Programmation : Animation d'une image en langage C et SDL2 sur MorphOS

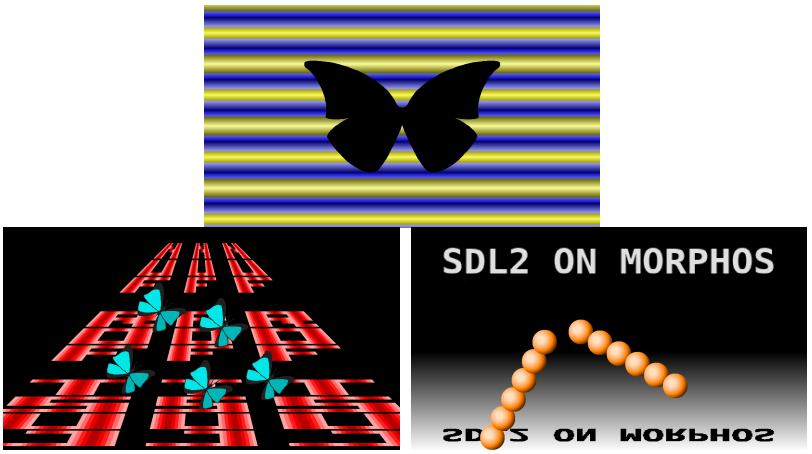
(Tutoriel développé et écrit par Sébastien Jeudy - mars 2024)

Présentation

SDL2 (Simple DirectMedia Layer version 2) est une bibliothèque logicielle libre utilisée pour créer des applications multimédias en deux dimensions (jeux, démos,...). Son succès est dû à sa portabilité sur la plupart des plates-formes et à sa facilité d'implémentation dans de nombreux langages, dont le langage C (ce qui nous intéresse ici).

SDL2 est disponible sur MorphOS depuis plusieurs années, notamment grâce à Bruno "BeWorld" Peloille qui assure aujourd'hui le portage de ses nouvelles versions en y incluant un maximum de ses fonctionnalités, mais aussi en interaction avec l'équipe de développement de MorphOS pour que le système puisse apporter tout son potentiel à cette bibliothèque. Du coup, Bruno est également l'auteur du portage d'un grand nombre de jeux et de logiciels à base de SDL2, dont plusieurs en montrent toute la puissance : www.morphos-storage.net/?all=1&dev=BeWorld.

Tout cela a fini par susciter mon intérêt et en découvrant la facilité de son implémentation en langage C, je me suis mis à développer une démo "oldschool" (JUST FOR FUN, sortie en 2022) : www.morphos-storage.net/?find=just-for-fun.



Démo JUST FOR FUN

Puis un petit jeu, réadaptation du célèbre Pong (MorPhONG, sorti en 2023) : www.morphos-storage.net/?find=MorPhONG.

Vidéos

Téléchargement

Documents
Jeux
Logiciels
Magazines
Divers

Liens

Associations
Jeux
Logiciels
Matériel
Magazines et médias
Pages personnelles
Réparateurs
Revendeurs
Scène démo
Sites de téléchargement
Divers

Partenaires

A Propos

A propos d'Obligement

Contact

David Brunet

obligement@free.fr

Jeu MorPhONG

Aujourd'hui, ce tutoriel est là pour vous introduire un cas simple d'utilisation de SDL2 : l'animation d'une image dans une fenêtre (un joli morpho bleu qui papillonne). Ceci sur MorphOS, mais le programme peut tout aussi bien être recompilé sur d'autres plates-formes comme AmigaOS 4 (histoire de voir un joli morpho bleu papillonner sur AmigaOS 4 ^^). Et tant qu'à faire, en langage C afin de profiter des meilleures performances du système.

La présentation est volontairement la plus détaillée possible. Avec des rudiments en programmation (variables, fonctions, if, while) et sans maîtriser le langage C, vous devriez en comprendre le principe tant son approche est simple.

En espérant susciter d'autres intérêts et au final, pourquoi pas, motiver le développement de nouveaux jeux et démos sur nos "Amiga NG". ;)

Prérequis

Disposer de la dernière version de MorphOS : www.morphos-team.net/downloads.

