Multidiszciplináris tudományok, 3. kötet. (2013) 2. sz. pp. 259-268.

OSZTOTT 2D RASZTERIZÁCIÓS MODELL TÖBBMAGOS PROCESSZOROK SZÁMÁRA

Mileff Péter
Adjunktus, Miskolci Egyetem, Informatikai Intézet, Általános Informatikai Tanszék
3515 Miskolc, Miskolc-Egyetemváros, e-mail: mileff@iit.uni-miskolc.hu

Dudra Judit
Tudományos munkatárs, Bay Zoltán Alkalmazott Kutatási Közhasznú Nonprofit Kft.
3519 Miskolc, Iglói út 2., e-mail: judit.dudra@bayzoltan.hu

Összefoglalás

A grafikus feldolgozó egység (GPU) mára életünk szerves részévé vált mind az asztali mind pedig a hordozható eszközök révén. A dedikált hardvernek köszönhetően a vizualizáció jelentősen felgyorsult, a szoftverek pedig kizárólag ma már csak a GPU-t használják raszterizációra. Nem szabad elfeledkeznünk azonban, hogy a grafikus eszközök mellett a központi egység (CPU) is jelentős fejlődésen ment keresztül. Megjelentek a többmagos architektúrák és modern utasításkészletek, melyek új lehetőségeket rejtenek magukban. Jelen publikáció célja annak a vizsgálata, hogy egy mai többmagos központi egység hogyan és milyen hatékonysággal illeszkedhet be a kétdimenziós raszterizáció folyamatába, milyen előnyökkel és szűk keresztmetszetekkel rendelkezik az erre alapozott vizualizációs modell. Továbbá választ kapunk arra a kérdésre, hogy lehetséges-e egy olyan szoftveres megjelenítő motort készíteni, amely megfelel a mai játékok igényeinek.

Kulcsszavak: *szoftveres raszterizáció, párhuzamos programozás, sebesség optimalizálás*

Abstract

The graphics processing unit (GPU) has become part of our everyday through desktop computers and portable devices. Because of the dedicated hardware the visualization has been significantly accelerated and today's software uses only the GPU for rasterization. Beside the graphical devices, the central processing unit (CPU) has also made remarkable progress over. The multi-core architectures and new instruction sets have been appeared. This paper aims to investigate how effectively this multi-core architecture can be applied at the two-dimensional rasterization process, what are the benefits and bottlenecks of this rendering model. We answer the question, that would it be possible to design a software rendering engine to meet the requirements of today's computer games.

Keywords: *software rasterization, multi-threading, performance optimization*

1. Bevezetés

A számítógépes grafika területét napjainkban a GPU-k piaca utalja, amely sok éves fejlődés eredményét tükrözi. Kezdetben nem állt rendelkezés grafikus gyorsító eszköz, így minden számítást a központi egység (CPU) végzett el. Az ipar azonban hamar felismerte,

hogy megjelenítésben alkalmazott számítások eltérő igényekkel rendelkeznek, mint egy általános szoftver, egy megfelelő céleszköz segítségével így jelentősen gyorsíthatók. Az átalakulás legfőbb indítómotorja tehát a grafikai számítások, a képi minőség növelésének megcélzása volt. Mivel kezdetben a központi egység fejlődése korántsem zajlott olyan gyors ütemben, mint ma, így a piaci igényeknek megfelelően egy dedikált hardver, a GPU vette át a raszterizációs feladatokat.

A grafikus chipek folyamatos szárnyalása mellett azonban a központi egységek is folyamatosan fejlődtek. A fejlesztés vonala bár nem volt annyira látványos, mint a GPU-ké, kétségtelenül jelentős. Egy-egy új GPU vagy videokártya megjelenését mindig nagyobb média reklám övezte szemben a központi egységekkel. A CPU-k esetében kezdetben két vonal bontakozott ki. Első megközelítésben a központi egységek számának növelését tartották célszerűnek (Multiprocessing Systems), majd ezek után a központi egység magjainak a számának növelése lett a cél. 2005-től pedig már megjelentek az első többmagos (2) processzorok teljesen új irányt adva a fejlődésnek.

