Formation à POV-RAY

Sébastien Bancquart

3 juillet 2006

Introduction

Si vous êtes ici, il est fort probable que vous sachiez déjà ce qu'est POV-RAY. Un petit rappel, néanmoins : POV-RAY est un logiciel de création d'images de synthèses, capable de créer des images de qualité photoréaliste. Contrairement à la plupart des produits sur le marché, POV-RAY n'est pas basé sur une interface graphique, où l'on prévisualise sa scène (soit en fil de fer soit déjà en 3D) et où l'on crée en ajoutant des objets à la souris. Dans POV-RAY, les scènes sont programmées. Pas de panique, en fait c'est très simple : POV-RAY est basé sur un langage descriptif : vous dites au logiciel ■ *Ici, place une sphère de rayon 1, ayant une texture réfléchissante. Là, place une source de lumière bleue. Ici, place la caméra. Et maintenant, débrouille-toi !* ■ et POV-RAY se débrouille pour créer l'image.

Cette méthode, bien que paraissant déroutante, a de nombreux avantages, notamment au niveau précision et souplesse d'utilisation. Des fonctions de programmation permettent de plus de faciliter la création de groupes d'objets, d'objets semi aléatoires ou de structure complexe. L'expérience montre que lorsque l'on a goûté à ce type d'interface, il est difficile de retourner en arrière.

POV-RAY existe depuis la préhistoire de l'informatique et a grandement évolué avec la technologie et les possibilités des ordinateurs. Il est disponible sur quasiment toutes les plateformes, et dispose d'une importante communauté d'utilisateurs prêts à échanger des astuces ou des éléments d'image. Enfin, un point important, POV-RAY est gratuit.

Dans ce manuel, nous considèrerons que vous savez utiliser les fonctions courantes de votre système d'exploitation, mais ne connaissez pas du tout POV-RAY. Nous construirons ensemble et pas-à-pas des images de bonne qualité visuelle.

Lorsque vous rencontrerez ce type de boite, elle contiendra une astuce. Prenez-en note, elle peut vous faire gagner du temps

Table des matières

Chapitre 1

Premiers pas

Sommaire

1.1 Pré-requis

1.1.1 Logiciel

Pour travailler sous POV-RAY, il vous faut...POV-RAY. Si vous ne l'avez pas déjà installé, le fichier d'installation est disponible sur le site officiel, à l'adresse http ://www.povray.org . Profitez-en pour regarder les images que les spécialistes ont créées avec ce logiciel.

Une fois POV-RAY installé, il vous faut un éditeur de texte pour créer les fichiers-source. Si vous travaillez sous Windows, l'éditeur est intégré dans le logiciel. Sous les autres systèmes, tels Linux, vous devez bien avoir un éditeur de textes sous la main.

Ce document est essentiellement basé sur la version Windows de POV-RAY. Mis à part des différences d'interface, le cœur du logiciel est identique.

Pour profiter à fond des possibilités de POV-RAY, un éditeur d'images peut être utile. Par exemple, GIMP (http ://www.gimp.org) est gratuit et très performant.

Enfin, dans un premier temps nous n'avons besoin de rien de plus. Un grand nombre de logiciels ont néanmoin été développés pour travailler avec et autour de POV-RAY. Nous en reparlerons en cas de besoin.

1.1.2 Mathématiques

Je vois déjà certains d'entre vous pâlir. Rassurez-vous, la partie mathématiques nécessaire à l'utilisation de POV-RAY est ridiculement faible et sera très simple. Elle se limite à comprendre le système de coordonnées.

Sur une feuille de papier quadrillée, on peut facilement repérer un point de cette feuille en comptant les carreaux. Le point est donc défini par deux nombres : le nombre de carreaux depuis la gauche, et le nombre de carreaux depuis le bas. Ces deux nombres sont appelés coordonnées.
En trois dimensions, c'est aussi simple, il vous faudra juste trois coordonnées au lieu de deux : la distance au bord gauche, la distance au bas et la profondeur. Ces trois dimensions sont notées x, y et z et sont les coordonnées du point.
Un point est donc défini par ses trois coordonnées. Deux points peuvent définir un *vecteur* : c'est une flêche, avec une direction et une longueur. Un vecteur est aussi défini par trois coordonnées (qui sont la différence des coordonnées des deux points qui le composent).

Le système de coordonnées de POV-RAY est présenté à la figure 1.1. Ainsi, l'axe x est l'axe gauche-droite. L'axe y est l'axe bas-haut et l'axe z est l'axe arrière-avant.

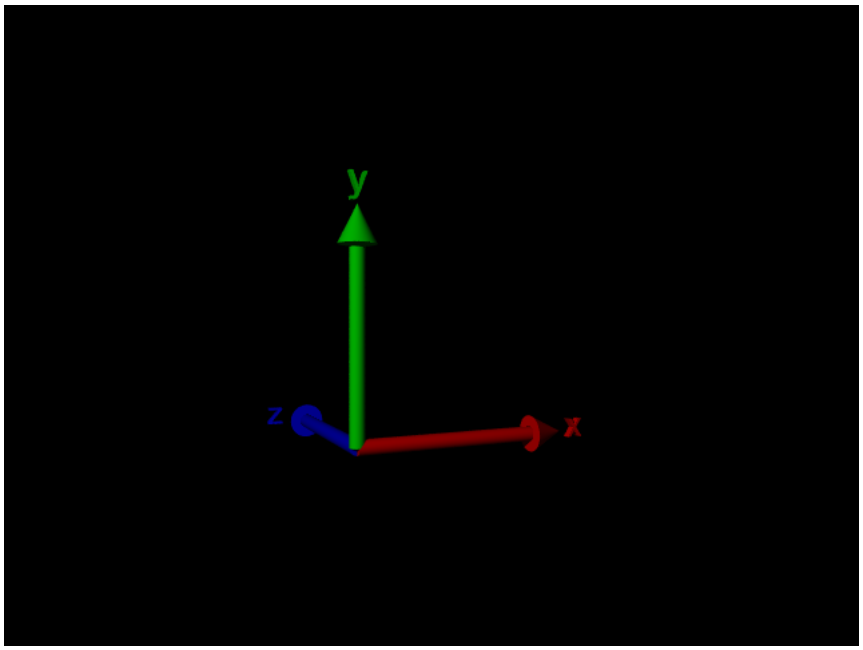

FIG. 1.1 – Le système d'axes

En cas de besoin, la longueur d'un vecteur peut être calculée par Pythagore : $l = \sqrt{x^2 + y^2 + z^2}$

C'est tout pour les mathématiques. Enfin pour l'instant. Vous avez survécu ? Alors on se lance dans le vif du sujet !

1.1.3 Ma première scène

Ouvrez l'éditeur de POV-RAY. Nous allons entrer notre première scène, qui est un grand classique du genre : une sphère réfléchissante sur un damier. Ne vous inquiétez pas si vous ne comprenez pas tout, c'est normal, on aura le temps d'y revenir en temps voulu.

```
sphere{
   <0,1,0>,1
   pigment {color rgb <1,1,1>}
   finish {reflection 0.5}
}

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

