Chapitre 16

Graphismes en Noir et Blanc

Nous avons déjà vu au cours de certains exemples de ce Tutorial, comment effacer l'écran, afficher une ligne, ou comment sauvegarder puis restaurer l'écran...
Mais ce chapître va nous permettre d'aller une étape plus loin, puisqu'il va nous permettre de découvrir une partie des fonctions du TIOS permettant de réaliser simplement des affichages graphiques, en noir et blanc.

I:\ Afficher du Texte

Nous avons déjà utilisé la fonction `printf` pour afficher du texte à l'écran... Mais nous allons voir, au cours de cette partie, qu'il y a d'autres possibilités, plus avancées.

A: Afficher une Chaîne de Caractères

Tout d'abord, rappelons rapidement que la fonction `DrawStr` permet d'afficher une chaîne de caractères à la position souhaitée sur l'écran ; voici son prototype :

```
void DrawStr(short x, short y, const char *str, short Attr);
```

Le dernier paramètre correspond à la façon dont la chaîne de caractères sera affichée : en noir sur blanc, en blanc sur noir, ...
Par exemple, exécutez la portion de code suivante, pour avoir quelques idées de ce qui est possible :

```
DrawStr(0, 0, "Hello World !", A_NORMAL);
DrawStr(0, 10, "Hello World !", A_REVERSE);
DrawStr(0, 20, "Hello World !", A_SHADED);
```

Et pour plus d'informations, notamment sur les autres types d'affichage possible, n'hésitez pas à consulter la documentation...

B: Taille de la Police

Cela dit, pourquoi se limiter à la taille de police par défaut, qui, qui plus est, n'est pas toujours la même, puisqu'elle dépend souvent de ce que vous faisiez avant de lancer votre programme, de la façon dont il a été lancé, et de la plate-forme sur laquelle il s'exécute ?
En effet, le TIOS vous propose une fonction qui permet de déterminer quelle est la police actuellement sélectionnée ; voici son prototype :

```
unsigned char FontGetSys(void);
```

La valeur retournée est soit `F_4x6`, qui correspond à la petite police, `F_6x8`, qui correspond à la police de taille moyenne, soit `F_8x10`, qui correspond à la grande taille de police.
Et, naturellement, voici la fonction qui permet de modifier la taille de police courante :

```
unsigned char FontSetSys(short Font);
```

Elle prend en paramètre la nouvelle taille de police, parmis les trois présentées juste au dessus, et retourne la taille de police précédemment sélectionnée.
Pour illustrer ceci, voici un bref exemple :

```
unsigned char save_taille = FontSetSys(F_4x6);
DrawStr(0, 0, "Bonjour !", A_NORMAL);
FontSetSys(F_6x8);
DrawStr(0, 10, "Bonjour !", A_NORMAL);
FontSetSys(F_8x10);
DrawStr(0, 20, "Bonjour !", A_NORMAL);
FontSetSys(save_taille);
```

Cette portion de code va commencer par sauvegarder la police courante, puis en changer trois fois, en affichant, à chaque fois, un bref message avec la police courante.
Enfin, nous restaurons la police de caractères qui avait été, au départ, sauvegardée.

C: Quelques autres fonctions

A présent, principalement pour montrer qu'il existe d'autres fonctions, en voici deux qui peuvent vous être utiles.

Tout d'abord, la fonction `DrawChar`, qui permet d'afficher le caractère que vous voulez à la position que vous voulez ; voici son prototype :

```
void DrawChar(short x, short y, char c, short Attr);
```

On spécifie le caractère par son code (vous trouverez la liste des codes de cacactères dans le manuel de votre calculatrice), et le dernier paramètre fait parti de la liste de ceux qui sont acceptés par la fonction `DrawStr`, que nous venons d'étudier.
Par exemple, pour afficher un caractère ©, nous pourrions utiliser l'instruction suivante :

```
DrawChar(10, 10, 169, A_NORMAL);
```

Sachant que le caractère © a pour code 169.

