Didacticiel avancé: vos premières animations avec POV-Ray!

2ème partie

Merci à Diamond Editions pour son aimable autorisation pour la mise en ligne de cet article, initialement publié dans Linux Magazine N°66

Saraja Olivier – olivier.saraja@linuxgraphic.org

Nous avons vu dans le précédent article comment réaliser de simples animations avec POV-ray, de la création des fichiers d'initialisation aux scènes elles-mêmes. Dans cet article, nous allons reprendre brièvement, sans les ré-expliquer, des notions vues la fois passée, et nous allons les mixer avec une autre méthode d'animation, plus souple et moins mathématique: l'animation le long d'une spline.

1. Les splines

Comme nous allons le voir ultérieurement, les splines se révèlent le compagnon indispensable des animateurs en herbe. Avec elles, il est possible d'imposer à un objet des trajectoires complexes qu'il serait infernal d'obtenir à l'aide du simple usage de la variable `clock` et de formules mathématiques pas toujours faciles à apprivoiser. Les splines ont une présence plus ou moins discrètes au sein de plusieurs types de primitives de POV-ray: SOR, lath, prism... qui emploient des versions 2D des splines pour définir leur profil général. Les splines que nous allons étudier maintenant se composent de façon légèrement différente, car elles peuvent se concevoir directement en 3D, mais présentent toutefois de nombreux points communs. Nous allons tenter ici de recenser tout ce qu'il est bon de savoir au sujet des splines pour un usage immédiat et pragmatique dans une scène animée.

2. Définition de la spline

Le principe est assez simple, puisqu'il vous suffit de prescrire une série de points de coordonnées quelconques. POV-ray va alors de charger de tracer la courbe qui passera par tous ces points. L'ordre de la courbe (ou plutôt sa forme, pour rester simple et le moins mathématique possible) dépendra du type de spline que vous souhaitez (voir Types de Spline, ci-après): segments de droite, courbes cubiques ou quadratiques, courbes de Bezier... Le bloc de description d'une spline prendra donc la forme suivante:

```
#declare Ma_Spline =
spline {
type_de_spline
Val_1, <x_1, y_1, z_1>
Val_2, <x_2, y_2, z_2>
...
Val_n, <x_n, y_n, z_n>
}
```

S'il est facile de comprendre que `<x_n, y_n, z_n>` sont les différents points qui permettent de définir la trajectoire de la spline, `Val_n` permet d'associer une valeur numérique unique au point considéré. Si deux points ont la même valeur numérique, le second point remplacera le premier. Toutefois, pour tracer un « huit » (voir figure 01), rien ne vous empêche d'avoir un point unique pour deux valeurs numériques différentes. Quant à ces valeurs numériques, elles vont vous permettre de spécifier la vitesse instantanée de votre objet, en déterminant la position de l'objet sur la pline à un moment précis de l'animation.

Pour illustrer, admettons le code suivant pour notre scène, `huit_spline.pov` (nous nous contenterons de l'admettre car nous verrons la mise en oeuvre des animations le long d'une spline plus en détail un peu plus loin):

```
light_source {
<4, 5, -5>, rgb <1, 1, 1>
}

sky_sphere {
pigment {
color rgb <0.752941, 0.752941, 1>
}
}

#declare Ma_Spline =
spline {
linear_spline
...,<0, 0, 0>
..., <2, 2, 0>
..., <0, 4, 0>
..., <-2, 6, 0>
..., <0, 8, 0>
..., <2, 6, 0>
..., <0, 4, 0>
..., <-2, 2, 0>
..., <0, 0, 0>
}

#declare Nr = 0;
#declare EndNr = 1;
#while (Nr< EndNr)
sphere{
<0,0,0>,0.07
texture{ pigment{color rgb <1,1,0>}
}
translate Ma_Spline(Nr)
}
#declare Nr = Nr + 0.001;
#end
sphere {
<0, 0, 0>, 0.5
texture {
pigment {
color rgb <1, 0, 0>
}
translate Ma_Spline(clock)
}

camera {
location <0, 5, -12>
look_at <0, 4, 0>
}
```

