LE KIT DE DÉMARRAGE DU PROGRAMMEUR DE JEUX VIDÉO

Avec Lua et Love2D

David Mekersa

Version 1.0a

ISBN : 978-2-9589424-4-1

Distribué par :
David Mekersa
128 rue de la Boétie
75008 Paris Cedex 8

SOMMAIRE DU GUIDE

A propos de ce guide

Depuis plus d'une décennie, je partage ma passion pour la programmation de jeux vidéo, principalement via ma plateforme *gamecodeur.fr*. Mes formations et ma pédagogie ont aidé des dizaines de milliers d'apprentis programmeurs à se lancer.

Avant d'enseigner la programmation, je l'ai pratiquée plus de 30 ans dans de nombreux domaines en France et à l'international : programmation de progiciels, programmation d'AGL (WinDev), programmation C et C++ pour l'informatique embarquée (téléphonie mobile), et bien sûr pour le jeu vidéo avec plus de 25 productions de jeux, serious games et applications multimédia.

Au fil de ma carrière, j'ai recruté, formé et dirigé des équipes de programmeurs. J'ai acquis une compréhension profonde de ce qui fait un bon programmeur, mais aussi des pièges grossiers dans lesquels ils peuvent tomber.

Cette expérience m'a permis de développer une approche d'enseignement unique. J'ai pu démontrer que presque tout le monde pouvait apprendre à programmer à partir de zéro. Et qu'il fallait souvent défier les idées reçues, comme la survalorisation des tutoriels qui, selon moi, contribue largement à l'échec de toute une génération, persuadée qu'il y a "un tuto" pour tout.

Ma philosophie : comme un artisan, il faut construire des bases solides et universelles. Il faut apprendre à raisonner comme un programmeur. Et arrêter de faire le singe savant en recopiant des tutos sans rien y comprendre.

Ce guide est une version révisée et enrichie de ma méthode dont la 1ère version date de 2016. Elle intègre désormais un module consacré à la création d'un premier jeu vidéo : un space shooter.

En plus d'apprendre de solides bases de programmation, vous pourrez donc passer immédiatement en pratique en créant un premier jeu vidéo.

Si vous suivez ma méthode, vous avez la quasi certitude d'apprendre à programmer.

Voici ce que feront ceux qui n'y parviendront pas :

- Il vont survoler les concepts, en les lisant seulement.
- Cette lecture sera bien sûr rapide,car "ils n'ont pas le temps".
- Il ne vont rien mettre en pratique, car "ils ont compris en lisant et ça suffit"
- A porté de main ils auront Discord, Messenger, Insta… voire Twitch avec des parties de LOL sur un second écran
- Une fois la formation terminée ils vont se vanter auprès de leurs amis : "j'apprend à coder !"
- N'ayant rien compris au final, ils accuseront la méthode et retourneront vers des "tutos".

La réalité est parfois difficile à entendre pour certains :

- On n'apprend pas en lisant. Coder ce n'est pas du cinéma. Il faut pratiquer, essayer, galérer, se planter, faire n'importe quoi mais TAPER UN MAX DE CODE tous les jours !
- Il faut du temps… beaucoup de temps... Le mythe du mec qui apprend en 2 jours, c'est sur Tik Tok ou Instagram, pas dans la vraie vie.
- Il faut se couper des réseaux sociaux et réapprendre à se placer dans "le temps long".
- On n'apprend pas pour les autres. On n'apprend pas par orgueil. Le seul moteur c'est l'envie et la passion.

J'espère que vous allez prendre du plaisir à me suivre dans cette aventure. En tout cas, j'ai eu beaucoup de plaisir à créer ce contenu pour vous. Et même s'il ne change la vie que d'une personne, j'aurais réussi quelque chose d'incroyable.

Prêt ?

Éteignez votre téléphone, asseyez vous devant votre ordinateur, programmez un compte à rebours de 25 minutes pendant lesquelles personne ne doit vous déranger (même pas vous même).

On va commencer !

Comment on apprenait facilement en 1985

En 1985, l'apprentissage de la programmation était souvent une expérience simple et directe, grâce à des langages comme le BASIC. Sur des ordinateurs comme l'Amstrad CPC ou le Commodore 64, les débutants pouvaient rapidement comprendre les concepts fondamentaux de la programmation et créer des programmes.

Pourquoi le BASIC est si facile à apprendre ?

Le BASIC est simple : Le BASIC, avec ses mots-clés clairs et sa syntaxe concise, est accessible aux débutants. Le manuel d'utilisation suffit à comprendre son fonctionnement.

Le BASIC est lisible : Les lignes numérotées du BASIC permettent de visualiser facilement le déroulement du programme et de comprendre la logique de l'ordinateur.

Le BASIC est intégré : Le BASIC était intégré aux ordinateurs de l'époque, accessible dès la mise sous tension. Pas besoin d'installations complexes, juste le plaisir de taper du code et de voir le résultat instantanément.

Le BASIC n'est pas élitiste : A l'époque, programmer n'était pas élitiste. La simplicité était considérée comme une vertue, et non comme une tare comme parfois aujourd'hui.

Vous voulez suivre votre première leçon de programmation en BASIC ?

Voici un exemple simple de programme BASIC pour apprendre à saisir et afficher une information :

```
10 cls
20 input "Quel est votre âge ?", age
30 print "Vous avez ", age, " ans !"
```

Chaque ligne est numérotée et son rôle est clair :

`10 cls` :
Efface l'écran

`20 input "Quel est votre âge ?", age` :
Demande à l'utilisateur son âge et le stocke dans la variable `age`.

`30 print "Vous avez ", age, " ans !"` :
Affiche un message personnalisé avec l'âge saisi, en construisant une phrase.

Le BASIC aujourd'hui

Même si le BASIC n'est plus aussi populaire qu'autrefois, son influence se retrouve dans de nombreux langages modernes. L'apprentissage de ses concepts fondamentaux reste une base solide pour tout programmeur débutant.

Mon premier jeu commercial, "Geisha : Le jardin secret", a été codé en Blitzmax, un langage moderne s'inspirant largement du BASIC. La syntaxe familière de Blitzmax, (qui est un BASIC avec en plus la Programmation Orientée Objet) m'a permis de créer un jeu complet en exploitant les connaissances acquises avec le BASIC. Ce jeu m'a coûté 6000 € à produire (je ne compte pas le temps passé) et rapporté plus de 100000 euros à l'époque. Comme quoi, pas besoin de coder avec des langages élitistes pour gagner sa vie dans le jeu vidéo.

Pour aller plus loin

Découvrez le manuel d'un ordinateur des années 80, comme celui de l'Amstrad CPC :
https://archive.org/details/Amstrad_CPC464_Guide_de_lutilisateur_1984_AMSOFT_FR

Blitzmax est maintenant gratuit et toujours utilisé pour créer des jeux professionnels :
https://blitzmax.org/

C’est quoi du code ?

Imaginez le code comme une recette de cuisine spéciale que vous écrivez pour votre ordinateur.

Au lieu de préparer un plat, votre ordinateur "cuisinera" un programme en suivant les instructions que vous lui donnez. Ces instructions, écrites dans un langage que l'ordinateur peut comprendre, forment ce qu'on appelle le "code source" ou plus simplement "le code".

Le code se compose d'instructions, disposées ligne par ligne, que nous appelons "lignes de code".

Chaque ligne a un but spécifique, guidant l'ordinateur étape par étape, un peu comme les étapes d'une recette de cuisine.

Prenons un exemple simple pour illustrer :

```
Début de la recette

- Aller au marché
- Total = 0
- Chercher du beurre
- SI tu trouves du beurre ALORS
  - Prendre du beurre
  - Total = Total + Prix du beurre
- SINON
  - Prendre de la margarine
  - Total = Total + Prix de la margarine
- FIN DE SI
- Payer Total
- Revenir à la maison

Fin de la recette
```

Dans cet exemple, chaque action (comme "`Aller au marché`") est une instruction que l'ordinateur doit suivre. Les conditions (comme "`Si tu trouves du beurre alors`") permettent de prendre des décisions en fonction de la situation. On appelle cela une structure de contrôle. Cela permet à votre code de prendre des décisions.

Chaque langage de programmation a sa propre "syntaxe", c'est-à-dire un ensemble de règles définissant comment les instructions doivent être écrites pour être comprises par l'ordinateur. Cela inclut la manière d'organiser les mots, leur orthographe, et l'utilisation de symboles spéciaux, tout comme les règles de grammaire dans une langue humaine.

On ne triche pas avec la recette

Un aspect fondamental de la programmation à bien comprendre c'est que le code est exécuté de manière séquentielle, c'est-à-dire ligne par ligne, du haut vers le bas.

Imaginez que vous lisiez une recette de cuisine ou un itinéraire de voyage : vous commencez au début et avancez étape par étape, en suivant les instructions dans l'ordre où elles apparaissent.

Le faire dans le désordre provoquerait le chaos. De la même manière, l'ordinateur lit et exécute votre code instruction par instruction, en respectant l'ordre que vous avez défini.

Cependant, il y a des moments où votre programme peut "sauter" vers un autre endroit de votre programme ou répéter certaines sections avant de continuer son chemin. Ces "sauts" sont contrôlés par des structures spéciales dans le code, telles que les **fonctions**, les **conditions** et les **boucles**.

Par exemple, une condition peut dire à l'ordinateur de passer à une partie différente du code si une certaine condition est remplie, comme choisir entre prendre du beurre ou de la margarine dans notre exemple précédent.

Pensez à ces sauts comme à des détours ou des boucles dans votre itinéraire. Parfois, vous suivez le chemin tout droit, et d'autres fois, vous prenez un chemin différent en fonction de ce que vous rencontrez, mais vous avancez toujours de manière séquentielle, en prenant une décision à la fois, jusqu'à ce que vous rencontriez la fin de votre programme.

Cette manière séquentielle de lire et d'exécuter le code est essentielle pour comprendre comment construire des programmes et prévoir comment ils se comportent à l'exécution.

En gardant à l'esprit ce principe de séquentialité, vous pourrez mieux anticiper le comportement de votre programme ou bien comprendre, en cas de bug, pourquoi il ne se comporte pas comme c'était prévu.

Les mots clés : chasse gardée du langage

Les "mots-clés" sont des termes spéciaux réservés par le langage de programmation (comme `if`, `then`, `else`, présents dans de nombreux langages).

Ils ont une signification particulière pour l'ordinateur. Ils sont les ingrédients principaux de votre recette de programmation. Il y en a peu, et ils sont donc facile à apprendre.

Que Veut Dire "Réservé" ?

Lorsque nous disons qu'un mot est "réservé" dans un langage de programmation, cela signifie que ce mot a une signification spéciale et ne peut pas être utilisé à d'autres fins, comme nommer une variable ou une fonction. C'est un peu comme si certains mots avaient un statut VIP dans la langue de programmation : ils sont réservés pour des tâches importantes et ne peuvent pas être utilisés pour représenter autre chose.

Par exemple, le mot-clé `if` est utilisé pour introduire une condition. Si vous essayez d'utiliser `if` comme nom pour une variable (par exemple, pour stocker un nombre), l'ordinateur s'en plaindra parce qu'il s'attend à ce que if annonce une décision à prendre, pas à représenter une quantité ou une information.

L'importance de la précision

L'ordinateur est bête et méchant. Et donc la programmation c'est tout aussi bête et méchant. Il est crucial de suivre exactement la syntaxe d'un langage de programmation.

Les programmes nécessitent une précision absolue, car votre ordinateur n'interprètera pas les nuances ou les erreurs comme le ferait un humain. Une erreur de syntaxe, même mineure, peut empêcher votre programme de fonctionner correctement.

Appels de fonctions : Les outils de votre cuisine

En programmation, nous utilisons également ce qu'on appelle des "appels de fonctions".

Ces fonctions sont comme des outils ou des appareils dans votre cuisine de programmation : elles accomplissent des tâches spécifiques, comme afficher du texte à l'écran ou effectuer des calculs.

Vous pouvez utiliser des fonctions fournies par le langage de programmation (comme les fonctions de base dans Lua et Love2D) ou créer les vôtres pour personnaliser votre programme.

Dans notre exemple, les fonctions seraient : `Aller`, `Payer`, `Revenir`, etc.

Introduction aux blocs de code

Dans cet exemple, notez comment nous avons regroupé certaines instructions ensemble sous `ALORS` et sous `SINON`.

```
- SI tu trouves du beurre ALORS
  - Prendre du beurre
  - Total = Total + Prix du beurre
- SINON
  - Prendre de la margarine
  - Total = Total + Prix de la margarine
- FIN DE SI
```

Ces groupes d'instructions sont appelés "blocs" de code.

Un bloc commence par exemple après une instruction conditionnelle comme `SI` et se termine par une instruction comme `FIN`. Les blocs aident à organiser le code de manière que certaines actions soient exécutées de manière groupées et uniquement dans certaines conditions. C'est comme si vous suiviez différentes parties de la recette en réalisant une série de gestes regroupés, en fonction du déroulé de la recette.

Les variables : stocker des ingrédients

Maintenant, parlons du "`Total`". Dans notre recette, nous utilisons "`Total`" pour stocker combien d'argent nous dépensons. Vous pouvez imaginer que "`Total`" est comme une boîte. Chaque fois que vous le désirez, vous pouvez changer son contenu (ici le prix du beurre ou de la margarine).

En termes de programmation, "`Total`" est une "variable", un espace nommé dans la mémoire de l'ordinateur où vous pouvez stocker, mettre à jour et récupérer des données. Et comme ces données peuvent varier tout au long du programme, on parle de "variables".

L'expression "`Total = 0`" signifie que vous initialisez votre variable avec la valeur 0 au départ.

L'expression "`Total = Total + Prix du beurre`" signifie que vous prenez la valeur actuelle de "`Total`", ajoutez le prix du beurre, puis mettez à jour "`Total`" avec cette nouvelle valeur. C'est une façon de dire : "Mets à jour cette variable avec cette nouvelle valeur."

Comment penser comme un ordinateur

Lorsque vous écrivez du code, vous donnez des instructions à votre ordinateur. Mais comment l'ordinateur comprend-il ces instructions ? Dans ce chapitre, nous allons explorer comment l'ordinateur interprète votre code et comment vous pouvez apprendre à penser comme lui pour devenir un meilleur programmeur.

L'exécution séquentielle du code

L'ordinateur interprète votre code ligne par ligne, de haut en bas, dans un ordre séquentiel. C'est la base de la programmation et il est essentiel de comprendre ce concept. Imaginez votre code comme une recette : l'ordinateur suit chaque étape dans l'ordre pour obtenir le résultat final.

Les sauts dans le code

Parfois, le code peut obliger l'ordinateur à faire un "saut", c'est-à-dire à ne pas passer à la ligne suivante mais à aller directement à un autre endroit du code. C'est le cas des instructions conditionnelles ("si... alors..."), des boucles ("tant que... faire"), et des fonctions.

L'importance de penser comme l'ordinateur

Apprendre à penser comme l'ordinateur est crucial pour devenir un bon programmeur. Cela signifie être capable de visualiser "mentalement" l'exécution du code et d'anticiper les résultats.

Vous ferez moins d'erreurs. En comprenant le fonctionnement de l'ordinateur, vous pouvez identifier les erreurs potentielles dans votre code avant même de l'exécuter.

Vous déboguerez plus facilement. Si une erreur survient, vous serez mieux préparé pour la comprendre et la corriger.

Voici quelques exercices pour développer votre pensée informatique :

Lisez votre code à voix haute :
Cela vous aidera à visualiser l'exécution du code et à identifier les erreurs potentielles.

Exécutez votre code ligne par ligne dans votre tête :
Imaginez l'état des variables après chaque ligne, anticipez le résultat des conditions, etc.

Utilisez des outils de débogage :
Les outils de débogage vous permettent de suivre l'exécution du code ligne par ligne et d'examiner les valeurs des variables. Cela peut être une méthode pour vous projeter dans l'exécution du code.

Apprendre à penser comme l'ordinateur est une compétence essentielle pour tout programmeur.

En maîtrisant l'exécution séquentielle du code, les sauts et en développant votre capacité à visualiser l'exécution du code, vous deviendrez un programmeur plus efficace et plus précis.

Parfois, c'est même l'étincelle qui débloque tout !

Configurer son ordinateur pour programmer

Les outils du programmeur de jeu vidéo débutant

Comme pour un cuisinier, il faut posséder quelques outils pour se lancer dans la programmation. Et pas de panique, tout est gratuit, à part le matériel.

Un ordinateur

N'importe quel PC ou Mac récent suffit pour apprendre à programmer et créer un premier jeu simple en 2D. S'il a moins de 10 ans, ça peut aller. S'il a moins de 5 ans c'est mieux car il sera plus réactif. Avoir un ordinateur à 3000 euros ne fera pas de vous un meilleur programmeur.

Mais posséder un ordinateur ne suffit pas, encore faut-il savoir s'en servir.

Quelques conseils pour mieux maîtriser l'ordinateur avant de se lancer dans la programmation

1. Apprendre les bases du fonctionnement d'un ordinateur

- Qu'est-ce que le système d'exploitation ? (Windows, Mac OSX, Linux…)
- Quels sont les différents types de fichiers ? (Exécutables notamment)
- C'est quoi une extension de fichier ? (et comment les afficher dans l'explorateur)
- Comment fonctionne le clavier et la souris ? (C'est quoi un raccourci clavier par exemple)
- Comment installer et désinstaller des logiciels ?

2. Se familiariser avec les outils bureautiques

- Traitement de texte (Word, Google Docs, etc.)
- Tableur (Excel, Google Sheets, etc.)
- Présentation (PowerPoint, Google Slides, Etc.)
- Navigateur web (Chrome, Firefox, Edge)

3. Développer sa pensée logique

- Faire des jeux de logique et des puzzles
- Apprendre les bases du raisonnement mathématique
- Suivre des cours en ligne sur la logique informatique

4. Exécuter des tâches simples

- Créer des dossiers et des fichiers
- Copier, coller et supprimer des fichiers
- Installer et utiliser des logiciels
- Naviguer sur internet

5. Se familiariser avec le vocabulaire informatique

- Apprendre les termes courants liés à l'informatique
- Consulter des glossaires et des dictionnaires en ligne

6. Ne pas avoir peur de faire des erreurs

- L'apprentissage par l'erreur est une méthode efficace
- Il est important de persévérer et de ne pas se décourager

7. Demander de l'aide

- N'hésitez pas à demander de l'aide à vos amis, à votre famille ou sur des forums

En suivant ces conseils, vous serez mieux préparé pour vous lancer dans la programmation.

Rien de pire que de commencer à programmer sans savoir se servir de son ordinateur.

Un éditeur de code

En plus de votre ordinateur, vous aurez besoin d'un logiciel pour écrire et exécuter votre code.

Imaginez que vous souhaitiez écrire un livre. Vous pouvez utiliser un simple crayon et du papier, mais cela peut être fastidieux et compliqué pour corriger les erreurs. Un traitement de texte est plus adapté.

Un éditeur de code est comme un traitement de texte pour les programmeurs. Il vous permet d'écrire du code de manière facile et efficace.

Voici quelques fonctionnalités d'un éditeur de code :

- Coloration syntaxique : Les différents types de code sont mis en évidence avec différentes couleurs, ce qui facilite la lecture et la compréhension du code.
- Auto-complétion : L'éditeur peut vous suggérer des mots clés et des fonctions pendant que vous tapez, ce qui vous permet de gagner du temps et d'éviter les erreurs.
- Vérification des erreurs : L'éditeur peut détecter les erreurs de syntaxe dans votre code, ce qui vous permet de les corriger avant d'exécuter votre programme.
- Débogage : L'éditeur peut vous aider à identifier et à corriger les erreurs dans votre programme.

Voici quelques options gratuites et populaires :

- Visual Studio Code (Windows/Mac)
- Notepad++ (Windows)
- Sublime Text (Windows/Mac)

Un framework 2D

Imaginez que vous souhaitez construire une maison. Vous pouvez acheter tous les matériaux nécessaires (briques, bois, ciment, etc.) et commencer à construire à partir de zéro. Cela prendra beaucoup de temps et d'efforts, et vous risquez de faire des erreurs.

Un framework est comme un kit de construction préfabriqué. Il vous fournit certains éléments de base dont vous avez besoin pour construire votre maison, tels que les murs, les fenêtres et les portes. Vous n'avez plus qu'à ajouter vos propres éléments et à les personnaliser selon vos besoins.

En programmation, un framework est un ensemble d'outils et de bibliothèques qui vous permet de créer des applications plus rapidement et facilement. Il vous fournit les fonctionnalités de base dont vous avez besoin, telles que la gestion des fenêtres, l'affichage d'images et la gestion des interactions utilisateur.

Sans framework, le simple fait de vouloir ouvrir une nouvelle fenêtre sous Windows représenterait un travail considérable et très technique. Et vouloir afficher une image n'en parlons pas !

Exemple de framework : Love2D

Love2D est un framework de développement de jeux vidéo 2D gratuit et open-source. Il est idéal pour les débutants car il est simple à utiliser et ne nécessite aucune connaissance en programmation 3D.

Voici quelques avantages d'utiliser Love2D :

- Facile à apprendre : La documentation est claire et concise, et de nombreux tutoriels sont disponibles en ligne.
- Puissant et flexible : Love2D peut être utilisé pour créer des jeux simples et des jeux plus complexes.
- Communauté active : Il existe une communauté active de développeurs Love2D qui peuvent vous aider si vous rencontrez des problèmes.

Si vous souhaitez commencer à développer des jeux vidéo 2D, Love2D est un excellent choix. Il vous permettra de créer des jeux rapidement et facilement, sans avoir à vous soucier des détails techniques.

Voici quelques ressources pour apprendre à utiliser Love2D :

Site officiel de Love2D: https://love2d.org/
Tutoriels Love2D: https://love2d.org/wiki/Tutorials
Forum Love2D: https://love2d.org/forums/

Un langage de programmation

Le choix du langage de programmation est important, car il peut influencer votre expérience d'apprentissage et la complexité de vos premiers projets. Pour débuter, je vous recommande Lua.

Pourquoi Lua est-il le langage idéal pour commencer ?

- Il est facile à apprendre : Lua est un langage simple avec une syntaxe claire et concise. Il est proche du BASIC, ce qui le rend accessible aux débutants.
- Il est puissant et flexible : malgré sa simplicité, Lua est un langage puissant qui peut être utilisé pour créer des jeux complets et professionnels. Il est également utilisé dans de nombreux autres domaines, tels que l'intelligence artificielle et le développement web.
- Il est largement utilisé : Lua est un langage universel et libre de droits, utilisé par des millions de développeurs à travers le monde.
- Il est parfait pour le jeu vidéo : Lua est particulièrement bien adapté au développement de jeux vidéo grâce à sa légèreté et sa rapidité.
- Il est associé à des frameworks comme Love2D qui facilitent la création de jeux 2D.

Avantages de Lua pour les débutants :

- Très peu de notions à comprendre pour commencer.
- Syntaxe légère, peu de mots "magiques".
- Pas de "compilation", ce qui facilite l'apprentissage et le débogage.
- Permet de créer des jeux professionnels.
- Tremplin vers d'autres langages de programmation.

Lua est un excellent choix pour commencer à apprendre à programmer. Il est simple, puissant et largement utilisé. Si vous souhaitez vous lancer dans le développement de jeux vidéo, Lua est un langage idéal pour vous.

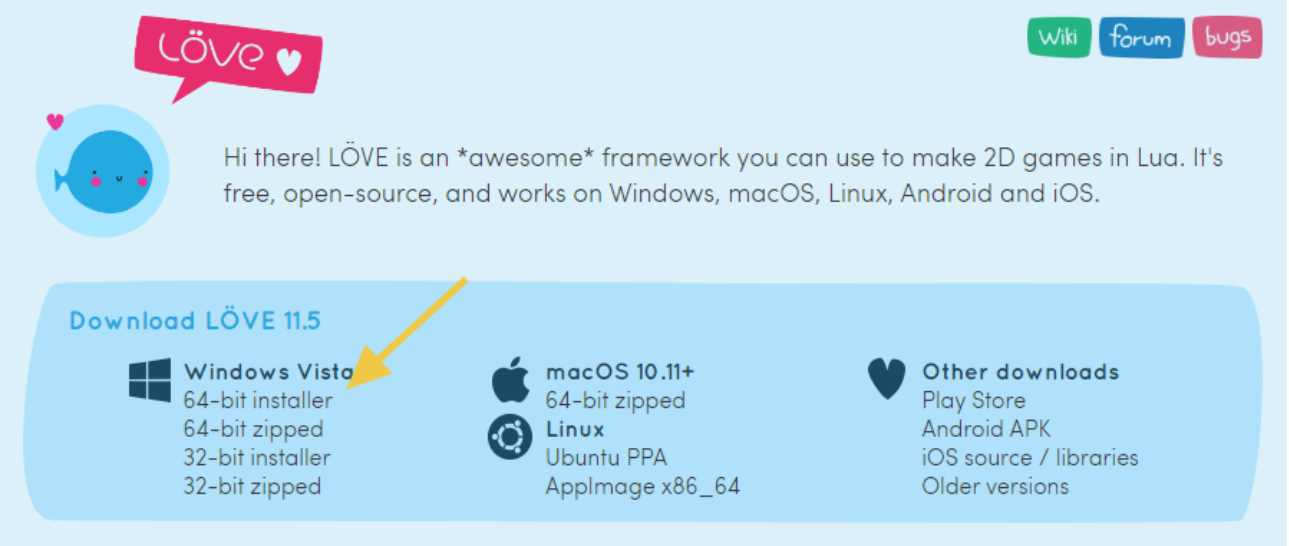
Le saviez-vous ?

Le célèbre jeu à succès BALATRO est conçu avec Love2D. Son auteur a annoncé avoir vendu 1 million de copies en mars 2024. La preuve que Lua et Love2D sont un excellent choix.

Installation de Visual Studio Code et de Love2D

Tout d'abord installez Love2D dans son répertoire d'installation par défaut.

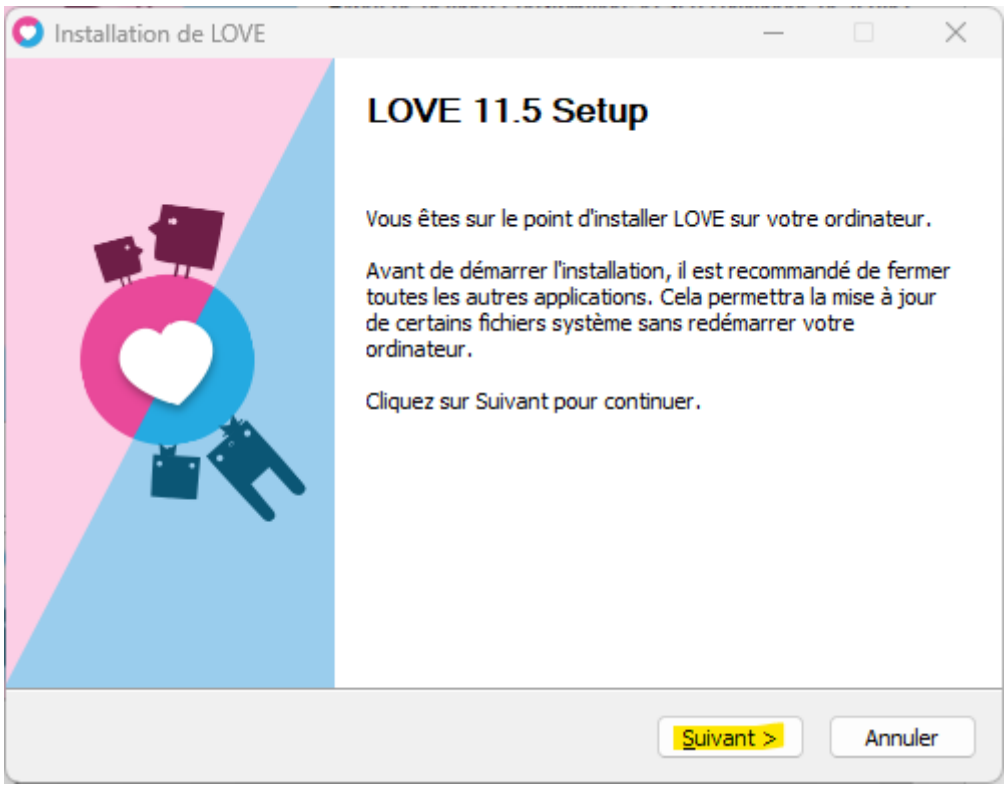
Téléchargez Love2D ici : https://love2d.org/



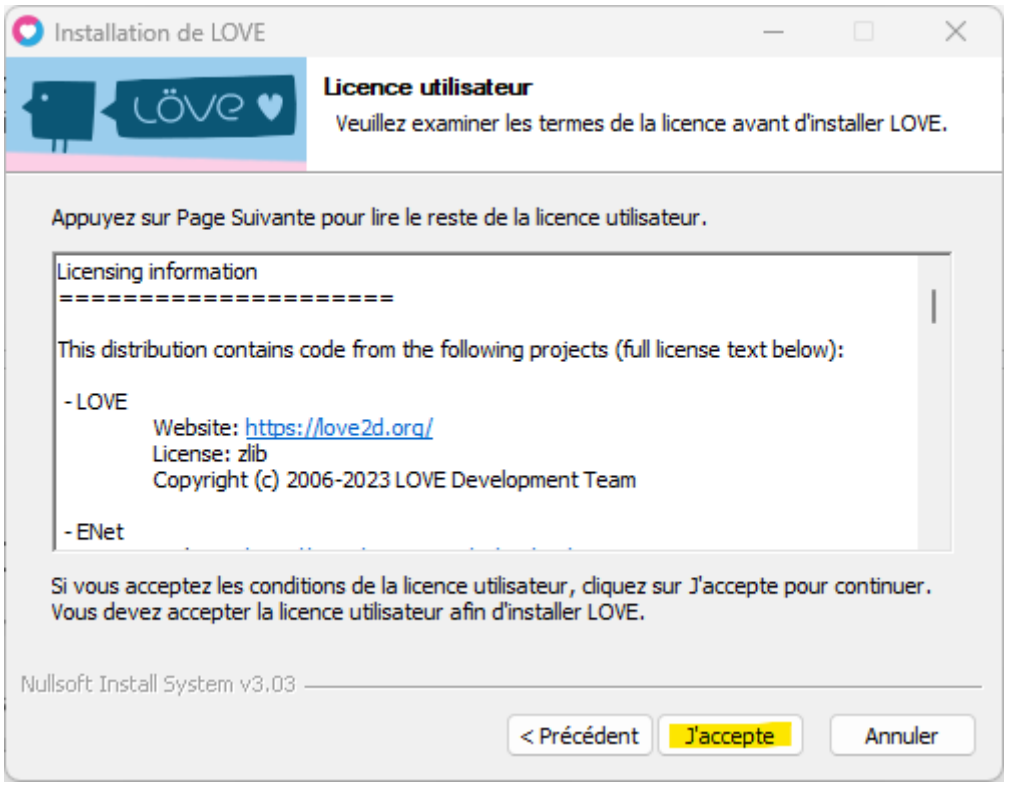
Choisissez la version Windows 64 bits avec installateur si vous êtes sous Windows.

