Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Vizi Előd

ANDROID ALAPÚ JÁTÉK FEJLESZTÉSE UNITYBEN MOZGÁSKÖVETŐ SZENZOROK SEGÍTSÉGÉVEL

Virtuális valóság technológia használata Cardboard segítségével

KONZULENS

Hideg Attila

BUDAPEST, 2019

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2019. 12. 12.

...…….
Vizi Előd

Összefoglaló

Személy szerint úgy gondolom, hogy nagy jövő áll még Virtuális- és Kiterjesztett valóságon (VR és AR) alapuló technológiák előtt és rengeteg kiaknázatlan lehetőséget rejtegetnek még magukban. Annak ellenére, hogy most már legalább 5 éve elég erősen jelen vannak a játékiparban, illetve más többnyire 3D-s megvalósításokkal kapcsolatos (tervezés, lakberendezés, filmek világa, művészetek) iparágakban, még mindig gyerekcipőben jár ez a technológia és a felhasználási területének elterjedése.

Mivel szakképzett Személyi edzőként fontosnak tartom az egészséges és aktív életmódot, valamint én magam is megszállottja vagyok a testmozgásnak, adott volt az alap és jött az ötlet, hogy valami olyan alkalmazás fejlesztésébe fogjak ami segít az embereket mozgásra bírni, kimozdítani a videójátékozni imádó fiatalokat is a komfortzónájukból úgy, hogy mindezt élvezettel tegyék, sőt vágyjanak rá.

Az Önálló laboratórium tárgy keretein belül sikerült megismerkedjek a Notch Pioneer Kit 3D mozgáskövető rendszerrel és a hozzá tartozó telefonos (esetemben Android) alkalmazással illetve fejlesztői környezettel (SDK).

A Szakdolgozatom folyamán először egy működő szolgáltatás (Android Service) megvalósítására törekedtem, ami biztosítja a valós idejű kommunikációt a mozgáskövető szenzorok és a Unityben összerakásra kerülő telefonos játék között. Majd a projekt folytatásaként, erre a szolgáltatásra alapozva egy olyan telefonos játékot írtam, amiben a játékos feladata, hogy egy helyben állva, kizárólag a felső testét használva olyan pozicíót vegyen fel, amivel el tudja kerülni, hogy a felé közelítő fal a mögötte lévő medencébe lökje. Mindezen mozdulatokat az említett szenzorok segítségével valós időben lekövetve.

A játék ráadásul virtuális valóságbeli élményt nyújt, ami már egy egyszerű Google Cardboard szemüveg használatával is élvezhető, sőt mi több a jelenlegi verzió erre lett kitalálva és ezzel teszteltem magamon és ismerőseimen.

Abstract

Personally, I think there is a huge future for any kind of Virtual Reality and Augmented Reality based technologies. However, in the last 5+ years, these technologies were strongly present in the Game industry alongside some other 3D based fields such as: architecting, designing, furnishing and so forth. Still, there are a lot of untapped fields full of opportunities using both VR, AR, and Mixed Reality technologies.

As a certified Personal Trainer, I think health should be primary for everybody and living an active lifestyle is pretty important for long-lasting life. Since I am also a fitness enthusiast I came up with the idea to combine software development with my hobby. In this project, my number one goal was to create a demo application which gets more and more people to stand up and burn some calories. Also to get those youngsters who spend most of their time playing video games to step out from their comfort zone without even realizing it. Moreso to get them to enjoy moving and jumping around.

In the last semester, I had a project in which I started to work, understand and get to know a 3D Motion Tracking Sensor System called Notch Pioneer Kit. I also got myself introduced to the Android SDK of this system.

At the beginning of my project implemented an Android Service to support real-time communication between the sensors and a sample Unity App. Based on this service designed and developed an Android game in Unity where the player has to get into the right position using only his/her upper body to avoid to get pushed into a water pool by a wall moving towards the player. The player's body position is captured in real-time using previously mentioned motion tracking sensors.

The Virtual Reality feeling can be experienced using a simple VR headset. I tested and optimized the game with a Google Cardboard based on my own and my friends' user experience.

1 Bevezetés

1.1 Projekt célja, motiváció, indíttatás, ötlet részletesebb leírása és bemutatása

Az elmúlt időszakban, ami körülbelül 1-1,5 évet ölel fel egyre jobban kezdtek érdekelni a különböző Virtuális valósággal (VR - Virtual Reality), Kiterjesztett valósággal (AR - Augmented Reality) és Kevert valósággal (MR - Mixed Reality) kapcsolatos témák, projektek és fejlesztések.

Személy szerint úgy gondolom, hogy nagy jövő áll még ezen technológiák előtt és rengeteg kiaknázatlan lehetőséget rejtegetnek még magukban. Annak ellenére, hogy most már legalább 5 éve masszívan jelen vannak a játékiparban, illetve más, többnyire 3D világgal kapcsolatos (tervezés, lakberendezés, filmek világa) iparágakban, még mindig gyerekcipőben jár ez a technológia és a felhasználási területének elterjedése. Amiről talán még nem is sejtjük, hogy mi lesz az igazán átütő erejű. Gondoljunk csak bele, a telefonokkal is hasonlóképp volt.

Amikor megjelentek az első hordozható nyomógombos telefonok, de még az első okostelefonok megjelenésekor sem gondoltuk, hogy napjainkra szerves részét fogják képezni ezek az eszközök a mindennapi életünknek és ekkora befolyással lesznek ránk. Használhatjuk őket könyv helyett olvasásra, TV helyett filmezésre, rádió helyett podcastek (online rádióműsor) hallgatására. Már akár játszásra is PC, laptop és konzolok helyett és lassan minden másra is amire eddig csak a komolyabb számítógépek voltak képesek.

Így adott volt, hogy a 6. féléves Önálló laboratórium tárgy keretein belül valamilyen ezekhez a technológiákhoz tartozó témán szeretnék dolgozni.

Épp ezért próbáltam minél következetesebb lenni az érdeklődési körömmel és a potenciális Szakdolgozat témámmal kapcsolatban és már a szakirányválasztás első félévében a Témalaboratórium nevű tárgy keretein belül elkezdtem ismerkedni a Unity keretrendszerrel, ami az egyik legelterjedtebb különböző 3D-s és VR-os szoftverek fejlesztésére, az Unreal Engine-el egyetemben.

Ezen technológiák és az egyetem mellett az elsődleges dolog ami leginkább foglalkoztat, ami a hobbim és egyben a szenvedélyem is az a fitnesz, testépítés, izom-, erő- és állóképesség fejlesztésének világa.

Szakképzett Személyi edzőként fontosnak tartom az egészséges és aktív életmódot, a rendszeres testmozgást és a tudatos és egészséges táplálkozást. Innen jött az ötlet, hogy valami olyan alkalmazás fejlesztésébe fogok ami segít az embereket mozgásra bírni, kimozdítani a videójátékozni imádó fiatalokat is a komfort zónájukból úgy, hogy mindezt élvezettel tegyék, sőt vágyjanak rá. Azt szeretném ha ezek a fiatalok vagy bárki aki túlnyomórészt passzív életet él, kezdjen rendszeresen mozogni. Az észrevételem az, hogy ha elérjük, hogy az adott személyt szórakoztassa és lekösse az adott tevékenység, ez esetben a játék, akkor észre sem veszi, hogy tulajdonképpen egy jól átgondolt edzésmunkát végez. Továbbá már az sem fogja zavarni ha épp megizzad eközben, mivel teljesen a játékra kell koncentrálnia.

Úgy a sportokban mint az egészségügyben, gyógypedagógiában, rehabilitációban hatalmas szerepet fognak játszani (és bizonyos helyeken már játszanak is) ezek a technológiák a közeljövőben. Különböző VR és AR alkalmazások segíthetnek embereknek a félelmeik leküzdésében, traumákon való túllendülésen vagy akár egy komolyabb sérülés után újra tanulni a végtagok mozgatását, a járást és megannyi más fantasztikus dolgot fogunk ezeknek a technológiáknak köszönni. Az oktatási rendszer reformját nem is említve, amiről úgyszintén egy külön dolgozatot lehetne írni.

Az eddig leírtakat felvázolva még az előző félév megkezdése előtt megkerestem a korábbi Témalaboratórium - Unity keretrendszerrel való ismerkedésben segítő konzulensem, hogy milyen lehetőségek lennének most a tanszéken ezekben a témákban? Mik azok amik a kiírt témák közül esetleg érdekelnének? Nagy szerencsém volt, mivel erre a megkeresésre egy remek hírt kaptam, miszerint pont a félév elején volt esedékes, hogy kapjon a tanszék egy mozgáskövető (Motion Tracking) rendszert a Notch Interfaces cég jóvoltából. Ezeket a kis hatszög alakú testre rögzíthető bluetooth szenzorokat pedig Notch-oknak hívják.

Kapva kaptam az alkalmon és lecsaptam erre a lehetőségre ami alapján az Önálló laboratórium tárgy céljának azt tűztük ki, hogy megismerkedjek ennek a rendszernek a működésével, a hozzá tartozó telefonos alkalmazással illetve fejlesztői környezettel (SDK), ami esetemben Android Studióban és Android készüléken zajlott. Illetve, hogy megnézzem és tanulmányozzam, hogy mi lenne a legideálisabb mód a

telefonos alkalmazás és a Unity projekt közötti kommunikációra, aminek valós időben kellene történnie és többnyire ez okozta a félév legnagyobb fejtörését. Erre a problémára végül a félév végén találtam meg a legjobb megoldást aminek implementálására a szakdolgozat projektem keretén belül került sor.

A szakdolgozatom első fázisában az említett mozgáskövető szenzorok és a Unityben összerakásra kerülő telefonos játék közötti valós idejű kommunikációs problémát áthidaló megoldásnak a megvalósítására törekedtem. Amihez egy háttérben futó szolgáltatás (Android Service) megfelelő implementálása és felhasználása vezetett.

Majd folytatásként, erre a szolgáltatásra alapozva egy olyan telefonos játékot írtam, amiben a játékos feladata, hogy helyben állva, kizárólag a felsőtestét használva olyan pozíciót vegyen fel, amivel el tudja kerülni, hogy a felé közelítő fal a mögötte lévő medencébe lökje. A mozdulatokat pedig az említett szenzorok segítségével valós időben leköveti az alkalmazás.

A játék ráadásul virtuális valóságbeli élményt nyújt, ami már egy egyszerű Google Cardboard szemüveg használatával is élvezhető, sőt mi több a jelenlegi verzió erre lett kitalálva és ezzel teszteltem magamon és ismerőseimen.

A játékélmény fokozása érdekében és a benne rejlő lehetőségek továbbfejlesztéséhez, a későbbiekben át lehet ültetni a jelenlegi verziót valamilyen komolyabb virtuális valóság alapú rendszerre, mint például az Oculus Rift vagy a PlayStation VR és kibővíteni újabb játékmódokkal, 12 szenzort használó mozgáskövetéssel, amivel már a térben való elmozdulás, a teljes test irányítása is lehetővé válna.

1.2 Használt keretrendszerek rövid bemutatása

1.2.1 Unity (version: 2019.2.7f2) [1]

A **Unity** egy multi-platform videójáték-motor, amelyet a Unity Technologies fejleszt. A Unity segítségével háromdimenziós illetve kétdimenziós videójátékokat, ezen kívül egyéb interaktív jellegű tartalmakat lehet létrehozni, például építészeti látványterveket vagy valós idejű háromdimenziós animációkat. Többek között előnye, hogy a szoftver képes nagyméretű adatbázisokat kezelni, kihasználni a kölcsönhatások és animációk képességeit, előre kiszámított vagy valós idejű világítást biztosítani.

Továbbá használható geometriai eszközcsomagok továbbítására, illetve viselkedési elemek hozzáadására egyes objektumokhoz.

A Unitynek két főbb alkotó része van: az egyik játékok fejlesztésére és tervezésére használható szerkesztő, a másik pedig maga a videójáték-motor, amely a végleges változat kivitelezésében nyújt segítséget, ám háromdimenziós modelleket nem képes létrehozni, így azokhoz mindenképp szükséges egy 3D modellező program.

Az egyszerű 2D-s játékoktól a bonyolultabb 3D-s stratégiai játékokon keresztül a brutális grafikával rendelkező action RPG-ig minden platform megtalálható a repertoárban. Akár egy virtuális valóságbeli First Person Shooter vagy mint esetemben, egy Androidos ügyességi játék is létrehozható a Unity keretrendszert használva.

Eddigi tapasztalataim alapján mindig nehéz eldönteni egy projekt kezdetén, amennyiben erre nincs kikötés, hogy az adott fejlesztőkörnyezet melyik verzióját használjuk. Egy már stabilan működő de kicsit régebbi verziót válasszunk vagy használjuk a legújabb verziót, amiben még lehetnek hibák, de esetenként segíthetnek megoldani olyan problémákat, amikre az előző verziók még nem, vagy csak nagyon korlátoltan tudtak megoldást nyújtani.

Az előző félév folyamán a Unity **2018.2.10f** verzióját használtam, majd félév közepén jött egy 2019-es verzió, amit telepítettem is. Ám ezzel szomorúan tapasztaltam, hogy az addig probléma nélkül működő projektem olyan hibát produkált amire még a hivatalos fórumokon sem tudtak abban a pillanatban megoldást mondani.

Ebben a félévben annak ellenére, hogy tudtam a 2018-as verzióval biztosan működik a projektem úgy döntöttem, hogy inkább tiszta lappal kezdek, minden szoftvert frissítek a legújabb verzióra. Miután kipróbáltam, hogy reprodukálom az Önálló laboratórium tárgy keretében elkészült projektemet a korábbi dokumentációm [2] alapján és meggyőződtem arról, hogy az megfelelőképp működik, úgy gondoltam, hogy hasznosabb lehet maradni a legnaprakészebb verziónál, ami 2019 szeptemberében a **2019.2.7f2** volt.

Azóta már jött néhány frissítés és jelenleg a legaktuálisabb a **2019.2.11**-es verzió, de a korábban tapasztaltakból kiindulva a projekt lezárásáig továbbra is a 2.7f2-es működő kiadást használtam.

1.2.2 Android Studio

Az Android Studio esetében sokkal fontosabb a használt SDK és *buildToolsVersion*-ök száma:

- Android Studio 3.4
- Android SDK Tools: 26.1.1
- Android Platform Version:
  - API 28: Android 9.0 (Pie) revision 6
- build.gradle(Module: app) - ban:
  - compileSdkVersion 28
  - buildToolsVersion ‘28.0.3’
  - targetSdkVersion 28
  - minSdkVersion 19

Jelenleg az alábbi, legfrissebben kiadott Notch Android SDK verziót használom, ami úgyszintén a *build.gradle*-ben *(Module: app)* található a *dependencies* (függőségek) listában:

- `implementation ‘com.notch.sdk:sdk-android:1.1.355’`

A Notch-hoz tartozó Android SDK működését és installálását a következő fejezetekben mutatom be részletesen.

1.3 Fejlesztéshez használt eszközök

- **Mobiltelefon készülék:**

Huawei P10 Androidos készülék, 7.0-ás Android verzióval.

- **Mozgáskövető rendszer:**

Notch Pioneer Kit a Notch Interfaces Inc. jóvoltából

- **Laptop:**

ASUS ROG GL552J:

  - Processzor: Intel Core i7-4720HQ CPU @ 2.60 GHz
  - Video Card: NVIDIA GeForce GTX 950M 4GB
  - RAM: 16.0 GB
  - HDD: 1 TB
  - OS: Windows 10 Education 64-bit, x64-based processor

2 Mozgáskövető rendszer (MoCap) [3]

A motion capture, magyarul digitális mozgásrögzítés vagy mozgáskövetés egy olyan folyamat, melynek során mozgást rögzítenek, majd ezt egy digitális modellre ültetik át. Ezt a technológiát főként a szórakoztatóipar használja. Számítógépes és egyéb digitális illetve virtuális játékok gyártására, filmgyártásra. De úgyszintén alkalmazzák sportokban és a hadseregben is. Valamint ahogyan a bevezetőben már említettem, egyre elterjedtebbé válik az orvostudomány különböző területein is (rehabilitáció, szellemi betegségek gyógyítása és egyebek) ezeknek a rendszereknek az alkalmazása.

A filmgyártásban és a játékiparban az előadó mozgása kerül rögzítésre, majd a felvett adatokat a digitális karakter animálására használják. Manapság már az arc mozgásának különálló rögzítése is nagy hangsúlyt kap ezekben a folyamatokban. Az arckifejezéseket, artikulációkat is egyre élethűbben vissza lehet adni egy teljesen animált, többnyire élőszereplős felvételen is a modern mozgáskövető rendszerek használatával.

A mozgások digitális rögzítésére számos megoldást dolgoztak ki. Ezek általában a testre elhelyezett érzékelők (markerek) és ezek térbeli elmozdulását rögzítő eszközök (többnyire kamerák) típusában különböznek. A felvétel során általában több, de minimum kettő, kamerával követik nyomon a mozgást, majd ezt követően a markerek helyzetéből a számítógép háromszögeléses módszerrel (Rövid kitérő: Háromszögeléssel egy háromszög két csúcsának koordinátái, valamint a belső szögek ismeretének függvényében meghatározhatóak a harmadik csúcs koordinátái, ahol az ismert csúcsok a kamerák pozíciói, a meghatározandó csúcs pedig a marker pozíciója) meg tudja állapítani a színész vagy mozgásművész csontvázának helyzetét. Kulcsfontosságú a mintavételezés gyakoriságának megfelelő beállítása, ami másodpercenként akár pár száz is lehet, hogy garantálni tudják a rögzített mozgás folyamatosságát.

2.1 MoCap rendszerek típusai

A dolgozatnak nem célja ezen rendszerek minél részletesebb és tudományos megközelítésen alapuló leírása, de fontosnak tartom, hogy felületesen a teljesség igénye nélkül ebben a fejezetben ezeket bemutassam. Ezáltal megalapozva az ismereteket

későbbi tervezői döntések, választások és fennakadások illetve ezek orvoslásának megértéséhez. Ugyanis vannak olyan use-casek amiket rendszerenként eltérően szükséges kezelni, vagy esetenként egyáltalán nem kezelhetőek a felhasznált technológia nyújtotta korlátozások miatt.

2.1.1 Optikai rendszerek

Ezeknek a rendszereknek az alapját az optikai felvevő készülékek, a különböző típusú kamerák szolgáltatják. A rögzített mozgás folyamatosságát és pontosságát nagyban befolyásolja a kamerák minősége és milyensége. Ezekben a rendszerekben még lényeges különbséget a markerek típusa szolgáltat így az optikai alapú rendszereket az alábbi kategóriákba csoportosítják:

a. Passzív érzékelő alapú rendszerek

Régebb többnyire a passzív érzékelő alapú rendszerek voltak használatban. A passzív érzékelők többnyire valamilyen retroreflexiós technológián alapulnak és egy külső jelet vernek vissza, ezt a módszert visszatükrözésnek hívják.

A visszatükrözés [4] az a jelenség, melynek során a beeső hullám nagy része a beesés szögétől függetlenül a beesés irányával ellentétes irányba verődik vissza. Viszont itt nagyon fontos és kritikus tényező, hogy a kibocsátott hullámforrások minél nagyobb részét sikerüljön a hullámforrás felé visszaverni, hogy a visszavert hullám jól érzékelhető legyen. A mozgáskövető rendszerek esetében ezzel biztosítva az adott mozgás minél pontosabb és precízebb rögzítését.

