# Usage des boucles avec POV-Ray

*Merci à Diamond Editions pour son aimable autorisation pour la mise en ligne de cet article, initialement publié dans Linux Magazine N°64*

**Olivier Saraja - olivier.saraja@linuxgraphic.org**

**Certains objets complexes peuvent être très simplement créés par l'usage de boucles « tant que » (while) directement dans le script de description des objets de POV-ray. Le procédé tient de la duplication paramétrable, puisqu'il va s'agir de définir un objet, et de lui donner de façon itérative de nouvelles positions, dimensions et angles de rotation entre chaque boucle de calcul. Bien sûr, cette approche sous-entend que les objets seront placés de façon régulière, selon des règles que vous serez toutefois libre de fixer.**

L'avantage des boucles est d'économiser considérablement la mémoire, puisque la géométrie de l'objet dupliqué est définie une seule fois, et que seuls ses paramètres de transformation (`translate`, `scale`, `rotate`) varient d'un clone à l'autre. En revanche, plus vous aurez de boucles, plus vous obtiendrez de clones et, par conséquent, plus les temps de parsing et de rendu de l'image seront long. On n'a jamais rien sans rien, me direz vous, mais c'est un lieu commun dont nous nous sommes depuis longtemps accommodés avec POV-ray, dont les maître-mots sont rigueur et... patience! Au passage, cela vous permettra de définir, à l'aide de primitives simples, des objets de structure pourtant complexe (voir Figure 01).

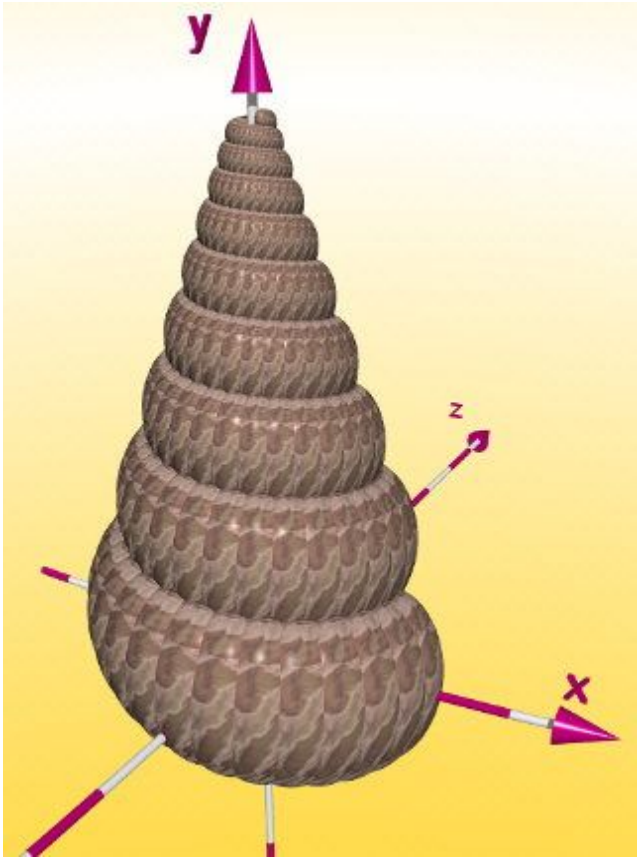



*Figure 01: Un exemple de ce que l'on apprendra à réaliser*

*(image de Friedrich A. Lohmüller)*

## 1. Syntaxe élémentaire

Une boucle tant que commence par une ligne `#while (...)` et termine par une ligne `#end`. Il n'y a pas de limite au nombre d'actions que peut contenir une boucle `#while`, mais idéalement l'une

d'elle est de nature itérative, afin de permettre de faire évoluer un état de départ vers l'état final qui permet de sortir de la boucle `#while`. Il est donc souvent utile de définir à l'aide de variables #declare les états initiaux et finaux de votre système, ainsi que nous le verrons plus loin.

```
#while ( [condition] )
[action 1]
[action 2]
[...]
[action n]
#end
```

# 2. Notre scène de base

Comme d'habitude, nous nous en tiendrons à une scène très basique, orientée sur la seule mise en valeur du résultat de nos expérimentations. Au cours de l'article, nous ferons toutefois varier les paramètres de la caméra au fur et à mesure des exemples, afin de s'assurer que nos travaux apparaîtront le plus avantageusement possible lors des rendus successifs. Le premier bloc décrira donc une lampe classique, tandis que le second décrira notre caméra. Le troisième bloc figure le sol, en fait un plan doté d'un pigment en forme de damier bicolore. Enfin, j'ai trouvé commode de figurer en rouge l'origine du repère cartésien de la scène, et en bleu l'objet à dupliquer, à l'aide de petites sphères colorées (voir Figure 02).

