

TP N°9 Application graphique avec SDL (1)



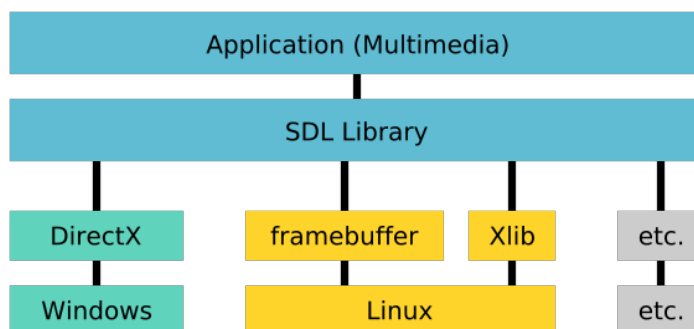
Objectifs du TP : développer une application graphique simple en C à l'aide de la bibliothèque SDL (multi-plateforme). Pour atteindre cet objectif, il faudra :

- Savoir installer une bibliothèque (*library*) et faire en sorte que notre programme puisse faire appel à cette bibliothèque,
- Connaître les fonctions d'initialisation graphique (mode vidéo, couleurs, taille de la fenêtre, ...),
- Connaître le principe des formats d'image (codage des couleurs, transparence,...). Nous n'utiliserons ici que le format «*bmp*».
- Comprendre la notion d'évènement et de boucle d'attente d'évènements.

1. L'installation de la bibliothèque SDL

1.1. Présentation de la SDL

- **Simple DirectMedia Layer (SDL)** est une bibliothèque (*library*) très utilisée dans le monde de la création d'applications multimédias en deux dimensions comme les jeux vidéo, les démos graphiques, les émulateurs, etc. Sa simplicité, sa flexibilité, sa portabilité et surtout sa licence GNU LGPL contribuent à son grand succès.
- SDL est disponible sous de nombreux systèmes d'exploitation incluant *Linux*, *Windows*, *Mac OS X*, *FreeBSD*, *Android*, *iOS*,... et des systèmes pour consoles de jeux. Elle est écrite en C, mais peut être utilisée par d'autres langages de programmation (C++, C#, ObjectiveC, Java, Perl, Python, Ruby,...).
- SDL permet de gérer l'affichage 2D, le son, les périphériques d'entrée (clavier, souris, *joystick*), le *multithreading*, les *timers*,... Elle agit en tant que couche (*layer*) logicielle au-dessus du système graphique sous-jacent. Sous *Windows*, SDL 1.2 (la dernière version stable) s'appuie sur *DirectX* 7. Sur les plateformes *Unix/Linux*, et autres où le système graphique est *XWindow*, la SDL s'appuie sur la bibliothèque *Xlib*.



Source :
en.wikipedia.org/wiki/Simple_DirectMedia_Layer

1.2. Vérification de l'installation

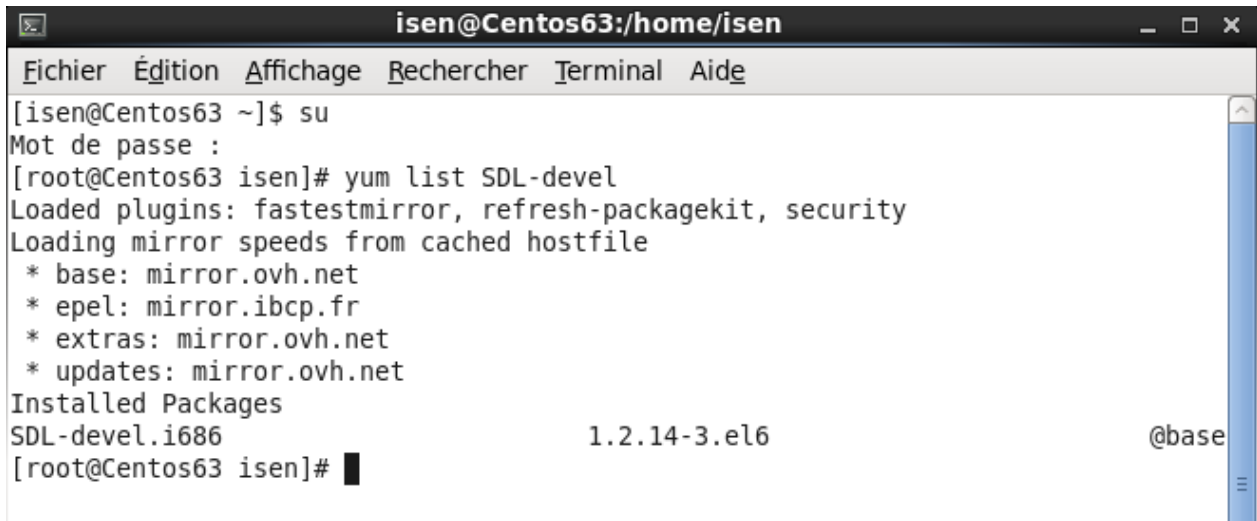
Dans les distributions *Red Hat* (*Centos*, *Fedora*), on utilise la commande *yum*; dans les distributions *Debian* (*Ubuntu*) c'est *apt-get*.

