Guide de formation

Formation : Apprendre à programmer des jeux vidéo - Bases complètes de programmation

Site officiel : www.gamecodeur.fr
Auteur : David MEKERSA

Version 1.1a
28/03/18 - Correction des liens dans la section BASIC (merci à kenesy05)

Sommaire :

IMPORTANT :Les vidéos de formations correspondant à ce support PDF sont accessibles ici :

http://gamecodeur.fr/module-1-lua-love2d

Comprendre ce qu'est la programmation

C'est quoi du code ?

Le code ce sont des instructions, sous forme de texte, qui constituent un programme.
On parle aussi de **code source**. Le code se divise en lignes. On parle de "**lignes de code**".

Un code, c'est comme écrire une liste de courses pour quelqu'un, avec des choix ou des conditions qu'il doit suivre à la lettre. On dit "**Exécuter le code**" ou "**Interpréter le code**".

```
Début du programme

Va au supermarché !
Cherche du beurre
Si tu ne trouve pas de beurre
      Prend de la margarine
Sinon
      Prend du St Hubert Omega 3
Fin
Reviens du supermarché

Fin du programme
```

Tous les langages de programmation sont similaires, mais leur syntaxe diffère.

La **syntaxe** c'est comme avec une langue étrangère : ce sont les verbes, les noms, la manière dont les mots sont ordonnés, et leur orthographe.
Certains mots, dans un code, sont appelés "**mots clés**". Ils ont une orthographe très précise.

L'ordinateur est bête et méchant, la syntaxe d'un langage est très précise. On ne peut pas faire de faute dans un mot clé, sinon il ne le comprend pas.
Exemple de mots clés : if / then / else / print / for.

Les mots clés sont en anglais, mais inutile d'être bon en anglais car ils sont peu nombreux, on peut facilement les apprendre.

Comment on apprenait facilement en 1985 grâce au BASIC (et ça vaut toujours aujourd'hui !)

Quand on apprenait à programmer sur un Amstrad CPC, ou encore sur un Commodore 64, en 1985, on comprenait très rapidement comment programmer. Pourquoi ?

1) Le langage était simple, il s'appelait d'ailleur le **BASIC**. Il existe toujours.
2) La liste de ses mots clés et leur utilisation tenait en quelques pages dans le manuel.
3) Les lignes de code des programmes BASIC étaient numérotées, on comprenait donc immédiatement comment l'ordinateur les interprétait.
4) Le langage était intégré à l'ordinateur, on allumait et on tapait du code. Directement.
5) A l'époque, programmer n'était pas élitiste. La simplicité était considérée comme une vertue, et non comme une tare comme parfois aujourd'hui.

Voici comment on pouvait programmer à l'époque, et votre première leçon de programmation !

```
5 cls
10 input "Quel est votre âge";age
20 print "Vous avez";age;"ans !"
```

Comme vous le voyez, les lignes sont numérotées, on comprend ainsi aisément dans quel ordre l'ordinateur les interprète.

Maintenant avec des conditions, comme dans la liste de courses :

```
5 cls
10 input "Quel est votre âge";age
20 if age < 13 then goto 50
30 if age < 20 then goto 60
40 if age >= 20 then goto 70
50 print "Vous êtes vraiment jeune pour apprendre à programmer
!":end
60 print "Bravo, vous êtes un jeune adolescent qui ira loin !"
65 end
70 print "Vous faites moins de";age;"ans dites-moi !"
75 end
```

On peut toujours programmer en BASIC aujourd'hui, tout du moins dans des langages qui s'en inspirent. Mon 1er jeu commercial "Geisha : Le jardin secret", est codé en **Blitzmax**. On retrouve dans ce langage la syntaxe du BASIC.

Voir https://nitrologic.itch.io/blitzmax. Il est maintenant gratuit et toujours utilisé pour créer des jeux professionnels.

Pour en savoir plus et découvrir le manuel d'un ordinateur dans les années 80, regardez par exemple le manuel de l'Amstrad CPC :
https://archive.org/details/Amstrad_CPC464_Guide_de_lutilisateur_1984_AMSOFT_FR

Comment l’ordinateur va interpréter votre code et comment penser comme lui

L’ordinateur interprète votre code dans l’ordre en suivant des règles simples, notamment celle de les interpréter dans l’ordre…

↓ Ligne 1
↓ Ligne 2
↓ Ligne 3

On parle aussi d’EXECUTER du code.

Il est indispensable d’intégrer comment l’ordinateur va interpréter votre code.

Il le fait donc de manière séquentielle. Ligne par ligne, dans l’ordre. C'est la clé pour comprendre.

Le code d’une ligne peut obliger l’ordinateur à faire un “saut”, c’est à dire ne pas passer à la ligne suivante mais à aller directement à un autre endroit du code.

C’est le cas dans notre exemple en BASIC avec les GOTO (qu veut dire "Aller à").

Nous découvrirons plus tard les conditions, les boucles, les fonctions, qui vont avoir ce type d’influence sur le déroulement du code. Sans ces “sauts”, on ne pourrait pas programmer car le code ferait toujours la même chose.

En apprenant à penser comme votre ordinateur, en déroulant mentalement l’ordre dans lequel votre code est exécuté, vous ferez moins d’erreurs et serez un meilleur programmeur.

