

Utiliser SDL2 avec eForth Windows

version 1.2 - mercredi 6 novembre 2024



Auteur

- Marc PETREMANN

Contents

Auteur.....	1
Introduction.....	3
Installation de eForth et SDL2.....	4
Installation de eForth pour Windows.....	4
Installation de la librairie SDL2.....	5
Organisation des fichiers.....	6
Le fichier main.fs.....	7
Edition et gestion des fichiers sources pour eForth.....	10
Les éditeurs de fichiers texte.....	10
Utiliser un IDE.....	10
Stockage sur GitHub.....	12
Quelques bonnes pratiques.....	13
Création du vocabulaire SDL2.....	15
Création et utilisation d'un ticket de liaison logicielle.....	15
Définition du vocabulaire SDL2.....	15
Définition des liaisons logicielles.....	16
Passage des paramètres vers les liaisons logicielles.....	16
Comprendre la documentation SDL2.....	17
Initialisation de l'environnement SDL2.....	18
Quitter l'environnement SDL2.....	19
Gestion des erreurs.....	20
Les structures dans SDL2.....	22
Utilisation d'une structure.....	22
Lire et écrire dans une structure.....	23
Créer une fenêtre avec SDL2.....	24
Initialiser SDL.....	24
Créer une fenêtre.....	24
Redimensionner, réduire et restaurer une fenêtre.....	27
Visibilité des fenêtres.....	27
Afficher une image BMP avec SDL2.....	29
Chargement d'une image BMP.....	29
Affichage de l'image.....	30
Contrôle des dimensions d'affichage d'une image.....	31
Ressources.....	33
GitHub.....	33
SDL.....	33

Introduction

SDL2, ou Simple DirectMedia Layer 2, est une bibliothèque multiplate-forme conçue pour la gestion des graphiques, du son, des entrées et d'autres éléments multimédias. Elle est largement utilisée pour le développement de jeux vidéo et d'applications multimédias.

Voici quelques caractéristiques clés de SDL2 :

1. Multiplateforme : SDL2 fonctionne sur Windows, macOS, Linux, iOS et Android, ce qui permet aux développeurs de créer des applications qui s'exécutent sur plusieurs systèmes d'exploitation.
2. Gestion des graphiques : SDL2 permet de dessiner des graphiques en 2D et de gérer des textures, des surfaces et des images.
3. Support audio : La bibliothèque permet de jouer des sons et de la musique, facilitant l'intégration audio dans les applications.
4. Gestion des entrées : SDL2 gère les entrées provenant de claviers, souris, manettes de jeu et autres dispositifs.
5. Extensibilité : SDL2 est souvent utilisée avec d'autres bibliothèques pour étendre ses fonctionnalités, comme OpenGL pour les graphiques 3D.

En résumé, SDL2 est un outil puissant et flexible pour les développeurs souhaitant créer des applications multimédias et des jeux.

Installation de eForth et SDL2

L'installation de l'environnement de développement en langage Forth pour SDL2 ne nécessite que deux composants :

- la version eForth pour Windows de Brad NELSON ;
- la librairie SDL2 dans un fichier dll ;
- Un bon éditeur de fichiers texte.

C'est un environnement très compact qui associe, grâce à eForth, un interpréteur et un compilateur.

Installation de eForth pour Windows

La version eForth pour Windows est disponible ici :

<https://eforth.appspot.com/windows.html>

Téléchargez la version **uEf64-7.0.7.20.exe**. C'est une version 64 bits. Elle est très stable et très robuste. Le fichier ne fait que 265 Ko.

Créez un dossier **eforth** dans votre espace de travail habituel :

 eforth

Copiez le fichier précédemment téléchargé dans ce dossier :

 eforth
 uEf64-7.0.7.20.exe

Exécutez ce programme depuis ce dossier **eforth**. Souvent, Windows émet un avertissement. Si c'est le cas, acceptez l'exécution de ce programme. Vous devez vous retrouver avec cette fenêtre :

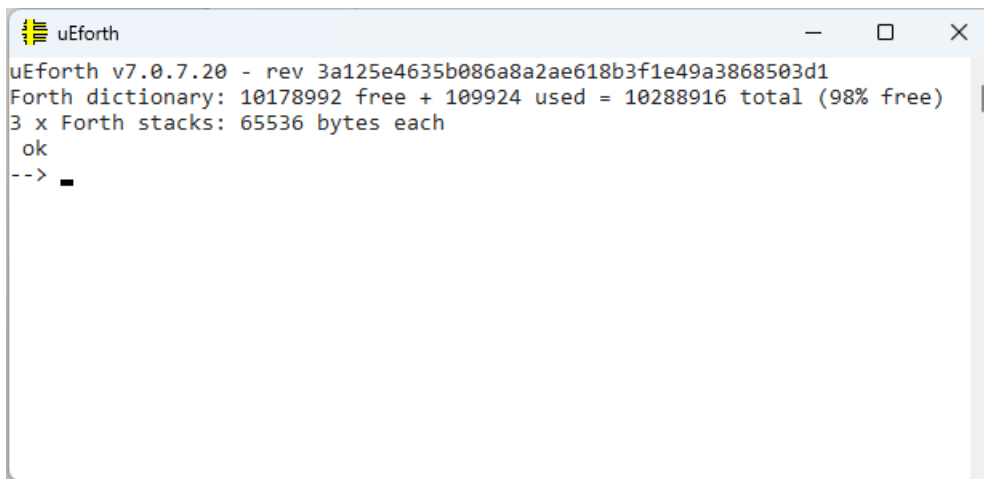


Figure 1: fenêtre eForth Windows

Voilà ! Vous avez la main sur une version Forth disposant de trois piles :

- une pile de données
- une pile de retour
- une pile pour les nombres réels

Chaque pile dispose d'un espace de données de 64Ko. Comme chaque élément pèse 64 bits (8 octets) dans la pile, ce sont 8000 valeurs qui peuvent être empiilées !

Le dictionnaire dispose d'un espace libre de 10.178.992 octets ! On dispose donc d'un espace de développement plus que confortable.

eForth c'est un interpréteur ET un compilateur. Le tout premier mot à votre disposition, histoire de voir ce qu'il y sous le capot est **words** dont l'exécution affiche ceci, résumé aux trois premières lignes :

```
FORTH graphics argv argc visual set-title page at-xy normal bg fg ansi
editor list copy thru load flush update empty-buffers buffer block save-buffers
default-use use open-blocks block-id scr block-fid file-exists? needs required...
...etc.
```

Tous ces mots constituent le langage permettant de compiler et exécuter les programmes écrits en langage Forth.

Installation de la librairie SDL2

Accès au site SDL2 : <https://www.libsdl.org/>

Ce site fournit beaucoup de ressources et documentations pour comprendre l'utilisation de la librairie SDL2. ATTENTION : les fonctions en langage C sont adaptées à eForth Windows. Nous verrons ceci plus loin.

Pour télécharger la version SDL2 la plus récente :

<https://github.com/libsdl-org/SDL/releases/tag/release-2.30.8>

Récupérez le fichier **SDL2-2.30.8-win32-x64.zip**.

Ouvrez ce fichier zip et transférez le seul fichier **SDL2.dll** dans le répertoire eforth :

```

└─ eforth
   └─ uEf64-7.0.7.20.exe
   └─ SDL2.dll
```

Voilà ! Il n'y a rien d'autre à faire coté installation. Voyons maintenant comment préparer l'environnement de développement.

