Introduction à POV-Ray

par khayyam90

Date de publication : 27/10/2005

Dernière mise à jour : 27/10/2005

Cet article a pour but de présenter le langage POV-Ray.

I - Introduction

I-1 - POV-Ray, c'est quoi ?

POV-Ray (Persistence of Vision Raytracer) est un un outil permettant d'interpréter un code source écrit dans le langage du même nom. Ce langage permet de décrire une scène tridimensionnelle et l'interprétation de ce code va fournir l'image correspondante. POV-Ray est donc à voir comme un compilateur qui ne fournira pas un binaire, mais une image. Les images sont générées via un algorithme de raytracing.

I-1-A - Le raytracing, c'est quoi ?

C'est une méthode de rendu d'image dans laquelle la couleur de chaque pixel est le résultat du trajet d'un rayon lumineux. Les rayons partent de la caméra et subissent tous les effets créés par l'environnement décrit (réflection, réfraction, diffraction). Cette technique permet d'obtenir des images très réalistes.

POV-Ray exploite cette technique jusque dans les objets de la scène : en effet, ils sont gérés comme des objets mathématiques parfaits et non comme des ensembles de polygones. Il devient donc possible de rendre une image dans une résolution trés élevée sans avoir d'effet de facettes.

I-2 - Télécharger POV-Ray

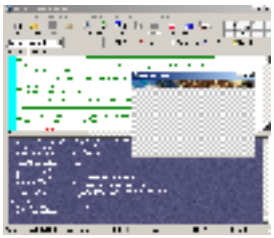
POV-Ray est disponible pour les plateformes Windows, Linux et MacOS X. Les sources ainsi que les binaires sont disponibles sur le site officiel www.povray.org, section download.

La version Windows comporte un éditeur de code, à la manière d'un IDE (integrated development environment). De tels environnements de développement sont aussi disponibles sous Linux, mais ce sont des versions non officielles (QTPovEditor, Pyvon ...).

I-3 - Utiliser POV-Ray

I-3-A - Version Windows

Cliquez sur la miniature pour agrandir l'image



L'interface graphique sous Windows est composée de :

- La barre de menus
- La barre d'outils, parmis lesquels le bouton 'Run' permettant de démarrer le rendu d'une image

- Une liste déroulante permettant de choisir la résolution de l'image qui sera générée
- Une zone de saisie pour définir des options
- La zone d'édition du code
- Le log d'interprétation, affichable/masquable par CTRL-M

I-3-B - Version Linux

Les différentes interfaces graphiques développées pour POV-Ray sous Linux sont proches de la version Windows, je vais donc expliquer ici l'utilisation de POV-Ray en ligne de commande.

Plusieurs syntaxes permettent de lancer la rendu d'une image. La première lance le rendu d'un code source avec les options par défaut. Elle va générer un fichier bmp du même nom que le fichier source.

```
$ povray source.pov
```

Vous pouvez ensuite enrichir cette syntaxe pour rajouter des options comme par exemple la résolution souhaitée. Mais la méthode la plus souvent utilisée s'apparente à l'utilisation des fichiers de compilation Makefile : on regroupe toutes les options souhaitées dans un fichier .ini et on lance le rendu via la commande :

```
$ povray source.ini
```

Je détaillerais plus tard la construction d'un fichier ini.

I-3-C - Version Mac

La version Mac comporte également une interface graphique à la manière d'un IDE. Son utilisation est très proche de la version Windows.

II - Premiers pas avec POV-Ray

II-1 - Les conventions de POV-Ray

POV-Ray est sensible à la casse, ce qui signifie que camera n'est pas la même chose que Camera. camera est un mot réservé du langage tandis que Camera ne l'est pas.

Il est possible (et heureusement) d'insérer des commentaires dans un code POV-Ray. Ils utilisent la syntaxe du C++ :

```
/*
ceci
est
un
commentaire
sur
plusieurs
lignes
*/
ceci n'est plus un commentaire
// mais ceci est encore un commentaire
```

POV-Ray utilise un repère cartésien x,y,z direct. Toutes les coordonnées sont à exprimer dans ce système. Par défaut, la verticale de la caméra est suivant l'axe y, on a donc communément l'habitude de considérer le plan x,z comme horizontal.

POV-Ray n'est pas un langage typé, ce qui signifie que l'on déclare de la même manière un entier, un flottant, un vecteur ou encore un objet. Un vecteur doit être encadré de chevrons <>. Un objet doit être décrit entre accolades {}.

Toutes les instructions du langage commencent par un #, ce qui n'est pas sans rappeler les directives préprocesseur du langage C. En effet, POV-Ray analyse d'abord une première fois le code source pour générer un code ne contenant plus aucune instruction, uniquement des descriptions d'objets, pour ensuite lancer le rendu de l'image.

Il est parfois possible de spécifier un scalaire quand POV-Ray attend un vecteur, il effectue alors la transformation suivante pour obtenir un vecteur.

$$\mathbb{R} \rightarrow \mathbb{R}^3$$
$$x \rightarrow \langle x, x, x \rangle$$

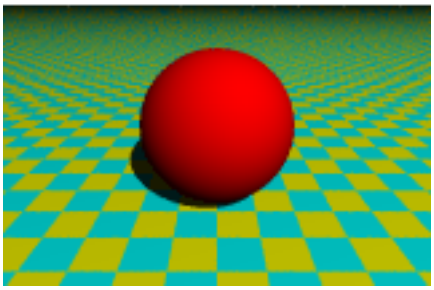
Mais ceci n'est réellement utile que quand vous souhaitez alléger du code, car la compréhension du code s'en retrouve compliquée.

