Debreceni Egyetem
Informatikai Kar

Részecskerendszer OpenGL-ben

Témavezető:
Dr. Tornai Róbert
egyetemi adjunktus

Készítette:
Debreczeni András
PTI szakos hallgató

Debrecen
2008

Tartalomjegyzék

1. Bevezetés

A részecskerendszer fogalmát nem lehet pontosan meghatározni. Nem kötődik szigorúan a számítógépes grafikához, bár legtöbbször ezen a területen alkalmazzák. A részecskerendszer egy elv, modellezési mód. Jean-Christophe Lombardo[1] szavai állnak talán legközelebb az általános megfogalmazáshoz: „Általánosságban, mozgási szabályokkal felruházott pontoknak a halmazát tekinthetjük részecskerendszernek." Ezt a meghatározást alapul véve, a részecskerendszerre adható példák tárháza kimeríthetetlen. Részecskerendszernek tekinthetjük például a fizikusok által használt folyadék vagy gáz halmazállapotú részecskék mozgását leíró szabályrendszert, a szökőkútból előtörő víz útját szemléltető leírást, de vehetjük példának a GTA című számítógépes játékban az autókat, amelyek a KRESZ szabályai szerint össze-vissza mozognak a városban, a gyalogosokat, akik szintén a járdán sétálva keltik a játékosban az élő környezet érzetét. Tehetjük mindezt azért, mert az adott definíció nem határozza meg a pontok méretét, melyek ezáltal lehetnek parányiak vagy egészen nagyok is, s nem korlátozzuk a mozgásuk megadására használható szabályrendszert sem. A számítógépes grafikai megközelítésről először William T. Reeves[2] készített publikációt, azokról az eredményekről, amelyeket a Star Trek II – Khan haragja c. film vizuális hatásainak készítése közben elért. Az általa leírt általános megközelítések mind a mai napig aktuálisak, használhatóak. Ez a dokumentum tekinthető a grafikában használatos részecskerendszerek alapjának.

Jelen dolgozat célja egy vizuális hatás elkészítése OpenGL használatával, mindezt oly módon, hogy az elkészülő mű, minél könnyebben felhasználható legyen más programokban, lehetőleg kiaknázva napjaink grafikus hardvereinek képességeit. Vizsgálat tárgya lesz mindemellett, hogy ezen célok megvalósítása elérhető-e egy általánosított rendszerrel vagy specializálni kell a részecskerendszerek fejlesztését, illetve elemzésre kerül a manapság elterjedt grafikus kártyák képességének tényleges sebességbeli hatása.

2. A részecskerendszer

Mint a Bevezetésben említésre került, a részecskerendszer nehezen körülhatárolható fogalom, részét képezi több diszciplínának, de egyik sem tekintheti sajátjának. Egyszerűsítve azt mondhatjuk, hogy amiből sok van és mozog, az lehet részecskerendszer.

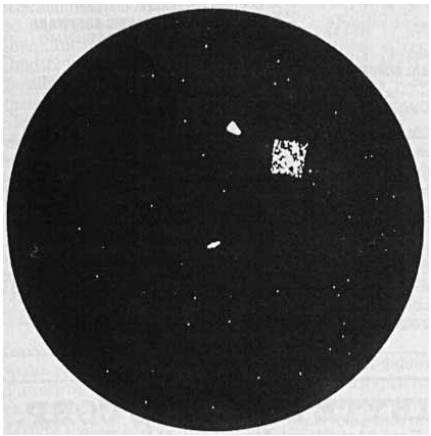
2.1. Miért felkapott téma a részecskerendszer?

Általában azért vannak előtérben más számítógépes grafikai megoldásokkal szemben a részecskerendszerek, mert olyan lehetőségekkel rendelkeznek, melylyel más módszerek nem. Ugyanis a valóság folytonos mozgás, tele véletlenszerű és konkrét körvonallal nem rendelkező elemekkel. Ezeket nevezte Reeves[3] 'fuzzy' objektumoknak. Ilyen például a tűz, ahogyan ég, a füst, ahogyan száll, a légbuborék mozgása a szénsavas ásványvízben. Ha létrehozunk egy fizikailag korrekt, a valóságnak megfelelően működő részecskerendszert, a virtuális világunkba juttathatjuk el ezt a látványvilágot s hangulat javító elemeket, valós időben alkalmazva azokat.

2.2. A részecskerendszerek története

2.2.1. SpaceWar! *(1. ábra)*

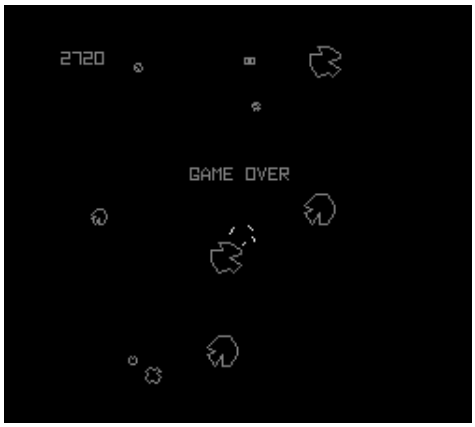
1962-ben, Steve Russel, Martin Graetz és Wayne Wiitanen a massachusettsi egyetem PDP-1-es számítógépére írták meg az egyik legelső számítógépes játékot, a SpaceWar!-t. A játékmenet alapja, hogy egy-egy játékos egy-egy űrrepülőt irányít, s a játékosok lövöldöznek egymásra. A cél az ellenfél eltalálása úgy, hogy eközben a repülő elkerülje az ütközést a középen található csillaggal. A játék elkészítése körülbelül kétszáz órájába került a fejlesztőknek, s ezalatt közel negyven A/4-es oldalnyi forráskódot írtak. Bár ekkor még nem létezett a részecskerendszer fogalom, de az űrhajó szétrobbanásakor létrejövő effektust részecskerendszernek lehet tekinteni. Egy közös pontból véletlenszerű irányokba kiinduló pöttyöket alkalmaztak a robbanás látványának megjelenítésére.



1. ábra A SpaceWar!

2.2.2. Asteroids *(2. ábra)*

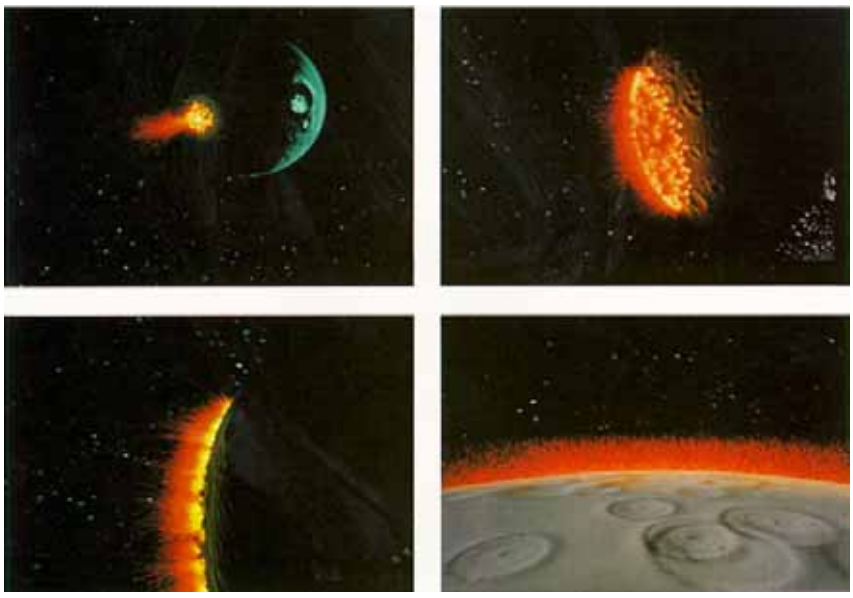
1979-ben az Atari készített egy játékot. Ebben a játékban is űrrepülőt kell irányítani, de csak egyet. A cél egy aszteroida mezőben röpülve elpusztítani az összes kisbolygót, elkerülve a velük való ütközést. Ezt a tevékenységünket időnként felbukkanó repülő csészealjak nehezítik, akik tüzelnek is ránk. A játék az arcade játékok aranykorát hozta el, és az elsők között volt a fizikailag hű részecske szimulációban. Ugyanis a szétlőtt aszteroidák szakaszokra robbantak szét, s ütköztek az útjukba kerülő tárgyakkal, aminek a hatására az útjuk megváltozott.



2. ábra Az Asteroids

2.2.3. Star Trek II – Khan haragja *(3. ábra)*

1983 a részecskerendszerek történetének fontos pontja volt, mikor a Star Trek II. című filmnek a látványelemeként elkészített részecskerendszerből megszületett William T. Reeves az előző fejezetekben már hivatkozott, mai napig alapvető írása. A filmhez szükséges volt egy bolygó felszínén végigfutó tűzfal animáció, ami a lehető leghűebben ábrázolta a tüzet. Ez olyan jól sikerült, olyan új, addig tudatosan nem használt modellezési módot sikerült megvalósítani általa, hogy a koncepció maradandó lett, s még ennyi idő elteltével is használható, pedig 25 év a számítástechnikában elképesztően sok idő.



3. ábra Jelenetek a Star Trek II – Khan Haragja c. filmből

2.3. Az alapok

A részecskerendszerek felépítése és ciklikussága általában megegyezik, kisebb eltérések az alkalmazásukból kifolyólag létezhetnek, de a jellegüket alapvetően nem változtatják meg. Beszélhetünk részecskeforrásról, aminek feladata a részecskék kezelése, s részecskékről, amelyek önálló entitásként vannak jelen a rendszerben. A részecskéket létrehozzuk a rendszerben, ott élnek egy ideig, mozognak, viselkednek, majd kihalnak a rendszerből, s ekkor eltávolítjuk őket, ezzel tartva szinten a részecskék számát. A részecskékre vonatkozó életciklust az emberi életciklussal analóg fogalomrendszerrel hozták létre. Körforgás figyelhető meg, aminek lépései a következőek: létrehozás, frissítés, eltávolítás és kirajzolás. Ezt a cirkulációt lehet állandónak tekinteni, bár vannak olyan rendszerek, amelyek állandó részecske mennyiséggel dolgoznak, nem igénylik a részecskék elhalálozását, s újragenerálását.

2.4. A részecskerendszer alapvető típusai

Két típust lehet megkülönböztetni grafikai szempontból: az egyik az állapotmentes részecskerendszer, a másik az állapotot megőrző részecskerendszer. Ezt a kategorizálást használja Andreas Kolb[4] is.