Organisation des fichiers

Il faut maintenant organiser l'espace de développement. Pour ce faire, on crée un sous-répertoire **SDL2** :

```

└─ eforth
   └─ uEf64-7.0.7.20.exe
   └─ SDL2.dll
   └─ SDL2
```

On va remplir ce répertoire avec les fichiers disponibles ici :

<https://github.com/MPETREMANN11/SDL2-eForth-windows/tree/main/SDL2>

Ne récupérez que ces fichiers :

```

└─ eforth
   └─ uEf64-7.0.7.20.exe
   └─ SDL2.dll
   └─ SDL2
      └─ SDL2.fs
      └─ SDLconstants.fs
      └─ main.fs
      └─ tests.fs
```

Contenu de ces fichiers :

- **SDL2.fs** contient tous les mots accédant à la librairie SDL contenue dans **SDL2.dll** ;
- **SDLconstants.fs** contient un certain nombre de constantes d'usage courant et propres à leur emploi avec les mots définis dans le vocabulaire **SDL2** ;
- **main.fs** est le script principal chargé d'agréger les divers composants de votre application ;
- **tests.fs** fichier fourre-tout pour réaliser des tests.

ATTENTION : le contenu de ces fichiers est susceptibles d'évoluer en permanence sur le dépôt Github. Il est donc fortement conseillé d'en surveiller le contenu.

Il reste un dernier fichier à installer dans le répertoire **eforth** :

```

└─ eforth
   └─ uEf64-7.0.7.20.exe
   └─ SDL2.dll
   └─ SDL2
      └─ SDL2.fs
      └─ SDLconstants.fs
      └─ main.fs
      └─ tests.fs
   └─ SDL2.fs
```

Contenu de ce fichier **SDL2.fs**

```
\ pre-load tools
\ s" tools/dumpTool.fs" required
\ load SDL2
s" SDL2/main.fs" included
```

Considérez le contenu de ce fichier comme un traitement par lot, mais exécutable seulement depuis eForth.

Pour vérifier si la librairie SDL2 fonctionne bien, lancez eForth, puis entrez cette commande :

```
include SDL2.fs
```

Cette commande charge le contenu du fichier **SDL2.fs** situé dans la racine du sous-répertoire **eforth**. Tapez ensuite :

```
SDL2 vlist
```

Vous devez voir apparaître les mots définis dans le vocabulaire SDL2, ici les trois premières lignes :

```
SDL.CreateWindow SDL.init SetRenderDrawColor Quit RenderPresent RenderClear
PollEvent Init GetError GetCursor GetCPUCount GetBasePath GetAudioStatus
DestroyWindow DestroyRenderer CreateWindow CreateRenderer SDL_MAX_LOG_MESSAGE
...etc...
```

ATTENTION : le contenu de ce vocabulaire évolue en permanence. Il ne représente que la compilation des définitions décrites dans le fichier **SDL2/SDL2.fs**.

Le fichier main.fs

C'est le fichier contenant le script chargé d'agréger les composants de vos applications. Exemple de contenu de ce fichier :

```

\ load SDL2 library
s" SDL2.fs"      included

\ load SDL2 tests
s" tests.fs"     included

```

Vous pouvez modifier son contenu sans souci. Prenons le cas où vous voulez tester l’affichage de fenêtres. Vous créez un fichier **testWindows.fs** dans lequel vous allez mettre les tests de fenêtres avec les mots du vocabulaire **SDL2**. Voici comment intégrer ce fichier **testWindows.fs** dans **main.fs** :

```

\ load SDL2 library
s" SDL2.fs"      included

\ load SDL2 tests
\ s" tests.fs"    included

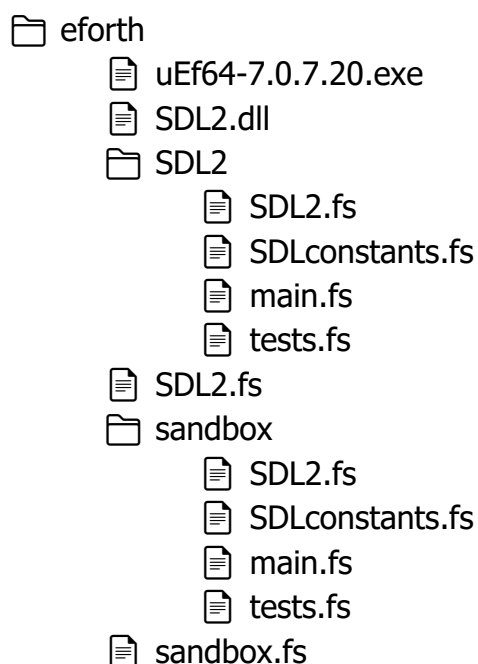
\ load windows tests with SDL2
s" testWindowss.fs" included

```

Le mot `\` met en commentaire le reste de la ligne. Au prochain chargement de **SDL2.fs** qui est dans le répertoire racine **eforth**, seul le contenu des fichiers **SDL2/SDL2.fs** et **SDL2/testWindows.fs** sera traité par eForth.

Vous pouvez donc facilement enchaîner des tests ou des portions de code pour l’application finale.

Pour éviter l’écrasement accidentel de votre travail, il est conseillé de créer plusieurs sous-répertoires, par exemple sandbox où vous pourrez faire plein de petits tests :



Exemple du contenu de **sandbox.fs** :


```
\ s" tools/dumpTool.fs" required
\ load sandbox
s" sandbox/main.fs" included
```

Ainsi, au lancement de eForth, il suffira de saisir :

```
include sandbox.fs
```

A vous de vous organiser pour être le plus efficace lors de vos développements.

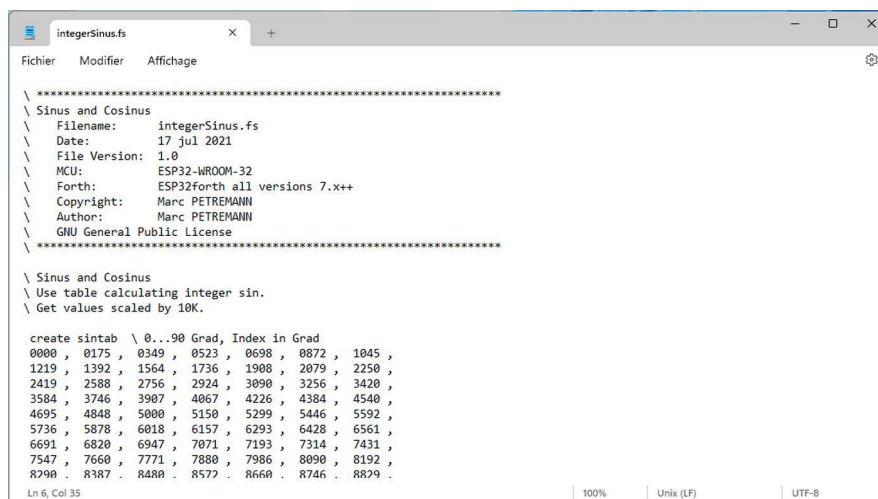
Edition et gestion des fichiers sources pour eForth

Comme pour la très grande majorité des langages de programmation, les fichiers sources écrits en langage Forth sont au format texte simple. L'extension des fichiers en langage Forth est libre :

- **txt** extension générique pour tous les fichiers texte ;
- **forth** utilisé par certains programmeurs Forth ;
- **fth** forme compressée pour Forth ;
- **4th** autre forme compressée pour Forth ;
- **fs** notre extension préférée...

