Programfejlesztés az OpenGL segítségével (2. rész)

Irrlicht 3D

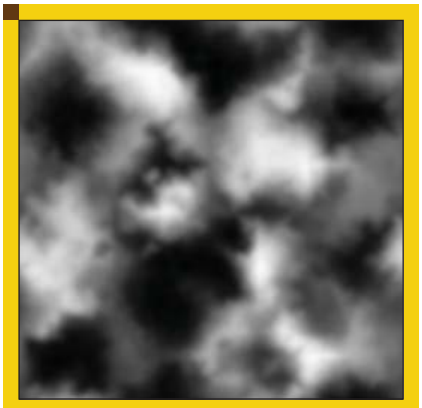
Az előző OpenGL cikkben megismerkedtünk a GLUT-tal és az OpenGL alapjaival. Most egy kis kitérőt teszünk a játékfejlesztés felé, megismerkedünk egy kiváló, nyílt forráskódú 3D engine-nel az Irrlicht-tel és, hogy saját OpenGL-es tapasztalatunkat is tovább bővítsük betekintünk az OpenGL anyag és fény kezelésébe is.

Mi az a 3D motor?

Sokan vagyunk akik már játszottak életükben legalább egy *3D*-s játékkal, de igen kevesen vannak azok akik bele gondoltak már, hogy micsoda munka elkészíteni egy ilyen játékot. A három dimenziós játékok „szíve” egy motor amely rengeteg munkát végez: kezdve az objektumok és a rajtuk található mintázatok megjelenítésétől a fizika modellezésén keresztül a speciális effektek megjelenítéséig. Általában ezek a motorok több jól elkülöníthető részből állnak, hiszen ostobaság volna mindent egybe zsúfolni.
Egy játék motor általában az alábbi részekre bontható szét:

- Grafikai megjelenítés
- Fizika
- Hangrendszer
- Mesterséges intelligencia
- Hálózatkezelés

Ezek a részek természetesen tovább bonthatók további részfeladatokra is, de mi most csak a grafikával fogunk foglalkozni.

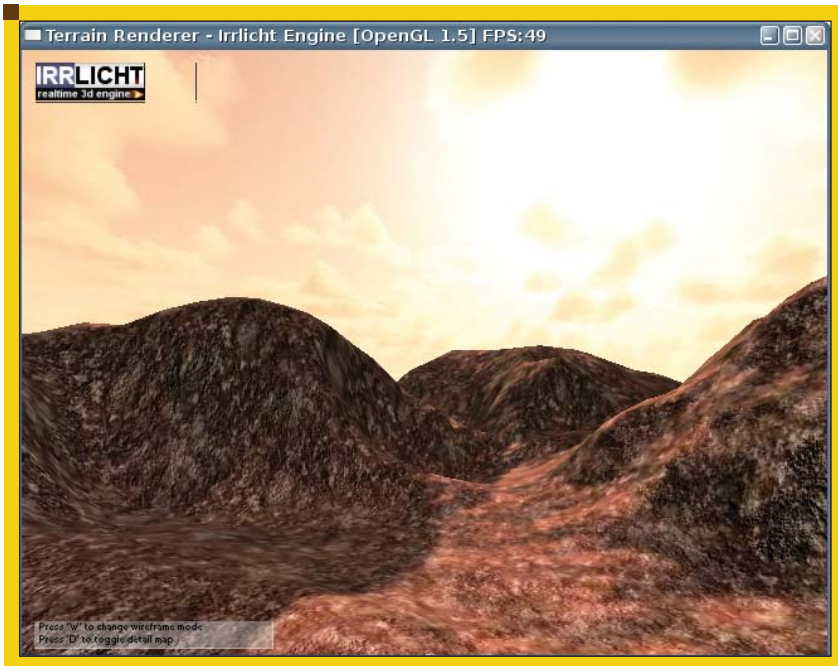


1. ábra Egy felülnézeti domborzati kép...

