

LSL

Guide de programmation

par Bestmomo Lagan

Version 2.3

29/08/09

Guide de programmation du LSL

***Avertissement :** Ce guide n'est pas un document officiel de Second Life. Il ne prétend à aucune exhaustivité mais se veut suffisamment complet pour être utile. Il n'engage que la responsabilité de son auteur. Les exemples de scripts n'ont qu'une visée didactique et n'ont pas pour vocation à être utilisés en production. Ce n'est pas un didacticiel progressif mais un guide de programmation. Il est destiné à combler la lacune francophone de LSL. Le présent guide est appelé à évoluer et toute participation est la bienvenue.*

*Pour plus de renseignement concernant les **rotations** se reporter à mon guide spécialisé, de même pour la création d'un **serveur** externe à SL.*

***Remerciements :** Un grand merci à tous ceux qui m'encouragent dans cette démarche ! Et merci à Seb_01 pour ses remarques pertinentes sur le contenu de ce document.*

Table des Matières

1. Introduction.....	1
1.1 Hello world.....	1
1.2 Structure des scripts.....	2
1.3 Types et variables.....	3
1.4 Expressions.....	3
1.5 Instructions.....	5
1.6 Fonctions.....	6
1.6.1 Paramètres.....	6
1.6.2 Corps de fonction.....	7
1.6.3 Événements.....	7
2. Opérateurs.....	8
2.1 Opérateurs unaires.....	8
2.2 Opérateurs binaires.....	9
2.2.1 Opérateurs booléens.....	10
2.2.2 Opérateurs logiques.....	11
2.2.2.1 Opérateur ET.....	11
2.2.2.2 Opérateur OU.....	12
2.2.2.3 Opérateur NON.....	13
2.2.2.4 Opérateur OU Exclusif.....	13
2.2.2.5 Opérateurs de décalage.....	14
3. Types.....	16
3.1 Type integer.....	16
3.2 Type booléen.....	16
3.3 Type float.....	16
3.4 Type key.....	16
3.5 Type string.....	19
3.5.1 Caractères d'échappement.....	20
3.5.2 Insertion, suppression et extraction dans une chaîne.....	20
3.5.3 Recherche d'une sous-chaîne.....	21
3.5.4 Manipulation de chaîne.....	22
3.5.5 Manipulations pour requête HTTP.....	23
3.6 Type vector.....	23
3.7 Type rotation.....	25
3.7.1 Combinaisons de rotations.....	26
3.7.2 Rotation de vecteur.....	26
3.7.3 Fonctions mathématiques sur les rotations.....	27
3.8 Type list.....	28
3.8.1 Extraire des éléments d'une liste.....	29
3.8.2 Recherche dans une liste.....	31
3.8.3 Extraction de liste.....	31
3.8.4 Manipulation de listes.....	32
3.8.5 Classement de listes.....	33
3.8.6 Transformation de listes en string et inversement.....	34
3.8.7 Strided Lists.....	36
3.8.8 Statistiques.....	37
3.9 Constantes.....	38

4. Variables.....	40
4.1 Catégories de variables.....	40
4.1.1 Variable globale.....	40
4.1.2 Variable locale.....	40
4.1.3 Variable paramètre.....	40
4.2 Valeurs par défaut.....	40
4.3 Appellation des variables.....	41
5. Constantes.....	42
5.1 Constantes de type float.....	42
5.2 Constantes de type integer.....	42
5.3 Constantes de type string.....	42
5.4 Constantes de type rotation.....	42
5.5 Constantes de type vector.....	43
6. Conversions.....	44
7. Instructions.....	47
7.1 Point de sortie et accessibilité.....	47
7.2 Blocs.....	47
7.3 Instructions nommées.....	47
7.4 Déclarations de variable.....	47
7.5 Instructions de sélection if et else.....	48
7.6 Instructions d'itération.....	50
7.6.1 Instruction while.....	50
7.6.2 Instruction do.....	51
7.6.3 Instruction for.....	51
7.7 Instructions de saut.....	53
7.7.1 Instruction jump.....	53
7.7.2 Instruction return.....	53
7.7.3 Instruction state.....	53
8. Etats et événements.....	54
8.1 state_entry et state_exit.....	54
8.2 touch_start touch et touch-end.....	55
8.2.1 Détecter le numéro de la face touchée.....	56
8.2.2 Connaître l'orientation de la face touchée.....	56
8.2.3 Connaître l'emplacement du click.....	57
8.3 listen.....	60
8.4 timer.....	61
8.5 sensor et no_sensor.....	61
8.6 changed.....	66
8.7 run_time_permissions et control.....	67
8.8 money.....	69
8.9 on_rez et object_rez.....	70
8.10 attach.....	71
8.11 at_target et not_at_target.....	71
8.12 moving_start et moving_end.....	72
8.13 dataserver.....	73
8.13.1 Notecard.....	73
8.13.2 Informations sur un avatar.....	74
8.14 collision, collision_start, collision-end.....	74

8.15	land_collision, land_collision_start, land_collision-end.....	76
8.16	link_message.....	76
9.	Agir sur un objet.....	78
9.1	Les propriétés persistantes.....	78
9.2	Translation.....	79
9.2.1	Objet libre ou root.....	79
9.2.2	Objet lié.....	81
9.2.2.1	Script dans le prim lié.....	82
9.2.2.2	Script dans le root.....	83
9.3	Rotations.....	83
9.3.1	Objet libre ou root.....	83
9.3.2	Objet lié.....	88
9.3.2.1	Script dans le prim lié.....	88
9.3.2.2	Script dans le root.....	90
9.4	Dimensions.....	90
9.4.1	Objet libre ou root.....	90
9.4.2	Objet lié.....	90
9.4.2.1	Script dans le prim lié.....	90
9.4.2.2	Script dans le root.....	90
9.5	Couleur et transparence.....	91
9.5.1	Objet libre ou root.....	91
9.5.2	Objet lié.....	92
9.5.2.1	Script dans le prim lié.....	92
9.5.2.2	Script dans le root.....	92
9.6	Statut.....	92
9.6.1	Objet libre ou root.....	92
9.6.2	Objet lié.....	93
9.7	Texture.....	93
9.7.1	Appliquer une texture et lire une texture.....	93
9.7.2	Dimensionner une texture.....	93
9.7.3	Affecter une rotation à une texture.....	94
9.7.4	Décaler une texture (offset).....	94
9.7.5	Grouper les modifications de texture avec llSetPrimitiveParams.....	94
9.7.6	Animer une texture.....	94
9.7.6.1	Animation « standard ».....	95
9.7.6.2	Animation en rotation.....	96
9.7.6.3	Animation en dimension.....	96
9.7.7	Agir sur la texture d'un objet lié.....	96
9.8	Inventaire (Inventory).....	96
9.8.1	Donner un objet d'un inventaire.....	97
9.8.2	Se renseigner sur le contenu d'un inventaire.....	97
9.8.3	Connaître le nombre d'éléments d'un type dans l'inventaire.....	99
9.9	Lumières.....	99
9.9.1	Position du soleil et de la lune.....	99
9.9.2	Lampes.....	100
9.9.2.1	Full bright.....	100
9.9.2.2	Lumière.....	100
9.10	Liaisons.....	100
9.10.1	Numérotation des prims liés et nombre de prims.....	101
9.10.2	Communication entre prims liées.....	101

9.10.3 Liaisons et rupture de liaisons.....	101
9.10.3.1 Création de liaison.....	101
9.10.3.2 Rupture de liaison.....	102
9.10.4 Actions entre prims liés.....	102
9.10.5 Autres propriétés.....	103
9.10.5.1 ClickAction.....	103
9.10.5.2 Type de primitive.....	103
9.10.5.3 Paramètres de primitive.....	104
10. Les medias	107
10.1.1 Les sons.....	107
10.1.1.1 Précharger un son.....	107
10.1.1.2 Jouer un son une fois.....	107
10.1.1.3 Limiter un son spacialement.....	108
10.1.1.4 Jouer un son ou des sons en boucle.....	109
10.1.1.5 Jouer plusieurs sons successifs.....	109
10.1.2 La musique.....	110
10.1.3 La video.....	110
11. Agir sur un avatar.....	112
11.1 Communication.....	112
11.2 Attachements.....	117
11.3 Animation.....	119
11.4 Obtenir des informations sur l’avatar.....	119
11.5 Caméra.....	122
11.5.1 Mode Mouselook.....	122
11.5.2 Régler la caméra pour un avatar assis.....	122
11.5.3 Où est la caméra ?.....	122
11.5.4 Contrôle de la caméra.....	123
12. Agir sur un terrain.....	124
12.1 Relief d’un terrain.....	124
12.1.1 Modifier le relief.....	124
12.1.2 S’informer sur le relief.....	125
12.2 Gérer un terrain.....	125
12.2.1 Obtenir des renseignements pour une région.....	125
12.2.2 Obtenir des renseignements pour une parcelle.....	126
12.2.2.1 Commutateurs d’état.....	126
12.2.2.2 Détails.....	127
12.2.2.3 Les primitives de la parcelle.....	127
12.2.3 Gérer les avatars.....	129
12.2.3.1 Liste d’accès.....	129
12.2.3.2 Liste de bannissement.....	130
13. La gestion du temps.....	131
13.1 Connaître l’heure.....	131
13.1.1 Heure PST ou PDT.....	131
13.1.2 Heure GMT ou UTC.....	131
13.1.3 Heure SL.....	132
13.2 Connaître la date.....	132
13.3 Connaître le temps écoulé.....	133
13.4 La dilatation temporelle.....	134

13.5 Délai entre événements.....	134
14. Le travail de scripteur.....	135
14.1 Le cahier des charges.....	135
14.2 L'organisation du code.....	136
14.3 La NoteCard.....	136
14.4 L'écriture du code.....	137
14.4.1 Les variables utiles.....	137
14.4.2 Initialisation.....	137
14.4.3 Etat porte fermée.....	140
14.4.4 Etat porte ouverte.....	141
14.5 Test et debogage du code.....	142
INDEX.....	143
Annexe 1.....	147
Synoptique de fonctionnement du script de la porte.....	147

1. Introduction

Linden Scripting Language, en abrégé **LSL**, est un langage de script simple et puissant destiné à conférer des comportements aux objets de **Second life**. Il est fondé sur la syntaxe classique du langage **C**. Il comporte une machine d'états implicite pour chaque script et gère des événements. Il comporte des fonctions intégrées pour manipuler les objets et **avatars**.

Un objet de **Second Life** peut comporter plusieurs scripts, ce qui permet de répartir les tâches à réaliser.

Un script **LSL** est en mode texte et peut être édité facilement. Il doit être ensuite compilé pour pouvoir être exécuté par une machine virtuelle d'un simulateur. Le simulateur partage son temps de calcul entre tous les scripts actifs. Chaque script dispose de son propre espace mémoire, ce qui évite qu'il écrive dans des zones utilisées par le simulateur et ne provoque le crash de celui-ci. Le revers de ce fonctionnement est que les scripts ont du mal à communiquer entre eux.

Il faut toujours garder à l'esprit que **LSL** n'est qu'un langage de script, qu'il ne possède aucune caractéristique des langages de haut niveau, comme les capacités « objet ».

Ce document s'inspire de différentes sources, il n'est en aucun cas une documentation officielle du langage et n'engage que son auteur.

1.1 Hello world

Il est de tradition de commencer la présentation d'un langage en créant un petit programme "Hello world". Je ne faillirai pas à cette tradition.

```
// Un programme Hello World en LSL
default {
    state_entry() {
        llSay(0, "Hello, World!");
    }
}
```

La première ligne de ce programme contient un commentaire :

```
// Un programme Hello World en LSL
```

Les caractères `//` convertissent le reste de la ligne en commentaire. Il est toujours utile de commenter son code pour des modifications ultérieures. Ces commentaires sont ignorés par le compilateur.

Ce script commence par une déclaration d'état, en l'occurrence **default** :

```
default
```

En effet un script possède un état implicite obligatoire qui se nomme **default**. Lorsqu'un script démarre ou est réinitialisé il se met dans cet état par défaut. En cela **LSL** est un langage qui s'éloigne de **C** ou **java** qui ignorent ces notions d'états. Il faut penser un script **LSL** comme un ensemble d'états. Remarquez également les accolades qui déterminent un bloc de code, ici celui qui constitue l'état par défaut. Cette syntaxe est issue du

langage C, elle est partagée par de nombreux autres langages comme **java**, **PHP** ou **C++**. Les blocs de code permettent de localiser un ensemble fonctionnel d'instructions.

Il est ensuite déclaré un événement **state_entry** :

```
state_entry() {  
    llSay(0, "Hello, World!");  
}
```

Cet événement se déclenche lorsque le script entre dans l'état par défaut. Cet événement existe pour tous les états et permet de réaliser des initialisations lors de l'entrée dans un état. De la même façon qu'un script **LSL** doit se penser en terme d'état (aspect statique) il doit aussi s'organiser autour d'événements (aspect dynamique). Lorsque vous concevez un script **LSL** vous devez vous interroger sur ses différents états et les événements associés.

Vous pouvez voir également dans ce programme, dans le bloc de code de l'événement **state_entry**, une instruction :

```
llSay(0, "Hello, World!");
```

Une instruction est l'élément de base d'un script **LSL**. Cette instruction est constituée par l'appel de la fonction **llSay** qui est destinée à écrire un texte dans le **Chat**. Une fonction est constituée d'un nom (**llSay**) et de paramètres séparés par une virgule (0 et « Hello, World »). Pour **LSL** l'appellation « fonction » est générique et concerne aussi bien les méthodes qui renvoient une valeur que celles qui se contentent d'effectuer certaines tâches.

Il ne s'agit pour le moment que d'une présentation générale volontairement sommaire dont tous les éléments vont être détaillés plus loin dans ce document.

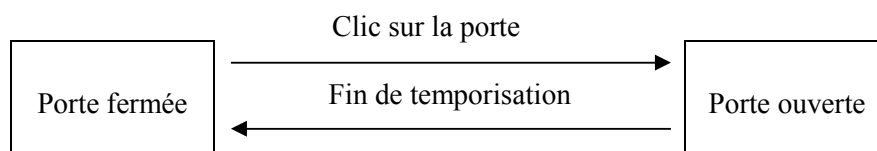
1.2 Structure des scripts

La structure des scripts est simple. Ils sont composés essentiellement d'états et d'événements. Imaginez que vous devez commander une porte. Il existe deux états évidents : porte ouverte et porte fermée. Votre script doit logiquement comporter ces deux états.

Il est évidemment possible de structurer son script différemment et d'utiliser une variable booléenne pour distinguer les deux états mais le script obtenu me semble à la fois moins lisible et éloigné des concepts de base. Vous trouverez toutefois des scripteurs qui vous expliqueront le contraire avec force arguments.

Vous devez ensuite envisager comment vous allez passer d'un état à l'autre. Ce changement est déclenché par un événement, par exemple un clic sur la porte pour le passage de l'état fermé à l'état ouvert. En terme de script cela correspond à l'événement **touch_start**. Pour le changement inverse (de ouvert à fermé) vous pouvez par exemple attendre le terme d'une temporisation, ce qui correspond à l'événement **timer**.

Tout cela peut s'illustrer sous forme de diagramme :



Cette étape est essentielle dans l'élaboration d'un script et permet d'obtenir une structure cohérente qui facilite le codage.

1.3 Types et variables

LSL est un langage fortement typé. Ce qui signifie qu'une variable ne peut pas contenir n'importe quoi, elle doit être d'un certain type, par exemple chaîne de caractères ou nombre entier. Dans **LSL** il n'existe pas de type référence mais uniquement des types valeur. Une variable de type valeur contient tout simplement la valeur qui lui est attribuée alors qu'une variable de type référence nous indique seulement à quel emplacement se trouve la valeur. Voici un tableau résumant les types utilisés par **LSL** :

Type	Description	Plage	Exemple
integer	Un entier signé codé sur 32 bits	-2147483648 à 2147483647	53
float	Une valeur en virgule flottante codée sur 32 bits	1,17594351E-38 à 3,402823466E+38	2.65
string	Une chaîne de caractères		« coucou »
key	Un identifiant unique qui caractérise un objet ou un agent		66864f3c-e095-d9c8-058d-d6575e6ed1b8
vector	3 valeurs de type float utilisées conjointement. Un vector est utilisé pour représenter une donnée en 3 dimensions : position, rotation, direction, velocity , force, impulse et color .	En général <0,0,0> à <1,1,1>	<.3,.8,.2>
rotation	4 valeurs de type float utilisées conjointement et qui représentent une rotation interprétée comme un quaternion.	<0,0,0,0> à <1,1,1,1>	<.1,.32,.56,.2541>
list	Une liste hétérogène qui peut contenir d'autres types.	[2, « coucou », <.3,.8,.2>]	

1.4 Expressions

Une expression est un fragment de code qui peut prendre la valeur d'une valeur ou d'une fonction. Les expressions peuvent contenir une valeur littérale, un appel de fonction, un opérateur et ses opérandes ou un nom simple. Les noms simples peuvent être le nom d'une variable ou un paramètre de fonction.

Les deux types d'expression les plus simples sont les littéraux et les noms simples. Un littéral est une valeur constante qui n'a aucun nom. Prenons un exemple :

```
integer i = 12 ;
```

On définit une variable de type **integer**, qui a un nom simple **i** et on lui affecte la valeur 12 avec l'opérateur d'assignation **=**. Dans ce code 12 est une valeur littérale constante sans nom. Une fois cette affectation effectuée la variable contient la valeur affectée, en l'occurrence ici **i** a désormais la valeur 12. Si on utilise ultérieurement cette variable elle aura cette valeur. Par exemple considérez l'expression suivante :

```
integer n = 10 + i ;
```

Guide de programmation du LSL

Cette expression est constituée d'opérandes et d'opérateurs. Les opérateurs d'une expression indiquent quelles opérations doivent être appliquées aux opérandes. Les opérateurs classiques sont +, -, * et /. Ici on a utilisé le +. Les opérandes peuvent être des littéraux, des variables ou des expressions. Dans cette expression on affecte à **n** le résultat de l'opération **10 + i**. Comme **i** a la valeur 12, **n** aura la valeur 22 après cette assignation.

Nous avons vu dans le programme « Hello World ! » un autre type d'expression, l'expression d'appel :

```
llSay(0, "Hello, World!");
```

Une expression d'appel nécessite le nom d'une fonction suivie d'une parenthèse ouverte, d'une série éventuelle de paramètres séparés par des virgules et une parenthèse fermée.

Remarquez également qu'une expression se termine toujours par le signe « ; » que l'on a tendance à régulièrement oublier...

Lorsqu'une expression contient plusieurs opérateurs ceux ci sont considérés selon un certain ordre. Si vous avez par exemple l'expression **3 + 6 * 4**, celle-ci sera évaluée comme **3 + (6 * 4)** parce que l'opérateur * a une plus grande priorité que +.

Voici un tableau qui donne les priorités dans LSL :

Catégorie	Expression	Description	Commentaires
Primaire	x.m	Accès d'un membre	Par exemple pour un vecteur V de position on extrait la valeur de x en faisant V.x
	x(...)	Appel de fonction	Exemple : llSay(0, « coucou »)
	x++	Post incrémentation	Ajoute 1 à x après l'affectation
	x--	Post décrémentation	Retire 1 à x après l'affectation
Unaire	+x	Identité	
	-x	Négation	
	!x	Négation logique	
	~x	Complément de bits	Inverse tous les bits
	++x	Pré incrémentation	Ajoute 1 à x avant l'affectation
	--x	Pré décrémentation	Retire 1 à x avant l'affectation
Multiplication	x * y	Multiplication	Dans le cas de deux rotations, effectue la seconde sur la première
	x / y	Division	Dans le cas de deux rotations, effectue l'inverse de la seconde rotation sur la première
	x % y	Modulo ou produit vectoriel	Le modulo est le reste de la division entre x et y
Addition	x + y	Addition	
Soustraction	x - y	Soustraction	
Décalage	x << 2	Décalage à gauche	
	x >> 2	Décalage à droite	
Relationnel	x < y	Plus petit que	
	x > y	Plus grand que	
	x <= y	Plus petit ou égal à	
	x >= y	Plus grand ou égal à	
Égalité	x == y	Egal	
	x != y	Pas égal	
Logique	x & y	ET logique	

	x y	OU logique	
Conditionnel	x && y	ET conditionnel	
	x y	OU conditionnel	
Assignement	x = y	Affectation	Affecte la valeur de y à x

1.5 Instructions

Finalement un script **LSL** est un ensemble d'expressions à évaluer séquentiellement. Une instruction est un bloc de construction procédural qui sert à la construction de tous les scripts **LSL**. Une instruction peut déclarer une constante ou une variable locale, appeler une fonction, ou assigner une valeur à une variable. Une instruction de contrôle peut créer une boucle, une boucle **for** par exemple, ou prendre une décision et se brancher à un nouveau bloc de code (ensemble d'instructions entre accolades), comme une instruction **if**. **LSL** comprend un certain nombre de types d'instructions dont la plus simple est le bloc de code :

```
{
    ...
}
```

Nous avons aussi les instructions de sélection **if** et **else** :

```
integer valeur = TRUE;
if (valeur) {llSay(0, "La valeur est vraie");}
else {llSay(0, "La valeur est fausse");}
```

Une instruction de sélection a une fonction d'aiguillage de l'information selon l'état d'une valeur booléenne. Si elle est vraie le bloc situé après **if** est exécuté, dans le cas inverse c'est le bloc situé après **else** qui est exécuté.

Nous avons l'instruction d'itération **for** :

```
integer c;
for(c = 0; c < 10; c++) {llSay(0, (string)c);}
```

Les instructions d'itération permettent d'exécuter plusieurs fois un bloc de code. Ici on initialise une variable entière **c** avec la valeur **0** ; on l'incrément de **1** (**c++**) tant qu'elle reste inférieure à **10**. La boucle **for** est donc exécutée 10 fois. Pour matérialiser cette boucle on envoie dans le **Chat** la valeur de **c**.

Nous pouvons aussi utiliser l'instruction **while** pour faire une boucle :

```
integer c = 1;
while (c ++ < 10) {llSay(0, (string)c);}
```

Ici la boucle est aussi parcourue 10 fois avec **c** qui s'incrémente de 1 à 10.

Une autre façon de faire avec l'instruction **do** :

```
integer c = 1;
do {llSay(0, (string)c);}
while (c ++ < 10);
```

Dans ce cas le test est fait à la sortie de la boucle, donc celle-ci n'est parcourue que 9 fois. Nous verrons ces instructions plus en détail dans un autre chapitre.

Il existe aussi dans LSL une instruction de saut **jump** :

```
if (c > 10) {jump sortie;}
if (d == 8) {jump sortie;}
...
@sortie;
```

Dans ce cas il s'agit d'un saut absolu vers une étiquette.

1.6 Fonctions

Une fonction est un bloc de code qui contient une série d'instructions. Les fonctions dans **LSL** sont déclarées en tête de script. Une fonction a un nom et éventuellement des paramètres. Elle peut avoir une valeur de retour. Voici un exemple de méthode :

```
integer ajoute (integer a, integer b) { return a + b;}
```

Cette fonction a une valeur de retour de type **integer**, deux paramètres également de type **integer**. Elle retourne, grâce à l'instruction **return**, la somme des deux paramètres.

Lorsqu'une fonction ne possède pas de paramètres les parenthèses restent vides. Voici une fonction sans valeur de retour et sans paramètres :

```
incrémente () {n += 10;}
```

Ici **n** est une variable globale, le but de la fonction est d'ajouter la valeur **10** à cette variable.

LSL propose de nombreuses fonctions intégrées qui permettent d'intervenir efficacement au niveau du jeu.

1.6.1 Paramètres

Les paramètres sont utilisés pour transmettre des valeurs aux fonctions. Les paramètres des fonctions obtiennent leur valeur à partir de la valeur des arguments utilisés dans leur appel. Dans **LSL** la transmission se fait uniquement par valeur. Autrement dit le paramètre d'une fonction est une variable locale à la fonction qui s'initialise avec les valeurs passées lors de l'appel au niveau de son argument. Considérez le script suivant :

```
integer ajoute (integer a, integer b) { return a + b; }

integer soustrait (integer a, integer b) {return a - b;}

default {
    state_entry() {
        integer n1 = 9;
        integer n2 = 5;
        llsay(0, (string)ajoute(n1, n2));
        llsay(0, (string)soustrait(n1, n2));
    }
}
```

}

Il comporte en tête deux fonctions. La première destinée à ajouter les deux paramètres transmis et retourner le résultat de l'opération. La seconde fonction fait la même chose avec une soustraction. Au niveau de l'état **default**, dans l'événement **state_entry**, nous déclarons et initialisons deux variables locales de type **integer** : **n1** et **n2**. Nous affichons ensuite sur le **Chat** le résultat de l'appel de la fonction **ajoute**, puis de la fonction **soustrait**. Lors de l'appel des fonctions nous passons en argument les deux variables locales, leur valeur est transmise à la fonction pour initialiser les variables locales **a** et **b**. Autrement dit les variables **n1** et **n2** qui sont locales à l'événement **state_entry** ne sont jamais modifiées par la fonction appelée, c'est juste leur valeur qui est transmise.

1.6.2 Corps de fonction

Le corps de la fonction est l'ensemble des instructions qui sont exécutées lorsqu'elle est appelée. Il est possible dans un corps de fonction de déclarer des variables locales. Les variables locales peuvent être explicitement initialisées. Si ce n'est pas le cas elles prennent la valeur par défaut de leur type. Par exemple une variable de type **integer** est initialisée à la valeur 0.

Si une fonction spécifie un type de retour devant son nom elle doit utiliser l'instruction **return** pour retourner une valeur de ce type à l'appelant de la fonction. Le mot clé **return** arrête l'exécution de la fonction. Observez les script suivant :

```
float surface (integer rayon) {
    float aire = rayon * rayon * PI;
    return aire;}

default {
    state_entry() {
        integer r = 9;
        llSay(0, "L'aire d'un cercle de rayon " + (string)r + " est " + (string)surface(r));}
}
```

Une fonction nommée **surface**, avec **float** comme type de retour, possède un paramètre nommé **rayon**. Cette fonction déclare et initialise une variable locale nommée **aire** avec une expression qui donne la superficie d'un cercle en fonction du rayon. Le mot clé **return** est ensuite utilisé pour retourner la valeur de **aire** à l'appelant qui peut alors l'afficher dans le **Chat**.

1.6.3 Événements

Un événement est une fonction sans valeur de retour, qui est appelée lorsque se produit une action particulière : clic de souris, terme de temporisation, changement au niveau d'un objet... Un événement est déclaré au sein d'un état. Dans le script du point précédent l'événement **state_entry** est déclenché lors de l'entrée dans l'état. De la même façon la sortie d'un état déclenche l'événement **state_exit**. LSL propose de nombreux événements. Un événement très utilisé est **touch_start** qui se déclenche lors d'un clic sur un objet. LSL n'est pas multithread ce qui signifie que les événements sont traités les uns après les autres dans leur ordre d'arrivée. Ils sont stockés dans une file d'attente et exécutés séquentiellement. La file d'attente peut comporter 64 événements. Ce qui signifie que lorsque cette file est pleine les autres événements qui arrivent sont perdus.

❶ Il n'est pas possible de créer des événements, il faut se contenter de ceux qui sont fournis par LSL. Une astuce consiste à utiliser l'événement **timer** pour gérer des interceptions non prévues. Il suffit au niveau de cet événement de procéder à un test de l'élément que l'on veut surveiller.

2. Opérateurs

2.1 Opérateurs unaires

Les opérateurs unaires agissent sur un seul argument :

Opérateur	Nom	Exemple	Retour	Effet	
++	Incrémentation	post-incrémentation	x++	x	Ajout de 1
		pré-incrémentation	++x	x + 1	
--	Décrémentation	post-décrémentation	x--	x	Retrait de 1
		prédécrémentation	--x	x - 1	
!	Non logique	!x	1 si x = 0, sinon 0	aucun	
-	Négation	-x	-x		
~	Complément de bits	~x	Inverse les bits		

Observez le script suivant en essayant de deviner le résultat :

```
default {
    touch_start(integer total_number) {
        integer x = FALSE;
        if(!x) {
            llWhisper(0, (string)(x++));
            llWhisper(0, (string)(++x));
            llWhisper(0, (string)(x--));
            llWhisper(0, (string)(--x));
        }
    }
}
```

Résultat :

```
new whispers: 0
new whispers: 2
new whispers: 2
new whispers: 0
```

Au départ la variable **x** est définie comme **FALSE**, donc sa valeur est à 0. Au niveau du **if**, puisqu'on considère l'inverse de **x**, on a la valeur **TRUE**, donc le bloc s'exécute. Dans le premier **Whisper** on a une post-incrémentation, donc on prend la valeur de **x** pour l'affichage, donc 0, et ensuite on ajoute 1 à **x**. Dans le deuxième **Whisper** on a une pré-incrémentation, donc on ajoute 1 à **x**, ce qui donne donc une valeur de 2, et on affiche. Dans le troisième **Whisper** on a une post-décrémentement, donc on prend la valeur actuelle de **x** pour l'affichage, donc 2, et ensuite on lui enlève 1, ce qui donne comme valeur 1. Pour le dernier **Whisper** on a une pré-décrémentement, donc on commence par enlever 1 à **x**, ce qui amène sa valeur à 0 et on affiche.

2.2 Opérateurs binaires

Les opérateurs binaires acceptent deux arguments et retournent une seule valeur.

Opérateur	Nom	Valeur de retour
+	Addition	Somme
-	Soustraction	Différence
*	Multiplification	Produit
/	Division	Division
%	Modulo	Reste de la division
>	Plus grand	TRUE ou FALSE
<	Moins grand	TRUE ou FALSE
>=	Plus grand ou égal	TRUE ou FALSE
<=	Moins grand ou égal	TRUE ou FALSE
!=	Différent	TRUE ou FALSE
==	Egal à	TRUE ou FALSE

Voici à l'œuvre ces opérateurs :

```
default {
    touch_start(integer total_number) {
        integer i = 3;
        float f = 2.6;
        llWhisper(0, "Nombres " + (string)f + " et " + (string)i);
        llWhisper(0, "Somme : " + (string)(f + i));
        llWhisper(0, "Différence : " + (string)(f - i));
        llWhisper(0, "Produit : " + (string)(f * i));
        llWhisper(0, "Division : " + (string)(f / i));
        llWhisper(0, "Reste de la division : " + (string)(f % i));
        llWhisper(0, (string)f + " plus grand que " + (string)i + " : " + (f > i));
        llWhisper(0, (string)f + " plus petit que " + (string)i + " : " + (f < i));
        llWhisper(0, (string)f + " plus grand ou égal a " + (string)i + " : " + (f >= i));
        llWhisper(0, (string)f + " plus petit ou égal a " + (string)i + " : " + (f <= i));
        llWhisper(0, (string)f + " différent de " + (string)i + " : " + (f != i));
        llWhisper(0, (string)f + " égal a " + (string)i + " : " + (f == i));}
}
```

Résultat :

```
new whispers: Nombres 2.60000 et 3
new whispers: Somme : 5.60000
new whispers: Différence : -0.40000
new whispers: Produit : 7.80000
new whispers: Division : 0.86667
new whispers: Reste de la division : 2
new whispers: 2.60000 plus grand que 3 : False
new whispers: 2.60000 plus petit que 3 : True
new whispers: 2.60000 plus grand ou égal a 3 : False
new whispers: 2.60000 plus petit ou égal a 3 : True
new whispers: 2.60000 différent de 3 : True
```


new whispers: 2.60000 egal a 3 : False

❶ Prenez garde à ne pas confondre l'opérateur d'égalité == avec l'opérateur d'assignation =. Cette erreur est fréquente et n'est pas détectée par le compilateur parce qu'il ne s'agit pas pour lui d'une erreur !

Certains de ces opérateurs peuvent être combinés à une assignation pour rendre le code plus concis :

Opérateur	Exemple	Equivalent
+=	x += 2	x = x + 2
-=	x -= 2	x = x - 2
*=	x *= 2	x = x * 2
/=	x /= 2	x = x / 2
%=	x %= 2	x = x % 2

2.2.1 Opérateurs booléens

Les opérateurs booléens sont des opérateurs binaires un peu particuliers qui n'acceptent que deux valeurs au niveau de leurs arguments : 0 ou 1, ou encore **FALSE** ou **TRUE**. Toutefois toute valeur différente de 0 est considérée comme **TRUE**.

Opérateur	Nom	Valeur de retour
&&	ET	TRUE ou FALSE
 	OU	TRUE ou FALSE
!	NON	TRUE ou FALSE

Considérez le script suivant en essayant de deviner le résultat :

```
default {
    touch_start(integer total_number) {
        integer a = TRUE;
        integer b = FALSE;
        llWhisper(0, (string)(a && b));
        llWhisper(0, (string)(a && a));
        llWhisper(0, (string)(a || b));
        llWhisper(0, (string)(b && b));
        llWhisper(0, (string)!a);
        llWhisper(0, (string)!b);}
}
```

Résultat :

new whispers: 0
new whispers: 1
new whispers: 1
new whispers: 0
new whispers: 0
new whispers: 1

2.2.2 Opérateurs logiques

Les opérateurs logiques sont destinés à opérer au niveau des bits de variables de type **integer**. Quel intérêt ? Et bien il y a des informations qui se contentent de deux états : vrai ou faux. Ce type d'information peut être mémorisé au niveau d'un bit. Etant donné qu'une variable de type **integer** contient 32 bits, on peut ainsi mémoriser 32 valeurs distinctes. Pour manipuler ces bits existent des opérateurs :

Opérateur	Valeur de retour
&	ET
	OU
~	NON
^	OU EXCLUSIF
>>	Décalage à droite
<<	Décalage à gauche

2.2.2.1 Opérateur ET

L'opérateur **ET** compare chaque bit de deux integer. Si les deux sont à 1 alors le résultat est 1 sinon c'est 0. Voici la table de vérité de cet opérateur :

ET	0	1
0	0	0
1	0	1

Prenons un exemple :

```
default {
  touch_start(integer total_number) {
    integer a = 12;
    integer b = 26;
    llWhisper(0, (string)(a & b));}
}
```

Résultat :

```
new whispers: 8
```

Pour comprendre ce résultat il nous faut connaître la représentation binaire des nombres 12 et 26 :

Valeur décimale	Valeur binaire
12	01100
26	11010
8	01000

Vous constatez que l'application du **ET** bit à bit donne bien le résultat affiché. Maintenant que vous avez compris le principe voyons un cas d'application. Considérons l'événement **changed**. Celui-ci comporte un paramètre de type **integer** qui est justement à considérer comme un ensemble de bits significatifs. Lorsque nous voulons détecter qu'un changement de liaison a eu lieu nous écrivons :

```
changed(integer change) {  
    if(change & CHANGED_LINK)  
        ...  
}
```

La valeur de la constante **CHANGED_LINK** est 32 en décimal, donc 100000 en binaire. Si la variable **change** contient le même bit à 1 alors le résultat de l'opération sera **TRUE**. Il se trouve que ce bit correspond justement à un changement dans les liaisons.

2.2.2.2 Opérateur OU

L'opérateur **OU** compare chaque bit de deux integer. Si les deux sont à 1 ou un seul à 1 alors le résultat est 1 sinon c'est 0. Voici la table de vérité de cet opérateur :

OU	0	1
0	0	1
1	1	1

Prenons un exemple :

```
default {  
    touch_start(integer total_number) {  
        integer a = 12;  
        integer b = 26;  
        llWhisper(0, (string)(a | b));    }  
}
```

Résultat :

```
new whispers: 30
```

Pour comprendre ce résultat il nous faut connaître la représentation binaire des nombres 12 et 26 :

Valeur décimale	Valeur binaire
12	01100
26	11010
30	11110

Vous constatez que l'application du **OU** bit à bit donne bien le résultat affiché. Maintenant que vous avez compris le principe voyons un cas d'application. Considérons encore l'événement **changed**. Celui-ci comporte un paramètre de type **integer** qui est justement à considérer comme un ensemble de bits significatifs. Lorsque nous voulons détecter qu'un changement de liaison a eu lieu ou qu'un changement de simulateur à eu lieu nous écrivons :

```
changed(integer change) {  
    if(changed & CHANGED_LINK | changed & CHANGED_REGION)  
        ...  
}
```

La valeur de la constante **CHANGED_LINK** est 32 en décimal, donc 100000 en binaire, la valeur de la constante **CHANGED_REGION** est 256 en décimal, donc 100000000 en binaire. Si la variable **change** contient un de ces bits à 1 alors le résultat de l'opération sera **TRUE**.

2.2.2.3 Opérateur NON

L'opérateur NON s'applique à un seul argument. Il a pour effet d'inverser la valeur des bits, le 0 se transforme en 1, et le 1 en 0. Voici la table de vérité de cet opérateur :

NON	
0	1
1	0

Prenons un exemple :

```
default {
    touch_start(integer total_number) {
        integer valeur = 20;
        integer bit = 4;
        valeur = valeur & ~bit;
        llWhisper(0, (string)valeur);
    }
}
```

Résultat :

```
new whispers: 16
```

Pour comprendre ce code il faut connaître les représentations binaires des valeurs concernées :

Valeur décimale	Valeur binaire
20	10100
4	00100
~4	11011
16	10000

Le but est de mettre à 0 l'un des bits de la variable **valeur**. En l'occurrence le troisième (2^2). On prend un integer dans lequel seul ce bit est à 1, ce qui donne 4 en décimal. On applique ensuite l'opérateur **NON** pour inverser les bits et avoir le seul bit concerné à 0. Ensuite il suffit d'appliquer un opérateur **ET** pour mettre à 0 le bit concerné dans la variable **valeur**. Ce qui amène cette variable à 16 en décimal.

2.2.2.4 Opérateur OU Exclusif

L'opérateur **OU Exclusif** compare chaque bit de deux integer. Si l'un des deux est à 1 et l'autre à 0 alors le résultat est 1, sinon le résultat est 0. Voici la table de vérité de cet opérateur :

OU Exclusif	0	1
0	0	1
1	1	0

Prenons un exemple :

```
default {
    touch_start(integer total_number) {
        integer a = 20;
```

```
integer b = 4;  
integer exclusif = a ^ b;  
llWhisper(0, (string)exclusif);  
}
```

Résultat :

new whispers: 16

Pour comprendre ce code il faut connaître les représentations binaires des valeurs concernées :

Valeur décimale	Valeur binaire
20	10100
4	00100
16	10000

On se retrouve dans le résultat avec seulement à 1 les bits à 1 dans une seule des deux valeurs.

Considérez cet exemple du **wiki** :

```
control(key name, integer levels, integer edges) {  
    if(((levels & CONTROL_FWD) == CONTROL_FWD) ^ ((levels & CONTROL_RIGHT) ==  
CONTROL_RIGHT))  
        ...  
}
```

L'événement **control** concerne la prise de contrôle des commandes de l'avatar (voir page 67 pour la description de cet événement). Il convient d'effectuer des tests pour connaître les touches actionnées. Ici on veut savoir si la touche pour avancer (**CONTROL_FWD = 1**) est actionnée en même temps que la touche pour aller vers la droite (**CONTROL_RIGHT = 8**). Considérons l'élément de code suivant :

```
(levels & CONTROL_FWD) == CONTROL_FWD
```

Cet élément ne peut être vrai (**TRUE**) que si **levels** a son premier bit à 1, c'est à dire si la touche pour aller en avant est actionnée. De la même façon avec :

```
(levels & CONTROL_RIGHT) == CONTROL_RIGHT
```

qui ne peut être vrai (**TRUE**) que si **levels** a son quatrième bit à 1, c'est à dire si la touche pour aller à gauche est actionnée.

Le fait d'utiliser un **OU exclusif** entre ces deux éléments signifie qu'on veut l'action d'une seule des deux touches.

2.2.2.5 Opérateurs de décalage

Les opérateurs de décalage **>>** et **<<** servent simplement à décaler tous les bits à droite ou à gauche. L'intérêt de ces opérateurs dans les scripts est plus que limité. A la rigueur pour faire des multiplications ou divisions rapides. En effet le fait de décaler les bits à droite d'un cran est équivalent à une division par 2. De la même façon un décalage à gauche d'un cran équivaut à une multiplication par 2. Observez cet exemple :

```
default {  
    touch_start(integer total_number) {  
        integer a = 20;  
        integer a_mult2 = a << 1;  
        integer a_mult4 = a << 2;  
        integer a_div2 = a >> 1;  
        integer a_div4 = a >> 2;  
        llWhisper(0, (string)a_mult2);  
        llWhisper(0, (string)a_mult4);  
        llWhisper(0, (string)a_div2);  
        llWhisper(0, (string)a_div4);}  
}
```

Résultat :

```
new whispers: 40  
new whispers: 80  
new whispers: 10  
new whispers: 5
```

Donc si vous avez besoin de vitesse et que vos calculs sont adaptés vous pouvez utiliser cet opérateur.

3. Types

Les types de LSL sont tous des types “valeur”, c’est-à-dire que les variables contiennent directement la valeur mémorisée. Il existe 7 types dans LSL. Deux types numériques : **integer** et **float**, un type chaîne de caractère **string** et 4 types spéciaux : **key**, **vector**, **rotation** et **list**.

