Cross-Lib

https://github.com/Fabrizio-Caruso/CROSS-LIB

Fabrizio Caruso

fabrizio_caruso@hotmail.com

La première version de cet article est apparue dans «Programmez ! Hors Série #6»
https://www.programmez.com/magazine-papier/programmez-hors-serie-6

BIOGRAPHIE

Fabrizio Caruso, sicilien mais niçois d'adoption depuis 10 ans, ingénieur en informatique, est passionné par la programmation pour d'anciens ordinateurs et consoles et par l'histoire de l'informatique, sur lesquelles il a travaillé dans le cadre de plusieurs projets. Ses travaux sont concentrés sur la programmation en C pour les machines 8-bit des années 80 ; sur la programmation en BASIC des années 80 et son histoire. Il participe régulièrement à des compétitions de programmation pour les ordinateurs 8-bit en C ou BASIC avec plusieurs jeux dans la compétition internationale BASIC 10-liner (classé deuxième en 2021).

INTRODUCTION

Cross-Lib est un projet dont la finalité est le retro-développement (en anglais *retro-coding*). Celui-ci est lié au *retro-computing* et au *retro-gaming*.
Tous ces termes sont liés aux anciens ordinateurs et consoles mais il faut d'abord clarifier leur définition.
Qu'est-ce que le *retro-gaming*, le *retro-computing* et le *retro-développement* ?
- Le *retro-gaming* est probablement le plus connu des trois. Il s'agit de l'amour pour les jeux sur d'anciens ordinateurs et consoles.
- Le *retro-computing* est le hobby plus générique qui consiste à utiliser des ordinateurs et consoles anciens pas seulement pour y jouer mais aussi pour d'autres activités telles que par exemple le traitement de texte.
- Le retro-développement (ou en anglais *retro-coding*) est le développement de logiciels pour d'anciennes machines, c'est-à-dire ordinateurs, consoles, bornes d'arcades, calculatrices scientifiques, etc.

Cross-Lib est un outil de *retro-coding* libre et open source pour plus de 200 machines à 8-bit des années 80 (ordinateurs, consoles, calculatrices, bornes d'arcade, etc.) comme le *Commodore 64, le Commodore VIC 20* et les autres *Commodores*, les *Thomsons Mo5 e To7*, l'*Apple][*, l'*Atari 800*, l'*Amstrad CPC*, le *Sinclair ZX 81 et ZX Spectrum*, les *MSX*, l'*Oric 1* et *Atmos*, le *BBC Micro*, le *TRS-80 CoCo*, le *Dragon 32*, la *GameBoy*, la *Sega Master System*, la *GameGear*, la *Nintendo NES*, les calculatrices Texas Instruments comme la *TI 83*, etc. avec pour principales architectures Zilog 80, MOS 6502 et Motorola 6809. Une liste partielle avec plus de 160 machines supportées se trouve à la page :https://github.com/Fabrizio-Caruso/CROSS-LIB/blob/master/docs/STATUS.md. Cross-Lib supporte des machines plutôt rares comme le *Philips VG-5000*, le *Jupiter Ace*, l'*Alice Matra*, etc. et des machines très exotiques comme par exemple plusieurs ordinateurs de l'Europe de l'Est comme le yougoslave *Galaksija,* le hongrois *Homlab-2*, plusieurs ordinateurs de la série *Robotron* comme le *Robotron KC 85*, etc. Cela pourra sûrement intéresser aussi les passionnés de *retro-computing* et de *retro-gaming* parce que pouvoir coder sur les retro-ordinateurs et consoles permet de produire des outils et des jeux pour ces machines.

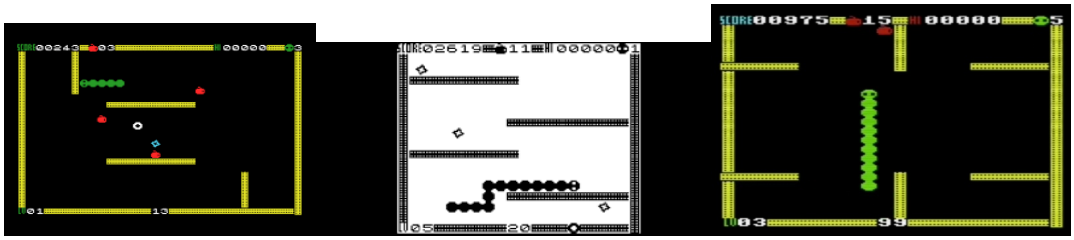
MOTIVATION : Pourquoi fait-on du *retro-coding* ?
Il constitue une source de plaisir, une façon de s'entraîner à coder pour des machines avec des ressources très limitées [c8b] comme on ferait pour des microcontrôleurs [effc], de la nostalgie et un moyen de faire des « découvertes historiques ou archéologiques » (comme par exemple le travail académique [east] fait par Caruso *et al.*).

DISCLAIMER : Il faut préciser que Cross-Lib dans son état actuel n'est pas complet mais est déjà utilisable. Ici, je présente Cross-Lib dans son état de développement actuel. Certains scripts de Cross-Lib peuvent effacer des fichiers. L'auteur décline toute responsabilité pour la perte de fichiers.

1. RÉSULTATS POSSIBLES

Ici figurent des captures d'écran des cinq jeux que j'ai écrits avec Cross-Lib (pour plus d'informations voir [xlib], [app], [ceo]). Les images montrent les résultats sur des machines différentes où, par exemple, le nombre de couleurs disponibles diffère. Les graphismes possibles avec Cross-Lib seront ceux que l'on pouvait trouver dans les ordinateurs du début des années 80, comme par exemple sur le *Sinclair ZX Spectrum*.

CROSS SNAKE (Sinclair ZX Spectrum, GameBoy, Commodore VIC 20)



CROSS HORDE (Sega Master System, Amstrad CPC, Atari 800)

CROSS BOMBER (Atari 800), CROSS CHASE (Atari 800), CROSS SHOOT (Sega Master System)

2. RETRO-CODING SANS CROSS-LIB

Aujourd'hui, il y a beaucoup d'outils qui permettent de coder pour les retro-machines très facilement sur un ordinateur moderne. Dans ce cas-là on parle de cross-développement.
Historiquement, les machines à 8-bit étaient programmées directement sur la machine surtout en Assembleur ou, pour des programmes plus simples, en BASIC interprété.
Aujourd'hui le cross-développement simplifie beaucoup la tâche du développeur (éditeurs modernes, debugging simplifié, compilation efficace et rapide, etc.). La programmation pour ces machines aujourd'hui se fait encore très souvent en Assembleur pour pouvoir exploiter au maximum les ressources hardware de chaque machine, mais on ne disposera pas de code portable.
En revanche, grâce à des cross-compilateurs et bibliothèques très performantes, il existe des alternatives comme le

langage C, qui est un bon compromis entre performance et rapidité d'écriture du code et permet d'avoir du code au moins partiellement portable.