Une autre fonction qui peut vous être utile, par exemple si vous souhaitez centrer une chaîne de caractères sur l'écran, est la fonction `DrawStrWidth`, dont le prototype est le suivant ; elle permet de déterminer la longueur d'une chaîne de caractère en fonction d'une des trois tailles de police que nous avons déjà évoqué :

```
short DrawStrWidth(const char *str, short Font);
```

Et voici un petit exemple illustrant l'emploi de cette fonction :

```
char str[] = "Hello World !";
unsigned char save_taille_2 = FontSetSys(F_4x6);
printf("%s => %d\n", str, DrawStrWidth(str, FontGetSys()));
FontSetSys(F_6x8);
printf("%s => %d\n", str, DrawStrWidth(str, FontGetSys()));
FontSetSys(F_8x10);
printf("%s => %d\n", str, DrawStrWidth(str, FontGetSys()));
FontSetSys(save_taille_2);
```

Cet exemple affiche à l'écran, pour chacun des trois tailles de police, la longueur d'un bref message.

II:\ Afficher des Graphismes Simples

A présent, nous allons voir quelques fonctions nous permettant d'afficher des graphismes simples à l'écran. Nous commencerons par une fonction permettant d'afficher des pixels, puis nous passerons à l'affichage de lignes.

Notez que les fonctions que nous verrons au cours de cette partie ne fonctionnent que si leurs coordonnées sont valides : vous ne **devez pas** essayer d'afficher quelque chose en dehors de l'écran avec ces fonctions ! Dans le cas contraire, votre programme risque de planter !
Nous verrons au cours de la partie suivante qu'il existe d'autres fonctions, pour lesquelle ce problème ne se pose pas.

A: Afficher des Pixels

Tout d'abord, commençons par afficher des pixels seuls. Pour cela, on utilise cette fonction :

```
void DrawPix(short x, short y, short Attr);
```

Les deux premiers paramètres correspondent aux coordonnées à l'écran du pixel que l'on souhaite afficher, et le dernier correspond au mode d'affichage : A_NORMAL pour allumer le pixel, A_REVERSE pour l'éteindre, et A_XOR pour inverser son état.

Et voici un petit exemple :

```
DrawPix(10, 20, A_NORMAL);
DrawPix(11, 20, A_REVERSE);
DrawPix(12, 20, A_XOR);
```

Ici, on allume le pixel de coordonnées (10, 20), on éteint celui de coordonnées (11, 20), et on inverse l'état de celui de coordonnées (12, 20).

Simple, n'est-ce pas ?
Souvenez-vous juste que les coordonnées doivent correspondre à des points valides, ce qui signifie que x doit varier entre 0 et 239 compris, et que y doit être compris entre 0 et 127 inclus.

B: Afficher des Lignes

A présent, nous allons voir comment il nous est possible d'afficher des lignes à l'écran. Puisque vous connaissez le nom de la fonction d'affichage de pixels, il devrait vous être relativement simple de deviner celui de la fonction de tracés de lignes, dont le prototype est le suivant :

```
void DrawLine(short x0, short y0, short x1, short y1, short Attr);
```

Cette fonction prend en paramètres les coordonnées des deux points formants les extrémités de la ligne, ainsi que le mode dans lequel elle doit être affichée.
Pour plus de détails au sujet des modes disponibles, qui sont assez nombreux, je vous invite à consulter la documentation de TIGCC.

Et voici un petit exemple de programme traçant trois lignes en utilisant des modes différents :

```
#include <tigcclib.h>

void _main(void)
{
    ClrScr();

    DrawLine(10, 10, 50, 50, A_NORMAL);
    DrawLine(10, 20, 60, 60, A_THICK1);
    DrawLine(30, 10, 30, 80, A_XOR);

    ngetchx();
}
```

Exemple Complet

Vous remarquerez ici que le mode A_XOR a pour effet d'inverser la couleur de la ligne : les pixels qui étaient blancs deviennent noir, et vice-versa.

