Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Warp metódusok vizsgálata objektum központú remote rendering rendszerekben

TDK dolgozat

Készítette:

Szögi Péter

Konzulens:

Nagy Bálint György
Dóka János
dr. Sonkoly Balázs

2023

Tartalomjegyzék

Kivonat

A kiterjesztett valóság (Augmented Reality / AR) lehetőséget teremt, hogy a valóságot kiegészítsük virtuális tartalmakkal, információkkal, mely funkcionalitás újfajta élményt nyújthat a felhasználóknak. Bár napjainkban elsősorban a játékok jutnak eszébe az embereknek a kiterjesztett valóságról, fontos ipari jelentősége is van, felhasználják oktatásban, egészségügyben és számos gyártási folyamatot is támogatnak vele. Bár jelenleg a legelterjedtebb AR eszközök a telefonok, folyamatosan jelennek meg a cégek palettáin a kiterjesztett valóság szemüvegek, melyek jelentősen könnyebb használatot biztosítanak, és amennyiben ezeket a szemüvegeket az átlagemberek is elérhető áron tudják majd beszerezni, várható a technológia ugrásszerű elterjedése a társadalomban.

Mivel a kiterjesztett valóság eszközök általában hordozható, könnyű súlyú készülékek, így nem rendelkeznek olyan erőforrásokkal, hogy a lehető legjobb élményt biztosítsák a felhasználók számára, hiszen az élethű nagy poligon számú objektumok renderelése a mai számítógépek többségének is komoly kihívást jelent. Erre megoldást jelenthet a felhő által biztosított számítási teljesítmény és grafikai erőforrások. Manapság egyre jobban elterjedt a remote rendering fogalma, mely során a számításigényes renderelést, vagyis a képalkotást, kiszervezik felhőbe, ezáltal például gyengébb számítógépekkel rendelkező felhasználók is élvezhetik a magas rendszerkövetelményű AAA játékokat. Ez a technika AR alkalmazásokra is átültethető, ám míg videojátékok esetében egy teljesen virtuális világ képét kell elküldeni a felhasználóknak, az AR alkalmazások esetében általában csak pár virtuális objektumot, melynek helyzete a valós környezethez van kötve, így az AR alkalmazások jóval érzékenyebbek a késleltetésre nézve. A ”motion sickness”-t vagyis azt a rosszullétet, hányingert amelyet az AR alkalmazások okozhatnak, jelentősen befolyásolja az, ha renderelt kép nem megfelelő helyen, vagy nem megfelelő perspektívából van megjelenítve. Erre megoldást jelenthet a kép warpolása, vagyis torzítása, mely segítségével csökkenthetjük a késleltetés által okozott kárt.

Warp technikákat lokális renderelés esetében is használnak az élmény javítására, ám kiemelten fontos remote renderelés esetében is, hiszen még nagyobb késleltetés hatását kell leküzdeni. A létező technikák különböző rendszerekre és problémákra vannak, és ”jóságuk” megállapítása igen nehézkes.

A dolgozatban javasoltam egy objektum központú remote rendering platformhoz (vagyis olyan remote renderelő rendszerhez, ahol minden objektumhoz külön képet kül-

dünk és nem egyben a felhasználó által látott képet) több warp technikát, valamint kidolgoztam egy objektív kiértékelést lehetővé tevő vizsgálati módszertant a szükséges metrikákkal, amelyek segítségével elemezhetők az algoritmusok. Ehhez szükséges volt feltárni az objektív és szubjektív jellemzők közötti összefüggéseket is, amihez szubjektív teszteket is végeztem. Létrehoztam egy új szimulációs keretrendszert, és részletesen kiértékeltem vele az implementált warp megoldásokat, feltártam az egyes megközelítések előnyeit és hátrányait.

Abstract

The concept of Augmented Reality (AR) creates an opportunity to enhance reality with virtual content and information, offering a new kind of experience for users. While most people associate AR primarily with gaming today, it also holds significant industrial importance, being used in education, healthcare, and supporting various manufacturing processes. Although smartphones are currently the most prevalent AR devices, augmented reality glasses are continuously emerging in the market, providing more user-friendly options. As these glasses become more affordable to the average consumer, we can expect a significant adoption of the technology in society.

Since AR devices are generally lightweight and portable, they lack the computational resources to provide the best possible experience to users. Rendering highly detailed objects with a large number of polygons is a challenging task even for most modern computers. To address this, cloud-based computing power and graphical resources are increasingly utilized. The concept of remote rendering, where computationally intensive rendering is offloaded to the cloud, allows users with less powerful devices to enjoy high-end AAA games, for example. This technique can also be applied to AR applications. However, while video games typically transmit an entire virtual world to users, AR applications usually involve only a few virtual objects, tightly integrated into the real environment. This makes AR applications more sensitive to latency. The "motion sickness" or discomfort that AR applications can cause is significantly affected when the rendered image is not displayed in the right place or from the proper perspective. Warping the image, i.e., distorting it, can help reduce the damage caused by latency.

Warp techniques are used to enhance the experience in both local rendering and, particularly, in remote rendering scenarios where overcoming greater latency is crucial. Existing techniques vary in terms of the systems they are designed for and the problems they aim to solve. Evaluating their effectiveness can be challenging.

In this thesis, I proposed several warp techniques for an object-centric remote rendering platform (where each object is separately rendered, rather than sending a single image seen by the user). I also developed an objective metric to assess these algorithms. I created a simulation system and evaluated the implemented warp solutions using it.

1. fejezet

Bevezető

A kiterjesztett valóság (Augmented Reality, AR) a valóság virtuális kiterjesztése valamilyen rétegen, általában kijelzőn keresztül. Lehet nem az első gondolata az embereknek, de ez az iparban is nagy jelentőségű technológia. Felhasználható az oktatásban, egészségügyben és a gyártósoroknál is. Ahogy elérhető áron kezdenek megjelenni a kiterjesztett valóság szemüvegek piacon, a mindennapokban is egyre inkább használhatóvá válik a technológia, ezért ugrásszerű elterjedésre számíthatunk a jövőben.

A kényelem és hordozhatóság érdekében a virtuális valóság (Virtual Reality, VR), kiterjesztett valóság (AR) vagy összefoglaló néven XR (Extended Reality) eszközök általában könnyűek, ennek okán viszont nem rendelkeznek megfelelő erőforrásokkal nagy poligon számú objektumok megjelenítéséhez. Az ilyen alkalmazásokhoz használt fejre rögzíthető kijelző (Head-mounted display, HMD) eszközöknél már jó ideje folyamatosan aktuális nehézség a teljesítmény és önálló (standalone) működés összeegyeztetése. Ez utóbbi alatt a felhasználó szabad mozgásának biztosítását érthetjük, hogy például nem korlátozza a mozgásban egy személyi számítógéphez rögzített kábel. A két nehézség természetesen összefügg, hiszen ha akkumulátorról szeretnénk működtetni egy eszközt, akkor érdemes minimalizálni az energiafelhasználást. Viszont ez nehézen megoldható, mivel egy XR alkalmazásnál az egészen kicsi kijelzésbeli hibák, késleltetés és akadozás is drasztikus mértékben rontja a felhasználói élményt. A mérnöki probléma mintapéldája.

A fennálló probléma megoldására számos cég különböző technológiákat fejlesztett ki és implementált az eszközeikbe. Ezek az időwarp (Timewarp) algoritmusok. Sok fajtája létezik, amelyek közül bemutatom a leginkább témához kapcsolódó és releváns megvalósításokat a következő fejezetben.

Egy másik megközelítése a megoldásnak, ha a renderelést a felhő által biztosított erőforrások végzik. Egy játékoknál már gyakran igénybevett szolgáltatás a remote rendering, ahol a felhasználó által látott teljes képet a szerver rendereli, majd leküldi a kliensnek egy adatfolyamon. Virtuális valóság (VR) alkalmazásokban még esetleg használható megoldás, azonban kiterjesztett valóságnál a valós teret kell vegyíteni a virtuálissal, ezért szükségesnek látszik egy alfaja, vagy másik verziója a remote renderingnek.

Egy kísérleti fázisban lévő, felhasználó élmény és megjelenítési minőség javításának érdekében létrejött technológia az objektum központú távoli renderelés (object-centric remote rendering). A lehetőségekhez mérten egyszerűen megfogalmazva annyit jelent, hogy a kliensek az objektumokról csak képet jelenítenek meg, melyeknek a renderelése a szerveren történik. Tehát a szerver minden pillanatban elküldi a kliensnek kép formájában, hogy az mit látna, ha az objektum ott lenne lokálisan előtte. Eme leírás leginkább csak a remote rendering részét magyarázza a fogalomnak, amiből két fő fajtát különböztetünk meg. Egyik, amikor a felhasználó által látott teljes kép kerül továbbításra. Ez jelenleg is használatban van, a távoli játék szolgáltatások lényegében így működnek. A másik pedig, amikor csak egy darab objektumhoz tartozó képet kap a kliens, minden más ténylegesen lokális, vagy egy másik streamből származó remote renderelt kép. Ez utóbbi kapja a fő hangsúlyt a dolgozatomban, ugyanis három újszerű metódust vizsgálok meg és javaslok egy vizsgálati módszertant azok jóságának mérésére.

A továbbiakban a 2. fejezetben bemutatom a témához kapcsolódó elméleti tudnivalókat és a technológiat hátteret. A 3. fejezetben ismertetem a objektum központú remote renderelés, valamint a vizsgált technikák és azok kiértékelésének tervét, majd a 4. fejezetben taglalom az implementációs részleteket. Az eredményeimet a 5. fejezetben értékelem ki, majd összegzem a munkámat a 6. fejezetben és részletezem a továbbfejlesztési lehetőségeket.

2. fejezet

Elméleti és technológiai háttér

A következő fejezet részletes áttekintést ad a kutatási munka elméleti és technológiai hátteréről. Egyrészt bemutatásra kerülnek a kiterjesztett valóság különböző változataiban létfontosságú mechanizmusnak számító warp metódusok, amik fő célja a felhasználói élmény javítása a megjelenítendő képkockák utólagos, megjelenítés előtti „torzításával" vagy mesterséges képkockák generálásával. Ilyenkor a torzítással javítjuk a képet, mert azt a felhasználó aktuális (renderelés utáni) helyzetéhez igazítjuk. Az áttekintett, általánosan előforduló problémákra közelmúltban javasolt megoldások, mint pl. az aszinkron időtorzítás (Asynchronous Timewarp, ATW), aszinkron tértorzítás (Asynchronous Spacewarp, ASW) vagy aszinkron újravetítés (Asynchronous Reprojection, ASR) és ezek különböző variánsai, sok esetben még aktív kutatás alatt levő témák. Másrészt ez a fejezet ismerteti röviden a legfontosabb vonatkozó szoftvereket és felhasználható szoftver könyvtárakat.

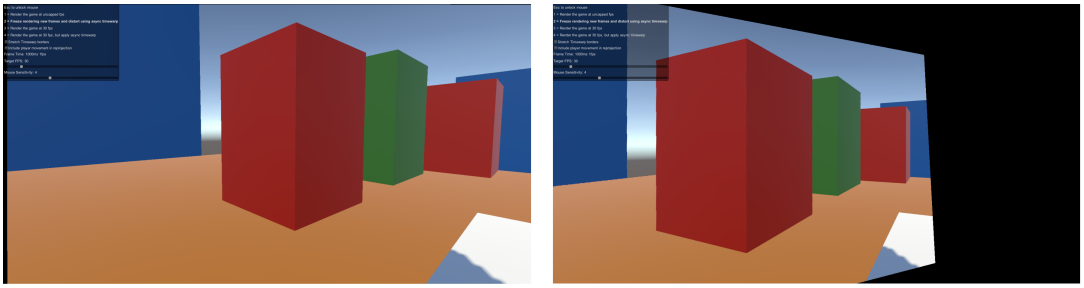
2.1. Kiterjesztett valóság különböző változatai

A kiterjesztett valóság különböző változataira összefoglaló néven az Extended Reality (XR) angol kifejezéssel szoktunk hivatkozni. Ez egyrészt magában foglalja a kiforrottabb, hosszabb ideje létező virtuális valóság (VR) alkalmazásokat. Itt a felhasználó teljes egészében el van zárva a külvilágtól, tipikusan egy fejre csatolható HMD eszköz segítségével egy vetített, virtuális környezet kerül valós időben megkelenítésre, amiben különböző kontrollerek segítségével interakció alakítható ki virtuális elemekkel. Ilyen eszköz például a Meta Oculus Quest termékcsalád. Másrészt az XR tartalmazza a kiterjesztett valóság (AR) különböző megjelenési formáit, amikor a felhasználó látja a saját fizikai környezetét és ez van kiegészítve virtuális objektumokkal (amikre sokszor, nem feltétlenül precízen hologramként szoktak hivatkozni). Ennek a technológiának a legbonyolultabb változata, az ún. „optical see-through" eszközökben érhető el, amikor a felhasználó optikai úton látja a saját környezetét és a virtuális tartalom ehhez igazodva valamilyen lencsén jelenik meg. Egy ilyen eszköz például a Microsoft Hololens 2 szemüvege. Egy fokkal egyszerűbb realizácót adnak a „video see-trough" eszközök, amikor a környező valóságot egy kamerán keresztül

érzékeljük és a végeredmény, ami a felhasználó kijelzőjén jelenik meg, ebből és a virtuális tartamakat hordozó video rétegekből áll össze. Ez megvalósítható például mobiltelefonok segítségével is (mobil AR). Újabban szokás megkülönböztetni még a kiterjesztett valóság egy következő generációs formáját, a kevert valóságot (Mixed Reality, MR), ami lehetővé teszi a fizikai és virtuális objektumok közötti interakciót is. Például a Hololens 2 eszköz támogatja ezt a fajta működést.