```
camera {
   location <0,1,-3>
   look_at <0,0,0>
}

light_source{
   <2,1,-3>
   color rgb <1,1,1>
}
```

Cliquez sur *Run* et regardez le résultat. Si tout se passe bien, vous devriez obtenir une jolie image. Celle-ci est dans le même temps enregistrée dans le dossier où se trouve votre fichier source. Faites quelques essais en changeant les valeurs numériques du code source. Essayez de comprendre à quoi correspondent ces paramètres. Lorsque vous voudrez en savoir plus, passez au paragraphe suivant.

Vous êtes sous Linux ? N'appuyez pas sur *Run*, ce bouton n'existe pas. A la place, sauvez votre fichier, par exemple sous *monfichier.pov* et tapez en ligne de commande *povray monfichier.pov*

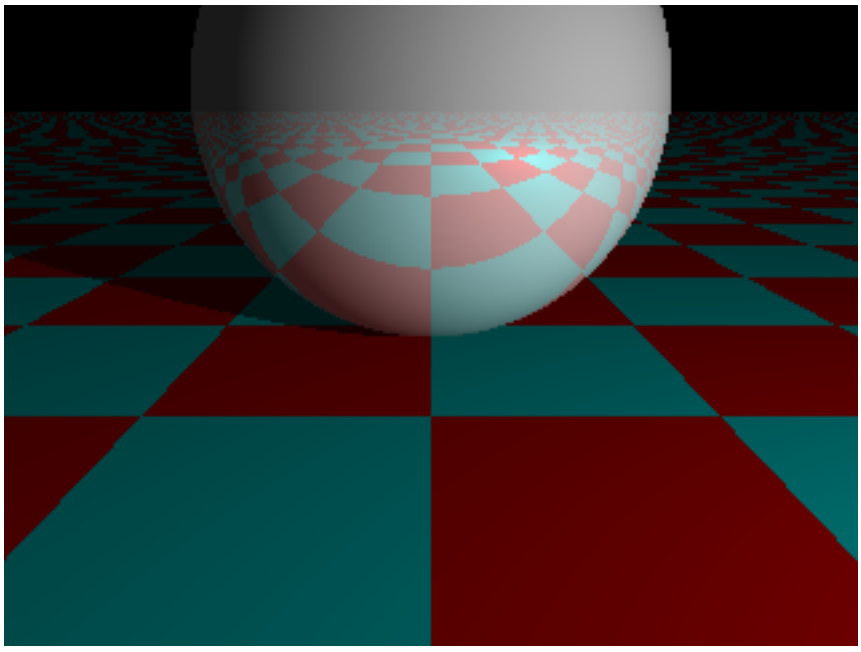

FIG. 1.2 – Ma première scène

1.1.4 Commentaires sur cette scène

Le langage POV-RAY est assez simple, une fois que l'on a retenu quelques règles élémentaires : tout d'abord, les différents objets, tels *sphere* ou *camera* prennent leurs arguments entre accolades { }. Les vecteurs et les coordonnées de points sont entrés entre les signes < et >. Des virgules sont utilisées pour séparer les arguments. Si besoin, le symbole décimal est le point.

Lorsque l'on ouvre une accolade, on définit un nouveau niveau logique. Par souci de lisibilité, on prend l'habitude de décaler le texte entre les accolades de quelques caractères. Ce n'est pas indispensable, mais c'est très utile, notamment dans le cas (très courant) d'accolades imbriquées.

Voyons les différents objets inclus dans cette scène :

sphere

Cette section crée la sphère réfléchissante. Le premier paramètre *<0,1,0>* est la coordonnée du centre de la sphère. La sphère est donc une unité au dessus du centre du monde.

Le point *<0,0,0>* n'est pas la limite du monde mais son centre. Les coordonnées négatives sont bien entendues autorisées.

Le paramètre suivant est le rayon de la sphère. Avec un rayon de 1, la sphère aura donc une élévation comprise entre 0 et 2 (rappel : son centre aune élévation de 1).

Les unités sont arbitraires, 1 ne représente pas 1 mètre, un milimètre ou 1 pouce. Il représente ce que vous voulez.

La section *pigment* définit la couleur de la sphère. Ici, l'argument définit la couleur blanche (*color rgb* indique que l'on va utiliser une couleur dans le système rgb. Le paramètre *<1,1,1>* indique que la couleur comprend 100% de rouge, 100% de vert et 100% de bleu, ce qui est la couleur blanche.

Au départ, il est difficile de se représenter ces codes couleur. Votre logiciel de traitement d'image a des palettes de couleur qui vous permettent de trouver les pourcentages de rouge, vert et bleu de la couleur qui vous intéresse. D'autres méthodes de création de couleur sont présentées plus loin).

La section *finish* définit la finition de l'objet. Ici, *reflection 0.5* indique que l'objet va réfléchir 50% de la lumière.

Exercice 1 *Modifiez la scène pour obtenir une sphère deux fois plus petite, mais qui repose encore sur le plan. Cette sphère sera bleue et très légèrement réfléchissante.*

plane

L'objet *plane* définit un plan infini. Cet objet n'a pas d'épaisseur, pas de volume, mais s'étend à l'infini. Le premier paramètre est le vecteur normal au plan. En clair, pour les non mathématiciens, c'est la direction perpendiculaire au plan. En encore plus clair, *y* signifie que le plan s'étend dans les directions *x* et *z*. En encore plus clair, c'est un plan horizontal.

Comme on le verra plus tard, *y* est équivalent à *<0,1,0>*. Il est possible de définir des plans selon n'importe quelle orientation en utilisant des vecteurs normaux quelconques. Pour les non matheux ou ceux qui ont du mal à voir les vecteurs normaux, on se contentera de plans ■ simples ■ que l'on fera tourner, comme on apprendra bientôt à le faire.

Le nombre *0* est la distance du plan au centre du monde. Ici, le plan passera donc par l'origine.

pigment donne ici encore la couleur du plan. Ici, on n'utilise pas une couleur unie mais un motif. Le motif *checker* est l'un des plus simples disponible. Il crée un quadrillage bicolore, dont les couleurs sont définies par les deux arguments suivants, soit la couleur rouge (*<1,0,0>*) et cyan *<0,1,1>*. Ici, pas de finition, on aura donc un objet mat, sans réflection.

A noter : les motifs sont toujours créés avec une dimension spécifique de l'ordre de l'unité. Ici, les cases de l'échiquier ont un coté de 0,5 unité. Nous apprendrons rapidement à renommer tout ça très bientôt.

Tous les motifs ne seront pas étudiés dans ce livre. Lorsque vous aurez une vision plus approfondie du logiciel, reportez-vous à l'aide de POV-RAY pour en apprendre plus.

Exercice 2 *Rendez le plan aussi réfléchissant que la sphère.*

camera

Maintenant que nous avons créé nos objets, il faut pouvoir les regarder. C'est le but de l'objet camera. Celle-ci fonctionne comme une vraie caméra, avec des possibilités de réglage d'ouverture focale, de largeur de champ et autres. . .Dans cet exemple, nous utilisons une caméra très simple, mais fonctionnelle. Le premier paramètre, *location* donne la position de la caméra. Le point choisi ici (*<0,1,-3>*) se trouve en face du centre de la sphère, et un peu en avant. Le second paramètre, *look_at* indique vers quel point regarde

la caméra. Ici, ce point étant le centre de la sphère, la caméra regarde vers notre objet.

Exercice 3 *Dans notre scène, la sphère sort partiellement de notre champ de vision. Essayez de régler ce problème en bougeant la caméra.*

Il ne peut y avoir qu'une seule caméra. Si vous en définissez plusieurs, POV-RAY va râler et ne tenir compte que de la dernière.

light_source

Dans l'obscurité, on ne voit pas grand chose...C'est pour celà que l'on ajoute dans notre scène une source de lumière. La source choisie est la plus simple disponible : c'est une source omnidirectionnelle où les rayons ne sont pas atténués par la distance. Ce n'est pas très réaliste, mais c'est suffisant pour commencer. Quels sont nos paramètres ? Le premier paramètre est la position de la source de lumière. Ici (*<2,1,-3>*), la lumière se trouve un peu sur le coté de la caméra.

Eclairer directement un objet dans l'axe de la caméra écrase les reliefs et met moins bien les objets en valeur. Un contre-jour n'est pas conseillé non plus, car il éclaire la face que l'on ne voit pas...Après, tout dépend de ce que vous voulez faire

Le second paramètre est la couleur de la source de lumière. Nous avons donc une lumière blanche.

A noter : il est autorisé d'avoir plusiques sources de lumière. La seule conséquence est de ralentir les calculs.

Exercice 4 *Essayons un éclairage différent : éclairez la sphère par deux sources de lumière, placées assez loins sur la gauche et la droite de la sphère, et de couleur respective verte et magenta.*

Voilà, vous avez les bases. Dès à présent, nous allons voir quelques détails d'interface, puis nous reprendrons notre scène pour la compliquer un peu.

1.1.5 Options de l'interface

Lancer la scène

Nous en avons déjà parlé. Sous Windows, il est facile de lancer la scène, c'est le bouton *Run*. Sous Linux, nous tapons en ligne de commande *povray monfichier.pov*. Cette opération lance les calculs, affiche l'image et la sauvegarde dans le dossier où se trouve votre fichier source. Différentes options sont disponibles pour modifier la manière dont POV-RAY crée son image.

Résolution

La résolution, c'est la taille de l'image finale. Sous Windows, un menu déroulant vous permet de choisir la résolution voulue. Sous Linux, c'est presque aussi simple. Il suffit de rajouterr deux arguments en fin de ligne. Par exemple, la ligne de commande *povray monfichier.pov -W800 -H600* Va lancer le calcul d'une image de 800 points de large (Width en anglais) sur 600 de haut (Height).

Sous windows, l'interface propose une petite ligne permettant d'entrer les arguments de ligne de commande. C'est le moyen le plus simple pour créer une image de taille non standard par exemple

Pour l'instant nos images sont très simples, et sont calculées en un temps très court. Celà ne sera pas toujours le cas. Augmenter la résolution d'une image augmente nécessairement son temps de rendu.

Si vous tentez un rendu dans une résolution qui n'est pas au format 4/3, vous observerez une distorsion de l'image. Pas de panique, nous apprendrons à régler la caméra en conséquence bientôt.

Désaliasage

Cet acronyme est connu en anglais sous le nom d'Antialiasing. Son but est de gommer les marches d'escalier que l'on peut voir sur une ligne droite penchée. Cette opération est réalisée en calculant si nécessaire plusieurs points pour chaque point de l'image finale. POV-RAY est capable de savoir de lui-même s'il est nécessaire de calculer ces sous-points. comment activer cette option ? Sous Windows, l'option est dans le menu déroulant de choix de résolution : ce sont toutes les résolutions indiquées par le code *AA 0.3*. Sous Linux, ajoutez *-A0.3* à votre ligne de commande.

La valeur 0.3 est un indice de qualité. Elle est comprise entre 0 et 1. Plus elle est élevée, plus l'image est fine, mais plus le temps de calcul augmente. 0.3 est généralement un bon compromis.

Le désaliasage augmente de manière conséquente le temps de rendu de l'image.

Reprise d'images

Il peut arriver de ne pas avoir lee temps de terminer un calcul. Dans ce cas, pas de panique. Arrêtez POV-RAY. Pour reprendre le calcul, ajoutez *-C*

en ligne de commande. Evidemment, celà ne fonctionne correctement que si la scène n'a pas changé et si les paramètres de rendu (notamment la taille de l'image) sont identiques.

1.2 De nouveaux objets

Nous ne pouvons pas nous contenter de sphères et de plans infinis. Nous allons donc ajouter quelques objets supplémentaires à notre base de connaissance. Ces objets sont désignés sous le terme général de *primitives*. Reprenons le code source de notre sphère sur damier et effaçons la sphère. Profitons-en pour reculer un peu la caméra afin de voir l'objet dans son ensemble. Ce code minimal sera la base de nos essais dans ce chapitre. Vous avez oublié ou perdu ce code source ? Pas de panique, le voici !

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

camera {
   location <0,1,-6>
   look_at <0,0,0>
}

light_source{
   <2,1,-3>
   color rgb <1,1,1>
}
```

Avant de commencer, un dernier détail : l'ordre dans lequel nous insérons les objets dans le code n'a, pour l'instant, aucune importance.

1.2.1 box

Allons-y, ajoutons un nouvel objet : la boite. Une boite est définie par les coordonnées de deux de ses sommets opposés. Il n'est pas nécessaire (et pas possible) de définir à la main les autres points, POV-RAY s'en charge. Ajoutez-donc le code suivant à votre source :

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

Evidemment, nous sommes en plein dans l'axe, ce qui cache la profondeur de l'objet. D'où :

Exercice 5 *Déplacez la caméra afin d'obtenir quelque chose ressemblant à la figure 1.3*

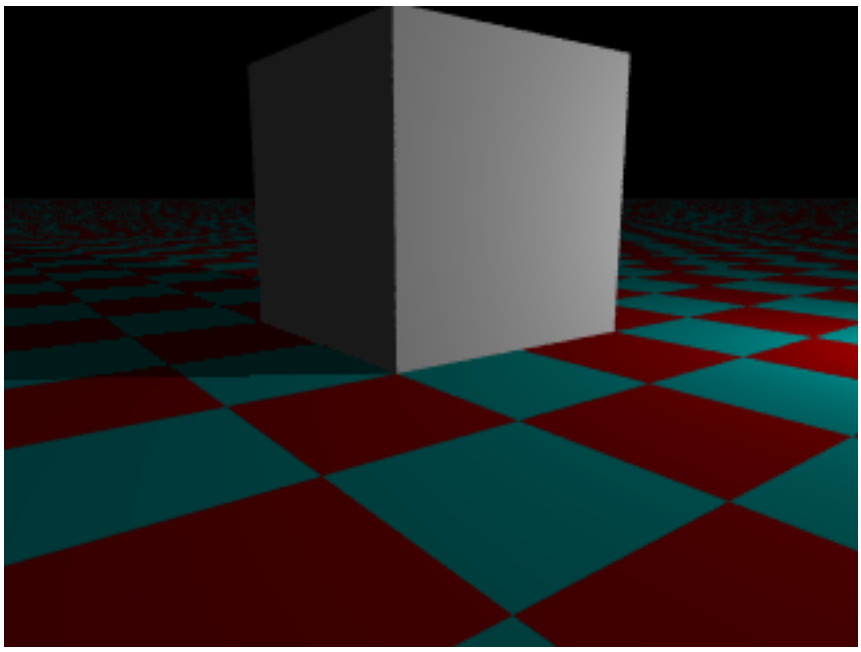

FIG. 1.3 – Une boite, sous un autre angle de vue

Au niveau du code, rien de particulier : juste les coordonnées de deux sommets, séparés par une virgule. N'oubliez surtout pas les < et > ! Ensuite, on retrouve notre *pigment*, et on peut éventuellement y ajouter des *finish* ou bien d'autres surprises (ne gâchons pas le suspense, nous ne sommes qu'au début du livre).

Quelque chose peut frapper rapidement : cette syntaxe crée systématiquement des boites dont les cotés sont parallèles aux axes de notre repère. Et ici, par exemple, parallèles au quadrillage du plan. C'est très restrictif. Heureusement, il sera très facile de régler ça. Avant la fin de ce chapitre, vous saurez tout à ce sujet...

une boîte très plate peut faire une bonne planche. Beaucoup d'objets manufacturés sont composés de ■ boîtes ■

1.2.2 cylinder

Présent, un objet un peu moins régulier. J'ai nommé le Cylindre. Comment créer un cylindre ? Avec le mot-clef *cylinder*. La syntaxe

est simple : il nous faut les coordonnées du centre des deux faces circulaires, ainsi que le rayon de ces faces. Un exemple ? faites sauter le code de la boite précédente et injectez ceci :

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

Un cylindre avec un petit rayon et des centres très éloignés peut donner une barre ou le corps d'un crayon. Un cylindre plus écrasé avec un gros rayon donnerait un camembert. En écrasant encore un peu le cylindre, on peut obtenir une crêpe.

Pas de commentaire ? C'est facile, hein ? Alors on continue !

1.2.3 cône

t si l'on pouvait faire varier le rayon du disque le long du cylindre ? Qu'obtiendrait-on ? Une section de cône. Et avec quoi crée-t-on une section de cône ? Avec le mot-clef *cone*.

Section de cône

La primitive *cone* demande 4 arguments : les coordonnées du centre d'une face, le rayon de cette face, les coordonnées de la seconde face et enfin son rayon, le tout séparé par des virgules, s'il-vous-plaît. En bref, il faut juste un paramètre de plus que précédemment : le second rayon. Allez, l'exemple !

```
cone{
   <0,0,0>,1,<0,2,0>,0.5
   pigment {color rgb <1,1,1>}
}
```

Vrai cône

Un vrai cône est une section de cône où l'un des rayons est égal à zéro. Essayez donc par exemple :

```
cone{
   <0,0,0>,1,<0,2,0>,0.5
   pigment {color rgb <1,1,1>}
}
```

Exercice 6 *Sauriez-vous poser le cône sur sa pointe ?*

Cas particuliers

Des choses amusantes surviennent parfois lorsque l'on pousse un objet à la limite de ce que pourquoi il est prévu. Voyons dans le cas du cône ce qu'il peut arriver.

Qu'est une section de cône lorsque ses deux rayons sont égaux ? Un cylindre, tout simplement.

Et si ses deux rayons sont nuls ? L'objet disparaît, mais celà ne gêne pas POV-RAY qui calcule tranquillement son image, sans même un message d'avertissement.

Par contre, si les deux centres sont confondus, POV-RAY s'arrête net et nous parle de cône dégénéré.

D'une manière assez curieuse, les rayons peuvent être négatifs. Si les deux rayons sont négatifs, cela ne change pas grand chose, l'objet est équivalent à son homologue aux rayons positifs. Par contre, si l'un des deux seulement est négatif, celà devient intéressant : lorsque l'on parcourt la hauteur du cylindre, le rayon diminue, passe par zéro, puis, augmente à nouveau. Voyez l'exemple ci-dessous :

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

Nous avons assez d'objets pour commencer à travailler. Dans la section suivante nous apprendrons à transformer nos objets. En attendant, un dernier petit exo.

