

ETDK-dolgozat

Furu Árpád

Zsigmond - Jurj Mark

XXIX. reál- és humántudományi Erdélyi Tudományos Diákköri Konferencia (ETDK)

Informatika II.: innovatív számítástechnikai termékek, alkalmazások szekció

Kolozsvár, 2026. május 14–17.

Realm

Többfelhasználós terepkészítő alkalmazás kiterjesztett valóságot használva



Szerzők:

Furu Árpád

III. év, Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Zsigmond - Jurj Mark

III. év, Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Témavezetők:

dr. Sulyok Csaba, egyetemi adjunktus,

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Mátis Krisztián, szoftverfejlesztő, SOFTECH SRL

Szász Arnold-Levente, szoftverfejlesztő, SOFTECH SRL

Szilágyi Krisztián-Attila, szoftverfejlesztő, SOFTECH SRL



XXIX. ERDÉLYI TUDOMÁNYOS DIÁKKÖRI KONFERENCIA REÁL- ÉS HUMÁNTUDOMÁNYOK

CONFERINȚA ȘTIINȚIFICĂ STUDENTEASCĂ DIN TRANSILVANIA, EDIȚIA A 29-A
29TH TRANSILVANIA STUDENTS' SCIENTIFIC CONFERENCE
ȘTIINȚE REALE ȘI UMANISTE // FORMAL AND EMPIRICAL SCIENCES



Realm: Többfelhasználós terepkészítő alkalmazás kiterjesztett valóságot használva

Furu Árpád, III. év, Babeș–Bolyai Tudományegyetem, Matematika és Informatika Kar
Zsigmond - Jurj Mark, III. év, Babeș–Bolyai Tudományegyetem, Matematika és Informatika Kar

dr. Sulyok Csaba, egyetemi adjunktus,
Babeș–Bolyai Tudományegyetem, Matematika és Informatika Kar
Mátis Krisztián, szoftverfejlesztő, SOFTECH SRL
Szász Arnold-Levente, szoftverfejlesztő, SOFTECH SRL
Szilágyi Krisztián-Attila, szoftverfejlesztő, SOFTECH SRL

A Realm alkalmazást az XREAL Air 2 Ultra okos szemüveg által nyújtott funkciók tesztelésére fejlesztik. Az okos szemüveg funkcióit a Beam Pro okos telefon segítségével sikerül kihasználni, ide tartozik a kezek és a síkok detektálása. Ezeket a funkciókat alapul véve tervezik meg a világok létrehozását, valamint az ezekhez való csatlakozást.

A dolgozat a rendszer főbb architekturális elemeit, a háttér-infrastruktúrát, valamint az alkalmazás játékmenethez kapcsolódó megoldásokat mutatja be. Központi szerepet kapnak azok a mechanizmusok, amelyek lehetővé teszik a 3D objektumok dinamikus betöltését, az AR-térben történő elhelyezését és manipulációját, továbbá a többfelhasználós együttműködéshez szükséges valós idejű szinkronizációt.

A Realm nemcsak AR-szemüveggel használható, hanem önálló mobilalkalmazásként is képes teljes értékű élményt biztosítani. A bemutatott rendszer igazolja, hogy az AR-hardverek és a mobilplatformok egyaránt alkalmasak közös, interaktív digitális környezetek megvalósítására.



Témavezetői igazolás

Alulírott *dr. Sulyok Csaba, Mátis Krisztián, Szász Arnold-Levente és Szilágyi Krisztián-Attila* igazoljuk, hogy *Furu Árpád és Zsigmond - Jurj Mark* hallgatók a(z) *Realm: Többfelhasználós terepkészítő alkalmazás kiterjesztett valóságot használva* című dolgozatát az alulírottak szakmai irányításával készítették el, és javasoljuk az említett tudományos munka bemutatását a 2026-os XXIX. Reál- és Humántudományi Erdélyi Tudományos Diákköri Konferencián (ETDK).

Mindezek mellett igazoljuk, hogy a dolgozat megfelel a korszerű tudományos kritériumoknak, nem OTDK-n vagy nemzetközi szintű konferenciákon már bemutatott dolgozat, már megvédett államvizsga vagy magiszteri dolgozat, sem korábban már publikált dolgozat.

dr. Sulyok Csaba, egyetemi adjunktus,

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Mátis Krisztián, szoftverfejlesztő, SOFTECH SRL

Szász Arnold-Levente, szoftverfejlesztő, SOFTECH SRL

Szilágyi Krisztián-Attila, szoftverfejlesztő, SOFTECH SRL

Kolozsvár, 2026. október 16.

Tartalomjegyzék

1. Bevezető	1
2. Követelmények és rendszerterv	2
2.1. Célrendszer és fejlesztői környezet	2
2.2. Funkcionális követelmények	3
2.3. Nem-funkcionális követelmények	4
2.4. Rendszerkép: jelenetek és fő komponensek	4
2.5. A két üzemmód modellje	4
3. A játéktér fő moduljai (játékoldal)	6
4. A rendszer megvalósítása	9
4.1. Főbb funkcionalitások	9
4.2. Architektúra	17
5. Munkamenet bemutatása	20
5.1. A szakmai gyakorlat	20
5.2. Alkalmazott technológiák	20
6. Következtetések és továbbfejlesztési lehetőségek	23

1. Bevezető

Az elmúlt években a kiterjesztett valóság (Augmented Reality, AR) hardverei és szoftveres ökoszisztémái jelentős fejlődésen mennek keresztül, és egyre elterjedtebbé válnak az interaktív, digitális objektumok valós térben való megjelenítésére képes AR szemüvegek és mobil platformok. Mindazonáltal ezen technológiák adaptációja továbbra is kihívásokat tartogat mind a fejlesztési folyamatok, mind a felhasználói élmény terén, hiszen az eltérő eszközök különféle bemeneti módszereket — például érintést, kéz-követést és gesztusvezérlést — és eltérő futtatási környezeteket követelnek meg. Az AR klasszikus definíciója értelmében a rendszer a valós világot virtuális elemekkel egészíti ki, valós időben és térben precíz illesztéssel [1].

Ezen lehetőségek kiaknázása hívta életre a Realm alkalmazást, amely egy virtuális terepkészítő platform formájában ölt testet. A projekt a Softech Codespring nyári szakmai gyakorlatának keretein belül valósul meg egy kétfős fejlesztőcsapat munkájának eredményeként, amelyet *Mátis Krisztián*, *Szász Arnold-Levente* és *Szilágyi Krisztián-Attila* mentorok támogatnak. A Realm elsődleges célkitűzése egy olyan AR játéktér (gamespace) megalkotása, amely egyszerre funkcionál pályaszerkesztőként és interaktív játszható térként. Az alkalmazás asztali játék jellegű, ami azt jelenti, hogy a felhasználók egy valós, sík felületen — például egy asztallapon vagy a padlón — hozzák létre és pozicionálják a játék elemeit, amelyeket azután aktív játék módban is használhatnak.

A fejlesztés kiindulópontja az volt, hogy az okoszemüveghez egy bemutató demóalkalmazás kerüljön kidolgozásra. A kezdeti ötletelési fázisban számos ígéretes, de nehezen megvalósítható téma merül fel; ezek közül a fejlesztőkhöz közel álló játékkészítő alkalmazás koncepciója bizonyul a legalkalmasabbnak.

A jelen dolgozat elsődleges célja a szoftver játékkoldali megoldásainak és architektúrájának részletes elemzése. A dolgozat két fő területet fed le: az első a végfelhasználói élményre összpontosít, tárgyalva az AR játéktér inicializálását, a szerkesztési interakciókat, az állapotmentést, valamint a két üzemmód közötti váltási logikát; a második terület a rendszer technológiai hátterét mutatja be, kitérve a globális architektúrára, a 3D objektumok futásidejű importálására és a multiplayer funkciók hálózati megvalósítására.

2. Követelmények és rendszerterv

A projekt hardveres alapját az *XREAL Air 2 Ultra*¹ kiterjesztett valóság szemüveg, valamint a hozzá optimalizált *Beam Pro*² okostelefon biztosítja. Az Android operációs rendszert futtató telefonhoz csatlakoztatott szemüveg egy USB-C kábelén keresztül kommunikál, és második kijelzőként funkcionál. Ebben a felállásban a számítási feladatokat maga az okostelefon végzi el a gyártó által biztosított segédalkalmazás támogatásával. Mivel az átlagos okostelefonok nem AR alkalmazások futtatására lettek tervezve, bizonyos funkciók a Beam Pro eszközre korlátozódnak. A tesztelések során kizárólag ezzel az eszközzel sikerült az elvárható működést elérni; a gyártó dokumentációja szerint más kompatibilis telefonokkal³ is lehetséges az alkalmazás futtatása, ezek vizsgálatára azonban a fejlesztés keretében nem kerülhetett sor. Ennek kompenzálására egy egyszerűbb telefonos felület is rendelkezésre áll, amelyet a szemüveggel nem rendelkező játékosok tudnak használni.