Vous pouvez, et devez, penser comme lui et être capables de lire votre code et de l’exécuter mentalement. Cet exercice va vous aider à mieux apprendre au départ, mais aussi par la suite à coder en faisant moins d’erreurs, et à trouver les erreurs beaucoup plus rapidement.

Configurer son ordinateur pour programmer

Quel matériel pour programmer ?

- N’importe quel PC ou MAC convient pour apprendre et créer un premier jeu simple, en 2D (2 dimensions).
- Une connexion Internet (mais là je vous apprend rien…)

Présentation du langage Lua, et pourquoi c’est le langage idéal pour commencer

- Le langage Lua (prononcer Lua, et non L.U.A. car c’est un mot, ça veut dire Lune en Portugais) a été inventé en 1993.
- Lua est un “langage de script” car il est interprété par l’ordinateur.

C’est à dire que l’ordinateur lit votre code au moment où il l’exécute. D’autres langages comme le C++, sont des langages qui sont “compilés” et ainsi, l’ordinateur n’exécute pas directement le code C++ mais un programme qui a été précédemment compilé par le programmeur. Le compilateur transforme le code dans un autre code qui est le langage maternel de l’ordinateur : le langage machine.

- Lua est le langage idéal pour commencer :
    - Très peu de notions à comprendre pour commencer (proche du BASIC)
    - Syntaxe légère, peu de mots "magiques"
    - Idéal pour commencer le jeu vidéo grâce à Love2D
    - Pas de "compilation"
    - Universel, utilisé dans le monde entier et libre de droits
    - Il permet de créer des jeux pros
    - Tremplin pour les autres langages

Installation des outils nécessaires pour apprendre à programmer des jeux vidéo en Lua

Pour suivre ma formation, il faut installer 2 choses :

1) **ZeroBraneStudio** (téléchargez la version "Gamecodeur" sur la page de l'atelier : https://www.gamecodeur.fr/liste-ateliers/module-1-lua-love2d/)

C'est l'éditeur dans lequel vous allez saisir votre code Lua et le tester. Il va vous faciliter la vie.

NOUVEAU : Il est préconfiguré avec Love2D (le "moteur" 2D utilisé pour les cours).

On peut utiliser d'autres éditeurs que ZeroBraneStudio.

J'utilise par exemple Sublime Text 2, mais sa configuration est moins aisée (voir https://www.youtube.com/watch?v=evmHMjGxfPI). Vous êtes libres de l'utiliser.

2) **Love2D**

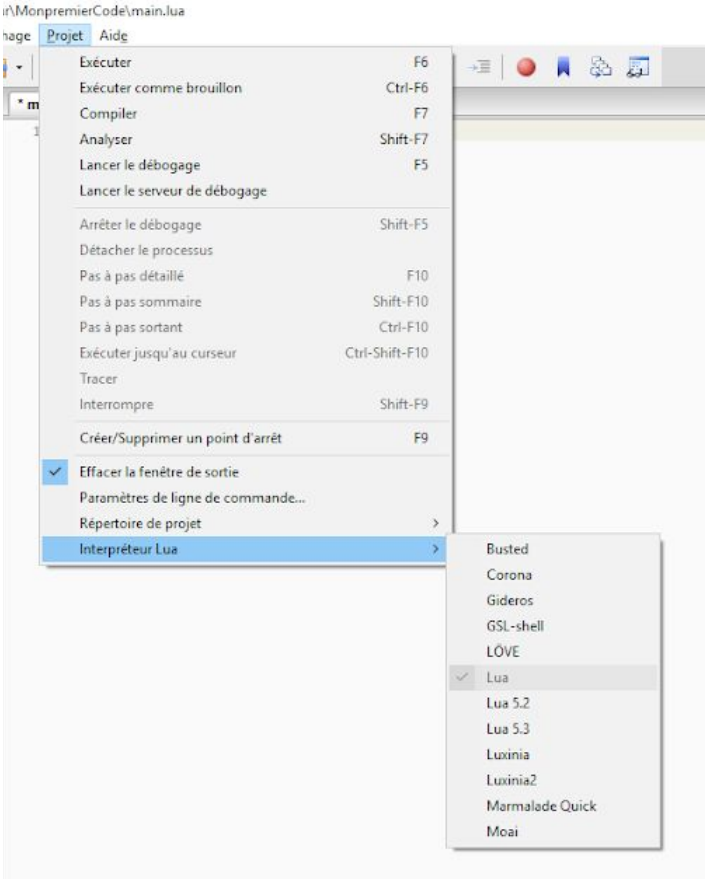
- C'est un "framework" (un ensemble de composants logiciels)
- Il contient le langage Lua et un ensemble de fonctions pour créer des jeux (afficher des images, jouer des sons, etc.).

Important : Il est préinstallé avec Zerobrane Studio Gamecodeur Edition donc vous n'avez rien à faire (voir ci-dessus). Mais si vous utilisez un autre éditeur il faudra l'installer :

https://love2d.org/
Pour savoir comment l'utiliser c'est ici

Votre première ligne de code pour vérifier que tout fonctionne !

