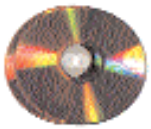
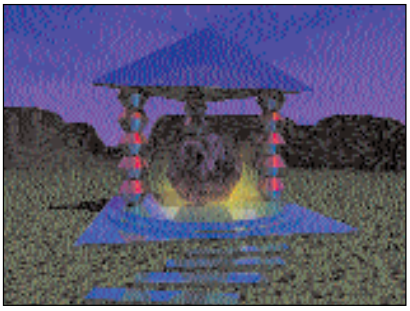


PoV-Ray ismeretek (3. rész)

Ebben a részben az objektumok leírásának további módszereivel foglalkozunk.

Most részletesebben tárgyaljuk a Bezier-patch modelleket, a forgástesteket, a magassági térképek alapján készített domborzatokat, a metaball-objektumokat és a háromszög által meghatározott objektumokat. Az olvasottak elsajátítása után – feltéve, hogy a korábbi részeket is „befogadtuk” – elvileg képesek leszünk az alábbi képhez hasonlót alkotni.



A látvány megéri a fáradságot…

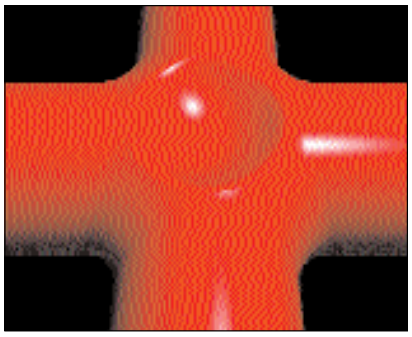
Kezdjük a metaball-objektummal, amit a PoV szóhasználatában egyszerűen *blob*-nak nevezünk. Aki használta valaha a LightWave valamelyik változatát, annak szeme előtt nyilván megjelenik szóban forgó objektumunk képe, a többiek pedig képzeljenek el pontszerű, egypólusú mágneses tereket, amelyek tetszőleges helyzetben egymás mellé vannak téve. A mágneses terek által alkotott eredő mágneses tér határozza meg a *blob* objektum formáját. Természetesen a „mágneseket” elmozdíthatjuk, elérve, hogy a tárgy formája rugalmasan változzék.
Ilyen objektumokat a PoV-Ray leírónyelvével a következő formában adhatunk meg:

```
blob {
     threshold THRESHOLD_ RT K
     cylinder { <EGYIK_V GE>,
     ➥<M`SIK_V GE>, SUG`R,
     ➥[ strength ] VONZER  }
     sphere { <K ZEPE>, SUG`R,
     ➥ [ strength ] VONZER  }
  }
```

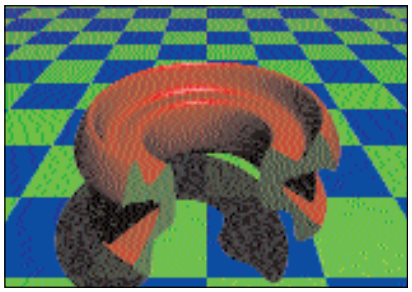
Az objektum látható részeinek kiszámítása során a PoV-Ray ott határozza meg a felület egy pontját, ahol az összetevők által alkotott erőtér és a TRESHOLD_ÉRTÉK megegyezik. Az egyes összetevők által keltett mágneses mező erőssége az összetevő középpontjában a VONZERŐ értékkel egyezik meg, és az objektum határáig folyamatosan csökken nullára. A VONZERŐ értéke negatív is lehet, ekkor a mágneses mező objektumunk felületén nem kidudorodásként jelenik meg, hanem horpadásként.
Itt kell megjegyeznem, hogy egy objektum több összetevőből is állhat, melyeket szabadon torzíthatunk és átméretezhetünk.
A *blob* objektumot gömbökből és hengerekből építhetjük fel; ezen építőelemek segítségével olyan változatos formákat alkothatunk, melyek összetettségének csupán alkotóerőnk szabhat határt. Ezekből a formákból akár egészen bonyolult élőlényt is teremthetünk: a Jurassic Park közkedvelt dinoszaurusza, a Tyrannosaurus Rex szintén metaball-modellezéssel készült, noha nem PoV-Rayjel.
A *lathe* objektum (magyar elnevezése a matematikaórákról ismerősen csengő „forgástest”), melynek segítségével egyszerűen hozhatunk létre vázákat, kelyheket, oszlopokat és hasonló tárgyakat. Az objektum megadásának módját az alábbi lista mutatja:

```
lathe {
     [ linear_spline |
     ➥quadratic_spline |
     ➥cubic_spline ]
     PONTOK_SZ`MA,
     <PONT_1>, <PONT_2>, ...,
     ➥<PONT_PONTOK_SZ`MA>
  }
```