II-2 - Hello World

Un Hello World en raytracing consiste en une sphere posée sur un échiquier. Dans le code source il va nous falloir

une sphère, un échiquier, une caméra, une source de lumière et un arrière-plan. C'est parti.

hello.pov
```
camera{
    location <0,5,-8>
    look_at <0,2,0>
}

light_source{
     <20,50,-50>, color rgb<1,1,1>
}

background{
     color rgb<1,1,1>
}

plane{
  y,0
  pigment{
       checker color rgb <1,1,0>
               color rgb <0,1,1>
  }
}

sphere{
  <0,2,0>,2
  pigment{
       color rgb <1,0,0>
  }
}
```



Le code source parle presque de lui-même. Il y a :

- Une caméra positionnée au point <0,5,-8> et qui regarde le point <0,2,0>.
- une source de lumière [light_source] positionnée au point <20,50,-50> de couleur <1,1,1> soit 100% de rouge, 100 % de vert et 100% de bleu, donc du blanc.
- un arrière-plan [background] coloré en blanc (<1,1,1>).
- un plan de normale y, situé a 0 unité de l'origine du repère (c'est donc le plan d'équation y=0). Ce plan est coloré selon un motif d'échiquier [checker] à deux couleurs : jaune (<1,1,0>) et cyan (<0,1,1>).
- une sphère centrée en <0,2,0>, de rayon 2 et colorée en rouge (<1,0,0>).

Essayez de calculer cette image dans differentes résolutions, vous ne verrez jamais d'effet de pixellisation, le contour de la sphere sera toujours arrondi et jamais en escalier.

II-3 - Les variables

Dans tout langage de programmation qui se respecte, il a des variables. En POV-Ray, les variables n'ont pas besoin d'être déclarées. La seule règle est de ne pas faire appel à une variable non initialisée dans des calculs. La syntaxe d'affectation d'une valeur à une variable est la suivante :

```
#declare i=0;
```

POV-Ray n'est pas typé, rien ne vous empêche donc de faire

```
#declare i=0;
#declare i=i+1;
#declare i="developpez.com";
#declare i=<1,2,3>;
```

Pensez juste à ne pas mélanger des chaînes de caractères, des scalaires et des vecteurs dans des opérations non prévues pour ça.

Toutes les variables sont globales, sauf mention explicite (voir section sur les macros), elles seront donc connues dans tout le script y compris dans les fichiers annexe.

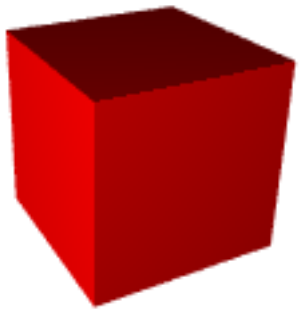
III - Les objets de base

Chaque objet doit être décrit entre {}. Il suivra toujours le modèle suivant (l'ordre des attributs n'a pas d'importance):

```
type_de_l_objet {
   [attributs de l'objet pour définir sa forme]
   [attributs de l'objet pour définir son aspect]
   [attributs de l'objet pour définir les transformations géométriques à lui appliquer]
}
```

Chaque objet demande des arguments différents. Voici la description des attributs de forme des objets les plus courants dans POV-Ray. Nous verrons par la suite les attributs pour définir l'aspect et les transformations des objets.

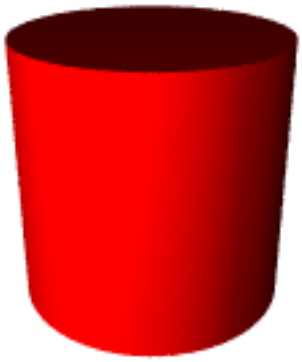
III-A - box



Cet objet correspond à un parallélépipède rectangle dont les arrêtes sont parallèles aux axes du repère. Il est défini par 2 points opposés suivant la grande diagonale.

```
box{<0,0,0>,<1,1,1>}
```

III-B - cylinder



Cet objet correspond à un cylindre. Il est defini par les centres des 2 disques le délimitant ainsi que par son rayon.

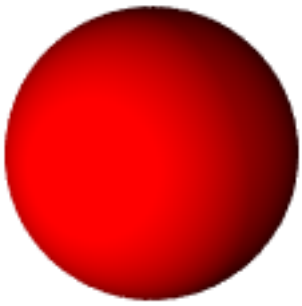
```
cylinder{<0,0,0>,<1,1,1>,.5}
```

III-C - torus

Cet objet correspond à un tore. Il est défini par ses rayons majeur et mineur. Le tore généré est centré en O, autour de l'axe y.

```
torus{10,1}
```

III-D - sphere



Cet objet correspond à une sphère. Elle est définie par son centre et son rayon.

```
sphere{<0,0,0>,2}
```

III-E - plane

Cet objet correspond à un plan infini. Il est défini par un de ses vecteurs normaux et la distance qui le sépare de l'origine du repère.

```
plane{y,0}
```

Notez que y est un mot réservé du langage et qu'il est équivalent à <0,1,0>, de même que x est équivalent à <1,0,0> et z à <0,0,1>

III-F - triangle

Cet objet correspond à un triangle. Il est défini par ses 3 sommets.

```
triangle{<0,0,0>,<1,5,1>,<5,1,4>}
```

III-G - smooth_triangle

Cet objet correspond à un triangle à l'apparence 'bombé'. Il est géometriquement plat, mais sa normale est modifiée pour lui donner une apparence bombée. Il est défini par 3 couples de points / vecteurs normaux aux points. Les triangles et les smooth_triangles sont largement utilisés dans les mesh (voir plus loin).

```
smooth_triangle
{
    // premier point et vecteur normal au premier point
    <-2,-1,0>,<-3,0,-1>,
    // second point et vecteur normal au second point
    <5,3,2>,  <0,3,-1>,
    // troisième point et vecteur normal au troisième point
    <3,-2,-3>,<-1,0,-1>
}
```

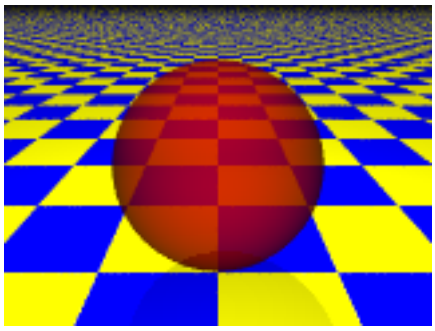
IV - L'aspect des objets

Il n'est pas nécessaire de renseigner tous les champs de l'aspect des objets.