2. ábra A felülethez tartozó textúra

A megjelenítés

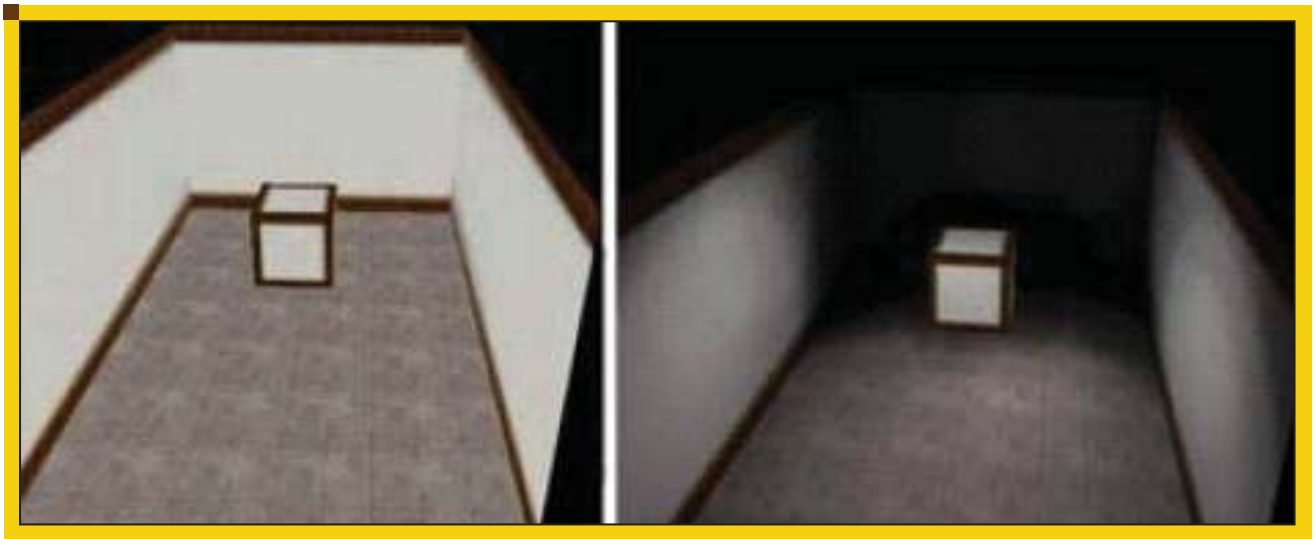
A legelső szempont amit egy grafikai rendszer megtervezésénél figyelembe kell venni, hogy milyen feladatot fog betölteni az általunk fejlesztett alkalmazás. Ha *PacMan*-t szeretnénk írni akkor teljesen felesleges *3D*-s objektumokkal foglalkozni. A „piacon” lévő engine -ek (*Source*, *Quake 1-2-3*, *LithTech*) nagy része *FPS* nézetre hajaz, de találhatunk olyat is ami általános grafikus motorként is megállja a helyét, így bármilyen típusú alkalmazásban megállja a helyét.
Sok helyről hallottam, hogy: *„Én az OpenGL/DirecX engine-t használom”*. Sem az *OpenGL*, sem a *Direct3D* nem grafikus motor. Ezek alacsony szintű programozói könyvtárak, amelyekre a bátor titánok kellő bátorság és szakértelem birtokában építhetnek egy motort. Hogy mi a különbség? Az *OpenGL*-t és a *D3D*-t azért nevezzük alacsony szintű *API*-nak,



3. ábra ...és a végeredmény!

mert a tudásuk csak az alapvető három dimenziós műveletek megvalósításáig terjed, míg egy valódi enginek ennél sokkal többet tudnak. Nézzük meg, hogy mik ezek az „extra szolgáltatások”!
A legalapvetőbb igény ami egy grafikai rendszerrel kapcsolatban felmerülhet az, hogy kezeljen objektumokat és ezeket a megfelelő adatok alapján el tudja helyezni a térben. A modellek típusától függően több fájltípus támogatása az optimális.
Animált objektumok (játékosok, szörnyek, …) legegyszerűbben *MD3* fájlformátumban tárolhatók.

Ezt a fájlszerkezetet az *idSoftware* fejlesztette ki a *Quake* játékokhoz. Az *MD* család lényege, hogy képes „csontváz” felépítésű modellek kezelésére. Ez azt jeleni, hogy a modellezők egy valódi csontrendszer elkészítése után rá tudják „húzni” a testet a csontvázra és a továbbiakban nem kell minden egyes animációs fázisban az összes testrész mozgását megszerkeszteni, elég csak a csontvázat mozgatni amely a ráhúzott testet a megfelelően módosítja. Egy ilyen fájlban el vannak tárolva különböző mozdulatsorok amelyekre az adott karakter képes lehet (gyaloglás, támadás, ugrás, esés, …). Az engine-nek képesnek kell lennie a fájl betöltésére és a mozgási fázisok animált megjelenítésére.
Statikus objektumok (asztal, szék, …) ennél lényegesen egyszerűbbek hiszen nem kell őket animálni, elégséges csak egyszer megszerkeszteni őket. Erre a feladatra kiváló a *3DS*, *OBJ*, *LWO*, *X* (és még sorolhatnánk) formátumok bármelyike.
Nagyon fontos, hogy a modellezők által elkészített objektum hiteles legyen ezért nem szabad megfeledkeznünk a textúrázásról sem. A fent említett fájltípusok mindegyike képes tárolni textúrázásra vonatkozó adatokat.
Maradt még egy harmadik objektum típus is: a pályaszerkezet. Ez nem más mint az objektumokat körülvevő világ. Beláthatjuk, hogy felesleges volna egy hegységrendszer minden egyes pontjának az előzetes eltárolása ezért itt alternatív megoldások után kell néznünk. Az egyik legegyszerűbb eljárás a domborzati kép eltárolása. Egy két dimenziós képen eltároljuk a domborzatot úgy, hogy csak fekete és fehér színeket, illetve ezek átmenetei használjuk.
Ezzel a módszerrel már lehet szép tájakat készíteni, de még mindig nincsenek belső tereink, épületeink. Erre szintén az *idSoftware* által kidolgozott fájlformátum kínálja az egyik legjobb alternatívát ami a *BSP*. A *BSP* fájlformátum lényege, hogy tárolja a tér geometriai felépítésén kívül a hozzá tartozó *BSP* fát és az előre számolt megvilágítási modelleket is.