2.4.1. Az állapotmentes részecskerendszer

Ilyeneknek tekinthetjük azokat a rendszereket, amelyeknél a részecske pozíciója egyértelműen meghatározható a kiindulási adatokból (hogy melyek ezek, arról később írok), tudva azt, hogy mennyi ideje is él a rendszerben. Ennek fényében látszik, hogy az ebbe a típusba tartózó rendszerek egyszerű hatások leírására valók, ez az erősségük. Könnyen lehet velük látványos, a környezet által nem befolyásolt effektusokat készíteni. Az akciójátékokban ezek lehetnek például a fegyverek tüzelésekor létrejövő szikrák, a lövedék becsapódásakor keletkező foltok. Fontos, hogy a részecskék pozíciójának kiszámításához olyan függvényt kell megadnunk, mely rendelkezik zárt alakkal. Minél bonyolultabb hatást szeretnénk elérni, annál komplexebb alakot fognak ölteni ezek a képletek.

2.4.2. Az állapotot megőrző részecskerendszer

Az állapotot megőrző részecskerendszereket szokták használni nagyobb látványelemek elkészítéséhez, amikor szükség van arra, hogy a rendszer reagáljon a környezetre. Minden egyes lépésben frissülnek a részecskék tulajdonságai annak függvényében, hogy hol voltak előzőleg, s milyen kölcsönhatásokba kerül-

tek más részecskékkel vagy a környezettel. Ezt követően kell eldönteni, hogy mely részecskéket kell eltávolítani a rendszerből, s mennyi új részecskét kell a rendszerbe behelyezni.

2.5. A részecske

A részecskét akár konténernek tekinthwtjük, amelyben tulajdonságokat tárolunk. Ezek a tulajdonságok adják a részecske aktuális állapotát. A legalapvetőbb tulajdonságok:

– pozíció
– sebesség
– méret
– szín
– átlátszóság
– alak
– élettartam.

Ezek általános elemek, amelyeknél többet is tartalmazhat egy részecske, de ezeket a tulajdonságokat sem kötelező mindet tartalmaznia. Erősen befolyásoló tényező, hogy mit akarunk modellezni a részecskével. Lehetnek olyan esetek, amikor nem szükséges eltárolni a színt, mert az minden részecskénél egységes, vagy külön kell még azt is tartalmaznia részecskének, hogy miként változik a szín. Erre nagyon jó leírást ad a 'Rendu réaliste de pluie en temps-réel'[5] című közlemény. A cikk az eső valósághű létrehozásával foglalkozik, s a valósághűség nagyban függ attól, hogy az egyes esőcseppeknek milyen a színe. Itt az egyes esőcseppek képezik a részecskéket.

2.6. A forrás

A részecskeforrás olyan entitás, amely a részecskék létrehozásáért felelős. Ez az az objektum, amit elhelyezünk háromdimenziós világunkban, hogy létrejöjjön a kívánt hatás. A forrás felelős azért, hogy mennyi részecskét hozunk létre egyszerre, s hogy a létrehozott részecskék milyen kezdeti értékekkel rendelkeznek.

3. Az OpenGL

Az OpenGL (Open Graphics Library) egy szabvány specifikáció, ami több nyelven és több platformon elérhető API-t (Application Programming Interface) definiál, két- és háromdimenziós grafikai programok írásához. Ez az interfész több mint 250 különböző függvényhívást tartalmaz, melyek segítségével komplex háromdimenziós jeleneteket rajzolhatunk egyszerű primitívekből. Az OpenGL részletes leírása példakódokkal, megtalálható az OpenGL Programming Guide c. könyvben[6].

3.1. Hogy alakult ki?

A ’80-as években olyan szoftvert fejleszteni, amely majdnem minden grafikus hardveren működőképes, szinte lehetetlen próbálkozásnak bizonyult, ugyanis minden egyes hardverelem meghajtójának más-más interfészei voltak. Egyre időszerűbbé vált valamilyen egységesítés, ami megoldhatta ezt a problémát. A SIGGRAPH Core csoport egy új ANSI szabvány API-t dolgozott ki megoldásként, melyet PHIGS-nek (Programmer’s Hierarchical Interactive Graphics System) nevezett. Az évtized végére ez el is terjedt, de a megvalósítások oly mértékben különböztek, ami már ellehetetlenítette a szoftvergyártók azon törekvéseit, hogy platformfüggetlen programokat készítsenek.

A Silicon Graphics Inc. is kifejlesztette saját szabadalmaztatott API-ját, az IrisGL-t, ami annyira jól sikerült, hogy piacvezetővé vált a grafika terén. A programozók szerint a PHIGS-nél sokkal rugalmasabb és könnyebben használható volt. De így is sok probléma adódott, ugyanis az IrisGL tartalmazott ablak-, egér- és billentyűzetkezelő API részeket is, amelyek gyakran nem működtek együtt a különböző rendszereken. Így a szoftvergyártók nyomására az SGI kidolgozott egy nyílt szabványt, az OpenGL-t. Azért nem az IrisGL-t tették nyílttá, mert az sok, szabadalmaztatott eljárást tartalmazott.

3.1.1. ARB

1992-ben az SGI segítségével jöhetett létre az OpenGL ARB (Architectural Review Board), egy cégeket tömörítő csoport, aminek feladata az OpenGL fenntartása és fejlesztése volt. Megalakulása évében kiadta az OpenGL 1.0-s változatát, amely a szoftvergyártók igényeit kielégítette, mivel ők képviselhették magukat az ARB-ben, irányítva ezzel az API fejlődését.

3.1.2. Az IrisGL és OpenGL hasonlósága

Mivel az OpenGL fejlesztését az SGI kezdte, s neki már volt egy jól bevált API-ja, nem meglepő, hogy nagyon sokat merített belőle. A hasonlóság nyomban szembe tűnik az alábbi példakódokból.

IrisGL:

```
bgnpolygon();
c3s(red);
v3f(v0);
c3s(green);
v3f(v1);
c3s(blue);
v3f(v2);
c3s(black);
v3f(v3);
c3s(white);
v3f(v4);
endpolygon();
```

OpenGL:

```
glBegin(GL_POLYGON);
glColor3sv(red);
glVertex3f(v0);
glColor3sv(green);
glVertex3f(v1);
glColor3sv(blue);
glVertex3f(v2);
glColor3sv(black);
glVertex3f(v3);
glColor3sv(white);
glVertex3f(v4);
glEnd();
```

3.1.3. Fejlődés

Az OpenGL tehát nem más, mint egy specifikáció. Olyan dokumentációt jelent, ami függvényeket definiál és meghatározza, hogy milyen működést kell azoknak megvalósítani. Ebből a specifikációból kell a hardvergyártóknak elkészíteni az implementáció(i)kat a saját hardvereikhez oly módon, hogy teljes mértékben megfeleljenek az OpenGL specifikációban leírtaknak, és amit lehet, hardver szinten gyorsítva valósítanak meg.

Az egyes ARB tagok, ha valamit hozzá akarnak tenni az OpenGL funkcionalitásához, akkor azt saját gyártóspecifikus kiterjesztéseiken keresztül valósíthatják meg. Később, ha az ARB-n belül megegyezésre jutnak afelől, hogy a kiterjesztéseknek van haszna, a későbbi OpenGL verziókba beemelik azokat, mint alap funkciót vagy mint ARB kiterjesztést.

A többoldali egyeztetések körülményessége miatt lassan haladt a fejlődés, de széles körben támogatottan. 1995-ben a Microsoft kiadta a Direct3D API-ját, amit az OpenGL ellenfelének szánt, de ez gyermekbetegségei és csak Windows támogatása miatt kezdetben nem lett széleskörben elfogadott. De mivel a Microsoft (mint a Word esetében) most is oly mértékű lobbit és fejlesztést tudott felmutatni, méghozzá a fejlesztők igényeit kiszolgálva, hamarosan egyre több

(főként játék) fejlesztőt nyertek meg. Ekkor volt egy próbálkozás az OpenGL és Direct3D egységesítésére, de a piaci támogatás hiánya miatt ez a próbálkozás befulladt. A Microsoft a saját API preferálása miatt 2003-ban kilépett az ARB-ből (így a Windows rendszerekben a mai napig az OpenGL akkori, 1.4-s változata található meg).

3.1.4. OpenGL 2.0

Az OpenGL 2.0-s változatára nagy hatást gyakorolt a 3Dlabs, ugyanis szerintük az OpenGL fejlődése ekkor erősen stagnált, érződött az erős irányítás hiánya. A 3Dlabs alapvető változásokat javasolt a szabványba vételhez, amelyek egy részét ugyan elutasította az ARB, más részét csak módosításokkal fogadta el, de ezzel legalább kimozdította a szabványt a stagnálásból. Legfőbb javaslatuk viszont, a C-szerű árnyaló programozási nyelv viszont bekerült a szabványba GLSL (OpenGL Shading Language) néven. Előnye, hogy ezzel az árnyalók programozása egyszerűsödött, ugyanis addig assembly-szerű nyelv állt csak rendelkezésre, amelynek használata nehézkes volt.

A legutolsó hivatalosan kiadott változatot, az OpenGL 2.1-t 2006 második felében adta ki az ARB. Ezzel egy időben az ARB is megújításra került, s az OpenGL szabványok felügyelete átkerült a Khronos-csoporthoz, ami szintén egy piaci érdekeket is figyelembe vevő, piaci résztvevőket tömörítő szervezet, de kezében tartja az összes OpenGL szabványt, az asztali számítógépekre készített szabványokat éppúgy, mint a mobil, illetve a beágyazott készülékekre megvalósítottakat, ezáltal fűzve szorosabbra a kapcsolatot köztük. A Khronos-csoport tagjai megtalálhatóak az A függelékben

A Khronos-csoport feladata mindemellett a legújabb OpenGL szabvány kidolgozása is, ami a közeljövőben 3.0-s verziószámmal jelenik meg, bár már 2007 végére ígérték, de újabb felülvizsgálatra visszavonták. Az új verzió ilyen alapvető átdolgozásának a hátterében a DirectX 10-s verziójának megjelenése áll, ugyanis ezzel a Microsoft nagy átalakításokat végzett az API-ján, s ezekkel a változtatásokkal az OpenGL-nek is lépést kell tartania, nehogy lemaradjon a versenyben, ami itt is, mint a piacon bárhol, erőteljes.

