



HATÉKONY PIXELALAPÚ RAJZOLÁS A GYAKORLATBAN

MILEFF PÉTER

Miskolci Egyetem

Általános Informatikai Intézeti Tanszék

mileff@iit.uni-miskolc.hu

Absztrakt: A grafikus feldolgozóegység (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 a rasterizáció folyamatában. A fejlődés eredményeképpen a háromszögalapú renderelés vált gyakorlatilag kizárólagossá a GPU-k miatt, a pixelalapú képmanipulációt leggyakrabban shaderok segítségével végzik el. A mai GPU-alapú csővezeték azonban nem tudja azt a rugalmasságot nyújtani, mint a korábbi szoftveres megvalósítás, amikor sokkal több lehetőség volt a pixelek manipulációjára. Jelen cikk a pixelalapú rasterizáció hatékony szoftveres megvalósításával foglalkozik. A jelenlegi GPU-alapú rajzolás folyamatának áttekintése után megmutatjuk, hogy ebben a környezetben miként érhető el mégis a pixelalapú rajzolás. Végül a klasszikus pixeltárolási megoldástól hatékonyabb tárolási és megjelenítési forma kerül bemutatásra, amely teljesítménye messze felülmúlja a korábbi megoldást.

Kulcsszavak: szoftveres rasterizáció, optimalizáció, pixelrajzolás

1. Bevezetés

A magas minőségű számítógépes vizualizáció napjaink egyik fontos területe és követelménye. A modern képszintézis gyakorlatilag ma már szinte minden területen jelen van, legyen az számítógépes játék, multimédiás alkalmazás, tervezőprogram vagy egyéb grafikus szoftver. Ma a vizualizáció területét a GPU-alapú megjelenítés uralja. Maga a hardver nagymértékű fejlődésen van túl, az ipar közel 20 év fejlesztést tudhat maga mögött. Legfőbb mozgatórugója a számítógépes játékipar volt, ahol állandó törekvés a jobb, a valósághoz még inkább közelebb álló megjelenítési minőség. A GPU-k használata gyakorlatilag standarddá vált, minden mainstream operációs rendszer rendelkezik zárt vagy akár nyílt forráskódú megfelelő meghajtó-programmal annak ellenére, hogy a videokártyák felépítése nem nyitott a meghajtó programot írni kívánók számára.

A GPU azonban nem csodaeszköz. Áruk és áramfogyasztásuk az utóbbi években jelentősen megnövekedett. Egy modern videokártya fogyasztása aktív használat közben elérheti akár a 280 W-ot is. Megjelenésük jelentősen átalakította a vizualizáció addig ismert folyamatát. A fejlesztőknek szakítani kellett az addigi szoftveres megjelenítés módszereivel és alkalmazkodni kellett a videokártya-meghajtók programozható API-jához (OpenGL, DirectX). A vizualizáció eme meghatározott folyamata az évek során már nem nagyon változott. Kezdetben úgynevezett fixfunkciós csővezeték használhattunk a képszintézisben, később az árnyalók megjelenésével a csővezeték különböző részei már a fejlesztők által is programozhatóvá vált ezzel utat nyitva a programokban használt egyedi megoldások felé. Nagyon leegyszerűsítve talán azt is mondhatjuk, hogy a vizualizáció GPU előtti időszakában a programozónak teljhatalma volt, gyakorlatilag minden pixelt képes volt kezelni a képernyőn és a memóriában. A GPU-k megjelenésével ez a nagymértékű rugalmasság eltűnt, főleg a shadervilág előtt.

A mai vizualizáció drótvázmodellre épül. Minden objektum mögött ott kell legyen egy háromszögekből álló háló, amely az objektum „alakját” definiálja. Az úgynevezett textúra pedig erre a hálózatra kerül rá a Texture mapping technikával. Ahhoz, hogy a vizualizáció gyors legyen, a videokártya úgy lett kifejlesztve, hogy saját memóriával rendelkezzen. Tartalmaznia kell az összes olyan elemet, objektumot, amely rajzolásra kerül. Ezért minden szoftver egyik első lépése (pályák betöltése) az, hogy a rajzolás előtt minden elemet feltölt a GPU RAM-ba. A GPU tehát olyan, mint egy külön sziget. A jelenlegi GPU-alapú raszterizáció jól működik, hatalmas ipar épül rá. Előnyként jelenik meg, hogy rögzített programozási logikának megfelelően a programozók egy egységes magasabb absztrakciós szintet kapnak a kezükbe, nem szükséges a vizualizáció alacsony szintű programozása. A shaderek megjelenésével lehetővé vált a fixfunkciós csővezeték leváltása, egy fejlesztői oldalról való jobb programozhatóság bevezetése.

