-

This french translation of the FUZE documentation is a work in progress done by Jean-Marc “jihem” QUERE (@wdwave) for the FUZE team, released as-is due to COVID-19 quarantine. Will be finished as soon as possible. #savelivestayhome

Introduction

Il y a des millions d’années, notre monde a été envahi par une espèce de créatures sans corps, les Geeks. Millénaires après millénaires, ils ont attendu et même encore plus. Ils ont grandi dans l’ennui. Finalement, les humains ont évolué. Les Geeks ont regardé. Au début, ils étaient amusés par la capacité incessante de l’humain à inventer des choses absolument inutiles auxquelles ils allaient accorder une grande importance. Ils ont regardé un peu plus et se sont de nouveau ennuyés.

Finalement, les Geeks sont devenus capables d’influencer les pensées et les actions des humains. Au cours de centaines d’années et ce jusqu’au milieu du XXe siècle, les Geeks ont mis au point une méthode permettant de conquérir le cœurs et l’esprit de milliers d’humains. Au lieu d’inventer de nouveaux types de chaise, de sacs à main ou d’autres fromages fantaisistes, ils se sont concentrés sur le développement de puissantes technologies de calcul ouvrirant la voie à une nouvelle ère d’invention et de divertissement.

La première vague de ces nouveaux systèmes était beaucoup trop complexe pour que les humains ordinaires et inoccupés puissent la comprendre mais d’autres suivirent rapidement. Ainsi, dans les années 1970, l’ordinateur domestique est arrivé et avec lui l’informatique destinée aux masses que même la plus petite intelligence humaine pouvait comprendre.

Alan Turing, Bill Gates, Steve Wozniak, Steve Jobs, Grace Murray Hopper, Nolan Bushnel, Jack Tramiel, Sir Clive Sinclair, Seymour Crey, Ada Lovelace, Charles Babbage, Gottfried Leibniz, Hermann Hauser et Chris Curry ne sont que quelques-uns des Geesk les plus célèbres et reconnus. Il y en a beaucoup, beaucoup plus et beaucoup trop nombreux pour être mentionnés, mais une chose est sûre. Il n’y a pas si longtemps, ils ont réussi à réaliser leur rêve. Regardez autour de vous… Assurément, les Geeks ont hérité de la Terre !

-

FUZE Technologies Ltd sait reconnaître une bonne chose quand elle la voit et elle veut prendre part à l'action. FUZE^4 Nintendo Switch est notre offre aux Geeks en herbe du monde entier. Nous espérons que vous l'apprécierez.

Merci

FUZE^4 Nintendo Switch a été développé au Royaume-Uni par une très petite équipe. Ce projet s'est déroulé parallèlement aux activités habituelles de FUZE, en particulier l'organisation de centaines d'ateliers de codage FUZE destinés à des jeunes du Royaume-Uni. Au départ, lorsque nous avons annoncé que nous nous lançions dans le projet, une page Web acceptant les dons a été ajoutée sur le site Web de FUZE.

Nous ne pourrons jamais le crier suffisamment fort : énorme merci de la part de notre équipe à tous nos donateurs ! Cela signifie énormément pour nous de recevoir votre soutien et nous n'aurions pas pu le faire sans vous !

Un merci tout particulier pour leur soutien sans faille, leurs encouragements et leur génie à :

Charlie! Shila Odedra-Silvera, Leonard Teague, Helen & Anna Silvera, Emma, Mark and Josh Mulcahy, Nic, Caroline and Josh, Philip Prall, Jared Forrester, James Hands, Lucie Dickinson, Luke Schofield, Christian Hegyi, Martin White, Anil & Milan Modhwadia, Andrew and Lorna Johns, Ellis Durden, James Silcox of Reach the Core, John Ronane, Dawn, Chris, Charlotte and Elizabeth Basnett, Rod and Iris Ellis, Shane, Leanne, Luke and Mia Ellis, The Rgt Honourable John Bercow, Ickford Combined School, Wheatley Park School, The Hall School, John Hampden School, Alex and Vicki and Adrian Mietusiewicz

L'équipe FUZE

Jon Silvera Parangon de l'Arbitrage et de l'Exécution, Fondateur et PDG / Chef de Projet et Investisseur

Jon Clough Parangon de l'Intégrité, Directeur Financier et Investisseur

-

Derek Taylor Investisseur

Colin Bodley Parangon de Stoïcisme, Guide de Référence du Programmeur, Contributeur à l'Aide, Testeur produit et Investisseur

Luke Mulcahy Paragnon de Sagesse, Programmer Principal des moteurs (2D, 3D et Audio), de l'Editeur et bien bien plus encore

David Silvera Parangon de Vérité, Concepteur, Programmeur, Didacticiels et Aide, Assets Audio / consultant technique et Professeur Principal

Will Tice Programmeur de l'interface utilisateur, du support de l'éditeur, des sprites 2D, des outils de carte et d'image et bien plus encore

Kat Deak Parangon de l'Art, artiste graphique 3D et contributeur, contrôle de la qualité des modèles 3D et tests

Mike Green Support additionnel à la programmation, développement Web et contrôle qualité

Molly Odedra-Silvera Parangon d'Éloquence, support marketing, testeur produit, contrôle de la qualité et Professeur Assistant

Lizzie Botelle Parangon d'Orthographe, Testeur produit et contrôle qualité

Ben Taylor Testeur produit, contrôle qualité et Professeur Assistant

Grace Odedra-Silvera, Charlotte Reeve, Hannah & Mica Professeurs Assistants

Contact

FUZE Technologies Ltd 15 Clearfields Farm Wotton Underwood Aylesbury, HP18 0RE United Kingdom

courriel: contact@fuze.co.uk téléphoine: 44 (0)1844 239 432 réseaux sociaux: @fuzecoding site Internet: fuze.co.uk

Contrats de licence

-

Conventions d’artistes (Jessica Clayton / Alienoutcast) - (Luis Zuno / ANSIMUZ) - (Hasan Bayat / Bayat) - (Michael Lohr / Broken Vector) - (Colin Brown) - (David Silvera / FUZE Technologies Ltd) - (Dominik Gabriel / Cryptogene) - (Ajay Karat / Devil’s Garage) - (Valerij Golovin / DinV Studio) - (Eder Avila Muniz / EderMuniz) - (Emerald Eel Entertainment) - (Keith Fox / Fertile Soil Productions) - (Jason Perry / finalbossblues) - (Gijs De Mik) - (Kat Deak / FUZE Technologies Ltd) - (Kenney Vleugels / Pixeland Parkstad / Kenney) - (Graeme Houston / Kuro Ren) - (Hamza Cavus / MoonStar) - (Pipo / Hiroshi Suzuki) - (Jei Oakley / Pixelsnplay) - (Tomás Laulhé / Quaternius) - (Krzysztof Dycha / Ravenmore) - (Joshua Von Ros / RVROS) - (Lavinia / Selavi Games) - (Felipe Díaz Flores / Sinestesia) - (Stijn Van Wakeren) - (William S. Tice / Untied Games) - (Vadi Godfried / Vadi-Va)

Tous les éléments du jeu sont fournis dans le cadre de l’accord suivant :

Contrat de licence utilisateur pour les éléments du jeu inclus dans FUZE®4 Nintendo Switch

Définitions:

FUZE FUZE® est un environnement de développement conçu pour faciliter l’apprentissage du codage et de la programmation de jeux. Il est disponible sur plusieurs plates-formes matérielles.

ASSETS Illustrations 2D, cartes de tuiles, feuilles de sprite, modèles 3D, animations et matériaux 3D, polices d’écran, clips audio, musique et tout autre contenu lié au jeu, sous forme numérique, fournis avec l’intention d’être utilisés pour créer des jeux et/ou des applications dans l’environnement de développement **FUZE**.

Artiste [**TITULAIRE DES DROITS**] **TITULAIRE DES DROITS** Auteur original responsable de la création et/ou de la fourniture d'**ASSETS**.

ENTREPRISE FUZE Technologies Ltd. Le développeur et éditeur mondial exclusif de **FUZE**.

UTILISATEUR Le destinataire, ayant acquis l’utilisation de l’environnement de développement **FUZE** sur n’importe quelle plate-forme.

Préambule: Il est entendu que le **TITULAIRE DES DROITS** a accordé à l' **ENTREPRISE** le droit non exclusif d'inclure ses **ASSETS** avec **FUZE** sur ses plates-formes associées. Bien que ses **ASSETS** puissent être disponibles auprès d'autres sources, la licence attribuée à l'**ENTREPRISE** est propre à l'**ENTREPRISE** et ne doit pas être interprétée de manière erronée avec une autre licence fournie par le **TITULAIRE DES DROITS** à une autre partie.

Plus précisément: L'**ENTREPRISE** accorde à l'**UTILISATEUR** le droit d'inclure, de modifier et de manipuler les **ASSETS** pour une utilisation au sein de ses propres projets **FUZE**. L'**UTILISATEUR** doit clairement les attribuer aux **Titulaires de droits** et à **FUZE** en citant leurs noms dans le projet et, dans le cas de versions commerciales, dans tout support marketing destiné à promouvoir le projet.

Sans l'accord écrit explicite du **TITULAIRE DES DROITS**, l'**UTILISATEUR** ne peut pas redistribuer les **ASSETS** gratuitement ou commercialement en dehors des projets dans lesquels ils sont inclus. Les projets de l'**UTILISATEUR** ne peuvent pas permettre l'extraction des **ASSETS**.

L'**UTILISATEUR** ne peut sous-licencier ou redistribuer les **ASSETS** au-delà de la portée du présent contrat ou dans un format autome à tout tiers.

Pour éviter toute ambiguïté, l'**UTILISATEUR** peut inclure les **ASSETS** dans ses propres projets **FUZE**, sans toutefois procéder à une nouvelle distribution en dehors du champ d'application du présent contrat.

Il n'y a pas de date limite à cet accord. Il reste en vigueur à moins d'être révoqué par l'**ENTREPRISE**.

En cas de doute ou si vous souhaitez discuter de cette licence, veuillez envoyer un email à contact@fuze.co.uk

FUZE Contrat de **Licence** de l'utilisateur final **1) Définitions** Les définitions suivantes s'appliquent à cet accord:

a. "**FUZE**" FUZE Technologies Ltd est la société enregistrée au Royaume-Uni, numéro d'enregistrement : 8837428 b. "**LOGICIEL**" désigne la LICENCE du produit en vertu du présent contrat, l'application en code objet et/ou en formats binaires, inclusant les MISES A JOUR et les ASSETS. c. "**ASSETS**" désigne les contenus du "jeu vidéo" inclus dans un **PRODUIT FUZE** en plus du **LOGICIEL**, tels que des fichiers de modèle 3D et des fichiers objet, des fichiers audio, des fichiers vidéo et

des fichiers image, tels que ainsi que d'autres sons et modèles contenant de tels fichiers. d. "**APPAREIL**" désigne tout dispositif informatique physique ou virtuel électronique (ex. un ordinateur, un ordinateur portable, un poste de travail, une console de jeu vidéo, une tablette, un téléphone mobile, une instance de machine virtuelle, etc.). e. "**MISE A JOUR**" désigne à une version mise à jour du **LOGICIEL**. Une **MISE A JOUR** constitue une version modifiée, améliorée ou corrigée du **LOGICIEL**. Il n'inclut pas les nouvelles versions du **LOGICIEL**. f. "**LICENCE**" désigne la **LICENCE** attribuée à un **APPAREIL** spécifique appartenant au client après l'installation et, si nécessaire, l'activation du **LOGICIEL**. g. "**ARTISTE CONTRIBUTEUR**" désigne l'artiste responsable de la création d'**ASSETS** inclus avec le **LOGICIEL**. h. "**CANAUX OFFICIELS**" désigne le réseau de distribution des revendeurs agréés désignés par **FUZE**. i. "**UTILISATION COMMERCIALE**" désigne l'utilisation du **LOGICIEL** ou des **ASSETS** à des fins directes ou indirectes d'avantage financier (ex. par le biais de vente, de licence, de publicité, etc.).

2) Usage

a. Sous réserve des conditions énoncées dans le présent contrat et pour sa durée, **FUZE** vous (l'utilisateur) accorde le droit non exclusif et non transférable d'utiliser le **LOGICIEL** sur un **APPAREIL**. **FUZE** conserve la propriété, les droits d'auteur et autres droits de propriété relatifs au **LOGICIEL**. Vous (le client) reconnaissez la propriété de **FUZE** ainsi que tous les droits de propriété sur le **LOGICIEL**, les **ASSETS**, les copies de sauvegarde et la documentation. L'acheteur du **LOGICIEL** est seul responsable de la bonne utilisation contractuelle du **LOGICIEL**. b. Seuls les utilisateurs ayant acheté le **LOGICIEL** via les **CANAUX OFFICIELS** sont autorisés à recevoir les **MISES À JOUR**.

3) Installation et inscription a. En fonction de l'**APPAREIL** sur lequel vous installez le **LOGICIEL**, vous pouvez recevoir un numéro de **LICENCE** unique à saisir lors de l'installation. Une inscription en ligne peut également être requise avant l'activation du **LOGICIEL**. Si le nombre d'installations utilisateur dépasse le nombre d'installations autorisées spécifié dans la **LICENCE**, la **LICENCE** peut être désactivée par **FUZE**. Dans ce cas, l'utilisateur doit contacter **FUZE** pour demander sa réactivation.

4) Vérification de la LICENCE a. Le **LOGICIEL** nécessite généralement une connexion Internet pour être installé et activé.

5) Utilisation du LOGICIEL et des ASSETS à des fins commerciales a. Lorsque c'est techniquement possible, le **LOGICIEL** peut être utilisé à des fins commerciales, sous réserve des conditions énoncées à la clause 6.

6) ASSETS a. Lorsque c'est techniquement possible, les **ASSETS** peuvent être utilisés à des fins commerciales, sous réserve des conditions énoncées à la clause 6 et telles que spécifiées dans les accords **ARTISTE CONTRIBUTEUR**. Ceci s'applique aux jeux, aux démos, aux applications, aux programmes d'exemples et/ou aux versions modifiées des **ASSETS** ou à tout projet incluant des **ASSET**. b. **FUZE** accorde à l'utilisateur le droit d'inclure, de modifier et de manipuler les **ASSETS** pour une utilisation dans ses propres projets **FUZE**. L'utilisateur doit clairement les attribuer aux **ARTISTE CONTRIBUTEUR** et à **FUZE** au sein du projet et de tout matériel faisant la promotion de celui-ci. c. L'exploitation des **ASSETS** en dehors des réalisation personnelle, c'est-à-dire en dehors du **LOGICIEL**, est interdite. Pour éviter tout doute, les **ASSETS**, les programmes, les projets, les manuels, les démonstrations et exemples ne peuvent pas être extraits et utilisés séparément à des fins commerciales ou non commerciales. d. L'utilisateur ne peut, sans l'approbation écrite expresse du **TITULAIRE DES DROITS**, redistribuer des **ASSETS** gratuitement ou commercialement, en dehors du cadre de son inclusion au sein d'un projet de l'utilisateur. Les projets utilisateurs ne peuvent pas permettre l'extraction des **ASSETS**. e. Les utilisateurs ne peuvent sous-licencier ou redistribuer les **ASSETS** au-delà de la portée du présent contrat ou dans un format autonome à tout tiers.

7) Copie, location et redistribution a. Il vous est interdit de copier le logiciel sous licence et la documentation écrite, en tout ou en partie. Cela exclut votre droit de faire une copie numérique du logiciel à des fins de sauvegarde. Les copies de sauvegarde ne peuvent pas être redistribuées. b. Le **LOGICIEL** ainsi que la documentation écrite ne peuvent être loués ou prêtés sous une autre forme à un tiers, moyennant un paiement. Ceci s'applique également au prêt du **LOGICIEL** sous une forme préinstallée sur un **APPAREIL** offert commercialement à des tiers en échange d'un paiement. c. Vous ne pouvez apporter aucune modification au **LOGICIEL**, personnellement ou par des tiers. Vous ne pouvez pas désassembler le **LOGICIEL** en ses composants, ni modifier le code de l'objet, le décoder, le copier ou l'utiliser de quelque manière que ce soit autre que celle prévue dans le contrat.

8) Transfert de droits a. Le transfert des droits et obligations découlant du présent contrat à des tiers n'est autorisé que sur autorisation de **FUZE**, à l'exception du transfert personnel du **LOGICIEL** légalement acquis par le propriétaire légitime. En cas de cession de cette manière de la propriété du **LOGICIEL** légitimement acquis, le propriétaire initial est obligé de détruire toutes les copies de sauvegarde et de supprimer l'installation. Un transfert numérique d'un **LOGICIEL** (un téléchargement) est interdit.

9) Garantie et responsabilité a. Vous êtes informé que les logiciels, les **ASSETS** et la documentation associée peuvent contenir des erreurs et qu'il est impossible de développer des programmes de traitement de données de manière à ce qu'ils

-

soient exempts d'erreurs pour tous les scénarios d'utilisation et toutes les exigences du client, ou sans erreur en conjonction avec tous les programmes et matériels tiers. **FUZE** ne fournit aucune garantie ni explicite ni implicite quant aux fonctionnalités proposées et leur employabilité au sein d'applications spécifiques prévues par le client. b. En ce qui concerne les produits et services payants, **FUZE** est uniquement responsable des légers dommages mineurs causés par lui-même ou son (ses) assistant(s) en cas de violation de ses obligations, même s'il s'agit d'une obligation extracontractuelle, dont le respect revêt une importance particulière pour être en conformité avec l'utilisation contractuelle (obligation fondamentale), ainsi que dans les cas de dommages à la vie, au corps et à la santé. c. En cas de non-respect d'une obligation fondamentale, la responsabilité est limitée aux dommages auxquels il faut normalement s'attendre dans le cadre du présent contrat en l'absence d'intention ou de faute lourde ou si la responsabilité de **FUZE** est due à une blessure mortelle, blessure physique ou risques pour la santé. d. **FUZE** ne saurait être tenu pour responsable des dommages pouvant être maîtrisés par l'autre partie contractante ou que celle-ci aurait pu prévenir en prenant des mesures raisonnablement prévisibles. **FUZE** n'est pas responsable des pertes de données. e. En tout état de cause, la responsabilité de **FUZE** est limitée à quatre fois le montant payé pour les frais de **LICENCE** par le client. Cette exclusion ne s'applique pas aux dommages causés intentionnellement ou par négligence grave de la part de **FUZE**. f. Dans le cas de produits et services payants, la garantie contre les défauts matériels et les défauts de propriété est limitée à la dissimulation frauduleuse de défauts par **FUZE** en contrepartie de l'obtention de la licence gratuite du produit. g. La responsabilité légale en cas de dommages personnels et de dommages-intérêts en vertu de la loi sur la responsabilité du fait des produits reste inchangée. h. Une modification de la charge de la preuve au détriment du client n'est pas liée à la disposition précédente. i. Dans la mesure où le **LOGICIEL** contient des fonctions qui fonctionnent via un serveur en ligne, **FUZE** conserve le droit de mettre fin à l'offre à tout moment. La disponibilité n'est pas garantie.

10) Conditions de licence d'autres fabricants a. Si le **LOGICIEL** contient des logiciels supplémentaires d'autres fabricants ou si un logiciel supplémentaire doit être intégré, le respect des conditions d'utilisation et de licence du fabricant du logiciel supplémentaire livré est également obligatoire. Si le **LOGICIEL** contient un logiciel supplémentaire, vous pouvez afficher les conditions d'utilisation et de licence respectives dans le fichier correspondant.

11) Support a. ** FUZE ** offre un support Internet électronique pendant la période de garantie. Cela englobe la clarification des questions d'installation et des problèmes d'installation par Internet ou par courrier électronique. Le support est à la seule discrétion de **FUZE** et n'est associé à aucune garantie.

-

12) Autre a. Cet accord constitue l'intégralité de l'accord des parties concernant l'objectif du contrat. Aucun accord collatéral ne doit pas exister. Aucune déclaration verbale ou écrite faite par **FUZE** ou tout employé de **FUZE** ne peut altérer ou remettre en question la validité du présent contrat **LICENCE**.

13) Validité des conditions contractuelles a. Si une ou plusieurs des conditions de ce contrat étaient ou devenaient invalides, cela n'affecterait pas la validité du contrat restant. Une disposition de substitution remplacera la condition invalide, de sorte à se rapprocher au plus de l'objectif recherché. Le contrat est soumis aux lois du Royaume-Uni.

GETTING STARTED

Avertissement sur l'épilepsie

Ce programme permet aux utilisateurs de créer des images clignotantes.

Certaines personnes sont susceptibles de faire des crises lorsqu'elles sont exposées à certaines images, notamment des lumières clignotantes ou des motifs visuels.

Le risque de crises d'épilepsie photosensible peut être réduit en prenant les précautions suivantes :

Jouez dans une pièce bien éclairée.

Ne jouez pas si vous êtes somnolent ou fatigué.

Éloignez-vous de l'écran pour limiter sa présence dans votre champ de vision.

Si vous ressentez des sensations d'étourdissements, de vertiges, de nausées, de troubles de la vue, de tremblements des yeux ou du visage, de saccades ou de tremblements des bras ou de jambes, de désorientation, de confusion ou de perte de conscience momentanée, cessez immédiatement de jouer et consultez un médecin.

Introduction

-

Bonjour ! Bienvenue dans **FUZE^4 Nintendo Switch**. Félicitations pour votre excellent achat !

Si vous êtes novice en matière de codage et que vous ne savez pas par où commencer, vous êtes au bon endroit ! Dans cette introduction, nous aborderons quelques points importants à garder à l'esprit lorsque vous utiliserez **FUZE**.

Si vous êtes déjà un programmeur expérimenté, nous vous recommandons de vous lancer et d'activer vos muscles de codeur ! Vous pouvez trouver une description détaillée de chaque fonction, mot-clé et opérateur dans le guide de référence des commandes.

Le codage est une compétence incroyable. Tous les appareils électroniques, jeux vidéo et logiciels que vous avez déjà utilisés fonctionnent grâce au code. Sans lui, le monde serait très différent.

Avant de plonger dans quelques détails du logiciel, effectuons une mise au point.

Apprendre à coder, c'est comme apprendre un superpouvoir. Vous pouvez trouver qu'il est difficile de faire les choses correctement au début, vous ferez beaucoup d'erreurs, vous pourriez même penser que vos nouveaux pouvoirs ne fonctionnent pas pour vous. Comme pour toute chose dans la vie, c'est en vous exerçant que vous vous améliorerez. Un proverbe français ne dit-il pas "c'est en forgeant que l'on devient forgeron" ?

Continuez à perfectionner vos compétences, à résoudre vos problèmes et à mener à bien vos projets. Bientôt, vous serez un superhéros du codage !

L'apprentissage prend du temps, alors ne vous découragez pas si vous vous trompez! Chaque erreur (ou bugs comme nous les appelons) que vous corrigez fait de vous un meilleur programmeur.

FUZE^4 Nintendo Switch vous offre un environnement léger pour créer tout ce que vous voulez. Mais par où commencer?

Eh bien, voici quelques points importants à garder à l'esprit.

Langages informatiques

-

Vous avez peut-être entendu dire que les ordinateurs sont des choses très compliquées. C'est vrai dans une certaine mesure, mais on dit aussi qu'à certains égards, ils sont très simples et logiques.

Un ordinateur **distingue deux états** usuellement nommés *Allumé* et *Éteint*, *Vrai* et *Faux* ou *1* et *0*.

Le cerveau d'un ordinateur est appelé **CPU** (en anglais Central Processing Unit). Il est composé d'un très grand nombre de commutateurs qui sont simplement *Allumés* ou *Éteint*s.

Là où ça se complique, c'est qu'il y en a vraiment beaucoup. Des milliards et des milliards en fait.

En réalité, dès que vous utilisez un ordinateur pour jouer à un jeu, envoyer un message à vos amis, surfer sur le Net ou faire vos devoirs, vous *changez des milliards et des milliards de commutateurs* de façon incroyablement rapide.

Comment votre ordinateur parvient-il à faire toutes ces choses en utilisant *seulement* des 1 et des 0 ? Un gars très intelligent appelé Gottfried Leibniz a inventé ce que l'on appelle le **Système binaire**, et il est vital pour tous les ordinateurs. Nous parlerons de **binaire** plus en détail plus ultérieurement.

Maintenant, il serait assez ennuyeux et incroyablement difficile d'écrire un programme entier en utilisant les 1 et les 0, aussi des personnes incroyablement intelligentes ont développé **les langages informatiques** pour "parler" à l'ordinateur de manière plus compréhensible pour nous.

Il existe de très **nombreux langages informatiques** dans le monde : des milliers et des milliers... Ils sont très différents les uns des autres et sont conçus pour des raisons différentes.

Le **langage** que vous utiliserez ici s'appelle **FUZE** !

FUZE est similaire aux **langues** que vous pourriez utiliser si vous étudiez l'informatique à l'école, ou même celles utilisées par des programmeurs professionnels. Cependant, avec FUZE, nous nous sommes efforcés de rendre l'apprentissage et de l'utilisation simple et l'intuitive.

Formatage

-

Lorsque nous écrivons du code, nous pouvons le présenter de différentes manières, nous appelons cela **le formatage**.

Voici un exemple de code peu soigné ci-dessous :

```
1. loop
2.               ink(fuzepink)
3. print("BONJOUR"  )
4.
5.        update(          )
5.     sleep(    1)
6.
7.          repeat
```

Nous avons beaucoup d'espaces aléatoires et de lignes vierges sans raison, ***mais***, et il est important de le dire, le code ci-dessus fonctionnera *très bien*.

Le programme est une petite boucle. Nous imprimons le mot "BONJOUR" dans une jolie couleur rose, attendons 1 seconde, puis recommençons (en boucle).

Le même code formaté différemment fonctionnera *exactement de la même manière*. Voir l'exemple ci-dessous :

```
1. loop ink(fuzepink) print("BONJOUR") update() sleep(1) repeat
```

FUZE n'accorde pas d'importance au formatage de votre programme qu'il soit écrit sur une seule ligne ou plusieurs. Par contre, à mesure que vos programmes grandissent et se complexifient, un formatage soigné facilite la correction des bogues et les modifications de votre code !

La façon dont nous formatons notre code a un impact important sur la façon dont nous le lisons. Ainsi apprendre dès le début à formater correctement notre code nous aidera grandement dans le futur.

Comparez les exemples précédents avec la mise en forme du code ci-dessous :

```
1. loop
2.     ink(fuzepink)
3.     print("BONJOUR")
5.     update()
```

```
6.     sleep(1)
7. repeat
```

N'a-t-il pas l'air plus net, pourrait-on dire plus logique ?

Maintenant, nous pouvons facilement voir où notre boucle commence et se termine, et tout ce qui est entre les deux est **indenté**.

Rappelez-vous: ce code ci-dessus sera **exécuté** *exactement de la même manière* que les exemples précédents. La seule différence est qu'il est plus facile à lire et à modifier.

Les projets et les tutoriels dans FUZE sont tous **formatés** de cette façon. Vous n'êtes pas obligé de nous copier. En fait, vous pouvez adopter le style de **formatage** qui fonctionne pour vous !

Syntaxe

Voici un mot étrange que vous pouvez ne pas connaître !

La **syntaxe** décrit la **structure des instructions** dans un langage informatique. Certaines instructions doivent être structurées d'une manière particulière si nous voulons qu'elles fonctionnent. Prenons l'exemple ci-dessous :

```
1. prin t("Voyez-vous l'erreur ?")
```

Pouvez-vous repérer l'erreur que nous avons faite?

Nous avons commis une erreur cruciale dans la **syntaxe**. Nous mettons un espace entre "prin" et "t" dans le mot "print". FUZE va lire ceci comme **deux** instructions séparées.

Si nous corrigeons cette erreur **syntaxe**, nous obtenons:

```
1. print("Voyez-vous l'erreur ?")
```

Maintenant, FUZE sait exactement quoi faire et nous n'obtiendrons aucune erreur.

-

Autre exemple très similaire :

```
1. print("Et maintenant?)
```

Celle-ci est peut-être plus difficile à voir, mais nous avons commis une autre erreur de **syntaxe**.

Pouvez-vous la repérer ?

Un {code: "} manque avant le dernier crochet ! Sans lui, FUZE ne peut déterminer ce que vous voulez afficher.

Certaines instructions doivent être présentées de manière particulière, et cela est souvent lié à la ponctuation. Vérifiez *toujours* l'emplacement de vos virgules, guillemets, parenthèses et crochets.

Fonctions et mots-clés

Pendant les tutoriels, vous allez voir quelques mots les uns après les autres.

Pour les projets à venir, vous devez savoir ce que nous entendons par **fonction**. Sans entrer dans les détails qui seront abordés dans les didacticiels à venir, jetons un coup d'œil à la ligne de code ci-dessous :

```
1. print("Print est une fonction.")
```

Lorsque nous voulons afficher du texte à l'écran, nous utilisons la **fonction** {code:print()}.

Avez-vous remarqué les parenthèses après le mot {code: print} ? Elles vous indiquent que nous utilisons une **fonction** ! Dans **FUZE**, les **fonctions** apparaissent également en bleu clair.

Les **fonctions** nécessitent généralement certaines informations entre parenthèses pour fonctionner. Par exemple, la *fonction* {code: print ()} a besoin d'un élément entre parenthèses pour l'afficher à l'écran.

En voici une autre :

-

```
1. ink(green)
```

Il s'agit de la **fonction** {code: ink ()}. Nous l'utilisons pour changer la couleur du texte à l'écran. Cette fois, nous mettons une couleur entre parenthèses !

Dès que vous voyez le nom d'une **fonction** dans les tutoriels FUZE, des parenthèses la suivent, comme cela : {code: print ()}

Pour être aussi clair que possible : les **fonctions** sont *toujours* suivies par des parenthèses.

Les **mots-clés** sont un peu différents. Ils n'utilisent pas de parenthèses et, dans **FUZE^4 Nintendo Switch**, ils apparaissent en rouge/rose. Regardez ci-dessous :

```
1. loop
2. repeat
```

Les deux lignes ci-dessus constituent une **boucle vide**. {code: loop} et {code: repeat} sont des **mots-clés**

Gardez les yeux ouverts dans les projets à venir pour reconnaitre les **fonctions** et les **mots-clés**.

Commençons !

Félicitations pour la lecture de toutes les informations ci-dessus. Vous êtes maintenant tout à fait prêt pour vous lancer dans les projets du tutoriel et commencer votre parcours vers la maîtrise du code ! Dans l'attente de vous retrouver dans les tutoriels, lancez-vous et prenez du plaisir à coder !

Utiliser FUZE

Lorsque vous chargez **FUZE^4 Nintendo Switch** pour la première, vous êtes accueilli par l'écran du menu principal :

-

À partir de là, vous pouvez accéder à tous les outils et fonctionnalités proposés par FUZE.

Barre des Commandes

Avant d'examiner chaque élément de menu, examinez de plus près le bas de votre écran :

Cette petite barre très utile apparaît au bas de chaque écran de **FUZE^4 Nintendo Switch**. Elle s'appelle la **Barre des Commandes** et vous indique tous les contrôles auxquels vous avez actuellement accès.

Assurez-vous de consulter la **Barre des Commandes** pour vous guider dès que vous ne savez plus quel contrôle utilser !

Code (Code)

Avec le curseur de sélection sur le bouton "Code", appuyez sur le bouton A de votre contrôleur Joy-Con (touche Entrée sur le clavier USB) pour accéder à l'éditeur de code.

-

L’éditeur de code est l’endroit où vous passerez le plus de temps dans FUZE. C’est l’endroit où nous allons écrire un programme ! Dans l’éditeur de code, vous pouvez également accéder au Navigateur de Média pour charger des assets (sons, images, modèles 3D, ...) ou consulter les didacticiels.

Il y a beaucoup de contrôles auxquels il faut s’habituer dans l’éditeur de code. Gardez un œil sur la **Barre des Commandes** en bas de l’écran!

Média (Media)

En appuyant sur A sur le bouton “Media”, vous accédez au Navigateur de Média de **FUZE^4 Nintendo Switch**.

Avec lui, vous pouvez parcourir la vaste collection de ressources visuelles et audio à votre disposition. Peut-être tomberez-vous sur quelque chose de génial pour votre prochaine idée de jeu !

Programmes (Programs)

Le bouton “Programs” dans le menu principal donne accès aux programmes existants de **FUZE^4 Nintendo Switch**. À partir de là, vous pouvez démarrer un nouveau projet, charger l’un des programmes de démonstration fournis, partager ou charger l’un de vos propres projets. Vous devrez d’abord en faire, bien sûr !

Projet (Project)

Le menu de “Project” de **FUZE^4 Nintendo Switch** contient des informations sur le projet actuellement chargé. Vous pouvez modifier les détails de votre projet, enregistrer et commencer de nouveaux projets.

Amis (Friends)

En sélectionnant l’option “Friends” dans le menu principal, vous accédez à une page affichant tous vos amis Nintendo Switch. S’ils disposent également de **FUZE^4 Nintendo Switch**, vous pouvez télécharger leurs projets partagés à partir de cet écran.

-

Outils (Tools)

Vous voulez faire votre propre niveau en utilisant les assets existant ? Peut-être vous voulez aussi créer vos propres assets ! Quoi qu'il en soit, le menu "Tools" est l'endroit que vous recherchez.

En appuyant sur le bouton "Tools", vous pourrez choisir entre l'éditeur de carte ou l'éditeur d'image. Utilisez l'Éditeur de cartes pour créer des cartes de niveau à partir des ressources de **FUZE^4 Nintendo Switch**. Utilisez l'éditeur d'images pour créer vos propres ressources !

REMARQUE : Les cartes et les images créées à l'aide de l'éditeur de cartes ou de l'éditeur d'images sont enregistrées **dans un projet spécifique**. Lorsque vous ouvrez l'éditeur de carte ou d'image, il vous sera d'abord demandé de sélectionner un projet.

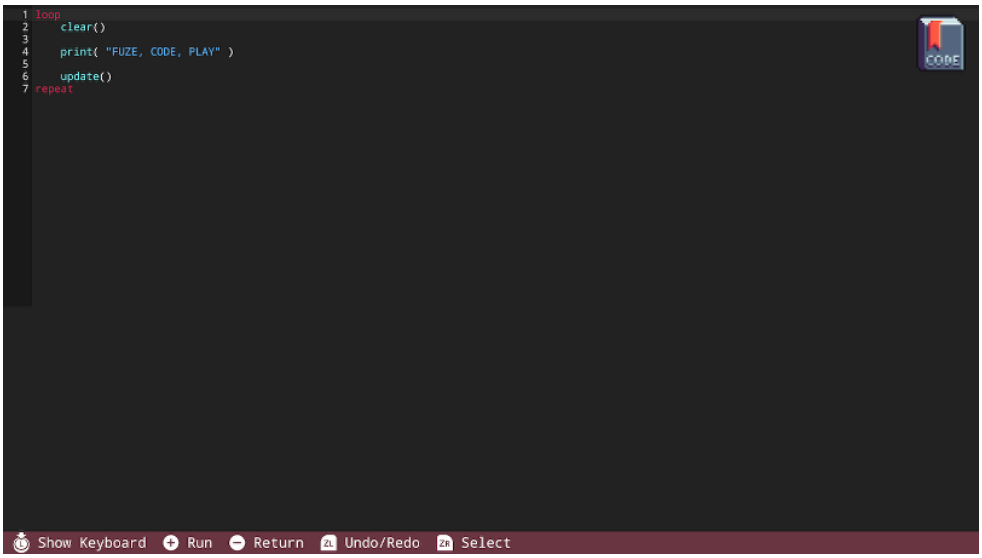
Paramètres (Settings)

Comme vous pouvez l'imaginer, le bouton "Settings" du menu principal vous amène à la page des paramètres de **Fuze^ 4 Nintendo Switch**. À partir de là, vous pouvez adapter FUZE à votre goût, en modifiant l'apparence et le comportement de l'application.

Éditeur de code

Pour entrer dans l'éditeur de code, sélectionnez le bouton "Code" dans le menu principal. Il ressemble à ceci :

-

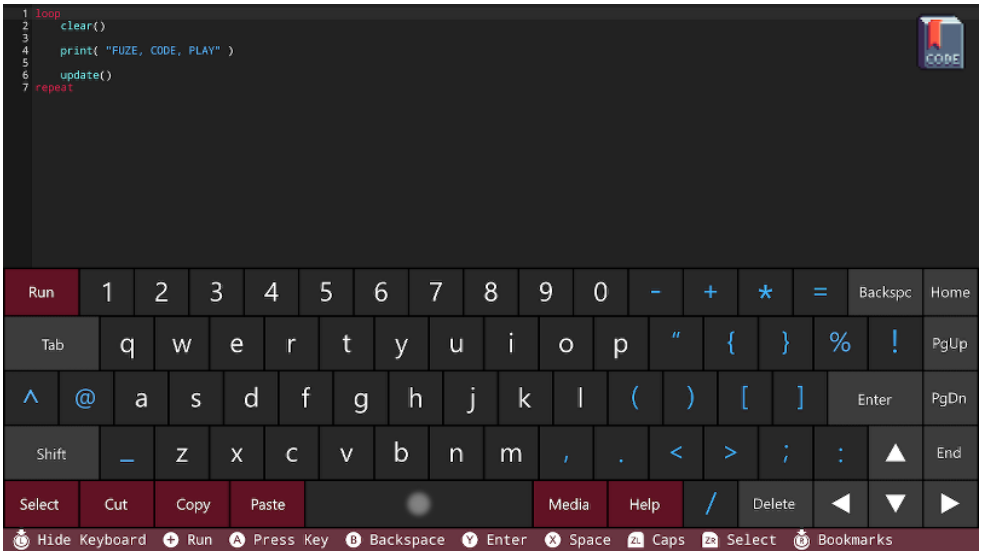


Jetez un coup d'œil à la **Barre des Commandes** en bas de l'écran. Ici vous pouvez trouver des informations sur les boutons pour vous aider à naviguer dans l'éditeur.

Appuyez sur le bouton **+** pour lancer l'**exécution** du programme. Appuyez sur le bouton **-** pour revenir au menu principal.

Clavier Virtuel

Appuyez sur le stick gauche pour ouvrir le clavier :



-

Vérifiez la **Barre des Commandes** en bas de l'écran et vous verrez qu'elle a été modifiée pour afficher les commandes du clavier à l'écran. Assurez-vous de vous référer à la **Barre des Commandes** si vous ne savez pas comment interagir avec FUZE !

Avec le clavier à l'écran, déplacer le stick gauche vous permettra de sélectionner les touches à presser. Utilisez le bouton A pour presser une touche.

En appuyant sur le bouton "Media" du clavier, vous accédez rapidement et facilement au Navigateur de Média si vous recherchez des ressources à utiliser.

De même, en appuyant sur le bouton "Help" (Aide) du clavier, vous ouvrirez l'Aide dans l'éditeur :

Lorsque l'Aide est ouverte dans l'éditeur, la **Barre des Commandes** est modifiée pour afficher les commandes de l'Aide. À partir de là, vous pouvez naviguer vers n'importe quelle page et la consulter en gardant votre code à l'écran.

Si vous souhaitez entrer ou modifier du code avec l'Aide à l'écran, cliquez dans le stick gauche pour afficher le clavier.

Clavier USB

Vous pouvez connecter un clavier USB à votre console Nintendo Switch pour rendre la frappe plus rapide et plus précise.

-

Lorsque vous utilisez un clavier USB, il existe un certain nombre de raccourcis très utiles à connaître !

En haut du clavier, vous trouverez une rangée de touches F1, F2, ... (touches de fonction).

Elles sont associées à des actions dans **FUZE^4 Nintendo Switch**. Regardez la liste ci-dessous :

F1 - Ouvre l'Aide. Si vous êtes dans l'éditeur de code, l'Aide de l'éditeur s'ouvrira. Si vous êtes dans le menu principal, cela vous mènera à la section principale de l'Aide.

F2 - La touche F2 affiche le **Navigateur de Média** ("Media").

F3 - F3 **sauvegarde** votre projet.

F5 - F5 lance l'**exécution** de votre programme. Elle fonctionne où que vous soyez dans FUZE, sauf si vous modifiez la description d'un programme ! Lorsque vous modifiez la description d'un programme, appuyez sur **F5** pour valider et retourner au programme.

F6 - Affiche l'**Éditeur d'Image** ("Image Editor").

F7 - Affiche l'**Éditeur de Carte** ("Map Editor").

F8 - Cette touche donne accès aux **Paramètres** ("Settings").

F9 - Affiche l' **Éditeur de Code** ("Code").

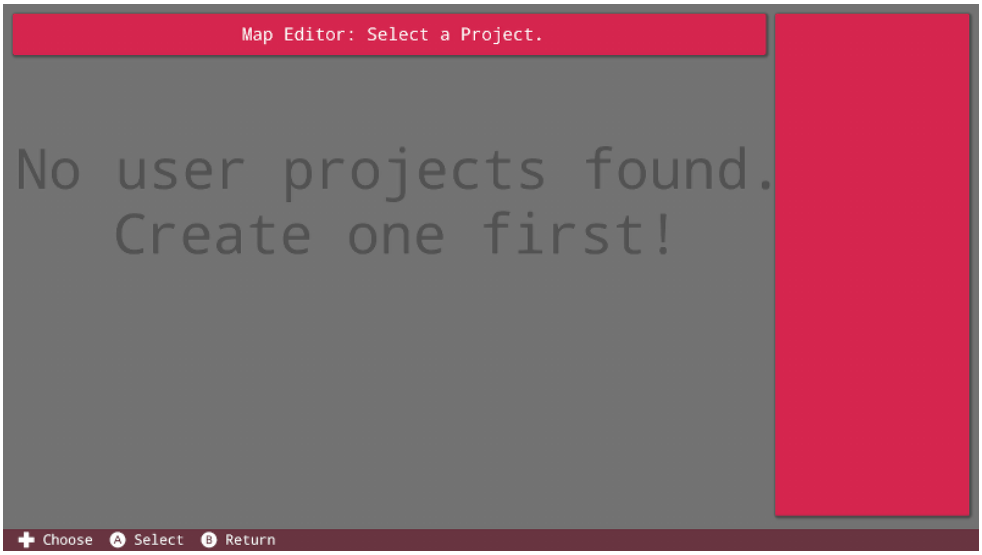
F10 - Ramène au **Menu Principal**.

Éditeur de Carte

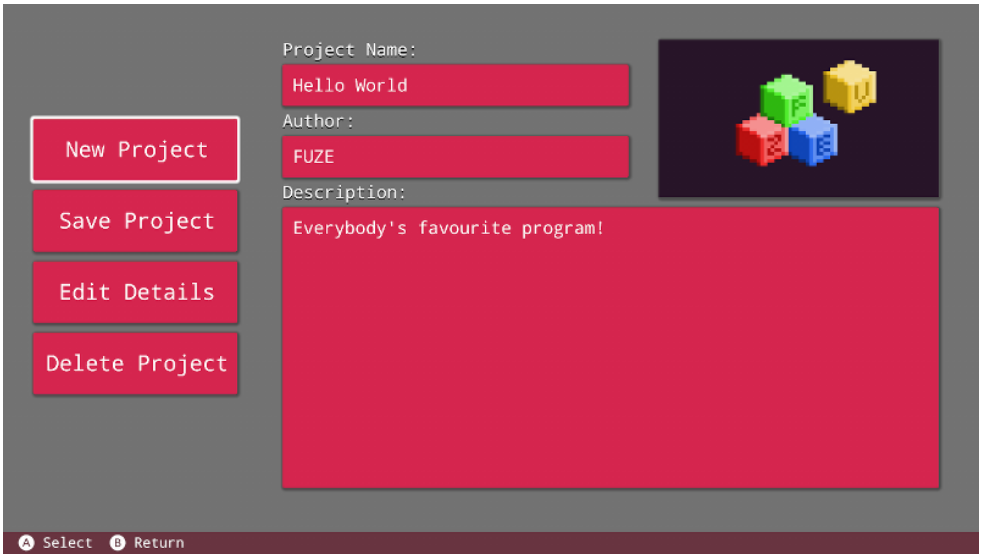
<>

-

L'éditeur de carte **FUZE^ 4 Nintendo Switch** est conçu pour faciliter la création de vos propres cartes. Accédez à l'éditeur de carte en cliquant sur l'icône "Tools" (Outils) du menu principal, puis sélectionnez "Map Editor" (Éditeur de Carte). Si c'est la première fois que vous utilisez FUZE, vous verrez l'écran ci-dessous :



Comme nous n'avons aucun projet sauvegardé, nous ne pouvons rien faire. Commençons par créer un projet pour pouvoir stocker une carte. Retournez au menu principal et cliquez sur le bouton "Projet". L'écran ci-dessous s'affiche :



Il comporte les détails du projet pour le programme **FUZE** par défaut, nommé "Hello World" (Bonjour le Monde).

