Pour nous habituer aux événements, nous allons apprendre à traiter le plus simple d'entre eux : **la demande de fermeture du programme**.
C'est un événement qui se produit lorsque l'utilisateur clique sur la croix pour fermer la fenêtre (fig. suivante).

C'est vraiment l'événement le plus simple. En plus, c'est un événement que vous avez utilisé jusqu'ici sans vraiment le savoir, car il était situé dans la fonction `pause()` !
En effet, le rôle de la fonction `pause` était d'attendre que l'utilisateur demande à fermer le programme. Si on n'avait pas créé cette fonction, la fenêtre se serait affichée et fermée en un éclair !

À partir de maintenant, vous pouvez oublier la fonction `pause`. Supprimez-la de votre code source, car nous allons apprendre à la reproduire.

La variable d'événement

Pour traiter des événements, vous aurez besoin de déclarer une variable (juste une seule, rassurez-vous) de type `SDL_Event`. Appelez-la comme vous voulez : moi, je vais l'appeler `event`, ce qui signifie « événement » en anglais.

```
SDL_Event event;
```

Pour nos tests nous allons nous contenter d'un `main` très basique qui affiche juste une fenêtre, comme on l'a vu quelques chapitres plus tôt. Voici à quoi doit ressembler votre `main` :

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL;
    SDL_Event event; // Cette variable servira plus tard à gérer les événem
ents
    SDL_Init(SDL_INIT_VIDEO);
    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
    SDL_WM_SetCaption("Gestion des événements en SDL", NULL);
    SDL_Quit();
    return EXIT_SUCCESS;
}
```

C'est donc un code très basique, il ne contient qu'une nouveauté : la déclaration de la variable `event` dont nous allons bientôt nous servir.

Testez ce code : comme prévu, la fenêtre va s'afficher et se fermer immédiatement après.

La boucle des événements

Lorsqu'on veut attendre un événement, on fait généralement une boucle. Cette boucle se répètera tant qu'on n'a pas eu l'événement voulu.
On va avoir besoin d'utiliser un booléen qui indiquera si on doit continuer la boucle ou non. Créez donc ce booléen que vous appellerez par exemple `continuer` :

```
int continuer = 1;
```

Ce booléen est mis à 1 au départ car on veut que la boucle se répète TANT QUE la variable `continuer` vaut 1 (vrai). Dès qu'elle vaudra 0 (faux), alors on sortira de la boucle et le programme s'arrêtera.

Voici la boucle à créer :

```
while (continuer)
{
    /* Traitement des événements */
}
```

Voilà : on a pour le moment une boucle infinie qui ne s'arrêtera que si on met la variable `continuer` à 0. C'est ce que nous allons écrire à l'intérieur de cette boucle qui est le plus intéressant.

Récupération de l'événement

Maintenant, faisons appel à une fonction de la SDL pour demander si un événement s'est produit.
On dispose de deux fonctions qui font cela, mais d'une manière différente :

- `SDL_WaitEvent` : elle attend qu'un événement se produise. Cette fonction est dite bloquante car elle suspend l'exécution du programme tant qu'aucun événement ne s'est produit ;
- `SDL_PollEvent` : cette fonction fait la même chose mais n'est pas bloquante. Elle vous dit si un événement s'est produit ou non. Même si aucun événement ne s'est produit, elle rend la main à votre programme de suite.

Ces deux fonctions sont utiles, mais dans des cas différents.
Pour faire simple, si vous utilisez `SDL_WaitEvent` votre programme utilisera très peu de processeur car il attendra qu'un événement se produise.
En revanche, si vous utilisez `SDL_PollEvent`, votre programme va parcourir votre boucle `while` et rappeler `SDL_PollEvent` indéfiniment jusqu'à ce qu'un événement se soit produit. À tous les coups, vous utiliserez 100 % du processeur.

Mais alors, il faut tout le temps utiliser `SDL_WaitEvent` si cette fonction utilise moins le processeur, non ?

Non, car il y a des cas où `SDL_PollEvent` se révèle indispensable. C'est le cas des jeux dans lesquels l'écran se met à jour même quand il n'y a pas d'événement.
Prenons par exemple Tetris : les blocs descendent tout seuls, il n'y a pas besoin que l'utilisateur crée d'événement pour ça ! Si on avait utilisé `SDL_WaitEvent`, le programme serait resté « bloqué » dans cette fonction et vous n'auriez pas pu mettre à jour l'écran pour faire descendre les blocs !

Comment fait `SDL_WaitEvent` pour ne pas consommer de processeur ?
Après tout, la fonction est bien obligée de faire une boucle infinie pour tester tout le temps s'il y a un événement ou non, n'est-ce pas ?

C'est une question que je me posais il y a encore peu de temps. La réponse est un petit peu compliquée car ça concerne la façon dont l'OS gère les processus (les programmes).
Si vous voulez – mais je vous en parle rapidement –, avec `SDL_WaitEvent`, le processus de votre programme est mis « en pause ». Votre programme n'est donc plus traité par le processeur.
Il sera « réveillé » par l'OS au moment où il y aura un événement. Du coup, le processeur se remettra à travailler sur votre programme à ce moment-là. Cela explique pourquoi votre programme ne consomme pas de processeur pendant qu'il attend l'événement.

Je comprends que ce soit un peu abstrait pour vous pour le moment. Et à dire vrai, vous n'avez pas besoin de comprendre ça maintenant. Vous assimilerez mieux toutes les différences plus loin en pratiquant.
Pour le moment, nous allons utiliser `SDL_WaitEvent` car notre programme reste très simple. Ces deux fonctions s'utilisent de toute façon de la même manière.

Vous devez envoyer à la fonction l'adresse de votre variable `event` qui stocke l'événement. Comme cette variable n'est pas un pointeur (regardez la déclaration à nouveau), nous allons mettre le symbole `&` devant le nom de la variable afin de donner l'adresse :