Exercice 7 *Sauriez-vous dessiner un cylindre surmonté d'un ■ chapeau ■ ? Ce chapeau pourra être un cône ou une demi sphère.*

Observez les objets créés dans cet exercice. On arrive déjà à des solides de forme plus complexe. Observez les objets de tous les jours et essayez de les décomposer en boites, cylindres, sections de cônes et sphères. Vous avez une idée de ce qui peut être réalisé.

1.3 Transformations !

Dans cette section, nous allons tout d'abord apprendre à créer des raccourcis qui nous éviterons de taper plusieurs fois des sections de code utilisées à différents endrois de notre scène. Ensuite, et fort de ce nouvel outil, nous allons transformer nos objets, changer leur taille, leur position, les faire tourner. . .Une fois ces derniers concepts digérés, nous serons prêts à nous lancer dans un vrai projet.

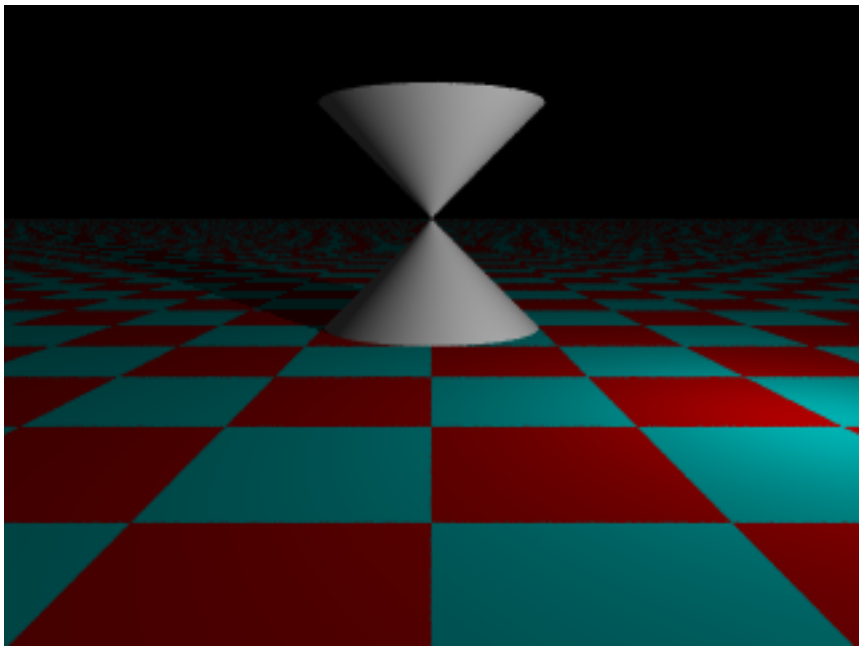

FIG. 1.4 – Un cône avec un seul rayon négatif

1.3.1 Déclarations

Où l'art de recycler. Pour l'instant nous avons des objets assez simples. Cela va se compliquer rapidement. Si une définition d'objet prend dix lignes et que je dois la retaper à chaque fois que je m'en sers, c'est du temps de perdu. La solution à tous nos malheurs se trouve dans le mot-clef *#declare*.

Introduction

Cette commande est la première commande à commencer par un # que nous rencontrons. Ces commandes ont toutes un point commun : contrairement aux autres commandes, elles ne créent pas directement un objet.

La commande *#declare* permet de définir un nouveau mot-clef. Celui-ci peut contenir un chiffre, un objet, un pigment, un finish, une source de lumière, une couleur, et bien d'autres objets encore. La syntaxe est extrêmement simple, comme vous pouvez le juger sur l'exemple suivant : comparez ce code-source avec le suivant.

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

```
   location <0,1,-6>
   look_at <0,2,0>
}
light_source{
   <2,1,-3>
   color rgb <1,1,1>
}
sphere{
   <0,1,0>,1
   pigment {color rgb <1,1,1>}
   finish{ reflection 0.5}
}
sphere{
   <-2,1,0>,1
   pigment {color rgb <1,1,1>}
   finish{ reflection 0.5}
}
sphere{
   <2,1,0>,1
   pigment {color rgb <1,1,1>}
   finish{ reflection 0.5}
}
sphere{
   <0,3,0>,1
   pigment {color rgb <1,1,1>}
   finish{ reflection 0.5}
}
```

```
#declare BLANC=pigment{color rgb<1,1,1>}
#declare REFLECHISSANT=finish{reflection 0.5}

plane {
   y,0
   pigment {checker color rgb <1,0,0> color rgb <0,1,1>}
}
camera {
   location <0,1,-6>
   look_at <0,2,0>
}
light_source{
   <2,1,-3>
   color rgb <1,1,1>
}
sphere{
```

```
   <0,1,0>,1
   pigment {BLANC}
   finish{REFLECHISSANT}
}
sphere{
   <-2,1,0>,1
   pigment {BLANC}
   finish{REFLECHISSANT}
}
sphere{
   <2,1,0>,1
   pigment {BLANC}
   finish{REFLECHISSANT}
}
sphere{
   <0,3,0>,1
   pigment {BLANC}
   finish{REFLECHISSANT}
}
```

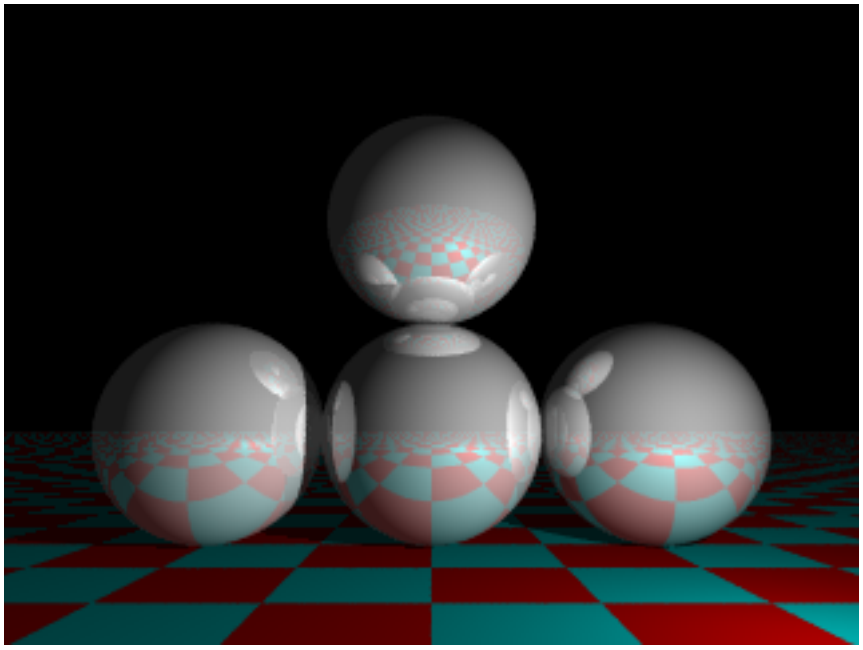

FIG. 1.5 – Une scène d'exemple pour l'utilisation de #declare

Outre une plus grande lisibilité, l'intérêt est immédiat. Si l'on veut modifier la couleur des sphères, il suffit de changer une ligne et non pas quatre.

Exercice 8 *A ce propos, sauriez-vous recolorer les sphères en vert ?*

texture

Une solution plus élégante peut être utilisée lorsque l'on sait que *texture* est un conteneur servant à englober *pigment* et *finish* (ainsi que *normal* qui sera vu plus tard. Regardez ce bout de code :

```
#declare CHROME=texture{
   pigment{color rgb<1,1,1>}
   finish{reflection 0.5}
}
sphere{
   <0,1,0>,1
   texture{CHROME}
}
```

objets entiers

Enfin, et cela sera utile dès le paragraphe suivant, un objet entier peut être englobé dans un *#declare*. Il sera alors appelé par *object*, comme dans l'exemple suivant :

```
#declare BOULE=sphere{
   <0,1,0>,1
   texture{CHROME}
}

object{BOULE}
```

Notez au passage l'utilisation de définitions dans les définitions.

Il est nécessaire que la déclaration soit effectuée avant l'endroit où elle est appelée.

Dans ce paragraphe, nous avons vu plusieurs notions. Avant de passer à la suite, vérifiez que les mots-clef suivant veulent dire quelque chose pour vous : *#declare*, *texture*, *object*.

Un dernier point : *#declare* peut aussi accepter des valeurs numériques. Dans ce cas, la commande est suivie d'un point-virgule. En cas d'oubli, POV-RAY va râler.

```
#declare TroisFoisHuit=24;
```

POV-RAY respecte la casse. Ainsi, pour lui, *blanc*, *Blanc* et *BLANC* sont trois objets différents.

La commande *#include*

Une application intéressante de la fonction *#declare* est la création d'un catalogue de couleurs. Il peut être intéressant d'inclure en début de fichier une partie de code qui ressemblerait à ceci :

```
#declare ROUGE=color rgb <1,0,0>;
#declare JAUNE=color rgb <1,1,0>;
#declare ORANGE=color rgb <1,0.5,0>;
```

Cependant, il est assez fastidieux de retaper ceci à la main à chaque fois que celà est nécessaire. La solution est plus élégante et fait appel à un fichier d'inclusion. Un fichier d'inclusion est un fichier portant le suffixe *.inc*. Il est appelé par la fonction *#include*, suivie du nom du fichier entre guillemets. Lors de son appel, POV-RAY se comporte exactement comme si l'on remplaçait cette commande *#include* par le contenu du fichier.

De tels catalogues de couleur existent. Ainsi, *colors.inc* est distribué avec POV-RAY et contient un assez grand nombre de couleurs (noms en anglais). Il est donc appelé par la commande

```
#include "colors.inc"
```

D'autres catalogues existent. Par exemple, l'auteur propose le package *couleur.inc* sur le site web *http://croquis.over-blog.com*.

Nous en savons assez pour commencer les transformations proprement dites.

1.3.2 La translation

Voyons dès à présent la première et la plus simple des transformations : la translation. Lorsque la commande *translate* <*x,y,z*> est utilisée à l'intérieur de la définition d'un objet, d'une texture ou d'un groupe *object*, elle décale la position de cet élément d'un vecteur <*x,y,z*>, soit, de x unités vers la droite, y vers le haut et z vers le fond. Ainsi, les trois exemples suivants donnent le même résultat :

```
sphere{
   <0,1,0>,1
```

```
   texture{CHROME}
}
```

```
sphere{
   <0,0,0>,1
   translate <0,1,0>
   texture{CHROME}
}
```

```
#declare boule=sphere{
   <0,0,0>,1
   texture{CHROME}
}
object{boule translate <0,1,0>}
```

Ceci va nous être très utile pour rendre nos scènes plus compactes et plus lisibles.

Exercice 9 *Réécrivez de manière plus compact la scène exemple de ce chapitre avec les quatre sphères.*

Maintenant que nous connaissons la translation, prenons l'habitude, lorsque nous créons un nouvel objet, de le centrer sur l'origine. Simplicité et lisibilité...

1.3.3 La mise à l'échelle

Uniforme

maginons que je veuille réaliser une scène comportant plusieurs sphères de taille différente. Je ne peux plus, à priori, utliser la technique des déclarations, à moins de disposer d'un outil permettant de changer leur taille. Cet outil, c'est la fonction *scale*. En effet, il suffit par exemple d'ajouter *scale 2* pour doubler la taille de l'objet.

Attention : *scale* multiplie les coordonnées : pour un objet non centré sur l'origine, la taille varie, mais la position aussi. Ainsi, *box*{*<1,1,1>,<2,2,2> scale 10*} est équivalent à *box*{*<10,10,10>,<20,20,20>*}. Le piège est classique lorsque l'on débute sur POV-RAY.

Comme pour la translation, tous les objets, ainsi que les textures, les pigments ou tout ce qui vous passe par la tête, acceptent la mise à l'échelle.

