Rotations pour les nuls

par Bestmomo Lagan

Version 2.1
17/04/10

Avertissement *: Ce guide n'est pas un document officiel de Second Life. Il n'engage que la responsabilité de son auteur.*

Table des Matières

1. Introduction....4

2. Eléments de base....5

2.1 Radians et degrés....5
2.2 Constantes utiles....6
2.3 Quaternions....6

3. L'espace....7

3.1 Les axes....7
3.2 Coordonnées globales et locales....8

4. Rotations simples (root ou prim libre)....11

4.1 Vecteur rotation....11
4.2 Rotation globale....11
4.3 Combinaisons de rotations....12
4.4 Rotation locale....15
4.5 Translation indépendante de la rotation....18

5. Rotation continue....22

5.1 Rotation côté client....22
5.2 Rotation côté serveur....25
5.2.1 Avec llSetRot....25
5.2.2 Avec llRotLookAt....26
5.2.3 Avec llSetLinkPrimitiveParamsFast....26

6. Rotation excentrée....27

6.1 Une planète....27
6.2 Porte rotative....28

7. Objet lié....36

7.1 Quelques généralités....36
7.1.1 Translations....36
7.1.2 Rotations....37
7.2 Un métronome....39
7.3 Une planète....43

8. Généralisations....44

8.1 Rotations absolues....45
8.2 Rotations relatives....46
8.3 Rotations excentrées....47
8.3.1 Rotation simple....47
8.3.1.1 Axe global et centre global (prim libre, racine ou enfant)....50
8.3.1.2 Axe auto et centre global (prim libre, racine ou enfant)....51
8.3.1.3 Axe global et centre auto (prim libre, racine ou enfant)....52
8.3.1.4 Axe auto et centre auto (prim libre, racine ou enfant)....53
8.3.1.5 Axe local et centre global (prim enfant)....54
8.3.1.6 Axe local et centre local (prim enfant)....55
8.3.1.7 Axe local et centre auto (prim enfant)....56
8.3.1.8 Axe global et centre local (prim enfant)....57
8.3.1.9 Axe auto et centre local (prim enfant)....58

8.3.2 Rotations combinées.....58
8.4 Les attachements.....61

1. Introduction

Les rotations ! Bête noire du scripteur, le grand fléau qui décime nos rangs ! Rien qu'à l'évocation de ce mot la plupart des scripteurs commencent à sentir de la sueur perler sur leur front, leur cœur s'accélérer. Des journées de cauchemars à essayer des centaines de formules pour finir par tomber par hasard sur un truc qui fonctionne. Une bête infâme qui s'acharne sur nous : les quaternions, êtres étranges, impalpables, mystérieux, sournois, déconcertants, incompréhensibles. Une difficulté à visualiser les objets dans leurs trois dimensions sur nos écrans déserpérément plats.

Y aurai-il un remède pour soulager nos maux ? Ce petit guide est une tentative dans ce sens.

Quoi de neuf dans la version 2.0 ?

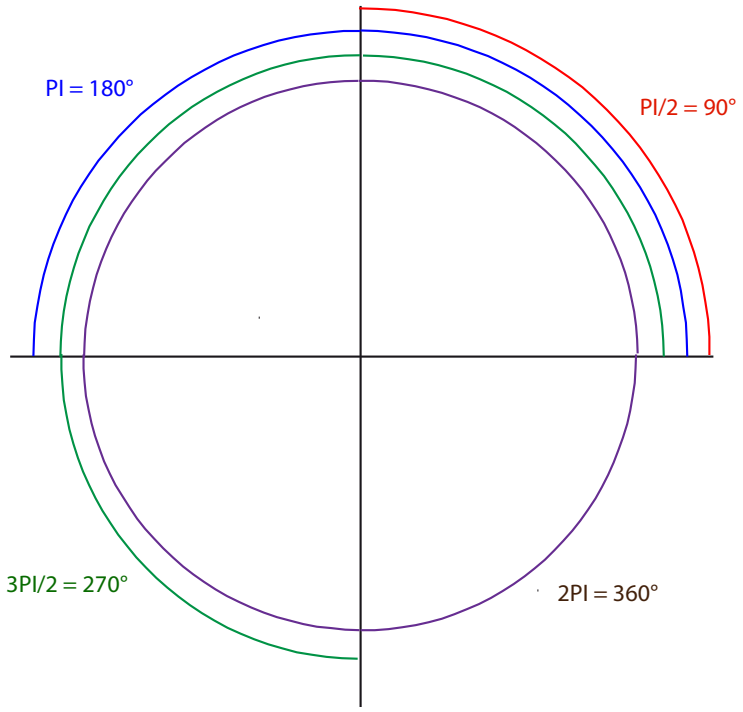
Cette nouvelle version amène de grandes nouveautés. Je me suis rendu compte au fil du temps et des scripts que j'ai eu à réaliser des insuffisances de ce document tel qu'il se présentait. J'ai donc remanié, complété et illustré pour rendre tout ça le plus accessible possible. Il n'en demeure pas moins malgré tout que ce domaine demande toujours un effort et ne se laisse pas maîtriser facilement. J'aurais aimé lire ce document à mes débuts lorsque j'ai découvert LSL, ça m'aurait évité bien des problèmes !

2. Eléments de base

Les fonctions de rotation attendent un quaternion, c'est quoi cette bête ? En fait c'est un formalisme mathématique qui facilite le traitement des rotations. Si la compréhension de ce formalisme est loin d'être évidente sa mise en oeuvre va nous faciliter la vie. En fin de compte vous n'avez absolument pas besoin de savoir ce que c'est exactement mais juste savoir les mettre en œuvre. D'autre part les valeurs des angles sont attendues en radians alors que vous êtes plus habitués aux degrés.

2.1 Radians et degrés

Pour commencer un petit rappel sur ce que sont les radians. Vous savez que la formule pour calculer le périmètre d'un cercle est **2 * PI * Rayon**. Si on admet que le rayon est égal à 1 la formule se simplifie en **2 * PI**. Autrement dit il faut **2 * PI** pour faire le tour complet du cercle, ce qui équivaut à 360°. Les radians ne sont rien d'autre que cette façon particulière de mesurer les angles. Voici une illustration avec les valeurs caractéristiques :



Pour convertir les radians en degrés et inversement utilisez les constantes suivantes :

Constante	Valeur
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

La stratégie à mettre en œuvre est la suivante : vous raisonnez en degrés comme vous en avez spontanément envie et ensuite vous utilisez les fonctions de conversion pour obtenir des radians et des quaternions.

2.2 Constantes utiles

Voici d'autres constantes utiles qui simplifieront l'écriture de vos scripts :

Constante	Valeur
PI	**3.14159265358979323846264338327 95** `(pi = 180°)`
TWO_PI	**6.28318530717958647692528676655 9** `(pi * 2 = 360°)`
PI_BY_TWO	**1.57079632679489661923132169163 98** `(pi / 2 = 90°)`

2.3 Quaternions

Un quaternion se représente mathématiquement par 4 nombres x, y, z et s. Mais nous n'auront pas à manipuler directement ces valeurs. Nous représenterons les rotations par un vecteur comportant trois valeurs correspondant respectivement à la rotation sur les axes X, Y et Z. Ces valeurs doivent être en radians. Il existe deux fonctions qui permettent de passer d'une représentation euclidienne en version quaternion et réciproquement :

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

3. L’espace

3.1 Les axes

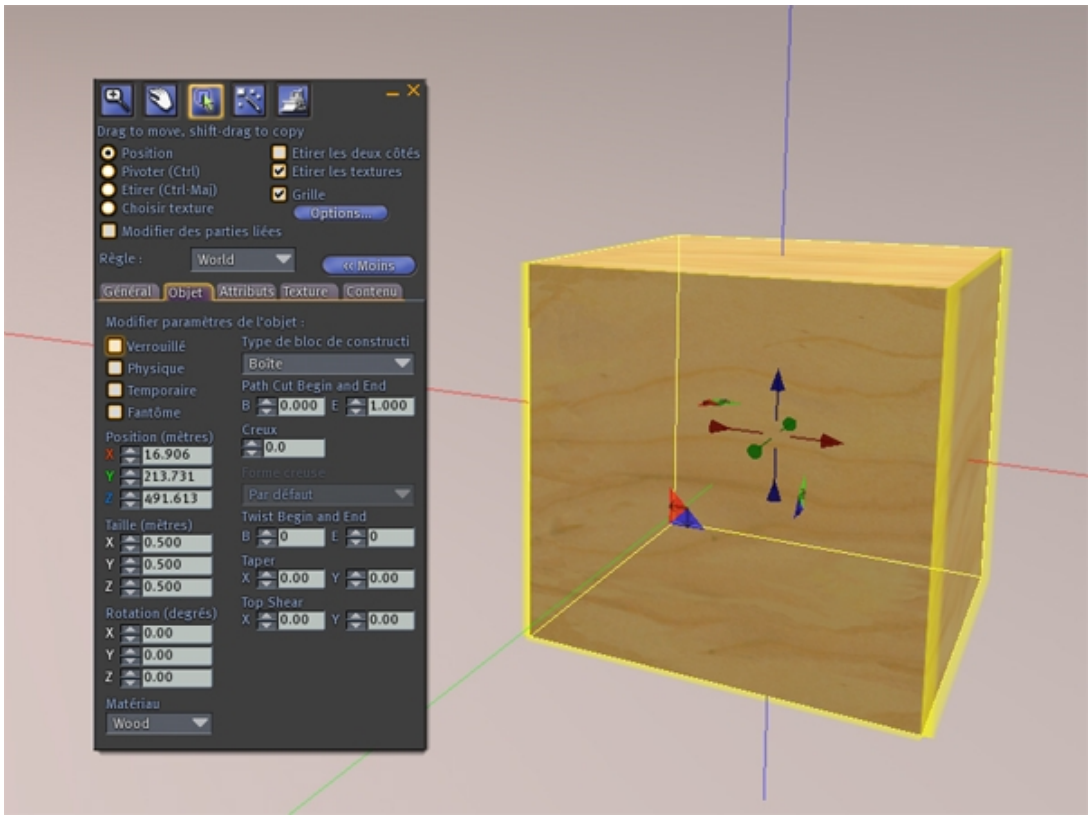
Notre espace est en trois dimensions, celui de SL aussi. Ces dimensions sont nommées classiquement :

- Largeur
- Profondeur
- Hauteur

Pour se repérer dans un tel espace nous avons besoin de repères. Par définition il est déterminé trois axes pour les trois dimensions :

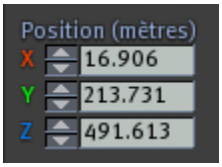
- Axe X : Largeur
- Axe Y : Profondeur
- Axe Z : Hauteur

Quand vous créez un objet vous voyez apparaître ces axes en mode build :



Remarquez l'orientation des axes. Pour un prim sans rotation particulière lorsque l'axe X est dirigé vers la droite, l'axe Y est dirigé vers l'arrière et l'axe Z vers le haut. Vous pouvez mémoriser ces directions en utilisant les trois premiers doigts de votre main gauche : pouce pour axe X, index pour axe Z et majeur pour axe Y.

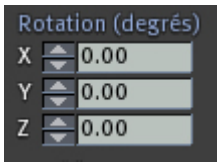
Dans la fenêtre de build vous voyez apparaître des renseignements concernant les trois axes. En premier la position de l'objet par rapport à la sim (simulateur) concernée.



Une sim est un espace virtuel de largeur et profondeur 256 mètres et de hauteur indéterminée. Les valeurs partent d'un coin de la sim. Par rapport à ce coin mon objet est à 16,906 mètres sur l'axe X et 213,731 mètres sur l'axe Y. Et il se situe en hauteur à 491,613 mètres (j'aime la tranquillité).

La taille de l'objet est aussi repérée par rapport à ces axes :

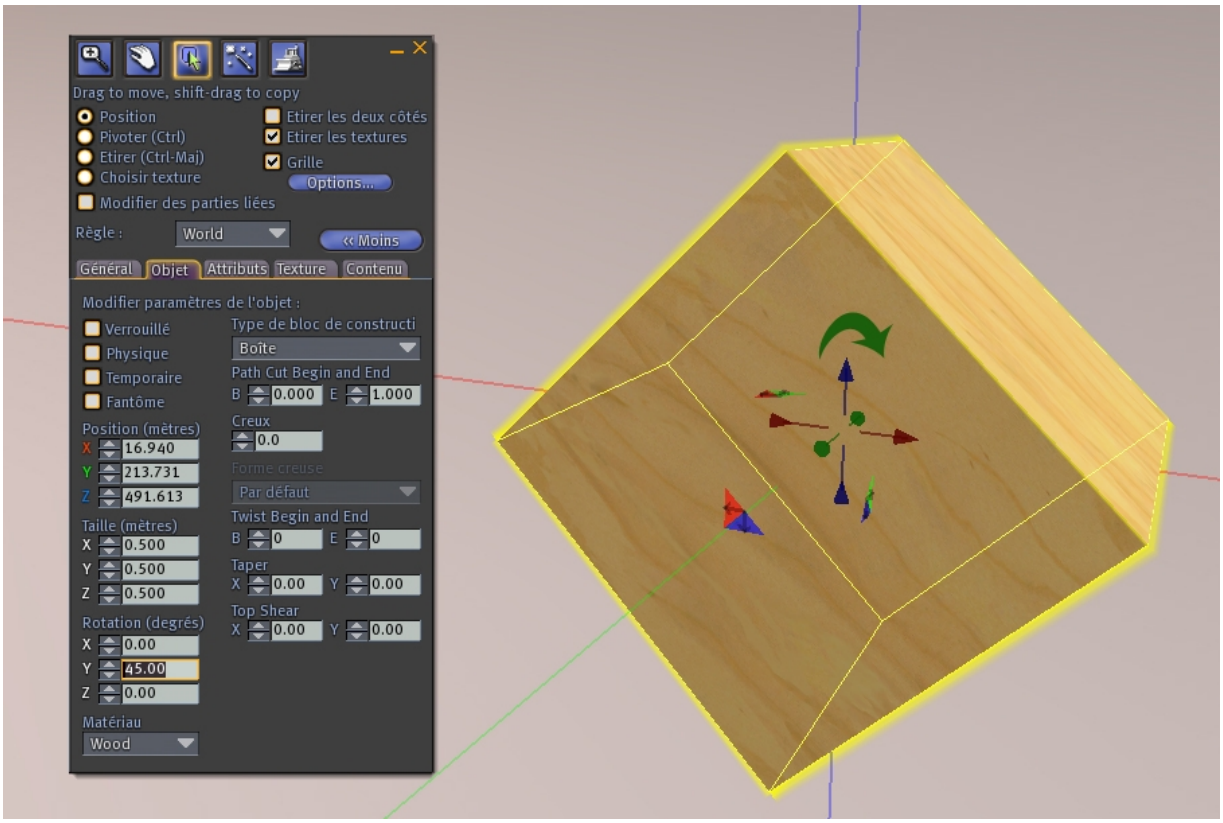
Mais ce qui va surtout nous intéresser ce sont les rotations. A la création de l'objet elles sont nulles sur les trois axes :



Pour le moment tout va pour le mieux dans le meilleur des mondes mais nous allons faire bouger tout ça !

3.2 Coordonnées globales et locales

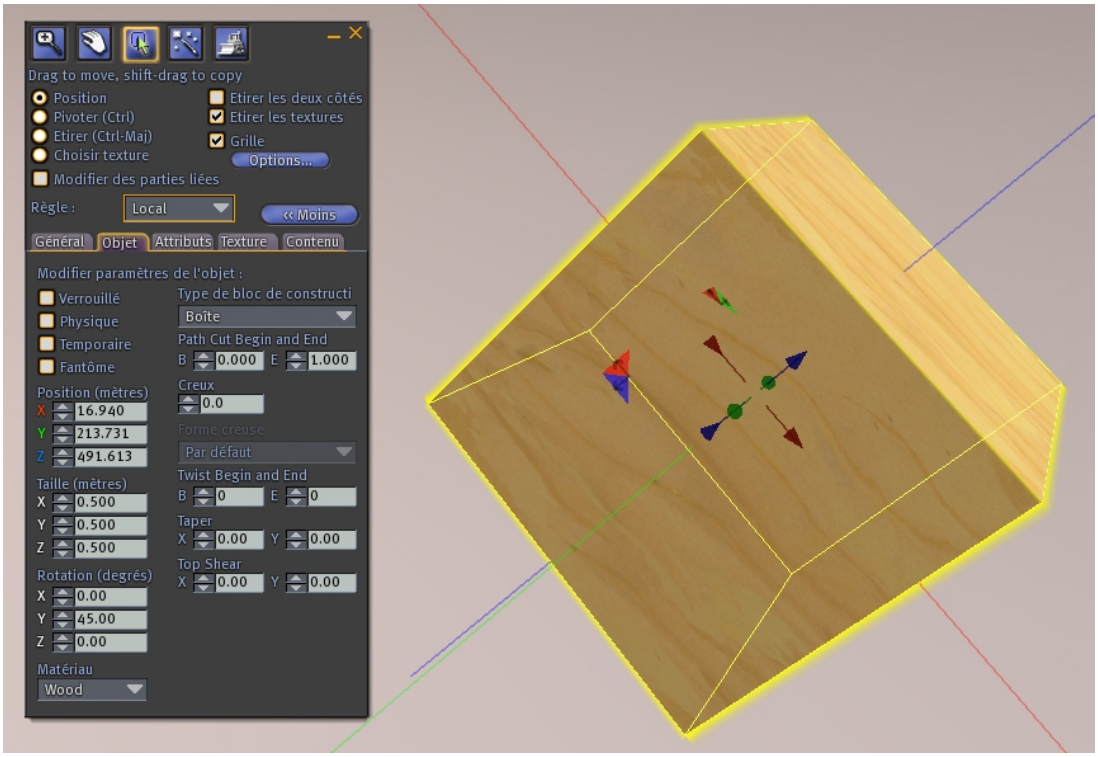
Les coordonnées dont j'ai parlé ci-dessus sont les coordonnées globales, c'est-à-dire que le repère est la sim et ses axes. A la création d'un objet celui-ci est parfaitement aligné sur ces axes globaux. Mais que se passe-t-il si on lui fait effectuer une rotation ? Il est alors évident que les axes locaux de l'objet ne sont plus alignés sur les axes globaux. Donnons une rotation positive de 45° sur l'axe Y à notre objet :



Vous voyez facilement que l'objet à tourné autour de l'axe Y, cet axe restant évidemment fixe. La rotation étant positive le mouvement s'est fait dans le sens horaire au niveau de l'illustration.

Pour connaître le sens de rotation imaginez un tire-bouchon sur l'axe concerné, dans le cas d'une rotation positive le tire-bouchon avance dans la direction de l'axe :

Observons les coordonnées locales pour ce même objet :



Pour mémoire vous effectuez ce changement de repère avec le **combobox** situé vers le haut. Nous constatons maintenant que les axes locaux de l'objet ont bougé par rapport aux axes globaux. Les axes X et Z ont subit une rotation de 45° par rapport aux repères globaux.

Il est important de conserver toujours cela à l'esprit lorsque vous traitez des rotations.

Vous suivez toujours ? Parfait !