Azonban a programozó rá van kényszerítve azokra a megoldásokra, amelyet a GPU API-k (OpenGL, DirectX) nyújtanak. Bár az árnyalók nagymértékben javítják a GPU-k programozhatóságát, nem tudják elérni a szoftveres raszterizáció adta lehetőségeket és vélhetően soha nem is fogják, amely megadja azt a rugalmasságot, mint a korai időkben volt. Ma már nem tudunk pixelekké érdemben foglalkozni és a számítógépes vizualizációt tanulni kívánók számára jelentősen megnőtt a belépési pont [16].

Jelen cikk ezért azt a célt tűzte ki, hogy megvizsgálja azokat a lehetőségeket, amellyel igaz, hogy szoftveresen, de elérhető a pixelszintű programozhatóság. Megmutatjuk, hogy a CPU segítségével hogyan lehet hatékonyan rajzolni, mindezeket pedig mérési tesztekkel igazoljuk.

2. Pixelalapú szoftveres vizualizáció

A számítógépes vizualizáció korai szakaszában csak szoftveres, azaz CPU-alapú megjelenítést használt minden szoftver. Ezek a szoftverek maguk kezelték a pixeleket, a VESA VBE (VESA BIOS Extensions) szabványosított interfésznek megfelelően

pedig nem volt szükség speciális videokártya-meghajtóra vagy egyéb API-ra (OpenGL, DirectX). A VBE egy kiegészítése volt az INT 10h BIOS videószoftvereknek. A kívánt felbontás és a pixelek elérése (írás/olvasás) néhány soros programkódból megvalósítható volt még egy kezdő számára is.

A modern videokártyák megjelenésével megszűnt a pixelalapú rajzolás közvetlen támogatása. Így amennyiben ma valaki pixelel szeretne dolgozni, más utakat kell keresnie. Sajnos az interneten kevés információ áll rendelkezésre ennek hatékony megvalósítására. A gyakorlatban két lehetséges irány áll rendelkezésre, melyeket az alábbiakban ismertetjük.

2.1. Rajzolás hardveres API-n keresztül

A mai videokártyák képességeit a két elérhető API-n (OpenGL, DirectX) keresztül tudjuk csak megfelelően kihasználni. Amennyiben magas teljesítményt szeretnénk elérni, célszerű tehát ezekből kiindulni, követni azt a folyamatot, amik szükségesek és amit elvárnak az API-k. Jelen cikkben a továbbiakban kizárólag az OpenGL API-t választjuk arra, hogy a pixelalapú megjelenítés megvalósítását bemutassuk.

Meg kell említeni, hogy az OpenGL-ben már a kezdetektől rendelkezésre állt egy olyan lehetőség, amellyel ez megvalósítható volt. A *glDrawPixels* egy rendkívül könnyen használható opció volt arra, hogy egy négyszögletes pixelhalmazt, egy adott memóriaterületet a központi memóriából a videokártya framebufferjébe írjunk. Bár nagyon kényelmes megoldás volt, az OpenGL 3.2-től kikerült a szabványból és többé nem támogatott. Hordozható/mobileszközök esetén pedig sosem volt a szabvány (OpenGL ES) része [10]. Mindezekon felül a módszer teljesítménye nem feltétlenül volt kielégítő.

2.1.1. A pixelalapú rajzolás módszere

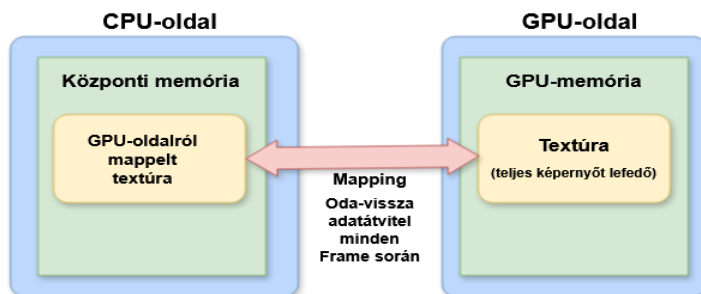
A GPU működésénél fogva arra van tervezve, hogy vertex- és textúraadatokkal hatékonyan tudjon dolgozni. Olyan megoldást érdemes tehát választani, amely valamilyen szinten alkalmazza ezeket a feldolgozási folyamatokat.