Ensuite, exécutez le programme que vous venez de télécharger.

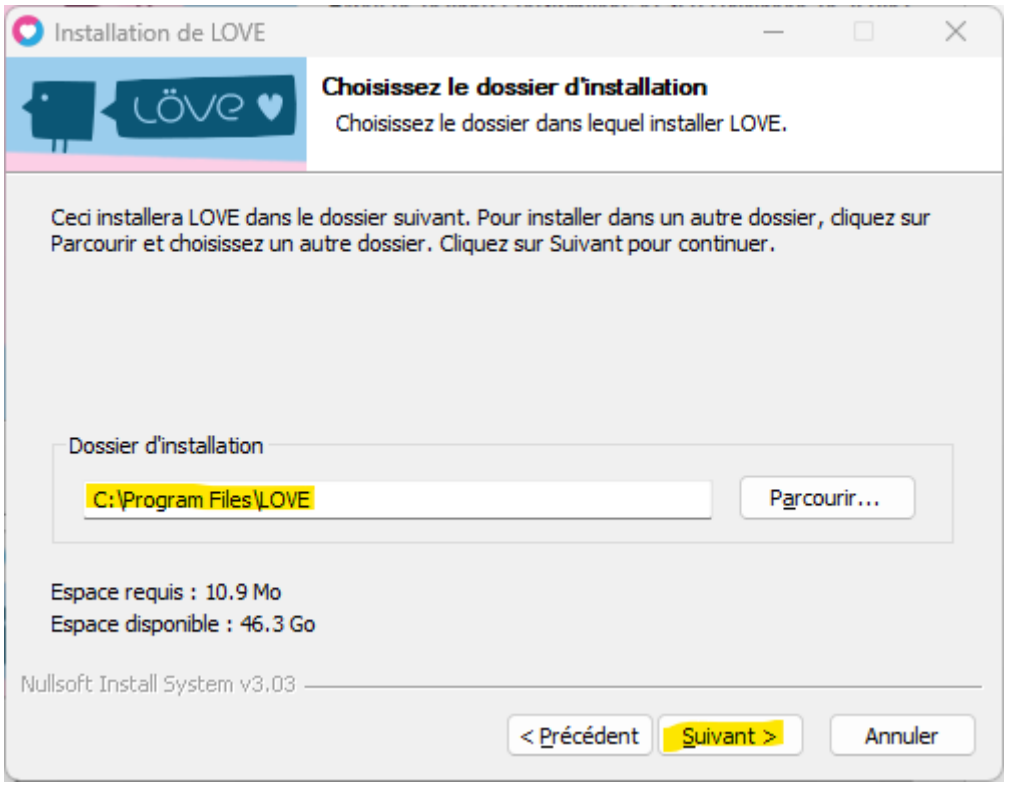
Voici la procédure que vous allez suivre ensuite :



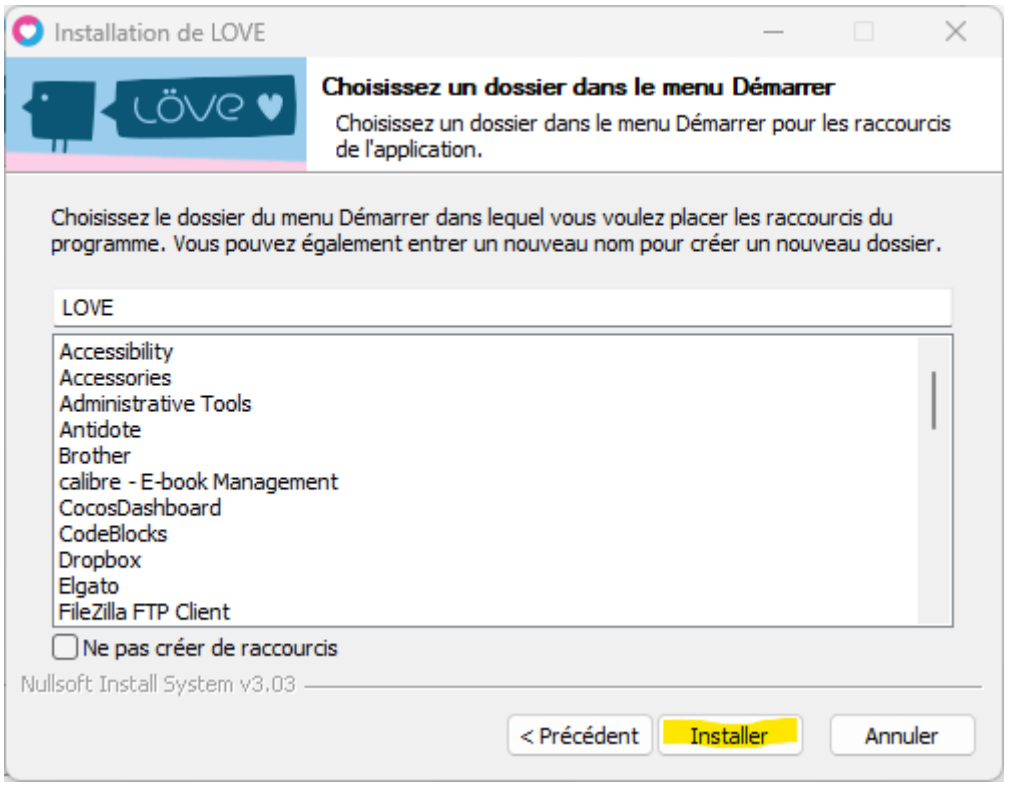
Cliquez sur "Suivant".



Acceptez la licence.



Cliquez sur "Suivant" sans modifier le dossier d'installation.

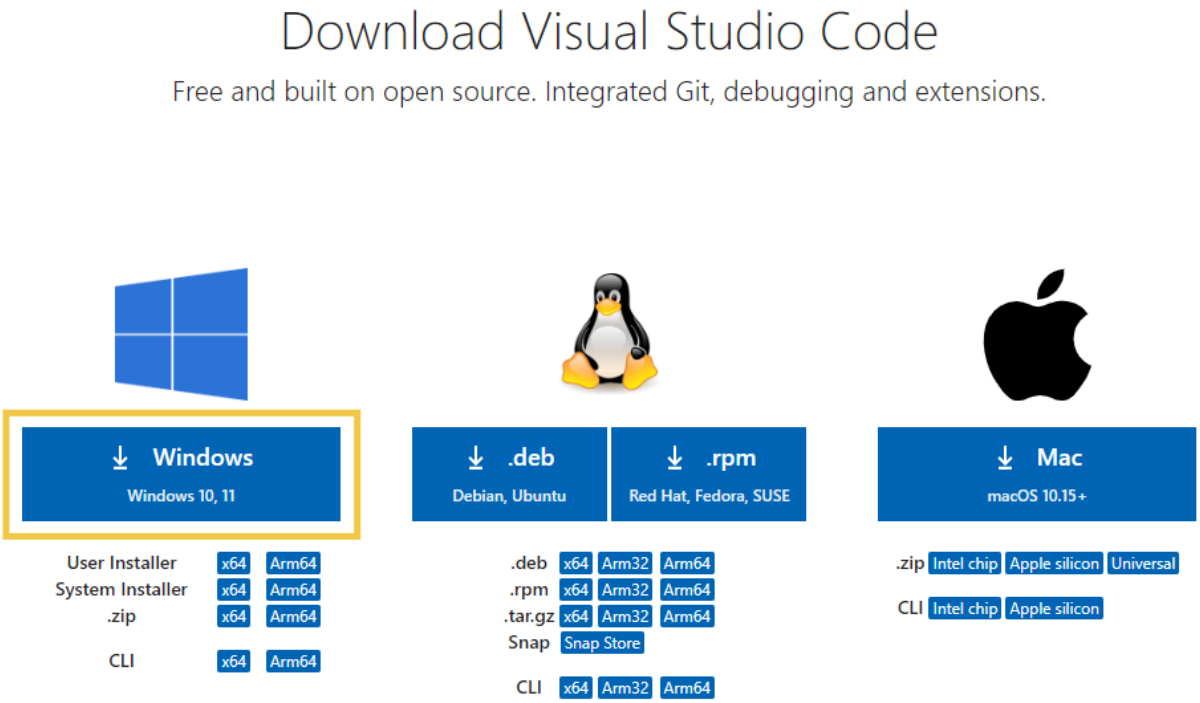


Ne touchez à rien et lancez l'installation en cliquant sur "Installer".

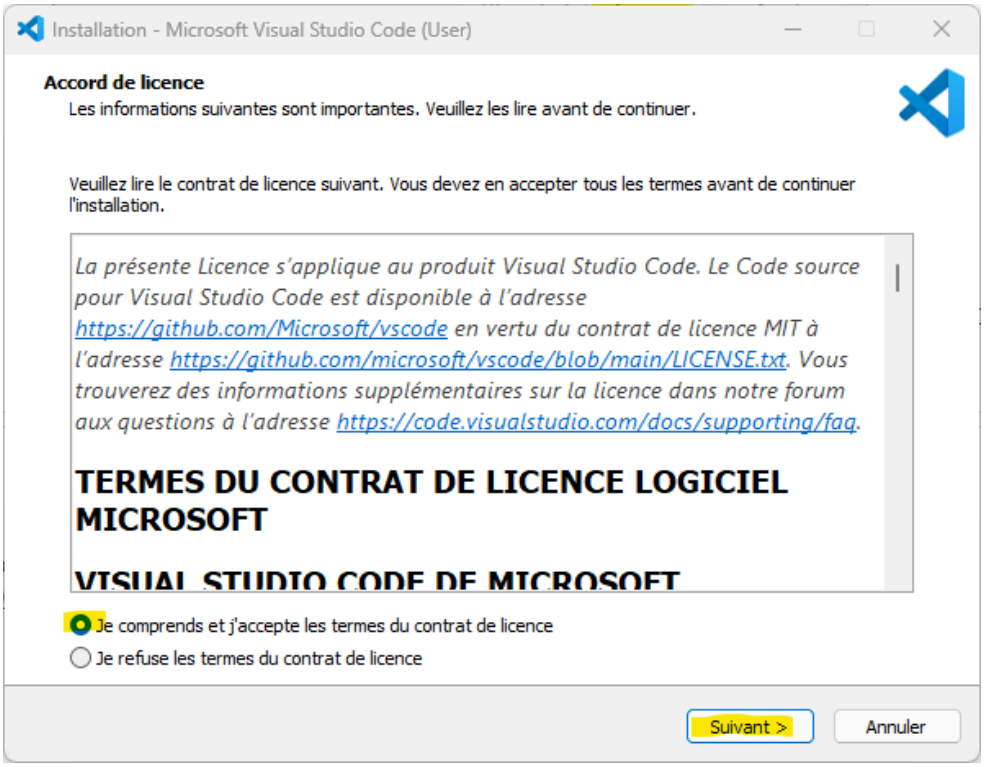
C'est terminé ! Cliquez sur "Fermer".

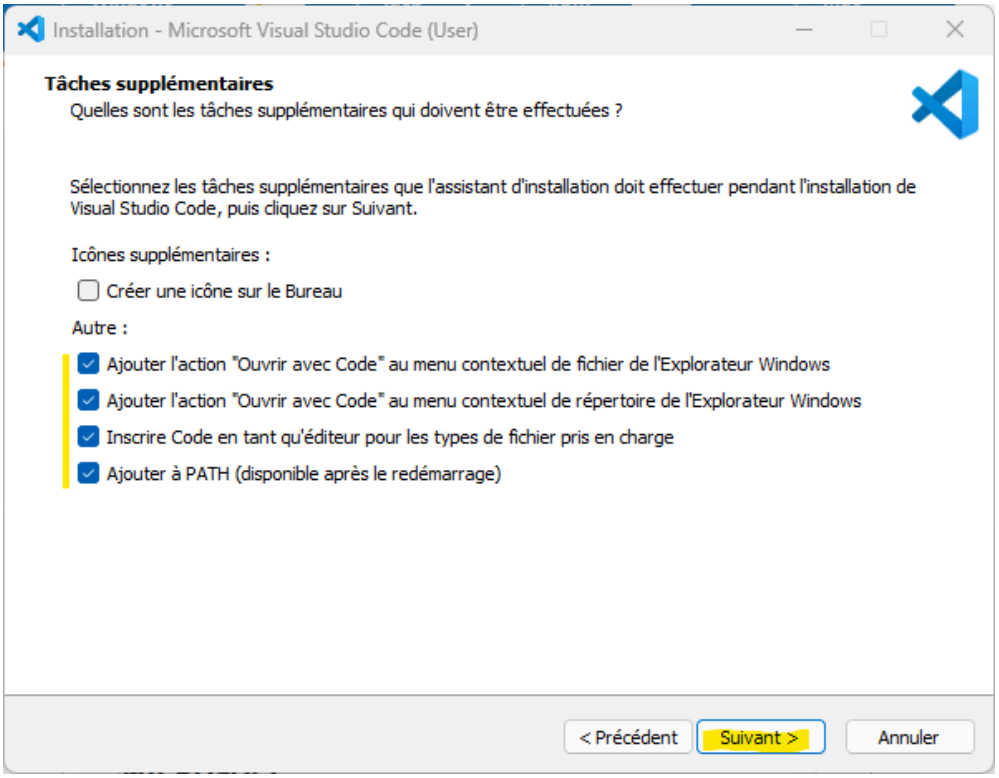
Bravo, une première étape de franchie !

Une fois Love2D installé, vous pouvez procéder à l'installation de Visual Studio Code depuis : https://code.visualstudio.com/download

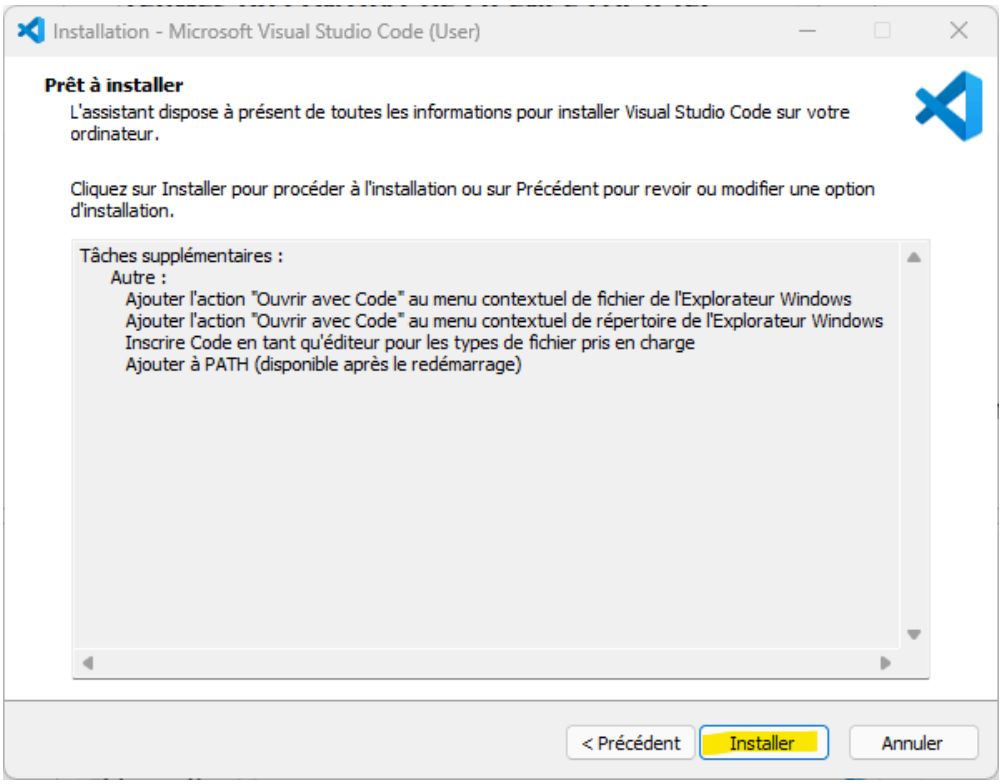


Cliquez sur le bouton bleu et suivez les instructions.





Cochez toutes ces options et cliquez sur "Suivant".

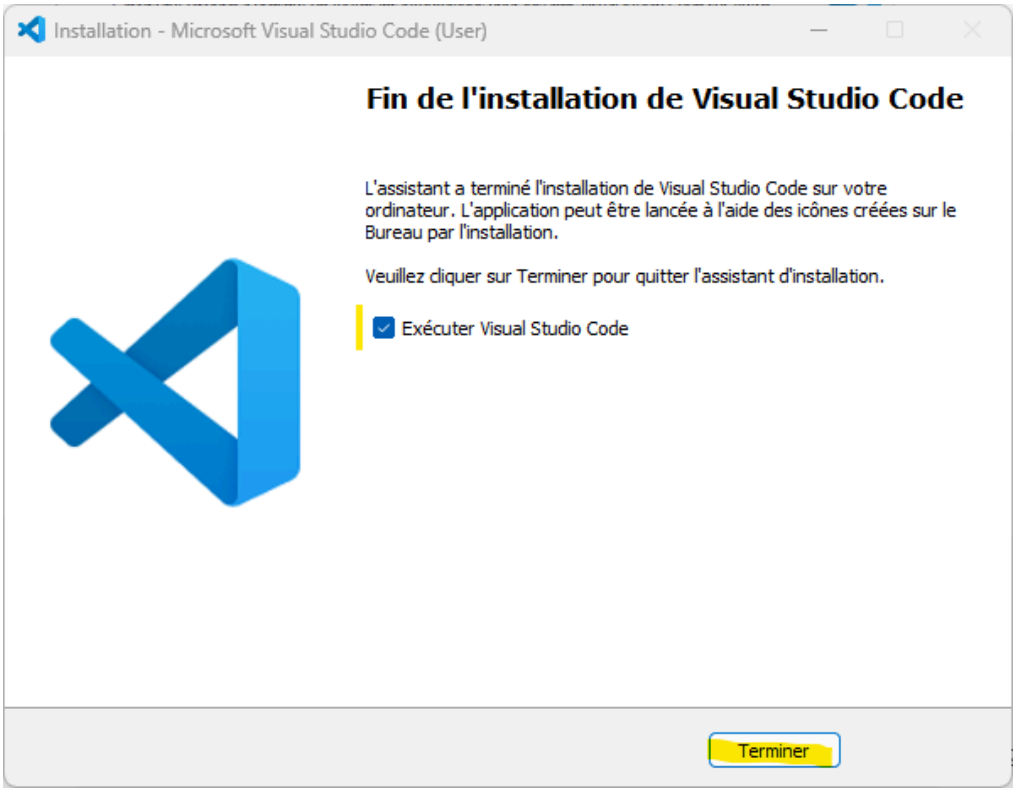


Terminez l'installation en cliquant sur "Installer".

Une fois Visual Studio installé procédez à son paramétrage pour l'adapter à Lua et Love2D.

Pour cela, commencez par le lancer.

Une icône a été ajoutée dans le menu démarrer ou bien acceptez la proposition d'exécuter Visual Studio Code en fin d'installation :



Pourquoi paramétrer Visual Studio Code ?

Sans paramétrage, Visual Studio Code sera un simple éditeur (comme Word) et ne vous permettra pas de lancer facilement vos programmes.

De plus, il ne reconnaîtra pas le langage Lua correctement et ne permettra pas le débogage pas à pas de vos programmes.

Heureusement, Visual Studio Code est améliorable à l'aide d'extensions créées par Microsoft ou par la communauté.

Elles sont faciles à installer et cela ne vous prendra que quelques minutes.

Nous allons installer 4 extensions qui vont vous faciliter la vie :

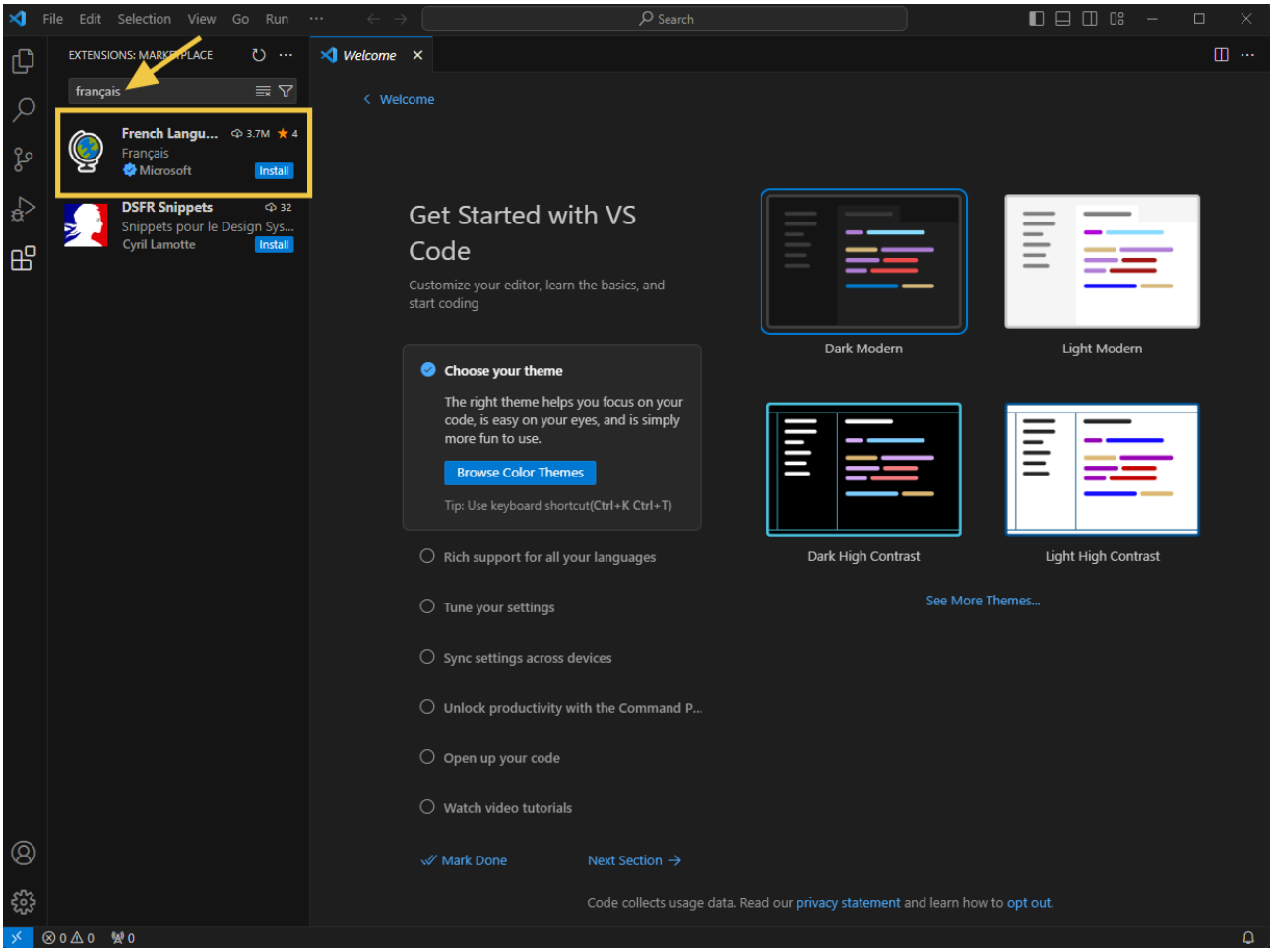
- La traduction française de Visual Studio Code
- Löve2D Launcher de Menerv
- vscode-lua de trixnz
- Local Lua Debugger de Tom Blind

Pour installer une extension :

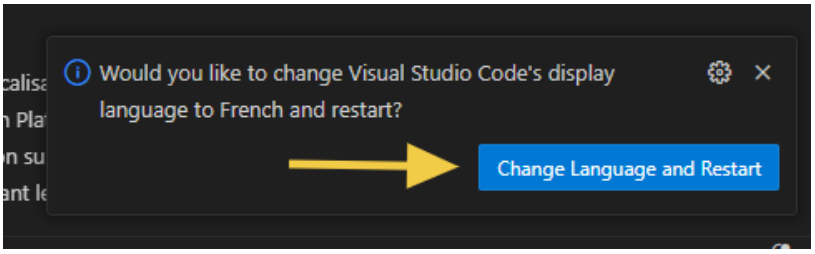
1. Cliquez sur l'onglet "Extensions" dans la barre de gauche. Il a la forme de 4 carrés.
2. Dans le champ de recherche, tapez le nom de l'extension suivi du nom de son auteur
3. Cliquez sur le petit bouton "installer" à droite de l'extension correspondante dans la liste

Exemple pour installer la traduction française :

4. Cliquez sur l'onglet "Extensions" dans la barre de gauche. Il a la forme de 4 carrés.
5. Dans le champ de recherche, tapez "français" ou "french"
6. Cliquez sur le petit bouton "install" à droite de l'extension correspondante dans la liste

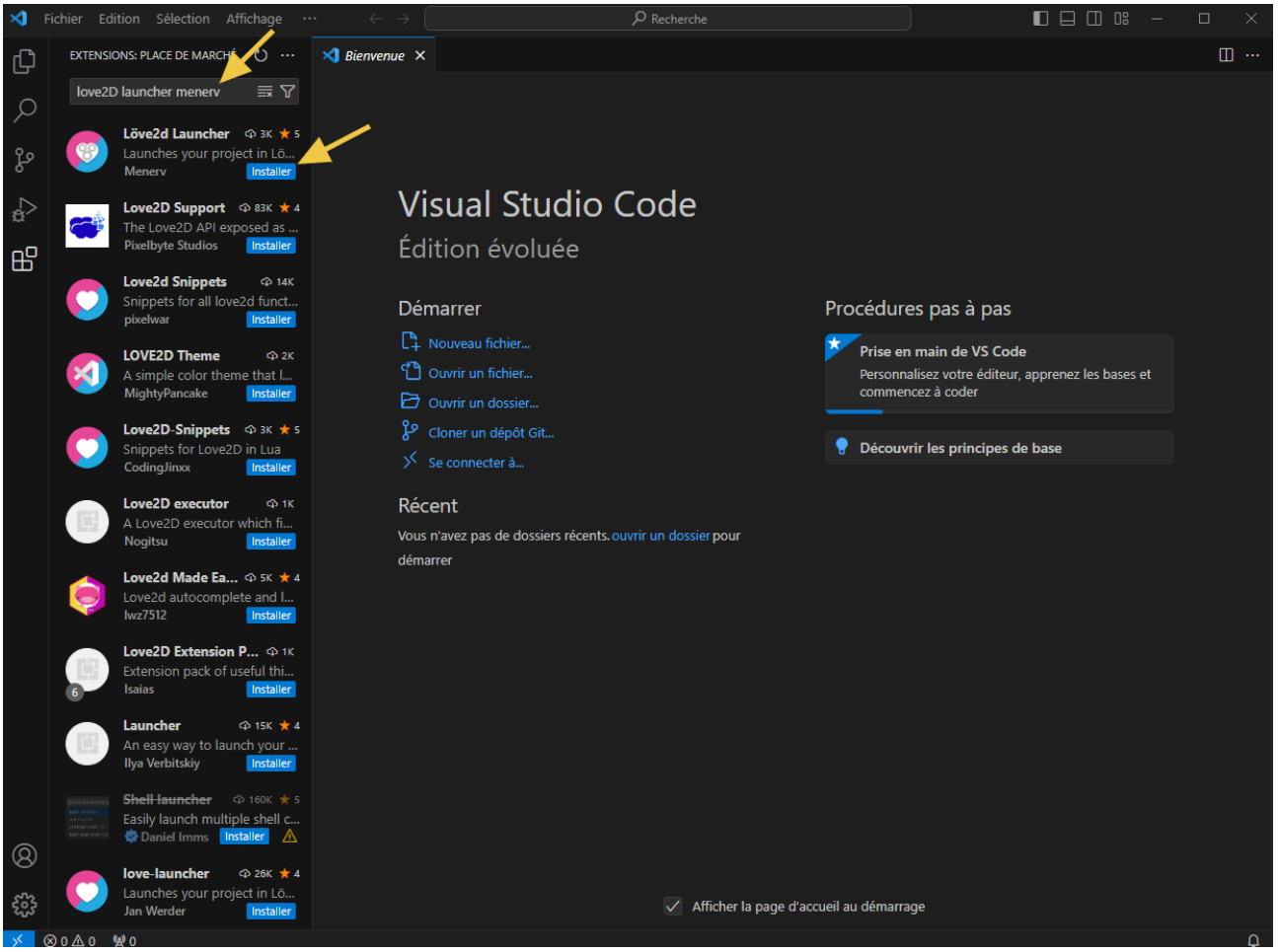


Le programme va ensuite vous proposer de se relancer pour prendre en compte la traduction :



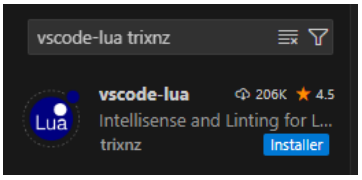
Cliquez sur "Change Language and Restart" et profitez de VS Code en français !