Az aktív érzékelő alapú rendszerekkel ellentétben itt nincsenek kábelek vagy elektromos kütyük az alany teste körül. Helyette több tíz vagy több száz kis gumilabdát rögzítenek a testre vagy egy ruhára, speciális reflexiós ragasztószalaggal.

b. Aktív érzékelő alapú rendszerek

Az aktív érzékelő alapú rendszerek többnyire ledek különböző intenzitású villogtatásán alapulnak. Itt ismételten kritikus tényező, hogy a villogás megfelelően gyors legyen. A ledek villogását kamerák rögzítik és egy szoftver dolgozza fel. Majd a térben elhelyezi a markereket, azok csillagászati

navigációhoz hasonló módszer használatával kiszámolt relatív pozíciójuk alapján.

c. Idő-modulált aktív érzékelő alapú rendszerek

Ez a módszer az eddigiektől is bonyolultabb számításokon alapszik. Itt egyszerre több marker mozgását is követik. A beérkező jelekből egy szoftver amplitudó modulálással különíti el az adott ID-val rendelkező érzékelő alapján rögzített mozgás útvonalát. Ez a módszer nagyobb számításigényű mint az eddigiek, ellenben pontosabb, kevésbé zajos a rögzített adat és a színész mozgása valós időben is követhetővé válik akár már az előre elkészített számítógépes grafikájú karakteren is.

d. Részben passzív észlelhetetlen érzékelő alapú rendszerek

Ennél a technikánál retroreflexív anyaghasználat vagy ledek helyett fotoszenzitív jelölőket használnak az optikai jelek dekódolására. Ilyen fotoszenzitív jelölők elhelyezésével az adott jelenetben nem csak azok adott pillanatbeli pozícióját, de az elmozdulás irányát, a beeső megvilágítást és a fényvisszaverődést is meg tudják határozni. Egy adott jelenetben nincs korlátozva ezen jelölők számának használata továbbá alacsonyabb sávszélesség is elég ezeknek a használata esetén.

e. Érzékelők nélküli (markerless) rendszerek

Vannak esetek amikor kényelmetlen és nem is célravezető különböző jelölőket elhelyezni körben a színész testén vagy az adott felületen. Ezekben az esetekben kizárólag a kamera rendszer által rögzített képre hagyatkozunk. Így jogosan merül fel az igény a kép- és videó feldolgozó algoritmusok nagyléptékű és precíz fejlesztésére. Az általam érintett mozgáskövetés területén a fő szempont és a lényeg, hogy a kamerák képét feldolgozó szoftverek minél pontosabban felismerjék és elkülönítsék az emberi formákat és azok mozgásait.

2.1.2 Nem optikai rendszerek

f. Inerciális (tehetetlenségi) rendszer [5][6]

Az inerciális mozgáskövető technológia többnyire kis inerciális szenzorokon, biomechanikai modelleken és szenzorfúziós algoritmusokon alapszik. Ezekben a rendszerekben a szenzorok által rögzített mozgás

modellezésére szolgáló adatok gyakran valamilyen vezeték nélküli (wireless, bluetooth) közegben továbbítódnak a feldolgozó eszközre, amely egy speciális szoftver segítségével ezeket feldolgozza, szükség esetén eltárolja és akár valós időben megjeleníti azt.

Az inercia alapú rendszerek IMU-kat (Inertial Measurement Units, magyarul inerciális mérő egységek) használnak. Ezekre az egységekre a dolgozatomban helyenként mint szenzorok hivatkozok.

A minél pontosabb mérési adatokhoz az IMU-k legtöbb esetben tartalmaznak giroszkópot, magnetométert és gyorsulásmérőt. Ezek segítségével mérhetjük és meghatározhatjuk a szenzor forgási sebességét amit egy szoftver dolgoz fel és egy digitális csontvázra (skeleton) vetít.

Az inerciális navigációs módszeren alapuló rendszer által rögzített mozgások szabadsági foka hat (6DoF [7]). Ami azt jelenti, hogy a háromdimenziós térben a részecske vagyis ez esetben az adott szenzor szabadon mozoghat előre/hátra (surge - áradás), fel/le (heave - emelkedés), jobbra/balra (sway - lengés) és foroghat a tér bármely irányába (yaw - legyezőmozgás vagy kitérés, pitch - bólintás, roll - gurulás), három egymásra merőleges tengely (X, Y, Z) mentén. Így elég kétharmad annyi szenzort használni, mint amennyi markerre szükség lenne ugyanazon mozgás rögzítésére egy optikai rendszerben. Mivel ebben a rendszerben egy felkaron és egy alkaron elhelyezett szenzor által meghatározható a könyökízület pontos helyzete, elmozdulása és szöge.

Az optikai rendszerekben használt markerekhez hasonlóan ebben a rendszerben is minél több IMU-t használunk annál pontosabb és természetesebben kinéző mozgást kapunk.

Itt nincs szükség külső kamerákra, pont ezért ez a megvalósítás sokkal többoldalúan használható. Könnyen hordozható rendszer, amelyet egyszerű telepíteni, alkalmazni és az árazását tekintve is több mindenki számára elérhető. Így akár már kis fejlesztő cégek, alacsony költségvetésű filmstúdiók is megengedhetik maguknak, hogy használják a MoCap technológia nyújtotta lehetőségeket.

Azonban a rendszer előnyei mellett fontos megemlíteni, hogy ez a módszer alacsonyabb pozíció pontossággal bír és minimális sodródás is bekerül a

rendszerbe, amik idővel összeadódnak és már a valóságtól jelentős, érzékelhető és zavaró eltérést eredményezhetnek.

Ilyen inerciális rendszer alapú a Notch Interfaces Inc. által fejlesztett és a projektemben használt Notch Pioneer Kit is, melynek további specifikációiról a következő alfejezetben lehet olvasni.

g. Mechanikai/gépi rendszer

Ez a megvalósítás lényegében egy exoskeleton amit jelen esetben úgy fordítanék le egyszerűen, hogy vázszerkezet. Ez a vázszerkezet az emberi csontváz alapján illeszkedik a testre. Képes követni a vázat viselő ember mozgását, az ízületekben azok mozgástartományának megfelelően elmozdulni, míg a csontok egyenes szakasza mentén stabil maradni, de például a gerincoszlop, a csigolyák mozgását is annak anatómiai viselkedése szerint lekövetni.

Ezzel a vázszerkezettel lehetséges valós idejű vezeték nélküli mozgáskövetés és relatív elérhető árban van egy kisebb stúdió számára is.

h. Mágneses rendszer

A mágnesességen alapuló MoCap rendszerek úgy az adó- (transmitter), mint a vevő (reciever) készülékeken található, egységenként három, térben egymásra merőlegesen elhelyezett (ortogonális) réztekercs relatív mágneses fluxusának kiszámítását használja fel a pozíció és az irány meghatározásához.

Az inerciarendszer alapú megvalósításhoz hasonlóan itt is egy hat szabadságfokkal rendelkező, valós idejű mozgáskövetésre alkalmas rendszerről beszélünk.

Ennek a rendszernek az egyik legnagyobb hátránya a korábbiakban bemutatottakkal szemben, hogy míg azoknál a környezeti tényezők, mint interferencia és egyéb behatások, nem, vagy csak nagyon kis mértékben befolyásolhatják a rögzített mozgás pontosságát. Addig itt már akár az épület vasbeton szerkezete, illetve egyéb erős mágneses jelleggel bíró, rendszer közelében található elemek is torzíthatják az eredményt.

2.2 Notch Pioneer eszköz [8]

Az általam használt mozgáskövető termék a Notch Pioneer nevet viseli, ami megvásárolható a hivatalos honlapon (www.wearnotch.com) 379 amerikai dollárért (USD), ami jelen árfolyamot nézve 115.000 forint (HUF) körül mozog. A pénzünkért cserébe megkapjuk a fizikai rendszert ami a Notch Pioneer Kit névre hallgat. Ez a készlet tartalmazza a szenzorokat, ezek testre rögzítését segítő kényelmes gumipántokat és egy töltőállomást. Továbbá kapunk egy licenszelt felhasználói fiókot a Play Store-ból (iPhone esetén App store) letölthető Notch Pioneer alkalmazáshoz.

A terméket forgalmazó és fejlesztő cég pedig a Notch Interfaces Inc. Egy amerikai startup, akiknek jelenleg néhány aktív munkatársból álló irodájuk Budapesten is megtalálható. A cég remek sikereket ért el a 2017-es CES-en ezzel a termékkel. Továbbá több közismert céggel, mint a RedBull, EIT Health, NYU, Vodafone és társaik is partnerségi kapcsolatban állnak. Úgyszintén támogatnak és közreműködnek több egyetemi projektben is szerte a világon. Így volt szerencsém nekem is hozzájutni, a cég jóvoltából, a BME Automatizálási és Alkalmazott Informatikai Tanszék számára, fejlesztési célokból felajánlott készülékhez és fejlesztői licenszhez.

A Notch Pioneer mozgáskövető rendszere a korábban részletesebben bemutatott nem optikai rendszerek közül az inerciális technológiát használja. Az IMU-k vagyis a szenzorok maguk az esztétikus kinézetű hatszög alakú Notchok, amikből egy készletben hat darab található, az alkalmazáson keresztül pedig egyszerre tizennyolc darabot tudunk párosítani, hogy a mozgást a lehető legpontosabb tudjuk rögzíteni.

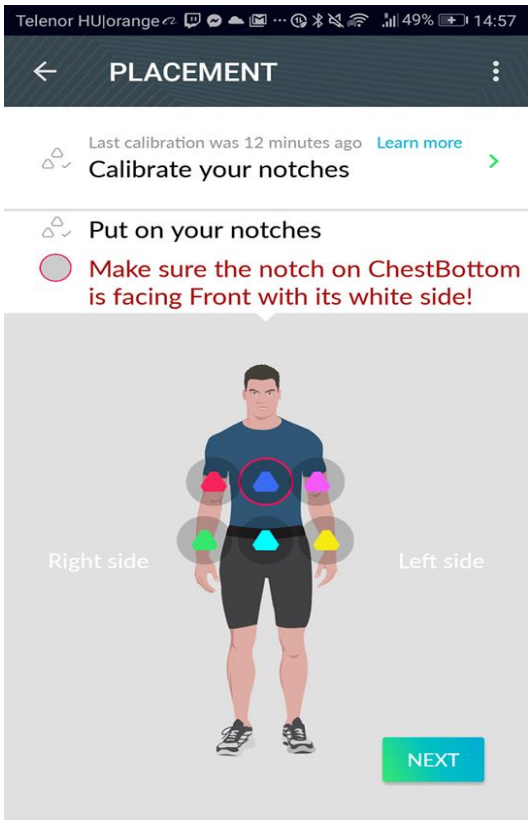
Habár a korábbiakban említett ár első ránézésre borsósnak tűnhet és nem mindenki engedheti meg magának, még mindig az egyik legelérhetőbb árkategóriába tartozó MoCap rendszerről beszélünk. Ez az árazás eltörpül a több százezres optikai rendszerek ára mellett, de még a néhány-ezer -tízezer dollárba kerülő egyéb nem optikai rendszerek költségéhez képest is messzemenőleg a legpénztárcabarátabb.

A cég és az eszköz célja, hogy ne csak indusztriális és fejlesztői környezetbe használják és vásárolják ezeket az eszközöket. Szeretnék, hogy a hétköznapi emberek, mint magánszemélyek is hozzá tudjanak jutni és megvásároljanak egy ilyen készüléket, ami egy megfelelő alkalmazás segítségével, segítheti őket a sport-teljesítményük fejlesztésében, új mozgásformák elsajátításában és tökéletesítésében. Vagy akár

virtuális- és kiterjesztett valóság alapú videójátékokon keresztül szórakoztató élményt nyújtson számukra.

2.3 Notch Pioneer felhasználói App [10]

A Notch Pioneer Kit valóban egy nagyon könnyen hordozható, kényelmesen és egyszerűen használható készülék. Az alkalmazáson keresztül feltűntetett instrukciók és ábrák segítségével pofonegyszerű és egyértelmű az eszköz beüzemelése. Az ábrák és útmutatások megmutatják, hogy melyik testrészre, azon belül is hova szükséges elhelyezni az adott színnel világító Notchot.



2.1. ábra: Notch Pioneer App Placement nézete, ami szemlélteti, hogy mely testrészekre helyezzük fel a különböző színnel világító ledeket. Külön felhívva a figyelmet arra, hogy a mellkasra elhelyezett Notch megfelelő pozícióba a fehér oldalával előrefele legyen felhelyezve.

Első alkalommal egy kicsit több időt vesz igénybe a beüzemelés, ugyanis ekkor egyesével párosítani szükséges az összes szenzort bluetooth-on keresztül a mobil készülékünkre telepített alkalmazáshoz. Ezek után ajánlott első alkalommal egy firmware update-t futtatni, majd kalibrálni a szenzorokat. A kalibrálás nem szükséges minden alkalommal, de ajánlott ha megszakítjuk a telefon és az eszköz közötti összeköttetést, más vezeték nélküli készüléket csatlakoztatunk a telefonhoz, vagy ha egy

új környezetbe, más frekvenciális viszonyok, külső behatások, interferenciák között kívánjuk használni a rendszert.

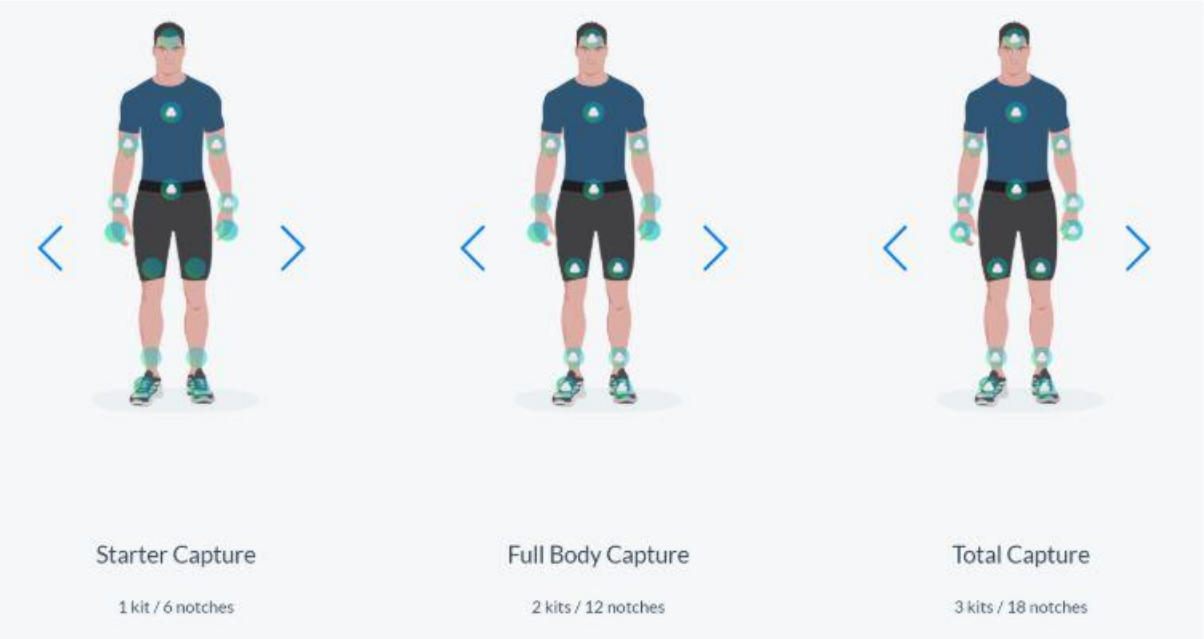
Amit viszont minden felvétel készítés előtt elengedhetetlen megtenni az a Steady (vigyázzállás) pozíció felvétele, hogy a felvétel során a szoftver pontosan tudja a szenzorok kiinduló pozíciójától eltérő mozgást, a különböző testrészek egymáshoz viszonyított elmozdulást kiszámolni és akár valós időben lemodellezni.

Az alkalmazás akár egy, de ahogy említettem, maximum tizennyolc IMU-val használható. Minden esetben egy szenzornak kötelező módon a mellkasra (chest) kell kerülnie, az a master bone. Ezt kék színben felizzó led jelzi a Notch-on. Az alkalmazásban több, különböző számú szenzort igénylő, előre konfigurált állapot is található, mint például:

- 1 Notch: mellkas
- 3 Notch: jobb alkar, jobb felkar és mellkas
- 6 Notch - felsőtest: jobb és bal alkar illetve felkar, mellkas és csípő
- 6 Notch - alsótest: jobb és bal boka illetve comb, mellkas és csípő
- 11 Notch - teljes test és fej: jobb és bal boka, comb, alkar és felkar, mellkas és fej

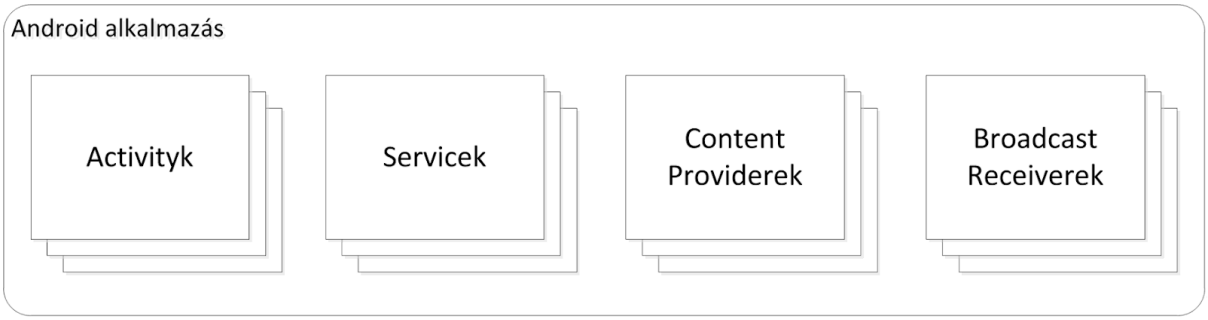
Az alkalmazásom fejlesztése folyamán leggyakrabban az egy Notch-os beállítást, majd a fejlesztés előrehaladtával esetenként a három Notch-os (mellkas és kar) beállítást használtam. Ezt a fejlesztői és tesztelői döntést a megfelelő fejezetben részletezni fogom.

A standard 6-12-18 szenzorhoz tartozó beállítási lehetőséget pedig nem csak az alkalmazásban de a honlapon is megtekinthetjük egy szemléletes ábrán (2.2. ábra). Amiből az is kiderül, hogy egy készlettel akár 90 különböző beállítási lehetőségünk is lehet. Míg két készlettel ez már 120-ra, hárommal pedig nem kevesebb mint 180-ra bővíthetjük a lehetőségek tárházát.



2.2. ábra: Különböző szenzor elhelyezés variációk szemléltetése egy, kettő és három készlet használat esetén. [8]

2.4 Kitekintés: Android alkalmazások felépítése [11][12][13]



2.3. ábra: Android alkalmazások felépítése. Mobil- és webes szoftverek tantárgy hallgatói jegyzetek [11].