IV-A - Les pigments

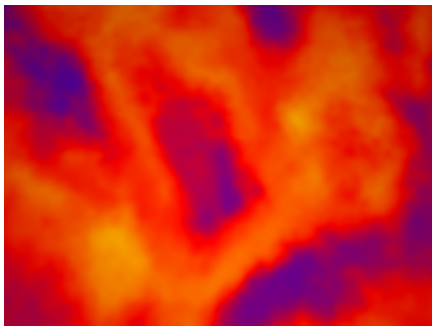
Le pigment correspond à la couleur proprement dite de l'objet. Il peut aussi modifier la transparence de l'objet avec l'option transmit. Exemple :

```
sphere{<0,1,0>,1
    pigment{
                color rgb<1,0,0>
                transmit .5
        }
}
```



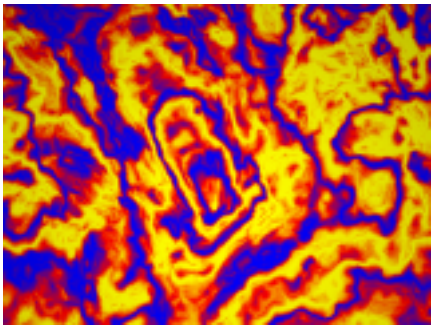
Il est possible de déclarer une couleur ou bien un motif alliant plusieurs couleurs. Les motifs alliant plusieurs couleurs utilisent généralement une carte de couleurs (color_map). Une carte de couleurs donne un éventail de couleurs au motif. Vous pouvez aussi spécifier un facteur de turbulence dans le motif. Voici quelques motifs utilisant les color_map

IV-A-1 - bozo



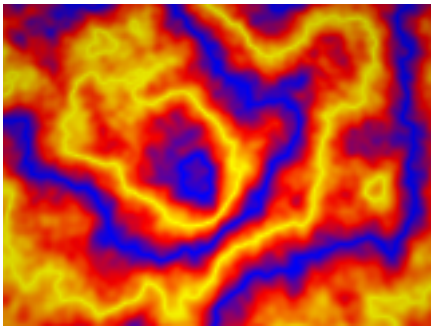
```
pigment {
     bozo
     turbulence 2
     color_map {
       [0  color rgb<0,0,1>]
       [.5 color rgb<1,0,0>]
       [1  color rgb<1,1,0>]
     }
}
```

IV-A-2 - agate



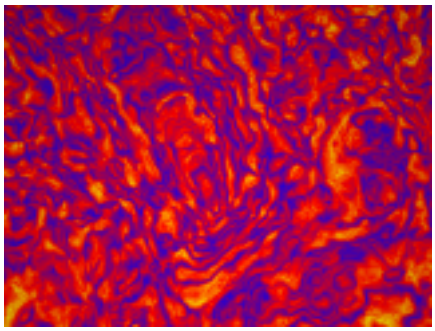
```
pigment {
     agate
     turbulence 2
     color_map {
       [0  color rgb<0,0,1>]
       [.5 color rgb<1,0,0>]
       [1  color rgb<1,1,0>]
     }
}
```

IV-A-3 - marble



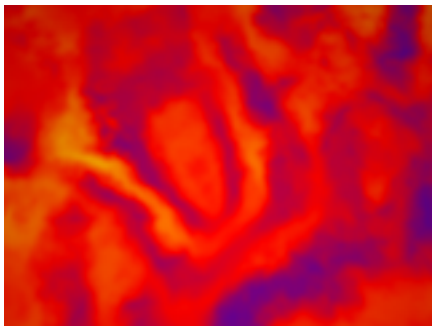
```
pigment {
     marble
     turbulence 2
     color_map {
       [0  color rgb<0,0,1>]
       [.5 color rgb<1,0,0>]
       [1  color rgb<1,1,0>]
     }
}
```

IV-A-4 - granite



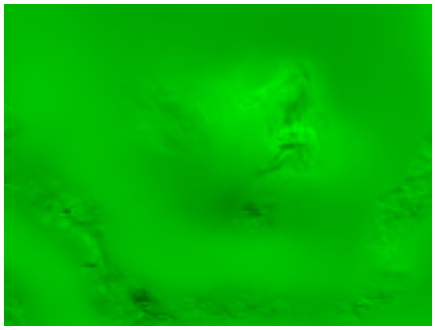
```
pigment {
     granite
     turbulence 2
     color_map {
       [0  color rgb<0,0,1>]
       [.5 color rgb<1,0,0>]
       [1  color rgb<1,1,0>]
     }
}
```

IV-A-5 - ripples



```
pigment {
     ripples
     turbulence 2
     color_map {
       [0  color rgb<0,0,1>]
       [.5 color rgb<1,0,0>]
       [1  color rgb<1,1,0>]
     }
}
```

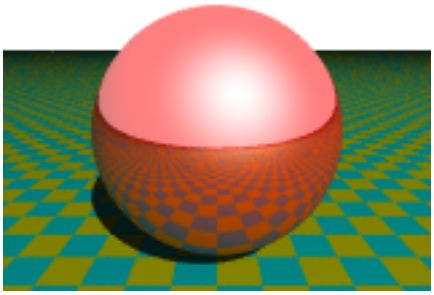
IV-B - Les normales



La modification de la normale permet de simuler une surface d'un objet différente de sa surface géométrique, de manière à ce que les rayons lumineux se reflettent selon un angle différent. Cela permet de faire passer un objet lisse pour un objet rugueux ou érodé. L'image ci-dessus est réalisée à partir d'un objet parfaitement lisse, muni d'un modificateur de normale. A la manière des pigments, il est possible de spécifier des motifs pour les normales, et cette fois-ci, en utilisant une carte de normale (normal_map). L'utilisation des normales est très intéressante pour un objet qu'on souhaite calculer rapidement. C'est un bon moyen d'économiser des ressources. Attention cependant, si la caméra est trop près de l'objet, on pourra voir que celui-ci est parfaitement lisse. Exemple de normale :

```
    normal {
         ripples
         turbulence .2
         normal_map {
           [0  bozo]
           [.5 ripples]
           [1  agate]
         }
    }
```

