LSL

Guide des machines

par Bestmomo Lagan

Version 1.1

11/12/2010

***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

Un guide sur les machines ? Quelle drôle d'idée ! On n'en voit pas beaucoup dans SL. Cela est sans doute dû au fait qu'il n'est pas forcément aisé au niveau script de mettre en mouvement des primitives et de les synchroniser. D'autant que la séparation entre buildeurs et scripteur est bien souvent étanche. Ce guide est destiné à démystifier ce domaine pour créer des vocations et voir SL se peupler de machines simples ou exotiques. A l'image du guide sur les rotations les chapitres sont progressifs et doivent être étudiés dans l'ordre, sauf pour ceux ou celles qui ont déjà de solides connaissances et peuvent ainsi sauter des étapes. Il est d'ailleurs conseillé de maîtriser un minimum les éléments développés dans le guide sur les rotations avant d'aborder le présent manuel.

Même si le domaine mathématique a été simplifié au maximum il n'en demeure pas moins nécessaire d'avoir quelques connaissances de base en géométrie, trigonométrie, vecteurs… mais rien qui dépasse ce qui est enseigné au Lycée. Il faut aussi prendre le temps de bien assimiler les notions essentielles pour ne pas se perdre dans les développements plus délicats. Les trois premiers chapitres sont d'ailleurs destinés à établir ces bases indispensables avec le traitement de déplacements simples et isolés et des rappels généraux. Les chapitres suivants traitent des machines de plus en plus complexes.

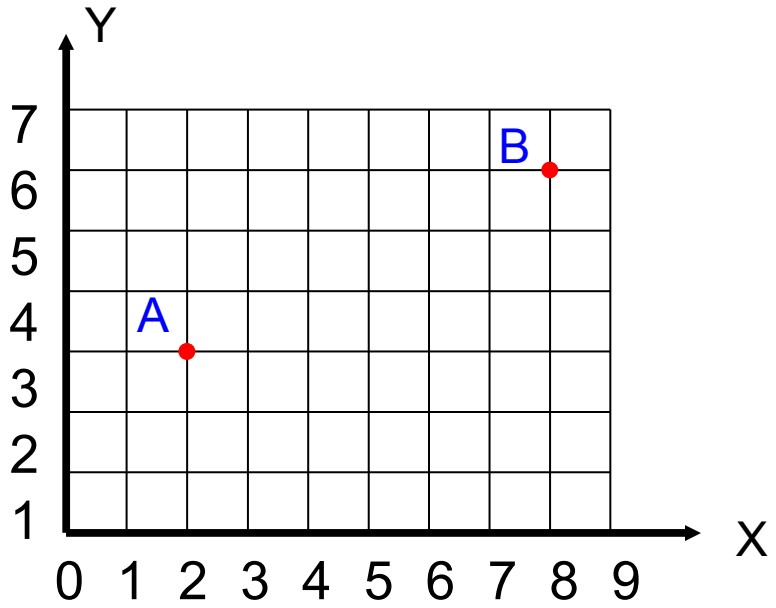
Les scripts présents dans ce manuel ont pour seule vocation une destination didactique. Ils ne sont pas forcément finalisés et exempts de bugs. Il ne sont pas destinés à figurer tels quels dans des réalisations commerciales.

2. Translation

Notre premier projet est tout simple : déplacer un objet en ligne droite, autrement dit lui faire effectuer une translation.

2.1 Système de coordonnées

Pour repérer un objet dans l'espace nous avons besoin de repères. Dans la vie de tous les jours si on vous demande où se trouve un objet vous utilisez forcément des repères, par exemple « sur la table », « derrière l'armoire », « dans le tiroir »… Ce n'est pas très précis mais en général largement suffisant pour la personne qui vous pose la question. Dans le domaine qui nous intéresse il nous faut plus de rigueur et de précision sinon les résultats seront décevants. Pour simplifier nous allons considérer un plan, donc un espace aplati qui ne comporte que 2 dimensions, forcément plus simple que l'espace réel qui lui en possède 3, mais largement suffisant pour comprendre. Pour repérer un point dans cet espace nous allons utiliser un système de coordonnées. Regardez cette figure :



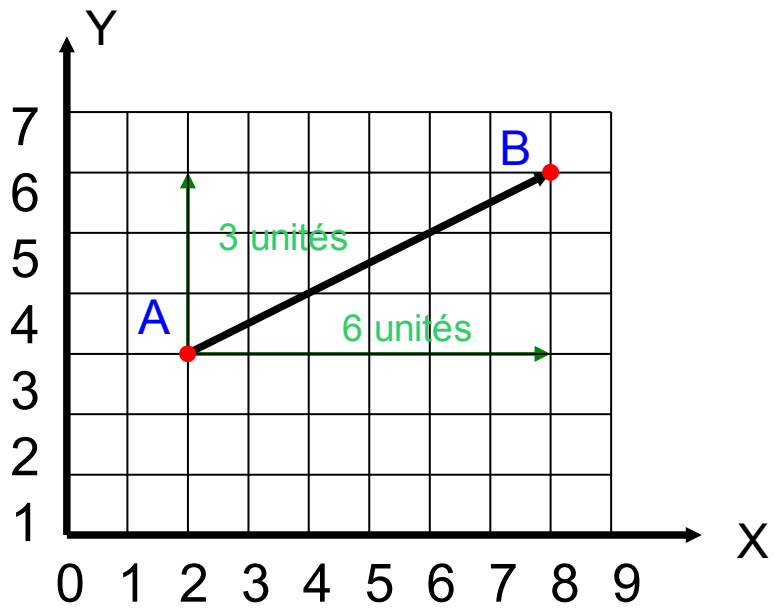
L'espace à 2 dimensions a été découpé en petits carrés égaux. Un axe X horizontal et un axe Y vertical servent de repères de base. Ils sont découpés et numérotés de 0 à 9. De cette façon il est facile de repérer un point à partir de ces axes. Considérez le point A, il se situe sur le 2 au niveau de l'axe X et sur le 3 au niveau de l'axe Y. Ce sont les coordonnées de ce point. Nous avons besoin de deux valeurs et ces deux valeurs sont suffisantes pour déterminer la position du point. De la même façon le point B a pour coordonnées 8 sur l'axe X et 6 sur l'axe Y. Dans un espace à 3 dimensions il nous faudrait 3 valeurs mais le raisonnement est strictement identique.

Il nous faut une représentation de ces valeurs. Pour rester au plus proche du LSL nous allons adopter celle-ci :

Pour le point A : <2, 3>
Pour le point B : <8, 6>

2.2 Les vecteurs

Supposons maintenant qu'un objet se trouve au point A, du moins son centre de gravité, et que nous voulons le déplacer jusqu'au point B. Voici une représentation de ce déplacement :



La flèche illustre le déplacement du point A vers le point B. Au niveau de l'axe X on passe de la valeur 2 à la valeur 8. Autrement dit il faut se déplacer de 8 – 2 = 6 unités. Au niveau de l'axe Y on passe de la valeur 3 à la valeur 6, le déplacement est donc de 6 – 3 = 3 unités. On appelle cet élément un vecteur.

Un vecteur est défini par :

- Une direction (la droite support)
- Un sens (ici de A vers B)
- Une norme (c'est sa longueur, ici à peu près 6,7 unités).

Pour rester encore au plus près du LSL on représente un vecteur ainsi :

Vecteur AB = <6, 3>

Vous constatez qu'on a la même représentation que pour un point alors qu'il s'agit de deux entités vraiment différentes. Un point est juste une localisation virtuelle dans l'espace alors qu'un vecteur représente un déplacement. C'est une source de confusion et vous devez être très attentif à ne pas confondre les deux même si dans les deux cas le LSL parle de vecteur.

Le calcul du vecteur AB est donc :

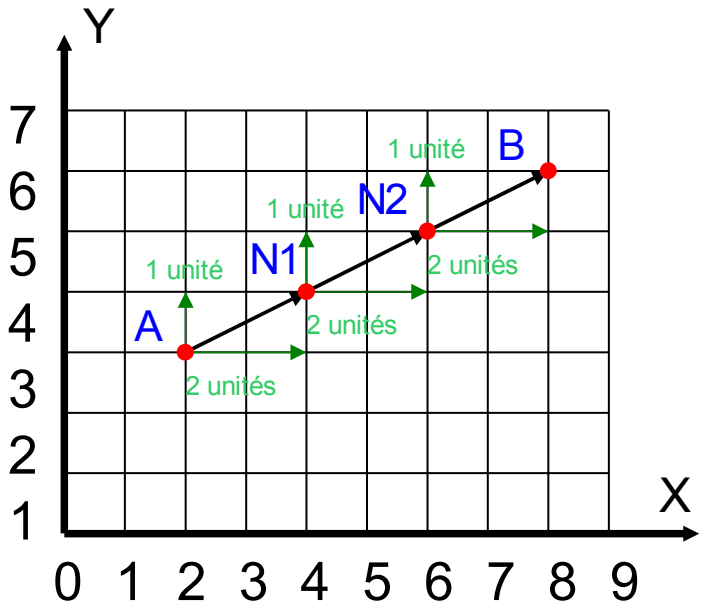
Vecteur AB = point B – point A = <8, 6> - <2, 3> = <8 – 2, 6 – 3> = <6, 3>

Il peut paraître étrange d'obtenir un vecteur en soustrayant deux points, nous allons éclaircir cela plus loin.

2.3 Interpolation

Pour réaliser le déplacement de A vers B on pourrait tout simplement définir directement la position finale et hop c'est fini. Mais ce ne serait pas très esthétique. Il est préférable de faire un mouvement progressif. Il faut

donc définir des étapes pour le déplacement. En langage mathématique on parle d'interpolation. Observez la figure :



Le trajet est découpé en trois en définissant deux étapes intermédiaires : N1 <4, 4> et N2 <6, 5>. Le vecteur global est donc tronçonné en 3 vecteurs égaux : A-N1, N1-N2 et N2-B. Il suffit de déplacer dans un premier temps l'objet en N1, puis en N2 et finalement en B. Le déplacement devient plus fluide. Rien n'empêche de multiplier ces étapes pour obtenir quelque chose de satisfaisant.

Comment est-ce que sont définies ces étapes ? On voit facilement sur la figure que les vecteurs sont égaux mais d'un point de vue plus mathématique ? Intéressons-nous aux coordonnées. Le vecteur A-B est égal à <6, 3>. On découpe ce vecteur en 3 :

$$<6, 3> / 3 = <2, 1>$$

Que donne la division d'un vecteur par un scalaire (un scalaire est l'appellation mathématique d'un simple nombre) ? C'est un vecteur de même direction et même sens mais de norme divisée par le scalaire. Autrement dit :

$$<6, 3> / 3 \ = <6 / 3, 3 / 3> = <2, 1>$$

Les petits vecteurs sont tous égaux et de valeur <2, 1>. Ici remarquez qu'un vecteur n'a pas de position précise dans l'espace puisque ces trois vecteurs strictement égaux (même norme, même sens et même direction) n'ont pas la même position.

Pour trouver les coordonnées du point N1 il suffit d'ajouter aux coordonnées du point A le petit vecteur :

$$\text{Coordonnées du point N1} = <2, 3> + <2, 1> = <2 + 2, 3 + 1> = <4, 4>$$

Mais quelle est cette étrange opération qui rappelle la soustraction vue plus haut ? On ajoute un point et un vecteur ! Pourtant le résultat est correct… Voyons ça de plus près…

Pour aller du point O, origine des axes et de valeur <0, 0> jusqu'au point A, on a le vecteur OA égal à <2, 3>. C'est-à-dire exactement la même valeur que les coordonnées du point A. On peut donc définir que les coordonnées d'un point sont en fait le vecteur entre l'origine et ce point (que les mathématiciens puristes me pardonnent !). Donc en fait on fait une addition de deux vecteurs et on obtient un vecteur :

$$O\text{-}N1 = \text{Vecteur } O\text{-}A + \text{Vecteur } A\text{-}N1 = \langle 2, 3\rangle + \langle 2, 1\rangle = \langle 4, 4\rangle$$

Cette valeur <4, 4> représente aussi bien les coordonnées du point N1 que le vecteur O-N1. Au passage remarquez que si en terme de déplacement la ligne droite est équivalente au chemin détourné il n'en serait pas de même en terme de délai.

On en déduit la position de N2 :

$$O\text{-}N2 = \text{Vecteur } O\text{-}A + 2 * \langle 2, 1\rangle = \langle 2, 3\rangle + 2 * \langle 2, 1\rangle = \langle 2, 3\rangle + \langle 4, 2\rangle = \langle 6, 5\rangle$$

On peut aussi la déduire à partir du point N1 :

$$O\text{-}N2 = \text{Vecteur } O\text{-}N1 + \langle 2, 1\rangle = \langle 4, 4\rangle + \langle 2, 1\rangle = \langle 6, 5\rangle$$

Nous avons ainsi fait une interpolation linéaire.

2.4 Gestion du temps

Un déplacement se fait dans un certain temps, il nous faut prendre en compte cette dimension temporelle. L'objet, que nous appelons « mobile » puisqu'il se déplace, a une certaine vitesse. Pour simplifier on admet que la vitesse est constante. L'unité de distance étant le mètre et celle de durée étant la seconde (même dans SL !) la vitesse s'exprime en mètres par seconde. Si notre mobile va du point A au point B à la vitesse de 2 m/s quelle est la durée d'une petite étape de l'interpolation ?

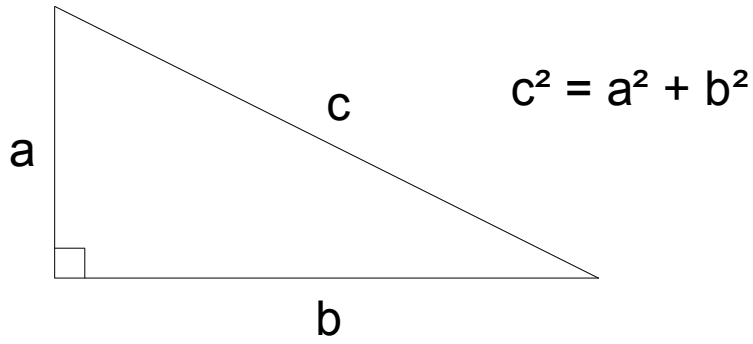
On sait que :

$$\text{Vitesse} = \text{Distance} / \text{Temps}$$

D'où on en conclue que :

$$\text{Temps} = \text{Distance} / \text{Vitesse} \quad \text{et} \quad \text{Distance} = \text{Vitesse} / \text{Temps}$$

Quelle est la distance pour aller de A vers B ? Autrement formulé quelle est la norme du vecteur A-B ? Nous allons la déterminer grâce à Pythagore. Dans un triangle rectangle on a la relation :



On peut le formuler : le carré de l'hypoténuse est égal à la somme des carrés des deux autres côtés.

D'où on en déduit la longueur de l'hypoténuse :

$$c = sqr\ (a^2 + b^2)$$

Pour notre vecteur (on considère l'unité de base étant 1 mètre) :

$$Norme = sqr\ (6^2 + 3^2) = sqr\ (36 + 9) = sqr\ (45) = 6{,}708 \text{ mètres}$$

On peut ainsi déterminer la durée du déplacement :

$$Durée = Distance / Vitesse = 6{,}708 / 2 = 3{,}354 \text{ secondes}$$

On peut en déduire la durée d'un déplacement élémentaire. Mais il y a un petit défaut dans ce raisonnement. Diviser la distance totale à parcourir pour avoir des déplacements élémentaires n'est pas une approche judicieuse. Si vous avez par exemple 1 mètre à parcourir vous pouvez déterminer que des sauts de 20 cm sont suffisants pour une vitesse de 2 m/s. Par contre vous prendrez des sauts de 10 cm pour une vitesse de 1 m/s. On voit donc que la longueur des sauts élémentaires dépend de la vitesse, ce qui est normal. Le bon repère n'est donc pas la distance mais la durée.

Si on définit une durée de base pour avoir un déplacement fluide, cette durée de base ne variera jamais, quelle que soit la distance totale à parcourir et la vitesse du mobile. En général on considère qu'un délai de base de 0,04 seconde est un bon équilibre entre la fluidité du déplacement et les calculs exigés qui génèrent forcément du lag. Ça correspond à 25 petits déplacements par seconde (1 / 0,04). C'est la vitesse de défilement des images d'un téléviseur classique. Nous allons donc partir de cette durée de base et trouver la longueur du déplacement correspondant :

$$Distance\ élémentaire = Temps * Vitesse = 0{,}04 * 2 = 0{,}08 \text{ mètre} = 8 \text{ centimètres}$$

Les petits bonds du mobile seront donc de 8 centimètres. Mais combien y en aura-t-il ?

$$Nombre\ de\ bonds = Distance\ Totale / Distance\ d'un\ bond = 6{,}708 / 0{,}08 = 83{,}85$$

Pas de chance : on n'obtient pas un nombre exact. Ce serait d'ailleurs exceptionnel que cela se produise. Comment allons-nous gérer cette fraction de bond ? Nous avons le choix :

- on fait 83 bonds et on ajuste la distance
- on ajuste la durée élémentaire pour rendre le nombre exact

La première solution est plus simple et généralement utilisée.

$$Distance\ élémentaire = Distance\ Totale / Nombre\ de\ bonds = 6{,}708 / 83 = 0{,}0808$$

La différence est infime et la tricherie sur la vitesse indécelable surtout dans un monde où le lag est la norme…

2.5 Premier codage

Est-ce que nous avons suffisamment d'éléments pour coder un déplacement par translation ? Nous allons voir ça…

```
// Durée de base en secondes
float TIME_BASE = .04;
// Vitesse de déplacement en m/s
float VITESSE = 2.0;
// Position d'arrivée
vector v_pos_arrivee = <150.0, 150.0, 2.0>;
// Position de départ
vector v_pos_depart = <140.0, 140.0, 5.0>;

// Vecteur élémentaire
vector v_elementaire;
// Nombre de bonds
integer i_bonds;
// Position intermédiaire
vector v_pos;

default
{
	touch_start(integer total_number)
	{
		// Vecteur de déplacement total
		vector v_deplacement_total = v_pos_arrivee - v_pos_depart;
		// Distance à parcourir
		float f_longueur_totale = llVecMag(v_deplacement_total);
		// Durée du déplacement total
		float f_duree_totale = f_longueur_totale / VITESSE;
		// Nombre de bonds
		i_bonds = (integer) (f_duree_totale / TIME_BASE);
		// Vecteur de déplacement élémentaire
		v_elementaire = v_deplacement_total / i_bonds;
		// Initialisation de la position intermédiare
		v_pos = v_pos_depart;
		// Position de départ
		llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION, v_pos_depart]);
		// Mise en route du timer
		llSetTimerEvent(TIME_BASE);
	}
	timer()
	{
		if(i_bonds--)
		{
			// Calcul nouvelle position
			v_pos += v_elementaire;
			// Positionnement
			llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION, v_pos]);
		}
		else
		{
			// Confirmation position
			llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION, v_pos_arrivee]);
			// Arrêt du timer
```

```
                llSetTimerEvent(.0);
            }
        }
}
```

Nous allons un peu analyser ce code. Au départ quelques paramètres :

```
// Durée de base en secondes
float TIME_BASE = .04;
// Vitesse de déplacement en m/s
float VITESSE = 2.0;
// Position d'arrivée
vector v_pos_arrivee = <150.0, 150.0, 2.0>;
// Position de départ
vector v_pos_depart = <140.0, 140.0, 5.0>;
```

Le temps de base est celui que nous avons déterminé précédemment, de même que la vitesse. Comme ce sont des constantes j'ai mis leurs noms en majuscules. Ensuite on détermine les points de départ et d'arrivée. Ajustez ces valeurs selon le lieu où vous testez le script au risque de voir s'éloigner votre objet de manière inattendue ! En particulier rappelez-vous qu'on ne peut pas déplacer en une seule fois un objet sur une distance de plus de 10 mètres.

```
// Vecteur élémentaire
vector v_elementaire;
// Nombre de bonds
integer i_bonds;
// Position intermédiaire
vector v_pos;
```

Ici nous avons 3 variables globales. La première est le vecteur élémentaire de déplacement. Il correspond aux éléments d'interpolation. Suit le nombre de bonds à faire et enfin une variable qui va garder en mémoire la position intermédiaire du mobile.

```
    touch_start(integer total_number)
    {
        // Vecteur de déplacement total
        vector v_deplacement_total = v_pos_arrivee - v_pos_depart;
        // Distance à parcourir
        float f_longueur_totale = llVecMag(v_deplacement_total);
        // Durée du déplacement total
        float f_duree_totale = f_longueur_totale / VITESSE;
        // Nombre de bonds
        i_bonds = (integer) (f_duree_totale / TIME_BASE);
        // Vecteur de déplacement élémentaire
        v_elementaire = v_deplacement_total / i_bonds;
        // Initialisation de la position intermédiare
        v_pos = v_pos_depart;
        // Position de départ
        llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION, v_pos_depart]);
```

```
        // Mise en route du timer
        llSetTimerEvent(TIME_BASE);
    }
```

Lorsque l'objet est touché l'événement **touch_start** se déclenche. On commence par calculer le vecteur total de déplacement, ce qui correspond au vecteur A-B vu précédemment. Il suffit de soustraire la position de départ à la position d'arrivée. Ce qui donne :

v_deplacement_total = v_pos_arrivee - v_pos_depart = <150, 150, 2> - <140, 140, 5> = <10, 10, -3>

On calcule ensuite la distance totale à parcourir qui correspond à la norme du vecteur précédent. LSL nous offre la fonction **llVecMag** pour ce calcul. Sinon il faudrait faire :

f_longueur_totale = sqr ($10^2 + 10^2 + (-3)^2$) = sqr (209) = 14,457 mètres

On peut alors calculer la durée du déplacement :

f_duree_totale = f_longueur_totale / VITESSE = 14,457 / 2 = 7,228 secondes

On en déduit le nombre de bonds :

i_bonds = f_duree_totale / TIME_BASE = 7,228 / 0,04 = 180,7

Dont on ne conserve que la partie entière 180.

On peut alors trouver le vecteur de déplacement élémentaire :

v_elementaire = v_deplacement_total / i_bonds = <10, 10, -3> / 180 = <.055, .055, -.016>

Ensuite on initialise la variable **v_pos** avec la position de départ, on place le mobile dans cette position avec la fonction **llSetLinkPrimitiveParamsFast** et on lance le timer. Le plus gros du travail a été fait.

```
timer()
{
    if(i_bonds--)
    {
        // Calcul nouvelle position
        v_pos += v_elementaire;
        // Positionnement
        llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION, v_pos]);
    }
    else
    {
        // Confirmation position
        llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION, v_pos_arrivee]);
        // Arrêt du timer
        llSetTimerEvent(.0);
    }
}
```

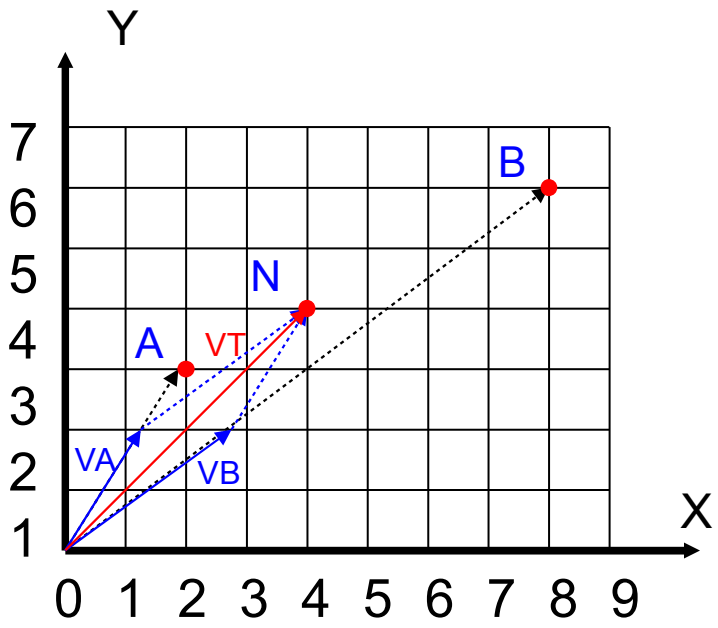
Tout les 0,04 seconde se déclenche l'événement **timer**. On décrémente la variable **i_bonds** de telle façon qu'on activera cet événement le nombre de fois voulu. Le calcul de la nouvelle position devient très simple puisqu'il suffit d'ajouter à la position actuelle le vecteur élémentaire. Lorsque tous les bonds ont été effectués on prend la précaution d'ajuster la position du mobile (ça annule les petites erreurs sur le calcul et ça absorbe ce qui peut être manqué à cause du lag) et on arrête le timer.

2.6 Second codage

Nous allons voir un codage un peu plus formel. En général les interpolations se calculent sur une plage allant de 0 à 1. Par exemple pour une interpolation linéaire entre deux vecteurs nous avons la formule de Nexii Malthus qui a créé une intéressante librairie d'interpolations :

```
vector vLin(vector v0, vector v1, float t){return v0 * (1.0 - t) + v1 * t;}
```

Voyons comment opère cette formule :



Considérons l'étape intermédiaire N située au tiers de la distance à parcourir.

$$VA = V0 * (1 - t) = OA * (1 - 1/3) = \langle 2, 3\rangle * 2/3 = \langle 4/3, 2\rangle = \langle 1.33, 2\rangle$$
$$VB = V1 * t = OB * 1/3 = \langle 8, 6\rangle * 1/3 = \langle 8/3, 2\rangle = \langle 2.66, 2\rangle$$
$$VT = VA + VB = \langle 1.33, 2\rangle + \langle 2.66, 2\rangle = \langle 4, 4\rangle$$

On constate graphiquement qu'on retrouve bien le point N intermédiaire.

