MŰEGYETEM 1782

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Varga Levente

HIBRID RENDERELÉS A UNITY JÁTÉKMOTORBAN

KONZULENS

Dr. Magdics Milán

BUDAPEST, 2021

Tartalomjegyzék

Összefoglaló ... 6
Abstract ... 7
1 Bevezetés ... 8
1.1 Sugárkövetés ... 8
1.1.1 Sugarak indítása ... 8
1.1.2 Rekurzió ... 9
1.2 Sugárkövetési modellek ... 10
1.2.1 Whitted ... 10
1.2.2 Cook ... 11
1.3 Inkrementális képszintézis ... 11
1.3.1 Transzformációk ... 11
1.3.2 Screen space sugárkövetés ... 13
2 DirectX ... 14
2.1 Árnyalók ... 14
2.2 DirectX csővezetékek ... 14
2.2.1 Raszter csővezeték ... 15
2.2.2 RayTracing csővezeték ... 16
3 Unity játékmotor ... 18
3.1 Unity alapok ... 18
3.2 Unity csővezetékek ... 19
3.2.1 Built-in Render Pipeline ... 19
3.2.2 Universal Render Pipeline ... 19
3.2.3 High Definition Render Pipeline ... 19
3.2.4 Scriptable Render Pipeline ... 20
3.3 Shader Object ... 20
4 Fényeffektusok Unity-ben ... 22
4.1 Árnyékok ... 22
4.2 Contact Shadows ... 24
4.3 Tükröződés ... 24
4.4 Törés ... 25
4.5 Ambient Occlusion ... 26

4.6 Global Illumination ... 27
4.7 Subsurface Scattering ... 29
5 Raytracing a Unity-ben ... 31
5.1 Volume Override ... 31
5.2 DirectX API alapjai ... 32
5.2.1 SceneAccelerationStructure ... 32
5.2.2 RayPayload ... 33
5.2.3 RayDesc ... 33
5.2.4 AttributeData ... 33
5.2.5 TraceRay() ... 34
5.2.6 Raygeneration Shader ... 34
5.2.7 Miss Shader ... 35
5.2.8 Closest-Hit Shader ... 36
5.2.9 Any-Hit Shader ... 36
5.3 Sugárkövető árnyaló kezelése C# kódból ... 36
6 Törésszámítás sugárkövetéssel ... 38
6.1 Baricentrikus koordináták ... 39
6.2 Normálvektor kiszámítása ... 39
6.3 Törésirány ... 40
6.4 Új sugár indítása ... 41
6.5 Egyszerű anyag ... 42
6.6 Eredmények ... 42
7 Fényeffektek tesztelése ... 44
7.1 Ambient Occlusion ... 45
7.2 Tükröződés ... 46
7.2.1 Tükröződés hengerben ... 47
7.3 Törés ... 48
7.4 Global Illumination ... 49
7.5 Árnyékok ... 50
7.6 Contact shadow ... 51
7.7 Subsurface Scattering ... 51
7.8 Következtetés ... 52
8 Deep Learning Super Sampling (DLSS) ... 54
8.1 DLSS Unity-ben ... 54

8.2 DLSS tesztek ... 55
9 Demonstráció ... 56
10 Összegzés ... 58
11 Irodalomjegyzék ... 59

HALLGATÓI NYILATKOZAT

Alulírott **Varga Levente**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2021. 12. 10.

...…………………………………………….

Varga Levente

Összefoglaló

A számítógépes játékok megjelenése óta rendszeresen felmerülő kérdés, hogy a grafikus megjelenítés minőségét, valósághűségét meddig lehet fokozni a teljesítmény megtartása mellett. Hol húzódik a határ a látvány és a képkockaszám (frame per second, FPS) között? Az elmúlt évtizedekben számos megoldás született a különböző fényeffektek realisztikus reprezentációjára, melyek egyre jobban megközelítették a háromdimenziós (3D) modellek renderelésénél használt, a valós fényterjedést modellező sugárkövetéses technikákat.

2018-ban az Nvidia olyan saját gyártású videokártya sorozatot mutatott be, melyek hardveresen támogatják a sugárkövetést, ezzel olyan gyorssá és hatékonnyá téve a technológiát, hogy az már alkalmas lett az átlagember számítógépében is valósághű jelenetek megjelenítésére és valós idejű alkalmazások megjelenítésére magas FPS szám megtartása mellett. Ezzel egy új korszak kezdődött a valós idejű renderelés területén.

A dolgozatomban ismertetem a korábbi, raszteres megjelenítés és az új, sugárkövetéses technika közötti különbségeket és kompromisszumokat, bemutatom a Unity Engine-ben nemrég megjelent DirectX 12 RayTracing (DXR) technológián alapuló sugárkövetést, megvalósítok egy natívan nem támogatott fényeffektust High Level Shading Language (HLSL) segítségével, majd egy erre alkalmas jelenetben demonstrálom az eredményeket.

A dolgozat célja, hogy egy átfogó képet alkosson a Unity jelenlegi képességeiről a raszteres és sugárkövetéses megjelenítés területén.

Abstract

Since the invention of modern computers, we have always been pushing the limits of computer-generated imagery while keeping an eye on the performance as well, and have been pondering the question: where does the boundary between visual quality and framerate per second (FPS) lie? Numerous solutions have been introduced in the last few decades for computing realistic light effects in real time, which started to approach the quality of ray tracing techniques used in traditional three-dimensional (3D) rendering.

In 2018, Nvidia introduced a new line-up of video cards with hardware-implemented ray tracing support. This move made the technology so efficient and powerful in the average user's personal computer, it did not only speed up the rendering process, but also allowed raytracing to be used in real time applications like videogames, keeping the FPS above a reasonable value. This meant the beginning of a new era of real time rendering.

In my thesis, I am going to point out the differences and compromises one has to make between ray tracing and rasterization. I will introduce the recently released DirectX RayTracing (DXR) compatibility in Unity Engine, then create a ray tracing shader for an effect not yet implemented in Unity using High Level Shading Language (HLSL). Finally, I will demonstrate the results in an applicable scene.

My goal is to give an overall picture of Unity's current capabilities regarding raster and ray tracing rendering.

1 Bevezetés

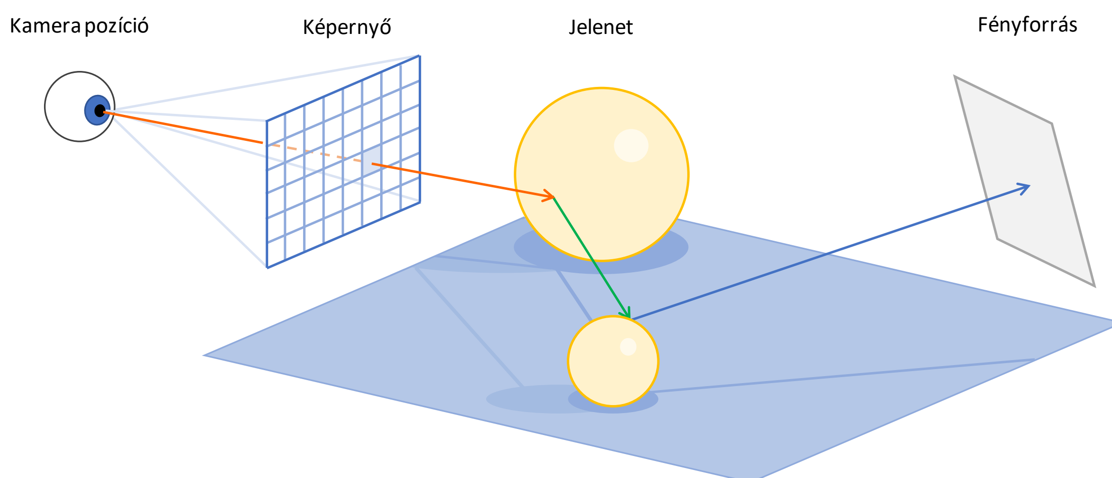
A digitális képalkotás mindenkori célja, hogy a megjelenített jelenet a lehető legjobban hasonlítson a valóságra. Ez a törekvés a filmekben használt Computer Generated Imagery (CGI) esetében gyakorlatilag már majdnem megvalósult, azonban egy valósághű, néhány másodpercnyi CGI rendereléséhez órákra, ha nem napokra van szükség, emiatt valós idejű alkalmazására eddig nem volt lehetőség. A folyamat időigényének oka, hogy a lehető legszebb eredmény eléréséhez a renderelő programok sugárkövetést használnak, ami a fény fizikai terjedésének, törésének és visszaverődésének szimulációját jelenti, és rengeteg számítással jár képkockánként. Egyetlen kép előállításához akár több százmillió sugár létrehozása szükséges, melyeknek egyenként kell kiszámítani az útját a bonyolult geometriákkal teletűzdelt jelenetben.

1.1 Sugárkövetés

A sugárkövetés alapja a fotonok mozgásának, valamint a virtuális térben elhelyezett tárgyakkal történő interakciójának fizikai szimulációja [1]. Bár a valóságban a fotonok a fényforrásból indulnak útjukra, ez a sugárkövetés esetén általában fordítva történik: a fénysugarakat a kamerából indítjuk. Ennek az az egyszerű oka, hogy mi csak a kamerába jutó fényre vagyunk kíváncsiak, így fölösleges lenne minden fényforrásból az összes lehetséges irányba sugarakat indítani: nagy részük nem, vagy csak nagyon sok törés és/vagy tükröződés után jut el a kamerába, rengeteg többletszámítást eredményezve és nagyon kevéssé befolyásolva a végleges képet. A fotonok (közel) egyenes vonalú mozgást végeznek, így az útjuk mindkét irányból indulva könnyen utánozható.

1.1.1 Sugarak indítása

Képzeljük a saját szemünket a kamerának, vagyis a fotonok kiindulási pontjának. Osszuk fel a képernyőt apró területekre (pixelekre), majd a kamerából minden egyes területen keresztül indítsunk el egy sugarat. Ezek a sugarak előbb vagy utóbb keresztezni fognak egy játékobjektumot, majd annak anyagától függően visszaverődnek és/vagy megtörve az objektum belsejébe jutnak, veszítenek intenzitásukból, esetleg teljesen elnyelődnek.



1.1 A sugárkövetés egyszerű ábrázolása

Bizonyos esetekben a sugarak egy fényforrást metszenek. Ebben az esetben az adott sugár egy olyan világító objektumot talált, ami a képernyőnek azt a pixelét világítja meg, amin a sugár induláskor áthaladt (mivel, ha a fényforrást eltaláló **A** sugár beérkezési irányának fordítottjába elindítanánk egy **B** sugarat, az ugyanazt az utat tenné meg a jelenetben, mint **A**, és végül a kamerában kötne ki). Ennek a pixelnek a színét tehát befolyásolja az említett fényforrás színe és intenzitása.

1.1.2 Rekurzió

Mikor egy sugár találkozik egy anyaggal, egy visszavert vagy megtört sugár keletkezik (kivéve, ha az anyag teljesen elnyelte a sugarat), ezért a metszéspontból a törés vagy visszaverődés irányába egy új sugarat indítunk. Az új sugarak ugyanúgy viselkednek, mint a régiek, azzal a különbséggel, hogy a megszerzett fényadatot „visszaadják” az őket létrehozó sugárnak.

A fent leírt folyamatot a sugarak pattanásának (bounce) nevezzük. Mindig fontos egy maximális bounce szám megadása, ellenkező esetben a program túl mélyre mehet a rekurzióban (túl sok sugarat hoz létre), felhasználva az összes memóriát (nem is beszélve a megnövekedett számítási időről), esetleg végtelen ciklusba kerül és lefagy az alkalmazás. Azonban minél nagyobb bounce értéket adunk meg, annál pontosabb eredményt kapunk, fontos tehát egy köztes megoldás megtalálása.

Az anyagok felülete a valóságban nem tükörsima, ezért a fényt nem szabályosan verik vissza. Emiatt nem hagyatkozhatunk arra, hogy csak egy visszavert sugarat hozunk létre, amit a tökéletes visszaverődés irányába indítunk: az érdes felületeket utánozva véletlenszerű irányokba újabb sugarakat indítunk. Az irányok szórását és eloszlását az

anyag érdesség (roughness) tulajdonsága befolyásolja. Ezeknek a sugaraknak a visszaadott értékeit súlyozással átlagoljuk (például a tökéletes tükröződés irányába indított sugár színe befolyásolja leginkább az őt létrehozó sugár színét, mert a valóságban is valószínűleg ebbe az irányba történik a legtöbb visszaverődés), és az így kapott értékkel tér vissza az anyag felszínét eltaláló sugár.

1.2 Sugárkövetési modellek

A sugárkövetés elméletének kialakulása egészen a XVI. századig vezethető vissza [2]. Albrecht Dürer 1522-ben a Négy Könyv a Mérésről című művében írta le a sugárkövetés elméletének alapjait, amit a mai napig hasonlóan alkalmaznak. Dürer sugarak helyett fonalat használt, annak végét illesztette a lerajzolni kívánt tárgy markáns pontjaihoz. A fonál áthaladt egy kereten (ennek neve Dürer Ajtó), ami a képernyő akkori megfelelője volt. A fonál a keret másik oldalán egy pontban volt rögzítve, ami ma a kamera pozíciójával egyezik meg. Természetesen ennek a módszernek nem a fény terjedésének modellezése volt a célja, csupán a lehető legpontosabb ábrázolása a kívánt tárgynak. Ez a technika nem volt alkalmas a sugarak visszaverődését, illetve törését utánozni, mondhatni a sugarak maximális bounce értéke nulla volt.

A modern számítógépek megjelenésével, a huszadik század második felében megszülettek az elmélet szoftveres megvalósításai is. Az első ilyen programot Arthur Appel valósította meg, aki a sugarakat a kamerához legközelebb eső objektumok megtalálására használta, majd ezek felületéről a fényforrások felé indított sugarakkal győződött meg arról, hogy az adott pont árnyékba esik-e vagy sem.

1.2.1 Whitted

Turner Whitted 1979-ben publikálta a saját, továbbfejlesztett sugárkövetési modelljét, mely már alkalmas volt a sugarak törését és tükröződését is szimulálni.

Emellett először alkalmazta a **Bounding Volume Hierarchy (BVH)** nevű módszert is [3]. Ennek lényege, hogy a virtuális tér objektumait közrefogjuk egy őket befoglaló, minél egyszerűbb és minimális méretű alakzattal (általában téglalappal két-, és téglatesttel háromdimenzió esetén). Ezeket a térfogatokat úgy szervezzük fa struktúrába, hogy a fa minden csomópontja egy újabb alakzatot definiál, mely közrefogja a csúcs ágainak alakzatait, és térfogata ismét minimális. Továbbá még cél, hogy a közös részfában található objektumok közel legyenek egymáshoz a háromdimenziós térben,

miközben a többi objektum minél távolabb helyezkedik el egy másiktól a fában, annál nagyobb legyen köztük a térbeli távolság is. A módszer segítségével jelentősen növelhető az ütközésdetektálás – és ezzel a sugarak és objektumok közti metszésdetektálás – gyorsasága és hatékonysága, mivel korlátozni tudjuk a megvizsgált objektumok számát az alapján, hogy a sugár a BVH mely csúcsain halad át.

1.2.2 Cook

Míg Whitted tökéletes töréseket és tükröződéseket számított a modelljében, addig Robert L. Cook az 1984-ben megjelent elosztott sugárkövetés nevű módszerével [4] számításba vette az anyagok felületének tökéletlenségét is. Ütközésenként tehát nem egy, hanem több, a tökéletes visszaverődéstől véletlenszerűen, de kismértékben eltérő irányokba is indított sugarakat, majd ezek eredményeit összegezte, így kapva a végső árnyalást.

Cook-ot nem csak ez foglalkoztatta, sok más területen szerette volna a korábban túl pontos sugárkövetést valósághűbbé tenni az emberi szem és a kamerák természetes „hibáinak" utánozásával. Fontos lépéseket tett többek között az elmosódás (motion blur), mélységélesség (depth of field) területén. Az addig használt lyukkamerát, ami minden objektumot fókuszba helyezett, a valósághoz közelebbi módszerre cserélte, ezzel megvalósítva a mélységélességet. Hasonlóan, az elmosódásokhoz a kamerák zársebességét modellezte azzal, hogy egy képkocka sugarait nem egy pillanatban indította, hanem időben fokozatosan elosztva azokat.

1.3 Inkrementális képszintézis

A videojátékok – és más, valós idejű megjelenítést alkalmazó szoftverek – a sugárkövetés erőforrásigénye miatt más módszerekkel próbálták meg elérni a lehető leglátványosabb megjelenítést. Az inkrementális képszintézis (más szóval raszteres megjelenítés) sugarak útjának számítása helyett a jelenetben található geometriákat dolgozza fel, transzformációs mátrixok segítségével rávetíti őket a kijelzőre. Ez a módszer lényegesen gyorsabb a sugárkövetésnél, de cserébe ront a valósághűségen és néhány fontos effektus használhatóságát jelentősen korlátozza.

1.3.1 Transzformációk

Ahhoz, hogy a jelenetben található geometriákon egyszerűen tudjunk számításokat végezni, célszerű azokat egy erre a célra megfelelő koordináta-rendszerben

tárolni. A renderelés során történő módosítások öt különböző koordináta-rendszer alkalmazását követelik meg [5]. Minden játékobjektum rendelkezik egy saját koordináta-rendszerrel, melynek középpontja az objektum origója. Ezt hívjuk **local space**-nek (lokális koordináták). Ahhoz, hogy az objektumok egymásra hatását könnyen tudjuk számolni, őket egy közös, világ koordináta-rendszerben (**world space**) szoktuk elhelyezni. A két rendszer közti koordináta átszámítást egy négydimenziós mátrixszal való szorzás segítségével tudjuk megtenni. A local space-ből world space-be transzformáló mátrixot **modell mátrixnak** nevezzük. Egy ilyen transzformációs mátrixot egy forgatás-, eltolás- és skálázás mátrix összeszorzásával kaphatunk meg.

Ahhoz, hogy a mátrix szorzás lehetséges legyen, a koordinátákat négyelemű vektorokkal, úgynevezett homogén koordinátákkal ábrázoljuk. A homogén koordináták fontos tulajdonsága, hogy nullától különböző számmal való megszorzás esetén a reprezentált pont változatlan marad, valamint lehetővé teszik, hogy a vektorokon végzett forgás, skálázás és eltolás műveleteket mátrixokként írjuk le.