Afin de disposer du langage C et de son compilateur GCC, télécharger et installer la dernière version du SDK de MorphOS (Software Development Kit ou trousse de développement logiciel) : www.morphos-team.net/downloads.

Puis télécharger et installer la dernière version de SDL2 sur MorphOS : www.morphos-storage.net/?find=SDL_2.

Lors de l'installation, pour vos développements avec SDL2, attention à bien cocher l'option "Yes i have SDK installed" pour la question "Do you want SDK Files? (you need last SDK installed)".

Fichiers du programme étudié

- Code source : www.boingball.net/AMIGA_FOR_EVER/Codes/C/Butterfly/Butterfly.c.
- Programme exécutable (sur MorphOS) : www.boingball.net/AMIGA_FOR_EVER/Codes/C/Butterfly/Butterfly.
- Image bitmap (BMP) utilisée : www.boingball.net/AMIGA_FOR_EVER/Codes/C/Butterfly/Butterfly.bmp.

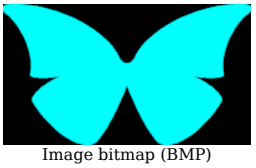


Image bitmap (BMP)

Tous ces fichiers doivent être dans le même répertoire.

Afin d'avoir tout de suite une idée du résultat, n'hésitez pas à lancer le programme exécutable "Butterfly" (pour le quitter : touche "Échap" ou un clic sur la croix de la fenêtre).

Venons-en maintenant à l'explication détaillée du code source, de la première ligne à la dernière ligne.

Attention : le langage C est sensible à la casse, il fait la différence entre les minuscules et les majuscules.

Inclusion du fichier d'en-tête "SDL.h" de la bibliothèque SDL2

```
#include <SDL2/SDL.h>
```

Ce fichier d'en-tête contient les déclarations et les définitions nécessaires pour utiliser les fonctionnalités de SDL2 dans notre programme.

Déclaration des constantes et des variables globales à tout le programme

```
#define MAXW 800
#define MAXH 600
```

Ces lignes déclarent les constantes MAXW à la valeur 800 et MAXH à la valeur 600 qui seront utilisées pour la taille de notre fenêtre (en pixels).

```
SDL_Window *window;
SDL_Renderer *renderer;
SDL_Surface *butterfly;
SDL_Texture *texture_butterfly;
```

Ces lignes déclarent des pointeurs (nom de variable commençant par *) vers des objets SDL2.

- **SDL_Window *window** : déclare le pointeur "window" vers un objet de type SDL_Window (utilisé pour définir notre fenêtre).
- **SDL_Renderer *renderer** : déclare le pointeur "renderer" vers un objet de type SDL_Renderer (utilisé pour effectuer le rendu des graphiques dans notre fenêtre).
- **SDL_Surface *butterfly** : déclare le pointeur "butterfly" vers un objet de type SDL_Surface (utilisé pour charger notre image à partir de son fichier).
- **SDL_Texture *texture_butterfly** : déclare le pointeur "texture_butterfly" vers un objet de type SDL_Texture (une texture est une structure de données utilisée pour stocker des données d'image afin de pouvoir les exploiter dans le rendu/renderer de la fenêtre).

Définition de la fonction "fin"

```
void fin() {
```

Cette fonction sera appelée pour quitter proprement le programme. Elle n'a pas besoin de paramètres (parenthèses vides) et ne renvoie rien (void). Son code est entre accolades {}.

```
if (texture_butterfly) SDL_DestroyTexture(texture_butterfly);
if (butterfly) SDL_FreeSurface(butterfly);
if (renderer) SDL_DestroyRenderer(renderer);
if (window) SDL_DestroyWindow(window);
```

Chacune de ces lignes "if" teste si l'objet en question (texture_butterfly, butterfly, renderer, window) est défini et dans ce cas le détruit avec la fonction SDL2 correspondante (SDL_DestroyTexture, SDL_FreeSurface, SDL_DestroyRenderer, SDL_DestroyWindow). Ceci afin de libérer proprement les ressources allouées à ces objets (et éviter les fuites de mémoire).

```
SDL_Quit();
```

Cette fonction nettoie et libère toutes les ressources utilisées par la bibliothèque SDL2 (celles allouées lors de son initialisation).

```
exit(0);
```

Cette fonction quitte définitivement le programme en renvoyant le code retour 0 (pas d'erreur).