3.1 Type integer

LSL comporte un seul type entier : **integer**. Il représente un nombre entier signé codé sur 32 bits dont la valeur peut aller de -2147483648 à 2147483647. Une valeur de type **integer** peut être entrée en hexadécimal, ce qui signifie que ces deux écritures donnent un résultat identique :

```
integer i = 17;
integer i = 0x11;
```

3.2 Type booléen

LSL ne comporte aucun type booléen (vrai ou faux). Pour simuler ce type il faut utiliser un type **integer** en lui affectant les valeurs constantes **TRUE** (valeur 1) et **FALSE** (valeur 0, en pratique toute valeur différente de 1).

3.3 Type float

LSL comporte un seul type virgule flottante : **float**. Il représente un nombre en virgule flottante signé codé sur 32 bits dont la valeur peut aller de 1,17594351E-38 à 3,402823466E+38. Il est possible d'utiliser la notation scientifique dans LSL, ainsi ces deux écritures sont équivalentes :

```
float f = .23;
float f = 23E-2;
```

3.4 Type key

Le type **key** est une valeur unique qui permet d'identifier un objet, un agent, une texture, un script, un son... Il s'agit d'un **string** spécialisé de la forme : "66864f3c-e095-d9c8-058d-d6575e6ed1b8". Vous rencontrerez fréquemment l'appellation **UUID** (Universal Unique IDentifier). Le fait qu'il s'agit d'un **string** permet parfois d'utiliser un paramètre **key** pour transmettre un **string**, c’est le cas par exemple pour le dernier paramètre de la fonction **llMessageLinked**. Il y a quelques fonctions spécialisées concernant les **key** :

Fonction	Action
llDetectedKey	Retourne la clé d'un objet détecté
llGetKey	Retourne la clé de l'objet dans lequel est le script
llGetInventoryKey	Retourne la clé d'un objet dans l'inventaire
llGetOwnerKey	Retourne la clé du propriétaire de l'objet
llGetOwner	Retourne la clé du propriétaire du script
llKey2Name	Retourne le nom correspondant à la clé (uniquement dans le simulateur)

```
default {
    touch_start(integer total_number) {
        llWhisper(0, "Clef de celui qui clique : " + llDetectedKey(0));
        llWhisper(0, "Clef du proprio de l'objet : " + llGetOwnerKey(llGetKey()));
    }
}
```

```

    llWhisper(0, "Clef de l'objet : " + llGetKey());
    llWhisper(0, "Clef du proprio du script : " + llGetOwner());
    llWhisper(0, "Nom du proprio de l'objet : " + llKey2Name(llGetKey()));
}

```

La clef peut changer selon l'élément concerné et les circonstances, il est important de le savoir. En particulier, si la clef d'un avatar, un son, une texture ou une animation ne change jamais, celle d'un objet change à chaque création. Cela est tout à fait normal parce qu'on ne doit jamais se retrouver avec deux objets ayant la même clef. Il en est de même pour les scripts qui changent de clef à chaque modification, il serait perturbant d'avoir deux scripts différents ayant la même clef.

Il existe une collection de clefs de base dans SL, la liste exhaustive en est donnée dans le **wiki**. On y trouve des sons (par exemple celui d'une téléportation « 3d09f582-3851-c0e0-f5ba-277ac5c73fb4 »), des animations (par exemple un avatar qui boit « 0f86e355-dd31-a61c-fdb0-3a96b9aad05f »), des textures (par exemple de l'herbe « 79504bf5-c3ec-0763-6563-d843de66d0a1 »)...

❶ Il n'existe malheureusement pas de fonction **llName2Key**, on ne peut donc pas connaître la clef d'un avatar à partir de son nom, alors que de nombreuses fonctions attendent cette clef. Il existe des bases de données externes à SL qui tentent de rassembler le maximum de ces clefs mais aucune n'est réellement exhaustive. Voici un script qui fait appel à deux de ces serveurs de façon séquentielle :

```

// -----
//          Variables de paramétrage
// -----
integer    canal          = 12547;
list       URL             = ["http://w-hat.com/name2key?terse=1&name=",
                              "http://vision-tech.org/name2key/search.php?name="];
// -----
//          Variables de travail
// -----
string     ERROR           = "1";
string     NAME_NO_VALID  = "2";
string     resident        = "";
key         requestid      = NULL_KEY;
integer     indexURL       = 0;
float      t               = .0;
// -----
//          Réponse
// -----
answer (string value) {llMessageLinked(LINK_THIS, canal + 1, value, NULL_KEY);}
// -----
//          Délai entre deux requêtes
// -----
delai(float d) {
    t = llGetTime();
    if(t < d) llSleep(d - t);
    llResetTime();
}

```



```
// -----
//      Requête
// -----
request() {
    delai(1.0);
    requestid = llHTTPRequest(llList2String(URL, indexURL) + llEscapeURL(resident), [], "");
    llSetTimerEvent(5.0);}

// -----
//      Test index des URL
// -----
integer testIndex () {
    if(++indexURL < llGetListLength(URL)) {
        request();
        return FALSE;}
    else {
        answer(ERROR);
        return TRUE;}
}

// -----
//      Etat par défaut
// -----
default {
    link_message(integer sender_number, integer number, string message, key id) {
        if(number == canal) {
            resident = message;
            if(llSubStringIndex(resident, " ") == -1 || llStringLength(resident) < 4)
                answer(NAME_NO_VALID);
            else state essai;}
    }
}

// -----
//      Envoi des requêtes
// -----
state essai {
    state_entry(){
        indexURL = 0;
        request();}

    http_response(key request_id, integer status, list metadata, string body) {
        if (request_id != requestid) return;
        llSetTimerEvent(.0);
        if (llStringLength(body) == 36 && (key)body != NULL_KEY) answer(body);
        else if(!testIndex ()) return;
        state default;}

    timer() {
```

```

    llSetTimerEvent(.0);
    if(testIndex ()) state default;}
}

```

Et un script pour tester :

```

string      nom                = "Bestmomo Lagan";
integer     canal              = 12547;
string      ERROR              = "1";
string      NAME_NO_VALID      = "2";
default {
    state_entry() {llMessageLinked(LINK_THIS, canal, nom, NULL_KEY);}
    link_message(integer sender_number, integer number, string message, key id) {
        if(number == canal + 1) {
            if(message == ERROR)
                llOwnerSay("Name2key request failed for " + nom);
            else if(message == NAME_NO_VALID)
                llOwnerSay("This name is not a valid name : " + nom);
            else
                llOwnerSay("Key of " + nom + " : " + message);}
        }
    }
}

```

Il suffit de mettre ces deux scripts dans le même prim pour voir le résultat. Vous pouvez adapter ces scripts en fonction de vos besoins.

3.5 Type string

Le type **string** représente une chaîne de caractères Unicode dont la longueur n'est limitée que par la mémoire disponible. Une déclaration de **string** se fait simplement :

```
string s = "test";
```

On peut ajouter des **string** (concaténation) avec l'opérateur + :

```

default {
    touch_start(integer total_number) {
        string prenom = "Charles";
        string nom = "Attan";
        string nomCompleto = prenom + " " + nom;
        llWhisper(0, nomCompleto);
    }
}

```

Résultat :

```
new whispers: Charles Attan
```

On peut connaître le nombre total de caractères dans une chaîne grâce à la fonction **llStringLength** :

```
default {  
    touch_start(integer total_number) {  
        string s = "Un deux trois quatre cinq";  
        llWhisper(0, "Cette chaîne a " + (string)llStringLength(s) + " caracteres");  
    }  
}
```

Résultat :

```
new whispers: Cette chaîne a 25 caracteres
```

En ce qui concerne les conversions entre **string** et **list** consultez le chapitre sur les **list**.

3.5.1 Caractères d'échappement

Il existe 4 caractères d'échappement dans **LSL** :

Caractère	Fonction
<code>\t</code>	Tabulation de 4 espaces
<code>\n</code>	Saut de ligne
<code>\"</code>	Guillemets
<code>\\</code>	Backslash

Voici un exemple d'utilisation :

```
default {  
    touch_start(integer total_number) {  
        llWhisper(0, "Niveau \"un\" \n\tNiveau \"deux\" \n\t\tNiveau \"trois\"");  
    }  
}
```

Résultat :

```
new whispers: Niveau "un"  
             Niveau "deux"  
             Niveau "trois"
```

3.5.2 Insertion, suppression et extraction dans une chaîne

Deux fonctions spécifiques permettent d'insérer une chaîne dans une autre chaîne (**llInsertString**) et de supprimer une sous-chaîne (**llDeleteSubString**), comme on peut le voir dans cet exemple :

```
default {  
    touch_start(integer total_number) {  
        string s = "Un deux quatre cinq";  
        s = llInsertString(s, 8, "trois ");  
        llWhisper(0, s);  
        s = llDeleteSubString(s, 8, 13);  
        llWhisper(0, s);  
    }  
}
```

```
}
```

Résultat :

```
new whispers: Un deux trois quatre cinq
new whispers: Un deux quatre cinq
```

Pour extraire une sous-chaîne il faut utiliser la fonction **IlGetSubString** qui comporte trois paramètres : le premier pour la chaîne, le second pour le début de la sous-chaîne et le troisième pour la fin de la sous-chaîne :

```
default {
    touch_start(integer total_number) {
        string s = "Un deux trois quatre cinq";
        s = IlGetSubString(s, 8, 13);
        IlWhisper(0, s);}
}
```

Résultat :

```
new whispers: trois
```

Il est aussi possible d'utiliser un index négatif, le dernier élément ayant l'index -1, le précédent -2... Autrement dit le code suivant donne le même résultat :

```
default {
    touch_start(integer total_number) {
        string s = "Un deux trois quatre cinq";
        s = IlGetSubString(s, -17, -13);
        IlWhisper(0, s);}
}
```

Si vous utilisez une valeur de début supérieure à la valeur de fin vous effectuez une exclusion. Voici une façon d'exclure une portion de la chaîne :

```
default {
    touch_start(integer total_number) {
        string s = "Un deux trois quatre cinq";
        s = IlGetSubString(s, 20, 12);
        IlWhisper(0, s);}
}
```

Résultat :

```
new whispers: Un deux trois cinq
```

3.5.3 Recherche d'une sous-chaîne

Pour déterminer si une sous-chaîne existe dans une chaîne et pour déterminer son index vous devez utiliser la fonction **IlSubStringIndex**. Cette fonction retrouve l'index de la sous-chaîne ou -1 si elle n'existe pas :

```
default {  
    touch_start(integer total_number) {  
        string s = "un deux trois quatre cinq six sept huit";  
        llWhisper(0, "Index de \"trois\" : " + (string)llSubStringIndex(s, "trois"));  
        llWhisper(0, "Index de \"neuf\" : " + (string)llSubStringIndex(s, "neuf"));}  
}
```

Résultat :

```
new whispers: Index de "trois" : 8  
new whispers: Index de "neuf" : -1
```

3.5.4 Manipulation de chaîne

Il est possible de forcer tous les caractères d'une chaîne en majuscules (**llToUpper**) ou minuscules (**llToLower**):

```
default {  
    touch_start(integer total_number) {  
        string s = "J'ai un beau script";  
        s = llToUpper(s);  
        llWhisper(0, s);  
        s = llToLower(s);  
        llWhisper(0, s);}  
}
```

Résultat :

```
new whispers: J'AI UN BEAU SCRIPT  
new whispers: j'ai un beau script
```

Il y a parfois des espaces intempestifs en début ou en fin de chaîne qu'on aimerait bien voir disparaître, la fonction **llStringTrim** est alors la bienvenue :

```
default {  
    touch_start(integer total_number)  
        string s = " J'ai un beau script ";  
        llWhisper(0, llStringTrim(s, STRING_TRIM) + " moi");  
        llWhisper(0, llStringTrim(s, STRING_TRIM_HEAD) + " moi");  
        llWhisper(0, llStringTrim(s, STRING_TRIM_TAIL) + " moi");  
}
```

Résultat :

```
new whispers: J'ai un beau script moi  
new whispers: J'ai un beau script  moi  
new whispers:  J'ai un beau script moi
```

Les trois constantes permettent de déterminer les espaces à supprimer :

Constante	Fonction
-----------	----------

STRING_TRIM	Suppression des espaces en tête et en queue
STRING_TRIM_HEAD	Suppression des espaces en tête
STRING_TRIM_TAIL	Suppression des espaces en queue

Il est possible de transformer un **string** en **list** et inversement. Cette possibilité est développée au point 3.8.6

3.5.5 Manipulations pour requête HTTP

Le **LSL** permet d'utiliser les requête **HTTP** pour communiquer avec des serveurs externes. Dans une requête **HTTP** un certain nombre de caractères sont interdits comme les caractères accentués et les espaces. Pour transformer ces caractères il existe une fonction dédiée : **lIEscapeURL**, un espace est par exemple remplacé par **%20**. La fonction **lIUnescapeURL** opère l'inverse et restitue dans la chaîne les caractères d'origine.

① Pour une explication détaillée des requêtes **HTTP** se reporter à mon guide spécifique sur les serveurs externes.

3.6 Type vector

Le type **vector** est un ensemble de 3 **float** utilisés comme une unité. Un **vector** peut être utilisé pour représenter :

- une position (**position**),
- une direction,
- une vitesse (**velocity**),
- une force (**force**),
- une impulsion (**impulse**),
- une couleur (**color**),
- une dimension (**scale**)

Un vecteur nul est égal à $\langle 0, 0, 0 \rangle$. Chacun des membres peut être accédé par un point suivi de « x », « y » et « z ». Considérez par exemple le **vector** appelé **Position** de valeur $\langle 2, .36, 1.0 \rangle$, on peut accéder à la première valeur .2 avec **Position.x**.

Il est possible de faire des opérations sur les vecteurs :

Opérateur	Description	Exemple
+	Addition	$\langle 3, 6, 1 \rangle + \langle 6, 1, 2 \rangle = \langle 3+6, 6+1, 1+2 \rangle = \langle 9, 7, 3 \rangle$
-	Soustraction	$\langle 3, 6, 1 \rangle - \langle 6, 1, 2 \rangle = \langle 3-6, 6-1, 1-2 \rangle = \langle -3, 5, -1 \rangle$
*	Produit	$\langle 3, 6, 1 \rangle * \langle 6, 1, 2 \rangle = (3*6) + (6*1) + (1*2) = 26$
%	Produit croisé	$\langle 3, 6, 1 \rangle \% \langle 6, 1, 2 \rangle = \langle 6*2 - 1*1, 1*6 - 3*2, 3*1 - 6*6 \rangle = \langle 11, 0, -33 \rangle$

L'addition et la soustraction sont fréquemment utilisées, par exemple pour déterminer une position. Admettons que nous voulons déplacer un objet de 6 mètres vers le haut. Une façon simple de le faire est d'ajouter à la position actuelle représentée par un vecteur un autre vecteur qui contient le déplacement souhaité :

```
lISetPos(lIGetPos() + <.0,.0,6.0>);
```

De la même manière si nous voulons déplacer le même objet de 6 mètres vers le bas nous pouvons écrire :

```
lISetPos(lIGetPos() - <.0,.0,6.0>);
```

ou encore :

```
llSetPos(llGetPos() + <.0,0,-6.0>);
```

Les deux autres opérations sont beaucoup moins utilisées.

Il y a trois fonctions qui concernent directement les vecteurs :

Fonction	Action
llVecDist	Donne la distance euclidienne entre deux vecteurs.
llVecMag	Donne la distance euclidienne entre deux vecteurs dont le second est <0,0,0>
llVecNorm	Retourne un vecteur normé (valeur 1) du vecteur passé en paramètre

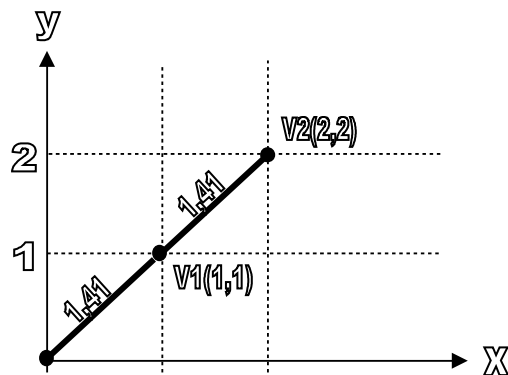
Pour comprendre ces fonctions observez le script suivant :

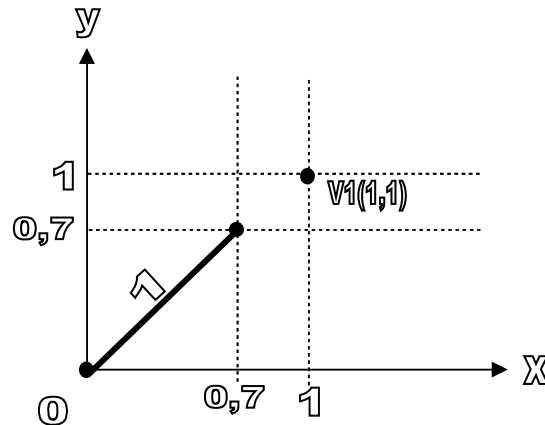
```
default {
    touch_start(integer total_number) {
        vector v1 = <1,1,0>;
        vector v2 = <2,2,0>;
        float d = llVecDist(v1, v2);
        llWhisper(0, "Distance entre les vecteurs " + (string)v1 + " et " + (string)v2 + " : " + (string)d);
        float m = llVecMag(v1);
        llWhisper(0, "Magnitude du vecteur " + (string)v1 + " : " + (string)m);
        vector n = llVecNorm(v1);
        llWhisper(0, "Vecteur normé du vecteur " + (string)v1 + " : " + (string)n);}
}
```

Le résultat à l'exécution donne dans le Chat :

```
new whispers: Distance entre les vecteurs <1.00000, 1.00000, 0.00000> et <2.00000, 2.00000, 0.00000> : 1.41421
new whispers: Magnitude du vecteur <1.00000, 1.00000, 0.00000> : 1.41421
new whispers: Vecteur normé du vecteur <1.00000, 1.00000, 0.00000> : <0.70711, 0.70711, 0.00000>
```

Voici deux dessins pour vous aider à visualiser ces fonctions :





3.7 Type rotation

❶ Pour une description complète des rotations se reporter à mon guide spécifique.

Le type **rotation** est un ensemble de 4 **float** utilisés comme une unité. Une **rotation** représente une rotation sous la forme d'un quaternion. Chaque élément de ce quaternion peut être accédé sous la forme **Rotation.x**. Les rotations sont certainement ce qu'il y a de plus délicat à appréhender au niveau des scripts. Il y a une façon intuitive de comprendre une rotation dans l'espace à partir de rotations autour de 3 axes X (**roll**), Y (**pitch**) et Z (**yaw**). Malheureusement cette représentation intuitive se prête mal à certaines opérations mathématiques et les créateurs de **LSL** ont préféré faire appel à un objet mathématique fréquemment utilisé dans les représentations tridimensionnelles : le quaternion. Pour ceux qui connaissent les nombres complexes qui peuvent être visualisés dans un espace à deux dimensions, et bien les quaternions sont de la même nature avec une représentation dans un espace à 3 dimensions. Un quaternion se représente mathématiquement par 4 nombres x, y, z et s. Mais ceux-ci n'ont absolument rien à voir avec les rotations intuitives dont je vous parlais plus haut. En fait dans les scripts on n'intervient jamais au niveau de ces éléments constitutifs. Il est beaucoup plus simple de raisonner à partir de rotations euclidiennes et d'utiliser les fonctions de conversion qui permettent de passer d'une représentation à l'autre :

Fonction	Action
llEuler2Rot	Transformation de Euler en quaternion
llRot2Euler	Transformation de quaternion en Euler

Prenons un exemple. Vous voulez que votre objet ait une rotation de 10 degrés autour de l'axe X, voici le script correspondant :

```
default {
    state_entry() {
        vector eulerRot = <10.0,0,0>;
        eulerRot *= DEG_TO_RAD;
        rotation rot = llEuler2Rot(eulerRot);
        llSetRot(rot);
    }
}
```


Vous commencez par définir votre rotation avec un vecteur pour lequel vous prévoyez la valeur 10 au niveau de **X**. Comme les fonctions **llEuler2Rot** et **llRot2Euler** attendent des paramètres en radians il faut ensuite convertir la valeur du vecteur avec la constante **DEG_TO_RAD**. On obtient ensuite le quaternion grâce à la fonction de conversion **llEuler2Rot**. Enfin la rotation est appliquée avec la fonction **llSetRot**. Maintenant l'objet a une rotation de 10 degrés au niveau de l'axe **X**.

Nous venons de voir les bases simples pour les rotations. Voyons maintenant un peu les combinaisons de rotations.

3.7.1 Combinaisons de rotations

Reprenons notre exemple ci-dessus en voulant maintenant effectuer une rotation de 10 degrés autour de l'axe **X** pour notre objet mais à partir de sa rotation actuelle. Voici le script correspondant :

```
default {
    state_entry() {
        vector eulerRot = <10.0, .0, .0>;
        eulerRot *= DEG_TO_RAD;
        rotation rot = llEuler2Rot(eulerRot) * llGetRot();
        llSetRot(rot);
    }
}
```

Le code est presque le même que précédemment à part au niveau de la ligne :

```
rotation rot = llEuler2Rot(eulerRot) * llGetRot();
```

Nous avons effectué une multiplication entre deux rotations : celle de 10 degrés sur l'axe **X** et la rotation actuelle de l'objet. Le fait de multiplier deux rotations permet de les ajouter. Dire qu'on les multiplie est d'ailleurs un abus de langage, en toute rigueur il faudrait dire qu'on applique l'opérateur *****. Cette opération n'est pas commutative, autrement dit l'ordre des opérandes est important. Ici on commence par effectuer la rotation de 10 degrés sur l'axe **X** et ensuite on ajoute la rotation actuelle. Cela a pour effet d'effectuer la rotation de 10 degrés à partir des coordonnées locales de l'objet, alors qu'il est considéré avec une rotation nulle.

Maintenant si nous voulons effectuer la rotation inverse, c'est à dire de 10 degrés dans l'autre sens il suffit de diviser les rotations (application de l'opérateur **/**) et la ligne en question devient :

```
rotation rot = llEuler2Rot(eulerRot) / llGetRot();
```

Vous commencez à apprécier les quaternions ? Mais maintenant que se passe-t-il si on inverse les opérandes ?

```
rotation rot = llGetRot() * llEuler2Rot(eulerRot);
```

Cette fois on commence par considérer la rotation de départ de l'objet et ensuite on applique la rotation de 10 degrés. Cette fois la rotation de 10 degrés s'effectue selon l'axe **X** global et non plus local.

Soyez donc vigilant sur l'ordre des opérandes pour vos opérations concernant les rotations !

3.7.2 Rotation de vecteur

Pour appliquer une rotation à un vecteur il suffit de le multiplier par la rotation mais comme nous l'avons vu ci-dessus il faut prendre garde à positionner la rotation à droite de l'opérateur *****. De toutes façons si vous faites

l'inverse vous serez rappelé à l'ordre par le compilateur. Peut-être vous demandez-vous à quoi peut bien servir de faire tourner un vecteur ? Un vecteur peut représenter bien des choses comme nous l'avons vu ci-dessus, par exemple une position. Si vous voulez faire tourner un objet sur lui-même vous pouvez utiliser des fonctions comme **IITargetOmega** ou **IISetRot**. Mais si le centre de rotation est en dehors de l'objet ? Vous pouvez évidemment utiliser un objet lié qui tourne sur lui-même mais cette possibilité n'est pas toujours applicable ne serait-ce qu'à cause de la distance. D'autre part dans ce dernier cas votre objet tourne aussi autour de lui-même ce que vous ne désirez peut-être pas. C'est alors que vous pouvez utiliser le produit d'un vecteur par une rotation. Votre centre de rotation est représenté par un vecteur, de même que la position de votre objet. Le vecteur qui doit tourner est celui qui résulte de la différence entre la position de l'objet et le centre de rotation. C'est à ce vecteur qu'il faut appliquer la rotation. Considérez le script suivant :

```
default {
    state_entry() {
        float pas = 10.0;
        vector eulerRot = <.0, .0, pas * DEG_TO_RAD>;
        rotation rot = IIEuler2Rot(eulerRot);
        vector centreRotation = IIGetPos() + <5.0, .0, .0>;
        integer c;
        integer nombrePas = 360 / (integer)pas;
        for(c = 0; c < nombrePas; c++) {
            vector pos = IIGetPos();
            vector newPos = centreRotation + ((pos - centreRotation) * rot);
            IISetPos(newPos);}
    }
}
```

L'objet qui contient ce script va faire un tour complet autour d'un centre de rotation défini à 5 mètres de sa position de départ sur l'axe X par pas de 10 degrés.

3.7.3 Fonctions mathématiques sur les rotations

LSL comporte un certain nombre de fonctions mathématiques concernant les rotations :

Fonction	Action
IIAngleBetween	Retourne l'angle entre deux rotations
IIAxes2Rot	Convertit 3 axes en une rotation
IIAxisAngle2Rot	Retourne la rotation obtenue par une rotation d'un angle sur un axe
IIEuler2Rot	Transformation de Euler en quaternion
IIList2Rot	Retourne une rotation d'un élément de liste
IIRot2Angle	Retourne l'angle d'une rotation
IIRot2Axis	Retourne l'axe d'une rotation
IIRot2Euler	Transformation de quaternion en Euler
IIRot2Fwd	Retourne un vecteur modulaire représentant l'axe avant/arrière après une rotation
IIRot2Left	Retourne un vecteur modulaire représentant l'axe horizontal après une rotation
IIRot2Up	Retourne un vecteur modulaire représentant l'axe vertical après une rotation
IIRotBetween	Retourne le plus petit angle (rotation) entre deux vecteurs

Voyons un peu certaines de ces fonctions. Dans ce script nous définissons deux rotations et nous cherchons ensuite l'angle entre les deux avec la fonction **IIAngleBetween** :

```
default {
```

```
state_entry() {  
    rotation X_60 = llEuler2Rot(<60.0 * DEG_TO_RAD, .0, .0>);  
    rotation X_90 = llEuler2Rot(<PI_BY_TWO, .0, .0>);  
    float angle = llAngleBetween(X_90, X_60);  
    llWhisper(0, "Angle : " + (string)(angle * RAD_TO_DEG) + " degrees");  
}
```

Voyons un script qui regroupe quelques fonctions :

```
default {  
    touch_start(integer total_number) {  
        rotation rot = llAxisAngle2Rot(<1,1,0>, 30 * DEG_TO_RAD);  
        vector vRot = llRot2Euler(rot) * RAD_TO_DEG;  
        llWhisper(0, "Angle sur l'axe X : " + (string)vRot.x);  
        llWhisper(0, "Angle sur l'axe Y : " + (string)vRot.y);  
        llWhisper(0, "Angle sur l'axe Z : " + (string)vRot.z);  
        vector f = llRot2Fwd(rot);  
        llWhisper(0, "Vecteur avant : " + (string)f);  
        vector l = llRot2Left(rot);  
        llWhisper(0, "Vecteur gauche : " + (string)l);  
        vector u = llRot2Up(rot);  
        llWhisper(0, "Vecteur haut : " + (string)u);  
    }  
}
```

A l'exécution nous obtenons dans le Chat :

```
new whispers: Angle sur l'axe X : 22.20725  
new whispers: Angle sur l'axe Y : 20.70448  
new whispers: Angle sur l'axe Z : -4.10646  
new whispers: Vecteur avant : <0.93301, 0.06699, -0.35355>  
new whispers: Vecteur gauche : <0.06699, 0.93301, 0.35355>  
new whispers: Vecteur haut : <0.35355, -0.35355, 0.86603>
```

Le débogueur de **LSLEditor** dans sa grande mansuétude nous indique aussi au passage la magnitude et le vecteur normé :

```
VecMag(<1.00000, 1.00000, 0.00000>)=1.4142135623731  
VecNorm(<1.00000, 1.00000, 0.00000>)=<0.70711, 0.70711, 0.00000>
```

3.8 Type list

Une liste est un ensemble d'éléments hétérogènes. Au niveau syntaxique il suffit d'aligner les éléments séparés par des virgules, le tout entre crochets :

```
list l = ["coucou", 1.2, <.0,.0,2.3>, NULL_KEY];
```

La contenance d'une liste est limitée par la mémoire autorisée pour un script qui est de 16Ko dans la version classique, portée à 64K avec Mono. Si vous avez besoin de plus de place il faut bricoler un peu pour lier des listes dans plusieurs scripts. Une liste ne peut pas contenir une autre liste.

Il est possible d'ajouter des listes avec l'opérateur +, on parle de concaténation :

```
default {
    state_entry() {
        list l1 = ["coucou", 1.2];
        list l2 = [<0,0,2.3>, <0,0,0,1>];
        list l3 = l1 + l2;
        llWhisper(0, (string)l3);}
}
```

Vous obtenez au niveau du Chat :

```
new whispers: coucou1.20000<0.00000, 0.00000, 2.30000><0.00000, 0.00000, 0.00000, 1.00000>
```

Pour ajouter un élément à une liste vous utilisez aussi l'opérateur + :

```
l += "albert"
```

Pour vider une liste il suffit de ne rien mettre entre les crochets :

```
l = [];
```

3.8.1 Extraire des éléments d'une liste

Les éléments dans une liste sont indexés en partant de 0 pour le premier élément, 1 pour le suivant, et ainsi de suite jusqu'au dernier élément. Pour extraire un élément d'une liste il suffit donc de préciser cet index si on le connaît :

```
default {
    state_entry() {
        list l = ["I0", "I1", "I2", "I3"];
        integer i;
        integer total = llGetListLength(l);
        for(i = 0; i < total; i++)
            llWhisper(0, "Element " + (string)i + " : " + llList2String(l, i));}
}
```

Ce qui donne en sortie :

```
new whispers: Element 0 : I0
new whispers: Element 1 : I1
new whispers: Element 2 : I2
new whispers: Element 3 : I3
```

Au passage vous avez pu voir que la fonction **llGetListLength** permet de connaître le nombre d'éléments dans une liste. Vous avez remarqué également l'utilisation de la fonction **llList2String** pour l'extraction des éléments. En effet une liste pouvant contenir n'importe quel type d'élément il convient de préciser ce type lors de l'extraction. Dans notre exemple il n'y avait que des types **string**. Voici un autre exemple avec différents types :

```
default {
```

```
touch_start(integer total_number) {
    key clef = llGetKey();
    rotation rot = llEuler2Rot(<PI, PI_BY_TWO, .0>);
    integer i = 63;
    float f = 10.2;
    vector v = <.0,.0,.2>;
    string s = "coucou";
    list l = [clef, rot, i, f, v, s];
    llWhisper(0, (string)llList2Key(l, 0));
    llWhisper(0, (string)llList2Rot(l, 1));
    llWhisper(0, (string)llList2Integer(l, 2));
    llWhisper(0, (string)llList2Float(l, 3));
    llWhisper(0, (string)llList2Vector(l, -2));
    llWhisper(0, llList2String(l, -1));
    llWhisper(0, (string)llList2CSV(l));
}
```

Cet exemple est évidemment à considérer seulement comme un élément pédagogique puisque nous définissons des types spécifiques pour ensuite tous les caster en **string**. Il existe une fonction d'extraction par type comme vous pouvez le voir dans ce script. Il est aussi possible d'utiliser un index négatif, le dernier élément ayant l'index -1, le précédent -2... L'utilisation de ces fonctions suppose évidemment que vous connaissez le type de l'élément mais ce n'est pas toujours le cas. Si vous ignorez le type de l'élément la fonction **llGetListEntryType** peut vous sauver :

```
default {
    touch_start(integer _number) {
        key clef = llGetKey();
        rotation rot = llEuler2Rot(<PI, PI_BY_TWO, .0>);
        integer i = 63;
        float f = 10.2;
        vector v = <.0,.0,.2>;
        string s = "coucou";
        list l = [clef, rot, i, f, v, s];
        integer c;
        integer total = llGetListLength(l);
        for(c = 0; c < total; c++) {
            integer t = llGetListEntryType(l, c);
            if (t == TYPE_KEY)
                llWhisper(0, (string)llList2Key(l, 0));
            else if (t == TYPE_ROTATION)
                llWhisper(0, (string)llList2Rot(l, 1));
            else if (t == TYPE_INTEGER)
                llWhisper(0, (string)llList2Integer(l, 2));
            else if (t == TYPE_FLOAT)
                llWhisper(0, (string)llList2Float(l, 3));
            else if (t == TYPE_VECTOR)
                llWhisper(0, (string)llList2Vector(l, -2));
            else if (t == TYPE_STRING)
                llWhisper(0, llList2String(l, -1));
        }
    }
}
```

```
}
```

3.8.2 Recherche dans une liste

Dans les exemples précédents on a vu l'extraction d'une valeur d'une liste en connaissant son index. Mais il arrive fréquemment de rechercher un élément dans une liste sans connaître son index. Dans ce cas la fonction **IIListFindList** est bien utile :

```
default {
    touch_start(integer total_number) {
        list l = ["un", "deux", "trois", "quatre"];
        integer i = IIListFindList(l, ["trois"]);
        IIWhisper(0, "Index de l'element 'trois' : " + (string)i);}
}
```

Résultat :

```
new whispers: Index de l'element 'trois' : 2
```

Cette fonction a deux paramètres : le premier pour la liste dans laquelle on cherche et le second la liste que l'on cherche. Dans cet exemple la liste de recherche possède un seul élément. Mais il est possible de rechercher une liste comportant plusieurs éléments :

```
default {
    touch_start(integer total_number) {
        list l = ["un", "deux", "trois", "quatre", "cinq", "six", "sept"];
        integer i = IIListFindList(l, ["trois", "quatre"]);
        IIWhisper(0, "Index de la liste : " + (string)i);}
}
```

Résultat :

```
new whispers: Index de la liste : 2
```

3.8.3 Extraction de liste

Il est parfois nécessaire d'extraire une liste d'une autre liste. La fonction **IIList2List** permet de réaliser cela :

```
default {
    touch_start(integer total_number) {
        list l = ["un", "deux", "trois", "quatre", "cinq", "six", "sept"];
        list extrait = IIList2List(l, 2, 3);
        IIWhisper(0, "Extrait de liste : " + IIList2CSV(extrait));}
}
```

Résultat :

```
new whispers: Extrait de liste : trois, quatre
```

Cette fonction possède trois paramètres : le premier pour la liste de départ, le second pour le premier élément à extraire, le troisième pour le dernier élément. Comme pour la fonction **IIListFindList** il est possible d'utiliser des valeurs négatives. Nous avons déjà vu cette possibilité pour les extractions de chaînes (point 3.5.2), vous pouvez aussi l'utiliser avec la fonction **IIListFindList**. Ainsi le code suivant est équivalent au précédent :

```
default {
    touch_start(integer total_number) {
        list l = ["un", "deux", "trois", "quatre", "cinq", "six", "sept"];
        list extrait = IIList2List(l, 2, -4);
        IIWhisper(0, "Extrait de liste : " + IIList2CSV(extrait));}
}
```

Une astuce consiste à prévoir une valeur de début supérieure à la valeur de fin. Ce défi à la logique peut avoir son utilité. Observez le code suivant :

```
default {
    touch_start(integer total_number) {
        list l = ["un", "deux", "trois", "quatre", "cinq", "six", "sept"];
        list extrait = IIList2List(l, 4, 2);
        IIWhisper(0, "Extrait de liste : " + IIList2CSV(extrait));}
}
```

Résultat :

```
new whispers: Extrait de liste : un, deux, trois, cinq, six, sept
```

Vous remarquez que l'élément « quatre » a été exclu. C'est une façon d'exclure des éléments lors d'une extraction de liste qui, même si elle n'est pas vraiment intuitive, est plutôt efficace.

3.8.4 Manipulation de listes

Il est possible d'insérer une liste dans une autre avec la fonction **IIListInsertList** :

```
default {
    touch_start(integer total_number) {
        list l1 = ["un", "deux", "cinq", "six", "sept"];
        list l2 = ["trois", "quatre"];
        list insertion = IIListInsertList(l1, l2, 2);
        IIWhisper(0, "Insertion de liste : " + IIList2CSV(insertion));}
}
```

Résultat :

```
new whispers: Extrait de liste : un, deux, trois, quatre, cinq, six, sept
```

Il est aussi possible de remplacer une portion de liste la fonction **IIListReplaceList** :

```
default {
    touch_start(integer total_number)
    {
```

```
list l1 = ["un", "deux", "huit", "neuf", "cinq", "six", "sept"];
list l2 = ["trois", "quatre"];
list remplacement = l1ListReplaceList(l1, l2, 2, 4);
l1Whisper(0, "Remplacement de liste : " + l1List2CSV(remplacement));}
}
```

Résultat :

```
new whispers: Remplacement de liste : un, deux, trois, quatre, six, sept
```

On peut aussi supprimer une partie de liste la fonction **l1DeleteSubList** :

```
default {
  touch_start(integer total_number) {
    list l = ["un", "deux", "trois", "quatre", "huit", "neuf", "cinq", "six", "sept"];
    list suppression = l1DeleteSubList(l, 4, 5);
    l1Whisper(0, "Suppression de liste : " + l1List2CSV(suppression));}
}
```

Résultat :

```
new whispers: Suppression de liste : un, deux, trois, quatre, cinq, six, sept
```

Dans ces différents cas on peut utiliser des valeurs négatives ou une valeur de début plus grande que la valeur de fin pour effectuer des exclusions.

ⓘ Attention avec toutes ces fonctions ! La liste de référence n'est jamais modifiée ! Ce qui peut générer des erreurs pas faciles à déceler. Prenez ce code d'exemple :

```
list l1 = ["moi", "toi"];
list l2 = ["lui", "elle"];
l1ListInsertList(l1, l2, 1);
```

Vous pourriez penser que la list l1 est maintenant égale à ["moi", "lui", "elle", "toi"]. Mais il n'en est rien, elle n'a pas été modifiée. La fonction **l1ListInsertList** a renvoyé la liste complétée mais cette valeur de retour n'a pas été utilisée. Le bon code est :

```
l1 = l1ListInsertList(l1, l2, 1);
```

Cette fois la valeur de l1 est changée.

3.8.5 Classement de listes

Il est parfois nécessaire de mélanger des éléments d'une liste ou de les classer :

```
default {
  touch_start(integer total_number) {
    list l = ["a", "b", "c", "d", "e", "f", "g"];
    list melange = l1ListRandomize(l, 1);
```



```
    IIWhisper(0, "Melange de liste : " + IIList2CSV(melange));  
    list ordre = IIListSort(melange, 1, TRUE);  
    IIWhisper(0, "Tri de liste : " + IIList2CSV(ordre));}  
}
```

Résultat :

```
new whispers: Melange de liste : f, e, c, d, a, g, b  
new whispers: Tri de liste : a, b, c, d, e, f, g
```

Le deuxième paramètre des deux fonctions **IIListRandomize** et **IIListSort** mérite un commentaire. Dans l'exemple la valeur 1 signifie que nous considérons les éléments isolément. Observez la différence si nous utilisons la valeur 2 pour les considérer deux par deux :

```
default {  
    touch_start(integer total_number) {  
        list l=["a", "b", "c", "d", "e", "f"];  
        list melange = IIListRandomize(l, 2);  
        IIWhisper(0, "Melange de liste : " + IIList2CSV(melange));  
        list ordre = IIListSort(melange, 2, TRUE);  
        IIWhisper(0, "Tri de liste : " + IIList2CSV(ordre));}  
}
```

Résultat :

```
new whispers: Melange de liste : a, b, e, f, c, d  
new whispers: Tri de liste : a, b, c, d, e, f
```

3.8.6 Transformation de listes en string et inversement

Si les **list** sont bien pratiques pour stocker des éléments il est souvent utile d'avoir ces mêmes éléments disponibles sous forme d'un **string** :

```
default {  
    touch_start(integer total_number) {  
        list l = ["un", "deux", "trois", "quatre", "huit", "neuf", "cinq", "six", "sept"];  
        string s = IIDumpList2String(l, "|");  
        IIWhisper(0, s);}  
}
```

Résultat :

```
new whispers: un|deux|trois|quatre|huit|neuf|cinq|six|sept
```

La fonction **IIDumpList2String** permet de choisir le caractère de séparation. Rien ne vous empêche d'utiliser une chaîne vide pour concaténer les éléments :

```
default {  
    touch_start(integer total_number) {  
        list l = ["pa", "ra", "pluie"];
```

```

string s = lIDumpList2String(l, "");
llWhisper(0, s);}
}

```

Résultat :

```
new whispers: parapluie
```

Une autre opération bien pratique consiste à transformer un **string** en **list**. Il arrive fréquemment de devoir passer plusieurs informations dans une variable de type **string** parce que la fonction concernée ne possède qu'un paramètre disponible. Prenons un exemple avec la lecture d'une **notecard**. Celle-ci peut contenir, par exemple, des valeurs de position et de rotation :

```

<0.000000, 0.000000, 0.300005>;<0.000000, 0.000000, 0.000000, 1.000000>
<0.200000, 0.000000, 0.500006>;<0.000000, 0.000000, 0.000000, 1.000000>
<0.099997, 0.000000, 0.300005>;<0.000000, 0.000000, 0.519623, 3.946929>

```

Chaque ligne comporte un vecteur pour la position, un caractère séparateur, ici le point-virgule, et une rotation. Écrivons un script pour lire cette **notecard**. Dans le scénario notre script fait partie d'un ensemble, il attend l'ordre de procéder à la lecture de la note et doit renvoyer les informations :

```

integer gNotecardLine;
string gNotecard;
decodeLigne(string ligne) {
    list Valeurs = lIParseString2List(ligne, [";"], []);
    llMessageLinked(LINK_THIS, -1, lIDumpList2String(Valeurs, 0), lIDumpList2String(Valeurs, 1));
}
default {
    link_message(integer sender_number, integer number, string message, key id) {
        if (number == -2) {
            gNotecard = message;
            gNotecardLine = 0;
            llGetNotecardLine(gNotecard, gNotecardLine++);}
        }
    dataserver(key id, string data) {
        if (data != EOF) {
            decodeLigne(data);
            llGetNotecardLine(gNotecard, gNotecardLine++);}
        }
    }
}

```

Intéressons-nous à la fonction **decodeLigne**. Cette fonction possède un paramètre de type **string**. Ce paramètre est destiné à transmettre une ligne de la **notecard**. La première opération consiste à extraire les deux éléments constitutifs de cette ligne. La fonction **lIParseString2List** permet de transformer un string en liste en découpant au niveau d'un ou plusieurs caractères de séparation. Ici nous avons déclaré uniquement le point-virgule. Ensuite nous transmettons ces deux éléments comme paramètres d'une fonction **llMessageLinked** qui permet justement de transmettre deux paramètres de type **string**. En fait le deuxième paramètre de cette fonction est de type **key** qui n'est au final qu'un cas particulier de **string**. Ce genre de manipulation est fréquent dans les scripts et doit être bien maîtrisé.

