

Visite guidée dans le monde du script (Ep. 2)

Au terme de notre premier voyage dans le monde du script, nous avons pu cartographier nos futurs itinéraires. En effet, le but de ces excursions, est d'explorer le niveau débutant dans l'apprentissage du script LSL. C'est pourquoi, nous allons, pour ce deuxième voyage, observer la faune de ce monde encore inconnu.

Il va s'agir maintenant d'étudier ce qui se trame sur ce continent. On va adopter une démarche un peu plus technique, ce qui j'en suis sûr, vous ravira, amis explorateurs mais rassurez vous tout de même, je vais tacher de laisser assez d'exemples et d'illustrations.

Les sections « Pour mieux comprendre » s'adressent à ceux qui veulent en savoir plus, il n'est pas nécessaire de les parcourir et surtout de les comprendre.

Note : Ceci ne présente pas un cours débutant complet sur les points abordés dans le chapitre, il s'agit d'une initiation « en douceur » au monde du LSL, mais tout retour positif ou négatif est le bien venu.

Tout un écosystème :

Un script, on l'a déjà dit, est un **programme**. Un programme, de part son fonctionnement, manipule des **données**. On appellera notre programme un script même si l'appellation « script » ne concerne pas directement l'état dynamique.

Un script va donc travailler avec des données, l'idée c'est qu'un script manipule des **informations** et les communique. Les informations sont traitées sous la forme de données. Pour qu'une donnée soit manipulée, il faut qu'elle soit placée dans un espace mémoire disponible à partir du script. En effet, voyez par vous même, un objet n'est utilisable que si un lien vous est offert vers cet objet, direct ou indirect. Depuis un script, vous avez accès à ces espaces mémoire indirectement. On va donc devoir définir des données avant de s'en servir. Je précise au passage qu'il existe, pour d'autres langages de programmation, d'autres moyens d'accès aux données, tout dépend du niveau du langage en terme d'implémentation (Par exemple, en PHP, pas besoin de définir les données, ou, au contraire, pour le C, on peut définir une donnée sur plusieurs tailles).

Intéressons nous à ces objets qu'on peut manipuler dans un script pour traiter l'information. Vous avez accès à plusieurs types de données que vous utiliserez suivant vos besoins : integer, float, string, key, list, vector et rotation. Survolons les :

- integer : donnée entière. « Nombre sans virgule »
Exemples : 1, -5, 9999, 568, -69854
- float : donnée réelles ou décimale. « Nombre à virgule »
Exemples : 1.5, -584644.5454, 3.14, 0.0, 1
- string : donnée de type chaine de caractères. « Mot »
Exemples : « Bonjour », « Je m'appelle Ahuri », « a »
- key : donnée UUID d'un objet, chaine de caractères spéciale. « clé »
Exemples : « 5490efac-40fb-4a0f-e866-73577ecc8483 »

- vector : donnée de type vecteur.
Exemples : < 1.0, 1.0, 9.5 >, < 1.5140, 545864.0, -944554.54541 >
- rotation : donnée de type rotation.
Exemples : < 1.0, 1.0, 9.5, 1.0 >
- list : donnée de type 'liste de données' (en fait, il s'agit plutôt d'une structure de données que d'une donnée).
Exemples : [1, « message », 1543.154575, < 1, 1, 1 >] (liste contenant 4 éléments de types différents)

Nous allons consacrer un paragraphe par type de donnée, ces animaux méritent toute notre attention !