Définition de la fonction "gestion_evenements"

```
void gestion_evenements() {
```

Cette fonction sera appelée pour gérer les évènements éventuellement survenus (frappe de la touche "Échap", fermeture de la fenêtre,...). Elle n'a pas besoin de paramètres (parenthèses vides) et ne renvoie rien (void). Son code est entre accolades {}.

```
SDL_Event event;
```

Cette ligne déclare la variable locale "event" de type SDL_Event. Dans SDL2, les évènements sont représentés par des structures de données de type SDL_Event.

```
while (SDL_PollEvent(&event)) {
```

Tous les évènements survenus (clavier, souris,...) sont en attente dans la file d'attente des évènements de SDL2. La fonction "SDL_PollEvent(&event)" vérifie s'il y a un évènement en attente dans cette file, dans ce cas renvoie 1 sinon 0. La fonction copie également cet évènement dans la structure "event" passée en paramètre entre parenthèses (le & est obligatoire pour pouvoir récupérer en retour la modification de "event").

Cette boucle "while" permet donc d'appeler la fonction SDL_PollEvent tant qu'il y a un évènement dans la file d'attente, et pour chaque "event" récupéré de le traiter (grâce au traitement du "while" qui suit, entre accolades).

```
switch (event.type) {
```

La structure "switch" permet de comparer une variable à plusieurs valeurs (cas/case) et pour chacune de réaliser un traitement spécifique (on pourrait le faire avec des "if"). Ici, la variable comparée est "event.type" contenant le type de l'évènement à traiter.

```
case SDL_KEYDOWN:
   if (event.key.keysym.sym == SDLK_ESCAPE) fin();
   break;
```

Dans le cas où le type de l'évènement est la frappe d'une touche (case SDL_KEYDOWN), si la touche est la touche "Échap" (if (event.key.keysym.sym == SDLK_ESCAPE)) alors la fonction "fin" est appelée pour quitter le programme. Enfin, le "break" permet de sortir du "switch" (car cas traité).

```
case SDL_WINDOWEVENT:
   if (event.window.event == SDL_WINDOWEVENT_CLOSE) fin();
   break;
```

Dans le cas où le type de l'évènement est un évènement sur la fenêtre (case SDL_WINDOWEVENT), si l'évènement est la fermeture de la fenêtre (if (event.window.event == SDL_WINDOWEVENT_CLOSE)) alors la fonction "fin" est aussi appelée pour quitter le programme. Enfin, le "break" permet de sortir du "switch" (car cas traité).

D'autres évènements peuvent survenir (autres touches, autres clics,...), mais ceux-ci ne sont donc pas traités.

Définition de la fonction "main"

```
int main(int argc, char *argv[]) {
```

Cette fonction est la fonction principale du programme. Lors du lancement de celui-ci, c'est la première appelée. Son code est entre accolades {}.

Ses paramètres définis entre parenthèses (int argc, char *argv[]) permettent de gérer les arguments éventuellement transmis au programme lors de son lancement. "argc" contient le nombre d'arguments transmis (entier, type "int") et "argv" la liste des arguments transmis (tableau pointant sur des chaînes de caractères, type "char"). Mais ceux-ci ne seront pas exploités dans notre programme (car lancé sans arguments).

Elle renvoie un entier ("int" devant "main"), le code retour de la fonction "exit" servant à quitter le programme (revoir la fonction "fin" ci-dessus).

```
unsigned int butterfly_w=200, butterfly_h=113, vitesse=10;
```

Cette ligne déclare des variables locales de type entier non signé (unsigned int), donc positives.

