Le LUA sur g100 et g85: programmer

AVANT PROPOS:

Pour la g100:
Il est interressant de comparer le lua sur g100 aux autres langages disponibles:

-Le Basic: Le lua est extrement plus rapide que le basic pour les conditions, les calculs, et toute la gestion graphique, cependant les calculs peuvent être un peu plus précis en Basic. Le Basic se programme on-calc. Le lua peut avoir 3 nuances de gris et des animations très fluides.

-Le MLC: Le lua est plus agréable à programmer que le MLC. Le lua a une bien meilleur gestion des conditions (le mlc se restreind à des if et des 'fgob'(revient au début de la fonction). Le lua gère les while, for, if ,elseif,else,until,...
Au niveau de la rapidité, le lua est en moyenne 2 fois plus860 rapide. Le lua gère les nombres à virgules, les parenthèses, et a de nombreuses fonctions que le MLC n'a pas. Toute fonction gérée par le MLC est gérée par le lua. En lua on peut aussi faire des librairies à la place de programme. Au MLC sur g100, il faut se limiter à 20000 octets avant d'avoir un bug et des 'mémoires vidées'. Avec le lua, il faut se limiter à 44000 octets avant d'avoir un message d'erreur et un retour au menu sans problèmes.
-Le C: Le C est beaucoup plus long à apprendre que le lua. Il est plus difficile à gèrer, et il faut petit à petit creer 'une bibliothèque' de fonctions, avant de faire des choses interressantes. Cependant le C est plus rapide que le lua. Un programme lua bien écrit peut égaler un programme C.

On précompile les .lua sur pc en .lc et on les transferts sur g100 pour les executer.

Pour la g85:
Sur g85 (et autres calculatrices de la même famille), le basic et le C sont bien plus performants que sur g100, mais le lua reste un langage attractif pour ses possibilitées, sa simplicité et sa rapidité.

La taille maximale d'un .lc est plus faible que sur g100 pour la g85. Par contre on peut executer des .lua on-calc (mais la taille maximale est plus faible et il peut y avoir des problèmes de compilations: préférez compiler sur pc).

La vitesse du lua sur g85 a été harmonisée avec celle du lua pour g100 afin que les programmes soient compatibles. Mais l'harmonisation est partielle et un programme lua sur g85 peut être jusqu'à 5 fois plus rapide que sur g100.

Pour tous:

Ce document vise à vous donner les bases de la programmation du lua sur g100.
Ceci n'est pas une bible. La partie langage lua en lui même peut être complété par les nombreuses documentations anglaises. (par exemple on peut aussi déclarer une variable 2::=x au lieu de x=2)

Le lua est un des plus rapide langage interpreté. Il fonctionne en deux étapes: d'abord le programme .lua est lu et est précompilé en .lc, puis il est executé via le programme précompilé.

I/ Le lua (le langage en lui même)

Commençont par les variables:

Il y a différents types de variable.
Une variable peut contenir des majuscules, des minuscules ou des chiffres (à condition que le nom ne commence pas par un chiffre)
exemples: B R55 nom Nom Rnom2 sont des variables différentes.

Les différents type sont nil, nombre, table, string, userdata et fonction

- nil: la variable est vide. Si on ne lui donne aucune valeur, elle reste à nil. Une variable ayant pour valeur nil est supprimée de la mémoire par le 'garbagecollector' du lua.
- Nombre: la variable contient un nombre de définition 'float' (c'est à dire que le nombre correspond aux nombres float en C) : le nombre peut aller de $-3.4*e^{-38}$ à $3.4*e^{38}$
  Exemple: nb= 4.7e9 ou nb = 534
- table: la variable est une table (= équivalent liste ou matrice)
  exemple: C= {} --on initialise la table
  C[40]=5 –on met 5 dans la 40e case.
  Une table peut contenir des nombre, des strings, voir des fonctions.
- String: la variable contient un mot ou une phrase. (='chaine de caractères')
  Un string prend très peu de place.
  un sprite est contenu dans un string pour prendre moins de place.
- Booléen: la variable contient true ou false.
  A= (1==2) --A contient false
  A= ((y+x)<w) --A contient true ou false
  A noter que nil et false ont même table de vérité, (if nil then ... pareil que if false then ..) alors que les autres types et true ont même table de vérité (if nombre then .. pareil que if true)
- Userdata: Réservé aux fichiers (voir librairie file)
- Fonction: la variable réfère à une fonction de la librairie ou une fonction lua
  exemple: (ce qui suit '--' est un commentaire et ne fait pas partie du code)

```
line=nbdraw.line -- la fonction ligne réfère à la fonction nbdraw.line
function rectangle(x1,y1,x2,y2)  --rectangle est une fonction lua de 4 paramètres
line(x1,y1,x1,y2)  --dessine le rectangle
line(x2,y1,x2,y2)
line(x1,y1,x2,y1)
line(x1,y2,x2,y2)
end -- on signale la fin de la fonction

rectangle(10,10,20,20) --on l'appelle dans la boucle principale
```

remarque: placer local devant un variable permet d'accélerer son appel et donc la vitesse du programme. Si la variable locale est déclarée dans une fonction, elle lui est spécifique. Une variable non local est une variable globale. Exemple:

```
local resultat  -- la variable est équivalente à une globale mais en plus rapide
y=2   --  y est une globale
```

```
function calcul(entree) --entree est automatiquement locale à la fonction
local x --x est locale à la fonction
x=(2/entree)
return y+x
end

resultat=calcul(y)
x=resultat+2  -- x est different de x dans la fonction calcul. Ici x est globale.
```

Pour apprendre plus en profondeur le lua, je vous conseil se site:
http://lua.gts-stolberg.de/fr/index.php

Les différentes opérations possibles sur les nombres sont:
a = a + b
a= a – b
a= -a
a= a%b (modulo; c'est à dire que a contient le reste de la divistion euclidienne de a par b)
a = a * b (multiplication)
a = a / b
a = a^b (puissance. Ne marche qu'avec b entier. Si vous utilisez un b à virgule, utilisez pow(a,b) de math.lua)
On peut utiliser les parenthèses.

