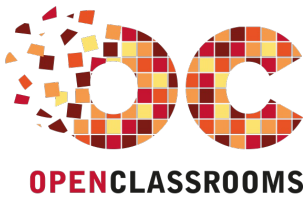
Programmez sur votre DS avec MicroLua

Par Reylak



www.openclassrooms.com

Dernière mise à jour le 17/09/2011

Sommaire

Programmez sur votre DS avec MicroLua

Par  Reylak

Mise à jour : 17/09/2011
Difficulté : Intermédiaire

Vous voulez programmer pour votre DS simplement ?

Ce cours est fait pour vous.

Je vais donc vous apprendre à utiliser MicroLua. Comme son nom l'indique, ~~il est microscopique~~ il est basé sur le langage de programmation Lua.

Vous n'aurez pas à apprendre toutes les facettes du Lua pour bien maîtriser µLua. En revanche, je considèrerai que vous avez les bases de la programmation en général, même si vous ne maîtrisez pas forcément le Lua. Le chapitre "Les bases de la programmation avec Lua" n'a pour but que de vous inculquer la syntaxe basique du langage. Bien sûr, avoir des connaissances dans ce langage est une bonne chose

Bon allez au boulot !

Partie 1 : Les bases

Dans ce chapitre je vais vous apprendre à installer et tester Microlua (que j'abrègerai en µLua) sur votre DS et les bases de la programmation avec µLua.

C'est parti !

A propos de MicroLua et installation

Dans ce chapitre (le plus petit mais le plus important), je vais vous enseigner comment installer et tester µLua. Je vais aussi vous faire un petit briefing sur µLua.

C'est parti !

Introduction à MicroLua

Lua

Je dois commencer par vous faire comprendre la différence entre Lua et MicroLua.
Le premier est un langage informatique, créé par une équipe de développeurs portugais d'une université de Rio de Janeiro au Brésil (au passage, "Lua" signifie "lune" en portugais, vous comprendrez ainsi son logo). Tel qu'il est dit sur le site officiel, c'est un langage "puissant, rapide, léger et embarquable". Pour vous donner une idée de ses utilisations, ce langage de script est souvent utilisé par des applications pour le développement et l'intégration d'addons, je citerai notamment World of Warcraft ou encore Allods Online, ainsi que VLC.
Le second, et bien... Lisez la suite

MicroLua

MicroLua est un homebrew (c'est-à-dire un programme amateur) développé pour la DS par Risike. C'est un portage du langage de script Lua sur la DS, par l'intermédiaire de la µLibrary de Brunni (d'où le "Micro"). Vous ne devez donc pas confondre Lua, qui est un langage à part entière, et MicroLua, qui est une adaptation du Lua sur la DS.

Je tiens aussi à faire remarquer qu'il existe un projet DSLua. Il n'y a pas vraiment de rapport entre celui-ci et MicroLua, si ce n'est le même langage (et aucun n'a copié sur l'autre). Ce que je veux dire, c'est que vous ne devez pas les confondre, car les caractéristiques, les fonctions, etc. ne sont pas les mêmes.

Il y a un site officiel (carrément pas à jour) et surtout un forum, qui lui est très actif et possède une communauté vivante (nan, nan, j'fais pas de la pub). Donc, si vous avez le moindre souci, vous savez où chercher !

A l'heure où ces lignes sont écrites ~~d'une main experte~~, la dernière version de MicroLua est la 4.1. Quelques membres s'occupent de son amélioration. Je parle bien de membres de la communauté puisque le 9 juillet 2009 Risike a libéré les sources du homebrew, en expliquant qu'il n'avait plus trop le temps de s'y investir pleinement.

Maintenant, nous pouvons passer à l'installation de MicroLua !

Installation de MicroLua

Avant de commencer à programmer, on va installer µLua !

Tout d'abord, de quoi avons-nous besoin ?

- D'une DS (encore que vous pouvez très bien n'utiliser que des émulateurs pour ordinateur, mais c'est mieux en vrai).
- D'un linker.
- De MicroLua 4.0.2, disponible sur le Google Code.
- D'un éditeur de texte (le mieux est de pouvoir bénéficier de la coloration syntaxique pour Lua, même si bien sûr les fonctions de µLua ne seront pas reconnues – je pense que la plupart des IDE permettent cependant de personnaliser la coloration en rajoutant des mots-clés).

En décompressant l'archive de MicroLua, vous obtenez un dossier *Micro Lua 4.0*, qui contient :

- Un dossier ***Documentation***, sous forme de fichiers HTML : c'est en fait une version hors-ligne du wiki auquel vous pouvez accéder via le Google Code ici (majoritairement en anglais) ; il va de soit que la version en ligne sera plus à jour que celle de votre dossier. Il y a par ailleurs un dossier "fr" contenant les pages traduites
- Un dossier ***lua*** : c'est ce dossier qui contient les fichiers nécessaires au bon fonctionnement de MicroLua.
- Trois fichiers ***microlua.nds***, ***microlua.sc.nds*** et ***microlua.ds.gba*** : ce sont les trois versions différentes de MicroLua, à choisir en fonction de votre linker.
- Un fichier ***README*** : il est là pour présenter MicroLua, son installation, et donne encore quelques informations intéressantes ; cela dit, ce tutoriel se suffit à lui-même
- Un fichier ***CHANGELOG*** : si vous voulez savoir ce qui a changé depuis la dernière version 3, lisez-le !

Suivant votre linker, choisissez *microlua.nds* (le plus souvent), *microlua.sc.nds* (pour les SuperCard) ou *microlua.ds.gba* (pour les linkers GBA). Copiez ce fichier ainsi que le répertoire *lua* à la racine de votre linker.

Copiez bien le dossier *lua* à la racine de votre linker ! En revanche, vous pouvez mettre où vous voulez l'exécutable.

Dans ce fameux dossier *lua*, vous trouverez le dossier *libs*, qui contient à juste titre les fichiers que MicroLua utilise, ainsi que le dossier *scripts* : c'est celui qui est prévu pour contenir vos scripts Lua, bien qu'il ne soit pas obligatoire de les mettre là. Ce répertoire contient des exemples d'utilisation des fonctions que nous allons voir.
C'est aussi dans *libs* que se trouve le *shell* de µLua. C'est l'interface noire avec des écritures jaunes que vous voyez en le lançant ; vous pouvez le remplacer sans aucun problème par un des shells qui se trouvent sur le forum.

Vous devriez avoir une arborescence comme ceci :

- *Racine de votre carte*
  - lua
    - libs
    - scripts
      - Examples

Et votre exécutable quelque part dans votre carte.

Test de MicroLua

Hello World!... ou pas

Nous allons maintenant vérifier le bon fonctionnement de µLua.

Créez le fichier *test.lua* dans le dossier *lua/scripts*, et copiez/collez-y le code suivant :

Code : Lua

```lua
while not Keys.held.Start do
        Controls.read()

 screen.print(SCREEN_UP, 0, 0, "O.K. !")

 render()
end
```

Ne vous inquiétez pas de la signification de ce code : nous verrons ça plus tard.

Démarrez votre DS et lancez votre linker. Lancez le fichier *microlua.nds* (ou celui qui correspond à votre linker) puis sélectionnez le fichier *test.lua* dans la liste (avec la croix directionnelle) et appuyez sur A.

Si vous voyez "O.K. !" sur l'écran du haut c'est que tout fonctionne : vous pouvez alors éteindre votre DS, sauter de joie et courir partout dans votre maison, et même à poil dans la rue (nan, mais revenez quand même ).

Ça marche pas !

En général, MicroLua marche tout de suite correctement. Cependant, il est possible que vous ayez l'erreur "couldn't open libs.lua". Elle signifie que MicroLua ne peut pas trouver le fichier *libs.lua*, qui doit être placé dans le dossier */lua*. Vérifiez donc que votre arborescence est correcte.

On a maintenant installé µLua et nous l'avons testé, il ne nous reste donc plus qu'à apprendre à programmer 

Les bases de la programmation avec Lua

Je vais maintenant vous apprendre les bases de la programmation Lua, et rattacher tout ça à µLua.
C'est une partie tout ce qu'il y a de plus théorique, mais c'est obligatoire. Désolé

Les bases de la programmation en Lua (1)

Cette partie explique les bases de la programmation en **Lua**. Vous n'y apprendrez rien qui concerne directement µLua, cependant c'est indispensable. Les connaisseurs du langage Lua peuvent quant à eux directement passer à la partie "Et pour MicroLua ?".

Les commentaires

Pour écrire un commentaire en Lua, c'est très simple :

Code : Lua

```
-- Ça, c'est un commentaire

Ici, des instructions      -- Et là, un commentaire en fin de ligne

--[[ Et ça, c'est
un commentaire sur plusieurs lignes ]]--
```

Les variables

En Lua, comme dans la plupart des langages, on affecte une variable de cette façon :

Code : Lua

```
variable = valeur
```

Il est aussi possible d'affecter plusieurs variables en même temps :

Code : Lua

```
i, j = 1, 2
-- Et on peut même échanger les valeurs en une seule ligne !
i, j = j, i
```

Et comme dans la plupart des langages (tous ?), Lua est sensible à la casse, c'est-à-dire que *variable*, *VARIABLE* et *vARiaBlE* sont trois variables différentes.

A la base, toutes les variables sont globales, mais vous pouvez définir une variable locale à un bloc d'instructions (voir à la partie suivante) grâce au mot clé *local*. La portée des variables n'étant pas une notion essentielle pour coder avec µLua, je ne développerai pas ce point.

Les nombres

Rien de bien particulier :

Code : Lua

```
variable = 12
variable2 = 45.72
```

Mis à part le fait qu'il n'y a pas de moyen pour incrémenter ou décrémenter une variable rapidement (*variable++* ou *variable--* par exemple en C). Vous serez obligés de faire :

Code : Lua

```
variable = variable + 1
variable = variable - 1
-- Et pas non plus d'opérateurs +=, -=, etc. !
```

Les chaînes de caractères

Les chaînes de caractères (ou strings) sont délimitées par des guillemets (""), des apostrophes ('), voire même des doubles-crochets ([[]]). Cette dernière notation permet de présenter des strings sur plusieurs lignes, cependant elle ne permet pas l'insertion de caractères spéciaux ; un équivalent à cette notation est l'utilisation de l'antislash "\" en fin de ligne pour écrire un String sur plusieurs lignes (et là vous pouvez mettre des caractères spéciaux) :

Code : Lua

```
string = "Chaîne\
sur deux lignes."
```

Comme dans de nombreux langages, on représente le retour à la ligne avec "\n".

Un petit mot sur la concaténation : ce terme barbare signifie que l'on rassemble plusieurs variables en une seule. Elle peut se faire pour des tables grâce à une fonction, mais c'est surtout utile pour les strings. Pour concaténer deux chaînes, on fait comme ceci :

Code : Lua

```
string = chaine.."du texte"..chaine2
```

Comme vous le voyez, on utilise deux points ".." pour relier les chaînes.

Les tableaux

Les listes, ou tableaux, sont appelés **tables** en Lua, et sont délimités par des accolades. Les éléments sont séparés par des virgules ou des points-virgules.
Lua est très souple, vous pouvez mettre n'importe quel type de variable dans les tables (nombres, strings, d'autres tables, et même des fonctions, qui sont considérées en Lua comme des variables), excepté bien évidemment le type *nil*.
Vous pouvez accéder à un élément de la table en faisant :

Code : Lua

```
table[index]
```

Code : Lua

```
table = {stringCle = valeur, [fonction] = valeur2} -- Notez les
crochets pour un index du type fonction

-- Et pour y accéder :
table["stringCle"] = 123
table.stringCle = 123 -- Ça marche aussi, j'en reparlerai plus tard
var = table[fonction]
```

</information>

Mais attention : Lua est rebelle, et les indices numériques des tables commencent à **1**, et non pas à 0 comme dans la plupart des langages.

Les booléens

Les booléens en Lua, ce sont deux valeurs : *true* et *false*.

Code : Lua

```
vrai = true
faux = false
-- Attention, il n'y a pas de guillemets autour !
```

Au passage, pour Lua, tout ce qui ne vaut pas *false* ou *nil* (à voir juste après) vaut *true*. Autrement dit, le nombre 0, ou même un string ou une table vide valent *true*.

Les userdatas

Les userdatas sont la représentation en Lua des structures en C (il faut savoir que Lua est très lié au C, il possède par exemple une API complète de contrôle en C). Elles contiennent donc des variables. Et comme les fonctions sont des variables... Les userdatas peuvent être utilisées comme des classes d'objets pour une orientation POO de Lua
Vous trouverez donc des variables du style :

Code : Lua

```
Objet.attribut.sousAttribut    -- Un objet avec plein d'attributs
Objet.methode()                 -- Un objet avec une méthode
Objet:methode2()                -- Une autre façon d'appeler une
méthode
```

A vrai dire, pour que l'appel avec un seul point soit équivalent à celui avec les deux points, il faut passer en argument à *Objet.methode() self* ; en effet, l'appel *Objet:methode()* donne implicitement en argument l'objet parent, désigné par le mot-clé *self*.

Le nil

nil, c'est la valeur vide. Si une variable vaut nil, elle ne vaut rien. C'est un peu le *NULL* des pointeurs en C, sauf que *nil* s'utilise pour toutes les variables.

La doc officielle de Lua décrit aussi le type thread, qui représente un sous-processus du programme en Lua, mais il n'a pas d'utilité sur la Nintendo DS, qui a une architecture différente de celle d'un PC (et qui a surtout un processeur à 66MHz).

Les bases de la programmation en Lua (2)
Les opérateurs

Mathématiques

En Lua, il y a les cinq opérateurs de calcul "classiques" :

- addition : **+**
- soustraction : **-**
- division : **/**
- multiplication : *****
- modulo : **%**

Il y a aussi "^", pour représenter une puissance.

De comparaison

- Est égal à : **==**
- Est différent de : **~=**
- Est supérieur à : **>**
- Est inférieur à : **<**
- Est supérieur ou égal à : **>=**
- Est inférieur ou égal à : **<=**

L'opérateur d'égalité comporte DEUX signes égal "=". Notez aussi que contrairement à beaucoup de langages, l'opérateur "différent de" est "~=" et non pas "!=".

L'opérateur de longueur

Cet opérateur est particulier à Lua : le symbole dièse "#" représente la longueur d'une variable possédant des éléments, autrement dit la longueur d'une table ou d'un string.

Exemple :

Code : Lua

```
chaine = "bla bla"
tableau = {0, "Hey !", 36.12, fonction}

#chaine    -- Donne 7
#tableau   -- Donne 4
```

L'opérateur de concaténation

Ce n'est pas un "+" ; en Lua, on effectue une concaténation avec ".." :

Code : Lua

```
chaine1 = "Bonjour, "
chaine2 = "ça va ?"

chaine3 = chaine1.."est-ce que "..chaine2    -- chaine3 vaut
"Bonjour, est-ce que ça va ?"
```

Opérateurs logiques

Ce sont les mots anglais pour dire "ou", "et", et "non" :

- Ou : *or*
- Et : *and*
- Non (inverser le résultat de l'expression) : *not*

Les mots-clés des blocs d'instructions

Un bloc d'instruction est un ensemble d'instructions plus ou moins séparées des autres. Un bloc délimite aussi la "zone" dans laquelle une variable locale est accessible.

Blocs conditionnels

Une condition se construit de cette façon :

Code : Lua

```
if <condition> then         -- si <condition> alors
 <instructions>
elseif <condition2> then    -- sinon si <condition2> alors
 <instructions>
elseif <condition3> then
 <...>
else                        -- sinon
 <instructions>
end                         -- fin de la condition
```

Boucles

Il y a en tout trois types de boucles en Lua. Tout d'abord, deux premières que j'appellerai "classiques" :

Code : Lua

```
while <condition> do       -- tant que <condition> faire
 <instructions>
end                        -- fin de la boucle

repeat                     -- répéter
 <instructions>
until <condition>          -- jusqu'à ce que
```

Ces deux boucles sont sensiblement identiques, sauf que dans le cas de la seconde, la condition de sortie de la boucle est évaluée à la fin de chaque tour, et non au début. Ce qui signifie que, quoi qu'il arrive, les instructions de la boucle *repeat ... until* seront exécutées au moins une fois.

La troisième boucle est la boucle *for* :

Code : Lua

```
for <variable> [= <borne>, <borne>, <pas>] [in <iterateur>] do    --
Pour <variable> [qui prend les valeurs <borne> à <borne>, en allant
de <pas> en <pas>] [dans <iterateur>] faire
 <instructions>
end
```

Celle-ci est un peu particulière.
Tout d'abord, voyons le cas sans itérateur : à chaque tour de boucle, *variable* va prendre une nouvelle valeur (sa valeur précédente plus le pas, qui peut être négatif). On sort de la boucle dès que l'on a atteint la deuxième borne.

Et avec un itérateur...

Houlàààààà, c'est quoi un itérateur ?

Bonne question. Je vous enjoins à lire cette page (en anglais, désolé), qui vous explique le fonctionnement de la fonction *next* ; cette fonction permet de "traverser une table ou assimilé". En fait, à chaque fois que cette fonction est appelée, avec en argument une table, elle retourne la valeur suivante de la table ainsi que son index. On appelle ça un itérateur. Regardez aussi les fonctions *ipairs* et *pairs* (sur le même site).
Ainsi, pour comparer avec ce qu'on a dit plus haut, à chaque tour *variable* prend la valeur suivante dans la table. On sort de la boucle quand on a fait un tour de boucle avec la dernière valeur.

Enfin, sachez que toutes ces boucles peuvent être "cassées" à l'aide de l'instruction *break*, c'est-à-dire qu'elles sont arrêtées.

Il y a aussi le bloc d'instructions do <instructions> end, qui... ne fait rien si ce n'est exécuter les instructions. Il ne sert qu'à délimiter un bloc.

Les fonctions

Les fonctions, c'est tout simple :

Code : Lua

```
-- Déclaration de la fonction
function nomDeLaFonction(arguments, attendus)
 <instructions>
 return variable, autreVariable     -- instruction pour retourner
une voire PLUSIEURS variables
end

-- Appel de la fonction
retour1, retour2 = nomDeLaFonction(arg1, arg2)
```

Vous avez pu apercevoir une autre des libertés de µLua : une fonction peut retourner plusieurs valeurs. Mais elle peut très bien n'en retourner qu'une seule, ou même aucune

Bien, je crois n'avoir rien oublié. C'est un peu condensé, peut-être pas très clair, mais je dois vous apprendre µLua moi, pas le Lua ! Namého.
...
Quoi qu'il en soit, si vous avez besoin d'aide, deux sites utiles : la doc officielle de Lua (en anglais) et des tutoriels sur le wiki Lua (en anglais). Et bien sûr, le forum du SdZ

Et pour MicroLua ?

Et pour MicroLua, deux choses :

Détruisez vos variables à la fin de vos programmes !

Il faut savoir que notre pauvre DS ne possède pas beaucoup de mémoire, du moins pas assez pour la gâcher. Il faut donc la libérer à la fin de vos programmes, pour la laisser libre aux autres.

Vous vous souvenez de *nil* ? La "valeur rien". Et bien, c'est ici qu'elle va nous être utile.
En effet, pour détruire une variable, il suffit de faire :

Code : Lua

```
variable = nil
```

A la fin du code, et ce pour toutes les variables globales (les variables locales sont détruites automatiquement à la fin de leur bloc d'instructions). Il y a aussi des façons spécifiques à certains types de variables spéciales pour les détruire, qui seront décrites en temps voulu.
En fait, ça ne les détruit pas à proprement parler : le fait de les rendre "vides" fait comprendre à Lua qu'il peut les détruire en faisant ce que l'on appelle un *garbage collect* (littéralement "collecte d'ordure"). Cette action est réalisée de façon assez aléatoire, mais on peut la demander avec la fonction *collectgarbage("collect")*

D'autres types de variables

MicroLua définit d'autres types de variables relativement variés :

Color

Ouaip, ces variables contiennent une couleur. On verra ça plus loin...

