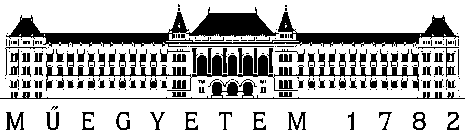
FELADATKIÍRÁS

Az elektronikusan beadott változatban ez az oldal törlendő. A nyomtatott változatban ennek az oldalnak a helyére a diplomaterv portálról letöltött, jóváhagyott feladatkiírást kell befűzni.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Müller Gergő

Biofeedback és VR vezérlésű RPG játék készítése Unity és

Daydream környezeten

Konzulens

Dr. Forstner Bertalan

BUDAPEST, 2017

Tartalomjegyzék – Végén Generált.

Összefoglaló

Abstract

Hallgatói nyilatkozat

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

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

Kelt: Budapest, 2017. 11. 20

...…………………………………………….

Müller Gergő

# Bevezetés

## Motiváció

Az okostelefonok, szerves részévé váltak az életünknek az elmúlt években. Majdnem mindenki kényelmesen, otthonosan tudja kezelni őket, akár már egész fiatal kortól is, így kézenfekvővé vált, hogy a mindennapi felhasználáson kívül egyéb, akár tudományos területeken is hasznát vegyük sokrétű funkcionalitásuknak.

Modern okostelefonok olyan érzékenységű szenzorokkal vannak ellátva, amik tovább növelhetik felhasználhatóságukat különféle mérésekhez, nem is beszélve a sokrétű külső eszközről és szenzorról, amikhez különféle technológiákkal (Wifi, Bluetooth Stb…) kapcsolódni tudnak és így tovább szélesíteni a lehetőségek tárházát.

A pedagógia és pszichológia területén is megjelentek a különböző okos készülékeket használó megoldások, mind egyszerű mérések, mind összetettebb kezelések és készségfejlesztő feladatok esetén.

A virtuális valóság (Virtual Reality – VR) szemüvegek elterjedése az elmúlt pár évben új kapukat nyithat érdekes és az eddigieknél hatékonyabb megoldások felé. Ezen technológia segítségével az alany egy általunk kitalált térbe és helyzetbe helyezhető, így a méréshez, vizsgálathoz vagy feladathoz ideálisabb, hatékonyabb környezetet teremthetünk.

## A Virtuális valóság

A VR-szemüvegeknek egy teljesen új, az eddigieknél sokkal magával ragadóbb élményt mutatott be a szórakoztatóelektronika világának. A technológia lényege röviden, hogy úgy jeleníti meg a tartalmat, hogy azt egy sztereó látásrendszeren át nézve a valóság élményét kelti. A tény, hogy így részévé válhatunk a háromdimenziós, világnak jobban, mint eddig valaha lehetővé teszi, hogy a felhasználóból olyan érzelmeket, reakciókat váltsunk kis, amik nagyon hasonlítanak azokra, amiket egy valós szituációban produkálna.

Ennek a technológiának két különböző típusa van használatban.

* Az egyikbe az olyan eszközök tartoznak, melyek saját beépített kijelzővel rendelkeznek, de ezek csak a megjelenítésért felelősek, a számítások, a logika egy külső eszközön fut (pl.: asztali számítógép vagy laptop) és ehhez van – általában vezetékesen – csatlakoztatva a szemüveg.
* A másik csoportba olyan szemüvegek tartoznak, melyek saját kijelzővel nem rendelkeznek, hanem egy okostelefont kell helyezni beléjük, ami biztosítja mind a megjelenítést mind a számítási teljesítményt, amit az adott szoftver igényel.

Az utóbbi megoldás előnye, hogy azáltal, hogy a telefon biztosítja a számítási teljesítményt, így teljesen vezeték és késleltetés nélküli a megjelenítés, nagyobb mozgásszabadságot adva a viselőnek. Egy másik kedvező mellékhatása az okostelefont használó megoldásnak, hogy mivel a szemüveg nem rendelkezik saját kijelzővel, csak két lencsével, ezért árban is jóval megengedhetőbb. Ami az előnye, az részben a hátránya is ennek a megoldásnak: A telefon. Míg az első esetben a számítógépek teljesítményét használhatjuk, addig a másodikban az okostelefonok korlátozott erősforrásaira vagyunk utalva.

## A feladat

A feladat egy olyan játék alkalmazás készítése, mely a virtuális valóság nyújtotta lehetőségek használata mellett a játékos mentális állapotát is folyamatosan monitorozza és adatokat rögzít róla, a rögzített adatokat pedig különféleképpen felhasználja.

A virtuális valóságot a feladatban a Google DayDream néven futó VR szemüvege biztosítja, mely egy Android operációs rendszert futtató „DayDream-ready” telefont vár el. Ez egy a piacon található más szemüvegeknél könnyebb, jóval kisebb és kényelmesebb konstrukció, ami egyszerűbb felhasználást tesz lehetővé. A Google szemüvegéhez - a saját kategóriáján belül elsőként - egy vezetéknélküli kontroller is tartozik, ami lehetővé teszi, hogy nem csak, hogy részesei lehetünk a játék világának, de egyszerűen, már ismert gesztusokkal kapcsolatba is léphetünk vele, irányíthatjuk.

Ami a mentális állapot megfigyelését illeti, egy úgynevezett neuroheadset segítségével biztosítjuk. Ezt az eszközt úgy kell elképzelni, mint egy kicsit szokatlan fejpántot. A benne található szenzorok folyamatosan monitorozzák a viselője agyi aktivitását, melyből kinyert származtatott adatokat utána továbbküldi a hozzá vezeték nélkül csatlakoztatott okostelefonnak, így azok mi alkalmazásunk számára is elérhetővé válnak.

## A célom

A projekt célja egy olyan alkalmazás megvalósítása mely egy pszichológiai tesztet – a feladat esetében az úgynevezett Frostig teszt – új, innovatív kontextusba foglal a VR és a neuroheadset segítségével.

A különféle pszichológiai kísérletek, mérések, de főleg a készségfejlesztő gyakorlatok, ahol nagyon hasonló feladatokat kell megoldani repetatív módon rövid idő után unalmassá válhatnak. Az ilyen feladatok során, ha az alany figyelmét veszti és már nem koncentrál a feladatra a mért eredmények sem lesznek relevánsak, mert nem azt fogják mutatni, ami az alany legjobb tudása. A cél az lenne, hogy mérést, feladatot végző személy figyelmét és érdeklődését a feladat teljes hossza alatt fenttartsuk. Ennek jó eszköze lehet, ha a megoldandó feladatokat egy játék menetébe foglaljuk bele. Ha ez a játék elég izgalmas és interaktív, akkor a játékos (alany) nem veszti figyelmét és könnyedén használható adatokhoz juthatunk róla.

A Frostig tesztek egy a vizuális percepciós teszthalmaz mely szem kéz koordináció javítására és mérésére alkalmas. Azért esett erre a tesztre a választásom, mert ez olyan készségek használatát és javítását vonja maga után melyek szorosan kapcsolódnak a háromdimenziós érzékeléshez, így erre a területre nagy hatással lehet a megoldásom.

Ezeket a teszteket gyerekeken végzik (általában 4-7 éves kor között), így, ha ezt egy játék keretein belül lehetne megtenni, akkor mindkét fél számára kedvezőbb kimenethez juthatunk. Nem is igazán egy egyszeri mérés esetén látom ennek nagy jelentőségét, hanem a készségfejlesztő gyakorlatok végzésénél, ahol egy feladat többszöri végrehajtása könnyedén unalmassá válhat és ezáltal a pozitív hatása pedig gyengül.

