Abordando Geometria através de SOFTWARE TRAÇADOR DE RAIOS

HUGO ALEX DINIZ

editAedi

I Colóquio de Matemática
Região Norte

Sociedade Brasileira de Matemática

Abordando Geometria através de Software Traçador de Raios

Hugo Alex Diniz
Universidade Federal do Oeste do Pará

Prefácio

Este texto não é um tutorial sobre o aplicativo POV-Ray[1]. É apenas uma coleção de idéias de como utilizar esta ferramenta computacional, e ao mesmo tempo abordar diversos tópicos de Geometria. Foi escrito como texto-base de um mini-curso apresentado no I Colóquio de Matemática - Região Norte promovido pela Sociedade Brasileira de Matemática.

Ao orientar trabalhos acadêmicos em Matemática, notei que os alunos tinham dificuldades de produzir gráficos, figuras e imagens de qualidade e inéditos. Em uma destas andanças pela Internet, fui capturado pela beleza e realismo das imagens produzidas através do POV-Ray. Ao descobrir que o mesmo possui uma linguagem interna, na qual os objetos podem ser descritos matematicamente, vislumbrei a possibilidade de utilizá-lo como ferramenta na produção científica e no ensino de Matemática, especialmente Geometria.

Agradeço em especial aos Professores Cristina Vaz, Marcos Diniz e Euna Diniz que me incentivaram a escrever estas experiências.

Santarém - PA, setembro de 2010

Hugo Alex Diniz

[1]Persistence of Vision Raytracer (Version 3.6) [3].

Sumário

Lista de Figuras

Introdução

A importância da descoberta e da experimentação na aprendizagem da Matemática tem sido alvo de muitas pesquisas e discussões desde o último século. Principalmente com o advento dos computadores, criando um novo ambiente (virtual) para experimentação, visualização e criação. No intuito de contribuir nesta direção, apresentamos uma proposta de utilização de uma ferramenta computacional, chamada POV-Ray[2], como motivação e auxílio na aprendizagem de Matemática, especialmente a Geometria.

Não intentamos fazer um tutorial sobre POV-Ray. Até porque existem muito bons tutoriais disponíveis na Internet. Nosso intuito é apresentar um olhar matemático sobre esta ferramenta, que utiliza muita Matemática no processo de criação de imagens belíssimas e com muito realismo. Existem vários bons modeladores para POV-Ray, que são ferramentas gráficas, onde os objetos podem ser posicionados de forma visual. Aqui não utilizaremos estas ferramentas. Entraremos em contato com a *SDL* (Scene Description Language), uma linguagem interna para descrição de cenários, explorando a construção de algoritmos para resolução de problemas.

O POV-Ray é um software gratuito e livre, tornando-se um excelente programa para compor os pacotes de aplicativos das Escolas e Universidades públicas. Ele tem sido utilizado na criação de imagens e filmes educacionais de altíssima qualidade.

[2]Persistence of Vision Raytracer (Version 3.6) [3].

No Capítulo 1, estudamos um exemplo básico com um paralelepípedo, introduzindo os primeiros passos no POV-Ray. No Capítulo 2, apresentamos outras formas básicas como: esfera, cilindro, cone, plano e toro. No Capítulo 3, exploramos operações para gerar novos objetos. No Capítulo 4, estudamos objetos mais complexos em suas descrições matemáticas. No Capítulo 5, apresentamos os *loops*, permitindo a criação de objetos recursivamente, como por exemplo, as aproximações de fractais. Terminamos, no Capítulo 6, utilizando o POV-Ray para gerar sequências de imagens, que podem ser utilizadas na criação de animações e vídeos. Cada capítulo traz atividades envolvendo o assunto do capítulo e vários conhecimentos geométricos.

Para demonstrar o poder deste software, apresento a seguir algumas imagens do "Hall of Fame" do site do POV-Ray. Bom estudo!

Figura 1: Círculos de Villarceau [2]

Figura 2: Parafusos [4]

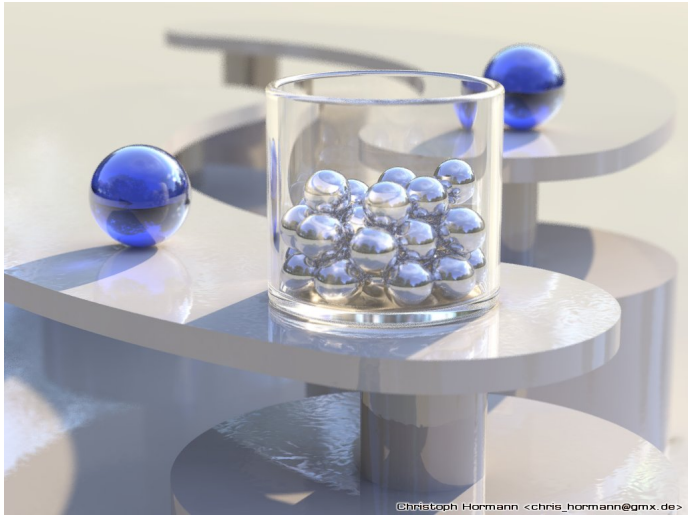

Figura 3: Petecas [1]

Capítulo 1

Introdução ao POV-Ray

POV-Ray é abreviatura de "Persistence of Vision Ray-Tracer", um software capaz de produzir imagens tridimensionais de alta qualidade e realismo, utilizando uma técnica chamada "ray-tracing", que significa "traçado de raios".

Descrevemos através de um arquivo de texto e uma linguagem interna própria chamada **SDL** (Scene Description Language), a posição e a textura dos objetos no cenário, as fontes de luz e a câmera, que faz o papel do observador. A técnica ray-tracing consiste em simular o caminho dos raios de luz no sentido inverso, isto é, do olho do observador (da câmera, de fato) até as fontes de luz. Para cada ponto da imagem a ser formada (em frente à câmera), sai um raio da câmera que passa pelo ponto, com o intuito de determinar a cor deste ponto. Para cada raio é checado se intercepta algum objeto do cenário. Os objetos são definidos matematicamente. Logo essencialmente estamos falando de resolver um sistema de equações para cada objeto e descobrir qual é interceptado primeiro (se houver). Do ponto de interseção, enviamos um raio para cada fonte de luz, a fim de descobrir se este ponto é iluminado. A relação entre os ângulos do raio incidente e do raio para a fonte de luz, em relação ao vetor normal à superfície no ponto, determinará a contribuição que a cor

da superfície terá na cor do pixel. A textura do objeto, o seu grau de difusão da luz e sua capacidade de refletir ou refratar também são considerados.

No caso de uma superfície refletora, enviamos um novo raio a partir do ponto de interseção, obedecendo ao ângulo de incidência em relação ao vetor normal. E repetimos o processo para este novo raio. É limitado o número de reflexões encadeadas. A cada reflexão, diminui a influência deste raio na cor do ponto da imagem que estamos considerando.

1.1 Primeiros Passos

O POV-Ray pode ser obtido no endereço

http://www.povray.org,

em sua versões para as plataformas *Windows*, *Mac* e *Linux*. Para utilizá-lo, é necessário um editor de texto, para gerar um arquivo com a descrição dos objetos, das fontes de luz e da câmera que "fotografa" o objeto. Após isto, deve-se executar o POV-Ray, informando o arquivo de texto (normalmente com a extensão .pov), a fim de ser gerada uma imagem dos objetos descritos, com a iluminação escolhida, a partir do ponto-de-vista da câmera posicionada. Existem algumas opções para o formato da imagem gerada, sendo que o padrão depende da plataforma utilizada. Neste material foi utlizada a versão 3.6 para GNU/Linux.