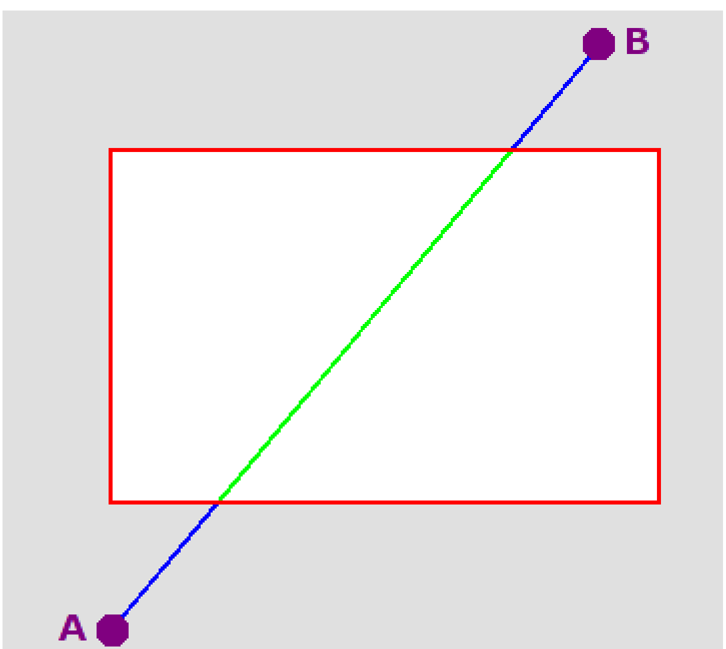
III:\ Graphismes et Clipping

Les fonctions que nous venons d'étudier nous permettent déjà d'afficher quelques graphismes simples... Cela dit, nous devons, en les employant, prendre garde à ne pas dépasser de l'écran, sous peine de plantages de notre programme.
Pour remédier à ce problème, nous allons voir qu'il existe la notion de "clipping", et des fonctions utilisant ce principe.

A: Notion de Clipping

"Clipper" un graphisme par rapport à une "zone de clipping" signifie ne conserver de ce graphisme que les parties qui se trouvent sur ladite zone de clipping.

Je pense que ceci sera plus compréhensible avec un exemple et un petit schéma...
Admettons que l'on veuille tracer une ligne entre deux points A et B. Ces deux points sont en dehors de la zone de clipping que l'on accepte, celle-ci étant souvent l'écran.
Etant donné qu'afficher des graphismes en dehors de l'écran peut provoquer un plantage de notre programme, il va nous falloir éliminer les portions de la ligne qui ne sont pas sur celui-ci, comme ceci :



La rectangle rouge correspond à notre zone de clipping ; les deux points A et B, les extrémités de la ligne, sont en dehors de celle-ci...
Le fait de clipper par rapport à notre zone de clipping aura pour effet d'éliminer les deux parties bleues de la ligne, ce qui nous permettra de ne conserver que la partie verte, que nous pourrons alors afficher sans danger.

B: Quelques fonctions graphiques avec clipping

Le TIOS nous fourni de nombreuses fonctions graphiques avec clipping. Nous n'en verrons que quelques unes, afin d'illustrer leur principe de fonctionnement, et je vous laisse vous reporter à la documentation pour toutes les connaître.

1: Clipping par défaut

Certaines fonctions du TIOS travaillent avec une zone de clipping par défaut. C'est par exemple le cas de la fonction suivante, qui permet d'afficher un pixel :

```
void DrawClipPix(short x, short y);
```

Pour ces fonctions, la zone de clipping doit être fixée en utilisant la fonction `SetCurClip`, dont le prototype est le suivant :

```
void SetCurClip(const SCR_RECT *clip);
```

Cette fonction prend en paramètre un pointeur sur une union de type `SCR_RECT`, qui est définie de la manière suivante :

```
typedef union {
    struct {
        unsigned char x0, y0, x1, y1;
    } xy;
    unsigned long l;
} SCR_RECT;
```

Donc pour définir une zone de clipping pour les fonctions telles `DrawClipPix`, il nous est possible d'utiliser quelque chose comme ceci (Dans ce cas, on définit comme zone de clipping une surface correspondant à l'écran d'une TI-92+ ou v200 ; on aurait pu choisir d'autres valeurs, naturellement) :

```
SCR_RECT rect;
rect.xy.x0 = 0;
rect.xy.y0 = 0;
rect.xy.x1 = 239;
rect.xy.y1 = 127;

SetCurClip(&rect);
```

Cela dit, si vous n'avez pas l'intention d'utiliser plusieurs fois votre variable que nous avons ici appelé "rect", vous pouvez, du fait que TIGCC est un compilateur GNU-C, utiliser ce type de syntaxe, plus concise :

```
SetCurClip(&(SCR_RECT){{0, 0, 239, 127}});
```