Egy Android alkalmazás egy vagy több alkalmazáskomponensből épül fel:

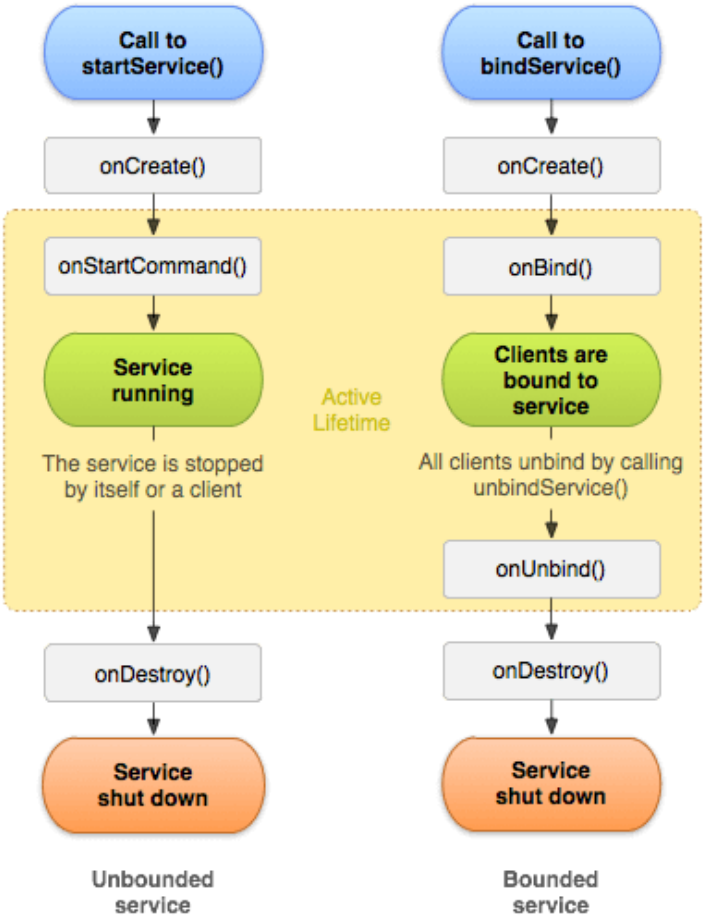
- **Activityk (a program egy "ablaka")**

Android platformon a taskok neve az Activity. Az android.app.Activity osztályból származnak le. Minden alkalmazásnak kell legyen egy belépési pontja, amely a felhasználónak először megmutatott Activity lesz. Az Androidos alkalmazások általában több tevékenységből állnak. Tipikusan egy Activity-hez tartozik egy képernyő is. Egyszerre mindig csak egy képernyő látható, de egy alkalmazáshoz több képernyőkép tartozhat, amelyeket futás közben szabadon cserélhetünk. Az Activity-k életciklusát jól ábrázolja a 2.5. ábra.

- **Service-ek (háttérben futó szolgáltatás)**

A Service egy olyan komponens az Android alkalmazásokban, ami képes hosszú ideig tartó műveletek végrehajtására a háttérben és nem rendelkezik felhasználói felülettel. Az alkalmazás egy komponense, például egy Activity elindíthat egy Service-t, ami nem csak az alkalmazáson belüli Activity váltások során marad továbbra is aktív a háttérben, kivéve ha onDestroy() állapotba kényszerítik. Hanem abban az esetben is futó állapotban marad ha a jelenlegi alkalmazásból átváltunk vagy megnyitunk egy másik alkalmazást.

Egy Service képes internetes tranzakciók, adatforgalom kezelésére, zenelejátszásra, fájlok írására és olvasására mindezt a háttérből vezérelve, mint egy láthatatlan komponens a felhasználó számára.



2.4. ábra: Különböző Android Service típusok életciklus (lifecycle) modelljei. [14]

Amint a 2.4. ábra is mutatja, alapvetően a Service-eknek két nagy csoportját különböztetjük meg egymástól:

1. **Futó szolgáltatás (Started Service)**

Ezek olyan szolgáltatások, amelyek akkor indulnak el, ha egy komponens meghívja a startService() metódust. Ekkor a szolgáltatás futó állapotba kerül. Majd egy későbbi stopService() hívás hatására pedig leáll. A szolgáltatások ezen csoportjának két fajtáját különböztetjük meg:

a. **Előtérben futó szolgáltatás (Foreground Service)**

Ezek a szolgáltatások általában valami olyan műveletet végeznek, amit a felhasználó is érzékel. Például egy zenelejátszó esetében, miközben a telefon más alkalmazásait nyomkodja a felhasználó, folyamatosan hallgatni tudja az elindított zenét, anélkül, hogy az megállna amikor egy másik applikációt megnyit. De egy hasonló szemléletes példa, amikor egy alkalmazásban beépített térképet használunk, ami ráadásul felhasználja a telefon szerinti tartózkodási helyet.

b. **Háttérben futó szolgáltatás (Background Service)**

Itt olyan műveletekről beszélünk, amit a felhasználó közvetlenül nem érzékel, de fontos szerepet tölt be az alkalmazás működésében. Például egy olyan szolgáltatás ami a háttérben gondoskodik arról, hogy az adott app megfelelően használja a rendelkezésre álló memóriát, tárhelyet és optimalizálja a processzor kihasználtságot.

2. **Összeköttetés alapú- vagy kötött szolgáltatás (Bound Service)**

Talán a legsokrétűbbnek ez a fajta Service mondható. Egy Service akkor van összekötve az alkalmazással, amikor az meghívja a megfelelő bindService() metódust. Ezáltal lényegében egy kliens-szerver minta szerinti összeköttetés jön létre, ahol a szolgáltatás áll a szerver szerepében, az alkalmazás pedig a kliens, aki kérésekkel fordul a szolgáltatáshoz és használja fel az attól érkező válaszokat. Az alkalmazások életében talán leggyakoribbnak ez a típusú Service-használat mondható. Ebben az esetben az adott szolgáltatás addig fut amíg még van hozzá csatlakozott alkalmazás. Több komponens is lehet egyidőben a szolgáltatásra csatlakozva, de amint mind leszakadnak, a szolgáltatás is leáll és megsemmisül.

A Service-k nagy előnye, hogy nem feltétlenül szükséges a működésükhöz internet kapcsolat. Vannak olyan szolgáltatások, amelyek kizárólag a telefon különböző komponensei közötti kommunikációért, vagy mint esetemben is egy külső

eszközzel való kommunikációért felelősek. Valamint úgyszintén képesek kétirányú, full duplex összeköttetés megvalósítására.

- **Content Provider-ek (adatok kezelése)**

A Content Provider (tartalom szolgáltató) komponens feladata egy megosztott adatforrás kezelése. A tartalom szolgáltató komponensen keresztül más alkalmazások hozzáférhetnek az adatokhoz, vagy akár módosíthatják azokat. Az adat tárolódhat fájlrendszerben, SQLite adatbázisban, weben vagy egyéb perzisztens adattárban, amihez az alkalmazás hozzáfér. A android.content.ContentProvider osztályból származik le és kötelezően felül kell definiálni a szükséges API hívásokat.

- **Broadcast Reciever-ek (rendszerszintű események kezelése)**

A Broadcast Reciever a rendszer szintű eseményekre reagáló komponens. Nem rendelkezik saját UI-al, inkább valamilyen figyelmeztetést írnak ki például a státusz kijelzőre, vagy elindítanak egy másik komponenst. Az alkalmazás is indíthat saját "broadcast"-ot, például ha jelezni akarja, hogy valamilyen művelettel végzett vagy kikapcsolt a képernyő, vagy esetleg, hogy az akkumulátor töltöttsége alacsony.

Az *android.content.BroadcastReciever* osztályból származik le, az esemény egy Intent formájában érhető el.

Minden komponensnek különböző szerepe van az alkalmazáson belül. Bármelyik komponens önállóan aktiválódhat. Akár egy másik alkalmazás is aktiválhatja az egyes komponenseket.

Egy Android alkalmazás felépítésének szempontjából még fontos megemlíteni a következő elemeket:

- **Context**

Ezen keresztül lehet elérni rendszer erőforrásokat. Lényegében az App minden része hivatkozhat rá, elérheti.

- **Fragmentek**

A Fragmentek a képernyő egy nagyobb részéért felelős objektumok. Ezek segítségével modulárisabb, rugalmasabb architektúra építhető. Egy Activityhez több, akár egymástól független Fragment is tartozhat. Egy Fragment mindig egy Activityhez

csatoltan jelenik meg. Életciklusuk nagyrészt megegyezik, mint azt az alábbi ábra is mutatja:

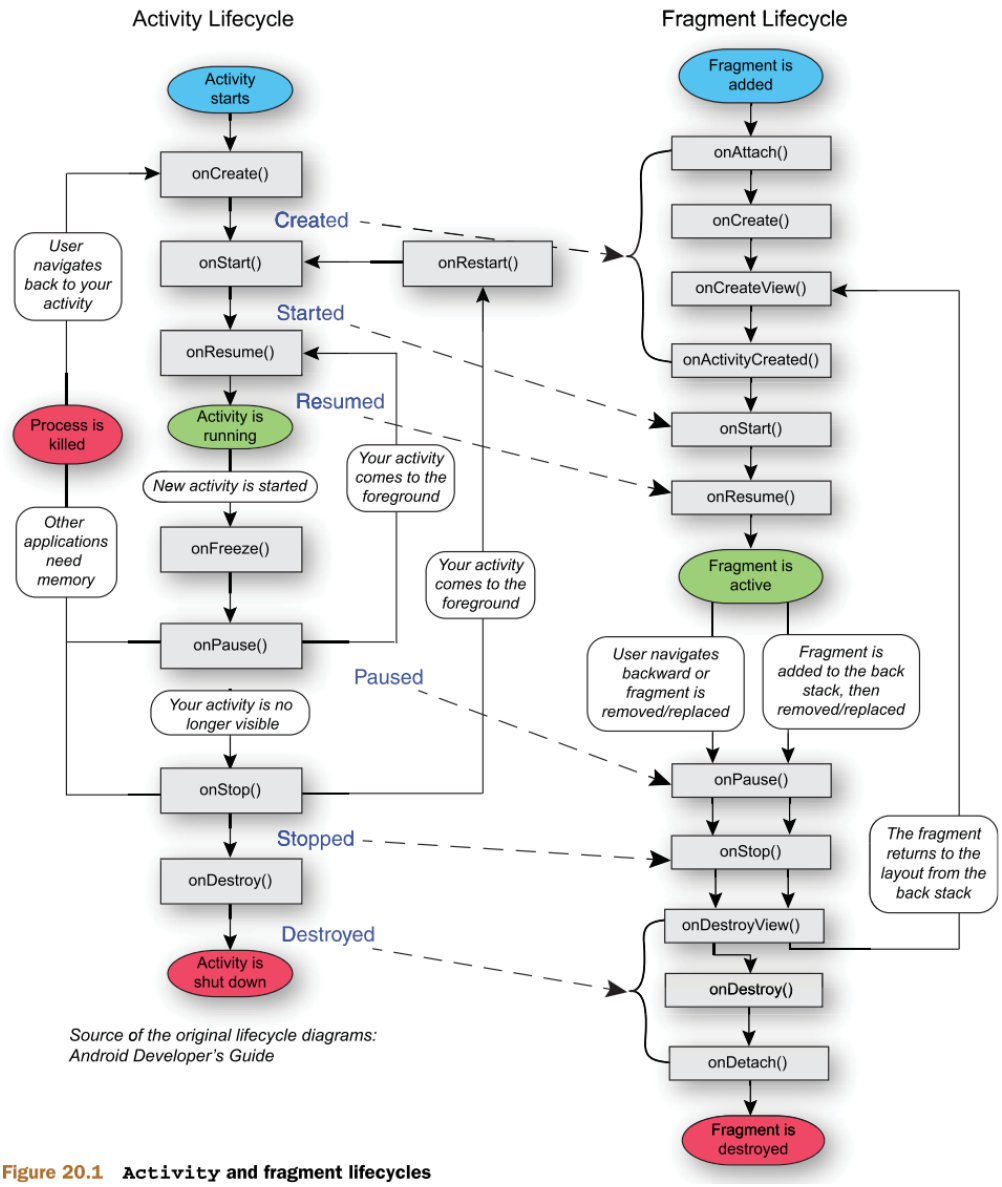
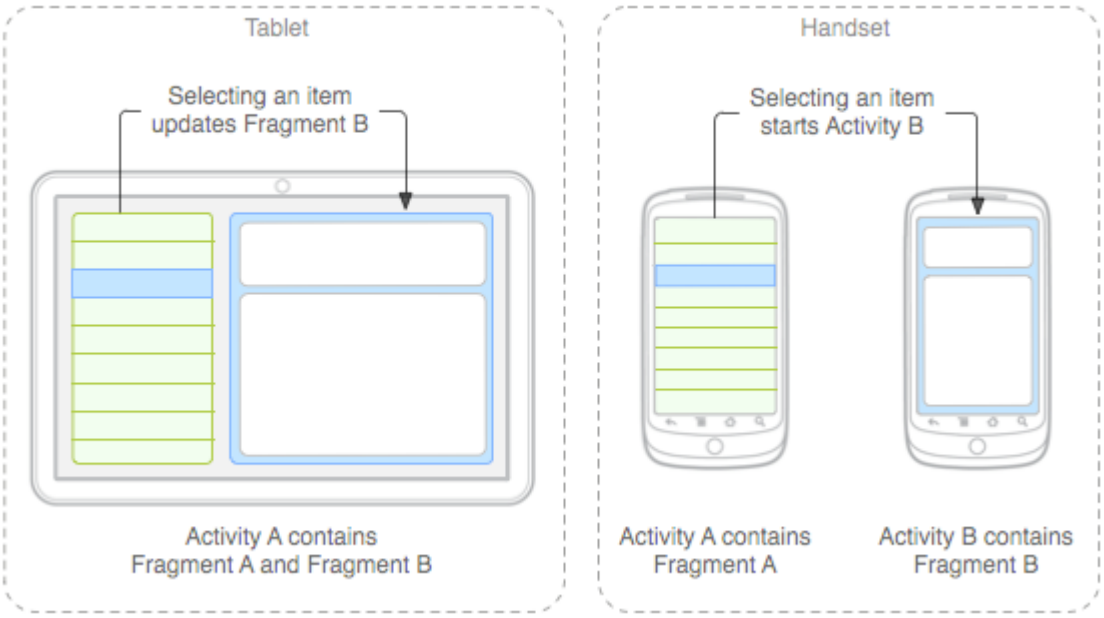


Figure 20.1 `Activity` and fragment lifecycles

2.5. ábra: Az Android Activityk és Fragmentek teljes életciklus modelljei. Készítette: Steve Pomeroy. [16]

Viszont ha egy Activityt leállítunk, akkor azzal leállítjuk az összes hozzá csatolt Fragmentet is. Ha egy Fragmentet bezárunk, attól nem záródik be az Activity is. A Fragmentek első sorban a felhasználói élmény javításának érdekében jöttek létre. A 2.6. ábra jól szemlélteti a Fragmentek gyakorlati hasznát:



2.6. ábra: Android Activityk és Fragmentek közötti összefüggések. Mobil- és Webes szoftverek tárgy 04.-es hallgatói jegyzetéből. [11]

- **Intent (szándék, kérés)**

Az Intent az eseményvezérelt fejlesztésben megszokott eseményhez hasonló objektum. Android komponensek közötti aszinkron üzenetküldő mechanizmus. Két típusát különböztetjük meg: az Explicit Intent-et, ami esetén konkrétan megnevezzük a cél komponenst, illetve az Implicit Intent-et, ahol a végrehajtandó feladatot írjuk le úgy, hogy a rendszer abból egyértelműen meg tudja határozni az arra megfelelő komponones hivást.

2.5 Fejlesztői Demo App

Tehát mostanra már „mezei felhasználóként” sikerült megismerni a rendszer és a Notch Pioneer app működését. Ideje viharosabb vizekre evezni és megismerkedni a fejlesztőknek szánt Demo App-al és az alkalmazás által is felhasznált fejlesztőkészlettel, a Notch Android SDK-val.

A Demo alkalmazást a honlapra való belépés után a *Developer* menüponton érhetjük el, ahol az Android SDK Overview-hoz navigálva [9], először egy telepítési útmutató található. Amit egy átfogó, részletes leírás, dokumentáció követ.

Mivel az oldalon található útmutatóban leírt lépések többnyire egyszerűen értelmezhetők és könnyen követhetők, de helyenként úgy éreztem, hogy kicsit

hiányosak. Ezért az Önálló laboratórium tárgy keretein belül készített dolgozatom GitHub Wiki oldalának [2] „Android SDK és Demo App (Template) megismerése” nevű (3.) fejezetében röviden összefoglaltam lépésről-lépésre, útmutatásként más magyar nyelvű fejlesztő társaim számára a Demo App letöltéséhez és beüzemeléséhez szükséges teendőket.

Nagyjából a Demo alkalmazás is azokat a funkciókat látja el mint a korábbi fejezetben bemutatott felhasználói alkalmazás csak egy jóval egyszerűbb kezelőfelülettel, hogy a fejlesztést gyorsabbá és átláthatóbbá tegye.

A Demo alkalmazás első futtatásakor ismételten párosítanunk majd kalibrálnunk kell a Notch-okat. Annak ellenére, hogy ezt már a rendes applikációban megtettük. Ennek a lépéseit viszont nem részletezném mivel a honlapon a *Developers → Android SDK → Android SDK Overview* menüpont alatt követhetően és érthetően dokumentálva van. [9]

A párosítást csak egyszer kell elvégezni mindaddig amíg ezt a rendszer nem igényli újra vagy ha valamiért “eldobja” a korábban párosított eszközöket.

Viszont a *Calibration → Unchecked Init* minden futtatás alkalmával szükséges, hogy inicializálja az épp bekapcsolt állapotban lévő Notch-okat. A *Calibration → Configure Calibration, Start Calibration, Get Calibration Data* funkciókon viszont többnyire csak akkor szükségszerű újra végigmenni, ha más környezetben, más helységben foglalkozunk ismét az eszközzel.

Ugyanígy a *Steady* alatt található funkciókat se szükséges minden egyes futtatás alkalmával újra végignyomkodni. Elég csak akkor, ha új szenzorokat kapcsolunk be vagy néhányat kikapcsolunk és ezáltal a steady pozíció beállításai is változnak, illetve ha helyszínt vagy pozíciót váltunk.

Tehát ha ugyanazokat a szenzorokat, vagy ugyanazt az egy szenzort használjuk fejlesztés közben az adott napon, akkor futtatásonként az első alkalom után elég mindig csak a *Calibration → Unchecked Init* illetve a *Capture → 3 Notch Init, Configure Real-Time, Start Real-Time* funkciókat végigjárni. Abban az esetben ha valós időben szeretnénk a mozgást követni, amihez szükséges még a *Real-Time* checkbox bepipálása. Ezek hatására ha mindent jól csináltunk akkor el is indul a valós idejű motion tracking a Demo app-ban, ami úgyszintén egy a Notch Pioneer alkalmazásnál egyszerűbb UI-on történik meg.

Személyes tapasztalatom a fejlesztés és tesztelés folyamán, főképp mikor már a Unityben készülő játékban lévő karakter mozgását teszteltem, hogy érdemesebb inkább minden alkalommal kivétel nélkül a Steady pozíció felvétele is, hogy a lehető legpontosabb kimenetet kapjuk.