Pour savoir le nombres de caractères dans une string ou dans une table ainsi:

```
phrase = "coucou," -- le nom de la variable import peu
suite = "je m'appelle veb"
longueur = #phrase  -- contient le nombre de lettres. Ici 7.
phase =  phrase .. suite -- on ajoute suite à phrase
-- phrase contient "coucou,je m'appelle veb"
longueur = #phrase -- contient la nouvelle longeur de la string
```

Si on veut que cela contienne "coucou, je m'appelle veb" (on rajoute un espace après ,),
on fait:

```
phrase = phrase .. " " .. suite
```

on peut aussi connaitre la longeur d'une table:

```
table = {} --on initialise la table
table [1] = {1,2,3} -- on crée une 2 e dimension (matrice) dans cette case
-- #table contient 1 mais #table[1] contient 3
```

Le paragraphe suivant n'est pas obligatoire, mais il peut permettre de mieux comprendre quelques imprécisions, ou légères erreurs.

--- les nombres plus en détail

Les float:
Le Lua utilise le système des nombres flottants de précision simple tenant sur 4 octets (float).
La précision est assez élévé et les calculs rapides, mais on est moins précis que pour le système

utilisé par casio par exemple.
Tout les entiers de 0 à 16777215 (et leurs négatifes associés) sont précis à 100%
Au delà les nombres seront arrondis à l'entier pair, puis tout les 4, ...
Pour les nombres non entiers, la précision est en fait une sorte d'arrondi:

Par exemple le codage précis de 3,3 est impossible
si vous rentrez x = 3.3, en réalité x contient **3.299999952316284179 6875**
mais x == 3.3 renvoira vrai.
Lors de l'affichage de x, nbdraw.print affichera 3,3 car la précision d'affichage est de 6 par défaut (c'est à dire que l'arrondi se fait pour que le nombre tienne dans 6 chiffres et l'on enlève les 0)
Si vous augmentez la précision d'affichage vous pourrez voir l'imprécision.

Si vous voulez par exemple mettre 2*pi dans une variable, pour ne pas cumuler les erreurs il vaut don mieux rentrer manuellement 2pi plutot que de la calculer.

II/ Les librairies lua sur g100/85 (les fonctions intégrées)

a) les fonctions générales

1)les fonctions non-graphiques existant dans l'edition standart du lua

Il se peut que vous n'utilisiez jamais ces fonctions, Néammoins sachez qu'elles existent.

--base.collectgarbage(option [,argument])
Option peut prendre un de ces mots:
-**"stop":** arrète la collecte des valeurs inutilisées.
-**"restart":** recommence la collecte.
-"collect": Fait une collecte complète (à utiliser pour liberer de la mémoire)
Cette fonction libère de la place pour l'execution du programme. Attention, la place libérée n'est que pour de nouvelles valeurs lua (exemple: tableau).
-"count": retourne en octets la place approximative utilisé par lua.
-"step": il faut mettre la taille de la place que vous voulez tenter de liberer dans argument, et le programme va essayer de le libérer.(en koctets)
-"setpause": la valeur de argument se met dans la valeur pause du 'garbagecollector'. Retourne l'ancienne valeur de pause. Si pause=100, alors le garbage collector collecte tout le temps (de même pour des valeurs inférieurs). Par défaut, pause=200, c'est à dire que la garbage collector se lance lorsque l'utilisation de la mémoire double.
-"setstepmul": le step multiplieur prend la valeur de argument. Retourne l'ancienne valeur.
Si c'est 200 (valeur par défaut), le garbage collector a une vitesse normale. 100 signifie un controle plus poussé, et plus lent,...

--base.error(message[,niveau])
arrète le programme en renvoyant un message d'erreur donné par message.
Si niveau=1(défaut) l'erreur est complétée de la fonction dans lequelle l'erreur a été déclaré.
Si niveau=0 il n'y a pas d'informations supplémentaires.
Si niveau=2 on indique où on a appelé la fonction qui a rapporté l'erreur.
Si la fonction rapportant l'erreur a été appelé avec base.pcall, l'erreur est capturé par base.pcall et le programme ne s'arrète pas

--base.next(table [,index])
Cette fonction renvoi la prochaine case non vide du tableau. Si base.next(table) renvoi nil, la table est vide.

--base.pcall(fonction, argument1delafonction, argument2,...)
Lance fonction dans un mode protégé. Si une erreur arrive, elle est capturée et le programme ne s'arrète pas. Retourne true et les résultats si il n'y a pas eu d'erreur et false et l'éventuel message d'erreur sinon.

--base.tonumber(e [,base])
si e n'est pas un nombre ou un string convertible en un nombre, la fonction retourne nil. Sinon elle renvoi le nombre converti dans la base demandé (défaut = base 10)
en base 10 on peut avoir des nombres à exposants ou virgules, ce qui n'est pas le cas des autres bases. La base 10 est celle où sont écrit les nombres courants.
La base est comprise entre 2 et 36. Si la base n'est pas 10, le nombre doit être un entier.

