

# L'émulation console

Par BestCoder



[www.openclassrooms.com](http://www.openclassrooms.com)

*Licence Creative Commons 6 2.0  
Dernière mise à jour le 14/02/2012*

# Sommaire

Sommaire .....	2
Lire aussi .....	1
L'émulation console .....	3
Partie 1 : Notre premier émulateur .....	4
Un tour d'horizon .....	4
Définition de l'émulation .....	4
Émulation .....	4
Les ROM .....	5
Les consoles de jeu .....	6
Les informations sur les consoles de jeu .....	7
Par où commencer .....	7
Ce qu'il faut rechercher .....	7
Les obstacles liés aux nouvelles consoles .....	8
Législation .....	9
Les obstacles .....	9
Une lueur d'espoir .....	10
Quelle console émuler ? .....	10
Critères de sélection .....	11
La Chip 8 .....	11
Définition .....	11
Caractéristiques de la Chip 8 .....	11
La RCA 1802 .....	13
La base .....	14
L'implémentation de la machine .....	14
La mémoire .....	14
Les registres .....	14
La pile ou stack .....	15
Les compteurs .....	15
Le graphique .....	16
Création des pixels .....	16
Modifier l'écran .....	23
Simulation des instructions .....	24
Le cadencement du CPU et les FPS .....	25
Comment allons-nous procéder ? .....	25
Récapitulatif .....	25
Lecture de l'opcode .....	27
Identification de l'instruction .....	27
Notre table de correspondance .....	28
Retour sur le graphique .....	31
Ligne importante .....	33
Exemples avec quelques instructions .....	39
Divers .....	39
Le mode de dessin intégré .....	43
Charger un jeu .....	45
L'interface homme-machine .....	59
Les entrées utilisateur .....	59
Le son .....	62
À vous de jouer ! .....	62
Améliorations .....	62
La compatibilité .....	78
À suivre .....	79



# L'émulation console



Mise à jour : 14/02/2012

Difficulté : Difficile  Durée d'étude : 20 jours



Vous vous êtes toujours demandé par où commencer pour programmer un émulateur ? Vous voulez mettre en pratique vos connaissances dans un langage de programmation ? Ou vous voulez ramener à la vie votre console des années 70 ? 😊

Ce tutoriel est alors fait pour vous. Un émulateur, en général, permet de simuler une machine spécifique. Dans notre cas, nous allons copier le fonctionnement d'une console de jeu.

Étant donné l'abondance et la diversité de ces machines, il est utile de préciser que ce tutoriel n'expliquera pas tout ce qu'il faut pour émuler n'importe quelle console, mais il sera surtout un support pour bien débiter dans le passionnant domaine de l'émulation.

Nous allons donc voir quelques généralités sur l'émulation avant d'entrer dans le vif du sujet.

## Partie 1 : Notre premier émulateur

Vous avez sûrement fait des détours sur de nombreux sites web ou forums où vous trouvez un membre qui vous dit que pour créer un émulateur, il faut programmer pendant dix ans et connaître l'assembleur, etc. 🐼

Moi, je vous assure le contraire, et vous en aurez le cœur net à la fin de ce tutoriel. Pour programmer son émulateur, il suffit d'être patient et méthodique, rien de plus.

Pour commencer, nous allons passer en revue les différents aspects de l'émulation avant de nous lancer dans la programmation proprement dite.



Notre émulateur en action

### Un tour d'horizon

Nous allons commencer notre aventure avec un voyage dans les coulisses de l'émulation. Comment est-il possible de créer un émulateur ? Comment faut-il s'y prendre pour espérer en créer un ? Cette partie nous permettra de répondre à ces différentes interrogations.

#### Définition de l'émulation

Je parie que vous avez, plusieurs fois, fait des recherches pour savoir comment programmer un émulateur et que vous vous êtes retrouvés sur des sites vous montrant comment en configurer un, déjà existant.

Votre calvaire vient de prendre fin. Grâce à ce tutoriel, vous serez en mesure de programmer le vôtre de A à Z.

Avant de commencer notre passionnante aventure, nous allons définir les différentes notions que nous utiliserons afin de mieux cerner nos objectifs.



Depuis un moment, tu nous parles d'émulateur, d'émulation, mais en quoi cela consiste-t-il réellement ?

### Émulation

#### Définition

D'après Wikipédia, en informatique, l'émulation consiste à substituer un élément de matériel informatique – tel un terminal informatique, un ordinateur ou une console de jeu – par un logiciel.

La définition du terme *émuler* est « chercher à imiter ». Il faut voir dans l'émulation une imitation du comportement physique d'un matériel par un logiciel, et ne pas la confondre avec la simulation. [Wikipédia](#)

Théoriquement, il est donc possible de créer un émulateur pour toutes les machines électroniques. Pour ce faire, il suffit juste de :

- connaître ses caractéristiques ;
- trouver un support au moins aussi puissant que la machine à émuler ;
- traduire les caractéristiques.

#### Difficultés

Vous verrez tout au long de ce tutoriel que programmer un émulateur n'a rien d'extraordinaire. Le plus difficile est de connaître le fonctionnement exact de la machine que l'on désire émuler.

Eh oui, les constructeurs n'ont aucun intérêt à publier les caractéristiques de leurs consoles et ce n'est pas aujourd'hui qu'ils le feront. 🤔

Pour trouver les informations qu'il leur faut, les programmeurs doivent donc avoir recours à diverses méthodes que l'on abordera dans le prochain chapitre.

## Comment fonctionne un émulateur ?

Un émulateur fonctionne avec une facilité qui pourrait même faire pleurer.

Voyez par vous-mêmes :

Code : C

```
while("Machine Fonctionne")
{
    regarder_ce_qu'il_faut_faire();
    Le_faire();
}
```

Tous les émulateurs ont la même structure de base. Le principe consiste tout simplement à regarder l'opération qu'il faut effectuer, puis l'effectuer. Cette routine sera exécutée tant que la machine – la console de jeu – est en marche.



Mais où est-ce qu'on va regarder ce qu'on doit faire ?  
Et comment va-t-on le faire ?

Patience, patience. 😊

## Les ROM

### Définition

Littéralement, ROM signifie « *Read Only Memory* », ou en français « mémoire à lecture seule ». C'est un support qui ne permet que la lecture des données qu'il contient. C'est le cas pour presque tous les supports de jeux vidéo.

Il existe divers moyens de copier le contenu de ces supports sur votre ordinateur. Pour ce faire, on utilise des ROM Dumper. Les fichiers binaires ainsi obtenus sont communément appelés : roms.

Les roms sont donc aux émulateurs ce que les cartouches de jeu – CD, DVD, etc. – sont aux consoles de jeu.



### Utilité

Je vous ai dit plus haut que la première étape pour créer un émulateur est de regarder ce qu'il faut faire. C'est là que les roms interviennent. Elles contiennent toutes les instructions à exécuter. C'est-à-dire que par « regarder ce qu'il faut faire », il faut comprendre « lire le contenu du fichier rom ».

Et comme vous le savez, lire le contenu d'un fichier binaire, ce n'est pas le diable. Je le dis et je le répète :

#### Citation : BestCoder

Vous verrez tout au long de ce tutoriel que programmer un émulateur n'a rien d'extraordinaire.

## Les consoles de jeu

### Définition

Une console de jeu est un appareil électronique conçu pour permettre de lire, interpréter et afficher les informations contenues dans un support conçu à cet effet (les cartouches, CD ou DVD de jeu).

Il existe deux principaux types de consoles :

- les consoles de salon, qui se branchent sur un écran pour afficher le jeu, et auxquelles on connecte accessoirement des manettes ;
- les consoles portables, de petite taille, qui possèdent leur propre écran et sont de ce fait autonomes et facilement transportables.

Wikipédia



Nos éventuelles victimes !

### Caractéristiques

La définition importe peu (je sais que vous le saviez déjà), mais je veux attirer votre attention sur les caractéristiques des consoles. Toutes les consoles de jeu sont basées sur le même principe. Elles sont constituées :

- d'un microprocesseur qui effectue les calculs ;
- de mémoire vive pour stocker les données ;
- d'une carte graphique pour afficher le rendu graphique ;
- de périphériques de contrôle et le plus souvent de manettes de jeu pour interagir avec la console.

Notre travail sera donc de remplacer tous ces éléments par ceux de notre ordinateur.

microprocesseur	processeur
mémoire vive	RAM
carte graphique	carte graphique

périphériques de contrôle	clavier, souris, <i>joystick</i>
---------------------------	----------------------------------

Ça y est, nous venons de faire le tour de l'émulation console. Vous êtes donc en mesure de programmer votre émulateur. 😊

Allez !

Non je blague, je ne vais pas vous abandonner si tôt. 😊 J'ai beaucoup de choses à vous dire avant de vous lâcher dans la nature.

Avec cette petite présentation, nous venons de voir en gros le travail que nous aurons à faire pour réaliser notre émulateur. Cette définition assez simpliste nous permet de mieux cerner la notion d'émulation et plusieurs questions doivent vous traverser l'esprit :



Comment doit-on s'y prendre pour remplacer les composants de la console ?



Et tant qu'on y est, est-ce que l'on a le droit de le faire ?

Pas de panique, vous serez éclairés sous peu.

## Les informations sur les consoles de jeu

Maintenant que nous savons à peu près comment programmer notre émulateur, certains se sont rués sur leur moteur de recherche (ou pas). Bon réflexe ! Pour simuler une machine, il faut connaître ses caractéristiques au maximum pour espérer des résultats satisfaisants. Ici, « connaître » veut juste dire qu'il faut avoir sous la main des documents qui décrivent exhaustivement la machine à émuler. Si vous possédez toutes les informations qu'il vous faut, il ne restera plus qu'à traduire ces informations dans un langage de programmation, sans même les comprendre si cela vous chante.

Mais la compréhension sera un atout majeur si vous ne voulez pas passer des journées à chercher un bug. Rassurez-vous, on n'y est pas encore.

## Par où commencer

C'est beau de dire « chercher », mais que faut-il chercher ? Si vous vous rendez sur un moteur de recherche et que vous tapez « caractéristiques de la Gameboy », vous obtiendrez des informations loin d'être suffisantes pour programmer votre émulateur. Si vous êtes malchanceux, vous verrez des liens vers des sites pour configurer des émulateurs déjà existants.

### *Savoir ce que vous voulez*

C'est bizarre de savoir ce qu'on est censé chercher, mais c'est la vie. Imaginez-vous dans un supermarché et que vous ne savez pas ce que vous souhaitez acheter. C'est la même chose que de lancer son moteur de recherche sans savoir ce que vous cherchez.

Donc pour obtenir des résultats qui correspondent à vos attentes, il faut connaître un tout petit peu le jargon de l'émulation.

### *Prendre le maximum possible*

Si vous faites vos recherches, prenez le maximum d'informations possible : soyez « boulimiques ». Cela vous permettra de faire des comparaisons et de voir celles qui sont le plus utilisées. En outre, la plupart des documentations ne sont pas claires sur tous les points. Elles seront donc complémentaires, ce qui facilitera grandement l'implémentation de notre émulateur.

Jetons maintenant un coup d'œil sur quelques mots clefs pour trouver notre bonheur :

- *opcode* ;
- CPU.

## Ce qu'il faut rechercher

Dans le chapitre précédent, j'avais dit :

#### Citation : BestCoder

C'est-à-dire que par « regarder ce qu'il faut faire », il faut comprendre « lire le contenu du fichier rom ».



Comment se passe réellement cette lecture ?

Lorsque le fichier **binaire** est à notre disposition, on le lit en entier et on le stocke dans un tableau. Je précise quand même que cela ne se passera pas comme cela pour un DVD : on n'interprétera pas le contenu d'un seul coup mais bout par bout, étape par étape. Ces « bouts » sont appelés *opcodes*.

Par exemple, nous pouvons avoir un fichier de 1 000 ko, mais les informations seront lues à coups de 1 ko. Chaque ko représentera notre *opcode*.

#### Opcode : Operation Code

L'*opcode* définit en quelque sorte toute action que peut effectuer une console du point de vue calcul et rendu graphique. Si vous regardez par exemple les *opcodes* de la Chip 8, vous en verrez 35. La Chip 8 ne peut donc effectuer que 35 opérations, et c'est tout. Ce sera au programmeur de voir comment faire son jeu avec 35 opérations. Il va de soi qu'une console récente aura des centaines d'*opcodes* ! 😊 Eh oui, plus c'est récent, plus vous allez en baver avant d'élaborer votre émulateur.

Selon la console qu'on émule, les *opcodes* ne sont pas de la même taille (en bits), mais le principe d'interprétation reste le même.

#### CPU : Central Processing Unit

Comme son nom l'indique, le CPU gère le fonctionnement de votre console de jeu. Il effectue les différentes opérations définies par les *opcodes*.

Cette exécution se fait toujours à une vitesse donnée : on parle de cadencement de la console. C'est une valeur qui est donnée en hertz (Hz). Par exemple, mon PC est cadencé à 2.0 GHz, soit  $2 \times 10^9$  cycles d'horloge par seconde.

Si vous désirez des informations portant sur la vitesse d'interprétation des *opcodes*, jetez un coup d'œil sur le CPU.

Si vous avez déjà testé des émulateurs et qu'ils vont trop vite ou trop lentement, le problème vient du fait que le CPU est mal implémenté (à moins que votre PC ne soit du dixième siècle). 😊



Il faudra toujours axer ses recherches autour de ces mots. J'utilise en priorité « *opcode* » parce que les résultats sont le plus souvent très fructueux. Mais « CPU » donne aussi des informations très complètes.

Preuves à l'appui, recherchez :

- « Gameboy caractéristiques » ;
- puis « Gameboy opcode » ;
- et enfin « Gameboy CPU ».

Vous verrez par vous-mêmes qu'il n'y a aucune comparaison à faire. Si vous trouvez des documents avec des choses peu courantes, n'ayez pas peur, vous serez en mesure de tout déchiffrer à la fin de ce tutoriel.

## Les obstacles liés aux nouvelles consoles

#### Le manque d'informations

Un gros obstacle à la programmation d'émulateurs reste le manque d'informations concernant la machine à émuler. En effet, pour bien développer son programme, il faut réunir un maximum d'informations avant de débiter. Pour ce faire, il existe des moyens



comme la rétro-ingénierie (*reverse engineering*) que l'on ne va pas aborder ici. Je ne la maîtrise d'ailleurs pas. 😊 En gros, la **rétro-ingénierie** permet de retrouver les caractéristiques d'une machine en effectuant divers tests sur celle-ci. Pour les machines anciennes, vous trouverez votre bonheur sur le Net étant donné que d'autres auront déjà fait les investigations à votre place. Cependant, vous pourrez trouver des caractéristiques différentes pour une même machine, chaque auteur pouvant avoir une compréhension différente de son fonctionnement. Malgré ces divergences, les principales caractéristiques restent le plus souvent les mêmes dans la plupart des documentations disponibles.

Maintenant, pour les machines récentes, le problème est tout autre. Vous pourrez obtenir des résultats, mais il n'y aura sûrement pas assez d'informations pour créer votre émulateur. Des séances de tâtonnement ne seront pas à exclure, ce qui rendra le travail particulièrement difficile. À moins que vous ne fassiez le *reverse engineering* vous-mêmes. 🤖

### Des consoles de plus en plus performantes

Mis à part la récupération des informations essentielles, il existe un plus gros obstacle devant vous : l'évolution des consoles. Faisons une petite comparaison :

Caractéristique	Game Boy	PSP
Cadence	2,2 Mhz	333 Mhz
Résolution	160 × 144 pixels	480 × 272 pixels
Couleur	14 nuances de gris	16,77 millions de couleurs

Ces chiffres parlent d'eux-mêmes : les nouvelles consoles sont de plus en plus puissantes. Donc, pour que votre émulateur ne tourne pas à deux à l'heure, il faudra une machine très, **très** puissante. Je doute même que l'émulation puisse suivre l'évolution fulgurante des consoles de jeu (avis personnel) : la XBOX 360 et la PS3 sont déjà cadencées à 3,2 GHz (plus que mon ordinateur à 2 GHz).

En plus de cela, l'organisation de ces consoles est tellement complexe qu'il est très difficile de tout implémenter en solo. Mais avec tout ce qu'il existe comme consoles à émuler, je vous assure qu'une vie entière ne suffirait pas. Jetez un coup d'œil [ici](#) pour en avoir le cœur net.

Voilà, nous savons maintenant ce qu'il faudra rechercher comme documentation autour des consoles de jeu pour les émuler toutes une par une.

À votre moteur de recherche ; prêts ? Partez !

### Législation



Est-ce que vous avez déjà vu un domaine dans lequel les juristes ne sont pas impliqués ? Bien qu'il n'existe pas encore, à ma connaissance, de lois spécifiques à l'émulation, il en existe qui défendent le droit d'auteur et la propriété intellectuelle.

Les consoles de jeu sont toutes sous licences propriétaires et sources de revenus. Le marché est très lucratif et ne cesse de s'étendre. Dès lors, créer un outil gratuit pour les substituer peut susciter des débats.



[A-t-on réellement le droit de créer un émulateur console ?](#)

### Les obstacles

L'un des plus gros problèmes pour l'émulation est sans doute le respect du droit d'auteur (*copyright*). En plus de ne divulguer aucune information sur les caractéristiques techniques de leurs consoles, les constructeurs les protègent jalousement. Ainsi, avec les licences utilisées, il est **formellement interdit** de prendre ou d'utiliser une partie ou la totalité de leur travail. D'ailleurs, les discussions sur la légalité de l'émulation font couler beaucoup d'encre.



Nous pouvons cependant retenir que la programmation d'un émulateur **est totalement légale** tant qu'on **ne fait pas usage de ressources sous droits d'auteur** (le BIOS, par exemple).

Ce genre de symboles vous est sûrement familier, ils sont présents sur presque tous les jeux vidéo pour signifier qu'ils sont sous droits d'auteur.

Tout au long de ce tutoriel, nous ferons du HLE (*High Level Emulation*), c'est-à-dire que nous programmerons tout ce dont nous

aurons besoin.



En résumé, votre émulateur restera légal tant que vous le développez avec vos propres ressources et que vous le redistribuez sans aucun jeu sous droits d'auteur.

Cependant, l'utilisation d'un émulateur nécessite des jeux ou roms qui sont le plus souvent difficiles d'accès et non libres d'utilisation. (Sinon : illégalité. 🙄)



Mais donc, ça sert à quoi de programmer un émulateur si on ne peut pas l'utiliser ?

## Une lueur d'espoir

Après ce que l'on vient de voir, il est légitime de se poser cette question. Mais il existe toujours des exceptions à toute règle.



Certains jeux (notamment ceux que nous allons utiliser) sont dans le domaine public et sont dès lors libres d'utilisation et de redistribution.

En plus de cela, vous êtes autorisés à posséder une rom à condition d'être en possession du jeu original, cette rom étant considérée comme une sauvegarde.

À défaut de tout cela, vous pourrez utiliser un *homebrew*. Un *homebrew* est un jeu de console créé par un amateur et ces jeux sont le plus souvent libres. Pour éviter de faire un cours de droit ici, je vous laisse creuser si cela vous chante. Pour de plus amples explications ou preuves, vous pouvez consulter :

- [Wikipédia : Émulation](#) ;
- [Wikipédia : Propriété intellectuelle](#) ;
- et pour les plus téméraires, [Legifrance](#).



Il existe des jeux avec des mentions « Non utilisable en dehors du support d'origine », donc attention !

Une dernière chose, je vous laisse le fameux :



Je, BestCoder, décline toute responsabilité sur des agissements qui pourraient survenir à la suite de la lecture de ce tutoriel.

BestCoder ne pourrait être tenu responsable sur aucun plan et ce en aucun cas. 🕵️

Je vous invite à accorder une grande importance à la législation pour éviter des ennuis inutiles.

Après ce long discours, j'espère que vous avez bien cerné le sujet et que votre curiosité a été bien assouvie. À présent, nous allons voir ce qu'il nous faut comme bagage pour nous lancer dans notre périlleuse aventure.

Nous voilà fin prêts et avertis pour nous lancer dans notre aventure. Puisque nous savons à la fois ce que nous voulons et comment l'obtenir, la moitié du travail est déjà effectué, il ne reste plus qu'à passer à l'action.



Je signale qu'il faut une certaine maîtrise de l'hexadécimal, du binaire et des opérations logiques pour suivre ce tutoriel sans trop de difficultés.

## Quelle console émuler ?

Pour construire un gratte-ciel, il faut des fondations solides. Nous aussi, pour faire des prouesses en émulation, nous allons débiter par des choses simples.

Ça ne sert à rien d'attaquer la NDS ou la PSP NGP en premier, vous allez vous casser les dents et détester l'émulation à jamais. 😞

Nous allons donc progresser lentement mais sûrement.

### Critères de sélection

Dans tous les domaines où l'on évolue, il est toujours préférable de procéder du plus facile au plus compliqué. Pour l'émulation, cette règle reste valable et la « machine » qui me paraît la plus appropriée est la Chip 8.

En effet, avec un faible nombre d'instructions (35 pour être plus précis) et un rendu graphique de  $64 \times 32$  pixels en noir et blanc, la Chip 8 est l'un des meilleurs supports pour débiter en émulation.

En plus de sa facilité d'implémentation, un bon nombre de jeux Chip 8 sont dans le domaine public, ce qui limite (supprime même) les problèmes liés à la légalité.



Mais c'est quoi la Chip 8 en réalité ?



Pourquoi as-tu mis « machine » entre guillemets ?

Vous serez éclairés sous peu.

### La Chip 8

#### Définition

La Chip 8 est en réalité un langage interprété qui a été utilisé sur le RCA TELMAC-1800 et le COSMAC VIP en 1977. Elle est constituée d'un ensemble d'instructions qui permettent une programmation facile pour lesdites machines.