4. ábra Radiosity-vel és nélküle...

1. táblázat

Paraméter neve	Alapérték	Leírás
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	A fény ambient RGBA erőssége
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	A fény diffuse RGBA erőssége
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	A fény specular RGBA erőssége
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	A fény (x, y, z, w) pozíciója a térbe
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	A fény (x, y, z) iránya (irányvektor), csak reflektor típusnál adjuk meg! Ha elhagyjuk akkor pont típusként fog viselkedni.
GL_SPOT_CUTOFF	180.0	Reflektorfény sugárzásának kúpszöge.
GL_CONSTANT_ATTENUATION	1.0	Konstans tompítófaktor
GL_LINEAR_ATTENUATION	0.0	Lineáris tompítófaktor
GL_QUADRATIC_ATTENUATION	0.0	Négyzetes tompítófaktor

A ***BSP*** fa lényege, hogy megkímélje a processzorunkat a felesleges munkától úgy, hogy eltárolja, hogy melyik pontból melyik pontokat ***nem*** látjuk. Ezáltal nem kell kirajzolnunk olyan dolgokat amelyek nem is esnek bele a látóterünkbe. Akit mélyebben érdekel a BSP működése az nézzen utána a ➲ ***www.gamedev.net*** oldalon a ***Binary Space Partitioning (Bináris Térfelosztás)*** című témának.
Természetesen a pályákat tárolhatjuk ugyanolyan statikus formátumban is mint az objektumainkat, de akkor nekünk kell kiszámolnunk, hogy melyek azok a felületek amelyen nem láthatók és mindemellett olyan extráktól is elesünk mint a ***radiosity*** ami nem más mint a fény előre kiszámított visszaverődés a különböző objektumokról.
Ez olyasmi mint a sugárkövetés, de itt a pályát leíró fájlstruktúrában tároljuk el a fény útját ezért lényegesen szebb képet kapunk mintha nem használnánk ***radiosity***-t, viszont meg sem közelíti egy valós sugárkövetéssel készült kép minőségét. A ***radiosity*** algoritmus csak statikus fényekre vonatkozik a ***BSP***-ben! Értelemszerűen a futásidőben létrehozott fények (rakéta, tűzgolyó, …) számára nekünk kell számolnunk, vagy más alternatíva után néznünk. Ahhoz, hogy virtuális világunk még tökéletesebb legyen nem szabad elfelejtenünk a fényeket sem.
A fényeknél megvilágítási modelleket definiálhatunk amelyek leírják a fények és a modelltér kapcsolatát. Az ***OpenGL*** által támogatott három megvilágítási modell: szórt háttér világítás (***ambient***), diffúz visszaverődés (***diffuse***) és a fényvisszaverődés csillogó felületekről (***specular light***). Ha egy fényt hozunk létre ***OpenGL*** segítségével akkor a fenti három tulajdonságot mindenképpen meg kell adnunk, illetve meg kell határoznunk, hogy a fényforrásunk pont (nap), vagy reflektor típusú-e (elemlámpa). A három fő tulajdonság (***ambient, diffuse, specular***) beállítása az RGBA paletta használatával történik a már jól ismert ***OpenGL***-es módon. Minden érték 0-tól 1-ig vehet fel értéket. Ha ezeket beállítottuk akkor jöhetnek az egyéb beállítások is (*1. táblázat*). Az utolsó három tulajdonság a fény gyengülését határozza forrástól való távolság függvényében.

A fény tulajdonságainak beállítása a `glLightfv` paranccsal történik a következő módon:

```
float light_ambient[]  = { 0.0,
➥0.0, 0.0, 1.0 };
float light_diffuse[]  = { 1.0,
➥1.0, 1.0, 1.0 };
float light_specular[] = { 1.0,
➥1.0, 1.0, 1.0 };
float light_position[] = { 1.0,
➥1.0, 1.0, 0.0 };
float light_direction[] = {
➥12.0, 14.0, -3.0 };
float light_cutoff[] = { 90.0 };

glLightfv(GL_LIGHT0,
➥GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0,
➥GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0,
➥GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0,
➥GL_POSITION, light_position);
glLightfv(GL_LIGHT0,
GL_SPOT_DIRECTION,
➥light_direction);
glLightfv(GL_LIGHT0,
➥GL_SPOT_CUTOFF, light_cutoff);
```

2. táblázat

Paraméter neve	Alapérték	Leírás
GL_LIGHT_MODEL_AMBIENT	(0.2,0.2,0.2,1.0)	A szórt háttérvilágítás intenzitása (RGBA)
GL_LIGHT_MODEL_LOCAL_VIEWER	0.0	Hogyan számolódjon a fény specular fényvisszaverődés szöge.
GL_LIGHT_MODEL_TWO_SIDE	0.0	Egy vagy két oldalas megvilágítás. Ha kétoldalas ra állítjuk akkor a felület belső oldalának anyagjellemzői is beleszámítanak a fényvisszaverődésbe.