A jelenetet a kamera szemszögéből figyeljük meg, ezért célszerű lenne egy olyan koordináta-rendszer használni, aminek az origója a kamera középpontja. Ezt a rendszert **view space**-nek hívjuk, bele a **view mátrix** segítségével transzformálhatunk world space koordinátákat. A kamera szemszögéből tudunk számításokat végezni, megtörténik a vágás, ami a kamera látószögén kívül eső objektumok eldobását jelenti, mivel ezek renderelésére később nem lesz szükség. A vágás előtt az objektumokat normalizált eszköz koordináta-rendszerbe (normalized device coordinates, NDC) kell átszámítani. DirectX-ben az NDC-k fontos tulajdonsága, hogy az X és Y koordinátájuk értéke -1 és 1 közé esik, ahol az X elem -1 értéke a kijelző bal széle, 1 a jobb széle, Y -1 értéke a kijelző alja, 1 pedig a teteje. DirectX-ben a Z koordináta 0 és 1 között mozog, ahol 1 a távoli, 0 a közeli vágósíkot jelenti (vagyis a Z tengely a kamera nézeti irányába mutat). Ezek a szabályok technológiánként kissé eltérhetnek, például OpenGL esetén Z is -1 és 1 közé esik, és a tengely pont az ellentétes irányba néz. Változhat még az Y tengely iránya. A transzformációt a **projekciós mátrix** végzi el. A vágás következtében az 1 és -1 értékeken kívül eső csúcspontokat eldobjuk, az így kapott koordináta-rendszert **clip space**-nek nevezzük. Az utolsó lépésként a csúcspontokat átadjuk a raszterizáló egységnek, ami interpolációval a háromdimenziós primitíveket a kétdimenziós kijelző pixeleire, a **screen space** rendszerbe rajzolja.

A teljes folyamatot, vagyis a három transzformációs mátrix egymás utáni végrehajtását (szorzatukat) röviden MVP-nek, **Model View Projection Matrix**-nak hívjuk.

DirectX-ben a kijelző pixeleit egy olyan koordináta-rendszerrel írjuk le, aminek origója a bal felső sarokban található, tehát X tengely jobbra, Y pedig lefelé néz.

1.3.2 Screen space sugárkövetés

A raszteres megjelenítésben létezik egy technika, ami valamilyen szinten megközelíti az igazi sugárkövetés tulajdonságait. Az ezzel a technikával megvalósított effektusokat a „screen space" jelzővel látták el, utalva ezzel arra, hogy a számítások a képtérben zajlanak. Ezek a raszteres csővezetékkel előállított mélység, szín és normál bufferek segítségével végzik el a számításokat. Ezekben a bufferekben a képernyő minden pixelére az abból látható felület kamerától mért távolsága (mélység- vagy z-buffer), színe, és felületi normálisa található. A mélység- és normál bufferek együtt gyakorlatilag egy háromdimenziós teret alkotnak, hiszen segítségükkel a kép minden pixelében tudjuk, hogy a látható felület milyen távolságra van, és melyik irányba néz. Ezekkel az adatokkal már tudunk sugárkövetést végezni.

Mivel a bufferek mérete megegyezik a renderelt kép felbontásával, az ilyen módszerrel utánzott fényterjedés nem tudja figyelembe venni azokat az objektumokat, amik a vágás során nem fértek fel a képernyőre, vagy egy másik játékelem miatt takarásban vannak. Mégis (ahogy azt később látni fogjuk), ezek a hibák nem mindig észrevehetőek, a sugárkövetéshez képest pedig nagyságrendekkel gyorsabb megjelenítés érhető el ezekkel a módszerekkel.

2 DirectX

A Microsoft DirectX egy főleg játékfejlesztéshez használt Application Programming Interface (API) gyűjtemény. A név sok különböző komponenst takar, melyek közül a leggyakrabban használt, és a dolgozat témájából kifolyólag számunkra legfontosabb a Direct3D, vagyis a valós idejű háromdimenziós megjelenítéssel foglalkozó és videokártya (GPU) programozást lehetővé tévő interfész. Az API kizárólag a Microsoft platformjain (Windows operációs rendszert használó asztali számítógépeken és Xbox játékkonzolokon) fut. Jelenleg a legfrissebb elérhető verzió a 2015-ben kiadott DirecX 12. Ennek a változatnak egy későbbi frissítése vezette be a sugárkövetés támogatását 2018 október 2-án.

Programozási nyelve a High-Level Shading Language (HLSL), ami egy C nyelvhez nagyon hasonlító szintaktikát használó, alacsony szintű videokártya programozási nyelv. Szinte teljesen megegyezik a Unity által használt Cg nyelvvel, de sok a hasonlóság az OpenGL által fejlesztett OpenGL Shading Language (GLSL) felé is.

2.1 Árnyalók

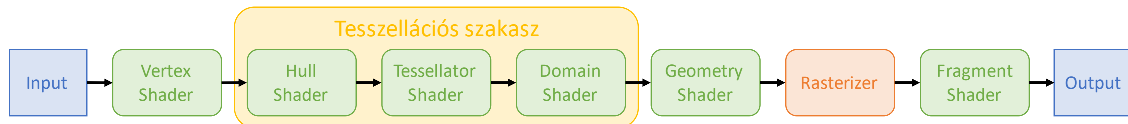
Videokártya programozásakor a fejlesztők rövid kódokat írnak, melyeket majd a videokártya fog futtatni. Az ilyen programokat hívjuk árnyalóknak (angolul shader). Árnyalókat a DirectX API használatakor HLSL-ben írunk. Az árnyalóknak több típusa is létezik, ezek között többnyire csak a rájuk bízott feladat jelenti a különbséget, szintaktikában mind azonosak. Mivel egy átlagos GPU-nak akár több ezer processzormagja is van, és minden magon külön futtatható egy árnyaló példánya, léteznek olyan shader programok is, amelyek ezt a számítási kapacitást más célokra, például bonyolult számítások elvégzésére, vagy kriptobányászatra használják.

2.2 DirectX csővezetékek

A renderelési folyamatot több program egymás után futtatásával hajtja végre a videokártya. Az árnyalók (és egyéb programok) sorozatát nevezzük csővezetéknek (angolul pipeline).

2.2.1 Raszter csővezeték

Évtizedek óta a valós idejű megjelenítésben a raszteres megoldás volt a bevált módszer. A folyamat legtöbb része olyan árnyalókból áll, melyet a fejlesztők tetszés szerint írhatnak meg, míg más részeibe csak minimális beleszólásuk van. DirectX-ben a raszteres csővezetéknek hét fázisa van [6], ezeket a 2.1 ábra szemlélteti:



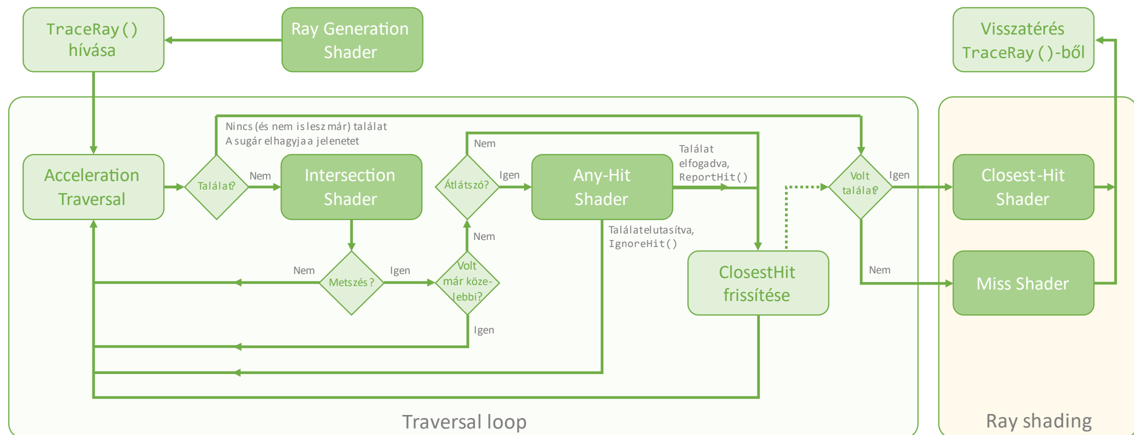
2.1 ábra: DirectX 12 raszteres csővezeték

A csővezeték bemenetként a jelenetben található játékobjektumok 3D modelljeinek csúcspontjait (vertex) kapja meg, és mindegyikre külön lefuttatja a **Vertex Shader**-t (csúcspont árnyaló, VS). Ennek feladata, hogy esetleges transzformációkat, megvilágítás számítást és egyéb módosításokat hajtson végre a csúcspontokon. A VS kimenete a módosított vertex.

A következő három shader alkotja a tesszellációs szakaszt, ami során a kapott vertex sorozatokból álló térbeli sokszögeket könnyen renderelhető háromszögekre, vonalakra és pontokra bontjuk (háromszögelés). A DirectX 11-ben már hardveresen implementált folyamat nem csak a meglévő csúcspontokkal tud dolgozni, de újakat is létrehozhat, így egy alacsony felbontású modellből részletgazdag formákat tud előállítani. Ez nagyban felgyorsítja a modellek renderelését: csökken a memória és videokártya közti adatátvitel, mert nincs szükség nagy részletességű modelleket átküldeni a videokártya számára.

Az így kapott geometriai primitíveket kapja meg a **Geometry Shader** (esetleg az adott primitívvel élszomszédos más alakzatokat is meg lehet adni bemenetként), és ezen végez módosításokat, transzformációkat és számításokat. A végeredményt továbbadja a **Rasterizer**nek, aminek a feladata az alakzatokat kép-térbe transzformálni (raszterizálás). Ez annyit jelent, hogy a 3D világból egy kétdimenziós síkra (a kijelzőre) kell vetíteni a jelenet objektumait. Mivel a képernyő véges számú pixelből áll, az alakzatok éleit interpolálni kell a négyzet alakú területeken. Az így előállított képen még minden pixelre lefuttatjuk a **Fragment Shader**-t (más nevén Pixel Shader), amely képpontonként végezhet utólagos feldolgozási folyamatokat (post-processing), fényszámítást és egyéb számításokat.

2.2.2 RayTracing csővezeték



2.2 ábra: DXR csővezeték

A sugárkövetéses renderelési folyamatnak három fő fázisa van: először sugarakat indítunk a képernyőn keresztül a virtuális térbe, ezek a sugarak objektumokat metszenek, tükröződnek, megtörnek stb. (**Traversal Loop**), eközben színadatot tárolnak el az érintett anyagok típusa és az azokra eső fény alapján, végül kiszámoljuk a pixelek színét a sugarak információi alapján (**Ray Shading**).

A folyamat öt darab árnyalóval zajlik [7] [8]. A **Ray Generation Shader** feladata a sugarak létrehozása, felparaméterezése és útra bocsájtása. A sugár útjának számítását az **Acceleration Traversal** komponens végzi a jelenetben található objektumokból alkotott BVH segítségével. Ha az elindított sugár valószínűleg ütközik egy objektummal, meghívódik az **Intersection Shader**. Ez egy olyan program, aminek a viselkedése különböző geometriákra (például háromszög, gömb stb.) külön implementálható. Ha a shader számításai szerint mégsem történt ütközés, a sugár tovább halad.

Ütközés észlelése esetén két dolog történhet: ha az eltalált objektum átlátszó (például üvegből van), meghívódik az **Any-Hit Shader** és a sugár az anyagon keresztül halad tovább, potenciálisan eltalálva más objektumo(ka)t is. Ha az objektum átlátszatlan, az ütközési pont elmentésre kerül a **ClosestHit** változóba, ha az esetleg korábban már benne eltárolt adat nem egy a kamerához közelebbi pontot reprezentál.

Az **Any-Hit Shader** feladata eldönteni, hogy a metszett átlátszó objektum hogyan befolyásolja a sugarat (szín változása, út módosítása stb.). Új sugar(ak)at is létrehozhatunk benne, vagy akár teljesen figyelmen kívül hagyhatjuk ezt az ütközést az IgnoreHit() függvényhívással.

Ha a sugár már (valószínűleg) nem fog újabb tárgyakat metszeni, kilépünk a hurokból. Abban az esetben, ha a **ClosestHit** változó üres (a sugár semmivel sem ütközött), akkor a **Miss Shader**, különben pedig a **Closest-Hit Shader** kerül meghívásra. Ezekben a shader-ekben szabadon árnyalhatjuk a sugarat, utóbbi esetében birtokunkban a **ClosestHit**-ben eltárolt legközelebbi ütközés helyével és az eltalált objektummal.

3 Unity játékmotor

A Unity egy elsősorban játékok fejlesztéséhez szánt grafikus motor és 3D modellező eszköz, sok más egyéb funkcióval ellátva. Egyéni használatra a program ingyenesen elérhető, ezzel ideális választás a játékfejlesztés alapjainak elsajátításához.

Programkódok (script) írásához a C# programozási nyelvet használhatjuk, míg az árnyaló programok nyelve Unity-ben elsősorban Cg, ami igazából a HLSL egy variációja, így valójában ez utóbbiban programozunk (gyakorlatilag semmi különbség nincs köztük).

3.1 Unity alapok

Minden Unity projekt alapja egy (vagy több) jelenet (Scene), melyben a virtuális teret építhetjük fel. A jeleneteket képzelhetjük olyan tárolóknak, melyek a játék különböző objektumait tartalmazzák. Egy jelenet reprezentálhat egy játékon belüli pályát, a főmenü egyik almenüjét (például a beállításokat), vagy akár a betöltő képernyő is kaphat egy külön Scene-t.

A jelenetekben tárolt elemeket játékobjektumoknak (GameObject) nevezzük, ezek tekinthetők a Unity-ben történő fejlesztés legfontosabb építőelemeinek. Lehetnek 3D modellek, kamerák, fények, kétdimenziós képek (sprite) és még sok minden más. Az objektumokra különböző komponensek illeszthetők. A rengeteg beépített típus közül az egyik legfontosabbak a **Renderer**, aminek segítségével az objektum megjelenítését módosíthatjuk és anyago(ka)t állíthatunk be. Az anyagoknak külön megadható, hogy milyen shader segítségével jelenítse meg őket a program. Egy másik nagyon fontos komponens a **Script**, vagyis kódfájl, amivel tetszőleges funkcióval ruházhatjuk fel a játékobjektumot. A szkripten belül C# nyelven írhatunk saját viselkedéseket a játékmotor beépített, indításkor (Start, OnEnable stb.), képkockafrissítéskor (Update, OnRenderImage stb.) és egyéb események esetén automatikusan hívott függvényeivel.

A legegyszerűbb játékobjektumot, ami mellőz minden felesleges komponenst, a Unity-ben Empty-nek hívjuk. Ezek ideálisak olyan scriptek tárolására és futtatására, amik nem egy konkrét játékobjektum tulajdonságait (például mozgását) hivatottak befolyásolni, hanem mondjuk a jelenet létrehozásáért és karbantartásáért felelősek.

3.2 Unity csővezetékek

Minden játéknak mások az igényei a grafikai megjelenítés szempontjából. Lehet, hogy az egyikben csak minimális, low-poly stílusú játékelemeket szeretnénk megjeleníteni, míg a másikban a lehető legjobban megközelítenénk a valóságot. Látható tehát, hogy egy sokrétű játékmotornak milyen elvárásoknak kellene megfelelnie. A Unity három beépített csővezetéket is biztosít, de a fejlesztőknek lehetősége van saját maguknak is létrehozni egyet.

3.2.1 Built-in Render Pipeline

Ez egy egyszerű, minden általános igényt kielégítő csővezeték. Épp ezért nagyon egyedi megoldásokhoz valószínűleg nem ez a megfelelő választás. Nem igazán személyre szabható, csupán a renderelési útvonalat (rendering path) lehet manuálisan állítani benne. Egy rendering path a megjelenítés és a teljesítmény között próbál valamilyen kompromisszumot találni, ezekből összesen négy félét biztosít a Built-in Render Pipeline: a Forward Rendering és Deferred Shading a két legfontosabb. Míg az előbbi egy általánosan mindenre jó renderelési mód, sok fényforrás használata esetén célszerűbb az utóbbit választani. Ezek kissé elavult, más módszerrel megvalósított változatai rendre a Legacy Deferred és Legacy Vertex Lit.

3.2.2 Universal Render Pipeline

Az URP egy főleg dizájnereknek, grafikusoknak és a játékon dolgozó egyéb kreatív munkát végző, kevés programozási ismerettel rendelkező személyeknek szól. Több területen biztosít kódmentes fejlesztési lehetőséget (például HLSL kód nélkül, shadergraph-ban lehet árnyalókat írni). Néhány apró paraméterben eltér a Built-in Render Pipeline-tól, egy-két technológiát az egyik támogat, a másik nem, de legnagyobb részt megegyeznek és a bennük létrehozott projektek kompatibilisek is egymással.

3.2.3 High Definition Render Pipeline

Ellentétben az előző kettővel, a HDRP teljesen máshogy közelíti meg a renderelést. A legtöbb modern technológia megtalálható benne [9], köztük a dinamikus felbontás, realisztikus árnyékok és fények, screen space tükröződések, törések, globális illumináció, ambient occlusion, és nem utolsó sorban hardveresen támogatott valós idejű sugárkövetés, bár ez utóbbi hivatalosan még csak kísérleti jelleggel üzemel. A HDRP a

Unity zászlóshajója renderelés terén, mindenben a Unity legmagasabb színvonalát biztosítja.

3.2.4 Scriptable Render Pipeline

Ez egy olyan csővezeték, aminek egyetlen célja, hogy amennyire csak lehet, testreszabható legyen (ezért sem tekinthető beépített csővezetéknek). A fejlesztő az SRP (Scriptable Render Pipeline) segítségével beleszólhat a renderelési folyamatba, ezzel lehetővé téve, hogy egyéni hardverre optimalizálják a készülő alkalmazást. Az URP és a HDRP is az SRP-re épül.

3.3 Shader Object

A Unity az árnyaló programokat egy olyan fájlban tárolja, amiben egyszerre akár több shader kód is helyet kaphat. Ezeket a fájlokat hívjuk Shader Object-eknek. A fájlon belül „SubShader"-ekbe írhatunk különböző árnyalókat. Egy SubShader több Pass-ból állhat, amik egy-egy shader megvalósítást tartalmaznak. A Unity futás közben dönti el, hogy egy anyaghoz rendelt Shader Object-en belül melyik Pass használatával jelenítse meg az objektumot. Ezt a döntést befolyásolják a grafikus beállítások, de C# kódból egy shader objektumon is beállíthatjuk a kívánt Pass-t annak nevének megadásával a SetShaderPass() függvényhíváson belül. Egy ShaderObject felépítése a következő:

```
Shader "Árnyaló neve"
{
    Properties
    {
        _Color ("Color", Color) = (1, 1, 1, 1)
        // ... többi tulajdonság
    }
    SubShader
    {
        Tags
        {
            "RenderType"="Opaque"
            // ... többi címke
        }
        LOD 200 // árnyaló erőforrásigénye

        Pass
        {
            CGPROGRAM / HLSLPROGRAM // az árnyaló nyelvétől függően

            // ... shader kód

            ENDCG / ENDHLSL
        }
    }
    // ... többi SubShader
}
```

A shader kód lehet raszteres megvalósítás (például csúcspont és fragmens árnyalók), vagy akár sugárkövető árnyalót is írhatunk (Closest-Hit Shader). Ha egy sugár eltalálja az anyagot, akkor ez az árnyaló fog meghívódni. A SubShader-en belül található LOD érték a shader-ek közötti teljesítménybeli különbséget hivatott jelezni. Magasabb LOD érték nagyobb komplexitást jelez a játékmotornak. Egy Shader Object-en belül minden árnyalónak adhatunk egy ilyen értéket, ekkor a program automatikusan azt a változatot fogja kiválasztani, amelyik a számítógép képességeihez legjobban illik. Fontos, hogy a fájlon belüli árnyalókat ezen érték szerint csökkenő sorrendben vegyük fel.