2.1. Célszisztem és fejlesztői környezet

A piacon fellelhető AR-projektek jelentős hányadánál a fejlesztők különálló alkalmazásokat készítenek a hagyományos okostelefonos, illetve szemüveges felhasználási módokhoz. Ez a megközelítés redundáns fejlesztői munkát eredményez — külön grafikus felület, eltérő bemenet-kezelés és inicializálási metódusok szükségesek —, valamint töredezetté teszi a felhasználói élményt. Ennek elkerülése érdekében a Realm egyetlen integrált Android alkalmazásként működik, amely két eltérő üzemmódot támogat.

Az **XR mód** (XREAL szemüveggel) esetén, amennyiben a rendszer érzékeli a csatlakoztatott XREAL szemüveget, az XR alrendszer automatikusan inicializálódik. Ilyenkor a rendszer betölti az XR környezethez tartozó objektumhierarchiát, és aktiválja a térbeli UI-bemeneteket. Ez a mód kezdetektálást, síkdetektálást, építési és szerkesztési funkciókat, objektumimportálást, valamint többjátékos módot biztosít.

A mobil mód (szemüveg nélkül) esetén hagyományos érintésalapú bemenet lép működésbe, a funkciók a hardveres korlátokhoz igazodnak. A mobilalkalmazás objektumok importálását és megosztott világokhoz való csatlakozást tesz lehetővé; a navigációt képernyőn megjelenő joystick biztosítja (1. ábra.).

¹<https://next.xreal.com/us/air2ultra/>

²<https://www.xreal.com/beampro>

³<https://docs.xreal.com/XREALDevices/Compatibility>



1. ábra

A kettős üzemmód működésének kulcsa egy közös logikai mag: a játék alapvető mechanikái, a jelenetkezelés, az objektumok állapota és a mentési rendszer egységes marad, míg a megjelenítési és beviteli modulok felcserélhető réteggént illeszkednek a rendszerre [11].

Az alkalmazás fejlesztéséhez a **Unity 6** keretrendszer van használva, ezen belül **AR Foundation** a síkfelismeréshez, **XR Management** és **XR Interaction Toolkit** a térbeli interakciókhoz [11]. A szemüveg integrációját az XREAL ökoszisztéma biztosítja [12].

2.2. Funkcionális követelmények

A rendszer célja nem egy teljes játékmechanikai keretrendszer megvalósítása, hanem egy olyan digitális játéktér-infrastruktúra létrehozása, amelyben a felhasználó építhet, szerkeszthet, objektumokat importálhat, valamint az elkészült világot más játékosokkal közösen használhatja. Ennek megfelelően a fő funkcionális követelmények a következők:

1. **Játéktér létrehozása:** sík felület felismerése, valamint az induló objektumok elhelyezése.
2. **Szerkesztés:** blokkok elhelyezése és törlése kézgesztusokkal, vizuális cella-indikátor támogatásával.
3. **Tárgyinterakció:** beépített vagy importált objektumok megfogása, mozgatása és skálázása.
4. **Objektumimportálás:** futásidőben betölthető, felhasználó által biztosított .obj kiterjesztésű modellek támogatása.

5. **Mentés és visszatöltés:** a játéktér állapotának perzisztálása JSON formátumban.
6. **Jelenetek közötti átmenet:** *Menu* → *Editor* → *Game* váltás a releváns világállapot átadásával⁴.
7. **Többjátékos működés:** a közös világállapot szinkronizálása több kliens között.

2.3. Nem-funkcionális követelmények

A rendszerrel szemben több nem-funkcionális elvárás is megfogalmazható. A **robustus indítás** követelménye szerint a szemüveg jelenlétének detektálása után a rendszer XR módban induljon, ellenkező esetben stabil mobil módot biztosítson. A **valós idejű interakció** elvárása értelmében az objektummanipuláció és a vizuális visszajelzés késleltetése minimális legyen. Az **egyszerű bővíthetőség** igénye alapján új építőelemekkel vagy importált objektumokkal a játéktér könnyen kibővíthető legyen. Végül a **megbízható állapotkezelés** feltétele, hogy a világállapot lokálisan tárolható, illetve hálózaton keresztül frissíthető legyen.

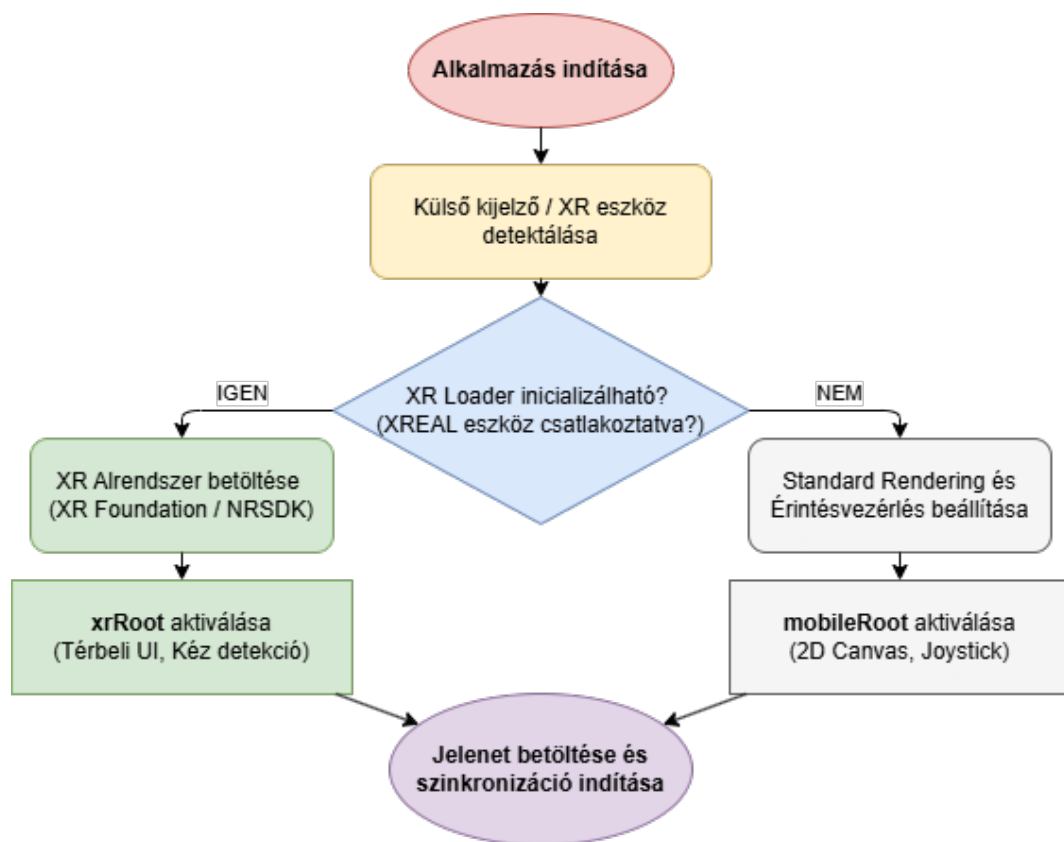
2.4. Rendszerkép: jelenetek és fő komponensek

A projekt három fő Unity jelenetre épül. A **Menu** jelenet a világ kiválasztását vagy létrehozását, a csatlakozást és az alap navigációt biztosítja. Az **Editor** jelenet a játéktér szerkesztését, az építést, a törlést és az objektumimportálást teszi lehetővé. A **Game** jelenet a létrehozott játéktér tényleges használatát valósítja meg.

2.5. A két üzemmód modellje

A projekt a két működési módot nem két külön alkalmazásként, hanem két, egymást kizáró gyökér-hierarchiával (**xrRoot** és **mobileRoot**) valósítja meg. Induláskor a mobil hierarchia aktív, majd ha a rendszer több kijelzőt érzékel, megkísérli az XR alrendszer inicializálását. Sikeres inicializáció esetén a mobil felület deaktiválódik, és az XR felület veszi át a szerepét (2. ábra.).