Au niveau du script appelant nous avons cette ligne de code :

```
llMessageLinked(LINK_THIS, -2, "Valeurs", NULL_KEY);
```

Et on récupère les valeurs dans un événement **link_message** :

```
link_message(integer sender_number, integer number, string message, key id) {  
    if (number == -1) {  
        string pos = message;  
        string rot = (string)id;  
        ...  
    }  
}
```

Si vous observez la fonction **llParseString2List** de plus près vous constatez la présence d'un troisième paramètre que nous avons négligé dans le script d'exemple. Ce paramètre permet de désigner des chaînes à séparer d'office. Observez le script suivant :

```
default {  
    touch_start(integer total_number) {  
        string s = "J'ai envie de sortir: mais je ne peux pas!";  
        list l = llParseString2List(s, [""], [":", "!"]);  
        llWhisper(0, llList2CSV(l));  
    }  
}
```

Résultat :

```
new whispers: J, ai, envie, de, sortir, :, mais, je, ne, peux, pas, !
```

❶ Les passages entre **list** et **string** et inversement peuvent aussi s'effectuer avec les fonctions **llList2CSV** et **llCSV2List**. Dans ce cas le caractère séparateur est automatiquement la virgule. La syntaxe est simplifiée par rapport à la fonction **llParseString2List** mais il faut de méfier de la présence éventuelle de virgules enfouies dans vos données qui pourraient être considérées comme des éléments séparateurs.

Il existe une fonction très proche de **llParseString2List**, c'est **llParseStringKeepNulls** qui fonctionne exactement pareil à une petite différence près. Imaginez que vous ayez le texte « toto,titi,,tutu ». Si vous utilisez la fonction **llParseString2List** avec la virgule comme séparateur vous obtenez 3 éléments : « toto », « titi » et « tutu ». L'élément nul entre les deux virgules accolées est ignoré. Si vous effectuez le même traitement avec **llParseStringKeepNulls** cette fois vous obtenez 4 éléments : « toto », « titi », « » et « tutu ». Comme son nom l'indique la fonction a conservé l'élément nul. Je ne me rappelle pas avoir utilisé cette fonction mais je l'ai vue passer dans quelques scripts.

3.8.7 Strided Lists

Les **list** sont bien pratiques mais connaissent des limitations. En particulier il n'est pas possible de stocker facilement un ensemble d'informations liées. Imaginez que vous voulez mémoriser les noms des avatars qui débarquent sur votre terrain ainsi que leur date de passage. Ces deux informations doivent être liées. Le plus simple est évidemment de tout empiler dans un **string** Avec un caractère séparateur pour récupérer facilement les infos. Vous aurez ainsi des éléments du genre : « Laval Babete;2007-12-10 ». Le **wiki** propose une autre solution avec les **strided lists**. Au lieu de tout empiler dans un **string** vous entrez les informations séparément, ainsi dans notre cas on a deux éléments : « Laval Babete » et « 2007-12-10 ». On peut s'interroger sur l'utilité de cette approche d'autant que les fonctionnalités spécifiques pour ce type de liste sont limitées. Pour le classement **llListSort** et **llListRandomize** prévoient un paramètre pour signaler le nombre d'éléments groupés. Et une

fonction spéciale permet d'extraire une sous liste : **lIList2ListStriped**. Mis à part ces fonctionnalités limitées le reste est moins évident : suppression, ajout, insertion... Ce qui fait que cette possibilité est assez peu utilisée.

3.8.8 Statistiques

La fonction **lIListStatistics**, peu utilisée mais bien pratique, permet d'avoir des informations statistiques pour des listes comportant uniquement des **integer** et des **float**. Voici ses paramètres :

Paramètre	Nom	Type	Fonction
1	Operation	Integer	Opération à effectuer
2	input	list	Liste de valeurs

La valeur retournée est de type **float**. Voici les opérations autorisées :

Opération	Valeur	Action
LIST_STAT_RANGE	0	Etendue des valeurs
LIST_STAT_MIN	1	Plus petite valeur
LIST_STAT_MAX	2	Plus grande valeur
LIST_STAT_MEAN	3	Moyenne : Somme des valeurs divisée par le nombre de valeurs
LIST_STAT_MEDIAN	4	Médiane entre valeurs hautes et valeurs basses
LIST_STAT_STD_DEV	5	Ecart type
LIST_STAT_SUM	6	Somme
LIST_STAT_SUM_SQUARES	7	Somme des carrés
LIST_STAT_NUM_COUNT	8	Nombre d'éléments integer et float
LIST_STAT_GEOMETRIC_MEAN	9	Variance : racine ennième (nombre de valeurs) du produit des valeurs

Un petit script pour tester les fonctions les plus utiles :

```
liste(list l) {
    integer c = 0;
    integer n = lIGetListLength(l);
    string s = "";
    for(c = 0; c < n; c++) s += lIList2String(l, c) + ",";
    lIOwnerSay("Valeurs : " + lIGetSubString(s, 0, -2));
default {
    state_entry() {
        list l = [2,5,12,8,21,4];
        liste(l);
        lIOwnerSay("Etendue : " + (string)((integer)lIListStatistics(LIST_STAT_RANGE, l)));
        lIOwnerSay("Plus petite valeur : " + (string)((integer)lIListStatistics(LIST_STAT_MIN, l)));
        lIOwnerSay("Plus grande valeur : " + (string)((integer)lIListStatistics(LIST_STAT_MAX, l)));
        lIOwnerSay("Moyenne : " + (string)lIListStatistics(LIST_STAT_MEAN, l));
        lIOwnerSay("Mediane : " + (string)((integer)lIListStatistics(LIST_STAT_MEDIAN, l)));
        lIOwnerSay("Somme : " + (string)((integer)lIListStatistics(LIST_STAT_SUM, l)));
        lIOwnerSay("Nombre de valeurs : " + (string)((integer)lIListStatistics(LIST_STAT_NUM_COUNT, l)));
    }
}
```

Résultat :

Valeurs : 2,5,12,8,21,4
Etendue : 19
Plus petite valeur : 2
Plus grande valeur : 21
Moyenne : 8.666667
Mediane : 6
Somme : 52
Nombre de valeurs : 6

Cette fonction s'avère très utile pour traiter des valeurs numériques : votes, tirage, détections, achats, mesures...

3.9 Constantes

LSL est équipé de constantes pour représenter les types :

Constante	Valeur
TYPE_INTEGER	1
TYPE_FLOAT	2
TYPE_STRING	3
TYPE_KEY	4
TYPE_VECTOR	5
TYPE_ROTATION	6
TYPE_INVALID	7

Ces constantes sont utiles lorsque vous désirez connaître le type d'une variable. Le wiki fournit une fonction bien pratique pour déterminer un type :

```
// Written by Chad Statosky
integer GetType(string var) {
    integer n = llGetListLength(llParseStringKeepNulls(var, ["1", "2", "3", "4"], [])) - 1;
    n += llGetListLength(llParseStringKeepNulls(var, ["5", "6", "7", "8"], [])) - 1;
    n += llGetListLength(llParseStringKeepNulls(var, ["9", "0", ".", "<"], [])) - 1;
    n += llGetListLength(llParseStringKeepNulls(var, [">", " ", ",", "-"], [])) - 1;
    if(n == llStringLength(var)) {
        if(llSubStringIndex(var, "<") != -1 || llSubStringIndex(var, ">") != -1) {
            if(llGetListLength(llParseStringKeepNulls(var, [","], [])) == 3) return TYPE_VECTOR;
            else return TYPE_ROTATION;
        }
        else {
            if(llSubStringIndex(var, ".") != -1) return TYPE_FLOAT;
            else return TYPE_INTEGER;
        }
    }
    else {
        if(llStringLength(var) == 36 &&
            llGetListLength(llParseStringKeepNulls(var, ["-"], [])) == 5) return TYPE_KEY;
        else return TYPE_STRING;
    }
}
```

Je vous laisse le plaisir de l'analyser...

Je vous propose toutefois une méthode plus légère pour trouver le type d'une variable qui se fonde sur une fonction des **list** :

```
default {
  touch_start(integer total_number) {
    rotation rot = II Euler2Rot(<30 * DEG_TO_RAD, .0, .0>);
    integer i = IIGetListEntryType([rot], 0);
    IIWhisper(0, "Type de la variable : " + (string)i);}
}
```

Résultat :

```
new whispers: Type de la variable : 6
```

La fonction **IIGetListEntryType** donnant le type d'un élément d'une liste il suffit de mettre la variable dont on veut connaître le type dans une **list** !

4. Variables

Une variable représente une zone de stockage en mémoire. Les types des variables déterminent la valeur qui peut être mémorisée. Dans LSL les variables sont fortement typées, ce qui signifie qu'il faut définir leur type avant de leur affecter une valeur. La valeur d'une variable peut être modifiée à l'aide de l'opérateur d'assignation = ou des opérateurs ++ et -. Une variable peut être assignée dès sa définition ou être assignée ultérieurement.

4.1 Catégories de variables

LSL définit 3 catégories de variables :

1. Variable globale
2. Variable locale
3. Variable paramètre

Dans le script suivant :

```
integer i = 0;
integer ajoute(integer a, integer b) {
    return a + b;
}
default {
    state_entry() {
        string s = "Bonjour";
        ...
    }
    ...
}
```

i est une variable globale, **s** une variable locale, **a** et **b** des variables paramètres.

4.1.1 Variable globale

Une variable déclarée en tête de script est visible pour l'ensemble du script.

4.1.2 Variable locale

Une variable locale est visible uniquement dans le bloc de code dans lequel elle est définie.

4.1.3 Variable paramètre

Une variable paramètre devient active lors de l'appel de la fonction dans laquelle elle est utilisée et son existence s'arrête lors de l'exécution d'une instruction **return** ou de la sortie normale de la fonction.

4.2 Valeurs par défaut

Lorsque vous définissez une variable sans lui assigner de valeur explicite elle prend une valeur par défaut qui dépend de son type. Ainsi une variable de type **integer** est initialisée avec la valeur 0 et une variable de type **string** est assignée avec une chaîne vide «».

4.3 Appellation des variables

Vous avez une grande liberté dans l'appellation des variables. Il faut évidemment éviter les mots clefs du langage. Le nom d'une variable peut comporter des lettres minuscules ou majuscules, des chiffres et le caractère « _ ». Le premier caractère ne doit pas être un chiffre. Il peut être judicieux de nommer les variables avec un intitulé qui rappelle son type. Par exemple commencer avec un « i » pour les **integer**, un « k » pour les **key**...

5. Constantes

Les constantes sont des valeurs prédéfinies dans LSL qui ne peuvent pas changer, c'est d'ailleurs pour cela qu'on les appelle des constantes ! Leur utilisation n'est pas obligatoire mais fortement conseillée pour simplifier l'écriture du code et le rendre plus lisible. Il existe des constantes spécifiques à certaines catégories de fonctions et des constantes générales. Dans ce chapitre je n'évoquerai que celles-ci, les constantes plus spécifiques sont décrites dans les chapitres correspondants.

5.1 Constantes de type float

Constante	Valeur
PI	3.1415926535897932384626433832795 (π) = 180°
TWO_PI	6.283185307179586476925286766559 ($\pi * 2$) = 360°
PI_BY_TWO	1.5707963267948966192313216916398 ($\pi / 2$) = 90°
DEG_TO_RAD	Conversion de degrés en radians : radians = degrés * DEG_TO_RAD
RAD_TO_DEG	Conversion de radians en degrés : degrés = radians * RAD_TO_DEG
SQRT2	1.4142135623730950488016887242097 (racine carrée de 2)

5.2 Constantes de type integer

Constante	Valeur
TRUE	1 cette constante est utilisée dans les expressions booléennes
FALSE	0 cette constante est utilisée dans les expressions booléennes
DEBUG_CHANNEL	2147483647 un canal spécial pour le débogage

5.3 Constantes de type string

Constante	Valeur
NULL_KEY	"00000000-0000-0000-0000-000000000000" Représente un key de valeur nulle, mais la constante est typée comme un string et pas comme un key
EOF	"/n/n/n" indique que la dernière ligne d'un notecard est atteinte en lecture

ⓘ Attention à vos déclarations de variables de type **key**. La valeur par défaut n'est pas **NULL_KEY** ce qui peut générer des erreurs pas faciles à décoder (par exemple si vous faites un test de cette variable). Une bonne habitude est de déclarer systématiquement vos variables **key** en initialisant la valeur.

5.4 Constantes de type rotation

Constante	Valeur
ZERO_ROTATION	<0.0, 0.0, 0.0, 1.0> qui représente une rotation nulle sous forme

	de quaternion
--	---------------

5.5 Constantes de type vector

Constante	Valeur
ZERO_VECTOR	$\langle 0.0, 0.0, 0.0 \rangle$ qui représente un vector null

6. Conversions

Une conversion permet à une expression d'être considérée comme étant d'un certain type. Elle permet également à une expression d'un certain type d'être considérée comme étant d'un autre type. Une conversion peut être implicite ou explicite. Par exemple la conversion d'un type **integer** en type **float** est implicite, ce qui signifie que vous pouvez utiliser des **integer** dans des expressions qui attendent un **float**. Par contre la conversion inverse implique une conversion explicite. Considérez le script suivant :

```
default {
    state_entry() {
        integer n = 320;
        float f = n;           // Conversion implicite de integer vers float
        integer i = (integer)f; // Conversion explicite de float vers integer
        ...
    }
}
```

La conversion d'**integer** vers **float** (variable **f** affectée avec la valeur contenue dans la variable de type **integer** **n**) est implicite, parce qu'il n'y a aucune perte d'information. Par contre la conversion inverse (de **float** vers **integer**) doit être explicite parce qu'il peut y avoir une perte d'information.

La conversion explicite (casting) se fait en écrivant le nom du type entre parenthèses devant le nom de la variable. Par exemple pour afficher dans le **Chat** la valeur d'une variable numérique :

```
default {
    state_entry() {
        integer n = 320;
        llSay(0, (string)n); // Conversion explicite d'integer vers string
    }
}
```

Il existe d'autres façons de convertir un **float** en **integer**. Vous disposez des fonctions mathématiques suivantes :

Fonction	Action	Exemple
llRound	Arrondit à la valeur la plus proche	1.2 -> 1, 1.9 -> 2
llCeil	Transmet l'entier au-dessus	1.2 -> 2, 1.9 -> 2
llFloor	Transmet l'entier en dessous	1.2 -> 1, 1.9 -> 1

❗ Il faut se méfier des valeurs littérales dans les expressions. Considérez cet exemple :

```
float f = 2 / 3;
```

Vous vous attendez certainement à obtenir dans la variable **f** une valeur du style 0,666. Mais en fait lorsque l'expression située sur la droite est évaluée elle l'est faite en considérant des valeurs de type **integer**. Le résultat va donc être une valeur nulle. La syntaxe correcte dans ce cas est donc :

```
float f = 2.0 / 3.0;
```

Conservez cela à l'esprit dans vos expressions numériques pour éviter une laborieuse recherche de bug.

Voici la liste des conversions explicites de LSL :

Type de départ	Type d'arrivée
integer	string
float	integer
float	string
vector	string
rotation	string
integer	list
float	list
key	list
string	list
vector	list
rotation	list
string	integer
string	float
string	vector
string	rotation

Le wiki propose une fonction plus performante que le casting pour convertir les **float** en **string** :

```
string hexc="0123456789ABCDEF";
string Float2Hex(float input)
{
    // Copyright Strife Onizuka, 2006-2007, LGPL, http://www.gnu.org/copyleft/lesser.html or (cc-by)
    http://creativecommons.org/licenses/by/3.0/
    if(input != (integer)input)
    {
        float unsigned = IIFabs(input);
        integer exponent = IIFloor((IILog(unsigned) / 0.69314718055994530941723212145818));
        integer mantissa = (integer)((unsigned / IIPow(2., exponent -= ((exponent >> 31) | 1))) * 0x4000000);
        integer index = (integer)(IILog(mantissa & -mantissa) / 0.69314718055994530941723212145818);
        string str = "p" + (string)(exponent + index - 26);
        mantissa = mantissa >> index;
        do
            str = IIGetSubString(hexc, 15 & mantissa, 15 & mantissa) + str;
        while(mantissa = mantissa >> 4);
        if(input < 0)
            return "-0x" + str;
        return "0x" + str;
    }
    return IIDeleteSubString((string)input, -7, -1);
}
```

L'astuce consiste à passer par une représentation hexadécimale sans perte de précision. Cette fonction est à utiliser pour vos calculs demandant une grande précision.

7. Instructions

LSL propose un certain nombre d'instructions qui sont issues du langage C.

7.1 Point de sortie et accessibilité

Chaque instruction possède un point de sortie qui est l'emplacement qui suit immédiatement la fin de l'instruction. Les règles d'exécution d'instructions composées (c'est à dire que nous avons plusieurs instructions) déterminent ce qui doit être fait lorsque toutes les instructions ont été exécutées. Par exemple lorsqu'une instruction est totalement exécutée au sein d'un bloc, on passe à l'instruction suivante. Lorsqu'une instruction peut être accédée au cours de l'exécution elle est qualifiée d'accessible, dans le cas inverse elle est inaccessible. Considérez le code suivant :

```
affichage() {
    jump fin;
    lISay(0, "coucou");
    @fin;
}
```

L'instruction `lISay(0, "coucou");` ne peut jamais être exécutée, on dit qu'elle est inaccessible. Le compilateur nous fournit un message d'avertissement dans ce cas. Cet exemple est évidemment caricatural mais il y a parfois du code inaccessible pas si évident que ça.

7.2 Blocs

Un bloc permet à de multiples instructions d'être considérées comme une seule instruction. Un bloc est un ensemble d'instructions contenues entre des crochets. Un bloc peut contenir des déclarations de variables locales. L'exécution d'un bloc se fait séquentiellement à partir de la première instruction rencontrée. Une liste d'instruction est une suite d'instructions exécutées séquentiellement. Lorsque la première instruction est exécutée le contrôle passe à la seconde instruction et ainsi de suite.

7.3 Instructions nommées

Une instruction nommée, comme on peut s'en douter, possède un nom qui permet de l'identifier. Il doit être unique. Elle est référencée par une instruction `jump`. Nous en avons vu un exemple au point 7.1.

7.4 Déclarations de variable

Une variable doit être déclarée, qu'elle soit globale ou locale, en précisant son type. Il est possible de lui affecter une valeur lors de l'initialisation. Il est possible d'utiliser une expression pour initialiser une variable. Une variable ne doit être déclarée qu'une fois dans une même portée.

```
integer n = 320;           // Déclaration avec initialisation
string n = "n° " + (string)n; // Déclaration avec initialisation avec une expression
float f;                  // Déclaration sans initialisation
integer n = 12;           // Erreur, variable déjà déclarée
```

7.5 Instructions de sélection if et else

Les instructions de sélection permettent de déterminer les instructions qui doivent être exécutées selon certains critères. C'est un élément essentiel de la programmation. L'instruction **if** est l'instruction de sélection de base qui permet d'exécuter une instruction ou un bloc d'instructions selon la valeur d'une expression booléenne. Il peut être utilisé l'instruction complémentaire **else**. Considérez l'exemple suivant :

```
default {
    state_entry() {
        integer i = TRUE;
        if (i == TRUE) {
            llSay(0, "La valeur est vraie");}
        else {
            llSay(0, "La valeur est fausse");}
    }
}
```

Si l'expression dans la parenthèse est vraie l'instruction

```
llSay(0, "La valeur est vraie");
```

est exécutée. Le contrôle continue ensuite avec l'instruction suivante, le **else** est ignoré. Si l'expression était fausse ce serait évidemment l'inverse, le **if** serait ignoré et l'expression

```
llSay(0, "La valeur est fausse");
```

serait exécutée. Il est possible d'avoir plusieurs instructions dans les blocs de **if** et de **else**. S'il n'y a qu'une instruction à exécuter il est possible d'omettre les accolades. Le code précédent peut ainsi être simplifié :

```
default {
    state_entry() {
        integer i = TRUE;
        if (i) llSay(0, "La valeur est vraie");
        else llSay(0, "La valeur est fausse");}
}
```

L'instruction **else** est évidemment optionnelle et il est fréquent de rencontrer des **if** sans **else** dans les scripts. Remarquez également que je me suis contenté de tester la valeur de la variable **i** pour l'instruction **if**, en effet le fait de mettre cette variable comme argument de **if** revient à tester si elle est égale à **TRUE**.

Certains scripteurs aiment les raccourcis et le code compact. Considérez cette ligne de code :

```
if (++i) {...}
```

La variable **i** est d'abord incrémentée d'une unité avant d'être testée. Sans cette astuce il faudrait écrire :

```
++i ;
if (i) {...}
```

Parfois ce genre de manipulation n'est pas faite pour clarifier le code :

```
if (i = a + b) {...}
```

On commence par affecter la variable **i** pour ensuite la tester. Ne pas confondre avec :

```
if (i == a + b) {...}
```

Dans ce cas on teste si la variable **i** est égale à la somme de **a** et **b**.

❶ Ne pas confondre l'opérateur d'affectation = avec l'opérateur de comparaison ==

Considérez maintenant le script suivant :

```
default {
    touch_start(integer total_number) {
        if (llDetectedKey(0) == llGetOwner())
            llSay(0, "Vous etes le proprietaire");
    }
}
```

Le **if** est utilisé sans **else**.

Il est possible d'imbriquer des instructions **if**. Comme dans ce code :

```
default {
    state_entry() {
        integer i = 12;
        if (i > 10)
            if (i < 100) llSay(0, "La valeur est superieure a 10 et inferieure à 100");
            else llSay(0, "La valeur est superieure a 100");
        else llSay(0, "La valeur est inferieure à 10");
    }
}
```

Nous avons deux **if** imbriqués.

❶ Remarquez qu'un **else** correspond au dernier **if** sans **else**. Il est nécessaire d'utiliser des accolades pour obtenir un fonctionnement différent.

Voici un autre exemple :

```
default {
    state_entry() {
        integer i = 12;
        if (i < 10) llSay(0, "La valeur est inferieure a 10");
        else if (i < 100) llSay(0, "La valeur est inferieure a 100");
        else llSay(0, "La valeur est superieure à 100");
    }
}
```


Nous avons une combinaison **else if** qui permet d'évaluer une succession d'expressions.

Certaines limitations de **LSL** sont à connaître : seulement quatre niveaux d'imbrication sont autorisés et on ne peut avoir que 23 **else** au maximum pour un **if**.

7.6 Instructions d'itération

7.6.1 Instruction while

L'instruction **while** permet d'exécuter une instruction ou un groupe d'instruction zéro ou plusieurs fois selon la valeur d'une expression booléenne. Tant que l'expression est vraie les instructions sont exécutées. Lorsqu'elle est fausse l'exécution est passée à l'instruction suivante. Considérez le code suivant :

```
default {  
    state_entry() {  
        integer n = 6;  
        while(n > 1) {  
            llSay(0, (string)n);  
            n--;}  
    }  
}
```

Résultat :

```
new: 6  
new: 5  
new: 4  
new: 3  
new: 2
```

Le bloc d'instruction est exécuté tant que la valeur de variable **n** est supérieure à 1. Observez maintenant le code suivant :

```
default {  
    state_entry() {  
        integer n = 6;  
        while(n-- > 1){llSay(0, (string)n);}  
    }  
}
```

Cette fois nous obtenons à l'exécution :

```
new: 5  
new: 4  
new: 3  
new: 2
```

Une des caractéristiques d'une boucle **while** est que le test est effectué en entrée de boucle, il est donc possible que les instructions ne soient jamais exécutées si l'expression booléenne est fausse.

7.6.2 Instruction do

Une instruction **Do** permet l'exécution d'une instruction ou d'un groupe d'instructions tant qu'une expression reste vraie. Contrairement à l'instruction **while** le test s'effectue en sortie de boucle. La ou les instructions sont donc exécutées au moins une fois :

```
default {
    state_entry() {
        integer n = 6;
        do {
            llSay(0, (string)n);
            n--;
        } while(n > 1);
    }
}
```

Résultat :

```
new: 6
new: 5
new: 4
new: 3
new: 2
```

7.6.3 Instruction for

La boucle **for** répète une instruction ou un bloc d'instructions jusqu'à ce qu'une expression soit fausse. Considérez le code suivant :

```
default {
    state_entry() {
        integer n;
        for(n = 1; n < 5; n++) {
            llWhisper(0, (string)n);
        }
    }
}
```

Résultat :

```
new whispers: 1
new whispers: 2
new whispers: 3
new whispers: 4
```

La variable **n** est initialisée avec la valeur 1. A chaque boucle elle est incrémentée de 1. On reste dans la boucle tant que **n** est inférieur à 5. L'instruction **for** comporte 3 parties :

```
for(initialisation; test; actualisation)
```

Généralement on utilise une variable de type **integer** que l'on initialise dans l'élément **initialisation**. L'élément **test** sert à vérifier la valeur de cette variable à chaque boucle, il peut s'agir de n'importe quelle expression dont la valeur est de type **integer**. L'élément **actualisation** enfin sert à modifier la valeur de la variable, en général une simple incrémentation ou décrémentation. Observez cet autre exemple :

```
default {  
    state_entry() {  
        integer n;  
        for(n = 20; n > 0; n-=5) {  
            llWhisper(0, (string)n);}  
        }  
    }  
}
```

Résultat :

```
new whispers: 20  
new whispers: 15  
new whispers: 10  
new whispers: 5
```

Cette fois la variable n est diminuée à chaque boucle de 5. Le test consiste à vérifier que la variable est toujours positive. Considérez maintenant ce script :

```
default {  
    state_entry() {  
        string s = "abcde";  
        integer n;  
        for(n = 0; n < llStringLength(s); n++)  
            llWhisper(0, llGetSubString(s, n, n));  
    }  
}
```

Résultat :

```
new whispers: a  
new whispers: b  
new whispers: c  
new whispers: d  
new whispers: e
```

Le but est d'extraire les caractères d'une chaîne. Une boucle **for** est tout à fait adaptée pour ce genre d'opération. Remarquez au passage que nous avons négligé de mettre des accolades étant donné qu'il n'y a qu'une seule ligne de code dans la boucle. Par contre ce script comporte une erreur fréquente qu'il vaut mieux éviter. Au niveau du test le fait d'utiliser une fonction oblige à chaque boucle à refaire le calcul du nombre de caractères dans la chaîne, ce qui alourdit l'exécution. Voici une meilleure version de ce script :

```
default {  
    state_entry() {  
        string s = "abcde";  
        integer total = llStringLength(s);
```

```

integer n;
for(n = 0; n < total; n++)
    IlWhisper(0, IlGetSubString(s, n, n));
}

```

Cette fois le calcul du nombre de caractères est effectué une seule fois et mémorisé dans la variable **total**.

Dans une instruction **for** le test est effectué avant l'exécution des instructions, comme dans une boucle **while**, il est donc possible que la boucle ne soit jamais parcourue.

7.7 Instructions de saut

7.7.1 Instruction jump

La principale instruction de saut de LSL est l'instruction **jump** qui permet un branchement inconditionnel au niveau d'une étiquette nommée. Nous avons vu un exemple d'utilisation de cette instruction au point 7.1 ci-dessus.

7.7.2 Instruction return

L'instruction **return** retourne le contrôle de l'exécution à l'appelant de la fonction dans laquelle est située cette instruction. Si la fonction ne retourne aucune valeur l'instruction **return** est utilisée isolément. Dans le cas où la fonction doit retourner une valeur celle-ci doit figurer après l'instruction **return**. Il peut s'agir directement d'une valeur ou alors d'une variable contenant la valeur à retourner.

7.7.3 Instruction state

L'instruction **state** provoque un changement d'état. Elle doit être suivie du nom de l'état qui doit être activé.

8. Etats et événements

Un script **LSL** est un ensemble d'états. Il possède au moins un état par défaut nommé **default**. Cet état doit être le premier rencontré dans un script sinon une erreur de compilation est obtenue. Un état comporte au moins un événement.

Il est possible de créer des états pour les besoins d'un script avec le mot-clé **state**. A noter toutefois une limitation : un état ne peut pas s'appeler lui-même. D'une façon générale **LSL** n'autorise pas la récursivité. D'autre part le nombre maximum d'états possibles dans un script est de 144 y compris l'état par défaut, ce qui ne représente pas une limitation bien contraignante.

8.1 state_entry et state_exit

Lors de l'entrée dans un état l'événement **state_entry** (entrée dans l'état) se déclenche et permet d'effectuer les initialisations nécessaires pour cet état. Les exemples de code des chapitres précédents comportent un état par défaut **default** et l'événement **state_entry**. La sortie de l'état déclenche l'événement **state_exit** (sortie de l'état) qui permet de libérer des ressources avant la sortie de l'état. Considérez l'exemple suivant :

```
default {
    state_entry() {llSetTimerEvent(60.0);}
    timer() {state action;}
    state_exit() {llSetTimerEvent(.0);}
}
state action {
    state_entry() {
        ...
    }
    ...
}
```

Vous trouvez l'état par défaut **default** et un autre état défini à l'aide du mot clé **state** suivi du nom de l'état. A l'entrée dans l'état par défaut **default** l'événement **state_entry** se déclenche, on initialise alors un **timer** à 60 secondes avec la fonction **llSetTimerEvent**. L'événement timer se déclenche à l'issue de ce délai. On passe alors à l'état **action** à l'aide de l'instruction de changement d'état **state**. Mais avant d'entrer dans le nouvel état **action** l'événement **state_exit** de l'état **default** se déclenche, on peut alors désactiver la temporisation. Sans cela la temporisation continuerait à fonctionner et à générer l'événement **timer** toutes les 60 secondes. Pour le nouvel état **action** l'événement **state_entry** est activé... Dans cet exemple nous temporisons donc une action de 60 secondes en utilisant les états de **LSL**.

Avec le recul on se rend compte que l'événement **state_exit** est très peu utilisé au niveau des scripts. Dans l'exemple précédent il est possible de s'en passer en désactivant la temporisation au niveau de l'événement **timer**, avant la commande de changement d'état. Il génère pourtant un code plus propre. On remarque également un style de programmation qui évite les états avec l'utilisation de variables globales pour obtenir le même résultat. Cette approche est contraire à l'esprit de **LSL** sans compter l'occupation en mémoire des variables globales et le manque de lisibilité du code obtenu.

① Une erreur fréquente consiste à penser que l'événement **state_entry** est déclenché lorsqu'un objet est créé à partir de l'inventaire, ce qui n'est pas le cas. Lorsque vous mettez un objet dans votre inventaire l'état actif de votre script est sauvegardé. Et lorsque vous le déposez dans le jeu cet état est donc actif et l'événement **state_entry** ne se déclenche pas. Si vous avez besoin d'initialisations à ce moment là passez par l'événement **on_rez** décrit à la page 70.

8.2 touch_start touch et touch-end

Dans LSL il y a des événements plus importants que d'autres, ceux qui concernent l'interception d'un clic de souris sont très souvent utilisés. Il y a en fait trois événements :

touch_start	Déclenché au premier clic sur un objet
touch	Déclenché pendant toute la durée du clic
touch-end	Déclenché à la fin du clic

De ces trois états **touch_start** est utilisé le plus couramment. Cet événement comporte un paramètre qui indique le nombre d'avatars qui ont cliqué sur l'objet qui comporte le script. Considérez cet exemple :

```
default {
    touch_start(integer total_number) {
        llSay(0, "Nombre de clics : "+(string)total_number);}
}
```

L'événement **touch_start** se déclenche lors d'un clic ou de plusieurs clics sur l'objet, on affiche alors un texte dans le **Chat** qui nous indique le nombre de clics. Lors d'un clic sur un objet il est souvent nécessaire de connaître l'identité de l'avatar qui a cliqué. Il existe plusieurs fonctions qui permettent d'obtenir des informations sur l'avatar détecté :

```
default {
    touch_start(integer total_number) {
        llSay(0, "Clef de l'avatar : " + (string)llDetectedKey(0));
        llSay(0, "Nom de l'avatar : " + (string)llDetectedName(0));
        llSay(0, "Position de l'avatar : " + (string)llDetectedPos(0));
        llSay(0, "Rotation de l'avatar : "+ (string)llDetectedRot(0));
        llSay(0, "Vitesse de l'avatar : " + (string)llDetectedVel(0));
        llSay(0, "Avatar et objet de meme groupe ? : " + (string)llDetectedGroup(0));}
}
```

Lorsque des objets sont liés la règle est la suivante : si l'objet enfant contient un script avec un événement **touch_start** celui-ci est déclenché, s'il ne comporte pas de script ou un script sans événement **touch_start** le clic est transmis au **root** qui peut l'intercepter. Si vous voulez que le clic soit transmis au **root** même si l'événement est intercepté par un enfant vous devez utiliser la fonction **llPassTouches**.

L'événement **touch_end** se rencontre beaucoup moins souvent que son symétrique **touch_start**. En effet on veut plus souvent savoir quand un objet est cliqué que à quel moment il a cessé de l'être. Dans ma chasse au lag j'utilise cette fonction dans les situations où un clic provoque un changement dans mon objet (comme un changement d'animation) et où j'ai également besoin d'afficher un menu pour changer des options. A ce moment là je veux faire dire à mon click sur l'objet plusieurs choses. La solution généralement adoptée dans ce cas-là est d'utiliser un canal de Chat pour déclencher le menu du genre */9Options*. Mais ça oblige à laisser un **listen** en action ce qui ne me satisfait pas au niveau du lag. Une solution consiste à utiliser un délai dans le click.

Par exemple au-delà de 2 secondes de maintien du click c'est que l'utilisateur désire le menu. Voici le style de script à utiliser :

```
Default {
    touch_start(integer total_number) {
        ...
        llResetTime();
    }
    touch_end(integer total_number) {
        if (llGetTime() > 2.0) {
            llDialog(avatar, "\n\nChoose an option", menu, canalMenu);
        }
    }
}
```

Lors de l'événement **touch_start** on initialise le temps du script avec la fonction **llResetTime**. Lorsque le bouton de la souris est relâché, ce qui nous est indiqué par l'événement **touch_end**, on vérifie avec la fonction **llGetTime** le temps qui s'est écoulé depuis le click. S'il est supérieur à 2 secondes on affiche le menu.

De nouvelles fonctions sont venues enrichir nos possibilités d'action au niveau de ces événements. Jusque là on avait surtout des informations sur l'avatar qui clique et aucune sur l'objet touché. Tout cela a changé, nous allons analyser ces nouvelles possibilités.

8.2.1 Détecter le numéro de la face touchée

Il est possible de connaître la face de l'objet touchée avec la fonction **llDetectedTouchFace**. Elle renvoie un **integer** contenant le numéro de la face cliquée. Jusque là connaître le numéro de la face d'un objet n'était pas forcément toujours évident, en particulier pour les objets un peu torturés. Avec cette nouvelle fonction et un simple script c'est devenu tout simple :

```
default {
    touch_start(integer total_number) {
        llOwnerSay("Face numero : " + (string)llDetectedTouchFace(0));
    }
}
```

Connaître ce numéro est nécessaire pour un certain nombre d'actions, changer la couleur, la valeur de l'alpha et bien d'autres choses encore avec la fonction **llSetPrimitiveParams**.

8.2.2 Connaître l'orientation de la face touchée

Il est maintenant aussi possible de connaître l'orientation dans l'espace de la face touchée. Nous avons deux fonctions pour cela. La première est **llDetectedTouchNormal**. Elle renvoie un vecteur indiquant la perpendiculaire à la face. La seconde est **llDetectedTouchBinormal** qui renvoie quant à elle un vecteur représentant la tangente à la face. Prenons un exemple avec ce script :

```
default {
    touch_start(integer total_number) {
        llOwnerSay("Face numero : " + (string)llDetectedTouchNormal(0));
        llOwnerSay("Face numero : " + (string)llDetectedTouchBinormal(0));
    }
}
```

Pour un cube sans rotation en cliquant la face numéro 3 j'obtiens :

Object: normale : <0.00000, 1.00000, 0.00000>

Object: tangente : <0.00000, 0.00000, 1.00000>

La face numéro 3 est celle qui est du côté des Y croissants. On trouve tout naturellement un vecteur avec la valeur y à 1 pour la fonction **IIDetectedTouchNormal**. Quant à la fonction **IIDetectedTouchBinormal** elle renvoie un vecteur avec z à 1, ce qui représente bien la tangente à cette face.

En faisant subir une rotation positive de 45° sur l'axe X, j'obtiens les résultats suivants :

Object: normale : <0.00000, 0.70710, 0.70712>

Object: tangente : <0.00000, -0.70712, 0.70710>

Ce qui est conforme à ce qu'on peut attendre de ces fonctions. Le seul intérêt de ce type de fonction se situe dans un cadre de traitement dynamique pour connaître par exemple la direction d'une impulsion à donner.

8.2.3 Connaître l'emplacement du click

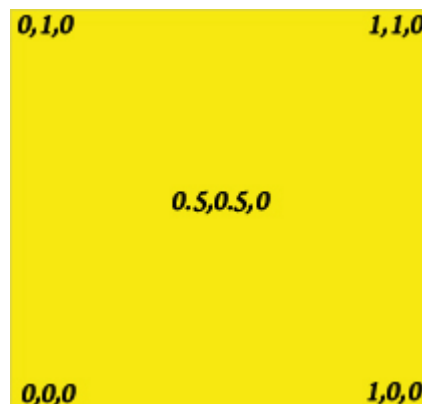
La fonction **IIDetectedTouchPos** renvoie un vecteur qui indique la position du point touché sur la face en coordonnées globales. Avant cette fonction la seule information qu'on pouvait obtenir était la position du centre géométrique de l'objet. Maintenant on peut affiner cette approche en connaissant exactement l'emplacement du point cliqué. Vous pouvez vous en rendre compte avec ce script :

```
default {
    touch_start(integer total_number) {
        IIOwnerSay("Position : " + (string)IIDetectedTouchPos(0));}
}
```

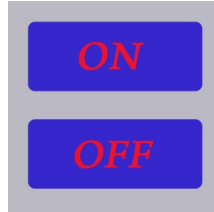
La fonction **IIDetectedTouchST** renvoie aussi un vecteur mais cette fois on obtient les coordonnées locales de la face concernée. Comme il s'agit d'un espace à deux dimensions seules deux valeurs sont utilisées dans le vecteur, x et y, correspondant respectivement aux valeurs horizontales et verticales avec une plage allant de 0. à 1. La valeur de z est sans signification et toujours égale à 0. Voici un script pour tester cette fonction :

```
default {
    touch_start(integer total_number) {
        IIOwnerSay("Position : " + (string)IIDetectedTouchST(0));}
}
```

Voici ce que vous devez obtenir comme genre de résultat :



On peut donc connaître très exactement l'emplacement où l'on clique, ce qui ouvre des perspectives intéressantes pour des tableaux de commande par exemple qui nécessitaient avant plusieurs prims. Il suffit de quelques lignes de code pour agir en conséquence. Prenons un exemple simple avec un objet et deux boutons :



On se contente d'appliquer la texture sur une face de l'objet et de créer un petit script pour déterminer le bouton cliqué :

```
default {  
    touch_start(integer total_number) {  
        vector v = llDetectedTouchST(0);  
        if(v.x > .1 && v.x < .9 && v.y > .1 && v.y < .4) llOwnerSay("Bouton OFF");  
        else if(v.x > .1 && v.x < .9 && v.y > .6 && v.y < .9) llOwnerSay("Bouton ON");  
    }  
}
```

Une application séduisante serait un sélecteur de couleurs basé sur ce principe et évidemment des jeux.