```
SDL_WaitEvent(&event);
```

Après appel de cette fonction, la variable `event` contient obligatoirement un événement.

Cela n'aurait pas forcément été le cas si on avait utilisé `SDL_PollEvent` : cette fonction aurait pu renvoyer « Pas d'événement ».

Analyse de l'événement

Maintenant, nous disposons d'une variable `event` qui contient des informations sur l'événement qui s'est produit.
Il faut regarder la sous-variable `event.type` et faire un test sur sa valeur. Généralement on utilise un `switch` pour tester l'événement.

Mais comment sait-on quelle valeur correspond à l'événement « Quitter », par exemple ?

La SDL nous fournit des constantes, ce qui simplifie grandement l'écriture du programme. Il en existe beaucoup (autant qu'il y a d'événements possibles). Nous les verrons au fur et à mesure tout au long de ce chapitre.

```
while (continuer)
{
    SDL_WaitEvent(&event); /* Récupération de l'événement dans event */
    switch(event.type) /* Test du type d'événement */
    {
        case SDL_QUIT: /* Si c'est un événement de type "Quitter" */
            continuer = 0;
            break;
    }
}
```

Voici comment ça fonctionne.

1. Dès qu'il y a un événement, la fonction `SDL_WaitEvent` renvoie cet événement dans `event`.
2. On analyse le type d'événement grâce à un `switch`. Le type de l'événement se trouve dans `event.type`
3. On teste à l'aide de `case` dans le `switch` le type de l'événement. Pour le moment, on ne teste que l'événement `SDL_QUIT` (demande de fermeture du programme), car c'est le seul qui nous intéresse.<liste>
4. Si c'est un événement `SDL_QUIT`, c'est que l'utilisateur a demandé à quitter le programme. Dans ce cas on met le booléen `continuer` à 0. Au prochain tour de boucle, la condition sera fausse et donc la boucle s'arrêtera. Le programme s'arrêtera ensuite.
5. Si ce n'est pas un événement `SDL_QUIT`, c'est qu'il s'est passé autre chose : l'utilisateur a appuyé sur une touche, a cliqué ou tout simplement bougé la souris dans la fenêtre. Comme ces autres événements ne nous intéressent pas, on ne les traite pas. On ne fait donc rien : la boucle recommence et on attend à nouveau un événement (on repart à l'étape 1).

Ce que je viens de vous expliquer ici est extrêmement important. Si vous avez compris ce code, vous avez tout compris et le reste du chapitre sera très facile pour vous.

Le code complet

