# TP N°8 Application graphique avec SDL (2)


**Objectifs du TP** : poursuivre l'apprentissage du développement d'applications graphiques en C à l'aide de la bibliothèque SDL. Ce TP est inspiré de l'application *xroach*, créée en 1991 pour les environnements *XWindow* sous *Unix*. Cette application amusante affiche sur le fond d'écran des cafards (*roach*), qui vont se cacher sous les fenêtres affichées à l'écran. L'adaptation de ce thème à la SDL, dans une fenêtre, va vous demander de :

- Charger et afficher une image (*bitmap*) de fond,
- Charger et afficher les images (*bitmap*) de l'insecte sous différentes orientations («*sprites*»),
- Gérer les déplacements des cafards : tester les obstacles (bord de la fenêtre, autres cafards), changer de direction,
- Gérer la vitesse des cafards,
- Mettre en place une action de «*drag an drop*» pour un objet graphique,
- Adapter la boucle d'attente d'évènements du TP précédent.

Contrairement au TP précédent, il vous sera demandé de séparer le code en fonctions, en fichiers séparés : les fichiers *sdl2.c*, *fonctions.c*, *sdl2.h*, accompagnés d'un *Makefile*.

## 1. L'initialisation et le chargement de l'image de fond

On vous demande de créer une fonction *init( )* qui fera l'initialisation et la chargement de l'image de fond et dont le prototype sera :

```
SDL_Surface *init ( char * imgFond_filename, SDL_Surface **pFond );
```

La fonction *init( )* va, successivement, initialiser la SDL (cf. TP SDL 1), charger l'image de fond, puis créer la fenêtre mère avec une taille correspondant à l'image de fond et les options vidéo SDL_HWSURFACE et SDL_DOUBLEBUF (cf. TP SDL 1). Elle a un paramètre d'entrée : le nom du fichier *Bitmap* contenant l'image de fond. Elle retourne le pointeur sur la surface de la fenêtre mère et fournit un paramètre de sortie : la surface contenant l'image de fond.

Secondairement, non demanderons aussi à notre fonction *init( )* de fournir une «graine» pour la génération de nombres pseudo-aléatoires (fonction *srand()* cf. TP SDL 1).

Nous allons détailler uniquement ci-dessous la partie de code concernant l'image de fond, les autres parties ayant été vues au TP précédent.

### 1.1. Chargement et «*blitting*» d'une image de fond (dans le fichier *fonctions.c*)

- Elle se charge comme une image *Bitmap* quelconque (cf. TP SDL 1) :

```
SDL_Surface *fond;
fond = SDL_LoadBMP(imgFond_filename);
if (fond == NULL) {
     fprintf(stderr, "Impossible de charger le fichier %s: %s\n",
                          imgFond_filename, SDL_GetError());
     exit(EXIT_FAILURE); // On quitte le programme
}
```

- Création de la fenêtre mère : la différence avec le code C vu au TP précédent est que la taille (largeur, hauteur) de la fenêtre doit être la même que celle de l'image de fond, donc :

```
SDL_Surface *ecran;
ecran = SDL_SetVideoMode(fond->w, fond->h, 32, SDL_HWSURFACE|
                                                SDL_DOUBLEBUF);
/* Test si ecran est NULL */
...
```

- «*Blitting*» de l' image de fond :

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

- Retour de la surface correspondant à l'image de fond (`fond`) et à la fenêtre mère (`ecran`) :

```
*pFond = fond;
return ecran;
```

## 1.2. Initialisation et arrêt dans le *main()*

- Votre fichier source C principal complet (*sdl2.c*) contient donc maintenant :