❶ La multiplication des boutons peut rapidement rendre laborieuse l'écriture du code. Il semble plus judicieux de créer un maillage systématique. Imaginez que vous divisez une face d'une boîte en rangées et colonnes uniformes dans ce genre avec un repérage numérique des zones :

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Le codage devient alors très simple :

```
integer touchZone(integer x, integer y) {  
    vector v = llDetectedTouchST(0);  
    return (integer)(v.x * x) + ((integer)(v.y * y)) * x;  
}  
default {  
    touch_start(integer total_number) {  
        llOwnerSay("Clic sur la zone " + (string)touchZone(4, 4));  
    }  
}
```

La dernière nouvelle fonction est **llDetectedTouchUV** qui ressemble beaucoup à la précédente puisqu'elle renvoie un vecteur avec seulement les deux premières valeurs significatives appelées cette fois u et v. Mais au lieu d'obtenir l'emplacement sur la face on obtient l'emplacement sur la texture, ce qui n'est pas toujours la même chose. En effet tant qu'on n'a fait subir aucune modification à la texture et qu'elle occupe l'emplacement exact de la face les deux fonctions renvoient les mêmes valeurs mais si on a modifié la répétition ou un offset les

valeurs deviennent différentes. En effet la référence est maintenant la texture indépendamment de son emplacement sur la face concernée. Voici un petit script pour tester tout ça :

```
default {
    touch_start(integer total_number) {
        llOwnerSay("llDetectedTouchUV renvoie " + (string)llDetectedTouchUV(0));
        llOwnerSay("llDetectedTouchST renvoie " + (string)llDetectedTouchST(0));}
}
```

Je fais subir une répétition de 0.5 horizontale et verticale à ma texture. Voici ce que j'obtiens avec un click dans la partie inférieure gauche, vers le milieu et en haut à droite :

[1:59] Object: llDetectedTouchUV renvoie <0.27562, 0.27933, 0.00000>

[1:59] Object: llDetectedTouchST renvoie <0.05123, 0.05866, 0.00000>

[1:59] Object: llDetectedTouchUV renvoie <0.50225, 0.53394, 0.00000>

[1:59] Object: llDetectedTouchST renvoie <0.50450, 0.56787, 0.00000>

[2:00] Object: llDetectedTouchUV renvoie <0.74134, 0.74258, 0.00000>

[2:00] Object: llDetectedTouchST renvoie <0.98268, 0.98516, 0.00000>

Très logiquement les valeurs s'étendent de 0.25 à 0.75 étant donné la répétition imposée. J'applique maintenant un offset de -0.25 aux deux dimensions. Voici les nouvelles valeurs obtenues :

[2:03] Object: llDetectedTouchUV renvoie <0.02663, 0.02585, 0.00000>

[2:03] Object: llDetectedTouchST renvoie <0.05327, 0.05171, 0.00000>

[2:03] Object: llDetectedTouchUV renvoie <0.24467, 0.26897, 0.00000>

[2:03] Object: llDetectedTouchST renvoie <0.48935, 0.53795, 0.00000>

[2:03] Object: llDetectedTouchUV renvoie <0.49387, 0.49036, 0.00000>

[2:03] Object: llDetectedTouchST renvoie <0.98775, 0.98073, 0.00000>

Maintenant j'ai une plage qui va de 0 à 0.5.

Si on applique une répétition de 2 pour les deux dimensions et un offset de 0.5 pour recadrer la texture voilà les valeurs obtenues :

[2:04] Object: llDetectedTouchUV renvoie <0.22846, 0.16253, 0.00000>

[2:04] Object: llDetectedTouchST renvoie <0.11422, 0.08126, 0.00000>

[2:04] Object: llDetectedTouchUV renvoie <0.99386, 1.09771, 0.00000>

[2:04] Object: llDetectedTouchST renvoie <0.49692, 0.54885, 0.00000>

[2:04] Object: llDetectedTouchUV renvoie <1.94902, 1.95729, 0.00000>

[2:04] Object: llDetectedTouchST renvoie <0.97450, 0.97864, 0.00000>

On se rend compte que la plage s'étend maintenant de 0 à 2 et correspond au nombre de répétitions.

8.3 listen

L'événement **listen** est mis en oeuvre par l'intermédiaire d'une instruction **llListen** comme dans l'exemple suivant :

```
integer écoute;
default {
    touch_start(integer total_number) {
        écoute = llListen(0, "", NULL_KEY, "coucou");
    }
    listen(integer channel, string name, key id, string message) {
        llListenRemove(écoute);
        llSay(0, "Merci de me saluer " + name);
        llSay(0, "Vous m'avez dit " + message); }
}
```

Dans cet exemple l'objet attend un clic. Lors de celui-ci une écoute du **Chat** général est activée pour le message « coucou », dès que ce message est détecté au niveau du **Chat** l'événement **listen** se déclenche, l'écoute est désactivée et un message est affiché au niveau du Chat.

La fonction **llListen** comporte 4 paramètres. Le premier concerne le canal du **Chat** sur lequel s'effectue l'écoute. Il y a 2147483647 canaux (limite du type **integer**). Le canal 0 est particulier, c'est celui utilisé pour l'affichage pour tous les avatars, donc la fenêtre du **Chat** que vous utilisez pour discuter avec les autres utilisateurs. La constante **PUBLIC_CHANNEL** a justement la valeur 0. Le fait d'utiliser cette constante au lieu de la valeur 0 permet au code d'être plus explicite. Les trois derniers paramètres sont des filtres pour l'écoute :

Paramètre	Nom	Fonction
1	name	Nom de l'objet ou de l'avatar
2	id	clé de l'objet ou de l'avatar
3	msg	message attendu

Il est important d'utiliser des filtres pour être sélectif, éviter des mélanges de messages et pour alléger la charge du simulateur.

① Une écoute doit être désactivée lorsqu'elle est devenue inutile. La fonction **llListen** renvoie une valeur **integer** qui référence l'écoute mise en place. Il faut mémoriser cette valeur, comme dans l'exemple ci-dessus, pour pouvoir désactiver l'écoute avec la fonction **llListenRemove** qui prend justement comme paramètre une valeur **integer** qui référence une écoute.

Pour envoyer un message sur le Chat il existe plusieurs fonctions de portée croissante :

Fonction	Portée
llWhisper	10 mètres
llSay	20 mètres
llShout	100 mètres
llRegionSay	Région

8.4 timer

Il arrive fréquemment dans un script d'avoir besoin d'une temporisation. Une façon simple d'obtenir un délai dans l'exécution est d'insérer une instruction **IISleep** qui interrompt l'exécution d'une durée dont la valeur est déterminée par l'unique paramètre de cette fonction. Cette possibilité résout les cas simples de délai dans un script. Pour les situations qui réclament un traitement plus élaboré il faut utiliser l'événement **timer**. Nous avons vu un exemple de l'utilisation de cet événement au point 8.3 ci-dessus. Il est activé au moyen de la fonction **IISetTimerEvent** qui prend en paramètre la valeur de la temporisation. Une fois celle-ci en action l'événement se déclenche indéfiniment chaque fois que le terme de la temporisation est atteint. Pour désactiver la temporisation il faut utiliser la même fonction en passant la valeur **0.0** en paramètre.

8.5 sensor et no_sensor

Il existe plusieurs systèmes de détection dans **SL**, l'un d'eux est le **sensor** qui permet de repérer un ou plusieurs objets ou avatars dans un certain espace environnant. L'événement **sensor** est mis en œuvre à l'aide d'une des deux fonctions **IISensor** ou **IISensorRepeat**. La différence entre les deux est que la première effectue une seule détection alors que la seconde assure une détection cyclique. Voici un exemple élémentaire :

```
default {
    state_entry() {IISensorRepeat("", NULL_KEY, AGENT, 10, PI, 2.0);}
    sensor(integer total_number) {ISay(0, "Bonjour " + IIDetectedName(0));}
}
```

Dans ce **script** un **sensor** est mis en place pour détecter les avatars dans une sphère de 10 mètres de rayon. La détection a lieu toutes les 2 secondes. Lorsqu'un avatar est détecté un message d'accueil lui est envoyé sur le **Chat**.

Les fonctions **IISensor** ou **IISensorRepeat** ont 5 paramètres en commun :

Paramètre	Nom	Fonction
1	name	Filtre sur le nom de l'avatar ou l'objet à détecter
2	id	Filtre sur la clé de l'avatar ou l'objet à détecter
3	type	Type d'élément à détecter
4	range	Distance de détection en mètres
5	arc	Arc de détection en radians

Un certain nombre de constantes permettent de simplifier l'écriture des scripts. La constante **NULL_KEY** définit une valeur nulle de clé, c'est-à-dire qu'aucun filtre n'est établi au niveau de la clé. En ce qui concerne le type d'objet les constantes suivantes sont à utiliser :

Constante	Signification
AGENT	Objets qui sont des Agents
ACTIVE	Objets qui ont un script actif ou qui ont un mouvement physique
PASSIVE	Objets statiques
SCRIPTED	Objets scriptés

Ces constantes peuvent être combinées avec l'opérateur **|**. Pour le paramètre de l'arc les constantes suivantes peuvent aussi être utilisées :

Constante	Valeur	Cercle
PI	Pi	Cercle

PI_BY_TWO	Pi / 2	Demi-cercle
-----------	--------	-------------

La fonction **llSensorRepeat** possède un sixième paramètre de type **float** qui représente la période de détection en secondes.

Au niveau de l'événement **sensor** nous trouvons un seul paramètre qui représente le nombre d'éléments détectés. Les informations sur l'élément détecté peuvent être récupérées à l'aide des fonctions de détection :

Fonction	Élément détecté
llDetectedName	Nom
llDetectedKey	Clé
llDetectedOwner	Propriétaire de l'objet
llDetectedType	Type : AGENT, ACTIVE, PASSIVE ou SCRIPTED
llDetectedPos	Position
llDetectedRot	Rotation
llDetectedGroup	Groupe d'appartenance
llDetectedVel	Vélocité
llDetectedGrab	Grab offset

❶ Ces éléments de détection sont aussi actifs pour les événements de **touch** et de **collision**. Par contre ils ne sont pas actifs pour tous les autres événements !

L'événement **no_sensor** est déclenché lorsque aucun élément n'est détecté, ce qui risque d'être plutôt fréquent dans des zones désertes ! Cet événement est très rarement utilisé.

Un cas d'utilisation fréquent des **sensors** est celui des radars et autres instruments destinés à repérer des objets ou avatars. Supposez que vous désirez connaître dans quelle direction est le plus proche avatar. Vous mettez en œuvre un **sensor** et un objet qui s'oriente dans la direction de l'avatar détecté. Voici un script tout simple qui peut réaliser cela :

```
default{
    state_entry(){llSensorRepeat("", NULL_KEY, AGENT, 10.0, PI_BY_TWO, 5.0);}
    sensor(integer total_number) {llLookAt(llDetectedPos(0), 1.0, 1.0);}
}
```

On met en place un sensor qui détecte les avatars, on règle la distance de détection à 10 mètres, l'angle à 180 degrés et la fréquence de détection à 5 secondes. Dans l'événement **timer** on récupère la position du premier avatar détecté avec la fonction **llDetectedPos** et on oriente l'objet qui contient le script avec la fonction **llLookAt** (voir point 3.6).

A noter une limitation des **sensors** qui ne peuvent détecter que 16 objets ou avatars au maximum sur une portée maximale de 96 mètres. D'autre part les **sensors** sont gourmands en ressources au niveau des simulateurs et contribuent au lag, il ne faut donc les utiliser qu'avec modération. Pour une détection il vaut mieux utiliser si possible des collisions avec un objet fantôme invisible par exemple.

Voici un exemple complet de radar :

```
//
```

```

// Variables de paramétrage
// -----
list      menuMaster = ["Exit", "Scan", "List", "Die"];
list      etapes      = [
    <64.0,64.0,20.0>,
    <64.0,192.0,20.0>,
    <192.0,64.0,20.0>,
    <192.0,192.0,20.0>];
integer    division    = 3;
float      distance    = 96.0;
// -----
// Variables de travail
// -----
integer     canalMenu;
integer     ecouteMenu;
integer     etape;
integer     sousEtape;
integer     scanEnCours;
rotation    rot;
vector      myPos;
float       angle;
list        clefs;
list        noms;
list        positions;
// -----
// Gestion de l'écoute du menu
// -----
updateEcoute(key id) {
    cancelEcoute();
    canalMenu = genCanal();
    ecouteMenu = IListen(canalMenu, "", id, "");
    ISetTimerEvent(30.0);}
// -----
cancelEcoute() {
    if(ecouteMenu > 0) {
        ISetTimerEvent(.0);
        IListenRemove(ecouteMenu);
        ecouteMenu = 0;}
}
// -----
endMenu() {
    ISetTimerEvent(.0);
    cancelEcoute();
    IOwnerSay("Délai écoulé, veuillez recliquer pour obtenir un nouveau menu");}
// -----
// Canal aléatoire
// -----
integer genCanal() {return IFloor(IFrand(2000000.0)) + 100;}
// -----
// Translation supérieure à 10 mètres

```

```
// -----
object_move_to(vector position) {
    vector last;
    do {
        last = llGetPos();
        llSetPos(position);
    } while ((llVecDist(llGetPos(),position) > 0.001) && (llGetPos() != last));
}
// -----
//      Test de position
// -----
testPos(vector pos) {
    if(llGetPos() == pos)
        llOwnerSay("Scan à la position " + (string)pos);
    else
        llOwnerSay("Impossible d'atteindre la position " + (string)pos + ", scan à la position " +
(string)llGetPos());}
// -----
//      Traitement détection
// -----
detect() {
    if(++sousEtape < division) {
        llSetRot(rot * llGetRot());
        llSensor("", NULL_KEY, AGENT, distance, angle);}
    else {
        if(++etape < llGetListLength(etapes)) {
            sousEtape = 0;
            vector v = llList2Vector(etapes, etape);
            object_move_to(v);
            testPos(v);
            llSensor("", NULL_KEY, AGENT, distance, angle);}
        else {
            object_move_to(myPos);
            scanEnCours = FALSE;
            llSetAlpha(1.0, ALL_SIDES);
            llOwnerSay("Fin du scan");
            updateEcoute(llGetOwner());
            llDialog(llGetOwner(), "\nChoisissez une option", menuMaster, canalMenu);}
    }
}
// -----
//      Etat par défaut
// -----
default
{
    state_entry() {llSetStatus(STATUS_PHANTOM, TRUE);}

    listen(integer channel, string name, key id, string message) {
        if(channel == canalMenu) {
            cancelEcoute();
```

```

        if(message == "Die")
            llDie();
        else if(message == "List") {
            integer n = llGetListLength(clefs);
            if(n) {
                if(n > 1)
                    llOwnerSay("Liste des " + (string)n + " avatars détectés :");
                else
                    llOwnerSay("Un seul avatar détecté :");
                integer c;
                for(c = 0; c < n; c++)
                    llOwnerSay(llList2String(noms, c) + " ("
                        + llList2String(clefs, c) + "), position = "
                        + llList2String(positions, c));}
            else
                llOwnerSay("La liste est vide !");}
        else if(message == "Scan") {
            llSetAlpha(.0, ALL_SIDES);
            scanEnCours = TRUE;
            clefs = [];
            noms = [];
            positions = [];
            etape = 0;
            sousEtape = 0;
            angle = PI_BY_TWO;
            rot = llEuler2Rot(<.0, .0, angle>);
            myPos = llGetPos();
            llOwnerSay("Le scan commence, soyez un peu patient");
            vector v = llList2Vector(etapes, 0);
            object_move_to(v);
            testPos(v);
            llSensor("", NULL_KEY, AGENT, distance, angle);}
    }
}

touch_start(integer total_number) {
    if(llDetectedKey(0) != llGetOwner() || scanEnCours) return;
    updateEcoule(llGetOwner());
    llDialog(llGetOwner(), "\nChoisissez une option", menuMaster, canalMenu);}

sensor(integer total_number) {
    integer c;
    key k;
    for(c = 0; c < total_number; c++) {
        k = llDetectedKey(c);
        if(llListFindList(clefs, [k]) == -1) {
            clefs += k;
            noms += llDetectedName(c);
            positions += llDetectedPos(c);}
    }
}

```



```
        detect();}

    no_sensor()    {detect();}

    timer() {endMenu();}
}
```

L'objet est paramétrable. La **list etapes** contient les destinations successives du radar. Vous pouvez les changer et en ajouter. Pour le reste je vous laisse analyser le code.

8.6 changed

Cet événement se déclenche lors d'un changement au niveau d'un objet. Voici un exemple exhaustif des possibilités de cet événement :

```
default {
    changed(integer change) {
        if (change & CHANGED_LINK)
            llSay(0, "Changement dans une liaison");
        else if (change & CHANGED_ALLOWED_DROP)
            llSay(0, "Ajout dans l'inventaire");
        else if (change & CHANGED_COLOR)
            llSay(0, "Changement de la couleur");
        else if (change & CHANGED_INVENTORY)
            llSay(0, "Changement dans l'inventaire");
        else if (change & CHANGED_OWNER)
            llSay(0, "Changement de propriétaire");
        else if (change & CHANGED_REGION)
            llSay(0, "Changement de region");
        else if (change & CHANGED_SCALE)
            llSay(0, "Changement de dimension");
        else if (change & CHANGED_SHAPE)
            llSay(0, "Changement de forme : boite, cylindre...");
        else if (change & CHANGED_TELEPORT)
            llSay(0, "Teleportation");
        else if (change & CHANGED_TEXTURE)
            llSay(0, "Changement de texture");}
}
```

Essayez par exemple le script suivant :

```
default {
    state_entry() {llSetColor(<1.0,1.0,.0>, ALL_SIDES);}
    changed(integer change) {
        if (change & CHANGED_COLOR)
            llOwnerSay("Couleur changee !!!");}
}
```

Le changement dans les liaisons est utilisé en particulier pour détecter qu'un avatar s'assoit sur un objet. En effet dans ce cas l'avatar se trouve lié à l'objet qui détecte l'ajout d'une liaison donc un changement. Il est alors possible de détecter la clé de l'avatar par l'intermédiaire de la fonction **IIAvatarOnSitTarget**.

Il est possible de combiner des valeurs avec l'opérateur |.

Le point suivant donne un autre exemple de l'utilisation de cet événement.

8.7 run_time_permissions et control

Une permission est nécessaire pour animer un avatar, débiter de l'argent, ou capturer des contrôles de mouvement. Cet événement est mis en œuvre par la fonction **IIRequestPermissions** qui interroge l'avatar pour autoriser une action sur lui. L'événement **run_time_permissions** permet de récupérer les permissions éventuelles. C'est son unique paramètre qui nous renseigne à ce sujet. S'il est égal à 0 c'est qu'aucune permission n'est accordée, dans le cas contraire il faut analyser cette valeur au niveau des bits constituant pour déterminer la permission accordée. L'événement **control** se déclenche lorsqu'une permission de prise de contrôle des commandes de l'avatar a été accordée. Voici un exemple simple :

```
default {
    touch_start(integer total_number) {
        key avatar = IIDetectedKey(0);
        IIRequestPermissions(avatar, PERMISSION_TAKE_CONTROLS);
    }
    run_time_permissions(integer permissions)
    if(permissions & PERMISSION_TAKE_CONTROLS)
        IITakeControls(CONTROL_FWD, TRUE, FALSE);
    control(key name, integer levels, integer edges) {
        if(levels & CONTROL_FWD)
            IIWhisper(0, "C'est moi qui controle votre marche en avant !");
    }
}
```

Lorsqu'un **avatar** clique sur l'objet qui contient ce script l'événement **touch_start** est déclenché. On récupère alors la clé de l'avatar et on utilise la fonction **IIRequestPermissions** pour lui demander l'autorisation de détourner ses contrôles clavier. La réponse de l'avatar déclenche l'événement **run_time_permissions**. L'unique paramètre de cet événement contient les permissions accordées. On teste la réponse de l'avatar et, si elle est positive, on utilise la fonction **IITakeControls** pour prendre le contrôle des touches passées en paramètres, en l'occurrence la marche avant. Désormais dès que l'avatar utilise cette touche l'événement **control** se déclenche. On peut alors effectuer les actions désirées.

La fonction **IIRequestPermissions** possède deux paramètres, le premier pour passer la clé de l'avatar concerné, le second la ou les autorisations demandées. Les constantes suivantes permettent de coder simplement ce paramètre :

Constante	Autorisation demandée	Fonction à utiliser
PERMISSION_DEBIT	Autorisation de débit	IIGiveMoney
PERMISSION_TAKE_CONTROLS	Contrôles disponibles	IITakeControls
PERMISSION_TRIGGER_ANIMATION	Autorisation d'animer l'avatar	IIStartAnimation
PERMISSION_ATTACH	Autorisation d'attacher un objet	IIAttachToAvatar
PERMISSION_CHANGE_LINKS	Autorisation de changer les liaisons	IICreateLink IIBreakLink IIBreakAllLinks

PERMISSION_TRACK_CAMERA	Autorisation de connaître la position et la rotation de la caméra	llGetCameraPos llGetCameraRot
PERMISSION_CONTROL_CAMERA	Autorisation de contrôler la caméra	llSetCameraParams

Ces constantes peuvent être combinées avec l'opérateur | .

La fonction **llRequestPermissions** n'arrête pas l'exécution du script dans l'attente de la réponse de l'avatar. Cette réponse déclenche l'événement **run_time_permissions**. Cet événement possède un seul paramètre qui renseigne sur les permissions accordées (voir le point 2.2.2.1). Le type de ce paramètre est un **integer**, autrement dit une variable de 32 bits. A chacun de ces bits correspond une valeur vraie ou fausse. Ce qui permet de stocker 32 valeurs booléennes. Pour extraire une de ces valeurs il faut utiliser un "et" logique, on parle de « masque » pour cette opération. C'est pour cette raison que vous trouvez dans le script d'exemple la ligne de code :

```
if(permissions & PERMISSION_TAKE_CONTROLS)
```

qui permet de tester le bit qui correspond à l'autorisation de prise de contrôle.

La fonction **llTakeControls** permet la prise de contrôle une fois que nous avons obtenu l'autorisation **PERMISSION_TAKE_CONTROLS**. Cette fonction possède 3 paramètres. Le premier définit le ou les contrôles que l'on veut intercepter. Voici la liste de constantes à utiliser :

Constante	Mouvement
CONTROL_FWD	Marche avant
CONTROL_BACK	Marche arrière
CONTROL_LEFT	Va à gauche
CONTROL_RIGHT	Va à droite
CONTROL_ROT_LEFT	Rotation vers la gauche
CONTROL_ROT_RIGHT	Rotation vers la droite
CONTROL_UP	Monte
CONTROL_DOWN	Descend
CONTROL_LBUTTON	Bouton gauche en "Mouse look"
CONTROL_ML_LBUTTON	Bouton droit en « Mouse look »

Les deux paramètres suivants acceptent des valeurs booléennes. Le deuxième indique si le contrôle est transmis à l'objet, le troisième si le contrôle est transmis à l'avatar.

Il est possible de libérer les contrôles avec la fonction **llReleaseControls**.

Nous avons vu l'événement **changed** au point 8.6, voici une utilisation de cet événement combiné à une demande d'autorisation pour animer un **avatar** :

```
string Animation = "sleep";
default
{
    state_entry()
    {
        llSetSitText("Sleep");
        llSitTarget(<0,0,1>, ZERO_ROTATION);
    }
    changed(integer change)
    {
        if (change & CHANGED_LINK)
```

```

key AgentKey = IIAvatarOnSitTarget();
if (AgentKey)
    IIRequestPermissions(AgentKey, PERMISSION_TRIGGER_ANIMATION);
else {
    IIStopAnimation(Animation);
    IIResetScript();}
}
}
run_time_permissions(integer permissions) {
    if(permissions & PERMISSION_TRIGGER_ANIMATION) {
        IIStopAnimation("sit");
        IIStartAnimation(Animation);}
    }
}
}

```

Dans cet exemple nous détectons un changement de liaison par le masque **CHANGED_LINK** au niveau du test dans l'événement **changed**. On récupère la clé de l'avatar pour lui demander une autorisation d'animation avec la constante **PERMISSION_TRIGGER_ANIMATION**. L'événement **run_time_permissions** nous permet alors d'animer l'avatar avec la fonction **IIStartAnimation**. Un nouveau changement de liaison sans obtention de clé d'avatar signifie que l'avatar s'est levé, on stoppe alors l'animation avec la fonction **IIStopAnimation**.

Pour un exemple de permission d'attachement se reporter au point 11.2.

8.8 money

Le débit d'argent sur **SL** passe obligatoirement par une autorisation comme nous l'avons vu au point précédent. Lorsqu'un paiement est demandé l'événement **money** est déclenché. Observez l'exemple suivant :

```

key avatar;
default
    touch_start(integer total_number) {
        IIRequestPermissions(IIGetOwner(), PERMISSION_DEBIT);
        avatar = IIDetectedKey(0);}
    run_time_permissions(integer permissions) {
        if(permissions & PERMISSION_DEBIT)
            IISetPayPrice(10, [10, PAY_HIDE, PAY_HIDE, PAY_HIDE]);}
    money(key giver, integer amount) {
        IIGiveMoney(avatar, 10);}
}

```

Ce script est destiné à un avatar généreux prêt à donner des lindens. Lors d'un clic sur l'objet qui contient ce script l'événement **touch_start** est déclenché. La fonction **IIRequestPermissions** permet de demander une autorisation de débit (constante **PERMISSION_DEBIT**) au propriétaire (**IIGetOwner**). On récupère également la clé de l'avatar qui a cliqué avec la fonction **IIDetectedKey**. et on la mémorise dans la variable **avatar**. Si le propriétaire accepte le paiement l'événement **run_time_permissions** se déclenche. Après un test au niveau de la permission avec le masque **PERMISSION_DEBIT** on utilise la fonction **IISetPayPrice** pour définir le paiement. La boîte de dialogue de paiement apparaît alors. Quand l'avatar clique sur le bouton de paiement l'événement **money** se déclenche. La fonction **IIGiveMoney** permet alors de réaliser le paiement. Notez que la fonction **IIGiveMoney** ne renvoie aucune information au niveau du script lorsque le crédit est insuffisant pour opérer le transfert d'argent.

Cet événement est aussi utilisé pour réaliser les **tipjars**. En voici un exemple simple :

```
integer totalDons;
string proprio;
default
{
    state_entry() {
        proprio = llKey2Name(llGetOwner());
        llSetText("Boîte de don de " + proprio + ". Tous les dons sont acceptes...\nClic droit sur moi et selection de Pay pour un don.", <1.0,1.0,1.0>, 1.0);
    }
    money(key giver, integer amount) {
        totalDons += amount;
        llSetText("Boîte de don de " + proprio + ". Tous les dons sont acceptes...\n$" + (string)totalDons + " de dons actuellement.\nClic droit sur moi et selection de Pay pour un don.", <1.0,1.0,1.0>, 1.0);
        llInstantMessage(giver, "Merci beaucoup pour votre don.");
        llInstantMessage(llGetOwner(), llKey2Name(giver) + " a fait un don de $" + (string)amount);
    }
}
```

8.9 on_rez et object_rez

Les événements **on_rez** et **object_rez** sont déclenchés lors de la création d'un objet, soit qu'il soit créé à partir d'un autre objet, soit qu'il soit déposé à partir de l'inventaire. L'événement **on_rez** est déclenché dans le prim créé alors que l'événement **object_rez** est déclenché dans le prim créateur de l'objet lorsqu'il y a création à partir d'un script situé dans ce prim. Considérez l'exemple suivant :

```
default {
    on_rez(integer start_param) {
        vector position = llGetPos();
        llWhisper(0, "Bonjour je suis a la position " + (string)llGetPos());
    }
}
```

Chaque fois que l'objet qui contient ce script est créé l'événement **on_rez** se déclenche et un message indique la position de l'objet.

Maintenant créez un objet que vous nommez « Module » dans lequel vous insérez le script suivant :

```
default {
    on_rez(integer start_param) {
        llWhisper(0, "Merci !!!");
    }
}
```

Mettez l'objet « Module » dans votre inventaire. Créez un second objet dans lequel vous insérez ce script :

```
default {
    touch_start(integer total_number) {
        llRezObject("Module", llGetPos() + <.0, .0, 1.0>, ZERO_VECTOR, ZERO_ROTATION, 0);
    }
}
```

```

object_rez(key id) {
    IISOwnerSay("L'objet que j'ai fait possède la clef : " + (string)id);}
}

```

Faites glisser l'objet « Module » dans l'inventaire de cet objet puis cliquez dessus. Avec la fonction **IISRezObject** un objet « Module » est créé, celui-ci voit son événement **on_rez** déclenché et un message est affiché. L'événement **object_rez** est déclenché dans l'objet créateur et celui-ci affiche la clef du nouvel objet. Il est très utile de connaître cette clef pour pouvoir gérer cet objet ultérieurement.

8.10 attach

Cet événement se déclenche lorsque l'objet qui contient le script est attaché à un avatar ou est détaché d'un avatar. Cet événement ne comporte qu'un paramètre qui contient la valeur de la clé de l'avatar dans le cas d'un attachement et la valeur de la constante **NULL_KEY** dans le cas d'un détachement. Observez le script suivant :

```

key avatar;
default
{
    attach(key attached) {
        if(attached == NULL_KEY)
            IISInstantMessage(avatar, "Vous m'avez perdu !!!");
        else {
            IISInstantMessage(attached, "Merci de me porter !");
            avatar = attached;
        }
    }
}

```

Ce script ne comporte que l'événement **attach**. Lorsque l'objet est attaché à un avatar le paramètre de l'événement comporte la clé de cet avatar. On lui envoie un message personnel avec la fonction **IISInstantMessage** et on mémorise sa clé dans la variable globale **avatar**. Lorsque l'objet est ensuite détaché l'événement se déclenche à nouveau mais avec une clé nulle. On envoie alors un nouveau message à l'**avatar**.

8.11 at_target et not_at_target

L'événement **at_target** se déclenche lorsqu'un objet « physique » est arrivé à une certaine distance de son objectif (**target**). Cet événement est activé par la fonction **IITarget**. Cet événement comporte trois paramètres :

Paramètre	Nom	Type	Fonction
1	tnum	integer	Identifiant de l'objectif à atteindre (target)
2	targetpos	vector	Localisation de l'objectif à atteindre
3	ourpos	vector	Position actuelle

```

integer targetID;
default {
    state_entry() {
        IISetStatus(STATUS_PHYSICS, TRUE);
        vector destination = IIGetPos() + <10.0, 10.0, .0>;
        targetID = IITarget(destination, 1.0);
        IIMoveToTarget(destination, 10.0);
    }
    at_target(integer tnum, vector targetpos, vector ourpos) {

```

```

        llWhisper(0, "Objectif atteint");
        llTargetRemove(targetID);
        llStopMoveToTarget();
        llSetStatus(STATUS_PHYSICS, FALSE);
    }
    not_at_target() {
        llWhisper(0, "Objectif non atteint");
    }
}

```

Un script très classique qui permet de voir en oeuvre les événements **at_target** et **not_at_target**. A l'entrée l'objet est transformé en objet « physique » avec la fonction **llSetStatus**. On détermine ensuite un vecteur représentant l'objectif à atteindre, en l'occurrence un point situé dans le plan XY en prenant un offset de 10 mètres sur ces deux axes. On définit ensuite le point de destination (**target**) avec la fonction **llTarget**. On mémorise un identifiant pour ce **target** dans la variable globale **targetID**. La fonction **llTarget** a deux paramètres : le premier est un vecteur pour la destination, le second est la proximité requise du **target** pour déterminer qu'on y est arrivé. Dans notre cas nous avons choisi un mètre. Ensuite la fonction **llMoveToTarget** met l'objet en mouvement. Cette fonction comporte deux paramètres, le premier donne la destination, le second le délai pour y parvenir. A partir de ce moment l'objet est en mouvement en direction du **target**.

Tant que l'objectif n'est pas atteint l'événement **not_at_target** se déclenche régulièrement et envoie un message. Dès que l'objet est à moins d'un mètre de la cible l'événement **at_target** se déclenche. A ce moment là on envoie un message pour indiquer que l'objet a bien atteint sa destination, on désactive l'événement avec la fonction **llTargetRemove**. On arrête le mouvement avec la fonction **llStopMoveToTarget**. Et on change le statut de l'objet en enlevant la propriété « physique » avec la fonction **llSetStatus**.

❶ Pour déplacer un objet non « physique » de façon fluide il est judicieux d'utiliser ce type de mouvement. En effet, la fonction **llSetPos** impose un délai de 0,2 secondes au niveau du script, ce qui ne permet pas d'obtenir un déplacement fluide. Une autre solution consiste à utiliser plusieurs scripts avec un décalage temporel, mais c'est une catastrophe au niveau du lag généré.

8.12 moving_start et moving_end

L'événement **moving_start** se déclenche normalement dès qu'un objet, qu'il soit « physique » ou non, se met en mouvement, ce qui signifie que sa position change. A l'inverse l'événement **moving_end** se déclenche lorsqu'un objet arrête de bouger, donc que sa position est constante. D'autre part ces deux événements sont déclenchés si l'objet entre dans un simulateur. Le problème est que cet événement semble sacrément bogué. J'ai essayé le script suivant sans trop de succès :

```

affichagePosition() {
    vector position = llGetPos();
    llSay(0, "Ma position actuelle est : " + (string)position);
}
default {
    touch_start(integer total_number) {
        llSetPos(llGetPos() + <5.0, .0, .0>);
    }
    moving_start() {
        llSay(0, "Je commence un mouvement");
        affichagePosition();
        llSleep(2.0);
    }
}

```

```

    }
    moving_end() {
        llSay(0, "Je viens de stopper");
        affichagePosition();
    }
}

```

Lorsque nous cliquons sur l'objet qui contient ce script l'événement **touch_start** est déclenché. Nous utilisons alors la fonction **llSetPos** pour modifier la position de l'objet : un décalage de 5 mètres sur l'axe X. Dès que l'objet se met en mouvement l'événement **moving_start** devrait se déclencher. On affiche alors sa position dans le Chat. La fonction **llSleep** permet de créer une temporisation, ici de 2 secondes, pour éviter la survenue trop rapide de l'événement **moving_end**. Celui-ci normalement se produit alors et on affiche la nouvelle position.

En faisant des essais de mouvement d'objets à la main, seul l'événement **moving_end** se déclenche. Moralité : attention à ces événements !

L'utilité de ces événements pour détecter une entrée dans un simulateur est limité étant donné qu'il est très simple de passer par un événement **changed** avec le masque **CHANGED_REGION**.

8.13 dataserver

L'événement **dataserver** se déclenche lorsque des informations sont disponibles suite à une demande. La demande peut être émise par l'une des fonctions suivantes :

Fonction	Utilisation
llGetNotecardLine	Lecture d'une ligne de notecard
llNumberOfNotecardLines	Nombre de lignes dans une notecard
llRequestAgentData	Informations sur un avatar
llRequestInventoryData	Informations sur un inventory
llRequestSimulatorData	Informations sur un simulateur

L'événement possède deux paramètres :

Paramètre	Nom	Type	Fonction
1	queryid	key	key renvoyée par l'une des fonctions ci-dessus
2	data	string	Informations demandées par la fonction

L'événement **dataserver** étant déclenché pour tous les scripts présents dans un **prim** il est nécessaire d'utiliser la clé renvoyée par la fonction. D'autre part si plusieurs demandes ont été effectuées leur ordre d'arrivée n'est pas garanti et la clé est aussi nécessaire pour les distinguer.

8.13.1 Notecard

Voici un exemple de lecture de **notecard** :

```

integer ligne;
key indexDemande;
default {
    touch_start(integer total_number) {
        ligne = 0;
    }
}

```



```

        indexDemande = llGetNotecardLine("MaNotecard", ligne++);
    }
    dataserver(key requested, string data) {
        if (requested == indexDemande && data != EOF) {
            llWhisper(0, data);
            indexDemande = llGetNotecardLine("MaNotecard", ligne++);
        }
    }
}

```

Dans cet exemple nous supposons que nous avons une **notecard** qui s'appelle **MaNotecard**. Dès que l'objet est touché l'événement **touch_start** est déclenché. La variable **ligne** est initialisée à 0 et on utilise la fonction **llGetNotecardLine** pour demander la lecture d'une ligne de la **notecard**. Cette fonction comporte deux paramètres, le premier pour le nom de la **notecard**, le second pour le numéro de la ligne à lire. Dès que l'information est disponible l'événement **dataserver** se déclenche. On teste qu'il s'agit bien de notre demande et que nous n'avons pas atteint la fin de la note (**EOF**). A ce moment là nous affichons le contenu de la ligne et demandons la lecture de la ligne suivante.

Remarquez l'utilisation d'une post-incrémentation dans ce script au niveau de la variable **ligne**. Pour plus de précision reportez-vous au point 2.1.

8.13.2 Informations sur un avatar

La fonction **llRequestAgentData** permet de récupérer des informations sur un **avatar** à partir de sa clé :

Constante	Valeur	Retour
DATA_ONLINE	1	TRUE si l'avatar est en ligne, sinon FALSE
DATA_NAME	2	Nom complet de l'avatar, équivalent à llKey2Name
DATA_BORN	3	Naissance de l'avatar sous la forme AAAA-MM-JJ
DATA_RATING	4	Plus utilisé
DATA_PAYINFO	8	Informations de paiement

Voici un exemple pour récupérer le nom d'un **avatar** :

```

default {
    touch_start(integer total_number) {
        llRequestAgentData(llDetectedKey(0), DATA_NAME);
    }
    dataserver(key requested, string data) {
        llWhisper(0, "Vous vous appelez " + data);
    }
}

```

Cet événement ne peut retourner qu'une valeur à la fois dans le paramètre data, vous ne devez donc pas combiner des demandes mais les enchaîner.

8.14 collision, collision_start, collision-end

Ces événements permettent de détecter une collision entre un objet ou un **avatar** et l'objet dans lequel se situe le script. La différence entre **collision** et **collision_start** est subtile : les deux événements se déclenchent lorsqu'un

objet ou un avatar entre en collision avec l'objet qui contient le script, mais le premier événement ne se déclenche pas en cas d'utilisation de la fonction **IIVolumeDetect**. Voici un exemple classique avec **collision** :

```
default {
    collision(integer total_number) {IIWhisper(0, IIDetectedName(0) + " me heurte !");}
}
```

Pour les objets liés l'événement dans le **root** est déclenché lors d'une collision avec n'importe quel enfant, sauf si l'enfant contient lui même un événement de collision même vide. Pour le cas où l'enfant contient un événement de collision et que l'on veut quand même que l'événement soit transmis au **root** il faut utiliser la fonction **IIPassCollisions**. Voici un exemple de script dans un enfant avec passage de l'événement au **root** :

```
default {
    state_entry() {IIPassCollisions(TRUE);}
    collision_start(integer total_number) {IIWhisper(0, IIDetectedName(0) + " me heurte !");}
    collision_end(integer total_number) {IIWhisper(0, IIDetectedName(0) + " ne me touche plus.");}
}
```

Evidemment, pour les objets fantôme les collisions ne fonctionnent plus sauf en cas d'utilisation de **IIVolumeDetect**. Cette fonction avec **TRUE** comme paramètre transforme l'objet complet en fantôme. Voici un détecteur simple qui évite de détecter le propriétaire avec utilisation de la fonction **IICollisionFilter** pour filtrer les collisions et changement du son par défaut avec la fonction **IICollisionSound** :

```
default {
    state_entry() {
        IIVolumeDetect(TRUE);
        key owner = IIGetOwner();
        IICollisionFilter("", owner, FALSE);
        IICollisionSound("MonSon", 1.0);}
    collision_start(integer total_number) {
        IIWhisper(0, IIDetectedName(0) + " arrive !");}
    collision_end(integer total_number) {
        IIWhisper(0, IIDetectedName(0) + " part.");}
}
```

Dix fonctions renseignent sur la nature et les caractéristiques de ce qui entre en collision :

Fonction	Retour
IIDetectedGrab	Direction de l'objet manipulé
IIDetectedGroup	TRUE si l'objet ou l'avatar a le même groupe actif
IIDetectedKey	Clef de l'objet ou avatar
IIDetectedLinkNumber	Numéro du prim enfant qui est collisionné
IIDetectedName	Nom de l'objet ou l'avatar
IIDetectedOwner	Clef du propriétaire de l'objet
IIDetectedPos	Position de l'objet ou l'avatar
IIDetectedRot	Position de l'objet ou l'avatar
IIDetectedType	Type (AGENT, ACTIVE, PASSIVE, SCRIPTED)

llDetectedVel	Vélocité de l'objet ou l'avatar
----------------------	---------------------------------

8.15 land_collision, land_collision_start, land_collision-end

Ces événements permettent de détecter une collision entre un objet et le sol.. Voici un exemple classique :

```
default {
    land_collision(vector position) {llWhisper(0, "Je touche le sol a la position " + (string)position);}
}
```

8.16 link_message

Cet événement se déclenche lors de l'appel de la fonction **llMessageLinked**. Cette fonction sert à transmettre un message entre prims liés, ou au sein d'un prim entre deux scripts. Cet événement comporte 4 paramètres :

Paramètre	Type	Rôle
sender_num	integer	Numéro du prim émetteur du message
num	integer	Valeur numérique
str	string	Valeur chaîne de caractère
id	key	Valeur de type key

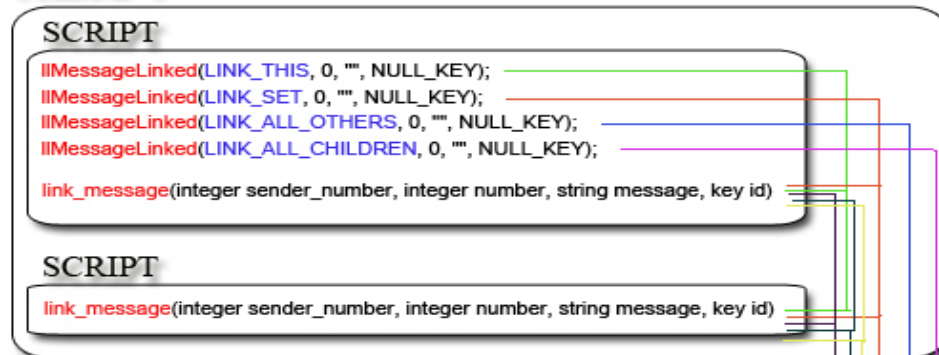
Lorsque cet événement se déclenche on peut donc récupérer le numéro du prim émetteur et un certain nombre d'informations de différents types grâce aux trois derniers paramètres. Si cela ne vous suffit pas il faut s'arranger pour grouper des informations. Par exemple une variable de type **string** n'est limitée dans sa taille que par la mémoire allouée au script. Vous pouvez donc regrouper des informations de tous types, converties en **string** et empilées dans le paramètre **str**. Prévoyez un caractère séparateur et utilisez la fonction **llParseString2List** pour récupérer les valeurs individuelles dans une **List**. Vous trouverez des exemples d'utilisation de cet événement aux points 3.8.6 et 3.4.