Commencez par obtenir les privilèges administrateur en tapant dans un terminal la commande *su* suivie du mot de passe administrateur *isen29*. Pour vérifier si les paquets correspondant à

SDL sont bien installés, entrez ensuite la ligne de commande :

```
yum list SDL-devel
```

Vous devriez obtenir un affichage du type suivant :



```
isen@Centos63 ~/home/isen
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
[isen@Centos63 ~]$ su
Mot de passe :
[root@Centos63 isen]# yum list SDL-devel
Loaded plugins: fastestmirror, refresh-packagekit, security
Loading mirror speeds from cached hostfile
* base: mirror.ovh.net
* epel: mirror.ibcp.fr
* extras: mirror.ovh.net
* updates: mirror.ovh.net
Installed Packages
SDL-devel.i686                               1.2.14-3.el6                @base
[root@Centos63 isen]#
```

Si les paquets ne sont pas installés, veuillez vous reporter au paragraphe suivant sinon vous pouvez vous déconnecter de la session administrateur en tapant *exit* et passer directement à la partie 2.

1.3. Installation de la SDL sous *Linux (si non installée)*

- Installation de SDL sous *Centos* ou *Fedora* :

```
$ yum install SDL-devel
```

- Installation de SDL sous *Ubuntu* :

```
$ apt-get install libsdl1.2-dev
```

2. La mise en place de la partie graphique de votre programme C

1.1. Code C minimal : inclusion des *headers* et initialisation de la SDL

- Vous devez ajouter en début de programme l' *include* suivant :

```
#include <SDL/SDL.h>
```

- **Initialisation** : ce bout de code C est à placer au début de la fonction *main()* : il permet d'appeler la fonction *SDL_Init()* qui prend en paramètre le nom de la partie de la SDL à charger (*SDL_INIT_VIDEO*, *SDL_INIT_AUDIO*, *SDL_INIT_TIMER*, *SDL_INIT_JOYSTICK*, *SDL_INIT EVERYTHING*, ...). Dans ce TP, seule la partie "vidéo" est nécessaire.

Remarque : on peut combiner les paramètres à l'aide de l'opérateur « | » (ou binaire). Ex : *SDL_INIT_VIDEO | SDL_INIT_AUDIO*

Il est nécessaire de tester le retour de la fonction *SDL_Init()* : elle renvoie -1 si quelque chose s'est mal passé.

```
if ( SDL_Init ( SDL_INIT_VIDEO ) == -1 ) {
    fprintf(stderr, "Erreur d'initialisation de la SDL : %s\n",
            SDL_GetError());
    exit(EXIT_FAILURE); // On quitte le programme
}
```

- **Fermeture** : ce bout de code C est à placer à la fin de la fonction *main()* : il permet d'appeler la

fonction `SDL_Quit()` qui ne prend aucun paramètre. Cette fonction libère la mémoire allouée par la SDL.

```
SDL_Quit ();
```

- Votre fichier source C complet (appelez le `sd/1.c`) contient donc :

```
#include <stdio.h>
#include <stdlib.h>
#include <SDL/SDL.h>
int main()
{
    if ( SDL_Init ( SDL_INIT_VIDEO ) == -1 ) {
        fprintf(stderr, "Erreur d'initialisation de la SDL : %s\n",
            SDL_GetError());
        exit(EXIT_FAILURE); // On quitte le programme
    }
    /* Programme à placer ici */

    SDL_Quit ();
    return EXIT_SUCCESS;
}
```

1.2. Compilation et test du programme C minimal

```
$ gcc sdl1.c -lSDL -o sdl1
```

- L'option `-o` indique au compilateur de générer un fichier exécutable s'appelant `sd/1`. Si on ne met pas `-o sdl1`, le nom par défaut de l'exécutable sera `a.out`.
- Remarquez `-lSDL`, qui signale à l'éditeur de liens de charger la bibliothèque (*library*) SDL.
- Exécutez le programme :

```
$ ./sdl1
```

Et là, il ne se passe ... rien. En effet, vous n'avez encore rien mis dans votre programme : il faut, pour un programme graphique, au moins ouvrir une fenêtre. C'est le but du prochain paragraphe.

1.3. Ouverture d'une fenêtre

Pour l'ouverture de la première fenêtre de votre application (dite fenêtre mère ou «*main window*»), il faut indiquer à la SDL les caractéristiques graphiques de l'application : au minimum, il faut lui donner la taille de la fenêtre mère, la profondeur des couleurs et les options mémoire vidéo. On peut aussi ajouter d'autres caractéristiques : couleur de fond (par défaut noir), titre sur le cadre de la fenêtre (par défaut aucun),...

Nous allons présenter ces différentes caractéristiques :

- **taille de la fenêtre** (largeur *width*, hauteur *height*) : les deux dimensions (largeur : axe X, hauteur : axe Y) doivent être données en pixels^{*}.
- **profondeur de couleur** (*color depth*) : en nombre de bits par pixel (*bpp*). Les valeurs fréquemment utilisées sont 8 *bpp* (256 couleurs distinctes), 16 *bpp* (*high color* : 65536 couleurs distinctes), 24 *bpp* ou 32 *bpp* (*true color* : 16 millions de couleurs distinctes). Avec

* **Pixel** : contraction de «*picture element*» (élément d'image). C'est l'unité minimale d'une image matricielle numérique.

une faible profondeur de couleur (< 16 *bpp*), les valeurs des couleurs sont stockées dans une carte de couleur appelée une palette (principe des couleurs indexées). Sinon, on utilise les couleurs «directes» (*direct color*), selon le codage RGB (*Red Green Blue*). Dans certains codages «directs», un ou plusieurs bits servent à coder la transparence (canal *Alpha*) : c'est le cas notamment pour la profondeur 32 *bpp* où le codage RGBA (ou ARGB) réserve 8 bits pour le canal *Alpha*.