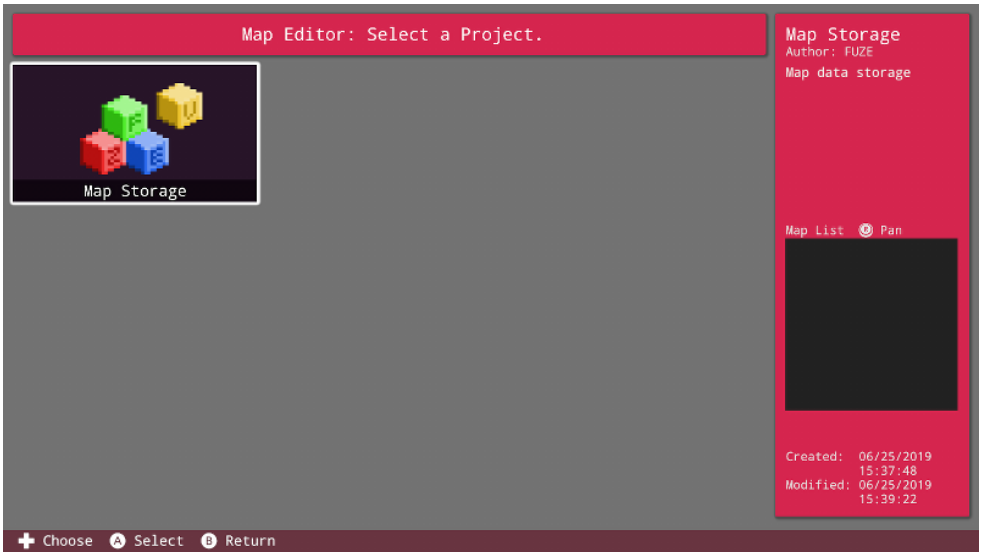
-

Sélectionnez le bouton “New Projet” (Nouveau Projet) sur la gauche pour créer un nouveau fichier de projet dans l’espace de stockage. Cela nous permettra par la suite de créer des cartes pour ce projet.

Entrez le titre, l’auteur et la description de votre premier projet.

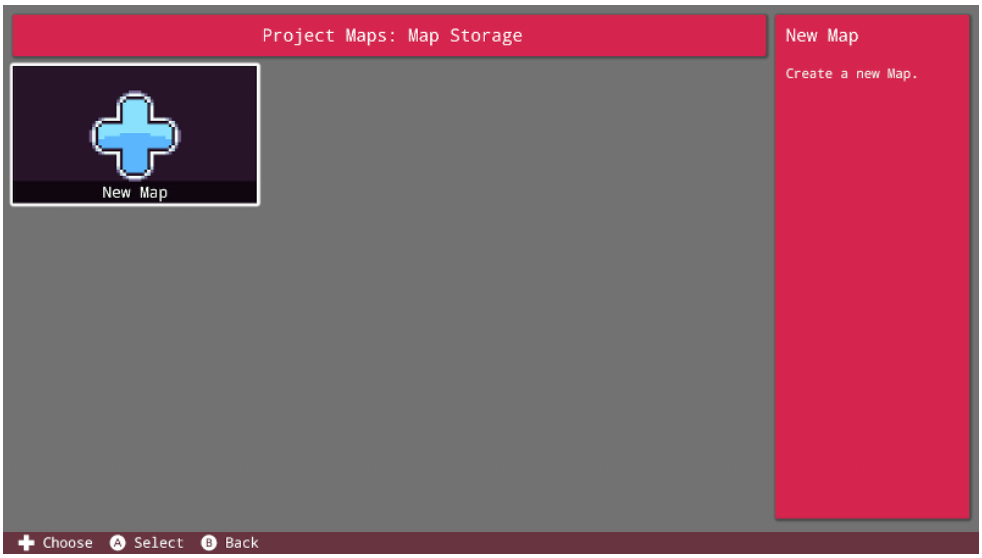
C’est une bonne idée de créer un projet dans lequel regrouper toutes vos cartes. C’est un moyen pratique de savoir où se trouvent toutes vos cartes pour tout usage ultérieur. Pourquoi ne pas l’appeler “Stockage des Cartes” ou “Atlas” ?

Une fois le projet créé, vous serez dirigé vers l’Éditeur de Code. Revenez au menu principal en utilisant le bouton moins du contrôleur Joy-Con, puis cliquez sur l’icône “Tools” (Outils), puis sur “Map Editor” (Éditeur de carte) :



Comme vous pouvez le constater, nous pouvons maintenant voir notre projet nouvellement créé. Cliquez sur l’icône du projet et vous passerez à la fenêtre suivante :

-

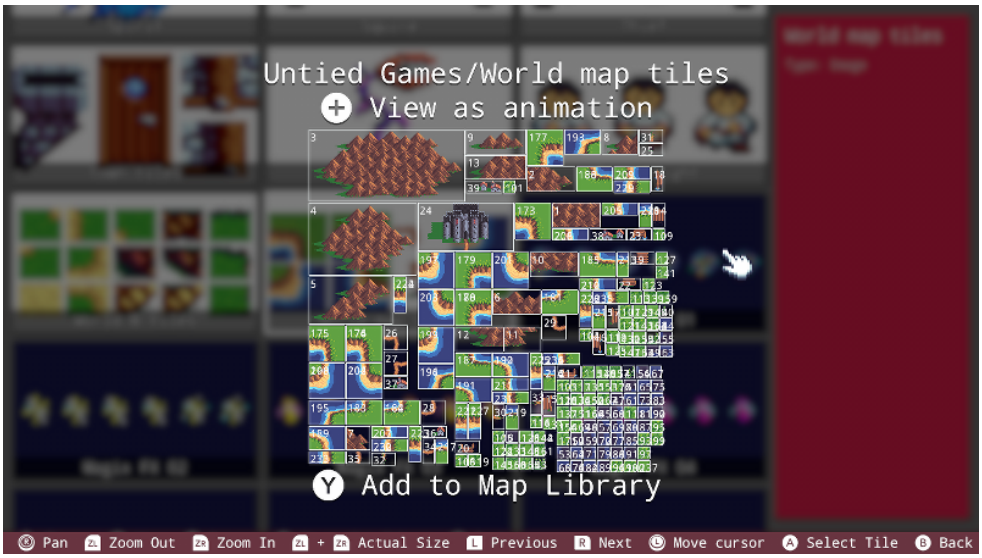


C’est dans cette fenêtre que nous pouvons voir toutes les cartes stockées au sein de ce projet. Puisque nous n’en avons pas encore, cliquons sur le bouton “New Map” (Nouvelle carte) pour commencer !

Vous serez invité à saisir un nom pour votre carte, puis à appuyer sur le bouton plus pour confirmer.

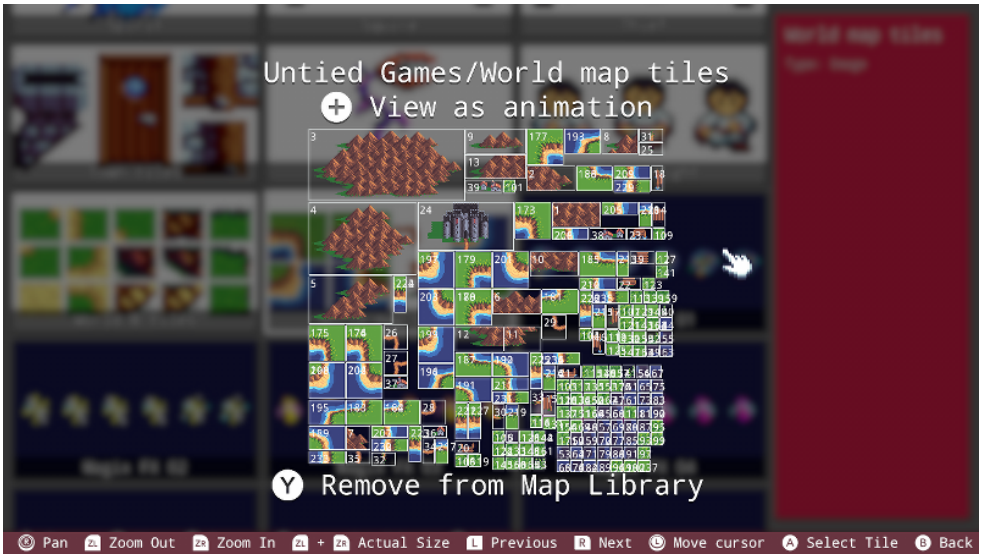
Nommez votre carte et appuyez sur le bouton plus. Le message suivant s’affiche : “Choose assets to use in your map!” (Choisissez des assets à utiliser dans votre carte !). En appuyant sur “OK”, vous accéderez au Navigateur de Média de **FUZE** afin de sélectionner des assets. Pour cet exemple, nous allons utiliser quelques assets du très talentueux “Untied Games”. Sélectionnez l’icône de l’artiste “Untied Games” dans le Navigateur de Média :

-



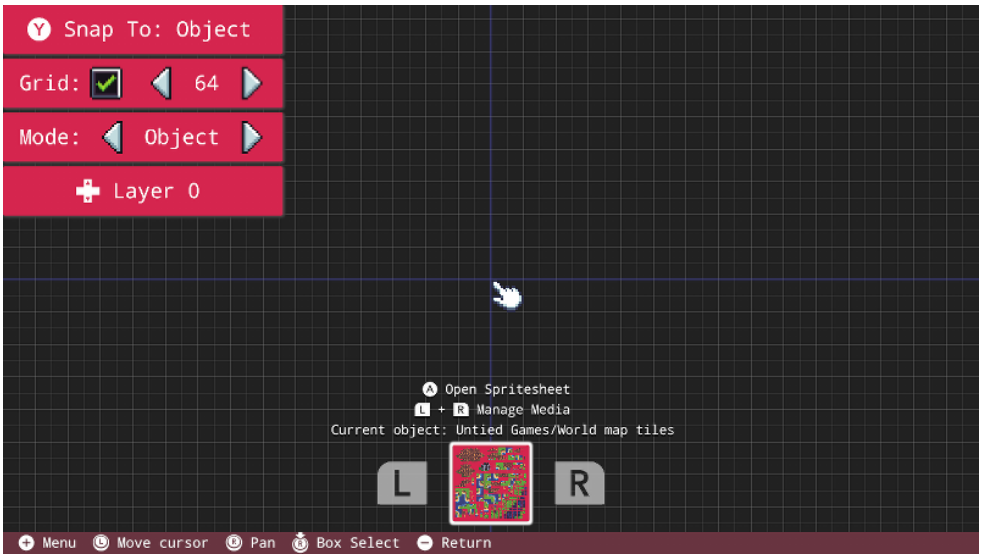
Faites défiler l'écran vers le bas et retrouvez la feuille de tuiles intitulée "Untied Games/World map tiles". Ouvrez cette feuille de tuiles en appuyant sur le bouton A, puis sur le bouton Y pour l'ajouter à votre bibliothèque de cartes, comme indiqué en bas de l'écran.

Une fois le bouton Y pressé, vous remarquez que le texte en bas de l'écran change. Nous pouvons appuyer à nouveau sur le bouton Y pour supprimer cet élément de notre bibliothèque de cartes.



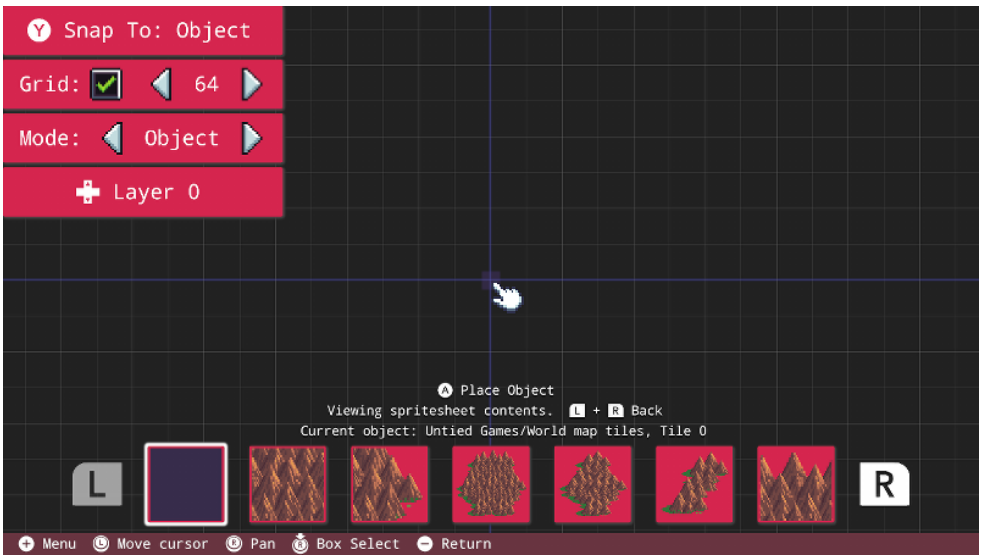
Nous avons maintenant la possibilité de revenir dans le Navigateur de Média et d'ajouter de nouvelles ressources à utiliser en conception de carte. Pour le moment, nous allons nous en tenir à cette feuille de tuiles. Appuyez sur le bouton B pour revenir en arrière, puis appuyez sur le bouton plus pour accéder à l'éditeur de carte :

-



Nous y voilà ! Maintenant que nous sommes dans l'éditeur de cartes, jetez un coup d'œil au bas de notre écran. Nous avons deux options. Nous pouvons ajouter davantage d'assets à notre bibliothèque en appuyant simultanément sur les boutons L et R (Manage Media). Nous pouvons également ouvrir la feuille de tuiles (Open Spritesheet) sélectionnée pour commencer à dessiner avec ses tuiles.

Puisque nous n'avons qu'une seule feuille de tuiles chargée dans notre bibliothèque, nous ne pouvons voir qu'une seule option en bas de l'écran. Si plus de ressources étaient chargées dans la bibliothèque de cartes, elles seraient affichées ici. Ouvrons la feuille de tuiles pour commencer à dessiner la carte. Appuyez sur le bouton A pour ouvrir feuille de tuiles :



-

Avant de commencer à placer des tuiles, il est important de jeter un coup d'œil aux options de contrôle situées dans le coin supérieur gauche de l'écran. Appuyez sur le bouton Y pour changer la manière dont les tuiles se lient ensemble (Snap To:). En choisissant "Object" (Objet), vous pourrez placer les tuiles les unes à côté des autres. L'accrochage sur "Grille" force les tuiles à se placer sur les emplacements de la grille, tandis que l'option "None" (Aucun) permet un placement totalement libre.

L'option "Grille" juste en dessous indique la taille des emplacements de la grille. Un nombre plus élevé permettra un placement plus précis sur la grille.

L'option 'Mode' juste en dessous vous permet de changer le mode d'édition. Par défaut, nous sommes en mode "Object" (objet). Cela signifie que nous posons simplement des tuiles. En appuyant sur les flèches de cette case, nous pouvons changer notre mode d'édition en Collision. Cela deviendra utile quand nous aurons une carte ! Alors faisons-en une.

La dernière option à aborder ici se trouve juste en dessous de la sélection du mode. En utilisant les boutons directionnels haut et bas, vous pouvez changer le calque sur lequel vous placez les tuiles. C'est idéal pour construire des cartes complexes.

Une fois la feuille de tuiles ouverte, nous pouvons voir toutes les tuiles individuelles en bas de l'écran. Utilisez les boutons L et R pour naviguer vers la tuile souhaitée.

Appuyez sur le bouton A pour placer une tuile. Si vous faites une erreur et devez supprimer une tuile, il suffit de déplacer le curseur pour mettre la tuile en surbrillance et d'appuyer sur le bouton B.

-

Nous avons ici une petite section de carte du monde que nous pouvons potentiellement utiliser. Ajoutons des données de collision en utilisant le mode d'édition Collision.

Déplacez le curseur sur les flèches affichées dans les cases en haut à gauche et appuyez sur le bouton A pour passer en mode Collision :

Notez que le bas de l'écran a changé et que nous sommes maintenant en mode Collision. Maintenant, nous avons la possibilité de délimiter les zones pour lesquelles nous voulons adopter un comportement particulier. Par exemple, il est possible d'empêcher le joueur d'entrer dans l'eau. Pour y parvenir facilement, nous pouvons simplement dessiner des boîtes de collision autour des zones concernées :

-

Il est très utile en mode Collision d'utiliser le mode d'accrochage "Object" (Objet). Déplacez le curseur sur votre zone de collision, puis appuyez sur le bouton A pour commencer à placer une boîte.

Déplacer le curseur ajustera la position de votre boîte. Utilisez les boutons de direction pour régler avec précision la taille de votre boîte de collision.

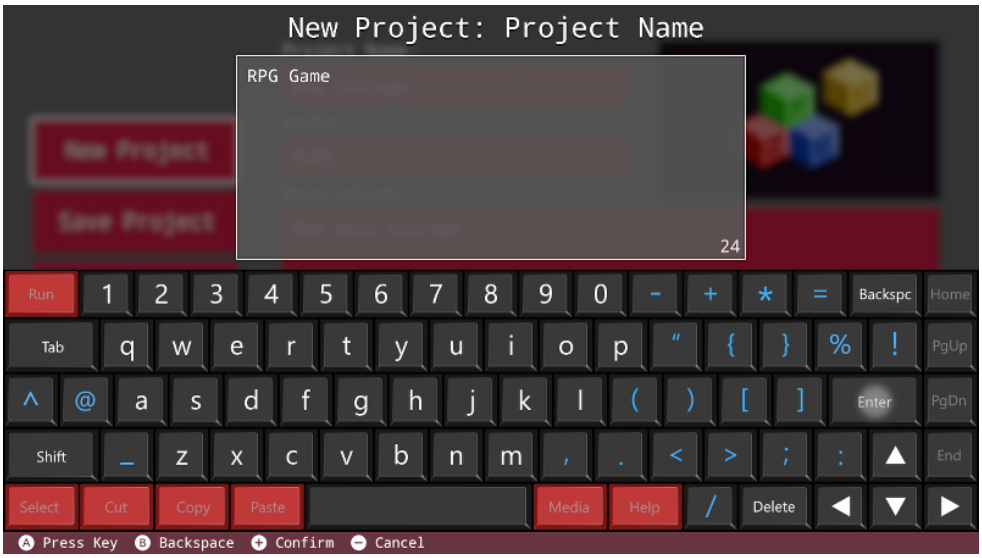
C'est une bonne idée de faire chevaucher des boîtes de collision afin d'éviter tout comportement étrange avec les coins. Maintenant que nous avons défini l'ensemble de données de collisions, nous sommes prêts à utiliser cette carte !

Pour enregistrer votre carte, appuyez simplement sur le bouton moins pour revenir au Menu Principal. Vous verrez l'icône de sauvegarde **FUZE** apparaître en haut à droite.

Copier une carte dans un nouveau projet

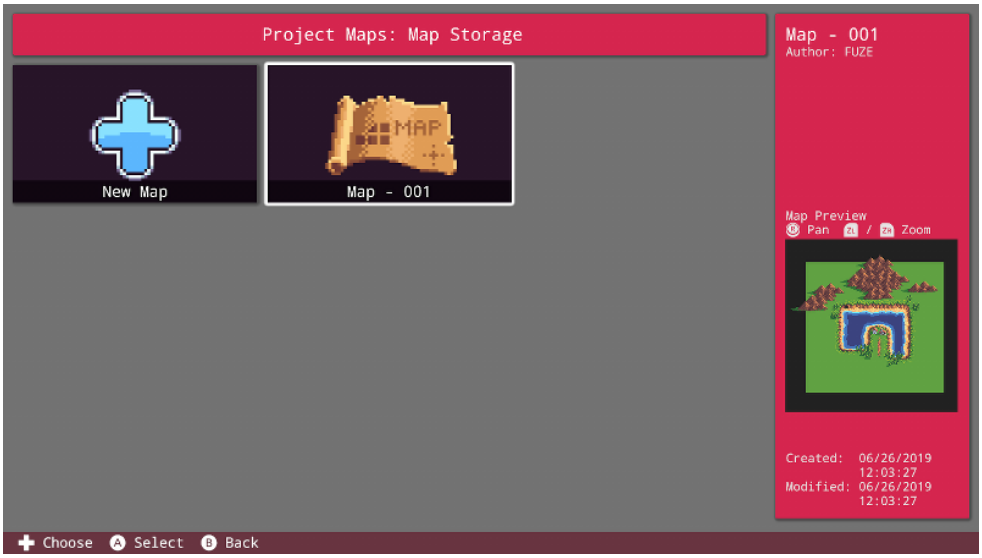
Chargeons cette carte dans un projet. Puisque nous utilisons le fichier de projet actuel comme stockage de carte, nous souhaiton copier cette carte dans un nouveau projet. Dans le menu principal, sélectionnez l'icône "Projet" et créez un nouveau projet :

-



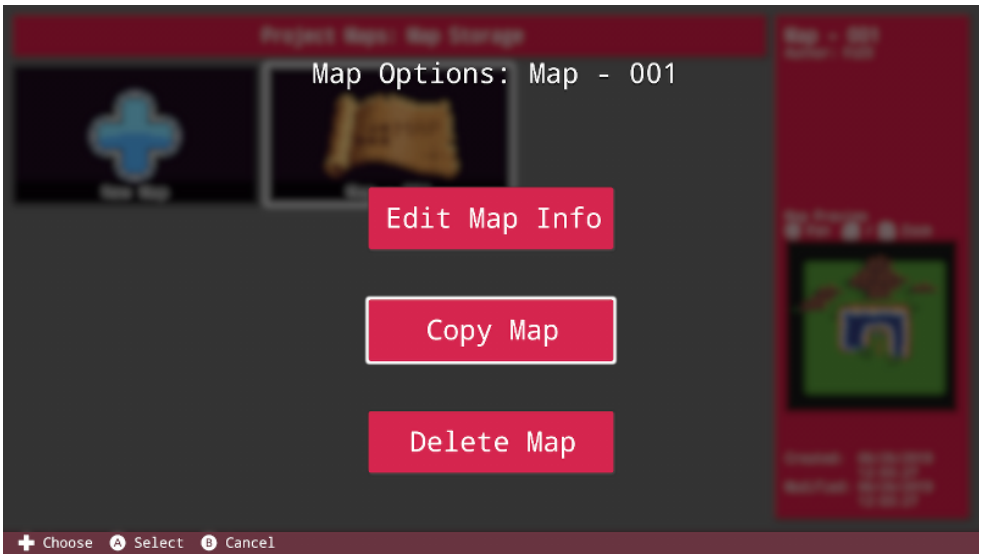
Une fois ce projet créé, retournez au menu principal et sélectionnez l'icône "Tools" (Outils) puis "Map Editor" (Editeur de carte).

Sélectionnez le projet 'Stockage de carte' pour voir notre carte :



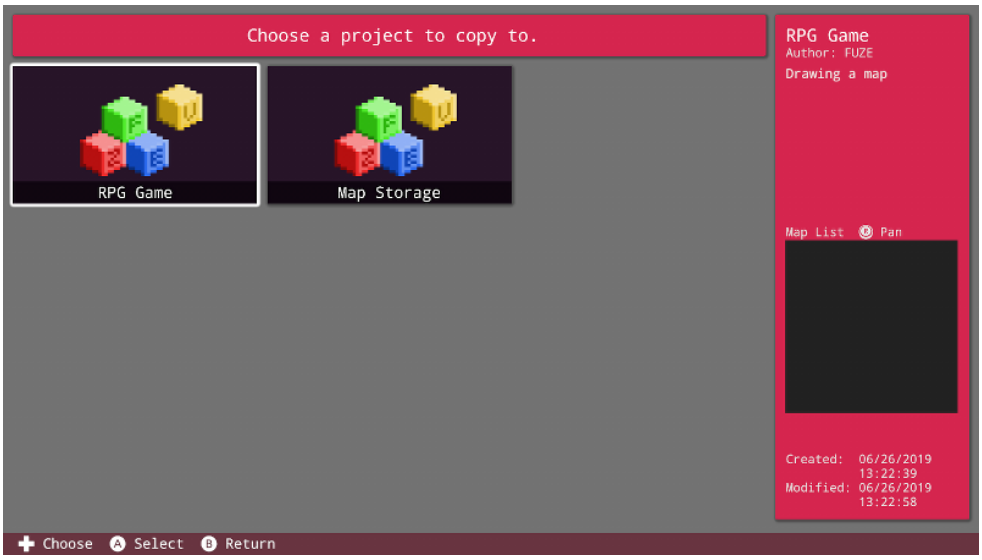
Pressez le bouton X pour afficher les actions possibles sur notre carte. Un menu avec trois options s'affiche. Sélectionnez "Copy Map" (Copier la carte) :

-



Parmi les deux options affichées, sélectionnez “Copy To Another Project” (Copier dans un autre projet).

La liste des projets vers lesquels nous pouvons copier notre carte s’affiche. Nous voulons copier la carte dans notre nouveau projet :



Sélectionnez le projet à copier, vous êtes alors invité à choisir un nouveau nom pour la carte copiée. Une fois que vous avez terminé, retournez au menu principal.

Sélectionnez l’icône “Programs” (Programmes) et ouvrez le nouveau projet (avec la copie de la carte). Cela mène à l’éditeur de code.

La première chose à faire est de charger la carte en mémoire. Pour cela, nous utilisons la **fonction** {code: loadMap ()} :

-

```
loadMap( "Map - 001" )

setSpriteCamera( gWidth() / 2, gHeight() / 2, 2 )

loop
    clear()
    centreSpriteCamera( 0, 0 )
    drawMap()
    update()
repeat
```

Ce programme ci-dessus va simplement charger notre carte et la dessiner au centre de l'écran avec un zoom de 2.

Consultez les commandes relatives aux cartes dans le Guide de Référence ; leurs liens sont listés ci-après. À partir d'ici, avec l'ensemble des commandes graphiques 2D de **FUZE**, vous pouvez à peu près tout faire avec votre carte. Consultez également les démonstrations présentant les cartes dans les projets FUZE pour plus d'inspiration et savoir comment utiliser vos données de collision !

Commandes relatives aux Cartes

[collideMap()](../Command Reference/2D Graphics/collideMap.md), [detectMapCollision()](../Command Reference/2D Graphics/detectMapCollision.md), [drawMap()](../Command Reference/2D Graphics/drawMap.md), [drawMapLayer()](../Command Reference/2D Graphics/drawMapLayer.md), [loadMap()](..Command Reference/2D Graphics/loadMap.md), [unloadMap()](..Command Reference/2D Graphics/unloadMap.md)

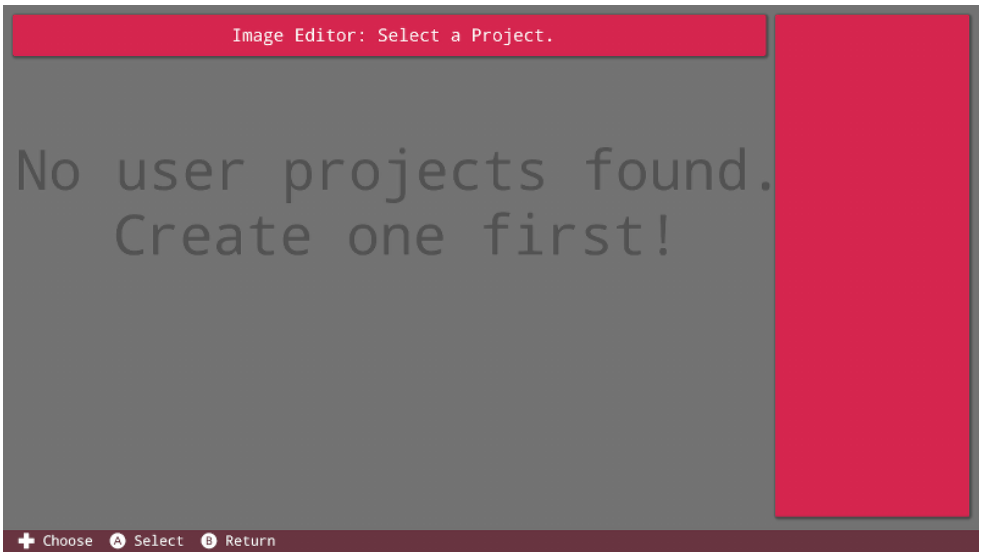
Éditeur d'Image (Image Editor)

FUZE^4 Nintendo Switch vous donne accès à une vaste bibliothèque d'assets à utiliser dans vos projets, mais il vous permet également de créer vos propres sprites et niveaux en utilisant les Éditeurs d'Image et de Carte !

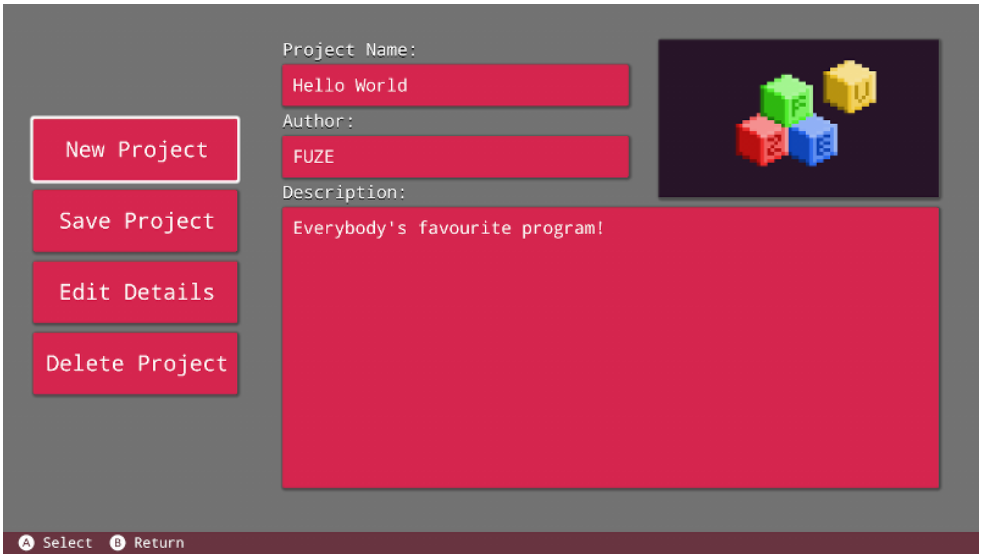
<>

-

À partir du Menu Principal, sélectionnez l'icône "Tools" (Outils) puis celle nommée "Image Editor" (Éditeur d'Image). Si c'est la première fois que vous utilisez FUZE, vous verrez apparaître l'écran suivant :



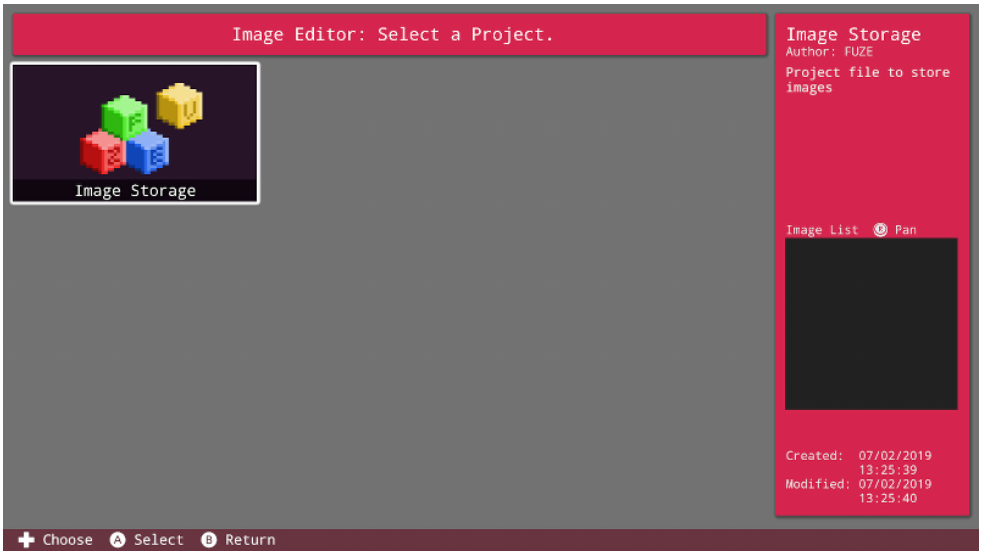
Comme vous pouvez le voir, nous ne pouvons rien faire ! Les images créées par l'utilisateur sont stockées dans des fichiers de projet individuels. Commençons par créer un projet qui peut stocker nos images. Retournez au Menu Principal et sélectionnez l'icône "Project" (Projet) :



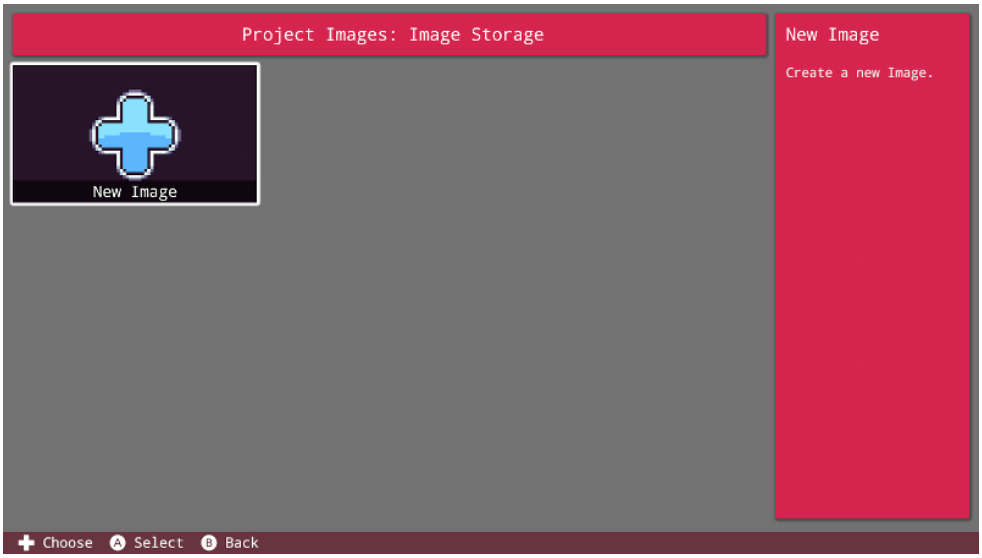
Ici, nous pouvons voir le projet chargé par défaut qui est fourni avec FUZE. Connu et adoré de tous, le fameux "Hello World" (Bonjour le Monde). Nous voulons créer un nouveau projet dédié au stockage des images. Sélectionnez le bouton "New Project" (Nouveau Projet) et entrez un nom pour ce nouveau projet. Nous allons poursuivre avec "Image Storage" dans le cadre de cet exemple.

-

Une fois le projet créé, vous accéderez automatiquement à l'Éditeur de Code. Retournez au Menu Principal en utilisant le bouton moins du contrôleur Joy-Con. À partir d'ici, cliquez sur les icônes "Tools" (Outils) et "Image Editor" (Éditeur d'Image) :



Nous voyons maintenant notre nouveau projet. Cliquez sur l'icône du projet pour accéder à la fenêtre suivante :

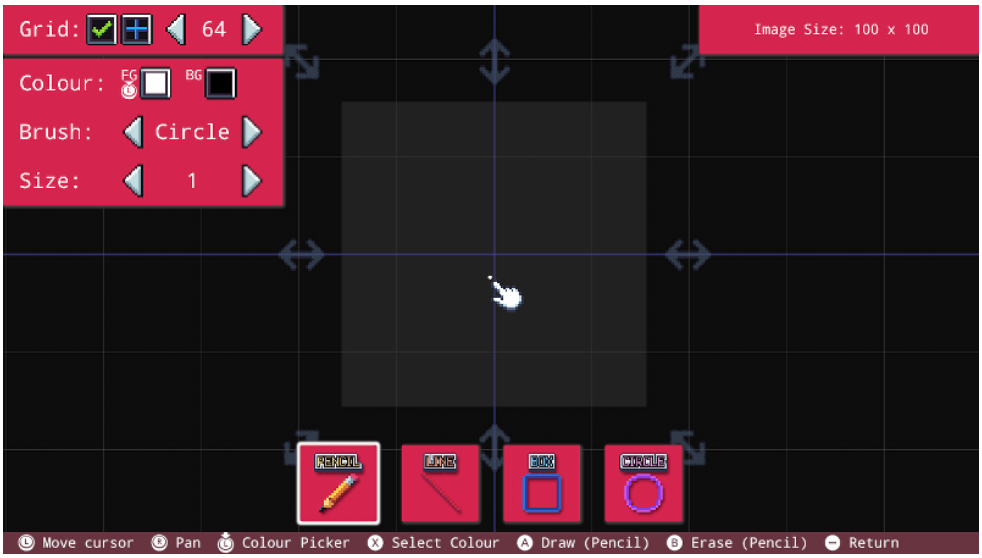


Cette fenêtre présente toutes les images stockées au sein de ce projet. Puisque nous n'en avons aucune, cliquons sur le bouton "New Image" (Nouvelle Image) pour commencer !

Vous être invité à entrer le nom de votre image, pressez le bouton plus pour valider. Une fois l'opération terminée, l'Éditeur d'Image s'affiche.

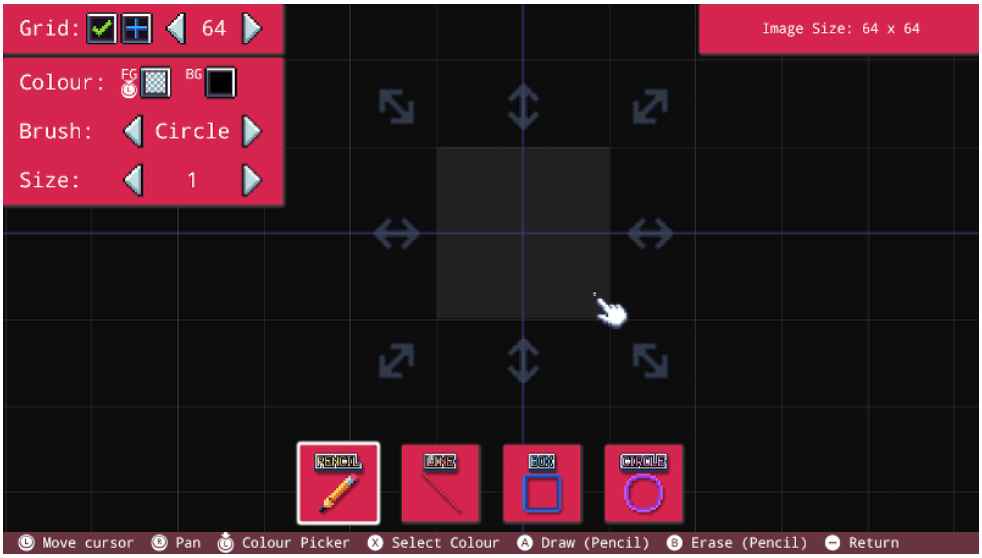
-

L’Écran de l’Éditeur d’Image



Dès à présent, quelques notions doivent être prises en compte. Regardez la boîte en haut à droite de l’écran. Elle indique la taille de votre image en pixels. Par défaut cette taille est 100 x 100 pixels.

Pour ajuster la taille de votre image, déplacez le curseur sur l’une des flèches grises aux bords de l’image. Sélectionnez-la avec une pression sur le bouton A, déplacez ensuite le stick gauche pour ajuster sa taille.



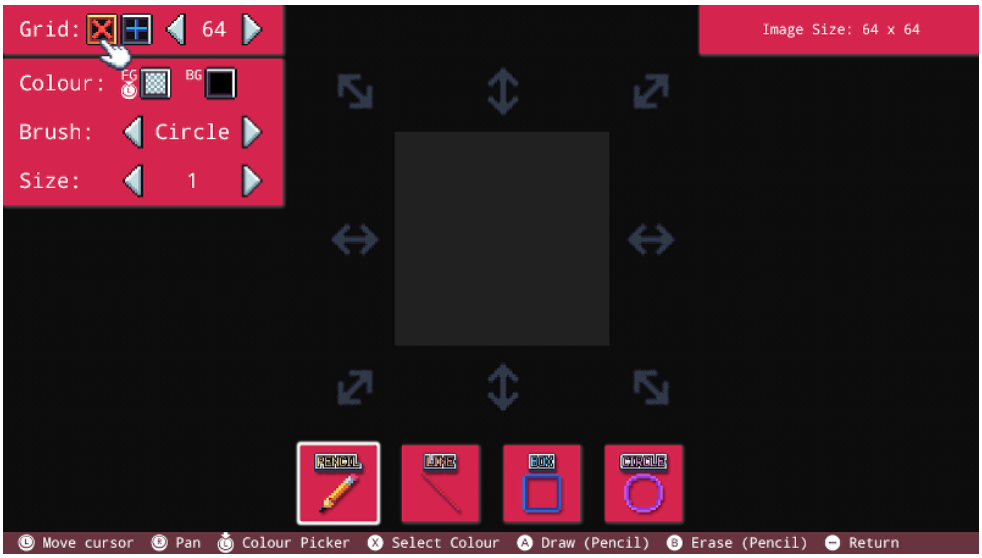
Une fois que vous avez obtenu la taille souhaitée, pressez le bouton A à nouveau pour valider. L’image va également être recentrée. Comme vous pouvez le voir, nous sommes partis sur une taille d’image de 64 x 64 pixels et sa surface est centrée à

-

l'écran.

Options de la grille

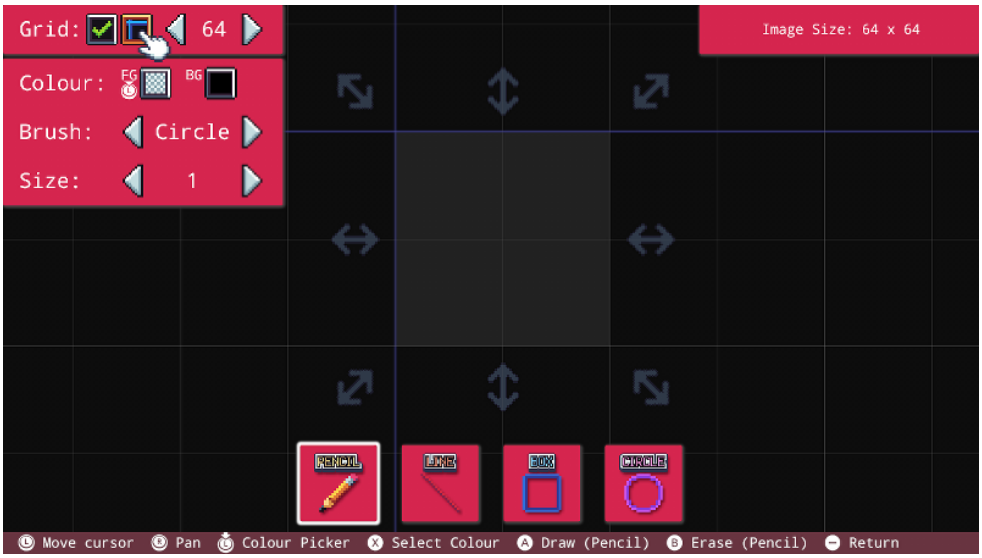
Ensuite, jetez un coup d'œil à la boîte en haut à gauche de l'écran. Nous y trouvons les options de la grille. Déplacez votre curseur sur la petite case avec une coche verte et appuyez sur le bouton A pour afficher ou masquer la grille :



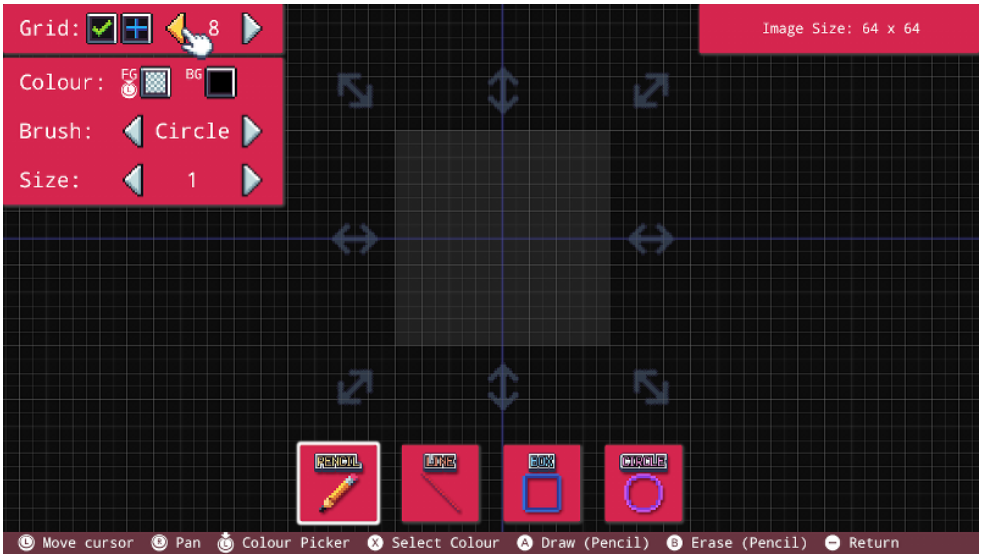
La grille a disparu et la petite boîte affiche maintenant une croix rouge à la place de la coche verte. Appuyez à nouveau sur la case pour réafficher la grille.

Vous pouvez appuyer sur la case suivante pour modifier le point d'origine de l'image.