Par défaut, ZeroBraneStudio est configuré pour interpréter du Lua. Le choix de l'interpréteur se fait ici :



1) Créez un répertoire et nommez le par exemple “MonPremierCode”
2) Lancez **ZeroBraneStudio**
3) Dans le menu choisissez Project/Project Directory/Choose
   (ce qui veut dire Projet / Répertoire du projet / Choisir)
4) Choisissez le répertoire “MonPremierCode”
5) Créez un nouveau fichier de code (1er icône, la feuille avec l’étoile)
6) Saisissez :
   ```
   print("Hello !")
   ```
7) Sauvegardez (bouton en forme de disquette)
   Nommez le fichier “***main.lua***”
8) Cliquez sur le bouton “Start” (la flèche verte simple)
9) Regardez si vous avez bien écrit “Hello !” dans la partie inférieure de ZeroBrane

Bravo !

Apprendre facilement à programmer avec les 5 fondamentaux

Introduction

Les 5 fondamentaux, dans l'ordre de ma méthode

1) Les variables et les expressions
2) Les fonctions
3) Les structures de contrôle
4) Les tableaux et les listes
5) Les objets

Pourquoi ces 5 fondamentaux vont faire de vous un vrai programmeur

- En apprenant ces 5 fondamentaux, vous pouvez déjà commencer à programmer
- En les apprenant dans l'ordre de ma méthode, vous allez apprendre vite
- En les appliquant à un langage simple comme le Lua et avec un outil simple comme Love2D, vous pourrez créer des jeux vidéo
- On peut transposer ces connaissances à quasiment tous les autres langages

Recommandations

- Approfondissez chacun de ces concepts, expérimentez, jusqu'à les maîtriser
- Prenez cela comme un jeu et donnez vous le temps de comprendre
- Nous utiliserons 2 tirets pour taper des lignes de code qui seront ignorées par l'ordinateur :
  ```
  -- Ceci est un commentaire
  ```
- Nous utiliserons ce code pour afficher des "traces" afin de voir si notre code fonctionne :
  ```
  print(valeur)
  ```

1 - Apprenez ce qu'est une variable et une expression

Les variables servent à stocker des données

Pour programmer vous devez pouvoir stocker des données dans la mémoire de l'ordinateur.

Voici des exemples de données : le nombre de vies de votre héros, son énergie, son nom, la liste des vaisseaux ennemis, …

Ces données, on les appelle aussi des "valeurs".

Les valeurs ont un "type", dont les **3** plus courants sont :

- Les valeurs **numériques** (les nombres), avec lesquels on peut faire des calculs.
  Exemple : 5
  Ou encore un nombre à virgule comme : 5.2 (on appelle cela un nombre "**flottant**")
- Les valeurs **booléennes**, qui ne peuvent contenir que 2 valeurs : vrai (true) ou faux (false). On utilise ces deux mots clés : `true` et `false` pour désigner leur valeur.
  Exemple : `true`
- Les **chaînes de caractères** (les textes), que l'on utilise couramment pour afficher du texte. Exemple "Nombre de vies : ".
  Les chaînes sont encadrées de guillemets.

Comment créer une variable

Une variable c'est comme une boite sur laquelle on inscrit un nom et on met une valeur à l'intérieur.

On dit "variable" car les valeurs peuvent changer pendant leur durée de vie (une partie de votre jeu vidéo par exemple).

Par exemple quand le héros perd une de ses 5 vies, la variable passe de 5 à 4.

On donne un nom à une variable, pour se souvenir de ce qu'elle contient. Voici des exemples de noms :

- `energie`
- `vies`

IMPORTANT : Le nom est sans espaces, sans accents, sans caractères spéciaux et commence par une lettre.

Pour donner une valeur à une variable, on utilise le signe égal

- `energie = 100`
- `nom = "conan"`

Mise en pratique

Testez ce code :

```
energie = 100
print(energie)
energie = 150
print(energie)
nom = "conan"
print(nom)
```

Et voici deux exercices :

Créez une variable pour stocker un score.
Donnez lui le nom de votre choix et une valeur de 0.
Affichez la.

Créez une variable pour stocker le titre de votre jeu.
Donnez lui le nom de votre choix et affichez le.

Les expressions

Une expression décrit un calcul qui donne une nouvelle valeur.
Exemple : `1 + 1`
Cette expression va donner 2 comme nouvelle valeur (le résultat de 1 + 1).

Une expression peut utiliser des variables en tant que valeurs.
Exemple : `vies - 1`

Le résultat dépend du contenu des variables au moment où l'expression est calculée. Si la variable *`vies`* contient 5 au moment où l'expression est exécutée, le résultat vaudra 4.

Les signes utilisables couramment pour des calculs sont :

```
+ (additionner)
- (soustraire)
* (multiplier)
/ (diviser)
% (modulo)
```

Pour additionner des chaînes de caractère, le signe est un double point

```
nouvellechaine = chaine1..chaine2
```

On dit qu'on ***concatène*** 2 chaînes de caractères. C'est très utile pour construire des phrases qui contiennent des valeurs. Exemple :

```
chainescore = "Vous avez gagné "..score.." points"
```

Exemple d'expression plus complexe, avec plus de 2 valeurs :

```
(score + points) * 10
```

Les parenthèses permettent de découper l'expression pour changer la priorité du calcul. Par défaut les multiplications sont calculées en 1er. Ici `score + points` sera calculé en 1er par l'ordinateur, puis le résultat sera multiplé par 10.