2.5.1 Notch Android SDK és Service

A Software Development Kit (SDK), magyarul szoftverfejlesztő készlet egy adott program vagy platform szolgáltatásainak és funkcióinak használatát, az azon futó programok készítését teszi lehetővé, a fejlesztők számára. [17]

Az SDK-ban találhatóak olyan osztályok és metódusok, amik segítségével a telefonunk a szenzorokkal folyamatos kapcsolatot tud tartani, nyílvántartani azok státuszát és feldolgozni a beérkező adatokat. Majd az adatok feldolgozása után az SDK osztályai és metódusai segítenek a rögzített- vagy a valós idejű mozgás vizualizálására.

Az SDK legtöbb függvénye egy aszinkron (*async*) *NotchCallback* visszahívással tér vissza, ami egy NotchService nevű interfészhez tartozik. Ez az interfész pedig egy olyan szolgáltatást reprezentál, ami a korábban leírtakhoz hasonlóan felelős az IMU-kkal való folyamatos kommunikációért és ezekkel kapcsolatos műveletek elvégzéséért. Tulajdonképpen egy, a 2.4. fejezetben, részletesen bemutatott Android Service-ről beszélünk. Tetszés szerint az alkalmazás bármely komponense csatlakozhat a szolgáltatáshoz. Ezáltal transzparensen hozzáférhetünk a szenzorok aktuális állapotához és paramétereihez.

Ehhez a szolgáltatáshoz az alkalmazás alap komponenséhez (*BaseActivity*) tartozó *BaseFragment* a *bindNotchService()* metóduson keresztül csatlakozik. Majd a *NotchService*-hez hozzákapcsolódik a *MainActivity* is, így biztosítva a különböző komponensek Notch-okra gyakorolt hatása közötti transzparenciát és következetességet.

2.6 IMU-k adatfeldolgozása SDK-val

Tudtam, hogy ha valós idejű kommunikációt szeretnék megvalósítani a IMU-k és a Unityból Android Studióba importált alkalmazásom között, akkor tudnom kell, hogy milyen formában adják át az általuk rögzített adatokat a jelenlegi Demo app-nak. Vagyis inkább azt volt szükséges kiderítenem, hogy az átvett adathalmazt az app hogyan és milyen formában dolgozza fel az SDK segítségével és milyen kimenet születik ebből.

Mint azt már tudjuk a Notch-okat bluetooth-on keresztül tudjuk csatlakoztatni a telefonunkhoz. Viszont érdekességként megemlíteném, hogy mindegyik Notch egyenként rendelkezik egy kis memóriával is, amin el tudja tárolni a rögzített mozgásból származó adatokat. Amit lementhetünk és később feldolgozhatunk, vagy ha már nincs szükség ezekre az adatokra, vagy az IMU memóriája elkezd megtelni, akkor az alkalmazáson keresztül az SDK segítségével ezt egy törlési (erase) metódussal kiüríthetjük.

Az IMU-król már tudjuk, hogy ezek olyan mérő egységek amelyekben található giroszkóp, magnetométer és gyorsulásmérő. Egy Notch a benne lévő IMU segítségével rögzíti ezeket az adatokat és beállítástól függően elmenti vagy továbbítja a szoftver számára feldolgozásra.

A szoftver nyers adatokat kap, amik nem elegendőek arra, hogy meghatározzuk az adott szenzorok elmozdulását és virtuálisan modellezzük a mozgást. A mozgás modellezéséhez szükséges a fejlesztőkészlet nyújtotta osztályok és metódusok rendeltetésszerű használata. Erre nagyon jó példát ad a Demo alkalmazás *calculateAngles()* függvénye, ahol az SDK-ban található *VisualiserData* osztály *calculateRelativeAngle()* függvényének segítségével a nyers adatot megfelelő matematikai számításokkal feldolgozza. A kimenet pedig egy olyan háromdimenziós vektor lesz (vec3) ami leírja a szenzor X, Y és Z tengely szerinti elmozdulásának mértékét. Ezt a vektor értéket felhasználva már ábrázolni tudjuk a mozgást az alkalmazásban található modellen.

(A dolgozatomnak nem célja a mért adatok háttérben történő feldolgozásához használt nem triviális matematikai műveletek bemutatása és elemzése.)

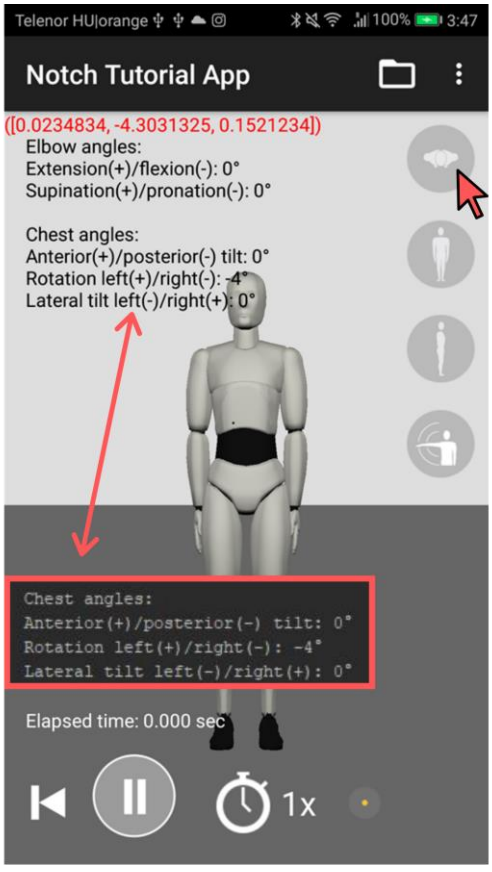
Annak érdekében, hogy megértsem a Demo App működését, logikáját valamint a Notch Android SDK osztály-hierarchiáját és a különböző osztályok és metódusok funkcióját elkezdtem használni a Android Studióba beépített debuggolást elősegítő LogCat-et. Vagyis Log üzeneteket helyeztem el a kódban bizonyos helyekre, hogy figyelni tudjam az alkalmazás működését. ezeknek a log üzeneteknek adtam egy "*NotchPosition*" tag-et, hogy egyszerűen szűrhetőek legyenek a LogCat ablakban:

```
Private static final String LOGTAG = "NotchPosition";
```

Eleinte csak ismerkedtem ezzel a módszerrel, beépítettem egy-két funkcióra, hogy követni tudjam a függvényhívásokat. Például a *getSteadyData()* függvényben az alábbi kódot helyeztem el ennek érdekében:

```
Log.i(LOGTAG, „getSteadyData”);
```

Mivel inerciarendszerről és háromdimenzióban való elmozdulásról beszélünk tudtam, hogy valamilyen fvec3 formájú kimenetel szükséges amit már lehet modell szinten vizualizálásra használni. Így találtam rá az említett *VisualiserActivity* osztály *calculateAngles()* függvényére. Ebbe a függvénybe beleírtam egy Log üzenetet aminek segítségével sikerült beigazoljam a korábbi feltételezésem, hogy a beérkező adathalmazt feldolgozó pipeline végére háromdimenziós lebegőpontos vektorokat (Fvec3) kapunk kimenetnek. Ezzel pedig a valós idejű kommunikációt is tesztelni tudtam a LogCat ablak és a futó alkalmazás futtatásánál kapott eredmények egyidejű összehasonlításával, amit a 2.7. ábra is mutat.



2.7. ábra: A Demo App ‘Start Real-Time’ Capture utáni nézete, amit a VisualiserActivity Android osztály segítségével nyitunk meg. A pirossal bekeretezett képernyőkép kivágáson pedig egyazon időpollanatban készített állapota látható a Log üzeneteket megjelenítő LogCat ablaknak.

3 Unity Project és Android Studio közötti kapcsolatépítés

Ahhoz, hogy a végső alkalmazásom megfelelően működjön, szükségem lesz a Unityben megtervezett és elkészülő játékom valamint az Android Studióban átírt, Demo applikáció között valós idejű kommunikáció felépítésére.

A Szakdolgozatomban nagyobb hangsúlyt főképp a Unityben Androidos eszközökre készülő játékfejlesztésre fektettem. Azt már kitapasztaltam, hogy a Notch Pioneer alkalmazáshoz hasonlóan a fejlesztőknek szánt Demo alkalmazással is kimeríthető a mozgáskövető rendszer minden funkciója. Ezért úgy döntöttem, hogy ezt az alkalmazást mint egy megbízhatóan működő sablont, felhasználom kiindulási alapnak a szenzorokkal való kommunikációhoz és valamilyen módon összekötöm a Unity projektemmel.

Virtuális Valóság alapú környezetben létfontosságú, hogy a késleltetés (latency) minimális legyen és a játékos ne észleljen jelentős eltérést a testrészek pozíciójának megváltoztatása és a játékbeli karakter Notch-októl kapott adatok alapján vizualizált elmozdulása között.

3.1 Lehetőségek valós idejű kommunikáció felépítésére

Pár napnyi böngészés, kutatás, ezen a területen jártas ismerősökkel való konzultáció, elemzések- és programozói cikkek olvasása, tutorial és know-how videók nézegetése után arra jutottam, hogy ha ezt a kommunikációt valós időben szeretném kezelni, ami márpedig ez esetben elengedhetetlen, akkor arra az alábbi megoldások biztosítanak járható utat:

3.1.1 WebSocket [18][19]

A WebSocket olyan internetes technológia, amely lehetővé teszi folyamatos valós idejű kétirányú, duplex kapcsolatfolyamok létesítését a kliens és a szerver között egyetlen TCP protokollon keresztül. Kifejlesztésének fő motivációja volt, hogy a webböngészőben futó alkalmazás képes legyen a szerverrel való kétirányú kommunikációra.

Bár a fő cél a webszervereken és a webböngészőkben való implementáció, más kliensszerver megoldásokban is használható ez a valós idejű kommunikációs megoldás.

A szerver-kliens kapcsolat a Kommunikációs hálózatok tantárgy keretein belül tanult úgynevezett kézfogási (handshake) mechanikával működik. A WebSocket is ezt felhasználva építi ki a kétirányú valós idejű kapcsolatot, amihez először a kliens egy kézfogási kérelmet küld (handshake request), amire a szerver kézfogási válasszal (handshake response) válaszol. A kézfogás a HTTP-re hasonlít, de valójában annyi történik, hogy a kézfogási kérelmet a szerver a HTTP protokoll szerint értelmezheti, majd átvált a WebSocket használatára.

Nekem arra volt szükségem, hogy a Unity alkalmazás valós időben hozzáférhessen és felhasználhassa a Notch-októl érkező és már feldolgozott adathalmazt. Esetenként akár vissza is tudjon szólni, hogy most már nincs szüksége az adatokra, mert véget ért a játék. Esetleg szüneteltetve legyen a kommunikáció mert a játékos háttérbe helyezte vagy csak megállította kis időre az alkalmazást. Ezek alapján arra tudtam asszociálni, hogy itt nincs szó másról mint egy a korábbiakban leírt kliens-szerver kapcsolatról, ahol a kliens szerepében a Unityben készülő alkalmazásom kerülne. A kiszolgáló fél, vagyis a szerver pedig az Androidos Demo app.

Viszont a WebSocketes megközelítésnek van egy egyértelmű hátránya, mégpedig az, hogy ehhez a kommunikációhoz rendszeres internetkapcsolat szükségeltetik. Ez annak ellenére, hogy manapság a lefedettség már nem jelent problémát a világ nagy részén és szinte mindenki 4G-s telefonokkal járkál az utcán, nem föltétlenül jó megoldás egy alapvetően egyéni játékmódra épülő alkalmazás esetében.

Van még néhány WebSocket-hez hasonló kommunikációs megoldás, mint például a Server-sent Events. Ez a megközelítés ideális olyan közegben, ahol egyirányú úgynevezett push technológia alapú (push notification vagy push data), kizárólag a szervertől a kliens irányába közölt adatkapcsolatra van szükség. Erre gyakori jó példa a HTTP Post metódusa. A legtöbb olyan esetben ahol nem szükséges a kliens valós idejű adatközlése a szerver számára, kényelmesen használható a Server-Sent Events is. Például online multiplayer játékokban vagy üzenet- de akár videó- vagy hang alapú társalgást megvalósító alkalmazásokban.

A Server-Sent Event-hez hasonlóan használható Long Polling is. Ami úgyszintén egyirányú, de előnye hogy minden környezetben működik, standardizált és jól kezeli, ha egyidőben több felhasználótól is érkezik kérés.

3.1. táblázat: A WebSocket, Server Sent Event és a Long Polling technológiák közötti összefüggések és különbségek. [18]

	WebSockets	Server Sent Events	Long Polling
Number of parallel connections from Browser	1024	~6 per server	~6 per server
Load Balancing and Proxying	Non-Standard / Complicated	Standard / Easy	Standard / Easy
Supported on all browsers	Yes (90%)	No (84% - not on IE and Edge)	Yes (100%)
Dropped Client Detection	Yes	No	No
Reconnection Handling	No	Yes	No

Egyre jobban beleásva magam a WebSocket-ről szóló irodalomba és látva a leggyakoribb felhasználási területeit, megvalósításokat. Egyre inkább azt realizáltam, hogy ezt a megközelítést használni az én problémám megoldására túlzás (overkill) lenne. Hiába, hogy nem túl bonyolult megvalósítani, és amire kitalálták tökéletesen működik, de már csak a korábban említett folyamatos internet hozzáférés is abba az irányba terelt, hogy kell legyen ennél egyszerűbb, szebb és számomra praktikusabb megoldás is.

3.1.2 Service

Az Androidos kontextusban értelmezett Service komponenst a 2.4-es fejezetben már részletesen bemutattam, valamint a 2.5.1-es fejezetből nagyvonalakban az is kiderült, hogy a Notch Android SDK részeként fellelhető Notch Service, hogyan működik és hogyan használja a Demo alkalmazás ezt a szolgáltatást.

Melyiket választottam és miért azt? – tevezői gondolatmenet és döntés bemutatása:

Az eddigi megközelítésem az volt, hogy nekem valós idejű kommunikációt szükséges létesítenem az Android app és a számítógépen készülő Unity projekt között úgy, hogy míg a telefonnal összepárosítom a Notchokat és a Unity Editor-ban futtatom és tesztelem a játékot ezeknek kommunikálniuk kell egymással. Ezért gondolkodtam eleinte a WebSocket alapú megoldáson. Viszont az igaz, hogy ha például Unity Editor-ban szeretném ténylegesen valós idejű adatok alapján tesztelni a játékot, miközben az Android Studióból futtatott alkalmazás kommunikál a szenzorokkal, akkor erre továbbra is ez a megoldás az egyik legmegfelelőbb.

Ahogy egyre jobban kezdtem megismerkedni a Unity nyújtotta lehetőségekkel illetve a Unityben történő telefonos alkalmazásfejlesztéssel, rájöttem, hogy ezt ki tudom kerülni. Unityből lehet projektet direkt Androidos készülékre fordítani (Build and Run) vagy exportálni olyan formában amit egy Androidos fejlesztőkörnyezet (Java alapú) kezelni tud.

Tehát ettől kezdve arra kerestem a konkrét megoldást, hogy hogyan lehet kommunikálni, valós idejű összeköttetést felépíteni az alapvetően Java alapú Demo alkalmazás és a Unity-ben C# nyelven íródó app között.

Az Unityből lefordított, Android Studióba importált projekt egy UnityPlayerActivity nevű osztályt tartalmaz ami a beépített Activity osztály leszármazottja így rendelkezik annak minden tulajdonságával és metódusával. Vagyis az alapul szolgáló Demo alkalmazásból meghívható a Unityben fejlesztett játék mint egy új Activity. Így már világossá vált számomra a “Mobil- és webes szoftverek” tárgy keretében tanultak alapján, hogy amik általában felelősek egy alkalmazás különböző komponensei közötti valós idejű kommunikációért Androidos környezetben, azok a Service-k.

3.2 Android – Unity összekötése

3.2.1 Plugin Library létrehozása

A Plugin Library létrehozásához először győződjünk meg arról, hogy Unityben az *External Tools* menüpontnál, hogy a szükséges Android SDK, JDK és NDK

csomagok és hívatkozások be vannak állítva. [20] Ezután a *Build Setting*-nél a platformot átállíthatjuk Androidra. Majd a *Player Settings*-nél fontos az alapértelmezett Package namet átírni valami másra ‘com.unity3d.<projekt_neve>’ formátumban. Tapasztalataim alapján annak érdekében, hogy az exportálás és esetleges buildelés pár perc alatt meglegyen és ne kelljen több mint fél órát rá várni, érdemes a *Scripting Backend*-et *IL2CPP*-re állítani és az *ARM64* checkbox-ot bepipálni, valamint a *Graphics APIs*-nál a *Vulkan*-t az *OpenGLES3* fölé helyezni. Végezetül exportálás előtt még a *Build Settings*-nél válasszuk a *Build System Gradle* opciót és a *Development Build*-et is engedélyezzük.

Ha megvagyunk a fenti beállításokkal akkor exportálhatjuk a projektet az Android projekt mappájába, ami következtében az Android Studio automatikusan importálja is a könyvtárat egy *plugin library*-ként.

Ezt követően már csak néhány apró módosítást szükséges megtenni a pluginhoz tartozó *build.gradle* és *AndroidManifest.xml* tartalmában, hogy orvosoljuk az esetlegesen fellépő konfliktusokat. Ehhez egy Stackoverflow felhasználó által nyújtott részletes leírást használtam. [21]

3.2.2 Kommunikáció üzenetküldéssel

Itt már alapul vehetjük azt, hogy van egy működő Androidos alkalmazásunk, a 2.5. fejezetben bemutatott Notch Pioneer Demo app. Most már létrehozhatunk egy Unity projektet, hogy tesztelhessük a kommunikációt egyelőre egy egyszerű szövegmezőbe való adatmegjelenítéssel. A korábbiakban is hívatkozott Önlaboratórium tárgy keretein belül készített GitHub Wiki dokumentációm [2] „Demo App működése és kommunikációja a beimportált Unity Projekttel” (6.) fejezetében részletesen leírom az ehhez szükséges lépéseket és a szükséges kódrészleteket is bemutatom, ami alapjául a Jinbom Heo által készített nyílt forráskódú Unity-Android-Communication projekt [22] és Anna Kravchuk által írt Android+Unity Integration cikk [23] szolgált.

Ezeket implementálva és az előző fejezetben leírt plugin library létrehozását követően az Androidos eszközömön sikeresen tudtam futtatni az alkalmazást ami a *myText* mezőbe kiírta a kívánt üzenetet, ami a szenzortól kapott adat volt.

Azonban ezen a ponton csak a *UnityPlayerActivity* meghívásának pillanatában kapott utolsó adatot jelenítette meg az alkalmazás, mivel alapból ez az Activity nem

kapcsolódik a háttérben futó Notch Service-hez, így valós idejű kommunikáció sem valósul meg.

3.2.2.1 Valós idejű megvalósítás

Az eredeti Demo alkalmazásban a főképernyőn vagyis a *MainFragment* osztályban, ami része a *MainActivity*-nek, a *Capture* menüpont alatti *Start Real-Time* gomb megnyomásával tudjuk elindítani a valós idejű mozgáskövetést. A gombnyomás hátására a háttérben meghívódik a *MainFragment.capture()* függvénye, ami a *NotchService.capture()* eljárást használva továbbhív a *VisualiserActivity*-be, az updateRealTime() metódus segítségével.

Ahhoz, hogy valós idejű kommunikációt érjünk el a Unity által generált Activity meghívásakor, fenn kell tartanunk a kapcsolatot a Notch Service és az éppen futó alkalmazás ablak között. Ehhez a BaseActivity és BaseFragment osztályban fellelhető eljárások kódjait: *addNotchServiceConnection(NotchServiceConnection), onDestroy(), onServiceDisconnected(ComponentName), onServiceDisconnected(), onServiceConnected(ComponentName, IBinder), onServiceConnected(NotchService), bindNotchService()* vettem alapul és ültettem át a VisualiserActivity osztályomba.