-

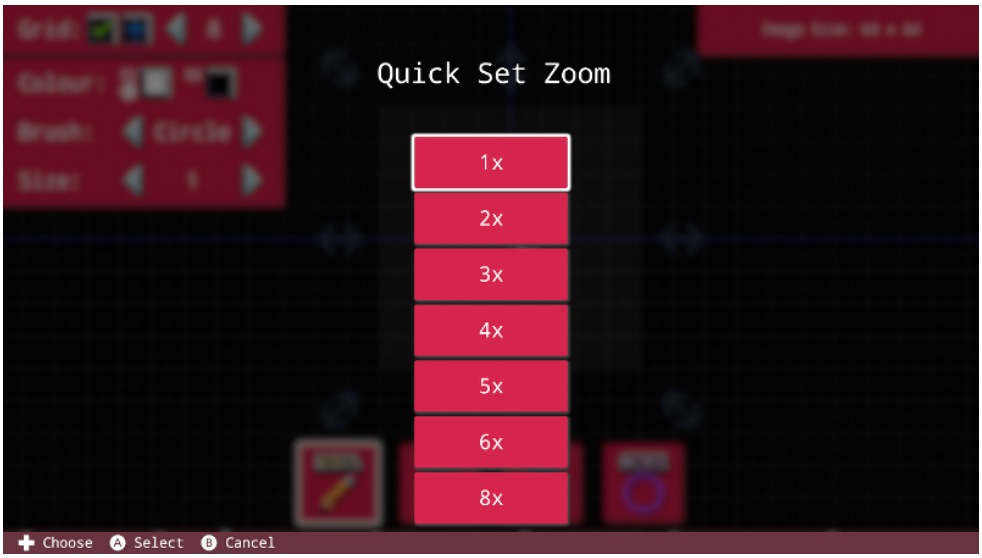


Enfin, les boutons fléchés de part et d'autre du nombre "64" du panneau d'options de la grille modifient la densité de la grille. Sélectionner un nombre inférieur augmente la quantité de carrés visibles (ce qui aide pour les détails) :



Utilisez les boutons ZR et ZL pour contrôler le zoom. ZR effectue un zoom avant sur l'image, tandis que ZL effectue un zoom arrière. Appuyez simultanément sur les boutons ZR et ZL pour afficher le menu de zoom :

-



Avec lui, vous pouvez sélectionner directement le niveau de zoom souhaité.

C’est à peu près tout pour la préparation de notre image ! Voyons ce que nous pouvons faire avec le pinceau.

Options de pinceau

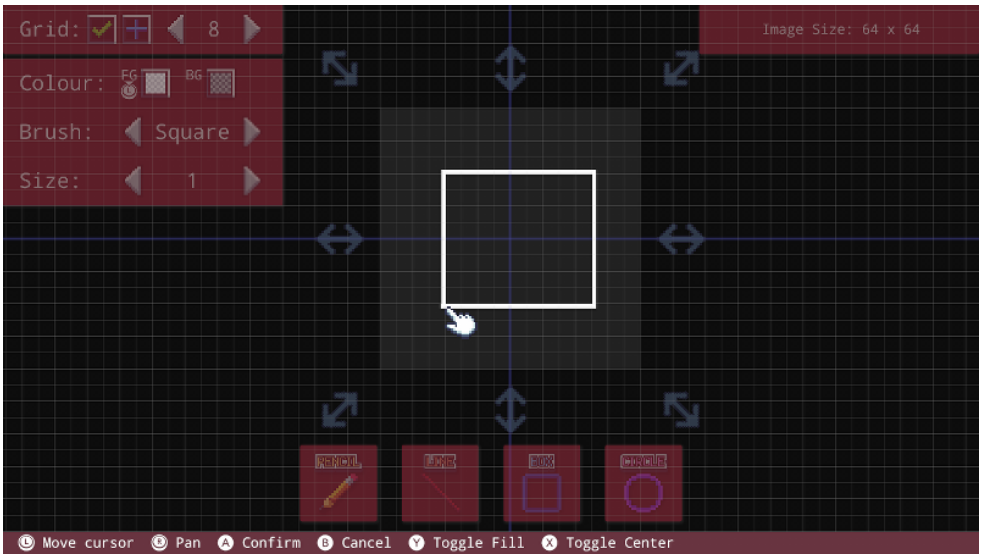
La boîte juste en dessous des options de la grille contient les options du Pinceau. Nous pouvons changer la couleur, la forme et la taille. Nous allons bientôt colorier. Pour le moment, concentrons-nous sur les options ci-dessous.

Le paramètre “Brush” (Pinceau), juste en dessous, change la forme du pinceau : “Square” (Carré) ou “Circle” (Cercle). Lorsque vous dessinez de petites images pixel par pixel, l’option Pinceau “Square” (Carré) est idéale. Tandis que pour colorer une grande zone, vous pouvez trouver le pinceau “Circle” (Cercle) plus adapté.

Enfin, le paramètre “Size” (Taille) situé en dessous modifie la taille de votre pinceau. Qui l’aurait imaginé ?!

Appuyez sur les boutons L et R pour changer la sélection en bas de l’écran entre les modes “Pencil” (Crayon), “Line” (Ligne), ‘Rectangle’ et ‘Cercle’. Cela change la façon de dessiner. La sélection de l’un de ces outils vous permet de dessiner facilement des formes simples. Lorsque vous dessinez un rectangle ou un cercle, vous verrez deux nouvelles options affichées :

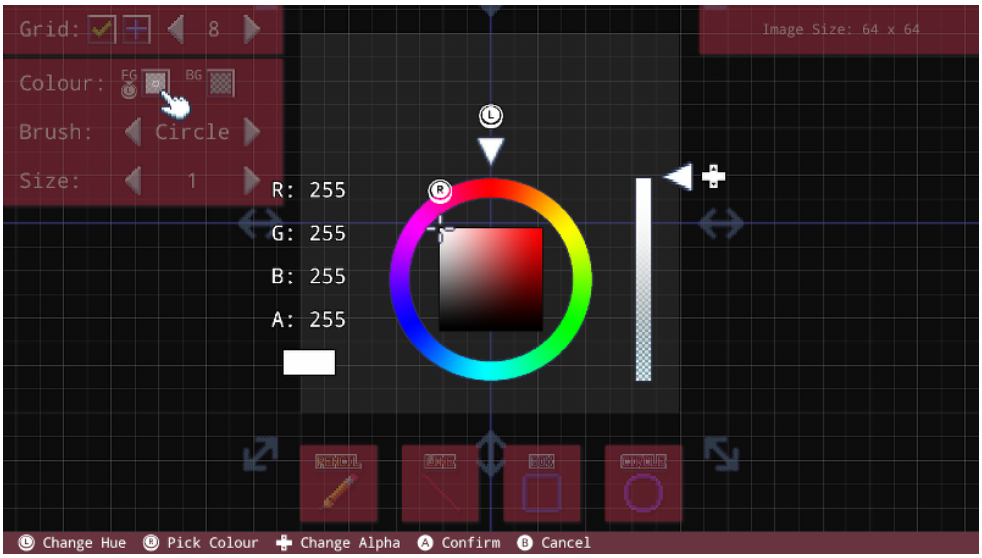
-



Lorsque vous tracez un rectangle ou un cercle, la barre de commandes est affichée en bas de l'écran. Appuyez sur le bouton Y pour remplir l'objet avec la couleur d'arrière-plan sélectionnée. Appuyez sur le bouton X pour faire basculer le centre de l'objet.

Couleurs

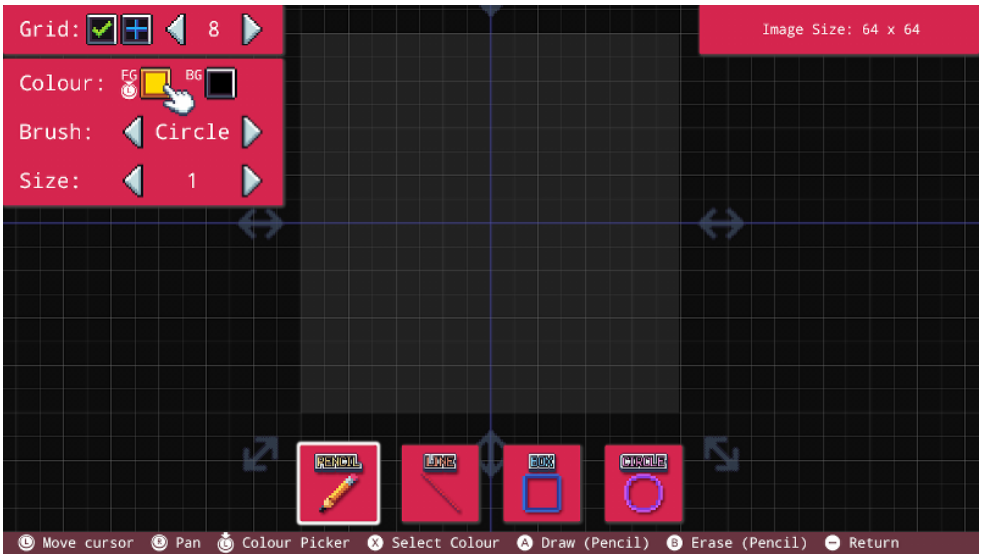
Posons notre regard sur l'option "Colour" (Couleur) en haut à gauche. Deux cases sont présentes : une pour la couleur du tracé "premier plan" (FG) et une pour la couleur de remplissage "arrière-plan" (BG). Sélectionnez la zone de premier plan et pressez A pour choisir une couleur. Le sélecteur de couleur s'affiche à l'écran :



-

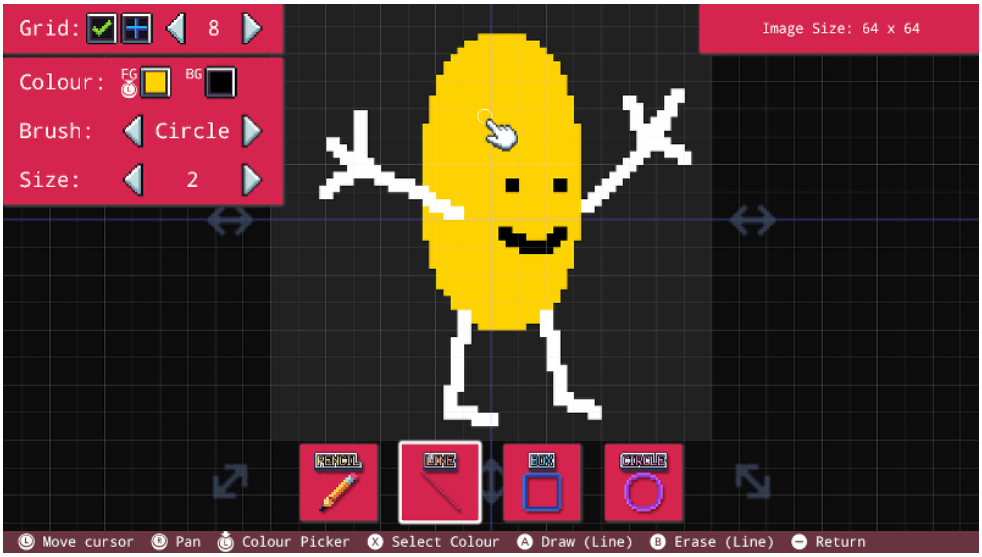
Déplacez les sticks gauche et droit pour sélectionner une couleur et une teinte. Vous pouvez également utiliser les boutons directionnels haut et bas pour régler l'alpha (transparence).

Une fois que vous avez trouvé la couleur que vous souhaitez utiliser, appuyez sur le bouton A pour valider. Cela change la couleur de la boîte associée au "premier plan" (FG) :



Lorsque nous dessinons à l'aide du bouton A, nous utilisons la couleur de premier plan (FG).

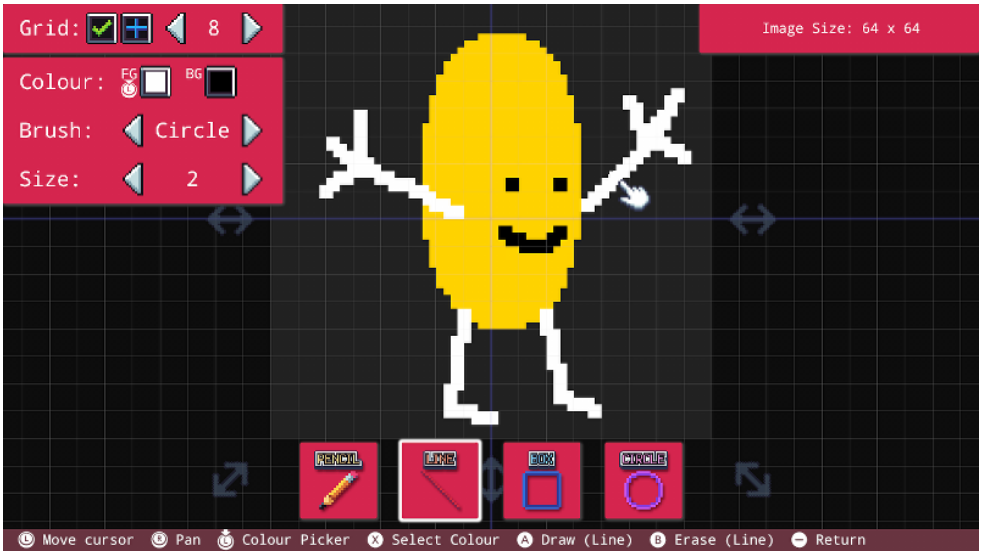
Dessinons une image simple en utilisant quelques couleurs :



Nous y voilà ! Voici mon superbe personnage. (Si, si... Il est parfait !)

-

Il existe un outil très utile dans l'éditeur d'image pour sélectionner une couleur déjà employée. Actuellement, la couleur de tracé (FG) est définie sur la couleur jaune. Si nous voulons sélectionner le même blanc que nous avons utilisé dans l'image, il suffit de placer le curseur sur un pixel de la couleur souhaitée et d'appuyer sur le bouton X :



Avez-vous remarqué la couleur blanche de la case "premier plan" ? En utilisant cet outil, nous pouvons facilement choisir n'importe quelle couleur déjà présente sur l'image.

À tout moment, vous pouvez également d'une pression sur le stick gauche appeler le sélecteur de couleur.

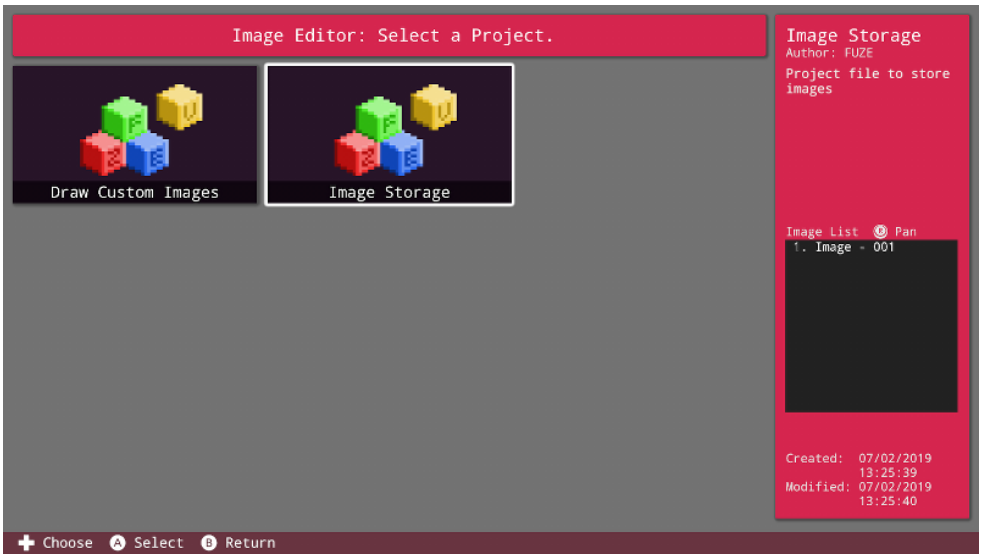
Chargement et dessin de votre image

Parfait, nous disposons maintenant d'une image. Chargeons-la dans un programme. Comme nous utilisons ce projet uniquement pour stocker les images, nous allons copier cette image dans un nouveau projet.

Commencez un nouveau projet en revenant au Menu Principal et en sélectionnant l'icône "Project" (Projet). Entrez le titre, l'auteur et la description si souhaitée. Une fois terminé, vous vous retrouvez dans l'éditeur de code. Retournez au menu principal en utilisant le bouton moins.

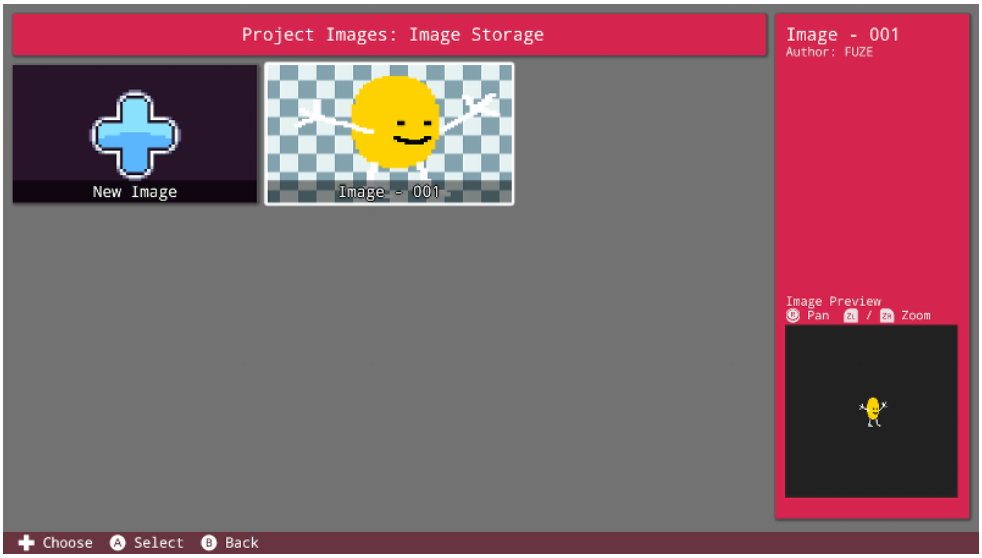
Allez dans "Tools" (Outils), puis "Image Editor" (Éditeur d'Image) comme auparavant. Vous devriez voir quelque chose comme ça :

-



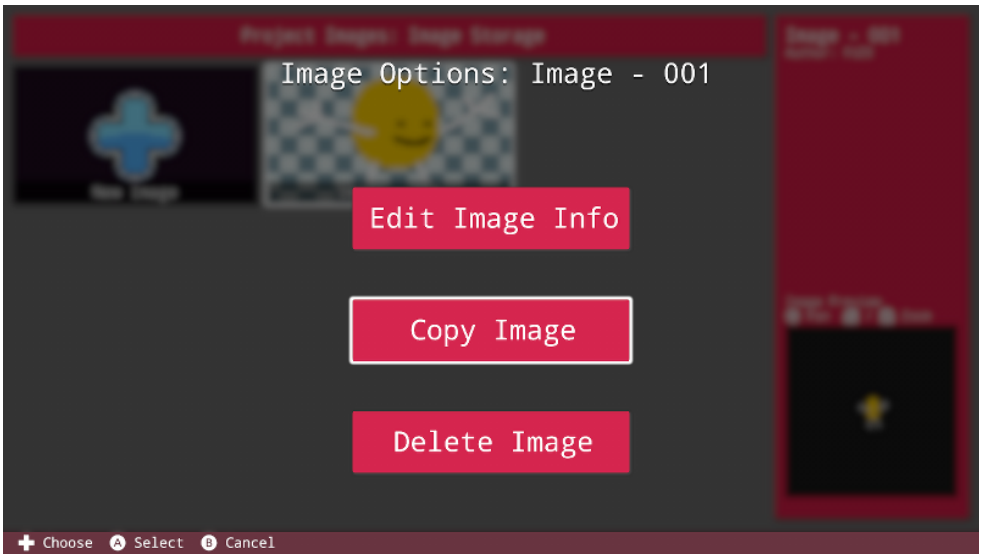
Sur la gauche, nous avons l'icône du nouveau projet dans lequel nous souhaitons utiliser l'image. Sur la droite, le curseur se trouve sur notre projet de stockage d'images (où notre image est actuellement enregistrée). À droite de l'écran, dans le panneau d'informations sur le projet, vous pouvez voir la liste des images stockées dans le projet.

La sélection de l'icône de projet "Image Storage" (Stockage d'Images) ouvre le dossier des images stockées dans ce projet.



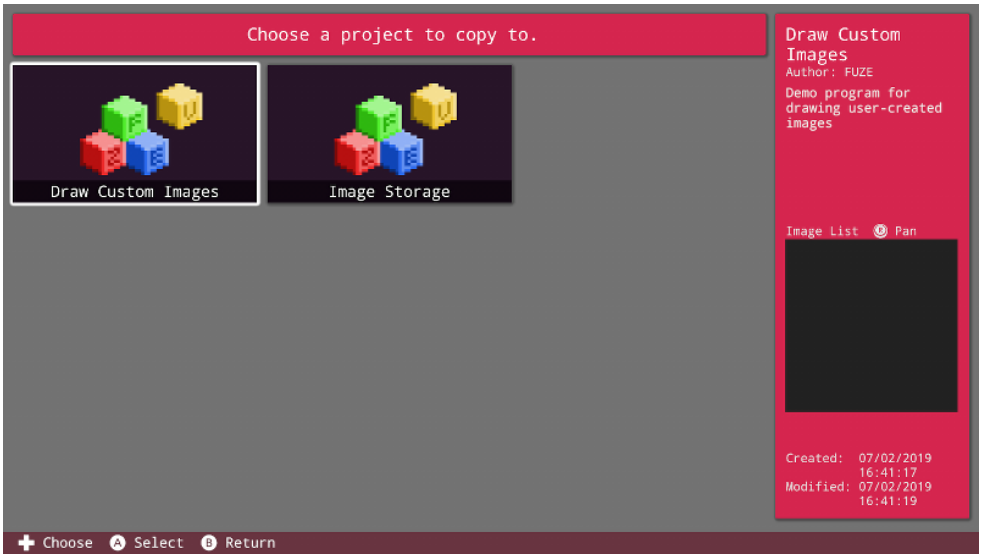
Comme vous pouvez le constater, notre image est maintenant affichée. Déplacez le curseur sur l'image et appuyez sur le bouton X pour afficher les options disponibles :

-



Sélectionnez l'option "Copy Image" (Copier l'Image). Elle propose deux autres choix. Vous pouvez soit copier l'image dans le même projet (la dupliquer), soit copier l'image dans un autre projet (où vous souhaitez l'utiliser).

Sélectionnez "Copy To Another Project" (Copier dans un autre projet), puis sélectionnez le projet nouvellement créé dans lequel nous voulons copier l'image.



Une fois le projet sélectionné, vous êtes invité à saisir un nouveau nom pour l'image (si vous le souhaitez).

Le Code

-

Lorsque votre fichier de projet contient l'image à utiliser, vous pouvez la charger et l'afficher à l'aide d'un peu de code !

Entrez ce qui suit dans l'éditeur de code :

```
1. img = loadImage( "Image - 001", false )
2.
3. loop
4.     clear()
5.     drawImage( img, gWidth() / 2, gHeight() / 2, 4 )
6.     update()
7. repeat
```

Sur la ligne 1, nous stockons notre image dans une **variable** à l'aide de la **fonction** {code:loadImage()}. Biensûr, le nom du fichier sur la ligne 1 {code:"Image - 001"} doit être remplacé par le nom de votre image. Le paramètre {code:false} entre les parenthèses de {code:LoadImage()} inhibe l'application du filtre d'anticrénelage à l'image. Cette option permet de garder des bords de pixels bien nets. Si vous souhaitez adoucir les bords, substituez {code: true} à {code:false}.

Ensuite, une boucle ({code: loop}) très simple est utilisée. Elle expoite la **fonction** {code: drawImage ()} pour dessiner notre image au milieu de l'écran, avec un multiplicateur d'échelle de 4.

Ça y est, nous l'avons ! Nous avons créé notre propre image et l'avons dessinée à l'écran. Que pourriez-vous faire de plus avec ? La limite est votre seule imagination ! Consultez les commandes de manipulation d'image référencées ci-dessous pour découvrir plus de fonctions graphiques 2D.

Vous pouvez très facilement dessiner successivement plusieurs images, stockées dans un grand tableau pour créer votre propre sprite animé !

Commandes relatives aux Images

drawImage()(../Command Reference/2D Graphics/drawImage.md), [drawImageEx()](../Command Reference/2D Graphics/drawImageEx.md), [freeImage()](../Command Reference/2D Graphics/freeImage.md), [imageSize()](../Command Reference/2D Graphics/imageSize.md), [loadImage()](../Command Reference/2D Graphics/loadImage.md)

-

Tutoriel 1 : les Boucles

Chaque programmeur doit commencer quelque part. Dans ce tutoriel, nous allons écrire une version du programme le plus célèbre qui soit : Hello World (Bonjour le Monde).

Avant de nous lancer, nous avons besoin de quelques **fonctions**. Nous avons besoin de la **fonction** {code:print()} pour indiquer à **FUZE^ 4 Nintendo Switch** le texte à afficher. Nous avons également besoin de la **fonction** {code:update()} pour actualiser l'écran avec ce texte et, enfin, nous avons besoin de la **fonction** {code:sleep()} pour garder notre programme à l'écran !

Tapez (ou copiez/collez) le code ci-dessous dans l'éditeur de code de **FUZE**. Exécutez ce programme avec **+** ou la touche **F5** si vous utilisez un clavier USB.

```
print( "Bonjour Nintendo" )
update()
sleep( 2 )
```

Vous devez voir le texte choisi apparaître dans toute sa beauté à l'écran pendant 2 secondes. Maintenant, faisons-le maintenant encore et encore, en utilisant l'un des concepts les plus importants de la programmation : une **boucle** ({code:loop}). Pour cela, nous pouvons supprimer la fonction {code:sleep()} car notre programme s'exécutera en continu. Nous aurons quand même besoin du {code:update()} pour faire apparaître le texte à l'écran !

```
loop
    print( "Bonjour Nintendo" )
    update()
repeat
```

Imaginez une **boucle** comme un sandwich. Nous avons un morceau de pain ({code:loop}), des morceaux au milieu et à nouveau un morceau de pain ({code:repeat}) ! Sans l'un des morceaux de pain, notre sandwich n'est plus un sandwich !

-

Ce que je dis c'est que...

Sans {code:loop}, {code:repeat} ne fera rien et vice-versa !

Il y a un point important en ce qui concerne les **boucles**. Regardez ci-dessous:

```
1. loop
2.     print( "Bonjour Nintendo" )
3.     update()
4. repeat
5. print( "Et moi ?" )
```

Vous remarquez la nouvelle **fonction** {code:print()} à la ligne 5 ? Son texte n'apparaîtra jamais à l'écran. Savez-vous pourquoi ?

Lorsque **FUZE** arrive au **mot-clé** {code:repeat} de la ligne 4, il retourne au dernier **mot-clé** {code:loop} rencontré. Notre programme s'exécute donc comme suit :

Ligne 1, ligne 2, ligne 3, Ligne 4, Ligne 1, ligne 2, ligne 3, Ligne 4, Ligne 1, ligne 2, ligne 3, Ligne 4...

ETERNELLEMENT! Eh bien ... pas tout à fait. Il fonctionnera ainsi jusqu'à ce que nous lui disions d'arrêter. Appuyez sur **+** pour arrêter le programme et revenir à l'éditeur (ou la touche **F5** si vous utilisez un clavier USB).

La ligne 5 n'est pas **incluse** dans la **boucle** ({code:loop} .. {code:repeat}). Pour que cette ligne soit exécutée, notre programme doit ressembler à ceci :

```
1. loop
2.     print( "Bonjour Nintendo" )
3.     print( "Et moi ?" )
4.     update()
5. repeat
```

Maintenant, la nouvelle ligne {code:print()} est incluse **à l'intérieur** de la **boucle**. Elle en est même très contente.

Bien. Ajoutons de la couleur à notre programme. Pour cela, nous utilisons la **fonction** {code:ink()}.

-

```
1. ink( white )
2. loop
3.     print( "Bonjour Nintendo" )
4.     print( "Et moi ?" )
5.     update()
6. repeat
```

Nous avons utilisé le *nom* pour désigner la couleur que nous voulons, mais nous pouvons utiliser des nombres à la place. Chaque couleur est stockée dans une grande base de données, chacune avec son numéro. Notre choix s'étend sur une palette de 64 couleurs.

Cela implique que nous pouvons choisir notre couleur de façon aléatoire. Nous aurons besoin de la **fonction** {code:random()} à cet effet !

```
1. ink( random( 64 ) )
2. loop
3.     print( "Bonjour Nintendo" )
4.     print( "Et moi ?" )
5.     update()
6. repeat
```

L'emploi des parenthèses peut sembler un peu délicat. N'oubliez pas que l'intégralité de {code:random(64)} doit figurer entre parenthèses pour la ligne {code:ink()}. Cela peut paraître bizarre au début, mais nous devons avoir le même nombre de parenthèses **ouvrantes** et **fermantes**.

Pour changer la couleur, vous devez arrêter et redémarrer le programme. C'est fastidieux... Améliorons ça !

```
1. loop
2.     ink( random( 64 ) )
3.     print( "Bonjour Nintendo" )
4.     print( "Et moi ?" )
5.     update()
6. repeat
```

Nous avons maintenant introduit la ligne effectuant le changement de couleur **dans** la **boucle** {code:loop} ! Pouvez-vous prédire ce qui va arriver?

-

De cette façon, **FUZE** définit aléatoirement la couleur à chaque répétition de la **boucle** {code:loop} !

Imaginons que nous souhaitons une couleur pour “Bonjour Nintendo” et une autre couleur pour "Et moi ?".

```
1. loop
2.     ink( fuzepink )
3.     print( "Bonjour Nintendo" )
4.     ink( fuzeblue )
5.     print( "Et moi ?" )
6.     update()
7. repeat
```

Dans l’exemple ci-dessus, la ligne 2 définit une couleur pour la **fonction** {code:print()} de la ligne 3. Ensuite, la ligne 4 définit une couleur différente pour la **fonction** {code:print()} de la ligne 5. Pour qu’une action affecte une ligne spécifique, nous devons mettre les instructions correspondantes *avant* cette ligne.

Pour terminer, changeons la taille de notre texte avec la **fonction** {code:textSize()} :

```
1. textSize( 100 )
2. loop
3.     ink( fuzepink )
4.     print( "Bonjour Nintendo" )
5.     ink( fuzeblue )
6.     print( "Et moi ?" )
7.     update()
8. repeat
```

{code:textSize()} nous permet de définir la taille de notre texte en pixels. Avec {code: textSize(100)}, la hauteur maximale de nos lettres est de 100 pixels.

Félicitations ! Vous venez d’écriture le meilleur programme de tous les temps. Vous êtes sur la bonne voie !

Fonctions et mots-clés utilisés dans ce tutoriel

[ink()](../Command Reference/Text Handling/ink.md), loop, [print()](../Command Reference/Text Handling/print.md), repeat, [sleep()](../Command Reference/Time and Date/sleep.md), [textSize()](../Command Reference/Text

-

Handling/textSize.md), [update()](../Command Reference/Screen Display/update.md)

Tutoriel 2 : les Variables

Les **variables** sont une autre des composantes les plus importantes de la programmation.

Que sont-elles ? Eh bien, très simplement, une **variable** est une étiquette. Une étiquette que *vous* créez pour stocker une information.

Pourquoi avons-nous besoin d'étiquettes ? Eh bien, imaginez que vous cherchiez une aiguille dans une botte de foin. C'est un sacré défi, non ? En fait, pas vraiment si nous avons un énorme panneau marquant exactement son emplacement !

La principale raison pour laquelle nous utilisons des **variables** est qu'elles nous permettent de modifier et de manipuler les éléments d'un programme.

Quasiment tous les jeux utilisent des **variables** tout le temps, pour mémoriser tout et n'importe quoi : *score*, *santé*, *statistiques*, *position d'écran*, *état des boutons des contrôleurs*, etc.

Commençons. J'espère que vous aimez les bonbons !

```
1. bonbons = 3
```

Tapez le code ci-dessus dans l'éditeur de code de **FUZE^4 Nintendo Switch** et appuyez sur **+** pour exécuter le programme.

Rien ne va se passer. En fait, vous serez immédiatement redirigé vers l'éditeur !

Cependant, dans le cerveau de l'ordinateur, nous avons pris le nombre **3** et l'avons stocké dans un emplacement appelé {code:bonbons}.

-

Maintenant, modifiez votre programme pour qu'il ressemble à celui ci-dessous et lancez-le.

```
1. bonbons = 3
2. print( bonbons )
3. update()
4. sleep( 2 )
```

Tout ce que nous avons fait est de demander à **FUZE** d'imprimer le contenu de la **variable** {code:bonbons} à l'écran. Vous devriez voir le nombre **3** apparaître ! **FUZE** retrouve l'information étiquetée {code:bonbons} et l'affiche.

Nous pouvons tout stocker dans une **variable**. Le code ci-dessous fonctionne également :

```
1. bonbons = "Délicieux"
2. print( bonbons )
3. update()
4. sleep( 2 )
```

Cette fois cependant, nous aurons le mot "Délicieux" au lieu du nombre **3**. Vous avez compris ? Génial !

Compliquons un peu les choses... Nous allons utiliser quelques nouvelles notions très importantes.

Modifiez votre code dans l'éditeur **FUZE^4 Nintendo Switch** afin qu'il ressemble au programme ci-dessous et exécutez-le. Il devrait décompter nos bonbons jusqu'à ce qu'il n'en reste plus.

```
 1. bonbons = 3
 2. while bonbons > 0 loop
 3.     clear( black )
 4.     print( "J'ai ",bonbons," bonbons dans mon sac." )
 5.     print( "Si j'en mange un... alors..." )
 6.     bonbons -= 1
 7.     update()
 8.     sleep( 1 )
 9. repeat
10. print( "Il ne me reste plus de bonbon... Oh, non." )
11. update()
12. sleep( 2 )
```

-

Tout d'abord, nous avons une nouvelle **fonction** : {code:clear()}. Elle est utilisée pour effacer l'écran avec une couleur. Essayez de mettre une couleur différente entre parenthèses.

Sur la ligne 4, nous avons la **variable** nommée {code:bonbons} entre des virgules et non dans le texte. Si nous avions écrit {code:print("bonbons")} nous aurions affiché le mot "bonbons". Avez-vous remarqué la présence des *guillemets* entre parenthèses ?

Rappelez-vous que lorsque nous écrivons {code:print(bonbons)}, sans *guillemet*, nous affichons le contenu de la **variable**.

Abordons les nouveaux éléments présentés ici. Le premier est un nouveau type de **boucle** appelé boucle **while** (tant que).

While (Tant que)
Une boucle **while** est une boucle condionnelle, qui se répète jusqu'à ce qu'une certaine condition soit vérifiée.

Cette boucle **while** se répète aussi longtemp que la **variable** {code:bonbons} a une valeur *supérieure à zéro* ({code:bonbons > 0}).

Pour que notre boucle **while** se termine et que le programme se poursuive, nous devons réduire la valeur de la **variable** {code:bonbons} jusqu'à 0.

La prochaine partie délicate se situe à la ligne 6.

Moins égal (-=) et Plus égal (+=)
La ligne de code qui réduit la valeur de la **variable** {code:bonbons} est la ligne 6.

```
6. bonbons -= 1
```

Le signe après "bonbons" est appelé **moins égal**. Nous allons souvent rencontrer ces notations au fur et à mesure que nous progressons, alors faisons de notre mieux pour les comprendre. {code:-=} signifie que nous soustrayons à la valeur stockée dans notre **variable**. {code:+=} ou **plus égal** ajoute à la valeur.

En réalité, la ligne 6 se lit comme suit :

-

```
6. bonbons = bonbons - 1
```

Ce qui signifie : Redéfinit la **variable** {code:bonbons} de sorte à ce qu'elle soit égale au contenu de la **variable** {code:bonbons} *moins* 1.

L'emploi de {code:-=} et {code:+=} nous aide à gagner du temps parce qu'ainsi nous n'avons pas à écrire chaque nom de **variable** deux fois !

Toujours avec nous ?

Impressionnant ! L'emploi des **variables** devient rapidement une seconde nature, mais il faut prendre le temps de s'y habituer.

Essayez de réécrire le programme de manière à ce que, au lieu de prendre des bonbons, vous commencez par 0 bonbon et en gagnez au fur et à mesure de l'avancement de la boucle. Pensez également à changer la condition de la boucle **while** afin qu'elle se termine lorsque vous atteignez un certain nombre de bonbons.

Bien joué! Vous êtes avez arrivé à la fin. Rendez-vous dans le prochain tutoriel où nous parlerons d'une autre technique essentielle en programmation : la structure **If Then** (Si ... Alors...).

Fonctions et mots-clés utilisés dans ce tutoriel

[clear()](../Command Reference/Screen Display/clear.md), loop, [print()](../Command Reference/Text Handling/print.md), repeat, [sleep()](../Command Reference/Time and Date/sleep.md), [update()](../Command Reference/Screen Display/update.md), while

Tutoriel 3 : l'alternative "Si..."

En avant ! Ce tutoriel couvre un autre des concepts les plus importants de la programmation : l'alternative **"Si..."**.

-

Elle a l'avantage de ne pas nécessiter de longues explications... Tout le temps, dans la vie normale, nous utilisons l'alternative : "si... alors...".

Cela vous semble peut-être familier... ?

"**Si** tu fais tous tes devoirs, **alors** tu pourras avoir plus de temps pour jouer sur ta Nintendo Switch !"

Ou peut-être...

"**Si** tu manges tous tes légumes, **alors** tu pourras avoir un peu de gâteau au chocolat en plus, **sinon** tu iras au lit sans dîner !"

Espérons que ces phrases apportent un peu de sens à tout cela ! Une alternative teste **si** une condition est vérifiée et **alors** fait quelque chose. **Sinon**, nous faisons autre chose !

D'accord, assez de détours ! Le lexique de **FUZE^4 Nintendo Switch** étant issu l'anglais, les mots-clés "si"... "alors"... "sinon"... "finsi", deviennent respectivement {code:if}... {code:then}... {code:else}... {code:endif}. Écrivons un petit programme de quiz pour les mettre en oeuvre.

```
  1. print( "Bienvenue à l'ultime quiz FUZE, réputé pour son
extrêmement difficiculté. \n" )
  2. update()
  3. sleep( 2 )
  4. print( "Question 1 : Quel est la capital du Japan ? \n" )
  5. update()
  6. sleep( 2 )
  7. proposition = input( "Quel est la capital du Japan ?" )
  8. if proposition == "Tokyo" then
  9.     print( "CORRECT ! Bien joué ! \n" )
 10. else
 11.     print( "INCORRECT ! Je ne peux pas croire que tu
l'ignorais... \n" )
 12. endif
 13. update()
 14. sleep( 3 )
```

Nous avons une nouvelle **fonction**, regardez à la ligne 7.

input()

-

Sur la ligne 7, nous utilisons la **fonction** {code:input()} pour permettre au joueur d'entrer une réponse à notre question. La **fonction** {code:input()} dévoile le clavier de **FUZE^4 Nintendo Switch** à l'écran afin de l'autoriser à saisir ce qu'il veut. Quelles que soient les indications saisies, elles sont stockées dans une **variable**. Nous avons appelé notre **variable** {code: proposition}.

Lorsque nous utilisons la **fonction** {code:input()} nous devons indiquer un message à afficher. Nous avons rappelé la question {code:"Quelle est la capitale du Japon ?"} au joueur mais vous pouvez indiquer ce que vous voulez ! Si vous ne souhaitez pas afficher de message, mettez simplement une paire de guillemets entre les parenthèses.

L'alternative "Si..."

Les lignes 8 à 12 constituent notre alternative. Elle commence avec {code:if} pour introduire la condition. Dans l'exemple, nous vérifions **si** (if) le contenu de la **variable** nommée {code:proposition} est *exactement égale au* texte de la chaîne de texte "Tokyo". **Si** (if) ça l'est, **alors** (then) nous affichons "CORRECT ! Bien joué !"

Nous utilisons le **mot-clé** {code:else} au sein de l'alternative **"Si..."** pour agir différemment si la condition **n'** est **pas** remplie. Dans ce cas, nous savons que notre condition n'est pas remplie, la solution proposée est incorrecte et nous en informons le joueur.

Finalement, nous utilisons {code:endif} pour terminer l'alternative **"Si..."**, en son absence nous pouvons avoir de gros problèmes. En effet, sans {code:endif}, l'ordinateur considère toutes les lignes qui suivent comme faisant partie de l'alternative **"Si..."** (puisqu'elle n'a pas de fin).

Éqal (=) et Double Éqal (==)

Avez-vous noté que le signe égal est doublé sur la ligne 8 ? Lorsque nous **comparons** deux choses, nous devons employer un **double égal** ({code:==}). Lorsque nous **assignons** une valeur à une variable, nous utilisons un **simple égal** ({code:=}).

Mémoriser le score

-

D’accord… Améliorons ça. Nous pouvons utiliser une **variable** pour garder une trace du **score** du joueur et rendre le quiz vraiment amusant.

```
  1. score = 0
  2. print( "Bienvenue à l'ultime quiz FUZE, réputé pour son
extrêmement difficiculté. \n" )
  3. update()
  4. sleep( 1 )
  5. print( "Question 1 : Quel est la capital du Japan ? \n" )
  6. update()
  7. sleep( 2 )
  8. proposition = input( "Quel est la capital du Japan ?" )
  9. if proposition == "Tokyo" then
 10.     print( "CORRECT ! Bien joué ! \n" )
 11.     score += 1
 12. else
 13.     print( "INCORRECT ! Je ne peux pas croire que tu
l'ignorais... \n" )
 14. endif
 15. update()
 16. sleep( 2 )
 17. print( "Vous avez remporté... ",score, " point(s) \n" )
 18. update()
 19. sleep( 1 )
 20. if score == 1 then
 21.     print( "Félicitations ! Vous avez la totalité des points
!" )
 22. else
 23.     print( "Vous aurez plus de chance la prochaine fois." )
 24. endif
 25. update()
 26. sleep( 3 )
```

Dans l’exemple ci-dessus, à la ligne 1, nous avons ajouté la **variable** score. Elle prend **0** pour valeur car notre joueur n’a encore répondu à aucune question !

La ligne 11 est une nouvelle ligne qui augmente le contenu de la variable de score de 1 **si** le joueur répond correctement. Rappelez-vous, nous utilisons {code:+=} pour le faire.

Ensuite, les lignes 20 à 24 constituent une nouvelle alternative **Si…** pour afficher un message au joueur en fonction de son score.

-

En utilisant ce que vous avez appris, pouvez-vous écrire 2 autres questions pour le quiz ? Vous devez placer vos nouvelles questions entre les lignes 16 et 17, car les lignes 17 à 26 terminent le quiz et indiquent le score du joueur.

Vous devrez changer la ligne 20 {code:if score == 1 then} pour remplacer la valeur 1 par le nouveau score maximal. Par exemple, si votre questionnaire contient 3 questions et que vous marquez 1 point pour chaque réponse correcte, la ligne doit indiquer {code:if score == 3 then}.

N'hésitez pas à utiliser la **fonction** {code:ink()} pour donner un peu de couleur à votre quiz !

Fonctions et mots-clés utilisés dans ce tutoriel

else, endIf, if, input()(../Command Reference/Input/input.md), [print()](../Command Reference/Text Handling/print.md), [sleep()](../Command Reference/Time and Date/sleep.md), then, [update()](../Command Reference/Screen Display/update.md)

Tutoriel 4 : l'Écran

Rebonjour ! Dans ce didacticiel, nous verrons ce qu'est l'écran, comment il fonctionne et comment nous pouvons l'utiliser dans nos programmes.

L'écran est une énorme collection de minuscules petites lumières appelées **pixels**. Il y en a beaucoup, vraiment beaucoup. L'écran de votre **Nintendo Switch** en comporte **921600**. Presque un million !

Chaque fois que votre écran est allumé, que vous jouez à un jeu, parcourez l'eShop ou simplement le menu HOME, votre **Nintendo Switch** change ces 921600 petites lumières à une vitesse incroyable. Environ 60 fois par seconde, en fait.

Pensez-y juste un instant, c'est stupéfiant !

X et Y

-

Vous êtes probablement déjà familier avec les axes **x** et **y**. Mais, juste au cas où vous n'en auriez jamais entendu parler auparavant, regardons cela rapidement.

L'énorme collection de pixels que nous avons mentionnée est organisée avec un **axe x** et un **axe y**. Si quelque chose se déplace le long de l'**axe x**, il se déplace de vers la gauche ou la droite. Si quelque chose se déplace le long de l'**axe y**, il se déplace vers le haut ou le bas !

Jetez un coup d'œil à cette image pour vous faire une idée :

Avez-vous remarqué les flèches ? Elles indiquent le sens croissant (des valeurs) sur les axes.

Les deux axes commencent à 0 et augmentent d'un pixel à la fois, vers la droite et le bas de l'écran.

L'écran de votre **Nintendo Switch** a 1280 pixels sur l'**axe x** et 720 pixels sur l'**axe y**. Multipliez ces valeurs ensemble et vous obtiendrez le nombre 921600 mentionné précédemment !

Voyez ci-après la façon dont les nombres croissent le long des axes :

-

Le zéro dans le coin en haut à gauche indique que les axes **x** et **y** prennent leur origine (0) dans le coin supérieur gauche de l'écran.

La valeur maximale pour l'**axe x** est celle la plus éloignée à droite de l'écran, et la valeur maximale pour l'**axe y** est en bas de l'écran.