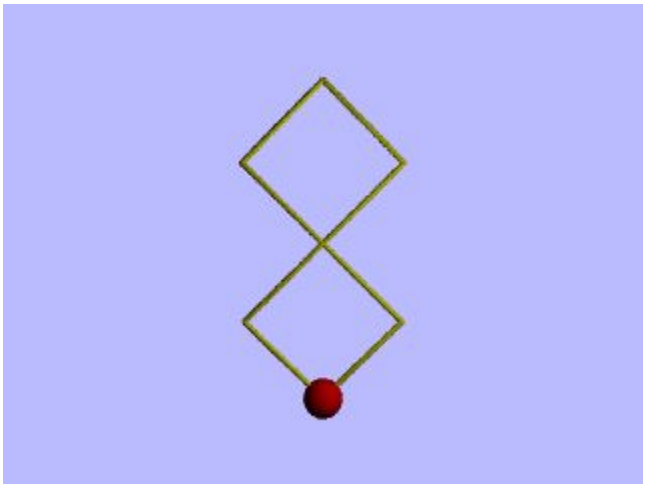

Figure 01: l'objectif de cette étude de spline

Ne sachant encore pas trop comment interpréter les valeurs numériques que doivent prendre les points de notre spline, supposons simplement que la variable sera égale à `0` lorsque `clock=0` et égale à `1` lorsque `clock=1`. Comme il y a neuf points à notre spline, en commençant de zéro, nous allons attribuer à chaque point les fractions suivantes: `0/8` (soit notre premier point), `1/8`, `2/8`... jusqu'à `8/8` (soit notre neuvième point). Nous obtenons alors le code suivant, les fractions remplaçant les `'...'` du code précédent:

```
#declare Ma_Spline =
spline {
linear_spline
0/8,<0, 0, 0>
1/8, <2, 2, 0>
2/8, <0, 4, 0>
3/8, <-2, 6, 0>
4/8, <0, 8, 0>
5/8, <2, 6, 0>
6/8, <0, 4, 0>
7/8, <-2, 2, 0>
8/8, <0, 0, 0>
}
```

L'incrément de valeur numérique entre deux points consécutifs de notre spline étant constant (et en l'occurrence égale à 1/8ème), nous pouvons supposer que notre balle rouge se déplacera toujours à la même vitesse sur la trajectoire du huit, tout au long de l'animation. Mais essayons maintenant de raisonner par rapport au code suivant:

```
#declare Ma_Spline =
spline {
linear_spline
0/8,<0, 0, 0>
5/8, <2, 2, 0>
5.5/8, <0, 4, 0>
6/8, <-2, 6, 0>
6.5/8, <0, 8, 0>
7/8, <2, 6, 0>
7.333/8, <0, 4, 0>
7.666/8, <-2, 2, 0>
8/8, <0, 0, 0>
}
```

Rien de fondamentalement changé, à l'exception des valeurs numériques qui différent grandement. Ici, nous démarrons bien à notre premier point lorsque `clock = 0`. Mais il nous faut les 5/8 de l'animation pour arriver au second point, soit plus de la moitié de l'animation! L'incrément entre le premier point et le second, est très important, cela se traduit donc par une vitesse très lente. Du troisième point au sixième point, en revanche, nous avons un incrément plus faible, ce qui veut dire qu'il y a une accélération entre le deuxième point et le troisième point, mais la vitesse est désormais plus rapide. Enfin, entre le sixième et le septième point, il y a encore une accélération car du septième point au dernier point, l'incrément est encore plus faible, ce qui veut dire que la vitesse s'est encore accrue. Vous noterez au passage qu'au neuvième point, nous avons atteint une valeur numérique de `8/8`, soit `1`, soit encore `clock = 1` et donc, la fin de notre animation.

Vous pouvez constater vous même ces changements de vitesse en effectuant le rendu de l'animation à l'aide du fichier `huit_spline.ini` suivant, et en saisissant les deux configurations de splines:

```
Antialias=On
Antialias_Threshold=0.1
Antialias_Depth=2
Input_File_Name=huit_spline.pov
Initial_Frame=1
Final_Frame=72
Initial_Clock=0
Final_Clock=1
Cyclic_Animation=on
Pause_when_Done=off
```

3. Types de Spline

Comme nous l'avons déjà vu dans GNU/LMag N°53, ou en ligne sur **http://www.linuxgraphic.org,** pour l'étude des prismes, POV-ray admet plusieurs types de splines. Nous allons principalement nous intéresser à deux de celles-ci en particulier; à noter que ce sont les deux seules que j'utilise habituellement, mais comme j'ai souvent de sales habitudes, rien ne vous empêche d'employer les autres, en fonction de vos besoins.