- **butterfly_w=200** : variable initialisée à 200 correspondant à la largeur (width) de notre image animée (en pixels).
- **butterfly_h=113** : variable initialisée à 113 correspondant à la hauteur (height) de notre image animée (en pixels).
- **vitesse=10** : variable initialisée à 10 correspondant à la vitesse de déplacement de notre image animée (en pixels aussi).

```
int x=0, y=0, sens_x=vitesse, sens_y=vitesse, butterfly_l=butterfly_w, battements=20;
```

Cette ligne déclare des variables locales de type entier signé (int), donc positives ou négatives.

- **x=0, y=0** : variables initialisées à 0 correspondant à la position (horizontale et verticale) de notre image animée (en pixels).
- **sens_x=vitesse, sens_y=vitesse** : variables initialisées à la valeur de la variable "vitesse" (ci-dessus) correspondant aux sens de déplacement (horizontal et vertical) de notre image animée (en pixels aussi). Car notre image changera de sens de déplacement à l'écran.
- **butterfly_l=butterfly_w** : variable initialisée à la valeur de la variable "butterfly_w" (ci-dessus) correspondant à la largeur de notre image qui diminuera et augmentera (en pixels). Car notre papillon battra des ailes.
- **battements=20** : variable initialisée à 20 correspondant à la vitesse des battements d'ailes de notre papillon (en pixels aussi).

Précisions

- La position initiale "x=0, y=0" correspond au coin supérieur gauche de la fenêtre.
- Plus la vitesse et les battements sont élevés (en pixels), plus notre papillon se déplacera rapidement à l'écran et battra vite des ailes, et inversement.

```
SDL_Init(SDL_INIT_VIDEO);
```

Cette fonction initialise la bibliothèque SDL2 pour utiliser la vidéo (SDL_INIT_VIDEO) afin de créer des fenêtres et effectuer leurs rendus graphiques. Il pourrait y avoir d'autres paramètres, par exemple pour utiliser l'audio.

```
window = SDL_CreateWindow("Butterfly", SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED, MAXW, MAXH, SDL_WINDOW_SHOWN);
```

Cette fonction crée notre fenêtre "window" (pointeur de fenêtre déclaré plus haut) avec les paramètres entre parenthèses.

- **"Butterfly"** : titre de la fenêtre, affiché dans sa barre de titre.
- **SDL_WINDOWPOS_UNDEFINED (deux fois, pour x et y)** : position indéfinie de la fenêtre à l'écran. Sans définir soi-même de valeurs (en pixels et par rapport au coin supérieur gauche de l'écran), c'est SDL2 qui choisit automatiquement une position appropriée pour la fenêtre.
- **MAXW, MAXH** : largeur et hauteur de la fenêtre (constantes déclarées plus haut avec les valeurs 800x600).
- **SDL_WINDOW_SHOWN** : la fenêtre est visible à l'écran dès sa création (il y a d'autres paramètres).

```
renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);
```

Cette fonction crée le moteur de rendu "renderer" (pointeur de renderer déclaré plus haut) pour notre fenêtre "window" avec les paramètres entre parenthèses.

- **window** : fenêtre à laquelle le renderer est associé.
- **-1** : signifie que le premier pilote graphique disponible sera utilisé par SDL2.
- **SDL_RENDERER_ACCELERATED** : indique que le rendu sera accéléré par matériel (carte graphique), si possible.
- **SDL_RENDERER_PRESENTVSYNC** : active la synchronisation verticale V-Sync pour améliorer la fluidité des animations (en synchronisant le taux de rafraîchissement de la fenêtre avec celui du moniteur). Grâce à V-Sync, la vitesse de défilement est aussi maintenue relativement constante, indépendamment de la puissance du microprocesseur de la machine utilisée (tant que celle-ci n'est pas surchargée).

"SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC" active les deux options (ce n'est pas l'une ou l'autre).

Ce renderer sera donc utilisé pour effectuer le rendu des graphiques dans notre fenêtre.

```
SDL_SetRenderDrawColor(renderer, 0, 0, 0, SDL_ALPHA_OPAQUE);
```

Cette fonction définit la couleur de dessin de notre renderer : noire (0, 0, 0) et sans transparence (SDL_ALPHA_OPAQUE).