IV-C - Les finishs

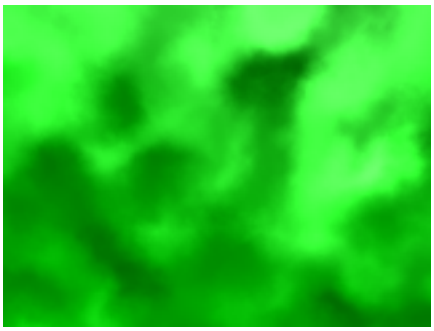


L'attribut finish permet de définir la manière dont l'objet interagira avec la lumière : réflection, brillance, tâches spéculaires ... Exemple :

```
    sphere{<0,0,0>,2 pigment{Red transmit .5}
           finish{
               reflection .5   // défini le pourcentage de réflection
               brilliance .8   // défini le niveau de brillance de l'objet
               phong .5        // défini l'intensité de la tâche spéculaire
               phong_size 3    // défini la taille de la tâche spéculaire
       }
    }
```

C'est sur le finish qu'il faut jouer pour obtenir un effet neuf ou ancien, verni ou mat, en plastique, en métal ...

IV-D - Les textures



Une texture est un regroupement d'un pigment, d'un finish et d'une normale. Une texture peut ne regrouper pr exemple qu'un pigment, étant donné qu'il n'est pas obligatoire de renseigner les champs normal ni finish. Exemple d'utilisation :

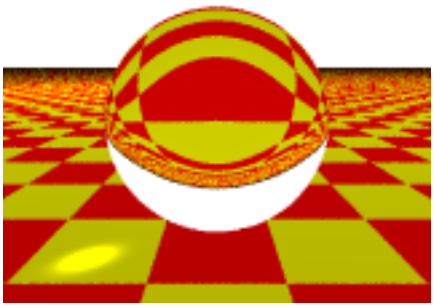
```
plane
{
   z,0
   texture
   {
       pigment{color rgb<0,1,0>}
       normal{
                   // nom du motif de normale
                       bumps
                       turbulence .3
               }
       finish{
                   // caractéristiques des tâches spéculaires
                       phong .3 phong_size 2
               }
   }
}
```

Mais au fait, ça sert à quoi, si c'est juste pour rajouter une paire d'accolades autour du triplet pigment/normal/finish ?

Ca devient très intéressant lors de la réutilisation de textures déja existantes (voir les sections sur les fichiers include standards et sur la manipulation d'objets). Il suffira d'appeler une texture par

```
texture{ nom_de_la_texture }
```

IV-E - Les interiors

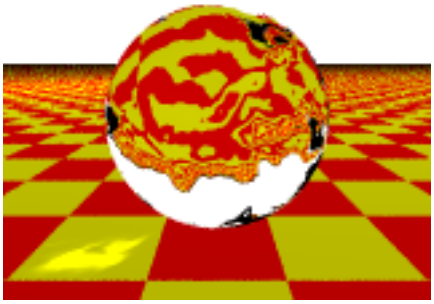


L'attribut interior décrit l'intérieur d'un objet. Pour que cet attribut ait une quelconque incidence sur l'image, l'objet concerné doit être transparent (au moins un peu). L'attribut interior permet de définir des options comme l'indice de réfraction de l'objet ou le faux caustique. Exemple :

```
interior
{
   ior 1.33    // indice de réfraction
   caustics 3  // intensité du faux caustique
}
```

Mais aussi et surtout, l'attribut interior permet de définir des medias. Ce sont des objets qui vont interagir d'une manière particulière avec la lumière : ils pourront émettre / absorber ou 'attraper' de la lumière. Les possibilites des médias étant immenses, il faudrait consacrer plusieurs articles rien que sur leur utilisation. Et le but de cet article n'est pas de présenter toutes les fonctionalites de POV-Ray, mais juste une présentation des plus courantes.

IV-F - Les materials



Un material est un regroupement d'une texture et d'un interior. Exemple d'utilisation :

```
sphere
{
 // centre et rayon
 <0,.5,0>,.5
 material
 {
     texture
     {
         pigment
         {
             color <1,0,0>
             transmit 1          // objet entièrement transparent
         }

         // modificateur de normale
         normal
         {
                         ripples      // nom du motif
                         scale 2      // agrandissement du motif
                         turbulence 3 // ajout de turbulence dans le motif
         }
     }
     interior
     {
        ior 1.33   // indice de réfraction
        caustics 3 // intensité du faux caustique
     }
  }
}
```

Et pour les mêmes raisons que les textures, material n'est pas qu'une paire d'accolades en plus. Cela permet de réutiliser des materials déja existants.

V - Les transformations

V-A - Les transformations géométriques

V-A-1 - La translation

Elle permet de déplacer un objet d'un vecteur donné. Exemple

```
sphere
{
    // centre et rayon
    <4,0,0>,1
    // translation du vecteur (1,2,3)
    translate <1,2,3>
}
```

V-A-2 - La rotation

Elle permet de faire tourner un objet autour d'un axe, d'un angle donné en degrés. La rotation s'effectue autour de l'axe passant par O et dirigé selon le vecteur passé en argument. Exemple :

```
sphere
{
    // centre et rayon
        <4,0,0>,1
        // rotation de 50° autour de l'axe y
        rotate y*50
}
```

y est un mot réservé du langage, il correspond au vecteur <0,1,0>

L'axe de rotation passe toujours par O, ainsi, un objet peut se retrouver déplacé après une rotation.

V-A-3 - L'homothétie

Elle permet d'agrandir / rapetisser un objet selon un facteur donné. Le centre d'homothétie est le centre du repère. Le facteur donné est un vecteur décrivant les homothéties à réaliser suivant x,y et z. Exemple :

```
sphere
{
    // centre et rayon
        <4,0,0>,1
    // agrandissement une fois suivant x, 2 fois suivant y et 3 fois suivant z
        scale<1,2,3>
}
```

Si l'une des composantes du vecteur est nulle, POV-Ray la remplace par 1 et vous averti via un Warning :

```
Parse Warning : Illegal Value: Scale by 0.0. Changed to 1.0
```

Le centre d'homothétie est O, ainsi, un objet peut se retrouver déplacé après une homothétie.