La formule est logique puisqu'on ajoute la fraction atteinte du second vecteur à la fraction restante du premier. On a donc ici le tiers du second vecteur et il reste les deux tiers du premier. On peut établir le codage sur ces bases :

```
// Durée de base en s
float TIME_BASE = .04;
// Vitesse de déplacement en m/s
```

```
float VITESSE = 2.0;
// Position d'arrivée
vector v_pos_arrivee = <150.0, 150.0, 2.0>;
// Position de départ
vector v_pos_depart = <140.0, 140.0, 5.0>;

// Fraction d'interpolation
float f_fraction;
// Valeur d'interpolation (entre 0 et 1)
float f_valeur;

// Interpolation linéaire entre deux vecteurs
vector vLin(vector v0, vector v1, float t){return v0 * (1 - t) + v1 * t;}

default
{
	touch_start(integer total_number)
	{
		// Vecteur de déplacement total
		vector v_deplacement_total = v_pos_arrivee - v_pos_depart;
		// Distance à parcourir
		float f_longueur_totale = llVecMag(v_deplacement_total);
		// Durée du déplacement total
		float f_duree_totale = f_longueur_totale / VITESSE;
		// Nombre de bonds
		float f_bonds = f_duree_totale / TIME_BASE;
		// Fraction d'interpolation
		f_fraction = 1.0 / f_bonds;
		// Initialisation valeur d'interpolation
		f_valeur = .0;
		// Position de départ
		llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION, v_pos_depart]);
		// Mise en route du timer
		llSetTimerEvent(TIME_BASE);
	}
	timer()
	{
		if(f_valeur < 1.0)
		{
			// Fraction d'interpolation
			f_valeur += f_fraction;
			// Positionnement
			llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION,
				vLin(v_pos_depart, v_pos_arrivee, f_valeur)]);
		}
		else
		{
			// Confirmation position
			llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION, v_pos_arrivee]);
			// Arrêt du timer
			llSetTimerEvent(.0);
```

```
            }
        }
}
```

Cette fois c'est la valeur de la fraction entre 0 et 1 mémorisée dans la variable **f_valeur** qui sert de référence pour la boucle. On ne triche plus sur la vitesse. On a un code plus générique.

2.7 Cas d'une primitive liée

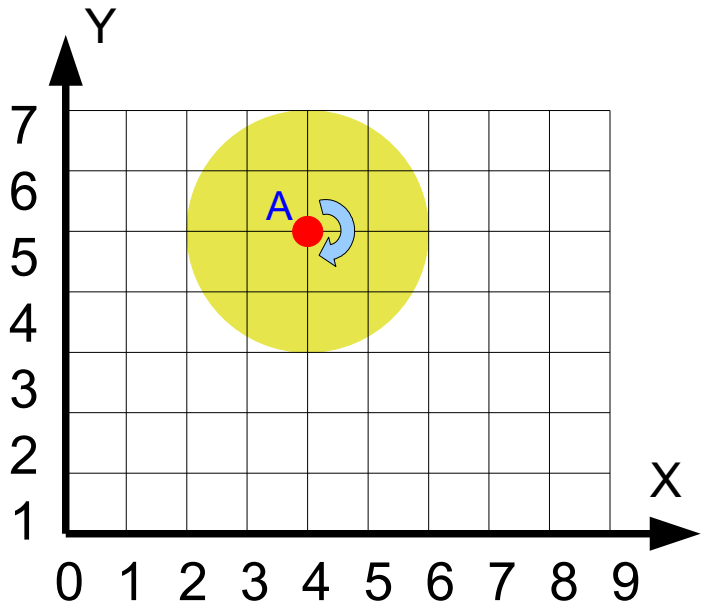
Notre étude de cas concernait un objet libre (racine d'un ensemble ou primitive isolée). Mais bien souvent la translation concerne une primitive liée. Y a-t-il des différences ? Tel que le code a été rédigé il n'y a aucune différence. La fonction **llSetLinkPrimitiveParamsFast** gère directement la position locale. Il faut juste rester dans les limites de liaison entre primitives pour les coordonnées de départ et d'arrivée. Il faut aussi que le script soit situé dans la primitive enfant concernée (à cause de la constante **LINK_THIS**), ou alors spécifier le numéro de la primitive.

3. Rotation

L'univers des machines est fait de mouvements dont l'un des plus fréquents est la rotation. Notre second projet va alors consister à faire tourner un objet. Nous avons vu précédemment pour les translations le système de coordonnées qui reste évidemment le même pour les rotations.

3.1 Une histoire d'angle

Prenons un disque (jaune sur la figure) centré au point A de coordonnées <4, 5> devant tourner dans le sens de la flèche avec une vitesse angulaire constante d'un demi tour par seconde :



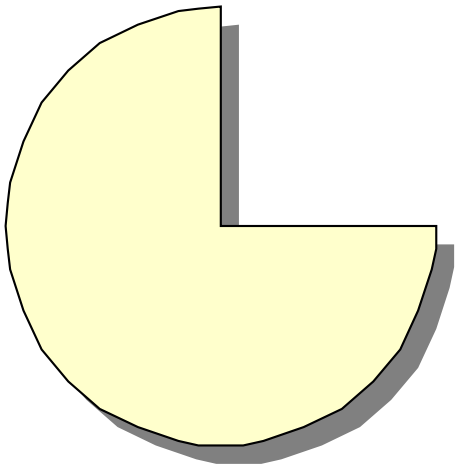
L'axe de rotation est évidemment l'axe Z perpendiculaire au disque et au schéma. C'est l'axe Z global, celui de la région dans laquelle se situe l'objet.

Cette fois ce n'est plus un déplacement comme pour la translation, l'objet ne change pas de position, mais une rotation. On va donc considérer un angle. L'unité la plus courante pour la mesure d'un angle est le degré. On sait qu'un tour complet correspond à 360 degrés, donc une moitié à 180 degrés et un quart à 90 degrés. Mais les fonctions du LSL attendent des radians. Donc un petit rappel sur le sujet est peut-être utile.

La conversion entre degrés et radians est simple, il suffit de savoir que :

360° = 2 * PI radians

En version radians un demi tour fait donc PI et un quart de tour PI / 2. Le LSL propose des constantes pratiques sur le sujet comme nous le verrons dans ce manuel.

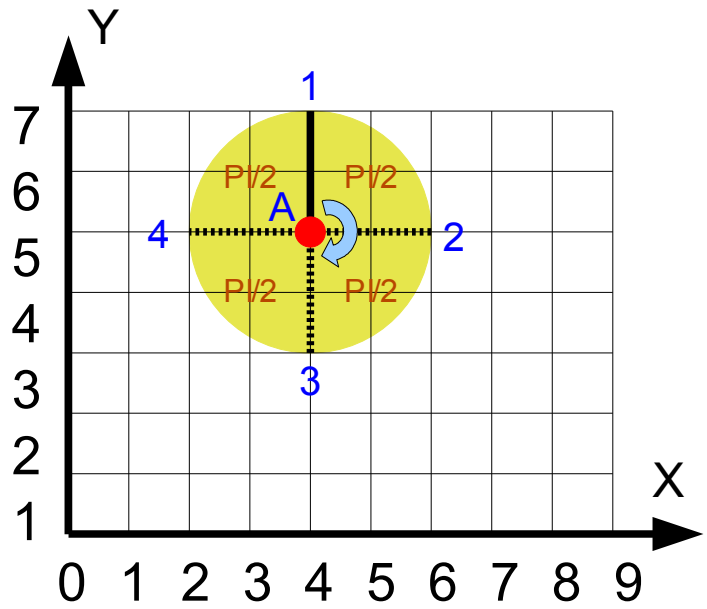


Dans la figure ci-dessus la partie manquante correspond à PI / 2 et la partie présente correspond à 3 * PI / 2.

Puisqu'il s'agit d'une rotation on parle de vitesse angulaire et en général elle est exprimée en radians par seconde. Ce qui est logique puisque les angles sont exprimés en radians. Mais ça ne parle pas à beaucoup de monde et lorsqu'on fait un script paramétré avec une vitesse angulaire il est plutôt mal venu d'utiliser les radians. Il vaut beaucoup mieux, pour être compréhensible, s'exprimer en tours par seconds. D'une certaine façon c'est la même chose puisqu'on sait qu'un tour complet correspond à 2 * PI. Il y a donc un rapport de 2 entre une valeur en tours par seconde et celle en radians par seconde. Par exemple dans notre cas nous voulons un demi-tour par seconde, ça correspond à une vitesse de PI radians par seconde.

3.2 Interpolation

Comme pour la translation il nous faut déterminer une interpolation régulière pour obtenir un rotation fluide. Adoptons le même raisonnement que celui que nous avons utilisé pour la translation.



Prenons un point de repère initial sur le disque (représenté par le trait noir sur la figure). Au départ le trait est face à la position marquée 1. La seconde position est marquée 2 et est atteinte après une rotation des 90° ou PI/2

radians. La Troisième position est marquée 3 et est atteinte également après une rotation des 90° ou PI/2 radians par rapport à la position 2, ou de 180° ou PI radians par rapport à la position 1. Et ainsi de suite pour la position 4 et retour à la 1…

Contrairement à la translation du chapitre précédent où on voulait s'arrêter pile sur le point d'arrivée, ici on désire une rotation continue. On va donc déterminer l'angle élémentaire directement à partir du temps de base sans se soucier du nombre d'interpolations nécessaires. Voilà ce que ça donne avec une vitesse en tours par seconde :

Angle élémentaire = 2 * PI * Vitesse * Temps de base = 2 * PI * 0.5 * 0.04 = 0.125 rd

3.3 Premier codage (axe global et objet libre)

Maintenant nous avons tout en mains pour coder la rotation :

```
// Durée de base en s
float TIME_BASE = .04;
// Vitesse angulaire en tours/s
float VITESSE = .5;
// Axe de rotation
vector AXE = <.0, .0, 1.0>;


// Rotation initiale
rotation r_rot_init;
// Angle de base
float f_elementaire;
// Angle cumulé
float f_angle;

default
{
   touch_start(integer total_number)
   {
      // Angle de base
      f_elementaire = TWO_PI * VITESSE * TIME_BASE;
      // Rotation intiale
      r_rot_init = llGetRot();
      // Mise en route du timer
      llSetTimerEvent(TIME_BASE);
   }
   timer()
   {
      // Calcul nouvel angle
      f_angle += f_elementaire;
      // Rotation
      rotation r_rot = llAxisAngle2Rot(AXE, f_angle);
      // Application rotation
      llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_ROTATION, r_rot_init * r_rot]);
   }
}
```

Analysons ce code.

```
// Durée de base en s
float TIME_BASE = .04;
// Vitesse angulaire en tours/s
float VITESSE = .5;
// Axe local de rotation
vector AXE = <.0, .0, 1.0>;
```

On commence avec 3 paramètres : durée de base, vitesse angulaire et axe local de rotation. Cet axe est l'axe global de la région.

```
// Rotation initiale
rotation r_rot_init;
// Angle de base
float f_elementaire;
// Angle cumulé
float f_angle;
```

Nous avons ensuite 3 variables, la première **r_rot_init** mémorise la rotation intiale. La seconde **f_elementaire** contient l'angle d'interpolation à appliquer toutes les 0.04 s. La troisième **f_angle** mémorise l'angle cumulé.

```
touch_start(integer total_number)
{
    // Angle de base
    f_elementaire = TWO_PI * VITESSE * TIME_BASE;
    // Rotation intiale
    r_rot_init = llGetRot();
    // Mise en route du timer
    llSetTimerEvent(TIME_BASE);
}
```

Lorsqu'on touche l'objet on déclenche l'événement **touch_start**. On en profite pour initialiser les variables. On commence par calculer l'angle de base :

$$f\_elementaire = PI * 2.0 * VITESSE * TIME\_BASE = 0.125$$

On mémorise la rotation initiale de l'objet récupérée par la fonction **llGetRot**.

On lance le timer.

```
timer()
{
    // Calcul nouvel angle
    f_angle += f_elementaire;
    // Rotation
    rotation r_rot = llAxisAngle2Rot(AXE, f_angle);
    // Application rotation
```

```
    llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_ROTATION, r_rot_init * r_rot]);
  }
```

Toutes les 0.04s on actualise la variable **f_angle** en ajoutant l'angle élémentaire **f_elementaire**. Remarquez que la valeur croit indéfiniment mais la limite en valeur du type float nous laisse une marge confortable. Pour être puriste on pourrait tester le dépassement de 2 * PI et limiter la valeur au-dessous de ce seuil. On calcule ensuite la rotation correspondante avec la fonction **llAxisAngle2Rot**. Il ne reste plus qu'à ajouter la nouvelle rotation à l'objet avec la fonction **llSetLinkPrimitiveParamsFast**. Pour mémoire on ajoute les rotations avec l'opérateur « * ».

Si la rotation est inversée par rapport au résultat attendu il suffit de rendre négatif l'angle ou alors d'inverser l'axe, le résultat sera le même.

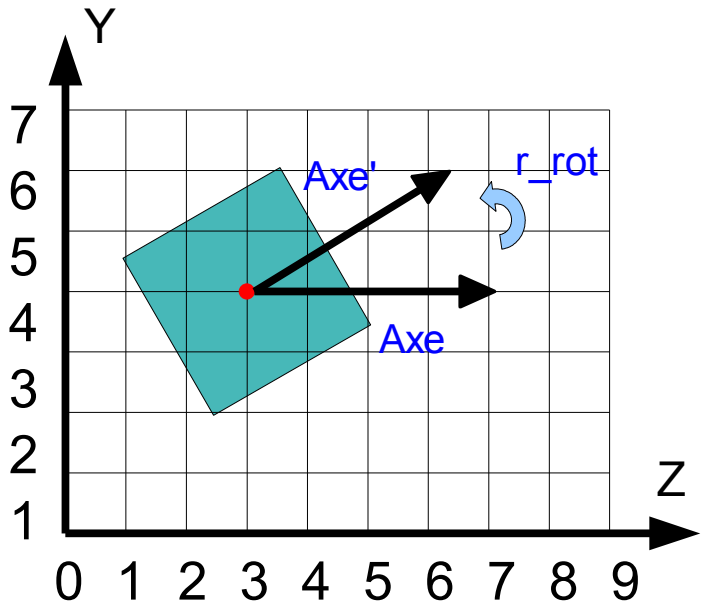
3.4 Second codage (axe local et objet libre)

Nous avons fait tourner l'objet sur un axe global de la région. Comment faire pour que cette rotation s'effectue sur un axe local ? Il suffit d'adapter l'axe selon la rotation de l'objet et tout marche pareil (les changements apparaissent en gras et gros) :

```
// Durée de base en s
float TIME_BASE = .04;
// Vitesse angulaire en tours/s
float VITESSE = .5;
// Axe de rotation
vector AXE = <.0, .0, 1.0>;


// Rotation initiale
rotation r_rot_init;
// Angle de base
float f_elementaire;
// Angle cumulé
float f_angle;

default
{
  touch_start(integer total_number)
  {
    // Angle de base
    f_elementaire = TWO_PI * VITESSE * TIME_BASE;
    // Rotation initiale
    r_rot_init = llGetRot();
     // Adaptation de l'axe
     AXE *= r_rot_init;
    // Mise en route du timer
    llSetTimerEvent(TIME_BASE);
  }
  timer()
  {
    // Calcul nouvel angle
    f_angle += f_elementaire;
```

```
        // Rotation
        rotation r_rot = llAxisAngle2Rot(AXE, f_angle);
        // Application rotation
        llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_ROTATION, r_rot_init * r_rot]);
    }
}
```

Mais que signifie réellement cette expression : AXE *** r_rot_init** ?



On multiplie un vecteur par une rotation, qu'est-ce que cela donne ? Tout simplement un changement de la direction du vecteur de la valeur de la rotation. En d'autres termes on fait tourner le vecteur de l'angle correspondant à la rotation. Sur la figure est représenté l'axe parallèle à l'axe Z global au niveau de l'objet. Cet objet possède une rotation dans ce référentiel. Cette rotation c'est justement **r_rot_init**. Lorsqu'on multiplie le vecteur qui représente l'axe par la rotation de l'objet (**r_rot_init**) on fait tourner cet axe pour l'amener à la position représentée par Axe' sur la figure qui est l'axe Z local de l'objet. Le script permet alors de faire tourner l'objet autour de cet axe local.

3.5 Troisième codage (axe local et primitive liée)

Ici les choses se compliquent un peu. Si nous utilisons le script du point 3.4 que se passe-t-il ? Si le script est dans la primitive la fonction **llGetRot** nous donne toujours la rotation globale et la fonction **llSetLinkPrimitiveParamsFast** est toujours aussi fantaisiste. Mais ça devrait marcher. Mais rappelez-vous comment nous avons déterminé l'axe de rotation :

AXE *** r_rot_init**

La variable **r_rot_init** contenant la rotation globale de la primitive. Que se passe-t-il si l'objet change de rotation ? Cet axe devient faux. D'autre part la variable **r_rot_init** conserve la rotation globale actuelle de la primitive, cette valeur devient aussi fausse en cas de rotation de l'objet. Moralité :

Pour une primitive enfant il faut travailler en repère local

pour rester indépendant des changements de rotation de l'objet.

Nous avons donc besoin de la rotation locale de la primitive. Il existe la fonction **llGetLocalRot** qui permet de la récupérer. Si le script est dans la primitive enfant tout va bien. Mais si le script est dans la racine ? A ce moment-là on récupère la rotation de la primitive enfant avec la fonction **llGetLinkPrimitiveParams** qui donne la rotation locale grâce au nouveau paramètre PRIM_ROT_LOCAL.

Un autre problème se présente ensuite dans l'application de la rotation calculée. Mais le nouveau paramètre PRIM_ROT_LOCAL de la fonction **llSetLinkPrimitiveParamsFast** simplifie le problème,

On a donc deux versions du script. Dans la première le script se situe dans la primitive enfant (les changements ont été caractérisés) :

```
// Durée de base en s
float TIME_BASE = .04;
// Vitesse angulaire en tours/s
float VITESSE = .5;
// Axe de rotation
vector AXE = <.0, .0, 1.0>;


// Rotation initiale
rotation r_rot_init;
// Angle de base
float f_elementaire;
// Angle cumulé
float f_angle;

default
{
   touch_start(integer total_number)
   {
      // Angle de base
      f_elementaire = TWO_PI * VITESSE * TIME_BASE;
      // Rotation initiale
      r_rot_init = llGetLocalRot();
      // Adaptation de l'axe
      AXE *= r_rot_init;
      // Mise en route du timer
      llSetTimerEvent(TIME_BASE);
   }
   timer()
   {
      // Calcul nouvel angle
      f_angle += f_elementaire;
      // Rotation
      rotation r_rot = llAxisAngle2Rot(AXE, f_angle);
      // Application rotation
      llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_ROT_LOCAL,  r_rot_init * r_rot]);
   }
```

```
}
```

Dans cette seconde version le script se situe dans la racine ou une autre primitive enfant :

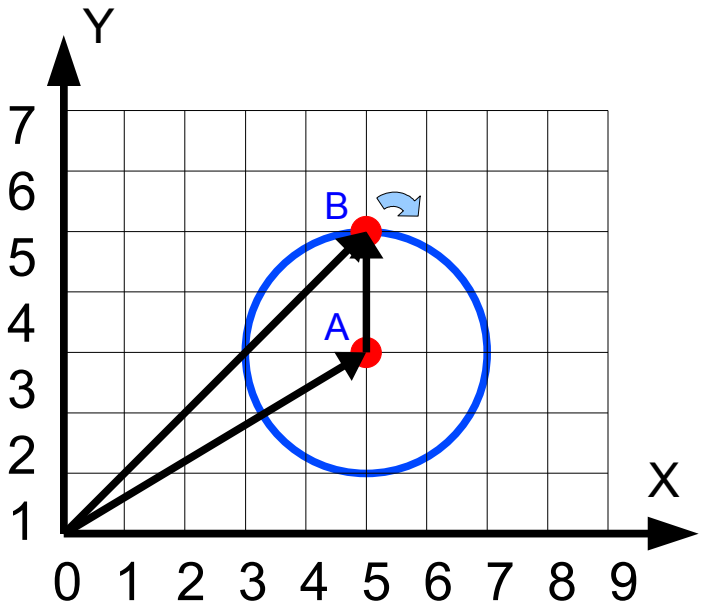
```
// Durée de base en s
float TIME_BASE = .04;
// Vitesse angulaire en tours/s
float VITESSE = .5;
// Axe de rotation
vector AXE = <.0, .0, 1.0>;
// Numéro de liaison primitive enfant
integer PRIM = 2;

// Rotation initiale
rotation r_rot_init;
// Angle de base
float f_elementaire;
// Angle cumulé
float f_angle;

default
{
    touch_start(integer total_number)
    {
        // Angle de base
        f_elementaire = TWO_PI * VITESSE * TIME_BASE;
        // Rotation initiale
        r_rot_init = llList2Rot(llGetLinkPrimitiveParams(PRIM, [PRIM_ROT_LOCAL]), 0);
        // Adaptation de l'axe
        AXE *= r_rot_init;
        // Mise en route du timer
        llSetTimerEvent(TIME_BASE);
    }
    timer()
    {
        // Calcul nouvel angle
        f_angle += f_elementaire;
        // Rotation
        rotation r_rot = llAxisAngle2Rot(AXE, f_angle);
        // Application rotation
        llSetLinkPrimitiveParamsFast(PRIM, [PRIM_ROT_LOCAL, r_rot_init * r_rot]);
    }
}
```

4. Déplacement circulaire

Nous avons vu la translation et la rotation. Nous allons envisager à présent le déplacement circulaire, c'est-à-dire autour d'un axe. Nous avons appelé ce déplacement rotation excentrée dans le guide sur les rotations. Nous n'allons pas développer ce chapitre déjà suffisamment explicité dans le guide sur les rotations mais traiter les cas généraux en resituant le problème.

4.1 Une rotation à distance

Observez la figure suivante. Le point B doit tourner autour du point A en suivant le cercle bleu, avec une vitesse uniforme d'un demi-tour par seconde.



Le problème se résume à faire tourner le vecteur A-B autour de son origine A. Nous avons vu que pour faire tourner un vecteur il faut le multiplier par une rotation. Notre réflexion du chapitre précédent reste applicable ici mais au lieu d'appliquer la rotation directement à l'objet on va l'appliquer au vecteur A-B.

4.2 Premier codage (axe global et objet libre)

```
// Durée de base en s
float TIME_BASE = .04;
// Vitesse angulaire en tours/s
float VITESSE = .5;
// Axe de rotation
vector AXE = <.0, .0, 1.0>;
// Position du centre de rotation
vector CENTRE = <140.0, 140.0, 3.0>;
```

```
// Angle de base
float f_elementaire;
// Angle cumulé
float f_angle;
// Vecteur tournant
vector v_tournant;

default
{
   touch_start(integer total_number)
   {
      // Angle de base
      f_elementaire = TWO_PI * VITESSE * TIME_BASE;
      // Vecteur tournant
      v_tournant = llGetPos() - CENTRE;
      // Mise en route du timer
      llSetTimerEvent(TIME_BASE);
   }
   timer()
   {
      // Calcul nouvel angle
      f_angle += f_elementaire;
      // Rotation
      rotation r_rot = llAxisAngle2Rot(AXE, f_angle);
      // Positionnement
      llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION, CENTRE + v_tournant * r_rot]);
   }
}
```

Analysons ce code…

```
// Durée de base en s
float TIME_BASE = .04;
// Vitesse angulaire en tours/s
float VITESSE = .5;
// Axe de rotation
vector AXE = <.0, .0, 1.0>;
// Position du centre de rotation
vector CENTRE = <140.0, 140.0, 3.0>;
```

Nous retrouvons les mêmes paramètres que précédemment à l'exception d'un nouveau venu **CENTRE** qui mémorise la position du centre de rotation.

```
// Angle de base
float f_elementaire;
// Angle cumulé
float f_angle;
// Vecteur tournant
vector v_tournant;
```

Seulement trois variables : **f_elementaire** mémorise comme précédemment l'angle élémentaire, alors que **f_angle** cumule l'angle. Quant à **v_tournant** elle conserve l'information du vecteur tournant.

```
touch_start(integer total_number)
{
    // Angle de base
    f_elementaire = TWO_PI * VITESSE * TIME_BASE;
    // Vecteur tournant
    v_tournant = llGetPos() - CENTRE;
    // Mise en route du timer
    llSetTimerEvent(TIME_BASE);
}
```

La seule nouveauté ici est le calcul du vecteur tournant **v_tournant** à partir de la position actuelle de l'objet et du centre de rotation.

```
timer()
{
    // Calcul nouvel angle
    f_angle += f_elementaire;
    // Rotation
    rotation r_rot = llAxisAngle2Rot(AXE, f_angle);
    // Positionnement
    llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION, CENTRE + v_tournant * r_rot]);
}
```

Ici on cumule l'angle et on calcule la rotation. On fait tourner le vecteur en le multipliant par la rotation et on applique la nouvelle position qui est la somme du centre de rotation et du vecteur tournant. Au niveau de la figure il s'agit du vecteur O-B :

$$O\text{-}B = O\text{-}A + A\text{-}B$$

4.3 Second codage (axe local à l'objet et primitive enfant)

En général nous avons ce genre de déplacement circulaire au niveau d'une primitive enfant dans un objet. Si le script est dans la primitive enfant le changement est minime :

```
// Durée de base en s
float TIME_BASE = .04;
// Vitesse angulaire en tours/s
float VITESSE = .5;
// Axe de rotation
vector AXE = <.0, .0, 1.0>;
// Position du centre de rotation
vector CENTRE = <1.0, 1.0, 2.0>;

// Angle de base
float f_elementaire;
```

```
// Angle cumulé
float f_angle;
// Vecteur tournant
vector v_tournant;

default
{
   touch_start(integer total_number)
   {
      // Angle de base
      f_elementaire = TWO_PI * VITESSE * TIME_BASE;
      // Vecteur tournant
      v_tournant = llGetLocalPos() - CENTRE;
      // Mise en route du timer
      llSetTimerEvent(TIME_BASE);
   }
   timer()
   {
      // Calcul nouvel angle
      f_angle += f_elementaire;
      // Rotation
      rotation r_rot = llAxisAngle2Rot(AXE, f_angle);
      // Positionnement
      llSetLinkPrimitiveParamsFast(LINK_THIS, [PRIM_POSITION, CENTRE + v_tournant * r_rot]);
   }
}
```

Il suffit de récupérer la position locale avec la fonction **llGetLocalPos**. Ensuite la fonction **llSetLinkPrimitiveParamsFast** applique directement une position locale.