La plupart des retro-développeurs veulent exploiter une machine particulière au maximum. Dans le passé, la portabilité n'a pas été une priorité pour le retro-coding parce qu'elle est difficile à mettre en pratique sur des machines si différentes et parce qu'elle impose des limitations sur la possibilité d'exploiter toutes les caractéristiques de chaque machine. L'exemple le plus évident de retro-développement qui exploite les limites (et en effet dépasse les limites hardware connues) sont les productions de la *demo-scene*. On voit des jeux très récents pour certains ordinateurs 8-bit (comme les Commodores, les Sinclairs et les Ataris 8-bit) qui ont des graphismes que l'on a crus impossibles sur ces machines jusqu'il y a 20 ans, bien que rien n'ait été changé dans l'hardware. Par exemple il y a des jeux avec sprites multi-couleurs pour le Sinclair ZX Spectrum qui n'a ni sprites hardware ni modes graphiques multi-couleurs.

Récemment on a vu apparaître des outils qui permettent la production de code partiellement portable et des outils pour plusieurs machines différentes. Un exemple est le dev-kit *Z88DK* avec ses bibliothèques pour le développement en C sur presque une centaine d'ordinateurs et consoles basés sur les architectures Zilog 80 et Intel 8080. Deux autres exemples sont le dev-kit *CC65* et le dev-kit *8-Bit-Unity* [un8] (basé sur *CC65*) pour le développement en C sur certaines architectures basées sur le MOS 6502. Grâce à ces outils on a commencé à voir des jeux 8-bit multi-plateforme qui n'exploitent pas forcement tous les détails de chaque machine mais qui sont très soignés comme par exemple *8-bit Slicks* écrit avec *8-Bit-Unity* et *CC65*.

Le problème de la portabilité du code C est lié au fait que les machines sont très différentes et que le langage C n'a pas de bibliothèques standard pour les graphismes, les sons, l'input de manettes, etc.
Une solution partielle est donnée par certains dev-kits comme *CC65*, *Z88DK*, *LCC1802* (pour l'architecture RCA 1802), *CMOC* (pour l'architecture Motorola 6809), etc. En utilisant l'un de ces dev-kits, on peut utiliser des bibliothèques pour l'input et l'output qui sont compatibles sur plusieurs machines de la même architecture.
En revanche deux problèmes se posent :
1. Chaque dev-kit ne couvre qu'un sous-ensemble des architectures 8-bit.
2. Les bibliothèques fournies avec les dev-kit ne couvrent pas toujours tous les graphismes, le son ou l'input sur toutes les machines supportées.

3. AVANTAGES ET LIMITES DE CROSS-LIB

Avec Cross-Lib on veut faire encore plus que ce qui est possible avec les autres outils :
1. couvrir toutes les architectures vintage 8-bit, y compris les plus exotiques ;
2. permettre l'écriture de code 100% *WORA* ("*write once, run anywhere*") ;
3. permettre la création de ressources graphiques *WORA*, c'est-à-dire que l'on pourra définir les éléments graphiques (les « tiles ») une seule fois pour (presque) toutes les machines ;
4. fournir une *tool-chain* simplifiée pour créer, compiler et tester les jeux.

J'ai choisi le nom *Cross-Lib* comme contraction de "*cross-compilable library*" (bibliothèque logicielle cross-compilable). Donc Cross-Lib est (surtout) une bibliothèque logicielle pour écrire du code en C qui peut être compilé par plusieurs cross-compilateurs pour presque toutes les consoles et ordinateurs 8-bit des années 80. On pourrait dire que le but principal de Cross-Lib est de fournir une abstraction qui puisse couvrir toutes les machines 8-bit des années 80. Il permet l'écriture de jeux universels pour plus de 200 machines vintage différentes (surtout 8-bit mais pas seulement).

En revanche Cross-Lib n'est pas un compilateur. Cross-Lib a besoin d'un, voire plusieurs compilateurs pour pouvoir produire des fichiers binaires (exécutables, images des disquettes, images des cassettes, images des cartouches, etc.) pour les retro-ordinateurs et retro-consoles.

Cross-Lib est aussi un ensemble d'outils (scripts) qui permettent la compilation universelle et l'adaptation automatique des ressources graphiques (« graphics assets ») à toutes les machines.

Le fait que Cross-Lib fournisse une couche d'abstraction pour (presque) toutes les machines 8-bit, implique beaucoup de compromis par rapport à ce que l'on pourrait faire avec une machine spécifique si l'on ne voulait programmer que spécifiquement pour cette seule machine. Notamment Cross-Lib permet de programmer des graphismes limités parce qu'on veut pouvoir les afficher sur toutes les machines équipées de graphismes. On aura également des effets sonores mais très limités.

Il y a de nombreux dev-kits pour programmer pour plusieurs machines mais très peu permettent de le faire de façon *WORA* comme par exemple *Z88DK* et *8-Bit-Unity*. Il y en a d'autres qui ne sont pas *WORA* et qui, dans certains cas, n'utilisent pas forcement le langage C comme c'est le cas de *TRSE* [trse] (« *Turbo Rascal Syntax Error* »). Ici on fait

une comparaison entre des dev-kit *WORA* :

	Inclut compilateur(s)	**Ressources *WORA***	**APIs pour les *sprites***	**Nombre de machines**	**Architectures**
Cross-Lib	Non	Oui	Non	**~200**	Intel 8080, MOS 6502, Motorola 6809, RCA 1802 et autres
8-bit Unity	Non	Non	Oui	5	MOS 6502
Z88DK	Oui	Oui	partiellement	~100	Intel 8080, Zilog 80

Ce tableau montre les points forts et faibles de Cross-Lib : il est le seul dev-kit WORA qui permet de coder sur (presque) toutes les architectures et machines 8-bit. Par contre Cross-Lib ne propose pas d'APIs pour les *sprite* et en général pas d'APIs avancées. La raison est que Cross-Lib couvre toutes les machines, y compris celles qui n'ont pas de graphismes ou pas de *sprites* hardware ou pas de modes graphiques qui permettent d'implémenter des *sprites* software génériques. La seule manière d'implémenter des *sprites* avec Cross-Lib est l'utilisation de *pre-shifted tiles* dont on parlera dans les sections suivantes.

4. COMPILATEURS C ET LEURS LIMITES

Cross-Lib est censé être utilisé avec des compilateurs C mais lesquels ?

Un compilateur natif (par exemple GCC) peut être utilisé pour produire des fichiers exécutables natifs qui tournent par exemple sous Windows. Aujourd'hui le support pour la compilation native est limité au terminal. Donc pas de graphismes sur la machine native.

Actuellement Cross-lib supporte pleinement les suivants cross-compilateurs et dev-kits avec graphismes et effets sonores :
- **CC65** [cc65] pour la plupart des machines basés sur l'architecture MOS 6502 ;
- **Z88DK** [z88dk] pour les machines basés sur Zilog 80 et Intel 8080 ;
- **CMOC** [cmoc] pour les machines basés sur Motorola 6809 ;
- **LCC1802** [lcc1802] pour les machines basés sur le COSMAC RCA 1802 ;
- **GCC for TI** [ti] pour le Texas Instruments TI99/4A basé sur la CPU 16-bit TMS9900.

Cross-Lib supporte aussi d'autres compilateurs (pour l'instant sans graphismes et sans effets sonores) comme `ACK` [ack] for PC 8088/8086, CP/M Intel 8080, 386/68K/PPC/MIPS, PDP11, `XTC68` [xtc68] pour le Sinclair QL, `VBCC` [vbcc] pour le BBC Micro, BBC Master, l'Amiga et d'autres, `CC6303` [cc6303] pour le Motorola 6803, et plusieurs versions de `GCC` modifiés pour cibler des retro-ordinateurs (Atari ST [st], Olivetti M20 [m20]).