2.2. Asynchronous Timewarp (ATW)

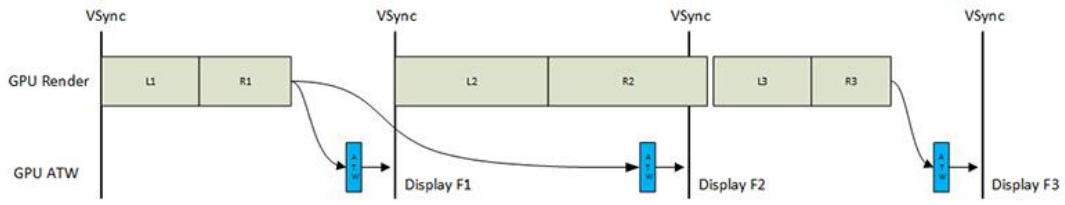
Az ATW egy olyan technika, amely a vizuális felhasználói élményt hivatott javítani abban az esetben, ha az alkalmazást futtató hardware nem képes konzisztens időközzel előállítani a képkockákat (frame-eket)[11]. Ennek kivitelezéséhez legfrissebb fej pozíció felhasználásával újravetíti az utolsó frame-ként szolgáló sztereoszkópikus képeket az elfordulás mértékével matematikailag torzítva[22].



2.1. ábra. ATW működése.

Ezt a 2.1 ábrán tekinthetjük meg. Bal oldalon látszik a legutolsó frame, míg jobb oldalon, hogy mi történne, ha két frame között el tudnánk fordítani a fejünket akár több mint 30 fokot. Aszinkron volta, ami a nevében is megjelenik abból eredeztethető, hogy külön szálon fut, a lehető legközelebb a megjelenítés idejéhez, hogy hatékonyan csökkentse a mozgástól-fotonig (motion-to-photon) késleltetést (lásd 2.2.1 fejezet). Minden V-Sync[1] előtt ez az aszinkron szál generál egy új, torzított frame-et a legutóbbi frame-ből [3]. Így pótolja az elejtett vagy kiesett (dropped) frame-eket, és csökkenti az ún. judder-t, tehát a kijelzőn megjelenő kép rezgését, ugrálását vagy szaggatott mozgását [5]. Erről a folyamatról jobb megértést nyújthat a 2.2 ábra. Ezen azt láthatjuk, hogy az ATW a megjelentés előtt fix idővel mindig elindul, és abban az esetben, ha a képgenerálása (renderelés) nem fejeződik be időben, mint ahogy az a középső „oszlopon” is látható, akkor az ATW által módosított előző frame kerül megjelenítésre.

[1] A Vertikális szinkronizáció vagy V-Sync egy grafikus technológia, amely szinkronizálja az alkalmazás képkocka per másodpercét (FPS) a kijelző frissítési frekvenciájával.



2.2. ábra. ATW működés közben [11].

2.2.1. Motion-To-Photon késleltetés (MPD)

Az MPD, aminek legismertebb elnevezése a Motion-to-Photon Delay, egy teljesítmény metrika VR alkalmazások számára. Azt a késleltetést jelenti, amíg a felhasználó fejmozgása megjelenik az alkalmazásban [7]. Tehát az eltelt idő az érzékelőkkel követett objektum mozgása és a grafikus kimeneti képernyőn megjelenő megfelelő mozgás között [17]. Érdemes ezt a késleltetést minél kisebbnek tartani a jó XR élmény érdekében, ugyanis az emberi vizuális rendszer akár pár pixeles hibát is képes érzékelni. Ha pedig sok hibát érzékel, mert nagy az MPD, az a felhasználóban kellemetlen érzéseket, vagy rosszullétet kelthet. Egy széles körben elfogadott felső korlátja a késleltetésnek a 15-20 ms [8].

2.3. Asynchronous Spacewarp (ASW)

Az ASW olyan esetek kezelésére lett létrehozva, amikor a rendszer nem képes stabilan tartani a kívánt képkocka per másodpercet (Frame Per Second / FPS), ami általában VR alkalmazásoknál (főleg játékoknál) 90 FPS. Célja a vizuális gördülékenység javítása az XR élmény során. Ez kísértetiesen hasonlít az ATW-re, de persze nem véletlenül, szándékosan épít rá. Míg az ATW csak a forgó mozgást kezeli és másra nem igazán alkalmas, az ASW a detektált animációt, kamera eltolást, karakter és játékos elmozdulást veszi górcső alá. Ezek figyelembevételével pedig közeli testvéréhez hasonlóan extrapolál egy új frame-et a legutolsó frame-ből. Ennek hatására a mozgások gördülékenyebbé válnak, és gyengébb eszközök számára is lehetőség nyílik kielégítő vizuális élmény nyújtása mellett futtatni korábban túlzottan megterhelő alkalmazásokat. Egy szerintem remek megfogalmazás, hogy az ATW jó, sőt tökéletes álló képeknél a fej forgatásának kezelésére, ilyenkor az ASW szükségtelen. Megfordítva pedig az ASW igen jól működik animált közeli objektumok esetén [25]. További hasonlóság az ATW és ASW között, hogy mindkét technológia automatikusan lép működésbe szükség esetén. Általában a HMD gyártó által telepített szoftver végzi ezeket a műveleteket, ezért a technológia használatához nem kell külön ráfordítás sem az XR az alkalmazás fejlesztőktől, sem a felhasználók részéről.

Működésének elve viszonylag egyszerű, és a 2.3 ábra nyújt segítséget a megértésében. Amennyiben egy alkalmazás nem képes tartani a stabil 90 FPS-t, akkor bekapcsol az ASW (nem megy folyamatosan) és lekorlátozza az alkalmazást 45 FPS-re. Ekkor minden máso-

2.3. ábra. ASW működés közben [20].

dik frame-et az ASW generálja. A metódus első generációja azonban elérte a képességei határát, ha még a 45 FPS-t sem volt képes tartani egy eszköz. További gyengéi [6]:

- Gyors fényerő változás
- Objektum újrafeltűnési nyomok (Object disocclusion trails): Az a jelenség, amikor egy korábban takarásban lévő objektum ismét láthatóvá válik, de nincsen elérhető adat arról, hogy minek kellene ott megjelennie. Ekkor az algoritmus a háttérben lévő teret nyújtja meg annyira, hogy betöltse az ürességet.
- Gyorsan mozgó ismétlődő minta, például sűrű kerítés futás közben. A mélységi adatok segítségével jobban számom tudja tartani az alkalmazás, hogy a példánál maradva a kerítés milyen gyorsan mozog a kerítésen keresztül látható térhez viszonyítva, ami viszont arányosan alig változik.
- Fejhez kötött elemek, mint például a szem elé vetített kijelzők (Head-up display, HUD). Például egy első személyes lövöldözős játéknál a térkép, a tárban lévő lövedékek száma vagy akár az életerő.

2.3.1. ASW 2.0

Pár év elmúltával az Oculus kiadta az ASW 2.0-át, az Asynchronous Spacewarp fejlesztett változatát. Azt állítják, hogy ez az új verzió már akkor is állja a sarat, ha az FPS 45 alá esne, de messze nem ez a legnagyobb újítása. Beleolvasztottak egy akkor még kísérleti technológiát, a helyzet alapú időtorzítást (Positional Timewarp, PTW). Így már 6 szabadságfokon (6DOF) javítja a fejre rögzíthető kijelző követési késleltetését akkor is, ha az ASW éppen nem aktív. Az utolsó nagyobb javulás is a PTW-hez kötődik, mégpedig amiatt, hogy a mélységi adatokat is figyelembe veszi ezáltal a technológia, amelyekről korábban semmit sem tudott. Ez az eredmény a sűrűn ismétlődő minták esetén látszódik igazán, például ismét egy kerítésnél. Ugyan szórványosan előforduló szituációnak hangoz-

hat, de négyzetrácsos vagy vonalas minták egész gyakran jelennek meg virtuális valóság alkalmazásokban [2, 13].

2.3.2. Positional Timewarp (PTW)

A PTW egy verziója az időtorzítási technikáknak, amely csökkenti mind a HMD forgásának, mind a helyváltozásának késleltetését is, ezért 6 szabadságfokú időtorzításnak (6DOF TW) is nevezik. Ennek véghezviteléhez szüksége van távolsági adatokra, amelyeket több módon is meghatározhat, például analitikus vagy raszterizált mélység alapján [4, 1].

2.4. Motion Smoothing

A mozgássimítás (Motion Smoothing), vagy gyakran mozgás interpoláció nem csak az XR világban található meg. A modern TV-kkel kapcsolatban talán még gyakrabban felmerülő kulcsszó, egy televízióba beépített technológia, amely a judder-t és elmosódást hivatott csökkenteni [15]. A SteamVR által használt Motion Smoothing az elejtett képkockák helyett (az ASW-hez nagyon hasonlóan) teljesen új képkockákat generál. Ehhez az utolsó két képkockából számítja ki, hogy hogyan kell kinéznie a soron következőnek [14, 18]. A TV-kben is nagyon hasonlóan működik, csak nem elejtett képkockát pótolnak így, hanem csak kiegészítik a korábban kis FPS-en rögzített felvételt. Ezáltal folyékonyabbá téve a mozgásokat, különösen tempós jeleneteknél.

2.5. Asynchronous Reprojection (ASR)

Az ASR egy grafikus technológia, amely minőségi élményt biztosít VR-ban egyenletes és judder mentes mozgás szolgáltatásával, még nagy rendszerterhelés mellett is [10]. Itt fontosnak tartom megjegyezni, hogy a terminológia elég zavaros, és az Asynchronous Reprojection alatt alapvetően az előző mondatban megfogalmazott definíciót értjük, ami egy tágabb fogalma a technológiának. Ezt több cég is implementálta a saját módján, amiből lett az ASR mint implementált technika a Google és Valve-tól, valamint ATW és ASW az Oculustól [24]. Így nem meglepő talán egyeseknek, hogy az ATW és ASR hasonlóak abban, hogy mindkettő technika geometrikus warpot használ a kieső frame-ek kompenzálására.

2.6. Unity

A Unity [21] egy már-már platformfüggetlennek nevezhető, de mindenesetre több platformot támogató játékmotor, amelyet a Unity Technologies fejlesztett, és adott ki először 2005-ben. Két- és háromdimenziós játékok fejlesztését is támogatja egészen elképesztő mennyiségű, több mint 19 különböző platformra. Nyitott a virtuális valóság és annak egyéb válfajai felé is, mint például a kiterjesztett valóság. A fejlesztőkörnyezete az egyik

leginkább felhasználóbarát a konkurens opciók közül, valamint C# az alapvető programozási nyelve, tovább erősítve az állítást. A cég sokáig nagy népszerűségnek örvendett, azonban egy rosszul fogadott üzleti döntés nem elhanyagolható erősségű ütést mért a nevére.

2.7. FFmpeg

Az FFmpeg [9] egy nyílt forráskódú szoftvercsomag, amelyet elsősorban multimédiás adatok kezelésére használnak. Tartalmazza könyvtárak és eszközök gyűjteményét különböző audio- és videófájlok dekódolásához, kódolásához, átkódolásához, lejátszáshoz és pár egyébhez. Az FFmpeg egy sokoldalú és nélkülözhetetlennek nevezett eszköz a multimédiás szakemberek számára. Ennek oka, hogy lehetővé teszi számukra a multimédiás tartalmak sokféle formátumba való konvertálását és kezelését. Használható parancssorból, vagy más alkalmazásokba integrálva olyan feladatok elvégzésére, mint többek között a videó- és hangszerkesztés.