Mode TV et Mode Portable

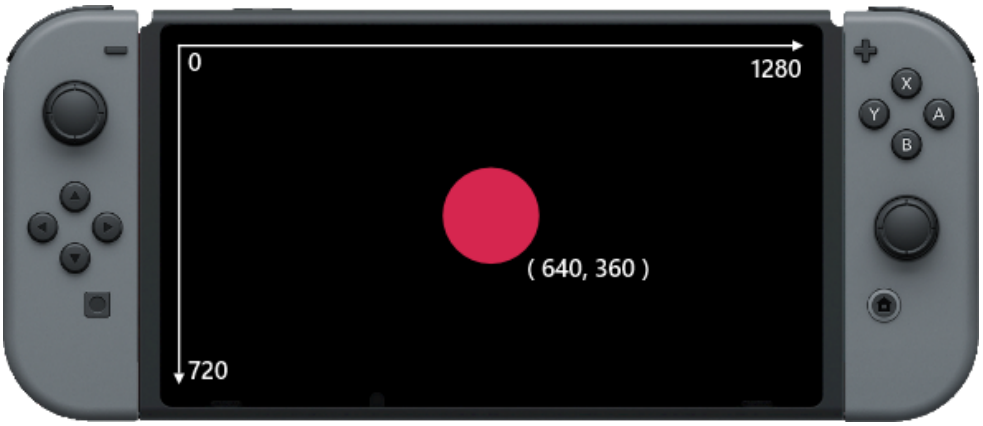
Une chose importante à garder en mémoire est que la résolution de l'écran (le nombre de pixels à l'écran) change entre le mode Portable et mode TV. Lorsque votre **console Nintendo Switch** est sur sa station d'accueil, la résolution est de **1920** par **1080**, au lieu de **1280** par **720**.

Utiliser les Coordonnées de l'écran dans un Programme

Disons que nous souhaitons placer un cercle exactement au milieu de l'écran. Nous avons besoin de connaître le point central de chaque axe. La moitié de 1280 est 640, et la moitié de 720 et 360.

Si nous utilisons 640 et 360 comme coordonnées pour la position **x** et **y** à l'écran, nous obtenons quelque chose comme cela :

-



Voyons à quoi ressemble le code.

```
1. loop
2.     clear()
3.     circle( 640, 360, 100, 32, fuzepink, false )
4.     update()
5. repeat
```

Simple et concis. Dans la **boucle** ({code:loop), nous utilisons les **fonctions** {code:clear()} pour effacer l'écran et {code:update()} pour envoyer les informations à l'écran.

Regardez la ligne 3. Cette ligne dessine le cercle coloré.

La **fonction** {code:circle()} requiert 6 informations, séparées par des virgules, entre ses parenthèses.

Le premier nombre est la position sur l'**axes x**. Le second est logiquement la position sur l'**axe y** ! Ce sont les coordonnées du cercle à l'écran !

Le troisième nombre ({code:100}) est le *rayon* du cercle. C'est la distance en pixels de son centre à sa circumference. Si le cercle a un rayon de 100 pixels, cela implique que sa largeur est de 200 pixels (car le diamètre est égal à deux fois le rayon).

Ensuite, nous trouvons le nombre de côtés. Oui, vous avez bien lu. Ce cercle a 32 côtés. Nous ne les distinguons pas car les côtés sont incroyablement petits. Nous pouvons changer ce nombre pour modifier l'aspect de notre cercle. En réalité, nous pouvons transformer notre cercle en une forme totalement différente en changeant cette valeur ! Essayer d'utiliser des valeurs différentes pour voir ce que cela implique.

-

Déplacer le Cercle

Pour faire bouger notre cercle, nous avons besoin de quelques **variables** pour stocker les positions **x** et **y**.

Modidiez votre code pour qu'il ressemble au programme ci-dessous. Son comportement n'est pas différemment pour l'instant.

```
1. x = 640
2. y = 360
3. rayon = 100
4.
5. loop
6.     clear()
7.     circle( x, y, rayon, 32, fuzepink, false )
8.     update()
9. repeat
```

Nous utilisons des **variables** pour afficher le cercle à l'écran. Aucun changement notable, sauf qu'à partir de maintenant nous pouvons les *modifier* en cours d'exécution pour déplacer le cercle.

Vous souvenez-vous de {code:+=} du tutoriel sur **les Variables** ? Nous allons l'utiliser. Nous ajoutons seulement une ligne de code, vérifiez par vous-même :

```
1. x = 640
2. y = 360
3. rayon = 100
4.
5. loop
6.     clear()
7.     circle( x, y, rayon, 32, fuzepink, false )
8.     x += 1
9.     update()
10. repeat
```

Nous avons ajouté la ligne juste après la **fonction** {code:circle()}. Cette ligne incrémente la valeur de la **variable** {code:x} de 1 à chaque répétition de la boucle ({code:loop}). Du fait de l'emploi de la **variable** {code:x}, le cercle traverse l'écran (vers la droite).

-

Lancez l'exécution du programme pour le voir ! Néanmoins, vous devez constater un petit problème... Le cercle ne s'arrête pas !

Faire rebondir Cercle

Lorsque le contenu de la **variable** {code:x} finit par être trop grand, le cercle est tracé en dehors de à l'écran. Afin de permettre à notre cercle de rebondir et de revenir, nous avons besoin de plusieurs choses.

Tout d'abord, vous devez comprendre la raison pour laquelle le cercle se déplace dans une direction. C'est liée à notre action sur les coordonnées **x** et **y**.

Pour l'instant, concentrons-nous sur l'axe *x*.

Si nous *augmentons* la coordonnée *x*, nous déplaçons le cercle vers la droite. Si nous *diminuons* la coordonnée *x*, nous le déplaçons vers la gauche.

```
8.    x += 1
```

Cette ligne est responsable du déplacement du cercle.

La vitesse à laquelle le cercle se déplace est entièrement liée à la valeur utilisée pour incrémenter la **variable** {code:x}. Actuellement, nous incrémentons celle-ci de 1 pixel à chaque fois que la ligne est exécutée. Si nous augmentons cette valeur, le cercle se déplacera plus vite. Diminuez cette valeur et le cercle se déplacera plus lentement. Tout va bien jusqu'ici ?

Lorsque cette valeur est stockée dans une **variable**, nous pouvons faire des choses intéressantes. Modifiez votre code de la façon suivante :

```
 1. x = 640
 2. y = 360
 3. rayon = 100
 4. xVitesse = 3
 5.
 6. loop
 7.     clear()
 8.     circle( x, y, rayon, 32, fuzepink, false )
 9.     x += xVitesse
10.     update()
11. repeat
```

-

Nous avons créé une nouvelle variable nommée {code:xVitesse} sur la ligne 4. Cette variable est utilisée pour contrôler la vitesse à laquelle se déplace le cercle le long de l'**axe x** (d'où son nom !).

La ligne {code:x += 1} évolue pour utiliser la **variable** xVitesse et devient {code:x += xVitesse} (voir ligne 9). Le nombre (**1**) est remplacé par la variable **xVitesse**.

Bon, maintenant nous sommes prêts à faire rebondir le cercle !

La **variable** {code:x} augmente tout le temps. L'écran a une largeur de 1280 pixels. Lorsque {code:x} devient *plus grand que* la largeur de l'écran, nous savons que le cercle est en dehors de l'écran. Nous devons alors faire agir.

Pour cela, nous avons besoin d'une **alternative "Si..."**. Voyez ci-dessous :

```
 1. x = 640
 2. y = 360
 3. rayon = 100
 4. xVitesse = 3
 5.
 6. loop
 7.     clear()
 8.     circle( x, y, rayon, 32, fuzepink, false )
 9.     x += xVitesse
10.     if x > gwidth() then
11.         xVitesse = -xVitesse
12.     endif
11.     update()
12. repeat
```

L'**alternative "Si..."** se situe sur la ligne 10. Elle vérifie si la variable {code:x} est *plus grande que* ({code:>}) {code:gwidth()} (largeur de l'écran en pixels). Lorsque vous lancez l'exécution de ce programme, le cercle doit rebondir sur le côté droit de l'écran. Si vous regardez attentivement, vous devez remarquer que quelque chose ne va pas.

La variable {code:x} détermine le centre du cercle. Ainsi, lorsque {code:x} est plus grand que la largeur de l'écran, la moitié du cercle a déjà disparu !

-

Pour que le cercle rebondisse correctement, nous devons vérifier si {code:x + rayon} est devenu plus grand que {code:gwidth()}. Pour rappel, la **variable** *rayon* correspond à la *moitié* de la largeur totale du cercle. Pour cette raison, {code: x + rayon} nous donne la position exacte du bord droit du cercle. Modifions légèrement le code :

```
x = 640
y = 360
rayon = 100
xVitesse = 3

loop
    clear()
    circle( x, y, rayon, 32, fuzepink, false )
    x += xVitesse
    if x + rayon > gwidth() then
        xVitesse = -xVitesse
    endif
    update()
repeat
```

Bien ! Maintenant, le cercle rebondit correctement… Mais nous avons un autre problème. Le cercle traverse l'autre côté de l'écran !

Lorsque {code:x} devient trop grand, le cercle sort du côté droit de l'écran, mais si {code:x} devient trop petit alors il sort du côté gauche !

Avant, {code:x} devenait plus grand que {code: gwidth ()}. Cependant, le côté gauche de l'écran est à 0 pixel. Notre problème est donc que {code:x} est devenu *plus petit que* 0.

Il y a une solution très intelligente à ce problème :

```
x = 640
y = 360
rayon = 100
xVitesse = 3

loop
    clear()
    circle( x, y, rayon, 32, fuzepink, false )
    x += xVitesse
    if x + rayon > gwidth() or x - rayon < 0 then
        xVitesse = -xVitesse
```

-

```
    endif
    update()
repeat
```

Nous pouvons simplement ajouter un **ou** ({code:or}) à notre **alternative "Si..."** pour ce cas. Le côté droit du cercle se situe en {code:x + rayon}, logiquement le côté gauche (de l'autre côté du centre du cercle) est en {code:x - rayon}. Ainsi, nous devons vérifier si {code:x - rayon} est devenu *plus petit que* 0. Si c'est le cas, nous devons faire exactement la même chose que précédemment. Nous rendons notre {code:xVitesse} négative.

Mais {code:xVitesse} est déjà négative ! Eh bien, grâce à la magie des mathématiques, lorsque vous rendez négatif un nombre déjà négatif, il devient positif !

Si {code:xVitesse} est positive, {code:xVitesse = -xVitesse} la rend négative.

Si {code:xVitesse} est négative, {code:xVitesse = -xVitesse} la rend positive !

Ceci va permettre à notre cercle de rebondir le long de l'**axe x**.

Très bien, faisons de même avec l'axe **y** pour terminer ce projet. N'hésitez pas à essayer par vous-même. Consultez le code ci-dessous si vous avez besoin d'aide :

```
x = 640
y = 360
rayon = 100
xVitesse = 3
yVitesse = 3

loop
    clear()
    circle( x, y, rayon, 32, fuzepink, false )
    x += xVitesse
    y += yVitesse
    if x + rayon > gwidth() or x - rayon < 0 then
        xVitesse = -xVitesse
    endif
    if y + rayon > gheight() or y - rayon < 0 then
        yVitesse = -yVitesse
    endif
    update()
repeat
```

-

Rendez le stupéfiant !

Ce que nous venons juste de créer, est la base du jeu **Pong** ! Il ne manque plus qu'une paire de raquettes et un score. Nous n'allons pas compléter le programme ici, mais vous pouvez en découvrir une version complète parmi les programme de démonstration.

Cependant nous pouvons rendre ce programme visuellement plus attractif.

La première chose que nous faisons dans la **boucle** est d'effacer l'écran avec la **fonction** {code:clear()}. Nous devons procéder ainsi pour voir le mouvement du cercle, sinon nous verrions tous les cercles se superposer. Cela pourrait être intéressant à tenter. Essayez de supprimer la ligne {code:clear()}.

```
 1. x = 640
 2. y = 360
 3. rayon = 100
 4. xVitesse = 3
 5. yVitesse = 3
 6.
 7. loop
 8.     circle( x, y, rayon, 32, fuzepink, false )
 9.     x += xVitesse
10.     y += yVitesse
11.     if x + rayon > gwidth() or x - rayon < 0 then
12.         xVitesse = -xVitesse
13.     endif
14.     if y + rayon > gheight() or y - rayon < 0 then
15.         yVitesse = -yVitesse
16.     endif
17.     update()
18. repeat
```

Nous voyons maintenant une ligne se tracer sur tout l'écran. Essayez de changer le {code:false} dans la ligne comportant la **fonction** {code:circle()} en {code:true} pour voir la (grande) différence.

```
 8.    circle( x, y, rayon, 32, fuzepink, true )
```

Cela affiche le contour du cercle.

-

Essayez de changer la valeur {code:32} dans la ligne {code:circle()} pour changer totalement de forme. Voyez à quoi ça ressemble avec un 3 !

```
8.    circle( x, y, rayon, 3, fuzepink, true )
```

Maintenant, faisons *vraiment* évoluer le projet.

Ce serait vraiment génial si nous pouvions changer les couleurs pendant que nous dessinons les cercles pour créer un bel effet arc-en-ciel.

Ce code va vous paraître assez difficile à comprendre, et nous n'allons certainement pas tout expliquer dans ce didacticiel, car ce n'est pas le bon endroit. Cependant, le résultat final est tellement cool que nous avons pensé qu'il serait préférable de l'inclure ici pour vous permettre d'en profiter.

Pour obtenir l'effet arc-en-ciel avec le changement de couleur, nous modifions les valeurs de rouge, de vert et de bleu (RVB) d'un **vecteur** de couleur. Confus? Ne craignez rien, vous apprendrez tout sur les **vecteurs** dans les derniers tutoriels.

Tous les paramètres importants (à modifier sans retenue) du programme sont placés dans les **variables** au début du code. Modifiez ces paramètres pour voir quelles différences cela fait !

Copiez le code ci-dessous dans l'éditeur :

```
 1. // propriétés du cercle
 2. x = gwidth() / 2
 3. y = gheight() / 2
 4. rayon = 100
 5. cotes = 6
 6. contour = false
 7.
 8. // vitesses
 9. xVitesse = 33
10. yVitesse = 66
11.
12. // variables associées à la couleur
13. cVitesse = 0.01
14. couleur = { 1, 0, 0, 1 }
15.
16. loop
17.     if couleur.r > 0 and couleur.b <= 0 then
18.         couleur.r -= cVitesse
```

-

```
19.         couleur.g += cVitesse
20.     else
21.         if couleur.g > 0 then
22.             couleur.g -= cVitesse
23.             couleur.b += cVitesse
24.         else
25.             couleur.b -= cVitesse
26.             couleur.r += cVitesse
27.         endif
28.     endif
29.     circle( x, y, rayon, cotes, couleur, contour )
30.     x += xVitesse
31.     y += yVitesse
32.     if x + rayon > gwidth() or x - rayon < 0 then
33.         xVitesse = -xVitesse
34.     endif
35.     if y + rayon > gheight() or y - rayon < 0 then
36.         yVitesse = -yVitesse
37.     endif
38.     update()
39. repeat
```

Là nous l'avons ! Exécutez ce programme pour voir apparaître des motifs arc-en-ciel à l'écran. Modifiez les **variables** {code:radius}, {code:cotes} et {code:contour} pour expérimenter différents types de forme.

Changez les **variables** {code:xVitesse} et {code:yVitesse} pour modifier le motif dessiné.

Modifiez la **variable** {code:cVitesse} pour accélérer ou ralentir l'effet de changement de couleur.

Amusez-vous bien et rendez-vous dans le prochain tutoriel !

Fonctions et mots-clés utilisés dans ce tutoriel

[circle()](../Command Reference/2D Graphics/circle.md), [clear()](../Command Reference/Screen Display/clear.md), [gWidth()](../Command Reference/Screen Display/gWidth.md), [gHeight()](../Command Reference/Screen Display/gHeight.md), loop, repeat, [update()](../Command Reference/Screen Display/update.md)

-

Tutoriel 5 : les Tableaux

Waouh, nous progressons vraiment rapidement ! Dans ce tutoriel, nous allons aborder les **tableaux**. Ce qu'ils sont, comment les utiliser et pourquoi nous le devons.

Les **tableaux** sont des outils incroyables et très puissants en programmation. En fait, quasiment tous les jeux vidéo du monde utilisent beaucoup de tableaux pour garder une trace de tout ce dont ils ont besoin.

Un **tableau** est une table de **variables**. Nous pouvons donner à chaque position dans le tableau une valeur et l'utiliser ultérieurement dans le programme.

Les **tableaux** sont utilisés pour une grande variété de choses, des inventaires aux cartes spatiales pour placer des étoiles dans votre arrière-plan. Dans ce premier tutoriel, nous allons créer un simple programme "tireur de cartes" (cartomancien).

Un "tireur de cartes" a besoin d'avoir une sélection de cartes stockée de sorte à nous permettre d'en tirer une au hasard pour répondre au joueur.

Avant d'écrire notre mécanisme de sélection de la carte réponse, nous devons commencer par créer notre tableau. Pour ce faire, nous devons utiliser le mot {code:array} (tableau en anglais).

Taper la ligne suivante dans l'éditeur de code **FUZE^4 Nintendo Switch** :

```
1. array cartes[4]
```

Cette ligne crée un **tableau** vide. Imaginez-le comme une commode avec des tiroirs vides. Notez l'emploi des crochets sur cette ligne. Lorsque nous *créons* un tableau ou *accédons* à l'un de ces emplacements, nous devons toujours utiliser des crochets. Puisque nous avons mis un **4** entre les crochets, nous disposons de **4** emplacements de stockage.

Nous avons appelé notre **tableau** "cartes". Vous pouvez le nommer comme vous le souhaitez, toutefois c'est une bonne habitude de choisir pour vos **variables** et **tableaux** des noms qui ont du sens (pour vous et les autres).

Ensuite, nous stockons quelques informations dans les *emplacements* de notre **tableau**. Dans chacun de ces emplacement, nous allons stocker une déclaration.

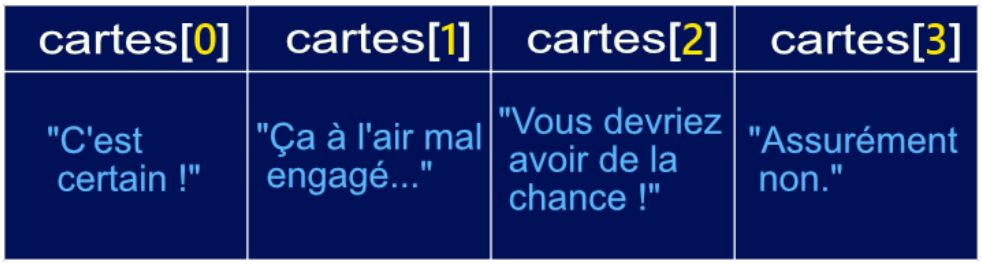
-

```
1. array cartes[4]
2. cartes[0] = "C'est certain !"
3. cartes[1] = "Ça a l'air mal engagé..."
4. cartes[2] = "Vous devriez avoir de la chance !"
5. cartes[3] = "Assurément non."
```

Modifiez votre code de sorte à ce qu'il ressemble au programme ci-dessus. Sentez-vous libre de copier/coller le code si vous n'avez pas envie de tout taper !

Le programme ci-dessus stocke les quatre textes différents dans les quatre emplacements de notre tableau. Ils sont respectivement étiquetés **0**, **1**, **2** et **3**.

L'image ci-dessous devrait vous aider à vous faire une meilleure idée :



Comme vous pouvez le voir, chaque position de la table est étiquetée par un nombre (de 0 à 3). Grâce à cela, nous pouvons aisément accéder à n'importe quel emplacement et récupérer l'information stockée.

Il existe une autre façon de disposer notre tableau avec un résultat identique. Voir ci-dessous :

```
1. cartes = [
2.     "C'est certain !",
3.     "Ça a l'air mal engagé...",
4.     "Vous devriez avoir de la chance !",
5.     "Assurément non."
6. ]
```

Les deux écritures produisent exactement le même tableau. C'est à vous d'adopter l'écriture qui vous paraît la plus compréhensible !

Ci-dessous, nous avons effectué quelques ajouts au programme. Ajoutez les lignes 6 à 19 et lancez son exécution.

-

```
  1. array cartes[4]
  2. cartes[0] = "C'est certain !"
  3. cartes[1] = "Ça a l'air mal engagé..."
  4. cartes[2] = "Vous devriez avoir de la chance !"
  5. cartes[3] = "Assurément non."
  6. print( "Bienvenue chez le tireur de carte, posez-moi une
question ! \n" )
  7. update()
  8. question = input( "Tapez votre question ici." )
  9. sleep( 1 )
 10. print( "Voulez-vous connaître la réponse des cartes ? \n" )
 11. update()
 12. sleep( 1 )
 13. print( "La réponse à votre question est... \n" )
 14. update()
 15. sleep( 1 )
 16. num = random( 4 )
 17. print( cartes[num] )
 18. update()
 19. sleep( 2 )
```

Maintenant nous pouvons jouer ! Avec quelques commandes d'attente ({code:sleep}, nous pouvons vraiment faire monter la tension avant de donner la réponse !

Accéder au tableau

Pour afficher une de nos réponses à l'écran, nous devons *accéder* au **tableau**. La ligne 17 est celle où nous le faisons.

Pour proposer une réponse aléatoire, nous utilisons la **fonction** {code:random()} qui retourne un nombre aléatoire. Nous pouvons ensuite l'utiliser comme *index* dans notre **tableau**.

Nous stockons le nombre aléatoire dans une **variable** appelée {code:num} et utilisons cette **variable** pour accéder au **tableau** sur la ligne 17 :

```
 17. print( cartes[num] )
```

Défi

-

Pourriez-vous ajouter quelques réponses au **tableau** ? Pour ce faire, vous devrez créer des lignes de code supplémentaires. Essayez d'ajouter 3 autres réponses.

Astuce: N'oubliez pas de regarder cette ligne :

```
16. num = random( 4 )
```

Résumé

Un **tableau** est une table de **variables** utilisée pour stocker de l'information.

Chaque emplacment dans le tableau est associé à un nombre que nous pouvons utiliser pour *accéder* à l'information qui y est stockée.

Les **tableaux** peuvent être utilisés *partout* ! Les ordinateurs gèrent les écrans en utilisant un tableau gigantesque, où chaque élément du **tableau** est un unique pixel.

Restez avec nous pour le prochain projet sur les **tableau** dans lequel nous allons jeter des formes autour de l'écran !

Fonctions et mots-clés utilisés dans ce tutoriel

array, input()(../Command Reference/Input/input.md), [print()](../Command Reference/Text Handling/print.md), [sleep()](../Command Reference/Time and Date/sleep.md), [update()](../Command Reference/Screen Display/update.md)

a

Tutoriel 6 : Utilisation des contrôleurs

Ravi de vous revoir !

Dans ce tutoriel, nous allons apprendre à utiliser les **contrôleurs Joy-Con** dans nos programmes.

-

Si nous voulons commencer à écrire quelque chose qui s'apparente à un jeu, nous devrons utiliser les contrôleurs Joy-Con tôt ou tard !

Pour ce faire, nous aurons besoin de notre bonne vielle **alternative "Si..."**. Après tout, nous vérifions **si** un bouton est enfoncé !

Voici comment nous allons faire. Comme d'habitude, nous commençons simplement.

Entrez le code suivant dans l'éditeur de code **FUZE^ 4 Nintendo Switch** :

```
1. loop
2.     clear(black)
3.
4.     joy = controls(0)
5.
6.     if joy.a then
7.         print("Vous appuyez sur le bouton A !")
8.     endif
9.
10.    update()
11. repeat
```

Voici un programme simple et efficace pour vous expliquer. Tout ce que nous voulons, c'est afficher le texte "Vous appuyez sur le bouton A !" lorsque nous appuyons sur le bouton A.

Notez que ce programme est totalement inclus dans une **boucle**. Nous voulons que notre programme fonctionne en permanence. La première chose à faire dans notre **boucle** est d'effacer l'écran. Nous employons les **fonctions** {code: clear()} et {code:update()} car nous modifions ce qui doit apparaître à l'écran.

Maintenant, passons à la partie importante.

Regardez la ligne 4. Nous *appelons* une **fonction** appelée {code:controls()}. Cette **fonction** nous donne l'*état actuel de tous les contrôleurs* et c'est exactement ce dont nous avons besoin pour utiliser les contrôleurs Joy-Con dans notre programme.

Nous stockons le résultat de la **fonction** {code:controls()} dans une **variable** appelée {code:joy}.

-

Maintenant que cela est fait, nous pouvons accéder à l'état de *n'importe* quel bouton en utilisant {code:joy.a}, {code:joy.b}, {code:joy.x}, etc.

La ligne 6 contient notre **alternative "Si..."** qui utilise l'état mémorisé dans la **variable** {code:joy} du contrôleur. Nous vérifions **si** le bouton A est enfoncé avec {code:if joy.a then}.

Soit plus précisément...
Il est important de comprendre que lorsqu'un ordinateur évalue une **alternative "Si..."**, il ne peut interpréter que deux résultats : {code:true} (vrai:1) ou {code:false} (faux:0).

Lorsque FUZE exécute la ligne 6, il vérifie si la valeur de {code:joy.a} est {code:true} (1) ou {code:false} (0).

Si vous *appuyez* sur le bouton A, la valeur de {code:joy.a} est {code:true}.

Si le bouton A *n'est pas enfoncé*, la valeur de {code:joy.a} est {code:false}.

L'expression {code:if joy.a then} comporte un sous-entendu. En réalité, la ligne 6 se lit : {code:if joy.a == true then}.

Défi
Pouvez-vous ajouter une **alternative "Si..."** à ce programme pour vérifier un autre bouton du contrôleur ? Vous devez également écrire une ligne de texte pour indiquer à l'utilisateur son état : pressé ou non.

Consultez la page du guide de l'utilisateur pour la **fonction** {code:controls()} pour voir *toutes* les entrées disponibles sur les contrôleurs Joy-Con. Vous pouvez trouver cette page juste ici.

Déplacer un cercle à l'aide du Stick

Vous souvenez-vous du tutoriel sur l'écran ? Dans ce projet, nous avons appris à déplacer un cercle à l'écran en utilisant des **variables**.

Et si nous déplacions maintenant le cercle en utilisant un stick du **Joy-Con** ? En fait c'est incroyablement simple.

Nous aurons besoin de la **fonction** {code:controls ()}. Tout d'abord, rappelons-nous le modèle du programme de base.

-

```
x = gwidth() / 2
y = gheight() / 2
rayon = 100

loop
    clear()
    circle( x, y, rayon, 32, fuzepink, false )
    update()
repeat
```

Nous avons une simple **boucle** qui affiche un cercle à l'écran aux coordonnées des **variables x** et **y** définies au début du programme.

Ajoutons la **fonction** {code:controls()} et définissons une **variable** pour l'utiliser.

```
x = gwidth() / 2
y = gheight() / 2
rayon = 100

loop
    clear()
    joy = controls( 0 )
    circle( x, y, rayon, 32, fuzepink, false )
    update()
repeat
```

Comme précédemment, nous avons créé une **variable** appelée {code:joy} pour stocker le résultat de la **fonction** {code:controls()}.

On accède au stick gauche avec {code:.lx} et {code:.ly} pour les axes **x** et **y** du stick.

Il est important de comprendre exactement comment la **fonction** {code:controls()} fonctionne avec le contrôleur. Regardez l'image ci-dessous :

-

Comme vous pouvez le voir, les valeurs d'axe renvoyées par la **fonction** {code:controls()} qui represente le stick gauche sont **nulles** (=0) lorsqu'il *n'est poussé dans aucune* direction.

Lorsque vous appuyez sur l'un des côtés, la valeur change vers 1 ou -1. Il y a beaucoup de valeurs intermédiaires : Un axe comporte 65 000 positions différentes !

Si la valeur {code:.lx} est *supérieure à* 0, nous savons que le stick est poussé *vers* un nombre positif, ce qui nous indique que le stick est poussé vers la droite. Si la valeur est *inférieure à* 0, nous savons qu'il est poussé vers un nombre négatif et qu'il est donc poussé à gauche.

Utilisons ce nouveau savoir afin de modifier notre code pour pouvoir déplacer le cercle dans les deux sens sur l'axe **x**.

```
 1. x = gwidth() / 2
 2. y = gheight() / 2
 3. rayon = 100
 4.
 5. loop
 6.     clear()
 7.     joy = controls( 0 )
 8.     x += joy.lx
 9.     circle( x, y, rayon, 32, fuzepink, false )
10.     update()
11. repeat
```

Voyez ! Nous n'avons ajouté qu'une seule ligne de code.

Sur la ligne 8, nous ajoutons simplement la valeur du stick gauche à la **variable** {code:x}. Si le stick est poussé vers la droite, nous avons un nombre positif et le cercle se déplace vers la droite. Si le stick est poussé à gauche, nous avons un nombre négatif et le cercle se déplace vers la gauche.

Maintenant, comme ces chiffres sont très petits, notre cercle se déplacera incroyablement lentement. Pas trés convaincant. Introduisons une nouvelle **variable** {code:vitesse} que nous utilisons comme multiplicateur.

```
1. x = gwidth() / 2
2. y = gheight() / 2
3. rayon = 100
```

-

```
 4. vitesse = 8
 5.
 6. loop
 7.     clear()
 8.     joy = controls( 0 )
 9.     x += joy.lx * vitesse
10.     circle( x, y, rayon, 32, fuzepink, false )
11.     update()
12. repeat
```

Avec ce changement, nous constatons un effet sur le mouvement beaucoup plus important.

Sur la ligne 9, nous augmentons la **variable** {code:x} de la valeur du joystick gauche *multipliée par* la **variable** {code:vitesse}. Cela nous donne un mouvement 8 fois plus rapide. Essayez de changer la valeur de la **variable** {code:vitesse} pour constater son action.

Bon, déplaçons également le cercle sur l'axe **y** ! Celui-ci est un peu plus compliqué à gére.

```
 1. x = gwidth() / 2
 2. y = gheight() / 2
 3. rayon = 100
 4. vitesse = 8
 5.
 6. loop
 7.     clear()
 8.     joy = controls( 0 )
 9.     x += joy.lx * vitesse
10.     y -= joy.ly * vitesse
10.     circle( x, y, rayon, 32, fuzepink, false )
11.     update()
12. repeat
```

La ligne 10 comporte notre nouvelle ligne. Notez que nous utilisons {code:-=} au lieu de {code:+=} pour l'axe **y**. En effet, le haut de notre écran correspond à 0 sur l'axe **y**, mais l'axe de contrôle du manche **y** est *positif* lorsqu'il est poussé vers le haut. Jetez un œil à notre image du stick :

-

Lorsque nous déplaçons le stick vers le haut, nous recevons un nombre positif. Afin de déplacer le cercle vers le haut sur l'écran, nous devons *diminuer* la valeur de la **variable** {code: y} et non l'augmenter. C'est pour cette raison que nous utilisons {code:-=}.

Mettre des limites

Tout ce dont nous avons besoin maintenant, c'est quelques **alternatives "Si..."** pour empêcher notre cercle de sortir de l'écran. Comme dans le tutoriel sur l'écran, nous vérifions les **variables** {code:x} et {code:y}, mais au lieu d'inverser le sens, nous redéfinirons celles-ci.

Commencer par arrêter le cercle sur les bords de l'axe **x**.

```
x = gwidth() / 2
y = gheight() / 2
rayon = 100
vitesse = 8

loop
    clear()
    joy = controls( 0 )
    x += joy.lx * vitesse
    y -= joy.ly * vitesse
    if x + rayon > gwidth() then
        x = gwidth() - rayon
    endif
    if x - rayon < 0 then
        x = rayon
    endif
    circle( x, y, rayon, 32, fuzepink, false )
    update()
repeat
```

-

Les lignes 11 à 16 contiennent notre première série d'**alternatives "Si..."**. Il suffit de vérifier si le bord droit du cercle ({code: x+rayon}) est devenu *supérieur au* bord droit de l'écran ({code:gwidth()}). Si c'est le cas, on redéfinit {code:x} pour qu'il soit *égal à* au bord de l'écran *moins* le rayon ({code:x = gwidth() - rayon}).

Et inversément pour le côté gauche de l'écran... Nous vérifions si {code:x - radius} est devenu *inférieur à* 0, et si c'est le cas, nous redéfinissons {code:x} comme étant le côté gauche (0) *plus* le rayon du cercle. On peut simplement écrire ceci sous la forme {code:x = radius}.

Assez facile ! Faisons maintenant la même chose pour l'axe **y** :

```
 1. x = gwidth() / 2
 2. y = gheight() / 2
 3. rayon = 100
 4. vitesse = 8
 5. 
 6. loop
 7.     clear()
 8.     joy = controls( 0 )
 9.     x += joy.lx * vitesse
10.     y -= joy.ly * vitesse
11.     if x + rayon > gwidth() then
12.         x = gwidth() - rayon
13.     endif
14.     if x - rayon < 0 then
15.         x = rayon
16.     endif
17.     if y + rayon > gheight() then
18.         y = gheight() - rayon
19.     endif
20.     if y - rayon < 0 then
21.         y = rayon
22.     endif
23.     circle( x, y, rayon, 32, fuzepink, false )
24.     update()
25. repeat
```

Là nous l'avons ! Ce code de mouvement peut être appliqué à n'importe quel programme. Dans des tutoriels ultérieurs, nous aborderons des techniques de mouvement plus avancées, mais la compréhension de ce simple type de mouvement ouvre un monde d'expérimentation dans vos propres projets !

Rendez-vous dans le prochain tutoriel !

-

Fonctions et mots-clés utilisés dans ce tutoriel

[clear()](../Command Reference/Screen Display/clear.md), [controls()](../Command Reference/Input/controls.md), endIf, if, input()(../Command Reference/Input/input.md), loop, [print()](../Command Reference/Text Handling/print.md), repeat, then, [update()](../Command Reference/Screen Display/update.md)

Tutoriel 7 : les boucles “Pour...”

Bonjour !

Nous savons comment les **boucles** fonctionnent et nous maîtrisons **les variables**. Nous pouvons maintenant découvrir un autre outil incroyablement utile pour la programmation : les **boucles "Pour..."**.

Les **boucles "Pour..."** sont utilisées tout le temps en programmation. Quelques fois pour l’animation, quelques fois comme compteur, d’autres fois pour parcourir très rapidement un tableau, les **boucles "Pour..."** sont particulièrement efficaces et polyvalentes.

```
1. for i = 0 to 10 loop
2.     print( i )
3.     update()
3. repeat
4. sleep( 2 )
```

Le programme ci-dessus est un compteur de 0 à 9. Les **boucles "Pour..."** sont spéciales car elles définissent une **variable** exploitable à l’intérieur d’une **boucle**.

Dans notre exemple, nous avons défini une **variable** appelée {code:i}. A la première occurrence de cette **boucle**, {code:i} est égal à 0. Comme nous le savons, lorsque FUZE rencontre la commande {code:repeat}, il retourne à la dernière ligne comportant l’instruction {code:loop}. Cependant, maintenant {code: i} est égal à 1. La prochaine fois que la **boucle** sera exécutée, {code:i} sera égal à 2. Cela continuera d’augmenter de 1, à chaque fois, jusqu’à 9.

-

Lorsque {code: i} sera égal à 10, la condition sera remplie et la **boucle** se terminera !

Note: La **boucle** compte *jusqu'à* la dernière valeur *non incluse*. Ainsi notre exemple, {code:for i = 0 to 10 loop} va compter de 0 à 9 *sans atteindre* 10.

Utiliser une boucle "Pour.." pour faire de la musique

Alors, pourquoi vouloir faire cela ? Vous trouverez ci-dessous un exemple assez intéressant d'utilisation d'une **boucle "Pour..."** créant des sons intéressants à l'aide de notre **fonction** {code: playNote()}. Changez votre code pour qu'il ressemble au programme ci-dessous et lancez-le. Assurez-vous d'avoir monté un le volume !

```
1. for i = 0 to 900 loop
2.     playNote( 0, 1, i, 1, 10, 0.5 )
3.     update()
4.     sleep( 0.01 )
5. repeat
```

Ici, nous utilisons notre variable {code:i} à nouveau, mais cette fois nous comptons jusqu'à 900 (au lieu de 10). Souvenez-vous que nous n'allons jamais attendre 900. {code:i} va seulement atteindre 899.

La **fonction** {code:playNote()} est assez sympathique à maîtriser car elle permet de créer votre propre musique ! {code:playNote()} joue une note à une certaine *fréquence* (pitch). Dans l'exemple, notre fréquence est déterminée par la **variable** {code:i}.

Lors de la première itération de la **boucle "Pour..."**, notre {code:playNote()} va jouer une fréquence de 0hz. La fois suivante, la fréquence sera de 1Hz, puis 2Hz et ainsi de suite jusqu'à 899Hz. Ce qui va produire une séquence de notes dans la **boucle**.

Nous utilisons également la **fonction** {code:sleep()} avec un *très* court délai de 0.01 seconde. Avec un délai aussi court, les notes sonnent comme un balayage, un peu comme une alarme ou une sirène.

Et pour jouer une série de notes plus "musicales" ? Eh bien, nous avons besoin de deux choses: un délai plus long et d'un **pas**.

Nous utilisons un **pas** (step) dans une **boucle "Pour..."** pour compter avec une quantité spécifique. Regardez ci-après :

-

```
1. for i = 0 to 900 step 100 loop
2.     playNote( 0, 1, i, 1, 10, 0.5 )
3.     update()
4.     sleep( 0.1 )
5. repeat
```

Dans cet exemple, notre **boucle "Pour..."** va toujours compter de 0 vers 900, mais par saut de 100 à chaque fois. Cela signifie que lors de la première itération {code:i} vaut toujours 0, mais à la suivante {code:i} vaudra 100, puis 200, 300 et ainsi de suite. Cette fois, la **boucle** se terminera lorsque {code:i} atteindra 800.

Nous avons utilisé un délai plus long pour nous obtenir une durée de note plus longue. Le résultat ressemblera à une échelle ascendante. Beaucoup plus musical !

Expérimentez différents **pas** et valeurs de début et de fin pour obtenir des résultats variés !

Utiliser une boucle “Pour...” avec des formes

Un autre exemple d’utilisation d’une boucle For est l’animation. Supposons que nous voulions dessiner une ligne, pixel par pixel, à l’écran.

```
1. for x = 0 to gwidth() loop
2.     box( x, gheight() / 2, 1, 1, white, false )
3.     update()
3. repeat
```

Notez que dans cet exemple, nous nommons notre **variable** {code:x}. Elle est nommée ainsi car nous l’utilisons pour changer la *position de l’axe x* de la boîte. Vous pouvez bien sûr appeler vos **variables** comme bon vous semble !

Notre **variable** {code:x} comptera de 0 au nombre maximal de pixels le long de *l’axe x* de l’écran moins un. En effet, la valeur de fin {code:gwidth()} n’est pas *incluse*.

Lorsque nous utilisons la **fonction** {code:box()}, nous devons mettre certaines informations entre les parenthèses. Les deux premières sont les coordonnées **x** et **y** de l’emplacement où nous voulons que notre boîte apparaisse à l’écran. Ensuite, nous avons la largeur et la hauteur de la boîte. Dans notre exemple, notre boîte a une largeur et une hauteur de seulement un pixel. Une très petite boîte en fait ! La

-

prochaine information est la **couleur** que nous voulons donner à notre boîte. Et enfin, nous spécifions si nous voulons voir le contour d'une boîte ({code:true}) ou une boîte remplie ({code:false}).

Pour la position **x**, nous utilisons notre nombre croissant {code: x}. Pour la coordonnée **y**, nous utilisons {code:gheight()/2}, ce qui nous donne le point central de l'axe **y** de l'écran.

En bref, ce petit programme trace une ligne au milieu de l'écran, pixel par pixel. Regardez-le dans toute sa splendeur !

Changez votre code pour qu'il ressemble au programme ci-dessous. Essayer de comprendre ce qui se passera avant de lancer le programme !

```
1. for y = 0 to gheight() loop
2.     circle( random( gwidth() ), y, 100, 32, fuzepink, true )
3.     update()
4. repeat
```

{code:gheight()} comme nous le savons est une **fonction** qui nous donne la hauteur de l'écran en pixels.

En mode de portable, ce nombre est de 720. En mode TV, sa valeur passe à 1080.

Selon la manière dont vous utilisez votre console Nintendo Switch, vous verrez apparaître 720 ou 1080 cercles sur votre écran, un par un, en descendant de l'écran.

Puisque la partie **x** de la **fonction** {code:circle()} est un nombre aléatoire choisi parmi le nombre total de pixels sur la largeur de l'écran, à chaque itération un nouveau cercle apparaît à une position aléatoire le long de *l'axe x*.

Utiliser une boucle "Pour..." avec un tableau

L'une des applications les plus utiles et pratiques d'une **boucle "Pour..."** est son utilisation pour parcourir un **tableau** d'informations.

Voyons un exemple d'utilisation d'une **boucle "Pour..."** avec un **tableau** pour afficher de nombreux noms différents à l'écran.

-

Tout d'abord, nous devons créer un **tableau** et le remplir avec des informations.

```
1. array noms[5]
2. noms[0] = "Dave "
3. noms[1] = "Kat "
4. noms[2] = "Luke "
5. noms[3] = "Jon "
6. noms[4] = "Rob "
```

Avant d'aller plus loin, rappelons-nous ce que nous avons fait. Nous avons créé une table d'information. Chaque partie de la table a un numéro.

Utilisons maintenant une **boucle "Pour..."** afin d'afficher les noms à l'écran.

```
 1. array noms[5]
 2. noms[0] = "Dave "
 3. noms[1] = "Kat "
 4. noms[2] = "Luke "
 5. noms[3] = "Jon "
 6. noms[4] = "Rob "
 7. loop
 8.     clear()
 9.     for i = 0 to 5 loop
10.         print( noms[i] )
11.     repeat
12.     update()
13. repeat
```

Nous avons une **boucle "Pour..."** sur les lignes 9 à 11. Elle permet d'afficher les 5 noms à l'écran. Sans utiliser de **boucle "Pour..."**, nous aurions eu besoin de 5 lignes {code:print()} différentes pour obtenir ce que nous souhaitons.

L'astuce réside en l'utilisation de la variable {code:i} **croissante** dans la ligne {code:print()}. Au premier parcours de la, la ligne s'exécute comme suit :

```
10. print( noms[0] )
```

Au parcours suivant, {code:i} est égal à 1, la ligne devient :

```
10. print( noms[1] )
```

-

Cela continue jusqu'à ce que tous les noms du **tableau** soient affichés.

Résumé

Une **boucle "Pour..."** est une **boucle** qui se répète un nombre défini de fois.

Nous définissons une **variable** dans la **boucle** qui augmente (ou diminue) à chaque répétition.

Il existe de très nombreuses applications pour les **boucles "Pour..."**. Il est impossible de toutes les couvrir ici. Gardez toutefois un œil attentif sur les prochains tutoriels et programmes de démonstration de **FUZE^ 4 Nintendo Switch** pour voir comment elles peuvent être utilisées.

Excellent ! Votre niveau progresse ! Rendez-vous dans le prochain tutoriel !

Fonctions et mots-clés utilisés dans ce tutoriel

[box()](../Command Reference/2D Graphics/box.md), [circle()](../Command Reference/2D Graphics/circle.md), [gWidth()](../Command Reference/Screen Display/gWidth.md), [gHeight()](../Command Reference/Screen Display/gHeight.md), for, loop, playNote()(../Command Reference/Sound and Music/playNote.md) [print()](../Command Reference/Text Handling/print.md), repeat, [sleep()](../Command Reference/Time and Date/sleep.md), step

Tutoriel 8 : les Fonctions

Vous sentez-vous funky ?

Toutes blagues à part, c'est une affaire très sérieuse. Les **fonctions** sont des outils extrêmement importants dans la programmation et on a vraiment l'impression de gravir un échelon quand on les comprend !

-

Presque tous les langages de programmation contiennent déjà de nombreuses **fonctions**. Si vous n'êtes pas sûr de ce qu'elles sont exactement, il est fort probable que vous les ayez déjà utilisées de nombreuses fois sans vous en rendre compte !

Qu'est-ce qu'une Fonction ?

Une **fonction** est comme un mini-programme que nous pouvons utiliser encore et encore. Il existe deux types de **fonctions**. Il y a les **fonctions intégrées** qui existent déjà, et il y a les **fonctions utilisateur** que nous créons.

Vous pouvez repérer une **fonction** très facilement car elle est toujours suivie d'une paire de parenthèses {code:()}. Ces parenthèses peuvent contenir de l'information ou être vide ({code:()}).

La **fonction** la plus simple qui nous connaissons et aimons est {code:print}. Nous utilisons {code:print()} tout le temps pour afficher des mots à l'écran :

```
print( "Comme cela !" )
```

Entre les parenthèses de {code:print()} nous mettons les informations nécessaires à la **fonction**.

{code:print()} est sans intérêt jusqu'à ce que nous lui disions *ce que nous voulons afficher* ! Ainsi, nous devons transmettre notre texte à la **fonction**.

les Arguments

Quelques **fonction** nécessitent plus d'une seule information. Par exemple, {code:printAt()} a non seulement besoin du texte à afficher, mais également de la **position** de ce texte à l'écran :

```
printAt( 0, 0, "message" )
```

Les pièces d'information entre les parenthèses sont séparées par des virgules et se nomment des **arguments**. La fonction {code:printAt()} **doit** avoir ses **arguments** organisés comme suit :

-

```
printAt( x, y, texte )
```

Dans notre exemple {code:printat(0, 0, message)} affiche le mot “message” à la position **x** 0 et **y** 0.

Êtes-vous toujours avec nous ? Biensûr vous l’êtes ! Oh, c’est facile, vous dites ?

Bien, engageons-nous plus avant.

Une fonction est *vraiment* une pièce de code séparée que l’ordinateur doit retrouver et utiliser. Lorsqu’un ordinateur lit {code:print("Bonjour !")}, il prend le texte “Bonjour !” et le **transmet** au code qu’il doit exécuter pour permettre aux mots de s’afficher sur l’écran.