Les éditeurs de fichiers texte

Sous Windows, l'éditeur de fichiers **edit** est le plus simple :



```
integerSinus.fs
Fichier  Modifier  Affichage

*****
\ Sinus and Cosinus
\ Filename:   integerSinus.fs
\ Date:      17 jul 2021
\ File Version: 1.0
\ MCU:       ESP32-WROOM-32
\ Forth:     ESP32Forth all versions 7.x++
\ Copyright:  Marc PETREMANN
\ Author:    Marc PETREMANN
\ GNU General Public License
*****

\ Sinus and Cosinus
\ Use table calculating integer sin.
\ Get values scaled by 10K.

create sintab \ 0...90 Grad, Index in Grad
0000 , 0175 , 0349 , 0523 , 0698 , 0872 , 1045 ,
1219 , 1392 , 1564 , 1736 , 1908 , 2079 , 2250 ,
2419 , 2588 , 2756 , 2924 , 3090 , 3256 , 3420 ,
3584 , 3746 , 3907 , 4067 , 4226 , 4384 , 4540 ,
4695 , 4848 , 5000 , 5150 , 5299 , 5446 , 5592 ,
5736 , 5878 , 6018 , 6157 , 6293 , 6428 , 6561 ,
6691 , 6820 , 6947 , 7071 , 7193 , 7314 , 7431 ,
7547 , 7660 , 7771 , 7880 , 7986 , 8090 , 8192 ,
8290 , 8387 , 8480 , 8572 , 8660 , 8746 , 8829 ,

Ln 6, Col 35      100%      Unix (LF)      UTF-8
```

Figure 2: édition avec edit sous windows 11

Les autres éditeurs, comme **WordPad** sont déconseillés, car vous risquez de sauvegarder le code source en langage Forth dans un format de fichier non compatible avec eForth.

Si vous utilisez une extension de fichier personnalisé, comme **fs**, pour vos fichiers source en langage Forth, il faut faire reconnaître cette extension de fichiers par votre système pour permettre leur ouverture par l'éditeur de texte.

Utiliser un IDE

Rien ne vous empêche d'utiliser un IDE¹. Pour ma part, j'ai une préférence pour **Netbeans** que j'utilise aussi pour PHP, MySQL, Javascript, C, assembleur... C'est un IDE très puissant et aussi performant qu'**Eclipse** :

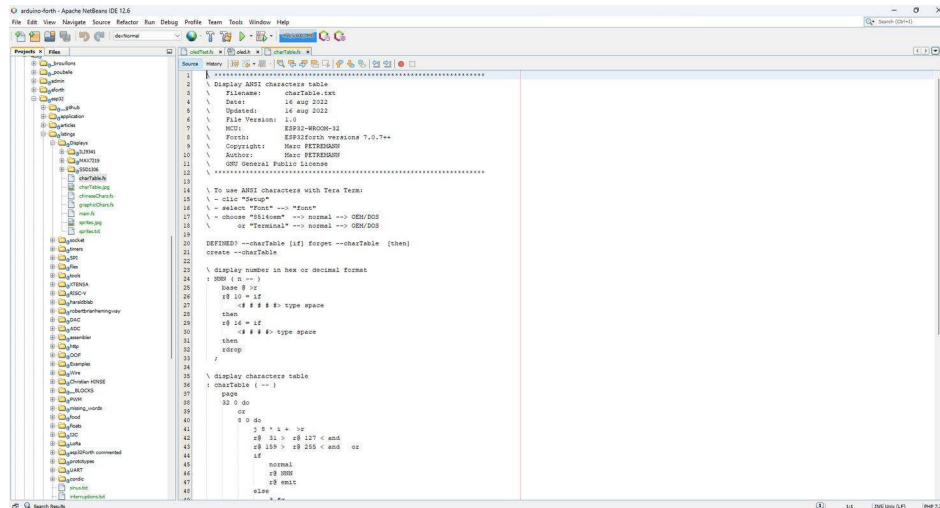


Figure 3: édition avec Netbeans

Netbeans offre plusieurs fonctionnalités intéressantes :

- gestion de versions avec **GIT** ;
- récupération de versions précédentes de fichiers modifiés ;
- comparaison de fichiers avec **Diff** ;
- transmission en un clic en **FTP** vers l'hébergement en ligne de votre choix ;

Avec l'option **GIT**, possibilité de partager des fichiers sur un dépôt et de gérer des collaborations sur des projets complexes. En local ou en collaboration, **GIT** permet la gestion des versions différentes d'un même projet, puis de fusionner ces versions. Vous pouvez créer votre dépôt GIT local. A chaque *Commit* d'un fichier ou d'un répertoire complet, les développements sont conservés en l'état. Ceci permet de retrouver d'anciennes versions d'un même fichier ou dossier de fichiers.

1 Integrated Development Environment

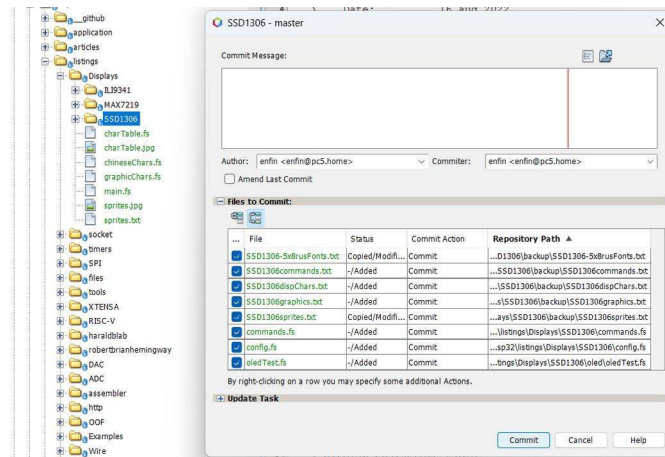


Figure 4: opération GIT dans Netbeans

Avec NetBeans, vous pouvez définir un embranchement de développement pour un projet complexe. Ici on crée une nouvelle branche :

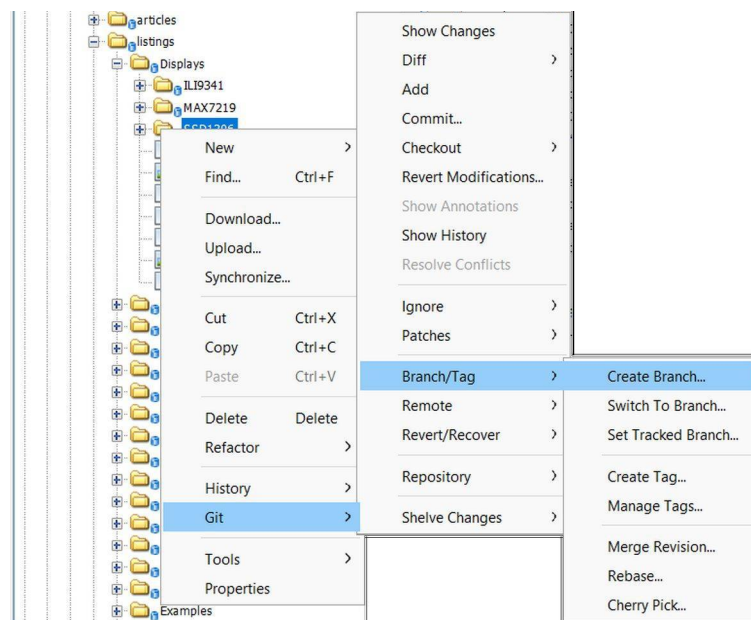


Figure 5: création d'une branche sur un projet

Exemple de situation qui justifie la création d'une branche :

- vous avez un projet fonctionnel ;
- vous envisagez de l'optimiser ;
- créez une branche et faites les optimisations dans cette branche...

Les modifications de fichiers sources dans une branche n'ont pas d'influence sur les fichiers dans le tronc *main*.