2: Clipping par fonction

En revanche, une bonne partie des fonctions de `graph.h` - la plupart, en fait - prennent la zone de clipping en paramètre, sous forme, ici aussi, d'une union de type `SCR_RECT`.
Par exemple, pour afficher un caractère, une élipse, ou un triangle, vous pourrez utiliser une des fonctions qui suit :

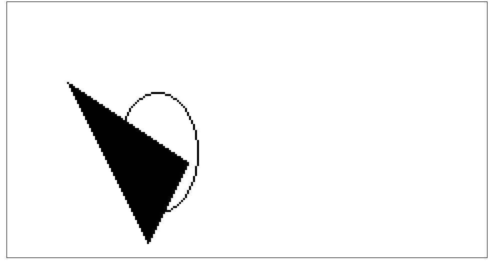
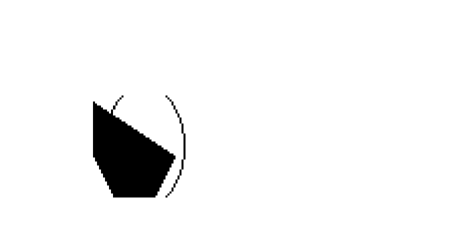
```
void DrawClipChar(short x, short y, short c, const SCR_RECT *clip, short Attr);

void DrawClipEllipse(short x, short y, short a, short b, const SCR_RECT *clip, short Attr);

void FillTriangle(short x0, short y0, short x1, short y1, short x2, short y2, const SCR_RECT *clip, short Attr);
```

Et voici un petit exemple utilisant deux de ces fonctions :
La variable `ScrRect` est définie au niveau de TIGCC comme un pointeur sur un `SCR_RECT` faisant la taille de l'écran ; plus rapide à utiliser qu'à re-définir !

```
#include <tigcclib.h>

void _main(void)
{
    SCR_RECT rect;

    ClrScr();
    DrawClipEllipse(75, 75, 20, 30, ScrRect, A_NORMAL);
    FillTriangle(30, 40, 90, 80, 70, 120, ScrRect, A_NORMAL);
    ngetchx();


    rect.xy.x0 = 50;
    rect.xy.y0 = 50;
    rect.xy.x1 = 100;
    rect.xy.y1 = 100;

    ClrScr();
    DrawClipEllipse(75, 75, 20, 30, &rect, A_NORMAL);
    FillTriangle(30, 40, 90, 80, 70, 120, &rect, A_NORMAL);
    ngetchx();
}
```

Exemple Complet

Et voici les captures d'écran correspondantes : sur la première, la zone de clipping est l'écran. Sur la seconde, elle est plus petite ; les graphismes sont donc "coupés" :

Ce qui nous montre toute l'efficacité du clipping.

Pour finir, deux dernières fonctions que je souhaite vous présenter ici :

```
void DrawClipLine(const WIN_RECT *Line, const SCR_RECT *clip, short Attr);

void DrawClipRect(const WIN_RECT *rect, const SCR_RECT *clip, short Attr);
```

Comme leurs noms respectifs l'indiquent, ces fonctions permettent d'afficher une ligne et un rectangle, avec clipping.
Leur particularité, qui explique qu'elles méritent de figurer ici est qu'elles prennent en paramètre un pointeur sur une struture de type `WIN_RECT` pour désigner les deux points constituant les extrémités de la droite, ou les deux coins supérieur gauche et inférieur droit du rectangle.
Cette structure est définie ainsi :

```
typedef struct {
    short x0, y0, x1, y1;
} WIN_RECT;
```

Comme pour les unions de type `SCR_RECT`, on peut utiliser deux types de syntaxe :
Tout d'abord, la syntaxe la plus "conventionnelle", si je puis dire :

```
WIN_RECT rect;
rect.x0 = 50;
rect.y0 = 50;
rect.x1 = 100;
rect.y1 = 100;
DrawClipRect(&rect, ScrRect, A_NORMAL);
```

Ou alors, en utilisant les cast-constructor du GNU-C :

```
DrawClipLine(&(WIN_RECT){30, 40, 90, 78}, ScrRect, A_NORMAL);
```

Vous ne manquerez pas de noter que, une fois encore, on a utilisé `ScrRect` pour signifier que nous voulions utiliser tout l'écran comme zone de clipping.

IV:\ Manipulation d'Images

Pour terminer ce chapitre, nous allons voir une série de fonctions dont le but est de nous permettre d'enregistrer ou d'afficher des images complètes.