A pixelszintű rajzolás megvalósításához alapesetben nincs szükség semmilyen speciális API-ra az OpenGL, vagy DirectX-en kívül. A megoldás kulcsgondolata az, hogy ortogonális projekciót használva létre kell hozni egy, a képernyő felbontásával egyező textúrát és ennek megjelenítéséhez pedig egy háromszöghálót, amely elegendő, ha két háromszögből tevődik össze. A két háromszögből alkotott négyszöget úgy határozzuk meg és pozicionáljuk felbontástól függően, hogy teljes mértékben fedje a képernyőt. Ezzel gyakorlatilag egy virtuális vásznat készítettünk, amire rajzolni kellene valahogy.

A rajzolás menete azonban nem teljesen triviális. Ugyanis bármely grafikai objektum, amelyet meg szeretnénk jeleníteni, a grafikus API-k (OpenGL, DirectX) elvárják, hogy az objektum minden adata a GPU-memóriában legyen (kivéve nagyon nagy világok esetében [streaming]). Természetesen innen fakad az az igény, hogy

egyre több memóriával rendelkezzenek a GPU-k, hiszen a játék teljes aktuális megjeleníteni kívánt objektumai a GPU-ban kell elérhetők legyenek. OpenGL esetében VAO/VBO az az elvárt tárolási struktúra, amellyel hatékony rajzolási sebesség érhető el.

A probléma ott jelentkezik, hogy a GPU tárolja azt az adatstruktúrát, memóriaterületet, amelyet a CPU-oldalról kell módosítani. A textúraobjektum pixeleinek módosítására már több megoldás is jelentkezik, azonban mindegyik esetében számolni kell azzal, hogy a textúra adatokat, a memóriaterületet többször mozgatni kell a GPU-memória (GPU-oldal) és a központi memória (CPU-oldal) között. A következő ábra ezt a folyamatot mutatja be:



2. ábra. Szoftveres renderelés megvalósítása a mai GPU-kon

Első esetben akkor történik mozgatás, amikor rajzolni szeretnénk, másodszor pedig akkor, amikor végeztünk a pixelek módosításával, és az adatot ismét a video RAM-ba töltjük fel. Ráadásul mindezt másodpercenként legalább 50-60-szor, amely egy kritikus elvárás a játékok esetében. Hogy az átvitt adatmennyiséget érzékeltesük, egy 1920×1080 pixeles kép RGBA-minőségben 8 294 400 byte mozgatást jelent, amelyet 50-60-szor kell megtenni. Egy számítógépes játékban ritkán dolgozunk ekkora képekkel (esetleg a háttér), de kisebb méretű képekből azonban egyszerre általában több van képernyőn.

Tehát bármilyen megoldást is választunk azok közül, melyeket az OpenGL, vagy DirectX API-k nyújtanak, limitálva vagyunk az aktuális architektúra BUS-sebessége által. A mai modern gépek többségében PCI Express 4.0-t használunk, amely egy 2011-es szabvány, melynek elméleti maximális átviteli sebessége 16 GT/s bit, amely éppen duplázza a PCI Express 3.0-t. Bár már rendelkezésre állnak a PCI Express 5.0 (2017) és a PCI Express 6.0 (2022) szabványok is, az azokat támogató hardverek családja csak évek múlva fog megjelenni.

A GPU-memóriában elhelyezkedő pixeladatok CPU-oldalon való módosítására több megoldás is rendelkezésre áll az OpenGL-ben:

- **glTexImage2D/ glTexSubImage2D**: klasszikus megoldás, amikor a *glTexImage2D* függvénnyel létrehozunk egy textúrát, a *glTexSubImage2D*

függvénnyel pedig igény szerint módosítjuk bizonyos részeit, vagy akár az egészet.

- **Pixel Buffer Object (PBO):** A PBO legnagyobb előnye, hogy gyors pixeladatátvitelt tesz lehetővé a DMA (*Direct Memory Access*) segítségével a videokártya és a központi memória között úgy, hogy a CPU-nak nem kell részt vennie ebben a folyamatban. Másik előnye pedig az aszinkron DMA-átvitel.