Pour nos essais, je vous propose de créer un nouveau script, d'enlever la partie commençant par `touch_start` et terminée par une accolade fermante, et d'enlever la ligne entre les accolades suivant le `state_entry`, vous devriez obtenir ceci :

```
default
{
    state_entry()
    {
    }
}
```

Sauvegardez ce script dans un coin de votre inventaire, il sera notre planche de travail pour plusieurs chapitres encore et nous verrons tant qu'à faire à quoi correspondent les bidules déjà écrits à l'intérieur dans quelques chapitres ;)

Il nous reste encore une notion à aborder avant de nous lancer dans les détails, la notion de **variable** ! Dans les langages de programmation, les données sont stockées dans des espaces mémoire, mais comment pouvons nous y accéder depuis le script ? Et bien, c'est très simple, en identifiant l'espace en question grâce à une adresse ! On a vu qu'on pouvait définir des données entières, réelles, voir même des listes de données ! Mais comment les retrouve t-on dans la mémoire ces informations ? Il nous faut les identifier dans la banque mémoire. Pour cela, on se sert d'objets qu'on appelle des variables, voyons ca de plus près :

On peut définir une variable suivant 3 critères :

- Son *étiquette* : son nom.
- Son *type* : integer, float, string, ...
- Sa *portée* : espace de définition, domaine d'existence.

Juste pour comprendre : une variable est donc un objet qu'on nomme pour le reconnaître parmi d'autres et pour pouvoir accéder au bon espace mémoire ;), le type précise notamment la taille de l'espace mémoire à préparer pour accueillir les informations et la portée on verra ca plus tard ;)

Je vais faire une petite analogie pour que vous finissiez de comprendre cette notion : imaginez que vous vous trouvez devant une TRES grande armoire avec tout pleins d'étagères et sur chaque étagère une lignée de pots de confiture. Pour pouvoir étaler gaiement votre confiture préférée sur votre pain sorti du toaster, il vous faut reconnaître votre pot parmi tous les pots ! Que faites vous donc ? Le plus efficace est d'étiqueter chaque pot avec ce qu'il contient et de lui attribuer

un nom, exemple ici, « ma confiture préférée ! » :D Il vous suffira alors d'aller ouvrir le bon pot pour accéder au contenu.

Votre script ici, vous permet d'ouvrir les portes de l'armoire, les variables correspondent aux étiquetages et les pots aux espaces mémoire ! A chaque script son armoire. Vous allez donc procéder exactement de la même manière ! Ici, la taille du pot dépend du contenu.

Allons plus loin avec cette image des pots de confiture, on s'aperçoit qu'on peut aisément utiliser le contenu en mémoire, pour le lire ou le modifier ;) La confiture d'un pot, je peux si l'envi me prend, la mettre dans un autre pot, la remplacer, y ajouter des ingrédients ... et bien c'est pareil pour les variables en LSL ! En ce qui concerne l'idée de portée, ce qu'il faut voir c'est que tout pot dans l'armoire, n'est pas accessible forcément tout le temps. Nous le verrons plus bas.

Les commentaires, un bon moyen de s'y retrouver et de prendre des notes sur les spécimens rencontrés au fur et à mesure de la visite !

Et oui ! Vous pouvez annoter autant de **commentaires** que vous le voulez sur vos scripts ! Quels intérêts ? Il en existe vraiment beaucoup et cela ne concerne pas que vous ;) En effet, si vous devez revenir dans votre script ultérieurement ou une tierce personne, mieux vaut avoir des explications sur ce qui compose votre script ! Les commentaires dans un script sont en orange. Ils fonctionnent par ligne ; pour mettre une ligne en commentaire, vous devez faire précéder le texte à commenter par `//`.

```
default
{
    state_entry()
    {
        // Ma première fonction :
        llSay(0, "Hello, Avatar!");
        // Elle permet de "faire coucou" dans le chat publique :)
    }
}
```

Les lignes de commentaires ne sont pas interprétées lors de l'évaluation de votre script, donc profitez en mais n'en abusez pas, pour le bien être du lecteur ;))

La faune et sa petite variété :

Nous stockerons nos données dans des variables. Celles ci se définissent en fonction de leur type et avec un nom unique dans l'espace de travail.

On définit une variable de la manière suivante (attention au point virgule) :

```
<type> <nom de la variable> [= <valeur initiale> ] ;
```

Les crochets signifient que l'initialisation de la variable est facultative.

