3D ábrázolás a PoVRay segítségével (1. rész)

A számítógépek megjelenése előtt is már voltak törekvések a valós tárgyak ábrázolására két dimenziós felületen, gondoljunk csak az ókori és középkori freskókra és festményekre. A mérnökök is szerették az általuk használt műszaki rajzolásban különféle eljárásokat használni, amelyekkel már a tervezőasztalon elképzelhető lett egy-egy munkadarab. A két dimenziós felületeken leképezett három dimenziós világnak nagy hagyománya van, bármennyire is törnek előre a valódi 3D megjelenítők.

Alapelvek

Minden matematikai és technikai elvet mellőzve annyit lehet kijelenteni, hogy a 3D ábrázolásnak kettő elterjedt módszere van, amelyeket a különféle megjelenítő programok használnak: a ***poligon alapú ábrázolás*** és a ***sugárkövetés***.

A poligonok ábrázolása során a számítógép általában háromszögekből készíti el az ábrázolni kívánt test felületét, s a műszaki rajzok ábrázolásából szivárgott át más területekre is. Egy kockához hat négyszög kell, azaz 12 háromszög, így viszonylag kevés koordinátával leírható lesz a test. A síklapok térbeli elhelyezése nagyon kevés számítással elvégezhető. Egy gömb ábrázolásához viszont már sok száz-sok ezer háromszöget kell meghatározni, csak a sík felületek leírásához célszerű poligonokból összeállítani egy-egy testet. Kétségtelen, hogy a számítógépes játékok elterjedése a poligonok ábrázolása révén vált lehetővé, mivel ez a módszer kevesebb számítási erőforrást igényel, viszont az elkészült „világ" is szögletes és elnagyolt lesz. Ha növeljük a poligonok számát, a minőség és a szögletesség is lassan eltűnik, viszont – mivel felületekről van szó – négyzetesen növekszik a szükséges számításigény. Különféle trükkök szükségesek, hogy a fénybe helyezett tárgy árnyékot vessen (***shading***), vagy a felületén az egyenetlenségeket észrevehetően ábrázoljuk (***bump mapping***). Ha tükröződés is cél, akkor a poligonok ábrázolása már kevés lesz, kénytelenek leszünk némi sugárkövetést is alkalmazni.

A sugárkövetés szinte egy időben jelent meg a számítógépekkel, mivel pontos és összetett számolások ismételgetése szükséges egy kép összeállításához. A sugárkövetés valójában a létező fizikai világ modellezése, a fényforrásból induló különböző energiájú (színű) fotonok a tárgyakon áthaladnak, elnyelődnek, újra kibocsátódnak, szóródnak, visszaverődnek, módosulnak és végül egy nagyon kis részük a szemlélődő ember szeméig jutnak el. A sugárkövetés ezen okból fordítva működik a számítógépes valóságban, ugyanis pazarlás olyan sugarakat követni, amelyek nem jutnak el a megfigyelőig. Ennek megfelelően az egész eljárás vektorok kezeléséből áll, a nézőpontból (kamera) a képernyő felbontásának megfelelő számú „látósugár" indul meg a virtuális világ felé, ahol kölcsönhatásba lépnek a testek modelljeivel, s végül egy fényforrásba vagy a végtelenbe jutnak el. Nagy felbontás és sok test esetén többmilliárd visszaverődés és szóródás is lehetséges, illetve több fényforrásban is „véget érhet" egy-egy sugár élete. Megfelelő számítási kapacitás hiányában a sugárkövetés nem alkalmas valós idejű 3D mozgások ábrázolására, így a számítógépes játékok sem tudják kihasználni a tudását; a számítógéppel készített filmek és egyéb tévéjátékok viszont előszeretettel alkalmaznak sugárkövetést is, hiszen van elegendő idejük a használatára.

PoVRay – Persistence of Vision Raytracer

David K. Buck és ***Aaron A. Collins*** által készített ***DKBTrace 2.12*** program alapján készült el a ***PoVRay*** első verziója, amelyet szakemberek tucatjai fejlesztenek azóta is, s a forrása is hozzáférhető bárki számára.