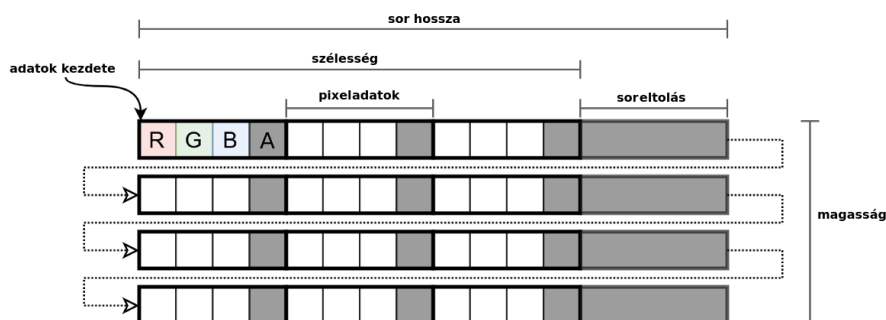
3. Szoftveres rajzolás megvalósítása

A szoftveres raszterizáció gyakorlati hatékonysága nagymértékben függ a megvalósítás módjától. Alapvető igény a magas teljesítmény elérése, a széles körű optimalizáció, amely sok esetben komoly hozzáértést kíván meg. A következőkben néhány fontos szabályt és technikát tekintünk át.

3.1. Pixelalapú rajzolás a modern GPU-n

A mai szoftverek 32 bites (4 byte – RGBA) színmélységű képekkel dolgoznak. 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. Az alfa csatornát nem tartalmazó képek 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 [18].

Az alfa csatornát tartalmazó képek kezelése bár nem nehezebb, de sokkal számításigényesebb. Ennek oka az, hogy egy textúrán belül tetszőlegesen váltakozhatnak az átlátszó és a nem átlátszó részek (pl.: karakteranimáció, felhő, részecskeeffektek stb.), amelyek miatt a kirajzolás ilyenkor pixelenként történik [18]. Amikor szoftveres raszterizációról beszélünk, akkor a képi objektum pixeleit általában az alábbi formában tároljuk a központi memóriában:



3. ábra. Pixelek tárolási formája a központi memóriában

A pixelek tehát négy komponensből (R, G, B, A) tevődnek össze, melyek mindegyike 1 byte. Ezek az értékek akkor érdekesek, amikor a videó RAM-ba kell mozgatni a pixeltömböt. CPU-oldalon, amikor számításokat végzünk, a pixeleket szokás floatként is tárolni $[0,1]$ intervallumban.

A következőkben C++ nyelvű mintakódok segítségével bemutatjuk, miként lehet a pixeleket, magát a képet hatékonyan tárolni és kirajzolni. Fontos megjegyezni, hogy mivel a pixelenkénti rajzolás jelentős teljesítményt igényel a rengeteg memórialvasás és -írás művelete miatt, így a kirajzolás ciklusában egy-egy nem optimalizált művelet erős csökkenést jelent a végső teljesítményben.

3.2. Naiv pixelalapú renderelés

A pixelek reprezentációjának legegyszerűbb formája, ha egy pixel minden komponense egy külön unsigned charként (0–255) kerül tárolásra. Ez a tárolási forma, amennyiben nem igénylünk semmilyen speciális tárolási formát, gyakorlatilag egyezik a GPU-ban létrehozott textúra (RGBA) tárolási formájával, melyet a CPU-oldalra való mappelés során érhetünk el. Ez alapján egy képet tároló struktúra könnyedén létrehozható:

```
struct texture t
{
    unsigned int width;           // szélesség
    unsigned int height;          // magasság
    unsigned int internalFormat;   // a textúra formátuma
    unsigned char *texels;         // képpontok memória címe
};
```

A fenti definícióból a texels lesz az a memóriablokk, amely majd tartalmazza a betöltött kép pixeleit, amely egy RGBA-kép esetén a következőképpen jön létre:

```
texinfo.internalFormat = 4; // RGBA-t jelöl
texinfo.texels = (unsigned char *)malloc (sizeof (unsigned char) *
texinfo.width * texinfo.height * texinfo.internalFormat);
```