--base.tostring(e)
Convertit n'importe quel e donné en string.

--base.type(v)
retourne dans un string le type de l'argument:
peut retourner: "nil", "number", "string", "boolean", "table", "function", "thread", ou "userdata"

--base.unpack(list [,i [,j]])
retourne tout les élements d'une table à une dimension entre i(1par défaut) et j(par défaut la longueur de la liste)
C'est pareil que en lua: return list[i], list[i+1], ···, list[j]

--base.getfenv(fonction)
retourne la tableau contenant l'environnement global de la fonction (le tableau où les variables globales sont lues et enregistrées pour cette fonction. Le tableau par défaut est la variable globale _G . (rq: _G._G=_G)

--base.setfenv(fonction,tableau)
change l'environnement global de la fonction

--base.setmetatable(table, metatable)
Attribue metatable comme metatable de la table. On ne peut que changer la métatable d'une table en lua. Si metatable vaut nil, alors enlève la métatable de la table, et si la métatable originale avait un index “__metatable” (métatable protégée) alors provoque une erreur.

La fonction retourne table.

--base.getmetatable(object)
Si object n'a pas de métatable, retourne nil.
Sinon si sa métatable a un index “__metatable”, retourne le contenu de cet index.
Sinon retourne la métatable de l'objet.

--table.insert(table [,pos],valeur)
insère valeur à la position donnée (si pos n'est pas donné, alors pose à la toute fin)dans la table.

--table.remove(table [,pos])
retire la valeur à la position donnée et décale le tableau. Retire la dernière valeur si pos n'est pas donné.

--string.byte (s [, i [, j]])
Retourne le contenu numérique des char `s[i]`, `s[i+1]`, ···, `s[j]`.
La valeur défaut de i est 1 et de j est i.

--string.char(...)
reçoit des nombres entre 0 et 255 et en fait un string. (en correspondance avec les valeurs ascii)

--string.sub(s, i [, j])
retourne la sous-string commençant au caractère i et finissant par j ème caractère.

Les fonctions de gestion des strings peuvent vous être très utile car:
-dans une string chaque caractère ne prend qu'un octet (au lieu de plus de 15 pour la case d'un tableau)
vous pouvez donc créer par exemple un map d'un niveau avec uniquement des nombres entre 0 et 255 et lire la map avec string.byte . Cela peut vous permettre de compresser des données.
ASTUCE: Stockez vos map dans des strings puis écrivez la string dans un fichier (librairie file). Réouvrez le fichier en mode "char". Hop: vous pouvez lire dans le fichier toutes les valeurs de votre

niveau (nombres entre 0 et 127).
-dans un sprite 5 couleurs(contenu dans un string), les 2 premières valeurs correspondes aux coordonnées à l'origine (x et y), c'est à dire que si x= 4 et y = 4, le sprite sera affiché) x+4 et y+4
Cela peut-être pratique pour éviter de recalculer des x et des y pour chaques sprites. Ces fonctions de manipulement des strings peut vous permettre à tout moment de changer les 2 premières valeurs du string du sprite.

2)Les fonctions non-graphiques spécifiques à nos graph

--key (numero de touche) attention : fonction usuelle
retourne false si la touche n'est pas pressée, sinon retourne true si elle est pressée.
Voici le tableau de correspondance des touches.
Si le numero est 0, renvoi si au moins une touche est pressée. (pas 100% fonctionnel sur g100, car renvoi des fois false même si quelque chose est pressé)

--misc.numcalc ()
retourne 100 sur g100 et 85 sinon

--misc.exit ()
le programme quitte brutalement et retourne au menu.

--misc.tostring2 (nombre)
retourne le nombre en chaine de caractère (uniquement nombres entiers entre -2147483647 et 2147483647). plus rapide que base.tostring . De plus le nombre n'est jamais affiché sous forme exponentielle.

--misc.contrast(nb)
Le nombre doit être 1 ou -1. diminue(-1) ou augmente le contraste(1) (attention: pas de limite est fixée et la calc peut ne pas supporter tout les contrastes)

--misc.math ([parametre] ,nombre)
Fait quelque chose de différent en fonction de paramètre:
-"ln" retourne ln(nombre) ln est le logarithme népérien (log(x) = ln(x)/ln(10)
-"exp" retourne exp(nombre)
-"sqrt" ou "racine" retourne la racine carrée du nombre
On peut aussi calculer la racine avec exp((ln(x))*0.5)), mais sqrt est plus rapide (et plus précis dans certains cas, comme les racines des grands nombres)

Si le calcul est impossible pour sqrt ou ln (nombre négatif par exemple) la fonction retourne nil

--misc.wait(nb de fois 2centisecondes)
met le proccesseur (et donc le programme) en pause pendant le temps indiqué.
Utilisez cette fonctions pour ralentir les menus,...
Elle permet d'économiser les piles (elle consomme 10 fois moins qu'une instruction normale sur g100 par exemple)
exemples:
local wait=misc.wait
wait(10) -- attends 0,2 secondes
wait(20) -- 0,4secondes
wait(50) -- 1seconde
wait(100) -- 2 secondes
wait(3000) – 1minute

Par contre à faible nombre, la fonction n'est pas exacte, alors utilisez wait(1) et wait(2) dans un boucle attendant une pression de touche afin de garder un bonne réactivité et diminuant drastiquement la consommation, mais ne l'utilisez pas dans une boucle de jeux, car en fonction de certains paramètres, l'attente est différente. (De plus le résultat est différent sur g100 et g85)

--misc.random(max)
renvoi un nombre aléatoire entre 0 et max-1. Le nombre renvoyé est un entier.
max doit être inférieur à 32767
Exemple: misc.random(1000) retourne un nombre entre 0 et 999

--int(nombre) attention : fonction usuelle
renvoi la partie entière d'un nombre.