Le choix de la profondeur de couleur est lié aux limites matérielles de la mémoire : par exemple la mémoire vidéo de la *Nintendo DS* ne permet pas plus de 16 *bpp*. Les mémoires vidéo des PC actuels sont suffisamment dimensionnées pour qu'on ne se pose pas la question: on choisira donc la valeur de 32 *bpp*. Pour cette valeur, chaque couleur (R, G ou B) est codée sur 8 bits (0 à 255, ou 0x00 à 0xFF), le canal *Alpha* (s'il est défini) prenant aussi 8 bits (0 = transparent, 255 = opaque).

- **options vidéo** : plusieurs constantes permettent de définir le mode vidéo, notamment le type de mémoire «graphique» utilisé et le mode de fenêtrage:
 - type de mémoire «graphique» : `SDL_SWSURFACE` si on veut utiliser la mémoire «système» (*software*), `SDL_HWSURFACE` si on veut utiliser la mémoire «vidéo» (*hardware*), c'est-à-dire celle de la carte graphique. Si c'est cette dernière qui est choisie, on peut ajouter l'option `SDL_DOUBLEBUF` (*double buffering* : évite les problèmes de scintillement d'image, mais consomme le double d'espace mémoire).
 - mode de fenêtrage : `SDL_FULLSCREEN` si on veut que l'application soit en plein écran, `SDL_RESIZABLE` si on veut autoriser le redimensionnement de la fenêtre, `SDL_NOFRAME` si on veut supprimer le cadre (pas de barre de titre, de bouton de fermeture, de mise en icône, etc...). Le mode «*full screen*» est automatiquement sans cadre.

La fonction **`SDL_SetVideoMode()`** permet d'initialiser les caractéristiques que nous venons de voir. Elle prend en paramètres la largeur et la hauteur de la fenêtre, la profondeur de couleur et une combinaison d'options vidéo.

Elle renvoie un pointeur sur une structure **`SDL_Surface`** (cette structure permet l'accès à une zone de la mémoire «graphique», appelée «surface»), pointant ici vers le «*framebuffer*», c'est-à-dire la zone mémoire «graphique» de l'application SDL. Donc la fonction **`SDL_SetVideoMode()`** crée une surface qui correspond à la fenêtre mère de l'application. Si quelque chose s'est mal passé, la fonction renvoie le pointeur `NULL`.

Ex : création d'une fenêtre de 640x480 pixels, 32 bits par pixel, redimensionnable, utilisant la mémoire «vidéo» en «*double buffering*» .

```
SDL_Surface *ecran = NULL;
ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_RESIZABLE
                        | SDL_DOUBLEBUF);
```

Remarque : la zone mémoire «graphique» allouée lors de l'appel à **`SDL_SetVideoMode()`** est libérée lors de l'appel à **`SDL_Quit()`**.

- **Caractéristiques complémentaires de la fenêtre :**

D'autres caractéristiques peuvent être fixées pour la fenêtre, notamment le titre et la couleur de fond, mais il faut appeler d'autres fonctions.

- Donner un **titre** à la fenêtre : le titre apparaîtra sur le haut du cadre (sauf si les options `SDL_FULLSCREEN` ou `SDL_NOFRAME` ont été positionnées). Pour cela, on fait un appel à la fonction **`SDL_WM_SetCaption()`** avec le titre en tant que premier paramètre, de type «chaîne de caractères». Le second paramètre est le titre de la fenêtre quand elle est en icône (si `NULL`, le titre de l'icône est identique au premier). Ex :

```
SDL_WM_SetCaption("Ma première fenêtre !", NULL);
```

- Donner une **couleur de fond** (*background color*) à la fenêtre : par défaut le fond est noir. Une couleur doit être indiquée en tant que valeur RGB. Pour cela, on fait un appel à la fonction `SDL_FillRect()` avec la surface en tant que premier paramètre, de type `SDL_Surface`. Le second paramètre est inutilisé (NULL) et le troisième paramètre est la couleur. La couleur peut être donnée directement en codage RGB hexadécimal (par exemple `0x00ff00` pour vert), ou à l'aide de la fonction `SDL_MapRGB()` (par exemple `SDL_MapRGB(ecran->format, 0, 255, 0)` pour vert). Ex :

```
SDL_FillRect(ecran, NULL, 0x00ff00);  
SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 0, 255, 0));
```

Remarque : la couleur de la fenêtre ne sera pas immédiatement modifiée lors de l'appel à `SDL_FillRect()`. Il faudra appeler une fonction de mise à jour (cf. §3.3).