Vamos ao nosso primeiro exemplo:
cubo.pov

```
#include "colors.inc"

camera {
 location <4, 3, -2.5>
 look_at  <1, 1,  1>
```

```
}
background { color White }
light_source { <3, 4, -2> color White}
box {
 <2, 2, 2>, 0
 texture { pigment { color Blue }}
}
```

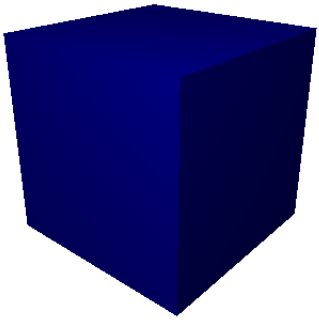

Figura 1.1: Cubo

Vamos analisar linha por linha deste exemplo. É importante salientar que o POV-Ray faz distinção entre maiúsculas e minúsculas, por isso muita atenção na digitação dos comandos.

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

Aqui solicitamos que seja incluído o arquivo com as definições de cores, para que possamos chamá-las pelos nomes. Em inglês! No Apêndice A, você vai encontrar os nomes das cores definidos em **colors.inc**.

```
camera {
 location <4, 3, -2.5>
 look_at  <1, 1,  1>
}
```

Com estes comandos posicionamos a câmera no cenário. Definimos sua localização (location) e a direção (look_at) para qual ela está apontando. Está na hora de falarmos sobre o sistema de coordenadas do POV-Ray.

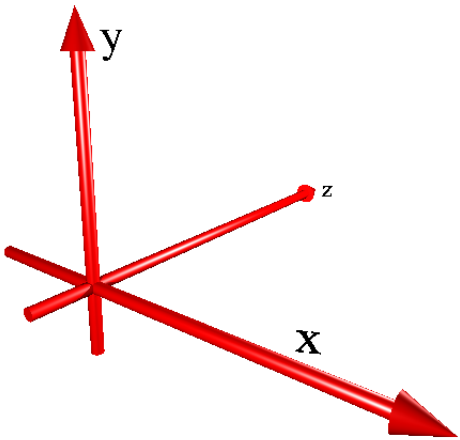

Figura 1.2: Sistema de Coordenadas (Orientação Negativa)

Por padrão, o POV-Ray situa o plano xy no mesmo plano da tela do computador, com o eixo z tendo sentido positivo para "dentro" da tela (se afastando do observador). Esta orientação do espaço é diferente da que utilizamos geralmente em Matemática, que é baseada na "regra da mão direita". Aqui utilizaremos, para a orientação do espaço e para as rotações, a "regra da mão esquerda". Falaremos disto mais tarde.

Um vetor (x, y, z) é representado por `<x, y, z>`. Neste exemplo, posicionamos a câmera no ponto $(4, 3, -2.5)$ e a direcionamos para olhar para o ponto $(1, 1, 1)$.

```
background { color White }
```

Agora definimos a cor do "ambiente". Os raios que partem da câmera e que não tocam nenhum objeto possuem esta cor.

```
light_source { <3, 4, -2> color White}
```

Colocamos uma fonte de luz no ponto $(3, 4, -2)$, com a cor branca.

```
box {
 <2, 2, 2>, 0
 texture { pigment { color Blue }}
}
```

Finalmente, descrevemos um objeto visível no cenário. Como o nome do comando (`box`) sugere, contruímos uma caixa. Para isto determinamos as coordenadas de dois vértices opostos, no exemplo, os pontos $(0, 0, 0)$ (que representamos apenas por `0`) e $(2, 2, 2)$. Na verdade, aqui construímos um cubo. O comando `texture` especifica as característica da superfície do objeto. Aqui especificamos apenas sua cor. Mas é possível colocar texturas que simulam ferro, madeira, pedra e até vidro!

Atividade 1.1 *Com o arquivo* ***cubo.pov****, experimente novos parâmetros. Altere coordenadas e cores. Adicione novas fontes de luz e objetos.*

Atividade 1.2 *Ao longo do eixo z, desenhe quatro cubos enfileirados em ordem crescente de tamanho. Não esqueça de posicionar a câmera adequadamente para uma melhor visualização do cenário.*

Capítulo 2

Formas Básicas

Apresentaremos agora mais algumas formas básicas, com as quais podemos formar cenários mais complexos. Já vimos na seção anterior, como construir um paralelepípedo através do comando `box`.

2.1 Esfera

esfera.pov

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

camera {
 location <3, 3, -3>
 look_at  <0, 1,  0>
}
background { color Gray70 }
light_source { <0, 3, -10> color White}
light_source { <0, 3, 10> color White}
sphere {
 <0, 1, 0>, 1
```

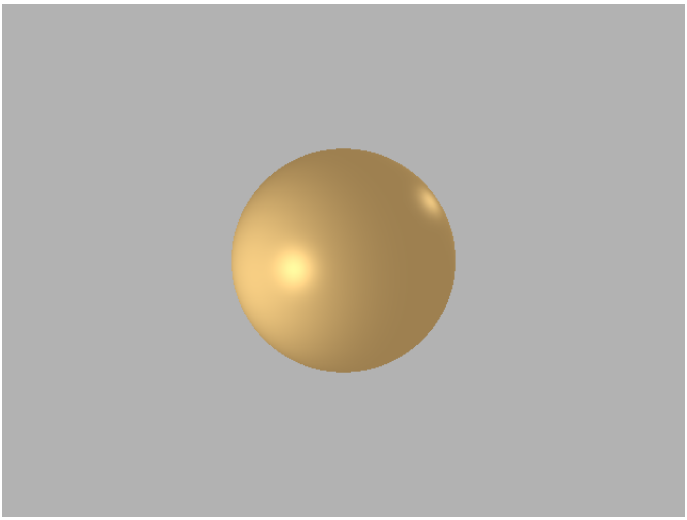

Figura 2.1: Esfera

```
 texture { T_Gold_5B }
}
```

Para o objeto `sphere` precisamos determinar o centro e o raio da esfera, que no nosso exemplo são $(0, 1, 0)$ e 1. Utilizamos outra diretiva `#include`, agora com o arquivo *metals.inc*. É necessária para podermos utilizar a textura `T_Gold_5B`, que nos dá uma aparência metálica. Podemos utilizar os metais Brass, Copper, Chrome e Silver, fazendo o números variarem de 1 a 5 e as letras de A a E. Por exemplo, `T_Chrome_2E`.

2.2 Cilindro

cilindro.pov

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

Figura 2.2: Cilindro

```
#include "woods.inc" //textura de madeira

camera {
 location <3, 3, -4>
 look_at  <0, 0,  0>
}
background { color White }
light_source { <0, 0, -3> color White}
light_source { <5, 5, 0> color White}
cylinder {
 <0, 0, -2>, <0, 0, 3>, 1
 //open

 //você pode comentar o arquivo
 //utilizando duas barras
```

```
 /*
 Ou abrindo e fechando um
 grupo de linhas para comentários.
 É muito importante documentar e
 inserir observações.
 Pode ser muito útil em futuras modificações.
 */
 texture { T_Wood7 }
}
```

Você pode utilizar comentários no seu arquivo de texto. Retire as duas barras que estão em frente à palavra `open`. Isto vai gerar um cilindro sem as "tampas".

2.3 Cone

Figura 2.3: Cone

cone.pov

```
#include "colors.inc"
#include "stones.inc" //textura de pedra

camera {
 location <3, 3, -3>
 look_at  <0, 0,  0>
}
background { color White }
light_source { <0, 0, -3> color White}
light_source { <5, 7, 0> color White}