Ma már kijelenthetjük, hogy a magok számának növelése jelentősen előremozdította mind az operációs rendszerek használhatóságát (multitasking), mind a többszálas alkalmazások fejlesztését. Egy a hardver lehetőségeit megfelelően kihasználó többszálas szoftver teljesítményben messze felülmúlhatja a klasszikus megközelítést alkalmazó szoftverek teljesítményét. A tendenciát megfigyelve jól látható, hogy a processzorgyártó és tervező cégek berendezkedtek a mag alapú processzorok tervezésére és gyártására. Ma már a mobil eszközökben is több (2-4) mag található, a PC-k esetében pedig a magok száma már a 8 darabot is eléri köszönhetően a Hyperthreading technológiának.

A magok számának növekedése mellett a processzor gyártók kibővített utasításkészlettel reagáltak a piaci igényekre, lehetővé téve a központi egységeken is a gyorsabb és főként vektorizált (SIMD) feldolgozást. Csaknem minden gyártó elkészítette saját bővítését. Úgy mint az Intel által kidolgozott és mára már szinte minden központi egység által támogatott MMX, SSE és AVX (2008) utasításcsalád. Az AMD kezdetekben a 3DNow csomagjával próbált erősíteni, napjainkban pedig az ARM Cortex-A8 típusú architektúráknál kezdetben bevezetett Vector Floating Point (VFP) technológia, majd pedig a NEON utasításkészlet a fejlődés iránya.

Az új technológiáknak köszönhetően célszerű tehát újragondolni a raszterizáció folyamatát. Van-e szükség új modellre és kihasználhatók-e az új lehetőségek. Van-e alapja egy olyan szoftveres renderer készítésének, amely képes a mai számítógépes játékok igényeinek megfelelni.

2. Irodalmi áttekintés

A szoftveres képszintézis az első számítógépek óta jelen van domináns szereppel bírva egészen 2003-ig, a GPU megjelenéséig. A korai évek során született raszterizálók közül legkiemelkedőbb eredmény az ID software által készített Quake I, II szoftveres renderer (1996), amely az első MMX utasításkészletre optimalizált valós háromdimenziós motor volt. A később született eredmények közül főleg az Unreal motort (1998) lehet kiemelni, amely szintén nagyon gazdag funkcionalitással rendelkezett.

A GPU renderelés folyamatos térnyerése után a szoftveres megjelenítés egyre inkább háttérbe szorult. Ennek ellenére született néhány nagyszerű eredmény, mint a Rad Game

Tools által fejlesztett Pixomatic 1, 2, 3 [9] és a TrasGaming által készített Swiftshader [3]. Mindkét termék nagyon komplex, magas szinten optimalizált raszterizációt tesz lehetővé.

Bár a Microsoft a DirectX fejlesztésével nyújtott alapot a GPU technológiák terjedésének, mindezek mellett kifejlesztette saját szoftveres megjelenítőjét a WARP-ot. A renderer egyaránt jól skálázható több szálra és teljesítményben felveszi a versenyt az alacsony kategóriás integrált grafikus kártyákkal.

A problémákat és az igényeket jól felmérve az Intel 2008-ban egy hibrid, x86 alapú videokártya (Larrabee) fejlesztését tűzte ki célul [4], melynek célja egy teljesen programozható csővezeték kidolgozása volt [4].

Az NVidia saját GPGPU CUDA platformjának segítségével készített tanulmányt a szoftveres megjelenítésről [5]. Napjaink vezető grafikai színvonalú számítógépes játéka [8] új technológiájaként egy részben SPU alapú raszterizációt valósított meg a fényforrás hatékonyan kezelésére. Az [2] publikációban a szerző egy több szálat használó modern „Tile" alapú megjelenítő technikát vázol.

3. 2D vizualizáció gyorsítása szálkezeléssel

A raszterizáció nagyon számításigényes folyamat, főként nagyszámú alfa csatornát tartalmazó képi elemek estében [1]. A mai grafikus alkalmasokban pedig dominálnak ezek a megoldások a jobb vizuális élmény végett. Mivel a grafikus motor a képi elemeken ilyenkor pixelenként halad végig sok elem esetében ez több ezer függvényhívással és redundáns számítással jár. Pixelről pixelre külön ki kell olvasni a memóriából a képpont színét, majd a környezeti adatok függvényében pedig meghatározni a képernyőn való pozícióját, és elvégezni az új szín framebufferbe való írását. A klasszikus megközelítésben ezt a folyamatot a CPU egy feldolgozó szállal végzi el. Ez megközelítés azonban teljesítmény szempontjából nem elég hatékony, mert a modell nem veszi figyelembe a rendelkezésre álló hardver jellegzetességeit [1].