- **Code à rajouter** au fichier source C `sd1.c` du §2.1 :

```
/* Programme à placer ici */  
SDL_Surface *ecran = NULL;  
ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_RESIZABLE  
                        | SDL_DOUBLEBUF);  
if (ecran == NULL) {  
    fprintf(stderr, "Impossible de charger le mode vidéo : %s\n",  
            SDL_GetError());  
    exit(EXIT_FAILURE);  
}  
/* Titre fenetre */  
SDL_WM_SetCaption("Ma première fenêtre !", NULL);  
  
/* Fond d'ecran vert sombre */  
SDL_FillRect(ecran, NULL, 0x326E00);  
// La meme couleur definie d'une autre façon :  
//SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 50, 110, 0));
```

- **Compilez et testez** (comme en §2.2) :

La fenêtre apparaît et disparaît aussitôt ! En effet, la fonction `SDL_SetVideoMode()` est immédiatement suivie de `SDL_Quit()`. La méthode habituelle pour régler ce problème est de rajouter une boucle d'attente infinie avant d'appeler `SDL_Quit()`. Vous pouvez donc rajouter une ligne à la suite du code précédent :

```
while (1) { }
```

- **Re-compilez et re-testez** :

Maintenant la fenêtre apparaît et reste affichée. Il y a cependant deux problèmes :

- on ne peut pas fermer la fenêtre, ni en cliquant sur le "x" de fermeture en haut du cadre, ni en tapant `Ctrl/C`. En effet, le programme reste engagé dans sa boucle infinie, et la seule commande qui peut l'arrêter est un « `kill -9` » bien senti !
- la couleur de fond de la fenêtre est noire et pas verte.

Nous allons dans le paragraphe suivant traiter le premier problème, à l'aide d'une boucle d'attente d'évènements. Le second problème sera traité ensuite (§3.3).

1.4. La boucle d'attente d'évènements

Toute application graphique, quels que soient la plate-forme, le langage ou la bibliothèque graphique, gère les actions de l'utilisateur ou «évènements» (frappe clavier, clic ou déplacement souris, ouverture/fermeture fenêtre,...) à l'aide d'une **boucle d'évènements** (*Event Loop*, appelée souvent aussi *Main Loop*). Dans certains langages ou bibliothèques graphiques (ex: *Java*), la boucle d'évènements est sous-jacente et cachée, dans d'autres elle doit être écrite explicitement : c'est le cas de la SDL.

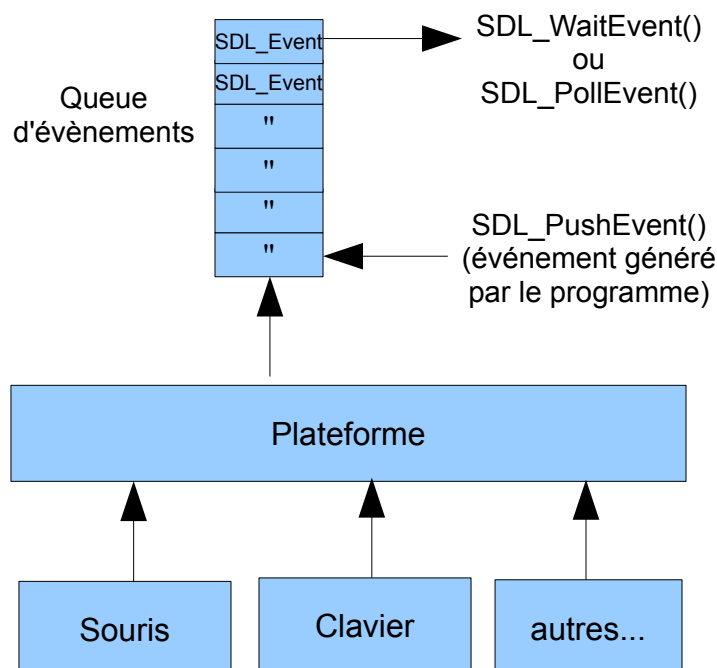
La boucle d'attente d'évènements va remplacer notre `while(1)` du paragraphe précédent. Elle est basée aussi sur le principe de la boucle infinie, mais, à chaque tour de boucle, le programme regarde si un évènement a eu lieu, et le traite si nécessaire. En général, un programme graphique passe la majeure partie de son temps dans cette boucle d'attente d'évènements.

La SDL définit un certain nombre de «types» d'évènements (dans le fichier *SDL_events.h*), dont les plus courants sont :

- `SDL_KEYDOWN` : appui d'une touche du clavier,
- `SDL_MOUSEMOTION` : déplacement de la souris,
- `SDL_MOUSEBUTTONDOWN` : appui d'un bouton de la souris,
- `SDL_MOUSEBUTTONUP` : relâchement d'un bouton de la souris,
- `SDL_QUIT` : clic sur le "x" de fermeture (ou raccourci clavier équivalent).

Les évènements, dès qu'ils sont détectés par la plateforme sous-jacente (ex: *XWindow* sous *Linux*), sont mis dans une queue (FIFO). La SDL permet de lire le premier évènement et de le sortir de la queue, à l'aide de la fonction *SDL_WaitEvent()* (bloquante) ou *SDL_PollEvent()* (non bloquante). Ces deux fonctions ont un paramètre de sortie de type *SDL_Event*, qui stocke toutes les informations sur l'évènement : type (cf. plus haut), touche de la souris et position x,y pour un clic de souris, symbole de la touche pour un évènement «clavier», etc...

Le type *SDL_Event* est une *union* (sorte de «super structure» en C), qui possède toujours un champ *type*, et qui se décline en autant de structures C distinctes que de types d'évènements : par exemple, un évènement de type `SDL_MOUSEBUTTONUP` sera décrit par une sous-structure nommée *button* (qui contient elle-même les champs *button* : `SDL_BUTTON_LEFT`, `SDL_BUTTON_MIDDLE` ou `SDL_BUTTON_RIGHT`, et les champs *x* et *y* : position où a eu lieu le clic dans la fenêtre).