Az első sorban található aposztrófok közé a shader kívánt elnevezését írhatjuk, ezen a néven fogjuk megtalálni a játékobjektumok Material komponensének „Shader" nevű legördülő menüjében. A Properties alatt a shader által megjelenített anyag tulajdonságait tudjuk megadni. Ha a Shader Object-et egy geometriára helyezzük, ezek a paraméterek az objektum Inspector ablakában legalul lesznek szerkeszthetőek.

4 Fényeffektusok Unity-ben

A HDRP csővezeték különböző módszereket biztosít a fényjelenségek megjelenítésére. A sugárkövetésen kívül elérhetőek még a hagyományos raszteres árnyalási technikák, köztük az újabb screen space megoldások is. A pár éve bevezetett sugárkövetés támogatás a screen space effektek helyett nyújt egy sokkal valósághűbb, de számításigényesebb alternatívát.

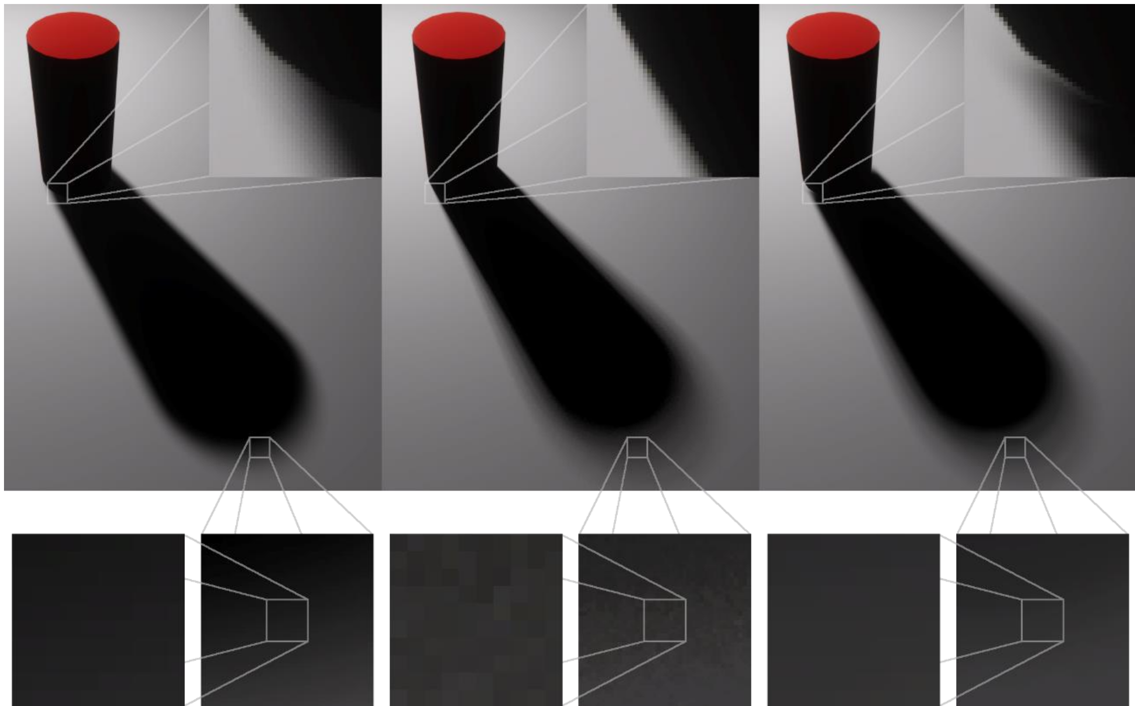
Egy másik lehetőség, hogy szondákat (probe) használunk. Egy szonda a tér egy olyan pontja, ahol információkat gyűjtünk a környezetről. Ez lehet a bejövő fények intenzitása, vagy a jelenetről alkotott körkép egy cubemap-ben eltárolva. A cubemap egy kocka alakba rendezett hat darab képet jelent, amiken a középpontból a háromdimenziós tér hat irányába alkotott képek láthatók. A szondákban tárolt adatokat beállíthatjuk a projekt szerkesztése közben (bake-elés), vagy valós időben, utóbbi esetben a jelenet változásait is figyelembe véve. Egy felület a hozzá legközelebb található szondá(k)ból kap információt a tükröződés és fényerősség számításához.

4.1 Árnyékok

Az árnyék a tér azon része, ahová takarás vagy a terjedés iránya miatt nem jut elég fény a fényforrásokból. A legegyszerűbb előállítási mód akkor adódik, ha eltekintünk a fényforrások térbeli kiterjedésétől. Természetesen a valóságban minden fényforrásnak felülete van, így a realisztikus képalkotáshoz elengedhetetlen ezeket kellőképpen modellezni. A kiterjedt fényforrások miatt az árnyékok széle elmosódott lesz, mértéke pedig a tárgy és a fényforrás közti, valamint az árnyék és a tárgy közti távolságától függ.

Unity-ben az árnyékok beállításait a fényforrásokon végezhetjük akár külön-külön is. Amellett, hogy be- és kikapcsolhatjuk a vetett árnyékokat (shadow map), ki is választhatjuk, hogy a motor milyen módon számítsa ki őket. A beállítások elérhetőségei függenek a fényforrás típusától (pont, irányított, terület). A 4.1 ábrán egy terület fényforrás világítja meg a piros hengert. A kiterjedt fényforrások fontos tulajdonsága, hogy az árnyékok szélei elmosódottak lesznek a fényforrástól távolodva. Bár az elmosódás megfigyelhető screen space árnyékok esetén, ez nem közelíti eléggé a valóságot, mivel az elmosódás mértéke az árnyék minden részén megegyezik. Sokkal valósághűbb képet ad a sugárkövetéses modell, azonban a pontosság a mintavételezéstől

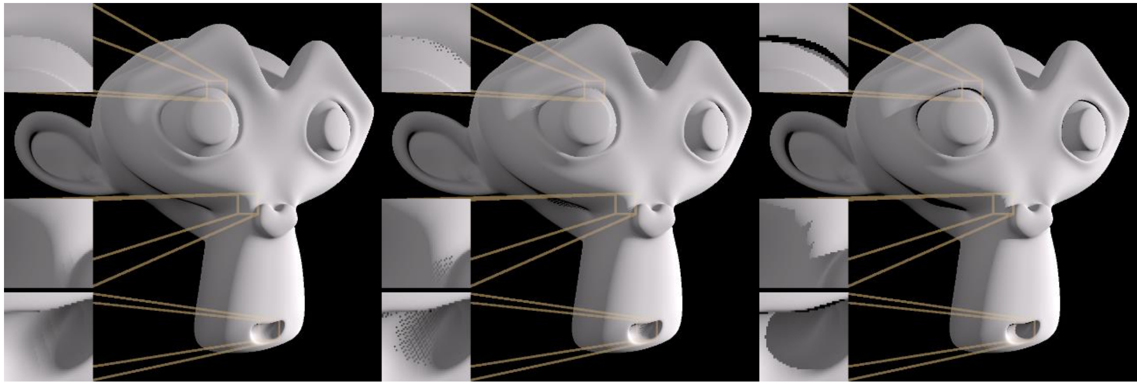
(az indított sugarak számától) erősen függ. A második kép felnagyított részén látszik, hogy az árnyék széle nagyon pixeles. Ez javítható lenne a mintavételezés (indított sugarak számának) növelésével, de ennek hatására jelentősen nőne a számítás mennyisége is, így ez, ha lehet, kerülendő. Egy kedvezőbb megoldást biztosít a szűrés, ami a ritkásabban számolt sugáradatokból simítással állít elő egy sokkal tisztább képet (harmadik kép).



4.1 ábra: Árnyékok shadow map-pel, szűretlen és szűrt sugárkövetéssel

Szűrő használata esetén az árnyék tárgyhoz közeli szélein nagyobb átmenet látható, mint annak lennie kéne. Ennek oka, hogy a szűrő a kép egy pontját az azt körülvevő pixelek értéke szerint simítja, így az árnyék szélén ebbe a szűrési területbe sok olyan felületi pont is kerül, ahová fény jut. A fényes és árnyékos pixelek közti simítás eredményezi a valóságnál nagyobb átmenetet a széleken. Ennek a mértékét redukálhatjuk a szűrési terület méretének csökkentésével, de ezzel a képzaj visszatérését kockáztatjuk. Az árnyék és tárgyak közti érintkezési pontokban megfigyelhető egy kisebb szűrési hiba (harmadik kép jobb felső sarka).

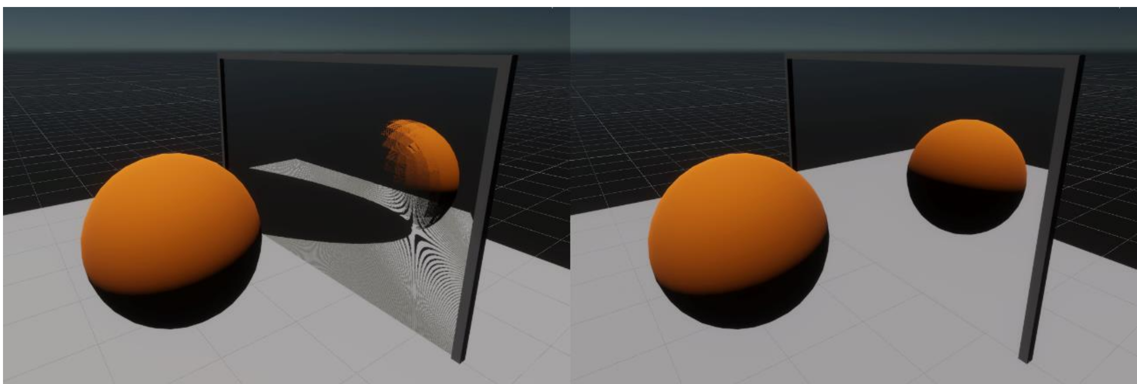
4.2 Contact Shadows



4.2 ábra: Kikapcsolt, screen space, és sugárkövetéses contact shadow

A contact shadow az árnyékok egy olyan fajtája, ami az apró részleteket próbálja hihetőbbé, kontrasztosabbá tenni. A 4.2 ábrán látható alakzat két téglalap alakú kiterjedt fényforrással van megvilágítva. Mindhárom képen screen space árnyékok is be vannak állítva. Az első esetben a contact shadow ki van kapcsolva, a második képen pedig már a bekapcsolt contact shadow látható, míg az utolsón ennek sugárkövetéssel számolt változata. Utóbbi esetében jól láthatóak a résekben létrejövő határozott árnyékok (a felülről második nagyított képen látható, már túl határozott árnyalás is előfordul), míg a screen space változat (második kép) nagyon visszafogott eredményt ad. Sugárkövetéses árnyékok esetén contact shadow nélkül is a harmadik képhez hasonló eredményt kapunk.

4.3 Tükröződés

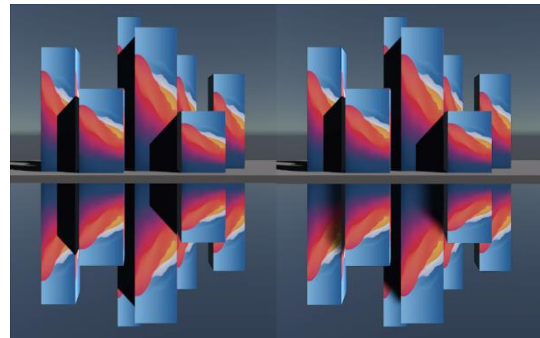


4.3 ábra: Screen space (balra) és sugárkövetéses (jobbra) tükröződés

Screen space tükröződés esetén csak azok a részletek látszanak, amiket a kamera közvetlenül lát, mivel ekkor a fény terjedésének útját a mélység- és normál bufferek segítségével számítja a játékmotor. Ez megfigyelhető a 4.3 ábrán látható narancssárga

gömbön, de a fehér talajon is: a tükörben látható fekete, ellipszis alakú terület nem a gömb árnyéka (az árnyékok ki vannak kapcsolva), hanem az általa a kamera elől kitakart felület. A sugárkövetéssel fizikailag modellezett fényterjedés segítségével könnyen rálátunk a kamera számára takarásban lévő részekre, ezzel pontos képet kapva a teljes jelenetről.

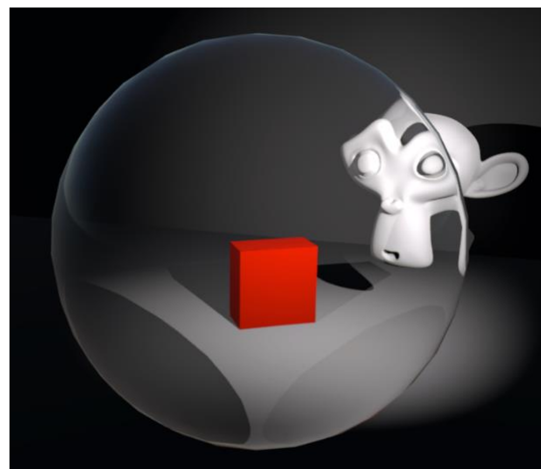
Fontos megjegyezni, hogy a 4.3 ábra célja a markáns különbségek demonstrálása egy szélsőséges helyzeten keresztül. Sok esetben a screen space tükröződés is tökéletes megoldást tud nyújtani, ahogy az a 4.4 ábrán látható tükröző talajon is látszik. Az ilyen helyzetekben mindenképpen kifizetődőbb ezt a módszert használni, hiszen a különbségek már-már észrevehetetlenek (ha a tükrözni kívánt tárgyak a képernyőn szerepelnek).



4.4 ábra: Screen space (balra) és sugárkövetéses (jobbra) tükröződés

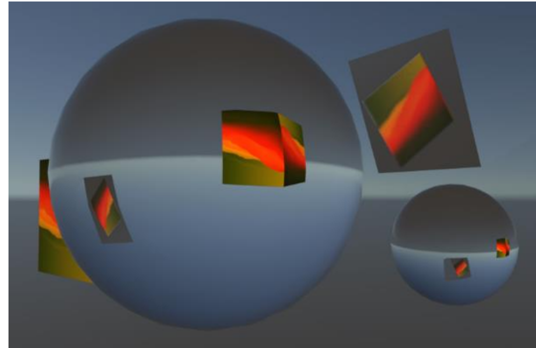
4.4 Törés

Fénytörést Unity-ben jelenleg csak screen space technikával lehet megvalósítani, nincs beépített sugárkövetéses megoldás. Bár a raszteres módszer is szép eredményeket ad, alkalmazása sajnos limitált a gömb és téglatest alakú geometriákra, emellett pedig csak a képernyőn szereplő objektumokról tud képet alkotni. A 4.5 ábrán megfigyelhető mellékhatás még, hogy a jelenetet megvilágító fényforrások a tökéletesen átlátszó objektum árnyékát is rávetítik a síkra raszteres és sugárkövetésen alapuló árnyékok esetén egyaránt.



4.5 ábra: Törés átlátszó gömb esetén

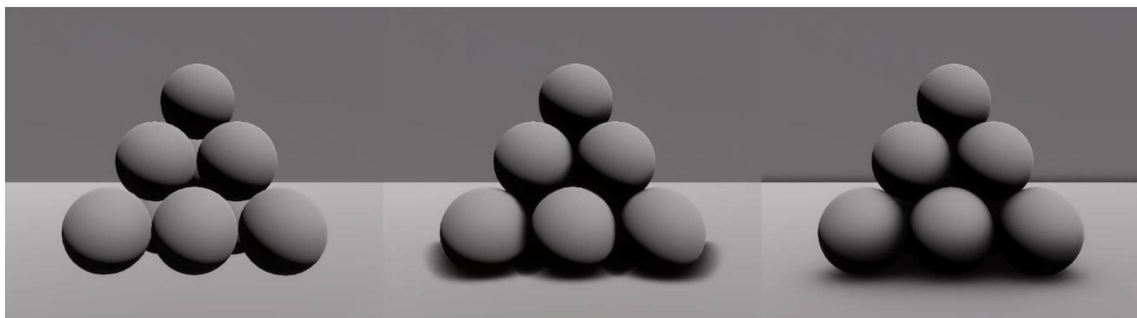
Sajnos a törő objektumok képe nem látszik se más törő, se tükröződő objektumokon, viszont a tükrök és az általuk tükrözött kép megfigyelhetőek a törő objektumokon (4.6 ábra). Ennek oka, hogy az átlátszó objektumokat a raszteres megjelenítéskor a motor nem írja bele a mélység bufferbe, hiszen átlátunk rajta. Egy egyszerű post-processing effekt hatására alakul ki az utánzott törés. Emiatt látjuk a színes kocka takarásban lévő részét is a screen space tükröződésben (jobb felső sarok), és ezért nem látszik a két átlátszó gömb képe egymáson.



4.6 ábra: Törő és tükröző objektumok

4.5 Ambient Occlusion

Az effekt célja, hogy realisztikusabb árnyalást adjon olyan szűkebb helyeken, ahová a szórt (ambiens) fényből kevesebb jut. Ahhoz, hogy az ilyen sötétebb pontokat kiszámoljuk, túl részletes és erőforrásigényes sugárkövetésre lenne szükség, ehelyett egy egyszerű technikával is látványos eredményeket érhetünk el. Fontos megjegyezni, hogy ez nem az árnyék alternatívája, csak egy extra effektus azon felül.



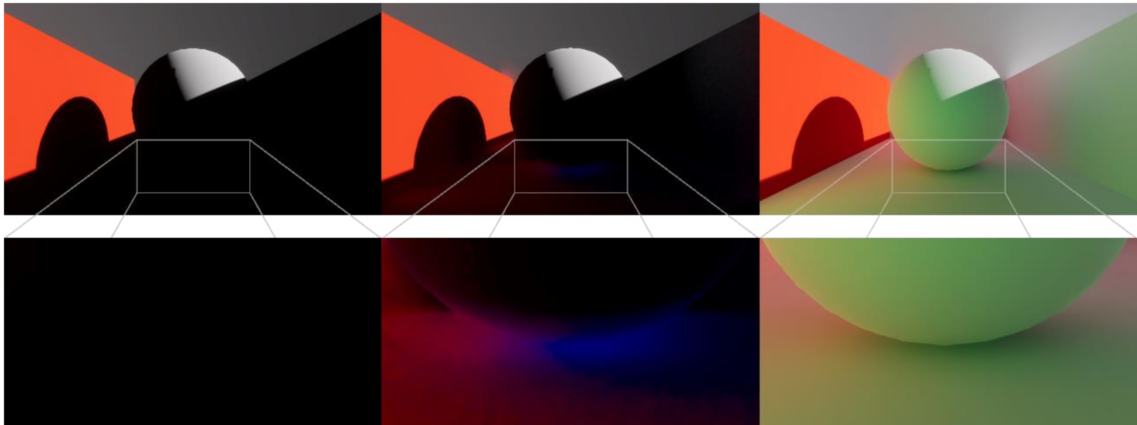
4.7 ábra: Kikapcsolt, screen space-, és raytracing alapú ambient occlusion

Működése nagyon egyszerű: feltételezzük, hogy a jelenetben mindenhova azonos mértékű ambiens fény jut. Az árnyékolás mértékét egy adott pontban látható felületből, annak normálvektora körüli félgömbben indított sugarak ütközései befolyásolják. Amelyik irányban egy sugár tárggyal ütközik, arról biztosan nem jön ambiens fény, ezért a sugár indulási pontjában sötétebb lesz a kép. Megadható az egy pontból indított sugarak száma, és hogy ezek milyen távolságon belül érzékeljenek ütközést. Screen space esetén

a sugarak a mélység buffert használják a metszések detektálásához, míg sugárkövetéssel valódi sugarakat indítunk a felületről.