--misc.chrono_set([numero_du_chrono])
permet de démarrage d'un chronomètre (deux emplacements : 1 (par défaut) et 2).
Si le chronomètre était déjà démarré, permet de le réinitialiser.
--misc.chrono_read([numero_du_chrono],[mode])
lit le temps chronométré par un chronomètre (le 1 si vous ne spécifiez pas le numéro)
deux formats sont possibles:

.si vous ne rentrez pas de second argument, alors il sera retourné un entier correspondant au temps passé en 1/50 de secondes sur g100 et en 1/64 de secondes sur g85.
.si vous rentrez un second argument, alors il sera retourné dans cet ordre le temps passé : centiseconde, seconde, minute

Le chronomètre est précis au 1/50e de secondes sur g100 et 1/64e de secondes sur g85.
La premier mode d'affichage permet de gérer des évènements arrivant à fréquence élevée (déplacement ennemi, ...), alors que le second mode permet plutôt d'avoir une idée du temps qui passe sur un durée plus longue.

--misc.modlist()
--misc.modload(nom)
Le lua est assez limité en place sur nos graph.

-> La solution est de charger un .lua principal et de charger ensuite, quand on a besoin, d'autres .lua (qui peuvent contenir des données sur un niveau, des fonctions, une librairie en lua, ...).
Sur g100 il faut que tout soit précompilé en .lc . C'est conseillé sur g85 car la lecture de .lua consomme beaucoup de mémoire.

Utiliser un module lua consomme un peu plus de mémoire que si on avait tout mis dans un seul fichier, mais cela apporte des avantages:
-On ne peut supprimer des fonctions de la mémoire que si c'est dans un module.
-Si on a beaucoup de données, on peut utiliser plusieurs modules pour les stocker. En chargeant un module à la fois on consomme moins de mémoire que si tout était dans un seul fichier.
-Cela permet aussi de faire des librairies.

A noter: seules les variables (et fonctions) globales dans le module sont visibles.

Un fichier .lua chargeable avec modload s'appelle un module.
Le .lua doit contenir une string finissant par "module nom_du_module" (le nom doit avoir une taille inférieure ou égale à 8)
modlist renvoi les noms de tout les modules trouvés.
modload tente de charger le module avec le nom indiqué.

Exemple 1: Je veux charger une librairie et je ne souhaite pas la décharger.

```
f,err = misc.modload ( nom)
if f == nil -- si il y a erreur, f = nil
 then
  print(err) -- err contient le message d'erreur
 end -- affiche l'erreur lors du chargement du fichier
f() -- execute le fichier (et donc défini les fonctions et les variables globales qui deviennent accessibles)
```

Remarque: f correspond en fait à une fonction qui contien tout le code du module. L'executer revient à lancer le programme contenu dans le module.

Exemple 2: Je veux charger un module, mais je n'ai pas confiance dans le contenu: j'aimerais qu'il ne puisse pas voir les variables globales que j'ai défini et qu'il ne puisse pas les modifier.
OU : Je veux charger le module et pouvoir le supprimer de la mémoire entièrement ensuite.

```
f,err = misc.modload ( nom )
```

```
if f == nil -- si il y a erreur, f = nil
 then
  print(err) -- err contient le message d'erreur
 end -- affiche l'erreur lors du chargement du fichier
tab = {base = base ; misc = misc ; nbdraw = nbdraw ; graydraw = graydraw}
base.setfenv(f, tab) – tab est maintenant l'environnement de f (contient toutes les variables globales)
f()
```

Toutes les variables globales de f seront définies dans tab. (on y accède avec ' tab. ')

Pour supprimer le module de la mémoire:

```
f = nil
tab = nil – il faut mettre nil aux 2 pour que ça marche
base.collectgarbage("collect") – on supprime tout le module d'un coup.
```

la librairie file:

cette librairie permet de stocker et de lire facilement dans un fichier stocké dans la ram de votre casio. Le nom du fichier doit commencer par LF et ne pas faire plus de 8 caractères. Il sera protégé d'un mot de passe pour prévenir la modification des données par l'utilisateur.
Le système permet de stocker juste ce que vous avez besoin dans un minimum de place.
Par exemple un nombre normal lua prend 4 octet dans un fichier, alors que sinon il en prend bien plus (environ 20 dans une variable locale sur g100 et plus sur g85)
Les fichiers sont ouverts sous différent mode selon le contenu des données:
"string" est le mode pour stocker des string (1 octet par caractère)
"char" est pour les entiers de -128 à 127 (1 octet)
"int" est pour les entiers de -32 768 à 32 767 (2 octets)
"float" est pour les nombres normaux utilisés par le lua (4 octets)
les données du fichier ouvert sont contenu dans une variable du type userdata.

Un fichier ne peut pas être ouvert si la taille ne correspond pas à un multiple de le taille du type demandé.
La taille totale du fichier est limitée à 32750.

A chaque création de fichier/changement de taille/suppression , il faut rechercher de nouveau tout les fichiers pour pouvoir lire/ecrire dedans/les supprimer (sauf celui que l'on vient de manipuler).

--file.new(nom,nombre_de_cases,[mode])
crée un fichier et renvoi l'userdata associé. Il y a une erreur si le fichier existe.
La taille réelle du fichier est un multiple du nombre de cases.

--file.search(nom,[mode])
renvoi un userdata contenant les données sur le fichier recherché et nil s'il n'existe pas.

