_bitmap(BITMAP *bmp);

La fin du programme précédent est donc facile à écrire maintenant :

```
    /* On libère la RAM du sprite */
    destroy_bitmap(sprite1) ;

    /* On libère la RAM du buffer*/
    destroy_bitmap(buffer) ;

    /* Fin de la fonction main*/
    return 0 ;
}
END_OF_MAIN() ;
```

4.4 Différentes méthodes d'affichage...

On m'a demandé de parler des différentes méthodes d'affichage, il est grand temps qu'on s'y mette en effet! Je vous rassure cependant, Allegro est tellement simple à utiliser que la gestion des différents modes d'affichage ne requiert pas de compétence particulière. Comme je vous l'ai déjà dit, la méthode que je vous ai montrée n'est ni plus ni moins que le **double buffering**. On va maintenant s'interesser à un mode plus répendu et aussi plus performant, le page flipping. On verra aussi le triple buffering, qui lorsqu'il est bien géré donne des résultats excellents. Le tout est de faire des tests entre les 3 méthodes pour voir laquelle est la plus efficace chez vous. Mais ne perdez jamais de vu que les performances de votre jeu vont dépendre de la configuration matérielle de votre ordinateur. Ainsi, telle méthode sera plus efficace sur tel ordinateur mais en revanche sera minable sur un autre, et vice-versa... A mon avis, le mieux en fait est de laisser le choix à votre utilisateur. J'ai toujours procédé ainsi et je pense que c'est une très bonne solution pour contenter tout le monde. Bon, assez bavardé, on va passer enfin aux choses sérieuses ! Ici, on va s'atteler à l'initialisation du page flipping, qui comme vous allez le voir, n'a rien de bien compliqué :

4.4.1 Le Page Flipping

```
    /* On déclare les deux buffers dont on va avoir besoin et le buffer qui pointera vers un des deux autres... */
    BITMAP* page1, page2, buffer;
    int current_page = 1;

    set_color_depth(16);

    /* Comme d'hab', on initialise d'abord le mode vidéo */
    if (set_gfx_mode(GFX_AUTODETECT_FULLSCREEN, 640, 480, 0, 0)  != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        return 1;
    }

    /* Maintenant, c'est au tour de l'initialisation du page flipping... */
```

```
page1 = create_video_bitmap(SCREEN_W, SCREEN_H);
page2 = create_video_bitmap(SCREEN_W, SCREEN_H);

/* Ici, on vérifie que les DEUX pages sont bien créés */
if ((!page1) || (!page2)) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message ("Pas assez de mémoire pour faire du double buffering !");
    return 1;
}
```

Et voilà pour l'initialisation! On va bien sur expliquer tout ça. Alors, quoi qu'il arrive, il faut comme d'habitude initialiser son mode vidéo en premier, c'est obligatoire. Obligatoire aussi de vérifier le résultat, cela va sans dire... Un page flipping digne de ce nom repose sur deux buffers vidéo placés directement dans la mémoire vidéo, au contraire du mode que nous avons déjà vu qui ne fait que créer un buffer dans la RAM. Et ceci afin de bénéficier d'une accélération matérielle pour un affichage direct de la mémoire vidéo vers cette même mémoire! En effet, la variable **screen** pointe bel et bien sur un emplacement en mémoire vidéo! Il faut donc utiliser une fonction spéciale pour créer un bitmap en mémoire vidéo :

BITMAP* create_video_bitmap(int width, int height);