```
int main(int argc, char *argv[])
{
    SDL_Surface *ecran = NULL;
    SDL_Event event; /* La variable contenant l'événement */
    int continuer = 1; /* Notre booléen pour la boucle */
    SDL_Init(SDL_INIT_VIDEO);
    ecran = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE);
    SDL_WM_SetCaption("Gestion des événements en SDL", NULL);

    while (continuer) /* TANT QUE la variable ne vaut pas 0 */
    {
        SDL_WaitEvent(&event); /* On attend un événement qu'on récupère dan
s event */
        switch(event.type) /* On teste le type d'événement */
        {
            case SDL_QUIT: /* Si c'est un événement QUITTER */
                continuer = 0; /* On met le booléen à 0, donc la boucle va
s'arrêter */
                break;
        }
    }
    SDL_Quit();
    return EXIT_SUCCESS;
}
```

Voilà le code complet. Il n'y a rien de bien difficile : si vous avez suivi jusqu'ici, ça ne devrait pas vous surprendre.
D'ailleurs, vous remarquerez qu'on n'a fait que reproduire ce que faisait la fonction `pause`. Comparez avec le code de la fonction `pause` : c'est le même, sauf qu'on a cette fois tout mis dans le `main`. Bien entendu, il est préférable de placer ce code dans une fonction à part, comme `pause`, car cela allège la fonction `main` et la rend plus lisible

SOURIS

Vous vous dites peut-être que gérer la souris est plus compliqué que le clavier ?
Que nenni ! C'est même plus simple, vous allez voir !

La souris peut générer trois types d'événements différents.

- `SDL_MOUSEBUTTONDOWN` : lorsqu'on clique avec la souris. Cela correspond au moment où le bouton de la souris est enfoncé.
- `SDL_MOUSEBUTTONUP` : lorsqu'on relâche le bouton de la souris. Tout cela fonctionne exactement sur le même principe que les touches du clavier : il y a d'abord un appui, puis un relâchement du bouton.
- `SDL_MOUSEMOTION` : lorsqu'on déplace la souris. À chaque fois que la souris bouge dans la fenêtre (ne serait-ce que d'un pixel !), un événement `SDL_MOUSEMOTION` est généré !

Nous allons d'abord travailler avec les clics de la souris et plus particulièrement avec `SDL_MOUSEBUTTONUP`. On ne travaillera pas avec `SDL_MOUSEBUTTONDOWN` ici, mais vous savez de toute manière que c'est exactement pareil sauf que cela se produit plus tôt, au moment de l'enfoncement du bouton de la souris.
Nous verrons un peu plus loin comment traiter l'événement `SDL_MOUSEMOTION`.

Gérer les clics de la souris

Nous allons donc capturer un événement de type `SDL_MOUSEBUTTONUP` (clic de la souris) puis voir quelles informations on peut récupérer.
Comme d'habitude, on va devoir ajouter un `case` dans notre `switch` de test, alors allons-y gaiement :

```
switch(event.type)
{
    case SDL_QUIT:
        continuer = 0;
        break;
    case SDL_MOUSEBUTTONUP: /* Clic de la souris */
        break;
}
```

Jusque-là, pas de difficulté majeure.

Quelles informations peut-on récupérer lors d'un clic de la souris ? Il y en a deux :

- **le bouton de la souris** avec lequel on a cliqué (clic gauche ? clic droit ? clic bouton du milieu ?) ;
- **les coordonnées de la souris** au moment du clic (x et y).

Récupérer le bouton de la souris

On va d'abord voir avec quel bouton de la souris on a cliqué.
Pour cela, il faut analyser la sous-variable `event.button.button` (non, je ne bégaie pas) et comparer sa valeur avec l'une des 5 constantes suivantes :

- `SDL_BUTTON_LEFT` : clic avec le bouton gauche de la souris ;
- `SDL_BUTTON_MIDDLE` : clic avec le bouton du milieu de la souris (tout le monde n'en a pas forcément un, c'est en général un clic avec la molette) ;
- `SDL_BUTTON_RIGHT` : clic avec le bouton droit de la souris ;
- `SDL_BUTTON_WHEELUP` : molette de la souris vers le haut ;
- `SDL_BUTTON_WHEELDOWN` : molette de la souris vers le bas.

Les deux dernières constantes correspondent au mouvement vers le haut ou vers le bas auquel on procède avec la molette de la souris. Elles ne correspondent pas à un « clic » sur la molette comme on pourrait le penser à tort.

On va faire un test simple pour vérifier si on a fait un clic droit avec la souris. Si on a fait un clic droit, on arrête le programme (oui, je sais, ce n'est pas très original pour le moment mais ça permet de tester) :