Utóbbi módszerrel számított árnyalás a 4.7 ábrán látható módon sokkal valósághűbb képet ad a screen space változatnál, azonban nem annyira drasztikus az eltérés, mint más effektek esetén. Megfelelő finomhangolással a különbségek már elhanyagolhatónak mondhatók.

4.6 Global Illumination



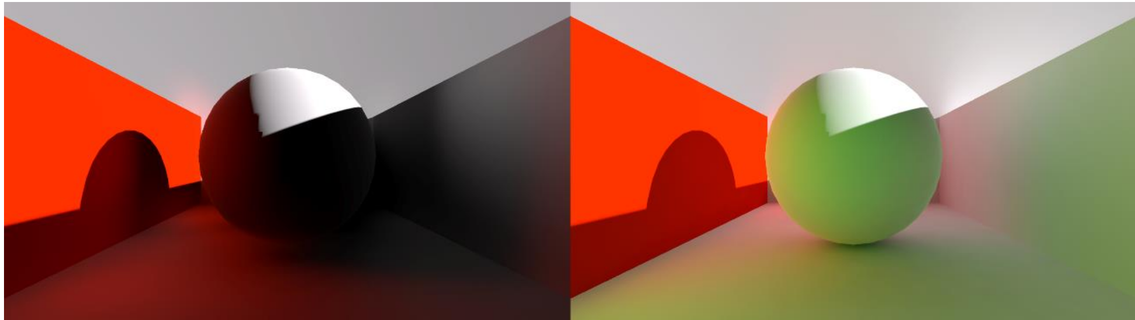
4.8 ábra: Kikapcsolt, screen space-, és sugárkövetés alapú globális illumináció

A globális illumináció célja a jelenetben található nem tükröző felületekről visszaverődő fény modellezése. A valóságban minden anyag valamilyen mértékben visszaveri a fénysugarakat, emiatt szinte sehol nem alakul ki teljes sötétség. Diffúz anyagok esetében ráadásul a visszaverődés nem mondható szabályosnak: a valóságban az ilyen felületek mikroszkopikus szinten nagyon rücskösek, így a beérkező fényt véletlenszerűnek mondható irányokba szórják szét. A tükröződés során az eltalált objektum anyagától függően megváltozhat a fény hullámhossza, vagyis színe, ennek hatására úgynevezett **color bleeding** jöhet létre azokon a felületeken, melyekre ez a színezett sugár vetül. A 4.8 ábra második és harmadik képén jól kivehető a fehér talajon és falon létre jövő piros elszíneződés, amit a bal oldali megvilágított piros fal okoz.

A Unity-vel érkező screen space alapú globális illumináció a 4.8 ábrán láthatóan kicsit javít az árnyékos helyek láthatóságán, azonban nagyon sok mellékhatással jár: a kijelzőn nem látható felületekkel érthető módon nem tud számolni (a jelenet bal szélén a piros fal folytatásaként egy zöld színű is el volt helyezve, ez okozza a harmadik kép domináns zöldességét), de emellett még a jelenetben található felületektől eltérő színű

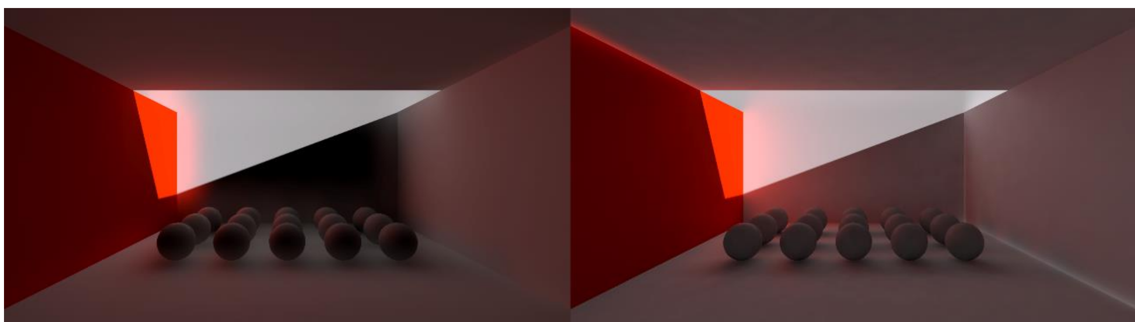
color bleedingek is előfordulnak (a második kép felnagyított – és a láthatóság miatt kissé kivilágosított - részén a gömb alatt egy sötétkék folt látható, ami valószínűleg egy hiba eredménye a beépített algoritmusban). A fenti három kép elkészítéséhez kikapcsoltam a Unity automatikus expozíció beállítását az összehasonlíthatóság érdekében, valamint mindkét technikát a legmagasabb minőségi, mintavételezési és szűrési beállítással használtam. Az expozíció állítgatásával egyébként a screen space globális illumináció hatása jobban láthatóvá tehető, de ez az anyagok színének eltorzulásával járna.

Érdekesség, hogy a Unity 2021.2.5f1 verzióval már nem észleltem az említett kék foltot, ráadásul sokkal szebb képet eredményez a screen space globális illumináció ebben a verzióban. Bár a 4.8 és 4.9 ábrán nem pont ugyanaz a jelenet látható (sajnos időben távol készült egymástól a két kép), a lényeges beállítások nem változtak.



4.9 ábra: Sugárkövetéses és screen space globális illumináció a 2021.2.5f1 verzióban

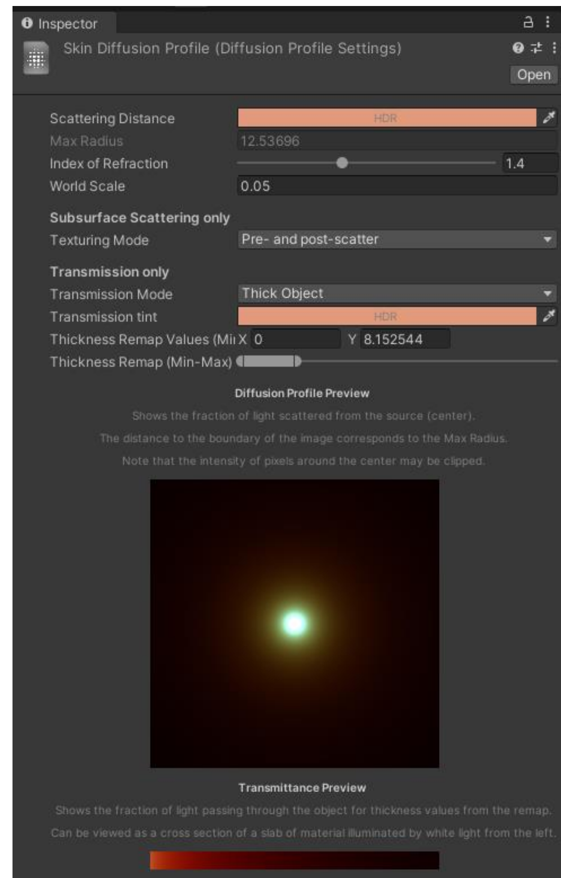
Egy sokkal igazságosabb jelenet látható a 4.10 ábrán: itt nem volt a jelenetben zöld fal, és igyekeztem a magas beállítások megtartása mellett hasonló eredményeket kapni a két technikától.



4.10 ábra: Screen space és sugárkövetéses globális illumináció a 2021.2.5f1 verzióban

4.7 Subsurface Scattering

A subsurface scattering segítségével azokat az eseteket szimulálhatjuk, mikor a fény nem csak visszaverődik, de bizonyos mélységig be is lép az objektum belsejébe. A valóságban szinte az összes anyagba nagyon kis mértékben, de behatol a fény, ezek közül a viasz, műanyag és a bőr a leglátványosabbak. Az effekt megjelenítéséhez Unity-ben először a szokásos módon engedélyezni kell azt az eszközfájlban és a kameráknál, valamint létre kell hozni egy Volume Override-ot. Szükség lesz egy anyag létrehozása is, ami ezt az effektet használja. Ehhez egy újonnan létrehozott anyagnál a Material Type beállítást Subsurface Scattering-re kell állítani, aminek hatására megjelennek az ehhez kapcsolódó beállítások a Surface Inputs fül alatt. Itt a Diffusion Profile nevű mező fogja tárolni az effekt beállításait, amit egy erre létrehozott Diffusion Profile fájllal tudunk megadni. A fájlon belül a 4.11 ábrán látható paramétereket állíthatjuk. A Scattering Distance segítségével megadható, hogy az anyag belsejébe jutó fény milyen színt adjon vissza (ez nem azonos az anyag felületének színével), a World Scale pedig az effekt számításához szükséges nagyságrendet adja. Az ábrán a HDRP-vel érkező beépített, bőrt utánzó beállítások láthatók.



4.11 ábra: Diffusion Profile beállításai

4.12 ábra: Subsurface Scattering különböző beállításokkal és megvilágítással

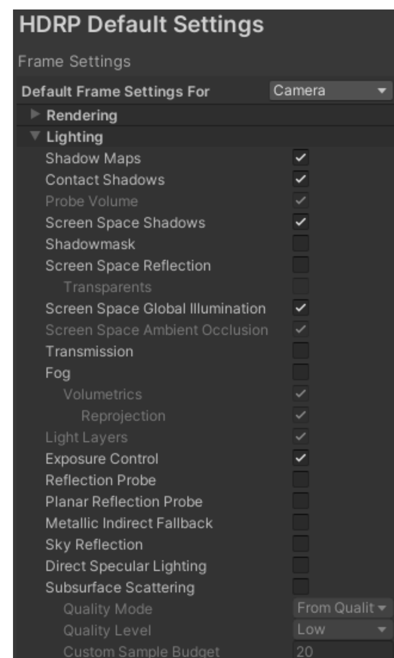
A 4.12 ábrán látható a fényeffektus különböző beállítások mellett renderelt képe. A felső sor esetében a jelenetet egy jobb-fentről érkező irányított fényforrás, és egy hátsó, az objektum miatt takarásban lévő spotlámpa világította meg. Az alsó sor esetén csak az utóbbi fényforrás volt használatban. Balról jobbra haladva az első képen a screen space módszerrel renderelt subsurface scattering látható, ami egészen szép eredményt ad és vetekszik a sugárkövetéssel számított második képpel. A sorban az utolsó kép pedig a Subsurface Scattering típusú anyag kinézete látható kikapcsolt subsurface scattering számítás mellett. Az alsó sorban az első kép szintén a screen space módszer eredménye. Látható, hogy az objektum mögött elhelyezett fényforrás a tárgynak egy egységes, vöröses színt ad, függetlenül a vastagságtól. A következő két képen látható sugárkövetéssel számolt megvilágítás össze sem hasonlítható az előző képpel. A valóságnak megfelelően, ahol az anyag vékony (a füleknél), ott több fény jut át, mint a vastag részeken. Az második sor utolsó két képe közötti egyetlen különbség a mintavételezés darabszáma. Míg az előbbinél a maximális 32 érték volt beállítva, addig az utolsónál csak 1. Ez elég látványos különbségeket eredményez (ahogy az a felnagyított részeken látszik), azonban nem szabad figyelmen kívül hagyni azt sem, hogy a kevesebb érték esetében az FPS körülbelül a hatszorosa volt a másik esetben mért számnak. Nagyjából a 8 mintavételezéstől fölfelé már elfogadható minőségűnek mondható az alkotott kép.

5 Raytracing a Unity-ben

A Unity játékmotor 2019 óta natívan támogatja a sugárkövetéses renderelést a Microsoft által 2018 október 10-én kiadott DirectX Raytracing (DXR) alkalmazásával, ami a DirectX 12 Application Programming Interface (API) része. Segítségével a szoftver képes kihasználni a modern videokártyák számítási erejét. Bár ez a technológia ebben a játékmotorban még kísérleti fázisban tart, mégis rengeteg fontos újdonság már működőképes állapotban van és látványos eredményeket mutat, így érdemes a használatra.

5.1 Volume Override

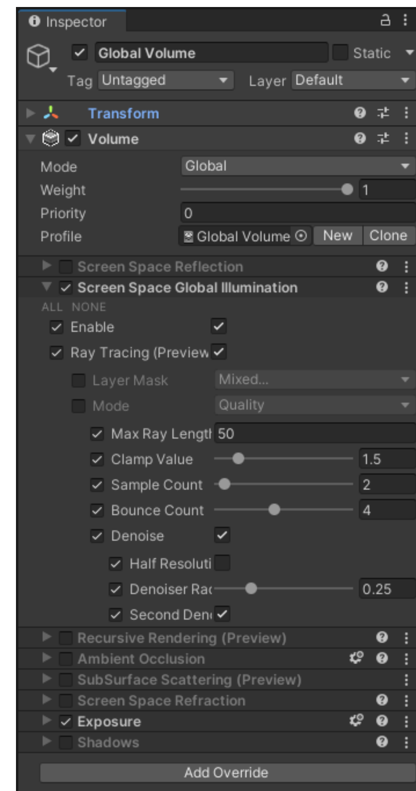
Minden egyes fényeffektus esetén annak megjelenítési módját a Unity által biztosított renderelési technikák (raszteres, szondás vagy sugárkövetéses) közül szabadon választhatjuk. Fontos először, hogy a jelenetben használt kamera képes legyen ezt az effektet megjeleníteni – ehhez a kamera General beállításai közül a Custom Frame Settings opciót ki kell pipálni, majd a megjelenő listában a Lighting menüpont alatt a megfelelő effektet engedélyezni kell. Ezt a folyamatot egyszerre az összes kamerára is elvégezhetjük az 5.1 ábrán látható *Edit → Project Settings → HDRP Default Settings* menüponton belül.



5.1 ábra: Kamerák alapértelmezett beállításai

Ha ezzel megvagyunk, a játékmotornak is el kell juttatnunk az igényünket a fényeffektus megjelenítésével kapcsolatban. Ehhez létre kell hoznunk egy minőség-beállítást a *Project Settings → Quality* menüpont alatt (Add Quality Level), majd meg kell adnunk egy eszközfájlt (Pipeline Asset), ami a renderelésre vonatkozó beállításokat tartalmazza. Ezt a fájlt a Project ablakban jobb klikkelve a *Create → Rendering → High Definition Render Pipeline Asset* opcióval hozhatjuk létre. Az eszköz beállításait az Inspector ablakban tudjuk szerkeszteni, itt szintén a kívánt effekteket kipipálva fejezzük ki a szándékunkat a motor felé.

Az egyes effektek saját beállításait itt még nem érjük el, ahhoz egy Volume Override létrehozása szükséges, ami a jelenet terének egy részét közrefogó objektum. Ha a kamera a játéktér ezen területén belül helyezkedik el, akkor rá a tér által felülírt beállítások érvényesek. Térbeli alakzat megadása helyett választható a Global mód is, ami a teljes jelenetre érvényesíti a beállításokat. A Hierarchy ablakban jobb egérgombot nyomva a *Create → Volume → Global Volume* létrehozásával kapunk egy az egész jelenetet beterítő objektumot. Az 5.2 ábrán szemléltetett Inspector ablakban a Profile mező mellett található New gombbal létrehozhatunk egy profilfájlt, ami a felülírt beállításokat fogja tárolni. Az alul megjelenő Add Override gombbal adhatunk meg egy renderelési beállítást a legördülő listából. A választott effektushoz megjelennek a felülírható beállítások. Itt tudjuk azt is kiválasztani, hogy a hagyományos raszteres, vagy az új sugárkövetéses módszert használja a játékmotor.



5.2 ábra: Volume Override-ok beállításai

5.2 DirectX API alapjai

A DirectX API számos beépített változót és függvényt biztosít a grafikus számításokhoz szükséges adatok egyszerű eléréséhez shader kódból. Az egyik ilyen a DispatchRaysDimensions(), ami egy háromdimenziós vektorként adja vissza a rendereléshez használt felbontást. Szintén fontos a DispatchRaysIndex(), ami az épp feldolgozásra kerülő sugár felbontásbeli elhelyezkedését adja vissza szintén egy háromdimenziós vektor formájában.

5.2.1 SceneAccelerationStructure

A jelenethez tartozó gyorsító struktúra feladata a háromdimenziós térben a sugarak terjedésének kiszámítása. Létrehozását külső kódból (jelen esetben C#-ban) kell megvalósítani, majd a felépített struktúrát át kell adni az őt használó sugárkövetés árnyalónak. A struktúra gyakorlatilag a jelenet általunk választott objektumait tartalmazza (az AddInstance függvénnyel adhatunk hozzá tetszőleges játékobjektumot,

amelyik rendelkezik Renderer komponenssel), és azokon végzi el a sugárkövetés modellezését.

5.2.2 RayPayload

A sugarak indításához először szükségünk lesz a sugarak adatainak (visszatérési szín, és egyéb, felhasználástól függő paraméterek) eltárolására. A RayPayload (sugár rakomány) egy olyan struktúra, melynek felépítését teljes mértékben a fejlesztő határozhatja meg. Célszerű azonban a méretét a lehető legkisebbre venni, mivel ezt az adatot a sugár útja során az árnyalóprogramok egymás között adogatják. Nagy méret esetén drasztikusan csökkenhet a sugárkövetés sebessége.

```
struct RayPayload {
    float3 color;       // sugár színe
};
```

A Closest-Hit, Any-Hit és Miss Shader-ek paraméterként megkapják ezt a struktúrát, így a sugár adatait egyszerűen tudják módosítani.