## A dolgozatról

A továbbiakban a dolgozat során részletesebben ismertetni fogom az ebben a fejezetben megemlített technológiák működését és pontos felhasználását a feladatban. Továbbá egyéb a fejlesztés során használt technológiákat is bemutatok. A Frostig teszt is részletesebb bemutatásra kerül majd és tovább pontosítom, hogy annak melyik részhalmaza került megvalósításra a játékban,

Ezután a végzett munkára, menetére, egyes állomásaira, a munka közben felmerült nehézségekre kerül a hangsúly majd részletesen ismertetem a kész szoftver részeit és megoldásait, végül a mért eredményekről lesz szó.

# Irodalomkutatás és technológiák

Ebben a fejezetben ismertetni fogom a munka során felhasznált technológiákat, illetve, illetve indoklom kiválasztásukat, továbbá pontosításra kerül a Frostig tesztek felhasználása is.

## A VR technológia

A technológia fő feladata, hogy valamilyen eszköz segítségével háromdimenziós virtuális világot vetítsen a felhasználó köré, melyben nézelődhet, mozoghat vagy akár különféle interakciókat is végezhet vele.

A virtuális valóság szemüvegek és különféle kontrollerek ( akár bionikus kéz is) már a 90-es évek közepén megjelentek. Nem a szórakoztató elektronika területe volt az egyetlen ahol felütötte a fejét a VR használata. A hadsereg pilóták képzésére, mérnökök a háromdimenziós tervezést igénylő feladatok megkönnyítésére, de még művészek is nyitottak a technológia felé. A legnagyobb problémát ekkor még az okozta, hogy a mindennapi felhasználó számára történő tömeggyártáshoz túlságosan is drága volt a technológia.

Az ember számára a háromdimenziós érzékelést a binokuláris látás biztosítja. Ez annyit tesz, hogy a két szem vízszintesen különböző pozícióban helyezkedik el így két enyhén különböző kép vetül a két retinára. Ennek a két képnek az összekombinálásával alkotja meg az agy az általunk észlelt térbeli képet. Ezt a jelenséget sztereopszisnak nevezik. Az agy több különböző információ alapján következtet az objektumok térbeli helyzetére. Ilyen például a tárgyak fedése, színe, homályossága, ismert tárgyak mérete és a két enyhén eltérő kép. Az utolsót leszámítva ezeket kétdimenziós képekből is ki tudjuk nyerni, de azok mégsem tűnnek térbelinek.



Sztereoszkópikus képpár 2.1

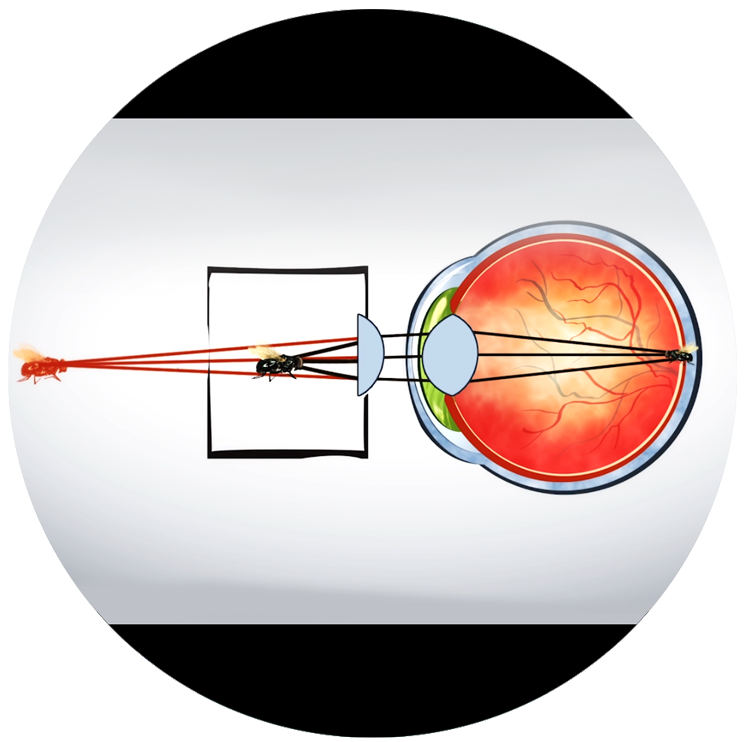


A két kép összekombinálása, az egyik pirosra, a másik kékre színezve 2.2

A két eltérő kétdimenziós kép érzékelésével történő térbeli észlelés mesterségesen is előidézhető, ha két szemnek két enyhén eltérő képet mutatunk, oly módon, ahogy az a binokuláris látás segítségével történne. Ezt a technikát sztereoszkópiának nevezik, és pontosan ezt használja ki a VR is. A szemüvegek kijelzője, vagy esetekben a belehelyezett okostelefoné viszonylag közel pár centire helyezkedik el a szemtől, így segítve azt, hogy az előállított két kép közül az egyiket csak az egyik, a másikat csak a másik szem láthassa. Így mesterségesen reprodukálva a binokuláris érzékelést és előidézve a háromdimenziós kép kialakulását, észlelést.

VR android kepernyo screenshot a jatekrol

Az térbeli látás egyik legfontosabb eszköze, hogy az emberi szem tud mind a távoli mind a közeli dolgokra fókuszálni, így mindkét esetben éles képet tud alkotni. Ezt a szemlencse és a lencsefeszítő izmok teszik lehetővé. Az éles látás feltétele, hogy a szembe beérkező fénysugarak a szemlencsén megtörve a szem hátsó részén található retinán metszék egymást. A különböző távolságból érkező fénysugarak más szögben esnek a lencsére, így azon megtörve máshol metszik egymást. Amikor távolra fókuszálunk a lencsefeszítő izmok megfeszülnek, és a lencse domborulata csökken, ha közelre, akkor pedig elernyednek és a lencse görbülete nő. A szemlencse ezen változása teszi lehetővé, hogy különböző távolságokra is élesen lássunk, ám ha egy tárgy túl közel van, akkor a lencse nem tudja korrigálni a beérkező sugarakat, hogy azok a retinán metsszék egymást, így mögötte fogják, és a kép homályos lesz. Ez történne a szemtől pár centire elhelyezett kijelzővel is, ha nem lenne a VR szemüvegek egy másik közös jellemzője, a két lencse, melyeken keresztül a szeteroszkóp képet nézzük. Ezen lencsék feladat, hogy beérkező fénysugarakat úgy törjék meg, hogy azok kisebb szöget zárjanak be a szemlencse síkjával, és így lehetővé téve az éles látást. Ennek egy másik hatása, hogy a szem távolabbinak érzékeli, látja az adott alakzatot.



3 – A VR szemüvegek lencséinek szerepe a képképzésben

A VR szemüvegek és kontrollerek mind olyan technológiákat használnak, melyeket a mobil illetve okostelefonokra fejlesztettek ki. A kicsi, de nagyfelbontású képernyő, ami az éles valósághoz közelítő kép megjelenítését biztosítja, a giroszkóp, ami a fej és a kontroller helyzetének változását detektálja a fejmozgás követése érdekében, illetve kicsi, energiatakarékos, de mégis hatékony processzorok, amik a számítási teljesítményt biztosítják. Ezen technológiák hatékony gyártásának elterjedése vezetett a VR elterjedéséhez.