Remodelée, la Chip 8 fut utilisée plus tard pour créer des calculatrices graphiques. D'ailleurs, plusieurs jeux ont été développés pour ce système et on ne manquera pas d'en discuter plus tard.



### Caractéristiques de la Chip 8

Voici les principales caractéristiques de la Chip 8. Le document est une traduction de la [présentation de Wikipédia](#).

#### La mémoire

Les adresses mémoire de la Chip 8 vont de \$200 à \$FFF (l'hexadécimal revient), faisant ainsi 3 584 octets. La raison pour laquelle la mémoire commence à partir de \$200 est que sur le VIP et Cosmac Telmac 1800, les 512 premiers octets sont réservés pour l'interpréteur. Sur ces machines, les 256 octets les plus élevés (\$F00-\$FFF sur une machine 4K) ont été réservés pour le rafraîchissement de l'écran, et les 96 octets inférieurs (\$EA0-\$EFF) ont été réservés pour la pile d'appels, à usage interne, et les variables.

#### Les registres

La Chip 8 comporte 16 registres de 8 bits dont les noms vont de V0 à VF (F = 15, encore l'hexadécimal). Le registre VF est utilisé pour toutes les retenues lors des calculs.

En plus de ces 16 registres, nous avons le registre d'adresse, nommé I, qui est de 16 bits et qui est utilisé avec plusieurs *opcodes* qui impliquent des opérations de mémoire.

#### La pile

La pile sert uniquement à stocker des adresses de retour lorsque les sous-programmes sont appelés. Les implémentations modernes doivent normalement avoir au moins 16 niveaux.

## Les compteurs

La Chip 8 est composée de deux compteurs. Ils décomptent tous les deux à 60 hertz, jusqu'à ce qu'ils atteignent 0.

**Minuterie système** : cette minuterie est destinée à la synchronisation des événements de jeux. Sa valeur peut être réglée et lue.

**Minuterie sonore** : cette minuterie est utilisée pour les effets sonores. Lorsque sa valeur est différente de zéro, un signal sonore est émis. Sa valeur peut être réglée et lue.

## Les contrôles

L'entrée est faite avec un clavier qui possède 16 touches allant de 0 à F. Les touches « 8 », « 4 », « 6 » et « 2 » sont généralement utilisées pour l'entrée directionnelle.

## Le graphique

La résolution de l'écran est de  $64 \times 32$  pixels, et la couleur est monochrome. Les dessins sont établis à l'écran uniquement par l'intermédiaire de *sprites*, qui font 8 pixels de large et avec une hauteur qui peut varier de 1 à 15 pixels. Les *sprites* sont codés en binaire. Pour une valeur de 1, le pixel correspondant est allumé et pour une valeur 0, aucune opération n'est effectuée. Si un pixel d'un *sprite* est dessiné sur un pixel de l'écran déjà allumé, alors les deux pixels sont éteints. Le registre de retenue (VF) est mis à 1 à cet effet.

## Liste des instructions

La Chip 8 possède 35 *opcodes*, qui sont tous de deux octets de long.

Ils sont énumérés ci-dessous, en hexadécimal et avec les symboles suivants :

- NNN : adresse de 12 bits ;
- NN : constante de 8 bits ;
- N : constante de 4 bits ;
- X et Y : identifiant registre de 4 bits.



**Rappel** : un *opcode* est la valeur lue à partir de la mémoire.

Opcode	Explication, description
0NNN	Appelle le programme de la RCA 1802 à l'adresse NNN. (Voir plus bas.)
00E0	Efface l'écran.
00EE	Retourne à partir d'une sous-fonction.
1NNN	Effectue un saut à l'adresse NNN.
2NNN	Exécute le sous-programme à l'adresse NNN.
3XNN	Saute l'instruction suivante si VX est égal à NN.
4XNN	Saute l'instruction suivante si VX et NN ne sont pas égaux.
5XY0	Saute l'instruction suivante si VX et VY sont égaux.
6XNN	Définit VX à NN.
7XNN	Ajoute NN à VX.
8XY0	Définit VX à la valeur de VY.
8XY1	Définit VX à VX OR VY.
8XY2	Définit VX à VX AND VY.

8XY3	Définit VX à VX XOR VY.
8XY4	Ajoute VY à VX. VF est mis à 1 quand il y a un dépassement de mémoire ( <i>carry</i> ), à 0 quand il n'y en a pas.
8XY5	VY est soustrait de VX. VF est mis à 0 quand il y a un emprunt et à 1 quand il n'y en a pas.
8XY6	Décale ( <i>shift</i> ) VX à droite de 1 bit. VF est fixé à la valeur du bit de poids faible de VX avant le décalage.
8XY7	$VX = VY - VX$ , VF est mis à 0 quand il y a un emprunt, et à 1 quand il n'y en a pas.
8XYE	Décale ( <i>shift</i> ) VX à gauche de 1 bit. VF est fixé à la valeur du bit de poids fort de VX avant le décalage.
9XY0	Saute l'instruction suivante si VX et VY ne sont pas égaux.
ANNN	Affecte NNN à I.
BNNN	Passe à l'adresse NNN + V0.
CXNN	Définit VX à un nombre aléatoire inférieur à NN.
DXYN	Dessine un <i>sprite</i> aux coordonnées (VX, VY). Le <i>sprite</i> a une largeur de 8 pixels et une hauteur en pixels N. Chaque rangée de 8 pixels est lue comme codée en binaire à partir de l'emplacement mémoire I. I ne change pas de valeur après l'exécution de cette instruction.
EX9E	Saute l'instruction suivante si la clé stockée dans VX est pressée.
EXA1	Saute l'instruction suivante si la clé stockée dans VX n'est pas pressée.
FX07	Définit VX à la valeur de la temporisation.
FX0A	L'appui sur une touche est attendu, puis stocké dans VX.
FX15	Définit la temporisation à VX.
FX18	Définit la minuterie sonore à VX.
FX1E	Ajoute VX à I. VF est mis à 1 quand il y a <i>overflow</i> ( $I+VX > 0xFF$ ), et à 0 si ce n'est pas le cas.
FX29	Définit I à l'emplacement du caractère stocké dans VX. Les caractères 0-F (en hexadécimal) sont représentés par une police 4x5.
FX33	Stocke dans la mémoire le code décimal représentant VX (dans I, I+1, I+2).
FX55	Stocke V0 à VX en mémoire à partir de l'adresse I.
FX65	Remplit V0 à VX avec les valeurs de la mémoire à partir de l'adresse I.

## La RCA 1802

0NNN	Appelle le programme de la RCA 1802 à l'adresse NNN.
------	--

Cette instruction ne nous intéresse pas. En réalité, la RCA 1802 est un microprocesseur 8 bits qui a été utilisé dans certains micro-ordinateurs et consoles de jeu tels que la RCA Studio II. Ce microprocesseur possédait un jeu d'instructions intégrées que les nouvelles implémentations ignorent. Nous allons donc faire de même. 🤔



Mais ça m'avance à quoi, tous ces détails ?

Ne paniquez pas si cette description ne vous éclaire pas, nous allons expliquer ligne par ligne tout ce qui a été dit ci-dessus. C'est grâce à ce document que nous allons programmer notre émulateur ; nous allons le traduire en langage machine. 😊

Vous pouvez lancer votre IDE, et que la programmation commence !



Pour créer l'émulateur, je donne des explications par rapport à une citation dans la présentation de la Chip 8 puis le code C/SDL pour le réaliser. Je vous conseille donc d'essayer avec les explications de créer votre propre code pour mieux comprendre. Vous pourrez éventuellement jeter un coup d'œil sur le code proposé pour vous faire une idée plus précise.

## La base

Voici la première partie liée à de la programmation pure et dure. Ouvrez grand vos yeux ! La partie de plaisir peut *enfin* commencer. Je fournirai le code pour presque toutes les actions à effectuer, mais il est inutile de préciser qu'il vaut mieux comprendre et écrire son propre code que d'effectuer des copier-coller.

*Let's gooo!*

### L'implémentation de la machine

Nous allons commencer par récupérer une citation dans la description de la Chip 8, que nous nous contenterons de traduire en langage machine.

Cette partie concernera le CPU de la Chip 8. Le CPU est l'organe central de notre émulateur : c'est le chef d'orchestre.

*Are you ready?*

## La mémoire

### Citation

Les adresses mémoire de la Chip 8 vont de \$200 à \$FFF (l'hexadécimal revient), faisant ainsi 3 584 octets. La raison pour laquelle la mémoire commence à partir de \$200 est que sur le VIP et Cosmac Telmac 1800, les 512 premiers octets sont réservés pour l'interpréteur. Sur ces machines, les 256 octets les plus élevés (\$F00-\$FFF sur une machine 4K) ont été réservés pour le rafraîchissement de l'écran, et les 96 octets inférieurs (\$EA0-\$EFF) ont été réservés pour la pile d'appels, à usage interne, et les variables.

Bien que cette citation soit assez longue, ce qui nous intéresse est : « Les adresses mémoire vont de \$200 à \$FFF, faisant ainsi 3 584 octets » et « les 512 premiers octets sont réservés ». Je rappelle que  $\$200 = 512$ .

On peut déduire de ces deux informations que la Chip 8 a une mémoire de  $3\,584 + 512 = 4\,096$  octets (un octet = huit bits, ne l'oubliez jamais). Le reste n'est que culture générale.


Et comme nous allons simuler le fonctionnement de notre machine, le rafraîchissement sera géré par une autre méthode. Il existe des fonctions dédiées pour toutes les bibliothèques graphiques (update, repaint, SDL\_Flip, etc). Les 512 premiers octets ne serviront donc à rien (pour le moment).

Dans mon cas, la variable mémoire prendra la forme d'un tableau de 4 096 octets.



La mémoire est utilisée pour charger les jeux(roms) et pour la gestion des périphériques de la machine.

J'en profite pour vous dire qu'il faut bien prendre en compte la taille spécifiée pour chaque variable. En plus, elles sont toutes non signées. En cas de non-respect de ces indications, votre programme buggera à coup sûr.

Je parle en connaissance de cause. 

Donc, à nous les unsigned dans tous les sens ! Pour ma part, j'utilise SDL, donc à moi les Uint.

Déclaration de la mémoire :

```
Uint8 memoire[4096]; // la mémoire est en octets (8 bits), soit un tableau de 4096 Uint8.
```

Maintenant, pour pointer sur une adresse donnée, il faut une autre variable qui sera initialisée à  $\$200 = 512$  comme nous le dit la description.

Nous la nommons pc comme « *program counter* ». La variable doit être de 16 bits au minimum car nous devons être en mesure de parcourir tout le tableau mémoire qui va de 0 à 4095.

## Les registres

### Citation

La Chip 8 comporte 16 registres de 8 bits dont les noms vont de V0 à VF (F = 15, encore l'hexadécimal). Le registre VF est utilisé pour toutes les retenues lors des calculs.

En plus de ces 16 registres, nous avons le registre d'adresse, nommé I, qui est de 16 bits et qui est utilisé avec plusieurs *opcodes* qui impliquent des opérations de mémoire.

Ici, il n'y a rien de compliqué, nous nous contenterons donc juste de déclarer les variables. Les registres permettent à la Chip 8 – et à tout processeur en général – de manipuler les données. Ils servent en gros d'intermédiaires entre la mémoire et l'unité de calcul, ou l'UAL (Unité Arithmétique et Logique) pour les intimes. Le processeur gagne en vitesse d'exécution en manipulant les

registres au lieu de modifier directement la mémoire.

## La pile ou *stack*

### Citation

La pile sert uniquement à stocker des adresses de retour lorsque les sous-programmes sont appelés. Les implémentations modernes doivent normalement avoir au moins 16 niveaux.

Lorsque le programme chargé dans la mémoire s'exécute, il se peut qu'il fasse des sauts d'une adresse mémoire à une autre. Pour revenir de ces sauts, il faut sauvegarder l'adresse où il se trouvait avant ce saut (*pc*) : c'est le rôle de la pile, appelée *stack* en anglais. Elle autorise seize niveaux, il nous faudra donc un tableau de seize variables pour stocker les seize dernières valeurs de *pc* ; on le nommera *saut*.

Et comme pour la mémoire, on aura besoin d'une autre variable afin de parcourir ce tableau. Cette fois-ci, le type `Uint8` fera l'affaire puisqu'on ne parcourt que seize valeurs. Je l'ai nommée *nbrsaut*.

## Les compteurs

### Citation

La Chip 8 est composée deux compteurs. Ils décomptent tous les deux à 60 hertz, jusqu'à ce qu'ils atteignent 0.

**Minuterie système** : cette minuterie est destinée à la synchronisation des événements de jeux. Sa valeur peut être réglée et lue.

**Minuterie sonore** : cette minuterie est utilisée pour les effets sonores. Lorsque sa valeur est différente de zéro, un signal sonore est émis. Sa valeur peut être réglée et lue.

La Chip 8 a besoin de deux variables pour se charger de la synchronisation et du son. Nous les appellerons respectivement *compteurJeu* et *compteurSon*.

Puisqu'elles doivent décompter à 60 hertz, il faut trouver une méthode pour les décrémenter toutes les  $1 / 60 = 0,016 = 16$  millisecondes. Les *timers* restent une bonne solution pour effectuer ce genre d'opération. En SDL, on implémente cette action avec `SDL_Delay`.

Toutes les caractéristiques de la Chip 8 seront stockées dans une structure qui représentera le CPU.

**Secret (cliquez pour afficher)**

Code : C - cpu.h

```
#ifndef CPU_H
#define CPU_H

#define TAILLEMEMOIRE 4096
#define ADRESSEDEBUT 512

typedef struct
{
    Uint8 memoire[TAILLEMEMOIRE];
    Uint8 V[16]; //le registre
    Uint16 I; //stocke une adresse mémoire ou dessinateur
    Uint16 saut[16]; //pour gérer les sauts dans « mémoire »,
16 au maximum
    Uint8 nbrsaut; //stocke le nombre de sauts effectués pour
ne pas dépasser 16
    Uint8 compteurJeu; //compteur pour la synchronisation
    Uint8 compteurSon; //compteur pour le son
    Uint16 pc; //pour parcourir le tableau « mémoire »
} CPU;

CPU cpu; //déclaration de notre CPU

void initialiserCpu() ;
void decompter() ;
```

```
#endif
```

#### Code : C - cpu.c

```
#include "cpu.h"

void initialiserCpu()
{
    //On initialise le tout

    Uint16 i=0;

    for(i=0;i<TAILLEMEMOIRE;i++) //faisable avec memset, mais je
n'aime pas cette fonction ^^
    {
        cpu.memoire[i]=0;
    }

    for(i=0;i<16;i++)
    {
        cpu.V[i]=0;
        cpu.saut[i]=0;
    }

    cpu.pc=ADRESSEDEBUT;
    cpu.nbrsaut=0;
    cpu.compteurJeu=0;
    cpu.compteurSon=0;
    cpu.I=0;
}

void decompter()
{
    if(cpu.compteurJeu>0)
        cpu.compteurJeu--;

    if(cpu.compteurSon>0)
        cpu.compteurSon--;
}
```

Maintenant, attaquons le graphique, cela nous permettra de voir rapidement les différents résultats. L'ordre d'implémentation des caractéristiques importe peu, vous pourriez commencer par le graphique ou même l'exécution des instructions si vous le vouliez (par contre, je ne vous le conseille pas). 🤔

Cet ordre nous permettra de faire des tests le plus tôt possible.

### Le graphique

Jetons un coup d'œil à la description de la Chip 8 :

#### Citation

La résolution de l'écran est de  $64 \times 32$  pixels, et la couleur est monochrome .

Pour simuler notre écran, nous allons créer un panneau divisé en  $64 \times 32$  pixels.

### Création des pixels



Un pixel est un petit carré (ou rectangle) caractérisé par son abscisse, son ordonnée et sa couleur (ici, elle sera noire ou blanche car l'écran est monochrome). Dans notre cas, j'ai choisi des pixels carrés de côté 8. Vous pouvez fixer une dimension qui vous convient.

Voici le code C/SDL qui permet de définir notre pixel. (Un vrai zéro se doit de maîtriser la SDL.) 😊

Secret (cliquez pour afficher)

Code : C - pixel.h

```
#ifndef PIXEL_H
#define PIXEL_H
#include <SDL/SDL.h>

typedef struct
{
    SDL_Rect position; //regroupe l'abscisse et l'ordonnée
    Uint8 couleur;     //comme son nom l'indique, c'est la couleur
} PIXEL;

#endif
```

Après la création de notre pixel, nous allons maintenant créer l'écran en tant que tel, qui sera constitué de 64 × 32 pixels. Nous allons donc d'abord déclarer un tableau de 64 × 32 pixels et l'écran qui les contiendra. Cet écran aura des dimensions proportionnelles au nombre de pixels et à leur largeur.

Secret (cliquez pour afficher)

Code : C - pixel.h

```
#ifndef PIXEL_H
#define PIXEL_H
#include <SDL/SDL.h>

#define NOIR 0
#define BLANC 1
#define l 64 //nombre de pixels suivant la largeur
#define L 32 //nombre de pixels suivant la longueur
#define DIMPIXEL 8 //pixel carré de côté 8
#define WIDTH l*DIMPIXEL //largeur de l'écran
#define HEIGHT L*DIMPIXEL //longueur de l'écran

typedef struct
{
    SDL_Rect position; //regroupe l'abscisse et l'ordonnée
    Uint8 couleur;     //comme son nom l'indique, c'est la couleur
} PIXEL;

SDL_Surface *ecran, *carre[2];
PIXEL pixel[l][L];

#endif
```



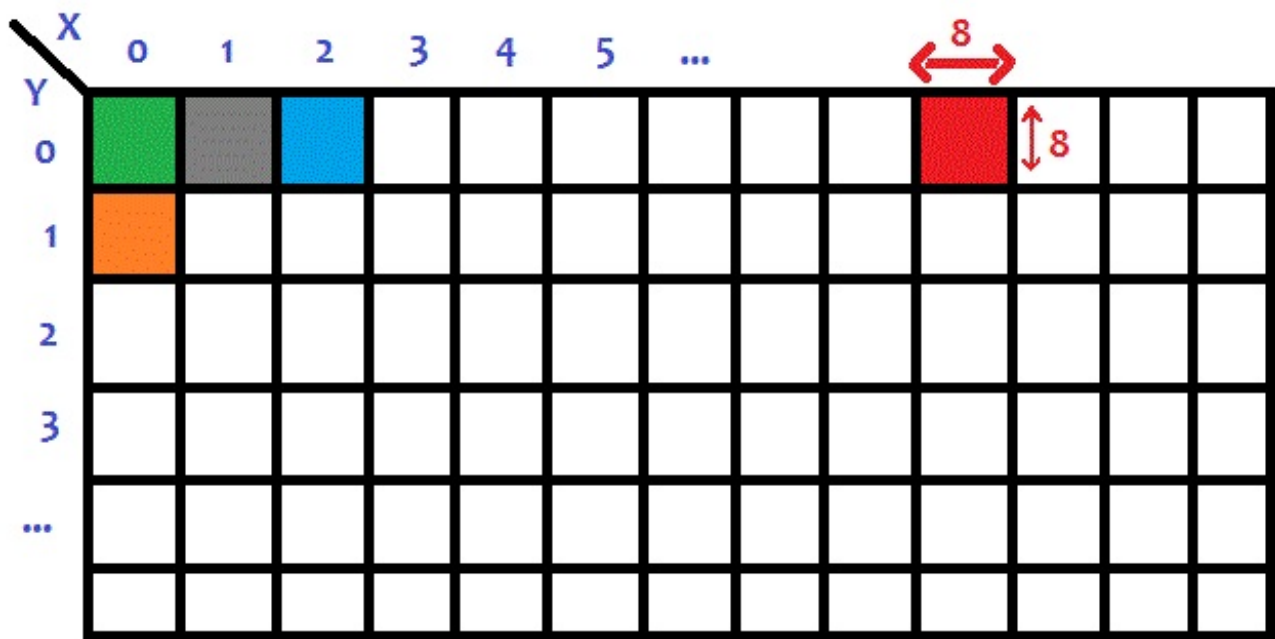
Le tableau `carre` nous permettra de définir nos deux types de pixels : Noir et Blanc. Il sera utilisé pour dessiner à l'écran (avec la SDL, c'est le tableau que l'on va *blitter* sur l'écran à différentes positions).

Maintenant que nous avons déclaré notre tableau de pixels, le premier petit problème pointe le bout de son nez.



Comment calculer les coordonnées de notre pixel à partir de l'indice du tableau ?

La technique est assez utilisée et connue mais un petit rappel est toujours le bienvenu. Jetons un coup d'œil sur notre futur panneau avec tous ses pixels.



Chaque carré représente un pixel. Le **pixel** en (0,0) a pour coordonnées (0,0). De même, le **pixel** en (2,0) a pour coordonnées (2\*8,0) soit (16,0). Enfin, le **pixel** en (0,1) a pour coordonnées (0,1\*8) soit (0,8).

D'une manière générale, pour trouver l'abscisse et l'ordonnée d'un pixel, il suffit de multiplier ses indices respectifs (X,Y) par la largeur et la longueur d'un pixel. Nous les avons fixés tous les deux à 8 (les pixels sont carrés).