Linear_spline: si vous spécifiez ce type de spline, de simples segments de droite sont tracés entre chacun des points définissant celle-ci (figure 02). Son usage est courant pour les phases de « débogage » de vos splines, lorsque vous cherchez encore la trajectoire optimale de voter objet au travers de votre scène. Le principal inconvénient, c'est que les changements de trajectoires sont très violents (comprenez par là que les virage sont très serrés...).

```
#declare Ma_Spline =
spline {
linear_spline
0/8,<0, 0, 0>
1/8, <2, 2, 0>
2/8, <0, 4, 0>
3/8, <-2, 6, 0>
4/8, <0, 8, 0>
5/8, <2, 6, 0>
6/8, <0, 4, 0>
7/8, <-2, 2, 0>
8/8, <0, 0, 0>
}
```

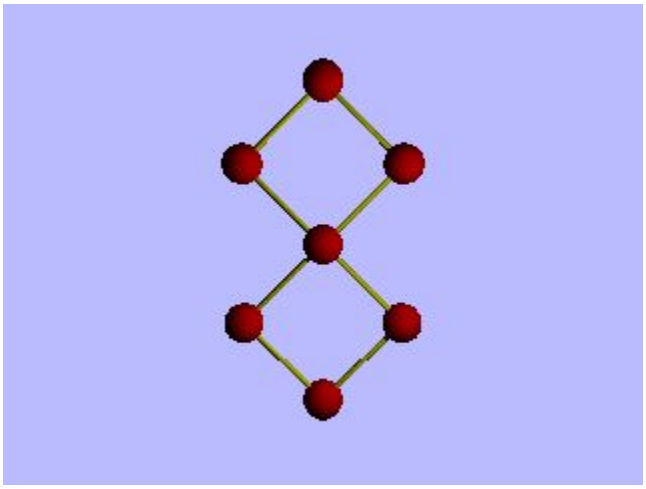

Figure 02: la spline (en jaune) passe linéairement par les points (en rouge) qui la définissent

natural_spline: ce type se spline est celui qui produit les transitions les plus douces entre deux points (figure 03). Malheureusement, la courbe qui passe entre les points est du troisième ordre, et pour définir l'inflexion à l'origine de la courbe et l'inflexion à l'extrémité de la courbe, il vous faut définir des points de contrôle supplémentaires. Nous allons donc encadrer notre spline actuelle par deux points supplémentaires (représentés en vert).

```
#declare Ma_Spline =
spline {
natural_spline
-1/8,<-2,-1,0>
0/8,<0, 0, 0>
1/8, <2, 2, 0>
2/8, <0, 4, 0>
3/8, <-2, 6, 0>
4/8, <0, 8, 0>
5/8, <2, 6, 0>
6/8, <0, 4, 0>
7/8, <-2, 2, 0>
8/8, <0, 0, 0>
9/8,<.25,-2,0>
}
```

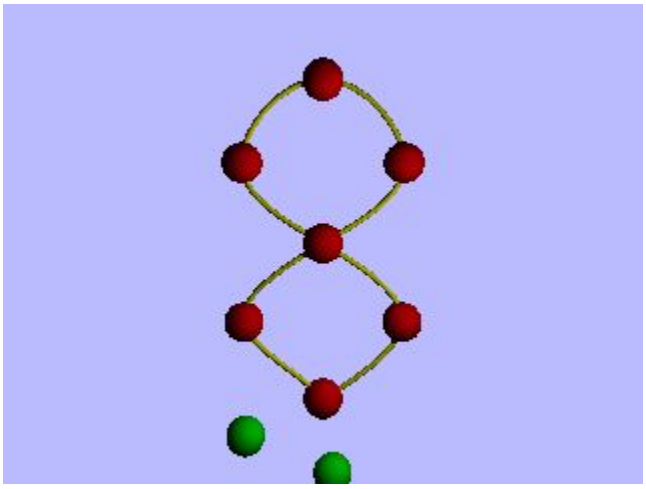

Figure 03: la spline se courbe naturellement pour passer au travers des points rouges, les points verts définissent son inflexion au départ et à l'arrivée.

4. Animation le long d'une spline

Nous allons dans un premier temps bâtir une scène d'expérimentation très simple, que nous allons très simplement nommer `anim_spline.pov`. L'objectif sera de faire tourner autour de lui-même un tore, dans le sens trigonométrique (le sens inverse des aiguilles d'une montre) et de faire sautiller, le long de sa génératrice, une petite balle rouge, dans le sens anti-trigonométrique (le sens des aiguilles d'une montre). Mais toute scène nécessitant une source de lumière et une caméra, nous allons commencer à nous occuper en premier de ces deux éléments. Nous les voulons simples et sans fioritures, afin de ne pas se compliquer la vie.

```
light_source {
<4, 5, -5>, rgb <1, 1, 1>
}

camera {
location <5, 5, -5>
look_at <0, 0, 0>
}
```

Afin de bien distinguer notre scène et éviter d'avoir un horrible et désagréable fond noir, j'ai opté, pour changer, pour un « ciel » d'un bleu léger. Pour cela, déclarons un objet de type sky_sphere, et dans son bloc de description, insérons un pigment de la couleur appropriée.

```
sky_sphere {
pigment {
color rgb <0.752941, 0.752941, 1>
}
}
```