3.1.5. Az OpenGL 3.0 tervezett újításai:

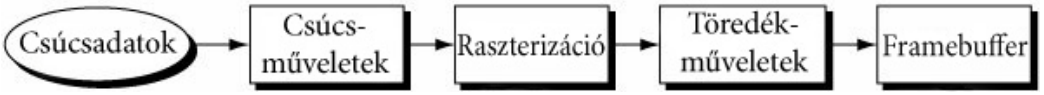
Az előző verziók óta a hardverek erőteljes feljődést mutatnak, új irányt vettek, ezért az API-t oly módon kell átalakítani, hogy a hardver változásokat jobban

követni tudja, közelebb kerüljön az eszközhöz. („Getting 'back to the bare metal' for performance" nyilatkozta Michael Gold[7] az Nvidia részéről.) Ezt pedig a legkönnyebb úgy elérni, hogy eltüntetik azokat a funkciókat, amelyeket már alig használnak, s csak a kompatibilitás megőrzése miatt hagytak meg eddig is meg. (ilyen például a közvetlen mód, immediate mode, `glBegin()` / `glEnd()`). A rögzített leképezési sor is megszűnik, és helyette teljesen programozható shadereket kell használni. Egy új objektum modell lesz az OpenGL 3.0 középpontjában. Ennek két fő oka van: az egész rendszer és a driver teljesítményének növelése, a kliensoldali objektum kezelés könnyítése és robusztussá tétele. Ez a modell lehetővé teszi a teljesítmény növelését azzal, hogy az egyazon objektumhoz tartozó összes tulajdonságot magában foglalja az objektum, és ezeket atomi egységként adja át az API-nak. Így biztosítható az átadott objektumok teljessége, elkerülve a banális, ám nehezen felderíthető programozói hibákat. Az általános cél, hogy csökkentsük a driver által keltett többletterheléseket, mindeközben egységesítve és egyszerűsítve az objektumok kezelését a programozó szemszögéből. Az OpenGl 3.0-ról részletes bemutató cikket írt Cyril Crassin[8].

3.2. Az OpenGL rögzített csővezetéke

Az OpenGL egy állapotgép. A program a függvényeket meghívja, s így állítja be az OpenGL állapotait, amelyek a rögzített leképezést befolyásolják. Ilyenek a színbeállítás, a normálvektor beállítás, az anyagbeállítás, a mátrixműveletek, a textúra-koordináták és a fények állítása.

Az OpenGL csővezetéke:



3.2.1. Csúcsadatok

Ezeket az adatokat adjuk meg, mikor meghívunk egy glVertex vagy glDrawArrays függvényt.

3.2.2. Csúcsműveletek

Az OpenGL a programtól kapott csúcsok mindegyikén elvégzi az alábbi műveleteket:

– Transzformáció: az OpenGL a csúcsokat az objektumkoordináta-térből ablakkordináta-térbe transzformálja.
– Megvilágítás: amennyiben az alkalmazás bekapcsolta a megvilágítást, kiszámolásra kerülnek az egyes csúcsok megvilágítási értékei.
– Levágás: ha egy primitív egyáltalán nem látható, mert kívül esik a nézet terén, minden csúcsa elvetésre kerül. Ha csak részben látható, csonkolással csak a látható része marad meg.

3.2.3. Raszterizáció

A raszterizáció során a geometriai adatokból töredékek (fragments) jönnek létre. A töredékek olyan helyzeti, szín-, mélység- és mintázatkoordináta adatokból álló egységek, amelyek még feldolgozásra kerülnek a képpufferbe írás előtt. A töredékek tehát különböznek a pixelektől. A pixelek azok a helyek a képpufferben, ahová a töredékek kerülnek. Általában egy töredék egy pixelhez rendelődik, de ettől eltérhetnek a multisampling (többszörös mintavételezéses) élsimítást támogató megvalósítások, ahol egy fragment kisebb, mint egy pixel.

3.2.4. Töredékműveletek

A töredékek a képpufferbe kerülés előtt még hosszas feldolgozás alá kerülnek. Ezalatt eldől, hogy ténylegesen bekerülnek-e a képpufferbe, illetve hogy milyen értékekkel kerülnek be.

Az alkalmazható műveletek:
– Ollóteszt: amennyiben a képpuffer egy adott helye kívül esik az alkalmazás által meghatározott ablakkoordináták téglalapján, az OpenGL elveti a töredéket.
– Áttetszőségi teszt: ha egy töredék alfa értéke nem felel meg a programban beállított követelményeknek (`glAlphaFunc()`), akkor a töredék elvetésre kerül.
– Stencil teszt: az OpenGL megvizsgálja a képpufferben tárolt stencilértéket, és az állapot beállításoknak megfelelően megállapítja, hogy a töredékkel mit tegyen, illetve hogy miként módosítsa a stencilértéket (`glStencilFunc()`, `glStencilOp()`).
– Mélységteszt: a mélységteszt akkor vet el egy töredéket, ha a töredék mélységértékének és a mélységpufferben tárolt értékkel való összehasonlítása megbukik. Ez az OpenGL mélységteszt, a Z-pufferelés egy fajtája.

– Blending: a blending a töredék RGB és alfaértékeit kombinálja a képpufferben tárolt RGB és alfaértékekkel.
– Dithering: ha a képpuffer színmélysége kisebb, mint a csúcsokhoz rendelt színmélység értékek, akkor az OpenGL átmenetet számol és azt írja a frampufferbe.
– Logikai műveletek: a végső színértéket a beállított logikai műveleteknek megfelelően írja a képpufferbe (`glLogicOp()`)

Ezen műveletek után előáll a megjeleníthető kép a képpufferben.

3.3. Az OpenGL segédkönyvtárai

Az OpenGL specifikáció csak és kizárólag a megjelenést írja le. Nincs benne támogatás ki- és bemeneti eszközök kezelésére. Ezeket a tevékenységeket tehát egyéb segédeszközökkel lehet megvalósítani.

3.3.1. GLU – OpenGL Utility Library

A GLU egy olyan (ARB által jóváhagyott) függvénykönyvtár, amely majdnem minden OpenGL implementációban megtalálható. A könyvtár olyan szolgáltatásokat nyújt, amelyek OpenGL-ből közvetlenül nem elérhetőek, szolgáltatásai magasabb szintű műveletekhez nyújtanak segítséget, például a kamera helyzetének és tájolásának szabályozása, a harmadrendű görbék és felszínek, több sokszögből álló primitívek rajzolása, a sokszögek háromszögelése, a leképezések egyszerű beállítása, textúra mipmap-ek generálása. A rendszer segítséget nyújt az OpenGL hibakódjainak kinyeréséhez, feldolgozásához.

3.3.2. GLUT – OpenGL Utility Toolkit

Az OpenGL csakis a grafikai elemek megjelenítéséért felelős, nincsenek benne ablak-, egér- és billentyűzetkezeléshez szükséges funkciók. Ám ezek nélkül nem tudunk megírni egy programot sem. Ugyanis ahhoz, hogy az OpenGL-l rajzolhassunk valamit, létre kell hoznunk egy grafikus kontextust, aminek kimenete az ablak, s interakciókat kell végeznünk az operációs rendszerrel. Ehhez el is készültek a megfelelő függvénykönyvtárak, WGL Windowson, GLX Linuxon, AGL MacOS-en, de ezek alkalmazásával platformfüggő kódot kell írnia a programozónak. Az ilyen problémák kiküszöbölésére hozta létre az SGI egyik alkalmazottja a GLUT függvénykönyvtárat, ami az ablakkezelést, a kontextus létrehozást, a bemeneti eszközök kezelését hivatott transzparens módon átvállalni a programozótól. Azaz ha minden platformra létrehoznak egy GLUT-ot, akkor

az OpenGL programok kvázi platform függetlenné válhatnak. Mindemellett a programozó pár sorral aktiválja a környezetet, és jobban tud koncentrálni az OpenGL-ben megoldandó feladatára.

Vannak hátrányai is, amelyek a tervezés alapelveiből következnek. Saját eseményciklust (`glutMainLoop()`) használ, amiből sohasem tér vissza. Ezáltal a programozó nem használhatja a saját ciklusát a program megvalósításához. Ezt az eseményciklust a főszálban kell meghívni, ami szinkronizációs problémákhoz vezethet.

3.3.3. Glee vagy GLEW

A Microsoft az ARB-ből való kilépése óta nem frissítette a rendszerbe integrált OpenGL függvénykönyvtárat, ami ebből kifolyólag az 1.4-es verzión ragadt. Ha olyan fügvényeket, újításokat szeretnénk kihasználni amelyek ezután készültek, akkor minden egyes alkalommal különféle kiterjesztéseken érhetjük el az amúgy meghajtó programban és hardveresen megvalósított funkciókat, vagy egy betöltő programot használunk, ami ezeket a betöltéseket számunkra láthatatlan módon végzi el. Ilyen függvénykönyvtár a Glee (GL Easy Extension library) és a GLEW (OpenGL Extension Wrangler Library).

4. Az árnyalók (shaderek)

A grafikus hardverek folyamatos fejlődésével a programozók részéről egyre nagyobb igény mutatkozott a fix funkcionalitású csővezeték adta lehetőségeknél nagyobb, a kép elkészítését jobban kontrolláló eszközök létrehozására. Ezt a hardvergyártók is felismerték, s újításokat vezettek be, melyek segítségével a csővezeték bizonyos részeit a programozó saját maga által írott kóddal helyettesítheti. Ezek a kis programok az árnyalók (shaderek).

4.1. Az árnyalók típusai

Két alapvető árnyaló jelent meg, a csúcsárnyaló (vertex shader) és a töredékárnyaló (fragment shader), ezek eleinte csak gyártóspecifikus kiterjesztéseken keresztül voltak elérhetőek, később ARB szabvánnyá léptek elő. Saját programozási nyelvvel rendelkeznek, ami leginkább az assemblyhez hasonlít. Gyors terjedésük hatására magasabb szintű nyelveket is létrehoztak, ami jelentősen leegyszerűsítette használatukat.

4.1.1. A csúcsárnyaló

A csúcsadatokon végrehajtható műveleteket lehet elvégezni bennük:

- A csúcs pozíciójának transzformációja.
- A normálvektorok számítása, állítása.
- A textúra-koordináta generálás, transzformálás.
- A csúcsszintű fényszámolás.
- A pixelszintű fényszámolás adatainak előkészítése.
- A csúcsszintű színértékek beállítása.

Nincs szükség az itt felsorolt összes művelet elvégzésére, de fontos megjegyezni, ha csúcsárnyaló programot írunk, akkor teljes mértékben lecseréljük a fix funkcionalitást. Ez azt jelenti, hogy nem számíthatunk arra, hogy majd mi megírjuk a normálvektorok kiszámításához szükséges csúcsárnyalót, s hogy a textúra-koordinátákat majd a fix funkcionalitás elvégzi. Ha árnyalót használunk, minden feladat elvégzése miránk hárul.