Cette fonction crée un bitmap de largeur **width** et de hauteur **height**. Elle retourne un pointeur vers le bitmap crée. Normalement, cette image crée dans la **mémoire vidéo** n'est pas complètement vide (noire), il doit y rester des résidus. Il faut donc la vider avant de l'utiliser. Vous voyez, je me suis pas foulé, j'ai recopié ce que j'avais déjà écrit pour la fonction **create_bitmap**, tellement la modification est simple... Cependant, tout n'est pas rose dans la mémoire vidéo. En effet, bien que les cartes vidéos récentes embarquent pas mal de mémoire, il existe encore beaucoup de cartes vidéo dont les capacités sont très limitées. De plus, certains OS ont du mal à détecter le maximum de mémoire disponible avec certaines cartes vidéo -ça m'est hélas arrivé sous BeOS avec une GeForce2MX-, ce qui rend l'utilisation d'un tel procédé très délicat. Je m'explique : deux buffers crées en 1024*768*32 bits prennent tout de même plus de trois mégaoctects de mémoire vidéo... Bien sur vous pouvez toujours essayer de copier un maximum de graphismes de votre jeu en mémoire vidéo, afin de gagner un maximum de vitesse, mais sachez juste que de grandes incompatibilités peuvent en découler. De plus, je ne crois pas que les résultats soient toujours garantis, c'est à dire que le jeu pourrait être plus lent sur certaines vieilles configurations -En fait, il faut que la carte vidéo supporte matériellement la copie de mémoire vidéo à mémoire vidéo, sinon le page flipping en lui même est totalement inefficace en plus des blitting de sprites...-. Pour résumé le tout, vous n'avez à priori que les deux buffers vidéo à placer en mémoire vidéo si vous voulez faire du page flipping; je pense qu'il est inutile d'essayer de copier quoi que ce soit d'autre dedans, à moins que vous ne possédiez de bonnes raisons de le faire bien entendu.

Le page flipping repose sur un principe très simple : on dessine sur une page, on affiche cette page, on dessine sur l'autre puis on affiche l'autre et ainsi de suite... On alterne en fait le dessin de la scène du jeu entre les deux pages, c'est tout! Voilà comment ça va se traduire si on reprend notre exemple du disque qui bouge de gauche à droite.

```
while (key[KEY_ESC]) {

    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 */
        show_video_bitmap(buffer);

        /* Ici, on s'occupe de l'alternance des deux buffers */
        if (current_page == 1) {
                current_page = 2;
                buffer = page2;
        }
        else  {
                current_page = 1;
                buffer = page1;
        }
}
```

Voilà qui est fait! Rajoutez les deux lignes pour détruires les deux bitmaps, et bien sur le END_OF_MAIN();. Executez le programme. Si vous avez une configuration qui tient la route, vous devriez obtenir un resultat nettement moins saccadé que dans le tout premier exemple qui utilisait le double buffering! Une seule nouvelle fonction, on va voir ce qu'elle fait exactement :

int show_video_bitmap(BITMAP *bitmap);

Autant casser le suspense tout de suite, cette fonction porte très bien son nom : elle essaye de faire un "page flip" du **screen** vers le **bitmap** passé en paramètre. Ce bitmap doit donc faire exactement la taille de l'écran et doit avoir été créé par **create_video_bitmap** sinon gros ennuis en perspective. Histoire de dire, cette fonction retourne zero si elle a réussi sinon n'importe quoi d'autre. Vous voyez, rien de bien compliqué encore. Sachez juste que **show_video_bitmap(buffer)** est l'équivalent de **blit (buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H)** pour le page flipping avec les restrictions indiquées. Il est bon cependant d'aborder le problème de la synchronisation verticale ici... Ah, je vous vois perplexe, je vais vous raconter tout ça.