### Google DayDream

A Google második VR platformja és eszköze a DayDream 2016 novemberében jelent meg. Elődje a Cardboard egy könnyű olcsó eszköz, ami a VR népszerűsítésére volt rendeltetett. Míg a Cardboardnál az alkalmazások tartalmazták a szükséges szoftvert, addig a DayDream esetén az Android operációsrendszer. Ebből kifolyólag csak az Android 7.1-et (Nougat) vagy annál újabb operációs rendszert futtató okostelefonok lehetnek képesek a használatára. De ez nem minden. Ahhoz, hogy egy telefon alkalmas legyen a DayDream használatára a „DayDream-ready” besorolást kell kapnia, ami mind hardver, mind szoftver oldalról vizsgálja a telefon képességeit. Ilyen kritériumok például a képernyő mérete (4.6” és 6.0” között) és felbontása (minimum Full HD, 1080p, de a 1440p a javasolt), vagy a minimum képernyő frissítési ráta (60Hz) és a legalább két processzor mag. Ezeken kívül még megköti a különböző megjelenítési késleltetések felső határát és különböző API-k támogatását ( pl.: Vulkan).

A Google szemüvege az eddig megjelent eszközökhöz képest kisebb és kompaktabb. Az eddig megszokott öntött műanyag házat egy teljesen új puha, könnyű szövet borításra cserélte, ami kényelmesebbé tette a viselését. Egy másik előnye a piacon lévő vetélytársaival szemben, hogy nem csak a Google saját telefonjait támogatja, hanem az összes cégét, aki csatlakozott a DayDream programhoz (pl.: Huawei, Motorola, LG, Sasmung).



4 – A Google VR szemüvege a DayDream

Egy vezetéknélküli kontroller is tartozik a szemüveghez, a telefonos VR eszközök között először. Ez új kapukat és lehetőségeket nyit a virtuális világgal való kommunikációhoz. Az irányítást és különböző gesztusok felismerését a beépített giroszkóp, az elején található érintő felület és az alatta található két gomb teszi lehetővé.

## Játékmotor

A modern játékok rengeteg funkciót vesznek igénybe, mint például a háromdimenziós kép kirenderelése, megjelenítése, hangok lejátszása vagy a fizika megvalósítása. Ezek mind olyan funkciók, amiket majdnem minden játék használ, így kézenfekvővé vált, hogy egy egységbe szervezve, újrafelhasználhatóan elérhetővé tegyék ezeket a komponenseket ( és még egyebeket is). Ezek a keretrendszerek a játékmotorok. Ezek több külön motort foglalnak magukba, amik alacsonyabb szintű Alkalmazás programozói interfészekre (Application Programming Interface, API) építenek (pl.: Direct3D, OpenGl, WebGl stb…), ilyen motorok például a megjelenítő motor, fizikai motor vagy az audió motor. A játékmotorok így egyszerűbbé és egységesebbé teszik a játékfejlesztés folyamatát.

A fejlesztés elkezdéséhez játékmotort kellett választanom. Napjainkban a két legelterjedteb eszköz a Unity Engine és az Unreal Engine 4. Ez a két motor között kellett meghoznom a döntést. Mindkét játékmotornak megvannak a saját erősségei és gyengeségei, de mindkét termék kiemelkedő erejű és minőségű szoftver, így a fő szempont az volt, hogy melyik termék illik legjobban az én projektemhez.

### Grafika

Az Unreal Egine grafikus teljesítménye mindig is a Unity-é előtt járt, így élethűbb világot lehet vele teremteni. Ez az én esetmben nem nyom sokat a latba, mert a DayDream esetén a bele helyezett okostelefon biztosítja a számítási kapacitást, és így korlátozottak az erőforrások, ami lekorlátozza a megjelenített kép részletességét. Ezért van, hogy a legtöbb telefonos VR alkalmazás úgynevezett „Low Poly” objektumokat használ, ami annyit tesz, hogy az alakzatokat határoló háló polygonokból épül fel, így felgyorsítva a raszterizáció folyamatát.

### Támogatott platformok

Habár mindkét eszköz támogatja az Android platformot és azon belül a DayDream-et is, mégis a Unity a standard, legszéleskörűbben használt és legtámogatottabb játékmotor mobil platformokra. A Google már a megjelenés előtt együtt dolgozott mindkét céggel annak érdekében, hogy a szemüveg debütálásakor mindkét játékmotor biztosítson API-t a fejlesztéshez.

### Támogatott programozási nyelvek

Az Unreal Engine kizárólag a C++ nyelvet támogatja, míg a Unity a C# illetve egészen az tavaly kiadott 5-ös verzióig a Boo nyelvet és az idén augusztusban megjelent 2017.1-es verzióig a UnityScript-et is, amik már elavultnak (deprecated-nek) lettek nyilvánítva.

### Dizájn vs. Programozás

Az egyik legnagyobb faktor a döntésben az volt, hogy az Unreal Engine sokkal inkább dizájner barát mintsem programozó. Egy úgynevezett „blueprint” technológiát használ, ami lehetővé teszi, csomópontok és köztük lévő kapcsolatok grafikus tervezésével generálható a kód, így lehetővé téve, hogy, ha helyenként korlátozottan is, de tényleges kód írása nélkül is készíthető legyen játék. Továbbá a Unity programozási API-ja jobban dokumentált és széleskörűbb szupport érhető el hozzá a Unity saját, illetve egyéb külső fórumokon.

### Döntés

Ezen szempontok figyelembevételével főleg a C# nyelv használata, a programozói hozzáállás nyomatékosabb támogatása és a Google VR-t támogató átlátható, jól dokumentált API miatt a Unity Engin-re esett a választásom.

## A Frostig tesztek

### Használt szubszet

## NeuroSky neuroheadset

### Eszközválaztás

Az alkalmazás egyik fő komponense a felhasználó pszichés állapotának megfigyelése, rögzítése és felhasználása. Erre a feladatra a NeuroSky MindWave eszközét választottam, mert ez egy kicsi, könnyen és gyorsan használható szárazelektródás eszköz, ami más bonyolultabb headset-ekkel ellentétben nem zavarja a VR szemüveg használatát, kényelmesen elfér egymás mellett a kettő. Továbbá ez az eszköz a tanszéken is rendelkezésre áll, így tényleges alanyokon is végezhető mérés. Ez az eszköz nem csak a pszichológiai mérésekhez lett tervezve, hanem például a szórakoztató elektronika egyes területeire is, így ez egy egyszerűen használható könnyű kis eszköz.



### ThinkGear

Itt jön képbe a Neurosky technológiája és fő terméke a ThinkGear. Ez magába foglal egy könnyen és gyorsan használható szárazelektródás szenzort és egy beépített chippet. A ThinkGear a cég minden eszközében megtalálható és sok külső cégnek is szállítják. A homlokra helyezett szárazelektróda a homloklebeny aktivitását méri. A mért hullámokat a chip dolgozza fel és biztosít belőlük származtatott adatokat, mint például a koncentrációnak vagy a nyugodtságnak a mértéke. Ezek az adatokat ”eSense” adatoknak nevezi és egy egytől százig terjedő számmal reprezentálja. ( A nyers adatok is elkérhetőek az eszköztől, ha másfajta, vagy részletesebb feldolgozást szeretnénk rajtuk végezni.)