La fonction "SDL_RenderClear(renderer);" (plus bas) effacera donc le renderer en noir, d'où le fond noir de la fenêtre.

Cette fonction est juste là pour vous introduire cette fonctionnalité. On aurait pu s'en passer car la couleur de dessin d'un renderer est déjà par défaut le noir.

```
butterfly = SDL_LoadBMP("Butterfly.bmp");
```

Cette fonction charge l'image bitmap (BMP) du fichier "Butterfly.bmp" en objet surface "butterfly" (pointeur de surface déclaré plus haut).

```
texture_butterfly = SDL_CreateTextureFromSurface(renderer, butterfly);
```

Cette fonction crée la texture "texture_butterfly" pour notre renderer (pointeur de texture déclaré plus haut), cela à partir des données de la surface "butterfly" (de notre image).

Pour rappel : une texture est une structure de données utilisée pour stocker des données d'image afin de pouvoir les exploiter dans le rendu/renderer de la fenêtre.

```
SDL_Rect srcrect_butterfly = { 0, 0, butterfly_w, butterfly_h };
```

Cette ligne crée la structure "srcrect_butterfly" de type SDL_Rect. Celle-ci contient les valeurs du rectangle de notre image d'origine (en pixels).

- **0, 0** : point de départ (x, y), coin supérieur gauche de l'image.
- **butterfly_w, butterfly_h** : largeur et hauteur de l'image (variables déclarées ci-dessus avec les valeurs butterfly_w=200 et butterfly_h=113).

Cette structure "srcrect_butterfly" servira à préciser les valeurs de départ (source) de notre texture "texture_butterfly" qu'on va utiliser/transformer dans le renderer.

```
while (1) {
```

Ce "while" est une boucle infinie (1 correspond à "vrai"). C'est elle qui gère le renderer et donc l'animation

de l'image dans la fenêtre. Elle n'est quittée que par la frappe de la touche "Échap" ou par la fermeture de la fenêtre. Son code est entre accolades {}.

Le principe de ce genre de traitement est toujours le même avec SDL2, pour chaque cycle :

- Gérer les évènements éventuellement survenus (clavier, souris,...).
- Effacer le renderer.
- Déterminer les nouveaux éléments graphiques pour le renderer (position et transformation des textures, dessin d'autres éléments).
- Mettre à jour l'affichage de la fenêtre avec tout ce qui a été rendu sur le renderer.

A grande vitesse, cela produit l'animation (avec la prise en compte rapide des éventuels évènements).

```
gestion_evenements();
```

Notre fonction qui gère les évènements éventuellement survenus (frappe de la touche "Échap", fermeture de la fenêtre,...), voir plus haut.

```
SDL_RenderClear(renderer);
```

Cette fonction efface le contenu actuel du renderer, en le remplissant avec la couleur de fond actuellement définie (le noir défini plus haut par la fonction SDL_SetRenderDrawColor).

```
x = x + sens_x;
```

Incrémente la position horizontale "x" de notre image de la valeur "sens_x" (initialisée plus haut à la valeur de la variable "vitesse").

```
y = y + sens_y;
```

Incrémente la position verticale "y" de notre image de la valeur "sens_y" (initialisée plus haut à la valeur de la variable "vitesse").

Notre papillon se déplacera donc en diagonale, d'autant plus rapidement que la valeur de "vitesse/sens_x /sens_y" est grande.

```
if (x < 0 || x > MAXW) sens_x = -sens_x;
```

Si la position horizontale "x" de notre image est inférieure à 0 ou supérieure à "MAXW" (largeur maximale de la fenêtre), alors le sens de déplacement horizontal de notre image change (sens_x = -sens_x : valeur opposée de "sens_x").

```
if (y < 0 || y > MAXH-butterfly_h) sens_y = -sens_y;
```

Si la position verticale "y" de notre image est inférieure à 0 ou supérieure à "MAXH-butterfly_h" (hauteur maximale de la fenêtre moins la hauteur de notre image papillon), alors le sens de déplacement vertical de notre image change (sens_y = -sens_y : valeur opposée de "sens_y").