Majd ugyanide a *MainFragment.capture()* mintájára implementáltam egy *startUnityPlayerActivity()* és egy *updateRealTime()* függvényt, amik segítségével már a UnityPlayerActivity is hozzákötődik a NotchService-hez és létre tud jönni a folyamatos kommunikáció.

A startUnityPlayerActivity() metódus lényeges kódrészlete (kipontozott részen az *onSuccess(), onFailure()* és *onCancelled()* függvények felüldefiniálásai):

```
void startUnityPlayerActivity() {
     mUnityPlayerActivity = null;
     c = mNotchService.capture(new NotchCallback<Void>() {
          @Override
          public void onProgress(NotchProgress progress) {
               if                (progress.getState()                ==
NotchProgress.State.REALTIME_UPDATE) {
                    mData = (VisualiserData) progress.getObject();
                    refreshAngles();
                    Log.i(LOGTAG, RelativeNotchPosition);
                    updateRealTime();
               }
          }
          (. . .)
     });
}
```

Az *updateRealTime()* metódus lényeges kódrészlete (kipontozott részeken a bal és jobb fel- illetve alkarhoz tartozó adatok üzenetként való továbbítása, a mellkashoz tartozó Chest Angles üzenetekhez hasonlóan

```
private void updateRealTime() {
     if (mUnityPlayerActivity == null) {
          unityIntent = new Intent(this, UnityPlayerActivity.class);
          unityIntent.putExtra("message", RelativeNotchPosition);
          /** Chest Angles **/
          unityIntent.putExtra("message", RelativeNotchPosition);
          unityIntent.putExtra("ChestAnteriorTilt",
ChestAnteriorTiltAngle);
          unityIntent.putExtra("ChestRotation", ChestRotationAngle);
          unityIntent.putExtra("ChestLateralTilt", ChestLateralTiltAngle);
          (. . .)
          startActivity(unityIntent);
     }
     else {
          unityIntent.putExtra("message", RelativeNotchPosition);
          /** Chest Angles **/
          unityIntent.putExtra("ChestAnteriorTilt",
ChestAnteriorTiltAngle);
          unityIntent.putExtra("ChestRotation", ChestRotationAngle);
          unityIntent.putExtra("ChestLateralTilt", ChestLateralTiltAngle);
          (. . .)
     }
}
```

Tehát ezekből a függvényekből láthatjuk, ahogy a *VisualiserActivity*-ben kiszámolt és megfelelő elnevezésű változóknak értékül adott adatok egy *Intent.putExtra(String name, int value)* metódusának segítségével továbbítódnak a *UnityPlayerActivity* számára. Ahol a kórábban említett és hívatkozott módon implementáltam a *javaTestFunc(String)* illetve *javaGetCoord(String)* függvényeket amikben a *UnityPlayer.UnitySendMessage(String, String, String)* beépített eljárás használatával megtörténik az üzenet alapú kommunikáció. Az így kapott adatokat pedig már folyamatosan ki tudjuk írni a képernyőre, vagy ami még fontosabb, ezekkel már implementálni tudjuk a karakter mozgatásához szükséges metódusokat, amik működéséről többet a 4.6. fejezetből tudhatunk meg.

4 Játék készítés Unityben

Mostanra sikerült alaposan megismerni a mozgáskövető rendszert és a hozzá tartozó fejlesztőkészletben rejlő lehetőségeket és a Demo alkalmazás működését. Ezt követően pedig egy egyszerű szövegmegjelenítésre alkalmas Unity projektet felhasználva megoldani az Android - Unity kommunikációt. Először egyetlen üzenetként átvett adat megjelenítését sikerült megvalósítani. Majd a korábban bemutatott Service segítségével sikerült elérjem, a számomra jelentős áttörést nyújtó, valós idejű folyamatos adatátvitelt.

Számomra elképzelhető egy olyan nem túl távoli jövő, ahol a videójátékok túlnyomó része már virtuális valóság alapú lesz. Ahol a megfelelő ruhát, szemüveget és egyéb kiegészítőket viselve valóban úgy érezhetjük magunkat mintha teljes valójában mi irányítanánk a játékbeli karakterünket. Vagy ha sportolni támad kedvünk, például egy olyan focis játékot veszünk elő ahol az egyik játékos bőrébe tudunk bújni és úgy átélni egy teljes meccs élményét minden korlát nélkül.

Viszont ha kinyitom a szemem akkor látnom kell, hogy még korántsem tart itt a technológia. Valamint azt is be kell látni, hogy az általam elérhető technika erőforrásai is végesek. Mindössze egy átlagos Androidos telefon, egy belépő szintű MoCap készülék és egy otthon is összerakható Cardboard VR szemüveg áll a rendelkezésemre.

Olyan játékot szerettem volna tervezni ami kivitelezhető és élvezhető játékélményt nyújt csupán a felsorolt eszközök segítségével és egy rendelkezésre álló két négyzetméter alapú szabad területtel.

Egyelőre egy egyszerű, minimalista kinézetű de szórakoztató élményt nyújtó alkalmazást szerettem volna megvalósítani, ami tovább fejleszthető és kinőheti magát egy komolyabb projektnek, annak függvényében, hogy a végtermék mennyire lesz használható és milyen felhasználói visszajelzések érkeznek majd.

4.1 A Játék alapötlete és koncepciója

Az alap koncepciót végül az egyik korábbi TV műsor, a Kalandra fal motiválta. Ebben a műsorban a versenyző egy vízzel teli medence szélén áll, majd elindul vele szembe egy fal. A falon van egy speciális, hol egyszerűbb hol bonyolultabb testpozíciót

mintázó kivágás. A játékos ezt a pozíciót kell felvegye így el tudja kerülni, hogy a fal a vízbe taszítja. Ha nem ütközik a fallal, akkor teljesítette az adott próbát, viszont ha beleesik a medencébe, vagy összeütközik bármilyen formában a fallal, akkor veszít és nem kap pontot.

Tehát ebből inspirálódva az eredeti tervem az volt, hogy készítsek egy olyan játékot, amiben a játékos egy emberszerű karaktert irányít, pontosabban annak is a felső testét a korábban részletesen bemutatott Notch Pioneer MoCap eszköz segítségével. A kiindulási pozícióban a játékos egy medence szélén áll és a helyes testpozíció felvételével megpróbálja elkerülni, hogy a vele szembe sikló fal belelökje őt a medencébe.

Rég megterveztem a fenti koncepciót és már hetek óta fejlesztettem, mikor észrevettem, hogy tulajdonképpen létezik már ilyen WII-s és Xbox Kinect-es játék. Amik természetesen nem virtuális valóság alapúak, se nyomkövető rendszert nem használnak. Szimplán egy kamera rendszerrel működnek és azzal figyelik a mozgásunkat. Ez a kamera rendszer sokkal pontatlanabb mint egy akár egyéni használatra kapható IMU-kkal működő mozgáskövető rendszer. Ugyanis ezek olyan optikai mozgásérzékelő rendszerek, amikhez minimális számú érzékelőt használnak vagy a Kinect-hez például egyáltalán nem szükséges érzékelő. Ezekkel az észlelt mozgás átvetítése a játékra sem teljesen pontos így esetenként csökkenhet a játékélmény. Valamint a fejlesztők, csak úgy ahogy nekem is tennem kellett, olykor kompromisszumokat kell kössenek az élvezhetőség és a technika nyújtotta lehetőségek oltárán.

A célom viszont nem ezekkel a játékokkal való versenyzés és azok túlszárnyalása volt, hanem egy másabb játékélmény nyújtása első sorban a virtuális valóság implementálásával és a Notch eszköz használatával.

4.2 Játék specifikáció

Az alkalmazás lényege, hogy a játékos elkerülje, hogy a fal a medencébe lökje a karakterét.

A falak egy kötött pálya mentén a kiinduló pozícióból automatikusan haladnak a célpozíciójukba, ami során útjukba esik a játékos karaktere és átsiklanak a mögötte

található medence fölött is. Ha a játékos nem helyezi a karakterét a megfelelő pozícióba, akkor a fal belelöki azt a medencébe és a játékos életeinek száma lecsökken.

A játékos a karakterét a testére rögzített Notch mozgáskövető rendszerrel tudja irányítani first-person nézetből. A jelen verzióban a karakternek kizárólag a felső teste mozgatható. Lehetséges mozgások:

- a törzs forgatása, döntése, hajlítása;
- az alkarok feszítése/hajlítása, ki-/berotációja;
- a felkarok feszítése/hajlítása, ki-/berotációja, közelítése/távolítása.

Ha a fal ütközik a karakterrel, attól a ponttól a játékos elveszti a kontrollt a karakter felett, az pedig mint egy rongybaba összeesik vagy beledől a medencébe.

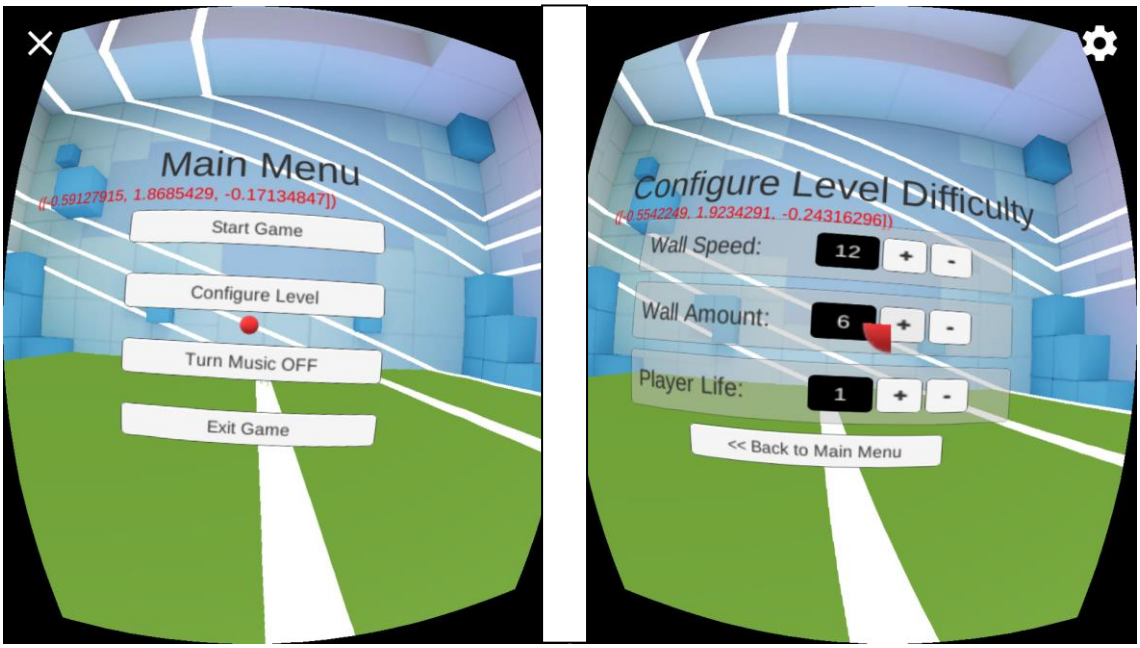
Többféle testpozíciónak megfelelő kivágású fal van jelenleg a játékba, amik véletlenszerűen váltják egymást, így a játékos nem tudja kitanulni az algoritmus működését és az alapján már előre felvenni a következő pozíciót.

Egyszerre egy fal van a pályán, aminek sebessége a játék kezdése előtt beállítható (alapértelmezetten 12 egység). Értelemszerűen minél gyorsabban közeledik a fal, annál nehezebb dolga van a játékosnak, mert annál gyorsabbnak kell lennie.

Ha a játékosnak nem sikerül elkerülnie, hogy a fallal ütközzön, akkor eggyel csökken az életeinek száma (alapértelmezetten 3). Ha az életek száma eléri a nullát akkor a játékos veszít, egy piros “GAME OVER” felirat jelenik meg és visszakerül a Főmenübe (Main Menu).

A játékos nyer, ha marad élete miután az utolsó fal is elért a céljába. Ekkor egy úgyszintén piros “YOU WIN” felirat jelenik meg és kis késleltetéssel akár új játékot is kezdhet más pálya nehézségi beállítások mellett, miután visszakerült a Főmenübe. A specifikációnak megfelelő teljes játékmenetet az 5.1. ábra szemlélteti és az előtte lévő bekezdés magyarázza.

Az alkalmazás aktuális verziójában jelenleg egy pálya van, végtelen nehézségi fokkal, amit a játékos tetszés szerint állíthat magának. A játék nehézsége teljesen testreszabható a *Configuration Level* menüpont alatt. Könnyíteni vagy nehezíteni lehet azáltal, hogy kezdés előtt beállítható a falak mozgásának sebessége, a játékbeli életek száma. Ahogyan az is konfigurálható, hogy összesen hány darab fal próbálja meg a medencébe taszítani a játékost.



4.1. ábra: Ezen az ábrán két virtuális valóság nézet hozzáadása (5.2. fejezet) utáni képernyőkép összevágása látható. Bal oldalon a Cardboard VR reszponzív menü látható. Jobb oldalon pedig a menü Configure Level kiválasztása után megjelenő úgyszintén VR reszponzív játék nehézség beállítására szolgáló UI fogad. A menü bármely elemére úgy tudunk kattintani, hogy arra megfelelő ideig irányítjuk a tekintetünket, amit a képernyő közepén lévő piros pötty egyértelműen jelez (A piros számsor a fejlesztői munka elősegítésére szolgál, amivel folyamatosan követni lehet az Android és Unity közötti valós idejű kommunikáció helyes működését.)

A felhasználó a szenzorokat továbbra is Notch Pioneer Demo alkalmazáson keresztül csatlakoztatja, konfigurálja. Valamint itt állítja be a játékhoz szükséges Steady pozíció alapján az IMU-k relatív kiindulási pozícióját.

A Unityben elkészült játék a Demo app Capture Real-Time nézetéből indítható el a jobb felső sarokban található gomb megnyomásával, amit a korábbi 2.7. ábra is szemléltet. Ekkor vált át az alkalmazás a virtuális valóságos nézetbe és indulhat a játék!

4.3 Unity alapfogalmak [24]

Az alábbiakban szeretnék bevezetni néhány alapfogalmat amikkel gyakran találkozhatunk a Unity keretrendszerében való fejlesztés folyamán. Sok esetben egyszerűbb és jobban érthető bizonyos fogalmak eredeti angol nyelvű változatának használata és jobb nem erőltetett fordításokkal leírni azokat. Ezért szeretném röviden definiálni és megmagyarázni jelentésüket, így a későbbiekben szabadon hivatkozhatok rájuk az eredeti nevükön anélkül, hogy ez nehezen érthetővé vagy teljesen értelmezhetetlenné tenné a szöveget, szövegkörnyezetet.

Az itt felsorolt és definiált fogalmakat a Unity hivatalos angol nyelvű dokumentációja alapján írtam át szabad fordításban magyar nyelvre, a teljesség igénye nélkül. Ezzel a célom többnyire az, hogy bárki olvassa a dolgozatot megértse ezeket a fogalmakat és könnyebben értelmezni tudja a dolgozat következő fejezeteit.

Alapfogalmak, eszközök, komponensek:

- **Unity Editor**

  A Unity Editor lényegében nem más mint maga a Unity keretrendszer és főképp az ehhez tartozó felhasználói felület. Ezen az interfészen keresztül tudjuk megtervezni és akár tesztelni, módosítani a játékunkat. Az adott pályához tartozó objektumokat, azok komponenseit, a komponensekhez tartozó paramétereket menedzselhetjük. A Unity Editorhoz egy UnityEditor nevű alaposztály is tartozik. Ezen keresztül minden funkciót ami az UI-on megtalálható, elérünk a szkriptekből is és így ezekre programozással is hatást tudunk gyakorolni.

- **Prefab**

  Unity környezetben a Prefab nem más mint olyan előre legyártott elem, ami támogatja az egyszerű újrahasználhatóságot és menedzselhetőséget, kikerülve azt, hogy sok objektumot kelljen előre definiálni a pályán (Scene). Ezáltal fölöslegesen használva a memóriát és növelve a teljesítményigényt.

  A Unity hivatalos definíciója a Prefabról: “A Unity Prefab rendszere lehetővé teszi GameObject-ek létrehozását, konfigurálását és tárolását, azok minden komponensével és tulajdonságaival illetve azokhoz rendelt értékekkel együtt, mint újrafelhasználható elemek. A Prefab továbbá eltárolja az alá rendelt összes többi úgynevezett gyerek (child) GameObject-et is. A prefab lényegében úgy működik mint egy sablon amit tetszés szerint inicializálhatunk a Scene-n.”

- **Scene**

  A Scene lényegében nem más mint a környezeti elemek és menük összessége. Egy Scene lényegében a játék egy pályájának (level) felel meg. Az Editor felhasználásával egész egyszerűen össze lehet rakni egy Scene-t és egy kis egyszerű játékot, anélkül, hogy akár egy sor kódot is írnunk kellene.

- **GameObject**

Mint a neve is mondja ez egy játék objektum. Unityben tulajdonképpen majdnem minden amit a Scenere felveszünk egy GameObject. Ezek mondhatni a játék építő kockái. Önmagukban nagy jelentőséggel nem bírnak, sokkal inkább tekinthetők úgy mint egy komponenseket tartalmazó konténer. Az objektumnak valódi értelmet pedig a hozzá rendelt komponensek adnak.

- **RigidBody**

Ez a komponens lehetővé teszi, hogy a GameObjectre hassanak a fizika törvényei és úgy viselkedjen mint egy tárgy a valóságban. Hathat rá a gravitációs erő, vagy más erők és forgatónyomatékok befolyásolhatják a test mozgását és állapotát.

- **Collider**

Colliderek segítségével meghatározhatjuk egy test, egy objektum formáját. A collider alapvetően láthatatlan és nem szükséges, hogy a formája megegyezzen annak az objektumnak a formájával, amelyikhez hozzárendeljük azt. A Collider lényegében azt adja meg, hogy a játék világában a többi tárgy hogyan látja azt az objektumot. Colliderek segítségével tudunk ütközést detektálni avagy észlelni és ez alapján akciókat és reakciókat kiváltani. Ha egy tárgy nem rendelkezik a Collider tulajdonsággal akkor lényegében a többi objektum szemében olyan mint egy szellem és ha útjukba esik akkor egyszerűen áthaladnak rajta. Unityben van néhány előre definiált formájú primitív collider típus: Box Collider, Sphere Collider és Capsule Collider. Leggyakrabban ezeket használjuk amiket nagyon egyszerűen lehet módosítani és a GameObject formájához igazítani.

- **Script**

Unity környezetben a Script lényegében nem más mint egy fájl ami kódot tartalmaz. Lehet benne egy, de akár több osztály is definiálva, tetszőleges mennyiségű attribútummal és metódussal. Van rengeteg alapértelmezett szkript, de mi magunk is újakat hozhatunk létre. A Unityben való alkalmazásfejlesztés egyik alappillére az úgynevezett szkriptelés, vagyis a kódolás. Szkriptekkel készíthetünk grafikus effekteket, irányíthatunk objektumokat, befolyásolhatjuk a játék fizikáját, vagy akár saját mesterséges intelligenciát is írhatunk a játék karaktereinek.