A ThinkGear chipjének nem csak pusztán a bejövő agyhullámok feldolgozása a feladata, hanem tartalmaz egy bonyolult algoritmust a bejövő adatok zajmentesítésére is. Ahogy az agyi neuronok elektromos tevékenysége, úgy más a környezetben található elektromos eszköz is kibocsát hullámokat. Ezek a hullámok torzítják a mért adatokat így azok hitelességüket vesztik. A zajmentesítő algoritmus ezt a hatást rendeltetett ellensúlyozni.

# Tervezés

Egy klasszikus full-stackes háromrétegű alkalmazás fejlesztése esetén a tervezés folyamata során olyan feladatokkal találkozunk, mint az adatbázis séma megtervezése, a rétegek elválasztása és a köztük történő kommunikáció megtervezése, vagy a szolgáltatást végző osztályok megtervezése. Ez a projekt során nem csak a játékfejlesztés, a virtuális valóság és a neuroheadset volt újdonság, hanem egy ilyen alkalmazás tervezésének metodikája is.

## A játékmenet

Az első feladat a játék stílusának és témájának meghatározása volt. Miután kicsit körülnéztem az elérhető daydream-es alkalmazások között, és azt tapasztaltam, hogy a legtöbb alkalmazásban nem adják meg a játékosnak azt a lehetőséget, hogy szabadon mozogjanak világban, vagy kötött pályán mozog vagy egyáltalán nem is mozog. Habár körültekintés lehetőségét megadják, de sokszor mégis úgy éreztem, hogy egy olyan lehetőséget veszítenek el ezek az alkalmazások, ami sokkal jobban be tudja vonni a játékost a virtuális világba. Mind az ilyen jellegű alkalmazások hiánya, mind a plusz hozzáadott élmény, ami abból származik, hogy a játékos saját maga szabadon fedezheti fel az általunk kreált világot, miatt úgy döntöttem, hogy egy kis Open World RPG játékot fogok készíteni, melyben habár szabadon mozoghat a játékos mégis egy lineáris küldetéssorozatot követ végig.

Egy másik lényeges aspektusa a témának, amit mindenképp figyelembe kellett vennem, az hogy a játék gyerekeknek készül így mind témában mind megjelenésben meg kellett ennek felelni. Ezért döntöttem úgy, hogy egy fantázia világban fog helyet kapni a történet egy klasszikus mesebeli történet menettel, ahol egy az elején megismert fő gonoszon kell fölülkerekedni, sok munka és egy segítő jó karakter segítségével.

### A történet

A történet egy kis mesebeli erdőben játszódik. Az elején az egyetlen karakter rajtunk kívül egy házikó előtt álldogáló alak. Egy Ogre. Miután odamegyünk hozzá kiderül, hogy a segítségünkre van szüksége, mert elszöktek a macskái és kell valaki, aki megkeresi és visszahozza őket az erdő mélyéről.

Miután elfogadtuk a küldetést (kötelező) elindulunk a macskák felkutatására. Ha a nyomukra akadtunk, már csak el kell kapni őket, de ez nem olyan egyszerű, mert minduntalan elszaladnak ügyesen cikázva a fák között. Mindegyik cica egy az erdő végében rejlő kis gombaházhoz vezetett minket.

Mikor a cicák eltűntek a házikó ajtaja mögött egy öregasszony lép ki belőle és számokéri tőlünk, hogy miért kergetjük a macskáit. Hamar rájön, hogy az Ogre vert át minket, aki csak a vacsorájához tartotta az állatokat. Az öregasszony is a segítségünket kéri, de ő a gonosz Ogre legyőzésében, aki régóta sanyargatja az erdő lakóit. Ám előbb fel kell készülnünk a harcra, ezért elküld minket a varázslónő, hogy keressük meg az erdőben rejlő mágikus ősi rúnákat és azok végigrajzolásával sajátítsuk el a titkukat.

Ha ezzel végeztünk visszatérünk a varázslónőhöz, aki megdicsér minket és elküld, hogy gyakoroljuk a varázslást a rúnák segítségével. Ez úgy zajlik, hogy a megjelenő rúnákat minél kevesebb hibával végig kell rajzolni. Sikeres teljesítéskor most már egy a rúna típusának megfelelő varázsgömb is megjelenik, amit el tudunk dobni utána. Miután ezzel is kész vagyunk ismét visszatérünk az öregasszonyhoz, aki most már úgy ítéli meg hogy készen állunka küzdelemre. Irány az Ogre!

Mikor odaérünk a viskója elé már vár minket, de amikor rájön, hogy milyen szándékkal érkeztünk hatalmas haragra gerjed és megkezdődik a harc. Az Ogre előtt a már begyakorolt rúnák egyesével megjelennek és teljesítésük után megjelenő varázsgömböt az ellenfélre dobva tudjuk megsebezni. Minél kevesebbet hibáztunk rajzolás közben annál több sebzést tudnak bevinni a varázslatok. Ő eközben hordókat dobál felénk bizonyos időközönként, amik elől ki kell térnünk. Ha az élete a fele alá csökken, akkor tovább nehezedik a küzdelem. A megjelenő rúnák most már mozognak is , így nehezítve a végigrajzolásukat. Ha legyőztük az Ogrét vége a játéknak és lehetőséget kapunk az újrakezdésre.

Habár szabadon mozoghatunk a világban a történet mégis lineáris, tehát a csak úgy lehet előre haladni, ha az előző küldetést, feladatot teljesítettük.

## Játék állapotának kezelése

Fontos szempont volt, hogy a játékmenet állapotai között ugrálni lehessen, anélkül, hogy küldetéssorozat megelőző állomásait megcsinálnánk. Erre azért van szükség mert a játékmenet 15-20 perc is lehet, attól függően, hogy mennyi ügyes a játékos. Egy másik ok a demózás lehetősége volt, hogy egy rövid bemutató során ne csak a játék elejét, hanem a végét, a kicsúcsosodását is meg lehessen mutatni. Így lehetőséget biztosítok arra hogy a játék elején választani lehessen, hogy teljes végigjátszást szeretnénk, vagy rögötön a harcra ugrani.

### Meghalás

A játék állapotának vissza ( vagy akár előre) állítása fontos szerepet kap, a végső harc során is, ahol, ha alulmaradunk akkor nem az egész játékot kell újra kezdenünk, hanem a harc elejéről lehet újból próbálkozni.

## Frostig teszt beépítése a játékba

A következő lépés az volt, hogy kitaláljam miként fogom beépíteni a Frostig tesztek rajzolási és vonalkövetési feladatait a játékba. A koncepcióm az volt, hogy a jó, vagy kevés hibával történő végig rajzolást jutalmazzam, hogy a játékmenet szempontjából ne lehessen megkerülhető vagy kihagyható, illetve, hogy a rajzolás egy nagyobb folyamat része legyen, hogy ne váljék unalmassá.

A vonalkövetési feladatokat varázs rúnák formájában integráltam a játékba. Ezeket, ha a kontroller segítségével végig rajzolja a játékos, akkor megszerzi a képességet a varázslásra, majd később a harc közben megjelenő rúnák teljesítésével varázslatot tud szórni az Ogréra. Mivel az Ogre elleni végső küzdelem során már stresszhelyzetben kell teljesítenie az alanynak, ezért előtte lehetőséget akartam biztosítani a gyakorlásra. A játék folyamán, miután tudomást szerzünk az Ogre igaz valójáról, a varázslóasszony elküld minket megtanulni a rúnákat (vizuális percepció vonalkövetési teszteket), és ilyenekor nem is enged tovább a játék, amíg egy bizonyos hibaszám alatt nem teljesítjük azt.

