ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

DIPLOMAMUNKA

OE-NIK	Hallgató neve:	**Kergyó István**
2023	Hallgató törzskönyvi száma:	**T/008843/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Kiberfizikai Rendszerek Intézet

DIPLOMAMUNKA FELADATLAP

Hallgató neve: **Kergyó István**
Törzskönyvi száma: T/008843/FI12904/N
Neptun kódja: [redacted]

A diplomamunka címe:

3D megjelenítés a Ray Tracing módszer alkalmazásával és a GPU felhasználásával
3D rendering with Ray Tracing technique using the GPU

Intézményi konzulens: Sziklai Zsolt
Külső konzulens:

Beadási határidő: 2022. december 15.

A feladat

Tervezzen meg és készítsen el egy olyan alkalmazást, amely képes 3D modellek megjelenítésére, Ray Tracing módszert alkalmazva GPU használatával. A grafikus kártyák teljesítményének növekedésével, valamint a GPGPU megjelenésével elérhetővé vált, hogy a különböző megjelenítési technikákat, GPU használatával végezzük. A Ray Tracing segítségével valósághű megjelenítést érhetünk el, ami számos területen felhasználható, mint a háromdimenziós animációk vagy a videójátékok terén.
Az alkalmazáson belül a felhasználó elhelyezhet objektumokat, amik egy jelenetként megjelenítésre kerülnek az alkalmazás által. Egy jelenet tartalmazza a kamerát, valamint az összes objektumot, amik valamilyen modellező programban készültek.

A diplomamunkának tartalmaznia kell:

- a feladat részletes leírását,
- a Ray Tracing bemutatását és felhasználási területeit,
- a feladat tervezési és megvalósítási lépéseit,
- mérések és eredmények ismertetését,
- továbbfejlesztési lehetőségeket.

Ph.

.....
Dr. Fleiner Rita
intézetigazgató

A diplomamunka elévülésének határideje: **2024. december 15.**
(OE TVSz 55.§ szerint)

A diplomamunkát beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

HALLGATÓI NYILATKOZAT

Alulírott Kergyó István [redacted] hallgató kijelentem, hogy a szakdolgozat / diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban / diplomamunkámban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022. december 5.

.....

hallgató aláírása

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar
9. sz. melléklet

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Kergyó István ..... Neptun kód: [redacted] ..... Tagozat: Esti .....

Telefon: [redacted] Levelezési cím (pl: lakcím): [redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

3D megjelenítés a Ray Tracing módszer alkalmazásával és a GPU felhasználásával ...

Szakdolgozat / Diplomamunka[2] címe angolul:

3D rendering with Ray Tracing technique using the GPU .....

Intézményi konzulens: Sziklai Zsolt .....

Külső konzulens: .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2021.09.05	Diplomamunka bemutatása	
2.	2021.09.20	Diplomamunka állapotának ismertetése	
3.	2021.10.08	Feldolgozott irodalom áttekintése	
4.	2021.10.23	További kutatások áttekintése	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2021.12.12.....

.....
intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar
9. sz. melléklet

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Kergyó István ..... [redacted] ..... Esti .....

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

3D megjelenítés a Ray Tracing módszer alkalmazásával és a GPU felhasználásával ...

Szakdolgozat / Diplomamunka[2] címe angolul:

3D rendering with Ray Tracing technique using the GPU.....

Intézményi konzulens: Külső konzulens:

Sziklai Zsolt..... .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.01.15	Feldolgozott irodalom áttekintése	
2.	2022.01.25	Konzultáció a megvalósításról	
3.	2022.02.21	Meglévő terv áttekintése	
4.	2022.04.08	Megvalósítás állapotának áttekintése	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2022.04.24.....

.....
intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar
9. sz. melléklet

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Kergyó István ..... Neptun kód: [redacted] ..... Tagozat: Esti .....

Telefon: [redacted] Levelezési cím (pl: lakcím): [redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

3D megjelenítés a Ray Tracing módszer alkalmazásával és a GPU felhasználásával ...

Szakdolgozat / Diplomamunka[2] címe angolul:

3D rendering with Ray Tracing technique using the GPU .....

Intézményi konzulens: Sziklai Zsolt.....

Külső konzulens: .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.09.10	Terv áttekintése	
2.	2022.10.22	Megvalósítás áttekintése	
3.	2022.11.20	Dolgozat véglegesítése	
4.	2022.12.02	Dolgozat áttekintése	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2022.12.05.....

.....
intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

TARTALOMJEGYZÉK

1 BEVEZETÉS

1.1 Probléma bemutatása

A számítógépek megjelenésekor még nem létezett a karakteres megjelenítésen kívül más grafikai elemek ábrázolása a számítógépeken. Ahogy fejlődött a tudomány és az eszközök nagyobb teljesítménnyel rendelkeztek, először megjelent a kétdimenziós ábrázolási mód, majd később a háromdimenziós. Ezzel együtt kialakult a Számítógépes Grafika tudományterület, ami a számítógépes megjelenítéssel foglalkozik. A terület célja, hogy a valósághoz minél közelebbi megjelenítést tudjon a számítógép nyújtani.

A háromdimenziós megjelenítési technológiákat leginkább a film, illetve a játékipar használja fel, viszont a két iparág különböző szempontokat helyez előtérbe. A játékiparban a valós idejű megjelenítés a legfontosabb, ezáltal fontos, hogy a grafikai elemek feldolgozása gyors legyen, míg a filmiparban a valósághű megjelenítés kerül előtérbe, a megjelenítési idő pedig mellékes.

Ahogy videókártyák fejlődtek, úgy került egyre inkább előtérbe a valós idejű megjelenítés az erőforrásigényesebb megjelenítési módszerek közül is. Míg kezdetben a háromdimenziós megjelenítésnél kizárólag a raszterezés működhetett valós időben, az utóbbi években megjelent videókártyák teljesítménye már képes hatékonyan használni más technológiákat is, úgy mint a ray tracing, amivel képesek szintén valós idejű megjelenítést elérni.

A diplomamunkám során, a ray tracing technológiát szeretném bemutatni, valamint elkészíteni egy megjelenítési motort, ami képes háromdimenziós grafikai modelleket megjeleníteni ray tracing technológiát használva úgy, hogy a számításokat a GPU-n végzi. Végezetül a méréseket szeretnék végezni különböző jelenetbeállításokkal és összehasonlítani azokat CPU-n elért eredményekkel.

1.2 Diplomamunkával kapcsolatos elvárások

- A ray tracing technológia bemutatása
- Hasonló projektek és szoftverek kutatása és bemutatása
- Az alkalmazás elkészítése és ennek ismertetése
- Eredmények kiértékelése

2 IRODALOMKUTATÁS

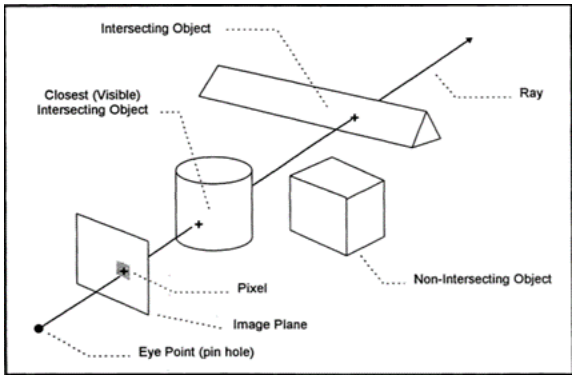
2.1 Előzetes kutatás

2.1.1 A technológia kialakulásának története

A ray tracingről szóló első dokumentáció az 1960-as években került nyilvánosságra Arthur Appel által, amiben ray tracing technológiát alkalmazott különböző felületek megjelenítéséhez egy megadott nézőpontból.

Ebben a tanulmányban Appel, egy olyan rendszert alkotott, ahonnan egy szemnek/origónak nevezett pontból sugarakat hozott létre, amik egy bizonyos síkot metszettek. Ez volt a kép sík. A síkon túl háromdimenziós objektumok szerepeltek, amiket az adott sugár vagy metszett, vagy nem. Amely objektumokat metszette a sugár, azokat kiválasztotta, valamint meghatározta ezek közül melyik van a legközelebb a sugár kezdőpontjához és ez megadta azt, hogy az adott pixelen melyik objektum látható, majd végezetül meghatározta a pixel színét.

Annak ellenére, hogy ebben az időben erősen korlátozott volt a számítási kapacitás, Appel jól felismerhető képeket tudott alkotni ezzel a technológiával.[1] [2]



2.1. ábra: Appel megvalósítása a Ray Tracing alapú megjelenítéshez [2]

Az 1970-es években további fejlődéseken esett át a számítógépes grafika területe, ekkor a kutató közösségek több megjelenítési megoldást is fejlesztettek és felhasználtak, ezek között volt a ray tracing is. Ebben az időszakban jelent meg a Gouraud és Phong modell, ami az objektumok megjelenítésénél figyelembe vett további tényezőket, így a megjelenítendő felület tartalmazott további tulajdonságokat (anyagtulajdonságokat), mint: diffúz (diffuse), tükröződés (specular), környezet (ambient). Ezen tulajdonságok

függőségben állnak a fényforrás és a felület közötti szöggel, valamint a nézőpont és a felület közötti szöggel egyaránt. Ezen összetevők által a képesek voltak jellemvonásokat adni az adott felületekhez, amik által valósághűbbnek tűntek és különböző anyagokhoz köthető tulajdonságokat tartalmaztak.[2]

Appel a megjelenítéshez egyetlen folyamatot alkalmazott, sugarakat küldött a megadott pixelen keresztül egy origóból indulva és kereste a metszéspontokat a megadott térben szereplő objektumokkal. Viszont a 70-es években ehhez hozzáadtak még egy lépést, ami azóta a technika része. Amikor egy metszéspontot talált az első folyamat, akkor a talált metszéspontból további sugarakat kerülnek kiküldésre. Ezek az úgynevezett árnyék sugarak, amik az árnyékokért felelnek, és a metszéspont fényforráshoz való viszonyát vizsgálják. A vizsgálat során azt nézik, hogy van-e bármilyen objektum a fényforrás és a metszéspont között. Ha nincs, akkor az adott fényforrás megvilágítja az adott pontot, míg ahol van, abból az irányból fedésben van, tehát nem érkezik fény az adott pontra. Végezetül, ahol a fény közvetlenül eléri a metszéspontot, ott kiszámításra kerülnek a tulajdonságok, mint a diffúz, illetve a tükröződés iránya kiszámításra kerül és ezáltal a színösszetétel.

Majd harmadik lépésben a tükröződés és fényvisszaverődés irányába további sugarakat keletkezhetnek rekurzívan, amik a többi tárgyról való vetületet vizsgálják, a reflexió tulajdonság alapján. Ezáltal amelyik objektum anyaga képes a tükröződésre, arra nagyobb hatással lesznek a környezetében lévő objektumok a kirajzolásnál.

Ebben az időszakban viszont, habár a technológia kidolgozott volt és könnyedén lehetett elvégezni vele a fényvisszaverődéshez szükséges számításokat, a számítási kapacitás nem volt elegendő a megjelenítéshez. Egy-egy alacsony felbontású kép megalkotása órákba és napokba is telhetett. Ennek a legfőbb oka a rengeteg sugár megalkotása és a rekurzió volt.

Az 1980-as években már a meglévő folyamat felgyorsítása volt a cél az algoritmikus után. A legfőbb terület a metszéspontok számítása jelentette, hiszen ez foglalta el a kalkuláció legnagyobb részét. Ekkor a négy legfontosabb kutatási területek a következőek voltak:

- Hatékonyabb metszéspont meghatározás sugarak és objektumok között.
- A metszéspontkeresés számának a csökkentése (hiszen minden objektumot megvizsgáltak az adott jelenetben, ezáltal nagyban függ az objektumok számától a se-

besség).
- A másodlagos sugarak csökkentése, amelyek nincsenek szignifikáns hatással az előállított képre.
- Hardverek fejlesztése, amelyek elősegítik a sugárkövetéses megjelenítést

Ezek után az 1990-es években kezdődött egy olyan fejlődési hullám, ami lehetővé tette a ray tracing alapú megjelenítést. Ekkor jelentek meg különböző szoftverek, amik képesek voltak a jelenetek kirajzolására ezen módszerek felhasználásával sokkal rövidebb időn belül. [2]

Napjainkban, a GGPU megjelenésével, új lehetőségek nyíltak meg a videókártyán lévő számítások elvégzésére, ezáltal egy új terület nyílt meg a ray tracinggel való megjelenítés előtt. Ennek következtében lehetőség nyílt, hogy akár valós idejű megjelenítést érjünk el, a hardver teljesítmény növekedéssel és a megfelelő optimalizáló algoritmusok használatával. Ez azt jelenti, hogy akár másodpercenként 30-60 kép megjelenítését is képes lehet elvégezni számítógép.