Pourquoi utilise-t-on un sous-ensemble du standard C89 ?
Tout simplement parce que la plupart des cross-compilateurs disponibles pour les architectures 8-bit n'implémentent qu'un sous-ensemble du C89 avec très peu de C99 (les commentaires `//`).
Donc on écrit en C (un sous-ensemble du ANSI C C89) avec les APIs de Cross-Lib pour les graphismes, les effets sonores, l'input (clavier/joystick/manette) et d'autres fonctions qui permettent l'écriture de code universel quand l'équivalent du standard C n'est pas défini de manière univoque.

En particulier il faut programmer en C89 sans
- "`float`" et "`double`",
- copies et passage par valeur d'objets de type "`struct`",
- le heap,
- fonctions récursives.

La contrainte sur les fonctions récursives réside dans son coût, surtout pour l'architecture MOS 6502. On pourrait quand même utiliser des fonctions récursives si cela était vraiment nécessaire.

5. INSTALLATION

Dans cette section on montrera comment installer Cross-Lib.

Cross-Lib nécessite :
1. un environnement POSIX (Linux, FreeBSD, Cygwin sous Windows, Windows Subsystem for Linux, etc.) ;
2. des outils de build (GNU `make`) et de développement (`python`, `java`) ;
3. au moins un compilateur ANSI C natif et/ou un des cross-compilateurs supportés ;
4. des émulateurs (optionnels mais très utiles).

Il n'y a pas une procédure standard pour installer tous les outils et les compilateurs parce que cela dépend de l'environnement spécifique choisi. Ici on donnera des exemples et des conseils génériques pour l'installation. Un exemple d'installation très détaillée se trouve dans l'annexe à la fin de cet article.
Pour tout simplifier, on a prévu pour le futur une version de Cross-Lib dans une image *Docker* avec tous les outils et compilateurs.

5.1 OUTILS

Il suffira d'installer GNU `make` et `python` (version 2.x ou 3.x).
Sous Linux il existe des commandes qui simplifient l'installation. Par exemple sur certaines distributions Linux on pourra utiliser `apt-get install make`.
Sous Cygwin/Windows, cela se fait à travers son installer en choisissant le package `make`.
Sous FreeBSD il faudra installer `gmake` parce que `make` sur les Unix de la famille BSD lance par défaut BSD `make`, qui n'est pas compatible avec Cross-Lib.

5.2 COMPILATEURS ANSI C

Il faut installer au moins un des compilateurs supportés. Cross-Lib ne supporte que les toutes dernières versions des cross-compilateurs.

Pour commencer, il faudrait installer un compilateur natif lequel nous permettra de compiler des versions natives, c'est-à-dire pour le terminal qui n'a pas des graphismes mais cela nous permettra de démarrer et voir des résultats. En plus un compilateur natif est utile pour tester et debugger le code. `GCC` est un compilateur natif qui est facile à installer. Par exemple sur Ubuntu et d'autres distributions Linux qui utilise `apt` il pourrait suffire :

```
sudo apt install build-essential
```

Installation de Cross-Compilateurs et Dev-kits 8-bit : La majorité des machines supportées est couverte par deux dev-kits:
- `Z88DK` (qui supporte environ une centaine de machines)
- `CC65` (qui supporte plus de 30 machines)

La procédure d'installation de CC65 dépend de l'environnement. En général on peut toujours compiler le code source.
Sous Windows on peut juste télécharger la *snapshot* de `CC65` (voir [cc65] pour le lien), le décompresser sur le disque dur de votre ordinateur et rajouter la location de `cc65-snapshot-win32/bin` à la variable `PATH`. Cela suffira pour utiliser CC65 sous Windows. Sur Linux, FreeBSD ou MacOS on peut, soit compiler le source, soit installer depuis un répertoire (si disponible). Par exemple sur Ubuntu on pourrait installer CC65 avec:

```
sudo apt-get install cc65
```

Si l'on utilise un package précompilé il faut vérifier que le répertoire contient une version récente.
Avec une version ancienne, on risque d'avoir de problèmes de compatibilité avec Cross-Lib.

Sous Cygwin/Windows on peut tout simplement installer CC65 sur Windows et le lancer depuis Cygwin.
Comme pour CC65, la procédure d'installation du dev-kit Z88DK dépend de l'environnement.
Une description des différentes procédures est disponible sur:

```
https://github.com/z88dk/z88dk/wiki/installation
```

Par exemples sous Linux on peut suivre:

`https://github.com/z88dk/z88dk/wiki/installation#linux--unix`

Sous Windows, la solution la plus simple est d'utiliser une des snapshots précompilées disponibles sur `http://nightly.z88dk.org/` e définir la variable d'environnement `ZCCCFG` comme décrit dans la procédure d'installation spécifique de l'environnement.

6. PROGRAMMER EN C AVEC CROSS-LIB

Coder avec CROSS-LIB implique trois choses :
1. programmer en C (un sous-ensemble du standard C89) un peu comme on le ferait si l'on codait pour un microcontrôleur en ne pas oubliant les limites des appareils 8-bit des années 80 ;
2. utiliser les APIs de Cross-Lib pour l'input/output et pour très peu d'autres choses ;
3. programmer des façon abstraite grâce aussi à certaines macros pour tous les machines supportées, y compris les machines qui ont des fonctionnalités réduites.

Le langage C est bien adapté pour des architectures à 16, 32 et 64-bit et beaucoup moins pour les 8-bit des années 80, qui sont très proches des microcontrôleurs 8-bit d'aujourd'hui (voir [c8b] et [effc] pour plus de détails).

Par exemple il faudra:
- éviter les fonctions récursives parce que certaines architectures n'ont pas de stack ou elles ont un stack hardware très peu profond ;
- se limiter aux types `unsigned` à 8 et 16 bit (`uint8_t` et `uint16_t`) parce que les opérations avec signe ne sont pas très efficientes sur les architectures vintage à 8-bit ;
- réduire le nombre des variables locales pour limiter l'utilisation du stack ;
- réduire le nombre des paramètres pour limiter l'utilisation du stack ;
- réduire le nombre d'unités de compilation (nombre de fichiers `.c` avec leurs *headers*) parce que de nombreux compilateurs n'optimisent pas le code qui est partagé entre plusieurs unités de compilation différentes (pas de « *link-time optimization* ») ;
- éviter, lorsque cela est possible, les produits, les divisions et le modulo en utilisant des opérateurs sur les bits (comme `&`, `<<`, `>>`, etc.) parce que le produit et la division sont des opérations très lourdes ;
- parfois optimiser la mémoire plutôt que la performance parce que certaines machines ont une quantité minuscule de mémoire vive (dans certains cas moins d'un kilooctet).

7. COMMENCER AVEC CROSS-LIB

Pour commencer, il faut comprendre la structure des dossiers de Cross-Lib. Tout d'abord il faudra se mettre dans le répertoire `src/` où se trouve tout le code C et les scripts Python.

Le code de Cross-Lib se trouve dans le répertoire `src/cross-lib/`.

On appellera *projet* un logiciel écrit avec Cross-lib. On a 3 types de projets : les jeux built-in, les exemples built-in et les nouveaux projets définis par l'utilisateur.

Le code C des projets se trouve dans :
- `src/games/` qui contient le code des jeux built-in ;
- `src/examples/` avec plusieurs exemples ;
- `src/projects/` qui contient le code des nouveaux projets définis par l'utilisateur et c'est le seul code que l'utilisateur est censé écrire et modifier.