Image

MicroLua définit un type pour les images. On verra ça plus loin...

Font

Ca, c'est une variable de police d'écriture. On verra ça plus loin...

Map et ScrollMap

Ces deux types de variables correspondent à des cartes et cartes défilantes. On verra ça plus... comment vous avez deviné ?

Canvas et Canvas object

Nous verrons plus loin ces variables qui correspondent respectivement à des canevas (des zones pour le dessin ultra-rapide) et des objets de canevas.
(vous voyez je change un peu)

DateTime

Objet représentant... une date ! On verra ça plus loin...

Voilà donc les types de variables spécifiques à MicroLua.
(vive le copier/coller ! \o/ Nan, je rigole)
Vous avez maintenant les bases de la programmation avec Lua, et ce qui en découle en µLua. C'est maintenant que ça va devenir vraiment intéressant. Enfin, ça l'était déjà avant, n'est-ce pas ?

Le BA.-Ba de MicroLua

Bon, maintenant que tout le monde est au point avec Lua, passons à µLua (parce que c'est pas tout ça, mais on n'a toujours rien vu sur µLua...) !

Faites chauffer Notepad++ ou Vim ou autre (j'suis pas raciste), c'est parti !

Généralités sur les fonctions de dessin

Description générale des fonctions de dessin

De manière générale, les fonctions de dessin de µLua se présentent de cette manière :

Code : Lua

```
screen.FONCTION(écran, coordonnées, contenu, couleur)
```

Dans l'ordre :

- *screen* : c'est la table qui contient les fonctions d'affichage primaires
- FONCTION : c'est le nom de la fonction (logique)
- écran : argument permettant de choisir l'écran de dessin
- coordonnées : pour que MicroLua sache où dessiner sur l'écran Ça peut aller d'un seul couple (x, y) (du texte par exemple) jusqu'à deux couples (lignes, rectangles...)
- contenu : disons que pour les fonctions qui ne se contentent pas de dessiner des formes, c'est l'élément principal (le contenu du texte, l'image à afficher, etc.)
- couleur : argument pour choisir la couleur du dessin

Repérage sur la surface de dessin

Quel écran ?

Déjà, pour afficher quelque chose, il faut savoir où le mettre.

Pour choisir entre les écrans, ça n'est pas bien compliqué (il n'y en a que deux après tout) :

- L'écran du haut est SCREEN_UP
- Celui du bas est SCREEN_DOWN

Ces variables prennent place dans l'argument *écran* vu ci-dessus.

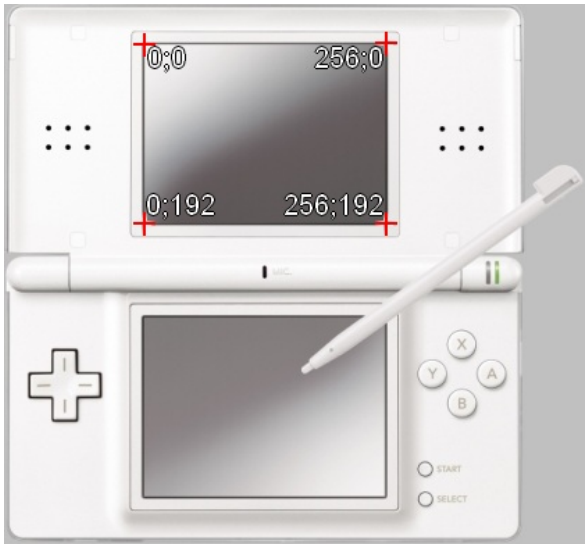
Ces variables sont en fait des "constantes" prédéfinies par µLua ("constantes" entre guillemets car il n'existe pas de vraies constantes en Lua ; disons que ce sont des mnémoniques plus pratiques mais plus long à utiliser que les chiffres directs). Toutes les "constantes" sont décrites au début de la doc.

Attention à bien taper ces variables tout en majuscules !

Repérage dans l'écran

Pour se situer sur les écrans de la DS, on utilise les abscisses et les ordonnées, en partant du coin en haut à gauche de l'écran. Ces écrans sont larges de 256 pixels et hauts de 192 pixels.

La démo en image :



(il va de soi que c'est la même chose pour l'écran du bas)

Vous remarquerez aussi que ce tuto ne brille pas par la qualité de ses images

Attends, y a un problème : si ça commence à zéro, et s'il y a 256px de long sur 192px de haut, les coordonnées maximales devraient être 255 et 191, et non pas 256 et 192 !

Oui mais non. Imaginez une feuille quadrillée (ou prenez-en une, comme ça vous pourrez voir par vous-mêmes) ; on prend une partie de cette feuille qui fait dix carreaux de long sur quatre de haut. Chaque carreau représente un pixel. Les coordonnées désignent logiquement les intersections des lignes ; à partir de là, je pense que vous voyez où je veux en venir : si le point tout en haut à gauche est (0, 0), alors le point tout en bas à droite est (10, 4), ce qui correspond bien à la taille du rectangle.

Convaincus ? Non ?
Bon, alors essayez ça (vous allez voir bientôt la signification de ce code) :

Code : Lua

```
while true do
 screen.drawRect(SCREEN_UP, 0, 0, 256, 192, Color.new(31, 31, 31))
 screen.drawRect(SCREEN_UP, 0, 0, 255, 191, Color.new(31, 0, 0))
 render()
end
```

Ce code dessine deux rectangles vides, l'un blanc et l'autre rouge, de coordonnées respectives (coin haut-gauche puis coin bas-droit) ((0, 0),(256, 192)), et ((0, 0),(255, 191)). Que remarque-t-on ? Que le rectangle blanc colle au bord de l'écran, contrairement au rectangle rouge

Maintenant que nous avons mis les choses au point, nous pouvons continuer.

Les couleurs

Pur gérer les couleurs, MicroLua utilise un type particulier : le type *Color*. Pour colorer vos dessins, vous devrez passer une variable de ce type à la fonction.

Afin de créer une telle variable, vous devez taper ceci :

Code : Lua

```
couleur = Color.new(r, g, b)
```

Chaque argument correspond à un masque de couleur (*r* pour *red*, ou rouge, *g* pour *green*, ou vert, et enfin *b* pour *blue*, ou bleu).

Une particularité de MicroLua est de devoir donner ces masques avec des valeurs allant de 0 à 31, plutôt que jusqu'à 255 comme souvent.

Voici une petite liste de quelques couleurs :

- bleu = *Color.new(0, 0, 31)*
- rouge = *Color.new(31, 0, 0)*
- vert = *Color.new(0, 31, 0)*
- orange = *Color.new(31, 17, 0)*
- noir = *Color.new(0, 0, 0)*
- blanc = *Color.new(31, 31, 31)*
- jaune = *Color.new(31, 31, 0)*

C'est quand même carrément pas pratique !

En fait, cette "limitation" vient de la µLibrary, qui code ses couleurs sur 5 bits au lieu de 8, donc on n'y peut rien.
Il n'empêche que c'est vrai, c'est donc pour ça qu'il existe aussi la fonction *Color.new256()*, qui requiert les mêmes arguments et s'utilise de la même façon, sauf que cette fois-ci vous pouvez donner des masques allant de 0 à 255 !

Et pour finir, une petite "astuce" (qui n'en est pas vraiment une sauf pour les néophytes de la programmation) : si vous n'utiliserez une couleur qu'une seule fois, pas la peine de créer une variable rien que pour ça. Vous pouvez donner le retour de la fonction *Color.new()* directement à l'instruction de dessin, ce qui donne :

Code : Lua

```
screen.FONCTION(ecran, coordonnées, contenu, Color.new(31, 0, 0))
```

Afficher du texte à l'écran

Il est donc temps de voir la fonction que vous utiliserez probablement le plus souvent : *screen.print()* !

Dis bonjour

Si vous avez compris le chapitre précédent, le simple fait de vous donner la structure de la fonction devrait vous permettre de l'utiliser (je devient fainéans) :

Code : Lua

```
screen.print(écran, x, y, texte[, couleur])
```

Le paramètre *couleur* est facultatif pour cette fonction, et **uniquement** pour cette fonction et celle que nous verrons juste après, d'où les crochets.

Et donc si vous voulez que votre console vous dise bonjour (sur l'écran du haut, tout en haut à gauche, mais bon finalement c'est comme vous voulez), ça va donner...

Secret (cliquez pour afficher)

J'espère que vous avez réfléchi un peu !

Code : Lua

```
screen.print(SCREEN_UP, 0, 0, "Bonjour toi !")
```

Et bien allez-y ! Écrivez cette ligne dans un fichier de test, allumez votre console, toussa, et testez pour voir si votre DS est devenue un peu plus polie.

Ça marche pas ! Tu nous as eus !

En fait si ça marche, c'est juste que ça dure une milliseconde grand maximum avant d'atteindre la fin du script et de revenir au shell. Ce qui me permet d'effectuer la transition vers...

La boucle principale !

C'est pas un peu moisi comme transition ?

Si, mais la suite va remonter le niveau

L'astuce si l'on peut dire, afin d'éviter que le résultat du programme ne disparaisse ~~dans les profondeurs insondables et pleines de microbes de l'oubli~~, c'est de tout placer dans une boucle qui va se répéter le plus vite possible. Cette boucle devient l'élément principal du programme. C'est le "*while not Keys.newPress.Start do*" que vous voyez depuis le début

Par conséquent, pour afficher votre message, le code sera :

Code : Lua

```
while not Keys.held.Start do
    screen.print(SCREEN_UP, 0, 0, "Bonjour toi !")
end
```

Toutes les instructions de votre programme devront être placée à l'intérieur de cette boucle, sauf évidemment les déclarations de variables et autres fonctions, et deux-trois choses particulières. Autrement dit, schématiquement, votre script principal ressemblera à ça :

Code : Lua

```
-- Inclusion de code depuis d'autres fichiers
-- Déclaration de variables
-- Déclaration de fonctions
-- Autres procédures à n'effectuer qu'au début du programme

while ... do

    -- Code principal

end

-- Autre procédures à n'effectuer qu'à la fin du programme
-- Destruction de variables, fonctions, etc.
```

Selon le langage que vous utilisiez avant, l'adaptation à ce "système" de programmation sera plus ou moins difficile.

Ça marche toujours pas !

Je sais, c'était prévu, car attention, seconde transition...

La fonction *render()* !

Il manque une fonction essentielle dans notre boucle principale : la fonction *render()*. Elle fait deux choses : commander l'affichage du résultat des fonctions de dessin, et synchroniser les tours de boucle avec les cycles d'affichage de notre console.

L'intérêt de la première, et c'est évident, c'est de demander à la NDS d'afficher ce qu'on vient de lui demander. Pour faire simple, ça valide les instructions et les envoie au GPU (processeur graphique) de la machine.
La seconde fonctionnalité est de nous permettre d'avoir un nombre d'images par seconde (FPS pour *frames per second* en anglais) constant. La NDS nous permet d'atteindre 60FPS, bien que pour des raisons techniques liées à la µLibrary, MicroLua tourne avec 30FPS. Cela dit, ça reste tout à fait jouable

Sinon, trêve de blabla, voilà le script finalisé (promis, là votre console vous dira vraiment bonjour) :

Code : Lua

```
while not Keys.held.Start do
    screen.print(SCREEN_UP, 0, 0, "Bonjour toi !")

    render()
end
```

En général, *render()* se trouvera toujours à la fin de votre boucle, étant donné que toutes les instructions de dessin placées après ne seront effectuées qu'à la prochaine boucle.

Zone de texte

Avant de voir les images, il faut connaître l'existence de la fonction

Code : Lua

```
screen.drawTextBox(ecran, x1, y1, x2, y2, texte[, couleur])
```

qui affiche le texte dans une zone de texte, qui est un rectangle dont le coin haut-gauche est aux coordonnées (x1, y1) et le coin bas-droit aux coordonnées (x2, y2).

Concrètement, ça fait quoi ? Ça affiche le texte en le limitant à la zone définie, et en effectuant un retour à la ligne quand le texte atteint le bord droit de la zone. Si le texte est trop long, il sera coupé (il n'y a pas d'ascenseur intégré aux TextBox). Cette fonction est très utile pour bien agencer des paragraphes.

Afficher des images à l'écran

Bon maintenant passons aux images  !

Chargement de l'image

Il vous faut d'abord savoir que la DS à deux mémoires :

- la VRAM (video RAM) (mémoire vidéo) qui est de 656 Ko
- la RAM (mémoire vive) qui est de 4Mo

La VRAM a peut-être beaucoup moins de place, mais c'est celle-là que vous devez utiliser pour vos images. Lorsqu'on charge une image dans la RAM, on a parfois (j'ai eu ) des problèmes. Par ailleurs, la VRAM est optimisée pour le dessin.

MicroLua supporte actuellement les formats PNG (transparence non supportée), JP(E)G et GIF (non animés, transparence supportée).

MicroLua ne gère actuellement la transparence que sur les GIF, pas sur les PNG. Cela dit, vous verrez plus loin qu'il est possible d'utiliser la "transparence alpha" sur les dessins.

D'abord voici la commande pour charger l'image :

Code : Lua

```
image = Image.load(chemin, destination)
```

Image.load() retourne donc une "variable image".

Le chemin de l'image peut-être un chemin relatif ou absolu et la destination est VRAM ou RAM (deux constantes, comme SCREEN_UP et SCREEN_DOWN).

Affichage de l'image

Maintenant affichons l'image, voici la commande :

Code : Lua

```
screen.blit(ecran, x, y, image)
```

N'oubliez pas de charger l'image avant.

Exemple :

Code : Lua

```
-- On charge l'image dans la VRAM
image = Image.load("image.png", VRAM)

while not Keys.held.Start do
```

```
        -- On affiche l'image en haut à gauche de l'écran tactile
        screen.blit(SCREEN_DOWN, 0, 0, image) -- Affiche l'image

        render()
    end

    -- Et on efface l'image de la mémoire
    Image.destroy(image)
    image = nil
```

D'abord on décharge l'image de la mémoire, PUIS on détruit sa variable, et pas l'inverse, sinon on ne peut plus la décharger !

screen.blit() nous cache des choses

Si vous jetez un coup d'œil à la documentation, vous verrez que celle-ci nous donne pour *screen.blit()* :

Citation : Documentation

Void screen.blit(ecran, x, y, image [,sourceX, sourceY] [, largeur, hauteur])

Blitte (affiche) une image à l'écran.

ecran (Number) : écran sur lequel afficher l'image (SCREEN_UP ou SCREEN_DOWN)
x (Number) : abscisse du coin haut-gauche de l'image
y (Number) : ordonnée du coin haut-gauche de l'image
image (Image) : image à blitter
sourceX, sourceY (Number) : coordonnées de la partie de l'image source à blitter
largeur, hauteur (Number): dimensions de la partie de l'image source à blitter

La fonction nous permet en effet de n'afficher qu'une partie de l'image ; cette partie commence aux coordonnées (sourceX, sourceY), et s'étend vers la droite et vers le bas, soit jusqu'aux limites de l'image, soit sur les dimensions données par *largeur* et *hauteur*. Ça n'est pas indispensable, mais je devais vous en parler 

Bonus

Et en cadeau deux petites fonctions :

Code : Lua

```
longueur = Image.width(image)
largeur = Image.height(image)
```

La première sert à connaître la longueur d'une image en pixels, la seconde à obtenir sa largeur.

Il y a d'autres choses à savoir à propos des images, mais ça fera l'objet d'une partie de la manipulation de l'affichage.

Utiliser les contrôles de la Nintendo DS
Interagir avec le joueur

Comme vous devez le savoir, la DS a un écran tactile, huit boutons (sans le *power* et le son) ainsi qu'un pad directionnel en croix qui en comporte quatre (et qu'on appelle *D-pad*).

Pour réagir en fonction des actions de l'utilisateur, on va utiliser les conditions. Mais d'abord, il faut savoir sur quoi l'utilisateur a appuyé !
C'est l'objectif de la fonction :

Code : Lua

```
Controls.read()
```

Elle est à appeler à chaque fois qu'il faut mettre à jour les contrôles pour réagir en fonction de l'utilisateur (c'est-à-dire au moins à chaque tour de boucle). En général, ce sera la première instruction de votre boucle, comme *render()* en sera probablement la dernière.

Les contrôles sous MicroLua

Maintenant, il faut tester l'état des boutons. Pour cela, les conditions vont porter sur des variables du type

Code : Lua

```
Keys.[Etat].[Bouton]
```

pour les boutons "classiques", ou bien de la forme

Code : Autre

```
Stylus.[Etat]
```

pour l'écran tactile.
Ces variables sont des booléens (elles valent donc *true* ou *false*).

Afin de les utiliser, il faut remplacer *[Etat]* et *[Bouton]* par les valeurs suivantes :

- [Bouton]
  - A : *A*
  - B : *B*
  - X : *X*
  - Y : *Y*
  - L : *L*
  - R : *R*
  - Start : *Start*
  - Select : *Select*
  - Flèche directionnelle haut : *Up*
  - Flèche directionnelle bas : *Down*
  - Flèche directionnelle gauche : *Left*
  - Flèche directionnelle droite : *Right*
- [Etat]
  - Etat appuyé : *held*
  - Etat relâché : *released*
  - Etat nouvelle pression : *newPress*

Sachez de plus qu'avec le Stylet, on peut récupérer quatre informations :

- Abscisse : *Stylus.X*
- Ordonnée : *Stylus.Y*
- Augmentation de X dans ce mouvement : *Stylus.deltaX*
- Augmentation de Y dans ce mouvement : *Stylus.deltaY*

Faites attention aux majuscules, elles peuvent être sources d'erreurs car indispensables (ça plante si vous n'en mettez pas !). Lua est sensible à la casse, et ça demande de la rigueur 

Exemple :

Code : Lua

```
Controls.read()

if Keys.newPress.A then

-- A faire si A est re-pressé

end
if Stylus.held then

-- A faire si le stylet est appuyé

end
```

Retour sur la boucle principale

Vous vous souvenez de la condition ? Maintenant, vous pouvez la comprendre :

Code : Lua

```
while not Keys.held.Start do
```

Elle signifie "tant qu'on n'appuie pas sur le bouton Start". Remarquez qu'elle est tout à fait équivalente à *Keys.newPress.Start* dans la cas présent.
Voilà, voilà... Passons maintenant aux fonctions de dessin primaires, partie essentielle de l'affichage !

Un peu d'art...

Ce chapitre vous présente les fonctions de dessin de MicroLua, mais aussi les Canevas. Un chapitre important donc, à lire avec attention

Le dessin classique

Pour dessiner, la façon la plus simple et intuitive est d'utiliser les fonctions associées à *screen*.

Toutes les fonctions que je vais aborder ici n'auront aucun effet si elles ne sont pas suivies de l'instruction *render()*, comme vu au chapitre précédent. Je ne la mettrai pas, à vous d'y penser

Dessine-moi... Un point.

On commence doucement
La fonction est de la forme :

Code : Lua

```
screen.drawPoint(ecran, x, y, couleur)
```

Notez bien que pour toutes ces fonctions, l'argument couleur est obligatoire, contrairement à screen.print().

Ce n'est pas bien compliqué, je pense que je n'ai rien à ajouter

Tracer une ligne

Code : Lua

```
screen.drawLine(ecran, x1, y1, x2, y2, couleur)
```

C'est à mon avis très simple à comprendre : on choisit l'écran, on indique les coordonnées de début (x1, y1) et les coordonnées de fin (x2, y2), et enfin on choisit une couleur.

Tracer des rectangles

Vous avez à votre disposition trois fonctions pour dessiner un rectangle.

Rectangle simple

La fonction

Code : Lua

```
screen.drawRect(ecran, x1, y1, x2, y2, couleur)
```

dessine un rectangle "vide", dont le coin haut-gauche se trouve en (x1, y1) et le coin bas-droite en (x2, y2).