Notre papillon rebondira donc sur les bords de la fenêtre (droit/0 et gauche/MAXW, haut/0 et bas/MAXH-butterfly_h), sans sortir de celle-ci.

```
butterfly_l = butterfly_l + battements;
```

Augmente la largeur actuelle de notre image papillon "butterfly_l" de la valeur "battements".

```
if (butterfly_l < 0 || butterfly_l > butterfly_w) battements = -battements;
```

Si la largeur actuelle de notre image papillon "butterfly_l" est inférieure à 0 ou supérieure à "butterfly_w" (largeur maximale de notre image papillon), alors le sens du battement d'ailes change (battements = -battements : valeur opposée de "battements").

Notre papillon battra donc des ailes, d'autant plus vite que la valeur de "battements" est grande.

```
SDL_Rect dstrect_butterfly = { x-butterfly_l/2, y, butterfly_l, butterfly_h };
```

Cette ligne crée la structure "dstrect_butterfly" de type SDL_Rect. Celle-ci contient les valeurs du rectangle de notre image résultat (en pixels), déterminées à partir de celles qu'on vient de calculer.

- **x-butterfly_l/2** : position horizontale de l'image (il faut soustraire "butterfly_l/2" à "x" sinon le déplacement ne sera pas régulier, car la largeur de notre image papillon "butterfly_l" change).
- **y** : position verticale de l'image.
- **butterfly_l** : largeur de l'image (qui change).

- **butterfly_h** : hauteur de l'image (qui ne change pas).

Précision

La position (horizontale et verticale) est toujours déterminée par rapport au coin supérieur gauche de l'image (et non son centre).

```
SDL_RenderCopy(renderer, texture_butterfly, &srcrect_butterfly, &dstrect_butterfly);
```

Cette fonction copie sur notre "renderer" le rectangle résultat "dstrect_butterfly" (image transformée), dont les données sont issues de notre texture "texture_butterfly" et sélectionnées par le rectangle de départ "srcrect_butterfly" (correspondant à notre image d'origine).

Dans notre programme, le rectangle de départ "srcrect_butterfly" prend toute la texture (car on veut toute l'image), mais on pourrait en prendre qu'une partie. Toutes les transformations entre source et destination sont possibles, permettant tous types d'animations.

Pour le même cycle, on pourrait copier sur le renderer d'autres textures transformées, voire dessiner d'autres éléments (lignes, cercles,...). Tous les rendus graphiques sont donc possibles.

```
SDL_RenderPresent(renderer);
```

Enfin, cette fonction met à jour l'affichage de la fenêtre avec tout ce qui a été rendu sur le renderer.

Fin du code source.

Compilation et exécution du programme

Une fois le fichier du code source enregistré (avec le nom "Butterfly.c"), il faut le compiler pour obtenir son fichier exécutable. Il faudra également le recompiler après chaque modification du code source.

Pour cela, ouvrir un terminal Shell et aller dans le répertoire contenant ce fichier (à l'aide de la commande de changement de répertoire "cd").

Pour rappel : le fichier de l'image utilisée "Butterfly.bmp" doit être présent dans ce même répertoire (vérifier avec la commande "dir").

Puis taper la commande suivante pour le compiler :

```
gcc Butterfly.c `sdl2-config --cflags --libs` -o Butterfly
```

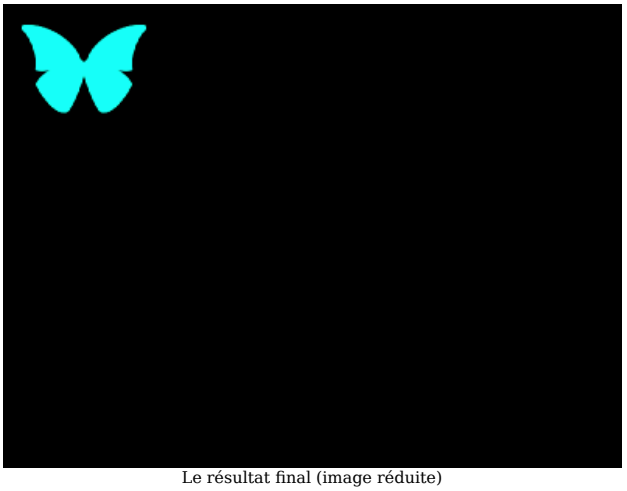
- **gcc** : commande du compilateur GCC permettant de compiler un fichier de code source C en fichier exécutable.
- **Butterfly.c** : nom du fichier source à compiler.
- **`sdl2-config --cflags --libs`** : paramètre de compilation nécessaire pour utiliser la bibliothèque SDL2 (attention au symbole accent grave " ` ", ce n'est pas une apostrophe " ' ").
- **-o** : option permettant de spécifier le nom du fichier exécutable généré.
- **Butterfly** : nom du fichier exécutable généré (sans extension).