A modern hardveregységekben rejlő párhuzamosítási lehetőségek jó alapot biztosítanak egy nagy teljesítményű, fokozott párhuzamos számítási környezet kialakításához a számítógépes vizualizáció területén belül is. Nyilvánvaló, hogy a raszterizáció folyamatában ki kell használni ezeket és új megközelítésű megjelenítő modellt kell tervezni. Egy párhuzamosítási technológiával elkészített szoftveres renderer várhatóan lényegesen jobb eredményt képes elérni, mint a klasszikus megközelítés. A továbbiakban egy ilyen modell alapjait vázolunk részletes tesztelési eredményekkel alátámasztva.

3.1. Elosztott raszterizációs modell

A raszterizáció során tehát célszerű a megjelenítő motor tervezését valamilyen osztott modellre alapozni. Olyan megoldást kell keresni, amely jól párhuzamosítható folyamatokból építhető fel. A párhuzamosítás mértéke azonban maximalizált. Amdahl [10] törvénye szerint egy szoftver párhuzamosíthatóságának mértéke nagymértékben függ a nem párhuzamosan végrehajtandó kód mennyiségétől. Ezért elsőként meg kell vizsgálnunk azt, hogy maga a renderelési folyamat milyen mértékben feleltethető meg ezen elveknek.

A párhuzamosítás technológiai alapját a CPU-ban rejlő magok megfelelő kihasználása adja, amelyet a szálak segítségével tehetünk meg leghatékonyabban. Mindamellett, hogy a

szálkezelés számos optimalizációs kérdés felvet, fontos szabályként a kialakított modellnek figyelembe kell venni az aktuális hardver központi egységében rejlő magok számát. A teljesítmény nem optimális, ha az adott szoftver feldolgozási szálainak száma meghaladja a CPU magok számát. Amikor is a logikai szálak száma eléri a rendelkezésre álló hardveres szálak számát, a teljesítmény lassan csökkenni kezd a kontextus váltások miatt [10].

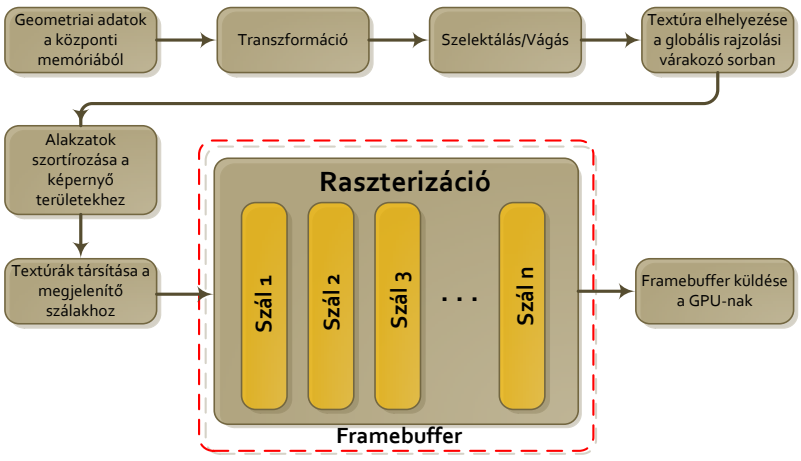
Szerencsére a kétdimenziós vizualizáció folyamata matematikai szempontból egyszerű, a párhuzamosítás így könnyebben értelmezhető, mint a 3D esetében. A renderelés alapja a pixelek framebufferbe való írása. Mivel a pixelek függetlenek egymástól, így a renderelés folyamata bizonyos kritérium mellett nagyon jól párhuzamosítható. A szükséges kritérium ehhez az, hogy a framebuffer egyazon pontját két szál ne írja egyszerre, valamint a szálak közötti szinkronizáció minimális legyen. Amennyiben meg tudjuk oldani, hogy a szálak ne várjanak egymásra a framebuffer írása közben, úgy a teljesítménynövekedés várhatóan jelentős lesz. Az eredmény egy osztott raszterizációt megvalósító többszálas raszterizációs modell.