Az utóbbi évek során számos projekt és alkalmazás jelent meg, amik a számítógépes megjelenítés ezen területével foglalkoznak, irodalomkutatásom során ezek közül szeretnék ismertetni néhányat.

2.2 Houdini

A Houdini egy olyan modellező program, ami a kezdetektől fogva a procedurális, dinamikus megjelenítésre épült. Ez azt jelenti, hogy a jeleneteket képes dinamikus módon létrehozni algoritmikus úton, ezáltal a lehető legtöbb folyamatot automatizálni és segíteni a művészeket abban, hogy a munkafolyamatokat lerövidítsék.

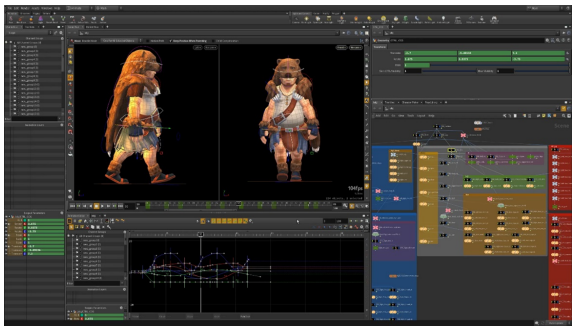
2.2.1 Megjelenítési motor

A Houdini a Mantra Engine-t használja a megjelenítéshez. Ez a motor lehetőséget ad arra, hogy több megjelenítés mód közül válasszon a felhasználó. Ezek a módok: Scanline, Ray Tracing, valamint fizika alapú megjelenítés. [3]

Gyorsaság szempontjából a scanline a leggyorsabb, viszont a fizika alapú megjelenítés van a valósághoz a legközelebb, az esetek többségében ez is javasolt. A megjelenítési

idő ilyenkor nem feltétlenül fontos, hiszen a legtöbb esetben egy videó megalkotása és megjelenítése nem probléma, ha nem valós időben történik.

A scanline az egyszerű raszterizálást használja fel megjelenítés során, amely az évek során talán a legkiforrottabb technikává vált és a játékiparban a gyorsasága miatt a legjobban elterjedt. A fizika alapú megjelenítés (PBR) pedig úgy próbálja elérni a valósághű megjelenítést, hogy a fény terjedését szimulálja a megadott térben. Célja a fotorealisztikus kép előállítása, ezáltal rendkívül számításigényes.



2.2. ábra: Houdini kezelőfelülete [3]

2.2.2 Összegzés

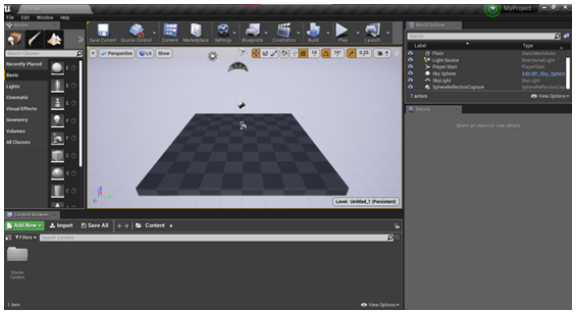
A Houdini egy erőteljes modellező motor, ami a Mantra motort használja a megjelenítéshez, ami biztosítja számára a 3 lehetséges megjelenítési módot. Az alkalmazás széles körben elterjedt a videós tartalomkészítők körében, valamint a procedurális megoldása miatt a játékiparban is szívesen használják

2.3 Unreal Engine

Az Unreal Engine egy széles körben elterjedt játékmotor. Az első verziója 1998-ban jelent meg, amikor az Unreal videójáték a piacra került. Azóta a motor ingyenesen elérhető vált, és jelenleg az Unreal Engine 5 verziónál tart.

A motor rengeteg komponensből áll és rengeteg platformot támogat, mint: Windows, Linux, Android, IOS. Modellek betöltése is egyszerű, több formátumot képes feldolgozni, majd átalakítani a számára legkönnyebben használható formátumra. [4]

A megjelenítési mód szintén személyre szabható a motoron belül, a sima raszterezéses megoldás és ray tracing megoldások között lehet választani, ezen kívül mindkettőhöz még rengeteg opció áll rendelkezésre, a megjelenítési motornak köszönhetően. Az Unreal Engine 5 a játékmotorok közül az elsők között volt, ami ingyenes elérhető játékmotorként ray tracing támogatottsággal rendelkezett.



2.3. ábra: Unreal Engine szerkesztőfelülete

2.3.1 Ray Tracing megjelenítés Unreal Engineben

Amíg a film iparban, valamint a képek létrehozásánál nem probléma ha a végső képi anyag előállítása és kirajzolása akár órákat vagy napokat vesz igénybe, a játékiparban ez nem megoldható. Ezáltal nagyon fontos, hogy az adott megjelenítő motor képes legyen a lehető leggyorsabban működni, feldolgozni a háromdimenziós grafikai modelleket és megjeleníteni azokat a képernyőn. Ez egy minimum elvárás, ami megnehezíti a számításigényes ray tracing technológia használatát ezen a területen. [5]

Az Unreal Engine -nek a valós idejű ray tracing megjelenítés nagy előnyt adott a videójáték ipar piacán

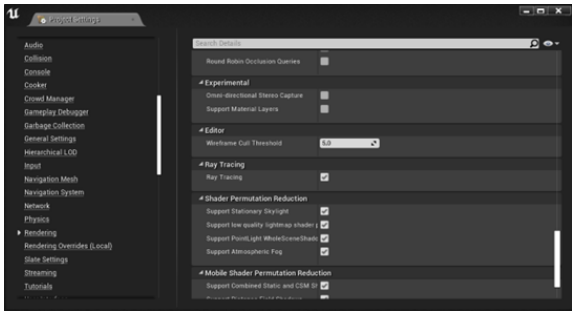
A motor két technikát alkalmaz ennek elérésére:

- Egy ray tracing megjelenítő, ami összekapcsolja a már meglévő raszterezési eljárással a ray tracing technikát
- Path tracer, a referenciák generálásához

A ray tracer lehetővé teszi az árnyékok és tükröződések, valamint az áttetszőségek valósághű megjelenítését, amik mind szükségesek a globális megvilágításhoz. Ezek természetesen mind valós időben történnek. Alacsony számú mintát használ, egy olyan zajcsökkentő algoritmussal, ami nagyon közeli eredményt képes adni a path tracer ered-

ményeihez. [5]

A path tracer egy fizika alapú path tracer ami alkalmas referencia képek megjelenítésére az adott jelenetben. A működése hasonló az offline megjelenítést alkalmazó módszerekhez, ugyanis megadott számú mintákat gyűjt az idő előrehaladtával és ezzel segíti elő az élethű megjelenítést.



2.4. ábra: A ray tracing engedélyezése a projekthez

A ray tracing támogatás jelenleg csak az NVIDIA RTX és GTX típusú videókártyákkal támogatott. Látható, hogy a megfelelő hardver elemek megléte után, a beállítása egyszerű.

A sugárkövetéses árnyékok megjelenítéséért felel. A fényforrások helyzetétől függően alkotja meg a jelenetben az árnyékokat, a fényforrás méretétől és a fény beesési szögétől lehetnek élesebbek és elmosott árnyékok az adott objektumokhoz. [5]

A fényvisszaverődéseket is képes szimulálni a rendszer úgy, hogy több rétegben vizsgálja azokat rekurzívan, több visszaverődést is számításba vesz. Ez személyre szabható, a felhasználó megadhatja, hogy mennyi legyen a maximális visszaverődés, mennyi visszaverődésig számolja ki a motor a fényvisszaverődések hatását. Természetesen ez a szám minél több, annál élethűbb lesz a megjelenített kép, viszont radikálisan csökken a teljesítmény is. [5]

Az áttetszőséget is képes kezelni, ami felelős a folyékony, illetve az üveg és a további átlátszó anyagok megjelenítéséért. Itt kezelni kell a fénytörést, valamint a részleges áttetszőséget.[5]

A globális megvilágítási megoldás pedig képes interaktív fényvisszaverődéseket szimulálni szintén valós időben képes olyan területeket megvilágítani, amelyek nem jutnak

közvetlen fényforráshoz. Ez egy fontos tulajdonság, amit alapból a ray tracing második lépése nem tartalmaz, hiszen ott, ha egy pontot nem ér megvilágítás semmilyen fényforrásból, akkor az árnyékban marad, hiába vetül rá fény más objektumról.[5]

2.3.2 Összegzés

Összegezve az Unreal Engine egy nagyon erőteljes eszköz a játékfejlesztőknek, de más iparágaknak is, ahol szükséges a gyors és mégis valósághű megjelenítés.

A motor a hagyományos megjelenítés mellett, tartalmaz egy implementációt a ray tracing által támogatott megjelenítéshez, ahol három módszert együtt használva képesek elérni azt, hogy mindez valós időben történjen. A három módszer ez esetben a raszterizálás, a ray tracing és a path tracing.

Egy másik nagy előnye, hogy a forráskódja szabadon hozzáférhető, ezáltal könnyedén tanulmányozható.

Rengeteg helyen használják jelenleg az Unreal Enginet-t. A játékiparban számos AAA videójáték készült a felhasználásával, de ezen kívül kutatási projektben, mint az önvezető autók szimulációjában, animációk elkészítésében, filmekben és számos más területen is előtérbe került a használata.

2.4 Ray Tracing implementáláció CUDA használatával

2.4.1 Projekt bemutatása

A kutatás célja az volt, hogy létrehozzanak ray tracing alapú megjelenítést CUDA használatával, tehát a számítások nagy részét a GPU végezze. A kutatás 2008-ban ért véget, ekkor már elérhetőek voltak a GPGPU által használható technológiák, viszont nem volt még sok valós idejű ray tracing megjelenítést alkalmazó szoftver a piacon, illetve a grafikus API-k is beépített raszterezéssel működtek, emiatt is volt szükség a CUDA használatára.[6]

A technológia lehetővé tette, hogy már nagyobb felbontású képek kirajzolása is végezhető legyen ray tracinget alkalmazva, akár interaktív környezetben, ahol a jelenetben

dinamikus modellek is vannak, esetleg a nézőpont is változik. Ehhez mindenképp valós idejű megjelenítés szükséges, minimum 30 képkocka másodpercenként. [6]

A CUDA ehhez biztosította a GPU-n végrehajtandó számításokat, amik szükséges a cél elérése érdekében. A projekt során egy GPU alapú megközelítést mutattak be a CUDA felhasználásával. Az optimális teljesítmény eléréséhez a kernel hangolásával érték el. [6]

2.4.2 Motiváció

A GPU-t felhasználó ray tracing megoldások sokkal hatékonyabban működnek mint a CPU alapú alkalmazások. Ennek az oka az erősen párhuzamosítható számítások, hiszen ugyanannyi sugarat kell vizsgálni, ahány pixelen történik a megjelenítés, ezen felül még a rekurziótól függően ez megsokszorozódhat, illetve minden pixel esetében meg kell vizsgálni rengeteg objektumot a metszés szempontjából. Rengeteg technika jött létre a GPU gyorsítást felhasználva a ray tracing megjelenítés szempontjából. A legtöbb esetben a SIMD (Single Input, Multiple Output) hardver jellemzőit használják fel a szomszédos sugarak közötti koherencia felhasználásával és korlátozzák a sugarankénti bejárást a gyorsítási struktúrák használatával.[6]

Mindkét megvalósítás bonyolítja a GPU sugárkövetés implementálását, ugyanis a szükséges adatstruktúrák és algoritmusok erősen eltérnek a CPU megfelelőiktől.[6]

A CUDA viszont támogatja az általános célú program futtatását rengeteg szálon, mindezt a GPU felhasználásával. [6]

2.4.3 Megvalósítás

A megvalósítás során a CUDA kernelben klasszikus verem alapú bejárást alkalmaztak a CUDA architektúrában, számos optimalizálással együtt. A textúrákat gyorsítótárba helyezték és ennek előnyét hatékonyan kihasználva, lineáris textúrákat használnak a k-d (k dimenziós) fák és a háromszög információk tárolására. Ez jelentős gyorsulást eredményez a leggyakrabban használt és bejárt csomópontok számára, valamint felgyorsítja a számítást azokon a területeken ahol a sugarak koherenciát mutatnak. Ezen kívül ugyanazon memóriába tárolják a sugarak kiindulópontját és irányát, hogy könnyen indexelhetővé te-

hessék azokat a hozzáférési mintától függetlenül.