Mise en pratique

```
vies = 5
print(vies)
vies = vies - 1
print(vies)
print(vies + 1)
print(vies)
titrevies = "Nombre de vies: "
print(score+vies)
```

Stocker des valeurs complexes

En Lua on peut facilement stocker des valeurs complexes qui représentent des entités et non de simples valeurs.

Par exemple un personnage, ou un ennemi, est une entité dans votre jeu. On veut pourtant pouvoir le manipuler comme une valeur. C'est en fait un ensemble de valeurs.

On utilise pour cela une "table".

Une table c'est une variable qui peut contenir plusieurs variables.

On crée une variable en lui donnant un nom et en la connectant à une boite vide

- `heros = {}`

On peut ensuite lui connecter d'autres variables, en les attachant avec un point :

- `heros.vies = 5`
- `heros.energie = 100`

On peut aussi créer une table en lui donnant directement son contenu :

- heros = { vies = 5, energie = 100 }

C'est l'équivalent de :

- `heros.vies = 5`
- `heros.energie = 100`

2 - Apprenez à créer des fonctions, les appeler, et leur ajouter des paramètres

C'est quoi une fonction

Une fonction c'est un ensemble de lignes de code qui a une "fonction" comme son nom l'indique. Par exemple de réaliser un calcul, ou une série d'actions.

Cela vous évite d'avoir du code en double ou triple dans votre programme et cela est indispensable pour le structurer.

Une fonction a un nom et peut, ou pas, retourner un résultat (une valeur).

Par exemple, quand votre héros meurt et que le joueur perd la partie, on doit faire une série d'actions pour revenir au menu du jeu (remettre le score à zéro, etc.). Or, on peut mourir de différentes façon dans un jeu, et à chaque fois on devra faire ces actions. On crée alors une fonction "Gameover" (c'est vous qui choisissez le nom des fonctions) qui pourra faire toutes ces actions lorsqu'on en aura besoin.

Autre exemple, plus simple. Lorsqu'on réalise un calcul comme par exemple calculer la distance entre 2 points. On crée une fonction CalculeDistance qui reçoit les valeurs des 2 points, et qui retourne le résultat. On peut l'appeler comme on veut, mais son nom devra vous paraître clair, même plusieurs semaines plus tard. Évitez les noms en abrégé.

Comment créer une fonction

On dit qu'on "déclare" une fonction.

Une fonction commence par le mot clé `function` et termine par le mot clé `end`.

```
function nom(paramètres séparé par des virgules)
      corps de la fonction
end
```

L'ordinateur n'exécute pas une fonction lorsqu'il en trouve la déclaration dans votre code. Il l'exécutera lorsque vous "appellerez" la fonction par son nom plus tard dans votre code.

Une fonction a un nom. Les règles sont les mêmes que pour une variable. Exemple : `function Addition`

Une fonction reçoit (ou pas) des paramètres. Ce sont ses valeurs d'entrées. C'est vous qui décidez si votre fonction a besoin de paramètres ou pas, en fonction de son rôle dans votre jeu. On donne la liste des paramètres entre les parenthèses, après le nom de la fonction.

```
function Addition(valeur1,valeur2)
```

Ici, la fonction `Addition` reçoit 2 paramètres sont les noms seront `valeur1` et `valeur2`. (On peut ensuite utiliser ces variables dans le corps de la fonction)

On se servira de la fonction ainsi, par exemple pour additionner 1 et 1 :

```
Resultat = Addition(1,1)
```

Une fonction retourne (ou pas) une valeur. On utilise le mot clé `return` (retourne) suivi de la valeur à renvoyer.

```
function Addition(valeur1,valeur2)
      return valeur1 + valeur2
end
```

Cette fonction reçoit donc deux valeurs, dans deux variables (`valeur1` et `valeur2`) ensuite elle retourne l'addition des deux.

Une fonction qui retourne une valeur pourra s'utiliser comme une variable dans une expression.

```
Resultat = 5 + Addition(10,2)
```

Dans ce cas, `Resultat` contiendra 17 (5 + 12), 12 étant le résultat de la fonction `Addition` qui additionnera 10 et 2.

Mise en pratique

Nous allons ici utiliser votre savoir concernant les variables, et créer une fonction qui applique un dégât au héros et modifie son état pour prendre en compte qu'il est blessé.

Testez ce code :

```
heros = {}
heros.energie = 100
heros.blesse = false
function dommage(degats)
     heros.energie = heros.energie - degats
heros.blesse = true
end
print("Energie avant : "..heros.energie)
print("Blessure avant : "..tostring(heros.blesse))
dommage(10)
print("Energie après : "..heros.energie)
print("Blessure après : "..tostring(heros.blesse))
```

Note : La fonction `tostring` permet de convertir un booléen en chaîne de caractère. On pourrait la traduite en français par "VersChaine".

Et voici un exercice :

Créez une autre fonction, à la suite de ce code, qui aura pour rôle de restaurer l'énergie du héros et d'enlever son état de blessé. Testez les 2 fonctions.

