Valósidejű grafika a videojátékokban

Szabó Dávid[1], Dr. habil. Illés Zoltán[2]

{[1]sasasoft,[2]illes}@inf.elte.hu
ELTE IK

Absztrakt. Mit csinál egy játékfejlesztő? A játék szónak köszönhetően komolytalan a téma hangzata, de valóban így van? Hogyan kell játékokat készíteni? Olyan könnyű és egyszerű ez a feladat, mint a játékokkal játszani? Ezekre a kérdésekre szeretnénk választ adni ebben a cikkben, mely a videojátékok, mint valósidejű grafikus alkalmazások fejlesztésébe biztosít betekintést. Bemutatjuk a szükséges lépéseket, melyeket egy interaktív játékalkalmazás elvégez a képernyőn megjelenéshez, illetve bepillantást nyújtunk a valósághű 3D grafika előállításához használt technológiákba és algoritmusokba. Cikkünkben szemléltetjük, hogy a játékfejlesztés miért nem játék, miért egy komplex, nehéz ágazata a szoftverfejlesztésnek.

Kulcsszavak: valósidő, grafika, videojáték, shader, játék motor, grafikus motor

1. Bevezetés

A legtöbb videojáték a valóságnak egy szimulált és egyszerűsített változatát formálja meg. Szükség van egy virtuális világ (pálya) leírására, melyben különböző egymással interaktáló objektumok találhatók és jellemzően a felhasználó is befolyásolhatja, irányíthatja ezen objektumokat. A felhasználó felé képszekvencia és hanghatások segítségével mutatja be a virtuális világot. Mivel a felsorolt funkciókra szinte az összes játékban szükség van, ezért a videojáték ipar a fejlődése során a szimulációt végrehajtó és prezentáló algoritmusok jelentős részét újra felhasználható modulokba szervezte, ezeket nevezzük játék motoroknak.

1.1 Játékmotor

Egy játékmotor egy szoftverfejlesztői eszköz, melyet felhasználva nem szükséges egy új játék fejlesztéséhez a teljes szimulációs kódot újra implementálni. A motorban elérhető funkciókat módosítva és kiegészítve, mindössze a program egyedi részének elkészítése szükséges. Ezáltal egy általános játékmotor, mint a Unity3D [1], Unreal Engine [2] vagy bármely más, akár nyílt hozzáférésű, akár belső fejlesztésű motor tartalmaz különböző eszközöket, melyek gyakorian (vagy szinte mindig) szükségesek egy játék fejlesztésében.

Ilyen eszközök például a játékelemek menedzselése (Hogy néz ki? Hol van a térben? Hogy hat rá a fizika? Stb.), az erőforrások menedzselése (több gigabájtnyi kép és geometria adat dinamikus ki-és betöltése a memóriába), mesterséges intelligencia (gépi ellenfelek), audio motor (hangeffektek és zenék), grafikus motor (képi megjelenítés) és magát a szimulációt végző Game-Loop (periodikus reagálás a felhasználó interakcióra majd a hatásának prezentálása).

Ebben a cikkben a játékmotorok működését a bennük alrendszerként működő grafikus motor irányából megközelítve ismertetjük, mivel általában a játékmotorok architektúrájának megtervezésében jelentős szerepet játszik a grafikus megjelenítés.

1.2 Grafikus motor

A videojátékok és az átlagos grafikus felületű alkalmazások, mint például a Microsoft Word, Spotify és egyéb asztali programok megjelenítésében több közös vonás is felismerhető. A felsoroltak mind valósidejű grafikus alkalmazások, melyek a felhasználó interakcióira azonnal reagálnak és prezentálják azt a felhasználó felé. Mikor egy gombra mutatunk az egérkurzorral elvárjuk, hogy a gomb felvillanjon,

mikor kattintunk elvárjuk, hogy az alkalmazás reagáljon. Ugyanígy egy játékban mikor az "előre mozgás" billentyűjét lenyomjuk elvárjuk, hogy lássuk karakterünket elindulni a képernyőn.

Az azonnali reakció egy informális definíció, de a gyakorlatban vannak erősen definiált korlátok, melyek meghatározzák, hogy milyen gyorsan tud programunk reagálni a felhasználó mozdulataira. Ez a korlát a megjelenítőnk (monitor, televízió, képernyő stb.) frissítési rátája, a másodpercenként megjeleníthető képkockák száma. Grafikus alkalmazásainknak ezen megjelenítőket kell kiszolgálnia és a futás során folyamatosan a megfelelő számú képkockát kell előállítania az elvárt időzítéseknek és határidőknek megfelelően. [3]