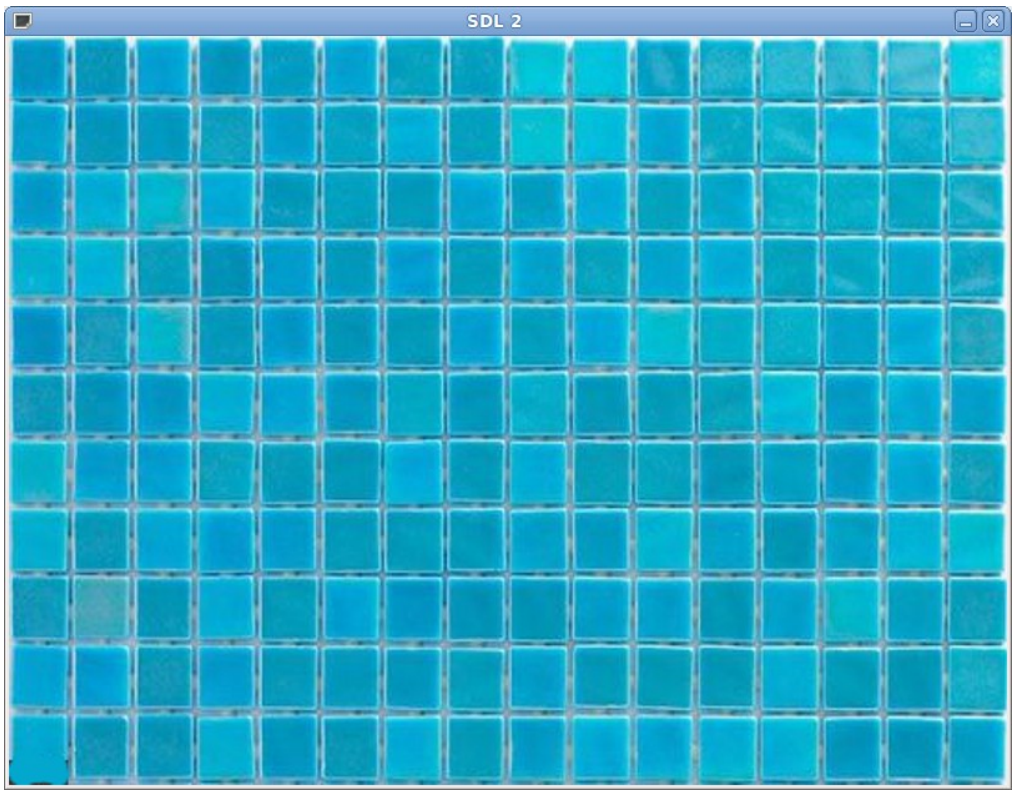
```
#include "sdl2.h"

int main()
{
SDL_Surface *floor = NULL;
SDL_Surface *ecran = init ( "floor.bmp", &floor );
SDL_Flip(ecran);

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

## 1.3. Compilation et test du programme C minimal

Il faut, bien entendu, créer un *Makefile* qui fait référence aux fichiers *sdl2.c*, *fonctions.c* et *sdl2.h*. Nous avons appelé l'exécutable *sdl2*.

```
$ make
$ ./sdl2
```

Si tout a bien fonctionné, vous devez obtenir l'affichage suivant :




## 2. Chargement des «*sprites*» et initialisation d'un cafard (*roach*)

Nous allons d'abord définir quelques constantes et une structure C appelée `struct Roach` (typedef `Roach`), dans le fichier *sdl2.h* :

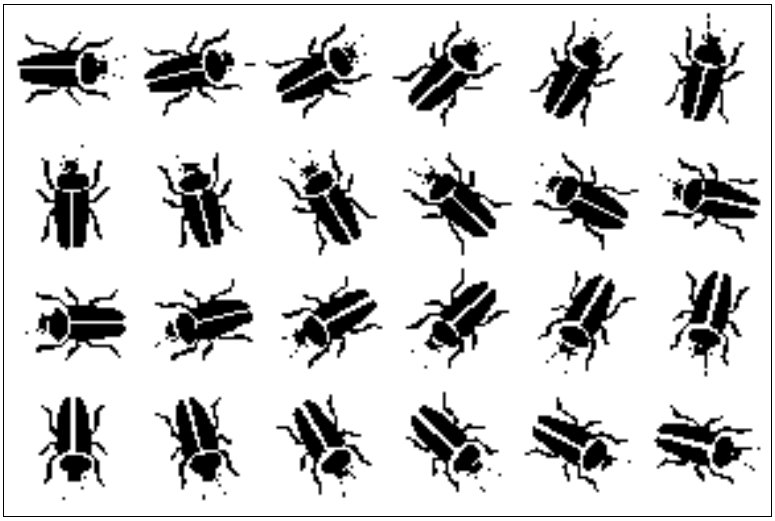
```
#define ROACH_ORIENTATIONS 24    /* nbre d'orientations distinctes */
#define ROACH_ANGLE        15    /* angle (deg) entre orientations */
#define ROACH_WIDTH        48    /* pixels */
#define ROACH_HEIGHT       48    /* pixels */
#define NB_SPRITES_P_LINE  6     /* dans bitmap des sprites */
#define ROACH_SPEED        20.0  /* pixels/tour */
#define MAX_STEPS          50    /* nb de deplacements max avant
                                    changement d'orientation */

