Apprenez les bases du C en réalisant un économiseur d'écran avec Xscreensaver

Cet article a pour but de proposer aux personnes qui ne sont pas totalement familiarisées avec le C, de s'exercer avec Xscreensaver, tout en apprenant quelques bases de la bibliothèque Xlib.

Xscreensaver est une des distributions d'économiseurs d'écran les plus populaire pour Linux. Ces économiseurs d'écran peuvent être en 2D, en utilisant la bibliothèque Xlib, ou en 3D en utilisant OpenGL. (Seule la 2D sera vue dans cet article.)

Si vous souhaitez réaliser un économiseur d'écran pour Xscreensaver avec OpenGL, je vous recommanderais d'utiliser OpenGL ES pour qu'il puisse être compatible avec les RaspberryPi.

A la date de rédaction de cet article, la dernière version Xscreensaver est la version 6.09, et les économiseurs d'écran sont rassemblés dans le répertoire `hacks/` (et ceux en 3D sont dans `hacks/glx/`).

Pour réaliser votre propre économiseur (aussi appelé démo), vous pouvez vous contenter de ne réaliser que l'étape du `configure` dans les sources, et `make` dans le répertoire `utils/`. Ensuite les démos du répertoire `hacks/` peuvent chacune être compilées individuellement, comme par exemple `munch.c` avec `make munch`, puis exécutées localement avec `./munch`. La démo se lancera alors dans une fenêtre. Vous pourrez procéder de la même manière avec votre propre réalisation pour tester le résultat. Pour compiler votre démo, vous pouvez soit éditer le `Makefile` pour y ajouter votre démo à côté des autres, soit reproduire la commande qui s'exécute quand vous compilez un économiseur existant.

Le fichier d'entête à inclure au début de votre économiseur d'écran est :

```
#include "screenhack.h"
```

L'inclusion d'un fichier d'entête est la méthode standard en C pour inclure des prototypes de fonction à utiliser, ainsi que les types de données associées.

Pour votre économiseur, pas besoin d'inclure une fonction `main()`, qui est généralement le point de départ dans les programmes C, mais ici vous devrez inclure cinq fonctions de rappel (callback) : "init", "draw", "reshape", "event" et "free". Chacune de ces fonctions devra être préfixée avec le nom de votre démo, et ce préfix devra être déclaré avec la macro `XSCREENSAVER_MODULE()`.

```
/* Entry Points */

static void *
my_xss_init(Display *display, Window window) { ... }

static unsigned long
my_xss_draw(Display *display, Window window, void *closure) { ... }

static void
my_xss_reshape(Display *display, Window window, void *closure,
               unsigned int w, unsigned int h) { ... }

static Bool
my_xss_event(Display *display, Window window, void *closure,
             XEvent *event) { ... }

static void
my_xss_free(Display *display, Window window, void *closure) { ... }
```

```
XSCREENSAVER_MODULE("my_xss", my_xss)
```

Les Fonctions en C

Les fonctions en C, comme dans les autres langages de programmation, regroupent des séries d'instructions dans des blocs de code. Les blocs de code en C sont regroupés à l'intérieur d'accolades `{ ... }`.

Voici une fonction simple qui réalise une addition entre deux nombres entiers passés en paramètre, et qui retourne le résultat :

```
int add(int a, int b) {
  int c;
  c = a + b;
  return c;
}
```

La première ligne correspond au prototype de la fonction, qui inclut le type de retour, le nom de la fonction, suivit des paramètres de la fonction, eux aussi précédés par leur type. Le corps de la fonction placé entre des accolades contient toutes les instructions qui seront exécutées quand la fonction sera appelée.

Les Variables en C

En C, les variables doivent d'abord être déclarées avec leur type avant d'être utilisées. C'est ce qu'on observe dans la fonction précédente avec la ligne de code `int c;`.

Les Structures en C

Les Structures en C regroupent plusieurs éléments similaires à des variables. Chacuns de ces éléments sont appelés des champs, et dans la déclaration du type de la structure chaque nom de champ doit être précédé par son type.

Pour réaliser un économiseur d'écran avec Xscreensaver vous devez regrouper toutes les variables d'état dans une structure principale. Ces structures sont nommée `state` par convention dans les économiseurs d'Xscreensaver.

```
/* State struct */

struct state {
  Display *display;
  Window window;
  GC gc;
  int width;
  int height;
  // ajoutez ici vos propres éléments
};
```

Ici, Les trois type `Display`, `Window` et `GC` correspondent à des types de la bibliothèque Xlib. Cette bibliothèque permet de communiquer avec le serveur X.org, qui est responsable de l'affichage graphique sous Linux.
La variable `display` est la variable principale qui permet la communication entre l'application cliente (le logiciel avec une fenêtre) et le serveur X (qui réalise l'affichage, juste au-dessus de l'OS). La variable `window` contient l'identifiant de la fenêtre où dessiner les éléments. La variable `gc` contient un context graphique (graphics context), qui est nécessaire pour réaliser des opérations de dessin.