3. táblázat

Paraméter neve	Alapérték	Leírás
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Az anyag ambient RGBA tükröződése
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	Az anyag diffuse RGBA tükröződése
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	Az anyag specular RGBA tükröződése
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	Az anyag saját fénye
GL_SHININESS	0	A fényvisszaverődéskor létrejövő „fényfolt” intenzitása. Minél magasabb annál élesebb.

1. Lista

```
void myinit(void)
{
//A fény tulajdonságainak beállítsa (ambient,
diffuse, elhelyezkedés
  GLfloat ambient[] =  {0.0, 0.0, 0.0, 1.0};
  GLfloat diffuse[] =  {1.0, 1.0, 1.0, 1.0};
  GLfloat position[] = {0.0, 3.0, 3.0, 0.0};

  GLfloat lmodel_ambient[] =  {0.2, 0.2, 0.2,
  ➥1.0};
  GLfloat local_view[] =  {0.0};

//A fény létrehozása a tulajdonságok
//megadásával...
  glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
  glLightfv(GL_LIGHT0, GL_POSITION, position);
//..és a megvilágítási modell definiálása
//(alap beállítások)
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT,
  ➥lmodel_ambient);
  glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER,
  ➥local_view);

//A fények engedélyezése
  glFrontFace(GL_CW);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_AUTO_NORMAL);
  glEnable(GL_NORMALIZE);
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LESS);
}
void renderTeapot(GLfloat x, GLfloat y,
  GLfloat ambr, GLfloat ambg, GLfloat ambb,
  GLfloat difr, GLfloat difg, GLfloat difb,
  GLfloat specr, GLfloat specg, GLfloat specb,
  ➥GLfloat shine)
{
  float mat[4];
  glPushMatrix();
  glTranslatef(x, y, 0.0);

//Ambient összetevő beállítása
  mat[0] = ambr;  mat[1] = ambg;  mat[2] =
  ➥ambb;  mat[3] = 1.0;
  glMaterialfv(GL_FRONT, GL_AMBIENT, mat);

//Diffuse összetevő beállítása
  mat[0] = difr;  mat[1] = difg;  mat[2]
  ➥= difb;
  glMaterialfv(GL_FRONT, GL_DIFFUSE, mat);

//Specular összetevő beállítása
  mat[0] = specr;  mat[1] = specg;  mat[2]
  ➥= specb;
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat);

  glMaterialf(GL_FRONT, GL_SHININESS, shine
  ➥* 128.0);
  glutSolidTeapot(1.0);
  glPopMatrix();
}
```

És végül meghívhatjuk a renderTeapot függvényt megfelelően paraméterezve:

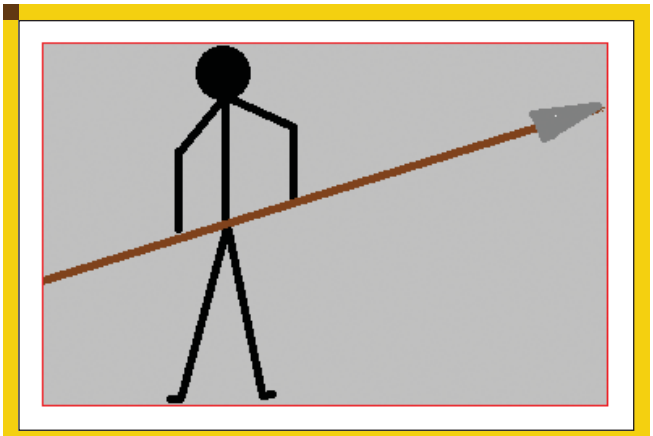
```
//Teáskanna smaragdból...
  renderTeapot(2.0, 17.0, 0.0215, 0.1745,
  ➥0.0215,
    0.07568, 0.61424, 0.07568, 0.633, 0.727811,
    ➥0.633, 0.6);
//...ezüstből...
  renderTeapot(6.0, 2.0, 0.19225, 0.19225,
  ➥0.19225,
    0.50754, 0.50754, 0.50754, 0.508273,
    ➥0.508273, 0.508273, 0.4);

//...és fekete gumi kiszerelésben.
  renderTeapot(14.0, 17.0, 0.02, 0.02, 0.02,
  ➥0.01, 0.01, 0.01, 0.4, 0.4, 0.4, .078125);
```