Voici donc comment j'ai procédé pour le calcul :

**Secret** (cliquez pour afficher)

Code : C - pixel.h

```
#ifndef PIXEL_H
#define PIXEL_H
#include <SDL/SDL.h>

#define NOIR 0
#define BLANC 1
#define l 64 //nombre de pixels suivant la largeur
#define L 32 //nombre de pixels suivant la longueur
#define DIMPIXEL 8 //pixel carré de côté 8
#define WIDTH l*DIMPIXEL //largeur de l'écran
#define HEIGHT L*DIMPIXEL //longueur de l'écran

typedef struct
{
    SDL_Rect position; //regroupe l'abscisse et l'ordonnée
```

```

    Uint8 couleur;    //comme son nom l'indique, c'est la couleur
} PIXEL;

SDL_Surface *ecran,*carre[2];
PIXEL pixel[1][L];

void initialiserPixel() ;
#endif

```

#### Code : C - pixel.c

```

#include "pixel.h"

void initialiserPixel()
{
    Uint8 x=0,y=0;

    for(x=0;x<1;x++)
    {
        for(y=0;y<L;y++)
        {
            pixel[x][y].position.x=x*DIMPPIXEL;
            pixel[x][y].position.y=y*DIMPPIXEL;
            pixel[x][y].couleur=NOIR; //on met par défaut les pixels en noir
        }
    }
}

```

Pour la couleur des pixels, j'ai adopté le même codage que la Chip 8, à savoir :

- 0 pour le noir ou éteint ;
- 1 pour le blanc ou allumé.



Envie de faire quelques tests ?

Rajoutons des fonctions pour initialiser les variables `ecran` et `carre` et pour dessiner sur notre écran.

**Secret** (cliquez pour afficher)

#### Code : C - pixel.h

```

#ifndef PIXEL_H
#define PIXEL_H
#include <SDL/SDL.h>

#define NOIR 0
#define BLANC 1
#define l 64
#define L 32
#define DIMPPIXEL 8
#define WIDTH 1*DIMPPIXEL
#define HEIGHT L*DIMPPIXEL

typedef struct
{

```

```

        SDL_Rect position; //regroupe l'abscisse et l'ordonnée
        Uint8 couleur;     //comme son nom l'indique, c'est la couleur
    } PIXEL;

    SDL_Surface *ecran,*carre[2];
    PIXEL pixel[1][L];
    SDL_Event event; //pour gérer la pause

    void initialiserEcran() ;
    void initialiserPixel() ;
    void dessinerPixel(PIXEL pixel) ;
    void effacerEcran() ;
    void updateEcran() ;

#endif

```

### Code : C - pixel.c

```

#include "pixel.h"

void initialiserPixel()
{
    Uint8 x=0,y=0;

    for (x=0;x<L;x++)
    {
        for (y=0;y<L;y++)
        {
            pixel[x][y].position.x=x*DIMPPIXEL;
            pixel[x][y].position.y=y*DIMPPIXEL;
            pixel[x][y].couleur=NOIR;
        }
    }
}

void initialiserEcran()
{
    écran=NULL;
    carre[0]=NULL;
    carre[1]=NULL;

    écran=SDL_SetVideoMode(WIDTH,HEIGHT,32,SDL_HWSURFACE);
    SDL_WM_SetCaption("BC-Chip8 By BestCoder",NULL);

    if (écran==NULL)
    {
        fprintf(stderr,"Erreur lors du chargement du mode vidéo
%s",SDL_GetError());
        exit(EXIT_FAILURE);
    }

    carre[0]=SDL_CreateRGBSurface(SDL_HWSURFACE,DIMPPIXEL,DIMPPIXEL,32,0,0,0,0);
    //le pixel noir
    if (carre[0]==NULL)
    {
        fprintf(stderr,"Erreur lors du chargement de la surface
%s",SDL_GetError());
        exit(EXIT_FAILURE);
    }

    SDL_FillRect(carre[0],NULL,SDL_MapRGB(carre[0]-
>format,0x00,0x00,0x00)); //le pixel noir

```

```

carre[1]=SDL_CreateRGBSurface(SDL_HWSURFACE,DIMPIXEL,DIMPIXEL,32,0,0,0,0);
//le pixel blanc
if(carre[1]==NULL)
{
    fprintf(stderr,"Erreur lors du chargement de la surface
%s",SDL_GetError());
    exit(EXIT_FAILURE);
}

SDL_FillRect(carre[1],NULL,SDL_MapRGB(carre[1]-
>format,0xFF,0xFF,0xFF)); //le pixel blanc
}

void dessinerPixel(PIXEL pixel)
{
    /* pixel.couleur peut prendre deux valeurs : 0, auquel cas on dessine le
    pixel en noir, ou 1, on dessine alors le pixel en blanc */

    SDL_BlitterSurface(carre[pixel.couleur],NULL,ecran,&pixel.position);
}

void effacerEcran()
{
    //Pour effacer l'écran, on remet tous les pixels en noir

    Uint8 x=0,y=0;
    for(x=0;x<L;x++)
    {
        for(y=0;y<L;y++)
        {
            pixel[x][y].couleur=NOIR;
        }
    }

    //on repeint l'écran en noir
    SDL_FillRect(ecran,NULL,NOIR);
}

void updateEcran()
{
    //On dessine tous les pixels à l'écran
    Uint8 x=0,y=0;

    for(x=0;x<L;x++)
    {
        for(y=0;y<L;y++)
        {
            dessinerPixel(pixel[x][y]);
        }
    }

    SDL_Flip(ecran); //on affiche les modifications
}

```

**Code : C - main.c**

```

#include <SDL/SDL.h>
#include "cpu.h"

void initialiserSDL();
void quitterSDL();
void pause();

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

```

```
        initialiserSDL();
        initialiserEcran();
        initialiserPixel();

        updateEcran();

        pause();

return EXIT_SUCCESS;
}

void initialiserSDL()
{
    atexit(quitterSDL);

    if(SDL_Init(SDL_INIT_VIDEO)==-1)
    {
        fprintf(stderr, "Erreur lors de l'initialisation de la SDL
%s", SDL_GetError());
        exit(EXIT_FAILURE);
    }

}

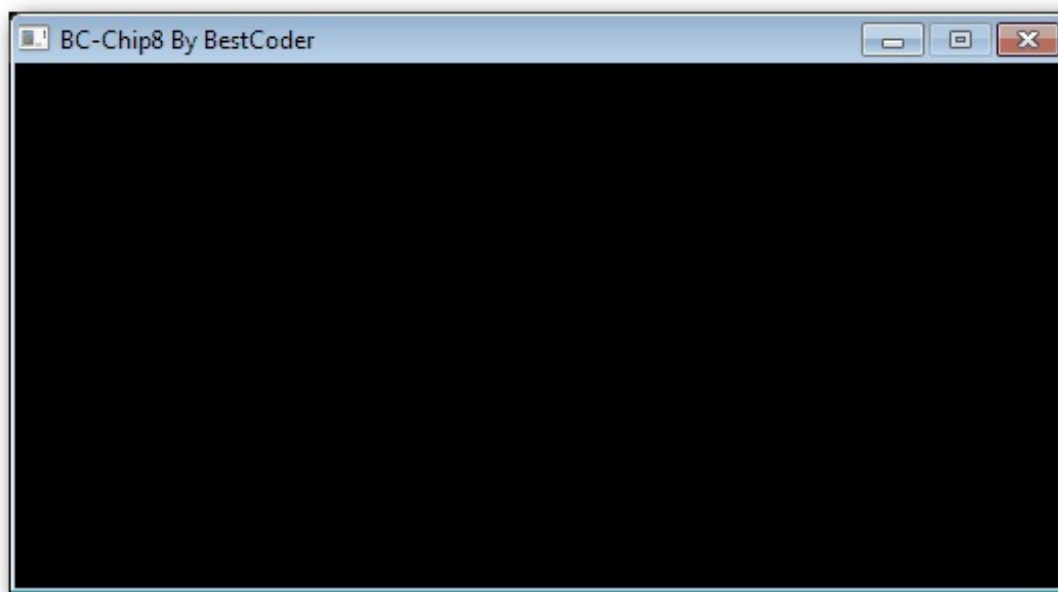
void quitterSDL()
{
    SDL_FreeSurface(carre[0]);
    SDL_FreeSurface(carre[1]);
    SDL_Quit();
}

void pause()
{
    Uint8 continuer=1;

    do
    {
        SDL_Event event;
        SDL_WaitEvent(&event);

        switch(event.type)
        {
            case SDL_QUIT:
                continuer=0;
                break;
            case SDL_KEYDOWN:
                continuer=0;
                break;
            default: break;
        }
    }while(continuer==1);
}
```

Et voilà le résultat de tout ce travail. 🤖



🤔 Quoi ? Tout ce travail pour cette merde ! Un simple `SDL_FillRect` aurait fait l'affaire.

## Modifier l'écran

Derrière tout ce travail se cache un grand secret. Je vous donne ce bout de code qui va vous éclaircir les idées. Remplacez la fonction `initialiserPixel()` par celle-ci :

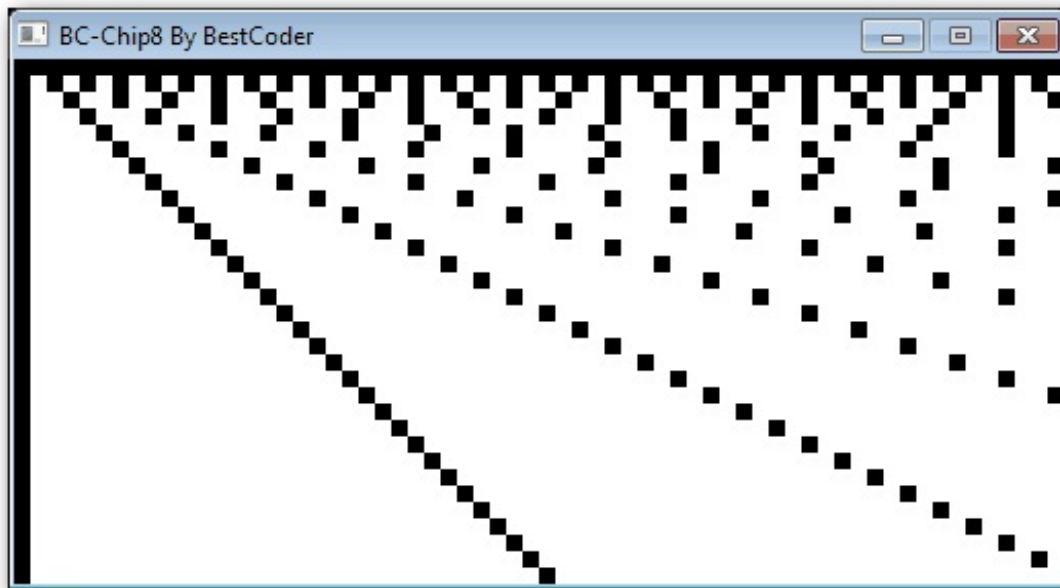
Code : C

```
void initialiserPixel()
{
    Uint8 x=0,y=0;
    for(x=0;x<L;x++)
    {
        for(y=0;y<L;y++)
        {
            pixel[x][y].position.x=x*DIMPPIXEL;
            pixel[x][y].position.y=y*DIMPPIXEL;

            if(x%(y+1)==0)
                pixel[x][y].couleur=NOIR;
            else
                pixel[x][y].couleur=BLANC;
        }
    }
}
```

Secret (cliquez pour afficher)

Et voilà le travail. 😎



Vous pouvez même changer la condition pour voir ce que donne le résultat. C'est tout simplement *AMAZING!*

C'est comme ça que le jeu se dessinera à l'écran. Il n'y aura pas de fichier image à charger ni quoi que ce soit ! Tout se fera en positionnant les pixels noirs et blancs comme il faut et en effectuant les différentes instructions requises. Nous les aborderons dans la partie suivante.

En revanche, n'oubliez pas de rétablir l'ancienne fonction `initialiserPixel()`. 🤔

Nous avons fini de remplacer le matériel utilisé, il suffit maintenant de simuler les différents calculs que peut effectuer la Chip 8 et notre émulateur sera fini et opérationnel. Je sais : vous vous dites que c'est trop facile pour être vrai, mais c'est comme ça.

La suite... au prooochain numérooo !



## Simulation des instructions

La Chip 8 ne sait exécuter que 35 opérations. 🤖 Si, si, vous avez bien entendu, ce sera aux programmeurs des jeux et applications de combiner les différentes opérations pour arriver à leurs fins. Nous nous contenterons d'implémenter les 35 opérations et notre émulateur sera opérationnel. C'est parti, nous abordons le dernier virage.

### Le cadencement du CPU et les FPS

Il y a deux choses qu'il faut absolument connaître quand on veut programmer un émulateur :

- la vitesse d'exécution des instructions ou le cadencement de la machine ;
- la période de rafraîchissement de l'écran ou le nombre de FPS.

### Comment allons-nous procéder ?

Nous savons qu'une console de jeu est un système embarqué et ceux qui ont déjà programmé pour de l'embarqué (microprocesseur, microcontrôleur, etc.) savent que dans le code, il faut systématiquement une boucle infinie ou un système de timer car le code doit s'exécuter aussi longtemps que la console est en marche.

Code : C

```
while("Machine Fonctionne")
{
    regarder_ce_qu_il_faut_faire();
    Le_faire();
}
```

La fréquence d'exécution des instructions reste assez méconnue. Dans notre cas, nous allons utiliser une fréquence de 250 hertz. Cette fréquence nous donne une période de 4 ms (rappel mathématique :  $f=1/T$  avec T la période en secondes et f la fréquence en hertz).

Donc toutes les quatre millisecondes, nous devons effectuer une opération. Soit quatre opérations, toutes les seize millisecondes.

Pour les FPS aussi, la valeur est assez méconnue (🤖), on prendra donc la valeur la plus courante, à savoir 60. Soixante images par seconde, c'est une image toutes les 16,67 ms ( $1000 / 60 = 16,67$ ). Un *timer* SDL aussi précis n'existe pas, on se limitera donc à une image toutes les 16 ms.

### Récapitulatif

Il faut quatre opérations et une nouvelle image toutes les seize millisecondes. Donc la fonction `updateEcran` sera appelée après l'exécution de quatre opérations. Je gère le tout dans le *main*. Nous allons nous contenter de la boucle principale (le fameux `while(continuer)`) et des `SDL_Delay`.

Secret (cliquez pour afficher)

Code : C

```
#include <SDL/SDL.h>
#include "cpu.h"

#define VITESSECPU 4 //nombre d'opérations par tour
#define FPS 16 //pour le rafraîchissement

void initialiserSDL() ;
void quitterSDL() ;
void pause() ;

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

```
        initialiserSDL() ;
        initialiserEcran() ;
        initialiserPixel() ;

        Uint8 continuer=1;

    do
    {
        //On effectuera quatre opérations ici

        updateEcran() ;
        SDL_Delay(FPS) ; //une pause de 16 ms
    }while(continuer==1);

    pause();

    return EXIT_SUCCESS;
}

void initialiserSDL()
{
    atexit(quitterSDL) ;

    if(SDL_Init(SDL_INIT_VIDEO)==-1)
    {
        fprintf(stderr, "Erreur lors de l'initialisation de la SDL
%s", SDL_GetError());
        exit(EXIT_FAILURE);
    }
}

void quitterSDL()
{
    SDL_FreeSurface(carre[0]) ;
    SDL_FreeSurface(carre[1]) ;
    SDL_Quit() ;
}

void pause()
{
    Uint8 continuer=1 ;

    do
    {
        SDL_WaitEvent(&event) ;

        switch(event.type)
        {
            case SDL_QUIT:
                continuer=0;
                break;
            case SDL_KEYDOWN:
                continuer=0;
                break;
            default: break ;
        }
    }while(continuer==1);
}
```



Ne testez surtout pas ce code. 😊



Le choix de 250 Hz et 60 FPS est subjectif. Le problème est que les caractéristiques de la Chip 8 sont très méconnues, mais pour beaucoup d'autres consoles, vous trouverez facilement la fréquence exacte de cadencement du CPU. Et 250 Hz, c'est relativement facile à simuler. (Je sais, je suis paresseux. 😊)

## Lecture de l'opcode

Pour connaître l'action à effectuer, il faut lire dans la mémoire. Mais le problème est qu'elle est en octets (8 bits) et que les *opcodes* sont, eux, de 16 bits.

Par exemple, pour l'*opcode* 8XY0 – qui, je le rappelle, est en hexadécimal – on a : 4 bits pour le « 0 » + 4 bits pour le « X » + 4 bits pour le « Y » + 4 bits pour le « 0 ». Ce qui nous donne  $4 \times 4 = 16$  bits.

Si on effectue `opcode=cpu.memoire[cpu.pc]`, il nous manquerait 8 bits. Il faut alors récupérer 16 bits dans la mémoire, à savoir `cpu.memoire[cpu.pc]` et `cpu.memoire[cpu.pc+1]`.

Maintenant, il faut juste trouver un moyen pour les mettre « côte à côte » pour ne former qu'un unique nombre de 16 bits.

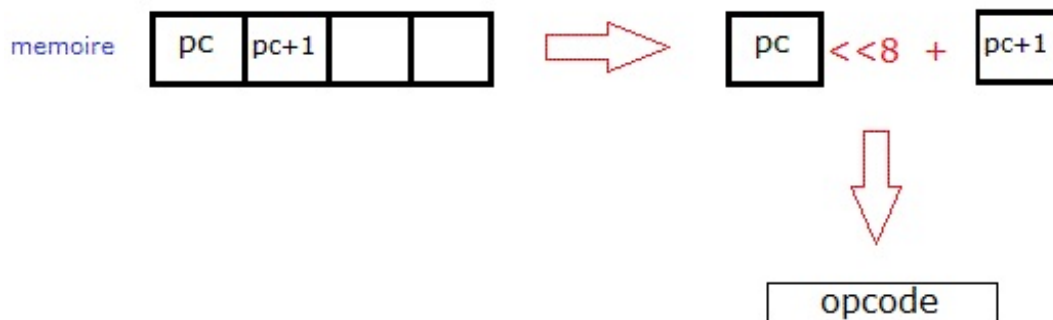


Les décalages (*bit shift*), ça vous dit quelque chose ?

Eh bien, ce sont eux qui vont nous faciliter la vie. Il suffit de décaler vers la gauche `cpu.memoire[cpu.pc]` de 8 bits et de faire la somme avec `cpu.memoire[cpu.pc+1]`. On aura ainsi un *opcode* de 16 bits.

Donc, on aura en définitive `opcode=cpu.memoire[cpu.pc]<<8+cpu.memoire[cpu.pc+1]`.

copyright BestCoder



La fonction pour effectuer ce calcul sera introduite dans `cpu.h` et `cpu.c`.

Code : C

```

Uint16 recupererOpcode() ; //prototype dans cpu.h

Uint16 recupererOpcode() //dans cpu.c
{
    return (cpu.memoire[cpu.pc]<<8)+cpu.memoire[cpu.pc+1];
}

```

On vient de passer une étape, mais il reste encore deux ou trois détails et le tour sera joué.

## Identification de l'instruction

Après avoir récupéré notre *opcode*, il ne reste plus qu'à l'interpréter. Par interpréter, il faut comprendre effectuer l'opération qui lui est associée.

Jetons un coup d'œil à nos *opcodes* (qui sont en hexadécimal) : 0NNN, 00E0, 00EE, 1NNN, 2NNN, 3XNN, 4XNN, 5XY0, 6XNN, 7XNN, 8XY0, 8XY1, 8XY2, 8XY3, 8XY4, 8XY5, etc. (ces valeurs proviennent du tableau de la partie [Quelle machine émuler ?](#)).

Tous les X, Y et N sont supposés inconnus. Pour connaître l'action à exécuter, il faut donc trouver un moyen d'identifier chaque

*opcode* en ne tenant pas compte de ces valeurs. Pour ce faire, nous allons utiliser les opérations bits à bits combinées à une grosse amélioration de [Pouet\\_forever](#).

## Notre table de correspondance

Nous avons 35 opérations à effectuer. Pour chacune d'elles, j'ai donc associé un nombre unique compris entre 0 et 34. Ensuite, suivant le nombre obtenu, on effectuera l'opération souhaitée en utilisant un bloc **switch**.

### Trouver le masque et l'identifiant de l'opcode

Prenons l'*opcode* 0x8XY2 comme exemple. Pour l'identifier, on doit vérifier que les 4 bits de poids fort donnent 8 et les 4 bits de poids faible donnent 2.

Pour ce faire, on peut effectuer l'opération 0x8XY2 & 0xF00F qui nous donne 8002. À chaque fois que l'on effectue *opcode\_quelconque* & 0xF00F et qu'on trouve 8002, il s'agit donc de 8XY2. Ingénieurs, n'est-ce pas ? (Honte à moi, grand copieur de Pouet\_Forever. 🙄) L'*opcode* 8XY2 a donc pour masque 0xF00F et pour identifiant 0x8002.

Pour tous les autres *opcodes*, le principe reste le même. Exemples :

Opcode	Masque	Identifiant
00E0	FFFF	00E0
1NNN	F000	1000
8XY3	F00F	8003
FX15	F0FF	F015

Je stocke le tout dans une structure que j'ai appelée JUMP.

#### Code : C

```
//Dans cpu.h
#define NBROPCODE 35

typedef struct
{
    Uint16 masque [NBROPCODE]; //la Chip 8 peut effectuer 35
    //opérations, chaque opération possédant son masque
    Uint16 id[NBROPCODE]; //idem, chaque opération possède son
    //propre identifiant
} JUMP;

JUMP jp;

void initialiserJump () ;

//Dans cpu.c

void initialiserJump ()
{
    jp.masque[0]= 0x0000; jp.id[0]=0x0FFF; /* 0NNN */
    jp.masque[1]= 0xFFFF; jp.id[1]=0x00E0; /* 00E0 */
    jp.masque[2]= 0xFFFF; jp.id[2]=0x00EE; /* 00EE */
    jp.masque[3]= 0xF000; jp.id[3]=0x1000; /* 1NNN */
    jp.masque[4]= 0xF000; jp.id[4]=0x2000; /* 2NNN */
    jp.masque[5]= 0xF000; jp.id[5]=0x3000; /* 3XNN */
    jp.masque[6]= 0xF000; jp.id[6]=0x4000; /* 4XNN */
    jp.masque[7]= 0xF00F; jp.id[7]=0x5000; /* 5XY0 */
    jp.masque[8]= 0xF000; jp.id[8]=0x6000; /* 6XNN */
}
```