2.8. OpenCV

Az OpenCV [19] (Open Source Computer Vision Library), nyílt forráskódú gépi látás könyvtáraként fordítható, de a továbbiakban OpenCV-ként hivatkozom rá. Ahogy a neve sugallja, ez egy publikusan elérhető forráskódú könyvtár, aminek segítségével különböző képfeldolgozási feladatok valósíthatóak meg, akár gépi tanulással. Több ezer algoritmust tartalmaz, amik között megtalálhatóak régi klasszikus és modern, még fejlődő algoritmusok is. Ezen eljárások lehetőséget adnak többek között arcok, tárgyak felismerésére, az emberi tevékenység felismerésére, mozgó tárgyak és emberek követésére, 3D modellek kinyerésére, 3D pontfelhők előállítására és még sok egyéb dologra.

2.9. Kiértékelési módszerek

A következő alfejezetekben röviden ismertetem a vizsgált technikák kiértékeléséhez felhasznált módszereket, mint az átlagos véleménypontszám (Mean Opinion Score, MOS), átlagos négyzetes hiba (Mean Square Error, MSE) és strukturális hasonlósági index mérték (Structural Similarity Index Measure, SSIM). Ezek közül egyedül SSIM kizárólag képkiértékelési módszer. Gyakran MSE mellett használják gépi tanulás modellekhez a pontossága miatt. A MOS szubjektív módszer lévén megfelelő kreativitással sok szakterületen felhasználható.

Sok esetben nem lehet számszerűsíteni egy technológia vagy megoldás jóságát, így szubjektív tesztekhez kell folyamodni jobb híján. A szubjektív volta miatt azonban az értékek leginkább egy kísérleten belül hordoznak csak jelentést. Két eltérő tesztelés eredményeinek összehasonlítása alapján nem érdemes erős következtetéseket levonni.

2.9.1. Mean Opinion Score (MOS)

A MOS [12] egy főként telekommunikációs és multimédiás szakterületen használt metrika audió vagy vizuális kommunikációs megoldások minőségének értékeléséhez. Nagyon gyakran használják mesterséges intelligenciával szintetizált beszéd színvonalának mérésére például. A MOS érték szubjektív teszteléssel kapható meg úgy, hogy kísérletben részvevő emberek értékelik a nekik mutatott megoldásokat egy általában 1-től 5-ig terjedő skálán, ahol az 1 a legrosszabb és 5 a legjobb, majd a kapott eredményeket átlagoljuk.

2.9.2. Mean Square Error (MSE)

Számos helyen felüti a fejét az átlagos négyzetes hiba [16], de egyik legnépszerűbb felhasználási módja az AI modellek tanításánál van. Különösen gyakran használják regressziós analízis során regressziós modell jóságának mérésére. Ez jelentheti egész pontosan akár képek hasonlóságának meghatározását. Ilyenkor értéke a két kép pixelenek négyzetes különbségének átlagolásával kapható meg. Van általánosabb megfogalmazása is predikciókra vonatkozóan, de az a dolgozat szempontjából nem releváns. Minden esetre minél kisebb az érték, annál kisebb az eltérés, tehát például annál hasonlóbbak a képek.

Az MSE egy viszonylag egyszerű és könnyen számolható metrika, de megvannak a maga korlátjai. Nem veszi figyelembe az emberi látás tulajdonságait, ezért a kisebb MSE érték nem feltétlenül jelent jobb vizuális élményt egy ember számára. Kis különbségek a pixelek értékeiben aránytalanul megnövelhetik az eredményt, tehát érzékeny a zajra is. Mindezek miatt általában más technikákkal kombinálva szokás használni, mint az SSIM (ld. 2.9.3 fejezet) vagy csúcs jel-zaj arány (Peak Signal-to-Noise Ratio, PSNR). Ezáltal átfogóbb értékelést nyújt a képminőségről és hasonlóságról.

2.9.3. Structural Similarity Index Measure (SSIM)

Az SSIM [23] egy széleskörűen használt metrika két kép vagy képkocka közötti hasonlóság számszerűsítésére. Úgy tervezték, hogy a kép érzékelt minőségét mérje fel. Ehhez számításba vesz olyan dolgokat, mint a megvilágítás, kontraszt és struktúra. Ennek megfelelően három fő komponensből áll:

- **Fényerő összehasonlítás:** Kiértékeli a képek közötti fényerőbeli különbséget.
- **Kontraszt összehasonlítás:** Felméri a kontranszt és élességbeli különbségeket a variációk és lokális minták (pattern-ek) figyelembevételével.
- **Strukturális összehasonlítás:** Felméri a képek szerkezetében vagy textúrájában található hasonlóságokat a lokális minták összefüggéseinek figyelembevétele mellett.

Az eredménye egy -1 és 1 közötti szám, ahol az 1 azt jelenti, hogy a két kép teljes mértékben megegyezik. Így a magasabb SSIM érték nagyobb vizuális hasonlóságot jelent a referencia és torzított kép között.

3. fejezet

Tervezés

Ebben a fejezetben ismertetem magát az objektum központú remote renderinget, valamint a vizsgált technikákat, a vizsgálati környezet- és kiértékelések tervét.

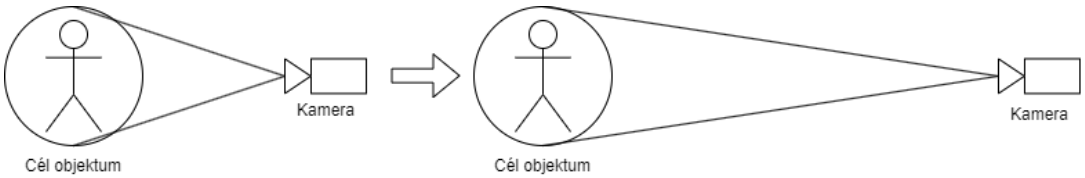
3.1. Objektum központú remote rendering

Az objektum központú remote rendering elnevezés arra utal, amikor a már jól ismert remote renderinggel ellentétben (ahol a felhasználó által látott kép teljes egésze a szerveren készül), csak 1 objektum képét továbbítja a szerver a kliensnek egy adatfolyamon. Több adatfolyam (stream) esetén megvalósítható több objektum is, de mindegyik stream csak 1 objektum képét hordozza. A kliensen a megkapott képeket mindig kitesszük a vászonra, még ha ez pazarlónak is tűnhet elsőre. A szerver akkor is kirendereli és elküldi a képet, ha a felhasználó nem az objektumra néz. Így jobban lekezelhető az az eset, amikor a felhasználó hirtelen néz rá az objektumra, hiszen már ott van a képe, pont mintha lokálisan létezne. Amennyiben a vászon is változtatja a helyzetét időről időre, vagy folyamatosan, úgy különösen igaz, hogy könnyebben kezelhető. A megoldásnak tehát az az előnye, hogy könnyen tudja kezelni a különböző mozgásokat a rendszerben.

A dolgozatban 3 különböző, objektum központú remote renderinget megvalósító algoritmust vizsgáltam meg melyek mindegyikére igaz, hogy van egy szerver és kliensből álló rendszer. A szerveren van egy objektum, amit meg szeretnénk jeleníteni e módon távolról a kliensen. A szerveren lévő objektum helyzete megegyezik egy, a kliensen elhelyezett vászon (canvas) objektuméval, ami meg fogja jeleníteni a szerveren lévő objektumról elküldött képet. A szerveren van egy kamera, aminek a képét továbbítani fogjuk a kliens felé és pozíciója szinkronban van tartva a kliensen lévő HMD pozíciójával. Forgási vagy rotációs értékei azonban e kettőnek nem egyeznek meg. A szerveroldali kamera mindig pontosan a cél objektum középpontja felé néz, míg felhasználó értelemszerűen arra, amerre szeretne.

Egy egész intuitív megoldás volna szimplán lekövetni a HMD mozgását a szerveroldali kamerával. Ilyenkor azonban duplán vesszük az objektumtól való távolságot, mivel a vásznon megjelenő objektum egyre kisebbé válik a szerveroldali kamera távolodásának

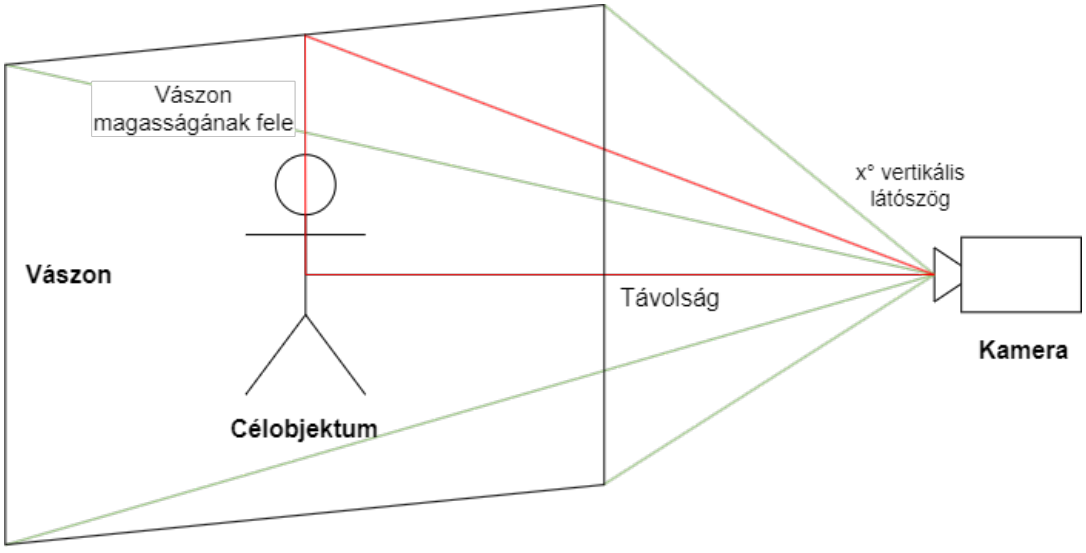
hatására. Továbbá a felhasználó önmaga is távolodik a vászontól, ami emiatt megint csak egyre kisebb lesz perspektivikusan. Ebből az következik, hogy az egyiket ki kell venni az egyenletből. Logikusan pedig nem a felhasználót érdemes korlátozni. Tehát azt kell megoldani, hogy a vásznon látható objektum a szerveroldali kamera távolságától függetlenül a valós méretének megfelelően jelenjen meg. Ehhez kínálnak a javasolt technikák három különböző megközelítési módot kisebb-nagyobb sikerrel, mint azt később láthatjuk is.



3.1. ábra. A látószög változtatás alapú warp metódus működése.

3.1.1. Látószög változtatás alapú warp

A technika elmagyarázása képek segítségével a legegyszerűbb, így tekintsük is meg a 3.1 ábrát. Két állapotot figyelhetünk meg. Bal oldalon a cél objektumhoz közel van a kamera, míg a jobb oldalon valamivel távolabb. A két oldal között csak a kamera pozíciója, és annak látószöge változott. A látószög változása ellensúlyozza a távolodás általi kicsinyülését a célobjektumnak a kamera képén. Felfogható úgy, hogy minél távolabb van a kamera, annál inkább közelít rá a célobjektumra, és fordítva.



3.2. ábra. A látószög kiszámítása a látószög változtatás alapú warp metódusnál.

A 3.2 ábra a technikához szükséges matematikai számítás megértését segíti. Látható rajta jobb oldalt a szerveroldali kamera, bal oldalt a vászon, és annak a közepén a célobjektum. Jelölve van a kamera és célobjektum közepe közötti távolság, a vászon magasságának

fele és a kamera vertikális látószöge mint ismeretlen. A látószög változtatás esetén a távolság (ahogy mindig) kiszámítható, a vászon vagy canvas magassága pedig adott. Ezekből egy kis trigonometria segítségével pedig kiszámítható a keresett látószög:

$$x = 2 \arctan\left(\frac{\text{magasság}}{\text{távolság} * 2}\right) \frac{180}{\pi} \tag{3.1}$$

Ha a számításnak megfelelően beállítjuk a szerveroldali kamera látószögét minden képkockagenerálás alkalmával, akkor elérjük a kívánt hatást. A célobjektum mérete a kliens oldali vásznon látszólag függetlenné vált a szerveroldali kamera távolságától.

Értelemszerű, de érdemesnek tartom kiemelni, hogy a vásznon megjelenő objektum képélessége attól függ, hogy mekkorára van választva a vászon mérete. Ezért érdemes azt úgy választani, hogy képaránya megegyezzen a felvétel képarányával, és méreteit tekintve magába foglalja a célobjektumot, de ne legyen aránytalanul nagy. A canvas magassága és szélessége dinamikusan változtatható az arány megtartása mellett. Ezáltal extra funkcióként lehetőség nyílik a kép élességének testszőleges konfigurálására. A vizsgálatok során a lehető legélesebb képpel dolgoztam (egy kis ráhagyást leszámítva, hogy bőven beleférjen az objektum), ami 2.278125 x 1.2814453125 métert jelent a vászon x és y tengelyén egy 1 méter magas szekrény esetén. Ha tökéletesen szeretnénk szimulálni egy lokális objektumot, akkor célszerű kidolgozni a homályosságra vonatkozó számításokat is.