Le script `xl` est un script Python qui simplifie la création de nouveaux projets et leur compilation, test et exécution.
Ce script doit être utilisé depuis le répertoire `src/` et peut effacer des fichiers donc il faudra l'utiliser avec précaution.

On montrera comment utiliser `xl` avec des exemples. Pour avoir les instructions complètes on peut lancer `xl help` depuis le répertoire `src/`.

Cross-Lib est fourni avec de nombreux exemples de code qui montrent comment l'utiliser. Tous les exemples simples se trouvent dans `src/examples/`. En plus on a le code des 5 jeux complets dans `src/games/`.

En particulier on peut commencer avec `helloworld` qui se trouve dans `src/examples/helloworld/`, dans lequel

on trouve le fichier `main.c` qui contient :

```
#include "cross_lib.h"

int main(void)
{
    _XL_INIT_GRAPHICS();

    _XL_CLEAR_SCREEN();

    _XL_SET_TEXT_COLOR(_XL_WHITE);

    _XL_PRINT_CENTERED("HELLO WORLD");

    while(1){};

    return EXIT_SUCCESS;
}
```

On remarque tout de suite que les fonctions de Cross-Lib commencent par `_XL_` :
1. `_XL_INIT_GRAPHICS()` qui est nécessaire pour initialiser le display ;
2. `_XL_CLEAR_SCREEN()` qui efface l'écran ;
3. `_XL_SET_TEXT_COLOR(_XL_WHITE)` qui définit la couleur du texte qui suivra ;
4. `_XL_PRINT_CENTERED("HELLO WORLD")` qui affiche la chaîne de caractères « `HELLO WORLD` » centrée au milieu de l'écran.

Pour compiler un projet on peut utiliser le script `xl` avec la syntaxe suivante :

```
xl <project> <target>
```

où `<project>` est le nom du project et `<target>` est le nom de la machine (par défaut : `ncurses`, c'est-à-dire la build compilée avec `GCC` pour le terminal natif avec `ncurses`).

Donc pour compiler `helloworld` avec `GCC` et `ncurses` on fera :
`xl helloworld`

qui produira un fichier exécutable natif dans le répertoire `build` qu'on pourra lancer avec

```
xl run helloworld
```

Pour cross-compiler `helloworld` il faudra spécifier un *target* différent. Par exemple

```
xl helloworld c64
```

que l'on pourra lancer avec un émulateur pour le Commodore 64 et si l'on a installé l'émulateur `x64` (émulateur C64 contenu dans `Vice`), on pourrait le lancer directement depuis le terminal avec

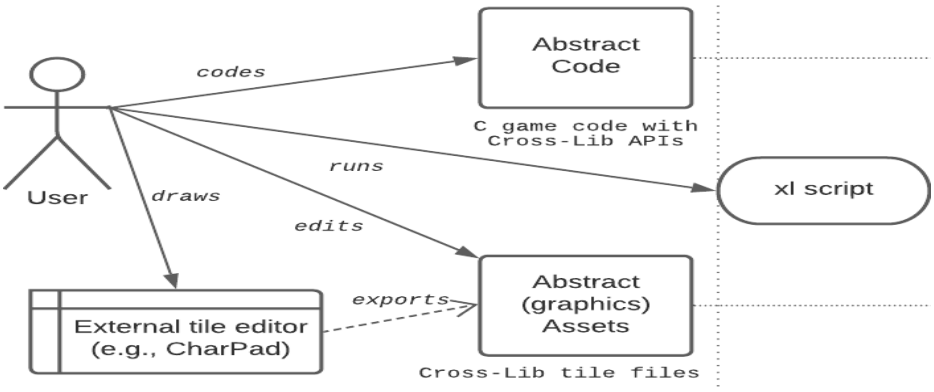
```
xl run helloworld c64
```

Dans la plupart des cas, Cross-Lib produira des images binaires (image descassettes, disquettes, cartouches, etc.) prêtes à être utilisées avec des émulateurs ou avec des machines réelles. Actuellement `xl run` ne supporte qu'un nombre très limité d'émulateurs. En général il faudra lancer un émulateur et charger l'image produite par Cross-Lib depuis l'émulateur. On trouve plus de détails dans la page *GitHub* du projet [xlib].

Pour charger les images sur des machines réelles, il faudra soit les convertir dans un format lisible pour la machine spécifique soit utiliser un périphérique moderne, comme par exemple *SD2IEC* pour les Commodores 8-bit et les images de type `.prg` et `.d64`, depuis une machine réelle en émulant un lecteur de disquettes. Pour la plupart des images des cassettes on peut les convertir dans un format lisible par le magnétophone (par exemple en format WAV encodé spécifiquement pour la machine) avec des outils spécifiques pour chaque machine. Il faudra connecter la sortie audio du PC à l'entrée audio du retro-ordinateur. Dans certains cas on pourrait utiliser un smartphone ou lecteur MP3 à la place du PC et le connecter à un adaptateur de cassettes de voiture (celui qu'on utilise normalement pour reproduire des MP3 ou WAV dans un magnétophone) et on mettra l'adaptateur de cassettes dans le magnétophone du retro-ordinateur.

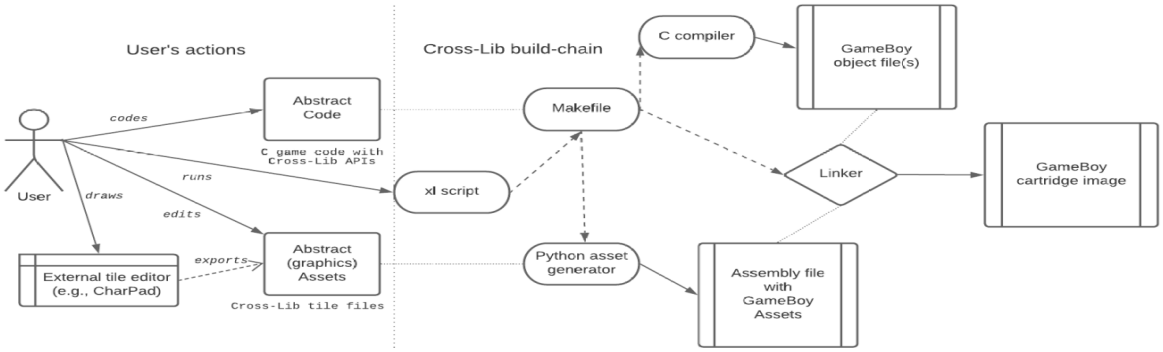
8. LE FLUX DU DÉVELOPPEMENT

Dans ce diagramme on décrit le flux de développement suivi par le développeur :



où l'on voit les opérations que l'utilisateur peut faire pour coder avec Cross-Lib, notamment :
1. écrire du code abstrait avec les APIs Cross-Lib ;
2. dessiner les graphismes (*tiles*) avec un outil externe ou directement en éditant les ressources graphiques (« assets ») ;
3. utiliser le script `xl` pour compiler, lancer, tester, etc.