Egy így létrehozott pixeltömb kirajzolása viszonylag egyszerű. A példában feltételezzük, hogy a GPU-oldalon megtörtént a textúra és a mögöttes, képernyőt lefedő mesh létrehozása. A textúra (képernyő) képpontjait pedig a *gGraphics.GetFrameBuffer()* wrapper osztállyal elfedve tudjuk elérni. A rajzolás mechanizmusa ilyenkor:

```
unsigned int wHelp = mTexture.width*4;
CFrameBuffer fbuffer = gGraphics.GetFrameBuffer();

for(unsigned int i=0; i < mTexture.width; i++){
    for(unsigned int j=0; j < mTexture.height; j++){

        unsigned int iHelp = i*4;

        unsigned char a = *(mTexture.texels + j*wHelp + iHelp + 3);
        unsigned char r = *(mTexture.texels + j*wHelp + iHelp + 2);
```

```

        unsigned char g = *(mTexture.texels + j*wHelp + iHelp + 1);
        unsigned char b = *(mTexture.texels + j*wHelp + iHelp);

        // do not draw if pixel is fully transparent
        if (a == 255) continue;

        if (position.x + x >= fbuffer.width || position.y + y >= fbuffer.height)
            continue;

        unsigned int offset = position.y*y*fbuffer.width + position.x + x;

        fbuffer.mFrameBuffer[offset] = r;
        fbuffer.mFrameBuffer[offset + 1] = g;
        fbuffer.mFrameBuffer[offset + 2] = b;
        fbuffer.mFrameBuffer[offset + 3] = a;
    }
}

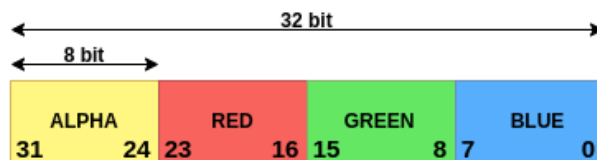
```

Mivel az ilyen típusú képek alfa csatornát is tartalmaznak, a rajzolás pixelenként történik [18]. Amennyiben a képernyőn számos objektum van, melyek mérete akár nagyobb is lehet, akkor jelentős teljesítményt igényel a másodpercenkénti 50-60 képernyőfrissítés. Természetesen a fenti mintakód már tartalmaz egy-két optimalizációt (*wHelp*, *iHelp*), melyek segítenek, de sajnos ezek sem lesznek elegendők. Fontos megjegyezni, hogy a fenti példa nem tér ki arra, hogy az átlátszóság miatt az adott pixel írásakor az alatta lévő pixel színével azt össze kellene mosni.

A pixelek komponenseinek külön-külön tárolása nem csak a kirajzolásnál nem hatékony. Bármilyen egyéb műveletet kell elvégezni a képpel (forgatás, tükrözés, nyújtás stb.), a teljesítmény sosem lesz kielégítő.

3.2.1. Módosított pixelreprezentáció

Hatékonyabb megközelítés, ha a pixelek komponenseit valahogyan egyben tudjuk kezelni. RGBA esetén minden komponens 1 byte, azaz egy pixel tárolásához 4 byte = 32 bit szükséges. Ez a méret pedig gyakorlatilag megfelel egy integerértéknek. Tehát egy pixel komponensei „becsomagolhatók” egy integerváltozóba a következő bitpozíciókkal: kék komponens: 7-0, zöld komponens: 15-8, piros komponens: 23-16, alfa komponens: 31-24.



4. ábra. Pixel komponensek 32 bites integer esetén

Ahhoz, hogy ez az új tárolási forma kihasználható legyen, a meglévő, központi memóriában lévő képi elemeket konvertálni kell ebbe a formába. Ezt a legcélszerűbb közvetlenül a képek fájlrendszerről való betöltése után megtenni. A következő mintakód bemutatja ezt az átalakítást:

```

int texel_length = mTexture.width*mTexture.height;
mTexture.int texels = (uint32_t *)malloc(sizeof(uint32_t)*texel_length);

for(unsigned int i=0; i < mTexture.width; i++){
    for(unsigned int j=0; j < mTexture.height; j++){

        unsigned char r = *(mTexture.texels+j*mTexture.width*4 + (i*4) + 2 );
        unsigned char g = *(mTexture.texels+j*mTexture.width*4 + (i*4) + 1 );
        unsigned char b = *(mTexture.texels+j*mTexture.width*4 + (i*4) );

        uint32_t rr = (uint32_t)(r);
        uint32_t gg = (uint32_t)(g);
        uint32_t bb = (uint32_t)(b);
        uint32_t aa;

        unsigned char a = *(mTexture.texels+j*mTexture.width*4 + (i*4) + 3);
        aa = (uint32_t)(a);

        uint32_t color = (aa << 24) | (rr << 16) | (gg << 8) | bb;
        mTexture.int_texels[j * mTexture.width + i] = color;
    }
}