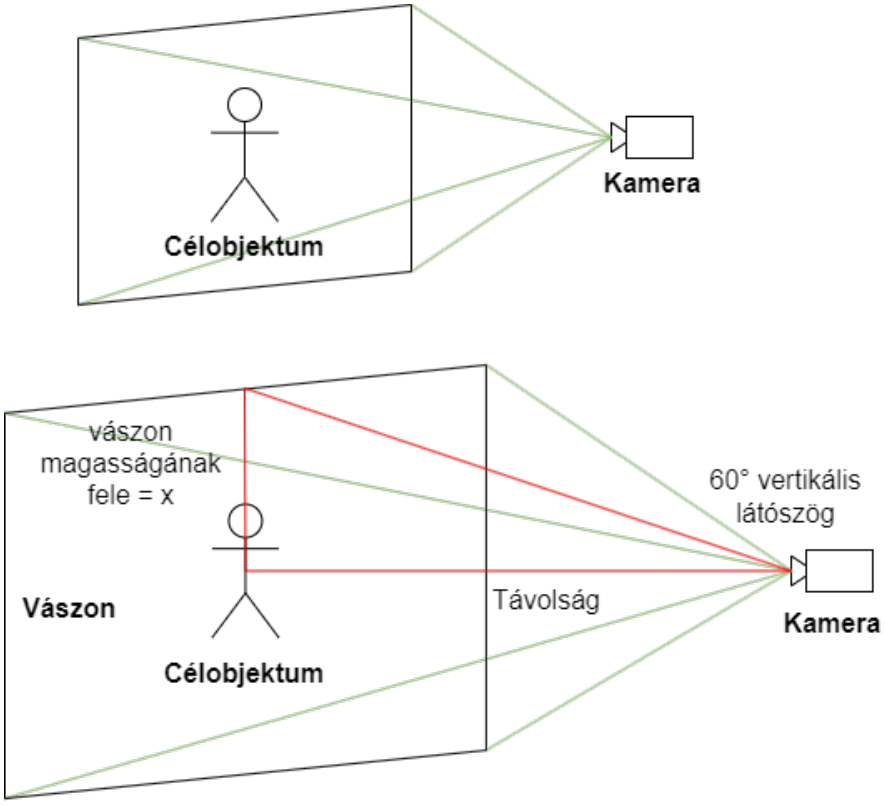
3.3. ábra. A látószög változtatás alapú metódus képélessége a vászonmérettől függően.

A 3.3 ábrán láthatjuk hogyan néz ki ez a homályosság és élesség a szimulációs környezetben. A szekrény 1 méter magas és a $(0, -0.5, 0)$ vektoron helyezkedik el. Mindkét

oldalon lévő kép a $(0, 0, -2)$ pozícióból készült a felhasználó szemszögéből. Alul, a vászonra továbbítandó, szerveroldali kamera képe látható. Bal oldalon a vászon 12.8 x 7.2 méteres, a jobb oldalon pedig a már korábban is említett 2.278125 x 1.2814453125 méretű. Ekkora méretkülönbség mellett tisztán látható az élességbeli eltérés.

3.1.2. Vászonméretezés alapú warp

A vászonméretezés alapú warp technikánál a látószög változatlan, a távolság kiszámítható és a vászon mérete az ismeretlen az egyenletben, amit folyamatosan változtatunk. Ez figyelhető meg a 3.4 ábrán. A képen látható felülső állapotot tekintsük az alap állapotnak. Alatta pedig a vászon méretének megnövekedése látható a kamera távolítása végett. Vizuális típusoknak talán úgy a legkönnyebben érthető a megoldás lényege, ha úgy fogják fel, hogy a kamerából kiinduló 4, látószöget határoló félegyenes kimetsz egy téglalapot a kamerára merőleges síkból, aminek középpontja a célobjektum.



3.4. ábra. A vászonméretezés alapú metódus működése és kiszámítása.

A szükséges matematikai számítás magyarázatában megintcsak a 3.4 ábra segít. Számítások tekintetében nagyon hasonlít a látószög változtatós technikára, hiszen trigonometrián alapszik. Ha minden képkockgenerálást követően beállítjuk a vászon méretét a kiszámolt értékre, akkor a duplán érvényes távolság problémája ezáltal megoldódik.

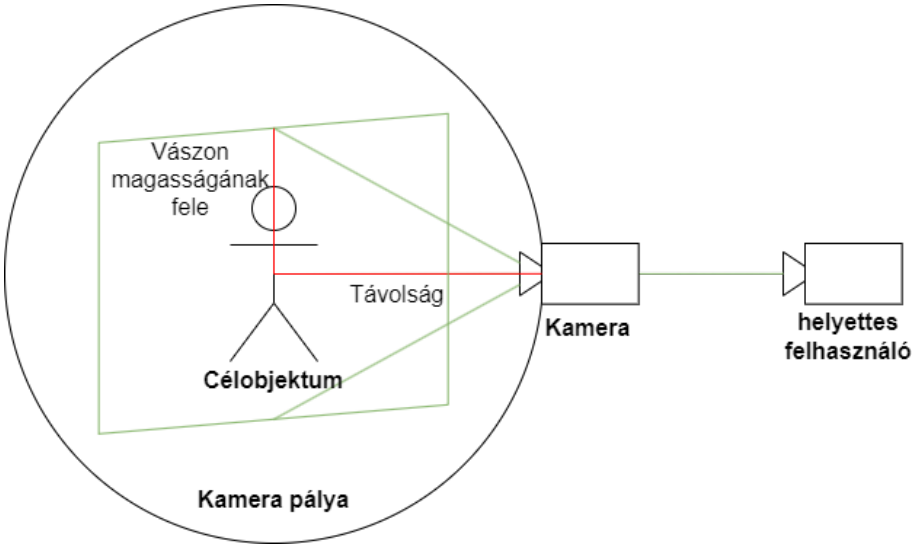
Kicsivel homályosabb a kép az elmentés és hálózaton való átvitel végett, de cserébe közel marad a valóshoz. Tehát mindig egy kicsivel homályosabb annál, mintha lokálisan

ott lenne az objektum, de úgy homályosodik a távolság növekedésével, ahogy azt a lokális objektumon érzékelnénk.

$$\text{magasság} = 2x = 2 * \text{távolság} * \tan\left(\frac{\text{kamera látószög}}{2} * \frac{\pi}{180}\right) \tag{3.2}$$

3.1.3. Konstans távolság alapú warp

A konstans távolság alapú warp technika kissé eltér az előző kettőtől, mivel nem trigonometria felhasználásával éri el, hogy az objektum állandó méretűnek látszódjon a vásznon. A 3.5 ábrán látható a működése, miszerint először is van egy helyettes felhasználó objektum. Ez minden pillanatban szinkronban van a felhasználó helyzetével. A kamera helyzetét a proxy objektum és a célobjektum közepei között húzódó szakaszból állapíthatjuk meg az alapján, hogy hol metszi az ábrán Kamera pálya elnevezésű kört (ami a szimulációs környezetben egy gömb). Miután megvan a pozíciója a 3D térben, még a célobjektum közepe felé kell forgatni. Talán már érezhető is, de kényelmetlenné válik a helyzet abban az esetben, ha a felhasználó túl közel, tehát a gömbön belülre menne. Sebtapaszként a cél, -és proxy objektumok közötti szakaszt meg lehet hosszabbítani a célobjektum középpontjától növekvő távolság irányába, és ott elkapni a pályán való áthaladást. Ennek köszönhetően valamennyit javítottunk a helyzeten, de így sem tökéletes.



3.5. ábra. A konstans távolságú metódus működése.

Három értékkel is lehet operálni, hogy megkapjuk a leginkább valósághű beállítást. Egyik a célobjektum körüli gömb mérete, második a kamera látószöge és utolsó a vászon mérete. A második és harmadik a jó eredmények érdekében igazából fixálhatóak a látószög alapú technikához hasonlóan. Tehát a gömb sugarából és a vászon méretéből kiszámítható az ideális látószög (lásd 3.2 fejezet), a vászon mérete pedig emiatt meghatározza a kép élességét. Marad tehát a gömb mérete, aminek változtatása a perspektívára van hatása. Jól látszik a jelenség a 3.6 ábrán. A bal szélső kép 1.07415, a középső 1.953, a jobb szélső

pedig 6.5 méteres sugárrel készült. Könnyen észlelhető a különbség a három kép között szabad szemmel is.

3.6. ábra. A gömb pálya méretváltozásának hatása a konstans távolságú metódusnál.

3.2. Szimulációs rendszer

A szimulációs környezet megvalósításához a Unity-t választottam, ugyanis van tapasztalatom benne és potenciálisan ezen a platformon lesznek a későbbiekben felhasználva a tárgyalt technikák.

Az egyszerűség érdekében a szerver és kliens oldalt is egy scene[1]-ben valósítottam meg. Könnyen lehetségessé válik így a szimulált hálózati késleltetés és a képküldés frekvenciájának konfigurálása. Továbbá nem mellékesen elég egyetlen alkalmazást futtatni és a hálózati konfigurációval sem kell bajlódni.

A környezet adottságaiból adódóan 7+1 kamerát használok fel, minden technikának 2, a referenciának 1. A ráadás kamera pedig azért szükséges, hogy vizsgálat során felhasznált videók kimentése zavartalan legyen. Ezt úgy kell érteni, hogy kamerák képének kimentéséhez futnia kell az alkalmazásnak, ilyenkor pedig az a kamera egy kicsit másképp működik, amelyiknek a képét a felhasználó éppen látja. A kiemelt szerepet ezért egy olyan kamerának tulajdonítom, amelynek nem számít a képe.

A kliens szerepébe bújó 4 kamerát elkerítem egy-egy 100x100 méteres térben. A maradék 3, szervert reprezentáló kamera pedig jól megfér egymással azonos térben is, így ezek együtt kapnak egy 100x100-as kockát. Az utolsó, 8. kamera pedig csak annyit kell, hogy teljesítsen, hogy messze van a többiektől és nem is néz arrafelé.

[1]Unity-ben a gyökere egy objektum hierarchiának. Tartalmazhat karaktereket, környezetet, világítást és sok egyebet, ami szükséges lehet egy alkalmazás fejlesztése során.

3.3. Új warp technikák mérése

A javasolt warp technikákat két megközelítéssel vizsgálom. Az első egy objektív metrika, ennél pixelről pixelre hasonlítom össze az eredményt egy referencia képpel, ami a kívánt minőséget reprezentálja. Második egy szubjektív módszer, az átlagos véleménypontszám (Mean Opinion Score, MOS). Ezt is szükségesnek találom, ugyanis a pixelek egyezésének mértéke nem feltételnül határozza meg a felhasználói élmény minőségét. A szubjektív teszt során olyan kérdések elé állítom az alkalmazás kipróbálóit, mint:

- Mennyire magabiztosan tudod megkülönböztetni, hogy éppen a valós objektumot vagy a remote renderelt objektumot látod?
- Hogyan értékelnéd összességében a vizuális minőség tekintetében az első megoldást?
- Mennyire vagy elégedett az első megoldással?
- Vettél észre bármiféle visual artifactot vagy torzítást?
- Melyik megoldás volt a legpontosabb a valós objektum reprezentálásában?

4. fejezet

Implementáció

Ebben a fejezetben bemutatom a szimulációs környezet, a technikák és a vizsgálat implementálásának részleteit.

4.1. Környezet

Ahogy már említettem, a szimulációt Unityben készítettem, egyetlen scene alá csoportosítva. Mivel a három módszer megvalósításának módja és hierarchiája közel azonos játékobjektumok tekintetében, alapvetően a látószög változtatás alapú warpot fejtem ki részletesen. Említést azonban teszek az eltérésekről a másik kettőhöz képest.

A látószög változtatás alapú megoldás a szerver oldalon a kamerából, a kliens oldalon pedig kamarából és a képet megjelenítő vászonból áll. Tehát szerver oldalon van egy kamera, a kliens oldalon pedig egy hierarchia, aminek fő elemei egy kamera és egy vászon a tér közepén. A szerver és kliens rész egymástól falakkal van elválasztva, így bizonyos keretek között egymástól izoláltan léteznek és működnek. Az izolált terek vagy kockák 6 darab 100x100 méteres síkból állnak, amelyek egy kocka formájába vannak rendezve. Mind a szerver, mind a kliens oldali kamera és környezet ezeknek a közepén van, így biztosítva, hogy amíg a felhasználót reprezentáló kamera nem megy át a falakon, addig a 3 technika egymástól teljesen zavartalanul működhet. Zavarás alatt is igazából csak annyi értendő, hogy a vizsgálat során felhasznált felvételbe belelátszódna két vászon is, és egyebek. Magukra a megoldásokra nem volna hatással, csak a vizsgálatot rontaná el, hiszen a pontos mérésekhez azonos környezet, azonos útvonal és azonos időközök kellenek a képek között.