Si le script est dans le root alors il faut aller récupérer la position locale de l'enfant. Voici une fonction qui permet de le faire :

```
// Récupération position locale prim enfant
vector GetLinkPos(integer link_num) {
        vector v = llList2Vector(llGetLinkPrimitiveParams(link_num, [PRIM_POSITION]), 0);
        return (v - llGetRootPosition()) / llGetRootRotation();
}
```

La démonstration de cette fonction figure dans le guide sur les rotations. Voici le code avec les changements caractérisés :

```
// Durée de base en s
float TIME_BASE = .04;
// Vitesse angulaire en tours/s
float VITESSE = .5;
// Axe de rotation
vector AXE = <.0, .0, 1.0>;
// Position du centre de rotation
```

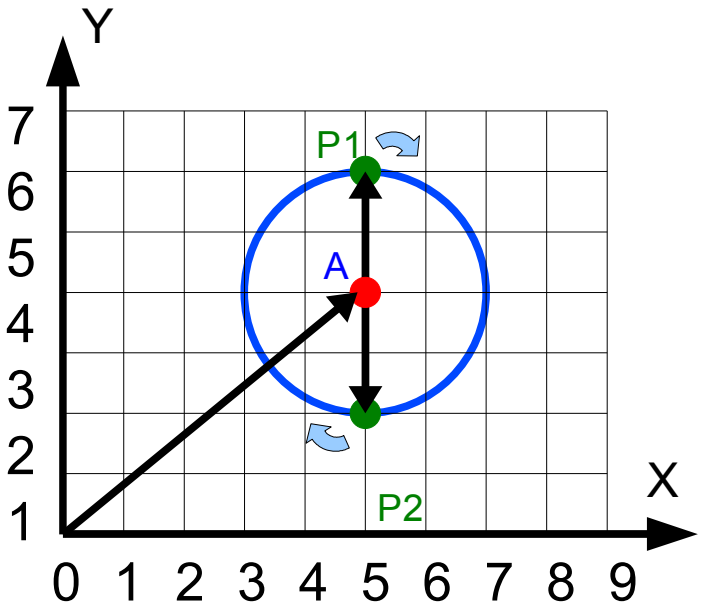
```
vector CENTRE = <140.0, 140.0, 3.0>;
// Numéro de liaison de la primitive enfant
integer PRIM = 2;

// Angle de base
float f_elementaire;
// Angle cumulé
float f_angle;
// Vecteur tournant
vector v_tournant;

// Récupération position locale prim enfant
vector GetLinkPos(integer link_num) {
        vector v = llList2Vector(llGetLinkPrimitiveParams(link_num, [PRIM_POSITION]), 0);
        return (v - llGetRootPosition()) / llGetRootRotation();
}

default
{
   touch_start(integer total_number)
   {
      // Angle de base
      f_elementaire = TWO_PI * VITESSE * TIME_BASE;
      // Vecteur tournant
      v_tournant = GetLinkPos(PRIM) - CENTRE;
      // Mise en route du timer
      llSetTimerEvent(TIME_BASE);
   }
   timer()
   {
      // Calcul nouvel angle
      f_angle += f_elementaire;
      // Rotation
      rotation r_rot = llAxisAngle2Rot(AXE, f_angle);
      // Positionnement
      llSetLinkPrimitiveParamsFast(PRIM, [PRIM_POSITION, CENTRE + v_tournant * r_rot]);
   }
}
```

On pourrait décliner encore plusieurs versions mais l'important est de comprendre le principe.

5. Un pédalier

5.1 Enoncé du projet

Ce projet a fait l'objet d'un concours sur JOL. Le problème posé est de synchroniser la rotation d'un pédalier sur un de ses axes avec celle des pédales autour du même axe.



On considère que le pédalier est constituée d'une seule primitive représentée par le point A ainsi que les pédales représentées par les points P1 et P2. Les 3 primitives étant des enfants.

Nous pouvons résoudre ce problème avec les éléments exposés aux chapitres précédents. La rotation du pédalier est celle vue au point 3.5, la rotation des pédales est pratiquement celle vue au point 4.3. La seule nouveauté est la synchronisation des deux mouvements.

En regardant de plus près on se rend compte que la rotation du pédalier et des deux vecteurs tournants des pédales est la même, ce qui va nous simplifier les calculs. On se rend compte aussi que les vecteurs des pédales sont opposés : ils ont la même direction, la même norme mais un sens opposé. Il suffît donc d'en calculer un pour obtenir facilement l'autre.

La principale difficulté de ce projet se situe dans la gestion des coordonnées locales avec toujours le problème des fonctions qui travaillent en coordonnées globales.

5.2 Codage

```
// *****************
//   Paramètres
// *****************
```

```
// Axe de rotation du pédalier
vector AXE = <1.0, .0, .0>;
// Vitesse de rotation en tours/seconde
float VITESSE = .4;
// Numéro de liaison du pédalier
integer NUMPEDALIER = 5;
// Numéro de liaison pédale 1
integer NUMPEDALE1 = 4;
// Numéro de liaison pédale 2
integer NUMPEDALE2 = 6;

// Variables de travail
integer         iOn;                // on-off
float           f_elementaire;      // Angle de base
float           f_angle;            // Angle cumulé
rotation        rRotPedalier;       // Rotation locale du pédalier
vector          vPosPedalier;       // Position locale pédalier
vector          vPosPedale;         // Position locale pédale 1
float           fDelai = .04;       // Délai de base en secondes
vector          vTournant;          // Vecteur tournant

// Récupération position locale prim enfant
vector GetLinkPos(integer link_num) {
   vector v = llList2Vector(llGetLinkPrimitiveParams(link_num, [PRIM_POSITION]), 0);
   return (v - llGetRootPosition()) / llGetRootRotation();
}

// Récupération rotation locale prim enfant
rotation GetLinkRot(integer link_num) {
   return llList2Rot(llGetLinkPrimitiveParams(link_num, [PRIM_ROT_LOCAL]), 0);
}

default
{
   state_entry()
   {
      // Position locale du pédalier
      vPosPedalier = GetLinkPos(NUMPEDALIER);
      // Rotation locale du pédalier
      rRotPedalier = GetLinkRot(NUMPEDALIER);
      // Angle de base
      f_elementaire = VITESSE * 14.4 * DEG_TO_RAD;
      // Adaptation de l'axe
      AXE *= rRotPedalier;
      // Position locale pédale 1
      vPosPedale = GetLinkPos(NUMPEDALE1);
      // Vecteur tournant
      vTournant = (vPosPedale - vPosPedalier) / rRotPedalier;
      // Petit message
      llOwnerSay("Cliquez sur le vélo pour activer ou désactiver la rotation");
   }
```

```
    touch_start(integer total_number)
    {
        if(iOn = !iOn) llSetTimerEvent(fDelai);
        else llSetTimerEvent(.0);
    }
    timer()
    {
        // Calcul nouvel angle
        f_angle += f_elementaire;
        // Rotation
        rotation r_rot = rRotPedalier * llAxisAngle2Rot(AXE, f_angle);
        // Application de la rotation au pédalier transposée en global
        llSetLinkPrimitiveParamsFast(NUMPEDALIER, [PRIM_ROT_LOCAL, r_rot]);
        // Vecteur rotatif pour les pédales (les vecteurs sont opposés)
        vector vPedale = vTournant * r_rot;
        // Positionnement pédale 1
        llSetLinkPrimitiveParamsFast(NUMPEDALE1, [PRIM_POSITION, vPosPedalier + vPedale]);
        // Positionnement pédale 2
        llSetLinkPrimitiveParamsFast(NUMPEDALE2, [PRIM_POSITION, vPosPedalier - vPedale]);
    }
}
```

Les paramètres et variables sont suffisamment commentés pour être explicites. Les deux fonctions **GetLinkPos** et **GetLinkRot** on déjà été utilisées dans les précédents chapitres. Nous allons nous intéresser aux calculs.

```
        // Position locale du pédalier
        vPosPedalier = GetLinkPos(NUMPEDALIER);
        // Rotation locale du pédalier
        rRotPedalier = GetLinkRot(NUMPEDALIER);
```

Nous avons besoin de la position locale du pédalier (point A de la figure) ainsi que de sa rotation initiale.

```
    // Angle de base
    f_elementaire = VITESSE * 14.4 * DEG_TO_RAD;
```

Ici nous calculons l'angle de base comme nous l'avons fait précédemment.

```
    // Adaptation de l'axe
    AXE *= rRotPedalier;
```

L'axe de rotation est ramené en local au pédalier en ajustant avec sa rotation locale.

```
        // Position locale pédale 1
        vPosPedale = GetLinkPos(iNumPedale1);
        // Vecteur tournant
        vTournant = (vPosPedale - vPosPedalier) / rRotPedalier;
```

Nous avons aussi besoin du vecteur tournant de la première pédale, le vecteur de la seconde sera déduit de celui-ci puisque les deux pédales sont en principe symétriques.

```
    // Calcul nouvel angle
    f_angle += f_elementaire;
    // Rotation
    rotation r_rot = rRotPedalier * llAxisAngle2Rot(AXE, f_angle);
```

Ici on incrémente l'angle et on calcul la rotation.

```
    // Application de la rotation au pédalier transposée en global
    llSetLinkPrimitiveParamsFast(iNumPedalier, [PRIM_ROT_LOCAL, r_rot]);
```

La rotation est appliquée directement au pédalier avec la fonction **llSetLinkPrimitiveParamsFast**.

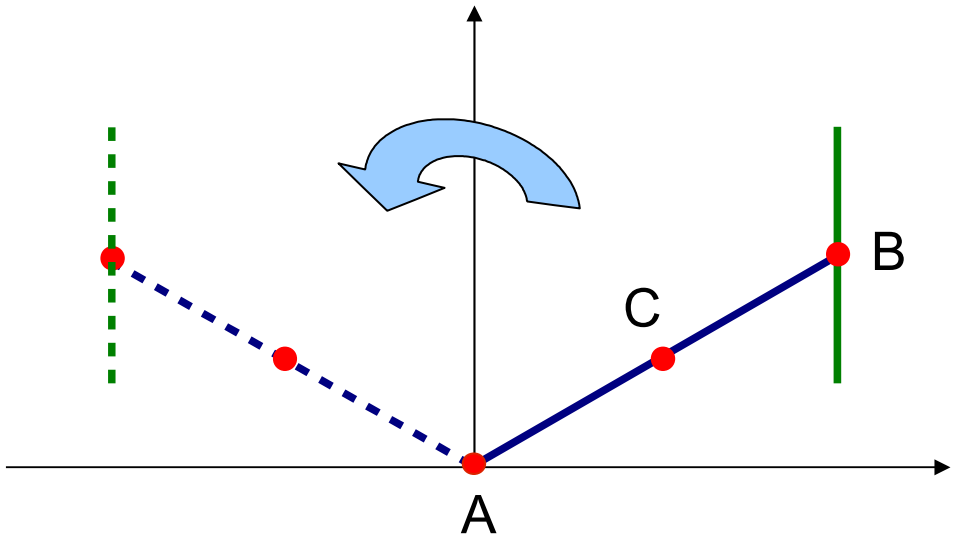
```
    // Vecteur rotatif pour les pédales (les vecteurs sont opposés)
    vector vPedale = vTournant * r_rot;
    // Positionnement pédale 1
    llSetLinkPrimitiveParamsFast(iNumPedale1, [PRIM_POSITION, vPosPedalier + vPedale]);
    // Positionnement pédale 2
    llSetLinkPrimitiveParamsFast(iNumPedale2, [PRIM_POSITION, vPosPedalier - vPedale]);
```

Pour les pédales on commence par calculer le nouveau vecteur tournant sur la base du vecteur initial auquel on applique la rotation. Enfin on positionne les pédales avec ce nouveau vecteur tournant. Comme les deux vecteurs sont opposés on ajoute et on soustrait le vecteur tournant pour obtenir les deux positions.

6. Un essuie-glace

Dans tous nos exemple précédents il y avait des translations et rotations mais jamais les deux en même temps. Nous allons maintenant aborder un projet qui allie les deux transformations simultanément. Il s'agit d'un mécanisme d'essuie-glace.

6.1 Enoncé du projet

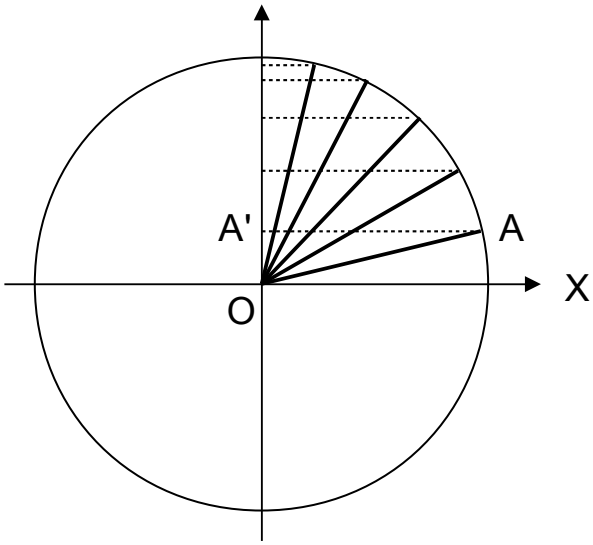


La bielle AB pivote autour de son point A jusqu'à atteindre la position figurée en pointillés puis revient à sa position intiale et ainsi de suite. Le balai représenté en vert reste vertical et suit le point B symétrique du point A par rapport au centre C de la bielle.

Le déplacement cette fois ne sera pas linéaire mais sinusoïdal pour ajouter du réalisme. D'autre part le système doit démarrer sur un clic et s'arrêter sur un clic mais comme un vrai essuie-glace il ne doit pas rester en position intermédiaire pour ne pas gêner la vision du conducteur mais indifféremment à droite ou à gauche.

6.2 Variation sinusoïdale

Qu'est-ce qu'une variation sinusoïdale ? Tous simplement celle qui suit une fonction sinusoïdale. Voici une illustration :



Si on fait tourner le rayon OA de façon linéaire et que l'on observe la projection OA' sur l'axe vertical on se rend compte que celle-ci n'est pas linéaire. La variation diminue progressivement jusqu'à devenir très faible. La valeur OA' est tout simplement le sinus de l'angle XOA. De la même manière la projection sur l'axe X nous donnerait le cosinus.

Il y a beaucoup de phénomènes à variation sinusoïdales dans le monde réel. Dès qu'une rotation est transformée en translation c'est le cas. Lorsque vous ouvrez une porte la rotation n'est jamais linéaire. Elle commence par tourner lentement au départ puis plus rapidement jusqu'à diminuer progressivement pour s'arrêter. Evidemment dans ce cas on aura pas un mouvement sinusoïdal pur mais pour une simulation c'est tout-à-fait convaincant.

Pour notre essuie-glace une variation linéaire ne ferait pas du tout naturel. C'est pour cette raison que nous allons adopter une interpolation sinusoïdale.

6.3 Interpolation sinusoïdale

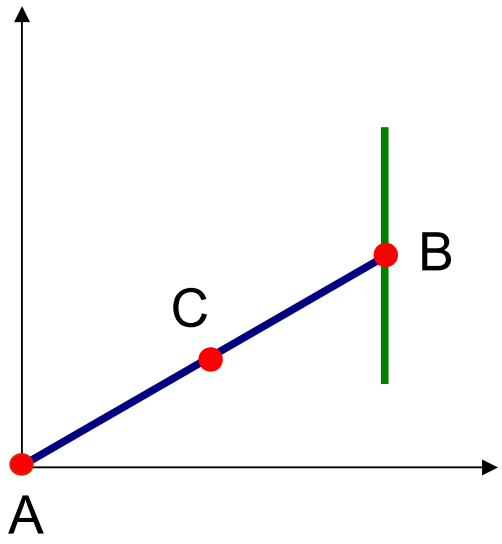
Nous allons encore utiliser la librairie d'interpolations de Nexii Malthus. Voici ce qu'il nous propose pour une variation sinusoïdale d'un **float** :

```
float fCos(float v0,float v1,float t){
        float f = (1.0 - llCos(t * PI)) / 2.0;
        return v0 * (1.0 – f) + v1 * f;}
```

La valeur de t variant de 0 à 1 la fonction retourne le cosinus de la fraction d'intervalle correspondante entre les valeurs des paramètres v0 et v1. Il faut donc ramener la variation d'angle dans cet intervalle de 0 à 1, ce qui n'est pas difficile.

6.4 Paramètres

La première chose à faire est de définir la position de départ. Nous allons positionner tous les éléments dans un certain état et lancer le script:



De quoi avons-nous besoin ?

6.4.1 Axe de rotation

Nous devons connaître l'axe de rotation (perpendiculaire à la figure ci-dessus). Il dépend uniquement de la manière dont la bielle doit bouger et doit figurer dans les paramètres.

6.4.2 Centre de rotation

La position du centre de rotation (le point A) est nécessaire pour déterminer le vecteur tournant. Les axes représentés sont ceux de la primitive racine, référence de l'objet. Dans cette position initiale la bielle a une position et une rotation quelconques par rapport à ces axes mais faciles à déterminer. Donc on peut trouver la position du point C. Pour connaître la position du point A de quoi avons-nous besoin ? De connaître la distance AC et l'axe situé dans la longueur de la bielle. Nous ne savons rien de ces éléments. Nous devons donc les entrer en paramètres.

6.4.3 Angle de balayage

Il faut aussi entrer en paramètre l'angle total de balayage parce que nous n'avons que la position initiale de la bielle sans aucune idée quant à l'angle de l'horizontale par rapport à elle.

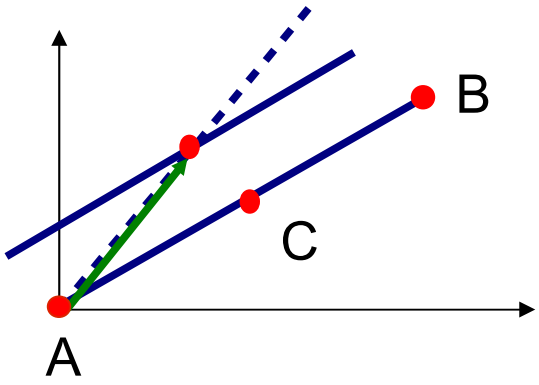
6.4.4 Numéro de liaison des primitives

Pour commander les primitives nous devons connaître leur numéro de liaison.

6.5 Analyse du mouvement

6.5.1 Mouvement de la bielle

Nous devons déterminer le vecteur tournant A-C pour positionner correctement la bielle. Mais ça ne suffit pas. Sinon nous allons obtenir ça (trait plein) :

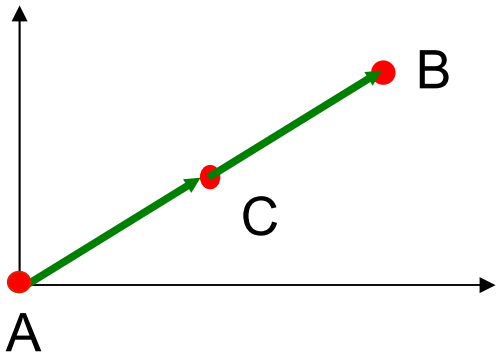


On voit que le repositionnement ne suffit pas. Il faut aussi appliquer une rotation. De quelle valeur ? Tout simplement la même que celle du vecteur tournant !

6.5.2 Mouvement du balai

Pour le mouvement du balai c'est beaucoup plus facile puisqu'il n'y a pas de rotation pour lui, juste un repositionnement.

Il faut déterminer la position du point B de la bielle qui est le centre du balai :



On voit facilement sur cette figure que puisqu'on connaît le vecteur A-C on en déduit le vecteur A-B qui est égal à :

$$A\text{-}B = 2 * A\text{-}C$$

6.6 Codage

```
// -------------------------
//    Constantes
// -------------------------
// Durée de base en s
float TIME_BASE = .04;
// Vitesse en battements/s
float VITESSE = 1.0;
// Angle total de balayage
float ANGLE_TOTAL = 120.0;
```

```
// Demie longueur de la bielle
float DEMIE_BIELLE = .75;
// Axe de rotation de la bielle
vector AXE_ROTATION = <1.0, .0, .0>;
// Axe de la longueur de la bielle
vector AXE_BIELLE = <.0, .0, 1.0>;
// Numéro de liaison de la primitive de la bielle
integer PRIM_BIELLE = 2;
// Numéro de liaison de la primitive du balai
integer PRIM_BALAI = 3;

// ------------------------
//    Variables globales
// ------------------------
// Centre
vector v_centre;
// Vecteur tournant
vector v_tournant;
// Etat
float fEtat;
// Incrément
float fStep;
// Rotation bielle
rotation r_bielle;
// Sens
integer iSens = TRUE;
// Marche-Arrêt
integer iOn;
// Demande d'arrêt
integer iDemande_Arret = FALSE;

// ------------------------
//    Fonctions - Librairie
// ------------------------
// Récupération position locale prim enfant
vector GetLinkPos(integer link_num) {
        vector v = llList2Vector(llGetLinkPrimitiveParams(link_num, [PRIM_POSITION]), 0);
        return (v - llGetRootPosition()) / llGetRootRotation();
}

// Récupération rotation locale prim enfant
rotation GetLinkRot(integer link_num) {
        return  llList2Rot(llGetLinkPrimitiveParams(link_num, [PRIM_ROT_LOCAL]), 0);
}

// Interpolation cos entre deux float (Nexii Malthus)
float fCos(float v0,float v1,float t){
        float F = (1.0 - llCos(t * PI)) / 2.0;
        return v0 * (1.0 - F) + v1 * F;}

// ------------------------
```

```
//    Fonctions du script
// ------------------------
// Gestion de la fin
gestion_fin(float f) {
        fEtat = f;
        iSens = !iSens;
        if(iDemande_Arret) {
                llSetTimerEvent(.0);
                iDemande_Arret = FALSE;
        }
}

// ------------------------
//    Etat
// ------------------------
default
{
        state_entry()
        {
                // Angle total en radians
                ANGLE_TOTAL *= DEG_TO_RAD;
                // Angle de base
                float f_angle = ANGLE_TOTAL * VITESSE * TIME_BASE;
                // Incrément
                fStep = f_angle / ANGLE_TOTAL;
                // Position de la bielle
                vector v_pos_bielle = GetLinkPos(PRIM_BIELLE);
                // Rotation de la bielle
                r_bielle = GetLinkRot(PRIM_BIELLE);
                // Calcul du centre
                v_centre = v_pos_bielle - AXE_BIELLE * DEMIE_BIELLE * r_bielle;
                // Vecteur tournant
                v_tournant = v_pos_bielle - v_centre;
                // Rotation de l'axe
                AXE_ROTATION *= r_bielle;
        }
        touch_start(integer total_number)
        {
                // Inhibition touch si demande d'arrêt
                if(iDemande_Arret) return;
                // Gestion du touch
                if(iOn = !iOn)
                        llSetTimerEvent(TIME_BASE);
                else {
                        llWhisper(0, "Demande d'arrêt prise en compte");
                        iDemande_Arret = TRUE;
                }
        }
        timer()
        {
                // Test du sens
```

```
            if(iSens)
            {
                // Incrémentation de l'angle
                fEtat += fStep;
                // Test d'arrivée en position finale
                if(fEtat >= 1.0) gestion_fin(1.0);
            }
            else
            {
                // Décrémentation de l'angle
                fEtat -= fStep;
                // Test d'arrivée en position de départ
                if(fEtat <= .0) gestion_fin(.0);
            }
            // Rotation
            rotation rot = llAxisAngle2Rot(AXE_ROTATION, fCos(.0, ANGLE_TOTAL, fEtat));
            // Vecteur tournant
            vector vec = v_tournant * rot;
            // Positionnement bielle
            llSetLinkPrimitiveParamsFast(PRIM_BIELLE, [
                PRIM_POSITION, v_centre + vec,
                PRIM_ROT_LOCAL, r_bielle * rot]);
            // Positionnement balai
            llSetLinkPrimitiveParamsFast(PRIM_BALAI, [
                PRIM_POSITION, v_centre + vec * 2.0]);
        }
}
```

Voyons le fonctionnement depuis l'initialisation du script.

6.6.1 Initialisation du script

L'angle total de balayage, entré en degrés doit être traduit en radians pour les calculs :

ANGLE_TOTAL *= DEG_TO_RAD = 180 * 0.011745329 = 2.11 rd

On peut en déduire l'angle de base en fonction de la vitesse de battement et le temps de base :

f_angle = ANGLE_TOTAL * VITESSE * TIME_BASE = 2.11 * 1 * 0.04 = 0.085 rd

Il faut transformer cet angle en valeur ramenée dans l'intervalle 0 à 1 pour l'interpolation :

fStep = f_angle / ANGLE_TOTAL = 0.04

Ensuite on calcule la position et la rotation de la bielle (point C)

v_pos_bielle = GetLinkPos(PRIM_BIELLE)
r_bielle = GetLinkRot(PRIM_BIELLE)

On en déduit la position du centre (point A) avec la demi-longueur de la bielle :

v_centre = v_pos_bielle - AXE_BIELLE * DEMIE_BIELLE * r_bielle

Et le vecteur tournant dans son état initial :

v_tournant = v_pos_bielle - v_centre

Il ne nous manque plus que l'axe de rotation ramené dans l'orientation de la bielle :

AXE_ROTATION *= r_bielle

Le script est maintenant en attente

6.6.2 Mise en route et arrêt

Le balayage démarre sur un **touch**.

```
// Inhibition touch si demande d'arrêt
if(iDemande_Arret) return;
// Gestion du touch
if(iOn = !iOn)
        llSetTimerEvent(TIME_BASE);
else {
        llWhisper(0, "Demande d'arrêt prise en compte");
        iDemande_Arret = TRUE;
}
```

Dans cet événement est pris en compte aussi bien le démarrage que l'arrêt de balayage.