Portée des variables

En Lua, si vous ajoutez le mot clé "local" devant la déclaration d'une variable, elle n'est visible que dans le "bloc" dans laquelle elle a été déclarée :

- Si elle a été déclarée dans un "if ... then", elle n'est visible/accessible que dans le code contenu dans le "if", elle n'existe plus une fois le "end" du if atteint
- Si elle a été déclarée dans une fonction, elle n'est visible/accessible que dans cette fonction, elle n'existe plus une fois le "end" de la fonction atteint
- Si elle a été déclarée hors d'un bloc, donc par exemple au début d'un fichier .lua, elle n'est visible/accessible que dans ce fichier

PAR CONTRE :

- Si le mot clé "local" n'est pas précisé, la variable devient globale !

C'est à dire que tout le monde peut la voir ! De partout !!

Il faut au maximum éviter de créer des variables globales, car cela n'est pas super propre. Cela crée des dépendances entre fichiers et elle sont difficiles à voir. En effet quand on relit notre code, on voit des variables sans savoir d'où elles viennent. Elles peuvent en plus être modifiée par tout le monde et ça génère des bugs difficiles à trouver : "mince, ma variable est à 0, qui l'a modifiée ??".

A réserver à des constantes, qu'on est sûr de vouloir partager par exemple.

Astuce : nommez vos variables globales de manière à ce que leur nom soit explicite, et qu'on comprenne qu'elles sont globales :

- En les commençant par "g" ou "g_" : gScore ou g_Score
- En les écrivant en majuscule si vous souhaitez exprimer qu'elles ont été initialisées une seule fois et qu'il ne faut pas les modifier : TILE_WITDH

Quelques mots de plus sur la portée des variables

Lorsqu'on crée beaucoup de variables, on encombre la mémoire de l'ordinateur. Ce n'est pas un véritable problème de nos jours car nos ordinateurs ont une mémoire phénoménale. Par contre, un problème demeure, celui des variables qui portent des noms identiques.

L'erreur courante est déjà de ne pas donner de nom clairs à ses variables. C'est une erreur classique des débutants. On se retrouve alors avec des variables nommées `x` ou `y` ou `val`, et autre noms farfelus et illisibles qu'on ne reconnaîtra pas le lendemain en reprenant son code.

Et l'autre erreur est d'utiliser plusieurs fois des variables de même noms, ce qui va provoquer des bugs.

Si on combine les 2 erreurs, on aura des soucis très vite...

Exemple de bug :

```
maVariable = 10
function maFonction(valeur1,valeur2)
      maVariable = valeur1 + valeur2
      return maVariable
end
print(maVariable)
maFonction(5,2)
print(maVariable)
```

Le résultat affiché sera :
10
7

Dans ce code, la fonction `maFonction` utilise une variable nommée `maVariable` pour stocker son résultat avant de le renvoyer. Mais comme il existe déjà une variable `maVariable` créée précédemment. Sa valeur sera modifiée à chaque fois que la fonction sera exécutée, ce qui n'est sûrement pas ce que veut le programmeur !

L'utilisation du mot clé `local` va régler le problème. Voici la solution :

```
maVariable = 10
function maFonction(valeur1,valeur2)
      local maVariable = valeur1 + valeur2
      return maVariable
end
print(maVariable)
maFonction(5,2)
print(maVariable)
```

Le résultat affiché est maintenant :
10
10

Que fait ce mot clé `local` ?

Il va indiquer à Lua que la variable `maVariable` créée dans la fonction `maFonction` ne doit exister que jusqu'à la fin de la fonction. Ainsi, la variable maVariable créée dans la fonction n'est pas la même que celle créée au début du programme, elle n'existera plus une fois l'appel de la fonction terminé..

La fonction est ainsi qualifiée de "bloc" de code. Un bloc de code se termine par le mot clé `end` (ou par la fin du fichier contenant le code si la variable est déclarée en dehors de toute fonction ou bloc de code). Nos verrons dans le 3ème fondamental d'autres types de blocs de code.

3 - Apprenez les structures de contrôle (boucles, conditions, etc.)

C'est quoi les structures de contrôle ?

Il s'agit de la véritable intelligence de votre code.

C'est ce qui lui permet de ne pas s'exécuter de manière totalement linéaire.

C'est un concept fondamental que vous devez maîtriser. C'est assez facile à comprendre, ce sont surtout les possibilités offertes par ces structures qui sont complexes car elles sont infinies.

Il existe 2 principales structures de contrôle : les conditions et les boucles.

Les conditions permettent à votre programme de réagir différemment en fonction de ce qui se passe dans vos variables ou dans votre programme en général. Les boucles servent à répéter un code plusieurs fois.

Les conditions

C'est la structure de contrôle la plus courante :
`if… then… else`

que vous pouvez traduire par :
`si… alors… sinon`

Les conditions permettent d'éxécuter du code seulement si une certaine condition est vérifiée.

Si il pleut alors prend un parapluie, sinon n'en prend pas.

Voici une condition qui s'appliquerait à un jeu vidéo :

```
if CalculeDistance(heros, enemi) < 100 then
     print "Alerte"
end
```

Ce qui signifiera : Si la distance entre le héros et l'ennemi est inférieure à 100, alors affiche le mot "Alerte".

Les opérateurs de comparaisons

Une condition compare ou teste des valeurs et ne se vérifie que si le résultat est "vrai".