⁴A jelenetek fogalma részletesebben el van magyarázva a 2.4. alfejezetben.



2. ábra. Üzem mód választás rendszerindításkor

3. A játéktér fő moduljai (játékoldal)

Ebben a fejezetben a REALM játéktér-koncepcióját támogató legfontosabb modulok kerülnek bemutatásra. A cél nem a teljes forráskód részletes ismertetése, hanem azoknak a komponenseknek a kiemelése, amelyek közvetlenül az AR gamespace szerkeszthetőségéhez kapcsolódnak.

Üzem mód-váltás: XROnDemand

Az alkalmazás indulásakor a mobil mód aktív, és csak akkor vált XR módra, ha a rendszer detektálja a szemüveghez tartozó extra kijelzőt. Több kijelző esetén az extra kijelző aktiválódik, majd késleltetéssel — a race condition elkerülése érdekében — megtörténik az XR loader inicializálási kísérlete. Sikeres inicializáció esetén az XR UI modul bekapcsol, a hagyományos UI modul kikapcsol, végül a hierarchiacsere is lezajlik: az **xrRoot** aktívvá válik, a **mobileRoot** inaktívvá.

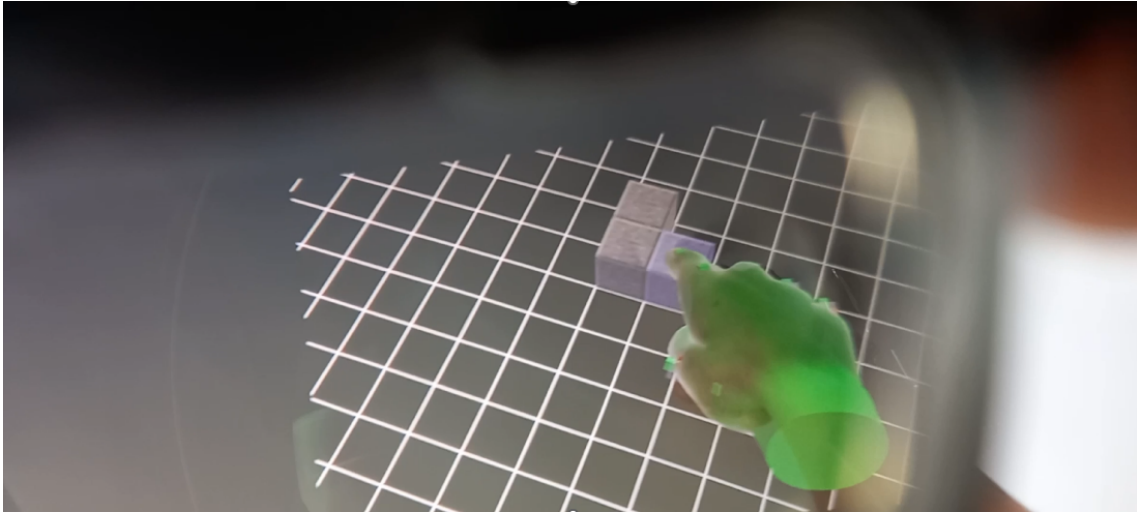
Ez a struktúra fejlesztői szempontból azt jelenti, hogy ugyanaz a menürendszer, mentés, jelenetkezelés és objektum-logika használható mindkét módban, kizárólag a perifériák cserélődnek.

AR gamespace inicializálása: ARSpawnManager

A játéktérnek szüksége van egy kezdő keretre, amely magában foglalja a gridet, a spawnert és az alap UI elemeket. Az **ARSpawnManager** a legközelebbi követett síkot választja ki, majd a kamera síkra vetített helyéhez képest spawnolja a beállított prefabokat. Ez egy praktikus kompromisszum: elegendően jó alapélményt nyújt, ugyanakkor nem igényel bonyolult kalibrációt.

Grid-alapú szerkesztés: PlacementSystem

A **PlacementSystem** felel a cella-indikátor megjelenítéséért és a következő elem elhelyezési logikájáért. A rendszer raycast alapján kiválasztja az aktuális grid cellát, majd a felület normálvektorából kiszámítja a szomszédos cellát, lehetővé téve ezzel, hogy a felhasználó a már meglévő elemek mellé vagy rá tudjon építeni. Ez a megoldás stabil, mivel az elhelyezés nem folyamatos koordinátákon, hanem diszkrét rácspontokon történik (3. ábra.).



3. ábra. Az XR nézetben megjelenő rácsos elrendezés, kézkövetéssel.

Kéz-követés és gesztusok: **InputManager**

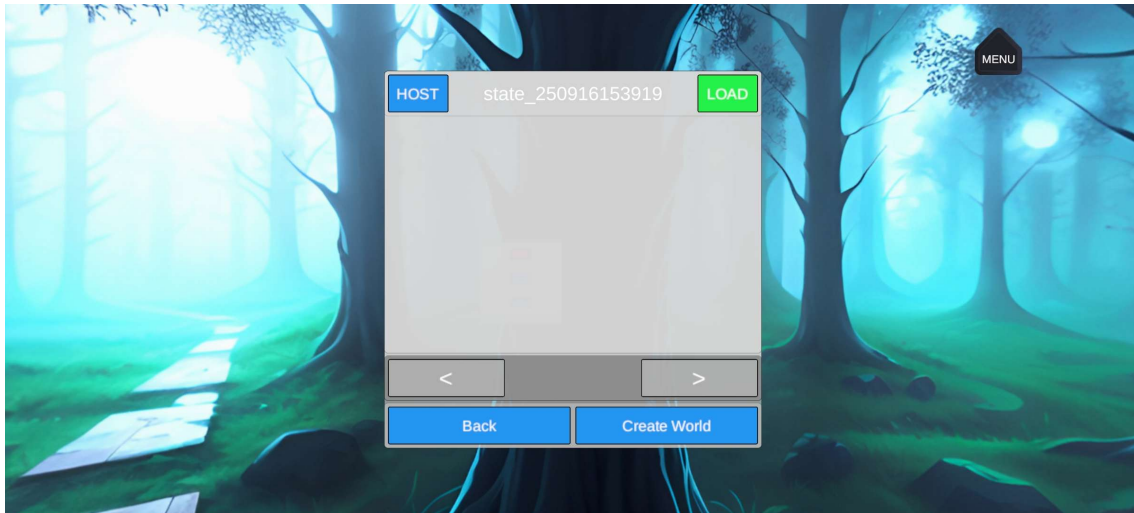
Az **InputManager** szerepe kettős: egyrészt összegyűjti az aktuális kézpozíciót (indexujjhegy), másrészt a gesztusfelismerés eredményét egyszerű állapotként tárolja, jelezve, hogy az elhelyezési vagy törlési gesztus aktív-e. A dolgozat szempontjából lényeges, hogy a gesztus eredménye leválasztott a szerkesztési logikáról: a **PlacementSystem** csupán egy logikai értéket kérdez le. Ez azt eredményezi, hogy a gesztusfelismerés később lecserélhető — például más gesztusmodellre — anélkül, hogy a világépítés logikája jelentősen változna.

Akció kiváltása ütközéssel: **FingerCollisionDetector**

Az elhelyezés és törlés a fizikai világra emlékeztető mechanikával történik: az ujjhegy collider hozzáér a célhoz. Ennek előnye, hogy a felhasználó számára természetes interakciót biztosít: nem szükséges menüből gombot megnyomni, elegendő a megfelelő gesztussal megérinteni a célcellát vagy blokkot.

Objektum-azonosítás és lépték: **SceneObjectID** és **ObjectSizeNormalizer**

A játéktérben az objektumoknak egyedi azonosítóra van szükségük a mentés, visszatöltés és hálózati szinkronizáció céljából. A **SceneObjectID** ezt GUID-generálással oldja meg. Az **ObjectSizeNormalizer** gondoskodik arról, hogy a vizuális méretek kezelhetőek maradjanak: a befoglaló doboz legnagyobb dimenzióját egy előre meghatározott célértékre normalizálja.



4. ábra. A világlista az appon belül.

Világállapot mentése: **StateSaverLoader** és **JsonSaveManager**

A szerkesztett játéktér csak akkor tekinthető teljes értékűnek, ha visszatölthető állapotban is megőrizhető. A mentési folyamat során a mentendő szülőobjektum összes gyereke bejárásra kerül, és eltárolódik a pozíció, a rotáció, a skála, valamint az objektum neve és tagje. A **JsonSaveManager** az adatokat a Unity **persistentDataPath** könyvtárába menti, így a világok az eszközön maradnak. Ezzel a megoldással a Menu jelenetben könnyen listázhatók a mentett állapotok, az Editor és a Game jelenet pedig ugyanazt az állapotot tölti vissza.