4.1.2. A töredékárnyaló

A csúcstranszformációk és a raszterizáció után előálló töredékek mindegyikén a programozó által írt töredékárnyaló program fog lefutni. A csúcsoknál

kiszámolt értékek az egyes töredékeknél interpolálva jelennek meg. Az elvégzendő műveletek:

– Színértékek, és textúra-koordináták számolása pixelenként.
– Textúrák felhasználása.
– Ködkészítés.
– Normálvektorok számolása a pixelenkénti megvilágításhoz.

Itt is elmondható, hogy nem fontos minden feladat elvégzése, de ha egyet mi magunk valósítunk meg, akkor ezzel minden feladat ránk hárul. Mivel a töredékárnyaló egyetlen töredéken manipulál, nincs tudomása a körülötte lévő más töredékekről. Fontos megemlíteni, hogy a töredék koordinátáit nem tudjuk megváltoztatni.

4.1.3. Geometriai árnyalók

Meglehetősen új fejlesztés, a geometriai árnyalókat a DirectX 10-es változata hozta magával jelentős újításként, s elkészültek az OpenGL-hez is kiterjesztésként. Használatuk nem terjedt el oly mértékben mint az előző két árnyalóé, tekintve, hogy a DirectX 10 jelenleg csak Windows Vistán használható, s ennek jelenleg elég kis részesedése van az operációs rendszerek piacán, OpenGL alatti változata pedig csak nVidia kártyákon érhető el. Jelentősége, hogy az elküldött csúcsadatokból újabb csúcsok adatainak kiszámítását teszi lehetővé. Egy csúcsból legfeljebb 128 új csúcsot lehet a jelenlegi meghajtó programokkal és eszközökkel elkészíteni[9]. Előnye lesz a közeljövőben, hogy segítségével sokkal részletgazdagabb modelleket lehet létrehozni kevesebb adatból. Mint a cikk is rámutat, még gyermekcipőben jár, tehát a jól bevált módszerekkel jelenleg nagyobb teljesítmény érhető el.

4.2. A GLSL – OpenGL Shading Language

Az OpenGL árnyalóinak nyelve, ami legelőször az OpenGL 1.5-ben jelent meg, az OpenGl 2.0-ig jelentős fejlődésen ment keresztül, s ekkor vált teljes mértékben a szabvány részévé. A GLSL egy C-szerű programozási felületet ad a programozók kezébe, amit sokkal könnyebben lehet használni, mint az eddigi assembly-szerű nyelveket, amelyek még gyártók között is eltérhettek. Előnye tehát a GLSL-nek, hogy több platformon működik, a megírt árnyalókat minden grafikus kártya tudja használni, amely támogatja a GLSL-t. Fontos megjegyezni, hogy a gyártó cég feladata elkészíteni a GLSL-fordítót, ami a saját kártyájának megfelelő kódot generál, ezáltal olyan optimalizációkat tudnak használni, ame-

lyet a programozó assemblyben csak nagyon nagy tapasztalattal, illetve sok árnyaló megírása után képes. A gyorsabb futtatás érdekében a felcserélhető utasításokat felcserélik, ezzel fedve el egyes utasítások hosszabb végrehajtási idejét. A grafikus hardver masszív párhuzamossága miatt a GLSL kódot a gyártó fordítója tudja párhuzamosságra is optimalizálni, de mindenképp figyelni kell az árnyaló kódjának a lehető legegyszerűbben tartására. A GLSL specifikációja, a hivatalos dokumentációban[10], használatának ismertetése, példákkal való illusztrálása pedig az OpenGL Shading Language c. könyvben[11] található meg.

A GLSL használatának folyamata:

– Csúcs- és töredékárnyalókat kell írnunk GLSL használatával, akár külön fájlban, akár a programkódban karakterláncként tárolva.
– Árnyaló objektumokat kell kérnünk a rendszertől, majd ebbe betöltenünk a forrásokat, és ezeket OpenGL függvényhívással le kell fordítanunk.
– Programobjektumot kell kérnünk a rendszertől, és ehhez kell az elkészített árnyaló objektumokat hozzácsatolni.
– Az így elkészített programot használat előtt be kell kapcsolni, így az ezt követő leképezési műveleteket már a beállított árnyalók fogják elvégezni.
– Leképezéskor kell beállítanunk az árnyalók által használt változók értékeit.

Az alábbi mintakód egy csúcs- és egy töredékárnyaló alkalmazását mutatja be. A csúcsárnyaló csak a transzformációs mátrixot alkalmazza a csúcspontokra, a töredékárnyaló pedig fix színnel rajzol.

Csúcsárnyaló:

```
void main(void){
	gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Töredékárnyaló:

```
void main(void){
	gl_FragColor = vec4( 0.5, 1.0, 0.0, 0.0);
}
```

C++ kódrészlet:

Objektumtárolókat hozunk létre, amelyekbe az OpenGL-től igényelt objektumainkat tesszük:

```
GLhandleARB FragmentShader,VertexShader,ShaderProgram;
```

Az objektumok kérése:

```
VertexShader=glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
FragmentShader=glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
ShaderProgram = glCreateProgramObjectARB();
```

Az árnyalók forráskódjainak átadása az OpenGL-nek:

```
char* vs= /* vertexshader kód */
char* fs=/* fragmentshader kód */
glShaderSourceARB(bezierVertexShader, 1, &vs, NULL);
glShaderSourceARB(bezierFragmentShader, 1, &fs, NULL);
```

A kódok fordítása:

```
glCompileShaderARB(bezierVertexShader);
glCompileShaderARB(bezierFragmentShader);
```

A lefordított árnyalók egy teljes árnyaló programhoz csatolása:

```
glAttachObjectARB(ShaderProgram ,VertexShader);
glAttachObjectARB(ShaderProgram , FragmentShader);
```

A program elkészítése:

```
glLinkProgramARB(ShaderProgram);
```

A program használatba vétele:

```
glUseProgram(ShaderProgram);
```

A `glUseProgram(0)` hívással térhetünk vissza a fix csővezetékhez.

4.ábra A fix funkcionalitású-(balra) és a pixelenkénti megvilágítás (jobbra)

4.3. Miért jó árnyalókat használni?

Az árnyalók azért lettek nagyon hirtelen felkapottak, mert olyan dolgokat lehet velük megvalósítani, amelyeket a fix csővezetékkel nem lehet, vagy ha meg is lehet, akkor gazdaságtalan, mert sok erőforrást vesz el a program más részeitől. Nem valósítható meg például a pixelenkénti megvilágítás. *(4. ábra)*

Ezzel a számítógépes grafika nagy lépést tett előre a valósághű megjelenítés felé. A programozók nagyobb szabadságot kaptak, amit nem is restek kihasználni. Nagyon fontos megemlíteni, hogy jelentős mértékben tehermentesíti a processzort, ha árnyalókat használunk. Árnyalóból ugyanis nem csak egy van a mai grafikus vezérlőkben, hanem általában 8–128 darab, ami lehet dedikált vagy egységesített. Ez azt jelenti, hogy a csúcspontok és a töredékek feldolgozása párhuzamosan is végezhető, ezáltal sokkal gyorsabb megjelenítést elérhetünk használatukkal.

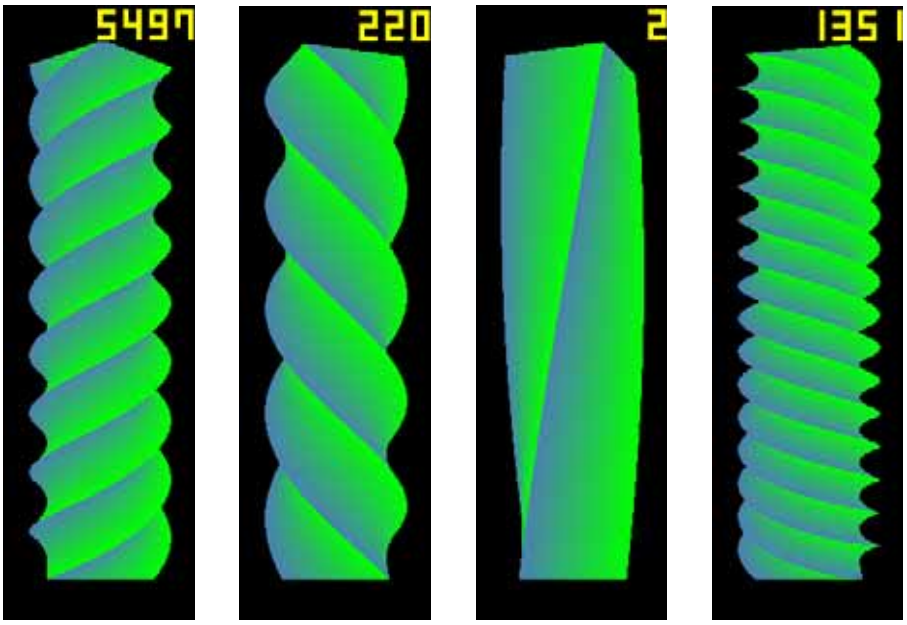
Az előző állításaimat alátámasztandó elkészítettem egy tesztprogramot. Azt az egyszerű problémát vettem alapul, hogy meg kell jeleníteni egy megcsavarodó négyzetes hasábot. Első feladat a hasáb modelljének előállítása. Legegyszerűbben nyolc ponttal tudunk megadni egy hasábot, de az OpenGL számára ez nem elég, ebből legfeljebb két négyszögünk lesz. Tehát az oldalakat is meg kell adnunk. Ha ezzel megvagyunk, akkor már ki tudunk rajzolni egy hasábot, ám megcsavarni még nem. A megcsavaráshoz az oldalakat további részekre kell felbontani. Minél részletesebben bontjuk fel, annál simább lesz a spirál. Árnyaló használata nélkül minden kirajzolás előtt el kellene készítenünk a megcsavarás következő lépését, s utána megjeleníteni a modellt. Árnyalóval elég mindig ugyanazt a modellt rajzolásra küldenünk, egy az árnyalónak szóló paraméterrel, ami tartalmazza a csavarás mértékét.

A tesztkonfigurációk:

– AMD Athlon64 3000+, nVidia Geforce 8600GT.
– Intel Core2Duo T7200, ATI Radeon Mobile HD2600

Az ábrán a programból mentett képernyőképek láthatóak *(5. ábra)*
Az eredményeket a következő táblázat foglalja össze:

Szeletek száma	*A64 3000+ GF8600*		*C2D T7200 HDm2600*	
	VS nélkül	*VS-el*	*VS nélkül*	*VS-el*
100	*1846 fps*	*5497 fps*	*1182 fps*	*1823 fps*
1000	*220 fps*	*1351 fps*	*228 fps*	*954 fps*
10000	*23 fps*	*164 fps*	*23 fps*	*127 fps*
100000	*2 fps*	*17 fps*	*2 fps*	*14 fps*



5. ábra A tesztprogram

5. A részecskerendszer első változatának mérlege

A hivatkozott cikkek[12][13] alapján kijelenthető, hogy nagyon fontos a részecskerendszer alapos megtervezése.

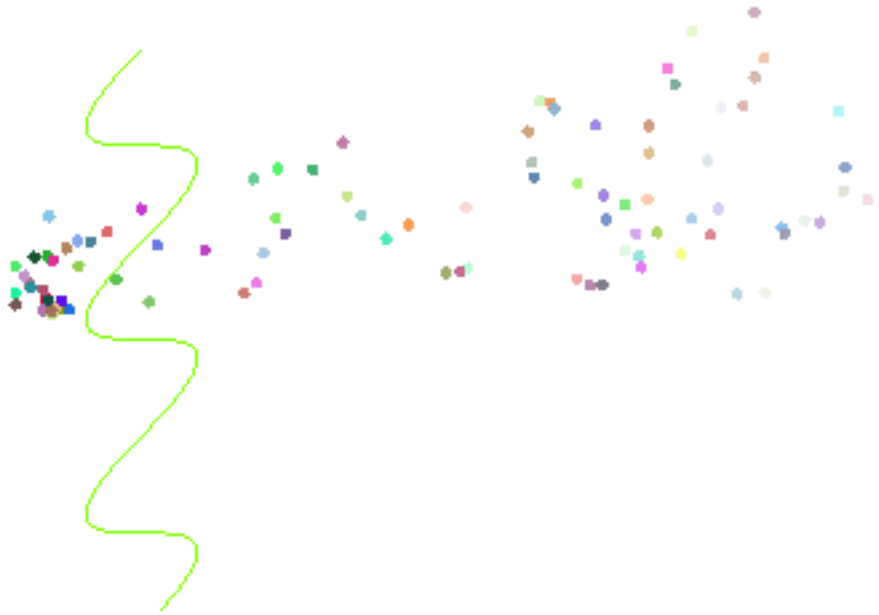
Átgondolva a kivitelezendő részecskerendszert, egyetemi tanulmányaim sugallatára leglogikusabbnak egy egyszerű módon felépülő, könnyen módosítható, átlátható osztályokból álló rendszert készítettem el. A *6. ábrán a* programról készült képet lehet látni.

Egy osztály reprezentál egy részecskét, a másik egy részecske forrást, s a harmadik pedig a részecskeforrások irányítóját.

A részecskeosztály kezdetben tartalmazta:

– a kiindulási pont koordinátáit,
– a haladási irányát,
– a színét,
– mennyi ideig élt,
– mennyi ideig élhet.

Az osztály rendelkezik konstruktorral, ami a részecske kiindulási pontját és maximális életét kapta paraméterül. A kezdőponton kívül minden tulajdonság



6. ábra A részecskerendszer első változata

(a maximális élet is, bár az paraméterezhető módon) véletlenszerűen került beállításra. Az első tervek szerint ezekből az adatokból minden egyes lépésnél ki lehetett volna számolni az aktuális értékeket, mint egy állapotmentes részecskerendszer esetén. Úgy látszott, ez elég is lesz, de a számolást olyan sokszor kellett végrehajtani, hogy jelentős teljesítménybeli csökkenést jelentett.Így bevezetésre kerültek az aktuális színt és pozíciót tároló változók is, amelyeknek a módosítása sokkal kevesebb számolással járt. Itt már kezdett elveszni a tiszta állapotmentes részecskerendszer jellege. A részecskék rendelkeznek újrainicializáló függvénynyel, amit a részecskeforrás hívhat meg, ha úgy érzékeli, hogy a részecske meghalt. E függvény bevezetésének oka szintén a teljesítményben keresendő. Már cikkekben is tárgyalták, hogy a sok memóriafoglalás és felszabadítás nagyon visszavetheti a részecskerendszer teljesítményét[13], s ennek számít a konstruktor és destruktor hívása is. Így inkább egy részecske életét figyeljük, s ha meghalt, egy függvény hívással azonnal új értékeket kaphat, ezáltal újra kezdheti életét.

5.1. A részecskeforrás

A részecskeforrás feladata abból áll, hogy egy görbe vonalát követve mozog, s kezeli a részecskék életciklusát. Ehhez tartalmazta a következőket:

– a görbe kontrollpontjai,
– pozíció a görbén,
– haladási sebesség,
– részecskék száma,
– részecskék objektumai.

Legelőször a megvalósítás úgy nézett ki, hogy minden alkalommal kiszámolásra került a részecske forrás helye, mikor egy részecske úrjainicializálása miatt erre szükség volt. A részecskék számának növelésével viszont olyan sokszor kellett ugyanazt az értéket újra és újra kiszámolni, hogy hatékonyabb volt letárolni az aktuális pozíciót.

5.2. A részecskeforrások irányítója

Ez az osztály tartalmazta a részecskeforrás objektumokat, s vezérelte azokat. Tudnia kellett mindemellett a görbe kontrollpontjait, ugyanis ha új forrást adtunk hozzá, akkor meg kellett adnia azt annak.

Ezek az osztályok először jó ötletnek, jól megtervezettnek tűntek. Azonban az egyik legfontosabb cél, nevezetesen a teljesítmény tekintetében, az elkészített

részecskerendszer kudarcot vallott. Sikerült alkotni egy okos félmegoldást. A részecskerendszer szépen működik, teszi, amit kell, de…

Lutz Latta[14] is rámutat arra, hogy a processzoron számolt részecskerendszerek egyik legnagyobb korlátozó tényezője a CPU és a GPU közti sávszélesség. Tényleges alkalmazásban ilyen módszerekkel maximum 15–20000 részecskét lehet kezelni jelentős teljesítményveszteség nélkül.

Továbbá, a kirajzoláshoz az OpenGL úgynevezett közvetlen módját használtam, ami már a kezdetek óta benne található, könnyű kezelni, ám ennek szintén kihatása van a teljesítményre. Ugyanis minden egyes részecske pontokként való kirajzolása, egy színbeállításból, és egy csúcspontadat küldésből áll, azaz részecskékként két függvényhívást jelent, ami óriási többletteher.

5.3. VBO – Vertex Buffer Object

A közvetlen mód leváltására hivatott megoldás a Vertex Buffer Object. Lényege, hogy kihasználja az OpenGL-ben megtalálható kliens-szerver megközelítést, s megszünteti a fölösleges adatforgalmat. Vegyük példának megint az árnyalóknál bemutatott demonstrációs programot. Ebben a programban eljutottunk addig, hogy mindig ugyanazokat az adatokat elküldve, egy árnyalónak szóló paraméter állításával megkaptuk a megcsavart hasábot. Jöhet akkor a kérdés, hogy miért küldjük el minden alkalommal az összes adatot. A válasz egyszerű: mert közvetlen módot használunk. Ilyen módban mindig el kell küldenünk az adatokat.

Az OpenGL 1.5-be került bele teljes mértékben a VBO. Segítségével a szerver oldalon, azaz a grafikus vezérlőn le tudunk foglalni egy tárolót, amibe bele tudjuk tölteni a kirajzolandó csúcsadatainkat. Majd a kliens oldalon (a programunkban) egyetlen hívással megkérhetjük a kártyát, hogy a nála tárolt adatokat rajzolja ki. Ezáltal az adatokat egyszer kell mozgatni, és razjolásonként egy függvényhívás szükséges. A programozók számára a VBO-k paramétereiről részletes leírást készített az nVidia[15], használatukról pedig Christophe Riccio[16].

A mérések eredményét az alábi táblázat tartalmazza:

Szeletek száma	*A64 3000+ GF8600*		*C2D T7200 HDm2600*	
	VBO nélkül	*VBO-val*	*VBO nélkül*	*VBO-val*
100	*5497 fps*	*6622 fps*	*1823 fps*	*2018 fps*
1000	*1351 fps*	*4624 fps*	*954 fps*	*1652 fps*
10000	*164 fps*	*878 fps*	*127 fps*	*514 fps*
100000	*17 fps*	*100 fps*	*14 fps*	*161 fps*

A részecskerendszerre visszatérve, VBO-t alkalmazni nem lehet rá, ami abból az egyszerű tényből következik, hogy objektumokat használ a részecskerendszer megvalósítása, s az OpenGL, bár tartalmaz objektumokat, nem objektumorientált módon épül fel. Nem tud adatokat kiszedni objektumban tárolt objektumokból, és felhasználni azokat rajzoláshoz vagy akár VBO-ba töltéshez. Ezt mindet a programozónak kell megvalósítani, létre kell hozni egy tárolót, olyan formában, ahogy az OpenGL elvárja, amibe belekerül minden részecske minden adata. Ezáltal feltölthetővé válik. A részecskék viszont folyamatosan mozognak, s mivel CPU-n kerülnek kiszámolásra, s nem állapotmentes módon, ezért az adatok, amelyeket el kellene helyezni a szerver oldalon, folyamatosan változnának, tehát nem tudnánk kiküszöbölni az összes adat feltöltését.

Ezzel az elképzelés teljes mértékben meg is bukott. Nem sikerült általános megközelítéssel jó teljesítményű részecskerendszert létrehozni, más irányba kellett elmenni, a problémára kellett erősen specializálni a részecskerendszert, úgy, hogy a CPU-nak minél kevesebb dolga legyen vele.

6. A végleges megoldás

Már a tervezés kezdeténél szükséges tudni azt, hogy milyen gyorsító funkciókat akarunk használni, s megismerni azoknak a tárolási módjait, akadályait, határait.

Így sikerült ráakadni egy olyan funkcióra, amely alapján a végleges program ötlete is megszületett. Ez alapján a program elkészült, s a kívánalmaknak is eleget tesz.

6.1. Csúcsponttextúrázás – Vertex Texturing, Vertex Texture Fetch

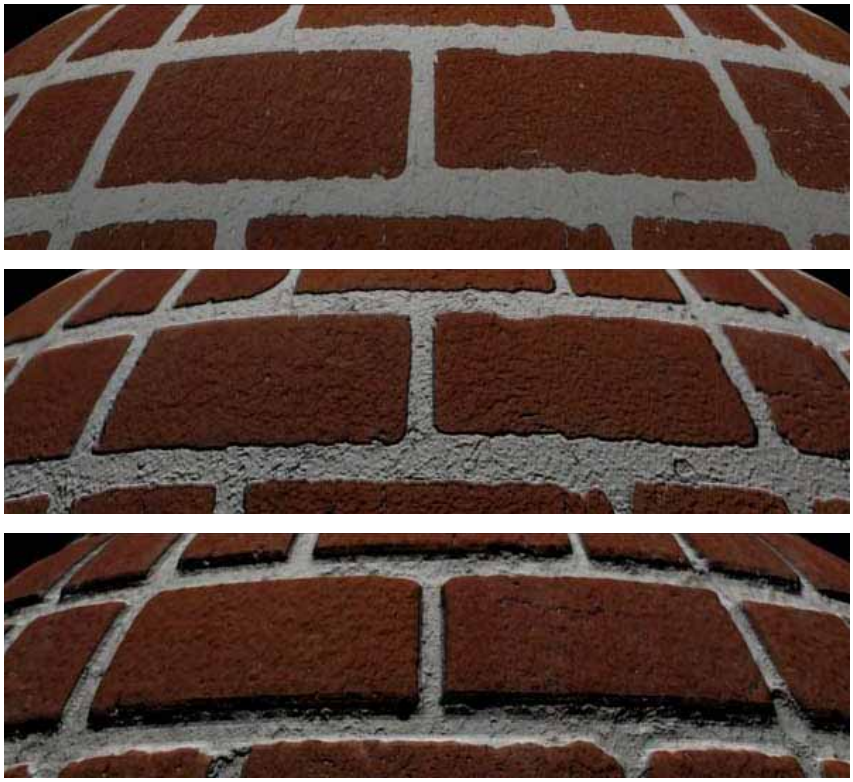
A programozható grafikus csővezeték bemutatkozása óta a csúcs- és töredékárnyalók teljesen más felépítésűek voltak. A Shader Model 3 bevezetésével nagy lépéssel közelebb kerültek egymáshoz funkcionalitásban. A VTF ugyanis lehetővé teszi a csúcsárnyalók számára, hogy textúrákból olvassanak, csakúgy, mint a töredékárnyalók. Ugyanis az a tendencia érvényesült, hogy a programok sebességének korlátozó tényezője a töredékárnyaló, memória sávszélesség vagy a központi processzor volt. Ezáltal, ha a komplexitást növeljük a csúcsárnyalóban, azzal sebességet nem vesztünk, viszont sokkal gazdagabb jelenetet hozhatunk létre. Ezt az extra komplexitást felhasználhatjuk olyan hatások készítésére, mint displacement mapping, folyadék szimuláció vagy robbanások modellezése.

6.2. Displacement Mapping

A bumpmapping utódjaként is lehet tekinteni. A bumpmapping egy textúrát használt, ami alapján a modell normálvektorait megváltoztatva olyan hatást keltett, mintha kitüremkedések és mélyedések lennének egy teljesen sima felület helyett. Ez a kis csalás akkor volt észrevehető, ha megfigyeltük a modell szélét. Ugyanis ott egyáltalán nem látszódott ez a hatás. A displacement mapping viszont a modell csúcsait változtatja meg, s ezzel nem csak hatást kelt, hanem tényleges magaslatok és hasadékok jönnek létre. Ezt természetesen meg lehet valósítani úgy is, hogy minden alkalommal módosítjuk a modellünket, de ezáltal az amúgy is terhelt részeket terheljük tovább (CPU, memória). A VTF-el viszont az egészet bevihetjük a csúcsárnyalóba, s kvázi teljesítmény veszteség nélkül készíthetjük el azt. A bumpmapping és displacement mapping közötti különbségek láthatóak a *7. ábrán.*

Fontos megemlíteni, hogy a csúcsárnyalóban történő textúra hozzáférés rendelkezik egy kis késleltetéssel. Erre a tényre maga az nVidia hívja fel a figyelmet[17]. A Shader Model 4 újításával viszont ez a késleltetés eltűnt, köszönhetően a legnagyobb újításnak, az egységesített árnyalóknak. A Shader Model 4 készítésénél azt figyelték meg, hogy a szoftverek, melyek árnyalókat használnak, általában csak az egyik árnyaló típust részesítik előnyben, míg a másik üresjáratban van. Ezt ellensúlyozták azzal, hogy csak egyfajta árnyaló van, s ez képes csúcs- és töredékárnyalóként is működni, méghozzá dinamikusan változtatva azt, hogy melyik fajtához tartozik. Ha most is létezne a késleltetés, akkor az érintené a töredékárnyalókat is, amelyeknek ez a fő feladata.

Egy másik különös dolog, hogy az nVidia mellett az ATI is jelentetett meg Shader Model 3-as grafikus vezérlőket, ám ezek a VTF-et mégsem támogatják[18].



7. ábra Normál megjelenítés (felül), bumpmapping (középen), displacement mapping (alul)

7. Eredmények – (statelessGPUparticlesystem)

A végső megvalósítás tehát nagyban támaszkodik az előzőekben ismertetett módszerre. A részecskerendszer megvalósítása teljes mértékben a csúcsárnyalón történik, amelyhez speciálisan előkészített textúrát és VBO-kat használtam fel. A modulszerű megvalósítás is sikerrel járt, próbaként témavezetőmmel rövid idő alatt könnyen be tudtuk illeszteni a rendszert egy már létező példaprogramba. Egy példányosítás, majd paraméterező függvény hívása után, a rajzolófüggvény meghívása már látható eredményt hozott az OpenGL környezeti állapotának megváltoztatása nélkül.

7.1. Az előkészítés

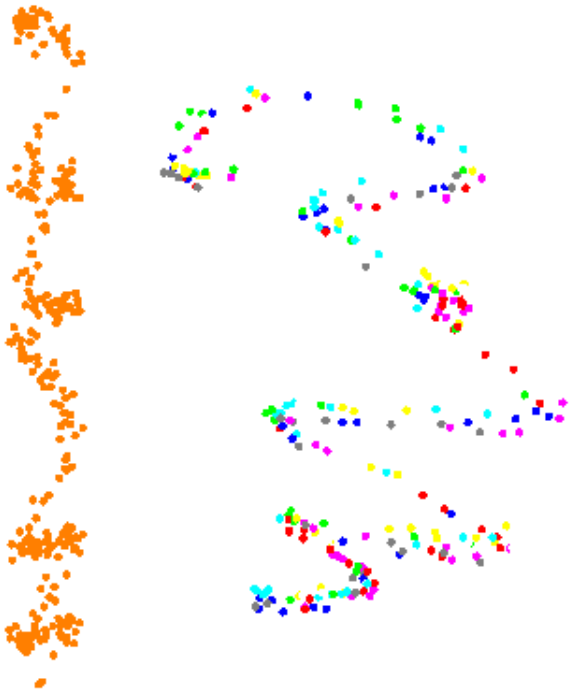
A részecskék egy görbe által leírt pályán mozognak. A választás a bézier ívre esett, mert könnyen számolható és egyszerűen csatolhatóak össze egymással, s vektorgrafikus rajzolóprogramokban is előszeretettel alkalmazzák. Természetesen a görbe típusa kis változtatással cserélhető.

A példányosítás folyamán a görbe pontjai fix részletességgel kerülnek kiszámolásra.

A kiszámolt pontok egy 128x128 képpontot tartalmazó textúrába kerültek, oly módon hogy egy bézier ív rész 128 ponttal lett kiszámolva. Így összesen 128 ívet lehet csatolni s eltárolni. Ezt természetesen meg lehet növelni 256x256-ra, esetleg nagyobbra, addig míg a vezérlő támogatja, de ezzel együtt nő az erőforrásigény is.

Ezután kerülnek létrehozásra a részecskék. Minden részecskét két lebegőpontos szám ad meg, s a látvány javítása érdekében egy vektor is definiálásra kerül, ami a részecskét mozdítja el a görbe ívéről, ezzel keltve a csőben való áramlás hatását. A két lebegőpontos szám egyike a sebességét adja meg, vagyis azt, hogy milyen gyorsan változtatja a helyét a görbén. A legkisebb sebesség 1/128, a másik adat pedig a kezdőpozícióját adja meg a görbén, egy lebegőpontos számba kombinálva. A szám egész része adja, hogy hányadik csatolt íven tart, a törtrész pedig az íven belüli pozíciót. Mivel 128 fix ponton számoltattam ki a görbét, ezáltal a törtrész 1/128 valahányszorosa.

Ezek után a létrehozott adatok áttöltésre kerülnek a kliensoldalról a szerver-oldalra a gyors elérés érdekében.



8. ábra Két részecskerendszer összesen 600 részecskével

9. ábra Két részecskerendszer összesen 6 millió részecskével

7.2. A használata

A rajzoló függvény meghívásával az osztályban eltárolt, a részecskerendszer aktuális pozícióját tartalmazó változó növelése történik meg, s ez az új érték paraméterként átkerül az árnyalóhoz. Az árnyaló ennek segítségével a kezdőpozícióból és a sebességből kiszámolja a részecske aktuális helyét a görbén a korábban említett formában, majd ezt felbontja az ívre és az íven belüli helyre, és egy VTF kéréssel a textúrából megkapja a részecske tényleges pozícióját a térben. Ez a pozíció kerül eltolásra a kisorsolt eltolás értékkel, s kerül megrajzolásra.

7.3. Értékelés

A kitűzött célok megvalósultak, a létrehozott modul könnyen beépíthető létező programokba. Teljesítmény terén is sikerként könyvelem el a megvalósítást, ugyanis a beépülő modul önmagában futtatva nem mérhető processzorhasználattal rendelkezik, akár 600 részecskéről van szó (*8. ábra*), akár 6 millióról (*9. ábra*). 6 millió fölött viszont hardveres korlátba ütköztem, elfogyott a grafikus memória. Ekkor a központi memória használata miatt drasztikusan leesett a másodpercenként megjelenített képkockák száma, s a CPU is teljes mértékben az adatok mozgatásával volt elfoglalva. Az elkészült program forráskódja, részletes magyarázatokkal, a B függelékben található.

7.4. Jövőbeli elképzelések, tervek

Mint minden elkészült szoftveren, a dolgozat mellett elkészült programon is számos olyan apróbb változtatást lehet eszközölni, amelyek a későbbiekben értékét növelhetik, használhatóságát könnyebbé tehetik,

- A görbe típusának változtathatósága.
- A fix 128 ponton való számolás dinamikussá alakítása konstruktor paraméter segítségével.
- Az eltolás mértékének megadása, felső korláttal.
- A sebesség beállításának lehetősége, szintén felső korlát megadásával.
- A részecskék áttetszőségének beállítása.
- A részecske méretének paraméterezhetősége.

8. Köszönetnyilvánítás

Köszönöm a sok szakmai segítséget témavezetőmnek, Dr. Tornai Róbertnek.

Külön köszönöm szüleimnek a sok bátorítást, segítséget.

Barátaimnak pedig köszönöm az építő jellegű kritikákat.

9. Hivatkozások

[1] Jean-Cristophe Lombardo. Modélisation d'objets déformables avec un système de particules orientées. *PhD thesis université Joseph Fourier*. p22 1996.

[2] William T. Reeves. Particle systems – techniques for modelling a class of Fuzzy objects. *ACM Transactions on Graphics Vol. 2, No. 2,* p92-95, 1983.

[3] William T. Reeves. Particle systems – techniques for modelling a class of Fuzzy objects. *ACM Transactions on Graphics Vol. 2, No. 2,* p91-92, 1983.

[4] Andreas Kolb, Lutz Latta, Christof Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware,* p123-125, 2004.

[5] P. Rousseau & V. Jolivet & D. Ghazanfarpour. Rendu réaliste de pluie en temps-réel. *Journées de l'Association Francophone d'Informatique Graphique, Bordeaux,* 2006

[6] OpenGL ARB. OpenGL Programming Guide: The Official Guide to Learning OpenGL., *Addison-Wesley Professional,* 2005.

[7] Micheal Gold. OpenGL 3 Overview. *Khronos Siggraph 2007 - OpenGL Birds of a Feather Presentation.* 2007. http://www.khronos.org/library/detail/siggraph_2007_opengl_birds_of_a_feather_bof_presentation/

[8] Cyril Crassin. OpenGL 3.0: Présentation. *http://icare3d.org/,* 2007.

[9] Suryakant Patidar, Shiben Bhattacharjee, Jag Mohan Singh, P. J. Narayanan. Exploiting the Shader Model 4.0 Architecture. *Center for Visual Information Technology, IIIT Hyderabad.* p4-5. 2007.

[10] John Kessenich. The OpenGL Shading Language Rev.: 8, *3Dlabs, Inc. Ltd.,* 2006. http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf

[11] Randi J. Rost. OpenGL Shading Language. *Addison Wesley Professional,* 2006.

[12] Jeff Lander. The Ocean Spray in Your Face. *Game Developer Magazine.* p13-19. 1998/07.

[13] John van der Burg. Building an Advanced Particle System. Game Developer Magazine. p44-50. 2000/03.

[14] Lutz Latta. Building a Million-Particle System. *Gamasutra,* 2004/07.

[15] NVIDIA Corp. Using Vertex Buffer Objects. Technical report, NVIDIA Corporation, 2003.

[16] Christophe Riccio. Vertex Buffer Objects. *http://www.gtruc.net*, 2006.

[17] Philipp Gerasimov; Randima Fernando; Simon Green. Shader model 3.0 using vertex textures. *Technical report, NVIDIA Corporation*, 2004.

[18] Fuad Abazovic. ATI has work around VTF lack. *The Inquirer*, 2005. http://www.theinquirer.net/en/inquirer/news/2005/10/25/ati-has-work-around--vtf-lack

A függelék – A Khronos-csoport tagjai

Kezdeményező tagok:

AMD
Apple Computer
ARM
Creative
Dell
Ericsson Mobile Platforms
Freescale Semiconductor
Imagination Technologies
Intel
IBM Corp.
Motorola
Nokia
NVIDIA Corporation
Samsung Electronics Co. Ltd
SK Telecom
Sony Computer Inc.
Sun Microsystems
Texas Instruments

Közreműködő tagok

Acrodea Inc.
ADI Tech
ALT Software
Anark Corporation
Antix Labs Limited
Aplix Corporation
ARC International
ArcSoft Inc.
Ardites Ltd.
AZTEQ Mobile
Barco
Beatnik
BitFlash
Blizzard Entertainment
Broadcom Corporation
Codeplay Software Ltd
Coding Technologies
Conexant Systems, Inc
Core Logic
Daimler
DAZ 3D
Diehl Aerospace
Digital Media Professionals Inc.
Emuzed
ETRI
Feeling Software
Fraunhofer IIS
Fujitsu
Futuremark Corporation
Google
Graphic Remedy
GraphTech Computer Systems, Ltd.
Hantro Products
HI Corporation
Hooked Wireless
HUONE
Ideaworks3D Ltd
Incoras Solutions
LG Electronics
Marvell
Matrox Graphics, Inc
Mentor Graphics
Micron Technology, Inc.
Mitsubishi Electric Corporation
Monotype Imaging
Movidia Limited
NDS Limited
NEC Electronics Corporation
NEC System Technologies, Ltd
NXP
Omegame

Panasonic
QNX Software Systems
QSound Labs, Inc.
QUALCOMM
Quantum3D
Renesas Technology Corp.
S3 Graphics Co., Ltd.
Sasken Communication Technologies Ltd
Scaleform
Seaweed Systems
Sharp Corporation
SKY MobileMedia
SOFTBANK MOBILE Corp
Softimage
Sony Ericsson Mobile Communications
SRS Labs, Inc.
STMicroelectronics
Sunplus
Symbian
TAKUMI Corporation
TAT
TES Electronic Solutions GmbH
Toshiba
TTPCom
Tungsten Graphics Inc
Vincent
Vivante Corporation
Yumetech, Inc

B függelék - Az elkészült program forráskódja