V-B - Les opérations booléennes (CSG)

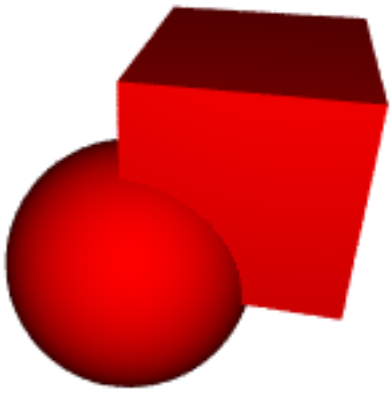
Ces opérations s'effectuent sur des objets, exemple :

```
[opération booléenne]{
objet1
objet2
objet3
[attributs pour définir un aspect]
[attributs pour définir les transformations géométriques à appliquer]
}
```

Les opérations booléennes nécessitent au moins 2 objets. Une opération booléenne sur un seul objet vous donnera comme résultat ce même objet, vous aurez juste un Warning lors de l'analyse du code :

```
Parse Warning : should have at least two objects in csg
```

V-B-1 - L'union



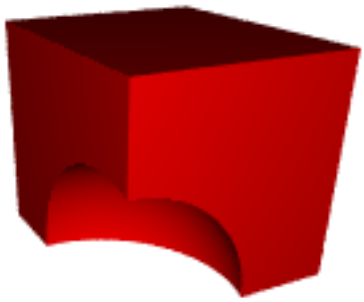
Elle permet de regrouper plusieurs objets dans le but de leur appliquer des modifications communes. Cette opération correspond au OU logique.

V-B-2 - L'intersection

Elle permet créer l'intersection de plusieurs objets. Cette opération correspond au ET logique. Exemple :

```
intersection{
   box{<0,0,-.2>,<1,1,1>}
   sphere{<0,0,0>,.6}
   pigment{Red}
}
```

V-B-3 - La différence



Elle permet créer une différence entre plusieurs objets. Cette opération créé un nouvel objet composé tous les points du premier objet qui ne sont pas dans les autres objets. Exemple :

```
difference{
   box{<0,0,-.2>,<1,1,1>}
   sphere{<0,0,0>,.6}
   pigment{Red}
}
```

VI - Les fichiers include

Ces fichiers *.inc permettent de regrouper des ensembles de déclarations de pigments / textures / objets / ... dans un fichier annexe et d'y faire appel dans le script principal. L'appel d'un fichier annexe se fait par l'instruction

```
#include "machin.inc"
```

VI-A - Les fichiers standards

La version officielle de POV-Ray comprend plusieurs fichiers *.inc. Ces fichiers sont dans le répertoire include de POV-Ray. Ils contiennent les définitions de certains attributs fréquemment utilisés.

- colors.inc contient, entre autres, les définitions des symboles 'Red', 'Blue' ... autorisant l'écriture de

```
sphere
{
    // centre et rayon
       <0,0,0>,1
    pigment{Red}
}
```

au lieu de

```
sphere
{
    <0,0,0>,1
    pigment{color rgb<1,0,0>}
}
```

- textures.inc contient des définitions de pigments et textures en tout genre, allant de la roche au ciel étoilé, en passant par le bois ou le verre. Exemple :

```
#include "textures.inc"
sphere{<0,0,0>,1 texture{Yellow_Pine }}
```

La texture Yellow_pine

- glass.inc contient des definitions de pigments / finish / interior relatifs au verre
- woods.inc contient des definitions de pigments et textures relatifs au bois
- ...

VI-B - Faites vos propres fichiers include

Un fichier include n'est qu'un fichier texte contenant du code POV-Ray. Vous pouvez lui donner l'extension que vous voulez. Par habitude, on lui donne l'extension .inc. Si vous souhaitez faire appel à un fichier annexe dans votre script principal, il doit se trouver dans le répertoire include de POV-Ray ou bien dans le répertoire de votre script (c'est d'ailleurs préférable pour des questions de simplicité). On a l'habitude de mettre toutes les déclarations d'objets / textures / macros dans des fichiers annexes, de manière à n'avoir plus qu'un petit script principal dont l'architecture est aisée à comprendre.

VII - Utilisation (un peu) plus avancée

VII-A - #if #else #end

Le branchement conditionnel permet d'effectuer un traitement si une condition est verifiée. Exemple :

```
#if (truc = 1)
        height_field{"machin.bmp" pigment{Red}}
#else
        sphere{<0,0,0>,1 pigment{Blue}}
#end
```

dessinera un height_field ou une sphère selon la valeur de la variable truc.

Il n'est pas obligatoire de mettre un traitement alternatif #else.

VII-B - #switch #case #range #break #else #end

L'instruction #switch permet de réaliser des branchements selon la valeur d'une variable. C'est un #if particulier.

```
#switch(var)
   /* si var vaut 0 */
      #case (0)     sphere{<0,2,0>,2 pigment{ Red   }}     #break;
   /* si var vaut 1 */
      #case (1)     sphere{<0,2,0>,2 pigment{ Blue  }}     #break;
   /* si var est entre 2 et 10  */
      #range(2,10)  sphere{<0,2,0>,2 pigment{ Green }}     #break;
   /* tous les autres cas  */
      #else         sphere{<0,2,0>,2 pigment{ Brown }}     #break;
#end
```

Ce code dessinera une sphere dont la couleur depend de la valeur de var. Si var vaut 0, elle sera rouge, si var vaut 1, elle sera bleue, si var est entre 2 et 10, elle sera verte et elle sera brune dans tous les autres cas.

Pensez bien à ne pas oublier les instructions #break en fin de chaque cas. Sans quoi toutes les conditions décrites apres la condition résolue seront interprêtées. Et vous obtiendrez plusieurs sphères de couleurs différentes, les unes dans les autres.