Exemples :
`10 > 5` est une condition **vraie**.
`5 > 10` est une condition **fausse**.
`true` est une condition **vraie**. Rappelez vous, *true* veut dire *vrai*.
`false` est une condition **fausse**. Rappelez vous, *false* veut dire *faux*.

Du coup, on peut utiliser des variables booléennes dans nos conditions.

On peut ainsi reprendre notre exemple du héros (voir la leçon sur les variables) :

```
if heros.blesse == true then
     print("Le héros est blessé !")
end
```

Important :
On utilise un **==** (double égal) dans une condition, afin de ne pas confondre avec l'affectation de valeur. Dans certains langages comme le C, écrire :
`if (heros.blesse = true) ...`
Aurait pour effet de modifier la valeur de `heros.blesse` pour y inscrire `true` !
En Lua, si vous oubliez le double égal, vous aurez l'erreur suivante :
`'then' expected near '='`
Ce qui est peu compréhensible… mais une fois qu'on le sait c'est bon :-) !

Pour comparer des valeurs on utilise donc des opérateurs, voici leur liste :

==	Est égal
~=	N'est pas égal (est différent)
>	Est supérieur
<	Est inférieur
>=	Est supérieur ou égal
<=	Est inférieur ou égal

Attention :
Dans un if, le langage Lua va considérer la condition vérifiée si l'expression vaut "vrai" (exemple `energie > 10` si la valeur de `energie` est 11 ou plus), mais aussi si l'expression retourne une valeur non nulle (`nil` en Lua). On peut ainsi écrire des bugs sans le vouloir.

Exemple :

```
if energie then
     ...
```

```
end
```

Dans ce code, nous avons oublié de faire une comparaison. Mais si la variable `energie` existe alors la condition sera vraie et le corps de la condition va s'exécuter, ce qui n'est pas obligatoirement ce que nous avions décidé !

On peut utiliser ce principe à notre avantage, pour vérifier si une variable existe:

```
function test(valeur)
  if valeur then
    print("j'ai bien reçu une valeur")
  else
    print("je n'ai pas reçu de valeur")
  end
end
test() -- Cet appel affichera "je n'ai pas reçu de valeur"
```

Les conditions multiples, imbriquées ou complexes

Voici un exemple avec le mot clé `else` qui veut dire "sinon" et qui permet de faire de multiples conditions à la suite jusqu'à qu'une d'elle soit vérifiée.

```
if CalculeDistance(heros, ennemi) < 100 then
     print("Alerte")
else
     print("OK tout va bien")
end
```

On peut enchainer plusieur "if" au même niveau grâce au mot clé `elseif`.

```
if energie < 10 then
     print("Alerte rouge")
elseif energie < 50 then
     print("Alerte orange")
else
     -- Comportement par défaut
     print("Pas d'alerte")
end
```

On peut aussi imbriquer des conditions :

```
if energie < 10 then
     if alerte == false then
          print("Alerte rouge")
          alerte = true
     end
end
```

Pour des conditions plus complexes, nous utiliserons les mots clés `or` et `and`.

Exemples :

```
if energie < 10 and alerte == false then
     print("Alerte rouge")
     alerte = "rouge"
end
if alerte == "rouge" or alerte == "orange" then
     print("Alerte rouge ou orange !")
end
if alerte ~= "rouge" then
     print("Pas d'alerte rouge en cours")
end
```

Mise en pratique

Expérimentez ces codes avec différentes valeurs pour la variable `energie`.

Astuce : relisez mentalement vos conditions, surtout quand elles sont imbriquées, et exécutez les mentalement en imaginant différents cas de figures (notamment la valeur des variables testées). Cela vous évitera souvent des bugs car vous vous rendrez facilement compte des éventuels oublis ou erreurs dans vos conditions.

Les boucles

Les boucle "For"

La boucle "For" est utilisée pour répéter un bloc de code (une ou plusieurs lignes de code) un nombre précis de fois. Et elle permet d'utiliser la valeur de ce nombre dans son code.

La boucle For incrémente (augmente) la valeur d'une variable à chaque étape, en partant d'une valeur donnée, jusqu'à une autre valeur. Exemple : de 1 à 10.
Elle peut aussi diminuer la valeur, exemple de 10 à 1.

Voici un exemple simple :

```
for compteur=1,10 do
   print(compteur)
end
```

Ce code va afficher : 1…2…3…4… jusqu'à 10. Le 1er paramètre (1) est la valeur de début de la variable `compteur`, et le seconde (10) est la valeur de fin. Par défaut la variable va augmenter de 1 en 1.

C'est l'équivalent de :

```
compteur = 1
print(compteur)
compteur = compteur + 1
print(compteur)
compteur = compteur + 1
print(compteur)
… 10 fois
```

Si on veut afficher de 10 à 0 on fera :

```
for compteur=10,0,-1 do
   print(compteur)
end
```

Les boucles "While"

Une boucle "While" va répéter un bloc de code jusqu'à qu'une condition ne soit plus vraie. C'est assez simple à utiliser et à lire. Voici un exemple :

```
while NombreEnnemis < 10
      AjouteEnnemi()
      NombreEnnemis = NombreEnnemis + 1
end
```

Dans cet exemple, la fonction AjouteEnnemi sera appellée 10 fois.