Ez a mechanizmus segítségével mire a tényleges nyomás alatt végzendő teszthez ér (az Ogre elleni harc) addigra nem a háromdimenziós virtuális térben való rajzolás újdonsága és technikájának elsajátítása fog dominálni, hanem az alany tényleges képessége a területen.

## A DayDream nyújtotta lehetőségek

Habár a daydream kontroller gombjai és érintő felülete lehetőséget biztosít a számítógépen vagy konzolon megszokott gombok és billentyűk általi irányításhoz, én mégis szerettem volna, ahol csak tudom kihasználni a DayDream nyújtotta lehetőségeket, hogy így tegyem még valósabbá és különlegesebbé az élményt.

### Mozgás

Az első és egyik legfontosabb feladat a szabad mozgás vezérlése volt. Szerettem volna itt is kicsit elrugaszkodni a megszokott „megnyomok egy gombot és elindulok” módszertől.

Úgy döntöttem, hogy a mozgást a fej dőlésszögéhez fogom kötni az irányt pedig a fej forgatása határozza meg, tehát, ha enyhén lefele dönti a fejét a játékos, akkor elindul, ha még lejjebb néz, akkor tovább gyorsul a mozgása, de ha a vízszintes fölé fordítja a fejét kicsivel, akkor megáll. Ez a megoldás könnyen tanulhatónak és könnyen használhatónak bizonyult és nem igényli, hogy egy amúgy is új környezet és eszköz mellett még különböző gombok működését is meg kelljen jegyezni, hiszen ne felejtsük el a játékot és egyben az ebbe beépített mérést valószínűleg olyan alanyok, gyerekek fogják használni, akik még nem találkoztak vele.

Egy másik alternatív megoldás lett volna, hogy a kontroller elején található érintő felület segítségével irányíthat a játékos, de úgy éreztem, hogy az első megoldás valósabb érzést kelthet a játékosban.

### Rajzolás

A rajzolás a Frostig teszt lelke, így erre külön hangsúlyt kellett fektetnem. A játékban a rajzolás a rúnák teljesítéséhez kell, ahol a pályán, illetve a végén a harc közben megjelenő alakzatokon kell a daydream kontroller mutatóját végig húzni. Így a térben a kezünkben tartott kontrollerrel azt a levegőben mozgatva rajzolhatunk a játékban is. Rajzolás közben szikrázás jelzi a mutató helyzetét. Nem csak a rajzolás élményére kell összpontosítani, hanem a közben elkövetett hibákra is. Az alakzat felületéről való letérésre vagy egy nagyobb szakasz átugrására, mert ezeket az adatokat kell a gyakorlási fázisnál felhasználni arra, hogy a rúna teljesítve lett-e, illetve a harcnál a bevitt sebzés meghatározására is.

### Varázslás

A varázslás egy rúna sikeres végigrajzolása után megjelenő varázsgolyó eldobását jelenti. Itt szerettem volna megint kicsit eltérni a megszokott „megnyomok egy gombot és ettől lövök” iránytól. A célom az volt, hogy a játékba a varázsgömb eldobása egy tényleges dobómozdulat hatására történjen, hogy a játékos még jobban beleélhesse magát a virtuális valóságba. Ehhez a daydream kontrollerbe beépített giroszkópot használom fel. Ennek aktuális szögsebessége kérdezhető le adott tengely körül, a Unity DayDream API-ján keresztül.

### Elugrás

A végső harc közben az Ogre adott időközönként egy hordót hajít a játékos felé (akárhol is áll az), akinek ki kell térni előle. Az elugrás úgy működik, hogy a harc közben, a szabad mozgás megszűnik és a játékos három különböző pozícióban tartózkodhat. Ennek ok az, hogy a megjelenő rúnák rajzolásához ideális pozícióban maradjon mindig, és ne kelljen a rajzolás, a hordók elkerülése és a varázsgömbök eldobása mellett még a megfelelő pozicionálásra is figyelni, mert ez már túl sok figyelmet vonna el az alanytól és a teszt eredménye látná kárát.

A három pozíció közti mozgást a kontroller elején található érintő felület segítségével oldom meg, mégpedig úgy, hogy azon jobbra, ballra „swipe-olva” tehát az ujjunkat jobbra, balra húzva tudunk odébb ugrani. Ez azért kihívás, mert a kontroller API-ja nem kezel ilyen eseményt, így ezt saját magamnak kell implementálni. Egy másik kritérium, hogy a három helyzet közül a középső egy fix pont legyen, mégpedig az, ahol elkezdtük a harcot, és innen lehessen odébb csúszni az játékos aktuális helyzetéhez képest jobbra vagy balra. Ha az egyik irányba elugrik a játékos, akkor onnan már nem léphet még egyet ugyanarra, csak vissza középre. Ezzel védem ki azt, hogy a sok elugrálás során nagyon elkeveredjen a játékos a harc színteréről.

## A MindWave headset beépítése a játékba

A neuroheadset a beépített szenzorok és komplex feldolgozó és zajszűrő algoritmusa segítségével, lehetőséget nyújt nekünk abban, hogy a bonyolult agyhullámok helyett már könnyen kezelhető származtatott értékekkel dolgozhassunk.

A feladat ezen értékek felhasználása és értelmezése a játék folyamán. Az alkalmazás során két fő hely van, ahol a MindWave szolgáltatta adatok felhasználásra kerülnek.

### AdaptED keretrendszer

A feladat során felmerült, hogy dolgozhatok egy a tanszék által fejlesztett keretrendszer az AdapatED bevonásával. Ez egy androidos keretrendszer, ami különböző fiziológiai jellemzőket mérő eszközök csatlakoztatását, monitorozását, és a mért adatok megjelenítéséért felelős. Tartozik hozzá egy webes felület is, ahol felhasználókhoz és játékokhoz kötötten tekinthetők meg a mért adatok, amikből grafikont is rajzol, így tovább könnyítve a mérést végző személy számára a feldolgozást. A grafikonon saját a események is megjeleníthetők. Ezek az android alkalmazásban létrehozott esemény osztályok, amiket a megfelelő metódussal felküldünk az adapted szerver felé.

Mivel ez egy androidos keretrendszer így meg kellett oldanom a Unity-ben buildelt alkalmazásom integrációját. Ez több csapdát is rejtett.

Az első probléma, az volt, hogy, ha a keretrendszer csatlakozik egy addot eszközhöz pl.: neuroheadset, akkor a Unity-ben készült alkalmazásból a C# kódból már nem lehet csatlakozni. Ennek az az ára, hogy a Unity által biztosított egyszerű C# API helyett, egy viszonylag bonyolult esemény vezérelt módon kellett megoldani a játékom, a MindWave és az AdaptED kommunikációját. Ez magával von olyan dolgokat is, hogy meg kell oldani a Java nyelven írt android alkalmazás és keretrendszer, illetve a C#-ban írt játék kommunikációját, hogy bizonyos események hatására mindkét irányba át tudjanak hívni egymásba.

A másik probléma a tényleges integráció. Szerencsére mind a keretrendszer, mind a Unity fel van készítve hasonló feladatra. A játék kiexportálható Unity-ből egy Android Studio projekt formájában, ami később egy új modulként importálható az AdaptED projektjébe.

### Játék módosítása

Az egyik ilyen hely a játék végén az Ogre elleni harc, ahol bizonyos időközönként felénk dob egy hordót, ami elől el kell ugrani. A headsettől kapott adatok közül itt a játékos nyugodtságát használom fel, mégpedig oly módon, hogy, ha túl nyugodt az alany, akkor a hordódobások gyakoribbá válnak, ha túl ideges, akkor pedig ritkulnak. Ennek a célja, hogy egy állandó stresszhelyzetben legyen tartva az alany, hogy nyomás alatt vizsgálhassam a teljesítményét.

