Multidiszciplináris tudományok, 12. kötet. (2022) 2 sz. pp. 3-13 **https://doi.org/10.35925/j.multi.2022.2.1**

RLE ALAPÚ SZOFTVERES RASZTERIZÁCIÓ

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

Absztrakt

A számítógépes grafika piacát napjainkban a GPU alapú technológiák uralják. A modern architektúrával rendelkező, egyre gyorsabb központi egységek azonban új lehetőségeket nyitnak meg a klasszikus szoftveres raszterizáció területén. A GPU technológiák fejlődése funkcionalitás szempontjából csaknem elérte határait, így a jövőben technológiai átalakulásuk a szoftveres megjelenítés irányába fog elmozdulni. Szoftveres esetben a rugalmasság még mindig nagyobb, mint GPU esetén. Jelen publikáció gyakorlati szempontból tekinti át a kétdimenziós megjelenítés problémáit és lehetőségeit. Hatékony megoldást javasol az átlátszó színkomponenssel rendelkező textúrák gyors raszterizációjára. Az eljárás alkalmazhatóságát teszteredményeken keresztül mutatja be összehasonlítva más technikákkal és a GPU megvalósítással.

Kulcsszavak: *valós idejű megjelenítés, szoftveres raszterizáció, optimalizáció*

Abstract

The market of computer graphics is dominated by GPU based technologies. However today's fast central processing units (CPU) based on modern architectural design offer new opportunities in the field of classical software rendering. Because the technological development of the GPU architecture has almost reached the limits in the field of the programming model, the CPU-based solutions will become more popular in the near future. This publication reviews the problems and opportunities of two-dimensional rendering from a practical point of view. An efficient, software based rasterization method is presented for textures having transparent color components. The applicability of the solution is proved through measurement results compared to other methods and the GPU based implementation.

Keywords: *real-time rendering, software rasterization, optimization*

1. Bevezetés

A számítógépes grafika sok éves fejlődésen ment keresztül az utóbbi néhány évtizedben, melynek egyik fontos állomása a grafikus célprocesszorok megjelenése volt. Az átalakulás legfőbb indítómotorja a grafikai számítások, a képi minőség növelésének megcélzása volt. Kezdetben a központi egység fejlődése korántsem zajlott olyan gyors ütemben, mint ma, így a piaci igényeknek megfelelően szükség volt egy olyan dedikált hardver megjelenésére, amely átvette a CPU-tól a raszterizációs feladatokat.

A grafikai számítások eltérő igényekkel rendelkeznek, mint a központi egységgel támasztott követelmények. Ez lehetővé tette, hogy a grafikus hardver így a központi egységtől függetlenül fejlődjön, és új lehetőségeket nyisson meg a fejlesztők, a mérnökök és a számítógépes játékok előtt. A fejlődés elsődleges irányvonala kezdetben a teljesítmény növelésére irányult, majd az igények egyre inkább a grafikus csővezeték programozhatóságának növelésére irányultak. Ennek eredményeképpen a mai GPU-k már elég hatékonyan programozható csővezetékkel rendelkeznek támogatva a magas szintű

árnyalónyelvek használatát. Ma a fejlődés pedig már egészen kiforrt irányba halad tovább, bevezetve a grafikus processzorok egy teljesen új generációját, az általános célú grafikus processzorokat, melyek már nem csupán a megjelenítés gyorsítására alkalmasak, hanem a CPU-hoz hasonlóan a funkcióit tekintve az általános számítások irányába tendál.

Mindezek ellenére ki kell emelni a GPU alapú számítógépes raszterizáció problémáit. A programozási logikája és modellje lassan eléri határait, a fejlődés intenzitása láthatóan csökken. Hiába áll rendelkezésre gyors hardveres támogatású pipeline, az évek során a hardver korlátaihoz igazított csővezeték az árnyalónyelvek alkalmazásainak megkötései nem adnak olyan szintű rugalmasságot a programozónak a megjelenítés kezelésében, mint korábban azt a szoftveres, CPU alapú raszterizáció nyújtotta. Bár a jelenlegi pipeline és az 3D API-k sok funkciót biztosítanak a fejlesztők számára, amennyiben a megszokott működéstől eltérő logikát szeretnénk megvalósítani, sok nehézségbe ütközünk. A mai GPU architektúra megkérdőjelezhető és átalakításra szorul, melyet a vezető ipari felhasználók már erősen szorgalmaznak (Swenney, 2009, Coffin, 2011). A videóvezérlők ipara jelenleg is fejlődés alatt áll. Például az AMD vállalat új APU (Accelerated Processing Units) chipjei már egy teljesen új generációt képviselnek, ahol a grafikus processzor és a központi egység már egy tokban helyezkednek el.