Normalement, la carte vidéo affiche ce qu'elle contient à une certaine adresse -ici cela concerne la variable globale **screen**- à une certaine fréquence. Par exemple, vous copiez un écran complet rouge sur le bitmap **screen**. Là, si votre écran est par en 75Hz, votre carte vidéo va envoyer exactement 75 fois ce qu'il y à l'adresse **screen** sur l'écran. En passant, plus le raffraichissement est élevé, meilleur est le confort pour vos yeux. C'est là que les complications arrivent. Si le programme n'attend pas que la carte vidéo finisse de dessiner son image sur l'écran, le contenu de **screen** va changer en plein milieu du dessin sur l'écran -c'est votre jeu qui bien sur continuera d'essayer d'afficher ses images en continu-. Si ce phénomène se produit tout le temps, cela explique pourquoi on obtient des petits déchirement lors des scrollings -l'exemple ici du disque qui se déplace est frappant-. Or, la fonction **show_video_bitmap** attend patiemment que la carte vidéo finisse de dessiner avant de changer le contenu de la mémoire vidéo prette à être affichée! Vous pouvez faire le test plus tard : vous comptez le nombre d'images par seconde dans votre jeu que vous affichez quelque part : il sera identique au taux de raffraichissement de l'écran! Il faut pour cela bien sur que l'ordinateur soit assez puissant, car si ce n'est pas le cas, cet indice va s'effondrer et on aura perdu car le programme ne pourra pas suivre la "cadence" imposée par le taux de rafraichissement

du moniteur. Revenons au double buffering, la fonction finale qui nous sert à copier le contenu du buffer sur l'écran se moque de s'avoir où en est le dessin et modifie la mémore vidéo sans aucun scrupule. Mais il existe une fonction qui corrige cela :

void vsync(void);

Cette fonction attend en fait que l'image sur le moniteur soit completement dessinée. A la fin de ce dessin, le faisceau d'electrons passe d'en bas à droite de l'écran à sa position de départ, c'est à dire en haut à gauche. Pendant ce laps de temps, la carte vidéo n'envoie rien à l'écran puisque ce dernier ne peut rien faire d'autre que de se replacer pour dessiner la prochaine image. C'est la que ça devient interessant pour vous :

Pour le mode double buffering simple, essayez de placer un **vsync()** juste avant d'appeler la fonction qui "blit" le resultat final à l'écran. Là, c'est toujours la même chose, si votre ordi est rapide, la limitation effectuée va vous permettre d'obtenir autant de FPS -Frames per second, ou images par seconde- que la valeur de votre taux de raffraichissement de l'écran. Si l'ordi est trop lent, même chanson, les performances s'écroulent. C'est pour ça que je vous ai conseillé de toujours proposer différentes methodes d'affichages dans vos jeu...

J'ai fini la partie sur le double buffering et le page flipping. Amusez vous à faire des test, y compris avec les jeux déjà programmés avec Allegro qui proposent différentes méthodes d'affichages (comme la démo officielle). Encore une fois, n'hesitez pas à utiliser le page flipping, il est souvent très efficace et très répendu!

4.4.2 Le Triple Buffering

On pourrait se demander à quoi sert le triple buffering, puisque le page flipping semble très bien marcher. On croit assister à une course à la prolifération des termes techniques qui déroute l'utilisateur. Et bien en fait, le triple buffering, c'est grosso-modo la même chose que le page flipping sauf qu'on utilise ici trois pages de mémoire vidéo au lieu de deux. Evidemment, c'est encore plus gourmant en ressource vidéo, mais s'il est géré, il donne lui aussi d'excellents résultats. Fort heureusement, Allegro fournit de quoi tester si ce mode est supporté par la carte vidéo. Et bien, je n'ai plus grand chose à dire dans la rubrique "généralités" alors on va se lancer, hein?

```
/* On déclare les trois buffers dont on va avoir besoin et le buffer qui pointera vers un des deux autres... */
BITMAP* page1, page2, page3, buffer;
int current_page = 1;

set_color_depth(16);

/* Comme d'hab', on initialise d'abord le mode vidéo */
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 non supporté!");
      return 1;
}

/* Maintenant, c'est au tour de l'initialisation du 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);

/* Ici, on vérifie que les TROIS pages sont bien créées */
if ((!page1) || (!page2) || (!page3)) {
      set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
      allegro_message ("Pas assez de mémoire pour faire du triple buffering !");
      return 1;
}
```