C’est la même chose pour toutes les **fonctions**. Par exemple, considérons la **fonction** {code:box()}.

```
box( x, y, largeur, hauteur, couleur, remplir )
```

Comme vous pouvez le voir, les **arguments** de la fonction box sont également ordonnés. Les deux premiers nombres entre parenthèses sont les coordonnées **x** et **y** du coin supérieur gauche de la boîte. Ensuite, les 3e et 4e nombres sont utilisés comme largeur et hauteur de la boîte. Suivent ensuite la couleur de la boîte et le dernier argument qui précise si la boîte doit être remplie ou seulement délimitée.

Toutes ces informations sont **transmises** à la **fonction** box et sont exploitées pour afficher la boîte à l’écran !

Parenthèses vides

Il est important de mentionner que toutes les **fonctions** *ne requierent pas* d’arguments. Quelques fois, elles **renvoient** une information, exemple : les **fonctions** {code:gwidth()} et {code:gheight()}.

Remarquez-vous les paranthèses vides de {code:gwidth()} et {code:gheight() ? Nous n’avons pas besoin de **transmettre** d’argument et elles **renvoient** un nombre. Ce nombre est respectivement la largeur ou la hauteur de l’écran en pixels.

-

Cela peut paraître un peu étrange d'avoir à ajouter des paranthèses vides (sans rien entre elles), mais souvenez-vous, les **fonctions** nécessitent *toujours* une paire de paranthèses (avec ou sans information).

les Fonctions définies par l'utilisateur

Toujours là ?

Tout ceci n'était que l'introduction !

Il est très courant en programmation d'avoir à faire plusieurs fois la même chose. C'est le contexte idéal pour une **fonction** définie par l'utilisateur**, ce qui est une façon élégante de dire une **fonction** personnalisée.

Nous pouvons créer une **fonction** personnalisée pour faire n'importe quoi. Commençons gentiment et simplement. Nous souhaitons afficher le mot "Bonjour !" à l'écran en bleu avec une taille spécifique.

Sans notre propre fonction, nous pourrions faire quelque chose comme ça :

```
1. textSize( 50 )
2. ink( blue )
3. print( "Bonjour !" )
4. update()
```

Chaque fois que nous voudrons le faire, nous devrons écrire ces 4 lignes encore et encore. Jusqu'à ce que nous créions une **fonction** pour le faire !

```
 1. function fuzePrint()
 2.     textSize( 50 )
 3.     print( "Bonjour !" )
 4.     update()
 5. return void
```

Une fois la portion de code ci-dessus écrite, nous pouvons simplement utiliser {code:fuzePrint()} pour obtenir le même résultat (chaque fois que cela est nécessaire).

Nous pouvons améliorer cette **fonction**. Imaginons que nous souhaitions afficher n'importe quel texte, de n'importe quelle taille avec n'importe quelle couleur !

-

En **transmettant** quelques **variables** à notre **fonction**, nous pouvons le faire très facilement :

```
1. function fuzePrint( texte, taille, couleur )
2.     textSize( taille )
3.     ink( couleur )
4.     print( texte )
5.     update()
6. return void
```

Maintenant notre **fonction** {code:fuzePrint()} a trois **arguments** (pièces d'information entre les parenthèses). Dans notre code, nous pouvons taper :

```
1. fuzePrint( "Bonjour !", 50, blue )
```

Lorsque nous utilisons cette **fonction**, les informations entre les paranthèses sont **transmises** à la **fonction** {code:fuzePrint()} que nous avons créée. "Bonjour !" est stocké dans la **variable** {code:texte} et est utilisé dans la ligne {code:print()}. Le nombre 50 est stocké dans la **variable** appelée {code:taille} et est utilisé dans la ligne {code:textSize()}. La couleur est stockée dans la **variable** appelée {code:couleur} et est exploitée dans la ligne {code:ink()}.

Nous pouvons maintenant utiliser notre nouvelle **fonction** {code:fuzePrint()} encore et encore dans notre programme et ainsi nous épargner beaucoup de tracas.

Renvoyer Void

Vous avez sans doute remarqué l'étrange ligne dans la **fonction** {code:fuzePrint()} que nous avons créée.

```
return void
```

Toutes les **fonctions** doivent renvoyer quelque chose. Même lorsqu'il n'y a rien à renvoyer !

À la fin de la **fonction** que vous avez créée, vous devez indique ce que vous voulez renvoyer.

-

Si votre **fonction** ne doit rien renvoyer, écrivez simplement la ligne {code:return void}.

Quelques fois nous voulons que la **fonction** renvoie quelque chose de calculé dans la **fonction** elle-même. À titre d'illustration, voici une **fonction** personnalisée qui convertie les mètres en centimètres :

```
1. function metre2cm( nombre )
2. return nombre * 100
```

Mettez simplement le nom de la variable que vous voulez renvoyer à la fin de la **fonction** ! Vous pouvez aussi effectuer des opérations simple, voir ci-dessus.

Essayer d'écrire quelques **fonctions** de votre propre crû.

On se retrouce dans le tutoriel suivant !

Fonctions et mots-clés utilisés dans ce tutoriel

[box()](../Command Reference/2D Graphics/box.md), [circle()](../Command Reference/2D Graphics/circle.md), [clear()](../Command Reference/Screen Display/clear.md), [controls()](../Command Reference/Input/controls.md), for, [gWidth()](../Command Reference/Screen Display/gWidth.md), [gHeight()](../Command Reference/Screen Display/gHeight.md), loop, [random()](../Command Reference/Arithmetic/random.md), repeat, [update()](../Command Reference/Screen Display/update.md)

Tutoriel 9 : Et, Ou, Non

Dans ce tutoriel, nous allons aborder de façon plus avancée **l'alternative "Si..."** et ce que l'on peut en faire.

Nous savons que si nous voulons vérifier une condition, nous devons utiliser une **alternative "Si..."**. Par exemple :

-

```
1. dave = true
2.
3. if dave then
4.     print( "Hourra !" )
5. endif
```

Simple et idéal pour commencer. Nous avons une **variable** appelée {code:dave} que nous avons mis à **vrai** ({code:true}).

Notre **alternative "Si..."**, sur la ligne 3 vérifie si la **variable** {code:dave} est **vrai**. Si elle l'est, nous affichons "Hourra !". Fantastique.

Introduisons un peu de compléxité. C'est le moment de parler de deux nouveaux mots : **et** ({code:and}) et **ou** ({code:or}).

Ces mots sont appelés des **opérateurs**. Nous pouvons les utiliser pour vérifier plusieurs conditions simultanément dans notre **alternative "Si..."**.

Vérifiez par vous-même.

```
1. dave = true
2. kat = true
3.
4. if dave and kat then
5.     print( "Hourra !" )
6. else
7.     print( "Oh, non..." )
8. endif
```

Tout d'abord, nous avons introduit une nouvelle **variable** dans l'équation. C'est {code:kat} !

Maintenant notre **alternative "Si..."** vérifie deux conditions. Nous vérifions si {code:dave} est **vrai** ({code:true}) **et** si {code:kat} is **vrai** ({code:true}). Cette **alternative "Si..."** est soit **vrai** ({code:true}), soit **fausse** ({code:false}). Elle ne sera **vraie** que si les deux variables sont **vraies**. Si nous passons l'une ou l'autre des **variables** à **faux** ({code:false}), notre **alternative "Si..."** sera également **fausse** ({code:false}) et nous afficherons "Oh, non...".

L'opérateur **et** ({code:and}) fonctionne de façon très similaire à l'emploi que nous en faisons lorsque nous parlons, il combine les conditions.

-

Ou ({code:or}) est un peu différent, mais se comporte d'une façon très similaire à l'usage oral. **Ou** vérifie si *l'une ou l'autre* des conditions est **vrai** ({code:true}).

Pouvez-vous voir la différence avec le code ci-dessous ?

```
1. dave = true
2. kat = true
3.
4. if dave or kat then
5.     print( "Hourra !" )
6. else
7.     print( "Oh, non..." )
8. endif
```

Si nous exécutons ce code, nous obtiendrons "Hourra !" car les deux **variables** sont **vraies** ({code:true}). Puisque **l'alternative "Si..."** vérifie si *l'une ou l'autre* est **vraie** ({code:true}), si nous passons l'une des **variables** à **faux** ({code:false}), nous obtiendrons encore "Hourra !".

Pour que notre **alternative "Si..."** produise "Oh, non...", nous devons passer l'ensemble des **variables** {code:dave} et {code:kat} à **faux** ({code:false}).

Résumé

Avant d'aborder quelques cas pratiques d'emploi de **et** et **ou**, passons en revue les notions présentées.

Essayez de penser aux opérateurs **et** et **ou** comme à des entités qui vous donnent pour réponse : vrai ou faux.

Et

Regardez les exemples ci-après. Pas besoin de les taper dans l'éditeur puisqu'ils sont incomplets.

L'alternative "Si..." ci-dessous est vérifiée : **vrai** ({code:true})

```
1. dave = true
2. kat = true
```

-

```
3.
4. if dave and kat then
```

Ici, **l'alternative "Si..."** n'est pas vérifiée : **faux** ({code:false})

```
1. dave = false
2. kat = true
3.
4. if dave and kat then
```

Ici aussi, **l'alternative "Si..."** n'est pas vérifiée : **faux** ({code:false})

```
1. dave = false
2. kat = false
3.
4. if dave and kat then
```

Ou

L'alternative "Si..." ci-dessous est vérifiée : **vrai** ({code:true})

```
1. dave = true
2. kat = true
3.
4. if dave or kat then
```

Ici, **l'alternative "Si..."** est vérifiée : **vrai** ({code:true})

```
1. dave = false
2. kat = true
3.
4. if dave or kat then
```

Ici par contre, **l'alternative "Si..."** n'est pas vérifiée : **faux** ({code:false})

```
1. dave = false
2. kat = false
```

-

```
3.
4. if dave or kat then
```

Non

La population comporte des individus violents et non violents.

Ceux qui ont obtenu leur examen sont admissibles, les autres sont non admissibles.

Avant que cela ne devienne confus...

Non ({code:not}) fait exactement ce que vous imaginez. Il détecte les conditions **non** vérifiées ({code:not true}). Dans les expressions, nous pouvons utiliser le symbole {code:!} à la place de {code:not}, mais c'est parfois plus lisible d'employer le mot.

Employons **non** {code:not} dans notre programme avec Dave et Kat.

```
1. dave = false
2. kat = false
3.
4. if not dave == true or not kat == true then
```

Cette **alternative "Si..."** est vérifiée ({code:true}) parce que nous vérifions si **l'une ou l'autre** des **variables** {code:dave} ou {code:kat} n'est pas **vrai** ({code:not true}). Et puisque ce qui n'est pas vrai est faux...

Pour écrire la même condition en utilisant le symbole {code:!}, la syntaxe est un peu différente :

```
1. dave = false
2. kat = false
3.
4. if dave != true or kat != true then
```

Avez-vous vu comment nous avons placé le symbole {code:!} juste devant le signe {code:=} ? {code:!=} se lit "non égal à".

-

Nous pouvons également l'écrire sous une forme plus compacte !

```
dave = false
kat = false

if !dave or !kat then
```

Les deux notations sont valides, c'est à vous de choisir celle que vous préférez.

Utilisation de Et, Ou et Non

Parfait ! Voyons quelques exemples ou cela est vraiment utile dans le cadre d'un projet de jeu.

Lorsque nous voulons vérifier les boutons du Joy-Con, nous utilison des **alternatives "Si..."** afin de déterminer quels boutons sont pressés. Ici, **et** et **ou** peuvent être d'une grande aide.

Disons que nous souhaitons que quelque chose arrive **seulement si** nous nous déplaçons en pressant le bouton A.

```
loop
    clear()

    j = controls( 0 )

    if j.lx != 0 and j.a then
        print( "Hourra !" )
    endif

    update()
repeat
```

Comme vous pouvez le voir, nous nous limitons à une simple **boucle**. Nous effaçons l'écran dès le début, et actualisons l'écran en toute fin.

La ligne {code:4} utilise la **fonction** {code:controls()} qui indique l'état courant de tous les contrôles. Le résultat est stocké dans une **variable** appelée {code:j}.

-

Notre **alternative "Si..."** vérifie si le stick gauche du **Joy-Con** (lx) est poussée dans une direction {code:j.lx != 0}. Si la valeur du stick est **0**, il est totalement au repos (et pile au milieu) !

Nous vérifions aussi si le bouton A est pressé avec {code:joy.a = true}.

L'emploi du **et** {code:and} entre les deux conditions permet de s'assurer qu'elles sont toutes deux vérifiées afin de permettre à **l'alternative "Si..."** de devenir vrai ({code:true}).

Voyons un exemple avec **ou** ({code:or}).

```
1. loop
2.     clear()
3.
4.     joy = controls( 0 )
5.
6.     if joy.zl or joy.zr then
7.         print( "Hourra !" )
8.     endif
9.
10.    update()
11. repeat
```

Cette fois nous vérifions différents boutons. Nous vérifions maintenant si *l'une ou l'autre* des gâchettes {code:joy.zl} ou {code:joy.zr} est pressée.

De cette façon, nous pouvons produire la même action pour les gâchettes gauche et droite.

Terminons par un exemple avec **non**({code:not}).

Cette fois, nous allons afficher "Hourra !" seulement si l'une des gâchettes *n'est pas* pressée.

```
1. loop
2.     clear()
3.
4.     j = controls( 0 )
5.
6.     if !j.zl or !j.zr then
7.         print( "Hourra !" )
8.     endif
```

-

```
 9.
10.     update()
11. repeat
```

Nous avons seulement ajouté les symboles {code:!} à notre code et maintenant le résultat est totalement différent !

Nous vérifions maintenant si les gâchettes sont au repos : **fausses** ({code:false}). Lorsque c'est le cas "Hourra !" apparaît à l'écran mais dès que l'une ou l'autre est pressée (devient **vrai** ({code:true}), **l'alternative "Si..."** n'est plus vérifiée et nous ne voyons plus "Hourra !".

Récapitulatif

Et ({code:and}), **ou** ({code:or}) et **Non** ({code:not}) sont des **opérateurs**. Ils effectuent une opération.

Nous les utilisons principalement dans les **alternatives "Si..."** pour évaluer plusieurs conditions.

Ils sont particulièrement efficaces avec la **fonction** {code:controls(0)}.

Imaginez que nous voulions un personnage dans un jeu qui soit capable d'effectuer une attaque sautée. Nous aurons besoin d'un **et** ({code:and}) dans cette situation car nous devrons vérifier que le personnage saute et que le bouton d'attaque est pressé.

On se retrouve au prochain tutoriel !

Fonctions et mots-clés utilisés dans ce tutoriel

and, [clear()](../Command Reference/Screen Display/clear.md), [controls()](../Command Reference/Input/controls.md), else, endIf, if, loop, not, or, [print()](../Command Reference/Text Handling/print.md), repeat, then, [update()](../Command Reference/Screen Display/update.md)

-

Tutoriel 10 : Notions avancées sur l'emploi des Variables

Dans ce tutoriel nous allons aborder quelques propriétés avancées des **variables**, ce que des mots comme "Global" et "Local" signifie et un concept appelé **étendue**.

Jusqu'à présent nous savons que nous pouvons utiliser une **variable** pour stocker une information. Cela peut être n'importe quoi: nous pouvons stocker un nombre dans une **variable**, nous pouvons stocker un morceau de texte (une chaîne de caractères {code:string}), nous pouvons stocker la valeur renvoyée par une **fonction**, nous pouvons même stocker plusieurs informations dans un type de **variable** appelé **tableau** ({code:array}).

Ce sont toutes des informations merveilleuses. Cependant, il nous manque quelque chose d'important pour les programmes avancés.

Toute **variable** dans un programme est sujette à ce que l'on appelle une **étendue**. Il y a deux niveaux **d'étendue** : **global** et **local**. **L'étendue** d'une **variable** détermine quelle portion du programme peut l'utiliser.

Les **variables globales** peuvent être exploitées **n'importe où** dans le programme. Les **variables locales** peuvent seulement être utilisées par la portion du programme située dans la **même étendue**.

À première vue, cela peut paraître nébuleux...

Regardons un exemple :

```
 1. a = 10
 2.
 3. loop
 4.     clear()
 5.     print( a + modifier( a ) )
 6.     update()
 7. repeat
 8.
 9. function modifier( number )
10.     number += a
11. return number
```

Ici nous avons un programme très simple qui somme deux nombres.

-

La **variable** au début du programme {code:a} est **globale**. Une fois définie en début de programme, nous pouvons l'utiliser où bon nous semble. Dans cet exemple, cela est mis en évidence par l'utilisation de la **variable** {code:a} :

- dans la **boucle** ({code:loop}) principale et
- dans notre **fonction** (définie par l'utilisateur) appelée {code:modifier()}.

La seconde **variable** dans notre programme {code:number} est **locale** à la **fonction** {code:modifier()}. Parce qu'elle est définie dans cette **fonction**, elle **n'est exploitable qu'à l'intérieur** de cette-ci.

Les Types de Données

Lorsque nous stockons une information dans un **variable** avec **FUZE^4 Nintendo Switch**, la plupart du temps nous n'avons pas besoin de spécifier le type de cette information. Par exemple, imaginons que nous souhaitons stocker un nombre :

```
1. a = 10
```

C'est fait ! Rien de plus n'est requis. Imaginons que nous souhaitons stocker une portion de texte ({code:string}) :

```
1. a = "Dave"
```

Bien ! Et pour stocker un tableau dans une **variable** ?

```
1. a = [ 0, 1, 2, 3 ]
```

Aussi simple que ça ! Et pour une **structure** ?

```
1. a = [ .name = "Dave", .age = 27, .interests = "Music" ]
```

Je vous entends crier : "Et pour un **vecteur** ({code:vector}) ?!". Si vous ne savez pas ce qu'est un **vecteur** ({code:vector}), n'ayez pas peur ! Nous aborderons ce sujet dans les prochains tutoriels.

-

```
1. a = { 0, 0, 0, 0 }
```

Aussi simple qui cela peut être ! Nous n'avons pas besoin d'indiquer le **type** de l'information à stocker dans la **variable**.

Néanmoins dans **FUZE^4 Nintendo Switch**, il existe un cas particulier pour lequel nous **devons** spécifier le **type** des données.

Il existe une seconde façon de définir une structure, en créant ce que nous appellerons un "type de structure" :

```
1. struct type_de_personne
2.     string nom
3.     int    age
4.     array  interets[3]
5. endstruct
6.
7. type_de_person personnes[10]
```

Dans l'exemple ci-dessus nous définissons un **type de structure** appelé {code:type_de_personne} qui comporte trois propriétés (caractéristiques).

La ligne 7 crée un tableau de structures appelé {code:personnes}. Il a 10 éléments, et chaque élément contient une structure avec ses trois propriétés : une **variable** ({code:string}) appelée {code:nom}, une **variable** {code:int} appelée {code:age} et une **variable** {code:array} appelée {code:interets} avec trois éléments.

Lors de la définition d'une structure à l'aide d'un type de structure comme ci-dessus, nous **devons** indiquer le type de chaque propriété. Si vous ne le faites pas, **FUZE^4 Nintendo Switch** signalera une erreur.

Cette méthode pour créer les structures est très utile pour les programmes de grande taille où vous aurez besoin d'utiliser un même type de structure de multiples fois en différents endroits.

Fonctions et mots-clés utilisés dans ce tutoriel

array, [clear()](../Command Reference/Screen Display/clear.md), endStruct, function, int, loop, [print()](../Command Reference/Text Handling/print.md), repeat, return, string, struct, [update()](../Command Reference/Screen Display/update)

-

Tutoriel 11 : Chargement et Affichage d'une Image

Bonjour ! Content de vous revoir.

Dans ce tutoriel nous allons voir comment charger des images à partir du navigateur de média **FUZE^4 Nintendo Switch** pour les afficher à l'écran à l'aide des **fonctions** internes.

Nous allons également nous intéresser à ce qui peut être fait pour redimensionner, déformer et pivoter une image pour la personnaliser !

Pour utiliser une image présente dans le navigateur de média **FUZE^4 Nintendo Switch**, nous devons tout d'abord commencer à **charger** l'image et à l'assigner à une **variable**.

Pour cela, nous utilisons la **fonction** {code:loadImage()} :

```
1. img = loadImage()
```

Dans **FUZE^4 Nintendo Switch**, lorsque vous tapez la **fonction** {code:loadImage()} et placez le curseur entre les parenthèses, vous remarquerez un contour brillant autour de la touche "média" du clavier à l'écran. Cliquez sur celle-ci pour accéder au navigateur de média.

Dans le navigateur de média, vous pouvez sélectionner l'image (asset) que vous souhaitez utiliser et FUZE vous permettra de coller le nom du fichier entre guillemets dans votre code.

Pour l'instant, nous avons choisi une image de l'artiste Selavi.

```
1. img = loadImage( "Selavi Games/JapaneseSetting", false )
```

-

Notez la présence de {code: false} parmi les arguments de la **fonction** {code:loadImage()}. Cet argument indique à FUZE s'il faut (ou non) appliquer un filtre à l'image. Nous ne souhaitons pas appliquer de filtre, d'où la présence de {code:false}.

Bon, maintenant nous avons enregistré le fichier image dans une **variable** appelée {code:img} (image en abrégé !). Nous pouvons maintenant utiliser cette **variable** dans beaucoup d'autres **fonctions**.

Commençons par dessiner simplement notre image à l'écran. Nous aurons besoin de la boucle standard, que nous connaissons et aimons tous, avec {code:clear ()} et {code:update()} !

```
1. img = loadImage( "Selavi Games/JapaneseSetting", false )
2.
3. loop
4.     clear()
5.
6.     update()
7. repeat
```

Tout ce dont nous avons besoin maintenant, c'est d'ajouter la **fonction** {code:drawImage()} !

```
1. img = loadImage( "Selavi Games/JapaneseSetting", false )
2.
3. loop
4.     clear()
5.
6.     drawImage( img, 0, 0, 1 )
7.
8.     update()
9. repeat
```

Lancez l'exécution du programme pour voir l'image à l'écran. Charmant !

drawImage()

Intéressons-nous d'avantage à cette **fonction** et voyons exactement ce qui se passe.

-

Le premier argument entre parenthèses est la **variable** dans laquelle nous avons stocké l'image. Facile !

Les deux arguments suivants sont les positions d'écran **x** et **y** auxquelles vous souhaitez dessiner l'image. {code: 0, 0} est le coin supérieur gauche de l'écran.

Le dernier argument est le **multiplicateur d'échelle**. Ce nombre est utilisé pour multiplier les dimensions de l'image lorsque nous la dessinons à l'écran.

Ici, nous avons utilisé {code:1} pour afficher l'image à sa taille réelle. Réduisons cette valeur :

```
1. img = loadImage( "Selavi Games/JapaneseSetting", false )
2.
3. loop
4.     clear()
5.
6.     drawImage( img, 0, 0, 0.1 )
7.
8.     update()
9. repeat
```

Avec une échelle de {code:0.1}, l'image est dix fois plus petite.

Expérimentons des manipulations plus sophistiquées de l'image avec la **fonction** {code:drawImageEx()}.

drawImageEx

En utilisant {code:drawImageEx} à la place de la **fonction** {code:drawImage}, nous avons accès à un contrôle plus poussé sur l'image. Quelques arguments supplémentaires sont nécessaires pour cette **fonction**, regardez ci-dessous :

```
1. img = loadImage( "Selavi Games/JapaneseSetting", false )
2. xPos = gwidth() / 2
3. yPos = gheight() / 2
4. rotation = 0
5. lEchelle = 0.7
6. hEchelle = 0.7
7. r = 1
8. g = 1
```

-

```
 9. b = 1
10. a = 1
11. origineX = 0
12. origineY = 0
13.
14. loop
15.     clear()
16.
17.     drawImageEx( img, xPos, yPos, rotation, lEchelle,
hEchelle, r, g, b, a, origineX, origineY )
18.
19.     update()
20. repeat
```

Waouh ! Regardez tous ces arguments! En utilisant {code:drawImageEx()}, nous pouvons contrôler presque tout ce qui concerne l'image.

Pour expliciter l'usage de la **fonction**, nous avons utilisé des **variables** en en-tête de la boucle pour mettre en évidence le rôle de chaque argument. Les **variables** sont présentées dans le même ordre que les arguments de la **fonction**.

Nous connaissons déjà les trois premiers arguments. {code:img} est la **variable** qui stocke le fichier image. Les deux arguments suivants, {code:xPos} et {code:yPos}, sont les coordonnées **x** et **y** de l'emplacement à partir duquel nous souhaitons que l'image soit dessinée.

Rotation

Le prochain argument est celui où les choses deviennent un peu plus intéressantes ! Cet argument contrôle l'angle de rotation de l'image. Pour le moment, la valeur de la **variable** {code:rotation} est 0. Si nous remplaçons cette valeur par {code:180}, l'image pivote et se retrouve à l'envers (la tête en bas).

Échelle

Les deux arguments suivants sont l'échelle en largeur et en hauteur. Lorsque ces deux arguments prennent 1 pour valeur, l'image est dessinée à son échelle normale. Changer ces nombres provoque un l'effet d'étirement de l'image. Par exemple, si vous passez la **variable** {code:hEchelle} de 1 à 0.5, l'image sera deux fois plus large que haute.

-

RGBA

Les prochains arguments sont vraiment très cool ! Ici, nous avons des valeurs séparées pour les éléments rouge, bleu, vert et alpha de l'image. Essayez de changer les **variables** {code:r}, {code:g} et {code:b} en {code:1}, {code:0} et {code:0} pour obtenir une image teintée de rouge. Essayez toutes les valeurs de votre choix entre {code: 0} et {code: 1} pour créer votre propre teinte !

Origine

Celui-ci est un peu étrange. Le point d'origine d'une image est l'endroit **à partir duquel** l'image est dessinée. Par défaut, l'origine est définie à 0 sur les axes **x** et **y**. N'oubliez pas que lorsque nous parlons de **l'écran**, (0, 0) fait référence au coin supérieur gauche. Cependant, en ce qui concerne les images (0,0) est le **milieu** de l'image. Regardez l'image ci-dessous :

Avec une position d'origine en {code:(0, 0)} et une position **x** et **y** de {code:(gwidth() / 2, gheight() / 2)}, notre image est affichée pile au milieu de l'écran.

Que se passe-t-il si nous modifions l'origine tout en conservant les mêmes positions **x** et **y** ?

-

Si nous définissons l'origine **x** sur {code:-imageW(img)/2} (moins une fois la moitié de la largeur de l'image), tout en conservant les emplacements **x** et **y** de l'image à l'identique, notre origine est sur le côté le plus à gauche de l'image. L'image est dessinée à partir de ce point sur l'écran.

Manipuler une Image dans un Programme

Modifions un peu le code pour tirer parti de toutes ces fonctionnalités. Tout d'abord, nous allons faire pivoter l'image pendant la boucle :

```
 1. img = loadImage( "Selavi Games/JapaneseSetting", false )
 2. xPos = gwidth() / 2
 3. yPos = gheight() / 2
 4. rotation = 0
 5. lEchelle = 0.7
 6. hEchelle = 0.7
 7. r = 1
 8. g = 1
 9. b = 1
10. a = 1
11. origineX = 0
12. origineY = 0
13.
14. loop
15.     clear()
16.
17.     drawImageEx( img, xPos, yPos, rotation, lEchelle,
hEchelle, r, g, b, a, origineX, origineY )
18.
19.     rotation += 1
20.
```

-

```
    update()
repeat
```

Exécutez le programme pour voir l'image tourner à l'écran.

Notez que l'image tourne **autour de l'origine**. Si nous devions changer l'origine, nous changerions également le point de rotation. Essayons cela :

```
img = loadImage( "Selavi Games/JapaneseSetting", false )
xPos = gwidth() / 2
yPos = gheight() / 2
rotation = 0
lEchelle = 0.7
hEchelle = 0.7
r = 1
g = 1
b = 1
a = 1
origineX = 800
origineY = 300

loop
    clear()

    drawImageEx( img, xPos, yPos, rotation, lEchelle, hEchelle, r, g, b, a, origineX, origineY )

    rotation += 1

    update()
repeat
```

Tout ce que nous avons fait est de changer la valeur des **variables** {code: origineX} et {code: origineY} en leur affectant respectivement les valeurs 800 et 300. Exécutez le programme pour constater que la rotation est maintenant très différente !

Bien. Ça suffit pour les rotations. En manipulant les valeurs rouge, verte, bleue et alpha, nous pouvons contrôler l'éclairage de l'image de manière très convaincante :

```
img = loadImage( "Selavi Games/JapaneseSetting", false )
xPos = gwidth() / 2
yPos = gheight() / 2
```

-

```
 4. rotation = 0
 5. lEchelle = 0.7
 6. hEchell = 0.7
 7. r = 1
 8. g = 1
 9. b = 1
10. a = 1
11. origineX = 0
12. origineY = 0
13.
14. loop
15.     clear()
16.     j = controls( 0 )
17.
18.     a += j.ly / 50
19.     g += j.ly / 50
20.     r += j.ly / 50
21.
22.     if r < 0 then r = 0 endif
23.     if g < 0 then g = 0 endif
18.
19.     drawImageEx( img, xPos, yPos, rotation, lEchelle,
hEchelle, r, g, b, a, origineX, origineY )
20.
21.     update()
22. repeat
```

Exécutez le programme et déplacez le Stick gauche vers le haut et le bas pour contrôler l'éclairage. Déplacer le Stick vers le bas rend l'image plus sombre et plus bleue, tandis que le déplacer vers le haut rend l'image plus lumineuse. Nous avons également ramené notre origine en (0, 0) dans cet exemple.

Les nouvelles lignes ajoutées vont de 16 à 23. Premièrement, nous appelons la **fonction** {code:controls()} pour accéder aux contrôleurs Joy-Con. Nous utilisons une **variable** appelée {code:j} pour stocker l'état des contrôles.

Aux lignes 18 à 20, nous utilisons le Stick gauche pour changer les valeurs des **variables** {code:a}, {code:g} et {code:r}. La **variable** {code: a} stocke la valeur **alpha** (la transparence de l'image). Si nous **réduisons** cette valeur, nous assombrissons l'image. Si nous **l'augmentons**, elle s'éclaircit. Comme la valeur du contrôle peut être positive ou négative, il suffit de l'ajouter à la **variable**.

Nous faisons la même chose avec les **variables** {code:g} et {code:r} pour contrôler la quantité de vert et de rouge de notre image. Lorsque nous déplaçons la manette de commande vers le bas, le rouge et le vert quittent l'image, nous laissant juste le

-

bleu. Cela se produit alors que la transparence augmente, nous permettant de voir plus d'écran noir derrière l'image. Cela nous donne un bel effet bleu foncé.

Ce type de technique est parfait pour un jeu avec beaucoup de dialogues entre les personnages.

Pourquoi ne pas essayer de changer l'image de fond et de créer différents effets d'éclairage ?

Rendez-vous dans le prochain tutoriel !

Fonctions et mots-clés utilisés dans ce tutoriel

[clear()](../Command Reference/Screen Display/clear.md), drawImage() (../Command Reference/2D Graphics/drawImage.md), [drawImageEx()] (../Command Reference/2D Graphics/drawImageEx.md), else, endIf, [gWidth()] (../Command Reference/Screen Display/gWidth.md), [gHeight()](../Command Reference/Screen Display/gHeight.md), if, [loadImage()](../Command Reference/2D Graphics/loadImage.md), loop, repeat, then, [update()](../Command Reference/Screen Display/update.md)

Tutoriel 12 : Structures

Bienvenue ! Activez toutes vos neurones pour ce tutoriel - nous allons examiner un concept légèrement plus avancé.

Ce projet introduit les **structures** : ce qu'est une structure, comment nous pouvons en utiliser dans nos programmes. Et par ailleurs, présente quelques exemples.

Les **structures** sont des outils incroyablement efficaces pour la programmation et rendent le code bien plus lisible.

Qu'est qu'une Structure ?

-

Une **structure** ressemble à un tableau. Nous l'utilisons pour stocker des informations.

Là où un tableau utilise un *nombre* pour accéder à un élément d'information, une **structure** utilise un *nom*.

Nous pouvons en utiliser une pour stocker toutes les informations du personnage d'un joueur dans un jeu à défilement horizontal. Le personnage du joueur dispose d'une position à l'écran, d'une valeur de santé, d'une vitesse, de valeurs d'attaque et de défense, etc. Toutes ces informations peuvent être stockées dans une **structure**, ce qui en rend l'accès très facile et pratique.

Revenons à notre projet et créons une **structure** très simple qui stocke des informations sur une personne. Copiez le code ci-dessous dans l'éditeur de code **FUZE^ 4 Nintendo Switch**. Bien sûr, n'hésitez pas à modifier les informations !

```
1. personne = [
2.     .nom = "Luke",
3.     .aime = "Chocolat",
4.     .deteste = "Legumes",
5.     .aptitudes = "Programmation"
6. ]
```

Là, nous l'avons. Ce code prépare une structure que nous pouvons utiliser. Nous pouvons accéder aux informations avec quelque chose comme : {code:print(personne.nom)} ou {code:print(personne.deteste)}. C'est assez simple !

Mise en page

Passons rapidement à l'étrange façon dont ce code est présenté. Pour un débuttant en programmation, cela peut paraître assez étrange.

L'essentiel réside en l'usage des crochets et des virgules. Premièrement, nous nommons la **structure**. Nous l'avons appelé {code:personne}. Ensuite, nous ouvrons les crochets pour décrire la **structure** et définir ses propriétés. Chacune de ces propriétés doit être séparée de la suivante par une virgule. Enfin, nous fermons les crochets pour terminer la description de la **structure**.

Il est très important de réaliser que ce *n'est pas* la seule façon de mettre en forme une **structure**. Vous voudrez peut-être la présenter ainsi :

-

```
personne = [.nom = "Luke",
            .aime = "Chocolat",
            .deteste = "Legumes",
            .aptitudes = "Programmation"]
```

Ce qui fonctionne *exactement de même la façon*. Vous pouvez même tout écrire sur une seule ligne, comme ceci :

```
personne = [.nom = "Luke", .aime = "Chocol", .deteste =
"Legumes", .aptitudes = "Programmation"]
```

Comme mentionné précédemment, les parties importantes sont les crochets et les virgules. C'est à vous de décider comment vous souhaitez structurer votre code dans **FUZE^ 4 Nintendo Switch**.

Créer un Tableau de Structures

C'est bien beau, mais que se passe-t-il si vous souhaitez stocker les informations de plusieurs personnes ?

Ici, nous pouvons utiliser un tableau de structures. Il s'agit simplement d'un tableau, où chaque élément est une structure. Jetez un oeil ci-dessous, nous avons ajouté une nouvelle personne au programme :

```
personnes = [
    [
        .nom = "Luke",
        .aime = "Chocolate",
        .deteste = "Legumes",
        .aptitudes = "Programmation"
    ],
    [
        .nom = "Colin",
        .aime = "Trains",
        .deteste = "Choux de Bruxelles",
        .aptitudes = "Dance"
    ]
]
```

-

Maintenant, le code est un peu plus compliqué. Essayez de ne pas être rebutés par les crochets !

Notre première ligne ne crée plus une structure appelée {code:personne}. Maintenant, elle crée un tableau appelé {code:personnes} avec deux éléments. Chacun de ces éléments est une structure, tout comme la structure unique précédente.

La première structure est stockée dans {code:personnes[0]} et la seconde dans {code:personnes[1]}.

Vous voulez afficher les goûts de Colin ? Nous pouvons le faire avec {code:print(personnes[1].aime)}. Vous voulez afficher le nom de la première personne (Luke) ? Nous pouvons le faire avec {code:print(personnes[0].nom)}.

L'utilisation d'ensemble de structures fournit une méthode pratique et puissante pour stocker des informations exploitables dans nos programmes. Avec une combinaison de **structures**, de **tableaux** et de **boucles "Pour..."**, vous pouvez réaliser des choses absolument incroyables.

Emploi d'une Structre dans un Jeu simple

Pour la partie principale de ce tutoriel, nous utiliserons un **tableau** de **structures** pour créer un jeu de course simple, dans lequel 3 formes traversent l'écran. À vous de deviner le gagnant !

Nous allons utiliser trois formes, un triangle, un cercle et un pentagone. Chacune a besoin d'un nom, d'une position **x** et **y**, d'un certain nombre de côtés (voir plus bas), d'une couleur et d'une vitesse de déplacement à l'écran.

Notre vitesse sera un nombre choisi au hasard sur 50. Ainsi, chaque fois que nous exécutons le programme, nous obtiendrons un résultat différent !

Construisons d'abord la **structure**. Elle comporte beaucoup de code, il est donc recommandé de *copier/coller* celle-ci dans l'éditeur de code **FUZE^ 4 Nintendo Switch**.

```
1. formes = [
2.     [
3.         .nom     = "Triangle",
4.         .x       = 0,
```

-

```
 5.          .y       = gheight() / 2 - gheight() / 3,
 6.          .cotes   = 3,
 7.          .couleur = red,
 8.          .vitesse = random( 50 )
 9.      ],
10.      [
11.          .nom     = "Cercle",
12.          .x       = 0,
13.          .y       = gheight() / 2,
14.          .cotes   = 32,
15.          .couleur = green,
16.          .vitesse = random( 50 )
17.      ],
18.      [
19.          .nom     = "Pentagone",
20.          .x       = 0,
21.          .y       = gheight() / 2 + gheight() / 3,
22.          .cotes   = 5,
23.          .couleur = blue,
24.          .vitesse = random( 50 )
25.      ]
26. ]
```

Pfiou... Nous y voilà. Regardez bien ce **tableau** de **structures**. Nous avons créé un **tableau** appelé {code:formes} avec 3 éléments. Chacun de ces éléments est une **structure** avec 6 **propriétés**.

Comme précédemment dans l'exemple avec le **tableau** (de **structures**) personnes, nous pouvons accéder à n'importe laquelle d'entre elles avec une instruction du type : {code:print(formes[1].nom)}.

Ajoutons maintenant le code qui utilise cette information. Nous avons besoin d'une boucle pour animer l'écran et d'une boucle "Pour..." pour parcourir notre tableau de formes.

```
28. loop
29.     clear()
30.     for i = 0 to len( formes ) loop
31.         circle( formes[i].x, formes[i].y, 100,
formes[i].cotes, formes[i].couleur, false )
32.     repeat
33.     update()
34. repeat
```

-

Voici un exemple qui met vraiment en évidence l'utilité des **tableaux** de **structures**. Les lignes 30 à 32 constituent une **boucle "Pour..."** qui se répète 3 fois. Nous créons une **variable** appelée {code:i} qui passe successivement de 0, à 1 et finalement à 2.

Notez que nous utilisons la **fonction** {code:len()} pour obtenir la taille du tableau. Notre tableau étant composé de 3 éléments, la valeur renvoyée est 3.

Cette **variable** {code: i} est utilisée comme index dans le **tableau** {code: formes} pour tracer un "cercle" à l'écran pour chaque forme, aux coordonnées **x** et **y** stockées dans la **structure**. En raison de la manière dont la **fonction** {code:circle()} fonctionne dans **FUZE^ 4 Nintendo Switch**, nous pouvons simplement modifier le nombre de côtés et créer une forme différente ! C'est pourquoi nous stockons le nombre de côtés en tant que propriété pour chaque **structure**.

Une fois que la **boucle "Pour..."** terminée, nous actualisons l'écran avec {code:update()} et répétons la **boucle** principale.

Maintenant, déplaçons les formes à travers l'écran !

En ajoutant quelques lignes de code dans la **boucle "Pour..."**, nous pouvons déplacer chaque forme :

```
 28. loop
 29.     clear()
 30.     for i = 0 to len( formes ) loop
 31.         formes[i].x += formes[i].vitesse
 32.         circle( formes[i].x, formes[i].y, 100,
formes[i].cotes, formes[i].couleur, false )
 33.     repeat
 34.     update()
 35. repeat
```

La ligne 31 incrémente la position **x** de charque forme en accord avec sa vitesse. Lancez l'exécution du programme pour voir toutes les formes se déplacer à différentes vitesses !

Tout ce qui nous reste à faire est de créer un écran de victoire lorsqu'une forme gagne la course !

-

C’est le moment idéal pour créer notre propre **fonction** ! Nous pouvons passer le nom et la couleur de la forme gagnante à notre **fonction** et utiliser ces informations sur l’écran de victoire. Entrez le code ci-après à la *fin* du programme, après le {code:repeat} de la ligne 35.

```
 37. function ecranVictoire( nom, couleur )
 38.     texte = nom + " gagne !"
 39.     tsize = 100
 40.     textSize( tSize )
 41.     loop
 42.         clear()
 43.         drawText( gwidth() / 2 - textWidth( texte ) / 2,
gheight() / 2, tSize, couleur, texte )
 44.         update()
 45.     repeat
 46. return void
```

Ici nous avons défini une **fonction** utilisateur. C’est cette portion de code qui s’exécutera lorsque nous *appelons* la fonction.

Le but de cette **fonction** est d’afficher le nom de la forme gagnante dans la bonne couleur à l’écran. Pour cela, nous avons besoin de deux informations : le nom et la couleur de la forme gagnante !

A la première ligne, nous nommons la **fonction** {code:ecranVictoire()} et la dotons de deux **arguments**. Le premier argument est stocké dans une **variable** nommée {code:nom} et le second est stocké dans une variable appelée {code:couleur}. Ainsi, nous pouvons utiliser ces **variables** dans notre **fonction**.

Nous commençons par définir une nouvelle **variable** appelée {code:texte}. Elle stocke l’intégralité du texte que nous souhaitons afficher. Nous prenons la valeur de la **variable** {code:nom} issues des arguments de la **fonction** et lui ajoutons le texte " gagne !". Cette **variable** {code:texte} sera utilisée lorsqu’il s’agira de faire apparaître le texte à l’écran.

Ensuite, nous définission une **variable** pour stocker la taille du texte. Procéder ainsi nous permet de centrer avec précision le texte à l’écran.

Sur la ligne 40, nous utilisons la **fonction** {code:textSize()} pour définir la taille du texte en accord avec le contenu de la **variable** {code:tSize}.

-

Passons maintenant à la partie principale de notre écran de victoire : la **boucle**. Nous avons besoin d'une **boucle** car nous voulons que notre écran affiche le texte jusqu'à ce que le programme soit arrêté. Bien sûr, nous aurons besoin d'un {code:clear()} et d'un {code:update()} car nous voulons afficher quelque chose à l'écran.

L'instruction principale de notre **boucle** est la **fonction** {code:drawText()} que nous utilisons pour afficher le texte à l'écran d'une façon plus détaillée que ne le permettrait un simple {code:print()}. Nous pouvons positionner le texte au pixel près, définir sa taille et même sa couleur.

Comme vous pouvez le voir, la positon **x** de la **fonction** {code:drawText()} est un peu étrange :

```
43.         drawText( gwidth() / 2 - textWidth( texte ) / 2
```

Nous souhaitons que le texte soit exactement centré à l'écran, sans avoir à nous soucier de sa taille. Si nous positionnons notre texte à des coordonnées fixes, nous obtiendrons des résultats différents en fonction du mode de la console (TV ou Portable). Nous obtiendrons également des résultats changeants selon la forme gagnante car la longueur du message de victoire varie selon son nom.

À cause de toutes ces raisons, nous devons être en mesure d'ajuster la position en fonction de la taille réelle de notre texte. C'est là qu'interveint notre **variable** {code:texte} !

Nous utilisons la **fonction** {code:textWidth()} pour obtenir la largeur de la représentation du texte en pixels. Nous pouvons alors la diviser par deux pour obtenir la moitié de sa largeur. Si nous la soustrayons à la position centrale de l'écran ({code:gwidth() / 2}), nous centrerons toujours parfaitement notre texte à l'écran !

Cela semble un peu fastideux, mais c'est une technique très utile pour positionner du texte ! Assurez-vous de vous en souvenir pour vos prochains projets !

Bien. Notre **fonction** utilisateur est complète. Nous terminons la **fonction** par {code:return void} car nous n'avons pas besoin de renvoyer quoi que ce soit.

Il ne reste plus qu'à *appeler* la **fonction** dans la boucle de notre programme principale !

-

Modifiez le code de la **boucle** principale pour qu'elle ressemble à celle ci-dessous. Nous avons simplement ajouté une **alternative "Si..."** pour vérifier si une forme a atteint la ligne d'arrivée.

```
 28. loop
 29.     clear()
 30.     for i = 0 to len( formes ) loop
 31.         formes[i].x += formes[i].vitesse
 32.         if formes[i].x > gwidth() then
 33.             ecranVictoire( formes[i].name, formes[i].couleur
)
 34.         endif
 35.         circle( formes[i].x, formes[i].y, 100,
formes[i].sides, formes[i].couleur, false )
 36.     repeat
 37.     update()
 38. repeat
```

Les lignes 32 à 34 comporte notre **alternatice "Si..."**. Nous vérifions simplement si la forme courante dans la **boucle "Pour..."** a atteint l'extrémité de l'écran. Si c'est le cas, nous *appelons* la **fonction** {code:ecranVictoire()} en lui *passant* le nom et la couleur de la forme comme **arguments**.

C'est terminé ! Amusez vous avec ce programme ! Peut-être pouvez-vous ajouter quelques formes à la course ? De la façon dont notre programme est écrit, la seule chose à faire est d'ajouter les nouvelles formes au **tableau** !

Résumé

Une **structure** est un outil pour stocker des données. Elle est très similaire à un **tableau** à part que chaque élement a un *nom* plutôt qu'un *numéro*. Vous pouvez mélanger et assortir ces techniques pour la tâche à accomplir. Vous pouvez stocker vos **tableaux** dans des **structures**, et stocker vos **structures** dans **tableaux** !