Ensuite, recommencez pour l'extension suivante : Love2D Launcher de Menerv.

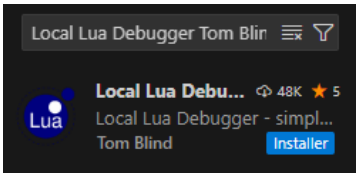


Inutile de relancer le logiciel pour celle-ci.

Puis pour "vscode-lua trixnz" :



Et enfin "Local Lua Debugger Tom Blind" :



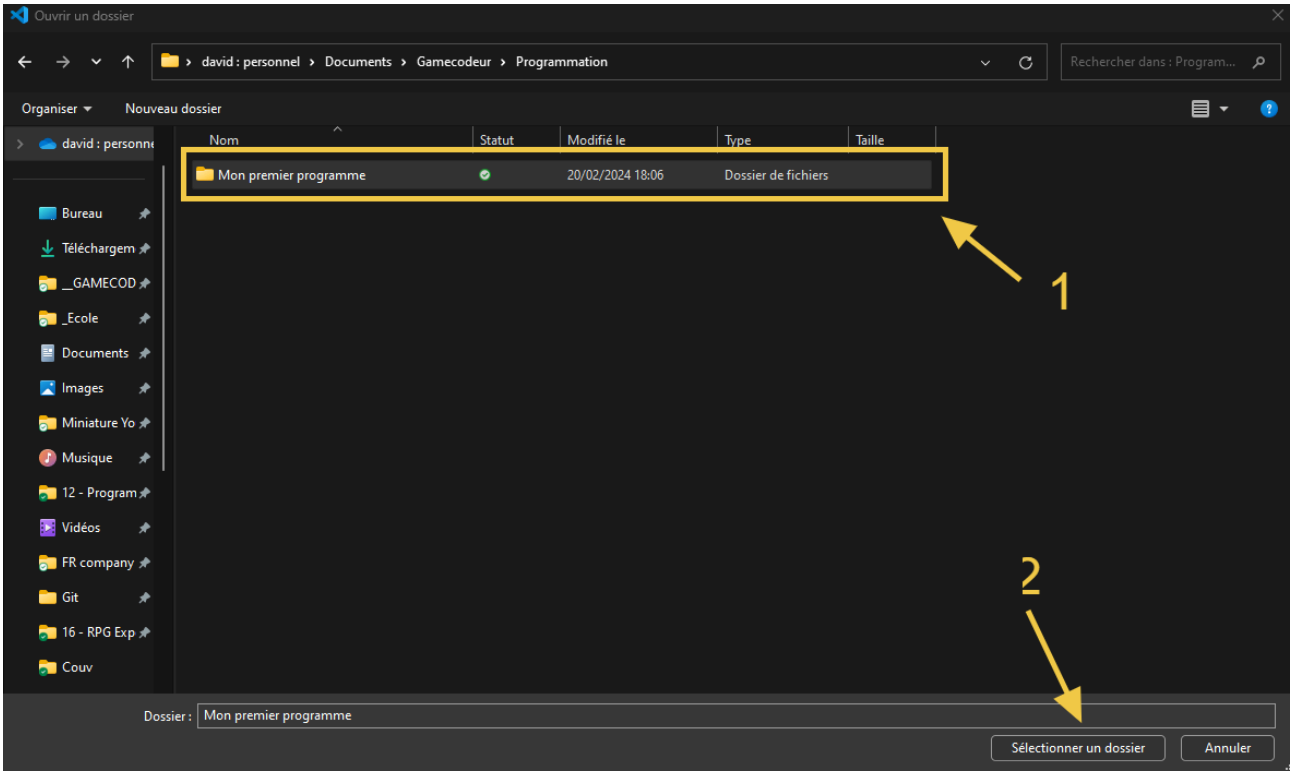
J'ai enregistré plusieurs vidéos qui vous expliquent les étapes une par une :
https://www.gamecodeur.fr/visual-studio-code-lua-love2d/

Votre premier programme qui ne fait rien

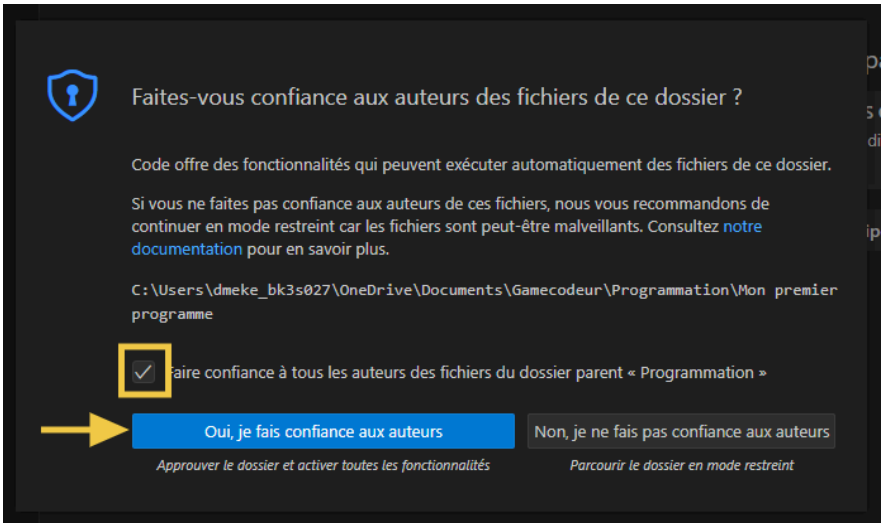
Tout d'abord téléchargez mon template de projet sur ce lien :
https://bit.ly/templatelovegc

Ensuite renommer le dossier du template, par exemple nommez-le "Mon premier programme", avec l'explorateur de fichiers.

Ensuite ouvrez le dossier "Mon programme" en cliquant sur le bouton "Ouvrir le dossier" proposé par Visual Studio Code ou en passant par son menu "Fichier / Ouvrir le dossier" et en allant sélectionner le dossier en question puis en cliquant sur "Sélectionner le dossier"..



Ensuite acceptez de faire confiance à ce dossier :



Mon template contient tout ce qu'il faut pour créer et déboguer un programme Love2D.

Vous préférez tout faire à la main et le débogueur ne vous intéresse pas ?

Vous pouvez simplement créer un dossier vide, puis créer un fichier "`main.lua`" :

1. Créez un dossier via l'explorateur de fichiers
2. Ouvrez ce dossier avec Visual Studio Code (Fichier / Ouvrir le dossier)
   (Suivez les instructions de la page précédente, notamment l'approbation de confiance)
3. Créez un nouveau fichier (Fichier / Nouveau fichier texte)
4. Enregistrez le fichier avec le nom "`main.lua`". (CTRL + S ou Fichier / Enregistrer)

Tapez le code suivant :

```lua
function love.load()

end

function love.update(dt)

end

function love.draw()

end

function love.keypressed(key)

end
```

Exécutez ensuite votre programme avec CTRL+B ou F5.

Vous obtenez une fenêtre noire :

Bravo, vous avez créé votre premier programme Love2D !

Apprendre facilement à programmer avec les 5 fondamentaux

Introduction

Apprendre à programmer peut sembler intimidant au premier abord. Mais en réalité, il suffit de maîtriser quelques concepts fondamentaux pour pouvoir créer des programmes simples, et des jeux vidéo complets !

Dans ce chapitre, nous allons explorer les 5 piliers de la programmation qui vous permettront de démarrer votre aventure dans le monde du code.

Les 5 fondamentaux en un clin d'oeil

1. Les variables et les expressions :

Variables : Imaginez des boîtes où vous stockez des informations, comme un nombre, un mot ou une phrase.

Expressions : Combinez des variables et des opérateurs mathématiques (+, -, *, /) pour obtenir de nouveaux résultats.

Exemple :

```
-- Déclaration de variables
vie = 100
score = 0

-- Expression pour calculer le nouveau score
score = score + 10

-- Affichage du score
print("Score:", score)
```

2. Les fonctions :

Ce sont des blocs de code réutilisables pour éviter de répéter les mêmes instructions.

Les fonctions permettent de découper votre programme en modules plus petits et plus faciles à comprendre.

Exemple :

```
-- Fonction pour afficher un message texte
function afficherMessage(message)
  print(message)
end

-- Appel de la fonction
afficherMessage("Bravo ! Vous avez gagné !")
```

3. Les structures de contrôle :

Elles permettent de contrôler le déroulement de votre programme.

Il s'agit des conditions et des boucles. Cela permet de diriger l'exécution du code en fonction de situations spécifiques.

Exemple:

```
if vie > 0 then
  -- Le joueur est encore en vie
  print("Continuez à jouer !")
else
  -- Le joueur a perdu
  print("Game Over !")
end
```

4. Les tableaux et les listes :

Il s'agit de stocker plusieurs valeurs de même type dans une structure ordonnée.

Exemple :

```
ennemis = {"Gobelin", "Orc", "Troll"}

-- Afficher le deuxième ennemi
print(ennemis[2])
```

5. Les objets :

Les objets regroupent des données et des fonctions liées à une entité du programme.

Ils permettent de créer des structures de données plus complexes.

Exemple :

```
objet = {
  x = 100,
  y = 200,
  vitesse = 10
}

-- Déplacer l'objet
objet.x = objet.x + objet.vitesse
```

Pourquoi ces 5 fondamentaux vont faire de vous un vrai programmeur ?

Croyez-moi. J'enseigne avec cette méthode depuis des années. Elle est redoutable. En maîtrisant ces concepts, vous aurez une compréhension solide des bases de la programmation. Vous pourrez ensuite explorer des langages et des technologies plus avancées avec aisance.

Voici quelques raisons pour vous convaincre…

En apprenant ces 5 fondamentaux, vous pouvez déjà commencer à programmer

Ne vous laissez pas intimider par la complexité des gros moteurs. Commencez par des programmes simples en utilisant un langage accessible comme Lua et un framework convivial comme Love2D. Vous serez surpris de voir ce que vous allez accomplir en peu de temps !

En les apprenant dans l'ordre de ma méthode, vous allez apprendre vite

L'ordre que je propose est pédagogique et permet de construire votre compréhension étape par étape, en commençant par le plus simple jusqu'au plus complexe.

Chaque concept s'appuie sur les précédents, ce qui facilitera votre apprentissage.

En les appliquant à Lua et Love2D, vous pourrez créer des jeux vidéo

C'est une manière ludique et motivante de mettre en pratique vos connaissances. En créant des jeux, vous explorez les différents aspects de la programmation et développez votre créativité.

On peut transposer ces connaissances à quasiment tous les autres langages

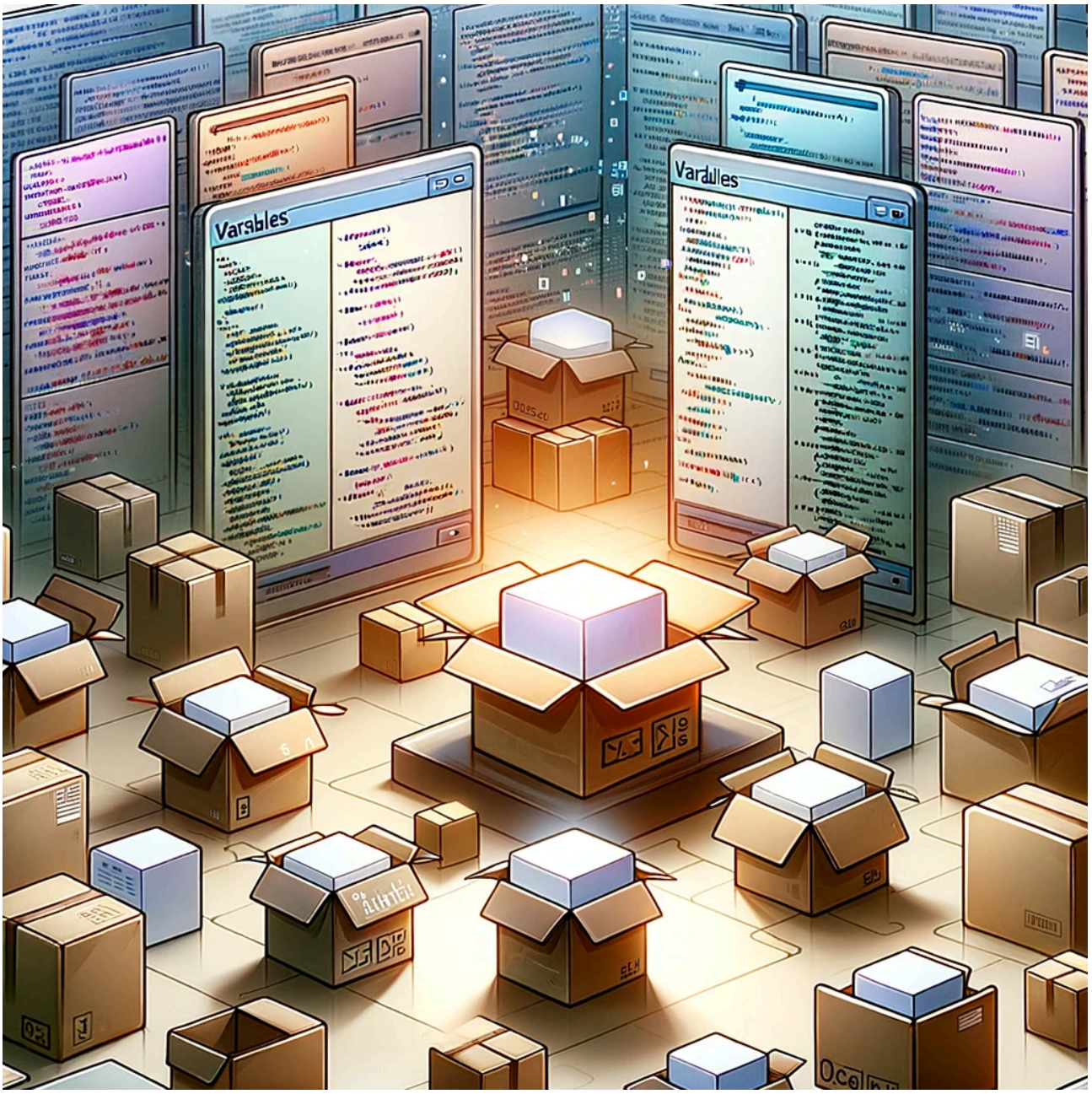
Les concepts que vous allez apprendre sont universels. Une fois que vous les aurez maîtrisés, vous pourrez facilement apprendre de nouveaux langages et vous adapter à différents environnements de développement.

Recommandations

- Approfondissez chacun des 5 fondamentaux avec sérieux, expérimentez beaucoup, jusqu'à les maîtriser.
- Prenez cela comme un jeu et donnez vous le temps de comprendre.
- La programmation est un processus itératif qui demande du temps et de la patience. N'abandonnez pas si vous rencontrez des difficultés. Persévérez !
- Ne vous encombrez pas d'outils lourds, vous pouvez même apprendre sur un site comme https://www.tutorialspoint.com/execute_lua_online.php qui permet de taper du code Lua et de l'exécuter directement.

Maintenant passons aux choses sérieuses, ouvrez votre éditeur de code, on va commencer !

Fondamental 1 : Les variables et les expressions



Qu'est-ce qu'une variable ?

Imaginez des boîtes dans lesquelles vous stockez des informations, comme un nombre, un mot…. Ces boîtes sont appelées des variables. Elles permettent de conserver des données que votre programme peut ensuite utiliser et modifier.

J'aime utiliser la métaphore de la boîte. On ouvre une boîte vide et on y met quelque chose dedans, et pour se rappeler de ce qu'elle contient, on écrit le nom de son contenu sur le couvercle.

Une variable c'est la même chose !

Exemples de données stockées dans des variables :

- Le nombre de vies de votre personnage
- Son niveau d'énergie
- Son état : mort ou pas
- La liste des ennemis à l'écran
- Le score du joueur

Types de données :

Chaque variable possède un type qui détermine la nature des données qu'elle peut stocker.

Voici les trois types les plus courants :

- Numérique : Pour les nombres entiers (5) ou flottants (5.2).
- Booléen : Pour les valeurs vraies (true) ou fausses (false).
- Chaîne de caractères : Pour les textes ("Nombre de vies : ").

Comment créer une variable :

C'est comme écrire le nom de la variable sur la boîte, puis mettre quelque chose dedans.

Exemple :

```
-- Déclaration de variables
vie = 100
nom = "Conan"
```

Comment nommer une variable ?

C'est ce que vous écrivez sur la boîte pour vous rappeler de ce qu'elle contient.

Il y a des règles simples à respecter :

- Il doit être unique (on ne peut pas avoir 2 variables du même nom).
- Il ne peut pas contenir d'espaces, d'accents ou de caractères spéciaux.
- Il doit commencer par une lettre.

Comment affecter une valeur à une variable ?

Pour affecter une valeur à une variable on utilise le signe égal : =.

Exemple :

```
energie = 100
```

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, un peu comme un dossier thématique.

On utilise pour cela une "table" qui permet de regrouper plusieurs valeurs dans une variable.

Comment créer une table ?

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

```
heros = {}
```

Ceci crée une table vide appelée `heros`.

Comment ajouter des valeurs à une table ?

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

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

Ceci ajoute deux valeurs à la table heros : `vies` et `energie`. Dans cet exemple `vies` est définie à 5 et `energie` est définie à 100.

Alternativement, on peut créer une table en lui donnant directement son contenu :

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

Ceci est équivalent à l'exemple précédent, mais en une seule ligne de code.

Comment accéder aux valeurs d'une table ?

Pour accéder à une valeur d'une table, on utilise le même point que pour l'ajouter :

```
vie = heros.vies
energie = heros.energie
```

Nous reparlerons plus tard des tables dans le 5e fondamental.

Qu'est-ce qu'une expression ?

C'est une combinaison de variables et d'opérateurs mathématiques (+, -, *, /) qui permet de calculer de nouvelles valeurs.

Exemple : `1 + 1`

Cette expression va donner 2 comme nouvelle valeur (le résultat de 1 + 1).

Une expression peut utiliser des variables ou des valeurs constantes.

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)
```

Exemple :

```
vie_restante = vie - 10
```

Comment cette expression est évaluée ?

- L'ordinateur recherche la valeur de la variable `vie`.
- Il soustrait `10` à cette valeur.
- Il stocke le résultat dans la variable `vie_restante`.

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"
```

Comprendre la précédence des opérations dans les expressions

Voici un exemple d'expression particulier :

```
score + points * 10
```

En Lua, et comme dans tous les langages de programmation, l'ordre d'évaluation des expressions est défini par la notion de précédence.

Par défaut, les multiplications et les divisions sont effectuées avant les additions et les soustractions.

Ceci signifie que l'expression suivante :

```
score + points * 10
```

…sera calculée de la manière suivante :

Multiplication : `points * 10`
Addition : `score + (points * 10)`

Le résultat sera donc score + (10 * points).

Les parenthèses permettent de modifier l'ordre d'évaluation par défaut.

Dans l'exemple suivant :

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

Les parenthèses forcent l'addition de score et points à être effectuée avant la multiplication par 10.

Voici quelques exemples pour illustrer la notion de précédence :

```
-- 10 + 5 * 2 = 20
valeur = 10 + 5 * 2

-- (10 + 5) * 2 = 30
valeur = (10 + 5) * 2

-- 10 * 5 + 2 = 52
valeur = 10 * 5 + 2

-- 10 * (5 + 2) = 70
valeur = 10 * (5 + 2)
```

Constatez comment les parenthèses peuvent changer le résultat.

Une expression peut aussi calculer si c'est VRAI ou FAUX :

Une expression booléenne est une combinaison de variables, d'opérateurs logiques (ET (and), OU (or), NON (not)) et de valeurs booléennes (VRAI (true) ou FAUX (false)) qui permet d'obtenir une valeur booléenne finale.

```
resultat = (vie > 0) and (score > 100)
```

Cette expression est vraie (true) si le personnage a encore de la vie (vie > 0) et si son score est supérieur à 100 (score > 100).

Les opérateurs booléens sont les suivants :

`and`	Vrai si les deux expressions sont vraies.
`or`	Vrai si au moins une expression est vraie.
`not`	Inverse la valeur booléenne de l'expression.

Chaque élément d'une expression peut être :

- Une constante : c'est une valeur fixe, comme un nombre (5) ou une chaîne de caractères ("Bonjour").
- Une variable : c'est une valeur qui peut changer et qui a un nom, comme le score d'un joueur ou la position d'un objet.
- Une fonction : C'est un code qui calcule et renvoie une valeur, par exemple la distance entre deux points.On verra les fonctions plus tard.

Note : on peut utiliser des expressions partout où une valeur est attendue, donc on peut les utiliser en arguments de fonctions ou dans des conditions. On verra ça plus tard mais c'est important de le préciser si vous en êtes à votre 2e lecture.

Pause exercice

Réalisez ces exercices sur papier ou directement dans Visual Studio Code.

Vous pouvez aussi les réaliser en ligne sur un site comme :
https://www.mycompiler.io/fr/online-lua-compiler

Exercice 1 : Score

Créez une variable pour stocker le score du joueur.

Donnez-lui le nom de votre choix et une valeur de 0. Affichez la valeur de la variable.

Exercice 2 : Titre

Créez une variable pour stocker le titre de votre jeu.

Donnez-lui le nom de votre choix et affichez-le.

Exercice 3 : Score final

Écrivez le code Lua pour calculer le score final d'un jeu.

Pour cela, additionnez le score du joueur et le bonus du niveau.

Exercice 4 : Déplacement

Écrivez le code Lua pour déplacer un personnage vers la droite en fonction de sa vitesse de déplacement en pixels

Exemple : Sa vitesse est de 10 pixels.

Exercice 5 : Attaque

Un personnage à 100 points de vie. Il est attaqué par un monstre qui lui inflige 20 points de dégâts.

Écrivez une expression pour calculer les points de vie restants du personnage après l'attaque.

Exercice 6 : Fin de niveau

Un personnage se trouve à une position sur l'axe X. La fin du niveau se trouve à la position 100.

Écrivez une expression pour déterminer si le personnage a atteint la fin du niveau.

Solution de l'exercice 1 :

```
score = 0
print("Score:", score)
```

Solution de l'exercice 2:

```
titre = "L'aventure du Héros"
print(titre)
```

Solution de l'exercice 3 :

```
score_joueur = 1000
bonus_niveau = 200

score_final = score_joueur + bonus_niveau
```

Solution de l'exercice 4 :

```
position_x = 0
vitesse_deplacement = 10
position_x = position_x + vitesse_deplacement
```

Solution de l'exercice 5 :

```
vie = 100
vie = vie - 20
```

Solution de l'exercice 6 :

```
position_x = 50
fin_niveau = 100
fin = position_x >= fin_niveau
```

Il y a plusieurs solutions possibles aux exercices donc si votre solution diffère mais qu'elle donne le bon résultat, pas de soucis.

La portée des variables en Lua

La portée des variables est un concept fondamental en programmation qui détermine où une variable peut être accédée dans votre code. En Lua, il existe deux types principaux de portée : globale et locale.

Variables globales

Une variable globale en Lua est accessible de n'importe où dans votre programme, après son point de définition. Les variables globales sont pratiques mais peuvent conduire à des conflits et des erreurs difficiles à tracer, surtout dans les grands programmes.

```lua
variableGlobale = "Je suis accessible partout !"

function afficheVariable()
    print(variableGlobale)  -- Affichera "Je suis accessible partout !"
end
```

Variables locales

Les variables locales, en revanche, ont une portée limitée au bloc où elles sont déclarées. Un bloc peut être une fonction, une boucle ou tout bloc de code délimité par un `end`. Utiliser des variables locales aide à éviter les conflits de noms et conserve la mémoire en libérant des variables inutilisées lors de la sortie du bloc de code.

```lua
function testLocal()
    local variableLocale = "Je suis locale à testLocal."
    print(variableLocale)  -- Fonctionne parfaitement
end

print(variableLocale)
```

Ce code provoquerait une erreur car `variableLocale` n'est pas accessible.

Portée dans les boucles et des structures de contrôle

En Lua, même les variables déclarées dans des boucles ou des conditions peuvent avoir une portée locale à ces blocs.

```lua
if true then
    local variableDansIf = "Accessible seulement dans ce if."
    print(variableDansIf)  -- Fonctionne
end
print(variableDansIf)  -- Ne fonctionne pas
```

Utilisez le mot clé "local" le plus possible et adaptez votre code pour éviter d'utiliser des variables globales, par exemple en passant la variable en paramètre. Comprendre la portée vous aidera à éviter des bugs parfois subtils.

Fondamental 2 : Les fonctions

Qu'est-ce qu'une fonction ?

Une fonction est un ensemble de lignes de code regroupées pour accomplir une tâche spécifique et optionnellement retourner un résultat. Cela permet d'éviter de répéter le même code à plusieurs endroits et cela structure votre programme pour le rendre lisible et plus facilement maintenable.

Exemples :

- Créer une fonction "`InitJeu`" pour gérer toutes les actions nécessaires pour remettre le jeu à zéro et recommencer une nouvelle partie.

- Créer une fonction "`SpawnEnnemi`" qui se chargera de calculer où est-ce qu'il spawne et de l'ajouter à l'écran. Vous pourrez appeler cette fonction à chaque fois qu'un nouvel ennemi est nécessaire.

- Si nous avons un calcul de distance à réaliser plusieurs fois au cours de l'exécution de notre jeu, on peut créer une fonction "`CalculDistance`" pour calculer la distance entre deux points.

Comment créer une fonction ?

On utilise le mot clé `function` pour déclarer une fonction.

La syntaxe est la suivante :

```
function nom_fonction(parametre1, parametre2, ...)
  -- Corps de la fonction
end
```

Note : Les "..." signifient juste qu'on peut avoir autant de paramètres que l'on souhaite.

Fonctions avec retour de valeur

Une fonction peut retourner une valeur en utilisant le mot clé `return`.

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

Explication :

`Addition` est le nom de la fonction.
`valeur1` et `valeur2` sont les paramètres de la fonction.

Ici le corps de la fonction est composé d'une seule ligne qui additionne les deux arguments et retourne le résultat. Cette fonction retourne donc directement le résultat d'une expression.

Une fonction peut contenir autant de lignes de code que nécessaire.

Voici la même fonction avec un code décomposé :

```
function Addition(valeur1, valeur2)
  local resultat = 0
  resultat = valeur1 + valeur2
  return resultat
end
```

Comment appeler une fonction ?

Pour utiliser une fonction, il faut simplement l'appeler par son nom, suivi de parenthèses. Entre les parenthèses il faut lister les valeurs qu'elle attend, dans l'ordre exact de sa déclaration.

On parle alors d'arguments de fonctions. Perso parfois je dis paramètre, parfois argument…

Voici le VRAI vocabulaire :

- Le paramètre est le nom de la variable dans la déclaration de la fonction.
- L'argument est la valeur fournie à la fonction lors de l'appel.

Si la fonction ne reçoit pas de paramètres, il faut utiliser les parenthèses mais sans rien entre les deux. Omettre les parenthèses désignerait l'adresse de la fonction, et non pas son appel.

Exemple :

```
InitJeu() -- Appelle la fonction qui remet le jeu à zéro
```

Si la fonction retourne un résultat, on utilisera alors la fonction exactement comme si on utilisait une valeur ou une variable. Au moment du calcul de l'expression, l'appel de la fonction sera remplacé par la valeur qu'elle retournera.

Exemple d'utilisation d'une fonction au coeur d'une expression:

```
resultat = Addition(1, 2)
autreResultat = 10 + Addition(5, 20)
```

Exercice

Changement de niveau

Objectif :

Écrire une fonction nommée `prochain_niveau` qui calcule le nouveau niveau d'un joueur à partir de son niveau actuel.

Consignes :

- La fonction `prochain_niveau` doit prendre un argument en paramètre que vous nommerez `niveau_actuel`, et qui représentera le niveau actuel du joueur dans le jeu.
- La fonction doit calculer le nouveau niveau du joueur en ajoutant 1 au `niveau_actuel`.
- La fonction doit ensuite retourner le nouveau niveau du joueur.
- Testez la fonction en démarrant au niveau 1 et appelez-la plusieurs fois pour simuler la progression du joueur à travers les niveaux. Affichez le niveau actuel du joueur après chaque appel.