A PONTOK_SZÁMA érték mutatja meg, hány ponttal kívánjuk a forgástestet leíró görbét megadni. Ha a pontok koordinátáit kétdimenziós vektorokként írjuk be, a program a pontokat *lineáris* (linear_spline), *négyzetes* (quadratic_spline) vagy *köbös* (cubic_spline) spline-görbékkel köti össze, és az y tengely mentén forgatva létrehozza a testet. A fentebb említett görbék részletesebb leírását nem kívánom ismertetni, akit bővebben érdekel matematikai előállításuk módja, a könyvtárakban könnyen utánanézhet. Itt elegendő annyit tudnunk, hogy lineáris spline-görbék előállításánál a két pont közti görbeszakasz csak ettől a két ponttól függ, míg a négyzetes és a köbös görbék esetében a pontok közti szakasz alakját a szomszédos pontok is befolyásolják, módosításukkal tehát finomabb formákat hozhatunk létre.



A negatív erő hatása a blob objektumra



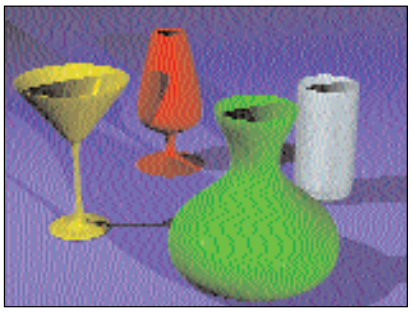
Egyszerű forgástest keresztmetszete

Azt is érdemes megjegyeznünk, hogy a PoV-Ray nem zárja le önműködően a görbét, ha ezt meg akarjuk tenni, az első és az utolsó pontot – azonos koordinátákkal – nekünk kell megadnunk.
A következő objektum, amit tárgyalni fogunk, a *Surface of revolution* – a kifejezést magyar fordításban szintén „forgástestként” adhatjuk vissza. PoV-Ray-beli neve *sor*, és a következő módon hozható létre:

```
sor {
     PONTOK_SZ`MA
     <PONT0>, <PONT1>, ...,
     ➥<POINTPONTOK_SZ`MA-1>
     [ open ]
  }
```

Az értékek megegyeznek az előbbi objektumnál látottakkal, különbséget csupán a testek matematikai előállításának módjában fedezhetünk fel. Ugyanebből a különbségből származik az a tény is, hogy ezek az objektumok kevésbé rugalmasak, mint a *lathe*-objektumok. Ennek oka, hogy min-

den y értékhez csak egyetlen x érték tartozhat. A PoV-Rayt az `open` kulcsszóval utasíthatjuk, hogy a létrehozott test végeit ne zárja le. Ilyenkor a CSG-műveletekben ne használjuk a létrehozott objektumot, mert – a PoV-Ray leírása szerint – az eredmény nem lesz a várakozásainknak megfelelő. Ezzel a megadási móddal zárt görbéket nem határozhatunk meg.



Változatok egy témára

Felmerülhet a kérdés, hogy miért szükséges ez az utasítás? A választ szintén a matematika adja meg: a *lathe*-objektumok kiszámítása során a fénysugár és a test metszéspontjának meghatározásához egy hatodrendű polinomot kell megoldanunk, míg ugyanennek a műveletnek az elvégzéséhez – egy *sor* (Surface of revolution) objektum esetén – egy harmadrendű polinom megoldása is elegendő. A sebességbeli különbség pedig magáért beszél.
A bevezetőben nem említettem ugyan, de szeretnék bemutatni egy nagyon hasznos objektumot, melynek segítségével egyszerűen állíthatunk elő látványos hatásokat, például logókat saját honlapjaink díszítésére. A szövegről, avagy PoV-Ray-es szóhasználattal élve a *text*-ről van szó. Létrehozása igen egyszerű:

```
text {
     ttf "BET K SZLET.TTF",
     "A SZ VEG",
     VASTAGS`G, <ELTOL`S_VEKTOR>
   }
```

A BETŰKÉSZLET.TTF érték határozza meg a használni kívánt betűkészletet, ami jelenleg csak TrueType formátumú lehet. A szöveget ezt követően a C nyelv karakterlánc meghatározási szabályai szerint adhatjuk meg : macskakörmök között. Ez például azt jelenti, hogy ha a kiszámolt képen ' karaktert szeretnénk viszontlátni, akkor az objektum létrehozásakor a szöveg értékben az idézőjelet a következő módon kell meghatároznunk: "\"".
Így a PoV-Rayt arra utasítottuk, hogy a '\' utáni karakter a szöveghez tartozik, és nem annak lezárására szolgál.