Mivel a raszterizációs folyamatot leginkább az átlátszó objektumok kirajzolása lassítja, így a továbbiakban főként e kategóriával foglalkozunk.

3.1.1 Osztott renderelő modell

Egy minimális szinkronizációt megvalósító raszterizációs modell tervezésekor a megjelenítendő felület területének logikai felosztásából kell kiindulnunk hasonlóan a „Tile” alapú megjelenítéshez. Mivel a felosztás meghatározza a pixeleket tároló mögöttes framebuffer felosztását is, így célszerű egymástól független területekben gondolkozni. Ekkor, ugyanis ha a terület megjelenítését külön feldolgozó szálak végzik, úgy a szükséges szálak közötti szinkronizáció minimális lesz. Ennek oka az, hogy egyik megjelenítő szál sem fog a másik területéhez tartozó framebuffer részben pixelműveleteket végezni. Példaként egy 4 magos központi egység számára a framebuffer négy részre bontható, a megjelenítés pedig négy szálban párhuzamosan végezhető el.

A következő ábra a megjelenítés logikai lépéseit mutatja be:



1. ábra. *Osztott raszterizáció modellje*

Az alkalmazott grafikus csővezeték első része azonos a klasszikus megoldással, amikor is első lépésként az objektumok pozíciójának és helyzetének kiszámítása történik meg és az, hogy benne van-e a képernyő tartományában. Amennyiben látszik az objektum akár egy pixele is, úgy a megjelenítése elengedhetetlen. Ennek folyamata azonban eltér a klasszikus eljárástól. Míg korábban egy valamilyen *Draw* metódus meghívásával az objektum képe rögtön a framebufferbe íródott, itt szükség van egy olyan konténerre, amely a kirajzolás során összegyűjti ezeket a textúrákat egy listára. A tényleges raszterizáció akkor következik, amikor minden kirajzolásra szánt objektum képe szerepel a listán.

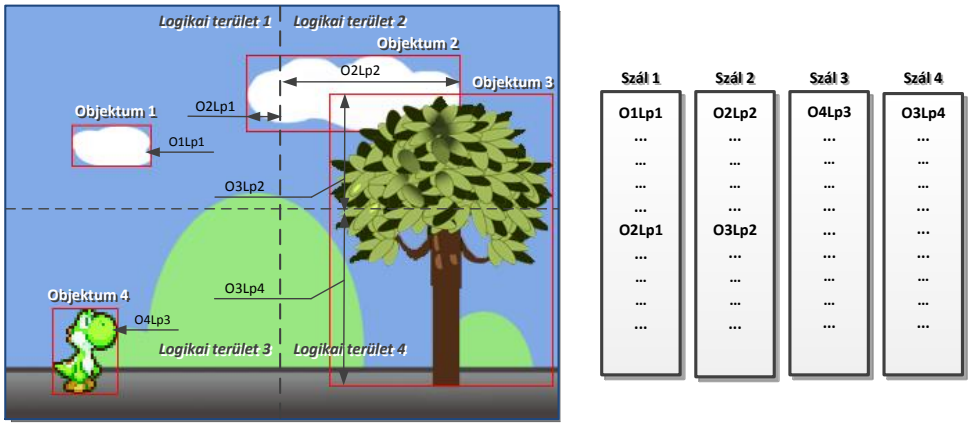
A párhuzamos raszterizáció alapja az, hogy a framebuffer különböző területeit egymástól függetlenül párhuzamosan lehessen írni. Ennek megvalósításához szükség van egy gyors osztályozó algoritmusra, amely eldönti, hogy az aktuális objektum melyik területhez fog tartozni, melyik renderelést végző szál fogja kirajzolni. Az osztályozási feladat azonban egy újabb kérdést is felvet. Biztosan lesznek olyan objektumok, melyek képei átfedik az egyes logikai területeket. Az osztályozás során így mindkét szál megkapná a textúrát renderelésre, amely már megsérti a lefektetett párhuzamossági szabályokat. Ennek megoldására egy egyszerű modellként alkalmazható az a megközelítés, ahol az osztályozó algoritmus olyan elven társítja a területekhez a textúrákat, hogy az egyszerre több területhez tartozó elemeket mindig egy specifikus, az alkalmazás fő szálához társítja. Az ehhez a szálhoz tartozó textúrákat ebben az esetben csakis a többi szál renderelése előtt vagy után lehet elvégezni, mivel az átlapolások miatt a szál a framebuffer többi szál logikai területeit is érinti. A megoldás bár működőképes és gyors is, azonban nem alkalmazható minden esetben. A számítógépes játékokban általában előre meghatározott megjelenítési sorrendre van szükség (pl. a repülő bizonyos felhők előtt, és bizonyos felhők mögött repül). Ez a megoldás az átlapolt területek miatt azonban nem tartja meg a renderelési sorrendet. A modell egy olyan kiegészítése, amely megtartaná a kirajzolási sorrendet túl sok kommunikációt igényelne a szálak között, amely pedig a párhuzamosság rovására menne.