```

jp.masque[9]= 0xF000; jp.id[9]=0x7000;      /* 7XNN */
jp.masque[10]= 0xF00F; jp.id[10]=0x8000;    /* 8XY0 */
jp.masque[11]= 0xF00F; jp.id[11]=0x8001;    /* 8XY1 */
jp.masque[12]= 0xF00F; jp.id[12]=0x8002;    /* 8XY2 */
jp.masque[13]= 0xF00F; jp.id[13]=0x8003;    /* 8XY3 */
jp.masque[14]= 0xF00F; jp.id[14]=0x8004;    /* 8XY4 */
jp.masque[15]= 0xF00F; jp.id[15]=0x8005;    /* 8XY5 */
jp.masque[16]= 0xF00F; jp.id[16]=0x8006;    /* 8XY6 */
jp.masque[17]= 0xF00F; jp.id[17]=0x8007;    /* 8XY7 */
jp.masque[18]= 0xF00F; jp.id[18]=0x800E;    /* 8XYE */
jp.masque[19]= 0xF00F; jp.id[19]=0x9000;    /* 9XY0 */
jp.masque[20]= 0xF000; jp.id[20]=0xA000;    /* ANNN */
jp.masque[21]= 0xF000; jp.id[21]=0xB000;    /* BNNN */
jp.masque[22]= 0xF000; jp.id[22]=0xC000;    /* CXNN */
jp.masque[23]= 0xF000; jp.id[23]=0xD000;    /* DXYN */
jp.masque[24]= 0xF0FF; jp.id[24]=0xE09E;    /* EX9E */
jp.masque[25]= 0xF0FF; jp.id[25]=0xE0A1;    /* EXA1 */
jp.masque[26]= 0xF0FF; jp.id[26]=0xF007;    /* FX07 */
jp.masque[27]= 0xF0FF; jp.id[27]=0xF00A;    /* FX0A */
jp.masque[28]= 0xF0FF; jp.id[28]=0xF015;    /* FX15 */
jp.masque[29]= 0xF0FF; jp.id[29]=0xF018;    /* FX18 */
jp.masque[30]= 0xF0FF; jp.id[30]=0xF01E;    /* FX1E */
jp.masque[31]= 0xF0FF; jp.id[31]=0xF029;    /* FX29 */
jp.masque[32]= 0xF0FF; jp.id[32]=0xF033;    /* FX33 */
jp.masque[33]= 0xF0FF; jp.id[33]=0xF055;    /* FX55 */
jp.masque[34]= 0xF0FF; jp.id[34]=0xF065;    /* FX65 */
}

```

### Trouver l'opcode à interpréter

Le plus difficile est fait, il ne reste plus qu'à implémenter un algorithme nous permettant de retrouver le nombre associé à un *opcode*. Pour chaque *opcode*, il faut récupérer son identifiant en appliquant un `&` avec le masque et le comparer avec ceux de notre structure JUMP. Un exemple vaut mieux que mille discours.

Prenons le nombre `0x8475`. Grâce à notre structure JUMP, nous devons être en mesure de retrouver 15, qui est le nombre associé aux *opcodes* `0x8XY5`.



Comment ?

Il faut parcourir la structure JUMP pour trouver à quel indice *i* la condition `0x8475 & jp.masque[i] == jp.id[i]` est vraie.

Pour ce cas-ci, *i* vaut 15, on a donc `0x8475 & jp.masque[15] == jp.id[15]`, soit `0x8475 & 0xF00F == 0x8005`, ce qui est vrai. Pour toutes les autres valeurs de *i*, cette condition sera toujours fausse. Vérifiez par vous-mêmes pour voir. 😊

Voici le code de cet algorithme :

**Code : C**

```

//Dans cpu.h
uint8 recupererAction(uint16) ;

//Dans cpu.c

uint8 recupererAction(uint16 opcode)
{
    uint8 action;
    uint16 resultat;

    for(action=0; action<NBROPCODE; action++)
    {
        resultat= (jp.masque[action]&opcode); /* On récupère les

```

```

bits concernés par le test, l'identifiant de l'opcode */

    if(resultat == jp.id[action]) /* On a trouvé l'action à
effectuer */
        break; /* Plus la peine de continuer la boucle car la
condition n'est vraie qu'une seule fois*/
    }

    return action; //on renvoie l'indice de l'action à effectuer
}

```



Cas spécial : si vous regardez le masque et l'identifiant de l'opcode 0NNN, vous verrez que `opcode&0x0000` – qui est toujours égal à 0x0000 – est toujours différent de 0xFFFF. En gros, on ne pourra jamais retrouver `action=0`. Comme je l'avais dit dans la partie de présentation de la Chip 8, « 0NNN appelle le programme de la RCA 1802 à l'adresse NNN. », cela ne nous intéresse donc pas.

À présent, pour simuler une instruction, il suffit de placer notre bloc **switch**.

Code : C

```

//Dans cpu.h

void interpreterOpcode(Uint16) ;

//Dans cpu.c

void interpreterOpcode(Uint16 opcode)
{
    Uint8 b4;

    b4= recupererAction(opcode); //permet de connaître l'action à
effectuer

    switch(b4)
    {
        case 0:{
            //Cet opcode n'est pas implémenté
            break;
        }
        case 1:{
            //00E0 : efface l'écran
            break;
        }
        case 2:{//00EE : revient du saut

            break;
        }
        case 3:{ //1NNN : effectue un saut à l'adresse 1NNN
            break;
        }
        case 4:{
            //2NNN : appelle le sous-programme en NNN, mais on
            revient ensuite
            break;
        }

        // etc. jusqu'à 34
    }
}

```



Il n'y a aucune condition à poser sur X, Y, N, NN et NNN, ces valeurs seront utilisées pour réaliser l'instruction souhaitée.

Par exemple, pour l'opcode 0x8XY2, on aura :

8XY2	Définit VX à VX AND VY
------	------------------------

Le code pour le réaliser est le suivant : il faut récupérer X et Y et effectuer  
`V[X]=V[X]&V[Y];` //C'est trop simple, non ?!.

Pour récupérer les valeurs de X, Y, NN et NNN, il faut prendre les 12 bits de poids faible et les associer si l'opération en a besoin.

#### Code : C

```
//Ce code sera placé dans la fonction interpreterOpcode
Uint8 b3,b2,b1;

X b3=(opcode&(0x0F00))>>8; //on prend les 4 bits, b3 représente
b2=(opcode&(0x00F0))>>4; //idem, b2 représente Y
b1=(opcode&(0x000F)); //on prend les 4 bits de poids faible

/*
Pour obtenir NNN par exemple, il faut faire (b3<<8) + (b2<<4) + (b1)
*/
```

Passons maintenant au clou du spectacle : le graphique.

### Retour sur le graphique

Mesdames, messieurs, veuillez attacher vos ceintures : nous allons entrer dans une zone de turbulences. 🐼

Pour dessiner à l'écran, la Chip 8 dispose d'un unique *opcode* (une seule instruction permet de dessiner à l'écran).

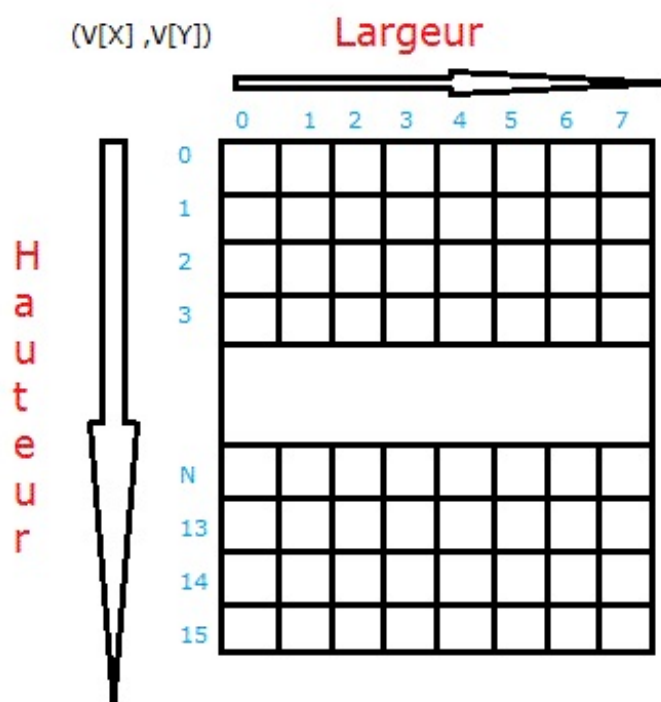
#### Citation

DXYN	Dessine un <i>sprite</i> aux coordonnées (VX, VY). Le <i>sprite</i> a une largeur de 8 pixels et une hauteur de pixels N. Chaque rangée de 8 pixels est lue comme codée en binaire à partir de l'emplacement mémoire. Il ne change pas de valeur après l'exécution de cette instruction.
------	---

#### Citation

Les dessins sont établis à l'écran uniquement par l'intermédiaire de *sprites*, qui font 8 pixels de large et avec une hauteur qui peut varier de 1 à 15 pixels. Les *sprites* sont codés en binaire. Pour une valeur de 1, le pixel correspondant est allumé et pour une valeur 0, aucune opération n'est effectuée. Si un pixel d'un *sprite* est dessiné sur un pixel de l'écran déjà allumé, alors les deux pixels sont éteints. Le registre de retenue (VF) est mis à 1 à cet effet.

copyright BestCoder



Comme vous le voyez sur l'image, chaque *sprite* peut être considéré comme un tableau à deux dimensions. Pour parcourir tout le *sprite*, il faudra donc deux boucles imbriquées.

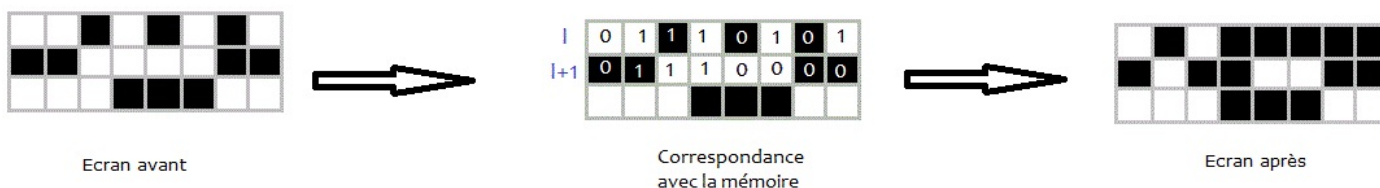
Pour le codage des lignes, on récupère les valeurs dans la mémoire en commençant à l'adresse I. Si par exemple, on doit dessiner un *sprite* en (0,0) avec une hauteur de 3 et le codage `memoire[I]=11010101`, `memoire[I+1]=00111100` et `memoire[I+2]=11100011` //ces nombres sont en binaire, nous devons obtenir :

copyright BestCoder



Ensuite, si l'on souhaite dessiner un autre *sprite* en (0,0) avec une hauteur de 2 et le codage `memoire[I]=01110101`, `memoire[I+1]=01110000`, nous devons obtenir :

copyright BestCoder



Pour réaliser tout cela, il faut donc récupérer la couleur du pixel à dessiner, la comparer avec son ancienne valeur et agir en conséquence. On gardera en tête que pour « 0 », la couleur désirée est le noir, et pour « 1 », le blanc. Voici le code C/SDL qui permet de réaliser tout ce bazar.



**Secret (cliquez pour afficher)****Code : C**

```

void dessinerEcran(UInt8 b1, UInt8 b2, UInt8 b3)
{
    UInt8 x=0, y=0, k=0, codage=0, j=0, decalage=0;
    cpu.V[0xF]=0;

    for(k=0; k<b1; k++)
    {
        codage=cpu.memoire[cpu.I+k]; //on récupère le codage
        de la ligne à dessiner

        y=(cpu.V[b2]+k)%L; //on calcule l'ordonnée de la ligne
        à dessiner, on ne doit pas dépasser L

        for(j=0, decalage=7; j<8; j++, decalage--)
        {
            x=(cpu.V[b3]+j)%l; //on calcule l'abscisse, on ne
            doit pas dépasser l

            if(((codage)&(0x1<<decalage))!=0) //on
            récupère le bit correspondant
            {
                //si c'est blanc
                if(pixel[x][y].couleur==BLANC) //le
                pixel était blanc

                {
                    pixel[x][y].couleur=NOIR; //on
                    l'éteint
                    cpu.V[0xF]=1; //il y a donc
                    collusion

                }
                else //sinon
                {
                    pixel[x][y].couleur=BLANC; //on
                    l'allume

                }
            }
        }
    }
}

```

**Ligne importante**

$(\text{codage}) \& (0x1 \ll \text{decalage})$

Les décalages pointent encore le bout de leur nez. 🤪

Tout d'abord, il faut savoir que  $0x1 = 00000001$  en binaire sur 8 bits. L'instruction  $0x1 \ll \text{decalage}$  permet de placer le « 1 » à l'endroit correspondant au codage de notre pixel.

Si par exemple, nous voulons dessiner le troisième pixel de la ligne,  $j$  vaut 2 (l'indice commence par 0) et  $\text{decalage}$  vaut 5. Donc  $0x1 \ll \text{decalage} = 00100000$  en binaire sur 8 bits. Le « 1 » se place au troisième rang (il faut compter en partant de la gauche).

Donc, pour récupérer le bit correspondant à notre pixel, il suffit d'appliquer le `and` ou `&` et le tour est joué.

Je vous donne en prime tout le bloc **switch** de notre émulateur. Je vous préviens, il fait peur ce bloc. 😬

**Secret** (cliquez pour afficher)

Code : C

```
void interpreter(Uint16 opcode)
{
    Uint8 b4,b3,b2,b1;

    b3=(opcode&(0x0F00))>>8;  //on prend les 4 bits représentant
X    b2=(opcode&(0x00F0))>>4;  //idem pour Y
    b1=(opcode&(0x000F));      //les 4 bits de poids faible

    b4= recupererAction(opcode);

    switch (b4)
    {
        case 0:{
            //Cet opcode n'est pas implémenté.
            break;
        }
        case 1:{
            //00E0 efface l'écran.
            break;
        }
        case 2:{//00EE revient du saut.

            break;
        }
        case 3:{ //1NNN effectue un saut à l'adresse 1NNN.

            break;
        }
        case 4:{
            //2NNN appelle le sous-programme en NNN, mais on
            revient ensuite.

            break;
        }
        case 5:{//3XNN saute l'instruction suivante si VX est égal à
NN.

            break;
        }
        case 6:{//4XNN saute l'instruction suivante si VX et NN ne
sont pas égaux.

            break;
        }
        case 7:{
            //5XY0 saute l'instruction suivante si VX et VY sont
égaux.

            break;
        }
        case 8:{
            //6XNN définit VX à NN.
```

```

        break;
    }
    case 9:{
        //7XNN ajoute NN à VX.

        break;
    }
    case 10:{
        //8XY0 définit VX à la valeur de VY.

        break;
    }
    case 11:{
        //8XY1 définit VX à VX OR VY.

        break;
    }
    case 12:{
        //8XY2 définit VX à VX AND VY.

        break;
    }
    case 13:{
        //8XY3 définit VX à VX XOR VY.

        break;
    }
    case 14:{
        //8XY4 ajoute VY à VX. VF est mis à 1 quand il y
        a un dépassement de mémoire (carry), et à 0 quand il n'y en pas.

        break;
    }
    case 15:{
        //8XY5 VY est soustraite de VX. VF est mis à 0
        quand il y a un emprunt, et à 1 quand il n'y a en pas.

        break;
    }
    case 16:{
        //8XY6 décale (shift) VX à droite de 1 bit. VF
        est fixé à la valeur du bit de poids faible de VX avant le
        décalage.

        break;
    }
    case 17:{
        //8XY7 VX = VY - VX. VF est mis à 0 quand il y a
        un emprunt et à 1 quand il n'y en a pas.

        break;
    }
    case 18:{
        //8XYE décale (shift) VX à gauche de 1 bit. VF
        est fixé à la valeur du bit de poids fort de VX avant le
        décalage.

        break;
    }
    case 19:{

        //9XY0 saute l'instruction suivante si VX et VY

```

```
ne sont pas égaux.

        break;
    }
    case 20:{
        //ANNN affecte NNN à I.

        break;
    }
    case 21:{
        //BNNN passe à l'adresse NNN + V0.

        break;
    }
    case 22:{
        //CXNN définit VX à un nombre aléatoire inférieur à
NN.

        break;
    }
    case 23:{
        //DXYN dessine un sprite aux coordonnées (VX, VY).

        dessinerEcran(b1,b2,b3);

        break;
    }
    case 24:{
        //EX9E saute l'instruction suivante si la clé
stockée dans VX est pressée.

        break;
    }
    case 25:{
        //EXA1 saute l'instruction suivante si la clé stockée
dans VX n'est pas pressée.

        break;
    }
    case 26:{
        //FX07 définit VX à la valeur de la
temporisation.

        break;
    }
    case 27:{
        //FX0A attend l'appui sur une touche et la stocke
ensuite dans VX.

        break;
    }
    case 28:{
        //FX15 définit la temporisation à VX.
```

```
        break;
    }
    case 29:{
        //FX18 définit la minuterie sonore à VX.

        break;
    }
    case 30:{
        //FX1E ajoute à VX I. VF est mis à 1 quand il y a
        overflow (I+VX>0xFFFF), et à 0 si tel n'est pas le cas.

        break;
    }

    case 31:{
        //FX29 définit I à l'emplacement du caractère
        stocké dans VX. Les caractères 0-F (en hexadécimal) sont
        représentés par une police 4x5.

        break;
    }

    case 32:{
        //FX33 stocke dans la mémoire le code décimal
        représentant VX (dans I, I+1, I+2).

        break;
    }
    case 33:{
        //FX55 stocke V0 à VX en mémoire à partir de
        l'adresse I.

        break;
    }
    case 34:{
        //FX65 remplit V0 à VX avec les valeurs de la
        mémoire à partir de l'adresse I.

        break;
    }

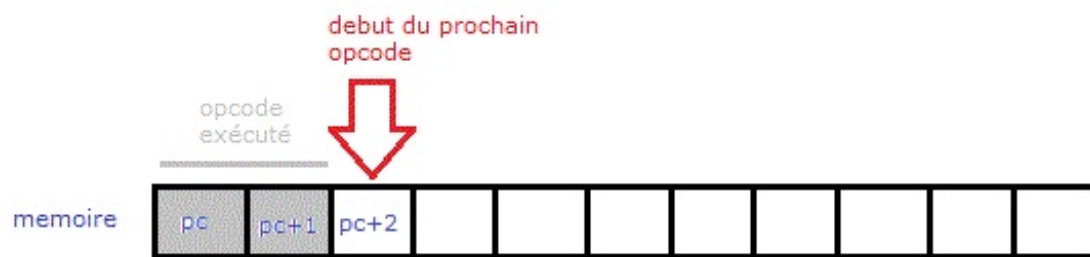
    default: { //si ça arrive, il y un truc qui cloche