typedef struct {
    SDL_Surface *sprites;
    int dir; // orientation (0 à ROACH_ORIENTATIONS-1)
    int x; // pixels
    int y; // pixels
    int hidden; // 0 : visible, 1 : caché
    float angle; // orientation en radians
    int turnLeft; // 0 : droitier, 1 : gaucher
    int steps;
} Roach;
```

## 2.1. Image («*sprite*») d'un cafard

Comme l'insecte peut se déplacer dans plusieurs directions, il faut plusieurs images distinctes pour couvrir 360° (24 images, avec un angle de 15° entre chaque orientation). Chaque image fait 48x48 pixels. En voici un exemple (orientation 45°) :

L'image n'a pas de canal *Alpha*, mais le fond blanc peut être rendu transparent (cf. TP SDL 1).

L'application pourrait charger (avec 24 appels de *SDL_LoadBMP()*) les 24 images nécessaires, mais, pour améliorer le temps de chargement, les 24 images ont été regroupées en une seule : c'est ce qu'on appelle un «*sprite sheet*». Cette image unique (*roach.bmp*) est la suivante :



Le premier «*sprite*» a une orientation 0° (convention du cercle trigonométrique), le second 15°, etc... jusqu'au dernier qui est orienté au 345°. Pour afficher un seul «*sprite*» (fonction *SDL_BlitSurface()*), il faudra donner un rectangle source de 48x48 pixels (*bounding box*) correspondant à la position du «*sprite*» dans l'image entière (cf. plus bas fonction *DrawRoach()*).

- La surface contenant le «*sprite sheet*» sera créée par la **fonction *LoadSprites()*** (dans *fonctions.c*), dont le prototype est le suivant :

```
SDL_Surface *LoadSprites ( char * sprites_filename );
```

La fonction *LoadSprites()*, que vous devez écrire, va, successivement, charger l'image (avec *SDL_LoadBMP()*), indiquer que la couleur de fond blanche doit être transparente (avec *SDL_SetColorKey()*, cf TP SDL 1), puis retourner la surface créée.

Voici l'appel de la fonction *LoadSprites()* dans le *main()* :

```
SDL_Surface *sprites = LoadSprites ( "roach.bmp" );
```

Cet appel doit être placé après l'appel de fonction *init().*

- Le tracé d'un cafard sera réalisé par la **fonction *DrawRoach()*** (dans *fonctions.c*), dont le prototype est le suivant :

```
void DrawRoach ( Roach roach, SDL_Surface *ecran );
```

La fonction *DrawRoach()*, qui vous est fournie ci-dessous, consiste à déterminer le rectangle de l'image source contenant juste (*bouding box*) l'image correspondant à l'orientation du cafard. Il suffit ensuite de «blitter» ce rectangle sur la surface de fond (`ecran`) à la position de destination (c'est-à-dire la position x,y du cafard).

Voici le code source de la fonction *DrawRoach()*, dans le fichier *fonctions.c* :

```
void DrawRoach ( Roach roach, SDL_Surface *ecran )
{
    SDL_Rect rect_src; // Rectangle source
    SDL_Rect rect_dest; // Rectangle destination

    rect_src.x = ( roach.dir % NB_SPRITES_P_LINE ) * ROACH_WIDTH;
    rect_src.y = ( roach.dir / NB_SPRITES_P_LINE ) * ROACH_HEIGHT;
    rect_src.w = ROACH_WIDTH;
    rect_src.h = ROACH_HEIGHT;

    rect_dest.x = roach.x;
    rect_dest.y = roach.y;

    SDL_BlitSurface(roach.sprites, &rect_src, ecran, &rect_dest);
}
```

Voici l'appel de la fonction *DrawRoach()* dans le *main()*. Ce code doit être inclus dans la boucle `while` d'attente d'évènements. En effet, pour l'instant le tracé d'un cafard est fixe, mais bientôt nous allons le faire se déplacer (§3) :

```
     DrawRoach ( roach, ecran );
     SDL_Flip(ecran);
```

Cet appel doit être placé après l'appel de fonction *init().*

## 2.2. Initialisation (création) d'un cafard

Avant d'afficher un cafard, il faut le créer. La création d'un cafard va consister à donner des valeurs initiales à tous les champs de la structure `Roach`.

La plupart des champs numériques (`x`, `y`, `dir`, `steps`, `turnLeft`) seront initialisés à une valeur entière aléatoire. Pour améliorer la lisibilité du programme, on va créer une fonction *RandInt()* qui va renvoyer une valeur pseudo-aléatoire entière adaptée selon son paramètre d'entrée. Voici le code source de la fonction *RandInt()*, dans le fichier *fonctions.c* :

```
/*   Génère un entier aléatoire entre 0 et maxVal-1   */
int RandInt(int maxVal) {
     return rand() % maxVal;
}
```

Cette fonction va être utilisée dans la fonction *CreateRoach()* qui crée un cafard. Voici le code source de la fonction *CreateRoach()*, dans le fichier *fonctions.c* :

```
/*   Création d'un cafard   */
Roach CreateRoach( SDL_Surface *ecran, SDL_Surface *sprites )
{
    Roach roach;
    roach.sprites = sprites; // surface correspondant au sprite sheet
    roach.dir = RandInt(ROACH_ORIENTATIONS);
    roach.x = RandInt(ecran->w - ROACH_WIDTH);
    roach.y = RandInt(ecran->h - ROACH_HEIGHT);
```