1.2.1 Valósidejű grafika

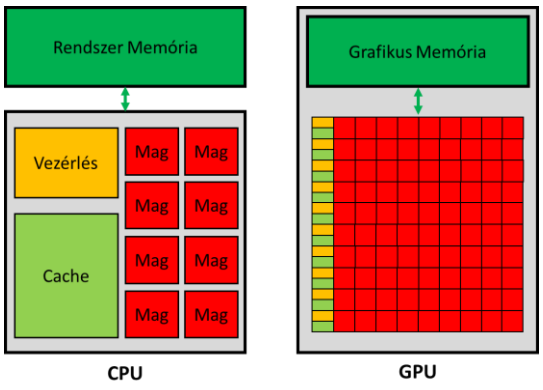
Egy mai átlagos monitor 60 hertz-es panellel rendelkezik, így 60 képkockát kell másodpercenként továbbítani az eszköz felé. Modernebb telefonok és televíziók 120 hertz-es, míg speciálisabb monitorok akár 144 vagy 240 hertz-es panelt használnak. Ez azt jelenti, hogy egy képkocka feldolgozására és előállítására nagyjából 16 milliszekundum számítási ideje van az alkalmazásunknak (1 másodperc / 60 hertz), melyet a program végrehajtása során folyamatosan és periodikusan be kell tartani, különben a grafika "szaggat", veszít a folyamatosságából és kizökkenti a felhasználót.

A manapság elterjedt televíziók 4k felbontásúak, tehát közel 8 millió képpontot (pixelt) jelenítenek meg egy képkockán (3840 pixel szélesség * 2160 pixel magasság). Egy grafikus alkalmazásnak az összes pixelre meg kell adnia egy színt, gyakorlatilag egy RGB byte struktúrát (a három színkomponensből kikeverhetők a megjelenítők által előállítható színek). Ebből kiszámítható, hogy körülbelül 25 Mb egy 4k képkocka eltárolásához szükséges memóriaigény (3 byte * ~8 millió pixel) és ilyen képkockából másodpercenkét 60-ra van szükség a megjelenítő kiszolgálásához, így egy grafikus alkalmazásnak nagyjából 1.5 Gb adatot kell előállítania másodpercenként (60 hertz * ~25 Mb).

A fejlesztők hamar felismerték, hogy egy CPU ekkora adatmennyiség párhuzamosított előállítására nem alkalmas, ezért megalkották a grafikai feladatok végrehajtására specializált GPU-t (Graphics Processing Unit).

2. CPU és GPU

Egy általános CPU 4-8 nagy teljesítményű processzormaggal rendelkezik, mely hosszú szekvenciába rendezett programok változatos utasításainak végrehajtására megfelelő, de egy 4k felbontású képkocka előállítása során 4-esével / 8-asával kellene feldolgoznia a ~8 millió pixelt. Egy GPU ezzel szemben alacsonyabb teljesítményű processzormagokat használ, de több tízezer ilyen mag érhető el egyetlen egységen. [4]



1. ábra: CPU és GPU architektúrájának összehasonlítása

2.1 GPU

Egy GPU nagyszerűen alkalmazható erősen párhuzamosított feladatokra és a grafika (a ~8 millió pixel feldolgozása), pontosan ilyen feladat. Ezen specializált egyszerű processzormagok nem előnyösek hosszú, változatos szekvenciális programok végrehajtására, de ez a grafikában ritkán szükséges. Mivel ekkora az architektúrális különbség a két processzortípus között, ezért eltérő a programozásuk módja is. Egy általános CPU-ra C-ben, C#-ban, Java-ban vagy hasonló programozási nyelvekben írt alkalmazást nem lehet egyszerűen GPU-ra lefordítani és ott futtatni. A GPU programozáshoz Grafikus API programfejlesztői könyvtárakat kell használnunk, melyek specifikációjukon keresztül elérhetővé teszik a GPU vezérlését és programozását. Ilyen könyvtárak az OpenGL [5], DirectX, Vulkan [6] és Metal, melyek C nyelvhez biztosítanak interfészeket. Manapság biztonsággal kijelenthetjük, hogy ha egy alkalmazás képes valamilyen grafikus felületet megjeleníteni a képernyőn (legyen szó, akár gombokból, feliratokból és egyéb komponensekből összerakott felületekről vagy vászonra rajzolt alakzatokról, formákról), akkor a háttérben a felsorolt Grafikus API-k egyikét használja a látvány prezentálásához.