        break;
    }
}
cpu.pc+=2; //on passe au prochain opcode
}
```



Je vous fais remarquer le `pc+=2` à la fin du bloc **switch**. Après avoir exécuté l'*opcode*, il faut passer au suivant en incrémentant `pc` de 2 et non pas de 1. En effet, dans la fonction de récupération de l'*opcode*, on prend `memoire[pc]` et `memoire[pc+1]`

copyright BestCoder



Le gros du travail vient d'être effectué. Il ne nous reste plus qu'à remplir les cases vides de notre **switch** avec les instructions qu'il faut, puis le tour est joué. Courage, on entrevoit le bout du tunnel ! 😊

## Exemples avec quelques instructions

Maintenant que le terrain a été bien préparé, il ne nous reste plus qu'à simuler les instructions de calcul une par une. Si vous avez suivi jusque-là, vous n'aurez aucun souci puisque cette partie est la plus facile.



Pour effectuer les tests, vous n'avez pas besoin d'implémenter les *opcodes* qui traitent les entrées utilisateur et le son. On les verra plus tard.

### Divers

00E0

#### Citation

00E0	Efface l'écran.
------	-----------------

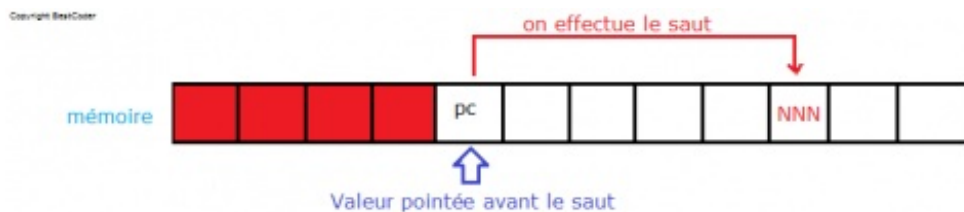
Pour effacer l'écran, il suffit de mettre tous les pixels en noir. Et comme nous avons déjà défini une méthode pour modifier les pixels, il faudra juste l'appeler dans la bonne case du **switch**.

1NNN

#### Citation

1NNN	Effectue un saut à l'adresse NNN.
------	-----------------------------------

La variable qui permet de pointer sur une adresse est la *program counter* : `pc`. Il faudra l'affecter de la valeur `NNN-2`. Le « -2 » vient du fait qu'il faut aussi prendre en compte l'incrément de `pc` à la fin du bloc **switch**.



#### Code : C

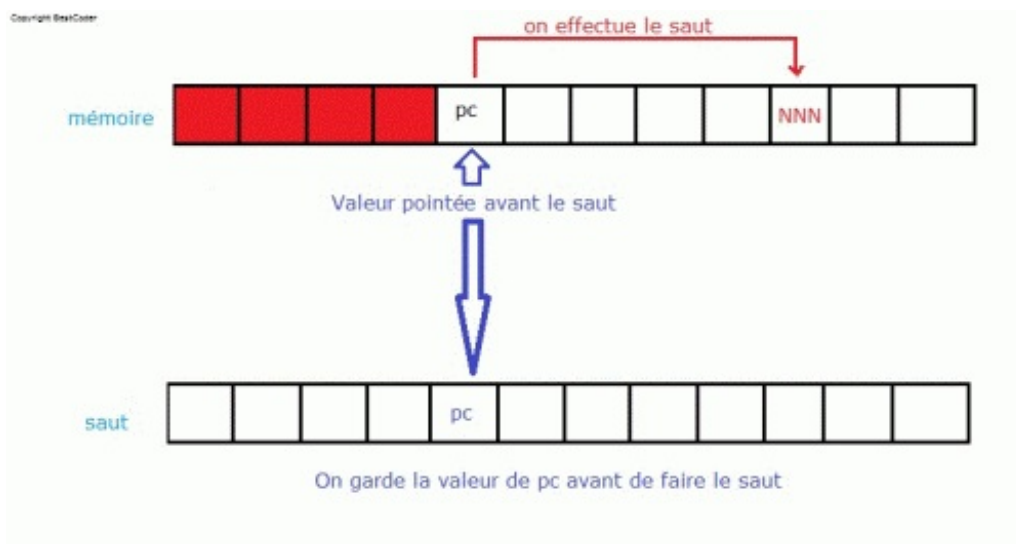
```
cpu.pc = (b3 << 8) + (b2 << 4) + b1; // on prend le nombre NNN (pour le saut)
cpu.pc -= 2; // n'oublions pas le pc += 2 à la fin du bloc switch
```

2NNN

#### Citation

2NNN	Exécute le sous-programme à l'adresse NNN.
------	--

Cette instruction est proche de la 1NNN. Mais dans ce cas-ci, il faudra récupérer l'ancienne valeur de `pc` afin d'y revenir après. La variable `saut` sera utilisée.



## Code : C

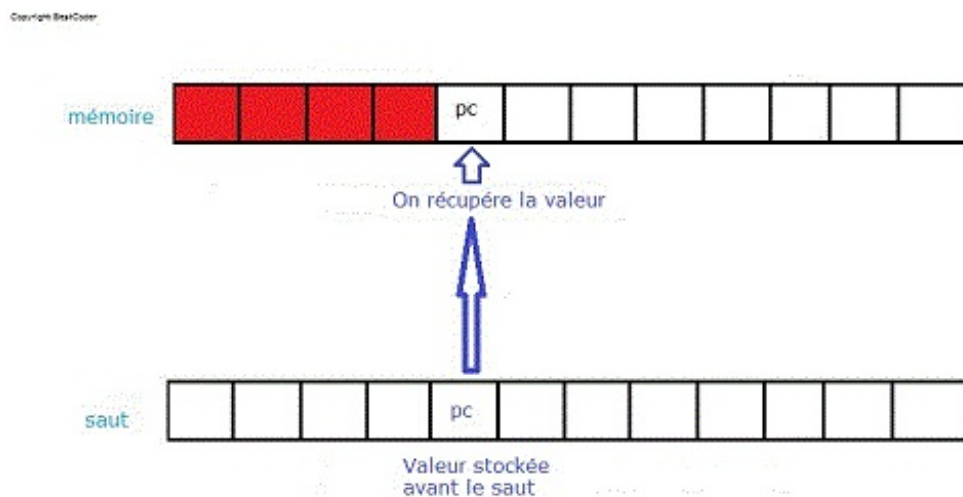
```
cpu.saut[cpu.nbrsaut]=cpu.pc; //on reste là où on était
if(cpu.nbrsaut<15)
{
    cpu.nbrsaut++;
}
//sinon, on a effectué trop de sauts
cpu.pc=(b3<<8)+(b2<<4)+b1; //on prend le nombre NNN (pour le saut)
cpu.pc-=2; //n'oublions pas le pc+=2 à la fin du block switch
```

## 00EE

## Citation

00EE	Retourne à partir d'un sous-programme.
------	--

La variable `pc` reçoit son ancienne valeur stockée dans `saut`.





**Code : C**

```

if(cpu.nbrsaut>0)
{
    cpu.nbrsaut--;
    cpu.pc=cpu.saut[cpu.nbrsaut];
}
//sinon, on a effectué plus de retours que de sauts

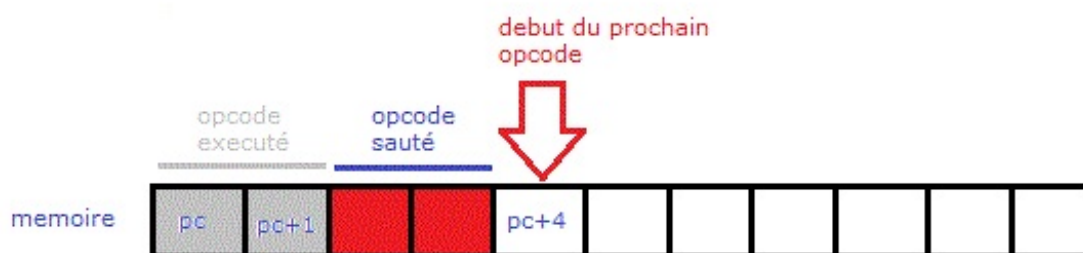
```

**3XNN****Citation**

3XNN	Saute l'instruction suivante si VX est égal à NN.
------	---

Pour sauter une instruction, il faut juste incrémenter `pc` de 2 dans la condition « VX est égal à NN ».  
Avec l'incrémentation de `pc` à la fin du bloc **switch**, on se retrouvera à `pc + 4`.

copyright BestCoder

**Code : C**

```

if(cpu.V[b3]==(b2<<4)+b1))
{
    cpu.pc+=2;
}

```

**8XY0****Citation**

8XY0	Définit VX à la valeur de VY.
------	-------------------------------

VX reçoit VY.

**Code : C**

```
cpu.V[b3]=cpu.V[b2];
```

## CXNN

### Citation

CXNN	Définit VX à un nombre aléatoire inférieur à NN.
------	--

Avec  $(\text{nombre\_aleatoire}) \% (\text{NN}+1)$ , le résultat ne pourra jamais dépasser NN. 🧙

Je vous épargne la démonstration que vous connaissez sûrement.

### Code : C

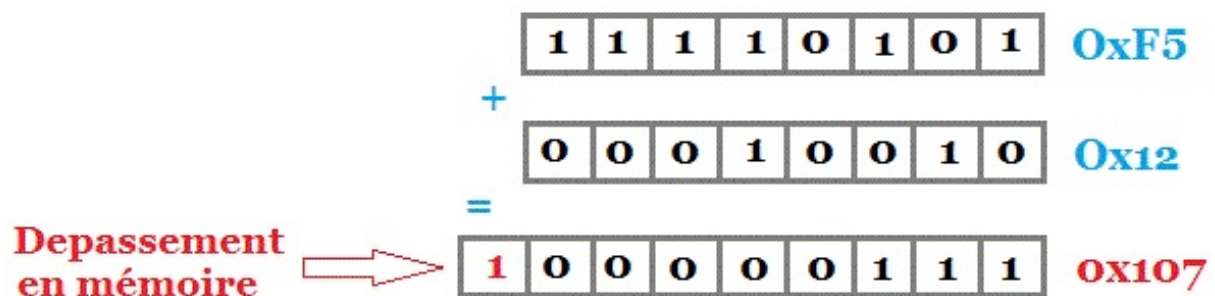
```
cpu.V[b3]=(rand())%(b2<<4)+b1+1;
```

## 8XY4

### Citation

8XY4	Ajoute VY à VX. VF est mis à 1 quand il y a un dépassement de mémoire ( <i>carry</i> ), et à 0 quand il n'y en pas.
------	---

Je profite de cette instruction pour parler un peu de la notion de *carry*. Lors des calculs, il se peut que le résultat obtenu ne puisse pas être contenu dans notre variable. On parle alors de « dépassement » ou de « *carry* ». Par exemple, si nous devons effectuer la somme  $0xF5 + 0x12$ , tout en étant sur 8 bits, le résultat est :  $0x107$ . Cette valeur ne peut être contenue sur 8 bits, il y a donc un dépassement en mémoire (*carry*). Voici un petit schéma pour illustrer le tout :



Dans ce cas-ci, il y a un dépassement si le résultat de l'opération ne peut tenir sur 8 bits (taille des variables  $V_i$ ). Il faut donc vérifier si la somme  $VX + VY$  est inférieure ou supérieure à  $0xFF$ .  $0xFF = 255$ , c'est la valeur maximale que peut prendre un nombre non signé sur 8 bits.

### Code : C

```
if (cpu.V[b3]+cpu.V[b2])>0xFF)
{
    cpu.V[0xF]=1; //V[15]
}
else
{
    cpu.V[0xF]=0; //V[15]
}
cpu.V[b3]+=cpu.V[b2];
```

## 8XY7

### Citation

8XY7	VX = VY - VX. VF est mis à 0 quand il y a un emprunt et à 1 quand il n'y en a pas.
------	--

Il y a un emprunt si le résultat de l'opération est négatif. Il faut alors vérifier que  $VX > VY$  ou  $VY < VX$ , c'est vous qui voyez. Puisque nous avons déclaré nos variables comme étant non signées, les *casts* seront effectués pour nous.

### Code : C

```
if ((cpu.V[b2] < cpu.V[b3])) // !\ VF est mis à 0 quand il y a
    emprunt !
{
    cpu.V[0xF] = 0; //cpu.V[15]
}
else
{
    cpu.V[0xF] = 1; //cpu.V[15]
}

cpu.V[b3] = cpu.V[b2] - cpu.V[b3];
```

## FX33

### Citation

FX33	Stocke dans la mémoire le code décimal représentant VX (dans I, I+1, I+2).
------	--

Le code décimal communément appelé **BCD** est la représentation d'un nombre en base 10.

Pour cette instruction, on doit stocker dans `memoire[I]` les centaines, dans `memoire[I+1]` les dizaines et dans `memoire[I+2]` les unités. Le nombre ne peut avoir de milliers ou plus puisqu'il est sur 8 bits. (La valeur maximale est donc 255 non signé.)

### Code : C

```
cpu.memoire[cpu.I] = (cpu.V[b3] - cpu.V[b3] % 100) / 100; //stocke les centaines
cpu.memoire[cpu.I+1] = ((cpu.V[b3] - cpu.V[b3] % 10) / 10) % 10; //les dizaines
cpu.memoire[cpu.I+2] = cpu.V[b3] - cpu.memoire[cpu.I] * 100 - cpu.memoire[cpu.I+1] * 10; //les unités
```



Pour ceux qui ont opté pour Java, les variables sont signées. C'est à vous de vérifier que vous ne dépassez pas la capacité des variables non signées ou que votre variable est négative en faisant des *casts*.

Par exemple, l'instruction `nombre &= 0xFFFF` permet de maintenir la variable `nombre` sur 16 bits.

## Le mode de dessin intégré

### Citation

FX29	Définit I à l'emplacement du caractère stocké dans VX. Les caractères 0-F (en hexadécimal) sont représentés par une police 4x5.
------	---



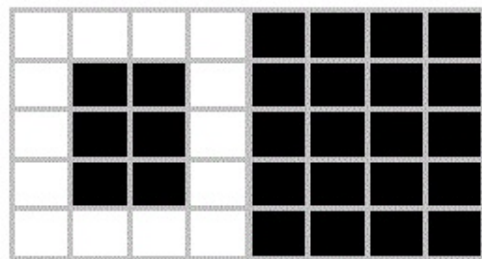
Mais c'est quoi cette histoire ?

Il est vrai que si on se limite à la description, on peut ne pas comprendre de quoi il s'agit (comme je l'avais dit les documentations,

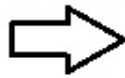
ne sont pas très exhaustives sur tous les plans). Après un petit détour [sur le Web](#), on voit que la Chip 8 possède en mémoire les caractères 0, 1, 2, 3, 4, 5, 6, 7, 8, A, B, C, D, E et F.

Comme pour le graphique, ces caractères sont codés en binaire et ont tous une largeur de 4 pixels et une longueur de 5 pixels. Voici un petit schéma pour éclaircir les idées.

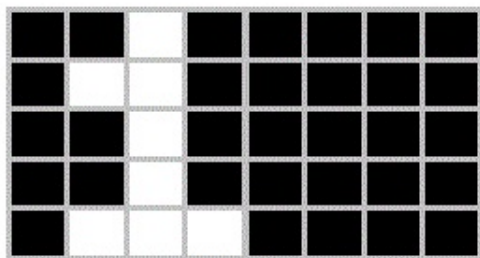
copyright BestCoder



Codage



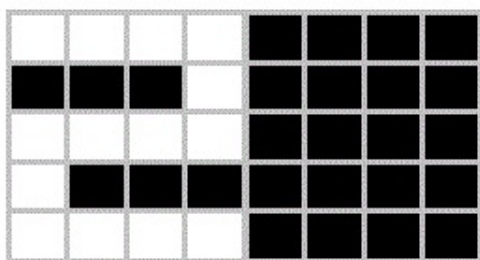
Binaire	Hexadecimal
11110000	0xF0
10010000	0x90
10010000	0x90
10010000	0x90
11110000	0xF0



Codage



Binaire	Hexadecimal
00100000	0x20
01100000	0x60
00100000	0x20
00100000	0x20
01110000	0x70



Codage



Binaire	Hexadecimal
11110000	0xF0
00010000	0x10
11110000	0xF0
10000000	0x80
11110000	0xF0

Tous ces caractères seront stockés dans mémoire à partir de l'adresse 0. Si vous vous souvenez, les 512 premiers octets sont inutilisés. Chaque chiffre occupera cinq cases en mémoire.

Le caractère 0 occupera donc : `memoire[0]`, `memoire[1]`, `memoire[2]`, `memoire[3]` et `memoire[4]`.

De même, le caractère 1 occupera : `memoire[5]`, `memoire[6]`, `memoire[7]`, `memoire[8]` et `memoire[9]`.

N' imaginez même pas la patience qu'il faut pour écrire tout ça sans erreur ni péter un câble. 😊

**Code : C**

```
void chargerFont ()
{
    cpu.memoire[0]=0xF0;cpu.memoire[1]=0x90;cpu.memoire[2]=0x90;cpu.memoire[3]=0
    // 0

    cpu.memoire[5]=0x20;cpu.memoire[6]=0x60;cpu.memoire[7]=0x20;cpu.memoire[8]=0
    // 1

    cpu.memoire[10]=0xF0;cpu.memoire[11]=0x10;cpu.memoire[12]=0xF0;cpu.memoire[1
    cpu.memoire[14]=0xF0; // 2

    cpu.memoire[15]=0xF0;cpu.memoire[16]=0x10;cpu.memoire[17]=0xF0;cpu.memoire[18]=0
    // 3

    cpu.memoire[20]=0x90;cpu.memoire[21]=0x90;cpu.memoire[22]=0xF0;cpu.memoire[23]=0
    // 4
```

```
cpu.memoire[25]=0xF0;cpu.memoire[26]=0x80;cpu.memoire[27]=0xF0;cpu.memoire[28]=0
// 5

cpu.memoire[30]=0xF0;cpu.memoire[31]=0x80;cpu.memoire[32]=0xF0;cpu.memoire[33]=0
// 6

cpu.memoire[35]=0xF0;cpu.memoire[36]=0x10;cpu.memoire[37]=0x20;cpu.memoire[38]=0
// 7

cpu.memoire[40]=0xF0;cpu.memoire[41]=0x90;cpu.memoire[42]=0xF0;cpu.memoire[43]=0
// 8

cpu.memoire[45]=0xF0;cpu.memoire[46]=0x90;cpu.memoire[47]=0xF0;cpu.memoire[48]=0
// 9

cpu.memoire[50]=0xF0;cpu.memoire[51]=0x90;cpu.memoire[52]=0xF0;cpu.memoire[53]=0
// A

cpu.memoire[55]=0xE0;cpu.memoire[56]=0x90;cpu.memoire[57]=0xE0;cpu.memoire[58]=0
// B

cpu.memoire[60]=0xF0;cpu.memoire[61]=0x80;cpu.memoire[62]=0x80;cpu.memoire[63]=0
// C

cpu.memoire[65]=0xE0;cpu.memoire[66]=0x90;cpu.memoire[67]=0x90;cpu.memoire[68]=0
// D

cpu.memoire[70]=0xF0;cpu.memoire[71]=0x80;cpu.memoire[72]=0xF0;cpu.memoire[73]=0
// E

cpu.memoire[75]=0xF0;cpu.memoire[76]=0x80;cpu.memoire[77]=0xF0;cpu.memoire[78]=0
// F

//OUF !

}
```

En définitive :

#### Citation

FX29	Définit I à l'emplacement du caractère stocké dans VX.
------	--

... revient à faire `cpu.I=5*cpu.V[X];`.

#### Exemple

Si `V[X]` contient 1, I vaudra 5 qui est l'adresse de début de stockage du caractère 1.

Si `V[X]` contient 2, I vaudra 10 qui est l'adresse de début de stockage du caractère 2.

Pour pratiquer un peu, je vous laisse finir les autres instructions. Cette phrase est courte mais ne soyez pas surpris si cela prend quelques heures. 🐼

**Charger un jeu**

Il ne reste plus qu'à charger nos jeux afin de faire nos premiers tests.

Les *roms* contiennent toutes les instructions à exécuter, il faudra donc charger le contenu du fichier binaire dans la mémoire.



Dans notre cas, les jeux sont chargés à partir de l'adresse  $512 = 0 \times 200$ .

Voici un lien pour télécharger ceux qui nous intéressent : [jeux Chip8](#). J'ai programmé une *rom* pour vous afin de tester quelques *opcodes*. Vous pourrez l'utiliser pour déboguer votre émulateur. Le voici : [BC\\_Chip8Test](#).

#### Code : C

```

Uint8 chargerJeu(char *nomJeu)
{
    FILE *jeu=NULL;
    jeu=fopen(nomJeu,"rb"); /* Fichier binaire, donc rb */

    if(jeu!=NULL)
    {
        fread(&cpu.memoire[ADRESSEDEBUT],sizeof(Uint8)*(TAILLEMEMOIRE-ADRESSEDEBUT),
        1, jeu);
        fclose(jeu);
        return 1;
    }
    else
    {
        fprintf(stderr,"Problème d'ouverture du fichier");
        return 0;
    }
}

```

Nous sommes fin prêts pour faire nos premiers tests avec la BC-Chip8.

Voici le code mis à jour.

**Secret** ([cliquez pour afficher](#))

#### Code : C - pixel.h

```

#ifndef PIXEL_H
#define PIXEL_H
#include <SDL/SDL.h>

#define NOIR 0
#define BLANC 1
#define l 64
#define L 32
#define DIMPIXEL 8
#define WIDTH l*DIMPIXEL
#define HEIGHT L*DIMPIXEL

typedef struct
{
    SDL_Rect position; //regroupe l'abscisse et l'ordonnée
    Uint32 couleur; //comme son nom l'indique, c'est la couleur
} PIXEL;

SDL_Surface *ecran,*carre[2];
PIXEL pixel[l][L];
SDL_Event event;

void initialiserEcran() ;
void initialiserPixel() ;
void dessinerPixel(PIXEL pixel) ;

```

```

void effacerEcran() ;
void updateEcran() ;

#endif

```

### Code : C - pixel.c

```

#include "pixel.h"

void initialiserPixel()
{
    Uint8 x=0,y=0;

    for(x=0;x<L;x++)
    {
        for(y=0;y<L;y++)
        {
            pixel[x][y].position.x=x*DIMPPIXEL;
            pixel[x][y].position.y=y*DIMPPIXEL;
            pixel[x][y].couleur=NOIR;
        }
    }
}

void initialiserEcran()
{
    ecran=NULL;
    carre[0]=NULL;
    carre[1]=NULL;

    ecran=SDL_SetVideoMode(WIDTH,HEIGHT,32,SDL_HWSURFACE);
    SDL_WM_SetCaption("BC-Chip8 By BestCoder",NULL);

    if(ecran==NULL)
    {
        fprintf(stderr,"Erreur lors du chargement du mode vidéo
%s",SDL_GetError());
        exit(EXIT_FAILURE);
    }

    carre[0]=SDL_CreateRGBSurface(SDL_HWSURFACE,DIMPPIXEL,DIMPPIXEL,32,0,0,0,0);
    //le pixel noir
    if(carre[0]==NULL)
    {
        fprintf(stderr,"Erreur lors du chargement de la surface
%s",SDL_GetError());
        exit(EXIT_FAILURE);
    }
    SDL_FillRect(carre[0],NULL,SDL_MapRGB(carre[0]-
>format,0x00,0x00,0x00)); //le pixel noir

    carre[1]=SDL_CreateRGBSurface(SDL_HWSURFACE,DIMPPIXEL,DIMPPIXEL,32,0,0,0,0);
    //le pixel blanc
    if(carre[1]==NULL)
    {
        fprintf(stderr,"Erreur lors du chargement de la surface
%s",SDL_GetError());
        exit(EXIT_FAILURE);
    }
    SDL_FillRect(carre[1],NULL,SDL_MapRGB(carre[1]-
>format,0xFF,0xFF,0xFF)); //le pixel blanc

```

```

}

void dessinerPixel(PIXEL pixel)
{
    /* pixel.couleur peut prendre deux valeurs : 0, auquel cas on dessine le
    pixel en noir, ou 1, on dessine alors le pixel en blanc */
    SDL_BlitSurface(carre[pixel.couleur], NULL, ecran, &pixel.position);
}

void effacerEcran()
{
    //Pour effacer l'écran, on remet tous les pixels en noir
    Uint8 x=0,y=0;
    for(x=0;x<L;x++)
    {
        for(y=0;y<L;y++)
        {
            pixel[x][y].couleur=NOIR;
        }
    }

    //on repeint l'écran en noir
    SDL_FillRect(ecran, NULL, NOIR);
}

void updateEcran()
{
    //On dessine tous les pixels à l'écran
    Uint8 x=0,y=0;

    for(x=0;x<L;x++)
    {
        for(y=0;y<L;y++)
        {
            dessinerPixel(pixel[x][y]);
        }
    }