5.2.3 RayDesc

Egy sugár adatait leíró struktúra. Megadható a kezdőpont, irány, az ütközésig megteendő legkisebb és legnagyobb távolság. Ez egy beépített típus, így a fejlesztőnek nincs beleszólása a felépítésébe.

```
struct RayDesc {
    float3 Origin;      // kiindulási pont
    float3 Direction;   // irányvektor
    float  TMin;        // minimális távolság ütközésig
    float  TMax;        // maximális távolság ütközésig
};
```

5.2.4 AttributeData

Ha a sugár ütközik egy objektummal, valahonnan tudnunk kellene a metszett háromszög tulajdonságait. Ezt az információt hivatott tárolni az AttributeData. HLSL-ben található egy beépített struktúra háromszögek metszési adatának tárolására, azonban egy objektum nem feltétlenül csak háromszögekből áll. Lehetnek a jelenetben akár implicit felületek vagy procedurális geometriák, ezek esetében egy ezekkel kompatibilis AttributeData struktúra definiálása szükséges. A beépített struktúra a következő:

```
struct BuiltInTriangleIntersectionAttributes
{
     float2 barycentrics;
};
```

5.2.5 TraceRay()

A Ray Generation Shader-ben hívandó legfontosabb függvény a TraceRay(). A paraméterben kapott beállításokkal és a ray sugár példánnyal elindítja a sugárkövetést. A függvény visszatérésével a payload paraméterben a sugár által keresztezett pixel árnyalása lesz elérhető. Ezt a paramétert kapják meg az árnyalóprogramok is.

```
TraceRay(
    accStruct,          // gyorsító struktúta (SceneAccelerationStructure)
    RAY_FLAG_NONE,      // sugár opcionális beállításai
    0xFF,               // uint típusú változó az ütközések szűréséhez
    0,                  // választott Hit Group indexe
    1,                  // Hit Group-ok száma
    0,                  // választott Miss Shader indexe
    ray,                // sugár adatai (RayDesc)
    payload             // rakomány (RayPayload)
);
```

A fenti kódban látható "Hit Group" a sugárkövetés során alkalmazni kívánt Closest-Hit, Any-Hit és Intersection shader-ek előre létrehozott csoportját jelenti.

5.2.6 Raygeneration Shader

A sugarak indítását végző árnyaló egy tetszőlegesen elnevezett, paraméter nélküli függvény. A metódus neve elé a "raygeneration" jelzést írva hozzuk a csővezeték tudomására, hogy milyen típusú shader-ről van szó. A fejezet bevezetőjében említett két beépített függvénnyel lekérdezhetjük az jelenleg árnyalt pixel koordinátáit és a teljes felbontást.

```
[shader("raygeneration")]
void RayGenerationShader() {
    uint2 pixel = DispatchRaysIndex().xy;
    uint2 resolution = DispatchRaysDimensions().xy;
    ...
```

Majd külső kódból beállított paraméterek segítségével kiszámoljuk az indítandó sugár irányát. Ehhez a kamera frustumának (a látóteret leíró négyzet alapú csonkagúla, melynek alja és teteje a kamera közeli és távoli vágósíkja, oldalai a látóteret határoló síkok) négy oldalélének irányát használhatjuk:

```
    ...
    float2 uv = (float2)pixel / (float2)resolution;

    float3 topDir = lerp(_TopLeftFrustumDir, _TopRightFrustumDir, uv.x);
    float3 botDir = lerp(_BottomLeftFrustumDir, _BottomRightFrustumDir, uv.x);

    float3 rayDir = normalize(lerp(botDir, topDir, uv.y));
    ...
```

A lerp() függvénnyel lineáris interpolációt végezhetünk az első és második paraméter között. A harmadik paraméter egy lebegőpontos szám, ami az átmenet mértékét adja meg (0 esetén az első, 1 esetén a második paraméter értékét kapjuk vissza, a kettő közötti, vagy azokon kívül eső szám esetén pedig átmenetet végez az értékeken).

Az előállított adatokkal már csak a sugarat kell felparaméterezni és útjára bocsátani a TraceRay() hívással, majd a payload-ban visszakapott color mező értéke alapján a kimeneti textúra megfelelő pixelét színezni.

```
    ...
    RayDesc ray;
    ray.Origin     = _WorldSpaceCameraPos;
    ray.Direction  = rayDir;
    ray.TMin       = 0.0f;
    ray.TMax       = 100000;

    RayPayload payload;
    payload.color   = float3(0, 0, 0);

    TraceRay(_AccelerationStruct, RAY_FLAG_NONE, 0xFF, 0, 1, 0, ray, payload);

    outTex[pixel] = float4(payload.color, 1);
}
```

A fent bemutatott kódban minden aláhúzással ("_") kezdődő változó külső kódból lett beállítva, a _WorldSpaceCameraPos pedig a Unity által biztosított egyik shader változó, ami abban az esetben kerül automatikusan feltöltésre, ha a kamerát az árnyaló indítását végző Dispatch() függvénynek átadjuk paraméterként a C# kódban.

5.2.7 Miss Shader

Formailag megegyezik a sugarakat létrehozó árnyalóval („miss" címkét kell a függvény neve elé írni), de ebben az esetben az árnyalónak van egy RayPayload paramétere, amiben a sugár adatait kapja meg. Tetszőleges számításokat végezve árnyalhatjuk a sugár color mezőjét. Legtöbbször itt azt keressük, hogy a sugár hol metszi az ég textúráját, mivel a Miss Shader csak abban az esetben hívódik meg, ha a sugár semelyik játékobjektummal nem ütközött. A lehető legegyszerűbb Miss Shader csak egy színt ad vissza:

```
[shader("miss")]
void MissShader(inout RayPayload payload)
{
    payload.color = float3(0, 1, 1);
}
```

5.2.8 Closest-Hit Shader

Ezt a shader változatot definiálhatjuk a .raytrace fájlban, hogyha minden anyagra ugyanazt a megjelenítést szeretnénk, de általában anyagonként eltérő viselkedést akarunk elérni. Helyette minden sugárkövetésre felkészített anyag saját shader-ébe felvehetünk egyéni Closest-Hit árnyalókat, amik az anyag egyedi tulajdonságaihoz is hozzáférnek. Célszerű azonban a biztonság kedvéért létrehozni egy árnyalót a .raytrace fájlba is arra az esetre, ha a sugárkövetéssel renderelni kívánt anyagnak nincs saját Closest-Hit árnyalója. Ebben az esetben ugyanis az előbbi program fog lefutni.

A „closesthit" jelölővel ellátott shader programnak két paramétere van, egyik a szokásos RayPayload, a másik az ütközés információit tároló struktúra, amit saját magunk definiáltunk korábban. Ebben a shader-ben meghívható a TraceRay() függvény, vagyis lehetőségünk van új sugarakat indítani, ha például az anyag tükröző tulajdonságú.

5.2.9 Any-Hit Shader

Ahogy korábban már említésre került, ennek a shader-nek a feladata a sugár metszésének elfogadása a ReportHit(), vagy elutasítása az IgnoreHit() függvényhívások valamelyikével. HLSL-ben az „anyhit" címkével ellátott függvény veszi fel ezt a szerepet.

5.3 Sugárkövető árnyaló kezelése C# kódból

A sugárkövetést megvalósító shader elindítását C# oldalról tudjuk megvalósítani. Tetszőleges játékobjektumra csatolt szkriptben először létre kell hoznunk egy RenderTexture típusú változót, amibe az árnyaló kimenetét fogjuk eltárolni. Itt kell felvenni a RayTracingAccelerationStructure típusú gyorsító struktúrát is, amibe az AddInstance() függvénnyel manuálisan fel kell venni az összes játékobjektumot, amire szeretnénk, hogy lefusson a sugárkövetés. A struktúra konstruktorában megadható egy RASSettings nevű osztály, aminek a managementMode paraméterét automatikus módra állítva később új objektumok létrehozása esetén azokat nem kell majd manuálisan beletenni a struktúrába. A hozzáadandó objektumok halmazát szűkíthetjük a beállítás osztály layerMask és rayTracingModeMask paramétereivel, előbbi esetében csak az benne tárolt értékű layer(ek)ben létrehozott, utóbbinál pedig a vele megegyező sugárkövetési módot támogató objektumok kerülnek csak automatikusan bele a gyorsító struktúrába. Ha szeretnénk, hogy a struktúrába felvett objektumok mozgatáskor

frissüljenek a struktúrán belül is, akkor azt manuálisan kell megtennünk az UpdateInstance függvény meghívásával, minden objektum esetén külön.

A shader kódját tartalmazó .raytrace kiterjesztésű fájlt az editoron keresztül a szkript egy RayTracingShader típusú változójába kell bekötni, ezen később képkockánként a Dispatch() függvényt hívva tudjuk elindítani a sugárkövetést. Előtte viszont be kell állítani a shader számára szükséges változókat, amit a kimeneti textúra, gyorsító struktúra, valamint az egyéb típusú változók saját setter függvényével tudunk megtenni:

```
// a program indításakor:
shader.SetAccelerationStructure("_AccStruct", accStruct);
shader.SetTexture("outTex", resultTexture);

// minden képkockában:
shader.SetVector("_TopLeftFrustumDir", topLeft);
shader.SetVector("_TopRightFrustumDir", topRight);
shader.SetVector("_BottomLeftFrustumDir", bottomLeft);
shader.SetVector("_BottomRightFrustumDir", bottomRight);
shader.SetVector("_CameraPos", mainCamera.transform.position);
```

A Dispatch() függvénynek három kötelező és egy opcionális paramétere van. Először a shader-ben található RayGenerationShader nevét kell karakterláncként átadni, majd a renderelni kívánt kép felbontását két egész számként. Negyedik paraméternek átadhatjuk a használni kívánt kamera objektumot, ekkor a Unity a kamera adatait automatikusan betölti a HLSL kódból elérhető globális változókba, így ezt nem nekünk kell manuálisan setter függvényekkel megtenni.

6 Törésszámítás sugárkövetéssel

A HDRP-ben a fényeffektusok többsége már implementálva van raszteres és sugárkövetéses módszerrel is, de akad néhány speciális eset, melyek megjelenítésére saját shader-t kell írni. Ilyen például a törő objektumok sugárkövetéssel számított változata is. Egy ilyen típusú árnyaló megírását fogom most bemutatni.

Saját sugárkövetéses árnyaló létrehozásához mindenekelőtt át kell gondolnunk a shader programok között sugaranként küldött információkat. Az árnyaláson kívül szükség lehet egy mélység számlálóra is, ami azt jelzi, hogy az adott sugár hányadik leszármazottja az eredeti, kamerából indított sugárnak. Minden tükröződéskor és töréskor az újonnan létrehozott fénysugárnak az őt létrehozó sugár saját értékénél egyel nagyobbat adunk át. Ha ez az érték meghalad egy előre meghatározott mennyiséget, egy egyszerű elágazással megszakíthatjuk a sugár útjának számítását, ezzel megakadályozva, hogy túl mélyre menjünk a rekurzióban.

Először a külön .raytrace fájlban az 5.2.6 fejezetben látott módon létre kell hozni egy Ray Generation Shader-t, ami a sugarakat fogja útnak indítani. Ezután fel kell venni egy új Shader Object-et a Hierarchy ablakban, amibe a megvalósítani kívánt anyag (jelen esetben átlátszó üveg) megjelenítéséért felelős Closest-Hit Shader fog kerülni. Ahhoz, hogy a shader-ekben használt struktúrákat ne kelljen kétszer definiálni, célszerű kiszervezni azokat egy külön fájlba.

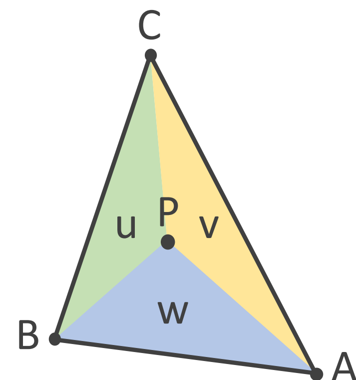
```
Pass
{
    Name "RayTracing"
    Tags { "LightMode" = "RayTracing" }

    HLSLPROGRAM
    #pragma raytracing test
    #include "Common.cginc" // a közös struktúrákat tartalmazó fájl

    uint _MaxBounce;

    if (rayPayload.depth == _MaxBunce)
    {
        return;
    }

    // ... további HLSL kód
    ENDHLSL
}
```

Kényszeríthetjük, hogy a Ray Generation Shader kizárólag egy adott Pass-t futtasson. Ehhez C# kódban a RayTracingShader objektum SetShaderPass() függvényét kell meghívni, paraméterként átadva a kívánt Pass nevét:

```
rayTracingShader.SetShaderPass("RayTracing");
```

6.1 Baricentrikus koordináták

A törés irányának kiszámításához fontos lenne tudnunk az eltalált háromszög adatait (például a csúcspontok world space pozícióit). Nekünk jelen esetben csak a felület normál vektorára lesz szükség. A Closest-Hit árnyaló paramétereként megkapjuk az eltalált pontot leíró AttributeData struktúrát, amiben összesen két float típusú változó található, ezek az eltalált háromszögön keletkezett metszéspont baricentrikus koordinátái [10]. Ez a koordináta típus egy geometriai primitív (ebben az esetben háromszög) felületének egy pontját írja le, beleértve az éleket és a csúcspontokat is. A csúcspontok koordinátáinak ismeretében a baricentrikus koordinátákkal ezek között súlyozással tudunk meghatározni egy újabb pontot. Háromszög esetén a három csúcs legyen A, B és C, a baricentrikus koordináták pedig rendre u, v és w. Ekkor egy tetszőleges pontot a háromszög felületén az alábbi egyenlet ír le: $P = u \cdot A + v \cdot B + w \cdot C$



6.1 ábra: baricentrikus koordináták ábrázolása

A baricentrikus koordinátákat normalizálva használjuk, emiatt a három szám összege mindig 1. Emiatt a három számot leírhatjuk kettővel is, hiszen a harmadikat a $w = 1 - u - v$ egyenlettel bármikor kiszámolhatjuk. Ezért kapunk az árnyaló paramétereként csak két számot, ráadásul így az adatátvitel mértéke is csökken a GPU-n.

6.2 Normálvektor kiszámítása

A Unity tartalmaz egy „UnityRaytracingMeshUtils.cginc” fájlt, melyben a csúcspontok adatait lekérdező metódusok találhatóak. Ezekkel megtudhatjuk az eltalált háromszög csúcspontjainak normálvektorát is. A UnityRayTracingFetchTriangleIndices függvénnyel a paraméterben átadott háromszög csúcspontjait kapjuk vissza, míg a UnityRayTracingFetchVertexAttribute3 függvénnyel a paraméterben átadott csúcspont egy választott tulajdonságát kérdezhetjük le. A háromszög azonosítóját a HLSL beépített

PrimitiveIndex() tudhatjuk meg. A vektorok és a baricentrikus koordináták ismeretében, ha a $P = u \cdot A + v \cdot B + w \cdot C$ képletben a pontokat vektorokra cseréljük, egyszerűen kiszámíthatjuk a metszéspontbeli normálvektort is [11].

```
// Az eltalált háromszög csúcspontjainak lekérdezése
uint3 triangleIndices = UnityRayTracingFetchTriangleIndices(PrimitiveIndex());

// A csúcspontok normálvektorainak kiszámítása
float3 vertexNormal1 = UnityRayTracingFetchVertexAttribute3(
    triangleIndices.x, kVertexAttributeNormal);
float3 vertexNormal2 = UnityRayTracingFetchVertexAttribute3(
    triangleIndices.y, kVertexAttributeNormal);
float3 vertexNormal3 = UnityRayTracingFetchVertexAttribute3(
    triangleIndices.z, kVertexAttributeNormal);

// Baricentrikus koordináták kiszámolása
float3 barycentricCoordinates = float3(
      1.0 - attributeData.barycentrics.x - attributeData.barycentrics.y,
      attributeData.barycentrics.x,
      attributeData.barycentrics.y);

// Súlyozott normálvektor kiszámítása a baricentrikus koordinátákkal
float3 objectNormal = barycentricCoordinates.x * vertexNormal1
    + barycentricCoordinates.y * vertexNormal2
    + barycentricCoordinates.z * vertexNormal3;
```

Ezzel még nem vagyunk kész, mert az előző függvények a normálvektorokat **object space**-ben, az eltalált objektum saját koordináta-rendszerében adták meg. Nekünk ezt transzformálni kellene world space-be. Ehhez a szintén beépített ObjectToWorld3x4() függvény visszatérési értékéül adott transzformációs mátrixszal kell megszoroznunk a normálvektort:

```
float3x3 objectToWorld = (float3x3)ObjectToWorld3x4();
float3 worldNormal = normalize(mul(objectToWorld, objectNormal));
```

6.3 Törésirány

Most, hogy ismerjük az eltalált felület normálvektorát, már csak a törési irány megkeresése maradt hátra. Ehhez a ShaderObject tulajdonságai közé fel kell vennünk egy törésirány változót, hogy azt kényelmesen tudjuk állítani az anyag Inspector ablakából. Itt a sor elején a kódban használható változónév, aposztrófok között az Inspector-ban megjelenő név és beállítható intervallum, végül az alapértelmezett érték adható meg.

```
Properties
{
    _MaxBounce("Max bounce", Range(0, 16)) = 8;
    _IoR ("IoR", Range(0, 3)) = 1.5
}
```

Az épp feldolgozott sugár adatait is beépített függvényekkel érhetjük el. A WorldRayOrigin() a sugár kiindulási pontját, a WorldRayDirection() az irányát adja meg

world space koordinátákban, a RayTCurrent pedig az ütközésig megtett távolságot adja vissza. Ezek közül az irányra most, a többire később lesz szükségünk:

```
float3 rayOrigin = WorldRayOrigin();
float3 rayDirection = WorldRayDirection();
float3 worldPosition = rayOrigin + RayTCurrent() * rayDirection;
```

A törést leíró egyenlet szerencsére már meg lett írva HLSL-ben, és a refract() függvénnyel tudjuk használni. Paraméterként a bejövő sugárirányt, a felület normálvektorát és a törésmutatót kell átadni. Előtte fel kell készülni arra az esetre, hogy a sugár a háromszöget hátulról találja el (az objekum belsejéből). Ezt a normál- és irányvektor közti skaláris szorzással ellenőrizhetjük (negatív eredmény esetén a szög nagyobb, mint kilencven fok, tehát hátulról találta el). Rossz irány esetén egyszerűen megfordítjuk a normálvektort és a törésindex inverzével számolunk:

```
// Helyes törésmutató kiválasztása
float correctIoR = dot(rayDirection, worldNormal) < 0 ? 1 / _IoR : _IoR;

// Normálvektor esetleges megfordítása
worldNormal = dot(rayDirection, worldNormal) < 0 ? worldNormal : -worldNormal;

float3 refractionDirection = normalize(refract(
    rayDirection,
    correctNormal,
    correctIoR));
```

6.4 Új sugár indítása

Minden szükséges adat rendelkezésre áll ahhoz, hogy a törés irányába egy új sugarat indítsunk. Először egy új sugár objektumot kell létrehozni (RayDesc), majd egy rakományt (RayPayload), végül ezeket a TraceRay() függvénynek átadni.

```
// Új sugár adatai
RayDesc newRay;
newRay.Origin = worldPosition;
newRay.Direction = refractionDirection;
newRay.TMin = 0.001; // Kis eltolás, nehogy ugyanazzal a ponttal ütközzön
newRay.TMax = 100000;

// Új sugár rakománya
RayPayload newRayPayload;
newRayPayload.color = float3(0.0, 0.0, 0.0);
newRayPayload.depth = rayPayload.depth + 1;

// Az új sugár elindítása
TraceRay(_RaytracingAccelerationStructure, RAY_FLAG_NONE, 0xFF, 0, 1, 0,
    newRay, newRayPayload);

// Eredmény visszaadása a beérkező sugárnak
rayPayload.color = _Color * newRayPayload.color;
```

A visszakapott rakomány szín mezőjét végül átadjuk a beérkező sugárnak. Ha a törő objektumnak van saját színe, azzal itt megszorozhatjuk a visszakapott árnyalást, ezzel színezett üveget hozva létre.