2.4.4 Eredmény

Az eredmények mérése érdekében ugyanazon felbontás mellett tesztelték az általános teljesítményű, illetve az optimalizált kernelüket is. A méréseket 1024 X 1024 pixelen végezték. A legtöbb jelenet esetében az akkori GPU alapú implementációkat erősen felülmúlták a teljesítmény tekintetében, az árnyékolt, valamint a nem árnyékolt jelenetekben is. A legfigyelemreméltóbb eredményük az volt, hogy a 12,7 millió háromszöget tartalmazó, erőművet reprezentáló jelentben 30 képkocka megjelenítését képesek voltak elérni másodpercenként, árnyékokkal pedig 15,5 képkockát másodpercenként. [6]

2.4.5 Összegzés

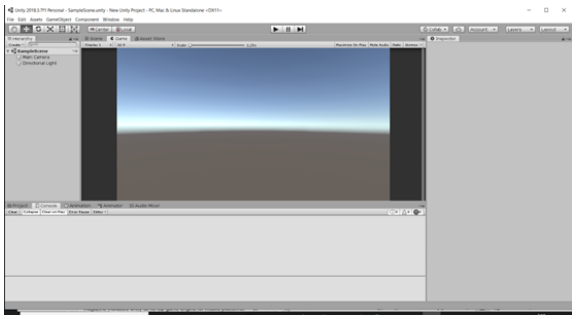
A projekt célja, hogy GPU-n megvalósított valós idejű ray tracinget alkalmazzanak, a mérések alapján sikerült. Természetesen attól függően, hogy a jelenet milyen komplex felépítésű, mennyi modellt, poligont és háromszöget tartalmaz. Ezen tulajdonságok nagy hatással vannak a teljesítményre, valamint az is, hogy kizárólag elsődleges sugarakat vizsgálnak, vagy esetleg árnyékokat is.

2.5 Unity

A Unity talán a jelenleg egyik legnépszerűbb játékmotor, ami szintén ingyenesen elérhető, ezáltal hatalmas felhasználói bázisa van. [7]

A Unity 2002 -ben került megalapításra, amikor egy nyílt forráskódú shader fordítót szeretett volna létrehozni Nicholas Francis és Joachim Ante, valamint David Helgason. A projektből később 2007-ben megjelent a Unity 2.0, ami már egy teljes játék motor volt rengeteg újdonsággal. Kezdetektől fogva a háromdimenziós megjelenítés volt az elsődleges célja, viszont a későbbiekben kétdimenziós megjelenítést is támogatta. [7]

Jelenleg a Unity az egyik legnépszerűbb játékmotorrá vált.



2.5. ábra: Unity szerkesztőfelülete

2.5.1 Ray Tracing megjelenítés Unityben

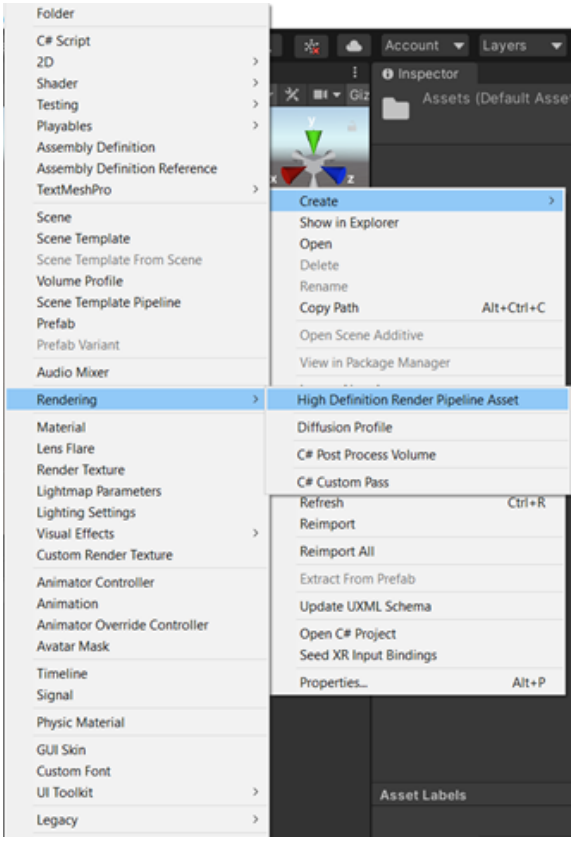
Unity-ben a valós idejű ray tracing technológia képes szimulálni, hogy a fény hogyan hat a többi objektumra, valós fizikai számításokat alkalmazva. Ez a technológia lehetővé teszi a globális megvilágítást, valamint az egyéb effektek, mint a fotorealisztikus vagy stilizált megjelenést, ezen kívül a tükröződést és az okklúziót is. [8]

Unity-ben a különböző megjelenítési eljárások szintén beállíthatóak, viszont kicsit más megközelítésből mint az eddig bemutatott rendszerekben. Unityben két megjelenítési módszer (pipeline) található. Az egyik az „Universal Render Pipeline" (URP), a másik pedig a „High Definition Render Pipeline" (HDRP). A Ray Tracing támogatáshoz, a HDRP-t szükséges használni. A HDRP hardveres gyorsítást is tartalmaz, elérhetővé téve azt, hogy a fényvisszaverődések megtörténnek akkor is, ha az adott objektumok a képernyőn kívülre esnek. [8]

A HDRP leginkább DirectX12 illetve DXR támogatással rendelkezik és szintén NVIDIA RTX vagy GTX típusú videókártyák szükségesek hozzá. A célközönsége jelenleg leginkább a mérnöki ipar, tervezők, grafikusok és autóipari szakemberek.

2.5.2 Ray Tracing használata Unityben

Az adott projekthez fel kell telepíteni a HDRP támogatást, majd ezután megjelenik egy ablak, ahol a szükséges további komponenseket lehet telepíteni. Ha ezek megvannak, akkor létre lehet hozni a pipeline-t, amit az adott projekthez hozzá is kell adni annak érdekében, hogy az a HDRP -t használja a továbbiakban.



2.6. ábra: HDRP létrehozása

Mindezek után a projekt kész arra, hogy ray tracinget használjon. A Unity ezen felül támogatja a ray tracing shadereket, amik szükségesek lehetnek a ray tracing alapú megjelenítéshez, ez azt is jelenti, hogy a már meglévő shadereket át kell konvertálni az új rendszerbe.

2.5.3 Összegzés

A Unity kezdetben egy webes alapú háromdimenziós videójáték-motor volt, ami az évek során hatalmas fejlődésen ment keresztül, ma már talán a legtöbb platformot támogató játékmotor a piacon. A ray tracing támogatással pedig elérték, hogy ne csak a játékiparban, hanem más területeken is szívesen használják a szoftvert.

A ray tracing megoldása ugyanazon területeket érinti, mint az Unreal Engine-nél, támogatja a globális megvilágítást, az áttetsző objektumokat, fényvisszaverődéseket rekurzívan, valamint az árnyékok képzését. Jelenleg a legjobb teljesítményt a DirectX12 használatával lehet elérni és a használatához a hardverkövetelményeknek meg kell felelni.

2.6 CryEngine

A CryEngine szintén egy játékmotor, amit a német Crytek cég fejleszt. Rengeteg csúcskategóriás játék készült a motort felhasználva, így széles körben ismerté vált, a korai években pedig a grafikai újításai tették egyedivé. [9]

2.6.1 Ray Tracing a CryEngineben

A CryEngine 2019-ben mutatta be a „Neon Noir" című projektet, amit teljes mértékben valós idejű ray tracing megoldását egy bemutató példa alkalmazáson belül. Valós idejű hálósugaras fénytöréseket és fényvisszaverődéseket használ a CryEngine totális megvilágítás megoldásával.[10]

A ray tracing támogatást amiatt szerették volna implementálni a motorba, mert ez egy új szintre emeli a grafikai megjelenítést, a felhasználói élmény nagyságrendekkel nőhet ennek hatására. A ray tracing elérhetővé teszi, hogy a fény fizikáját felhasználva jelenítsenek meg jeleneteket amik a valósághoz közelebb állnak mint az eddigi megjelenítési megoldások. [10]

A technológia fejlődésével elérhetővé vált a váltás az offline ray tracing megjelenítőkből a valós idejű megjelenítéshez, tehát nem kell órákig feldolgozni az adott jelenetet, hogy az a képernyőre kerüljön. A teljesítménynövekedésben kulcsfontosságú volt a gyors zajszűrő előfeldolgozó eljárás, ami előállítja a magas minőségű fényeket egy erősen korlátozott számú sugarak esetében.

A CryEngine 2015 óta használt ray tracing alapú megjelenítést különböző játékok esetében, viszont 2019-ig a CryEngine limitációja a globális megvilágítás esetén a tükröződésnek a részletessége volt valós időben. A legtöbb felülettel jól működött, viszont a nagyon sima és tükröződésképes felületen, mint a tükrök esetében ez problémás lehetett és ez késztette a készítőket arra hogy előre elkészített cube-map -re és helyi képernyőtér tükröződéseire hagyatkozzanak.[10]

Így 2019-ben megalkották az új globális megvilágítás rendszerüket. Az új háló (mesh) alapú sugár követő rendszer képes volt a magas minőségű hálókat és textúrákat, tökélete-

sen tükröződő tükröket és felületeket megjeleníteni kizárólag a GPU-n végző számítások felhasználásával. A projekt természetesen tartalmazta a szokásos ray tracing által használható előnyöket, mint a fénytörés, árnyékok megalkotása és okklúzió, viszont a projekt leginkább a tökéletesen tükröződő tárgyakra fókuszált, ugyanis azok sok esetben erőforrásigényesebbek.[10]

Mindezek segítenek a játékkészítőknek, művészeknek és tartalomkészítőknek, hogy olyan vizuális megjelenítést legyenek képesek előállítani egyszerű módon, ami valóságosabb érzetet ad a felhasználóknak. Ezen felül mindezek segítik a tartalomlétrehozását is, hiszen könnyebbé teszik az ilyen jelenetek megalkotását. [10]

A régi megoldásokkal, mint cube map használat vagy árnyék térképek használata nem teszik lehetővé a tükröződések teljes értékű megjelenítését, sőt ezen technikákkal rendkívül nehéz ezeket megoldani. Emiatt fontos, hogy ray tracinggel hatékonyan legyen megjeleníthető a fényvisszaverődés. [10]

2.7. ábra: CryEngine Neon Noir projektje, fényvisszaverődés kromatikus felületen [10]

A képen jól látható, hogy természetesnek tűnik, ahogyan a tér többi objektuma látható a felületen a tükröződés hatására. Ezt az új hálósugaras megoldással érték el. [10]

Az egyik legfontosabb komponense a legtöbb ray tracing alapú alkalmazásnak az a sugár metszeteket segítő gyorsító struktúrák alkalmazása, amik segítik a metszéspontok gyorsabb megtalálását. A CryEngine totális megvilágítási rendszere ezt már tartalmazta, ezután már viszonylag egyszerű volt számukra a továbbfejlesztés. Minden voxelnél tárolnak egy hivatkozást az azt átfedő háromszögekre, ezenkívül pedig annak tulajdonságait, mint az albedó, átlátszóság, és a normál adatokat. [10]

A voxel és a ray tracing adatokat összekötve biztosítják a flexibilitást. A diffúz sugarak esetében valódi hálókövetésre csak a sugár kezdőpontjának a közelében van szükség, de a sugár többi részén már cone tracinget is lehet alkalmazni vizuális effektek nélkül.

Hasonló gyorsítások működnek a többi típusú sugarak esetében is.[10]

2.8. ábra: Neon Noir példa jelenet[11]

A tükröződés nagyon jól kivehető a folyékony anyagok esetén, mint a képen szereplő pocsolya esetén is. Valamint látható, ahogyan a fény átszűrődik a rácsok között. Ez hagyományos raszterezési megoldással nem működhet dinamikus úton.