Au départ la variable **iDemande_Arret** est à **FALSE** et cette instruction est ignorée.

La variable **iOn** est également à **FALSE** mais change ici de valeur et le timer est lancé avec le temps de base.

Au **touch** suivant la variable **iOn** étant à **TRUE** elle change de valeur et la variable **iDemande_Arret** est mise à **TRUE**.

La variable **iDemande_Arret** est utilisée dans la fonction **gestion_fin** dont le code ne présente aucune particularité.

6.6.3 Traitement du mouvement

Le traitement du mouvement se trouve dans l'événement **timer**.

```
// Test du sens
    if(iSens)
    {
        // Incrémentation de l'angle
        fEtat += fStep;
        // Test d'arrivée en position finale
        if(fEtat >= 1.0) gestion_fin(1.0);
    }
    else
    {
        // Décrémentation de l'angle
        fEtat -= fStep;
```

```
            // Test d'arrivée en position de départ
            if(fEtat <= .0) gestion_fin(.0);
        }
        // Rotation
        rotation rot = llAxisAngle2Rot(AXE_ROTATION, fCos(.0, ANGLE_TOTAL, fEtat));
        // Vecteur tournant
        vector vec = v_tournant * rot;
        // Positionnement bielle
        llSetLinkPrimitiveParamsFast(PRIM_BIELLE, [
            PRIM_POSITION, v_centre + vec, PRIM_ROT_LOCAL, r_bielle * rot]);
        // Positionnement balai
        llSetLinkPrimitiveParamsFast(PRIM_BALAI, [
            PRIM_POSITION, v_centre + vec * 2.0]);
        }
```

Passé la gestion du sens sans réelle difficulté on en arrive au calcul du mouvement.

On calcule la rotation de la bielle à partir de l'axe et de l'angle actuel en variation sinusoïdale :

```
rot = llAxisAngle2Rot(AXE_ROTATION, fCos(.0, ANGLE_TOTAL, fEtat))
```

D'où on en déduit le vecteur tournant (A-C) :

```
vec = v_tournant * rot
```

On peut alors positionner la bielle en ajoutant le vecteur tournant à la position du centre :

```
v_centre + vec
```

Et la rotation de la bielle qui est la somme de sa rotation initiale et la rotation partielle :

```
r_bielle * rot
```

Pour la position du balai c'est tout simple, on utilise le même vecteur que pour la bielle mais avec une norme doublée :

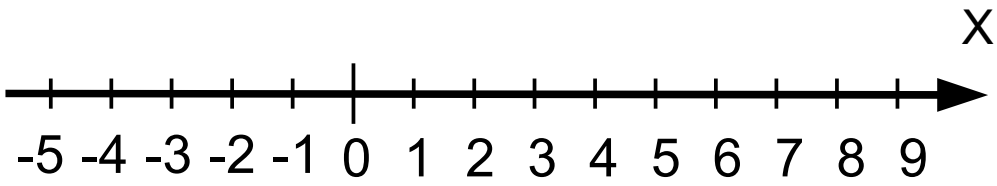
```
v_centre + vec * 2.0
```

7. Interlude vectoriel

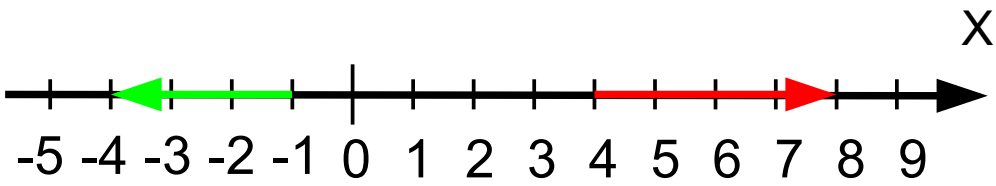
Les projets vus jusqu'à présent étaient plutôt simples à traiter et ne mettaient en œuvre que des notions mathématiques élémentaires. Les prochains chapitres demanderont de votre part plus d'attention. Non pas que nous allons rencontrer des choses très complexes mais le traitement fera appel à des notions sur les vecteurs un peu plus précises. C'est la raison d'être de ce chapitre intermédiaire. Les lecteurs déjà à l'aise avec les vecteurs peuvent sauter ce chapitre qui ne leur apportera pas grand-chose. Par contre il développe des notions essentielles à la compréhension des prochains chapitres.

7.1 Vecteur unitaire

Nous avons déjà beaucoup évoqué les vecteurs. Ceux que nous avons utilisés avaient une norme quelconque. Il existe des vecteurs particuliers qui ont pour norme l'unité. Autrement dit leur norme est égale à un. Quelle en est l'utilité ? Nous allons voir cela. Observez la figure :

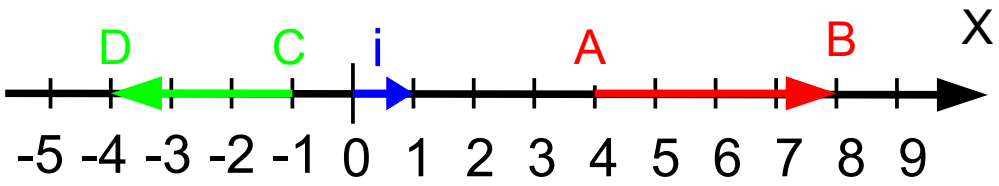


Il s'agit d'un axe X orienté avec une origine 0 et une graduation en centimètres. C'est un monde avec une seule dimension. On peut représenter des vecteurs sur cet axe :



Le vecteur rouge a pour direction l'axe X, comme sens le sens positif et pour norme 4 (8 – 4).
Le vecteur vert a pour direction l'axe X, comme sens le sens négatif et pour norme 3 (-1 – (- 4)).

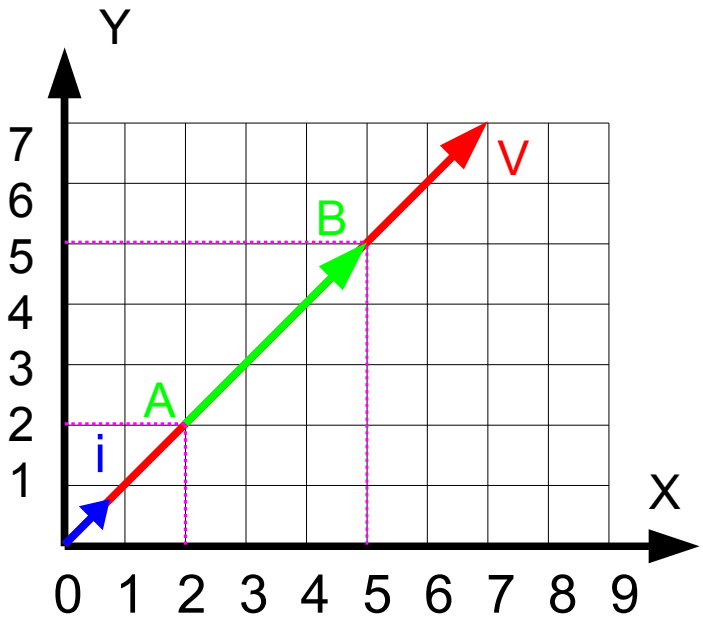
Définissons un vecteur unitaire sur cet axe :



Le vecteur bleu repéré « i » est un vecteur unitaire puisque son module est 1. D'autre part il se situe dans le sens positif de l'axe. Il est appelé vecteur unitaire de l'axe X. Nous avons les relations :

vecteur AB = 4 * i
vecteur CD = -3 * i

L'utilité de ce vecteur unitaire va apparaître plus clairement si nous considérons un plan et que nous considérons un axe dans ce plan :



Notre axe est l'axe V représenté en rouge. Le vecteur unitaire de cet axe est le vecteur « i ». La norme de ce vecteur est donc 1. Quelles sont les coordonnées de ce vecteur dans le repère orthonormé représenté ? Pythagore nous apprend facilement que c'est la racine de 2. Mais peu importe ce calcul puisque le **LSL** nous offre une fonction qui rend n'importe quel vecteur unitaire : **llVecNorm**. Donc notre vecteur unitaire est égal à :

Vecteur i = <0.707, 0.707>

Si nous considérons maintenant le vecteur AB situé sur l'axe V. Vous voyez facilement sur la figure les coordonnées des points A et B et donc le vecteur AB est facile à déterminer à partir de cette figure :

Vecteur AB = <5 – 2, 5 – 2> = <3, 3>

On en déduit facilement la norme de ce vecteur :

Norme AB = ||AB|| = sqr (3 * 3 + 3 * 3) = 4.24

On peut aussi dire :

Vecteur AB = Vecteur i * 4.24

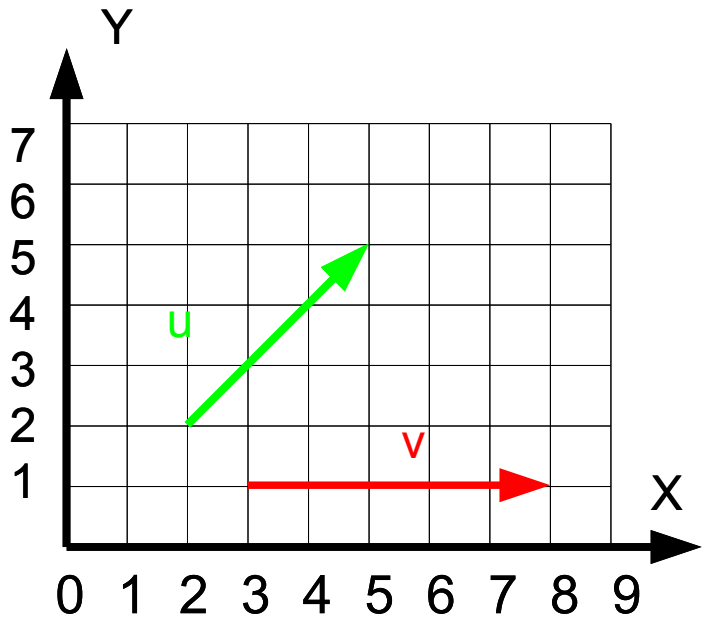
Vérifions cette relation :

<0.707, 0.707> * 4.24 = <3, 3>

Nous aurons bientôt l'usage de cette propriété.

7.2 Produit scalaire

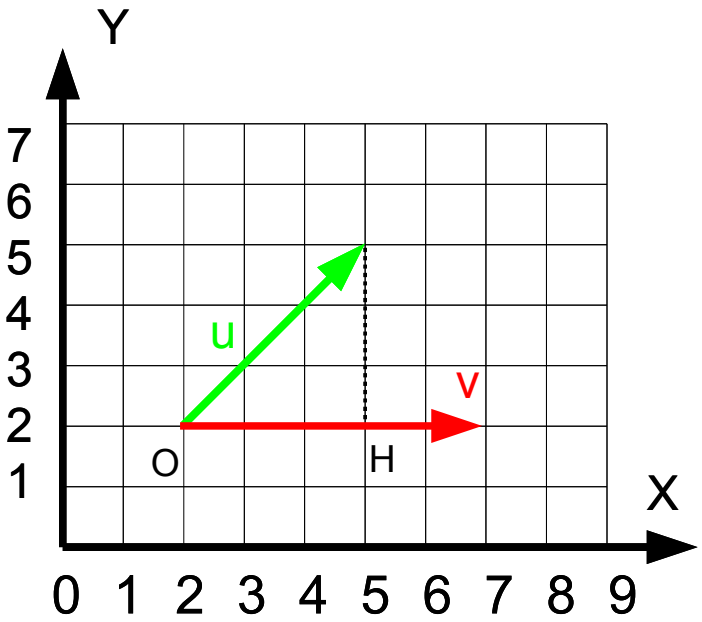
Que peut donner la multiplication d'un vecteur par un autre vecteur ? La réponse intuitive n'est pas évidente. Nous allons voir qu'il y a deux façons de répondre à cette questions. Abordons la première. En général deux vecteurs n'ont pas la même direction, on peut les situer dans un plan :



Le produit scalaire des vecteurs u et v se définit ainsi :

$$\mathbf{u * v = ||u|| \,.\, ||v|| \,.\, cos(u,v)}$$

L'opérateur "*" est utilisé pour représenter le produit scalaire alors que le "." sert à représenter la multiplication entre deux scalaires. Donc le produit scalaire est égal au produit des deux modules par la cosinus de l'angle entre les deux vecteurs. Modifions la figure pour représenter cela :



Si nous considérons l'expression **||u|| . cos(u,v)** on constate facilement qu'il s'agit de la projection de u sur la direction de v, autrement dit le segment OH. On peut donc réécrire le produit sclaire ainsi :

$$\mathbf{u * v = OH . ||v||}$$

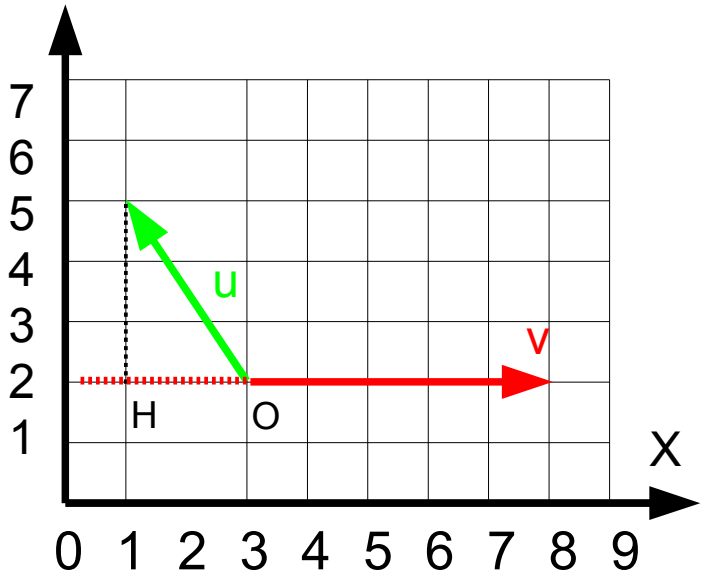
On peut faire la calcul dans notre cas :

$$u * v = 3 . 5 = 15$$

Il saute aux yeux que l'on a (commutativité):

$$\mathbf{u * v = v * u}$$

Un produit scalaire peut-il être négatif ? Observez la figure :



Si on considère la direction support du vecteur v, celle-ci est orientée et OH est négatif de valeur -2. D'où un produit scalaire négatif :

$$u * v = -2 . 5 = -10$$

Une autre propriété importante :

$$\mathbf{u * u = u^2 = ||u||^2}$$

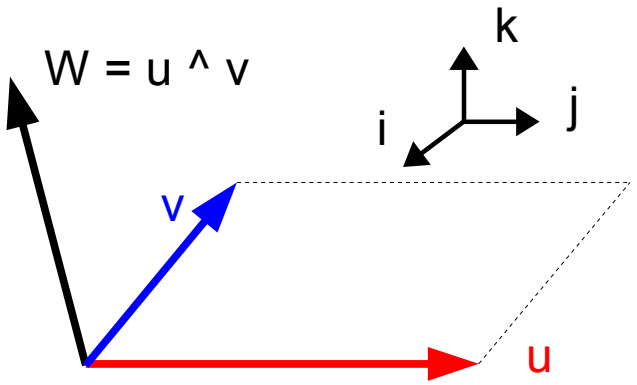
D'autre part si on considère que v est un vecteur unitaire on a :

$$u * v = ||u|| . \cos(u,v)$$

Nous utiliserons cette particularité dans le prochain projet.

7.3 Produit vectoriel

Considérons une deuxième façon de multiplier 2 vecteurs. Cette fois la réprésentation en deux dimensions ne nous suffit plus. Observez cette figure :



On veut faire le produit vectoriel de u par v. Ces deux vecteurs étant quelconques, donc pas forcément avec des directions perpendiculaires comme la figure pourrait le suggérer. Le résultat du produit vectoriel entre u et v est aussi un vecteur dont :

- la direction est perpendiculaire au plan (u,v) donc forcément à chacun des deux vecteurs u et v,
- le sens dépend de la base dans laquelle on se situe, le trièdre (u, v, w) doit avoir la même orientation que la base (i, j, k),
- la norme est égale à la surface du parallélogramme construit sur (u, v), ce qui donne :

$$||\mathbf{u} \wedge \mathbf{v}|| = ||\mathbf{u}|| \,.\, ||\mathbf{v}|| \,.\, |\mathbf{sin(u, v)}|$$

Propriétés du produit vectoriel :

- antisymétrie — $u \wedge v = -(v \wedge u)$
- k est un scalaire — $k(u \wedge v) = ku \wedge v$
- double produit vectoriel — $u \wedge (v \wedge w) = u \wedge v + u \wedge w$
- distributivité — $u \wedge (v + w) = v(u.w) - w(u.v)$
- colinéarité — $u \wedge v =$ vecteur nul

8. Bielle-manivelle (1)

Ce projet commence à présenter quelques difficultés. C'est pour cette raison qu'il va être suffisamment détaillé. Il existe plusieurs façons de le traiter. Essentiellement au moyen de calcul trigonométrique ou vectoriel. L'expérience montre que la deuxième façon de procéder se révèle à terme plus efficace dans un espace en 3 dimensions. C'est pour cette raison que les calculs seront orientés sur des éléments vectoriels.

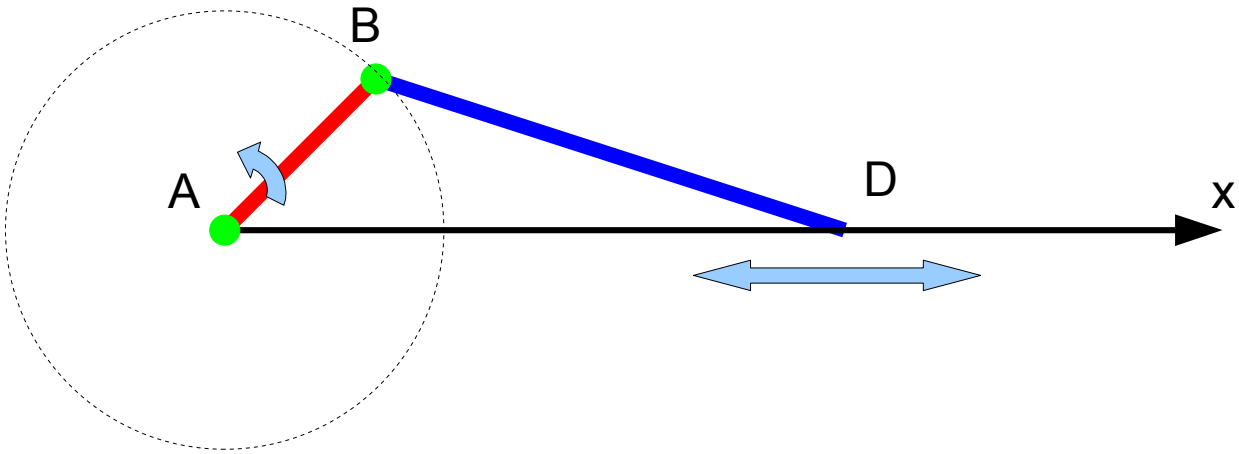
Ce projet à fait l'objet d'un fil de discussion sur le forum de JOL :

http://forums.jeuxonline.info/showthread.php?t=1081201

BalckCats y présente une démonstration mathématique très pertinente pour la résolution du problème. J'adopte dans ce manuel une approche plus pragmatique pour ne pas dérouter ceux que les mathématiques effraient.

8.1 Enoncé du projet

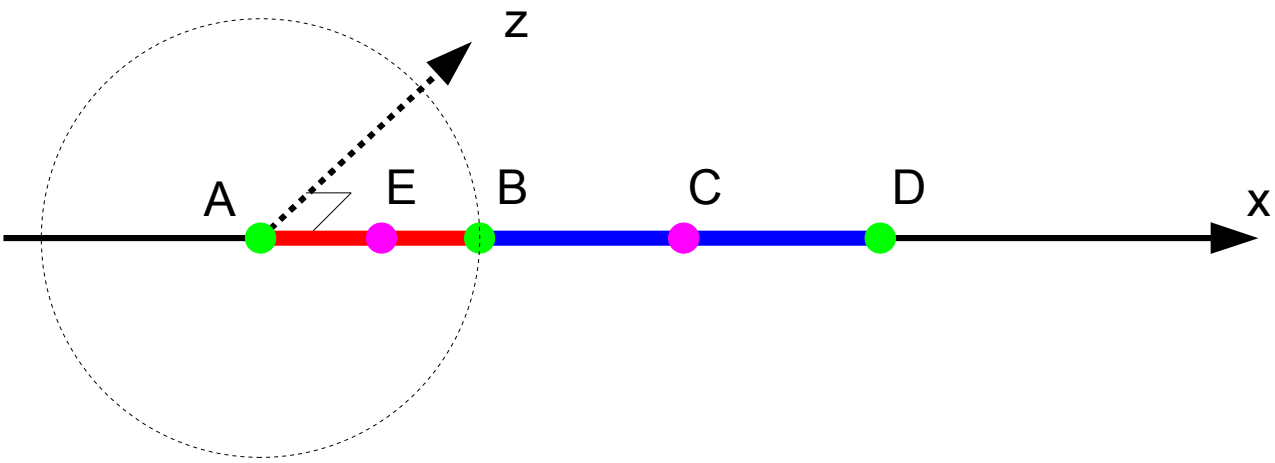
Le but de ce système est de transformer un mouvement circulaire en translation et réciproquement.



La manivelle (rouge) tourne dans le sens de la flèche. Le centre de rotation est le point A. B décrit le cercle figuré en pointillés. L'extrémité D de la bielle (bleue) a comme contrainte de devoir rester sur l'axe x. La manivelle et la bielle sont articulées en B. Donc le mouvement rotatif de la manivelle entraine un mouvement de tranlation de l'extrémité D de la bielle, cette translation alternative est représentée par une double flèche.

8.2 Paramètres

La première chose à faire est de définir la position de départ. Nous allons positionner tous les éléments dans un certain état et lancer le script :



Au départ on aligne la bielle et la manivelle, ce qui va nous simplifier la détermination des paramètres. Que pouvons-nous connaître ? La position locale de la manivelle qui est le centre du segment AB (point E) et la position de la bielle qui est le centre du segment BD (point C).

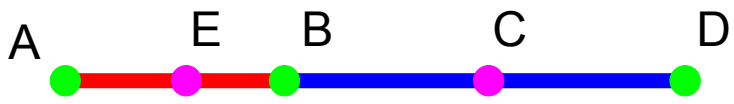
Il nous faut comme informations supplémentaires à entrer en paramètres :

- la longueur de la manivelle (du moins la distance entre les points A et B)
- la longueur de la bielle (du moins la longueur entre les points B et D)
- l'axe de rotation de la manivelle appelé ici Z
- le numéro de liaison de la manivelle
- le numéro de liaison de la bielle

8.3 Calculs préliminaires

8.3.1 Lecture position manivelle et bielle

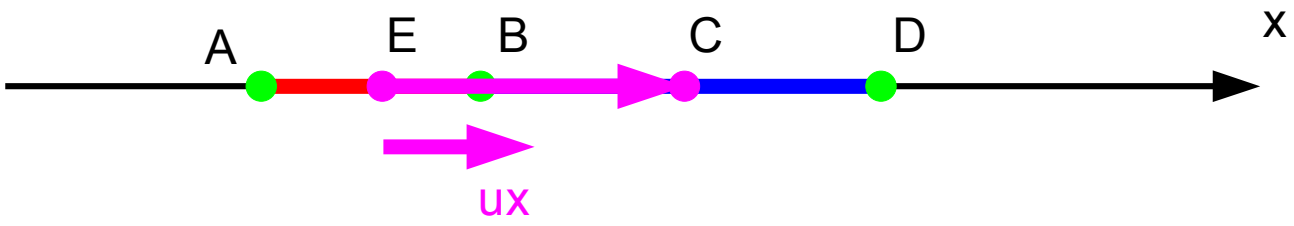
Nous avons déjà vu comment faire dans les projets précédents. Nous avons déjà la fonction qui donne la position locale d'une primitive enfant. Nous lisons donc les coordonnées des points C et E :



8.3.2 Axe x

Il nous faut déterminer l'axe x qui est le principal support du mouvement. Hors nous connaissons deux points situés sur cet axe : C et E. Avec ces deux points nous pouvons déterminer un vecteur et le rendre unitaire.

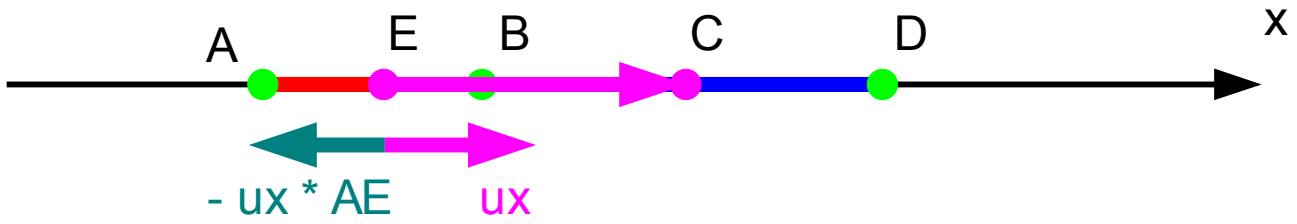
ux = (Point C – Point E) rendu unitaire



8.3.3 Centre de rotation de la manivelle

Maintenant que nous avons déterminé l'axe x il est facile de trouver le point A :

Point A = Point E – ux * ½ longueur de la manivelle



Sur la figure a été représenté le vecteur qui permet de passer du point E au point A. Nous voyons ici l'utilité d'un vecteur unitaire sur un axe.