Rectangle plein

La seconde fonction est :

Code : Lua

```
screen.drawFillRect(ecran, x1, y1, x2, y2, couleur)
```

Cette fonction dessine un rectangle plein de la couleur passée en argument. Les coordonnées suivent la même logique que *screen.drawRect()*.

Rectangle dégradé

La troisième et dernière fonction est la plus intéressante et la plus originale.

Code : Lua

```
screen.drawGradientRect(ecran, x1, y1, x2, y2, couleur1, couleur2,
couleur3, couleur4)
```

Hein ? Il y a quatre couleurs ?

Et oui ! Si vous avez une certaine connaissance de l'anglais, vous aurez certainement compris ce que fait cette fonction : elle affiche un rectangle dégradé.

C'est en fait très simple : chaque couleur correspond à un coin du rectangle, de cette façon :

Et si on veut un dégradé monochrome ou bichrome ?

Dans ce cas-là, il faut passer deux fois la même couleur dans les bons coins. Pour le monochrome, il va falloir passer la couleur du fond, ce qui empêche de faire un dégradé sur une image de façon propre...

Inverser les écrans

Un gros titre pour une petite fonction de rien du tout

Code : Lua

```
screen.switch()
```

Celle-ce permet d'échanger l'affichage des deux écrans. En fait, un unique appel de cette fonction inverse *SCREEN_UP* et *SCREEN_DOWN*. Par exemple, si vous faites

Code : Lua

```
screen.drawFillRect(SCREEN_UP, 0, 0, Color.new(31, 0, 0))
```

Vous dessinez sur tout l'écran du haut un rectangle rouge. Appelez *screen.switch()*, et le rectangle se dessinera en bas. Imaginons qu'ensuite vous fassiez

Code : Lua

```
screen.drawFillRect(SCREEN_DOWN, 0, 0, Color.new(0, 0, 31))
```

Vous dessineriez un rectangle bleu sur tout l'écran. Mais pas sur l'écran du bas comme indiqué ; le rectangle sera sur l'écran du haut à cause de *screen.switch()*. Vous pouvez tout remettre à l'endroit en appelant une deuxième fois la fonction.

Le dessin avec un Canevas

Présentation des Canevas

Donc, si j'ai bien compris, il y a deux façons de dessiner ? Mais pourquoi ?

C'est une bonne question ; à première vue, les fonctions de *screen* suffisent. C'est d'ailleurs celles-ci que vous utiliserez la plupart du temps.
L'intérêt du Canevas est situé dans un aspect plus technique :

- il offre une bien meilleure vitesse d'affichage que les fonctions classiques
- il permet de manipuler les dessins

Vous pouvez apprécier les capacités d'affichage des Canevas en exécutant l'exemple canevas dans le dossier "Examples" fourni dans le pack de téléchargement de MicroLua.

Ce qui paraît plus attrayant, c'est la "manipulation" des dessins. Mais d'abord, nous allons voir comment utiliser les Canevas.

Vie et mort d'un Canevas

Pour créer un canevas, rien de plus simple ; il suffit d'écrire en instruction :

Code : Lua

```
canevas = Canvas.new()
```

avant la boucle principale, sinon vous allez recréer le canevas à chaque tour, et ça ne marchera pas.

J'attire votre attention sur le fait que si le nom français est "canevas", le nom anglais est "canvas" (**sans le "e"**). Ceci peut être une source d'erreurs parfois difficiles à trouver.

La destruction d'un canevas est tout aussi simple :

Code : Lua

```
Canvas.destroy(canevas)
canevas = nil
```

On détruit le canevas à l'aide de *Canvas.destroy()*, fonction à laquelle on passe en argument le canevas à détruire. Il ne faut pas oublier ensuite de détruire la variable en elle-même, en y donnant la valeur *nil*.

Affichage

Maintenant, comment se passe l'affichage d'un canevas ?

On utilise la fonction *Canvas.draw()*, de cette façon :

Code : Lua

```
Canvas.draw(ecran, canevas, x, y)
```

C'est assez explicite, remarquez juste que l'on ne donne que les coordonnées du coin haut-gauche, ce qui signifie que le canevas remplira toute la partie de l'écran situé entre ce point et le coin inférieur droit de l'écran indiqué.

Dessiner sur un canevas

C'est bien beau tout ça, mais comment fait-on pour afficher du dessin ?

Cela se déroule en deux étapes : d'abord, on crée un objet Canevas, puis on l'ajoute au canevas approprié.
Je ne vais pas détailler toutes les fonctions de création d'objets canevas, elles sont tout à fait compréhensibles dans la doc et sont quasiment les mêmes que les "classiques".

Je vais donc plutôt décrire cet exemple :

Code : Lua

```
-- On crée l'objet canevas (ici un rectangle)
objet = Canvas.newRect(x1, y1, x2, y2, couleur)
-- On l'ajoute au canevas
Canvas.add(canevas, objet)
```

Comme vous le voyez, c'est très simple : on appelle une fonction *Canvas.newXXX()* selon l'objet que l'on veut ; cette fonction retourne un *Canvas Object* (c'est un type spécial de µLua), qui sert de *handler* (de gestionnaire) pour cet objet (vous verrez plus loin quelle en est l'utilité). Ensuite, on appelle *Canvas.add()*, qui ajoute l'objet au canevas spécifié.
Lors du prochain appel de la fonction *Canvas.draw()* (et après le *render()*), un rectangle des coordonnées et de la couleur demandées sera affiché.

Notez que, comme pour les couleurs, vous n'êtes pas obligé de passer par une variable : vous pouvez directement donner l'objet à la fonction *Canvas.add()* comme ceci :

Code : Lua

```
Canvas.add(canevas, Canvas.newRect(x1, y1, x2, y2, Color.new(31, 0,
0)))
```

Cependant, vous perdez l'intérêt des canevas : la manipulation des dessins !

Manipuler les dessins

Depuis le temps que j'en parle... On y arrive enfin !

Changer les caractéristiques des dessins

La grande utilité des Canevas est que les Canvas Objects sont modifiables. Concrètement, ça veut dire quoi ? Cela signifie que vous pouvez changer les coordonnées d'un rectangle, la couleur d'un texte, etc. Vous pouvez changer toutes les propriétés de tous les objets du canevas !

Pour changer une propriété, on utilise :

Code : Lua

```
Canvas.setAttr(objet, nomPropriete, nouvelleValeur)
```

C'est tout 

Et comment on connaît le nom de la propriété ?

Ces noms sont indiqués au début de la doc ; ils sont de la forme ATTR_XXX, où XXX va être :

- X1 : abscisse 1

- Y1 : ordonnée 1
- X2 : abscisse 2
- Y2 : ordonnée 2
- X3 : longueur de la zone à prendre dans la source (pour une Image)
- Y3 : hauteur de la zone à prendre dans la source (pour une Image)
- COLOR : couleur
- COLOR1 : couleur 1
- COLOR2 : couleur 2
- COLOR3 : couleur 3
- COLOR4 : couleur 4
- TEXT : texte
- IMAGE : image soure
- FONT : police spéciale
- VISIBLE : état visible ou non de l'objet

Bien entendu, tous les objets n'ont pas les mêmes attributs ; le tableau ci-dessous indique quels attributs possèdent tels objets.

Objet	X1	Y1	X2	Y2	X3	Y3	COLOR	COLOR1	COLOR2	COLOR3	COLOR4	TEXT	IMAGE	FONT	VISIBLE
Line	Coordonnée	Coordonnée	Coordonnée	Coordonnée	/	/	Couleur	/	/	/	/	/	/	/	Est visible
Point	Coordonnée	Coordonnée	Coordonnée	Coordonnée	/	/	Couleur	/	/	/	/	/	/	/	Est visible
Rect	Coordonnée	Coordonnée	Coordonnée	Coordonnée	/	/	Couleur	/	/	/	/	/	/	/	Est visible
FillRect	Coordonnée	Coordonnée	Coordonnée	Coordonnée	/	/	Couleur	/	/	/	/	/	/	/	Est visible
GradientRect	Coordonnée	Coordonnée	Coordonnée	Coordonnée	/	/	/	Couleur	Couleur	Couleur	Couleur	/	/	/	Est visible
Text	Coordonnée	Coordonnée	/	/	/	/	/	/	/	/	/	Texte	/	/	Est visible
TextFont	Coordonnée	Coordonnée	/	/	/	/	/	/	/	/	/	Texte	/	Police spéciale	Est visible
TextBox	Coordonnée	Coordonnée	Coordonnée	Coordonnée	/	/	/	/	/	/	/	Texte	/	/	Est visible
Point	Coordonnée	Coordonnée	Coordonnée dans l'image source	Coordonnée dans l'image source	Longueur dans la source	Hauteur dans la source	/	/	/	/	/	/	Ressource image	/	Est visible

Sachez aussi que vous pouvez récupérer la valeur d'un attribut grâce à :

Code : Lua

```
valeur = Canvas.getAttr(objet, attribut)
```

Placer un objet au premier plan

Si vous avez tout plein d'objets dans votre canevas, il peut être difficile de les organiser après les avoir affichés. Pour cela, vous pouvez remettre un dessin au premier plan : il sera donc affiché par dessus les autres.

La fonction pour ce faire est :

Code : Lua

```
Canvas.setObjOnTop(canevas, objet)
```

Enlever un objet d'un canevas

Si vous avez toujours tout plein d'objets dans votre canevas, vous pouvez aussi faire de l'ordre par le vide ; autrement dit, enlever un objet d'un canevas. Notez bien que ça ne détruira pas l'objet de la mémoire, pour cela un bon vieux = *nil* s'impose.

Code : Lua

```
Canvas.removeObj(canevas, objet)
```

Que choisir ?

Bon c'est bien, on a vu qu'on pouvait dessiner de deux manières assez différentes.
On a vu aussi que les canevas, c'était quand même largement plus intéressant que le dessin basique.

Alors pourquoi je vous explique quand même comment dessiner sans les canevas ? Et pourquoi on voit autant de scripts qui n'utilisent pas les canevas ?

Oui, c'est vrai ça, pourquoi ?

Chut, c'est moi qui pose les questions ici 

Et je donne aussi les réponses : parce que les programmeurs sont des flemmards. Et croyez-moi, entre écrire

Code : Lua

```
while not Keys.newPress.Start do
    Controls.read()

    screen.print(SCREEN_UP, 0, 0, "Comment ça va ?")

    render()
end
```

et

Code : Lua

```
canevas = Canvas.new()
Canvas.add(canevas, Canvas.newText(0, 0, "Comment ça va ?"))

while not Keys.newPress.Start do
    Controls.read()

    Canvas.draw(canevas, 0, 0)

    render()
end

Canvas.destroy(canevas)
canevas = nil
```

mon choix est très vite fait !

Disons que l'utilisation un peu lourde des canevas et leurs performances, les destinent à du "dessin de masse". La possibilité d'agir sur vos dessins facilement doit aussi faire partie des arguments en leur faveur.

En résumé, pour des scripts rapides, sans grande prétention graphique, vous utiliserez le dessin classique. Et dès que vous voudrez passer à quelque chose d'un peu plus soigné esthétiquement, vous vous tournerez vers les canevas (cela dit, pour les jeux, il y a d'autres fonctionnalités de MicroLua qui y sont dédiées, donc vous n'utiliserez pas les Canevas).

La transparence Alpha

Vous pouvez dessiner tout ce que vous voulez, mais il manque quand même quelque chose : la transparence.
Si MicroLua affiche correctement la transparence "totale", c'est-à-dire qu'il élimine le fuschia des sprites par exemple, vous ne pouvez pas afficher quelque chose de semi-transparent, de sorte que l'on voit les dessins qui sont derrière. C'est ce que nous allons aborder maintenant.

Les *layers*

Le système de transparence de MicroLua repose sur des *layers*, autrement dit des couches de dessin. Ces couches s'empilent au fur et à mesure que vous dessinez, et vous, vous voyez la pile depuis en haut. Pour les graphistes, considérez que ce sont des calques (sur le principe).

Chaque *layer* possède son propre **coefficient de transparence**, qui définit l'opacité de la couche. Un coefficient de 1 équivaut à un *layer* totalement transparent, tandis qu'une couche à 99 est opaque.

C'est carrément bizarre comme plage de valeurs !

Et bien, de base la µLibrary gère le coefficient de 0 à 31. Nous le prenons donc entre 0 et 99, cependant la valeur de 0 se contente d'afficher les contours du dessin ; par exemple, si vous dessinez un rectangle plein à 0, c'est comme si vous dessiniez un rectangle "vide". Notez que même si vous pouvez donner 0 en coefficient, MicroLua le "transforme" en 1.

Par ailleurs, les *layers* sont repérés par leur index numérique (en partant de 1). Notez qu'il y a un nombre maximal de *layers*, cependant il est suffisamment important pour que vous n'ayez aucun souci à vous faire de ce côté là

Dessine-moi un mouton transparent !

La fonction principale est

Code : Lua

```
screen.setAlpha(coefficient[, layer])
```

Dès lors que vous l'appelez, **tous** les dessins qui suivront seront assignés au layer dont vous aurez donné l'index et seront affectés du coefficient de transparence spécifié.

Si vous voulez réinitialiser le système d'affichage transparent, c'est-à-dire revenir à la première couche et dessiner totalement opaque, faites :

Code : Lua

```
screen.setAlpha(ALPHA_RESET)
```

Le layer est optionnel, vous comprenez pourquoi : on revient au premier
Je précise que µLua effectue cette instruction automatiquement dans la fonction *render()*.

Notez aussi que si vous n'avez aucun besoin de gérer vous-même les *layers*, vous pouvez laisser µLua s'en charger. En effet, vous l'aurez remarqué : le paramètre *layer* est optionnel. Si vous ne le mettez pas, ça passe tout simplement au suivant

D'autres infos

Vous pouvez cependant suivre le layer actuellement utilisé grâce à la fonction

Code : Lua

```
screen.getLayer()
```

Elle retourne l'index du prochain layer qui sera utilisé.

De plus, la fonction

Code : Lua

```
screen.getAlphaLevel()
```

Vous donnera le coefficient de transparence du layer actuel.

Exemple - TP

Pour vous expliquer plus clairement comment on utilise cette fonctionnalité, je vais vous guider dans la réalisation d'un petit script qui utilise la transparence.

Le principe

C'est très simple : on dessine deux rectangles sur l'écran du bas (vous les ferez comme vous voulez, tant qu'il y en a deux et qu'il ne se chevauchent pas entièrement ça va). L'un deux sera transparent avec un coefficient contenu dans la variable *coefficient*. L'autre aura le coefficient opposé ; c'est-à-dire que si le premier est à 50, l'autre est aussi à 50, si le premier est à 1, l'autre est à 99, et vice-versa.

Le coefficient sera réglé à l'aide des flèches Haut et Bas, et sera de plus affiché sur l'écran du haut (de la façon que vous voudrez). Sa valeur par défaut est d'ailleurs 50.

La base

Je vous donne le code à partir duquel vous allez faire le script :

Code : Lua

```
coefficient = 50

while not Keys.newPress.Start do
    Controls.read()

    -- Ce que vous allez coder sera ici.

    render()
end

coefficient = nil
```

La sélection du coefficient de transparence

On veut de façon logique augmenter le coefficient en appuyant sur la flèche Haut et le diminuer en appuyant sur la flèche Bas. N'oubliez pas qu'il doit être compris entre 1 et 99 inclus

Vous devriez donc avoir quelque d'équivalent à :

Secret (cliquez pour afficher)

Code : Lua

```
coefficient = 50

while not Keys.newPress.Start do
    Controls.read()

    if Keys.held.Up and coefficient < 99 then
        coefficient = coefficient + 1
    end
    if Keys.held.Down and coefficient > 1 then
        coefficient = coefficient - 1
    end

    render()
end

coefficient = nil
```

L'affichage du coefficient de transparence

Vous n'avez qu'à afficher *coefficient*, aucune difficulté

Secret (cliquez pour afficher)

Code : Lua

```
coefficient = 50

while not Keys.newPress.Start do
    Controls.read()

    if Keys.held.Up and coefficient < 99 then
        coefficient = coefficient + 1
    end
    if Keys.held.Down and coefficient > 1 then
        coefficient = coefficient - 1
    end

    screen.print(SCREEN_UP, 0, 0, "Coefficient de transparence :
"..coefficient)

    render()
end

coefficient = nil
```

Les deux rectangles

Voici donc la partie la plus intéressante, celle qui utilise la transparence.

D'abord, le rectangle transparent de coefficient donné par la variable (moi je le dessine sur la première moitié haute de l'écran du bas, et bleu, mais vous faites comme vous le sentez).
Je vous rappelle donc que vous réglez d'abord la transparence, et qu'ensuite vous dessinez ce que vous voulez.

Secret (cliquez pour afficher)

Code : Lua

```
coefficient = 50

while not Keys.newPress.Start do
    Controls.read()

    if Keys.held.Up and coefficient < 99 then
        coefficient = coefficient + 1
    end
    if Keys.held.Down and coefficient > 1 then
        coefficient = coefficient - 1
    end

    screen.print(SCREEN_UP, 0, 0, "Coefficient de transparence :
"..screen.getAlphaLevel())

    screen.setAlpha(coefficient)
    screen.drawFillRect(SCREEN_DOWN, 0, 0, 256, 96, Color.new(0,
0, 31))

    render()
end

coefficient = nil
```

Si vous avez trouvé ça, l'autre rectangle ne devrait pas poser de problèmes

Secret (cliquez pour afficher)

Code : Lua

```
coefficient = 50

while not Keys.newPress.Start do
    Controls.read()

    if Keys.held.Up and coefficient < 99 then
        coefficient = coefficient + 1
    end
    if Keys.held.Down and coefficient > 1 then
        coefficient = coefficient - 1
    end

    screen.print(SCREEN_UP, 0, 0, "Coefficient de transparence :
"..screen.getAlphaLevel())

    screen.setAlpha(coefficient)
    screen.drawFillRect(SCREEN_DOWN, 0, 0, 256, 96, Color.new(0,
0, 31))

    screen.setAlpha(100 - coefficient)
    screen.drawFillRect(SCREEN_DOWN, 0, 96, 256, 192, Color.new(0,
0, 31))

    render()
```

```
end

coefficient = nil
```

Bilan

Voici donc le script final :

Code : Lua

```
coefficient = 50    -- Coefficient de transparence, par défaut à 50

while not Keys.newPress.Start do
    Controls.read()

    -- Gestion de la modification du coefficient
    -- Il doit rester entre 1 et 99 !
    if Keys.held.Up and coefficient < 99 then
        coefficient = coefficient + 1
    end
    if Keys.held.Down and coefficient > 1 then
        coefficient = coefficient - 1
    end

    -- Affichage du coefficient
    screen.print(SCREEN_UP, 0, 0, "Coefficient de transparence :
"..screen.getAlphaLevel())

    -- Dessin du premier rectangle
    screen.setAlpha(coefficient)
    screen.drawFillRect(SCREEN_DOWN, 0, 0, 256, 96, Color.new(0, 0,
31))

    -- Dessin du second rectangle, de transparence "opposée" au
premier
    screen.setAlpha(100 - coefficient)
    screen.drawFillRect(SCREEN_DOWN, 0, 96, 256, 192, Color.new(0,
0, 31))

    render()
end

coefficient = nil
```

Croyez-moi, la transparence permet de faire de très jolis effets graphiques qui amélioreront considérablement l'aspect esthétique de vos homebrews

Maintenant, vous pouvez déjà faire de nombreuses choses avec MicroLua (et commencer à développer vos propres homebrews !).

Mais avant, je vous conseille de continuer votre cours vers... les TPs !

TPs à gogo

Vous pouvez déjà faire de très bonnes choses avec µLua ! Mais je vous conseille de suivre ces quelques TPs, il vous donneront un bon entraînement pour réutiliser tout ce que vous avez appris.

TP1 : L'image qui bouge

Dans ce premier TP vous allez devoir faire bouger une image grâce au stylet.

On va utiliser cette image (enfin, vous prenez ce que vous voulez !) :

(pour l'enregistrer, faites Clic droit - Enregistrer (l'image) sous...)