2.2 Grafikus API programozás

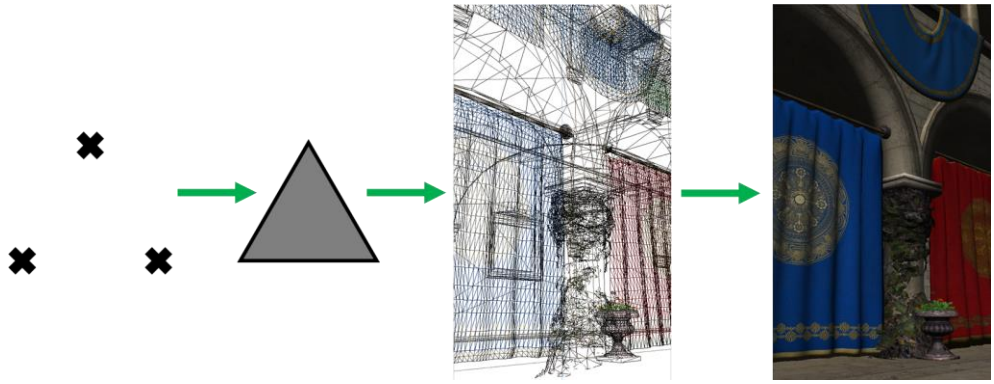
Grafikus programozás során megkülönböztetjük a feladatokat melyeket a CPU és a GPU végez. A két eszköz párhuzamosan dolgozik és végrehajtásuk adott pontjain szinkronizálják munkáik eredményét. A Grafikus API struktúráit és függvényeit használva a CPU-n kell a GPU-nak szánt feladatokat és parancsokat előkészíteni, melyhez jellemzően C vagy C++ nyelvet használnak. A parancsokat a GPU-nak átadva a GPU megkezdi a parancsok végrehajtását a CPU pedig folytathatja más feladatokkal (vagy akár a következő GPU parancsok előállításával is).

2.2.1 GPU parancsok

Ezek a parancsok jellemzően a GPU erőforrásaink konfigurálása és a rajzolási állapot beállítása. [7] A GPU állapot alapú rajzolással dolgozik. Hasonlóan, mint például a Logo teknőcgrafikája esetén, először be kell állítani rajzolási információkat (Logo-ban szín, tollvastagság, irány stb.), majd erre az állapotra ki kell adni a rajzolási parancsot (Logo-ban előre/hátra), mely elindítja a számítást és elkezdi a megfelelő pixelek színezését. Az állapothoz jellemzően különböző erőforrásokra van szükség a GPU-n, ezek közül a legtöbbet előre létre kell hozni és inicializálni kell, majd rajzolás előtt már csak az aktiválásuk szükséges. Szükség van buffer-ekre az adatok tárolásához a GPU memóriájában, textúrákra a kép-szerű adatokhoz, shader-ekre, melyek a GPU-n futó programok és algoritmusok és pipeline-ra (szerelőszalag), mely az ezek közötti kapcsolatot írja le az a következő rajzoláshoz. Összegezve: ahhoz, hogy a GPU-val „rajzoljunk" először be kell állítani a rajzolni kívánt geometria adatait, a rajzoláshoz használatos szerelőszalagot, majd ki kell adni a rajzolási parancsot.

2.2.2 Geometriák, modellek

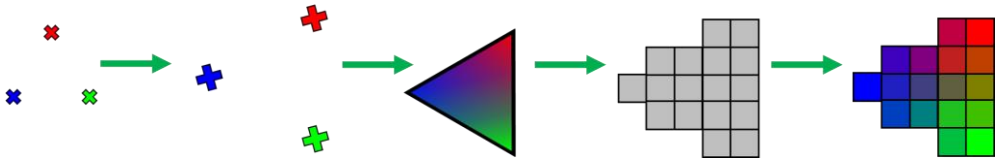
A Grafikus API-kkal jellemzően primitív geometriákat rajzolunk, pontokat, vonalakat és háromszögeket. Ha bármi komplexszebb alakzatra van szükség, akkor ezen primitívek összetételéből kell a geometriát felépíteni. A primitíveket a térbeli háromdimenziós csúcspontjaik felsorolásával adjuk meg (kétdimenziós esetben elég csupán az X és Y koordinátát használni a pozícióhoz, a Z-vel pedig a sorrendet adhatjuk meg).



2. ábra: Virtuális világ grafikájának összeállítása geometriák csúcspontjaiból. Balról jobbra: csúcspontok, háromszög, komplex geometriák háromszögekből felépítve, végső kép [8] fénymodellel megjelenítve

3. Grafikus Szerelőszalag

Egy rajzolási parancs kiadásakor a GPU a grafikus szerelőszalagot futtatja, mely a GPU-ba beprogramozott és beégetett algoritmusok sorozata. A szerelőszalag feldolgozza a beállított állapotot és ez alapján pixeleket színez ki a képernyőn. Feladatai, hogy a megadott geometria csúcspontjait transzformálja a térben, a csúcspontok összeszerkesztése komplexszebb alakzatokká, majd az alakzatok képernyőre vágása. Ezután meg kell állapítani, hogy melyik alakzat mely pixeleket fedi le, végül a lefedett pixeleket ki kell színezni.