Exemples :

```
default
{
    state_entry()
    {
        // Definition d'un entier non initialisé
        integer mon_entier;

        // Définition d'un décimal initialisé à 6.5
        float mon_reel = 6.5;
    }
}
```

Les noms de variables ne doivent pas commencer par un chiffre, sont sensibles à la casse, n'acceptent pas de caractères spéciaux (dont l'espace), pour en voir les détails, se référer à la convention du langage C, c'est la même. Moi je vous conseille d'utiliser les caractères [a-z, A-Z, 0-9, _] (alphanumériques minuscule et majuscule, et tiret bas). Je vous conseille également d'employer la convention d'écriture pour plus de clarté dans votre script : <caractère de typage><Nom de la variable> (c.f. Exemple ci dessous).

Toute donnée est traitée par la machine comme un mot binaire. Le langage binaire, vous savez, c'est cette suite de 1 et 0 (bits) composant toute information implémentée sur une machine « moderne ». Quand vous définissez le type de votre variable, vous ordonnez à la machine de vous attribuer un espace mémoire et donc une quantité de 1 et 0 pour stocker votre information.

Exemples :

```
float    fDistance = 536.45;
integer  iNb = 4;
list     lDetails = [ fDistance, iNb ];
```

« Woot ! On vient de mettre mes variables dans la liste, c'est permit ca ??? Ne dois je pas mettre des valeurs ? »

Et bien, oui c'est permit et on met bien les valeurs :) Vous comprendrez au fur et à mesure qu'il faut faire la distinction entre le nom de la variable et sa valeur, ici on met les valeurs des variables dans la liste lDetails. Cela est permit car les variables fDistance et iNb sont définies plus tôt dans le script, elles sont donc connues pour les lignes suivantes.

Nous allons utiliser une petite instruction nommée lOwnerSay mais nous n'allons pas préciser comment configurer cette instruction, cela sera l'objet d'un autre cours, sachez juste qu'elle permet d'afficher un message au propriétaire du script. Modifiez juste votre script de base pour qu'il soit comme ceci :

```
default
{
    state_entry()
    {

        // Instruction d'affichage
        lOwnerSay("Contenu de ma variable = " + "(string)<ajouter ici le nom
de la variable>");
    }
}
```

Variable entière 'integer' :

On l'a dit, un integer, est un nombre entier relatif (pas de virgule). Sa valeur est comprise entre -2 147 483 648 et +2 147 483 647.

Pour mieux comprendre, retrouvons l'ordre de grandeur de la taille d'un integer : un entier est codé sur 32 bits. Si on détermine la valeur au bit de poids fort, on trouve : $2^{32} = 4294967296$ qu'on divise en suite par 2 pour trouver la plage dans le négatif = 2 147 483 648 pas tout à fait égale à celle dans le positif pour des raisons d'implémentation. Cependant, on voit bien qu'on n'utilise pas tous les bits disponibles car tous les bits ne sont pas à 1 pour ce max en valeur absolue.

On peut définir des entiers sous forme décimale ou hexadécimale (0x....) :

```
integer iPremier      = 5512623;  
integer iDeuxieme    = 0x61EC1A;  
integer iTroisieme   = -160693;
```

Ils vont nous permettre notamment de compter, d'arrondir des résultats, d'utiliser des drapeaux (flags, c.f. prochains cours), des constantes ...

Variable réelle ou décimale 'float' :

Un float (pour virgule flottante) est un nombre décimal (à virgule) allant de 1.175494351E-38 à 3.402823466E+38. Euh ... j'ai besoin de vous dire que ca fait beaucoup ? :D

On peut définir des floats sous la forme décimale classique ou scientifique comme ci dessous. Attention, lors de leur utilisation sous la forme numérique, à bien les écrire avec des points pour faire les décimales sans quoi ca peut donner de bonnes surprises :D :

exemple classique :