A játékipar vezetői által is felvetett (Swenney, 2009) problémára a megoldás a visszatérés a szoftveres renderelés technikájához, ahol logikailag és technikailag is a megjelenítendő tartalmat a szoftverrel azonos programozási nyelven kell programozni. Ezzel lehetővé válna egy sokkal rugalmasabb fejlesztési környezet létrehozása, amely új irányba lendíthetné a számítógépes grafikát.

Mindehez jó alapot kínál, hogy az elmúlt években a számítógépek központi egysége hatalmas fejlődésen ment keresztül. 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 és SSE utasításcsalád. Az új technológiáknak köszönhetően egy az utasításkészletet megfelelően kihasználó szoftver akár 2-5x sebességnövekedést is elérhet.

Jelen cikkben a kétdimenziós szoftveres raszterizáció gyakorlati megvalósítási kérdéseit vizsgáljuk meg, és olyan speciális optimalizációs megoldásokat mutatunk be, amelyekkel jelentősen felgyorsítható a GPU nélküli raszterizáció sebessége.

2. Irodalmi áttekintés

A szoftveres képszintézis az első számítógépek óta jelen van, és főként a személyi számítógépek megjelenésével kerül még inkább előtérbe körülbelül 2003-ig. Ezután már csaknem minden vizualizáció GPU alapú lett. A korai évek során született raszterizálók közül legkiemelkedőbb eredmény az ID software által készített Quake I, II szoftveres renderer (1996), amely az első valós háromdimenziós motor volt (Eberly, 2007). Grafikus megjelenítő alrendszereit Michael Abrash koordinációjával készült jellegzetesen a Pentium processzorcsaládra optimalizálva kihasználva a nagyszerű MMX utasításkészlet. A később született eredmények közül főleg az Unreal motort (1998) lehet kiemelni, amely nagyon gazdag funkcionalitással rendelkezett (Swenney, 2009).

A GPU renderelés folyamatos térnyerése után a szoftveres megjelenítés egyre inkább háttérbe szorult. Ennek ellenére születtek néhány nagyszerű eredmény, mint a Rad Game Tools által fejlesztett Pixomatic 1, 2, 3 és a TransGaming (ma már Google) által készített Swiftshader (TransGaming/Google, 2022). Mindkét termék nagyon komplex, magas szinten optimalizált kihasználva a többmagos CPU-k modern szálkezelési lehetőségeit és önmagát dinamikusan módosító pixel pipeline-al rendelkezik,

mellyel futás közben lehet kódot generálni a teljesítmény maximalizálásához. Mindezek mellett a Pixomatic 3 és a Swiftshader raszterizáló 100%-ig DirectX 9 kompatibilis. Sajnos mivel ezek a termékek mind szabadalmazva vannak, az architekturális felépítésüket nem tárják a nyilvánosság elé.

Bár a Microsoft a DirectX fejlesztésével nyújtott alapot a GPU technológiák terjedésének, mindezek mellett kifejlesztette saját szoftveres megjelenítőjét a WARP-ot (Microsoft, 2022). A renderer jól skálázható több szálra egyaránt és teljesítményben felveszi a versenyt az alacsony kategóriás (low-end) integrált grafikus kártyákkal.

A problémákat és az igényeket jól felmérve az Intel 2008-ban a Larrabee project keretein belül saját szoftveres megoldású videókártyák fejlesztését tűzte ki célul (Seiler et al, 2008). A kártya technológiai értelemben egy hibrid a többmagos CPU-k és a GPU-k között. Célja egy teljesen programozható pipeline kidolgozása volt, melyhez több x86 alapú magokat használt fel 16 byte szélességű SIMD vektor egységekkel (16-wide SIMD vector units) kiegészítve. Mindezek lehetővé tették, hogy a grafikai számításokat teljes egészében x86-os utasításkészlettel rugalmasabban programozzuk (Michael Abrash, 2009).

Napjainkban a szoftveres megjelenítés teljesen új irányára van lehetőség a GPGPU technológiának köszönhetően. (Loop et al, 2009) parametrikus felületek számára készített GPU alapú szoftveres megjelenítőt. (Liu et al, 2010) által elkészített Freepipe szoftver raszterizáló pedig a különböző fregment effektekkel próbált hatékony megoldást találni, ahol minden háromszög raszterizációját külön szál végez el. Az NVidia a szoftver renderer egy érdekes megvalósíthatósági tanulmányát (Laine, 2011) készítette a saját GPGPU CUDA platformjának segítségével. Az algoritmus a napjainkban népszerű tile-based renderinget alkalmazza a renderelési feladatok GPU-ra való szétosztására. Napjaink vezető grafikai színvonalú számítógépes játéka, a BATTLEFIELD 3 (Coffin, 2011) új technológiájaként egy SPU (Cell Synergistic Processor Unit) alapú tile-based deferred rendering-et valósított meg, amely segítségével nagyszámú fényforrás hatékonyan és optimalizáltan kezelhető.