Le flux de développement complet avec les opérations déclenchées par la *tool-chain* est décrit par le diagramme suivant :



9. CRÉER UN NOUVEAU PROJET

Un projet est juste un dossier dans `src/games/`, `src/examples/` ou `src/projects`. Ce dernier répertoire contient les projets définis par l'utilisateur.
On pourrait créer manuellement un répertoire avec des fichiers `.c`, un répertoire `tiles` (avec tous ses sous-répertoires) et un fichier `Makefile.<project>` mais il est possible de faire tout cela plus simplement avec

```
xl create <project> <type>
```

où `<project>` est le nom du nouveau projet et le paramètre optionnel `<type>` est le type (`arcade, text, helloworld, test`) de projet qui définira le code initial du projet.

Par exemple :

```
xl create foo arcade
```

va créer un projet `foo` avec du code initial pour un jeu de type arcade (avec certaines routines pour gérer score, record, input, etc.).

On pourra le compiler pour le terminal natif avec

```
xl foo
```

ou cross-compiler pour une machine, comme par exemple la *GameBoy*, avec

```
xl foo gb
```

Un autre manière de créer un projet est de cloner un projet avec :

```
xl clone <source project> <new project>
```

Par exemple, on peut cloner le jeu *Cross Horde* et créer un projet `bar` qui au début aura le même code avec :

```
xl clone horde bar
```

10. LE DISPLAY

Dans cette section on montrera les principales fonctions pour afficher des objets sur l'écran.

10.1 INITIALISER ET EFFACER L'ÉCRAN

Tout d'abord avant de faire quoi que ce soit il faudra initialiser le display avec

```
void _XL_INIT_DISPLAY(void)
```

Avant de commencer à utiliser l'écran, il faudra aussi effacer le contenu de l'écran avec

```
void _XL_CLEAR_SCREEN(void)
```

comme on a vu dans l'exemple qui affiche « HELLO WORLD ».

10.2 LES COULEURS

Tout d'abord il faut rappeler que chaque machine spécifique peut ne pas avoir de couleurs ou avoir un nombre très limité de couleurs possibles. Quand on code avec Cross-Lib, on choisit des couleurs abstraites qui, ensuite, seront approximées sur chaque machine spécifique de façon différente. Sur les machines qui n'ont pas de couleurs, le choix d'une couleur n'aura aucun effet. Sur les machines où une couleur est absente, cette couleur sera approximée avec une autre qui ne sera pas forcément très proche à celle choisi avec Cross-Lib surtout si la machine a un nombre très limité de couleurs possibles. Par exemple la couleur bleue pourrait être rendue avec du vert sur une machine qui n'a pas cette couleur ou qui n'a pas assez de couleurs affichables en même temps.
Il faut aussi remarquer que l'écran a une couleur d'arrière-plan qui sera fixée par le paramètre `BACKGROUND_COLOR` spécifié dans le fichier `Makefile.<project>` du projet `<project>` avec la couleur noire par défaut. Actuellement, les seules couleurs supportées pour l'arrière-plan ne sont que le noir (`_XL_BLACK`) et le blanc (`_XL_WHITE`).

Les couleurs du premier plan avec un arrière plan noir (couleur par défaut) actuellement supportées sont : `_XL_WHITE, _XL_RED, _XL_GREEN, _XL_CYAN, _XL_YELLOW, _XL_BLUE ;`
et avec un arrière plan blanc :
`_XL_BLACK, _XL_RED, _XL_GREEN, _XL_CYAN, _XL_YELLOW, _XL_BLUE.`
Donc on ne supporte pas une couleur de premier plan qui coïncide avec celle de l'arrière-plan (qui n'a pas beaucoup d'intérêt).

Dans le futur on prévoit davantage de couleurs mais il ne faudra jamais oublier que les couleurs utilisées avec Cross-Lib dans le code seront toujours approximées sur chaque machine différente avec la couleur possible la plus proche sur la machine spécifique. Pour cette raison il faudra toujours choisir des couleurs très différentes si l'on veut être sûr qu'elles soient affichées avec une couleur réelle différente sur la plupart des machines.

10.3 TILES ET CARACTÈRES

Cross-Lib permet d'afficher deux types d'objets différents :
- *caractères* : caractères alphanumériques (dont on ne peut pas définir la forme, qui restera la même pour chaque machine sur tous les projets mais pas forcément exactement la même sur toutes les machines) ;
- *tiles* : formes qu'on peut définir avec une liberté plus ou moins étendue selon la machine.

L'écran est vu comme une sorte de grille de taille donnée par les *macros* `XSize` par `YSize` dans laquelle chaque « cellule » contient une des *tiles* ou caractères possibles.

Actuellement chaque tile et caractère ne peut avoir qu'une seul couleur de premier plan et la couleur du fond est partagée sur tout l'écran.

10.4 TILES

Les graphismes sur Cross-Lib sont exclusivement basés sur les tiles. Il existe principalement deux types de machines sur Cross-Lib : celles sans graphismes et celles avec graphismes. La différence principale entre les deux types de machines sera donnée par les « formes » possibles qu'on pourra donner aux tiles.

Dans le code les tiles sont représentées par les entiers non-négatifs à 8-bit (`uint8_t`) donnés par les macros :

```
_TILE_0, _TILE_1, ...,_TILE_{N-1}
```

Actuellement N est égal à 26. Donc on a un nombre très limité de tiles dans la version actuelle de Cross-Lib.

Le code « ne connaît pas la forme » des tiles et celle-ci n'est même pas définie dans le code. Toutes les formes des tiles sont définies dans les répertoires dans `src/<project>/tiles/`.

Pour afficher la tile `<tile>` sur la « cellule » `(x,y)` avec la couleur de premier plan `<color>` on utilise

```
void _XL_DRAW(uint8_t x, uint8_t y, uint8_t tile, uint8_t color)
```

et pour effacer la « cellule » `(x,y)` :

```
void _XL_DELETE(uint8_t x, uint8_t y)
```

Par exemple, la ligne de code suivante

```
_XL_DRAW(12,5,_TILE_7,_XL_WHITE);
```

affichera la tile numéro 7, avec la couleur blanche de premier plan ayant pour coordonnées x=12, y=5 où la position initiale (0,0) correspond à l'angle supérieur gauche.

L'exemple `tiles` qui se trouve dans le répertoire `src/examples/tiles` montre comment utiliser `_XL_DRAW` et affiche les mêmes tiles sur tous les machines. Par exemple on peut le compiler pour le Commodore VIC 20 avec `xl tiles vic20`. Lorsqu'on lance cet exemple sur le Commodore VIC 20 on verra que certains tiles sont des *pre-shifted tiles*, c'est-à-dire qu'elles décrivent l'animation et le mouvement d'un personnage :

MACHINES SANS GRAPHISMES : Pour les machines qui n'ont pas de mode graphique, les seules formes possibles pour les tiles sont celles des codes ASCII (ou de l'extension ASCII spécifique à chaque machine).
Dans ce cas-là, le code ASCII de chaque tile est défini dans le fichier `src/<project>/tiles/ASCII /char_tiles.h`. Par exemple `src/snake/tiles/ASCII/char_tiles.h` contient les définitions des « formes » des tiles pour le jeu *Cross Snake* pour toutes les machines qui n'ont pas de graphismes.

MACHINES AVEC GRAPHISMES : Pour les machines avec de vrais graphismes possibles, on pourra définir la forme dans les moindres détails, pixel par pixel. La taille des tiles (nombre de pixels horizontaux et verticaux) dépend de la machine. Pour chaque taille de tiles possible, on définira une seule fois la forme qui sera partagée sur toutes les machines avec la même taille de tiles. Par exemple toutes les machines avec des tiles de 8 pixels par 8 pixels utilisent les mêmes définitions.