Voilà pour le décor. Nous allons maintenant nous occuper de la création du tore (figure 04), et nous essaierons de lui donner une apparence un peu fantaisiste. Nous choisirons le rayon majeur de sorte qu'il soit cinq fois plus important que le rayon mineur, avec d'avoir un objet qui tient plus du joint torique que du Donut cher à Homer Simpson (« huuuum, regarde par terre, Marge! Un Donut à peine mangé! ») et nous lui donneront une échelle (`scale 6`) qui lui permettra d'occuper un peu mieux le centre de la scène. Pour l'habillage, nous déclarerons un pigment de type damier en spécifiant, pour les deux couleurs, de très traditionnels blanc et noir. Quant à la finition, nous opterons pour une surbrillance de type `Phong` ainsi que pour de légers et discrets reflets. Enfin, comme notre tore sera animé, nous nous inspirerons de ce que nous avons appris au cours de la première partie pour le faire tourner autour de son axe local y, en exprimant le souhait suivant: au terme de l'animation, le tore aura fait un demi-tour (180°) autour de lui-même, dans le sens inverse des aiguilles d'une montre, ce qui se traduit par une valeur de rotation autour de l'axe y de `-clock*180`.

```
torus {
0.5, 0.1
pigment {
checker
color rgb <0, 0, 0>
```

```
color rgb <1, 1, 1>
scale 0.2
}
finish {
phong 0.75
phong_size 20
reflection {
rgb <0.1, 0.1, 0.1>
}
}
scale 6
rotate <0, -clock*180, 0>
}
```

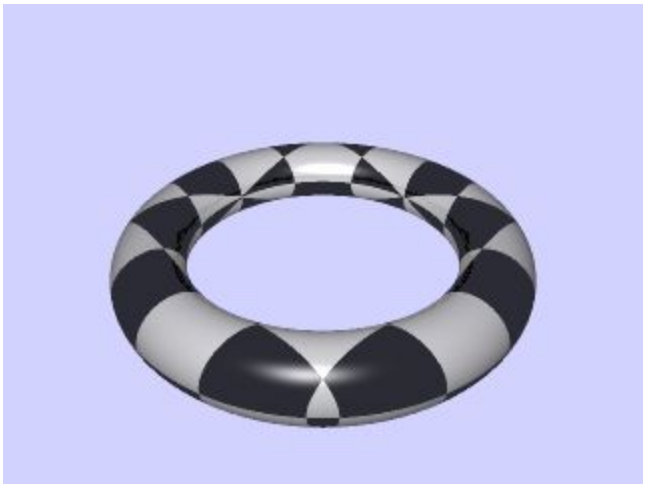

Figure 04: notre scène de base. Dur de le voir en PDF, mais elle est animée! ;)

Maintenant que nous avons une bonne visualisation de notre scène de base, allons-y pour la définition d'une petite spline, pas très compliquée: on part sur la base d'un hexagone dont tous les points sont à une même altitude (1.1 sur l'axe y). Ensuite, coupons virtuellement chaque segment en deux, puis surélevons chaque point de coupure d'une demi-unité vers le haut (1.1 devient 1.6 sur l'axe y). Nous devrions donc obtenir le code suivant pour notre spline (nous en restons au cas très simple de la spline linéaire, avec des incréments fixes d'un point à l'autre, égaux à 1/12ème à chaque fois):

```
#declare Ma_Spline =
spline {
linear_spline
0/12,<1.5, 1.1, 2.6>
1/12, <2.25, 1.6, 1.3>
2/12, <3, 1.1, -0>
3/12, <2.25, 1.6, -1.35>
4/12, <1.5, 1.1, -2.7>
5/12, <-0.05, 1.6, -2.65>
6/12, <-1.6, 1.1, -2.6>
7/12, <-2.35, 1.6, -1.25>
8/12, <-3.1, 1.1, 0.1>
9/12, <-2.4, 1.6, 1.3>
10/12, <-1.7, 1.1, 2.5>
11/12, <-0.1, 1.6, 2.55>
12/12, <1.5, 1.1, 2.6>
}
```

L'image (figure 05) qui suit symbolise la spline, et donc la trajectoire que suivra très prochainement notre petite balle rouge. Les petits curieux qui se demandent comment nous avons procédé pour illustrer la spline trouveront leur bonheur dans le bout de code suivant. Nous ne l'expliquerons pas ici, nous contentant de vous renvoyer à GNU/LMag N°64 (ou en ligne sur **http://www.linuxgraphic.org**) pour l'usage des boucles avec POV-ray.