Essayez de préparer vos propres structures et d'accéder à leurs valeurs pour pleinement comprendre comment elles fonctionnent !

Comme toujours, félicitation pour être parvenu jusqu'ici et à bientôt dans le prochain tutoriel !

Fonctions et mots-clés utilisés dans ce tutoriel

-

else, endIf, [circle()](../Command Reference/2D Graphics/circle.md), [clear()](../Command Reference/Screen Display/clear.md), [drawText()](../Command Reference/Text Handling/drawText.md), for, if, loop, repeat, [textSize()](../Command Reference/Text Handling/textSize.md), then, [update()](../Command Reference/Screen Display/update.md)

Tutoriel 13 : Jouons de la Musique

Vous avez l'oreille musicale ? Non ? Et bien, vous l'aurez bientôt ! Dans ce tutoriel, nous utiliserons la **fonction** {code:playNote()} pour créer notre propre musique.

Nous avons déjà examiné très brièvement {code:playNote()}, lors du tutoriel sur **les boucles "Pour..."**. Dans ce projet, nous entrerons beaucoup plus dans les détails.

playNote()

Jetons un coup d'œil à la **fonction** et parcourons ses arguments :

```
playNote( canal, frequence, type_d_onde, volume, vitesse,
panoramique )
```

L'argument {code:canal} indique à FUZE le canal audio sur lequel jouer la note souhaitée. Il y a 16 canaux disponibles, ce qui permet de jouer jusqu'à 16 sons en même temps, qu'il s'agisse de notes simples, de pistes musicales ou d'effets sonores.

L'argument {code:frequence} est la **fréquence** de la note à jouer. Un autre mot pour la fréquence est le **ton** (pitch). Plus la fréquence est élevée, plus le ton de la note est élevé.

La fréquence est mesurée en Hertz (hz). Cela signifie **cycles par seconde**. Si votre tympan vibre à 440hz (440 fois par seconde), vous entendrez la note **La** ! Les humains peuvent entendre des sons d'environ 20hz à 20 000hz. Tout ce qui est en dehors de cette plage n'est pas audible pour nous.

-

L’argument {code:type_d_onde} est le **type de forme d’onde** que nous voulons utiliser pour jouer la note. FUZE en propose 5 types différents, chacun portant un numéro: carré (0), dent de scie (1), triangle (2), sinus (3) et bruit (4). Chacune de ces formes d’onde a un son très différent pour nos oreilles. Testez-les tous !

L’argument {code:volume} est simplement le volume de la note. Cette valeur doit être comprise entre 0 et 1, mais peut être augmentée si souhaité.

L’argument {code:vitesse} décrit la forme d’enveloppe de la note. Celui-ci est un peu difficile à imaginer. Une valeur peu élevée pour cet argument {code:vitesse} entraîne une durée de note plus longue. Un valeur plus élevée entraîne une durée de note plus courte.

Enfin, l’argument {code:panoramique} est la position stéréo du son. Votre console Nintendo Switch a deux canaux de haut-parleur : un gauche et un droit. Ce nombre est la position du son entre ces canaux. La valeur de {code: 0.5} correspond au centre. Un nombre plus proche de 0 entraîne un déplacement du son vers la gauche, tandis qu’un nombre plus proche de 1 entraîne un déplacement du son vers la droite.

Un exemple rapide

Plaçons quelques valeurs dans cette **fonction** pour créer un son :

```
1. playNote( 0, 3, 432, 1, 0.5, 0.5 )
2. loop
3.     clear()
4.     update()
5. repeat
```

Nous avons ici un petit programme qui joue une note puis entre dans une boucle. Notez que la ligne {code:playNote()} n’est pas **dans la boucle** pour ne l’exécuter qu’une seule fois. Notre note est jouée pendant quelques secondes avant de disparaître.

Essayez de changer ces valeurs pour obtenir différents sons.

note2Freq()

-

Dans FUZE, nous avons une **fonction** très pratique appelée {code:note2Freq()}. Lorsque nous faisons de la musique, nous n'avons pas tendance à penser en termes de **fréquence**, mais plutôt en termes de notes de solfége. Par exemple, la note *La* au-dessus du *Do moyen* (également appelé La3) se trouve à la fréquence de 440hz. Pour dire à un musicien comment jouer une mélodie, nous ne lui listons pas des fréquences !

Il y a quelque temps, Dave Smith, un type très intelligent, a inventé un procédé pour envoyer des données à des instruments électroniques pour leur indiquer les notes à jouer. Ce système s'appelle MIDI (Musical Instrument Digital Interface). En MIDI, vous avez le choix entre 128 notes, chacune avec un numéro. La note La3 mentionnée précédemment est associée au numéro 69.

{code:note2Freq()} **reçoit** un numéro de note MIDI et le convertit en sa **fréquence**. Cela s'avère vraiment très utiles lors de la création de musique !

Créer un projet vide avant de passer à la partie suivante !

Créer un tableau de notes

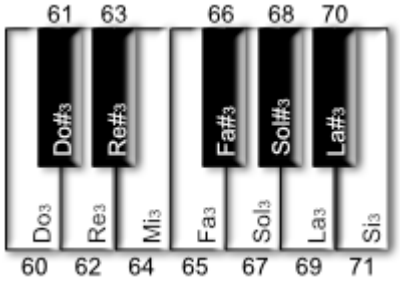
Si nous voulons utiliser les noms des notes pour écrire de la musique, nous devons stocker les numéros des notes MIDI correspondantes en mémoire. Nous pouvons utiliser une structure pour cela, en donnant à chaque note son numéro :

```
1. n = [
2.     .do   = 60,
3.     .dod  = 61,
4.     .re   = 62,
5.     .red  = 63,
6.     .mi   = 64,
7.     .fa   = 65,
8.     .fad  = 66,
9.     .sol  = 67,
10.    .sold = 68,
11.    .la   = 69,
12.    .lad  = 70,
13.    .si   = 71
14. ]
```

Nous y voilà. Maintenant la structure {code:n} (pour **note**) comporte l'ensemble des notes que nous souhaitons jouer. Par exemple, nous pouvons utiliser {code:n.re} pour obtenir la note Ré.

-

Les notes suivies d'une lettre "d" sont des notes intermédiaire (demi-ton) appelées "dièse". Dans la séquence Do, Ré, Mi, Fa, Sol, La, Si, les notes Do, Ré, Fa, Sol, La sont séparées de la note suivante par un ton. Les notes Si et Mi sont séparées de la note suivante par un demi-ton. C'est la raison pour laquelle sur le clavier d'un piano, une touche noire est disposée après Do, Ré, Sol, La afin que l'écart entre chaque touche soit d'un demi-ton.



Nous avons donc maintenant une échelle de 12 notes avec toutes les notes intermédiaires. C'est ce qu'on appelle l'échelle **chromatique** grâce à laquelle nous pouvons composer à peu près tout ce que nous aimons !

Avant de composer une mélodie, nous aurons besoin de deux autres **variables**. Les notes ont non seulement une hauteur, mais aussi une longueur. En musique, il existe des noms spécifiques pour la longueur des notes. Nous en utiliserons 4 : **double croche**, **croche**, **noire** et **blanche** :

```
16. double_croche = 0.25
17. croche = 0.5
18. noire = 1
19. blanche = 2
```

Nous créons ici quelques **variables** pour stocker les valeurs des longueurs de chaque type de note à utiliser. Celles-ci sont relatives à un "temps" : une noire correspond à un temps, une croche est la moitié d'une noire (0,5), et une double croche est la moitié d'une croche (0,25).

Utilisons maintenant ces variables pour composer une petite mélodie, en stockant les données pour chaque note à jouer dans un **tableau** :

```
21. melodie = [
22.     [ .note = n.do,  .vitesse = 40, .l = croche      ],
23.     [ .note = n.mi,  .vitesse = 40, .l = croche      ],
24.     [ .note = n.fa,  .vitesse = 40, .l = croche      ],
25.     [ .note = n.sol, .vitesse = 40, .l = croche      ],
```

-

```
26.     [ .note = n.mi,  .vitesse = 20, .l = noire       ],
27.     [ .note = n.do,  .vitesse = 40, .l = croche      ],
28.     [ .note = n.re,  .vitesse = 20, .l = blanche * 4 ]
29. ]
```

Voici notre mélodie ! Comme vous pouvez le constater, nous créons un **tableau** appelé {code: melodie} qui stocke 7 **structures**. Chaque **structure** est une note de notre mélodie, nous donnant un total de 7 notes. Chaque note a une propriété {code:.note} qui stocke la valeur du code MIDI récupérée à partir de la **structure** des notes, une valeur {code: .vitesse} qui stockera la valeur de vitesse de la **fonction** {code:playNote()} et enfin une valeur {code: .l} qui stocke la longueur de la note.

En utilisant ce format, vous pouvez écrire un morceau de musique très facilement ! Même s'il faut taper un peu de code...

Avant de jouer du son, nous devons commencer par convertir ces valeurs de longueur de note en valeurs chronologique. La valeur de la longueur de la note indique la durée de la note, pas **quand** la note doit être jourée. Il existe un moyen astucieux de faire cette conversion en utilisant une **boucle "Pour..."** :

```
31. duree = 0
32. compteur = 0
33.
34. for i = 0 to len( melodie ) loop
35.     temp = melodie[i].l
36.     melodie[i].l = compteur
37.     compteur += temp
38. repeat
39.
40. duree = compteur
```

Nous créons d'abord deux **variables**. {code:duree} finit par stocker la durée totale de la mélodie. {code:compteur} est utilisé pour garder une trace du temps des notes pendant la **boucle "Pour..."**.

Notre **boucle "Pour..."** compte de 0 à la longueur du **tableau** {code:melodie} en utilisant la **variable** {code:i}. Pour chaque note de la mélodie, nous stockons la valeur de la longueur de cette note dans une **variable** appelée {code:temp}. Nous définissons ensuite la longueur de cette note pour qu'elle soit égale à la **variable**

-

{code:counter}. Puisque {code:compteur} commence à 0, la première valeur de longueur de note est 0. Parfait ! Nous voulons que notre première note débute immédiatement.

Nous augmentons ensuite la **variable** {code:compteur} du nombre contenu dans la **variable** {code:temp}, ce qui nous donne un nouveau point dans le temps où la prochaine note devrait commencer. Ce processus se produit pour chaque note de la mélodie, en convertissant la valeur de sa longueur en une position chronologique de départ.

Lorsque la **boucle "Pour..."** est terminée, notre **variable** {code:compteur} comporte la durée de la mélodie. Nous affectons donc la valeur de {code:compteur} à la **variable** {code:duree}.

Nous avons juste besoin de deux autres **variables** avant de commencer la **boucle** principale dans laquelle la mélodie sera jouée.

```
42. compteurDeNote = 0
43. chrono = 0
```

Ces deux **variables** sont importantes. {code:compteurDeNote} est utilisé comme l'index du **tableau** {code:melodie} pour jouer chaque note en séquence. {code:chrono} est une **variable** qui garde la trace du temps écoulé. Nous utiliserons la **variable** {code:chrono} comme déclencheur pour indiquer à FUZE quand passer à la note suivante (du **tableau** {code:melodie}).

Jouer la mélodie

Bon, mettons tout cela à profit :

```
45. loop
46.     clear()
47.
48.     if compteurDeNote < len( melodie ) then
49.         if chrono >= melodie[compteurDeNote].l then
50.             note = note2Freq( melodie[compteurDeNote].note )
51.             vitesse = melodie[compteurDeNote].vitesse
52.             playNote( 0, 3, note, 1, vitesse, 0.5 )
53.             compteurDeNote += 1
54.         endif
55.     endif
```

-

```
56.
57.     chrono += ( 120 / 60 ) / 60
58.
59.     if chrono >= duree then
60.         stopChannel( 0 )
61.     endif
62.
63.     update()
64. repeat
```

Une fois le code terminé, exécutez le programme pour entendre la mélodie dans toute sa splendeur. Jazzy !

Dans cette **boucle**, nous avons quelques astucieuses **alternatives "Si..."**. Avant de nous penchez sur celles-ci, jetez un coup d'œil à la ligne 57 :

```
57.     chrono += ( 120 / 60 ) / 60
```

Cette ligne de code est la partie responsable de la gestion du temps. La vitesse d'un morceau de musique s'appelle le tempo. Celui-ci est mesuré en battements par minute, ou BPM. Notre mélodie est jouée à 120 battements par minute.

Notre boucle effectue 60 cycles par seconde, avec la **fonction** {code:update()} qui s'exécute à chaque itération.

Pour obtenir le bon tempo, nous devons déterminer non pas le nombre de battements par minute, mais le nombre de battements **par image**.

Pour ce faire, nous prenons le nombre de battements par minute (120) et nous le divisons par 60 pour obtenir le nombre de battements par seconde ({code:120/60}). Nous divisons ensuite ce résultat par 60 images seconde pour obtenir des battements par image.

Tout cela signifie que notre chrono augmente maintenant à un taux qui nous donnera 120 battements par minute à 60 images par seconde. Intelligent !

Voyons maintenant les **alternatives "Si..."** :

```
48.     if compteurDeNote < len( melodie ) then
49.         if chrono >= melodie[compteurDeNote].l then
50.             note = note2Freq( melodie[compteurDeNote].note )
51.             vitesse = melodie[compteurDeNote].vitesse
```

-

```
52.             playNote( 0, 3, note, 1, vitesse, 0.5 )
53.             compteurDeNote += 1
54.         endif
55.     endif
```

Nous vérifions d'abord si la **variable** {code:compteurDeNote} est inférieure à la longueur de la mélodie. Ensuite, nous vérifions si la **variable** {code:chrono} a atteint la valeur de départ de la note actuelle de la mélodie.

Si ces deux conditions sont remplies, nous définissons quelques **variables** locales afin de rendre notre ligne {code:playNote()} beaucoup plus facile à lire. La **variable** {code:note} stocke la valeur de fréquence de la note, calculée à l'aide de la **fonction** {code:note2Freq()} dont nous avons parlé précédemment.

La **variable** {code:vitesse} stocke la valeur {code:.vitesse} du **tableau** {code:melodie}.

Nous utilisons ensuite la **fonction** {code:playNote()} pour jouer la note désirée en utilisant une belle onde sinusoïdale sur le canal 0 avec le volume maximum et une position stéréo centrale.

Avant de terminer **l'alternative "Si..."**, nous augmentons la valeur de {code:compteurDeNote} pour nous assurer de jouer la note suivante de la séquence.

Terminé !

Essayez de changer la mélodie, d'ajouter vos propres notes, de changer les timings et de changer les battements par minute. Aucun risque d'erreur ici ! C'est groovy !

Avant de commencer, appliquons un peu de **reverb** à cette mélodie pour la rendre plus douce :

```
44. setReverb( 0, 8000, 0.5 )
```

Juste avant le début de la **boucle**, ajoutez la ligne ci-dessus à votre programme. Cette **fonction** définit une quantité de réverbération, ou d'écho, sur un canal.

Le premier chiffre entre parenthèses est le numéro du canal (0). Comme notre mélodie est jouée sur le canal 0, il faut utiliser 0 pour entendre cet effet !

-

Le nombre suivant est la quantité de millisecondes avant que nous n'entendions l'écho.

Le dernier nombre est le multiplicateur appliqué au volume de l'écho au fil du temps. Un nombre faible entraîne une réduction rapide du volume, tandis qu'un nombre plus élevé entraîne une réduction de volume plus lente. Ce nombre peut être compris entre 0 et 1.

Maintenant, lancez le programme et nous écoutez la différence !

Allez-y et faites vos propres musiques !

Jouer votre mélodie en boucle

Le programme que nous avons écrit ne joue la mélodie qu'une fois, puis arrête tous les sons du canal :

```
59.     if chrono >= duree then
60.         stopChannel( 0 )
61.     endif
```

Avec cette **alternatice "Si..."**, nous vérifions si la **variable** {code:chrono} a atteint la valeur stockée dans la **variable** {code:duree}. Si tel est le cas, nous émettons une commande {code:stopChannel()} pour cesser de jouer du son sur ce canal. Si nous voulons que notre mélodie soit jouée en boucle, nous pouvons changer ceci comme suit :

```
59.     if chrono >= duree then
60.         compteurDeNote = 0
61.         chrono = 0
62.     endif
```

Maintenant, lorsque notre {code:chrono} atteint la {code:duree}, le {code:chrono} et le {code: compteurDeNote} sont réinitialisés à 0, ce qui permet de tout recommencer.

Pour référence, voici le projet complet ci-dessous, assurez-vous que le vôtre fonctionne correctement avant de l'utiliser dans un autre projet !

-

```
n = [
    .do   = 60,
    .dod  = 61,
    .re   = 62,
    .red  = 63,
    .mi   = 64,
    .fa   = 65,
    .fad  = 66,
    .sol  = 67,
    .sold = 68,
    .la   = 69,
    .lad  = 70,
    .si   = 71
]

double_croche = 0.25
croche = 0.5
noire = 1
blanche = 2

melodie = [
    [ .note = n.do,  .vitesse = 40, .l = croche      ],
    [ .note = n.mi,  .vitesse = 40, .l = croche      ],
    [ .note = n.fa,  .vitesse = 40, .l = croche      ],
    [ .note = n.sol, .vitesse = 40, .l = croche      ],
    [ .note = n.mi,  .vitesse = 20, .l = noire       ],
    [ .note = n.do,  .vitesse = 40, .l = croche      ],
    [ .note = n.re,  .vitesse = 20, .l = blanche * 4 ]
]

duree = 0
compteur = 0

for i = 0 to len( melodie ) loop
    temp = melodie[i].l
    melodie[i].l = compteur
    compteur += temp
repeat

duree = compteur

compteurDeNote = 0
chrono = 0
setReverb( 0, 8000, 0.5 )
loop
    clear()

    if compteurDeNote < len( melodie ) then
```

-

```
        if chrono >= melodie[compteurDeNote].l then
            note = note2Freq( melodie[compteurDeNote].note )
            vitesse = melodie[compteurDeNote].vitesse
            playNote( 0, 3, note, 1, vitesse, 0.5 )
            compteurDeNote += 1
        endif
    endif

    chrono += ( 120 / 60 ) / 60

    if chrono >= duree then
        stopChannel( 0 )
    endif

    update()
repeat
```

Jouer plusieurs mélodies simultanément

Ahah! Alors vous voulez ajouter des accords ?

Et bien, c'est possible. Comme nous l'avons mentionné précédemment, nous disposons de 16 canaux. Cela signifie que nous pouvons jouer 16 mélodies simultanément, par contre cela peut devenir assez difficile à écouter !

Commencez un nouveau fichier de projet avant de continuer. Nous allons avoir besoin des mêmes données de note que précédemment, alors copiez la section suivante dans votre nouveau projet :

```
n = [
    .do   = 60,
    .dod  = 61,
    .re   = 62,
    .red  = 63,
    .mi   = 64,
    .fa   = 65,
    .fad  = 66,
    .sol  = 67,
    .sold = 68,
    .la   = 69,
    .lad  = 70,
    .si   = 71
]

```

-

```
16. double_croche = 0.25
17. croche = 0.5
18. noire = 1
19. blanche = 2
```

Pour que deux mélodies soient jouées en même temps, nous devons simplement dupliquer de nos **variables**. La meilleure façon de faire est de convertir les **variables** comme {code:compteurDeNote} et {code:duree} en **tableaux**. Bien entendu, notre **tableau** {code:melodie} devra également être placé dans un **tableau**. Déroutant? Ne vous inquiètez pas :

```
21. melodie = [
22.     [
22.         [ .note = n.do,  .vitesse = 40, .l = croche      ],
23.         [ .note = n.mi,  .vitesse = 40, .l = croche      ],
24.         [ .note = n.fa,  .vitesse = 40, .l = croche      ],
25.         [ .note = n.sol, .vitesse = 40, .l = croche      ],
26.         [ .note = n.mi,  .vitesse = 20, .l = noire       ],
27.         [ .note = n.do,  .vitesse = 40, .l = croche      ],
28.         [ .note = n.re,  .vitesse = 20, .l = blanche * 4 ]
30.     ],
31.     [
32.         [ .note = n.mi, .vitesse = 10, .l = blanche      ],
33.         [ .note = n.la, .vitesse = 10, .l = blanche      ],
34.         [ .note = n.re, .vitesse = 10, .l = blanche      ]
35.     ]
36. ]
```

Ici, nous avons ajouté un autre tableau de structures à notre **tableau** {code:melodie}. Pour accéder à la 2e note du 2e tableau de {code:melodie}. Cela ressemblerait à quelque chose comme {code: melodie[1][1].note}. Si nous voulions accéder à la 7ème note du premier tableau de mélodie, ce serait: {code: melody [0] [6] .note}.

Nous devons maintenant modifier les autres parties du code pour utiliser ces **tableaux** plutôt que des valeurs uniques. Puisque nous avons besoin de deux compteurs de notes et de deux durées, ceux-ci deviennent également de petits **tableaux** :

```
38. duree = [ 0, 0 ]
39. compteurDeNote = [ 0, 0 ]
40.
41. for i = 0 to len( melodie ) loop
```

```
    compteur = 0
    for j = 0 to len( melodie[i] ) loop
        temp = melody[i][j].l
        melody[i][j].l = compteur
        compteur += temp
    repeat
    duree[i] = compteur
repeat
```

Tout est similaire, si ce n'est que les traitements s'effectuent deux fois : une fois pour chaque mélodie de notre tableau.

Nous souhaitons que nos mélodies soient jouées avec différents types d'ondes pour reproduire le son de différents instruments. Nous pouvons utiliser de petits **tableaux** comme {code:duree} et {code:compteurDeNote} pour stocker les informations à appliquer à chaque mélodie :

```
typeOnde = [ 3, 1 ]
octave = [ 24, 12 ]
volume = [ 0.3, 0.2 ]
```

Nous avons ici trois petits **tableaux** qui stockent le modificateur du type d'onde, d'octave et de volume pour chaque mélodie.

Avant de regarder la boucle principale, définissons notre **varaiable** {code:chrono} et la réverbération pour chaque canal :

```
setReverb( 0, 8000, 0.5 )
setReverb( 1, 8000, 0.5 )

chrono = 0
```

Maintenant, créons la boucle principale.

```
loop
    clear()

    for i = 0 to len( melodie ) loop
        if compteurDeNote[i] < len( melodie[i] ) then
            if chrono >= melodie[i][compteurDeNote[i]].l then
                note = note2Freq( melodie[i]
```

-

```
[compteurDeNote[i]].note + octave[i] )
 67.                 vitesse = melodie[i]
[compteurDeNote[i]].vitesse
 68.                 playNote( i, typeOnde[i], note, volume[i],
vitesse, 0.5 )
 69.                 compteurDeNote[i] += 1
 70.             endif
 71.         endif
 72.     repeat
 73.
 74.     chrono += ( 120 / 60 ) / 60
 75.
 76.     if chrono >= duree[0] and chrono >= duree[1] then
 77.         compteurDeNote[0] = 0
 78.         compteurDeNote[1] = 0
 79.         chrono = 0
 80.    endif
 81.
 82.    update()
 83. repeat
```

Là nous l'avons ! Exécutez le programme pour écouter les deux mélodies jouées simultanément. Magnifique !

Ce projet fonctionne dans à peu près tous les scénarios imaginables. La **boucle** que nous utilisons est votre boucle de jeu principale et vous voudrez peut-être qu'elle ne se déclenche qu'à un certain moment. N'hésitez pas à utiliser ce modèle pour vos propres projets !

Bonne composition et à bientôt dans le prochain tutoriel !

Voici le projet terminé ci-dessous au cas où vous auriez du mal à assembler les différentes sections. N'hésitez pas à démarrer un nouveau projet et à copier l'intégralité du projet ci-dessous :

```
  1. n = [
  2.     .do   = 60,
  3.     .dod  = 61,
  4.     .re   = 62,
  5.     .red  = 63,
  6.     .mi   = 64,
  7.     .fa   = 65,
  8.     .fad  = 66,
  9.     .sol  = 67,
 10.     .sold = 68,
```

```
    .la   = 69,
    .lad  = 70,
    .si   = 71
]

double_croche = 0.25
croche = 0.5
noire = 1
blanche = 2

melodie = [
   [
       [ .note = n.do,  .vitesse = 40, .l = croche      ],
       [ .note = n.mi,  .vitesse = 40, .l = croche      ],
       [ .note = n.fa,  .vitesse = 40, .l = croche      ],
       [ .note = n.sol, .vitesse = 40, .l = croche      ],
       [ .note = n.mi,  .vitesse = 20, .l = noire       ],
       [ .note = n.do,  .vitesse = 40, .l = croche      ],
       [ .note = n.re,  .vitesse = 20, .l = blanche * 4 ]
   ],
   [
       [ .note = n.mi, .vitesse = 10, .l = blanche      ],
       [ .note = n.la, .vitesse = 10, .l = blanche      ],
       [ .note = n.re, .vitesse = 10, .l = blanche      ]
   ]
]

duree = [ 0, 0 ]
compteurDeNote = [ 0, 0 ]

for i = 0 to len( melodie ) loop
    compteur = 0
    for j = 0 to len( melodie[i] ) loop
        temp = melody[i][j].l
        melody[i][j].l = compteur
        compteur += temp
    repeat
    duree[i] = compteur
repeat

typeOnde = [ 3, 1 ]
octave = [ 24, 12 ]
volume = [ 0.3, 0.2 ]

setReverb( 0, 8000, 0.5 )
setReverb( 1, 8000, 0.5 )

chrono = 0
```

-

```
 59.
 60. loop
 61.     clear()
 62.
 63.     for i = 0 to len( melodie ) loop
 64.         if compteurDeNote[i] < len( melodie[i] ) then
 65.             if chrono >= melodie[i][compteurDeNote[i]].l then
 66.                 note = note2Freq( melodie[i]
[compteurDeNote[i]].note + octave[i] )
 67.                 vitesse = melodie[i]
[compteurDeNote[i]].vitesse
 68.                 playNote( i, typeOnde[i], note, volume[i],
vitesse, 0.5 )
 69.                 compteurDeNote[i] += 1
 70.             endif
 71.         endif
 72.     repeat
 73.
 74.     chrono += ( 120 / 60 ) / 60
 75.
 76.     if chrono >= duree[0] and chrono >= duree[1] then
 77.         compteurDeNote[0] = 0
 78.         compteurDeNote[1] = 0
 79.         chrono = 0
 80.    endif
 81.
 82.    update()
 83. repeat
```

Fonctions et mots-clés utilisés dans ce tutoriel

[clear()](../Command Reference/Screen Display/clear.md), else, endIf, for, if, loop, note2Freq()(../Command Reference/Sound and Music/note2Freq.md), playNote() (../Command Reference/Sound and Music/playNote.md), repeat, [setReverb] (../Command Reference/Sound and Music/setReverb.md), then, [update()] (../Command Reference/Screen Display/update.md)

Tutoriel 14 : une Introduction aux Vecteurs

-

Activez vos superpouvoirs de concentration, car nous allons aborder quelque chose d'assez délicat. Si vous pouvez comprendre cela, vous avez sérieusement progressé. Comme toujours, maîtriser une nouvelle technique implique de la pratique et d'en faire usage dans vos propres programmes. Il est **fortement** recommandé d'avoir consulté le projet du tutoriel sur les coordonnées d'écran avant d'essayer celui-ci. Si vous êtes déjà à l'aise avec ce sujet, n'hésitez pas à continuer !

Cela dit, nous devrions probablement introduire le sujet

Ce tutoriel concerne les vecteurs. Ce qu'ils sont, pourquoi ils sont utiles et quelques notions simples pour commencer à les utiliser.

Dans ce tutoriel, nous ne traiterons que des aspects très élémentaires sur l'emploi des vecteurs. Pour des informations plus avancées sur leur usage dans FUZE, consultez le prochain projet de vecteur !

Vec-quoi ?!

Qu'est-ce qu'un vecteur ? Bien... Pour répondre aussi simplement que possible :

Un vecteur est *un ensemble de nombres* traité comme une *seule entité*.

Entrons un peu plus dans les détails.

Certaines choses dans la vie peuvent être décrites avec une seule mesure. Prenez la température, par exemple. La température est **seulement** une mesure de chaleur. Rien d'autre.

Nous les appelons des grandeurs **scalaires**. La hauteur, le poids, le volume (sonore et physique) sont tous des exemples de grandeurs **scalaires**.

Mais ... Et si nous voulions décrire quelque chose comme une *position* ? Nous vivons en 3 dimensions, donc décrire la position de quelque chose en utilisant un seul chiffre ne nous donnerait pas beaucoup d'informations !

Prenons l'exemple d'un cercle au centre de notre écran. Voici un code pour le tracer :

```
1. loop
2.     clear()
```

-

```
  3.
  4.     circle( gWidth() / 2, gHeight() / 2, 100, 16, white,
false )
  5.
  6.     update()
  7. repeat
```

Assez simple ! Nous avons une seule **boucle** qui place un cercle à l'écran à l'endroit choisi.

Nous plaçons le cercle à {code:gwidth() / 2,gheight() / 2} qui se trouve exactement au centre de notre écran.

Maintenant, si quelqu'un demandait : "Hé ... Où est ce cercle?", Et si nous répondons avec une seule dimension: "C'est en largeur divisée par 2" ... Vous voyez le problème ?

Le centre du cercle *est* en {code:gwidth() / 2} sur l'axe **x** et en {code:gheight() / 2} sur l'axe **y**. Cela signifie qu'il se situe exactement à l'intersection des lignes droites suivantes :

- verticale (parallèle à l'axe **y**) qui traverse l'axe **x** en {code:gwidth() / 2} et
- horizontale (parallèle à l'axe **x**) qui traverse l'axe **y** en {code:gheight() / 2}.

Ces deux indications sont indispensables pour situer le point. Si seule une coordonnée est utilisée alors ce sont tous les points constituant la ligne droite associée qui sont désignés. Une seule coordonnée n'est donc pas suffisante pour situer un point sur l'écran ! Cette manière de représenter les coordonnées est appelée "cartésienne", et a été inventé au 17e siècle par René Descartes pour mettre en relation la géométrie euclidienne (aussi appelée géométrie plane) et l'algèbre.

Un **vecteur** indique ces deux informations en même temps.

Utiliser un Vecteur dans un Programme

Nous pouvons très facilement créer un vecteur et l'utiliser pour placer notre cercle à l'écran. Découvrez le code utilisant un **vecteur**.

```
1. position = { gWidth() / 2, gHeight() / 2 }
2.
```

-

```
3. loop
4.     clear()
5.
6.     circle( position.x, position.y, 100, 16, white, false )
7.
8.     update()
9. repeat
```

Comme vous pouvez le constater, la ligne 1 définit une **variable** appelée {code:position}. Cette **variable** stocke une paire de coordonnées entre accolades {code:{}}.

Repérez les accolades ! Si vous voyez des accolades dans un code FUZE, vous savez que c'est un **vecteur** !

La partie intelligente sur les **vecteurs** dans **FUZE^ 4 Nintendo Switch** est que nous pouvons simplement définir une **variable** en tant que **vecteur**, mettre ce que nous voulons dans les accolades et FUZE mettra en place une **structure** avec les propriétés **x** et **y**. Dans notre exemple, nous disposons maintenant de {code:position.x} et de {code:position.y}.

"Mais c'est exactement la même chose!" pensez-vous. Eh bien oui, c'est vrai. Mais c'est surtout ce que nous pouvons faire avec les vecteurs qui importe.

Utiliser des Vecteurs pour Déplacer un Cercle

Vous souvenez-vous comment, dans le projet du tutoriel 4 (l'Écran), nous avons appris à déplacer un cercle à travers de l'écran en modifiant ses coordonnées **x** et **y** à l'aide de **variables** ? Dans ce tutoriel, nous améliorer ce projet.

Les **vecteurs** sont composés d'un ensemble de nombres traité comme *une seule entité*, nous pouvons donc appliquer *une* opération à cet ensemble pour tout faire. Regardez :

```
1. position = { gWidth() / 2, gHeight() / 2 }
2.
3. loop
4.     clear()
5.
6.     joy = controls( 0 )
7.     position += { joy.lx, -joy.ly } * 8
8.
```

-

```
 9.     circle( position.x, position.y, 100, 16, white, false )
10.
11.     update()
12. repeat
```

Regardez comment ce code est optimisé !

Au lieu d'avoir des **variables** distinctes pour stocker les positions **x** et **y** et les utiliser séparément, nous pouvons simplement *ajouter* la valeur du joystick gauche à la position sous forme de **vecteur**. Notez que nous ajoutons *les* valeurs {code:joy.lx} et {code:joy.ly} ensemble sous forme de **vecteur**.

Nous avons multiplié ce **vecteur** par 8 pour augmenter la vitesse de déplacement, sinon nous obtenons un mouvement très lent !

Souvenez-vous que le 0 de l'axe **y** est en haut de l'écran et la coordonnée {code:gheight()} tout en bas. Nous devons utiliser un signe moins (-) avant {code:joy.ly} pour que le cercle se déplace dans le bon sens lorsque nous poussons le manche de contrôle vers le haut (ou le bas).

Pour détailler davantage, la ligne 7 ajoute à chaque partie du **vecteur** {code:position} la partie correspondante, soit la valeur {code:joy.lx} à {code:position.x} et la valeur {code:joy.ly} à {code:position.y}.

Cela revient à exécuter deux lignes de code en une seule !

Améliorons un peu plus ce mouvement, puisque nous progressons dans ce tutoriel après tout.

Actuellement, lorsque nous lâchons le manche, notre cercle s'arrête immédiatement. C'est peut-être ce que nous voudrons parfois, mais il est plus réaliste et certainement plus satisfaisant d'avoir un bel effet de ralentissement.

Nous pouvons y parvenir avec quelques lignes de code supplémentaires, et grâce aux **vecteurs**, cela devient plus simple.

Pour obtenir un bel effet de ralentissement, nous avons besoin de la *vélocité*. Modifiez votre code pour qu'il ressemble au programme ci-dessous :

```
1. position = { gWidth() / 2, gHeight() / 2 }
2. velocite = { 0, 0 }
3.
```

-

```
 3. loop
 4.     clear()
 5.
 6.     joy = controls( 0 )
 7.     velocite += { joy.lx, -joy.ly } * 8
 8.     position += velocite
 9.     velocite *= 0.9
10.
11.     circle( position.x, position.y, 100, 16, white, false )
12.
13.     update()
14. repeat
```

Nous avons maintenant une autre **variable** au début du programme appelée {code:velocite}. Cette **variable** stocke un **vecteur nul** ({code: {0, 0}}).

Sur la ligne 7, vous pouvez constater que plutôt que d'augmenter la **variable** {code:position} en fonction des valeurs du stick de contrôle, nous augmentons le **vecteur** {code:velocite}. Séparer des éléments comme celui-ci nous permet de contrôler *combien* de décélération (ralentissement) nous souhaitons produire lorsque nous lâchons le Stick.

Sur la ligne 9, nous appliquons une multiplication au **vecteur** {code:velocite}. En multipliant par 0,9 chaque fois que la **boucle** se répète, nous réduisons constamment ses composantes. Toutefois, cela ne prend vraiment effet que lorsque nous lâchons le Stick, car si nous le maintenons poussé dans une direction, nous ajoutons continuellement les valeurs lues.

Dès que nous relâchons le Stick, l'action de la ligne 9 devient beaucoup plus perceptible. La *vélocité* appliquée à la position du cercle diminue. Nous constations l'un effet de ralentissement.

Pour faire sens, la valeur de ce multiplicateur peut évoluer de 0 à 1. Plus la valeur est proche de 0, plus l'effet est perceptible. À 0 l'arrêt est immédiat. Lorsque la valeur s'approche de 1, le ralentissement devient de moins en moins apparent. À 1, aucun ralentissement ne s'applique.

Les Couleurs sous forme de Vecteurs

Vous le savez peut-être ou non, mais toutes les couleurs à l'écran sont créées à l'aide d'un mélange de lumière rouge, verte et bleue.

-

Lorsque nous configurons un **vecteur**, nous savons que FUZE nous fournit une **structure** avec un {code:.x} et un {code:.y}. En fait, FUZE crée automatiquement un {code:.z} et un {code:.w} en plus de ceux-ci, ce qui nous donne 4 valeurs au total. Si nous ne définissons pas ces valeurs dans un **vecteur**, leur valeur est 0 par défaut.

Les **vecteurs** dans **FUZE^4 Nintendo Switch** permettent *également* d'accéder à ces 4 valeurs avec {code:.r}, {code:.g}, {code:.b} et {code:.a}, correspondant respectivement à rouge, vert, bleu et alpha.

Ce qui signifie que nous pouvons utiliser un **vecteur** comme couleur ! Nous pouvons spécifier la quantité de chaque couleur et une transparence (alpha). Plutôt cool, non ?

Ces nombres sont compris entre 0 et 1: 0 couleur absente, 1 couleur au maximum. Prenons la couleur rouge ({code:red}) par exemple.

Dans **FUZE^4 Nintendo Switch**, la couleur rouge est désignée par le mot {code:red}. C'est en réalité une étiquette pour le **vecteur** {code:{1, 0, 0, 1}}: rouge au maximum, pas de vert, pas de bleu et pleine opacité.

Imaginons que nous voulions créer notre propre couleur. Voyez ci-dessous ce que cela peut donner :

```
1. col = { 0.5, 0.8, 0.1, 1 }
2.
3. loop
4.     clear( col )
5.     update()
6. repeat
```

Ici, nous préparons **vecteur** sur la ligne 1, puis utilisons une simple **boucle** pour effacer l'écran avec notre couleur. Un joli vert menthe !

Pour augmenter la quantité de bleu dans la couleur *pendant* l'exécution de la boucle, nous pouvons faire quelque chose comme ceci :

```
1. col = { 0.5, 0.8, 0.1, 1 }
2.
3. loop
4.     clear( col )
5.     col.b += 0.001
```

-

```
    update()
repeat
```

Exécutez le programme pour voir notre couleur vert menthe passer doucement en bleu clair. Très relaxant !

Résumé

Un **vecteur** est un ensemble de plusieurs nombres traités comme une seule entité.

Vous pouvez facilement les repérer à l'aide des accolades {code:{}}.

Nous les utilisons pour toutes sortes de choses. Ils sont principalement utilisés pour stocker des positions et des couleurs.

Ce tutoriel ne couvre que les bases de l'utilisation des **vecteurs** dans un programme. L'emploi des **vecteurs** nous donne accès à des opérations mathématiques très impressionnantes et utiles, mais nous y reviendrons plus en détail ultérieurement.

Si nous voulons faire *rebondir* des objets les uns sur les autres comme ils le feraient dans le monde réel, les **vecteurs** facilitent grandement les calculs. Cependant, ceci est beaucoup trop compliqué pour notre introduction aux **vecteurs**, nous y reviendrons donc plus tard.

Bien joué. Nous allons de l'avant maintenant ! Essayez de définir vos propres **vecteurs** dans vos projets et utilisez-les pour contrôler la position des objets ou pour changer la couleur de certaines choses. Comprendre tout ceci ouvre vraiment un monde d'expérimentation !

Rendez-vous dans le prochain tutoriel!

Fonctions et mots-clés utilisés dans ce tutoriel

[circle()](../Command Reference/2D Graphics/circle.md), [clear()](../Command Reference/Screen Display/clear.md), loop, repeat, [update()](../Command Reference/Screen Display/update.md)

-

Tutoriel 15 : les Sprites

Dans **FUZE^4 Nintendo Switch**, il existe toute une série de **fonctions** et de commandes qui nous aident à utiliser la grande quantité d'assets disponibles pour créer un jeu.

Dans ce tutoriel, nous allons créer un jeu simple en utilisant le système de sprite. Comme toujours, nous allons commencer simplement et complexifier au fur et à mesure.

Créer un Sprite

Mettons les bases en place. Nous devons charger une image, puis utiliser cette image pour créer un sprite :

```
1. joueurSpr = createSprite()
2. joueurImg = loadImage( "Untied Games/Player ships", false )
3. setSpriteImage( joueurSpr, joueurImg )
```

Terminé ! Nous utilisons d'abord la **fonction** {code:createSprite()} et stockons le résultat dans une **variable** appelée {code:joueurSpr}. Cela crée un sprite nommé {code:joueurSpr} que nous pouvons manipuler en utilisant les autres **fonctions** dédiées aux sprites.

Avant d'y arriver, nous devons assigner un fichier image au sprite sinon nous n'aurons rien à voir !

Sur la ligne 2, nous utilisons la **fonction** {code:loadImage()} pour stocker un fichier image dans une **variable** appelée {code:joueurImg}. Nous utilisons ensuite la **fonction** {code:setSpriteImage()} pour affecter ce fichier image à notre sprite sur la ligne 3.

Maintenant, nous pouvons plein de choses cool !

setSpriteLocation()

Commençons simplement par afficher le sprite à l'écran. Nous devons d'abord définir son emplacement à l'écran :

-

```
  4. setSpriteLocation( joueurSpr, { gwidth() / 2, gheight() / 2 }
)
```

Ici, nous utilisons la **fonction** {code:setSpriteLocation()} pour donner une position à notre sprite de joueur. Nous avons utilisé {code:gwidth() / 2, gheight() / 2} pour nous donner le milieu de l'écran.

C'est vraiment tout ce dont nous avons besoin pour commencer. Dessinons notre sprite sur l'écran à l'aide d'une **boucle**.

```
  6. loop
  7.     clear()
  8.     drawSprites()
  9.     update()
 10. repeat
```

Exécutez le programme pour voir 4 minuscules vaisseaux spatiaux au milieu de l'écran.

setSpriteScale()

Vous avez peut-être remarqué que ces vaisseaux sont un peu petits... Nous pouvons **aggrandir** l'image pour la rendre plus exploitable. Ajoutez la ligne suivante avant {code: loop} :

```
  5. setSpriteScale( joueurSpr, { 4, 4 } )
  6.
  7. loop
  8.     clear()
  9.     drawSprites()
 10.     update()
 11. repeat
```

La **fonction** {code:setSpriteScale()} nous permet de **multiplier** les tailles **x** et **y** d'un sprite. Nous avons utilisé le nombre {code:4} pour **x** et **y** afin de rendre notre sprite 4 fois plus grand. Si ces nombres sont différents, l'image est étirée !

Exécutez le programme et maintenant notre sprite devrait avoir une taille plus appropriée.

-

setSpriteAnimation()

Notre fichier image comporte une collection de 4 vaisseaux pour vous permettre de faire votre choix. Puisque le sprite est créé à partir de l'image entière, nous avons un sprite composé des 4 vaisseaux. Pour l'utiliser dans un jeu, il est préférable de n'avoir qu'un seul vaisseau, et non les 4 à la fois.

Nous pouvons utiliser la **fonction** {code:setSpriteAnimation()} pour le faire. Cette intelligente **fonction** indique à FUZE les **tuiles** d'une image à afficher. Elle s'utilise pour animer un sprite, mais nous l'emploierons ici pour afficher une seule tuile.

Ajoutez la ligne suivante avant {code: loop} :

```
 6. setSpriteAnimation( joueurSpr, 0, 0, 0 )
 7.
 8. loop
 9.     clear()
10.     drawSprites()
11.     update()
12. repeat
```

Examinons cette **fonction** plus en détail. Le premier argument est la **variable** qui stocke notre sprite. Facile.

Ensuite, nous avons trois nombres. Le premier d'entre eux est la **tuile de départ** pour l'animation. Cela indique à FUZE par laquelle des tuiles de l'image commencer. Regardez le graphique ci-dessous :

Comme vous pouvez le constater, la numéroation des tuiles commencent à 0 et augmente de 1 en 1. Nous souhaitons utiliser la première tuile (0). Cela signifie que le premier nombre de notre **fonction** {code:setspriteAnimation ()} est égal à 0.

Le deuxième numéro est celui de la tuile **finale** de notre animation. Puisque nous voulons que notre vaisseau reste le même, ce nombre est également 0.

-

Le dernier numéro de la **fonction** est la **vitesse** de l'animation en **images par seconde**. Comme notre animation n'est composée que d'une seule tuile, la vitesse est de 0.

Exécutez le programme et admirez votre magnifique vaisseau parfaitement dimensionné à l'écran.

Vous trouvez cela ennuyeux? Très bien, déplaçons le sprite.

setSpriteSpeed()

Pour déplacer réellement le sprite, nous devons utiliser la **fonction** {code:setSpriteSpeed()}.

Nous pouvons utiliser cette **fonction** pour appliquer une direction et une vitesse de déplacement à un sprite :

```
 7. setSpriteSpeed( joueurSpr, { 60, 0 } )
 8.
 9. loop
10.     clear()
11.     updateSprites()
12.     drawSprites()
13.     update()
14. repeat
```

Notez que nous avons ajouté deux lignes cette fois-ci. Puisque nous voulons que la position de notre sprite change pendant l'exécution du programme, nous devons utiliser la **fonction** {code:updateSprites()} dans la **boucle** principale. Elle **doit** être exécutée avant la fonction {code:drawSprites()}.

Dans la **fonction** {code:setSpriteSpeed()} requiert deux arguments. Le premier est bien sûr la **variable** qui stocke le sprite que nous voulons déplacer.

Le second est un vecteur bidimensionnel qui définit la vitesse de déplacement en pixels pour chaque axe. Dans notre ligne, nous avons appliqué une vitesse de {code:60} à l'axe **x** et une vitesse de {code: 0} à l'axe **y**. Cela signifie que notre vaisseau se déplace vers la droite à 60 pixels par seconde.

