

Les structures conditionnelles

Fiche pratique de script LSL niveau débutant

Ahuri Serenity

Professeur de script à l'Ecole SL

EM@iL

10 juillet 2010

Résumé

Dans cette fiche nous allons voir comment manipuler l'information au sein d'un script, nous verrons l'importance des structures conditionnelles pour effectuer des opérations basiques et nécessaires.

Ce document n'est pas une substitution aux nombreux tutoriels disponibles sur Internet et ne vaut pas un cours, il est cependant indispensable et impératif que vous ayez bien intégré tous les points évoqués ici.

Table des matières

1	Introduction	1
2	Regrouper des instructions	2
3	La notion de vrai et de faux	2
4	La structure conditionnelle la plus élémentaire : if	2
5	Les structures de boucle	5
5.1	Les boucles (do) while	5
5.2	La boucle for	6
6	Les structures inconditionnelles	7
6.1	Saut dans le code : jump	7
6.2	Changement d'état : state	8
6.3	Retourner une valeur avec return	8

1 Introduction

Dans un script on manipule l'information mais cela devient très compliqué voir même impossible lorsque l'on veut mettre en place un contrôle sur les opérations effectuées sans structures adaptées. Voilà pourquoi, le Linden Scripting Language (LSL) comme tout autre langage de programmation est muni de structures conditionnelles.

Le principe de ces structures est basé sur les tests logiques qui vont guider le déroulement de vos algorithmes vers les bonnes séries d'instructions et qui vont permettre de répéter des séries d'instructions. Les tests sont basés sur l'évaluation d'expressions logiques.

Tout au long de ce document, nous allons présenter l'ensemble des structures conditionnelles et inconditionnelles du langage LSL tout en nous intéressant aux notions de vrai/faux, de condition et de répétition.

2 Regrouper des instructions

Une instruction est une expression qui possède un sens dans le langage informatique associé. Il existe plusieurs types d'instructions et elles finissent bien souvent par un point-virgule.

Comme vous le savez probablement, un script n'est vulgairement rien d'autre qu'une suite d'instructions. Dans cette suite d'instructions, nous pouvons définir une répartition hiérarchiques d'exécution. Par exemple, les évènements sont visibles à l'intérieur de leur état seulement, et les instructions sont à placer dans les évènements. Si vous regardez bien comment est fait un script vous allez remarquer que pour descendre d'un niveau de hiérarchie, on utilise les accolades.

1
2
3

```
{  
  ...  
}
```

En fait, les accolades correspondent à quelque chose de plus général : elles permettent de délimiter un **bloc**. Un bloc n'est rien d'autre qu'un regroupement d'instructions. Nous aurons besoin de regrouper des instructions en utilisant les structures conditionnelles.

3 La notion de vrai et de faux

Comme nous allons le voir juste après, tout le principe des structures conditionnelles repose sur la **condition**. Une condition est une expression composée de différents éléments : des [opérateurs](#) et des opérandes. Elle est soit vraie soit fausse. Les opérandes peuvent être des variables ou bien des valeurs explicites, ils sont donc de nature limitée par l'ensemble des types de données du langage. Les opérateurs permettent de manipuler les opérandes afin qu'au final on ait notre vérification.

Une condition est logique. Il s'agit d'une proposition qui suit les [principes de la logique classique](#). Par exemple, une condition ne peut pas être vraie et fausse en même temps. Je vous conseille fortement de lire la discussion du lien précédent car elle vous montre la base de la logique appliquée ici en LSL ainsi que beaucoup d'exemples très parlant ! Puis ... c'est très intéressant :) Les opérations/opérateurs dont il est question dans l'article vous les retrouvez dans le langage LSL d'une façon ou d'une autre.

4 La structure conditionnelle la plus élémentaire : if

Imaginons que pendant votre script vous devez exécuter différentes fonctions selon que ce soit le propriétaire de l'objet qui fasse l'action ou non. Comment feriez vous ?

Ce n'est pas évident de trouver une solution lorsque l'on ne connaît pas les structures conditionnelles voir même impossible n'est ce pas ? :) Voilà pourquoi vous devez absolument maîtriser ces structures qui, disons le maintenant, sont à la base de la base du LSL et de tout autre langage informatique.

Avec la structure **if** on se donne les moyens d'effectuer des tests et de pouvoir agir en fonction du résultat du test. Le mot "if" en français veut dire "si" et cela va nous aider !