```
    roach.hidden = 0;
    roach.steps = RandInt(MAX_STEPS);
    roach.angle = roach.dir * ROACH_ANGLE / 180.0 * M_PI; // radians
    roach.turnLeft = RandInt(2); // droitier: 0, gaucher: 1

    return roach;
}
```

Ces deux fonctions doivent bien entendu être déclarées dans *sdl2.h* :

```
int RandInt(int maxVal);
Roach CreateRoach( SDL_Surface *ecran, SDL_Surface *sprites );
```

Voici l'appel de la fonction *CreateRoach()* dans le *main()* :

```
Roach roach = CreateRoach ( ecran, sprites );
```

Cet appel doit être placé après l'appel de fonction *LoadSprites().*

- **Compilez et testez** (comme en §1.3) :
  Vos devez voir apparaître un cafard sur le sol (image de fond). Si vous relancez le programme plusieurs fois, le cafard apparaît à une position et une orientation différente.

# 3. Déplacement d'un cafard

## 3.1. Comportement de déplacement d'un cafard

Le déplacement d'un cafard (fonction *MoveRoach()*) va consister, pour chaque tour de la boucle `while` d'attente d'évènements, à calculer une nouvelle position (x,y) en tenant compte de l'orientation et de la vitesse (la vitesse est une constante paramétrable : ROACH_SPEED dans *sdl2.h*). Si la nouvelle position sort de l'écran, x et y ne sont pas mis à jour; par contre, une nouvelle orientation est donnée au cafard (fonction *TurnRoach()*). Le changement d'orientation est de 15, 30 ou 45°, vers la droite ou la gauche. Pour donner un aspect plus «vivant» aux déplacements, deux champs de la structure `Roach` sont utilisés :

- `steps` : ce champ est décrémenté à chaque tour de boucle. Quand il vaut 0, l'orientation est changée (fonction *TurnRoach()*), et `steps` est réinitialisé à une valeur aléatoire.
- `turnLeft` : si ce champ est à 1, le cafard est «gaucher» (il tourne toujours à gauche lors des changements d'orientation), s'il est à 0, le cafard est «droitier».

Pour tester si le cafard est dans l'écran, on va créer une fonction *RoachInRect()* qui va, plus généralement, tester si une position (x,y) du cafard implique que le cafard est complètement inclus dans un rectangle dont on donne le (x, y) du coin supérieur gauche, la largeur et la hauteur. Cette fonction va être utilisée dans la fonction *MoveRoach()*. Voici le code source de la fonction *RoachInRect()*, dans le fichier *fonctions.c* :

```
/* Teste si le cafard est complètement dans le rectangle spécifié */
int RoachInRect(int x, int y,
                int rectx, int recty, int rectwidth, int rectheight)
{
    if (x < rectx) return 0;
    if ((x + ROACH_WIDTH) > (rectx + rectwidth)) return 0;
    if (y < recty) return 0;
    if ((y + ROACH_HEIGHT) > (recty + rectheight)) return 0;
```

```
    return 1;
}
```

Voici le code source de la fonction *TurnRoach()*, dans le fichier *fonctions.c* :

```
/* Changement de direction */
void TurnRoach(Roach *roach)
{
    if (roach->turnLeft) {
        roach->dir += RandInt(3) + 1; // +1 à 3
        if (roach->dir >= ROACH_ORIENTATIONS)
            roach->dir -= ROACH_ORIENTATIONS;
    } else {
        roach->dir -= RandInt(3) + 1; // -1 à 3
        if (roach->dir < 0)
            roach->dir += ROACH_ORIENTATIONS;
    }
    roach->angle = roach->dir * ROACH_ANGLE / 180.0 * M_PI; //radians
}
```

Voici le code source de la fonction *MoveRoach()*, dans le fichier *fonctions.c* :

```
/*   Déplacement d'un cafard   */
void MoveRoach(Roach *roach, float roachSpeed, SDL_Surface *ecran)
{
    int newX, newY;
    newX = roach->x + (int)(roachSpeed * cos (roach->angle) );
    newY = roach->y - (int)(roachSpeed * sin (roach->angle) );

    /* Si dans la fenetre */
    if (RoachInRect(newX, newY, 0, 0, ecran->w, ecran->h)) {

        roach->x = newX;
        roach->y = newY;

        if (roach->steps-- <= 0) {
           TurnRoach(roach);
           roach->steps = RandInt(MAX_STEPS);
        }
    } else {
        TurnRoach(roach);
    }
}
```

Ces trois fonctions doivent bien entendu être déclarées dans *sdl2.h* :

```
int RoachInRect(int x,int y,
          int rectx, int recty, int rectwidth, int rectheight);
void TurnRoach(Roach *roach);
void MoveRoach(Roach *roach, float roachSpeed, SDL_Surface *ecran);
```

Voici l'appel de la fonction *MoveRoach()* dans le *main()* :