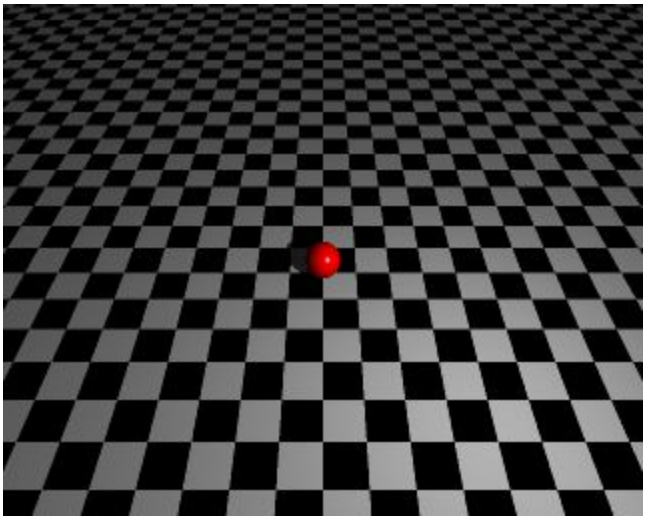


*Figure 02: la scène de départ*

```
// Source de lumière
light_source {
<4, 5, -5>, rgb <1, 1, 1>
}

// Caméra
camera {
perspective
location <0, 10, -10>
look_at <0, 0, 0>
}

// Sol
plane {
<0, 1, 0>, -0.5
pigment {
checker
color rgb <1, 1, 1>
color rgb <0, 0, 0>
}
scale 1
rotate <0, 0, 0>
translate <0, 0, 0>
}

// Origine (boule rouge)
```

```
sphere {
<0, 0, 0>, 0.51
texture {
pigment {
color rgb <1, 0, 0>
}
finish {
ambient rgbft <0.15, 0.15, 0.15, 0.15, 0.15>
diffuse 0.85
phong 1
}
}
}

// Sphères dupliquées (boules bleues)
#declare Sphere_bleue = sphere {
<0, 0, 0>, 0.5
texture {
pigment {
color rgb <0, 0, 1>
}
finish {
ambient rgbft <0.15, 0.15, 0.15, 0.15, 0.15>
diffuse 0.85
phong 1
}
}
}
```

# 3. Approche linéaire: notre première boucle

Nous allons mettre en oeuvre, tout au long de cet article, une boucle de type `#while`. A chaque étape de calcul, elle répétera les mêmes opérations jusqu'à ce qu'une condition donnée soit atteinte. Il faut donc définir un état de départ, un état final et la façon dont va évoluer le système entre deux incréments.

L'état de départ sera attribué à la variable (que nous pouvons nommer arbitrairement) `NiniX` (« valeur N initiale dans la direction X »). De même, l'état final sera attribué à la variable `NfinX` (« valeur N finale dans la direction X »). Par soucis de simplicité, établissons également une variable de travail `NX` par défaut égal à l'état de départ `NiniX`. Le but de la boucle `#while` sera de faire varier `NX` entre la valeur `NiniX` et `NfinX`.

Considérons maintenant le code suivant, à insérer à la suite du code de notre scène de base. Il nous permet de disposer des boules le long de l'axe X, depuis X = -3 jusqu'à X = +3 avec un pas régulier de 1:

```
// Boucle simple
#declare NiniX = -3; //Etat de départ
#declare NfinX = 3; //Etat final
#declare NX = NiniX;
#while (NX <= NfinX)
object{Sphere_bleue translate <NX,0,0>}
#declare NX = NX + 1; // Définition du pas
#end
```

L'état de départ est donc fixé à -3 et l'état final à +3. Par commodité, la variable de travail `NX` est initialisée avec la valeur de `NiniX`. Pour le reste, il suffit de « lire » le code: tant que `NX` est inférieur ou égal à `NfinX`, placer une sphère bleue à l'abscisse `NX` et augmenter de 1 la variable de travail. Nous obtenons le résultat (très simple) de l'illustration suivante (voir Figure 03).

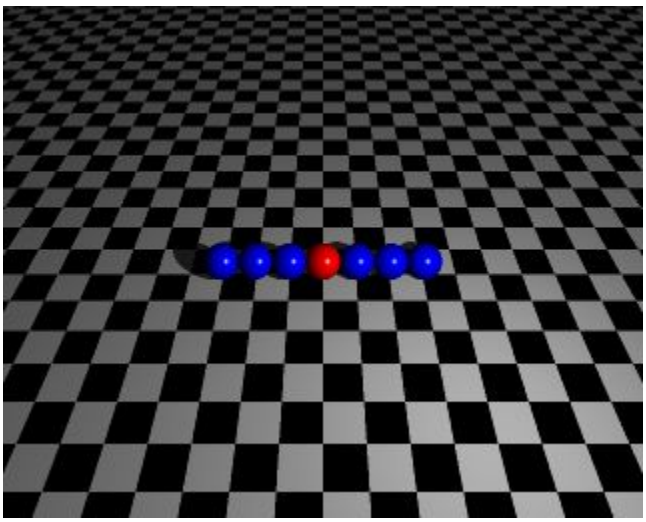


*Figure 03: notre premier résultat!*

Pour mieux visualiser le fonctionnement de la boucle `#while`, nous allons faire varier les variables de départ et de fin. Par exemple, avec:

```
#declare NiniX = -3;
#declare NfinX = 1;
```

nous allons placer, toujours sur l'axe X, des boules depuis X = -3 jusqu'à X = +1 avec un pas régulier de 1. Nous obtenons alors 3 boules bleues sur la gauche de la boule rouge, et une boule bleue la droite (voir Figure 04).

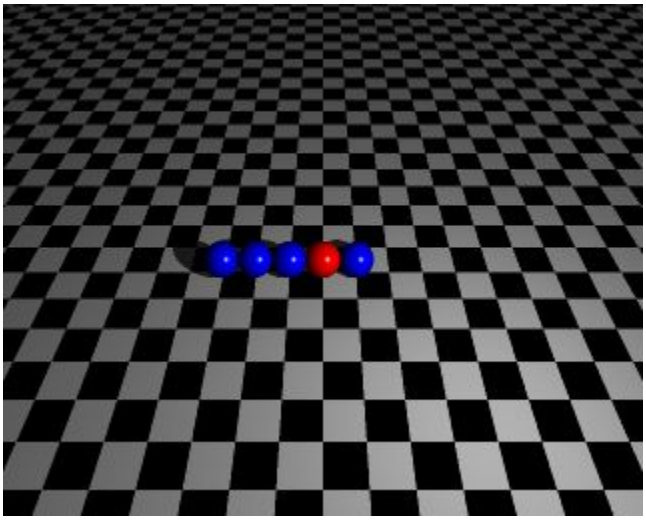


*Figure 04: avec NiniX = -3 et NfinX = 1*

Bien sûr, nous pouvons fixer différemment le pas de l'incrément. D'abord, posons:

```
#declare NiniX = -6;
#declare NfinX = 6;
```