```
switch(event.type)
{
    case SDL_QUIT:
        continuer = 0;
        break;
    case SDL_MOUSEBUTTONUP:
        if (event.button.button == SDL_BUTTON_RIGHT) /* On arrête le progra
mme si on a fait un clic droit */
            continuer = 0;
        break;
}
```

Vous pouvez tester, vous verrez que le programme s'arrête si on fait un clic droit.

Récupérer les coordonnées de la souris

Voilà une information très intéressante : les coordonnées de la souris au moment du clic !
On les récupère à l'aide de deux variables (pour l'abscisse et l'ordonnée) : `event.button.x` et `event.button.y`.

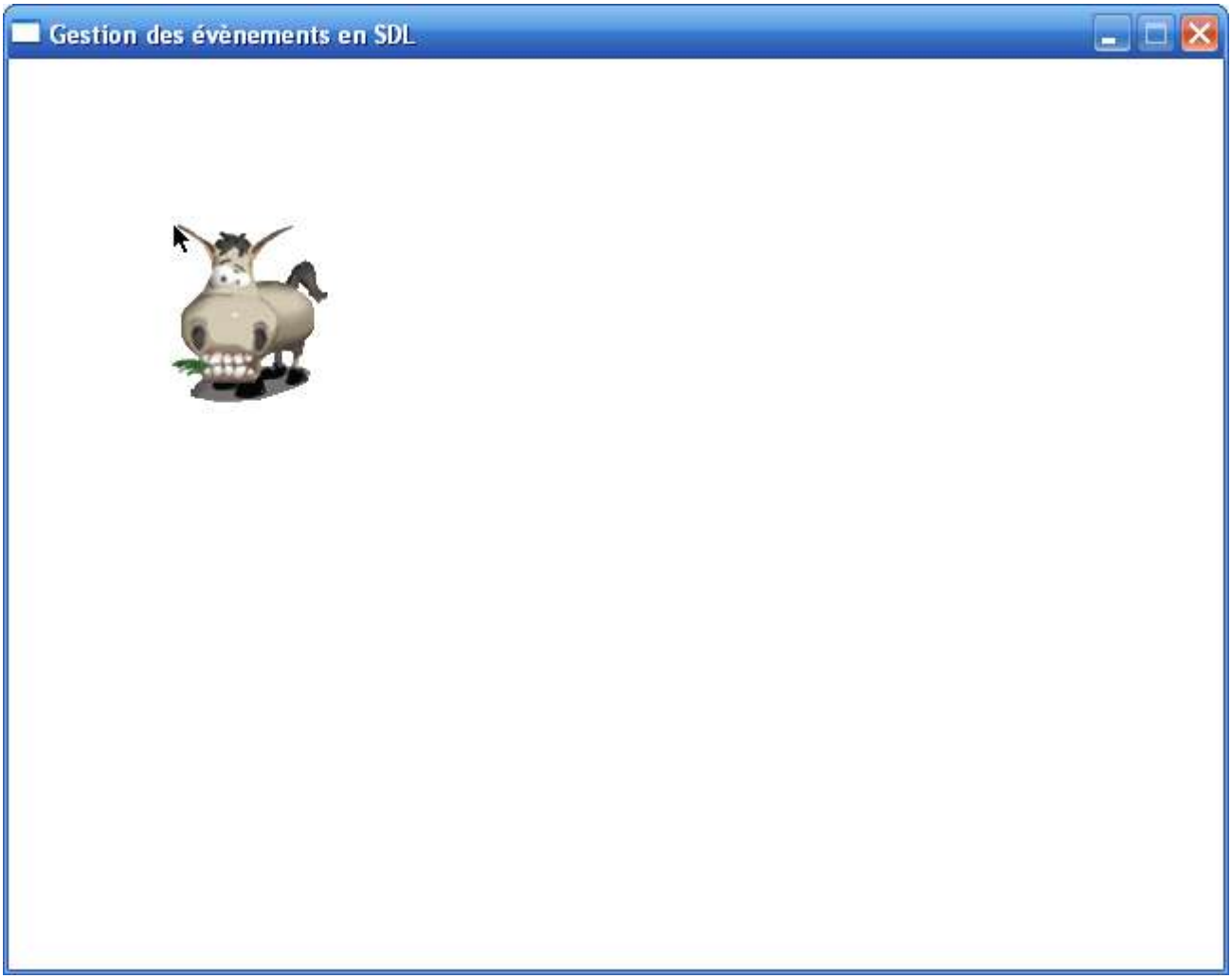
Amusons-nous un petit peu : on va blitter Zozor à l'endroit du clic de la souris.
Compliqué ? Pas du tout ! Essayez de le faire, c'est un jeu d'enfant !

Voici la correction :