```
MoveRoach( &roach, ROACH_SPEED, ecran);
```

Cet appel doit être placé avant l'appel de fonction *DrawRoach(),* dans la boucle `while` d'attente d'évènements.

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

  Maintenant le cafard se déplace (par moments), mais il y a deux problèmes :

  - le cafard ne se déplace que quand il y a un événement (par exemple quand vous bougez la souris et que vous appuyez sur une touche). Ceci provient du fait que la fonction *SDL_WaitEvent()* est bloquante : les fonctions suivantes (*MoveRoach()*, *DrawRoach()*) ne seront pas appelées tant qu'aucun événement n'est détecté.
  - le fond de la fenêtre n'est pas rafraichi quand le cafard se déplace : les différentes images du cafard restent affichées.

  Nous allons dans le paragraphe suivant traiter ces deux problèmes.

## 3.2. Boucle d'attente d'évènements non bloquante et «nettoyage» de l'écran

- **Boucle d'attente d'évènements non bloquante** :

  La SDL permet de lire un événement et de le sortir de la queue à l'aide de la fonction *SDL_PollEvent()* qui est non bloquante, c'est-à-dire que s'il y a aucun événement dans la queue, la fonction se termine. C'est ce que nous voulons ici, car nous devons, dans la boucle `while`, afficher le cafard en déplacement indépendamment de tout événement. Nous allons donc remplacer, dans cette boucle, l'appel à *SDL_WaitEvent()* par un appel à *SDL_PollEvent()*.

  Cependant, le traitement des évènements est prioritaire par rapport au tracé (pour qu'il n'y ait pas de délai dans le traitement d'un événement important) et nous allons faire en sorte que la fonction *SDL_PollEvent()* traite tous les événements de la queue et non pas un seul. Pour cela, nous remplaçons l'ancien code :

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

  par le nouveau code :

```
while ( SDL_PollEvent ( &event ) ) { // tant qu'il y a un évènement
     switch ( event.type ) {
          case SDL_QUIT :
               continuer = 0;
               break;
     }
}
```

  Nous avons donc maintenant, au lieu de la boucle simple d'attente d'évènements que nous connaissions, deux boucles imbriquées : `while(continuer)`, qui peut être qualifiée de boucle principale et `while(SDL_PollEvent(&event))`, qui peut être qualifiée de boucle de traitement d'évènements (il y aura d'autres évènements que SDL_QUIT, cf. §6).

- **«Nettoyage» du fond d'écran** :

  Ici nous avons deux options possibles : soit, à chaque tour de boucle, refaire un «*blitting*» complet de la surface de fond (*floor*) sur la surface de la fenêtre mère (*ecran*), soit, si cette opération est trop lente (ça dépend des performances de votre machine/carte graphique), refaire un «*blitting*» uniquement du rectangle de la surface de fond qui était occupé par le cafard avant son déplacement (position x,y avant *MoveRoach()*).

  La première option est évidemment plus facile à coder (un simple appel à la fonction *SDL_BlitSurface()*, avec les 2ème et 4ème paramètres à NULL).

  Nous donnons ci-dessous le prototype de la fonction *CleanRoach()*, que vous devez écrire si vous adoptez la seconde option :

```
void CleanRoach(Roach roach, SDL_Surface *ecran, SDL_Surface *floor);
```

  L'appel soit à cette fonction, soit au «*blitting*» complet (première option), doit être placé avant l'appel de fonction *MoveRoach(),* dans la boucle `while` principale.

- **Re-compilez et testez** :
  Maintenant les problèmes vus précédemment doivent être réglés. Dépendant des performances de votre machine/carte graphique, le mouvement peut être trop rapide. Nous allons voir ce réglage de vitesse dans le § suivant.

### 3.3. Réglage de la vitesse

Pour rendre la vitesse de déplacement d'un cafard (à l'écran) indépendante de votre matériel, il faut utiliser les **fonctions de gestion du temps de la SDL**. Il y en a plusieurs, selon que l'on veut gérer le temps de façon très précise ou non.

Ici la précision n'est pas importante, nous allons donc utiliser la fonction la plus simple : la fonction *SDL_Delay()*. Cette fonction attend un nombre (spécifié en paramètre) de millisecondes avant de se terminer : il s'agit d'une sorte de «pause» de durée déterminée.

Pour que cette durée soit facilement paramétrable, nous allons rajouter la constante `DELAI_P_TOUR` dans *sdl2.h* :

```
#define DELAI_P_TOUR       20  // en millisecondes
```

L'appel de la fonction *SDL_Delay()* dans le *main()* doit être placé à la fin de la boucle `while` principale, après toutes les fonctions de tracé :

```
SDL_Delay(DELAI_P_TOUR);
```

## 4. Une colonie de cafards

### 4.1. Création et tracé de la colonie

Nous allons maintenant étendre l'application à plusieurs cafards. Pour que ce nombre soit facilement paramétrable, nous allons rajouter la constante `MAX_ROACHES` dans *sdl2.h* :