Voici une solution possible à l'exercice :

```lua
function changerNiveau(niveauActuel)
    -- Calculer le nouveau niveau
    local nouveauNiveau = niveauActuel + 1
    return nouveauNiveau
end

-- Démarrage au niveau 1
local niveauJoueur = 1

-- Simuler la progression
niveauJoueur = changerNiveau(niveauJoueur)
print("Le joueur est maintenant au niveau " .. niveauJoueur)

-- Répétez l'opération pour simuler d'autres changements de niveau
niveauJoueur = changerNiveau(niveauJoueur)
print("Le joueur est maintenant au niveau " .. niveauJoueur)
```

Résultat Attendu :

```
Le joueur est maintenant au niveau 2
Le joueur est maintenant au niveau 3
```

Explication :

- La fonction `changerNiveau` prend un argument `niveauActuel`.
- Elle reçoit donc "1" lors du 1er appel.
- La fonction stocke le résultat de l'expression `niveauActuel + 1` dans `nouveauNiveau`.
- La fonction retourne la valeur de `nouveauNiveau`.
- `niveauJoueur` est donc modifiée avec la valeur retournée par `changerNiveau`.

Lors du 1er appel :
`niveauJoueur` contient la valeur 1, donc `changerNiveau` retourne 2.

Lors du 2ème appel :
`niveauJoueur` contient la valeur 2, donc `changerNiveau` retourne 3.

Fondamental 3 : Les structures de contrôle

C'est quoi les structures de contrôle ?

Les structures de contrôle constituent l'intelligence de votre code, lui permettant de ne pas s'exécuter de manière linéaire ce qui n'aurait aucun intérêt.

Essentielles en programmation, elles injectent de la dynamique au code, lui offrant la capacité de réagir et de s'adapter selon divers contextes ou stimuli : état du jeu, évènements, etc.

Sans structures de contrôle, votre programme suivrait un chemin fixe, exécutant bêtement chaque ligne de code de haut en bas.

Il serait difficile, voire impossible, de créer des programmes interactifs qui répondent aux actions du joueur ou à ce qui se passe dans votre jeu, car le programme ne pourrait pas ajuster son comportement. Donc vous l'avez compris, c'est un gros morceau !

Nous avons deux familles de structures de contrôle :

- **Les conditions** :
  On utilise les conditions pour que le code s'exécute différemment selon une situation, ou selon le résultat d'un calcul.

- **Les boucles :**
  On utilise les boucles pour répéter un ensemble d'instructions un certain nombre de fois.

Découvrons tout ça en détail.

Les conditions : Le code à la carte

La structure de contrôle `if… then… else` est l'une des plus utilisées en programmation pour exécuter des blocs de code en fonction de conditions spécifiques. Cette structure peut être conceptualisée comme un simple "si… alors… sinon", où le code ne s'exécute que si une condition prédéfinie est remplie.

Prenons un exemple non informatique pour illustrer :

"Si il pleut, alors prends un parapluie… sinon, ne prends pas de parapluie."

Dans le contexte du développement de jeux vidéo, cette logique conditionnelle est fondamentale. Énormément de conditions sont présentes dans le code d'un jeu vidéo.

Exemple de condition :

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

Cette instruction signifie que si la distance séparant le héros de l'ennemi est inférieure à 100 unités, le jeu affichera le message "Alerte".

Cela permet de créer des interactions dynamiques et réactives dans le jeu, où les actions et les réactions des personnages dépendent de leur environnement et de circonstances.

Sans conditions, votre code ne prendra aucune décision et s'exécutera de manière linéaire.

Vous comprendrez donc que ce concept est plus que fondamental.

Pratiquez-le jusqu'à l'avoir compris à 100 % !

Une condition compare ou teste des valeurs et elle 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** (*true* est le mot clé qui veut dire *vrai* en code Lua).
`false` est une condition **fausse**. (*false* est le mot clé qui veut dire *faux* en code Lua).

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

Reprenons notre exemple du héros (voir la leçon sur les variables) et son état `blesse` :

```
if heros.blesse then -- équivalent de ; 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, ce code serait problématique :

```
if (heros.blesse = true) ...
```

Il aurait pour effet de modifier la valeur de `heros.blesse` pour y inscrire `true` !

Heureusement, en Lua, si vous oubliez le double égal, vous aurez l'erreur suivante :

```
'then' expected near '='
```

Donc si vous voyez cette erreur s'afficher, vous saurez !

Les différents opérateurs de comparaisons :

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, elle est donc non nulle. 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 !

Mais 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"
test(100) -- Cet appel affichera "j'ai bien reçu une valeur"
```

Les conditions multiples et imbriquées :

Voici un exemple avec le mot clé `else` qui veut dire "sinon" :

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

On peut enchaîner plusieurs "`if`" au même niveau grâce au mot clé `elseif` (sinon si).

```
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 (une condition dans une condition) :

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

Il n'y a aucune limite, sauf celle de la lisibilité de votre code…

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

Ces mots clés permettent de combiner plusieurs conditions.

AND (ET)

L'opérateur `and` est utilisé pour vérifier si toutes les conditions spécifiées sont vraies.

Pour qu'une expression utilisant `and` soit évaluée comme vraie, chaque condition individuelle liée par `and` doit être vraie. Si au moins une des conditions est fausse, l'expression entière sera évaluée comme fausse.

Cet opérateur est particulièrement utile lorsque vous avez besoin que plusieurs critères soient remplis pour exécuter un bloc de code.

Exemple :

```
if niveauJoueur > 5 and possedeCle == true then
    print("Vous avez débloqué la porte secrète !")
end
```

Le message ne sera affiché que si le joueur est de niveau 6 ou plus ET possède la clé.

OR (OU)

L'opérateur `or` est utilisé pour vérifier si au moins une des conditions spécifiées est vraie. Une expression utilisant `or` est évaluée comme vraie si au moins une des conditions individuelles est vraie. Cet opérateur est utile lorsque vous souhaitez exécuter un bloc de code lorsqu'au moins l'un des critères de la condition est rempli.

Exemple :

```
if jourJeu == "samedi" or jourJeu == "dimanche" then
    print("Événement spécial weekend activé ! Bonus doublés.")
    bonus = true
end
```

Dans cet exemple, le bonus sera activé si le jour est SOIT "samedi" SOIT "dimanche".

Autres exemples plus complexes :

```
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
```

Analyse du Code :

Première condition :

```
if energie < 10 and alerte == false then
```

Cette ligne vérifie si deux conditions sont remplies : d'abord, si la variable `energie` est inférieure à 10, et ensuite, si la variable `alerte` est égale à `false`. L'opérateur `and` s'assure que les deux conditions doivent être vraies pour que le bloc de code qui suit soit exécuté. Si ces conditions sont remplies, le programme affiche "Alerte rouge" et change la valeur de `alerte` à "rouge".

Deuxième condition :

```
if alerte == "rouge" or alerte == "orange" then
```

Cette condition utilise l'opérateur `or` pour vérifier si l'une des deux conditions est vraie : si `alerte` est égale à "rouge" ou "orange". Si au moins l'une de ces conditions est remplie, le programme affiche "Alerte rouge ou orange !". L'utilisation de `or` permet d'exécuter le bloc de code si l'une des conditions spécifiées est vraie.

Troisième condition :

```
if alerte ~= "rouge" then
```

Cette ligne vérifie si alerte n'est pas égale à "rouge" en utilisant l'opérateur `~=`, qui signifie "différent de" en Lua. Si `alerte` a une valeur différente de "rouge", le programme affiche "Pas d'alerte rouge en cours".

Note concernant le caractère ~ (tilde) :

Pour obtenir le caractère ~ (tilde) sur un clavier AZERTY (qui est le type de clavier couramment utilisé dans les pays francophones) vous devez généralement suivre ces étapes :

- Maintenez la touche Alt Gr (Alt Graphique) située à droite de la barre d'espace.
- Tout en maintenant Alt Gr, appuyez sur la touche comportant le symbole ~. (Sur la plupart des claviers AZERTY, cette touche est avec le chiffre 2 situé en haut à gauche du clavier.
- Appuyez sur la barre d'espace ou tapez un autre caractère (comme le caractère = si vous voulez écrire ~=) : le caractère apparaît

Il est important de noter que la disposition des touches peut légèrement varier selon les claviers ou les configurations régionales. Cependant, la méthode décrite ci-dessus est celle communément utilisée, et c'est celle que j'utilise.

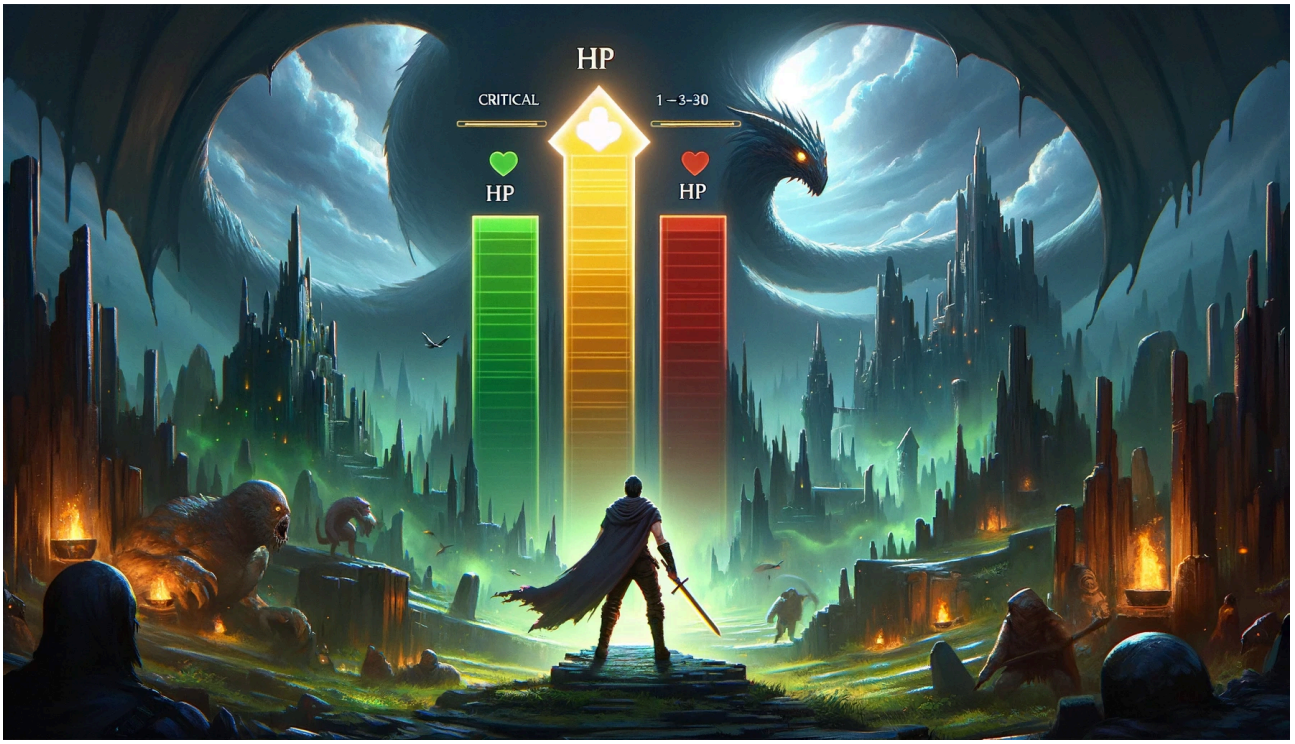
Pourquoi utiliser un caractère aussi compliqué à taper ?

Lua est un langage de programmation créé par un groupe de développeurs brésiliens constitué de chercheurs de l'Université Pontificale Catholique de Rio de Janeiro, au Brésil, dans les années 1990.

Sa relation avec le caractère tilde ("~") vient de son origine brésilienne.

Sur les claviers brésiliens, qui suivent généralement la disposition QWERTY avec quelques adaptations pour les caractères spécifiques au portugais, le caractère ~ est plus facilement accessible qu'il ne l'est sur les claviers AZERTY. En portugais, le tilde est utilisé comme un accent graphique, notamment sur les voyelles a et o pour indiquer une nasalisation, et il occupe donc une place directement accessible sur le clavier. Ce n'est pas le cas en France, d'où la petite gymnastique pour l'obtenir !

Exercice : Points de Vie



Dans un jeu vidéo, le personnage du joueur possède des points de vie (PV).

Vous êtes chargé de programmer la logique qui gère les situations suivantes :

- Si les PV du personnage tombent à 0 ou en dessous, le personnage meurt, et le jeu affiche "Vous avez perdu !".

- Si les PV sont entre 1 et 30, le jeu affiche "Attention : santé critique !", incitant le joueur à être prudent ou à chercher des soins.

Dans tous les autres cas, le jeu affiche "Le personnage est en bonne santé."

Les points de vie du personnage devront être stockés dans la variable `pv`.

Vous pouvez juste afficher des messages, inutile de tout programmer.

Ne regardez pas la page suivante avant d'avoir essayé de réaliser l'exercice !

Solution de l'exercice

Voici une solution possible :

```lua
-- Initialisation des points de vie à 25 pour l'exemple
local pv = 25

-- Vérification de l'état de santé du personnage
if pv <= 0 then
    print("Vous avez perdu !")
elseif pv <= 30 then
    print("Attention : santé critique !")
else
    print("Le personnage est en bonne santé.")
end
```

J'ai utilisé 25 mais vous pouvez modifier cette valeur pour tester différents scénarios.

> Note :
>
> Je vous conseille d'ailleurs de tester tous les cas de figure quand vous codez des conditions, pour ne pas laisser de bug caché dans votre code.

Explications :

Condition de défaite : `if pv <= 0 then`

Cette ligne vérifie si les points de vie sont inférieurs ou égaux à 0. Si c'est le cas, cela signifie que le personnage du joueur n'a plus de vie, donc le jeu affiche le message indiquant la défaite du joueur.

Santé critique : `elseif pv <= 30 then`

Si la première condition n'est pas remplie (c'est-à-dire que les PV sont supérieurs à 0), le programme vérifie ensuite si les PV sont inférieurs ou égaux à 30. Cette plage de valeurs indique que le personnage est en vie mais avec une santé critique, nécessitant probablement des actions pour récupérer des PV. Un message d'avertissement est alors affiché.

Bonne Santé : `else`

Si aucune des conditions précédentes n'est vraie (les PV sont donc supérieurs à 30), cela signifie que le personnage est en bonne santé. Le jeu informe alors le joueur que son personnage est dans un état stable.

Avez-vous réussi cet exercice ?

Si ce n'est pas le cas : Relisez la solution, analysez le raisonnement jusqu'à le comprendre, cachez la solution et recommencez sans recopier.

Les boucles : et ça continue encore et encore

Les boucles avec `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. Un peu comme un compteur.

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, par 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 second (`10`) est la valeur de fin. Par défaut, la variable `compteur` 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)
```

… et ceci 10 fois en tout !

Pour une boucle à l'envers, et donc si on veut afficher de 10 à 1 on devra ajouter un paramètre "-1" pour indiquer qu'on recule de 1 en 1, on parle alors de "step" négatif :

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

Important : une boucle peut contenir un nombre illimité de lignes de code :

```
for …
   [ligne de code]
   [ligne de code]
   [ligne de code]
   ...
end
```

Les boucles avec `while`

Une boucle `while` va répéter un bloc de code "tant que" une condition est 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 souhaitez interrompre une boucle pour continuer le programme, utilisez le mot clé `break` (qui veut dire grosso modo "casser" en français).

La boucle interrompt alors et son 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
```

Exercices

Exercice 1 :

Créer une boucle qui affiche (avec print) les valeurs de 0 à 100

Exercice 2 :

Créer une boucle qui affiche les valeurs de 100 à 0

Exercice 3 :

Créer une fonction qui affiche la table de multiplication du chiffre reçu en paramètre.

Solutions

Exercice 1 :

Créer une boucle qui affiche les valeurs de 0 à 100 :

```
for i = 0, 100 do
    print(i)
end
```

Exercice 2 :

Créer une boucle qui affiche les valeurs de 100 à 0 :

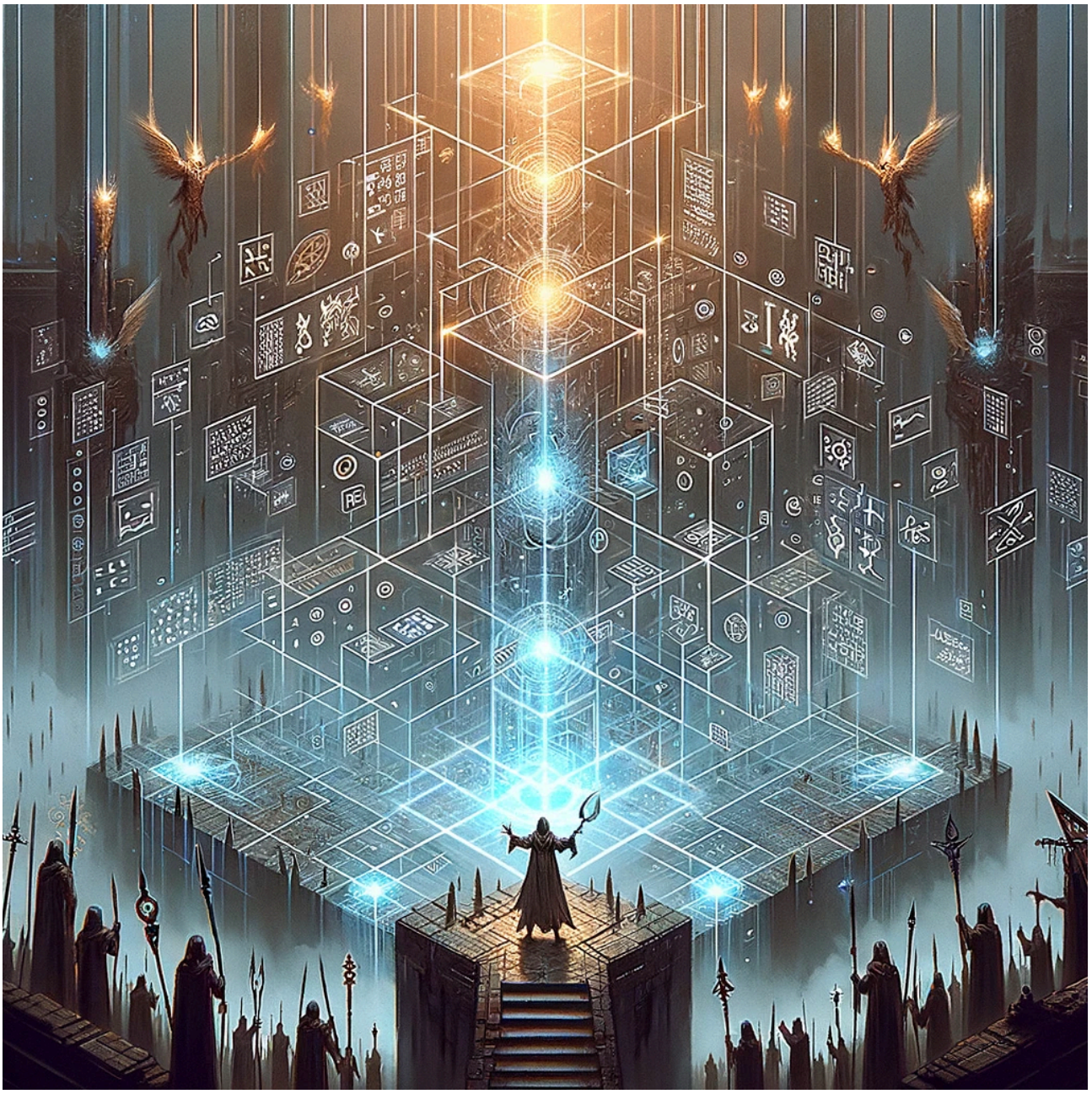
```
for i = 100, 0, -1 do
    print(i)
end
```

Exercice 3 :

Créer une fonction qui affiche la table de multiplication du chiffre reçu en paramètre :

```
function afficherTableMultiplication(chiffre)
    for i = 1, 10 do
        local resultat = chiffre * i
        print(chiffre .. " x " .. i .. " = " .. resultat)
    end
end
```

Fondamental 4 : les listes et les tableaux



Dans ce chapitre, nous allons explorer en profondeur les concepts des tableaux et des listes en Lua, des structures de données essentielles pour le développement de jeux vidéo.

Ces structures permettent de stocker des niveaux, des objets à l'écran, des ennemis, des tirs, etc., de manière organisée et pratique.

Encore un fondamental à travailler en profondeur. Ne passez pas dessus trop rapidement !

La nature universelle des tables en Lua

Retenez ceci :

En Lua tout ce qui contient plusieurs valeurs est une table : un tableau est une table, une liste est une table, une structure de données (variable composée) est une table…

OK… Mais qu'est-ce qu'une table ?

En Lua, une table est une collection hétérogène de clés et de valeurs. Les clés peuvent être de n'importe quel type de données, tandis que les valeurs peuvent être de tout type, y compris d'autres tables ou des références de fonctions, créant ainsi des structures de données imbriquées et complexes.

Une table en Lua peut être utilisée à la fois comme un tableau (array) pour stocker des éléments de manière séquentielle, ou comme une liste (ou un dictionnaire) pour associer des valeurs à des clés spécifiques et les parcourir.

Nous avons déjà abordé les tables, dans une utilisation simple, quand nous avons parlé des variables complexes, et de l'exemple du héro.

Ici on va utiliser les tables pour créer des collections : listes ou tableaux.

Et dans un jeu vidéo on a quasiment systématiquement besoin de gérer des listes de choses : tirs, ennemis, objets, particules… et des tableaux : maps des niveaux par exemple.

Comment créer une table et accéder à ses éléments ?

Les tables permettent de stocker une collection de valeurs. Pour accéder à une valeur, on utilise une clé après un point, ou entre crochets après le nom de la table. Par exemple :

Pour définir une table et y accéder avec la syntaxe utilisant un point :

```
monTableau = {}
monTableau.maCle = "ma valeur"
print(monTableau.maCle)
```

Et la syntaxe avec les crochets :

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

Bluffant cette polyvalence non ?

Personnellement j'utilise essentiellement la syntaxe avec un point (`table.valeur`). Je réserve la syntaxe avec les crochets pour créer des tableaux associatifs entre du texte et des valeurs (traductions, inventaire, etc.).

Utiliser les tables Lua pour créer des listes

Une liste est une table ordonnée par des index numériques.

Comment créer une liste ?

Voici la méthode "à la volée", tout en une ligne de code :

```
maListe = { "Valeur 1", "Valeur 2" }
```

Ou en la créant vide puis en ajoutant les valeurs une par une en précisant l'index entre crochets :

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

On peut aussi ajouter dynamiquement des éléments :

Utilisez la fonction `table.insert` pour ajouter des éléments à une liste :

```
maListe = {}
table.insert(maListe, "Valeur libre")
```

Ici, c'est Lua qui va calculer l'index pour vous. Voici ce qui se passe lorsque vous utilisez `table.insert` sur une liste :

- Lua commence par déterminer l'index sous lequel insérer la nouvelle valeur. Cette clé correspond à l'index numérique le plus élevé existant dans la liste, augmenté de 1. Si la liste est vide (comme c'est le cas au début de l'exemple), le premier index sera 1.

- Insertion de la valeur : Une fois l'index déterminé, Lua insère la valeur à cette position dans la table. Dans l'exemple, "Valeur libre" est ajouté à `maListe` avec l'index 1.

Si vous ajoutez une autre valeur ensuite avec `table.insert` :

- Lua continuera d'incrémenter automatiquement l'index pour chaque nouvel élément. Par exemple, si vous exécutez `table.insert(maListe, "Une autre valeur")` immédiatement après mon 1er exemple, "Une autre valeur" sera ajoutée à `maListe` avec l'index 2.

Ce qui donnerait en mémoire :

maListe	
Index	*Valeur*
1	"Valeur libre"
2	"Une autre valeur"

Cette gestion automatique des clés numériques par table.insert simplifie la création et la manipulation de listes en Lua. Vous n'avez pas besoin de vous soucier de la gestion des indices car Lua s'en charge pour vous, garantissant que les éléments sont toujours ajoutés à la fin de la liste et que les clés numériques restent continues.

Cela rend `table.insert` particulièrement utile pour gérer dynamiquement des données telles que des objets collectés, des ennemis générés, ou des projectiles tirés, sans vous préoccuper de leur indexation dans la liste.

Pour connaître le nombre d'éléments d'une liste, ajoutez le caractère `#` (dièse) devant son nom :

```
print(#maListe) -- Affiche le nombre d'éléments
```

Cet opérateur agit comme un interrogateur qui demande à la liste de renvoyer sa taille. Dans cet exemple, si `maListe` contient trois éléments, l'exécution de cette ligne de code affichera `3`.

Supprimer un élément d'une liste

Pour supprimer un élément d'une liste, il faut connaître sa position (son index).

Supprimer des éléments d'une liste est un besoin fréquent, par exemple, lors de la gestion des projectiles, des ennemis à l'écran, ou d'objets à collecter.

Pour supprimer un élément il faut utiliser la fonction `table.remove`. Elle permet de retirer des éléments d'une liste tout en maintenant l'intégrité des index (elle décale les index des éléments qui suivent celui qui a été supprimé).

La syntaxe de base de `table.remove` est la suivante :

```
table.remove(table, [position])
```

- `table` : La liste (ou tableau) à partir de laquelle un élément sera supprimé.
- `[position]` : L'index de l'élément à supprimer. Si ce paramètre est omis, c'est le dernier élément de la liste qui sera supprimé.

Exemple :

Imaginons que nous ayons une liste de projectiles lancés par un joueur :

```
projectiles = {}

table.insert(projectiles, {type = "laser", position = 100})
table.insert(projectiles, {type = "missile", position = 200})
table.insert(projectiles, {type = "laser", position = 300})
```

Supposons maintenant que le projectile en position 200 (le missile) atteigne sa cible. Pour le supprimer de la liste, nous utilisons table.remove :

```
-- Suppression du missile qui est à la position 2 dans notre liste
table.remove(projectiles, 2)
```

Dans cet exemple, après la suppression du missile, le projectile de type "laser" en position 300 se décale pour occuper l'index 2 car comme je l'ai dit les index sont automatiquement décalés.

ATTENTION : *C'est un exemple fictif car dans la réalité de la programmation, on ne connaît pas à l'avance l'index de l'élément à supprimer, on doit le rechercher. Par exemple, on va tester chaque projectile pour détecter ceux qui doivent être supprimés. Nous verrons plus tard comment supprimer des éléments en fonction d'un contexte.*

Parcourir les listes

Une liste ne sert à rien si l'on ne peut pas la parcourir pour utiliser son contenu.

Pour parcourir les listes, nous avons 3 possibilités : `for`, `ipairs` et `pairs`.

1. Méthode avec `for`

C'est ma méthode préférée :

- Elle est simple à écrire et à lire
- Elle est polyvalente
- Elle fonctionne aussi pour les suppressions (voir plus bas)

Prenons comme exemple une liste `maListe` qui ressemblerait à quelque chose comme :

```
maListe = {"dragon", "gobelin", "orc"}
```

Notre objectif est de parcourir cette liste et d'afficher chaque élément. Pour ce faire, nous utilisons la structure de contrôle `for` suivante :

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

Lisez ce code ainsi :

"Pour chaque `n`, en partant de `1` jusqu'à `#maliste`, exécute le code qui suit…". Dans notre cas c'est l'équivalent de "Pour chaque `n`, en partant de `1` jusqu'à `3`" puisque `#maListe` renvoie 3.

Note : *J'ai appelé ma variable "`n`" mais on peut lui donner le nom qu'on veut.*

Je récapitule ce que fait ici la boucle `for` :

- Initialisation : La boucle commence avec `n=1`. Cela définit la valeur de départ de `n`.

- Condition de continuation : `#maListe` renvoie le nombre d'éléments dans la liste, qui est 3 dans notre cas. La condition est donc d'exécuter le bloc de code contenu dans le `for` tant que `n` est inférieur ou égal à 3.

Rappel : Quand je dis "bloc de code", cela représente toutes les lignes de code entre le `for` et le `end` qui le termine. On peut avoir autant de lignes de code que l'on veut :

```
for …
   [ligne de code]
   [ligne de code]
   [ligne de code]
   ...
end
```

Allez, comme je sais que beaucoup galèrent à comprendre les boucles, je vais décomposer encore mon exemple en vous montrant ce qui se passe pour les 3 itérations (répétitions) :

Première Itération (`n=1`) :

Avec `n=1`, on peut accéder au premier élément de la liste. C'est l'équivalent d'avoir écrit `maListe[1]` car `n` contient la valeur `1`.

`print(maListe[n])` affichera donc "`dragon`".

Deuxième Itération (`n=2`) :

Maintenant, `n` vaut `2`. `maListe[n]` fait donc référence à l'équivalent de `maListe[2]`, donc le deuxième élément de la liste qui est "`gobelin`".

`print(maListe[n])` affiche donc "`gobelin`".

Troisième Itération (`n=3`) :

Avec `n=3`, `maListe[n]` pointe vers le troisième élément qui est "`orc`".

L'exécution de `print(maListe[n])` affiche donc "`orc`".

Fin de la Boucle :

La boucle `for` se termine car elle a respecté la condition fixée au départ (`n=1,3`).

Note : *Il est important de comprendre qu'on peut aussi créer des boucles et ne pas utiliser la valeur de n. La boucle ne sert alors qu'à répéter une action un certain nombre de fois.*

2. Méthode avec `ipairs`

L'utilisation de `ipairs` en Lua offre une autre méthode pour parcourir les listes, chaque méthode ayant ses propres avantages et inconvénients. `ipairs` est une fonction itératrice qui parcourt une table (liste) depuis le premier indice jusqu'au premier indice.

Voici comment `ipairs` est utilisé :

Supposons que nous ayons la même liste `maListe` avec trois éléments :