- Notre problème est ici de récupérer l'évènement **SDL_QUIT**, puis de sortir de la boucle d'attente d'évènements et d'appeler la fonction *SDL_Quit()* pour terminer l'application «proprement». Pour ce faire, le bout de code C suivant est à mettre à la place du `while(1) { }`.

```
int continuer = 1;
SDL_Event event;
while ( continuer ) {
    SDL_WaitEvent ( &event ); // attente d'un évènement
    switch ( event.type ) {
        case SDL_QUIT :
            continuer = 0;
            break;
    }
}
```

- **Re-compilez et testez :**
Maintenant la fenêtre peut être fermée en cliquant sur "x" de fermeture, ou en tapant «Alt F4» au clavier quand la fenêtre est sélectionnée (quand elle a le «focus»).

Vous pouvez voir aussi :

- que la fenêtre a une couleur de fond noire au départ et n'est colorée en vert que si un évènement implique son retraçage (masquage par une autre fenêtre ou mise en icône par exemple). Pour forcer le retraçage, on peut appeler la fonction *SDL_Flip()* que nous verrons en §3.3.
- que si vous redimensionnez la fenêtre en l'agrandissant, seule la surface initiale est colorée en vert. C'est le fonctionnement «normal» avec la SDL, et il ne peut être amélioré qu'en rajoutant du code assez complexe (récupérer l'évènement **SDL_VIDEORESIZE**, puis recréer la *main window*).

2. Afficher une image

La SDL ne propose à la base que le chargement d'images au format *Bitmap* (extension ".bmp"). Pour charger d'autres types d'images (*JPEG*, *PNG*, *GIF*, ...) il faut une extension de la SDL, la bibliothèque *SDL_Image*. Dans ce TP, nous ne traiterons que des images *Bitmap*.

Par définition, un «**bitmap**» est une image matricielle c'est-à-dire une image numérique qui se compose d'un tableau à 2 dimensions de pixels (le terme «*pixmap*» serait plus approprié, mais il est moins employé).

Le format *Bitmap* (appelé aussi *Windows Bitmap*) a été développé par *Microsoft* et *IBM*, mais il est libre de droits et bien documenté. Il est assez multiforme : selon la description donnée dans l'entête de fichier, les pixels peuvent être codés sur 1, 4, 8, 16, 24 ou 32 bits, les couleurs peuvent être codées différemment, un canal *Alpha* peut être présent ou pas, etc.... Certaines caractéristiques du fichier doivent être indiquées explicitement à la SDL : c'est le cas de la présence et du codage d'un canal *Alpha*.

Vous trouverez, avec le sujet du TP, 8 fichiers *bmp* : *fleur1a.bmp*, *fleur2a.bmp*,..., *fleur8a.bmp*. Nous allons, dans un premier temps, charger et afficher l'image *fleur1a.bmp*.

2.1. Chargement d'une image

Elle est réalisée par la fonction *SDL_LoadBMP()*, qui prend en paramètre le nom du fichier *bmp* et renvoie une «surface» (pointeur sur *SDL_Surface*) ou NULL en cas de problème (par exemple le

fichier demandé n'existe pas dans le répertoire courant). Ex :

```
SDL_Surface *fleur;
/* Chargement fichier BMP dans une surface */
fleur = SDL_LoadBMP("fleur1a.bmp");
if (fleur == NULL) {
    fprintf(stderr, "Impossible de charger le fichier bmp: %s\n",
            SDL_GetError());
}
```

Vous mettrez ce bout de code C avant la boucle d'attente d'évènements. Si vous compilez et testez, vous vous apercevrez qu'aucune image ne s'affiche dans la fenêtre. En effet, il faut, après la création de la surface liée à cette image, réaliser le «*blitting*» de cette surface sur la surface de la fenêtre mère : cette technique est expliquée dans le paragraphe suivant.

Attention ! Contrairement à la surface de la fenêtre mère, qui est libérée automatiquement par l'appel à *SDL_Quit()*, chaque surface correspondant à une image doit être libérée explicitement quand l'image n'est plus utile, à l'aide de la fonction *SDL_FreeSurface()*.

2.2. «*Blitting*» de l'image

Le terme *BLIT* est un acronyme de *B*lock *I*mage *T*ransfer. Le «*blitting*» est une opération, classique en application graphique 2D, qui combine plusieurs *bitmaps* en un seul, à l'aide d'opérateurs appelés «opérateur *raster*» (*Raster Operator* ou *ROP*).

Il s'agit généralement de combiner, pixel par pixel, un *bitmap* source (souvent une petite image 2D appelée «*sprite*» dans le monde du jeu vidéo) avec un *bitmap* destination (souvent l'image de fond). L'opérateur *raster* le plus simple «écrase» le pixel de destination par le pixel source, mais la gestion de la transparence implique des opérateurs plus complexes.

Les cartes graphiques modernes réalisent ces opérations de «*blitting*» en parallèle sur tous les pixels concernés, grâce à des circuits «*hardware*» dédiés, ce qui est beaucoup plus rapide que de traiter par programme un *bitmap* pixel par pixel.