Ahhoz, hogy a modell megtartsa a párhuzamosítás előnyeit, az átlapolási probléma megoldására van szükség. Mivel kétdimenziós megjelenítésről van szó, így minden objektum képe egy téglalappal leírható Ez maga a textúra szélessége és magassága által határolt terület. A probléma megoldásának hatékony megközelítése az, ha minden objektumhoz tartozó képet elvágunk a logikai területhatárok mentén. Ehhez az osztályozási folyamat módosítására van szükség. Míg korábban minden több logikai területet átfedő textúra a fő render szálhoz társult, ebben a modellben az ilyen textúrákat minden olyan logikai területhez társítunk, amelybe belelóg.

A továbbiakban egy négy magos központi egységre szánt négy logikai területbontást vizsgálunk meg. Leginkább komplex esetben ilyenkor a textúrát négy darabra kell vágni. Valójában nem tényleges vágásról beszélünk, csupán a megjelenítendő területek korlátozásáról. A következő ábra az átnyúló objektumok esetét mutatja be.

A megjelenítés folyamata során tehát további ellenőrzésekre és számításokra van szükség. Mivel egy logikai területhez átlapolt textúrák is tartozhatnak, így a tényleges pixel szintű raszterizáció során meg kell határozni a renderelés pontos pixel szintű határait, hogy elkerüljük a másik szállal való ütközést. A fenti vágás szó erre utal. Mindezek plusz erőforrást igényelnek a CPU-tól a korábbi egyszerű megoldáshoz képet, de mégsem annyira jelentősek, hogy gyökeresen befolyásolnák a sebességet. Jól látható azonban, hogy a megoldás megtartja a kirajzolási sorrendet. Ha a megjelenítési listát az osztályozási

folyamat előtt a megfelelő sorrendbe rendezzük, a sorrend a raszterizáció közben is megmarad.



2. ábra. *Normál és átnyúló objektumok rajzolási szálakhoz való rendelése*

3.1.2 A megjelenítési rendszer jellemzői

Természetesen mivel szálakat alkalmaz a megjelenítő rendszer, így elkerülhetetlen a szálak közötti szinkronizáció. A modellben két helyen van szükség szinkronizációra. Elsőként, amikor az osztályozási folyamat után a szálak megkapják a feladatokat, és szabad feldolgozás jelzést kapnak. A második pont pedig a raszterizáció végén kell legyen azért, hogy megvárjuk az összes szál munkájának befejezését. Ezeket a pontokat a fő szálban célszerű elhelyezni, a változók megosztását pedig mutex-ekkel megvalósítani. Mivel a szál létrehozási folyamat költséges, ezért célszerű úgy megvalósítani a megjelenítő rendszert, hogy a szálak ne az osztályozási folyamat után jöjjenek létre, és ne szűnjenek meg a raszterizáció végén lévő szinkronizációs pont után. Helyette a szálak folyamatosan élnek. Amikor szükség van a munkájukra, akkor aktivizálódnak, egyébként pedig várakoznak. Ezen elvek alapján megvalósított megjelenítő előnye az, hogy várhatóan lényegesen jobb sebességgel fog rendelkezni. Hátrányként viszont elmondható, hogy a megvalósítás komplexebb.

