# Ekstremno brzi uvod u PovRay s primjerima

A. Šiber

# I. Općenitosti

• PovRay – ***P***ersistence ***O***f ***V***ision ***RAY***tracer.

• Dohvatljiv na: www.povray.org

• Verzije za MS Windows OS, Mac OS i Linux OS podržane, postoje i verzije za druge operativne sustave koje se ne razvijaju.

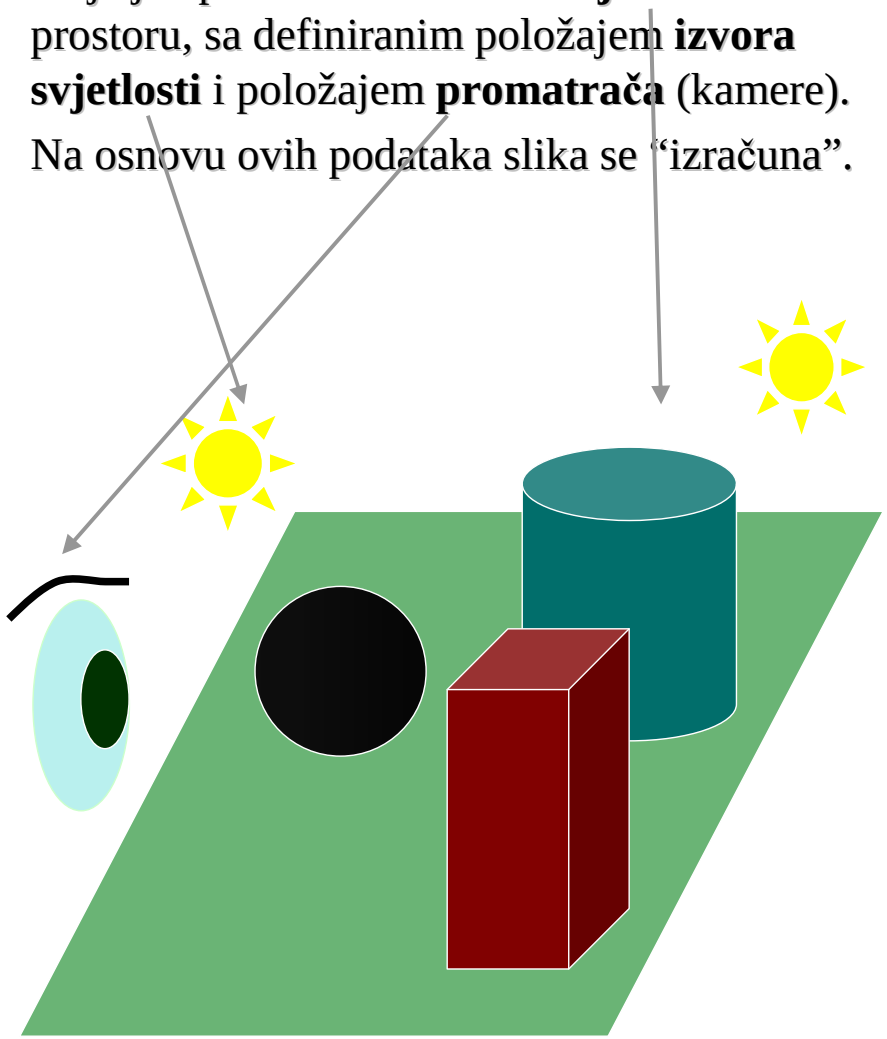
• PovRay predstavlja alat za **"programiranje"** slike: Slika se reprezentira programskim kodom koji prilikom izvršavanja generira sliku u BMP, TGA (Targa) ili nekom od drugih podržanih formata.

• Koristiti se PovRay-em znači naučiti njegov jezik za prikaz objekata, slike.



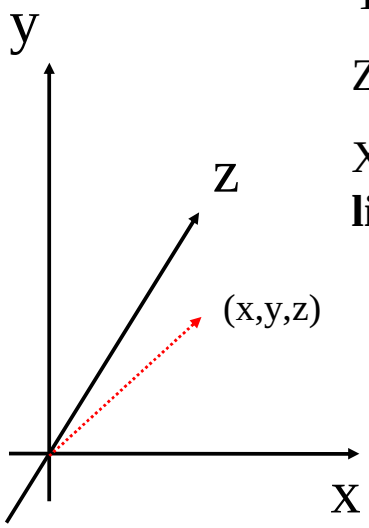
# II. Opis ideje iza Povray-a

- Ideja je opisati sliku kao niz **objekata** u 3D prostoru, sa definiranim položajem **izvora svjetlosti** i položajem **promatrača** (kamere).
- Na osnovu ovih podataka slika se "izračuna".



# III. Koordinatni sustav

- LIJEVI (!) koordinatni sustav




Točka u prostoru opisana sa **3 koordinate**,

**r** = (x,y,z).

Y – uobičajeno "**visina**"

Z – uobičajeno "**dubina**"

X – uobičajeno "**desno-lijevo**"

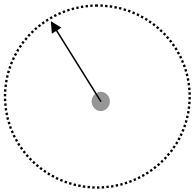
Objekte, kameru, izvore svjetlosti postavljamo u koordinatni sustav specificirajući 3D koordinate koje opisuju objekt.



# IV. Osnovni objekti u Povrayu

- **SFERA:**

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



Centar sfere (3D vektor)

Radijus sfere (skalar)

**Pigment**, boja (color) **površine** kao RGB (red, green, blue) vektor. U ovom slučaju, sfera je plava **Red=0, Green=0, Blue=1**

Općenitu boju prikazujemo kao trojku brojeva: Red, Green, Blue ∈ [0,1].

Tako je npr. yellow boja dana kao:

```
#declare Yellow  = rgb <1,1,0>;
```

**;** nakon **declare** statementa (v 3.5)

Deklaracijski (**declare**) statement u Povrayu

- **CILINDAR:**

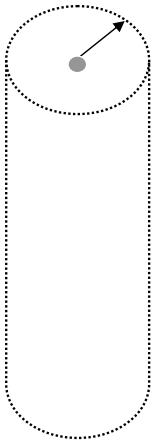
```
cylinder { <0, 1, 0>, <1, 2, 3>, 0.5
   open texture { T_Stone25} scale 4 }
```

Uklonjena vidljivost baza (“otvoreni” cilindar)

Centar jedne baze

Centar druge baze

Radijus baza



Predefinirana (T_Stone25) **tekstura** (prije deklarirana). Kompliciranije od pigmenta koji sadrži samo boju. Teksture mogu sadržavati i opis reflektivnosti površine.

Skaliranje cijelog cilindra za faktor 4 **u sve tri dimenzije**. Ekvivalentno: **scale <4,4,4>**



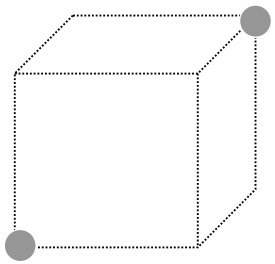
- **KVADAR,** kutija, box:

```
box { <-1, 0, -1>, < 1, 0.5, 3> texture {
   T_Stone25 scale 4} rotate <0,20,0>}
```

“Donji lijevi” vrh

“Gornji desni” vrh

**scale** unutar **texture** statementa, skalira se tekstura a ne objekt !



Rotacija (**rotate**) oko y-osi za 20 stupnjeva (degrees).

Rotacija objekta oko z-osi za 45 stupnjeva bila bi dana kao rotate <0,0,45>. **Rotacija je uvijek oko vektora kojemu je ishodište u <0,0,0>** !

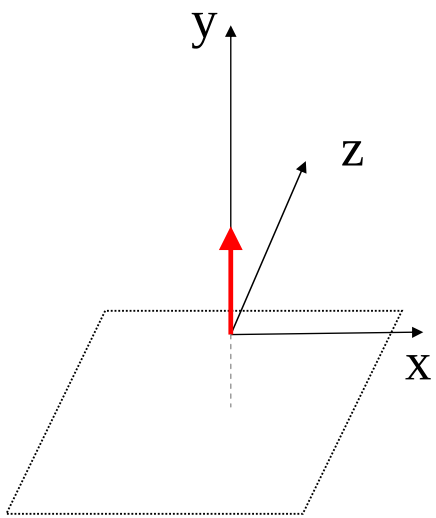
- **RAVNINA**, poluprostor, plane:

```
plane {<0, 1, 0>,-1 pigment {color
   rgb<0.5,0,0.3>}}
```

Vektor normale (okomice) ravnine

Pomak površine **duž vektora okomice** s obzirom na iskodište. U ovom slučaju, ravnina je udaljena od y-osi (vektor okomice <0,1,0>) za -1.

"Ravnina" u Povray-u je poluprostor ograničen ravninom !





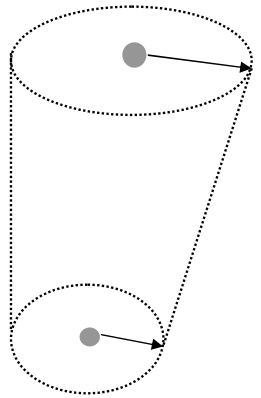

- **KONUS** (odrezani):

```
cone { <0, 1, 0>, 0.3
<1, 2, 3>, 1.0
texture {Dark_Wood} }
```

Centar "donje" baze (<0,1,0>) i radijus donje baze (0.3)

Centar "gornje" baze (<1,2,3>) i radijus gornje baze (1.0)

Predefinirana tekstura (npr **#declare** statementom ili uvezena iz nekog definicijskog file-a, tzv **include** fileovi)



Slično kao i cilindar, samo sa specifikacijom **dva** radijusa baza. Ako "gornji" (ili "donji") radijus pustimo u nulu (0), dobivamo oštri konus. Moguće je koristiti **open** ključnu riječ koja čini baze nevidljivim (isto kao i kod cilindra).

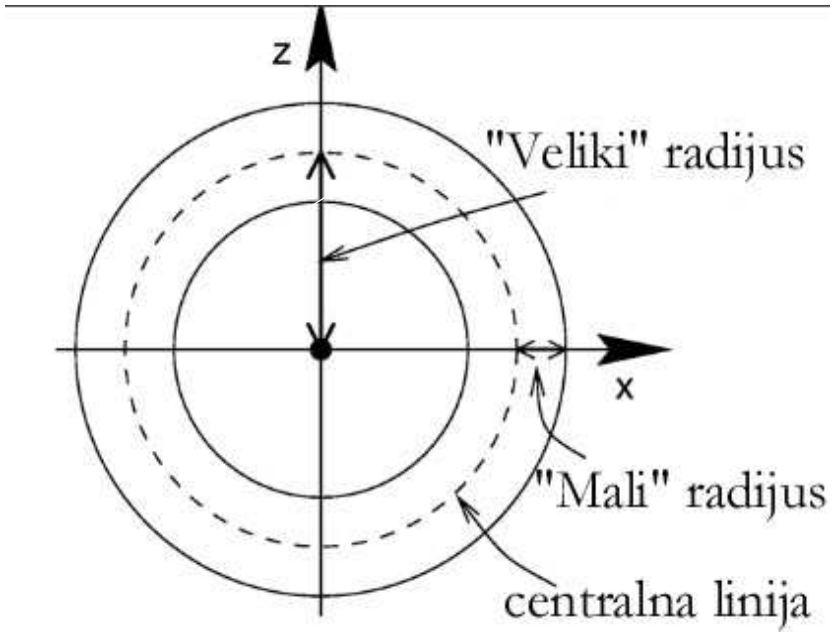
- **TORUS:**

```
torus {
4, 1
rotate <-90,0,0>
pigment { Green }
}
```

"Veliki" i "mali" radijus

Rotacija oko x-osi za -90 stupnjeva tako da nakon te transformacije torus leži u x-y ravnini.

Torus bez specificiranih geometrijskih transformacija (npr. **rotate** kao u primjeru gore) uvijek leži u x-z ravnini !

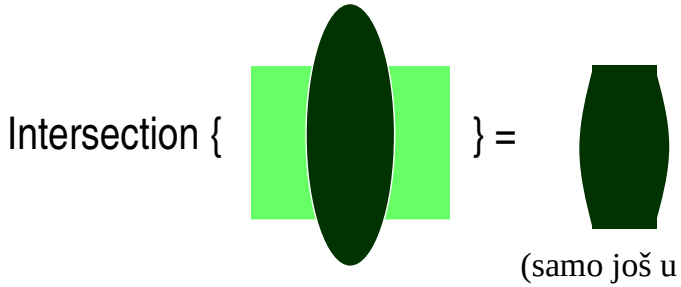





# V. CSG koncept (ili skupovne operacije s objektima)

- CSG = Constructive Solid Geometry

  **Osnovna ideja:** reprezentirati kompleksnije objekte kao presjek (**intersection**), uniju (**union**) ili razliku (**difference**) jednostavnijih objekata. CSG radi i s prije definiranim objektima koji također mogu biti iskonstruirani CSG-om.




(samo još u 3D ☺)

- **UNIJA (UNION):**

```
union{
sphere { <0, 0, 0>, 1
pigment { Blue }
translate <-0.5,0,0>}
sphere { <0, 0, 0>, 1
pigment { Red }
translate <0.5,0,0>}
}
```

**Translate** transformacija translatira objekt duž 3D vektora (u ovom slučaju <-0.5,0,0>). Slično kao i rotate, specificira se unutar zagrada {} u kojima se nalazi definicija objekta (u ovom slučaju sphere).