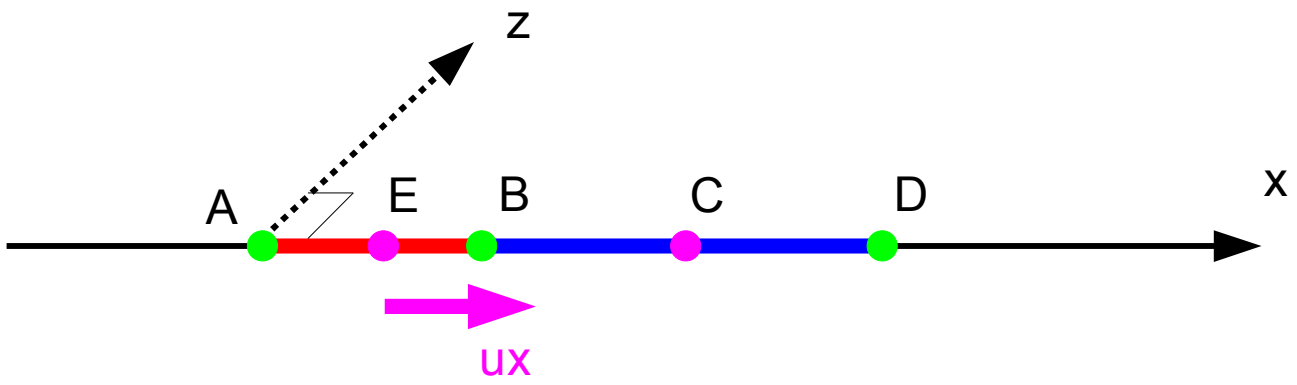
8.3.4 Rotation de la manivelle et de la bielle

L'axe x est quelconque par rapport aux axes de base de la primitive racine. De même la manivelle et la bielle ont une rotation quelconque par rapport à ce référentiel de base. Nous avosn besoind e connaître ces rotations de départ. Il suffit de les calculer avec la fonction que nous avons déjà utilisée qui donne la rotation locale d'une primitive enfant.

8.3.5 Axe de rotation de la manivelle (axe z)

Nous avons vu que nous entrons l'axe de rotation de la manivelle dans les paramètres. Mais cet axe est local à la manivelle. Il nous faut connaître l'orientation de cet axe dans le référentiel du la primitive racine. On le fait en multipliant l'axe par la rotation de la primitive enfant, comme nous l'avons déjà réalisé dans les projets précédents :

Axe de rotation de la manivelle = Axe de rotation de la manivelle * Rotation de la bielle



Cet axe est perpendiculaire à l'axe x.

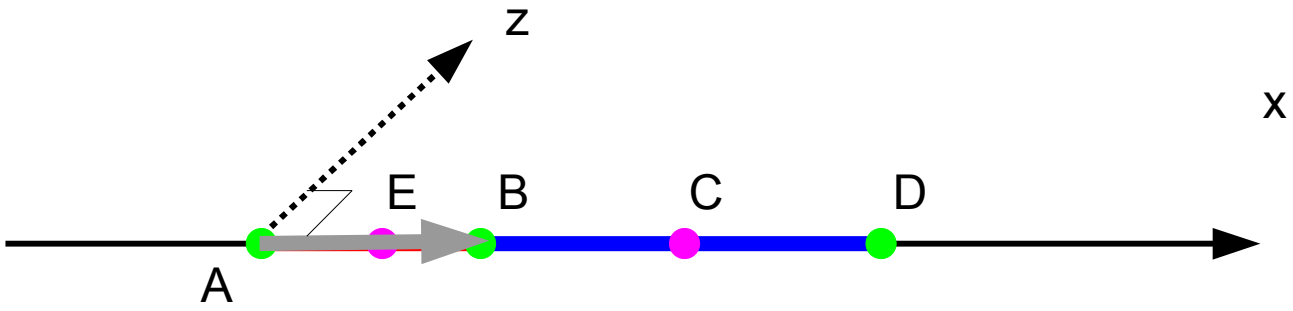
8.3.6 Angle élémentaire

En fonction du temps de base et de l'axe de rotation z on peut en déduire facilement l'angle correspondant. Nous l'avons déjà fait pour les projets précédents

8.3.7 Vecteur tournant

Le vecteur tournant pour la manivelle est le vecteur AB. Pour le déterminer on fait à nouveau appel au vecteur unitaire ux :

Vecteur AB = ux * Longueur Manivelle



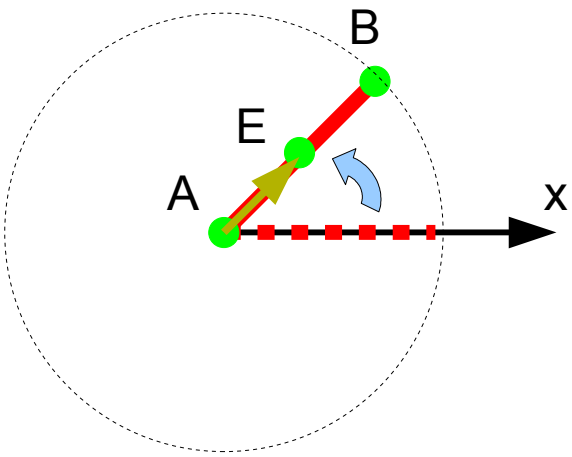
Il est représenté ici en grisé.

Tous les calculs priliminaires sont maintenant effectués.

8.4 Mouvement

8.4.1 Rotation de la manivelle

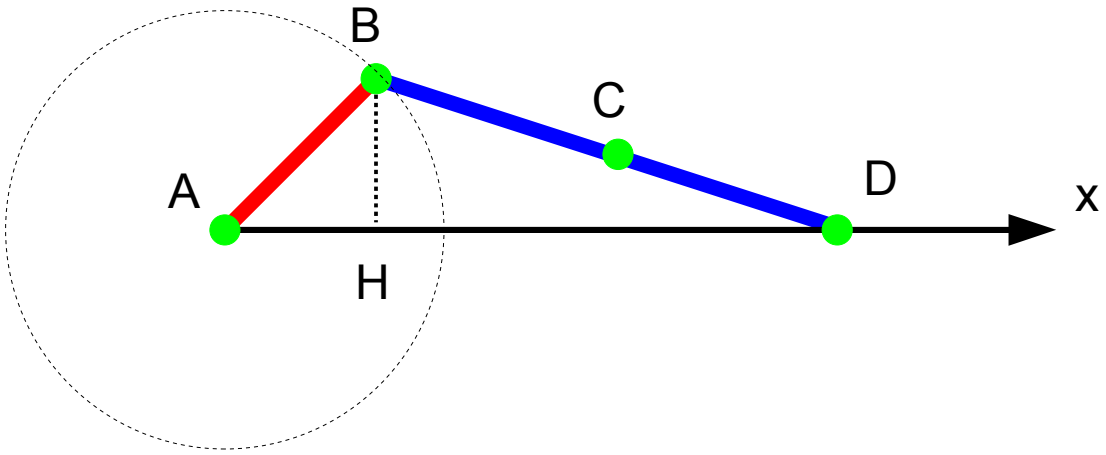
La rotation de la manivelle est composée d'une rotation et d'un repositionnement, ce que nous avons appelé précédemment déplacement circulaire. Il faut faire tourner le point E (centre de la manivelle) autour du point A (centre de rotation).



Il faut donc faire tourner le vecteur AE. Nous avons déjà réalisé cela dans les projets précédents. On cumule l'angle élémentaire dans une variable pour obtenir la rotation depuis l'état de départ. Cette rotation est appliquée au vecteur et aussi directement à la manivelle pour quelle tourne en même temps qu'elle se repositionne.

8.4.2 Mouvement de la bielle

Pour la bielle c'est un peu plus délicat.



Le but est de déterminer la position du point C et la rotation de la bielle par rapport à l'axe x. Voyons cela.

On peut connaître la position du point B grâce au vecteur tournant utilisé pour la rotation de la manivelle. Par contre on ne connaît pas la position du point D à part qu'il se trouve quelque part sur l'axe x.

Considérons le vecteur AB. Il constitue l'un des côté du triangle rectangle ABH rectangle en H qui est la projection du point B sur l'axe x. Si nous continuons à raisonner avec des vecteurs nous connaissons le vecteur AB qui a été calculé pour la rotation de la manivelle. Pouvons-nous connaître AH ?

Le produit scalaire du vecteur AB avec le vecteur unitaire ux est justement AH.

$$AH = \text{Vecteur AB} * ux$$

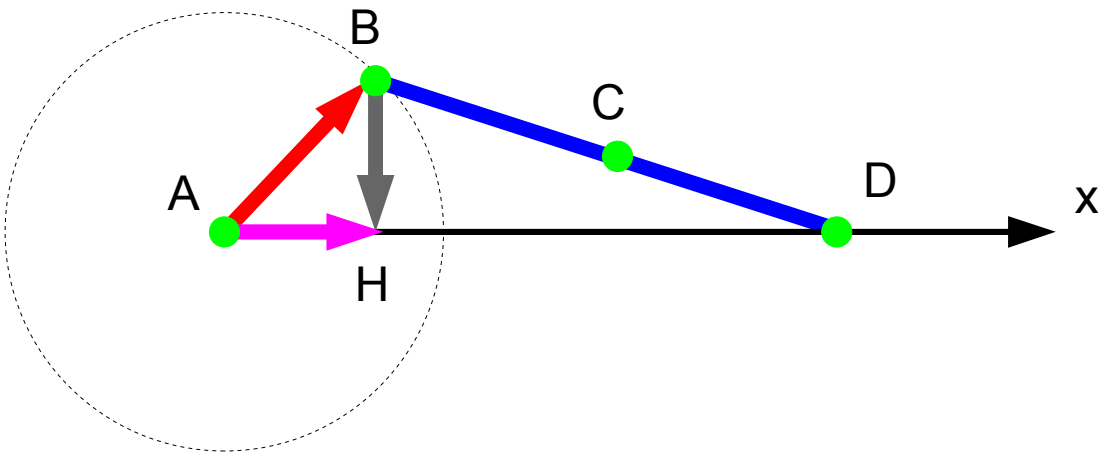
Il s'agit là d'une valeur scalaire. Comment obtenir le vecteur AH ? Tout simplement en multipliant ce scalaire par le vecteur unitaire ux :

Vecteur AH = (Vecteur AB * ux) . ux

Pour éviter les confusions l'opérateur “*” est utilisé pour le produit scalaire et “.” pour le produit entre un vecteur et un scalaire. Lorsqu'on utilise le LSL on a seulement l'opérateur “*” pour toutes les multiplications quels que soient les opérandes en rpésence, ce qui n'est pas forcément d'une grande clarté.

Maintenant qu'on a les vecteurs AB et AH il est facile de déterminer le vecteur BH :

Vecteur BH = Vecteur AH – vecteur AB



Si on considère le triangle BDH il est rectangle en H. On peut y appliquer le théorème de Pythagore. On connait la longueur BD (longueur de la bielle) et la longueur BH (norme du vecteur BH). Donc :

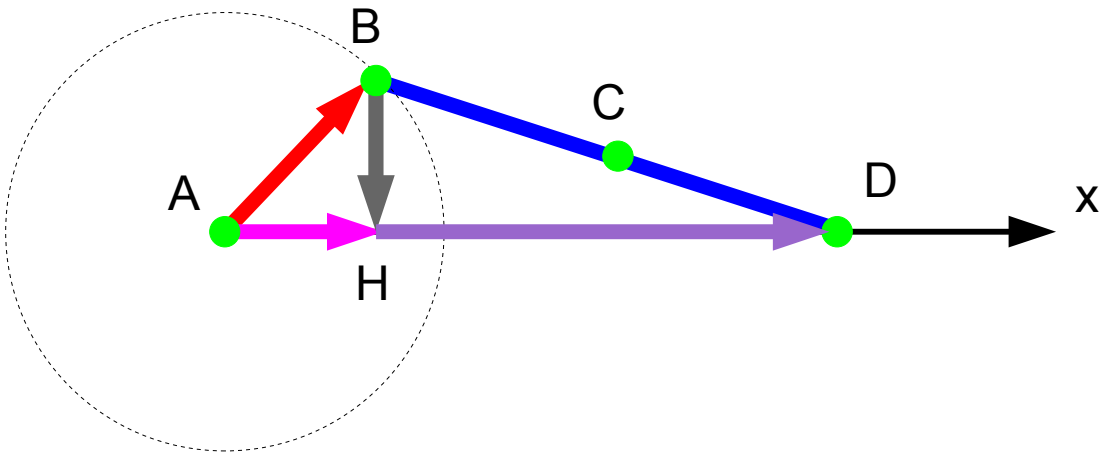
$$HD = sqr (BD^2 – BH^2)$$

On peut ainsi connaître la position du point D. Il suffit de prendre le vecteur HD :

$$\text{Vecteur } HD = sqr (BD^2 – BH^2) . ux$$

D'où :

$$\text{Vecteur } AD = \text{Vecteur } AH + \text{Vecteur } HD = (\text{Vecteur } AB * ux) . ux + sqr (BD^2 – BH^2) . ux$$

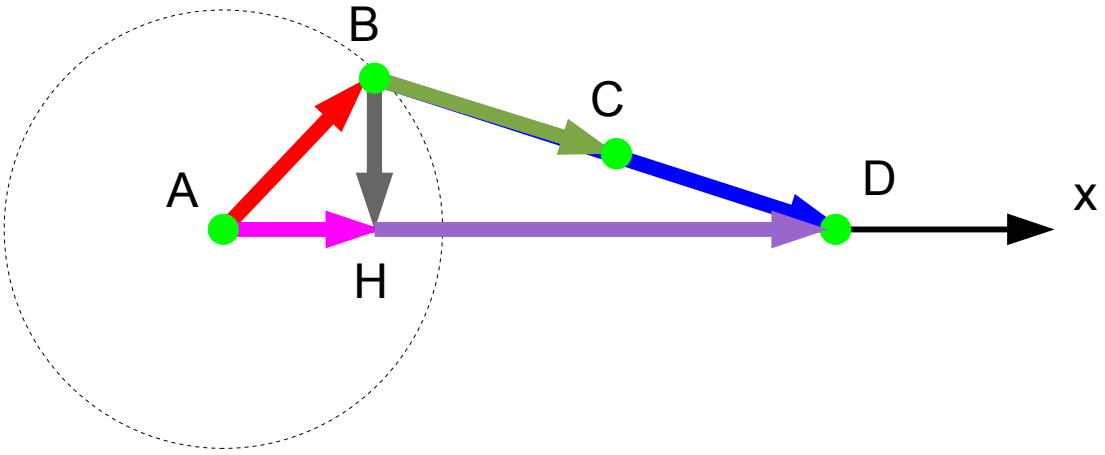


Nous avons bien avancé ! Il ne nous manque plus que le vecteur BD, facile à déterminer :

$$\text{Vecteur BD} = \text{Vecteur AD} - \text{Vecteur AB}$$

On en déduit facilement la position du point C qui se trouve à mi-distance entre B et D :

$$\text{Point C} = \text{Point B} + \text{Vecteur BC} = \text{Point B} + \tfrac{1}{2}\ \text{Vecteur BD}$$



Pour finir la rotation de la bielle nous est donnée en extrayant la rotation entre le vecteur ux et le vecteur BD.

8.4.3 Codage

```
// ------------------------
//    Constantes
// ------------------------
// Durée de base en s
float TIME_BASE = .04;
// Vitesse en tours/s
float VITESSE = .4;
```

```
// Longueur de la bielle (entre axes)
float LONGUEUR_BIELLE = 1.0;
// Axe de rotation de la manivelle
vector AXE_ROTATION = <1.0, .0, .0>;
// Longueur de la manivelle (entre axes)
float LONGUEUR_MANIVELLE = .5;
// Numéro de liaison de la primitive de la bielle
integer PRIM_BIELLE = 3;
// Numéro de liaison de la primitive de la manivelle
integer PRIM_MANIVELLE = 2;

// ------------------------
//    Variables globales
// ------------------------
// Centre (point A)
vector v_A;
// Vecteur unitaire axe x
vector v_ux;
// Rotation de la bielle
rotation r_bielle;
// Rotation de la manivelle
rotation r_manivelle;
// Vecteur tournant
vector v_tournant;
// Carré de longueur bielle
float f_carre_bielle;
// Angle de base
float f_elementaire;
// Angle cumulé
float f_angle;

// ------------------------
//    Fonctions - Librairie
// ------------------------
// Récupération position locale prim enfant
vector GetLinkPos(integer link_num) {
        vector v = llList2Vector(llGetLinkPrimitiveParams(link_num, [PRIM_POSITION]), 0);
        return (v - llGetRootPosition()) / llGetRootRotation();
}

// Récupération rotation locale prim enfant
rotation GetLinkRot(integer link_num) {
        return  llList2Rot(llGetLinkPrimitiveParams(link_num, [PRIM_ROT_LOCAL]), 0);
}

// ------------------------
//    Etat
// ------------------------
default
{
        state_entry()
```

```
{
        // Rotation de la manivelle
        r_manivelle = GetLinkRot(PRIM_MANIVELLE);
        // Rotation de la bielle
        r_bielle = GetLinkRot(PRIM_BIELLE);
        // Position bielle (point C)
        vector v_C = GetLinkPos(PRIM_BIELLE);
        // Position manivelle
        vector v_pos_manivelle = GetLinkPos(PRIM_MANIVELLE);
        // Axe de rotation de la manivelle
        AXE_ROTATION *= r_manivelle;
        // Angle de base
        f_elementaire = TWO_PI * VITESSE * TIME_BASE;
        // Vecteur unitaire axe x
        v_ux = llVecNorm(v_C - v_pos_manivelle);
        // Position du centre de rotation (point A)
        v_A = v_pos_manivelle - v_ux * LONGUEUR_MANIVELLE / 2.0;
        // Carré de longueur bielle
        f_carre_bielle = LONGUEUR_BIELLE * LONGUEUR_BIELLE;
        // Vecteur tournant
        v_tournant = v_ux * LONGUEUR_MANIVELLE;
}
touch_start(integer total_number)
{
        // Mise en route du timer
        llSetTimerEvent(TIME_BASE);
}
timer()
{
        // Calcul nouvel angle
        f_angle += f_elementaire;
        // Rotation
        rotation r_rot = llAxisAngle2Rot(AXE_ROTATION, f_angle);
        // Rotation du vecteur
        vector v_rot = v_tournant * r_rot;
        // Position extrémité manivelle (point B)
        vector v_B = v_A + v_rot;
        // Projection vecteur tournant sur axe x
        vector v_AH = v_rot * v_ux * v_ux;
        // Distance BH
        float f_BH = llVecMag(v_AH - v_rot);
        // Distance HD
        float f_HD = llSqrt(f_carre_bielle - f_BH * f_BH);
        // Vecteur v_HD
        vector v_HD = f_HD * v_ux;
        // Position extrémité bielle
        vector v_D = v_A + v_AH + v_HD;
        // Vecteur axe bielle
        vector v_bielle = v_D - v_B;
        // Position centre bielle (point C)
        vector v_C = v_B + v_bielle / 2.0;
```

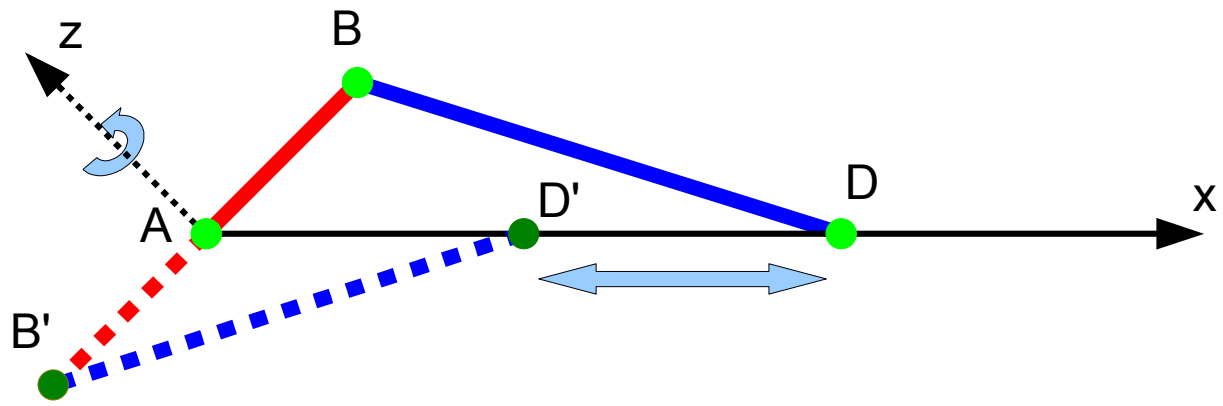
```
                // Rotation de la bielle
                rotation r_rot_bielle = r_bielle * llRotBetween(v_ux, v_bielle);

                // Rotation manivelle
                llSetLinkPrimitiveParamsFast(PRIM_MANIVELLE, [
                        PRIM_ROT_LOCAL, r_manivelle * r_rot,
                        PRIM_POSITION, v_A + v_rot / 2.0]);
                // Ajustement de la bielle
                llSetLinkPrimitiveParamsFast(PRIM_BIELLE, [
                        PRIM_ROT_LOCAL, r_rot_bielle,
                        PRIM_POSITION, v_C]);
        }
}
```

Le code a été abondamment commenté et correspond exactement aux éléments détaillés au-dessus.

9. Bielle-manivelle (2)

9.1 Généralisation 1

Le fait d'avoir utilisé des vecteurs pour traiter le problème trouve tout son intérêt si nous changeons un élément. Nous avons envisagé que la manivelle et la bielle évoluent dans un même plan, ce qui permet de raisonner en 2 dimensions pour le mouvement. Que se passerait-il si ce n'était plus le cas ?



On a changé l'orientation de la manivelle qui n'est plus alignée avec la bielle. Du coup la bielle va faire des mouvements différents. Les deux positions extrêmes ont été réprésentées sur la figure.

Il n'est plus possible de se fonder sur une position initiale particulière pour déterminer l'axe x. Du coup nous avons besoin de deux nouveaux paramètres :

- l'axe de la longueur de la manivelle
- l'axe de la longueur de la bielle

On peut alors définir le vecteur tournant :

Vecteur tournant = Axe Longueur Manivelle * Rotation Manivelle * Longueur Manivelle

Cela nous permet de déterminer le point A :

Point A = Position Manivelle – Vecteur Tournant / 2

D'où on en déduit la position du point D :

Point D = Position Bielle + Axe Longueur Bielle * Rotation Bielle * Longueur Bielle / 2

Pour le reste c'est pratiquement la même chose au niveau codage sauf pour la détermination de la rotation de la bielle dont la référence n'est plus la même. Dans le cas précédent nous avons mémorisé la rotation de départ qui est la même que l'axe x puisque elle est alignée dessus, il suffit alors d'ajouter la rotation entre le vecteur BD et

l'axe x. Mais maintenant La bielle n'est plus alignée sur l'axe x au départ. Il nous faut donc définir la rotation de cet axe x en coordonnées locales.