Une version plus aboutie de la mise à l'échelle est disponible, elle est décrite ci-dessous :

Non uniforme

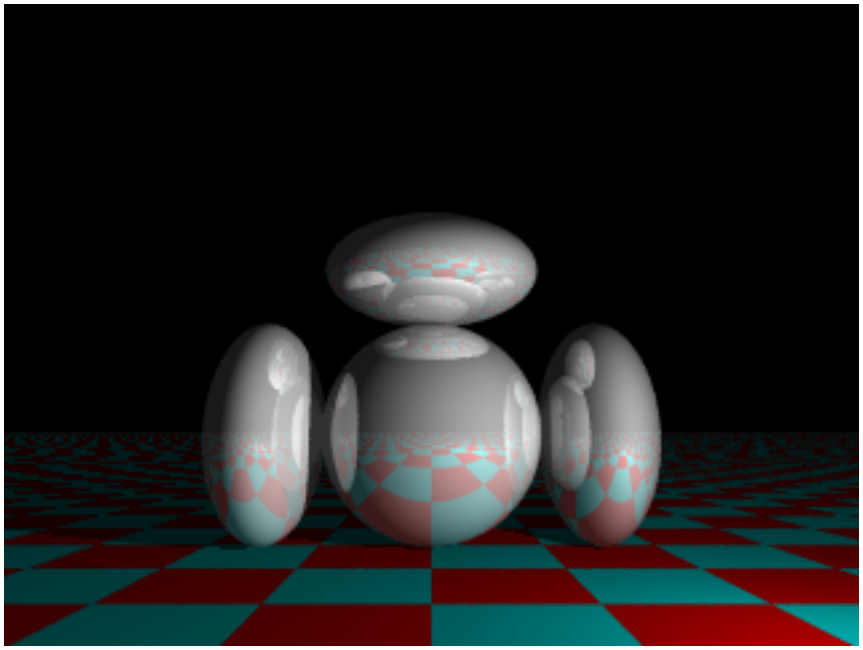

FIG. 1.6 – Application de la mise à l'échelle non uniforme

Voyez la scène exemple : les sphères ne sont plus, euh, sphériques, mais aplaties. Comment faire ? Utilisons la version vectorielle de la mise à l'échelle. La syntaxe est tout simplement *scale <x,y,z>*. Utiliser cette commande va multiplier la longueur de l'objet séparément sur les trois axes. Ainsi, *scale <1,0.5,4>* ne va pas modifier l'axe gauche-droite (les dimensions sont multipliées par 1), diviser par deux les dimensions sur l'axe haut-bas (et donc écraser l'objet) et multiplier par 4 les dimensions sur l'axe avant-arrière (étirant fortement l'objet).

Exercice 10 *Sauriez-vous recréer la scène de l'exemple ?*

Encore une fois, cette transformation donne des objets dont les axes sont alignés avec les axes du système. Nous avions eu le même cas avec les boites et les pigment en quadrillage, notamment, et ce sera le cas pour bien des objets. La solution pour combler ce manque est la rotation.

1.3.4 Rotation

Rotation autour d'un axe principal

La commande pour faire tourner un objet est *rotate <x,y,z>*. Son utilisation la plus intuitive est la rotation autour d'un des axes

principaux. ainsi, la commande *rotate <angle,0,0>* fait tourner l'objet d'un angle *angle* autour de l'axe x. De même, *rotate <0,angle,0>* et *rotate <0,0,angle>* effectuent les rotations selon les autres angles.

Attention à l'ordre des opérations. *rotate <10,0,0> rotate <0,10,0>* n'est pas équivalent à *rotate <0,10,0> rotate <10,0,0>*.

Dans quel sens s'effectue la rotation ? Il est dit que POV-RAY est gaucher. Levez le pouce gauche et alignez-le avec l'axe de rotation (le haut du pouce vers la direction positive). Fermez alors la main. Les doigts s'enroulent dans le sens de la rotation.

Les rotations sont toujours effectuées par rapport à l'origine. Un objet centré sur l'origine subira donc une vraie rotation. Voyez l'exemple suivant :

```
#declare boite=box{
   <-1,-0.5,-1>,<1,0.5,1>
   pigment {color rgb <1,1,1>}
}

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

camera {
   location <0,1,-6>
   look_at <0,2,0>
}

light_source{
   <2,1,-3>
   color rgb <1,1,1>
}


object{boite translate <0,0.5,0>}
object{boite rotate <0,30,0> translate <0,1.5,0>}
object{boite rotate <0,60,0> translate <0,2.5,0>}
```

L'exemple suivant montre une rotation excentrée. Dans ce cas, les objets ont subi un *translate* préliminaire et ne sont donc plus centrés sur l'origine. La rotation s'effectuant par rapport au centre de l'univers, les objets sont aussi déplacés, un peu comme une planète tournant autour du soleil. Notez

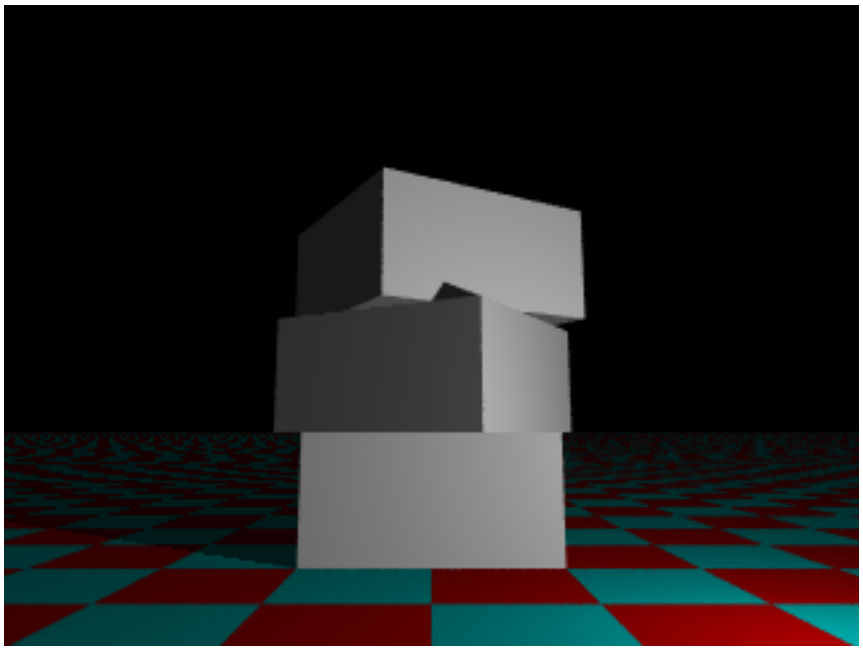

FIG. 1.7 – Exemple de rotation centrée

que la translation appliquée est la même pour toutes les boites. Enfin, une petite sphère verte indique le centre du monde.

```
#declare boite=box{
   <-1,-0.5,-1>,<1,0.5,1>
   pigment {color rgb <1,1,1>}
}

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

camera {
   location <0,15,0>
   look_at <0,2,0>
}

light_source{
   <15,15,0>
   color rgb <1,1,1>
}
```

```
object {boite translate <5,0.5,0>}
object {boite translate <5,0.5,0> rotate <0,30,0>}
object {boite translate <5,0.5,0> rotate <0,60,0>}
object {boite translate <5,0.5,0> rotate <0,90,0>}
object {boite translate <5,0.5,0> rotate <0,120,0>}
object {boite translate <5,0.5,0> rotate <0,150,0>}
object {boite translate <5,0.5,0> rotate <0,180,0>}
object {boite translate <5,0.5,0> rotate <0,210,0>}
object {boite translate <5,0.5,0> rotate <0,240,0>}
object {boite translate <5,0.5,0> rotate <0,270,0>}
object {boite translate <5,0.5,0> rotate <0,300,0>}
object {boite translate <5,0.5,0> rotate <0,330,0>}

sphere{<0,0,0>,0.3 pigment {color rgb <0,1,0>}}
```

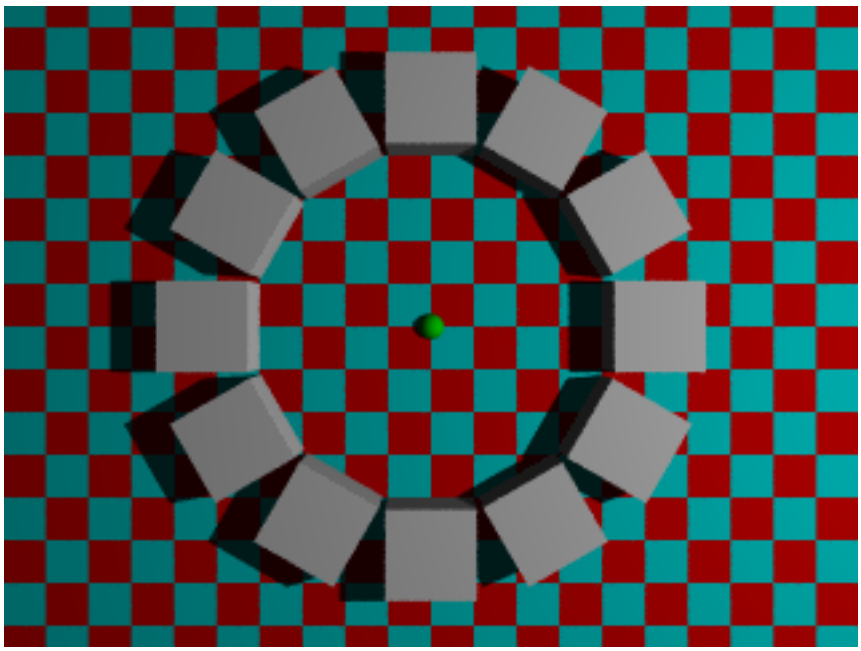

FIG. 1.8 – Exemple de rotation non centrée

concepts avancés

Commençons par une notation bien pratique : *rotate <30,0,0>* peut s'écrire aussi *rotate 30*x*. C'est plus court et plus lisible. Evidemment, les autres axes sont aussi définis.

Il est possible de composer plusieurs rotations. Ainsi, *rotate 30*x rotate 20*y* fait tourner l'objet de 30 degrés autour de l'axe x, puis (et dans cet ordre), de 20 degrés autour de l'axe y. Attention encore une fois à l'ordre des rotations.

Et attention, le résultat n'est absolument pas égal à *rotate 30*x+20*y*

La rotation est aussi définie pour des valeurs quelconques du vecteur x,y,z. Dans ce cas, la rotation a lieu autour d'un axe ayant la même direction que ce vecteur, et d'un angle égal à la norme de ce vecteur. Bref, c'est compliqué et pas facile à visualiser. On l'utilisera donc peu et on préfèrera composer plusieurs rotations entre elles.

Deux petits exercices pour conclure.

Exercice 11 *Sauriez-vous dessiner un cube reposant sur sa pointe ?*

Exercice 12 *Mon objet est centré sur le point* <1,0,0>. *Comment le faire tourner de 45 degrés autour de l'axe* y, *mais sans qu'il change de place ?*

1.4 Pour aller plus loin

Cette petite section concluera chaque partie de ce livre. Elle consistera en un petit bilan des notions apprises et des pistes d'approfondissement pour les curieux. Ces approfondissements ne seront jamais nécessaires pour comprendre la section suivante. Par contre, si certains concepts résumés ici ne vous rappellent rien, retournez un peu en arrière.

Dans cette première partie, nous avons fait connaissance avec POV-RAY et appréhendé les bases du logiciel. Nous avons vu la quantité minimale d'information nécessaire pour créer une scène : une *camera* et une *light_source*. Ces deux objets ont été survolés, nous y reviendrons plus tard.

Nous avons appris les bases de l'interface de POV-RAY. Comment lancer un calcul, le reprendre en cours, comment changer la résolution et le désaliasage. D'autres options de ligne de commande peuvent être utiles. Les curieux pourront parcourir ce chapitre de la documentation.

Nous avons vu un certain nombre d'objets de base, qui sont *box*, *sphere*, *cylinder* et *cone*. Vous pouvez jeter un œil sur la doc officielle pour avoir plus d'informations sur ces objets. D'autres objets peuvent vous intéresser. Voyez notamment *torus* et à la limite *text*. Les autres objets viendront en temps voulu.

Un autre type d'objet effleuré est le plan infini *plane*. Plus d'infos peuvent être trouvées dans la doc.

Nous avons effleuré le système de texturation, avec les concepts de *pigment* et *finish*. Un exemple de *finish* a été présenté : *reflection*, tandis qu'un

motif particulier a été utilisé comme *pigment* : *checker*. Nous en avons parlé car c'était nécessaire pour notre première scène. Nous reviendrons bien en détail un peu plus tard, ce n'est pas nécessaire d'approfondir maintenant. De même, nous avons noté le mot-clef *texture* qui reprend les *pigment* et *finish*.

Nous avons vu comment définir une couleur grâce à *color rgb*. Des infos intéressantes sont disponibles dans la doc. Par contre, ne vous embêtez pas avec les *rgbt* ou *rgbf*, nous en parlerons bientôt.