    SDL_Flip(ecran); //on affiche les modifications
}

```

#### Code : C - cpu.h

```

#ifndef CPU_H
#define CPU_H
#include "pixel.h"

#define TAILLEMEMOIRE 4096
#define ADRESSEDEBUT 512
#define NBROPCODE 35

typedef struct
{
    Uint8 memoire[TAILLEMEMOIRE];
    Uint8 V[16]; //le registre
    Uint16 I; //stocke une adresse mémoire ou dessinateur
    Uint16 saut[16]; //pour gérer les sauts dans memoire, 16
    au maximum
    Uint8 nbrsaut; //stocke le nombre de sauts effectués pour
    ne pas dépasser 16
    Uint8 compteurJeu; //compteur pour le graphisme
    (fréquence de rafraîchissement)
    Uint8 compteurSon; //compteur pour le son
    Uint16 pc; //pour parcourir le tableau memoire
} CPU;

```



```

CPU cpu;

typedef struct
{
    Uint16 masque[NBROPCODE];    //la Chip 8 peut effectuer 35
    opérations, chaque opération possédant son masque
    Uint16 id[NBROPCODE];        //idem, chaque opération possède son
    propre identifiant
} JUMP;

JUMP jp;

void initialiserJump() ;
void initialiserCpu() ;
void decompter() ;
Uint16 recupererOpcode() ;
Uint8 recupererAction(Uint16) ;
void interpreterOpcode(Uint16) ;
void dessinerEcran(Uint8,Uint8,Uint8) ;
void chargerFont() ;
#endif

```

**Code : C - cpu.c**

```

#include "cpu.h"

void initialiserCpu()
{
    //On initialise le tout

    Uint16 i=0;

    for(i=0;i<TAILLEMEMOIRE;i++)
    {
        cpu.memoire[i]=0;
    }

    for(i=0;i<16;i++)
    {
        cpu.V[i]=0;
        cpu.saut[i]=0;
    }

    cpu.pc=ADRESSEDEBUT;
    cpu.nbrsaut=0;
    cpu.compteurJeu=0;
    cpu.compteurSon=0;
    cpu.I=0;

    initialiserJump(); //n'oubliez surtout pas cette fonction
}

void decompter()
{
    if(cpu.compteurJeu>0)
        cpu.compteurJeu--;

    if(cpu.compteurSon>0)
        cpu.compteurSon--;
}

```

```

Uint16 recupererOpcode()
{
    return (cpu.memoire[cpu.pc]<<8)+cpu.memoire[cpu.pc+1];
}

Uint8 recupererAction(Uint16 opcode)
{
    Uint8 action;
    Uint16 resultat;

    for(action=0; action<NBROPCODE;action++)
    {
        resultat= (jp.masque[action]&opcode); /* On récupère les bits
        concernés par le test */

        if(resultat == jp.id[action]) /* On a trouvé l'action à effectuer */
            break; /* Plus la peine de continuer la boucle*/
    }

    return action;
}

void interpreterOpcode(Uint16 opcode)
{
    Uint8 b4,b3,b2,b1;

    b3=(opcode&(0x0F00))>>8; /*on prend les 4 bits suivants
    b2=(opcode&(0x00F0))>>4; /*idem
    b1=(opcode&(0x000F)); /*idem

    b4= recupererAction(opcode);

    switch(b4)
    {
        case 0:{
            //Cet opcode n'est pas implémenté.
            break;
        }
        case 1:{
            //00E0 efface l'écran.
            effacerEcran();
            break;
        }
        case 2:{
            //00EE revient du saut.

            if(cpu.nbrsaut>0)
            {
                cpu.nbrsaut--;
                cpu.pc=cpu.saut[cpu.nbrsaut];
            }
            break;
        }
        case 3:{
            //1NNN effectue un saut à l'adresse 1NNN.

            cpu.pc=(b3<<8)+(b2<<4)+b1; //on prend le nombre NNN (pour le
            saut)
            cpu.pc-=2; //on verra pourquoi à la fin

            break;
        }
        case 4:{
            //2NNN appelle le sous-programme en NNN, mais on revient ensuite

            cpu.saut[cpu.nbrsaut]=cpu.pc; //on reste là où on était

```

```

        if(cpu.nbrsaut<15)
        {
            cpu.nbrsaut++;
        }

        cpu.pc=(b3<<8)+(b2<<4)+b1; //on prend le nombre NNN (pour le
saut)
        cpu.pc-=2; //on verra pourquoi à la fin

        break;
    }
    case 5:{
        //3XNN saute l'instruction suivante si VX est égal à NN.

        if(cpu.V[b3]==( (b2<<4)+b1))
        {
            cpu.pc+=2;
        }

        break;
    }
    case 6:{
        //4XNN saute l'instruction suivante si VX et NN ne sont pas
égaux.

        if(cpu.V[b3]!=( (b2<<4)+b1))
        {
            cpu.pc+=2;
        }

        break;
    }
    case 7:{
        //5XY0 saute l'instruction suivante si VX et VY sont égaux.
        if(cpu.V[b3]==cpu.V[b2])
        {
            cpu.pc+=2;
        }

        break;
    }
    case 8:{
        //6XNN définit VX à NN.
        cpu.V[b3]=(b2<<4)+b1;
        break;
    }
    case 9:{
        //7XNN ajoute NN à VX.
        cpu.V[b3]+=(b2<<4)+b1;

        break;
    }
    case 10:{
        //8XY0 définit VX à la valeur de VY.
        cpu.V[b3]=cpu.V[b2];

        break;
    }
    case 11:{
        //8XY1 définit VX à VX OR VY.
        cpu.V[b3]=cpu.V[b3] | cpu.V[b2];

        break;
    }
    case 12:{
        //8XY2 définit VX à VX AND VY.
        cpu.V[b3]=cpu.V[b3] & cpu.V[b2];

        break;
    }
}

```

```
case 13:{
    //8XY3 définit VX à VX XOR VY.
    cpu.V[b3]=cpu.V[b3]^cpu.V[b2];

    break;
}
case 14:{
    //8XY4 ajoute VY à VX. VF est mis à 1 quand il y a un
    dépassement de mémoire (carry), et à 0 quand il n'y en pas.
    if((cpu.V[b3]+cpu.V[b2])>255)
    {
        cpu.V[0xF]=1; //cpu.V[15]
    }
    else
    {
        cpu.V[0xF]=0; //cpu.V[15]
    }
    cpu.V[b3]+=cpu.V[b2];

    break;
}
case 15:{
    //8XY5 VY est soustraite de VX. VF est mis à 0 quand il y a
    un emprunt, et à 1 quand il n'y a en pas.

    if((cpu.V[b3]<cpu.V[b2]))
    {
        cpu.V[0xF]=0; //cpu.V[15]
    }
    else
    {
        cpu.V[0xF]=1; //cpu.V[15]
    }
    cpu.V[b3]-=cpu.V[b2];

    break;
}
case 16:{
    //8XY6 décale (shift) VX à droite de 1 bit. VF est fixé à la
    valeur du bit de poids faible de VX avant le décalage.
    cpu.V[0xF]=(cpu.V[b3]&(0x01));
    cpu.V[b3]=(cpu.V[b3]>>1);

    break;
}
case 17:{
    //8XY7 VX = VY - VX. VF est mis à 0 quand il y a un emprunt
    et à 1 quand il n'y en a pas.
    if((cpu.V[b2]<cpu.V[b3]))
    {
        cpu.V[0xF]=0; //cpu.V[15]
    }
    else
    {
        cpu.V[0xF]=1; //cpu.V[15]
    }
    cpu.V[b3]=cpu.V[b2]-cpu.V[b3];

    break;
}
case 18:{
    //8XYE décale (shift) VX à gauche de 1 bit. VF est fixé à la
    valeur du bit de poids fort de VX avant le décalage.
    cpu.V[0xF]=(cpu.V[b3]>>7);
    cpu.V[b3]=(cpu.V[b3]<<1);

    break;
}
case 19:{
```

```

    //9XY0 saute l'instruction suivante si VX et VY ne sont pas
    égaux.
    if(cpu.V[b3] != cpu.V[b2])
    {
        cpu.pc+=2;
    }

    break;
}
case 20: {
    //ANNN affecte NNN à I.

    cpu.I = (b3 << 8) + (b2 << 4) + b1;

    break;
}
case 21: {
    //BNNN passe à l'adresse NNN + V0.

    cpu.pc = (b3 << 8) + (b2 << 4) + b1 + cpu.V[0];
    cpu.pc -= 2;

    break;
}
case 22: {
    //CXNN définit VX à un nombre aléatoire inférieur à NN.
    cpu.V[b3] = (rand()) % ((b2 << 4) + b1 + 1);

    break;
}
case 23: {
    //DXYN dessine un sprite aux coordonnées (VX, VY).

    dessinerEcran(b1, b2, b3) ;

    break;
}
case 24: {
    //EX9E saute l'instruction suivante si la clé stockée dans
    VX est pressée.

    break;
}
case 25: {
    //EXA1 saute l'instruction suivante si la clé stockée dans V
    n'est pas pressée.

    break;
}
case 26: {
    //FX07 définit VX à la valeur de la temporisation.
    cpu.V[b3] = cpu.compteurJeu;

    break;
}
case 27: {
    //FX0A attend l'appui sur une touche et stocke ensuite la
    donnée dans VX.

    break;
}

```

```
case 28:{
    //FX15 définit la temporisation à VX.
    cpu.compteurJeu=cpu.V[b3];

    break;
}
case 29:{
    //FX18 définit la minuterie sonore à VX.
    cpu.compteurSon=cpu.V[b3];

    break;
}
case 30:{
    //FX1E ajoute VX à I. VF est mis à 1 quand il y a overflow
    (I+VX>0xFFF), et à 0 si tel n'est pas le cas.

    if((cpu.I+cpu.V[b3])>0xFFF)
    {
        cpu.V[0xF]=1;
    }
    else
    {
        cpu.V[0xF]=0;
    }
    cpu.I+=cpu.V[b3];

    break;
}

case 31:{
    //FX29 définit I à l'emplacement du caractère stocké dans
    VX. Les caractères 0-F (en hexadécimal) sont représentés par une police 4x5.
    cpu.I=cpu.V[b3]*5;

    break;
}

case 32:{
    //FX33 stocke dans la mémoire le code décimal représentant V
    (dans I, I+1, I+2).

    cpu.memoire[cpu.I]=(cpu.V[b3]-cpu.V[b3]%100)/100;
    cpu.memoire[cpu.I+1]=(((cpu.V[b3]-cpu.V[b3]%10)/10)%10);
    cpu.memoire[cpu.I+2]=cpu.V[b3]-cpu.memoire[cpu.I]*100-10*cpu.memoire[cpu.I+1];

    break;
}

case 33:{
    //FX55 stocke V0 à VX en mémoire à partir de l'adresse I.
    Uint8 i=0;
    for(i=0;i<=b3;i++)
    {
        cpu.memoire[cpu.I+i]=cpu.V[i];
    }

    break;
}

case 34:{
    //FX65 remplit V0 à VX avec les valeurs de la mémoire à
    partir de l'adresse I.

    Uint8 i=0;

    for(i=0;i<=b3;i++)
    {
        cpu.V[i]=cpu.memoire[cpu.I+i];
    }
}
```

```

        }

        break;
    }

    default: { //si ça arrive, il y un truc qui cloche

        break;
    }

}
cpu.pc+=2; //on passe au prochain opcode
}

void dessinerEcran(Uint8 b1,Uint8 b2, Uint8 b3)
{
    Uint8 x=0,y=0,k=0,codage=0,j=0,decalage=0;
    cpu.V[0xF]=0;

    for(k=0;k<b1;k++)
    {
        codage=cpu.memoire[cpu.I+k]; //on récupère le codage de la ligne
à dessiner

        y=(cpu.V[b2]+k)%L; //on calcule l'ordonnée de la ligne à
dessiner, on ne doit pas dépasser L

        for(j=0,decalage=7;j<8;j++,decalage--)
        {
            x=(cpu.V[b3]+j)%1; //on calcule l'abscisse, on ne doit pas
dépasser 1

            if(((codage)&(0x1<<decalage))!=0) //on récupère le
bit correspondant

            { //si c'est blanc
                if( pixel[x][y].couleur==BLANC) //le pixel était
blanc

                {
                    pixel[x][y].couleur=NOIR; //on l'éteint
                    cpu.V[0xF]=1; //il y a donc collusion

                }
                else //sinon
                {
                    pixel[x][y].couleur=BLANC; //on l'allume

                }
            }
        }
    }
}

```

## Code : C - main.c

```

#include <SDL/SDL.h>
#include "cpu.h"

#define VITESSECPU 4 //nombre d'opérations par tour
#define FPS 16 //pour le rafraîchissement

void initialiserSDL();
void quitterSDL();
void pause();

```

```
Uint8 chargerJeu(char *);
Uint8 listen();

int main(int argc, char *argv[])
{
    initialiserSDL() ;
    initialiserEcran() ;
    initialiserPixel() ;

    Uint8 continuer=1,demarrer=0,compteur=0;

    demarrer=chargerJeu("MAZE.ch8") ;

    if(demarrer==1)
    {
        do
        {
            continuer=listen() ; //afin de pouvoir quitter l'émulateur

            for(compteur=0;compteur<VITESSECPU;compteur++)
            {
                interpreterOpcode(recupererOpcode()) ;
            }

            updateEcran();
            decompter();
            SDL_Delay(FPS); //une pause de 16 ms

        }while(continuer==1);
    }

    pause();

    return EXIT_SUCCESS;
}

void initialiserSDL()
{
    atexit(quitterSDL) ;

    if(SDL_Init(SDL_INIT_VIDEO)==-1)
    {
        fprintf(stderr,"Erreur lors de l'initialisation de la SDL
%s",SDL_GetError());
        exit(EXIT_FAILURE) ;
    }
}

void quitterSDL()
{
    SDL_FreeSurface(carre[0]) ;
    SDL_FreeSurface(carre[1]) ;
    SDL_Quit() ;
}

void pause()
{
    Uint8 continuer=1;

    do
    {
        SDL_WaitEvent(&event) ;
```



```
        switch(event.type)
        {
            case SDL_QUIT:
                continuer=0;
                break;
            case SDL_KEYDOWN:
                continuer=0;
                break;
            default: break;
        }
    }while(continuer==1);
}

Uint8 chargerJeu(char *nomJeu)
{
    FILE *jeu=NULL;
    jeu=fopen(nomJeu,"rb") ;

    if(jeu!=NULL)
    {
        fread(&cpu.memoire[ADRESSEDEBUT],sizeof(Uint8)*(TAILLEMEMOIRE-ADRESSEDEBUT),
        1, jeu) ;
        fclose(jeu) ;
        return 1 ;
    }
    else
    {
        fprintf(stderr,"Problème d'ouverture du fichier") ;
        return 0;
    }
}

Uint8 listen()
{
    Uint8 continuer=1;
    while(SDL_PollEvent(&event))
    {
        switch(event.type)
        {
            case SDL_QUIT: {continuer = 0;break;}
            case SDL_KEYDOWN:{continuer=0 ;break;}

            default:{ break;}
        }
    }
    return continuer;
}
```

### Quelques explications

Jetez un coup d'œil dans le *main* (et rien que le *main*, O.K. ?! 😄). Vous pourrez remarquer une fonction `listen()`. Cette fonction n'a rien à voir avec l'émulation : elle nous permet de quitter la boucle principale. Ensuite, nous avons :

Code : C

```
for(compteur=0;compteur<VITESSECPU;compteur++) //on effectue quatre
opérations
{
    interpreterOpcode(recupererOpcode());
}

updateEcran();
decompter(); //les timers

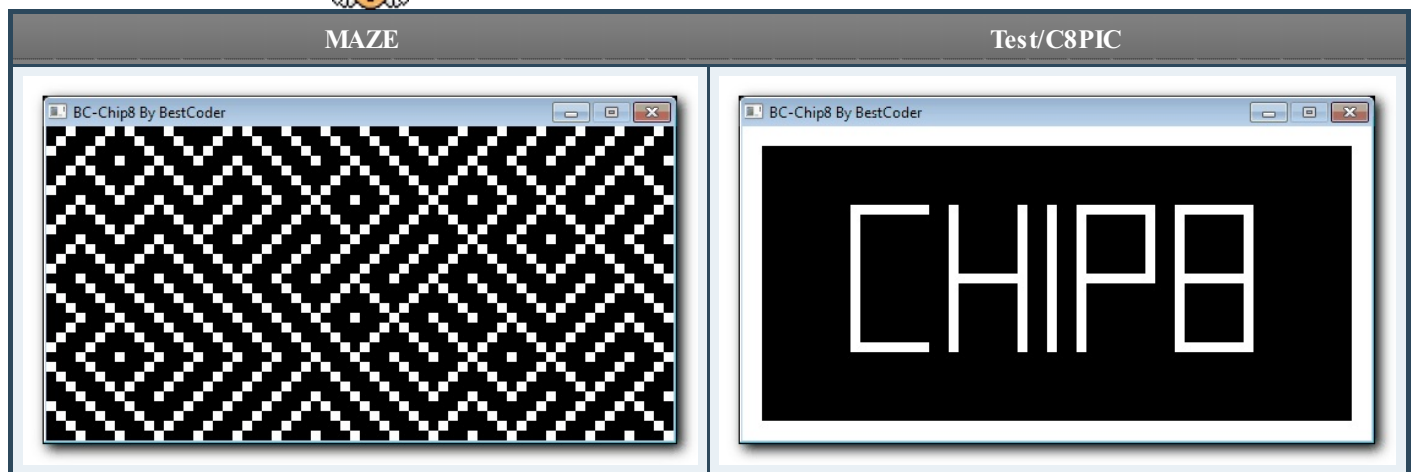
SDL_Delay(FPS); //une pause de 16 ms
```

C'est ici que réside le code qui illustre la partie « Le cadencement du CPU et les FPS ».  
Pour chaque tour de boucle, nous avons :

- les quatre opérations à effectuer grâce au **for** ;
- l'actualisation de l'écran grâce à `updateEcran()` ;
- et la pause de 16 ms grâce à `SDL_Delay()` .

N'oubliez pas non plus d'appeler la fonction `decompter()` ; qui nous permet de décompter à 60 Hz.  
Ainsi, on respecte les spécifications que l'on s'est fixées au début. 🤔

Après cela, on est aux anges. 🙌



Ça fait beaucoup de bien d'obtenir ces résultats. Je dirais que l'émulation est vraiment magique. 🧙

Avec des décalages par-ci, des XOR par-là, on arrive à faire des choses incroyables !  
Tout est bien qui finit bien. Notre émulateur donne des résultats plus que satisfaisants. Il ne nous reste plus qu'à implémenter le son et les entrées utilisateur, et ce sera terminé.



Qui a vu de l'assembleur ici ?

Pas moi, en tout cas. 🤔

## L'interface homme-machine

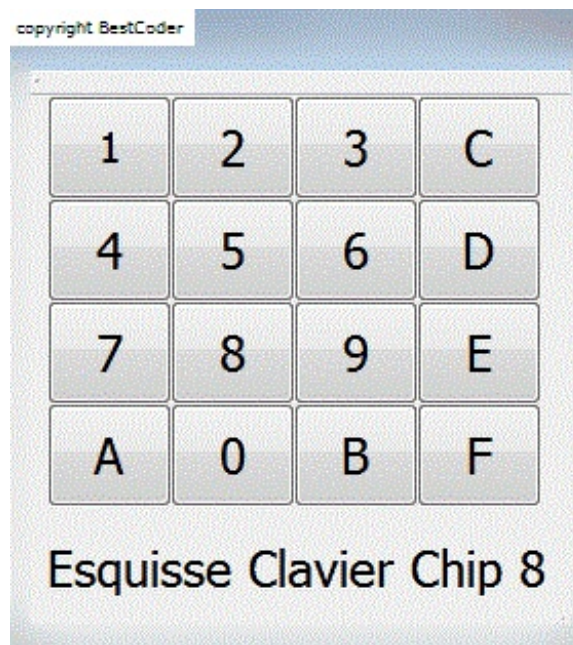
Maintenant que notre émulateur donne des résultats palpables, nous allons nous occuper des entrées utilisateur et du son. Allez... plus que quelques lignes et c'est fini !

### Les entrées utilisateur

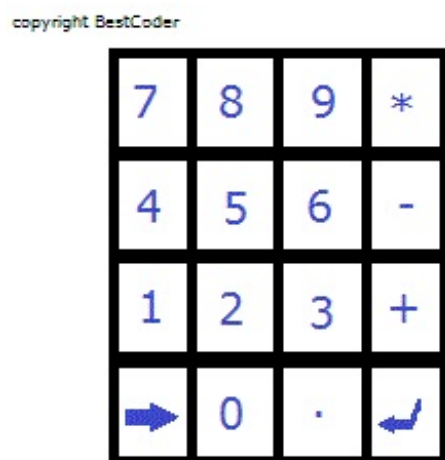
#### *Entrées utilisateur*

##### Citation

L'entrée est faite avec un clavier qui possède 16 touches allant de 0 à F. Les touches « 8 », « 4 », « 6 » et « 2 » sont généralement utilisées pour l'entrée directionnelle.



Pour simuler ce clavier, nous allons utiliser le pavé numérique et la touche de direction droite. Ce choix est purement subjectif. Vous pourrez le modifier à votre guise si vous avez bien compris. [http://uploads.siteduzero.com/files/32 \[...\] 00/327725.gif](http://uploads.siteduzero.com/files/32 [...] 00/327725.gif)



Correspondance avec le  
clavier de l'ordinateur

Il nous faudra une variable pour connaître l'état des boutons (pressés ou non). Pour ma part, j'ai utilisé un tableau de 16 Uint8

pour nos 16 boutons.

Et pour gérer les changements d'état de nos touches (pressé ou relâché), nous allons utiliser les événements « KEYDOWN » et « KEYUP » de SDL. On aura quelque chose qui ressemblera à ceci (les variables touches seront insérées dans la structure CPU).

**Code : C**