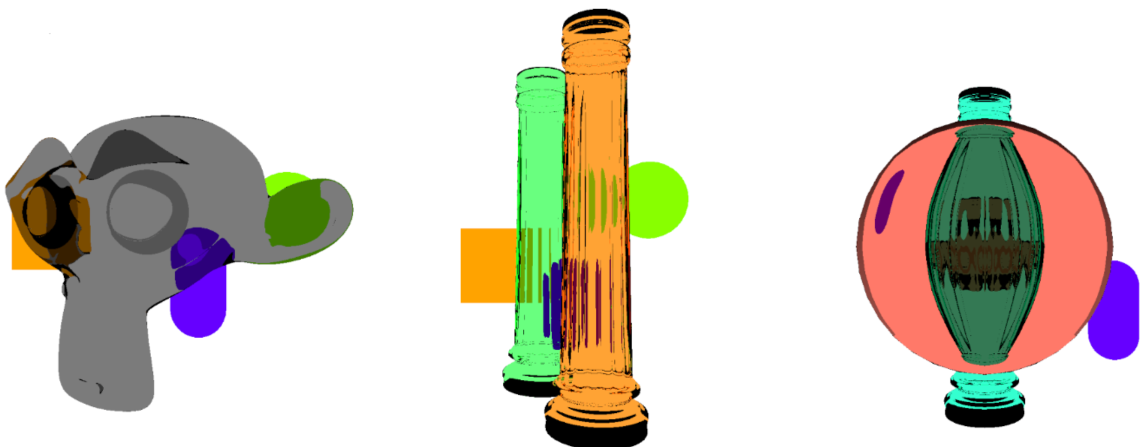
6.5 Egyszerű anyag

Ahhoz, hogy a törést szemléltetni lehessen, létre kell hozni egy másik anyagot, ami tartalmaz egy ugyanolyan nevű Pass-t, mint a törő objektum. Ehhez egyszerűen csak egy konstans színt visszaadó Closest-Hit Shader-t kell írni az üres ShaderObject-be:

```
Pass
{
    Name "RayTracing"
    Tags { "LightMode" = "RayTracing" }

    HLSLPROGRAM
    #pragma raytracing test
    #include "Common.cginc"

    [shader("closesthit")]
    void ClosestHit(inout RayPayload payload, AttributeData attrData)
    {
        rayPayload.color = float3(1.0, 1.0, 1.0);
    }

    ENDHLSL
}
```

Az anyag színe kiemelhető ShaderObject paraméterbe, így Unity-ből is könnyen állítható lesz, több példány létrehozása esetén pedig egyedi szín is megadható.

6.6 Eredmények



6.2 ábra: Saját sugárkövetéses árnyaló demonstrálása törő és egyszínű anyagokkal

6.3 ábra: Saját sugárkövetéses árnyaló különböző törésmutatókkal

A bemutatott árnyalóval renderelt jelenetek a 6.2 és 6.3 ábrákon látszanak. Megfigyelhető, hogy ez az egyszerű shader is elég valósághű eredményt ad. Nagy előnye a Unity-vel érkező screen space töréshez képest, hogy bármilyen formájú objektumra alkalmazható, valamint a törő objektumok felszínén látszanak a jelenetben szereplő más törő objektumok, és a kijelzőn nem szereplő tárgyak is. Persze bőven lehetne tovább fejleszteni, például elnyelődést, tükröződést, érdességet hozzáadni a képlethez. A jelenlegi megoldás erősen limitálva van, mivel csak olyan shader-eket jelenít meg, amelyek a „RayTracing" Pass-t tartalmazzák, ráadásul az elkészült textúra csak a képernyő GUI (General User Interface, általános felhasználói felület) rétegén jelenik meg. Egy lehetséges bővítés a HDRP csővezetékbe integrálni ezt az effektust is.

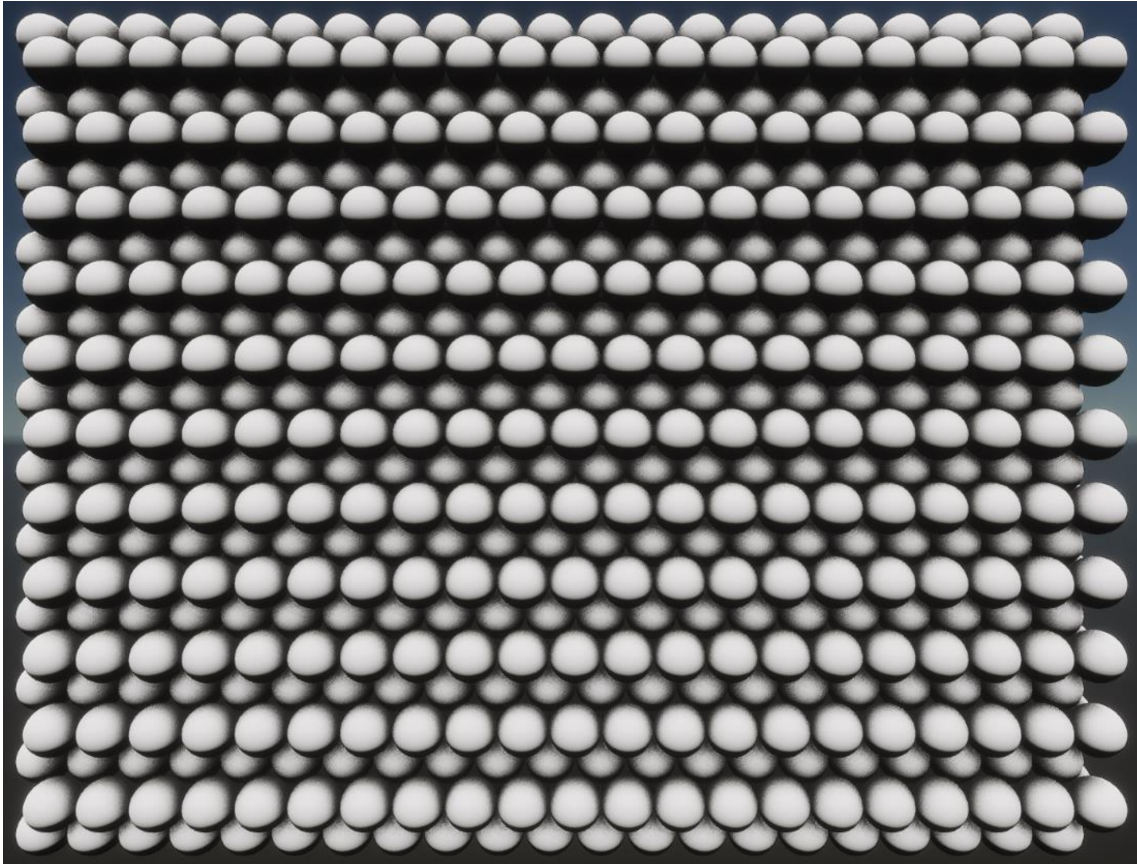
7 Fényeffektek tesztelése

A tesztek célja, hogy összehasonlítsa a Unity-vel érkező beépített screen space és sugárkövetéses megoldások hatékonyságát a különböző fényeffektusok esetén. A kapott eredmények alapján remélhetőleg választ kapunk arra, hogy egy éles projektben melyik technológiai megvalósítást alkalmazzuk a különböző fényjelenségek számításához.

A mérés alapjául az FPS adatok szolgáltak. A tesztek futtatásához készítettem egy jelenet-létrehozó szkriptet, ami tetszőlegesen paraméterezhető, ezzel felgyorsítva a tesztek eltérő beállításokkal való indítását. A teszteket különböző objektumszámmal is futtattam, szemléltetve ezzel az effektek függőségét a jelenetben egyszerre látható geometriák számától. A teszt alapjául szolgáló jelenetet adott számú gömb (a Unity beépített Sphere objektuma, ami közelítőleg 3600 háromszöglapból és 3000 darab csúcspontból áll) alkotta, szorosan kocka alakba rendezve. A gömb csak egy részletességi szinttel (Level Of Detail, LOD) rendelkezett, tehát a jelenetben egyszerre jelen lévő háromszöges és csúcspontok száma megegyezik a gömb ezen értékeinek és darabszámának szorzatával, függetlenül a kamera távolságától. A sűrű elhelyezkedés lehetővé tette az ambient occlusion, global illumination, árnyék, törés és tükröződés effektusok mérését ugyanazon a jeleneten. Utóbbi két effekt kivételével minden esetben fehér színű, diffúz anyagot használtam, míg a fénytöréshez egy tökéletesen tükröző, töréshez pedig tökéletesen törő anyag volt beállítva a testeken. Ahol nincs külön feltüntetve más, ott a jelenetben összesen egy darab irányított fényforrás adta a megvilágítást.

Adott effektus tesztelésekor mindkét technológia tesztjét ugyanazokkal a renderelési beállításokkal indítottam, a kamera pozíciója, a létrehozott jelenet, az effektek minőségi beállításai mind megegyeztek. Minden esetben az épp nem tesztelt effektusok beállításai ki voltak kapcsolva a HDRP eszközfájlban, így a játékmotor nem végzett fölösleges számításokat a háttérben. A tesztek futtatásához használt hardver egy Asus RTX 2070 videokártya volt, 1920x1080-as renderelési felbontásra állítva, a Unity 2021.1.21f verziójával. A tesztek az editor Game ablakában futottak bekapcsolt Maximize On Play beállítás mellett, ezzel limitálva az egyéb renderelési folyamatokat (például UI és Scene ablak). A kamera minden esetben ugyanolyan távolságra helyezkedett el a teszt során generált objektumoktól. A mért FPS értékek a program

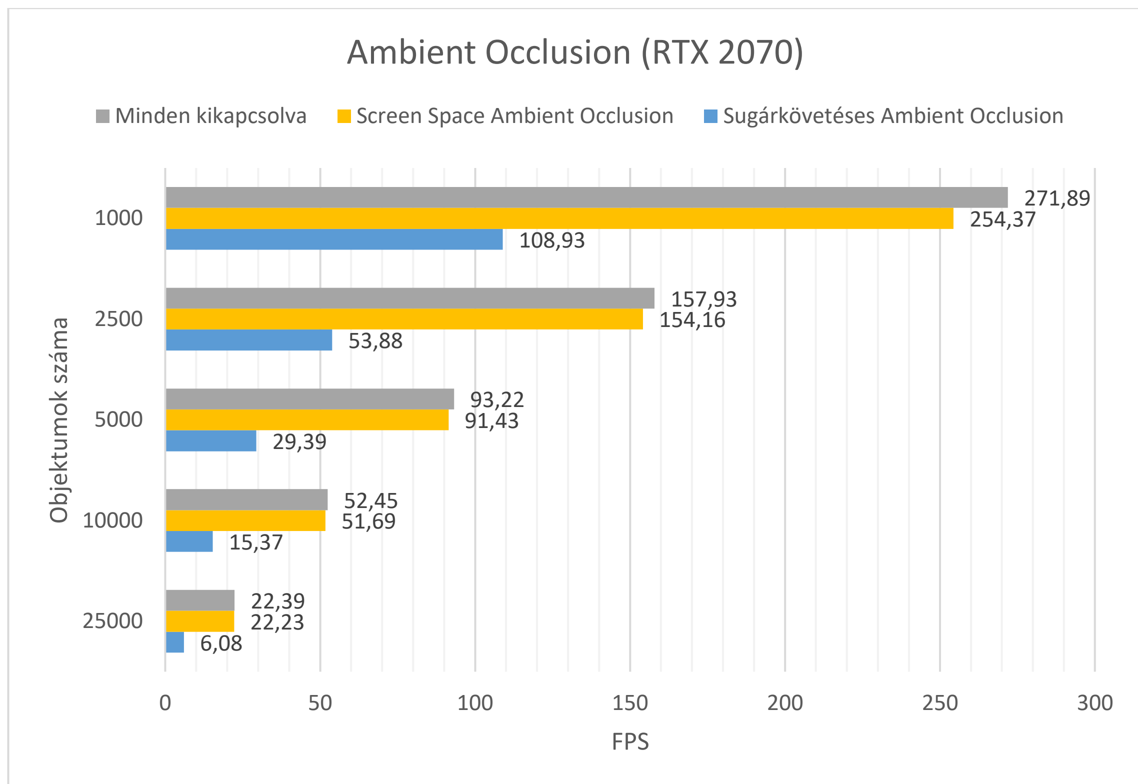
indítása után eltelt öt másodperctől kezdve, fél percen át tartó mintavételezés alatt gyűjtött képfrissítési adatok számtani közepe, két tizedes pontossággal kerekítve. Természetesen az adatok így sem voltak teljesen konzisztensek (egy mérés megismétlésekor időnként tapasztalható volt 1-2 százalékos eltérés), de ez olyan kismértékűnek bizonyult, hogy a tesztek eredményét nem befolyásolta.



7.1 ábra: Ambient occlusion teszt 10000 darab gömbbel

7.1 Ambient Occlusion

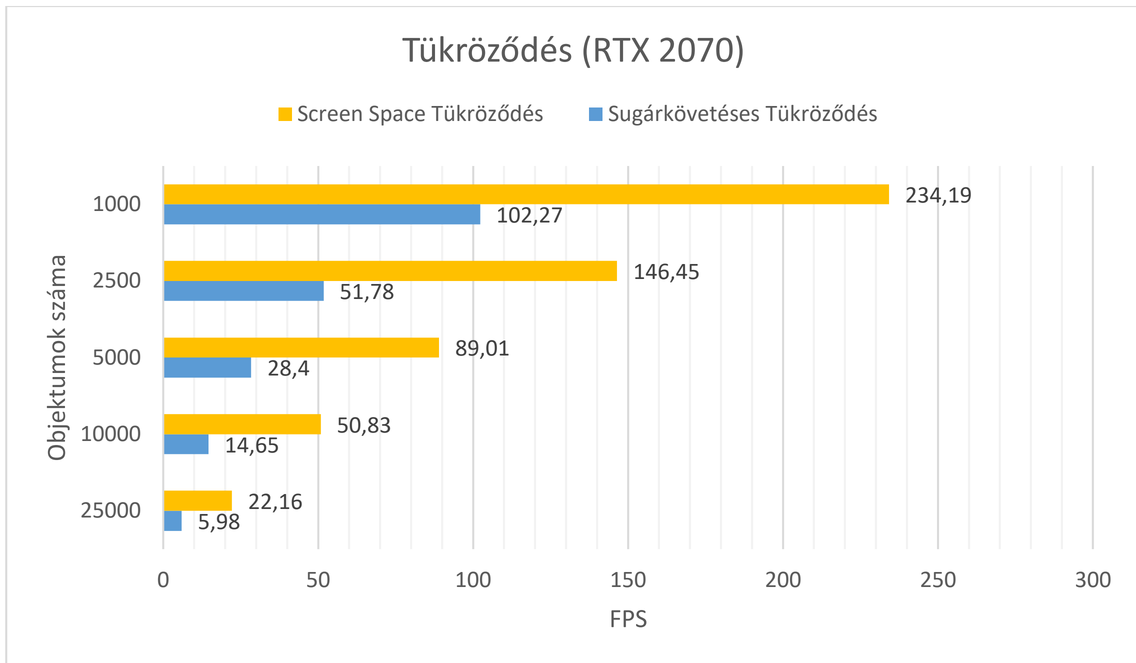
Az effekt a Unity alapértelmezett minőségi szintjei közül High-ra volt állítva. Az egyéb beállítások közül mindkét esetben az intenzitás értékének 2,5, a sugárnak 5, valamint a direkt fény erősségének 1 volt megadva. Az eredmények alapján a két technika közti különbség igencsak jelentős: a screen space ambient occlusion esetén átlagosan **207** százalékkal több lett a mért FPS. Összehasonlítási alapnak lemértem azt az esetet is, amikor semmilyen effekt nincs bekapcsolva (még a sugárkövetés támogatás se), csak az alakzatok egyszerű megjelenítése. Ezeket az adatokat mutatja a „Minden kikapcsolva" kategória.



7.2 ábra: Ambient occlusion teszteredmények

7.2 Tükröződés

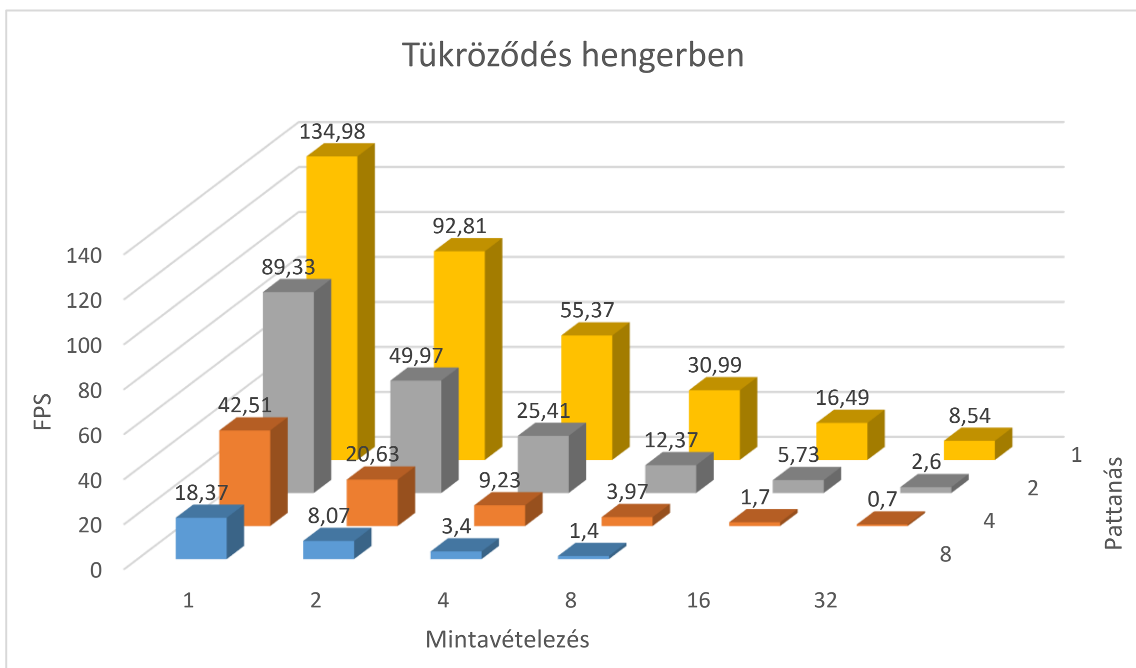
Ebben az esetben a gömbökön tükröző anyagokat használtam. Mindkét fényszámítási módszer esetén a legmagasabb beállítással készültek a mérések. Sugárkövetés esetén a maximális nyolc visszaverődés volt beállítva, mivel ez nem befolyásolta lényegesen a mért FPS-eket. (Érdekesség, hogy legtöbbször 8 bounce érték mellett átlagosan 1-2 százalékkal magasabb FPS-t mértem, ugyanazzal a jelenettel. Ez lehet, hogy csak mérési hiba, de több próbálkozás során is ezt az eredményt kaptam.) A mérést úgy is elvégeztem, hogy a gömbök között azok átmérőjével megegyező üres helyet hagytam, így a kamera számára láthatóak voltak az egyébként takarásban lévő objektumok is. Az így kapott eredmények csak elhanyagolható mértékben tértek el a sűrű jelenet méréseitől. Ennél az effektnél **204,3** százalék volt az átlagos gyorsulás



7.3 ábra: Tükröződés teszteredmények

7.2.1 Tükröződés hengerben

A sugárkövetéses tükrözés képességeinek további tesztelésére felállítottam egy nagyon egyszerű jelenetet, benne egy tükröző anyagú hengerrel, aminek belsejében egy megvilágított kockát helyeztem el. A tükrözés pattanásainak számát és mintavételezését kettőhatványok értékeire állítva próbáltam ki az effekt hatékonyságát. A kamera a henger belsejében volt és az oldalfalat nézte.