On réfléchit d'abord !

Avant de se lancer ~~corps et âme~~ dans le code, on va réfléchir un peu.

On va faire bouger l'image en fonction des déplacements du stylet, donc les coordonnées de l'image vont bouger. Le plus simple est donc de stocker l'abscisse et l'ordonnée de l'image dans des variables.
Les valeurs de ces variables seront donc les coordonnées du stylet.

N'oubliez pas d'initialiser les variables à zéro avant la boucle !

En fait, ça n'est pas vraiment gênant si vous ne le faites pas, mais c'est une bonne habitude à prendre (par exemple si vous essayez un jour un langage plus typé que Lua).

Maintenant, vous pouvez lâcher le codeur qui est en vous !

3, 2, 1, Partez !

C'est bon ?

C'était pas trop dur, non ?
Allez voici ma correction :

Secret (cliquez pour afficher)

Code : Lua

```
image = Image.load("tp.png", VRAM) -- On charge l'image
x = 0 -- On crée les variables abscisse et ordonnee
y = 0

while not Keys.held.Start do -- On crée la boucle de pause
    Controls.read() -- On met à jour les commandes

    if Stylus.held then -- La condition "si le stylet est appuyé"
    -- Alors les variables x et y prennent la valeur des
coordonnées du stylet

        x = Stylus.X
        y = Stylus.Y
    end

    screen.blit(SCREEN_DOWN, x, y, image) -- On affiche l'image

    render()
end

-- On efface l'image de la mémoire
Image.destroy(image)
image = nil

-- On efface nos variables abscisse et ordonnee
x = nil
y = nil
```

Si vous n'avez pas fait la même chose (enfin, ça doit pas être bien différent) mais que ça marche, c'est bien ! Si vous n'avez pas réussi, relisez ce code pour bien le comprendre, et votre cours.

Des améliorations

Si le cœur vous en dit, vous pouvez très bien améliorer ce script :

- On peut aller plus vite, et sans passer par des variables ! (réfléchissez-y avant de regarder)

Secret (cliquez pour afficher)

Code : Lua

```
image = Image.load("tp.png", VRAM)

while not Keys.newPress.Start do
    Controls.read()

    screen.blit(SCREEN_DOWN, Stylus.X, Stylus.Y, image)

    render()
end

Image.destroy(image)
image = nil
```

- Ce script est pas mal, mais ça serait peut-être mieux si on pouvait bouger l'image par son centre ?
  Aide :

  Secret (cliquez pour afficher)

  Aidez-vous de Image.width() et Image.height() !

  Secret (cliquez pour afficher)

  Code : Lua

```
image = Image.load("tp.png", VRAM)

while not Keys.newPress.Start do
    Controls.read()

    screen.blit(SCREEN_DOWN, Stylus.X - Image.width(image) /
2, Stylus.Y - Image.height(image) / 2, image)

    render()
end

Image.destroy(image)
image = nil
```

  Ce code repose sur une astuce tout bête : on obtient le centre de l'image avec la moitié de la longueur et de la largeur ; pour bouger le centre, on additionne ces moitiés aux coordonnées correspondantes.
- Allez, et on peut aussi, pourquoi pas, afficher les coordonnées !
  Bon, là je ne vous mets pas de correction, vous êtes grands hein ?

TP2 : Un morpion

En deuxième TP, je vous propose le classique, mais néanmoins efficace jeu du morpion.

Ce TP m'a été signalé comme assez ardu plus au niveau algorithmique que pour l'application de ce qu'on a vu. C'est pourquoi il est normal que vous puissiez ne pas le faire d'un seul coup, surtout si vous avez lu le tuto depuis le début d'une seule traite. Essayez un peu, et si ça vient pas faites autre chose avant d'y revenir (vous pouvez toujours le passer bien sûr, ça n'est qu'un TP, mais ça vous montre que vous pouvez déjà faire de bonnes choses). Et surtout, ne vous découragez pas !

Etude préliminaire

Avant de coder, on réfléchit un peu...

Déroulement général

Pour ceux qui ne connaîtraient pas le principe, le but du morpion est d'aligner trois croix (ou trois ronds) dans une grille de 3*3 cases, horizontalement, verticalement, ou en diagonale. Les deux joueurs jouent chacun leur tour. Les joueurs posent le premier symbole à tour de rôle.
On va convenir que le premier joueur a les croix, le second a les ronds.
On va se contenter d'un mode Deux joueurs, parce que c'est déjà un bon entraînement que de faire ça, et ensuite, la conception d'une IA est beaucoup plus longue. De plus, l'algorithmique n'est pas le but de ce TP.

Maintenant, comment va se dérouler exactement une partie ? A chaque tour, le joueur qui commence pose son symbole dans la case qu'il veut. Puis c'est à l'autre joueur. Dès que trois symboles sont alignés, la partie s'arrête.

Point de vue technique

Je pense que le plus simple pour choisir l'endroit où l'on met sa croix ou son rond est encore l'utilisation de l'écran tactile
Le joueur va toucher une case pour désigner l'endroit où il veut poser le symbole. Il faudra donc déterminer à partir des coordonnées du stylet la case désignée SI le stylet est appuyé (j'insiste là-dessus, sans cette précaution ça peut faire des choses bizarres).

Si vous avez visité le forum µLua, vous aurez certainement entendu parler de la lib StylusBox de killer01. Elle fait exactement ce que l'on va faire ici, cependant je pense qu'il est bien plus intéressant de le faire à la main (et c'est aussi plus rapide, si vous trouvez la bonne façon de faire).

Cependant, il faut aussi vérifier que la case n'est pas déjà prise. Ce qui nous amène à une autre question : comment stocker dans des variables les symboles ? On va utiliser ce que l'on appelle un tableau à deux dimensions.

Gné ? Cékoissa ?

C'est en fait très simple : on a une table, dont chaque élément va, par exemple, représenter une ligne (on peut très bien faire par colonne). Dans notre cas, cette table aura donc trois éléments. Cependant, chaque ligne a elle-même trois cases. Les éléments de la première table seront donc des tables de trois cases chacune. Vous avez compris ?

On revient au stockage des symboles : les cases contiendront des Strings, "X" pour les croix, et "O" pour les ronds.
Si le stylet est pressé entre telle et telle valeur en ordonnée, c'est qu'on est dans telle ligne. S'il est pressé entre telle et telle valeur en abscisse, c'est qu'on est dans telle case. Et maintenant qu'on sait quelle case a été choisie... je vous laisse faire

Bon, d'accord. C'était pas si compliqué. Mais j'ai une question : comment on sait si on a aligné trois symboles ?

Ah, ça je vous laisse réfléchir
Si vraiment vous ne trouvez pas, il y a un peu d'aide plus bas... Mais n'allez pas voir tout de suite !

Et l'affichage de la grille, et des symboles ?

Cessez de poser tant de questions ! Vous tracez quatre lignes, vous faites deux boucles l'une dans l'autre, et c'est bon !

A vos éditeurs de texte !

L'heure est venue de coder !

Prenez votre temps, ce script est relativement simple à mettre en place. Si vous bloquez, réfléchissez bien à la façon dont se déroule la partie.

Vous ne trouvez pas comment vérifier si trois symboles sont alignés ? Vous avez pourtant réfléchi toute la journée, sans succès ? Vous êtes désespéré au point de vous jeter par la fenêtre ? Alors avant de sauter, regardez plutôt ça :

Secret (cliquez pour afficher)

Il y a, pour chaque symbole, huit configurations possibles de victoire :

- la diagonale haut-gauche / bas-droite, qui correspond donc à la première case de la première ligne, la seconde case de la seconde ligne, et la troisième case de la troisième ligne
- la diagonale haut-droite / bas-gauche, qui correspond à la troisième case de la première ligne, etc.
- les lignes
- les colonnes (première case de la première ligne, première case de la seconde ligne, etc.)

Quand le joueur qui a les croix a fini de jouer, on regarde si des croix correspondent à cette configuration ; et pareil pour les ronds.
Si avec ça vous ne trouvez pas...

3, 2, 1, Partez !

Correction

son de cloche C'est fini !

Alors, vous y êtes arrivés ?
Voici ma version du code ; bien sûr, si vous n'avez pas la même mais que ça marche, c'est bien !

Code : Lua

```
symboles = {{"", "", ""}, {"", "", ""}, {"", "", ""}}
joueur = "X"
quitter = false

-- Couleur
blanc = Color.new(31, 31, 31)
bleu = Color.new(0, 0, 31)
rouge = Color.new(31, 0, 0)


function alignes(symboles, joueur)
    -- On inverse le symbole du joueur reçu, car on reçoit le
joueur suivant, pas celui qui vient de jouer
    local joueurPrecedent = ""

    if joueur == "X" then
        joueurPrecedent = "O"
    else
        joueurPrecedent = "X"
    end

    -- Teste si on a un alignement sur une ligne
    for i, ligne in ipairs(symboles) do
        if ligne[1] == joueurPrecedent and ligne[2] ==
joueurPrecedent and ligne[3] == joueurPrecedent then
            return true
        end
    end
    -- Teste si on a un alignement sur une colonne
    for i = 1, 3 do
        if symboles[1][i] == joueurPrecedent and symboles[2][i] ==
joueurPrecedent and symboles[3][i] == joueurPrecedent then
            return true
```

```
        end
    end
    -- Teste si on a un alignement sur la diagonale haut-gauche /
bas-droite
    if symboles[1][1] == joueurPrecedent and symboles[2][2] ==
joueurPrecedent and symboles[3][3] == joueurPrecedent then
        return true
    end
    -- Teste si on a un alignement sur la diagonale haut-droite /
bas-gauche
    if symboles[1][3] == joueurPrecedent and symboles[2][2] ==
joueurPrecedent and symboles[3][1] == joueurPrecedent then
        return true
    end
    -- Si on est arrivé jusqu'ici, c'est qu'on n'a pas trouvé
d'alignement, donc on renvoie false
    return false
end


while not quitter and not alignes(symboles, joueur) do
    --[[\\ AFFICHAGES DIVERS //]]--

    -- Notez que j'ai fait des cases de 20*20, c'est totalement
arbitraire
    -- J'admets qu'un caractère fait 5px de large sur 8px de haut

    -- Affichage des lignes verticales
    screen.drawLine(SCREEN_DOWN, 20, 0, 20, 60, blanc)
    screen.drawLine(SCREEN_DOWN, 40, 0, 40, 60, blanc)

    -- Affichage des lignes horizontales
    screen.drawLine(SCREEN_DOWN, 0, 20, 60, 20, blanc)
    screen.drawLine(SCREEN_DOWN, 0, 40, 60, 40, blanc)

    -- Affichage des symboles
    for i, elem in ipairs(symboles) do              -- Utiliser la
boucle for de cette façon est un choix personnel, on aurait tout
aussi bien pu faire "for i = 1, 3 do"
        for j = 1, 3 do         -- Pour chaque case de la ligne...
            screen.print(SCREEN_DOWN, 7+20*(j-1), 6+20*(i-1),
elem[j])  -- Les calculs bizarres, c'est pour centrer à peu près les
symboles dans les cases (c'est pas grave si vous ne les comprenez
pas, mais c'est à mon sens la seule façon de les centrer)
        end
    end

    -- Affichage du joueur qui doit jouer
    screen.print(SCREEN_DOWN, 0, 100, "C'est au joueur des
"..joueur.." de jouer.", bleu)

    screen.print(SCREEN_DOWN, 0, 180, "Appuyer sur Start pour
quitter.")

    render()
    --[[\\ AFFICHAGES DIVERS //]]--

    --[[\\ GESTION DES EVENEMENTS //]]--
    Controls.read()

    if Stylus.held then
        for y = 1, 3 do
            if Stylus.Y >= 20*(y-1) and Stylus.Y < 20*y then
                for x = 1, 3 do
                    if Stylus.X >= 20*(x-1) and Stylus.X < 20*x then
                        if symboles[y][x] == "" then
                            symboles[y][x] = joueur
                            if joueur == "X" then
                                joueur = "O"
                            else
                                joueur = "X"
                            end
                        end
                    end
                end
            end
        end
    end

    if Keys.newPress.Start then
        quitter = true
    end
    --[[\\ GESTION DES EVENEMENTS //]]--
end

-- Si on sort de la boucle, c'est qu'un joueur a gagné ou qu'on
veut quitter
while not quitter and not Keys.newPress.Start do
    Controls.read()

    local joueurPrecedent = ""

    if joueur == "X" then
        joueurPrecedent = "O"
    else
        joueurPrecedent = "X"
    end

    -- Affichage des lignes verticales
    screen.drawLine(SCREEN_DOWN, 20, 0, 20, 60, blanc)
    screen.drawLine(SCREEN_DOWN, 40, 0, 40, 60, blanc)

    -- Affichage des lignes horizontales
```

```
        screen.drawLine(SCREEN_DOWN, 0, 20, 60, 20, blanc)
        screen.drawLine(SCREEN_DOWN, 0, 40, 60, 40, blanc)

        -- Affichage des symboles
        for i, elem in ipairs(symboles) do
            for j = 1, 3 do
                screen.print(SCREEN_DOWN, 7+20*(j-1), 6+20*(i-1),
    elem[j])
            end
        end

        -- Affichage du gagnant
        screen.print(SCREEN_DOWN, 0, 100, "LE JOUEUR DES
    "..joueurPrecedent.." A GAGNE !", rouge)

        render()
    end


    symboles = nil
    joueur = nil
    blanc = nil
    bleu = nil
    rouge = nil
```

Voilà, voilà.
Pour déterminer l'endroit où l'on choisit de mettre le symbole, la première chose qui vient à l'esprit est de faire plusieurs if, mais c'est long, et on répète plusieurs fois la même chose ; et ça, en programmation, saymal. Alors que si l'on réfléchit un minimum, on s'aperçoit qu'un schéma est visible. On n'a plus qu'à le mettre en place avec deux boucles !

Mine de rien, avec le peu qu'on a appris, on peut faire beaucoup de choses !

Améliorations

On ne fait pas de TP sans améliorations à proposer !

- Faites en sorte que l'on puisse rejouer après avoir fini la partie. N'oubliez pas qu'il faut changer de premier joueur à chaque partie !
- Mettez en place un système de score ; chaque partie gagnée rapporte un point
- Vous pouvez aussi améliorer l'esthétisme de ce morpion (parce que là, j'veux pas dire, mais c'est moche)
- Si vous vous en sentez le courage, programmez une IA ! C'est un excellent exercice de réflexion
- Et tant que vous y êtes, concevez un menu pour choisir entre le mode Un joueur et Deux joueurs !

Vous avez vu, c'est pas trop difficile si on connaît bien son cours, non ?

Allez, vous pouvez vous reposer. Vous avez bien travaillé !

Voilà, vous connaissez maintenant les bases de Microlua !
Vous pouvez déjà faire des petits jeux ou programmes. C'est en essayant plein de choses que vous vous améliorerez !

Partie 2 : Tout ce qu'il faut pour un bon jeu

Afficher des images, du texte, dessiner... C'est déjà pas mal, mais µLua offre beaucoup plus que ça !
Alors si vous avez en tête un projet vidéo ludique, poursuivez votre lecture par ici

Les Maps et ScrollMaps

Mettons qu'il vous est passé par la tête de faire un super RPG avec en héros principal un elfe tout habillé de vert (toute assimilation à un célèbre jeu de Nintendo est fortuite bien entendu). Première question : comment allez-vous afficher le décor ?
C'est là que les *maps* (ou *cartes* en bon français) interviennent : elles vont vous permettre de créer l'univers de votre jeu rien qu'avec deux fichiers, et sans vous prendre la tête !

Préparation de la map

Présentation

Tout d'abord, il faut savoir ce qu'est une map.

Une *map*, c'est une carte (traduction littérale du mot anglais). Et cette carte est constituée de *tiles* (ou tuiles en français), qui correspondent en fait à des cases.

Dans les jeux vidéo, ça ressemble à ça :

Le décor (le sol, les arbres, tout) est géré par une map (enfin, disons que les maps correspondent à ça, Oracle of Ages a pas été codé en MicroLua). Les personnages que l'on voit par dessus, ce sont des sprites, et on s'y intéressera juste après.

Avec MicroLua, un map est constituée de deux fichiers : une image PNG, qui contient les images des tuiles, et un fichier *.map*, qui contient la structure de la carte. On y reviendra.
Du point de vue du script ensuite : la map peut être modifiée (on peut changer une tuile depuis le programme très facilement) et bougée (on peut la faire défiler). On l'affiche et la détruit comme l'on manipulerait une image.

Les images des tiles

Premièrement, il nous faut un fichier qui contient les images des tiles.

C'est un simple fichier PNG de taille carrée, qui a en général comme longueur et largeur des multiples de 8. La raison est que les écrans de la DS ont eux-mêmes des longueurs multiples de 8 (256 / 8 = 32 et 192 / 8 = 24), ainsi en affichant la map, on est sûr qu'elle prendra bien tout l'espace.

Le fichier des tiles tel qu'il est dans l'exemple (dans votre dossier "Examples") est celui-ci :

Ce fichier contient donc les tiles. Ce sont des carrés en général de 8*8px (vous savez maintenant pourquoi).
Si vous ne prenez pas tout l'espace de l'image avec vos tuiles, ça n'est pas grave, vous pouvez laissez l'espace vide tel quel.

Le fichier .map

Mais qu'est-ce donc que cet obscur fichier .map ?
Ça n'est rien de plus qu'un simple fichier texte qui contient comme je l'ai dit plus haut la structure de la map. Et comment il fait ?

Chaque ligne représente une ligne de la carte. Et chaque ligne est constituée de chiffres, séparés par un *pipe* (caractère sur la touche du chiffre 6 sur la partie alphanumérique, accessible via Alt Gr. + 6). Les chiffres correspondent aux numéros des tiles dans le fichier PNG. Notez que la numérotation commence à 0, et se fait de gauche à droite sur chaque ligne.

En schéma c'est mieux :

Ce fichier ressemble donc à quelque chose comme :

Code : Autre

```
1|2|3|1|2|3|1|2|3|1|2|3|1|2|3|
1|2|3|1|2|3|1|2|3|1|2|3|1|2|3|
1|2|3|1|2|3|1|2|3|1|2|3|1|2|3|
1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|
1|2|3|1|2|3|1|2|3|1|2|3|1|2|3|
1|2|3|1|2|3|1|2|3|1|2|3|1|2|3|
1|2|3|1|2|3|1|0|0|1|2|3|1|2|3|
1|1|1|1|1|1|1|0|0|1|1|1|1|1|1|
1|2|3|1|2|3|1|2|3|1|2|3|1|2|3|
1|2|3|1|2|3|1|2|3|1|2|3|1|2|3|
1|2|3|1|2|3|1|2|3|1|2|3|1|2|3|
1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|
1|2|3|1|2|3|1|2|3|1|2|3|1|2|3|
1|2|3|1|2|3|1|2|3|1|2|3|1|2|3|
1|2|3|1|2|3|1|2|3|1|2|3|1|2|3|
```

Pour la suite de ce tutoriel, et pour ceux qui voudront tester les fonctions étudiées, vous pouvez utiliser l'exemple fourni dans le dossier "Examples/maps", qui contient un fichier map et son fichier PNG associé.

Utilisation dans vos programmes

Vie et mort de la map

Première chose à faire pour utiliser notre map : la charger !

D'abord, on charge le fichier PNG comme une image :

Code : Lua

```lua
tiles = Image.load("tiles.png", VRAM)
```

Ensuite, on charge le fichier map :

Code : Lua

```lua
map = Map.new(tiles, "map.map", 15, 15, 32, 32)
```

"15, 15", ce sont la longueur et la largeur de la map, en **tiles**.
"32 , 32", ce sont les dimensions d'une tile, en **pixels**.

Comme vous le voyez, utiliser des tiles carrées n'est pas une obligation, mais c'est en général ce qui est utilisé.