puis, au lieu de:

```
#declare NX = NX + 1;
```

essayons:

```
#declare NX = NX + 2;
```

La notion de pas est intéressante. Vous pouvez ajouter ou soustraire n'importe quel nombre à votre variable de travail, même une autre variable que vous auriez défini avant. De même, vous pouvez diviser ou multiplier votre variable de travail par un nombre ou une variable quelconque. La Figure 05 suivante témoigne de l'influence du pas d'incrément sur le nombre de boules dupliquées le long de l'axe X.

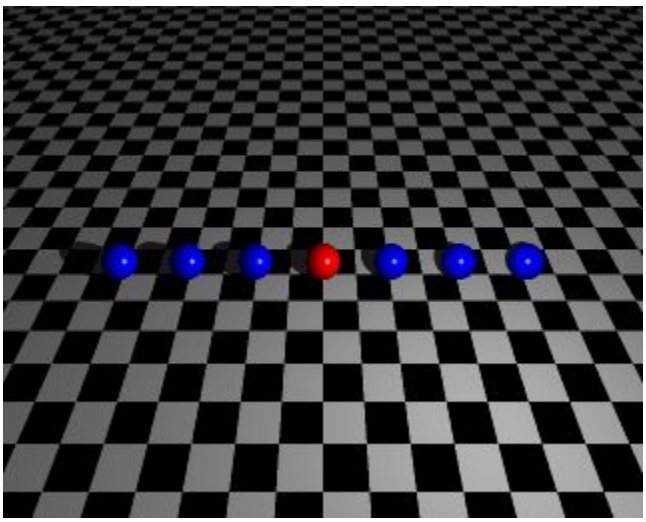


*Figure 05: avec NiniX = -6, NfinX = 6 et un pas de 2 au lieu de 1*

# 4. Imbriquer des boucles

Il vous est possible de spécifier à peu près n'importe quelle action dans votre boucle `#while`. Nous allons essayer, pour nous amuser, de spécifier une autre boucle `#while` en guise d'action de la première. Mais alors que la première boucle dispose de façon régulière des boules bleues le long de l'axe X, nous allons demander à la seconde de les disposer le long de l'axe Z.

Essayons le code suivant:

```
#declare NiniX = -3;
#declare NfinX = 3;
#declare NX=NiniX;
#while (NX <= NfinX)
#declare NiniZ = -3;
#declare NfinZ = 3;
#declare NZ=NiniZ;
#while (NZ <= NfinZ)
object{Sphere_bleue translate <NX,0,NZ>}
#declare NZ = NZ + 1;
#end
#declare NX = NX + 1;
#end
```

Nous avons pris soin de déclarer des variables initiales et finales pour l'axe X (`NiniX` et `NfinX`), et nous avons ouvert notre première boucle `#while` pour gérer le placement des boules le long de l'axe X. Mais au lieu d'ordonner immédiatement le placement de la boule bleue, nous ouvrons une seconde boucle, destinée, celle-ci, à gérer le placement des boules dans la direction Z. Pour ce faire, nous procédons à la déclaration des variables initiales et finales pour l'axe Z (`NiniZ` et `NfinZ`) à l'intérieur de la première boucle, juste avant la seconde. Il ne nous reste plus qu'à commander le placement des boules bleues à l'aide de la ligne `object{..}` mais en prenant soin de faire varier les composantes X et Z de la fonction `translate`. Nous obtenons donc un joli « quadrillage » de boules bleues, parfaitement réglé, comme en témoigne la Figure 06 ci-après.

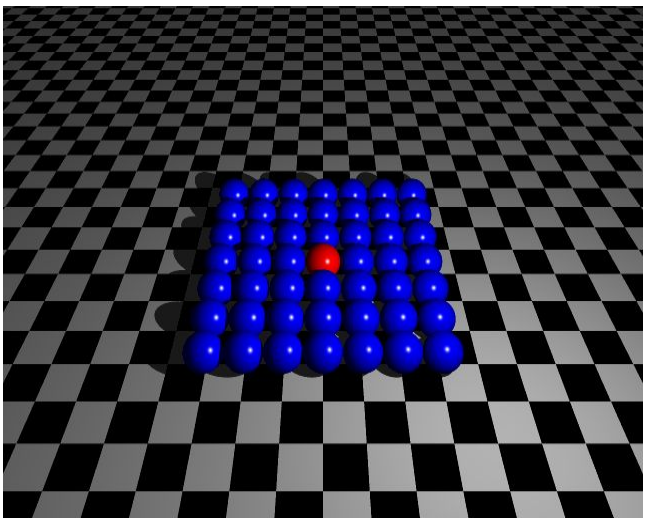


*Figure 06: test avec deux boucles imbriquées*

Est-il aussi simple que cela d'imbriquer des boucles les unes dans les autres? La réponse est oui. Pour nous en donner la preuve, nous allons imbriquer une troisième boucle `#while` afin de gérer, également, le placement des boules bleues en altitude. Le résultat est sans appel, sans plus d'imagination non plus. Cette méthode est, au plus, une méthode rapide et économe en efforts pour « remplir » des volumes simples (voir Figure 07), mais je suis persuadé que vous lui trouverez de nombreuses applications intéressantes.

```
#declare NiniX = -3;
#declare NfinX = 3;
#declare NX=NiniX;
#while (NX <= NfinX)
#declare NiniZ = -3;
#declare NfinZ = 3;
#declare NZ=NiniZ;
#while (NZ <= NfinZ)
#declare NiniY = -3;
#declare NfinY = 3;
#declare NY=NiniY;
#while (NY <= NfinY)
object{Sphere_bleue translate <NX,NY,NZ>}
#declare NY = NY + 1;
#end
#declare NZ = NZ + 1;
#end
#declare NX = NX + 1;
#end
```