7.4 ábra: Tükröződés hengerben teszteredmények

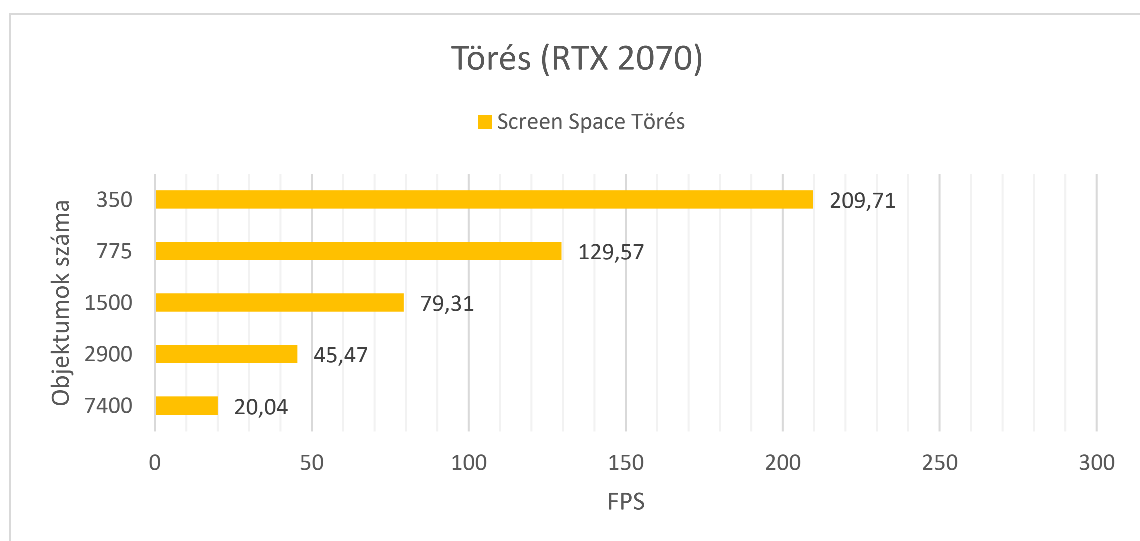
Látható, hogy bármelyik érték növelése nagyon erősen befolyásolja a számítási időt annak ellenére, hogy a jelenetben összesen csak 278 háromszög és 514 darab csúcspont volt. Az üresen maradt helyek esetében a Unity felmondta a szolgálatot. A drasztikus eredmények oka, hogy a kamerából kilőtt sugarak száma a mintavételezés értékével szorzódik, valamint egy sugár a pattanásszámmal megegyező darab új sugarat hoz létre az útja során. 1-1 értéknél pixelenként egy sugár jön létre, de 32-8 értéknél már pixelenként 32 x 8 = 256 darab sugár keletkezik. Ezt megszorozva a renderelt kép felbontásával több mint 530 millió sugarat kapunk képkockánként.

Persze nincs értelme az ilyen magas mintavételezésnek, hiszen a sugárkövetéssel számolt tükröződésekre rendelkezésre áll egy zajszűrő alkalmazása is, ami nagy terhet vesz le a videokártyáról. A maximális terhelés eléréséhez a teszt bekapcsolt szűrővel futott.

7.3 Törés

Most a gömbökre átlátszó anyag került. Miután engedélyeztem az eszközfájl Reflections beállítását, az anyagok tulajdonságai között megjelentek a törés paraméterei. Itt a törésmutató (Index of Refraction, IoR) állítgatásával lehet a törés mértékét befolyásolni, 1-es érték esetén nincs törés, afölött pedig egyre nagyobb szögben térülnek el az anyagba hatoló fénysugarak. A teszthez az 1.5 érték volt beállítva.

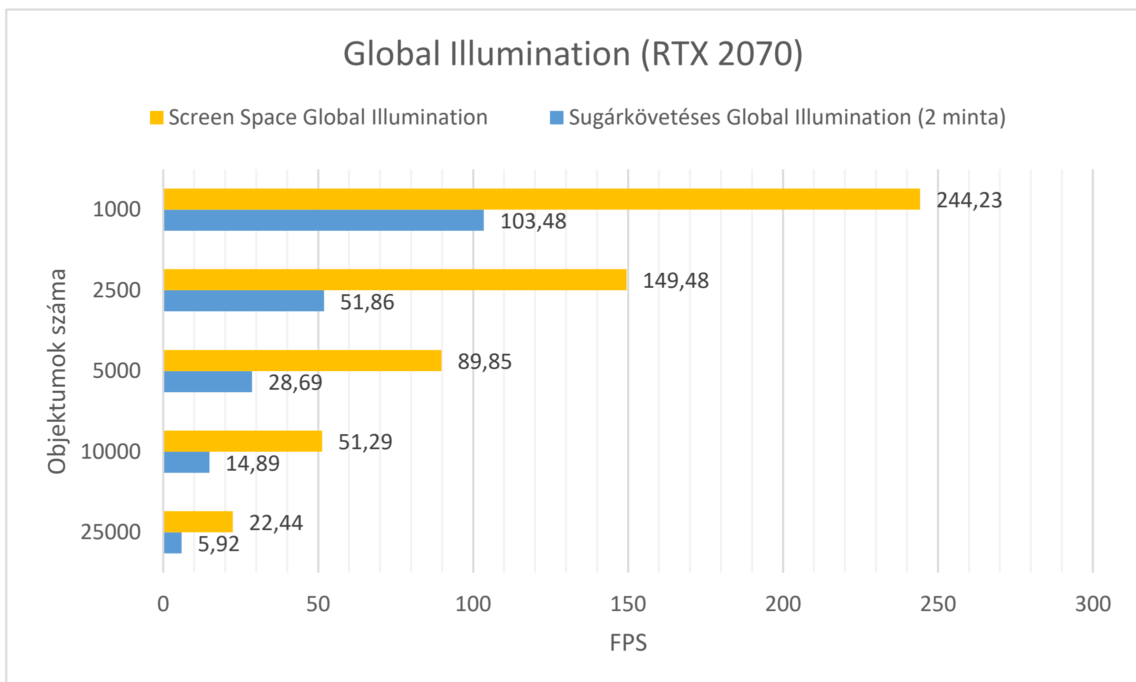
Az effektus korábban bemutatott tulajdonságai miatt (miszerint törő objektum képe nem látszik törő objektumon) a jelenetben minden második függőleges golyó réteg átlátszatlan anyaggal volt ellátva. Törésből csak a screen space típust támogatja a Unity.



7.5 ábra: Törés teszteredmények

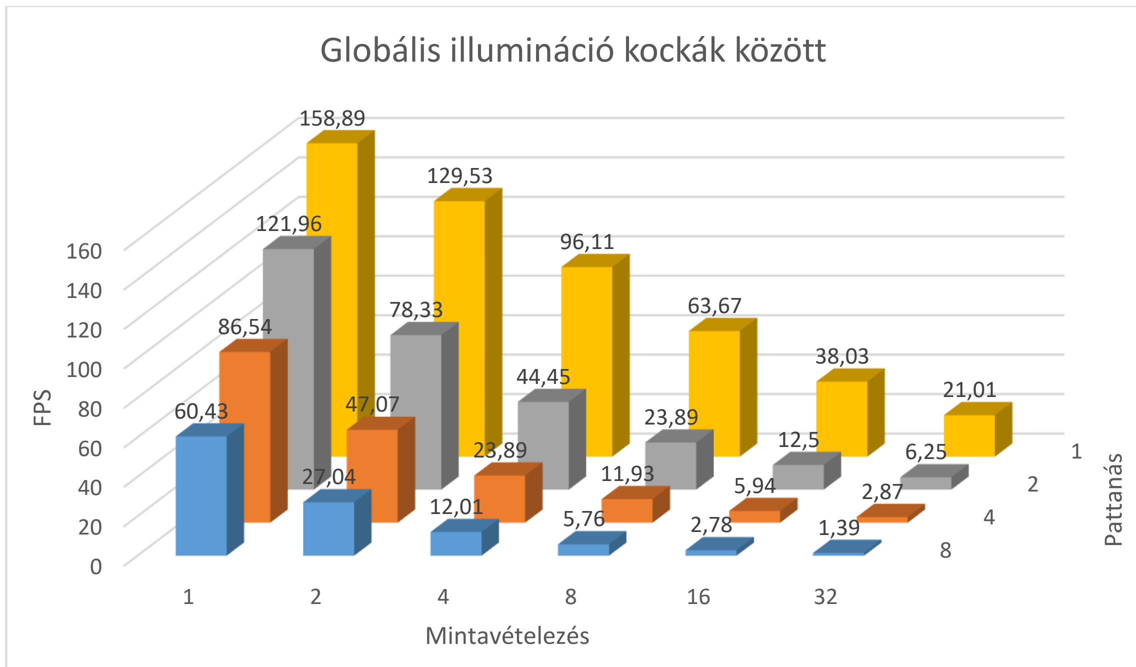
7.4 Global Illumination

A tesztek mind sugárkövetés, mind raszteres megjelenítés esetén ismét magas beállítások mellett futottak. A sugárkövetés bounce értéke itt is a maximális nyolcra volt állítva, mivel ez elhanyagolható (kevesebb, mint egy százalékos) különbséget jelentett a mért adatokban (bár ez a mérték természetesen jelenetfüggő lehet). Az indított minták száma pedig 2 volt, mivel ez is megfelelő minőségű képet állított elő. Egyébként utóbbi növelése a subsurface scattering-nél később tapasztalt drasztikus teljesítményromlást vonzza maga után. Itt a két módszer közti különbség **212** százalékos volt.



7.6 ábra: Globális illumináció teszteredmények

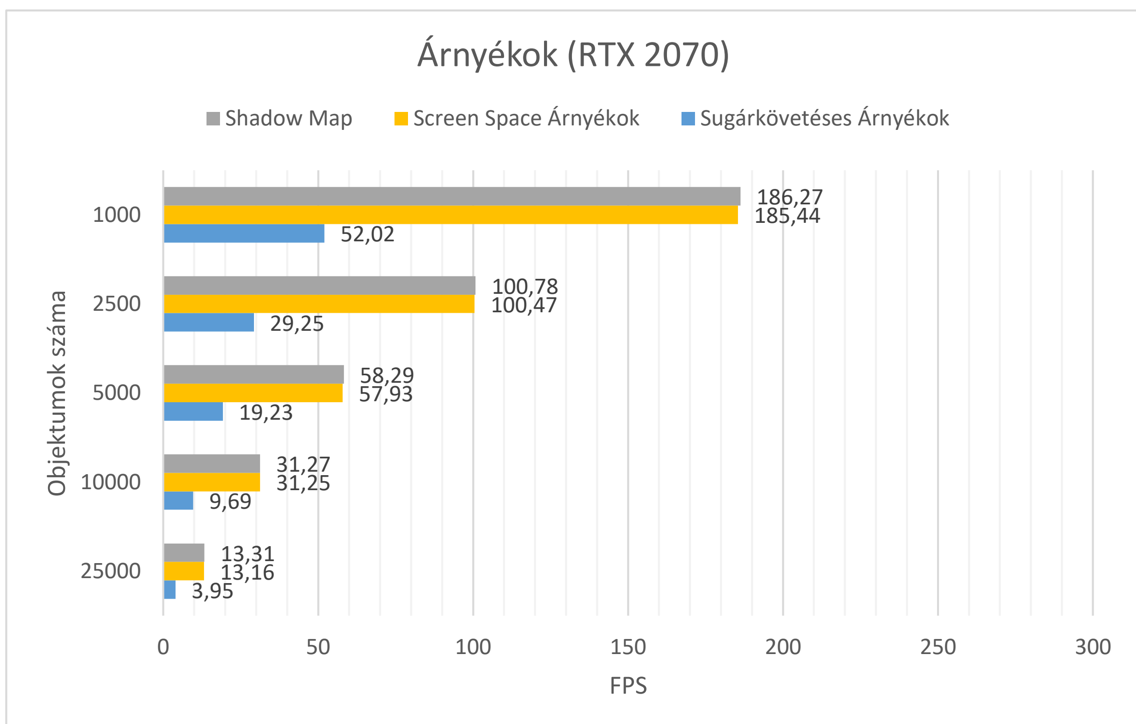
Az effektet leteszteltem egy másik jelenettel is: itt kockákat helyeztem el egymáshoz közel, hogy a létrehozott sugarak többszörösen is tükröződjenek a felületeken. Ugyanazon a jeleneten, különböző bounce és mintavételezés értékekre az alábbi eredményeket kaptam:



7.7 ábra: Kockák közötti globális illumináció teszt eredményei

7.5 Árnyékok

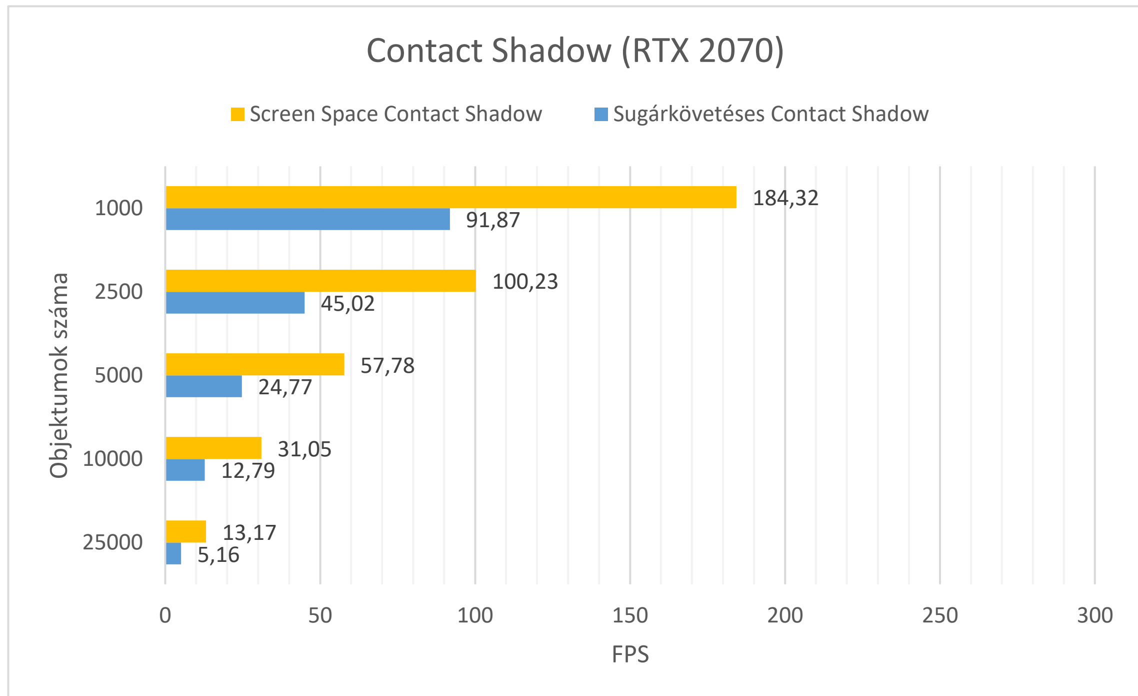
Az árnyékok mindhárom megvalósítási módjával lefuttattam a teszteket. Az eredmények alapján a screen space és shadow map technikák között érdemi különbség nem volt megfigyelhető. A raszteres módszer **231** százalékkal, a shadow map esetén **0,5** százalék volt a gyorsulás az előzőhöz képest.



7.8 ábra: Árnyék teszteredmények

7.6 Contact shadow

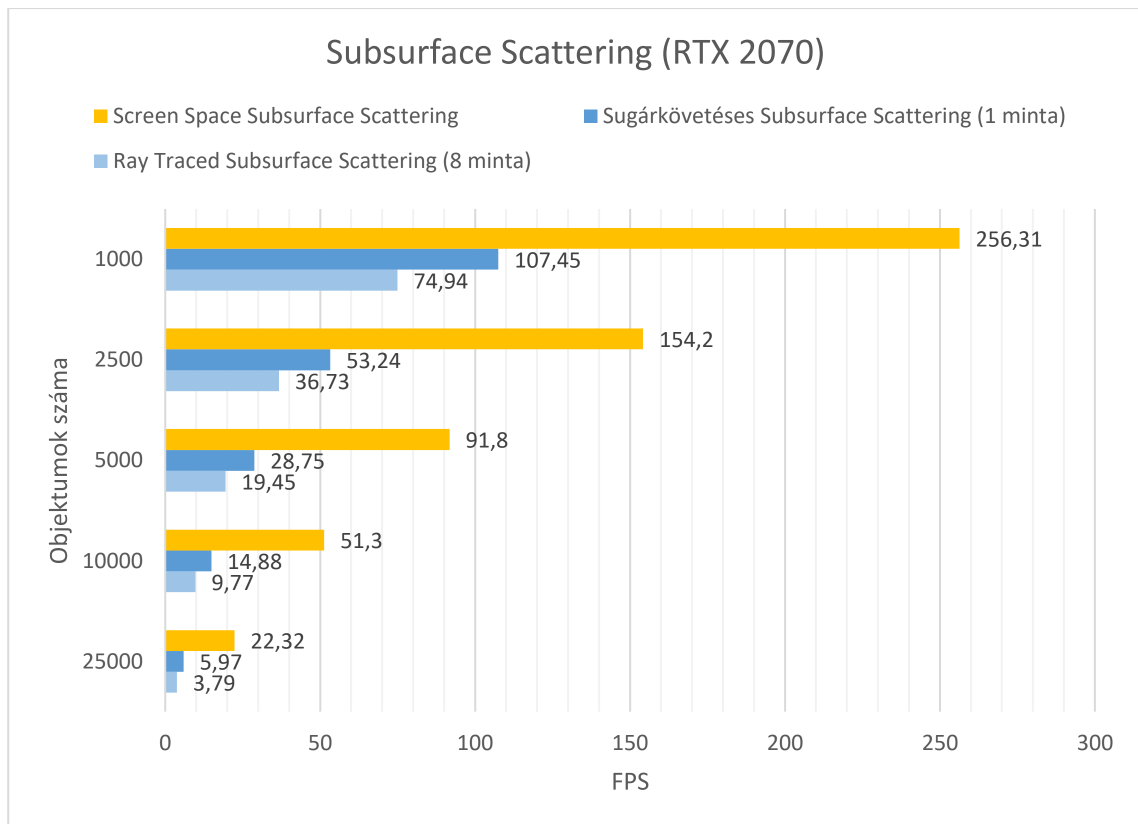
A teszt futtatásához a projektben és a fényforrásban engedélyezni kellett a shadow map-eket, különben a contact shadow sem került volna megjelenítésre. Ennél a tesztnél **131** százalékos volt a teljesítmény növekedés raszteres megjelenítés esetén.