Reprenons le problème :

```
si proprio = utilisateur alors  
  Executer l'action  
fin si
```

On voit clairement qu'en pensant le problème en français on obtient une syntaxe particulière qui va coller avec celle du LSL :

```
1 if(proprio == utilisateur)
2 {
3     Action();
4 }
```

La structure générale de cette structure conditionnelle est la suivante :

```
1 if( <condition> )
2 {
3     <Instructions>
4 }
```

<condition> est la condition à vérifier pour réaliser les instructions. Selon la nature de ce qui est testé entre les parenthèses on aura une manière différente de regarder si la condition est vraie ou fautive. Par exemple, si la nature de l'élément testé est un entier, on dira qu'elle est vraie si et seulement si sa valeur est non nulle.

Remarquez que nous avons utilisé un double signe égal pour comparer proprio et utilisateur, il s'agit d'un opérateur de comparaison parmi d'autres utilisés dans la formulation des propriétés à tester. Je vous conseille fortement de prendre connaissance de l'ensemble des [opérateurs](#) via le [wiki](#) afin ne pas faire de bêtise lors de vos tests. L'erreur classique est de mettre un seul signe égal dans ce cas là, ce qui ne cause aucune erreur de syntaxe mais une grosse erreur de logique. Le simple égal étant l'affectation tandis que le double est la comparaison par égalité.

Une condition peut être composée de plusieurs sous conditions que l'on doit séparer par des opérateurs logiques tels que && (ET) et || (OU). Exemple :

```
1 if( condition1 && condition2 ) // Si condition ET condition2
2 {
3     <Instructions>
4 }
5 if( condition1 || condition2 ) // Si condition OU condition2
6 {
7     <Instructions>
8 }
```

Remarque : Il existe une nette différence entre les opérateurs && et || avec les opérateurs & et |. Les premiers font partie des opérateurs logiques qui exploitent l'état de vérité de chaque opérande tandis que les seconds font partie des opérateurs bit-à-bits qui exploitent la valeur sur chaque bit de chaque opérande. Ces derniers permettent notamment l'utilisation des flags/masques que nous ne détaillerons pas ici. En terme de performance, les opérateurs logiques sont préférables car ils permettent de court-circuiter les tests, c'est à dire de ne pas tout tester quand cela n'est pas nécessaire. Pour plus d'informations vous pouvez visiter ce [lien](#).

C'est bien beau tout cela mais ci maintenant je veux exécuter des instructions dans le cas contraire, c'est à dire quand ma condition n'est pas vérifiée, je fais quoi ? En LSL, c'est très simple de rajouter ça grâce à l'instruction [else](#) ("sinon" en français). Ici je vais adapter mon script pour que, lorsque le owner est l'utilisateur il dise bonjour et lorsque ce n'est pas le cas il dise à bientôt :

```
1 if(proprio == utilisateur)
2 {
3     llSay( 0, "bonjour !");
4 }
```

```

5 else
6 {
7     llSay( 0, "A bientôt !");
8 }

```

Avec ce couple d'instructions if/else vous êtes en mesure de tester tous les cas possibles et de réagir aux différentes situations se présentant à vous en même temps. Mais dans votre quête de simplification de la vie vous serez amené un moment ou à un autre à utiliser une combinaison des deux. Cela peut vite devenir barbant d'imbriquer des if dans d'autres if sur plusieurs étages (car oui on peut et même autant qu'on veut), voilà pourquoi il existe une notation raccourci sympa : **else if**.

Cette instruction vous permet d'aller chercher un cas particulier dans le cas contraire du if. Voyons un exemple simple : Si notre utilisateur n'est pas le propriétaire, je ne souhaite pas forcément lui dire à bientôt si par exemple il s'agit d'un ami. Regardons ce que cela donne sans la simplification else if puis avec :

```

1 if(proprio == utilisateur)
2 {
3     llSay( 0, "bonjour !");
4 }
5 else
6 {
7     if("Ahuri Serenity" == utilisateur)
8     {
9         llSay( 0, "Salut mon pote !");
10    }
11    else
12    {
13        llSay( 0, "A bientôt !");
14    }
15 }

```

```

1 if(proprio == utilisateur)
2 {
3     llSay( 0, "bonjour !");
4 }
5 else if( "Ahuri Serenity" == utilisateur )
6 {
7     llSay( 0, "Salut mon pote !");
8 }
9 else
10 {
11     llSay( 0, "A bientôt !");
12 }

```

On gagne un étage, de la clarté et des efforts! Bon OK c'est pas super flagrant sur un tel exemple mais croyez moi, sur un code qui occupe beaucoup de lignes ça peut rendre grandement service! :)

```

si proprio = utilisateur alors
    Dire "bonjour!"
sinon si "Ahuri Serenity" = utilisateur alors
    Dire "Salut mon pote!"
sinon
    Dire "A bientôt!"
fin si