4. A rendszer megvalósítása

A rendszer specifikációjában a *játéktér* (gamespace) egy olyan perzisztens állapotot reprezentál, amelyben a különböző entitások — építőblokkok, külső forrásból importált modellek, a mozgatható szemetes és a betöltő kocka — pontos térbeli pozícióval, rotációval és mérettel bírnak, továbbá ez az állapot szabadon szerkeszthető és bármikor újratölthető.

A következőkben a projekt azon részei kerülnek részletesebb bemutatásra, amelyek az objektumok importálásához, a többjátékos funkcionalitáshoz, a szerver működéséhez és a világállapotok tárolásához kapcsolódnak. Mivel az alkalmazás Unity környezetben készült, a példakódok C# nyelven íródtak.

4.1. Főbb funkcionalitások

Kéz- és síkdetektálás

A kéz- és síkdetektálás megvalósításához a Unity beépített *Hand Tracking* és *Plane Detection* rendszere kerül felhasználásra, amelyeket az *XR Foundation* keretrendszer biztosít. Az érzékelési adatokat maga az XREAL szemüveg szolgáltatja a Unity számára az *XREAL SDK*-n keresztül — ezekhez az alacsony szintű szenzor folyamatokhoz az alkalmazás közvetlenül nem fér hozzá, azok teljes egészében a hardver és az SDK által kerülnek kezelésre. Fontos megjegyezni, hogy ezek a funkciók kizárólag XREAL által támogatott eszközökön⁵ működnek; kompatibilis eszköz hiányában a kéz- és síkdetektálás nem érhető el.

Objektumok importálása

A projekt tervezésekor az egyik kiemelt cél a rendszer *sandbox*-jellegének megteremtése volt. Ez azt jelenti, hogy a felhasználók saját készítésű modelleket, figurákat vagy építményeket is elhelyezhetnek az alkalmazás digitális játékterében, így a környezet különböző társasjátékok vagy egyedi jelenetek felépítésére is alkalmassá válik.

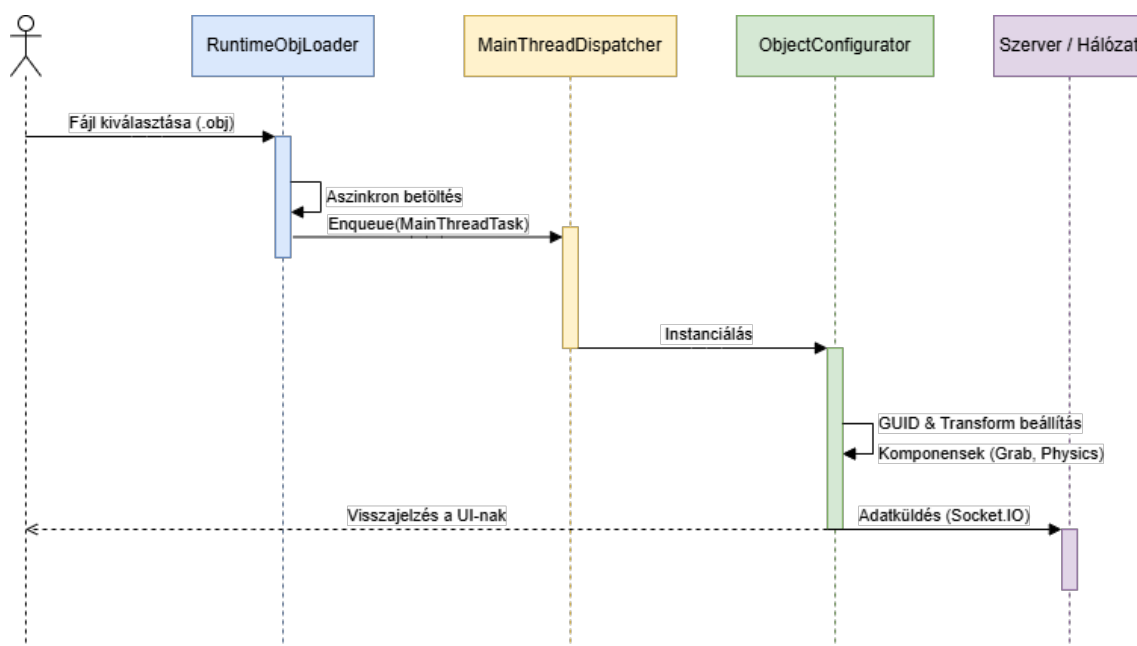
A futásidejű objektumimportálást két, a Unity *Asset Store*-ban⁶ elérhető csomag valósítja meg: a *Runtime Object Importer*⁷ és a *Runtime File Browser*⁸. A *Runtime File Browser* biztosítja a fájlböngésző felületet, amely lehetővé teszi, hogy a felhasználó kiválassza a betölteni kívánt

⁵<https://docs.xreal.com/XREALDevices/Compatibility>

⁶Külső fejlesztők által kínált digitális erőforrások és grafikai elemek beszerzésére szolgáló piactér.

⁷<https://assetstore.unity.com/packages/tools/modeling/runtime-obj-importer-49547>

⁸<https://assetstore.unity.com/packages/tools/gui/runtime-file-browser-113006>



5. ábra. Egy objektum életciklusa.

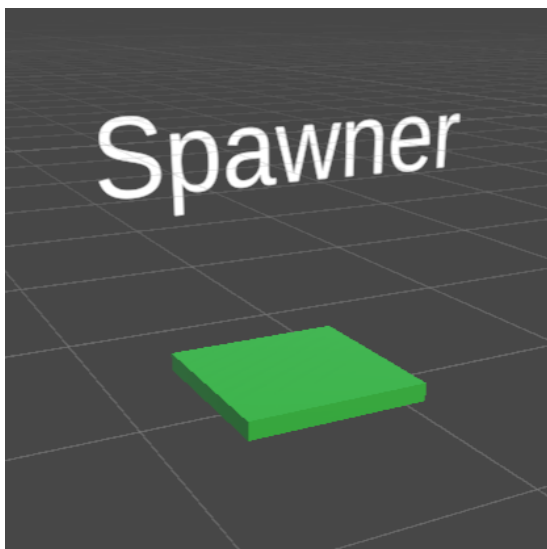
3D-s modellt (.obj fájlt), amelyből ezt követően a *Runtime Object Importer* készíti el a megfelelő Unity **GameObject**-et⁹.

A megoldás gyakorlati nehézségét az jelenti, hogy a **GameObject**-ek létrehozása a Unity működési modellje miatt a *fő szálon* kell történjen. Emiatt nagyobb objektumok importálásakor a játék futása rövid időre megakadhat, ami különösen mobil eszközön válik észrevehetővé. Ennek kezelésére egy **MainThreadDispatcher** (5. kódrészlet.) osztály szolgál, amely sorba rendezi a fő szálon végrehajtandó feladatokat, és azokat több kisebb részre felosztva hajtja végre — így az importálás ugyan a fő szálon történik, de nem egyetlen képkocka alatt, ezért kevésbé okoz érzékelhető megakadást.

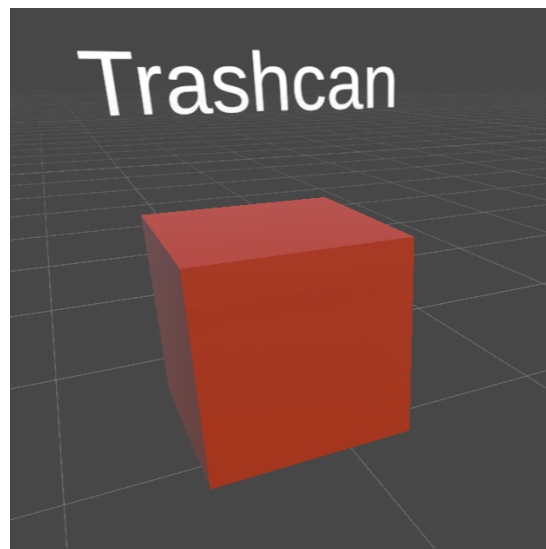
Az importált objektumok nem válnak automatikusan interaktívvá, ezért erre a célra egy **ObjectConfigurator** (6. kódrészlet.) segédosztály áll rendelkezésre. A **SetupObject** módszer segítségével az objektum megkapja azokat a komponenseket, amelyek szükségesek a térbeli manipulációhoz — például a **XRGrabInteractable** és **XRGeneralGrabTransformer** elemeket (3. illetve 10. sor) —, valamint ugyanitt kerül sor a kezdeti pozíció, a gravitációs viselkedés és a méret normalizálásának beállítására is (4-8. sorok).