2.6.2 Összegzés

A CryEngine egy nagyon hatékony játék motor. Bár a platformok közül nem támogat annyit, mint az eddig vizsgált motorok, viszont a megjelenítésben rendkívül hatékonyan működik és sok lehetőséget tartalmaz a valósághű megjelenítésre.

A ray tracing megoldásukat 2015 óta fejlesztik, ami lehetőséget adott arra hogy az évek során optimalizálják az eljárásukat a valósághű megjelenés mellett.

3 TERVEZÉS

3.1 Folyamat bemutatása

A ray tracing az egyik jelentős fejlesztések közé tartozik a számítógépes grafika történelmében. Ahhoz, hogy egy képet vagy jelenetet realisztikusnak érzékeljünk, el kell érnünk olyan hatásokat, amik a valóságban is tapasztalhatóak. Ilyenek a következőek:

- Árnyékok: a takarásban lévő objektumok, amiket nem ér közvetlen fényforrás árnyékba kerülnek.
- Átlátszóság: a fénytörések bizonyos anyagokon, mint az üveg, vagy a víz.
- Tükröződés: fényvisszaverődés és csillogás a különböző anyagokon és tárgyakon, mint a tükör.
- Teljes megvilágítás: területekre ható fények és természetes fények szimulációja. Több fajta megvilágítási forma létezik.
- Realisztikus anyagok: a természetben lévő anyagoknak vannak tulajdonságaik, amiket nem szabad figyelmen kívül hagyni ha realisztikus megjelenítést szeretnénk elérni.

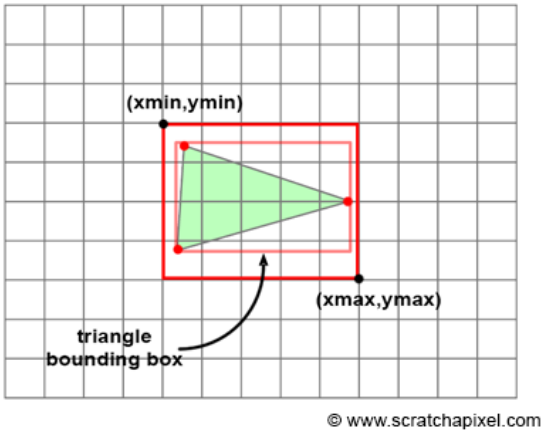
Ezen hatásokat hagyományos megjelenítési technikákkal nagyon nehéz vagy esetleg nem is lehet elérni.

3.2 Raszterizálás

Hagyományos megjelenítési technikának tekinthető a raszterizálás, ugyanis gyorsasága miatt a 3D megjelenítés területén az elsők között szerepelt, ezáltal a legtöbb helyen előforduló és jól kidolgozott technológiává alakult.
Habár manapság a nem valós idejű megjelenítéseknél már egyre inkább kiszorul, a valósidejű megjelenítésnél még mindig vezető szerepet tölt be.

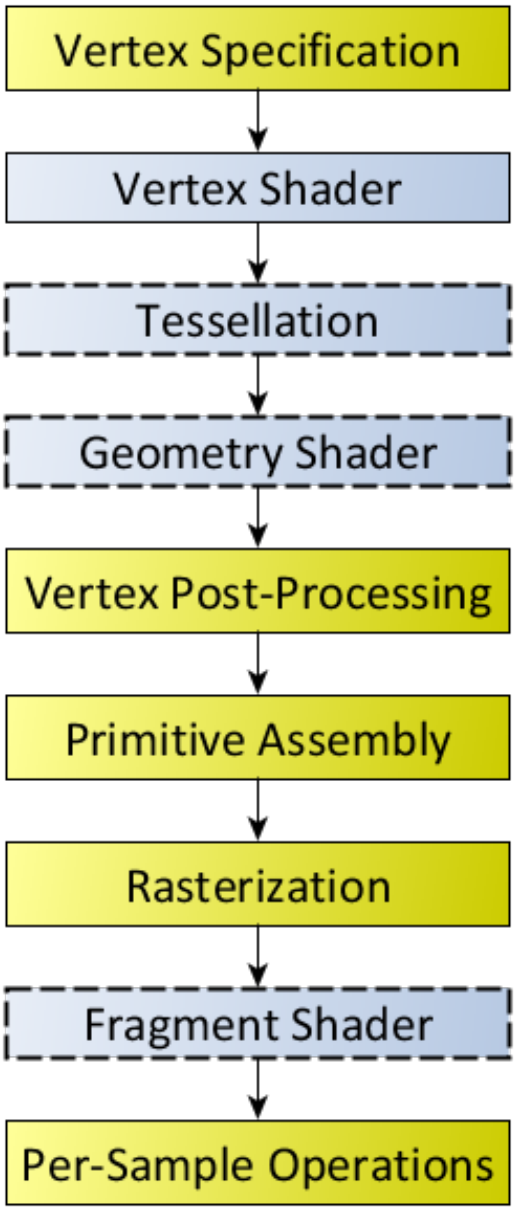
A technológia lényege, hogy meghatározza, milyen színű legyen az adott pixel a képernyőn. Ehhez az objektumoknak már a képernyő térben kell elhelyezkedniük, tehát a képernyőtérbe való vetítés transzformációnak már végbe kellett mennie. Ezután a raszterizáló végigmegy minden egyes objektumon, amik a leginkább háromszögek és megvizsgálja, hogy mely pixelek tartoznak hozzá a képernyőn, mely pixeleket fedi le az aktuális háromszög és mely pixelekhez kell meghatározni a színüket.

Ehhez több optimalizációs technika is létezik, alapvetően nem szükséges megvizsgálnia az összes pixelt, ugyanis elsősorban meghatározza, hogy mely keretek között lehetnek az érintett pixelek. Ha az objektum egy háromszög, akkor egy téglalapot határoz meg köré a minimum és maximum értékek alapján, tehát a határ koordináták alapján. Ezután ebben a téglalapban végigmegy minden egyes pixelen, és ha az adott pixel közepe a háromszögön belül esik, akkor az a pixel az adott háromszöghöz tartozik és megkapja az adott koordinátához tartozó színét.



3.1. ábra: Raszterizálás: pixelek meghatározása. [11]

Ezen felül, 3D-ben a raszterizálás során meg kell határozni azt is, hogy mely objektumok vannak az előtérben, ezáltal fedik a kamerához képest hátrébb lévő objektumokat. Ezt az úgynevezett Z- Buffer technikával lehet elérni. A Z-Buffer egy tömb, ami kamerához viszonyított távolságot jelöli. Mindegyik pixelhez eltárolásra kerül a Z-bufferben az objektumhoz tartozó érték. Ha egy olyan objektum kerül a raszterizálásra aminek a z értéke kisebb mint az eddig tárolt érték, akkor felülírásra kerül a pixel koordináta az új objektum koordinátájával.



3.2. ábra: Általános Rendering Pipeline. [12]

A raszterizálás során tehát objektumról objektumra haladva kerül vizsgálatra, hogy mely pixel, milyen színnel jelenik meg a képernyőn. Valamint egy-egy objektum vizsgálata során nem szükséges az összes pixelt megvizsgálni, elég annak csak egy szűk halmazát.

A raszterizálás egy már jól kifejlett folyamat, ezáltal ez a legtöbb grafikus API - ban nem is módosítható, hiszen nincs rá szükség. A pixel színek módosítására lehetőség van a fragment shaderekben, ami a raszterizálási folyamat után kerül végrehajtásra. Itt kerül

végrehajtásra a különböző anyagtulajdonságok és effektek, valamint árnyékok meghatározása is a hagyományos megjelenítési technikák során. Viszont mivel nincs elég információ az adott kamera pozíciójáról illetve a fények elhelyezkedéséről valamint a többi objektum elhelyezkedéséről a térben, ezáltal a realisztikus árnyékok és fénytörések nehezen megoldhatóak.

3.3 Ray Tracing

A Ray Tracing ezzel szemben egy teljesen más megközelítésből oldja meg az objektumok kirajzolását a képernyőre.

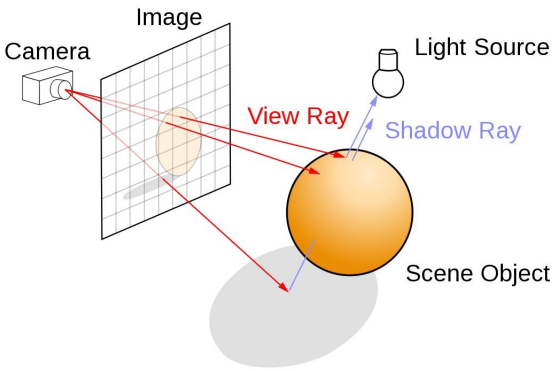
Ray tracing esetében egy alapvető különbség, hogy míg a reszterizálás során az objektumokon halad végig az algoritmus, a Ray Tracing esetén pixelről pixelre szükséges haladni.

A megjelenítés ray tracing esetében felbontható több lépésre.

3.3.1 Elsődleges sugarak

Az elsődleges sugarak célja, hogy megtalálják a metszéspontokat a megfelelő objektumokkal és azon belül is azzal, ami a legközelebb helyezkedik el a kamerától.

Az elsődleges sugarak a kamera origójától kerülnek kiküldésre úgy, hogy minden egyes pixelen keresztül kell menniük (nem feltétlenül a pixel középpontján). Miután kiküldésre kerül egy elsődleges sugár, meg kell vizsgálni, hogy mely objektumokkal kerül metszésre. Az objektumok közül pedig meg kell határozni a kamrához legközelebbit. A probléma az, hogy a metszéspont meghatározás nem olyan egyszerű, ugyanis optimalizáció nélkül végig kell menni minden egyes objektumon, ami komplex modellek esetében, több ezer, tízezer, százezer háromszöget is jelenthet objektumonként.



3.3. ábra: Ray Tracing elsődleges és másodlagos sugarai. [13]

3.3.2 Másodlagos sugarak/Árnyék sugarak

A másodlagos sugarak célja, hogy meghatározza, hogy az adott pont árnyékben van-e. Ezt úgy teszi meg, hogy az elsődleges sugár által meghatározott metszéspontból kiindulva az összes fényforrás felé egy másodlagos sugarat küld. Ezen sugarak esetében szintén az összes objektummal el kell végezni a metszéspont/ütközés vizsgálatot, ugyanis ha az elsődleges sugár által talált metszéspont és a fényforrás között egy másik objektum helyezkedik el, tehát van metszéspont találat, az azt jelenti, hogy az adott fényforrás közvetlenül nem éri el a pontot, így az nem világítja meg a pontot. Ha egy fényforrás sincs ami közvetlenül elér a ponthoz, akkor az árnyékban van.

Ez esetben szintén rengeteg számítást kell elvégezni, ugyanis ismét minden objektummal ütközés vizsgálatot kell végrehajtani, viszont ezt minden egyes fényforráshoz is meg kell tenni.

A fényforrások különbözőek lehetnek, és mindegyik esetében más algoritmus szükséges. A pont fényt (Point Light) esetében egy x, y, z koordinátával rendelkező pontból érkezik a fény, ami azt jelenti, hogy a metszéspont és pont fény közötti egyenes szakaszt kell vizsgálni.

Azonban van globális megvilágítás is, az úgynevezett irányfény (Directional Light), aminek csak az irányát vizsgáljuk, nem számít, hogy hol helyezkedik el, ez esetben egy "végtelen" hosszú szakaszt vizsgálunk, ami azt jelenti, hogy szintén minden objektumon végig kell menni, de ha bármelyik objektum is keresztezi ezt a szakaszt, akkor eltakarja a pontot a fényforrás elől. Míg a pontfény esetében ez nem így van, ha a fényforrás mögött helyezkedik el az objektum a metszésponthoz képest, akkor az nem takarja el azt.

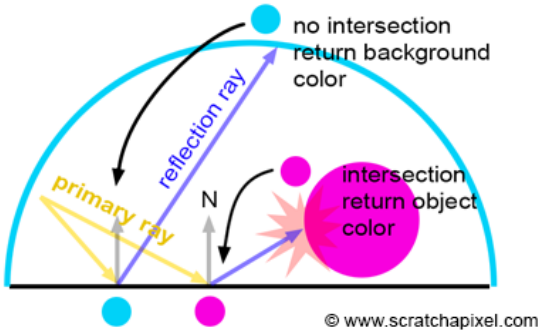
A másodlagos sugarakat tehát emiatt nevezik árnyék suagraknak (Shadow Rays), ugyanis ezek határozzák meg, hogy az adott pont árnyékban van-e.