Bár a többszálas megjelenítő bemutatását főként az átlátszó textúrák megjelenítésére éleztük ki a pixel szintű műveletek miatt, az elv alkalmazható nem átlátszó textúrák esetében is, kirajzolásuk beépíthető a modellbe a folyamat bővítésével. E típusú textúrák kirajzolása egy memória másolás műveletével gyorsabban és hatékonyabban elvégezhető [1]. Ahhoz, hogy ezt a tulajdonságot megtartsuk, a grafikus motornak már a képek betöltése után regisztrálnia kell, hogy mely textúrák átlátszóak és melyek nem. A kirajzolás folyamata során ezek a textúrák is azonos listára kerülnek az átlátszókkal. Az osztályozó algoritmus hasonlóan egy vagy több logikai területhez fogja rendelni őket. A tényleges kirajzoláskor azonban pontosan tudni kell a textúra típusát, mert az átlátszó képeket pixelenként, a nem átlátszókat pedig memóriamásolással (beleértve a területek logikai határainak ellenőrzési és „vágási” folyamatát) lehet leghatékonyabban kirajzolni. Ezzel a megoldással bár tovább bonyolódik a megjelenítő, teljesítményben azonban sokat jelent a két típus kirajzolásának megkülönböztetése.

4. Teszt eredmények

A továbbiakban különböző tesztelési esetekkel mutatjuk be a többszálas raszterizációs megközelítés teljesítményét. A tesztek során fontosnak tartottuk, hogy több különböző megoldással is összehasonlítsuk az eredményeket. Így minden különböző teszteset implementálásra került mind a klasszikus egy szálat alkalmazó raszterizációs megoldással is. Továbbá az eredmények viszonyítása miatt fontosnak tartottuk, hogy a teszteseteket GPU alapú implementációval is megvalósítsuk. Így jól látható a módszerek sebességének az egymáshoz viszonyított aránya. A GPU alapú referencia implementáció elkészítésére az OpenGL keretrendszert választottuk, ahol minden képi elemet a videokártyában tároltuk, megjelenítésre pedig a gyors VBO (Vertex Buffer Object) technikát alkalmaztuk [6].

Fontos kiemelni, hogy a textúrák kirajzolását GLSL segítségével végeztük, ahol két különböző tesztesetet is megkülönböztettünk. A nem optimalizált esetnek a sok textúra kirajzolásakor van jelentősége, miszerint minden kirajzolás előtt külön inicializáljuk az árnyaló objektumot, majd a rajzolás után lezárjuk azt. A nem optimalizált jelző itt a folyamatos árnyaló váltások költségét szimbolizálja. Az optimalizált megoldásban ideális megvalósításként a tömeges textúra renderelés előtt csak egyszer kerül inicializálásra a shader, a végén pedig lezárásra.

A programok elkészítéséhez a C++ nyelvet és a GCC 4.4.1 fordítót használtuk, a méréseket pedig egy Core i7-870-es 2.93GHz CPU-val végeztük el. A központi egység a Hyperthreading technológiának köszönhetően 8 szálat képes egyszerre futtatni. A teszt környezet egy 64 bites Windows 7 volt, az egyes implementációk 32 bitesek voltak. Egyik megvalósítás sem használt kézi SSE alapú részeket, csak a fordító által optimalizált kódot. A teszteléshez 800x600-as felbontást és ablakos módot alkalmaztunk. A felhasznált videó hardver pedig egy ATI Radeon HD 5670 1GB RAM videokártya. A szoftveres framebuffer megjelenítéséhez az OpenGL glDrawPixels megoldása került alkalmazásra optimalizált formában. A tesztekben felhasznált alfa csatornás képek átlagos mennyiségű átlátszó pixeleket tartalmaztak, körülbelül 50%-nyit. A tesztek eredményei 1 perc futási idő alatt mért átlagos képfrissítés (Frames Per Second) értékek alapján kerültek rögzítésre. Fontos kiemelni egyfajta referencia értékként, hogy a szoftveres megjelenítők esetében rajzolás nélkül 1714 FPS-t volt a renderelés sebessége, amely kizárólag az üres framebuffer-t küldte át a GPU-nak megjelenítésre.