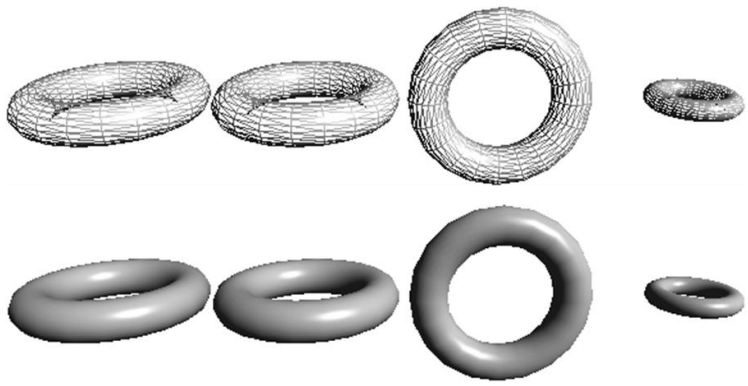


3. ábra: A Grafikus Szerelőszalag lépései. Balról jobbra: Csúcspontok (bemeneti adat), csúcspontok feldolgozása (transzformációk), geometria összeszerkesztése, lefedett pixelek meghatározása, pixelek színezése (kimeneti adat)

A szerelőszalag egyes lépései során gyakran előfordul, hogy nagy mennyiségű adatot kell feldolgozni. Amennyiben a teljes képernyőt lefedi egy geometria, akkor a pixelek színezését a teljes képernyőre le kell futtatni (tehát 4k megjelenítőn ~8 milliószor). Egy mai videojátékban átlagosan 1-3 millió csúcspont jelenik meg egyszerre a képernyőn, melyeket transzformálni kell majd összeszerkeszteni, így a szerelőszalag korai fázisai is jelentős adatmennyiséget dolgoznak fel. Míg egy mai modern CPU képtelen megbirkózni ekkora számossággal, a GPU-k előbb ismertetett architektúrája tökéletesen alkalmas, mert erre a feladatra lett tervezve.

3.1 Csúcspontok feldolgozása

A lineáris algebrában tanult vektor-mátrix szorzással transzformálhatjuk a geometriák csúcspontjait. A megfelelő transzformációs mátrixszal mozgathatjuk, forgathatjuk és méretezhetjük az alakzatokat. Inicializálásnál elég csak egyetlen egyszer identitás állapotban definiálnunk a geometriák csúcspontjait, később kirajzolásnál a transzformációkkal helyezhetjük el őket a virtuális világunkban, akár egyszerre több különböző helyen és orientációban is. [9] A kamera leírása, azaz a "virtuális térbeli szemünk" elhelyezése is transzformációkkal történik, ez határozza meg, hogy mit látunk a virtuális világunkból. A transzformációk alkalmazásához a geometriák minden csúcspontjának pozícióját meg kell szorozni az aktuális mátrixszal, ezáltal a megfelelő helyre és orientációba transzformálva a teljes alakzatot.



4. ábra: Csúcspontok transzformációi. Balról jobbra: identitás (nincs transzformáció), eltolás, elforgatás, méretezés

3.2 Pixelek feldolgozása

A geometriák összeszerkesztése után a szerelőszalag kiszámítja, hogy ezek a képernyő mely pixeleit fedik le. Az alakzatok csúcspontjaihoz további információkat is megadhatunk, például szín, normálvektor (felfelé mutató irány az alakzat felszínén) stb., mely adatok súlyozottan átlagolva lesznek elérhetők a lefedett pixelek színezésekor. A színezés történhet egyszerűen, például az átlagolt adatokból megkapott szín módosítás nélküli felhasználásával. Jellemzően a pixelek színezése során szimuláljuk a fénymodellt, valamilyen algoritmus alapján.

3.2.1 Fénymodellek

Az emberi szem a fényeket érzékeli, a fényeknek köszönhetően látunk. Ebből kifolyólag, ha képesek vagyunk a fényeket fizikailag korrekten megjeleníteni, akkor az összes valóságban megtalálható vizuális hatást (pl. árnyékok, tükröződések, fénytörés, fényszóródás stb.) is sikeresen megjelenítettük. A fények szimulálását a pixelek szintjén végezzük, mivel ez a legkisebb egység melyet a megjelenítő prezentálhat, tehát ez a felhasználó eszközén elérhető maximális részletességi szint.