Predefinirana boja (Red), npr #declare Red = rgb <0.87,0,0>; ili uvezena iz include file-a. Ukoliko se definicije "uvoze" iz include (.inc) fileova, ti definicijski fileovi se moraju specificirati na početku koda:

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

Ascii file sa definicijama (declare statementima i slično).



- **PRESJEK (INTERSECTION):**

```
intersection {
sphere {<0, 0, 0>, 1 translate <-0.5,0,0>}
sphere {<0, 0, 0>, 1 translate <0.5,0,0>}
pigment { Red }
}
```

U ovom slučaju, pojedini objekti unutar **intersection** statementa nemaju definirane boje (ili teksturu u općenitom slucaju), ali presjek (kao **novi objekt**) ima crvenu boju.

- **RAZLIKA (DIFFERENCE):**

```
difference {
intersection {
sphere { <0, 0, 0>, 1 translate <-0.5,0,0> }
sphere { <0, 0, 0>, 1 translate <0.5,0,0> }
pigment { Red } rotate <0,90,0>
}
cylinder { <0, 0, -1>, <0, 0, 1>, .35
pigment { Blue }
}
}
```

Kraj **intersection** objekta.

Rotacija intersection objekta.

Kraj **cilindra**.

Kraj **difference** objekta.

Primjer kompleksnije **difference** operacije koja kreira objekt koji je razlika između CSG intersection objekta i cilindra.



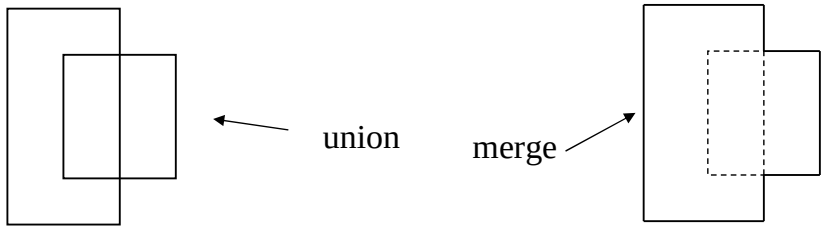
- **MERGE**

```
merge {
object {Lens_With_Hole translate <  .65, .65, 0> }
object {Lens_With_Hole translate <.65, .65, 0> }
object {Lens_With_Hole translate <  .65,-  .65, 0> }
object {Lens_With_Hole translate <.65,-  .65, 0> }
pigment { Red }
}
```

Predefinirani objekt, npr:

```
#declare
Lens_With_Hole =
object {
union{ … }
}
```

Isto kao i **union** operacija, ali uklanja **unutarnje površine,** što unija **ne radi** !



# VI. Izvori svjetlosti

- **TOČKASTI IZVOR** (pointlight source)**:**

```
light_source { <2, 10, -3> color White }
```

Ključna riječ

Pozicija izvora svjetlosti (koordinata)

Boja izvora svjetlosti (rgb vektor)

Izvor svjetlosti je općenito nevidljiv, ali se njegov utjecaj vidi na objektima i sjenama u "sceni".



- **SPOTLIGHT IZVOR** (lampa):

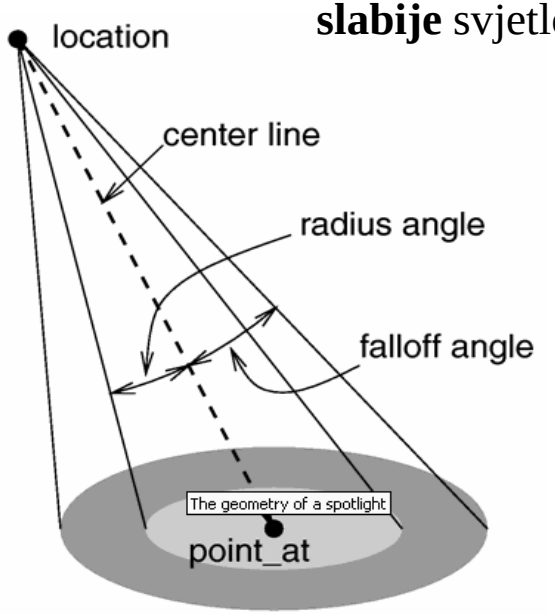
```
light_source {
<0, 10, -3>
color White
spotlight
radius 15
falloff 20
tightness 10
point_at <0, 0, 0>
}
```

Pozicija izvora svjetlosti

Ključna rijec

Kutevi (vidi sliku)

Definira način prijelaza iz **jakog** u **slabije** svjetlo.



# VII. Kamera, kreni !

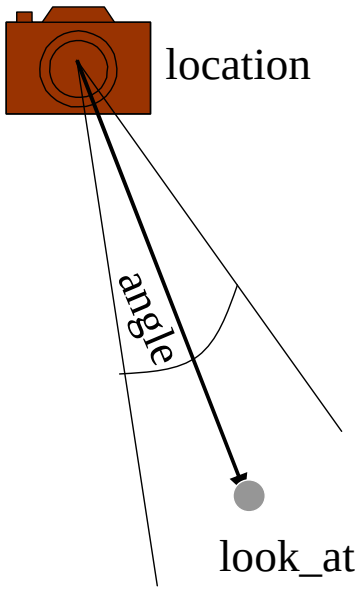
```
camera {
perspective
angle 55
location
   <3,5,-10>
look_at
   <0,2,1>
}
```

Tip “leće” odn. vrsta 3D projekcije. Druge mogućnosti: ortographic, spherical, ultra_wide_angle …

Vidni kut kamere

Pozicija kamere

Točka u koju kamera gleda






# VIII. Moja prva PovRay slika !

**(sphere, cylinder, torus, cone, plane, object, union, merge, light_source, camera, translate)**

Oznaka komentara u PovRay-u (//)

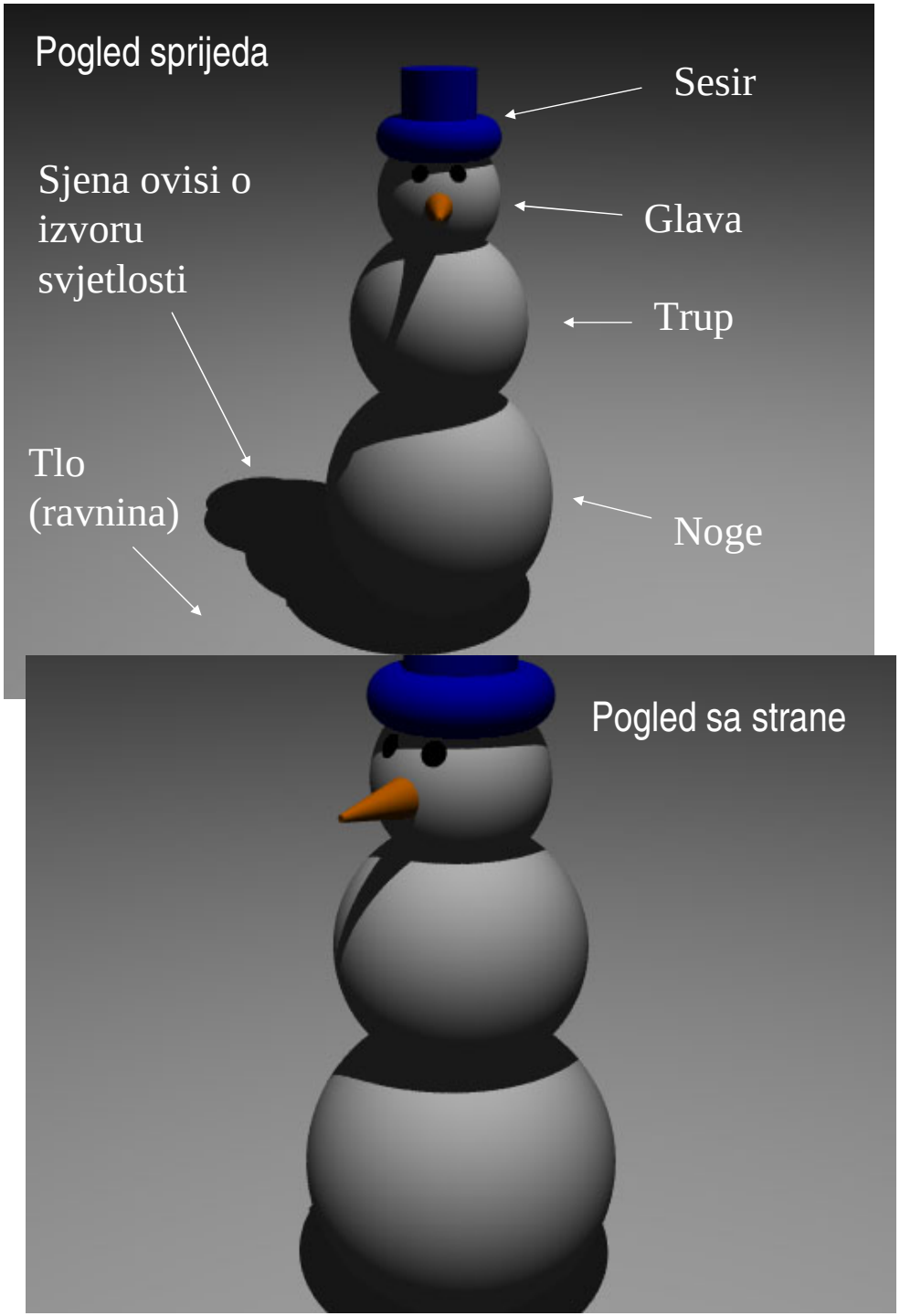
```
camera {
angle 45
location  <0.0 , 10.0 ,-17.0> // pogled sprijeda
// location  <7.0 , 10.0 ,-9.0> // pogled sa strane
look_at   <0.0 , 4.5 , 0.0> }

light_source{
<5.0, 20.0, -5.0>
color rgb<1,1,1>
}

plane{<0,1,0>, 0 pigment{color rgb<0.9,0.9,0.9>}} // tlo

#declare Glava =
union {
sphere{<0,0,0>, 1 pigment{color rgb<1,1,1>}} // glava
sphere{<-0.25,0.4,-0.7>, 0.2 pigment{color rgb<0,0,0>}} // lijevo oko
sphere{<0.25,0.4,-0.7>, 0.2 pigment{color rgb<0,0,0>}}  // desno oko
cone{<0,0,0>, 0.4 <0,0,-1.9>, 0.04 pigment{color rgb<1,0.5,0>}} // nos
};
```

```
#declare Trup =
sphere{<0,0,0>, 1.5 pigment{color rgb<1,1,1>}};

#declare Noge =
sphere{<0,0,0>, 2.0 pigment{color rgb<1,1,1>}};

#declare Sesir =
merge{
cylinder{<0,0,0>,<0,1,0>, 0.6 pigment{color rgb<0,0,0.9>}} //cilindar
    sesira
torus{0.7,0.3 pigment{color rgb<0,0,0.9>}} //obod sesira
};

#declare Snjesko =
union{
object{Sesir translate <0,7.9,0>}
object{Glava translate <0,7,0>}
object{Trup translate <0,5.0,0>}
object{Noge translate <0,2,0>}
};

object{Snjesko}
```

Translate operacijama pomičem Sesir na Glavu, Glavu na Trup, Trup na Noge, a Noge na Tlo, sve to unutar novog objekta Snjesko.

Ovo znači “prikaži objekt Snjesko”.





# IX. Još objekata

- **SPHERE_SWEEP OBJEKT:**

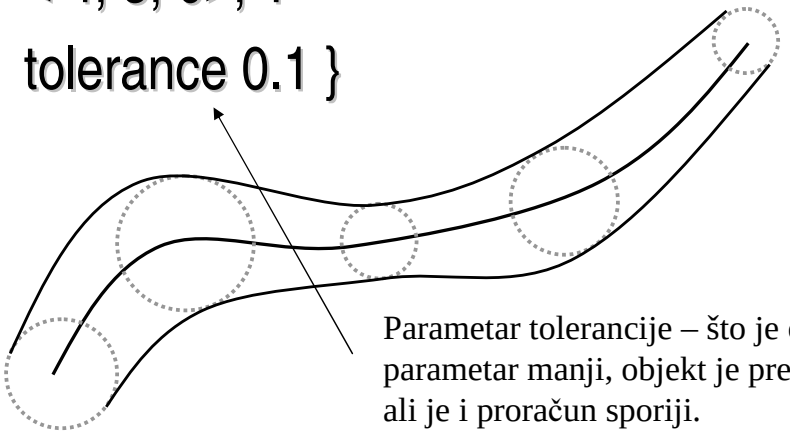
```
sphere_sweep {
cubic_spline
6,
<-4, -5, 0>, 1
<-5, -5, 0>, 1
<-5, 5, 0>, 0.5
< 5, -5, 0>, 0.5
< 5, 5, 0>, 1
< 4, 5, 0>, 1
tolerance 0.1 }
```

Objekt koji nastaje gibanjem sfere (moguće promjenjivog radijusa) po zadanoj krivulji

Vrsta krivulje (alternativa: linear_spline, b_spline)

Broj točaka krivulje

Koordinata, radijus sfere u točki (koordinati)



Parametar tolerancije – što je ovaj parametar manji, objekt je precizniji, ali je i proračun sporiji.