- **Ragdoll**

  A Ragdoll egy alapértelmezett 3D-s GameObject. Tartozik hozzá egy úgynevezett Ragdoll Wizard, ami segítségével egy karakter prefabból rongybabát hozhatunk létre amennyiben megfelelően párosítjuk a prefab egyes elemeit a testrészekkel.

  Ha ezt sikeresen végrehajtottuk akkor az egyes testrészek Character Joint komponensek segítségével csatlakoznak egymáshoz, mint csontok és ízületek az emberi anatómiában. Mostantól a karaktert úgy kezelhetjük mint egy rongybabát.

4.4 Játék fizikája és a Collision Detection [25]

A játék alapvetően próbálja betartani a fizika szabályait bizonyos korlátozásokkal.

Annak érdekében, hogy a fal és a karakter ütközése folyamán a fal ne térjen le a pályájáról, csak az adott sík mentén tud mozogni, amit a Unity Editor felületén egyszerűen be lehet állítani. Illetve a fal objektum tömegét is jóval nagyobbra állítottam mint a karakter tömege, így biztosítva, hogy ne akadjon bele a karakterbe, ami megakasztaná a játékot.

Jelenleg a fal a kiinduló pozíciójából (*wallSpawnPosition*) a medence mögött található célpozícióba (*wallDestinationPosition*) tart. Amint eléri azt a pozíciót és nem ért véget a játék, akkor eltűnik és egy újabb fal objektum jelenik meg a kiinduló pozícióban. Ezzel egyidőben a játékos karaktere is visszakerül a kiinduló pozícióba.

Több lehetőséget is kipróbálva arra jutottam, hogy jelen projektben az a legoptimálisabb, ha minden esetben amikor a fal célba ér törlöm a játékos karakterét és egy új player nevű *GameObject*-et inicializálok az elmentett karakter *Prefab* segítségével egy *PlayerManager* nevű osztály segítségével. Ehhez a megoldáshoz a karakter eredeti és rongybaba állapota közötti átváltásból és visszaváltásból fakadó bonyodalmak vezettek, amiket a következő fejezetben részletesen bemutatok.

A fal és karakter ütközésének detektálására készítettem egy *CollisionDetection* nevű szkriptet. Ezt a szkriptet a fal objektumhoz kell rendelni, mint komponens, hogy az elvártnak megfelelően működjön. A szkript felhasználja a *UnityEngine* által tartalmazott és előre definiált *OnCollisionEnter(Collision)* függvényt. Fontos, hogy ezt

a függvényt csak olyan objektumok esetén lehet alkalmazni, amelyek rendelkeznek collider vagy rigidbody komponenssel. A függvény pedig azt vizsgálja, hogy az adott elem érintkezik-e a pályán található más collider vagy rigidbody komponenst tartalmazó elemmel.

A metódusban, ahogy alább is látható definiálni lehet, hogy mely tárgyakkal való ütközés esetén milyen eseményt generáljon. Ezt többféleképpen is meg lehet tenni. Én egyszerűen az objektumoknál megadható címke (tag) mező használatát választottam. Így ebben a kódban a fal csak azokkal az elemekkel való ütközés esetén vált ki az alapértelmezettől eltérő viselkedést, amelyek rendelkeznek a *Player* vagy a *Zombie* címkékkel.

Ha a fal ütközik a játékos karakterével, akkor még egy kis *CrashParticle* nevű effekt is meghívódik, ami a falból való törmelék darabkák leszakadását hivatott jelképezni. Az ütközés hatására a fal mozgását beállítom egy fix relatíve lassú sebességre az előbbi kódrészletben látható *SetMoveSpeed(float)* függvény használatával, hogy a játékosnak legyen elég ideje átélni a VR nyújtatta medencébe zuhanás élményét.

Az *OnCollisionEnter(Collision)* függvény kódja:

```
public void OnCollisionEnter(Collision other)
{
     if(other.transform.tag   ==   "Player"   ||   other.transform.tag   ==
"Zombie")
     {
          //print("I hit a player");
          if (doCrashParticleEffect)
          {
               crashParticles                                                =
Instantiate(playerManager.GetCrashParticlesGameObject(),
playerSpawnPosition);
               doCrashParticleEffect = false;
          }
          if (decreaseLifeAmount)
          {
               playerLifeAmount--;
               playerManager.SetPlayerLifeAmount(playerLifeAmount);
               decreaseLifeAmount = false;
          }
          EnableRagdoll();
          current_WallPatrolScript.SetMoveSpeed(5f);
     }
}
```

4.5 A Játékos karakter- és rongybaba (Ragdoll) viselkedés beállítása [26][28][29]

A Unityhez található egy Asset store nevű online tárház, vagy ha úgy tetszik "bevásárlóközpont". ide bárki feltölthet általa készített kódbázist, modelleket, játékelemeket, igazából bármit amit arra érdemesnek talál. Itt található ingyenes és fizetéses tartalom is. A játékhoz végül az ingyenesen elérhető Artalasky által készített Modern Zombie Free Asset-et [27] töltöttem le, amiből számomra a zombi kinézetű karakter prefabra volt szükségem. Ezt a karaktert irányítja a játékos.

A *ZombieRig* prefab alapvetően egy *GameObject*, aminek rengeteg leszármazottja van. Ez a leszármazási fa pedig egy emberi test anatómiailag is helyes szerkezetének a hierarchiája szerint épül fel. Például a csipőízületet szimbolizáló elem alá tartoznak a combcsontok, ezek alá a lábszár, majd lábfej és így tovább. Ez a fa hierarchia és ennek az élettanilag helyes felépítése akkor válik igazán fontossá amikor a karaktert élethű mozgásra szeretnénk bírni. Nem szeretnék itt részletesen belemenni az emberi élettan és anatómia részleteibe, de érintőlegesen megemlíteném, hogy vannak különböző szabadsági fokkal rendelkező ízületeink. Míg a vállízület például az egyik legszabadabban mozgó szerkezet a testünkben. Úgynevezett gömbízület, amiben létrejöhet minden irányú és síkú mozgás. Addig a könyökízület sokkal korlátozottabb és többnyire csak hajlításra és feszítésra képes. Illetve minimális fokú rotálásra is, de itt is például az alkar rotálása esetén egy bizonyos fok után már szükséges a felkar vállízületbeli forgása is.

Ha a karakter objektumból egy ragdollt csinálunk akkor, amennyiben helyes a prefab modell, képesek leszünk a fentieknek megfelelő anatómiailag helyes mozgásokat mintázni. Mielőtt tovább haladtam volna fontosnak tartottam, hogy kipróbáljam, hogy a letöltött modell megfelel-e ezeknek az elvárásoknak és tudok-e belőle megfelelő ragdollt készíteni. Amennyiben ez nem sikerült volna, akkor tovább kerestem volna és más ingyenes modellel próbálkoztam volna, mindaddig amíg megkapom a számomra legmegfelelőbbet. Szerencsére találtam olyan internetes tutorial videót [30] amiben pont ezt a modellt használják, így követve az ott látott lépéseket, fennakadás nélkül meg tudtam oldani ezt a feladatot és kaptam egy jól működő rongybabát. A ragdoll készítésnél a letöltött modell objektum hierarchiájából a megfelelő elemeket össze kell párosítani a készülő rongybaba megfelelő testrészeivel, ami nem minden esetben

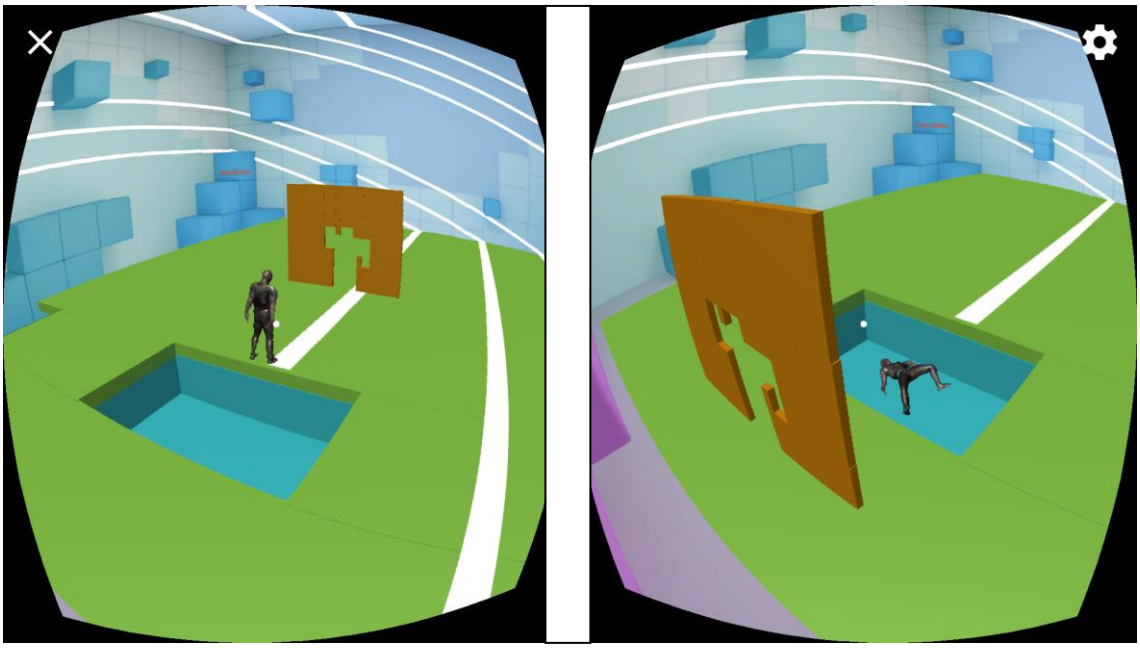
egyértelmű. Van amit a ragdoll készítésnél csak könyöknek hívnak, de oda valójában a modell alkar objektumát kell megadni. Tehát ezt nehéz mindig elsőre eltalálni és van, hogy többször is próbálkozni kell, hogy az adott karakter objektumból a megfelelő rongybaba modellt kapjuk.

Ha ezzel sikeresen megvagyunk akkor egy, a képen is látható összetett karakter modellt kapunk, aminek már a számunkra szükséges testrészeihez külön collider és rigidbody komponensek tartoznak, amelyek alapértelmezetten a keretrendszerben található character joint komponens segítségével csatlakoznak egymáshoz és alkotnak egy emberi csontváznak megfelelő szerkezetet. Ekkor ha elhelyezünk egy ilyen karaktert a pályán és elindítjuk a játékot, akkor a karakter mint egy rongybaba összerogy maga alatt, mivel alapértelmezetten minden rigidbody komponensére hat a fizika törvényeinek is megfelelő gravitációs erő. A rigidbody komponensnek van egy *isKinematic* nevű tulajdonsága, ami bekapcsolásával, nem csak azt érhetjük el, hogy ne hasson a gravitációs erő az adott testre, hiszen ahhoz elég lenne csak egyszerűen a *useGravity* nevű tulajdonságát letiltani, de ez esetben a súlytalanságnak megfelelően a test elkezdene lebegni és szép lassan elszállni a semmibe. Míg ha a komponens *isKinematic* tulajdonsága engedélyezve van akkor semmilyen alapértelmezett erő, ütközés vagy összeköttetés nem hat az elemre, vagyis ekkor lesz ez a testrész valóban merev.

Ha egy objektum rigidbody komponensének *isKinematic* tulajdonsága igazra van állítva, akkor az adott testet teljesen a rá helyezett animáció, vagy egy hozzárendelt szkript fog kontrollálni a pozíciójának (*transform.position* értékének) megváltoztatásával.

Ezt a részt azért tartom fontosnak legalább ennyire részletesen bemutatni, mert ezek a tapasztalatok és viselkedések szorosan hozzá fűződnek ahhoz, hogy miért hoztam meg azt a korábban említett döntésem, hogy minden körben a játékos karakterét törlöm és új prefabot inicializálok és helyezek el a medence szélére. Ahelyett, hogy ugyanazt az objektumot használnám, csak visszaállítanám az eredeti helyére és állapotába.

Azt szerettem volna ha a karakter minél élethűbben zuhan a medencébe, mintha a játékost valóban egy nagyobb ütés érné és már nem tudná kontrollálni a testét. Ehhez az az ötletem támadt, hogy amennyiben a fal elüti a játékost, a karakter, mint egy rongybaba összeesik, belezuhan a medencébe. Létrehoztam két állapotot, amit állapotonként egy-egy egyszerű függvény meghívásával állítok be a programkódban.



4.2. ábra: Itt ismét két külső nézőpontú (3rd person) képernyőkép összevágása látható. Bal oldali pillanatkép közvetlenül a játék indítása után készült, amikor a fal a kiinduló pizíciójából elindul a végcél irányába. A jobb oldalon pedig az látható, ahogy a fal magával sodorta a karaktert, mivel az nem vette fel a megfelelő pozíciót, így beletaszította a mögötte lévő medencébe. Az ütközést pedig a 4.4-es fejezetben bemutatott CollisionDetection eljárás detektálta.

Van egy alapértelmezett állapot, amikor a karakter a medence szélén áll és a játékos a MoCap eszköz segítségével irányítja a felsőtestet, az alsótest végig mereven áll. Illetve van egy rongybaba (*isRagdoll* értéke *true*) állapot, amibe akkor kerül, ha összeütközik a fallal, ilyenkor a játékos már teljesen elveszti a kontrollt a karaktere felett. Ezekhez az alábbi két egyszerű függvényt használtam fel, amikben továbbá beállítom az állapotot tároló isRagdoll bool típusú mező értékét is:

- Rongybaba állapot engedélyezése az *EnableRagdoll()* függvénnyel:

```
void EnableRagdoll()
{
     foreach(Rigidbody rb in player_rigidbodies)
     {
          isRagdoll = true;
          rb.useGravity = true;
          rb.isKinematic = false;
     }
}
```

- Rongybaba állapot letiltása a *DisableRagdoll()* fügvénnyel:

```
void DisableRagdoll()
{
     foreach(Rigidbody rb in player_rigidbodies)
     {
```

```
                isRagdoll = false;
                rb.useGravity = false;
                rb.isKinematic = true;
            }
        }
```

Ezek a tulajdonságok és függvények a korábban említett *CollisionDetection* szkript részét képezik. Mivel ez a szkript a fal prefabokhoz van hozzárendelve, így minden egyes alkalommal amikor egy fal objektum inicializálódik a pályán akkor lefut ennek az osztálynak a *Start()* metódusa. Ebben a metódusban ismét inicializálódik a játékos karaktere, visszakerül a medence szélén található kiinduló pozíciójába (*playerSpawnPosition*) és itt állítom be, hogy a rongybaba állapota tiltva legyen, az alábbi kódrészletben látható *DisableRagdoll()* függvényhívással. Valamint a *playerLifeAmount* lekérésével és a *decreaseLifeAmount* változó értékének igazra való beállításával segítem a játékos életeinek számontartását és kontrollálását, amire a korábban bemutatott *OnCollisionEnter()* függvény van közvetlen hatással.

A *CollisionDetection* szkript *Start()* függvényének ide vonatkozó része:

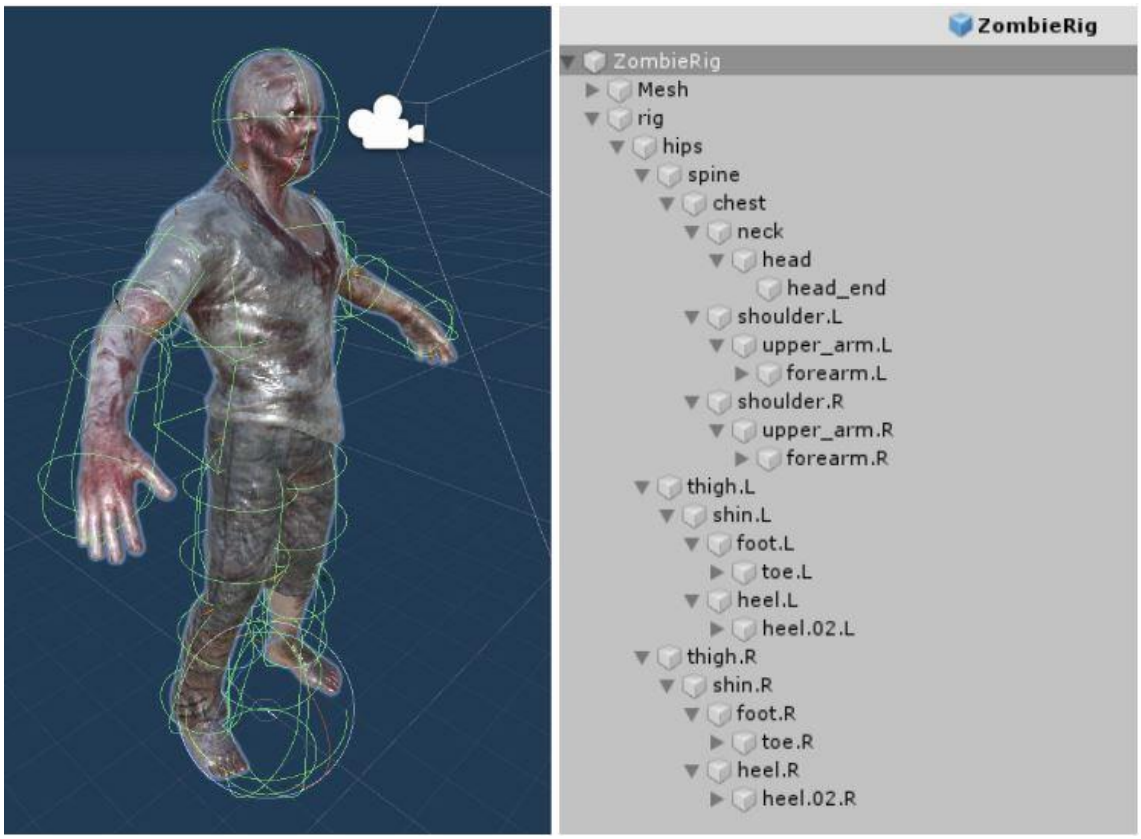
```
void Start()
{
     ...
     playerSpawnPosition                                                    =
(Transform)GameObject.Find("ZombiePlayer_Spawn_Postion").GetComponent<Tran
sform>();
     playerManager                                                          =
(PlayerManager)GameObject.Find("ZombiePlayer_Spawn_Postion").GetComponent<
PlayerManager>();
     sceneChanger                                                           =
(SceneChanger)GameObject.Find("SceneManager").GetComponent<SceneChanger>()
;
     spawnerComponent                                                       =
(Spawner)GameObject.Find("Wall_spawner").GetComponent<Spawner>();

     player       =       Instantiate(playerManager.GetPlayerGameObject(),
playerSpawnPosition);
     playerManager.SetPlayerRespawned(true);
     playerLifeAmount = playerManager.GetPlayerLifeAmount();
     isGameOver = spawnerComponent.GetIsGameOver();
     player_rigidbodies = player.GetComponentsInChildren<Rigidbody>();
     DisableRagdoll();
     player.GetComponent<Transform>().position                              =
playerSpawnPosition.position;

     wallSpawnPosition                                                      =
(Transform)GameObject.Find("Wall_Spawn_Position").GetComponent<Transform>(
);
     wallDestinationPosition                                                =
(Transform)GameObject.Find("Wall_Destination_Position").GetComponent<Trans
form>();
     wall = GetComponent<Transform>();
     doCrashParticleEffect = true;
```

```
        decreaseLifeAmount = true;
    }
```

Bármilyen megoldással próbáltam a kiinduló állapotot, a testrészek pozícióját, *transform* paramétereinek értékét elmenteni, majd az alapján visszaállítani az eredeti pozícióba a karaktert nem sikerült ezt elérjem. Annyira összetett már ez a karakter prefab, rengeteg leszármazott eleme van, ezek közül sok külön rendelkezik rigidbody, collider és character joint elemekkel. Egy ütközés esetén pedig ezek rengeteg attribútumának megváltozik az értéke, így végül egyszerűbb és célravezető volt azt a megoldást választani, hogy minden kör végén törlöm a jelenlegi karaktert és a következő kör elején újat hozok létre.