```

NB : Il n'est possible de ne pas écrire les accolades que si les actions concernées par la condition ne sont en fait qu'une unique instruction. Cette remarque s'applique aux autres structures conditionnelles. Exemple :

```
1 if(proprio == utilisateur)
2   llSay( 0, "bonjour !");
3 else if( "Ahuri Serenity" == utilisateur )
4   llSay( 0, "Salut mon pote !");
5 else
6 {
7   llSay( 0, "A bientôt !");
8   llSay( 0, "Et bonne route !!");
9 }
```

Conclusion : Avec cette nouvelle structure if/else, nous sommes capables de vérifier des informations et d'agir selon le résultat du test et ça ... ça déchire!!! non???? Mais ne nous arrêtons pas en si bon chemin, regardons maintenant ce qu'on peut faire d'encore plus fort!

5 Les structures de boucle

Lorsque dans un bloc vous voulez répéter une suite d'instructions, il n'y a pas 36 solutions, vous devez utiliser la structure conditionnelle de boucle. Cette structure est un peu plus compliquée que la précédente car elle récupère le principe du if auquel elle rajoute le principe de répétition.

L'idée c'est que le seul principe de répétition ne suffit pas car il faut savoir quand s'arrêter! On utilise en effet un test pour vérifier si on doit s'arrêter ou continuer. Voyons un exemple tout de suite, un code pour faire avancer un train :

```
si train n'est pas à destination alors
  Avancer de quelques mètres
sinon
  Dire "On est arrivé!!"
fin si
```

On part du principe que le train n'est initialement pas à destination. Si on exécute une seule fois cet extrait le train ne va avancer que de quelques mètres. L'idée est donc de répéter cet extrait non? Et oui! Comme vous le remarquerez, le fait de refaire le test à chaque fois pour savoir si on est arrivé ou non, c'est exactement ce que font les boucles.

5.1 Les boucles (do) while

Maintenant si on écrit ca :

```
tantque train n'est pas à destination faire
  Avancer de quelques mètres
fin tantque
Dire "On est arrivé!!"
```

Là on a aucun moyen de se tromper! Si le train est déjà à destination avant l'extrait il n'avancera pas et sinon il va avancer autant de fois que nécessaire avant la destination! Littéralement il avance tant qu'il

n'est pas à la destination. Le nom de ce type de boucle est `while` (en français 'tant que').

Voici la structure générale de cette boucle :

```
1 while( <condition> )
2 {
3   <Instructions>
4 }
```

<condition> est de la même nature que la condition du if.

Cette boucle peut s'écrire de la manière suivante (pour la compréhension) :

```
si <condition> alors
  <Instructions>
  Revenir avant le "si"
fin si
```

Attention ! On voit bien qu'il est nécessaire que les instructions fassent évoluer la condition sinon ça va boucler infiniment et ne jamais sortir de la boucle.

Il existe une autre variante sur cette boucle : `do .. while` (répéter .. tant que). La seule différence est que le tout premier test n'est pas exécuté. Voici comment on le rédige puis sa signification :

```
1 do
2 {
3   <Instructions>
4 }while( <condition> );
```

```
<Instructions>
si <condition> alors
  Revenir avant <Instructions>
fin si
```

Voilà qui devrait vous permettre beaucoup de choses mais ce n'est pas tout !

5.2 La boucle for

Il existe un dernier type de boucle, la boucle `for`. Il s'agit d'une boucle qui possède un peu plus d'options : une partie d'initialisation, une partie de test (comme pour `while`) et une partie de fin de cycle (pour les incréments essentiellement).

Voyons déjà ce que cela signifie :

```
<intialisation>
si <condition> alors
  <instructions>
  <fin de cycle>
  Revenir avant le "si"
fin si
```

<fin de cycle> est donc exécuté après toutes les instructions (à chaque cycle) et <initialisation> avant le tout premier appel des instructions (une seule fois). Voilà la structure générale :

```
1 for(<initialisation> ; <condition> ; <fin de cycle>)
2 {
3     <Instructions>
4
5 }
```

Cette dernière boucle est généralement faite pour répéter des actions d'un nombre de fois connu à l'avance tandis que les précédentes boucles sont plus utilisées sur des répétitions dont le nombre de tours ne peut être facilement déterminé à l'avance. En fin de compte, on pourrait très bien utiliser qu'un seul type de boucle parmi les trois pour tous les cas possibles, mais autant profiter de cette diversité n'est-ce pas ? :)

Un exemple d'utilisation de la boucle for pour terminer :

```
1 integer i = 0;
2 llOwnerSay("J'ai ... " + (string)i);
3 for( i = 1 ; i <= 10; i = i + 1)
4 {
5     llOwnerSay(" non ! ... " + (string)i);
6 }
7 llOwnerSay(" doigts ! Oui c'est bien ca, j'ai " + (string)i + " doigts !");
```

6 Les structures inconditionnelles

6.1 Saut dans le code : jump

Avant de préciser ce dont il s'agit je tiens à dire que personnellement je l'évite comme la peste. On ne dit pas que du bien d'elle et c'est pas pour rien :) Abusez en et vous rendrez votre code illisible. De plus, c'est souvent bien moins efficace que les structures conditionnelles et l'utilisation de variables booléennes! mais bon ... voyons cela tout de même!