Si tout cela parait à la fois clair et obscur, c'est normal. Alors, tout d'abord, il faut s'assurer que la carte vidéo supporte le triple buffering. Pour cela, on a une variable globale appelée **gfx_capabilities** :

int gfx_capabilities;

Cette variable est utilisée avec beaucoup de **flags**. 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). La je vous renvoie à l'aide car il y en a énormément et qu'il serait inutile de les passer en revue ici. (allegro/docs/html/allegro.html, je vous le rappelle). Tiens, j'en cite un qui est sympa quand même:

- GFX_HW_VRAM_BLIT : celui décrit si la carte vidéo ets capable d'accélérer matériellement le dessin à partir de la mémoire vidéo. Si c'est le cas, c'est bingo, vous pourrez gouter aux joies du page flipping.

- GFX_CAN_TRIPLE_BUFFER : c'est le flag qui nous interesse ici. Si il est armé, c'est parfait, sinon, il reste une dernière chance : la fonction **enable_triple_buffer**.

int enable_triple_buffer(void);

Désolé de casser l'ambiance mais cette fonction n'est efficace que sous certaines conditions... Apparemment, elle semble etre efficace sous DOS mais sous Windows, je ne l'ai jamais vu marcher... Bon, elle est sensée faire son possible pour autoriser le triple buffering, et elle retourne 0 si le triple buffering a été établi avec succès. C'est tout ce qu'il y a à savoir

Donc, reprenons le fil de l'initialisation. On regarde une première fois si le triple buffering est activé. Si non, on tente avec cette fonction et on fait une ultime vérification. Si tout va bien, on peut passer à l'allocation des trois pages vidéo et vérifier qu'elles ont bien été toutes crées avec succès. Rien n'est moins sur car si vous tentez en 1600*1200, tout ça prend de la place quand même! Enfin, voilà, l'initialisation du triple buffering est finie ! On embraye avec l'affichage du disque (le coup classique) :

```
while (key[KEY_ESC]) {
```

```
        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 */
        do {
        } while (poll_scroll());

        /* on poste la requete */
        request_video_bitmap(buffer);

        /* Ici, on s'occupe de l'alternance des deux 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;
        }
}
```

Bien, voilà une bonne chose de faîte ! Alors, comme le constater, le principe est grosso-modo le même à propos des changements de pages, on va expliquer les nouvelles fonctions :

int request_video_bitmap(BITMAP *bitmap);

Cette fonction ne devrait être utilisée que pour le triple buffering. Elle ne fait que poser une requete de "page flip" sur le **bitmap** passé en paramètre. Concrètement, contrairement aux autres fonctions, celle là rend la main tout de suite !

int poll_scroll(void);

Cette fonction est aussi utilisée seulement pour le triple buffering. Pour faire simple, elle vérifie si la fonction d'avant a fini son boulot. Elle retourne zéro si on peut rappeler **request_video_bitmap** sinon elle retourne autre chose...

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 "prête 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 empêche 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();**

int desktop_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

Nul besoin de diviser ce chapitre en sous chapitre, tellement l'utilisation de la souris est simple! Nous avions vu au début de ce tutorial comment intialiser la souris. Maintenant, on va voir comment on peut l'utiliser simplement. Mais pour commencer, je sais que je vais clarifier les esprits en recopiant notre code d'initialisation de la souris ici :

```
/* Si la fonction retourne un échec, alors… */
if(install_mouse() == -1) {
      /* On affiche le message d'erreur */
      allegro_message("Erreur ! %s", allegro_error) ;
      /* Et on quitte le programme ! */
      return -1 ;
}
/* Maintenant, on est sur que la souris marche ici ! */
```