A programnak saját speciális leíró nyelve van, amelyet ***SDL*** (***Scene Description Language***) rövidítéssel illetnek. Kismértékben hasonlít a C nyelvhez – minden bizonnyal a C nyelv volt a minta, de az a hasonlóság csak nagyon távoli rokonság.

A program maga csak parancssoros felülettel rendelkezik, bár van sok frontend hozzá, amelyekkel kényelmesen tudjuk használni (***Kpovray***, ***Blender***, stb).

Első feladatként készítsünk egy zöld színű gömböt, amelyet egy pontszerű fehér fényforrás világít meg megközelítőleg a kamera felől, s a kamera pont a gömbre figyel.

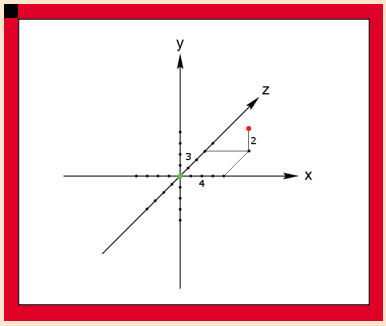
```
#include "colors.inc"

sphere{
 <0, 0, 0>, 2
 texture{
  pigment{
   color Green}}}
```

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

camera{
 location <0,2,3>
 look_at <0,0,0>}
```

Nevezzük *pov01.pov* néven, s helyezzük bele egy tetszőleges könyvtárba, amelyet arra a célra fogunk szánni, hogy képeket és *PoVRay* „programokat” helyezünk el benne. Ezt a fájlt le kell „fordítanunk” ahhoz, hogy képet kapjunk kimenetként. Ez a fordítási művelet ***renderelés*** álnéven dolgozik, s a következő parancssorral tudjuk előcsalogatni:

```
povray +L/usr/local/share/
➥povray-3.6/include/ +H600
➥+W800 +Ipov01.pov +Opov01
```

Figyeljünk arra, hogy a `+L` paraméter után a *PoVRay* tényleges elérési útját adjuk meg, ez ugyanis minden disztribúcióban más lehet. A `+H` paraméter a kép magasságát, a `+W` a kép szélességét határozza meg, a `+I` utáni szöveg határozza meg a lefordítandó állomány elérési útját, s végül a `+O` a készítendő kép neve lesz. Alapértelmezésben *png* típusa lesz a képnek (régebben *tga* formátumot használtak a készítők), de ezt felül lehet bírálni, s a program szinte az összes gyakori képformátumot képes felhasználni. Ha a grafikus felületünk elérhető a program számára a futásának idejében, akkor egy ablakot kapunk, ahol követhetjük a kép készítésének menetét.

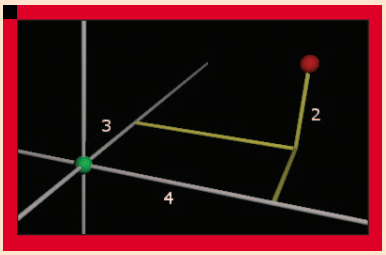


A 3D világ olyan koordináta rendszert kíván meg, amellyel meghatározhatunk pontokat a térben. Erre a célra a műszaki ábrázolás már kiötlött módszereket: egyszerűen a kettő dimenziós *x-y* koordináta rendszert kiegészítették egy harmadik – *z* jelű – tengellyel is, amely a megjelenítés síkjától tart tőlünk a végtelenbe. Egy pont megadása tehát három szám segítségével történik, mégpedig rendre az *x*, az *y* és a *z* koordinátával. Ha egy piros gömböt szeretnénk meghatározni a tér x=4, y=2 és z=3 pontjára, akkor ezt a *PoVRay* formátumában a

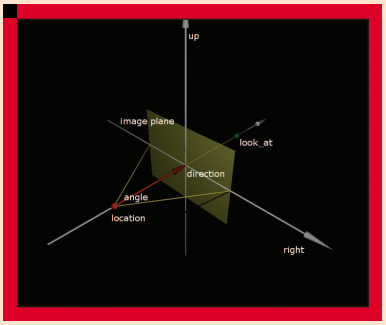
```
sphere{
 <4, 2, 3>, 0.2
 texture{
  pigment{
   color Red}}}
```

SDL részlettel tudjuk megoldani. Sokkal látványosabb lehet ez a *PoVRaY* által készített képet nézve amely egy példaprogramnak is tekinthető.