A továbbiakban a szerver oldali kamerára leginkább capture (rögzítő) kameraként fogok hivatkozni. A kliens vagy felhasználó kamerára pedig, mint roaming (vándorló) kamera. A referencia kamerával együtt tehát 4 roaming kamera van. Mind a 4-hez hozzácsatoltam egy *CameraMovement.cs* nevű szkriptet, amely felhasználói input hatására editor[1]-hoz (szerkesztő) hasonló mozgást tesz lehetővé. Tehát a balra, jobbra, előre, hát-

[1]A Unity vizuális szerkesztő felülete. A legtöbb interakció a fejlesztés során ezen keresztül történik. Futásidőben is szabadon mozgatható kamerával rendelkezik, amely hasonlóan működik bármely modellező alkalmazáséhoz.

ra, fel és le mozgást a billentyűzet segítségével lehet megoldani, míg a fordulást az egér segítségével. A shift billentyű nyomva tartása megnöveli a mozgási sebességet.

4.2. Kép továbbítás

A capture kamera képének továbbítását a roaming kamerával egy szobában lévő vászonra a *CaptureCameraManager.cs* szkript végzi. Ennél adható meg továbbá az FPS és a hálózati késleltetés mértéke is.

Mielőtt belekezdenék a kód relatíve pontos működésébe, ismertetnem kell röviden néhányat a Unity saját, beépített osztályaiból származó függvényekből.

- **Update**: Képkockánként meghívódik, a lefutások száma a futtatókörnyezet teljesítményén múlik.
- **FixedUpdate**: Másodpercenként 50-szer fut le, ha kell frame kihagyással is.
- **Start**: Az update legelső lefutása előtt egyszer fut le.

Nem szeretnék túlzottan elkanyarodni fejlesztői dokumentáció irányába, de azért a lényeget kiemelve fontosnak tartom a *CaptureCameraManager.cs*-ben megvalósított képtovábbítást bemutatni. A Start függvényben konfigurálom vászon anyagának beállításait. Az Update-ben először is beállítom az összes capture kamera lokális pozícióját az aktív (jelenleg vezérelt) roaming kamera lokális pozíciójára. Utána a capture kamerák orientációját a célobjektum felé transzformálom. Ez a kettő művelet elengedhetetlen feltétele mindhárom technika működésének. Ezt követően a roaming kamerák lokális pozícióját és orientációját szinkronizálom a jelenleg aktív kamerához. A vizsgálat során lesz ez igazán fontos, amikor is egy elmentett mozgást kell visszajátszania az összes roaming kamerának. A szinkronizáció nem mellékesen a felhasználói élményt is növeli a tesztelés közben, ha kamerát kell váltani.

Az Update függvényben valósítottam meg az alkalmazás számos fontos elemét. Ennek okán ebben a szakaszban már csak az Update különböző lefutási eseteiről írok. A képek mentésének és továbbításának folyamata abban az esetben kezdődik el, ha a legutóbbi Update óta elegendő idő eltelt. Az eltelt idő ellenőrzése által szabályozható, hogy hány kép készüljön egy másodperc alatt, tehát a felvételi FPS. Ha pozitív eredményt ad ez az ellenőrzés, akkor tehát elkezdődik a kép mentésének és továbbításának folyamata a technikák függvényeinek meghívásával. Természetesen ezek eltérnek kisebb-nagyobb dolgokban. A látószög alapú warp-nál a kamera látószögét, a vászonméretezés alapú warp-nál a hozzátartozó vászon méretét, a konstans távolság alapú warp-nál pedig a kamera látószögét és pozícióját állítom be. Ezekre részletesebben kitértem tulajdonságaikat is tekintve a sorrendben 3.1.1, 3.1.2 és 3.1.3-as pontokban. A folyamat a kamerák képének kézi (tehát explicit, kódból hívott) kirenderelésével folytatódik. Az elkészült képekről készül egy

értékmásolat RenderTexture[2] formájában, amit eltárolok egy FIFO[3]-ban a pontos idő, lokális pozíció és egy vászon méret mellett. Jobban átlátható formában tehát a következőket tárolom el:

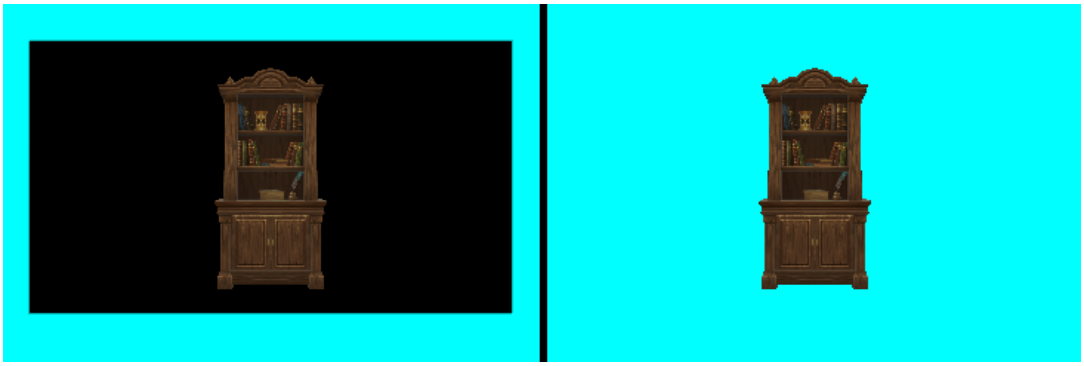
- **Pontos idő:** A hálózati késleltetés szimulációjához.
- **Aktív kamera lokális pozíciója:** A kliensen a vászon arra kell hogy nézzen, amerre a felvétel pillanatában a capture kamera pozíciója volt. Ha nem így volna, akkor végeznénk egy szükségtelen torzítást a képen, hiszen a felhasználó nem pontosan azt látná amit a szerveren lévő kamera.
- **Vászonméret:** A vászonméretezés alapú warp-hoz szükséges, hiszen a szerver határozza meg, hogy a kliens mekkorára állítsa a vásznat.
- **Látószög változtatás alapú warp rendertexture-je**
- **Vászonméret változtatás alapú warp rendertexture-je**
- **Konstans távolság alapú warp rendertexture-je**

A 3 rendertexture (a saját technikájukhoz tartozó) vásznon való megjelenítés céljából kerül elmentésre.

Felhasználásuk az update egy másik lefutási ágában történik, ami akár az előző bekezdésben bemutatott ággal egy cikluson belül is futhat. Mondjuk tehát, hogy elérkeztünk az Update azon ágához, ami felhasználja a FIFO-ba mentett adatokat. Először is lefut egy ellenőrzés, hogy van-e idősebb adat a FIFO tárolóban, mint a késleltetésben beállított idő. Ha a legidősebb adat is fiatalabb ennél, akkor itt véget ér az ág futása. Ha azonban van elég régen elmentett adat, akkor a kliens számára megjelenő vászont beforgatom a FIFO-ból megkapott lokális pozíció irányába. Magának a beforgatásnak az az oka, hogy így lett megtervezve és felépítve a remote rendering rendszer. A háromdimenziós hatást kettődimenziós képekkel ily módon, tehát sűrűn frissített 2D képekkel értem el. Ehhez pedig követnie kell a felhasználót a vászonnak, hiszen ha például nem fordulna és 90 fokos szögből nézné a rajta lévő képet a felhasználó (a teljesen szembelevő orientáltsághoz viszonyítva), akkor semmit sem látna belőle. Egy másik aspektusa a dolognak, hogy miért a múltbéli pozícióhoz forgatom be. Erre az a válasz, hogy a máskülönben rosszul torzulna a kép a felhasználó szemszögéből. Ezután felülírom az elmentett RenderTexture-ökkel a kliens kamerák anyagához tartozó RenderTexture-öket. Tehát megjelenítem a korábban elmentett képet a felhasználónak. Ha ez is megtörtént, következőnek beállítom a vászonméret változtatás alapú warp-hoz a hozzátartozó vászon méretét a FIFO-ból származó mérettel. A másik 2 technika nem igényel ezen a ponton semmit. Az utolsó lépés pedig a már nem használt RenderTexture-ök felszabadítása, így elkerülve a memóriaszivárgást.

[2]Futásidőben készíthető és változtatható textúra
[3]First In First Out memória

Mindezek helyes beállítása után működik a szimulációs környezetben a capture kamera képének továbbítása a megfelelő roaming kamerához, amit lehet mozgatni és váltani közöttük. A 4.1 ábra bal oldalán látható módon jelenik meg a továbbított kép. Ez nyilvánvalóan nem megfelelő. Ahhoz, hogy az ábra jobb oldalán megtekinthető módon látszódjon a célobjektum, két lehetőséget vizsgáltam meg. Az első, hogy a képen a célobjektumot és környezetét is elküldöm. Ennek hátránya, hogy a késleltetés nagyon szembetűnő és a környezetre vonatkozóan is mindennek meg kell egyeznie azzal, amit a felhasználó lát. Kiterjesztett valóság alkalmazásra nézve ez abszolút nem reális. Ezért a második lehetőséget valósítottam meg, ami egy Compute Shader írását jelentette. Sajnálatos, de várható módon ennek is van hátránya, mégpedig, hogy pontosan 1 színt fel kell áldozni a transzparencia oltárán.



4.1. ábra. A célobjektum és környezet a vásznon.

Unityben minden játékobjektum tartozik valamilyen rétegbe (layer) és minden kameránál megadható, hogy milyen rétegeket lásson. Így egyszerűen kezelhető, ha bizonyos kamerákkal csak bizonyos objektumokat szeretnénk rögzíteni, a hátteret pedig egy kitöltenénk egy beállított színnel. Tehát ahogy az jelen esetben is történik a célobjektummal.

4.2.1. Compute Shader

A Compute Shader-ek olyan programok a Unity-ben, amelyek a grafikus kártyán futnak a standard renderelési folyamaton kívül. Több dologra is kimondottan hasznosak, de én azon tulajdonsága miatt használtam, hogy képes meggyorsítani a program futásának egyes részeit. Minden kép pixelein kondícionálisan változtatni igen erőforrásigényes lehet, főleg hogy erre technikánként szükség van. A *Transparency.compute* shader-em leellenőriz minden egyes pixelt, hogy az az RGB (0,0,0) színkóddal rendelkezik-e. Ha igen, akkor az alfáját[4] 0-ra állítja. Ha nem, akkor változatlanul hagyja a pixelt. Inicializálása a Start függvényben történik (még mindig a *CaptureCameraManager.cs*-ben). Lefutását pedig minden képrenderelés után, de még az érték alapú mentés előtt kell meghívnom az Update ciklusában.

[4]RGBA színkódolásban az A, alfa (alpha) meghatározza az áttetszőség mértékét. 1 az átlátszatlan, 0 a teljesen átlátszó.

Az editorban beállítható, hogy a capture kamera milyen színnel helyettesítse azt ahol nem lát semmit. Például a 4.1 ábra bal oldalán látható fekete színű téglalap esetén a capture kamerának a színtiszta fekete volt megadva. Ebből következik, hogy bármilyen szín felhasználható erre a célra.

4.3. Vizsgálat

A dolgozatom egyik legfontosabb része a technikák több szempontból való vizsgálata. Ehhez elengedhetetlen, hogy pontosan megegyező paraméterek mellett teszteljem a különböző módszereket.

A felhasználói inputból származó reprodukálhatatlanságot a mozgás és orientáció elmentésével, majd visszajátsával oldottam meg a *PlaybackRecorder.cs* nevű szkripttel. Feladatköre a képek kimentésére és azokból videó készítésére is kiterjed, bár egy másik osztály segítségével. A „Start Recording" gomb megnyomása után a rendszer elmenti a jelenleg aktív kamera helyzetét és forgási adatait minden FixedUpdate futásban. Egész addig teszi ezt, amíg a felhasználó le nem állítja, vagy az alkalmazás ki nem fogy a területből. Ahhoz, hogy ez megtörténjen rettentő hosszú felvételt kellene készíteni egy átlagos gépen, így nem tettem biztonsági lépéseket. Azért választottam FixedUpdate függvényt mint az implementáció helye, mert így érhető el a legmagasabb szintű determinisztikusság Unity környezetben. A sima Update-be helyezve változna a „mintavétel" gyakorisága szinte minden pillanatban, ami belátható, hogy nem előnyös. A visszajátszással ugyanez a helyzet, az is a FixedUpdate-ben kapott helyet. A kívánt mozgás felvétele után ismételten megnyomva, a mostmár „Stop Recording" feliratúra változott gombot, állítható le a rögzítés. A Unity projekt gyökérmappájába tárolom el az adatokat .dat kiterjesztésű fájlként, ami így újrafelhasználható és visszajátszható.