Le premier paramètre de la fonction **llMessageLinked** permet de définir la cible du message. Il est possible d'utiliser le numéro précis du prim lié visé, il est aussi possible d'utiliser une des constantes suivantes :

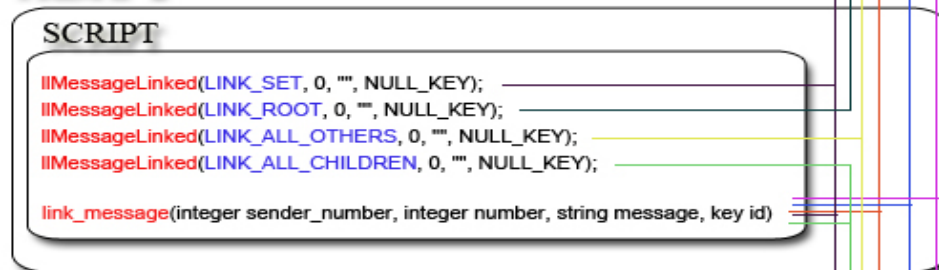
Constante	Valeur	Description
LINK_ROOT	1	Prim racine
LINK_SET	-1	Tous les prims dans un objet
LINK_ALL_OTHERS	-2	Tous les autres prims
LINK_ALL_CHILDRE	-3	Tous les prims enfants
LINK_THIS	-4	Prim dans lequel est le script

Comme il arrive souvent qu'on se pose des questions quant à ces constantes voici un petit tableau illustratif (le root est le prim 1) :

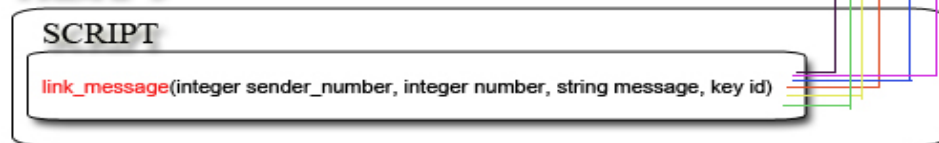
PRIM n° 1



PRIM n° 2



PRIM n° 3



9. Agir sur un objet

D'une façon générale vous pouvez accomplir par script tout ce qu'il est possible de faire en **build**. Il y a évidemment de subtiles différences si vous voulez manipuler un objet libre ou un **root**, ou un objet lié.

9.1 Les propriétés persistantes

Il y a des choses qui persistent dans un prim et d'autres qui disparaissent en même temps que le script qui les a générées. Il est bon de connaître cela pour éviter de chercher longuement un bug hypothétique ou à se débarrasser d'un effet qui semble coriace.

Voici un petit script qui efface la plupart des éléments persistants :

```
default
{
    state_entry()
    {
        llSetSitText("");
        llSetTouchText("");
        llParticleSystem([ ]);
        llSetText("", ZERO_VECTOR, 1.0);
        llTargetOmega(ZERO_VECTOR, .0, .0);
        llSetCameraAtOffset(ZERO_VECTOR);
        llSetCameraEyeOffset(ZERO_VECTOR);
        llSitTarget(ZERO_VECTOR, ZERO_ROTATION);
        llSetTextureAnim(FALSE, ALL_SIDES, 1, 1, .0, .0, .0);
        llSetColor(<1.0, 1.0, 1.0>, ALL_SIDES);
        llSetAlpha(1.0, ALL_SIDES);
        llSetTexture(TEXTURE_DEFAULT, ALL_SIDES);
        llSetStatus(STATUS_PHANTOM | STATUS_PHYSICS, FALSE);
        llSetClickAction(CLICK_ACTION_NONE);
        llStopSound();
        llOwnerSay("Nettoyage effectué... ");
        llRemoveInventory( llGetScriptName() );
    }
}
```

- ❖ **IISetSitText** et **IISetTouchText** sont destinées à purger le menu qui apparaît par le clic droit pour en revenir aux valeurs par défaut,
- ❖ **IIParticleSystem** supprime les effets de particules,
- ❖ **IISetText** efface le texte flottant au-dessus du prim,
- ❖ **IITargetOmega** arrête une rotation initialisée côté client,
- ❖ **IISetCameraAtOffset** et **IISetCameraEyeOffset** réinitialisent la caméra,
- ❖ **IISitTarget** remet à zéro la position de l'assise du prim,
- ❖ **IISetTextureAnim** arrête une animation de texture éventuelle,
- ❖ **IIStopSound** arrête un son permanent,
- ❖ **IISetAlpha** supprime la transparence
- ❖ **IISetTexture** remet la texture par défaut,
- ❖ **IISetStatus** supprime les propriétés « physique » et « fantôme »
- ❖ **IISetClickAction** remet le curseur par défaut.

Tout cela bien sûr avec les bonnes valeurs au niveau des paramètres et à adapter à vos besoins.

9.2 Translation

9.2.1 Objet libre ou root

Pour modifier la position d'un objet vous devez utiliser la fonction **IISetPos** qui attend un seul paramètre de type **vector** qui donne les coordonnées de la destination. La première dimension du vecteur exprime la position sur l'axe **X**, la seconde sur l'axe **Y** et la troisième sur l'axe **Z**. Un simulateur est un carré de 256 mètres de côté. Une position est exprimée en coordonnées régionales dans le simulateur. L'origine est située à l'extrémité sud-est. Ainsi le vecteur <0,0,0> exprime la position dans l'angle d'origine avec une altitude nulle alors que le vecteur <256,256,10> représente la position dans l'angle opposé avec une altitude de 10 mètres. Pour une description détaillée du type **vector** se reporter à la page 23.

Pour effectuer une translation d'un objet vous devez connaître sa position actuelle qui vous est donnée par la fonction **IIGetPos** et ensuite appliquer la fonction **IISetPos** :

```
default {
    touch_start(integer total_number) {
        IISetPos(IIGetPos() + <1.0,.0,.0>);}
}
```

A chaque clic l'objet subit une translation de 1 mètre sur l'axe **X**.

❶ La portée de la fonction **IISetPos** n'est que de 10 mètres. Si vous désirez un déplacement plus grand vous devez effectuer des sauts. Pour déplacer un objet d'une distance supérieure à 10 mètres utilisez cette méthode :

```
default {
    touch_start(integer _number) {
        vector destination = (IIGetPos()+ <100.0,50.0,20.0>);
        while(IIVecDist(IIGetPos(), destination) > .001)
            IISetPos(destination);}
}
```

Guide de programmation du LSL

① La fonction **llSetPos** impose un délai de 0,2 secondes ce qui ne permet pas d'effectuer des déplacements continus fluides. Une autre façon de procéder est présentée au point 8.11.

La fonction **llSetPrimitiveParams** permet aussi d'effectuer une translation :

```
default {  
    touch_start(integer _number) {  
        llSetPrimitiveParams([PRIM_POSITION, llGetPos() + <1.0,0,0>]); }  
}
```

Si on observe la syntaxe de la fonction **llSetPrimitiveParams** on se rend compte qu'elle est constituée, au niveau de ses paramètres, d'une liste de règles à appliquer. Il est possible de prévoir une liste avec la même règle répétée plusieurs fois, par exemple **PRIM_POSITION**. Ainsi, pour lever la limite des 10 mètres imposées au déplacement d'un objet non physique il est possible de prévoir une suite de petits sauts appliqués comme règle dans la liste de la fonction **llSetPrimitiveParams**. C'est cette astuce qui est utilisée dans la fonction **WarpPos** fournie par le **wiki** :

```
warpPos(vector destpos) {  
    integer jumps = (integer)(llVecDist(destpos, llGetPos()) / 10.0) + 1;  
    if (jumps > 100 ) jumps = 100;  
    list rules = [PRIM_POSITION, destpos];  
    integer count = 1;  
    while ((count = count << 1) < jumps)  
        rules = (rules=[]) + rules + rules;  
    llSetPrimitiveParams( rules + llList2List( rules, (count - jumps) << 1, count));  
    if ( llVecDist( llGetPos(), destpos ) > .001 )  
        while (--jumps) llSetPos(destpos);}
```

Un autre avantage de cette fonction est de permettre un mouvement ultra rapide puisque l'ensemble du déplacement est traité en 0,2 seconde. De fond la fonction est bridée à 100 sauts, ce qui correspond à 1 km, mais vous pouvez supprimer la ligne qui limite cette valeur, à vos risques et périls, en particulier vous pouvez rapidement atteindre la limite de mémoire de votre script !

Une utilisation fréquente des translations est la mise en action de portes simples. En voici un exemple élémentaire :

```
vector largeurPorte;  
default {  
    state_entry() {  
        vector dimensions = llGetScale();  
        largeurPorte = <dimensions.x, .0, .0>;  
        state ferme;}
```

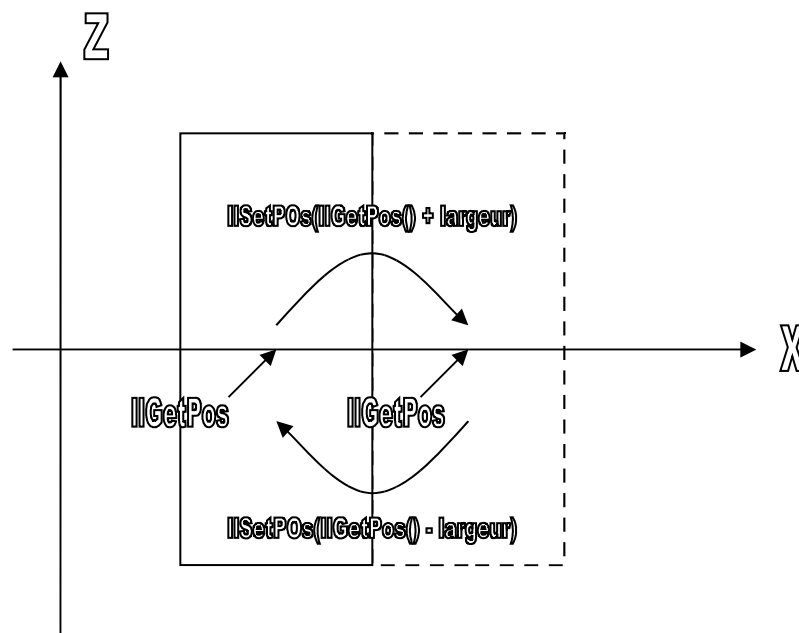
```

}
state ouvert {
    touch_start(integer total_number) {
        IISetPos(IIGetPos() - largeurPorte);
        state ferme;}
}
state ferme {
    touch_start(integer total_number) {
        IISetPos(IIGetPos() + largeurPorte);
        state ouvert;}
}

```

On admet dans ce script que la porte doit translater sur l'axe **X**. Cet exemple est également un cas typique d'utilisation judicieuse d'états pour obtenir un code clair et efficace.

Voici une petite illustration pour ce script :



9.2.2 Objet lié

Pour faire subir une translation à un objet lié il faut prendre en compte le fait que sa position est relative au **root**. Ainsi la fonction **IIGetPos** doit céder la place à son équivalent en coordonnées locales : **IIGetLocalPos**. D'autre part la fonction **IISetPos** se réfère maintenant aux coordonnées locales. Il faut aussi savoir si le script se situe dans le **root** ou dans le **prim** lié.

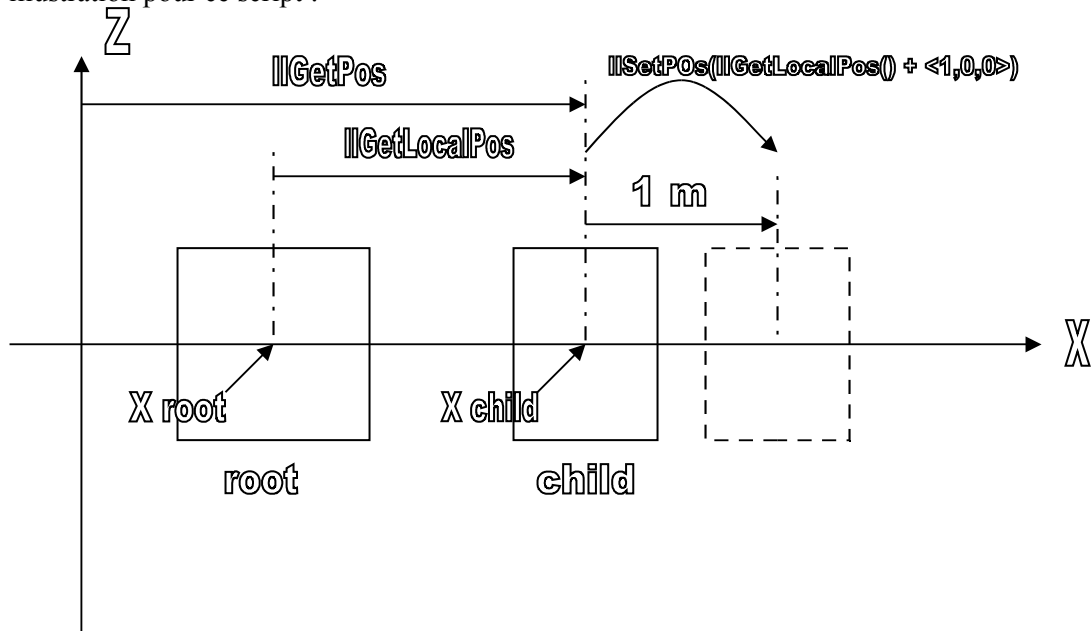
9.2.2.1 Script dans le prim lié

Pour effectuer une translation de 1 mètre sur l'axe X :

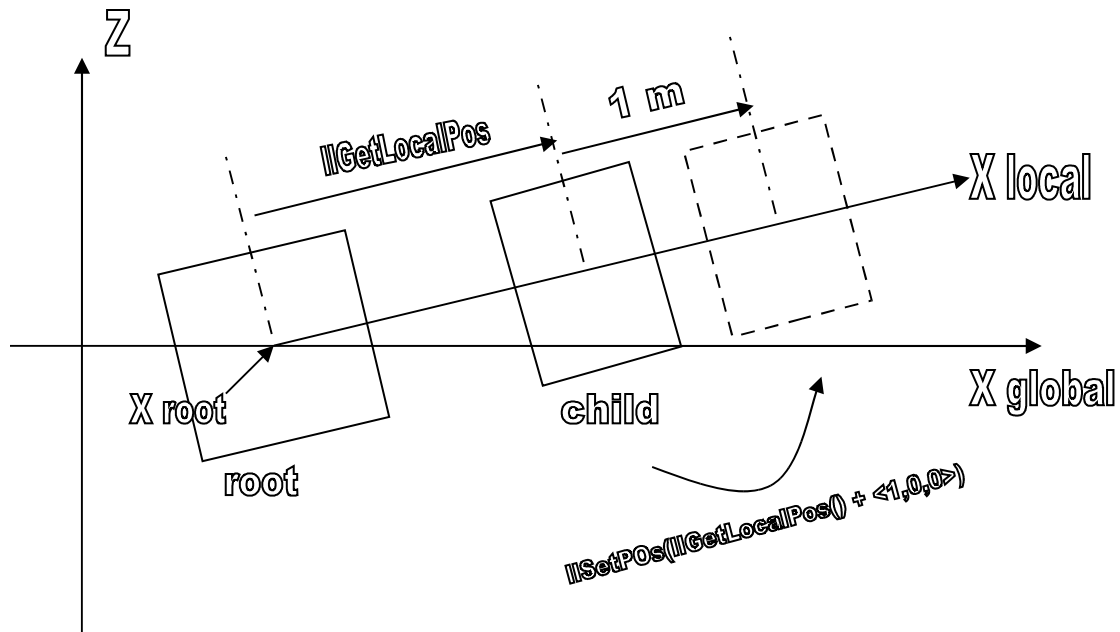
```
default {  
    touch_start(integer _number) {  
        llSetPos(llGetLocalPos() + <1.0,.0,.0>);  
    }  
}
```

De la même manière le script de la porte simple vu précédemment ne fonctionne plus si la porte est liée. Il faut modifier le code avec la fonction **llGetLocalPos** pour obtenir des valeurs correctes.

Voici une illustration pour ce script :



Dans cette illustration j'ai admis que l'axe X du root est aligné avec l'axe X global pour faciliter la compréhension. Mais évidemment le script fonctionne également en cas de non alignement :



La fonction **llSetPrimitiveParams** permet aussi d'effectuer une translation pour un objet lié, comme vu précédemment.

Il est possible de connaître la position du **root** avec la fonction **llGetRootPosition**.

9.2.2.2 Script dans le **root**

Lorsque le script est dans le **root** vous pouvez effectuer une translation d'un objet enfant à condition de connaître son index de liaison. Pour rappel le **root** a le numéro 1 et les enfants sont numérotés ensuite : 2, 3, 4... Quand vous connaissez l'index de l'enfant vous utilisez la fonction **llSetLinkPrimitiveParams** pour modifier la position. En admettant par exemple que cet index est 3 :

```
default {
    touch_start(integer _number) {
        llSetLinkPrimitiveParams(3, [PRIM_POSITION, <1.0,0,0>]); }
}
```

Mais comme le **root** ne peut pas connaître la position de ses enfants cette possibilité n'est pas vraiment utile, sauf à connaître la position d'origine.

9.3 Rotations

9.3.1 Objet libre ou root

Pour modifier la rotation d'un objet vous pouvez utiliser la fonction **llSetRot** qui attend un seul paramètre de type **rotation**. Le type **rotation** est expliqué en détail à la page 25. Pour effectuer une rotation d'un objet il faut multiplier sa rotation actuelle par la rotation désirée. Voici un exemple pour obtenir une rotation de 30 degrés sur l'axe X et 15 degrés sur l'axe Y :

```
default {  
    touch_start(integer _number) {  
        rotation rot = llEuler2Rot(<30.0 * DEG_TO_RAD, 15.0 * DEG_TO_RAD, .0>);  
        llSetRot(rot * llGetRot()); }  
}
```

Rappelez-vous que l'ordre des opérandes est important dans la combinaison des rotations ! Pour effectuer une rotation en continu vous pouvez évidemment passer par une boucle ou un **timer** :

```
float delai;  
float angle;  
rotation rot;  
default {  
    state_entry() {  
        delai = .3;  
        angle = DEG_TO_RAD * 15;  
        rot = llEuler2Rot(<.0, .0, angle>);  
        llSetTimerEvent(delai);}  
    timer() {  
        llSetRot(rot * llGetRot());}  
}
```

Dans cet exemple on utilise un **timer** réglé à 0,3 secondes. A chaque déclenchement du **timer** on effectue une rotation de 15 degrés. Avec ce type de script on obtient une rotation un peu saccadée. Vous avez une alternative avec la fonction **llTargetOmega** :

```
default {  
    touch_start(integer total_number) {  
        llTargetOmega(<.0,1.0,.0>, PI, 1.0);  
        llSay(0, "Tourne");  
        state tourne;}  
}  
state tourne {  
    touch_start(integer total_number) {  
        llTargetOmega(<.0,1.0,.0>, .0, 1.0);  
        llSay(0, "Arret");  
        state default;}  
}
```

Dans cet exemple on commande la rotation et l'arrêt de celle-ci avec un clic sur l'objet. La fonction **llTargetOmega** possède trois paramètres : le premier indique l'axe de rotation, le second la vitesse angulaire, le troisième la force. La vitesse angulaire est exprimée en radians par seconde. Ici nous avons choisi PI ce qui signifie qu'à chaque seconde l'objet aura tourné de 180 degrés.

Cette fonction crée un mouvement qui est calculé côté client. Ce qui signifie que c'est votre ordinateur en local qui gère cette rotation, c'est pour cette raison qu'elle est plus fluide qu'une rotation générée par le serveur. Par contre chaque client voit sa rotation à lui, il n'y a aucune synchronisation. D'autre part la rotation s'effectue selon les axes globaux. Dans notre exemple nous avons choisi l'axe **Y** en utilisant le vecteur $\langle 0, 1, 0 \rangle$. Nous aurions pu tout aussi bien choisir l'axe **X** ou **Z**. Il arrive parfois qu'on veuille effectuer une rotation à partir d'un axe local parce que notre objet possède déjà une rotation. En général l'axe **Z** est utilisé et il faut entrer **IIRot2Up(IIGetRot())** comme paramètre axial. Pour rappel la fonction **IIRot2Up** retourne un vecteur modulaire qui représente l'axe **Z** après une rotation. C'est donc exactement ce qu'il nous faut ici, l'axe **Z** local qui nous est donné après la rotation de base de l'objet. Pour les autres axes vous devez utiliser les fonctions **IIRot2Fwd** et **IIRot2Left**.

La fonction **IISetPrimitiveParams** permet aussi d'effectuer une rotation :

```
default {
    touch_start(integer _number) {
        rotation rot = IIEuler2Rot(<30.0 * DEG_TO_RAD, 15.0 * DEG_TO_RAD, .0>);
        IISetPrimitiveParams([PRIM_ROTATION, rot * IIGetRot()]);
    }
}
```

Les rotations sont très utilisées pour l'animation des portes. Le cas le plus classique étant un prim axial contenant le script et le prim de la porte liée. Voici un script simplifié pour assurer la commande :

```
rotation rot;
default {
    state_entry() {
        rot = IIEuler2Rot(<.0, .0, PI_BY_TWO>);
        state ferme;
    }
}
state ouvert {
    touch_start(integer total_number) {
        IISetRot(IIGetRot() / rot);
        state ferme;
    }
}
state ferme {
    touch_start(integer total_number) {
        IISetRot(rot * IIGetRot());
        state ouvert;
    }
}
```

On détermine une rotation de 90 degrés, soit $\pi/2$, grâce à la constante **Pi_BY_TWO**. On utilise deux états pour les deux positions de la porte : ouverte et fermée. Dans un cas on utilise la rotation directe, dans l'autre cas on prend l'inverse. Pour une explication détaillée des rotations se reporter au point 3.7.

Il est possible d'éviter l'utilisation d'un prim axial pour assurer la rotation. Après tout le mouvement d'une porte n'est jamais que la combinaison d'une rotation et d'une translation. La fonction **IISetPrimitiveParams** permet de faire les deux actions simultanément. Pour déterminer la translation il suffit de se rendre compte qu'il s'agit d'une valeur égale à la moitié de la largeur de la porte sur les axes **X** et **Y** :

```

rotation rot;
vector translation;
default {
    state_entry() {
        rot = llEuler2Rot(<.0, .0, PI_BY_TWO>);
        vector dimensions = llGetScale();
        float demiLargeurPorte = dimensions.x / 2;
        translation = < -demiLargeurPorte, demiLargeurPorte, .0> * llGetRot();
        state ferme;
    }
    state ouvert {
        touch_start(integer total_number) {
            llSetPrimitiveParams([PRIM_ROTATION, llGetRot() / rot,
                                PRIM_POSITION, llGetPos() + translation]);
            state ferme;
        }
    }
    state ferme {
        touch_start(integer total_number) {
            llSetPrimitiveParams([PRIM_ROTATION, rot * llGetRot(),
                                PRIM_POSITION, llGetPos() - translation]);
            state ouvert;
        }
    }
}

```

La fonction **llLookAt** permet d'orienter l'axe Z d'un objet en direction d'une cible. Elle provoque donc une rotation de l'objet qui contient le script. L'intérêt de cette fonction est qu'on n'a pas besoin de connaître la rotation actuelle de l'objet ni la valeur de la rotation à donner à l'objet. On se contente de lui dire « tu orientes ton axe Z en direction de ce point » et l'objet se tourne docilement dans la direction indiquée. Cette fonction comporte 3 paramètres :

Paramètre	Nom	Type	Fonction
1	target	vector	Vecteur de localisation de la cible
2	strength	float	Force du mouvement
3	damping	float	Vitesse du mouvement

Cette fonction concerne aussi bien les objets « physiques » que les autres. Le wiki conseille d'utiliser comme valeur du paramètre **strength** la moitié de la masse de l'objet et pour le paramètre **damping** le dixième de la valeur du paramètre **strength**. L'objet tourne autour de son centre de gravité. L'appel de la fonction **llStopLookAt** annule l'effet de cette fonction. La fonction **llLookAt** permet l'orientation uniquement par rapport à l'axe Z.

La fonction **IIRotLookAt** ressemble beaucoup à la précédente, en voici les paramètres :

Paramètre	Nom	Type	Fonction
1	target	rotation	Vecteur de localisation de la cible
2	strength	float	Force du mouvement
3	damping	float	Vitesse du mouvement

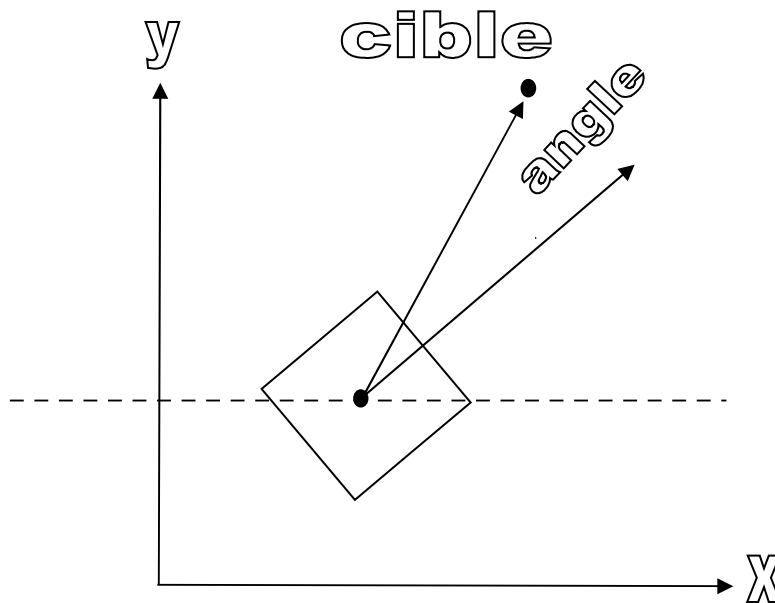
La seule différence réside dans le fait que la cible est maintenant une rotation. Il y a dans le wiki un script intéressant qui permet d'utiliser cette fonction pour orienter un objet selon un axe :

```
vector AXIS_UP = <0,0,1>;
vector AXIS_LEFT = <0,1,0>;
vector AXIS_FWD = <1,0,0>;
float strength = 1.0;
float damping = 1.0;
rotation getRotToPointAxisAt(vector axis, vector target) {
    return IIGetRot() * IIRotBetween(axis * IIGetRot(), target - IIGetPos());}
default {
    state_entry() {
        vector target = <10, 20, 30>;
        IIRotLookAt(getRotToPointAxisAt(AXIS_UP, target), strength, damping);
        IILookAt(target, strength, damping);
        IIRotLookAt(getRotToPointAxisAt(AXIS_FWD, target), strength, damping);
        IIRotLookAt(getRotToPointAxisAt(AXIS_LEFT, target), strength, damping);}
}
```

Pour utiliser la fonction **IIRotLookAt** il faut évidemment déterminer la rotation cible, ce qui n'est pas forcément évident. La fonction **getRotToPointAxisAt** présente dans ce script permet de déterminer la rotation nécessaire pour orienter l'axe passé en paramètre vers la cible également passée en paramètre. Analysons cette fonction :

- **IIGetRot()** nous donne la rotation actuelle de l'objet,
- **IIRotBetween()** est une fonction qui donne la rotation entre deux vecteurs :
 - **axis * IIGetRot()** nous donne l'axe avec la rotation actuelle de l'objet
 - **target - IIGetPos()** nous donne le vecteur de direction de la cible

Vite un dessin !



Ce dessin est une représentation en deux dimensions mais il est parfaitement transposable pour un monde en 3 dimensions. Il permet de visualiser ce qu'effectue la fonction. Un grand avantage de la fonction **IIRotLookAt** est qu'elle permet des rotations plus fluides que **IISetRot**.

9.3.2 Objet lié

Pour faire subir une rotation à un objet lié vous pouvez utiliser tout ce que nous avons vu pour un objet libre. Mais vous pouvez aussi utiliser des fonctions qui se réfèrent aux coordonnées locales : **IIGetLocalRot** et **IISetLocalRot**. Il faut aussi savoir si le script se situe dans le **root** ou dans le **prim** lié.

9.3.2.1 Script dans le prim lié

Considérons un exemple. Nous voulons créer un chronomètre avec une aiguille qu'il nous faut animer. Nous voulons que cette aiguille soit liée au chronomètre pour simplifier les déplacements de celui-ci. On prévoira également une texture pour l'aiguille de telle façon que seule la moitié soit visible, de cette façon on se retrouve avec l'axe de rotation au centre du chronomètre. Il est aussi possible de modifier le prim pour décaler le centre de rotation en bordure du prim. Le but final étant de pouvoir appliquer une simple rotation pour notre aiguille. Une fois tout cela mis en place il ne reste plus qu'à écrire le script :

```
rotation rot;
default {
    state_entry() {
        rot = IIEuler2Rot(<.0, -DEG_TO_RAD, .0>);
        state arret; }
}
state arret {
    touch_start(integer total_number) {
```

```

        state marche;}
}
state marche {
    state_entry() {
        IlSetLocalRot(ZERO_ROTATION);
        IlSetTimerEvent(1.0);}
    touch_start(integer total_number) {
        IlSetTimerEvent(.0);
        state arret; }
    timer() {IlSetLocalRot(rot * IlGetLocalRot());}
}

```

Nous définissons au départ une rotation d'un degré sur l'axe Y. Ensuite l'utilisation d'un **timer** réglé à 1 seconde permet la marche du chronomètre. Une utilisation judicieuse de deux états et d'événements **touch_start** permet le fonctionnement.

Si nous considérons le script de la porte sans axe vu dans le cas de l'objet libre, est-il possible de le rendre opérationnel pour une porte liée ? Voici ce que ça donne :

```

rotation rot;
vector translation;
default {
    state_entry() {
        rot = IlEuler2Rot(<.0, .0, PI_BY_TWO>);
        vector dimensions = IlGetScale();
        float demiLargeurPorte = dimensions.x / 2;
        translation = <-demiLargeurPorte, demiLargeurPorte, .0>;
        state ferme;}
}
state ouvert {
    touch_start(integer total_number) {
        IlSetPrimitiveParams([PRIM_ROTATION, IlGetRot() / rot / IlGetRootRotation() /
IlGetRootRotation(), PRIM_POSITION, IlGetLocalPos() + translation]);
        state ferme;}
}
state ferme {
    touch_start(integer total_number) {
        IlSetPrimitiveParams([PRIM_ROTATION, rot * IlGetRot() / IlGetRootRotation() / IlGetRootRotation(), PRIM_POSITION, IlGetLocalPos() - translation]);
        state ouvert;}
}

```



```
}
```

Il est possible de connaître la rotation du **root** grâce à la fonction **llGetRootRotation**. Pour effectuer une rotation d'un objet lié de manière globale il faut diviser la rotation globale deux fois (et oui il faut le savoir !) par la rotation du **root**.

9.3.2.2 Script dans le root

Lorsque le script est dans le **root** vous pouvez effectuer une rotation d'un objet enfant à condition de connaître son index de liaison. Pour rappel le **root** a le numéro 1 et les enfants sont numérotés ensuite : 2, 3, 4... Quand vous connaissez l'index de l'enfant vous utilisez la fonction **llSetLinkPrimitiveParams** pour modifier la rotation. En admettant par exemple que cet index est 3 :

```
default {  
    touch_start(integer _number) {  
        rotation rot = llEuler2Rot(<.0, .0, 90.0 * DEG_TO_RAD>);  
        llSetLinkPrimitiveParams(3, [PRIM_ROTATION, rot]);  
    }  
}
```

9.4 Dimensions

9.4.1 Objet libre ou root

Pour modifier les dimensions d'un prim il faut utiliser la fonction **llSetScale**. Son unique paramètre est un vecteur qui contient les dimensions sur les axes **X**, **Y** et **Z**. Pour connaître ses dimensions c'est la fonction **llGetScale** qui est concernée :

```
default {  
    touch_start(integer total_number) {  
        llSetScale(llGetScale() - <1.0, 1.0, 1.0>);  
    }  
}
```

Ce script diminue les 3 dimensions du prim de 1 mètre à chaque clic. La fonction **llSetPrimitiveParams** permet aussi d'effectuer une modification des dimensions :

```
default {  
    touch_start(integer total_number) {  
        llSetPrimitiveParams([PRIM_SIZE, llGetScale() - <1.0, 1.0, 1.0>]);  
    }  
}
```

9.4.2 Objet lié

9.4.2.1 Script dans le prim lié

Pour faire subir une modification des dimensions à un objet lié vous pouvez utiliser ce que nous avons vu pour un objet libre.

9.4.2.2 Script dans le root

Lorsque le script est dans le **root** vous pouvez effectuer une modification des dimensions d'un objet enfant à condition de connaître son index de liaison. Pour rappel le **root** a le numéro 1 et les enfants sont numérotés

ensuite : 2, 3, 4... Quand vous connaissez l'index de l'enfant vous utilisez la fonction **llSetLinkPrimitiveParams** pour modifier les dimensions. En admettant par exemple que cet index est 3 :

```
default {
    touch_start(integer total_number) {
        llSetLinkPrimitiveParams(3, [PRIM_SIZE, <2.0, .0, .0>]);}
}
```

9.5 Couleur et transparence

9.5.1 Objet libre ou root

Pour modifier la couleur d'un prim il faut utiliser la fonction **llSetColor**. Pour connaître la couleur d'un prim c'est la fonction **llGetColor** qui est concernée. La fonction **llSetColor** possède deux paramètres, le premier est un vecteur qui contient les couleurs primaires additives rouge, vert et bleu. Chacune de ces primaires reçoit une valeur de type **float** entre 0.0 et 1.0. Le second paramètre indique la ou les faces concernées, les faces étant identifiées par un numéro, la constante **ALL_SIDES** désignant toutes les faces. La fonction **llGetColor** possède un seul paramètre pour indiquer la ou les faces concernées. Voici un script qui permet de rendre la couleur de toutes les faces plus lumineuse à chaque clic en ajoutant la même valeur à chacune des trois primaires :

```
default {
    touch_start(integer total_number) {
        llSetColor(llGetColor(ALL_SIDES) + <.1, .1, .1>, ALL_SIDES);}
}
```

Pour modifier la transparence (**alpha**) d'un prim il faut utiliser la fonction **llSetAlpha**. Pour connaître la transparence d'un prim c'est la fonction **llGetAlpha** qui est concernée. La fonction **llSetAlpha** possède deux paramètres, le premier est un **float** qui représente la transparence avec une valeur de 1.0 (totalement opaque) à 0.0 (totalement transparent). Le second paramètre indique la ou les faces concernées, les faces étant identifiées par un numéro, la constante **ALL_SIDES** désignant toutes les faces. La fonction **llGetAlpha** possède un seul paramètre pour indiquer la ou les faces concernées, mais attention ! Dans ce cas vous obtenez la somme des valeurs de toutes les faces ! Voici un script qui permet de rendre la transparence de toutes les faces d'un cube plus grande à chaque clic :

```
default {
    touch_start(integer total_number) {
        llSetAlpha(llGetAlpha(ALL_SIDES) / 6 - .1, ALL_SIDES);}
}
```

La fonction **llSetPrimitiveParams** permet aussi d'effectuer une modification de couleur et de transparence. Le script suivant colore toutes les faces en rouge avec une transparence moyenne :

```
default {
    touch_start(integer total_number) {
        llSetPrimitiveParams([PRIM_COLOR, ALL_SIDES, <1.0, .0, .0>, .5]);}
}
```

9.5.2 Objet lié

9.5.2.1 Script dans le prim lié

Pour faire subir une modification de couleur et de transparence à un objet lié vous pouvez utiliser ce que nous avons vu pour un objet libre.

9.5.2.2 Script dans le root

Lorsque le script est dans le **root** vous pouvez effectuer une modification de couleur et de transparence d'un objet enfant à condition de connaître son index de liaison. Pour rappel le **root** a le numéro 1 et les enfants sont numérotés ensuite : 2, 3, 4... Quand vous connaissez l'index de l'enfant vous utilisez la fonction **llSetLinkPrimitiveParams** pour modifier la couleur et la transparence. En admettant par exemple que cet index est 3 :

```
default {
    touch_start(integer total_number) {
        llSetLinkPrimitiveParams(3, [PRIM_COLOR, ALL_SIDES, <1.0, .0, .0>, .5]);}
}
```

9.6 Statut

9.6.1 Objet libre ou root

Un certain nombre de constantes permettent de déterminer le statut d'un prim :

Constante	Statut	Défaut
STATUS_PHANTOM	Objet fantôme sans collision	FALSE
STATUS_PHYSICS	Objet « physique »	FALSE
STATUS_ROTATE_X	Autorise la rotation autour de l'axe X pour un objet « physique »	TRUE
STATUS_ROTATE_Y	Autorise la rotation autour de l'axe Y pour un objet « physique »	TRUE
STATUS_ROTATE_Z	Autorise la rotation autour de l'axe Z pour un objet « physique »	TRUE
STATUS_BLOCK_GRAB	Bloque l'outil « main » du build pour un tiers	FALSE
STATUS_SANDBOX	Empêche un objet de franchir la limite d'un simulateur et d'aller à plus de 20 mètres de son lieu de création	FALSE
STATUS_DIE_AT_EDGE	Contrôle le retour d'un objet dans l'inventaire	TRUE

La fonction **llSetStatus** permet de définir une ou plusieurs valeurs de statut, alors que la fonction **llGetStatus** permet de les lire. Voici un exemple de script qui vérifie qu'un objet est bien physique et modifie son statut au besoin :

```
default {
    touch_start(integer total_number) {
        if(!llGetStatus(STATUS_PHYSICS))
            llSetStatus(STATUS_PHYSICS, TRUE);}
}
```

La fonction permet de changer plusieurs statuts en une seule fois à condition d'affecter la même valeur.

La fonction **llSetPrimitiveParams** permet aussi d'effectuer des modifications de statut :

```
default {
    touch_start(integer total_number) {
        llSetPrimitiveParams([PRIM_PHYSICS, TRUE, PRIM_PHANTOM, FALSE]);}
}
```

L'avantage avec cette fonction est de grouper plusieurs affectations avec des valeurs différentes.

9.6.2 Objet lié

Modifier le statut d'un objet lié a peu de sens aussi ne nous attarderons-nous pas sur ce sujet.

9.7 Texture

Une texture est une image appliquée sur la face d'un objet, d'une particule ou d'un avatar. Des textures sont disponibles au sein du jeu. Vous pouvez aussi importer vos propres images moyennant le paiement de 10 l\$. Il est conseillé d'importer ces textures au format TGA. En build on applique simplement une texture sur une face d'un objet en la faisant glisser de l'inventaire sur la face concernée. Activer la touche SHIFT pendant le glisser-déposer a pour effet d'appliquer la texture sur toutes les faces de l'objet.

9.7.1 Appliquer une texture et lire une texture

L'opération la plus courante est certainement celle qui consiste à appliquer une texture sur une face d'un objet. La fonction **llSetTexture** comporte deux paramètres : le premier pour désigner la texture par son nom ou sa clé, le second pour définir la face sur laquelle la texture doit s'appliquer, ou les faces si on utilise la constante **ALL_SIDES**. Si on nomme la texture celle-ci doit se trouver dans l'inventaire de l'objet. Voici un exemple simple d'application d'une texture nommée et d'une texture identifiée par sa clé :

```
llSetTexture("MaTexture", 1);
llSetTexture("66862f3c-e090-d9c8-058d-d6575a6ed1b8", 2);
```

Si vous voulez obtenir une transparence il est conseillé d'utiliser la fonction **llSetAlpha**. Vous pouvez toutefois appliquer une texture transparente, SL en propose une en standard, sa clé est "f54a0c32-3cd1-d49a-5b4f-7b792bebc204".