```
maListe = {"dragon", "gobelin", "orc"}
```

Pour parcourir cette liste avec ipairs, nous utiliserons :

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

Dans cette boucle :

- `i` recevra l'indice de l'élément courant dans la liste.
- `v` recevra la valeur de l'élément à cet indice.

Vous pouvez bien sûr donner le nom que vous voulez aux variables `i` et `v`.

`ipairs(maListe)` génère donc une paire contenant l'index et la valeur pour chaque élément de la liste, en commençant par l'indice 1 et en continuant séquentiellement jusqu'à la fin de la liste.

Avantages de `ipairs`

- La syntaxe est plus compacte et plus claire (certains la trouveront moins claire…), on obtient l'indice et la valeur en 1 seule commande, pas besoin de passer par `[n]`.

- `ipairs` s'arrête automatiquement à la première lacune dans les index numériques, ce qui peut prévenir les erreurs dans des listes non continues. Note : Je vous déconseille dans tous les cas de mélanger dans vos listes des index numériques et non numériques.

Inconvénients de `ipairs`

- Dans un jeu vidéo nous aurons souvent besoin de supprimer des éléments dans une liste, et `ipairs` ne permet pas de suppression pendant le parcours de la liste (voir plus loin).

- Syntaxe moins lisible : Pour ceux qui sont nouveaux en programmation, comprendre `ipairs` peut être un peu plus complexe que la boucle `for` basique.

Limitation de `ipairs` lors de la suppression d'éléments

Lorsque vous utilisez ipairs pour parcourir une liste, si vous supprimez un élément de la liste en cours de parcours cela va provoquer des bugs :

- Saut d'éléments : La suppression d'un élément va décaler les indices des éléments suivants, ce qui va amener la boucle à sauter des éléments qui n'ont pas encore été traités.
- Incohérences : Modifier la structure de la liste pendant son itération peut rendre le parcours incohérent, car ipairs s'attend à ce que les indices soient continus et non modifiés pendant l'itération. En gros, ça fout le bordel… donc à éviter absolument !

Ben alors comment on supprime pendant une boucle ?

Pour gérer la suppression d'éléments d'une liste pendant son parcours, l'approche consiste à utiliser une boucle `for` inversée :

```
for n = #mesEnnemis, 1, -1 do
    if mesEnnemis[n].y > hauteurEcran then
        table.remove(maListe, n)
    end
end
```

Cette méthode parcourt la liste de la fin au début, ce qui permet de supprimer des éléments en cours de route en toute sécurité. Un peu comme descendre d'une échelle à l'envers.

Notez l'ajout d'un 3e paramètre à notre `for` : `-1`. Cela indique à Lua d'aller dans le sens inverse de la liste. Ce paramètre est appelé le "pas" d'itération (step en anglais). Il est obligatoire quand on veut inverser le sens du `for`, et si vous l'oubliez la boucle ne fonctionnera pas.

3. Méthode avec `pairs`

La fonction `pairs` propose une approche différente pour parcourir les tables dont les clés peuvent être de n'importe quel type, pas seulement des nombres.

`pairs` permet de parcourir chaque paire clé-valeur d'une table, quelle que soit la nature de la clé.

Imaginons une table avec un mélange de clés numériques et de chaînes :

```
maTable = {["clé1"] = "valeur1", 2 = "valeur2", ["trois"] = "valeur3"}
```

Un parcours avec `ipairs` ne fonctionnerait pas car certains index ne sont pas numériques et ne se suivent pas. La boucle se terminerait ici immédiatement sans rien afficher ou exécuter.

Voici comment parcourir cette table avec `pairs` :

```
for k, v in pairs(maTable) do
    print(k, v)
end
```

Dans cette boucle :

- `k` représente la clé de l'élément courant dans la table.
- `v` est la valeur associée à cette clé.

L'utilisation de `pairs(maTable)` génère une paire clé-valeur pour chaque élément de la table, sans se soucier de l'ordre ou du type des clés.

Avantage de pairs

- `pairs` fonctionne avec n'importe quelle table Lua, permettant de parcourir des structures de données où les clés ne sont pas nécessairement des indices numériques ou sont non séquentielles.

Inconvénient de pairs

- Imprévisibilité de l'ordre : Contrairement à `ipairs` qui garantit un parcours séquentiel, l'ordre dans lequel `pairs` parcourt les éléments n'est pas défini et peut varier d'une exécution à l'autre. C'est en fonction de l'ordre en mémoire et non pas de l'index.

Tables et mémoire : La notion de "référence"

En Lua, les tables sont des structures de données dynamiques stockées dans la mémoire.

Lorsque vous créez une table, Lua alloue de l'espace mémoire pour elle. La variable qui représente cette table ne stocke pas les données directement, mais "pointe" vers ces données. On dit alors que la variable contient une "référence" (une adresse mémoire). Cela signifie que plusieurs variables peuvent référencer la même table si elles contiennent la même référence. Vous pouvez le vérifier en exécutant ce code :

```
dragon = {type = "Dragon", vie = 100, force = 50}
creatureRencontree = dragon

print(dragon)
print(creatureRencontree)
```

Que voyez-vous s'afficher dans la console ?

```
table: 0x55f80820ad40
table: 0x55f80820ad40
```

Vous ne voyez pas s'inscrire le contenu de votre table mais à la place son adresse en hexadécimal. Et constatez que `creatureRencontree` contient la même adresse que `dragon`.

Bien sûr vous verrez des adresses différentes sur votre ordinateur, et même à chaque fois que vous testerez ce code, car la mémoire change en permanence.

On continue cet exemple pour illustrer encore plus en détail ce concept important :

```lua
dragon = {type = "Dragon", vie = 100, force = 50}

creatureRencontree = dragon

-- Le dragon subit des dégâts dans une bataille
creatureRencontree.vie = creatureRencontree.vie - 30

print("Après la bataille :")
print("Le dragon a maintenant " .. dragon.vie .. " points de vie")
```

Dans cet exemple :

- Nous créons une table `dragon` pour représenter un dragon avec des attributs spécifiques comme la vie et la force.
- Pour l'exemple, la variable `creatureRencontree` est déclarée et pointe vers la même table que `dragon`, ce qui signifie que les deux référencent la même entité dans le jeu.
- Quand `creatureRencontree` subit des dégâts, la vie dans la table `dragon` est mise à jour, car les deux variables pointent vers la même table.

Les tableaux

Dans la plupart des langages de programmation, les tableaux sont des collections dont la taille est fixe, alors que les listes sont des collections dont la taille peut évoluer, donc plus flexibles. En Lua les tableaux sont des listes…

Tableaux à une seule dimension

Un tableau à une dimension en Lua est donc une liste. Vous pouvez y stocker une série d'éléments dans un ordre spécifique. Ces éléments peuvent être des valeurs (numériques, texte, booléens) mais aussi des tables (donc des listes, ou des tableaux) !

Exemple : Inventaire d'un Personnage

```lua
inventaire = {}
inventaire[1] = "épée"
inventaire[2] = "bouclier"
inventaire[3] = "potion"

-- Affichage de chaque élément de l'inventaire
for i = 1, #inventaire do
    print(inventaire[i])
end
```

Dans cet exemple, inventaire est un tableau contenant trois objets.

Tableaux à Deux Dimensions

Les tableaux à deux dimensions sont parfaits pour créer des cartes de jeux (maps), comme des donjons, où chaque cellule de la grille peut représenter une salle ou un mur.

Dans cet exemple, nous utiliserons 0 pour représenter un espace vide et 1 pour un mur :

```
map = {
    {1, 1, 1, 1, 1},
    {1, 0, 0, 0, 1},
    {1, 0, 1, 0, 1},
    {1, 0, 0, 0, 1},
    {1, 1, 1, 1, 1}
}

-- Affichage de la map
for ligne = 1, #map do
    for colonne = 1, #map[ligne] do
       -- Code pour afficher une case
       local case = map[ligne][colonne]
    end
end
```

Ici, `map` est un tableau à deux dimensions, où chaque élément est lui-même un tableau représentant une ligne de la carte. La boucle imbriquée nous permet de parcourir chaque cellule de la map et de traiter sa valeur, par exemple pour afficher sa représentation graphique..

La double boucle

Le principe de la double boucle en programmation, particulièrement lorsqu'il est appliqué à des tableaux à deux dimensions, est un outil puissant pour accéder et manipuler chaque élément d'une structure de données en grille, comme une carte de jeu.

Dans l'exemple de la map du donjon, nous utilisons une double boucle pour parcourir le tableau `map`, où chaque élément du tableau représente une ligne de la carte, et chaque élément de ces sous-tableaux représente une colonne à l'intérieur de cette ligne.

Première boucle : sur les lignes

La première boucle parcourt chaque élément (ligne) du tableau map. Chaque ligne est elle-même un tableau représentant les cellules horizontales (colonnes) de la map.

```
for ligne = 1, #map do
```

Ici, `ligne` est la variable qui s'incrémente à chaque itération de la boucle et représente l'indice de la ligne courante dans le tableau map. Et `#map` représente le nombre d'éléments de la 1ère dimension.

Deuxième boucle : sur les colonnes

À l'intérieur de la première boucle, une deuxième boucle est imbriquée pour parcourir chaque élément (colonne) de la ligne courante.

```
 for colonne = 1, #map[ligne] do
```

Dans cette boucle, `colonne` représente l'indice de la colonne courante dans la ligne `map[ligne]`. Cette structure permet d'accéder à chaque cellule individuellement, en utilisant les indices `[ligne][colonne]`.

Notez comment nous utilisons `#map[ligne]` pour obtenir le nombre de colonnes.

Voici une version décomposée pour mieux visualiser l'accès au tableau de chaque ligne :

```
for ligne = 1, #map do
    local laLigne = map[ligne]
    for colonne = 1, #laLigne do
       -- suite du code
```

Comment lire ce code :

- La première boucle parcourt les lignes du tableau `map`. en utilisant un compteur `ligne` qui sera utilisé pour récupérer chaque élément de la 1ère dimension du tableau.

- Pour chaque ligne, la deuxième boucle itère sur chaque colonne de cette ligne en utilisant une variable `colonne` qui sera utilisée elle aussi comme compteur pour récupérer chaque élément de la 2ère dimension du tableau.

- Le contenu d'une case de la map est obtenu via `map[ligne][colonne]`.

Si l'on décompose l'ordre dans lequel la double boucle s'exécute ça donne :

```
ligne = 1
colonne = 1 - donc accès à map[1][1]
colonne = 2 - donc accès à map[1][2]
colonne = 3 - donc accès à map[1][3]
colonne = 4 - donc accès à map[1][4]
colonne = 5 - donc accès à map[1][5]
ligne = 2
colonne = 1 - donc accès à map[2][1]
colonne = 2 - donc accès à map[2][2]
colonne = 3 - donc accès à map[2][3]
colonne = 4 - donc accès à map[2][4]
colonne = 5 - donc accès à map[2][5]
ligne = 3
colonne = 1 - donc accès à map[3][1]
Etc.
```

Comment apprendre un autre langage de programmation

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

Voici par exemple un code Lua comparé à un code similaire en C++ :

En Lua

```lua
local valeur = 3
local titre = "Gamecodeur"

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

local resultat = Addition(10, 5)

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

En C++

```cpp
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 !";
}
```

Bien sûr chaque langage à sa syntaxe et ses spécificités mais le principe reste globalement le même et cet exemple vous le prouve.

Vous constatez par exemple qu'en C++ apparaît le mot clé "`int`" qui permet de spécifier le type de la variable (ici un entier) et que la ponctuation est différente (les accolades et les points virgule). De même, afficher une trace ne se fait pas de la même manière.

Mais vous pouvez reconnaître les fondamentaux : variables, fonctions, expressions, conditions…

Lua reste un des langages les plus simples à apprendre

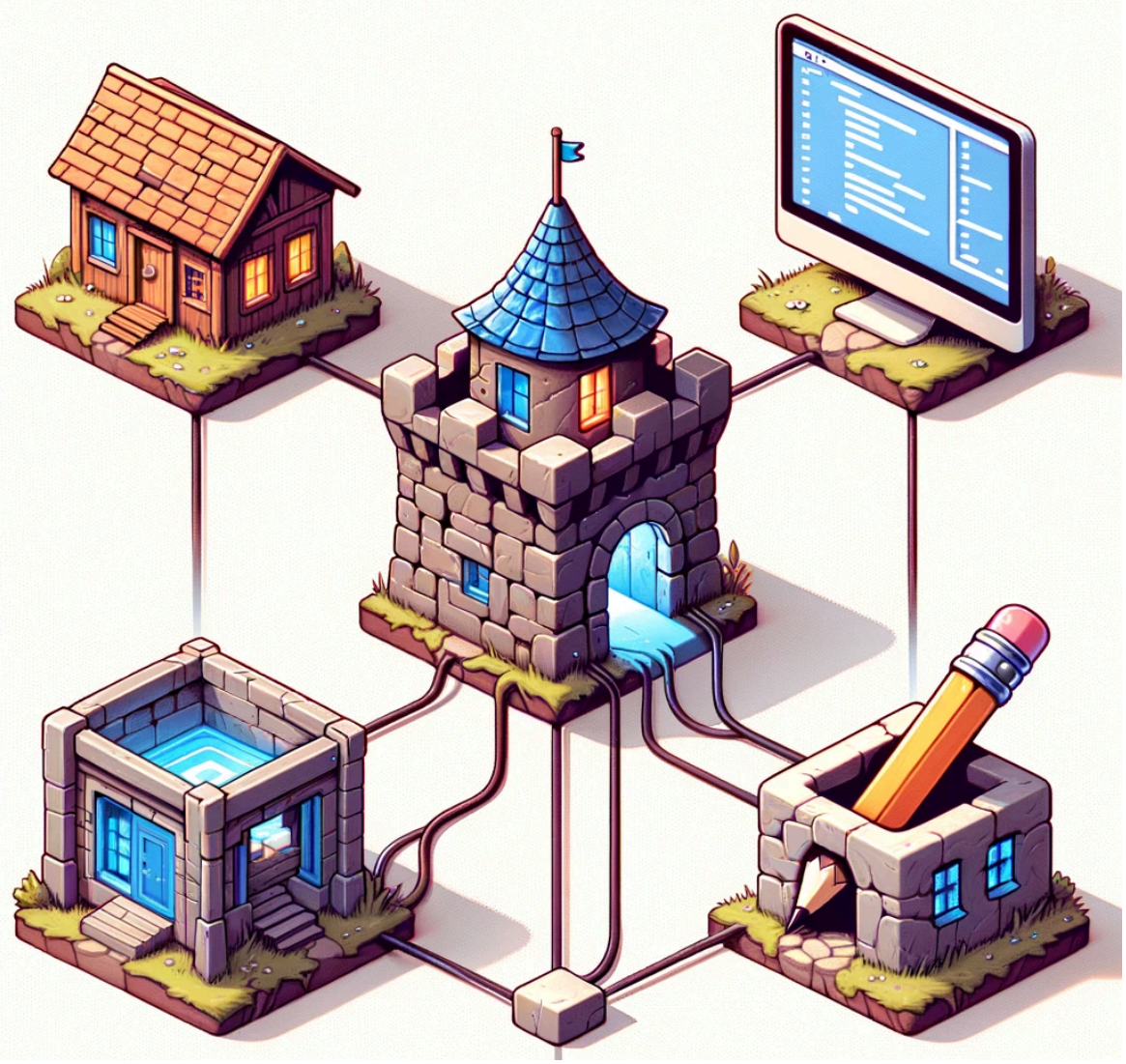
Les autres différences viendront essentiellement de la manière de gérer les tableaux et les listes, ainsi que dans la programmation objet.

Et bien entendu, les outils et les moteurs de jeu changent en fonction du langage de programmation que ce moteur utilise. La complexité va venir essentiellement de l'outil, pas du langage.

Par exemple en C++, qui est un langage qu'on va "compiler", la notion de compilation est complexe et apporte de nombreuses 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 progrès rapides. En commençant directement par le C++ ou le C# vous risquez de vous décourager devant la montagne de complexité de ces langages !

Fondamental 5 : Objets et modularité



La programmation orientée objet (POO) est une méthode de programmation qui permet de structurer son code autour d'entités appelées "objets". Ces objets regroupent des variables et des fonctions associées, facilitant ainsi l'organisation et la réutilisation du code.

Mais je dois commencer par une mise au point :

Le langage Lua n'est pas un langage orienté objet et je ne m'aventurerai avec vous dans les méandres des techniques chelous qu'on trouve sur le net pour "simuler" des objets en Lua.

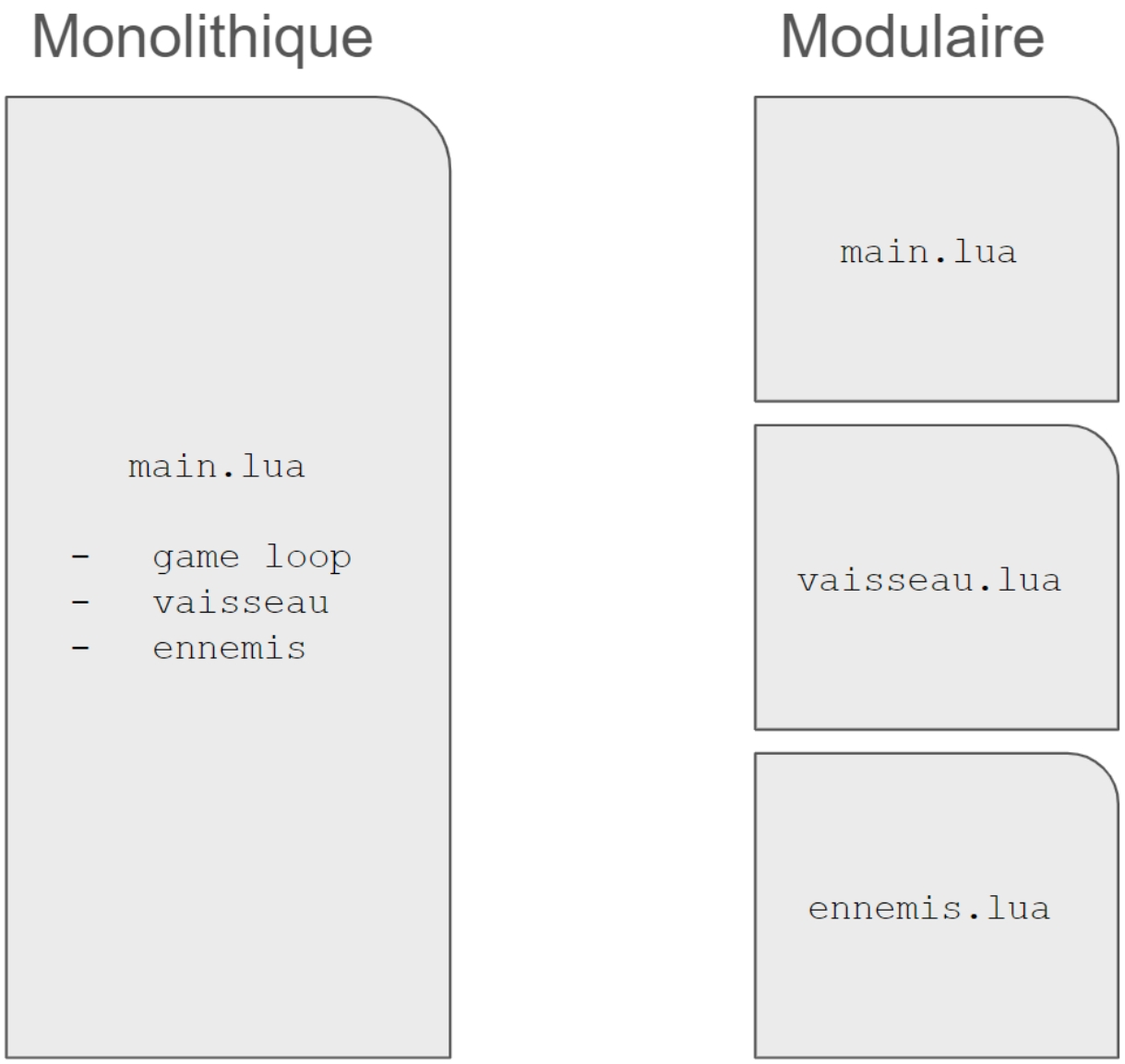
Les tables sont largement suffisantes pour un débutant. Elles permettent de décrire des données structurées proches des objets, et si l'on le souhaite, d'y associer des fonctions.

Par contre, en Lua, nous avons un autre concept : les modules. C'est un concept super puissant pour rendre notre code "modulaire" (comme son nom l'indique).

Dans cette section, nous allons apprendre à créer un module en Lua et à l'importer dans un projet.

C'est quoi un module ?

Un module en Lua est un script Lua, dans un fichier séparé, qui regroupe des fonctionnalités spécifiques, et réutilisable dans différentes parties de notre programme. Cela permet donc de séparer son code en plusieurs fichiers, en plus du "`main.lua`", pour le rendre plus lisible et mieux organisé. Regardez la différence quand on utilise des modules :



Vous comprendrez rapidement l'utilité des modules avec l'expérience et en pratiquant avec des exemples. Si vous débutez, oubliez les modules au départ.

Voici les étapes pour créer et utiliser un module :

Création d'un module

Pour créer un module, commencez par créer un nouveau fichier `.lua`.

À l'intérieur de ce fichier, vous allez créer une table qui représentera le module.

Dans cette table vous allez définir des fonctions, des variables et tout autre élément que vous souhaitez exposer à d'autres parties de votre programme. Les autres variables pourront être locales pour les garder privées.

```lua
local monModule = {}

monModule.variableExposee = 10
local variablePrivee = 99

function monModule.maFonction()
    print("Fonction dans monModule")
end

return monModule -- Important : toujours penser à retourner la table en fin de module
```

Dans cet exemple, on constate que `monModule` est tout simplement une table Lua qui contient toutes les fonctions et variables que vous souhaitez exporter. La table est créée, des variables et des fonctions y sont ajoutées, et la table est retournée via "`return`". Un module peut aussi contenir des variables locales (= privées) qui seront donc invisibles depuis les autres fichiers Lua.

Importation d'un module

Pour utiliser le module dans un autre script, vous devez l'importer à l'aide de la fonction `require`.

Par exemple, si vous souhaitez utiliser `monModule` dans votre `main.lua` :

```lua
-- main.lua
local monModule = require("monModule")

monModule.maFonction() -- Appelle la fonction définie dans monModule
print(monModule.variableExposee) -- Affiche une variable définie dans monModule
```

Un seul module en mémoire

L'utilisation de `require` charge le module et exécute son code UNE SEULE FOIS même si d'autres "require" du même module existent dans d'autres modules. Le module n'est chargé qu'une seule fois, peu importe le nombre de fois où il est requis. S'il contient du code (en dehors des fonctions), ce code n'est donc exécuté que lors du 1er `require`.

C'est là toute la puissance des modules.

Exemple :

1er require : charge le module, exécute son code et stocke sa référence.
2ème require : récupère la référence du 1er require sans rien charger ni exécuter

Note : Les modules sont un concept avancé, si vous trouvez cela confus c'est normal. Vous aurez l'occasion de les pratiquer dans le futur, quand vous aurez un bon niveau en Lua. Je souhaitais tout de même les aborder pour que la graine pousse dans un coin de votre tête.

Formation à Löve2D

Pour développer un jeu vidéo en 2D, l'utilisation d'un framework dédié est essentielle. Love2D est un excellent choix pour cela. Un framework 2D est une sorte de bibliothèque de fonctions conçue spécifiquement pour faciliter le développement de jeux vidéo.

Love2D est un framework open-source permettant de créer des jeux 2D avec le langage Lua. Il abstrait les complexités du développement de jeux en fournissant des fonctions faciles à utiliser pour les opérations courantes dans le développement de jeux, telles que le dessin de primitives graphiques ou d'images à l'écran, la manipulation des entrées clavier, gamepad, ou souris, et la gestion des ressources audio et graphiques.

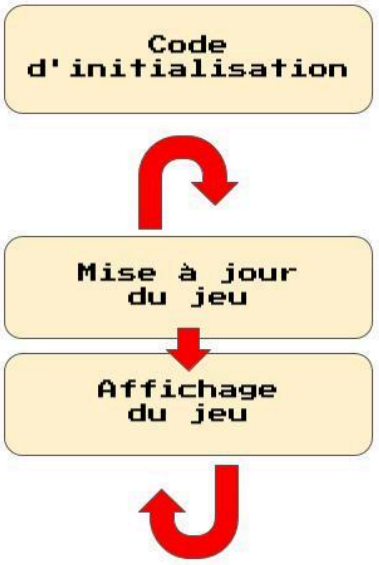
Comment un jeu vidéo est-il vivant ?

Un jeu vidéo est une série d'opérations, de calculs et d'affichages, exécutés à chaque frame. Une "frame" est une seule image générée par le jeu à un instant donné, et le taux de "frames par seconde" (FPS) indique combien de ces images sont produites chaque seconde.

Ces opérations sont réalisées dans le cadre de la boucle de jeu, ou "Game Loop". Elle agit comme un cœur qui bat plusieurs fois par seconde. Elle insuffle la vie à votre programme de jeu vidéo.

Visualisez le cœur humain : il bat continuellement pour permettre l'exécution de mécanismes physiologiques dans votre corps. De manière similaire, la "Game Loop" fait fonctionner votre jeu, en exécutant une suite d'opérations à chaque cycle.

Voici le fonctionnement typique de la boucle de jeu :



Au démarrage du programme :

- **Initialisation :** Au démarrage, le jeu exécute un bloc de code d'initialisation, qui peut se situer hors de toute fonction ou au sein d'une fonction spécifique prévue par le framework utilisé. Cette étape prépare le terrain en configurant les paramètres essentiels du jeu.

A chaque frame :

- **Mise à jour de l'état du jeu :** Cela peut inclure le déplacement d'un personnage, la mise à jour de son animation, la vérification des collisions, etc.
- **Affichage du jeu (rendu graphique) :** Après chaque mise à jour, le jeu efface les visuels précédents et redessine entièrement la scène en fonction du nouvel état du jeu.

En d'autres termes, à chaque cycle, le jeu recalcule ce qu'il est nécessaire de recalculer et efface puis redessine l'écran pour refléter les dernières interactions et changements d'état. Oui, vous lisez bien : on efface tout et on recommence à chaque frame !

Quand j'avais 13 ans et que j'ai lancé la première cassette de programme dans mon Amstrad CPC, j'ai tout de suite cherché à comprendre comment le jeu pouvait continuer à fonctionner alors que la cassette ne tournait plus ! (oui, en 1984, je ne connaissais que les magnétoscopes, et niveau technologie on en était à l'âge de pierre...).

J'avais cru au départ, que comme pour un magnétoscope, l'image était enregistrée sur la cassette… Mais je ne comprenais pas comment il était possible de faire changer de direction à mon personnage. Car si l'image était pré-enregistrée, il aurait été impossible de faire cela !

En réalité, bien entendu, la cassette contenait un programme qu'on chargeait en mémoire (en plusieurs minutes comme on le faisait avec les modems dans les années 90) puis on l'exécutait.

Mine de rien, cette réflexion m'a immédiatement permis de comprendre le secret du fonctionnement d'un jeu vidéo : la Game Loop !

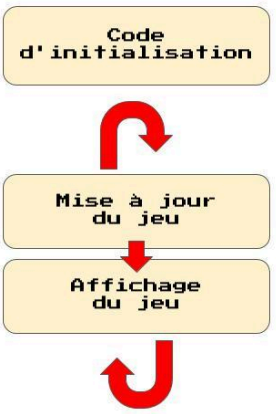
Explique encore, j'ai pas compris !

Rentrons un peu plus dans le détail, avec un exemple simplifié et en pseudo code :

```
Initialisation :
    Position X du personnage = 10
    Position Y du personnage = 100
Fin de Initialisation

Mise à jour :
    Si le joueur appuie sur la touche "droite" alors
        Ajoute 1 à la position X du personnage
    Fin de Si
Fin de Mise à jour

Affichage :
   Efface l'écran
   Affiche le décor
   Affiche le personnage à sa position X et Y
Fin de Mise à jour
```



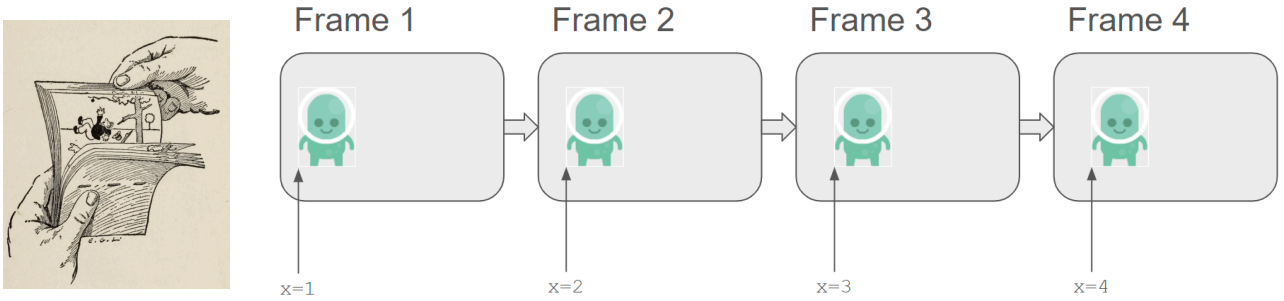
Les sections "Mise à jour" et "Affichage" vont ensuite se répéter à l'infini.