```

Uint8 listen()
{
    Uint8 continuer=1;
    while( SDL_PollEvent(&event))
    {
        switch(event.type)
        {
            case SDL_QUIT: {continuer = 0;break;}
            case SDL_KEYDOWN:{

                switch(event.key.keysym.sym)
                {
                    case SDLK_KP0:{
                        cpu.touche[0x0]=1;break;}
                    case SDLK_KP7:{
                        cpu.touche[0x1]=1;break;}
                    case SDLK_KP8:{
                        cpu.touche[0x2]=1;break;}
                    /*
                     Et toutes les autres touches
                     */
                    default:{ break;}
                } ;break;}
            case SDL_KEYUP:{

                switch(event.key.keysym.sym)
                {
                    case SDLK_KP0:{
                        cpu.touche[0x0]=0;break;}
                    case SDLK_KP7:{
                        cpu.touche[0x1]=0;break;}
                    case SDLK_KP8:{
                        cpu.touche[0x2]=0;break;}
                    /*
                     Et toutes les autres touches
                     */
                    default:{ break;}
                } ;break;}
            default:{ break;}
        }
    }
    return continuer;
}

```



J'ai adopté la convention suivante : « 0 » signifie que la touche est relâchée, « 1 » que la touche est pressée.

### Traitement des opcodes

On peut maintenant revenir sur tous les *opcodes* qui traitent les entrées utilisateur.

#### Citation

EXA1	Saute l'instruction suivante si la clé stockée dans VX n'est pas pressée.
------	---

EX9E	Saute l'instruction suivante si la clé stockée dans VX est pressée.
------	---

Ces deux instructions sont relativement simples. La variable VX contiendra un nombre variant de 0 à 15 (pour nos 16 touches). Il faudra vérifier la valeur de `cpu.touche[VX]` et agir en conséquence. Comme nous l'avons déjà vu précédemment pour sauter une instruction, il suffit d'incrémenter `pc` de 2.

#### Code : C

```
//EX9E saute l'instruction suivante si la clé stockée dans VX est
//pressée.
if(cpu.touche[V[b3]]==1) //1 = pressé ; 0 = relâché
{
    cpu.pc+=2;
}

//EXA1 saute l'instruction suivante si la clé stockée dans VX n'est
//pas pressée.
if(cpu.touche[V[b3]]==0) //1 = pressé ; 0 = relâché
{
    cpu.pc+=2;
}
```

Et pour finir avec le clavier, on a :

#### Citation

FX0A	L'appui sur une touche est attendu, puis la valeur correspondante est stockée dans VX.
------	--

Pour attendre l'appui sur une touche, on peut utiliser la fonction `SDL_WaitEvent`. Cette fonction s'exécute tant qu'aucune touche du jeu n'est appuyée. Si une touche quelconque est pressée, on inscrit sa valeur dans VX. Voici le code correspondant :

#### Code : C

```
Uint8 attendAppui(Uint8 b3)
{
    Uint8 attend=1,continuer=1;

    while(attend)
    {
        SDL_WaitEvent(&event);

        switch(event.type)
        {
            case SDL_QUIT:{ continuer=0; attend=0; break;}

            case SDL_KEYDOWN:{

                switch(event.key.keysym.sym)
                {

                    case SDLK_KP0:{
                        cpu.V[b3]=0x0;cpu.touche[0x0]=1;attend=0;break;}
                    case SDLK_KP7:{
                        cpu.V[b3]=0x1;cpu.touche[0x1]=1;attend=0;break;}
                    case SDLK_KP8:{
                        cpu.V[b3]=0x2;cpu.touche[0x2]=1;attend=0;break;}
                    /* Et le reste des touches */
                    default:{ break;}
                } break;}

            default:{ break;}
        }
    }
}
```

```

    }
    return continuer;
}

```

J'ai inséré quelques attributs supplémentaires car je ne voulais pas que l'utilisateur n'ait pas la possibilité de quitter l'émulateur lorsque la fonction est en cours d'exécution. Mais si vous avez compris le principe, le code ne devrait pas poser de problème.

### Testons le tout

Vous pouvez maintenant lancer un jeu comme « Breakout », par exemple. Si tout se passe bien, vous devriez obtenir quelque chose ressemblant à ceci :



## Le son

### Citation : Minuterie sonore

Cette minuterie est utilisée pour les effets sonores. Lorsque sa valeur est différente de zéro, un signal sonore est émis.

Le système sonore de la Chip 8 est relativement simple. Nous avons déjà déclaré notre minuterie sonore ; il suffit juste de poser une condition sur la variable `compteurSon` et de jouer un bip sonore de préférence si sa valeur est différente de zéro. Je ne vais pas détailler les méthodes pour charger un son. Personnellement, j'utilise `SDL_Mixer`.

### Code : C

```

if(cpu.compteurSon!=0)
{
    Mix_PlayChannel(0, son, 0); //permet de jouer le bip sonore
    cpu.compteurSon=0;
}
//Rien de plus simple ^_^

```

Pour ceux qui utilisent des bibliothèques haut niveau, il arrivera un moment où émuler le son deviendra impossible. N'hésitez donc pas à faire des recherches sur le son numérique en général pour vos futurs émulateurs.

Si toutes les caractéristiques essentielles ont été implémentées, il ne reste plus qu'à mettre en place l'interface homme-machine et apporter d'éventuelles améliorations à notre émulateur. À partir de là, c'est chacun pour soi ! 😊

## À vous de jouer ! Améliorations

Pour finir notre émulateur, j'ai rajouté quelques fonctionnalités, à savoir :

- faire une pause ;
- redémarrer un jeu.

Je ne détaillerai pas les actions effectuées, mais vous pouvez ajouter plein d'autres choses à votre émulateur : créer un système de sauvegarde, faire un écran de taille réglable, élaborer une interface pour faciliter le lancement des jeux, etc.

Voici le code final de notre projet.

**Secret** ([cliquez pour afficher](#))

**Code : C - pixel.h**

```
#ifndef PIXEL_H
#define PIXEL_H
#include <SDL/SDL.h>

#define NOIR 0
#define BLANC 1
#define l 64
#define L 32
#define DIMPIXEL 8
#define WIDTH l*DIMPIXEL
#define HEIGHT L*DIMPIXEL

typedef struct
{
    SDL_Rect position; //regroupe l'abscisse et l'ordonnée
    Uint32 couleur;    //comme son nom l'indique, c'est la couleur
} PIXEL;

SDL_Surface *ecran,*carre[2];
PIXEL pixel[l][L];
SDL_Event event;

void initialiserEcran();
void initialiserPixel();
void dessinerPixel(PIXEL pixel);
void effacerEcran();
void updateEcran();

#endif
```

**Code : C - pixel.c**

```
#include "pixel.h"

void initialiserPixel()
{
    Uint8 x=0,y=0;

    for (x=0;x<l;x++)
    {
        for (y=0;y<L;y++)
        {
            pixel[x][y].position.x=x*DIMPIXEL;
            pixel[x][y].position.y=y*DIMPIXEL;
            pixel[x][y].couleur=NOIR;
        }
    }
}
```

```
}

void initialiserEcran()
{
    ecran=NULL;
    carre[0]=NULL;
    carre[1]=NULL;

    ecran=SDL_SetVideoMode(WIDTH,HEIGHT,32,SDL_HWSURFACE);
    SDL_WM_SetCaption("BC-Chip8 By BestCoder",NULL);

    if(ecran==NULL)
    {
        fprintf(stderr,"Erreur lors du chargement du mode vidéo
%s",SDL_GetError());
        exit(EXIT_FAILURE);
    }

    carre[0]=SDL_CreateRGBSurface(SDL_HWSURFACE,DIMPIXEL,DIMPIXEL,32,0,0,0,0);
    //le pixel noir

    if(carre[0]==NULL)
    {
        fprintf(stderr,"Erreur lors du chargement de la surface
%s",SDL_GetError());
        exit(EXIT_FAILURE);
    }

    SDL_FillRect(carre[0],NULL,SDL_MapRGB(carre[0]-
>format,0x00,0x00,0x00)); //le pixel noir

    carre[1]=SDL_CreateRGBSurface(SDL_HWSURFACE,DIMPIXEL,DIMPIXEL,32,0,0,0,0);
    //le pixel blanc

    if(carre[1]==NULL)
    {
        fprintf(stderr,"Erreur lors du chargement de la surface
%s",SDL_GetError());
        exit(EXIT_FAILURE);
    }

    SDL_FillRect(carre[1],NULL,SDL_MapRGB(carre[1]-
>format,0xFF,0xFF,0xFF)); //le pixel blanc
}

void dessinerPixel(PIXEL pixel)
{
    /* pixel.couleur peut prendre deux valeurs : 0, auquel cas on dessine le
    pixel en noir, ou 1, on dessine alors le pixel en blanc */
    SDL_BlitSurface(carre[pixel.couleur],NULL,ecran,&pixel.position);
}

void effacerEcran()
{
    //Pour effacer l'écran, on remet tous les pixels en noir
    Uint8 x=0,y=0;
    for(x=0;x<L;x++)
    {
        for(y=0;y<L;y++)
        {
            pixel[x][y].couleur=NOIR;
        }
    }

    //on repeint l'écran en noir
}
```



```

        SDL_FillRect(ecran, NULL, NOIR);
    }

    void updateEcran()
    {
        //On dessine tous les pixels à l'écran
        Uint8 x=0,y=0;

        for(x=0;x<L;x++)
        {
            for(y=0;y<L;y++)
            {
                dessinerPixel(pixel[x][y]);
            }
        }

        SDL_Flip(ecran); //on affiche les modifications
    }

```

#### Code : C - cpu.h

```

#ifndef CPU_H
#define CPU_H
#include "pixel.h"

#define TAILLEMEMOIRE 4096
#define ADRESSEDEBUT 512
#define NBROPCODE 35

typedef struct
{
    Uint8 memoire[TAILLEMEMOIRE];
    Uint8 V[16]; //le registre
    Uint16 I; //stocke une adresse mémoire ou dessinateur
    Uint16 saut[16]; //pour gérer les sauts dans memoire, 16
    //au maximum
    Uint8 nbrsaut; //stocke le nombre de sauts effectués pour
    //ne pas dépasser 16
    Uint8 compteurJeu; //compteur pour le graphisme
    // (fréquence de rafraîchissement)
    Uint8 compteurSon; //compteur pour le son
    Uint16 pc; //pour parcourir le tableau memoire
    Uint8 touche[16]; //pour stocker l'état des touches
} CPU;

CPU cpu;

typedef struct
{
    Uint16 masque[NBROPCODE]; //la Chip 8 peut effectuer 35
    //opérations, chaque opération possédant son masque
    Uint16 id[NBROPCODE]; //idem, chaque opération possède son
    //propre identifiant
} JUMP;

JUMP jp;

void initialiserJump();
void initialiserCpu();
void decompter();
void chargerFont();
void dessinerEcran(Uint8, Uint8, Uint8);
void reset();

Uint16 recupererOpcode();
Uint8 interpreterOpcode(Uint16);

```

```

    Uint8 interpreterOpCode (Uint16);
    Uint8 recupererAction (Uint16);
    Uint8 attendreAppui (Uint8);

#endif

```

### Code : C - cpu.c

```

#include "cpu.h"

void initialiserCpu()
{
    //On initialise le tout
    Uint16 i=0;

    for (i=0; i<TAILLEMEMOIRE; i++)
    {
        cpu.memoire[i]=0;
    }

    for (i=0; i<16; i++)
    {
        cpu.V[i]=0;
        cpu.saut[i]=0;
        cpu.touche[i]=0;
    }

    cpu.pc=ADRESSEDEBUT;
    cpu.nbrsaut=0;
    cpu.compteurJeu=0;
    cpu.compteurSon=0;
    cpu.I=0;

    initialiserJump();
}

void reset()
{
    Uint8 i=0;
    for (i=0; i<16; i++)
    {
        cpu.V[i]=0;
        cpu.saut[i]=0;
        cpu.touche[i]=0;
    }

    cpu.pc=ADRESSEDEBUT;
    cpu.nbrsaut=0;
    cpu.compteurJeu=0;
    cpu.compteurSon=0;
    cpu.I=0;
    initialiserPixel();
    updateEcran();
}

void initialiserJump()
{
    jp.masque[0]= 0x0000; jp.id[0]=0x0FFF; /* 0NNN */
    jp.masque[1]= 0xFFFF; jp.id[1]=0x00E0; /* 00E0 */
    jp.masque[2]= 0xFFFF; jp.id[2]=0x00EE; /* 00EE */
    jp.masque[3]= 0xF000; jp.id[3]=0x1000; /* 1NNN */
    jp.masque[4]= 0xF000; jp.id[4]=0x2000; /* 2NNN */
    jp.masque[5]= 0xF000; jp.id[5]=0x3000; /* 3XNN */
}

```

```

    jp.masque[6]= 0xF000; jp.id[6]=0x4000;      /* 4XNN */
    jp.masque[7]= 0xF00F; jp.id[7]=0x5000;      /* 5XY0 */
    jp.masque[8]= 0xF000; jp.id[8]=0x6000;      /* 6XNN */
    jp.masque[9]= 0xF000; jp.id[9]=0x7000;      /* 7XNN */
    jp.masque[10]= 0xF00F; jp.id[10]=0x8000;     /* 8XY0 */
    jp.masque[11]= 0xF00F; jp.id[11]=0x8001;     /* 8XY1 */
    jp.masque[12]= 0xF00F; jp.id[12]=0x8002;     /* 8XY2 */
    jp.masque[13]= 0xF00F; jp.id[13]=0x8003;     /* BXY3 */
    jp.masque[14]= 0xF00F; jp.id[14]=0x8004;     /* 8XY4 */
    jp.masque[15]= 0xF00F; jp.id[15]=0x8005;     /* 8XY5 */
    jp.masque[16]= 0xF00F; jp.id[16]=0x8006;     /* 8XY6 */
    jp.masque[17]= 0xF00F; jp.id[17]=0x8007;     /* 8XY7 */
    jp.masque[18]= 0xF00F; jp.id[18]=0x800E;     /* 8XYE */
    jp.masque[19]= 0xF00F; jp.id[19]=0x9000;     /* 9XY0 */
    jp.masque[20]= 0xF000; jp.id[20]=0xA000;     /* ANNN */
    jp.masque[21]= 0xF000; jp.id[21]=0xB000;     /* BNNN */
    jp.masque[22]= 0xF000; jp.id[22]=0xC000;     /* CXNN */
    jp.masque[23]= 0xF000; jp.id[23]=0xD000;     /* DXYN */
    jp.masque[24]= 0xF0FF; jp.id[24]=0xE09E;     /* EX9E */
    jp.masque[25]= 0xF0FF; jp.id[25]=0xE0A1;     /* EXA1 */
    jp.masque[26]= 0xF0FF; jp.id[26]=0xF007;     /* FX07 */
    jp.masque[27]= 0xF0FF; jp.id[27]=0xF00A;     /* FX0A */
    jp.masque[28]= 0xF0FF; jp.id[28]=0xF015;     /* FX15 */
    jp.masque[29]= 0xF0FF; jp.id[29]=0xF018;     /* FX18 */
    jp.masque[30]= 0xF0FF; jp.id[30]=0xF01E;     /* FX1E */
    jp.masque[31]= 0xF0FF; jp.id[31]=0xF029;     /* FX29 */
    jp.masque[32]= 0xF0FF; jp.id[32]=0xF033;     /* FX33 */
    jp.masque[33]= 0xF0FF; jp.id[33]=0xF055;     /* FX55 */
    jp.masque[34]= 0xF0FF; jp.id[34]=0xF065;     /* FX65 */

}

Uint8 recupererAction(Uint16 opcode)
{
    Uint8 action;
    Uint16 resultat;

    for(action=0; action<NBROPCODE;action++)
    {
        resultat= (jp.masque[action]&opcode); /* On récupère les bits conce

        if(resultat == jp.id[action]) /* On a trouvé l'action à effectuer */
            break; /* Plus la peine de continuer la boucle */
    }

    return action;
}

void decompter()
{
    if(cpu.compteurJeu>0)
        cpu.compteurJeu--;

    if(cpu.compteurSon>0)
        cpu.compteurSon--;
}

Uint16 recupererOpcode()
{
    return (cpu.memoire[cpu.pc]<<8)+cpu.memoire[cpu.pc+1];
}

Uint8 interpreterOpcode(Uint16 opcode)
{
    Uint8 continuer=1;
    Uint8 b4,b3,b2,b1;

    b3=(opcode&(0x0F00))>>8; //on prend les 4 bits représentant X
    b2=(opcode&(0x00F0))>>4; //idem pour Y

```

```

b1=(opcode&(0x000F)); //idem

b4= recupererAction(opcode);

switch(b4)
{
    case 0:{
        //Cet opcode n'est pas implémenté.
        break;
    }

    case 1:{
        //00E0 efface l'écran.
        effacerEcran();
        break;
    }

    case 2:{
        //00EE revient du saut.

        if(cpu.nbrsaut>0)
        {
            cpu.nbrsaut--;
            cpu.pc=cpu.saut[cpu.nbrsaut];
        }
        break;
    }

    case 3:{
        //1NNN effectue un saut à l'adresse 1NNN.

        cpu.pc=(b3<<8)+(b2<<4)+b1; //on prend le nombre NNN (pour le
        cpu.pc-=2; //on verra pourquoi à la fin

        break;
    }

    case 4:{
        //2NNN appelle le sous-programme en NNN, mais on revient ensuite

        cpu.saut[cpu.nbrsaut]=cpu.pc; //on reste là où on était

        if(cpu.nbrsaut<15)
        {
            cpu.nbrsaut++;
        }

        cpu.pc=(b3<<8)+(b2<<4)+b1; //on prend le nombre NNN (pour le
        cpu.pc-=2; //on verra pourquoi à la fin

        break;
    }

    case 5:{
        //3XNN saute l'instruction suivante si VX est égal à NN.

        if(cpu.V[b3]==( (b2<<4)+b1))
        {
            cpu.pc+=2;
        }

        break;
    }

    case 6:{
        //4XNN saute l'instruction suivante si VX et NN ne sont pas égaux
        if(cpu.V[b3]!=( (b2<<4)+b1))
        {
            cpu.pc+=2;
        }

        break;
    }

    case 7:{
        //5XY0 saute l'instruction suivante si VX et VY sont égaux.

```

```

        if(cpu.V[b3]==cpu.V[b2])
        {
            cpu.pc+=2;
        }

        break;
    }

    case 8:{
        //6XNN définit VX à NN.
        cpu.V[b3]=(b2<<4)+b1;
        break;
    }

    case 9:{
        //7XNN ajoute NN à VX.
        cpu.V[b3]+=(b2<<4)+b1;

        break;
    }

    case 10:{
        //8XY0 définit VX à la valeur de VY.
        cpu.V[b3]=cpu.V[b2];

        break;
    }

    case 11:{
        //8XY1 définit VX à VX OR VY.
        cpu.V[b3]=cpu.V[b3] | cpu.V[b2];

        break;
    }

    case 12:{
        //8XY2 définit VX à VX AND VY.
        cpu.V[b3]=cpu.V[b3] & cpu.V[b2];

        break;
    }

    case 13:{
        //8XY3 définit VX à VX XOR VY.
        cpu.V[b3]=cpu.V[b3] ^ cpu.V[b2];

        break;
    }

    case 14:{
        //8XY4 ajoute VY à VX. VF est mis à 1 quand il y a un dépassement
        et à 0 quand il n'y en pas.
        if((cpu.V[b3]+cpu.V[b2])>255)
        {
            cpu.V[0xF]=1; //cpu.V[15]
        }
        else
        {
            cpu.V[0xF]=0; //cpu.V[15]
        }
        cpu.V[b3]+=cpu.V[b2];

        break;
    }

    case 15:{
        //8XY5 VY est soustraite de VX. VF est mis à 0 quand il y a
        n'y a en pas.

        if((cpu.V[b3]<cpu.V[b2]))
        {
            cpu.V[0xF]=0; //cpu.V[15]
        }
        else
        {
            cpu.V[0xF]=1; //cpu.V[15]
        }
    }

```

```

        cpu.V[b3] -= cpu.V[b2];

        break;
    }
    case 16: {
        //8XY6 décale (shift) VX à droite de 1 bit. VF est fixé à la
        faible de VX avant le décalage.
        cpu.V[0xF] = (cpu.V[b3] & (0x01));
        cpu.V[b3] = (cpu.V[b3] >> 1);

        break;
    }
    case 17: {
        //8XY7 VX = VY - VX. VF est mis à 0 quand il y a un emprunt
        pas.
        if((cpu.V[b2] < cpu.V[b3]))
        {
            cpu.V[0xF] = 0; //cpu.V[15]
        }
        else
        {
            cpu.V[0xF] = 1; //cpu.V[15]
        }
        cpu.V[b3] = cpu.V[b2] - cpu.V[b3];

        break;
    }
    case 18: {
        //8XYE décale (shift) VX à gauche de 1 bit. VF est fixé à la
        fort de VX avant le décalage.
        cpu.V[0xF] = (cpu.V[b3] >> 7);
        cpu.V[b3] = (cpu.V[b3] << 1);

        break;
    }
    case 19: {
        //9XY0 saute l'instruction suivante si VX et VY ne sont pas
        if(cpu.V[b3] != cpu.V[b2])
        {
            cpu.pc += 2;
        }

        break;
    }
    case 20: {
        //ANNN affecte NNN à I.

        cpu.I = (b3 << 8) + (b2 << 4) + b1;

        break;
    }
    case 21: {
        //BNNN passe à l'adresse NNN + V0.