Deux directives ont été abordées : *#declare* et *#include*. Il n'y a plus rien à ajouter dessus. Par curiosité, regardez dans le dossier */include* du répertoire d'installation de POV-RAY, vous aurez un aperçu de ce que les développeurs nous ont laissé comme cadeaux.

Enfin, nous avons vu quelques transformations : *translate*, *scale* et *rotate*, que vous pouvez aprrofondir un peu en lisant la doc. Une dernière transformation existe, c'est la transformation *matrix* qui est plus complexe et réservé aux moins allergiques aux mathématiques.

Vous avez survécu ? Alors préparez-vous pour la suite.

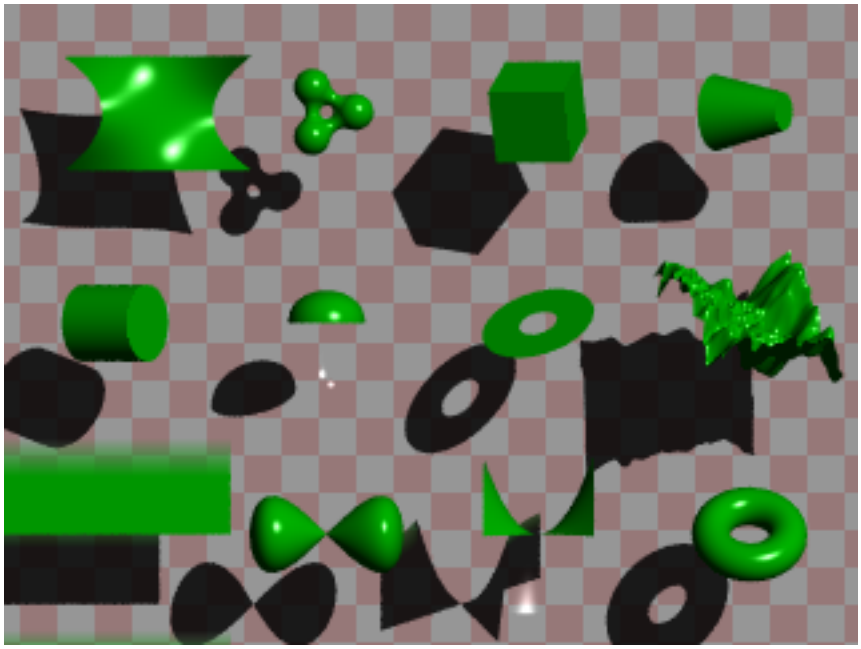

FIG. 1.9 – primitiv.pov : un fichier d'exemple montrant quelques objets de base

Chapitre 2

Un premier vrai projet : le 421

Sommaire

2.1 Mise en place

Nous allons commencer notre premier projet. Nous allons modéliser une partie de 421, qui comportera donc trois dés lancés sur une piste de jeu. Le tout sera posé sur une table en bois, à coté d'un verre de whisky.

2.1.1 Un fichier de test

La première étape va consister à créer un fichier test. Ce fichier servira à tester les objets indépendamment de la scène elle-même. Allons-y, créons donc une scène vide comportant une source de lumière, une caméra et un plan quadrillé, qui servira de repère au niveau dimensions. Ca vous rappelle quelque chose ? Alors voici le code source choisi :

```
plane{y,0
   pigment{checker color rgb <1,0.5,0.5>,color rgb <0.5,1,1>}
}
light_source{<-2,1,-6> color rgb <1,1,1>}
camera {location <0,3,-4> look_at <0,1,0>}
```

Sauvez ce code minimaliste dans un fichier, nommez-le par exemple *test.pov*.

Vous avez peut-être remarqué que parfois les arguments d'un objet sont séparés par une virgule, parfois pas. La virgule peut être supprimée s'il n'y a pas d'ambiguité. Dans les cas douteux, mettez des virgules.

2.1.2 Commentaires

Nous abordons un projet de taille appréciable. Si vous le reprenez dans quelques mois, il est fort possible que vous ne puissiez pas vous relire. Les commentaires sont là pour vous guider. Nous avons deux méthodes pour créer des commentaires.

Commentaires en fin de ligne

Le code // délimite des commentaires. Tout ce qui le suit ne sera pas lu par POV-RAY. Voyez l'exemple :

```
sphere{ // une boule
   <0,0,0> // centrée à l'origine
   1 // de rayon 1
   pigment {color rgb <1,0,0>} // rouge
}  // et c'est tout
```

paragraphes en commentaire

Tout bloc placé entre les codes /* et */ est considéré comme commentaire. Un intérêt de ce système est de commenter une partie du code source d'une scène afin de faire des rendus rapides. Voyez :

```
/*
sphere{
   <0,0,0>,1
   pigment {color rgb <1,0,0>}
}
Cette sphère n'est pas affichée
*/
```

2.1.3 Quelques déclarations de base

Nous allons commencer notre fichier en définissant quelques couleurs. Ouvrez un document vide et entrez le code suivant :

```
#declare T_BOIS=texture{pigment{color rgb <0.6,0.2,0.2>}}
#declare T_DE=texture{pigment{color rgb <0.9,0.9,0.7>}}
#declare T_POINT=texture{pigment{color rgb <0.2,0.2,0.2>}}
#declare T_PLATEAU=texture{pigment{color rgb <0.7,0.1,0.1>}}
#declare T_FEUTRE=texture{pigment{color rgb <0,1,0>}}
#declare T_VERRE=texture{pigment{color rgb <1,1,1>}}
#declare T_WHISKY=texture{pigment{color rgb <0.9,0.9,0>}}
```

Qu'avons-nous fait ? Nous avons créé sept textures, qui sont des pigments unis. Ce ne sera pas, bien évidemment, les textures finales, mais cela nous permettra de commencer à modeler les objets. Dans une autre étape nous travaillerons ces textures. Ces textures seront appliquées, dans l'ordre, au bois de la table, aux dés, aux points des dés, au plateau de jeu, à la toile de feutre du plateau, au verre et au whisky.

Pourquoi avons-nous utilisé des *texture* alors que nous n'avons que des *pigment* ? Nous savons que les *texture* servent à englober différents éléments, dont des *pigment* et des *finish*. Par la suite, nous modifierons le code, et le délimiteur *texture* sera nécessaire. Alors nous l'insérons tout de suite.

2.2 Sculpture du dé

llons-y pour notre premier objet : un dé. Qu'est-ce qu'un dé ? C'est un cube avec des points peints ou creusés sur chacune de ses faces.

Nous allons nous placer dans le cas le plus complexe, et donc le plus intéressant : les points seront creusés dans le dé, et l'intérieur de ces trous sera peint en noir. Pour rendre le dé encore plus esthétique, nous allons arrondir les coins.

Le cube servant de base à notre dé sera un cube de longueur 1, centré sur l'origine. ses coordonnées varieront donc de -0,5 à 0,5.

Evidemment, nous allons y aller par étapes.

2.2.1 Arrondir les coins

Regardez la figure 2.1 : le cube de droite, avec des coins arrondis, peut être obtenu en prenant le cube entier de gauche, auquel on a coupé tous les morceaux qui dépassent de la sphère. Autrement dit, ce qui nous intéresse, c'est l'intersection entre le cube et la sphère.

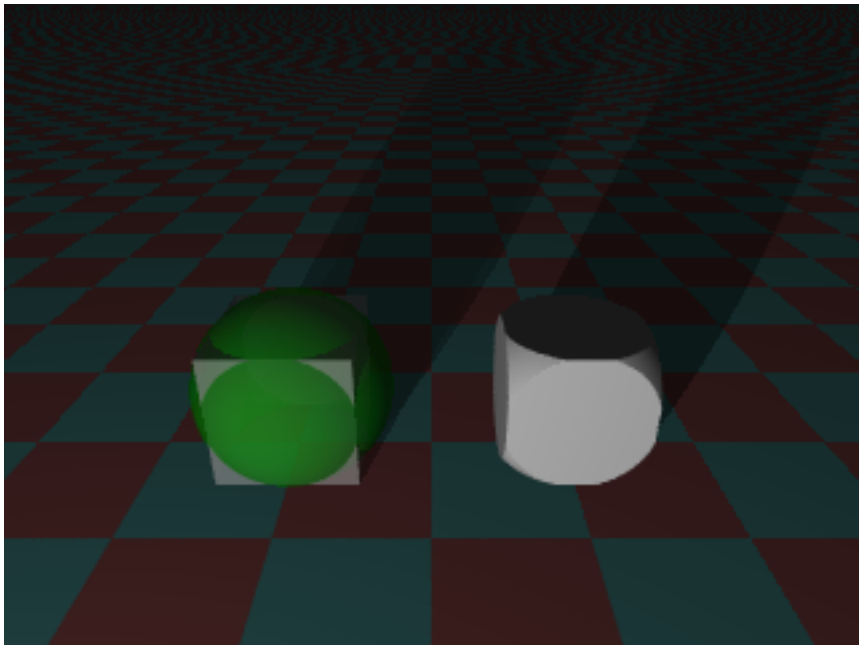

FIG. 2.1 – Arrondir les coins

Comment réaliser une intersection ? grâce au conteneur *intersection*{ }. Celui-ci doit contenir la liste des objets dont on veut l'intersection. On peut y ajouter par la suite une définition de texture ou des transformations, comme tout objet. Essayez par exemple de copier le code suivant dans la scène de démo :

```
intersection{
   sphere{<0,0,0>,0.72  }
   box{<0.5,0.5,0.5>,<-0.5,-0.5,-0.5>}
```

```
   texture{T_DE}
   translate 0.5*y
}
```

Si vous lancez le code tel quel, vous aurez une erreur, car *T_DE* n'est pas défini. Avez-vous pensé à copier la définition de la texture du document principal vers la test ? Par la suite, nous ne rappellerons pas ces lignes évidentes, lorsque les éléments ont déjà été déclarées dans notre document principal.

Exercice 13 *Faites varier le rayon de la sphère pour voir l'influence de ce paramètre. Essayez aussi de faire tourner ce dé. voyez-vous d'où vient la translation de 0,5 unités ?*

Voilà, nous avons le corps du dé de prêt. Avant de passer à la suite, voyons deux détails sur l'intersection.

Tout d'abord, il est possible de définir une texture différente pour les deux objets intervenant dans l'intersection, en incluant la définition des textures dans les objets individuels, et non pas dans l'objet final. Les différentes parties de l'objet final porteront la texture de l'objet initial. Essayez l'exemple ci-dessous.

```
intersection{
   sphere{
      <0,0,0>,0.7
      pigment{color rgb<0,1,0>}
   }
   box{
      <0.5,0.5,0.5>,<-0.5,-0.5,-0.5>
      pigment {color rgb <1,0,1>}
   }

   translate 0.5*y
}
```

Un autre point : lorsque vous effectuez une différence avec des objets qui ont des faces communes, le rendu peut être très bizarre. Dans ce cas, décalez très légèrement l'un des deux objets. Nous rencontrerons plusieurs fois ce phénomène par la suite.

Enfin, notez qu'il n'est pas interdit de faire des intersections de plus de deux objets. Il suffit de les inclure tous dans le bloc *intersection*.