A visszajátszás a „Load from file" gomb megnyomásával indítható. Ekkor minden FixedUpdate lefutáskor betölti az alkalmazás a legkorábbi, még fel nem használt adatsort. Ilyenkor attól függően, hogy milyen beállításokkal lett az alkalmazás buildelve (amikor Unity készít egy standalone futtatható állományt), kirendereli a 4 roaming kamera képét és elmenti azokat. Az elmentés lépésenként történik, ami egy Unity-ben beállítható minimális időegység. Egy lépés FixedUpdate-től FixedUpdate-ig tart. Szervergéppel előfordulhat, hogy ez nem volna szükséges, de a felszereltségemmel igen, ezért csak ezt a módszert implementáltam. Frame-enként összesen 10 darab 1920x1080-as kép feldolgozása történik ekkor, amelyekből négyet fájlba is kimentek. A 10 képfeldolgozás a 3 capture kamerával, az ezeken futatottott compute shaderrel és a referenciával együtt 4 roaming kamerával jön ki. Talán nem meglepő a kijelentés, hogy ez igen erőforrásigényes.

Az egy-egy kamerához tartozó frame képként való mentését a *VideoRecorder.cs* nevű szkript végzi. Ebből minden olyan kamerához példányosítok egyet, amelynek fájlba szeretném írni a kirenderelt képét. Továbbá ez az osztály végzi ténylegesen a képek vesz-

teségmentes videóvá konvertálását is. Ehhez ffmpeg (lásd 2.7) utasítást tartalmazó folyamatot indít el a `C#` beépített, *ProcessStartInfo* osztályának felhasználásával. A videóvá konvertálás a program futása közben tehető meg az editorban, a *PlaybackRecorder.cs* mezői között található „Make Video" gombbal. A folyamat 5-20 másodpercet vesz igénybe a képek mennyiségétől függően.

4.3.1. Pixel összehasonlítás

Most már minden rendelkezésre áll, ami a pixelenkénti összehasonlításhoz szükséges lehet. A kapott 4 videó összehasonlításáz egy általam készített python szkript végzi. Felhasználtam hozzá a cv2 (ld. 2.8) könyvtár néhány függvényét. Kiszámítom benne egyesével az összes framepárnak az abszolút távolságát, és azokat egycsatornás szürkeárnyaltúra konvertálom. Így gyorsabban elvégezhetőek a további számítások. A 4.2 ábra bal szélén látható az eredeti frame, jobbra mellette a másik frame-től való abszolút távolsága. Továbbhaladva jobbra található a framepár abszolút távolsága szürkévé konvertálva, az ábra jobb szélén pedig a maszk figyelhető meg. A különbség szürkeárnyalatúra konvertálása által kapott halmazban található nem nulla értékek, tehát eltérő pixelek számát hozzáadom egy 0-ra inicializált változóhoz, aminek neve legyen *differenceCount*. Készítek egy-egy maszkot a két, videókból kiolvasott frame-ből. A maszk egy számtömb, a frame-mel megegyező paraméterekkel. Értékeit jelölje x_i, az eredeti frame értékeit pedig y_i. Ekkor a következő összefüggés határozza meg az értékét:

$$x_i = \begin{cases} 1 & \text{ha } y_i \text{ türkiz kék} \\ 0 & \text{Különben} \end{cases} \tag{4.1}$$

Miután megvan mind a két frame maszkja, azokat egy logikai „és" művelettel összeolvasztom, és megszámolom az eredményül kapott halmazban a nem nulla értékek számát. Így megkaptam azon pixelek számát, amelyek mindkét képen türkiz színűek, tehát a háttér részei. Elmentem ezt is későbbi felhasználásra a *matchingTurquoisePixels* változóba.

Ha a ciklus végetért, tehát az összes framepár fel lett dolgozva, megkapom a százalékos eltérést a következő egyenlettel:

$$\text{percentDifference} = \frac{\text{differenceCount}}{\text{totalPixels} - \text{matchingTurquoisePixels}} * 100 \tag{4.2}$$

Ezzel a módszerrel mindhárom technika pixelpontosságát kiszámítom a referencia videóval összevetve. A 4.2 ábra bal szélén látható az eredeti frame, jobbra mellette a másik frame-től való abszolút távolsága, még jobbra mellette framepár abszolút távolsága szürkévé konvertálva, a jobb szélén pedig a maszk található.

4.2. ábra. A pixel összehasonlítást végző python szkript képfeldolgozásának 4 fázisa.

4.3.2. MSE és SSIM

Az átlagos négyzetes hiba és a szerkezeti hasonlósági index kiszámítását egy python szkriptben implementáltam hasonló kezelhetőségük végett. Több könyvtárat is igénybe vettem, mint a math, NumPy, Skimage.metrics (ebben implementálva van az ssim kiszámítása) és a korábban is használt cv2 (ld. 2.8). A kódban először is kigyűjtöm mindkét videó összes képkockáját, majd készítek minden frame-hez egy maszkot. A 4.3 ábra bal oldalán látható az eredeti frame, mellette jobbra pedig a hozzá készült maszk. Ha a maszk értékeit x_i jelöli, az eredeti frame értékeit pedig y_i, akkor a 4.3 egyenlet szerinti összefüggés határozza meg a maszk tartalmát. Ahhoz, hogy ezt megkapjam, futtatok az eredeti frame-en egy szűrést a türkiz színre, majd a kapott tömb értékeit (ami nullákból és egyekből áll) invertálom a cv2 beépített függvényével. Ezután átkonvertálom a videókból kimentett képkockákat egycsatornás szürkeárnyalatúra (4.3 jobbról második kép), ismét a további számítások meggyorsításáért. Ezen a ponton inicializálom a két módszerhez tartozó változókat, amelyekbe képkockánként elmentem a megkapott eredményt és a végén majd átlagolom.

$$x_i = \begin{cases} 0 & \text{ha } y_i \text{ türkiz kék} \\ 1 & \text{Különben} \end{cases} \tag{4.3}$$

Egy cikluson belül alkalmazom a megkapott maszkot a cv2 „és” műveletével (4.3 ábra jobb szélső képe) a szürke frame, leellenőrzöm, hogy a maszkolás után maradt-e egyáltalán vizsgálat szempontjából érdekes pixel. Ha a maszk minden értéket kinullázott a tömbben, mivel az egész kép türkiz volt, azon nincs értelme kiértékelést végezni, így továbblép az algoritmus a következő képkockára.

Amennyiben eddig minden rendben ment, lefuttatom a ciklusban a maszkolt képkocka párra a beimportált ssim algoritmust és elementem a kapott eredményt a korábban inicializált változóba. Ha esetleg *nan* értéket adna vissza a függvény, akkor azt eldobom.

Az MSE értékét is kiszámítom az SSIM után, még mindig a ciklusban. A függvényt saját kezűleg implementáltam. Paraméterként a két képkockát vesz át, és készít egy együttes maszkot, aminek az alkalmazása után csak olyan pixelpárok maradnak a képkockákban, ahol nem fordulhat elő, hogy mindkettő 0 értékű. Az így megkapott szűrt frame-eknek kiszámítja pixelenként a távolságát, majd azok négyzetéből átlagot számol és visszatér vele eredményként.

A ciklus befejeztével kiíratom az SSIM és MSE értékeket.

4.3. ábra. Az MSE és SSIM kiértékelést végző python szkript képfeldolgozásának 4 fázisa.

5. fejezet

Kiértékelés

Ebben a fejezetben bemutatom az eddigi munka során kapott eredményeket. A diagramokban a következő rövidítéseket alkalmazom a technikákra vonatkozóan.

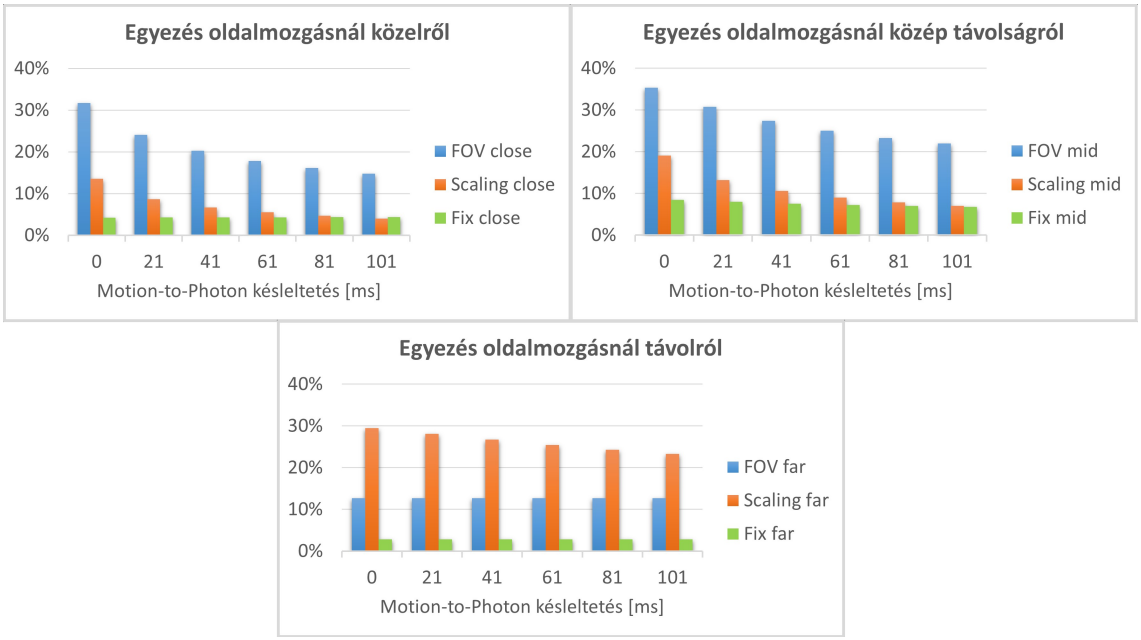
- **Látószög alapú:** FOV (Field Of View)
- **Vászonméretezés alapú:** Scaling
- **Konstant távolság alapú:** Fix

A megjelenő távolságok a közeli (close), közép távolságú (mid range) és távoli (far). Az alkalmazásban ezek 1.2, 2 és 6 méteres távolságokat jelentettek. Az 5.1 ábrán megtekinthető, hogy a kamera szempontjából hogyan látszódik a célobjektum balról jobbra a közeli, közép távoli és távoli beállításnál.

5.1. ábra. A kiértékeléshez használt távolságok

5.1. Pixel pontosság

A 4.3.1 fejezetben bemutatott módszerrel az 5.2 és 5.3 ábrákon látható eredményeket kaptam. A 5.2 ábrán úgy értelmezhetőek az eredmények, hogy közelről és közép távolságról

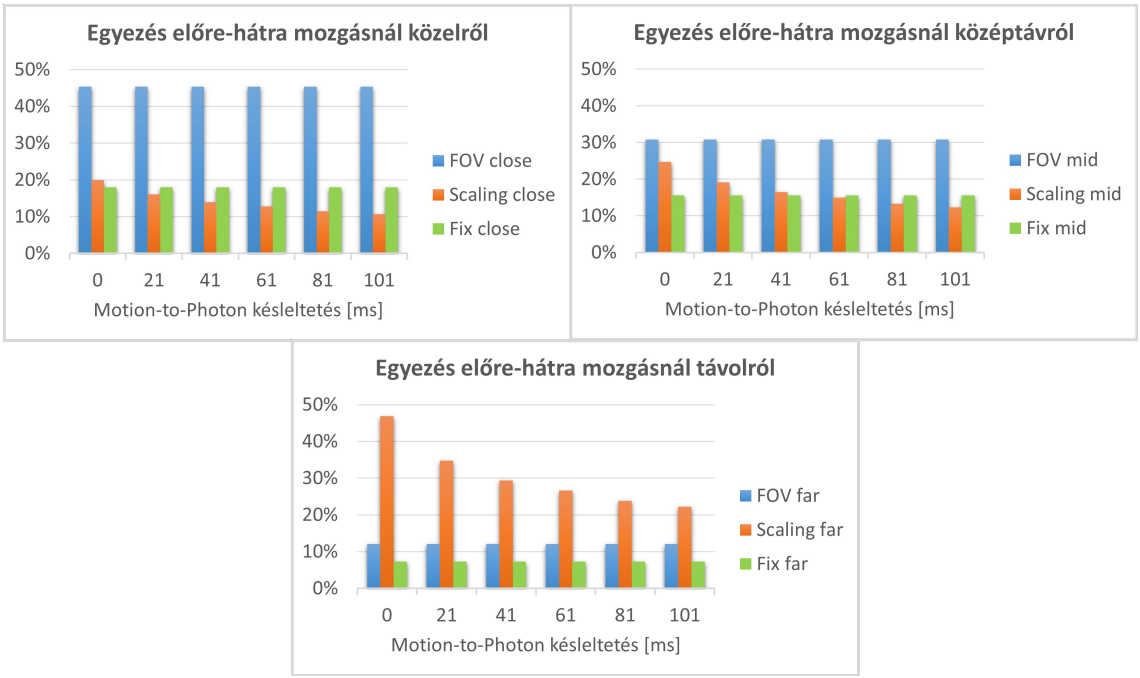


5.2. ábra. Pixel egyezés mérés eredmények oldalirányú mozgásnál.

egyértelműen a látószög alapú technika a leginkább pixelpontos. Várható mértékben csökken a pontossága a késleltetés növekedésével. A vászonméretezés és konstans távolságú technikák pedig a késleltetés növekedésével (számomra kissé meglepő módon) egy szintre kerülnek. Távolról azonban egészen más a helyzet. Könnyedén a Scaling metódus nyeri a pontossági versenyt. A távolság növekedésével pedig úgy látszik, hogy a késleltetés mértéke egyre kevésbé látványos, nincs akkora hatása a pontosságra. Úgy gondolom, hogy FOV metódus csak amiatt marad el ennyire, mivel az nem követi le jól a referencia természetes homályosodását, ahogy azt 3.1.1 fejezetben említettem.