4. Rotations simples (root ou prim libre)

4.1 Vecteur rotation

Nous avons vu qu'une rotation peut s'exprimer par un angle spécifique sur chacun des trois axes X, Y et Z. Pour représenter ces valeurs on peut utiliser un type **vector** dans LSL du style <angle_x, angle_y, angle_z>. Chacune des valeurs angulaires est un **float**. Par exemple la rotation positive de 45° (PI / 4) que nous avons donnée au prim au chapitre précédent peut se représenter <0.0, 45.0, 0.0>. Pour mémoire il est préférable pour les valeurs d'un vecteur d'adopter une représentation de type float en représentant le point de séparation. En d'autres termes si vous adoptez le vecteur <0, 45, 0> au lieu de <0.0, 45.0, 0.0> votre script va fonctionner mais il sera plus lent.

4.2 Rotation globale

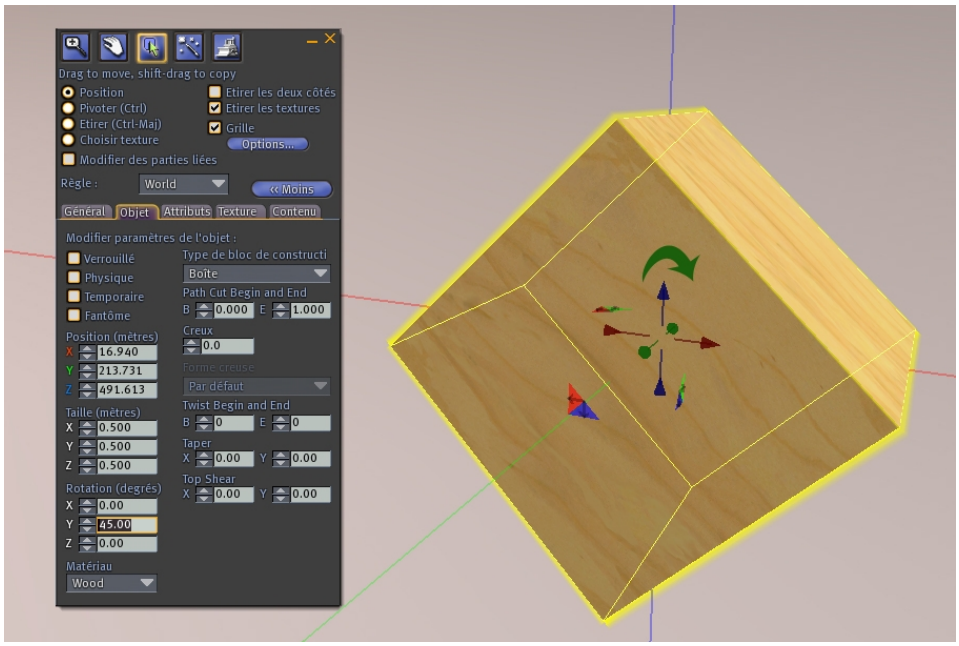
La rotation de 45° que nous avons effectuée au chapitre précédent est une rotation absolue, c'est-à-dire qu'elle fait référence aux coordonnées globales. Tout ce que nous faisons en mode build nous pouvons le faire avec un script. La première fonction à connaître est **llSetRotation** qui se contente de donner une rotation globale à l'objet qui contient le script. Cette fonction attend un quaternion. Voici le raisonnement à tenir :

1. Je veux une rotation positive de 45° sur l'axe Y
2. Je convertis cette valeur en radians : DEG_TO_RAD * 45.0
3. Je crée un vecteur représentant cette rotation sur l'axe Y : <.0, DEG_TO_RAD * 45.0, .0>
4. Je transforme ce vecteur en quaternion : llEuler2Rot(<.0, DEG_TO_RAD * 45.0, .0>)

A partir de là je n'ai plus qu'à utiliser la fonction **llSetRotation** pour obtenir ma rotation globale :

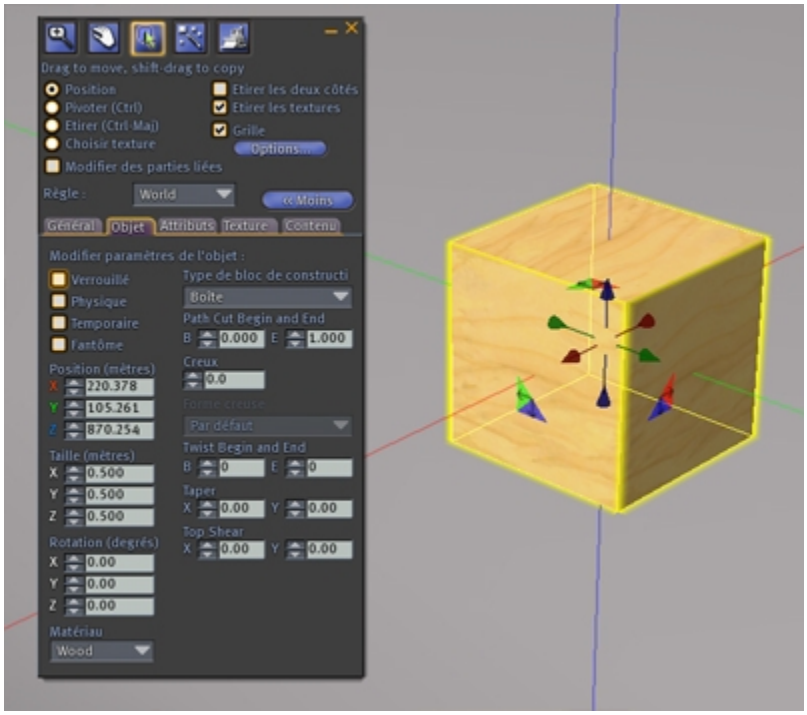
```
default{
	touch_start(integer total_number) {
		llSetRot(llEuler2Rot(<.0, DEG_TO_RAD * 45.0, .0>));}
}
```

Nous constatons en cliquant sur l'objet et en passant en mode édition que la rotation est bien celle attendue :



4.3 Combinaisons de rotations

Maintenant voyons un peu comment se comportent les rotations quand on les combine. Partons d'un simple cube sans rotation et appliquons lui quelques combinaisons et observons le résultat. Voici le cube de départ :

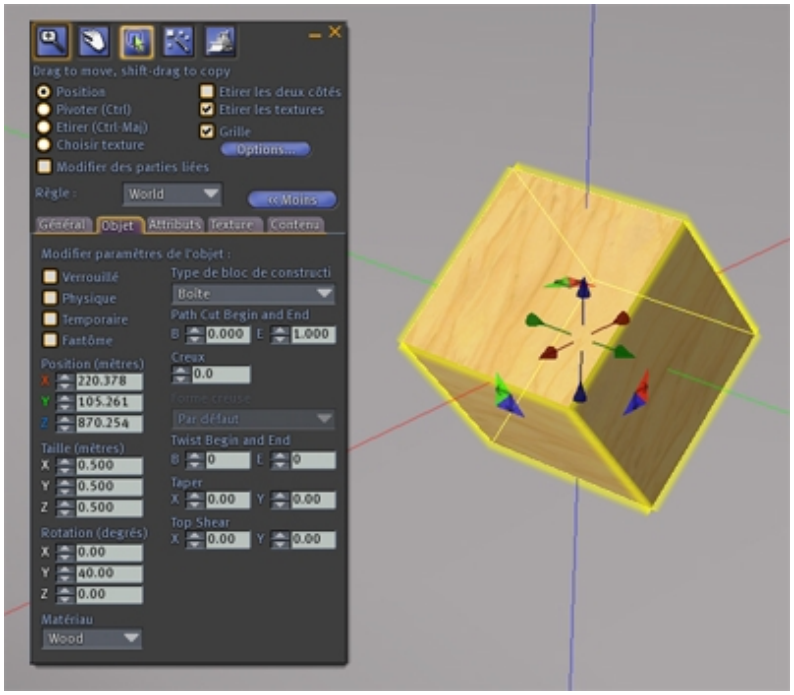


Commençons par ce script avec l'opérateur « * » :

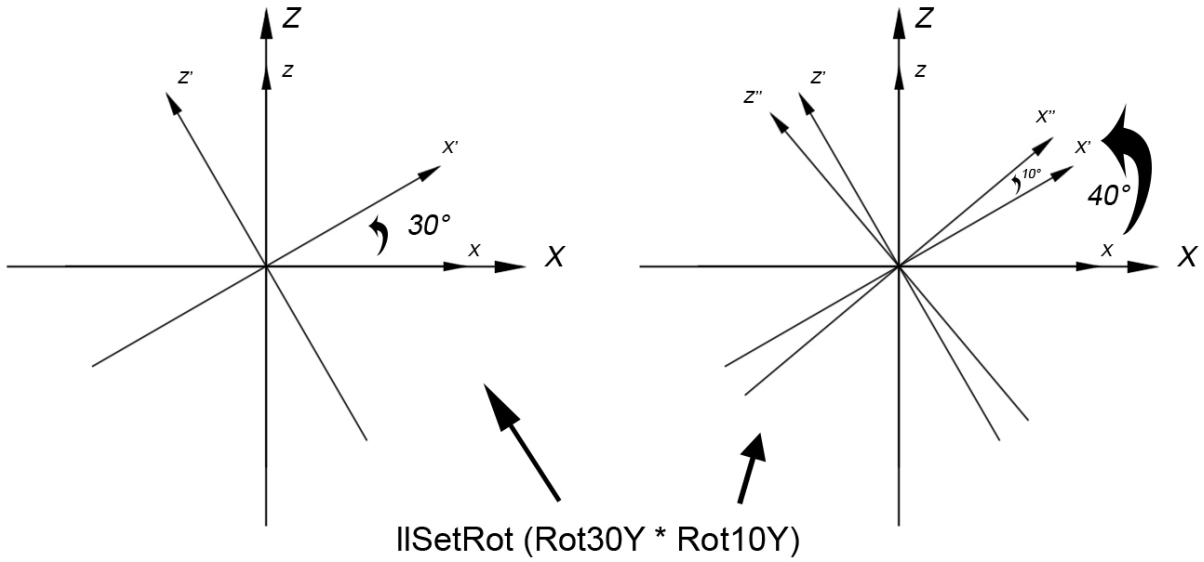
```
default {
```

```
        touch_start(integer total_number) {
                rotation rot30Y = llEuler2Rot(<.0, DEG_TO_RAD * 30.0, .0>);
                rotation rot10Y = llEuler2Rot(<.0, DEG_TO_RAD * 10.0, .0>);
                llSetRot(rot30Y * rot10Y);}
}
```

Résultat obtenu :



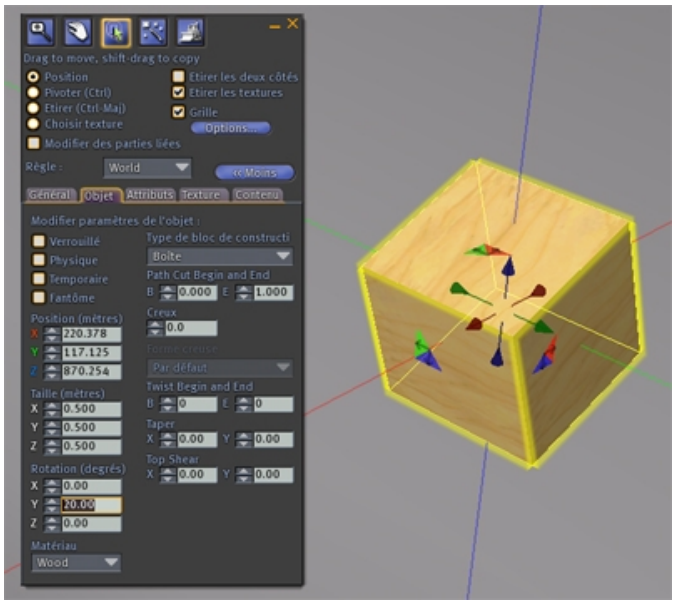
On a une addition des deux rotations avec 40° sur l'axe Y. En voici une illustration graphique :



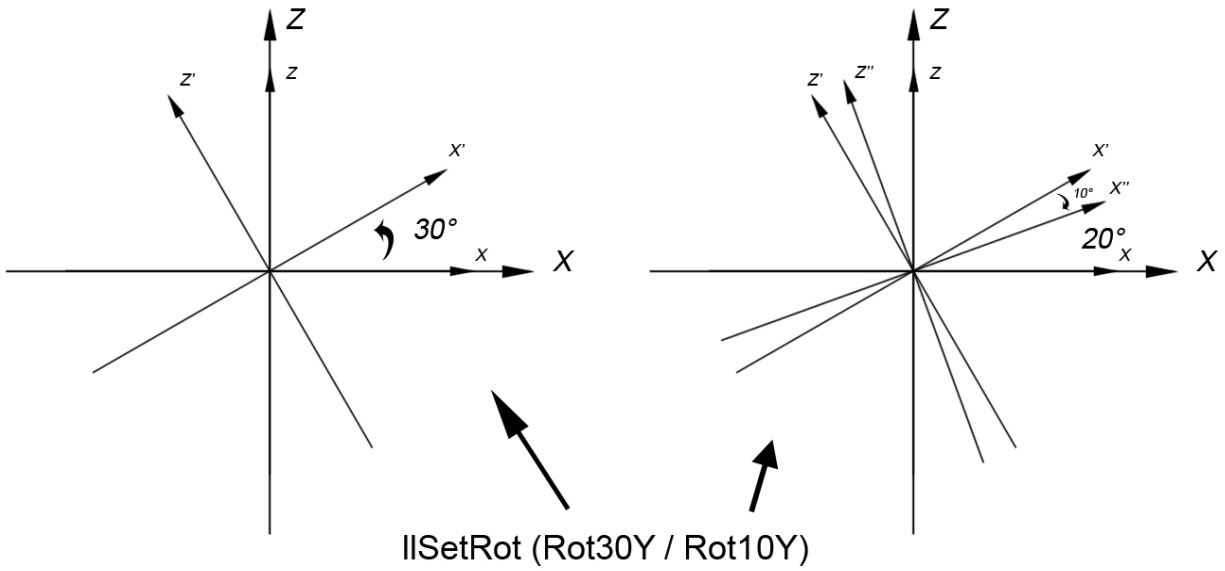
Que se passerait-il dans la même situation avec l'opérateur « / » ? Voici le code :

```
default {
        touch_start(integer total_number) {
                rotation rot30Y = llEuler2Rot(<.0, DEG_TO_RAD * 30.0, .0>);
                rotation rot10Y = llEuler2Rot(<.0, DEG_TO_RAD * 10.0, .0>);
                llSetRot(rot30Y / rot10Y);}
}
```

Et le résultat obtenu en partant d'un cube sans rotation :



La rotation obtenue est cette fois une soustraction entre la première et la deuxième rotation : 30 – 10 = 20°. En voici une illustration graphique :



Gardez bien à l'esprit que les deux opérateurs n'ont rien à voir avec des multiplications et des divisions !

Faites maintenant le même essai avec ce code :

```
default {
        touch_start(integer total_number) {
                rotation rot1 = llEuler2Rot(<.0,DEG_TO_RAD * 30.0,.0>);
                rotation rot2 = llEuler2Rot(<.0,DEG_TO_RAD * 10.0,.0>);
                llSetRot(rot1 * (ZERO_ROTATION / rot2));}
}
```

Vous constatez que vous obtenez le même résultat qu'avec l'opérateur « / », une soustraction des deux rotations. Mais que signifie ZERO_ROTATION / rot2 ? Vous savez que l'opérateur « / » applique l'inverse de la rotation située à droite à la rotation située à gauche. Il se trouve que la rotation située à gauche est nulle dans notre cas (ZERO_ROTATION), au final nous obtenons donc l'inverse de « rot2 ». Et puisque nous ajoutons l'inverse de rot2 le résultat paraît logique. Attention ! Vous n'avez pas d'autre possibilité pour obtenir l'inverse d'une rotation, vous ne devez pas utiliser une écriture du genre « –rot2 » ! Une étude un peu détaillée des quaternions, dont je vous ferai grâce, montre que pour obtenir l'inverse d'une rotation il faut prendre l'inverse des 3 valeurs **x**, **y** et **z**, et laisser **s** non touché. On pourrait utiliser une fonction pour effectuer cette opération :

```
rotation getInverseRotation(rotation rot) {
        rot.x = -rot.x;
        rot.y = -rot.y;
        rot.z = -rot.z;
        return rot;}
```

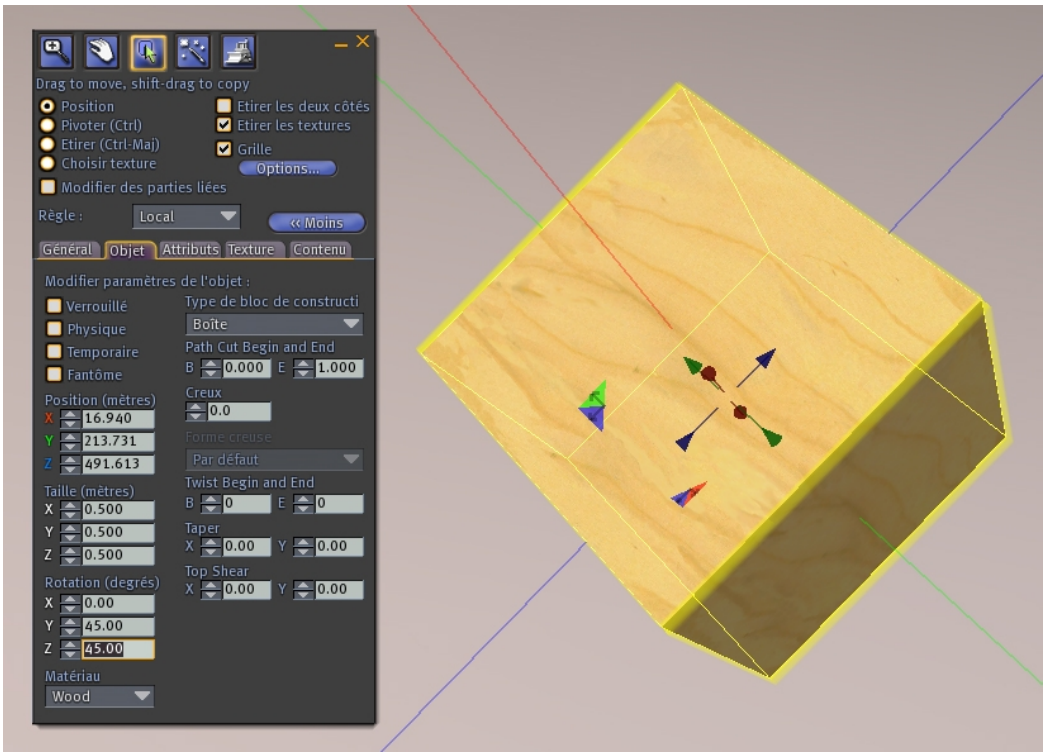
On peut aussi inverser **s** et ne pas toucher aux autres paramètres :

```
rotation getInverseRotation(rotation rot) {
        rot.s = -rot.s;
        return rot;}
```

Mais il est évidemment plus simple d'utiliser la forme « ZERO_ROTATION / rot ».