#declare Tronco = cone {
 <0, 0, 0>, 1.5, <0, 1.8, 0>, 0.5
 open
 texture { T_Stone9 } //experimente de 1 a 44
}

object {Tronco translate y*-2}
object {Tronco rotate x*180 translate <0,2,0>}
```

Com o objeto `cone` podemos construir troncos de cone. Devemos fornecer o centro e o raio da base do cone ($(0,0,0)$ e 1.5, respectivamente), e o centro e o raio do topo ($(0,1.8,0)$ e 0.5, respectivamente). Se quisermos um cone, basta fornecermos 0 como raio. Podemos utilizar o modificador `open` para abrir ou não o interior do cone.

Utilizando a diretiva `#declare` podemos declarar objetos para serem reutilizados. Note os modificadores `rotate` e `translate`.

O `rotate` executa uma rotação ao redor da origem, tomando como parâmetro um vetor (r_x, r_y, r_z), onde r_* indica quantos graus devemos rotacionar em torno do eixo $*$. Em qual sentido ? No sentido dado pela "regra da mão **esquerda**". Isto é, com a mão esquerda, aponte o polegar na mesma direção e sentido positivo do

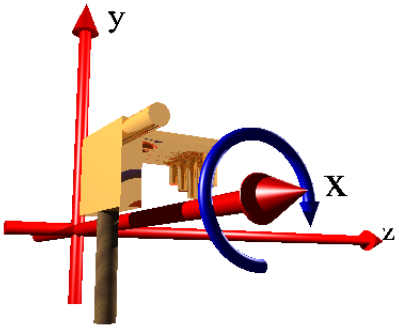

Figura 2.4: Regra da Mão Esquerda

eixo, os outros dedos indicarão o sentido positivo para rotação em torno deste eixo. Mas `rotate` recebeu como parâmetro `x*180`. Isto porque $x = (1, 0, 0)$, $y = (0, 1, 0)$ e $z = (0, 0, 1)$. Logo solicitamos que fosse feita uma rotação de 180° em torno do eixo x, no sentido positivo.

O `translate` executa uma translação do objeto, somando-o ao vetor informado. É possível fazer operações com os vetores, como adição e multiplicação por escalar.

2.4 Plano

plano.pov

```
#include "colors.inc"

camera {
```

```
 location <0, 3, -10>
 look_at  <0, 0,  0>
}
background { color LightBlue }
light_source { <0, 3, 10> color White}
light_source { <0, 3, -10> color White}

plane {
 y, -5
 pigment {brick Gray Blue}
}
```

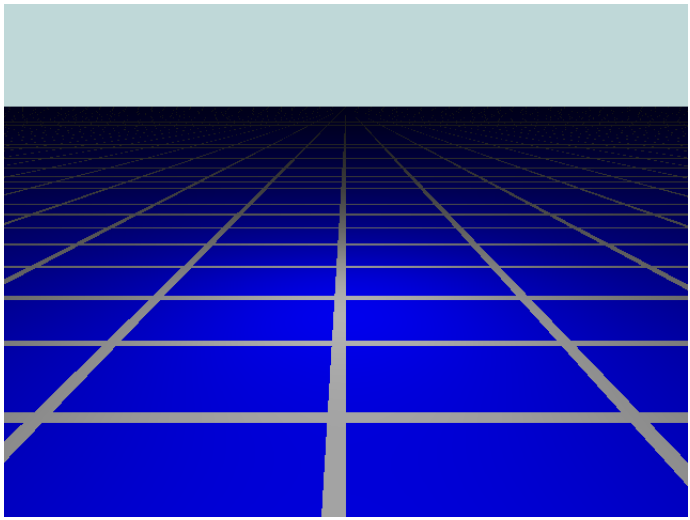

Figura 2.5: Plano

Utilizando `plane` construímos um plano, informando como parâmetros: o vetor normal (neste caso igual $y = (0, 1, 0)$) e a distância para a origem seguindo o sentido do vetor normal (neste caso como a distância é -5 o plano passa pelo ponto $(0, -5, 0)$).

Através do modificador de texturas `pigment` podemos estabeler as cores das superfícies, com cores sólidas ou com padrões de preenchimento, como vimos neste exemplo. Note que não utilizamos o modificador `texture`. Quando na textura tivermos apenas `pigment` podemos suprimir `texture`. Tente outros padrões no lugar de `brick`: `checker` e `hexagon` (é necessário informar uma terceira cor).

2.5 Toro

toro.pov

```
#include "colors.inc"
#include "metals.inc"
#include "glass.inc"

camera {
 location <0, 6, -7>
 look_at  <0, -2, 0>
}
background { color White }
light_source { <0, 5, 0> color White}
light_source { <5, -5, 0> color White}

plane {
 y, -10
 texture {T_Chrome_1A}
}
torus {
 3, 1
 texture {
  F_Glass2
  pigment {Col_Glass_Winebottle}
 }
```