Dans cet exemple, le framework va donc exécuter le code d'initialisation, puis à l'infini un enchaînement de Mise à jour (update) et Affichage (draw).

Si le joueur presse la touche "flèche droite" :

- La variable X représentant la position du personnage va changer (elle augmente de 1).
- Ce changement de position sera visible au moment de l'affichage qui suivra dans la frame.

Ceci donne l'illusion du mouvement, exactement comme dans un dessin animé ou un "flip book" :



Si vous comprenez cela, vous comprendrez beaucoup de choses. Notamment qu'un jeu vidéo n'est rien d'autre qu'une représentation graphique de valeurs. Ce qui compte, c'est comment vous modifiez ces valeurs entre chaque frame.

Note : Même quand vous utilisez un moteur comme Unity ou Unreal, le principe est exactement le même, en 2D comme en 3D : l'écran est effacé et ré-affiché à chaque frame. Seulement ce n'est pas vous qui êtes en charge d'afficher chaque frame, le moteur le fait à votre place.

Charger et afficher une image

Pour charger et afficher une image en Love2D, vous devez d'abord charger le fichier image depuis votre programme, puis le dessiner sur l'écran dans la fonction `love.draw()`. Les formats d'image que vous devez principalement connaître pour Love2D sont :

- JPEG

⇒ Privilégiez le format JPEG pour les images de fond, car ce format ne permet pas d'avoir des pixels transparents. L'image recouvre toute la surface.

- PNG

⇒ Utilisez systématiquement PNG pour vos éléments en mouvement (sprites), car vous pourrez avoir des pixels transparents et donc afficher vos éléments sur des décors.

IMPORTANT : Pour éviter toute confusion lors du développement de jeux avec Love2D ou en général pour toutes les tâches informatiques quand on est pro, il est crucial d'afficher les extensions de fichiers dans Windows. Sans cette visibilité, il peut être difficile de distinguer les types de fichiers différents, par exemple si vous avez des images en `.jpeg` et `.png`, ainsi que des scripts Lua (`.lua`) et des fichiers texte (`.txt`), etc.

L'affichage des extensions vous aide à identifier rapidement le type de fichier avec lequel vous travaillez, évitant ainsi les erreurs comme charger le mauvais fichier ou mal interpréter le contenu d'un fichier. Pour activer l'affichage des extensions de fichiers dans Windows, vous pouvez modifier les options d'affichage dans l'explorateur de fichiers : menu "Afficher" puis "Afficher / Extensions de noms de fichiers". Impossible de survivre sans cela quand on programme !

Voici comment procéder pour charger et afficher une image PNG ou JPEG dans Love2D :

1) **Chargez l'image :** utilisez la fonction `love.graphics.newImage` pour charger votre image depuis votre programme. Par exemple, si vous avez une image `monImage.png`, vous pouvez la charger comme suit :

```
monImage = love.graphics.newImage("monImage.png")
```

2) **Afficher l'image :** Dans la fonction `love.draw()`, utilisez la fonction `love.graphics.draw` pour dessiner l'image préalablement chargée. Vous pouvez spécifier la position en x et y où l'image doit être affichée :

```
function love.draw()
    love.graphics.draw(monImage, x, y)
end
```

Bon, tout ça c'est théorique, donc passons à la pratique…

Votre premier projet Love2D

Note : L'image d'un personnage, entourée d'un cadre blanc à visée pédagogique, et livrée dans le bonus numérique fourni avec ce guide. Si vous voulez utiliser votre propre image choisissez un fichier PNG de petite taille.

Pour créer un projet Love2D, suivez ces étapes :

- Créez un dossier comme expliqué au début de ce guide.
- Ouvrez-le avec Visual Studio Code.
- Créez un nouveau fichier dans le projet et enregistrez-le sous le nom "main.lua".

Maintenant, pour ajouter une image dans un sous-dossier "images", suivez ces étapes :

- Créez un dossier nommé "images" à l'intérieur du répertoire de votre projet Love2D.
- Placez l'image nommée "personnage.png" à l'intérieur du dossier "images".

Maintenant, tapez ce code (en pleine conscience, pour essayer d'en comprendre chaque ligne) :

```
local image
local largeur, hauteur

function love.load()
  image = love.graphics.newImage("images/personnage.png")
end

function love.draw()
  love.graphics.draw(image, 0, 0)
end
```

Explication ligne par ligne :

- `local image` : Déclare une variable locale nommée 'image' pour stocker l'image.

- `local largeur, hauteur` : Déclare deux variables locales pour stocker la largeur et la hauteur de la fenêtre.

- `function love.load()` : Cette fonction est appelée une fois au début du jeu. Elle charge l'image à partir du fichier "personnage.png" situé dans le dossier "images". La variable `image` contiendra alors la référence de l'image.

- `function love.draw()` : Cette fonction est appelée à chaque frame, juste après que l'écran ait été effacé automatiquement par Love2D. Elle dessine dans la fenêtre de jeu l'image chargée aux coordonnées (0,0) de la fenêtre à l'aide de la fonction `love.graphics.draw()` à laquelle on passe la référence de l'image et les coordonnées souhaitées, exprimées en pixels.

Déplacer une image

Pour déplacer votre personnage dans les quatre directions en utilisant les touches fléchées du clavier, vous devez introduire une logique à chaque frame pour détecter ce que fait le joueur et refléter ses actions sur les coordonnées de votre personnage :

```
local image
local x, y
local vitesse = 120 -- Définit la vitesse de déplacement du personnage

function love.load()
  image = love.graphics.newImage("images/personnage.png")
  x = 0  -- Position initiale en x
  y = 0  -- Position initiale en y
end

function love.update(dt)
  if love.keyboard.isDown("right") then
    x = x + vitesse * dt
  end
  if love.keyboard.isDown("left") then
    x = x - vitesse * dt
  end
  if love.keyboard.isDown("up") then
    y = y - vitesse * dt
  end
  if love.keyboard.isDown("down") then
    y = y + vitesse * dt
  end
end

function love.draw()
  love.graphics.draw(image, x, y)
end
```

Explication du code :

Variables x et y : Ces variables stockeront la position actuelle du personnage sur l'écran. Elles seront mises à jour chaque fois que l'utilisateur appuiera sur une touche fléchée.

Vitesse : Cette variable définit la rapidité du déplacement du personnage, en pixels / secondes.

`love.update(dt)` : Cette fonction est appelée à chaque frame avant de dessiner. Le paramètre `dt` est le temps écoulé depuis le dernier appel à `update`, ce qui permet de déplacer le personnage de manière fluide et indépendante de la vitesse de la frame. On va en reparler.

`love.keyboard.isDown("...")` : Cette fonction vérifie si une touche spécifique est pressée. Selon la touche pressée, la position `x` ou `y` du personnage est ajustée en conséquence.

`love.graphics.draw(image, x, y)` : Dessine l'image à la position mise à jour.

En utilisant ce code, votre personnage se déplacera vers la droite si vous appuyez sur la flèche droite, vers la gauche pour la flèche gauche, vers le haut pour la flèche du haut, et vers le bas pour la flèche du bas.

Mais vous noterez un calcul bizarre au moment de changer les coordonnées du personnage :

```
x = x + vitesse * dt
```

Mais c'est quoi ce "dt" ?

Le "`dt`" (delta time en anglais) dans l'expression `x = x + vitesse * dt` représente le delta de temps, c'est-à-dire le temps écoulé entre deux frames successives dans la boucle de jeu de Love2D. Utiliser `dt` permet d'uniformiser la vitesse des déplacements, indépendamment du taux de rafraîchissement de l'écran ou des performances de l'ordinateur.

Lorsque vous multipliez la vitesse par `dt`, vous ajustez le déplacement du personnage pour qu'il soit proportionnel au temps réel, et non au nombre de frames. Ainsi, peu importe si votre jeu s'exécute plus rapidement ou plus lentement sur différents appareils, le personnage se déplacera toujours à la même vitesse en termes de distance parcourue par seconde, assurant une expérience de jeu uniforme.

Exemple : Dans un écran à 60Hz, le taux de rafraîchissement est de 60 images par seconde. Cela signifie que chaque frame est affichée pendant 1/60ème de seconde. Le "`dt`" (delta time) dans ce contexte sera approximativement de 1/60, car le jeu, calé sur le frame rate, s'efforce de mettre à jour et de redessiner une nouvelle frame à chaque 1/60ème de seconde.

Si nous reprenons l'exemple du code de déplacement avec dt :

```
 x = x + vitesse * dt
```

Et disons que la vitesse est de 120 pixels par seconde comme dans notre exemple, alors à chaque frame sur un écran 60Hz le personnage se déplacera de :

```
120 pixels/sec * (1/60) sec/frame = 2 pixels par frame
```

En effet, sur un écran à 60Hz, le `dt` (delta time) est d'environ 1/60ème de seconde, soit environ 0.01667 secondes, car il y a 60 images ou frames par seconde.

Ainsi, même si le taux de rafraîchissement change (par exemple sur un écran 144Hz où dt serait 1/144), en utilisant `dt` le personnage se déplacera toujours de 120 pixels par seconde, car le calcul compensera automatiquement la différence de taux de rafraîchissement.

En résumé : multipliez toujours par `dt` toute valeur liée à un déplacement ou à un calcul de temps.

Pixels et système de coordonnées

Même si cela peut paraître trivial pour certains, je préfère revoir cette notion de base.

Un écran de jeu est composé de **pixels**. Leur taille et leur nombre dépendent de la **résolution** de l'ordinateur au moment où l'écran est affiché. Une résolution est donc exprimée en pixels.

En premier le nombre de pixels horizontaux, en deuxième le nombre de pixels verticaux.

Un peu de vocabulaire anglais :

```
largeur : width
hauteur : height
```

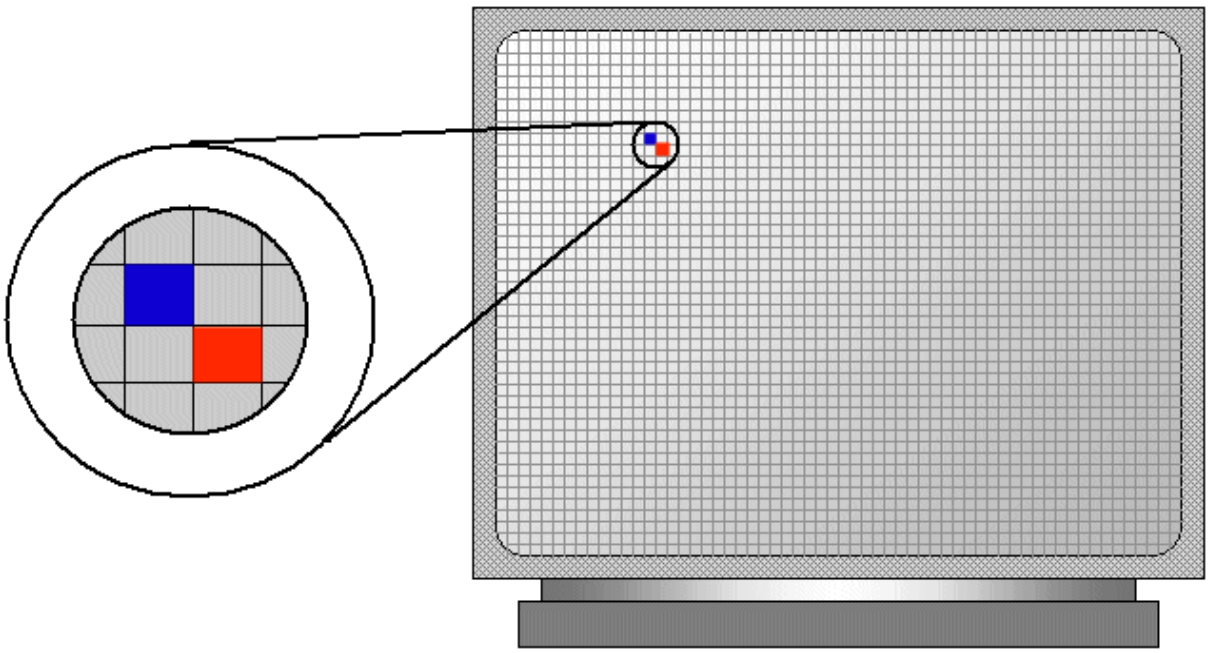
Moyen mnémotechnique retenir le sens et l'orthographe :

- ***h**eight ; commence par la lettre **h** comme **h**auteur, se termine par "j'ai acheté" ("G" "H" "T")*
- *widt**h** : le **h** est à la fin ("G" "T" "H").*

Exemple de résolution : 1024x768 (on dit alors "1024 par 768"), ce qui signifie 1024 pixels horizontaux (width) , et 768 pixels verticaux (height).

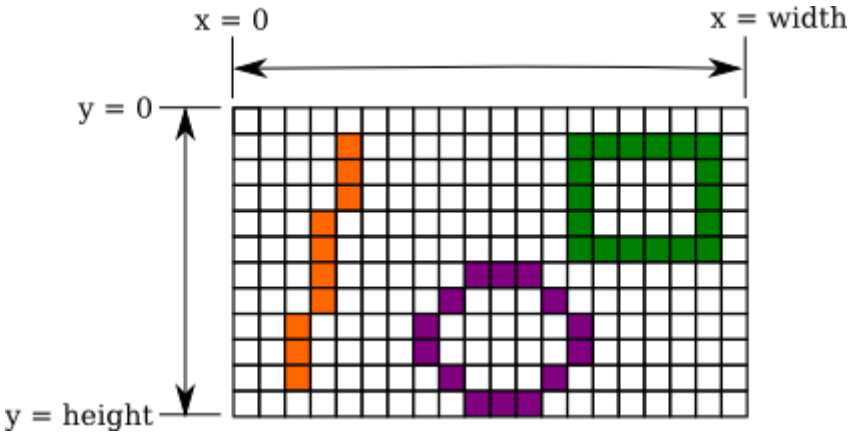
C'est quoi un pixel ?

Un pixel, ou élément d'image, est la plus petite unité de mesure affichable sur un écran d'ordinateur ou mobile, et c'est lui qui détermine la plus petite portion de l'écran pouvant être contrôlée individuellement. Chaque pixel peut afficher une couleur à la fois, et l'ensemble des pixels d'un écran travaille conjointement pour former l'image complète que vous voyez à l'écran.



Les pixels, en programmation, sont la plupart du temps numérotés à partir de 0, de gauche à droite et de haut en bas et la position horizontale est donnée en premier. On parle aussi de **colonnes** et de **lignes** d'affichage.

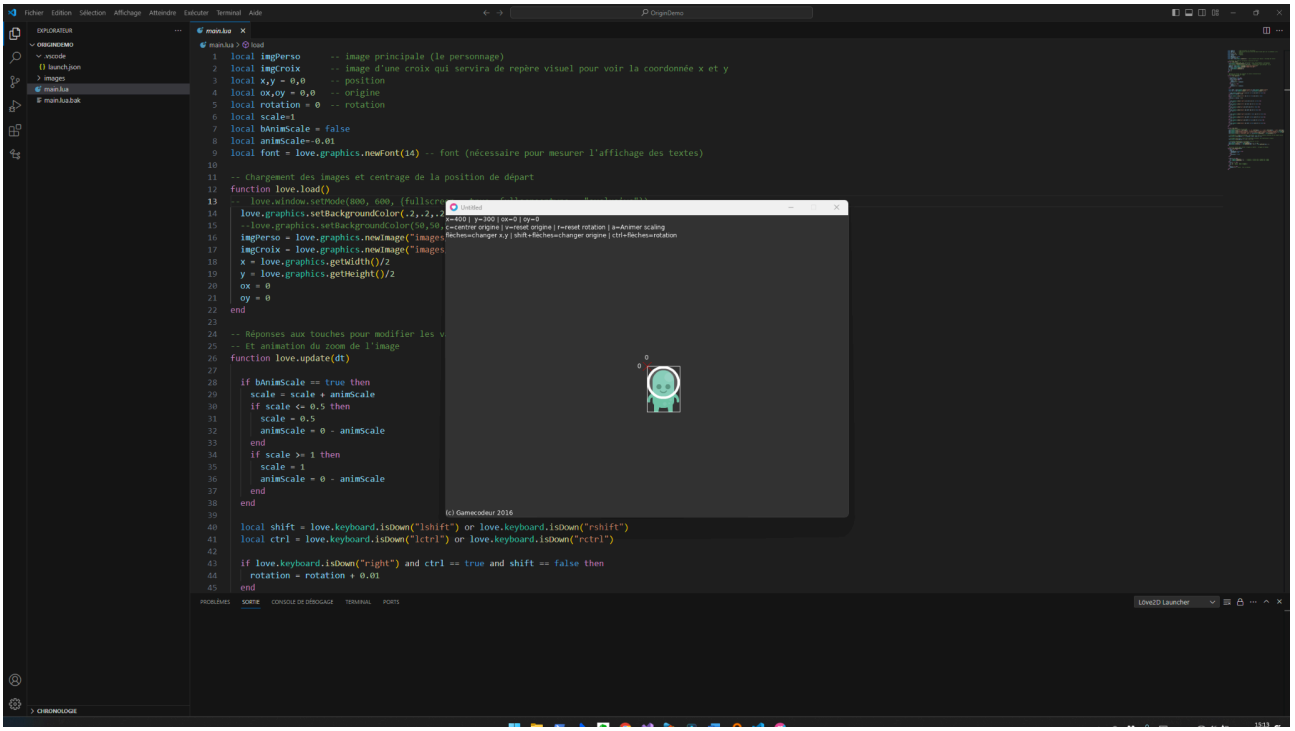
La coordonnée la plus en haut à gauche est donc 0,0 :



(source : http://math.hws.edu/javanotes/c6/pixel-coordinates.png)

L'écran peut être "fenêtré" (windowed) ou plein écran (full screen).

A noter qu'en mode fenêtré, l'écran du jeu possède une **barre de titre** (title bar) et utilise la taille des pixels du système en cours (windows, linux, OSX…). La taille de la fenêtre, elle, correspond à la résolution du jeu, par exemple 800x600 pixels par défaut pour un jeu Love2D. La fenêtre du jeu peut être déplacée comme une application en cliquant sur la barre de titre.



Dans la plupart des frameworks de programmation vous avez accès à des fonctions pour :

- Changer la taille de la fenêtre de jeu
- Obtenir la taille de la fenêtre de jeu
- Passer en plein écran (ou inversement)

Par exemple, avec Love2D, pour connaître la hauteur (height) actuelle de l'écran du jeu utilisez :

```
local height = love.graphics.getHeight()
```

Pour connaître la largeur (width) actuelle de l'écran du jeu utilisez :

```
local width = love.graphics.getWidth()
```

Conseil pour les débutants : Travaillez avec la résolution par défaut de Love2D et ne passez pas le jeu en plein écran. Cela vous simplifiera grandement la vie.

Les coordonnées d'affichage d'une image

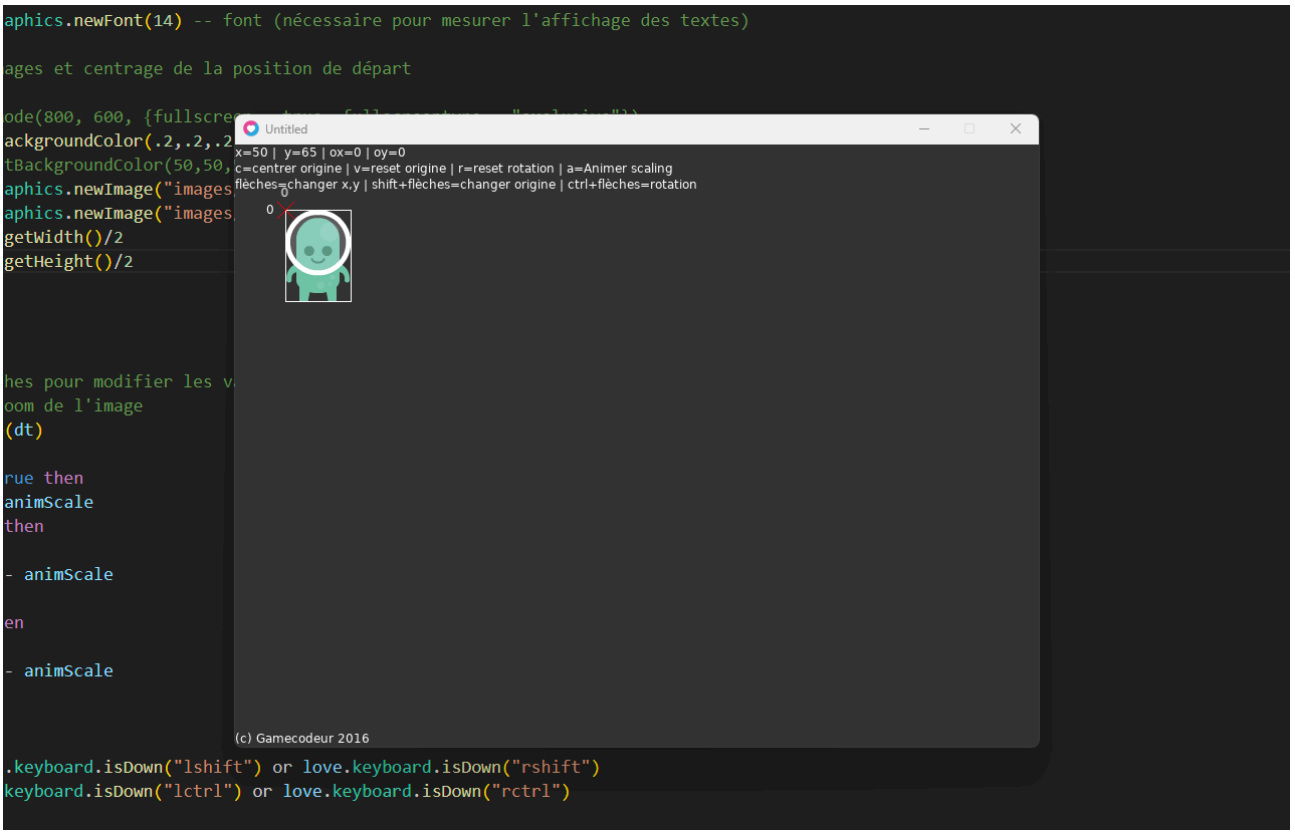
Pour placer une image à un endroit spécifique sur l'écran dans votre jeu, il est nécessaire de spécifier ses coordonnées.

Ces coordonnées sont définies par deux valeurs en pixels :

- Une pour l'axe horizontal (x)
- Une pour l'axe vertical (y).

Ainsi, une coordonnée 50,65 indique que l'image doit être positionnée à 50 pixels du bord gauche de l'écran et à 65 pixels de son bord supérieur.

Voilà ce que ça donne sur un écran de feu fenêtré, en 800x600 :



Quand on indique que l'image doit être positionnée à 50,65, il est crucial de comprendre à quel point précis de l'image correspond à ces coordonnées. Ce point de référence est ce qu'on appelle l'"origine" de l'image, un concept fondamental pour déterminer comment l'image sera alignée par rapport aux coordonnées fournies.

L'origine d'affichage d'une image

Dans la majorité des frameworks dédiés au développement de jeux en 2D, l'origine d'une image est définie par son coin supérieur gauche, correspondant à la position (0,0). Mais que signifie exactement cette origine en (0,0) ?

Cela indique que la position où l'image sera affichée sur l'écran est déterminée en référence à son coin supérieur gauche.

Pourquoi utilise-t-on (0,0) comme référence ?

- Visualisez l'image comme si c'était un petit écran doté de sa propre grille de pixels. Supposons que notre image ait une taille de 66x92 pixels, ce qui signifie qu'elle a une largeur de 66 pixels et une hauteur de 92 pixels.

- Si nous envisageons cette image comme un écran, le pixel situé le plus à gauche en haut correspond au point d'origine : la colonne 0, ligne 0.

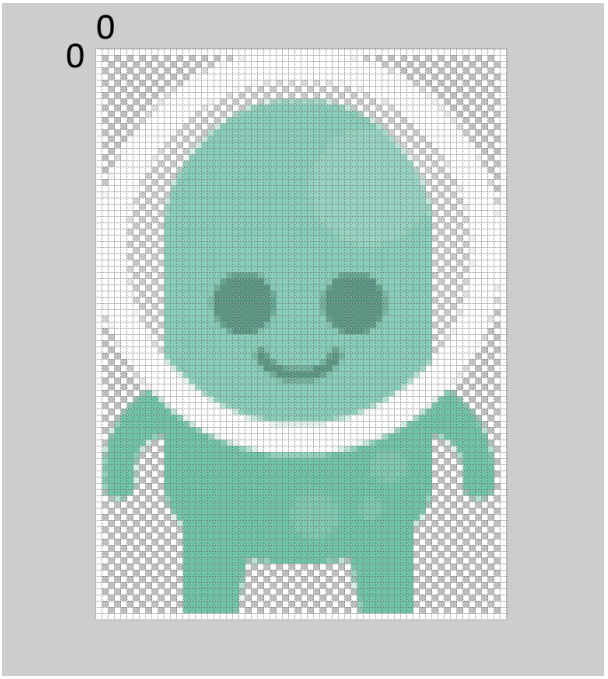
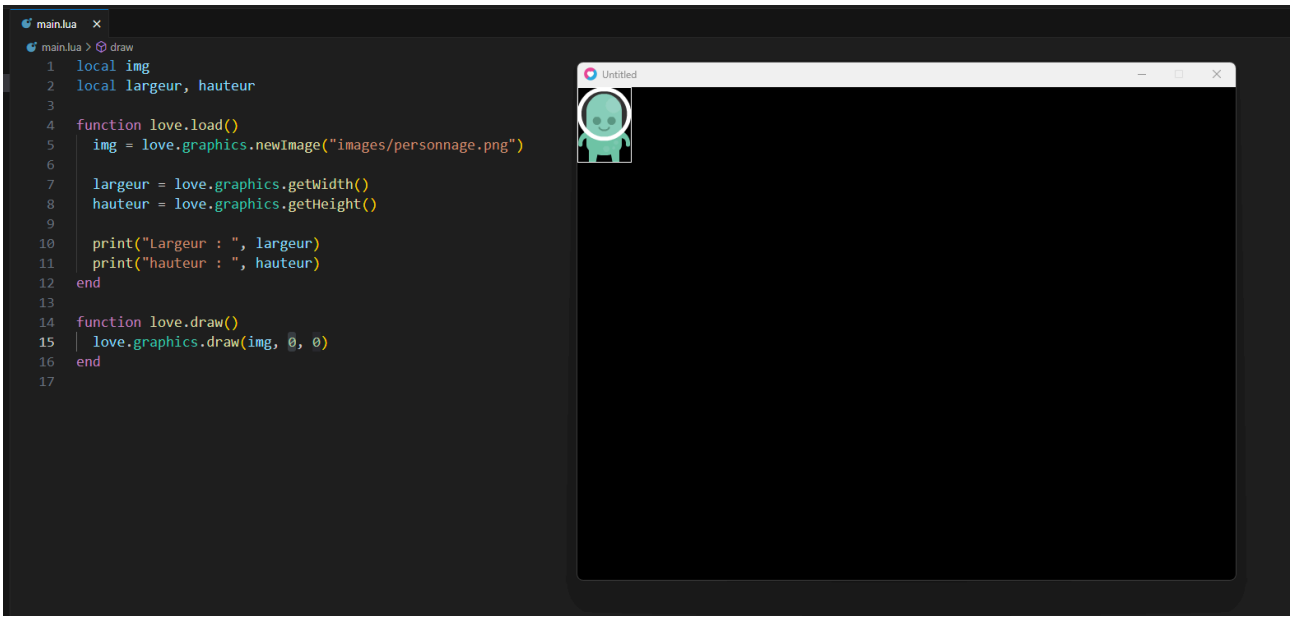


Image de 66x92 pixels, et son coin supérieur gauche à 0,0

- Le pixel situé le plus à droite de l'image sera le 65ème horizontalement (car on compte à partir de 0, donc 0 à 65 fait 66 pixels), et le pixel le plus en bas sera le 91ème verticalement (de nouveau, de 0 à 91 pour obtenir 92 pixels).