A szöveg origója az első betű bal alsó sarkában lesz, a további betűk pedig az x tengely pozitív irányának megfelelően következnek. Az egyes betűk vastagságát a VASTAGSÁG valós szám értéke határozza meg. A betűk alapértelmezésben 1 egység magasak lesznek azért, hogy a .TTF fájlban meghatározott betűközök valószerűek legyenek. Ezt a betűközt felülbírálhatjuk az ELTOLÁS_VEKTOR-ral.
A *text* objektumok kiszámításakor csak a nyomtatható karakterek fognak megjelenni, tehát a *soremelés*-nek, a *tabulátor*-nak és a hozzájuk hasonló vezérlőkarakntereknek a megjelenő szövegre semmiféle hatásuk nem lesz.
A fenti kitérő után visszatérnék a cikk fő témájához, most a domborzati képek megjelenítéséhez elengedhetetlen *height_field* objektumot mutatom be. Ha rendelkezésünkre áll egy olyan domborzati térkép, amelyen a magassági szinteket különböző, egyre világosabb színekkel jelölik, akkor a *height_field* segítségével könnyen és gyorsan alkothatunk tájképeket. Amint a kép elkészült, alakítsuk át a következő képformátumok valamelyikére: gif, pgm, ppm, png, tga, pot. A gif formátum sajátossága, hogy csak 256 színárnyalat tárolására alkalmas. Ebben az esetben domborzatunk nem a színek alapján fog elkészülni, és a domborzati képen létrejövő kiemelkedés az adott képpont (pixel) palettaindexének megfelelő magasságú lesz.
Ezt jól szemléltethetjük egy példával: tegyük fel, hogy a felülnézetből készített gif-képen egy tó látható, amely sötétkék színű, körülötte a hegyek és az erdők világoszöldek. Ekkor joggal feltételezhetjük, hogy a PoV-Ray az elkészített domborzaton a tavat mély területként, az erdőket, hegyeket magasabb területként fogja ábrázolni. Meglepetés érhet minket, ugyanis a gif-állományban a sötétkék színhez magasabb index tartozhat, mint a világoszöldhöz, így domborzatunk végeredményképpen teljesen valószerűtlen tájat mutathat. Ez a hiba nem fordulhat elő, ha a gif formátum 16-bites változatát használjuk (a .POT formátumot), ennek előállítására a FractInt program alkalmas. (A program lemezmellékletünkön is megtalálható, de ezzel a változattal nem lehet pot formátumú képeket létrehozni.) Ilyen gondok a nem indexelt képekkel nem fordulhatnak elő, mert ott a program a magassági értékeket az RGB-összetevők alapján számítja ki.
Ennyi bevezetés után lássuk, hogyan hozhatjuk létre magát a domborzatot:

```
height_field {
  F`JL_T´PUS "F`JLN V"
     [ smooth ]
     [ water_level VAL S ]
  }
```

Itt a FÁJL_TÍPUS a fent említettek valamelyike lehet. A *smooth* kulcsszóval adhatjuk meg, hogy a PoV-Ray alkalmazzon-e simítást a számolás során, míg a *water_level* a tengerszint magasságát határozza meg. A program tengerszint alatti területekkel nem számol.
A megjelenő kép domborzatunktól függetlenül 1x1x1 egység méretű lesz, tehát nagyobb felbontású képek használatával nem hozhatunk létre nagyobb objektumot, viszont növelhetjük a részletességét.
A következő sorok begépelése után a *hf_gray_16* kulcsszó használata a globális beállításoknál szintén részletesebb képeket eredményez:

```
global_settings {
       hf_gray_16 on
}
```

Az objektumok létrehozása során a képek 16-bites színmélységűek lehetnek és a program is eszerint fogja használni ezeket. A PoV-Ray legújabb (3.1g) változata sajnos e ponton nem adott élvezhető eredményt, a kép színei egyáltalán nem feleltek meg a leírófájl alapján várhatóknak.
A most következő objektum teszi lehetővé, hogy a PoV-Rayben régebben elkészített háromdimenziós modelljeinket is felhasználhassuk. Nem másról van szó, mint a háromszögek által meghatároott, a PoV-Rayben *mesh*-nek nevezett tárgyról, mely a következő formában adható meg:

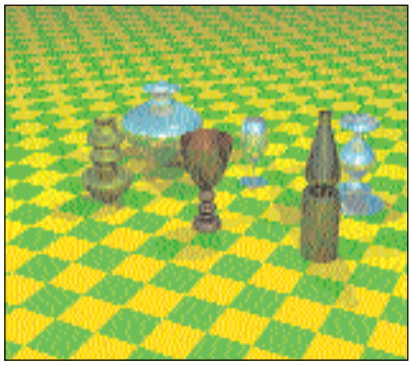
```
mesh {
  triangle {
    <CS CSPONT1>, <CS CSPONT2>,
    <CS CSPONT3>
    [ texture { MINT`ZAT_NEVE } ]
  }
  smooth_triangle {
    <CS CSPONT1>, <NORM`L1>,
    <CS CSPONT2>, <NORM`L2>,
    <CS CSPONT3>, <NORM`L3>
    [ texture { MINT`ZAT_NEVE } ]
  }
```

A fenti listából látható, hogy tárgyunkat kétféle háromszög alkothatja. Összetett objektumok esetén a görbefelületeket szintén háromszögekből építjük fel, de ahhoz, hogy az eredmény sima görbefelület legyen, nagyon sok kisméretű háromszög szükséges. Ezen segíthetünk a *smooth_triangle* alkalmazásával, ahol minden csúcspontnak megadhatjuk a normálvektorát is (ezt egyébként a program számítaná ki). Mivel egy háromszög vala-

melyik csúcspontja más háromszögekhez is tartozhat, nem egyértelmű, hogy az adott pontnak melyik lesz a megfelelő normálvektora, ezáltal a kiszámított képen nem kapunk sima görbületeket. Ha viszont mi határozzuk meg a csúcspont normálvektorát, akkor az eredmény is ránk van bízva. Természetesen a *mesh* objektumok felépítése során nem kell bonyolult számításokat végeznünk, hiszen ezeket az objektumokat általában valamilyen segédprogrammal állítjuk elő, a meglévő modellek átalakításával. E programok az egyes csúcspontok normálvektorát rendszerint a csúcsponthoz tartozó háromszögek normálvektorainak átlagolásával állítják elő. Ilyen átalakítóprogramot találhatunk CD-mellékletünkön is, melynek segítségével LightWave-objektumokat alakíthatunk át a PoV-Ray által ismert formába. Más operációs rendszerekhez több ilyen program is a rendelkezésünkre állhat.
Minden 3D modellezőprogram képes dxf formátumban menteni a modelleket, így ha valaki például a Blender-modellezés képességeit szeretné ötvözni a PoV-Ray sugárkövetéssel számított képeinek valószerűségével, kedvenc modelljeit a megfelelő program birtokában könnyedén átalakíthatja.
Utolsó témánk a *patch*-objektumok létrehozásának és használatának taglalása. A PoV-Ray szóhasználatával élve ezt az objektumtípust *bicubic_patch*-nek nevezzük, amit magyarul talán *folt*-objektumként adhatunk vissza. Ezek a tárgyak 3D-görbék által meghatározott felületek, ahol úgynevezett *vezérlőpontok* (controll point) alakítják ki a test felületét. Hasonlóan a Bezier-spline görbékhez, amelyekről a forgástesteknél olvashattunk, a felületet nem csupán a szomszédos pontok alakítják ki, hanem azok szomszédai is. Ilyen tárgyat a következő utasításokkal hozhatunk létre:

```
bicubic_patch {
     type PATCH_T˝PUSA
     flatness FLATNESS_ RT K
     u_steps U_L P SEK_SZ`MA
     v_steps V_L P SEK_SZ`MA
   <CP1>, <CP2>, <CP3>, <CP4>,
   <CP5>, <CP6>, <CP7>, <CP8>,
   <CP9>, <CP10>,<CP11>,<CP12>,
   <CP13>,<CP14>, <CP15>, <CP16>
 }
```