```
}
text {
 ttf "cyrvetic.ttf" "Toro", 1, 0
 rotate 45*x
 translate <-0.8, 0, 0>
 pigment {Col_Aquamarine_01}
}
```

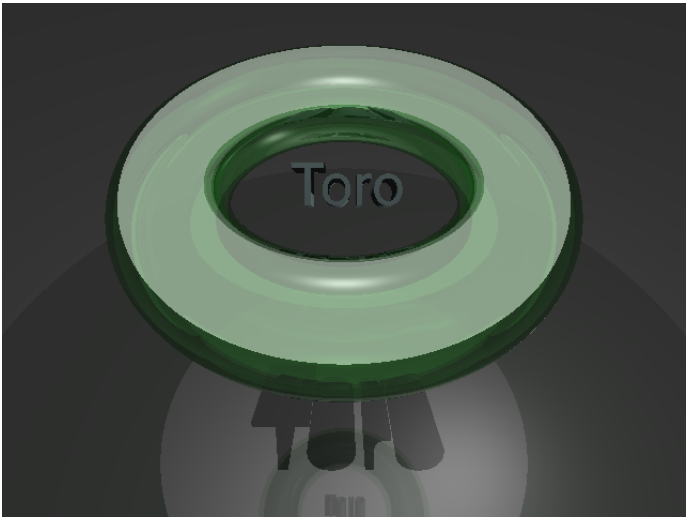

Figura 2.6: Toro

Um toro é gerado pela revolução de uma circunferência ao redor de um eixo coplanar a esta circunferência. Para gerar um toro, utilizamos o objeto `torus`, informando a distância do centro da circunferência para o eixo de rotação e o raio da circunferência. Em outras palavras, se informarmos a e b, a "rosquinha" terá um comprimento de $a + b$, uma altura de b e um buraco de raio $a - b$. Ele sempre será gerado tendo como eixo de rotação o eixo y e como centro a origem. Será necessário utilizar `rotate` e `translate` para reposicioná-lo.

No Apêndice B você vai encontrar as opções disponíveis no arquivo **glass.inc**.

Aqui apresentamos o objeto `text`. Devemos informar o nome da fonte (arquivo **.ttf** ou **.ttc**), o texto a ser gerado, a espessura da letra e o espaçamento adicional entre as letras. No pacote do POV-Ray temos as fontes **crystal.ttf**, **cyrvetic.ttf** e **timrom.ttf**. Para fontes adicionais, devemos informar ao POV-Ray um diretório com as fontes. O texto é escrito sobre o eixo x partindo da origem.

Atividade 2.1 *Desenhe o encontro de dois cilindros, cujos eixos se interceptam perpendicularmente.*

Atividade 2.2 *Desenhe o encontro de três cilindros, cujos eixos são ortogonais entre si.*

Atividade 2.3 *Desenhe seis esferas sobre um plano e posicione-as como na posição inicial das bolas em um jogo de sinuca (forma triangular).*

Atividade 2.4 *Utilizando toros como elos de uma corrente, desenhe uma corrente com 5 elos.*

Atividade 2.5 *Chamamos de link ao entrelaçamento mostrado na Figura 2.7. Vamos ao trabalho ?*

Atividade 2.6 *Arrume a mesa. Utilize os conhecimentos acumulados até agora, e sua criatividade, para gerar uma imagem semelhante à Figura 2.8.*

Figura 2.7: Link

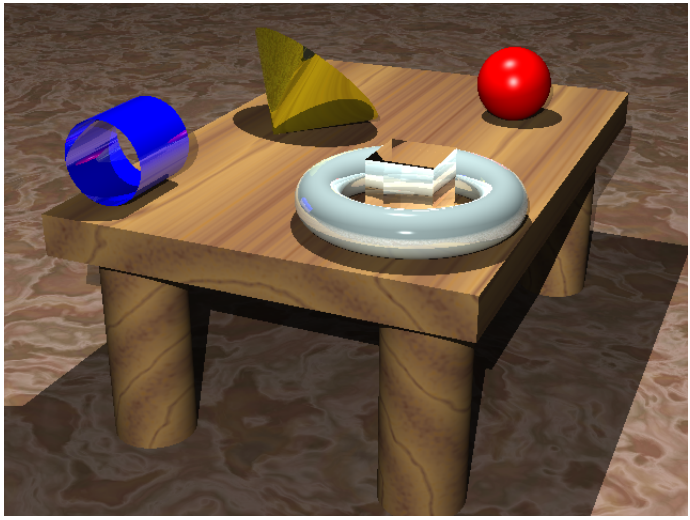

Figura 2.8: Sólidos Geométricos

Capítulo 3

Combinando Formas

Existe uma ferramenta no POV-Ray chamada CSG (Construtive Solid Geometry) que é capaz de contruir sólidos mais complexos, a partir da combinação das formas básicas vistas anteriormente. Podemos combinar sólidos através das seguintes operações: união (`union`), interseção (`intersection`), diferença (`diference`) e fusão (`merge`). Podemos realizar operações envolvendo resultados de operações anteriores.

3.1 União

union.pov

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

camera {
 location <10, 10, -20>
 look_at  <5, 0,  0>
}
light_source { <5, 5, -10> color White}
```

```
light_source { <10, 5, -5> color White}

#declare Esfera = sphere {
0, 0.5 scale <2,1,1> pigment {Red}
}
#declare Barra = box {
 <0, -0.2, -0.2>, <10, 0.2, 0.2> pigment {OldGold}
}

#declare Fosforo = union {
 object {Barra}
 object {Esfera}
 translate x*1
}

union {
 plane {y, 0}
 plane {x, 0}
 plane {z, 0}
 pigment {Bright_Blue_Sky}
}

object {
 Fosforo
 rotate z*-45
 translate <0.1,11.2*cos(pi/4),-2>
}
object {
 Fosforo
 rotate <0, 0, -degrees(atan(0.3/10))>
 translate <2,0.5,-4>
}
object {
```

```
 Fosforo
 rotate <0, 10, -degrees(atan(0.3/10))>
 translate <2,0.5,-6>
}
```

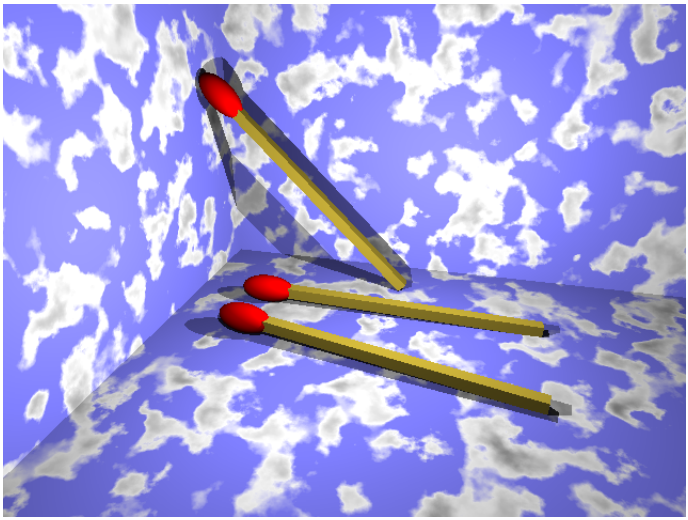

Figura 3.1: União

Podemos unir objetos a fim de que se comportem como se fossem um único objeto. Isto tem vantagem de aplicarmos transformações e texturas de uma única vez no novo objeto.

No exemplo mostrado, temos algumas novidades. Utilizamos o arquivo **textures.inc** (Apêndice C). Introduzimos também o modificador **scale**, que nos permite alterar a escala de um objeto. Por exemplo:

- `scale 5` - o objeto será 5 vezes maior que o original.
- `scale 0.4` - o objeto será 40% do tamanho original.

- `scale <2, 1, 0.3>` - o objeto será "esticado" na direção x, sem alteração na direção y e "encolhido" na direção z.

Assim aplicamos `scale <2, 1, 1>` em uma esfera para transformá-la em um elipsóide. Também utilizamos algumas funções matemáticas. No Apêndice D, listamos as funções matemáticas disponíveis no POV-Ray.

3.2 Interseção e Diferença

Com o comando `intersection` criamos um novo objeto que é a interseção dos objetos informados.

O comando `difference` gera a "subtração" do primeiro objeto pelos objetos seguintes. Podem acontecer coisas estranhas se as superfícies dos objetos coincidirem.

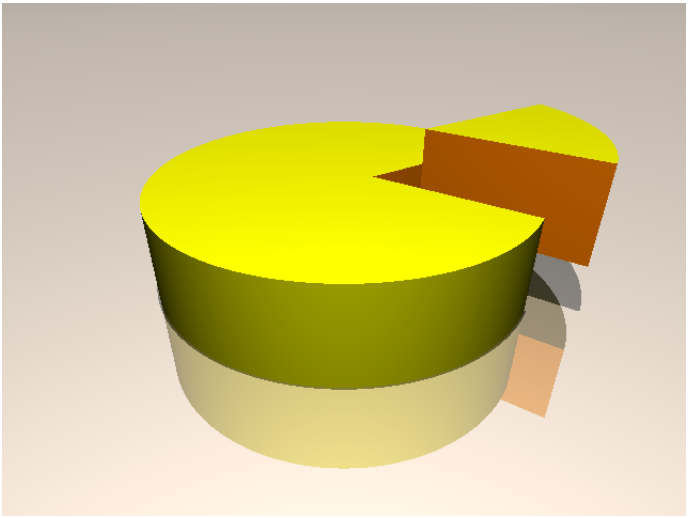

Figura 3.2: Interseção e Diferença

intersection.pov

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

camera {
 location <0, 3, -5>
 look_at <0, 0, 0>
}
background { color Gray }
light_source { <0, 100, 0> White }
light_source { <0, 3, -5> White }

plane {y, 0 texture {Aluminum}}

#declare Queijo = cylinder {
 0, y, 2
 pigment {Yellow}
}

#declare Faca = intersection {
 box {
  <0, 0, 0>, <3, 3, 3>
  rotate 45*y
  translate 0.2*x
  pigment {Orange}
  scale <1.5,1,1>
 }
}

difference {
 object {Queijo}
 object {Faca}
}
```

```
intersection {
 object {Queijo}
 object {Faca}
 translate <0.5, 0.5, 0>
}
```

3.3 Fusão

A diferença entre `union` e `merge` é que o primeiro preserva as superfícies do objetos, mesmo que estas estejam no interior de outros objetos. Quando executamos uma fusão (`merge`) removemos as partes que forem sobrepostas por outro objeto. Para podermos visualizar isto, utilizaremos um pouco de transparência através do modificador de pigmento `filter`, permitindo que uma parte da luz atravesse o objeto.

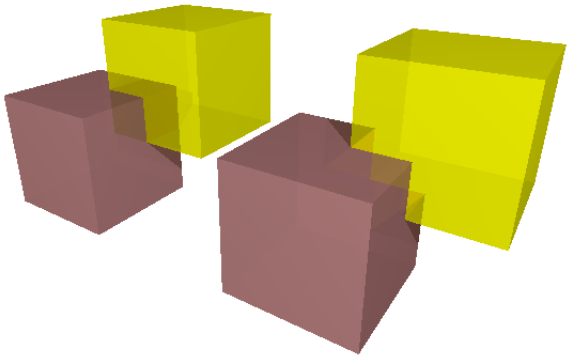

Figura 3.3: União e Fusão

merge.pov