VII-C - #while #end

Pour boucler tant qu'une condition est verifiée

```
#declare i=0;
#while (i < 10)
        sphere{<3*i,0,0>,1 pigment{Orange}}
    #declare i=i+1;
#end
```

dessinera 10 spheres alignées

VII-D - #macro #end

Voici l'instruction que tout le monde attend : la fonction. Elle permet d'effectuer des traitements ou des affichages

selon des arguments. Voici un exemple d'utilisation de macro dessinant des sphères entre deux abscisses données :

```
#macro dessiner_spheres(absc1, abcs2, pas, rayon)
   #if (absc1 < abcs2)
      #local i=absc1:
      #while (i < absc2)
          sphere{<i,0,0>,rayon pigment{Red}}
          #local i=i+pas;
      #end
   #end
#end
```

Et maintenant voici l'appel de la macro :

```
dessiner_spheres(0,50,1,.5)
```

On notera l'utilisation de l'instruction #local permettant de déclarer une variable locale à la macro. Cette variable ne sera pas connue en dehors de la macro.

VII-E - La manipulation d'objets

Supposez que vous ayiez une texture que vous utilisez sans arrêt, ce serait dommage de l'écrire à chaque fois, d'autant plus que si vous changez ensuite la texture, il vous faudra la changer partout où vous vous en servez. POV-Ray n'est pas typé, on peut tout à fait assigner une texture à une variable, ou un objet, un pigment, un finish ... Exemple d'utilisation pour factoriser un code :

```
#declare ma_texture = texture{
pigment{Blue}
normal {
        ripples
        turbulence 2
        normal_map {
                [0  bozo]
                 [.5 ripples]
                 [1  agate]
   }
}
// [...]
sphere{<0,0,0>,1 texture{ma_texture}}
torus{10,.1 texture{ma_texture}}
```

De même si vous avez un objet utilisé plusieurs fois avec des textures différentes :

```
#declare mon_objet = sphere{<0,0,0>,1}
// [...]
object{mon_objet texture{ma_texture1}}
object{mon_objet translate <10,0,0> texture{ma_texture2}}
```

VII-F - La manipulation de fichiers

POV-Ray permet de lire et écrire des fichiers texte. Il peut être intéressant de sauvegarder dans un fichier le résultat d'un traitement très long, de manière à le récuperer lors des prochains rendus d'image, sans avoir à le recalculer. Les procédures de sauvegarde et de chargement sont entièrement à la charge du programmeur. Vous n'avez que les outils de base suivants :

- #fopen. Cette instruction permet d'ouvrir un fichier texte. Cette instruction prend 3 arguments : un identifiant pour le fichier, le nom du fichier et le mode d'ouverture du fichier. Exemple :

```
#fopen mon_fichier "machin.txt" read
```

Le symbole mon_fichier n'a pas besoin d'avoir été déclaré précédemment.

Les différents modes d'ouverture de fichiers sont : read (ouverture en lecture), write (ouverture en écriture, si un fichier du même nom existe déjà, il est écrasé) et append (ouverture en écriture, si un fichier du même nom existe déjà, le pointeur interne est positionné à la fin du fichier).

- #read. Cette instruction permet de lire dans un fichier une liste de valeurs séparées par des virgules.

```
#read mon_fichier var1, var2, var3
```

Cette instruction est à nombre d'arguments variables. Il est donc possible de lire un grand nonbre de variables en une seule fois.

- #write. Cette instruction permet d'écrire dans un fichier. Le fichier doit auparavant avoir été ouvert en écriture. Cette instruction est aussi à nombre d'arguments variables. Son premier argument est l'identifiant du fichier. L'instruction reçoit ensuite une suite de variables ou de chaînes de caractères.

```
#write mon_fichier var1, ",", var2, ",", var3, "\n"
```

Toutes les variables sont écrites les unes à la suite des autres dans le fichier. Pensez donc à les séparer par des virgules ou des retour chariots (\n), sans quoi vous ne réussirez pas à relire vos données, qui seront considérées comme étant une seule grande chaîne de caractères.

- #fclose. Cette instruction ferme un fichier.

```
#fclose mon_fichier
```

VII-G - Une fonction très utile : trace

La fonction trace peut rendre bien des services, elle permet d'obtenir les coordonnées d'un point sur un objet, quel que soit cet objet, et aussi compliqué soit-il. La fonction trace va calculer le point d'intersection entre un objet et une droite (définie par un point et un vecteur). La syntaxe de la fonction trace est la suivante :

```
trace(nom d'un objet,
     point de depart de la droite,
     vecteur directeur de la droite)
```

Le nom de l'objet doit être une variable contenant un objet. Exemple :

```
#declare obj = sphere{<0,0,0>,1}
#declare intersect = trace(obj,<0,10,0>,<0,-1,0>);
// trace renvoie <0,1,0>
```

La fonction trace admet une seconde syntaxe : elle prend un quatrième argument : une variable dans laquelle elle renverra le vecteur normal à l'objet au point d'intersection. Exemple :

```
#declare result_norm = <0,0,0>;
#declare obj = sphere{<0,0,0>,1}
#declare intersect = trace(obj,<0,10,0>,<0,-1,0>, result_norm);
```

VII-H - Les fichiers ini

Le fichier ini est à un script POV-Ray ce qu'un fichier Makefile est à un projet C/C++, il regroupe toutes les options de rendu. Règles à respecter pour écrire un fichier ini :

une ligne commençant par un point-virgule est un commentaire

il faut indiquer le nom du fichier source ainsi que le nom de l'image à créer

il faut indiquer les dimensions de l'image à créer

Il y a une foultitude d'autres options que l'on peut rajouter, celles-ci ne sont que les plus courantes.

Exemple de fichier ini :

```
; ceci est un commentaire
; options d'antialing
Antialias=On
Antialias_Threshold=.4
Antialias_Depth=3

; dimensions, w pour width (largeur) et h pour height (hauteur)
+w1024
+h768

; noms des fichiers d'entrée et de sortie
Input_File_Name=developpez.pov
Output_File_Name=developpez.bmp
```

Les fichiers ini sont surtout utiles sous Linux, en ligne de commande. Leur utilisation sous Windows est transparente, l'IDE s'occupe de tout (ou presque).