4.4 Rotation locale

Reprenons notre objet que nous avons fait tourner de 45° sur l'axe Y au point 4.2. Notre objet n'est plus aligné sur les axes globaux. Supposez que vous vouliez faire tourner votre objet de 45° sur son axe Z et non pas sur l'axe Z global ! En mode édition nous passons en coordonnées locales et la rotation est facile à effectuer :

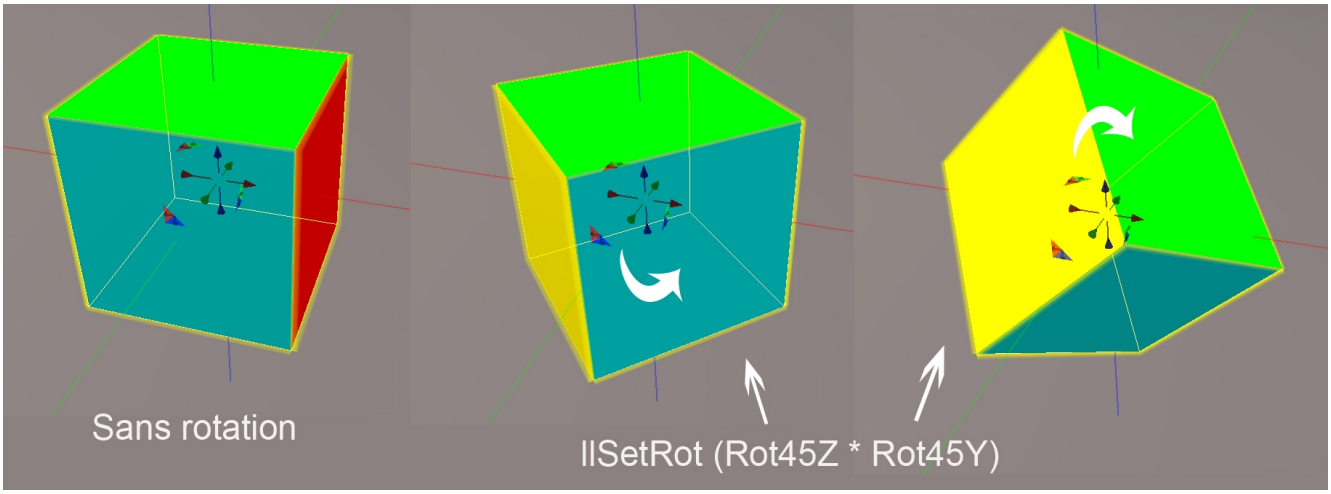


Mais comment faire dans un script ? Comment passer en mode local ? C'est ici que les quaternions montrent toute leur puissance.

Nous avons besoin de connaître la rotation actuelle de l'objet par rapport aux axes globaux, c'est la fonction **llGetRot** qui nous donne ce renseignement. Il suffit de multiplier la rotation désirée par la rotation actuelle pour obtenir le résultat :

```
default{
	touch_start(integer total_number) {
		llSetRot(llEuler2Rot(<.0, .0, DEG_TO_RAD * 45.0>) * llGetRot());}
}
```

Arrivé à ce stade vous vous posez certainement quelques questions sur cette rotation, c'est bien naturel et je vais vous éclairer un peu. Il faut bien comprendre le fonctionnement de la fonction **llSetRot**. Celle-ci applique les rotations passées en paramètres à partir d'un prim <u>sans rotation</u>. Dans le script ci-dessus on commence par appliquer une rotation positive de 45° au prim <u>sans rotation initiale</u>. Ensuite on applique une rotation positive de 45° sur l'axe Y qui correspond à la rotation initiale du prim. Voici une illustration détaillée pour bien comprendre ce processus :

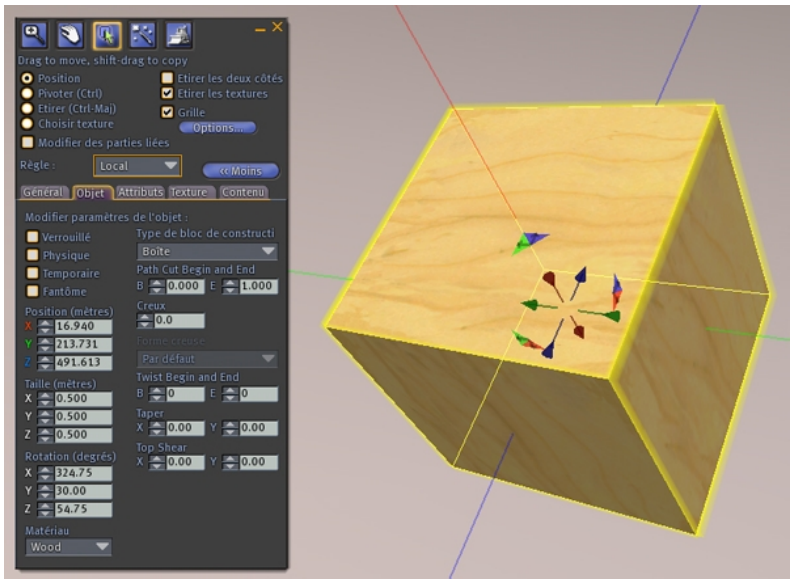


Vous remarquez qu'il n'y a pas vraiment de rotation locale mais un enchaînement de deux rotations globales.

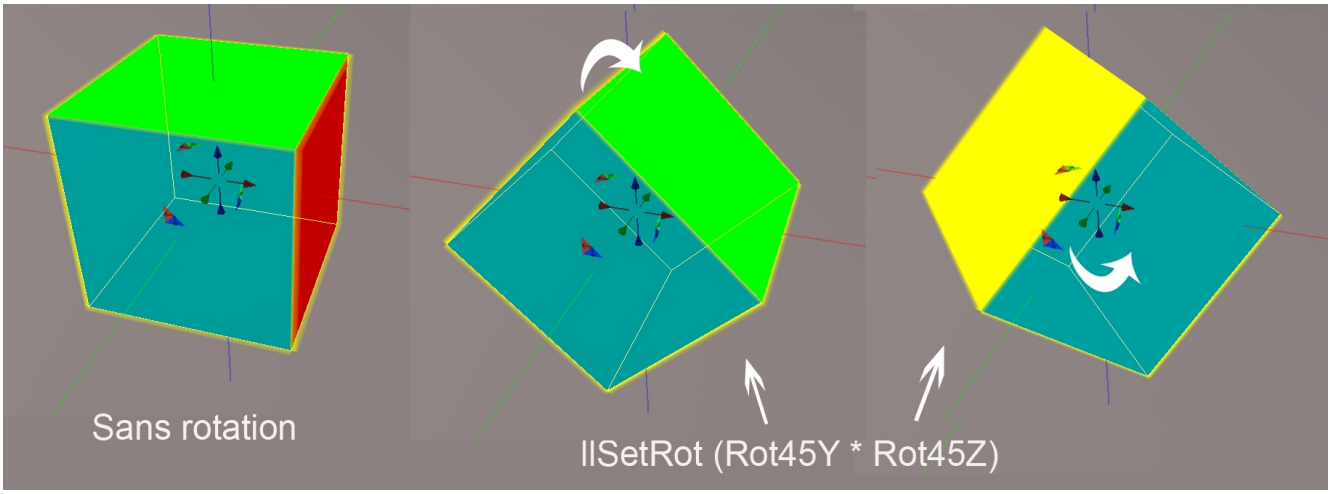
Mais attention cette opération n'est pas commutative ! Pour vous en assurer essayez le script suivant à la place :

```
default{
        touch_start(integer total_number) {
                llSetRot(llGetRot() * llEuler2Rot(<.0, .0, DEG_TO_RAD * 45.0>));          }
}
```

Vous obtenez ce résultat :



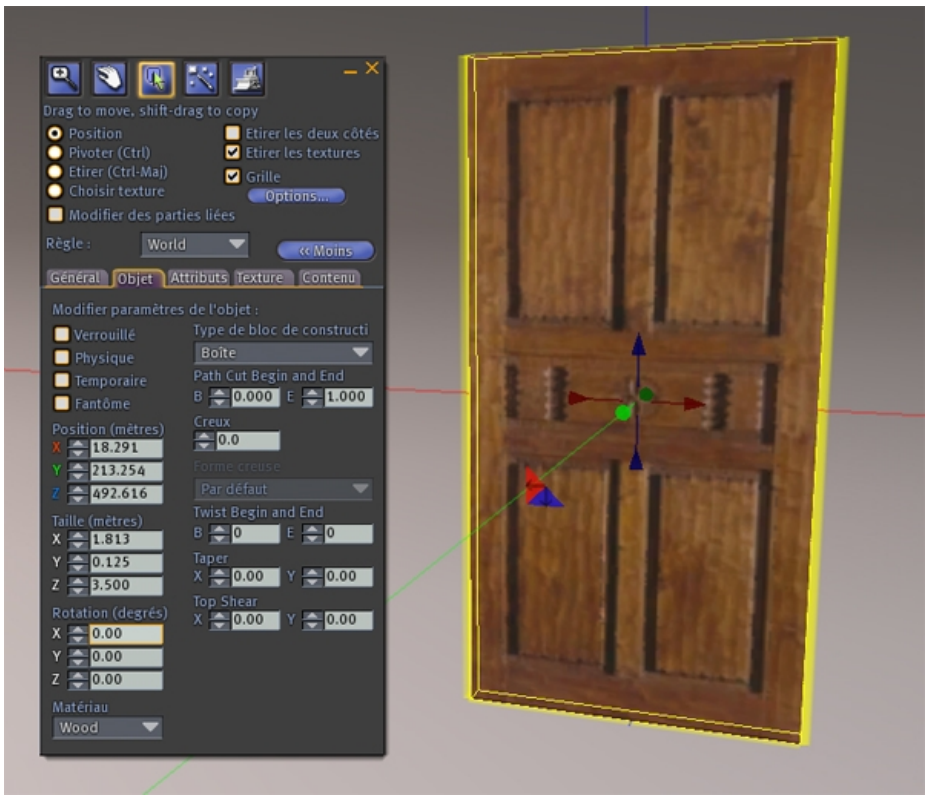
C'est devenu le bordel dans les rotations ! Que s'est-t-il passé ? En fait l'objet a tourné sur l'axe Z global et non pas l'axe Z local. Voici une autre illustration pour comprendre ce qu'il se passe :



J'espère que désormais vous serez attentif à l'ordre des opérandes dans vos opérations sur les rotations !

4.5 Translation indépendante de la rotation

Vous avez construit votre jolie maison sur SL avec des textures calibrées au millimètre et vous mettez de belles portes coulissantes.



Vous récupérez ce script dans mon guide pour manœuvrer vos portes :

```
vector largeurPorte;
```

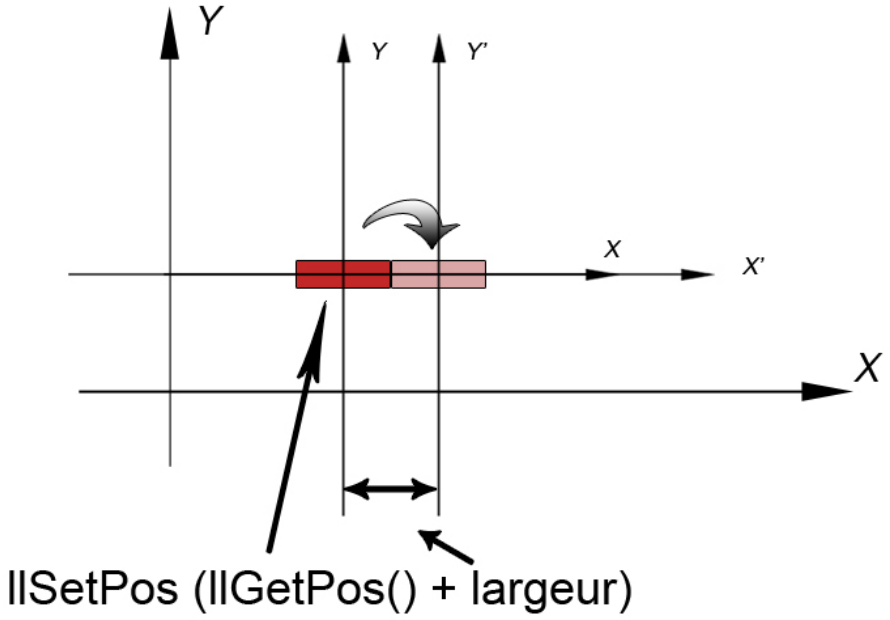
```
default {
	state_entry() {
		vector dimensions = llGetScale();
		largeurPorte = <dimensions.x, .0, .0>;
		state ferme;}
}
state ouvert {
	touch_start(integer total_number) {
		llSetPos(llGetPos() - largeurPorte);
		state ferme;
	}
}
state ferme {
	touch_start(integer total_number) {
		llSetPos(llGetPos() + largeurPorte);
		state ouvert;}
}
```

Vous testez votre script et tout fonctionne à merveille ! Vous remballez votre maison et vous la posez sur votre nouveau terrain. Pour vous adapter à l'environnement vous tournez votre maison de 45°. Vous êtes pleinement satisfait mais... au moment où vous manoeuvrez votre porte vous constatez avec horreur qu'elle part de travers !!! Que se passe-t-il ?

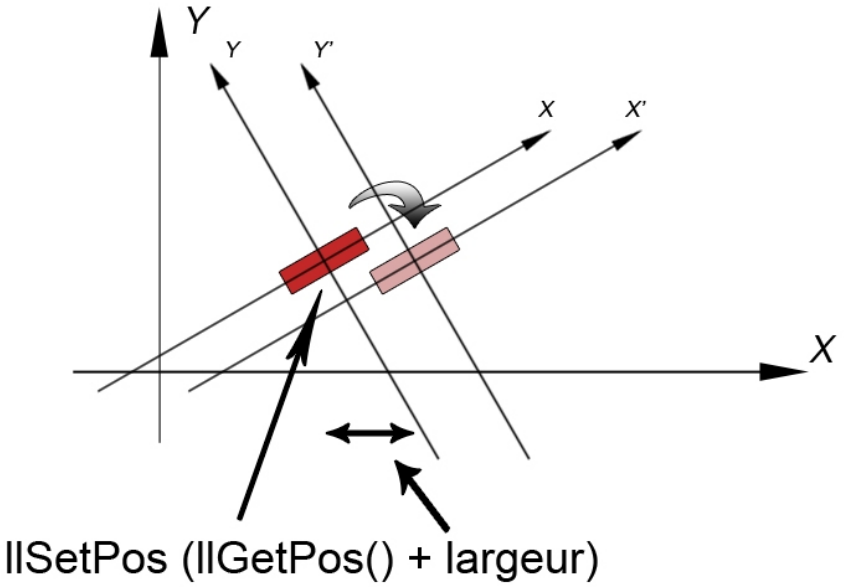
Votre translation s'effectue sur les axes globaux mais votre porte a tourné en même temps que votre maison. Pour que tout fonctionne correctement la translation devrait être orientée selon la nouvelle rotation. Il suffit donc d'appliquer la rotation de la porte à la translation pour que tout rentre dans l'ordre :

```
vector largeurPorte;
default {
	state_entry() {
		vector dimensions = llGetScale();
		largeurPorte = <dimensions.x, .0, .0>;
		state ferme;}
}
state ouvert {
	touch_start(integer total_number) {
		llSetPos(llGetPos() - largeurPorte * llGetRot());
		state ferme;
	}
}
state ferme {
	touch_start(integer total_number) {
		llSetPos(llGetPos() + largeurPorte * llGetRot());
		state ouvert;}
}
```

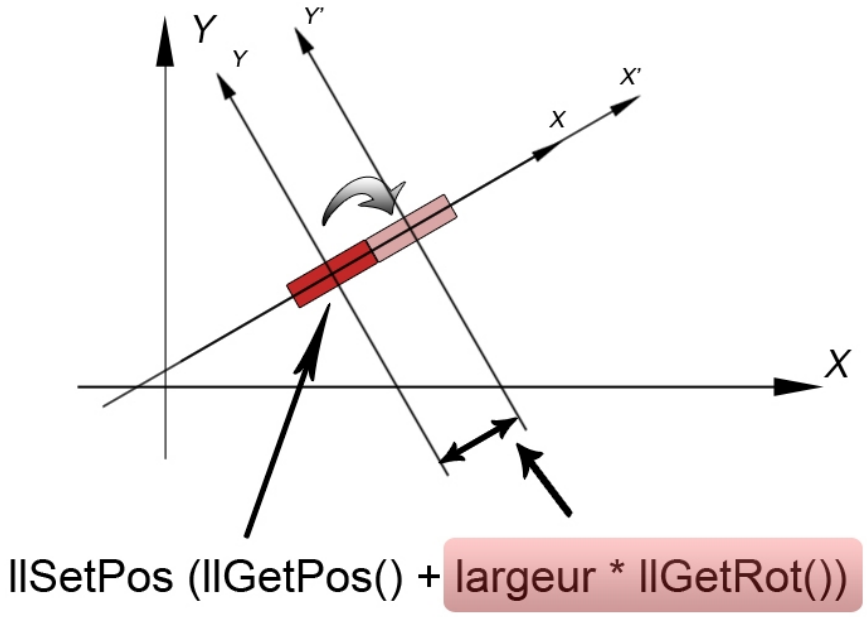
Des illustrations peuvent vous aider dans la compréhension de ce mouvement. Partons du premier script sans tenir compte de la rotation avec une porte parfaitement alignée sur les axes de la maison. Voici une vue de dessus :



On peut voir sur cette illustration que tout se passe bien parce que l'axe X de la porte est aligné sur l'axe X global. Mais imaginons maintenant qu'il n'en soit pas ainsi, autrement dit les axes X ne sont plus alignés :



Dans ce cas la porte se déplace toujours selon l'axe X global, mais comme elle n'est pas alignée sur cet axe le déplacement n'est pas correct. Il faut orienter le vecteur représentant le déplacement avec la rotation actuelle du prim :



Maintenant tout se passe bien parce que le vecteur de déplacement est aligné sur l'axe X local du prim.

5. Rotation continue

Les rotations que nous avons vues jusque là correspondent à des actions ponctuelles. Nous allons maintenant envisager des rotations continues, donc des successions de micro rotations qui vont nous donner l'illusion d'un mouvement continu. Voyons ce que nous propose LSL pour effectuer cela.

5.1 Rotation côté client

La rotation la plus fluide qu'il est possible d'obtenir simplement est celle offerte par la fonction **llTargetOmega**. C'est une fonction un peu particulière qui est finalement peu utilisée à cause de ses inconvénients. Le principal réside dans le fait qu'elle est exécutée côté client. Ce qui signifie que c'est votre ordinateur qui calcule la rotation en local sans avoir besoin d'une intervention du serveur. L'avantage évident est la fluidité incomparable du mouvement. L'inconvénient majeur réside dans le fait que chaque client voit sa propre rotation sans qu'il soit possible de synchroniser ces différents mouvements. Il faut tout de même signaler que dans le cas d'un objet « physique » le mouvement est régulièrement rafraîchi par le moteur de rendu.