Az (Zach, 2011) publikációban a szerző egy több szálat használó modern tile-based szoftver rendering technikát vázol. Az elmúlt évek eredményei tehát egyértelműen alátámasztják az a tényt, hogy a teljesítmény és a rugalmasság növelése érdekében ismét előtérbe kerülnek a CPU alapú megközelítések.

3. Szoftveres raszterizáció a gyakorlatban

Szofveres raszterizációnak nevezzük azt a képalkotási folyamatot, amikor a képszintézis teljes folyamatát nem egy célhardver végzi, hanem a számítógép központi egysége (CPU). Az alakzatokat felépítő geometriai primitívek a központi memóriában helyezkednek el tömbök, struktúrák és egyéb adatok formájában. A képalkotás logikája nagyon egyszerű: a központi egység elvégzi a szükséges műveleteket (színezés, textúra leképzés, színcsatornák állítása, forgatás, nyújtás, eltolás, stb.) a központi memóriában tárolt adatokon, ennek eredményét eltárolja egy framebufferben, majd az elkészített képet kiküldi a videóvezérlőnek.

A szoftveres képszintézis számos előnnyel rendelkezik a GPU alapú technológiával szemben. A feldolgozási folyamatot a CPU hajtja végre, így kevésbé kell aggódni a kompatibilitási problémák miatt. A képalkotás a szoftverrel azonos nyelven programozható egységesen, így nincs semmilyen megkötés az adatokra (Pl.: textúra maximális mérete) és folyamatra szemben a GPU nyelvek shader megoldásaival. A teljes grafikus csővezeték minden része egyedileg programozhatóvá válik. A szoftver több platformra való elkészítése pedig kevesebb problémát okoz, hiszen a megjelenítés mindig az operációs rendszer vezérlőjén keresztül történik, nincs szükség speciális videókártya meghajtóra.

Egy gyors szoftver renderer elkészítése alacsonyabb szintű programozási nyelvek használatát (pl. C, C++, D) és mélyebb programozási ismereteket igényel. Az alkalmazott technikák miatt operációs rendszer specifikus ismeretekre és kódokra van szükség. Tipikus példája ennek az elkészített képet tartalmazó framebuffer videómemóriába való másolása, megjelenítése. A gyakorlatban erre több megoldás alakult ki. Elsőként használhatjuk az operációs rendszer rutinjait a framebuffer célba juttatására, ez azonban erősen platformfüggő. A szoftver alsó rétegét ekkor minden operációs rendszeren külön meg kell írni. Ennél elegánsabb megoldás, ha használjuk az OpenGL (pl. glDrawPixels, texture) platformfüggetlen vagy a DirectX (pl. DirectDraw surface, texture) nyújtotta lehetőségeket.

4. 2D alapú megoldások

A kétdimenziós megjelenítés fontos szerepet tölt be a mai modern háromdimenziós leképezés mellett. Azt lehet mondani, hogy bár a világ a megjelenítés terén 3D irányába mozdult el az utóbbi évek során, a kétdimenziós megoldások kiegészítő technikaként mindig jelen voltak és lesznek is a későbbiekben. A számítógépes alkalmazások több rétege tartozik ide. Bármilyen olyan szoftver, amely rendelkezik valamilyen grafikus menü- vagy ablakozó rendszerrel, továbbá a kétdimenziós számítógépes játékok nagy képviselői a területnek. A menürendszer szempontjából mindig vannak próbálkozások arra, hogy ezeket a részrendszereket is a háromdimenziós térben próbálják szemléletesebbé tenni, bár működnek, de többnyire ezek a megoldások visszatérnek a 2D leképzéshez. Ezeknél a rendszereknél a sebesség a mai rendszerek teljesítményéhez viszonyítva nem kritikus. Leginkább statikus képek váltakoznak, melyek száma kevés, egyéb transzformációk (forgatás, nyújtás, döntés) azonban nem nagyon jellemzőek.