Mindkét szoftveres renderer (klasszikus és elosztott) által használt framebuffer és a pixel műveletek optimalizáltak. A framebuffer-t uint32_t típusként definiált, mert így lehetővé válik, hogy 1 pixel 4 színkomponensét egyszerre kezeljük (értékadás, mozgatás, stb.) [7].

4.1 Megjelenítők összetett tesztje

A továbbiakban célunk különböző teszteseteken keresztül bemutatni és összehasonlítani az egyes megoldások teljesítmény értékeit. Minden teszt egy-egy speciális faladat csoportot képvisel. Megpróbáltuk azokat a legfontosabbakat kiemelni, amelyek a számítógépes játékokban gyakran előfordulnak. Segítségükkel megállapítható, hogy milyen eredményeket képesek elérni a megjelenítő különböző helyzetekben.

Teszteset 1: a teszt során arra a kérdésre kerestük a választ, hogy egy nagyobb méretű, átlátszó területeket nem tartalmazó kép megjelenítésekor milyen teljesítményt képesek elérni az különböző megoldások.

Teszteset 2: a második teszt célja a nagyméretű átlátszó részeket tartalmazó képek megjelenítésének mérése. Az alkalmazott kép pixeleinek körülbelül a fele volt átlátszó.

Teszteset 3: a teszt egy átmenetet képez az első tesztek és a soron követő tesztek között. Viszonylag nagyobb nem átlátszó textúrából jelenít meg 10 darabot.

Teszteset 4: a harmadik teszteset alkalmazása átlátszó textúrákkal.

Teszteset 5: a harmadik tesztben a nagy volumenű textúrával terhelt rendszert szimulálunk 200 darab 64x64 méretű nem átlátszó animált objektum renderelésével. Egy objektum 8 darab animációs fázist tartalmaz, méretük egységes. Kirajzolási helyük egyenletes eloszlás alapján véletlenszerűen generált.

Teszteset 6: a harmadik teszt átlátszó textúrákkal elvégzett változata.

A következő táblázat a különböző megoldások által elért eredményeket foglalja össze:

1. táblázat. *Különböző megközelítések sebességtesztje*

	Darab	**Raszterizáció sebessége (FPS)**			
		Egyszerű raszterizáló (1 szál)	**Elosztott raszterizáló (4 szál)**	**Normál GPU megvalósítás**	**Optimalizált GPU megvalósítás**
800x600 textúra (RGB)	1	1580	**1710**	3012	3012
800x600 textúra (RGBA)	1	717	**1180**	3050	3050
256x256 textúra (RGB)	10	1192	**1336**	2960	3056
256x256 textúra (RGBA)	10	522	**950**	2987	3052
64x64 animáció (RGB)	200	666	**1002**	532	1108
64x64 animáció (RGBA)	200	380	**766**	538	1126

Mint azt gondolni lehetett a pixel szintű megjelenítés minden esetben a leglassabbnak bizonyult. Míg a nem átlátszó textúrák renderelése esetén teljesítménye magasabb, úgy átlátszó esetben ez sokkal rosszabb. Ennek oka, hogy a megjelenítő a nem átlátszó képeket a memória másolás műveletével rajzolja ki, míg átlátszó esetben pixel alapokon. A tesztesetek eredményeinél elért sebességértékek alátámasztják azt a tényt, hogy 1 szálat alkalmazó megjelenítő nem képes kihasználni a rendelkezésre álló CPU erőforrásokat.

A prezentált elosztott renderelő motor azonban minden esetben jól szerepelt. Bár a prototípusként megvalósított 4 szál még mindig nem használja ki a hardver lehetőségei

teljes mértékben, az eredmények meggyőzőek. Egy esetben a sebessége megelőzte a nem optimalizált GPU alapú implementációt is. Továbbá érdemes megjegyezni, hogy a megközelítés hatékonyabban kezeli a nagy volumenű grafikus terhelést. Míg a második esetben az optimalizált GPU megoldás és az elosztott megközelítés sebességeinek aránya *2,58*, addig az utolsó tesztnél ez már csak *2,09* volt.