Accessoirement, il est plus que conseillé de disposer d'un support physique de sauvegarde. Un disque dur SSD c'est environ 50€ pour 300Gb d'espace de stockage. La vitesse d'accès en lecture ou écriture d'un support SSD est tout simplement bluffante !

Stockage sur GitHub

Le site web **GitHub**² est, avec **SourceForge**³, un des meilleurs endroits pour stocker ses fichiers sources. Sur GitHub, vous pouvez mettre un dossier de travail en commun avec d'autres développeurs et gérer des projets complexes. L'éditeur Netbeans peut se connecter au projet et vous permet de transmettre ou récupérer des modifications de fichiers.

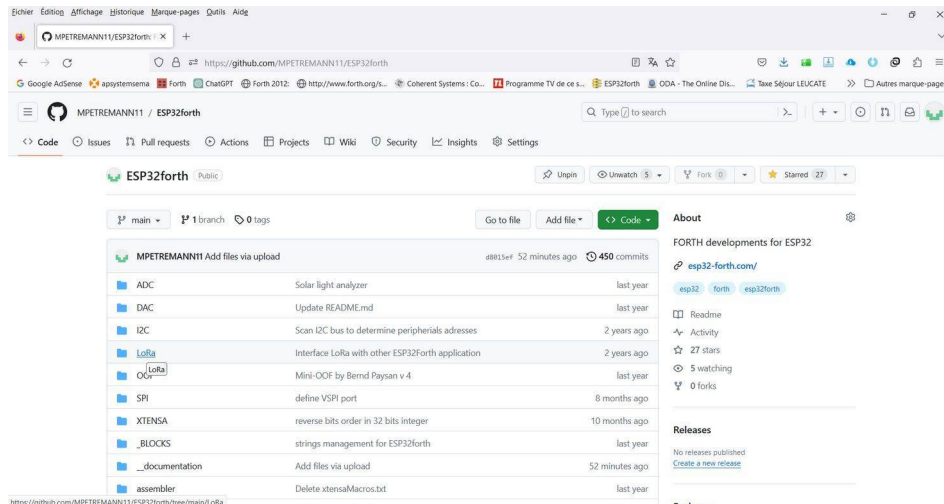


Figure 6: stockage des fichiers sur Github

Sur **GitHub**, vous pouvez gérer des embranchements de projets (*fork*). Vous pouvez aussi rendre confidentiel certaines parties de vos projets. Ici les branches dans les projets de flagxor/ueforth :

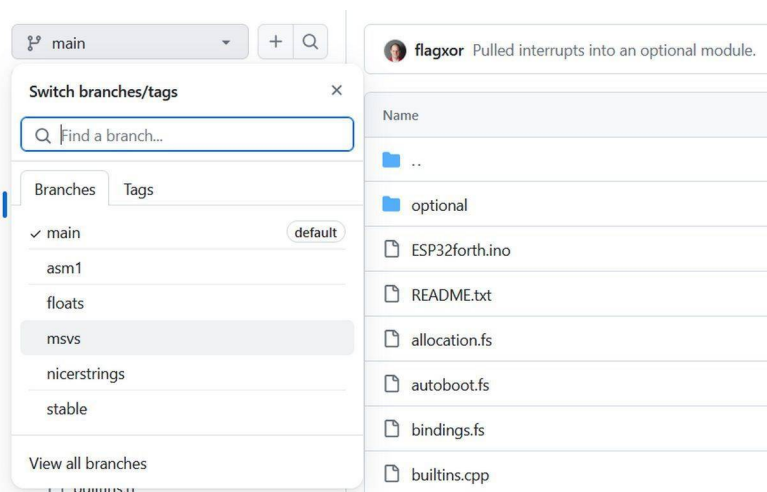


Figure 7: accès à une branche de projet

Quelques bonnes pratiques

La première bonne pratique consiste à bien nommer ses fichiers et dossiers de travail. Vous développez pour eForth, donc créez un dossier nommé **eforth**.

² <https://github.com/>

³ <https://sourceforge.net/>

Pour les essais divers, créez dans ce dossier un sous-dossier **sandbox** (bac à sable).

Pour les projets bien construits, créez un dossier par projet. Par exemple, vous voulez créer un jeu de cartes, créez le sous-dossier **cards**.

Si vous avez des scripts d'usage général, créez un dossier **tools**. Si vous utilisez un fichier de ce dossier **tools** dans un projet, copiez et collez ce fichier dans le dossier de ce projet. Ceci évitera qu'une modification d'un fichier dans **tools** ne perturbe ensuite votre projet.

La seconde bonne pratique consiste à répartir le code source d'un projet dans plusieurs fichiers :

- **config.fs** pour stocker les paramètres du projet ;
- répertoire **documentation** pour stocker des fichiers dans le format de votre choix, en rapport avec la documentation du projet ;
- **myApp.fs** pour les définitions de votre projet. Choisissez un nom de fichier assez explicite . Par exemple, pour gérer un jeu **pacman**, prenez le nom **pacman-commands.fs**.

..	
LOTTOinterface.jpg	Add files via upload
README.md	Create README.md
euroMillionFR.fs	LOTO wining combinaisons numbers
generalWords.fs	general words for LOTTO program
gridsManage.fs	Manage content of LOTTO grids
interface.fs	text interface for LOTTO program
main.fs	LOTTO game main file
numbersFrequency.fs	stats frequency for LOTTO numbers

Figure 8: exemple de nommage de fichiers source Forth

C'est le contenu de ces fichiers qui sera traité par eForth via **include** pour que eForth interprète et compile le code Forth.

Création du vocabulaire SDL2

Nous allons voir comment est assurée la liaison logicielle entre eForth Windows et les fonctions disponibles dans **SDL2.dll**.

Le mot clé est **dll**. Ce mot est disponible dans le vocabulaire **windows**. Il crée un ticket d'accès vers des ressources Windows. Si vous avez correctement installé eForth windows et **SDL2.dll**, on va créer notre ticket d'accès :

```
windows
\ Entry point to SDL2.dll library
z" SDL2.dll" dll SDL2.dll
```

Ici, notre ticket a été nommé **SDL2.dll**. On aurait pu le nommer plus simplement **SDL2** ou **SDL2ticket...**

Création et utilisation d'un ticket de liaison logicielle

Reprenons la ligne qui crée le ticket de liaison logicielle :

```
z" SDL2.dll" dll SDL2.dll
```

On y voit :

- en bleu, le nom du fichier de la librairie à laquelle on va se connecter par liaison logicielle ;
- en vert le mot **dll** qui car créer le ticket de liaison logicielle ;
- en rouge notre ticket de liaison logicielle.

Définition du vocabulaire SDL2

Afin d'éviter toute collision logicielle avec des mots prédéfinis du vocabulaire forth, il est fortement recommandé de définir tous les mots propres à la liaison logicielle dans un vocabulaire spécifique :

```
vocabulary SDL2
```

Ceci crée le vocabulaire **SDL2**. Tous les mots de liaison logicielle avec SDL2.dll seront définis dans ce vocabulaire. Pour créer ces nouvelles définitions, nous devons accéder aux autres vocabulaires **forth**, **windows** et **structures** :

```
only FORTH also windows also structures also
```

Et enfin, on force la définition des nouveaux mots dans ce vocabulaire **SDL2** :

```
SDL2 definitions
```


Voici la séquence complète des opérations de définition et exploitation de ce vocabulaire SDL2 :

```
vocabulary SDL2
only FORTH also windows also structures also
SDL2 definitions

\ Entry point to SDL2.dll library
z" SDL2.dll" dll SDL2.dll
SDL2
```

Si on exécute **vlist**, on doit voir ceci :

```
--> vlist
SDL2.dll
```

Ici, **SDL2.dll** est le premier mot dans notre vocabulaire **SDL2**.