A méretnormalizálás azért szükséges, mert a különböző forrásból származó modellek jelentősen eltérő skálázással rendelkezhetnek. Az alkalmazás ezért automatikusan egységesíti a betöltött objektum méretét egy erre kialakított algoritmus alapján (24. sor). Ekkor kapja meg az

⁹A **GameObject** egy gyűjtőnév azoknak a játékon belüli elemeknek, amiket a Unity fel tud ismerni és dolgozni tud velük.



6. ábra. A *Spawner* objektum



7. ábra. A *TrashCan* objektum

objektum az egyedi azonosítóját is, amelyet a mentés és a hálózati szinkronizáció során használ a rendszer (18. sor). Miután a konfiguráció befejeződik, az objektum egy közös szülőobjektum alá kerül, amely a játéktérben található importált elemek nyilvántartásáért felel; ugyanez a struktúra kezeli az építőkockákat is.

A felhasználói élmény támogatására a rendszer egy *Spawner* objektumot is megjelenít a detektált síkon, a játékos előtt kissé jobbra (6. ábra). Ez egy zöld, vízszintes négyzet alakú elem, amely egyszerre csak egyetlen importált tárgy elhelyezését teszi lehetővé, biztosítva ezzel az objektumok közötti helyes fizikai elkülönülést. *Collider*-ek segítségével valósítható meg a modellek közötti interakció, az ütközés. Hogyha két modell ugyan azt a térbeli helyet foglalja el megjelenéskor, akkor a collider-ek nemdeterminisztikus viselkedést eredményezhetnek. A *Spawner* mellett a játékos bal oldalán jelenik meg a *TrashCan*, egy piros kocka, amely lehetővé teszi a felesleges objektumok gyors törlését: ha egy importált objektum hozzáér ehhez az elemhez, eltávolításra kerül (7. ábra).

Egy világ állapotának mentése

A világállapot mentése a rendszer egyik alapvető funkciója, mivel ez teszi lehetővé, hogy a felhasználók a korábban létrehozott játéktérhez később visszatérjenek, és hogy ugyanaz az állapot több kliens között is megosztható legyen. A mentés során a rendszer három fő objektumcsoportot kezel: a konstans objektumokat (például a *Spawner*-t és a *TrashCan*-t), az importált objektumokat, valamint az építőkockákat.

Minden mentett objektumhoz tartozik egy egyedi azonosító, név, típus, pozíció, forgatás

Mező neve	Típus	Leírás
ID	Integer / UUID	Az objektum egyedi azonosítója a session-ben
Name	String	A betöltendő .obj fájl vagy erőforrás neve
Type	Enum / String	Az objektum típusa (pl. építőkocka, importált objektum)
Position	Vector3 (x,y,z)	Térbeli koordináták
Rotation	Quaternion	Az objektum forgatási adatai
Scale	Float / Vector3	Az objektum normalizált mérete

1. táblázat. Egy mentett objektum metaadatai a JSON struktúrában

```

1  {
2    "sceneObjectID": "7dfe39b4-b012-4ceb-9432-45513437cee3",
3    "name": "SpawnerObject",
4    "tag": "Spawner",
5    "pos": {
6      "x": 0.445375055,
7      "y": 1.20868623,
8      "z": 0.935973048
9    },
10   "rot": {
11     "x": -0.0141569721,
12     "y": -0.046161525,
13     "z": 0.0142303705,
14     "w": 0.998732328
15   },
16   "scale": {
17     "x": 0.200000003,
18     "y": 0.0199999996,
19     "z": 0.200000003
20   }
21 }
```

1. kódrészlet. Példa egy mentett objektumra JSON formátumban.

és méret. A típus alapján dönthető el, hogy az adott elemet milyen módon kell visszatölteni, mivel az építőkockák és az importált objektumok betöltése eltérő folyamatot követ. A név a megfelelő fájl vagy erőforrás azonosítására szolgál, az ID pedig a szerverrel való kommunikáció és a szinkronizáció szempontjából is kulcsszerepet játszik.

A világalapot JSON formátumban kerül tárolásra, mert ez a forma egyszerűen szerializálható és könnyen továbbítható a kliens és a szerver között, egyaránt támogatva a lokális perzisztenciát és a hálózati szinkronizációt.



8. ábra. A szerverrel való kommunikáció folyamata

A szerver és a többjátékos mód működése

Az alkalmazás egy Python-alapú szervert használ, amely *Kubernetes* környezetben fut. A szerver feladata a publikált világok tárolása és a többjátékos működés támogatása. A világokat a *Dungeons and Dragons* terminológiája nyomán *session*-nek nevezik, amelyeket a szerver egy *MongoDB* adatbázisban tárol. Az 8. ábrán a kliens-szerver-adatbázis kapcsolat látható.

Egy *session* tartalmazza a világ elmentett állapotát, valamint az ahhoz tartozó importált objektumokat. Az állapotadatokat külső *MongoDB* adatbázisban tárolják, míg az objektumfájlok tömörítve, a szerverhez kapcsolódó tárolóban helyezkednek el. A szerver két fő feladatot lát el: egyrészt biztosítja, hogy egy újonnan csatlakozó kliens megkapja a világ aktuális állapotát, másrészt gondoskodik arról, hogy a játékosok módosításai a többi résztvevőhöz is eljussanak. Amikor egy játékos csatlakozik egy adott *session*-höz, lekéri a szervertől az aktuális állapotfájlt, majd szükség esetén letölti a hiányzó objektumokat is. Ha ezt követően valamelyik kliens módosítja egy objektum helyzetét vagy állapotát, a változást a szerver fogadja, majd továbbítja a *session* többi résztvevője felé, biztosítva ezzel a csatlakozott játékosok közötti folyamatos szinkronizációt.

Jelenetbetöltés és állapotátadás

A jelenetváltásokért a **SceneLoader** felel. Ez a komponens aszinkron módon tölti be a céljelenetet, és igény esetén folyamatsávot vagy egyéb visszajelző felületet is megjelenít. A kiválasztott állapot — például a világ neve vagy az aktuális szerkesztési állapot azonosítója — egy statikus C# osztályon keresztül adódik át a jelenetek között a **SceneCommunicator** segítségével. Ez a megközelítés tudatosan minimalista: kis méretű projekt esetén csökkenti a globális menedzserobjektumok számát, és egyszerűvé teszi a *Menu*, *Editor* és *Game* jelenetek közötti adatátadást. Későbbi bővítésnél indokolt lehet a felelősségi körök erősebb rétegezése, a dolgozat céljaihoz azonban ez a megoldás megfelelőnek bizonyul.

```
1  if (Display.displays.Length >= 2)
2  {
3      yield return mgr.InitializeLoader();
4      if (mgr.activeLoader != null)
5      {
6          mgr.StartSubsystems();
7          ActivateXRUIModule();
8          xrRoot.SetActive(true);
9          mobileRoot.SetActive(false);
10     }
11 }
```

2. kódrészlet. XR inicializálás és gyökér-hierarchia váltás (rövidítve)

XR inicializálás igény szerint

A két üzemmód működésének központi eleme a **XROnDemand** komponens, amelynek célja, hogy az XR alrendszer ne induljon el minden eszközön automatikusan, hanem kizárólag abban az esetben, ha a szemüveg ténylegesen jelen van. Az implementáció egy gyakorlati heurisztikát alkalmaz: ha a **Display.displays** tömb mérete legalább 2, akkor valószínűsíthető, hogy külső megjelenítő — például a szemüveg — csatlakozott. Ilyenkor a komponens késleltetve megpróbálja inicializálni az XR rendszert a Unity XR Management API-ján keresztül. Ennek során megtörténik az XR loader inicializálása a **InitializeLoader()** segítségével, az alrendszerek indítása a **StartSubsystems()** hívással, a UI-bemenet átváltása úgy, hogy a **XRUIInputModule** aktívvá válik, míg az **InputSystemUIInputModule** kikapcsol, végül a megfelelő gyökér-hierarchia aktiválása, vagyis az **xrRoot** bekapcsolása és a **mobileRoot** deaktiválása.