Az 5.3 ábra azt árulja el, hogy ismét csak közelről és közép távolságról a FOV metódus domináns. Továbbá a Fix metódussal egyetemben nem érzékeny a késleltetésre. Ez furcsán hangozhat, azonban mélyebben átgondolva a látószög számítás alapú technikának a megoldásból jellegéből adódó tulajdonsága. Minden távolságból úgy végezzük a számításokat, hogy a vásznon változatlan méretűnek tűnjön a célobjektum. Ezért nem számít, hogy például egy 50 milliszekundummal ezelőtti állapothoz számítottuk ki, hogy 1.1 méter magas legyen az objektum vagy egy 2 másodperccel ezelőttihez. Az objektum mérete a vásznon nem változik. A tesztet teljesen szemből végeztem, perspektivikus változások miatt más szögből előfordulhatnak kisebb eltérések. Mivel a Fix pedig a FOV extra lépésekkel, így megörökölte ezen kellemes tulajdonságot.

Távolról azonban ismét a Scaling dominanciája figyelhető meg. Egész nagy hatása van rá a késleltetés, míg a FOV és Fix metódusokra ismét nincs. A látószög alapú technikánál ismét a homályossági érték kiszámításának hiányára gyanakszom mint a gyenge kiváltójára.



5.3. ábra. Pixel egyezés mérés eredmények előre-hátra mozgásnál.

A tesztelő könyezet nem tökéletesen determinisztukus, ugyanis ugyanabból az útvonalból készített két videó minimálisan eltérhet. A kapott eredményekben a tapasztalataim szerint ez kb plusz minusz 1%-ot jelent.

5.2. Átlagos négyzetes hiba

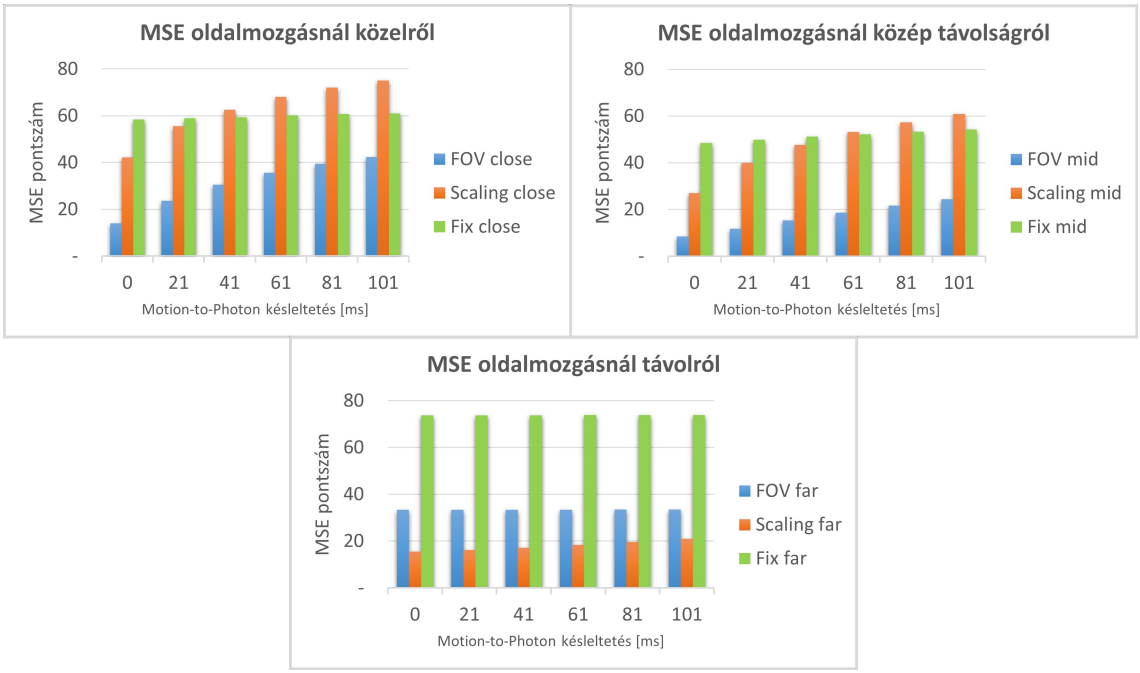
Az 5.4 és 5.5 ábrákon az ugyanazokon a videókon futtatott, de ezúttal MSE vizsgálat eredményei láthatóak. Emlékeztetőnek csak, hogy itt a minél kisebb eredmény a kívánatos.

Igazából ugyanazok olvashatóak le kis eltérésekkel, mint az előző szekcióban a pixel pontosságnál. Tehát mindkét teszt ugyanazokat a megfigyeléseket támasztja alá.

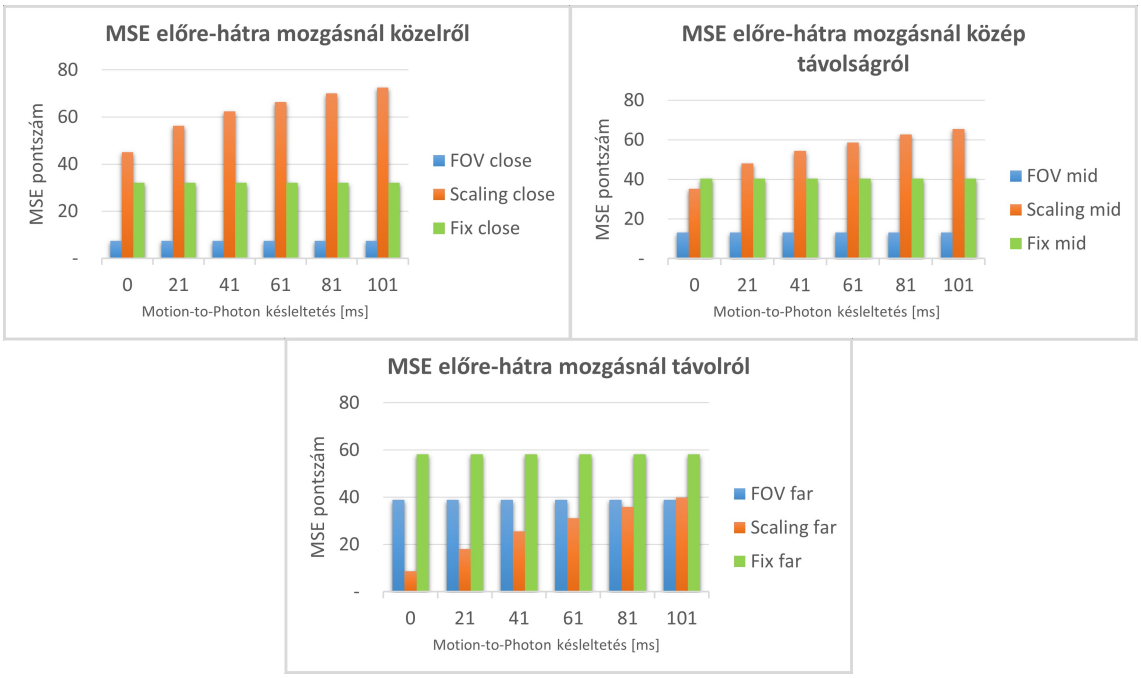
5.3. Strukturális hasonlósági index

Az emberi látás tulajdonságait is számításba vevő szerkezeti hasonlósági index eredményei láthatóak az 5.6 és 5.7 ábrákon. Nagyrészt ismét azonos eredmények olvashatóak le az ábrákról, de ezúttal ismét bemutatom őket. Oldalirányú mozgás esetén közelről és közép távon erőteljes vezetés látszik a látószög alapú technikától pontosság tekintetében. Távolról szoros verseny a vászonméretezés alapú warp javára, míg a konstans távolságos messze elmarad vetélytársaitól.

Előre-hátra mozgásnál a FOV és Fix érzéketlen a késleltetésre, de ez talán már várható is volt ezen a ponton. Mint ahogy az is, hogy közelről és közép távolságról a látószög alapú warp lesz a legpontosabb. Távolról pedig kis késleltetés mellett a Scaling, 61 ms és nagyobb késleltetéstől pedig megintcsak a FOV nyer. Az előre-hátra mozgásformánál

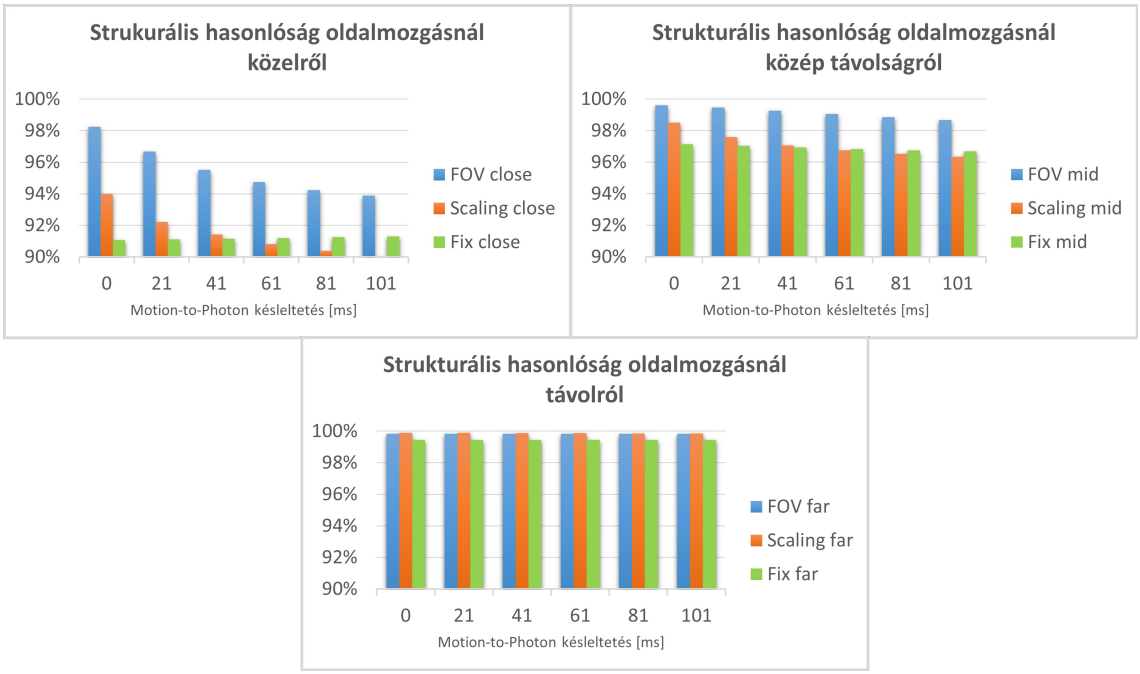


5.4. ábra. MSE mérés eredmények oldalirányú mozgásnál (az alacsonyabb pontszám a jobb).



5.5. ábra. MSE mérés eredmények előre-hátra mozgásnál (az alacsonyabb pontszám a jobb).

végig nagyon jó értékek figyelhetőek meg, de a mérések szerint távolról már gyakorlatilag megkülönbözhetetlen a vásznon lévő objektum a referenciától.