VII-I - Les animations

Il est possible de calculer des animations. Par défaut, POV-Ray ne calcule qu'une seule image par script, mais on peut lui en faire calculer plusieurs, à la manière d'une boucle. POV-Ray met à votre disposition la variable clock, qui est incrémentée à chaque nouvelle image. La valeur de clock depend du nombre d'images à calculer (depend des indices de la première et de la dernière image) ainsi que des valeurs minimum et maximum de clock. A vous ensuite de prendre en compte la valeur de clock dans votre scène pour la faire évoluer. Toutes ces informations peuvent figurer dans un fichier ini. Exemple

```
Initial_Frame=0
Final_Frame=10
Initial_Clock=0.0
Final_Clock=1.0
```

Sous Windows, ces valeurs sont à rajouter dans le fichier POVRAY.INI modifiable à partir du menu Tools -> Edit Master POVRAY.INI.

VII-J - Le hasard : les fonctions seed et rand

Comment obtenir des valeurs aléatoires en POV-Ray ? Le mot juste serait plutôt des valeurs 'pseudo-aléatoires',

ainsi d'un rendu à l'autre, nous obtiendrons les mêmes valeurs. Pour obtenir des valeurs aléatoires, il faut créer une variable représentant le générateur de nombres pseudo-aléatoires. Le générateur est initialisé avec une valeur entière (la racine du générateur). Par la suite, le générateur fournira des flottants compris entre 0 et 1. Exemple d'utilisation :

```
#declare le_generateur = seed(5); // la racine du générateur est 5
sphere{<6*rand(le_generateur), -3*rand(le_generateur), rand(le_generateur)+2>,1 pigment{Red}}
```

Ce code dessinera une sphère de rayon 1 et dont le centre sera quelque part dans l'objet

```
box{<0,0,2>,<6,-3,3>}
```

Notez qu'il est possible d'utiliser plusieurs générateurs de nombres pseudo-aléatoires, il suffit de déclarer plusieurs variables et de les initialiser avec des racines différentes.

VIII - Des objets plus complexes

VIII-A - function

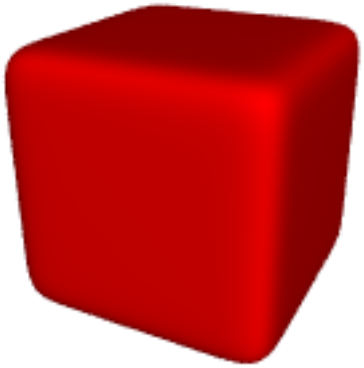
L'objet function ne peut pas être dessiné, c'est une fonction. Par contre il peut être utilisé dans d'autres objets comme les height_field et les isosurfaces. Il y a plusieurs syntaxes pour décrire une fonction, voici la plus simple, pour une fonction de 3 variables :

```
#declare func = function {x+y+z/2}
```

Les 3 variables sont notées x,y,z. Et pour obtenir la valeur de la fonction en un point donné, il faut appeler

```
#declare resultat = func(1,2,3);
```

VIII-B - superellipsoid



Le superellipsoid est une cube érodé. Il est défini par un indice compris en 0 et 1.

0 donne un cube parfait et 1 donne une sphère. Le superellipsoid est compris dans l'objet box{<-1,-1,-1>, <1,1,1>}. Exemple :

```
superellpsoid
{
    // indice du superellipsoid
    .2
    // pigment du superellipsoid
    pigment{Red}
    // translation suivant le vecteur 2*(1,-2,5)
    translate 2*<1,-2,5>
}
```

VIII-C - height_field

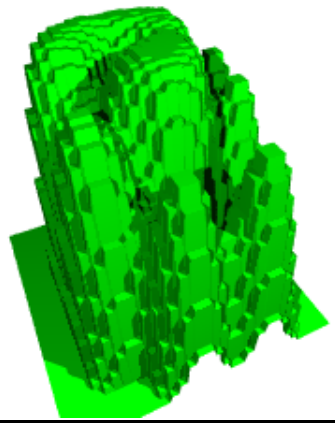
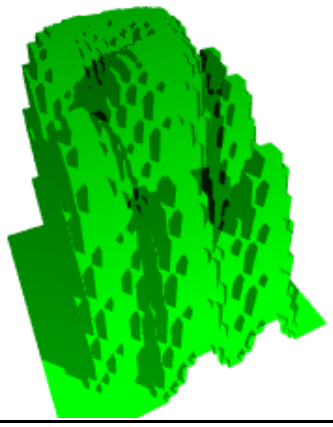
Le height_field est un objet très particulier, il correspond à la representation 3D d'une image dans laquelle les pixels clairs symbolisent des points élevés et les pixels foncés symbolisent des points bas. Un height_field est un ensemble de triangles et la finesse du tracé depend de la résolution de l'image. Le height_field supporte, entre autres, les formats d'image bmp, jpeg, gif, png et tiff. Pour rendre le height_field plus lisse il est possible de le faire utiliser des smooth_triangles au lieu des triangles, en rajoutant l'option smooth dans sa déclaration.

Attention, un height_field avec l'option smooth n'est pas lisse, il en a juste l'apparence.

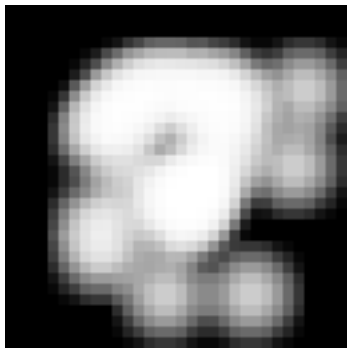
Exemple de height_field:

```
height_field{
                // type de l'image en entree
                png

                // nom de l'image en entree
                "machin.png"

                // option facultative permettant de lisser le height_field
                smooth

                // couleur du height_field
                pigment{Green}

                // agrandissement du height_field
                scale <10,1,10>
}
```

Le height_field est contenu dans l'objet box{<0,0,0>,<1,1,1>}. Il faut donc utiliser les transformations géometriques pour le mettre à la bonne taille et à la bonne place dans votre scène.

sans l'option smooth	avec l'option smooth
	

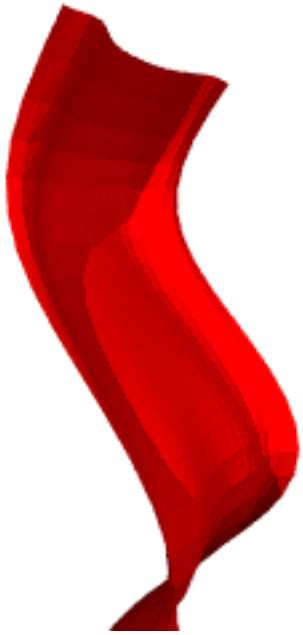
Et voici l'image png qui a donné ces deux height_fields :