```
#include "colors.inc"

camera {
 location <4, 3, -6>
 look_at  <0, 0,  0>
}
background { color White }
light_source { <10, 10, -20> color White}
light_source { <20, 20, -20> color White}
#declare Caixa1 = box {
 <-0.5, -0.5, -0.5>, <2, 2, 2>
 pigment { color Yellow filter .5}
}

#declare Caixa2 = box {
 <0.5, 0.5, 0.5>, <-2, -2, -2>
 pigment { color Pink filter .5}
}

union {
 object {Caixa1}
 object {Caixa2}
 scale 0.8
 translate -2.5*x
}

merge {
 object {Caixa1}
 object {Caixa2}
 scale 0.8
 translate 2*x
}
```

Atividade 3.1 *Desenhe a interseção de dois cilindros que se interceptam perpendicularmente. Analise a participação de cada cilindro, na superfície deste novo sólido.*

Atividade 3.2 *Desenhe a interseção de três cilindros que se interceptam perpendicularmente entre si. Este sólido tem um nome especial. Humm ?!*

Atividade 3.3 *Desenhe uma clépsidra e uma anti-clépsidra. Humm ?!!!*

Atividade 3.4 *Corte um cone em diferentes ângulos e analise as seções cônicas obtidas.*

Capítulo 4

Formas Avançadas

Há uma quantidade muito grande de objetos pré-definidos no POV-Ray. Dentre os principais podemos citar:

- `blob` - São superfícies de nível de campos escalares que atingem seus máximos nos centros dos componentes e zera nas superfícies dos mesmos. Tem a aparência de um "grude" que pode ligar esferas e cilindros. Muito interessante.
- `isosurface` - Superfícies de nível.
- `julia_fractal` - Fatia 3D de um objeto 4D que generaliza a contrução dos conjuntos de Júlia.
- `lathe` - Superfície de revolução.
- `parametric` - Superfície parametrizada.
- `polygon` - Polígonos.
- `prism` - Prismas e pirâmides.
- `sor` - Superfície de revolução.

- `sphere_sweep` - Sólidos gerados pela translação de uma esfera de raio variável através de uma curva.
- `superellipsoid` - Super-elipsóides (superfícies dadas implicitamente por $(|x|^r + |y|^r)^{\frac{t}{r}} + |z|^t = 1$, com $r, t > 0$).

Veremos em detalhes alguns destes objetos.

4.1 Prisma

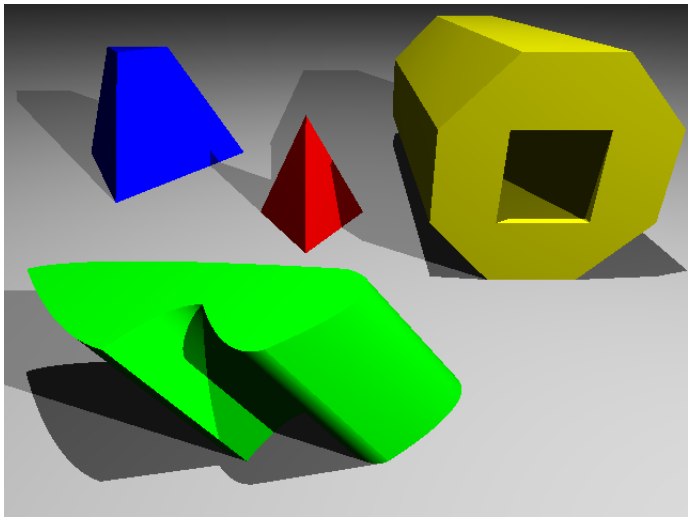

Figura 4.1: Prismas

prisma.pov

```
#include "colors.inc"

camera {
 location <0, 3, -4>
```

```
 look_at  <0, 0.8,  0>
}
background { color White }
light_source { <0, 10, 0> color White}
light_source { <5, 5, -5> color White}

plane {y, 0 pigment {Gray}}

// Pirâmide
prism {
 conic_sweep
 linear_spline
 0, 1.5, 4
 <-0.5, -sqrt(3)/6>, <0.5, -sqrt(3)/6>,
 <0, sqrt(3)/3>,
 <-0.5, -sqrt(3)/6>

 rotate <180, 0, 0>
 translate <-0.5, 1.5, 2.5>
 pigment {Red}
}

// Tronco de pirâmide
prism {
 conic_sweep
 linear_spline
 1, 3, 4
 <-0.5, -sqrt(3)/6>, <0.5, -sqrt(3)/6>,
 <0, sqrt(3)/3>,
 <-0.5, -sqrt(3)/6>

 rotate <180, 0, 0>
 translate <-3.5, 3, 5>
```

```
 pigment {Blue}
}

// Cisalhamento de prima de base spline
prism {
 linear_sweep
 quadratic_spline
 0, 1, 7
 0, <.5, 1>,
 <1, .5>,
 <1.5, 1>
 <2.5, 0>,
 0, <.5, 1>

 matrix <1, 0, 0, 0.8, 1, 0, 0, 0, 1, 0, 0, 0>
 rotate <0, 180, 0>
 translate <0.8, 0, -0.5>
 pigment {Green}
}

// Prisma octagonal com furo
prism {
 linear_sweep
 linear_spline
 0, 5, 14
 0, <1, 0>,
 <1+sqrt(2)/2, sqrt(2)/2>,
 <1+sqrt(2)/2, 1+sqrt(2)/2>,
 <1, 1+sqrt(2)>,
 <0, 1+sqrt(2)>,
 <-sqrt(2)/2, 1+sqrt(2)/2>,
 <-sqrt(2)/2, sqrt(2)/2>,
 0
```