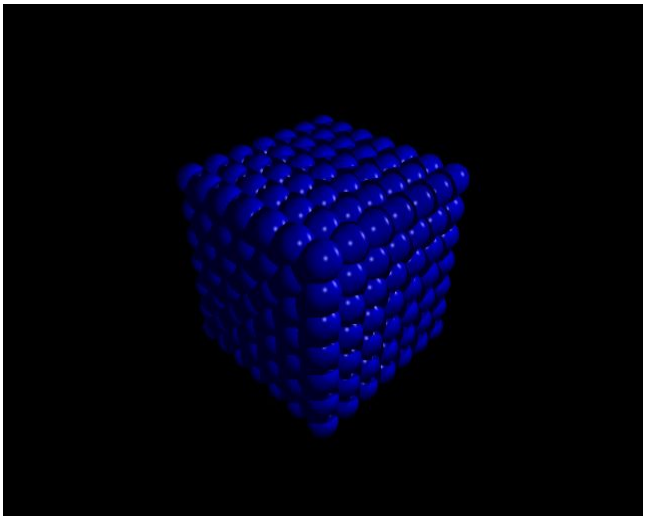


*Figure 07: test avec trois boucles imbriquées!*

Nous voici arrivés aux limites de l'approche linéaire. Nous sommes en effet emprisonnés dans un monde en trois dimensions dont il est difficile de se libérer. Certes, nous aurions très bien pu faire

l'usage de cubes au lieu de sphères, et imposer à chacun d'eux une variation angulaire pour mettre à contribution la fonction `rotate`, ou une variation d'échelle grâce à la fonction `scale`. Ce sont toutefois des notions que nous mettrons en oeuvre plus loin dans cet article, soyez rassurés. =)

# 5. Approche circulaire: une boucle simple à l'aide de la fonction Rotate

Les choses ne devraient pas être beaucoup plus complexes si l'on remplace les occurrences `translate` par des occurrences `rotate`. Nous allons commençons par définir le rayon de rotation autour de l'origine (`Rayon`). Ensuite, nous allons définir la position angulaire de la première boule (`AngleIni`) et la position angulaire de la dernière boule (`AngleFin`). Enfin, nous allons définir l'écart angulaire entre deux boules (`AnglePas`). Il ne nous reste plus qu'à écrire notre boucle `#while`, en précisant deux actions: la première pour décaler (par rapport à l'origine) la boule créée d'une valeur égale au `Rayon`; la seconde pour spécifier une position angulaire de la boule ainsi décalée. L'incrément est cette fois-ci une valeur angulaire, exprimée en degrés et non plus en unités POV-ray.

```
#declare Rayon = 5;
#declare AngleIni = 0;
#declare AngleFin = 360;
#declare AnglePas = 22.5;
#declare Angle = AngleIni;
#while (Angle < AngleFin)
object{Sphere_bleue
translate <Rayon,0,0>
rotate <0,Angle,0>
}
#declare Angle=Angle+AnglePas;
#end
```

Vous remarquerez que l'ordre des fonctions `translate` et `rotate` n'est pas indifférent. Ces deux fonctions s'appliquent en effet par rapport à l'origine; il en résulte (voir Figure 08) les observations suivantes:

- `Translate` puis `Rotate`: on décale d'abord la boule par rapport à l'origine, de sorte à créer une sorte de bras de levier fictif. La fonction `Rotate`, qui intervient après, fait donc pivoter la boule au bout de son bras de levier de l'angle fixé. On obtient donc des clones réparties régulièrement en cercle.
- `Rotate` puis `Translate`: comme la fonction s'applique par rapport à l'origine, la fonction `Rotate` fait tourner la boule (toujours placée à l'origine) autour d'elle-même. La fonction `Translate`, intervenant après, ne fait donc que décaler la boule sur l'axe des X. Toutes les boules, au lieu de se répartir en cercle, s'entassent donc dans le même espace physique!

*Figure 08: Translate d'abord, puis Rotate...: une distribution circulaire*

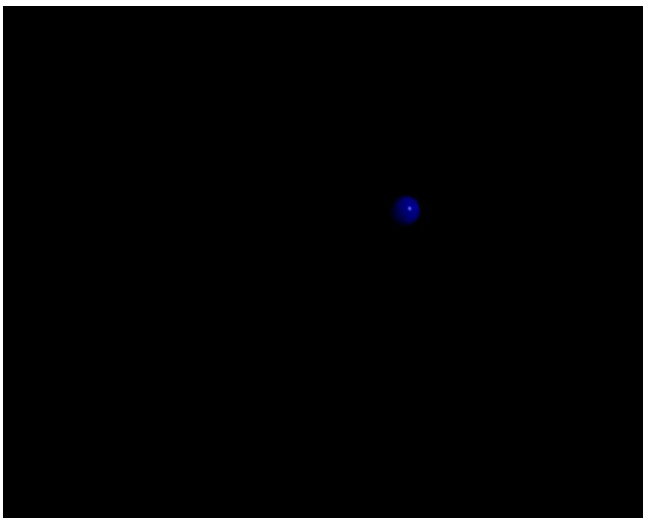


*Figure 08bis: Rotate d'abord, translate ensuite...: une distribution pour le moins ponctuelle!*

Cette boucle n'est toutefois intéressante que dans le cas simple où vous souhaitez piloter votre résultat final par l'angle séparant deux boules. Dans d'autres cas, cependant, vous préférerez spécifier le nombre de boules composant votre « anneau » et laisser POV-ray calculer lui-même le nombre de boules le constituant. Ce n'est guère plus difficile que ce que nous avons déjà réalisé, il suffit de changer les variables d'initialisation, ainsi que notre gestion de la fonction `Rotate`.