```
#include "colors.inc"

sphere{ // Zöld gömb az
        ➥origóban
 <0, 0, 0>, 0.2
 texture{
  pigment{
   color Green}}}

sphere{ // Piros gömb a
        ➥megadott pozícióban
 <4, 2, 3>, 0.2
 texture{
  pigment{
   color Red}}}

[...]

cylinder{ // A Z tengely
 <0,0,-10>, <0,0,10>, 0.05
 pigment{
  color White}}

[...]

cylinder{ // 3 egység hosszú
        ➥szakasz, Z
        ➥tengellyel párhuzamos
 <4,0,0>, <4,0,3>, 0.05
 pigment{
  color Yellow}}

camera{
 location <5,5,-10>
 look_at <0,0,0>}

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



A *pov02.pov* példa alapján – feltéve, ha erős mérnöki vizuális látással bírunk – azonnal „élni” látjuk a számokat. Ha pusztán az *SDL* fájlt szeretnénk használni, mindenképpen szükségünk lesz egyfajta belső látásra, amely biztosan kifejlődik a próbálkozások során. Nézzük tehát a virtuális világ alkotóelemeit, kezdjük a kamerával.
A *PoVRay* kamerája (*kamera.pov*) egyszerű felépítéssel bír, amely a látható képet is meghatározza. Bármerre is legyen a kamera (`location`), mindig van egy vektor (`direction`), amely meghatározza a képsík (`image plane`) középpontját. Ha megadunk nézőpontot (***`look_at`***), akkor a kamera átszámolja az irányvektorát, és a képsík közepén az a tárgy lesz, amelyet a nézőpont meghatároz. A képsík mérete alapesetben egységnyi, szélességét és magasságát közvetlenül a magasságvektor (`up`) és a jobbra mutató vektor (`right`) határozza meg, s ebből adódik a látószög (`angle`), amelyet megadunk közvetlenül is, ám ekkor módosulnak az egyéb vektorok is.



A program akkor is használ kamerát, ha erre külön nem utasítottuk. Az alapértelmezett kamera *SDL* leírója mindig felhasználásra kerül, s a saját kameránk csak felülírja pár alkotóelemét:

```
camera{
 perspective
 location <0,0,0>
```

```
 direction <0,0,1>
 right 1.33*x
 up y
 sky <0,1,0>
}
```

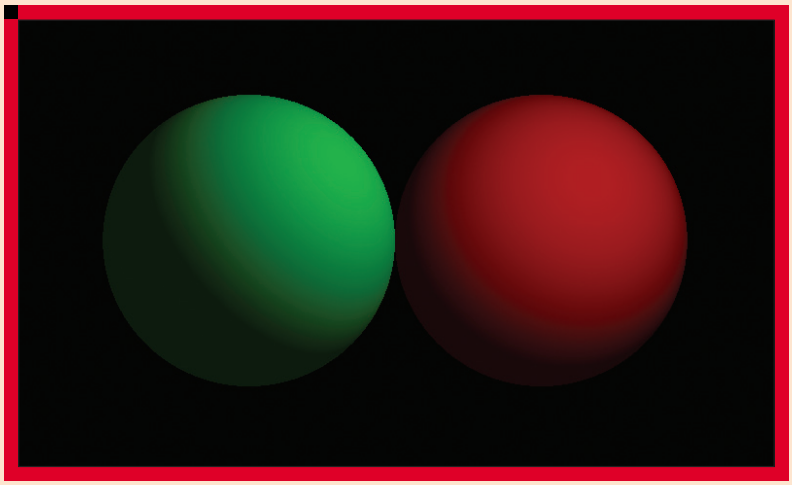
Az első kulcsszó jelenti, hogy ez a kamera (illetve az egész sugárkövetés) perspektívát is ábrázol, amely abban nyilvánul meg első látásra, hogy a távolabbi objektumok kisebbek, mint a közelebbi objektumok. Ezt átkapcsolhatjuk ortografikusra, és már teljesen más jellegű képet kapunk, amelyet inkább mérnöki ábrázoláshoz tudunk használni.