4.3. ábra: Bal oldalon a Ragdoll Wizard segítségével elkészített Zombie Prefab látható, ahol jól elkülöníthetőek a testrészek a varázsló segítségével hozzájuk rendelt Collidereknek köszönhetően. Jobb oldalon pedig a karakterhez tartozó bőséges objektum hierarchia látható.

4.6 Karakter mozgása, testrészek rotációja

A következő kihívást a karakter megfelelő mozgatása jelentette. Mint azt már korábban is írtam, egy Notch készlettel amiben hat szenzor van nincs lehetőség lefedni a teljes test mozgásának követését. Ahhoz minimum két, de optimális esetben három

készletre, vagyis tizennyolc szenzorra lenne szükség. Ezek fényében a játék egyelőre úgy működik mintha a karakter bele gyökerezett volna a földbe és nem tudja mozgatni az alsótestét, nem tud elmozdulni az adott pozícióból, csak a felsőtestét tudja irányítani.

4.6.1 LocalRotation, LocalEulerAngles, Rotation, EulerAngles, Quaternion [33]

Ezen a ponton vált igazán fontossá, hogy jobban elmélyítsem a tudásom a Unity-ben való testek és tárgyak mozgatását, fizikáját és transzformációját illetően. Ehhez egy könnyen feldolgozható, szemléletes és jól elmagyarázott tutorial sorozatot segített. [32] Amiből sikerült megértsem a különböző rotációval és pozícióval kapcsolatos terminológiák közötti különbséget. Például, hogy mi az eltérés a Rotation és a *LocalRotation* között vagy az *EulerAngles* miben különbözik a *LocalEulerAngles*-től és mi is az a *Quaternion*.

Ahhoz, hogy néhány kódrészlet, fejlesztői döntés, fennakadás és megoldás értelmezhető legyen a továbbiakban, szeretném ezeket a terminusokat a teljesség igénye nélkül, röviden a Unity dokumentációra hivatkozva saját szavaimmal bemutatni.

Unityben 3D-s rotációt végre lehet hajtani akár három megközelítésben is:

1. Euler szögekkel (Euler Angles)
2. Kvaterniókkal (Quaternions)
3. Megadott tengely körüli forgatással (Axis Angles)

Gimbal lock probléma: elveszítünk egy szabadsági fokot és a tárgyat tulajdonképpen csak két tengely körül tudjuk forgatni. Például egy kockát elforgatunk 90 fokkal az x tengely szerint, majd ebből a pozícióból hiába akarjuk y és z szerint is tovább forgatni, a kocka csak az y szerint fog már csak elfordulni.

Az *eulerAngles* típusuk igen, míg a *Quaternion* típusok nem rendelkeznek a Gimbal lock problémával.

4.6.1.1 Euler-szög alapú forgatás

Transform.eulerAngles: elfordulást ír le az x, y és z tengelyek körül az eredeti világkoordináta-rendszerhez (world coordinate space) viszonyítva. Mértékegysége tengelyenként a fok. Egy háromdimenziós vektorral (vec3) írjuk le. Például (90, 0, 0) az x tengely körüli 90 fokkal való elforgatást jelenti.

(Magyarul Euler-szögeknek nevezzük és egy merev test helyzetét írjuk le ennek segítségével egy euklideszi térben rögzített háromdimenziós koordináta-rendszerhez képest)

Transform.localEulerAngles: az *eulerAngles*-hez hasonló, csak ha van szülő (parent) objektum, akkor az elfordulás a szülő koordináta rendszeréhez viszonyítva írja le.

4.6.1.2 Kvaternió alapú forgatás

Quaternion: valamilyen forgást reprezentál x, y, z és w négykomponensű rendszer alapján. (Magyarul kvaternióknak nevezzük, amik a komplex számok négy dimenziójára történő nem kommutatív kiterjeztései.) Így például kvaterniókkal az x tengely körüli 90 fokos elfordulást az alábbi módon tudjuk leírni: (0.707, 0, 0, 0.707).

Transform.rotation: kvaternió alapú elfordulást ír le a világkoordináta-rendszerhez viszonyítva.

Transform.localRotation: mint a rotation, csak az elfordulást a szülő objektum koordináta rendszeréhez viszonyítva írja le.

Az utóbbi kettő, mivel kvaternió alapú ezért nem szenvednek a Gimbal lock problémától.

4.6.1.3 Megadott tengely körüli forgatás

Quaternion.AngleAxis(float angle, Vector3 axis): lényegében ez is kvaternió alapú, csak itt az első paraméterrel megadjuk az elforgatás fokát, majd a második paraméterrel egyszerűen azt állítjuk be, hogy melyik tengely körül szeretnénk az adott fokkal elforgatni a tárgyat.

Transform.Rotate(Vector3 axis, float angle): ezzel a függvénnyel tárgyakat tudunk forgatni. Az első paraméterben egy Euler-szögeket használó vec3 típussal adjuk meg a kívánt elfordulást. A második paraméterben pedig az adható meg, hogy mely koordináta-rendszerhez viszonyítva forogjon a test. Alapértelmezetten a saját (*Space.Self*) koordináta-rendszeréhez képest forog, de akár megadható a világ (*Space.World*) koordináta-rendszere is alapul.

Már megismertük, hogy hogyan lehet a Demo App-ban a Notch Service segítségével a szenzorok adatait megfelelő formában átadni a *UnityPlayerActivity*-nek és ennek

segítségével üzenetként továbbítani a Unity projektben a C# nyelven megírt szkripteknek és megjeleníteni a kijelzőn egy szöveges mezőben.

Viszont ezen a ponton már nem csak az adatokat szeretném kiírni a képernyőre, hanem azokat konkrétan felhasználni a C# alapú kódban a karakter testrészeinek mozgatására.

4.6.2 Tesztelési eljárás

A mozgatást sok esetben először egy egyszerű kocka objektumon teszteltem ami a karakter törzsét szimbolizálta, vagyis a mellkasra rögzített fő szenzort. Ha úgy tűnt, hogy a mozgás megfelelően működik a kockán, akkor a kódot átvetítettem a karakter objektum Chest elemére és kipróbáltam azon is. Csak azután próbálgattam három vagy öt szenzor használatával a törzs és karok mozgatását is, ha már meggyőződtem arról, hogy a törzs mozgása az elvártak alapján viselkedik.

A tesztelési fázist általtában mindig két fő lépésben valósítottam meg:

1. Billentyűzettel
2. Szenzorokkal

4.6.2.1 Tesztelés billentyűzettel

Akár a kocka objektumról, vagy a chest elemről legyen szó először minden módosítás alkalmával a billentyűzettel próbáltam ki az implementált vagy átírt kód aktuális működését. Így el tudtam kerülni azt a lassú folyamatot, hogy minden tesztelést igénylő változtatás alkalmával ismét exportálni kelljen a Unity projektet és felülírni az ebből létrehozott Android library plugin-t, build.gradle és AndroidManifest.xml fájlokat. Ami egy időigényes folyamat és amit mindezek ellenére sokszor meg kellett tennem, olyan változtatások esetén amikor fontos volt a működést tesztelni a szenzorokkal is. A billentyűzettel való forgatást egyszerűen meg lehet tenni a Unity környezetbe épített eszközökkel. Alapértelmezetten a nyilakkal illetve a WASD gombokkal lehet irányítani, mozgatni és/vagy forgatni az adott testet.

Egy egyszerű példa a forgatásra:

```
inputRotation = new Vector3(Input.GetAxisRaw("Vertical"), 0.0f, -
Input.GetAxisRaw("Horizontal"));
rb.transform.Rotate(inputRotation * Time.deltaTime * rotationSpeed);
```

Ezzel a kóddal például az adott test (rigidbody) a fel/le nyilakkal vagy a W/S billentyűkkel elforgatható az X tengely körül. míg a jobbra/balra nyilakkal vagy az A/D billentyűkkel elforgatható a Z tengely körül.

A fejlesztőkörnyezetben alapértelmezetten a mozgások és változások úgymond framenként (magyarul filmkockánként) történnek. Azt, hogy hány filmkocka frissítés történik egy másodperc alatt az úgynevezett framerate vagy FPS (frame-per-second) határozza meg. Ez általában 30 FPS-t jelent, de eszközönként eltérő lehet. A Unity alapbeállításként a használt eszköz FPS értékét használja a rendereléshez, ami az eltérő standardok miatt problémás lehet. Ezért ajánlott a *Time.deltaTime* használata ilyen kódrészletekben. Ekkor nem framenként, hanem másodpercenként történik a mozgás és egy eszközöktől független, egységes élményt kapunk.

A *PlayerRotation* és a *DemoRotation* nevű szkripteket a testrészek forgatásához használt ideális megoldás megtalálásához használtam. Ezekben próbáltam ki a korábban említett tutoriál sorozatban bemutatottakat és próbáltam gyakorlatba átültetni az ott látottakat és tanultakat. Ezekben a kódokban még továbbra is billentyűzettel teszteltem.

4.6.2.2 Tesztelés szenzorokkal

A szenzorokkal való tesztelést az előzőekkel ellentétben már kizárólag a telefonos alkalmazás használatán keresztül tudtam minden esetben kipróbálni. Ehhez a szükséges eljárásokat továbbra is a *PluginWrapper* nevű osztályba implementáltam.

A Chest master bone-hoz tartozó vektor értékeket a *PluginWrapper* szkript *chestAngleX_f, chestAngleY_f, chestAngleZ_f* float típusú változói veszik át. Ezeket felhasználva a szenzorokkal való tesztelésnél bemutatott kódrészlet pedig így néz ki:

```
inputRotation = new Vector3(chestAngleX_f, chestAngleY_f, -chestAngleZ_f
);
rb.transform.Rotate(inputRotation * Time.deltaTime * rotationSpeed);
```

Ez a kód első ránézésre megfelelőnek tűnt és a félév elején nagy sikerélmény volt mikor megláttam, hogy a zombi karakter törzse a játékban úgy mozog ahogy a szenzort forgatom. Később viszont rá kellett jöjjek, hogy ez több szempontból sem működik optimálisan és két fő probléma van vele:

- Az egyik, hogy a *Rotate()* metódust használom, amit akkor még nem, de az itt már bemutatottak alapján tudjuk, hogy Euler-szögekkel dolgozik míg a háttérben kvaternió alapú számítások zajlanak.

- A másik problémát ez esetben a *Time.deltaTime* használata jelentette, mivel ennek hatására a forgás soha nem állt le. Elég volt csak 10 fokkal megdönteni a szenzort valamelyik tengely mentén, aminek hatására másodpercenként elfordult a test 10 fokkal abba az irányba. Amit el szerettem volna érni az pedig csak annyi lett volna, hogy elforduljon 10 fokkal, majd megálljon abba a pozícióba.

Végül a problémák kiküszöbölésére és a helyes szenzor szerinti forgatásra a következő megoldást találtam:

```
inputChestRotation = new Vector3(chestAngleX_f % 360, -chestAngleY_f %
360, -chestAngleZ_f % 360);
Quaternion rotationY = Quaternion.AngleAxis(inputChestRotation.y, new
Vector3(0f, 1f, 0f));
Quaternion rotationX = Quaternion.AngleAxis(inputChestRotation.x, new
Vector3(1f, 0f, 0f));
Quaternion rotationZ = Quaternion.AngleAxis(inputChestRotation.z, new
Vector3(0f, 0f, 1f));
rb.transform.localRotation = rotationX * rotationY * rotationZ;
```

Ugyanezt a megoldást használtam rendre a felsőtest többi elemére is: *rb_leftUpperArm, rb_leftForeArm, rb_rightUpperArm, rb_rightUpperArm, rb_rightForeArm*

Az rb ezekben az esetekben a rigidbody rövidítését takarja. Ezek a változók a *PluginWrapper Update()* metódusában veszik fel az értéküket az alábbi kódrészlet segítségével:

```
rb =
(Rigidbody)GameObject.Find("ZombiePlayer_Spawn_Postion/1VRView_Zombie_Play
er(Clone)/rig/hips/spine/chest").GetComponent<Rigidbody>();
rb_leftForeArm =
(Rigidbody)GameObject.Find("ZombiePlayer_Spawn_Postion/1VRView_Zombie_Play
er(Clone)/rig/hips/spine/chest/shoulder.L/upper_arm.L/forearm.L").GetCompo
nent<Rigidbody>();
rb_rightUpperArm =
(Rigidbody)GameObject.Find("ZombiePlayer_Spawn_Postion/1VRView_Zombie_Play
er(Clone)/rig/hips/spine/chest/shoulder.R/upper_arm.R").GetComponent<Rigid
body>();
rb_rightForeArm =
(Rigidbody)GameObject.Find("ZombiePlayer_Spawn_Postion/1VRView_Zombie_Play
er(Clone)/rig/hips/spine/chest/shoulder.R/upper_arm.R/forearm.R").GetCompo
nent<Rigidbody>();
```

Látható, hogy értékül kapják a példányosított *1VRView_Zombie_Player* néven elmentett prefab karakter megfelelő testrészhez tartozó gyerek elemét.

Majd ennek a szkriptnek a *FixedUpdate()* nevű függvényében hajtódik végre a mozgatás amennyiben a karakter nincs rongybaba állapotba, amit az alábbi feltétellel ellenőrzök. Ahol a pontozott rész takarja a bemutatott kvaternió alapú tengelyek szerinti forgatást a felsorolt testrészekre:

```
if (!collisionDetection.IsRagdoll()) { . . . }
```

5 Virtuális valóság [34]

A virtuális valóság fogalom definíciója még a mai napig sem pontosan meghatározott. Többnyire amolyan gyűjtőfogalomként használjuk különböző széles körben alkalmazható technológiákra. Talán leginkább úgy tudnám meghatározni, hogy a virtuális valóság nem más mint egy mesterséges világ, ami valós érzetet nyújt és a valóság benyomását kelti de nem más mint egy számítógépek generálta világ. György Péter megfogalmazásában a digitális technikával létrehozott és az általa felkeltett perceptuális élmények egészét érjük virtuális valóság alatt. Ez nem csak a látó szervünket befolyásoló képi virtualitás lehet, hanem lehet hangi vagy akár érintésbeli virtuális valóság is. Tulajdonképpen bármely érzékszervünkre hatással lehetünk és próbálunk is hatással lenni a virtuális valóságban, hogy még inkább a realitás érzetét keltsük. Gyakran hívják ezeket napjainkban négy- vagy ötdimenziós élményhatásnak, ami nem több egy marketing fogásnál.

A virtuális- és kiterjesztett valóság alapú technológiákat manapság már nagyon széles körben kezdte használni az emberiség: sportokban, oktatásban, orvostudományban, hadászatban, tervezésben, rehabilitációban, modellezésben és természetesen a számítógépes és egyéb digitális eszköz alapú játékiparban is.

5.1 VR és AR eszközök

A virtuális valóságot támogató eszközöket már igazából az 1970-es évektől fejlesztenek. Mint sok egyéb digitális eszköz fejlesztési területén, így az első nagy átlépéseket ez esetben is a hadászatban érték el. A virtuális valóságot többnyire repülés szimulátorok kifejlesztésére majd katonasági gyakorlati-szimulációk céljából fejlesztették.

A játékiparban ma is használt eszközökhöz hasonló első VR szemüveg a Sega VR volt, amit a Sega Games Ltd. Japán játékfejlesztő cég jelentett be még 1991-ben. A 90-es évek elején ez az elképzelés túl nagy falatnak és túl merésznek tűnt. Néhány évnyi támogatottság után 1995 környékén viszont ez a projekt a Sega részéről valószínűleg a várt siker elmaradásából és egyéb anyagi problémákból kifolyólag háttérbe került.

A Sega VR fejlesztéseinek hátterébe kerülésével egyidőben viszont egy másik neves Japán cég a Nintendo Ltd. előrukkolt a Virtual Boy nevű VR konzoljukkal ami sajnos úgyszintén nem aratott túl nagy sikert.

A Sega és a Nintendo próbálkozásai után több mint másfél évtizedig nem rukkolt elő egy cég sem valami igazán nagy áttörést jelentő újítással. 2010-ben viszont az Oculus VR nevű vállalat előállt egy Oculus Rift nevű headset prototípussal, ami 2012-ben piacra is került. Úgy festett, hogy ez akkoriban sikerült újabb áttörést hozzon a virtuális valóság területén és több neves nagyvállalat, mint a Sony, a Valve, a HTC és mások is őrült fejlesztésekbe kezdtek. Ennek hatására megjelent rengeteg új virtuális valóságot támogató eszköz és kiegészítő. Valamint telefonokra is egyre több VR és AR alapú alkalmazást kezdtek fejleszteni, amihez különböző cégek, különböző dizájnnal dobták sorba piacra az úgynevezett szemüvegeiket, headseteket.

5.1.1 Google VR Cardboard headset

Az egyik leghíresebbé a Google terméke a Google Cardboard vált. A Cardboard kifejezés ezekben a körökben már egyértelműen egy olyan fejünkre helyezhető kis kartondobozra utal, amibe egy lencsepár elé helyezzük a telefonunkat, hogy a megfelelő VR élményt kapjuk az alkalmazás által.

Természetesen már ennek is található mindenféle változata a piacon, különböző mintájú és formájú dobozokat vásárolhatunk, de a legegyszerűbb verzió akár házilag is elkészíthető. A Google Cardboard lényegében egy külön brandé nőtte ki magát.

2016-ban a Cardboard nagy sikernek köszönhetően a Google bejelentette az újabb, eggyel modernebb telefonos VR alkalmazásokat támogató platformját, a Google Daydream-et. Ez az eszköz továbbra is egy teljesen elérhető árkategóriába tartozik és az egyik legnagyobb újítás, hogy ehhez a készülékhez már jár egy kis külső kontroller is, ami újabb lehetőségeket kínál a VR mobil-app fejlesztőknek. Mivel a felhasználóknak most már van egy bemeneti eszközük is, amivel sokoldalúan programozható műveleteket hajthatnak végre a játékokban.

5.2 Google VR SDK Android alapú használata Unityben [35][36]

A megnövekedett fejlesztői igényeknek köszönhetően egyre több fejlesztőkörnyezetet kezdett támogatni a Google VR SDK [37]. Az iOS, Web és Unreal

támogatása mellett, teljes körben támogatják a Unity és Android fejlesztői környezeteket is. Sőt az Androidra való fejlesztéshez egy NDK is rendelkezésre áll, így alkalmazásokat lehet készíteni akár natív kóddal is, C és C++ nyelven. A gépi kód előnye, hogy közvetlenül hozzáférhetünk és használhatunk bizonyos program-könyvtárakat, jobban optimalizált kódot tudunk készíteni. Bizonyos appok esetében fontos lehet a processzor kihasználtság minimalizálása, vagy a megfelelő memóriahasználat elérése, a fölösleges erőforrás használat csökkentése.