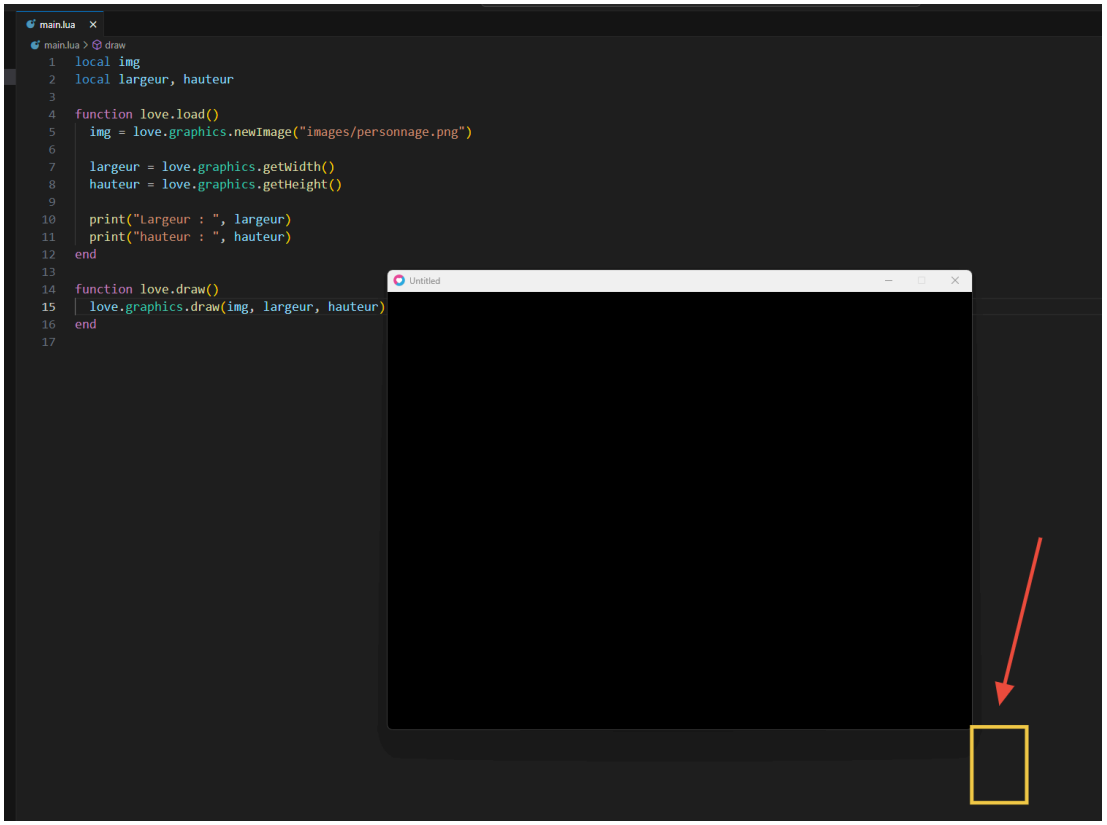
Démonstration avec l'origine 0,0

Pour afficher l'image dont l'origine est 0,0 (par défaut donc) tout en haut à gauche de l'écran, rien de plus simple : on l'affiche aux coordonnées 0,0 :



Mais pour l'afficher tout en bas à droite ?

Si on l'affiche à 800,600 alors que notre fenêtre de jeux mesure 800x600, elle sera hors écran :



On doit donc se livrer à un calcul simple :

- La coordonnée horizontale doit être la largeur de l'écran "moins" la largeur de l'image.
- La coordonnée verticale doit être la hauteur de l'écran "moins" la hauteur de l'image.

On savait déjà obtenir la largeur et la hauteur de l'écran (voir plus haut) mais pour l'image c'est comme ceci :

Obtenir la largeur :

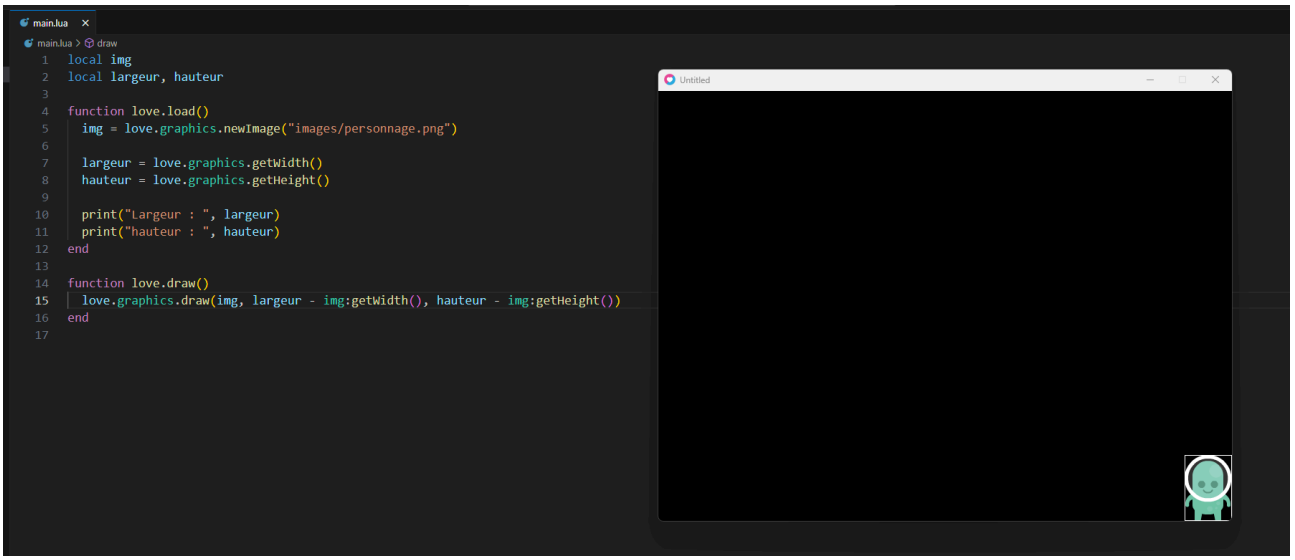
```
largeurImage = img:getWidth()
```

Obtenir la hauteur :

```
hauteurImage = img:getHeight()
```

`img` étant un exemple de nom de variable qui a été utilisé pour charger l'image avec *love.graphics.newImage*. On verra ça plus en détail dans les chapitres suivants.

Cela donne alors ça (le code est affiché à gauche) :



Décaler l'origine de l'image

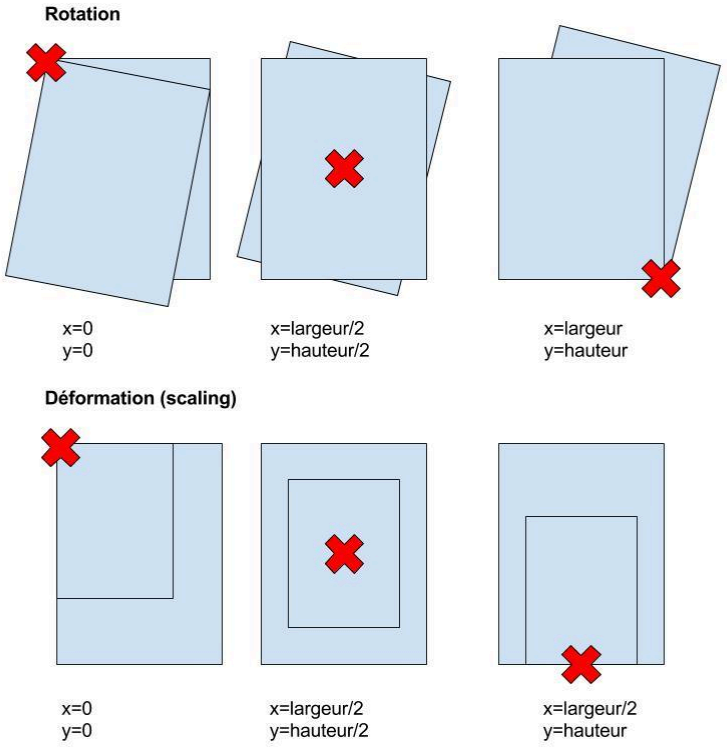
Si l'on ne souhaite pas que l'origine de l'image soit son coin supérieur gauche (0,0) alors on peut facilement appliquer un décalage à l'origine d'affichage. On parle alors d'"offset" (décalage).

Mais pourquoi changer l'origine de l'image ?

Plusieurs raisons possibles, voici les 2 principales qui justifient de changer l'origine d'une image :

1. Vous avez besoin de déformer l'image (scaling)
2. Vous avez besoin d'appliquer une rotation à l'image

Pour illustrer tout ça, je vous ai matérialisé ici plusieurs configurations possibles en utilisant la métaphore d'une feuille de papier. Voyez l'impact sur la rotation d'une image. La croix rouge représente l'origine, et je montre comment une feuille va tourner ou se déformer en fonction de celle-ci :



Comment changer l'origine avec Love2D

Pour changer l'origine d'affichage d'une image il faut utiliser une version de **draw** avec plus de paramètres, car l'origine est en 7e et 8e paramètre (ici ox et oy) :

```
love.graphics.draw(img, x, y, rotation, 1, 1, ox, oy)
```

Note : En 5e et 6e position on a la déformation (scaling). Il s'agit d'un facteur, donc 1 est la taille normale, 2 serait le double et 0.5 la moitié… On donne la déformation horizontale et la déformation verticale, donc 2 valeurs (ici 1,1 pour une taille normale). Si on donne une valeur négative, on obtient un effet miroir.

Si on veut déplacer l'origine en bas au centre ça va donc donner :

- pour le paramètre ox : la largeur de l'image divisée par 2 (pour obtenir le centre de l'image)
- pour le paramètre oy : la hauteur de l'image (pour obtenir le bas de l'image)

```
ox = image:getWidth() / 2
oy = image:getHeight()
```

Raison 2 : la rotation

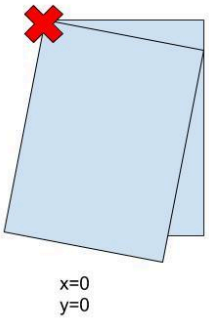
Si vous appliquez une rotation à votre image, elle va "tourner" autour de son origine. C'est très rarement l'effet voulu, on souhaite plutôt qu'elle tourne autour de son centre.

Pour expérimenter, tenez un post-it entre vos doigts, dans le coin en haut à gauche, et essayez de faire tourner cette feuille. Elle va tourner autour de vos doigts.

Dessinez maintenant un vaisseau spatial au centre de cette feuille, et imaginez que vous voulez le faire tourner sur lui même, où devriez-vous mettre vos doigts ? Au centre de la feuille bien sûr.

C'est le même principe avec l'origine de l'image.

Ici l'effet que l'on veut éviter :



L'image tourne par rapport à son origine, ici 0,0

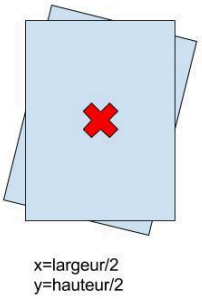
Pour résoudre le problème, nous devons changer l'origine de l'image en appliquant ce calcul :

- Origine x = largeur de l'image divisée par 2
- Origine y = largeur de l'image divisée par 2

Soit en code, pour les valeurs de ox et oy :

```
ox = image:getWidth() / 2
oy = image:getHeight() / 2
```

On utilise alors ces valeurs en 7e et 8e paramètres de draw (voir plus haut), et on obtient l'effet souhaité : l'image tourne autour de son centre.



Faire tourner une image

Pour appliquer une rotation à votre personnage en appuyant sur les touches Q et D, vous pouvez introduire une variable pour la rotation et la modifier en fonction des entrées clavier. En définissant ox et oy pour `love.graphics.draw`, vous déplacez l'origine de rotation au centre de l'image, ce qui permet une rotation autour du centre du personnage plutôt qu'autour de son coin supérieur gauche. Voici comment vous pourriez modifier l'exemple :

```
local image
local x, y
local vitesse = 200
local rotation = 0
local vitesseRotation = 1 -- Radians par seconde

function love.load()
  image = love.graphics.newImage("images/personnage.png")
  x = 400
  y = 300
end

function love.update(dt)
  if love.keyboard.isDown("right") then
    x = x + vitesse * dt
  end
  if love.keyboard.isDown("left") then
    x = x - vitesse * dt
  end
  if love.keyboard.isDown("down") then
    y = y + vitesse * dt
  end
  if love.keyboard.isDown("up") then
    y = y - vitesse * dt
  end
  if love.keyboard.isDown("d") then
    rotation = rotation + vitesseRotation * dt
  end
  if love.keyboard.isDown("q") then
    rotation = rotation - vitesseRotation * dt
  end
end

function love.draw()
  love.graphics.draw(image, x, y, rotation, 1, 1,
                     image:getWidth() / 2, image:getHeight() / 2)
end
```

Dans cet exemple :

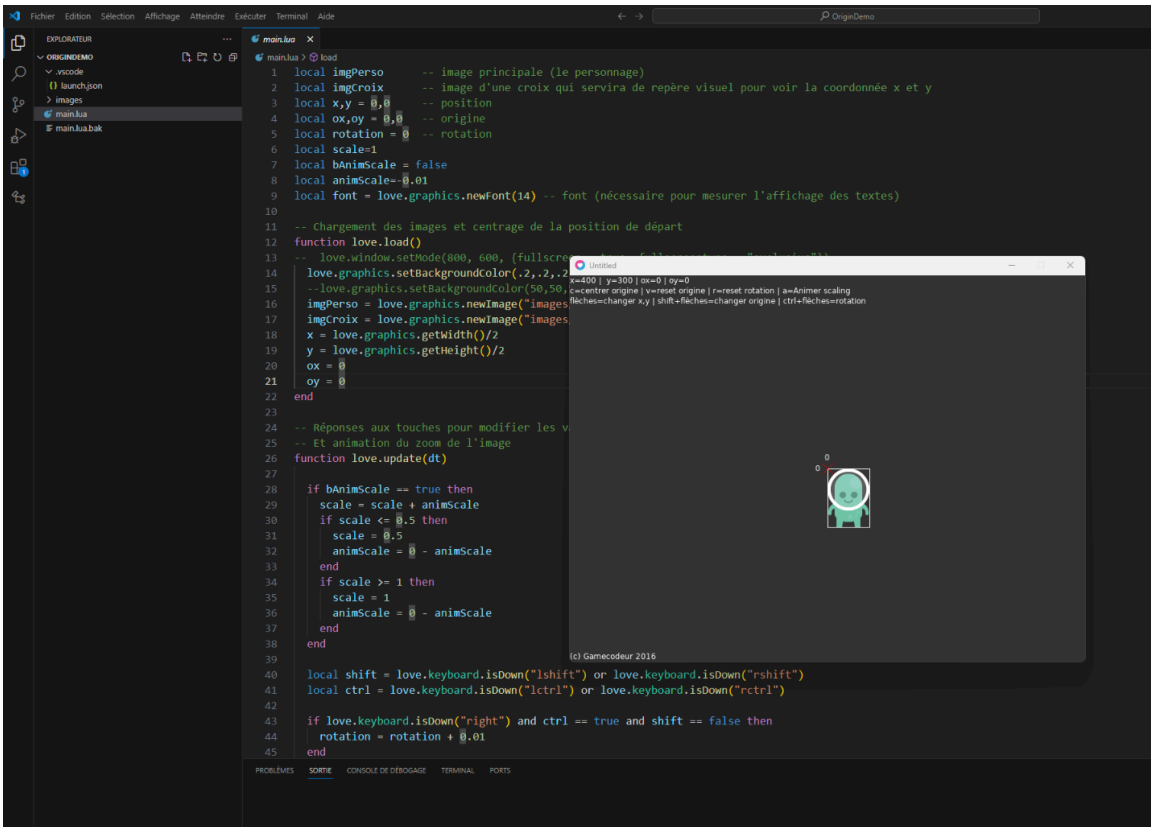
- `rotation` stocke l'angle de rotation actuel du personnage.
- `vitesseRotation` définit la vitesse à laquelle le personnage tourne.
- Lorsque l'utilisateur appuie sur D, la rotation augmente, et elle diminue avec Q.

Regardez les paramètres passés à `love.graphics.draw` : `rotation` pour l'angle, et les deux suivants (1, 1) pour l'échelle. Regardez aussi comment `image:getWidth() / 2` et `image:getHeight() / 2` déplacent l'origine de la rotation au centre de l'image. C'est la parfaite illustration de la leçon sur l'origine de l'image !

Si vous testez ce code sans modifier l'origine (en omettant les paramètres ox et oy), la rotation s'effectuera autour du coin supérieur gauche de l'image, ce qui donnera un effet visuel très différent (moche et inutilisable), où l'image tourne comme si elle était accrochée à un fil en son coin supérieur gauche.

Un projet de démo pour tout expérimenter !

J'ai créé un projet complet pour vous permettre d'expérimenter. Il permet, avec le clavier, de déplacer l'image, de changer son origine, de la faire tourner (rotation), et il affiche les valeurs à l'écran.



Utilisation :

- Déplacer l'image (changer ses coordonnées x et y) avec les flèches du clavier
- Maintenez SHIFT enfoncé + les flèches pour décaler l'origine
- Maintenez CTRL enfoncé + les flèches droite/gauche pour appliquer une rotation
- Pressez la touche "c" pour placer l'origine sur le centre de l'image
- Pressez la touche "v" pour replacer l'origine à 0,0 (valeur par défaut de Löve)
- Pressez la touche "r" pour annuler la rotation

Les sources du projet sont disponibles dans la partie bonus numérique de ce guide.

Programmez votre premier jeu : Space Attack

Face à une horde d'envahisseurs, éprouvez vos réflexes à bord de votre vaisseau spatial de combat. Ne laissez aucun ennemi passer, l'avenir de l'humanité dépend de vous !

Mode d'emploi :

Déplacez votre vaisseau avec les touches gauche et droite, et tirez avec la barre d'espace. Vous ne pouvez tirer que 3 projectiles à la fois. La difficulté augmente progressivement.

Ce jeu est inspiré du jeu Space Attack de mon guide '10 jeux supers faciles à programmer" :

https://school.gamecodeur.fr/guide-de-programmation-10-jeux-super-facile-a-programmer-50-pages-de-programmes-a-recopier-en-lua-love2d

Les fichiers (images, sons, police de caractères) nécessaires à la réalisation de ce jeu sont fournis avec la version numérique de ce guide.

Le code minimum pour démarrer

Créez un nouveau projet avec un fichier main.lua et tapez le code minimum pour notre projet :

```
function love.load()
end

function love.update(dt)
end

function love.draw()
end

function love.keypressed(key)
end
```

Si vous êtes perdus, paniquez ! A ce stade, savoir utiliser votre éditeur de code pour créer un projet doit être une seconde nature et prendre 50 secondes. Il n'y a rien à comprendre ici, juste créer un dossier, l'ouvrir avec Visual Studio Code et créer un nouveau fichier "`main.lua`".

Désolé mais si vous trouvez cela compliqué, comment allez vous faire pour :

- Programmer des milliers lignes de code
- Comprendre et utiliser des 100e de fonctions de programmation
- Résoudre des milliers de problématiques
- Comprendre les bugs que vous allez rencontrer…

Alors révisez la partie au début de ce guide pour créer un projet ou lancez-vous sans filet : rien ne va exploser et rien ne se passera si vous ne vous lancez pas ! Arrêtez d'avoir peur de votre ordi…

Autopsie du jeu Space Attack

A l'écran :

- Un fond qui scrolle à l'infini

- Le vaisseau principal

- Les projectiles

- Les vaisseaux ennemis

- Le score

Conceptuellement :

Le vaisseau principal se déplace de gauche à droite en faisant varier sa position x, sur la totalité de la largeur de l'écran.

Un « timer » va créer des ennemis, positionnés sur la largeur de l'écran à une position x aléatoire et légèrement en dessus de la position 0. Ce timer est cadencé sur une fréquence (`frequenceEnnemis`) qui réduit progressivement afin d'augmenter la difficulté. Les données sur chacun des ennemis sont stockées dans une liste (`ennemis`).

Les ennemis sont simplement déplacés, à chaque frame, vers le bas de l'écran, donnant l'impression que c'est le vaisseau principal qui avance !

Quand un ennemi sort de la limite inférieure de l'écran, il est supprimé.

Le joueur peut tirer en pressant la barre d'espace. Il s'agit d'ajouter un tir (dans la limite de 3) dans une liste (`tirs`). Ce tir va se comporter comme les ennemis, mais du bas vers le haut cette fois.

Si un tir entre en collision avec un ennemi, un son d'explosion est joué et l'ennemi et le tir sont supprimés de leur liste, et donc disparaissent de l'écran.

Dans ce jeu, j'ai ajouté un fond qui scrolle à l'infini, sur 2 niveaux.

Un cours complet sur les scrollings infinis est disponible à cette adresse :
https://www.youtube.com/watch?v=GgmNZCbFHJk

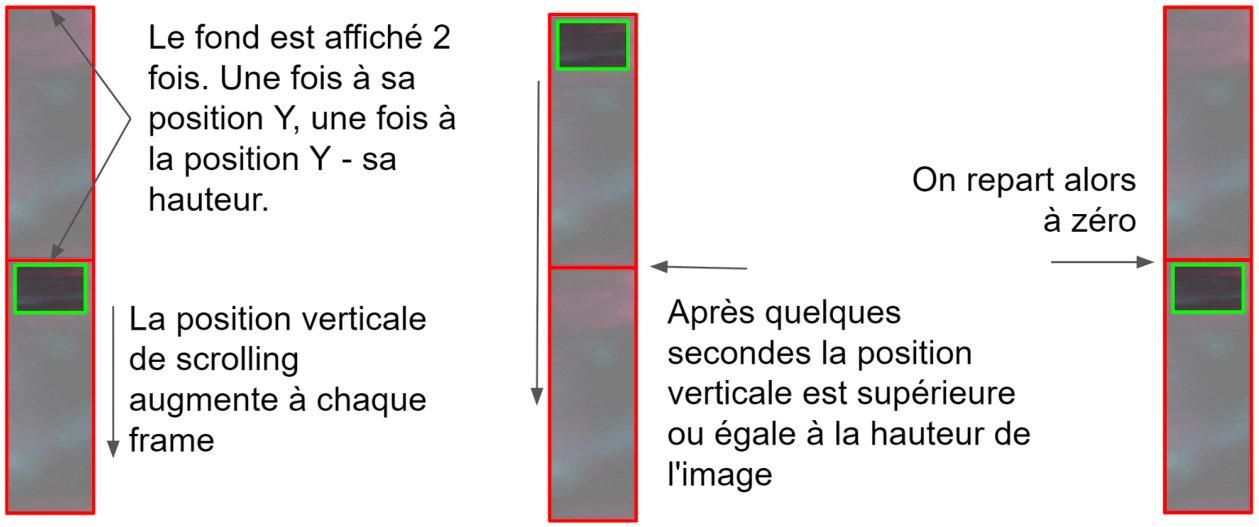
Programmer le scrolling infini

Le principe est d'afficher chaque fond 2 fois, pour couvrir tout l'écran, et de les déplacer vers le bas. Quand un fond a parcouru toute sa hauteur, son scrolling est réinitialisé.

J'ai essayé de vous représenter cela visuellement.

D'ailleurs, essayez toujours de visualiser les choses dans votre tête. La solution n'est pas de regarder vos lignes de code, mais bien de les comprendre et de vous faire une représentation mentale de ce que le code réalise…

Sur mon dessin, les fonds sont représentés par un cadre rouge. Et le cadre vert c'est l'écran.



Je vous propose de coder ça directement :

```
local scrolling = 0
local imageFond = love.graphics.newImage("fond.png")

function love.load()
    love.window.setTitle("Space Attack - (c) Gamecodeur 2023")
end

function love.update(dt)
    scrolling = scrolling + 25 * dt
    if scrolling >= imageFond:getHeight() then
        scrolling = 0
    end
end

function love.draw()
    love.graphics.draw(imageFond, 0, scrolling)
    love.graphics.draw(imageFond, 0, scrolling - imageFond:getHeight())
end
```

Regardez comment le fond est déplacé vers le bas, à la vitesse de 25 pixels par seconde. Et réinitialisé quand il a parcouru l'équivalent de sa hauteur.

Regardez comment le fond est dessiné 2 fois : une fois à sa position normale, une autre fois "au dessus" de lui-même. Enlevez une des 2 lignes et regardez le résultat pour comprendre comment la technique fonctionne. Vous verrez que sans dessiner 2 fois le fond, on a un moment où une partie de l'écran devient noire, la 2e version du fond vient couvrir cette zone.

Programmer le vaisseau principal

Ajoutons le vaisseau principal et permettons au joueur de le déplacer horizontalement :

```
local vaisseau = {}
local imageVaisseau = love.graphics.newImage("ship.png")

function InitJeu()
    vaisseau.x = 800 / 2
    vaisseau.y = 600 - imageVaisseau:getHeight()
end

function love.load()
    love.window.setTitle("Space Attack - (c) Gamecodeur 2023")
    InitJeu()
end

function love.update(dt)
    if love.keyboard.isDown("left") and vaisseau.x > 0 then
        vaisseau.x = vaisseau.x - (200 * dt)
    end
    if love.keyboard.isDown("right") and vaisseau.x - imageVaisseau:getWidth() < 800 then
        vaisseau.x = vaisseau.x + (200 * dt)
    end
end

function love.draw()
    love.graphics.draw(imageVaisseau, vaisseau.x, vaisseau.y)
end
```

Il y a un bug visuel dans ce code !

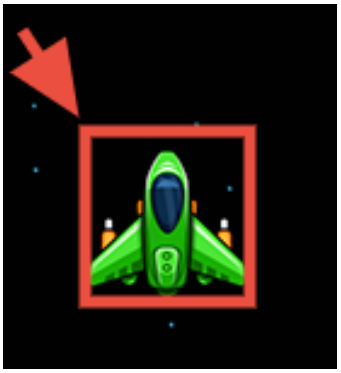
Le vaisseau est en effet affiché à une position `x` sensée être le centre de l'écran (800 / 2) :

```
    vaisseau.x = 800 / 2
```

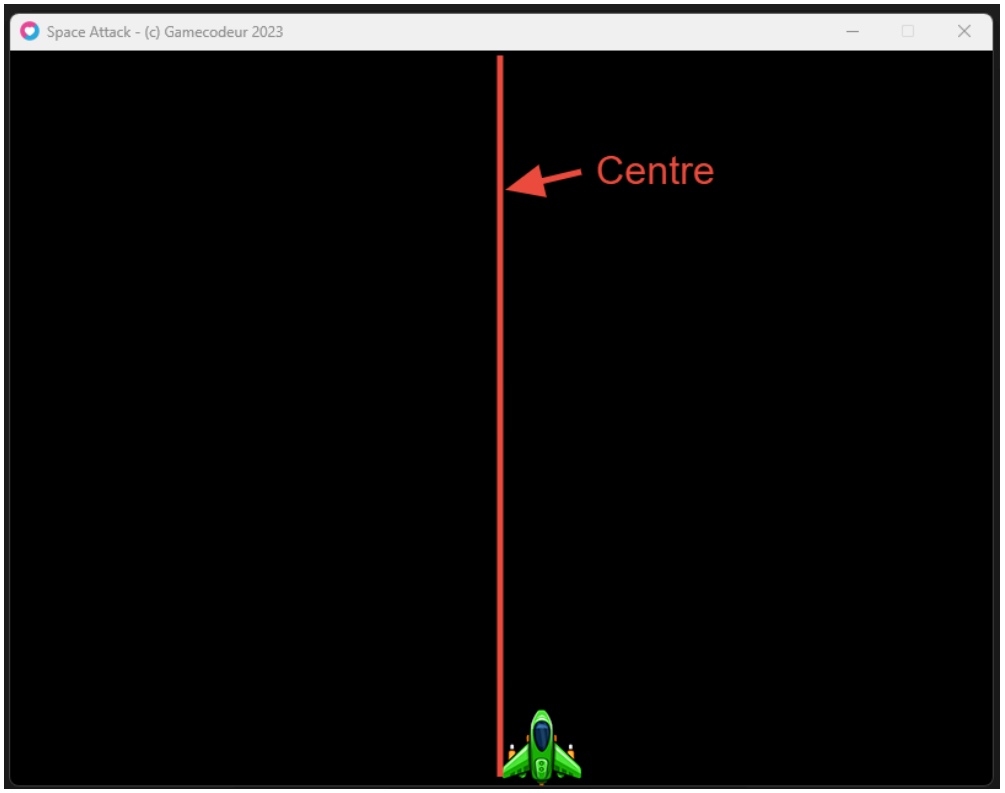
Mais en réalité, l'origine de l'image est son coin supérieur gauche :

```
    love.graphics.draw(imageVaisseau, vaisseau.x, vaisseau.y)
```

On a vu ça dans la section consacrée aux pixels et au point d'origine, rappelez-vous :



Et donc, même s'il en donne l'impression, notre vaisseau est décentré… Démonstration en traçant une verticale au centre de l'écran.



Comment centrer le vaisseau spatial à l'écran ?

Lorsque nous dessinons notre vaisseau spatial sur l'écran, nous voulons nous assurer qu'il est correctement centré. Pour ce faire, nous allons créer une fonction spéciale nommée `DrawCentre`. Cette fonction est conçue pour dessiner une image de sorte que les coordonnées (x, y) fournies correspondent au centre de l'image, et non à son coin supérieur gauche.

Voici à quoi ressemble la fonction `DrawCentre` :

```
function DrawCentre(image, x, y)
    love.graphics.draw(image, x, y, 0, 1, 1,
                       image:getWidth() / 2, image:getHeight() / 2)
end
```

Dans cette fonction, `image:getWidth() / 2` et `image:getHeight() / 2` calculent le centre de l'image pour les paramètres ox et oy de `love.graphics.draw`.

Dans la fonction `love.draw()`, nous allons maintenant utiliser `DrawCentre` pour dessiner le vaisseau afin qu'il soit correctement centré :

```
function love.draw()
    DrawCentre(imageVaisseau, vaisseau.x, vaisseau.y)
end
```

Programmer les ennemis

Pour illustrer une invasion d'ennemis en quête de conquête, imaginons des soucoupes volantes comme adversaires. Ces engins extraterrestres auront pour mission de traverser l'écran de haut en bas dans l'intention de percuter notre vaisseau.

Nous utiliserons un intervalle de temps entre l'apparition de chaque soucoupe, et cet intervalle se réduira progressivement, augmentant ainsi la fréquence des attaques.