A:\ Afficher une image définie dans le programme

Tout d'abord, nous allons voir comment encoder une image directement dans notre programme, afin de l'afficher par la suite.
La fonction que nous utiliserons au cours de cette partie pour afficher des images est `BitmapPut` ; son prototype est le suivant :

```
void BitmapPut(short x, short y, const void *BitMap, const SCR_RECT *clip, short Attr);
```

Le troisième argument attendu par cette fonction est un pointeur sur une struture de type `BITMAP`, définie comme ceci :

```
typedef struct
{
    unsigned short NumRows, NumCols;
    unsigned char Data[];
} BITMAP;
```

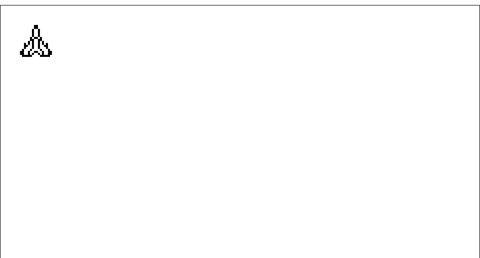
Cette structure correspond à un descripteur d'image : elle permet de mémoriser, en nombre de pixels, le nombre de lignes et le nombre de colonnes de l'image ; ces informations sont ensuite suivies des données de l'image elle-même, ligne par ligne, de gauche à droite.
Chaque pixel de l'image est encodé sous forme d'un bit, valant 1 si le pixel est noir et 0 sinon. Si la largeur de l'image n'est pas multiple de 8, les données des derniers octets de droite doivent être complétées par des 0, afin que l'on ait des octets complets.

Et voici un petit exemple où on déclare puis affiche une image de 16 pixels de coté :

```
#include <tigcclib.h>

void _main(void)
{
    ClrScr();

    static BITMAP bitmap =
    {
        16, 16,
        {
            0b00000001, 0b10000000,
            0b00000001, 0b10000000,
            0b00000010, 0b01000000,
            0b00000010, 0b01000000,
            0b00000010, 0b01000000,
            0b00000010, 0b01000000,
            0b00000101, 0b10100000,
            0b00000110, 0b01100000,
            0b00001010, 0b01010000,
            0b00101010, 0b01010100,
            0b00110010, 0b01001100,
            0b00100010, 0b01000100,
            0b01000100, 0b00100010,
            0b11001001, 0b10010011,
            0b11001010, 0b01010011,
            0b01111100, 0b00111110,
        }
    };

    BitmapPut(10, 10, &bitmap, ScrRect, A_NORMAL);

    ngetchx();
}
```

Exemple Complet

Et une capture d'écran de notre programme lors de son exécution :



Notons que BitmapPut est surtout adaptée à l'affichage de grandes images : pour des images de 8x8, 16x16, ou 32x32, nous aurions plutôt tendance, en pratique, à utiliser des fonctions d'affichage de sprites, telles celles que nous étudierons plus loin au cours de ce tutorial.

B:\ Sauvegarder une image, et l'afficher

A présent, nous allons voir comment sauvegarder une portion d'image affichée à l'écran, afin de pouvoir, par la suite, l'afficher avec `BitmapPut`, que nous avons déjà utilisé.
Pour enregistrer une portion d'image, nous utiliserons la fonction `BitmapGet`, dont le prototype est le suivant :

```
void BitmapGet(const SCR_RECT *rect, void *BitMap);
```

Cette fonction va sauvegarder les données contenues dans le rectangle décrit par son premier paramètre vers la zone mémoire sur laquelle pointera son second paramètre.
Cela signifie que nous allons avoir besoin d'une zone mémoire de taille suffisante... Pour calculer cette taille, nous allons utiliser la fonction `BitmapSize`, qui, elle aussi, prend en paramètre la zone de l'écran que l'on veut sauvegarder, et retourne le nombre d'octets dont nous allons avoir besoin :

```
unsigned short BitmapSize(const SCR_RECT *rect);
```