Important : Les boucles While sont dangereuses. Elle peuvent boucler à l'infini ! Dans l'exemple que je viens de donner, si on avait oubli d'augmenter la valeur de NombreEnnemis, la condition serait indéfiniment vrai et le programme se bloquerait.

S'échapper d'une boucle !

Si pour une raison ou pour une autre vous souhaitiez interrompre une boucle pour continuer le programme, utilisez le mot clé "break" (qui veut dire grossomodo "casser" en français).

La boucle s'interrompt alors et le code reprend à la fin du bloc (délimité par le mot clé `end`).

Voici un exemple fictif pour illustrer l'utilisation du mot clé `break`.

```
-- Sort de la boucle quand nous avons atteint la ligne du héros
for compteur=1,10 do
      if heros.ligne == compteur then
            break
      end
end
```

Mise en pratique et exercices

1) Voici un exercice qui va combiner boucles et conditions.

Exécutez une boucle 100 fois, et affichez une trace uniquement quand le compteur est arrivé à mi course.

Voir la vidéo pour la solution.

2) Voici un exercice de boucle imbriquée

Exécutez des boucles de manière à parcourir 40 colonnes et 10 lignes d'une grille imaginaire.

Voir la vidéo pour la solution.

4 - Apprenez à créer des tableaux et des listes (indispensables pour les niveaux, les objets à l'écran, les ennemis, les tirs, etc.)

Nous avons déjà vu comment stocker des valeurs complexes. Ce sont déjà, à ce stade, des tableaux pour Lua:

```
hero.x
```
est équivalent à
```
hero["x"]
```

Un tableau permet de stocker une collection de valeurs en les associant à une clé. Pour accéder à une valeur, la clé sera ensuite placée entre deux crochets, à la suite du nom du tableau : **monTableau[maClé]**

On peut donc créer un tableau simplement ainsi :

```
monTableau = {}
monTableau["ma clé"] = "ma valeur"
print(monTableau["ma clé"])
```

Les tableaux et les listes, en Lua, se confondent, et sont stockées de la même manière dans la mémoire. On peut quand même faire une différence entre les listes et les tableaux, notamment sur la clé utilisée qui est numérique pour une liste.

En Lua, on pourrait dire que les listes sont des tableaux ordonnés par un index numérique :

```
maListe[1]
maListe[2]
maListe[3]
```

Pour créer une liste on peut faire ainsi :

```
maListe = {}
maListe[1] = "Valeur 1"
maListe[2] = "Valeur 2"
```

Mais aussi via une fonction standard de Lua, **table.insert**. Voici comment :

```
table.insert(maListe, "Valeur libre")
```

L'index sera automatiquement calculé par Lua, et la valeur sera placée à la suite des précédentes.

Pour obtenir le nombre d'éléments d'une liste, il faut précéder le nom de la liste par un dièse (#). Exemple :

print(#maListe)

Affichera 3 si il y a 3 éléments dans la liste.

Pour parcourir une liste il y a 3 méthodes :

Méthode avec un **for** simple

```
for n=1,#maListe do
     print(maListe[n])
end
```

Méthode avec **ipairs**

```
for n,v in ipairs(maListe) do
     print(n,v)
end
```

n contiendra l'index, et **v** la valeur, pour chaque élément de la liste.

Cela ne parcourra que les éléments avec un index numérique et qui se suivent.

Méthode avec **pairs**

```
for c,v in pairs(maListe) do
     print(c,v)
end
```

c contiendra l'index, et **v** la valeur, pour chaque élément de la liste. Mais c pourra être un index non numérique, la liste n'a pas besoin d'être ordonnée (index qui se suivent), et on peut avoir une combinaison des deux. Exemple :

```
maListe = {}
maListe["Titre"] = "Couleurs"
maListe[1] = "rouge"
maListe[2] = "vert"

for c,v in pairs(maListe) do
     print(c,v)
end
```

Affichera à la fois les couleurs, avec leurs clés numériques (1 et 2) mais aussi le titre, stocké lui avec une clé texte : "Titre".

5 - Apprenez les rudiments de la programmation Orientée Objet (POO)

Les sources complète d'un exemple de 4 méthodes pour avoir un code modulaire est disponible ici :

https://github.com/dmekersa/Gamecodeur/tree/master/Module

Voir la vidéo pour plus de détails : http://gamecodeur.fr/module-1-lua-love2d

Vous savez programmer… Et maintenant ?

La suite ?

Vous pouvez à ce stade expérimenter par vous même la programmation de jeux vidéo en appliquant vos connaissances avec un moteur de jeu (appelé également Framework) tel que Love2D, Corona SDK, ou encore Gideros. Voir plus loin.

Suivez mon second cycle de formation pour apprendre très rapidement les 5 fondamentaux de la programmation d'un jeu vidéo et créer votre tout premier jeu vidéo très facilement.

Comment le langage Lua va vous permettre de créer des jeux professionnels

Lua est aujourd'hui un langage utilisé par de nombreux moteurs de jeux vidéo, et permet de créer des jeux professionnels qui pourront ensuite être diffusé sur internet, sur mobile et même sur Steam.

Voici un aperçu des moteurs de jeux Lua que vous allez pouvoir utiliser.

Love2D

http://love2d.org
Idéal pour commencer. Ce framework très léger permet de créer un premier jeu très rapidement et très facilement (voir mes autres modules de formation).