5. ábra A három teáskanna

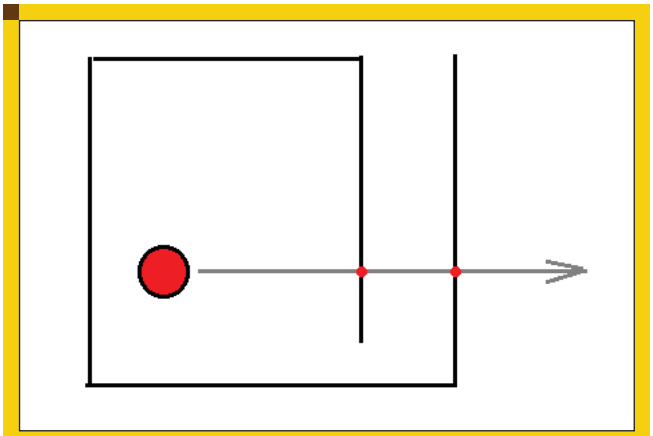
Jól látható, hogy első paraméter a fény azonosítója `GL_LIGHT7`-ig, majd ezt követi a változtatandó tulajdonság konstans neve és végül pedig a tulajdonság leírása vektor formájában.
Ha már létrehoztunk egy fényforrást, akkor azt is be kell állítanunk, hogy milyen megvilágítási modellt kívánunk használni. Ez a `glLightModelfv` utasítással történik amelynek paraméterként megadhatjuk, hogy egy-vagy két oldali megvilágítás modellt kívánunk-e használni, illetve itt definiálhatjuk a modelltér szórt háttérvilágításának intenzitását is (*2. táblázat*).
Már meg tudjuk világítani a teret, de még valami mindig hiányzik: az anyagjellemzők. Ezek azért fontosak, mert egy vas golyóról másképpen verődik vissza a fény mint egy narancsról, vagy mint egy üveggolyóról.
Az anyagjellemzők definiálása hasonlóan történik mint a fény tulajdonságainak beállítása.
Amikor egy objektum anyagát definiáljuk, nem azt mondjuk, hogy ez a teáskanna piros legyen, hanem úgy állítjuk be az anyagjellemzőket, hogy a piros fényt verje leginkább vissza. Ez a valóságban is így működik.
A fekete azért fekete, mert a ráeső fény nagy részét elnyeli, míg a fehér mindent visszaver. Az ***OpenGL*** segítségével meg kell határoznunk, hogy melyik fény típusokból (***ambient, diffuse, specular***) mennyit ver vissza az objektum.
Az anyag tulajdonságait a `glMaterialfv` paranccsal tudjuk beállítani, amelynek paramétereit a *3. táblázatban* láthatjuk.
Az *1. Listában* látható egy példa, amit az ***OpenGL Redbookból*** emeltem ki. Ebben a példában igen jól látható, hogy milyen egyszerű fényforrásokat, megvilágítási modellt és anyagjellemzőket beállítani az ***OpenGL*** segítségével.
A mostani cikkben ennyi az ***OpenGL*** specifikus rész, úgyhogy most kanyarodjunk is vissza eredeti témánkhoz a ***3D*** motorokhoz.
Most már tudjuk hogyan épülnek fel az alapvető modellek és pályák, valamint tisztában vagyunk a megvilágítás és az anyagjellemzők kapcsolatával is. Ejtsünk most egy pár szót a fizikáról, azon belül is programozástechnikailag az egyik legbonyolultabb alapvető műveletről az ütközésvizsgálatról.

Az ütközésvizsgálat (Collision detection)