### Statisztika készítése

A másik felhasználás, egy játék végi statisztika készítése és felküldése a keretrendszer felé, ahol ez megtekinthető egy esemény formájában ( EndGameStatistics). A statisztika a rajzolás közben mért adatokra összpontosít, mivel ez valósítja meg a Frostig tesztet, ez a mérés lényege. Terveztem még a játékos átlagos nyugalmának és koncentrációjának a bevételét a statisztikába, de mivel ezek az értékek elég széles spektrumon ugrálnak rövid idő alatt is, ezért nem szolgáltattak volna releváns információt az átlagos adatok.

A mérés során rögzítem, az összes rajzolás közben vétett hibát, a hiba pillanatában mért nyugalom és figyelem értékeket, illetve az adott rúna típusát. A játék végén ( amikor az Ogre meghal, vagy, ha a játék szüneteltetésénél megjelenő újrakezdés gombra kattintunk) az odáig összegyűjtött adatokból statisztikát készít, mely tartalmazza:

* A rúnarajzolás során átlagosan elkövetett hibák számát
* A hibázások átlagát rúnatípusokra lebontva, hogy kiszűrhető legyen melyik típusú feladat, melyik alakzat esett nehezére a játékosnak
* A hibázások pillanatában mért figyelem és nyugodtság értékek átlagát (itt azt várom, hogy magasabb értéket kapok, mint ami az AdaptED keretrendszer által rajzol grafikonról átlagosan leolvasható)

# Önálló munka bemutatása

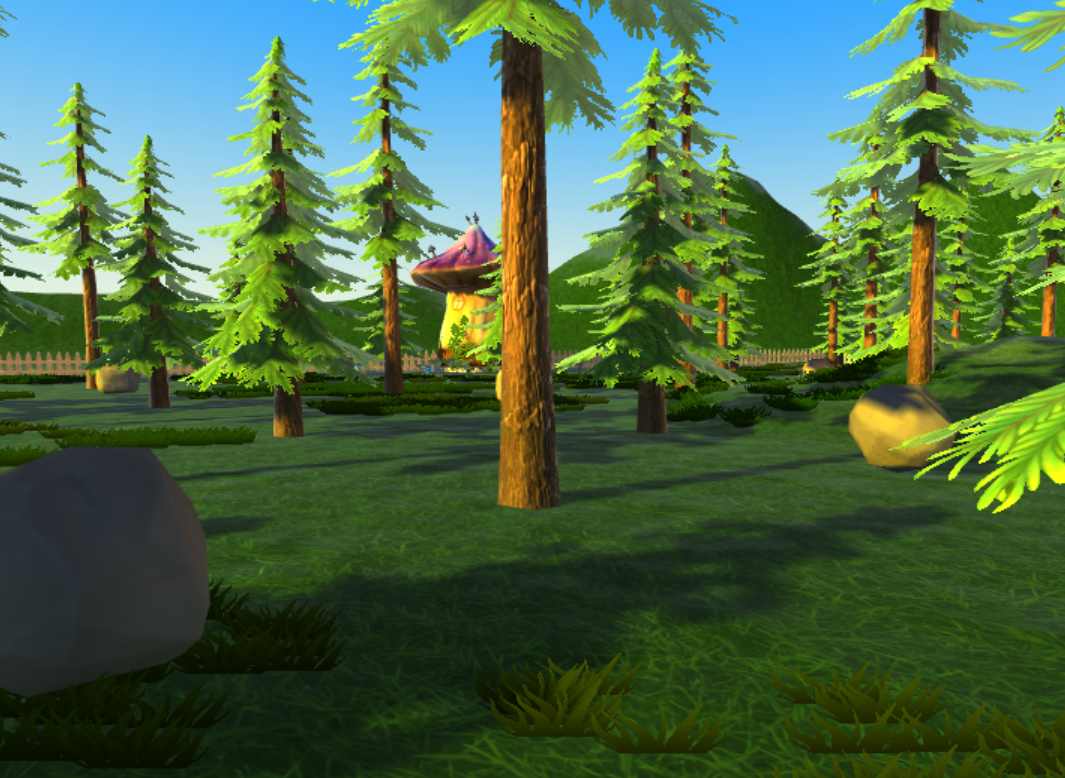
Ebben a fejezetben bemutatom, hogy hogyan is működnek az elkészült szoftver egyes részei, hogy a tervezés fejezetben leírt feladatokat hogyan oldottam meg, illetve, hogy milyen nehézségekkel találkoztam, milyen akadályokba ütköztem, és azokat hogyan oldottam meg.

A fejezet során a különböző elemeket és megoldásokat a játékmenet vonalán mutatom be.

## A virtuális világ megteremtése

A világ megteremtéséhez háromdimenziós objektumokra és szereplőkre volt szükségem. A Unity Asset Store-ja biztosít rengeteg karaktert és pályaelemet, de ezeknek nagy része fizetős. Így először alaposan körül kellett néznem a „piactéren”, hogy miből tudnék építkezni. Mivel az egy téma és stílus köré épülő csomagok mind fizetősek, ezért több külön egységből kellett olyanokat összeválogatnom, amik egymáshoz és a játékhoz is illenek. Egy másik nehezítés volt, hogy nem mindegy milyen sűrűségű az alakzatot leíró hálók (mesh). A telefon számítási kapacitása és más erőforrásai végesek, és nem birkóznak meg a sokszor PC játékokhoz készített textúrákkal, így erre is oda kellett figyelnem amikor a játék hangulatát megteremtő virtuális világ elemeit kerestem.

Az erdő összerakásánál cél volt, hogy a fák egy erdő hatását keltsék, de ugyanakkor ne legyenek túl közel egymáshoz, ahhoz, hogy a játékosnak túl sok energiájába kerüljön a folyamatos forgolódás és kerülgetés.



6 – A mesebeli erdő, ahol a történet játszódik

A karakterek kiválasztásánál fontos szempont volt, hogy olyan szereplőket találjak a játékhoz, amikhez tartoznak animációk is, amiket a játék során különböző eseményekhez vagy állapotokhoz tudok kötni.

Sajnos nem mindegyik szereplőhöz sikerült ilyen modellt találni. Míg a a rohangáló cicához, a végén a játékost követő tigrishez és a cicák gazdájához a varázslónőhöz sikerült megfelelő csomagra bukkanni, addig az ogre esetében ez nem sikerült, pedig a végső harcot egy ütés vagy dobás animáció még élvezetesebbé tehette volna.



7 – Az Ogre a háza előtt

## A GameManager (NPC-k állapota)

A játék állapotát egy központi egység, a játékvezérlő (GameManager) tárolja és irányítja. Ebben a játék aktuális állapotáról minden információ megtalálható ahhoz, hogy meghatározzuk, játék jelenlegi állását.

Ilyen adatok például a küldetéssorozat adott állomását reprezentáló kapcsolók, amik a pályán adott időpontban található karakterekkel történő kommunikáció megfelelő dialógusát, vagy harc állapotát, fázisát irányítják. Továbbá megszabják a rúnák viselkedését, rajzolásuknak mechanizmusát és legfőképpen a rajzolás eredményéről történő visszajelzést, hogy az egy sikerült/nem sikerült üzenet formájában jelenik meg vagy az Ogréra mért sebzés mennyiségeként, illetve, hogy varázsolhatunk-e utána, avagy sem.