Voilà qui est fait. install_mouse est une fonction d'Allegro. Elle retourne le nombre de boutons disponibles. C'est à dire que les plus chanceux veront la fonction retourner 3. Les autres ne recevront que 2. Enfin, les plus malheureux ne possedant pas de souris vont juste récupérer la valeur -1. N'oubliez jamais ce genre de test, ca ne coûte pas grand chose à mettre en place et ça peut vraiment simplifier la vie en cas de problèmes. Je concède que, de nos jours, ne pas avoir de souris est rarissime, mais sait-on jamais? Un utilisateur DOS par exemple peut ne pas avoir de driver installé pour sa souris... C'est une habitude qu'il est bon de prendre : vérifiez systématiquement les valeurs que retournent les fonctions d'initialisation. Mais trève de bavardages, voyons enfin comment s'utilise cette souris... C'est bien plus simple qu'il n'y paraît : vous n'aurez besoin que de quelques variables externes. Les voici :

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

Mais non, ce n'est pas déroutant : mouse_x représente la coordonnée actuelle de votre souris en abscisse. Même chose pour mouse_y sur l'axe des ordonnées. Je tiens à repréciser que l'origine du repère dans lequel les valeurs seront données est le coin supérieur gauche de votre écran. Ainsi, le pixel dans ce coin a pour coordonnées (0,0). Vous pouvez à tout instant savoir où se trouve votre souris : elle se trouve au point de coordonnées (mouse_x, mouse_y). C'est à dire de (0,0) à (SCREEN_W, SCREEN_H). Mais, alors, vous allez me dire : mais comment rafraichir ces fameuses variables ? Et bien, comme pour le clavier, Allegro va les rafraichir tout seul comme un grand ! Vous n'aurez strictement rien à faire, tout en étant sûr que votre programme connait à chaque instant la position de la souris que l'utilisateur -enfin le joueur...- est en train de manipuler. Quant à la variable mouse_z, elle représente la position de la molette de la souris, si bien sur molette il y a. Lorsque vous lancerez init_mouse(), mouse_z vaudra 0 par défaut. Si vous scrollez avec votre souris, vous incrémentez ou décrémentez la valeur de cette variable. mouse_b représente l'autre aspect essentiel de la souris : c'est la variable d'état des boutons de la souris. Grace à cette variable, vous saurez si l'utilisateur est en train de cliquer sur sa souris -bouton gauche, bouton droit et même celui du milieu...-. Le bit 0 est le bouton de gauche, le 1 celui de droite et le 2 celui du milieu.

On va voir ce que tout ça donne en pratique : créez un programme qui initialise la souris, le mode vidéo, qui charge un sprite en mémoire et qui crée un buffer de la taille de l'écran. Tout ça a déjà été vu, il ne devrait plus y avoir de problèmes. Par contre, on va mainteant refaire une boucle principale, en utilisant la souris. Pour le sprite, essayez de prendre une petite image de pointeur de souris, ça sera nettement plus parlant qu'une simple image représentant un mur!

```
while(!(mouse_b & 2)) { /* Tant que le bouton droit n'est pas préssé… */

    /* On commence par effacer le buffer */
    clear_bitmap(buffer) ;

    /* Si bouton gauche, on affiche le sprite aux coordonnées de la souris */
    if(mouse_b & 1)
        draw_sprite(buffer, sprite, mouse_x, mouse_y);

    /* Maintenant, il faut copier le contenu de notre buffer dans l'écran */
    blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H) ;
}
```

Après cette boucle, n'oubliez pas de libérer votre **sprite** et le **buffer** bien sûr. C'est le moment de vérité : compilez et lancez le programme. Si vous appuyez sur le bouton gauche, votre sprite devrait apparaître aux coordonnées de votre souris! En revanche, si vous ne cliquez pas, vous n'obtiendrez qu'un écran noir. Pour quitter, rien de plus simple, faîtes juste un clique droit ! Enfantin, n'est-ce pas? Il existe une autre méthode pour afficher la souris automatiquement. Mais personnelement, je ne vous la livrerai pas parce que je la trouve « inutile ». Enfin, disons que la façon que je vous ai montrée pour afficher la souris autorise un contrôle total et plus transparent qu'avec l'autre méthode. En effet, le fait de l'afficher dans la boucle principal avec tout le reste vous permet d'en faire ce que vous voulez le plus simplement du monde : il suffit de changer le **sprite** en un autre sprite pour afficher un curseur différent. Si l'envie vous prend de décaler la souris vers la droite ou la gauche, rien ne vous empêche de le faire :