Exécutez le programme pour voir le vaisseau spatial se déplacer gracieusement le long de l'axe **x**.

-

Créer un jeu en utilisant les fonctions Sprite

Dans le prochain tutoriel, nous utiliserons tout ce que nous avons appris pour créer un jeu à défilement horizontal dans lequel nous devrons éviter des météorites.

Assurez-vous de bien connaître les **fonctions** que nous avons décrites ici, avant de nous rejoindre dans le prochain projet !

Fonctions et mots-clés utilisés dans ce tutoriel

[createSprite()](../Command Reference/2D Graphics/createSprite.md), [clear()](../Command Reference/Screen Display/clear.md), [drawSprites()](../Command Reference/2D Graphics/drawSprites.md), loop, repeat, [setSpriteAnimation](../Command Reference/2D Graphics/setSpriteAnimation.md), [setSpriteImage()](../Command Reference/2D Graphics/setSpriteImage.md), setSpriteSpeed() (../Command Reference/2D Graphics/setSpriteSpeed.md), [update()](../Command Reference/Screen Display/update.md), [updateSprites()](../Command Reference/2D Graphics/updateSprites.md)

Tutoriel 16: Créer un jeu en utilisant les fonctions Sprite

Bienvenue ! Dans ce tutoriel, nous allons créer un jeu simple en utilisant les **fonctions** sprite présentées dans le projet précédent.

Nous utiliserons les mêmes vaisseaux spatiaux du très talentueux Untied Games, ainsi que de superbes météorites.

Dans ce jeu, notre vaisseau tombera toujours vers le bas. La difficulté réside dans le fait que nous devrons appuyer sur le bouton A pour que notre vaisseau continue à voler. Cela pourrait vous rappeler un jeu auquel vous avez déjà joué !

Commençons comme précédemment en créant le sprite du joueur à partir d'un fichier image :

```
1. joueurSpr = createSprite()
2. joueurImg = loadImage( "Untied Games/Player ships", false )
```

-

```
3. setSpriteImage( joueurSpr, joueurImg )
```

Ensuite, nous définissons l'emplacement du sprite.

Pour nous aider, nous devons créer une variable qui stocke la largeur et la hauteur de l'écran car nous aurons souvent besoin de ces valeurs :

Créez les nouvelles lignes à partir de la ligne 1. Cela décalera légèrement les autres lignes :

```
1. ecran = { gwidth(), gheight() }
2.
3. joueurSpr = createSprite()
4. joueurImg = loadImage( "Untied Games/Player ships", false )
5. setSpriteImage( joueurSpr, joueurImg )
```

Nous avons défini une **variable** appelée {code:ecran} qui stocke un vecteur. Les propriétés **x** et **y** de ce vecteur prennent respectivement {code:gwidth()} et {code:gheight()} pour valeur. Ce qui signifie que nous pouvons maintenant accéder à la largeur et à la hauteur de l'écran en utilisant {code:ecran.x} et {code:ecran.y}. Très pratique !

Ceci fait, nous devons créer une **variable** qui stocke le multiplicateur d'**échelle** de nos sprites. Cela nous évitera de le saisir à l'avenir ! Ajoutez la nouvelle ligne (2) :

```
1. ecran = { gwidth(), gheight() }
2. echelle = { ecran.y / 270, ecran.y / 270 }
3.
4. joueurSpr = createSprite()
5. joueurImg = loadImage( "Untied Games/Player ships", false )
6. setSpriteImage( joueurSpr, joueurImg )
```

La **variable** {code:echelle} stocke un **vecteur** à utiliser dans les **fonctions** {code:setSpriteScale()}. En utilisant {code:ecran.y / 270}, notre **échelle** change selon que la console est sur sa base ou non. Nous utilisons le même nombre pour les propriétés **x** et **y** du **vecteur** afin de ne pas déformer nos sprites.

Maintenant, nous pouvons créer une **variable** qui stocke la position du joueur pour pouvoir s'y référer facilement :

-

```
1. ecran = { gwidth(), gheight() }
2. echelle = { ecran.y / 270, ecran.y / 270 }
3.
4. joueurSpr = createSprite()
5. joueurImg = loadImage( "Untied Games/Player ships", false )
6. setSpriteImage( joueurSpr, joueurImg )
7. joueurPos = { screen.x / 20, screen.y / 2 }
```

La nouvelle **variable** s'appelle {code:joueurPos}. Elle stocke un **vecteur** désignant un emplacement particulier à l'écran. Nous pouvons maintenant accéder à la position en utilisant {code:joueurPos.x} et {code:joueurPos.y}.

Enfin, nous pouvons également stocker la vitesse du joueur dans une **variable** afin de faciliter l'écriture du programme par la suite. Ajoutons un **vecteur nul** pour l'instant :

```
1. ecran = { gwidth(), gheight() }
2. echelle = { ecran.y / 270, ecran.y / 270 }
3.
4. joueurSpr = createSprite()
5. joueurImg = loadImage( "Untied Games/Player ships", false )
6. setSpriteImage( joueurSpr, joueurImg )
7. joueurPos = { screen.x / 20, screen.y / 2 }
8. playerVel = { 0, 0 }
```

Très bien, utilisons les **fonctions** sprite pour exploiter toutes ces informations.

```
10. setSpriteAnimation( joueurSpr, 0, 0, 0 )
11. setSpriteScale( joueurSpr, echelle )
12. setSpriteLocation( joueurSpr, joueurPos )
```

Dessiner et Déplacer le joueur

Bien ! Créons une **boucle** pour dessiner le sprite à l'écran.

```
14. loop
15.     clear()
16.     updateSprites()
17.     drawSprites()
```

-

```
 18.     update()
 19. repeat
```

Actuellement, notre **boucle** affiche uniquement le sprite à l'écran. Ajoutons quelques contrôles à ce programme. Avant tout, nous voulons recalculer les **variables** {code:ecran} et {code:echelle} dans la **boucle**. Cela signifie que si nous passons du mode portable au mode TV en plaçant la console Nintendo Switch sur sa station d'accueil, l'écran et l'échelle sont mis à jour afin de s'adapter au changement de résolution :

```
 14. loop
 15.     clear()
 16.     ecran = { gwidth(), gheight() }
 17.     echelle = { ecran.y / 270, ecran.y / 270 }
 18.     setSpriteScale( joueurSpr, echelle )
 19.     updateSprites()
 20.     drawSprites()
 21.     update()
 22. repeat
```

Nous y voilà. Nous pouvons maintenant utiliser les **variables** {code:ecran} et {code:echelle} dans la **boucle** sans se soucier de savoir si la console est en mode de portable ou en mode TV. Pour que l'échelle change réellement, nous utilisons la **fonction** {code:setSpriteScale()} dans la **boucle** à la ligne 18.

Ajoutons maintenant quelques contrôles :

```
 14. loop
 15.     clear()
 16.     ecran = { gwidth(), gheight() }
 17.     echelle = { ecran.y / 270, ecran.y / 270 }
 18.
 19.     c = controls( 0 )
 20.
 21.     joueurVel = { c.lx * ecran.x / 4, ecran.y / 3 - c.a *
ecran.y / 1.5 }
 22.
 23.     setSpriteSpeed( joueurSpr, joueurVel )
 24.     setSpriteScale( joueurSpr, echelle )
 25.
 26.     updateSprites()
 27.     drawSprites()
```

-

```
28.     update()
29. repeat
```

Nous avons ajouté un espace entre les lignes du programme ci-dessus pour clarifier un peu les choses.

Trois nouveaux segments de code prennent place sur les lignes 19, 21 et 23.

Tout d'abord, nous utilisons la **fonction** {code:controls ()} pour stocker l'état des contrôles dans une **variable** nommée {code:c}.

Ensuite, nous mettons à jour la **variable** {code:joueurVel}. Les deux valeurs pouvant affecter la vitesse sont situées entre les accolades.

```
21.     joueurVel = { c.lx * ecran.x / 4, ecran.y / 3 - c.a *
ecran.y / 1.5 }
```

Nous voulons pouvoir faire un léger mouvement de va-et-vient avec le vaisseau pour accélérer et ralentir. Pour cela, nous utilisons le stick gauche ({code:c.lx}).

Étant donné que le stick renvoie une valeur comprise entre -1 et 1, son effet est vraiment minime. Nous l'avons multiplié par {code:ecran.x / 4} pour augmenter la vitesse de déplacement. En utilisant la variable {code:ecran.x}, nous pouvons nous assurer que la vitesse est la même en mode portable ou en mode TV.

De même, pour la vitesse de l'axe **y**, nous utilisons le {code:ecran.y} pour déplacer le vaisseau vers le bas à une vitesse dépendant de la taille de l'écran. Nous soustrayons la valeur du bouton A ({code:c.a}) pour déplacer le vaisseau vers le haut. Puisque le bouton A vaut 0 ou 1, nous le multiplions par {code:ecran.y / 1.5} pour augmenter l'effet.

Pour que le mouvement ait réellement lieu, nous devons appliquer les nouvelles valeurs de la **variable** {code:joueurVel} à la vitesse du sprite.

Sur la ligne 23, nous utilisons la **fonction** {code:setSpriteSpeed()} pour appliquer les valeurs de la variable {code:joueurVel} au sprite du joueur.

Lancez le programme et ressentez le mouvement de l'écran ! N'oubliez pas que vous devez maintenir le bouton A enfoncé pour pouvoir monter et que le levier de commande gauche accélère et ralentit le vaisseau.

-

Créer des limites

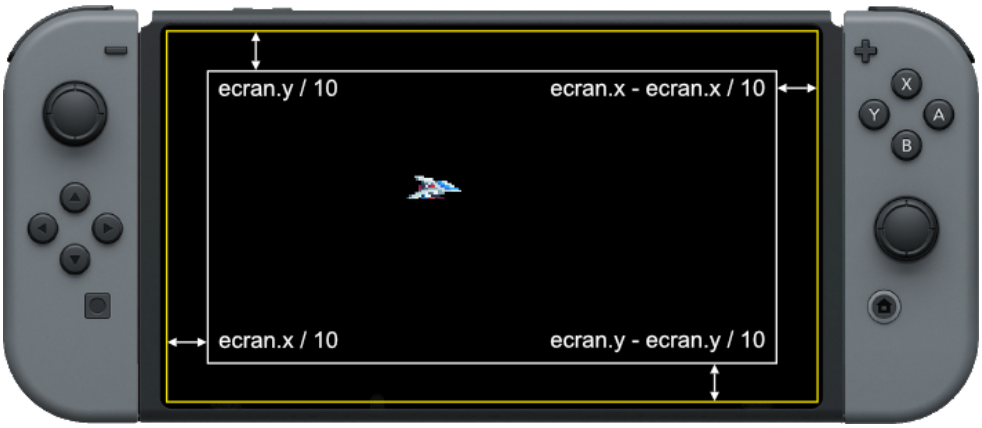
Pour le moment, nous pouvons déplacer notre vaisseau hors de l'écran. Ce n'est pas très bon et rend le jeu trop facile. Introduisons des limites à la zone de déplacement offerte au joueur.

Ci-dessous, nous utilisons une nouvelle **fonction** appelée {code:clamp()} pour restreindre les valeurs de la **variable** {code:joueurPos}. Utilisée correctement, cette **fonction** très utile peut nous faire économiser beaucoup **d'alternatives "Si..."**. La première indication entre parenthèses est la valeur à restreindre, la seconde est la limite inférieure et la troisième est la limite supérieure.

```
 14. loop
 15.     clear()
 16.     ecran = { gwidth(), gheight() }
 17.     echelle = { ecran.y / 270, ecran.y / 270 }
 18.
 19.     c = controls( 0 )
 20.
 21.     joueurVel = { c.lx * ecran.x / 4, ecran.y / 3 - c.a *
ecran.y / 1.5 }
 22.
 23.     joueurPos = getSpriteLocation( joueurSpr )
 24.
 25.     joueurPos.x = clamp( joueurPos.x, ecran.x / 10, ecran.x -
ecran.x / 10 )
 26.     joueurPos.y = clamp( joueurPos.y, ecran.y / 10, ecran.y -
ecran.y / 10 )
 27.
 28.     setSpriteLocation( joueurSpr, joueurPos )
 29.     setSpriteSpeed( joueurSpr, joueurVel )
 30.     setSpriteScale( joueurSpr, echelle )
 31.
 32.     updateSprites()
 33.     drawSprites()
 34.     update()
 35. repeat
```

Pour comprendre pourquoi nous utilisons les valeurs {code: ecran.x / 10} et {code: ecran.x - ecran.x / 10}, visualisons ce que nous réalisons :

-



Sur l'image, le rectangle extérieur **jaune** délimite l'écran de la Nintendo Switch. La largeur et la hauteur de cet écran sont stockées dans les **variables** {code:ecran.x} et {code:ecran.y}.

Le rectangle intérieur **blanc** dessine les limites que nous avons créées.

Si la position du joueur devient supérieure ou inférieure aux limites de notre rectangle intérieur, notre **fonction** {code:clamp()} limite les valeurs.

Pour effectivement placer le sprite du joueur à l'écran aux positions restreintes, nous devons définir l'emplacement du sprite à l'aide de la **fonction** {code:setSpriteLocation()}. Pour que cela fonctionne, nous utilisons la **fonction** {code:getSpriteLocation()} à la ligne 23 pour nous assurer que le sprite du joueur est toujours placé à l'emplacement où il doit être.

Créer des obstacles

Ce n'est pas vraiment un jeu très amusant à moins d'avoir quelque chose à faire ! Créons un tas de rocs volants à éviter.

Commençons par charger les images dont nous avons besoin et créons les sprites.

Nous utiliserons davantage d'assets du talentieux Untied Games pour cela. Il existe 4 types d'astéroïdes différents à utiliser. Nous allons tous les stocker dans un tableau.

Nous devons faire les changements suivants avant la **boucle** :

-

```
14. rocImgs = [
15.     loadImage( "Untied Games/Asteroid A", false ),
16.     loadImage( "Untied Games/Asteroid B", false ),
17.     loadImage( "Untied Games/Asteroid C", false ),
18.     loadImage( "Untied Games/Asteroid D", false )
19. ]
```

Voici le tableau d'images. Avec lui, nous pouvons accéder à n'importe quelle image avec une expression du type {code:rockImgs[1]}.

Ensuite, nous devons créer le tableau d'informations que nous utiliserons pour les rocs qui apparaissent à l'écran. Nous appellerons ce tableau {code:rocs} et nous le créons juste sous le tableau d'images :

```
21. array rocs[50] = [
22.     .spr = 0,
23.     .pos = { 0, 0 },
24.     .vel = { 0, 0 }
25. ]
```

C'est ça ! Nous avons créé un tableau de 50 éléments dotés de trois propriétés. Chaque roc a une propriété {code:.spr} pour stocker son sprite, une propriété {code:.pos} pour stocker sa position à l'écran et une propriété {code:.vel} pour stocker sa vélocité.

Maintenant, remplissons ce tableau avec les informations nécessaires. Nous avons besoin d'une **boucle "Pour..."** :

```
27. for i = 0 to len( rocs ) loop
28.     n = random( 4 )
29.     rocs[i].spr = createSprite()
30.     setSpriteImage( rocs[i].spr, rocImgs[n] )
31.     rocs[i].pos = { ecran.x + random( ecran.x * 2 ), random(
ecran.y ) }
32.     rocs[i].vel = { - random( ecran.x / 5 ), random( 33 ) -
16 }
33.     setSpriteAnimation( rocs[i].spr, 0, 39, 10 )
34.     setSpriteLocation( rocs[i].spr, rocs[i].pos )
35. repeat
```

-

Cette **boucle "Pour..."** parcours les indices des éléments du **tableau** {code: rocks}. Regardons chaque ligne en détail :

```
28.     n = random( 4 )
```

Tout d'abord, nous créons une **variable** appelée {code:n} qui stocke un nombre aléatoire renvoyé par {code:random(4)}. Cela nous donne 0, 1, 2 ou 3. Ce nombre aléatoire est utilisé pour sélectionner une image de roc dans le **tableau** {code:rocImgs}.

```
29.     rocs[i].spr = createSprite()
30.     setSpriteImage( rocs[i].spr, rocImgs[n] )
```

À la ligne 29, nous utilisons la **fonction** {code:createSprite()} pour créer un sprite pour chaque roc. Sur la ligne suivante, nous utilisons la **fonction** {code:setSpriteImage()} pour lui associer l'image de roc sélectionnée de façon aléatoire.

```
31.     rocs[i].pos = { ecran.x + random( ecran.x * 2 ), random(
ecran.y ) }
```

Ensuite, à la ligne 31, nous définissons la propriété {code:.pos} de chaque roc pour stocker une position. Nous voulons que les rocs soient répartis de manière aléatoire sur l'axe **y**, de sorte que la partie **y** du vecteur est {code:random(ecran.y)}. Puisque nous voulons commencer sans aucune pierre à l'écran et les faire voyager vers nous, la partie **x** du vecteur est {code:ecran.x + random (ecran.x * 2)}. Cela garantit que les rocs sont répartis de façon aléatoire au-delà du bord droit de l'écran pour commencer.

```
32.     rocs[i].vel = { - random( ecran.x / 5 ), random( 33 ) -
16 }
```

Sur la ligne 32, nous affectons à la propriété {code:.vel} la vélocité de chaque roc. La vitesse à laquelle les rocs se dirigent vers nous doit dépendre de la taille de l'écran, sinon, en fonction du mode portable ou TV, les rocs se déplaceraient à des vitesses différentes. Nous utilisons un nombre aléatoire choisi entre 0 et {code: -ecran.x / 5 + 1}. Nous devons mettre un {code: -} devant {code: ecran.x} pour nous assurer que les astéroïdes voyagent vers nous au lieu de s'éloigner !

-

Nous voulons que les rocs montent ou descendent lentement pour vraiment ajouter de la variété. Pour cela, nous utilisons un nombre aléatoire entre 0 et 32 dans la partie **y** du vecteur, mais si nous soustrayons 16 du résultat, nous avons une plage de -16 à 16, ce qui nous donne un mouvement choisi de manière aléatoire vers le haut ou vers le bas à des vitesses variées.

```
 33.     setSpriteAnimation( rocs[i].spr, 0, 39, random( 11 ) + 10
)
```

Sur la ligne 33, nous définissons l'animation pour chaque roc. Chaque image d'astéroïde comporte 40 tuiles d'animation, la première tuile porte le numéro 0 et la dernière le numéro 39. Pour obtenir différentes vitesses d'animation des rocs, nous choisissons une valeur aléatoire entre 10 et 20.

```
 34.     setSpriteLocation( rocs[i].spr, rocs[i].pos )
```

Pour terminer, nous définissons l'emplacement de chaque roc à l'aide de la **fonction** {code:setSpriteLocation()}.

Toujours là ? Je parie que vous vous dites : "Waouh, c'est clair comme de l'eau de roche !".

...

Vous pensez que ce n'est pas une bonne blague ? D'accord, continuons.

Dessiner les rocs à l'écran

Maintenant que nous avons toutes les informations nécessaires dans le **tableau** {code:rocs}, nous pouvons les dessiner à l'écran. Nous retournons dans la boucle principale pour cela. Parce que nous avons inséré pas mal de lignes avant la **boucle**, les numéros de lignes sont un peu différents maintenant. Pour être sûr que vous ajoutez le code au bon emplacement, assurez-vous que votre boucle principale ressemble à celle ci-après avant de commencer :

```
37. loop
38.     clear()
39.     ecran = { gwidth(), gheight() }
40.     echelle = { ecran.y / 270, ecran.y / 270 }
```

-

```
 41.
 42.     c = controls( 0 )
 43.
 44.     joueurVel = { c.lx * ecran.x / 4, ecran.y / 3 - c.a *
ecran.y / 1.5 }
 45.
 46.     joueurPos = getSpriteLocation( joueurSpr )
 47.
 48.     joueurPos.x = clamp( joueurPos.x, ecran.x / 10, ecran.x -
ecran.x / 10 )
 49.     joueurPos.y = clamp( joueurPos.y, ecran.y / 10, ecran.y -
ecran.y / 10 )
 50.
 51.     setSpriteLocation( joueurSpr, joueurPos )
 52.     setSpriteSpeed( joueurSpr, joueurVel )
 53.     setSpriteScale( joueurSpr, echelle )
 54.
 55.     updateSprites()
 56.     drawSprites()
 57.     update()
 58. repeat
```

Nous avons besoin d'ajouter une **bouble "Pour..."*** avant la ligne {code:updateSprites()}. Aller à la ligne 54 et créer quelques nouvelles lignes :

```
 53.     setSpriteScale( playerSpr, scale )
 54.
 55.
 56.
 57.     updateSprites()
```

Nous allons ajouter le début de la **boucle "Pour..."** à la ligne 55. Nous devons parcourir le **tableau** {code:rocs} et pour chaque roc faire un certain nombre de choses :

```
 55.     for i = 0 to len( rocks ) loop
 56.         setSpriteScale( rocs[i].spr, echelle )
 57.         setSpriteSpeed( rocs[i].spr, rocs[i].vel )
 58.         roc[i].pos = getSpriteLocation( rocs[i].spr )
 59.         rocTaille = getSpriteSize( rocs[i].spr )
 60.         if rocs[i].pos.x + rocTaille.x / 2 < 0 then
 61.             rocs[i].pos = { ecran.x + random( ecran.x ),
random( ecran.y ) }
 62.             setSpriteLocation( rocs[i].spr, rocs[i].pos )
```

-

```
63.         endif
64.     repeat
65.
66.     updateSprites()
67.     drawSprites()
68.     update()
69. repeat
```

Pfiu ! Ce morceau de code à l'air plutôt compliqué. N'ayez pas peur, en fait il est assez simple si nous l'examinons étape par étape.

```
56.         setSpriteScale( rocs[i].spr, echelle )
```

La première chose que nous effectuons dans la **boucle "Pour..."** est de mettre chaque roc à l'échelle souhaitée avec la **fonction** {code:setSpriteScale()}.

```
57.         setSpriteSpeed( rocs[i].spr, rocs[i].vel )
```

Ensuite, nous appliquons la vitesse de chaque roc au sprite. Nous utilisons la **fonction** {code:setSpriteSpeed()} pour appliquer le **vecteur** stocké dans {code:rocs[i].vel} à chaque sprite. Cette ligne déplace chaque roc à travers l'écran vers le haut ou le bas selon la composante **y** de la vélocité.

```
58.         rocks[i].pos = getSpriteLocation( rocs[i].spr )
```

Après cela, nous mettons à jour la variable de chaque roc contenant sa position. Procéder ainsi nous permet d'écrire du code optimisé par la suite dans dans la **boucle "Pour..."**. En effet dans une minute, nous allons vérifier la position courante de chaque roc grâce à cela.

```
59.         rocTaille = getSpriteSize( rocs[i].spr )
```

D'une façon similaire, nous créons une **variable** appelée {code:rocTaille} et l'utilisons pour stocker la taille de chaque roc à l'écran. Ce sera une **variable** très utile dans une minute !

```
60.         if rocs[i].pos.x + rocTaille.x / 2 < 0 then
61.             rocs[i].pos = { ecran.x + random( ecran.x ),
```

-

```
random( ecran.y ) }
 62.             setSpriteLocation( rocs[i].spr, rocs[i].pos )
 63.         endif
```

Lorsqu'un roc se déplace du côté gauche de l'écran, nous devons agir pour le ramener du côté droit. Sinon nous aurions besoin d'un nombre gigantesque de rocs ! Cette **alternative "Si..."** permet de réutiliser chaque roc dans le **tableau** lorsqu'il quitte l'écran.

Nous vérifions si la position **x** du côté droit de charque roc ({code:rocks[i].pos.x + rockSize.x / 2}) est devenue plus petite que 0. Si c'est le cas, nous mettons à jour la **variable** {code:rocks[i].pos} pour replacer aléatoirement le roc plus ou moins loin au-dela du côté droit de l'écran en agissant sur l'axe **x** ({code:screen.x + random(screen.x)}) et avec une position au hasard sur l'axe **y** ({code:random(screen.y)}). Enfin, nous utilisons la **fonction** {code:setSpriteLocation()} pour mettre à jour l'emplacement du sprite.

Terminé !

Lancer l'exécution du programme pour voir vos rocs voler à travers lécran vers vous. Lors qu'un roc sort de l'écran à gauche, il est revient éventuellement sur l'écran par la droite !

Exercez-vous à manoeuvrer autour des rocs ! Nous sommes sur le point d'ajouter les collisions.

Collision avec les rocs

Avant d'écrire le code qui nous permet de gérer les collisions avec un roc, ce serait très cool d'avoir un aperçu de ce que donnerait un effet d'explosion lorsque cela se produire.

Heureusement, nous avons beaucoup d'effets d'explosion impressionnants ! Vous connaissez l'exercice, nous devons charger une image et créer un sprite. Ajoutez les lignes suivantes juste avant la **boucle** principale (vous devrez créer quelques lignes) :

```
37. expSpr = createSprite()
38. expImg = loadImage( "Untied Games/Explosion 09", false )
39. setSpriteImage( expSpr, expImg )
```

-

Comme d'habitude, nous créons une **variable** pour stocker le fichier image. Nous avons appelé la nôtre {code:expImg}. Ensuite, nous créons une autre **variable** ({code: expSpr}) pour stocker le sprite.

Nous devons ensuite définir la visibilité de ce sprite sur {code:false}. Nous ne voulons pas encore voir l'explosion, elle s'affichera seulement lorsque le joueur heurtera un roc. Ajoutez la ligne suivante :

```
40. setSpriteVisibility( expSpr, false )
```

La **fonction** {code:setSpriteVisibility ()} est incroyablement utile et simple. Le premier argument est la **variable** sprite et le second est {code:true} pour visible ou {code:false} pour invisible.

Enfin, nous aurons besoin d'une sorte d'indicateur pour nous dire si le joueur est en vie ou non. Cela doit être une **variable**, placée au début de notre programme, prenant pour valeur {code:true} (vrai) ou {code:false} (faux). Ajoutez la ligne suivante juste avant la ligne {code:loop} :

```
41. vivant = true
```

Maintenant nous avons tout ce dont nous avons besoin pour produire l'effet d'explosion, nous avons juste besoin d'entrer en collision avec le roc !

Ce prochain morceau de code aura lieu dans la **boucle "Pour..."** qui dessine les rocs car nous devons vérifier la distance entre le joueur et **chaque roc**.

Ci-après l'ensemble de la **boucle "Pour..."** pour plus de clarté :

```
61.     for i = 0 to len( rocs ) loop
62.         setSpriteScale( rocs[i].spr, echelle )
63.         setSpriteSpeed( rocs[i].spr, rocs[i].vel )
64.         rocs[i].pos = getSpriteLocation( rocs[i].spr )
65.         rocTaille = getSpriteSize( rocs[i].spr )
66.
67.         if rocs[i].pos.x + rocTaille.x / 2 < 0 then
68.             rocs[i].pos = { ecran.x + random( ecran.x ),
random( ecran.y ) }
69.             setSpriteLocation( rocs[i].spr, rocs[i].pos )
70.         endif
71.
```

-

```
 72.         if distance( joueurPos, rocs[i].pos ) < rocTaille.x /
2 - 50 and vivant then
 73.             vivant = false
 74.             setSpriteAnimation( expSpr, 0, 89, 14 )
 75.             setSpriteLocation( expSpr, joueurPos )
 76.             setSpriteScale( expSpr, echelle )
 77.             setSpriteVisibility( expSpr, true )
 78.             setSpriteVisibility( joueurSpr, false )
 79.         endif
 80.     repeat
 81.
 82.     updateSprites()
 83.     drawSprites()
 84.     update()
 85. repeat
```

Notre nouvelle section de code est **l'alternative "Si..."** à partir de la ligne 72. Comme d'habitude, nous allons parcourir cette section, ligne par ligne :

```
 72.         if distance( joueurPos, rocs[i].pos ) < rocTaille.x /
2 - 50 and vivant then
```

Pour entrer en collision avec un roc, notre position doit chevaucher celle du roc. Nous vérifions si la distance entre le centre de notre vaisseau ({code:joueurPos}) et le centre de chaque roc ({code:rocs[i].pos}) est inférieure à la moitié de la taille de ce roc moins 50 pixels ({code: <rockSize.x / 2 - 50}). La marge de 50 pixels est une préférence personnelle, changez ce nombre rendra la collision plus ou moins serrée.

La seconde partie de **l'alternative "Si..."** vérifie si la variable {code:vivant} est {code: true}. Nous voulons entrer en collision avec un roc seulement **une fois** (car après le joueur ne sera plus vivant).

```
 73.             vivant = false
```

La première chose que nous faisons dans **l'alternatice "Si..."** consiste à affecter la valeur {code:false} à la **variable** {code:vivant}. Cela qui signifie que nous ne pouvons entrer en collision qu'une seule fois. Ainsi suite à la première colision, la condition de **l'alternative "Si..."** ne peut plus se vérifier.

```
 74.             setSpriteAnimation( expSpr, 0, 89, 14 )
```

-

Ensuite, nous configurons l'animation pour l'explosion du sprite. L'explosion est composée de 90 images, commençant par 0 et se terminant par 89. Une vitesse de 14 nous donne une belle explosion, n'hésitez pas à changer cette valeur !

```
75.             setSpriteLocation( expSpr, joueurPos )
```

Maintenant, nous devons définir l'emplacement du sprite d'explosion comme étant le même que celui du joueur ! Nous ne voudrions pas que l'explosion se produise à un autre endroit sur l'écran.

```
76.             setSpriteScale( expSpr, scale )
```

Ensuite, nous définissons le multiplicateur d'échelle pour le sprite d'explosion. Assez simple!

```
77.             setSpriteVisibility( expSpr, true )
78.             setSpriteVisibility( joueurSpr, false )
```

Les deux lignes suivantes définissent la visibilité du joueur sur {code:false} et la visibilité de l'explosion sur {code:true}. Sans cela, nous ne verrions rien !

C'est ça. Nous sommes très près de la ligne d'arrivée !

Nous avons une dernière chose à faire. Une fois l'animation de l'explosion terminée, nous devons redéfinir la visibilité sur {code:false}. Sans cela, l'animation d'explosion continuera à boucler pour toujours !

Nous avons besoin d'une **alternative "Si..."** juste avant l'appel de la **fonction** {code:updateSprites()} :

```
82.     if getSpriteAnimFrame( expSpr ) >= 89 then
83.         setSpriteVisibility( expSpr, false )
84.     endif
85.
86.     updateSprites()
87.     drawSprites()
88.     update()
89. repeat
```

-

Comme vous pouvez le voir, cette partie s'insère juste avant les 4 dernières lignes de la **boucle**.

Nous utilisons la **fonction** {code:getSpriteAnimFrame()} pour connaître l'image active de l'animation de notre explosion. Comme nous savons que la dernière image de l'animation de l'explosion est l'image {code:89}, nous disposons d'un moyen simple pour vérifier si elle est terminée.

Nous utilison la **fonction** {code:setSpriteVisibility()} une dernière fois pour masquer le sprite (visibilité à {code:false}) une fois la dernière image du cycle atteinte.

Terminé !

Félicitations pour être parvenu au terme de ce tutoriel ! Maintenant vous pouvez personnaliser le jeu à votre guise. Si malencontreusement vous cassez quelque chose, vous pouvez trouver une version complète du programme ci-dessous. N'hésitez pas à copier et coller tout le code ci-dessous dans un nouveau fichier de projet:

```
ecran = { gwidth(), gheight() }
echelle = { ecran.y / 270, ecran.y / 270 }

joueurSpr = createSprite()
joueurImg = loadImage( "Untied Games/joueur ships", false )
setSpriteImage( joueurSpr, joueurImg )
joueurPos = { ecran.x / 20, ecran.y / 2 }
joueurVel = { 0, 0 }

setSpriteAnimation( joueurSpr, 0, 0, 0 )
setSpriteechelle( joueurSpr, echelle )
setSpriteLocation( joueurSpr, joueurPos )

rocImgs = [
    loadImage( "Untied Games/Asteroid A", false ),
    loadImage( "Untied Games/Asteroid B", false ),
    loadImage( "Untied Games/Asteroid C", false ),
    loadImage( "Untied Games/Asteroid D", false )
]

array rocs[50] = [
    .spr = 0,
    .pos = { 0, 0 },
    .vel = { 0, 0 }
]
```

```
 26.
 27. for i = 0 to len( rocs ) loop
 28.     n = random( 4 )
 29.     rocs[i].spr = createSprite()
 30.     setSpriteImage( rocs[i].spr, rocImgs[n] )
 31.     rocs[i].pos = { ecran.x + random( ecran.x * 2 ), random(
ecran.y ) }
 32.     rocs[i].vel = { - random( ecran.x / 5 ), random( 32 ) -
16 }
 33.     setSpriteAnimation( rocs[i].spr, 0, 39, 10 )
 34.     setSpriteLocation( rocs[i].spr, rocs[i].pos )
 35. repeat
 36.
 37. expSpr = createSprite()
 38. expImg = loadImage( "Untied Games/Explosion 09", false )
 39. setSpriteImage( expSpr, expImg )
 40. setSpriteVisibility( expSpr, false )
 41. vivant = true
 42.
 43. loop
 44.     clear()
 45.     ecran = { gwidth(), gheight() }
 46.     echelle = { ecran.y / 270, ecran.y / 270 }
 47.
 48.     c = controls( 0 )
 49.
 50.     joueurVel = { c.lx * ecran.y / 4, ecran.y / 3 - c.a *
ecran.y / 1.5 }
 51.
 52.     joueurPos = getSpriteLocation( joueurSpr )
 53.
 54.     joueurPos.x = clamp( joueurPos.x, ecran.x / 10, ecran.x -
ecran.x / 10 )
 55.     joueurPos.y = clamp( joueurPos.y, ecran.y / 10, ecran.y -
ecran.y / 10 )
 56.
 57.     setSpriteLocation( joueurSpr, joueurPos )
 58.     setSpriteSpeed( joueurSpr, joueurVel )
 59.     setSpriteechelle( joueurSpr, echelle )
 60.
 61.     for i = 0 to len( rocs ) loop
 62.         setSpriteechelle( rocs[i].spr, echelle )
 63.         setSpriteSpeed( rocs[i].spr, rocs[i].vel )
 64.         rocs[i].pos = getSpriteLocation( rocs[i].spr )
 65.         rocTaille = getSpriteSize( rocs[i].spr )
 66.
 67.         if rocs[i].pos.x + rocTaille.x / 2 < 0 then
 68.             rocs[i].pos = { ecran.x + random( ecran.x ),
```

-

```
random( ecran.y ) }
 69.             setSpriteLocation( rocs[i].spr, rocs[i].pos )
 70.         endif
 71.
 72.         if distance( joueurPos, rocs[i].pos ) < rocTaille.x /
2 - 50 and vivant then
 73.             vivant = false
 74.             setSpriteAnimation( expSpr, 0, 89, 14 )
 75.             setSpriteLocation( expSpr, joueurPos )
 76.             setSpriteechelle( expSpr, echelle )
 77.             setSpriteVisibility( expSpr, true )
 78.             setSpriteVisibility( joueurSpr, false )
 79.         endif
 80.     repeat
 81.
 82.     if getSpriteAnimFrame( expSpr ) >= 89 then
 83.         setSpriteVisibility( expSpr, false )
 84.     endif
 85.
 86.     updateSprites()
 87.     drawSprites()
 88.     update()
 89. repeat
```

[clamp()](../Command Reference/Arithmetic/clamp.md), [clear()](../Command Reference/Screen Display/clear.md), [controls()](../Command Reference/Input/controls.md), [createSprite()](../Command Reference/2D Graphics/createSprite.md), [distance()](../Command Reference/Arithmetic/distance.md), [drawSprites()](../Command Reference/2D Graphics/drawSprites.md), else, endIf, for, [getSpriteAnimFrame()](../Command Reference/2D Graphics/getSpriteAnimFrame.md), [getSpriteSize()](../Command Reference/2D Graphics/getSpriteSize.md), if, loop, repeat, [setSpriteAnimation](../Command Reference/2D Graphics/setSpriteAnimation.md), setSpriteLocation() (../Command Reference/2D Graphics/setSpriteLocation.md), [setSpriteImage()](../Command Reference/2D Graphics/setSpriteImage.md), [setSpriteScale](../Command Reference/2D Graphics/setSpriteScale.md), setSpriteSpeed() (../Command Reference/2D Graphics/setSpriteSpeed.md), [setSpriteVisibility()](../Command Reference/2D Graphics/setSpriteVisibility.md), then, to, [update()](../Command Reference/Screen Display/update.md), [updateSprites()](../Command Reference/2D Graphics/updateSprites.md)

Mots-Clefs

-

and

Objet
Joindre deux conditions ensemble.

Description
La condition résultante est vraie si les deux conditions sont vraies.

Syntaxe

```
if condition1 and condition2 then ... endIf // ... est exécuté
UNIQUEMENT si les deux conditions sont remplies
```

Arguments
condition1 première condition

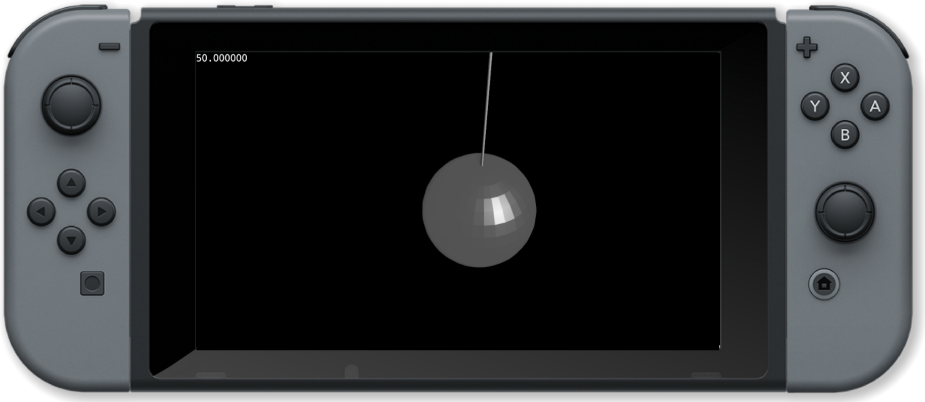
condition2 seconde condition

Exemple

```
setCamera( { 0, 10, 10 }, { 0, 0, 0 } )
luminosite = 50
lumiere = worldLight( { -5, -5, -5 }, white, luminosite )
lumiere_active = true
modele_de_balle = loadModel( "Kat Deak/Discoball" )
balle = placeObject( modele_de_balle, { 0, 0, 0 }, { 10, 10, 10 }
)
loop
    c = controls( 0 )
    if c.x and !lumiere_active then
        lumiere = worldlight( { -5, -5, -5 }, white, luminosite )
        lumiere_active = true
    endIf
    if c.a and lumiere_active then
        removeLight( lumiere )
        lumiere_active = false
    endIf
    rotateObject( balle, { 0, 1, 0 }, 1.0 )
    drawObjects()
```

-

```
    printAt( 0, 0, "Pressez X pour allumer la lumiere." )
    printAt( 0, 1, "Pressez A pour eteindre la lumiere." )
    update()
repeat
```



Commandes associées

and, else, endIf, if, or, then

array

Objet
Créer un tableau d'entiers.

Description
Définir un tableau de variables. Peut être utilisé dans une définition de structure ou de manière autonome. Par défaut, les éléments du tableau sont de type entier.

Syntaxe

```
struct nom
  array champ1
  ...
  typen champn
endStruct
```

-

Arguments
nom nom de la structure

champ1 nom du premier champ

champn nom du dernier champ

typen type du dernier champ

Exemple

```
// Définir une propriété d'un type de structure en tant que
tableau
struct personne
    string nom
    int    age
    float  taille
    array  interets[3]
endStruct

// Définir un tableau entier de 10 éléments
array data[10]
```

Commandes associées

array, int, float, endStruct, string, struct, vector

break

Objet
Sortir d'une boucle immédiatement.

Description
Empêcher une boucle de se répéter avant que la condition de sortie ne soit remplie.

Syntaxe

-

```
while condition loop ... break ... repeat // Boucle tant que la
condition est vraie ou que BREAK est exécuté
```

Arguments
condition condition booléenne qui interrompt la boucle quand elle est fausse

Exemple

```
loop
    c = controls( 0 )
    printAt( 0,0, "Pressez A pour quitter le programme." )
    if c.a then
        break
    endIf
    update()
repeat
```

Commandes associées

for, repeat, step, to, while

else

Objet
Exécuter un bloc de code lorsque la condition est fausse.

Description
Utiliser pour exécuter un bloc de code si la condition de l'alternative "Si..." n'est pas remplie.

Syntaxe

```
if condition then ... else ... endIf // si la condition est
remplie executer en premier ... sinon executer en second ...
```

Arguments

-

condition condition à tester. Cela peut être une condition composée en utilisant AND et OR.

Exemple

```
limite = 5
y = 0

for i = 0 to 11 loop
    if i < limite then
        printAt( 0, y, "Le nombre ", i, " est plus petit que ",
limite )
    else
        if i == limite then
            printAt( 0, y, "Le nombre ", i, " est equale a ",
limite )
        else
            printAt( 0, y, "Le nombre ", i, " est plus grand que
", limite )
        endif
    endif
repeat

update()
sleep( 5 )
```

Commandes associées

and, else, endIf, if, or, then

endIf

Objet
Marquer la fin d'un bloc de code conditionnel.

Description
Ceci termine l'instruction if conditionnelle et retourne à une exécution inconditionnelle.

-

Syntaxe

```
if condition then ... else ... endIf // si la condition est
remplie executer en premier ... sinon executer en second ...
```

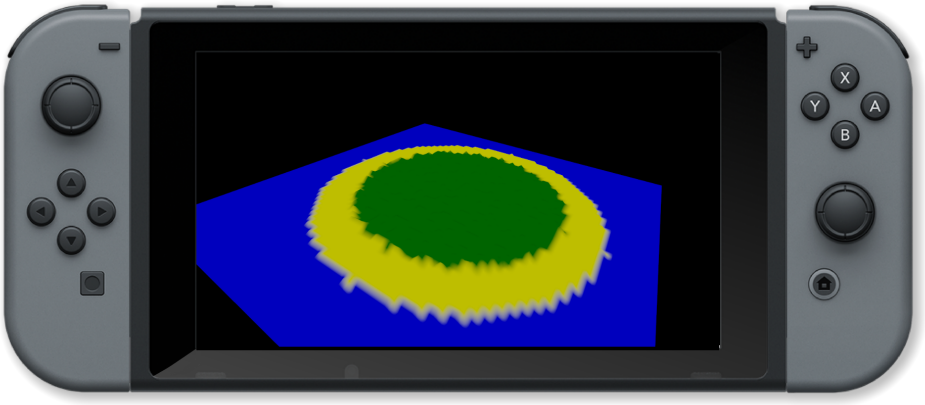
Arguments

condition condition à tester. Cela peut être une condition composée en utilisant AND et OR.

Exemple

```
taille_grille = 64
paysage = createterrain( taille_grille, 1 )
hauteur = 0
couleur = white
for x = 0 to taille_grille loop
    for y = 0 to taille_grille loop
        d = distance( { x, y }, { taille_grille / 2, taille_grille
/ 2 } )
        if d > 24 then    // sea level
            hauteur = 0
            couleur = blue
        else
            if d > 18 then  // plage
                hauteur = 1
                couleur = yellow
            else            // collines
                hauteur = rnd( 2 ) + 1
                couleur = green
            endIf
        endIf
        setTerrainPoint( paysage, x, y, hauteur, couleur )
    repeat
repeat
setCamera( { taille_grille / 2, 50, taille_grille / 2 }, {
taille_grille / 2.0, 0, taille_grille / 2.00001 } )
setAmbientlight( { 0.5, 0.5, 0.5 } )
ile = placeObject( paysage, { taille_grille / 2, 0, taille_grille
/ 2 }, { 1, 1, 1 } )
loop
    c = controls( 0 ) // rotate using joysticks
    rotateObject( ile, { 1, 0, 0 }, c.ly )
    rotateObject( ile, { 0, 0, 1 }, c.lx )
    rotateObject( ile, { 0, 1, 0 }, c.rx )
    drawObjects()
    update()
repeat
```

-



Commandes associées

and, else, if, or, then

endStruct

Objet
Terminer une définition de variable structurée

Description
Marque la fin d'une définition de variable de type structure.