Az optimalizált GPU alapú megvalósítás minden esetben a leggyorsabb volt, de nem szabad elfelejtenünk, hogy a számításokat ilyenkor a dedikált hardver végzi el, valamint mivel minden adat a videó-memóriában van eltárolva, nincs szükség adatmozgatásra a központi memória és a GPU memória között. Az első két esetben pedig az optimalizált és a nem optimalizált GPU megoldások teljesítményeknek meg kell egyeznie, hiszen ilyenkor csak egy darab textúra kerül renderelésre.

Természetesen a valóságban további (kivételes) esetek is lehetnek. Például ha egy játékban az összes objektum valamilyen oknál fogva egy logikai területbe kerül, kirajzolásukat ilyenkor egyetlen egy szálnak kell elvégeznie. Ebben az esetben a sebesség meg fog egyezni az egy szálat alkalmazó megoldással. Erre a problémára szolgáltathat az az elképzelés, ha a logikai területek méretei kisebbre állítjuk és a szálaknak nincs kijelölt területük, hanem az adott területeket mindig az a szál dolgozza fel, aminek éppen nincs feladata. Valamint a fenti példák nem foglalkoznak azzal az esettel, amikor egy objektum képét skálázni vagy forgatni kell. Ezekben az esetekben a CPU-ra még inkább több feladat hárul szemben a GPU adta hardveres megvalósítással.

5. Összefoglalás

Napjainkban a számítógépes grafika területét a GPU technológiák uralják, azonban nem szabad megfeledkeznünk a szoftveres képszintézisről sem. A központi egységek nagy fejlődésen mentek keresztül, melyek új lehetőségeket kínálnak fel e területen. Bár teljesítményben egy erős GPU még nem győzhető le, de egy megfelelően elkészített modern elvekre és megoldásokra épülő szoftveres raszterizáló képes jó eredményt elérni a megjelenítésben. Nemcsak sebességében, hanem rugalmasságában is. Hiszen nem szabad elfelejteni, hogy egy szoftveres csővezeték a mai hardveres megvalósításokhoz képest kevésbé kötött. Továbbá a cikkben tárgyalt megoldás rávilágít arra, hogy van alapja a szoftveres megjelenítőt alkalmazó kétdimenziós játékok és egyéb grafikus alkalmazások fejlesztésének is. A processzorok további fejlődésével (pl. AVX utasításkészlet) pedig egyre inkább újra megnyílnak a lehetőségek ezen a területen. Természetesen ez egy komplexebb grafikus motort, jól optimalizált szoftvert, sok erőfeszítést igényel és alacsonyabb szintű nyelvek (pl. C, C++, D) alkalmazását, amelyek képesek kihasználni a központi egység lehetőségeit.

6. Köszönetnyilvánítás

A tanulmány/kutató munka a TÁMOP-4.2.2.B-10/1-2010-0008 jelű projekt részeként – az Új Magyarország Fejlesztési Terv keretében – az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

7. Felhasznált irodalom

[1] Mileff P., Dudra J.: *Efficient 2D Software Rendering*, Production Systems and Information Engineering, Volume 6, 2012. pp. 99-110.
[2] Zach, B.: *A Modern Approach to Software Rasterization*. University Workshop, Taylor University, 14. dec 2011.
[3] TransGaming Inc: *Swiftshader Software GPU Toolkit,* 2012.
[4] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: a many-core x86 architecture for visual computing. ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2008 Volume 27 Issue 3, August 2008.
[5] Laine, S., Karras T.: High-Performance Software Rasterization on GPUs. High Performance Graphics, Vancouver, Canada, aug 5. 2011.
[6] Akenine-Möller, T., Haines, E.: Real-Time Rendering, A. K. Peters. 3nd Edition, 2008.
[7] Agner, F.: Optimizing software in C++ An optimization guide for Windows, Linux and Mac platforms. Study at Copenhagen University College of Engineering, 2011.06.08.
[8] Coffin, C.: *SPU-based Deferred Shading for Battlefield 3 on Playstation 3*. Game Developer Conference Presentation, March 8, 2011.
[9] RAD Game Tools: *Pixomatic advanced software rasterizer,* 2012.
[10] Akhter, S., Roberts, J.: *Multi-Core Programming - Increasing Performance through Software Multi-threading*, Intel Corporation; 1st edition, 2006.