```
class GPUps{
public:
	int		sectionCount;
	//a csatolt bezier ívek száma
	int		particleCount;
	//részecskék száma
	float**		controlPoints;
	//bézier ívek kontrollpontjai Pont Érintő Pont Érintő felirasban
	float*		bezTexture;
	//bezier iv kiszamolt pontjai Vertex Texture Fetch-hez
	float*		particlesVBO;
	//részecskék kiindulasi pozicioja és sebessége VBO-ba töltéshez
	float*		particlesVBO_offset;
	//részecskék eltolása VBO-ba töltéshez
	float		color[3];
	//részecskék színe
	int		usecolor;
	//1 megadott szín használata
	//0 pepita :)
	GLuint		pointer_particleVBO[2];
	GLuint		pointer_bezTexture[1];
	GLuint		location_bezTex;
	GLuint		location_step;
	GLuint		location_sectCount;
	GLuint		location_usecolor;
	GLhandleARB	VertexShader,ShaderProgram;
	//Az OpenGL-től igényelt objektumok tárolói
	GPUps(int,float[][4],int);
	//Konstruktor
	//int kontrollpontok száma
	//float[][] kotrollpontok
	//int részecskeszám
	void rendermakestep();
	//A részecskék kirajzolása, és a rendszer léptetése.
	void setcolor(float,float,float);
	//A szín beállítása a paraméterként kapott (R,G,B) értékre.
	void setusecolor(int);
	//A szín használatának engedélyezése és tiltása.
	int step;
	//lépésszámláló
	int maxstep;
	//Megadja, hogy hány lépés után kell nullázni a lépésszámlálót.
};
```