--file.mode(fichier)
renvoi le mode d'ouverture du fichier

--file.resize(fichier,nombre_de_cases)
change la taille du fichier

(indisponible sur g85)

--file.delete(fichier)
supprime le fichier (et l'userdata associé est inutilisable)

--file.length(fichier)
renvoi le nombre de positions disponibles dans le fichier (pareil que #fichier)

--file.ramsize()
renvoi la place restante dans la ram.
(non disponible pour g85)

--file.read(fichier,position,[fin])
renvoit le nombre contenu à position.
Si on attends une string, il faut indiquer la position finale. (1 caractère = 1case)
La première case est numérotée 1.

--file.write(fichier,position,nombre)
écrit le nombre (ou la string) à position.

Dans tout les cas, dès que vous utilisez une fonction à tort (en dehors du fichier, fichier non réactualisé, ...), une erreur est envoyée et le programme s'arrête, sauf si l'erreur est capturée par un pcall.

Exemple: pour rechercher un fichier et le créer s'il n'existe pas ou n'est pas de la bonne taille:

```
local delete = file.delete
local new = file.new
local search = file.search

-- mode est un argument facultatif
local function search_or_create ( name , taille,mode )
	local result = search (name , mode)
	if result then
		if #result == taille then result
		else delete(result) ; return new(name,taille , mode)
		end
	else
		return new(name,taille mode)
	end
end
```

b) le noir et blanc

Le mode noir est blanc a un repère partant du coin en haut à gauche de l'écran (le point 1,1) finissant en bas à droite (64, 128)
On a accès à une ligne et une collone de plus qu'en basic.

L'écran est aussi divisé en collones et lignes pour savoir où l'on va écrire avec nbdraw.print . Cela correspond à la fonction locate en basic, sauf que l'on a droit à une ligne de plus (collone de 1 à 21 et ligne de 1 à 8)

En lua, on a accès à des pictures virtuelles qui correspondes à des 'pages' d'écran. (En gros si vous voulez vous avez accès à 5 buffers.)
La page 0 est la page principale où tout est affiché. On peut copier la page 0 sur la page 2 ou 3,... et inversement. On peut écrire sur une autre page et copier sur la page 0, ainsi on évite que l'utilisateur voit le dessin se dessiner (et cela évite le clignotement)
la page maximale est 5.

Si vous ne comprenez pas le système de buffer, contendez vous de savoir que ce qui s'affiche est sur la page 0 qui est l'endroit où les fonctions écrivent par défaut.

--nbdraw.print(nombre ou/et booléen(true ou false) ou/et string,[autant de paramètres que l'on veut (max 200)])
affiche tout ce que vous demandez en noir et blanc à l'endroit ou se situe le curseur virtuel (invisible) (voir nbdraw.getcursor pour savoir ou est le curseur)
Le curseur se place après le dernier caractère que nbdraw.print a écrit.
Un nbdraw.print ne doit pas afficher plus de 190 caractères (plus grand que l'écran, heureusement)
Les caractères spéciaux sont:

- '%' doit être écrit %% pour être affiché (Ne pas mettre % tout seul!)
- '\n' fait un saut de ligne sans retour à la ligne
- '\r' revient au début de la ligne (donc la passage à la ligne suivante est '\n\r'. bizzarement, '\r\n' provoquera un erreur lors du chargement "bad constant")
- `$` ne doit jamais être écrit
- on peut accèder à des caractères secondaires (voir tableau p34 du manuel du programmeur http://gprog.tk/) . On peut accèder à la seconde collone de tableau en plaçant \246 juste avant un caractère. (pas encore dispo sur g85)
  Par exemple je peut afficher la flèche simple vers la droite avec : "\246\157", car \abc converti abc en caractère selon le code ascii (ex: \044)
- on peut afficher ' et " avec \' et \"

---nbdraw.getcursor ()
retourne la collone et la ligne du curseur où va écrire nbdraw.print

--nbdraw.setcursor (ligne,colonne)
place le curseur où nbdraw.print va écrire
Cela correspond à placer où l'on va écrire comme la fonction locate en basic sauf que l'on écrit rien. On a droit à la 8e ligne innaccessible avec locate en basic.

--nbdraw.precision(nb)
Attribue nb à la précision de la conversion par défaut des nombres (base.tostring et nbdraw.print) et chaine de caractère. Par défaut elle est de 6 chiffres significatifs (le reste est arrondi)
Si vous avez des imprecisions dans vos calculs, exemple: vous devez afficher 2.349999, la précision de 6 chiffres affichera 2.35 ou le nombre précédent si elle était de 7.

--clear(page) attention : fonction usuelle
efface la page que vous avez indiquée (celle affichée : 0)

--nbdraw.pixeltest (x,y,[page])
renvoi 1 si la pixel est allumé, 0 sinon

Remarque: le repère est de haut en bas et de droite à gauche.
Le pixel en haut à gauche (innaccessible en basic) a pour coordonnées x=1 et y=1
et celui en bas à droite x=128 et y =64.
C'est à dire que le basic Pixtest(30,20) devient nbdraw.pixeltest(31,21) (le repère est décalé d'une ligne et une collone, puisque une ligne et une collone de plus sont disponibles.)

--nbdraw.pixel(x,y,[couleur],[page])
affiche un pixel blanc (couleur = 0) ou noir (couleur=1, par défaut) sur la page principale ou celle que vous indiquez. Si vous écrivez en dehors de l'écran, le pixel ne sera pas affiché.