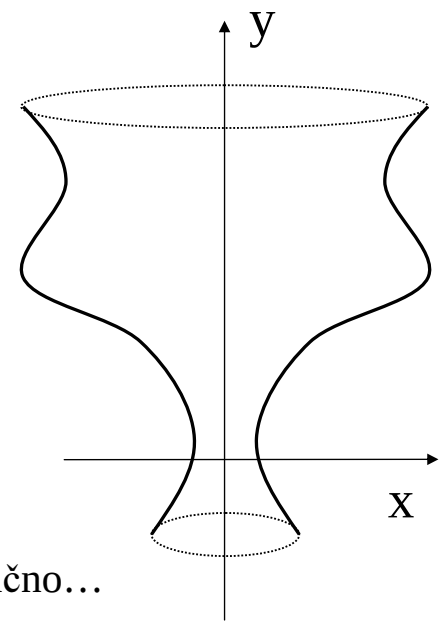
- **SURFACE OF REVOLUTION**

```
sor {
8,
<0.0, -0.5>,
<3.0, 0.0>,
<1.0, 0.2>,
<0.5, 0.4>,
<0.5, 4.0>,
<1.0, 5.0>,
<3.0, 10.0>,
<4.0, 11.0>
texture {T_Gold_1B}
}
```

Objekt koji nastaje rotacijom krivulje u x-y ravnini oko y-osi.

Broj točaka krivulje

(x,y) koordinate pojedine točke krivulje - možemo razmišljati i u terminima (radijus, visina).




Odlično za lonce, vaze i slično…

- **PRIZMA** (prism)

```
prism {
linear_sweep
linear_spline
0,
1,
7,
<3,5>, <-3,5>, <-5,0>, <-3,-5>, <3, -5>,
   <5,0>, <3,5>
pigment { Green }
}
```

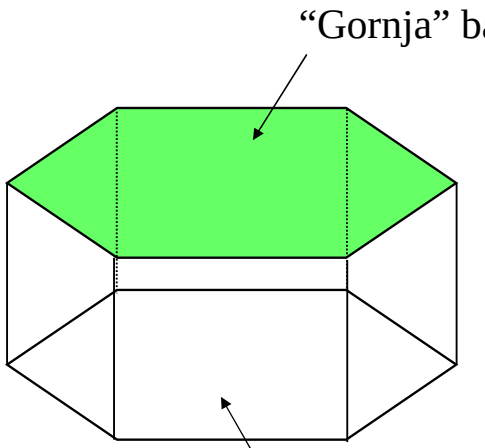
Vrsta krivulje koja opisuje bazu prizme (obrubnu liniju baze). Alternativa: cubic_spline

Y-koordinata "donje" baze prizme

Y-koordinata "gornje" baze prizme

Broj točaka koje definiraju krivulju koja opisuje bazu prizme

Točke krivulje baze: (X,Z) koordinate



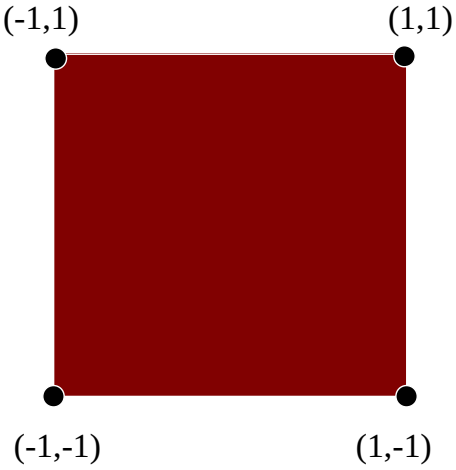



- **POLIGON** (polygon)

```
polygon { 4,
<-1, -1>,
<1, -1>,
<1, 1>,
<-1, 1>
pigment { color rgb <1, 0, 0> }
}
```

Broj točaka poligona

(x,y) koordinate pojedine točke poligona (z=0).




Ovo je u stvari 2D objekt koji možemo smatrati dijelom ravnine. Rotacijama, translacijama i slično, možemo ovaj 2D objekt pomaknuti iz x-y ravnine.

- **EKVI-PLOHA** (isosurface)

```
isosurface {
function { x }
contained_by {
box { <-2,-2,-2>, <2,2,2> }
}
open
}
```

Funkcija koja definira ekvi-plohu, funkcija kojoj tražimo nultočke

Objekt koji sadržava ekvi-plohu (unutar kojeg se proračun nul-točke funkcije provodi).

Ključnom rječju open možemo maknuti vidljivost objekta (containing objekta) koji sadržava isosurface objekt.

U gornjem slučaju, x=0 zadovoljeno je na y-z ravnini pa je rezultat **dio** y-z ravnine sadržan unutar contained_by objekta tj. kvadrat:


*Isosurface sample (function { x }, open)*



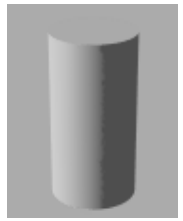
Mijenjanjem funkcije moguće je dobiti svašta. Npr. zamijenimo li function{x} sa

function { abs(x)+abs(y)+abs(z)-2 } dobivamo:

Koristeći:

function { sqrt(pow(x,2) + pow(z,2)) - 1 }

dobivamo cilindar (desno):



sqrt – drugi korijen

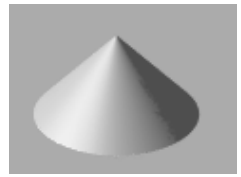
Potenciranje (power):

abs – modul, apsolutna vrijednost

pow(z,2)=z*z

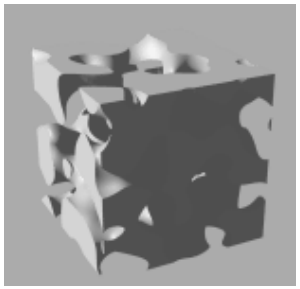
pow(z,5)=z*z*z*z*z

function { sqrt(pow(x,2) + pow(z,2)) + y } →



PovRay ima mnoštvo predefiniranih funkcija (moguće je naravno definirati i svoje). Na primjer:

function { f_noise3d(x, y, z)-0.5 } →



Premda je vrlo moćan, isosurface objekt se sporo računa!

# X. Jednostavne teksture

- **FINISH**

```
sphere { <0, 1, 2>, 2
texture {
pigment { color rgb <0,1,0> }
finish { phong 0.9 phong_size 60 }
}
}
```

**phong** je ključna riječ koja opisuje **model** reflektivnosti površine.
Postoji niz modela i modificirajućih parametara (npr. metallic, specular, ambient, diffuse, reflection …)

Finish blok unutar teksture specificira kako površina objekta reflektira svjetlo.

0.9 je "količina" phong-a, tj. parametar koji specificira phong refleksiju. Vrijednosti od 0 do 1 su dopuštene.

Phong model rezultira svijetlom točkicom na površini objekta čiji položaj ovisi o izvorima svjetlosti, a "sjajnost" o "količini" phong-a. **phong_size** parametar određuje veličinu točkice. Npr. phong_size 250 odgovara vrlo poliranoj površini, dok bi phong_size 40 odgovaralo "plastičnom" objektu.



- **NORMAL**

```
sphere { <0,0,0>, 1
pigment {Gray75}
normal { bumps 1 scale 0.2 }
}
```

Normal blok unutar teksture specificira modulaciju normale (okomice) površine objekta.

**bumps** je ključna riječ koja opisuje **model** modulacije površine.
Postoji niz modela i modificirajućih parametara (npr. dents, wrinkles, ripples, waves …)

Skaliranje modulacije ("kvrgavosti" u ovom slučaju) – manje skale odgovaraju "gušćoj" kvrgavosti.

Modificirajući parametar **bumps** modela perturbacije normale površine objekta: veći brojevi odgovaraju više izraženoj "kvrgavosti"

Treba napomenuti da normal blok **ne** modificira stvarnu 3D geometriju površine, on samo simulira kvrgavost objekta u refleksiji svjetlosti s objekta. Na ovaj način moguće je vrlo "jeftino" i numerički brzo dobiti efekt na površini objekta bez stvarne promjene površine objekta.

- **SLIKA KAO PIGMENT (image_map)**

```
plane { <0,0,-1>,0
pigment {
image_map {png "Eggs.png“
once
map_type 0
interpolate 2}
}
}
```

Projicira **png** (alternativno: **jpeg, gif, tga, tiff**, …) format slike Eggs.png na ravninu. Slika po defaultu popunjava kvadrat u x-y ravnini od (0,0) do (1,1) i ponavlja se “slaganjem” (tiling) duž cijele ravnine. Skaliranjem pigmenta možemo promijeniti i veličinu kvadrata na koji se slika projicira. **Once** ključna riječ znači da se eliminiraju sve kopije slike osim one između (0,0) i (1,1) koordinata.

Vrsta projekcije slike na objekt:

**map_type 0** – planarno mapiranje

**map_type 1** – sferično mapiranje

**map_type 2** – cilindrično mapiranje

**interpolate** ključna riječ pokušava “izgladiti” (interpolirati) sliku koja se deformira prilikom projekcije. **interpolate 0** – nema interpolacije, **interpolate 2** – bilinearna interpolacija, **interpolate 4** – normalized distance interpolacija.



# XI. Primjer

**(image_map, sor, intersection, normal, finish)**

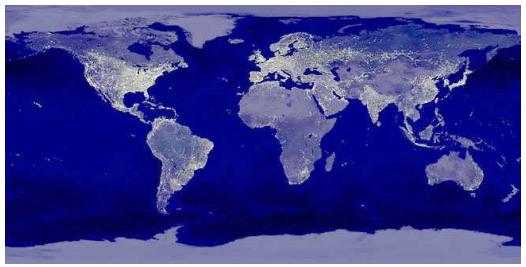
```
camera {
angle 45
location  <-7.0 , 3.5 ,-8.5> // pogled sa strane
look_at   <0.0 , 3.0 , 0.0>
//location <-7,9,-5> //pogled odozgo
//look_at <0,3.2,0>
}

light_source{ <0.0, 60.0, -100.0> color rgb<1,1,0.9>} // "sunce"

#declare dno = sor{6, // 6 tocaka u sor objektu
<0,0>,<1,0>,<1,0.2>,
<0.5,0.3>,<0.1,0.45>,<0,0.44>
texture{
pigment{color rgb<1,0,0>}
finish{phong 0.9 phong_size 40}  // "plastika"
} // kraj texture
}; // kraj sor

#declare stanga = cylinder{
<0,0,0>, <0,6.25,0>, 0.05
texture{
pigment{color rgb<0.97,0.97,0.97>} // siva, gotovo bijela
finish{phong 1.0 phong_size 250} // "metal"
} // kraj texture
}; // kraj cylinder
```

Profil koji rotiramo:

```
#declare luk = intersection{
box{<-1.2,-3,-0.08>, <5,3,0.08>} // kraj box
difference{
sphere{<0,0,0>, 2.7}
sphere{<0,0,0>, 2.4}
} // kraj difference, sferna "ljuska"
texture{
pigment{color rgb<0.75,0.1,0.1>}
finish{phong 0.5 phong_size 40} // "plastika"
} // kraj texture
}; // kraj intersection (luk)

#declare globus = sphere{<0,0,0>, 2.25
pigment{image_map {jpeg "Earth2.jpg" map_type 1 interpolate 0}}
}; // kraj globus

#declare full_globus = union{
object{dno}
object{stanga}
object{luk translate<0,3.3,0>}
object{globus translate<0,3.3,0>}
} // cijeli globus sa stativom

plane{<0,1,0>, 0 // tlo
pigment{color rgb<0.8,0.6,0.4>}
normal {bumps 0.5 scale 0.05} // kvrgice... primitivni "pijesak"
}
sphere{<0,0,0>, 10000
pigment{color rgb<0.5,0.5,1.0>}
} // ogromna sfera unutar koje se nalazi cijela scena, glumi "nebo"

object{full_globus}
```