Le dossier `src/<project>/tiles/` contient des dossiers pour chaque taille supportée. Actuellement on a :
- `src/<project>/tiles/8x8/` : pour la plupart des machines ;
- `src/<project>/tiles/6x8/` : pour la série `Oric` et des ordinateurs `NTSC` basés sur le `RCA 1802` ;
- `src/<project>/tiles/6x9/` : pour la plupart des ordinateurs `PAL` basés sur la puce `RCA 1802` ;
- `src/<project>/tiles/7x8/` : pour les `Apple][` et `Apple//e` en modalité `HGR`.

Par exemple `src/bomber/tiles/8x8/` contient les définitions de tiles pour le jeu *Cross Bomber* pour toutes les machines qui ont de vrais graphismes avec tiles de taille 8x8.

Dans chacun de ces dossiers on trouve les fichiers `TILE_0.txt, TILE_1.txt,…` qui définissent la forme de chaque tile. Le format de chacun de ces fichiers est un CSV avec des chiffres en décimal ou hexadécimal (notation `$XX`) où chaque élément du CSV représente une ligne de la tile.

Pour créer de nouvelles formes on peut :
1. dessiner les tiles avec des outils externes (comme *CharPad*) qui puissent exporter dans un format Assembleur 8-bit quelconque et les importer sur un projet ;
2. créer des fichiers de texte avec une « image » dont les pixels sont représentés par `` `#` `` pour le `1` (couleur du premier plan) et `` `.` `` pour le `0` (couleur de l'arrière-plan) et les importer sur un projet ;
3. éditer manuellement les fichiers `TILE_0.txt, TILE_1.txt,...`

OUTILS EXTERNES : Pour produire des fichiers Assembleur qu'on pourra convertir dans le format Cross-Lib il faudra des outils qui exportent en format Assembleur sous forme de directives « .byte » ou « .decb » ou « fcb » ou « db ».

Par exemple on peut utiliser :

`VChar64` https://retro.moe/2015/02/10/vchar64-character-editor-for-the-commodore-64/

`CharPad` https://subchristsoftware.itch.io/charpad-free-edition

`C64 Graphics Maker` https://agpx.itch.io/c64-graphics-maker

`Magellan` https://magellan.js99er.net/

Pour importer sur un projet il suffira d'utiliser `xl import <file> <project>`.
Par exemple on peut créer un nouveau project `foo` et importer les tiles décrites par le fichier `./assets/examples/tile_sets/tiles/tiles_all.asm` avec :

```
xl create foo
xl import ./assets/examples/tile_sets/tiles/tiles_all.asm foo
```

Il faut remarquer que cette méthode ne fera l'importation que pour les tiles en format 8x8 (qui couvre la plupart des cas). Pour les autres cas, il faudra adapter les tiles 8x8 ou éditer d'autres tiles.

Si l'on veut juste regarder les formes des tiles contenues dans le fichier Assembleur on peut utiliser `xl import <asm file>`. Par exemple :

```
xl import ./assets/examples/tile_sets/tiles/tiles_all.asm
```

nous montrera les formes de tiles décrites avec des caractères `` `#` `` et `` `.` ``.

Il faut remarquer que `xl import` est capable de faire beaucoup plus : il peut faire du *ripping* et importer les tiles depuis du code source écrit en plusieurs dialectes Assembleurs et BASIC 8-bit en cherchant des patterns dans le texte du code. On trouve des exemples en plusieurs dialectes BASIC (y compris des exemples de BASIC 10-liner qui ont participé à la compétition BASIC 10-liner [bas10]) et Assembleur dans `src/assets/examples/` .

DESSINER LES TILES COMME DU TEXTE : Une autre solution est d'éditer les tiles dans des fichiers de texte avec

`#` pour le `1` (couleur du premier plan) et `.` pour le `0` (couleur d'arrière-plan) et avec un maximum de 8 colonnes et un nombre libre de lignes.

Pour importer chaque tile pour chaque format possible on pourra simplement utiliser

```
xl tile <tile text file> <project> <index>
```

qui importera une tile sur un projet pour le format du fichier de texte.

Par exemple on pourra importer une tile décrite par un fichier de texte comme celui-ci :

dans un projet `foo` qu'on a créé (par exemple avec `xl create foo`) comme tile numéro 7 (format 8x8 détecté automatiquement) avec :

```
xl tile ./assets/examples/single_tiles/tile_shape0.txt foo 7
```

Cross-Lib n'expose pas d'APIs pour les *sprites* (des objets qu'on peut placer sur l'écran avec une précision supérieure à celle des tiles et même au pixel près si on le désire) mais on peut les implémenter à travers des *pre-shifted tiles*, c'est-à-dire des tiles qui décrivent le mouvement et/ou l'animation d'un *sprite*. Les jeux *Cross Bomber*, *Cross Snake*, et *Cross Horde* utilisent des *pre-shifted tiles* que l'on peut voir avec `xl show <project>` comme par exemple :

```
xl show bomber
```

qui montrera les tiles utilisées dans le jeu *Cross Bomber*. Plus d'information sur `xl show` peut être affichée avec `xl help show`.

10.5 CARACTÈRES

Cross-Lib ne permet d'afficher sur toutes les machines que les caractères suivants :
- lettres majuscules de l'alphabet (`A`-`Z`),
- les chiffres (`0`-`9`) et
- le caractère espace.

Sur certaines machines on peut afficher les lettres minuscules.
Les formes des caractères sont fixées pour chaque machine sur tous les projets.

Pour définir la couleur du texte que l'on veut afficher on utilise

```
void _XL_SET_TEXT_COLOR(uint8_t color)
```

La couleur est maintenue pour tous les commandes qui affichent des caractères mais pas forcement après des commandes pour afficher des tiles.

Pour afficher les lettres majuscules, les chiffres et l'espace on peut utiliser :

`void _XL_CHAR(uint8_t x, uint8_t y, char ch)` pour un seul caractère

et

`void _XL_PRINT(uint8_t x, uint8_t y, char * str)` pour une chaîne de caractères.

Pour afficher des nombres entiers non-négatifs à 16 bit (`uint16_t`) avec un nombre exact de chiffres donné on peut utiliser :

```
void _XL_PRINTD(uint8_t x, uint8_t y, uint8_t digits, uint16_t number)
```

Par exemple:

```
_XL_PRINTD(0,0,5,42U)
```

affichera `00042` dans la cellule `(0,0)`, en haut à gauche.

11. INPUT, MANETTE OU CLAVIER

Selon la machine, l'input se fera soit à travers la manette (*joystick*) soit à travers le clavier (par défaut avec les touches `` `I` ``,`` `J` ``,`` `K` ``,`` `L` `` pour les directions et `ESPACE` pour *fire*).