A fizikailag korrekt fények reprezentálásának egyenlete (rendering equation) [10], már 1986-ban be lett mutatva, melyet megvalósítva teljesen valósághű képet kaphatunk. Ez az egyenlet leírja, hogy egy adott felület miként reagál a beérkező fénysugarakra. Mivel az egyenlet komplex műveleteket használ (például integrálást), így a mai napig nem lehetséges számítógépen az egyenlet pontos implementálása, csupán közelíteni tudjuk azt. Minél pontosabb a közelítés annál valóságosabb a fény/felület interakció, de az approximációt kiszámító kód teljesítményigénye is ennek függvényében növekszik.

$$I(x,x') = g(x,x')\left[e(x,x') + \int_S p(x,x',x'')I(x',x'')dx''\right] \quad (1)$$

ahol:

I(x, x'): x' beérkező irányból x kimenő irányba továbbított fény intenzitása

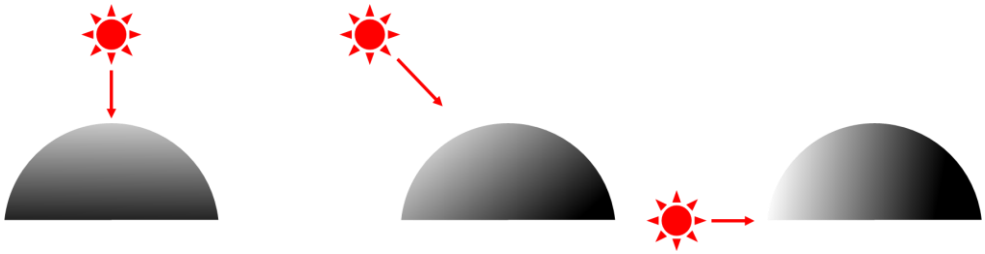
g(x, x'): geometria függvény

e(x, x'): x' irányból kibocsátott fény intenzitása x kimenő irányba

p(x, x', x''): x'' irányból a felületre beérkező és szétszóródó fény intenzitása x' kimenő irányba

Az egyenlet egy adott felületi pontra (pixelre) írja le a virtuális világ fényforrásai és környező objektumai alapján a visszavert fényt intenzitását (színét). A felületi pontot az x vektor irányából nézzük (a „virtuális szemünkből" a felületi pontra mutató irányvektor) és egy fényforrás (pl. nap, fáklya, villanykörte stb.) fénysugarai x' vektor irányából érkeznek a felületi pontra. Az egyenlet kimenete a

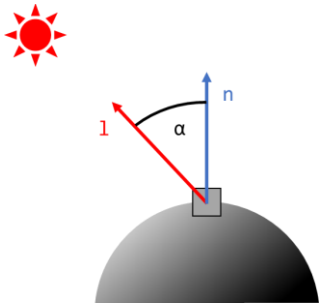
fényforrás fénysugarainak a felületi pontról a szemünkbe továbbított intenzitása (az adott szemszögből, milyen színűnek látjuk az objektumot azon a pixelen).



5. ábra: Fényforrás hatása egy objektum felületén.

3.2.2 Egyszerűsített fénymodell

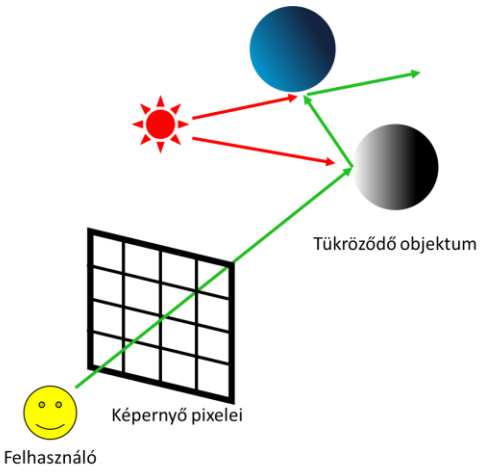
Általánosságban a fény szimulálásához egy pixelen szükségünk van a felületi normál vektorra, a fényforrás felé mutató irányvektorra, illetve a fény és a felület színére. A normálvektor és a fényforrás felé mutató vektor közötti szög határozza meg, hogy mennyire ideális a fény beesési szöge. Ha a két vektor párhuzamos, akkor 100%-ban megvilágítja a felületet a fény, ha merőleges, akkor egyáltalán nincs hatása a fénynek az adott pixelen. [11]



6. ábra: Egyszerűsített fénymodell kiszámításához szükséges adatok egy felületi ponton (pixelen). l: a fényforrás felé mutató irányvektor, n: a felületi pont normálvektora, α: l és n által bezárt szög (fény intenzitása a felületi ponton)

3.2.3. Fénymodell sugárkövetéssel

A videojátékok grafikus motorjainak számára az elmúlt években elérhetetlen volt a filmekben használt sokkal valósághűbb fény szimulációs metódus, a raytracing (sugárkövetés). [12] Ahogy a neve is sugallja, sugárkövetés esetén a fényforrások fénysugarainak útját követjük és szimuláljuk. Megvizsgáljuk, hogy mely felületi pontokra, mely fényforrások sugarai jutnak el (nem takarja egyéb objektum az utat a felületi pont és a fényforrás között), továbbá kiszámítjuk a felületen megtört és tükröződő sugarakat, melyeket tovább követve információkat kaphatunk a felületi pont tükröződéséről és belső fényvisszaverési tulajdonságairól.