Ha egy pont látható, akkor alkalmazzunk rá egy megvilágítási modell, amivel meghatározzuk a megvilágítások alapján a színét. Erre több modell is létezik, pl.: Phong modell.

3.3.3 Rekurzív sugarak

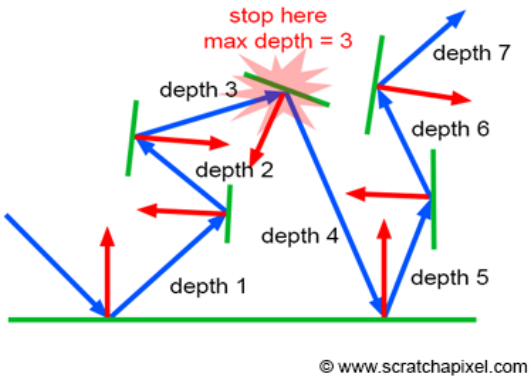
Végezetül a rekurzív sugarak kerülnek meghatározásra. Ez egy rekurzív művelet, emiatt is jött a rekurzív sugár elnevezés. Ez esetben rekurzívan a metszéspontból a tükröződés irányába küld sugarat és ismét metszéspontot vizsgál. Ha talál metszéspontot, akkor az adott pixel színe felülírásra kerül, ami nem jelenti azt, hogy teljes mértékben felülírja a megvilágításkor meghatározott színt, hanem azt egy súlyozott mértékben teszi, az anyag tükröződés/visszaverődés tulajdonsága alapján.

$$\text{szín} += \text{visszaverődés} \cdot \text{visszavert sugár színe} \tag{3.1}$$



3.4. ábra: Rekurzív sugarak. [14]

A művelet amiatt rekurzív, mert nem csak egy rekurzív sugarat hoz létre, hanem a talált metszéspontból újakat, szintén a visszaverődés irányába. Ez viszont akár a végtelenségig is eltarthat, pl.: egy olyan szobában ahol csak tükrök helyezkednek el. Emiatt megvalósításkor a legtöbb esetben érdemes egy limitet beállítani, hogy a rekurzió milyen mélységig mehet el.



3.5. ábra: Rekurzív sugarak mélysége. [14]

3.4 Összegzés

A raszterizálás és a ray tracing tehát két különböző technika. Habár a ray tracing könnyebb számításokat eredményez amikor a színek meghatározásáról van szó, hiszen több adat áll rendelkezésre és a megjelenítés alapja közelebb áll valósághoz mint a raszterizálás. Azonban a számítási igénye is nagyságrendekkel nagyobb.

A szűk keresztmetszet a ray tracing esetében az objektumok és sugarak közötti metszéspontok keresése és vizsgálata, ugyanis ezt kell a legtöbbször elvégezni. Így az optimalizálási technikák ezt szeretnék a lehető legjobban lecsökkenteni.

3.5 Optimalizálás

3.5.1 Párhuzamosítás

Az oka annak, hogy a ray tracing elérhető valós időben, az a videókártyák fejlődésének köszönhető, valamint annak, hogy az abban rejlő masszív párhuzamosítási potenciált felfedezték. Napjainkban a különböző grafikus API megoldásoknak köszönhetően lehetősége van a programozóknak arra, hogy a számítások elvégzéséhez a videókártyát is igénybe vegyék.

Mivel a ray tracing egy erősen párhuzamosítható feladat, ezáltal az egyik gyorsítási mód, a párhuzamosan elvégezhető feladatok maximális kihasználása. Ez azt jelenti, hogy a pixelenként elküldött elsődleges sugarak kalkulációja történhet egy időben, hiszen egymástól független feladatok, egymástól független pixeleken haladnak keresztül.

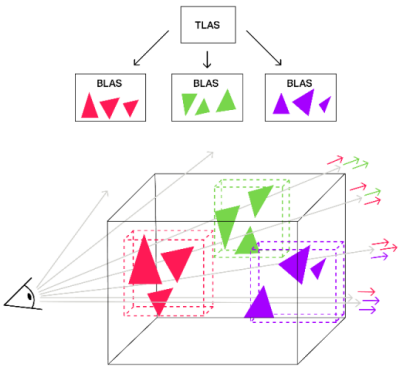
Ezen kívül a másodlagos sugarak is párhuzamosíthatóak fényforrások alapján. Ami azt jelenti, hogy vizsgálható a fényforrások közvetlen elérése fényforrásonként.

3.5.2 Gyorsítási struktúrák használata

Ahogy a raszterizálás esetében is volt módszer arra, hogy ne vizsgáljanak meg minden egyes pixelt, a ray tracing esetében is megoldható ez.

Ray tracing esetében a metszéspontvizsgálat az objektumtól függően különböző lehet. A leggyakoribb a háromszög és sugár metszéspont vizsgálat, de háromszög helyett lehet akár gömb, téglatest, négyzet. Természetesen ezen metszésvizsgálatok számítása is fontos része az optimalizációnak, azonban a legfontosabb, hogy az vizsgálatok számának csökkentése.

Ez úgy tehető meg, ha bizonyos téglatestek, úgynevezett határoló dobozok (Bounding Boxes) kerülnek definiálásra. Ezek hasonlóak a raszterizálás esetében definiált téglalappal a háromszög körül. Téglatestek definiálásra kerülnek egy egy objektum körül, illetve akár objektumok csoportja körül, fa struktúra vagy piramis szerűen. Mivel a téglatest és sugár metszéspontja könnyen meghatározható, ezáltal tudjuk, hogyha egy adott téglatestet nem metsz az adott sugár, akkor az abban lévő elemeket sem metszi. Ezáltal, ha egy objektum több tízezer háromszögből is áll, nem kell mindegyiken végigmenni, hiszen a téglatest vizsgálata során biztosra vehető, hogy nem lesz metszéspont találat. Ezen határoló téglatestek akár objektumokon belül is definiálásra kerülhetnek, ezáltal gyorsítva a keresést háromszögek egy csoportjára szűkítve a lehetséges metszéspontok halmazát.



3.6. ábra: Gyorsítási megoldás. [15]

3.6 Az alkalmazás tervezése

A cél egy olyan alkalmazás elkészítése, ami képes megjeleníteni objektumokat a képernyőn ray tracing technológia alkalmazásával, valós időben. Ehhez szükséges a párhuzamosítási technikák alkalmazása. Ezáltal szükséges egy API használata ami ezt lehetővé teszi.

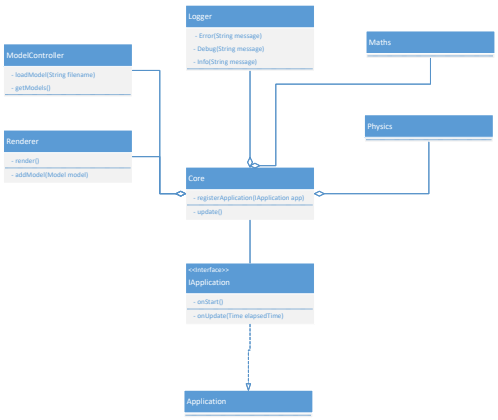
A DirectX12 tartalmazza a DXR komponenst ami segít a ray tracing megjelenítésben és ehhez a videókártyát használja. A megvalósítás során ez az API kerül használatra.

A programnak képesnek kell 3D objektumok megjelenítésére. Ezen felül a matematikai számítások elvégzésére mátrixok alapján és ezáltal a 3D transzformációk támogatására. A megjelenítéshez a ray tracing alapú megjelenítésre és a különböző anyagok definiálásának lehetőségére.

Minden fő része az alkalmazásnak, egy egy komponensként kerül implementálásra.

3.6.1 ModelController

A ModelController komponens felelős a modellek betöltéséért a memóriába és a megadott modellformátum feldolgozásáért. Ezen felül definiál egy model osztályt, amiben a modellhez tartozó további adatokat lehet tárolni, mint az anyag tulajdonságok és transzformációk.



3.7. ábra: Komponensek kapcsolata.

3.6.2 Logger

Általános logolásért felel úgy, hogy a program teljesítményét nem befolyásolja. Statikus osztály ami bárhonnan elérhető és a logolást fájlba, illetve terminálba is képes végezni.

Támogatja az Error, Info, illetve Debug típusú üzenetek létrehozását.

3.6.3 Maths

A számításokhoz szükséges matematikai számításokért felel. A legfontosabbak a mátrix transzformációk ezek közül: rotáció, eltolás, skálázás. Ezen kívül a különböző terek közti vetítésekért is felel.

3.6.4 Renderer

A legfontosabb komponens a Renderer. Ez az osztály felelős a jelenet képernyőn való megjelenítésért ray tracing módszer használatával. Tartalmazza a DirectX hívásokat és biztosítja a megjelenítésért felelős metódusokat.

3.6.5 Core

A Core kapcsolja össze a komponenseket úgy, hogy példányosítja azokat és a szükséges adatokat elküldi a megfelelő példánynak. Szintén tartalmazza a szimulációs ciklust, amiben megjelenítésre kerülnek az objektumok illetve kezeli a felhasználói interakciókat.

Ezen kívül tartalmaz IApplication példányokat amik az alkalmazást reprezentálják. Először az onStart() hívódik meg, amikor az applikáció betöltött, majd minden egyes megjelenítéskor az onUpdate(), elküldve az előző onUpdate() óta eltelt időt.

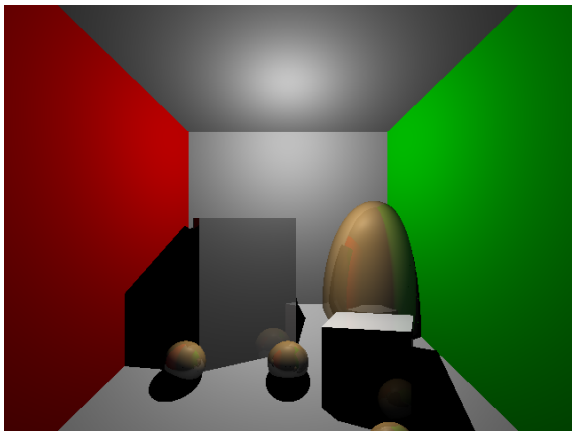
Tehát az IApplication interfészt implementáló osztállyal lehet kommunikálni a rendszerrel.

3.7 Kezdeti mérések

Kezdetben egy egy szálon futó, CPU használatával történő ray tracing alapú megjelenítést implementáló alkalmazást hoztam létre. Ez referenciának szolgál a későbbi mérésekhez.

A méréseket 640x480 felbontású képek megjelenítésén végeztem 5 mélységű rekurzív sugarakkal.

- CPU: Intel(R) Core(TM) i5-7600K CPU @ 3.80GHz
- GPU: NVIDIA GeForce GTX 1060 6GB
- Platform: Windows 10



3.8. ábra: Példa jelenet

Az 3.8 ábra egy egyszerű jelenetet tartalmaz, a ray tracing előnyeit bemutatva megtalálhatóak benne fénytörések, tükröződések és dinamikus árnyékok. A jelenet egy 640x480 felbontású képként lett megjelenítve. Egy szálon, csak a CPU-t használva a megjelenítési idő a megadott architektúrán: 4457 ms ~ 4,5 s, tehát 4,5 másodperc.