Voyons un peu les paramètres de cette fonction :

Paramètre	Nom	Type	Fonction
1	axis	**vector**	Axe de rotation
2	spinrate	**float**	Vitesse de rotation en radians par seconde
3	gain	**float**	Gain (uniquement pour objet « physique »)

Pour obtenir une vitesse de rotation fiable il est conseillé de « normaliser » le vecteur qui représente l'axe de rotation, c'est-à-dire le ramener à une valeur unitaire. En effet la vitesse de rotation est obtenue à partir de la formule suivante :

Vitesse de rotation = axis * spinrate

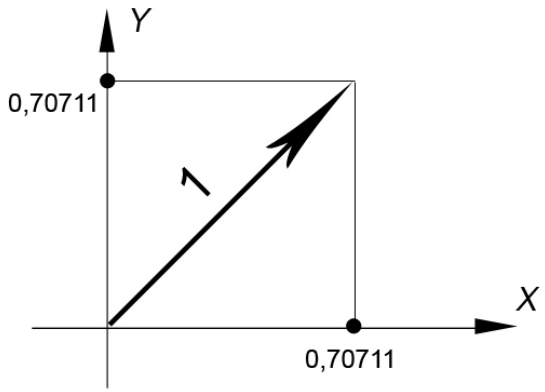
Prenons un exemple pour illustrer cela :

axis	spinrate	Vitesse angulaire
<1,0,0>	2 * PI	Un tour par seconde autour de l'axe X
<1,1,0>	2 * PI	Un tour par seconde autour de l'axe X et autour de l'axe Y
<0.70711, 0.70711, 0>	2 * PI	Un tour par seconde autour d'un axe médian entre les axes X et Y

Le premier cas est tout simple. J'ai considéré un vecteur unitaire sur l'axe X. Le résultat est facilement prévisible. Le second cas est plus problématique. J'ai défini un axe combiné à mi-chemin entre les axes X et Y. Mais le fait de définir le vecteur **<1,1,0>** ne donne pas d'information pertinente sur la valeur du module de ce vecteur. En faisant un calcul pas trop complexe on en déduit que le module fait en gros **1.414** et donc la vitesse de rotation résultante est de **2.828 * PI** radians par seconde. Mais il est quand même plus élégant de passer par le troisième cas avec un axe identique mais une vitesse plus facile à déterminer. Peut-être vous demandez-vous comment j'ai défini ce vecteur. Le but est d'obtenir une valeur de 1 pour le module du vecteur. Dans LSL cela est facile à déterminer avec la fonction **llVecNorm** qui renvoie justement un vecteur unitaire à partir d'un vecteur quelconque. En d'autres termes vous avez :

llVecNorm(<1,1,0>) = <0.70711, 0.70711, 0>

Et voici une illustration pour visualiser l'intérêt de cette fonction :



Pour arrêter un rotation démarrée avec la fonction llTargetOmega il suffit de réutiliser cette fonction avec des valeurs nulles pour les paramètres **spinrate** et **gain**.

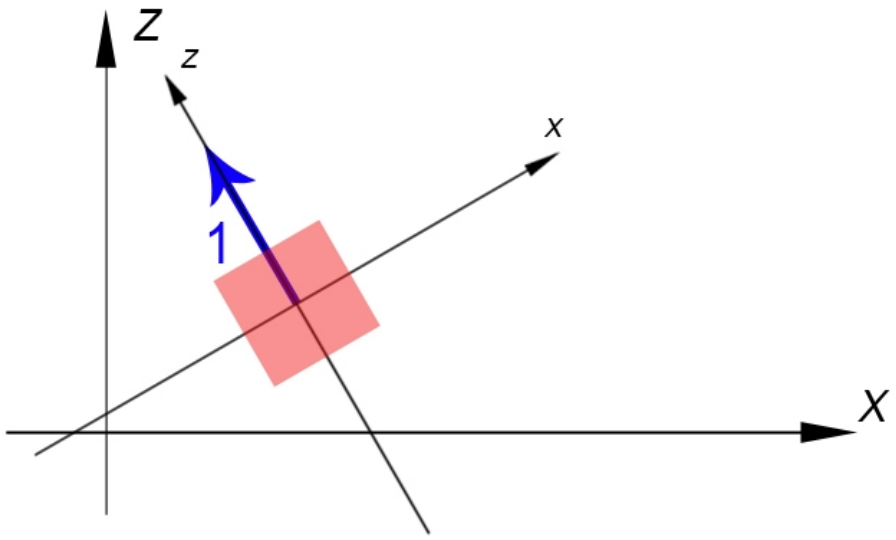
Et maintenant quelques exemples d'utilisation en commençant par un cas tout simple de rotation d'un demi-tour par seconde sur l'axe Z. Créez ce script dans un simple box sans rotation :

```
default {
	touch_start(integer total_number) {
		llTargetOmega(<.0,.0,1.0>, PI, 1.0);
		llOwnerSay("Marche");
		state marche;}
}
state marche {
	touch_start(integer total_number) {
		llTargetOmega(<.0,.0,1.0>, .0, .0);
		llOwnerSay("Arret");
		state default;}
}
```

Reprenons cet exemple en changeant l'axe (grande diagonale de la boîte) avec la même vitesse de rotation. Partez encore d'un box sans rotation.

```
default {
	touch_start(integer total_number) {
		llTargetOmega(llVecNorm(<1.0,1.0,1.0>), PI, 1.0);
		llOwnerSay("Marche");
		state marche;}
}
state marche {
	touch_start(integer total_number) {
		llTargetOmega(llVecNorm(<1.0,1.0,1.0>), .0, .0);
		llOwnerSay("Arret");
		state default;}
}
```

Dans les deux cas précédents la rotation s'effectue autour des axes globaux. Comme faire pour qu'une rotation se fasse autour d'un axe local ? Supposons que notre box utilisé ci-dessus possède maintenant une rotation négative de 30° sur l'axe Y. Et nous voulons que la rotation s'effectue autour de l'axe local Z. Il nous faut donc trouver le vecteur unitaire qui représente la direction de l'axe local Z en valeur globale. Vite un petit schéma pour visualiser le problème :



Encore une histoire de rotation ! Décomposons le problème :

- Lorsque le prim n'a aucune rotation le vecteur unitaire représentant l'axe Z est : <.0,.0,1.0>
- La rotation négative de 30° sur l'axe Y est : llEuler2Rot(<.0, -30 * DEG_TO_RAD, .0>)
- Pour orienter le vecteur selon la direction prise par le prim il suffit de multiplier ce vecteur par la rotation : <.0,.0,1.0> * llEuler2Rot(<.0,30 * DEG_TO_RAD, .0>), le calcul montre qu'on obtient le vecteur <-0.499, .0, 0.866>. Il s'agit bien évidemment d'un vecteur unitaire puisqu'on s'est contenté de faire une rotation d'un vecteur unitaire.

LSL propose une fonction qui aboutit directement à ce résultat, c'est **llRot2Up**. Cette fonction qui attend un seul paramètre de type **rotation** renvoie justement un vecteur unitaire représentant l'axe Z après la rotation transmise. Autrement dit on peut obtenir le vecteur souhaité ci-dessus avec la formule :

llRot2Up(llEuler2Rot(<.0, -30 * DEG_TO_RAD, .0>))

De la même manière il existe les fonction **llRot2Fwd** et **llRot2Left** pour les axes X et Y.

Maintenant que nous avons mis tout cela au point reprenons notre box pour effectuer la rotation souhaitée :

```
vector axeZ;
default {
	state_entry() {
		rotation rot = llEuler2Rot(<.0, -30 * DEG_TO_RAD, .0>);
		llSetRot(rot);
		axeZ = llRot2Up(rot);
```

```
                state stop;}
}
state stop {
        touch_start(integer total_number) {
                llTargetOmega(axeZ, PI, 1.0);
                llOwnerSay("Marche");
                state marche;}
}
state marche {
        touch_start(integer total_number) {
                llTargetOmega(axeZ, .0, .0);
                llOwnerSay("Arret");
                state default;}
}
```

Vous constatez que la rotation s'effectue bien maintenant selon l'axe souhaité en rotation locale.

La fonction **llSetTargetOmega** se prête mal à des rotations partielles parce que nous ne pouvons pas bien maîtriser l'aspect temporel et la synchronisation et nous allons voir des façons plus efficaces de procéder.

5.2 Rotation côté serveur

5.2.1 Avec llSetRot

Nous avons déjà vu la fonction **llSetRot**, il est possible d'obtenir avec elle une rotation continue avec des appels répétés. Testez ce script sur un simple prim :

```
default {
        state_entry() {
                rotation rot = llEuler2Rot(<.0, .0, PI / 20>);
                while(TRUE) llSetRot(llGetRot() * rot);}
}
```

Vous pouvez une rotation un peu saccadée. Mon code n'est pas très joli, j'ai créé une boucle infinie et l'appel à la fonction **llSetRot** se fait cycliquement, mais à quel rythme ? En fait la fonction **llSetRot** stoppe le script pendant 0,2 secondes, c'est donc le rythme des appels. Comparez avec ce script :

```
rotation rot;
default {
        state_entry() {
                rot = llEuler2Rot(<.0, .0, PI / 20.0>);
                llSetTimerEvent(.2);}
        timer() {llSetRot(llGetRot() * rot);}
}
```

Le résultat est strictement le même. Essayez de changer la valeur de la temporisation en la diminuant. Vous ne changerez rien au résultat. Moralité : la fonction **llSetRot** n'est pas vraiment idéale pour une rotation fluide. Existe-t-il une autre fonction plus sympathique ? La réponse est oui.

5.2.2 Avec llRotLookAt

Testez le script suivant :

```
rotation rot;
float strength;
float damping;
default {
	state_entry() {
		rot = llEuler2Rot(<.0, .0, PI / 80.0>);
		strength = llGetMass() / 2.0;
		damping = strength / 10.0;
		llSetTimerEvent(.05);}

	timer() {llRotLookAt(llGetRot() * rot, strength, damping);}
}
```

La rotation obtenue est déjà bien plus fluide. Voyons de plus près cette fonction **llRotLookAt**. Voici ses paramètres :

Paramètre	Nom	Type	Fonction
1	target	**rotation**	Rotation
2	strength	**float**	Force
3	damping	**float**	Délai

Par rapport à **llSetRot** on a deux paramètres supplémentaires. Ces derniers ne sont vraiment pertinents que pour les objets « physiques ». Par contre, l'immense avantage de cette fonction par rapport à **llSetRot** est qu'elle n'impose aucun délai au script ! De coup j'ai pu me permettre de baisser la valeur de la temporisation et de prendre une incrémentation angulaire plus fine. J'ai aussi pris les valeurs des deux derniers paramètres conseillées par le wiki, même si celles-ci n'ont pas vraiment d'importance dans le cas d'un objet non « physique ». Par contre si vous utilisez un objet « physique » soyez vigilant pour ces deux paramètres et n'hésitez pas à faire des tests.

5.2.3 Avec llSetLinkPrimitiveParamsFast

La version 1.8 des serveurs a introduit la nouvelle fonction **llSetLinkPrimitiveParamsFast** qui a la même syntaxe que la fonction **llSetLinkPrimitiveParams** mais qui évite la temporisation de 0,2s. On peut donc réaliser désormais une rotation fluide côté serveur avec cette fonction. Testez ce script :

```
rotation rot;
default {
	state_entry() {
		rot = llEuler2Rot(<.0, .0, PI / 200.0>);
		llSetTimerEvent(.02);
	}
	timer() {
		llSetLinkPrimitiveParamsFast(0, [PRIM_ROTATION, llGetRot() * rot]);
	}
}
```

6. Rotation excentrée

Dans les cas vus précédemment la rotation s'effectue autour du centre de gravité de l'objet. Nous avons besoin parfois d'une rotation dont le centre ne coïncide pas avec ce centre de gravité. Par exemple une porte qui doit pivoter sur un de ses bords, ou une planète qui tourne autour du soleil.

6.1 Une planète

Supposez que vous reconstituez une version schématique du système solaire dans **SL**. Vous réalisez un magnifique soleil et les planètes devant tourner autour de lui. Dans ce cas la planète subit une double rotation : sur elle-même et autour du soleil. La première rotation peut être simplement gérée par la fonction **llTargetOmega** qui effectue une rotation fluide calculée sur le poste client. Par contre la deuxième rotation doit être gérée par des repositionnements réguliers de la planète à l'aide de la fonction **llSetPos**. Comment allons-nous déterminer ces positions ?

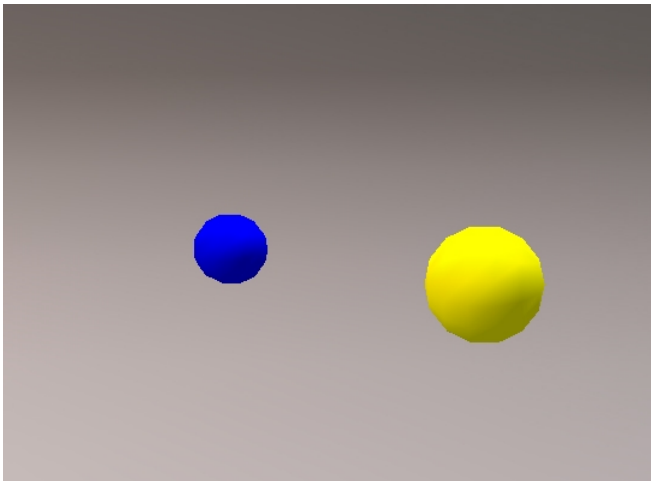
En fait il suffit de faire tourner un vecteur autour du centre de référence. Pour faire tourner un vecteur les quaternions viennent encore à notre secours : il suffit de multiplier le vecteur par la rotation pour obtenir le résultat, ce que nous avons vu au chapitre précédent. Résumons la démarche :

1. La position du centre de rotation **posSoleil** ne peut pas être connue si la planète n'est pas liée au soleil (sauf à communiquer par le Chat), ce que nous prenons comme hypothèse, il faut donc l'entrer directement dans le script, pour notre exemple je suppose que le soleil se situe à 2 mètres de la planète sur l'axe X
2. La position de la planète **posPlanete** nous est donnée par la fonction **llGetPos**
3. Le vecteur est la différence entre ces deux positions : **vecteur = posPlanete – posSoleil**
4. La rotation du vecteur est donnée par la multiplication par la rotation : **rotVecteur = vecteur * rot**
5. Le résultat final consiste à prendre le centre de référence pour positionner le vecteur : **nouvellePosition = posSoleil + rotVecteur**

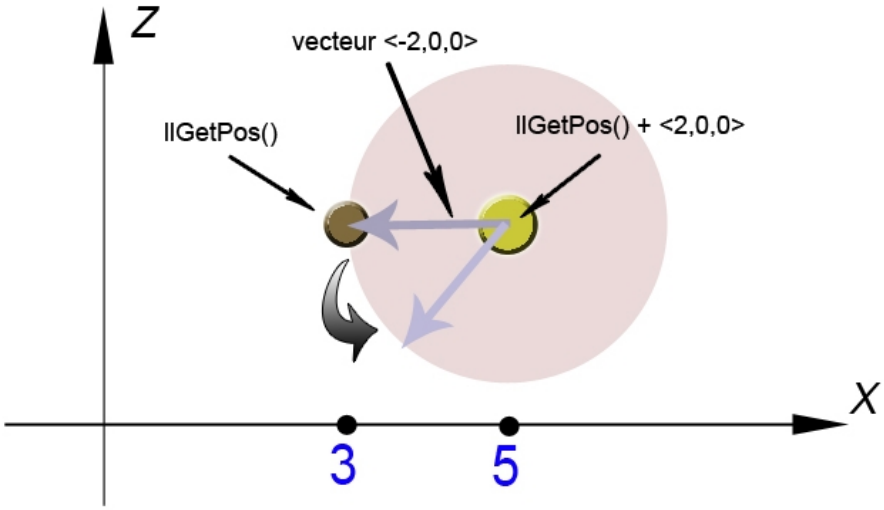
Nous utilisons un timer pour donner une rotation de 1° toutes les 0,04 secondes :

```
vector posSoleil;
rotation rot;
default
{
        state_entry(){
                rot = llEuler2Rot(<.0, .0, DEG_TO_RAD * 1.0>);
                vector posPlanete = llGetPos();
                posSoleil  = posPlanete + <2.0, .0, .0>;
                llSetTimerEvent(.04);}
        timer() {
                vector posPlanete = llGetPos();
                vector vecteur = posPlanete - posSoleil;
                vector rotVecteur = vecteur * rot;
                vector nouvellePosition = posSoleil + rotVecteur;
                llSetLinkPrimitiveParamsFast(0, [PRIM_POSITION, nouvellePosition]);}
}
```

Bien entendu ce script ne fonctionne que pour un mouvement se situant dans le plan XY.



Voici une illustration pour bien visualiser ce mouvement avec des valeurs simples :



6.2 Porte rotative

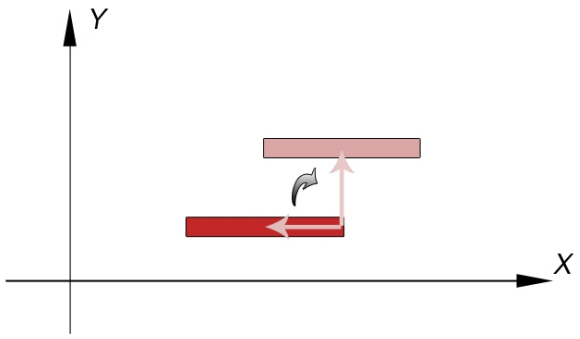
Le raisonnement utilisé précédemment pour la planète est-il applicable pour une porte rotative ? Faisons un essai :

```
vector posAxe;
animation(float angle) {
        rotation rot = llEuler2Rot(<.0, .0, DEG_TO_RAD * angle>);
        vector posPorte = llGetPos();
        vector vecteur = posPorte - posAxe;
        vector rotVecteur = vecteur * rot;
```

```
        vector nouvellePosition = posAxe + rotVecteur;
        llSetPos(nouvellePosition);
}
default {
        state_entry(){
                vector posPorte = llGetPos();
                vector dimensions = llGetScale();
                vector largeurPorte = <dimensions.x, .0, .0>;
                posAxe = posPorte + largeurPorte / 2.0;
                state ouvert;}
}
state ouvert {
        touch_start(integer total_number) {
                animation(90.0);
                state ferme;}
}
state ferme {
        touch_start(integer total_number) {
                animation(-90);
                state ouvert;}
}
```

Quand nous appliquons ce script à notre porte nous constatons qu'elle se positionne au bon endroit mais sans tourner sur elle-même lors de l'ouverture :

Observez cette illustration pour bien visualiser le problème :



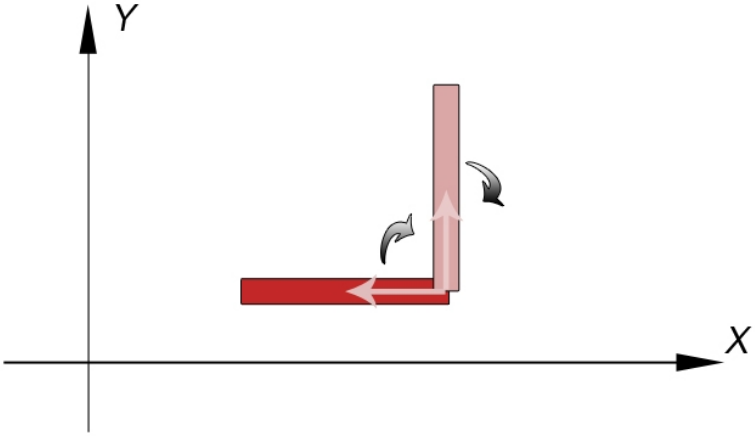
Il faut donc ajouter une rotation pour obtenir un fonctionnement convenable. Voici la fonction **animation** modifiée en conséquence :

```
animation(float angle) {
        rot = llEuler2Rot(<.0, .0, DEG_TO_RAD * angle>);
        vector posPorte = llGetPos();
        vector vecteur = posPorte - posAxe;
        vector rotVecteur = vecteur * rot;
        vector nouvellePosition = posAxe + rotVecteur;
        llSetPos(nouvellePosition);
        llSetRot(rot * llGetRot());}
```

Remarquez à nouveau l'ordre des termes pour la rotation : d'abord la rotation locale désirée et ensuite la rotation globale actuelle. Cette fois notre porte fonctionne parfaitement bien. Le seul petit défaut est le décalage temporel entre la translation et la rotation. En regroupant les deux avec une fonction **llSetPrimitiveParams** on corrige facilement ce défaut :

```
animation(float angle) {
  rot = llEuler2Rot(<.0, .0, DEG_TO_RAD * angle>);
  vector posPorte = llGetPos();
  vector vecteur = posPorte - posAxe;
  vector rotVecteur = vecteur * rot;
  vector nouvellePosition = posAxe + rotVecteur;
  llSetPrimitiveParams([PRIM_ROTATION, rot * llGetRot(), PRIM_POSITION, nouvellePosition]);}
```

Maintenant que la translation s'accompagne d'une rotation notre porte fonctionne correctement comme vous pouvez le constater en observant cette illustration :



Tout semble aller pour le mieux à présent mais n'y a-t-il pas quelque chose que nous avons oublié ? Si vous déplacez votre porte, où si vous lui faites subir une rotation, la position de l'axe que vous avez mémorisée n'est plus bonne. Il faut donc retrouver la position de l'axe à chaque commande, comment faire cela ?