La SDL réalise le «*blitting*» d'une surface sur une autre par la fonction *SDL_BlitSurface()*, qui prend 4 paramètres :

- la surface source (ici celle créée par *SDL_LoadBMP()*),
- un paramètre (rectangle source) mis à NULL si toute la surface source est copiée,
- la surface destination (ici la fenêtre mère),
- un rectangle indiquant la position (dans l'image destination) où l'image source sera copiée (seule la position x,y, en pixels, du coin en haut à gauche du rectangle est utilisée). Si ce paramètre vaut NULL, la position est 0,0.

Le code C suivant fait suite à celui du paragraphe précédent. Il permet de «blitter» une image dans le coin supérieur gauche de la fenêtre :

```
SDL_BlitSurface(fleur, NULL, ecran, NULL);
```

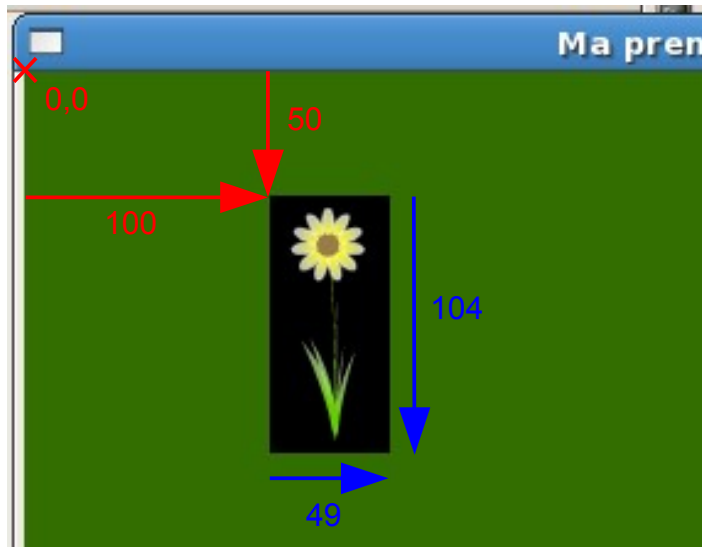
Dans le cas, plus général, où on veut afficher l'image à une position particulière, il faut créer un rectangle (type *SDL_Rect*) et lui donner la position (x, y) désirée, puis «blitter» l'image sur ce rectangle :

```
SDL_Rect rect_fleur;
rect_fleur.x = 100;
rect_fleur.y = 50;
```



```
SDL_BlitSurface(fleur, NULL, ecran, &rect_fleur);
```

Notez que la position de référence d'une image est toujours son coin supérieur gauche. Si on souhaite positionner une image à des coordonnées dépendantes de sa taille, il faut faire appel aux champs *w* et *h* (largeur et hauteur) de la surface associée à l'image. Dans l'exemple ci-dessous *fleur->w* et *fleur->h* valent respectivement 49 et 104 pixels.



Si vous compilez et testez, vous vous apercevrez que l'image ne s'affiche que si un événement implique le retraçage de la fenêtre, problème déjà signalé au §2. Nous allons maintenant voir comment forcer le rafraichissement de la fenêtre.

2.3. Mise à jour (rafraichissement) de la fenêtre

Deux fonctions SDL peuvent réaliser le rafraichissement de la fenêtre : *SDL_Flip()* et *SDL_UpdateRect()*.

- La fonction ***SDL_UpdateRect()*** met à jour le rectangle donné en paramètre (l'ensemble de la fenêtre par défaut). C'est un rafraichissement «*software*», qui retrace tous les pixels de la zone demandée.
- La fonction ***SDL_Flip()*** met à jour l'ensemble de la fenêtre en une seule opération, si le mode vidéo est *SDL_HWSURFACE* | *SDL_DOUBLEBUF*. C'est un rafraichissement «*hardware*», qui bascule très rapidement (*flip*) de la mémoire «vidéo» contenant la fenêtre actuellement affichée (*front buffer*), à la seconde mémoire «vidéo» (*back buffer*) contenant l'image mise à jour en temps réel (c'est-à-dire après appel de chaque fonction graphique). Après ce basculement c'est le «*back buffer*» qui passe «*front buffer*» et inversement. L'avantage d'utiliser la fonction *SDL_Flip()* est que, si on n'est pas en mode vidéo «*hardware+double buffering*», elle appelle automatiquement la mise à jour «*software*» de la fenêtre.

Chaque fois que, dans votre application, il est nécessaire que l'écran soit immédiatement rafraichi, vous devez donc ajouter cette ligne de code :

```
/* Rafraichissement */  
SDL_Flip( ecran );
```

2.4. Gestion de la transparence de l'image

Sans transparence, une image s'affiche toujours en tant que rectangle. C'est ce qui apparaît dans l'exemple du paragraphe précédent. Il est cependant très fréquent qu'on veuille afficher uniquement la partie significative de l'image (la fleur dans l'exemple), sans le fond. L'idée est donc de rendre «transparent» le fond de l'image. Pour cela, il existe deux méthodes :

- Si l'image source n'a **pas de canal Alpha**, il faut qu'elle ait une **couleur de fond unie** et unique et indiquer à la SDL que cette couleur doit être rendue transparente. Il faut, bien entendu, au moment où on crée l'image, choisir une couleur de fond qui ne soit pas (ou très peu) utilisée dans la partie utile de l'image, car tous les pixels de cette couleur seront rendus transparents.

La fonction qui indique à la SDL de rendre une couleur transparente pour une image est `SDL_SetColorKey()`, avec la surface source en tant que premier paramètre. Le second paramètre est la constante `SDL_SRCCOLORKEY` et le troisième paramètre est la couleur. La couleur peut être donnée directement en codage RGB hexadécimal ou à l'aide de la fonction `SDL_MapRGB()` (comme en §2.3).