7. ábra: Fénymodell szimulálása sugárkövetéssel

A modern GPU-k már képesek valós időben a teljes képernyő felbontás töredékén végezni a sugárkövetést és azt mesterséges intelligencia algoritmusokkal felnagyítva alkalmazni a teljes képre. Habár már az is nagyszerű eredmény, hogy erre 16 milliszekundum alatt képes a mai technológia, a filmekben látott teljes felbontású valósidejű sugárkövetésre még nem áll készen.

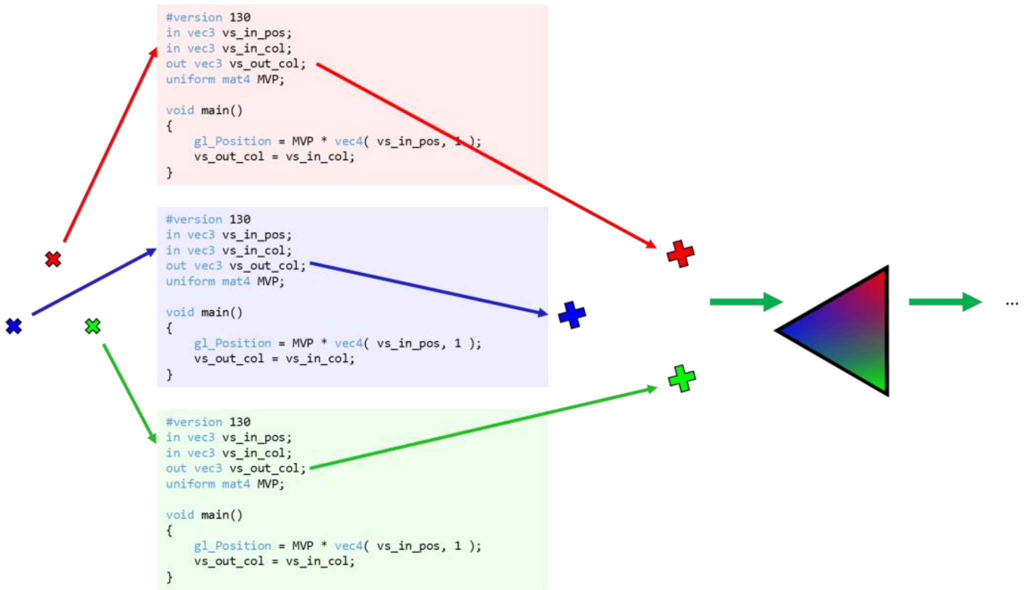
4. Shader

A szerelőszalag működésének jelentős része "be van égetve" a GPU architektúrájába. Speciális tranzisztorok és áramkörök találhatók minden GPU processzormagban, melyek kizárólag a szerelőszalag egyes feladatait képesek elvégezni, mindezt nagy sebességgel az általános számítást végző áramkörökhöz képes. A szerelőszalag egyes részei viszont programozhatók, saját algoritmust írhatunk, melyeket a GPU fog végrehajtani a szerelőszalag adott fázisaiban. Ezeket a kicsi programokat nevezzük shadereknek (árnyaló, először csak a fény árnyékolás kiszámítására használták), melyeket a használt Grafikus API által meghatározott specializált C szerű nyelven kell elkészíteni. [13]

OpenGL esetén ez a nyelv a GLSL (OpenGL Shading Language) [14], melynek a Vulkan egy bináris változatát használja. DirectX shader-eket HLSL (High-Level Shading Lanugage), Metal shader-eket pedig MSL (Metal Shading Language) nyelven kell készíteni. Az előbb bemutatott szerelőszalag két programozható fázisa a csúcspont feldolgozás és a pixel feldolgozás, melyekhez készíthetünk shader programokat.

4.1 Shader programozás koncepciója

Egy kirajzolás során ugyanazt a csúcspont shader programot használjuk a geometria összes csúcspontjára és ugyanazt a pixel shader programot az alakzatból keletkező összes pixelre. Így a shader programba írt main függvény többször is futtatva lesz egy kirajzolás során, csupán a program bemenete lesz a lefutások között más. A csúcspont transzformációs shader esetén minden csúcspontra ugyanaz a kód fut, melyben a pozíció vektort megszorozzuk a megadott mátrixszal, de a szerelőszalag mindig másik csúcspont pozícióját fogja a shader futtatásakor megadni, ezáltal az összes pontra futtatva a programot. A kirajzoláshoz használandó shader-eket, a GPU állapotában kell megadni a szerelőszalag beállításával.



8. ábra: Transzformációs csúcspont shader GLSL nyelven. Az ábrán a geometriánk három csúcspontból áll, ezért ugyanaz a csúcspont shader háromszor fog futni, mindhárom futtatás más csúcspont adatait kapja meg bemenetként és ezáltal más kimenetet is fognak generálni.