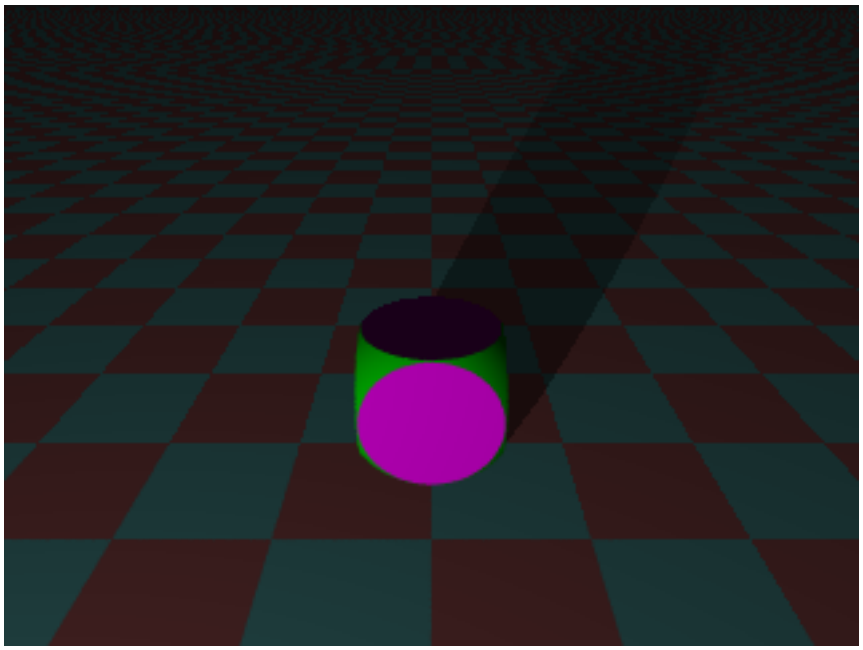

FIG. 2.2 – Une intersection bicolore

2.2.2 Creuser les trous

Nous allons raisonner de la même manière que précédemment. Si je veux créer un point sur mon dé, je vais creuser la surface de mon dé avec une petite sphère, comme le montre la figure suivante :

La solution pour obtenir ce genre d'effet est un conteneur du nom de *difference*. Celui-ci fonctionne d'une manière similaire à l'*intersection*. ici un code du type *difference* {*objetA objetB*} va soustraire l'objet B de l'objet A. C'est ce que nous voulons faire ici, en soustrayant une petite sphère de notre grand cube. Le code-source est donc le suivant (pensez à le coller dans votre document de test) :

```
difference{
   box{<0.5,0.5,0.5>,<-0.5,-0.5,-0.5>}
   sphere {<0,0,-0.5>,0.1}
   translate <0,0.5,0>
   texture{T_DE}
}
```

Avant de continuer, faites des tests avec ce bloc jusqu'à ce que vous compreniez son finctionnement. Notamment, notez que l'ordre des objets a une grande importance. Le premier est solide, le second est le trou. remarquez aussi que des morceaux de l'objet B peuvent dépasser de l'objet A. Ils seront

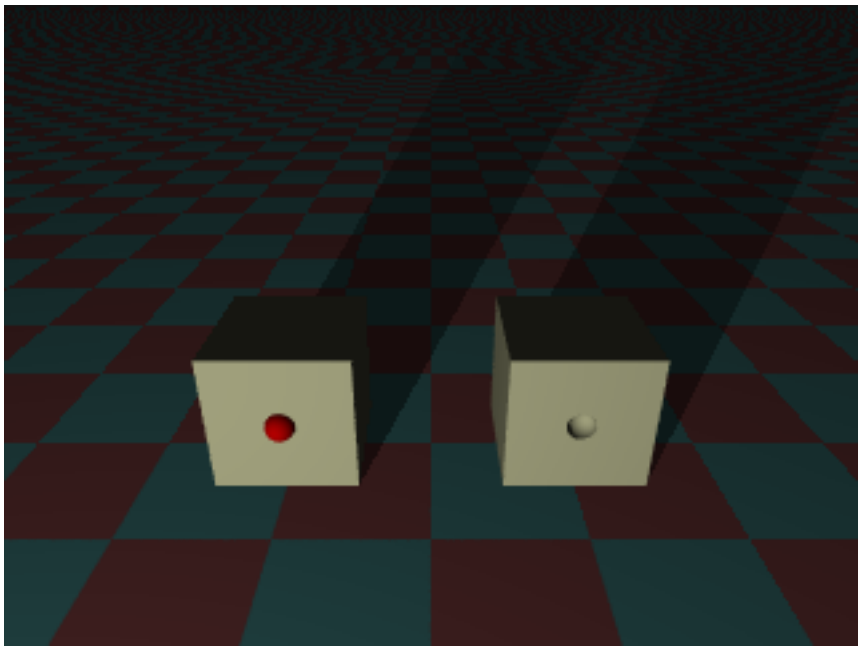

FIG. 2.3 – Comment creuser son trou ?

tout simplement ignorés.

Il nous reste deux choses à voir. Tout d'abord, dans un dé, il y a plus qu'un point. Le conteneur *difference* accepte plus de deux objets. Dans cee cas, le premier est positif, et tous les autres sont les trous. voyez le code suivant :

```
difference{
   box{<0.5,0.5,0.5>,<-0.5,-0.5,-0.5>}
   sphere {<0,0,-0.5>,0.1}   // les trous spheres
   sphere {<-0.3,-0.3,-0.5>,0.1}  // représentent la face 3
   sphere {<0.3,0.3,-0.5>,0.1}  // il reste 5 autres faces
   translate <0,0.5,0>
   texture{T_DE}
}
```

Un dernier point à régler, la couleur. Nous aimerions que les points du dé soient d'une autre couleur, et c'est pour celà que nous avions défini une texture *T_POINT*. Essayez l'exemple suivant :

```
difference{
   box{<0.5,0.5,0.5>,<-0.5,-0.5,-0.5>  texture{T_DE}}
   sphere {<0,0,-0.5>,0.1 texture{T_POINT}}
   sphere {<-0.3,-0.3,-0.5>,0.1 texture{T_POINT}}
```

```
   sphere {<0.3,0.3,-0.5>,0.1 texture{T_POINT}}
   translate <0,0.5,0>
}
```

Vous savez presque tout sur l'intersection. A présent, mettons en commun ce que nous avons appris lors des deux derniers paragraphes.

2.2.3 Le dé, version définitive

Nous avons appris à arrondir les angles d'un cube. Nous avons appris à creuser des trous dans un cube. Nous allons pouvoir composer ces deux concepts pour creuser un dé. L'astuce est simple : dans une différence, l'objet A peut être lui-même un objet composé, tel une intersection de deux objets. Une fois ceci vu, le code coule de source, même si il peut paraitre fastidieux.

N'oubliez pas : si vous avez du mal à voir les coordonnées, faites un schéma sur papier quadrillé.

```
#declare trou=sphere{<0,0,0>,0.1 texture {T_POINT} }
// ceci va permettre d'alléger l'écriture
#declare de=difference {
   intersection { // l'intersection est l'objet A
      sphere{<0,0,0>,0.8}
      box{<0.5,0.5,0.5>,<-0.5,-0.5,-0.5>}
      texture{T_DE}
   }
   object{trou translate <0,0.5,0>} //face 1
   object{trou translate <-0.3,-0.5,-0.3>} // face 6
   object{trou translate <0,-0.5,-0.3>}
   object{trou translate <0.3,-0.5,-0.3>}
   object{trou translate <-0.3,-0.5,0.3>}
   object{trou translate <0,-0.5,0.3>}
   object{trou translate <0.3,-0.5,0.3>}
   object{trou translate <-0.5,-0.3,-0.3>} // face 2
   object{trou translate <-0.5,0.3,0.3>}
   object{trou translate <0.5,-0.3,-0.3>} // face 5
   object{trou translate <0.5,-0.3,0.3>}
   object{trou translate <0.5,0.3,-0.3>}
   object{trou translate <0.5,0.3,0.3>}
   object{trou translate <0.5,0,0>}
   object{trou translate <-0.3,-0.3,-0.5>} // face 3
   object{trou translate <0,0,-0.5>}
   object{trou translate <0.3,0.3,-0.5>}
   object{trou translate <-0.3,-0.3,0.5>} // face 4
```

```
   object{trou translate <0.3,-0.3,0.5>}
   object{trou translate <-0.3,0.3,0.5>}
   object{trou translate <0.3,0.3,0.5>}
} // par lisibilité, respectez les espaces
```

Maintenant, copiez cette section de code dans le fichier *421.pov*. Cette définition est définitive, nous n'y reviendrons plus.

Nous avons créé un objet *de*, qui pourra être utilisé en plusiques exemplaires dans une scène.

Exercice 14 *Sauriez-vous créer une scène ressemblant à la figure 2.4 ?*

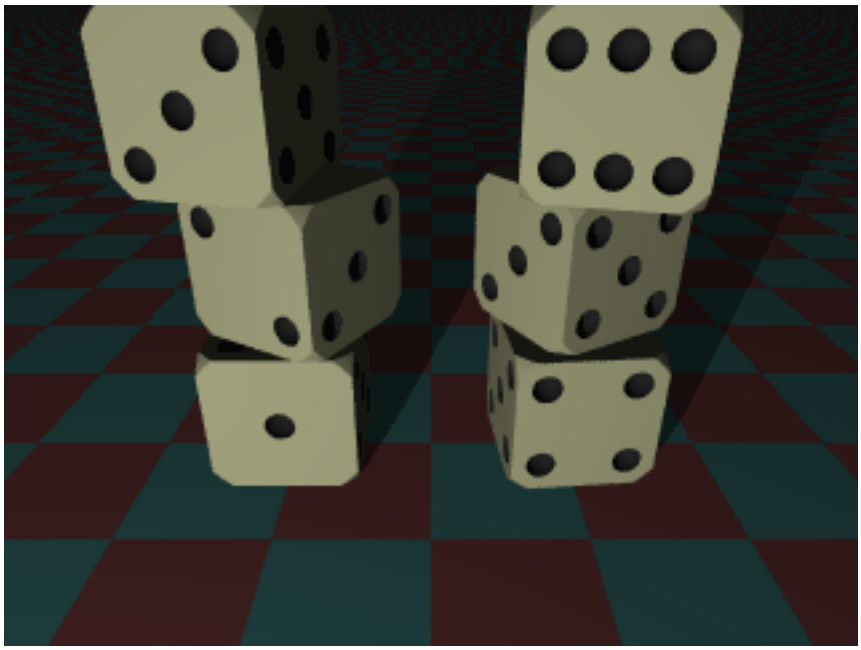

FIG. 2.4 – Plusieurs dés

Notre dé est géométriquement exact. Il lui manque néanmoins la petite touche finale : les textures. Nous verrons celà bien plus tard.

2.2.4 Avant de poursuivre

Avant de poursuivre, faisons le point : Nous maîtrisons les objets *box* et *sphere*
Nous avons compris le fonctionnement de *difference* et *intersection*
Nous savons déclarer un objet avec *#declare* et le réutiliser via *object*.

Si l'un de ces points n'est pas clair, revoyez cette section, ou consultez le manuel de POV-RAY.

Notons au passage que *difference* et *intersection* sont deux exemples de ce qui est appelé *opérations booléennes*. Nous n'avons pas parlé de la plus simple d'entr-elles : l'*union*. Les objets inclus dans un conteneur *union* sont affichés tels quels, sans aucune modification. L'intérêt est de pouvoir gérer cet ensemble comme un seul objet. Nous aurons mille occasions d'utiliser cette opération par la suite.

2.3 Piste de dés

Nous avons un joli dé. Nous savons le dupliquer et nous sommes capables de faire un 421 (avez-vous essayé ?) dans quoi allons nous le lancer , Dans une piste de dés. Alors, qu'est-ce qu'une piste de dés ? C'est un plateau circulaire, pourvu d'un rebord, et recouvert d'une couche de feutre, généralement de couleur verte. Dans cette section, nous nous attaquons à la création du plateau.

2.3.1 Premiers essais

Tout d'abord, reprenez la scène de test et ôtez tout ce qui a rapport au dé. Nous partons d'un fichier propre, il ne doit rester que le strict minimal.

Une piste de dés, c'est un plateau avec un rebord. On peut le voir aussi comme un cylindre trapu, dans lequel on creuse un autre cylindre, qui ne percera pas de part en part le premier. Essayez le code suivant :

L'objet est plus grand qu'un dé. Votre caméra est peut-être mal placée. Changez ses coordonnées afin d'avoir une vue d'ensemble de l'objet. Pensez aussi à déplacer votre suirce de lumière. Une source de lumière à l'intérieur d'un objet éclaire assez mal...

```
difference{
   cylinder {<0,0,0>,<0,1,0>,12}
   cylinder {<0,0.2,0>,<0,1,0>,11}
   texture{T_PLATEAU}
}
```

Résultat : ça ne marche pas. Essayons de comprendre pourquoi. Nous avons un cylindre, centré sur l'origine, de hauteur 1 et de rayon 12. De ce cylindre, nous soustrayons un autre cylindre, centré lui aussi, de rayon plus faible, et de hauteur comprise entre 0,2 et 1. On s'attend donc à le voir creusé dans le premier cylindre, de telle manière que le fond du plateau se trouve à une hauteur comprise entre 0 et 0,2. Alors, où est le problème. Rappelez-vous ce qui a été dit lorsque nous avons commencé à parlé des opérations

booléennes : POV-RAY gêre bizarrement les surfaces confondues. Or, ici, la surface supérieure des deux cylindres est confondue.