Dans l'exemple de l'image *fleur1a.bmp*, la couleur de fond est noire (0x000000), le code C à insérer, avant `SDL_BlitSurface()`, sera donc une des deux instructions suivantes :

```
SDL_SetColorKey ( fleur, SDL_SRCCOLORKEY, 0x000000 );
SDL_SetColorKey ( fleur, SDL_SRCCOLORKEY,
                  SDL_MapRGB( fleur->format, 0, 0, 0 ) );
```

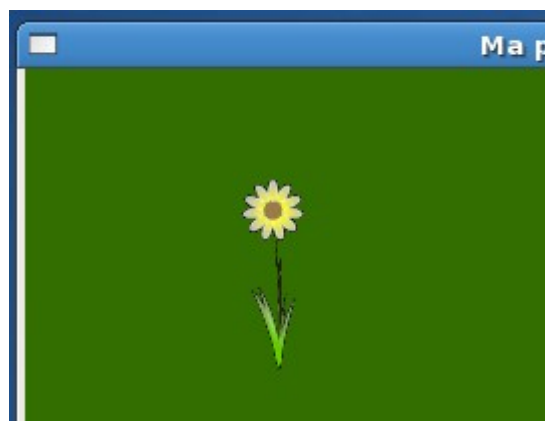
- Si l'image source a **un canal Alpha** (cf. §2,3), il faut indiquer à la SDL comment lire ce canal *Alpha* (c'est-à-dire comment le canal *Alpha* est codé pour chaque pixel, ce qui dépend du format du fichier image).

La fonction qui indique à la SDL que l'image source a un canal *Alpha* est `SDL_SetAlpha()`, avec la surface source en tant que premier paramètre. Le second paramètre est la constante `SDL_SRCALPHA` et le troisième paramètre est inutilisé dans ce cas. Il faut ensuite préciser la position de ce canal *Alpha* dans chaque pixel, en donnant un masque (*Amask*), et permettre sa lecture en donnant un décalage (*Ashift*).

Dans l'exemple de l'image *fleur1a.bmp*, les pixels de la partie significative de l'image (la fleur) ont un canal *Alpha* à 255 (opaque), tandis que le fond de l'image (noir) a un canal *Alpha* à 0 (transparent). Le canal *Alpha* est codé sur les 8 premiers bits/32 (masque : 0xFF000000, décalage : 24 bits). Le code C à insérer, avant `SDL_BlitSurface()`, sera donc :

```
SDL_SetAlpha(fleur, SDL_SRCALPHA, 0);
fleur->format->Amask = 0xFF000000;
fleur->format->Ashift = 24;
```

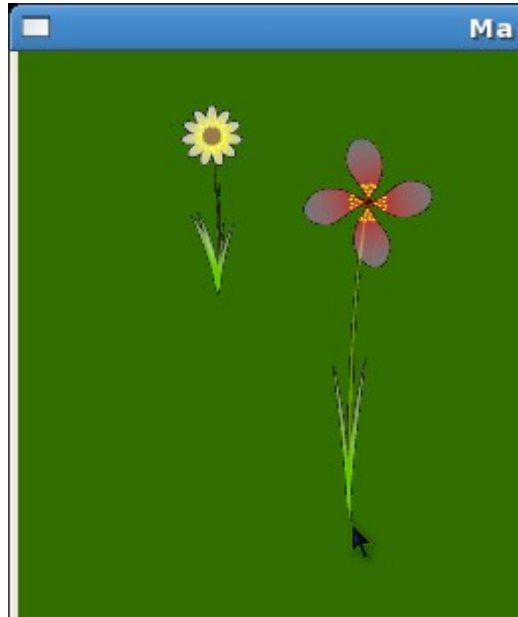
Vous devez utiliser soit la transparence stockée dans l'image elle-même (canal *Alpha*), soit, à défaut, la méthode de la couleur de fond. Ici, l'image *fleur1a.bmp* qui vous est fournie permet d'utiliser les deux méthodes, pour les besoins du TP. Quelle que soit la méthode employée, voici l'image qui doit s'afficher :



3. Réagir aux actions de l'utilisateur

Nous avons vu au §2.4 que les actions de l'utilisateur (frappe clavier, clic ou déplacement souris,...) sont reçus par l'application en tant qu'«événements». Nous n'avons traité en §2.4 qu'un seul type d'évènement, la fermeture de la fenêtre, et nous allons maintenant traiter un évènement de type «clic de souris».

Le but de l'exercice est d'afficher une fleur à l'endroit où l'utilisateur clique. A chaque clic, ce sera une fleur différente qui s'affichera (8 images distinctes sont fournies : *fleur1a.bmp*, *fleur2a.bmp*,..., *fleur8a.bmp*). Pour que l'affichage soit plus «naturel», il faut que la fleur soit positionnée de telle sorte que le bas de la tige de la fleur se place là où l'utilisateur a cliqué. Ex. d'affichage :



Pour réaliser cela, il faut :

- récupérer la position du pointeur de la souris au moment du clic,
- ouvrir l'image suivante de la liste des 8 images (avec *SDL_LoadBMP()*),
- la positionner correctement, avec la fonction *SDL_BlitSurface()*, selon les spécifications ci-dessus (clic = bas de la tige). On considérera que le bas de la tige est toujours situé verticalement 10 pixels au-dessus du bas de l'image et horizontalement au milieu de l'image.