Syntaxe

```
struct nom
  type1 champ1
  ...
  typen champn
endStruct
```

Arguments
nom nom de la structure

-

champ1 nom du premier champ

type1 type du premier champ

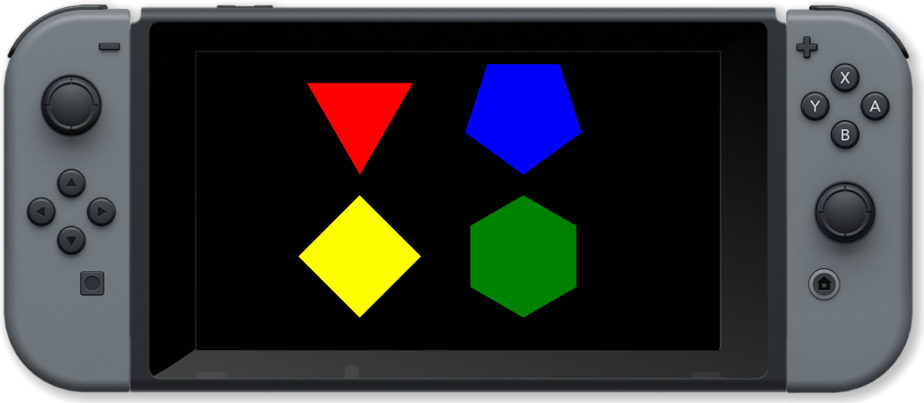
champn nom du dernier champ

typen type du dernier champ

Exemple

```
struct forme
    string nom
    int cotes
    int taille
    vector pos
    int coul
endStruct

forme formes[3]
formes[0] = [ .nom = "triangle", .cotes=3, .taille=150, .pos = {
400, 150 }, .coul = red ]
formes[1] = [ .nom = "square",   .cotes=4, .taille=150, .pos = {
400, 500 }, .coul = yellow ]
formes[2] = [ .nom = "pentagon", .cotes=5, .taille=150, .pos = {
800, 150 }, .coul = blue ]
formes[3] = [ .nom = "hexagon",  .cotes=6, .taille=150, .pos = {
800, 500 }, .coul = green ]

loop
    clear()
    printAt( 0, 0, "Pressez A pour voir les libelles." )
    c = controls( 0 )
    for i = 0 to 4 loop
        drawShape( formes[i], c.a )
    repeat
    update()
repeat

function drawShape( s, afficher_libelle )
    circle( s.pos.x, s.pos.y, s.taille, s.cotes, s.coul, 0 )
    if afficher_libelle then
        drawText( s.pos.x - s.size/2, s.pos.y, s.size / 5, black,
s.nom )
    endIf
return void
```

-



Commandes associées

array, int, float, string, struct, vector

float

Objet
Initialiser une variable à virgule flottante.

Description
Définir une variable comme étant de type float (virgule flottante).

Syntaxe

```
struct nom
  float champ1
  ...
  typen champn
endStruct
```

Arguments
nom nom de la structure

champ1 nom du premier champ

-

champn nom du dernier champ

typen type du dernier champ

Exemple

```
// Définir une variable float dans une définition de structure
struct personne
    string nom
    int    age
    float  taille
    array  interets[3]
endStruct

// Initialiser un tableau de nombre à virgule flottante de dix
éléments
```

Commandes associées

array, int, float, endStruct, string, struct, vector

for

Objet
Répéter une section de code plusieurs fois.

Description
La répétition des instructions dans la boucle est contrôlée par la variable d'index qui varie de la valeur initiale à la valeur finale moins l'incrément.

Syntaxe

```
for index = debut to fin loop ... repeat // Boucle sur les valeurs

for index = debut to fin step pas loop ... repeat // Boucle sur
les valeurs avec un pas
```

-

Arguments
index variable d'index de la boucle

debut valeur de départ de l'index

fin valeur de fin de l'index (la boucle n'est pas exécutée pour cette valeur)

pas montant pour changer la variable d'index (1 par défaut)

Exemple

```
// Dessine 100 boîtes au hasard
clear()
for i = 1 to 100 loop
    // Choisir une couleur au hasard
    couleur = { random( 101 ) / 100, random( 101 ) / 100, random(
101 ) / 100, random( 101 ) / 100 }
    x = random( gWidth() )
    y = random( gHeight() )
    largeur = random( gWidth() / 4 )
    hauteur = random( gHeight() / 4 )
    contour = random( 2 )
    box( x, y, largeur, hauteur, couleur, contour )
    update()
repeat
// Wait 3 seconds
sleep( 3 )
```

Commandes associées

for, repeat, step, to, while

-

function

Objet
Créer une fonction définie par l'utilisateur.

Description
Permet à l'utilisateur de créer ses propres fonctions. Cela permet de réutiliser le code et facilite sa lecture et sa maintenance.

Syntaxe

```
function nom() ... return valeur // fonction sans arguments

function nom(argument1, ... argumentn) ... return valeur //
fonction avec n arguments
```

Arguments
nom nom de la fonction

argument1 premier paramètre de la fonction

argumentn dernier paramètre de la fonction

valeur valeur de retour de la fonction (void si aucune valeur n'est retournée)

Exemple

```
for taille = 1 to 200 step 1 loop
    clear()
    centreTexte( "Bonjour le monde !", taille )
    update()
repeat

// Centre une chaine texte a l'ecran
function centreTexte( message, taille )
    textSize( taille )
    tw = textWidth( message )
    drawText( ( gWidth() - tw ) / 2, ( gHeight() - taille ) / 2,
```

-

```
taille, white, message )
return void
```

Commandes associées

function, return, void

if

Object
Exécute conditionnellement un bloc de code lorsque la condition est vrai.

Description
Exécute un bloc de code que si la condition spécifiée est vraie (1).

Syntaxe

```
if condition then ... endIf // ... est execute SEULEMENT si la
condition est rempli

if condition then ... else ... endIf // si la condition est
remplie executer en premier ... sinon executer en second ...
```

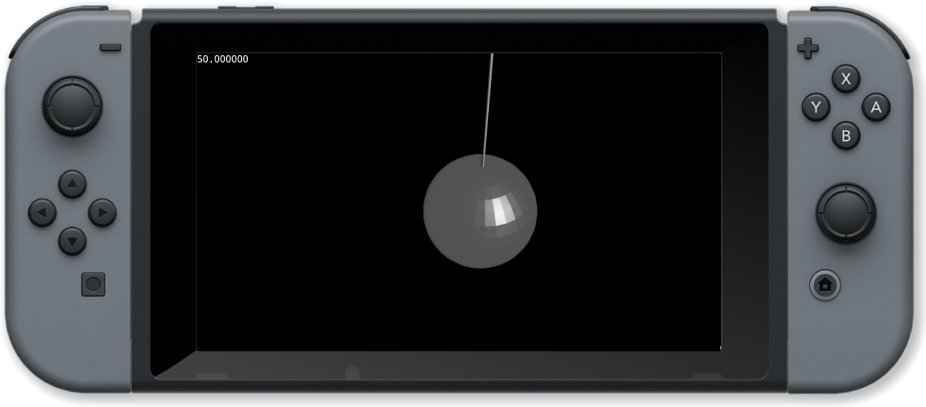
Arguments

-

condition condition à tester. Cela peut être une condition composée en utilisant AND et OR.

Example

```
setCamera( { 0, 10, 10 }, { 0, 0, 0 } )
luminosite = 50
lumiere = worldLight( { -5, -5, -5 }, white, luminosite )
lumiere_active = true
modele_de_balle = loadModel( "Kat Deak/Discoball" )
balle = placeObject( modele_de_balle, { 0, 0, 0 }, { 10, 10, 10 }
)
loop
    c = controls( 0 )
    if c.x and !lumiere_active then
        lumiere = worldlight( { -5, -5, -5 }, white, luminosite )
        lumiere_active = true
    endIf
    if c.a and lumiere_active then
        removeLight( lumiere )
        lumiere_active = false
    endIf
    rotateObject( balle, { 0, 1, 0 }, 1.0 )
    drawObjects()
    printAt( 0, 0, "Pressez X pour allumer la lumiere." )
    printAt( 0, 1, "Pressez A pour eteindre la lumiere." )
    update()
repeat
```



Commandes associées

and, else, endIf, if, or, then

-

int

Object
Créer une variable entière.

Description
Crée une variable du type entier. Peut également être utilisé comme définition de tableau et dans les définitions de structure.

Syntax

```
struct nom
  int champ1
  ...
  typen champn
endStruct
```

Arguments
nom nom de la structure

champ1 nom du premier champ

champn nom du dernier champ

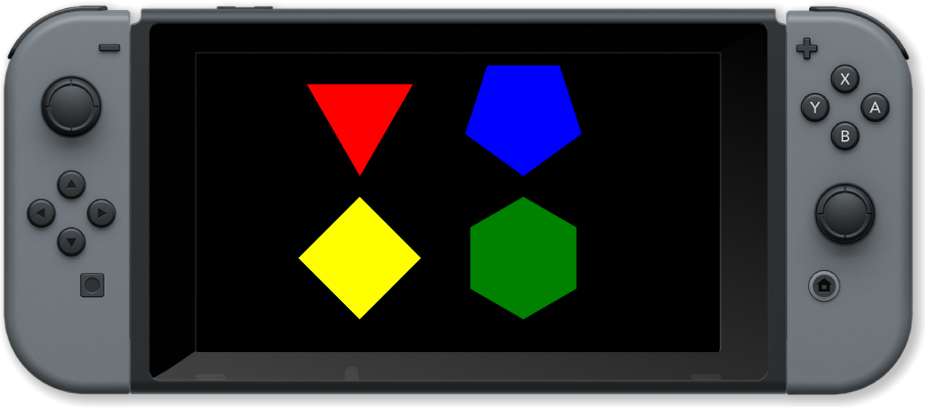
typen type du dernier champ

Exemple

```
struct forme
    string nom
    int cotes
    int taille
    vector pos
    int coul
endStruct

forme formes[3]
formes[0] = [ .nom = "triangle", .cotes=3, .taille=150, .pos = {
```

-

```
400, 150 }, .coul = red ]
formes[1] = [ .nom = "square",   .cotes=4, .taille=150, .pos = {
400, 500 }, .coul = yellow ]
formes[2] = [ .nom = "pentagon", .cotes=5, .taille=150, .pos = {
800, 150 }, .coul = blue ]
formes[3] = [ .nom = "hexagon",  .cotes=6, .taille=150, .pos = {
800, 500 }, .coul = green ]

loop
    clear()
    printAt( 0, 0, "Pressez A pour voir les libelles." )
    c = controls( 0 )
    for i = 0 to 4 loop
        drawShape( formes[i], c.a )
    repeat
    update()
repeat

function drawShape( s, afficher_libelle )
    circle( s.pos.x, s.pos.y, s.taille, s.cotes, s.coul, 0 )
    if afficher_libelle then
        drawText( s.pos.x - s.size/2, s.pos.y, s.size / 5, black,
s.nom )
    endIf
return void
```



Commandes associées

array, int, float, endStruct, string, struct, vector

-

loop

Objet
Répéter une section de code.

Description
Répéte une section de code un nombre spécifié de fois ou jusqu'à ce qu'une condition soit remplie (ou pour toujours).

Syntaxe

```
loop ... repeat // Boucle sans fin

while condition loop ... repeat // Boucle tant que la condition
est remplie (true)

for index = start to end loop ... repeat // Boucle sur les valeurs

for index = start to end step pas loop ... repeat // Boucle sur
les valeurs avec un pas
```

Arguments
condition condition booléenne qui arrête la boucle quand elle est fausse

index variable index de la boucle

start valeur de départ de l'index

end valeur de fin de l'index (la boucle n'est pas exécutée pour cette valeur)

pas pas pour changer la variable d'index (1 par défaut)

Exemple

```
// Dessine 100 boîtes au hasard
clear()
for i = 1 to 100 loop
    // Choisir une couleur au hasard
    couleur = { random( 101 ) / 100, random( 101 ) / 100, random(
101 ) / 100, random( 101 ) / 100 }
    x = random( gWidth() )
    y = random( gHeight() )
    largeur = random( gWidth() / 4 )
```

-

```
    hauteur = random( gHeight() / 4 )
    contour = random( 2 )
    box( x, y, largeur, hauteur, couleur, contour )
    update()
repeat
// Wait 3 seconds
sleep( 3 )
```

Commandes associées

for, repeat, step, to, while

not

Objet
Nie une condition.

Description
Exécute un bloc de code que si la condition spécifiée est fausse (0).

Syntaxe

```
if not condition then ... endIf // ... is executed ONLY if
condition is not met
```

-

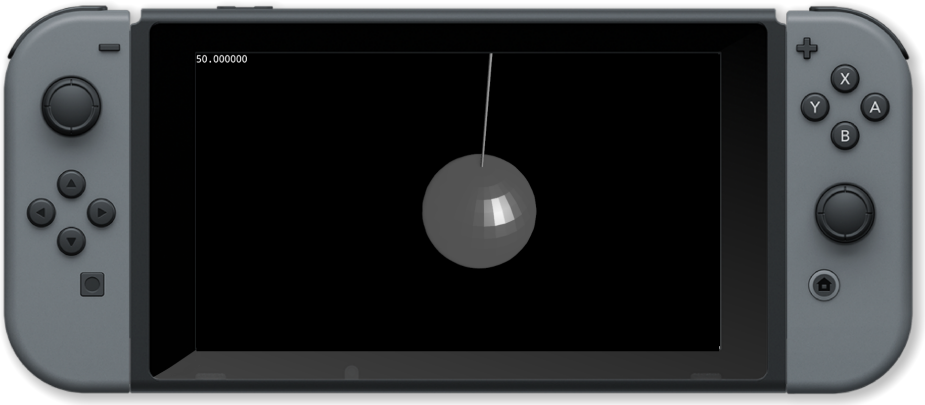
```
if not condition then ... else ... endIf // if condition is not
met execute first ... otherwise execute second ...
```

Arguments

condition condition to be tested. This can be a compound condition using AND and OR

Exemple

```
setCamera( { 0, 10, 10 }, { 0, 0, 0 } )
luminosite = 50
lumiere = worldLight( { -5, -5, -5 }, white, luminosite )
lumiere_active = true
modele_de_balle = loadModel( "Kat Deak/Discoball" )
balle = placeObject( modele_de_balle, { 0, 0, 0 }, { 10, 10, 10 }
)
loop
    c = controls( 0 )
    if c.x and !lumiere_active then
        lumiere = worldlight( { -5, -5, -5 }, white, luminosite )
        lumiere_active = true
    endIf
    if c.a and lumiere_active then
        removeLight( lumiere )
        lumiere_active = false
    endIf
    rotateObject( balle, { 0, 1, 0 }, 1.0 )
    drawObjects()
    printAt( 0, 0, "Pressez X pour allumer la lumiere." )
    printAt( 0, 1, "Pressez A pour eteindre la lumiere." )
    update()
repeat
```



-

Commandes associées

and, else, endIf, if, or, then

or

Objet
Spécifier une condition alternative

Description
La condition résultante est vraie si l'une de ces conditions est vraie.

Syntaxe

```
if condition1 or condition2 then ... endIf // ... est execute si
l'une ou l'autre des conditions est remplie
```

Arguments
condition1 première condition

condition2 seconde condition

Exemple

```
image = loadImage( "Untied Games/Enemy small top C", false )
vaisseau = createSprite()
setSpriteImage( vaisseau, image )
position = { gWidth() / 2, gHeight() / 2 }
setSpriteLocation( vaisseau, position )
setSpriteScale( vaisseau, { 20, 20 } )
rv = -0.5
gv = 0.5
bv = 0

loop
    clear()
    sc = getSpriteColour( vaisseau )
    if sc.r > 1 or sc.r < 0 then
```

-

```
        rv = -rv
    endIf
    if sc.b > 1 or sc.b < 0 then
        gv = -gv
    endIf
    setSpriteColourSpeed( vaisseau, { rv, gv, bv, 0 } )
    updateSprites()
    drawSprites()
    update()
repeat
```

Commandes associées

and, else, endIf, if, or, then

repeat

Objet
Fin d'une boucle ou d'une section de code répétée

Description
Marque la fin d'une section de code à répéter. Le contrôle est renvoyé au mot-clé LOOP précédent. La boucle est répétée un nombre spécifié de fois ou jusqu'à ce qu'une condition soit remplie (ou pour toujours).

-

Syntaxe

```
loop ... repeat // Boucle sans fin

while condition loop ... repeat // Boucle tant que la condition
est remplie (true)

for index = start to end loop ... repeat // Boucle sur les valeurs

for index = start to end step pas loop ... repeat // Boucle sur
les valeurs avec un pas
```

Arguments
condition condition booléenne qui arrête la boucle quand elle est fausse

index variable index de la boucle

start valeur de départ de l'index

end valeur de fin de l'index (la boucle n'est pas exécutée pour cette valeur)

pas pas pour changer la variable d'index (1 par défaut)

Exemple

```
// Dessine 100 boîtes au hasard
clear()
for i = 1 to 100 loop
    // Choisir une couleur au hasard
    couleur = { random( 101 ) / 100, random( 101 ) / 100, random(
101 ) / 100, random( 101 ) / 100 }
    x = random( gWidth() )
    y = random( gHeight() )
    largeur = random( gWidth() / 4 )
    hauteur = random( gHeight() / 4 )
    contour = random( 2 )
    box( x, y, largeur, hauteur, couleur, contour )
    update()
repeat
// Wait 3 seconds
sleep( 3 )
```

-

Commandes associées

for, repeat, step, to, while

return

Objet
Renvoyer une valeur à partir d'une fonction définie par l'utilisateur.

Description
Renvoie une valeur d'une fonction et reprend l'exécution à partir du point où la fonction a été appelée.

Syntaxe

```
function nom() ... return valeur // fonction sans arguments

function nom(argument1, ... argumentn) ... return valeur //
fonction avec n arguments
```

Arguments
nom nom de la fonction

argument1 premier paramètre de la fonction

-

argumentn dernier paramètre de la fonction

value valeur de retour de la fonction (void si aucune valeur n'est retournée)

Exemple

```
y = 0
for i = 1 to 11 loop
    carre = calculeCarre( i )
    printAt( 0, y, carre )
    y += 1
repeat

update()
sleep( 3 )

function calculeCarre( nombre )
    nombre *= nombre
return nombre
```

Commandes associées

function, return, void

step

-

Objet
Spécifie la valeur d'incrément de la variable d'index de boucle

Description
La boucle est exécutée jusqu'à ce que la valeur de la variable d'index de la boucle passe de la valeur initiale à l'étape juste avant la valeur finale, par incréments d'étape.

Syntaxe

```
for index = start to end step pas loop ... repeat // Boucle sur
les valeurs avec un pas
```

Arguments
index variable d'index de la boucle

debut valeur de départ de l'index

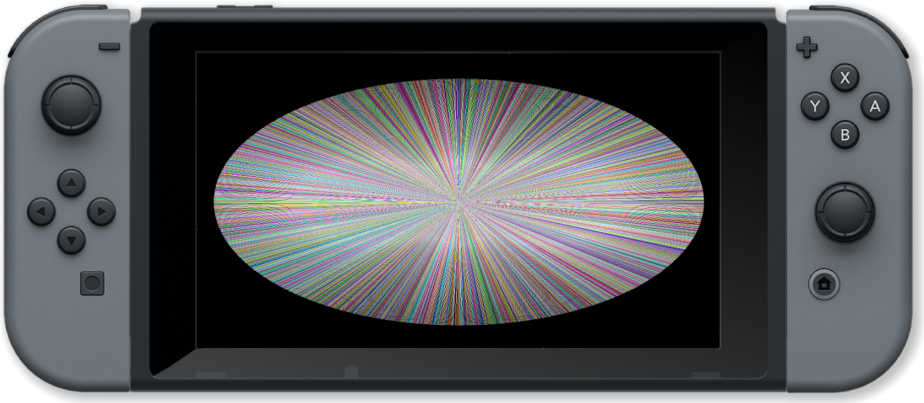
fin valeur de fin de l'index (la boucle n'est pas exécutée pour cette valeur)

pas montant pour changer la variable d'index (1 par défaut)

Exemple

```
clear()
radians( true )
centre = { gwidth() / 2, gHeight() / 2 }
for angle = 0 to 2 * pi step 0.005 loop
    couleur = { random( 101 ) / 100, random( 101 ) / 100, random(
101 ) / 100, 1.0 }
    resultat = sinCos( angle )
    point = { 600 * resultat.y + centre.x, 300 * resultat.x +
centre.y }
    line( centre, point, couleur )
    repeat
update()
sleep( 3 )
```

-



Commandes associées

for, repeat, step, to, while

string

Objet
Créer une variable de chaîne.

Description
Initialise une variable du type chaîne. Peut être utilisé pour initialiser un tableau ou dans une définition de structure.

Syntaxe

```
struct nom
  string champ1
  ...
  typen champn
endStruct
```

Arguments
nom nom de la structure

-

champ1 nom du premier champ

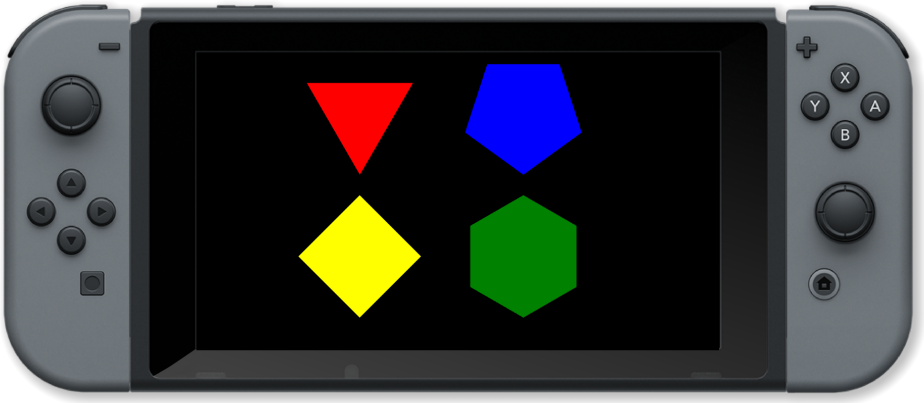
champn nom du dernier champ

typen type du dernier champ

Exemple

```
struct forme
    string nom
    int cotes
    int taille
    vector pos
    int coul
endStruct

forme formes[3]
formes[0] = [ .nom = "triangle", .cotes=3, .taille=150, .pos = {
400, 150 }, .coul = red ]
formes[1] = [ .nom = "square",   .cotes=4, .taille=150, .pos = {
400, 500 }, .coul = yellow ]
formes[2] = [ .nom = "pentagon", .cotes=5, .taille=150, .pos = {
800, 150 }, .coul = blue ]
formes[3] = [ .nom = "hexagon",  .cotes=6, .taille=150, .pos = {
800, 500 }, .coul = green ]

loop
    clear()
    printAt( 0, 0, "Pressez A pour voir les libelles." )
    c = controls( 0 )
    for i = 0 to 4 loop
        drawShape( formes[i], c.a )
    repeat
    update()
repeat

function drawShape( s, afficher_libelle )
    circle( s.pos.x, s.pos.y, s.taille, s.cotes, s.coul, 0 )
    if afficher_libelle then
        drawText( s.pos.x - s.size/2, s.pos.y, s.size / 5, black,
s.nom )
    endIf
return void
```

-



Commandes associées

array, int, float, endStruct, string, struct, vector

struct

Objet
Créer un type de variable structurée.

Description
Permet à l'utilisateur de créer ses propres types de variables complexes. Cela permet de regrouper des informations liées dans une seule variable.

Syntaxe

```
struct nom
  type1 field1
  ...
  typen fieldn
endStruct
```

Arguments
nom nom de la structure

-

field1 nom du premier champ

type1 type du premier champ

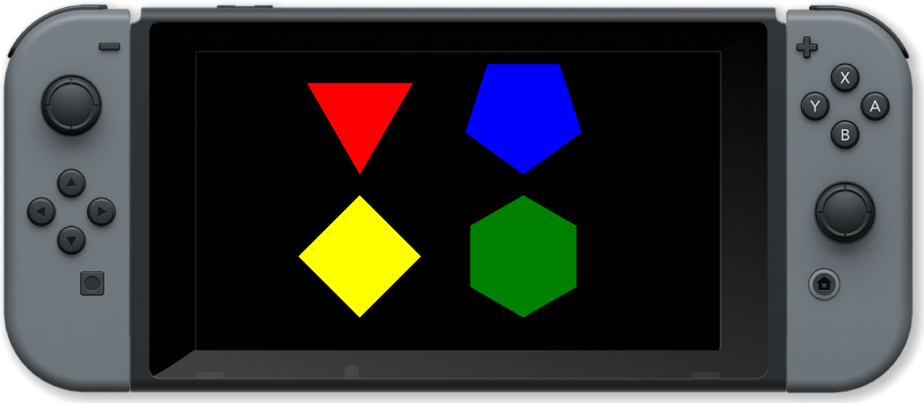
fieldn nom du dernier champ

typen type du dernier champ

Exemple

```
struct forme
    string nom
    int cotes
    int taille
    vector pos
    int coul
endStruct

forme formes[3]
formes[0] = [ .nom = "triangle", .cotes=3, .taille=150, .pos = {
400, 150 }, .coul = red ]
formes[1] = [ .nom = "square",   .cotes=4, .taille=150, .pos = {
400, 500 }, .coul = yellow ]
formes[2] = [ .nom = "pentagon", .cotes=5, .taille=150, .pos = {
800, 150 }, .coul = blue ]
formes[3] = [ .nom = "hexagon",  .cotes=6, .taille=150, .pos = {
800, 500 }, .coul = green ]

loop
    clear()
    printAt( 0, 0, "Pressez A pour voir les libelles." )
    c = controls( 0 )
    for i = 0 to 4 loop
        drawShape( formes[i], c.a )
    repeat
    update()
repeat

function drawShape( s, afficher_libelle )
    circle( s.pos.x, s.pos.y, s.taille, s.cotes, s.coul, 0 )
    if afficher_libelle then
        drawText( s.pos.x - s.size/2, s.pos.y, s.size / 5, black,
s.nom )
    endIf
return void
```

-



Commandes associées

array, int, float, endStruct, string, struct, vector

then

Purpose
Marque la fin d'une condition et le début d'un bloc de code conditionnel.

Description
Instruction après laquelle l'exécution se poursuit jusqu'à une instruction ELSE or endIf si la condition est remplie.

Syntax

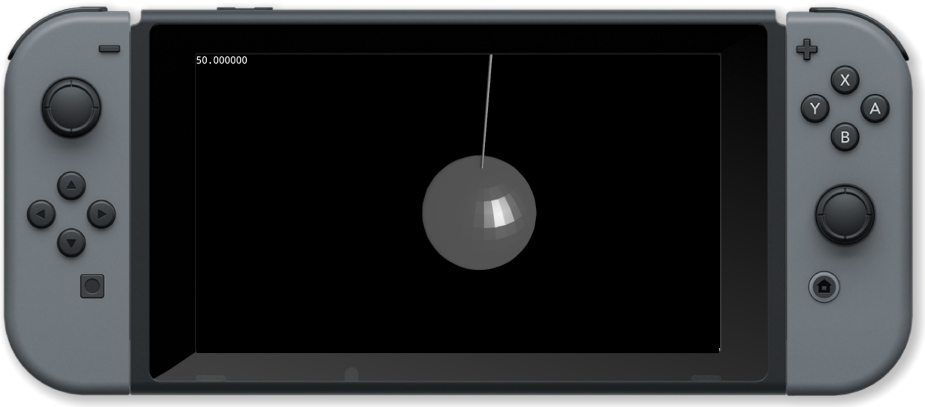
```
if condition then ... endIf // ... est execute SEULEMENT si la
condition est rempli

if condition then ... else ... endIf // si la condition est
remplie executer en premier ... sinon executer en second ...
```

Arguments
condition condition à tester. Cela peut être une condition composée en utilisant AND et OR.

-

Exemple

```
setCamera( { 0, 10, 10 }, { 0, 0, 0 } )
luminosite = 50
lumiere = worldLight( { -5, -5, -5 }, white, luminosite )
lumiere_active = true
modele_de_balle = loadModel( "Kat Deak/Discoball" )
balle = placeObject( modele_de_balle, { 0, 0, 0 }, { 10, 10, 10 }
)
loop
    c = controls( 0 )
    if c.x and !lumiere_active then
        lumiere = worldlight( { -5, -5, -5 }, white, luminosite )
        lumiere_active = true
    endIf
    if c.a and lumiere_active then
        removeLight( lumiere )
        lumiere_active = false
    endIf
    rotateObject( balle, { 0, 1, 0 }, 1.0 )
    drawObjects()
    printAt( 0, 0, "Pressez X pour allumer la lumiere." )
    printAt( 0, 1, "Pressez A pour eteindre la lumiere." )
    update()
repeat
```



Commandes associées

and, else, endIf, if, or, then

MOTS-CLES

-

to

Purpose
Sépare les valeurs de début et de fin dans une boucle FOR.

Description
La valeur située avant (le “to”) est la valeur de début de la variable d’index et celle située aprèsest la valeur de fin. La répétition des instructions dans la boucle est contrôlée par la variable d’index qui varie de la valeur initiale à la valeur finale moins l’incrément.

Syntaxe

```
for index = debut to fin loop ... repeat // Boucle sur les valeurs

for index = debut to fin step pas loop ... repeat // Boucle sur
les valeurs avec un pas
```

Arguments
index variable d’index de la boucle

debut valeur de départ de l’index

fin valeur de fin de l’index (la boucle n’est pas exécutée pour cette valeur)

pas montant pour changer la variable d’index (1 par défaut)

Exemple

```
// Dessine 100 boîtes au hasard
clear()
for i = 1 to 100 loop
    // Choisir une couleur au hasard
    couleur = { random( 101 ) / 100, random( 101 ) / 100, random(
101 ) / 100, random( 101 ) / 100 }
    x = random( gWidth() )
    y = random( gHeight() )
    largeur = random( gWidth() / 4 )
    hauteur = random( gHeight() / 4 )
    contour = random( 2 )
    box( x, y, largeur, hauteur, couleur, contour )
    update()
repeat
```

-

```
// Wait 3 seconds
sleep( 3 )
```

Commandes associées

for, repeat, step, to, while

vector

Objet
Créer une variable du type vecteur.

Description
Initialise une variable du type vecteur. Peut être utilisé pour initialiser un tableau ou dans une définition de structure. Le vecteur peut avoir jusqu'à 4 dimensions (x, y, z et w / r, g, b et a).

Syntaxe

```
struct nom
  vector champ1
  ...
  typen champn
endStruct
```

-

Arguments
name nom de la structure

champ1 nom du premier champ

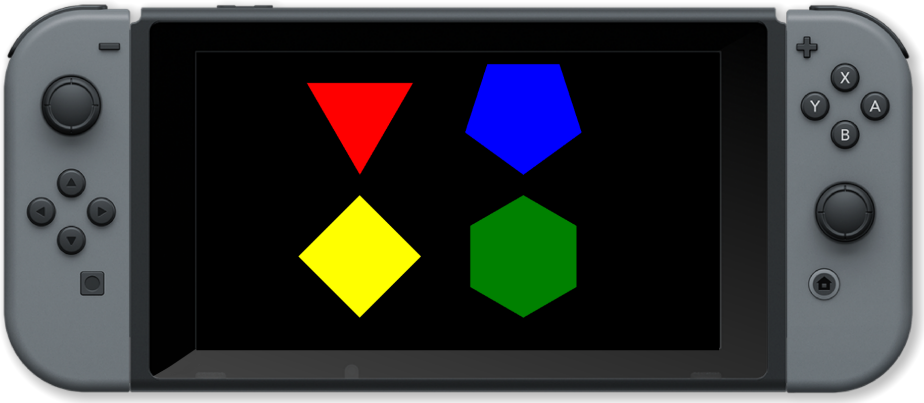
champn nom du dernier champ

typen type du dernier champ

Exemple

```
struct forme
    string nom
    int cotes
    int taille
    vector pos
    int coul
endStruct

forme formes[3]
formes[0] = [ .nom = "triangle", .cotes=3, .taille=150, .pos = {
400, 150 }, .coul = red ]
formes[1] = [ .nom = "square",   .cotes=4, .taille=150, .pos = {
400, 500 }, .coul = yellow ]
formes[2] = [ .nom = "pentagon", .cotes=5, .taille=150, .pos = {
800, 150 }, .coul = blue ]
formes[3] = [ .nom = "hexagon",  .cotes=6, .taille=150, .pos = {
800, 500 }, .coul = green ]

loop
    clear()
    printAt( 0, 0, "Pressez A pour voir les libelles." )
    c = controls( 0 )
    for i = 0 to 4 loop
        drawShape( formes[i], c.a )
    repeat
    update()
repeat

function drawShape( s, afficher_libelle )
    circle( s.pos.x, s.pos.y, s.taille, s.cotes, s.coul, 0 )
    if afficher_libelle then
        drawText( s.pos.x - s.size/2, s.pos.y, s.size / 5, black,
s.nom )
    endIf
return void
```

-



Commandes associées

array, int, float, endStruct, string, struct, vector

void

Objet
Une valeur qui indique qu'une fonction définie par l'utilisateur ne renvoie aucune valeur

Description
Une valeur spéciale qui n'a en réalité "aucune valeur".

Syntaxe

```
function nom() ... return valeur // fonction sans arguments

function nom(argument1, ... argumentn) ... return valeur //
fonction avec n arguments
```

Arguments
nom nom de la fonction

argument1 premier paramètre de la fonction

-

argumentn dernier paramètre de la fonction

Exemple

```
for taille = 1 to 200 step 1 loop
    clear()
    centreTexte( "Bonjour le monde !", taille )
    update()
repeat

// Centre une chaine texte a l'ecran
function centreTexte( message, taille )
    textSize( taille )
    tw = textWidth( message )
    drawText( ( gWidth() - tw ) / 2, ( gHeight() - taille ) / 2,
taille, white, message )
return void
```

Commandes associées

function, return, void

while

Objet

-

Répéter une section de code.

Description
Répéter une section de code jusqu'à ce qu'une condition soit remplie.

Syntaxe

```
while condition loop ... repeat // Boucle tant que la condition
est remplie (true)
```

Arguments
condition condition booléenne qui arrête la boucle quand elle est fausse

Exemple

```
image = loadImage( "Untied Games/Enemy small top C", false )
vaisseaux = []
for i = 0 to 2 loop
    vaisseaux[i] = createSprite()
    setSpriteImage( vaisseaux[i], image )
    setSpriteScale( vaisseaux[i], { 5, 5 } )
    setSpriteCollisionShape( vaisseaux[i], SHAPE_TRIANGLE, 25, 25,
180 )
    vaisseaux[i].show_collision_shape = true
repeat

setSpriteRotation( vaisseaux[0], 270 )
setSpriteSpeed( vaisseaux[0], { 240, 0 } )
setSpriteSpeed( vaisseaux[1], { 0, 120 } )
setSpriteColour( vaisseaux[1], { 0, 0, 1, 1 } )
setSpriteLocation( vaisseaux[0], { 0, gHeight() / 2 } )
setSpriteLocation( vaisseaux[1], { gWidth() / 2, 0 } )

collision = false
while !collision loop
    clear()
    updateSprites()
    drawSprites()
    update()
    collision = detectSpriteCollision( vaisseaux[0], vaisseaux[1]
)
repeat
```

-

Commandes associées

for, repeat, step, to, while

Opérateurs

addition

Objet
Opérateur de l'Addition **+**

Description
Trouve la somme des nombres de part et d'autre de l'opérateur +. Le résultat est le premier nombre augmenté de la valeur du second.

Syntaxe

```
resultat = nombre1 + nombre2
```

Arguments
resultat somme de nombre1 et nombre2

nombre1 premier nombre

-

nombre2 second nombre

Exemple

```
reponse = "0"
correct = 2 + 2
while int( reponse ) != correct loop
    reponse = input( "A quoi est égal 2 + 2 ?", false )
    if int( reponse ) != correct then
        print( "Desole mais c'est inexacte. S'il vous plait,
essayez encore." )
        for i = 0 to 200 loop
            update()
        repeat
    endIf
repeat
print( "C'est exact !" )
sleep( 3 )
```

Commandes Associées

addition, division, modulo, multiplication, soustraction

divide

Objet
Opérateur de la Division **/**

Description
C’est l’opposé de l’opérateur de multiplication. Le résultat est le nombre de fois que vous pouvez soustraire le second nombre au premier.

Syntaxe

```
resultat = nombre1 / nombre2
```

Arguments
resultat nombre1 divisé par nombre2

-

nombre1 premier nombre

nombre2 second nombre

Exemple

```
message = "Bonjour le monde !"
for taille = 1 to 200 step 1 loop
    clear()
    textSize( taille )
    tw = textWidth( message )
    drawText( ( gWidth() - tw ) / 2, ( gHeight() - taille ) / 2,
taille, white, message )
    update()
repeat
```

Commandes Associées

addition, division, modulo, multiplication, soustraction

modulo

Objet
Opérateur Modulo **%**

Description
Trouve le reste de la division d'un nombre par un autre.

Syntaxe

```
resultat = nombre1 % nombre2
```

Arguments
resultat le reste de la division de nombre1 par nombre2

nombre1 premier nombre

-

nombre2 second nombre

Exemple

```
textsize( 100 )
for y = 0 to theight() loop
    for x = 0 to twidth() loop
      printAt( x, y, ( x + 1 ) % 10 )
      update()
    repeat
repeat
for i = 1 to 100 loop
    update()
repeat
```

Commandes Associées

addition, division, modulo, multiplication, soustraction

multiplication

Objet
Opérateur de la Multiplication *****

Description
Trouve le produit de des nombres de part et d'autre de l'opérateur *****. Le résultat est le premier nombre ajouté à lui-même autant de fois qu'indiqué par le second nombre.

Syntaxe

```
resultat = nombre1 * nombre2
```

Arguments
resultat produit de nombre1 par nombre2

nombre1 premier nombre

nombre2 second nombre

Exemple

```
reponse = "0"
correct = 6 * 7
while int( reponse ) != correct loop
    reponse = input( "A quoi est égal 6 fois 7 ?", false )
    if int( reponse ) != correct then
        print( "Desole mais c'est inexacte. S'il vous plait,
essayez encore." )
        for i = 0 to 200 loop
            update()
        repeat
    endIf
repeat
print( "C'est exact !" )
sleep( 3 )
```

Commandes Associées

addition, division, modulo, multiplication, soustraction

soustraction

Objet
Opérateur de la Soustraction -

Description
Trouve la différence entre les nombres de part et d'autre de l'opérateur -. Le résultat est le premier nombre Finds the difference between the numbers either side of the - operator. The result is the first number diminué de la valeur du second.

Syntaxe

```
resultat = nombre1 - nombre2
```

Arguments
resultat différence entre nombre1 et nombre2

-

nombre1 premier nombre

nombre2 second nombre

Exemple

```
image = loadImage( "Untied Games/Enemy A", false )
ennemis = []
for i = 0 to 4 loop
    ennemis[ i ] = createSprite()
    setSpriteImage( ennemis[ i ], image)
    setSpriteAnimation( ennemis[ i ], 0, 4, 20 )
    setSpriteLocation( ennemis[ i ], { (i % 2) * 400 + 400, int(i
/ 2) * 300 + 200 } )
    setSpriteScale( ennemis[ i ], { 4, 4 } )
repeat
camera = getSpriteCamera()
rotation = getSpriteCameraRotation()
loop
    clear()
    c = controls( 0 )
    printAt( 0, 0, "Camera position: x = ", camera.x, " y = ",
camera.y, " z = ", camera.z, " rotation: ", rotation )
    printAt(0, 1, "Use left joypad to pan, right joypad to
zoom/rotate")
    if c.up then
        camera.y -= 5
    endIf
    if c.down then
        camera.y += 5
    endIf
    if c.left then
        camera.x -= 5
    endIf
    if c.right then
        camera.x += 5
    endIf
    if c.x then
        camera.z += 0.05
    endIf
    if c.b then
        camera.z -= 0.05
    endIf
    if c.y then
        rotation -= 0.5
    endIf
    if c.a then
        rotation += 0.5
```

-

```
    endIf
    setSpriteCamera( camera.x, camera.y, camera.z )
    setSpriteCameraRotation( rotation )
    updateSprites()
    drawSprites()
    update()
repeat
```

Commandes Associées

addition, division, modulo, multiplication, soustraction

and

Objet
Opérateur ET binaire &

Description
Met à 1 les bits de chaque position pour laquelle les bits correspondants des deux nombres de part et d'autre de l'opérateur sont à 1.

Syntaxe

```
resultat = nombre1 & nombre2
```

Arguments
nombre1 premier nombre binaire

nombre2 second nombre binaire

resultat nombre résultant avec uniquement les bits à 1 dans nombre1 ET nombre2

Exemple

```
loop
    clear()
```

-

```
    textSize( 50 )
    octet1 = 123
    octet2 = 234
    printAt( 0, 0, "octet1 =          ", bin2str( octet1 ) )
    printAt( 0, 1, "octet2 =          ", bin2str( octet2 ) )
    printAt( 0, 2, "octet1 & octet2 = ", bin2str( octet1 & octet2
) )
    update()
repeat

function bin2str( octet )
    resultat = ""
    for i = 0 to 8 loop
        bit = octet & 1
        if bit then
            resultat = "1" + resultat
        else
            resultat = "0" + resultat
        endIf
        octet = octet >> 1
    repeat
return resultat
```

Commandes Associées

and, not, or, shiftLeft, shiftRight, xor

not

Objet
Opérateur NON binaire ~

Description
Inverse les bits du nombre binaire de sorte à ce que les uns deviennent des zéros et les zéros des uns.

Syntaxe

```
resultat = ~nombre
```

-

Arguments
nombre nombre binaire

resultat résultat lorsque vous inverser tous les bits du nombre

Exemple

```
loop
    textsize( 50 )
    octet1 = 123
    octet2 = 234
    printat( 0, 0, "octet1 =         ", bin2str( octet1 ))
    printat( 0, 1, "octet2 =         ", bin2str( octet2 ))
    printat( 0, 2, "~octet1 =        ", bin2str( ~octet1 ))
    printat( 0, 3, "~octet2 =        ", bin2str( ~octet2 ))
    update()
repeat

function bin2str( octet )
    resultat = ""
    for i = 0 to 8 loop
        bit = octet & 1
        if bit then
            resultat = "1" + resultat
        else
            resultat = "0" + resultat
        endIf
        octet = octet >> 1
    repeat
return resultat
```

Commandes Associées

and, not, or, shiftLeft, shiftRight, xor

or

Objet
Opérateur OU binaire |

-

Description
Effectue l'opération OU bit à bit. Met à 1 les bits de chaque position pour laquelle l'un des bits correspondants des deux nombres de part et d'autre de l'opérateur est à 1.

Syntaxe

```
resultat = nombre1 | nombre2
```

Arguments
nombre1 premier nombre binaire

nombre2 second nombre binaire

resultat nombre résultant avec uniquement les bits à 1 dans nombre1 OU nombre2

Exemple

```
loop
    textsize( 50 )
    octet1 = 123
    octet2 = 234
    printAt( 0, 0, "octet1 =         ", bin2str( octet1 ) )
    printAt( 0, 1, "octet2 =         ", bin2str( octet2 ) )
    printAt( 0, 2, "octet1 | byte2 = ", bin2str( octet1 | octet2 )
)
    update()
repeat

function bin2str( octet )
    resultat = ""
    for i = 0 to 8 loop
        bit = octet & 1
        if bit then
            resultat = "1" + resultat
        else
            resultat = "0" + resultat
        endIf
        octet = octet >> 1
    repeat
return resultat
```

Commandes Associées

-

and, not, or, shiftLeft, shiftRight, xor

shiftLeft

Objet
Opérateur de décalage à gauche <<

Description
Shift all of the bits in a binary number to the left the specified number of times. The new rightmost bits are set to zero. Bitshift left has the effect of multipling the value of the binary number by 2

Syntaxe

```
result = number1 << number2
```

Arguments
number1 first binary number

number2 number of bits to shift

result resulting number with the bits of number1 shifted left number2 times

Exemple

```
loop
    textsize( 50 )
    byte1 = 123
    byte2 = 234
    printAt( 0, 0, "byte1 =         ", bin2str( byte1 ) )
    printAt( 0, 1, "byte2 =         ", bin2str( byte2 ) )
    printAt( 0, 2, "byte1 << 1 =    ", bin2str( byte1 << 1 ) )
    update()
repeat

function bin2str( byte )
    result = ""
    for i = 0 to 8 loop
        bit = byte & 1
        if bit then
```

-

```
            result = "1" + result
        else
            result = "0" + result
        endIf
        byte = byte >> 1
    repeat
return result
```

Commandes Associées

and, not, or, shiftLeft, shiftRight, xor

shiftRight

Objet
Opérateur de décalage à droite >>

Description
Shift all of the bits in a binary number to the right the specified number of times. The new leftmost bits are set to zero

Syntaxe

```
result = number1 >> number2
```

Arguments
number1 first binary number

number2 number of bits to shift

result resulting number with the bits of number1 shifted right number2 times

Exemple

```
loop
    textsize( 50 )
    byte1 = 123
    byte2 = 234
    printAt( 0, 0, "byte1 =         ", bin2str( byte1 ) )
```

-

```
    printAt( 0, 1, "byte2 =         ", bin2str( byte2 ) )
    printAt( 0, 2, "byte1 >> 1 =    ", bin2str( byte1 >> 1 ) )
    update()
repeat

function bin2str( byte )
    result = ""
    for i = 0 to 8 loop
        bit = byte & 1
        if bit then
            result = "1" + result
        else
            result = "0" + result
        endIf
        byte = byte >> 1
    repeat
return result
```

Commandes Associées

and, not, or, shiftLeft, shiftRight, xor

xor

Objet
Opérateur OU exclusif binaire ^

Description
Sets bits in the result where equivalent bits are different in the number either side of the operator

Syntaxe

```
result = number1 ^ number2
```

Arguments
number1 first binary number

number2 second binary number

-

result resulting number with the bits set that are different between number1 and number2

Exemple

```
loop
    textsize( 50 )
    byte1 = 123
    byte2 = 234
    printAt( 0, 0, "byte1 =         ", bin2str( byte1 ) )
    printAt( 0, 1, "byte2 =         ", bin2str( byte2 ) )
    printAt( 0, 2, "byte1 ^ byte2 = ", bin2str( byte1 ^ byte2 ) )
    update()
repeat

function bin2str( byte )
    result = ""
    for i = 0 to 8 loop
        bit = byte & 1
        if bit then
            result = "1" + result
        else
            result = "0" + result
        endIf
        byte = byte >> 1
    repeat
return result
```

Commandes Associées

and, not, or, shiftLeft, shiftRight, xor