A számítógépes játékoknál azonban pont az ellenkezője tapasztalható. Ilyen szoftverek esetén általában nagy mennyiségű folyamatosan változó objektumok halmazát kell raszterizálni, amely jelentős erőforrást igényel dinamikusan változva a mozgatott elemek függvényében. A mai rendszerek tipikus jellemzője a nagy textúrahalmaz, animációk és transzformációk dinamikus alkalmazása a magasabb felhasználói élmény elérése érdekében. Az igényeknek megfelelően a képernyőfelbontás nagy, minőségi textúrák használata már elengedhetetlen, amely tovább növeli a követelményeket a teljesítménnyel szemben.

A továbbiakban olyan 2D raszterizációs technikákat mutatunk be, amelyek lehetővé teszik egy nagy teljesítményű, robosztus szoftveres vizualizációs rendszer létrehozását.

4.1. Klasszikus 2D raszterizáció

A klasszikus kétdimenziós, szoftver megvalósítású raszterizációnak számos nehézséggel kell megküzdenie. A legfontosabb hátrány a GPU alapú megoldásokkal szemben a dedikált célhardver hiánya, így minden műveletet a CPU-nak kell elvégeznie.

A 2D megjelenítés kétdimenziós képi elemekkel (textúrákkal), objektumokkal dolgozik. A grafikus motor feladata a kép előállítása során, hogy sorra vegye az objektumokat és kirajzolja a képernyőre. A végleges kép tehát ezek kombinációjaként jön létre. A textúrák minden esetben a központi memóriában vannak tárolva egy-egy tömbként reprezentálva. A tömbök színinformációkat tartalmaznak az objektumokról, méretük a textúra minőségétől függ. A mai szoftverek 32 bites (4 byte - RGBA) színmélységű képekkel dolgoznak felbontásuk pedig a megjelenítendő elemek igényeitől függően akár 1024x768 pixel is lehet. A textúrákat színcsatornák alapján két csoportba sorolhatjuk: vannak alfa csatornával rendelkező és nem rendelkező képi elemek. A megkülönböztetés azért fontos, mert a megjelenítési eljárás, a gyorsítási lehetőségek a két csoportnál eltérnek.

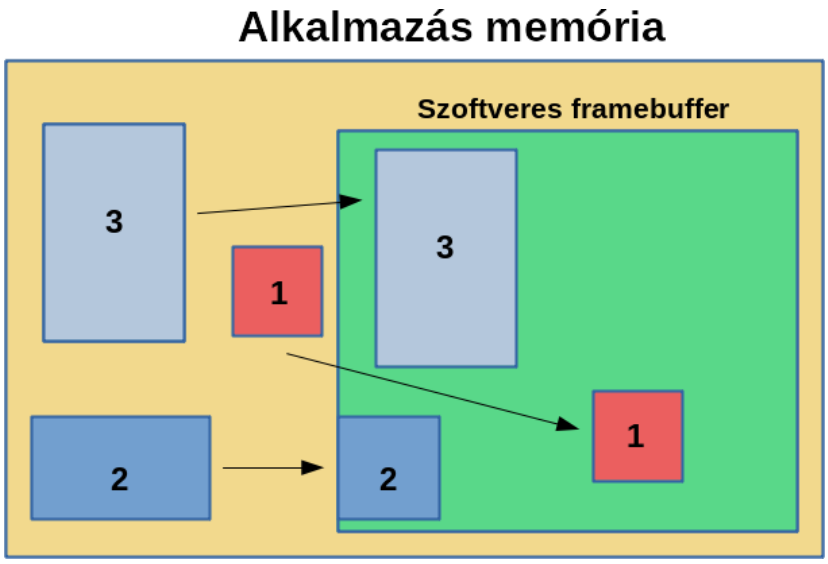
4.1.1. Renderelés alfa csatorna nélkül

Az alfa csatorna nélküli textúrák nem rendelkeznek átlátszósággal, csak RGB színkomponensekkel. Ez azt jelenti, hogy bármely két objektum egymásra rajzolható anélkül, hogy az egymás alatt lévő objektumok színét össze kellene mosni egymással, kirajzolási mechanizmusuk így gyorsabb és egyszerűbb lesz.

A klasszikus rajzolási mechanizmus a textúrák bármely típusánál a pixelenkénti raszterizáció, amely azonban nagyon lassú a mai igényeknek megfelelő mennyiségű és minőségi képi elemek megjelenítése során. A grafikus motor az objektumokat reprezentáló képi elemeken pixelenként halad végig és készíti el a képet. Ennek hátránya az, hogy rengeteg elemet kell kirajzolni a képernyőre, melyek szintén sok pontból állhatnak. A pixelenkénti rajzolás több ezer függvényhívással és redundáns számítással jár. Minden pixel esetén külön ki kell olvasni a memóriából a képpont színét, majd a környezeti adatok függvényében pedig meghatározni a képernyőn való pozícióját, és elvégezni a szín framebufferbe való írását (pl. *pFrameBuffer[y * screenWidth + x] = color*).