Pour connaître la texture appliquée à une face utilisez la fonction **llGetTexture**. Cette fonction a pour seul paramètre le numéro de la face concernée :

```
string texture = llGetTexture(1);
```

La valeur retournée est le nom de la texture si celle-ci se trouve dans l'inventaire, sa clé dans le cas contraire.

9.7.2 Dimensionner une texture

Il est possible de dimensionner par script une texture, plus exactement de définir ses répétitions horizontale et verticale. La fonction **llScaleTexture** comporte trois paramètres : le premier de type **float** pour la répétition horizontale, le second de type **float** pour la répétition verticale, le troisième pour définir la face de l'objet concernée (ou toutes en cas d'utilisation de la constante **ALL_SIDES**). Voici un exemple de redimensionnement de texture :

```
llScaleTexture(5.32, 8.65, 1);
```

❶ Le fait d'entrer des valeurs négatives pour les répétitions provoque une inversion de la texture

La fonction **llGetTextureScale** permet de connaître les répétitions d'une texture. Vous pouvez évidemment utiliser la fonction **llSetPrimitiveParams** à la place de **llScaleTexture**, et la fonction **llGetPrimitiveParams** à la place de **llGetTextureScale**.

9.7.3 Affecter une rotation à une texture

Il est possible d'appliquer une rotation par script à une texture. La fonction **llRotateTexture** comporte deux paramètres : le premier de type **float** pour l'angle de la rotation en radians, le deuxième pour définir la face de l'objet concernée (ou toutes en cas d'utilisation de la constante **ALL_SIDES**). Voici un exemple de rotation de **PI** des textures de toutes les faces d'un objet :

```
llRotateTexture(PI, ALL_SIDES);
```

La fonction **llGetTextureRot** permet de connaître la rotation d'une texture.

9.7.4 Décaler une texture (offset)

Il est possible de décaler par script une texture (**offset**). La fonction **llOffsetTexture** comporte trois paramètres : le premier de type **float** pour le décalage horizontal, le second de type **float** pour le décalage vertical, le troisième pour définir la face de l'objet concernée (ou toutes en cas d'utilisation de la constante **ALL_SIDES**). Voici un exemple de décalage de texture :

```
llOffsetTexture(2.0, 1.2, 0);
```

La fonction **llGetTextureOffset** permet de connaître les décalages d'une texture.

9.7.5 Grouper les modifications de texture avec llSetPrimitiveParams

La fonction **llSetPrimitiveParams** permet d'appliquer en une seule commande l'ensemble des modifications de textures vues ci-dessus. Voici un exemple :

```
llSetPrimitiveParams([PRIM_TEXTURE, ALL_SIDES, "MaTexture",  
                    <1.0,1.0,0>, <2,2,0>, PI_BY_TWO]);
```

La constante **PRIM_TEXTURE** indique qu'il s'agit d'une modification de texture, le second paramètre indique la ou les faces concernées, le troisième le nom de la texture à appliquer, le quatrième est un vecteur dont les deux premières valeurs indiquent les répétitions horizontales et verticales, le cinquième est également un vecteur dont les deux premières valeurs indiquent les décalages horizontal et vertical, le dernier paramètre enfin indique la rotation en radians.

9.7.6 Animer une texture

Vous pouvez animer une texture en modifiant les répétitions (**scale**) et les décalages (**offset**). La fonction **llSetTextureAnim** permet simplement de le faire. Voici ses paramètres :

Numéro	Nom	Type	Fonction
1	mode	integer	Mode d'animation (voir le tableau suivant)
2	face	integer	Face concernée (toutes si ALL_SIDES)
3	x_frames	integer	Nombre d'images horizontales (ignoré si ROTATE ou SCALE)
4	y_frames	integer	Nombre d'images verticales (ignoré si ROTATE ou SCALE)
5	start	float	Numéro de l'image de départ (ou valeur de départ si ROTATE ou SCALE)
6	length	float	Nombre d'images à animer (0 signifie toutes les images) ou valeur de

			départ si ROTATE ou SCALE)
7	rate	float	Vitesse d'animation en images par seconde

Constantes pour le mode d'animation :

Constante	Fonction
ANIM_ON	Animation en marche
LOOP	Bouclage de l'animation
REVERSE	Animation en sens inverse
PING_PONG	Animation en avant puis en arrière...
SMOOTH	Glissement progressif dans la direction X
ROTATE	Animation en rotation
SCALE	Animation en dimensions

Vous pouvez évidemment combiner ces valeurs sauf **ROTATION** et **SCALE**. Vous ne pouvez avoir qu'une animation par objet. Un nouvel appel de la fonction **llSetTextureAnim** démarre une nouvelle animation.

9.7.6.1 Animation « standard »

Une animation « standard » est une animation qui se contente d'effectuer des translations, à l'exclusion de toute rotation ou modification de dimension. La texture est divisée en images élémentaires et l'animation consiste dans le défilement de ces images. Les paramètres (**integer**) **x_frames** et **y_frames** déterminent comment le découpage est effectué, avec une limite supérieure de 255. Les images sont numérotées à partir de 0. La première image à afficher est déterminée par le paramètre **start**, le nombre d'images à animer est déterminé par le paramètre **length**. Fixer la valeur de **length** à 0 ne signifie pas, comme vous pourriez vous y attendre, à n'afficher aucune image, mais à les faire défiler toutes. Il ne reste plus qu'à fixer la vitesse d'animation à l'aide du paramètre **rate** en nombre d'images par seconde.

Voici un exemple classique d'animation standard de texture :

```
default {
    state_entry() {llSetTextureAnim(ANIM_ON|SMOOTH, ALL_SIDES, 2, 1, .0, 1.0, 0.2);}
}
```

Si nous analysons les paramètres transmis dans ce cas nous avons :

Nom	Type	Valeur	Fonction
mode	integer	ANIM_ON SMOOTH	L'animation est active (ANIM_ON) et progressive (SMOOTH).
face	integer	ALL_SIDES	Elle concerne toutes les faces (ALL_SIDES).
x_frame	integer	2	Deux images sur l'axe X
y_frame	integer	1	Une seule image sur l'axe Y
start	float	0	Départ de la première image
length	float	1	Une seule image à animer
rate	float	0.2	Un cinquième d'image par seconde

Nous avons donc une texture qui défile de façon fluide sur toutes les faces de l'objet concerné. Ce type d'animation est très utilisé sur **SL**. Le fait d'utiliser la valeur **SMOOTH** au niveau du mode impose quelques contraintes : le mouvement s'effectue sur l'axe X, si vous désirez avoir une succession correcte de vos images **y_frame** doit être fixé à 1 et le paramètre **length** doit avoir une valeur inférieure à 1 du paramètre **x_frame** et

start doit être fixé à 0. Si vous désirez avoir votre animation sur un axe différent de X il suffit de régler le paramètre de rotation de texture dans la fenêtre d'édition de votre objet.

9.7.6.2 Animation en rotation

Dans ce type d'animation la texture subit une rotation. La signification des paramètres change un peu :

Numéro	Nom	Type	Fonction
1	mode	integer	Mode d'animation
2	face	integer	Face concernée (toutes si ALL_SIDES)
3	x_frames	integer	Non utilisé
4	y_frames	integer	Non utilisé
5	start	float	Rotation de départ en radians
6	length	float	Rotation totale
7	rate	float	Vitesse d'animation en images par seconde sans utilisation de SMOOTH et en radians par secondes en cas d'utilisation de SMOOTH

9.7.6.3 Animation en dimension

Dans ce type d'animation la texture subit une modification de dimension, donc un effet de zoom. La signification des paramètres change un peu :

Numéro	Nom	Type	Fonction
1	mode	integer	Mode d'animation
2	face	integer	Face concernée (toutes si ALL_SIDES)
3	x_frames	integer	Non utilisé
4	y_frames	integer	Non utilisé
5	start	float	Nombre de répétition par face au début de l'animation
6	length	float	Augmentation de la répétition à la fin de l'animation
7	rate	float	Vitesse d'animation en nombre de répétitions par seconde

Pour stopper une animation en cours il suffit de fixer le mode à **FALSE** ou de désactiver la valeur **ANIM_ON**.

9.7.7 Agir sur la texture d'un objet lié

On peut appliquer une texture sur un objet lié à partir du **root** grâce à la fonction **llSetLinkTexture** à condition de connaître son index de liaison. Pour rappel le **root** a le numéro 1 et les enfants sont numérotés ensuite : 2, 3, 4...

```
llSetLinkTexture(3, "MaTexture", ALL_SIDES);
```

Vous pouvez également utiliser la fonction **llSetLinkPrimitiveParams** :

```
llSetLinkPrimitiveParams(3, [PRIM_TEXTURE, ALL_SIDES, "MaTexture",  
                             <1.0,1.0,0>, <2.2,0>, PI_BY_TWO]);
```

9.8 Inventaire (Inventory)

Il existe deux sortes d'inventaires (**inventory**) : celui de l'**avatar** où sont stockés de manière plus ou moins organisée (chez moi plutôt désorganisée...) des objets, textures, scripts, notecards... Il y a aussi l'inventaire des objets dans lesquels peuvent aussi être déposés les mêmes types d'éléments. On ne peut pas agir par script sur les inventaires des **avatars**, seulement sur ceux des objets.

9.8.1 Donner un objet d'un inventaire

Pour donner à un **avatar** un objet de l'inventaire d'un objet on peut utiliser la fonction **llGiveInventory**. Le premier paramètre est la clé de l'**avatar** et le second le nom de l'élément de l'inventaire. Voici un script tout simple qui offre un objet à l'avatar qui clique :

```
default {  
    touch_start(integer total_number) {  
        llGiveInventory(llDetectedKey(0), "MonObjet");  
    }  
}
```

Si on veut que l'objet se place dans un répertoire spécifique de l'inventaire de l'**avatar** il faut utiliser la fonction **llGiveInventoryList**. Le script suivant offre les objets "MonObjet1" et "MonObjet2" et les place dans le répertoire "MonRepertoire" en cas d'acceptation :

```
default {  
    touch_start(integer total_number) {  
        llGiveInventoryList(llDetectedKey(0), "MonRepertoire", ["MonObjet1", "MonObjet2"]);  
    }  
}
```

9.8.2 Se renseigner sur le contenu d'un inventaire

Un certain nombre de fonctions permettent d'obtenir des informations sur le contenu d'un inventaire. Voici un script qui donne des informations sur le premier objet de l'inventaire :

```
string Donne_Type(integer i) {  
    if (i == -1) return "NONE";  
    else if (i == 0) return "TEXTURE";  
    else if (i == 1) return "SOUND";  
    else if (i == 3) return "LANDMARK";  
    else if (i == 5) return "CLOTHING";  
    else if (i == 6) return "OBJECT";  
    else if (i == 7) return "NOTECARD";  
    else if (i == 10) return "SCRIPT";  
    else if (i == 13) return "BODYPART";  
    else if (i == 20) return "ANIMATION";  
    else if (i == 21) return "GESTURE";  
    else return "ERREUR";  
}  
  
string Ajoute_Perm(string s, string p) {  
    if(s == "") return p;
```



```

        else return "/" + p;
    }
    string Donne_Perm(integer i) {
        string s;
        if(i & PERM_ALL) return "MOVE/MODIFY/COPY/TRANFERT";
        else if(i & PERM_COPY) return Ajoute_Perm(s, "COPY");
        else if(i & PERM_MODIFY) return Ajoute_Perm(s, "MODIFY");
        else if(i & PERM_MOVE) return Ajoute_Perm(s, "MOVE");
        else if(i & PERM_TRANSFER) return Ajoute_Perm(s, "TRANSFERT");
        else return "ERREUR";
    }
    default {
        touch_start(integer total_number) {
            string nom = llGetInventoryName(INVENTORY_OBJECT, 0);
            llWhisper(0, "Clef du createur : " + (string)llGetInventoryCreator(nom));
            llWhisper(0, "Clef de l'objet : " + (string)llGetInventoryKey(nom));
            llWhisper(0, "Type de l'objet : " + Donne_Type(llGetInventoryType(nom)));
            llWhisper(0, "Permissions: " + Donne_Perm(llGetInventoryPermMask(nom, MASK_BASE)));
        }
    }
}

```

En ce qui concerne les clés du créateur et de l'objet pas de problème particulier. Pour le type de l'élément (**llGetInventoryType**) la valeur retournée correspond à une des constantes suivantes :

Constante	Valeur
INVENTORY_NONE	-1
INVENTORY_ALL	0
INVENTORY_TEXTURE	1
INVENTORY_SOUND	3
INVENTORY_LANDMARK	5
INVENTORY_CLOTHING	6
INVENTORY_OBJECT	7
INVENTORY_NOTECARD	10
INVENTORY_SCRIPT	13
INVENTORY_BODYPART	20
INVENTORY_ANIMATION	21
INVENTORY_GESTURE	

Le deuxième paramètre de la fonction **llGetInventoryPermMask** est un masque qui permet de déterminer les permissions à renvoyer :

Masque	Valeur	Description
MASK_BASE	0	Permissions de base de l'objet
MASK_OWNER	1	Permissions du propriétaire
MASK_GROUP	2	Permissions du groupe actif
MASK_EVERYONE	3	Permissions de tout le monde
MASK_NEXT	4	Permission du prochain propriétaire

Valeurs de retour :

Permission	Valeur	Description
PERM_ALL	2147483647	Bouger, modifier, copier, transférer
PERM_COPY	32768	Copier
PERM_MODIFY	16384	Modifier
PERM_MOVE	524288	Bouger
PERM_TRANSFER	8192	Transférer

9.8.3 Connaître le nombre d'éléments d'un type dans l'inventaire

Pour connaître le nombre d'éléments d'un type dans l'inventaire il faut utiliser la fonction **llGetInventoryNumber**. Les constantes sont les mêmes que pour la fonction **llGetInventoryType** vue précédemment. Par exemple pour connaître le nombre d'habits (**CLOTHING**) dans un inventaire :

```
llWhisper(0, "Il y a " + (string)llGetInventoryNumber(INVENTORY_CLOTHING) + " habits");
```

9.9 Lumières

Le monde de **SL** est éclairé par le soleil, la lune et un nombre maximal de 6 lampes. Pour évaluer ce nombre maximum il faut tenir compte du réglage de la caméra. Les lampes au-delà de ce nombre resteront irrémédiablement sombres. La lumière du soleil évolue au fil de la journée de façon réaliste.

9.9.1 Position du soleil et de la lune

On peut connaître la position du soleil avec la fonction **llGetSunDirection** qui renvoie un vecteur normé qui pointe en direction du soleil. En ce qui concerne la lune, elle est toujours en face du soleil, donc toujours pleine, pas étonnant qu'on trouve des tas d'animaux étranges sur **SL** ! Pour connaître la position de la lune il suffit donc de prendre l'inverse du vecteur retourné par la fonction **llGetSunDirection**. En principe la direction de ces astres est la même pour tout le monde mais un propriétaire de land peut régler une direction différente. On peut se demander à quoi peut bien servir de connaître la position du soleil. La première application est de déterminer s'il fait jour ou nuit. Lorsque vous récupérez le vecteur issu de la mise en oeuvre de la fonction **llGetSunDirection** il suffit de tester la valeur de la composante **z** pour savoir si le soleil est au-dessus (**z** positif) ou en-dessous (**z** négatif) de l'horizon. Il peut être intéressant d'ajuster la lumière d'un bâtiment en fonction de la position du soleil, donc de l'éclairage ambiant. Vous pouvez aussi déclencher un éclairage lorsque la nuit tombe et l'éteindre automatiquement à la levée du jour :

```
default {
    state_entry() {llSetTimerEvent(120.0);}

    timer(){
```

```
vector directionSoleil = llGetSunDirection();
if (directionSoleil.z < .2)
    llSetPrimitiveParams([PRIM_POINT_LIGHT, TRUE, <1.0,1.0,1.0>, 1.0, 10.0, .75]);
else if (directionSoleil.z < .0)
    llSetPrimitiveParams([PRIM_POINT_LIGHT, TRUE, <1.0,1.0,1.0>, .5, 10.0, .75]);
else
    llSetPrimitiveParams([PRIM_POINT_LIGHT, FALSE, <.0,.0,.0>, .0, .0, .0]);
}
```

Ce script prévoit une étape intermédiaire avec un éclairage moyen.

9.9.2 Lampes

Il est possible de créer des prims émetteurs de lumière directement en build. Un script devient intéressant pour modifier les paramètres de la lumière comme on en a vu un exemple ci-dessus. Il faut utiliser la fonction **llSetPrimitiveParams**. Voici les paramètres utiles :

Paramètre	Description
PRIM_FULLBRIGHT	Fait apparaître le prim lumineux
PRIM_POINT_LIGHT	Détermine les propriétés de la lumière

9.9.2.1 Full bright

Mettre une face ou un prim entier en full bright le fait apparaître lumineux mais il n'émet aucune lumière. Voici un exemple :

```
llSetPrimitiveParams([PRIM_FULLBRIGHT, ALL_SIDES, TRUE]);
```

Vous avez deux paramètres à déterminer : la face concernée ou toutes les faces avec la constante **ALL_SIDES**, et une valeur booléenne pour activer ou désactiver le full bright.

9.9.2.2 Lumière

Vous pouvez modifier les caractéristiques d'une lumière :

```
llSetPrimitiveParams([PRIM_POINT_LIGHT, TRUE, <1.0,1.0,.0>, 1.0, 10.0, .75]);
```

Le premier paramètre est booléen et détermine si la lumière est active ou pas. Le second concerne la couleur de la lumière (jaune dans cet exemple). Le troisième est l'intensité de la lumière (valeur **float** entre .0 et 1.0). Le quatrième est le rayon du cône de lumière en radians. Le dernier enfin est le falloff (valeur **float** entre .0 et 1.0).

9.10 Liaisons

Comme vous le savez on peut lier des prims pour créer des objets. Il y a certaines contraintes au niveau du build pour ces liaisons, en particulier la distance entre les prims à relier. Au niveau des script on peut faire un certain nombre de choses concernant les prims liés.

9.10.1 Numérotation des prims liés et nombre de prims

Lorsque vous liez des prims il en est un spécial désigné **root**, c'est celui que vous sélectionnez en dernier lors de la liaison, c'est aussi celui qui apparaît en jaune lors d'une sélection. Pour différencier les prims d'un bloc lié ceux-ci sont numérotés, le **root** recevant le numéro 1 et les autres prims étant numérotés successivement 2, 3, 4... Il est à noter qu'un prim libre possède aussi un numéro qui est par défaut 0, ceci peut être important à connaître pour la création de vos scripts.

Vous pouvez connaître le numéro d'un prim lié grâce à la fonction **llGetLinkNumber**. Mais cette fonction doit se trouver dans le prim concernée. Autrement dit le **root** ne peut pas connaître directement ce numéro, ce qui est parfois gênant. Par contre il peut connaître le nombre de prims liés avec la fonction **llGetNumberOfPrims**, ce que ne peut pas connaître directement un prim lié. Vous pouvez déduire rapidement de ces prémisses que les prims d'un bloc lié ont intérêt à se communiquer des informations.

9.10.2 Communication entre prims liées

Pour communiquer entre eux les prims liés utilisent la fonction **llMessageLinked** et l'événement **link_message** (cet événement est décrit au point 8.16). La fonction **llMessageLinked** possède les mêmes paramètres que l'événement **link_message**, ce qui paraît naturel :

Paramètre	Type	Description
sender_num	integer	Numéro du prim récepteur du message
num	integer	Valeur numérique
str	string	Valeur chaîne de caractère
id	key	Valeur de type key

Il faut toutefois apporter une précision au niveau du paramètre **sender_num**. Vous pouvez transmettre le numéro spécifique du prim lié qui doit recevoir le message ou une constante :

Constante	Valeur	Destinataires du message
LINK_ROOT	1	Root (on a vu qu'il est numéroté 1)
LINK_SET	-1	Tous les prims
LINK_ALL_OTHER	-2	Tous les prims sauf celui qui contient le script
LINK_ALL_CHILDREN	-3	Tous les prims enfants
LINK_THIS	-4	Prim dans lequel se situe le script

Ces constantes permettent donc toutes les combinaisons possibles.

9.10.3 Liaisons et rupture de liaisons

9.10.3.1 Création de liaison

La fonction **llCreateLink** permet de créer une liaison. Cette fonction comporte deux paramètres :

Paramètre	Type	Description
target	Key	Clef du prim auquel on se lie
parent	integer	TRUE : le prim qui se lie devient le root et le prim auquel on se lie devient le premier enfant

Cette fonction nécessite la permission **PERMISSION_CHANGE_LINK**. Nous avons vu comment fonctionnent les permissions au point 8.7. Pour que la fonction réalise le lien les deux prims doivent avoir le

même propriétaire. Si vous voulez savoir si la liaison s'effectue vous pouvez intercepter l'événement **changed** que nous avons vu en détail au point 8.6 en interceptant la valeur **CHANGED_LINK**.

9.10.3.2 Rupture de liaison

Vous pouvez supprimer toutes les liaisons d'un objet avec la fonction **llBreakAllLinks**. Ou alors la liaison d'un prim spécifique avec la fonction **llBreakLink** en transmettant le numéro du prim à délier. Dans les deux cas vous avez besoin de la permission **PERMISSION_CHANGE_LINK**.

Imaginez que vous désirez un outil pratique pour délier d'un clic un prim lié à un objet. Vous pouvez évidemment passer par l'interface classique de build de SL, mais vous désirez quelque chose de plus efficace. LSL est là pour vous apporter une solution. Le plus simple est de mettre un script dans le **root** qui attend un clic sur un prim lié. Le root peut connaître le numéro du prim cliqué grâce à la fonction **llDetectedLinkNumber**. Voici le script correspondant à mettre dans le **root** :

```
integer numLink;
default {
    touch_start(integer total_number) {
        numLink = llDetectedLinkNumber(0);
        if(numLink > 1) llRequestPermissions(llDetectedKey(0), PERMISSION_CHANGE_LINKS);
    }
    run_time_permissions(integer permissions) {
        if(permissions & PERMISSION_CHANGE_LINKS)
            llBreakLink(numLink);
    }
}
```

Lorsqu'un prim enfant est lié l'événement **touch_start** est transmis au **root** (s'il n'est pas intercepté par l'enfant : vous trouvez les règles de transmission de l'événement **touch_start** entre prims liés au point 8.2). On récupère le numéro de liaison de l'enfant, on teste que le clic ne vient pas du **root**, on demande la permission de délier et on effectue cette rupture à la réception de la permission. Il faut évidemment ne pas oublier de supprimer ce script à l'issue de votre construction pour éviter les surprises.

9.10.4 Actions entre prims liés

Un prim lié peut accomplir un certain nombre d'actions sur les autre prims liés. Voici les fonctions permettant ces actions :

Fonction	Description
llSetLinkColor	Changement de la couleur
llSetLinkAlpha	Changement de la transparence
llSetLinkTexture	Changement de la texture
llSetLinkPrimitiveParams	Changement divers simultanés

Per exemple vous désirez une fonction qui vous permette de modifier la couleur de l'ensemble des prims liés :

```
couleur (vector coul, integer face){llSetLinkColor(LINK_SET, coul, face);}
```

La fonction **llSetLinkPrimitiveParams** permet tous les changements prévus par la fonction **llSetPrimitiveParams** mais appliqués à un prim lié. On se rend ainsi compte que les prims liés possèdent une action complète sur les autres prims. Par contre il est assez démuné pour obtenir des informations les concernant. Il existe seulement deux fonctions permettant d'obtenir des renseignements :







Fonction	Description
llGetLinkKey	Clef du prim enfant
llGetLinkName	Nom du prim enfant

Si vous avez besoin de plus d'informations il faut passer par l'intermédiaire de messages liés comme vu au point 9.10.2.

9.10.5 Autres propriétés

9.10.5.1 ClickAction

Vous avez remarqué qu'un clic sur un objet est traité de différentes manières selon comment a été paramétré l'objet : assise, achat, paiement, ouverture, lancement d'un media... Il est possible de changer cette propriété par script avec la fonction **llSetClickAction**. Cette fonction possède un seul paramètre de type integer dont les constantes suivantes facilitent le codage :

Constante	Valeur	Description	Curseur
CLICK_ACTION_NONE CLICK_ACTION_TOUCH	0	Valeur par défaut, déclenche les événements touch	
CLICK_ACTION_SIT	1	Assoit l'avatar	
CLICK_ACTION_BUY	2	Ouvre le menu d'achat	
CLICK_ACTION_PAY	3	Ouvre le menu de paiement	
CLICK_ACTION_OPEN	4	Ouvre le menu d'inventaire	
CLICK_ACTION_PLAY	5	Démarre ou arrête le media du terrain	
CLICK_ACTION_OPEN_MEDIA	6	Démarre le media du terrain	

9.10.5.2 Type de primitive

La fonction **llSetPrimitiveParams** permet de changer le type d'une primitive. Des constantes facilitent le codage :

Constante	Valeur	Description
PRIM_TYPE_BOX	0	Boîte
PRIM_TYPE_CYLINDER	1	Cylindre
PRIM_TYPE_PRISM	2	Prisme
PRIM_TYPE_SPHERE	3	Sphère
PRIM_TYPE_TORUS	4	Tore
PRIM_TYPE_TUBE	5	Tube
PRIM_TYPE_RING	6	Anneau

PRIM_TYPE_SCULPT	7	Prim sculptée
------------------	---	---------------

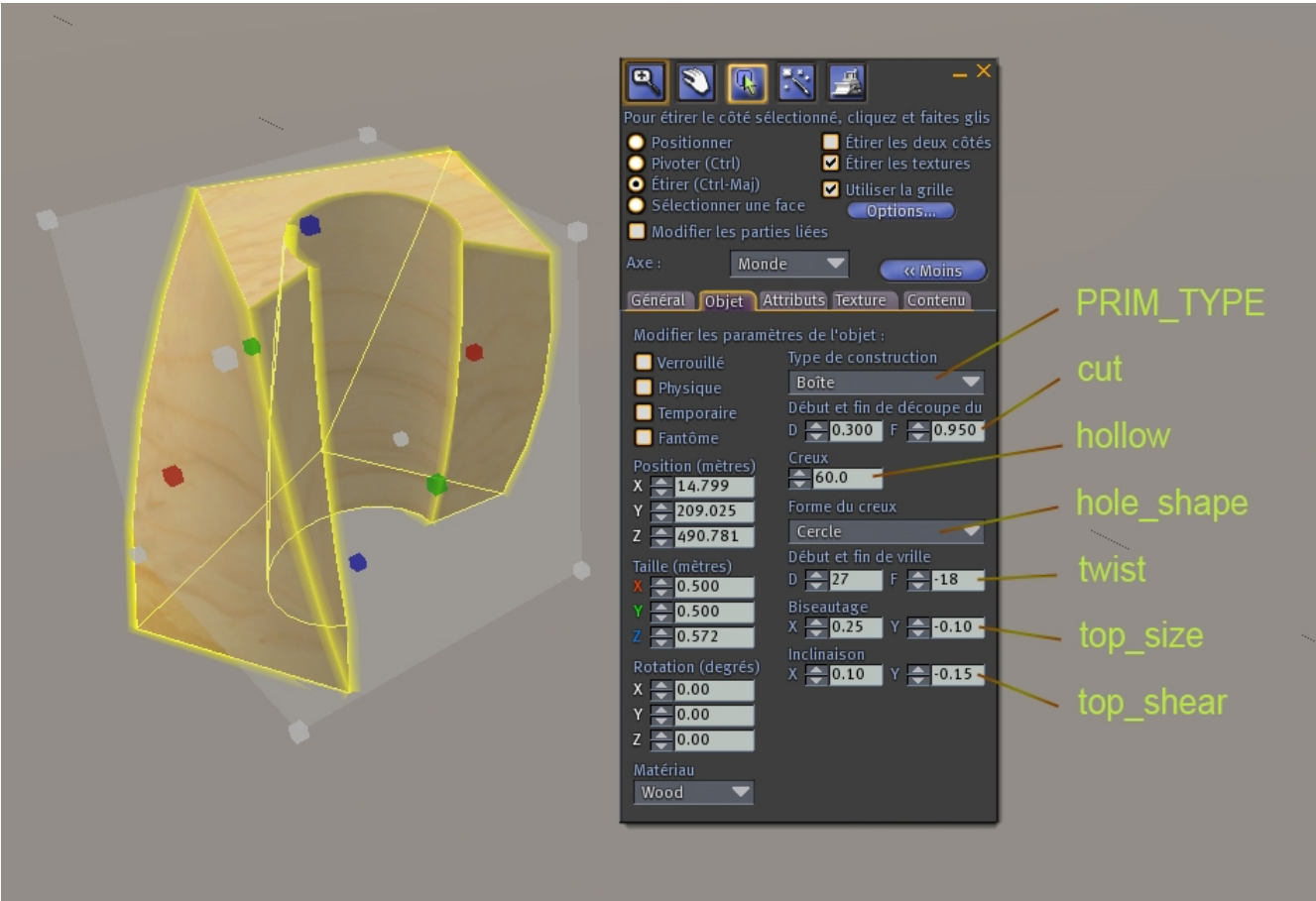
9.10.5.3 Paramètres de primitive

La fonction `llSetPrimitiveParams` permet de régler tous les paramètres des différentes formes :

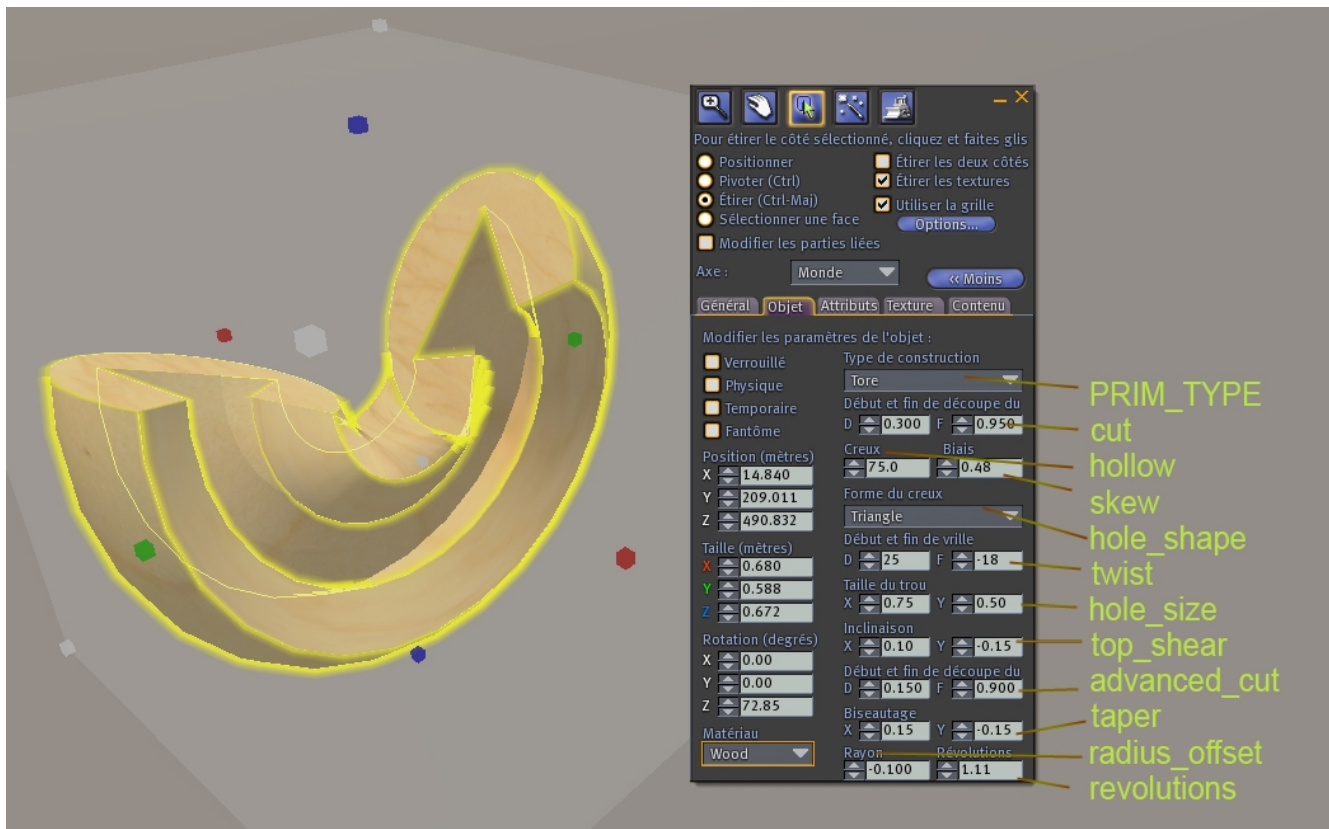
Type	hole_shape	cut	hollow	twist	dimp	top_size	top_shear	advanced_cut	taper	revolutions	radius_offset	skew
Boîte	*	*	*	*		*	*					
Cylindre	*	*	*	*		*	*					
Prisme	*	*	*	*		*	*					
Sphère	*	*	*	*	*							
Tore	*	*	*	*			*	*	*	*	*	*
Tube	*	*	*	*			*	*	*	*	*	*
Anneau	*	*	*	*			*	*	*	*	*	*

Le type **SCULPT** est un peu particulier avec ses deux paramètres **map** et **type**. Si l'intérêt de changer la forme d'une primitive par script est relativement limité (générateur de prim adaptées par exemple), l'action sur les paramètres est souvent intéressant, par exemple l'apparition ou la disparition d'une ouverture (une porte ou autre), ou alors la création d'une forme évolutive...

Voici un exemple de correspondance pour une boîte :



Vous constatez que tous les paramètres accessibles au niveau du build sont aussi modifiables par cette fonction. Voici un cas plus complexe de tore :



Voici un petit script de modification aléatoire de prim à partir des 3 formes de base : boîte, cylindre et prisme :

```
float randBetween(float min, float max) {
    return llFrاند(max - min) + min;}
default
{
    state_entry() {
        llSetTimerEvent(2.0);}

    timer() {
        integer prim_type = (integer)llFrاند(3.0);
        integer holeshape = (integer)llFrاند(4.0) * 16;
        vector cut;
        cut.y = randBetween(.1, 1.0);
        cut.y = randBetween(.0, cut.y - .05);
        float hollow = randBetween(.0, .95);
        vector twist;
```



```
twist.x = randBetween(-.5, .5);
twist.y = randBetween(-.5, .5);
vector top_size;
top_size.x = randBetween(.0, 2.0);
top_size.y = randBetween(.0, 2.0);
vector top_shear;
top_shear.x = randBetween(-.5, .5);
top_shear.y = randBetween(-.5, .5);
llSetPrimitiveParams([PRIM_TYPE, prim_type, holeshape, cut, hollow, twist, top_size,
top_shear]);
}
```

Vous pouvez constater ainsi la puissance de cette fonction et les possibilités qu'elle permet.

10. Les medias

SL est un univers visuel et sonore. Il est possible de gérer des sons, de la musique et même de la video. Voyons cela en détails.

10.1.1 Les sons

Les sons sur SL sont des fichiers au format WAV d'une longueur maximale de 10 secondes.

10.1.1.1 Précharger un son

Pour pouvoir être joué sur le poste client le fichier de son doit être téléchargé intégralement. Ce qui signifie que l'on a jamais un démarrage instantané d'un son. Il est toutefois possible de précharger un son avec la fonction **llPreLoadSound**. Cette fonction possède un seul paramètre qui peut être le nom du son (dans ce cas le son doit se trouver dans l'inventaire du prim qui contient le script) ou sa clé. Lorsqu'on utilise cette fonction tous les avatars qui se situent dans la zone d'audition du son chargent celui-ci, ce qui veut dire que le fichier correspondant est envoyé sur le poste client.

10.1.1.2 Jouer un son une fois

Pour jouer un son au niveau d'un script la première fonction à connaître est **llPlaySound**. Cette fonction comporte deux paramètres, le premier de type **string** est le nom (dans ce cas le son doit se trouver dans l'inventaire du prim qui contient le script) ou la clé du son, le deuxième de type **float** est le volume sonore (entre 0 et 1). Cette fonction joue le son une seule fois et suit les mouvement du prim qui contient le script. On ne peut jouer qu'un seul son à la fois dans un script. Voici un exemple élémentaire d'appel de cette fonction :

```
default {touch_start(integer total_number) {llPlaySound("MonSon", .5);}}
```

En cas de clic sur l'objet le son "MonSon" est joué avec un volume moyen. Le son s'arrête à la fin de lecture du fichier. Si pour une raison quelconque on veut interrompre le son avant cette fin on doit utiliser la fonction sans paramètre **llStopSound**. Voici un script mettant en oeuvre cette possibilité :

```
default {  
    on_rez(integer start_param){llPlaySound("MonSon", .5);}  
    touch_start(integer total_number){llStopSound();}  
}
```

Lors du rez de l'objet un son se déclenche mais il est possible de l'arrêter en cliquant sur l'objet.

Il est aussi possible de régler le volume sonore avec la fonction **llAdjustSoundVolume** qui a pour seul paramètre un **float** indiquant le volume sonore mais son utilité dans le cas d'un seul son est limitée.

Avec la fonction **llPlaySound** le son suit l'objet émetteur. Pour que le son reste à son point de génération initiale indépendamment des mouvements ultérieurs du prim qui contient le script il faut utiliser à la place la fonction **llTriggerSound** qui comporte les mêmes paramètres que **llPlaySound** et fonctionne de la même manière. Tout dépend donc de l'effet que vous désirez obtenir. Si votre objet reste fixe peu importe quelle fonction vous utilisez. Par contre si votre objet est mobile utilisez la fonction qui correspond à votre attente.

Mais ce n'est pas la seule différence entre ces deux fonctions. Comme je le signale ci-dessus avec **llPlaySound** vous ne pouvez générer qu'un seul son qui suit l'objet. Par contre avec **llTriggerSound** vous n'êtes pas limité en nombre de sons générés, ce qui est très pratique dans le cas de sons de collision par exemple. Par contre vous n'avez plus aucune action ultérieure possible au niveau du volume sonore sur les sons générés.

10.1.1.3 Limiter un son spatialement

llPlaySound et **llTriggerSound** génèrent le son sans limite spatiale si ce n'est le volume sonore que vous imposez au niveau de leur deuxième paramètre. Mais il est souvent judicieux de limiter la portée d'un son de façon précise, ne serait-ce que pour éviter de gêner les voisins. La fonction **llTriggerSoundLimited** permet justement de faire ça. Ses deux premiers paramètres sont identiques à ceux des deux fonctions vues précédemment, mais elle comporte deux autres paramètres pour définir l'espace dans lequel le son doit être généré. Cet espace a la forme d'une boîte et les deux paramètres en question en définissent les deux coins extrêmes.

Pour définir ces deux points vous avez deux méthodes : la première est manuelle :

1. créez une boîte de la dimension désirée pour votre espace sonore,
2. créez un petit objet,
3. positionnez ce petit objet sur un coin de la boîte et notez les coordonnées,
4. faites la même chose pour le coin opposé,
5. le vecteur comportant les valeurs les plus élevées constitue le troisième paramètre, l'autre constituant le quatrième.

La deuxième est assistée. Créez votre boîte, mettez le script suivant dedans et cliquez sur la boîte:

```
default {  
    touch_start(integer total_number){  
        vector pos = llGetPos();  
        vector dim = llGetScale() / 2.0;  
        vector n1 = pos + dim;  
        vector n2 = pos - dim;  
        llOwnerSay("Point 1 : " + (string)n1 + ", point 2 : " + (string)n2);}  
}
```

Vous obtenez les deux valeurs dans le Chat que vous pouvez copier. Mais dans les deux cas vous obtenez des coordonnées globales, ce qui n'est pas forcément très pratique. Il est plus judicieux de charger le script de définir le volume autour de l'objet, donc en valeurs relatives. Voici un exemple d'utilisation :

```
float distance = 20.0;  
default {  
    touch_start(integer total_number) {  
        vector pos = llGetPos();  
        vector P1 = pos + <distance,distance,distance>;  
        vector P2 = pos - <distance,distance,distance>;
```

```
IITriggerSoundLimited("MonSon", .8, P1, P2);}
```

Lors d'un clic sur l'objet on commence par déterminer les deux points de référence du volume enveloppe du son (20 mètres dans toutes les directions dans notre cas) et on utilise ensuite la fonction **IITriggerSoundLimited** pour jouer le son passé dans le premier paramètre. Le son sera donc entendu par tous les avatars qui se trouvent dans cet espace au début de l'émission du son mais pas par ceux qui pénètrent dans cet espace ensuite.

10.1.1.4 Jouer un son ou des sons en boucle

Les fonctions vues précédemment ne permettent de jouer un son qu'une seule fois. Il est aussi possible de faire plusieurs appels de ces fonctions pour jouer un son plusieurs fois. Mais il n'y a aucun moyen de savoir à quel moment un son se termine. Heureusement la fonction **IILoopSound** permet de jouer le même son en boucle indéfiniment. Elle possède les mêmes paramètres que **IIPlaySound**. De la même manière le son peut être stoppé avec la fonction **IIStopSound** et le volume réglé avec la fonction **IIAdjustSoundVolume**. Le son généré par **IILoopSound** est aussi attaché à l'objet qui l'a généré.