Presjek tankog kvadra i sferne ljuske

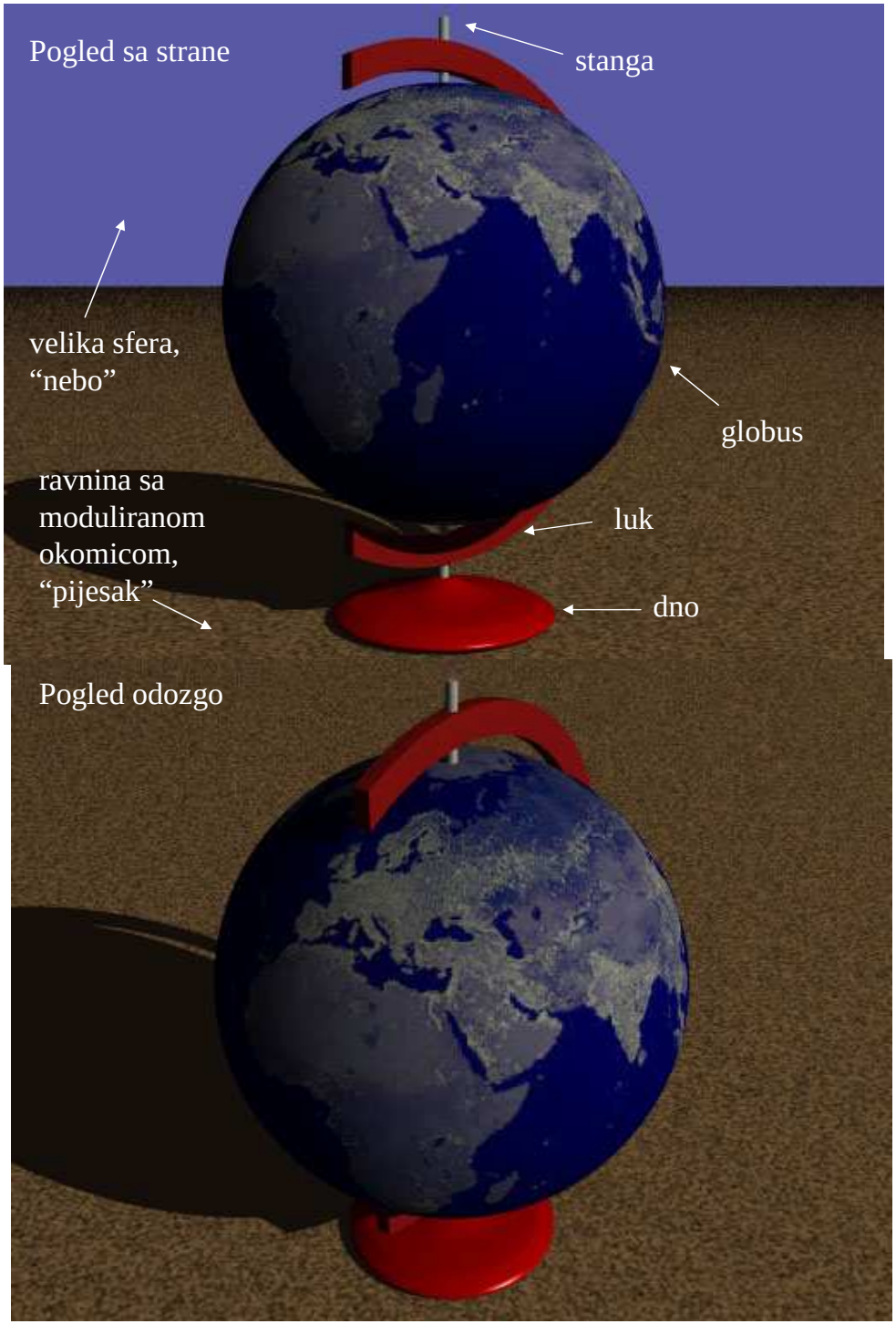
Sferno (map_type 1) mapiranje bez interpolacije



Slika mora biti u istom direktoriju kao i .pov file koji pišemo.

Ogroman radijus sfere, “nebo”. Objekti u PovRayu nisu tijela nego površine, nisu ispunjeni po defaultu.





# XII. 3-, 4- i 5-vektori boje

Standardna boja opisana je rgb 3-vektorom. Moguće je dobiti djelomično transparentne boje, pigmente specificirajući boju kao npr.

```
color rgbt <1,0.2,0.1,0.8>
```

Broj 0.8 govori da boja, pigment, tj. objekt na koji je primijenjen propušta 80% svjetlosti, što znači da je objekt djelomično proziran, "staklen" ili slično. Općenito, "boju" specificiramo 5-vektorom (rgbft) kao

```
color rgbft <1,0.2,0.1,0.5,0.8>
```

Četvrta komponenta specificira količinu "filtrirane" prozirnosti, tj. svjetlost koja prolazi kroz objekt poprima djelomično (f<1) ili u potpunosti (f=1) boju objekta.

rgbft skraćenica označava *red green blue filter transmit*

rgbt skraćenica označava *red green blue transmit*

PovRay računa uvijek s 5-vektorima, tj. ako filter i transmit komponente nisu specificirane, PovRay ih postavlja na nulu.



# XIII. PovRay i računanje; macro-i

Bilo bi mukotrpno kad bismo svaki objekt morali detaljno specificirati. PovRay-om možemo izračunati objekt. Macro-i su rutine koje na osnovu parametara kojim ih pozivamo računaju objekt prema zadanim pravilima. Macro-i ne moraju nužno generirati objekt. Rezultat njihovog pozivanja može biti niz, vektor i slično. Općenito, sintaksa macro-a izgleda kao

```
#macro ime_macroa (param1, param2, …)
```

… funkcije, declare statementi i slično

```
#end // kraj macro-a
```

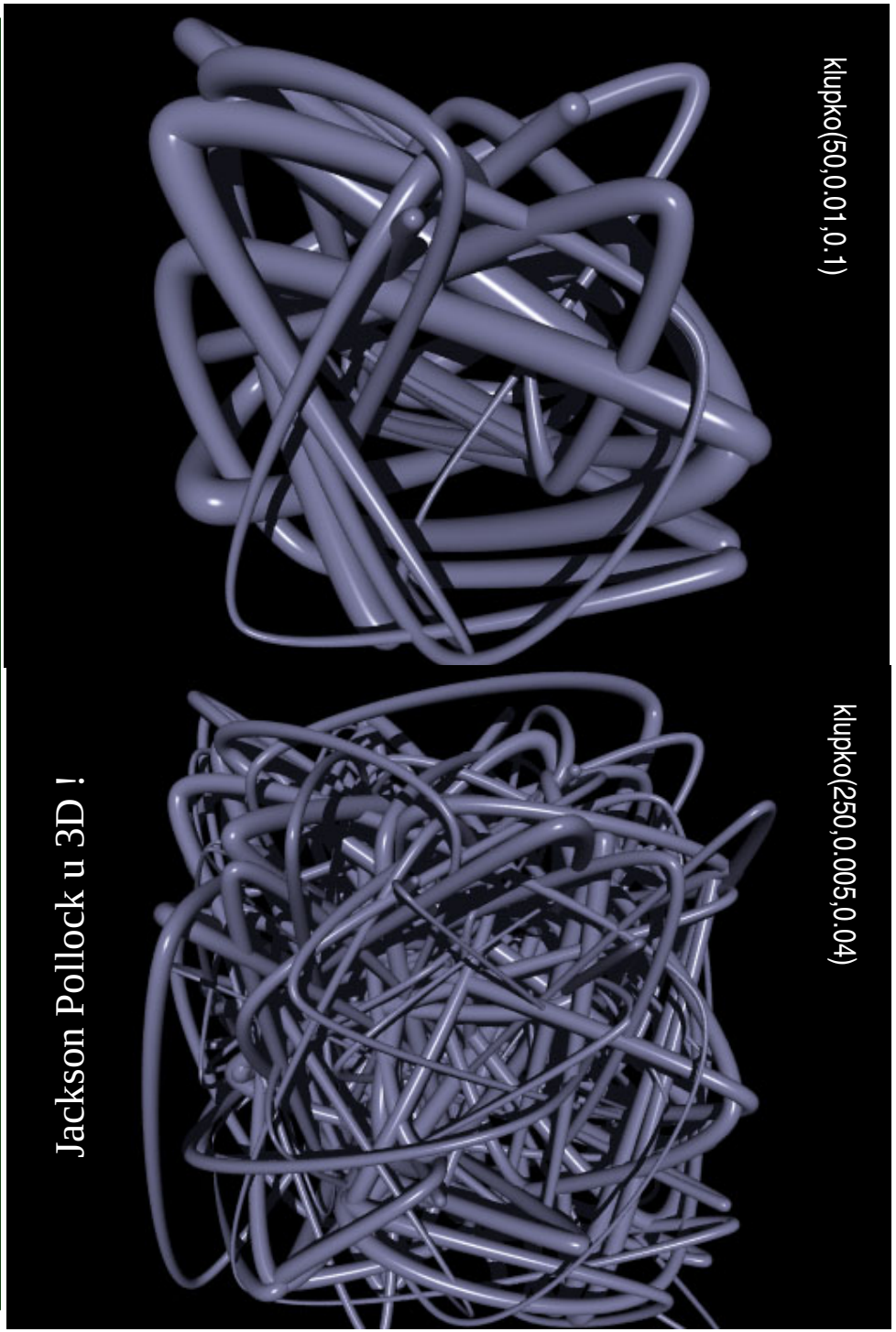
Pozivom macro-a sa različitim kombinacijama parametara (param1, param2,…) moguće je dobiti različite varijacije objekata koji macro proizvodi. Parametri mogu biti skalari, ali i vektori, nizovi i slično. Macro mora biti definiran prije (u smislu toka koda) nego što ga se poziva.

# XIV. Primjer

## (macro, while, sphere_sweep, seed, rand)

```
camera { angle 42 location <0,2,-4> look_at <0,0,0>}
light_source{ <0.0, 60.0, -100.0> color rgb<1,1,0.9>} // "sunce"
#macro klupko (npts,rmin,rmax) // macro koji proizvodi “klupko”
#declare seeedx=seed(110); // "sjemenske" (seed) varijable
#declare seeedy=seed(120); // sluze za "pumpanje" random
#declare seeedz=seed(150); // funkcija. Svaki random niz
#declare seeedr=seed(200); // ovisi o "sjemenu" iz kojeg pocne.
#declare i=0; // pocetni index
sphere_sweep{ cubic_spline npts , // kubicni spline sa npts tocaka
 #while (i<npts) // sve dok je i (indeks) manji od broja tocaka (npts)...
  #declare x1=-1 + 2*rand(seeedx); // x1 iz intervala [-1,1]
  #declare y1=-1 + 2*rand(seeedy); // y1 iz intervala [-1,1]
  #declare z1=-1 + 2*rand(seeedz); // z1 iz intervala [-1,1]
  #declare rad = rmin+(rmax-rmin)*rand(seeedr);
// rad iz intervala [rmin, rmax]          ← random number generator [0,1]
  <x1,y1,z1>, rad // unutar while petlje dodajem koordinatu i radijus sfere
// gornja linija je kao dinamicko dodavanje
// parametara unutar "otvorenog" sphere_sweep objekta
  #declare i=i+1;
// nakon dodane tocke u sphere_sweep povecavam "loop" index za 1
 #end // while
tolerance 0.05 // niska tolerancija, precizniji i sporiji racun
texture{pigment{color rgb<0.7,0.7,1.0>} // plavkasto klupko
finish{phong 0.7 phong_size 40} // "plastika"
} //kraj teksture
} // kraj sphere_sweep (zatvaranje objekta nakon petlje)
#end // kraj macro-a klupko
klupko(50,0.01,0.1) // pozivanje macro-a klupko
//klupko(250,0.005,0.04) // OPREZ !!! Vrlo sporo !
```





# XV. Primjer

**(macro, while, if, sky_sphere)**

```
camera {angle 60 location <0.0 , 20 ,-17> look_at <0.0 , 0.0 , 0.0>}
sky_sphere{pigment{color rgb<0.9,0.9,1.0>}}
light_source{<1000,2500,-50> color rgb<1.0,0.95,0.95>}
#macro nanocryst(radiu) // nanokristal, nanoklaster
#declare  r0 = 1.0/pow(2,0.5); // sqrt(2) / 2 = 1 / sqrt(2)
#declare m = int(radiu/r0) + 1; // int (a)  - integer vrijednost od (a)
#declare i=-m;
#declare textex = texture { pigment { rgb <0.36, 0.28, 0.20>*2 }
finish { phong 0.9 phong_size 250} };
union{
  #while (i<m)
   #declare j=-m;
    #while (j<m)
     #declare k=-m;
      #while (k<m)
       #declare rtmp = sqrt(i*i + j*j + k*k);
        #if (rtmp<radiu)
         #if (mod((i+j+k),2)=0) // mod(i,2) – ostatak od dijeljenja i sa 2
          sphere{<i,j,k>, r0 texture{textex}}
         #end // end if
        #end // end if
       #declare k=k+1;
      #end // end while
     #declare j=j+1;
    #end // end while
   #declare i=i+1;
  #end // end while
      } // end union
#end // end macro
//object{nanocryst(9.742) rotate<0,-7,0> }
object{nanocryst(5.5) rotate<0,-29,0> }
```

sky_sphere – sfera formalno beskonačnog radijusa u kojoj se scena nalazi, nije pravi objekt, treba je tretirati kao “pozadinu”.

udaljenost

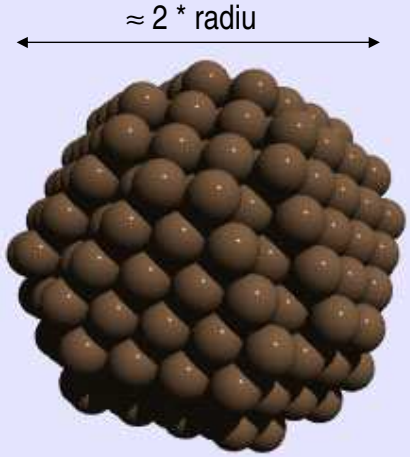
#if (uvjet) … #end

FCC (face centered cubic) slaganje unutar sfere radijusa radiu (parametar macro-a).