4 MEGVALÓSÍTÁS

4.1 DirectX 12 Ray Tracing bemutatása

4.1.1 Fejlődéstörténete

A DirectX egy API gyűjtemény amit a Microsoft fejlesztett. Ez különböző komponensekből áll, amik biztosítják az alacsony szintű hozzáférést a hardverekhez, különösképpen a videókártyához. A DirectX csak Windows rendszereken érhető el. [16]

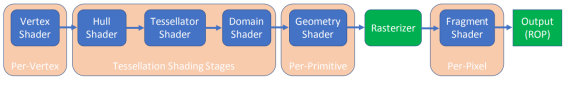
Az első verziója 1995-ben jelent meg Windows Game SDK név alatt, majd a DirectX 2.0 követte 1996-ban. Ezek után a DirectX 8.0 verzió volt az első nagyobb mérföldkő, ami lehetővé tette a fejlesztők számára, hogy Vertex és Pixel (Fragment) shadereket hozzanak létre, ezáltal a megjelenítési pipeline ezen részét módosíthatóvá tette. [16]

A DirectX 10.0 verziója vezette be a Geometry Shadert, ami által a programozó képes volt újabb geometriákat létrehozni már megadottakból. Ezután a DirectX 11 tette lehetővé a Tessallation shaderek testreszabását, ami által a fejlesztőknek lehetőségük nyílt a felületek és geometriák felbontásának növelésére a GPU-n keresztül. [16]

A DirectX 12 2015-ben jelent meg, ami újabb programozható elemet nem adott hozzá a megjelenítési pipelinehoz, viszont sokkal nagyobb szabadságot adott a fejlesztőknek azáltal, hogy alacsonyabb szinten érhetik el az erőforrásokat. Ezen felül a DXR komponens is bekerült a DirectX 12-be, ami egy teljesen új megjelenítési pipeline-t és shadereket tartalmaz a ray tracing megjelenítéshez. [16]

A megvalósítás során a DirectX 12 verzió került használatra a DXR komponenst felhasználva.

4.2 Raszterizálás és Ray Tracing folyamatainak összehasonlítása



4.1. ábra: Általános Rendering Pipeline raszterizálás esetében [13]

Az 4.1 ábrán látható, hogy a raszterizálás lépései a következőek:

– Vertex Shader
– Tessalation Shaders
– Geometry Shader
– Raszterizálás
– Pixel/Fragment Shader
– Kimenet

Tehát először a Vertex Shader kerül végrehajtásra, aminek a bemenete a pontok halmaza, tehát a vertexek, amiket megszeretnénk jeleníteni a képernyőn.

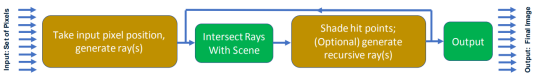
Ezek után következik a tesszallációs szakasz, amiben lehetősége van a fejlesztőnek az adott geometriát további geometriákra lehet bontani és ezáltal növelni a részletességet és felbontást.

A Geometry Shader lehetőséget ad további geometriák létrehozására a már meglévőkből.

Ezen folyamatok után következik a raszterizálás ami nem testreszabható, ez egy beépített dedikált része a pipelineknak.

Végezetül a Pixel Shader határozza meg a kimenetet, tehát azt, hogy melyik pixel pozícióban milyen színű pixel jelenjen meg.

Ezzel összehasonlítva a Ray Tracing pipeline a következő:



4.2. ábra: Ray Tracing Pipeline [13]

A 4.2 egy leegyszerűsített ábra a DXR szakaszairól. Az első rész párhuzamosan zajlik pixelenként, tehát a bemenete az összes pixel az adott felbontáson.

A következő lépésben az elsődleges sugarak generálása következik. Egy elsődleges sugár egyszer kerül létrehozásra pixelenként.

Minden egyes elsődleges sugár esetében következik a metszéspont vizsgálat a jelenet-

ben szereplő objektumokkal. Majd ezután, találat esetén következik az árnyék meghatározása, tehát a másodlagos sugarak kiküldése, attól függően milyen módszer van használatban. Ez attól függően történik, hogy erős vagy lágy árnyékokkal működik a program. Ugyanebben a szakaszban kerülnek kiküldésre a rekurzív sugarak amennyiben szükségesek és ez egyben jelenti azt is, hogy ez a rekurzív szakasz, ugyanis ekkor újra következik a metszéspont vizsgálat.

Ez a folyamat addig tart amíg elért a rekurzió egy bizonyos mélységet, vagy nincs több metszéspont találat az adott sugárhoz. Ezek után egy formula alapján meghatározható az adott pixel színe, ami a kimenete a folyamatnak.

Megfigyelhető, hogy a raszterizálás és a ray tracing megjelenítési folyamat teljes egészében különbözik egymástól.

4.3 DXR

A DXR bevezetett egy új pipeline-t valamint hat új shader típust. Ezeknek mind külön szerepe van a pipelineban.

4.3.1 DXR Shader Típusai

A DXR bevezetett hat új shader típust, amik az új pipelineban kerülnek felhasználásra.

- Ray Generation Shader
- Intersection Shader
- Miss Shader
- Closes-hit Shader
- Any-hit Shader
- Callable Shader

Az első a Ray Generation Shader, ami felelős az elsődleges sugarak definiálásáért. Ebből a shader típusból csak egy lehet, és a legtöbb esetben ebben kerül meghatározásra, hogy a sugarak honnan és milyen irányba indulnak. A legtöbb esetben sugarak kiindulópontja a kamera pozíciója és a megjelenítendő felbontás alapján minden egyes pixelen keresztül megy egy-egy sugár.

A második az Intersection Shader, ami a metszéspont vizsgálatért felel. Ebből több is lehet attól függően, hogy hány fajta geometria szerepel a jelenetben, vagy mennyit támogat a szoftver. Alapértelmezetten a háromszög metszéspontvizsgálata a sugarakkal már implementálva van a DirectX 12-ben.

A Miss Shader akkor kerül meghívásra, amikor a sugár nem talált metszéspontot. Ebből szintén több lehet, attól függően, hogy mennyi sugár típus szerepel a szoftverben pl.: elsődleges, másodlagos, rekurzív, stb.

A Closest-hit Shader akkor kerül meghívásra amikor megvan a találat és a legközelebbi metszéspont is egyben. Ebben a szakaszban kerül tehát meghatározásra valamilyen árnyalat vagy szín érték. Ebből a shader típusból szintén több is lehet sugár típusoktól függően.

Az Any-hit Shader minden egyes találatkor meghívódik és a legtöbb esetben arra kerül felhasználásra, hogy az áttetszőséget meghatározza. Ez azt jelenti, hogyha esetleg az objektumhoz tartozik egy texture, ami áttetsző, akkor azon a sugárnak át kell haladnia, viszont az Intersection Shader metszéspontot talál a megadott geometriához. Ez esetben a metszéspont találat az Any-hit shaderben ignorálható és folytatható tovább a folyamat, ugyanis ez nem számítható megfelelő találatnak, hiszen az objektum átlátszó.

4.3.2 HLSL Függvények

DXR bevezetett néhány új elérhető függvényt a HLSL-ben amik felhasználhatóak a shaderekben.

Az első ilyen a TraceRay függvény, ami egy új sugarakat képes indítani. Ez meghívható a Ray Generation Shaderből illetve a Closest vagy a Miss shaderből. Legtöbb esetben az elsődleges sugarak kiküldésénél van szerepe, majd a rekurzív sugarakéban.

A ReportHit függvény a metszéspont találatkor kerülhet meghívásra, ezáltal csak az Intersection Shaderekben érhető el. Ez a függvény jelzi, hogy az adott shader metszéspontot talált.

Az IgnoreHit függvény az Any-hit shaderből hívható meg és lehetőséget ad az adott

találat ignorálására, tehát a bejárás folytatódhat tovább és nem kell elmenteni a találatot.

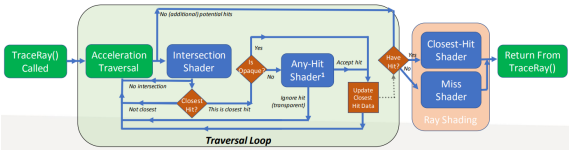
A AcceptHitAndEndSearch függvény pedig ennek az ellenkezője. Elfogadja az adott találatot és befejezi a keresést. Szintén csak az Any-hit shaderből hívható meg.

Ezeken kívül vannak olyan függvények amikkel lekérdezhető az adott folyamat egy-egy állapota. Ilyen a DispatchRaysDimension, ami megadja hogy milyen felbontáson fut az adott végrehajtás, mennyi elsődleges sugár került kiküldésre (pl.: 1280 x 720). A másik ilyen függvény a DispatchRayIndex, ami megadja, hogy az adott folyamatban melyik sugár vizsgálata van éppen soron. Ezen metódusok bármelyik shaderből meghívhatóak.

Az adott találattal kapcsolatban is lehet lekérdezéseket végezni, ilyen a HitKind függvény, ami információval szolgál a találatról.

Végül vannak metódusok amik a sugárral kapcsolatban szolgáltatnak információt, mint annak az aktuális hossza illetve minimum hossza, origója és az iránya.

4.3.3 Folyamat bemutatása



4.3. ábra: DXR Részletes Folyamata [13]

A 4.3 ábrán látható a részletes folyamatábrája a vizsgálatnak.

A folyamat onnan kezdődik, hogy az első TraceRay meghívásra került a Ray Generation Shaderből, tehát az elsődleges sugár kiküldésre került.

Ekkor kezdődik egy rekurzív végrehajtás. Ebben az első lépés a gyorsítási struktúra végigjárása. Ez a struktúra segíti a metszéspont gyorsabb megtalálását, valamilyen módszer szerint. Ha találatot talált ebben a szerkezetben, akkor meghívásra kerülnek a megadott Intersection Shaderek a geometriák további vizsgálatára, amik a szerkezeten belül helyezkednek el.

Az Intersection Shaderben meghatározásra kerül, hogy az adott sugár mely objektummal ütközik, melyiket metszi. Ha metszéspontot talál, akkor megnézi, hogy ez a metszéspont volt-e a legközelebbi. Ha ez a legközelebbi találat, akkor még nem tárolja el, hanem továbbküldi az Any-hit shadernek. Az Any-hit shader megvizsgálja, hogy az adott objektum áttetsző-e. Ha igen, akkor ignorálja a találatot és visszalép a vizsgálatra, hogy folytassa a gyorsítási struktúra bejárását. Ha pedig nem áttetsző az objektum, akkor elfogadja a találatot és frissíti az eltárolt legközelebbi metszéspont találatot.

Ha az Intersection Shader nem jelez találatot vagy az nem a legközelebbi, akkor szintén visszalép a struktúra bejárásához és folytatja azt.

Ez az iteráció akkor ér véget, ha minden potenciális találatot megvizsgált az algoritmus és nincs több elem a gyorsító struktúrában. Ekkor megnézi a rendszer, hogy volt-e találat. Ha volt találat akkor a Closest-hit shader kerül meghívásra, ellenkező esetben a Miss shader.

Ezután pedig a TraceRay visszatér a meghatározott értékkel.

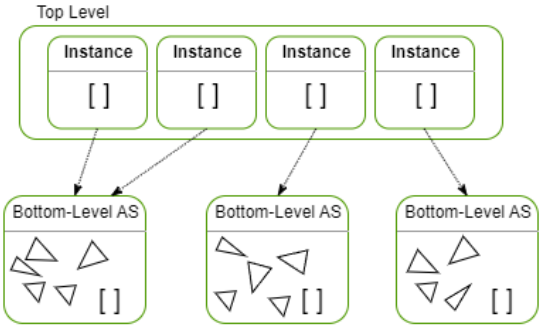
4.3.4 Gyorsítási struktúra

Annak érdekében, hogy ne kelljen a teljes objektum hierarchián végigmenni, lehetőségünk van gyorsítási struktúrákat létrehozni. Ezek olyan adatszerkezetek, amik segítik a keresést. Ez szinte bármilyen szerkezet lehet, a DirectX 12 lehetőséget ad a fejlesztőknek arra, hogy maguk hozzák létre ezen szerkezeteket.

A DirectX természetesen rendelkezik ennek támogatásához szolgáló eszközzel. A szerkezetek létrehozásánál két szintet definiál, Bottom-Level AS és Top-Level AS, tehát alacsony szintű gyorsítási szerkezeteket és magas szintű gyorsítási szerkezeteket.

Az alacsony szintű szerkezet tárolja magát az objektumokat. Ez azt jelenti, hogy ezekben a legtöbb esetben maguk a geometriák vannak, amik lehetnek komplex modellek, gömbök, téglatestek, stb.