A pixelenkénti megvalósítás tehát nem ad elég gyors megoldás. Túl sok apró műveletet kell elvégezni, amely felemészti a CPU erőforrásait. Egy valós idejű számítógépes játék esetén akár 100 különböző objektum is lehet egyszerre a képernyőn egymást átfedve. Kirajzolásuk sok CPU erőforrást igényel.

Megfelelő raszterizációs sebesség elérése érdekében további megoldásokra van szükség. Alfa csatornával nem rendelkező képek esetében ez viszonylag könnyen megoldható úgy, hogy a képet nem pixelenként, hanem egy, vagy több blokkban egyszerre mozgatjuk át a framebufferbe. A raszterizáció optimalizálásának legfontosabb célját tehát úgy fogalmazhatnánk meg, hogy meg kell próbálni minden műveletet a lehető legnagyobb blokkra kiterjedő módon elvégezni, ezzel elkerülve a felesleges adatmozgatásokat és számításokat. A következő ábra ennek folyamatát mutatja be:



1. ábra. *Blokk orientált textúra másolás*

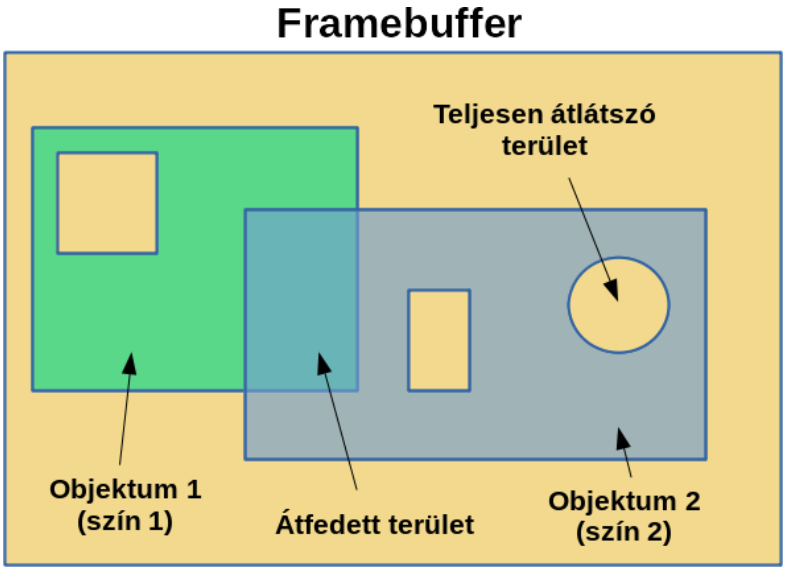
A kirajzolás ebben az esetben azt jelenti, hogy a központi memória blokkjait a framebuffer meghatározott területére mozgatjuk a memóriamásolás (pl. C++ - *memcpy()*) műveletével, melyekkel nagyságrendi sebességnövekedés érhető el. A módszer azonban nem teljes értékű így, mert míg a pixel

szintű raszterizáció kapcsán a framebuffer pozíció kiszámítása során közvetlenül elvégezhető egy framebuffer tartomány határ ellenőrzés (bound checking), addig a blokkorientált adatmozgatás során szegmentálni kell a másolandó blokkot a képernyőn látható tartalom függvényében, ha valamilyen irányba kilógna az objektum. Bár ez további számításokat igényel, a teszt eredményekből látni fogjuk, hogy a nagyságrendi teljesítménynövekedés így is megmarad.

4.1.2. Renderelés alfa csatornával

A textúrák másik csoportjába azok a képek tartoznak, amelyek rendelkeznek a negyedik, alfa (A) színkomponenssel is. Az alfa csatornás képek szerepe napjainkban megnőtt, számtalan helyen alkalmazzák a vizualizációs élmény növelése céljából (pl. árnyékolt ablakok, animációk elmosott szélekkel, félig átlátszó komponensek, stb.). Kezelésük, bár nem nehezebb, de sokkal számításigényesebb. Ennek oka az, hogy egy textúrán belül tetszőlegesen váltakozhatnak az átlátszó és a nem átlátszó részek (pl. karakter animáció, ablakok, részecske rendszer, stb.), amelyek miatt a kirajzolás ilyenkor pixelenként történik. Az átlátszó vagy félig átlátszó részeknél az objektum alatti színnel össze kell mosni az adott pixel színét.