```
 <0, sqrt(2)/2>, <1, sqrt(2)/2>,
 <1, 1+sqrt(2)/2>,
 <0, 1+sqrt(2)/2>
 <0, sqrt(2)/2>

 rotate 90*x
 translate <1.6, 1+sqrt(2), 1>
 pigment {Yellow}
}
```

O objeto `prism` possui várias opções. Primeiramente, podemos escolher entre `linear_sweep`(modo prisma) ou `conic_sweep`(modo pirâmide).

Para desenhar a base, podemos escolher entre `linear_spline` (simplesmente conectar os pontos fornecidos), `quadratic_spline` (spline quadrática - o último ponto fornece informação de inclinação), `cubic_spline` (spline cúbica - o primeiro e o último pontos fornecem informação de inclinação) e `bezier_spline` (spline cúbica - a cada quatro pontos, o segundo e o terceiro fornecem informação de inclinação).

Após estas opções, devemos informar a altura do topo, a altura do fundo e quantos pontos serão fornecidos. Os pontos estarão sobre o plano xz e altura será medida no eixo y. No modo `conic_sweep`, parte-se de um ponto no plano xz, expandindo até a altura desejada, sendo que o desenho da base corresponde à altura $y = 1$.

Introduzimos aqui mais um tipo de transformação. Através do comando `matrix`, fornecemos uma matriz 4×3, que executa a transformação $A(u) = M^t.u + T^t$ onde M são as três primeiras linhas e T é a última linha.

4.2 Superfície de Revolução

revolution.pov

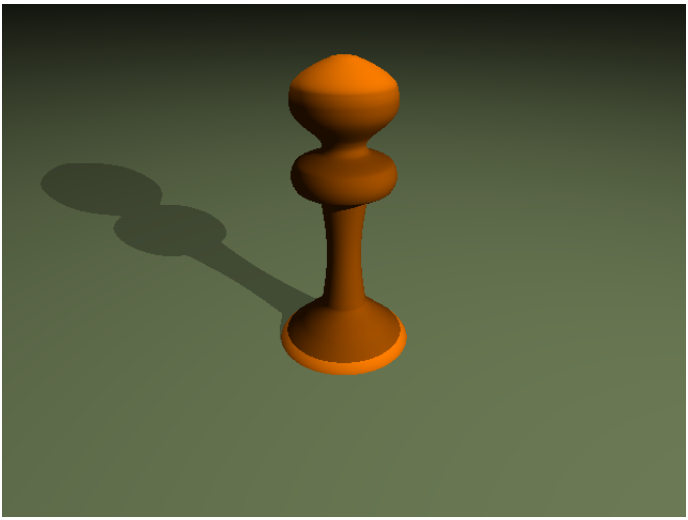

Figura 4.2: Superfície de Revolução

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

camera {
 location <0, 3, -4>
 look_at  <0, 0.8,  0>
}
background { color White }
light_source { <0, 10, 0> color White}
light_source { <5, 5, -5> color White}

plane {y, 0 pigment {Col_Amber_01}}

sor {
 15
 <0.6, -0.2>
```

```
 <0.6, 0>
 <0.4, 0.2>
 <0.2, 0.4>
 <0.2, 1.4>
 <0.4, 1.5>
 <0.4, 1.7>
 <0.2, 1.8>
 <0.2, 1.9>
 <0.4, 2.1>
 <0.4, 2.3>
 <0.3, 2.4>
 <0.1, 2.5>
 <0.05, 2.51>
 <0, 2.52>

 pigment {Orange filter 0.1}
}
```

Podemos utilizar dois objetos para gerar superfícies de revolução: `lathe` e `sor`. O `lathe` permite utilizar splines (como no exemplo anterior) e a curva não precisa ser uma função de x em y. O `sor` (surface of revolution) tem a vantagem de ser mais rápido no processo de geração da imagem (renderização).

4.3 Superfície Parametrizada

parametric.pov

```
#include "colors.inc"
#include "glass.inc"
#include "metals.inc"

camera {
 location <0, 3, -4>
```

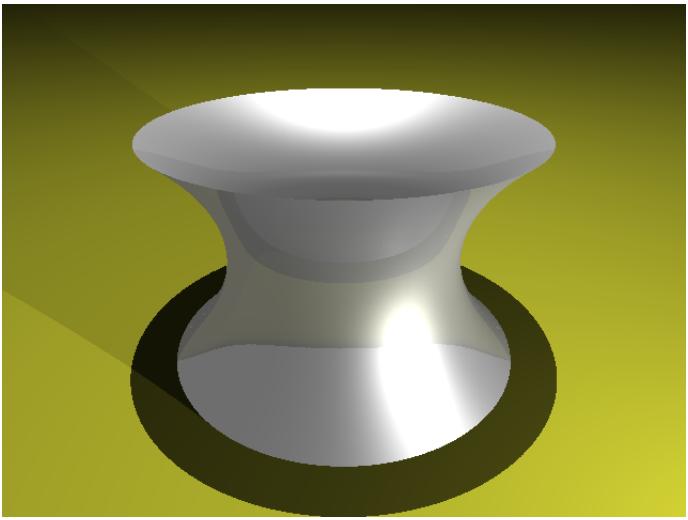

Figura 4.3: Catenóide

```
 look_at  <0, 0.8,  0>
}
background { color White }
light_source { <0, 10, 0> color White}
light_source { <3, 1, -3> color White}

plane {y, 0 pigment {Col_Yellow}}

parametric {
 function { cosh(v)*cos(u) }
 function { v }
 function { cosh(v)*sin(u) }
 <0, -1>, <2*pi, 1>
 contained_by { box {<-2,-2,-2>, <2, 2, 2>}}

 translate y
```

```
 texture {T_Chrome_5A}
}
```

Para o objeto `parametric`, primeiramente fornecemos a parameytrização e depois o intervalo de variação dos parâmetros na forma $(u_0, v_0), (u_1, v_1)$. Necessitamos estabelecer um delimitador, que pode ser uma caixa ou uma esfera.

4.4 Superfície de Nível

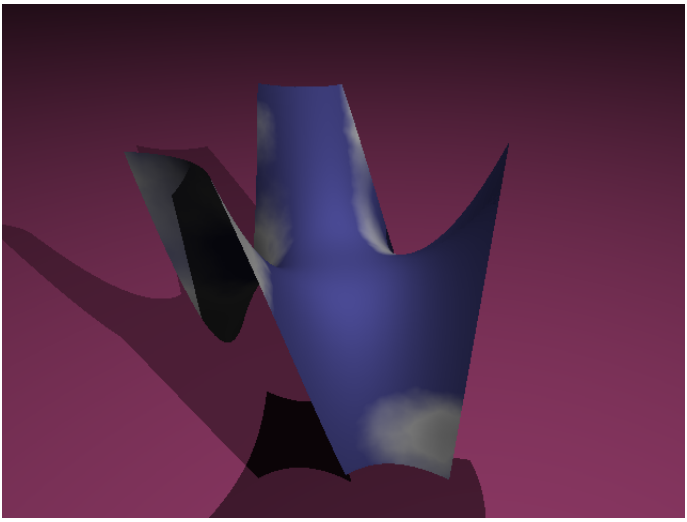

Figura 4.4: Sela de Macaco

isosurface.pov

```
#include "colors.inc"
#include "glass.inc"
#include "textures.inc"
```

```
camera {
 location <1, 2, -3>
 look_at  <0, 0,  0>
}
background { color White }
light_source { <0, 10, 0> color White}
light_source { <3, 3, -3> color White}

plane {y, -3 pigment {Col_Amethyst_01}}

isosurface {
 function { y-pow(x,3)+3*x*pow(z,2)  }
 contained_by { box {-1, 1}}
 max_gradient 5
 open

 texture {Blue_Sky}
}
```

O objeto `isosurface` necessita de uma função, de um delimitador, que pode ser uma caixa ou uma esfera, e de um parâmetro que define o gradiente máximo que será utilizado no algoritmo que determina o ponto de interseção de um raio e a superfície. O modificador `open` retira as bordas do delimitador.