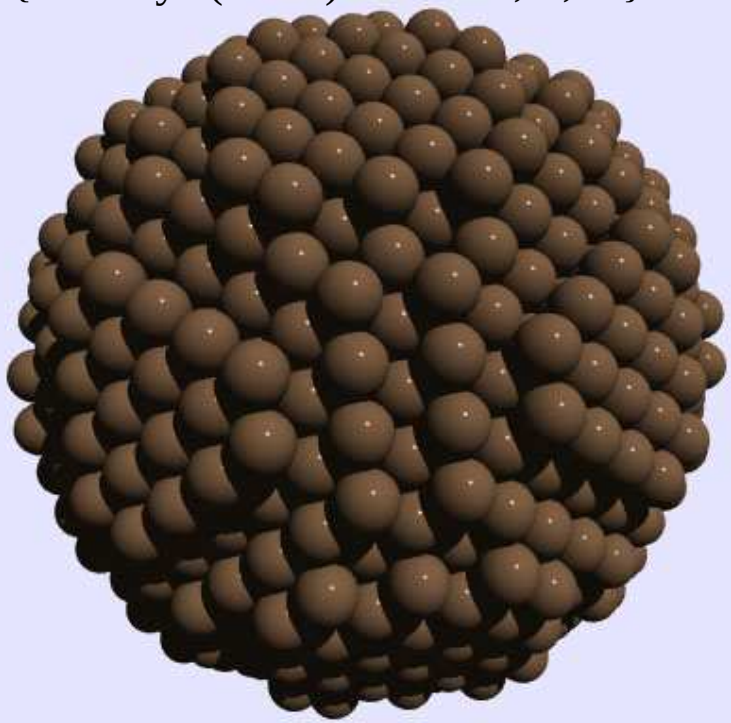
Povećavanje loop indeksa (k,j,i) za while petlje.



object{nanocryst(5.5) rotate <0,-29,0>}




object{nanocryst(9.742) rotate <0,-7,0>}



# XVI. Animacija u PovRayu

PovRay se može koristiti za animiranje scene. O animaciji trebamo razmišljati kao o nizu stacionarnih scena. Spajanje generiranih (izračunatih) stacionarnih scena (ili frame-ova) u kontinuirani tijek rezultira filmom (ova operacija mora se obaviti izvan PovRay-a).

Najjednostavnija verzija animacije je ona u kojoj se 3D koordinatama koje opisuju scenu pripisuje vremenska ovisnost ("gibanje" karakterističnih koordinata). Vremenska ovisnost u PovRay-u specificirana je rezerviranom skalarnom varijablom clock. Tako bismo npr. jednoliko gibanje sfere duž x-osi mogli napisati kao

```
sphere{<0,0,0>, 1 translate <4*clock,0,0>}
```

Interval u kojem se varijabla clock mijenja kao i broj slika (frame-ova) koje je potrebno generirati unutar tog intervala tipično su zadani u file-u odvojenom od .pov file-a koji opisuje scenu, u kratkom inicijalizacijskom file-u (.ini) koji sadrži referencu na .pov file scene. Kad želimo proizvesti animaciju, PovRay-evim kompajlerom procesiramo .ini file.



# XVII. Primjer

**(.ini, clock, rgbft, polygon)**

```
camera { angle 70 location <0,1,-10> look_at <0,0,0>}
light_source{ <0.0, 60.0, -100.0> color rgb<1,1,0.9>} // "sunce"
sky_sphere{pigment{color rgb<1,1,1>}}
#declare staklo = polygon{8,
<-2,-3>, <2,-3>, <3,-1>, <3,2>,
<1,3>, <-1,3>, <-3,2>, <-3,-1>
pigment{color rgbft<1.0,0.3,0.2,0.8,0.2>}};
#declare ocale = union{
object{staklo translate <-3.75,0,0>} // lijevo staklo
object{staklo translate <3.75,0,0>}  // desno staklo
cylinder{<-0.8,2,0>,<0.8,2,0>, 0.2 // precka
texture{pigment{color rgb<1,1,1>} finish{phong 0.9 metallic}}}};
#macro kuglice(n,cvect1,cvect2,l)
#declare i=0;
#declare s1=seed(1411);
#declare s2=seed(1241);
#declare s3=seed(1621);
#declare sc=seed(1901);
#while (i<n)
 sphere{<-l+2*l*rand(s1),-l+2*l*rand(s2),-l+2*l*rand(s3)>, 0.4
 texture{pigment{color cvect1 + rand(sc)*cvect2}
 // generiranje boje oko cvect1 boje dodavanjem cvect2 boje
 finish{phong 1.0 phong_size 250}}}
 #declare i=i+1; // povecavanje loop indeksa
#end  // while
#end // macro
kuglice(200,<0,0,0.5>,<0.5,0.7,0.4>,10)
// l=10 je velicina kutije u koju pospremam kuglice
object{ocale translate<0,0,9 - 15.0*clock>}
// translatiram ocale prema kameri, svijet izgleda ruzicasto
```

Najprije kreiramo .pov file koji opisuje scenu (ocale.pov).

U .pov file-u specificiramo i vremensku ovisnost objekata i slično, tj. uvodimo ovisnost o clock varijabli..

```
; Persistence Of Vision raytracer version 3.5 sample file.

Antialias=On

Antialias_Threshold=0.25
Antialias_Depth=3

Input_File_Name=ocale.pov

Initial_Frame=1
Final_Frame=12

Initial_Clock=0
Final_Clock=1.0

Cyclic_Animation=off
Pause_when_Done=off
```

Zatim napišemo .ini file koji opisuje clock varijablu, broj frame-ova (ocale.ini). PovRay-em kompajliramo ini file koji sadrži referencu na prije napisani .pov file (ocale.pov).

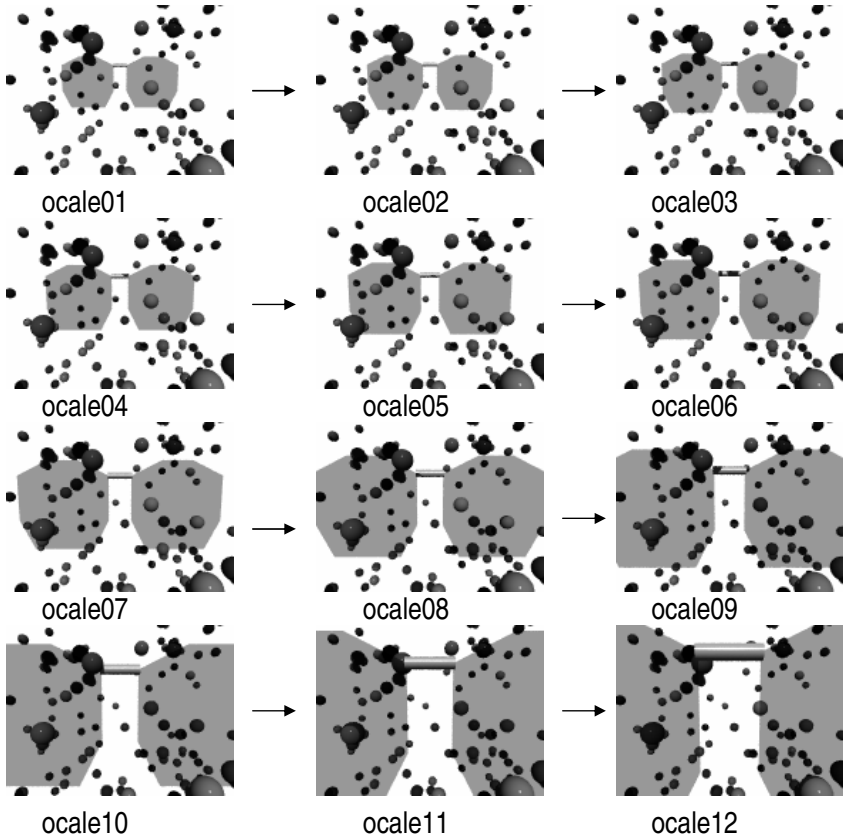
.pov file na koji se .ini file odnosi.

Početni i konačni frame (12 frame-ova ukupno).

Početna i konačna vrijednost clock varijable. Unutar 12 frame-ova clock se mijenja od 0 do 1.

Nakon kompajliranja ocale.ini file-a, PovRay proizvodi 12 slika jednu za drugom i sprema ih u direktorij kao ocale01, ocale02, … ocale12.






Nekim vanjskim programom (npr. Virtual Dub, avicreator ili Macromedia Flash, postoji niz freeware aplikacija) potrebno je dobivene slike spojiti u .avi animaciju ili neki drugi format (*.swf,*.mov, *.wmv …).

# XVIII. Primjer

**(napredniji macroi, naprednije modeliranje, geodetski model osvjetljenja)**

Osvjetljenje u «realnom» svijetu ne dolazi iz jednog, točkastog izvora svjetlosti. Čak ni u zatvorenoj prostoriji s jednom žaruljom, predmeti nisu osvijetljeni samo tim točkastim izvorom svjetlosti, nego i difuzno, tj. svjetlošću koja se reflektira s drugih predmeta, zidova prostorije... PovRay često proizvodi «neprirodno», «sintetski»-izgledajuće objekte, dijelom zbog pogrešno (i prejednostavno) postavljenih izvora svjetlosti. Slijedeći primjer pokazuje kako se difuzna rasvjeta scene može simulirati postavljanjem nizom točkastih izvora scene na veliku sferu koja scenu okružuje. Slijedi kod primjera s kratkim objašnjenjima. Elemente koda možete slijepiti jednog iza drugog, redoslijedom kojim se u ovom tekstu pojavljuju i dobit ćete funkcionalan PovRay file.

```
#macro trianguliraj (order, a1, a2, a3, radijus, l_color, prob)
// GEODETSKA KUPOLA SVJETALA
 #declare vec1 = (a2-a1)/(order+1);
 #declare vec2 = (a3-a1)/(order+1);
 #declare vec12 = (a2-a3)/(order+1);
 #declare j=0;
 #declare rs1 = seed (21432);
 union{
 #while (j <= order)
  #declare i=0;
  #declare tpoint = a1 + vec1*(order-j);
  #while(i <= j)
   #declare point = tpoint + vec2*i;
    light_source{
     ( vnormalize((1-prob)*point + prob*<rand(rs1),rand(rs1),rand(rs1)>)*radijus
     +   vnormalize((1-prob)*(point+vec1)   +   prob*<rand(rs1),rand(rs1),rand(rs1)>
)*radijus
     +   vnormalize((1-prob)*(point+vec2)   +   prob*<rand(rs1),rand(rs1),rand(rs1)>
)*radijus )
/ 3.0
       color l_color / 3.5/ pow((order+1),2)
      }


   #if (i != 0)
    light_source{
    (vnormalize(point)*radijus     +     vnormalize(point     +vec12)*radijus     +
vnormalize(point + vec1)*radijus) / 3.0
     color l_color / 3.5 / pow((order+1),2)
    }

   #end

   #declare i=i+1;
  #end
   #declare j=j+1;
 #end
} // kraj union objekta
#end
```

Gornji #macro je glavna «caka» u ovom primjeru. Ono što on radi je iterativna triangulacija trokuta s vrhovima u točkama `a1`, `a2`, `a3` i njihova projekcija na sferu radijusa `radijus`, te postavljanje točkastih izvora svjetlosti u dobivene koordinate. U daljnjoj implementaciji, točke `a1`, `a2`, `a3` opisivat će vrhove pravilnog poliedra (konkretno ikozaedra), što znači da gornji #macro proizvodi nešto slično mreži na jednoj od stranica geodetske kupole (vidi



http://nanoatlas.ifs.hr/hrv/geodesic_dome.html ). Ovisno o parametru `prob`, točke su postavljene ili savršeno pravilno na geodetske koordinate (za `prob=0`), ili sa određenom količinom nasumičnosti (`prob` ∈ `<0,1>`).

```
#macro geodome(order, skala, boja_svjet, nasumicnost)
 #declare phi = 1.618034; // zlatni rez
 #declare Nv = 12; // broj verteksa
 #declare v1 = array[Nv+1];
 #declare v1[0] = <0,0,0>;
// verteksi ikosaedra
 #declare v1[1] = <0,phi,1> * skala;
 #declare v1[2] = <0,phi,-1> * skala;
 #declare v1[3] = <0,-phi,1> * skala;
 #declare v1[4] = <0,-phi,-1> * skala;
 #declare v1[5] = <1,0,phi> * skala;
 #declare v1[6] = <1,0,-phi> * skala;
 #declare v1[7] = <-1,0,phi> * skala;
 #declare v1[8] = <-1,0,-phi> * skala;
 #declare v1[9] = <phi,1.0,0> * skala;
 #declare v1[10] = <phi,-1.0,0> * skala;
 #declare v1[11] = <-phi,1.0,0> * skala;
 #declare v1[12] = <-phi,-1.0,0> * skala;

 #declare Nc = 20; //broj stranica
 #declare conn = array[Nc+1];
 #declare conn[0] = <0,0,0>;
// matrica povezanosti ikosaedra
 #declare conn[1] = <1,2,9>;
 #declare conn[2] = <1,2,11>;
 #declare conn[3] = <1,5,7>;
 #declare conn[4] = <1,5,9>;
 #declare conn[5] = <1,7,11>;
 #declare conn[6] = <2,8,11>;
 #declare conn[7] = <2,6,8>;
 #declare conn[8] = <2,6,9>;
 #declare conn[9] = <3,4,10>;
 #declare conn[10] = <3,5,10>;
 #declare conn[11] = <3,4,12>;
 #declare conn[12] = <3,5,7>;
 #declare conn[13] = <3,7,12>;
 #declare conn[14] = <4,6,10>;
 #declare conn[15] = <4,6,8>;
 #declare conn[16] = <4,8,12>;
 #declare conn[17] = <5,9,10>;
 #declare conn[18] = <6,9,10>;
 #declare conn[19] = <7,11,12>;
 #declare conn[20] = <8,11,12>;

 #declare rad = sqrt(1+phi*phi) * skala; // radijus sfere opisane ikosaedru
 #declare ib=1;
 union{
 #while(ib<=Nc)
  trianguliraj(order,v1[int(conn[ib].x)],  v1[int(conn[ib].y)],  v1[int(conn[ib].z)],
rad, boja_svjet, nasumicnost)
  #declare ib=ib+1;
 #end
}
#end
```