Mint azt korábban említettük, ez bár alapjában véve nem számításigényes eljárás, a sok objektummal és nagyobb méretű textúrákkal dolgozó szoftverek esetén a megoldás sebessége nagyon lassú lesz. Az alábbi ábra bemutatja a problémát:



2. ábra. *Átfedésben lévő RGBA textúrák*

4.2 RLE alapú raszterizáció

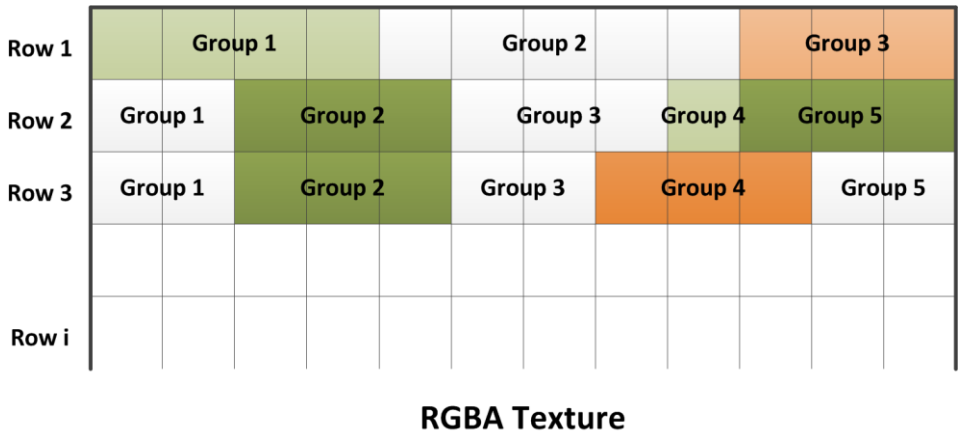
A fentiekből következik, hogy a kétdimenziós szoftveres raszterizáció (egyik) legnagyobb problémája az alfa csatornás, bizonyos helyeken átlátszó textúrák hatékony kezelése. A következőkben olyan technikát mutatunk be, amely kompromisszumok árán hatékony megoldást kínál a problémára.

Az eddigiek alapján belátható, hogy olyan eljárásra kidolgozására van szükség, amely valahogyan megpróbálja kihasználni a blokk alapú megjelenítés lehetőségeit, egységként kezelve a pixeleket. A probléma megoldásához induljunk ki egy átlagos textúra vizsgálatából. Ha elemezzük a pixeleket, akkor

rögtön látszik, hogy a képek többsége rendelkezik olyan részekkel, ahol az egymás mellett lévő pixelek színei megegyeznek egymással. Ez jó alapot kínál arra, hogy olyan algoritmust és adatstruktúrát alkalmazzunk, amely az *RLE (Run-length encoding)* kódoláshoz hasonlóan működve blokkosítja az azonos színű egymás mellett elhelyezkedő pixeleket.

4.2.1 Textúrák előkészítése

A megoldás alapgondolata tehát az azonos színű pixelek blokkokba gyűjtése. Ehhez bármely kép esetén egy előfeldolgozási műveletre van szükség, amely során kialakítjuk azt a megfelelő leíró adastruktúrát, amely majd a raszterizációs fázisban a blokkos renderelés elvégzéséhez ad segítséget. A következő ábra az előfeldolgozási folyamat lépéseit és logikáját reprezentálja.



3. ábra. *Textúrák előkészítése a blokk orientált megjelenítésre*

Az ábra jól mutatja, hogy a textúra színcsoportjainak blokkorientált tárolásához az eddiginél összetettebb struktúrára van szükség. A feldolgozás soronként történik, ahol külön csoportot kell létrehozni minden összefüggő pixelhalmaznak. Ehhez tárolni kell a pixelcsoport színét, a színcsoport hosszát, hogy átlátszó-e vagy sem, és végül egy mutatót, amely a csoport első pixelének memóriabeli címére mutat az eredeti adatstruktúrában. Valamint szükség van egy globális adatstruktúrára, amely tárolja a képhez tartozó egyes színcsoportokat és az eredeti pixeltömbre mutató pointert.