```
while (continuer)
{
    SDL_WaitEvent(&event);
    switch(event.type)
    {
        case SDL_QUIT:
            continuer = 0;
```

```
            break;
        case SDL_MOUSEBUTTONUP:
            positionZozor.x = event.button.x;
            positionZozor.y = event.button.y;
            break;
    }
    SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));
    SDL_BlitSurface(zozor, NULL, ecran, &positionZozor); /* On place Zozor
à sa nouvelle position */
    SDL_Flip(ecran);
}
```

Ça ressemble à s'y méprendre à ce que je faisais avec les touches du clavier. Là, c'est même encore plus simple : on met directement la valeur de `x` de la souris dans `positionZozor.x`, et de même pour `y`.
Ensuite on blitte Zozor à ces coordonnées-là, et voilà le travail (fig. suivante) !



Petit exercice très simple : pour le moment, on déplace Zozor quel que soit le bouton de la souris utilisé pour le clic. Essayez de ne déplacer Zozor que si on fait un clic gauche avec la souris. Si on fait un clic droit, arrêtez le programme.

Gérer le déplacement de la souris

Un déplacement de la souris génère un événement de type `SDL_MOUSEMOTION`.
Notez bien qu'on génère autant d'événements que l'on parcourt de pixels pour se déplacer ! Si on bouge la souris de 100 pixels (ce qui n'est pas beaucoup), il y aura donc 100 événements générés.

Mais ça ne fait pas beaucoup d'événements à gérer pour notre ordinateur, tout ça ?

Pas du tout : rassurez-vous, il en a vu d'autres !

Bon, que peut-on récupérer d'intéressant ici ?
Les coordonnées de la souris, bien sûr ! On les trouve dans `event.motion.x` et `event.motion.y`.

Faites attention : on n'utilise pas les mêmes variables que pour le clic de souris de tout à l'heure (avant, c'était `event.button.x`). Les variables utilisées sont différentes en SDL en fonction de l'événement.

On va placer Zozor aux mêmes coordonnées que la souris, là encore. Vous allez voir, c'est rudement efficace et toujours aussi simple !

```
while (continuer)
{
    SDL_WaitEvent(&event);
    switch(event.type)
    {
        case SDL_QUIT:
            continuer = 0;
            break;
        case SDL_MOUSEMOTION:
            positionZozor.x = event.motion.x;
            positionZozor.y = event.motion.y;
            break;
    }
    SDL_FillRect(ecran, NULL, SDL_MapRGB(ecran->format, 255, 255, 255));
    SDL_BlitSurface(zozor, NULL, ecran, &positionZozor); /* On place Zozor
à sa nouvelle position */
    SDL_Flip(ecran);
}
```

Bougez votre Zozor à l'écran. Que voyez-vous ?
Il suit naturellement la souris où que vous alliez. C'est beau, c'est rapide, c'est fluide (vive le double buffering).

Quelques autres fonctions avec la souris

Nous allons voir deux fonctions très simples en rapport avec la souris, puisque nous y sommes. Ces fonctions vous seront très probablement utiles bientôt.

Masquer la souris

On peut masquer le curseur de la souris très facilement.
Il suffit d'appeler la fonction `SDL_ShowCursor` et de lui envoyer un flag :

- `SDL_DISABLE` : masque le curseur de la souris ;
- `SDL_ENABLE` : réaffiche le curseur de la souris.

Par exemple :

```
SDL_ShowCursor(SDL_DISABLE);
```

Le curseur de la souris restera masqué tant qu'il sera à l'intérieur de la fenêtre.
Masquez de préférence le curseur avant la boucle principale du programme. Pas la peine en effet de le masquer à chaque tour de boucle, une seule fois suffit.

Placer la souris à un endroit précis

On peut placer manuellement le curseur de la souris aux coordonnées que l'on veut dans la fenêtre.
On utilise pour cela `SDL_WarpMouse` qui prend pour paramètres les coordonnées x et y où le curseur doit être placé.

Par exemple, le code suivant place la souris au centre de l'écran :

```
SDL_WarpMouse(ecran->w / 2, ecran->h / 2);
```

Lorsque vous faites un `WarpMouse`, un événement de type `SDL_MOUSEMOTION` sera généré. Eh oui, la souris a bougé ! Même si ce n'est pas l'utilisateur qui l'a fait, il y a quand même eu un déplacement.