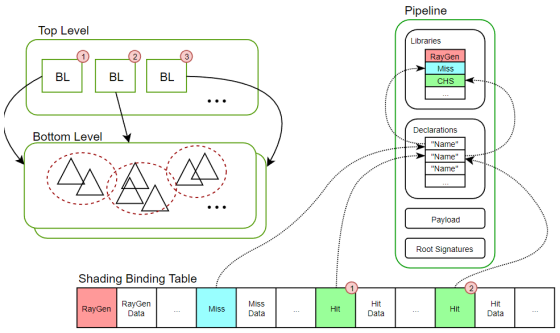
A magas szintű gyorsítási szerkezet pedig úgy működik mint egy pointer ami egy ala-



4.4. ábra: Alacsony és Magas szintű gyorsítási szerkezetek [13]

csony szintű szerkezetre mutat. Tehát nem tárol tényleges geometriákat. Ennek az előnye az, hogy lehetővé teszi ugyanabból az alacsonyabb szintű szerkezetből akár több példány létrehozását az adott jelenetben anélkül, hogy az objektumot kétszer vagy többször tárolnánk a GPU memóriában. A különböző példányokhoz pedig egy transzformációs mátrix rendelhető, ezáltal a pozíciójuk, méretük és elforgatásuk meghatározható.

Ezen építőelemekből biztosítja a DirectX szinte bármilyen típusú adatstruktúra felépítését. A egyik legelterjedtebb jelenleg a BVH, amiben téglatesteket definiálnak egy fa struktúrában és ez kerül bejárásra.

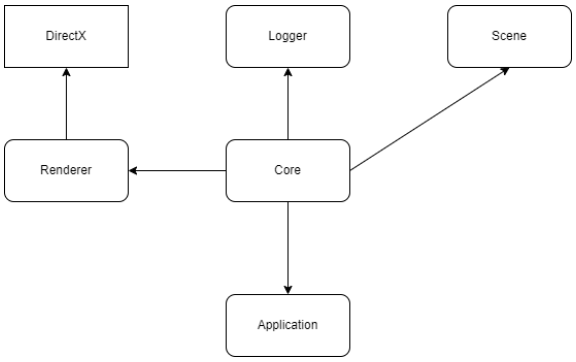


4.5. ábra: DXR Pipeline és Gyorsítási Struktúrák [13]

4.4 Implementáció

Az implementáció során a korábban említett eszközök kerültek felhasználásra, egy C++ programban DirectX 12 használatával, Windows 10 fejlesztőkörnyezetben.

Az implementáció során 5 fő komponens került létrehozásra amik a szoftver különbö-



4.6. ábra: Megvalósított szoftver osztályai

ző elemeit kezelik. Ezek a Core, Renderer, Logger, Scene és az Application osztályok.

4.5 Core

A Core osztály felelős az különböző komponensek összekapcsolásáért. Példányosítja a szükséges osztályok és feloldja a függőségeket. A Core határozza meg az adott jelenetet is, melyet átad a megfelelő Scene példánynak betöltésre.

Ezen felül a Core osztály felelős az ablak létrehozásáért Windowsban, tehát kommunikál a Windows kernellel, tárolja a létrehozott ablak méretét és az összes üzenetet kezeli. Tehát kezeli a HWND üzeneteket és továbbítja a megfelelően feldolgozott felhasználói bemenetet a többi osztálynak.

4.5.1 Scene

A Scene osztály felelős az adott jelenethez tartozó objektumok tárolásáért a CPU oldalon. Ezen felül a fájlokat is ez az osztály kezeli, amik a jelenetet tárolják.

4.5.2 Renderer

Az egyik legfontosabb eleme a szoftvernek a Renderer osztály. Ez az osztály felelős a megjelenítésért és a DirectX -el való kommunikációért. Tartalmazza az összes DirectX objektumot, létrehozza a szükséges DirectX objektumokat és nyomon követi azokat (Device, DescriptorHeap, Adapter, Resource-k). Ezen kívül a shadereknek való adatküldésért is felel valamint azok beolvasásáért.

4.5.3 Application

Az Application osztály egy példa osztály ami a Core-t használja fel, hogy kommunikáljon a rendszerrel. Ez nem része a programnak. Egy interfészen keresztül képes a Core-n keresztül definiálni a szükséges objektumokat, létrehozni az ablakot a megjelenítéshez és meghatározni, hogy mi jelenjen meg az adott jelenetben.

4.5.4 Felhasznált shaderek

A mérésekhez használt jelenethez szükség volt az alábbi shaderekre:

- Ray Generation Shader - Az elsődleges sugarak generálásához
- Intersection Shader - Vizsgálja az ütközést a gyorsítási struktrúával, illetve gömbbel
- Miss Shader - Elsődleges sugarakhoz
- Miss Shader - Másodlagos sugarakhoz
- Closes-hit Shader - Háromszögek esetén
- Closes-hit Shader - Minden más esetben

Tehát kezdetben a Ray Generation Shader indítja el az elsődleges sugarakat. Ezt úgy teszi, hogy minden egyes pixelnek a közepén keresztül küldi keresztül a sugarakat. A sugár kezdőpontja a kamera középpontja, az iránya pedig a kamera középpontjától az adott pixel közepe felé meghatározott irány. A sugarak minden pixel középpontján mennek keresztül.

A kamerához több adatot is tárolunk. Először a kamera pozícióját a térben, ez az úgynevezett "eye". Ezen kívül egy "up" vektort ami meghatározza a fel irányt a kamerához. Ezen kívül van egy "center/at" adat, ami egy vektor és meghatározza, hogy merre néz a kamera.

Ezek után a Miss shaderek kerülnek definiálásra. Az első miss shader egyszerűen csak a háttérszínt adja vissza, ami a teszt jelenetben egy egyszerű halványkék szín, ami az eget reprezentálja.

A másik Miss shader a másodlagos sugarakhoz tartozik, ez esetben ennek a visszatérési értéke csak egy "false" boolean érték, ami azt jelzi, hogy nem volt találat.

Az applikáció egyetlen Intersection Shadert tartalmaz. Ennek az oka az, hogy a DirectX alapértelmezetten tartalmaz egy Intersection Shadert a háromszögekre, ami az egyik leggyakoribb elem a modellekben. Ezáltal ezt nem volt szükséges létrehozni. Viszont két típusú primitívet szükséges vizsgálni, a téglatesteket és a gömböket, de ezek egyetlen Intersection Shaderben kerülnek vizsgálatra, attól függően, hogy az aktuális objektum milyen primitív.

A teszt jelenetben 3 típusú objektum szerepel, háromszög, téglatest, gömb. A téglatest a gyorsítási struktúrához kell, ugyanis abban kerülnek tárolásra a primitívek.

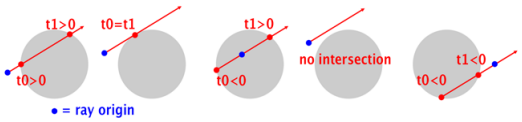
A sugár és a gömb metszéspontjának vizsgálatához adott a következő egyenlet:

$$(P - C) \cdot (P - C) - r^2 = 0 \tag{4.1}$$

Ahol a P az adott sugár, C (center) a gömb közepe, valamint r a sugara.

$$P = P_o + P_d t \tag{4.2}$$

Ahol t a sugár hossza,P_o a sugár kezdőpontja, P_d pedig az iránya. Ezután ezeket behelyettesítve a 4.1 egyenletbe, meg kell oldani az egyenletet t -re és megkapjuk a metszéspont vizsgálat eredményét.



4.7. ábra: Gömb metszéspontvizsgálat esetei [17]

Az eredmény esetei a 4.7 ábrán láthatóak. Tehát az egyenlet megoldásához egy másodfokú egyenletet szükséges megoldani. Ennek az eredményei következőek lehetnek:

- Két pozitív gyöke van - A sugár keresztül megy a gömbön
- Két gyöke egyenlő - A sugár érintője a gömbnek
- Egy pozitív és egy negatív gyöke van - a sugár a gombon belül van
- Komplex gyöke van - nincs metszéspont

Ez azt jelenti, hogy először érdemes a determinánst vizsgálni, ugyanis ezzel gyorsítható a folyamat, illetve elkerülhetőek az esetleges problémák.

Ha az első eset áll fent, és két pozitív gyöke van az egyenletnek, akkor a közelebbi találatot kell kiválasztani, tehát a kisebbet. Ha a második eset áll fent akkor csak egy találat van, hiszen érintője a sugár a gömbnek. Ha egy pozitív és egy negatív gyöke van, akkor a pozitívat kell választani, ugyanis a negatív a negatív távolságot jelenti, ami egyben a negatív irányt is jelenti, ami ellentétes a sugár irányával.

Végezetül két Closes-hit shader került definiálásra, az egyik a háromszögekre, a másik pedig a téglatest és gömb objektumokra. Mindkét shader a végső szín érték beállításáért felel úgy, hogy a rekurzív sugarakat is kiküldik majd azok eredményeiből kapják meg a végső színértéket. A szín meghatározásnál Phong formula által kerül kiszámításra a színérték.

$$I_p = K_a + K_e + \Sigma_{i=1}^{L_n} V_i L_i (K_d max(I_i \dot{n}, 0) + K_s (max(h_i \dot{n}, 0))^s) + K_S I_R + K_T I_T \tag{4.3}$$

Ahol:

- I_p - Az adott pixel intenzitása, színe
- K_a - Háttérfényforrás
- K_e - Kibocsátott fény
- L_n - Fényforrások száma
- L_i - Aktuálisan vizsgált fényforrás (színcsatornánként)
- n - Felület normálja
- V_i - Fényforrás láthatósága
- K_d - Az objektum színe
- K_s - A tükröződés
- h - Félszög
- s - A ragyogás mértéke
- K_S - Fényvisszaverődés mértéke
- I_R - Rekurzívan megkapott intenzitás a visszaverődésekből
- K_T - Fénytörés mértéke
- I_T - Rekurzívan megkapott intenzitás a fénytörésekből

4.5.5 Transzformált elemek vizsgálata

Az esetek nagy részében olyan objektumokon kell metszéspont vizsgálatot végezni amelyek valamilyen transzformáción átestek. Ilyen transzformáció lehet az eltolás, forgatás és skálázás. Azonban bizonyos primitívek a transzformációk során megváltozhatnak, olyannyira, hogy a vizsgálat nem működik, mert már nem olyan a felépítésük mint eredetileg volt.

Ebben az esetben ez történhet a gömbbel, ugyanis ha egy gömb skálázásra kerül valamilyen irányba nem egységesen, akkor már egy teljesen más primitívet kaphatunk és nem használható rajta a gömb-sugár metszéspont vizsgálat, hiszen az objektum már nem tekinthető gömbnek a továbbiakban.

Általános esetben az történik, hogy a transzformációs mátrix-al beszorzásra kerül az adott objektum majd megvizsgálásra kerül, hogy a sugár ütközik-e vele.

Azonban ebben az esetben egy univerzális megoldás az lehet, ha az adott transzformáció nem az adott primitíven kerül az végrehajtásra, hanem annak a transzformációnak az inverével kerül beszorzásra a sugár, majd kerül viszgálatra a módosított sugár az eredeti primitívvel. Ennek az eredménye egy pont, ha van találat, amin újból elvégzésre kerül a transzformáció, tehát beszorzásra kerül a transzformációs mátrix-al, ami megadja a végleges metszéspontot. Ez esetben nem számít, hogy milyen transzformáció került elvégzésre az objektumon.

5 VÉGEREDMÉNY

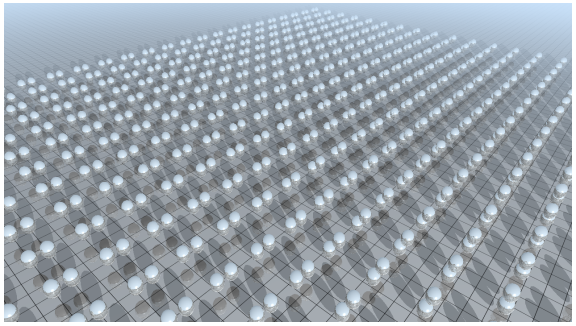
5.1 Teszt jelenet

A DirectX 12 lehetőségeit felhasználva egy jelenet került létrehozásra az elkészült applikációban.

A teszt jelenetben 1000 db gömb objektum szerepel. Gyorsítási struktúra alacsony szintjén téglatest szerkezetek szerepelnek, egy szerkezet 2 gömböt tartalmaz, ez azt jelenti, hogy a jelenet 500 téglatestet tartalmaz, mindegyikben két gömbbel.

A gömbök azonos anyagtulajdonsággal rendelkeznek ami tükröződő és ezüstös színnel rendelkezik.

A jelenet alján egy szintén reflektív textúrázott sík objektum helyezkedik el, ami egy rácsozott textúrát tartalmaz.