Ezek a kapcsolók átbillentésével ugrálhatunk a játék különböző állapotai között, úgy, hogy onnan úgy folytathassuk, mintha teljesítettük volna az azt megelőző küldetéseket.

## A mozgás

Ahogy elindul a játék egy rövid kis útmutató jelenik meg, ami elmagyarázza, hogy lehet mozogni, varázsolni és még egy két dolgot a játékban. „Mozgás: A fej enyhén lefele fordítása kezdi meg a mozgást, ha a controller elején lévő touchpaden tartod az ujjad.”

Az eredeti cél az volt, hogy csakis a fej helyzete határozza meg a mozgás sebességét és irányát. A Google Virtual Reality API (GVR API) hozzáférést biztosít a telefon ( és ezzel egyben a fej) helyzetéhez mindegyik tengely körüli elfordulás formájában ( egy háromdimenziós vektor) fokokban mérve. A mozgás elindítását a fej x-tengely körüli forgatásához kötöttem. Ha a játékos 20°-os szögnél lejjebb dönti a fejét a mozgás megkezdődik az x-z síkon, (a nézési irány függőleges komponensét nem használom, mert különben, ha lefele néz akkor lefele mozogna nem pedig előre) abba az irányba, amerre éppen néz. Ezután a mozgás csak akkor marad abba, ha a játékos a fejét a vízszinteshez képest legalább 10°-kal feljebb fordítja. Ez lehetővé teszi, hogy a mozgás megkezdése után újra egyenesen magunk elé nézve mozoghassunk, ha nem akarunk a lábunk elé nézni, lehetővé téve a mozgás közbeni nézelődést. Mozgás közben a fejet 15°-nál lejjebb fordítva gyorsul a mozgás, így imitálva a futást.

A játékos irányítását a VRPlayerController osztály, azon belül pedig az Update metódus végzi, ami minden képkocka kirenderelésénél lefut, így biztosítva, hogy mindig időben reagál a változásokra.

Sok próbálgatás és optimalizálsás után arra jutottam, hogy a megoldásom ebben a formában bizonyos esetekben limitálja a szabad nézelődést. Például, sokszor okozott problémát, hogy nem szerettem volna elindulni, mozogni csak megállni valami előtt, lenézni rá, hiszen a fej előredöntésére elindul a játékos. Ezt úgy oldottam meg, hogy ez az egész mozgási mechanizmus csak akkor lép érvénybe, ha a játékos az ujját a kontroller elején található érintőfelületen tartja. Nem kell megnyomni, csak hozzá kell érnie, így bármikor megállhat és szabadon nézelődhet anélkül, hogy a fej helyzetére oda kéne figyelnie. Ezt a Unity DayDream API-ja segítségével oldottam meg, A GvrControllerInput kontroller állapotot és eseményeket kezelő osztálytól elérem az IsTouching tulakdonságot, és ettől teszem függővé a további működést.

## Dialógusok

A program elkészítése során már bizonyos elemek készen voltak, mire először kipróbálhattam a tényleges DayDream szemüvegen. A nem irányítható karakterek ( Non-player character, NPC) megközelítésekor felugró dialógusablak is egy ilyen elem volt. Ezt egy „Screen-space canvas” segítségével oldottam meg, ami úgy működik, mintha egy külön önálló réteget rakna a kamera elé, így mindegy hova forgunk az mindig ugyan ott marad a képernyőn. Ilyet majdnem mindegyik játékban láthatunk, különböző információk közlésére (Head-up display, HUD) például: életcsík, töltény számláló vagy különböző képességek aktiváltsága.

Amikor először kipróbáltam szemüveg használatával a dialógusablakok nem jelentek meg egyáltalán. Ez azért van, mert a virtuális világ felépítésekor nem az ezt végző motor nem tudja megfelelően kirenderelni az ilyen típusú canvas-okat. Ez nem csak a DayDream esetében van így, hanem minden VR rendszernél (Pl.: HTC Vive, OculusRift). Így maradt a „world-space canvasok” használata, ami azt jelenti, hogy a dialógusablakokat a háromdimenziós tér részeként kell elhelyezni, pontos koordinátákkal megadni a helyzetüket.

A játékban a két NPC (Ogre és a varázslónő) felett megjelenő dialógusok ugyan azt a prefab-et a DialogeWorldSapce-t használják, csak létrehozáskor más koordinátákat kapnak, hogy hol jelenjenek meg. Az ablakok tartalma, a megjelenített szöveg dinamikusan változik a játék és egy párbeszéd alatt is. A szöveg megváltoztatásáért és a azért, hogy a dialógusablak mindig a játékos felé nézzen a DialogeController felelős, de ennek az osztálynak a metódusait mindig az adott NPC-t vezérlő script (WizzardController és CatOwnerController) hívja, mert az NPC-ékkel való interakció eseményei azok, amik a dialógus állapotát, tartalmát megváltoztatják. Ilyen esemény például, hogy a játékos megközelíti az egyik karaktert, ilyenkor a játék állapotának megfelelő dialógus indul el (ha először találkozunk a varázslónővel más tartalom jelenik meg, mintha a rúnák megkeresése után).

## Az okos macska

Miután először beszéltünk az Ogréval elküld minket, hogy keressük meg és a kapjuk el az elszökött cicáit. Amikor megtaláljuk a cicákat azok menekülni kezdenek előlünk, amíg menedékbe nem érnek a varázslónő gombaházában, vagy biztonságos távolságra nem érnek tőlünk. Először csak sétálva indul el előlünk, de amikor túl közel érünk begyorsulnak és futva menekülnek tovább. Mindeközben az eléjük kerülő akadályokat: fákat, köveket is kerülgetik. A macskák feladata, hogy soha ne lehessen elkapni őket, és így végül elvezessenek a gombaházhoz.

### A játékos kikerülése

Mind a játékos mind a macska rendelkezik egy TriggerCollider-el, ez abban különbözik egy normál collider-től, hogy tényleges fizikai ütközés nem történik, ha egy másik objektummal találkozik, de a találkozás eseményére fel lehet iratkozni az OnTriggerEnter vagy OnTriggerExit listenereken keresztül. A játékost egy nagy gömb veszi körül. Ha a cica találkozik ezzel a collider-el akkor megkezdődik a menekülés. A macska igyekszik mindig úgy eljutni a menedékbe (gombaház, nest), hogy aközben a játékost egy adott sugarú körben elkerüli. Thehát, ha például a menedék és a cica közé állunk, akkor az megkerül minket és csak akkor fordul a hát felé, amikor oda már egyenes útja van.