Il faut d'une part déterminer la position de l'axe à chaque commande et d'autre part rendre ce calcul indépendant de la rotation de la porte. Nous avons vu un cas de figure similaire au point 4.5. Voici le script résultant :

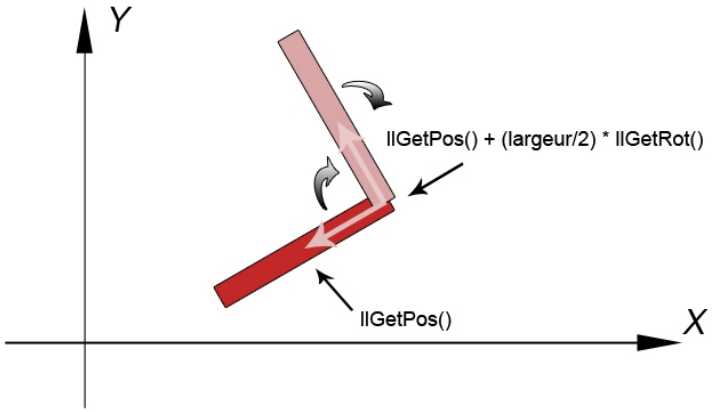
```
vector largeurPorte;
```

```
animation(float angle) {
	vector posPorte = llGetPos();
	vector posAxe = posPorte + (largeurPorte / 2.0) * llGetRot();
	rotation rot = llEuler2Rot(<.0, .0, DEG_TO_RAD * angle>);
	vector vecteur = posPorte - posAxe;
	vector rotVecteur = vecteur * rot;
	vector nouvellePosition = posAxe + rotVecteur;
	llSetPrimitiveParams([PRIM_ROTATION, rot * llGetRot(), PRIM_POSITION, nouvellePosition]);}
default {
	state_entry(){
		vector dimensions = llGetScale();
		largeurPorte = <dimensions.x, .0, .0>;
		state ouvert;}
}
state ouvert {
	touch_start(integer total_number) {
		animation(90.0);
		state ferme;}
}
state ferme {
	touch_start(integer total_number) {
		animation(-90);
		state ouvert;}
}
```

Comme vous pouvez le constater tous les calculs sont regroupés dans la fonction **animation** et l'indépendance par rapport à la rotation se situe dans la différence apportée dans cette ligne de code :

```
vector posAxe = posPorte + (largeurPorte / 2.0) * llGetRot();
```

Maintenant la porte peut être déplacée et subir des rotations sans dommage sur sa commande. Voici une nouvelle illustration :



A bien observer ce code on se rend compte qu'il est possible de le simplifier parce que nous connaissons déjà le vecteur . Voici une version plus élégante :

```
vector largeurPorte;
animation(float angle) {
	vector posPorte = llGetPos();
	rotation rot = llEuler2Rot(<.0, .0, DEG_TO_RAD * angle>);
	vector vecteur = -(largeurPorte / 2.0) * llGetRot();
	vector posAxe = posPorte - vecteur;
	vector rotVecteur = vecteur * rot;
	vector nouvellePosition = posAxe + rotVecteur;
	llSetPrimitiveParams([PRIM_ROTATION, rot * llGetRot(), PRIM_POSITION, nouvellePosition]);}
default {
	state_entry(){
		vector dimensions = llGetScale();
		largeurPorte = <dimensions.x, .0, .0>;
		state ouvert;}
}
state ouvert {
	touch_start(integer total_number) {
		animation(90.0);
		state ferme;}
}
state ferme {
	touch_start(integer total_number) {
		animation(-90);
		state ouvert;}
}
```

Tout va bien désormais... quoi que… tout va bien uniquement si l'axe Z de la porte est aligné sur l'axe Z global, ce qui paraît tout naturel pour une porte de maison. Mais imaginez que vous désirez utiliser ce script pour des volets situés sur une face de toit inclinée par exemple. Autrement dit si vous faites subir à votre prim une rotation sur l'axe X ou Y (à plus forte raison sur ces deux axes).

Dans ce cas notre rotation de vecteur, que nous avons limitée à l'axe Z, n'est plus valable parce que l'axe de rotation du vecteur n'est plus l'axe Z. Mais il n'est pas si difficile de trouver cet axe, après tout c'est seulement l'axe Z de la porte et on le trouve facilement avec la formule :

```
<.0, .0, 1.0> * llGetRot()
```

Mais nous avons vu aussi une fonction au chapitre précédent (point 5.1) qui permet d'obtenir l'axe Z à partir d'une rotation. La formule précédente est équivalente à celle-ci :

```
llRot2Up(llGetRot())
```

Comment déterminer la rotation sur cet axe ? Tout simplement en utilisant la fonction **llAxisAngle2Rot** qui donne justement la rotation en partant d'un axe et d'un angle :

```
rot = llAxisAngle2Rot(llRot2Up(llGetRot()), DEG_TO_RAD * angle)
```

Reprenons une dernière fois notre script pour tenir compte de ces considérations :

```
vector largeurPorte;
```

```
animation(float angle) {
	vector posPorte = llGetPos();
	rotation rot = llAxisAngle2Rot(llRot2Up(llGetRot()), DEG_TO_RAD * angle);
	vector vecteur = -(largeurPorte / 2.0) * llGetRot();
	vector posAxe = posPorte - vecteur;
	vector rotVecteur = vecteur * rot;
	vector nouvellePosition = posAxe + rotVecteur;
	llSetPrimitiveParams([PRIM_ROTATION, llGetRot() * rot, PRIM_POSITION, nouvellePosition]);}
default {
	state_entry(){
		vector dimensions = llGetScale();
		largeurPorte = <dimensions.x, .0, .0>;
		state ouvert;}
}
state ouvert {
	touch_start(integer total_number) {
		animation(90.0);
		state ferme;}
}
state ferme {
	touch_start(integer total_number) {
		animation(-90);
		state ouvert;}
}
```

Maintenant notre porte est parfaite, elle peut subir toutes les rotations que l'on veut sans être affectée. Remarquez la rotation de la porte, puisque nous avons déterminé la rotation globale pour le vecteur il suffit de l'utiliser aussi pour la porte en se contentant de l'ajouter.

Pour finir ce serait quand même plus joli une porte qui s'ouvre progressivement plutôt ce mouvement brutal. Nous pouvons réaliser facilement cela avec la nouvelle fonction **llSetLinkPrimitiveParamsFast** :

```
vector largeurPorte;
animation(float angle) {
	integer i;
	for(;i < 90; ++i) {
		vector posPorte = llGetPos();
		rotation rot = llAxisAngle2Rot(llRot2Up(llGetRot()), DEG_TO_RAD * angle);
		vector vecteur = -(largeurPorte / 2.0) * llGetRot();
		vector posAxe = posPorte - vecteur;
		vector rotVecteur = vecteur * rot;
		vector nouvellePosition = posAxe + rotVecteur;
		llSetLinkPrimitiveParamsFast(0, [PRIM_ROTATION, llGetRot() * rot,
						PRIM_POSITION, nouvellePosition]);
		llSleep(.01);}
}
default {
	state_entry(){
		vector dimensions = llGetScale();
		largeurPorte = <dimensions.x, .0, .0>;
```

```
                state ouvert;}
}
state ouvert {
        touch_start(integer total_number) {
                animation(1.0);
                state ferme;}
}
state ferme {
        touch_start(integer total_number) {
                animation(-1.0);
                state ouvert;}
}
```

7. Objet lié

Tous les exemples que nous avons vus concernent des objets libres, non liés à un autre prim. Que se passe-t-il lorsque des objets sont liés ?

7.1 Quelques généralités

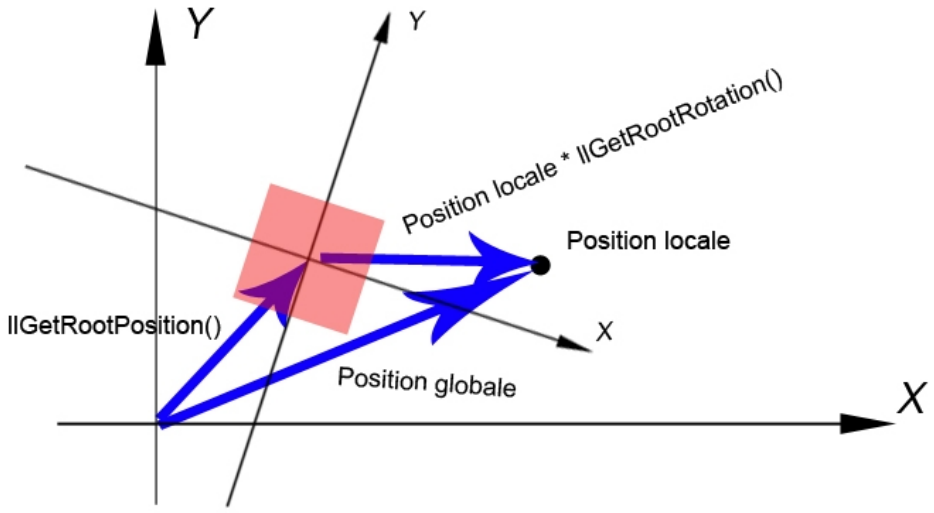
Lorsqu'un prim est lié à d'autres prims, et qu'il n'est pas la racine des liaisons, les translations deviennent relatives à la racine et de nouvelles fonctions apparaissent.

7.1.1 Translations

En ce qui concerne les translations la fonction **llSetPos** gère des coordonnées locales (il en est de même avec **llSetPrimitiveParams** ou la nouvelle fonction **llSetLinkPrimitiveParamsFast** qui évite la temporisation de 0,2s), c'est-à-dire relatives à la racine. Par contre la fonction **llGetPos** renvoie toujours les coordonnées globales. Une nouvelle fonction, **llGetLocalPos**, apparaît pour obtenir les coordonnées locales. Voici un tableau qui résume ces dispositions :

Fonction	Global	Local
llSetPos		X
llGetPos	X	
llGetLocalPos		X

Il est parfois nécessaire d'effectuer un changement de référentiel, que ce soit de global vers local ou inversement. Il est à noter qu'une prim enfant peut connaître la position et la rotation de la racine avec les fonctions **llGetRootPosition** et **llGetRootRotation**. La prim racine peut utiliser maintenant la fonction **llGetLinkPrimitiveParams** pour obtenir la position (position globale puisque c'est l'équivalent de **llGetPos**) ou rotation de ses prims enfants. Considérons en premier le passage de local vers global. Voici un petit schéma illustratif :



Nous avons un prim racine (en rose) qui a une certaine rotation par rapport aux axes globaux. On considère un point quelconque représenté « Position locale » sur l'illustration, dont nous ne connaissons que les coordonnées locales. Pour obtenir ses coordonnées globales il faut ajouter le vecteur représentant la position globale du prim racine obtenu par la fonction **llGetRootPosition** au vecteur représentant la position locale orientée par la rotation du prim racine obtenue par la fonction **llGetRootRotation**. Voici une fonction qui permet ce changement de référentiel :

```
vector Local2Global(vector position) {
	return llGetRootPosition() + position * llGetRootRotation();}
```

Son utilisation est simple : vous passez en paramètre un vecteur représentant des coordonnées locales, au retour vous obtenez les coordonnées globales.

Considérons maintenant la transformation inverse : de global vers local. Le schéma explicatif est évidemment le même mais cette fois la donnée d'entrée est le vecteur représentant les coordonnées globales. Pour obtenir les coordonnées locales il suffit de soustraire le vecteur représentant la position du prim racine et d'orienter le tout par la rotation du prim racine. Voici la fonction correspondante :

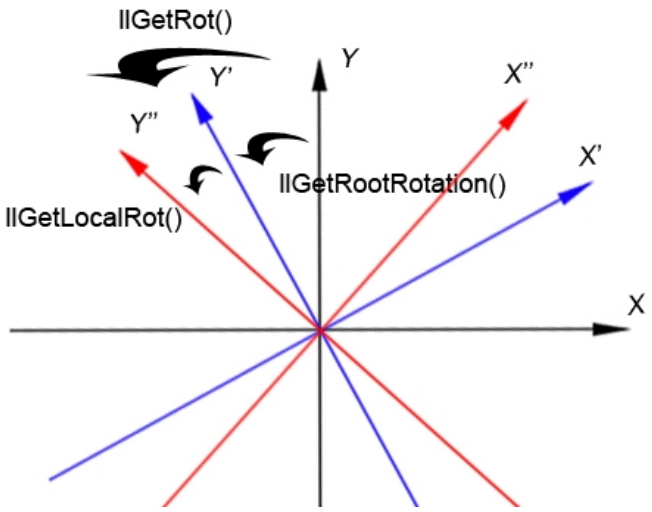
```
vector Global2Local(vector position) {
	return (position - llGetRootPosition()) / llGetRootRotation();}
```

7.1.2 Rotations

En ce qui concerne les rotations les fonctions **llSetRot** et **llGetRot** continuent de gérer des axes globaux (il en est de même avec **llSetPrimitiveParams** et **llSetLinkPrimitiveParamsFast**). Deux nouvelles fonctions, **llSetLocalRot** et **llGetLocalRot**, apparaissent pour gérer les axes locaux. Voici un tableau qui résume ces dispositions :

Fonction	Global	Local
llSetRot	Local avec ajout rotation racine	
llGetRot	X	
llSetLocalRot		X
llGetLocalRot		X

Il est parfois nécessaire d'effectuer un changement de référentiel, que ce soit de global vers local ou réciproquement. Considérons en premier le passage de local vers global. Voici un petit schéma illustratif :



Les axes du prim racine sont en bleu et ceux du prim enfant en rouge. On se rend facilement compte sur cette figure qu'on a la relation :

Rotation globale = llGetLocalRot() * llGetRootRotation()

Quant à la relation inverse (de Global vers Local) elle s'obtient facilement en soustrayant les rotations :

Rotation Locale = llGetRot() / llGetRootRotation()

Si vous voulez vérifier ces formules lancez le script suivant dans un prim lié avec une rotation fantaisiste :

```
Default {
        touch_start(integer total_number) {
                llOwnerSay((string)llGetRot());
                llOwnerSay((string)(llGetLocalRot() * llGetRootRotation()));
                llOwnerSay((string)llGetLocalRot());
                llOwnerSay((string)(llGetRot() / llGetRootRotation()));}
}
```

Voici un exemple des valeurs que j'obtiens :

Object: <-0.13134, -0.03667, 0.54494, 0.82731>
Object: <-0.13134, -0.03667, 0.54494, 0.82731>
Object: <-0.08052, -0.29147, 0.26334, 0.91608>
Object: <-0.08052, -0.29147, 0.26334, 0.91608>

Vous pouvez constater l'identité des valeurs pour les formules ci-dessus.

Je suis sûr que certains vont se poser la question : puisqu'une prim enfant dispose de fonctions lui permettant d'obtenir ou de fixer ses rotations indifféremment sur les axes locaux (**llSetLocalRot** et **llGetLocalRot**) ou globaux (**llSetRot** et **llGetRot**) à quoi bon s'inquiéter des changements de référentiels ? Le cas le plus classique est celui où on veut utiliser la fonction **llSetPrimitiveParams** qui impose une rotation globale alors que l'on ne

dispose que de valeurs locales. C'est encore davantage le cas maintenant que nous disposons de la nouvelle fonction **llSetLinkPrimitiveParamsFast** qui évite la temporisation de 0,2s ! C'est ce que nous allons voir dans le chapitre suivant.

Mais méfiez-vous de la fonction **llSetRot,** elle ne fixe pas directement la rotation globale mais la rotation locale + la rotation du prim racine. On peut d'ailleurs se questionner sur l'utilité d'une telle fonction ! D'après ce que nous savons maintenant de la fonction **llSetRot** on peut en déduire une fonction pour la fixation de la rotation locale sans utiliser **llSetLocalRot** :

```
SetLocalRot(rotation localrot) {
    llSetRot(localrot / llGetRootRotation());}
```

On peut aussi se demander comment fixer la rotation globale d'une prim lié. En suivant la même logique on en arrive à cette fonction :

```
SetGlobalRot(rotation globalrot) {
    llSetRot(globalrot / llGetRootRotation() / llGetRootRotation());}
```

On peut aussi utiliser la fonction **llSetLocalRot** pour obtenir le même résultat :

```
SetGlobalRot(rotation globalrot) {
    llSetLocalRot(globalrot / llGetRootRotation());}
```

Nous avons maintenant tous les éléments en main pour avancer dans les rotations des prims liés.

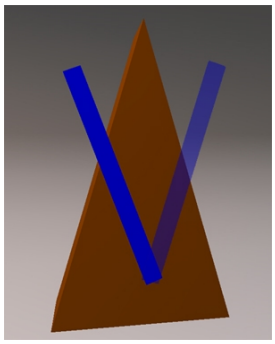
7.2 Un métronome

Vous désirez créer un métronome mécanique. Donc un socle et un balancier qui bat la mesure. Pour faciliter les déplacements de l'objet vous désirez avoir le tout lié. Le balancier sera enfant dans cette liaison. Vous devez donc écrire un script pour le balancement. En lui-même le mouvement du balancier est une rotation excentrée comme celle vue précédemment, la seule différence réside dans le fait qu'il s'agit d'un objet lié. Commençons par écrire un script comme s'il s'agissait d'un objet libre :

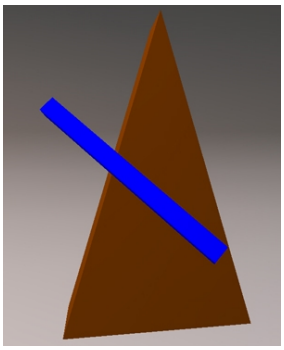
```
vector hauteurBalancier;
integer posGauche;
animation(float angle) {
        vector posBalancier = llGetPos();
        rotation rot = llAxisAngle2Rot(llRot2Left(llGetRot()), DEG_TO_RAD * angle);
        vector vecteur = (hauteurBalancier / 2.0) * llGetRot();
        vector posAxe = posBalancier - vecteur;
        vector rotVecteur = vecteur * rot;
        vector nouvellePosition = posAxe + rotVecteur;
        llSetPrimitiveParams([PRIM_ROTATION, llGetRot() * rot, PRIM_POSITION, nouvellePosition]);}
default {
        state_entry() {
                vector dimensions = llGetScale();
                hauteurBalancier = <.0, .0, dimensions.z>;
                state stop;}
}
```

```
state stop {
	touch_start(integer total_number) {
		state marche;}
}
state marche {
	state_entry() {
		posGauche = TRUE;
		animation(-30);
		llSetTimerEvent(.5);}
	touch_start(integer total_number) {
		state stop;}
	timer() {
		if(posGauche) animation(60.0);
		else animation(-60);
		posGauche = !posGauche;}
	state_exit() {
		llSetTimerEvent(.0);
		if(posGauche) animation(30.0);
		else animation(-30);}
}
```

Je précise pour le build que la rotation se fait sur l'axe Y. Je ne fais aucun commentaire sur ce code, il est calqué sur celui de la porte rotative vu au chapitre précédent. Faisons un essai de ce script. Nous constatons que tout se passe bien :



Maintenant lions le balancier au socle…



Cette fois rien ne va plus et le balancier fait n'importe quoi… Le centre de rotation n'est plus correct et la rotation assez étrange.