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

Il est aussi très facile de masquer le curseur quand on le désire. Vous pouvez de plus réaliser très facilement des curseurs animés ainsi... On verra comment faire simplement plus tard, en particulier grâce à l'utilisation de timers. Il existe quelques fonctions sympathiques qui peuvent vous sauver la mise pour certains types de jeu :

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

Cette fonction mesure la distance que la souris a parcourue depuis le dernier appel de cette même fonction. C'est à dire que si l'utilisateur s'amuse à aller très loin à droite en dehors de l'écran, cette fonction detectera quand même son mouvement. Vu qu'elle est basée sur un déplacement et non plus sur une position, elle peut être très pratique pour les jeux qui necessitent des mouvements infinis. Vous passez en paramètres deux variables, la fonction vous les modifiera avec les nouvelles valeurs.

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

Très simplement, cette fonction sert à définir la zone de déplacement de la souris. Par défaut, si vous n'appelez pas cette fonction, vous vous retrouvez avec la zone : (0,0)-(SCREEN_W-1, SCREEN_H-1). Vous passez en paramètre les coordonnées du coin supérieur gauche puis ceux du coin inférieur droit.

void position_mouse(int x, int y);

Cette fonction est utile si vous désirez recentrer ou placer où vous le désirez la souris. Pas grand chose à dire dessus : vous avez juste à passer les nouvelles coordonées, et le tour est joué!

Et bien que crois que vous connaissez l'essentiel pour vous en sortir avec votre souris. Il reste quelques fonctions intéressantes cependant, essayez de jeter un oeil à la documentation si vous recherchez d'autres possibilités plus pointues. Mais vous devriez pouvoir réaliser « facilement » n'importe quelle interface à la souris avec ce que je vous ai montré...

8. Les Timers

8.1 Introduction - Théorie (simplifiée...)

Ce sont des outils absolument indispensables, et quoi qu'il arrive, si vous décidez un jour de faire un jeu décent, vous ne pourrez pas y couper... Pourquoi? Et bien tout simplement parce que jusqu'à présent, la vitesse d'execution de tous les essais (ou jeux?) que vous avez pu programmer dépendent directement de la vitesse du processeur et de la carte graphique qui les font tourner! Alors évidemment, cela semble au premier aspect vraiment problématique! Comment pourrait-on faire? En fait, il faut plutot réfléchir autrement : sur quoi peut on se baser pour être sur que la vitesse du résultat final sera la même sur toutes les machines ? Evidemment pas sur un quelconque élément propre à l'ordinateur, sinon c'est la mort assurée. Et bien des programmeurs il y a fort longtemps ont trouvé la parade : il suffit de se baser sur quelque chose que tout le monde possède et qui est suffisemment fiable pour ce qu'on veut en faire : il s'agit de l'horloge interne de l'unité centrale. Tout cela est très vulgarisé mais déjà suffisant pour comprendre un peu ce qui se passe. Je vais entrer dans des considérations moins matérielles mais néanmoins plus importantes pour nous.

Vous avez tous remarqué une chose jusqu'à présent : un jeu est d'autant plus fluide qu'il arrive à avoir un taux de fps (Frames Per Second = Images par seconde) élevé. Vous avez remarqué aussi que les FPS les plus élevés s'obtiennent quand il s'agit d'afficher peu de choses... En fait, c'est très simple : jusqu'à présent le raisonnement était le suivant : plus l'ordinateur est rapide, plus le jeu va vite. On pourra 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) .