Nous définissons donc le nombre de boules qui seront placées sur notre « anneau » (`Ntotal`) ainsi que, tout à fait optionnellement, l'arc de l'anneau à construire (`Angle`, compris entre 0° et 360°). Comme précédemment, nous allons dans un premier temps décaler la boule à placer d'une distance par rapport à l'origine égale au `Rayon`. Si nous souhaitons bâtir l'anneau entier, il faudra que la dernière boule tourne de 360°; si nous souhaitons bâtir l'anneau avec 20 boules, il en résulte que la première boule devra être tournée de 1/20ème de 360° autour de l'origine. La relation paramétrique pour que POV-ray réalise toutes ces opérations automatiquement est alors très simple à déduire: chaque boule sera tournée de `Angle * N / Ntotal` autour de l'axe Y, et c'est cette formule que nous allons saisir dans la composante Y de notre fonction `rotate`.

```
#declare Rayon = 5;
#declare Ntotal = 20;
#declare Angle = 360;
#declare N = 0;
#while (N < Ntotal)
object{Sphere_bleue
translate <Rayon,0,0>
rotate <0,Angle*N/Ntotal,0>}
#declare N = N + 1;
#end
```

Vous pouvez bien sûr augmenter le nombre de boules `Ntotal` afin de resserrer les « perles » de votre collier virtuel, ou spécifier un `Angle` maxi à respecter pour n'obtenir qu'un arc d'anneau. La figure 09 qui suit illustre bien ces possibilités. Il ne me reste plus qu'à attirer votre attention sur le fait que le nombre de boules est définitivement réglé et que, par conséquent, si vous diminuez l'arc de votre « anneau », vous augmentez automatiquement la densité des boules le constituant.

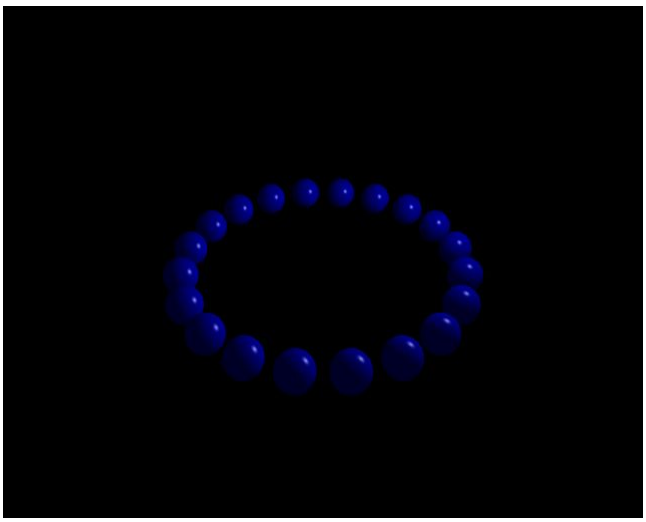


*Figure 09: Notre anneau, construit avec 20 boules...*

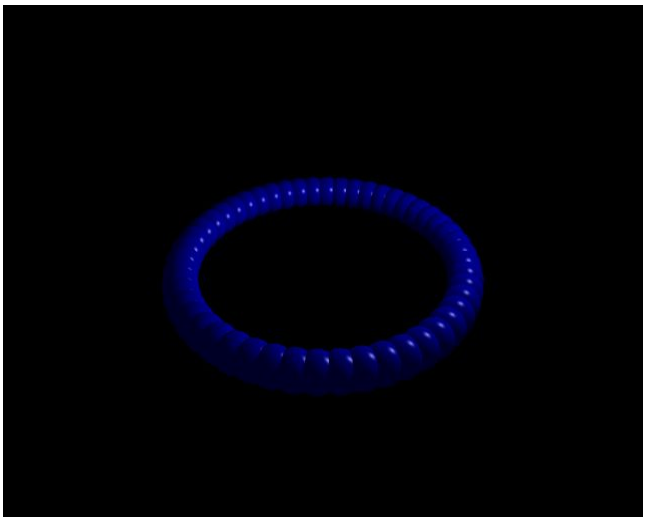


*Figure 09bis: ... le même, avec 50 boules...*

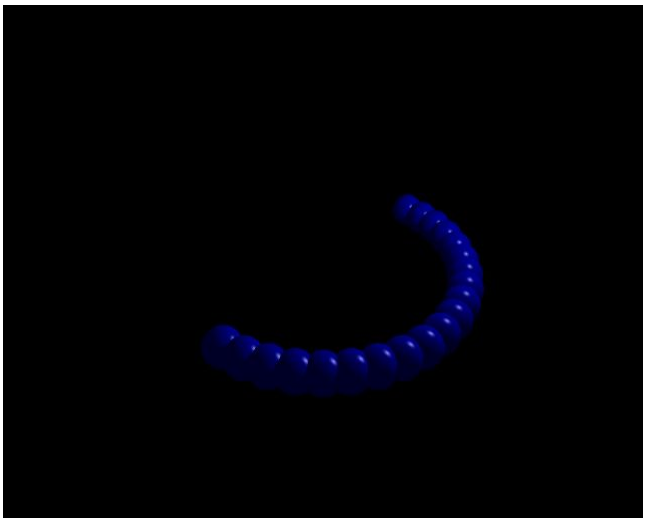


*Figure 09ter: ... et enfin 20 boules, mais réparties sur 180° seulement*

## 6. Combinaison des approches Linéaire et Circulaire: modéliser un ressort!

Voilà une opération qui devrait, désormais, vous paraître pratiquement triviale. Dans les exemples sur l'approche linéaire, nous avons vu que nous pouvions, à chaque itération de la