```

A mintakódban bár nem került külön kiemelésre, de az *mTexture* struktúrába egy *uint32_t *int_texels*; változó is bekerült, amely az integeralapú pixelek tömbjét hívott tárolni. A folyamat minden pixelen végighalad és elkészíti annak 32 bites integer megfelelőjét a szükséges bitművelettel:

```
uint32_t color = (aa << 24) | (rr << 16) | (gg << 8) | bb;
```

Ettől fogva minden pixelt egyetlen integer érték reprezentál, amely komoly előnyöket szolgáltat majd a további műveletekben.

3.2.2. Rajzolás integeralapú pixelekkel

Természetesen az egyik legfontosabb művelet mindig a rajzolás. De gyakorlatilag bármilyen műveletet is veszünk, jellegében nagyon hasonlítani fog az alább bemutatott kódhoz:

```

CFramebuffer fbuffer = gGraphics.GetFramebuffer();

for(unsigned int i=0; i < mTexture.width; ++i) {
    for(unsigned int j=0; j < mTexture.height; j++) {

        uint32_t w = j * mpTexture.width + i;
        uint32_t color = *(m_pTexture->int_texels + w);

        unsigned int offset = (position.y+j)*fbuffer.width + (position.x + i);

        if (offset < screen buffer size)
            fbuffer.mFramebuffer[offset] = color;
    }
}

```


Jól látszik, hogy a rajzolás művelete lényegesen egyszerűsödött, sokkal átláthatóbb megoldást kapunk ezzel az új struktúrával. Mindezek mellett természetesen a rajzolás sebességében is számottevő javulást kell tapasztalnunk, amelyet a tesztelés során mutatunk be.

3.2.3. Vertikális tükrözés

Egy további érdekes művelet a pixelhalmaz tükrözése. Erre gyakran van szükség grafikus alkalmazások esetében. Jelen példa a vertikális tükrözést mutatja be. Az integralapú pixelstruktúra szintén nagy előnyt jelent a műveletnél.

```
void CTexture::FlipVertical()
{
    int th = mTexture.height;
    int tw = mTexture.width;

    uint32 t *texels = new uint32 t[th * tw * 4];

    for(unsigned int i=0; i < th; i++){
        memcpy(texels+((th-1)-i)*tw, mTexture.int_texels+i*tw, tw*4);
    }

    delete [] m pTexture.int texels;
    mTexture.int texels = texels;
}
```

4. Futási eredmények

A különböző raszterizációs technikák különböző teljesítménnyel rendelkeznek. A következőkben ezek sebességbeli különbségeit több tesztfeladat segítségével mutatjuk be. A tesztprogramok elkészítéséhez a C++ nyelvet és a GCC 11.2 fordítót használtuk, a méréseket pedig egy Core i7-9700-es 3 GHz CPU-val végeztük el Linux operációs rendszeren. A tesztekhez Intel UHD Graphics 630 integrált videómegjelenítőt használtunk. A szoftveres framebuffer megjelenítéséhez az OpenGL keretrendszert alkalmaztuk, ahol a GPU-memóriában létrehozott teljes képernyős textúra megoldását választottuk. A pixelek raszterizációjában nem alkalmaztunk CPU-oldalon párhuzamosítást, az eredmények 1 processzormag teljesítményét tükrözik.

A tesztek három csoportba bonthatók az alkalmazott képméret alapján. A legnagyobb kép 1024×1024 , majd 256×256 végül 64×64 . A tesztek során ugyanabból a típusú képből kis és nagy volumennel is rajzoltunk.

1. táblázat

Futási eredmények

Textúra mérete	Darabszám	Raszterizálás sebessége (FPS)	
		Naív rajzolás	Integralapú pixelrajzolás
1024×768	1	108	550
1024×768	10	15	398

Textúra mérete	Darabszám	Raszterizálás sebessége (FPS)	
		Naív rajzolás	Integralapú pixelrajzolás
256 × 256	10	139	658
256 × 256	100	16	323
64 × 64	100	225	727
64 × 64	200	124	655

Az eredmények jól mutatják, hogy az integralapú pixeltárolás és -rajzolás minden esetben a legjobb eredménnyel szerepelt, amely megfelel az elvártaknak. Jól látszik, hogy a klasszikus byte-onként rajzolásra nem nagyon építhető egy számítógépes játék, míg az integralapú megoldás már alkalmas lenne erre a feladatra. Mindezek mellett nem szabad elfeledkezni arról, hogy itt még csak 1 CPU core került bevonásra. Többszálú rajzolás segítségével a fenti eredmények messze felülmúlhatók.