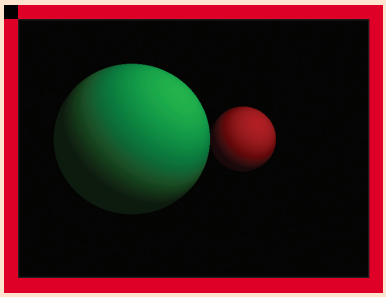
```
camera{
// orthographic
 location <0,0,-4>
 look_at <0,0,0>}
```

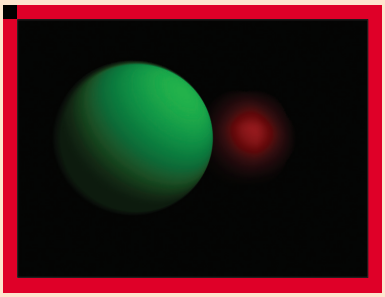
A *pov03.pov* által tartalmazott állományban az `orthographic` kulcsszó előtti komment jelet kitörölve-visszaírva a két vetítési mód között tudunk váltogatni. A vetítési módok között az a különbség, hogy a perspektíva esetén a végtelenben egy pontba futnak össze a nézetvonalak, az ortografikus esetén pedig a vetítősíkkal párhuzamos vetítővonalakkal számol a program. A két ábrán a vörös gömb mérete eltérőnek mutatkozik, holott a zöld és a vörös gömb mérete azonos, csak a vörös gömb hátrébb foglal helyet. A perspektívatorzítás miatt ezért kisebb mérete lesz (ezt szoktuk meg a valóságban is), mérnöki ábrázolásnál azonban nem szokás figyelembe venni a perspektívát, így a két gömb azonos méretűnek ***tűnik***. Mivel a *PoVRay* alapvető felhasználása a fotorealisztikus képek készítése, így a perspektíva az alapértelmezése, viszont vannak olyan mérnöki programok, amelyek az ortografikus ábrázolást igénylik.

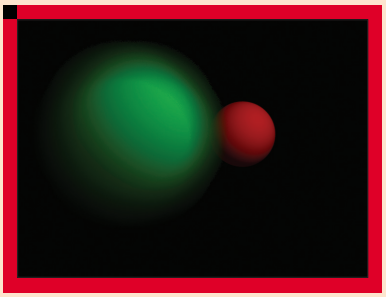
A perspektíva alapú kamera tovább állítható, például halszem optikát tudunk rátenni a kameránkra, amelynek az a lényege, hogy legalább 180° a látószöge. Eléggé furcsa élmény egy ilyen optika, viszont sok állat rendelkezik ilyen látással (a halakon kívül a madarak nagy része is), ugyanis ilyen szemmel mindent látnak, s ez a túlélésük tekintetében létfontosságú. *PoVRay* esetén a `fisheye` kulcsszó kell a kamera leírójába, s meg kell adni a szöget is, amely lehet akár 360° is (*pov04.pov*). További kameratípusok is találhatók a *PoVRay* eszköztárában, azonban ezek ritkán használatosak, némelyik pedig igen speciális célokat szolgál. Érdemes mindegyik vetítési módot kipróbálni, bár az eddig említett három mód tökéletesen elég az átlagos használathoz.

Ha megadunk egy kamerapozíciót és egy nézőpontot, akkor még alig határoztunk meg a kamerát. Kettő pont ugyanis egy vektort jelent csak, s meg kellene adnunk a kamera állását is, hiszen tarthatjuk akár fejjel lefelé is. Erre a célra szolgák a ***sky*** kulcsszó, amely a „felfelé” irányt adja meg. Bármilyen pontot meghatározhatunk a térber, a program ettől a ponttól egy merőlegest próbál húzni a már meghatározott szakaszra és ezzel a merőleges vektorral már meg tudja határozni a képsík pozícióját. Alapesetben az ***y*** koordináta lesz a képsík „felfelé” iránya.