--nbdraw.line(x1,y1,x2,y2,[page])
affiche une ligne noire du point (x1,y1) au point (x2,y2) sur la page principale ou celle que vous indiquez.

--nbdraw.copypage(page source, page destinataire)
Vous avez dans doute remarqué que de nombreuses fonctions peuvent prendre une 'page' comme argument.
En effet la page 0 est la page affichée et les pages de 1 à 5 sont des pages disponibles pour des dessins (vous pouvez dessinez qqch pendant que vous affichez autre chose)
Cette fonction permet par exemple de copier la page 5 sur la 0 pour afficher le contenu de la page 5.

c)les 5 couleurs (blanc – gris clair – gris moyen – gris foncé – noir)

Le repère utilisé (position des pixel avec x et y) est le même qu'en noir et blanc.
Le mode 5 couleurs s'active avec graydraw.setcolor(true) et s'éteint avec graydraw.setcolor(false)
Il s'éteind automatiquement lorsque l'on quitte.

Ce que l'on dessine est toujours d'abord dessiné dans un écran invible (buffer), puis une fois que l'on a tout dessiné, on demande à afficher l'écran invisible à l'écran (pendant que l'on redessine sur l'écran invisible, l'écran visible continu d'afficher l'ancienne image)

L'atout du mode 5 couleurs est que l'on peut utiliser des sprites: c'est à dire que l'on peut afficher un morceau d'image que l'on a prédessiné à n'importe quel x ou y (l'image peut même dépasser de l'écran sans bug)

Rq: sur g85 le gris foncé est en fait remplacé par: 1pix sur 2 du gris moyen et 1 pix sur 2 du noir.

Dessiner un sprite:
Pour dessiner un sprite, utilisez le logiciel **sprite maker** en mode C/C++ gxlib10 sur pc.
Une fois le sprite dessiné, enregistré et généré, copiez collez le contenu entre les guillemets entre les guillements de la table foo du fichier lua sprites.lua
Lancez dans windows sprites.lua (avec le lua pour windows)
un ficher Prog.lua est apparu avec le sprite sous forme de string et prète à être mise dans le .lua de votre projet.
Tout les sprites sont compressés dans des strings!
Remarque: si Prog.lua existe déjà il est écrasé.
Le string contenant un sprite peut apparaitre bizzare, ne vous en inquiétez pas.

Code du fichier sprites.lua (à executer avec lua windows et non pas le lua g100/g85):

```
foo = { /le code du sprite/ }
```

```
out = io.open("Prog.lua", "w") – l'ancien fichier Prog.lua sera effacé
foo2=string.char(unpack (foo));
out:write(("sprite = %q"):format(foo2));
--Prog.lua contient la string associé au sprite.
out:close()
```

Pour afficher un sprite:

--spritexy x,y, sprite_sous_forme_de_string attention : fonction usuelle
afficher le sprite aux coordonnées indiquées.

Exemple:

```
graydraw.setcolor(true) –on active le mode 5 couleurs

ballon ="bapzefbpazefpqkdù$"éù'*é""    --pas vraiment un sprite ici, mais ce sera quelque chose du genre
... --code
clear nil –efface l'écran invisible
spritexy 20, 30 , ballon -- dessine le sprite ballon dans l'écran invisible

refresh  --dessine l'écran invisible à l'écran
```

--graydraw.setcolor(true ou false)
active(true) ou désactive (false) le mode d'affichage 5couleur (au démarrage du programme, le mode d'affichage est Noir et blanc)

--graydraw.text(x,y,caractères ou nombre)
affiche aux coordonnées entrées une chaine de caractère ou un nombre.
Si on entre un nombre, il est converti en un entier entre -2147483647 et 2147483647 car l'algorithme utilisé est beaucoup plus rapide que l'algorithme normal (tostring) qui est très lent.(il s'agit de l'algorithme rapide de misc.tostring2
Si vous voulez vraiment afficher un nombre à virgule ou plus grand, utilisez:
`graydraw.text(x,y,base.tostring(nombre))`

Tout caractère ne peut pas être affiché: seul peut être affiché ces caractères (les minuscules sont converties en majuscules):
` ' ( ) * + , - / 0 1 2 3 4 5 6 7 8 9 : ; < > = ? @`
` ABCDEFGHIJKLMNOPQRSTUVWXYZ`
Les caractères sont affichés en noir avec 5 pixels de hauteur, 4 de largeur + 1 pixel d'espace entre chaque caractère.

--clear nil attention : fonction usuelle
efface l'écran invisible

--refresh attention : fonction usuelle
dessine l'écran invisible à l'écran.

--graydraw.pixel(x,y,[couleur])
dessine un pixel d'une couleur de 0(blanc) à 4 (noir) (1:gris clais 2: gris moyen 3: gris foncé) dans l'ecran invisible, et 5 inverse le pixel.

La couleur par défaut est le noir.

--graydraw.line(x1,y1,x2,y2,couleur)
Dessine une ligne de la couleur désirée (entre 0: blanc et 4: noir (5: inverser les couleurs)) dans l'écran invisible.
Les points en dehors de l'écran ne sont pas dessinés

--graydraw.rect(x1,y1,x2,y2,couleur1,couleur2)
Dessine un rectangle de contour couleur1 et de intérieur couleur2 .

--graydraw.pixeltest(x,y)
renvoie la couleur du pixel dans l'écran invisible

--graydraw.fill(couleur)
rempli l'écran invisible de la couleur donnée. 5 inverse l'écran.