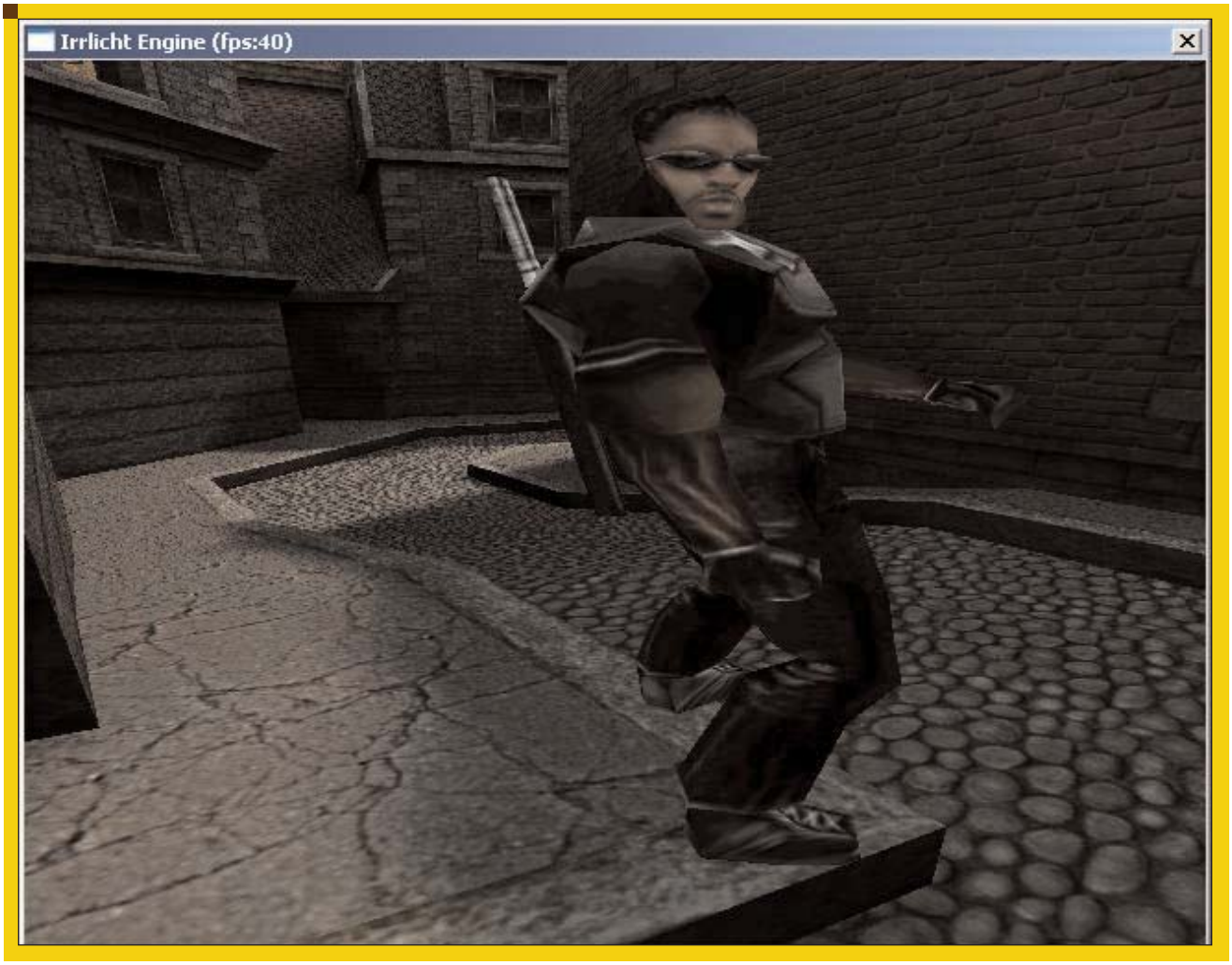
Ütközésvizsgálat nélkül a legtöbb játék játszhatatlan lenne, ezért igen nagy hangsúlyt kell fektetni a programozóknak egy gyors és pontos algoritmus kifejlesztésére. Az ütközésvizsgálat egy viszonylag nagy számítás igényű eljárás ezért célszerű a lehető legjobban optimalizálni ezt az algoritmus. Hogyan döntjük el, hogy két objektum metszi-e egymást? A legegyszerűbb módszer ha a játékban lévő minden objektum koordinátáját (pontosabban azoknak minden egyes „face”-ét) összehasonlítjuk az összes többivel. Egy 200 ezer poligonból álló objektum rendszer esetén ez „200000 a négyzeten” lépésből állna. Egyes játékok több millió poligonnal dolgoznak, úgyhogy ezt az eljárást el is vethetjük. A másik igen primitív eljárás, hogy a kamera látóterébe eső objektumokkal foglalkozunk, így jelentősen csökkenthető ennek az algoritmusnak a műveletigénye. De sajnos ez sem az igazi.
Másik igen butácska algoritmus a „dobozolás” módszere. Ezt már jó hatásfokkal lehet ***2D***-s játékoknál, vagy ***izometrikus (Diablo, Starcraft, Fallout)*** felépítésűeknél alkalmazni, de ***3D***-ben kerülendő a használata!
A lényege, hogy minden test köré teszünk egy akkora téglatestet amibe pontosan belefér, és ezeknek a téglatesteknek az ütközését vizsgáljuk. Egy 20-30 ezer poligonból álló ***MD3*** modellnél így már csak 12 poligonnal kell számolnunk. Ez igen jelentős sebesség javulást eredményez, viszont vannak olyan objektumok amiket nem lehet így bezárni, vagy ostobaság lenne ezt tenni.
Az alábbi képen egy stilizált ember figura van dárdával a kezében (ha valaki nem ismerte volna fel).