A látószöget (***angle***) alapvetően meghatározza a kamera iránya (***direction***), s az a tény, hogy a képsík kiterjedése alapesetben egy egységnyi négyszög. Ha közvetlenül megadjuk a látószöget, akkor a képsík mérete nem fog megváltozni, az irányvektor hossza módosul. Ez gyakorlatilag azt jelenti, hogy az irányt még azelőtt kell meghatározni, mielőtt szöget adunk meg, mert látószöget módosítja, ha utána adjuk meg az irányvektort.
A képsík relatív méreteit az `up` és a `right` kulcsszó határozza meg, s alapesetben a képernyő felbontásához van „torzítva". A szélesség 4/3 szorzóval szélesebb, mint a magasság. Ha eltérő arányokat szeretnénk (például A4 méretű papírra vagy 10x15 fotópapírra dolgozunk), akkor például az

```
up y
right 1.5*x
```

sorokat kell a kamerában elhelyezni. Ez az arány minden esetben egyezzen meg azzal az aránnyal, amekkora aránnyal a képet elkészítjük, ellenkező esetben a képen torzulás fog jelentkezni. Ez főleg abban jelentkezik, hogy a gömbök vetülete nem kör lesz, hanem ellipszis.

Alapesetben kamera végtelen mélységélességgel rendelkezik, amely egy kissé ront a valóságérzeten, hiszen minden nem lehet egyszerre éles (hacsak nem lyukkamerát használunk :). Ezen a problémán tudunk segíteni a fókuszpont meghatározásával, amely általában a kamera nézőpontja is egyben. A fókuszpont környéke szokott éles lenni, s az élességtartomány méretét nevezzük mélységélességnek. Minél nagyobb ez a jelzőszám, annál nagyobb távolságot látunk élesnek. Ennek eléréséhez néhány egyéb beállítást is el kell végeznünk, amelyek közül az apertúra (***aperture***) jelenti a legfőbb paramétert. Ez tulajdonképpen a kamera optikáján található nyílás méretét jelenti, s a valósághoz hasonlóan a nagyobb átmérő kisebb mélységélességet jelent. Ha nem definiáljuk, akkor ennek az értéke nulla, amely végtelen mélységélességet jelent: minden pengeéles lesz, legyen az pár egységnyire vagy a távoli végtelenben. A megjelenítés minőségét a felhasznált fénysugarak számával (***blur_samples***) tudjuk befolyásolni: a 10 még elnagyoltnak és gyengének tűnik (de a kép készítése közben megfelelő kompromisszum lehet), 500 fölötti szám már feleslegesen szép eredményt ad. Természetesen minél nagyobb számot adunk meg, annál több időbe telik a kép elkészítése.

```
camera{
 location <0,0,-4>
 aperture 2.0
 blur_samples 500
 focal_point <0,0,0>
 look_at <0,0,0>}
```

A ***pov05.pov*** egyszerű másolata a harmadik példának, viszont a kamera esetén megadtuk a szükséges adatokat, amelyekből a program már képes mélységélességet számolni. A bal oldali kép esetén a zöld, s a jobb oldali esetén a vörös gömb került a fókuszba. Számítsunk arra, hogy a fotorealisztikus mélységélesség esetén – a példában is látható – két gömböt tartalmazó világ renderelése akár 1-2 percet is igénybe vehet egy korszerű (2-3 GHz-es processzorral szerelt) gép esetén!
A következő részben a világítás és a különböző fényforrások részletes beállításaival fogunk foglalkozni, illetve a program fontosabb parancssori kapcsolóival.

Auth Gábor
(auth.gabor@enaplo.hu)
Egy pécsi középiskolában informatikát és programozást oktat. Tíz éve botlott először a UNIX rendszerekbe, 7 év Linux használat után kapta el a FreeBSD lázat, amiből máig nem tudott kigyógyulni.

KAPCSOLÓDÓ CÍMEK

A PovRay projekt honlapja
➲ http://www.povray.org