```
#define MAX_ROACHES        20    /* nb maxi de cafards */
```

Pour certaines fonctions que nous avons avons décrites aux §2 et 3 pour un cafard, nous allons créer une fonction équivalente pour tous les cafards :

- *CreateRoach() => CreateRoaches()*,
- *DrawRoach() => DrawRoaches()*,
- *MoveRoach() => MoveRoaches()*,
- *CleanRoach() => CleanRoaches()*.

Chaque fonction *xxxxRoaches()* va appeler la fonction *xxxxRoach()* équivalente, dans une boucle `for`. Pour les trois dernières, à la place du paramètre *Roach* (un cafard), elles vont avoir deux paramètres : un tableau de *Roach* et la taille de ce tableau (nombre de cafards). La fonction *CreateRoach()* va retourner non pas un *Roach*, mais un tableau de *Roach* (qu'elle devra allouer par un *malloc()*).

Voici la déclaration de ces quatre fonctions dans *sdl2.h* :

```
Roach *CreateRoaches ( SDL_Surface *ecran, SDL_Surface *sprites,
                       int nbRoach );
void DrawRoaches ( Roach *roaches, int nbRoach, SDL_Surface *ecran );
void MoveRoaches ( Roach *roaches, int nbRoach, float roachSpeed,
                       SDL_Surface *ecran);
void CleanRoaches ( Roach *roaches,int nbRoach, SDL_Surface *ecran,
                       SDL_Surface *floor);
```

En tant qu'exemple, voici le code source de la fonction *DrawRoaches()*, dans le fichier *fonctions.c* :

```
void DrawRoaches(Roach *roaches, int nbRoach, SDL_Surface *ecran)
{
    int i;

    for ( i = 0; i < nbRoach; i++ ) {
        DrawRoach( roaches[i], ecran);
    }
}
```

Voici l'appel de la fonction *CreateRoaches()* dans le *main()*, qui remplace l'appel de *CreateRoach()* :

```
Roach *roaches = CreateRoaches ( ecran, sprites, MAX_ROACHES );
```

Voici ce que devient la fin de la boucle `while` principale dans le *main()* :

```
    //SDL_BlitSurface(floor, NULL, ecran, NULL);// option 1
    CleanRoaches ( roaches, MAX_ROACHES, ecran, floor );// option 2

    MoveRoaches( roaches, MAX_ROACHES, ROACH_SPEED, ecran);
    DrawRoaches ( roaches, MAX_ROACHES, ecran );

    SDL_Flip(ecran);
    SDL_Delay(DELAI_P_TOUR);
```

- **Re-compilez et testez** :

  En regardant bien, vous pouvez remarquer que les cafards peuvent se superposer. Ce n'est pas trop gênant car, à cause du mouvement, ces recouvrements ne choquent pas l'œil. Cependant nous allons régler ce problème dans le §4.2 ci-dessous. Vous pouvez sauter ce paragraphe si vous êtes en retard dans votre TP, car le paragraphe suivant (§5) est plus important.

## 4.2. Gestion des collisions

Pour pouvoir gérer les collisions, il faut déterminer si deux cafards se recouvrent partiellement, c'est-à-dire si les «*bounding boxes*» correspondant à chaque cafard sont en intersection. Pour

tester cela, on va créer une fonction *RoachIntersectRect()* qui va, plus généralement, tester si une position (x,y) du cafard implique que le cafard est en intersection avec un rectangle dont on donne le (x, y) du coin supérieur gauche, la largeur et la hauteur. Cette fonction va être utilisée dans la nouvelle version de la fonction *MoveRoach()*. Voici le code source de la fonction *RoachIntersectRect()*, dans le fichier *fonctions.c* :

```
/*   Teste si le cafard est en intersection avec le rectangle
spécifié   */
int RoachIntersectRect(int x, int y, int rectx, int recty, int
rectwidth, int rectheight)
{
    if (x >= (rectx + rectwidth)) return 0;
    if ((x + ROACH_WIDTH) <= rectx) return 0;
    if (y >= (recty + rectheight)) return 0;
    if ((y + ROACH_HEIGHT) <= recty) return 0;

    return 1;
}
```

Le fait qu'un cafard doive, pour se déplacer, tenir compte de la position de tous les autres implique de fournir à la fonction *MoveRoach()* le tableau contenant tous les cafards (et la taille de ce tableau). Il faut donc modifier sa déclaration (dans *sdl2.h*), sa définition et son appel (dans *fonctions.c*). Voici sa nouvelle déclaration dans *sdl2.h* :

```
/* Version avec gestion des collisions */
void MoveRoach(Roach *roaches, int nbRoaches, int index,
                     float roachSpeed, SDL_Surface *ecran);
```

Voici le nouveau code source du début de la fonction *MoveRoach()*, dans le fichier *fonctions.c* :