Voici le codage :

```
// ------------------------
//    Constantes
// ------------------------
// Durée de base en s
float TIME_BASE = .04;
// Vitesse en tours/s
float VITESSE = .4;
// Longueur de la bielle (entre axes)
float LONGUEUR_BIELLE = 1.0;
// Axe de rotation de la manivelle
vector AXE_ROTATION = <1.0, .0, .0>;
// Longueur de la manivelle (entre axes)
float LONGUEUR_MANIVELLE = .5;
// Numéro de liaison de la primitive de la bielle
integer PRIM_BIELLE = 3;
// Numéro de liaison de la primitive de la manivelle
integer PRIM_MANIVELLE = 2;
// Axe de la longueur de la manivelle
vector AXE_LONGUEUR_MANIVELLE = <.0, .0, -1.0>;
// Axe de la longueur de la bielle
vector AXE_LONGUEUR_BIELLE = <.0, .0, -1.0>;

// ------------------------
//    Variables globales
// ------------------------
// Centre (point A)
vector v_A;
// Vecteur unitaire axe x
vector v_ux;
// Rotation de la bielle
rotation r_bielle;
// Rotation de la manivelle
rotation r_manivelle;
// Vecteur tournant
vector v_tournant;
// Carré de longueur bielle
float f_carre_bielle;
// Angle de base
float f_elementaire;
// Angle cumulé
float f_angle;

// ------------------------
//    Fonctions - Librairie
// ------------------------
```

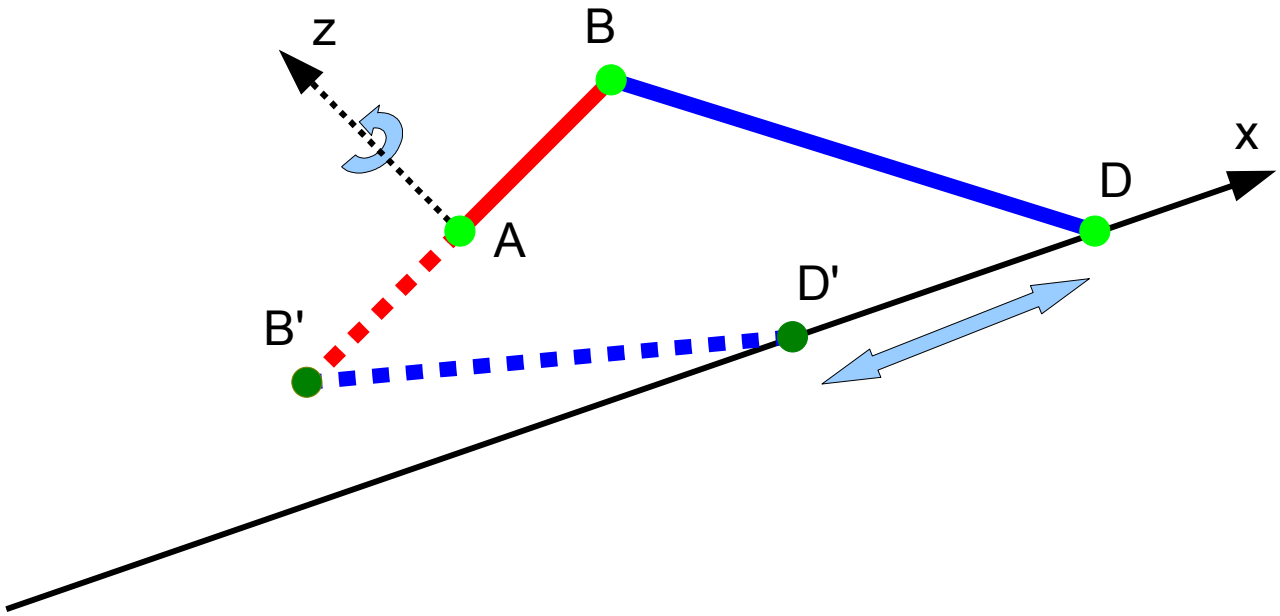
```
// Récupération position locale prim enfant
vector GetLinkPos(integer link_num) {
   vector v = llList2Vector(llGetLinkPrimitiveParams(link_num, [PRIM_POSITION]), 0);
   return (v - llGetRootPosition()) / llGetRootRotation();
}

// Récupération rotation locale prim enfant
rotation GetLinkRot(integer link_num) {
   return  llList2Rot(llGetLinkPrimitiveParams(link_num, [PRIM_ROT_LOCAL]), 0);
}

// ------------------------
//    Etat
// ------------------------
default
{
   state_entry()
   {
      // Rotation manivelle
      r_manivelle = GetLinkRot(PRIM_MANIVELLE);
      // Rotation bielle
      r_bielle = GetLinkRot(PRIM_BIELLE);
      // Position bielle (point C)
      vector v_C = GetLinkPos(PRIM_BIELLE);
      // Position manivelle
      vector v_pos_manivelle = GetLinkPos(PRIM_MANIVELLE);
      // Axe de rotation de la manivelle
      AXE_ROTATION *= r_manivelle;
      // Angle de base
      f_elementaire = TWO_PI * VITESSE * TIME_BASE;
      // Vecteur tournant
      v_tournant = AXE_LONGUEUR_MANIVELLE
                                    * r_manivelle * LONGUEUR_MANIVELLE;
      // Position du centre de rotation (point A)
      v_A = v_pos_manivelle - v_tournant / 2.0;
      // Position du point D extrémité de la bielle
      vector v_D = v_C + AXE_LONGUEUR_BIELLE * r_bielle * LONGUEUR_BIELLE / 2.0;
      // Vecteur unitaire axe x
      v_ux = llVecNorm(v_D - v_A);
      // Carré de longueur bielle
      f_carre_bielle = LONGUEUR_BIELLE * LONGUEUR_BIELLE;
      // On ramène la rotation de la bielle alignée avec l'axe x
      r_bielle /= llRotBetween(v_ux, AXE_LONGUEUR_BIELLE * r_bielle);
   }
   touch_start(integer total_number)
   {
      // Mise en route du timer
      llSetTimerEvent(TIME_BASE);
   }
   timer()
```

```
    {
        // Calcul nouvel angle
        f_angle += f_elementaire;
        // Rotation
        rotation r_rot = llAxisAngle2Rot(AXE_ROTATION, f_angle);
        // Rotation du vecteur
        vector v_rot = v_tournant * r_rot;
        // Position extrémité manivelle (point B)
        vector v_B = v_A + v_rot;
        // Projection vecteur tournant sur axe x
        vector v_AH = v_rot * v_ux * v_ux;
        // Distance BH
        float f_BH = llVecMag(v_AH - v_rot);
        // Distance HD
        float f_HD = llSqrt(f_carre_bielle - f_BH * f_BH);
        // Vecteur v_HD
        vector v_HD = f_HD * v_ux;
        // Position extrémité bielle
        vector v_D = v_A + v_AH + v_HD;
        // Vecteur axe bielle
        vector v_bielle = v_D - v_B;
        // Position centre bielle (point C)
        vector v_C = v_B + v_bielle / 2.0;
        // Rotation de la bielle
        rotation r_rot_bielle = r_bielle * llRotBetween(v_ux, v_bielle);

        // Rotation manivelle
        llSetLinkPrimitiveParamsFast(PRIM_MANIVELLE, [
            PRIM_ROT_LOCAL, r_manivelle * r_rot,
            PRIM_POSITION, v_A + v_rot / 2.0]);
        // Ajustement de la bielle
        llSetLinkPrimitiveParamsFast(PRIM_BIELLE, [
            PRIM_ROT_LOCAL, r_rot_bielle,
            PRIM_POSITION, v_C]);
    }
}
```

Les différences ont été mises en gras. Vous pouvez constater qu'elles ne sont pas très importantes, principalement dues à la détermination des éléments de référence. Vous pouvez voir ici la puissance des vecteurs et des quaternions appliqués à des mouvements en 3 dimensions.

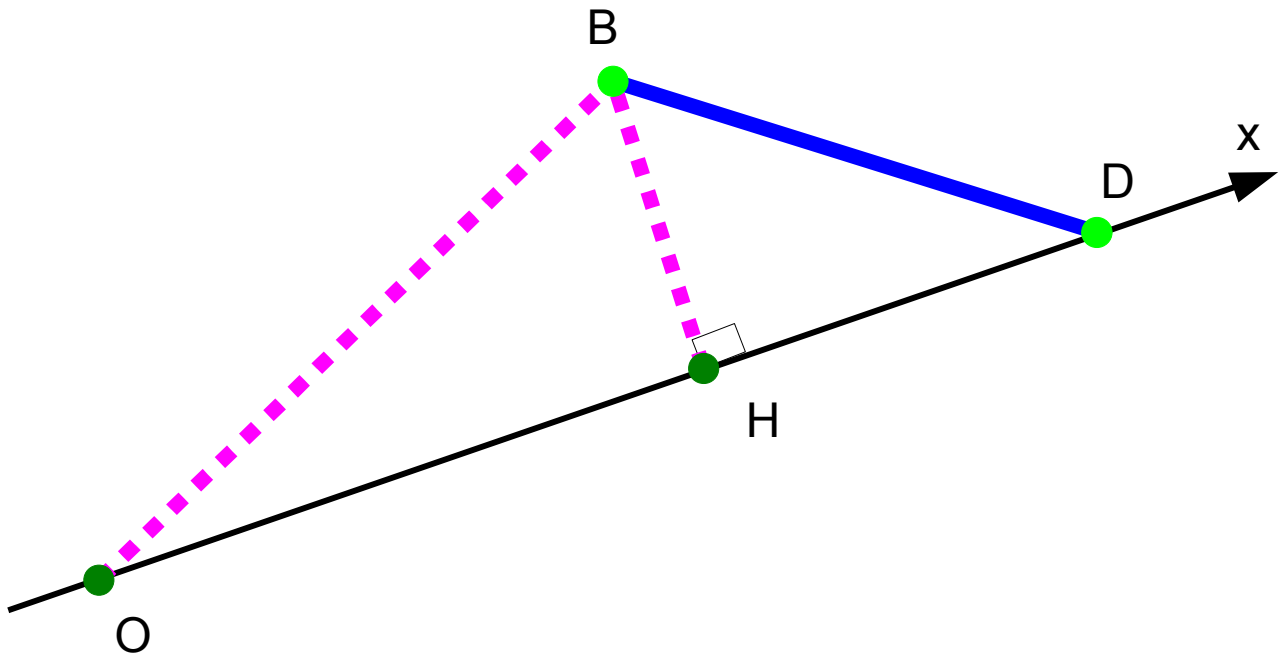
9.2 Généralisation 2

Allons plus loin dans la généralisation. Considérons maintenant un axe quelconque pour le déplacement de l'extrémité de la bielle. Jusqu'ici cet axe coupait le centre de rotation de la manivelle. Supposons que ce n'est plus le cas. Voici une représentation de cette situation :



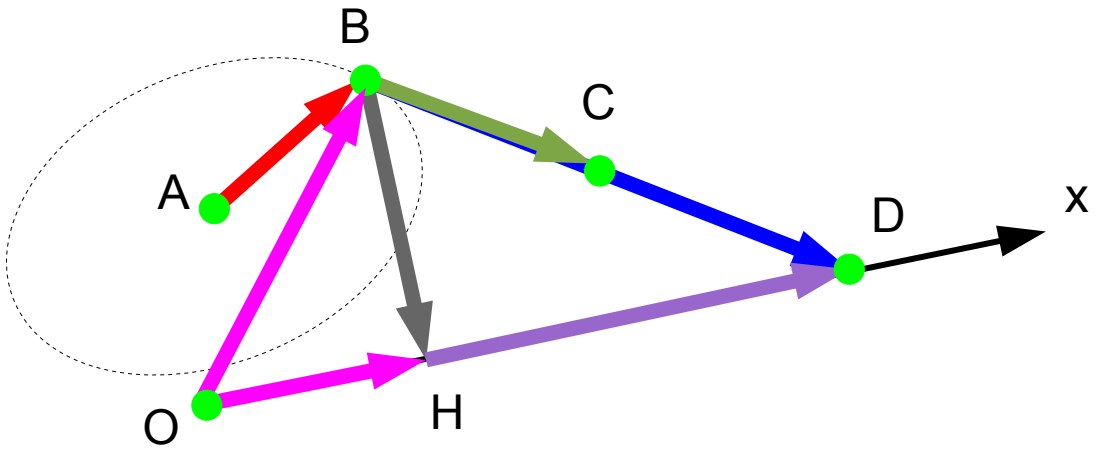
Cette fois l'axe X est choisi indépendamment des références concernant la manivelle. Qu'est-ce que cela change ?

Pour ramener le traitement à ce que nous avons vu précédemment nous allons considérer un point origine sur l'axe X :



Ce point O est pris de façon arbitraire, la seule condition étant qu'il se situe sur l'axe X, même confondu avec D ! On voit alors que le problème se ramène à ce que nous avons vu précédemment. Le point B étant toujours défini par la position de la manivelle.

Voici ce que ça donne au niveau traitement vectoriel :



On se rend compte que le seul changement est le remplacement du point A par le point O. Tout le reste est identique.

Pour simplifier la réalisation nous allons utiliser une primitive pour fixer l'axe X.

Voici le code correspondant :

```
// ------------------------
//    Constantes
// ------------------------
// Durée de base en s
float TIME_BASE = .04;
// Vitesse en tours/s
float VITESSE = .4;
// Longueur de la bielle (entre axes)
float LONGUEUR_BIELLE = 1.0;
// Axe de rotation de la manivelle
vector AXE_ROTATION = <1.0, .0, .0>;
// Longueur de la manivelle (entre axes)
float LONGUEUR_MANIVELLE = .5;
// Numéro de liaison de la primitive de la bielle
integer PRIM_BIELLE = 3;
// Numéro de liaison de la primitive de la manivelle
integer PRIM_MANIVELLE = 2;
// Numéro de liaison de la primitive de l'axe X
integer PRIM_AXE_X = 4;
// Axe de la longueur de la manivelle
vector AXE_LONGUEUR_MANIVELLE = <.0, .0, -1.0>;
// Axe de la longueur de la bielle
vector AXE_LONGUEUR_BIELLE = <.0, .0, -1.0>;
// Axe de la longueur de la primitive de l'axe X
vector AXE_LONGUEUR_AXE_X = <.0, .0, -1.0>;
```

```
// ------------------------
//    Variables globales
// ------------------------
// Centre (point A)
vector v_A;
// Référence (point O)
vector v_O;
// Vecteur unitaire axe x
vector v_ux;
// Rotation de la bielle
rotation r_bielle;
// Rotation de la manivelle
rotation r_manivelle;
// Vecteur tournant
vector v_tournant;
// Carré de longueur bielle
float f_carre_bielle;
// Angle de base
float f_elementaire;
// Angle cumulé
float f_angle;

// ------------------------
//    Fonctions - Librairie
// ------------------------
// Récupération position locale prim enfant
vector GetLinkPos(integer link_num) {
   vector v = llList2Vector(llGetLinkPrimitiveParams(link_num, [PRIM_POSITION]), 0);
   return (v - llGetRootPosition()) / llGetRootRotation();
}

// Récupération rotation locale prim enfant
rotation GetLinkRot(integer link_num) {
   return  llList2Rot(llGetLinkPrimitiveParams(link_num, [PRIM_ROT_LOCAL]), 0);
}

// ------------------------
//    Etat
// ------------------------
default
{
   state_entry()
   {
      // Rotation manivelle
      r_manivelle = GetLinkRot(PRIM_MANIVELLE);
      // Rotation bielle
      r_bielle = GetLinkRot(PRIM_BIELLE);
      // Rotation primitive de référence de l'axe X
      rotation r_axe_x = GetLinkRot(PRIM_AXE_X);
      // Position bielle (point C)
```

```
    vector v_C = GetLinkPos(PRIM_BIELLE);
    // Position manivelle
    vector v_pos_manivelle = GetLinkPos(PRIM_MANIVELLE);
    // Axe de rotation de la manivelle
    AXE_ROTATION *= r_manivelle;
    // Angle de base
    f_elementaire = TWO_PI * VITESSE * TIME_BASE;
    // Vecteur tournant
    v_tournant = AXE_LONGUEUR_MANIVELLE * r_manivelle * LONGUEUR_MANIVELLE;
    // Position du point D extrémité de la bielle
    vector v_D = v_C + AXE_LONGUEUR_BIELLE * r_bielle * LONGUEUR_BIELLE / 2.0;
    // Position du centre de rotation (point A)
    v_A = v_pos_manivelle - v_tournant / 2.0;
    // Vecteur unitaire axe x
    v_ux = llVecNorm(AXE_LONGUEUR_AXE_X * r_axe_x);
    // Position de la référence (point O), la valeur 2 est arbitraire
    v_O = v_D - v_ux * 2.0;
    // Carré de longueur bielle
    f_carre_bielle = LONGUEUR_BIELLE * LONGUEUR_BIELLE;
    // On ramène la rotation de la bielle alignée avec l'axe x
    r_bielle /= llRotBetween(v_ux, AXE_LONGUEUR_BIELLE * r_bielle);
}
touch_start(integer total_number)
{
    // Mise en route du timer
    llSetTimerEvent(TIME_BASE);
}
timer()
{
    // Calcul nouvel angle
    f_angle += f_elementaire;
    // Rotation
    rotation r_rot = llAxisAngle2Rot(AXE_ROTATION, f_angle);
    // Rotation du vecteur
    vector v_rot = v_tournant * r_rot;
    // Position extrémité manivelle (point B)
    vector v_B = v_A + v_rot;
    // Vecteur OB
    vector v_OB = v_B - v_O;
    // Projection vecteur OB sur axe x
    vector v_OH = v_OB * v_ux * v_ux;
    // Distance BH
    float f_BH = llVecMag(v_OH - v_OB);
    // Distance HD
    float f_HD = llSqrt(f_carre_bielle - f_BH * f_BH);
    // Vecteur HD
    vector v_HD = f_HD * v_ux;
    // Position extrémité bielle
    vector v_D = v_O + v_OH + v_HD;
    // Vecteur axe bielle
    vector v_bielle = v_D - v_B;
```

```
        // Position centre bielle (point C)
        vector v_C = v_B + v_bielle / 2.0;
        // Rotation de la bielle
        rotation r_rot_bielle = r_bielle * llRotBetween(v_ux, v_bielle);

        // Rotation manivelle
        llSetLinkPrimitiveParamsFast(PRIM_MANIVELLE, [
            PRIM_ROT_LOCAL, r_manivelle * r_rot,
            PRIM_POSITION, v_A + v_rot / 2.0]);
        // Ajustement de la bielle
        llSetLinkPrimitiveParamsFast(PRIM_BIELLE, [
            PRIM_ROT_LOCAL, r_rot_bielle,
            PRIM_POSITION, v_C]);
    }
}
```

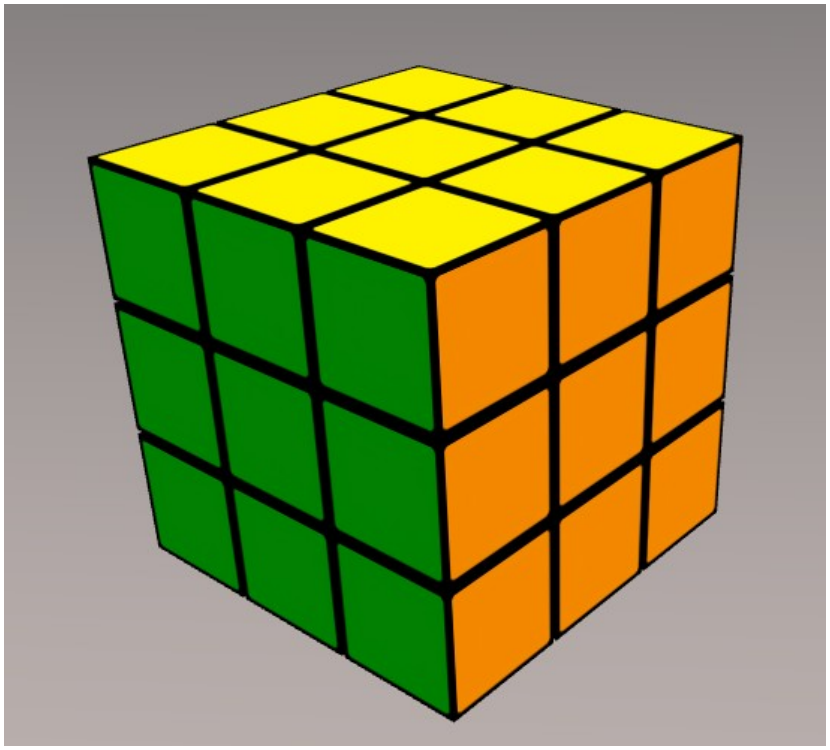
10. Rubik's cube

Ce projet ne comporte pas de réelle difficulté au niveau des rotations, mais elles sont multiples et la disposition quelconque du cube dans l'espace ne simplifie pas la tâche. D'autre part il faut gérer l'interface avec l'utilisateur de façon intuitive.

10.1 Enoncé du projet

Le cube est composé d'un élément central sur lequel sont articulés 26 petits cubes colorés. Il y a 6 couleurs de telle sorte qu'il est possible d'avoir une couleur par face. Le fait de manipuler le cube mélange ces couleurs. Le but est de reconstituer le cube avec ses faces colorées à partir de n'importe quel état. Dans notre projet nous allons envisager seulement l'aspect manipulation avec les rotations en laissant de côté le calcul des solutions possibles. On doit pouvoir faire tourner le cube globalement de 90° dans n'importe quel sens et également mettre en rotation de 90° n'importe quel étage de petits cubes.

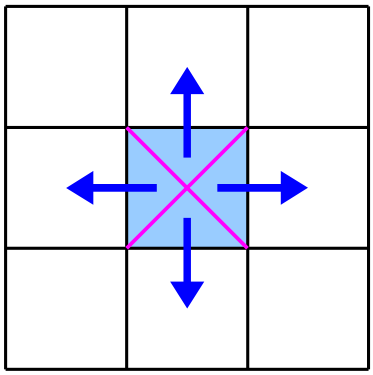
Au niveau de la constitution du cube la primitive racine doit être l'élément central et doit être aussi parfaitement centrée de telle façon que l'origine des axes se situe exactement au centre du cube. Il sera ainsi plus facile de gérer les rotations.



10.2 Interface intuitive

10.2.1 Rotation totale du cube

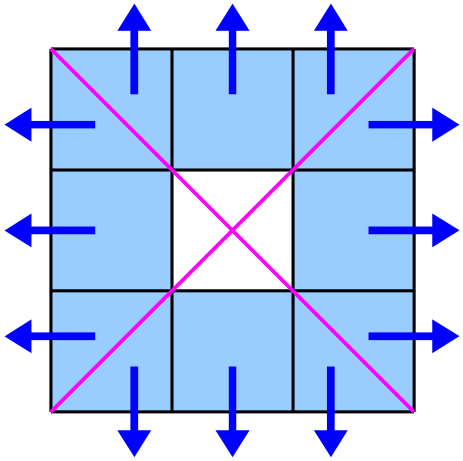
Un clic sur une des 6 facettes centrales doit provoquer la rotation totale de 90° du cube. L'axe et le sens de rotation nous sont donnés selon le point qui est cliqué en divisant la facette en 4 zones délimitées ci-dessous par les lignes magenta :



Les flèches bleues indiquent la direction de la rotation totale.

10.2.2 Rotation partielle

La rotation partielle (d'un étage de petits cubes) nous est donnée par un clic sur une facette externe. On adoptera le même mode de découpage pour les facettes angulaires qui peuvent gérer deux rotations différentes. Pour les facettes intermédiaires il n'y a aucune ambiguité.

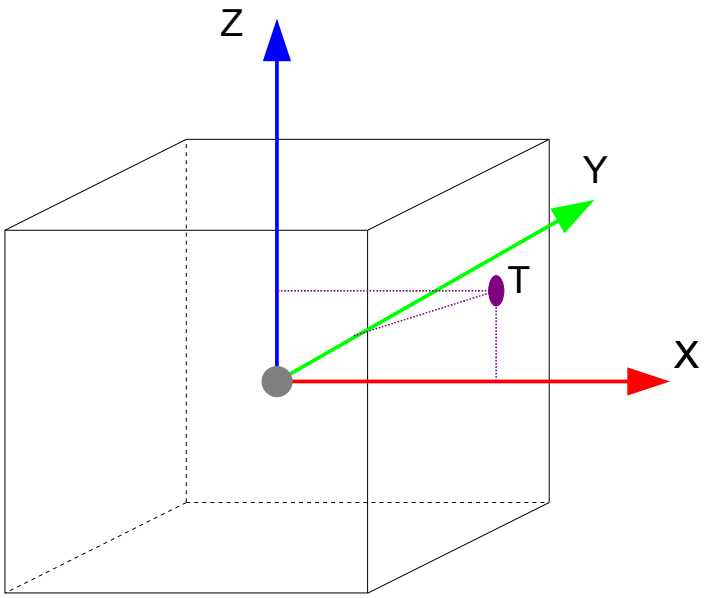


10.3 Détermination axe et sens de rotation

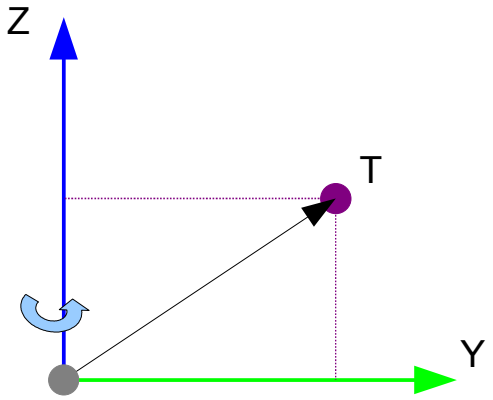
10.3.1 Axe de rotation

Le cube étant en 3 dimensions il faut lors d'un clic déterminer parmi les 3 axes possibles celui qui va constituer l'axe de rotation. On va le définir à partir du point cliqué. On va le faire en comparant les valeurs x, y et z, coordonnées du point touché.

Considérons la position du clic sur une face du cube et ses coordonnées sur les 3 axes :



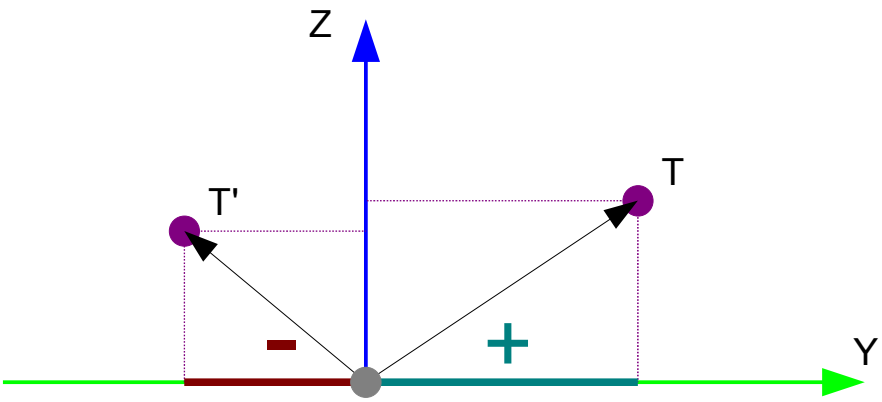
A partir du point touché T on crée des projections sur chacun des 3 axes (donc les produits scalaires). Si on compare les distances à l'origine on se rend compte que la plus grande est celle sur l'axe qui passe par la face touchée (ici l'axe X). Si on compare ensuite les deux valeurs restantes on constate que la plus faible nous indique l'axe de rotation. Ce n'est pas évident à voir sur le schéma en 3D mais devient évident dans une projection 2D :



Ici l'axe de rotation est de toute évidence l'axe Z.