        cpu.pc = (b3 << 8) + (b2 << 4) + b1 + cpu.V[0];
        cpu.pc -= 2;

        break;
    }
    case 22: {
        //CXNN définit VX à un nombre aléatoire inférieur à NN.
        cpu.V[b3] = (rand()) % ((b2 << 4) + b1 + 1);

        break;
    }
    case 23: {

```

```
//DXYN dessine un sprite aux coordonnées (VX, VY).

dessinerEcran(b1,b2,b3) ;

break;
}
case 24:{
    //EX9E saute l'instruction suivante si la clé stockée dans VX
    if(cpu.touche[cpu.V[b3]]==1)//1 pressé, 0 relaché
    {
        cpu.pc+=2;
    }

    break;
}
case 25:{
    //EXA1 saute l'instruction suivante si la clé stockée dans VX n'
    if(cpu.touche[cpu.V[b3]]==0)//1 pressé, 0 relaché
    {
        cpu.pc+=2;
    }

    break;
}

case 26:{
    //FX07 définit VX à la valeur de la temporisation.
    cpu.V[b3]=cpu.compteurJeu;

    break;
}

case 27:{
    //FX0A attend l'appui sur une touche et stocke ensuite la do
    continuer=attendAppui(b3);

    break;
}

case 28:{
    //FX15 définit la temporisation à VX.
    cpu.compteurJeu=cpu.V[b3];

    break;
}

case 29:{
    //FX18 définit la minuterie sonore à VX.
    cpu.compteurSon=cpu.V[b3];

    break;
}

case 30:{
    //FX1E ajoute VX à I. VF est mis à 1 quand il y a overflow (I+V
    n'est pas le cas.

    if((cpu.I+cpu.V[b3])>0xFFF)
    {
        cpu.V[0xF]=1;
    }
    else
    {
        cpu.V[0xF]=0;
    }
    cpu.I+=cpu.V[b3];

    break;
}
```

```

    case 31:{
        //FX29 définit I à l'emplacement du caractère stocké dans V
        hexadecimal) sont représentés par une police 4x5.
        cpu.I=cpu.V[b3]*5;

        break;
    }

    case 32:{
        //FX33 stocke dans la mémoire le code décimal représentant V

        cpu.memoire[cpu.I]=(cpu.V[b3]-cpu.V[b3]%100)/100;
        cpu.memoire[cpu.I+1]=(((cpu.V[b3]-cpu.V[b3]%10)/10)%10);
        cpu.memoire[cpu.I+2]=cpu.V[b3]-cpu.memoire[cpu.I]*100-10*cpu

        break;
    }

    case 33:{
        //FX55 stocke V0 à VX en mémoire à partir de l'adresse I.
        Uint8 i=0;
        for(i=0;i<=b3;i++)
        {
            cpu.memoire[cpu.I+i]=cpu.V[i];
        }

        break;
    }

    case 34:{
        //FX65 remplit V0 à VX avec les valeurs de la mémoire à par

        Uint8 i=0;

        for(i=0;i<=b3;i++)
        {
            cpu.V[i]=cpu.memoire[cpu.I+i];
        }

        break;
    }

    default: { //si ça arrive, il y un truc qui cloche

        break;
    }
}

cpu.pc+=2; //on passe au prochain opcode
return continuer;
}

void dessinerEcran(Uint8 b1, Uint8 b2, Uint8 b3)
{
    Uint8 x=0,y=0,k=0,codage=0,j=0,decalage=0;
    cpu.V[0xF]=0;

    for(k=0;k<b1;k++)
    {
        codage=cpu.memoire[cpu.I+k]; //on récupère le codage de la ligne

        y=(cpu.V[b2]+k)%L; //on calcule l'ordonnée de la ligne à dessine

        for(j=0,decalage=7;j<8;j++,decalage--)
        {
            x=(cpu.V[b3]+j)%1; //on calcule l'abscisse, on ne doit pas d

            if(((codage)&(0x1<<decalage))!=0) //on récupère le b

```



```

        { //si c'est blanc
          if( pixel[x][y].couleur==BLANC) //le pixel était
          {
            pixel[x][y].couleur=NOIR; //on l'éteint
            cpu.V[0xF]=1; //il y a donc collusion
          }
          else //sinon
          {
            pixel[x][y].couleur=BLANC; //on l'allume
          }
        }
      }
    }

void chargerFont()
{
    cpu.memoire[0]=0xF0;cpu.memoire[1]=0x90;cpu.memoire[2]=0x90;cpu.memoire[3]=0x90;cpu.memoire[4]=0xF0; //0

    cpu.memoire[5]=0x20;cpu.memoire[6]=0x60;cpu.memoire[7]=0x20;cpu.memoire[8]=0x20;cpu.memoire[9]=0x20; //1

    cpu.memoire[10]=0xF0;cpu.memoire[11]=0x10;cpu.memoire[12]=0xF0;cpu.memoire[13]=0xF0;cpu.memoire[14]=0xF0; //2

    cpu.memoire[15]=0xF0;cpu.memoire[16]=0x10;cpu.memoire[17]=0xF0;cpu.memoire[18]=0xF0;cpu.memoire[19]=0xF0; //3

    cpu.memoire[20]=0x90;cpu.memoire[21]=0x90;cpu.memoire[22]=0xF0;cpu.memoire[23]=0xF0;cpu.memoire[24]=0xF0; //4

    cpu.memoire[25]=0xF0;cpu.memoire[26]=0x80;cpu.memoire[27]=0xF0;cpu.memoire[28]=0xF0;cpu.memoire[29]=0xF0; //5

    cpu.memoire[30]=0xF0;cpu.memoire[31]=0x80;cpu.memoire[32]=0xF0;cpu.memoire[33]=0xF0;cpu.memoire[34]=0xF0; //6

    cpu.memoire[35]=0xF0;cpu.memoire[36]=0x10;cpu.memoire[37]=0x20;cpu.memoire[38]=0x20;cpu.memoire[39]=0x20; //7

    cpu.memoire[40]=0xF0;cpu.memoire[41]=0x90;cpu.memoire[42]=0xF0;cpu.memoire[43]=0xF0;cpu.memoire[44]=0xF0; //8

    cpu.memoire[45]=0xF0;cpu.memoire[46]=0x90;cpu.memoire[47]=0xF0;cpu.memoire[48]=0xF0;cpu.memoire[49]=0xF0; //9

    cpu.memoire[50]=0xF0;cpu.memoire[51]=0x90;cpu.memoire[52]=0xF0;cpu.memoire[53]=0xF0;cpu.memoire[54]=0xF0; //A

    cpu.memoire[55]=0xE0;cpu.memoire[56]=0x90;cpu.memoire[57]=0xE0;cpu.memoire[58]=0xE0;cpu.memoire[59]=0xE0; //B

    cpu.memoire[60]=0xF0;cpu.memoire[61]=0x80;cpu.memoire[62]=0x80;cpu.memoire[63]=0x80;cpu.memoire[64]=0x80; //C

```

```

cpu.memoire[65]=0xE0;cpu.memoire[66]=0x90;cpu.memoire[67]=0x90;cpu.memoire[6
//D

cpu.memoire[70]=0xF0;cpu.memoire[71]=0x80;cpu.memoire[72]=0xF0;cpu.memoire[7
//E

cpu.memoire[75]=0xF0;cpu.memoire[76]=0x80;cpu.memoire[77]=0xF0;cpu.memoire[7
//F

    //OUF !
}

Uint8 attendAppui(Uint8 b3)
{
    Uint8 attend=1,continuer=1;

    while(attend)
    {
        SDL_WaitEvent(&event);

        switch(event.type)
        {
            case SDL_QUIT:{ continuer=0;attend=0; break;}

            case SDL_KEYDOWN:{

                switch(event.key.keysym.sym)
                {

                    case SDLK_KP0:{ cpu.V[b3]=0x0; cpu.touche[0x0]=1
                    case SDLK_KP7:{ cpu.V[b3]=0x1; cpu.touche[0x1]=1
                    case SDLK_KP8:{ cpu.V[b3]=0x2; cpu.touche[0x2]=1
                    case SDLK_KP9:{ cpu.V[b3]=0x3; cpu.touche[0x3]=1
                    case SDLK_KP4:{ cpu.V[b3]=0x4; cpu.touche[0x4]=1
                    case SDLK_KP5:{ cpu.V[b3]=0x5; cpu.touche[0x5]=1
                    case SDLK_KP6:{ cpu.V[b3]=0x6; cpu.touche[0x6]=1
                    case SDLK_KP1:{ cpu.V[b3]=0x7; cpu.touche[0x7]=1
                    case SDLK_KP2:{ cpu.V[b3]=0x8; cpu.touche[0x8]=1
                    case SDLK_KP3:{ cpu.V[b3]=0x9; cpu.touche[0x9]=1
                    case SDLK_RIGHT:{ cpu.V[b3]=0xA; cpu.
attend=0;break;}
                    case SDLK_KP_PERIOD:{ cpu.V[b3]=0xB; cpu.
attend=0;break;}
                    case SDLK_KP_MULTIPLY:{ cpu.V[b3]=0xC; cpu.
attend=0;break;}
                    case SDLK_KP_MINUS:{ cpu.V[b3]=0xD; cpu.
attend=0;break;}
                    case SDLK_KP_PLUS:{ cpu.V[b3]=0xE; cpu.
attend=0;break;}
                    case SDLK_KP_ENTER:{ cpu.V[b3]=0xF; cpu.
attend=0;break;}
                    default:{ break;}
                } break;}

            default:{ break;}
        }
    }

    return continuer;
}

```

## Code : C - main.c

```
#include <SDL/SDL_mixer.h>
#include "cpu.h"

#define VITESSECPU 4 //nombre d'opérations par tour
#define FPS 16 //pour le rafraîchissement

void initialiserSDL();
void quitterSDL();
void pause();

Uint8 chargerJeu(char *);
Uint8 listen();

Mix_Chunk *son;

int main(int argc, char *argv[])
{
    initialiserSDL();
    initialiserEcran();
    initialiserPixel();
    initialiserCpu();
    chargerFont();

    Uint8 continuer=1,demarrer=0,compteur=0;

    son=NULL;

    son = Mix_LoadWAV("SON/beep.wav");

    if(son==NULL)
    {
        fprintf(stderr,"Problème avec le son");
        exit(EXIT_FAILURE);
    }

    if(argc>=2) //Permet de charger un jeu en ligne de commande ou en le
    plaçant dans l'exécutable
    {
        demarrer=chargerJeu(argv[1]);
    }

    if(demarrer==1)
    {
do
    {
        continuer=listen(); //pour les entrées utilisateur

        for(compteur=0;compteur<VITESSECPU && continuer==1;compteur++) //Si
        continuer=0, on quitte l'émulateur
        {
            continuer=interpreterOpcode(recupererOpcode());
        }

        if(cpu.compteurSon!=0)
        {
            Mix_PlayChannel(0, son, 0);
            cpu.compteurSon=0;
        }

        updateEcran();
        decompter();

        SDL_Delay(FPS); //une pause de 16 ms
```

```

    }while(continuer==1);

    }

    return EXIT_SUCCESS;
}

void initialiserSDL()
{
    atexit(quitterSDL);

    if(SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO)==-1)
    {
        fprintf(stderr, "Erreur lors de l'initialisation de la SDL
%s", SDL_GetError());
        exit(EXIT_FAILURE);
    }

    if(Mix_OpenAudio(22050, MIX_DEFAULT_FORMAT, MIX_DEFAULT_CHANNELS, 1024)
== -1) //Initialisation de Mixer
    {
        fprintf(stderr, "Problème d'initialisation de SDL_MIXER:
%s", Mix_GetError());
        exit(EXIT_FAILURE);
    }
    Mix_AllocateChannels(1);
}

void quitterSDL()
{
    SDL_FreeSurface(carre[0]);
    SDL_FreeSurface(carre[1]);
    Mix_FreeChunk(son);
    Mix_CloseAudio();
    SDL_Quit();
}

Uint8 chargerJeu(char *nomJeu)
{
    FILE *jeu=NULL;
    jeu=fopen(nomJeu, "rb");

    if(jeu!=NULL)
    {
        fread(&cpu.memoire[ADRESSEDEBUT], sizeof(Uint8) * (TAILLEMEMOIRE-ADRESSEDEBUT),
1, jeu);
        fclose(jeu);
        return 1;
    }
    else
    {
        fprintf(stderr, "Problème d'ouverture du fichier");
        return 0;
    }
}

Uint8 listen()
{
    Uint8 continuer=1;

    while(SDL_PollEvent(&event))
    {
        switch(event.type)

```

```

        {
            case SDL_QUIT: {continuer = 0; break;}
            case SDL_KEYDOWN: {
                switch(event.key.keysym.sym)
                {
                    case SDLK_KP0: {
                        case SDLK_KP7: {
                            case SDLK_KP8: {
                                case SDLK_KP9: {
                                    case SDLK_KP4: {
                                        case SDLK_KP5: {
                                            case SDLK_KP6: {
                                                case SDLK_KP1: {
                                                    case SDLK_KP2: {
                                                        case SDLK_KP3: {
                                                            case SDLK_RIGHT: {
                                                                case
                        cpu.touche[0x0]=1; break;}
                        cpu.touche[0x1]=1; break;}
                        cpu.touche[0x2]=1; break;}
                        cpu.touche[0x3]=1; break;}
                        cpu.touche[0x4]=1; break;}
                        cpu.touche[0x5]=1; break;}
                        cpu.touche[0x6]=1; break;}
                        cpu.touche[0x7]=1; break;}
                        cpu.touche[0x8]=1; break;}
                        cpu.touche[0x9]=1; break;}
                        cpu.touche[0xA]=1; break;}
                        case
                        SDLK_KP_PERIOD: {cpu.touche[0xB]=1; break;}
                        case
                        SDLK_KP_MULTIPLY: {cpu.touche[0xC]=1; break;}
                        case
                        SDLK_KP_MINUS: {cpu.touche[0xD]=1; break;}
                        case
                        SDLK_KP_PLUS: {cpu.touche[0xE]=1; break;}
                        case
                        SDLK_KP_ENTER: {cpu.touche[0xF]=1; break;}
                        case SDLK_p: {pause(); break;}
                        case SDLK_r: {reset(); break;}
                        default: { break;}
                    }
                }
                break;}
            case SDL_KEYUP: {
                switch(event.key.keysym.sym)
                {
                    case SDLK_KP0: {
                        case SDLK_KP7: {
                            case SDLK_KP8: {
                                case SDLK_KP9: {
                                    case SDLK_KP4: {
                                        case SDLK_KP5: {
                                            case SDLK_KP6: {
                                                case SDLK_KP1: {
                                                    case SDLK_KP2: {
                                                        case SDLK_KP3: {
                                                            case SDLK_RIGHT: {
                                                                case
                        cpu.touche[0x0]=0; break;}
                        cpu.touche[0x1]=0; break;}
                        cpu.touche[0x2]=0; break;}
                        cpu.touche[0x3]=0; break;}
                        cpu.touche[0x4]=0; break;}
                        cpu.touche[0x5]=0; break;}
                        cpu.touche[0x6]=0; break;}
                        cpu.touche[0x7]=0; break;}
                        cpu.touche[0x8]=0; break;}
                        cpu.touche[0x9]=0; break;}
                        cpu.touche[0xA]=0; break;}
                        case
                        SDLK_KP_PERIOD: {cpu.touche[0xB]=0; break;}
                        case

```

```

SDLK_KP_MULTIPLY:{cpu.touche[0xC]=0;break;}
                                case
SDLK_KP_MINUS:{cpu.touche[0xD]=0;break;}
                                case
SDLK_KP_PLUS:{cpu.touche[0xE]=0;break;}
                                case
SDLK_KP_ENTER:{cpu.touche[0xF]=0;break;}
                                default:{ break;}
                                }
        break;}

                                default:{ break;}
        }

return continuer;
}

void pause()
{
    Uint8 continuer=1;

    do
    {
        SDL_WaitEvent(&event);

        switch(event.type)
        {
            case SDL_QUIT:

                continuer=0;
                break;

            case SDL_KEYDOWN:

                if(event.key.keysym.sym==SDLK_p)
                    continuer=0;
                break;

            default:    break;
        }
    }while(continuer==1);

    SDL_Delay(200); //on fait une petite pause pour ne pas prendre le
    joueur au dépourvu
}

```



Le code final **diffère** légèrement des codes vus plus haut car j'ai moi-même ajouté de nouvelles fonctionnalités à mon émulateur. Vous verrez de nouvelles variables et fonctions qui ne sont pas indispensables. Essayez d'écrire votre propre code afin de vous améliorer.

Si vous désirez tester le code, il suffit de créer un dossier dans le même emplacement que l'exécutable. Nommez-le « SON » et placez-y un fichier son au format WAV. Le fichier son devra être nommé « beep ».

## La compatibilité

Si vous testez certains jeux, notamment blinky et blitz, vous verrez que l'émulateur ne peut pas les faire fonctionner. C'est un problème de compatibilité qui se pose.



« Pourquoi ? », me diriez-vous...



Je le dis et je le répète, les informations sur lesquelles nous nous basons pour construire notre émulateur ne sont pas unanimes. Certains jeux utilisent donc vraisemblablement des propriétés que nous n'avons pas implémentées. 😞

Mais en *bidouillant*, j'ai remarqué que le jeu blitz bugguait à cause du modulo dans la fonction de dessin sur l'écran. Donc pour ce jeu, tout ce qui sort de l'écran ne doit pas être redessiné.

De même pour blinky, il faut augmenter la vitesse de jeu pour obtenir un bon rendu ; d'où l'intérêt de développer un émulateur **configurable**.

Je ne laisse pas de fichier .zip à télécharger pour la bonne et simple raison qu'un émulateur Chip 8, ce n'est pas ce qui manque, et vous êtes censés programmer le vôtre. Allez, au boulot ! 😊

## À suivre

Vous n'avez là qu'une **petite** esquisse de l'émulation console. Plusieurs notions n'ont pas été abordées puisque n'étant pas utilisées par la Chip 8. En vous souhaitant une bonne continuation, je vous donne quelques pistes à suivre...

### Le son

L'émulation du son n'est pas aussi facile qu'on pourrait l'imaginer. La Chip 8 est un mauvais exemple dans ce domaine. Gardez en tête qu'il faut, dans la plupart des cas, générer le son grâce aux instructions du jeu sans faire appel à une ressource externe.

### Les cycles

Dans notre bloc **switch**, nous avons défini le même temps d'exécution pour tous les *opcodes* ; toutefois, certaines consoles introduisent la notion de cycle d'horloge.

En effet, il se peut que certaines instructions prennent plus de temps à être exécutées que d'autres. Il faudra dans ce cas trouver des astuces afin d'assurer une émulation optimale.

### Débugguer

Le nombre d'instructions de la Chip 8 est relativement faible. C'est d'ailleurs ce qui a motivé mon choix de la traiter. Pour les nouvelles consoles (même des anciennes), le nombre d'instructions est énorme, il sera donc utile de créer un débogueur pour voir quels sont les *opcodes* qui ne fonctionnent pas comme vous le souhaitez. Cela vous facilitera grandement la tâche.

### Optimisation

Certaines consoles sont cadencées à une fréquence tellement élevée qu'une simple approche avec un bloc **switch** ne pourrait satisfaire. Il existe une méthode appelée **recompilation dynamique** (dynarec) qui permet de contourner cet obstacle. Cette méthode, bien qu'un peu compliquée, offre des performances inégalables !

### Plus loin

Vous avez fini le tutoriel et vous en voulez encore plus ?

La Chip 8 a un descendant appelé SuperChip8. Vous pourrez facilement trouver tout ce qu'il vous faut pour l'émuler. D'ailleurs, la SuperChip8 partage beaucoup d'instructions identiques avec la Chip 8, vous pourrez donc mettre à jour votre émulateur Chip 8 avec seulement quelques modifications.

La Superchip 8 a aussi un descendant appelé Mégachip 8. 😊



Voici un système tout nouveau, **CHIP 16**, qui est conçu spécialement pour les débutants en émulation. C'est pas mal pour un second émulateur (un membre du SdZ, timmehboy, fait partie des contributeurs). Le système est en couleur et est très agréable. Toutes les informations qu'il faut pour l'implémenter sont disponibles. Vous pourrez donc espérer une compatibilité de 100 %.

Bonne chasse. 😊

Nous voici à la fin de notre premier épisode. 😊 Vous venez de faire vos premiers pas dans le monde de l'émulation console et

j'espère que ce tutoriel vous a été utile.

Merci de m'avoir lu, je vous donne rendez-vous bientôt pour une nouvelle aventure.

Si vous regroupez toutes les citations utilisées, vous retrouverez la présentation entière de Wikipédia. Eh oui, on vient de traduire le document en langage machine. 🧙‍♂️ Donc, lorsque je disais :

#### Citation : BestCoder

C'est grâce à ce document que nous allons programmer notre émulateur ; nous allons le traduire en langage machine. 😊

...j'avais entièrement raison et vous pouvez en témoigner.

Vous voilà fin prêts pour vous aventurer dans le monde de l'émulation console.

Ce tutoriel, loin d'être exhaustif, ne représente qu'un aperçu de ce vaste domaine. Après cette initiation, vous êtes en mesure de programmer des émulateurs bien plus complexes du point de vue architectural sans problème.



#### Comment programmer un émulateur ?

Vous devez me dire sans hésiter qu'il suffit de :

- trouver les caractéristiques de la machine ;
- traduire le tout dans le langage de programmation de votre choix.

Mais j'insiste : pour réaliser un émulateur Sony Next Generation Portable (NGP) ou Nintendo 3DS (ou même PS3 ou XBOX 360 pour les plus téméraires 😊), il faudra creuser un peu plus.

Deux ou trois autres TP verront bientôt le jour. Ce sera l'occasion pour vous d'apprendre des notions plus avancées et de passer en mode couleur.