La déclaration du type de la structure `state` ne provoquera pas la réservation d'un espace en mémoire pour contenir cet élément. On va donc allouer cet espace en mémoire dans la fonction "init" à l'aide de la fonction C

`malloc()`, comme ceci :

```
  struct state *st = (struct state *) malloc(sizeof(struct state));
```

Après cette allocation, tous les champs pourront être initialisés.

Même si toutes les fonctions de rappel transmettent les variables `display` et `window`, les inclure dans la structure `state` peut être une bonne idée pour pouvoir les passer plus facilement à des sous fonctions éventuelles.

```
  st->display = display;
  st->window = window;
```

La structure `state` ayant été ici allouée dynamiquement, on accède à ses champs avec une petite flèche `->`. Lorsqu'une structure ne provient pas d'une allocation dynamique (variable automatique), les champs sont accédés avec un point : `st.field`.

La première opération à inclure dans les autres fonctions de rappel, est la coercition (cast) du type de la structure `state`, car le type par défaut proposé par l'API d'Xscreensaver est un type générique. Ce type générique est ici un pointeur générique `void *closure` qui peut pointer vers l'espace mémore de n'importe quelle structure définie par les contributeurs à Xscreensaver.

```
  /* state cast */
  struct state *st = (struct state *) closure;
```

La coercition est réalisée en indiquant le type cible entre parenthèses, devant la variable.

Voici ci-dessous un template d'économiseur qui n'affichera qu'une fenêtre vide, mais que vous pouvez utiliser comme point de départ, et pour tester que la compilation et l'exécution fonctionnent correctement (compilez ce code comme expliqué au début de l'article).

```
#include "screenhack.h"

/* State struct */

struct state {
  Display *display;
  Window window;
  GC gc;
};

/* Entry Points */

static void *
my_xss_init(Display *display, Window window) {
  struct state *st = (struct state *) malloc(sizeof(struct state));
  return st;
}

static unsigned long
my_xss_draw(Display *display, Window window, void *closure) {
  return 100000;
}

static void
my_xss_reshape(Display *display, Window window, void *closure,
               unsigned int w, unsigned int h) { }

static Bool
my_xss_event(Display *display, Window window, void *closure,
             XEvent *event) { return False; }
```

```
static void
my_xss_free(Display *display, Window window, void *closure) {
}

/* Defaults and Options */
static const char *my_xss_defaults[] = {
  0
};

static XrmOptionDescRec my_xss_options[] = {
  { 0, 0, 0, 0 }
};

XSCREENSAVER_MODULE("my_xss", my_xss)
```

Les tableaux pour les paramètres par défaut et les options peuvent être laissés vide dans un premier temps.

Si la fenêtre est trop grande sur l'écran d'un ordinateur portable, vous pouvez spécifier la taille de la fenêtre comme ceci :

```
./myxss -geometry 400x300
```

Le callback "reshape" est appelé lorsque la fenêtre de la démo est redimensionnée. Il est également appelé une première fois au début de l'exécution du programme, juste *après* le callback d'initialisation "init". Aussi si vous avez besoin de ces dimensions pour initialiser votre démo dans le callback d'initialisation, la largeur et la hauteur de la fenêtre peuvent être récupérées avec la fonction `XGetWindowAttributes()`.

```
  {
    XWindowAttributes xgwa;
    XGetWindowAttributes(display, window, &xgwa);
    st->w = xgwa.width;
    st->h = xgwa.height;
  }
```

Le caractère '&' permet de transmettre l'adresse en mémoire de la structure, lorsque cette structure n'a pas été allouée dynamiquement.

Vous pourriez utiliser votre première primitive de dessin `XFillRectangle()` comme ceci, à partir du template vide précédent :

```
static void *
my_xss_init(Display *display, Window window) {
  struct state *st = malloc(sizeof(*st));
  unsigned long fg;
  st->display = display;
  st->window = window;
  st->gc = XDefaultGC(display, 0);
  fg = XWhitePixel(display, 0);
  XSetForeground(display, st->gc, fg);
  return st;
}

static unsigned long
my_xss_draw(Display *display, Window window, void *closure) {
  struct state *st = (struct state *) closure;
  XFillRectangle(st->display, st->window, st->gc,
    20, 20, 100, 60);
  return 200000;
}
```

Le fond étant noir par défaut, ici nous sélectionnons le blanc comme couleur de dessin.

Le nombre retourné par la fonction "draw" correspond au nombre de microsecondes à attendre avant le prochain appel à la fonction draw. Ainsi avec 200000, nous aurons 5 frames par seconde.

Pour dessiner avec une autre couleur, il faut allouer une couleur dans le serveur X avec la fonction `XAllocColor()`, puis sélectionner cette couleur avec `XSetForeground()`.

```
int screen_number = 0;
Colormap colormap = XDefaultColormap(display, screen_number);
XColor color;
color.red   = 65535;
color.green = 0;
color.blue  = 0;
XAllocColor(display, colormap, &color);
XSetForeground(display, st->gc, color.pixel);
```

`65535` est la valeur maximale que l'on peut obtenir avec une variable représentant un entier positif sur 16 bits.