Gornji #macro postavlja koordinate 12 vrhova ikozaedra u niz `v1`, te specificira njihovu povezanost u strane ikozaedra kroz «connectivity matrix» (matricu povezanosti), `conn`. Tako npr. `conn[10] = <3,5,10>` znači da desetu stranu ikozaedra čine vrhovi 3,5 i 10. Nadalje, u `#while` petlji, macro prolazi kroz svaku stranu ikozaedra i triangulira je tj. na mjesto triangulacije postavlja izvore svjetlosti pozivom #macro-a `trianguliraj`. Red

triangulacije dan je parametrom `order`. Što je `order` veći to je trokutasta mreža kojom se pokriva svaka od strana ikozaedra gušća (prouči `#macro trianguliraj` za detalje).

```
#macro omot_kotaca (N,lrad,thick,perc)
#declare abox = 6.28 * lrad / N * perc;
#declare obj = box{<-abox/2,-thick/2,-abox/2> ,<abox/2,thick/2,abox/2>};
#declare iobj = 0;
union{
 #while (iobj < N)
   object{obj translate <0,lrad,0> rotate<iobj * 360/N,0, 0>}
   #declare iobj = iobj + 1;
 #end //while
} //union
#end //macro

#declare legoc2=
//Crvena lego-kocka 2x2
union{
box{<-0.5,-0.25,-0.5>, <0.5,0.25,0.5>}
cylinder{<-0.25,0.2,-0.285>, <-0.25,0.35,-0.285>, 0.135}
cylinder{<0.25,0.2,-0.285>, <0.25,0.35,-0.285>, 0.135}
cylinder{<-0.25,0.2,0.285>, <-0.25,0.35,0.285>, 0.135}
cylinder{<0.25,0.2,0.285>, <0.25,0.35,0.285>, 0.135}
pigment{color rgb<0.89,0,0>}
}

#declare legoc4=
//Crvena lego-kocka 4x2
union{
object{legoc2 translate<0.4999999,0,0>}
object{legoc2 translate<-0.4999999,0,0>}
}

#declare legocr2=
//Crna tanka lego kocka 2x2 (ne postoji u strukturi)
union{
box{<-0.5,-0.06,-0.5>, <0.5,0.06,0.5>}
cylinder{<-0.25,0.05,-0.285>, <-0.25,0.16,-0.285>, 0.135}
cylinder{<0.25,0.05,-0.285>, <0.25,0.16,-0.285>, 0.135}
cylinder{<-0.25,0.05,0.285>, <-0.25,0.16,0.285>, 0.135}
cylinder{<0.25,0.05,0.285>, <0.25,0.16,0.285>, 0.135}
pigment{color rgb<0,0,0>}
}

#declare legocr10=
//Crna tanka lego kocka 10x2
union{
object{legocr2 translate<4*0.4999999,0,0>}
object{legocr2 translate<2*0.4999999,0,0>}
object{legocr2}
object{legocr2 translate<-2*0.4999999,0,0>}
object{legocr2 translate<-4*0.4999999,0,0>}
}

#declare legos2=
//Siva lego kocka 2x2 s osovinama za kotace
union{
box{<-0.5,-0.25,-0.5>, <0.5,0.25,0.5>}
cylinder{<-0.25,0.2,-0.285>, <-0.25,0.35,-0.285>, 0.135}
cylinder{<0.25,0.2,-0.285>, <0.25,0.35,-0.285>, 0.135}
cylinder{<-0.25,0.2,0.285>, <-0.25,0.35,0.285>, 0.135}
cylinder{<0.25,0.2,0.285>, <0.25,0.35,0.285>, 0.135}
//osovine
cylinder{<-1.0,0.1,0>, <-0.4,0.1,0>, 0.11}
cylinder{<1.0,0.1,0>, <0.4,0.1,0>, 0.11}
pigment{color rgb <0.59,0.59,0.59>}
}
```



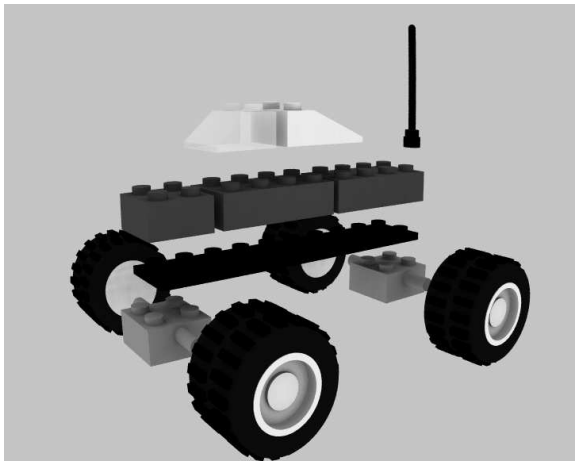
```
#declare kotac =
union{
object{omot_kotaca(21,0.75,0.15,0.5)  pigment{color  rgb<0,0,0>}  scale  <2.9,1,1>
translate<0.25,0,0>}
object{omot_kotaca(21,0.75,0.15,0.5)     pigment{color     rgb<0,0,0>}     rotate
<360/21/2,0,0> scale <2.9,1,1> translate<-0.25,0,0>}

difference{
cylinder{<-0.5,0,0>,<0.5,0,0>, 0.75 pigment{color rgb<0.05,0.05,0.05>} }
cylinder{<-0.51,0,0>,<0.51,0,0>, 0.4800011 pigment{color rgb<0.05,0.05,0.05>} }
}

difference{
cylinder{<-0.51,0,0>,<0.51,0,0>, 0.48 pigment{color rgb<1,1,1>} }
cylinder{<0.35,0,0>,<0.65,0,0>, 0.40 pigment{color rgb<1,1,1>} }
}

cylinder{<0.34,0,0>,<0.4,0,0>, 0.22 pigment{color rgb<1,1,1>} }
}

#declare legobk =
// kosa bijela lego kocka
union{
box{<-0.5,-0.25,-0.5>, <0,0.25,0.5>}
cylinder{<-0.25,0.2,-0.285>, <-0.25,0.35,-0.285>, 0.135}
cylinder{<-0.25,0.2,0.285>, <-0.25,0.35,0.285>, 0.135}
object{
prism{
linear_sweep
linear_spline
-0.5, // sweep the following shape from here ...
0.5, // ... up through here
5, // the number of points making up the shape ...
<0,-0.25>, <1.0,-0.25>, <1.0,-0.17>, <0,0.25>, <0, -0.25>
}
rotate<-90,0,0>
}
pigment{color rgb<1,1,1>}
}

#declare legozk =
// kosa zuta lego kocka
merge{
box{<-0.5,-0.25,-0.5>, <0,0.25,0.5>}
cylinder{<-0.25,0.2,-0.285>, <-0.25,0.35,-0.285>, 0.135}
cylinder{<-0.25,0.2,0.285>, <-0.25,0.35,0.285>, 0.135}
object{
prism{
linear_sweep
linear_spline
-0.5,
0.5,
5,
<-0.00001,-0.25>, <0.5,-0.25>, <0.5,-0.17>, <-0.00001,0.25>, <-0.00001, -0.25>
}
rotate<-90,0,0>
}
pigment{color rgbft<1,1.5,0, 0.0,0.7>}
}

#declare antena =
// antena kamioncica
union{
cylinder{<0,0,0>,<0,2,0>, 0.065}
sphere{<0,2,0>, 0.065}
cylinder{<0,0,0>,<0,0.17,0>, 0.15}
cylinder{<0,0,0>,<0,0.29,0>, 0.125}
```

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

U prethodnom dijelu koda nema ničeg posebno interesantnog – radi se o definiciji objekata koji će sačinjavati scenu. Jasno je da se radi o lego kockicama. Možda je donekle interesantan `#macro omot_kotaca` koji postavlja `box` objekte na kružnicu, rotirajući ih u skladu s njihovim kutnim položajem. I na kraju, slijedi postava scene:

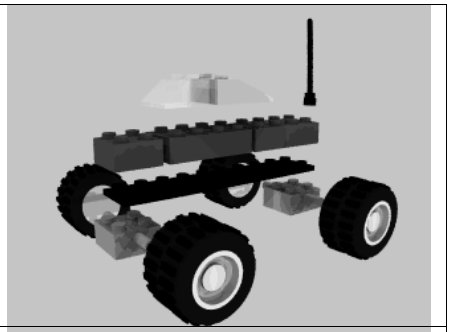
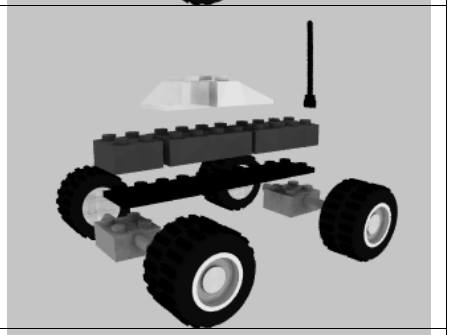
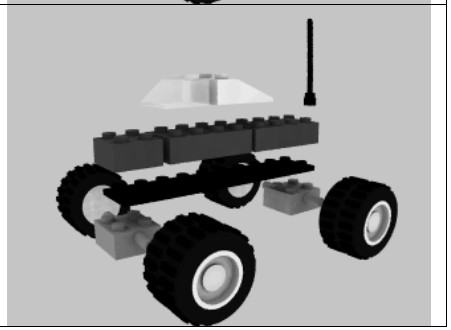
```
//SLIJEDI SETUP SCENE
object{ kotac rotate<0,90,0> translate <-2.05,0,-2.3> }
object{ kotac rotate<0,90,0> translate <2.05,0,-2.3> }

object{ kotac rotate<0,-90,0> translate <-2.05,0,2.3> }
object{ kotac rotate<0,-90,0> translate <2.05,0,2.3> }

object{legos2 rotate<0,90,0> translate  <-2.25, -0.1, 0>}
object{legos2 rotate<0,90,0> translate  <2.25, -0.1, 0>}

object{legocr10 translate <0, 0.7, 0>}

object{legoc4 translate <-1.6, 1.55, 0>}
object{legoc4 translate <0.5, 1.55, 0>}

object{legoc2 translate <2.15, 1.55, 0>}

object{legobk rotate<0,180,0> translate <-0.1, 2.6, 0>}
object{legozk translate <1.1, 2.6, 0>}

object{antena translate <-2.5,2.2,0.28>}

#declare boja_svjetla = rgb<1,1,1>;
object{geodome(4, 40, boja_svjetla, 0.7)} //osvjetljenje

background{rgb <0.79,0.77,0.78>}
camera{
ultra_wide_angle
location       <10.05*sin(0.1*6.283185),4*cos(0.1*6.283185),10.05*cos(0.1*6.283185)>
look_at <0,1,0> angle 51}
```

A rezultat koji se dobije izvršavanjem prikazanog koda prikazan je na slici desno. Nisam siguran koliko se toga vidi na ovoj slici, ali dobija se efekt gotovo hiper-realnosti i difuzno osvjetljenje. Pored geodetske kupole svjetala, mogao sam dodati i još jedan, poseban izvor svjetlosti koji bi simulirao stvarnu rasvjetu scene, «pravi» izvor svjetlosti.