Maintenant que nous connaissons la taille du bloc mémoire dont nous avons besoin pour sauvegarder notre portion d'image, nous pouvons, par exemple, l'allouer grâce à un `malloc`, puis, appeler `BitmapGet`, en lui passant en paramètre l'adresse de ce bloc mémoire.
Pour réafficher notre image, nous faisons comme plus haut. Voici un exemple illustrant mes propos :

```
#include <tigcclib.h>

void _main(void)
{
    unsigned short taille = BitmapSize(&(SCR_RECT){{100, 40, 200, 110}});

    unsigned char *buffer = malloc(taille);
    if(buffer)  // Important !
    {
        BitmapGet(&(SCR_RECT){{100, 40, 200, 110}}, buffer);

        ClrScr();

        BitmapPut(10, 10, buffer, ScrRect, A_NORMAL);

        ngetchx();
        free(buffer);  // Important !
    }
}
```

Exemple Complet

Vous noterez que, comme nous l'avons appris un peu plus tôt, nous avons pensé à vérifier que l'allocation mémoire avait bien réussi, et nous n'avons pas oublié de libérer la mémoire allouée.

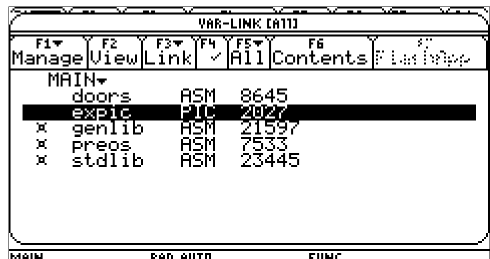
Etant donné que le programme que nous venons d'écrire sauvegarde une portion de l'écran, il peut être intéressant de savoir ce que celui-ci contenait au moment où j'ai exécuté le-dit programme :

Et voici l'affichage obtenu au cours de l'exécution de notre programme : on reconnait aisément la portion d'écran sauvegardée, et ré-affichée :

Naturellement, selon ce que vous avez d'affiché à l'écran au moment où vous lancez ce programme, il est probable que vous obteniez une sortie différente...
Notez aussi que le programme que j'ai choisi pour illustrer mes propos a été conçu pour une taille d'écran de TI-92+ ou de Voyage 200. Si vous avez une TI-89, je vous laisse le loisir de l'adapter - arrivé à ce niveau du Tutorial, vous devriez y parvenir sans difficulté.

C:\ Afficher une image contenue dans un fichier PIC

Pour le dernier point de cette partie, nous allons voir comment afficher à l'écran le contenu d'un fichier de type "PIC" - les images du TIOS. Pour l'exemple, je vais utiliser une des images fournie avec txtrider (notez que les images de txtrider sont compressées ; il m'a donc fallu la décompresser auparavant). Voici comment elle apparait une fois affichée dans l'écran graph, et dans le VAR-LINK :



Pour afficher une image contenue dans un fichier PIC, nous allons utiliser une fonction fournie dans la FAQ de TIGCC. Tout simplement, cette fonction prend en paramètre le nom du fichier sous forme d'un `SYM_STR`, ce qui implique d'utiliser la macro `SYMSTR` pour l'obtenir depuis une chaine de caractères, ainsi que les coordonnées où l'on veut afficher l'image, et le mode dans lequel on veut réaliser la sortie écran.

Et voici un petit exemple utilisant la fonction `show_picvar` qui nous est proposée dans la documentation, qui affiche à l'écran l'image que je vous ai présenté un peu plus haut :

```
#include <tigcclib.h>

short show_picvar(SYM_STR SymName, short x, short y, short Attr);

void _main(void)
{
    ClrScr();

    show_picvar(SYMSTR ("expic"), 10, 10, A_NORMAL);

    ngetchx();
}


short show_picvar(SYM_STR SymName, short x, short y, short Attr)
{
    SYM_ENTRY *sym_entry = SymFindPtr(SymName, 0);
    if(!sym_entry)
        return FALSE;
    if(peek(HToESI(sym_entry->handle)) != PIC_TAG)
        return FALSE;
    BitmapPut(x, y, HeapDeref(sym_entry->handle) + 2, ScrRect, Attr);
    return TRUE;
}
```

Exemple Complet

Et voici une capture d'écran prise pendant l'exécution du programme :

Naturellement, libre à vous de modifier cette fonction si vous souhaitez obtenir un comportement différent !

Le prochain chapitre va nous permettre de voir ce qu'est la "VAT", et comment s'en servir, voire la manipuler.
Plus loin au cours de ce tutorial, nous aurons l'occasion de revenir sur l'affichage de graphismes, que ce soit en niveaux de gris, ou en utilisant des sprites...