Si vous désirez jouer plusieurs sons en boucle en les synchronisant **LSL** vous offre deux fonctions bien pratiques : **IILoopSoundMaster** est équivalente à la fonction **IILoopSound** à la différence que le son correspondant sert d'élément de synchronisation. Si vous utilisez ensuite la fonction **IILoopSoundSlave** le son déclenché par celle-ci va démarrer en se callant sur le son joué par la fonction **IILoopSoundMaster**. Vous pouvez donc vous lancer dans une orchestration complète !

10.1.1.5 Jouer plusieurs sons successifs

Il peut arriver que vous ayez besoin de jouer séquentiellement plusieurs sons. Lorsque vous utilisez la fonction **IIPlaySound** et qu'un son est joué tout nouvel appel à cette fonction est ignoré tant que le premier son n'est pas terminé, et comme rien ne vous indique quand un son est terminé il est difficile d'enchaîner des sons. Mais la fonction **IISetSoundQueueing** a été créée spécialement pour apporter une solution à ce problème. Elle possède un seul paramètre de type **integer** qui sert de valeur booléenne. La valeur par défaut est **FALSE** et correspond au fonctionnement décrit ci-avant. Si on utilise cette fonction en transmettant **TRUE** au niveau du paramètre, un appel de la fonction **IIPlaySound** lors de l'émission d'un précédent son n'est plus ignorée mais simplement mise en mémoire de telle façon que le son correspondant démarre dès l'arrêt du son précédent. La seule limitation est dans le nombre d'appels gardés en mémoire : un seul... Donc si vous avez deux sons à jouer successivement pas de problème vous faites simplement ceci :

```
default {  
    state_entry() {  
        IIPreloadSound("Son1");  
        IIPreloadSound("Son2");  
    }  
    touch_start(integer total_number) {  
        IISetSoundQueueing(TRUE);  
        IIPlaySound("Son1", 1.0);  
        IIPlaySound("Son2", 1.0);  
    }  
}
```

Les problèmes commencent à apparaître si vous avez plus de deux sons à jouer. La seule façon de procéder est de se baser sur la durée connue de chaque son.

10.1.2 La musique

Pour avoir de la musique sur un terrain il faut déclarer une adresse HTTP de flux streaming. La fonction **llSetParcelMusicURL** permet de faire cela à partir d'un script. Son seul paramètre est un **string** contenant l'adresse HTTP du flux. La principale utilisation de cette fonction concerne les postes de radio ou jukebox qui mémorisent les adresses et permettent de changer la musique sur un simple clic de souris. Voici un exemple simplifié de radio :

```
list stations = ["http://hotmixradio80.hotmixradio.com:80", "http://64.62.253.225:8174"];
list noms = ["Radio 1", "Radio 2"];
integer ecouteMenu;
integer canalMenu = 25103;
default {
    touch_start(integer total_number) {
        key avatar = llDetectedKey(0);
        ecouteMenu = llListen(canalMenu, "", avatar, "");
        llDialog(avatar, "Choisissez une radio", noms, canalMenu);}

    listen(integer channel, string name, key id, string message) {
        if(channel == canalMenu) {
            llListenRemove(ecouteMenu);
            integer i = llListFindList(noms, [message]);
            llSetParcelMusicURL(llList2String(stations, i));}
    }
}
```

10.1.3 La video

La gestion de la video sur une parcelle est un peu spéciale. Il faut affecter une texture quelconque qui sert de lien avec la source video. Il suffit d'appliquer cette texture là où voulez voir apparaître la video. L'interface permet d'affecter cette texture, mais vous pouvez également le faire par script. La fonction qui permet de gérer la video est **llParcelMediaCommandList**. Cette fonction a un seul paramètre de type list qui permet toutes les actions au niveau de la video. Des constantes facilitent la programmation :

Constante	Val	Param	Description
PARCEL_MEDIA_COMMAND_STOP	0		Arrêt et retour au début
PARCEL_MEDIA_COMMAND_PAUSE	1		Stop sur image
PARCEL_MEDIA_COMMAND_PLAY	2		Démarrage et arrêt à la fin
PARCEL_MEDIA_COMMAND_LOOP	3		Démarrage et fonctionnement en boucle
PARCEL_MEDIA_COMMAND_TEXTURE	4	key	Clef de la texture support de la video
PARCEL_MEDIA_COMMAND_URL	5	string	Adresse de la video
PARCEL_MEDIA_COMMAND_TIME	6	float	Temporisation de démarrage

PARCEL_MEDIA_COMMAND_AGENT	7	key	Avatar qui a la commande
PARCEL_MEDIA_COMMAND_UNLOAD	8		Arrête la video et restitue la texture
PARCEL_MEDIA_COMMAND_AUTO_ALIGN	9	integer	Alignement automatique (TRUE/FALSE)

Pour obtenir des informations vous avez la fonction **llParcelMediaQuery** qui peut renvoyer la clef de la texture support de video et l'adresse de la video sous forme d'une **list**. Voici un exemple simple : vous créez un objet incluant la texture support dans son inventaire pour pouvoir récupérer facilement sa clef :

```
default {
    state_entry() {
        key clefTexture = llGetInventoryKey(llGetInventoryName(INVENTORY_TEXTURE, 0));
        llParcelMediaCommandList([PARCEL_MEDIA_COMMAND_TEXTURE, clefTexture,
                                PARCEL_MEDIA_COMMAND_URL, "http://mon.url/video"]);

        touch_start(integer total_number) {
            llParcelMediaCommandList([PARCEL_MEDIA_COMMAND_LOOP]);
            list l = llParcelMediaQuery([PARCEL_MEDIA_COMMAND_URL]);
            llSay(0, "Video en cours : " + llList2String(l, 0));
        }
    }
}
```

11. Agir sur un avatar

Tout d'abord une petite précision lexicale. Vous rencontrerez les termes **agent** et **avatar** souvent utilisés de façon similaire. Pour les puristes de SL un **agent** est une présence dans un simulateur. Dès que vous vous connectez à SL, vous vous trouvez pris en charge par un simulateur dont vous devenez un client, un **agent**. Un **agent** voit le mode de SL grâce à une caméra et possède un **avatar** pour que les autres utilisateurs le voient et puissent interagir avec lui. Un **avatar** porte des habits, a des attachements, peut être animé, et est affecté par les lois de la physique. Dans ce chapitre j'utiliserai uniquement le terme **avatar**.

11.1 Communication

Un avatar peut communiquer par l'intermédiaire du Chat. On peut lui envoyer des messages à l'aide des fonctions qui écrivent dans le Chat ou en IM:

Fonction	Portée
llInstantMessage	Avatar ciblé
llOwnerSay	Propriétaire
llWhisper	10 mètres
llSay	20 mètres
llShout	100 mètres
llRegionSay	Région

```
default {
    touch(integer total_number) {
        llInstantMessage(llDetectedKey(0), "Vous avez m'avez touche !");
        string nom = llDetectedName(0);
        llOwnerSay(nom + " m'a touche !");
        llShout(0, nom + " m'a touche !");}
}
```

Dans ce script on avise l'**avatar** qui a touché l'objet par **IM**, le propriétaire de l'objet par message personnel, et tous les **avatars** environnants par le Chat. Une grande discrétion !

On peut aussi écouter ce que raconte un **avatar** au niveau du Chat, nous en avons vu un exemple dans la présentation de l'événement **listen** à la page 60.

Une autre façon pour un avatar d'envoyer des informations est d'utiliser un **Dialog**. Il s'agit de ces fenêtres bleues qui apparaissent dans le coin supérieur droit et qui comportent des boutons à cliquer pour choisir des actions. La fonction à utiliser est **llDialog**. Voyons ses paramètres :

Numéro	Nom	Type	Fonction
1	avatar	key	Clé de l'avatar qui doit voir le Dialog
2	message	string	Message à afficher en tête du Dialog
3	button	list	Liste des noms des boutons à afficher
4	channel	integer	Canal du Chat à utiliser

Le troisième paramètre amène un petit commentaire. Nous avons vu le type **list** à la page 28, dans notre cas nous allons mémoriser des valeurs de type **string**. Chacune de ces valeurs constitue le texte à afficher sur un bouton. Le nombre de boutons nous est donc donné par le nombre d'éléments dans la liste.

Le quatrième paramètre concerne le canal du Chat à utiliser. Pour rappel le Chat dans lequel tout le monde peut se lire est le canal 0 (constante **0**). Mais il existe bien d'autres canaux, en fait 2147483647 (limite du type **integer**). Nous pouvons utiliser celui que nous voulons comme paramètre de notre fonction. Je vous conseille des valeurs fantaisistes pour éviter les mélanges !

Observez cet exemple simple :

```
list menu;
integer ecoute;
default {
    state_entry() {
        menu = ["un", "deux", "trois", "quatre"];
    }
    touch(integer total_number) {
        key avatar = IIDetectedKey(0);
        ecoute = IIListen(56124, "", avatar, "");
        IIDialog(avatar, "Choisissez une option", menu, 56124);
    }
    listen(integer channel, string name, key id, string message) {
        IIListenRemove(ecoute);
        IIListenRemove(ecoute);
    }
}
```

Pour la compréhension de l'événement **listen** reportez-vous à la page 60. En ce qui concerne la fonction **IIDialog** nous remarquons que nous avons sélectionné l'**avatar** qui a cliqué sur l'objet avec le premier paramètre. Le deuxième paramètre définit le message en tête du **Dialog**. Le troisième paramètre fait référence à une liste qui contient 4 valeurs de type **string** pour l'affichage des boutons. Enfin le dernier paramètre définit un canal du **Chat**. Lorsque l'**avatar** clique sur un bouton l'événement **listen** est déclenché et on affiche un message.

Tout ceci fonctionne très bien mais qu'arrive-t-il si l'on atteint la limite du nombre de boutons qui est de 12 ? Vous avez certainement remarqué qu'il est possible d'avoir des sous-menus qui se déclenchent par un clic sur un des boutons. Comment cela peut-il se réaliser ? Voici un exemple :

```
list menu1;
list menu2;
integer ecoute;
key avatar;
integer canal;
default {
```



```

state_entry() {
    menu1 = ["un", "deux", "trois", "quatre", "cinq", "suite"];
    menu2 = ["six", "sept", "huit", "neuf", "dix", "onze", "douze", "retour"];
    canal = 56124;
    touch(integer total_number) {
        avatar = lldetectedkey(0);
        écoute = llListen(canal, "", avatar, "");
        llDialog(avatar, "Choisissez une option", menu1, canal);}
    listen(integer channel, string name, key id, string message) {
        if (message == "suite")
            llDialog(avatar, "Choisissez une option", menu2, canal);
        else if (message == "retour")
            llDialog(avatar, "Choisissez une option", menu1, canal);
        else {
            llWhisper(0, "Vous avez choisi l'option " + message);
            llListenRemove(écoute);}
    }
}

```

Cet exemple est je pense suffisamment explicite sur la méthode à adopter. J'ai remarqué dans les forums que cette question revient souvent. J'espère que cette trame de base sera suffisante pour l'élaboration de tous les **Dialog** que vous pourrez imaginer.

Mais pour ceux qui auraient quand même des difficultés j'ai préparé cette routine :

```

//-----
//
//      Menu Manager V 1.0
//      by Bestmomo Lagan
//
// -----
//      Variables de paramétrage
// -----
float      timeOut          =      60.0;
integer     canal           =      60258;
string      PREVIOUS        =      "<< Previous";
string      NEXT            =      "Next >>";
// -----
//      Variables de travail
// -----

```

```

integer    écouteMenu      =    0;
integer    canalMenu      =    0;
integer    index          =    0;
list       listeGlobale   =    [];
string     texte          =    "";

// -----
//          Gestion de l'écoute du menu
// -----

updateEcoule(key id) {
    cancelEcoule();
    canalMenu = genCanal();
    écouteMenu = IIListen(canalMenu, "", id, "");
    IISetTimerEvent(timeOut);}

// -----

cancelEcoule() {
    if(ecouteMenu > 0) {
        IIListenRemove(ecouteMenu);
        écouteMenu = 0;
        IISetTimerEvent(.0);}
}

// -----
//          Canal aléatoire
// -----

integer genCanal() {return IIFloor(IIFrand(2000000.0));}

// -----
//          Gestion premier menu
// -----

gestPremierMenu(key id) {
    IIDialog(id, texte, IIListInsertList(IIList2List(listeGlobale, 0, 10), [NEXT], 2), canalMenu);}

// -----
//          Gestion menus autres que le premier
// -----

gestMenu(key id) {
    list l = IIDeleteSubList(listeGlobale, 0, index -1);

```

```

if(llGetListLength(l) > 11)
    l = llListInsertList(llDeleteSubList(l, 10, -1), [NEXT], 1);
llDialog(id, texte, [PREVIOUS] + 1, canalMenu);}

// -----
//          Etat par défaut
// -----

default {
    link_message(integer sender_number, integer number, string message, key id) {
        if(number == canal) {
            index = 0;
            listeGlobale = llParseString2List(message, ["|"], []);
            texte = llList2String(listeGlobale, 0);
            listeGlobale = llDeleteSubList(listeGlobale, 0, 0);
            updateEcoule(id);
            if(llGetListLength(listeGlobale) < 13)
                llDialog(id, texte, listeGlobale, canalMenu);
            else
                gestPremierMenu(id);}
    }

    listen(integer channel, string name, key id, string message) {
        if(channel == canalMenu) {
            if(message == NEXT) {
                if(index == 0) index += 11;
                else index += 10;
                gestMenu(id);}
            else if(message == PREVIOUS) {
                if(index == 11) {
                    index = 0;
                    gestPremierMenu(id);}
                else {
                    index -= 10;
                    gestMenu(id);}
            }
        }
    }
}

```

```

        else {
            cancelEcoule();
            IIMessageLinked(LINK_THIS, canal + 1, message, id);}
        }
    }
    timer() {cancelEcoule();}
}

```

avec ce script de test :

```

integer      canal = 60258;
appelMenu(string texte, list menu) {
    IIMessageLinked(LINK_THIS, canal, texte + ", " + IIList2CSV(menu), IIDetectedKey(0));}
default {
    touch_start(integer total_number) {
        integer c;
        list l;
        for(c = 1; c < 50; c++) l += (string)c;
        appelMenu("Choisissez : ", l);}

    link_message(integer sender_number, integer number, string message, key id) {
        if(number == canal + 1) IIOwnerSay("Vous avez choisi " + message);}
    }
}

```

11.2 Attachements

On peut attacher des tas de choses à un **avatar**. La fonction **IIMAttachToAvatar** permet d'attacher l'objet à l'**avatar**, mais il faut lui demander la permission. Observez cet exemple :

```

default {
    touch_start(integer total_number) {
        key avatar = IIDetectedKey(0);
        IIRequestPermissions(avatar, PERMISSION_ATTACH);}
    run_time_permissions(integer permissions) {
        if(permissions & PERMISSION_ATTACH) {
            IIMAttachToAvatar(ATTACH_HEAD);
            integer a = IIMGetAttached();
            IIMWhisper(0, "Point d'attachement : " + (string)a);
            IISleep(60.0);
            IIMDetachFromAvatar();}
    }
}

```

```

}
attach(key attached) {
    llSendMessage(attached, "Merci de me porter sur la tete !");}
}

```

Pour la compréhension de l'événement **run_time_permissions** reportez-vous à la page 67. Pour la compréhension de l'événement **attach** reportez-vous à la page 71. Vous remarquez que la fonction **llAttachToAvatar** possède un seul paramètre qui définit l'emplacement de l'attachement. Celui-ci nous est donné par l'utilisation d'une constante. Voici les constantes disponibles pour cette fonction :

Constante	Emplacement
ATTACH_BACK	Dos
ATTACH_BELLY	Ventre
ATTACH_CHEST	Poitrine
ATTACH_CHIN	Menton
ATTACH_HEAD	Tête
ATTACH_LEAR	Oreille gauche
ATTACH_LEYE	Oeil gauche
ATTACH_LFOOT	Pied gauche
ATTACH_LHAND	Main gauche
ATTACH_LHIP	Hanche gauche
ATTACH_LLARM	Avant-bras gauche
ATTACH_LLLEG	Jambe gauche
ATTACH_LPEC	Pectoral gauche
ATTACH_LSHOULDER	Epaule gauche
ATTACH_LUARM	Bras gauche
ATTACH_LULEG	Cuisse gauche
ATTACH_MOUTH	Bouche
ATTACH_NOSE	Nez
ATTACH_PELVIS	Pelvis
ATTACH_REAR	Oreille droite
ATTACH_REYE	Oeil droit
ATTACH_RFOOT	Pied droit
ATTACH_RHAND	Main droite
ATTACH_RHIP	Hanche droite
ATTACH_RLARM	Avant-bras droit
ATTACH_RLLEG	Jambe droite
ATTACH_RPEC	Pectoral droit
ATTACH_RSHOULDER	Epaule droite
ATTACH_RUARM	Bras droit
ATTACH_RULEG	Cuisse droite

Pour détacher l'objet nous utilisons la fonction **llDetachFromAvatar**. Pour connaître le point d'attachement d'un objet nous utilisons la fonction **llGetAttached**.

11.3 Animation

L'animation d'un **avatar** est certainement l'une des activités les plus intéressantes de SL. Il y a des animations intrinsèques dans **SL** pour les attitudes les plus courantes, vous pouvez en trouver la liste dans le **WIKI**. Ces animations sont mises en oeuvre par les fonctions **llStartAnimation** et **llStopAnimation**. Ces deux fonctions attendent pour seul paramètre de type **string** le nom de l'animation. Ainsi la ligne de code :

```
llStartAnimation("express_cry");
```

démarré des pleurs alors que celle-ci les arrête :

```
llStopAnimation("express_cry");
```

Mais pour animer un **avatar** il faut lui demander la permission, sinon où on va ? Pour cela il faut utiliser la fonction **llRequestPermissions** dont vous trouvez une explication détaillée au point 8.7, avec un exemple simple d'animation.

Vous pouvez utiliser une animation non incluse dans **SL** mais alors elle doit se trouver dans l'objet qui contient le script, ça ne fonctionne pas avec la clé.

Il est parfois utile de connaître toutes les animations en cours pour un **avatar**, la fonction **llGetAnimationList** permet de le savoir. On transmet la clé de l'**avatar** comme paramètre et elle nous renvoie une **list** de toutes les clés des animations en cours. Voici un exemple de script pour stopper toutes les animations en cours :

```
default {
    touch_start(integer total_number) {
        llRequestPermissions(llDetectedKey(0), PERMISSION_TRIGGER_ANIMATION);}
    run_time_permissions(integer permissions) {
        if(permissions & PERMISSION_TRIGGER_ANIMATION) {
            list l = llGetAnimationList(llGetPermissionsKey());
            integer i;
            for (i = 0; i < llGetListLength(l); ++i) llStopAnimation(llList2String(l, i));}
        }
    }
}
```

11.4 Obtenir des informations sur l'avatar

La fonction **llGetAgentSize** permet de connaître la taille d'un **avatar** :

```
default {
    touch_start(integer total_number) {
        vector v = llGetAgentSize(llDetectedKey(0));
        llWhisper(0, "Taille : " + (string)v.z + " metres");}
    }
}
```

```
}
```

Pour obtenir d'autres informations il faut utiliser la fonction **llGetAgentInfo** :

```
default {  
    touch_start(integer total_number) {  
        key avatar = llDetectedKey(0);  
        integer i = llGetAgentInfo(avatar);  
        if(i & AGENT_ALWAYS_RUN)  
            llWhisper(0, "Vous courez tout le temps");  
        if(i & AGENT_FLYING)  
            llWhisper(0, "Vous volez");  
        if(i & AGENT_ATTACHMENTS)  
            llWhisper(0, "Vous avez des attachements");  
        if(i & AGENT_MOUSELOOK)  
            llWhisper(0, "Vous etes en mode MOUSELOOK");  
        if(i & AGENT_SITTING)  
            llWhisper(0, "Vous etes assis");  
        if(i & AGENT_ON_OBJECT)  
            llWhisper(0, "Vous etes assis sur un objet");  
        if(i & AGENT_AWAY)  
            llWhisper(0, "Vous etes en mode AWAY");  
        if(i & AGENT_WALKING)  
            llWhisper(0, "Vous marchez");  
        if(i & AGENT_IN_AIR)  
            llWhisper(0, "Vous etes dans les airs");  
        if(i & AGENT_TYPING)  
            llWhisper(0, "Vous etes en train de pianoter");  
        if(i & AGENT_CROUCHING)  
            llWhisper(0, "Vous etes accroupi !!!");  
        if(i & AGENT_BUSY)  
            llWhisper(0, "Vous etes en mode BUSY");  
    }  
}
```

Vous trouvez une autre façon d'obtenir des informations sur un **avatar** au point 8.13.2.

Une fonction plus récente permet de connaître le langage de l'avatar, c'est **llGetAgentLanguage**. Cette fonction demande pour seul paramètre la clef de l'avatar et retourne un string qui référence le langage du client :

Valeur de retour	Langue
de	Allemand
en-us	Anglais
es	Espagnol
fr	Français
ja	Japonais
pt	Portugais
ko	Coréen
zh	Chinois

Il est ainsi possible d'adapter les textes selon l'interface choisi. Voici une trame de base pour cette adaptation :

```
listLANGUE = ["en-us", "fr"];
listT = ["Hello", "Bonjour"];
integer localise;

// -----
//  Localisation de la langue
// -----

string localisation(list l) {
    return llList2String(l, localise);}

// -----

trouveLangue(key id) {
    localise = llListFindList(LANGUE, [llGetAgentLanguage(id)]);
    if(localise == -1) localise = 0;}

// -----
//  Etat par défaut
// -----

default
{
    state_entry() {
        trouveLangue(llGetOwner());}

    changed(integer change) {
        if(change & CHANGED_OWNER)
```



```
trouveLangue(llGetOwner());}

touch_start(integer total_number) {
    llOwnerSay(localisation(T));}
}
```

Il peut être judicieux de donner le choix de la langue à partir d'un menu.

11.5 Caméra

La caméra est un outil important dans SL. Il existe deux modes principaux selon que le point de vue à pour origine les yeux de l'avatar (**Mouselook**) ou un point autour de lui. La manipulation du second mode est vraiment très pratique dans le jeu, ne serait-ce que pour explorer son environnement. Quant au premier mode il est utilisé pour les jeux, les armes, les véhicules et la saisie des objets. LSL offre 8 fonctions pour gérer la caméra.

11.5.1 Mode Mouselook

Une fonction très utilisée est **llForceMouselook**. Elle comporte un seul paramètre de type **integer**. Elle a pour but de forcer le mode **Mouselook** pour un avatar qui s'assoit. Il suffit que cette ligne s'exécute dans un script :

```
llForceMouselook(TRUE);
```

pour que le prim correspondant mémorise le fait qu'un avatar qui s'assoit passe automatiquement en mode **Mouselook** même si on supprime ensuite le script. C'est exactement le même fonctionnement que pour le **SitTarget**. La seule façon de désactiver cette fonction est de l'exécuter à nouveau en fixant son paramètre à **FALSE**:

```
llForceMouselook(FALSE);
```

11.5.2 Régler la caméra pour un avatar assis

Vous avez créé un joli siège et vous avez envie que l'avatar voit d'une certaine façon lorsqu'il s'assoit. Par exemple que le regard parte d'un point situé 2 mètres au-dessus de lui, 1 mètre en arrière et qu'il fixe un point situé 10 mètres devant lui et au niveau de ses yeux. Voici le code à utiliser :

```
llSetCameraEyeOffset(<-1.0,.0,2.0>);
llSetCameraAtOffset(<10.0,.0,.0>);
```

Ces deux fonctions appellent peu de commentaires si ce n'est qu'il faut les exécuter avant que l'avatar s'assoit pour que leur action soit prise en compte. Pour annuler leur effet il suffit de passer la valeur **ZERO_VECTOR** comme argument.

11.5.3 Où est la caméra ?

Il est souvent nécessaire de connaître la position et la rotation de la caméra. Deux fonctions sont affectées à cette tâche et présentent des noms tout à fait explicites :

```
touch_start(integer total_number) {
    llOwnerSay("Position de la camera : " + (string)llGetCameraPos());
    llOwnerSay("Rotation de la camera : " + (string)llGetCameraRot());}
```

Ces fonctions renvoient une valeur non nulle pour la caméra de l'avatar pour lequel une autorisation **PERMISSION_TRACK_CAMERA** a été demandée dans le même script.

11.5.4 Contrôle de la caméra

LSL vous permet de contrôler tous les paramètres de la caméra de manière efficace pour des scripts présents dans un attachement ou un véhicule. Il est nécessaire d'obtenir l'autorisation **PERMISSION_CONTROL_CAMERA** pour effectuer ce contrôle (voir point 8.7). La fonction **llSetCameraParams** comporte une liste de paramètres pour régler les mouvements de la caméra :

Règle	Valeur	Type	Défaut	Plage	Description
CAMERA_ACTIVE	12	integer	FALSE	TRUE ou FALSE	Marche ou arrêt du contrôle de la caméra
CAMERA_BEHINDNESS_ANGLE	8	float	10.0	0 à 180	Fixe l'angle sans contrainte de changement de positionnement de la caméra
CAMERA_BEHINDNESS_LAG	9	float	0.0	0 à 3	Vitesse de repositionnement de la caméra
CAMERA_DISTANCE	7	float	3.0	0.5 à 10	Distance par rapport à la cible
CAMERA_FOCUS	17	vector	n/a	n/a	Orientation de la caméra en coordonnées globales
CAMERA_FOCUS_LAG	6	float	0.1	0 à 3	Vitesse de visée
CAMERA_FOCUS_LOCKED	22	integer	False	TRUE ou FALSE	Verrouillage de la visée
CAMERA_FOCUS_OFFSET	1	vector	<0.0,0.0,0.0>	<-10,-10,-10> à <10,10,10>	Offset de la visée
CAMERA_FOCUS_THRESHOLD	11	float	1.0	0 à 4	Rayon de la sphère de mouvement de la visée sans mouvement de la caméra
CAMERA_PITCH	0	float	0.0	-45 à 80	Incidence
CAMERA_POSITION	13	vector	n/a	n/a	Position de la caméra
CAMERA_POSITION_LAG	5	float	0.1	0 à 3	Vitesse de la positionnement idéal
CAMERA_POSITION_LOCKED	21	integer	FALSE	TRUE ou FALSE	Verrouillage de la caméra
CAMERA_POSITION_THRESHOLD	10	float	1.0	0 à 4	Rayon de la sphère de mouvement de la caméra sans perdre la cible

La fonction **llClearCameraParams** remet tous les paramètres à leur valeur par défaut et la fonction **llReleaseCamera** redonne le contrôle de la caméra à l'avatar.

12. Agir sur un terrain

SL est divisé en simulateurs de dimensions 256 m². La hauteur du terrain de chaque simulateur est stockée dans une grille de résolution 1 mètre. En d'autres termes ça nous donne un maillage de 256x256 points de référence pour la hauteur du terrain. La hauteur du terrain entre ces points de référence est interpolée à partir des points avoisinants pour obtenir une progression fluide. Cette hauteur est mesurée sur l'axe Z alors que la localisation sur le simulateur utilise les axes X et Y. Le terrain est en continu, il n'y a pas de possibilité de faire des trous. Un simulateur peut être divisé en parcelles de dimensions minimales 4x4.

Les outils d'édition de l'interface permettent de modifier le terrain. LSL permet les mêmes modifications avec une batterie de fonctions efficaces.

12.1 Relief d'un terrain

12.1.1 Modifier le relief

La fonction **llModifyLand** permet toutes les modifications du terrain. Mais l'avatar doit être présent sur le simulateur pour que le script fonctionne. Cette fonction possède deux paramètres :

Paramètre	Description
action	Action à effectuer
size	Dimension de l'effet

Il existe des constantes pratiques pour renseigner ces paramètres :

action	Valeur	Description
LAND_LEVEL	0	Met le terrain à la hauteur du centre de l'objet contenant le script
LAND_RAISE	1	Soulève le terrain à la hauteur du centre de l'objet contenant le script
LAND_LOWER	2	Abaisse le terrain à la valeur 0
LAND_SMOOTH	3	Uniformise le terrain
LAND_NOISE	4	Crée des déformations aléatoires sur le terrain
LAND_REVERT	5	Annulation

size	Valeur	Description
LAND_SMALL_BRUSH	1	Petite brosse 2m x 2m
LAND_MEDIUM_BRUSH	2	Brosse moyenne 4m x 4m
LAND_LARGE_BRUSH	3	Grande brosse 8m x 8m

A l'inverse, pour connaître la hauteur du terrain à un point particulier, utilisez la fonction **llGround** qui renvoie un **float** donnant la hauteur du terrain et a pour seul paramètre un **offset** éventuel par rapport à la position du centre de l'objet.

Imaginez que vous désirez créer un outil pour relever un terrain simplement. Créez un box plat et mettez ce script dedans :

```

default {
    state_entry() { llSetTimerEvent(.2); }
    timer() {
        vector posObjet = llGetPos();
        if(llGround(ZERO_VECTOR) < posObjet.z)
            llModifyLand(LAND_RAISE, LAND_LARGE_BRUSH);
    }
}

```

Un timer est implémenté avec un délai de 0,2 seconde. Donc, régulièrement on mesure la hauteur du sol grâce à la fonction **llGround** et on relève celui-ci s'il est inférieur à la position du centre de l'objet grâce à la fonction **llModifyLand**. Vous n'avez plus qu'à promener votre objet sur votre terrain pour apprécier le résultat. On peut imaginer des tas d'autres outils dans le même style.

12.1.2 S'informer sur le relief

Il est parfois nécessaire de connaître l'inclinaison du sol à un point donné. La fonction **llGroundNormal** retourne un vecteur représentant la normale au point où se situe l'objet comportant le script modifié éventuellement par un offset passé en paramètre. Pour mémoire la normale à un plan est le vecteur perpendiculaire à ce plan. Sur un sol parfaitement plat ce vecteur est aligné sur l'axe Z : <0,0,1>. Sur un sol incliné à 45° sur l'axe X vous obtenez un vecteur de valeur <.0707,0,.707>.

La fonction **llGroundSlope** renvoie une valeur orthogonale à celle renvoyée par la fonction **llGroundNormal**. Autrement dit le vecteur obtenu est parallèle au sol. Elle possède aussi un paramètre permettant de préciser un offset.

12.2 Gérer un terrain

12.2.1 Obtenir des renseignements pour une région

Il est souvent utile de connaître un certain nombre de renseignements concernant une région. Evidemment vous obtenez tous ces renseignements à partir de l'interface de SL. Mais vous pouvez en avoir également besoin dans un script. La fonction à connaître est certainement **llGetRegionFlags**. Cette fonction sans paramètre renvoie une valeur integer qui est à considérer comme un ensemble de bits significatifs. Autrement dit nous obtenons des informations de type tout ou rien (référez-vous au point 2.2.2.1 pour une explication plus précise de ce système de stockage de donnée). Voici les informations livrées :

Nom	Valeur	Description
REGION_FLAG_ALLOW_DAMAGE	1	Dégâts autorisés
REGION_FLAG_FIXED_SUN	16	Position du soleil fixe
REGION_FLAG_BLOCK_TERRAFORM	64	Terraformation interdite
REGION_FLAG_SANDBOX	256	La région est un bac à sable
REGION_FLAG_DISABLE_COLLISIONS	4096	Collisions désactivées
REGION_FLAG_DISABLE_PHYSICS	16384	Objets physiques désactivés
REGION_FLAG_BLOCK_FLY	524288	Vol interdit
REGION_FLAG_ALLOW_DIRECT_TELEPORT	1048576	Téléportation directe autorisée
REGION_FLAG_RESTRICT_PUSHOBJECT	4194304	llPushObject désactivé

Exemple d'utilisation :

```
default {
    state_entry() {
        if(llGetRegionFlags() & REGION_FLAG_ALLOW_DIRECT_TELEPORT)
            llSay(0, "Cette region autorise les teleportations directes");
        else
            llSay(0, "Cette region a un lieu d'arrivee unique pour les teleportations");
    }
}
```

12.2.2 Obtenir des renseignements pour une parcelle

12.2.2.1 Commutateurs d'état

Il est souvent utile aussi de connaître un certain nombre de renseignements concernant une parcelle. La première fonction à connaître est certainement **llGetParcelFlags**. Cette fonction possède un paramètre de type vecteur qui indique un emplacement pour localiser la parcelle. Comme la fonction **llGetRegionFlags** renvoie une valeur integer qui est à considérer comme un ensemble de bits significatifs. Voici les informations livrées :

Nom	Valeur	Description
PARCEL_FLAG_ALLOW_FLY	1	Autorise le vol
PARCEL_FLAG_ALLOW_SCRIPTS	2	Autorise les scripts
PARCEL_FLAG_ALLOW_LANDMARK	8	Autorise les LandMarks
PARCEL_FLAG_ALLOW_TERRAFORM	16	Autorise la terraformation
PARCEL_FLAG_ALLOW_DAMAGE	32	Autorise les dégâts
PARCEL_FLAG_ALLOW_CREATE_OBJECTS	64	Autorise à créer des objets
PARCEL_FLAG_USE_ACCESS_GROUP	256	Limite l'accès au groupe
PARCEL_FLAG_USE_ACCESS_LIST	512	Limite l'accès à une liste de résidents
PARCEL_FLAG_USE_BAN_LIST	1024	Utilise une liste d'exclusions
PARCEL_FLAG_USE_LAND_PASS_LIST	2048	Utilise un droit de passage
PARCEL_FLAG_LOCAL_SOUND_ONLY	32768	Restriction sur les sons
PARCEL_FLAG_RESTRICT_PUSHOBJECT	2097152	Désactivation de llPushObject
PARCEL_FLAG_ALLOW_GROUP_SCRIPTS	33554432	Autorise les scripts du groupe
PARCEL_FLAG_ALLOW_CREATE_GROUP_OBJECTS	67108864	Autorise la création d'objets du groupe
PARCEL_FLAG_ALLOW_ALL_OBJECT_ENTRY	134217728	Permet l'entrée des objets
PARCEL_FLAG_ALLOW_GROUP_OBJECT_ENTRY	268435456	Permet seulement l'entrée des objets du groupe

Exemple d'utilisation :

```
default {
    touch_start(integer total_number) {
        if(llGetParcelFlags(llGetPos()) & PARCEL_FLAG_USE_ACCESS_GROUP) {
            llSay(0, "Cette parcelle n'est accessible qu'aux membres du groupe");
            if(llDetectedGroup(0))

```

```

    IISay(0, "et vous etes un heureux membre de ce groupe, vous pouvez passer !");
else
    IISay(0, "et vous n'etes pas membre de ce groupe ou votre groupe est inactif.");}
}
}

```

12.2.2.2 Détails

Vous pouvez obtenir plus de détails sur une parcelle en utilisant la fonction **IIGetParcelDetails**. Cette fonction attend deux paramètres: la position (**vector**) pour déterminer de quelle parcelle il s'agit, et les détails (**list**) désirés déterminés par les constantes suivantes :

Constante	Type	Valeur	Description
PARCEL_DETAILS_NAME	string	1	Nom de la parcelle
PARCEL_DETAILS_DESC	string	2	Description de la parcelle
PARCEL_DETAILS_OWNER	key	3	Clef du propriétaire de la parcelle
PARCEL_DETAILS_GROUP	key	4	Clef du groupe de la parcelle
PARCEL_DETAILS_AREA	integer	5	Superficie de la parcelle en m²

La fonction renvoie une liste contenant les détails demandés. Voici un exemple d'utilisation :

```

default {
    touch_start(integer total_number) {
        list details = IIGetParcelDetails(IIGetPos(), [PARCEL_DETAILS_NAME,
        PARCEL_DETAILS_DESC]);
        IISay(0, "Cette parcelle s'appelle " + IIList2String(details, 0));
        IISay(0, "Sa description est " + IIList2String(details, 1));}
}

```

12.2.2.3 Les primitives de la parcelle

Combien de primitives sur la parcelle ? A qui appartiennent-elles ? Combien de primitives au maximum ? Voici des questions fréquentes qui trouvent leur réponse dans des fonctions simples.

La fonction **IIGetParcelMaxPrims** renvoie le nombre de primitives maximum admis. Son premier paramètre de type **vector** permet de localiser la parcelle, le second de type **integer** est un commutateur tout ou rien, s'il est à **TRUE** la fonction renvoie le nombre maximum de primitives admises sur la région, et s'il est à **FALSE** le nombre maximum admis sur la parcelle.

La fonction **IIGetParcelPrimCount** donne le nombre de primitives utilisées sur la parcelle. Son premier paramètre de type **vector** permet de localiser la parcelle, le second de type **integer** la catégorie concernée (voir le tableau ci-après), le troisième de type **integer** est un commutateur tout ou rien, s'il est à **TRUE** la fonction renvoie le nombre de primitives utilisées sur la région pour la catégorie concernée, et s'il est à **FALSE** le nombre de primitives utilisées sur la parcelle pour la catégorie concernée. Voici les catégories disponibles :

Catégorie	Valeur	Description
PARCEL_COUNT_TOTAL	0	Tous les prims non temporaires

PARCEL_COUNT_OWNER	1	Prims du propriétaire
PARCEL_COUNT_GROUP	2	Prims du groupe mais pas du propriétaire
PARCEL_COUNT_OTHER	3	Prims ni du groupe ni du propriétaire
PARCEL_COUNT_SELECTED	4	Prims sélectionnés
PARCEL_COUNT_TEMP	5	Prims temporaires

La fonction **llGetParcelPrimOwners** renvoie une liste du nombre de prims par utilisateur. Son seul paramètre de type **vector** permet de localiser la parcelle. La liste retournée est une “strided list” (voir le point 3.8.7 pour une présentation de ce type de liste) de la forme [**key** agentKey1, **integer** agentCount1, **key** agentKey2, **integer** agentCount2...].

Voici un exemple d’utilisation de ces trois fonctions :

```
integer c;
integer n;
list l;
default {
    touch_start(integer total_number) {
        vector pos = llGetPos();
        llSay(0, "Nombre de prims admis sur la region : "
            + (string)llGetParcelMaxPrims(pos, TRUE));
        llSay(0, "Nombre de prims admis sur la parcelle : "
            + (string)llGetParcelMaxPrims(pos, FALSE));
        llSay(0, "Nombre de prims total sur la region : "
            + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_TOTAL, TRUE));
        llSay(0, "Nombre de prims total sur la parcelle : "
            + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_TOTAL, FALSE));
        llSay(0, "Nombre de prims du proprietaire sur la region : "
            + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_OWNER, TRUE));
        llSay(0, "Nombre de prims du proprietaire sur la parcelle : "
            + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_OWNER, FALSE));
        llSay(0, "Nombre de prims du groupe sur la region : "
            + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_GROUP, TRUE));
        llSay(0, "Nombre de prims du groupe sur la parcelle : "
            + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_GROUP, FALSE));
        llSay(0, "Nombre de prims des autres sur la region : "
            + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_OTHER, TRUE));
        llSay(0, "Nombre de prims des autres sur la parcelle : "
```

```

        + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_OTHER, FALSE));
    llSay(0, "Nombre de prims selectionnes sur la region : "
        + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_SELECTED, TRUE));
    llSay(0, "Nombre de prims selectionnes sur la parcelle : "
        + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_SELECTED, FALSE));
    llSay(0, "Nombre de prims temporaires sur la region : "
        + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_TEMP, TRUE));
    llSay(0, "Nombre de prims temporaires sur la parcelle : "
        + (string)llGetParcelPrimCount(pos, PARCEL_COUNT_TEMP, FALSE));
    l = llGetParcelPrimOwners(pos);
    n = llGetListLength(l);
    c = 0;
    llRequestAgentData(llList2Key(l, c), DATA_NAME);}
dataserver(key requested, string data) {
    llSay(0, data + " : " + (string)llList2Integer(l, c + 1));
    c += 2;
    if(c < n) llRequestAgentData(llList2Key(l, c), DATA_NAME);}
}

```

12.2.3 Gérer les avatars

12.2.3.1 Liste d'accès

Pour gérer l'accès à un terrain la liste d'accès est une méthode efficace. La méthode **llAddToLandPassList** permet d'ajouter un nom à la liste d'accès. Son premier paramètre est la clef de l'avatar concerné, le second est un **float** indiquant la durée de l'autorisation en heures. Une valeur nulle au niveau de ce deuxième paramètre indique une durée infinie. La réciproque de cette fonction est **llRemoveFromLandPassList** qui comporte comme seul paramètre la clef de l'avatar. Imaginez que vous désirez donner l'accès à votre terrain pour une durée d'une demie-heure. Il suffit de cliquer sur une borne pour obtenir cet accès. Et bien entendu cette action ne doit être active qu'une seule fois par avatar. Voici le genre de script qui vous permet d'obtenir ce résultat :

```

list listeAcces;
default {
    touch_start(integer total_number) {
        key avatar = llDetectedKey(0);
        if(llListFindList(listeAcces, [avatar]) == -1) {
            listeAcces += avatar;
            llAddToLandPassList(avatar, .5);}
    }
}

```



```
else llSendMessage(avatar, "Vous avez déjà utilisé votre droit d'accès");}

}
```

12.2.3.2 Liste de bannissement

De la même façon que pour la liste d'accès vous avez également des fonction permettant de gérer la liste de bannissement. La fonction **llAddToLandBanList** permet d'ajouter un nom à la liste de bannissement. Son premier paramètre est la clef de l'avatar concerné, le second est un **float** indiquant la durée de bannissement en heures. Une valeur nulle au niveau de ce deuxième paramètre indique une durée infinie. La réciproque de cette fonction est **llRemoveFromLandBanList** qui comporte comme seul paramètre la clef de l'avatar.