7.9 ábra: Contact shadow teszteredmények

7.7 Subsurface Scattering

Ezt a mérést sugárkövetés esetén elvégeztem két különböző mértékű mintavételezéssel is (1 és 8 értékkel), hogy az ezek közötti különbséget is szemléltessem. A HDRP-vel importálásra kerülő, bőrt utánzó Diffusion Profile fájlt használtam az anyaghoz, amit a teszt során a gömbökre helyeztem. A screen space módszer **213** százalékkal volt gyorsabb az 1 mintás sugárkövetésnél, míg utóbbihoz képest a 8 mintás tesztek **32, 9** százalékos lassulást mutattak.



7.10 ábra: Subsurface scattering teszteredmények

7.8 Következtetés

A tesztek eredményei szinte mind azonos különbséget mutatnak a raszteres és sugárkövetéses megjelenítési módok teljesítménye között. Ebből kiindulva csak abban az esetben célszerű az utóbbi módszert alkalmazni, mikor az kiemelkedően jobb látványt nyújt az előbbinél. Contact shadow és ambient occlusion esetén sugárkövetésre szinte semmi szükség, mivel alig van észrevehető eltérés a két megoldás között, teljesítményben viszont elég nagy nyereséget jelent a screen space változat használata a másikhoz képest. Az árnyékok esetén is megfontolandó a raszteres megoldás használata. Töréseknél nem sok lehetőség közül választhatunk, ha beépített megoldásokról van szó. A tükröződések egy nehéz kérdés, mivel alkalmazásuk erősen jelenetfüggő. A 4. fejezetben bemutatott példák alapján, ha például vízfelület tükrözését akarjuk megvalósítani, a screen space tökéletes eredményt ad, míg egy ablakban vagy tükörben látott tükröződéshez előnyösebb sugárkövetést használni. Sajnos egyszerre ugyanazon a területen (Volume Override-on) belül a két technika alkalmazása nem kivitelezhető, mert erre nincs külön beállítás, erre külön Volume-okat kell létrehozni. Subsurface scattering esetén kellő megvilágításnál megint csak fölösleges az erőforrásigényes sugárkövetést választani, utóbbi csak sötét terekben tud látványos különbséget mutatni. Ezzel szöges ellentétben áll a globális

illumináció, ahol a raszteres változat meg sem közelíti a sugárkövetés alkotta képminőséget. Itt érdemes áldozni az erőforrásokból, ha a valósághűségre törekszünk, bár az effekt használata ügyes megvilágítással akár el is hagyható.

Látható volt, hogy a valós idejű sugárkövetés adott objektumszám fölött már nem nevezhető valós idejűnek. 10000 gömb fölött (ami nagyjából 36 millió háromszöget jelent) a legtöbb esetben 15 alá esik az FPS értéke, ami már nem látszik folytonosnak.

8 Deep Learning Super Sampling (DLSS)

A mélytanuláson alapuló mintavételezés (Deep Learning Super Sampling, DLSS) egy Nvidia által fejlesztett technológia [12] [13], mely lehetővé teszi modern grafikájú játékok magasabb felbontáson való futtatását az FPS szám csökkenése nélkül. Működésének alapja egy betanított mesterséges intelligenciára (MI) támaszkodó renderelés. Az MI segítségével egy alacsony minőségű képkockát valós időben tudunk felskálázni magasabb felbontásra, a képkockaszám drasztikus csökkenése nélkül. A skálázott és a DLSS nélkül magas felbontáson renderelt képek között a különbségek szinte észrevehetetlenek. A videokártyáról ezzel elég teher kerül le ahhoz, hogy akár magasabb grafikus beállításokkal futtassuk a programot ugyanazon a felbontáson és FPS szám mellett.

A technológia 2019 februárjában jelent meg. Az első változatokat még játékonként külön kellett betanítani a kívánt minőség eléréséhez, ezért eleinte nagyon kevés játék támogatta. 2019 augusztusában a Remedy Entertainment által fejlesztett Control című játékban – egyéb áttörő grafikai megoldások mellett – először használták a DLSS 2.0 korai változatát, ami, bár még mindig nem volt általánosan alkalmazható, rengeteg javítást tartalmazott. Később, 2020 áprilisában jelent meg ténylegesen a 2.0 verzió, ami a fejlesztő cég szerint már alkalmas arra, hogy tetszőleges projektben alkalmazzák külön tanítás nélkül.

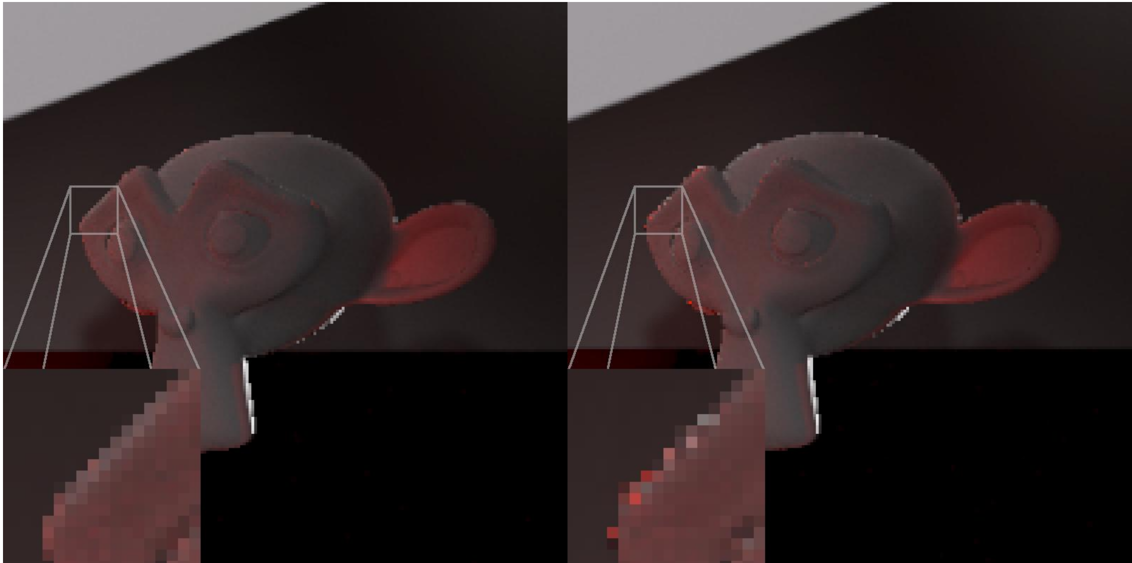
A DLSS az Nvidia saját gyártású videokártyái közül is csak azok esetében érhető el, amelyek rendelkeznek úgynevezett Tensor processzormaggal, mivel ez felelős a technológia futtatásáért. Ilyen GPU-k a 2000-es és 3000-es sorozat tagjai.

8.1 DLSS Unity-ben

Unity-ben a DLSS a HDRP 12.0.0 verziójában debütált. Használatához a Package Manager-ben le kell tölteni az NVIDIA nevű csomagot. Ekkor a HDRP eszközfájlban a dinamikus felbontás (*Rendering → Dynamic Resolution*) bekapcsolásával elérhetővé válik az Enable DLSS opció, aminek kipipálásával aktiváljuk a szolgáltatást a projektben. A gomb alatt megjelenő beállításokkal konfigurálni tudjuk a mintavételezés működését. Az eredmény akkor válik láthatóvá, ha először – a fényeffektusokhoz hasonlóan – a kívánt kamerákon is engedélyezzük a DLSS működését. Ehhez a kamera Inspector

ablakán belül a *Rendering → Allow Dynamic Resolution* bekapcsolása után megjelenő Allow DLSS menüpont mellé kell pipát tenni.

8.2 DLSS tesztek



8.1 ábra: Ki- és bekapcsolt DLSS

Ezzel az effekttel is futtattam FPS teszteket, de azt vettem észre, hogy nagyon kevés (egy-két százalékos), vagy szinte semmilyen javulás nem tapasztalható a be- és kikapcsolt DLSS között. Bekapcsolt állapotban megfigyelhető volt némi minőségbeli különbség. Ahogy a 8.1 ábra ezt szemlélteti, az objektumok szélei kissé pontatlanok voltak. Emellett minimális szinten, de ugrált a kép, vagyis képkockánként néhány pixeles eltolódások voltak érzékelhetőek.

9 Demonstráció

A meglévő adatokkal a feladat még egy olyan jelenet létrehozása, melyben a fényeffektusokhoz elérhető megvalósítási módok közül azok a változatok kerülnek felhasználásra, amik a megjelenítés és teljesítmény között megfelelő kompromisszumot biztosítanak. Láthattuk, hogy a sugárkövetés milyen jelentős teljesítmény-veszteséggel jár a valósághű megjelenítés javára, így ezt a technikát csak abban az esetben ajánlott használni, ha a többi módszer képi világa meg sem közelíti a kívánt eredményt.

Ezek tudatában a sugárkövetésen alapuló ambient occlusion levehető a listáról, mivel a közel két és félszer hatékonyabb, screen space megvalósítás is megfelelő árnyalást eredményezett. Hasonló a helyzet a contact shadow-nál is, ennél sem volt tapasztalható jelentősebb különbség, az átlagos felhasználó pedig nem valószínű, hogy az ilyen apró részleteket észrevenné.

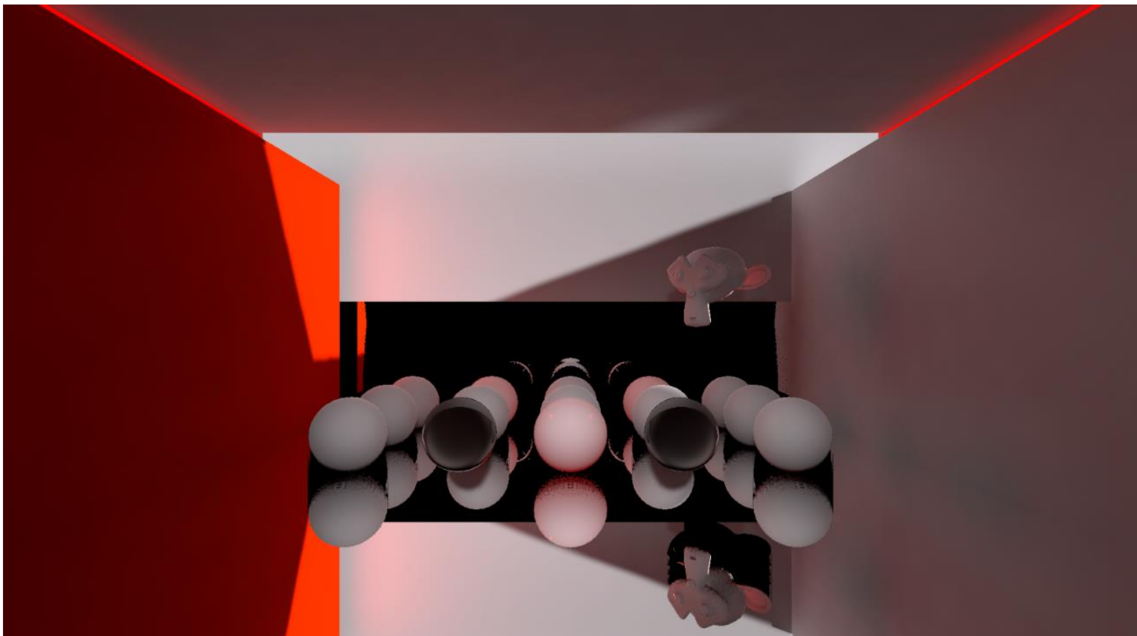
A 9.1 ábrán látható jelenetbe minden említett effektet belesűrítettem. A kamera egy szobában van, melynek négy fala közül három tükröződésmentes anyag, ebből kettő fehér, egy piros. A negyedik, szemközti falon és a padlón egy tükör van elhelyezve. A tető szintén fehér diffúz anyag, és tartalmaz egy nyílást az irányított fényforrás jelenetbe jutásához. Még két spotlámpa is el van helyezve: egy a majomfej mögött, egy a kamera háta mögött. A majomfej a subsurface scattering effektet valósítja meg, a húsz gömb közül kettő törő, három pedig fénykibocsájtó anyaggal van ellátva, a többihez a fehér falak anyagát használtam. Sajnos a törő anyagra csak a raszteres technikát tudtam alkalmazni a 6.6 fejezetben leírtak miatt. A jelenet két oldalsó falának tetején egy rés található egy-egy fénykibocsájtó piros lámpacsíknak.

Az első képen csak raytracing, a másodikon csak screen space technikák voltak beállítva. Előbbi esetében az effektek minőségét próbáltam elfogadható szinten tartani, de így is az átlagos FPS mindössze 3,9 volt, míg utóbbinál 44,3, ami óriási különbség, tekintettel arra, hogy a jelenet kevesebb, mint 700.000 háromszöget tartalmazott. Minőségben viszont a nagy fölény mellett is a raszteres változat vezetett, több szempontból is: nem kellett nagy kompromisszumokat kötni, a tükröződések sokkal szebbek is lettek, mivel sugárkövetés esetén a globális illumináció tükrözése ki volt kapcsolva. Screen space esetén a globális illumináció is kielégítő eredményt adott, de még a subsurface scattering-nél sincs szembeötlő különbség. Sugárkövetés esetén

előfordultak fény leak-ek a falak sarkaiban, ez a baloldali kép jobbalsó részének árnyékos területén jól látszik

9.1 ábra: Jelenet csak sugárkövetéssel (balra) és csak raszterizálással (jobbra)

A kapott eredmények alapján végül csak a globális illuminációt tartottam meg sugárkövetésre állítva (9.2 ábra), mert ennél az effektnél tapasztaltam a legkisebb teljesítményromlást a többi effekt közül, és okozza az egyik leglátványosabb különbséget. Érdekes megfigyelés, hogy már ez az egy effekt raytracing-re váltása leviszi az FPS értékét 8,8-ra.



9.2 ábra: Hibrid renderelés eredménye

Fontos megjegyezni, hogy a tapasztalt számoknál sokkal jobbakat is el lehet érni egy adott jelenetre az effektek megfelelő finomhangolásával. Jelen esetben a cél a különböző technikák hatékonyságának összehasonlítása volt.

10 Összegzés

A szakdolgozatnak köszönhetően lehetőségem volt részletesen megismerkedni a valós idejű renderelés típusaival, aminek köszönhetően úgy érzem sokkal jobban átlátom, mi áll egy játékmotor leegyszerűsített grafikus felülete mögött. Betekintést nyertem a fontosabb grafikus effektek működési elvébe több technikán keresztül, megismertem a Unity shader-ek felépítését, és a HLSL nyelvet is magabiztosabban tudom már használni.

Ezelőtt a Unity-ben inkább csak a szkriptek működésében és írásában éreztem otthon magam, de a dolgozat eredményeképp már a grafikus beállítások között sem érzem magam elveszettnek. Természetesen a szakdolgozat témám közel sem érintett minden területet a grafikus megjelenés világában, így bőven maradt még lehetőség tovább fejlődni, új módszereket megismerni, a meglévőket jobban tanulmányozni, esetleg továbbfejleszteni.

A tapasztaltak szerint a sugárkövetés támogatása Unity-n belül még nem igazán kiforrott, nem lehet például saját Intersection Shader-t létrehozni. A játékmotorral érkező, szabadon elérhető .hlsl és .raytrace fájlokban sok helyen találhatók „ez javítandó” és „ez így valószínűleg nem jó” kommentek, amiből az látszik, hogy még a fejlesztők is csak kísérleteznek a rendszerrel. A hivatalos dokumentáció sem fogalmaz túl bőbeszédűen, ha a sugárkövetéshez szükséges függvények használatáról van szó, ezzel eléggé megnehezítve azok működésének megértését. Sok a bizonytalanság a már meglévő függvények és beépített típusok körül is: egyes funkciók egyik verzióról a másikra megszűnnek létezni, és helyette nem biztosítanak megfelelő alternatívát.

Összességében örülök, hogy egy ilyen új, még alakulóban lévő technológiát volt lehetőségem kipróbálni, az útközben felmerülő nehézségek ellenére, mivel rengeteg tapasztalatot gyűjtöttem, ami a későbbiekben biztosan a hasznomra válik.

11 Irodalomjegyzék

[1] L. Szirmay-Kalos, G. Antal és F. Csonka, Háromdimenziós grafika, animáció és játékfejlesztés, ComputerBooks, 2006.

[2] Wikipedia, „Ray tracing,” [Online]. Available: https://en.wikipedia.org/wiki/Ray_tracing_(graphics). [Hozzáférés dátuma: 28 november 2021].

[3] Wikipedia, „Bounding Volume Hierarchy,” [Online]. Available: https://en.wikipedia.org/wiki/Bounding_volume_hierarchy. [Hozzáférés dátuma: 28 november 2021].

[4] R. L. Cook, T. Porter és L. Carpenter, „Distributed ray tracing,” in *Computer graphics and interactive techniques*, 1984.

[5] LearnOpenGL, „Coordinate Systems,” [Online]. Available: https://learnopengl.com/Getting-started/Coordinate-Systems. [Hozzáférés dátuma: 5 december 2021].

[6] Microsoft, „Graphics Pipeline Documentation,” [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline. [Hozzáférés dátuma: 29 november 2021].

[7] Microsoft, „Direct3D 12 Raytracing HLSL Shaders,” [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-raytracing-hlsl-shaders. [Hozzáférés dátuma: 9 december 2021].

[8] C. Wyman, „Introduction to DirectX RayTracing,” [Online]. Available: http://intro-to-dxr.cwyman.org/. [Hozzáférés dátuma: 7 december 2021].

[9] Unity, „High Definition Render Pipeline overview,” [Online]. Available: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@11.0/manual/index.html. [Hozzáférés dátuma: 7 december 2021].

[10] Scratchapixel, „Barycentric Coordinates,” [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates. [Hozzáférés dátuma: 9 december 2021].

[11] M. Skalsky, „Simple DXR Path Tracer,” [Online]. Available: https://github.com/SlightlyMad/SimpleDxrPathTracer. [Hozzáférés dátuma: 9 december 2021].

[12] A. Edelstein, P. Jukarainen és A. Patney, „Truly next-gen: Adding deep learning to games and graphics,” in *NVIDIA Sponsored Sessions (Game Developer Conference)*, 2019.

[13] L. Xiao, S. Nouri, M. Chapman, A. Fix, D. Lanman és A. Kaplanyan, *Neural Supersampling for Real-time Rendering,* Transactions of Graphics, 2020.