A következő C++ mintakód egy létező textúra RLE alapú tárolási struktúrába való transzformálásért mutatja be.

```
vector<vector<CRLEColor>> slab;
for j to mTexture.height
    vector<CRLEColor> row;
    count = 0;
    for i to mTexture.width
       count++;
       color = mTexture.texels + (j * mTexture.width + i);

       if (i != 0)
          if (color != prev_color)
             CRLEColor rle_color;
```

```
            // color, length, offset, x, y
            rle_color.Init(prev_color,count,mTexture.texels + j * mTexture.width +
            (i-count)+1, i-count+1, j);

            if (prev_color == m_uiColorKey)
               rle_color.invisible = true;
            end if

            row.push_back(rle_color);
            count = 0;
          else
             if (i == m_pTexture.width-1)
               CRLEColor rle_color;
               rle_color.Init(prev_color,count,mTexture.texels + j * mTexture.width +
               i-count,i-count,j);

               if (prev_color == m_uiColorKey)
                  rle_color.invisible = true;
               end if
               row.push_back(rle_color);
               count = 0;
             end if
          end if
       end if
       prev_color = color;
  end for
  slab.push_back(row);
end for
```

A tárolás megfelelő implementációs megvalósítása különösen fontos, mert sok objektumot tartalmazó rendszerek esetén renderelés során a ciklusok és műveletek száma jelentős. Apró, nem kellően hatékony megvalósítás jelentős sebességcsökkenést eredményezhet.

4.2.2 Renderelési folyamat

A kép renderelési folyamatában az előkészített adatstruktúra alapján történik az objektumok leképzése. A struktúra lehetővé teszi, hogy bár a kép "lyukas", mégis megvalósítható egyfajta blokkorientált színcsoportonkénti blittelés a framebufferbe, amellyel az alfa csatornás képek renderelési sebessége jelentősen növelhető. A megjelenítés annyi blokk mozgatásából fog állni, amennyi színcsoport az előfeldolgozás során született. Továbbá a raszterizációt képsoronként végezzük el. Ennek egyik oka, hogy maga a framebuffer sorfolytonos formában került megvalósításra, mint az a legtöbb rendszerben, így egy sor egy logikai egységet képez. Másik ok pedig az, hogy az egyes objektumok kilóghatnak a képernyőből. Bár a színek csoportosítva vannak, a nem megjelenítendő részeket le kell vágni a raszterizáció során. Ez további számítási feladatok elvégzését igényli. A következő mintakód a kirajzolási folyamatot mutatja be.

```
FrameBuffer framebuffer = gGraphicsEngine.GetFrameBuffer();

for i to row_group.size
   vector<CRLEColor> image_row = row_group[i];
   for j to image_row.size
      CRLEColor color = image_row[j];
      if (color.invisible == false)
        realposX = pos.x + color.x
        realposY = pos.y + color.y
        framebuffer.Blit(color.offset,color.real_length, realposX, realposY);
    end if
  end for
 end for
```

5. Teszt eredmények

A különböző raszterizációs technikák sebességbeli különbségeit három tesztfeladat segítségével mutatjuk be. 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 használt felbontás 800x600 volt ablakos módban. Az eredmények validitása miatt fontosnak tartottuk, hogy a minden tesztesetben elvégzett számítást GPU alapú implementációval is megvalósítsuk. Így jól látható a módszerek sebességének az egymáshoz viszonyított aránya. A tesztekhez ATI Radeon HD 5670 1GB RAM videókártyát használtuk. A szoftveres framebuffer megjelenítéséhez az OpenGL *glDrawPixels* megoldása került alkalmazásra optimalizált formában.

A GPU alapú referencia implementáció elkészítésére szintén az OpenGL rendszert választottuk, ahol minden képi elemet a videókártyában tároltuk, megjelenítésre pedig a VBO (Vertex Buffer Object) technikát alkalmaztuk. Jelenleg a VBO a leggyorsabb textúra rajzolási megoldás a videókártyáknál.

A teszt során felhasznált alfa csatornás képek átlagos mennyiségű átlátszó pixeleket tartalmaztak.

Teszteset 1: a teszt során arra a kérdésre kerestük a választ, hogy egy nagyobb méretű kép kezeléséhez hogyan viszonyultak az ismertetett megoldások. Bár ebben a példában az RLE alapú megoldás logikailag nem illik a megoldások közé, mert az első esetben a kép nem tartalmaz átlátszó területeket, mégis célszerű elvégezni ezt a mérést is.

Teszteset 2: a második teszt célja, hogy egy alfa csatornával és átlagos mérettel rendelkező textúra esetében vizsgálja meg az elkészített RLE alapú implementáció sebességét a klasszikus megoldással szemben. A blokk orientált megközelítés ilyen kép típusoknál nem alkalmazható.

Teszteset 3: a harmadik tesztben a nagy volumenű textúrával terhelt rendszert szimulálunk 200 darab 64x64 méretű alfás textúra renderelésével.