A shader-ek programozásának gondolatmenete eltér egy hagyományos C, C#, Java vagy hasonló nyelvekben írt szekvenciális programok fejlesztésétől. Meg kell barátkozni a koncepcióval, hogy ugyanaz a shader kód egyszerre többször fog futni, mindössze a bemeneti értékek lesznek mások (csúcspont feldolgozás esetén a geometria csúcspontok adatai, pixel feldolgozás esetén a lefedett pixelek meghatározása során előállított átlagolt csúcspont adatok). Ezen felül lehetnek további változóink, melynek értéke ugyanaz a geometria összes csúcspontjának és pixelének feldolgozása során. Ilyen értékek lehetnek a transzformációs mátrixok vagy a fényforrások információi. GLSL nyelvben az in, out és uniform kulcsszavakkal adhatjuk meg, hogy melyik változóink bemeneti, kimeneti, illetve az összes adathoz ugyanolyan értékkel bíró változók.

5. Modern videojáték motorok

Egy mai videojáték motor többnyire elrejti az előbbi alacsony szintű funkciókat a felhasználói elől, mivel ezek minden játék fejlesztéséhez szükséges metódusok. A shader-ek programozása a mai napig egy fontos lépése a grafikus effektek megjelenítésének, ezek továbbra is mind valamilyen szerelőszalagban futó egyedi shader-ekkel készülnek. A shader-ek ismétlődő részei, mint a transzformációk vagy a fények szimulációja beépített makrókként és függvényekként érhetők el, így ezen részek újra-implementálása nem szükséges, mindössze az egyedi effekt hozzáfűzése a feladat. Természetesen a hozzáadott kódnak kompatibilisnek kell lennie a motor elvárásaival és beépített algoritmusaival, különben ezeket is nekünk kell teljesen újra implementálnunk (feltéve, hogy a motor támogatja).

5.1 Grafika absztrakció a játékmotorokban

A motorok jellemzően megkülönböztetik a Mesh-t (modellek, geometriák) és Material-t (anyagtulajdonságok). [15] A Mesh-ek írják le, hogy milyen alakzatokat jelenítünk meg, mik a csúcspontok és milyen csúcspont adatok érhetők el. Itt lehet szó statikus objektumokról, mint épületek, domborzatok, fák stb. vagy akár dinamikusan formázható modellekről is, mint például a karakterek (hajló végtagok,

forgó fej). Ezen Mesh-ek megjelenítéséhez használt shader-eket a Material-ok határozzák meg. A Material írja le, hogy milyen shader-eket használjon a szerelőszalag a kirajzoláshoz és azon shader-ek milyen paraméterekkel dolgozzanak (textúrák, színek, változó értékek stb.). Végül szükség van valamilyen Transzformációs komponensre, amely megadja, hogy egy játékbeli objektum hol van a virtuális térben. A motor ezen komponensből számítja ki a transzformációs mátrixokat, melyeket a Materialban kiválasztott shader-nek ad át kirajzolás előtt.

5.1.1 Hagyományos grafikus motorok

Összegezve egy grafikus motor egy képkockájának megjelenítése elképzelhető a következő pszeudo algoritmussal. Az algoritmus több egymásba ágyazott foreach ciklusból épül fel, melyek a Materialokat, Mesh-eket és Objektum Transzformációkat sorolják fel. [16] Először végig iterálunk az összes jelenleg használt Material-on. Minden iterációban először a GPU állapotába beállítjuk az aktuális Material adatait, majd végig iterálunk az összes Mesh-en, amely használja az aktuális Material-t. Minden iterációban az aktuális Mesh-t beállítjuk a GPU állapotába, majd felsoroljuk az összes objektum Transzformációs komponenseit, amelyek a jelenlegi Mesh kirajzolásához a jelenlegi Material-t használják. Ha az objektum látható az adott pozícióban (nem mögöttünk van, nincs takarva), akkor beállítjuk a GPU állapotába a transzformációs mátrixot és kiadjuk az állapotra a rajzolási parancsot (ezen a ponton az állapotban a megfelelő Material, Mesh és Transzformáció aktív).

Ez a megközelítés univerzálisan képes több különböző geometriából, akár több példányt megjeleníteni a szükséges fény szimulációval és egyéb grafikus effektekkel. Több motor (pl. Unity3D, Godot [17]) még manapság is ezt a megközelítést alkalmazza, mivel megfelelően általános, valamint gyengébb teljesítményű vagy régebbi funkcionalitású GPU-kon is megfelelően fut. A Mesh-ek és Material-ok számának növelésével fokozatosan teljesítmény igényesebb lesz az algoritmus, míg el nem jutunk a küszöbig, ahol már a motor nem lesz képes a szükséges 60 képkockát előállítani másodpercenként.