boucle `#while`, décaler l'objet dupliqué selon un ou plusieurs axes. Par ailleurs, l'approche circulaire nous a permis de répartir un nombre donné de clones le long d'un arc donné. Imaginons maintenant que nous imposions un très grand nombre de boules pour faire plusieurs tours complets autour de l'axe Y, tout en imposant à chaque boule une légère translation selon l'axe Y, justement. Donnons nous tout de suite quelques objectifs: doter notre ressort de 5 spires, le ressort résultant ayant un pas de un entre chaque spire et 50 boules pour faire un tour complet. Il en résulterait le code suivant:

```
#declare Rayon = 5;
#declare NbParTour = 50;
#declare Ntours = 5;
#declare Pas = 1;
#declare Ntotal = NbParTour * Ntours;
#declare N = 0;
#while (N < Ntotal)
object{Sphere_bleue
translate <Rayon,N*Pas/NbParTour,0>
rotate <0,360*Ntours*N/Ntotal,0>}
#declare N = N + 1;
#end
```

ceci avec un rayon pour notre objet `Sphere_bleue` de 0.25 au lieu de 0.50 comme dans notre scène de base. Nous pouvons maintenant augmenter le nombre de boules par tour pour obtenir un résultat beaucoup plus probant. En revanche, si le rendu gagne en qualité et si le nombre de boules influe sur l'efficacité de l'illusion d'une surface lisse, cette amélioration se passe au détriment, comme d'habitude, du temps de *parsing* et de calcul de l'image. Sur la deuxième illustration de la Figure 10, il est possible d'observer à l'oeil nu des artéfacts sur la surface du ressort, témoignant de la nécessité d'augmenter encore un peu le nombre de boules par tour, mais vous ne considérerez certainement cette option que pour les plans rapprochés.

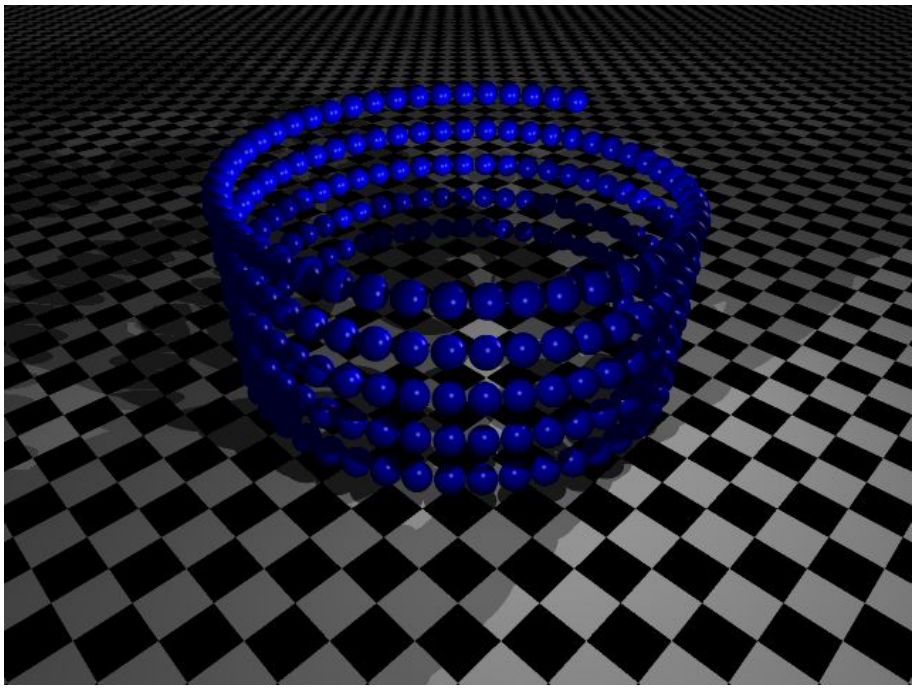


*Figure 10: Notre première tentative de modélisation d'un filetage... pas si mal!*