5.1. ábra: A tesztelt jelenet

A jelenet során felhasználásra kerültek a másodlagos sugarak illetve a rekurzív sugarak is, különböző mélységekkel. A jelenet egy pont fényt tartalmaz.

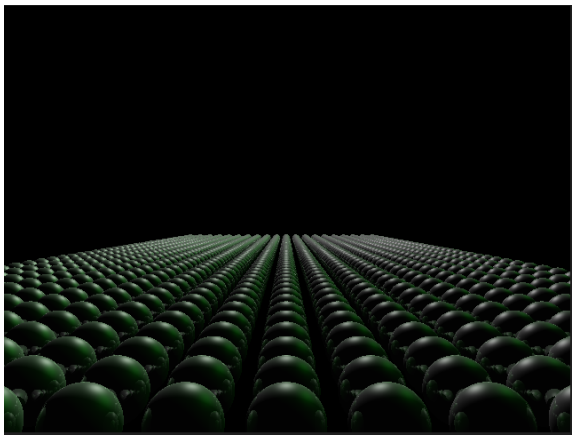
5.2 Mérések

5.2.1 CPU-n végzett mérés

A kezdeti applikációban egy CPU-n egy szálon végrehajtott ray tracing megoldással végeztem méréseket. A jelenetek forrása Ravi Ramamoorthi Advanced Computer Graphics

oktatási anyagából kerültek felhasználásra, ahol egy egyedi utasítás alapú fájl szerkezetben kerültek definiálásra a jelenetek. [1]

Ebben található egy jelenet, ami szintén 1000 gömböt tartalmaz. Ehhez a jelenethez szükséges a legtöbb erőforrás a minták közül, mivel a gömbök közel helyezkednek el egymáshoz és rengeteg objektumot kell megvizsgálni. A CPU-n végzett teszt applikációban a formula ugyanaz volt mint a végső alkalmazásban, valamint 5 mélységgel rendelkező rekurzív sugarakat használtam. A megjelenítési idő ez esetben: 1700000 ms, ami ~ 28 perc és 20 másodperc.



5.2. ábra: CPU jelenet [1]

A jelenet felbontása 640 x 480, ahol a háttér teljesen fekete. Egy pont és egy directional megvilágítást tartalmaz, illetve a kamera fix, nem mozgatható.

5.2.2 GPU-n végzett mérés

A 5.1 ábrán látható a GPU-n végzett jelenet. Több paraméterrel is mérésre került, 3 és 5 mélységgel rendelkező rekurzív sugarakkal, illetve 1280 x 768 -as felbontáson.

A kamera mozgatható, emiatt a kamera pozíciójától is függ, hogy milyen megjelenítési idővel rendelkezik az alkalmazás, valamint az átlag érték természetesen függ a mérés idejétől is. Emiatt a méréseket közel azonos pozíciókban végeztem és közel azonos ideig kerültek rögzítésre a különböző kamera pozíciók. Végezetül a minimum értékeket és az átlag értékeket hasonlítottam össze.

Rekurzív sugarak száma	Minimum FPS	Átlag FPS
3	27,8	36,2
5	14,3	28,1

5.1. táblázat: Mérési eredmények a GPU-n

A mérések a következő architektúrán kerültek rögzítésre:

- CPU: Intel(R) Core(TM) i5-7600K CPU @ 3.80GHz
- GPU: NVIDIA GeForce GTX 1060 6GB
- Platform: Windows 10

5.3 Továbbfejlesztési lehetőségek

Az alkalmazás jelenleg három típusú primitívet támogat, háromszögeket, téglatesteket és gömböket. Annak érdekében, hogy a program minél több területen legyen alkalmazható, ez bővíthető az Intersection Shaderek számának növelésével.

Különböző 3D modell formátumok támogatása is fontos lehet a későbbiekben, mint a .fbx vagy a .gltf. Jelenleg ezen formátumok a legismertebbek játékiparban, ezáltal ezen modellformátumokból betöltött modellek támogatása rengeteg lehetőségek nyit meg, ugyanis több modell érhető el és könnyebben használható.

A gyorsítási struktúrák létrehozása is történhet a későbbiekben automatikusan illetve manuálisan. Egy megoldás lehet, hogy a felhasználónak lehetősége legyen egy szerkesztőben a gyorsítási szerkezethez megfelelő primitívet definiálni az adott modellhez, pl.: egy téglatestek definiálása majd megjelölése, hogy mely primitívek tartoznak hozzá, mely primitíveket foglalja magába. Ezáltal többletmunkával, de teljesen testreszabhatóan lehet a gyorsítási struktúrát létrehozni.

Ezek mellett nem csak a valós idejű megjelenítésben alkalmazható a Ray Tracing, ugyanis segíthet bizonyos előfeldolgozásokat is elvégezhet amik segítenek a raszterizálást felhasznált megjelenítésben. Ilyen lehet például a fényvisszaverődések és tükröződések előre eltárolásra fájlokban, amiket később a raszterizálás során a Pixel Shaderben képesek felhasználni a fejlesztők.

Végezetül a DirectX API biztosítja a ray tracing és raszterizálás egy alkalmazáson

belüli használatát is, ezáltal egy újabb lehetőséget biztosít arra, hogy bizonyos részeket raszterizálással, másokat pedig ray tracinggel jelenítsünk meg, ami gyorsabb futási időt eredményezhet.

6 DIPLOMAMUNKA TARTALMI ÖSSZEFOGLALÓJA

6.1 A diplomamunka magyar nyelvű kivonata

A diplomamunkám során a feladatom a ray tracing technológia részletes bemutatása volt, majd egy alkalmazás létrehozása ami képes 3D jelenetek megjelenítésére ray tracing alapon úgy, hogy ehhez a GPU-t használja.

A diplomamunkámat a technológia bemutatásával kezdtem és annak a fejlődéstörténetével, valamint, hogy mely területeken került használatra a fejlődése során. Ezen keresztül bemutattam a technológia lépéseit valamint azt, hogy milyen sajátosságai vannak ennek a megjelenítési módszernek.

A bemutatás után következett irodalomkutatás, amiben tanulmányoztam már meglévő megoldásokat és hasonló projekteket. Ehhez megvizsgáltam néhány videójáték-motort, illetve néhány kutatási projektek és különösképpen az általuk megvalósított megjelenítési eljárásokat. Majd ezen projektek és szoftverek megoldásait összehasonlítottam.

A diplomamunkámat a tervezéssel folytattam, ahol kezdetben bemutattam a raszterizálás és ray tracing folyamatai közötti különbségek. Majd a ray tracing lépéseinek részletes bemutatásával folytattam.

A tervezés végén létrehoztam egy alkalmazást ami a ray tracing eljárást használva képes háromdimenziós jelenetek megjelenítésére úgy, hogy a számításokat a CPU-n végzi.

A megvalósítás során ismertettem, hogy mely technológiát fogom használni az alkalmazásfejlesztés során, ahol a DirectX 12 verziójára esett a választásom. Ezután részletesen ismertettem az eszköznek a sajátosságait, hogy milyen folyamat szerint működik és milyen személyreszabható elemei vannak, valamint a különbséget a hagyományos megjelenítési módszerekkel szemben. Ezek után következett az alkalmazás implementálása.

A megvalósítás után méréseket végeztem, ahol összehasonlítottam a CPU, valamint a GPU által végzett megjelenítési eredményeket. Ezen felül a GPU mérések elvégzésénél több beállítást is megvizsgáltam amiket szintén összehasonlítottam.

Végezetül ismertettem néhány továbbfejlesztési lehetőséget, amikkel a későbbiekben

kiegészíthető az alkalmazás.

6.2 A diplomamunka angol nyelvű kivonata

My task during my thesis was to introduce ray tracing technology in detail and create an application that can render three-dimensional scenes on the screen, using the GPU for that.

A started my thesis with the introduction of ray tracing technology, and also its development history. Through this, I showed the steps of the technology and the features of this rendering method.

After the introduction, I continued with the literature research, in which I studied existing researches and softwares. I examined some game engines, because that's the field, where real time ray tracing is the most important, and I also introduced some researches on this topic. I mainly focused to show how these softwares solved the rendering with ray tracing.

After that, I continued with the designing part. In this section, I showed the differences between rasterisation and ray tracing and the steps of ray tracing in general.

At the end of designing, I implemented a small application, which could render 3D scenes using the CPU.

After the designing section, I started the implementation. During the implementation, I decided to use DirectX 12 and I also introduced its features and processes. This section also contains a comparison between traditional rendering techniques and ray tracing. After this, I implemented the software.

After the implementation, I performed measurements, where I compared the rendering results of the CPU and GPU. In addition, I measured the GPU solution with different configurations and showed the results in a table.

Finally, I described some possible future developments.

7 IRODALOMJEGYZÉK

[1] R. Ramamoorthi. „Advanced Computer Graphics”. (2010), cím: https://inst.eecs.berkeley.edu/~cs283/fa10/lectures/283-lecture2.pdf (elérés dátuma 2022. 12. 05.).

[2] E. R. Freniere és J. Tourtellott, „Brief history of generalized ray tracing”, *Lens Design, Illumination, and Optomechanical Modeling*, SPIE, 3130. köt., 1997, 170–178. old.

[3] SideFX. „How to render using Mantra”. (2021. okt. 27.), cím: https://www.sidefx.com/docs/houdini/render/render.html (elérés dátuma 2022. 12. 05.).

[4] U. Engine. „Unreal Engine”. (2022. ápr.), cím: https://www.unrealengine.com/ (elérés dátuma 2022. 12. 05.).

[5] U. Engine. „Unreal Engine Ray Tracing”. (2022. ápr.), cím: https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/RayTracing/ (elérés dátuma 2022. 12. 05.).

[6] B. C. Budge, J. C. Anderson, C. Garth és K. I. Joy, „A straightforward CUDA implementation for interactive ray-tracing”, *2008 IEEE Symposium on Interactive Ray Tracing*, IEEE, 2008, 178–178. old.

[7] E. Peckham. „Unity Története”. (2019. okt. 17.), cím: https://techcrunch.com/2019/10/17/how-unity-built-the-worlds-most-popular-game-engine/ (elérés dátuma 2022. 12. 05.).

[8] Unity. „Ray Tracing Unityban”. (2022), cím: https://unity.com/ray-tracing (elérés dátuma 2022. 12. 05.).

[9] Wikipedia. „CryEngine Történeten”. (2022), cím: https://en.wikipedia.org/wiki/CryEngine (elérés dátuma 2022. 12. 05.).

[10] CryEngine. „CryEngione Neon Noir”. (2022), cím: https://www.cryengine.com/news/view/how-we-made-neon-noir-ray-traced-reflections-in-cryengine-and-more (elérés dátuma 2022. 12. 05.).

[11] Scratchapixel. „Raszterizálás”. (2022), cím: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation (elérés dátuma 2022. 12. 05.).

[12] Khronos. „Rendering Pipeline”. (2022), cím: https://www.khronos.org/opengl/wiki/Rendering%5C_Pipeline%5C_Overview (elérés dátuma 2022. 12. 05.).

[13] Nvidia. „Ray Tracing Alapok”. (2022), cím: https://developer.nvidia.com/discover/ray-tracing (elérés dátuma 2022. 12. 05.).

[14] Scratchapixel. „Rekurzív Sugarak”. (2022), cím: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel (elérés dátuma 2022. 12. 05.).

[15] K. Beets. „Gyorsítási Struktúrák”. (2020. jún. 4.), cím: https://www.embedded.com/what-is-ray-tracing-and-how-is-it-enabling-real-time-3d-graphics/ (elérés dátuma 2022. 12. 05.).

[16] Jeremiah. „Learning DirectX 12”. (2017. dec. 14.), cím: https://www.3dgep.com/learning-directx-12-1/#cite-14 (elérés dátuma 2022. 12. 05.).

[17] Scratchapixel. „A Minimal Ray-Tracer: Rendering Simple Shapes”. (2020), cím: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection (elérés dátuma 2022. 12. 05.).

8 ÁBRAJEGYZÉK

9 TÁBLAJEGYZÉK

10 RÖVIDÍTÉSEK

GPU	Graphics Processing Unit
GGPU	General-Purpose Computing on Graphics Processing units
PBR	Physically Based Rendering
API	Application Programming Interface
SDK	Software Development Kit
DXR	DirectX Raytracing
HLSL	High-level shader language
AABB	Axis aligned bounding box
AS	Acceleration Structure
BVH	Bounding Volume Hierarchy