Au début on aura une soucoupe de temps en temps, mais quand le rythme augmentera, il y aura plusieurs soucoupes en même temps à l'écran.

Et donc quand vous pensez "plusieurs" vous devez avoir un réflexe de crier le mot "LISTE !".

Le raisonnement, à chaque fois que vous avez dans votre jeu une "collection" d'éléments (ennemis, particules, tirs, etc.), c'est de découper le travail en 4 étapes :

- Créer une liste vide (et la vider quand on recommence une partie)
- Ajouter des éléments à cette liste à moment donné (ça dépend du gameplay)
- Mettre à jour cette liste à chaque update (par exemple déplacer les éléments si besoin)
- Afficher les éléments de la liste à l'écran dans le draw.

Etape 1 : Création et initialisation

Ajoutez ce code au début de votre programme pour charger l'image et déclarer la liste :

```
imageEnnemi = love.graphics.newImage("ennemi.png")
local ennemis = {}
```

Et initialisez la liste à vide à l'initialisation (dans InitJeu) :

```
ennemis = {}
```

Etape 2 : Ajouter des éléments

Cette étape dépend de votre gameplay. Par exemple pour des tirs, ce sera au moment où le joueur pressera un bouton, etc. Dans notre cas présent, c'est en fonction d'un intervalle de temps.

Utilisez un timer pour déterminer à quel moment ajouter une nouvelle soucoupe à la liste.

A déclarer au début de votre code :

```
local timerEnnemis = 0
local frequenceEnnemis = 3  -- Commence avec une soucoupe toutes les 3 secondes
```

Ensuite, intégrez le traitement du timer et le processus d'ajout d'un ennemi quand le timer dépasse le délai prévu (au départ 3 secondes) :

```
function love.update(dt)
    -- Code du vaisseau
    -- .... (voir pages précédentes)

    -- Code pour ajouter un ennemi toutes les x secondes :
    timerEnnemis = timerEnnemis + dt
    if timerEnnemis > frequenceEnnemis then
        -- Crée une table pour y stocker les infos du nouvel ennemis
        local nouvelEnnemi = {
            -- Commence à une position horizontale aléatoire
            x = love.math.random(0, 800),
            -- Commence verticalement en dehors de l'écran
            y = 0 - imageEnnemi:getHeight(),
        }
        -- Ajoute la table à notre liste
        table.insert(ennemis, nouvelEnnemi)
        -- Redémarre le timer
        timerEnnemis = 0
    end
end
```

Dans l'ordre, que fait ce code ?

Calcul du temps écoulé :

```
timerEnnemis = timerEnnemis + dt
```

Ici on augmente notre `timerEnnemi` en y ajoutant le delta time reçu par la fonction `love.update`. Rappel : Le delta time contient le temps en seconde qui s'est écoulé depuis la dernière frame. En additionnant le dt à chaque frame, on compte donc le temps qui passe tout simplement. Sur un écran à 60Hz, dt contient environ 0,016666.

Vérification du temps écoulé :

```
    if timerEnnemis > frequenceEnnemis then
```

Ici on vérifie si le délai est écoulé. Par exemple, quand `timerEnnemis` contiendra une valeur supérieure à 3, la condition sera exécutée.

Création d'un nouvel ennemi (si le délai est écoulé) :

```
        -- Crée une table pour y stocker les infos du nouvel ennemis
        local nouvelEnnemi = {
            -- Commence à une position horizontale aléatoire
            x = love.math.random(0, 800),
            -- Commence verticalement en dehors de l'écran
            y = 0 - imageEnnemi:getHeight(),
        }
        -- Ajoute la table à notre liste
        table.insert(ennemis, nouvelEnnemi)
```

Ici je vais m'attarder sur la façon dont je crée un nouvel ennemi :

Cela s'appelle créer une table "à la volée". C'est-à-dire que je crée la table `nouvelEnnemi` et j'y insère des données en même temps. Mais peut-être que vous aimerez cette autre méthode, en 2 étapes, qui est plus lisibl :

```
        -- Crée une table pour y stocker les infos du nouvel ennemis
        local nouvelEnnemi = {}
        -- Commence à une position horizontale aléatoire
        nouvelEnnemi.x = love.math.random(0, 800),
        -- Commence verticalement en dehors de l'écran
        nouvelEnnemi.y = 0 - imageEnnemi:getHeight()
        -- Ajoute la table à notre liste
        table.insert(ennemis, nouvelEnnemi)
```

Utilisez la méthode qui vous plaît le plus. Le résultat est le même : on crée une table `nouvelEnnemi` et on y stocke 2 valeurs : x et y, qui contiennent la position de départ de la soucoupe.

Remise à 0 du timer :

```
        -- Redémarre le timer
        timerEnnemis = 0
```

On remet ici le timer à 0, sinon il sera toujours dépassé à la prochaine frame et on se retrouvera avec des centaines de soucoupes en quelques secondes. En le réinitialisant à 0 on le recommence au début tout simplement.

Etape 3 : Mise à jour des ennemis

À chaque frame, vous devez mettre à jour la position de chaque ennemi dans la liste. Et parce qu'il sera nécessaire de supprimer parfois des ennemis, il est obligatoire de parcourir la liste à l'envers.

```
function love.update(dt)
    -- Code existant pour le vaisseau, le timer et l'ajout d'ennemis
    -- ...

    -- Mise à jour des ennemis
    for i = #ennemis, 1, -1 do
        ennemis[i].y = ennemis[i].y + 100 * dt  -- Déplacer l'ennemi vers le bas
        -- Si l'ennemi dépasse le bas de l'écran
        if ennemis[i].y > 600 + imageEnnemi:getHeight() / 2 then
            table.remove(ennemis, i)
        end
    end
end
```

Dans ce code, on parcourt la liste `ennemis` en utilisant un index (`i`). Et pour chaque élément de la liste (`ennemis[i]`) on change la valeur de `y` pour le faire descendre.

IMPORTANT : Notez qu'il faut multiplier la distance par le delta time (`dt`) afin que la vitesse de déplacement soit homogène quel que soit le taux de rafraîchissement de l'écran du joueur. Ce qui est pratique c'est que du coup la vitesse est exprimée en pixels par seconde. Donc ici nos soucoupes se déplacent de 100 pixels par seconde.

Ensuite, si l'ennemi a dépassé le bas de l'écran (on ajoute la moitié de la hauteur de l'image pour être précis), on le supprime de la liste. Si on ne le supprimait pas, on aurait au bout d'un moment plein de soucoupes volantes qui continuent à voler, sans qu'on ne les voient.

Variante pour la hauteur de l'écran : utiliser `love.graphics.getHeight()` et obtenir la taille via Love2D au lieu de l'imposer en dur. Ce sera plus évolutif. Voilà ce que ça donne :

```
if ennemis[i].y > love.graphics.getHeight() then
```

Je vous propose d'ajouter maintenant dans `love.update` le code pour accélérer la fréquence d'apparition des ennemis :

```
    if frequenceEnnemis > 0.1 then
        frequenceEnnemis = frequenceEnnemis - (0.1 * dt)
    end
```

Ce code va réduire l'intervalle à chaque frame, sans qu'il descende en dessous de 0.1.

Etape 4 : Affichage des ennemis

Enfin, dans la fonction love.draw(), vous devez dessiner chaque ennemi à sa position actuelle.

```
function love.draw()
    for i, ennemi in ipairs(ennemis) do
        DrawCentre(imageEnnemi, ennemi.x, ennemi.y)
    end
end
```

Pour ce traitement on utilise la technique du `ipairs()` qui permet de récupérer chaque index et chaque élément de la liste dans une variable. C'est la version académique de Lua, et la plus utilisée. Vous ne pouvez l'utiliser que si vous n'avez pas de suppression à faire dans la liste ! Je rappelle que s'il y a des suppressions à réaliser il faut obligatoirement parcourir la liste à l'envers, et pour cela la seule solution c'est d'utiliser un index.

Voici deux autres méthodes possibles pour parcourir une liste à l'endroit, sans nécessité de suppression :

Méthode avec index direct

```
function love.draw()
    for i = 1, #ennemis do
        DrawCentre(imageEnnemi, ennemis[i].x, ennemis[i].y)
    end
end
```

Méthode avec index et variable relai

```
function love.draw()
    for i = 1, #ennemis do
        local ennemi = ennemis[i]
        DrawCentre(imageEnnemi, ennemi.x, ennemi.y)
    end
end
```

Toutes ces méthodes font la même chose au final. Choisissez la méthode la plus lisible pour vous.

Programmer les tirs

Vous allez voir, programmer les tirs va partager une très grande partie des concepts de la programmation des ennemis.

En effet un tir c'est comme une soucoupe, sauf que le tir part du canon du vaisseau, donc de bas en haut. Et on peut avoir plusieurs tirs en même temps à l'écran si le joueur tire en rafale.

C'est donc le moment de ressortir le raisonnement que je vous ai enseigné plus haut. Je fais un copier/coller pour bien vous démontrer que c'est le même :

Le raisonnement, à chaque fois que vous avez dans votre jeu une "collection" d'éléments (ennemis, particules, tirs, etc.), c'est de découper le travail en 4 étapes :

- *Créer une liste vide (et la vider quand on recommence une partie)*
- *Ajouter des éléments à cette liste à moment donné (ça dépend du gameplay)*
- *Mettre à jour cette liste à chaque update (par exemple déplacer les éléments si besoin)*
- *Afficher les éléments de la liste à l'écran dans le draw.*

Vous voyez la similitude avec les ennemis ? Il suffit de personnaliser le raisonnement pour qu'il corresponde à ce besoin de générer des tirs.

Alors reprenons les étapes une par une, vous pourrez ainsi comparer avec la section précédente.

Etape 1 : Création et initialisation

Ajoutez ce code au début de votre programme :

```
imageTir = love.graphics.newImage("tir.png")
local tirs = {}
```

Et initialisez la liste à vide à l'initialisation (dans `InitJeu`) :

```
tirs = {}
```

Etape 2 : Ajouter des éléments

Dans notre cas présent, ce sera au moment où le joueur pressera un bouton ou une touche.

Mais je vous propose de préparer une fonction pour ajouter un tir. Cette fonction permettra d'avoir un code plus propre car mieux segmenté.

Vous me direz : pourquoi tu n'as pas créé une fonction pour créer un ennemi ?

Car j'ai voulu rester simple. Créer des fonctions n'est pas le réflexe premier d'un débutant, et j'essaye de coder "au plus simple" au départ, pour ne pas encombrer chaque chapitre de multiples concepts. Mais vous pouvez donc le faire comme exercice en vous inspirant de ce que je vais vous montrer ici pour les tirs.

Exercice : En vous inspirant de la fonction "`Tire`" ci-dessous, créez une fonction "`CreeEnnemi`" qui se chargera de créer et ajouter un ennemi à la liste `ennemis`, et utilisez-la dans votre code à la place du code précédemment enseigné.

En attendant, voici comment créer une fonction pour ajouter un tir. Insérez ce code par exemple au début de votre main.lua, juste après les déclarations de variables :

```
function Tire()
    local leTir = {
        x = vaisseau.x,
        y = vaisseau.y
    }
    table.insert(tirs, leTir)
end
```

Dans cette fonction, nous créons une table `leTir` contenant les informations sur le nouveau tir et nous l'ajoutons à la liste "`tirs`".

Notez que la position de départ du tir est le centre du vaisseau. La fonction a accès à la variable `vaisseau` car elle est écrite dans le même module (`main.lua`) que celui de la variable.

Ensuite, pour gérer le clavier, Notre cher Love2D nous fournit une call back super pratique : `love.keypressed`.

Cette call back, à l'instar de `love.load`, `love.update`, `love.draw`, est exécutée automatiquement par Love2D vous n'avez rien à faire à part la déclarer.

Dans le cas de `love.keypressed`, la fonction est appelée automatiquement à chaque fois qu'une touche est enfoncée.

```
function love.keypressed(key)
    if key == "space" then
        Tire()
    end
end
```

Si vous voulez limiter le nombre tirs à 3 en même temps au maximum, voici comment faire :

```
function love.keypressed(key)
    if key == "space" and #tirs < 3 then
        Tire()
    end
end
```

Etape 3 : Mise à jour des tirs

À chaque frame, vous devez mettre à jour la position de chaque tir dans la liste. Et parce qu'il sera nécessaire de supprimer parfois des tirs, il est obligatoire de parcourir la liste à l'envers.

```
for n = #tirs, 1, -1 do
    local leTir = tirs[n]
    leTir.y = leTir.y - (400 * dt)
    if leTir.y < 0 - imageTir:getHeight() / 2 then -- Sort de l'écran
        table.remove(tirs, n)
    end
end
```

Dans ce code, on parcourt la liste `tirs` en utilisant un index (`n`) et on en extrait les éléments via `tirs[n]`. Et pour chaque élément de la liste on change la valeur de `y` pour le faire descendre.

Rappel : Il faut multiplier la distance par le delta time (`dt`) afin que la vitesse de déplacement soit homogène quel que soit le taux de rafraîchissement de l'écran du joueur. Ce qui est pratique c'est que du coup la vitesse est exprimée en pixels par seconde. Donc ici nos soucoupes se déplacent de 400 pixels par seconde.

Ensuite, si le tir dépasse le haut de l'écran, on le supprime de la liste. Si on ne le supprimait pas, on aurait au bout d'un moment plein de tirs qui continuent à avancer, sans qu'on ne les voient. Cela utiliserait du temps de calcul pour rien.

Etape 4 : Affichage des tirs

Enfin, dans la fonction love.draw(), vous devez dessiner chaque tir à sa position actuelle.

```
for k, v in ipairs(tirs) do
    DrawCentre(imageTir, v.x, v.y)
end
```

Comparaison avec l'ajout des ennemis

Prenez le temps de lire le code de l'ajout des ennemis et de le comparer avec le code de l'ajout des tirs. Remarquez comme le concept est le même et quels sont les points communs et les différences au niveau du code.

Décomposer-le en suivant les 4 étapes :

- *Créer une liste vide (et la vider quand on recommence une partie)*
- *Ajouter des éléments à cette liste à moment donné (ça dépend du gameplay)*
- *Mettre à jour cette liste à chaque update (par exemple déplacer les éléments si besoin)*
- *Afficher les éléments de la liste à l'écran dans le draw.*

Détruire les ennemis

Lorsqu'un tir touche un ennemi, on va le détruire.

Pour cela nous devons, à chaque frame, regarder si chaque tir n'entre pas en contact avec chaque ennemi. C'est un concept traditionnel, et apprendre à le programmer va vous servir à l'infini.

En pseudo code ça donnerait :

```
POUR CHAQUE TIR
        ET POUR CHAQUE ENNEMI
                SI LA DISTANCE ENTRE LE TIR ET L'ENNEMI EST INFÉRIEURE À X ALORS
                        L'ENNEMI EST DÉTRUIT
                        LE TIR EST DÉTRUIT
                        ON SORT DE LA BOUCLE
                FIN DE SI
        FIN DE POUR CHAQUE ENNEMI
FIN DE POUR CHAQUE TIR
```

Pourquoi sortir de la boucle ? Car le tir étant détruit, pas la peine de continuer à le tester avec les autres ennemis, puisque de toute façon il a déjà explosé !

Note si vous trouvez ce code complexe : Quand on débute, on a un peu de mal avec les boucles imbriquées, pourtant c'est très souvent utilisé donc persévérez pour visualiser mentalement ce code et le comprendre.

Concernant le test de collision :

Dans un jeu vidéo on a très souvent besoin de "détecter des collisions". Cela signifie tester si un élément du jeu rentre en contact avec un autre.

Il y a 2 méthodes classiques pour tester les collisions dans un jeu vidéo en 2D :

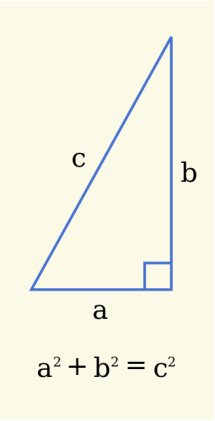
- La boite de collision (bounding box) avec la technique AABB
- La distance

Pour simplifier le code de ce premier jeu vidéo, j'utilise la technique de la distance. Le principe est de calculer la distance entre un tir et un ennemi, et si cette distance est inférieure à une certaine valeur, on considère qu'ils se touchent.

Pour calculer la distance entre 2 éléments, nous avons à notre disposition leurs coordonnées. Et pour déterminer la distance correspondant à un contact, nous prendrons le rayon de la soucoupe volante ennemie. Cela ne sera pas hyper précis mais c'est largement suffisant.

Et là je vais vous apprendre un truc : le théorème de pythagore sert à quelque chose ! C'est lui que nous allons utiliser pour calculer la distance entre un tir et un ennemi.

Pas de panique si vous détestez les mathématiques, vous pourrez recopier la formule et l'utiliser sans la comprendre. Pour les curieux, voici un rappel du théorème.

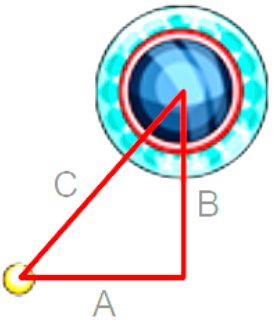


Le carré de la longueur de l'hypoténuse est égal à
la somme des carrés des longueurs des deux autres côtés

Traduction :

- On veut connaître la longueur du côté C
- On additionne "la somme des carrés des 2 autres côtés A et B".
- On obtient le carré de la longueur du côté C (faudra appliquer une racine carré pour obtenir la longueur réelle du coup)

Vous me direz, "mais nous on a pas de triangles dans notre jeu !". Détrompez-vous ! Regardez :



Nous avons donc bien 3 côtés, il suffit d'appliquer la formule de monsieur Pythagore dans une fonction qu'on appelera modestement "Distance" :

```lua
function Distance(x1, y1, x2, y2)
    -- Calcul de la différence en x
    local differenceX = x2 - x1
    -- Calcul de la différence en y
    local differenceY = y2 - y1
    -- Élévation au carré des différences
    local carreDifferenceX = differenceX ^ 2
    local carreDifferenceY = differenceY ^ 2
    -- Somme des carrés
    local sommeDesCarres = carreDifferenceX + carreDifferenceY
    -- Racine carrée de la somme pour obtenir la longueur
    local distance = sommeDesCarres ^ 0.5
    return distance
end
```

J'ai voulu ici tout décomposer, en stockant le résultat de chaque calcul dans une variable, et du coup la fonction est un peu longue. Mais cela me permet de vous expliquer chaque étape :

- `differenceX` : la différence entre les coordonnées x des deux points.
- `differenceY` : la différence entre les coordonnées y des deux points.
- `carreDifferenceX` : le carré de la différence en x.
- `carreDifferenceY` : le carré de la différence en y.
- `sommeDesCarres` : la somme des carrés des différences en x et en y.
- `distance` : comme le résultat du théorème est une valeur au carré, on calcul la racine carrée de cette valeur pour obtenir la distance

On peut en réalité simplifier cette fonction en la réduisant à une seule ligne :

```
function Distance(x1, y1, x2, y2)
    return ((x2 - x1) ^ 2 + (y2 - y1) ^ 2) ^ 0.5
end
```

Si vous cherchez plus de fonctions du genre, voici ma source :
https://love2d.org/wiki/General_math

Voilà, maintenant qu'on sait calculer une distance avec monsieur Pythagore, on peut convertir notre pseudo code en vrai code, ça donne donc :

```
for n = #tirs, 1, -1 do
    local leTir = tirs[n]
        for nc = #ennemis, 1, -1 do
            local lEnnemi = ennemis[nc]
            local tailleEnnemi = imageEnnemi:getWidth()
            if Distance(lEnnemi.x, lEnnemi.y,
                        leTir.x, leTir.y) < tailleEnnemi / 2 then
                table.remove(ennemis, nc)
                table.remove(tirs, n)
                break -- sort de la boucle
            end
        end
end
```

Je vous en explique chaque étape :

Parcours des tirs : La boucle `for n = #tirs, 1, -1 do` parcourt la liste des tirs en partant du dernier élément vers le premier. Cette manière de parcourir la liste permet de supprimer des éléments de la liste sans perturber l'ordre des indices pendant l'itération.

Accès au tir courant : `local leTir = tirs[n]` récupère le tir courant dans la liste pour le vérifier contre chaque ennemi.

Parcours des ennemis : À l'intérieur de la boucle des tirs, une deuxième boucle `for nc = #ennemis, 1, -1 do` parcourt de la même manière la liste des ennemis pour vérifier chaque ennemi contre le tir courant.

Accès à l'ennemi courant : `local lEnnemi = ennemis[nc]` récupère l'ennemi courant pour pouvoir ensuite vérifier s'il a été touché par le tir.

Calcul de la distance : La fonction `Distance` est appelée avec les positions de l'ennemi et du tir. Si cette distance est inférieure à la moitié de la largeur de l'ennemi (`tailleEnnemi / 2`), cela signifie que le tir a touché l'ennemi.

Suppression des éléments : Si un tir touche un ennemi, l'ennemi est retiré de la liste (`table.remove(ennemis, nc)`), et le tir est également retiré (`table.remove(tirs, n)`).

Sortie de la boucle : L'instruction `break` arrête la boucle interne après qu'un tir touche un ennemi, car il n'est pas nécessaire de vérifier ce tir contre les autres ennemis.

En résumé, ce code vérifie pour chaque tir s'il a touché un ennemi et si c'est le cas il supprime à la fois le tir et l'ennemi de leur liste respective.

Etant donné que nous avons déjà une boucle qui déplace les tirs, nous pouvons la modifier pour qu'elle réalise le test des collisions en même temps. Ceci, si le tir ne sort pas de l'écran (else).

```
for n = #tirs, 1, -1 do
    local leTir = tirs[n]
    leTir.y = leTir.y - (400 * dt)
    if leTir.y < 0 - imageTir:getHeight() / 2 then -- Sort de l'écran
        table.remove(tirs, n)
    else -- Sinon teste collisions avec chaque ennemi
        for nc = #ennemis, 1, -1 do
            local lEnnemi = ennemis[nc]
            local tailleEnnemi = imageEnnemi:getWidth()
            if Distance(lEnnemi.x, lEnnemi.y,
                        leTir.x, leTir.y) < tailleEnnemi / 2 then
                table.remove(ennemis, nc)
                table.remove(tirs, n)
                break -- sort de la boucle
            end
        end
    end
end
```

Score et sons

Pour donner un peu plus de dynamisme à notre jeu, ajoutons des sons et un score.

Pour le score, déclarez une variable au début de votre `main.lua` :

```
local score = 0
```

Et il faut penser à réinitialiser ce score dans la fonction qui remet toutes les valeurs du jeu à zéro :

```
function InitJeu()
    score = 0
    vaisseau.x = 800 / 2
    vaisseau.y = 600 - imageVaisseau:getHeight()
    vaisseau.explose = 0
    ennemis = {}
    tirs = {}
    frequenceEnnemis = 3
end
```

Et pour l'afficher ajoutez ce code à la fin de la fonction `love.draw` :

```
    love.graphics.print(score, 5, 5)
```

Pour les sons, il faut les charger au départ, insérez donc ce code au début de votre `main.lua` :

```
sonTir = love.audio.newSource("tir.wav", "static")
sonExplosion = love.audio.newSource("explosion.wav", "static")
```

Note : je vous rappelle que tous les fichiers sons, images et police de caractères nécessaires à la réalisation de ce projet sont fournis avec la version numérique de ce guide. Il faut préalablement tous les copier dans le dossier de votre projet. Vous pouvez aussi utiliser vos propres images et vos propres sons.

Ensuite, au moment de tirer, il faut lancer le son. Au passage, on le stoppe avant de le jouer, si jamais le joueur enchaîne les tirs, car un son ne peut être joué qu'une seule fois. Si le son du précédent tir n'est pas terminé, on pourrait avoir des sons qui ne sont pas joués.

```
function Tire()
    local leTir = {
        x = vaisseau.x,
        y = vaisseau.y
    }
    table.insert(tirs, leTir)
    sonTir:stop()
    sonTir:play()
end
```

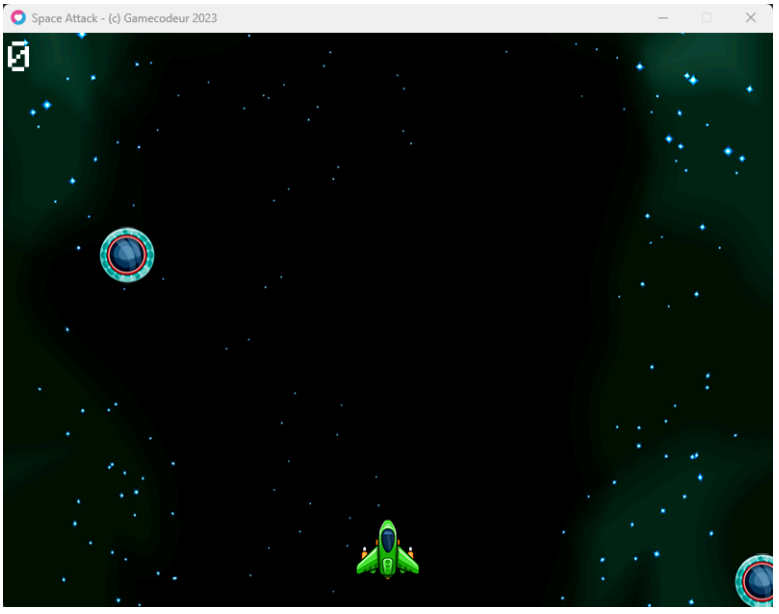
Ensuite, repérez le code qui détruit un ennemi et ajoutez :

```
for n = #tirs, 1, -1 do
    local leTir = tirs[n]
    leTir.y = leTir.y - (400 * dt)
    if leTir.y < 0 - imageTir:getHeight() / 2 then -- Sort de l'écran
        table.remove(tirs, n)
    else -- Sinon teste collisions avec chaque ennemi
        for nc = #ennemis, 1, -1 do
            local lEnnemi = ennemis[nc]
            local tailleEnnemi = imageEnnemi:getWidth()
            if Distance(lEnnemi.x, lEnnemi.y,
                        leTir.x, leTir.y) < tailleEnnemi / 2 then
                score = score + 1
                table.remove(ennemis, nc)
                table.remove(tirs, n)
                sonExplosion:stop()
                sonExplosion:play()
                break -- sort de la boucle
            end
        end
    end
end
```

On peut aussi améliorer l'aspect du score en utilisant une police de caractère façon pixels :

```
local scoreFont = love.graphics.newFont("pixelmix.ttf", 35)
love.graphics.setFont(scoreFont)
```

Voici le résultat :



Game Over !

Si votre vaisseau est touché, ce serait sympa qu'il explose et que le jeu s'arrête.

La première chose à faire est d'être capable de "noter" que le vaisseau a explosé, ajoutons donc une variable booléenne à notre vaisseau :

```
function InitJeu()
   …
   vaisseau.x = 800 / 2
   vaisseau.y = 600 - imageVaisseau:getHeight()
   vaisseau.explose = 0
   …
end
```

Ensuite, chargeons une image d'explosion au début de notre code :

```
local imageExplosion = love.graphics.newImage("explosion.png")
```

Pour savoir si le vaisseau est touché par un ennemi, vous serez peut-être étonné(e) si je vous dis que vous savez déjà le faire ? C'est le même principe que pour les tirs mais en plus simple, car ici nous n'avons pas à faire de double boucle. Allons donc modifier le code qui s'occupe de déplacer les ennemis, et si l'ennemi ne sort pas de l'écran alors testons sa distance avec le vaisseau :

```
    for n = #ennemis, 1, -1 do
        local ennemi = ennemis[n]
        ennemi.y = ennemi.y + 200 * dt
        if ennemi.y > 600 + imageEnnemi:getHeight() / 2 then
            table.remove(ennemis, n)
        else
            if Distance(vaisseau.x, vaisseau.y,
                        ennemi.x, ennemi.y) < imageEnnemi:getWidth() then
                vaisseau.explose = true
                sonExplosion:stop()
                sonExplosion:play()
            end
        end
    end
```

Pour relancer le jeu, je vous propose de détecter l'appui sur la touche "ECHAP" et dans ce cas, exécuter tout simplement notre fonction `InitJeu` :

```
function love.keypressed(key)
    if key == "escape" then
        InitJeu()
    elseif key == "space" and #tirs < 3 then
        Tire()
    end
end
```

Epilogue

Ouaw ! Vous êtes arrivé(e) au bout de cette formation ! Comment vous sentez-vous ?

Si c'est comme un héros, c'est bon signe, c'est que vous êtes fait(e) pour la programmation.

Vous avez, en quelques heures (ou quelques jours selon votre rythme) :

- Appris les bases de la programmation
- Codé un premier jeu vidéo

Si vous vous êtes contenté(e) de lire, alors vous n'avez pas vraiment appris. Je vous conseille de reprendre la lecture de ce guide et de taper du code.

Le code c'est la vie !

Vous n'êtes pas obligé(e) de simplement recopier. Vous pouvez essayer de changer un peu le code si des idées vous viennent. Et j'ai une bonne nouvelle : vous pouvez le faire même si vous cassez tout et que plus rien ne marche. Rien ne va exploser et rien n'est grave. C'est ce qui est génial avec la programmation si on la compare avec le bricolage…

Merci de m'avoir lu et de me faire confiance.

Je vous souhaite le meilleur pour la suite.

Bon code et restez libres !

David

Vous voulez continuer à apprendre ?

Retrouvez tous mes guides et formations sur ma boutique en ligne :

https://school.gamecodeur.fr