--state(false ou true) attention : fonction usuelle
C'est la fonction qui permet d'égaler un programme C!
state 'sauve' (false) le décor de manière très rapide pour le ressortir quand on le souhaite (true).
Ainsi vous pouvez dessiner un décor très fournit et le recharger en un rien de temps dans l'écran invisible.
false: sauve l'écran invisible
true: remet l'écran invisible à l'état où il était quand il était sauvegardé.
Remarque: une sauvegarde de l'écran suffit pour le recharger autant qu'on le désire.

--graydraw.map(image en string)
Ceci affiche une map dessinée avec sprite maker (il s'agit d'un mode spécial en dehors des sprites: "convertir et coder une image 128*64")

--scrollx ou scrolly (x ,couleur) attention : fonction usuelle
Vous avez affichez une map , et vous voulez la faire défiler sans tout redessiner, alors cette fonction est pour vous!

Un scroll signifie décaler tout les pixels de l'écran d'un coup. Les pixels qui disparaissent d'un coté réapparaissent de l'autre (vous pouvez les effacer et completer la map)

x est entre -8 et 8 (0 ne fait rien)

scrollx décale les pixels au niveau des x: si x est négatif, les pixels se décalent vers la gauche (leurs x diminue) si x = -1, ils se déplacent d'un pixel, si x = -3, de 3 pixels ,...
Si x est positif, les pixels se déplacent de la même manière vers la droite.

scrolly fait la même chose que la fonction précedente, mais sur l'axe des y.
Si x est négatif, les pixels vont vers le haut et inversement.

scrollx a la même vitesse peu importe la valeur de x. scrolly 1 100 est 8 fois plus rapide que scrollx maisla vitesse de scrolly y 100 dépend de y.

Si couleur est entre 0 et 5, alors ce sera remplacé par une ligne de la couleur indiquée.(0:blanc

4:noir 5: inverser) (sur g85, la couleur 3 ne donne pas un très bon rendu)
sinon (exemple: couleur = 100) ce qui sort d'un coté de l'écran réapparait de l'autre

L'interet de décaler tout les 2 ou 3 pixels au lieu de 1 est dans le cas où le programme fait un nombre important de calcul, et que le programme serait plus fluide si le scrolling est de plus de 1 pixel.
Un scrolling de seulement 1 pixel peut aussi dans certains cas faire apparaitre aux yeux l'image floue (l'oeuil distingue mieux un décalage rapide de 2 pixels).

d)les fonctions usuelles

Vous avez sans doute remarqué les 'attention : fonction usuelle'.
Certaines fontions sont appelées de nombreuses fois dans des programmes avec un affichage graphique dévellopé, et ce genre de programme néccessite le plus de fluidité possible, donc j'ai regroupé certains fonctions en les codant d'une certaines manière que leur appel est plus rapide.

Ce que l'on ne peut plus faire avec les fonctions usuelles:

```
local math=misc.math -- fonctionne
local touche= key  -- ne fonctionne pas
```

En effet, pour utiliser ces fonctions, il faut être plus stricte, et vous ne pourrez pas précompiler si vous ne respectez pas la syntaxe requise.
Mais vous pouvez faire par exemple:

```
clear 0
clear (0)
clear x -- avec x= 0
clear (x)
clear nil
local touche= key(5)  -- contient true si exe était pressé, sinon false
if key(5) then ... end  -- execute si exe était pressé.
scrolly 1,4 -- ne pas faire (1,4)
scrollx x,(-y)
spritexy x, y, sprite
spritexy (60), y, sprite
state true
state (false)
refresh
....
```

remarque:
--clear (num)
en Noir et Blanc, num doit contenir le numero de la page à effacer (page principale= ce que l'on voit à l'écran = page 0)
en 5 couleurs, num n'est pas important, mais vous devez le mettre. Conseil: mettez nil qui prend moins de place.

En fait une fonction usuelle est codée comme une opération de base du lua (+, ...) ce qui fait que son appel est quasi instantané et que contrairement à tout les appels de fonctions (lua ou C) il n'y a

pas de gestion de la mémoire (aucune allocation 'au cas où' et pas de perte de temps)

III/optimisations

A quoi sert t'il d'avoir le code le plus rapide possible?

-> c'est plus agréable pour l'utilisateur
-> dans le cadre d'un programme graphique; il faut savoir que plus l'affichage se 'raffraichis' (c'est à dire que l'image à afficher se met à jour) rapidement, plus on a l'impression que le mouvement est fluide (le jeux est agréable à regarder)
pas contre si le mouvement est trop rapide, on a l'impression de flou, d'où par exemple dans le jeux "ballon", les ballons ne montent pas d'un pixel par pixel, mais de 45% d'un pixel ou plus en fonction du niveau. (lorsqu'ils sont affichés avec une coordonnée à virgule, en réalité la fonction de sprite ne voit que la partie entière des coordonnées (comme toutes les fonctions graphiques qui le font intantanément))

Quelques optimisations à effets importants:
-Evitez au maximum d'appeler des fonctions (lua ou C) dans la boucle principale de votre programme. (sauf dans le cas des fonctions usuelles dont l'appel est instantané)
-Diminuez au maximum les calculs
-Utilisez les particularités du lua:

par exemple:

```
if key(5) == true then x=1 else x=y end
```

-> 1ère optimisation :

```
 if key(5) then x=1 else x=y end
```

-> 2e optimisation :

```
x = (key(5) and 1) or y
```

en effet or renvoi le premier résultat vrai et and le dernier (sauf si c'est faux) et un nombre est toujours considéré vrai

par exemple si exe (key(5)) est pressé:

vrai and 1 renvoie 1

-Utilisez la fonction state pour charger un décor permet de faire un bonne interface graphique tout en accélérant le code .
-Passez en local toutes les fonctions que vous pouvez
-while not condition do ... end est plus lent que son équivalent repeat ... until condition.