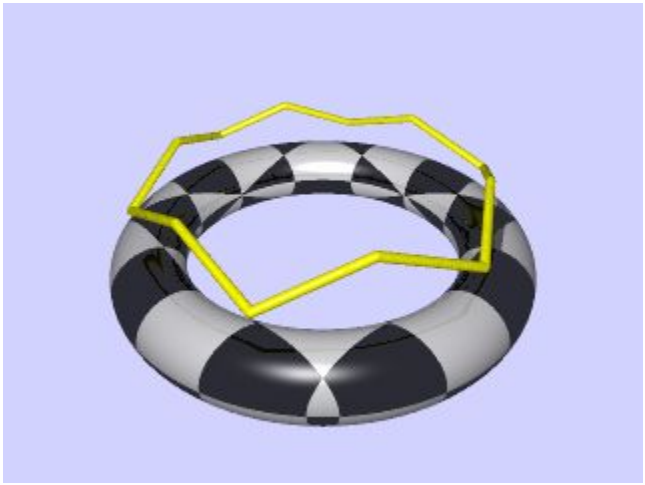

Figure 05: les segments jaunes figurent notre spline

```
#declare Nr = 0;
#declare EndNr = 1;
#while (Nr< EndNr)
sphere{
<0,0,0>,0.07
texture{ pigment{color rgb <1,1,0>}
finish {ambient 0.15 diffuse 0.85 phong 1}
}
translate Ma_Spline(Nr)
}
#declare Nr = Nr + 0.001;
#end
```

Et c'est là qu'intervient la magie des splines. Nous allons définir notre petite balle rouge, comme nous le ferions de tout autre objet statique (figure 06). Nous allons ensuite lui donner l'option de transformation translate, mais au lieu de spécifier des composantes x, y et z, nous allons faire appel à la fonction `Ma_spline(...)` pour ordonner à la balle rouge de conformer sa localisation, à tout instant, avec celle de la spline. Il ne nous reste plus alors qu'à définir les bornes de conformation, par rapport à la durée de l'animation. Ainsi, si nous voulons que la balle rouge fasse un tour complet au cours de l'animation, nous spécifions entre parenthèses, à la place des `(...)`, la variable `(clock)`. Si, par exemple, nous souhaitons qu'elle ait réaliser un demi-tour seulement, ou deux tours, nous spécifierons respectivement `(clock/2)` ou `(2*clock)`. Pour ce cas volontairement simple, nous nous contenterons d'un simple `(clock)` mais rappelez-vous qu'il est là possible d'utiliser des fonctions complexes pour simuler accélérations ou ralentissements, par exemple, si l'usage seuls des valeurs numériques aux points de votre spline (voir ci-avant, Définition d'une spline) ne vous suffit pas.

```
sphere {
<0, 0, 0>, 0.5
texture {
pigment {
color rgb <1, 0, 0>
}
finish {
phong 1
phong_size 80
}
}
scale 1
rotate <0, 0, 0>
translate Ma_Spline(clock)
}
```

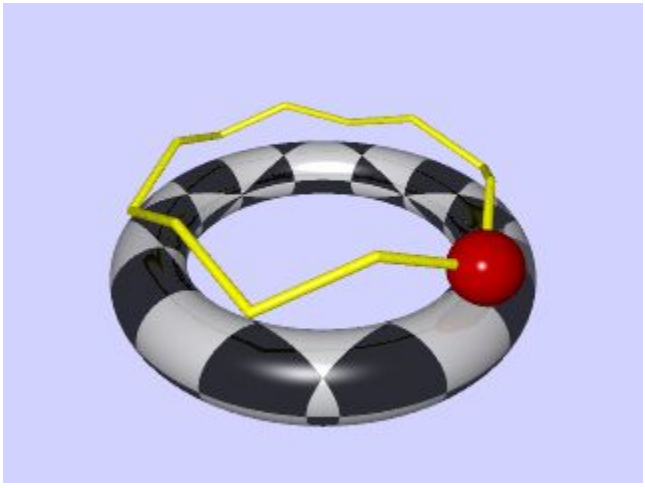

Figure 06: la petite balle rouge est désormais prête à suivre la trajectoire qui lui a été définie!

Notre fichier `*.pov` est désormais complet, prêt à être animé. Il ne nous reste finalement plus qu'à définir les paramètres de l'animation, au travers d'un fichier `anim_spline.ini` qu'il nous reste à écrire. A nouveau, nous allons réutiliser les connaissances acquises lors de la première partie de cet article.

```
Antialias=On
Antialias_Threshold=0.1
Antialias_Depth=2
Input_File_Name=anim_spline.pov
Initial_Frame=1
Final_Frame=48
Initial_Clock=0
Final_Clock=1
Cyclic_Animation=on
Pause_when_Done=off
```

Dans ce fichier, nous activons l'anti-crénelage pour obtenir un résultat de meilleur qualité, et prenons garde à bien désigner le fichier contenant la scène à animer (`anim_spline.pov`). Nous nous contenterons d'une animation de 2 secondes, prévue à une vitesse de 24 images par seconde. Notre animation prend donc fin après avoir rendu la 48ème image. Enfin, puisque nous souhaitons que tant notre balle que notre tore à carreaux puise tourner indéfiniment, nous activons l'option d'animation cyclique. Et voilà (figure 07)!