Avant de commencer il faudra toujours initialiser l'input avec

```
void _XL_INIT(void)
```

Les routines pour l'input, les plus importantes sont :

```
uint8_t _XL_INPUT(void)
uint8_t _XL_LEFT(uint8_t input)
uint8_t _XL_RIGHT(uint8_t input)
uint8_t _XL_UP(uint8_t input)
uint8_t _XL_DOWN(uint8_t input)
uint8_t _XL_FIRE(uint8_t input)
uint8_t _XL_WAIT_FOR_INPUT(void)
```

Avec `_XL_INPUT` on peut lire l'input, et avec `_XL_LEFT`, `_XL_RIGHT`, `_XL_UP`, `_XL_DOWN` on peut l'interpréter.
Par exemple pour détecter le mouvement à gauche :

```
...
uint8_t input;
...
input = _XL_INPUT();
..
if(_XL_LEFT(input))
{
    // Handle left direction
    ...
}
```

Avec `_XL_WAIT_FOR_INPUT` on bloque l'exécution jusqu'à la pression d'une touche (une touche du clavier ou le bouton fire pour les machines avec joystick).

12. EFFETS SONORES

La version actuelle de Cross-Lib ne prévoit qu'un nombre très limité d'effets sonores prédéfinis.

Trois effets sonores très courts sont produits par :

```
void _XL_PING_SOUND(void)
void _XL_TICK_SOUND(void)
void _XL_TOCK_SOUND(void)
```

Un effet plus long est produit par :

```
void _XL_ZAP_SOUND(void)
```

On peut produire deux bruits différents avec :

```
void _XL_SHOOT_SOUND(void)
void _XL_EXPLOSION_SOUND(void)
```

13. AUTRES FONCTIONS UTILES

Pour pouvoir coder un jeu avec le même code il faut aussi d'autres fonctions comme celle qui permet de mettre en pause le jeu et une fonction qui puisse générer des nombres aléatoires avec exactement le même range à 15 bit (0-32676) sur toutes le machines.

13.1 PAUSES

PAUSE LONGUE : Pour mettre en pause un jeu pour un nombre de secondes on peut utiliser :

```
_XL_SLEEP(uint8_t sec)
```

À noter que sur certaines machines la durée d'une seconde peut être une approximation.

MICRO-PAUSE : On peut mettre en pause un jeu pour un « nombre de boucles » avec :

```
_XL_SLOWDOWN(uint16_t loops)
```

où chaque boucle (qui correspond à une boucle vide de code C) aura une durée différente sur chaque machine et qu'il faudra donc multiplier par un facteur de proportionnalité qui est donné par la macro `_XL_SLOWDOWN_FACTOR`, qui fera en sorte d'avoir soit la même pause sur toutes les machines ou une pause qui dépendra par exemple de la taille de l'écran si l'on veut qu'un jeu tourne moins rapidement sur certaines machines. La valeur de `_XL_SLOWDOWN_FACTOR` est définie dans le fichier `Makefile.<project>` pour chaque projet.

13.2 NOMBRES ALÉATOIRES

En ANSI C la fonction `rand()` qui est incluse dans `stdlib.h` n'a pas un range exactement défini dans le standard et elle n'est pas présente sur tous les dev-kits 8-bit.
Pour cette raison Cross-Lib dispose d'une fonction qui produit des nombres pseudo-aléatoires non-négatifs à 15-bit (0-32767) sur toutes les machines :

```
uint16_t _XL_RAND(void)
```

14. UNE PETITE ANIMATION

Avec les commandes qu'on a vues jusqu'ici, on peut déjà créer un jeu embryonnaire avec un petit bonhomme contrôlé par la manette ou le clavier. Un tel exemple est déjà présent dans les exemples qu'on trouve dans le répertoire `src/examples/animate/` :

```
#include "cross_lib.h"

#define MIN_X 1
#define MAX_X (XSize-2)
#define MIN_Y 1
#define MAX_Y (YSize-2)
#define DOWN_TILE  _TILE_0
#define UP_TILE    _TILE_1
#define RIGHT_TILE _TILE_2
#define LEFT_TILE  _TILE_3
#define FIRE_TILE  _TILE_4

int main(void)
```

```
{
 uint8_t x;
 uint8_t y;
 uint8_t input;
 uint8_t tile;

 _XL_INIT_GRAPHICS();
 _XL_INIT_SOUND();
 _XL_INIT_INPUT();

 _XL_CLEAR_SCREEN();

 x = XSize/2;
 y = YSize/2;
 tile = DOWN_TILE;
 hile(1)
 {
        _XL_SET_TEXT_COLOR(_XL_WHITE);
        _XL_PRINTD(0,0,3,x);
        _XL_PRINTD(5,0,3,y);

        input = _XL_INPUT();
        if(_XL_UP(input))
        {
            _XL_DELETE(x,y);
            tile = UP_TILE;
            --y;
        }
        else if (_XL_DOWN(input))
        {
            _XL_DELETE(x,y);
            tile = DOWN_TILE;
            ++y;
        }
        else if (_XL_LEFT(input))
        {
            _XL_DELETE(x,y);
            tile = LEFT_TILE;
            --x;
        }
        else if(_XL_RIGHT(input))
        {
            _XL_DELETE(x,y);
            tile = RIGHT_TILE;
            ++x;
        }
        else if(_XL_FIRE(input))
        {
            _XL_DRAW(x,y,FIRE_TILE,_XL_WHITE);
            _XL_EXPLOSION_SOUND();
            _XL_SLOW_DOWN(16*_XL_SLOW_DOWN_FACTOR);
        }

        if((y>=MIN_Y)&&(y<=MAX_Y)&&(x>=MIN_X)&&(x<=MAX_X))
        {
            _XL_DRAW(x,y,tile,_XL_WHITE);
        }
        else
        {
            _XL_DELETE(x,y);
            _XL_ZAP_SOUND();
            _XL_SLOW_DOWN(16*_XL_SLOW_DOWN_FACTOR);
            x = XSize/2;
            y = YSize/2;
        }
        _XL_SLOW_DOWN(4*_XL_SLOW_DOWN_FACTOR);
    }
    return EXIT_SUCCESS;
}
```

15. COMMENT CODER POUR TOUTES LES MACHINES

Cross-Lib supporte des machines avec des ressources hardware très différentes: taille de l'écran différente, présence ou absence de graphismes, d'effets sonores possibles, couleurs, etc. Les APIs de Cross-Lib ne suffisent pas pour avoir du

code qui s'adaptera à tous les cas possibles. Par contre Cross-Lin expose des macros qui permettent d'écrire du code capable de s'adapter à toutes les machines.

Par exemple pour avoir un jeu capable de s'adapter à toutes les tailles d'écran il faudra utiliser des fractions entières de macros `XSize` et `YSize` pour s'adresser aux « cellules » de l'écran. Dans certains cas il faudra utiliser des directives `#if` pour gérer des écrans avec des tailles minuscules (comme c'est le cas pour certaines consoles *hand-held* ou ordinateurs moins équipés ou calculatrices).

Pour avoir un jeu qui marche aussi bien sur une machine avec couleurs que sur une sans couleurs il faudra soit éviter que le jeu utilise les couleurs dans sa logique ou prévoir des directives comme `#if defined(_XL_NO_COLOR)` pour implémenter une partie de la logique pour les machines avec couleurs et une autre pour les machines sans couleurs. Idéalement on veut minimiser ou ne pas avoir de directives mais parfois on n'a pas de choix si l'on veut un jeu qui tourne bien sur toutes les machines.