Ennek az elrendezésnek két fontos előnye van. Egyetlen build is elegendő, nincs szükség külön mobil- és XR-alkalmazásra. Emellett alacsony kockázatú fallback valósul meg: ha nincs szemüveg, vagy az XR inicializálás sikertelen, az alkalmazás stabil mobil módban fut tovább. Az alábbi kódrészlet rövidítve szemlélteti az XR igény szerinti inicializálásának fő lépéseit(2. kódrészlet.).

Játéktér inicializálása AR síkon

A játéktér a valós térben egy detektált síkra kerül. Ehhez a rendszer az AR Foundation síkdetektálását használja a **ARPlaneManager** komponensen keresztül. Az **ARSpawnManager** minden képkockában megkeresi a kamera számára legközelebbi követett síkot, majd erre

```
1 if (Physics.Raycast(fingerPos, fingerDir, out RaycastHit hit,
2     1f, buildableLayerMask))
3 {
4     Vector3Int hitCell = grid.WorldToCell(hit.point);
5     Vector3Int offsetCell = GetAdjacentCell(hitCell, hit.normal);
6     position = grid.GetCellCenterWorld(offsetCell);
7     return true;
8 }
```

3. kódrészlet. Raycast alapú cella-pozíció számítás (rövidítve)

a síkra — pontosabban a kamera síkra vetített pozíciójához képest — elhelyez egy előre definiált objektumlistát. Ez a megközelítés azt eredményezi, hogy a felhasználónak nem kell kézzel kijelölnie a kezdőpontot a virtuális tartalom számára; a játéktér elemei a felhasználó környezetében, értelmes kezdő elrendezésben jelennek meg. A kezdeti elhelyezés után a síkdetektálás kikapcsolható, ami teljesítmény és stabilitás szempontjából is előnyös.

Szerkesztési interakciók: kézjelek – visszajelzés – művelet

A szerkesztési mód három lényeges elemre épül. Az első az **InputManager** által kezelt gesztusállapot, amely két logikai változót tárol — **isPlacementGestureActive** és **isDestroyGestureActive** — és a kézcsontváz-driverből (**XRHandSkeletonDriver**) kiolvassa az indexujj vizuális pozícióját (**XRHandJointID.IndexTip**). A második elem a **PlacementSystem** által működtetett cella-indikátor: ha az elhelyezési gesztus aktív, a rendszer sugárvetéssel meghatározza a célcellát, majd oda helyezi az indikátort. A grid-alapú megközelítés segít abban, hogy a felhasználó szabályosabb építményt tudjon létrehozni, apró, nehezen reprodukálható eltérések nélkül. A harmadik elem a **FingerCollisionDetector** által végzett kontaktusalapú akció: ha az ujjhegy trigger collidere belép egy *Blueprint* jelölésű cellába, blokk-spawn történik; ha *Block* jelölésű objektumba lép és a törlési gesztus aktív, a blokk törlésre kerül.

Ez a csővezeték egyszerű, ugyanakkor jól bővíthető: új gesztus vagy új céltag hozzáadásával könnyen megvalósítható például forgatás, színválasztás vagy speciális objektumok elhelyezése is. A cellakiválasztás a **PlacementSystem**-ben sugárvetéssel történik; az alábbi kódrészlet a gridcella kiválasztásának és a cellaközéppont meghatározásának lényegét mutatja (3. kódrészlet.).

```
1 var grab = obj.AddComponent<XRGrabInteractable>();
2 obj.AddComponent<Rigidbody>();
3 var transformer = obj.AddComponent<XRGeneralGrabTransformer>();
4 transformer.allowTwoHandedScaling = true;
5 transformer.maximumScaleRatio = 1f;
6 transformer.minimumScaleRatio = 0.01f;
7 grab.useDynamicAttach = true;
8 grab.selectMode = InteractableSelectMode.Multiple;
```

4. kódrészlet. Megfogás és kétkezes skálázás beállítása (rövidítve)

Objektummanipuláció XR módban

Az XR mód egyik legfontosabb előnye a természetesebb tárgykezelés lehetősége. A **ObjectConfigurator** feladata, hogy a Unity XR Interaction Toolkit megfelelő komponenseit hozzáadja az objektumokhoz: az **XRGrabInteractable**-t a megfogáshoz, az **XRGeneralGrabTransformer**-t a kétkezes skálázáshoz, a **Rigidbody**-t megfelelő fizikai beállításokkal (kinematic mód, gravitáció nélkül), valamint a **SceneObjectID**-t az egyedi azonosításhoz.

Fontos gyakorlati probléma, hogy az importált modellek mérete sok esetben inkonzisztens. Ezt a **ObjectSizeNormalizer** kezeli: a rendererek alapján meghatározza az objektum határoló dobozát, majd egy célméretre — például 0.3 méterre — normalizálja a legnagyobb dimenziót, ami jelentősen javítja a játéktér használhatóságát, mivel az objektumok közel azonos léptékben jelennek meg. Az alábbi részlet a fő komponensek konfigurációját szemlélteti (4. kódrészlet.).

4.2. Architektúra

Az alkalmazás logikája a *MVC (Model-View-Controller)* mintát követi [3], amennyiben az állapotkezelés, a felhasználói felület és az interakciók kezelése elkülönített komponensekben valósul meg. Ez a szétválasztás megkönnyíti a tesztelést és a módosítást, bár Unity környezetben a *MonoBehaviour*-alapú osztályok természetüknél fogva nem mindig illeszkednek tisztán a mintához.

Jelenetkezelés

Az alkalmazás több jelenetet használ: menüt, játékszerkesztőt és játékteret. Az ezek közötti váltásért a **SceneLoader** komponens felel. A jelenetváltások közben szükséges adatok — például a session azonosítója vagy a kiválasztott világra vonatkozó információk — átadását a **SceneCommunicator** végzi.

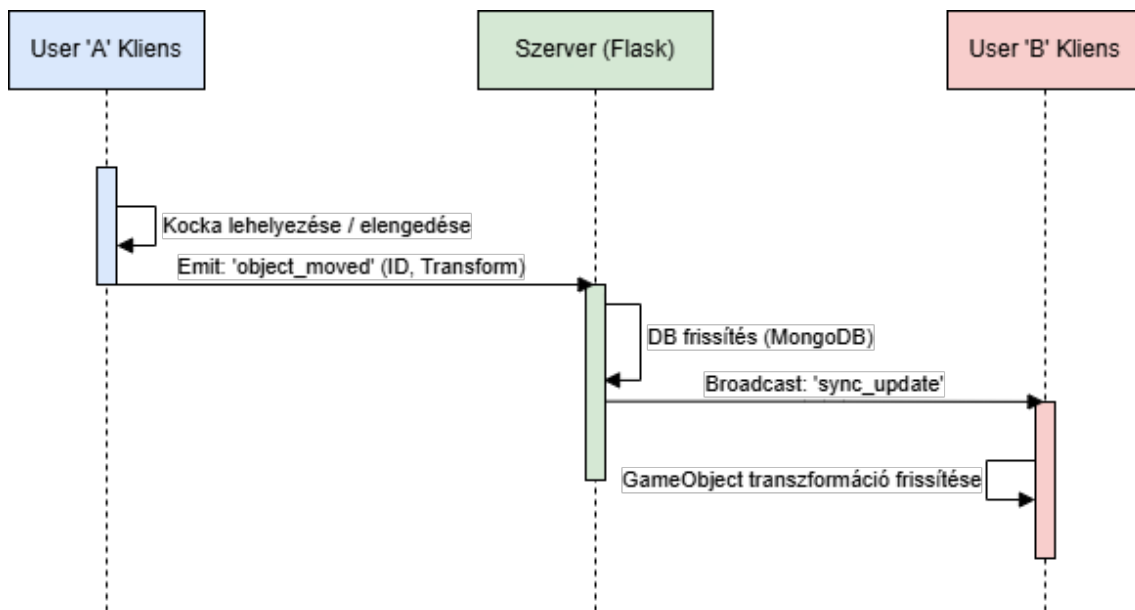
Fájlkezelés és objektumbetöltés

A felhasználók által kiválasztott objektumok kezelésében több komponens működik együtt. Az **ObjectCopier** feladata, hogy a kiválasztott objektumokat az alkalmazás saját könyvtárába másolja. A **RuntimeObjLoader** ezt követően betölti ezeket az objektumokat a játéktérbe — a betöltést a *Dummiesman OBJ Loader*¹⁰ nyílt forráskódú könyvtár végzi. Az **ImportedObjectsList** nyilvántartja, hogy mely objektumok vannak már letöltve, így elkerülhető a felesleges ismételt letöltés vagy másolás. A fájlkezelő felhasználói felületét a **FileBrowserCanvas** biztosítja, amely Unity UI-elemekre épül, ugyanakkor a virtuális térhez igazítva, kézgesztusokkal is kezelhető.