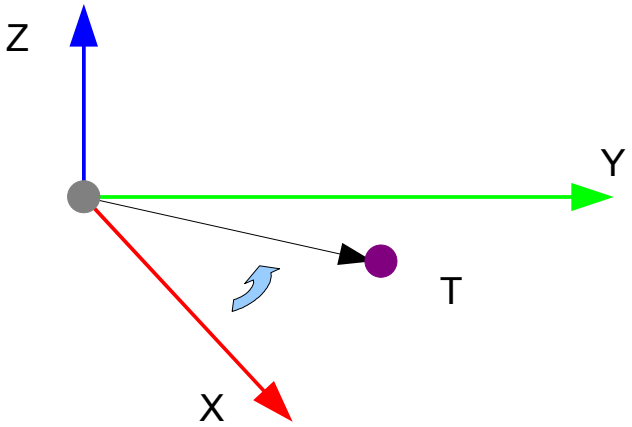
10.3.2 Sens de rotation

Les coordonnées nous indiquent-elles également le sens de rotation ? Reprenons la figure ci-dessus en considérant deux cas de touché :



Une coordonnée est positive ou négative selon la situation sur l'axe considéré. Si nous considérons le point T sa coordonnée sur l'axe Y est positive. Elle est par contre négative pour le point T'. Mais subsiste un problème : l'axe Y que nous prenons ici comme référence n'aura pas la même orientation selon le côté du cube que nous allons cliquer et il y a une ambiguité concernant le résultat.

Nous allons donc procéder différemment. Considérons l'angle entre le vecteur de la position du point touché et l'axe qui consitute la profondeur :



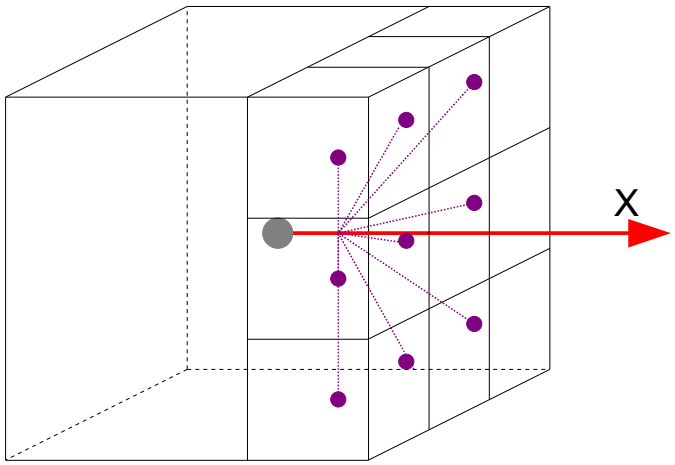
Le sens de la rotation nous est donné par le sens de cet angle. Comme nous pouvons savoir quel est l'axe de la profondeur (la plus grande valeur de coordonnée) le calcul devient simple. Il faut juste se méfier de l'orientation de cet axe en l'ajustant au besoin.

10.4 Détermination rotation partielle

Pour la rotation partielle, c'est-à-dire celle qui concerne uniquement un étage de petits cubes, la définition de l'axe et du sens de rotation tel que définis ci-deesus demeure valable. Il nous faut juste déterminer les petits cubes cocernés par la rotation.

Nous savons déjà que le cube touché fait partie de cet ensemble. Nous pouvons donc récupérer sa position et ensuite calculer le produit scalaire avec l'axe de rotation (ça revient à trouver la coordonnée sur cet axe). On peut

sans trop de risque dire que tous les cubes qui auront la même valeur de produit scalaire (en tenant compte des petits écarts de calcul ou de positionnement initial) constituent le groupe des cubes à mettre en mouvement :



Ici nous avons par exemple un étage de cubes qui vont tourner selon l'axe X. On voit bien que leurs coordonnées sur cet axe sont les mêmes.

10.5 Codage

Voici un code possible à partir des éléments définis ci-dessus :

```
// -----------------------------------------
//                Constantes Script
// -----------------------------------------
vector  AXE_X                   =       <1.0,.0,.0>;
vector  AXE_Y                   =       <.0,1.0,.0>;
vector  AXE_Z                   =       <.0,.0,1.0>;
integerROT_TOTAL                =       0;
integer ROT_PARTIEL             =       1;
float   TIME_BASE               =       .05;
float   ANGLE_TOTAL             =       PI_BY_TWO;
float   VITESSE                 =       .5;

// -----------------------------------------
//                Variables globales
// -----------------------------------------
list            l_box_centre_num;       // Cubes centraux numéro
list            l_box_angle_num;        // Cubes angulaires numéro
list            l_box_rot;              // Cubes à faire tourner
list            l_box_rot_vecteur;      // Vecteur tournant des cubes à faire tourner
list            l_box_rot_rot;          // Rotation des cubes à faire tourner
integer         i_inhibe;               // Inhibition touch pendant mouvement
integer         i_rot_type;             // Type de rotation : totale ou partielle
integer         i_box_number;           // Nombre de cubes à faire tourner (8 ou 9)
vector          v_axe;                  // Axe de rotation
integer         i_sens_normal;          // Sens de rotation
float           f_Step;                 // Incrément
float           f_Etat;                 // Etat intermédiaire
```

```
float           f_angle_fin;                    // Angle final
rotation        r_rot;                          // Rotation initiale

// ------------------------------------------------
//                  Fonctions librairie
// ------------------------------------------------

// Position Global -> Local
vector Global2Local(vector position) {
        return (position - llGetRootPosition()) / llGetRootRotation();}

// Récupération position locale prim enfant
vector GetLinkPos(integer link_num) {
        return Global2Local(llList2Vector(llGetLinkPrimitiveParams(link_num, [PRIM_POSITION]), 0));
}

// Récupération rotation locale prim enfant
rotation GetLinkRot(integer link_num) {
        return  llList2Rot(llGetLinkPrimitiveParams(link_num, [PRIM_ROT_LOCAL]), 0);
}

// Récupération couleur d'une face de prim enfant
vector GetColor(integer link_num, integer face) {
        return llList2Vector(llGetLinkPrimitiveParams(link_num, [PRIM_COLOR, face]), 0);
}

// Interpolation cos entre deux float
float fCos(float v0,float v1,float t){
        float f = (1.0 - llCos(t * PI)) / 2.0;
        return v0 * (1.0 - f) + v1 * f;
} // Nexii Malthus

// Retourne l'index dans une liste de la valeur
// correspondant au type statistique transmis
integer GetStat(list l_valeurs, integer i_type) {
        float f = llListStatistics(i_type, l_valeurs);
        return llListFindList(l_valeurs, [f]);
}

// ------------------------------------------------
//                  Fonctions du script
// ------------------------------------------------

// Récupération numéros de liaison des 26 cubes
// Détermination type = central ou angulaire
get_link_nums()
{
        // Nombre max
        integer n = llGetNumberOfPrims() + 1;
        // Variable de balayage
        integer i = 2;
```

```
        // Boucle
        for(;i < n; i++)
        {
                // Position locale
                vector v = GetLinkPos(i);
                // Test central
                integer i_central = (llFabs(v.x) < .01 && llFabs(v.y) < .01)
                        ||  (llFabs(v.x) < .01 && llFabs(v.z) < .01)
                        ||  (llFabs(v.y) < .01 && llFabs(v.z) < .01);
                // Mémorisation numéro
                if(i_central) l_box_centre_num += i;
                else l_box_angle_num += i;
        }
}

// Détermination cube central
integer is_central(integer link_num) {
        return llListFindList(l_box_centre_num, [link_num]) != -1;
}

// Trouve axe et sens pour la rotation
// selon la position touchée
TrouveAxeSens(vector pos) {
        // Recherche plus petite coordonnée (donne l'axe de rotation)
        integer i1 = GetStat([llFabs(pos.x), llFabs(pos.y), llFabs(pos.z)], LIST_STAT_MIN);
        // Axe de rotation
        v_axe = llList2Vector([AXE_X, AXE_Y, AXE_Z], i1);
        // Recherche plus grande coordonnée
        integer i2 = GetStat([llFabs(pos.x), llFabs(pos.y), llFabs(pos.z)], LIST_STAT_MAX);
        // Axe de référence profondeur
        vector v_axe_prof = llList2Vector([AXE_X, AXE_Y, AXE_Z], i2);
        // Ajustement sens de l'axe
        if(pos * v_axe_prof < .0) v_axe_prof = - v_axe_prof;
        // Angle entre référence et point touché (donne le sens de rotation)
        vector v_angle = llRot2Euler(llRotBetween(v_axe_prof, pos));
        // Sens de rotation
        if(i1 == 0) i_sens_normal = v_angle.x > .0;
        else if(i1 == 1) i_sens_normal = v_angle.y > .0;
        else i_sens_normal = v_angle.z > .0;
}

// Routine de balayage des cubes
BalayeCubes(integer i_max, list l_box, float f_base) {
        // Initialisation de variables
        integer index;
        vector v_pos;
        // Balayage
        while(i_max--) {
                // Index du cube
                index = llList2Integer(l_box, i_max);
                // Position du cube
```

```
                v_pos = GetLinkPos(index);
                // Test par produit scalaire sur axe de rotation
                if(llFabs(v_axe * v_pos - f_base) < .01) {
                        // Enregistrement index
                        l_box_rot += index;
                        // Enregistrement position
                        l_box_rot_vecteur += v_pos;
                        // Enregistrement rotation
                        l_box_rot_rot += GetLinkRot(index);
                }
        }
}

// Détermination des cubes concernés par la rotation
GetCubes(vector v_pos_box_touch) {
        // Initialisation des variables
        l_box_rot = [];
        l_box_rot_vecteur = [];
        l_box_rot_rot = [];
        // Produit scalaire de référence
        float f_base = v_axe * v_pos_box_touch;
        // Balayage des cubes angulaires
        BalayeCubes(20, l_box_angle_num, f_base);
        // Balayage des cubes centraux
        BalayeCubes(6, l_box_centre_num, f_base);
        // Nombre de cubes à faire tourner (8 ou 9)
        i_box_number = llGetListLength(l_box_rot);
}

// Rotation Totale
RotTotal(rotation rot) {
        llSetLinkPrimitiveParamsFast(LINK_ROOT, [PRIM_ROTATION, rot * r_rot]);
}

// Rotation partielle
RotPartiel(rotation rot) {
        integer i = i_box_number;
        while(i--) {
                llSetLinkPrimitiveParamsFast(llList2Integer(l_box_rot, i), [
                        PRIM_ROT_LOCAL, llList2Rot(l_box_rot_rot, i) * rot,
                        PRIM_POSITION, llList2Vector(l_box_rot_vecteur, i) * rot]);
        }
}

default
{
        state_entry() {
                // Initialisation des cubes
                get_link_nums();
                // Incrément de base selon vitesse
                f_Step = VITESSE * TIME_BASE;
```

```
}

// Détection du touché
touch_start(integer i_touch) {
	// Inhibition ?
	if(i_inhibe) return;
	// Numéro du cube touché
	integer i = llDetectedLinkNumber(0);
	// Face touchée
	integer i_face = llDetectedTouchFace(0);
	// Couleur de la face touchée (détection click interne sur face noire)
	if(GetColor(i, i_face) == <.0,.0,.0>) return;
	// Inhibition pour attendre fin rotation
	i_inhibe = TRUE;
	// Position touchée
	vector v = Global2Local(llDetectedTouchPos(0));
	// Trouve axe et sens de rotation
	TrouveAxeSens(v);
	// Angle de fin
	if(i_sens_normal) f_angle_fin = PI_BY_TWO;
	else f_angle_fin = -PI_BY_TWO;
	// Initialisation incrément
	f_Etat = .0;
	// Rotation totale
	if(is_central(i)) {
		// Rotation initiale
		r_rot = llGetRot();
		// Type de rotation
		i_rot_type = ROT_TOTAL;
	}
	// Rotation partielle
	else {
		// Position cube touché
		vector v_pos_box_touch = GetLinkPos(i);
		// Détermination des cubes concernés
		GetCubes(v_pos_box_touch);
		// Type de rotation
		i_rot_type = ROT_PARTIEL;
	}
	// Mise en route du timer
	llSetTimerEvent(TIME_BASE);
}

// Timer
timer()
{
	// Incrémentation
	f_Etat += f_Step;
	// Rotation
	rotation rot = llAxisAngle2Rot(v_axe, fCos(.0, f_angle_fin, f_Etat));
	// Application de la rotation
```

```
                if(f_Etat < 1.0) {
                        if(i_rot_type == ROT_TOTAL) RotTotal(rot);
                        else RotPartiel(rot);
                }
                // Fin de la rotation
                else {
                        // Arrêt timer
                        llSetTimerEvent(.0);
                        // Rotation finale
                        rotation r_rot_fin = llAxisAngle2Rot(v_axe, f_angle_fin);
                        if(i_rot_type == ROT_TOTAL) RotTotal(r_rot_fin);
                        else RotPartiel(r_rot_fin);
                        // Levée de l'inhibition
                        i_inhibe = FALSE;
                }
        }
}
```

Le mouvement a été défini avec une interpolation sinusoïdale pour plus de réalisme. La partie purement mouvement dans ce script est relativement simple et correspond à ce qui a déjà été vu dans les projets précédents.

La rotation partielle se fait dans une fonction qui boucle sur tous les cubes concernés :

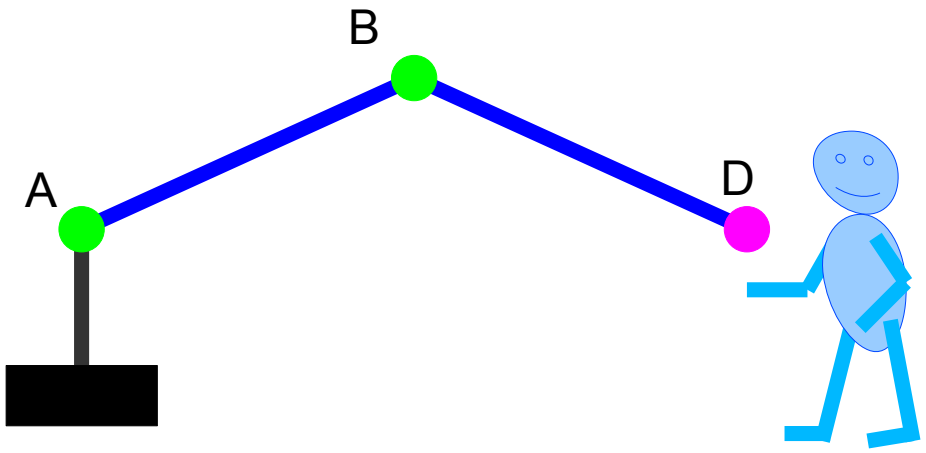
```
// Rotation partielle
RotPartiel(rotation rot) {
        integer i = i_box_number;
        while(i--) {
                llSetLinkPrimitiveParamsFast(llList2Integer(l_box_rot, i), [
                        PRIM_ROT_LOCAL, llList2Rot(l_box_rot_rot, i) * rot,
                        PRIM_POSITION, llList2Vector(l_box_rot_vecteur, i) * rot]);
        }
}
```

C'est un déplacement circulaire qui combine translation et rotation. Je vous laisse analyser ce code pour plus de détails.

11. Bras articulé

Ce projet va être un peu plus pointu parce que nous allons autoriser des déplacements dans toutes les directions de l'espace.

11.1 Enoncé du projet

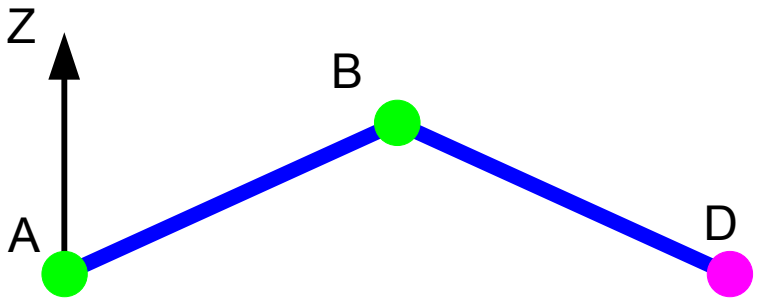


Un bras articulé est composé de deux bielles (bleues) et deux rotules (vertes) sur un support (noir). Un sensor est mis en place dans la rotule A. Dès qu'au moins un avatar est détecté à proximité, l'oeil (boule magenta) se déplace devant lui, les deux bielles s'adaptant.Si plusieurs avatars sont présents c'est le plus proche qui pris en compte. Si l'avatar se déplace latéralement ou verticalement l'oeil doit le suivre et les bielles continuer à s'adapter.

11.2 Paramètres

La première chose à faire est de définir la position de départ. Nous allons positionner l'ensemble dans un état quelconque. Il nous faut connaître :

- la longueur des bielles (on suppose qu'elles sont de même longueur),
- l'axe de la longueur des bielles,
- la position de la rotule A (on fixe cette primitive comme racine, donc elle devient origine des axes et le point est <0,0,0>,
- l'axe de référence qui sera l'axe vertical du support, pour simplifier il va coincider avec l'axe Z de la primitive racine positionnée en A,
- l'offset de positonnement de l'oeil en D par rapport à l'avatar en valeurs horizontale et verticale,
- le délai de scan du sensor en état d'attente et de présence d'avatars,
- les numéros de liaison des primitives concernées (rotules, bielles et oeil).



11.3 Calculs préliminaires

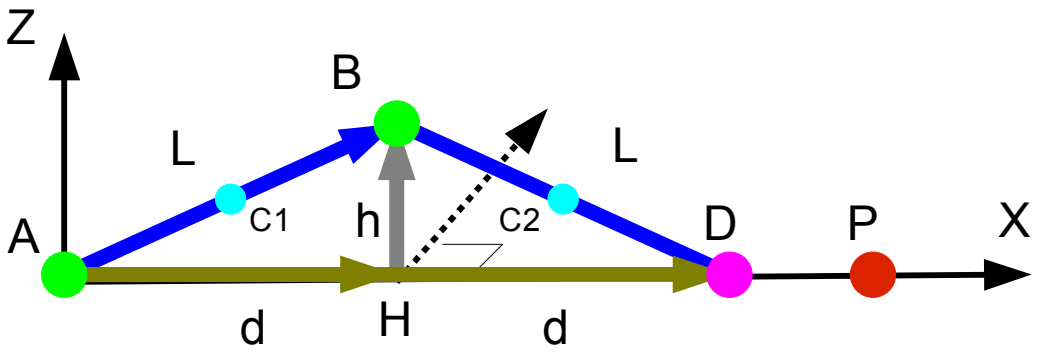
11.3.1 Lecture rotations bielles

Les bielles ont une certaine orientation dans l'espace selon leur constitution. Il nous faut connaître cette orientation. Nous allons simplement lire leur rotation locale.

11.3.2 Adaptation de l'axe des bielles

L'axe des bielles doit être adapté à la rotation de celles-ci pour avoir des axes corrects selon le référentiel local. On sait qu'il suffit de multiplier l'axe par la rotation de la bielle.

11.4 Traitement vectoriel du mouvement



Nous avons la position de l'avatar (point P) en coordonnées globales. Il faut appliquer les offsets et transformer en coordonnée locales pour définir le point D de position de l'oeil en tenant compte de la longueur maximale de déploiement qui correspond à la longueur totale des deux bielles.

Ca nous permet de déterminer l'axe X et le vecteur unitaire ux sur cet axe.

Il nous faut connaître l'axe H qui donne la projection du vecteur AB sur l'axe X. Cette fois le produit scalaire ne nous est d'aucun secours. Par contre le produit vectoriel va nous aider. Si on fait le produit vectoriel entre les axes Z et X on obtient un vecteur perpendiculaire à ces deux axes qui est représenté en pointillés sur la figure. On pourrait utiliser directement cet axe mais ça donnera un effet plus élégant avec un axe vertical. Il suffit de faire un nouveau produit vectoriel cette fois entre cet axe en pointillés et l'axe X, nous obtenons ainsi l'axe H voulu.

Nous connaissons la distance d qui est la moitié de la norme du vecteur AD. Nous connaissons aussi la distance L qui est la longueur de la bielle. Avec Pythagore il devient facile de calculer la distance h :

$$h = sqr\ (L^2 - d^2)$$

On peut alors déterminer la position du point B :

$$\text{Vecteur AB} = \text{Vecteur AH} + \text{vecteur HB} = d * ux + h * uh$$

La moitié de ce vecteur nous donne la position du centre de la première bielle :

$$\text{Vecteur AC1} = \text{Vecteur AB} / 2$$

D'où on en déduit le centre de la deuxième bielle :

$$\text{Vecteur AC2} = \text{Vecteur AC1} + \text{Vecteur BD} / 2$$

Il ne reste plus qu'à définir la rotation des bielles. On peut facilement calculer la rotation entre l'axe Z qui nous sert de référence fixe et l'axe des bielles. D'où on en déduit la rotation de ces dernières.

11.5 Codage (1)

Toute la partie détection et détermination de l'avatar le plus proche sera juste commentée dans le code sans donner lieu à plus d'explications parce que sortant du cadre de ce manuel.

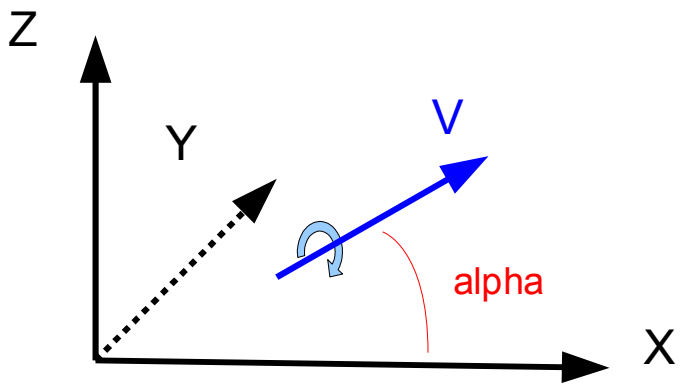
```
// ------------------------
//    Constantes
// ------------------------
// Durée de base en s en attente
float TIME_ATTENTE = 2.0;
// Durée de base en s en action
float TIME_ACTION = .1;
// Longueur des bielles (entre axes)
float LONGUEUR_BIELLE = 2.0;
// Numéro de liaison de la primitive de la bielle 1
integer PRIM_BIELLE_1 = 2;
// Numéro de liaison de la primitive de la bielle 2
integer PRIM_BIELLE_2 = 3;
// Numéro de la boule 1
integer PRIM_BOULE_1 = 4;
// Numéro de la boule 2
integer PRIM_BOULE_2 = 5;
// Axe de la longueur des bielles
vector AXE_LONGUEUR_BIELLE = <.0, .0, 1.0>;
// Vecteur unitaire axe Z
vector UZ = <.0, .0, 1.0>;
// Offset horizontal
float OFFSET_HOR = 1.0;
// Offset vertical
float OFFSET_VER = .5;
// Distance de détection
float RANGE = 8.0;

// ------------------------
//    Variables globales
```

```
// ------------------------
// Rotation de la bielle 1
rotation r_bielle_1;
// Rotation de la bielle 2
rotation r_bielle_2;
// Etat de fonctionnement
integer iOn;

// ------------------------
//    Fonctions - Librairie
// ------------------------
// Récupération rotation locale prim enfant
rotation GetLinkRot(integer link_num) {
        return  llList2Rot(llGetLinkPrimitiveParams(link_num, [PRIM_ROT_LOCAL]), 0);
}
// Transformation Position Globale -> Locale
vector GetLocalPos(vector v_pos_global) {
        return (v_pos_global - llGetRootPosition()) / llGetRootRotation();
}

action(integer total_number)
{
        integer n;
        // Si plusieurs avatars trouver le plus proche
        if(total_number > 1)
        {
                // Position du sensor (root)
                vector v_sensor = llGetPos();
                // Position premier avatar
                vector v_ava = llDetectedPos(0);
                // Distance
                float f_dt = llVecMag(v_sensor - v_ava);
                // Balayage
                integer i = 1;
                for (;i < total_number; i++)
                {
                        // Position avatar
                        v_ava = llDetectedPos(i);
                        // Distance
                        float f_d = llVecMag(v_sensor - v_ava);
                        // Test
                        if(f_d < f_dt) {
                                n = i;
                                f_dt = f_d;
                        }
                }
        }
        // Cible en coordonnées locales(Point P) = vecteur AP = axe X
        vector v_P = GetLocalPos(llDetectedPos(n));
        // Vecteur unitaire axe X
        vector v_ux = llVecNorm(v_P);
```

```
        // Position point D
        vector v_D;
        if(llVecMag(v_P) > OFFSET_HOR + .6)
                v_D = v_P - v_ux * OFFSET_HOR + UZ * OFFSET_VER;
        else
                v_D = v_ux * .6 + UZ * OFFSET_VER;
        // Correction Vecteur unitaire axe X
        v_ux = llVecNorm(v_D);
        // Test limite des bielles
        if (llVecMag(v_D) > LONGUEUR_BIELLE * 2.0 - .001)
                v_D = v_ux * LONGUEUR_BIELLE * 2.0 - .001;
        if (llVecMag(v_D) < .6)
                v_D = .6 * v_ux;
        // Correction Vecteur unitaire axe X
        v_ux = llVecNorm(v_D);
        // Axe H
        vector v_h = llVecNorm(v_ux % UZ % v_ux);
        // Distance d
        float f_dist_d = llVecMag(v_D) / 2.0;
        // Distance h
        float f_dist_h = llSqrt(LONGUEUR_BIELLE * LONGUEUR_BIELLE - f_dist_d * f_dist_d);
        // Position point B
        vector v_B = f_dist_d * v_ux + f_dist_h * v_h;
        // Position centre bielle 1
        vector v_C1 = v_B / 2.0;
        // Position centre bielle 2
        vector v_C2 = v_B + (v_D - v_B) / 2.0;
        // Rotation bielle 1
        rotation r_rot_bielle_1 = r_bielle_1 * llRotBetween(UZ, v_B);
        // Rotation bielle 2
        rotation r_rot_bielle_2 = r_bielle_2 * llRotBetween(UZ, v_D - v_B);

        // Ajustement de la bielle 1
        llSetLinkPrimitiveParamsFast(PRIM_BIELLE_1, [
                PRIM_ROT_LOCAL, r_rot_bielle_1,
                PRIM_POSITION, v_C1]);
        // Ajustement de la bielle 2
        llSetLinkPrimitiveParamsFast(PRIM_BIELLE_2, [
                PRIM_ROT_LOCAL, r_rot_bielle_2,
                PRIM_POSITION, v_C2]);
        // Ajustement boule 1
        llSetLinkPrimitiveParamsFast(PRIM_BOULE_1, [PRIM_POSITION, v_B]);
        // Ajustement boule 2
        llSetLinkPrimitiveParamsFast(PRIM_BOULE_2, [PRIM_POSITION, v_D]);
}

default
{
        state_entry()
        {
                // Rotation de la bielle 1
```

```
            r_bielle_1 = GetLinkRot(PRIM_BIELLE_1);
            // Rotation de la bielle 2
            r_bielle_2 = GetLinkRot(PRIM_BIELLE_2);
            // Axe bielle 1
            vector v_axe_1 = AXE_LONGUEUR_BIELLE * r_bielle_1;
            // Axe bielle 2
            vector v_axe_2 = AXE_LONGUEUR_BIELLE * r_bielle_2;
            // Rotation bielle 1 ramenée à l'axe Z
            r_bielle_1 /= llRotBetween(UZ, v_axe_1);
            // Rotation bielle 2 ramenée à l'axe Z
            r_bielle_2 /= llRotBetween(UZ, v_axe_2);
            // Mise en marche de la détection en mode attente
            llSensorRepeat("", NULL_KEY, AGENT, RANGE, PI, TIME_ATTENTE);
        }
        sensor(integer total_number)
        {
            if(iOn) {
                if(!total_number) {
                    iOn = FALSE;
                    llSensorRepeat("", NULL_KEY, AGENT, RANGE, PI, TIME_ATTENTE);
                }
                else action(total_number);
            }
            else if(total_number) {
                iOn = TRUE;
                action(total_number);
                llSensorRepeat("", NULL_KEY, AGENT, RANGE, PI, TIME_ACTION);
            }
        }
}
```

Ce code fonctionne parfaitement en choisissant des bielles cylindriques et des rotules sphériques. Parce qu'il subsiste tout de même un petit défaut concernant la rotation des bielles. La fonction **llRotBetween** a été utilisée. Cette fonction permet de connaître la valeur de la rotation entre deux axes. Autrement dit elle fonctionne au niveau d'un plan. Observez cette figure :



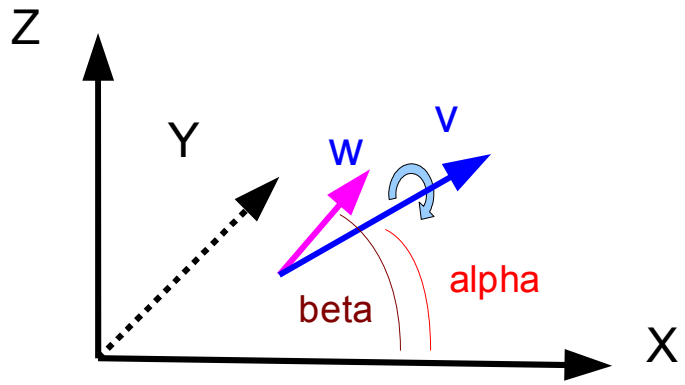
Le vecteur V se situe dans le référenciel représenté par les axe x, y et z. Si nous considérons l'axe X et le vecteur V, avec la fonction **llRotBetween** nous pouvons connaître la rotation correspondant à l'angle alpha représenté en

rouge. Nous n'avons par contre aucune idée de la rotation représentée par la flèche bleue parce qu'elle se situe sur un autre plan.