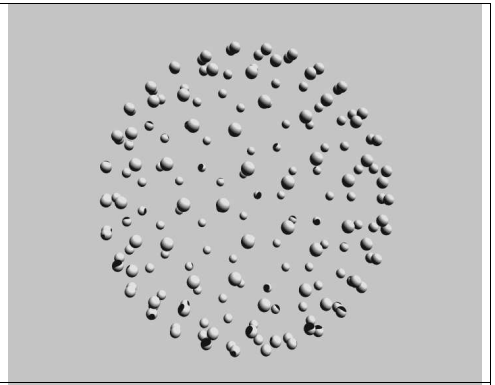
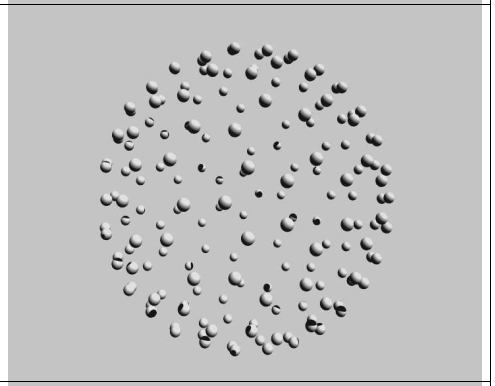
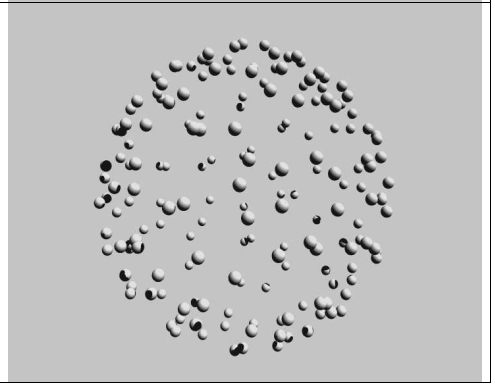




Slijedi nekoliko testova modela osvjetljenja:

| | |
|---|---|
| `prob = 0`<br>`order = 0`<br><br>Nizak iterativni red geodetske kupole, uočite kako se jasno vidi diskretnost difuznog osvjetljenja, niz sjena koje ocrtavaju oblike objekata u sceni. |  |
| `prob = 0`<br>`order = 1`<br><br>Diskretne scene se polako gube |  |
| `prob = 0`<br>`order = 2`<br><br>Gotovo da diskretnosti u sjenama već u ovom redu ni nema, opazite kako su sjene «mekane», nisu oštrih rubova kao što se u PovRay-u tipično događa kad je scena osvijetljena jednim točkastim izvorom svjetlosti. |  |

A evo kako izgledaju geodetske kupole svjetala ovisno o parametru nasumičnosti `prob`:

| | |
|---|---|
| `prob = 0`<br>`order = 2`<br><br>Ovo je savršeno pravilna geodetska kupola svjetala (u stvari se radi u nekoj vrsti fulerena – uočite peterokute i šesterokute u rasporedu svjetala). |  |
| `prob = 0.5`<br>`order = 2` |  |
| `prob = 0.9`<br>`order = 2` |  |

Ne preporučujem dizanje parametra `prob` iznad 0.7. Ovo je ostavljeno kao mogućnost samo ukoliko se na renderiranoj slici pojavi *banding* efekt (pravilna mreža sjena) – ukoliko toga nema, najbolje je držati parametar `prob` na nuli (`prob=0`).

# Putovanje kroz venu u PovRay-u

**(include fileovi, nizovi (arrays), zapisivanje i učitavanje podataka u PovRayu)**

**A. Šiber, 09.08.2004.**

Ideja je bila načiniti prikaz «putovanja» kroz venu punu «krvi» koja je predstavljena velikim brojem eritrocita. Scena je trebala biti dio zoom kadra gdje kamera zoomira ljudsko lice, povećava dio oka, žilice u oku, te na kraju promatrač «ulazi» u žilicu oka gdje se nađe u toku eritrocita. Ono što je potrebno za realizaciju ovakve ideje je definirati dinamiku eritrocita tj. njihovo vremenski ovisno gibanje moguće pod utjecajem vanjske sile neovisne o vremenu. Na primjer, vanjska sila bi mogla simulirati suženje u veni te nakupljanje eritrocita u manjem presjeku uz eventualno povećanje njihove brzine. U verziji koja će ispod biti objašnjena, eritrociti ne međudjeluju jedan s drugim i sve sile su vanjske.

Htio sam napraviti jedan općeniti include file (eritro_guide.inc) koji bi glavni kod scene pozivao. Taj include file bi sadržavao pravila za Newtonovsku dinamiku niza objekata s predefiniranim početnim uvjetima (brzinama i položajima). Važno je zapamtiti da PovRay nema mogućnost prijenosa informacije o podacima iz jednog framea u drugi u toku animacije. To znači da se sve varijable koje je potrebno prenijeti iz jednog framea u drugi moraju snimiti u file na disku koji PovRay snimi nakon generiranja prethodnog framea i učita prije generacije slijedećeg framea. Ovo je riješeno unutar include filea. Za tu potrebu definirana su dva macroa (`save_array` i `read_array`) koji snimaju i učitavaju podatke:

```
/*
************************************************************
PARTICLE GUIDER INCLUDE FILE FOR PERSISTENCE OF VISION 3.5
************************************************************
Created by Antonio Siber, August 2004
Parameters to be declared:
N_part - number of particles (integer)
R_max - container radius in x-y plane (real)
z_max - container box dimension in z direction (real)
cent_box - container box center (vector)
vel_mean - mean initial velocity (vector)
vel_noise - velocity noise (real)
rot_mean - mean rotation velocity (vector)
rot_noise - rotational velocity noise (real)
potential - potential field float function
mass_part - particle mass (real)
part_object - particle object (POV object)
move_start - clock value for the start of the animation (real)
min_plane - real number - this is the plane to which the particles stick
min_dist - minimal distance between the points (real)
N_time - real number
************************************************************
*/
//declaring random seed variables
#declare seeed1 = seed(1091);
#declare seeed2 = seed(1292);
#declare seeed3 = seed(1093);
#declare seeed4 = seed(1094);
#declare seeed5 = seed(1095);
#declare seeed6 = seed(1906);
#declare seeed7 = seed(1907);

#declare PosArray = array[N_part]; //position vector array
#declare PosArray[0] = <0,0,0>;
#declare VelArray = array[N_part]; //velocity vector array
#declare VelArray[0] = <0,0,0>;
```

```
#declare RotArray = array[N_part]; //rotation velocity array
#declare RotArray[0] = <0,0,0>;
#declare InRotArray = array[N_part]; //initial rotation vector array
#declare InRotArray[0] = <0,0,0>;


#macro save_array()
 #fopen arr_dataw "temp_array.dat" write
 #local i=0;
  #while (i<N_part)
   #write(arr_dataw,  PosArray[i]," ,  ",VelArray[i],  " ,  ",  RotArray[i],  " ,  ",
InRotArray[i], " , ")
   #local i=i+1;
  #end //while
 #fclose arr_dataw
#end //macro

#macro read_array()
 #fopen arr_datar "temp_array.dat" read
 #local i=0;
 #local vect1 = <0,0,0>;
 #local vect2 = <0,0,0>;
 #local vect3 = <0,0,0>;
 #local vect4 = <0,0,0>;
  #while (i<N_part)
   #read(arr_datar, vect1, vect2, vect3, vect4)
    #declare PosArray[i] = vect1;
    #declare VelArray[i] = vect2;
    #declare RotArray[i] = vect3;
    #declare InRotArray[i] = vect4;
    #local i=i+1;
   #end //while
  #fclose arr_datar
#end //macro
```

Unutar `save_array` macroa snimljeni vektorski podaci su razdvojeni zarezima što omogućuje njihovo ponovno učitavanje. Koriste se ugrađene PovRay funkcije `#fopen` za otvaranje datoteke i `#fclose` za zatvaranje datoteke, te `#write` za pisanje u datoteku i `#read` za čitanje iz datoteke. Evo kako izgleda prvih par podataka u «`temp_array.dat`» datoteci:

```
<2.6753,-3.28892,8.22042>  , <0.0157064,-0.0420946,-20.5187>  , <119.325,-146.993,146.148>  ,
<71.8743,156.933,-87.5881>  , <-5.82645,7.5267,54.324>  , <0.024819,0.0477041,-20.4851>  ,
<118.165,-86.5126,64.2896>  , <-110.904,85.8003,-179.786>  , <0.536962,13.5256,70.9095>  ,
<0.0196236,0.0173526,-20.4781>  , <48.057,-129.608,96.599>  , <-165.992,153.514,39.8691>  , <-
16.2591,-3.23507,13.6435>  , <-0.017011,0.0258046,-20.4749>  , <72.1586,-87.2806,116.903>  ,
<67.6656,100.991,82.8467>  , <-0.0912752,0.269989,17.9424>  , <-0.0243709,0.0366873,-20.5432>
, <110.458,-88.2463,106.807>  , <68.921,-11.4074,-37.6336>  , <-13.2658,2.35009,42.2193>  ,
<0.021323,-0.0425241,-20.5277>  , <121.331,-163.396,60.3694>  , <-88.0246,35.0125,-72.7501>  ,
<4.89446,6.37714,47.3612>  , <0.0161294,0.00602789,-20.5201>  , <125.003,-73.4427,98.4935>  ,
<-53.2115,43.5412,-18.3028>  , <8.5235,15.4068,73.2969>  ,  ...
```

`PosArray` je niz vektora položaja čestica, `VelArray` je niz vektora brzina čestica, `RotArray` je niz vektora rotacionih brzina čestica, a `InRotArray` je niz početnih rotacijskih vektora čestica. Elementu niza pristupamo uz pomoć `ImeNiza[i]`, gdje je `i` indeks koji u našem slučaju počinje od nule. Uvedeni su rotacioni stupnjevi slobode jer je zamišljeno da se eritrociti rotiraju kako se gibaju prema promatraču. Ono što dalje treba napraviti ovisi o tome da li je animacija upravo počela ili animacija već traje. Ukoliko animacija upravo počinje, potrebno je inicijalizirati početne uvjete tj. nizove položaja, brzina, rotacionih brzina i početnih rotacionih položaja i snimiti ih u datoteku. U suprotnom, potrebno je učitati te nizove koji su snimljeni nakon generacije prethodnog frame te ih koristiti kao početne uvjete za frame koji se trenutno generira, evoluirati koordinate, brzine te njihove nove vrijednosti ponovno snimiti u datoteku koja će se učitati prilikom generacije slijedećeg

framea. Za tu potrebu uvedena je `my_clock` varijabla čija vrijednost opisuje stanje animacije. Ukoliko je `my_clock < 0,` animacija još nije počela i include file ne radi ništa. Ukoliko je `my_clock = 0`, animacija treba početi, a ukoliko je `my_clock > 0`, animacija već traje:

```
#declare my_clock = clock - move_start;
#if (my_clock<0)
// do nothing, the animation should not yet start
#else

//N_part is the number of particles
#if (my_clock=0)
// initialize only for the first time
//initializing position and velocity arrays:
#declare i=0;
#while (i<N_part)
 #declare phi1 = rand(seeed1)*6.283185;
 #declare R1 = rand(seeed2)*R_max;
 #declare PosArray[i] = <R1*sin(phi1), R1*cos(phi1), -z_max/2 + z_max*rand(seeed2)>
                    + cent_box ;
 #declare j=0;
 #declare dist_flag=0;
 #while (j<i)
  #declare dist = sqrt( pow( PosArray[i].x - PosArray[j].x,2) + pow(PosArray[i].y -
PosArray[j].y,2) + pow(PosArray[i].z - PosArray[j].z,2));
  // this is the distance between the two points
  #if (dist < min_dist)
   #declare dist_flag=1;
   #declare j=i;
// getting out of the while loop here
  #end //if
  #declare j=j+1;
 #end //while j

 #if (dist_flag = 0)
//cent_box is the box center
 #declare  VelArray[i]  =  vel_mean  +  vel_noise*<-0.5  +  rand(seeed4),-0.5  +
rand(seeed5),-0.5 + rand(seeed6)>;
 #declare  RotArray[i]  =  rot_mean  +  rot_noise*<-0.5  +  rand(seeed2),-0.5  +
rand(seeed3),-0.5 + rand(seeed4)>;
 #declare InRotArray[i] = <-180 + rand(seeed2)*360,-180 + rand(seeed3)*360,-180 +
rand(seeed4)*360>;
// vel_mean is the mean velocity
 #declare i=i+1;
// increasing the loop index only for properly positioned particle
 #end // if
#end //while
#declare flag_initial = 1;
// this is the flag which serves to initialize the array only once
save_array()
```