```
/*   Déplacement d'un cafard : version avec gestion des collisions */

void MoveRoach(Roach *roaches, int nbRoaches, int index,
                     float roachSpeed, SDL_Surface *ecran)
{
    Roach *roach = &roaches[index];
    int newX, newY;
    int i;

    newX = roach->x + (int)(roachSpeed * cos (roach->angle) );
    newY = roach->y - (int)(roachSpeed * sin (roach->angle) );

    // Si dans la fenetre
    if (RoachInRect(newX, newY, 0, 0, ecran->w, ecran->h)) {

       // Gestion des collisions
       for ( i = 0; i < index; i++ ) {

         if (RoachIntersectRect(newX,newY,roaches[i].x, roaches[i].y,
                                ROACH_WIDTH, ROACH_HEIGHT)) {
             TurnRoach(roach);
             break;
         }
       }
       // Le reste du code de la fonction est identique
       ...
```

L'appel de la fonction *MoveRoach()* dans la fonction *MoveRoaches()* (dans *fonctions.c*) change :

```
for ( i = 0; i < nbRoach; i++ ) {
   //MoveRoach(&roaches[i], roachSpeed, ecran);//Sans gest collisions
   MoveRoach( roaches, nbRoach, i, roachSpeed, ecran );
}
```

# 5. Les cafards se cachent sous le tapis

Les cafards sont des insectes qui fuient la lumière. Ils essaient donc de se cacher à l'abri de la lumière. Nous allons afficher à l'écran un tapis et adapter le comportement de nos cafards pour qu'ils puissent se cacher dessous. C'est la raison d'être du champ `hidden` de la structure `Roach` : il sera mis à 1 quand le cafard sera caché, et alors il arrêtera tout mouvement.

## 5.1. Chargement de l'image et affichage du «tapis»

Le plus simple est de créer deux fonctions : la fonction *LoadImage()* pour le chargement de l'image *bmp*, et la fonction *DrawImage()* pour le «*blitting*» à l'écran.

L'image du tapis devra s'afficher sur l'écran à une position initiale (x,y du coin supérieur gauche). Pour que cette position soit facilement paramétrable, nous allons rajouter les constantes `X_INIT_TAPIS` et `Y_INIT_TAPIS` dans *sdl2.h* :

```
#define X_INIT_TAPIS       100
#define Y_INIT_TAPIS       100
```

- La **fonction *LoadImage()*** ressemble beaucoup à la fonction *LoadSprites()* (cf §2.1), mais elle doit aussi effectuer une autre opération : pour mémoriser dans la variable renvoyée (de type *SDL_Surface*) la position initiale (x,y), il faut renseigner le champ `clip_rect` avec le rectangle «destination» (il n'y a en effet pas de champs x, y dans la structure *SDL_Surface*). Le prototype de la fonction *LoadImage()* est le suivant :

```
SDL_Surface *LoadImage ( char * img_filename, int x, int y );
```

Son appel se situe après l'appel de *LoadSprites()* dans le *main()* :

```
SDL_Surface *tapis = LoadImage ( "bathmat.bmp",
                                 X_INIT_TAPIS, Y_INIT_TAPIS );
```

- La **fonction *DrawImage()*** prend en premier argument la surface (`tapis`) renvoyée par *LoadImage()*. La fonction effectue un «*blitting*» à la position spécifiée par le rectangle «destination» renseigné dans le champ `clip_rect` de la surface `tapis`.

Le prototype de la fonction *DrawImage()* est le suivant :

```
void DrawImage (SDL_Surface *img, SDL_Surface *ecran);
```

Son appel se situe après l'appel de *DrawRoaches()* dans la boucle `while` principale du *main()* :

```
DrawImage ( tapis, ecran );
```

- **Re-compilez et testez** :
  Si tout a bien fonctionné, le tapis s'affiche et les cafards n'apparaissent pas quand ils passent sur le rectangle du tapis. Cependant le mouvement des cafards n'a pas changé, ils ne se cachent pas sous le tapis. Nous allons voir comment modifier ce comportement dans le § suivant.

### 5.2. Marquer les cafards cachés

Pour marquer les cafards cachés, on va créer la fonction *MarkHiddenRoaches()* qui va, pour chaque cafard, tester s'il est complètement inclus dans le rectangle du tapis (à l'aide de la fonction *RoachInRect()*). Si oui, le champ `hidden` de ce cafard est mis à 1, sinon il est mis à 0. Voici le code source de la fonction *MarkHiddenRoach()*, dans le fichier *fonctions.c* :