Pour détruire la map, on fait comme cela :

Code : Lua

```lua
map = Map.destroy(map)
map = nil
```

N'oubliez pas de détruire aussi l'image des tiles 

Notre code pour afficher la map en est donc à :

Code : Lua

```lua
tiles = Image.load("tiles.png", VRAM)
map = Map.new(tiles, "map.map", 15, 15, 32, 32)

while not Keys.newPress.Start do
    Controls.read()

    -- Affichage de la map
    -- Et autres fonctions pour agir dessus

    render()
end

map = Map.destroy(map)
map = nil
Image.destroy(tiles)
tiles = nil
```

Affichage de la map

C'est la fonction :

Code : Lua

```lua
Map.draw(SCREEN_DOWN, map, 10, 10, 5, 5)
```

Avec dans l'ordre : l'écran, la map à afficher, les coordonnées X et Y du coin haut-gauche de la map, et le nombre de tiles à afficher, sur la longueur et sur la hauteur. Vous remarquez donc que la map occupe tout l'espace de l'écran se trouvant en bas et à droite du point donné.

Etat de notre script :

Code : Lua

```
tiles = Image.load("tiles.png", VRAM)
map = Map.new(tiles, "map.map", 15, 15, 32, 32)

while not Keys.newPress.Start do
    Controls.read()

    Map.draw(SCREEN_DOWN, map, 10, 10, 5, 5)
    -- Et autres fonctions pour agir dessus

    render()
end

map = Map.destroy(map)
map = nil
Image.destroy(tiles)
tiles = nil
```

Manipulation de la map

Il est possible de modifier deux choses dans une carte :

- l'espace entre les tiles :
  Code : Lua

  ```
  Map.space(map, longueur, hauteur)
  ```

  Très facile donc, on donne l'espace sur la longueur (entre deux tiles voisines horizontalement) et sur la hauteur (entre deux tiles voisines verticalement), tout ça en pixels bien sûr.
- la valeur d'une tile :
  Code : Lua

  ```
  Map.setTile(map, x, y, valeur)
  ```

  On donne à la fonction la map, puis le numéro de la colonne, puis de la ligne de la tile à changer ; enfin, on passe en dernier argument le nouveau numéro de la tile.
  Autrement dit, si on veut changer la toute première tile pour y mettre la troisième, il faut faire :
  Code : Lua

  ```
  Map.setTile(map, 0, 0, 3)
  ```

  Cette fonction **ne modifie pas le fichier map**, ce changement ne durera donc que le temps du script.

Notez qu'il existe la fonction Map.getTile(), qui à l'image de setTile(), permet d'obtenir la valeur d'une tuile.
Code : Lua

```
Map.getTile(map, x, y)
```

Pour l'exemple, on reprend notre script, et on veut faire en sorte de changer la première tile de la map pour y mettre la troisième quand le joueur appuie sur A :
Code : Lua

```
tiles = Image.load("tiles.png", VRAM)
map = Map.new(tiles, "map.map", 15, 15, 32, 32)

while not Keys.newPress.Start do
    Controls.read()

    Map.draw(SCREEN_DOWN, map, 10, 10, 5, 5)

    if Keys.newPress.A then
        Map.setTile(map, 0, 0, 3)
    end

    render()
end

map = Map.destroy(map)
map = nil
Image.destroy(tiles)
tiles = nil
```

Faire défiler la map

Ah, enfin quelque chose d'utile ! 

D'abord, petite explication sur comment les jeux avec un système de map sont faits : au centre de l'écran, le sprite du personnage, qui ne bougera jamais (sauf dans certains cas, à la limite de la carte par exemple, ou bien comme dans les vieux Zelda sur une map fixe ). L'image change suivant les actions du joueur (vue de dos, pendant qu'il attaque, etc.). Seule la carte bouge. Par

exemple, si on appuie sur la flèche droite, le personnage semble aller à droite ; en fait, c'est la carte qui défile vers la gauche.

Pour faire défiler notre map, on utilise la fonction :

Code : Lua

```
Map.scroll(carte, x, y)
```

"x" représente le **nombre de tiles** à faire défiler horizontalement, et "y" le **nombre de tiles** verticalement.
Bien entendu, x et y peuvent être négatif : si x est négatif, alors la carte défile vers la gauche, et si y est négatif, elle défile vers le haut.

Et si nous faisions le code qui va faire défiler cette carte ? Je vous laisse faire, ça n'est pas très dur, faites juste attention au sens du défilement
Pour être sur la même longueur d'onde, je vais simuler le déplacement d'un sprite sur la carte (même si on n'affiche aucun personnage). Ça signifie que le D-pad va diriger **le sprite** et non pas le sens du mouvement de la carte.

Secret (cliquez pour afficher)

Code : Lua

```
tiles = Image.load("tiles.png", VRAM)
map = Map.new(tiles, "map.map", 15, 15, 32, 32)

while not Keys.newPress.Start do
    Controls.read()

    Map.draw(SCREEN_DOWN, map, 10, 10, 5, 5)

    if Keys.newPress.A then
        Map.setTile(map, 0, 0, 3)
    end

    if Keys.newPress.Left then
        Map.scroll(map, 1, 0)
    end
    if Keys.newPress.Right then
        Map.scroll(map, -1, 0)
    end
    if Keys.newPress.Up then
        Map.scroll(map, 0, 1)
    end
    if Keys.newPress.Down then
        Map.scroll(map, 0, -1)
    end

    render()
end

map = Map.destroy(map)
map = nil
Image.destroy(tiles)
tiles = nil
```

Toutes ces fonctions demandent plein de nombres, certains en pixels et d'autres en tiles. Faites bien attention, sinon vos instructions pourraient faire n'importe quoi

Les Scrollmaps

Tu as parlé de Scrollmaps dans le titre. C'est quoi la différence avec les maps normales ?

C'est un peu comme le dessin classique et les canevas : les scrollmaps sont plus rapides à afficher, et on peut les faire *scroller* (défiler) pixel par pixel, alors que les maps "classiques" ne permettent le défilement que tile par tile. Cependant, on y perd aussi les fonctions *setTile()*, *getTile()* et *space()*. De plus, elles ne peuvent être affichées qu'en plein écran.

Pour les fonctions qui restent, les différences sont donc que ce n'est plus *Map.fonction()* mais *ScrollMap.fonction()* (c'est évident).
La fonction *ScrollMap.draw()* ne demande plus que la map à afficher (puisque l'on est forcé de l'afficher en plein écran), et *ScrollMap.scroll()* demande des valeurs **en pixels et non plus en tiles**.

A vous de voir ce qui vous intéresse le plus ; pour ma part, je pense que c'est bien de gagner la rapidité d'affichage, mais seulement si l'on est sûr que l'on n'aura pas besoin de modifier la composition de la map pendant le programme ; à mon avis, ce sont les principales caractéristiques des deux types de maps.
Et voilà, vous savez maintenant comment utiliser les maps dans vos homebrews !

Je ne peux cependant pas finir le chapitre sans vous parler du Tile Engine de thermo_nono, basé sur celui de Transdiv, tous deux membres de la communauté µLua. Reprenant donc le système des maps, il permet aussi de gérer des cases infranchissables et les directions en diagonales. Cependant, il est délaissé pour le moment, et comporte malheureusement un bug très gênant concernant les tiles infranchissables. Je vous conseille de vous rendre ici (topic officiel sur le forum MicroLua) si vous voulez en savoir plus.

Les sprites

Grâce aux Maps, vous savez comment gérer l'univers de votre jeu. Mais encore faut-il pouvoir afficher les personnages ! C'est là que les Sprites interviennent.

Qu'est-ce qu'un sprite ?

Il convient en premier lieu d'expliquer un peu ce qu'on entend par "sprite" avec MicroLua.

Voici la définition que nous livre la noble encyclopédie Wikipédia :

Citation : Wikipédia : Article 'Sprite (jeu vidéo)'

> Un sprite (en français lutin) est dans le jeu vidéo un élément graphique qui peut se déplacer sur l'écran. En principe, un sprite est en partie transparent, et il peut être animé (c'est-à-dire qu'il est formé de plusieurs bitmaps qui s'affichent les uns après les autres).

Concrètement, un sprite se compose de plusieurs images (issues d'un fichier image découpé, un peu comme les maps) ; en changeant l'image affichée, on crée une petite animation, d'un personnage par exemple.
Chaque image qui compose le sprite s'appelle une *frame* (mot anglais qui désigne ici une image ; les gamers connaissent ce terme par les FPS, ou *Frames per Second* (images par seconde)). Avec MicroLua, il est possible d'afficher une seule frame "fixe", ou bien de composer une animation de plusieurs frames choisies individuellement avant de l'afficher, de la mettre en pause, etc.

L'image des frames se comporte comme celle d'une Map ; par conséquent, après avoir chargé le sprite (prochain chapitre), on accède à ses frames par un numéro de frame, qui commence à 0.

Comme le dit si bien Wikipédia, "en principe un sprite est en partie transparent". Et pour avoir une partie transparente, µLua définit une couleur qui sera rendue transparente à l'affichage. Comme pour beaucoup de programmes, c'est le rose fuchsia qui est utilisé. C'est la couleur RVB (255, 0, 255) (en hexadécimal *#FF00FF*). Elle est utilisée pour être transparente car elle est en général peut présente dans les dessins.

Maintenant, on va voir comment justement charger un sprite, et comment afficher une de ses frames. A l'attaque ! 

On commence doucement

Charger le sprite

On commence logiquement par charger le sprite en mémoire. On utilise pour cela la fonction :

Code : Lua

```
sprite = Sprite.new("fichier_frames.png", longueur_frame,
hauteur_frame, destination)
```

On charge donc le fichier "fichier_frames.png", qui contient des frames qui font *longueur_frame* pixels par *hauteur_frame* pixels.
Enfin, on donne comme pour une image la destination du chargement (RAM ou VRAM).
Ça retourne un "objet sprite" (une table en fait), sur laquelle on va utiliser des méthodes (des fonctions).

Le fichier sprite des exemples ressemble à ceci :

Première méthode que l'on va essayer :

sprite:drawFrame(ecran, x, y, numFrame)

Certainement la plus simple, drawFrame() permet d'afficher une simple frame à l'écran. On l'utilise...

 STOOOOOP ! C'est quoi c't'appel de fonction bizarre avec des deux-points ?

C'est comme ça qu'on appelle en général une méthode en Lua. Tout le jargon de méthode et objet appartient à la POO (Programmation Orientée Objet). C'est un principe de programmation ; comme le but de ce tuto n'est pas de vous l'expliquer, je vous renvoie au cours de M@teo21 sur le C++, et plus précisément sur la partie expliquant ce qu'est la POO.
Je disais donc, que c'est comme cela qu'on appelle une méthode en Lua. On en verra d'autres dans ce cours, et vous serez très certainement amenés à en utiliser dans vos programmes étant donné que beaucoup de libs sont faites en POO 

Ou j'en étais moi ?... Ah oui ! On l'utilise donc en remplaçant "sprite" par le nom de la variable qui nous sert à garder le retour de la fonction Sprite.new(). On donne l'écran sur lequel on affiche, les coordonnées, et le numéro de la frame que l'on veut afficher. Et cette frame s'affiche (au prochain render() bien entendu) !

Mais on se retrouve vite limité avec un simple affichage d'une seule frame ; c'est pour cela que MicroLua permet de constituer des animations.

Les animations de sprites

Ajouter une animation au sprite

Tout commence par là.

Code : Lua

```
sprite:addAnimation(table_frames, delai)
```

table_frames est donc... une table, on suit un peu au fond ! Bref, c'est donc une table qui contient les numéros des frames dont se compose l'animation. Je rappelle que la numérotation des frames commence à 0.
delai est le délai (en millièmes de seconde) entre l'affichage de chaque frame.

Maintenant, on peut afficher l'animation.

Bouton play

On utilise la fonction :

Code : Lua

```
sprite:playAnimation(ecran, x, y, numAnim)
```

x et y sont bien sûr les coordonnées de l'animation. *numAnim* désigne le numéro de l'animation à jouer. En effet, un sprite peut avoir plusieurs animations : imaginez un personnage, il peut marcher vers le haut, le bas, etc., mais aussi attaquer, jeter un sort... Bref, il faut donner le numéro de l'animation de ce sprite.

La numérotation commence ici à 1, contrairement à celle des frames !

Soumettez-la à votre volonté

Je parle bien sûr de l'animation, pas de votre copine/amie/femme/soeur. 

Vous pouvez arrêter l'animation :

Code : Lua

```
sprite:stopAnimation(numAnim)
```

Et la remettre au début :

Code : Lua

```
sprite:resetAnimation(numAnim)
```

Et pour la relancer :

Code : Lua

```
sprite:startAnimation(numAnim)
```

Si vous pensez confondre *playAnimation()* et *startAnimation()* (car il faut bien dire que pour nous pauvres francophones, les deux veulent dire la même chose), dites vous que *playAnimation()* est le screen.blit() des Sprites, tandis que *startAnimation()* et compagnie sont comme les fonctions des Timers (que vous n'avez pas encore vues, c'est pas faux( ).

Et en bonus...

MicroLua fournit la fonction

Code : Lua

```
sprite:isAnimationAtEnd(numAnim)
```

qui renvoie *true* si l'animation a affiché sa dernière frame. Ça peut être utile pour coordonner des sortes de "cinématiques" ou certains effets en parallèle de l'image.

C'est pas fini !

On n'aurait pas oublié quelque chose ? Un Sprite, c'est une image non ? Donc logiquement, il faut vider la mémoire. On fait presque comme pour une image, à ceci-près que ici *destroy()* est une méthode. On a donc :

Code : Lua

```
sprite:destroy()
sprite = nil
```

Exemple

Cette partie est là pour vous montrer *in situ* comment ça marche. Merci à gaetan.tux pour cet exemple.

Code : Lua

```lua
-- animation d'un sprite par gaetan.tux
-- GNU/GPL
-- le sprite doit être placé dans le dossier lua (perso.png)
-- presser START pour quitter

function init()
 -- Chargement de l'image du sprite
 perso = Sprite.new('/lua/perso.png',32,48,VRAM)

 -- Direction du perso (utilisé pour le choix de l'animation)
 -- 1 : bas | 2 : gauche | 3 : haut | 4 : droite
 direction = 1
 -- Position du sprite sur l'écran
 x, y = 0, 0
 -- Détection de la pression d'une touche
 touche = false

 -- Ajout des animations, avec un intervalle de 200ms entre chaque
frame
 perso:addAnimation ({0,1,2,3},200) -- Marcher vers le bas
 perso:addAnimation ({4,5,6,7},200) -- Marcher vers la gauche
 perso:addAnimation ({12,13,14,15},200) -- Marcher vers le haut
 perso:addAnimation ({8,9,10,11},200) -- Marcher vers la droite
end
----------------------------- getInfos ()
function getInfos()
 -- Retourne la direction du sprite, le déplacement qu'il doit
effectuer (augmentation de ses coordonnées),
 -- et l'indication de la pression d'une touche
 touche = true
 if Keys.held.Down then     -- Si on appuie sur la touche bas...
  direction, y = 1, y+2  -- Alors la direction est 1, et le sprite
doit descendre
 elseif Keys.held.Left then   -- Sinon, si on appuie sur la touche
gauche...
  direction, x = 2, x-2  -- Alors etc.
 elseif Keys.held.Up then
  direction, y = 3, y-2
 elseif Keys.held.Right then
  direction, x = 4, x+2
 else      -- Sinon...
  touche = false   -- On signale qu'aucune touche n'est pressée
 end

 return direction, x, y, touche
end
----------------------------- anim() + affichage des FPS
function anim()
 -- On affiche le sprite
 perso:playAnimation (SCREEN_UP,x,y,direction)
 if touche then     -- Si le sprite doit bouger, on lance
l'animation (playAnimation le fait déjà, mais il
  perso:startAnimation (direction) -- faut quand même faire
startAnimation si l'animation a été réinitialisée, et donc arrêtée)
 else      -- Sinon, on remet l'animation à zéro pour le prochain
déplacement
  perso:resetAnimation (direction) -- (resetAnimation arrête aussi
l'animation)
 end
 screen.print(SCREEN_DOWN,0,0,'FPS : '..NB_FPS)  -- On affiche le
nombre d'images par seconde
end
----------------------------- programme principal
init ()                                                -- On
initialise les variables avec la fonction init()
while not Keys.held.Start do                           -- Boucle
tant qu'on appuie pas sur start
 Controls.read ()                                  -- On regarde si on
appuie sur une touche
 direction, x, y, touche = getInfos()              -- Fonction
getInfos() pour la position du personnage
 anim()                                            -- On anime l'image
avec la fonction anim()
 render()                                          -- Enfin on
actualise les écrans
end                                                    -- Toutes
les fonctions ont été définies plus haut
----------------------------- fin, nil, destroy
perso:destroy ()                                       -- On
détruit le sprite du personnage
x, y, direction, perso = nil, nil, nil, nil            -- Puis on
fait disparaître les variables pour libérer la RAM
```

Je pense qu'il n'y a rien de bien compliqué là-dedans 
Vous savez désormais bien utiliser les Sprites.
Passons sans plus tarder au son ! 

Le son

Votre jeu est magnifique avec plein de sprites partout qui défilent sur une Map tout aussi belle. Mais il manque quelque chose... Du son !

La gestion du son par MicroLua

MicroLua permet de jouer du son, mais c'est loin d'être aussi simple, parce que MicroLua utilise la célèbre librairie MixMod.

Les Mods et les SFX

µLua fait une distinction entre ce que l'on appelle "Mods" et "SFX". Les premiers sont du son en général, une musique d'ambiance par exemple, qui est jouée longtemps. Il ne peut y en avoir qu'un à la fois. Les seconds en revanche, sont destinés à n'être joués que ponctuellement ; en effet, comme leur nom l'indique (ou pas), ce sont des effets spéciaux sonores (*Special effects* en anglais, essayez de le dire à voix haute pour comprendre pourquoi SFX). De plus, contrairement aux Mods, ont peut en jouer plusieurs à la fois.

Partant de là, µLua détermine la nature du son en fonction du fichier utilisé :

- fichiers MOD, XM, S3M, IT pour les Mods
- fichiers WAV pour les SFX

Quoi, c'est tout ?

Et bien oui, même pas de lecture des MP3 ou des OGG... Mais actuellement, ça nous suffit Et de toute façon, pour un jeu, ce sont ces formats (plus ou moins) qui sont utilisés, les MP3 ou OGG sont trop lourds à stocker et/ou décoder.

Préparation des sources

Cette gestion du son a un (très ?) gros défaut : µLua ne lit pas directement les fichiers sonores.

Et comment il fait ?

Il fait, c'est déjà ça

En fait, il faut passer par votre ordinateur, et un utilitaire tout prêt fait par Risike (voyez l'annexe concerné pour en savoir plus). Cet utilitaire crée un fichier *soundbank.bin* à partir de vos fichiers musicaux (ainsi qu'un fichier *soundbank.h* qui ne sert à rien après la création du .bin, si ce n'est vous aider à vous repérer dans les ID de vos sons ; on y reviendra). Ce .bin est un condensé de vos sons, et c'est lui qu'utilise µLua.

Les index

Dans la banque son, µLua utilise des index pour se repérer. Ceux-ci commencent à **0**, et les index des Mods et des SFX sont distincts.
Afin de ne pas vous perdre, µLua crée le fichier *soundbank.h* qui contient la liste des fichiers sons chargés ainsi que l'index correspondant.

Voici un exemple de fichier *soundbank.h* :

Code : C

```
#define SFX_AMBULANCE                   0
#define SFX_BOOM                        1
#define MOD_KEYG_SUBTONAL               0
#define MOD_PURPLE_MOTION_INSPIRATION   1
#define MOD_REZ_MONDAY                  2
#define MSL_NSONGS                      3
#define MSL_NSAMPS                      67
#define MSL_BANKSIZE                    70
```

Il est tiré des exemples de votre package

Vous avez à chaque ligne un préfixe "MOD" (pour les Mods) ou "SFX" (pour les SFX) suivi du nom du fichier. Il y a aussi le préfixe "MSL", pour des informations utilisées durant la conception de *soundbank.bin*.