Remarque:

Un mouvement est saccadé à partir tu moment où la fréquence de rafraichissement est inférieur à celle de l'écran.
Sur g100 l'écran se raffraichit toutes les 0,1 secondes.
Si on raffraichit l'image toutes les 0,2 secondes, l'image sera un peu saccadée.
Si l'image se raffraichi toute les 0,04 secondes environ, le mouvement est fluide.

IV/ Remarques

-seclua (sur g100) est un programme basic qui est là pour empècher qu'un bug insoluble du lua qui à son lancement écrit quelque part au début de seclua, d'où le fait qu'il doit être en première position.
Seculua est aussi utilisé pour stocker les données de state (donc au rédémarrage d'un programme, les données de state contenues de la fois précédente peuvent avoir été très légèrement modifiés)
Il contient aussi (dans l'endroit plus sécurisé) les sprites de caractères.

-sur g85 , il reste quelques bugs. Si vous avez un plantage juste après avoir lancé le gris, mettez une petite attente avant d'activer le gris.

-un local est plus rapide et plus petite qu'une variable globale. Une fonction peut être local est donc son appel est plus rapide
-les goto n'existent pas en lua, il est donc difficile de retranscrire en lua un programme avec des goto. Cependant on peut couper un programme avec des goto en fonctions, à condition que l'on s'arrange pour quitter chaque fonction avant dans lancer une autre (voir programme bourse pour comprendre)

-chaque appel d'une fonction (lua ou C) alloue de la mémoire pour un éventuel retour de la fonction. Cet espace n'est libérable pas le garbagecollector qui si l'on a quitté la fonction (donc si A appelle B, la mémoire de A n'est pas libérable). Il est donc préférable dans une boucle d'éviter au maximum d'appeller une fonction chaque tour (pour la fluidité et le fait que l'on devrait souvent liberer la mémoire)
Si des fonctions s'entre-appellent à l'infini (A appelle B qui appelle A,...) vous aurez à un moment une pénurie de mémoire.

-le ramasseur d'ordure ne se déclenche que lorsque l'utilisation de la mémoire augmente beaucoup. On peut l'arrèter ou le déclencher avec la fonction base.collectgarbage.
Si il n'y a plus assez de mémoire il se déclenche de force pour en liberer, mais il peut échouer. Pour aider à gérer la mémoire, mettez à nil les objets que vous n'utilisez plus (c'est fait par défaut pour les variables locales d'une fonction que l'on quitte). De plus si vous mettez beaucoup de variables à nil d'un coup, forcez une collection.

Voici différents exemples de code qui peuvent vous aider:

-si vous avez une erreur du genre "invalid comparison, x is nil":

```
x=map[i] --par exemple
..... --code
if (x>100) --l'erreur est signalée ici
```

Dans ce code on remarque que cela vient probablement de la case i de map qui doit être nil. Pour connaitre i et comprendre l'erreur on peut faire:

```
..... --code
if (x) then base.error("\n\rerreur x est nil.\n\r la valeur de i était:\n\r
i="...tostring(i))
if(x>100)
```

Ainsi quand x sera égal à nil, au lieu d'avoir le message d'erreur précédent, vous aurez le message d'erreur:
x est nil.
La valeur de i était:
i=433
(le valeur de i est donnée en exemple.)

Une fois l'erreur trouvée, vous pouvez supprimer la ligne avec le test pour ne pas ralentir votre code.

-Vous pouvez vous aussi capturer l'erreur d'une fonction sans arrèter le programme.

```
function mafonction (i)
.... --code
end


....
arg= 5 --par exemple
a,b=base.pcall(mafonction,arg)
if (not a) -- il y a eu une erreur (equivalent à if (a == false) )
then affichererreur(b)  end --b est un string contenant le message d'erreur
.... --dans tout les cas le programme continu. (sauf erreur memoire)
```

-Vous pouvez vous même creer votre propre message d'erreur

```
function mafonction (i)
type=base.type(i)
if(type  ~= "number") then -- ~= signifie différent de
base.error("erreur d'argument, il faut rentrer un nombre, non pas un"...type,0)
end
.... --code
end
```

- Si vous voulez défiler les cases d'un tableau, deux cas se présentent à vous:
. Si le tableau ne contient que des cases numérotées par des entiers positifs supérieurs à 1, le plus rapide est d'utiliser une boucle for et d'accéder directement aux cases du tableau.
. Si le tableau contient aussi des cases que l'on n'accède pas avec un entier positif supérieur à 1 (par exemple la case 0, la case "feuille", ...), alors le plus rapide est d'utiliser le code suivant:

```
local next = base.next
for k,v in next, mytable do
.... -- k contient l'indice tel que mytable[k] = v
-- la boucle défile toutes les valeurs de k possibles
end
```

- Pour stocker des données (comme des maps), vous pouvez utiliser les string ou les fichiers. Le lecture d'une case d'un fichier (file.read) et la lecture de l'entier codé par une lettre de la string (string.byte) se fait quasiment à la même vitesse. Vous pouvez donc utiliser les deux systèmes.
Il faut cependant admettre que l'accès d'un tableau est plus rapide (deux fois plus rapide), mais les tableaux prennent énormément plus de mémoire qu'une string

Remerciements:

De grands remerciements à tout ceux qui m'ont aidé dans ce projet. La liste est longue (Orwell, PierrotLL, Eiyeron, Louloux, Purobaz, light_spirit, kristaba,... et j'en oublie)