Primitives de dessin

```
XDrawPoint(st->display, st->window, st->gc, x, y);
XDrawLine(st->display, st->window, st->gc, x1, y1, x2, y2);
XDrawRectangle(st->display, st->window, st->gc, x, y, width, height);
XDrawArc(st->display, st->window, st->gc, x, y, width, height, angle1, angle2);
```

Pour dessiner un cercle, vous pouvez spécifier `0` et `360*64` pour `angle1` et `angle2`.

```
XFillArc(st->display, st->window, st->gc, x, y, width, height, 0, 360*64);
```

Ces primitives de dessin sont basiques, cependant plein de choses très intéressantes peuvent être réalisées avec celles-ci. Cette liste est non-exhaustive, vous en trouverez bien plus au chapitre 8 de cette documentation sur la Xlib :

- `https://tronche.com/gui/x/xlib/`

Les Options

Pour un exemple d'utilisation des options que l'on peut passer en ligne de commande lors du lancement d'un économiseur, et de leurs valeurs par défaut, je vous recommanderais de regarder le code source de la démo `popsquares.c`.

Les événements

Pour un exemple de récupération des touches du clavier dans le callback "event", vous pouvez jeter un oeil à la démo `xmatrix.c`. Et vous trouverez dans la démo `pong.c`, qui est une démo jouable, comment utiliser l'évènement `MotionNotify` pour récupérer les mouvements de la souris. ("julia" et "attraction" sont également des démos interactives, mais avec un clic de la souris.)

Les évènements de type `XEvent` peuvent être accédés avec `event->xkey` ou `event->xbutton` ou autre, en fonction du type de l'évènement, car le type `XEvent` est un type `union` en C (défini dans `/usr/include/X11/Xlib.h`). Un type `union` rassemble plusieurs types en un seul, et qui peuvent être utilisés de manière alternative. Le premier champ de tous les types d'évènements est toujours le champ `int type`, qui permet de savoir lequel des types de l'union utiliser.

Comme vous pouvez le voir dans les démos d'Xscreensaver, on sélectionne et on utilise `event->xkey` après avoir vérifié que `event->type` a pour valeur `KeyPress`, et `event->xbutton` lorsque `event->type` a pour

valeur `ButtonPress`.

Les tableaux

Si vous souhaitez utiliser des tableaux C dans votre démo, vous pouvez réserver initialement un espace mémoire de taille fixe comme ceci :

```
// tableau de 10 entiers
{
  int elems[10];
}
```

Ou bien allouer dynamiquement de la mémoire à l'exécution avec `malloc()` ou `calloc()` :

```
{
  int *elems;
  elems = calloc(10, sizeof(int));
  elems[0] = 5;
}
```

(Tapez `man malloc` pour plus de détails.)

Les pointeurs

Ce que retournent les fonctions de type `malloc()` est une adresse mémoire. Ces adresses sont stoquées dans des variables de type pointeur, que l'on reconnait au caractère étoile (astérisque), dans le type de la déclaration de l'expression `int *elems;`.

Idées en vrac

Voici quelques idées d'économiseurs simples à réaliser si vous souhaitez vous exercer avec le C, et si vous manquez d'inspiration.

- Des éléments ressemblant aux avatars de github, constitués de carrés sur une grille symétrique de 5 par 5, et qui évoluent dans la fenêtre.
- Des cercles représentant des planètes avec des satelites qui gravitent autour.
- Des cercles qui suivent des chemins constitués de courbes de bezier.
- Utiliser le callback "event" pour ajouter des tours qui tirent sur les cercles de l'idée précédente, ce qui ferait un mini-jeu de type "tower-defense".
- Une démo avec des éléments géométriques simples qui s'inspire du mouvement pop-art, avec les peintres tels que Victor Vasarely et Bridget Riley (et notament leurs œuvres Nataraja et Orion, qui sont simples et intéressantes visuellement à reproduire).
- Des petites entités représentées par de petits carrés qui évoluent toutes seules dans la fenêtre, et collectent éventuellement des éléments qui apparaissent aléatoirement.

Pour cette dernière idée, une simple structure avec des coordonnées x et y, et avec une direction peut suffire.

L'étape de la conception

Lors de l'étape de conception, je vous recommanderais d'utiliser une feuille et un crayon. De faire des croquis pour rechercher des idées, puis définir plus précisément ce que vous voudriez obtenir.

Vous pourriez également réaliser un schéma sur la manière dont vous pensez organiser et structurer votre code.

Ensuite pour passer à la réalisation, je vous recommanderais de procéder de manière itérative, en commençant par une version extrèmement simple, à laquelle vous ajouterez les détails successivement, par étapes.