3.1. L'évènement «clic de souris»

Nous allons, dans la boucle d'attente d'évènements, ajouter le cas où le type d'évènement est *SDL_MOUSEBUTTONDOWN*. Dans ce cas, on obtient la position du pointeur de la souris au moment du clic à l'aide des champs *event.button.x* et *event.button.y*.

La boucle d'attente d'évènements devient donc (code non complet : ...) :

```
int continuer = 1;
SDL_Event event;
while ( continuer ) {
    SDL_WaitEvent ( &event ); // attente d'un évènement
    switch ( event.type ) {
        case SDL_QUIT :
            continuer = 0;
            break;
        case SDL_MOUSEBUTTONDOWN :
```

```

        /* Chargement d'un des 8 fichiers fleur?a.bmp */
        fleur = ...
        ...
        rect_fleur.x = event.button.x + ...;
        rect_fleur.y = event.button.y + ...;
        SDL_BlitSurface ( fleur, NULL, ecran, &rect_fleur );
        SDL_Flip ( ecran );
        SDL_FreeSurface ( fleur );
        break;
    }
}

```

Testez d'abord en chargeant à chaque fois le même fichier *fleur1a.bmp*. Si ça fonctionne correctement, vous chargerez ensuite un fichier différent à chaque fois (cf. §suivant).

3.2. Génération de noms de fichiers successifs

Pour générer les noms de fichiers *fleur?a.bmp*, «*a*» étant un chiffre de 1 à 8, vous pouvez utiliser une méthode classique en C, appelée «méthode *sprintf*». C'est la réponse classique au problème : comment introduire un entier dans une chaîne de caractères?

- **La méthode *sprintf* :**

Imaginez que vous vouliez générer, en boucle, plusieurs chaînes de caractères contenant «Numéro 1», «Numéro 2», «Numéro 3», etc. jusqu'à «Numéro 10».

Le plus simple est de faire une boucle faisant varier un indice *i* de 1 à 10. Le problème est alors d'introduire l'entier *i* à la suite de la chaîne de caractère fixe «Numéro ». Il existe une fonction standard, *sprintf()*, qui permet de faire ça.

La fonction *sprintf()* fonctionne comme *printf()*, mais elle prend un paramètre en plus (le premier) : une chaîne de caractères (préalablement allouée). Au lieu d'afficher un texte à l'écran, comme *printf()*, elle met ce texte dans la chaîne de caractères. Ex (différent du TP) :

```

char chaine[10];
int i = 1;
sprintf ( chaine, "Numéro %d", i );
/* Utilisation de chaine */
...
i++;

```

- **A faire dans le TP :** Vous devez créer le code C pour afficher successivement les 8 fleurs différentes, puis revenir sur la première quand l'utilisateur clique une 9ème fois, etc...

3.3. Génération aléatoire de noms de fichiers

Au lieu de parcourir successivement les 8 fichiers *fleur?a.bmp*, ce qui fait afficher les mêmes fleurs régulièrement, la dernière partie de ce TP consiste à afficher aléatoirement, à chaque clic, une fleur quelconque parmi les 8 possibles. Pour cela, il existe les fonctions *rand()* et *srand()* (déclarées dans *stdlib.h*).

- **La fonction *rand()* :**

La fonction *rand()*, qui ne prend pas de paramètres, renvoie un entier (pseudo-)aléatoire entre 0 et *RAND_MAX*. *RAND_MAX* (défini dans *stdlib.h*) peut être différent selon le système, mais sous *Linux* il vaut 2147483647.

Ici, nous voulons un entier (grossièrement) aléatoire compris entre 1 et 8. Nous pouvons utiliser l'instruction C suivante :

```
/* i : entier aléatoire entre 1 et 8 */  
i = rand() % 8 +1;
```

Cependant, si vous exécutez le programme plusieurs fois de suite, vous vous apercevrez que vous avez toujours la même succession de fleurs. Ceci provient du fait que la fonction *rand()* utilise toujours la même «graine» pour la génération de chaque nouvelle séquence de nombres pseudo-aléatoires. La «graine» utilisée par la fonction *rand()* est fournie par la fonction *srand()*, ou bien, à défaut d'appeler la fonction *srand()*, elle vaut 1, ce qui était le cas dans notre code.

- **La fonction *srand()* :**

La fonction *srand()* prend en paramètre un entier positif et ne renvoie rien. L'appel de *srand()* fournit une «graine» pour la génération par *rand()* d'une séquence de nombres pseudo-aléatoires. Donc, en général, la fonction *srand()* n'est appelée qu'une fois, en début de programme, contrairement à *rand()*, qui sera appelée plusieurs fois (à chaque fois qu'on veut un nombre aléatoire).

Pour que la «graine» soit différente à chaque appel de *srand()*, il faut que son paramètre entier soit différent. En général, pour s'assurer de cette différence, on fournit à *srand()*, le temps (en secondes) écoulé depuis le 1er janvier 1970 à 0H. C'est l'information renvoyée par la fonction *time()*, à laquelle on passe le paramètre *NULL* (déclarée dans *time.h*).

Nous pouvons donc écrire, une seule fois dans le programme (pas dans une boucle!) l'instruction C suivante :

```
/* graine pour rand() */  
srand ( time(NULL) );
```