Az SDK pedig egy úgynevezett eszköztár, amiben előre definiált osztályok és metódusok segítik a fejlesztők munkáját, ahogy ezt a 2.5-ös fejezetben kifejtettem. Míg a Notch Android SDK feladata, hogy támogassa az Androidos alkalmazásfejlesztést a Notch Pioneer eszköz használatával, addig a Google VR SDK natív API támogatást nyújt a virtuális valóság alapú fejlesztés kulcsfontosságú elemeihez. Ilyenek például a felhasználói bemenetek és ezek kezelése, a fejmozgás követése és Daydream esetén a kontroller támogatása.

A játékomat eddig a pontig mindig egy külső úgynevezett third-person nézetű kameraállásból futtattam és teszteltem. A VR technológia alapjainak, történelmének és fejlődésének megismerése után beleástam magam a Google Cardboardra való alkalmazásfejlesztésbe. Implementáltam ezt az alkalmazásba így ki lehet használni a virtuális valóság nyújtotta élvezeti faktort, ami igazán különössé tudja tenni a játékélményt.

Ahhoz, hogy a Unityben készülő játékot a telefonon is a tipikus osztott képernyős virtuális valóság nézetbe lehessen futtatni és mindez működjön az eddigi koncepcióval vagyis úgy, hogy a játék a Demo App-ból indul el, az alábbi lépéseket és beállításokat volt szükséges megtennem:

- A Unityhez tartozó Google VR SDK letöltése a Google hivatalos fejlesztői oldalán elérhető GitHub könyvtárból. Jelenleg a legfrissebb verzió a v1.200.1-es. [38]
- A letöltött SDK-t importáljuk az aktuális Unity projektbe.
- Ennek következtében a projekt mappa szerkezetében kapunk egy GoogleVR nevű alkönyvtárat, ami sok szkriptet, előre definiált komponenseket és objektumokat tartalmaz, amik mind hasznosak lehetnek egy VR alkalmazás felépítéséhez.

- Ezután győződjünk meg arról, hogy a a *Build Settings*-nél továbbra is az Android van kiválasztva, mint platform. Ha így van, akkor nyissuk meg a játék beállításokat a *Player Settings* gombra kattintva. Egyébként először állítsuk át Android Platformra.
- A *Player Settings*-nél először át kell állítani a *Minimum API Level*-t legalább 19-re, amennyiben eddig annál kisebb volt kiválasztva.
- Majd ugyanitt navigáljunk az *XR Settings* menüponthoz, ahol a *Virtual Reality Supported* opció engedélyezésével beállítjuk a virtuális valóság támogatottságot és a megjelenő üres *Virtual Reality SDKs* listához adjuk hozzá a *Cardboard* lehetőséget.
- Vagy töröljük a meglévő kamerát (az is elég ha a Tag mező értékét Untagged-re állítjuk és letiltjuk az *Audio Listener* komponenst, hogy ne okozzon konfliktust az új kamerával) és hozzunk létre egy újat úgy, hogy egy üres *GameObject* gyerek eleme legyen. Vagy a meglévő kamerát rendeljük egy üres *GameObject* alá, amit én *HeadCameraHolder*-nek neveztem el. Érdemes lehet az üres objektumot először a kamera gyerek elemeként létrehozni, majd megcserélni az alá-fölé rendelt viszonyt, annak érdekében, hogy a két tárgy középpontja megegyező legyen és ez ne okozzon problémát a későbbiekben a kamera pozíciójának beállításánál. A kamerát pedig *FirstPersonView_MainCamera*-nak neveztem el. Fontos, hogy a *Tag* mező értéke *MainCamera* legyen.
- Ahhoz, hogy a játék megkapja a megfelelő VR nézetet és az ahhoz tartozó alapértelmezett viselkedést, lényeges a jelen Scene-hez hozzáadni az SDK-val járó *GvrControllerMain* prefabot. Ez a prefab mindössze egyetlen szkript komponenst tartalmaz. Ez a komponens pedig a *GvrControllerInput*, amelynek segítségével kezelni tudunk különböző felhasználói bemeneteket.
- Továbbá nem kötelező de a számítógépes tesztelési folyamatot elősegíti, ha hozzá adjuk a *GvrEditorEmulator* nevű prefabot is. Ennek segítségével a fejlesztői felületen indított tesztelés során tudjuk emulálni a fejre helyezett Cardboard headset mozgását, a térben való nézelődést, a fej döntését és forgatását. Mindezt az Alt és Ctrl gombok lenyomásával és az egér mozgatásával tehetjük meg.

- Ezek voltak eddig a Unity-ben elengedhetetlen változtatások. Lehet ismét exportálni a projektet Androidra majd importálni Android Studióba és felülírni a jelenlegi plugin library-t a korábbi *AndroidManifest.xml* és *build.gradle* fájlok módosításának megtartásával. Valamint további módosításokat kell tenni ezekben a fájlokban, mert az alkalmazás jelen állapotában leáll, amikor a Demo App-ból próbáljuk a játékot elindítani.
- Az *Android_VR_Game_using_Notches* modul *build.gradle* kódjának dependencies részében include-olni kell az összes *.aar* végű fájlt is, ezt a megoldást a korábbi kóddal együtt így is implementálhatjuk:

```
implementation fileTree(dir: 'libs', include: ['*.jar', '*.aar'] )
```

- Ugyanitt a release résznél fontos, hogy a *ProGuard* fájlt a korábbitól eltérően az alábbi módon állítsuk be és a nem szükséges részeket távolítsuk el vagy kommentezzük ki:

```
release {
     minifyEnabled true
     proguardFiles.add(file('../../proguard-gvr.txt'))
     /*minifyEnabled false
     useProguard false
     proguardFiles    getDefaultProguardFile('proguard-android.txt'),
'proguard-unity.txt'
     signingConfig signingConfigs.debug*/
}
```

- A *minSdkVersion*-t mindenhol legalább 19-re kell állítani.
- Végezetül pedig az *AndroidManifest.xml* fájl tartalmát is módosítani szükséges:
    - A *WRITE_EXTERNAL_STORAGE* hozzájárulást (permission) tartalmazó tagben felveszünk egy attribútumot, amivel a ‘maxSDKVersion’-t a ‘TargetVersion’-nel megegyezően állítjuk be:

      ```
      android:maxSdkVersion = “28”
      ```

    - Valamint az intent-filter részben egy újabb Cardboard működést támogató elemet adunk hozzá a kategória listához:

      ```
      <category android:name=”com.google.intent.category.CARDBOARD”/>
      ```

Az itt leírt lépések és változtatások után már tesztelni tudtam a játékot. További apró simításokat végeztem rajta, elkészítettem a VR reszponzív menüt és javítottam a felmerülő bugokat. Továbbá ismerősöket, barátokat és családtagokat kértem meg arra,

hogy próbálják többször, több beálítással játszani a játékot és a tőlük érkező visszajelzések alapján finomhangoltam a menü működését, a zene alapértelmezett hangosságát és egyéb alapkonfigurációkat.

Az 5.1. ábra képernyőképein végigkísérhetjük a játékmenetet. Az 1-6 képek egy olyan szituációt ábrázolnak, amikor a fal eltalálja és belelöki a játékost a medencébe, de a játékosnak még marad élete viszont a falak már elfogytak, így a játékos sikerrel járt és győzött. Ezért a „YOU WIN” felírat fogadja pár másodpercig, majd visszakerül a főmenübe. A 7-es és 8-as kép pedig a használatban lévő Notchok láthatóak, a töltőállomásban illetve a pántok segítségével a megfelelő testrészekre felhelyezve. A 9-es képen a játékmenet egy olyan befejezése látható, amikor a játékosnak nem marad több élete, így veszített és a „GAME OVER” felírat fogadta. Ezután ismét visszakerül a főmenübe és tovább szórakozhat a játékkal, ha úgy tetszik más nehézségi beállításokon.

5.1. ábra: A játékmenetet ábrázoló képernyőképek összessége.

6 Hogyan tovább? – projektben rejlő lehetőségek

Egy élmény volt kipróbálni és alkalmazást fejleszteni a Notch Pioneer mozgáskövető rendszer segítségével. Habár olykor jelentős kihívást jelentett és hosszasabb elakadást is okozott, egy nagyon hasznos tapasztalat volt Unityben és Android Studióban párhuzamosan fejleszteni egy telefonos alkalmazást. Mindkét fejlesztőkörnyezetben jobban elmerülni és fejleszteni a programozói készségem úgy Javában, mint C#-ban egyaránt.

A projektemmel a félév végére kitűzött célom elértem és még sikerült extra funkciókat is megvalósítanom, amelyek nem voltak előre betervezve. Ez az alkalmazás megfelelő alapul szolgálhat egy összetettebb, komplexebb, több játékmóddal rendelkező alkalmazásnak, de akár egy olyan Fitness app-nak is, amit nem játékként, hanem edzői, oktatói segédeszközként a sportteljesítmény fejlesztésének céljából lehet majd alkalmazni. Például a helyes gyakorlat kivitelezés, helyes testtartás megtanítását segítheti kezdő sportolók számára, vagy haladóknál a kivitelezési forma tökéletesítésében tud segíteni, hogy a sérülések veszélyét csökkenteni, míg a teljesítményt növelni tudjuk. Mint személyi edző rengeteg ötletem van ezen technológia alkalmazásával kapcsolatban, aminek egy részéről még nem szeretném lerántani a leplet.

Következő lépésként, ha tulajdonomba kerülne legalább két vagy akár három ilyen MoCap eszköz, vagyis lenne összesen 12-18 Notch-om, akkor lefejleszteném, hogy a játékos a karakter teljes testét irányítani tudja, lépkedni, ugrani és guggolni is tudjon a virtuális térben. Ez azonban nem föltétlenül lenne egyszerű feladat az eddig szerzett tapasztalatok felhasználásával sem, hiába hangzik annak. Az inerciarendszer alapú mozgáskövetők bemutatásánál leírt enyhe pontatlanság és sodródás futásidő szerinti összeadódásából fakadó eltérés jelentős problémát és meglehetősen furcsa viselkedést produkálhat. Hogy ez mennyire lenne zavaró csak akkor tudnám megmondani, ha ki tudnám próbálni és így tesztelni a játékot. Nem tartom kizártnak, hogy ezen szoftveresen csökkenteni tudnék és optimalizálni lehetne annyira, hogy ne romoljon a játékélmény.

Továbbá a játék kinézetén még sokat lehet csiszolni. Egy grafikust és modellezőt érdemes lenne bevonni a fejlesztési folyamatokba, aki ezekért a feladatokért, úgymond a frontendért felelne.

Egy másik érdekes funkció, amivel tovább lehet fokozni a játék élvezeti faktorát és közösségi élményét, egy többjátékos mód implementálása lenne. Viszont egy ilyen játéknál ez csakis abban az esetben lenne élvezhető ha mindenki a saját telefonjáról, a saját MoCap eszközével játszaná és nem körönként kéne átadogatni egymásnak az egész felszerelést. Ehhez viszont valamilyen összeköttetést kellene létrehozni a telefonok között. Ha belefognék ennek a lefejlesztésébe, akkor valószínűleg már egyből egy internetes szerver-alapú összeköttetést próbálnék megvalósítani. Ezáltal több játékos úgyis versenybe tudna szállni egymással, hogy nem kellene feltétlen egy helyen tartózkodjanak. Így lényegében kapnánk egy multiplayer játékot.

A hab a tortán pedig az lenne, ha ez az egész átültetődne AR környezetbe. Virtuális valóságból áttérnénk kiterjesztett valóságra. Egy szabadtéri edzés élményt tovább lehetne fokozni kiterjesztett valóság alapú akadályokkal. Például futás közben időközönként ki kell kerülni vagy átugrani bizonyos akadályokat, amik a valóságban nincsenek ott. Ha ráadásul még egy további okos kütyüvel a felhasználó pulzusát is monitorozzuk, akkor olyan akadályokat rakhatunk elé, amik a pulzusát folyamatosan egy megadott zónában tartanák. Viszont erre az élményre sajnos a jelenlegi technikai korlátozások miatt még negatív hatással lenne az a tény, hogy folyamatosan fejünkön kell hordanunk egy kartondobozt benne a telefonunkkal, ami folyamatosan rázkódik, lötyög és elhúzza a fejünket.

Irodalomjegyzék

[1] Wikipedia: *Unity (játékmotor)*, https://hu.wikipedia.org/wiki/Unity_(játékmotor) (változat: 2019. október 30., 20:19)

[2] Vizi Előd: *Önlab – Android VR game using Unity and Wearnotch*, Github open repository és Wiki: https://github.com/vizielod/Onlab-Android_VRgame_using_Unity_and_Wearnotch (revision 24 May 2019, latest commit 25 May 2019)

[3] Wikipedia: *Motion Capture*, https://en.wikipedia.org/wiki/Motion_capture (revision ENG site: 01:31, 9 December 2019; HUN site: 2014. december 22., 13:05)

[4] Wikipédia: *Visszatükrőzés*, https://hu.wikipedia.org/wiki/Visszatükrözés (változat: 2019. október 27., 15:15)

[5] Wikipedia: *Inertial navigation system*, https://en.wikipedia.org/wiki/Inertial_navigation_system (revision 16:05, 4 December 2019)

[6] Wikipédia: *Inerciarendszer*, https://hu.wikipedia.org/wiki/Inerciarendszer (változat: 2018. november 2., 15:50)

[7] Wikipedia: Six degrees of freedom, https://en.wikipedia.org/wiki/Six_degrees_of_freedom (revision 05:49, 7 November 2019)

[8] Notch Interfaces Inc.: *Wearnotch Motion Capture Products*, https://wearnotch.com/ (2018)

[9] Notch Interfaces Inc.: *Wearnotch Developers SDK and Documentation*, https://wearnotch.com/developers/docs/sdk/ (2019)

[10] Google Play Store: *Notch Pioneer App*, https://play.google.com/store/apps/details?id=com.wearnotch.notchdemo (last refresh on 12 November 2019, current version: 1.10.0)

[11] BME Automatizálási és Alkalmazott Informatikai Tanszék: *Mobil és webes szoftverek (VIAUACoo)* tárgy, Hallgatói jegyzetek, https://www.aut.bme.hu/Course/mobilesweb (2019. szeptember)

[12] Szegedi Tudományegyetem hallgatói bejegyzések, Nagya felhasználó cikke: *Android programozás #3 | Alkalmazás felépítése*, http://www.dotnetszeged.hu/mic/?p=1537 (2016. november 3.)

[13] Android Developers Documentation: *android.app*, https://developer.android.com/reference/android/app/package-summary (2019)

[14] JavaTpoint: *Android Service Tutorial*, https://www.javatpoint.com/android-service-tutorial

[15] Stackoverflow: *Errors managing the UnityPlayer lifecycle in a native android application*, https://stackoverflow.com/questions/23467994/errors-managing-the-unityplayer-lifecycle-in-a-native-android-application (2014 május)

[16] Steve Pomeroy: *android-lifecycle* GitHub open repository, https://github.com/xxv/android-lifecycle (utolsó commit 2014. augusztus 18.-án)

[17] Pcforum Szótár: *SDK*, https://pcforum.hu/szotar/?term=SDK&tm=miaz

[18] Stanko Krtalić Rusendić: *Do you really need WebSockets?*, https://blog.stanko.io/do-you-really-need-websockets-343aed40aa9b (2018. március. 29.)

[19] Wikipédia: *WebSocket*, https://hu.wikipedia.org/wiki/WebSocket (változat: 2019. november 19., 00:45)

[20] BTO: *Unity Tutorial – How to Set Up the Android SDK, JDK (Fix Included) ~* 2018!, YouTube tutoriál videó, https://www.youtube.com/watch?v=GEQSaF4LPgk (2018. február 28.)

[21] Stackowerflow: *How to import unity project into existing Android studios project?* – velval nevű felhasználó által adott válasz, https://stackoverflow.com/questions/35535310/how-to-import-unity-project-into-existing-android-studios-project (2017. augusztus 11., 03:43)

[22] Jinbom Heo: *Unity-Android-Communication* GitHub open repository, https://github.com/inbgche/Unity-Android-Communication (utolsó commit 2015. május 19.-én)

[23] Anna Kravchuk: *Android+Unity Integration*, https://medium.com/@ashoni/android-unity-integration-47756b9d53bd (2017. január 7.)

[24] Unity Technologies: *Unity Documentation - Unity User Manual*, https://docs.unity3d.com/Manual/index.html (Version: 2019.2)

[25] Unity Technologies: *Unity Documentation – Scripting API - Collision*, https://docs.unity3d.com/ScriptReference/Collision.html (Version: 2019.2)

[26] Unity Technologies: *Unity Documentation – Unity Manual – Ragdoll Wizard*, https://docs.unity3d.com/Manual/wizard-RagdollWizard.html (Version: 2019.2)

[27] Unity Asset Store: *Artalasky – Modern Zombie Free*, https://assetstore.unity.com/packages/3d/characters/humanoids/modern-zombie-free-58134 (Version: 1.0)

[28] Packt: *Ragdoll Physics*, https://hub.packtpub.com/ragdoll-physics/ (2016. február 19, 12:00)

[29] Jason Weimann: *How to de Easy unity3D Ragdoll Physics with Source Code – unity ragdoll tutorial*, YouTube tutoriál videó, https://www.youtube.com/watch?v=kux48zqUQY8 (2019. július 1.)

[30] Firemind: *UNITY TUTORIAL – SIMPLE RAGDOLL SETUP*, YouTube tutoriál videó, https://www.youtube.com/watch?v=UeUgfA6ZWNs (2018. február 18.)

[31] Vjanomolee: *How to apply MoCap Animation to your 3D character in Unity! – Live Stream*, YouTube tutoriál videó, https://www.youtube.com/watch?v=nGdCPpWyjM8 (2016. szeptember 6.)

[32] Childre'sTechTutorials: *Unity Transform Essentials*, YouTube-on elérhető videósorozata, https://www.youtube.com/watch?v=UST0SwYGwjs&list=PLdE8ESr9Th_s2iBzpSHck5ICInQWAKRyw (2019)

[33] Unity Technologies: *Unity Documentation – Scripting API - Transform*, https://docs.unity3d.com/ScriptReference/Transform.html (Version: 2019.2)

[34] Wikipedia: *Virtual Reality*, https://en.wikipedia.org/wiki/Virtual_reality (revision ENG site: 06:05, 9 December 2019; HUN site: 2019. május 7., 22:01)

[35] Kristin Dragos: *Tutorial: Get Setup for Google Cardboard with Unity*, YouTube tutoriál videó, https://www.youtube.com/watch?v=Rxln-dshJJk (2018. január 25.)

[36] Zenva: *Unity VR Game Development Course – Google Cardboard Android*, YouTube tutoriál videó, https://www.youtube.com/watch?v=G1jR6q59iLg (2017. július 12.)

[37] Google VR: *Quickstart for Google VR SDK for Unity with Android*, https://developers.google.com/vr/develop/unity/get-started-android (2019)

[38] Google VR: *GVR SDK for Unity v1.200.1*, GitHub open repository https://github.com/googlevr/gvr-unity-sdk/releases (release version: v1.200.1, 2019. július 17.)

[39] Xlaugts: *Unity VR Tutorial – Button Gaze Timer Interaction*, YouTube tutoriál videó, https://www.youtube.com/watch?v=zdNBZsJdg9c (2019. január 17.)