Atividade 4.1 *Desenhe uma esfera inscrita em um tetraedro.*

Atividade 4.2 *Modele uma garrafa com água.*

Atividade 4.3 *Desenhe um helicóide.*

Atividade 4.4 *Desenhe um toro como superfície parametrizada e como uma superfície de nível.*

Capítulo 5

Loops

Através da diretiva `#while` construímos loops, possibilitando a definição de objetos de forma recursiva. As instruções contidas no bloco `#while` (*condição*) ... `#end` são realizadas enquanto *condição* estiver satisfeita. Podemos declarar variáveis através da diretiva `#declare`.
menger.pov

```
#include "colors.inc"

camera {
 location <1.3, 1.5, -1.3>
 look_at <0.5, 0.5, 0.5>
}
light_source { <0.5, 100, 0> color White}
light_source { <50, 0.5, -100> color White}
light_source { <0.5, 0.5, 0.5> color White}

background {White}

#declare Esponja = difference {
 box{ 0, 1 }
 box{ <1/3, 1/3, -0.1>, <2/3, 2/3, 1.1> }
```

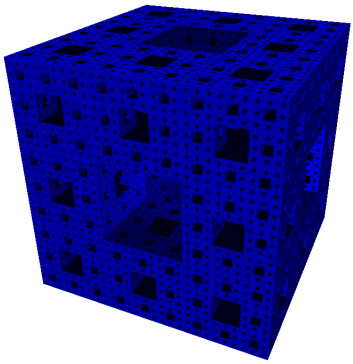

Figura 5.1: Esponja de Menger

```
 box{ <-0.1, 1/3, 1/3>, <1.1, 2/3, 2/3> }
 box{ <1/3, -0.1, 1/3>, <2/3, 1.1, 2/3> }
 texture { pigment {Blue} }
}

#declare num_iteracoes = 3;
#declare shift = array [20] {
 <0,0,0>, <0,0,1>, <0,0,2>,
 <0,1,0>, <0,1,2>,
 <0,2,0>, <0,2,1>, <0,2,2>,
 <1,0,0>, <1,0,2>,
 <1,2,0>, <1,2,2>,
 <2,0,0>, <2,0,1>, <2,0,2>
 <2,1,0>, <2,1,2>,
 <2,2,0>, <2,2,1>, <2,2,2>
}
```

```
#while (num_iteracoes > 0)
 #declare Esponja = union {
  #declare num = 19;
  #while (num >= 0)
    object {Esponja translate shift[num]}
   #declare num = num - 1;
  #end
  scale 1/3
 }
 #declare num_iteracoes = num_iteracoes - 1;
#end

Esponja
```

Atividade 5.1 *Refaça a Atividade 2.5 (link) utilizando um loop.*

Capítulo 6

Animações

Existe uma variável especial, chamada `clock`, que serve para gerar uma sequência de imagens. Por padrão, `clock` varre os valores de 0 a 1, divididos em quantos sub-intervalos for o número de imagens necessárias para a contrução da animação. Para gerar a sequência de imagens, devemos executar o POV-Ray com a opção *+KFFn*, onde n é o número de imagens a serem geradas.
solar.pov

```
#include "colors.inc"

/*
Qualquer semelhança com Sol, Terra e Lua
é mera coincidência
* Não mantivemos a proporção entre Sol e Terra (109:1)
* Não mantivemos a excentricidade
  da órbita terrestre (0.018)
* Nem seguimos as Leis de Cassini
*/

camera {
 location <10, 10, -10>
```

```
 look_at  <0, 0,  0>
}
background { color White }
light_source { <0, 10, 0> color White}
light_source { <5, 5, -5> color White}

sphere {
 0, 2
 texture {
  pigment {Yellow}
 }
}

union {
 sphere {
  0, 1
  texture {
   pigment {checker Brown Blue}
  }
 }
 cylinder {
  -1.5*y, 1.5*y, 0.05
 }
 rotate <0, degrees(clock*2*pi*365), -23.45>
 translate <
  10*cos(clock*2*pi) - 6,
  0,
  8*sin(clock*2*pi)
 >
}

union {
 sphere {
```

```
  0, 0.25
  texture {
   pigment {Gray}
  }
 }
 cylinder {
  -0.375*y, 0.375*y, 0.01
 }
 rotate <0, degrees(clock*2*pi*365/27.3), -6.68>
 translate <
  10*cos(clock*2*pi) - 6 + 1.5*cos(clock*2*pi*365/27.3),
  0,
  8*sin(clock*2*pi) + 1.5*sin(clock*2*pi*365/27.3)
 >
}
```

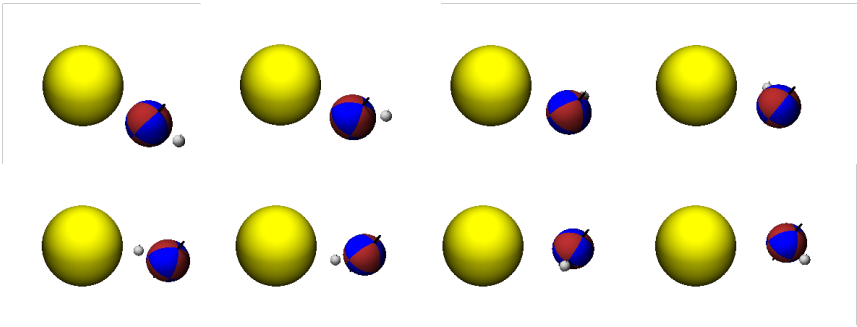

Figura 6.1: Solar

Com um auxílio de um programa que crie um arquivo de vídeo (como por exemplo *ffmpeg*), a partir de uma sequência de imagens, geramos um filme que pode ser utilizado para ilustrar diversas propriedades geométricas.

Atividade 6.1 *Crie uma sequência de imagens que ilustre a forma-*

ção de um sólido de revolução.

Apêndice A

colors.inc

Definições encontradas no arquivo **colors.inc**.

Cores Principais:
Red, Green, Blue, Yellow, Cyan, Magenta, Clear, White, Black.
Tons de Cinza:
Gray##, onde ## é um múltiplo de 5. DimGray, DimGrey, Gray, Grey, LightGray, LightGrey, VLightGray, VLightGrey.
Outras cores:
Aquamarine, BlueViolet, Brown, CadetBlue, Coral, CornflowerBlue, DarkGreen, DarkOliveGreen, DarkOrchid, DarkSlateBlue, DarkSlateGray, DarkSlateGrey, DarkTurquoise, Firebrick, ForestGreen, Gold, Goldenrod, GreenYellow, IndianRed, Khaki, LightBlue, LightSteelBlue, LimeGreen, Maroon, MediumAquamarine, MediumBlue, MediumForestGreen, MediumGoldenrod, MediumOrchid, MediumSeaGreen, MediumSlateBlue, MediumSpringGreen, MediumTurquoise, MediumVioletRed, MidnightBlue, Navy, NavyBlue, Orange, OrangeRed, Orchid, PaleGreen, Pink, Plum, Salmon, SeaGreen, Sienna, SkyBlue, SlateBlue, SpringGreen, SteelBlue, Tan, Thistle, Turquoise, Violet, VioletRed, Wheat, YellowGreen, SummerSky, RichBlue, Brass, Copper, Bronze, Bronze2, Silver, BrightGold,

OldGold, Feldspar, Quartz, Mica, NeonPink, DarkPurple, NeonBlue, CoolCopper, MandarinOrange, LightWood, MediumWood, DarkWood, SpicyPink, SemiSweetChoc, BakersChoc, Flesh, NewTan, NewMidnightBlue, VeryDarkBrown, DarkBrown, DarkTan, GreenCopper, DkGreenCopper, DustyRose, HuntersGreen, Scarlet, Med_Purple, Light_Purple, Very_Light_Purple.

Apêndice B

glass.inc

Definições encontradas no arquivo **glass.inc**.