Il en est de même pour les bielles de notre projet. On leur donne la bonne rotation par rapport à l'axe vertical de référence. Par contre elles conservent leur rotation sur elle-même et donc elles semblent tourner sur elles-même morsque le bras prend une autre direction latérale. C'est invisible avec des bielles cylindriques et des rotules sphériques mais ça devient apparent avec d'autres choix. En plus si nous choisissons d'appliquer une texture d'oeil sur la boule située au point D il serait souhaitable que cet oeil reste horizontal et ne s'amuse pas à tourner selon la direction du bras.

11.6 Rotation d'un vecteur sur lui-même

Le problème est celui-ci : connaissant un vecteur comment connaître sa rotation ? Mais le problème est mal posé. Un vecteur est parfaitement défini dans un plan et la fonction **llRotBetween** est suffisante pour régler la question. Si nous envisageons la rotation du vecteur sur lui-même c'est une autre histoire et un seul vecteur ne suffit plus, nous avons besoin d'une autre référence :



En considérant un deuxième vecteur, le vecteur W, perpendiculaire à V, nous avons cette référence par rapport à la rotation représentée par la flèche bleue, l'angle beta. Il suffit d'appliquer la fonction **llRotBetween** entre ce vecteur et un axe, par exemple encore X. Il faut donc gérer ce vecteur W. Il est facile à définir avec un produit vectoriel. Il suffit alors de mémoriser la valeur initiale et ensuite de faire une adaptation. Mais comment ensuite appliquer la rotation ? On constate que le problème se complique un peu.

Si nous regardons les fonctions que nous offre le LSL nous trouvons les fonctions **llRotBetween** et **llAxes2Rot**. Nous avons utilisé la première jusqu'ici. La seconde peut-elle résoudre notre problème ? Cette fonction réclame 3 vecteurs (avant, gauche et haut, qui sont respectivement X, Y et Z). Ces 3 vecteurs doivent être unitaires orthogonaux deux à deux. Le WIKI nous signale qu'on peut techniquement s'en sortir avec deux vecteurs :

llAxes2Rot (avant, gauche, avant % gauche);

On se rend compte que le vecteur “haut” est tout simplement défini à partir d'un produit vectoriel des deux premiers. Il semblerait que ça corresponde exactement à notre situation. Il suffit de bien définir l'orientation des axes parce que notre bielle n'est pas forcément idéalement orientée. On peut toujours envisager une bielle idéalement orientée et utiliser un offset de rotation pour retomber sur nos pattes.

11.7 Codage (2)

A partir de nos réflexions précédentes nous allons traiter le problème pour les bielles ainsi :

- au départ on récupère la rotation de chaque bielle,

- on définit un offset de rotation par rapport à une bielle idéalement orientée (avant, gauche, haut),
- à chaque changement on calcule la rotation de la bielle idéale avec le vecteur directif et le vecteur perpendiculaire défini à partir d'un produit vectoriel avec l'axe Z,
- on applique l'offset de rotation pour retomber sur nos pattes.

```
// ------------------------
//    Constantes
// ------------------------
// Durée de base en s en attente
float TIME_ATTENTE = 2.0;
// Durée de base en s en action
float TIME_ACTION = .1;
// Longueur des bielles (entre axes)
float LONGUEUR_BIELLE = 2.0;
// Numéro de liaison de la primitive de la bielle 1
integer PRIM_BIELLE_1 = 2;
// Numéro de liaison de la primitive de la bielle 2
integer PRIM_BIELLE_2 = 3;
// Numéro de la boule 1
integer PRIM_BOULE_1 = 4;
// Numéro de la boule 2
integer PRIM_BOULE_2 = 5;
// Axe de la longueur des bielles
vector AXE_LONGUEUR_BIELLE = <.0, .0, 1.0>;
// Vecteur unitaire axe Z
vector UZ = <.0, .0, 1.0>;
// Offset horizontal
float OFFSET_HOR = 1.0;
// Offset vertical
float OFFSET_VER = .5;
// Distance de détection
float RANGE = 8.0;

// ------------------------
//    Variables globales
// ------------------------
// Offset de rotation bielle 1
rotation r_offset_bielle_1;
// Offset de rotation bielle 2
rotation r_offset_bielle_2;
// Etat de fonctionnement
integer iOn;

// ------------------------
//    Fonctions - Librairie
// ------------------------
// Récupération rotation locale prim enfant
rotation GetLinkRot(integer link_num) {
        return  llList2Rot(llGetLinkPrimitiveParams(link_num, [PRIM_ROT_LOCAL]), 0);
}
```

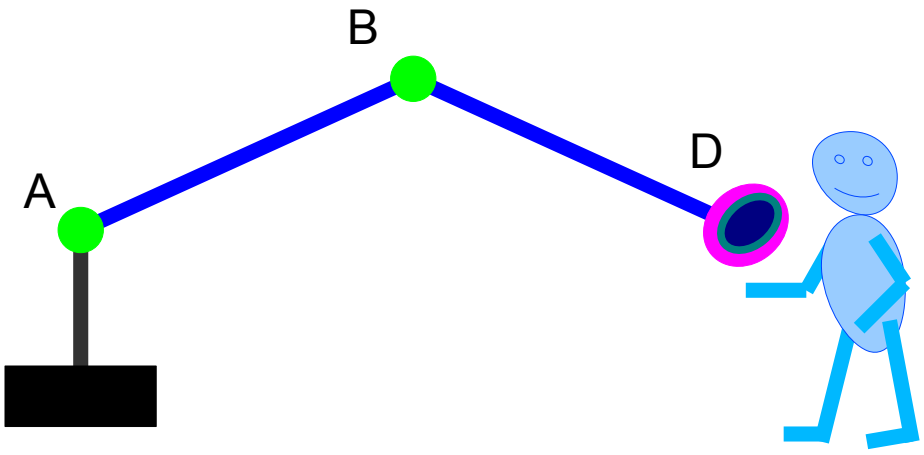
```
// Transformation Position Globale -> Locale
vector GetLocalPos(vector v_pos_global) {
	return (v_pos_global - llGetRootPosition()) / llGetRootRotation();
}

// ------------------------
//    Fonctions - Script
// ------------------------
// Calcul offset rotation locale
rotation get_offset(integer i_prim, vector v_axe, vector v_axe_ref, float f_longueur)
{
	// Rotation
	rotation r_rot = GetLinkRot(i_prim);
	// Axe
	vector v_axe_rot = v_axe * r_rot;
	// Vecteur unitaire
	vector v_u = llVecNorm(v_axe_rot * f_longueur);
	// Vecteur perpendiculaire
	vector v_per = llVecNorm(v_axe_ref % v_u);
	// Rotation idéale
	rotation r_ideale = llAxes2Rot(v_u, v_per, v_u % v_per);
	// Offset de rotation
	return r_rot / r_ideale;
}

action(integer total_number)
{
	integer n;
	// Si plusieurs avatars trouver le plus proche
	if(total_number > 1)
	{
		// Position du sensor (root)
		vector v_sensor = llGetPos();
		// Position premier avatar
		vector v_ava = llDetectedPos(0);
		// Distance
		float f_dt = llVecMag(v_sensor - v_ava);
		// Balayage
		integer i = 1;
		for (;i < total_number; i++)
		{
			// Position avatar
			v_ava = llDetectedPos(i);
			// Distance
			float f_d = llVecMag(v_sensor - v_ava);
			// Test
			if(f_d < f_dt) {
				n = i;
				f_dt = f_d;
			}
		}
```

```
    }
    // Cible en coordonnées locales(Point P) = vecteur AP = axe X
    vector v_P = GetLocalPos(llDetectedPos(n));
    // Vecteur unitaire axe X
    vector v_ux = llVecNorm(v_P);
    // Position point D
    vector v_D;
    if(llVecMag(v_P) > OFFSET_HOR + .6)
        v_D = v_P - v_ux * OFFSET_HOR + UZ * OFFSET_VER;
    else
        v_D = v_ux * .6 + UZ * OFFSET_VER;
    // Correction Vecteur unitaire axe X
    v_ux = llVecNorm(v_D);
    // Test limite des bielles
    if (llVecMag(v_D) > LONGUEUR_BIELLE * 2.0 - .001)
        v_D = v_ux * (LONGUEUR_BIELLE * 2.0 - .001);
    if (llVecMag(v_D) < .6)
        v_D = .6 * v_ux;
    // Correction Vecteur unitaire axe X
    v_ux = llVecNorm(v_D);
    // Axe H
    vector v_h = llVecNorm(v_ux % UZ % v_ux);
    // Distance d
    float f_dist_d = llVecMag(v_D) / 2.0;
    // Distance h
    float f_dist_h = llSqrt(LONGUEUR_BIELLE * LONGUEUR_BIELLE - f_dist_d * f_dist_d);
    // Position point B
    vector v_B = f_dist_d * v_ux + f_dist_h * v_h;
    // Position centre bielle 1
    vector v_C1 = v_B / 2.0;
    // Vecteur BD
    vector v_BD = v_D - v_B;
    // Position centre bielle 2
    vector v_C2 = v_B + v_BD / 2.0;
    // Vecteur unitaire sur AB
    vector v_ub = llVecNorm(v_B);
    // vecteur unitaire sur BD
    vector v_ud = llVecNorm(v_BD);
    // Vecteur perpendiculaire
    vector v_per = llVecNorm(UZ % v_ud);
    // Rotation bielle 1
    rotation r_rot_bielle_1 = r_offset_bielle_1 * llAxes2Rot(v_ub, v_per, v_ub % v_per);
    // Rotation bielle 2
    rotation r_rot_bielle_2 = r_offset_bielle_2 * llAxes2Rot(v_ud, v_per, v_ud % v_per);
    // Ajustement de la bielle 1
    llSetLinkPrimitiveParamsFast(PRIM_BIELLE_1, [
        PRIM_ROT_LOCAL, r_rot_bielle_1,
        PRIM_POSITION, v_C1]);
    // Ajustement de la bielle 2
    llSetLinkPrimitiveParamsFast(PRIM_BIELLE_2, [
        PRIM_ROT_LOCAL, r_rot_bielle_2,
```

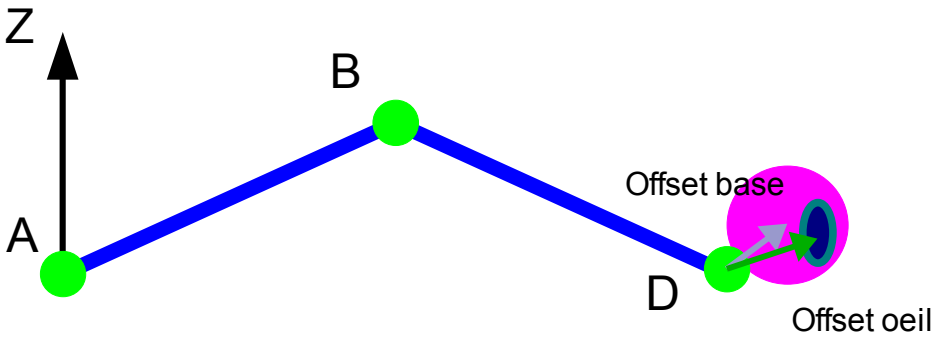
```
                PRIM_POSITION, v_C2]);
        // Ajustement boule 1
        llSetLinkPrimitiveParamsFast(PRIM_BOULE_1, [PRIM_POSITION, v_B]);
        // Ajustement boule 2
        llSetLinkPrimitiveParamsFast(PRIM_BOULE_2, [PRIM_POSITION, v_D]);
}

default
{
        state_entry()
        {
                // Offset de rotation bielle 1
                r_offset_bielle_1 = get_offset(PRIM_BIELLE_1, AXE_LONGUEUR_BIELLE,
                                                        UZ, LONGUEUR_BIELLE);
                // Offset de rotation bielle 2
                r_offset_bielle_2 = get_offset(PRIM_BIELLE_2, AXE_LONGUEUR_BIELLE,
                                                        UZ, LONGUEUR_BIELLE);
                // Mise en marche de la détection en mode attente
                llSensorRepeat("", NULL_KEY, AGENT, RANGE, PI, TIME_ATTENTE);
        }
        sensor(integer total_number)
        {
                if(iOn) {
                        if(!total_number) {
                                iOn = FALSE;
                                llSensorRepeat("", NULL_KEY, AGENT, RANGE, PI, TIME_ATTENTE);
                        }
                        else action(total_number);
                }
                else if(total_number) {
                        iOn = TRUE;
                        action(total_number);
                        llSensorRepeat("", NULL_KEY, AGENT, RANGE, PI, TIME_ACTION);
                }
        }
}
```

11.8 Codage (3)

Maintenant que les bielles sont bien gérées nous allons nous intéresser à l'oeil. C'est après tout l'élément le plus important du système. Il serait donc souhaitable d'avoir un oeil sympathique qui conserve sa direction dans le prolongement de la deuxième bielle. Pour bien faire on va constituer l'oeil avec deux primitives : la base et l'oeil lui-même :



Pour simplifier on va s'arranger pour que les deux parties de l'oeil aient la même rotation. D'autre part on sait que l'oeil est dans le prolongement de la deuxième bielle et subit la même rotation, donc on va s'appuyer sur la rotation de celle-ci pour obtenir la rotation de l'oeil. Pour la position des deux parties de l'oeil on va calculer l'offset par rapport au point D au départ et ensuite l'appliquer à chaque mouvement en orientant cet offset selon la rotation de l'oeil :



A partir de ces bases voici un code possible :

```
// ------------------------
//    Constantes
// ------------------------
// Durée de base en s en attente
float TIME_ATTENTE = 2.0;
// Durée de base en s en action
float TIME_ACTION = .1;
// Longueur des bielles (entre axes)
float LONGUEUR_BIELLE = 2.0;
// Numéro de liaison de la primitive de la bielle 1
integer PRIM_BIELLE_1 = 2;
// Numéro de liaison de la primitive de la bielle 2
integer PRIM_BIELLE_2 = 3;
// Numéro de la boule 1
integer PRIM_BOULE_1 = 4;
// Numéro de la base de l'oeil
integer PRIM_OEIL_BASE = 5;
```

```
// Numéro de l'oeil
integer PRIM_OEIL = 6;
// Axe de la longueur des bielles
vector AXE_LONGUEUR_BIELLE = <.0, .0, 1.0>;
// Vecteur unitaire axe Z
vector UZ = <.0, .0, 1.0>;
// Offset horizontal
float OFFSET_HOR = 1.0;
// Offset vertical
float OFFSET_VER = .5;
// Distance de détection
float RANGE = 8.0;

// -------------------------
//    Variables globales
// -------------------------
// Offset de rotation bielle 1
rotation r_offset_bielle_1;
// Offset de rotation bielle 2
rotation r_offset_bielle_2;
// Offset de rotation de l'oeil
rotation r_offset_oeil;
// Offset de position base de l'oeil
vector v_offset_base_oeil;
// Offset de position oeil
vector v_offset_oeil;
// Etat de fonctionnement
integer iOn;

// -------------------------
//    Fonctions - Librairie
// -------------------------
// Récupération position locale prim enfant
vector GetLinkPos(integer link_num) {
        vector v = llList2Vector(llGetLinkPrimitiveParams(link_num, [PRIM_POSITION]), 0);
        return (v - llGetRootPosition()) / llGetRootRotation();
}
// Récupération rotation locale prim enfant
rotation GetLinkRot(integer link_num) {
        return  llList2Rot(llGetLinkPrimitiveParams(link_num, [PRIM_ROT_LOCAL]), 0);
}
// Transformation Position Globale -> Locale
vector GetLocalPos(vector v_pos_global) {
        return (v_pos_global - llGetRootPosition()) / llGetRootRotation();
}

// -------------------------
//    Fonctions - Script
// -------------------------
// Calcul offset rotation locale
rotation get_offset(integer i_prim, vector v_axe, vector v_axe_ref, float f_longueur)
```

```
{
	// Rotation
	rotation r_rot = GetLinkRot(i_prim);
	// Axe
	vector v_axe_rot = v_axe * r_rot;
	// Vecteur unitaire
	vector v_u = llVecNorm(v_axe_rot * f_longueur);
	// Vecteur perpendiculaire
	vector v_per = llVecNorm(v_axe_ref % v_u);
	// Rotation idéale
	rotation r_ideale = llAxes2Rot(v_u, v_per, v_u % v_per);
	// Offset de rotation
	return r_rot / r_ideale;
}

action(integer total_number)
{
	integer n;
	// Si plusieurs avatars trouver le plus proche
	if(total_number > 1)
	{
		// Position du sensor (root)
		vector v_sensor = llGetPos();
		// Position premier avatar
		vector v_ava = llDetectedPos(0);
		// Distance
		float f_dt = llVecMag(v_sensor - v_ava);
		// Balayage
		integer i = 1;
		for (;i < total_number; i++)
		{
			// Position avatar
			v_ava = llDetectedPos(i);
			// Distance
			float f_d = llVecMag(v_sensor - v_ava);
			// Test
			if(f_d < f_dt) {
				n = i;
				f_dt = f_d;
			}
		}
	}
	// Cible en coordonnées locales(Point P) = vecteur AP = axe X
	vector v_P = GetLocalPos(llDetectedPos(n));
	// Vecteur unitaire axe X
	vector v_ux = llVecNorm(v_P);
	// Position point D
	vector v_D;
	if(llVecMag(v_P) > OFFSET_HOR + .6)
		v_D = v_P - v_ux * OFFSET_HOR + UZ * OFFSET_VER;
	else
```

```
        v_D = v_ux * .6 + UZ * OFFSET_VER;
    // Correction Vecteur unitaire axe X
    v_ux = llVecNorm(v_D);
    // Test limite des bielles
    if (llVecMag(v_D) > LONGUEUR_BIELLE * 2.0 - .001)
        v_D = v_ux * (LONGUEUR_BIELLE * 2.0 - .001);
    if (llVecMag(v_D) < .6)
        v_D = .6 * v_ux;
    // Correction Vecteur unitaire axe X
    v_ux = llVecNorm(v_D);
    // Axe H
    vector v_h = llVecNorm(v_ux % UZ % v_ux);
    // Distance d
    float f_dist_d = llVecMag(v_D) / 2.0;
    // Distance h
    float f_dist_h = llSqrt(LONGUEUR_BIELLE * LONGUEUR_BIELLE - f_dist_d * f_dist_d);
    // Position point B
    vector v_B = f_dist_d * v_ux + f_dist_h * v_h;
    // Position centre bielle 1
    vector v_C1 = v_B / 2.0;
    // Vecteur BD
    vector v_BD = v_D - v_B;
    // Position centre bielle 2
    vector v_C2 = v_B + v_BD / 2.0;
    // Vecteur unitaire sur AB
    vector v_ub = llVecNorm(v_B);
    // vecteur unitaire sur BD
    vector v_ud = llVecNorm(v_BD);
    // Vecteur perpendiculaire
    vector v_per = llVecNorm(UZ % v_ud);
    // Rotation bielle 1
    rotation r_rot_bielle_1 = r_offset_bielle_1 * llAxes2Rot(v_ub, v_per, v_ub % v_per);
    // Rotation bielle 2
    rotation r_rot_bielle_2 = r_offset_bielle_2 * llAxes2Rot(v_ud, v_per, v_ud % v_per);
    // Rotation de l'oeil
    rotation r_oeil = r_offset_oeil * r_rot_bielle_2;
    // Ajustement de la bielle 1
    llSetLinkPrimitiveParamsFast(PRIM_BIELLE_1, [
        PRIM_ROT_LOCAL, r_rot_bielle_1,
        PRIM_POSITION, v_C1]);
    // Ajustement de la bielle 2
    llSetLinkPrimitiveParamsFast(PRIM_BIELLE_2, [
        PRIM_ROT_LOCAL, r_rot_bielle_2,
        PRIM_POSITION, v_C2]);
    // Ajustement boule 1
    llSetLinkPrimitiveParamsFast(PRIM_BOULE_1, [
        PRIM_POSITION, v_B]);
    // Ajustement base de l'oeil
    llSetLinkPrimitiveParamsFast(PRIM_OEIL_BASE, [
        PRIM_ROT_LOCAL, r_oeil,
        PRIM_POSITION, v_D + v_offset_base_oeil * r_oeil]);
```

```
        // Ajustement de l'oeil
        llSetLinkPrimitiveParamsFast(PRIM_OEIL, [
                PRIM_ROT_LOCAL, r_oeil,
                PRIM_POSITION, v_D + v_offset_oeil * r_oeil]);
}

default
{
        state_entry()
        {
                // Offset de rotation bielle 1
                r_offset_bielle_1 = get_offset(PRIM_BIELLE_1, AXE_LONGUEUR_BIELLE,
                                                        UZ, LONGUEUR_BIELLE);
                // Offset de rotation bielle 2
                r_offset_bielle_2 = get_offset(PRIM_BIELLE_2, AXE_LONGUEUR_BIELLE,
                                                        UZ, LONGUEUR_BIELLE);
                // Rotation de l'oeil
                rotation r_oeil = GetLinkRot(PRIM_OEIL);
                // Rotation de la bielle 2
                rotation r_bielle_2 = GetLinkRot(PRIM_BIELLE_2);
                // Offset de rotation oeil
                r_offset_oeil = r_oeil / r_bielle_2;
                // Position Point D
                vector v_D = GetLinkPos(PRIM_BIELLE_2) +
                        llVecNorm(AXE_LONGUEUR_BIELLE * r_bielle_2) * LONGUEUR_BIELLE / 2.0;
                // Offset de position base de l'oeil
                v_offset_base_oeil = (GetLinkPos(PRIM_OEIL_BASE) - v_D) / r_oeil;
                // Offset de position oeil
                v_offset_oeil = (GetLinkPos(PRIM_OEIL) - v_D) / r_oeil;
                // Mise en marche de la détection en mode attente
                llSensorRepeat("", NULL_KEY, AGENT, RANGE, PI, TIME_ATTENTE);
        }
        sensor(integer total_number)
        {
                if(iOn) {
                        if(!total_number) {
                                iOn = FALSE;
                                llSensorRepeat("", NULL_KEY, AGENT, RANGE, PI, TIME_ATTENTE);
                        }
                        else action(total_number);
                }
                else if(total_number) {
                        iOn = TRUE;
                        action(total_number);
                        llSensorRepeat("", NULL_KEY, AGENT, RANGE, PI, TIME_ACTION);
                }
        }
}
```