Définition des liaisons logicielles

Notre ticket **SDL2.dll** est devenu un mot de définition. Pour créer le mot **GetError** :

```
\ Retrieve a message about the last error that occurred on the current thread
z" SDL_GetError"          0 SDL2.dll GetError ( -- zstr )
```

On retrouve ici :

- en bleu, le nom de la fonction tel qu'il est défini dans la librairie **SDL2.dll** sous Windows ;
- en vert le nombre de paramètres à passer et le ticket de liaison logicielle ;
- en rouge le nouveau mot créé dans le vocabulaire **SDL2**.

On peut vérifier la création de **GetError** en tapant **vlist** qui affiche :

```
--> vlist
GetError SDL2.dll
```

Passage des paramètres vers les liaisons logicielles

Dans l'exemple précédent, le mot **GetError** ne nécessite pas de paramètre. Voyons le cas d'un mot nécessitant plusieurs paramètres :

```
\ Set the color used for drawing operations (Rect, Line and Clear)
z" SDL_SetRenderDrawColor" 5 SDL2.dll SetRenderDrawColor
```

Ici, en rouge, la valeur 5 qui indique au ticket de liaison logicielle qu'il faudra prendre en compte cinq paramètres pour le mot **SetRenderDrawColor**.

Ici un mot qui ne nécessite qu'un seul paramètre :

```
\ Destroy the rendering context for a window and free associated textures
z" SDL_DestroyRenderer"      1 SDL2.dll DestroyRenderer
```

Il n'y a pas d'indication à donner sur la nature et le nombre de paramètres récupérés par Forth après exécution des mots définis par un ticket de liaison logicielle.

Comprendre la documentation SDL2

La documentation de référence est accessible ici :

<https://wiki.libsdl.org/SDL2/CategoryAPI>

Pour eForth Windows, il n'est pas question de créer tous les mots associés à ces fonctions. D'une part, certaines fonctions comme **SDL_GetAndroidSDKVersion** ne concernent pas l'environnement Windows. D'autre part, le but de cette documentation est de vous donner les clés pour rajouter des mots si vous voulez utiliser une fonction dont la liaison logicielle n'a pas été définie dans **SDL2**.

Voyons un cas précis, la fonction **SDL_SetTextureColorMod**, donc la documentation est accessible ici :

https://wiki.libsdl.org/SDL2/SDL_SetTextureColorMod

La partie qui nous intéresse est ici :

SDL_Texture *	texture	the texture to update.
Uint8	r	the red color value multiplied into copy operations.
Uint8	g	the green color value multiplied into copy operations.
Uint8	b	the blue color value multiplied into copy operations.

Figure 9: paramètres de **SDL_SetTextureColorMod**

On voit ici qu'il y a quatre paramètres : **texture**, **r**, **g** et **b**. la seule chose à retenir c'est ce nombre quatre pour définir notre mot **SetTextureColorMod** :

```
z" SDL_SetTextureColorMod"      4 SDL2.dll SetTextureColorMod
```

Et c'est tout !

Lors de son exécution, le mot **SetTextureColorMod** renvoie 0 ou une valeur négative en cas d'erreur. Cette donnée sera à traiter lors de l'exécution du programme.

La documentation de chaque mot du vocabulaire **SDL2** est disponible ici :

<https://eforth.arduino-forth.com/index/glossaire-windows/>

Utilisez la recherche textuelle pour retrouver la documentation complète d'un mot du vocabulaire forth ou SDL2. Ici la documentation détaillée de **GetWindowSizeInPixels** :

<https://eforth.arduino-forth.com/help/explain-eforth-windows/id/1141>

Initialisation de l'environnement SDL2

L'unique paramètre du mot **Init** dans le vocabulaire SDL2 permet de définir les composants que la bibliothèque doit initialiser. Ceux-ci peuvent être :

- **SDL_INIT_TIMER** Initialise le sous-système des chronomètres
- **SDL_INIT_AUDIO** Initialise le sous-système pour l'audio
- **SDL_INIT_VIDEO** Initialise le sous-système pour le rendu
- **SDL_INIT_JOYSTICK** Initialise le sous-système pour les joysticks
- **SDL_INIT_HAPTIC** Initialise le sous-système pour le retour de force
- **SDL_INIT_GAMECONTROLLER** Initialise le sous-système pour les contrôleurs de jeux
- **SDL_INIT_EVENTS** Initialise le sous-système pour les événements
- **SDL_INIT EVERYTHING** Initialise tous les sous-systèmes nommés ci-dessus.

Par exemple, pour charger les systèmes audio et vidéo, il nous faudra utiliser cette ligne :

```
SDL2 \ seledc SDL2 vocabulary
SDL_INIT_VIDEO SDL_INIT_AUDIO or Init
```

Le mot **Init** peut retourner deux valeurs :

- 0 si l'initialisation s'est bien passée;
- une valeur négative si l'initialisation n'a pas pu se faire correctement.

Avec cette option, la librairie SDL2 ne mettra pas en place de fonction de nettoyage lors de la réception de signaux fatals.

Dans le code mis en avant ci-dessus, la valeur de retour de la fonction est vérifiée afin de quitter le programme lors d'un cas d'erreur. Lorsqu'une erreur se produit, la valeur est différente de 0. Afin d'informer l'utilisateur (et, bien souvent le programmeur), un message clair est affiché sur la sortie standard d'erreur. Le mot `GetError` permet de récupérer un descriptif de l'erreur rencontrée par la bibliothèque SDL2 :

```
\ Initialize SDL with error management
: SDL.init ( n -- )
  z" Could not Initialize environnement " SetError drop
  Init 0= if
    -1 SDL.error
  then
  ;
SDL_INIT_VIDEO SDL_INIT_AUDIO or SDL.Init
```

Une fois que la bibliothèque est initialisée, vous pouvez l'utiliser et donc, ouvrir une fenêtre.

Quitter l'environnement SDL2

À la fin du programme, il est nécessaire de faire l'opération inverse à l'initialisation avec le mot **Quit**. Il faudra aussi le faire si vous rencontrez une erreur lors de l'exécution du programme.

Toutes les manipulations doivent être effectuées entre **Init**, point de départ de notre programme, et le mot **Quit** qui en est la fin.

Le mot **Quit** ne prend pas de paramètre et ne retourne pas de valeur, il suffit de l'appeler de la sorte :

```
SDL2
SDL_INIT_VIDEO SDL_INIT_AUDIO or SDL.Init
\ ...here the main program...
Quit
```

Note : nos essais de Quit laissent un paramètre sur la pile.... Aucune explication !

Gestion des erreurs

Le mot **Init** comme beaucoup d'autres mots du vocabulaire **SDL2**, que nous verrons au cours de ce livre, peut échouer. Dans ce cas, ces mots renvoient une valeur d'erreur, une valeur négative pour **Init**.

En cas d'erreur, il ne faut pas poursuivre le programme en cas d'erreur. Dans le vocabulaire **SDL2**, on a le mot **GetError**.

GetError renvoie un message contenant des informations sur l'erreur spécifique survenue, ou une chaîne vide si aucun message d'erreur n'a été défini depuis le dernier appel à **ClearError**. Le message n'est applicable que lorsqu'un mot du vocabulaire **SDL2** a signalé une erreur. Vous devez vérifier les valeurs de retour des mots **SDL2** pour déterminer quand appeler correctement **GetError**.

```
\ send error message and abort if fl is not null
: SDL.error ( fl -- )
  if
    exit
  then
    SDL.Quit
    GetError z>s type
    abort
  ;
```

Il est possible que plusieurs erreurs se produisent avant d'appeler **GetError**. Seule la dernière erreur est renvoyée.