5/2 donne 2 → quotient d'entiers renvoi un entier donc valeur entière du vrai résultat à peu près.
5/2.0 donne 2.5 → quotient de décimaux car l'un au moins en est un donc « vrai » résultat.

```
float min = 1.175494351E-38;  
float max = 3.402823466E+38;  
float sci = 2.6E-5;  
float sci_a = 2.6E+3;  
float sci_b = 2.6E3;  
float sci_c = 26000.E-1;  
float f = 2600; // déclaré en implicite, le .0 est rajouté.  
float E = 85.34859;
```

Les floats vont nous permettre d'évaluer, de calculer, d'être précis,

Variable chaîne de caractères 'string' :

Un string (litt. Chaîne) est une chaîne de caractères, comme un mot, une phrase, un caractère seul, etc. Sa taille est limitée par la mémoire restante pour le script.

On définit simplement un string de la manière suivante :

```
string chaine = "J'aime les bonbons mais ca fait maux dents";
```

Les strings vont nous permettre d'afficher des résultats, de communiquer avec les avatars, de stocker des messages ou de les traiter (cela peut être des commandes), de paramétrer un script

Variable clé 'key' :

Une key (clé) est une chaîne de caractères spécifique, elle permet d'identifier un élément de Second Life (primitive, avatar, texture, notecard, ...), c'est un identifiant unique (UUID). Elle est écrite avec des caractères hexadécimaux (a-f et 0-9) en sections séparées par des tirets.

On définit simplement une clé de la manière suivante (attention aux guillemets) :

```
key kAgent = "5490efac-40fb-4a0f-e866-73577ecc8483";
```

Les keys vous permettent d'identifier des éléments dans Second Life, pour interagir vous en aurez besoin très rapidement.

Variable vecteur 'vector' :

Un vecteur en LSL est un élément permettant de regrouper 3 données de type float. On y accède grâce aux opérateurs .x, .y et .z .

On définit un vecteur comme suit (attention aux <>) :

```
vector vMonVecteur = <1.0, 2.0, 3.0>;
```

Les vecteurs vous permettent de définir des valeurs de position, de vitesse, de force, de couleurs, ...

Variable rotation 'rotation' :

Une variable rotation est un vecteur possédant une composante supplémentaire, donc 4 composantes de type float.

On définit une rotation comme ceci (attention aux <>) :

```
rotation vMaRot = <1.2, 5.6, 8.9, 4.7>;
```

Les rotations permettent de travailler comme leur nom l'indique sur les rotations, notamment les rotations des différents objets sur un simulateur comme une primitive ou un avatar.

Pour mieux comprendre : une variable rotation correspond à un objet mathématique appelé quaternion qui est un élément d'un corps plus vaste que celui des complexes. Les opérations que vous effectuez entre quaternions caractérisent aisément des combinaisons de rotations, d'où leur utilisation. Par exemple, la multiplication permet de faire suivre des rotations, attention elle n'est pas commutative.

Variable liste 'list' :

Une liste est un ensemble rangé de données, indexé à partir de 0. Elle peut contenir tous les types de données cités précédemment. Elle stock des valeurs.

On définit une liste tout aussi simplement (attention aux crochets) :

```
list lListe = [ 1, « message », 1543.154575, < 1, 1, 1 >];
```

Les listes permettent de regrouper des données en suivant une logique. Cela peut être des positions successives par exemple, ou encore des groupements de phrases, beaucoup de choses sont possibles.

A vous :

Nous allons travailler dans le `state_entry` comme je vous l'ai sûrement dit précédemment, il va donc falloir que vous écriviez entre les accolades du `state_entry`. Je vous propose donc maintenant de définir chaque type de donnée, d'essayer d'en définir des autres, etc. Vous pourrez observer le contenu de vos variables en utilisant l'instruction `llOwnerSay`.