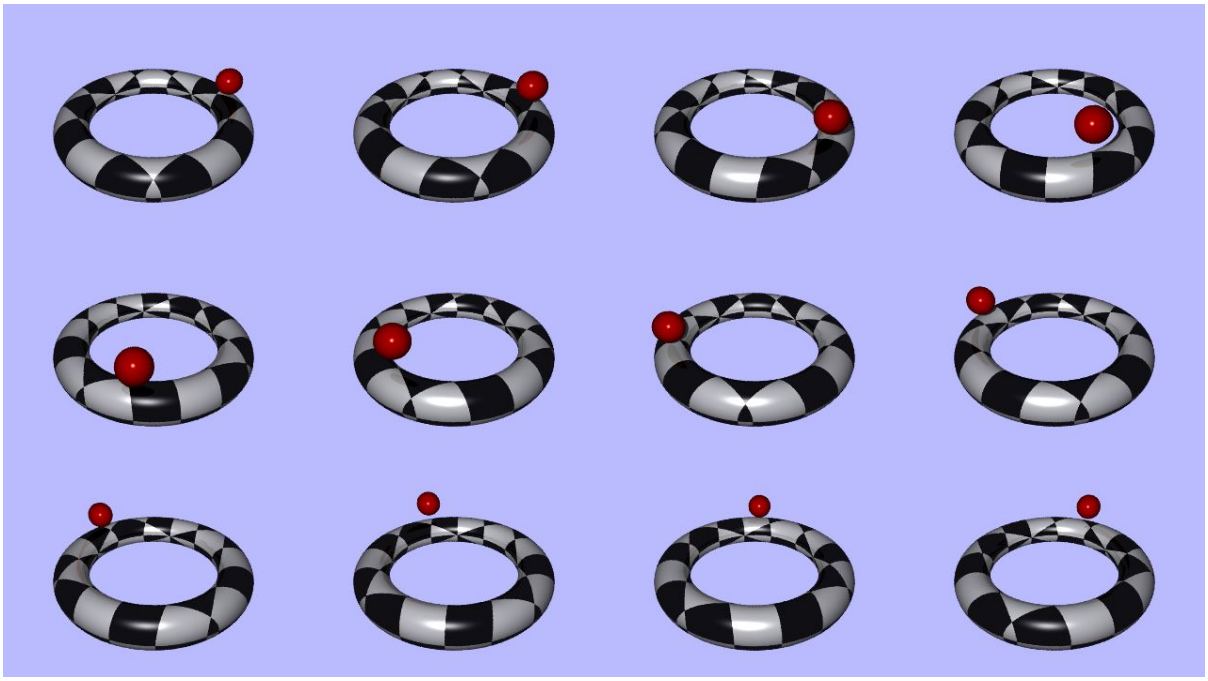

Figure 07: un échantillonnage de l'animation que l'on obtient, équivalent à du 6 images par seconde

5. Le montage des animations, 2ème partie

Le mois passé nous avons vu deux façons distinctes de produire de petites animations. La première (la production de GIF animés) demandait un travail long et répétitif, puisqu'il était nécessaire de créer une succession de calques vides (un par image de votre animation) et d'y coller l'image chronologiquement correspondante rendue par POV-ray. Un processus long et peu agréable... La seconde (la production d'un film AVI) nous a permis de voir que Blender, outre le fait d'être un excellent modeleur, animateur et moteur de rendu, était également un excellent petit studio de montage, pour le même prix. Malheureusement, ce dernier ne permet pas (encore!) de produire directement des films en mpeg ou en divX. C'est précisément ces deux types de vidéos auxquels nous allons nous attaquer maintenant.

5.1 Un film xvid avec Transcode

Cette méthode m'a directement été soufflée par le Rédac' Chef adoré de LinuxMag, aussi sera-t-il de bon ton, pour l'assistance, de ponctuer la lecture de cette partie par quelques ooooh et quelques aaaah hypocritement mais savamment placés. Plus sérieusement, pour moi, **transcode** était surtout cet obscur couteau suisse de la vidéo qui permettait à des interfaces graphiques de « ripper » des DVD-vidéos en **xvid**. Pour avoir observé les commandes émises par **DVD::rip**, par exemple, je savais que transcode était riche en possibilités et en options. Mais j'étais loin de me doutait, en bon néophyte de la ligne de commande que je suis, qu'avec les options idoines, transcode me permettrait également, un jour, d'assembler des vidéos à partir des images éparses générées par les différents moteurs de rendu que j'emploie.