5. Összefoglalás

A modern számítógépes vizualizációban ma leginkább csak a GPU-orientált megközelítésekkel találkozhatunk. A terület nagyon sokat fejlődött az elmúlt évek során, mára hatalmas piacot tudhat magának. A fejlődés előrehaladtával bár egyre jobb eszközök kerültek a programozó kezébe, megszűnt a fejlesztő teljhatalma a grafika programozásában. Ma már minden háromszögalapon készül, a pixel szintű egyedi képmanipulációk, és CPUoldal a renderelési folyamatba való bevonása eltűnt. Jelen cikkben bemutatásra került az a technika, amellyel bár körülményesen, de továbbra is lehetőség nyílik a szoftveres megoldások alkalmazására. A teljesítményadatokból jól látszik, hogy lehetne olyan magas szintű megoldásokat kidolgozni, amelyek nem kizárólag a GPU-ra épülnének. Az így létrejövő alkalmazások nem függenek annyira a GPU-tól, növelve ezzel a platformok közötti átjárhatóságot, valamint a pixelszintű hozzáférhetőség új, kreatív lehetőségeket adna a fejlesztők kezébe.

Irodalomjegyzék

- [1] Zach, B. (2011). *A Modern Approach to Software Rasterization*. University Workshop, Taylor University.
- [2] TransGaming Inc. (2023). *Swiftshader Software GPU Toolkit*.
- [3] Microsoft Corporation (2023). *Windows advanced rasterization platform (warp) guid*, <https://learn.microsoft.com/en-us/windows/win32/direct3darticles/directx-warp>, 2023
- [4] Michael Abrash (2009). Rasterization on Larrabee: A First Look at the Larrabee New Instructions (LRBni) in Action, Game Developers Conference, <https://gdcvault.com/play/1402/Rasterization-on-Larrabee-A-First>, 2009
- [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, Vol. 27, Issue 3.
<http://doi.acm.org/10.1145/1360612.1360617>
- [6] David W. (2018). *OpenGL 4 Shading Language Cookbook: Build high-quality, real-time 3D graphics with OpenGL 4.6, GLSL 4.6 and C++17, 3rd Edition*. Packt Publishing, 2018
 - [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., Naty H. (2018). *Real-Time Rendering, Fourth Edition 4th Edition*, A. K. Peters, 2018
 - [9] Sugerman, J., Fatahalian, K., Boulos, S., Akeley, K., Hanrahan, P. (2009). *Gramps: A programming model for graphics pipelines*. ACM Trans. Graph., 28, 4, 1–11.
<https://doi.org/10.1145/1477926.1477930>
 - [10] Khronos Group (2023). *OpenGL reference guide*. <https://www.khronos.org/opengl/>.
 - [11] Agner, F. (2021). *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.
 - [14] RAD Game Tools (2021). *Pixomatic advanced software rasterizer*.
 - [15] Eberly, D. (2006). *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. 2nd edition, CRC Press.
 - [16] Mileff, P., Dudra, J. (2022). The Past and the Future of Computer Visualization. *Production Systems and Information Engineering*, Vol. 10, No. 1, pp. 16–29.
<https://doi.org/10.32968/psaie.2022.1.2>.
 - [17] Video Electronics Standards Association (1198). *VESA BIOS Extension - Core Functions Standard*. Version 3.0.
 - [18] Mileff, P., Dudra, J. (2014). Advanced 2D Rasterization on Modern CPUs. *Applied Information Science, Engineering and Technology: Selected Topics from the Field of Production Information Engineering and IT for Manufacturing: Theory and Practice, Series: Topics in Intelligent Engineering and Informatics*, Vol. 7, Chapter 5, Springer International publishing, pp. 63–79. https://doi.org/10.1007/978-3-319-01919-2_5
 - [19] Xlib – C Language X Interface (2023). <https://www.x.org/wiki/>.
 - [20] *Windows GDI technology* (2023). https://learn.microsoft.com/en-us/windows/win32/api/_gdi/.