```
void GPUps::rendermakestep(){
	glPushAttrib(GL_ALL_ATTRIB_BITS);
	glPushClientAttrib(GL_CLIENT_ALL_ATTRIB_BITS);
	GLint currentprog;
	glGetIntegerv(GL_CURRENT_PROGRAM,&currentprog);
	//A környezet állapotának lementése.
	glDisable(GL_TEXTURE_1D);
	glDisable(GL_TEXTURE_2D);
	glDisable(GL_TEXTURE_3D);
	//A Textúrázó funkciók kikapcsolása.
	//Ha bekapcsolva maradnak, a részecskék kirajzolása
	//nem történik meg.
	glBindBuffer(GL_ARRAY_BUFFER, pointer_particleVBO[0]);
	glVertexPointer(4, GL_FLOAT, 0, 0);
	glBindBuffer(GL_ARRAY_BUFFER, pointer_particleVBO[1]);
	glTexCoordPointer(4, GL_FLOAT, 0, 0);
	glActiveTexture(GL_TEXTURE0+pointer_bezTexture[0]);
	glBindTexture(GL_TEXTURE_2D, pointer_bezTexture[0]);
	//VBO-k és VTF beállítása
	glUseProgram(ShaderProgram);
	glUniform1f(location_step,(float)step);
	//Shader program alkalmazása és paraméterezése
	glColor3fv(color);
	//Szín beállítás
	glEnableClientState (GL_VERTEX_ARRAY);
	glEnableClientState (GL_TEXTURE_COORD_ARRAY);
	glDrawArrays(GL_POINTS,0,particleCount);
	glDisableClientState(GL_VERTEX_ARRAY);
	glDisableClientState (GL_TEXTURE_COORD_ARRAY);
	//VBO-k rajzolása
	step++;
	if(step+1>=maxstep)step=0;
	//Lépésszámláló növelése, szükség esetén nullázása
	glUseProgram(currentprog);
	glPopAttrib();
	glPopClientAttrib();
	//Környezet állapotának visszaállítása
}
void GPUps::setcolor(float r,float g,float b){
	color[0]=r;
	color[1]=g;
	color[2]=b;
	//Paraméterként kapott színek elmentése
}
void GPUps::setusecolor(int v){
	GLint currentprog;
	glGetIntegerv(GL_CURRENT_PROGRAM,&currentprog);
	glUseProgram(ShaderProgram);
	usecolor=v;
	glUniform1i(location_usecolor, usecolor);
	glUseProgram(currentprog);
	//A színhasználat vizsgálate a shaderben történik,
	//ezért a kapott paraméter átadásra kerül a shadernek
}
```