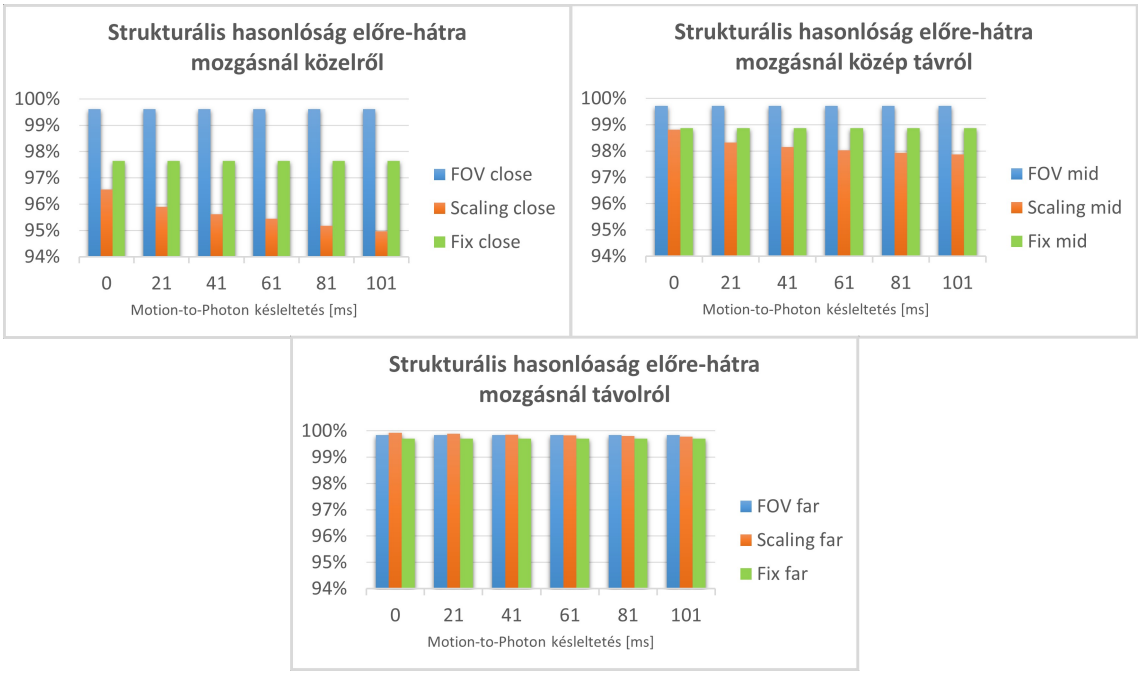
5.6. ábra. SSIM mérés eredmények oldalirányú mozgásnál.

5.4. Átlagos véleménypontszám

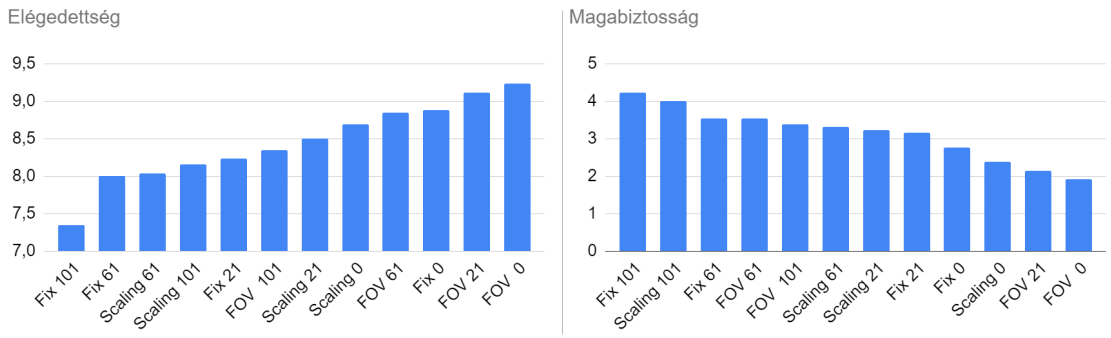
A kérdőívben mind a 3 technikához 4 késleltetésértékkel felvett videót mutattam meg a kitöltőknek. Azért, hogy ne legyen aránytalanul hosszú a kérdőív, a 41 és 81 milliszekundumos késleltetéseket kihagytam a többi méréstől eltérő módon. Így tehát a FOV, Scaling és Fix technikákhoz néztek meg a kitöltők 0, 21, 61 és 101 ms késleltetéssel felvett videókat.

A kérdőív kitöltésére 13 jelentkezőt sikerült találnom, ami nem sok, de tendencia már talán kiolvasható belőle. A mérés a legfontosabb eredményétt a 5.8 ábra bal oldalán láthatjuk. Már a kisszámú kitöltésből is körvonalazódni látszik az objektív mérések eredményével azonos minta. A látószög változtatás alapú warp érte el a legjobb eredményt 0 ms késleltetéssel, majd a második is a FOV lett a következő legkisebb, 21 ms késleltetéssel. A legrosszabb értékelést pedig a Konstans távolság alapú warp technika kapta. Kis fenntartással kell kezelni a kapott eredményeket, hiszen többen ott is láttak késleltetést, ahol nem volt, de ennek ellenére van információ tartalma az eredményeknek.

A 5.8 ábra jobb oldalán azt láthtajuk, hogy mennyire voltak biztosak benne az emberek, hogy remote renderelt objektumot láttak a videón, vagy a referenciában mutatott lokálisat. Jól tükrözi az elégedettségi eredményeket. Mindkét kérdésnél a 1-től 10-ig terjedő skálán kellett pontozni.



5.7. ábra. SSIM mérés eredmények előre-hátra mozgásnál.



5.8. ábra. MOS-sal mért elégedettségi és magabiztossági értékek.

5.5. Eredmény

A vizsgálat eredményeiből az látszik, néhány méteren belül egyértelműen a látószög változtatás alapú warp megoldás biztosítja a legpontosabb eredményt. Távolabbi (kb. 4 méternél messzebbi) objektumok esetén azonban a vászonméretezés alapú warp áll legközelebb a referenciához. Ezek igazak mind oldalirányú, mind előre-hátra mozgás esetén is, bár kissé különböző mértékben. Kiderült továbbá, hogy előre-hátra mozgásra indifferens a FOV és Fix metódus. A MOS vizsgálat megerősítette a korábbi eredményeket, és arra enged következtetni, hogy 61 milliszekundomos késleltetés még nem túl zavaró, 21 ms pedig nem is mindig észrevehető.

Ezen eredmények alapján az általam javasolt legjobb megoldásnak azt tartom, hogy monitorozzuk a felhasználó távolságát a szerveren, és a távolság függvényében váltsunk a látószög változtatás alapú és a vászonméretezés alapú technikák között, ezzel növelve a felhasználói élményt.

6. fejezet

Összegzés

A dolgozatban három új objektum központú remote rendering megoldást javasoltam és ezekhez egy vizsgálati módszert, amely objektív és szubjektív metrikák felhasználásával értékeli a technikák működését. Ezek megvalósításához és teszteléséhez készítettem egy szimulációs környezet Unity-ben, implementáltam a javasolt látószög változtatás-, vászonméretezés- és konstans távolság alapú warp megoldásokat. A tesztelés elvégzéséhez elmentett útvonalból és beállított, állandó FPS-el készítettem videókat a technikák használatáról, így biztosítva, hogy a videók kizárólag a rögzített technikai megoldásban térjenek el.

A vizsgálatokból az derült ki, hogy a látószög változtatás alapú warp technika a legígéretesebb, főleg közel és középtávon, amit jelen dolgozat keretein belül pár méteres távolságként definiáltam. A vászonméretezés alapú warp a legpontosabb távolról, a konstans távolság alapú warp pedig elmarad minden tekintetben a másik kettőtől. Ennek ellenére a szubjektív teszt során több ember is legpontosabbnak ítélte. A MOS szubjetív teszt eredményeire azonban csak fokozott óvatossággal szabad építeni, mivel kis számú, a változatosságot nem teljesítő embercsoport töltötte ki.

6.1. Továbbfejlesztési lehetőségek

Több továbbfejlesztési lehetőség vagy irány is nyitva áll.

- Warp technikák fejlesztése: Újszerű megoldások, javaslatok, ezért kicsi a valószínűsége, hogy már elérte volna a megoldás a tökéletes, nem javítható állapotát.
- Vizsgálatok finomítása
  - MOS fejlesztése: Több ember elérése a szubjektív teszttel, és javítani a hozzátartozó kérdőív tartalmán.
  - Többszöri vizsgálat futtatása az objektív metrikákkal, akár más környezetekben is implementálva

Irodalomjegyzék

[1] Volga Aksoy – Dean Beeler: Developer guide to asw 2.0. https://developer.oculus.com/blog/developer-guide-to-asw-20/, 2019. augusztus. [Online; accessed Oktober 15, 2023].

[2] Volga Aksoy – Dean Beeler: Introducing asw 2.0: Better accuracy, lower latency. https://www.meta.com/blog/quest/introducing-asw-2-point-0-better-accuracy-lower-latency/?utm_source=www.reddit.com&utm_medium=oculusredirect, 2019. április. [Online; accessed Oktober 15, 2023].

[3] Michael Antonov: Asynchronous timewarp examined. https://developer.oculus.com/blog/asynchronous-timewarp-examined/, 2015. március. [Online; accessed September 23, 2023].

[4] Russell M Barnes: *A positional timewarp accelerator for mobile virtual reality devices.* 2017, University of California, Santa Barbara.

[5] Dean Beeler – Anuj Gosalia: Asynchronous timewarp on oculus rift. https://developer.oculus.com/blog/asynchronous-timewarp-on-oculus-rift/, 2016. március. [Online; accessed September 23, 2023].

[6] Dean Beeler – Ed Hutchins – Paul Pedriana: Asynchronous spacewarp. https://developer.oculus.com/blog/asynchronous-spacewarp/, 2016. november. [Online; accessed Oktober 15, 2023].

[7] Akanksha Dixit – Smruti R. Sarangi: Minimizing the motion-to-photon-delay (mpd) in virtual reality systems, 2023.

[8] Mohammed S. Elbamby – Cristina Perfecto – Mehdi Bennis – Klaus Doppler: Toward low-latency and ultra-reliable virtual reality. *IEEE Network*, 32. évf. (2018) 2. sz., 78–84. p.

[9] FFmpeg: About ffmpeg. https://www.ffmpeg.org/about.html, 2006. [Online; accessed Oktober 24, 2023].

[10] Google: Asynchronous reprojection. https://developers.google.com/vr/discover/async-reprojection, 2018. szeptember. [Online; accessed Oktober 06, 2023].

[11] DAVID HEANEY: Vr timewarp, spacewarp, reprojection, and motion smoothing explained. https://www.uploadvr.com/reprojection-explained/, 2019. [Online; accessed September 23, 2023].

[12] ITU-T: Itu-t recommendation p.10: Vocabulary for performance and quality of service. https://www.itu.int/rec/T-REC-P.10-200607-S/en, 2006. [Online; accessed November 1, 2023].

[13] Ben Lang: Oculus launches asw 2.0 with positional timewarp to reduce latency, improve performance. https://www.roadtovr.com/oculus-launches-asw-2-0-asynchronous-spacewarp/, 2019. április. [Online; accessed Oktober 06, 2023].

[14] Ben Lang: Steamvr update brings motion smoothing to modern amd graphics cards. https://www.roadtovr.com/steamvr-update-brings-motion-smoothing-to-modern-amd-graphics-cards/, 2019. május. [Online; accessed Oktober 10, 2023].

[15] Tom Parsons: How to turn off motion smoothing on your tv. https://www.roadtovr.com/steamvr-update-brings-motion-smoothing-to-modern-amd-graphics-cards/, 2019. szeptember. [Online; accessed Oktober 10, 2023].

[16] Hossein Pishro-Nik: Mean squared error (mse). https://www.probabilitycourse.com/chapter9/9_1_5_mean_squared_error_MSE.php, 2014. [Online; accessed November 1, 2023].

[17] Jan-Philipp Stauffert – Florian Niebling – Marc Erich Latoschik: Latency and cybersickness: Impact, causes, and measures. a review. *Frontiers in Virtual Reality*, 1. évf. (2020). ISSN 2673-4192.
URL https://www.frontiersin.org/articles/10.3389/frvir.2020.582204.

[18] SteamVR: Introducing steamvr motion smoothing. https://store.steampowered.com/news/app/250820/view/2898585530113853534, 2018. november. [Online; accessed Oktober 10, 2023].

[19] OpenCV Team: About. https://opencv.org/about/. [Online; accessed November 1, 2023].

[20] tomshardware: Oculus reveals asynchronous spacewarp, lowers vr minimum spec. https://www.tomshardware.com/news/

asynchronous-spacewarp-lowers-min-spec-vr,32826.html, 2016. október. [Online; accessed Oktober 06, 2023].

[21] Unity: Unity. https://unity.com/. [Online; accessed November 1, 2023].

[22] J. M. P. van Waveren: The asynchronous time warp for virtual reality on consumer hardware. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology*, VRST '16 konferenciasorozat. New York, NY, USA, 2016, Association for Computing Machinery, 37–46. p. ISBN 9781450344913.
URL https://doi.org/10.1145/2993369.2993375. 10 p.

[23] Z. Wang – A. C. Bovik – H. R. Sheikh – E. P. Simoncelli: Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13. évf. (2004. április) 4. sz., 600–612. p.

[24] Wikipedia: Asynchronous reprojection. https://en.wikipedia.org/wiki/Asynchronous_reprojection, 2019. augusztus. [Online; accessed Oktober 07, 2023].

[25] xinreality: Asynchronous spacewarp. https://www.xinreality.com/wiki/Asynchronous_Spacewarp, 2016. november. [Online; accessed Oktober 06, 2023].