Un height_field peut aussi se baser sur une fonction. La syntaxe est différente :

```
height_field{
   function
   nombre de triangles suivant x, nombre de triangles suivant z
   {func(x,0,z)}
}
```

VIII-D - mesh



Un mesh est un ensemble de triangles ou de smooth_triangles. Syntaxe :

```
mesh
{
   // liste de triangles (ou de smooth_triangles)
   triangle{ coordonnées des 3 points du triangle }
   triangle{ coordonnées des 3 points du triangle }
   ....
   pigment{...}
}
```

Les meshs sont généralement générés par d'autres logiciels de modeling ou bien par des boucles générant des triangles. Des logiciels comme Wings3D ou Rhinoceros3D permettent directement d'exporter au format POV-Ray,

ils fournissent alors un fichier .inc qu'il vous suffit d'inclure dans votre script. Vous avez dès lors une variable qui représente votre mesh. Le mesh se dessinera avec la commande

```
object
{
  variable représentant l'objet
  // modificateurs d'aspect
  texture{...}
  // transformations geonetriques
  scale ...
  translate ...
  ...
}
```

Vous pouvez aussi importer des fichiers 3D dans d'autres formats (par exemple 3ds, obj) en utilisant un convertisseur de format du genre de PoseRay qui va créer le fichier POV-Ray correspondant.

Exemple de génération de mesh par une boucle (image ci-dessus):

```
#declare func = function {sin((x+2.5)/2.5)+sin(z/3)+.1*cos(x*x+6*z)}

mesh
{
      /*
        deux boucles imbriquées pour créer une nappe d'une fonction
                il suffit de diminuer les pas pour augmenter la finesse du mesh mais aussi le temps
de calcul
      */
      #declare i=-2.5;
      #declare pasi = .25;
      #declare pasj = 1;
      #declare taillei = 5;
      #declare taillej = 25;
      #while(i<2.5)
           #declare j=0;
           #while (j<25)
                triangle{ <i,func(i,0,j),j>                    ,<i+pasi,func(i+pasi,0,j),j>
,<i,func(i,0,j+pasj),j+pasj>  }
                triangle{ <i+pasi,func(i+pasi,0,j+pasj),j+pasj> ,<i+pasi,func(i+pasi,0,j),j>
,<i,func(i,0,j+pasj),j+pasj>  }
                #declare j=j+pasj;
           #end
           #declare i=i+pasi;
      #end

      // rotation / agrandissement / rotation
      rotate x*90
      scale <2,1,3>
      rotate y*-50

      pigment{Red}
}
```

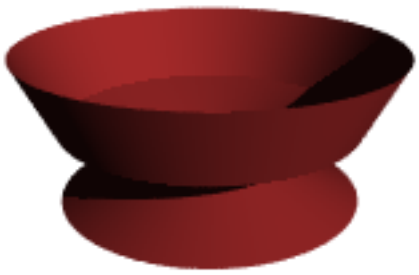
VIII-E - Des objets à base de splines

Pour definir une spline, il faut spécifier le type de la spline (linéaire, quadratique, cubique, naturelle) ainsi que les points par lesquels elle doit passer. Il faut aussi associer à chacun des points un indice permettant par la suite d'obtenir n'importe quel point de la spline. Exemple :

```
#declare ma_spline = spline{
       quadratic_spline
       0  <0,0,0>
       .1 <0,2,1>
       .6 <2,2,1>
       1  <3,1,2>
}
```

```
// [...]
// ma_spline(.5) renverra <1.33,3.33,1.66>
```

VIII-E-1 - lathe



Cet objet correspond à la surface décrite lorsqu'une spline tourne autour de l'axe y. La spline n'a besoin que d'être définie en 2 dimensions. Exemple :

```
lathe{
linear_spline
// nombre de points de la spline
4,
// coordonnées (x,z) des points de la spline
<2,0>,<1,.5>,<2,1>,<2.5,2>
pigment{Red}
}
```

VIII-E-2 - sphere_sweep

Cet objet correspond à l'extrusion d'une sphère à rayon variable le long d'une spline. Il faut définir le type de spline, le nombre de points décrivant la spline, puis les points avec, pour chacun d'eux, le rayon de la sphère en ce point. Exemple :

```
sphere_sweep{
  cubic_spline
  8  // nombre de points de la spline
  <0,0,0> ,.3,
  <2,2,0> ,.4,
  <3,3,0> ,.4,
  <2,5,0> ,.5,
  <4,5,0> ,.3,
  <5,3,0> ,.2,
  <4,2,0> ,.3,
```

```
      <3,1,0> ,.1

      pigment{Red}
   }
```

VIII-F - isosurface

Une isosurface est une surface solution d'une équation. La syntaxe est la suivante :

```
   isosurface {
     function { bla bla bla }
     contained_by { un objet sphere ou box}
   }
```

Cet objet correspond à l'ensemble des points (x,y,z) pour lesquels la fonction passée en argument est nulle. Pour ne pas obtenir un objet infini, celui-ci est restreint aux points contenus dans l'objet passé à l'attribut contained_by (une sphere ou une box). Exemple d'utilisation :

```
   #declare func = function {x*x+y}
   isosurface {
      function { func(x,y,z) }
          contained_by{box{<-1,-2,-1>,<1,1,1>}}
          pigment {Red}
          scale <2,1,1>
   }
```

IX - Liens et remerciements

IX-A - Liens

- Le site officiel de POV-Ray
- Le site du POV-Monde
- La référence POV-Ray francophone
- Le site de Gilles Tran, plusieurs centaines de magnifiques images faites avec POV-Ray
- Le site recensant toutes les textures officielles de POV-Ray.
- Un cours sur les isosurfaces.
- La page de PoseRay.

IX-B - Remerciements

Je tiens à remercier Loulou24 pour la relecture de cet article.