6. ábra Feleslegesen nagy „doboz” a lándzsás harcos körül



7. ábra A mozgó test irányvektora két poligont metsz el

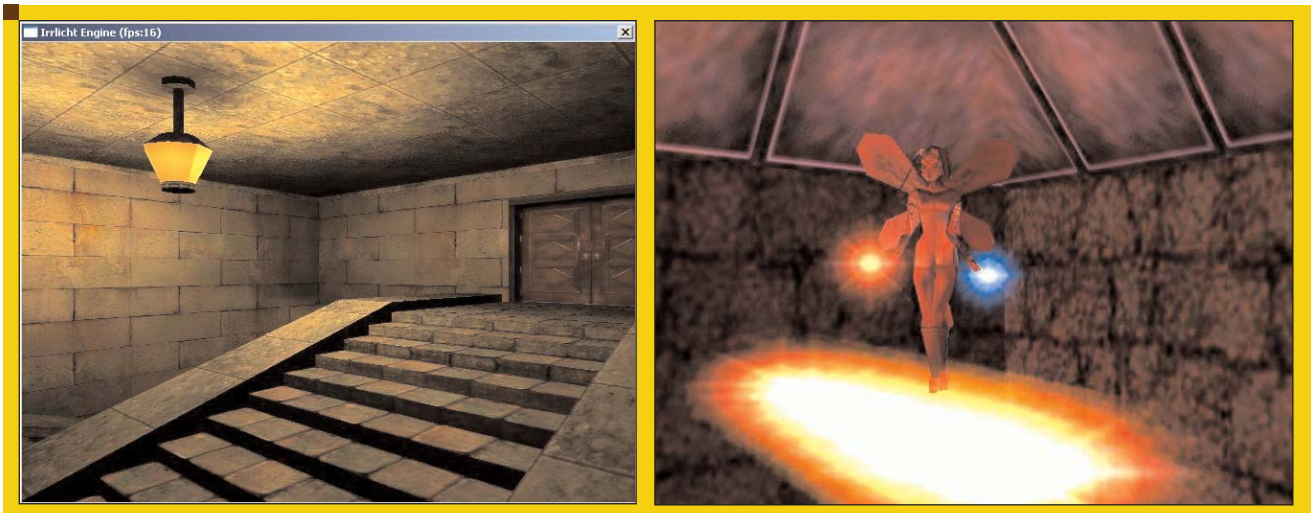


A körülötte látható téglalap a legkisebb „doboz” amibe bele rajzolni a harcost. Ha azt akarjuk megvizsgálni, hogy egy nyílvessző eltalálja-e a harcosunkat, akkor is találatot fog generálni az algoritmus ha nem a figurát, hanem a szürkére színezett területet találnák el. Ez sajnos nem jó. Ráadásul jelentős esélyünk van arra is, hogy egy ilyen „dobozolt” modell a pályán található ajtók 80%-án fenn fog akadni. A fenti példa a dobozolós algoritmus két dimenziós változata. Ehhez a primitív algoritmushoz képest jelentős előrelépés lehet valamilyen egyszerű sugárkövetési (***raytracing***) algoritmus használata. A sugárkövetési algoritmusok működése rettenetesen egyszerű, csak egy picit kell a geometriához érteni. Tételezzük fel, hogy van egy sugár (egyenes) amely keresztül halad „A” és „B” pontokon. Értelemszerűen a két pont távolsága az „A” és „B” pontokat összekötő szakasz hossza. Tegyük hát a következőt: a vizsgálni kívánt mozgó objektumunkból indítsunk egy egyenest. Az egyenes iránya az objektum haladási iránya. Vizsgáljuk meg, hogy milyen távolságra van a legközelebbi objektum amit el tudunk metszeni ezzel a sugárral. Ha találunk ilyen felületet akkor állapítsuk meg, hogy milyen távol van! Ha a távolság 0 körül van akkor az objektum nem haladhat tovább. Ha egy egyenessel akarunk elmetszeni sok kis háromszöget (poligont), akkor nincs más teendőnk mint egy nagy egyenletrendszerbe irni az egyenes irányvektoros egyenletét és a sík egyenletét, majd megoldani az egyenletrendszert. Az egyenlet gyökei a metszéspontot adják eredményül. Akit pontosan érdekel a dolog működése, látogasson el a ➲ ***http://www.lighthouse3d.com*** oldalra és az ***OpenGL***-es leírások között bőven találhat forráskóddal megtámogatott példát a fenti probléma megoldására.

A fenti algoritmus már igen jó hatásfokkal működik és jól optimalizálható. A ***Quake*** motor is ezt az algoritmust használja. Ha valaki tovább szeretné optimalizálni még ezt az algoritmust is, akkor javaslom a ***„térnyolcadoló fák” (Octree)*** utáni nézelődést az interneten. Az ***Octree*** egy nagyon hatékony eljárás, akár 70%-al is meggyorsíthatja a keresést. A lényege, hogy nyolc ágú fákat hoz létre úgy, hogy a modellteret nyolc részre osztja, majd a részeket ismét nyolc részre bontja, és így tovább. Ha felosztottuk a teret akkor már gyorsan meg tudjuk nézni, hogy az adott tér részben melyik objektumok vannak, az összes többit figyelmen kívül hagyhatjuk a keresés folyamán. Aki komolyan bele szeretné magát ásni a játékfejlesztésbe, annak javaslom a cikk végén található linkgyűjteményt, illetve a most következő ismertetőt az ***Irrlicht3D*** nevű játék motorról.