Supposez que vous avez mis en place des campings pour faire monter la fréquentation de votre sim. Ce n'est pas très élégant mais ça permet aux noob de se faire trois sous. Mais vous ne voulez pas que les avatars se mettent en mode AWAY et squattent indéfiniment vos camps. Vous pouvez installer un système de détection et bannir les avatars pour une certaine durée pour leur apprendre ce que vous considérez comme des bonnes manières. Voici le genre de code à mettre en place :

```
default {
    state_entry() {llSensorRepeat("", NULL_KEY, AGENT, 10, TWO_PI, 20.0);}
    sensor(integer total_number) {
        integer c;
        for(c = 0; c < total_number; c++) {
            key avatar = llDetectedKey(c);
            if(llGetAgentInfo(avatar) & AGENT_AWAY) {
                llSendMessage(avatar, "Le mode AWAY est interdit, vous êtes banni pour une
                durée de 10 minutes");
                llAddToLandBanList(avatar, 1.0/6.0);}
        }
    }
}
```

13. La gestion du temps

13.1 Connaître l'heure

13.1.1 Heure PST ou PDT

Vous avez sans doute remarqué que l'heure dans **SL** est celle de la Côte Ouest des USA, tout simplement parce que les serveurs sont situés là bas. Pour retrouver cette valeur par script il suffit d'utiliser la fonction **llGetWallclock**. Cette fonction renvoie une valeur de type **float** qui est le nombre de secondes (et fractions de secondes) écoulées depuis minuit. Mais ce n'est pas si simple parce qu'ils changent d'heure en cours d'année, on se retrouve donc avec deux types de temps selon la période de l'année: Pacific Standard Time (PST) du dernier dimanche d'octobre à 2h00 du matin au premier dimanche d'avril à 2h00 du matin également, pendant cette période le temps est GMT (Greenwich Mean Time) - 8. Le reste de l'année c'est le Pacific Daylight Time (PDT) avec comme référence temporelle GMT - 7.

13.1.2 Heure GMT ou UTC

Si vous préférez obtenir l'heure GMT, qui elle ne change pas en cours d'année et constitue la référence universelle (en fait remplacée le 1er janvier 1972 par l'Universal Time), utilisez la fonction **llGetGMTclock**. Vous pouvez passer à un temps local si vous connaissez la différence par rapport au temps GMT, en tenant compte évidemment des changements d'heure. Il est ainsi relativement facile de créer une horloge, du moins si vous vous sentez à l'aise avec les changements d'heure et que vous savez convertir les secondes en heures et minutes ! Voici par exemple un script qui donne l'heure d'été pour la France lorsqu'on clique sur l'objet :

```
integer decalage = 7200;
default {
    touch_start(integer total_number) {
        integer temps = (integer)llGetGMTclock() + decalage;
        integer heures = temps / 3600;
        integer minutes = (temps % 3600) / 60;
        integer secondes = temps % 60;
        llOwnerSay("Il est " + (string)heures + ":" + (string)minutes + ":" + (string)secondes);}
}
```

Pour peaufiner la sortie il faudrait s'arranger pour avoir toujours au moins 2 caractères. Voici une façon de modifier le code pour y parvenir :

```
integer decalage = 7200;
string ajusteSortie(integer valeur) {
    if(valeur < 10) return "0" + (string)valeur;
    else return (string)valeur;}
}
```

```
default {  
    touch_start(integer total_number) {  
        integer temps = (integer)llGetGMTclock() + decalage;  
        integer heures = temps / 3600;  
        integer minutes = (temps % 3600) / 60;  
        integer secondes = temps % 60;  
        llOwnerSay("Il est " + ajusteSortie(heures) + ":" + ajusteSortie(minutes) + ":" + ajusteSortie(secondes));  
    }  
}
```

13.1.3 Heure SL

Pour simplifier encore un peu plus notre tâche il faut savoir qu'un jour dans SL ne dure en fait que 4 de nos heures réelles. Le soleil se lève vers 0h30 et se couche vers 3h30, ce qui nous donne une durée de la journée de 3 heures pour 1 heure de nuit. La fonction **llGetTimeOfDay** nous donne cette valeur en secondes depuis minuit. Mais en cas de reboot le compteur est remis à zéro. D'autre part les propriétaires d'une île privée peuvent bloquer la position du soleil ! Dans ce dernier cas la valeur renvoyée par la fonction ne revient jamais à zéro et croît indéfiniment. Moralité : cette fonction est difficile à utiliser !

13.2 Connaître la date

La fonction **llGetDate** retourne la date en temps universel (UTC) sous la forme YYYY-MM-DD. Il est ainsi facile de manipuler la date dans un script :

```
default {  
    touch_start(integer total_number) {  
        list date = llParseString2List(llGetDate(), ["-"], []);  
        llOwnerSay("La date est " + llList2String(date, 2) + "/" +  
            llList2String(date, 1) + "/" +  
            llList2String(date, 0));  
    }  
}
```

Si vous désirez plus de précision utilisez la fonction **llGetTimestamp** qui vous donne en plus le moment exact dans la journée, le format de retour est *YYYY-MM-DDThh:mm:ss.ff.fZ*. Vous pouvez remarquer que cette fonction semble plus pratique à utiliser que **llGetGMTclock** pour obtenir l'heure parce que les valeurs des heures, minutes et secondes apparaissent explicitement. Le wiki nous propose une fonction pour extraire les heures, minutes et secondes de ce format compact. Je l'utilise dans le script ci-après qui est équivalent au précédent au niveau résultat :

```
float decalage = 2.0;  
string ajusteSortie(float valeur) {  
    string s = (string)(integer)valeur;
```

```

if(IIStringLength(s) == 1) return "0" + s;
else return s;}

vector time() {
    list timestamp = IIParseString2List( IIGetTimestamp(), ["T", ":", "Z"], [] );
    return < IIList2Float(timestamp, 1), IIList2Float(timestamp, 2), IIList2Float(timestamp, 3) >;}

default {
    touch_start(integer total_number) {
        vector heure = time();
        IIOwnerSay("Il est " + ajusteSortie(heure.x + decalage) + ":"
            + ajusteSortie(heure.y) + ":" + ajusteSortie(heure.z));}
}

```

Le vecteur de retour de la fonction du wiki contient les 3 valeurs : heures, minutes et secondes. Un exemple intéressant d'utilisation de la fonction **IIParseString2List**. Les trois séparateurs utilisés permettent d'isoler d'une part la date et de l'autre les éléments heures, minutes et secondes. Il suffit ensuite de formater correctement le résultat. Au final le code devient plus complexe que le précédent. J'ai un peu changé le code de la fonction **ajusteSortie** pour varier les plaisirs.

13.3 Connaître le temps écoulé

Nous avons vu l'événement **timer** activé par la fonction **IISetTimerEvent** au point Erreur : source de la référence non trouvée, c'est la façon la plus simple de gérer une durée dans un script. Mais on ne peut utiliser qu'un seul **timer** par **état** et pour certains scripts ce n'est pas suffisant. Il existe d'autres possibilités pour manipuler des intervalles de temps. La plus simple consiste à utiliser le timer associé au script. Lorsqu'un script démarre un timer se met en route. Vous pouvez connaître à tout moment où il en est avec la fonction **IIGetTime**. Vous pouvez aussi remettre ce timer à zéro avec la fonction **IIResetTime**. Mais l'inconvénient de ce timer est qu'il se remet aussi à zéro en cas de reboot de la sim, il peut être aussi victime de la dilatation du temps dont je vous parle plus loin. En conséquence il vaut mieux l'utiliser pour des durées courtes. Si vous désirez à la fois récupérer la valeur et remettre le timer à zéro utilisez la fonction **IIGetAndResetTime** qui accomplit les deux actions d'un seul coup. Observez le script suivant :

```

integer flag;

default {
    touch_start(integer total_number) {
        if(IIDetectedKey(0) == IIGetOwner()) {
            if(flag)
                IIOwnerSay("Vous avez déjà cliqué il y a " + (string)IIGetAndResetTime() + " secondes");
            else {
                flag = TRUE;
                IIResetTime();}
        }
    }
}

```

```
}  
}
```

A partir du deuxième clic le script annonce l'intervalle de temps en secondes et fraction de seconde écoulé depuis le dernier clic.

Une fonction plus fiable permet de connaître le nombre de secondes écoulées depuis le 1^{er} janvier 1970 à 0h00 en temps universel (UTC) : **llGetUnixTime**. Cette référence peut être intéressante dans des scripts pour évaluer des durées sans les inconvénients cités ci-dessus, mais la fonction renvoie une valeur de type **integer**, la précision est donc moins grande. Voici l'équivalent du script ci-dessus avec une précision d'une seconde :

```
integer flag;  
integer time;  
default {  
    touch_start(integer total_number) {  
        if(llDetectedKey(0) == llGetOwner()) {  
            if(flag)  
                llOwnerSay("Vous avez déjà cliqué il y a " + (string)(llGetUnixTime() - time) + " secondes");  
            else  
                flag = TRUE;  
            time = llGetUnixTime();  
        }  
    }  
}
```

13.4 La dilatation temporelle

On expérimente souvent dans **SL** le phénomène du lag. Si celui-ci est agaçant pour l'utilisateur il peut s'avérer désastreux pour certains scripts qui réclament une grande fiabilité temporelle. Lorsqu'un serveur est surchargé il provoque un ralentissement des scripts, c'est la dilatation temporelle. Autrement dit le temps devient plus long et une seconde n'est plus vraiment une seconde mais quelque chose d'un peu plus long. Il est possible de connaître la valeur de cette dilatation avec la fonction **llGetRegionTimeDilation** qui renvoie une valeur de type **float** comprise entre 0 (dilatation maximale) et 1 (pas de dilatation). Par exemple avec une dilatation de 0,5 une seconde de **SL** vaut en fait deux secondes de **RL**. Donc si vous avez lancé un **timer** ou si vous prenez comme référence la durée d'écoulement du script cette information peut vous être utile. Vous pouvez décider de stopper un script si la dilatation est trop importante ou alors compenser artificiellement ses effets.

13.5 Délai entre événements

Il peut vous arriver parfois de temporiser l'arrivée des événements. Le délai normal entre événements est de 0,05 secondes (sauf pour les événements **touch** pour lesquels le délai est de 0,1s). La fonction **llMinEventDelay** permet d'imposer une durée supérieure à cette valeur de référence entre deux événements successifs. Cette possibilité peut se montrer utile dans des situations aux événements fréquents. Mais le risque est de perdre des événements alors soyez prudents avec cette possibilité, d'autant que la manière dont cette fonction opère n'est pas des plus limpides.

14. Le travail de scripteur

Scripter peut être un jeu ou un travail, ou les deux à la fois. J'ai pensé judicieux d'ajouter ce chapitre pour montrer un peu la démarche du scripteur, du cahier des charges jusqu'à la livraison du script final. L'ensemble de ce chapitre correspond à mon expérience personnelle et je ne prétend ni à l'universalité ni à l'exhaustivité. Je souhaite juste qu'il aide le scripteur débutant à acquérir une méthodologie.

14.1 Le cahier des charges

Le cahier des charges est la première étape indispensable à la charge du client. L'expérience montre que celui-ci ne connaît jamais réellement ses besoins, sauf cas particulier. Il faut souvent passer un long moment avec lui pour définir les besoins réels. Bien souvent le client compte sur le scripteur pour finaliser sa demande. Il ne faut pas négliger cette phase d'élaboration précise du cahier des charges pour évaluer le volume de code à générer et sa complexité, pour deux raisons essentielles:

- déterminer le coût de la prestation,
- éviter la réalisation de versions multiples de scripts en se rendant compte à chaque fois qu'ils ne satisfont pas le client.

Les exemples valant toujours beaucoup plus que les longs discours je vous propose un exemple de réalisation que nous allons développer tout au long de ce chapitre. Pour qu'en plus cette réalisation ait une réelle utilité je vais me concentrer sur un thème récurrent, celui des portes. Supposons que mon client désire une application relativement riche en fonctionnalités pour ne pas être pris au dépourvu. Le bâtiment qu'il veut équiper comporte des portes à plusieurs prims, et des portes couplées par deux. Certaines de ces portes doivent fonctionner en translation et d'autres en rotation. Elles doivent fonctionner en local par un clic. Voici à quoi peut ressembler le cahier des charges :

Réalisation d'un ou plusieurs scripts pour commande de portes:

- Au niveau du build les portes sont constituées de plusieurs prims liés (la porte n'est pas liée au bâtiment).
- La largeur des portes peut être sur l'axe X ou l'axe Y (le script doit déterminer automatiquement ce paramètre).
- La manoeuvre des portes peut être en translation ou en rotation (dans ce cas sans prim axial), paramétrable par NoteCard. Le mouvement doit être opérationnel quelle que soit l'orientation du bâtiment.
- Les portes sont à droite ou à gauche.
- Les portes fonctionnent de façon isolée ou par groupe.
- Les portes sont commandées par un clic de souris (dans le cas de portes couplées les deux portes s'ouvrent), la fermeture intervenant au bout d'un délai paramétrable par NoteCard. Le client se réserve la possibilité d'une commande centralisée ultérieure.

A partir de ce cahier des charges précis il devient possible d'évaluer le temps de travail nécessaire et donc le prix de la prestation en fonction du coût horaire que vous estimez judicieux.

A partir de ce cahier des charges vous allez également commencer à concevoir l'organisation générale de votre code :

- un seul script ?
- combien états ?
- quels éléments de communication ?
- ...

14.2 L'organisation du code

La première question à se poser est : combien de scripts et placés où ? L'expérience montre que la commande d'une porte nécessite un script dans le root. Le volume de code nécessaire ne justifie a priori pas de scinder le code en plusieurs scripts. On pourrait envisager plusieurs scripts selon les différents cas demandés : translation, rotation, etc... Mais il est plus judicieux d'avoir un seul script qui couvre tous les cas de figure, même si son écriture doit se révéler plus laborieuse.

Il faut également déterminer les informations de paramétrage qui doivent figurer dans une NoteCard :

- type de manoeuvre de la porte : translation ou rotation,
- canal de communication éventuel entre deux portes fonctionnellement liées (porte non liée au bâtiment), ce canal sera le même pour toutes les portes à cause de la possibilité future d'une commande centralisée, ce qui implique la définition de groupes pour les portes liées fonctionnellement, cette méthode permet en outre d'homogénéiser les NoteCards pour le bâtiment,
- durée d'ouverture d'une porte,
- position de la porte,
- sens de manoeuvre de la porte.

Au niveau du code nous aurons ces éléments :

- lecture de la NoteCard et mémorisation des paramètres,
- paramétrage des translation et rotation
- commande d'ouverture de la porte
- commande de fermeture de la porte
- écoute du Chat et envoi du message sur le Chat pour les portes liées

Les cripts LSL étant par nature des machine d'état on peut s'interroger sur la pertinence de l'utilisation d'état dans ce cas. De toute évidence nous avons trois états :

- initialisation
- porte fermée
- porte ouverte

A partir de là il est judicieux d'établir un synoptique de fonctionnement (voir annexe 1). Bien souvent ce synoptique réside uniquement dans la tête du scripteur et n'est pas formalisé. Il permet toutefois de clarifier bien souvent la problématique et de trouver des solutions optimisées.

14.3 La NoteCard

Ce script implique la présence d'une NoteCard nommée "Config" :

Canal du Chat

5827

Type de porte : TRANSLATION or ROTATION

TRANSLATION

Position de la porte : LEFT or RIGHT

LEFT

Orientation : axe direction + or -

-

Temps d'ouverture en secondes

15

Groupe : 0 pour une porte libre, 1...n pour les groupes

0

14.4 L'écriture du code

A partir du synoptique on peut passer à l'écriture du code.

14.4.1 Les variables utiles

Il faut dénombrer, définir et nommer les variables qui vont servir dans ce script. Il faut déjà une variable par donnée récupérée dans la NoteCard., plus quelques autres variables :

```
// -----  
//          Variables de travail  
// -----  
integer comCanal;           // Canal du Chat  
integer notecardLine;       // Ligne de la note  
integer ecouteChat;         // Ecoute du Chat  
integer doorAxeX;           // Sens de largeur de la porte  
integer doorDirection;     // Sens de manoeuvre de la porte  
string doorGroup;           // Groupe de la porte  
string doorType;            // Type de porte  
string doorPosition;        // Position de la porte  
vector translation;         // Translation de la porte  
rotation rot90;             // Rotation de 90°  
rotation rotDepart;         // Rotation porte fermée  
float delai;                // Délai de maintien d'ouverture
```

14.4.2 Initialisation

Nous avons vu que l'initialisation consiste en la lecture de la note et au paramétrage de la porte :

```
// -----  
//          Initialisation  
// -----  
init() {  
    if (llGetInventoryType("Config") != INVENTORY_NONE) {  
        notecardLine = 1;  
        llGetNotecardLine("Config", notecardLine); }  
    else
```



```

    llOwnerSay("NoteCard Config not present !!!");}

// -----
//      Paramétrage
// -----
parametrage() {
    vector dimensions = llGetScale();
    doorAxeX = dimensions.x > dimensions.y;
    if (doorType == "TRANSLATION") {
        if (doorAxeX) translation = <dimensions.x, .0, .0>;
        else translation = <.0, dimensions.y, .0>;
        if (doorPosition != "LEFT") translation = -translation;}
    else {
        rot90 = llEuler2Rot(<.0, .0, PI_BY_TWO>);
        float dim = dimensions.x / 2.0;
        if (doorAxeX) {
            if (doorPosition == "LEFT") translation = <dim, -dim, .0>;
            else translation = <-dim, -dim, .0>;}
        else {
            if (doorPosition == "LEFT") translation = <-dim, -dim, .0>;
            else translation = <-dim, dim, .0>;}
    }
    if (doorDirection) translation = -translation;}

// -----
//      Etat initial
// -----
default
{
    state_entry() {init();}

    dataserver(key id, string data){
        if (data != EOF) {
            if (notecardLine == 1) comCanal = (integer)data;
            else if (notecardLine == 3) doorType = data;
            else if (notecardLine == 5) doorPosition = data;
            else if (notecardLine == 7) doorDirection = data == "+";
            else if (notecardLine == 9) delai = (float)data;
            else if (notecardLine == 11) {
                doorGroup = data;
                parametrage();
                state doorClose;}
            llGetNotecardLine("Config", ++notecardLine);}
        }
    }
}

```

Ce code appelle quelques commentaires. En ce qui concerne la lecture de la note et le fonctionnement de l'événement **dataserver** reportez-vous au chapitre 8.13.1. Par contre la fonction **parametrage** mérite d'être un peu explicitée. Les deux premières lignes :

```
vector dimensions = llGetScale();
```

```
doorAxeX = dimensions.x > dimensions.y;
```

permettent de déterminer si le builder a constitué la largeur de sa porte selon l'axe X ou Y, en partant du principe qu'une porte est plus large que profonde (pour les portes blindées de coffre fort il faudra vous adapter !). On renseigne la variable **doorAxeX** en conséquence. Cette variable est **TRUE** si la porte est constituée selon l'axe X, **FALSE** dans le cas contraire.

Une instruction de sélection **if** permet ensuite de séparer le traitement selon que la porte est du type **TRANSLATION** ou du type **ROTATION**. Cela nous est indiqué par la variable **doorType** :

```
if (doorType == "TRANSLATION") {
```

Le traitement du type **TRANSLATION** est le plus simple :

```
if (doorAxeX)      translation = <dimensions.x, .0, .0>;  
else translation = <.0, dimensions.y, .0>;  
if (doorPosition != "LEFT") translation = -translation;
```

Selon la valeur de la variable **doorAxeX** que nous avons vue ci-dessus on définit la translation sur la dimension **x** ou **y**. Il suffit ensuite de tenir compte de la position de la porte donnée par la variable **doorPosition** pour définir le sens de la translation.

Le traitement du type **ROTATION** est un peu plus complexe :

```
rot90 = II Euler2Rot(<.0, .0, PI_BY_TWO>);  
float dim;  
if (doorAxeX) {  
    dim = dimensions.x / 2.0;  
    if (doorPosition == "LEFT") translation = <dim, -dim, .0>;  
    else translation = <-dim, -dim, .0>;  
}  
else {  
    dim = dimensions.y / 2.0;  
    if (doorPosition == "LEFT") translation = <-dim, -dim, .0>;  
    else translation = <-dim, dim, .0>;  
}
```

On commence par définir une variable **rot90** donnant une rotation de 90 degrés sur l'axe **Z** qui servira pour la commande de la porte. Selon les valeurs des variables **doorAxeX** et **doorPosition** on ajuste la valeur de la translation. Cette translation étant toujours d'une moitié de la largeur de la porte sur les axes **X** et **Y**, seul le sens change selon les cas.

On finit par ajuster la valeur de la translation selon le sens d'ouverture de la porte qui nous est donné par la variable **doorDirection** :

```
if(doorDirection) translation = -translation;
```

14.4.3 Etat porte fermée

L'initialisation étant terminée on est envoyé sur l'état porte fermée :

```
// -----
//          Etat porte fermée
// -----

state doorClose
{
    changed(integer change) {if(change & CHANGED_INVENTORY) llResetScript();}

    state_entry(){if(doorGroup != "0") ecouteChat = llListen(comCanal, "", NULL_KEY, "");}

    touch_start(integer total_number) {
        if(doorGroup) llSay(comCanal, doorGroup);
        ouvrePorte();
        state doorOpen;}

    listen(integer channel, string name, key id, string message) {
        if (channel == comCanal && message == doorGroup) {
            ouvrePorte();
            state doorOpen;}
    }

    state_exit(){if(doorGroup != "0") llListenRemove(ecouteChat);}
}
```

C'est l'état de repos de la porte. En cas de changement dans l'inventaire (changement de la note) on réinitialise le script :

```
changed(integer change) {if(change & CHANGED_INVENTORY) llResetScript();}
```

Dans l'événement d'entrée dans l'état **state_entry** la ligne :

```
if (doorGroup != "0") ecoute = llListen(comCanal, "", NULL_KEY, "");
```

est destinée à définir une écoute du Chat dans le cas où la porte appartient à un groupe, ce que l'on teste avec la valeur de la variable **doorGroup** qui contient « 0 » pour une porte libre et un numéro de groupe pour les portes liées fonctionnellement.

Le script dans cet état attend deux événements : **touch_start** et **listen**. Dans les deux cas on fait appel à la fonction **ouvrirPorte** mais dans le cas de la réception du message par le Chat on vérifie que la porte est bien concernée par la commande en testant le numéro du groupe, dans le cas d'un clic sur la porte on transmet le message sur le Chat si la porte appartient à un groupe. Voyons la fonction d'ouverture de la porte :

```
// -----  
//          Ouverture de la porte  
// -----  
ouvrirPorte(){  
    rotation r = IIGetRot();  
    rotDepart = IIGetRot();  
    if (doorType != "TRANSLATION") {  
        if (doorPosition == "LEFT") r = rot90 * rotDepart;  
        else r = rotDepart / rot90;}  
    IISetPrimitiveParams([PRIM_ROTATION, r, PRIM_POSITION, IIGetPos() + translation * rotDepart]);  
}
```

On commence par initialiser deux variables **r** et **rotDepart** avec la valeur actuelle de la rotation. La variable **r** est une variable locale de la fonction alors que **rotDepart** est une variable globale dont la valeur nous servira pour la fermeture de la porte. On teste ensuite si la porte est de type **TRANSLATION**. Si c'est le cas on passe directement à la fonction **IISetPrimitiveParams** qui applique la translation en tenant compte de la rotation globale de la porte. Si on a affaire à une porte de type **ROTATION** il faut s'occuper de la rotation qui dépend de la position de la porte. Si vous avez des doutes sur vos connaissances concernant les rotations et les translations je vous conseille une relecture des chapitres 9.1 et 9.3. Une fois la porte commandée à l'ouverture on bascule sur l'état porte ouverte. Mais on passe auparavant par le traitement de l'événement de sortie de l'état **state_exit** dans lequel il nous faut arrêter éventuellement l'écoute du Chat :

```
state_exit(){if(doorGroup != "0") IIListenRemove(ecoute);}
```

14.4.4 Etat porte ouverte

```
// -----  
//          Etat porte ouverte  
// -----  
state doorOpen  
{  
    state_entry() {IISetTimerEvent(delai);}  
    timer(){
```

```
fermePorte();  
state doorClose;}  
state_exit() {llSetTimerEvent(.0);}  
}
```

Cet état est très simple, on se contente de mettre en marche une temporisation à l'entrée dans l'état. Au terme de cette temporisation l'événement **timer** se déclenche, on commande la porte à la fermeture avec la fonction **fermePorte** et on revient à l'état porte fermée. On arrête le **timer** à la sortie de l'état dans l'événement **state_exit**. Voyons la fonction **fermePorte** :

```
// -----  
//           Fermeture de la porte  
// -----  
fermePorte(){  
    rotation r = llGetRot();  
    if (doorType != "TRANSLATION") {  
        if (doorPosition == "LEFT") r = r / rot90;  
        else r = rot90 * r;}  
    llSetPrimitiveParams([PRIM_ROTATION, r, PRIM_POSITION, llGetPos() - translation *  
rotDepart]);  
}
```

Cette fonction fait écho à celle de l'ouverture de la porte avec inversion des translations et rotations.

14.5 Test et débogage du code

Il est rare qu'un code fonctionne du premier coup et une phase de tests est nécessaire pour la mise au point. Les outils fournis par **SL** ne sont pas très pratiques et la traque des erreurs est parfois fastidieuse. Il faut parsemer le code de fonctions (**llOwnerSay**) vous indiquant l'état de certaines variables pour détecter les erreurs. Si vous avez correctement commenté votre code cela peut grandement vous aider. Le fait de séparer votre code en différentes fonctions permet aussi plus facilement de localiser les problèmes. Les erreurs de syntaxe sont faciles à repérer parce que le compilateur vous les signale, les erreurs de logique sont de votre seul ressort et risquent de vous occuper quelques temps. Une des causes les plus fréquentes est la confusion entre “=” et “==” ou entre “&” et “&&”. Parfois aussi votre script devient si volumineux que vous dépassez la quantité de mémoire (16K en version classique et 64K avec Mono) qui lui est allouée. Il n'y a pas vraiment de méthodologie systématique pour cette phase, seule l'expérience permet de traverser cette étape en douceur.

Pour la pérenité de votre code pensez à lui attribuer un numéro de version et une date.

INDEX

A

ALL_SIDES · 91, 92, 93, 94, 95, 96, 100

ANIM_ON · 95, 96

C

CHANGED_LINK · 12, 66, 68, 69, 102

CHANGED_REGION · 12, 66, 73

CONTROL_FWD · 14, 67, 68

CONTROL_RIGHT · 14, 68

D

DEG_TO_RAD · 25, 26, 27, 28, 39, 42, 84, 85, 88, 90

do · 5

E

else · 5

F

for · 5

I

if · 5, 6, 8, 12, 14, 30, 35, 36, 38, 48, 49, 50, 66, 67, 68, 69, 71, 74, 92, 97, 98, 100, 102, 114, 117, 119, 120, 137, 138, 139, 140, 141, 142

INVENTORY_ALL · 98

INVENTORY_ANIMATION · 98

INVENTORY_BODYPART · 98

INVENTORY_CLOTHING · 98, 99

INVENTORY_GESTURE · 98

INVENTORY_LANDMARK · 98

INVENTORY_NONE · 98, 137

INVENTORY_NOTECARD · 98

INVENTORY_OBJECT · 98

INVENTORY_SCRIPT · 98

INVENTORY_SOUND · 98

INVENTORY_TEXTURE · 98

J

jump · 6

L

LAND_LARGE_BRUSH · 124, 125

LAND_LEVEL · 124

LAND_LOWER · 124

LAND_MEDIUM_BRUSH · 124

LAND_NOISE · 124

LAND_RAISE · 125

LAND_REVERT · 124

LAND_SMALL_BRUSH · 124

LAND_SMOOTH · 124

LINK_ALL_CHILDREN · 101

LINK_ALL_OTHER · 101

link_message · 35, 36, 76, 101

LINK_ROOT · 101

LINK_SET · 101, 124

LINK_THIS · 35, 36, 101

LIST_STAT_GEOMETRIC_MEAN · 37

LIST_STAT_MAX · 37

LIST_STAT_MEAN · 37

LIST_STAT_MEDIAN · 37

LIST_STAT_MIN · 37

LIST_STAT_NUM_COUNT · 37

LIST_STAT_RANGE · 37

LIST_STAT_STD_DEV · 37

LIST_STAT_SUM · 37

listen · 60, 112, 113, 114, 140, 141

llAddToLandBanList · 129, 130

llAddToLandPassList · 129

llAdjustSoundVolume · 107, 109

llAngleBetween · 27, 28

llAttachToAvatar · 67, 117, 118

llAxes2Rot · 27

llBreakAllLinks · 67, 102

llBreakLink · 67, 102

llClearCameraParams · 123

llCollisionSound · 75

llCreateLink · 67, 101

llDeleteSubList · 33

llDeleteSubString · 20

llDetachFromAvatar · 117, 118

llDetectedGrab · 62, 75

llDetectedGroup · 55, 62, 75, 126

llDetectedKey · 16, 49, 55, 62, 67, 69, 74, 75, 97, 102, 110, 112, 113, 114, 117, 119, 120, 129, 130

llDetectedLinkNumber · 75, 102

llDetectedName · 55, 61, 62, 75, 112

llDetectedOwner · 62, 75

llDetectedPos · 55, 62, 75

llDetectedRot · 55, 62, 75

llDetectedTouchBinormal · 56, 57

llDetectedTouchFace · 56

llDetectedTouchNormal · 56

llDetectedTouchST · 57

llDetectedTouchUV · 58

llDetectedType · 62, 75
llDetectedVel · 55, 62, 76
llDialog · 112, 113, 114
llDumpList2String · 34
llEscapeURL · 23
llEuler2Rot · 25, 26, 27, 28, 30, 39, 84, 85, 86, 88, 89, 90, 138, 139
llForceMouselook · 122
llGetAgentInfo · 120
llGetAgentLanguage · 122
llGetAgentSize · 119
llGetAndResetTime · 133
llGetAnimationList · 119
llGetAttached · 117, 118
llGetCameraPos · 123
llGetCameraRot · 123
llGetGMTclock · 131
llGetInventoryKey · 16, 98
llGetInventoryNumber · 99
llGetInventoryPermMask · 98
llGetInventoryType · 98, 99, 137
llGetKey · 16, 17, 30
llGetLinkKey · 103
llGetLinkName · 103
llGetLinkNumber · 101
llGetListEntryType · 30, 39
llGetListLength · 29, 119
llGetLocalPos · 81, 82, 89
llGetNumberOfPrims · 101
llGetOwner · 16, 17, 49, 69, 70, 75
llGetOwnerKey · 16
llGetParcelDetails · 127
llGetParcelMaxPrims · 127
llGetParcelPrimCount · 127
llGetParcelPrimOwners · 128
llGetPos · 23, 24, 27, 70, 71, 72, 79, 80, 81, 86, 141, 142
llGetPrimitiveParams · 94
llGetRegionFlags · 125, 126
llGetRegionTimeDilation · 134
llGetRootPosition · 83
llGetRootRotation · 89, 90
llGetScale · 80, 86, 89, 90, 138
llGetStatus · 92
llGetSubString · 21, 52, 53
llGetSunDirection · 99, 100
llGetTexture · 93
llGetTextureRot · 94
llGetTextureSacle · 94
llGetTime · 133
llGetUnixTime · 134

llGetWallclock · 131
llGiveInventory · 97
llGiveInventoryList · 97
llGround · 124
llGroundNormal · 125
llGroundSlope · 125
llInsertString · 20
llInstantMessage · 70, 71, 112, 118
llKey2Name · 16, 70, 74
llList2List · 31
llList2ListStriped · 37
llList2Rot · 27
llList2String · 29, 119
llListen · 60, 113, 114, 140
llListenRemove · 60, 113, 114, 140, 141
llListFindList · 31, 32
llListInsertList · 32, 33
llListRandomize · 33, 34, 36
llListReplaceList · 33
llListSort · 34, 36
llListStatistics · 37
llLookAt · 62, 86
llLoopSound · 109
llLoopSoundMaster · 109
llLoopSoundSlave · 109
llMessageLinked · 35, 36, 76, 101
llMinEventDelay · 134
llModifyLand · 124
llOffsetTexture · 94
llOwnerSay · 112, 138, 142
llParcelMediaCommandList · 110
llParcelMediaQuery · 111
llParseString2List · 35, 36, 76, 133
llParseStringKeepNulls · 36, 38
llPlaySound · 107, 108
llPreLoadSound · 107
llPushObject · 125
llRegionSay · 60, 112
llReleaseCamera · 123
llRemoveFromLandBanList · 130
llRemoveFromLandPassList · 129
llRequestPermissions · 67, 68, 69, 102, 117, 119
llResetTime · 56, 133
llRot2Angle · 27
llRot2Axis · 27
llRot2Euler · 25, 26, 27, 28
llRot2Fwd · 27, 28, 85
llRot2Left · 27, 28, 85
llRot2Up · 27, 28, 85
llRotateTexture · 94
llRotBetween · 27

llRotLookAt · 87, 88
 llSay · 1, 2, 4, 5, 6, 7, 44, 47, 48, 49, 50, 51, 55, 60,
 61, 66, 72, 73, 84, 112, 140
 llScaleTexture · 93, 94
 llSetAlpha · 91, 93
 llSetCameraAtOffset · 122
 llSetCameraEyeOffset · 122
 llSetCameraParams · 68, 123
 llSetClickAction · 103
 llSetLinkAlpha · 102
 llSetLinkColor · 102
 llSetLinkPrimitiveParams · 83, 90, 91, 92, 96, 102,
 103
 llSetLinkTexture · 96, 102
 llSetParcelMusicURL · 110
 llSetPos · 23, 24, 27, 72, 73, 79, 81, 82
 llSetPrimitiveParams · 56, 80, 83, 85, 86, 89, 90,
 91, 92, 93, 94, 100, 103, 104, 141, 142
 llSetRot · 25, 26, 27, 83, 84, 85
 llSetScale · 90
 llSetSoundQueueing · 109
 llSetStatus · 71, 72, 92
 llSetTexture · 93
 llSetTextureAnim · 94, 95
 llSetTimerEvent · 133
 llShout · 60, 112
 llStringLength · 19
 llStartAnimation · 67, 69, 119
 llStopAnimation · 69, 119
 llStopLookAt · 86
 llStopSound · 107
 llStringTrim · 22
 llSubStringIndex · 21, 22, 38
 llTargetOmega · 27, 84
 llToLower · 22
 llToUpper · 22
 llTriggerSound · 107, 108
 llTriggerSoundLimited · 108, 109
 llUnescapeURL · 23
 llVecDist · 24, 79
 llVecMag · 24
 llVecNorm · 24
 llWhisper · 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19,
 20, 21, 22, 24, 28, 29, 30, 31, 32, 33, 34, 35, 36,
 39, 51, 52, 53, 60, 67, 70, 72, 74, 75, 76, 98, 99,
 112, 113, 114, 117, 119, 120
 LOOP · 95
 M
 MASK_BASE · 98, 99
 MASK_EVERYONE · 99
 MASK_GROUP · 99

MASK_NEXT · 99
 MASK_OWNER · 99
 O
 object_rez · 70
 P
 PARCEL_COUNT_GROUP · 128
 PARCEL_COUNT_OTHER · 128
 PARCEL_COUNT_OWNER · 128
 PARCEL_COUNT_SELECTED · 128
 PARCEL_COUNT_TEMP · 128
 PARCEL_COUNT_TOTAL · 127
 PARCEL_DETAILS_AREA · 127
 PARCEL_DETAILS_DESC · 127
 PARCEL_DETAILS_GROUP · 127
 PARCEL_DETAILS_NAME · 127
 PARCEL_DETAILS_OWNER · 127
 PARCEL_FLAG_ALLOW_ALL_OBJECT_ENTR
 Y · 126
 PARCEL_FLAG_ALLOW_CREATE_GROUP_O
 BJECTS · 126
 PARCEL_FLAG_ALLOW_CREATE_OBJECTS ·
 126
 PARCEL_FLAG_ALLOW_DAMAGE · 126
 PARCEL_FLAG_ALLOW_FLY · 126
 PARCEL_FLAG_ALLOW_GROUP_OBJECT_EN
 TRY · 126
 PARCEL_FLAG_ALLOW_GROUP_SCRIPTS ·
 126
 PARCEL_FLAG_ALLOW_LANDMARK · 126
 PARCEL_FLAG_ALLOW_SCRIPTS · 126
 PARCEL_FLAG_ALLOW_TERRAFORM · 126
 PARCEL_FLAG_LOCAL_SOUND_ONLY · 126
 PARCEL_FLAG_RESTRICT_PUSHOBJECT ·
 126
 PARCEL_FLAG_USE_ACCESS_GROUP · 126
 PARCEL_FLAG_USE_ACCESS_LIST · 126
 PARCEL_FLAG_USE_BAN_LIST · 126
 PARCEL_FLAG_USE_LAND_PASS_LIST · 126
 PARCEL_MEDIA_COMMAND_AGENT · 111
 PARCEL_MEDIA_COMMAND_AUTO_ALIGN ·
 111
 PARCEL_MEDIA_COMMAND_LOOP · 110
 PARCEL_MEDIA_COMMAND_PAUSE · 110
 PARCEL_MEDIA_COMMAND_PLAY · 110
 PARCEL_MEDIA_COMMAND_STOP · 110
 PARCEL_MEDIA_COMMAND_TEXTURE · 110
 PARCEL_MEDIA_COMMAND_TIME · 110
 PARCEL_MEDIA_COMMAND_UNLOAD · 111
 PARCEL_MEDIA_COMMAND_URL · 110
 PERM_ALL · 98, 99
 PERM_COPY · 98, 99

PERM_MODIFY · 98, 99
PERM_MOVE · 98, 99
PERM_TRANSFER · 98, 99
PERMISSION_ATTACH · 67, 117
PERMISSION_CHANGE_LINK · 101, 102
PERMISSION_CHANGE_LINKS · 67, 102
PERMISSION_CONTROL_CAMERA · 68, 123
PERMISSION_DEBIT · 67, 69
PERMISSION_TAKE_CONTROLS · 67, 68
PERMISSION_TRACK_CAMERA · 68, 123
PERMISSION_TRIGGER_ANIMATION · 67, 69, 119
PING_PONG · 95
PRIM_TEXTURE · 94, 96
PUBLIC_CHANNEL · 113
R
REGION_FLAG_ALLOW_DAMAGE · 125
REGION_FLAG_ALLOW_DIRECT_TELEPORT · 125
REGION_FLAG_BLOCK_FLY · 125
REGION_FLAG_BLOCK_TERRAFORM · 125
REGION_FLAG_DISABLE_COLLISIONS · 125
REGION_FLAG_DISABLE_PHYSICS · 125
REGION_FLAG_FIXED_SUN · 125
REGION_FLAG_RESTRICT_PUSHOBJECT · 125
REGION_FLAG_SANDBOX · 125
return · 6, 7, 38, 40, 53, 97, 98
REVERSE · 95
run_time_permissions · 67, 68, 69, 102, 117, 118, 119

S

SMOOTH · 95, 96
state_entry · 1, 2, 6, 7, 25, 26, 27, 28, 29, 40, 44, 48, 49, 50, 51, 52, 54, 55, 61, 62, 68, 70, 71, 75, 80, 84, 85, 86, 88, 89, 95, 99, 113, 114, 138, 140, 141
STATUS_BLOCK_GRAB · 92
STATUS_DIE_AT_EDGE · 92
STATUS_PHANTOM · 92
STATUS_PHYSICS · 71, 72, 92
STATUS_ROTATE_X · 92
STATUS_ROTATE_Y · 92
STATUS_ROTATE_Z · 92
STATUS_SANDBOX · 92
T
timer · 133
touch_start · 2, 7, 8, 9, 10, 11, 12, 13, 15, 16, 19, 20, 21, 22, 24, 28, 30, 31, 32, 33, 34, 36, 39, 49, 55, 60, 67, 69, 72, 73, 74, 79, 80, 81, 82, 83, 84, 85, 86, 88, 89, 90, 91, 92, 93, 97, 98, 102, 117, 119, 120, 140, 141
TYPE_FLOAT · 30, 38
TYPE_INTEGER · 30, 38
TYPE_INVALID · 38
TYPE_KEY · 30, 38
TYPE_ROTATION · 30, 38
TYPE_STRING · 30, 38
TYPE_VECTOR · 30, 38
W
WarpPos · 80
while · 5, 50, 51, 53, 79
Whisper · 8

Annexe 1

Synoptique de fonctionnement du script de la porte