Játékállapot és vezérlési logika

A különböző komponensek együttműködésében központi szerepet játszik a **MainThreadDispatcher** (5. kódrészlet.), amely egy *singleton* mintát megvalósító osztályként kezeli azokat a feladatokat, amelyek végrehajtásához a Unity fő szála (*Main Thread*) szükséges. Ez különösen fontos az objektumbetöltésnél, mivel a Unity API legtöbb funkciója — így a használt OBJ betöltő is — csak ezen a szálon működik megbízhatóan. A komponens lelke egy belső, szálbiztosan kezelt várakozási sor (*_executionQueue*), amelybe bármely háttér szálról delegálhatók műveletek az *Enqueue* metódus segítségével.

¹⁰<https://assetstore.unity.com/packages/tools/modeling/runtime-obj-importer-49547>



9. ábra. Egy objektum frissítése két játékos esetén

A **MainThreadDispatcher** nem valódi párhuzamosítást végez, hanem a háttérszálakon előkészített feladatok végrehajtását ütemezi vissza a fő szálra. A megvalósítás a Unity `Update` ciklusát használja fel: minden képkocka frissítésekor a diszpécser zárolja a sort, majd egymás után kiveszi és meghívja az összes várakozó `Action` delegáltat. Ez a mechanizmus biztosítja, hogy a számításigényes előkészületek (például fájlok beolvasása vagy hálózati kommunikáció) külön szálon fussanak, míg az erőforrás-igényes Unity-specifikus műveletek — mint a mesh-ek példányosítása vagy a textúrák alkalmazása — szabályozott keretek között, a fő szál blokkolása nélkül menjenek végbe. A *Singleton* minta és a `DontDestroyOnLoad` hívás garantálja, hogy a diszpécser a teljes alkalmazás futása alatt elérhető maradjon, biztosítva a folyamatos és konzisztens feladatkezelést.

Valós idejű hálózati szinkronizáció

A többjátékos élmény integritását a virtuális tér állapotának folyamatos és alacsony késleltetésű szinkronizációja biztosítja. Amikor egy felhasználó interakcióba lép egy objektummal — például megragad egy entitást a térben —, a hozzá tartozó eseménykezelő *listener*-ek aktiválódnak. Az interakció fennállása alatt a kliens minden képkockafrissítéskor (`Update` ciklus) adatsomagot küld a központi szervernek a `Socket.IO` protokollon keresztül. Ez az üzenet tartalmazza az objektum egyedi `SceneObjectID` azonosítóját, valamint a frissített transzformációs adatokat: a térbeli pozíciót, a rotációt és az aktuális skálázási tényezőt.

A Python/Flask alapú háttérrendszer az üzenet vételekor azonnal frissíti a memóriában tárolt

aktuális világállapotot (*session state file*), majd ezzel párhuzamosan perzisztálja az adatokat a MongoDB adatbázisban is. Az állapot frissítését követően a szerver *broadcast* mechanizmust alkalmazva továbbítja a módosításokat a session-ben tartózkodó összes többi résztvevőnek. A hálózati forgalom optimalizálása érdekében a szerver intelligens szűrést végez: az adatokat csak a többi kliensnek küldi ki, az eredeti forrást — a módosítást kezdeményező felhasználót — kihagyja a szórásból, megelőzve ezzel a redundáns visszacsatolást és az esetleges lokális állapot-oszcillációt.

A fogadó oldalon a kliensek az üzenet beérkezésekor azonosítják az érintett objektumot, majd a `MainThreadDispatcher` segítségével ütemezik a transzformációs adatok frissítését. Ez a folyamat garantálja, hogy a más felhasználók által végzett manipulációk folytonos vizuális élményként, minimális késleltetéssel jelenjenek meg minden kliensnél, fenntartva a közös játéktér globális konzisztenciáját. A 9. ábrán a kommunikáció leegyszerűsített vázlata látható.

5. Munkamenet bemutatása

Ebben a fejezetben a csapattárrsal való munkafelosztásról, a szakmai gyakorlat során elvégzett feladatokról, valamint a megvalósítás főbb lépéseiről esik szó.

5.1. A szakmai gyakorlat

A szakmai gyakorlat a Softech SRL Codespring Mentorprogram keretein belül zajlik, a projekt elkészítését **Mátis Krisztián**, **Szász Arnold-Levente** és **Szilágyi Krisztián-Attila** mentorok segítik. A gyakorlat első két hetében a fejlesztéshez használt eszközök képességeinek feltérképezése zajlott. Ez idő alatt az XREAL által publikált dokumentáció részletes áttanulmányozására, valamint kisebb tesztprojektek elkészítésére került sor az elérhető funkciók kipróbálása céljából. Miután a megvalósítható funkcionalitások és a technológiai korlátok világossá váltak, megkezdődött a végleges projekt elkészítése.

A projekt Scrum keretrendszer szerint, sprintekre bontva halad [10]. A feladatok nyomon követésére a GitLab *Issue Board* szolgál, amely lehetővé teszi a teendők átlátható szervezését és az aktuális fejlesztési állapot követését.

5.2. Alkalmazott technológiák

Az alábbiakban az alkalmazás megvalósításában szerepet játszó technológiák kerülnek bemutatásra fejlesztési kategóriák szerint.

Fejlesztői környezetek és eszközök

Unity 6

A projekt alapját képező, nyílt forráskódú és ingyenes *játékmotor*. Az AR funkciók megvalósításához az *XReal AR SDK* kerül integrálásra [12], amely az XREAL szemüveg szenzorainak adatait teszi elérhetővé. A síkdetektálás, kézdetektálás és egyéb szenzor-alapú funkciók ezen a keretrendszeren keresztül érhetők el.

Unity Editor

A Unity 6 fejlesztői felülete, amelyen keresztül a jelenet- és projektszerkesztés, valamint a build folyamat elvégzése történik.

Visual Studio Code

Az elsődleges kódszerkesztő környezet. A C# *IntelliSense*, a *Git*-integráció és az egyéb bővítmények megkönnyítik a fejlesztést. A szerveroldali fejlesztéshez is ez a környezet kerül felhasználásra, *Python* bővítménnyel kiegészítve.

Git

Verziókezelő rendszer, amelyen keresztül a projekt menedzselése történik. A kód egy privát *GitLab Repository*-ban kerül tárolásra, amely támogatja a csapatmunkát és a verziókövetést.

Programozási nyelvek

C#

Az alkalmazás fő fejlesztési nyelve. A teljes kliensoldali logika, beleértve a Unity *component*, *behaviour* és *manager* osztályokat, C# nyelven íródik.

Python

A szerveroldali fejlesztés nyelve. A *Flask* keretrendszer segítségével egy JSON-alapú API kerül kialakításra, amely az állapotok lekérdezését és a módosítások szinkronizációját biztosítja.

Könyvtárak és csomagok

Dummiesman OBJ Loader

Egy nyílt forráskódú könyvtár *.obj* modellek betöltéséhez. Mivel az alkalmazásban futásidejű modellimportálásra van szükség, ez a könyvtár kerül integrálásra a **RuntimeObjLoader** komponens létrehozásához. A komponens aszinkron módon tölti be az objektumokat, megelőzve ezzel a játék lefagyását.

Unity Asset Store

Számos segédeszközt biztosít, amelyek közül kettő kerül felhasználásra: a **Runtime Object Importer** és a **Runtime File Browser** csomagok. Ezek az eszközök felgyorsítják a fejlesztést, mivel az OBJ betöltésre és a fájlböngészésre vonatkozó kód megírása nulláról nem szükséges.

Infrastruktúra és kommunikáció

MongoDB

Egy *NoSQL* adatbázis, amelyben a világok állapotai tárolásra kerülnek [6]. Mivel az alkalmazás nagy mennyiségű strukturálatlan adatot kezel, a dokumentum-alapú megközelítés megfelelő választásnak bizonyul. Az állapotok JSON formátumban tárolódnak, így könnyen szerializálhatók és deszerializálhatók.

Kubernetes

Egy konténer-orkesztrációs platform, amelyen a Python szerver fut. A szerver Docker konténerben kerül csomagolásra, amely biztosítja a hordozhatóságot és az egyszerű telepíthetőséget. A platform lehetővé teszi a manuális skálázást és a magas rendelkezésre állást.