Mais passons à la pratique. Supposons que je viens d'effectuer le rendu des 48 images qui composent les 2 secondes de mon animation. Toutes les images sont rangées, là, dans mon disque dur, dans le répertoire `/home/olivier/tmp/` sous des noms variant de `image01.png` à `image48.png`. Nous allons générer l'animation en deux toutes simples étapes:

Dans un premier temps, nous allons prendre la main dans un environnement en ligne de commande; nous pouvons bien sûr appeler un Terminal ou un Shell depuis notre environnement graphique préféré, ou bien simplement changer d'écran (en appuyant sur CTRL+ALT+F1 par exemple) et nous identifier en tant qu'utilisateur ayant des droits de lecture-écriture sur le répertoire où sont contenues les images.

Ensuite, nous allons trier les images dans l'ordre alphanumérique et lister le résultat dans un fichier que nous appellerons `ma_liste`. Dans la console, cela donnera quelque chose comme:

```
cd /home/olivier/tmp
ls *.png | sort > ma_liste
```

Il ne nous reste plus qu'à passer notre liste à transcode en tant qu'argument, et celui-ci se chargera de composer la vidéo. Dans la console, cela donnera la ligne unique suivante:

```
transcode -i ma_liste -x imlist,null -g 320x240 -y xvid,null -f 24 -o ma_video.avi
-H 0
```

Quelques uns des arguments sont particulièrement important pour nous autres animateurs: ma_liste, bien évidemment, qui contient toutes les images à assembler, dans l'ordre où elles doivent être assemblées (`-i ma_liste`); la résolution de notre vidéo, ici du 320x240 (`-g 320x240`); la vitesse de l'animation, ici du 24 images par seconde (`-f 24`); et enfin le fichier vidéo sous lequel notre animation est enregistrée (`-o ma_video.avi`) avec une extension conforme à l'encodeur employé (ici `xvid`).

A noter qu'il vous est également possible d'employer transcode pour encoder votre animation en simple MPEG. Il suffit alors d'employer, à la place de la ligne de commande précédente, celle qui suit:

```
transcode -i ma_liste -x imlist,null -g 320x240 -y ffmpeg,null -f 24 -o
ma_video.mpeg -F mpeg1video -z -k
```

5.2 Un film MPEG avec mpeg_encode

Mais transcode est peut-être un utilitaire qui vous rebute, ou peut-être préférez-vous les outils qui ne font qu'une seule chose, mais qui le font bien. Alors **mpeg_encode** est fait pour vous, malgré son âge désormais respectable (la dernière version date de 1995!) et son absence totale de suivi

depuis (ce qui en soit pourrait vous refroidir). Nous allons utiliser ce petit utilitaire qui s'occupera de l'ingrate tâche d'assembler les différentes images de l'animation en vidéo au format MPEG. Il fonctionne sur la base d'un fichier de paramètres, simplement nommé '`mon_film.param`' qui indiquera à l'encodeur ce qu'il a à faire. Voici un exemple de fichier qui permet de transformer des images au format PPM en vidéo MPEG. Il suppose que vous ayez rendu toutes vos images au format PPM (mpeg_encode n'admet malheureusement qu'un petit nombre de formats en entrée, et le seul qui soit compatible des formats de sortie standards de POV-ray est justement le format PPM) et que celles-ci sont, comme précédemment, stockées dans le répertoire `/home/olivier/tmp`. Au nombre de 48, elles sont prévues pour former une animation de deux secondes à une vitesse de 24 images par seconde.

```
PATTERN IBBBPBBBPBBBPBBBPBBBP
#-------------------------------
OUTPUT /home/olivier/tmp/mon_film.mpg
#-------------------------------
BASE_FILE_FORMAT PPM
INPUT_FORMAT UCB
INPUT_CONVERT *
GOP_SIZE 16
SLICES_PER_FRAME 1
#-------------------------------
INPUT_DIR /home/olivier/tmp
#-------------------------------
INPUT
#-------------------------------
mon_fichier*.ppm [01-48]
#-------------------------------
END_INPUT
PIXEL HALF
RANGE 10
PSEARCH_ALG LOGARITHMIC
BSEARCH_ALG CROSS2
IQSCALE 3
PQSCALE 4
BQSCALE 6
REFERENCE_FRAME ORIGINAL
BUFFER_SIZE 327680
FRAME_RATE 24
#-------------------------------
BIT_RATE 820100
#-------------------------------
```

Les paramètres les plus importants sont:

OUTPUT: vous spécifiez ici le nom complet du fichier vidéo MPEG produit, chemin de stockage compris.