En ce qui concerne le centre de rotation nous savons que lorsqu'on positionne une prim enfant avec la fonction **llSetPos** (et donc aussi par l'intermédiaire de **llSetPrimitiveParams**) ce positionnement se fait relativement à la position du parent. Autrement dit on ne raisonne plus en coordonnées globales mais relatives au root. Il nous faut donc connaître la position de l'enfant par rapport au parent, pour cela il faut utiliser la fonction **llGetLocalPos** au lieu de **llGetPos**. Il faut donc modifier cette ligne :

```
vector posBalancier = llGetLocalPos();
```

Nous obtenons ainsi la position du balancier par rapport au root du support. De la même façon pour les rotations il nous faut raisonner en valeurs locales avec la fonction **llGetLocalRot** et modifier la ligne suivante :

```
vector vecteur = (hauteurBalancier / 2.0) * llGetLocalRot();
```

La rotation du vecteur elle aussi doit s'adapter aux rotations locales. Pour mémoire la fonction **llRot2Left** donne l'axe Y pour la rotation passée en paramètre :

```
rotation rot = llAxisAngle2Rot(llRot2Left(llGetLocalRot()), DEG_TO_RAD * angle);
```

Pour donner une rotation locale à un prim lié on a la fonction **llSetLocalRot** et on pourrait s'en sortir dans le script en déterminant la rotation avec **llGetLocalRot() * rot** mais nous utilisons la fonction **llSetPrimitiveParams** pour coordonner les mouvements et cette fonction ne permet pas une rotation locale. Il nous faut donc raisonner en coordonnées globales et rapporter la rotation locale à une rotation globale. On peut connaître la rotation globale du root avec la fonction **llGetRootRotation**. Il suffit donc de tenir compte de cette rotation du root pour obtenir les valeurs globales de rotation pour le balancier.

Pour mémoire l'opérateur « / » appliqué aux rotations n'a rien à voir avec une division. C'est juste l'inverse de ce que vous obtenez avec « * », autrement dit une rotation en sens inverse, comme nous l'avons vu au point 4.3. Notre fonction animation devient donc finalement :

```
animation(float angle) {
	vector posBalancier = llGetLocalPos();
	rotation rot = llAxisAngle2Rot(llRot2Left(llGetLocalRot()), DEG_TO_RAD * angle);
	vector vecteur = (hauteurBalancier / 2.0) * llGetLocalRot();
	vector posAxe = posBalancier - vecteur;
	vector rotVecteur = vecteur * rot;
	vector nouvellePosition = posAxe + rotVecteur;
	llSetPrimitiveParams([PRIM_ROTATION, llGetLocalRot() * rot / llGetRootRotation(),
PRIM_POSITION, nouvellePosition]);}
```

Vous pouvez changer la rotation du support ainsi que la rotation relative du balancier par rapport au support, cela n'influe pas sur le mouvement du balancier.

Pour en finir avec ce métronomoe on va faire une version avec mouvement fluide en utilisant la nouvelle fonction **llSetLinkPrimitiveParamsFast** :

```
vector hauteurBalancier;
integer posGauche;
```

```
animation(float angle) {
	vector posBalancier = llGetLocalPos();
	rotation rot = llAxisAngle2Rot(llRot2Left(llGetLocalRot()), DEG_TO_RAD * angle);
	vector vecteur = (hauteurBalancier / 2.0) * llGetLocalRot();
	vector posAxe = posBalancier - vecteur;
	vector rotVecteur = vecteur * rot;
	vector nouvellePosition = posAxe + rotVecteur;
	llSetLinkPrimitiveParamsFast(2, [PRIM_ROTATION, llGetLocalRot() * rot / llGetRootRotation(),
	PRIM_POSITION, nouvellePosition]);}
set_animation(integer angle) {
	integer i;
	if(angle > 0) {
		for(; i <= angle; ++i) {
			animation((float)1);
			llSleep(.02);}
	}
	else {
		for(; i >= angle; --i) {
			animation(-(float)1);
			llSleep(.02);}
	}
}
default {
	state_entry() {
		vector dimensions = llGetScale();
		hauteurBalancier = <.0, .0, dimensions.z>;
		state stop;}
}
state stop {
	touch_start(integer total_number) {
		state marche;}
}
state marche {
	state_entry() {
		set_animation(-30);
		llSetTimerEvent(1.2);
		posGauche = FALSE;
		set_animation(60);}
	touch_start(integer total_number) {
		llSetTimerEvent(.0);
		if(posGauche) set_animation(30);
		else set_animation(-30);
		state stop;}
	timer() {
		if(posGauche) set_animation(60);
		else set_animation(-60);
		posGauche = !posGauche;}
}
```

7.3 Une planète

Reprenons maintenant notre exemple de planète vue au point 6.1 et lions la planète au soleil... que se passe-t-il ? Pour la planète il est facile de déterminer la position du soleil si celui-ci est le root, il suffit d'utiliser la fonction **llGetRootPosition**. Mais comme nous allons raisonner uniquement en coordonnées locales cette information ne nous est même pas utile, le root étant l'origine des axes il se situe au point <0,0,0> pour l'enfant. Du coup le vecteur de rotation est simplement la position de la planète et le script devient très simple :

```
animation(float angle) {
        rotation rotPourVecteur = llAxisAngle2Rot(llRot2Up(llGetLocalRot()), DEG_TO_RAD * angle);
        llSetLinkPrimitiveParamsFast(2, [PRIM_POSITION, llGetLocalPos() * rotPourVecteur]);}
default {
        state_entry(){llSetTimerEvent(.04);}
        timer() {animation(1.0);}
}
```

Allons un peu plus loin maintenant et admettons qu'il s'agit de la lune. Vous savez que celle-ci présente toujours à la Terre la même face parce que sa durée de révolution sur elle-même est égale à sa durée de révolution sidérale. Autrement dit il faut appliquer à la lune une rotation venant compenser celle de la rotation par rapport à la Terre. Voici le script correspondant

```
animation(float angle) {
        rotation rotPourVecteur = llAxisAngle2Rot(llRot2Up(llGetLocalRot()), DEG_TO_RAD * angle);
        vector nouvellePosition = llGetLocalPos() * rotPourVecteur;
        rotation rotPlanete =   llEuler2Rot(<.0 , .0, DEG_TO_RAD * angle>) * llGetLocalRot() /
llGetRootRotation();
        llSetLinkPrimitiveParamsFast(2, [PRIM_ROTATION, rotPlanete,
                                         PRIM_POSITION, nouvellePosition]);}

default {
        state_entry(){llSetTimerEvent(.04);}
        timer() {animation(1.0);}
}
```

8. Généralisations

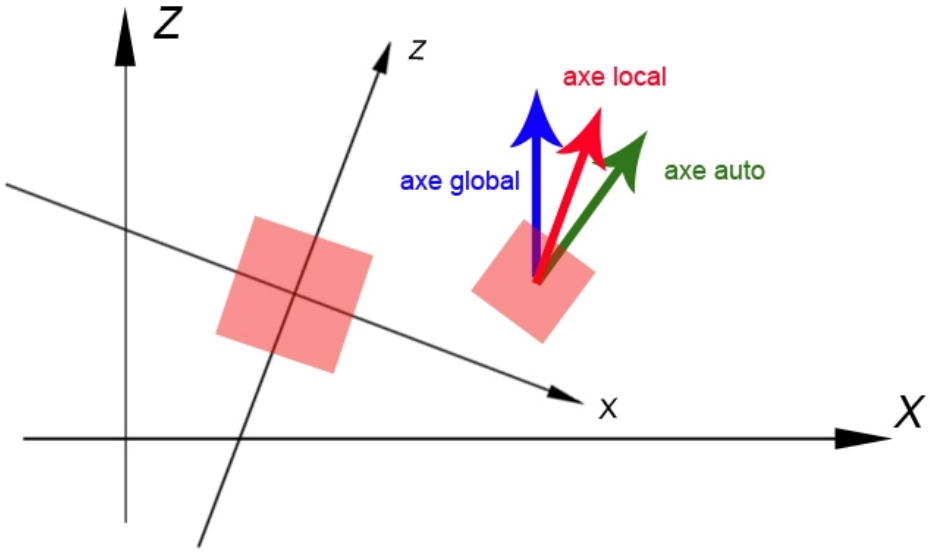
LSL propose peu de fonctions pour effectuer des rotations, essentiellement **llSetRot** (ou **llSetPrimitiveParams**) et **llRotLookAt** pour les rotations globales et **llSetLocalRot** pour les rotations locales. Il est finalement rare d'avoir à fixer une rotation de façon absolue, le plus souvent on désire effectuer une rotation relative, c'est-à-dire à partir d'une rotation actuelle. LSL ne propose pas de fonction pour les rotations relatives, il faut écrire un peu de code comme je l'ai fait pour les exemples des chapitres précédents. Mais pourquoi ne pas se faire une bibliothèque de fonctions toutes prêtes ? Il serait pratique d'avoir des fonctions du genre **SetRot** et **SetRotAngle** qui proposent comme paramètres l'angle de rotation désiré en degrés, la référence pour les coordonnées, et l'axe autour duquel doit s'effectuer cette rotation. Comme les rotations s'effectuent habituellement autour des trois axes on peut définir trois constantes :

```
vector AXE_X      =     <1.0, .0, .0>;
vector AXE_Y      =     <.0, 1.0, .0>;
vector AXE_Z      =     <.0, .0, 1.0>;
```

En ce qui concerne les références on en a trois : globale, locale (uniquement pour les prims liés), et auto (les axes propres du prim), donc aussi trois constantes :

```
integer REF_GLOBAL     =     0;
integer REF_LOCAL      =     1;
integer REF_AUTO       =     2;
```

Voici une illustration pour visualiser ces axes :



8.1 Rotations absolues

Même si **LSL** propose des fonctions pour les rotations absolues il est toujours pénible de convertir les angles en radians et de les transformer en type **rotation**. D'autre part on hésite des fois entre rotation globale et locale. Il serait plus simple d'avoir une seule fonction **SetRot** qui regroupe tous les cas. Elle pourrait en plus prendre en compte un axe de rotation quelconque et pas forcément X, Y ou Z. Voici le code correspondant :

```
SetRot(float angle, integer ref, vector axe) {
	rotation rot = llAxisAngle2Rot(axe, angle * DEG_TO_RAD);
	if(ref == REF_GLOBAL) {
		if(llGetLinkNumber() > 1) llSetRot(rot / llGetRootRotation() / llGetRootRotation());
		else llSetRot(rot);}
	else if(ref == REF_LOCAL)
		llSetLocalRot(rot);}
```

La rotation est déterminée simplement avec la fonction **llAxisAngle2Rot** à partir de l'axe et de l'angle passés en paramètres. Si la référence est globale il nous faut savoir si nous avons affaire à une prim enfant, ce qui est déterminé à partir du numéro de liaison, en effet on sait qu'un prim isolé retourne un numéro 0, un root un numéro 1 et les prims enfants un numéro à partir de 2. Dans le cas d'un prim enfant il nous faut tenir compte des particularités de la fonction **llSetRot**, comme nous l'avons vu au point 7.1.2. Il serait possible également d'utiliser la fonction **llSetLocalRot** comme nous l'avons vu au même point. Dans le cas d'une prim isolé ou racine on applique directement la fonction **llSetRot**. Dans le cas d'une référence locale il ne peut s'agir que d'un prim enfant et on utilise directement la fonction **llSetLocalRot**.

Pour utiliser cette fonction c'est tout simple : vous déterminez l'angle de rotation désiré en degrés, la référence (globale ou locale) et l'axe de rotation. Vous voulez par exemple un angle de rotation de 30° pour une prim lié sur l'axe Z local :

```
SetRot(30.0, REF_LOCAL, AXE_Z);
```

Ou alors une rotation de 45° pour un prim libre sur l'axe X global :

```
SetRot(45.0, REF_GLOBAL, AXE_X);
```

Ou alors une rotation de -60° pour un prim libre sur l'axe <1.0,1.0,.0> en référence globale :

```
SetRot(-60.0, REF_GLOBAL, <1.0,1.0,.0>);
```

Dans le cas où on désire des appels fréquents il vaut mieux utiliser la fonction **llSetLinkPrimitiveParamsFast** :

```
SetRotFast(float angle, integer ref, vector axe) {
	rotation rot = llAxisAngle2Rot(axe, angle * DEG_TO_RAD);
	integer num_link = llGetLinkNumber();
	if(ref == REF_GLOBAL) {
		if(num_link > 1)
			llSetLinkPrimitiveParamsFast(num_link, [PRIM_ROTATION, rot /
llGetRootRotation() / llGetRootRotation()]);
		else llSetLinkPrimitiveParamsFast(num_link, [PRIM_ROTATION, rot]);
	}
```

```
	else if(ref == REF_LOCAL)
		llSetLinkPrimitiveParamsFast(num_link, [PRIM_ROTATION, rot / llGetRootRotation()]);
}
```

Le raisonnement est le même mais on joue avec le numéro de liaison pour la commande.

8.2 Rotations relatives

Envisageons maintenant le cas d'une rotation à partir de la rotation actuelle, donc une rotation relative, ce qui est en général le plus utilisé. Voici le code de cette fonction :

```
SetRotAngle(float angle, integer ref, vector axe) {
	integer enfant = (llGetLinkNumber() > 1);
	rotation rot = llAxisAngle2Rot(axe, angle * DEG_TO_RAD);
	if(ref == REF_GLOBAL) {
		if(enfant) llSetRot(rot * llGetRot() / llGetRootRotation() / llGetRootRotation());
		else llSetRot(llGetRot() * rot);}
	else if(ref == REF_LOCAL)
		llSetLocalRot(llGetLocalRot() * rot);
	else if(ref == REF_AUTO) {
		if(enfant) llSetLocalRot(rot * llGetLocalRot());
		else llSetRot(rot * llGetRot());}
}
```

Une première variable, **enfant**, est utilisée pour savoir si on a affaire à un prim enfant. La rotation est déterminée simplement avec la fonction **llAxisAngle2Rot** à partir de l'axe et de l'angle passés en paramètres. Si la référence est globale et que nous sommes en présence d'un prim enfant il nous faut tenir compte des particularités de la fonction **llSetRot**, comme nous l'avons vu au point 7.1.2. Il serait possible également d'utiliser la fonction **llSetLocalRot** comme nous l'avons vu au même point. Dans le cas d'un prim isolé ou racine on applique directement la fonction **llSetRot**. Dans le cas d'une référence locale il ne peut s'agir que d'un prim enfant et on utilise directement la fonction **llSetLocalRot**. Si la référence est le prim lui-même (« AUTO ») pour un prim enfant la fonction **llSetLocalRot** nous donne directement la rotation alors qu'il faut utiliser **llSetRot** dans le cas d'un prim isolé ou d'un root. L'utilisation est la même que pour la fonction précédente.

On peut aussi faire une version pour les appels fréquents :

```
SetRotAngleFast(float angle, integer ref, vector axe) {
	integer num_link = llGetLinkNumber();
	integer enfant = (num_link > 1);
	rotation rot = llAxisAngle2Rot(axe, angle * DEG_TO_RAD);
	if(ref == REF_GLOBAL) {
		if(enfant)
			llSetLinkPrimitiveParamsFast(num_link, [PRIM_ROTATION, rot * llGetRot() /
llGetRootRotation() / llGetRootRotation()]);
		else
			llSetLinkPrimitiveParamsFast(num_link, [PRIM_ROTATION, llGetRot() * rot]);}
	else if(ref == REF_LOCAL)
		llSetLinkPrimitiveParamsFast(num_link, [PRIM_ROTATION, llGetLocalRot() * rot /
llGetRootRotation()]);
	else if(ref == REF_AUTO) {
```

```
		if(enfant)
			llSetLinkPrimitiveParamsFast(num_link, [PRIM_ROTATION, rot / llGetRootRotation()
* llGetLocalRot()]);
		else
			llSetLinkPrimitiveParamsFast(num_link, [PRIM_ROTATION, rot * llGetRot()]);}
}
```

8.3 Rotations excentrées

Pour les rotations excentrées il nous faut plus de paramètres, en effet il nous faut définir le point par lequel passe l'axe de rotation, donc encore un paramètre de type **vector**. En fait une rotation excentrée n'est pas vraiment une rotation pour l'objet mais un repositionnement. Ce qui tourne c'est le vecteur. En ce qui concerne le positionnement d'un objet il n'y a qu'une fonction : **llSetPos**. Celle-ci est relativement limitée en distance puisque sa portée est de 10 mètres. Je vais partir du principe que nous n'avons pas besoin d'une plus grande distance (si vous avez besoin de plus le **wiki** donne la formule des petits bonds).

Un autre inconvénient concerne les prims liés, pour ceux-ci la fonction **llSetPos** fonctionne en coordonnées relatives au **root**. Il n'existe pas de fonction permettant de positionner directement un prim lié en coordonnées globales. Nous avons vu au point 7.1.1 les changements de référentiel pour les translations. Mais le plus simple en cas de prim lié est de raisonner en coordonnées locales. Nous allons donc adopter comme attitude de considérer axe et centre en coordonnées globales pour les prims libres et racines et en coordonnées locales pour les prims liés.

Nous allons distinguer deux cas de figure : une rotation simple et une rotation combinée dans laquelle le mobile tourne aussi autour de lui-même.

8.3.1 Rotation simple

Dans ce type de rotation le mobile tourne autour d'un point excentré mais ne subit aucune rotation sur lui-même. Il nous faut distinguer le cas du prim libre ou racine et du prim lié. Il nous faut aussi tenir compte du référentiel pour l'axe et pour le centre. En ce qui concerne l'axe de rotation celui-ci peut être en coordonnée globales, locales (uniquement pour prim enfant) ou « auto ». En ce qui concerne le centre de rotation celui-ci peut aussi être en coordonnée globales ou locales, mais il faut tenir compte aussi d'un possible positionnement par rapport au prim lui-même (« AUTO »), ce qui arrive fréquemment comme par exemple avec une porte rotative.