WebSocket

A valós idejű kommunikációhoz választott protokoll. A HTTP kéréseken felül a WebSocket kapcsolat lehetővé teszi, hogy több játékos között minimális késedelemmel történjen az adatok szinkronizálása.

6. Következtetések és továbbfejlesztési lehetőségek

Az alkalmazás jelenlegi formájában egy jól körülhatárolt, működőképes alapot biztosít egy AR játéktérhez: a két üzemmódot egy alkalmazáson belül kezeli, és a felhasználó számára kézgesztusokra épülő szerkesztést tesz lehetővé. Azonban a projekt korántsem mutatja meg a lehetőségek maximumát — a továbbiakban a felhasználói élmény finomhangolására és a játékmenet mélyítésére kerülhet sor.

Pontosabb üzemmód-detektálás

A jelenlegi megoldás a több kijelző jelenlétét használja heurisztikaként. Ez gyakorlatban működő kompromisszum, de továbbfejleszthető:

- explicit felhasználói választás lehetősége (váltás „Mobil mód” és „XR mód” között),
- részletesebb hibakezelés implementálása az XR inicializálás elakadása esetére.

Szerkesztési eszköztár és tartalom bővítése

A jelenlegi Editor a blokk-elhelyezést és törlést mutatja be. Az építés az egyik legérdekesebb funkció, azonban a kézgesztus-alapú detektálás jelenleg nem elég intuitív. A fejlődés irányai:

- **Változatosabb építőelemek:** több textúrájú és formájú prefab választása (paletta bővítése),
- **Interakció finomítása:** az építés újragondolása egy felhasználóbarátabb, precízebb megközelítéssel,
- **Szerkesztő funkciók:** forgatás, rács-felbontás állítása, valamint visszavonás (undo/redo) bevezetése,
- **Metaadatok:** mentési adatok (név, leírás, bélyegkép) kezelése a világokhoz.

Objektumimport és teljesítménynövelés

A 4.1 fejezetben ismertetett objektumbetöltési megoldás jelenleg lassú, amelyet elsősorban az OBJ fájlformátumból való átalakítás okoz. A stabil pipeline érdekében a következő fejlesztések jönnek szóba:

- több formátum támogatása és az anyag- és textúrakezelés javítása,
- automatikus collider generálás és mobil-specifikus optimalizálás a betöltési idő csökkentésére,
- skálázási és forgatási alapbeállítások mentése objektumonként.

Hálózati kommunikáció és biztonság

Az architektúra robusztusabbá tétele a többfelhasználós környezetben az alábbi irányokban lehetséges:

- **Titkosított kommunikáció:** titkosítás bevezetése a szerver és a kliensek között,
- **Jelszóval védett világok:** jelenleg egy világba bárki beléphet az ID alapján; a jelszavas védelem lehetővé tenné a privát szobák létrehozását,
- **Felhasználókezelés:** hitelesítési rendszer beépítése az architektúrába.

Felhasználói felület és platformtámogatás

A jelenlegi UI nehezen kezelhető, friss felhasználók számára meredek a tanulási görbe. A fejlesztés két fő iránya a következő:

- **UI szépítése:** elérhetőbb menürendszer és a gombok érzékenységének növelése,
- **iOS-támogatás:** bár az alkalmazás elsődlegesen Androidra és a szemüvegre készül, a mobil kliens iOS-re való portolásával szélesebb felhasználói kör válik elérhetővé.

Játékmenet réteg

A dolgozat fókusz a játéktér infrastruktúrájára irányul; a következő logikus lépés a konkrét játékszabályok bevezetése:

- körökre osztott (turn-based) társasjáték-mechanikák a virtuális táblán,
- fizikai alapú minijátékok (például dobókocka, lövedékek kezelése),
- kooperatív puzzle elemek a felépített környezetben.

Összegzés

A Realm egy olyan kiterjesztett valóságot alkalmazó terepkészítő platform, amely egyetlen alkalmazáson belül képes kezelni az AR-szemüveges és a hagyományos mobilos felhasználási módot. A fejlesztés során megmutatkozott, hogy a két üzemmód közös logikai magra épülő megvalósítása nemcsak a fejlesztői munkát egyszerűsíti, hanem a felhasználói élményt is egységesebbé teszi. Az alkalmazás bemutatja, hogy fogyasztói szintű AR-hardver is elegendő alapot nyújt egy valós idejű, többfelhasználós interaktív rendszer megvalósításához. A megvalósított funkciók együttesen egy olyan platformot alkotnak, amely társasjáték-jellegű környezetek felépítésére és közös használatára egyaránt alkalmas. A rendszer számos ponton tovább finomítható, különösen a szerkesztési interakciók és a hardverkompatibilitás terén, ugyanakkor szilárd alapot nyújt egy teljesebb AR ökoszisztéma kialakításához.

Hivatkozások

- [1] Ronald T. Azuma. *A Survey of Augmented Reality*. Malibu, California: Hughes Research Laboratories, 1997.
- [2] I. Fette és A. Melnikov. „The WebSocket Protocol”. *RFC 6455* (2011).
- [3] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.
- [4] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. 2nd. O'Reilly Media, 2018.
- [5] Joseph Hocking. *Unity in Action: Multiplatform game development in C#*. 3rd. Manning, 2022.
- [6] *MongoDB Documentation*. <https://www.mongodb.com/docs/>. MongoDB Inc. 2024.
- [7] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014.
- [8] Nigel Poulton. *The Kubernetes Book*. Leanpub, 2020.
- [9] Dieter Schmalstieg és Tobias Hollerer. *Augmented Reality: Principles and Practice*. Addison-Wesley Professional, 2016.
- [10] Jeff Sutherland. *Scrum: The Art of Doing Twice the Work in Half the Time*. Crown Business, 2014.
- [11] *Unity User Manual 6*. <https://docs.unity3d.com/6000.0/Documentation/Manual/index.html>. Unity Technologies. 2024.
- [12] *XREAL NRSdk Documentation*. <https://docs.xreal.com/>. XREAL. 2024.

Fontosabb programkódok listája

Példakódok

Itt találhatók a fontosabb kódrészletek, amik segítettek az objektumok importálásában és betöltésében.

```
1 public class MainThreadDispatcher : MonoBehaviour {
2     public static MainThreadDispatcher Instance { get; private set; }
3     private static readonly Queue<Action> _executionQueue = new Queue<Action>();
4
5     void Awake() {
6         if (Instance != null && Instance != this) {
7             Destroy(gameObject);
8             return;
9         }
10        Instance = this;
11        DontDestroyOnLoad(gameObject);
12    }
13
14    public static void Enqueue(Action action) {
15        if (action == null)
16            return;
17        lock (_executionQueue) {
18            _executionQueue.Enqueue(action);
19        }
20    }
21
22    void Update() {
23        lock (_executionQueue) {
24            while (_executionQueue.Count > 0) {
25                _executionQueue.Dequeue().Invoke();
26            }
27        }
28    }
29
30    void OnDestroy() {
31        if (Instance == this)
32            Instance = null;
33    }
34 }
```

5. kódrészlet. MainThreadDispatcher.cs

```

1  public static class ObjectConfigurator {
2      public static void SetupObject(GameObject obj) {
3          var grabInteractable = obj.AddComponent<XRGrabInteractable>();
4          obj.AddComponent<Rigidbody>();
5          var rigidbody = obj.GetComponent<Rigidbody>();
6          rigidbody.useGravity = false;
7          rigidbody.isKinematic = true;
8          rigidbody.interpolation = RigidbodyInterpolation.Interpolate;
9
10         var generalGrabTransformer = obj.AddComponent<XRGeneralGrabTransformer>();
11         generalGrabTransformer.allowTwoHandedScaling = true;
12         generalGrabTransformer.maximumScaleRatio = 1f;
13         generalGrabTransformer.minimumScaleRatio = 0.01f;
14
15         grabInteractable.useDynamicAttach = true;
16         // ... a grabInteractable további beállításai
17
18         obj.AddComponent<SceneObjectID>();
19         var listener = obj.AddComponent<ObjectUpdateListener>();
20
21         grabInteractable.selectEntered.AddListener(_ => listener.HandlePickedUp());
22         grabInteractable.selectExited.AddListener(_ => listener.HandleDropped());
23
24         ObjectSizeNormalizer.NormalizeToSize(obj, .3f);
25     }
26 }

```

6. kódrészlet. ObjectConfigurator.cs