INPUT_DIR: il s'agit tout simplement du répertoire où sont stockées les images PPM.

*** [0001-9999]**: vous spécifiez ici les noms du premier et du dernier fichier PPM, en omettant l'extension. Seulement 4 chiffres sont permis par nombre dans les crochets, mais vous pouvez remplacer le joker '*' par le corps du fichier. Par exemple, une animation avec les fichiers `mon_fichier01.ppm` à `mon_fichier48.ppm` sera traduit par une ligne `mon_fichier*.ppm [01-48]`

FRAME_RATE: il s'agit du nombre d'images par seconde de votre vidéo.

Il ne vous reste plus qu'à enregistrer ce fichier sous un nom explicite dans votre répertoire de travail (par exemple `/home/olivier/tmp/mon_film.param`) et dans une console, taper la commande suivante:

```
mpeg_encode /home/olivier/tmp/mon_film.param
```

Astuce: La seule subtilité de cette méthode concerne le rendu des images de votre animation directement au format PPM. Pour y parvenir, plutôt que de taper:
`povray mon_fichier.ini`

pour obtenir les images de votre animation au format PNG standard, vous vous essaierez à:
`povray mon_fichier.ini -FP`
et obtiendrez les mêmes images au format PPM.

6. Conclusion

Nous en avons fini de notre tour des principales possibilités d'animation avec POV-ray, et des moyens (parmi des dizaines de solution!) de transformer des séquences d'images fixes en animations. Bien sûr, nous n'avons pas consacré de trop nombreuses explications aux splines en elles-mêmes, juste le minimum nécessaire pour avancer efficacement. Une fois de plus, nous vous recommandons chaleureusement le didacticiel de Friedrich A. Lohmüller dont vous trouverez l'URL en fin d'article. Celui-ci présente notamment la macro `Spline_Trans` qui vous permettra de simuler le roulis et/ou le dérapage d'une moto dans un virage, par exemple.

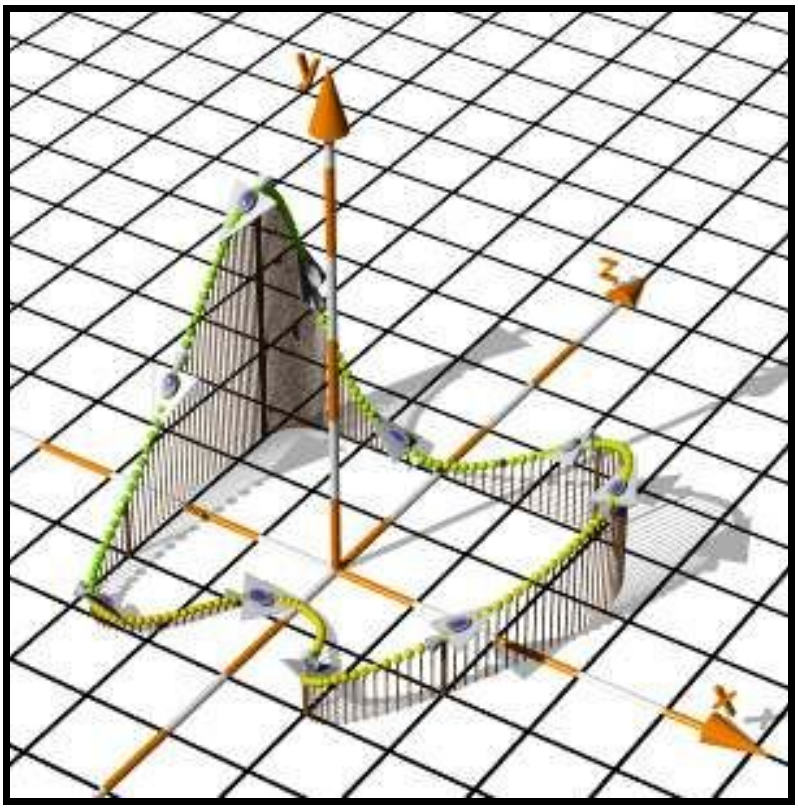

Figure 08: exemple d'usage de la macro Spline_Trans

7. Liens

-L'excellent didacticiel de Friedrich A. Lohmüller: Animations with POV-Ray: **http://www.f-lohmueller.de/pov_tut/animate/index.htm**

-La homepage de kpovmodeler: **http://www.kpovmodeler.org**

-La homepage de povray (version courante: v3.6): **http://www.povray.org**

-La documentation officielle en français de povray: **http://users.skynet.be/bs936509/povfr/index.htm**

-La homepage de transcode: **http://www.transcoding.org/**

-La homepage de mpeg_encode: **ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/mpeg/encode/**