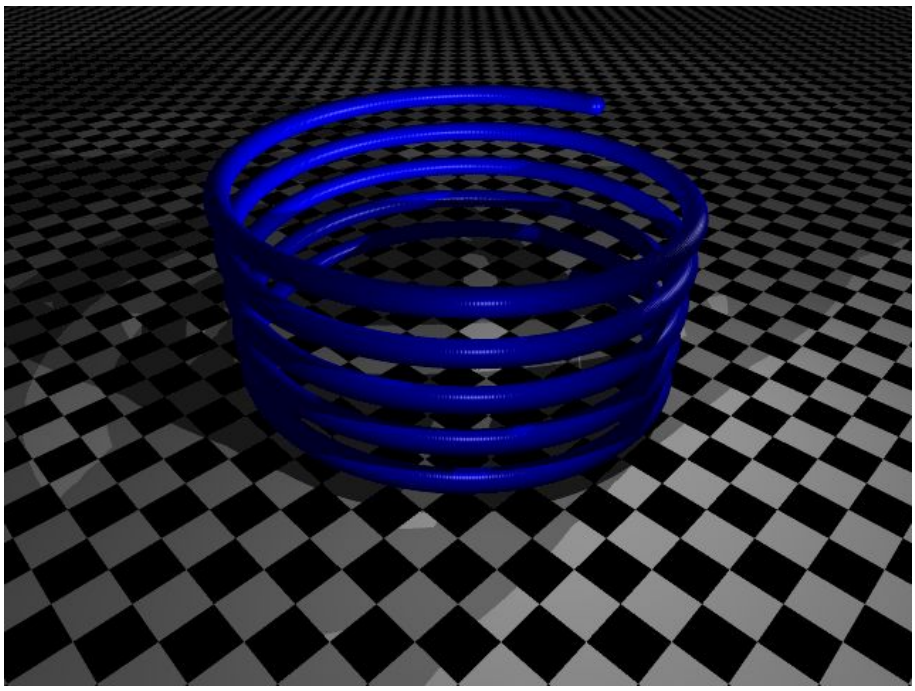


*Figure 10bis: Et voici en spécifiant 500 boules par tour!*

# 7. Exercice final: la modélisation d'une coquille d'escargot

Sur le même principe de modélisation que le ressort, nous allons imposer à chaque boule une translation dans le plan horizontal <X,Z> plutôt que dans la direction de l'axe Y. De plus, nous allons faire usage d'un rayon variable, afin de partir de l'origine et de décrire une spirale de rayon à chaque incrément plus important. Enfin, chaque boule se verra attribuer une variation de taille.

Tout d'abord, nous déciderons du rayon final de notre coquille d'escargot et du nombre de tours que celle-ci fera d'une extrémité à l'autre. Nous déciderons également du facteur d'échelle entre la première boule et la dernière, en spécifiant une échelle de départ, et le nombre de boules qui constitueront un tour. Souvent, ces deux paramètres seront ajustés après un ou plusieurs rendus, car il est parfois difficile de deviner de prime abord quels seront les paramètres les plus adéquats.

Les calculs sont relativement simples: on définit le nombre de boules total comme étant le nombre de boules par tour, multiplié par le nombre de tours. Chaque incrément, qu'il soit d'échelle ou de translation, sera donc un multiple de `N/(Ntours*NbParTour)`. En particulier, chaque boule sera déplacée sur l'axe X par rapport à l'origine de `N/(Ntours*NbParTour)*Rayon`: cela permet d'avoir un rayon nul à la première itération, et d'obtenir le rayon final lors de la dernière. De même, comme pour l'exemple du ressort précédent, la variation de l'angle auquel sera positionnée la boule est définie par `N*360*Ntours/Ntotal`, nous n'y reviendrons pas. La dernière subtilité réside dans la détermination de l'échelle de la boule positionnée: il s'agit tout simplement de notre désormais habituelle fraction `N/(Ntours*NbParTour)` de l'échelle finale définie comme étant égale à l'échelle de départ divisé par le facteur d'échelle que nous avons choisi au début de l'exercice.

```
#declare Rayon = 3;
#declare NbParTour = 50;
#declare Ntours = 3;
#declare Ntotal = NbParTour * Ntours;
#declare FacteurEchelle = .35;
#declare EchelleDepart = 1;
#declare EchelleFinale = EchelleDepart/FacteurEchelle;
#declare Echelle = EchelleDepart;
#declare N = 0;
#while (N < Ntotal)
object{Sphere_bleue
translate <N*Rayon/(Ntours*NbParTour),0,0>
rotate <0,360*Ntours*N/Ntotal,0>
scale <Echelle,Echelle,Echelle>
}
```

```
#declare N = N + 1;
#declare Echelle = N*EchelleFinale/(Ntours*NbParTour);
#end
```

Un premier rendu, observable sur la Figure 11 ci-après, nous donne un résultat presque satisfaisant, puisqu'il apparaît comme évident que le rayon de notre sphère bleue est insuffisant par rapport à l'échelle de nos paramètres. Plutôt que de remettre ceux-ci en question, nous augmenterons simplement le rayon de la sphère de 0.35 (valeur utilisée pour cet exemple) à 1.0 pour obtenir un résultat beaucoup plus satisfaisant. Ensuite, il ne vous reste plus qu'à recourir à l'aide de la CSG pour retailler l'extrémité prétendument ouverte de la coquille, et bien sûr pour l'évider, et l'exercice de modélisation touche à sa fin. Une texture granitique et hop! Le tour est définitivement joué.