Voici la fonction de rotation excentrée simple :

```
vector AXE_X		=	<1.0, .0, .0>;
vector AXE_Y		=	<.0, 1.0, .0>;
vector AXE_Z		=	<.0, .0, 1.0>;

integer REF_GLOBAL	=	0;
integer REF_LOCAL	=	1;
integer REF_AUTO	=	2;

vector Global2Local(vector position) {
	return (position - llGetRootPosition()) / llGetRootRotation();}

vector GetCentreAuto(vector centre) {
	if(llGetLinkNumber() > 1)
		return llGetLocalPos() + centre * llGetLocalRot();
	else
```

```
		return llGetPos() + centre * llGetRot();}

SetRotExc(float angle, integer refAxe, vector axe, integer refCentre, vector centre) {
	integer num_link = llGetLinkNumber();
	integer enfant = (num_link > 1);
	if(!enfant && (refAxe == REF_LOCAL || refCentre == REF_LOCAL)) {
		llOwnerSay("Incompatibilite entre REF_LOCAL et prim libre");
		return;}
	if(refCentre == REF_AUTO) {
		llOwnerSay("REF_AUTO impossible pour centre, utilisez la fonction GetCentreAuto");
		return;}
	if(refAxe == REF_AUTO) {
		if(enfant)
			axe *= llGetLocalRot();
		else
			axe *= llGetRot();}
	else if(refAxe == REF_GLOBAL && enfant)
		axe /= llGetRootRotation();
	if(refCentre == REF_GLOBAL && enfant)
		centre = Global2Local(centre);
	vector vecteur;
	if(enfant)
		vecteur = llGetLocalPos() - centre;
	else
		vecteur = llGetPos() - centre;
	rotation rotPourVecteur = llAxisAngle2Rot(axe, angle * DEG_TO_RAD);
	vector rotVecteur = vecteur * rotPourVecteur;
	vector nouvellePosition = centre + rotVecteur;
	llSetLinkPrimitiveParamsFast(num_link, [PRIM_POSITION, rot / nouvellePosition]);
}
```

Voyons un peu les paramètres de la fonction **SetRotExc** :

Paramètre	Nom	Type	Fonction
1	angle	**float**	Angle de rotation en degrés
2	refAxe	**Integer**	Référence pour l'axe : REF_GLOBAL, REF_LOCAL ou REF_AUTO
3	axe	**vector**	Axe de rotation
4	refCentre	**Integer**	Référence pour le centre : REF_GLOBAL, REF_LOCAL
5	centre	**vector**	Centre de rotation

La fonction **SetRotExc** appelle quelques commentaires. Dans un premier temps on détermine si on a affaire à un prim enfant en mémorisant cette information dans la variable **enfant**. On procède ensuite à quelques tests pour vérifier des cohérences au niveau des paramètres.

Suit la gestion de l'axe de rotation. Si la référence est « AUTO », donc le prim lui-même, on distingue le cas du prim enfant, dans ce cas il faut ramener l'axe en coordonnées locales avec la fonction **llGetLocalRot**, et dans le cas contraire on ramène l'axe en coordonnées globales avec la fonction **llGetRot**. Si la référence est globale et qu'on a affaire à un prim enfant il faut aussi ramener l'axe en coordonnées locale, c'est cette fois la fonction **llGetRootRotation** qui réalise ce changement de référentiel. Si la référence est locale c'est qu'il s'agit d'un prim enfant et l'axe est déjà dans le bon référentiel.

Suit la gestion du centre de rotation. Le seul cas particulier est celui du prim enfant avec une référence globale. Dans ce cas il faut ramener le centre en coordonnées locales avec la fonction **Global2Local**. Par contre si le centre est en référence globale pour un prim libre ou racine ou en référence locale pour un prim enfant rien de particulier n'est à prévoir.

Un cas particulier pour la détermination du centre est celui qui correspond à un positionnement relatif au prim concerné. Dans ce cas il est évident qu'il faut ramener ce positionnement en coordonnées globables pour un prim libre ou racine et en coordonnées locales pour un prim enfant. C'est la fonction **GetCentreAuto** qui est chargée de ce changement. Cette fonction teste la présence d'un prim enfant et ajoute au vecteur orienté par la rotation locale la position locale avec **llGetLocalPos**, dans le cas contraire elle ajoute la position globale avec **llGetPos** au vecteur orienté par la rotation globale. Il suffit ensuite d'utiliser la fonction **SetRotExc**.

Suit la détermination du vecteur. Dans le cas d'un prim enfant la référence est locale, donc on détermine le centre du prim avec **llGetLocalPos** et on soustrait la position du centre qui est elle aussi en coordonnées locales. Dans le cas d'un prim libre ou racine la position du centre du prim est déterminée avec **llGetPos**, on soustrait la position du centre qui est elle aussi en coordonnées globales.

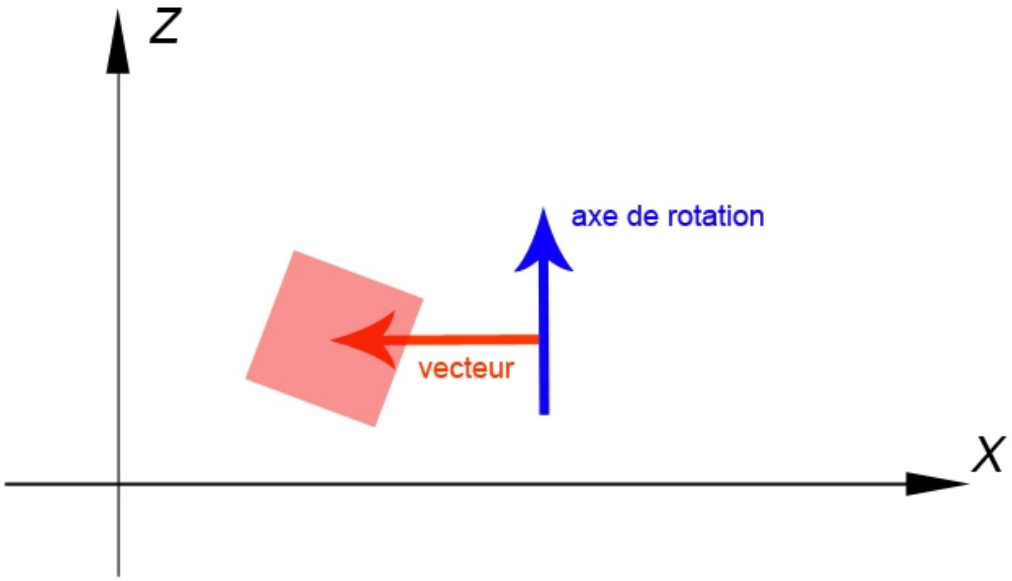
Il ne reste plus qu'à définir la rotation du vecteur avec la fonction **llAxisAngle2Rot** comme nous l'avons déjà vu précédemment, trouver la nouvelle position et l'appliquer avec la fonction **llSetPos** qui, comme nous le savons, fonctionne en coordonnées globales pour un prim libre ou racine et en coordonnées locales pour un prim enfant.

Nous allons analyser de façon exhaustive les cas d'utilisation de cet arsenal…

8.3.1.1 Axe global et centre global (prim libre, racine ou enfant)

```
vector position;
default {
        state_entry() {
                position = llGetPos() + <1.0,.0,.0>;}
        touch_start(integer total_number) {
                state marche;}
}
state marche {
        state_entry() {
                llSetTimerEvent(.2);}
        touch_start(integer total_number) {
                state default;}
        timer() {
                SetRotExc(5.0, REF_GLOBAL, AXE_Z, REF_GLOBAL, position);}
        state_exit() {
                llSetTimerEvent(.0);}
}
```

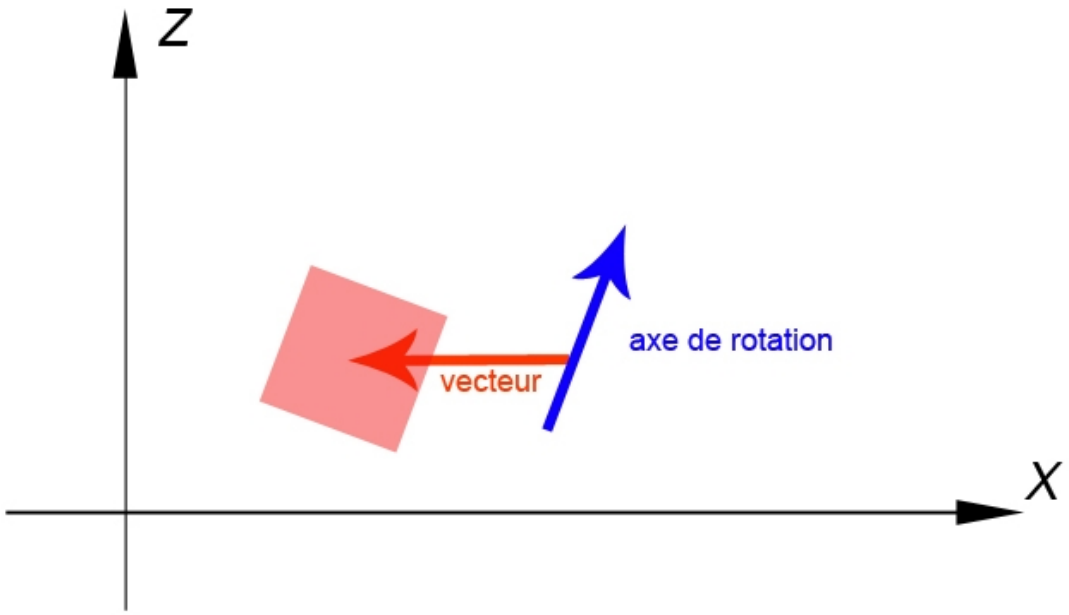
Cette trame de script va nous servir pour tous les exemples. Lorsque le script démarre on définit la position du centre que l'on mémorise dans la variable **position**. Dans ce premier cas on prend un point situé en coordonnées globales à 1 mètres sur l'axe X. Après un clic on passe dans l'état **marche**, un timer est initialisé avec un délai de 0,2 seconde. Lorsque l'événement **timer** se déclenche on appelle la fonction **SetRotExc**. L'angle de base est fixé à 5 degrés. On choisit l'axe Z en référence globale et le centre défini dans la variable **position** en référence globale également. Vous pouvez vérifier que la rotation obtenue ne dépend pas des rotations du prim en donnant à celui-ci une rotation aléatoire.



8.3.1.2 Axe auto et centre global (prim libre, racine ou enfant)

```
vector position;
default {
        state_entry() {
                position = llGetPos() + <1.0,.0,.0>;}
        touch_start(integer total_number) {
                state marche;}
}
state marche {
        state_entry() {
                llSetTimerEvent(.2);}
        touch_start(integer total_number) {
                state default;}
        timer() {
                SetRotExc(5.0, REF_AUTO, AXE_Z, REF_GLOBAL, position);}
        state_exit() {
                llSetTimerEvent(.0);}
}
```

Le code est pratiquement le même que précédemment mis à part la référence « AUTO » pour l'axe de rotation. Cette fois si vous changez la rotation du prim sur l'axe X ou/et Y vous allez constater une inclinaison corrélative de l'axe de rotation puisque maintenant on est aligné sur l'axe Z du prim. Par contre le centre de rotation est toujours le même.



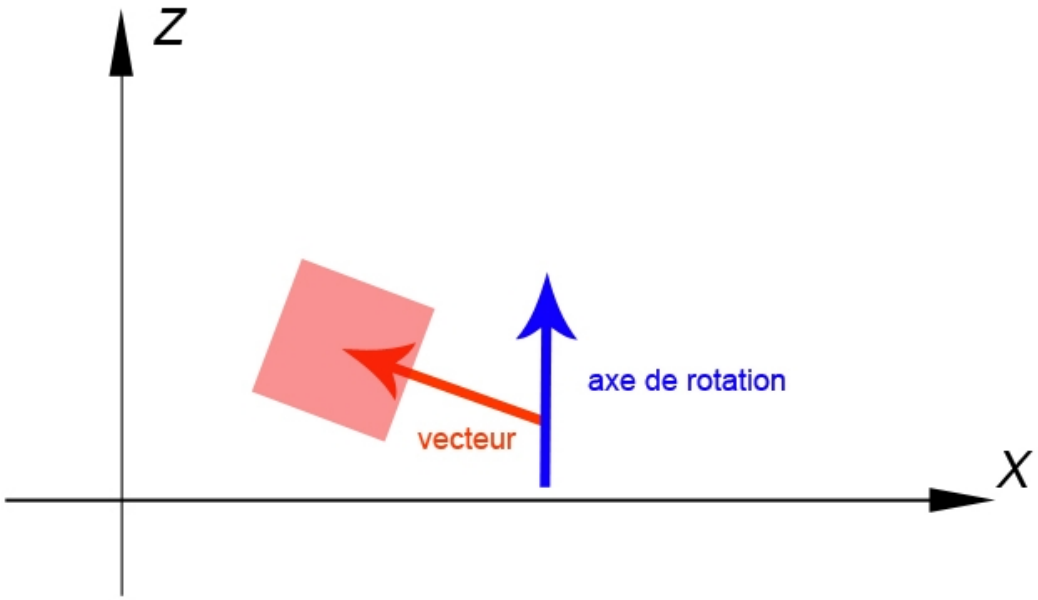
8.3.1.3 Axe global et centre auto (prim libre, racine ou enfant)

```
vector position;
default {
        state_entry() {
                position = GetCentreAuto(<1.0,.0,.0>);}
        touch_start(integer total_number) {
                state marche;}
}
state marche {
        state_entry() {
                llSetTimerEvent(.2);}
        touch_start(integer total_number) {
                state default;}
        timer() {
                SetRotExc(5.0, REF_GLOBAL, AXE_Z, REF_GLOBAL, position);}
        state_exit() {
                llSetTimerEvent(.0);}
}
```

Dans le cas d'un prim enfant il faut passer le centre en coordonnées locales et changer cette ligne :

SetRotExc(5.0, **REF_GLOBAL**, AXE_Z, **REF_LOCAL**, position);}

Le code est encore pratiquement le même que précédemment. On revient à une référence globale pour l'axe. Par contre le centre est positionné par rapport au prim.



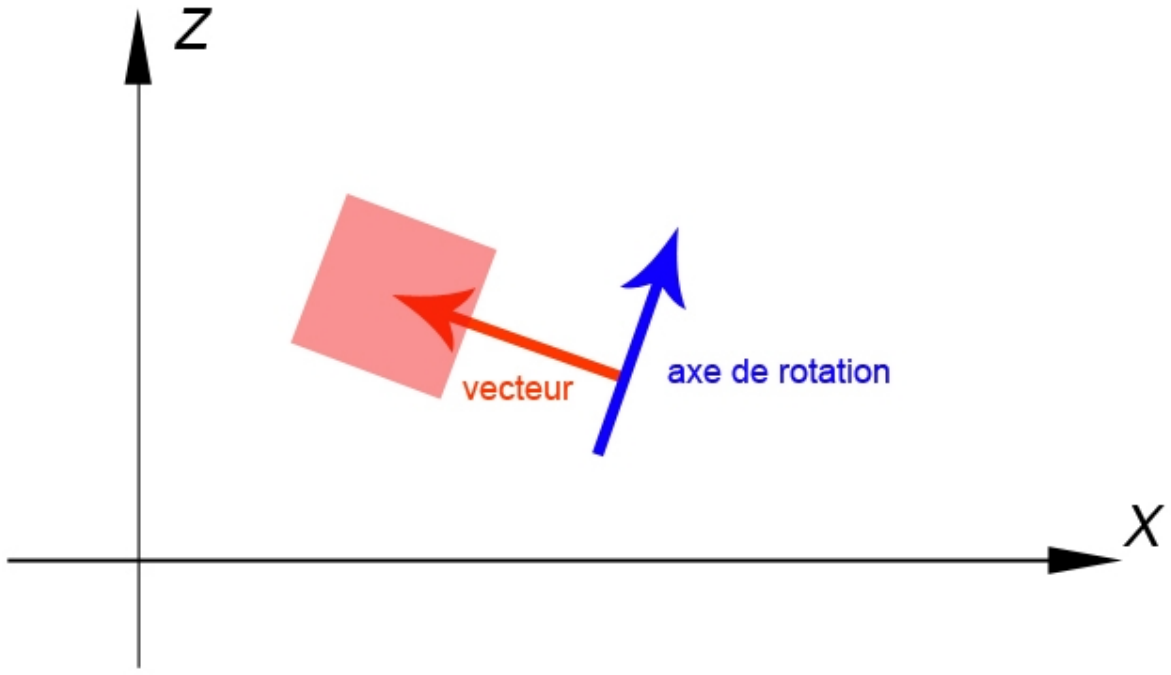
8.3.1.4 Axe auto et centre auto (prim libre, racine ou enfant)

```
vector position;
default {
        state_entry() {
                position = GetCentreAuto(<1.0,.0,.0>);}
        touch_start(integer total_number) {
                state marche;}
}
state marche {
        state_entry() {
                llSetTimerEvent(.2);}
        touch_start(integer total_number) {
                state default;}
        timer() {
                SetRotExc(5.0, REF_AUTO, AXE_Z, REF_GLOBAL, position);}
        state_exit() {
                llSetTimerEvent(.0);}
}
```

Dans le cas d'un prim enfant il faut passer le centre en coordonnées locales et changer cette ligne :

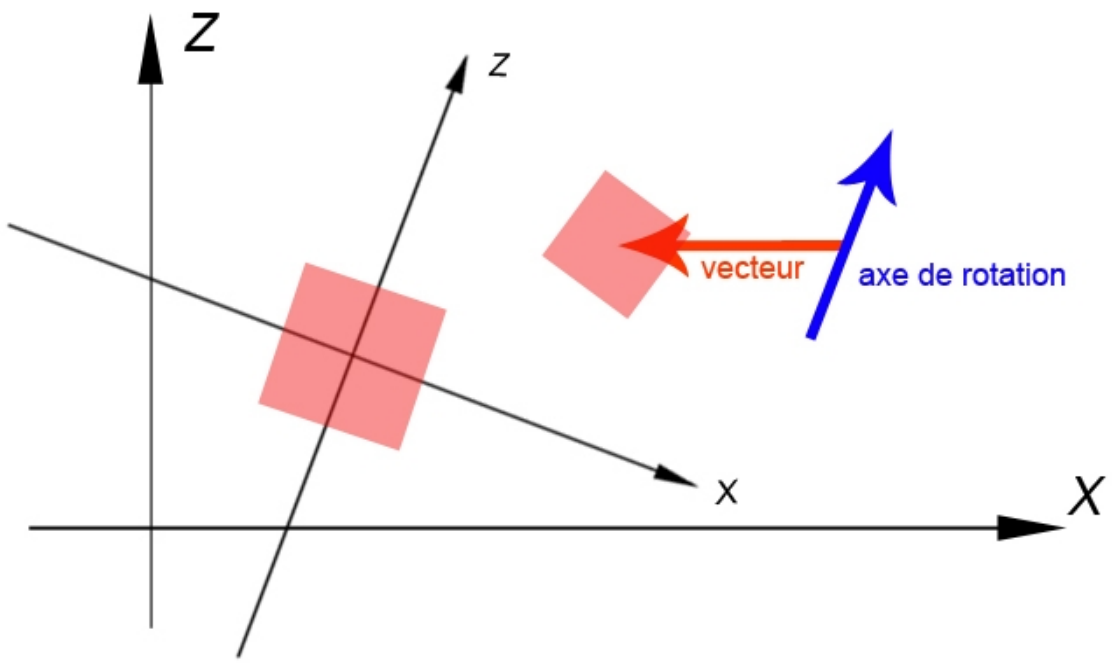
SetRotExc(5.0, **REF_GLOBAL**, AXE_Z, **REF_LOCAL**, position);}

Toutes les références sont « AUTO ».