Treba uočiti nezgrapnost ovakvog rješenja. Naime kako je `my_clock` varijabla vezana uz rezerviranu `clock` varijablu, ona će postići vrijednost egzaktno jednaku nuli samo ako su broj frameova u animaciji, početni i finalni `clock` te `move_start` podešeni precizno, tj. tako da se na nekom frameu dogodi da je `#declare my_clock = clock - move_start;` upravo jednak nuli. Ovo se naravno može napraviti i bolje, to ostavljam svima koji čitaju. Za korisnike koji žele koristiti include file bez njegovog razumijevanja preporučujem da definiraju uvijek `#declare move_start = 0;` u glavnom kodu scene (objasnit ću ga poslije). Što se inicijalizacije tiče treba uočiti da su početni položaji definirani nasumično unutar cilindra radijusa `R_max` u x-y ravnini i dužine `z_max` u z-smjeru čije je težište translatirano iz ishodišta u `cent_box` točku (3D vektor). Također treba uočiti da

dodatna petlja po `j`-varijabli osigurava da udaljenost između pozicioniranih objekata bude minimalno `min_dist`. Ovo je napravljeno zbog izbjegavanja preklapanja objekata u nizu. Dodatna petlja povećava vrijeme proračuna, ali samo u frameu u kojemu se događa inicijalizacija. Početne brzine su definirane nasumično unutar intervala `[vel_mean – vel_noise*<0.5,0.5,0.5>, vel_mean + vel_noise*<0.5,0.5,0.5>]`. Slično je i za rotacione brzine. Slijedi nastavak `#if` naredbe koji opisuje što se događa u slučaju kad je `my_clock > 0`.

```
#else
#declare time_step = clock_delta;
//#warning concat ("Time step is ",str(time_step,5,4),"\n")
read_array()
#declare i=0;
union{
#while (i<N_part)
 #declare j=0;
 #while (j<N_time)
  #declare   force_x   =   -(potential(PosArray[i].x   +   0.0005,   PosArray[i].y,
PosArray[i].z) - potential(PosArray[i].x, PosArray[i].y, PosArray[i].z)) / 0.0005 ;
  #declare   force_y   =   -(potential(PosArray[i].x,   PosArray[i].y   +   0.0005,
PosArray[i].z) - potential(PosArray[i].x, PosArray[i].y, PosArray[i].z)) / 0.0005 ;
  #declare  force_z  =  -(potential(PosArray[i].x,  PosArray[i].y,  PosArray[i].z  +
0.0005) - potential(PosArray[i].x, PosArray[i].y, PosArray[i].z)) / 0.0005 ;
  #declare new_pos = VelArray[i]*time_step / N_time + 0.5 *
 <force_x/mass_part * pow(time_step / N_time,2),
  force_y/mass_part * pow(time_step / N_time,2),
  force_z/mass_part * pow(time_step / N_time,2)>;
  #declare new_vel = VelArray[i] +
 <force_x/mass_part * time_step /N_time,
  force_y/mass_part * time_step /N_time,
  force_z/mass_part * time_step /N_time>;
//clock_delta gives the time between the two frames (reserved PovRay word)
//mass_part is the particle mass (defined in the main file)

  #if (PosArray[i].y > min_plane)
   #declare PosArray[i] = PosArray[i] + new_pos;
   #declare VelArray[i] = new_vel;
  #else
   #declare PosArray[i] = <PosArray[i].x, min_plane, PosArray[i].z>;
   #declare VelArray[i] = <0.0, 0.0, 0.0>;
  #end
// #warning str(PosArray[i].x,7,3)
 #declare j=j+1;
 #end // end time loop
//updating positions and velocities here
 object{part_object   rotate   InRotArray[i]   +   RotArray[i]*my_clock   translate
PosArray[i]} //translate moves relative to current position
#declare i=i+1;
#end //while
} //end union
save_array()
#end //if
#end //main if end
```

Izvan include filea potrebno je definirati skalarno polje potential tj. funkciju koja ovisi o `x,y` i `z` koordinatama. Divergencija ove funkcije predstavlja prostorno ovisnu silu s koordinatama `<force_x, force_y, force_z>` (vidi kod iznad). Iz sile se jednostavnim Newtonovim zakonima gibanja dobiju inkrementi novi položaji i nove brzine niza čestica. Za ovo je potrebna masa čestica, `mass_part`. Ono što treba opaziti je da se unutar `clock_delta` intervala koji je predstavlja `clock` interval između dva uzastopna framea (rezervirana PovRay varijabla), Newtonovi zakoni gibanja izračunaju na vremenskoj podjeli

koja je finija za `N_time` puta tako da se povećanjem N_time parametra proračun (integracija jednadžbi gibanja) može po volji utočniti. Ono što se računa je trivijalno ubrzano/usporeno (akcelerirano) gibanje s poznatom početnom brzinom i koordinatom:

$$\boldsymbol{F} = -\boldsymbol{\nabla}\phi$$
$$\boldsymbol{a} = \boldsymbol{F}/m$$
$$\boldsymbol{r} = \boldsymbol{r}_0 + \boldsymbol{v}_0 * \Delta t + \boldsymbol{a}\Delta t^2/2$$
$$\boldsymbol{v} = \boldsymbol{v}_0 + \boldsymbol{a}\Delta t,$$

gdje je $\phi$ potencijal (`potential`), $\boldsymbol{F}$ sila (`force_x, force_y, force_z`), $\boldsymbol{a}$ akceleracija, $\boldsymbol{r}$ koordinata (`PosArray`), $\boldsymbol{v}$ brzina (`VelArray`), a $\Delta t$ vremenski inkrement (`time_step/N_time = clock_delta/N_time`). Samo finalni proračun se snimi u datoteki (pozivanje `save_array` macroa). Objekti koji se konačno iscrtaju na mjestu koordinata u nizu su definirani izvan include filea (`part_object`). Treba opaziti da je definiran parametar `min_plane` (realni broj) koji opisuje minimalnu y-koordinatu iznad koje se evolucija nastavlja, inače se čestica zalijepi kad dostigne (ili padne na) visinu `min_plane` i njena dinamika se zaustavlja. Rotaciona dinamika ne ovisi o vanjskom potencijalu tj. čestice se rotiraju jednolikim brzinama. I to je to što se include filea tiče. A sad evo i *.pov filea (eritrocit_guider.pov) koji opisuje scenu.

```
//------------------------------------------------------------------------
#version 3.5;
global_settings {  assumed_gamma 1 }
//------------------------------------------------------------------------
#include "colors.inc"
#include "textures.inc"
#include "glass.inc"
#include "metals.inc"
#include "golds.inc"
#include "stones.inc"
#include "woods.inc"
#include "shapes.inc"
#include "shapes2.inc"
#include "functions.inc"

#declare Radiosity=on;

global_settings {
  assumed_gamma 1.0
  //max_trace_level 25
  #if (Radiosity)
    radiosity {
      pretrace_start 0.08          // start pretrace at this size
      pretrace_end   0.04          // end pretrace at this size
      count 35                     // higher -> higher quality (1..1600) [35]
      nearest_count 5              // higher -> higher quality (1..10) [5]
      error_bound 1.8              // higher -> smoother, less accurate [1.8]
      recursion_limit 3    // how much interreflections are calculated (1..5+) [3]
      low_error_factor .5          // reduce error_bound during last pretrace step
      gray_threshold 0.0           // increase for weakening colors (0..1) [0]
      minimum_reuse 0.015          // reuse of old radiosity samples [0.015]
      brightness 1                 // brightness of radiosity effects (0..1) [1]

      adc_bailout 0.01/2
      //normal on                  // take surface normals into account [off]
      //media on                   // take media into account [off]
      //save_file "file_name"      // save radiosity data
      //load_file "file_name"      // load saved radiosity data
      //always_sample off          // turn sampling in final trace off [on]
      //max_sample 1.0             // maximum brightness of samples
```

```
    }
  #end
}

#default {
  texture {
    pigment {rgb 1}
    #if (Radiosity)
      finish {
        ambient 0.0
        diffuse 0.6
        specular 0.3
      }
    #else
      finish {
        ambient 0.1
        diffuse 0.6
        specular 0.3
      }
    #end
  }
}

light_source{<9,101.0,1000> color White}
light_source{<-9,-101.0,-1000> color White}
#declare Camera_0 = camera {angle 59          // front view
                           location  <0.0 , 0.1 ,-5.5> + <0,3.5*clock>
                           right     x*image_width/image_height
                           look_at   <0.0 , 0.1 , 1005.5>}

#declare Camera_1 = camera {angle 75
                           location  <3.0 , 0.0 ,-2.5>
                           right     x*image_width/image_height
                           look_at   <1.0 , 0.0 , -2.5>}
sky_sphere{pigment{color Black}}
camera{Camera_0}

fog {
fog_type 2
    fog_offset 500
    fog_alt 150
    distance 1000
    color rgb<0.02, 0.0, 0.0>}

#declare eritrocit = object{

lathe{  // rotates a 2-D outline of points around the Y axis to create a 3-D shape
  cubic_spline // quadratic_spline| cubic_spline| linear_spline
  // number of points,  list of <x,y> points, spline rotates around y-axis
  13,
  <-0.25,-0.07>, <0,-0.05>,<0.25,-0.07>,<0.5,-0.1>,<0.82,-0.15>,<0.95,-0.1>,<1,0>,
  <0.95,0.1>,<0.82,0.15>,<0.5,0.1>,<0.25,0.07>,<0,0.05>,<-0.25,0.07>
 }
 texture{pigment{color Red*0.7} normal{bumps 0.17 scale 0.3}}
}
// alternative version is below
//difference
//{
//merge{
//      cylinder{<0,-0.1,0>,<0,0.1,0>, 1}
//      torus {1.218,0.22}
//     }
//union{
//sphere{<0,0,0>, 0.9999 scale <1,0.06,1> translate<0,0.1,0>}
//sphere{<0,0,0>, 0.9999 scale <1,0.06,1> translate<0,-0.1,0>}
//}
//}
//};
```

```
#declare alvtex = texture{
pigment{
crackle
turbulence 0.14
        colour_map {
            [0.03 colour rgb<0.15, 0, 0> ]
            [0.25 colour rgb<0.95, 0.4, 0.4> ]
            [0.59 colour rgb<0.9, 0.2, 0.2> ]
            [1.00 colour rgb<0.8, 0.1, 0.1> ]

        }
scale 70
}
finish{
        ambient .4
        diffuse .2
        phong .01
        phong_size 25
}
};
#declare alvnorm = normal {
      dents 1
      normal_map {
        [ 0.5 bumps 0.8 scale .1]
        [ 1.0 ripples 0.8 scale .1]
      }
      scale 100
      };

#declare alveola = cylinder{<0,0,-15000>, <0,0,15000>, 500 open texture{alvtex}
normal{alvnorm} hollow };

object{alveola}

#declare N_part = 800;
#declare R_max = 20;
#declare z_max = 70;
#declare cent_box = <0,0,50>;
#declare vel_mean = <0,0,-25.5>;
#declare rot_mean = <90.5,-120.5,100.5>;
#declare rot_noise = 100;
#declare vel_noise = 0.1;
#declare mass_part = 9;
#declare part_object = eritrocit;
#declare move_start = 0.0;
#declare min_plane = -1000.0;
#declare N_time = 4;
#declare min_dist = 2.2;

#declare potential = function (x,y,z)
{
0.0
// no force at all...
};
#include "eritro_guide.inc"
```

Osim upotrebe `radiosity` opcije, *.pov file sadrži definiciju izvora svjetlosti, magle, tekstura, žile (`alveola`) i njene teksture (`alvtex`), normale (`alvnorm`), definiciju `eritrocit` objekta, ali ono što je najvažnije nalazi se na kraju *.pov filea. Ovdje se nalaze parametri koji su potrebni include fileu i to:

| | |
|---|---|
| `N_max` | – ukupni broj čestica |
| `R_max` | – radijus cilindričnog kontejnera čestica |
| `z_max` | – dužina cilindričnog kontejnera |

`cent_box` – položaj težišta cilindra čija je visina duž z-osi
`vel_mean` – srednja početna brzina po ansamblu čestica
`rot_mean` – srednja rotaciona brzina po ansamblu
`rot_noise` – šum u rotacionoj brzini, max. odstupanje
`vel_noise` – šum u translacionoj poč. brzini, max. odstupanje
`mass_part` – mase čestica (sve čestice imaju istu masu)
`part_object` – objekt koji se pozicionira u koordinate niza čestica
`move_start` – početni trenutak za animaciju
`min_plane` – y-koordinata «ljepljivog tla», visine na kojoj se čestice zalijepe
`N_time` – broj iteracija Newtonovih jednadžbi unutar `clock_delta` intervala
`min_dist` – minimalna udaljenost između koordinata u nizu čestica
`potential` – funkcija koja opisuje prostornu ovisnost potencijala

Treba uočiti da se include file poziva (`#include "eritro_guide.inc"`) *tek nakon definiranja svih parametara koji su mu potrebni*. U ovoj implementaciji vanjska sila iščezava (`potential` je nula) što znači da se tijela gibaju jednoliko; tako mi se činilo zgodno, ali naravno da include file radi i za neiščezavajuće vanjske potencijale.

Evo i jednostavnog *.ini filea (eritrocit_guider_ini.ini) koji sadrži animacijske parametre i koji se kompajlira PovRayem:

```
; Persistence Of Vision raytracer version 3.5 sample file.
Antialias=On
Antialias_Threshold=0.25
Antialias_Depth=3

Input_File_Name=eritrocit_guider.pov

Initial_Frame=1
Final_Frame=200
Initial_Clock=0
Final_Clock=2.0

Cyclic_Animation=off
Pause_when_Done=off
```

Ono što bi se dodatno moglo napraviti je da se u vremenskoj evoluciji ugrade sudari među česticama. Ovaj zadatak bi mogao biti kompliciran jer bi realistični sudari morali ovisiti o obliku objekta. Za sferne objekte sudare bi bilo jednostavno implementirati. Dodatno, mogla bi se uvesti skala mase, vremena i prostora, tako da npr. 1 jedinica `clock`-a bude određeni broj sekundi (a ne 1 sekunda u prikazanoj implementaciji).

Cijela prikazana priča je podvrsta onoga što se u raytracingu zove «*particle systems*». Skupina čestica koje se ponašaju prema određenim pravilima često se koristi za simulacije dima, tekuće vode (kapljica), vatre i slično, a ja sam sličnu stvar koristio i u simulaciji depozicije atoma na površinu vođene laserskim (svjetlosnim) poljem (litografija, vidi http://nanoatlas.ifs.hr/images/litho_movie.wmv). Include file koji je predstavljen u ovom tekstu nastao je upravo na osnovu include filea korištenoj u toj animaciji.