Irrlicht3D

Az ***Irrlicht*** egy teljesen ingyenes, nyílt forráskódú, stabil és platformfüggetlen ***3D*** motor azok számára akik nem



8. ábra Néhány kép – Ilyen az Irrlicht munka közben

érzik még magukat annyira bátornak, hogy egy saját motor fejlesztésébe fogjanak, de szeretnének belevágni a játékfejlesztésbe. Pár hónappal ezelőtt jutott eszembe, hogy megnézzem hol is tartanak most a nyílt játékmotor fejlesztések, ezért ellátogattam a ***Crystal Space*** oldalára. (A ***Crystal Space*** egy szintén nyílt motor, rengeteg extra szolgáltatással). Örvendezve láttam, hogy hamarosan elkészül a végső, stabil 1.0 verziószámú kiadás. Gondoltam belenézek, megnézem mennyit fejlődött mióta utoljára láttam. Már akkor picit zavarba jöttem amikor forráskód címen egy majd 40 Mbyte-os állomány kezdett el letöltődni hozzám, de amikor kibontottam és megláttam a forráskódját, majdnem leestem a székről.
Rengeteg szolgáltatással, kiegészítővel rendelkező motor, tényleg mindent tud ami szem-szájnak ingere, de programozó legyen a talpán aki játékot kezd fejleszteni vele.
Ne szaporítsuk tovább a szót: komplex és ehhez mérten nagyon bonyolult is. Ezt a kis incidenst követően ajánlották egy fórumon az Irrlichtet. A CS-es kalandomat követően kicsit bátortalanul fogtam neki a keresgélésnek. Az ➲ ***irrlicht.sourceforge.net*** oldalon rá is találtam a keresett enginere és meglepve tapasztaltam, hogy „csak” 14 Mbyte a forráskód ami tartalmazza a dokumentációt és 15 példaprogramot teljesen az alapoktól. Letöltöttem, kicsomagoltam majd ismét meglepődtem. Ennyire jól áttekinthető, jól strukturált és dokumentált programot keveset találni.

Az alábbi dolgokat tudja a 0.12.0-s verzió:

- Platformfüggetlen megjelenítés (***OpenGL*** vagy ***Direct3D***)
- ***Pixel*** és ***vertex shader*** támogatás
- Beltéri és kültéri „jelenetek” teljes támogatása, észrevétlen átmenet közöttük.
- Csontváz vagy „morph” alapú karakter animáció támogatása
- Dinamikus-fény kezelés, környezet tükröző textúrák, tűz, víz, időjárás effektek
- Részecske rendszer (robbanások)
- Komplett ***2D***-s grafikus felület, a saját programunk kezelőfelületének kialakításához.
- Jól dokumentált, sok példával ellátott egyszerű dokumentáció
- Elterjedt modell formátumok támogatása: ***Maya (obj)***, ***3D Studio Max (3ds)***, ***COLLADA (dae)***, ***DeleD (dmf)***, ***Milkshape (ms3d)***, ***Quake 3*** pályaszerkezet ***(bsp)***, ***Quake 2*** karakter modellek ***(md2)***
- Elterjedt kép formátumok támogatása: ***bmp, png, psd, jpeg, tga, pcx...***
- Gyors és egyszerű ütközés vizsgálat.
- Közvetlen olvasás ***zip*** állományokból.
- *XML* parser
- *GCC 3.2+* támogatás

Ebből a listából egyértelműen kiderül, hogy az ***Irrlicht*** inkább „csak” egy *3D* engine mintsem teljes játék motor. Hiányzik belőle a hang- és hálózat kezelés, valamint nincsenek benne fizikai eljárások és mesterséges intelligencia sem. Ennek ellenére mindenkinek ajánlom figyelmébe aki eddig nem mert, vagy nem tudott belevágni a játékfejlesztés – nem is annyira – „misztikus” világába.
Végezetül álljon itt néhány kép amelyek az Irrlicht példaprogramjaiból vannak, valamint a cikkben említett linkek, leírások.

Tóth Péter
(thotacc@drotnet.hu)
A BMF hallgatója vagyok, mellette egy kis- és középvállalatok informatikai rendszereinek Linuxos átállításával és szoftverfejlesztéssel foglalkozó cég informatikai vezetőjeként tevékenykedem. Kevés szabadidőmet barátnőmmel és barátaimmal töltöm egy-két sör társaságában.

KAPCSOLÓDÓ CÍMEK

Az Irrlicht3D honlapja
➲ http://irrlicht.sourceforge.net

OpenGL példa programok, leírások, érdekességek
➲ http://nehe.gamedev.net

OpenGL, VRML, Sugárkövetés
➲ http://www.lighthouse3d.com

Egy nyílt forráskódú fizikai engine
➲ http://ode.org/

Egy másik nyílt fizikai motor
➲ http://www.newtondynamics.com