```
/*   Marque les cafards cachés  */
int MarkHiddenRoaches(Roach *roaches,int nbRoaches,SDL_Surface *rect)
{
    int i;
    int nVisible = 0;

    for ( i = 0; i < nbRoaches; i++ ) {
      if (RoachInRect( roaches[i].x, roaches[i].y, rect->clip_rect.x,
                rect->clip_rect.y, rect->w, rect->h)) {
         roaches[i].hidden = 1;
      }
      else {
         roaches[i].hidden = 0;
         nVisible++;
      }
    }
    return nVisible;
}
```

Son appel se situe avant l'appel des fonctions *xxxxRoaches()* dans la boucle `while` principale du *main()*. `nVis` est un entier qui contient le nombre de cafards visibles. S'il vaut 0, il est inutile d'appeler les autres fonctions (*CleanRoaches()*, *MoveRoaches()*, *DrawRoaches()*) :

```
nVis = MarkHiddenRoaches(roaches, MAX_ROACHES, tapis);
```

- **Re-compilez et testez** :
  Si tout fonctionne bien, au bout d'un certain temps tous les cafards sont cachés sous le tapis. Nous allons voir comment déplacer le tapis dans le § suivant.

## 6. Déplacer le tapis : «*drag and drop*»

Nous allons autoriser l'utilisateur à déplacer le tapis à l'aide d'un «*drag and drop*» (glisser-déposer). Un «*drag and drop*» est une opération qui consiste à déplacer un élément graphique d'un endroit à l'autre de l'écran : avec la souris, elle consiste classiquement à cliquer (bouton gauche) sur l'élément à déplacer, à bouger la souris en gardant le bouton enfoncé, puis à relâcher le bouton à l'endroit adéquat pour l'élément graphique.

Nous devrons gérer les évènements SDL_MOUSEBUTTONDOWN, SDL_MOUSEMOTION et SDL_MOUSEBUTTONUP dans la boucle `while` de traitement des évènements. Le principe est le suivant :

- si un appui sur le bouton gauche a lieu sur le tapis, on met une variable `btndown` à 1,
- si le bouton gauche est relâché, on met la variable `btndown` à 0,
- si un mouvement de la souris a lieu alors que `btndown` est à 1, ceci veut dire que l'utilisateur est en train de réaliser un «*drag and drop*» du tapis.

La boucle `while` de traitement des évènements que nous avons mis en place au §3.2 doit donc être complétée pour traiter ces trois types d'évènements.

Pour tester si la position du clic est dans le rectangle du tapis, on va créer une fonction *PointInRect()* qui va, plus généralement, tester si un point (x,y) est inclus dans un rectangle dont on donne le (x, y) du coin supérieur gauche, la largeur et la hauteur. Voici le code source de la fonction *PointInRect()*, dans le fichier *fonctions.c* :

```
/*   Teste si le point est dans le rectangle spécifié   */
int PointInRect(int x, int y, int rectx, int recty,
                                int rectwidth, int rectheight) {
    if (x < rectx) return 0;
    if (x > (rectx + rectwidth)) return 0;
    if (y < recty) return 0;
    if (y > (recty + rectheight)) return 0;

    return 1;
}
```

## 6.1. Modification de la boucle de traitement des évènements

La boucle va être modifiée comme suit (des entiers `deltax`, `deltay` et `btndown` seront déclarés auparavant). Pour la fonction *CleanSurf()*, cf. la fonction *CleanRoach()* §3.2.

```
while ( SDL_PollEvent ( &event ) ) { // tant qu'il y a un évènement
    switch ( event.type ) {
        case SDL_QUIT :
            continuer = 0;
            break;
        case SDL_MOUSEBUTTONDOWN:
            if (event.button.button == SDL_BUTTON_LEFT
                && PointInRect( event.button.x, event.button.y,
                           tapis->clip_rect.x,tapis->clip_rect.y,
                           tapis->w,tapis->h)){
                /* Clic sur le tapis */
                btndown = 1;
                // Ecart entre clic et coin sup gauche tapis
                deltax = event.button.x - tapis->clip_rect.x;
                deltay = event.button.y - tapis->clip_rect.y;
            }
            break;
        case SDL_MOUSEMOTION :
            if ( btndown ) { // Drag and drop
                // "Nettoyage" du tapis (2 options )
                //SDL_BlitSurface(floor,NULL, ecran,NULL);//option 1
                CleanSurf ( tapis, ecran, floor );// option 2
                // Mise à jour position du tapis
                tapis->clip_rect.x = event.button.x - deltax;
                tapis->clip_rect.y = event.button.y – deltay;
            }
            break;
        case SDL_MOUSEBUTTONUP :
            btndown = 0;
            break;
    }
}
```

- **Re-compilez et testez** :

Vous pouvez maintenant déplacer le tapis. Cependant, rien n'empêche de le déplacer pratiquement en dehors de la fenêtre. Nous allons essayer d'améliorer cela dans le § suivant.

## 6.2. Limites de déplacement du tapis

Contrôlez que le tapis reste entièrement à l'intérieur de la fenêtre : il faut, dans le code ci-dessus, limiter les valeurs données à `tapis->clip_rect.x` et à `tapis->clip_rect.y`.