Il n'y a pas de message d'erreur quand la compilation se passe bien. Dans le cas contraire, corriger le code source et recompiler.

Pour lancer le programme, il suffit de taper le nom de son fichier exécutable "Butterfly", ou de double-cliquer sur son icône dans le dossier concerné de l'interface graphique.

Et vous devriez voir apparaître la fenêtre avec notre joli morpho bleu qui papillonne à l'écran. :)

L'Amiga a sa Boing Ball, MorphOS a maintenant son Flying Morpho ! ^^



Le résultat final (image réduite)

Précision

L'animation doit être fluide. Si elle est saccadée, relancer le programme après avoir décoché l'option "Affichage amélioré" ("Enhanced display" en anglais) en éditant votre écran dans les préférences de MorphOS.

Code source complet

Pour conclure le tout, voici le code source complet du programme :

```
#include <SDL2/SDL.h>

#define MAXW 800
#define MAXH 600

SDL_Window *window;
SDL_Renderer *renderer;
SDL_Surface *butterfly;
SDL_Texture *texture_butterfly;

void fin() {
   if (texture_butterfly) SDL_DestroyTexture(texture_butterfly);
   if (butterfly) SDL_FreeSurface(butterfly);
   if (renderer) SDL_DestroyRenderer(renderer);
   if (window) SDL_DestroyWindow(window);

   SDL_Quit();

   exit(0);
}

void gestion_evenements() {
   SDL_Event event;

   while (SDL_PollEvent(&event)) {
      switch (event.type) {
         case SDL_KEYDOWN:
            if (event.key.keysym.sym == SDLK_ESCAPE) fin();
            break;
         case SDL_WINDOWEVENT:
            if (event.window.event == SDL_WINDOWEVENT_CLOSE) fin();
            break;
      }
   }
}

int main(int argc, char *argv[]) {
   unsigned int butterfly_w=200, butterfly_h=113, vitesse=10;
   int x=0, y=0, sens_x=vitesse, sens_y=vitesse, butterfly_l=butterfly_w, battements=20;

   SDL_Init(SDL_INIT_VIDEO);

   window = SDL_CreateWindow("Butterfly", SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED, MAXW, MAXH, SDL_WINDOW_SHOWN);
   renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);
   SDL_SetRenderDrawColor(renderer, 0, 0, 0, SDL_ALPHA_OPAQUE);

   butterfly = SDL_LoadBMP("Butterfly.bmp");
   texture_butterfly = SDL_CreateTextureFromSurface(renderer, butterfly);
   SDL_Rect srcrect_butterfly = { 0, 0, butterfly_w, butterfly_h };

   while (1) {
      gestion_evenements();

      SDL_RenderClear(renderer);

      x = x + sens_x;
      y = y + sens_y;
      if (x < 0 || x > MAXW) sens_x = -sens_x;
      if (y < 0 || y > MAXH-butterfly_h) sens_y = -sens_y;

      butterfly_l = butterfly_l + battements;
      if (butterfly_l < 0 || butterfly_l > butterfly_w) battements = -battements;

      SDL_Rect dstrect_butterfly = { x-butterfly_l/2, y, butterfly_l, butterfly_h };
      SDL_RenderCopy(renderer, texture_butterfly, &srcrect_butterfly, &dstrect_butterfly);

      SDL_RenderPresent(renderer);
   }
}
```

[Retour en haut] / [Retour aux articles]