Le message n'est applicable que lorsqu'un mot **SDL2** a signalé une erreur. Vous devez vérifier les valeurs de retour des appels de mots **SDL2** pour déterminer quand appeler correctement **GetError**. Vous ne devez pas utiliser les résultats de **GetError** pour décider si une erreur s'est produite ! Parfois, **SDL2** définit une chaîne d'erreur même en cas de succès.

SDL2 n'efface pas la chaîne d'erreur pour les appels d'API réussis. Vous devez vérifier les valeurs de retour pour les cas d'échec avant de pouvoir supposer que la chaîne d'erreur s'applique.

La chaîne renvoyée est allouée en interne et ne doit pas être libérée par l'application.

Ici, on a défini **SDL.error**. Si le paramètre est nul, sont exécution s'achève là. Si le paramètre est non nul, la définition exécute **GetError** en affichant le message d'erreur, puis exécute **abort**.

L'exécution du mot **SDL.error** exécute également **SDL.Quit** qui libère les ressources SDL.

Les structures dans SDL2

La librairie SDL2 exploite un certain nombre de structures. Ce sont des données structurées définies dans un espace mémoire unique.

Pour définir une structure, on utilise le vocabulaire **structures** et le mot **struct** :

```
struct SDL_Rect
    i32 field ->Rect-x
    i32 field ->Rect-y
    i32 field ->Rect-w
    i32 field ->Rect-h
```

Ici, on définit la structure **SDL_Rect** et quatre accesseurs **->Rect-x**, **->Rect-y**, **->Rect-w** et **->Rect-h**.

Ici, chaque champ de cette structure est définie sur 32 bits par la séquence **i32 fields**.

Pour définir la taille d'un champ dans une structure, on peut utiliser :

- **i8** pour un champ 8 bits, soit un octet ;
- **i16** pour un champ 16 bits, soit deux octets ;
- **i32** pour un champ 32 bits, soit quatre octets ;
- **i64** pour un champ 64 bits, soit huit octets.

Utilisation d'une structure

Avec eForth Windows, une structure permet :

- de réserver un espace mémoire dont la taille est celle définie par la structure ;
- d'écrire ou lire dans une structure.

Pour définir un rectangle sous forme de constante :

```
create MY_RECT
    SDL_Rect allot
```

Ce qui réserve un espace mémoire défini par la taille de la structure **SDL_Rect**. Pour enregistrer les dimensions dans cet espace mémoire, on peut utiliser nos accesseurs :

```
: rect!    { x y w h addr -- }
    x addr ->Rect-x L!
    y addr ->Rect-y L!
    w addr ->Rect-w L!
    h addr ->Rect-h L!
;
```

```
10 20 350 40 MY_RECT rect!
```

Lire et écrire dans une structure

Il faut prêter une attention particulière à la taille des données auxquelles on accède dans une structure. Pour cela, on utilisera :

- **C!** et **C@** pour écrire et lire dans un champ d'un octet ;
- **W!** et **UW@** pour écrire et lire dans un champ de deux octets ;
- **L!** et **UL@** pour écrire et lire dans un champ de quatre octets ;
- **!** et **@** pour écrire et lire dans un champ de huit octet ;

Comme on le voit dans le mot `rect!` Précédemment défini, il est souhaitable d'écrire des définitions adaptées à la lecture ou écriture dans les champs auxquels on souhaite accéder :

```
create RECTS
  10 L,   10 L,  200 L,  120 L,
  22 L,   22 L,  205 L,  125 L,
  34 L,   34 L,  210 L,  130 L,
  46 L,   46 L,  215 L,  135 L,
: ->SDL_Rect { item addr0 -- addr1 }
  SDL_Rect item * addr0 +
;
: rect@ { item addr0 - x y w h }
  ->SDL_Rect to rectN
  rectN ->Rect-x UL@
  rectN ->Rect-y UL@
  rectN ->Rect-w UL@
  rectN ->Rect-h UL@
;
3 rect@      \ empile: 34 34 210 130
```

Les structures SDL sont isolées dans le vocabulaire `SDL_structures`. Avant toute utilisation de ces structures, il faut définir l'ordre de recherche dans ce vocabulaire :

```
forth definitions
SDL2 also
SDL_structures
```

Le fichier **SDL2_STRUCTURES.fs** ne contient que les définitions des structures utilisées dans les autres fichiers **SDL2_*.fs**.

Créer une fenêtre avec SDL2

La création d'une fenêtre est l'étape fondamentale pour démarrer un projet avec SDL2. Elle sert de canevas à votre jeu ou application.

Initialiser SDL

La première opération consiste à initialiser l'environnement SDL. Normalement, on doit utiliser le mot **Init** défini dans le vocabulaire **SDL2**. Ce mot **Init** est intégré à une définition plus élaborée **SDL.Init** qui va gérer une éventuelle erreur d'initialisation :

```
\ initialize SDL environnement
: initEnvironnement
  SDL_INIT_VIDEO SDL.Init
;
```

Ici, **SDL_INIT_VIDEO** indique que nous voulons initialiser le sous-système vidéo de la SDL.

Créer une fenêtre

Pour ce test, la première opération consiste à définir nos tests dans le vocabulaire **forth**, mais en indiquant un ordre d'accès au vocabulaire **SDL2** :

```
forth definitions
SDL2
```

A tout moment, pour connaître l'ordre d'accès, on tape **order** :

```
order    \ affiche:  SDL2 >> FORTH
```

Ceci signifie que l'interpréteur Forth va d'abord chercher les mots dans le vocabulaire **SDL2**.

On définit ensuite quatre constantes indiquant la taille et la position initiale de la fenêtre :

```
\ define size and position for SDL window
800 constant SCREEN_WIDTH
400 constant SCREEN_HEIGHT
200 constant X0_SCREEN_POSITION
50 constant Y0_SCREEN_POSITION
```

On définit maintenant deux valeurs qui serviront de tickets pour accéder à la fenêtre et au rendu dans cette fenêtre :

```
0 value WIN0
0 value REN0
```

On utilise maintenant **CreateWindow** et **CreateRenderer** avec les paramètres attendus :

```

: initNewWindow ( -- )
  \ create new window
  z" My first window with SDL2"
  X0_SCREEN_POSITION Y0_SCREEN_POSITION
  SCREEN_WIDTH SCREEN_HEIGHT
  SDL_WINDOW_SHOWN
  SDL.CreateWindow to WIN0
  \ create a new renderer
  WIN0 -1 0 CreateRenderer to REN0
;

```

Paramètres attendus par **CreateWindow** :

- **zstr** est une chaîne terminée par le code ASCII 0 (pas le chiffre 0!!!) ;
- **x** et **y** position de départ de la fenêtre à ouvrir. Ici, ce sont le contenu des constantes **X0_SCREEN_POSITION Y0_SCREEN_POSITION** qui est utilisé pour définir cette position ;
- **w** et **h** déterminent la largeur et la hauteur de la fenêtre. Ici, ce sont le contenu des constantes **SCREEN_WIDTH SCREEN_HEIGHT** qui est utilisé pour définir les dimensions de cette fenêtre ;
- **fl** est un drapeau indiquant la manière dont la fenêtre va s'ouvrir. Ici, c'est la constante **SDL_WINDOW_SHOWN** qui sert de drapeau, indiquant que la fenêtre doit s'ouvrir immédiatement.

L'exécution de **CreateWindow** renvoie un ticket qui est stocké dans la valeur **WIN0**.