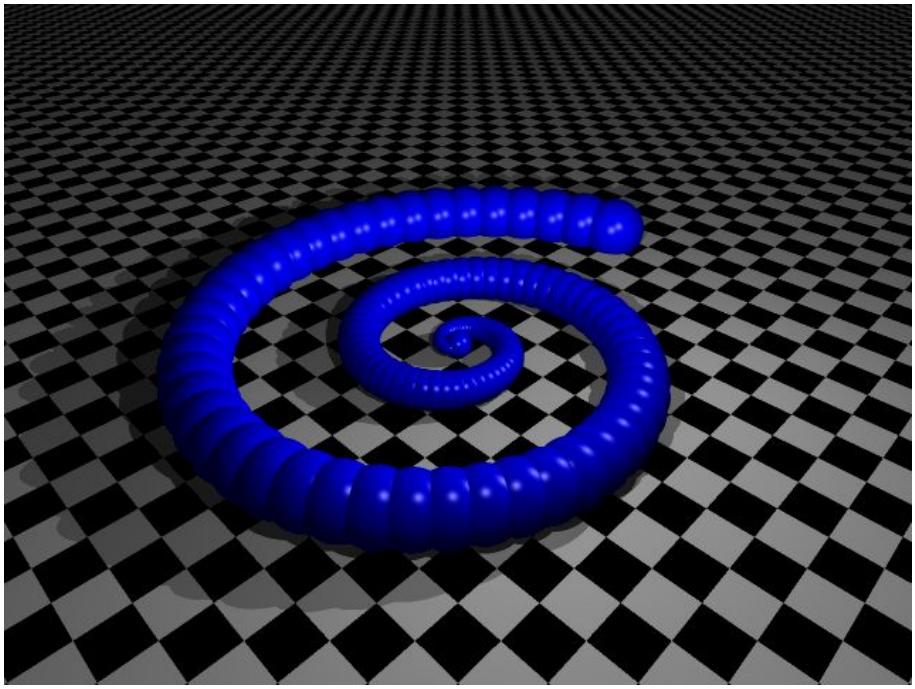


*Figure 11: Avec un rayon de sphère égal à 0.35...*

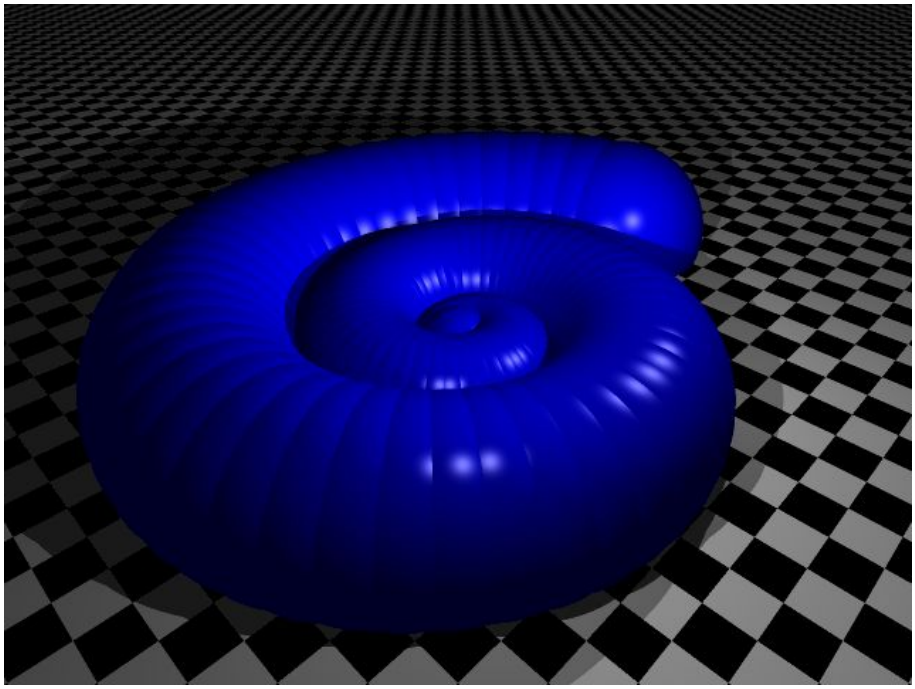
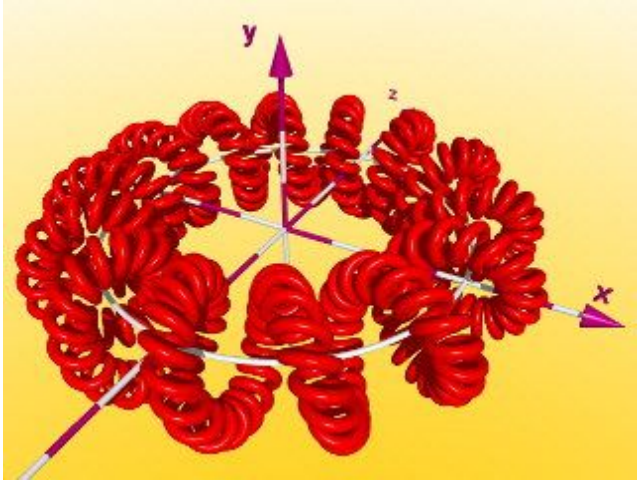


*Figure 11bis: ... et égal à 1.0: à la texture près, on jurerai un vrai! ;o)*

# 8. Conclusions

Nous voici arrivés au terme de cette découverte de la boucle `#while`. En passant par l'étude de plusieurs cas pratiques, il a été assez facile d'assimiler le fonctionnement et les possibilités offertes par cette fonction, qui rappelle que POV-ray est un outil fortement mathématique et « scriptable », et par conséquent versatile et très puissant entre les mains d'un utilisateur averti. Nous ne prétendrons pas que cela soit encore notre cas, mais nous venons de franchir une étape intéressante dans l'apprentissage de POV-ray. Si vous souhaitez approfondir l'étude des boucles `#while` et vous essayer à d'autres cas pratiques, nous ne saurions que trop vous conseiller la lecture de ce formidable didacticiel de Friedrich A. Lohmüller: http://www.f-lohmueller.de/pov_tut/loop/povlup1e.htm. Plusieurs autres formes concevables en termes de boucles `#while` y sont présentées, pouvant conduire à des expérimentations particulièrement rafraîchissantes, idéal pour le temps caniculaire qu'il fait peut-être au moment où vous lisez ces pages. Bonnes vacances pour les plus chanceux d'entre vous, et à la rentrée!

# 9. Liens




L'excellent didacticiel de Friedrich A. Lohmüller: About While-Loops with the POV-Ray Raytracer:
**http://www.f-lohmueller.de/pov_tut/loop/povlup1e.htm**

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

La homepage de povray (version courante: v3.5): **http://www.povray.org**

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