```
GPUps::GPUps(int cpC, float cp[][4], int pnum){
	glPushAttrib(GL_ALL_ATTRIB_BITS);
	glPushClientAttrib(GL_CLIENT_ALL_ATTRIB_BITS);
	//A környezet állapotának lementése
	sectionCount=(cpC-2)/2;
	step=0;
	color[0]=1.0f;
	color[1]=1.0f;
	color[2]=0.0f;
	usecolor=0;
	//Alapértelmezett kezdőértékek megadása
	maxstep=sectionCount*128;
	if(sectionCount>128){
		printf(“ERROR Too many bezier curve patches! \n”);
		exit(1);
	}
	//Maximális lépésszám kiszámolása,
	//túl sok ív esetén a program abortálása
	controlPoints=(float**)calloc(sectionCount*4,sizeof(float*));
	for(int i=0;i<sectionCount*4;i++)
		controlPoints[i]=(float*)calloc(4,sizeof(float));
	//Memóriafoglalások a kontrollpontok átalakításához
	// P1  E1  P2  E2  P3  E3  P4  E4 stb. alakról
	// P1  E1  P2  E2  P2  E2’ P3  E3  P3  E3’ P4 E4 stb alakra
	// a könyebb számolás érdekében
	// Ex’ pedig Ex középpontosan tükrözve a Px pontra
	for(int i=0;i<4;i++){
		controlPoints[i][0]=cp[i][0];
		controlPoints[i][1]=cp[i][1];
		controlPoints[i][2]=cp[i][2];
		controlPoints[i][3]=cp[i][3];}
	if(sectionCount*4>4)
	  for(int i=4;i<sectionCount*4;i+=4){
	    int sec=(i/4-1)*2;
		controlPoints[i][0]=controlPoints[i-2][0];
		controlPoints[i][1]=controlPoints[i-2][1];
		controlPoints[i][2]=controlPoints[i-2][2];
		controlPoints[i][3]=controlPoints[i-2][3];
		controlPoints[i+1][0]=
		  (float)(2.0*controlPoints[i-2][0]-controlPoints[i-1][0]);
		controlPoints[i+1][1]=
		  (float)(2.0*controlPoints[i-2][1]-controlPoints[i-1][1]);
		controlPoints[i+1][2]=
		  (float)(2.0*controlPoints[i-2][2]-controlPoints[i-1][2]);
		controlPoints[i+1][3]=
		  (float)(2.0*controlPoints[i-2][3]-controlPoints[i-1][3]);
		    controlPoints[i+2][0]=cp[4+sec][0];
		    controlPoints[i+2][1]=cp[4+sec][1];
		    controlPoints[i+2][2]=cp[4+sec][2];
		    controlPoints[i+2][3]=cp[4+sec][3];
		    controlPoints[i+3][0]=cp[5+sec][0];
		    controlPoints[i+3][1]=cp[5+sec][1];
		    controlPoints[i+3][2]=cp[5+sec][2];
		    controlPoints[i+3][3]=cp[5+sec][3];
	  }
```

```
bezTexture=(float*)calloc(65536,sizeof(float));
//VFT textúra foglalása
float actPos[3];
float t,t2,t3;
for(int sec=0;sec<sectionCount;sec++)
  for(int p=0;p<128;p++){
    int index=sec*512+p*4;
      t=(float)p/128.0f;
      int i=sec*4;
      t2=(float)(t*t);
      t3=(float)(t2*t);
      actPos[0]=(float)(
         t3*((-1.0f)*controlPoints[i][0]+
             (3.0f)*controlPoints[i+1][0]+
             (1.0f)*controlPoints[i+2][0]+
             (-3.0f)*controlPoints[i+3][0])+
         t2*((3.0f)*controlPoints[i][0]+
             (-6.0f)*controlPoints[i+1][0]+
             (3.0f)*controlPoints[i+3][0])+
         t*((-3.0f)*controlPoints[i][0]+
             (3.0f)*controlPoints[i+1][0])+
         controlPoints[i][0]);
      actPos[1]=(float)(
         t3*((-1.0f)*controlPoints[i][1]+
             (3.0f)*controlPoints[i+1][1]+
             (1.0f)*controlPoints[i+2][1]+
             (-3.0f)*controlPoints[i+3][1])+
         t2*((3.0f)*controlPoints[i][1]+
             (-6.0f)*controlPoints[i+1][1]+
             (3.0f)*controlPoints[i+3][1])+
         t*((-3.0f)*controlPoints[i][1]+
             (3.0f)*controlPoints[i+1][1])+
         controlPoints[i][1]);
      actPos[2]=(float)(
         t3*((-1.0f)*controlPoints[i][2]+
             (3.0f)*controlPoints[i+1][2]+
             (1.0f)*controlPoints[i+2][2]+
             (-3.0f)*controlPoints[i+3][2])+
         t2*((3.0f)*controlPoints[i][2]+
             (-6.0f)*controlPoints[i+1][2]+
             (3.0f)*controlPoints[i+3][2])+
         t*((-3.0f)*controlPoints[i][2]+
             (3.0f)*controlPoints[i+1][2])+
         controlPoints[i][2]);
      bezTexture[index]  =actPos[0];
      bezTexture[index+1]=actPos[1];
      bezTexture[index+2]=actPos[2];
      bezTexture[index+3]=1.0;}
      //A bézier görbe pontjainak kiszámolása
      //és elhelyzése a textúrában
particleCount=pnum;
particlesVBO_offset=(float*)calloc(particleCount*4,sizeof(float));
particlesVBO=(float*)calloc(particleCount*4,sizeof(float));
//VBO előkészítési terület foglalása
```

```
for(int i=0;i<particleCount;i++){
  particlesVBO[i*4]=(float(rand() % (sectionCount*128)) / 128);
//A részecskék helyének sorsolása
  particlesVBO[i*4+1]=(float)((rand() % 2)+1)/128.f;
//A részecskék sebességének sorsolása
  particlesVBO[i*4+2]=1.0f;
  particlesVBO[i*4+3]=1.0f;
//Jelenleg kihasználatlan értékek, de később felhasználhatóak
  particlesVBO_offset[i*4]=((float)(rand())/RAND_MAX/4)-0.125f;
  particlesVBO_offset[i*4+1]=((float)(rand())/RAND_MAX/4)-0.125f;
  particlesVBO_offset[i*4+2]=((float)(rand())/RAND_MAX/4)-0.125f;
//Az (x,y,z) tengely menti eltolás véletlenszerűsítése
  particlesVBO_offset[i*4+3]=1.0f;
//Jelenleg kihasználatlan érték, de később felhasználható
}
glGenBuffers(2,pointer_particleVBO);
glBindBuffer(GL_ARRAY_BUFFER, pointer_particleVBO[0]);
glBufferData(GL_ARRAY_BUFFER,
       sizeof(float)*4*particleCount,
       particlesVBO,
       GL_STATIC_DRAW);
glVertexPointer(4, GL_FLOAT, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, pointer_particleVBO[1]);
glBufferData(GL_ARRAY_BUFFER,
       sizeof(float)*4*particleCount,
       particlesVBO_offset,
       GL_STATIC_DRAW);
glTexCoordPointer(4, GL_FLOAT, 0, 0);
//PufferEK kérése, és feltöltése
glGenTextures(1, pointer_bezTexture);
glActiveTexture(GL_TEXTURE0+pointer_bezTexture[0]);
glBindTexture(GL_TEXTURE_2D, pointer_bezTexture[0]);
glTexParameteri(GL_TEXTURE_2D,
       GL_TEXTURE_MAG_FILTER,
       GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,
       GL_TEXTURE_MIN_FILTER,
       GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,
       GL_TEXTURE_WRAP_S,
       GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,
       GL_TEXTURE_WRAP_T,
       GL_REPEAT);
glTexImage2D(GL_TEXTURE_2D,
       0,
       GL_RGBA_FLOAT32_ATI,
       128,
       128,
       0,
       GL_RGBA,
       GL_FLOAT,
       bezTexture);
//Textúrahely foglalása, és VTF textúra feltöltése szeveroldalra
```

```
VertexShader=glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
const char * v =
 “uniform sampler2D       bezTex; \n\
  //textúra mintavételező\n\
  uniform float           step;\n\
  //lépésszámláló\n\
  uniform float           sect;\n\
  //csatolt bézier ívek száma\n\
  uniform int             usecolor;\n\
  //színhasználat
  void main(void){\n\
       vec4 inn= gl_Vertex;\n\
       vec4 off= gl_MultiTexCoord0;\n\
       //Bemenő adatok vec4-be töltése\n\
       float actualpos=inn.x+step*inn.y;\n\
       //aktuális pozíció kiszámítása
       float yc=floor(actualpos);\n\
       //egészrész képzés\n\
       float xc=actualpos-yc;\n\
       //törtrész képzés\n\
       yc=(yc-(floor(yc/sect)*sect))/128.0;\n\
       //az egészrész modulo ja az ívek számával
       vec2 coord=vec2(xc,yc);\n\
       vec4 bezpos=texture2D(bezTex,coord);\n\
       //Pozíció kiolvasása a textúrából\n\
       if(usecolor==1)gl_FrontColor = gl_Color;\n\
       else{vec4 col=vec4(0.0,0.0,0.0,1.0);\n\
             if(off.x>0.0)col.r=1.0;\n\
             if(off.y>0.0)col.g=1.0;\n\
             if(off.z>0.0)col.b=1.0;\n\
             gl_FrontColor = col;}\n\
       //Szín beállítás pepita/megadott színre\n\
       bezpos+=off*inn.z;\n\
       bezpos.w=1.0;\n\
       //eltolás\n\
       gl_Position = gl_ModelViewProjectionMatrix * bezpos; \n\
       //a végső pozíció továbbadása megadása\n\
  }”;
glShaderSourceARB(VertexShader, 1, &v, NULL);
glCompileShaderARB(VertexShader);
ShaderProgram = glCreateProgramObjectARB();
glAttachObjectARB(ShaderProgram ,VertexShader);
glLinkProgramARB(ShaderProgram);
GLint currentprog;
glGetIntegerv(GL_CURRENT_PROGRAM,&currentprog);
glUseProgram(ShaderProgram);
//Aktuális shader program mentése
location_bezTex=glGetUniformLocation(ShaderProgram,”bezTex”);
location_usecolor=glGetUniformLocation(ShaderProgram,”usecolor”);
location_step=glGetUniformLocation(ShaderProgram,”step”);
location_sectCount=glGetUniformLocation(ShaderProgram,”sect”);
//ShaderProgram paraméterhelyeinek kérése
```

```
	glUniform1i(location_bezTex, pointer_bezTexture[0]);
	glUniform1i(location_usecolor, usecolor);
	glUniform1f(location_sectCount, (float)sectionCount);
	//Nem változó paraméterek átadása
	glUseProgram(currentprog);
	glPopAttrib();
	glPopClientAttrib();
	//Mentett állapotadatok visszaállítása
	free(particlesVBO);
	free(particlesVBO_offset);
	free(bezTexture);
	for(int i=0;i<sectionCount*4;i++) free(controlPoints[i]);
		free(controlPoints);
	//Az előkészítéshez foglalt területek felszabadítása
}
```