Ce ticket est ensuite utilisé par **CreateRenderer**. Paramètres attendus :

- **window** est un ticket enregistré après exécution de **CreateWindow**. Correspond à la fenêtre où le rendu est affiché ;
- **index** du pilote de rendu à initialiser, ou -1 pour initialiser le premier prenant en charge les indicateurs demandés
- **flag** 0, ou un ou plusieurs flags combinés par OU.

L'exécution de **CreateRenderer** restitue un ticket qui est stocké dans la valeur **REN0**.

Maintenant, il faut prévoir la manière de libérer les ressources, c'est à dire comment fermer le rendu et la fenêtre. C'est le rôle des mots **DestroyRenderer**, **DestroyWindow** et **Quit** :

```

\ free ressources, end renderer and window
: freeRessources ( -- )
  REN0 DestroyRenderer drop
  WIN0 DestroyWindow drop
  Quit

```

```
;
```

Le mot **DestroyRenderer** a besoin du ticket correspondant au rendu précédemment ouvert, ici celui stocké dans **REN0**.

Le mot **DestroyWindow** a besoin du ticket correspondant au rendu précédemment ouvert, ici celui stocké dans **WIN0**.

Entre l'initialisation et la fermeture de la fenêtre, on définit les instructions qui vont colorer la fenêtre :

```
: draw ( -- )
  REN0 255 0 0 255 SetRenderDrawColor drop
  REN0 RenderClear drop
  REN0 RenderPresent drop
;
```

Dans le mot draw :

- **SetRenderDrawColor** définit la couleur utilisée pour les opérations de dessin (Rect, Ligne et Effacer) ;
- **RenderClear** efface la cible de rendu actuelle avec la couleur de dessin ;
- **RenderPresent** met à jour l'écran avec tout rendu effectué depuis l'appel précédent.

Chacun de ces mots restitue un code d'erreur dont nous ne tiendrons pas compte pour le moment.

Enfin, nous définissons le mot **main** qui va enchaîner les opérations d'initialisation, de tracé et de fermeture de la fenêtre :

```
: main ( -- )
  initEnvironnement
  initNewWindow
  draw
  begin
    key drop
  -1 until
  freeRessources
;
```

Voici le résultat de l'exécution de **main** :

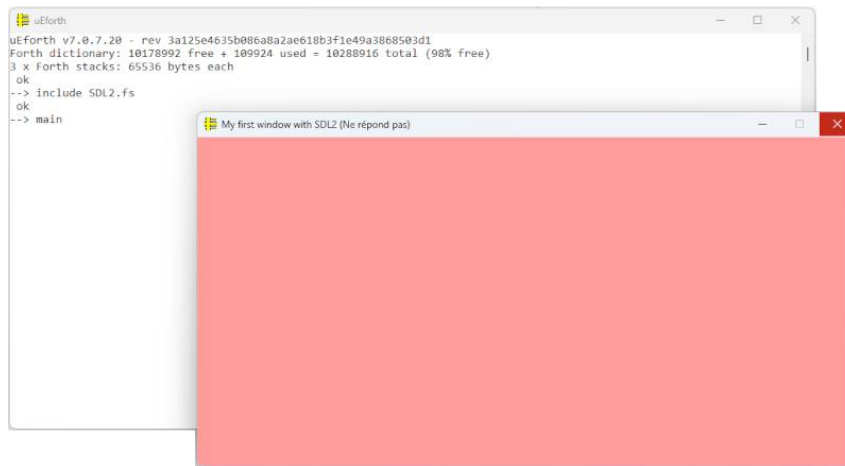


Figure 10: ouverture d'une fenêtre

Dans la définition de **main**, il y a une boucle **begin until**. Cette boucle doit héberger un gestionnaire d'événements. Pour le moment, on a simplement mis une détection de touche clavier. Un appui sur une touche ferme la fenêtre.

Redimensionner, réduire et restaurer une fenêtre

Pour réduire la fenêtre vers la barre des tâches, on utilise le mot **MinimizeWindow** :

Pour agrandir la fenêtre, nous devons utiliser le mot **MaximizeWindow** :

Pour restaurer une fenêtre, quand elle est accessible seulement sous forme d'icone, il faut utiliser le mot **RestoreWindow**.

```
: draw ( -- )
  REN0 255 255 255 255 SetRenderDrawColor drop
  REN0 RenderClear drop
  REN0 RenderPresent drop
  2000 Delay drop
  WIN0 MinimizeWindow drop
  2000 Delay drop
  WIN0 RestoreWindow drop
;
```

Visibilité des fenêtres

Une fenêtre peut avoir trois états :

- cachée en utilisant le mot **HideWindow** ;
- visible avec le mot **ShowWindow** ;
- en avant de toutes les autres fenêtres, avec **RaiseWindow**.

```
1 void SDL_RaiseWindow(SDL_Window* window)
```

Nous pouvons le voir, ces fonctions sont très simples à utiliser. Elles prennent en paramètre la

fenêtre sur laquelle on veut agir et ne renvoient pas de valeur.

2.3.2.3. Le plein écran

Nous pouvons placer notre fenêtre en plein écran durant sa création, mais nous pouvons également le faire plus tard. Par exemple, dans un jeu, nous pourrions proposer à l'utilisateur

de mettre la fenêtre en plein écran ou non. La fonction `SDL_SetWindowFullscreen` permet de le faire. Son prototype:

```
1 int SDL_SetWindowFullscreen(SDL_Window* window,  
2 Uint32 flags)
```

Elle prend en paramètre la fenêtre qu'on veut gérer et un drapeau. Ce drapeau peut-être:

— `SDL_WINDOW_FULLSCREEN` pour placer la fenêtre en plein écran;

Afficher une image BMP avec SDL2

SDL2, dans sa configuration de base, ne supporte nativement que le format BMP. Cela signifie que vous pouvez charger et afficher directement des images BMP sans avoir besoin d'une bibliothèque tierce comme **SDL_image**.

Il est entendu que l'environnement permettant d'afficher une image est celui-ci :

```
: main ( -- )
  initEnvironnement
  initNewWindow
  draw
  begin
    key drop
  -1 until
  freeRessources
;
```

Le principe est toujours le même :

- l'environnement graphique SDL2 est initialisé ;
- une nouvelle fenêtre est ouverte ;
- les tracés graphiques sont exécutés
- on attend l'appui sur une touche du clavier
- les ressources SDL sont libérées

Ici, nous ne gérons pas encore les événements. Le contenu de **main** est volontairement simpliste, car maintenant, on s'attachera surtout à expliquer le contenu de **draw** au travers de divers exemples.

Chargement d'une image BMP

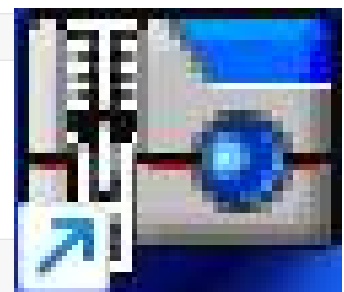
Ici, le mot **ZFILE** va pointer vers le chemin de notre image BMP :

```
z" SDL2/logo01.bmp" constant ZFILE
```

L'image BMP choisie est celle-ci :

C'est une toute petite image, de 61 x 54 pixels. Pour cette image, on réserve la surface et la texture de cette image :

```
0 value IMAGE01    \ is surface
0 value TEXTURE01
```



La surface **IMAGE01** pointe vers notre image BMP après ceci :

```
ZFILE SDL.load-image to IMAGE01
```

Le mot SDL.load-image encapsule LoadBMP :

```
: LoadBMP ( zstr -- 0|surface )
  z" rb" RWFromFile 1 LoadBMP_RW
;

: SDL.load-image ( zstr -- surface )
  LoadBMP ?dup 0= if
    drop -1 SDL.error
  then
;
;
```

Ces deux définitions sont pré-implantées dans le fichier **SDL2_images.fs**.