Voilà, maintenant que tout est dit, on peut y aller !

Manipuler la banque son

C'est un peu comme pour les images : la banque doit être chargée en mémoire puis déchargée.

En fait, ce n'est pas exactement ça. µLua ne charge jamais entièrement la banque son en mémoire, parce que ça la surchargerait inutilement. Je suppose que ce que l'on considère comme son chargement est un accès à un "index" permettant à µlua de se repérer entre les sons. Quoi qu'il en soit, les sons en eux-mêmes ne sont chargés que ponctuellement (et manuellement).

Code : Lua

```
Sound.loadBank("soundbank.bin")

-- Votre code

Sound.unloadBank()
```

C'est tout. Bien entendu, là où j'ai mis "soundbank.bin", vous devez mettre le chemin d'accès à votre banque son, qui ne se nomme pas nécessairement *soundbank.bin*.
Notez que vous pouvez utiliser différentes banques de son plutôt qu'une seule énorme (pour séparer les sons par thème par exemple). Vous ne pouvez cependant en utiliser deux en même temps, vous devez donc d'abord décharger la première avant d'utiliser la seconde.

Les Mods

Description du Mod

Un Mod (pour *module*) est un fichier musical contenant des échantillons sonores, et des "patrons" indiquant comme ces échantillons doivent être joués. Ils sont utilisés dans les jeux vidéo pour leur simplicité d'utilisation (ils sont rapides à lire).

Charger et détruire le Mod

Les dév se sont pas embêtés, pour ça on a :

Code : Lua

```
Sound.loadMod(index)

-- Votre code

Sound.unloadMod(index)
```

Ces deux fonctions, qui doivent bien sûr être appelées entre le chargement et le déchargement de la banque son correspondante, demandent à chaque fois l'index du Mod en question.
Bien que l'on soit limités à lire un seul Mod à la fois, on peut très bien en charger plusieurs en même temps ; en effet, même la fonction de déchargement demande l'index du Mod, de plus vous allez voir tout de suite que la lecture d'un Mod requiert son index.

Play, pause, toussa...

Code : Lua

```
-- Lance le Mod
Sound.startMod(index, modeLecture)

-- Met le Mod en pause (logique oui :) )
Sound.pause()

-- Reprend la lecture du Mod
Sound.resume()

-- Arrête la lecture du Mod
Sound.stop()
```

Voici donc les fonctions principales pour la lecture des Mods.
Sound.startMod() demande ainsi l'index du Mod, mais aussi un mode de lecture, qui doit être *PLAY_ONCE* pour le jouer une seule fois, ou *PLAY_LOOP* pour le jouer en boucle.
Vous remarquerez que les autres fonctions ne demandent pas d'index, ce qui est logique puisque comme je vous l'ai déjà précisé plus haut on ne lit qu'un Mod à la fois.

Et en plus de ça, une petite fonction qui vous permettra de savoir si un Mod est en cours de lecture :

Code : Lua

```
Sound.isActive()
```

Elle retourne un Booléen : *true* si la lecture est en cours, et donc *false* si elle ne l'est pas.

Manipuler le Mod

Vous pouvez changer la position de la lecture dans le Mod, le volume, mais aussi le tempo et le ton. Je dois avouer que ce genre de réglages me dépasse un peu, donc je vais me référer à la doc 

Code : Lua

```
-- Pour changer la position du "curseur" de lecture dans le Mod
Sound.setPosition(index, position)

-- Pour changer le volume
Sound.setModVolume(volume)

-- Pour changer le tempo
Sound.setModTempo(tempo)

-- Pour changer le ton du Mod
Sound.setModPitch(ton)
```

C'est à la fois simple et explicite je crois.
Je dois cependant préciser que le **volume doit être compris entre 0 et 1024**, et que le **tempo doit se trouver entre 512 et 2048**.

Nous pouvons donc passer aux SFX, vous allez voir c'est quasiment la même chose

Les SFX

Qu'est-ce qu'un SFX ?

Je vais reprendre ce que j'ai dit plus haut : un SFX est un "effet spécial sonore", joué ponctuellement. Ça signifie aussi qu'ils sont courts (quelques secondes), mais répétés plusieurs fois à intervalles variés.

Charger et détruire un SFX

Comme pour les Mods, nous avons :

Code : Lua

```
Sound.loadSFX(index)

-- Votre code

Sound.unloadSFX(index)
```

Voilà, c'est pareil

Play, pause, toussa...²

Bon, en fait c'était juste pour faire le parallèle avec les Mods, parce qu'on n'a que deux fonctions :

Code : Lua

```
handle = Sound.startSFX(index)

Sound.stopSFX(handle)
```

C'est quoi ce "handle" renvoyé par la première fonction ?

Ce handle est une référence renvoyée par µLua pour nous permettre de modifier individuellement les SFX. Bien sûr, il aurait été possible d'utiliser les index, mais avec les handles, on peut lire un même SFX plusieurs fois en même temps. N'oubliez donc pas ce handle, sans quoi vous ne pourriez pas agir sur votre SFX plus tard. J'en profite pour vous faire remarquer (si ce n'est pas déjà fait) que handle est une variable... A détruire à la fin (quand vous n'en aurez plus besoin bien sûr) !

Avant de passer à la suite, voici une fonction relativement utile :

Code : Lua

```
Sound.stopAllSFX()
```

Comme son nom l'indique, elle arrête la lecture de tous les SFX. Il va sans dire que c'est mieux que de faire dix fois *Sound.stopSFX()*, même avec une boucle

Manipuler le ~~Mod~~ SFX

Saleté de Mod, il s'incruste partout...

Nous avons donc la possibilité de changer le volume d'un SFX, son ton et même sa gamme de ton, mais aussi sa balance. Bien entendu, tout ceci est applicable à un SFX en particulier plutôt qu'à tous les SFX

Code : Lua

```
-- Changer le volume
Sound.setSFXVolume(handle, volume)

-- Changer le ton
Sound.setSFXPitch(handle, ton)

-- Changer la gamme
Sound.setSFXScalePitch(handle, gamme)

-- Changer la balance
Sound.setSFXPanning(handle, balance)
```

C'est limpide non ?
Sound.setSFXPanning() demande pour la balance une valeur entre 0 (tout à gauche) et 255 (tout à droite) ; par défaut, le SFX est donc joué à 128 (au "milieu").

La valeur maximale du volume d'un SFX est de **255** ! Rappelez-vous-en

Et bien voilà, maintenant vous pouvez donner à vos homebrews une ambiance digne de ce nom.

Fonctions spéciales pour linkers spéciaux

Pour achever cette partie plutôt dédiée aux jeux, je vais vous présenter des fonctionnalités que probablement peu parmi vous auront la chance d'expérimenter : le *motion* et la vibration.

Ça secoue !

Première fonctionnalité que je vais aborder ici : la vibration !

Prérequis

Vous vous en doutez bien, la console ne vibre pas toute seule. Notre chère petite Nintendo DS n'intègre pas de série un circuit de vibration ; ce sont les cartouches des jeux commerciaux qui le contiennent.
Mais si MicroLua vous dit qu'il peut faire vibrer votre DS, c'est qu'il peut ! En effet, certains linkers possèdent cette fonction *rumble*, ce sont des linkers pour le slot 2 d'ailleurs (trop gros pour les petites cartouches de slot 1).

Oh, ma DS vibre, j'ai dû recevoir un SMS !

et oui, je fais aussi dans la blague pas chère

Bon, niveau vibration, ça casse pas non plus trois pattes à un canard. On n'a que deux fonctions :

Code : Lua

```
Rumble.isInserted()
```

Celle-ci va vous dire si un pack de vibration est détecté dans le slot 2 (*true* pour un kit vibration inséré, *false* dans l'autre cas). Évidemment, cette fonction est indispensable, sinon quand vous voudrez demander à la console de vibrer et qu'il n'y aura pas de linker compatible, elle va vous crier dessus en disant qu'elle peut pas

Et la fonction essentielle :

Code : Lua

```
Rumble.set(etat)
```

Rien de plus, rien de moins. Pour allumer la vibration, donnez *true* à la fonction ; pour l'arrêter, enlevez le rumble pack...

Hein ?

C'est bien, vous suivez Oui, j'essaie de meubler, sinon cette partie va être très vide...
Pour l'arrêter, vous passez bien sûr *false* à la fonction.

Exemple

Un petit exemple pour voir à quoi peut ressembler un code avec le rumble.
C'est très simple : si on détecte un rumble pack, si le joueur appuie sur A, on envoie la sauce ; s'il appuie sur B, on arrête.

Code : Lua

```
while not Keys.newPress.Start do
    Controls.read()

    if Rumble.isInserted() then
        screen.print(SCREEN_DOWN, 0, 0, "C'est bien, tu peux tester
la vibration avec MicroLua !")
        screen.print(SCREEN_UP, 0, 0, "Appuie sur A pour faire
vibrer ta DS !")
        screen.print(SCREEN_UP, 0, 0, "Et pour arrêter, enlève le
rumble pack, ou alors appuie sur B ;)")

        if Keys.newPress.A then
            Rumble.set(true)
        end
    else
        screen.print(SCREEN_DOWN, 0, 0, "Bouh, tu peux pas faire
vibrer ta DS :(")
    end

    if Keys.newPress.B then
        Rumble.set(false)
    end

    render()
end
```

Voilà, c'est tout pour le rumble, passons maintenant au motion (carrément plus intéressant) !

Bougez avec votre DS !

Prérequis

Comme pour le rumble, le motion (la détection de mouvement) nécessite un linker adapté. Cette fonctionnalité est disponible sur des linkers slot 1, mais je pense qu'il existe des slot 2 qui possèdent à la fois le rumble et le motion.
Notez quand même que la "détection de mouvements" correspond ici à la détection de l'inclinaison de la console.

Je dois vous avouer une chose : contrairement au rumble, je n'ai jamais testé la détection de mouvements, étant donné que je n'ai pas de linker adéquat. Ça veut dire que tout ce que je vous sort là est une adaptation un peu plus agréable à lire de la documentation.


Ouh le mauvais prof !

Chut.

Pour commencer

La première chose à faire, c'est d'initialiser la fonction de détection de mouvements. Pour ça :

Code : Lua

```
Motion.init()
```

Cette fonction est 2-en-1 ! Elle détecte d'abord si vous avez un linker compatible, et renvoie *true* le cas échéant (et bien sûr *false* si elle n'a rien détecté). Ensuite, elle effectue l'initialisation.

Ce genre de système a besoin d'être calibré, pour cela, vous avez la fonction :

Code : Lua

```
Motion.calibrate()
```

Que calibre-t-elle exactement ? Aucune idée, je vous l'ai dit, je ne peux pas vérifier, mais je suppose que quand vous appelez cette fonction, le script définit la position "plane, au repos" par celle de la console à cet instant.

Détection de l'inclinaison

Vous avez pour cela trois fonction :

Code : Lua

```
Motion.readX()
```

Et l'équivalent pour Y et pour Z (trois fois la même fonction, c'est deux fois de trop à écrire).
Les fonctions retournent l'inclinaison sur les axes concernés (je suis pas sûr de savoir comment il s'y prend pour Z... Bref). Et comme je ne peux pas essayer, je ne sais même pas dans quelle fourchette sont situées les valeurs !

Par ailleurs, un peu comme pour *Stylus.deltaX*, vous avez trois fonctions pour obtenir l'accélération sur les trois axes :

Code : Lua

```
Motion.accelerationX()
```

Détection de ???

Là, je préfère ne rien dire que de dire n'importe quoi. La documentation précise deux fonctions de plus :

Code : Lua

```
Motion.readGyro()    -- Retourne la valeur de gyro du motion (?)
Motion.rotation()    -- Retourne la valeur de rotation du motion (?²
)
```

Je lance donc un appel à témoins : je remercierais toute les personnes pouvant le faire, de me renseigner sur les fonctions du motion, et notamment sur les deux dernières.

Quoi qu'il en soit, voilà tout ce que je peux dire sur le motion, qui s'il n'est pas clair à l'utilisation, a le mérite d'exister !
Ainsi s'achève cette dernière partie, pour transformer votre console en portable dernier cri ! (ben ouais, elle vibre et elle détecte les mouvements...)

Désormais, vous avez toutes les clés en main pour faire de vrais chefs-d'œuvres de homebrews !

Partie 3 : Tout le reste

Pour beaucoup, le contenu des deux premières parties devrait être suffisant. Cependant, MicroLua vous cache encore quelques fonctionnalités particulièrement intéressantes !

Les fonctions System et les fichiers INI

MicroLua propose des fonctions adaptées à la DS pour naviguer dans les dossiers et manipuler les fichiers.
Notez que les fonctions d'ouverture de fichiers, de modification de leur contenu, etc. sont celles fournies par Lua, à savoir *io.open()*, etc. Vous pouvez regarder ici (en anglais) pour lire la doc officielle de ces fonctions.

Les fichiers

µLua permet de faire deux choses : renommer et supprimer un fichier.

System.rename()

Pour renommer un fichier ou un dossier vide, la fonction est System.rename(), utilisée de cette façon :

Code : Lua

```
System.rename(ancienNom, nouveauNom)
```

Je pense que je n'ai rien de plus à dire là-dessus

System.remove()

C'est donc la fonction de suppression de fichier ou de dossier vide. Le seul argument requis est (logiquement) le nom du fichier ou du dossier vide à supprimer.
Rien de plus à dire là non plus

Il y a en fait plus de fonctions au niveau des dossiers, ce que nous allons voir maintenant.

Les dossiers

System.makeDirectory()

Ça, c'est la fonction pour créer un dossier. Tout ce que vous avez à faire, c'est de lui donner le nom du dossier.

A partir de là, on va voir des fonctions pour naviguer et se repérer dans les dossiers, ce qui est bien plus intéressant

System.changeDirectory()

Cette fonction sert à se déplacer dans l'arborescence. On lui donne un chemin, et on y est

Ce chemin peut très bien être relatif (c'est-à-dire par rapport à la position actuelle) ou absolu (c'est-à-dire par rapport à la racine du linker). Notez que la notation est de type Unix, par conséquent on utilise les slashes "/" et non pas les anti-slashes de Windows "\". La racine (le "premier grand dossier" qui contient tous les autres) est désignée par le slash "/" au début du chemin. Vous pouvez aussi utiliser "fat:/", ce qui revient au même. Enfin, sachez que "." (un point seul) correspond au dossier courant, et que ".." (deux points simples) correspond au dossier qui contient le dossier courant.

System.currentDirectory()

Fuaaaa, j'me suis perduuuuuu

Rien de grave, utilisez System.currentDirectory(), cette fonction vous retournera le dossier en cours (chemin absolu).

System.listDirectory()

Cette fonction retourne, sous forme de table, la liste de tous les fichiers et dossiers du dossier passé en argument (là aussi, ça peut être un chemin relatif ou absolu).

A moins d'être à la racine, les deux premières entrées de la table seront toujours "." (dossier courant) et ".." (dossier contenant le dossier courant). D'ailleurs, sachez que la fonction classe automatiquement les dossiers au début et les fichiers à la fin.
La table contient en fait des tables, qui possèdent elles mêmes deux entrées : l'une correspond au nom du fichier (ou dossier), et l'autre est un Booléen qui permet de savoir si c'est un fichier ou un dossier.

Code : Lua

```
files = System.listDirectory(System.currentDirectory())    -- On
obtient la liste du contenu du dossier courant

while not Keys.newPress.Start do
    for i, elem in pairs(files) do    -- C'est à mon sens la façon
la plus maligne d'exploiter la liste
        if elem.isDir then    -- Si c'est un dossier...
            screen.print(SCREEN_UP, 0, i*8-8, elem.name,
Color.new(31, 0, 0))    -- On affiche son nom en rouge
        else    -- Si au contraire c'est un fichier...
            screen.print(SCREEN_UP, 0, i*8-8, elem.name,
Color.new(0, 0, 31))    -- On affiche son nom en bleu
        end
```

```
        end
        render()
    end

    files = nil
```

Ce bout de code présente donc une façon très simple d'utiliser la table retournée par listDirectory().
Vous voyez aussi comment on accède au nom et à la variable qui permet de savoir si c'est un dossier ou un fichier : elem.name donne le nom, et elem.isDir est un booléen qui vaut true si c'est un dossier (et donc false si c'est un fichier). Bien entendu, vous pouvez accéder à ces variables sans passer par une boucle :

Code : Lua

```
nom = files[2].name
dossier = files[2].isDir
```

TP : utilisation des fichiers INI

Cette partie est un TP-tuto, qui va vous permettre d'appréhender les fonctions sur les fichiers INI, mais aussi d'utiliser les fonctions vues précédemment (enfin, surtout celles sur les dossiers).

But du TP

On va faire quelque chose de très simple : en nous basant sur un fichier INI, on va accueillir l'utilisateur en affichant son nom dans la couleur donnée par le fichier INI. On va aussi lui permettre de modifier son nom depuis le programme.

Les fichiers INI

C'est quoi ?

Déjà, c'est un fichier dont l'extension est ".ini".

C'était pas dur à deviner en même temps

On sait jamais.

Ces fichiers servent à conserver la configuration et la personnalisation de programmes. Ce type de fichier a été introduit par Windows dès les premières versions de l'OS, bien que maintenant ils aient été pour la plupart remplacés par le Registre.

Ils ont tous la même structure :

Code : Autre

```
[catégorie]
; Commentaire
variable=valeur
variable2=autreValeur ; Commentaire
```

Quel que soit le type de valeur (chaîne de caractères ou nombre), il n'y a pas de guillemets ou de notation spéciale (de toute façon, vous verrez que pour les fonctions de µLua, c'est tout du String).

Les utiliser avec µLua

Parce que c'est bien beau tout ça, mais on sait toujours pas comment nous en servir dans nos homebrews !

On y arrive

µLua met à notre disposition deux fonctions : une pour lire le fichier, et l'autre pour enregistrer les données.

Code : Lua

```
config = INI.load(fichier)    -- Chargement du fichier
param = config["catégorie"]["variable"]    -- Accès à la valeur de
"variable" dans la catégorie "catégorie" (j'ai vraiment trop
d'imagination pour les noms moi :p )
config["catégorie"]["variable2"] = "blabla"    -- On change la
valeur de "variable2" dans la catégorie "catégorie"
INI.save(fichier, config)    -- On sauvegarde la config
```

Voilà, INI.load() sert à lire le fichier : elle renvoie une table dans le genre tableau associatif (avec des index qui ne sont pas des nombres), qui contient pour chaque catégorie une table qui correspond à cette catégorie. Les index portent logiquement les noms des catégories et des variables.
INI.save() permet d'enregistrer la table dans le fichier.

La table enregistrée dans le fichier ne peut contenir que des Strings ! Si vous souhaitez enregistrer un nombre, vous devez le convertir en string à l'aide de la fonction *tostring()*. De même, si vous souhaitez utiliser une valeur numérique dans votre script tirée d'un fichier INI, il vaut mieux la convertir en nombre avec *tonumber()*

Et le nôtre ?

Pour être sûr d'avoir la même base, voici le fichier INI que je vais utiliser :

Code : Autre

```
[couleur]
; Par défaut, la couleur sera le blanc (31, 31, 31)

rouge=31
vert=31
bleu=31

[nom]
; Par défaut, la variable est vide

nom=
```

S'il ne vous plaît pas, vous pouvez très bien changer sa structure, mais il faudra alors penser à adapter le code en conséquence

Modifier le nom du bonhomme

Avant µLua 3.0, ça aurait été très simple, puisque µLua intégrait par défaut un clavier. Or, depuis cette version, il n'y est plus. Mais... des membres dévoués de la communauté ont mis au point des claviers particulièrement bien faits.