Quelle est la solution ? Remonter légèrement la seconde face. Essayez par exemple :

```
difference{
   cylinder {<0,0,0>,<0,1,0>,12}
   cylinder {<0,0.2,0>,<0,1.01,0>,11}
   texture{T_PLATEAU}
}
```

C'est mieux. Maintenant, nous allons mettre de la couleur, en appliquant la texture de feutre à l'intérieur du plateau. Le code devient :

```
difference{
   cylinder {<0,0,0>,<0,1,0>,12 texture{T_PLATEAU} }
   cylinder {<0,0.2,0>,<0,1.01,0>,11 texture {T_FEUTRE} }
}
```

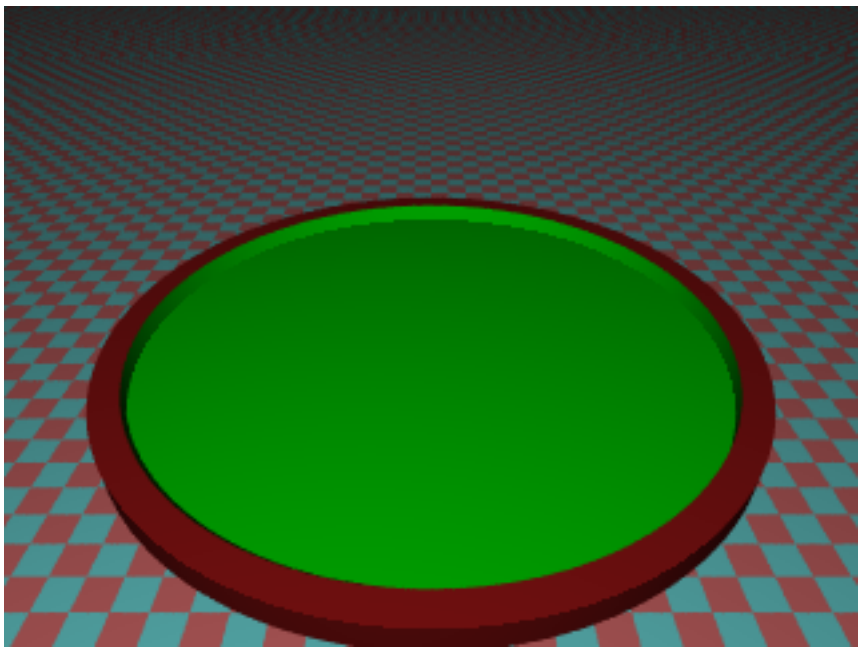

FIG. 2.5 – Piste de dés, première version

2.3.2 Version finale

Une fois que l'on a réussi à faire des dés, ce n'est pas une pauvre petite piste de dés qui va nous embêter. Voyons comment on peut compliquer les

choses. Il pourrait être du plus bel effet d'avoir le dessus de notre piste un peu bombé. comment faire ?

Dans un exemple précédent, nous avions placé une sphère au dessus d'un cylindre, pour donner un cylindre se terminant par une jolie surface courbe. Nous allons utiliser la même idée ici, avec un objet capable de créer une courbe le long de notre piste, c'est-à-dire un tore.

Le mot-clef *torus* crée un tore. Ici, la syntaxe est assez particulière, car il ne demande que deux arguments, un grand et un petit rayon. Nous pouvons imaginer le tore comme un cercle ayant ce grand rayon. Le trait délimitant ce cercle a un trait d'une certaine épaisseur, donnée par ce petit rayon. Le tore ainsi créé est systématiquement centré sur l'origine (il faudra donc le déplacer). Il est contenu dans le plan xz (coup de bol, c'est le bon. Sinon, il aurait fallu le tourner).

Le tore est compris dans le plan xz. Pour le ramener dans le plan yz, une rotation de 90 degrés selon x suffit. Pour le ramener dans le plan xy, appliquez une rotation de 90 degrés selon l'axe z.

Notre tore sera compris entre les rayons 11 et 12. Nous prendrons donc 11,5 comme grand rayon et 0,5 comme petit rayon (comme d'habitude, si vous ne voyez pas pourqoui, faites un dessin).

Essayez donc le code suivant :

```
difference{
   cylinder {<0,0,0>,<0,1,0>,12 texture{T_PLATEAU} }
   cylinder {<0,0.2,0>,<0,1.01,0>,11 texture {T_FEUTRE} }
}
torus {
   11.5,0.5 translate <0,1,0>
   texture {T_PLATEAU}
}
```

C'est pas mal, mais pas parfait. On pourrait préférer un dessus plus plat. C'est faisable, il suffit d'aplatir le tore. alors, disons merci à la fonction *scale* et admirons le résultat.

```
difference{
   cylinder {<0,0,0>,<0,1,0>,12 texture{T_PLATEAU} }
   cylinder {<0,0.2,0>,<0,1.01,0>,11 texture {T_FEUTRE} }
}
torus {
   11.5,0.5 scale <1,0.5,1> translate <0,1,0>
   texture {T_PLATEAU}
}
```

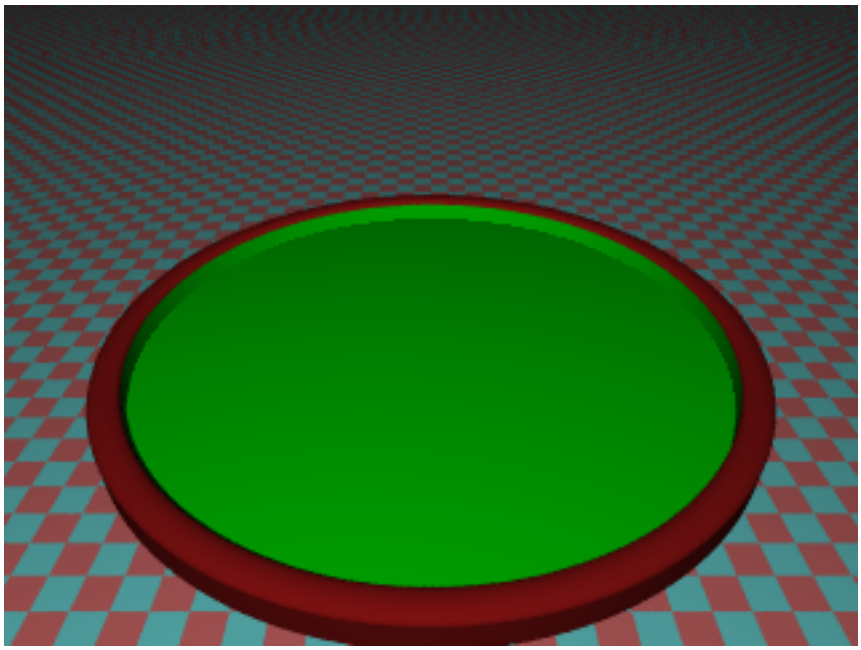

FIG. 2.6 – Piste de dés, version finale

C'est bien, la courbe est discrète, mais elle prendra bien la lumière lorsque nous soignerons les textures. En attendant, nous allons transférer notre objet dans notre fichier final. Il suffit d'ajouter un *#declare*, et...Oui, mais, il y a deux objets ici. Comment les faire tenir à deux dans un seul *#declare* ? Grâce à l'opération booléenne dont nous avons parlé récemment. J'ai nommé l'*union*. Le code à transférer est donc :

```
#declare plateau=union{
   difference{
      cylinder {<0,0,0>,<0,1,0>,12 texture{T_PLATEAU} }
      cylinder {<0,0.2,0>,<0,1.01,0>,11 texture {T_FEUTRE} }
   }
   torus {
      11.5,0.5 scale <1,0.5,1> translate <0,1,0>
      texture {T_PLATEAU}
   }
}
```

Nous avons une partie de nos objets. La table sera représentée par un plan infini. Le verre de whisky sera ajouté plus tard, il nécessitera en effet des objets transparents, ce que nous ne savons pas encore faire.

Mais avant de poursuivre, un petit exercice...

Exercice 15 *Sauriez-vous créer l'objet présent sur la figure 2.7 ?*

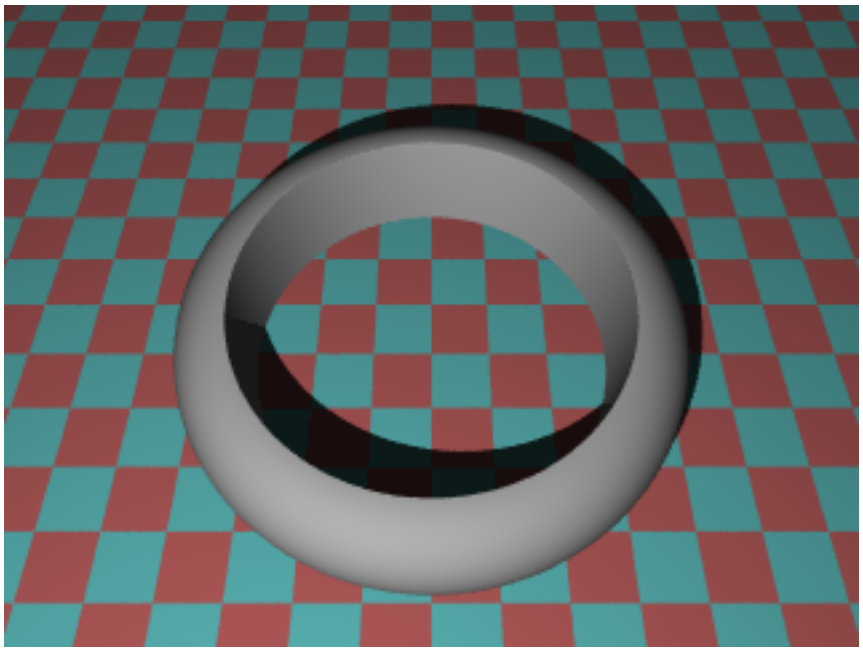

FIG. 2.7 – Un rond de serviette

2.3.3 Avant de poursuivre

Un petit bilan : notre compréhension de POV-RAY progresse petit à petit. Nous avons appris à utiliser un nouvel objet, le *torus*. Nous progressons dans la connaissance des opérations booléennes, et avons notamment vu le piège classique des surfaces confondues. Nous voyons la puissance de cette méthode de modélisation et commençons à voir comment créer des objets complexes à partir de primitives simples.

Exercice 16 *sauriez-vous appliquer ce que nous avons appris ? Créez un moule à baba au rhum, comme présenté à la figure 2.8.*

2.4 La scène

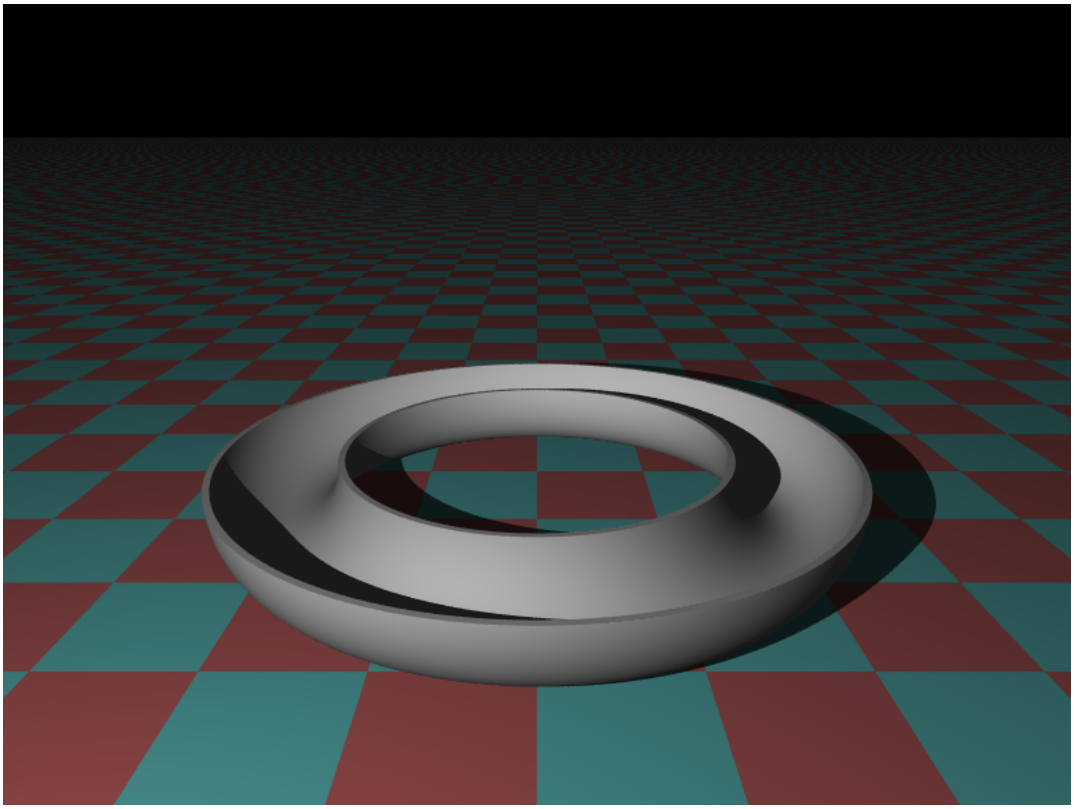

FIG. 2.8 – Un moule à baba

Annexe A

Correction des exercices

Exercice 1 *Remplacez le paragraphe de la sphère par :*

```
sphere{
   <0,0.5,0>,0.5
   pigment {color rgb <0,0,1>}
   finish {reflection 0.2}
}
```

Exercice 2 *Simple. La définition du plan devient :*

```
plane {
   y,0
   pigment {checker color rgb <1,0,0> color rgb<0,1,1>}
   finish {reflection 0.5}
}
```

Exercice 3 *Par exemple :*

```
camera {
   location <0,1,-6>
   look_at <0,0,0>
}
```

Exercice 4 *On note que vert vaut <0,1,0> et magenta <1,0,1>. Supprimez la lumière de la scène d'origine et remplacez-là par le code suivant :*

```
light_source{
   <-10,1,0>
   color rgb <0,1,0>
}
```

```
light_source{
   <10,1,0>
   color rgb <1,0,1>
}
```

L'image que vous obtenez devrait ressembler à celui de la figure A.1. La couleur du plan n'est pas perturbée car il reçoit la lumière des deux sources et la somme des deux éclairages vaut <1,1,1>, donc une lumière blanche. Notez les ombres colorées.