A côté des structures précédentes, celle ci va paraître triviale! En effet, elle permet simplement de sauter d'un endroit à l'autre dans le bloc. Le principe est tout aussi simple : vous posez un repère dans votre code grâce au caractère @ et vous bondissez dessus depuis où vous le voulez dans le bloc grâce à l'instruction [jump](#).

Avant de commencer à jouer avec jump, veuillez bien à avoir compris les structures conditionnelles précédentes car sans celles-ci, la structure jump n'a aucun intérêt et puis des conditions mal posées peuvent entraîner des boucles infinies! Je vous l'avais dit que ça pouvait être sale comme méthode!! Je pense que le plus grand intérêt du jump est de pouvoir sortir instantanément d'une imbrication multiple de boucles, voici un exemple qui n'a aucun intérêt particulier si ce n'est de vous montrer comment cela fonctionne :

```
1 integer i = 0;
2 integer j = 0;
3 integer k = 0;
4
5 for( ; i < 100 ; ++i)
6 {
7     for( j = 0; j < 100 ; ++j)
8     {
9         for( k = 0; k < 100 ; ++k)
10        {
11            if( i > 47 && j < 56 && k > 9 )
```

```

12     {
13         jump fin;
14     }
15 }
16 }
17 }
18 @fin;

```

Remarquez que cela peut s'écrire de manière plus compacte xD :

```

1 integer i = 0;
2 integer j = 0;
3 integer k = 0;
4
5 for( ; i < 100 ; ++i)
6     for( j = 0; j < 100 ; ++j)
7         for( k = 0; k < 100 ; ++k)
8             if( i > 47 && j < 56 && k > 9 )
9                 jump fin;
10 @fin;

```

6.2 Changement d'état : state

Cette structure inconditionnelle est spécifique au langage LSL! Vous vous zela (euh .. hum) rendez compte qu'il est fort pratique d'utiliser les [états](#) en LSL. Voici une petite phrase que j'aime sortir quand je parle des états :

Dans un script, en se basant sur le principe évènementiel, il doit y avoir autant d'états que de façon différentes de répondre aux mêmes évènements.

Normalement c'est assez clair, les évènements sont relatifs aux états, plus on a d'états et plus l'objet est complexe du point de vue évènementiel.

Afin d'utiliser ce principe des états pour lequel je pense je ferai aussi une fiche pratique, il nous faut une structure inconditionnelle pour passer d'un état à l'autre : [state](#).

Pour s'en servir rien de plus simple : après avoir défini les différents états dans son script, on peut passer d'un état à un autre en ajoutant, à l'endroit dans le code où ce changement doit se faire, l'instruction :

```

1 state <nom etat>;

```

Lorsqu'on se situe dans l'état A puis qu'on décide d'aller dans l'état B, les évènements `state_exit` de l'état A (bien pratique pour couper un timer déclenché dans cet état) et `state_entry` de l'état B sont activés dans cet ordre.

6.3 Retourner une valeur avec return

Cette instruction fait partie intégrante de la structure de fonction qui fera l'objet d'une autre fiche. Pour faire bref, l'idée c'est de retourner une valeur lors de l'utilisation d'une fonction.

```

1 return <valeur explicite ou nom de variable>;

```

La nature de la valeur retournée doit être identique à la valeur que retourne la fonction dans laquelle `return` est placé. Une fonction qui ne retourne rien de précis ne nécessite pas l'utilisation de `return` (on

peut cependant l'utiliser tout seul) mais les autres doivent impérativement l'avoir. Par exemple, la fonction suivante renvoi un type float ...

```
1 // Cette fonction renvoi le carre d'un nombre
2 float carre( float val)
3 {
4     float resultat = val * val;
5     return resultat;
6 }
```

... que je peux utiliser ensuite ailleurs lorsque j'appelle la fonction :

```
1 float val = 5;
2 float valAuCarre = carre(val);
3 llOwnerSay("val au carre vaut : " + (string)valAuCarre);
```