- Clavier de Papymouge : le premier, l'ancêtre
- Clavier de thermo_nono : amélioration de celui de Papymouge (le clavier est désormais personnalisable)
- Clavier de Quent42340 : un autre, le dernier à avoir vu le jour

J'ai une préférence pour celui de thermo_nono, car j'aime bien les skins et la personnalisation ; bien entendu, cet avis n'engage que moi. Mais comme le but de ce TP n'est pas de vous apprendre à embellir vos homebrews, je vais utiliser celui de Papymouge. Vous pouvez le trouver ici.

Utilisation du clavier

Je ne présenterai ici que le clavier de Papymouge, et encore juste pour que vous sachiez comment vous en servir pour que l'utilisateur puisse entrer son nom Si vous voulez en utiliser un autre, je ne peux que vous conseiller de farfouiller sur le forum de µLua (notez la fonction "Rechercher" en haut).

En premier lieu, il faut charger la lib :

Code : Lua

```
dofile("clavier.lua")
```

Le but n'étant pas de vous apprendre à intégrer une lib dans votre programme, je vous renvoie à cette annexe qui parle un peu plus de dofile().

Il faut donc que vous fournissiez avec votre programme le fichier "clavier.lua", mais aussi les fichiers "libClavier.lua" et "standard.lua" que le clavier utilise. Ils doivent être dans le même dossier, cependant ils peuvent être dans un dossier différent de celui de votre programme ; il suffit d'adapter en conséquence le chemin donné à *dofile()*.

Ensuite, il faut initialiser le clavier :

Code : Lua

```
clavier.activeScreen(clav,true)
```

Tout ce qu'il vous faut savoir, c'est que *clav* désigne votre clavier ; gardez cette variable sous le coude, vous en aurez besoin pour plusieurs fonctions.

Il faut aussi l'afficher à chaque tour de boucle :

Code : Lua

```
clavier.show(clav)
```

Pour récupérer l'entrée de l'utilisateur, c'est la fonction :

Code : Lua

```
touche = clavier.held(clav,Stylus.X,Stylus.Y)
```

Elle renvoie la lettre pressée, ou "ent" si on appuie sur "Entrée", ou "del" si on appuie sur "Retour arrière" (ou "" si on n'appuie sur rien). On doit lui passer en argument les coordonnées du stylet.
Notez qu'il vaut mieux vérifier d'abord que le joueur appuie bien sur l'écran

Enfin, pour détruire le clavier à la fin du script, c'est :

Code : Lua

```
clavier.del(clav)
```

On va dire que l'on enregistrera si l'utilisateur appuie sur "Entrée". Le bouton Start permettra de quitter sans sauvegarder.

Codons !

Bon, je vous ai déjà dit beaucoup de choses, et je sais que vous avez hâte de coder
Je vous conseille juste de réfléchir un peu avant de foncer tête baissée, même si ce script n'est pas bien compliqué. Rappelez-vous ce que doit faire le programme, et ça ira

3, 2, 1, Partez !

Need help

Vous galérez sur le retour arrière du clavier ? Si oui, vous avez de la chance : j'ai là une petite aide. Si non... et bien tant mieux pour vous

Indice 1 :

Secret (cliquez pour afficher)

Pour gérer le retour arrière, on doit utiliser la fonction string.sub() (fonction de base de Lua)... Lisez la doc

Indice 2 :

Secret (cliquez pour afficher)

Toujours pas trouvé ? Allez, je vous aide encore un peu... Il faut faire une galipette avec une des deux bornes données à la fonction string.sub() qui doit être négative. Mais comment ça négative ?

Correction

Ayé, fini !

Alors, facile hein ?

Voici ma correction :

Code : Lua

```lua
dofile("clavier.lua")
clavier.activeScreen(clav,true)

config = INI.load("tp_INI.ini")
couleur = Color.new(tonumber(config["couleur"]["rouge"]),
tonumber(config["couleur"]["vert"]),
tonumber(config["couleur"]["bleu"]))

while not Keys.newPress.Start do
    Controls.read()

    screen.print(SCREEN_UP, 0, 0, "Votre nom est :", couleur)
    screen.print(SCREEN_UP, 0, 8, config["nom"]["nom"], couleur)
    screen.print(SCREEN_UP, 0, 50, "Appuyez sur \"Entrée\" pour
enregistrer", couleur)
    screen.print(SCREEN_UP, 0, 58, "et sur Start pour quitter.",
couleur)

    clavier.show(clav)

    local touche = ""
    if Stylus.newPress then
        touche = clavier.held(clav,Stylus.X,Stylus.Y)
    end
    if(touche ~= "") then
        if touche == "ent" then        -- Si l'utilisateur a appuyé
sur "Entrée"...
   INI.save("tp_INI.ini", config)        -- ... on enregistre.
  elseif touche == "del" then       -- Si l'utilisateur a appuyé sur
"Retour arrière"
   config["nom"]["nom"] = string.sub(config["nom"]["nom"], 1, -2)  -
- On élimine la dernière lettre
  else          -- Si l'utilisateur a appuyé sur une touche
"normale"
   config["nom"]["nom"] = config["nom"]["nom"]..touche
  end
    end

    render()
end

config = nil
clavier.del(clav)
couleur = nil
```

Comme vous l'avez remarqué on a utilisé tonumber() car on n'enregistre que des strings dans un INI et *Color.new()* ne prend que des nombres donc on doit convertir nos strings en nombres grâce à *tonumber()*

Je ne pense pas que ça vous aura été difficile de trouver quelque chose de fonctionnel

Les améliorations !

Je vois pas grand-chose, le script n'est pas très conséquent en même temps...
Vous pourriez permettre à l'utilisateur de choisir la couleur parmi quelques-unes depuis le programme (faites des boutons par exemple ^^), et modifier tout de suite la couleur sans avoir à recharger le fichier INI. N'oubliez pas que INI.save() ne peut enregistrer qu'une table contenant des chaînes de caractères !
Vous pourriez aussi vérifier que les valeurs de rouge, vert et bleu du fichier INI sont bien entre 0 et 31 ; le cas échéant, affichez le texte en blanc et un message d'erreur en rouge

Avec ce que nous avons vu, vous pouvez très bien faire un petit explorateur de fichier pour la DS, avec toutes les fonctions de base (copie, suppression...)

Du temps et des polices (d'écriture)

Vous avez déjà pu découvrir de nombreuses fonctionnalités offertes par MicroLua, mais il reste encore deux ou trois choses intéressantes, que j'ai regroupées ici en un seul chapitre : dans l'ordre, les polices d'écriture (*Fonts*), les *timers* et la gestion du jour et de l'heure (avec *DateTime*). Le rapport n'est pas tout à faire évident, mais au moins ça remplit son chapitre

Les fontes

Je suis sûr que vous vous êtes dit que quand même, si on veut faire un jeu grand public, on ne peut raisonnablement pas garder cette immonde police d'écriture qui fait passer notre DS pour un vieux PC sous MS-DOS.

Et c'est là mes amis que j'agite devant vos yeux ébahis la gamme de fonctions **Font** !

Parce que quand même, MicroLua il est pénible...

Eeeeeet forcément, on ne peut pas charger directement nos jolies fontes en TrueType !
Il faut d'abord les passer à la moulinette du convertisseur qui s'appelle OSLFonts ; comme pour le son, je vous redirigerai vers l'annexe prévue à cet effet.

Vous vous retrouvez donc avec votre police convertie au format adéquat, on peut maintenant l'utiliser.

On charge, on décharge, on charge...

Première chose à faire évidemment : charger la fonte en mémoire, comme suit :

Code : Lua

```
police = Font.load("font.oft")
```

On obtient donc une variable du type *Font*. Comme d'habitude, vous devez remplacer *"font.oft"* par le bon chemin vers le fichier.

Quand vous aurez fini votre boulot, vous pouvez libérer la mémoire :

Code : Lua

```
Font.destroy(police)
police = nil
```

Et on en profite enfin !

Avec Font.print() c'est ça ? Hein j'ai bon, hein j'ai bon ?

Non.

Oui je sais, cette fonction est documentée, mais elle a quelques petits problèmes et est notée obsolète sur le Wiki. On lui préfère son équivalent dans les fonctions *screen* qui est :

Code : Lua

```
screen.printFont(ecran, x, y, texte, couleur, police)
```

Je pense que son utilisation est plus qu'évidente, c'est exactement comme *screen.print()* mis à part bien sûr qu'il faut lui donner une police

Au final ça nous donne quelque chose comme :

Code : Lua

```
police = Font.load("font.oft")

while not Keys.newPress.Start do
    Controls.read()

    screen.print(SCREEN_UP, 0, 0, "Hi world!", Color.new(0, 0, 31),
police)

    render()
end

Font.destroy(police)
police = nil
```

Datetime

Passons maintenant à la date et à l'heure. Ça ira très vite à vrai dire, il n'y a que deux fonctions

L'objet DateTime

D'abord un petit topo sur l'objet du type *DateTime*.
C'est donc une table, qui contient des informations sur le temps. Ses champs sont :

- *year* : années
- *month* : mois
- *day* : jours
- *hour* : heures
- *minute* : minutes
- *second* : secondes

Pour celui qui comprend l'anglais, rien de bien compliqué

Quand vous avez votre objet *DateTime*, vous pouvez :

Code : Lua

```
dateTimeObj.month = 11      -- Changer ses informations
screen.print(SCREEN_UP, 0, 0, dateTimeObj.month)     -- Et les
afficher de la même manière
```

Je suis tout bêtement en train de vous expliquer comment utiliser une table en Lua

Les fonctions

La première crée un objet *DateTime* vide. Disons que ça nous évite de créer la table nous-mêmes

Code : Lua

```
dateTimeObj = DateTime.new()
```

Mais vous utiliserez plus probablement celle-ci :

Code : Lua

```
dateTimeObj = DateTime.getCurrentTime()
```

Elle vous donne un objet *DateTime* qui contient l'heure actuelle (et le jour évidemment).

Bon voilà, c'est tout Maintenant, passons aux *Timers* !

Les timers

Les *timers* sont en quelque sorte des chronomètres. Vous en lancez un, et il va compter le nombre de millisecondes qui s'écoulent. Le principe est aussi simple que ça, et ils trouvent leur utilité dans plein de programmes

Dans un premier temps, il faut en créer un. Vous utiliserez pour cela la fonction

Code : Lua

```
Timer.new()
```

Elle retourne un objet du type *Timer*. Cet objet vous permettra de contrôler le *timer*. Dans la suite du cours, je vais l'appeler *john*. Parce que si je l'appelle *timer*, vous allez vous embrouillez, et si je l'appelle *time*, ça écrase les fonctions correspondantes (qui sont des fonctions de base du Lua). Donc ça sera *john*

Voyons maintenant comment on récupère l'information du temps.
On passe par la méthode

Code : Lua

```
john:time()
```

C'est bien une fonction, avec les parenthèses derrière. Si vous ne les mettez pas, vous obtiendrez une erreur.

Vous obtenez ainsi le nombre de millisecondes qui se sont écoulées depuis le démarrage de *john*.

Evidemment, pour que *john* se rende utile, il faut le démarrer. Vous pouvez ensuite l'arrêter, et/ou le remettre à zéro. Les méthodes respectives sont

Code : Lua

```
john:start()
john:stop()
john:reset()
```

Notez que la remise à zéro arrête automatiquement le *timer*.

TayPay

Je vais vous faire coder un script qui regroupe trois choses finalement assez utiles liées au temps : une horloge, un chronomètre, et un compte à rebours !
Ce TP est séparé en plusieurs parties, et chaque partie donne une explication sur le code à réaliser suivi de ma correction. Evidemment, essayez de trouver le script vous-mêmes à partir de l'explication avant de regarder ce que j'ai fait. Encore une fois, ma correction n'a qu'une valeur indicative ; je tente de faire un code clair pour que vous puissiez le comprendre aisément, par conséquent il sera rarement le "meilleur" d'un point de vue optimisation. Si ça marche avec votre version, c'est très bien !

La base

Nous aurons besoin d'au moins deux *timers* : l'un pour le chronomètre, et l'autre pour le compte à rebours. En utilisant deux *timers* différents, on peut faire tourner le chrono pendant qu'on utilise le compte à rebours
On va aussi tout de suite déclarer une variable *mode* qui correspondra au mode choisi : horloge, chronomètre, ou compte à

rebours. On n'affichera qu'un seul mode à la fois, et on pourra changer de mode avec les boutons L et R.
Il nous faut aussi un objet *DateTime* pour l'horloge, ainsi qu'une autre variable (une table) qui contiendra le réglage du compte à rebours, et enfin une dernière variable pour la position du "curseur" du compte à rebours (vous verrez plus loin).

On a donc pour base :

Code : Lua

```
horloge = DateTime.new()

mode = 1                              -- 1 : horloge | 2 : chronomètre
| 3 : compte à rebours
reglageRebours = {0, 0, timestamp = 0}        -- Premier index :
minutes | Second index : secondes | Troisième index : timestamp
(plus facile pour les calculs ;) )
curseur = 1                           -- 1 : minutes | 2 : secondes

chrono = Timer.new()
rebours = Timer.new()

while not Keys.newPress.Start do
    Controls.read()

    -- Votre code ici

    render()
end

horloge = nil
reglageRebours = nil
mode = nil
curseur = nil
chrono:stop()
chrono = nil
rebours:stop()
rebours = nil
```

Les trois modes différents

On va d'abord mettre en place la structure pour pouvoir afficher les modes et en changer.
On affiche un mode à la fois, et on change de mode avec les touches L et R dans cet ordre : horloge ; appui sur R ; chronomètre ; appui sur R ; compte à rebours ; appui sur R ; horloge. Et évidemment, on a la même chose avec L mais dans l'autre sens.

Changement de mode :

Secret (cliquez pour afficher)

Code : Lua

```
if Keys.newPress.R then
    mode = mode + 1
    if mode > 3 then mode = 1 end
end
if Keys.newPress.L then
    mode = mode - 1
    if mode < 1 then mode = 3 end
end

screen.print(SCREEN_UP, 0, 184, "Appuyez sur L et R pour changer
de mode.")
```

Structure conditionnelle pour afficher chaque mode séparément (à partir de la variable *mode*) :

Secret (cliquez pour afficher)

Code : Lua

```
if mode == 1 then
    screen.print(SCREEN_UP, 0, 0, "HORLOGE")
    -- Code pour l'affichage de l'horloge
elseif mode == 2 then
    screen.print(SCREEN_UP, 0, 0, "CHRONOMETRE")
    -- Code pour l'affichage du chronomètre
elseif mode == 3 then
    screen.print(SCREEN_UP, 0, 0, "COMPTE A REBOURS")
    -- Code pour l'affichage du compte à rebours
end
```

Horloge

Très certainement le mode le plus simple.
A chaque tour de boucle, vous mettez à jour les infos sur l'heure actuelle (objet *horloge*) et vous les affichez au format que vous voulez.

Moi j'ai fait comme suit :

Secret (cliquez pour afficher)

Code : Lua

```
if mode == 1 then
```

```
        screen.print(SCREEN_UP, 0, 0, "HORLOGE")

        horloge = DateTime.getCurrentTime()

        screen.print(SCREEN_UP, 0, 32, "Il est actuellement
"..horloge.hour.."h"..horloge.minute.." et "..horloge.second.."
secondes.")
        screen.print(SCREEN_UP, 0, 48, "Nous sommes aujourd'hui le
"..horloge.day.."/"..horloge.month.."/"..horloge.year..".")
```

Chronomètre

On va se mettre d'accord sur les contrôles. Pour lancer le chronomètre, ce sera le bouton A. Pour le mettre en pause, ce sera le bouton B. Et pour le remettre à zéro (et donc l'arrêter), on prendra le bouton X.
Vous affichez ensuite sa valeur, en la formatant un peu pour que ça ressemble à quelque chose Pour ce formatage, vous pourriez avoir besoin de la fonction Lua *math.floor()*, qui renvoie la partie entière d'un nombre (en fait, ça renvoie le nombre entier immédiatement inférieur ou égal au nombre donné en paramètre). Pour ceux qui ne le savent pas, il y a 3600 secondes dans une heure (60 minutes fois 60 secondes) et une seconde équivaut à mille millisecondes.

Secret (cliquez pour afficher)

Code : Lua

```
elseif mode == 2 then
    screen.print(SCREEN_UP, 0, 0, "CHRONOMETRE")

    if Keys.newPress.A then
        chrono:start()
    end
    if Keys.newPress.B then
        chrono:stop()
    end
    if Keys.newPress.X then
        chrono:reset()
    end

    local temps = DateTime.new()            -- Variable locale pour le
formatage
    temps.hour = math.floor(chrono:time() / 3600000)
    temps.minute = math.floor(chrono:time() / 60000)
    temps.second = math.floor(chrono:time() / 1000)
    -- Et on calcule les millisecondes restantes
    local millisecondes = chrono:time() - temps.hour * 3600000 - temps.minute
* 60000 - temps.second * 1000

    screen.print(SCREEN_UP, 0, 32, "Temps :")
    screen.print(SCREEN_UP, 0, 40,
temps.hour.."h"..temps.minute.."min"..temps.second.."s"..millisecondes.."ms")

    screen.print(SCREEN_UP, 0, 168, "A : démarrage | B : pause | X : RàZ")
```

Un peu plus long mais après tout c'est déjà plus utile que l'horloge

Compte à rebours

Le principe du compte à rebours, c'est de décompter le temps à partir d'un temps donné jusqu'à 0.
Pour régler ce temps, on va utiliser les flèches. On affiche sur une ligne les minutes et les secondes de la table *reglageRebours* ainsi qu'un "curseur" (affichage des minutes ou des secondes d'une autre couleur par exemple). Gauche et droite serviront à naviguer entre les minutes et les secondes, et les flèches haut et bas à changer la valeur.
Les autres boutons ont les mêmes actions que pour le chronomètre. Cela dit, lorsqu'on remet à zéro, on repasse en mode réglage.

Pour savoir si l'on est en mode réglage, plutôt que d'utiliser une nouvelle variable, on peut s'appuyer sur le *timer rebours*. Cherchez un peu, sinon voilà comment je vois les choses :

Secret (cliquez pour afficher)

Si le *timer* est à 0, alors on est en réglage. Tout simplement !

Pour calculer le temps restant, je vous recommande l'utilisation de ce que j'appelle de façon impropre un *timestamp*. C'est en fait le nombre de millisecondes correspondant au réglage. Pour le calculer, vous ferez :

Secret (cliquez pour afficher)

Code : Lua

```
reglageRebours.timestamp = reglageRebours[1] * 60000 +
reglageRebours[2] * 1000
```

Tout simplement !

Ça facilite grandement les calculs, croyez-moi

Première aide :

Secret (cliquez pour afficher)

Commencez par le "mode réglage". On a vu que si le *timer* est à 0, on est en réglage. Que doit-on y faire ? Ce ne sont que de

simples conditions sur les boutons concernés N'oubliez pas qu'il n'y a qu'au maximum 59 minutes et 59 secondes
Pensez aussi à mettre à jour le *"timestamp"*.

Seconde aide :

Secret (cliquez pour afficher)

Vous allez devoir vous creuser très légèrement la tête pour trouver le temps restant, mais grâce au *"timestamp"* c'est assez facile, puisque ce dernier et le temps de <italique>rebours<italique> sont en millisecondes Pensez bien aussi à remettre à zéro le compte à rebours une fois terminé.