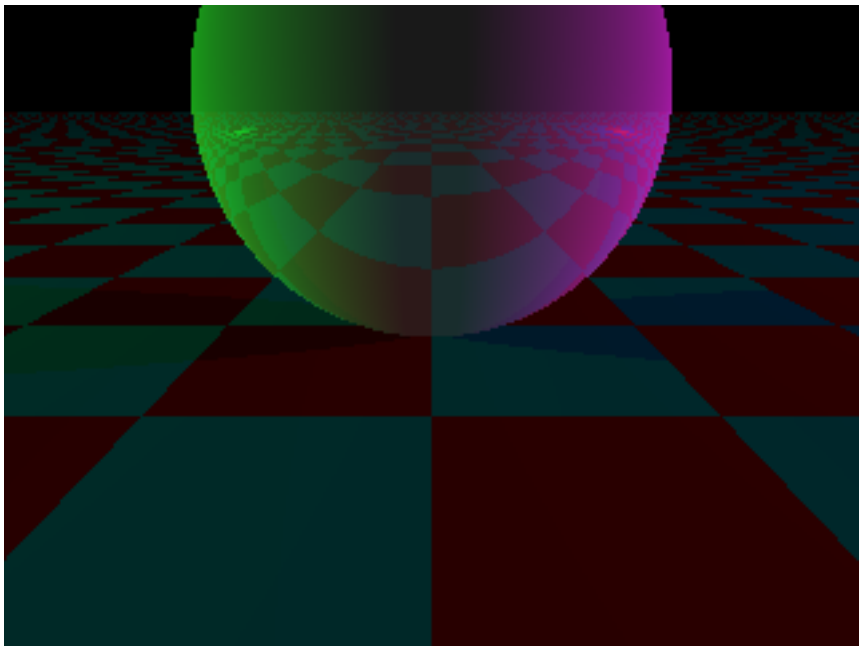

FIG. A.1 – Un exemple d'éclairage coloré

Exercice 5 *L'exemple présenté a été réalisé avec la caméra suivante :*

```
camera {
   location <-3,1,-4>
   look_at <0,0,0>
}
```

Exercice 6 *Allons-y au plus simple :*

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

Exercice 7 *Le code-source du premier cas contiendra :*

```
cylinder {
   <0,0,0>,<0,2,0>,1
   pigment{color rgb <1,1,1>}
}
cone{
   <0,2,0>,1,<0,3,0>,0
   pigment {color rgb <1,1,1>}
}
```

Le second cas donnera :

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

En fait, la sphère est entière, mais l'autre moitié est cachée à l'intérieur du cylindre. Nous apprendrons bientôt à créer de vraies demi-sphères lorsque cela sera nécessaire.

La commande pigment *est commune à tous les objets et il peut paraître fastidieux de la réécrire à chaque fois. Nous verrons très très bientôt comment contourner cette limitation.*

Exercice 8 *La première ligne devient :*

```
#declare BLANC=pigment{color rgb<0,1,0>}
```

Appeler BLANC un pigment vert est curieux, mais pas impossible...

Exercice 9 *Nous allons faire un usage intensif des fonctions vues jusqu'à présent :*

```
#declare CHROME=texture{
   pigment{color rgb<1,1,1>}
   finish{reflection 0.5}
}
#declare boule=sphere{
```

```
   <0,0,0>,1
   texture{CHROME}
}

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

camera {
   location <0,1,-6>
   look_at <0,2,0>
}

light_source{
   <2,1,-3>
   color rgb <1,1,1>
}


object{boule translate <0,1,0>}
object{boule translate <-2,1,0>}
object{boule translate <2,1,0>}
object{boule translate <0,3,0>}
```

Nettement plus compact et lisible, non ?

Exercice 10 *Le début du code est commun avec l'exercice précédent. remplacez juste les quatre objets par :*

```
object{boule translate <0,1,0>}
object{boule scale <0.5,1,1> translate <-1.5,1,0>}
object{boule scale <0.5,1,1> translate <1.5,1,0>}
object{boule scale <1,0.5,1> translate <0,2.5,0>}
```

Notez que le scale *est appliqué avant le* translate*, ce qui permet de garder les objets au centre avant de les déformer. Si vous avez trouvé du premier coup les valeurs numériques pour les translations, bravo ! Sinon, il n'est pas interdit de tatonner un peu, tant que vous aboutissez sur les bonnes valeurs.*

Exercice 11 *On applique deux rotations de 45 degrés selon deux axes différents. Si vous avez du mal à visualiser ce genre de transformation, munissez-vous d'un dé pendant la création de la scène.*

Une fois les deux rotations effectuées, il reste à déplacer le cube à la bonne place. Avez-vous trouvé facilement la distance de translation à appliquer ? C'est $\frac{\sqrt{3}}{2}$*. Un peu de géométrie peut aider. Sinon, on tatonne.*

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

camera {
   location <0,1,-3>
   look_at <0,1,0>
}

light_source{
   <-2,1,-6>
   color rgb <1,1,1>
}

box{
   <-0.5,-0.5,-0.5>,<0.5,0.5,0.5>
   pigment{color rgb <1,1,1>}
   rotate 45*y
   rotate 45*x
   translate 0.866*y
}
```

Avez-vous noté la syntaxe utilisée dans le translate *?*

Exercice 12 *Pas évident celui-ci. On y va en trois étapes : tout d'abord, on déplace l'objet au centre du monde, puis on lui applique la rotation, et enfin, on le replace à sa position d'origine.*

```
translate <-1,0,0>
rotate 45*y
translate <1,0,0>
```

Astuce à retenir, celà peut resservir.

Exercice 13 *Une rotation selon l'axe* y *permet de voir le dé sous différents angles sans le décoller du sol. La translation de 0,5 unités le ramène au niveau du plan.*

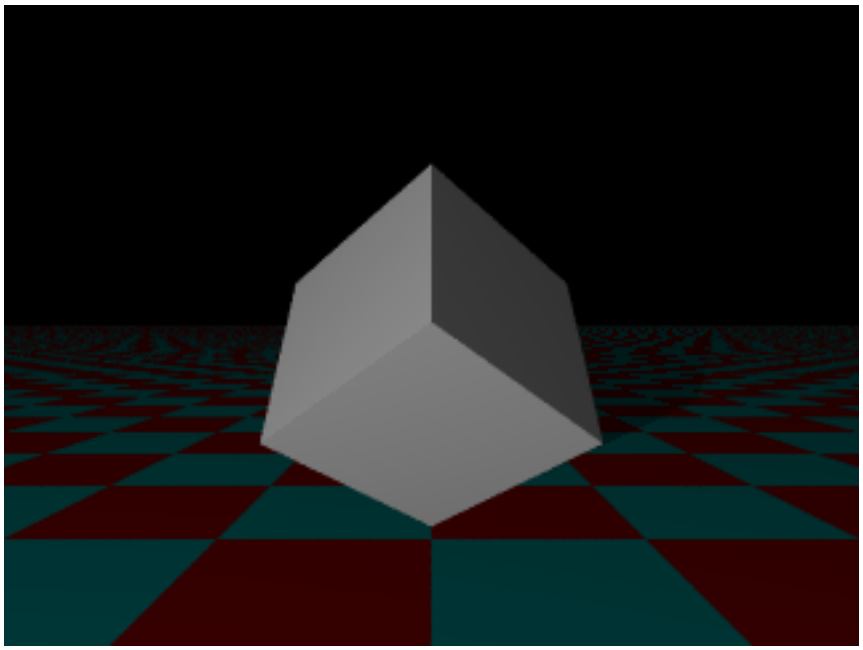

FIG. A.2 – Un cube sur la pointe

Exercice 14 *reprenez la scène de test, collez la définition du dé (n'oubliez pas la définitiondu point), les textures nécessaires, et ajoutez simplement les six lignes suivantes :*

```
object{de rotate -90*x translate <-1,0.5,0>}
object{de rotate -60*y translate <-0.9,1.5,0.1>}
object{de rotate 15*y translate <-1.2,2.5,0>}
object{de rotate 175*y translate <1,0.5,0>}
object{de rotate 60*y translate <1,1.5,0.2>}
object{de rotate 90*x rotate 15*y translate <1.1,2.5,-0.1>}
```

Exercice 15 *C'est un cylindre perçant un tore. Un exemple de code est donné ci-dessous. Notez comment le cylindre dépasse un peu. Et notez la translation pour ramener le tout au-dessus du sol. La figure A.3 montre les deux objets utilisés.*

```
difference {
   torus {3,1}
   cylinder {<0,-1.1,0>,<0,1.1,0>,3}
   pigment{color rgb 1}
   translate <0,1,0>
}
```

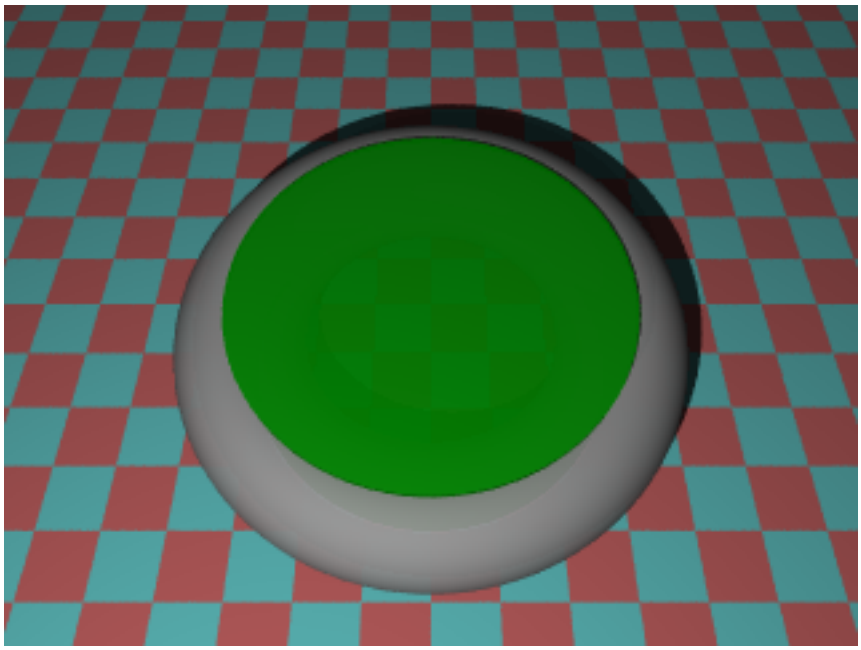

FIG. A.3 – Rond de serviette, objets mis en œuvre

Exercice 16 *Allons-y par étape. Il nous faut un demi tore. Nous l'obtenons par l'intersection entre un tore et une boîte qui englobe le tore et lui arrive à mi-hauteur. Ensuite, nous creusons le tore en faisant la différence entre notre demi-tore et un autre tore de rayon secondaire un peu plus petit. Le résultat semble écrasé. Qu'à celà ne tienne, nous appliquons une mise à l'échelle non uniforme. Il reste à déplacer l'objet pour qu'il soit au dessus du niveau du plan, et à le colorier. Le code-source correspondant est par exemple :*

```
difference{
   intersection{ // un demi tore
      torus {2,0.5}
      box{<-3,0,-3>,<3,-1,3>}
   }
   torus{2,0.45} // que l'on creuse
   scale <1,1.3,1>
   pigment {color rgb <1,1,1>} // blanc
   translate <0,0.65,0> // posé au dessus du plan
}
```