A PATCH_TÍPUSA jelenleg kétféle értéket vehet fel. Ha az értéke 0, a PoV-Ray csak a *vezérlőpont*-okat tárolja a memóriában, így sokkal kevesebb memória szükséges, viszont a tárgy ábrázolásakor több számítást kell végezni. Ha az érték 1, a program előzetes feldolgozásként több apró foltra bontja a tárgyat, ezáltal csökken az ábrázolás számításigénye, de a memóriaigénye növekszik.
A következő értékeknek a típus meghatározása után a listán látható sorrendben kell követniük egymást. Ezekután kell megadnunk azt a 16 *vezérlőpontot* (háromdimenziós vektorként), amelyek végső soron a test alakját adják. Elkészült tárgyunk felülete mindenképpen érinteni fogja a <CP1>, a <CP4>, a <CP13> és a <CP16> pontokat.
A V_LÉPÉSEK_SZÁMA és az U_LÉPÉSEK_SZÁMA értékekkel adhatjuk meg, hogy a PoV-Ray a testet legkevesebb mennyi sorra és oszlopra ossza fel, miközben a háromszög helyzetét próbálja kiszámítani. A háromszögek legnagyobb számát a következő képlet alapján számíthatjuk ki: darabkák_száma=(2^U_LÉPÉSEK_SZÁMA)*(2^V_LÉPÉSEK_SZÁMA).
Amint a képletből látható, ez az objektumtípus meglehetősen sok memóriát emészthet fel, cserébe valóban szép görbefelületeket eredményez. Ezeket az értékeket célszerű négynél kisebbre állítani, hiszen már hármasnál – amikor a felületet 64 részre osztjuk fel (ez 128 háromszöget jelent) – is megfelelő eredményt mutat.
Amikor a PoV-Ray feldolgozza ezt az objektumot, ellenőrzést végez a pillanatnyilag használt részegységen. Ennek során meghatározza, hogy az eléggé sima-e egy síkbeli téglalaphoz képest. Ennek a próbának az eredményét befolyásolhatjuk a FLATNESS_ÉRTÉK-kel, mely 0 és 1 között lehet. Amikor 0, akkor a PoV-Ray minden esetben felosztja a felületet a V_LÉPÉSEK_SZÁMA és az U_LÉPÉSEK_SZÁMA által meghatározott részre, míg ha a FLATNESS_ÉRTÉK 0-nál nagyobb, a program minden felosztás után megvizsgálja, hogy szükséges-e további darabolás. A nulla értéknél nagyobb FLATNESS_ÉRTÉK használatának előnye és hátránya egyaránt akadhat. Előnye, hogy ha az objektum nem túlságosan görbült, de tartalmaz néhány görbe felületet, a PoV-Ray meghatározza helyüket, és a részegységekre bontást ezeken a helyeken végzi el – vagyis a simább felületeket nem bontja fel feleslegesen. Legnagyobb hátránya, hogyha a program a tárgy egy részét nem osztja további részegységekre, de a velük határos részt viszont igen, ekkor előfordulhat, hogy tárgyunk kicsit összetörten kerül elő a számítások tengeréből. Rendszerint nagy lyukat fogunk rajta találni, amelyen át is lehet látni. E lyuk észlelhetősége nagymértékben attól a szögtől függ, amelyből az objektumot nézzük, tehát előfordulhat, hogy első ránézésre nem feltétlenül pillantjuk meg.
Ahhoz, hogy az elképzeléseinknek megfelelő objektumokat kapjuk, be kell szereznünk az előállításukhoz szükséges segédprogramokat. Olyan programot, ami kifejezetten Linux alá készült volna, mindeddig nem leltem, de Windows alá elérhető a Hamapatch. Aki szívesen kipróbálná, letöltheti a
➲ http://www.hamapatch.t2u.com címről.
Jelenleg készül a Nurbana nevű NURBS modellező, de még nincs használható állapotban. Kísérletező kedvű olvasóink a
➲ http://www.nurbana.cx címen találhatják meg.



Buli után…

A PoV-Ray által létrehozott *folt*-objektumok önműködően simított felülettel készülnek, a felhasználók viszont olyan segédprogramokat is alkalmazhatnak, amelyek a foltok csoportjait finomabban alakítják ki. Miután elsajátítottuk az összetett objektumok előállításához szükséges ismereteket, mindenkinek ajánlom a lemezmellékletünkön található példák és programok megtekintését. A programok között néhány átalakítóprogram is fellelhető, így a *Blender*-objektumok PoV-Ray-formátumra való átalakítását segítő Python parancsfájlok, és egy DOS(emu) alatt használható metaball-modellező program. A példák a PoV-Rayhez adott minták közül valók.
A szemet gyönyörködtető látvány megtekintése után fogjon mindenki bátran a PoV-Ray használathoz!
Mindenkinek kellemes alkotást kívánok!

Fábián Zoltán
(dzooli@freemail.hu, dzooli@yahoo.com)
23 éves, jelenleg programozóként dolgozik. Szabadidejében szívesen kirándul, túrázik. Emellett szeret rajzolni, érdekli a 3D grafika és a Linuxszal kapcsolatban minden olyan program és programnyelv, amit még nem ismer vagy nem próbált ki.