Cross-Lib expose les macros qui décrivent les détails de ce que Cross-Lib prévoit pour chaque machine pour permettre au développeur de gérer les différences. Si l'on veut connaître ce que Cross-Lib a implémenté pour une machine précise, il suffira de regarder et compiler l'exemple `target` (dont le code se trouve dans `src/examples/target/`) pour la machine qui nous intéresse. Le code de cet exemple utilise tous les macros disponibles pour gérer les différences entre les machines : `XSize`, `YYize`, `_XL_TILE_X_SIZE`, `_XL_TILE_Y_SIZE`, `_XL_NUMBER_OF_TILES`, `_XL_NO_COLOR`, `_XL_NO_TEXT_COLOR`, `_XL_NO_UDG`, `_XL_NO_JOYSTICK`, `_XL_NO_COLOR`, `_XL_NO_SOUND`, `_XL_NO_CAPITAL_LETTERS`.

Par exemple

```
xl target
```

produira un fichier exécutable pour le terminal natif que l'on pourra lancer avec

```
xl run target
```

qui montre que pour le terminal natif, Cross-Lib prévoit un écran 80x24 avec des tiles « ASCII » (sans graphismes) et avec tous les autres *flags* actifs à l'exception des effets sonores et de la manette (*joystick*) :

```
TARGET INFORMATION
XSIZE 80  YSIZE 24
TILES 26  ASCII
GRAPHICS       OFF
COLOR          ON
TEXT COLOR     ON
JOYSTICK       OFF
SOUND          OFF
SMALL CHARS    ON
```

Par contre avec

```
xl target msx
```

on produira l'image d'une cartouche (*rom*) pour les ordinateurs *MSX* qui nous montre que pour ces ordinateurs, Cross-Lib prévoit un écran 32x24 avec des tiles 8x8 (donc avec graphismes) et avec tous les *flags* actifs.

```
TARGET INFORMATION
XSIZE 32  YSIZE 24
TILES 26  8X8
GRAPHICS       ON
COLOR          ON
TEXT COLOR     ON
JOYSTICK       ON
SOUND          ON
SMALL CHARS    ON
```

CONCLUSION

On a montré les principales fonctions de Cross-Lib qui permettent l'écriture de jeux universels pour toutes les machines 8-bit y compris les plus exotiques. Les avantages actuels les plus importants de Cross-Lib sont le fait d'être WORA et universel avec environ 200 machines supportées mais tout cela a été possible aussi parce que Cross-Lib n'expose qu'une petite partie des caractéristiques de chaque machine. Une limitation importante actuellement est due au nombre de tiles disponibles. Cette contrainte pourrait être levée dans les versions futures de Cross-Lib mais le but de Cross-Lib restera toujours d'être universel. Cela n'est pas si simple parce que dans certaines machines la couleur d'une tile hardware n'est pas libre et certaines machines ont un nombre très limité de tiles hardware. Un compromis possible pour avoir plus de tiles et rester en même temps compatible avec toutes les machines pourrait être de créer une option pour sélectionner entre davantage de tiles et davantage de couleurs. Une autre limitation qui pourrait être réduite est celle du son, qui est actuellement limité à quelques effets sonores prédéfinis. Un autre aspect qu'on prévoit d'améliorer est l'installation des dépendances (compilateurs, outils de build et développement), qui pourrait ne plus être nécessaire avec une image Docker prête à être utilisée avec Cross-Lib et toutes ses dépendances.

ANNEXE : Installation sous Ubuntu 20.04

Sous Ubuntu 20.04 (aussi utilisable sous Windows 10 grâce au Windows Subsystem for Linux), on pourrait installer tout ce qu'il est nécessaire pour compiler pour la majorité des machines basés sur le MOS 6502 avec les commandes suivantes.
Pour installer les outils de build nécessaires :

```
sudo apt-get update -y
sudo apt install make
sudo apt-install python
```

Pour installer une version récente de *CC65* (en 2021) :

```
sudo apt-install cc65
```

Pour Cross-Lib, il faut tout simplement le télécharger avec

```
git clone https://github.com/Fabrizio-Caruso/CROSS-LIB.git
```

Pour émuler les Commodore 8-bit on peut installer Vice avec

```
sudo apt-install vice
```

et copier les roms (que l'on trouve « ailleurs ») dans les répertoires sous `/usr/lib/vice/`

On pourra compiler les jeux et les exemples distribués avec Cross-Lib avec

```
xl <projet> <target>
```

Donc pour compiler *Cross Snake* pour le VIC 20 il suffira :

```
xl snake vic20
```

Bibliographie

[ack] **ACK**, Tanenbaum A., Jacobs C. *et al.*, pour l'Intel 8088, PDP 11, CP/M-80,etc., https://github.com/davidgiven/ack

[app] **Cross Chase: A Massively 8-bi Multi-System Game**, Caruso F., *Call-A.P.P.L.E.*, vol. 28, No. 1, Feb 2018, pages 31-33, http://www.callapple.org

[bas10] **BASIC 10-liner competition**, Kanold G. : https://gkanold.wixsite.com/homeputerium/

[east] **Discovering Eastern Europe PCs by Hacking Them… Today**, Bodrato S., Caruso F., Cignoni G., *IFIP International Conference on the History of Computing (HC)*, Sep 2018, Poznan, Poland. pp.279-294

[c8b] **8-bit C**, Caruso F., https://github.com/Fabrizio-Caruso/8bitC

[cc6303] **CC6303**, Cox A., Compilateur C pour le Motorola 6800 et 6803, https://github.com/EtchedPixels/CC6303

[cc65] **CC65**, Compilateur C pour l'architecture MOS 6502, https://sourceforge.net/projects/cc65/

[ceo] **Cross Chase**, Caruso F., *CEO-MAG*, No. 327-328, Juillet-Août 2017, https://ceo.oric.org/

[cmoc] **CMOC**, Sarrazin P., Compilateur C pour le Motorola 6809, https://perso.b2b2c.ca/~sarrazip/dev/cmoc.html

[effc] **Efficient C Code for 8-bit Microcontrollers**, Jones N., https://barrgroup.com/embedded-systems/how-to/efficient-c-code

[m20] **Z8KGCC**, *Groessler C.,* GCC pour l'Olivetti M20, http://www.z80ne.com/m20/sections/download/z8kgcc/z8kgcc.html

[lcc1802] **LCC1802**, Rowe B., Compilateur C pour le RCA 1802, https://github.com/bill2009/lcc1802

[st] **GCC**, Rivière V., version pour l'Atari ST, http://vincent.riviere.free.fr/soft/m68k-atari-mint/

[ti] **GCC**, (*Insomnia*), version pour le TI99/4A, http://atariage.com/forums/topic/164295-gcc-for-the-ti

[trse] **TRSE**, Groeneboom N. E. *et al*., dev-kit basé sur un dialecte Pascal pour plusieurs machines 8-bit et 16-bit, https://lemonspawn.com/turbo-rascal-syntax-error-expected-but-begin/

[un8] **8-Bit Unity**, Beaucamp A. (*8-bit Dude*), http://8bit-unity.com/

[vbcc] **VBCC**, Barthelmann V., Compilateur C pour le BBC Micro, le BBC Master, l'Amiga, etc., http://www.compilers.de/vbcc.html

[xlib] **Cross-Lib**, Caruso F., https://github.com/Fabrizio-Caruso/CROSS-LIB/

[xtc68] **XTC68**, Hudson J., Compilateur C pour le Sinclair QL, https://github.com/stronnag/xtc68

[z88dk] **Z88DK**, Compilateur C pour les architectures Zilog 80 et Intel 8080, https://github.com/z88dk/z88dk