Pico-8

http://www.lexaloffle.com/pico-8.php
Le jouet des programmeurs Lua ! C'est une console imaginaire qu'on exécute sur son PC, son Mac ou sous Linux et qui inclue un éditeur de code, un éditeur de sprites, un éditeur de map et un éditeur de son/musiques, le dans un écran en 128x128 pixels. Les jeux sont ensuite diffusables en HTML5 (fonctionnent dans un navigateur) et s'échange à l'aide d'un simple fichier image. Un must !

Corona SDK

http://coronalabs.com
Ce framework, originellement prévu pour créer des jeux mobiles, est maintenant capable de créer des jeux pour PC et Mac (HTML5 étant prévu pour bientôt). Il est très puissant, très complet et assez simple à apprendre. De plus, pour créer des jeux mobiles il est idéal car il inclue en standard tout ce qu'il faut pour gérer les spécificités mobiles : Achats intégrés, tailles d'écrans multiples, réseaux sociaux, accéléromètre, GPS, etc.

Gideros

http://giderosmobile.com/

Créé par un français (Nicolas Bouquet), ce framework gratuit supporte le mobile (dont Windows Phone), le PC, le Mac et bientôt HTML5. Très prometteur et bénéficiant d'une communauté active notamment à travers le financement participatif.

Cocos2d-x

http://cocos2d-x.org
Cocos 2D, c'est un framework très connu. Notamment parce qu'il a été utilisé pour créer quelques rouleaux compresseurs du jeu vidéo mobile : Clash of Kings, Big Fish Casino, Heroes Charge, Badland, etc. On peut le programmer traditionnellement en C++, mais aussi en Lua ! C'est tout de même un outil assez complexe à prendre en main.

Defold

http://www.defold.com/
Vous connaissez le studio King ? Oui… ceux qui ont créé Candy Crush. Et bien ils distribuent depuis peu (et gratuitement) leur moteur de jeu vidéo qui se programme en… Lua !

Cryengine (3D !)

http://cryengine.com/
Et oui, vous pourrez même créer des jeux en 3D à l'aide de Lua et Cryengine. A réserver aux programmeurs avancés car un passage par le C++ s'avérera nécessaire pour aller plus loin.

Comment apprendre un autre langage de programmation avec la même méthode

Les 5 fondamentaux que vous maîtrisez maintenant vont vous ouvrir les portes de tous les autres langages de programmation.

Exemple avec le C++. Vous constatez que le principe est le même.

En Lua :

```
valeur = 3
titre = "Gamecodeur"

function Addition(a,b)
	local r
	r=a+b
	return r
end

resultat = Addition(10,5)

if resultat == 15
	print("Le résultat est 15 !")
end
```

En C++ :

```
int valeur = 3;
string titre = "Gamecodeur";

int Addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int resultat = Addition(10,5);

if (resultat == 15)
{
	cout << "Le résultat est 15 !";
}
```

Il faut juste comprendre que chaque langage à sa syntaxe et ses spécificités. Dans cet exemple vous constatez l'apparition d'un nouveau mots clés en C++ (`int`) qui permet de spécifier le type de la variable (ici un entier) et d'une ponctuation différente (les accolades et les points virgule). De même, afficher une trace ne se fait pas de la même manière.

Lua reste le plus simple n'est-ce pas ?

Les autres différences viendront essentiellement de la manière de gérer les tableaux et les listes, ainsi que dans la programmation objet.

Et bien entendu, les outils et les moteurs de jeu changent. La complexité va venir essentiellement de ces outils.

Par exemple en C++, qui est un langage qu'on va "compiler", la notion de compilation est complexe et met en oeuvre de nombreux paramètres, et avec eux de nouvelles sources d'erreurs, de termes techniques et autres joies.

Commencer par Lua et Love2D vous donnera la confiance nécessaire pour aborder ces nouveaux continents. Tout se fera en douceur et vous serez étonnés de vos capacités. Commencez directement par le C++ ou par Unity3D et préparez-vous à un découragement quasi inévitable !

Dans quel ordre apprendre les langages de programmation ?

Voici mon conseil :

1) Apprenez le Lua pour savoir programmer sans être gêné par une complexité inutile
2) Créez des petits jeux avec Love2D, Corona SDK, Pico-8…
3) Apprenez un langage objet, par exemple Haxe avec son Framework Haxeflixel, ou Java avec LibGDX
4) Si vous visez une carrière pro : Apprenez le C et le C++ avec SFML
5) Apprenez le C# avec une mise en oeuvre sur Unity3D et/ou Monogame

On peut aussi passer directement de l'étape 2 à l'étape 4 si on est un programmeur avancé (mais je conseille tout de même de connaître d'autres langages avant le C++ et le C#).

Vous aurez alors un profil de pro et pourrez créer le jeu de vos rêves et/ou trouver un job !

Restez aussi en contact avec Gamecodeur car chacun de ces autres langages vous sera enseigné avec la même méthode, et avec un ou plusieurs exemples de jeux en prime.

Si vous ne l'avez pas encore fait :

Abonnez-vous à ma newsletter pour recevoir mon guide "Comment devenir programmeur de jeux vidéo", mais aussi à ma chaîne Youtube.

Pour cela rendez-vous sur :
http://www.gamecodeur.fr