5.1.2 GPU vezérelt grafikus motorok

Egyes játékmotorok (pl. Unreal Engine, id Tech) az újabb GPU driven rendering (GPU vezérelt megjelenítés) megközelítését alkalmazzák, melynek hátránya, hogy gyengébb és régebbi GPU-k nem támogatják, de a modernebb eszközökön sokkal jobb teljesítményt biztosít, mint a hagyományos algoritmus. GPU driven rendering során az összes Mesh és Material információt egy-egy nagy GPU buffer-be vonjuk össze és a GPU-n egyetlen speciális rajzolási utasítással, megfelelően indexelve és összepárosítva rajzoljuk ki a játékelemeket. [18] Ezzel a metódussal nincs szükség a CPU kódban a többszörösen egymásba ágyazott nagy iterációszámú ciklusokra, mert ezt is a GPU végzi a saját párhuzamosítási eszközeivel.

6. Konklúzió

Cikkünkben a videojátékok grafikus alrendszeréhez használt szoftverfejlesztési eszközök használatáról biztosítottunk egy átfogó képet. A videójátékok, mint interaktív valósidejű grafikus alkalmazások a GPU (grafikus processzor) felhasználásával érik el a megjelenítéshez szükséges teljesítményt. A GPU-k programozása nagyban eltér az általános CPU programozástól. Áttekintettük a grafikus szerelőszalag működését és shader programok fejlesztését, melyekkel transzformálhatjuk a játék modelljeit és megfelelő fényszimulációval jeleníthetjük meg azokat.

Az ismertetett metódusokat egy mai játékmotor tartalmazza magában és felhasználhatóvá teszi a játékfejlesztők számára, de ezek kiegészítéséhez szükséges a belső rendszer alapjainak áttekintése. A technológia folyamatos fejlődéséhez pedig elengedhetetlen a grafikus motorok programozásának alapos ismerete, mivel a fejlesztés ezen rendszerek bővítésével vagy éppen teljesen új alapokra helyezésével jár.

Irodalom

1. Unity Technologies: *Unity*, https://unity.com/ (utoljára megtekintve: 2022.11.18.)
2. Epic Games, *Unreal Engine*, https://www.unrealengine.com/ (utoljára megtekintve: 2022.11.18.)
3. D. Szabó, Z. Illés, V. B. Heizlerné: *Real-Time functionality in Windows*, INFODIDACT (2018) 263-264
4. NVIDIA Corporation: *NVIDIA Fermi Architecture Whitepaper*, https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (utoljára megtekintve: 2022.11.18.)
5. D. Szabó, Z. Illés: *Real-time OpenGL graphics in modern C#*, DIDMATTECH (2019)
6. D. Szabó, Z. Illés: *Vulkan in C# for Multi-Platform Real-Time Graphics,* 18th International Conference on Emerging eLearning Technologies and Applications (2020) 687-692
7. D. Szabó, Z. Illés: *Real-time rendering with OpenGL and Vulkan in C#*, Recent Innovations in Computing: ICRIC (2021)
8. Jimmie Bergmann: *Sponza GitHub*, https://github.com/jimmiebergmann/Sponza (utoljára megtekintve: 2022.11.18.)
9. T. Akenine-Möller, E. Haines, N. Hoffman: Real-Time Rendering, Third Edition, Taylor & Francis Group (2008)
10. J. T. Kajiya: *The Rendering Equation*, SIGGRAPH (1986)
11. B. T. Phong: *Illumination for Computer Generated Pictures*, Communications of the ACM (1975)
12. T. Akenine-Möller, E. Haines: *Ray Tracing Gems*, Apress (2019)
13. Y. He, T. Foley, T. Hofstee, H. Long, K. Fatahalian: *Shader Components: Modular and High Performance Shader Development*, ACM Transactions on Graphics (2017)
14. John Kessenich, Dave Baldwin: *The OpenGL shading Language (Version 4.5)*, https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.50.pdf (utoljára megtekintve: 2022.11.18.)
15. B. Karis: *Real Shading in Unreal Engine 4*, SIGGRAPH 2013 Course: Physically Based Shading in Theory and Practice (2013)
16. Sebastian Aaltonen, Tim Cooper, Natalya Tatarchuk: *YouTube: SIGGRAPH 2021 REAC: Unity Rendering Architecture*, https://www.youtube.com/watch?v=6LzcXPIWUbc (utoljára megtekintve: 2022.11.18.)
17. Juan Linietsky, Ariel Manzur: *Godot Engine*, https://godotengine.org/ (utoljára megtekintve: 2022.11.18.)
18. Brian Karis, Rune Stubbe, Graham Wihlidal: *YouTube: A Deep Dive into Nanite Virtualized Geometry*, https://www.youtube.com/watch?v=eviSykqSUUw (utoljára megtekintve: 2022.11.18.)