La dernière opération de chargement consiste à affecter le pointeur **TEXTURE01** :

```
REN0 IMAGE01 CreateTextureFromSurface to TEXTURE01
```

Résumé des lignes de code de chargement d'une image au format BMP :

```
: draw ( -- )
  ZFILE SDL.load-image to IMAGE01
  REN0 IMAGE01 CreateTextureFromSurface to TEXTURE01
;
;
```

Affichage de l'image

L'affichage de notre image au format BMP peut maintenant s'effectuer :

```
REN0 TEXTURE01 NULL NULL RenderCopy drop
```

Voici le contenu complet du mot **draw** pour tester notre premier affichage d'image :

```
: draw ( -- )
  REN0 255 255 255 255 SetRenderDrawColor drop
  WIN0 z" Display BMP image with SDL2" SetWindowTitle drop
  REN0 RenderClear drop
  \ load an image
  ZFILE SDL.load-image to IMAGE01
  REN0 IMAGE01 CreateTextureFromSurface to TEXTURE01
  REN0 TEXTURE01 NULL NULL RenderCopy drop
  REN0 RenderPresent drop
;
;
```

Résultat :

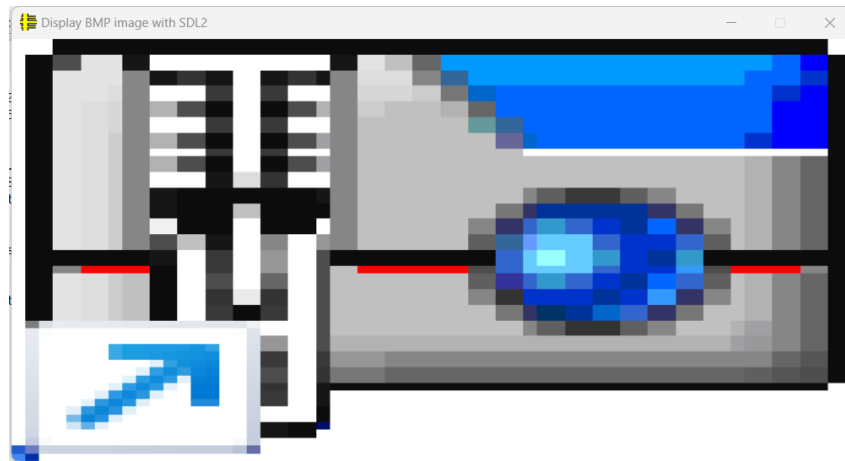


Figure 11: premier test d'affichage

Ici, l'image occupe la totalité de la fenêtre. Voyons comment contrôler la taille de l'affichage de notre image au format BMP.

Contrôle des dimensions d'affichage d'une image

Avant d'aller plus avant, il faut revenir sur les paramètres de **RenderCopy** :

Paramètres :

- **renderer** le contexte de rendu ;
- **texture** la texture source ;
- **srcrect** la structure SDL_Rect source ou NULL pour la texture entière ;
- **dstrect** la structure SDL_Rect de destination ou NULL pour la cible de rendu entière ; la texture sera étirée pour remplir le rectangle donné.

Rappelons qu'une structure est un modèle de données. Pour créer un espace de données conforme à SDL_Rect :

```
create RECT01
  20 L,  20 L,  61 L,  54 L,
```

Rappelons que eForth Windows gère des données sur 64 bits. Dans la structure RECT01 les paramètres sont stockés au format 32 bits qui est le format attendu par **RenderCopy**.

Ici, les données stockées dans RECT01 indiquent la position **x** et **y** de l'affichage de l'image (20 20), **w** et **h** étant les dimensions de l'affichage (61 54) :

```
REN0 TEXTURE01 NULL RECT01 RenderCopy drop
```

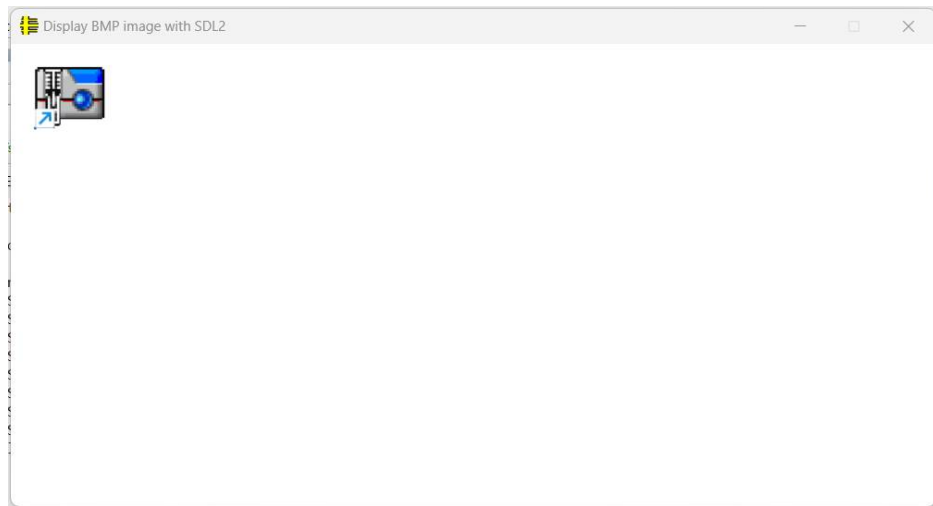



Figure 12: second exemple d'affichage d'une image

Contenu complet du code affichant cette image. En rouge la ligne qui provoque l'affichage de l'image :

```
z" SDL2/logo01.bmp" constant ZFILE

0 value IMAGE01      \ is surface
0 value TEXTURE01

create RECT01
    20 L,  20 L,  61 L,  54 L,

: draw ( -- )
    REN0 255 255 255 255 SetRenderDrawColor drop
    WIN0 z" Display BMP image with SDL2" SetWindowTitle drop
    REN0 RenderClear drop
    \ load an image
    ZFILE SDL.load-image to IMAGE01
    REN0 IMAGE01 CreateTextureFromSurface to TEXTURE01
    REN0 TEXTURE01 NULL RECT01 RenderCopy drop
    REN0 RenderPresent drop
;
```

En modifiant les dimensions **w** et **h** dans la structure **RECT01**, on peut modifier la taille d'affichage de notre image.

Ressources

- **eForth for Windows**
projet créé et maintenu par Brad NELSON
<https://eforth.appspot.com/windows.html>
- **Analyseur de code FORTH**
transforme un code FORTH non documenté en une version avec coloration syntaxique et liens hypertexte vers les mots connus.
<https://analyzer.arduino-forth.com/>

GitHub

- **SDL2 eForth windows project**
codes et documentations du projet SDL2
<https://github.com/MPETREMAN11/SDL2-eForth-windows>

SDL

- **Simple DirectMedia Layer**
plate-forme de développement de la librairie SDL donnant accès aux ressources audi, clavier, souris et matériel graphique via OpenGL et Direct3D.
<https://www.libsdl.org/>

Index

CreateRenderer.....	24	fields.....	22	Quit.....	19, 25
CreateTextureFromSurface...	30	GetError.....	20	RenderCopy.....	30
CreateWindow.....	24	GIT.....	11	struct.....	22
DestroyRenderer.....	25	Init.....	18, 24	structures.....	22
DestroyWindow.....	25	LoadBMP.....	30		
dll.....	15	Netbeans.....	10		