Ma solution :

Secret (cliquez pour afficher)

Code : Lua

```
    elseif mode == 3 then
        screen.print(SCREEN_UP, 0, 0, "COMPTE A REBOURS")

        local couleurs = {Color.new(31, 31, 31), Color.new(31, 31,
    31)}          -- Table pour les couleurs des minutes et des
    secondes

        if rebours:time() == 0 then              -- Mode réglage
            screen.print(SCREEN_UP, 0, 32, "Réglage :")

            couleurs[curseur] = Color.new(31, 0, 0)          -- On met
    en rouge la partie sélectionnée

            if Keys.newPress.Up then         -- On augmente la partie
    sélectionnée
                reglageRebours[curseur] = reglageRebours[curseur] + 1
                if reglageRebours[curseur] > 59 then
    reglageRebours[curseur] = 0 end
            end
            if Keys.newPress.Down then          -- On diminue la
    partie sélectionnée
                reglageRebours[curseur] = reglageRebours[curseur] - 1
                if reglageRebours[curseur] < 0 then
    reglageRebours[curseur] = 59 end
            end

            if Keys.newPress.Left then
                curseur = 1
            end
            if Keys.newPress.Right then
                curseur = 2
            end

            reglageRebours.timestamp = reglageRebours[1] * 60000 +
    reglageRebours[2] * 1000
        end

        if Keys.newPress.A then
            rebours:start()
        end
        if Keys.newPress.B then
            rebours:stop()
        end
        if Keys.newPress.X then
            rebours:reset()
        end

        -- Si le temps écoulé est supérieur ou égal au temps de
    réglage, alors le compte à rebours est terminé
        if reglageRebours.timestamp <= rebours:time() then
            rebours:reset()
        end

        local minutes = math.floor((reglageRebours.timestamp -
    rebours:time()) / 60000)
        local secondes = math.floor((reglageRebours.timestamp -
    rebours:time()) / 1000) - 60 * minutes

        screen.print(SCREEN_UP, 0, 40, minutes.."min", couleurs[1])
        screen.print(SCREEN_UP, 35, 40, secondes.."s", couleurs[2])
    end
```

Bilan

Voici donc notre script terminé :

Code : Lua

```
    horloge = DateTime.new()

    mode = 1                              -- 1 : horloge | 2 : chronomètre | 3 :
    compte à rebours
    reglageRebours = {0, 0, timestamp = 0}        -- Premier index : minutes |
    Second index : secondes | Troisième index : timestamp (plus facile pour les
    calculs ;) )
    curseur = 1                           -- 1 : minutes | 2 : secondes

    chrono = Timer.new()
    rebours = Timer.new()
```

```
while not Keys.newPress.Start do
    Controls.read()

    if Keys.newPress.R then
        mode = mode + 1
        if mode > 3 then mode = 1 end
    end
    if Keys.newPress.L then
        mode = mode - 1
        if mode < 1 then mode = 3 end
    end

    screen.print(SCREEN_UP, 0, 184, "Appuyez sur L et R pour changer de
mode.")

    if mode == 1 then
        screen.print(SCREEN_UP, 0, 0, "HORLOGE")

        horloge = DateTime.getCurrentTime()

        screen.print(SCREEN_UP, 0, 32, "Il est actuellement
"..horloge.hour.."h"..horloge.minute.." et "..horloge.second.." secondes.")
        screen.print(SCREEN_UP, 0, 48, "Nous sommes aujourd'hui le
"..horloge.day.."/"..horloge.month.."/"..horloge.year..".")
    elseif mode == 2 then
        screen.print(SCREEN_UP, 0, 0, "CHRONOMETRE")

        if Keys.newPress.A then
            chrono:start()
        end
        if Keys.newPress.B then
            chrono:stop()
        end
        if Keys.newPress.X then
            chrono:reset()
        end

        local temps = DateTime.new()            -- Variable locale pour le
formatage
        temps.hour = math.floor(chrono:time() / 3600000)
        temps.minute = math.floor(chrono:time() / 60000)
        temps.second = math.floor(chrono:time() / 1000)
        -- Et on calcule les millisecondes restantes
        local millisecondes = chrono:time() - temps.hour * 3600000 -
temps.minute * 60000 - temps.second * 1000

        screen.print(SCREEN_UP, 0, 32, "Temps :")
        screen.print(SCREEN_UP, 0, 40,
temps.hour.."h"..temps.minute.."min"..temps.second.."s"..millisecondes.."ms")

        screen.print(SCREEN_UP, 0, 168, "A : démarrage | B : pause | X :
RàZ")
    elseif mode == 3 then
        screen.print(SCREEN_UP, 0, 0, "COMPTE A REBOURS")

        local couleurs = {Color.new(31, 31, 31), Color.new(31, 31, 31)}
-- Table pour les couleurs des minutes et des secondes

        if rebours:time() == 0 then             -- Mode réglage
            screen.print(SCREEN_UP, 0, 32, "Réglage :")

            couleurs[curseur] = Color.new(31, 0, 0)         -- On met en
rouge la partie sélectionnée

            if Keys.newPress.Up then        -- On augmente la partie
sélectionnée
                reglageRebours[curseur] = reglageRebours[curseur] + 1
                if reglageRebours[curseur] > 59 then reglageRebours[curseur]
= 0 end
            end
            if Keys.newPress.Down then        -- On diminue la partie
sélectionnée
                reglageRebours[curseur] = reglageRebours[curseur] - 1
                if reglageRebours[curseur] < 0 then reglageRebours[curseur] =
59 end
            end

            if Keys.newPress.Left then
                curseur = 1
            end
            if Keys.newPress.Right then
                curseur = 2
            end

            reglageRebours.timestamp = reglageRebours[1] * 60000 +
reglageRebours[2] * 1000      -- On recalcule le timestamp
        end

        if Keys.newPress.A then
            rebours:start()
        end
        if Keys.newPress.B then
            rebours:stop()
        end
        if Keys.newPress.X then
            rebours:reset()
        end

        -- Si le temps écoulé est supérieur ou égal au temps de réglage,
alors le compte à rebours est terminé
        if reglageRebours.timestamp < rebours:time() then
            rebours:reset()
```

```
            end

            -- Formatage
            local minutes = math.floor((reglageRebours.timestamp -
    rebours:time()) / 60000)
            local secondes = math.floor((reglageRebours.timestamp -
    rebours:time()) / 1000) - 60 * minutes

            screen.print(SCREEN_UP, 0, 40, minutes.."min", couleurs[1])
            screen.print(SCREEN_UP, 35, 40, secondes.."s", couleurs[2])
        end

        render()
    end

    horloge = nil
    reglageRebours = nil
    mode = nil
    curseur = nil
    chrono:stop()
    chrono = nil
    rebours:stop()
    rebours = nil
```

Vous avez ainsi pu avoir un bon aperçu de tout ce que l'on peut faire principalement avec des *timers* Et bravo si vous avez tout trouvé tout seul !
Et bien voilà, que nous reste-t-il donc à voir désormais... Et bien plus grand-chose, vous en êtes presque au bout !

Vous connaissez désormais tout de MicroLua.

A vous maintenant de créer les magnifiques projets qui empliront les sujets du forum de MicroLua !

Partie 4 : Annexes

Dans ces annexes vous trouverez des liens ou informations utiles.
Et même sûrement des astuces .

Foire aux astuces

Voilà un peu le chapitre fourre-tout, avec (pour l'instant ?) des liens et deux-trois programmes intéressants pour la programmation avec µLua.

Des liens

Sites en rapport avec µLua

Le site officiel de µLua : ce site en anglais n'était malheureusement plus à jour. Il contenait la version 1, 1.0.1 et 2 de µLua ainsi que les documentations de µLua 1.0.1 et µLua 2.
On y trouvait aussi un tutoriel pour µLua 1.0.1.

Le forum officiel de µLua : ce forum très actif pourra être votre principale aide. La soixante-quinzaine de membres donc une dizaine d'actifs pourront vous aider avec un script défaillant ou autre.
Il y a une section française (très active) et une anglaise (moyennement active).

Le Google Code de µLua : ce site possède un Wiki et permet de télécharger la dernière version de MicroLua. Il est principalement en anglais, bien que certaines pages du Wiki soient traduites en français.

MicroLua DS Underground : ce site fait par un des membres du forum, thermo_nono, contient beaucoup de libs et de homebrews.

MicroLua : c'est le fansite de Aurel2108, fais pour être simple et rapide, il regroupe un uploader de fichiers et une section téléchargements contenant les versions de Micro/Nano Lua, les docs, les libs et les homebrews...
Il n'est malheureusement plus tenu à jour par son créateur qui a quitté la communauté MicroLua.

MicroLua4DS : c'est le fansite de Reylak, mettant à votre disposition comme µLua DS Underground des libs, des homebrews, etc.

Nano Lua : c'est le site de Quent42340. Il remplit le même rôle que les précédents fansites, avec un forum et deux ou trois sections hors sujet

La plupart de ces fansites sont hélas morts, vous pourrez cependant peut-être trouver quelques éléments qui vous seront utiles.

De la documentation

MicroLua

La doc de µLua 4 : elle est disponible dans le dossier *Documentation* du package d'installation. C'est une version hors-ligne de la doc du Google Code ci-dessous.

La doc de µLua 4 en ligne sur le Google Code : (<lien url="/http://code.google.com/p/microlua/wiki/ApiFourDotZeroDotOne?wl=fr">en français</lien> | en anglais)

Le tutoriel sur les canevas de MicroLua 2.0 : (en anglais)

Le tutoriel sur le son de MicroLua 3.0 : (en anglais)

La section des tutoriels sur le forum MicroLua : (en français)
Cette section contient quelques tutoriels utiles pour mieux appréhender certains aspects de MicroLua (voire de Lua).

Lua

La doc de Lua 5.1 : (en anglais)

Des tutos pour utiliser Lua 5.1 : (en anglais)

Des utilitaires

Pour le codage

Outre les "classiques" Notepad++ et autres Vims pour quasiment tous les langages, vous pourriez préférer un IDE plus spécifique à µLua, voire même vouloir coder directement sur votre DS !

µLua Creator

Cet IDE développé par Quent42340 offre quelques outils sympathiques permettant de développer plus vite, et en mettant moins les mains dans le code.
Vous pouvez télécharger la version 4.0 beta 3 ici (sur le Google Code dédié).

Il propose donc un générateur de code, un éditeur de texte avec coloration syntaxique, mais aussi un générateur de code pour le dessin, un éditeur d'évènements, et enfin un éditeur de maps. Il s'intègre de plus avec les testeurs MicroLua Simulator et No$GBA Tester.

Cependant, et sans vouloir dévaloriser le très bon travail de Quent, je pense que l'utilisation abusive de ces assistants n'est pas une bonne chose pour vous qui lisez ce tuto. Ils ne permettent pas d'apprendre à coder comme vous apprendriez en codant "à la main". Bien sûr, quand vous aurez de l'expérience, rien ne vous empêchera de vous servir de µLua Creator

Quoi qu'il en soit, ce logiciel n'est plus à jour.

Lua Editor DS

Ça, c'est le fin du fin, ~~c'est Plantafin~~ ! Lua Editor DS (raccourci en LED) vous permet tout bonnement... de coder sur votre DS ! Et on dit... merci samy !
La version 1.3, compatible avec MicroLua 3.0, est disponible ici.

LED est un véritable IDE pour votre console ; il propose des fonctions de base comme le copier-coller, la recherche/remplacement, l'affichage des numéros de ligne, l'indentation intelligente, mais aussi la coloration syntaxique et même l'auto-complétion ! Notez que cette dernière est basée sur le fichier Autocomp ; vous pouvez donc rajouter vos propres mots-clés.
Il permet aussi bien sûr, en plus d'éditer un script, de l'exécuter, puis de revenir l'éditer. C'est donc un *must-to-have*, indispensable pour coder en nomade.

Notez que vous pouvez personnaliser quelques aspects du logiciel dans le menu des options. Il est conseillé pour coder efficacement, de désactiver la coloration qui malheureusement fait ramer la console, et d'utiliser l'auto-complétion.

Pour tester sur l'ordinateur

Vous vous êtes certainement dit "Aah, si seulement je n'avais pas à mettre mon scripts sur mon linker pour le tester !". Et bien votre vœu est exaucé !

No$GBA Tester

La première façon est d'utiliser l'émulateur Nintendo DS No$GBA (ne vous laissez pas tromper par la mention "GBA", ça marche aussi pour la DS).
C'est un émulateur, ça signifie qu'il peut lancer sur l'ordinateur des roms (des programmes pour console), en l'occurrence pour la DS. Couplé à un petit système permettant de compiler plus ou moins à la volée une version de µLua pour l'ordinateur, il permet de tester vos scripts sans votre console !

Préparation

No$GBA est fourni dans un dossier *Utilities*, que vous pouvez trouver sur le Google Code ici. En fait, l'exécutable en lui-même (l'émulateur quoi) ne se trouve pas dans l'archive, mais vous pouvez le télécharger ici. Extrayez-le, et mettez-le dans le dossier *no$gba Tester*.
C'est prêt... ou pas.

Si vous êtes sous Vista (et certainement sous Seven aussi), la commande *sleep 2* a de fortes chances de ne pas fonctionner, pour la simple et bonne raison qu'elle n'y est plus. Or, elle est indispensable dans le fichier *create.bat*. Vous devez donc la remplacer par :

Code : Console

```
choice /c 1 /d 1 /t 2 > nul
```

Cette instruction a pour but de faire une pause de deux secondes, le temps que certaines choses se compilent.

Vous pouvez aussi trifouiller dans les options de *Options->Controls Setup*, afin de personnaliser les contrôles.

Testons !

Placez tous vos fichiers à tester dans le dossier *fat*. Maintenant, lancez *create.bat*. Une console toute moche en noir et blanc s'affiche (on est à l'époque de la couleur enfin !), avec plein de mots bizarres (on appelle ça de l'anglais), jusqu'à une phrase en bon français vous invitant à "Appuyer sur une touche pour continuer...". Obéissez. Puis lancez No$GBA. Dans la fenêtre d'explorateur qui s'est ouverte, sélectionnez la rom *luads_fs.nds*. Et vous voilà dans le shell de MicroLua ! Vous n'avez plus qu'à lancer votre script.

Ça marcheuh pas !

Il se peut que vous ayez cette erreur :

Code : Console

```
Couldn't open libs.lua
```

Ça veut dire que le fichier *libs.lua* est introuvable. Il doit en temps normal se trouve dans le dossier </italique>fat *(oui, au même niveau que vos scripts)*.

Parfois, on a cette erreur alors que <italique>libs.lua est bel et bien où il faut. Essayez alors de vider le dossier *fat*, et de ne garder que l'essentiel (les libs de µLua, votre script à tester et tout ce dont il a besoin pour tourner). C'est dû au fait que le "compilateur" qui crée la rom donne une taille prédéfinie, qui en général suffit... en général.

MicroLua Simulator

No$GBA, c'est bien ; mais pour le dév, MLS, c'est mieux.
MLS, c'est un programme développé en Lua par Ced-le-pingouin, qui propose une alternative intéressante à No$GBA.
Tout d'abord, pour le télécharger c'est ici. Vous trouverez sur cette page une version pour Linux, une pour Windows, et une pour Mac. Il y a aussi les sources pour les curieux. A vous de choisir la bonne version en fonction de votre OS. La dernière version au moment de l'écriture de ces lignes est la 0.5 beta 1.

Après extraction en bonne et due forme, vous vous retrouvez avec une floppée de fichiers, et au milieu un petit exécutable *mls.exe*. Lancez-le, puis cliquez-sur *File->Open*... (*Fichier->Ouvrir*... en français), et sélectionnez votre script.
Comme vous pouvez le voir, c'est moins fastidieux qu'avec No$GBA.

Je ne vais pas trop détailler son fonctionnement, le *Readme* en français et en anglais le fera mieux que moi. Vous pouvez trouver la liste des contrôles dans *Help->Show key bindings* (*Aide->Montrer les associations de touches* en français).

Alors, lequel des deux choisir ?

Chacun a ses avantages et ses inconvénients :

No$GBA Tester :

- de par la façon dont il teste, il est plus proche de la réalité que MLS, malgré tous les efforts de Ced
- pas très importants mais présents : lorsque que vous appuyez sur la touche Imprim. Ecran de votre clavier, No$GBA envoie dans le presse-papiers une image qui contient uniquement les deux écrans accolés, alors que MLS ne le propose pas
- procédure de test assez longue
- parfois pénible à faire fonctionner

MicroLua Simulator :

- outils pour le développeur (mise en pause et redémarrage du script, console d'erreur...)
- procédure de test rapide et facile
- le son ne fonctionne pas
- n'intègre pas les limitations de RAM/VRAM de la console
- simule à la fois µLua 2.0 et 3.0, sans prévention en cas d'utilisation de fonction spécifique à une version

En fin de compte, les deux se valent à peu près. C'est un peu une affaire de goût, mais de nombreuses personnes ont tendance à privilégier MLS, certainement pour la facilité de test.
Pour ma part, MLS me sert à tester "à la volée", tandis que je dépoussière No$GBA pour avoir un aperçu à coup sûr réaliste des dernières modifications. Cependant, ça ne me dispense pas de tester *in situ* (c'est-à-dire sur la DS).
Maintenant, vous savez où chercher en cas de besoin, et vous avez tout ce qu'il faut pour nous faire de jolis homebrews

Les librairies

En visitant le forum ou les sites dédiés à MicroLua, vous êtes certainement tombé au moins une fois sur une librairie.
Cette annexe va vous apprendre à vous en servir.

A titre d'exemple, nous allons nous servir de la libNumericUpDown de Reylak (qui permet d'inclure des contrôles GUI servant à entrer une valeur numérique). Elle est disponible ici.

Remarque relativement importante : on devrait en fait parler de "bibliothèque", et non pas de "librairie". Cela vient du fait que ces scripts sont appelés en anglais "libraries", qui est un faux ami, et signifie en réalité "bibliothèque". Malheureusement, par abus de langage, c'est "librairie" qui s'est imposé.

L'instruction dofile()

Afin de tester la procédure, il est conseillé de télécharger la libNumericUpDown qui va nous servir d'exemple.

La façon "classique" utilisée par la communauté MicroLua pour utiliser une librairie, et d'utiliser la fonction dofile() (qui, au passage, est une fonction de base de Lua, et non pas de µLua). La documentation de Lua nous dit que dofile() ouvre le fichier passé en argument, et l'exécute.

On s'en sert donc de cette manière :

Code : Lua

```
dofile("libNumericUpDown.lua")
```

Cette instruction est à placer au début du programme, pour que les fonctions, variables etc. crées par la libs soient accessibles dans tout le reste du programme.

Ensuite, et bien c'est très simple, vous n'avez qu'à appeler les fonctions définies par la libs comme des fonctions normales :

Code : Lua

```
objet = NumericUpDown.new(arguments)     -- Instanciation d'un nouvel objet de la classe NumericUpDown (appel du constructeur de classe)
objet:draw(Color.new(31, 31, 31))        -- Appel de la méthode draw() de cet objet
screen.print(SCREEN_UP, 0, 0, NumericUpDown.objectCount)     -- Accès à la variable objectCount
```

Toutes ces instructions doivent être placées après le dofile(), sans quoi ça ne marchera pas.

Tous les termes techniques d'instanciation, constructeur de classe, etc. sont liés à la conception Orientée Objet de cette lib (qui est entre nous soit dit certainement la meilleure façon de faire une lib). Je ne peux décemment pas vous faire un cours là-dessus, c'est pourquoi je vous renvoie au cours de M@teo21 sur le C++, et plus précisément sur la partie expliquant ce qu'est la POO.

L'instruction require()

Utiliser dofile() est en fait une mauvaise habitude. En effet, Lua dispose d'un système complet de modules, qui gère très bien les libs automatiquement.
N'utilisant que dofile(), je ne maîtrise pas les modules en Lua, c'est pourquoi je ne m'avancerais pas à faire un tuto là-dessus. Je vous invite donc à regarder la section sur les modules de la doc Lua (en anglais malheureusement), qui traite de tout ce qu'il faut savoir là-dessus.

Sachez donc que la meilleure méthode est celle de require(), bien que tout le monde utilise dofile()...
Et bien voilà, vous savez maintenant comment utiliser ce magnifique clavier ou ces superbes contrôles de GUI dans vos programmes !

Vous avez désormais terminé votre apprentissage de MicroLua, mais il reste encore plein de choses à connaître !

A force de coder des scipts, vous allez vous améliorer et petit à petit vos projets deviendront vraiment conséquents

Je vous souhaite donc bonne route et à très bientôt sur le forum de MicroLua !