A tesztek során a legalább 1 perc futási idő alatt mért átlagos *Frame Rate* (Frames Per Second) érték került rögzítésre. Az alábbi táblázat tartalmazza mindhárom feladat eredményeit:

1. táblázat. *Mérési eredmények*

	Darabszám	**Raszterizálás sebessége (FPS)**			
		Pixelszintű	**Blokk szintű**	**RLE alapú**	**GPU megvalósítás**
800x600 textúra	1	119	1290	1224	3522
256x256 textúra (alfás)	1	910	-	1798	3689
64x64 textúra (alfa nélküli)	200	143	-	794	1690

Az eredmények jól mutatják, hogy a pixel szintű megjelenítés minden esetben a leglassabbnak bizonyult a sok apró művelet miatt, azonban az RLE alapú megközelítés a minden tesztesetben jó eredménnyel szerepelt. Ez jól alátámasztja azt a tényt, hogy a pixelek blokkokban való mozgatása jelentős sebesség növekedést eredményez. A GPU alapú megvalósítás mindhárom esetben a leggyorsabb volt, de nem szabad elfelejtenünk, hogy a számításokat ilyenkor a dedikált hardver végzi el, valamint mivel minden adat a videómemóriában van eltárolva, nincs szükség adatmozgatásra a központi memória és a GPU memória között.

6. Összefoglalás

Bár napjainkban a GPU technológiák piaca uralja a számítógépes grafika területét, nem szabad megfeledkeznünk a szoftveres képszintézis kínálta lehetőségekről sem. A központi egységek nagy fejlődésen mentek keresztül, melyek új lehetőségeket kínálnak fel e területen. A cikkben tárgyalt RLE alapú megoldás rávilágít arra, hogy egy gyors szoftveres renderer készítése sok erőfeszítést igényel. Nélkülözhetetlen a több technológia együttes alkalmazása, valamint alacsonyabb szintű nyelvek (pl. C, C++, D) használata. A bemutatott modell már önmagában véve is jelentős gyorsulást eredményez bizonyos esetekben, de további kiegészítő megoldás lehet akár a többmagos processzorok logikai egységeinek hatékony kihasználása, amely sokszorosára emelheti a megjelenítés sebességét lehetővé téve ezzel akár a GPU nélküli számítógépes játékfejlesztést.

Irodalom

[1] Zach, B. (2011). *A modern approach to software rasterization*. University Workshop, Taylor University.
[2] Google Inc. (2022). *Swiftshader Software GPU Toolkit*. https://github.com/google/swiftshader
[3] Microsoft Corporation (2022). *Windows advanced rasterization platform (warp) guide*
[4] Abrash, M. (2009). *Rasterization on larrabee*. Dr. Dobbs Portal
[5] 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. (2008). *Larrabee: a many-core x86 architecture for visual computing.* ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH, 27(3). **https://doi.org/10.1145/1360612.1360617**
[6] Rost, R. (2004). *The OpenGL Shading Language*. Addison Wesley.

[7] Laine, S., Karras, T. (2011). *High-Performance software rasterization on GPUs.* High Performance Graphics, Vancouver, Canada. **https://doi.org/10.1145/2018323.2018337**
[8] Akenine-möller, T., Haines, E. (2008). *Real-time rendering*. A. K. Peters. 3nd Edition.
[9] Sugerman, J., Fatahalian, K., Boulos, S., Akeley, K., Hanrahan, P. (2009). *Gramps: A programming model for graphics pipelines*. ACM Trans. Graph. 28, 4:1–4:11. **https://doi.org/10.1145/1477926.1477930**
[10] Fang, L., Mengcheng, H., Xuehui, L., Enhua, W.: FreePipe (2010). *A programmable, parallel rendering architecture for efficient multi-fragment effects*. In Proceeding of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games.
[11] Agner, F. (2021.08.11). *Optimizing software in C++ An optimization guide for Windows, Linux and Mac platforms*. Study at Copenhagen University College of Engineering.
[12] Swenney, T. (2009). *The end of the GPU roadmap*. Proceedings of the Conference on High Performance Graphics, pp. 45-52.
[13] Coffin, C. (2011). *SPU-based deferred shading for battlefield 3 on Playstation 3*. Game Developer Conference Presentation, March 8.
[14] RAD Game Tools (2021). *Pixomatic advanced software rasterizer*.
[15] Eberly, H. D. (2006). *3D game engine design: A practical approach to real-time computer graphics.* CRC Press; 2nd edition.
[16] Mileff, P., Dudra, J. (2022). The past and the future of computer visualization. *Production Systems and Information Engineering*, 10(1), 16-29.