Cores com Transparência:
Col_Glass_Beerbottle, Col_Glass_Bluish, Col_Glass_Clear,
Col_Glass_Dark_Green, Col_Glass_General, Col_Glass_Green,
Col_Glass_Old, Col_Glass_Orange, Col_Glass_Ruby,
Col_Glass_Vicksbottle, Col_Glass_Winebottle,
Col_Glass_Yellow.
Cores sem Transparência:
Col_Amber_01, Col_Amber_02, Col_Amber_03,
Col_Amber_04, Col_Amber_05, Col_Amber_06,
Col_Amber_07, Col_Amber_08, Col_Amber_09,
Col_Amethyst_01, Col_Amethyst_02, Col_Amethyst_03,
Col_Amethyst_04, Col_Amethyst_05, Col_Amethyst_06,
Col_Apatite_01, Col_Apatite_02, Col_Apatite_03,
Col_Apatite_04, Col_Apatite_05, Col_Aquamarine_01,
Col_Aquamarine_02, Col_Aquamarine_03, Col_Aquamarine_04,
Col_Aquamarine_05, Col_Aquamarine_06, Col_Azurite_01,
Col_Azurite_02, Col_Azurite_03, Col_Azurite_04,
Col_Beerbottle, Col_Blue_01, Col_Blue_02, Col_Blue_03,
Col_Blue_04, Col_Citrine_01, Col_Dark_Green,

Col_Emerald_01, Col_Emerald_02, Col_Emerald_03,
Col_Emerald_04, Col_Emerald_05, Col_Emerald_06,
Col_Emerald_07, Col_Fluorite_01, Col_Fluorite_02,
Col_Fluorite_03, Col_Fluorite_04, Col_Fluorite_05,
Col_Fluorite_06, Col_Fluorite_07, Col_Fluorite_08,
Col_Fluorite_09, Col_Green, Col_Green_01, Col_Green_02,
Col_Green_03, Col_Green_04, Col_Gypsum_01,
Col_Gypsum_02, Col_Gypsum_03, Col_Gypsum_04,
Col_Gypsum_05, Col_Gypsum_06, Col_Orange, Col_Red_01,
Col_Red_02, Col_Red_03, Col_Red_04, Col_Ruby,
Col_Ruby_01, Col_Ruby_02, Col_Ruby_03, Col_Ruby_04,
Col_Ruby_05, Col_Sapphire_01, Col_Sapphire_02,
Col_Sapphire_03, Col_Topaz_01, Col_Topaz_02,
Col_Topaz_03, Col_Tourmaline_01, Col_Tourmaline_02,
Col_Tourmaline_03, Col_Tourmaline_04, Col_Tourmaline_05,
Col_Tourmaline_06, Col_Vicksbottle, Col_Winebottle,
Col_Yellow, Col_Yellow_01, Col_Yellow_02, Col_Yellow_03,
Col_Yellow_04.

Texturas:

T_Glass1, T_Glass2, T_Glass3, T_Glass4, T_Winebottle_Glass,
T_Beerbottle_Glass, T_Ruby_Glass, T_Green_Glass,
T_Dark_Green_Glass, T_Yellow_Glass, T_Orange_Glass,
T_Vicksbottle_Glass, F_Glass1, ..., F_Glass10.

Apêndice C

textures.inc

Definições encontradas no arquivo **textures.inc**.

Pigmentos (pedra): Jade, Red_Marble, White_Marble, Blood_Marble, Blue_Agate, Sapphire_Agate, Brown_Agate, Pink_Granite.
Pigmentos (céu): Blue_Sky, Bright_Blue_Sky, Blue_Sky2, Blue_Sky3, Blood_Sky, Apocalypse, Clouds, FBM_Clouds, Shadow_Clouds.
Pigmentos (madeira): Cherry_Wood, Pine_Wood, Dark_Wood, Tan_Wood, White_Wood, Tom_Wood, DMFWood1, DMFWood2, DMFWood3, DMFWood4, DMFWood5.
Texturas (madeira): DMFWood6, DMFLightOak, DMFDarkOak, EMBWood1, Yellow_Pine, Rosewood, Sandalwood.
Texturas (metal): Metal, SilverFinish, Metallic_Finish, Chrome_Metal, Brass_Metal, Bronze_Metal, Gold_Metal, Silver_Metal, Copper_Metal, Polished_Chrome, Polished_Brass, New_Brass, Spun_Brass, Brushed_Aluminum, Silver1, Silver2, Silver3, Brass_Valley, Rust, Rusty_Iron, Soft_Silver, New_Penny, Tinny_Brass, Gold_Nugget, Aluminum, Bright_Bronze.

Apêndice D

Funções Matemáticas

$abs(x)$ - valor absoluto de x.
$acos(x)$ - arco-cosseno de x (resultado em radianos).
$acosh(x)$ - inversa do cosseno hiperbólico.
$asin(x)$ - arco-cosseno de x (resultado em radianos).
$asinh(x)$ - inversa do seno hiperbólico.
$atan(x)$ - arco-tangente de x (resultado em radianos).
$atan2(x, y)$ - arco-tangente de $\frac{x}{y}$ (resultado em radianos).
$atanh(x)$ - inversa do tangente hiperbólico.
$ceil(x)$ - retorna o menor inteiro maior ou igual a x.
$cos(x)$ - cosseno de x (em radianos).
$cosh(x)$ - cosseno hiperbólico de x.
$degrees(x)$ - converte de radianos para graus.
$div(x, y)$ - parte inteira da divisão de x por y.
$exp(x)$ - exponencial de x.
$floor(x)$ - retorna o maior inteiro menor ou igual a x.
$int(x)$ - parte inteira de x.
$log(x)$ - logaritmo na base 10 de x.
$ln(x)$ - logaritmo natural de x.
$max(x, y, \cdots)$ - retorna o maior valor da lista.
$min(x, y, \cdots)$ - retorna o menor valor da lista.

$mod(a, b)$ - valor de a módulo b.
$pow(x, y)$ - retorna o valor de x(positivo) elevado a y(inteiro).
$radians(x)$ - converte de graus para radianos.
$rand(n)$ - retorna o número pseudo-aleatório relacionado a n pelo comando $seed(n)$.
$seed(n)$ - relaciona um número pseudo-aleatório ao natural n.
$select(x, y, z)$ - retorna y se $x < 0$ e z se $x \geq 0$.
$select(x, y, z, w)$ - retorna y se $x < 0$, z se $x = 0$ e w se $x > 0$.
$sin(x)$ - seno de x (em radianos).
$sinh(x)$ - seno hiperbólico de x.
$sqrt(x)$ - raiz quadrada de x.
$tan(x)$ - tangente de x (em radianos).
$tanh(x)$ - tangente hiperbólico de x.
$vaxis\_rotate(u, v, \theta)$ - retorna o vetor resultado da rotação de θ (em graus) do vetor u em torno do vetor v.
$vcross(u, v)$ - produto vetorial de u e v.
$vdot(u, v)$ - produto escalar dos vetores u e v.
$vnormalize(u)$ - retorna o vetor u normalizado.
$vlength(u)$ - módulo do vetor u.
$vrotate(u, \theta)$ - retorna o vetor resultado da rotação de θ (em graus) do vetor u em torno da origem.

Referências Bibliográficas

[1] **Hormann, C.** *Marbles*. Disponível em http://www.imagico.de/pov/pict/marbles.jpg. Acesso em: 09 de set. de 2010.

[2] **Kristensen, T.** *Villarceau Circles*. Disponível em http://hof.povray.org/Villarceau_Circles-CSG.html. Acesso em: 09 de set. de 2010.

[3] **Persistence of Vision Pty. Ltd.** *Persistence of Vision (TM) Raytracer*. Williamstown, Victoria, Australia. http://www.povray.org/. 2004.

[4] **Piqueres, J.** *Still with Bolts*. Disponível em http://hof.povray.org/boltstill3.html. Acesso em: 09 de set. de 2010.