Voici quelques exemples que vous devez tester et comprendre :

```
default
{
    state_entry()
    {
        float fVal = 54184.254;

        // Instruction d'affichage
        llOwnerSay("Contenu de ma variable fVal = " + (string)fVal);
    }
}
```

```
default
{
    state_entry()
    {
        float fVal = 54184.254;
        vector vVec = < fVal, 5.26, 87.14>;

        // Instruction d'affichage
        llOwnerSay("Contenu de ma variable vVec = " + (string)vVec);
    }
}
```

```
default
{
    state_entry()
    {
        string sPhrase = "Bonjour, je m'appel Ahuri Serenity.";

        // Instruction d'affichage
        llOwnerSay("Contenu de ma variable sPhrase = " + sPhrase);
    }
}
```

```

default
{
    state_entry()
    {
        integer iNb = 9;
        float fVal = 54184.254;
        vector vVec = < fVal, 5.26, 87.14>;
        string sPhrase = "Bonjour, je m'appel Ahuri Serenity.";

        list lTotal = [ iNb, fVal, vVec, sPhrase];

        // Instruction d'affichage
        llOwnerSay("Contenu de ma variable lTotal = " + (string)lTotal);
    }
}

```

Ce dernier script peut être modifié pour une meilleure observation du contenu de la liste en utilisant une instruction particulière, encore une fois je ne vous demande pas de comprendre cela tout de suite, nous verrons les instructions plus tard :

```

default
{
    state_entry()
    {
        integer iNb = 9;
        float fVal = 54184.254;
        vector vVec = < fVal, 5.26, 87.14>;
        string sPhrase = "Bonjour, je m'appel Ahuri Serenity.";

        list lTotal = [ iNb, fVal, vVec, sPhrase];

        // Instruction d'affichage
        llOwnerSay("Contenu de ma variable lTotal = " + llList2CSV(lTotal));
    }
}

```

Vous avez remarqué j'imagine que pour afficher on utilise un opérateur (string), nous l'étudierons dans le prochain chapitre sur les opérateurs, retenez qu'il permet de forcer la valeur d'une variable à devenir une chaîne de caractère pour être affichée.

Remarque : La valeur d'une variable peut être modifiée pendant l'exécution du script, nous le verrons plus tard. Notez qu'on peut définir une variable en fonction de la valeur d'autres variables comme sur les exemples ci-dessus. Notez également que des instructions peuvent renvoyer des données qui peuvent être alors stockées dans les variables.

Notion de constante :

En LSL, il existe aussi ce qu'on appelle des constantes, ce sont des données pré-implémentées dans le langage et disponibles à tout moment dans votre code, elles apparaissent en **bleu** et en majuscule dans votre programme. On peut citer par exemple : PI, TRUE, FALSE, ACTIVE, AGENT, CONTROL_FWD, ... Elles sont là pour vous simplifier la vie que ce soit pour des calculs, des paramètres ou encore des opérations binaires.

La portée d'une variable :

On ne peut utiliser une variable (en l'appelant par son nom dans le code) que dans l'espace où elle est créée et non dans les niveaux supérieurs, et cela à partir de la ligne où elle est définie. La portée caractérise donc un domaine d'existence de la variable, en dehors elle n'existe pas. On peut donc en définir une semblable avec le même nom si elle n'est pas dans le même bloc. Les blocs sont terminés par une accolade fermante. Une variable définie avant les états au départ du script (avant tout bloc donc), tout en haut, peut être qualifiée de **variable globale** même si elle possède aussi sa propre localité car elle est accessible depuis tout le script. Les autres variables, comprises dans un bloc, sont qualifiées de **variables locales**. Nous aurons l'occasion de revenir sur ces points après avoir vu les notions de blocs.

Conclusion :

Nous avons pu voir ici que le monde du script fourmille de petites bêtes très intéressantes et que sans ces bestioles, le continent tout entier ne servirait à rien. Un script repose sur la manipulation de données, il était donc essentiel de commencer par voir ce que sont ces données et comment on peut les définir. Nous allons voir dans le prochain chapitre comment manipuler les données grâce aux opérateurs. Nous verrons aussi qu'on peut transtyper des valeurs pour que certaines variables ou fonctions puissent profiter de la valeur d'autres variables ayant un type différent.

Merci de m'avoir suivi pour ce deuxième épisode,

Ahuri Serenity.