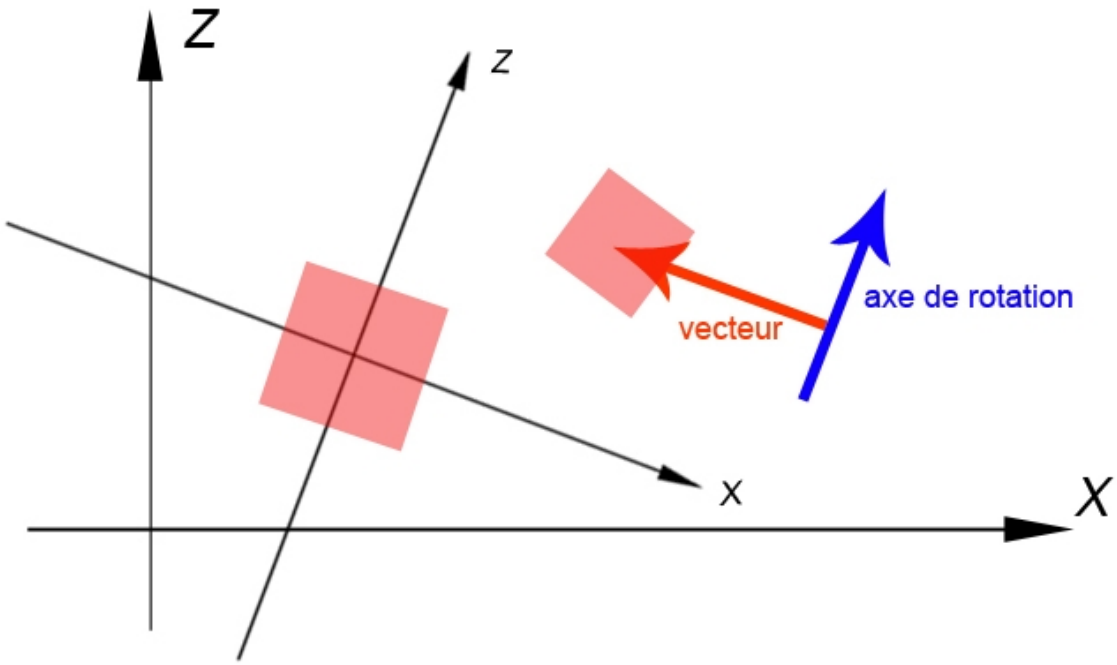
8.3.1.5 Axe local et centre global (prim enfant)

```
vector position;
default {
        state_entry() {
                position = llGetPos() + <1.0,.0,.0>;}
        touch_start(integer total_number) {
                state marche;}
}
state marche {
        state_entry() {
                llSetTimerEvent(.2);}
        touch_start(integer total_number) {
                state default;}
        timer() {
                SetRotExc(5.0, REF_LOCAL, AXE_Z, REF_GLOBAL, position);}
        state_exit() {
                llSetTimerEvent(.0);}
}
```



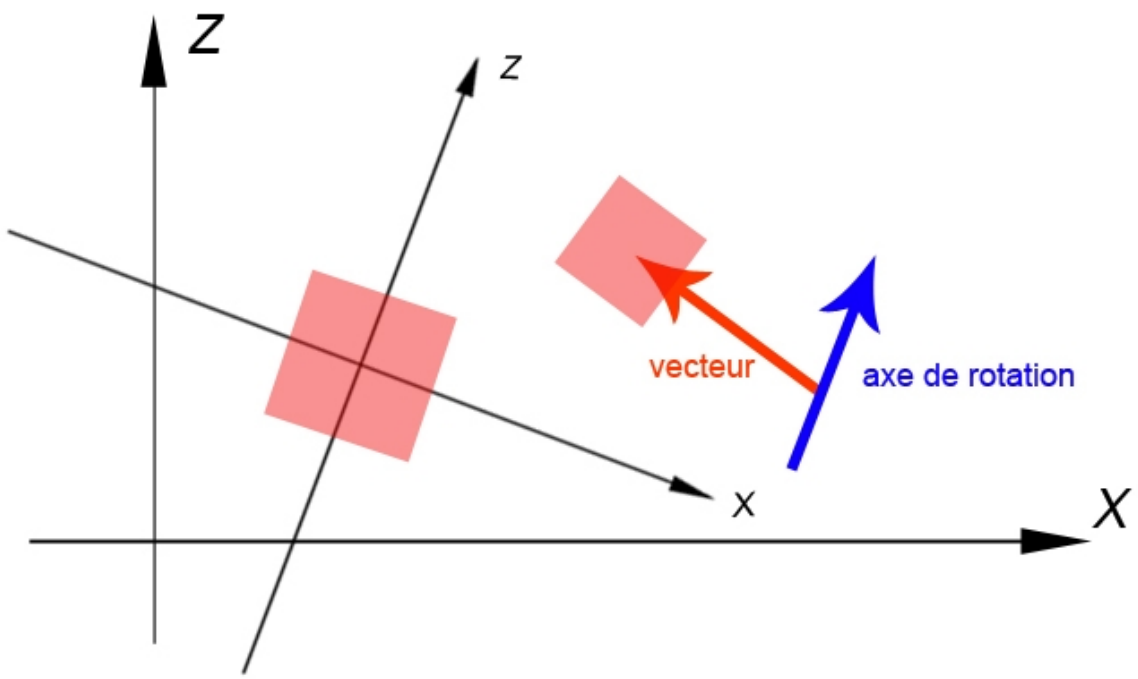
8.3.1.6 Axe local et centre local (prim enfant)

```
vector position;
default {
        state_entry() {
                position = llGetLocalPos() + <1.0,.0,.0> * llGetLocalRot();}
        touch_start(integer total_number) {
                state marche;}
}
state marche {
        state_entry() {
                llSetTimerEvent(.2);}
        touch_start(integer total_number) {
                state default;}
        timer() {
                SetRotExc(5.0, REF_LOCAL, AXE_Z, REF_LOCAL, position);}
        state_exit() {
                llSetTimerEvent(.0);}
}
```



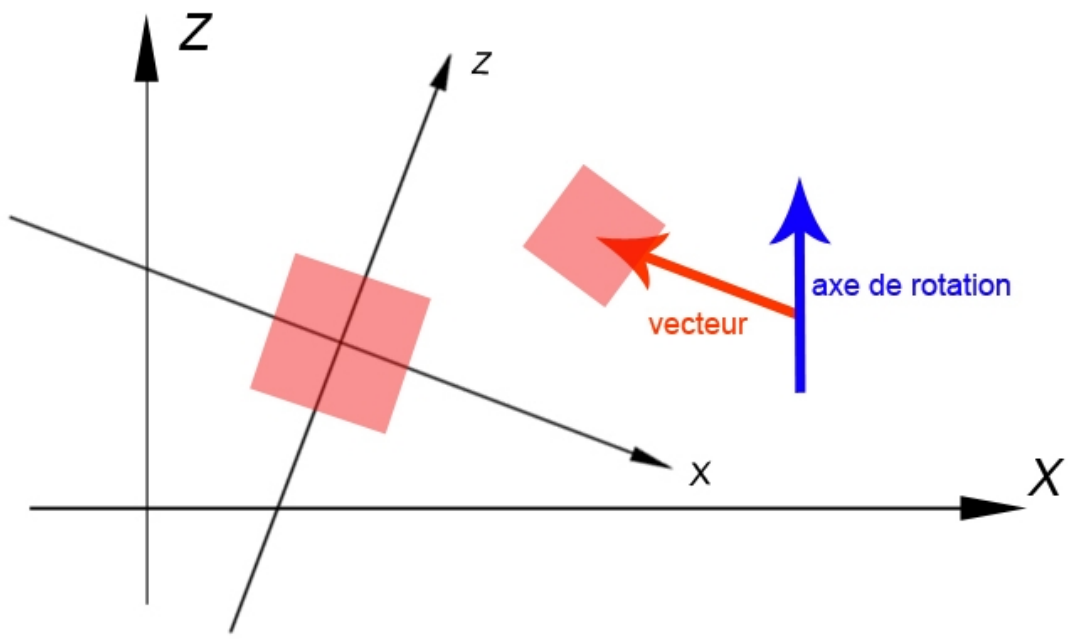
8.3.1.7 Axe local et centre auto (prim enfant)

```
vector position;
default {
        state_entry() {
                position = GetCentreAuto(<1.0,.0,.0>);}
        touch_start(integer total_number) {
                state marche;}
}
state marche {
        state_entry() {
                llSetTimerEvent(.2);}
        touch_start(integer total_number) {
                state default;}
        timer() {
                SetRotExc(5.0, REF_LOCAL, AXE_Z, REF_LOCAL, position);}
        state_exit() {
                llSetTimerEvent(.0);}
}
```



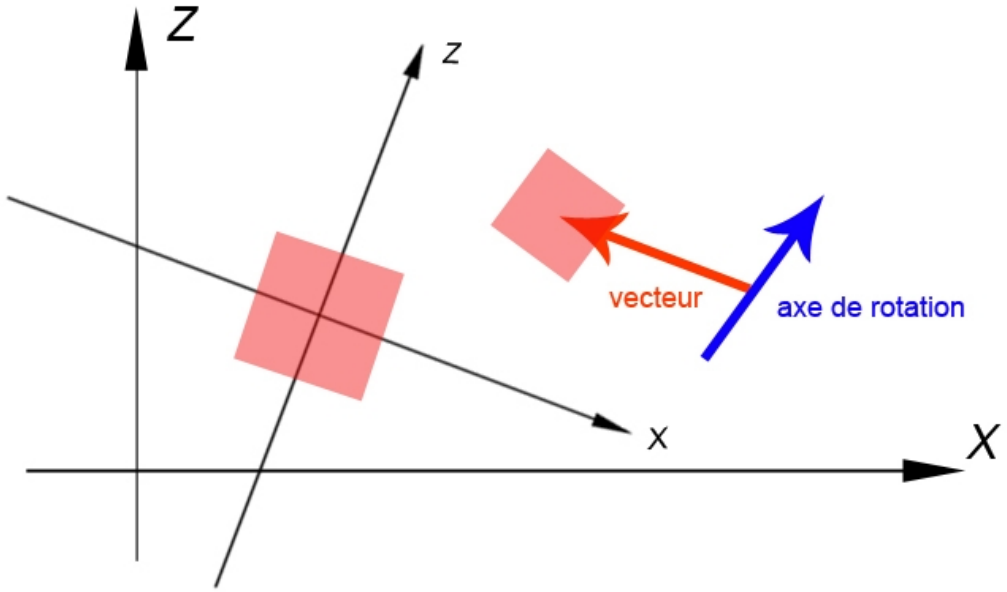
8.3.1.8 Axe global et centre local (prim enfant)

```
vector position;
default {
        state_entry() {
                position = llGetLocalPos() + <1.0,.0,.0> * llGetLocalRot();}
        touch_start(integer total_number) {
                state marche;}
}
state marche {
        state_entry() {
                llSetTimerEvent(.2);}
        touch_start(integer total_number) {
                state default;}
        timer() {
                SetRotExc(5.0, REF_GLOBAL, AXE_Z, REF_LOCAL, position);}
        state_exit() {
                llSetTimerEvent(.0);}
}
```



8.3.1.9 Axe auto et centre local (prim enfant)

```
vector position;
default {
        state_entry() {
                position = llGetLocalPos() + <1.0,.0,.0> * llGetLocalRot();}
        touch_start(integer total_number) {
                state marche;}
}
state marche {
        state_entry() {
                llSetTimerEvent(.2);}
        touch_start(integer total_number) {
                state default;}
        timer() {
                SetRotExc(5.0, REF_AUTO, AXE_Z, REF_LOCAL, position);}
        state_exit() {
                llSetTimerEvent(.0);}
}
```



8.3.2 Rotations combinées

Dans les rotations combinées le mobile tourne aussi sur lui-même d'une valeur équivalente à celle de la rotation excentrée mais en sens contraire. Ce qui fait que le mobile présente toujours la même face en direction du centre de rotation excentrée, à la manière de la Lune par rapport à la Terre. Au niveau de la rotation excentrée tout ce que nous avons vu au chapitre précédent reste évidemment valable. La seule différence va résider dans le fait qu'il faut compenser la rotation au niveau du mobile. Cette compensation doit se faire sur le même axe de rotation que celui de la rotation excentrée, il nous faut donc ramener cet axe en coordonnées locales au mobile en tenant compte de tous les cas de figure. Voici le code de cette fonction :

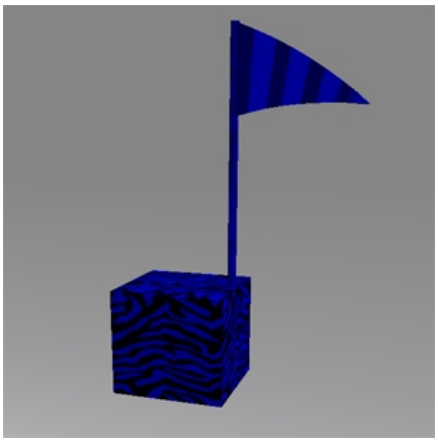
```
vector AXE_X            =       <1.0, .0, .0>;
vector AXE_Y            =       <.0, 1.0, .0>;
vector AXE_Z            =       <.0, .0, 1.0>;

integer REF_GLOBAL      =       0;
integer REF_LOCAL       =       1;
integer REF_AUTO        =       2;

vector Global2Local(vector position) {
        return (position - llGetRootPosition()) / llGetRootRotation();}

vector GetCentreAuto(vector centre) {
        if(llGetLinkNumber() > 1)
                return llGetLocalPos() + centre * llGetLocalRot();
        else
                return llGetPos() + centre * llGetRot();}

SetRotExcComb(float angle, integer refAxe, vector axe, integer refCentre, vector centre) {
        integer num_link = llGetLinkNumber();
        integer enfant = (num_link > 1);
        if(!enfant && (refAxe == REF_LOCAL || refCentre == REF_LOCAL)) {
                llOwnerSay("Incompatibilite entre REF_LOCAL et prim libre");
                return;}
        if(refCentre == REF_AUTO) {
                llOwnerSay("REF_AUTO impossible pour centre, utilisez la fonction GetCentreAuto");
                return;}
        if(refAxe == REF_AUTO) {
                if(enfant)
                        axe *= llGetLocalRot();
                else
                        axe *= llGetRot();}
        else if(refAxe == REF_GLOBAL && enfant)
                axe /= llGetRootRotation();
        if(refCentre == REF_GLOBAL && enfant)
                centre = Global2Local(centre);
        vector vecteur;
        if(enfant)
                vecteur = llGetLocalPos() - centre;
        else
                vecteur = llGetPos() - centre;
        rotation rotPourVecteur = llAxisAngle2Rot(axe, angle * DEG_TO_RAD);
        vector rotVecteur = vecteur * rotPourVecteur;
        vector nouvellePosition = centre + rotVecteur;
        rotation rotMobile;
        if(enfant)
                rotMobile = llGetLocalRot() * rotPourVecteur / llGetRootRotation();
        else
                rotMobile = llGetRot() * rotPourVecteur;
           llSetLinkPrimitiveParamsFast(num_link, [PRIM_ROTATION, rotMobile,
```

```
PRIM_POSITION, nouvellePosition]);}
```

La différence avec la précédente apparaît en gras. On définit une nouvelle rotation pour le mobile mémorisée dans la variable **rotMobile**. Si on a affaire à un prim enfant on ajoute à la rotation du prim (**llGetLocalRot**) la rotation imposée et on ramène le tout en coordonnées globales pour tenir compte de la rotation du prim racine (**llGetRootRotation**). En effet on sait que la fontion **llSetLinkPrimitiveParamsFast** traite les rotations en valeur globales. Dans le cas d'un prim libre ou racine il suffit d'ajouter à la rotation du prim (**llGetRot**) la rotation imposée.

Les cas d'utilisation sont exactement les mêmes que pour la rotation excentrée simple.

Un exemple étant toujours bien plus instructif qu'un long discours je vous propose la rotation d'un drapeau autour de son mat, le tout lié à un objet. Pas question de tricher en décalant un axe ou une texture pour un drapeau flex. Voici ma petite maquette pour faire des tests :



Le but est donc de faire tourner le petit fanion qui est lié à la structure et évidemment le tout doit pouvoir être repositionné et tourné sans que cela influe sur la rotation. Voici le code correspondant :

```
vector position;
default {
	state_entry(){
		vector dim = llGetScale();
		position = GetCentreAuto(<.0, .0, dim.z> / 2.0);
		llSetTimerEvent(.04);}
	timer() {
		SetRotExcComb(2.0, REF_AUTO, AXE_X, REF_LOCAL, position);}
}
```

Avec l'utilisation des fonctions que nous avons mises au point précédemment le code devient très simple comme vous pouvez le constater.

8.4 Les attachements

Lorsqu'un objet est attaché il se passe un certain nombre de modifications au niveau du résultat de certaines fonctions. Faisons un peu le point grâce au wiki :

Script dans le root d'un objet non attaché

Fonction	Action
llGetRot **llGetPrimitiveParams**	Rotation globale de l'objet
llGetLocalRot	
llGetRootRotation	
llSetRot **llSetPrimitiveParams**	
llSetLocalRot	

Script dans un prim lié à un objet non attaché

Fonction	Action
llGetRot **llGetPrimitiveParams**	Rotation globale du prim
llGetLocalRot	Rotation locale du prim
llGetRootRotation	Rotation globale du root
llSetRot **llSetPrimitiveParams**	Rotation globale dépendante de la rotation du root
llSetLocalRot	Rotation locale du prim

- Pour fixer une rotation locale avec **llSetRot** : rotation locale / rotation du root
- Pour fixer une rotation globale avec **llSetRot** : rotation locale / rotation du root / rotation du root ou utiliser **llSetLocalRot** comme ceci : rotation locale * rotation du root

Script dans le root d'un objet attaché

Fonction	Action
llGetRot **llGetPrimitiveParams**	Rotation globale de l'avatar
llGetLocalRot	Rotation de l'attachement par rapport au point d'attachement
llGetRootRotation	Rotation globale de l'avatar
llSetRot **llSetPrimitiveParams**	Rotation par rapport à l'avatar
llSetLocalRot	Rotation globale par rapport à l'avatar

Script dans un prim lié à un objet attaché

Fonction	Action
llGetRot **llGetPrimitiveParams**	Rotation globale de l'avatar * rotation globale du prim
llGetLocalRot	Rotation locale du prim (par rapport au root)
llGetRootRotation	Rotation globale de l'avatar

llSetRot llSetPrimitiveParams	Rotation globale du root * Nouvelle rotation
llSetLocalRot	Rotation locale du prim (par rapport au root)

On peut constater un certain nombre de choses. En particulier un prim lié à un objet attaché ne peut pas connaître la rotation de l'attachement. D'autre part la fonction **llGetRot** donne une valeur fantaisiste pour un prim lié à un objet attaché parce qu'il ne tient pas compte de la rotation de l'attachement.

Les fonctions vues précédemment doivent donc être relativisées pour des objets attachés. Ce n'est d'ailleurs qu'un euphémisme parce qu'elles ne fonctionnent tout simplement pas. Il faudrait compléter ce guide pour les attachements, peut-être pour une prochaine version !