Ezt a működést koordinátageometria felhasználásával értem el. Amikor a macska az „TriggerEnter” esemény hatására megkezdi a menekülést az Update függvényben (tehát minden kirszámított képkockánál) megnézi, hogy menedékhez húzott egyenes metszi-e a játékos adott sugarú környezetét, illetve, ha igen, akkor, hogy a macska közelebb van-e a házhoz, mint a játékos (tehát a játékos a cica mögött van és úgy kergeti). A metszést úgy számolom ki, hogy megnézem, hogy az egyenes és a játékos pozíciója (a kör középpontja) közti távolság kisebb-e, mint a kör sugara, ha igen, akkor metszi. Ebben az esetben meg kell találnia a legideálisabb utat, úgy, hogy közben megfelelő távolságra maradjon az üldözőjétől. Ha közelebb van a menedékhez, mint a játékos, akkor ez az út az egyenes vonal a ház felé, de, ha nem, akkor a játékos körüli körhöz húzott érintők közül az, amelyik egyenese közelebb van a házhoz. Az irányt az alábbi kódrészlet alapján számítottam ki. Mivel a számításokat végző kódrészlet viszonylag hosszú és komplex ezért csak pszeudó kóddal szemléltetem, de a függvénynevek nevei megegyeznek.

float pointLineDistance(){

return distance;

}

bool isLineIntersectsCircleOrCloserToNest(){

if(cat-nest distance < cat-player ditance)

return false

return pointLineDistance(player , from cat to nest line) < player's avoid range

}

circlesIntersections(cirle1, circle2, out i1, out i2){

calculate intersections

i1 = intersection1

i2 = intersection2

}

getTangentsFromPoint(out t1, out t2, catPosition){

distance = catPostion to playerPosition

playerCircle = (center = player.pos, radius = playerAvoidRange )

catCircle = (center = cat.pos, radius = distance )

circlesIntersections(playerCircle,catCircle,t1,t2)

}

pickCloserTangent(t1,t2){

d1 = t1 , player ditance

d2 = t2 , player distance

return d1 < d2 ? t1 : t2;

}

Update(){

if(!isLineIntersectsCircleOrCloserToNest()){

catDirection = towards the Nest

}

else{

t1, t2

getTangentsFromPoint(t1,t2, selfPos);

dir = pickCloserTangent(t1,t2)

catDirection = dir;

}

}

A számításokat a MathUtil segédosztályom metódusai végzik, amiket a KittenController hív. Mivel ezek a hívások az Update függvényben kaptak helyet ezért inden képkockánál újraszámítja az érintőket, így a folyamatos érintő irányú mozgás körmozgást eredményez a játékos körül.

### Akadályok kikerülése

Minden akadály rendelkezik egy ObstacleInfo scripttel, amiben meg van adva, hogy az adott elemet mekkora sugarú körben kell kikerülni. Az fák é kövek kikerülése nagyon hasonló módon történik, mint a játékosé. A módszer egészen az érintő választásig teljesen megegyezik, itt viszont nem az érintőt választja, amelyik a menedékhez van közelebb, hanem azt, amelyik az őt üldöző játékostól távolabb. Ha ez nem így lenne, akkor az akadály megkerülése közben a cica és a játékos találkozhatnának és elkaphatná a macskát. Ezzel a megoldással, akárhogy kergetjük a cicát a egy fa vagy egy kő körül azt sosem tudjuk elkapni. Egy akadály körüli érintő irányú mozgás akkor ér véget, ha a cica karakter és a gombaház között húzott egyenes már nem metszi az akadály adott sugarú körét, ekkor vagy egyenesen a kunyhó felé folytatja útját, vagy ha a játékos megfelelően közel van, akkor annak kikerülését kezdi meg.

### A sebesség

A cica karakter három különböző sebességgel tud mozogni, attól függően, hogy milyen távolságra van az őt üldöző játékostól. A játékos is két sebességgel tud mozogni, a feje dőlésszögétől függően. A játékos futás közben sebessége lehetővé teszi, hogy megközelítse a macskát, de ilyenkor az is begyorsít, és a maximális sebessége gyorsabb, mint a játékosé így el tud menekülni. Ha a cica elég távol ér a játékostól abbahagyja a futást és lassabb fokozatra kapcsol, majd megáll. Ha megint közel ér a játékos a hajsza újrakezdődik. Ezt a működést a játékos és a macska karakter közötti távolság folyamatos monitorozásával érem el és a KittenController Update függvényében és utána ez alapján állítom be speed tulajdonságot, ami a mozgás sebességét határozza meg.

## Rúnák és rajzolás

A rúnákkal először, akkor találkozunk, amikor a varázslónő elküld minket az erdőbe, a felkutatásukra, hogy felkészüljünk az Ogre elleni harcra.

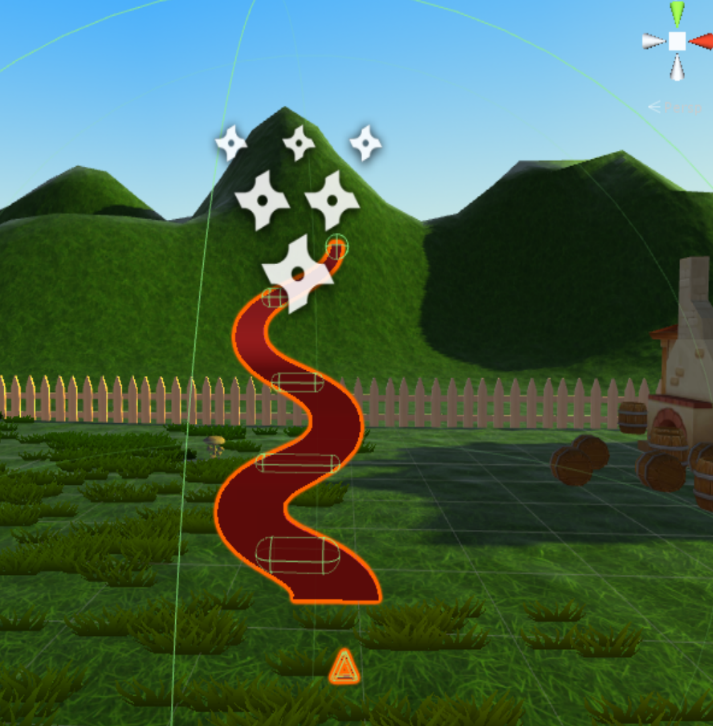
A rúnák funkcionalitásuk alapján kétdimenziósak lennének, de én olyan működést rendelni hozzájuk, amit egy Canvas használatával nem lehetett volna megoldani, így végül háromdimenziós objektumokként lettek megvalósítva.

Ha a DayDream kontroller kurzorját ( fehér kis korong ) a rúnára irányítom, és lenyomom az érintőfelületet, akkor megkezdődik a rajzolás. Ekkor létrejön egy szikra objektum (egy ParticleSystem) ami végig követi a rajzolást.

### Felépítésük

A négy rúna nagyon hasonlóan épül fel, a különbségek csak az alakjukból származnak.

A rúnák a törzsüket adó alakzatból, a rajzolás során felügyelt ellenőrzőpontokból és végpontból állnak. Az ellenőrzőpontok a rúnaalakzat mentén elhelyezett collider-ek, amik a szikrával történő ütközésre reagálnak. A rúna végén található még egy collider, amivel történő ütközés ( tehát, hogy odahúzzuk a kurzort rajzolás közben) jelzi a rajzolás végét és megtörténik az eredmény kiértékelése. A rúnáknak még két eleme van, amik a használatuk intuitívabbá tételéért kerültek be. Az egyik a rajzolás irányát jelző, kezdetben oda-vissza mozgó kis nyilacska, a másik pedig a pozíciójukat jelző nagy fénysugár, ami nem sokkal a rúna fölött kezdődik, és egészen az égbe tart. Ez azt szolgálja, hogy amikor meg kell keresni az erdőben őket, akkor támpontot kapjunk, hogy merre is induljunk. Ez, ha a játékos elég közel ér szép lassan eltűnik, ezt egy TriggerCollider-rel oldottam meg.



Ábra 4.1: Rúna szerkezete

Ábra 4.2 Rúna rajzolás közben

### Működésük

Mindegyik rúnát ugyanaz a script vezérli, a RuneController. A rúna objektumok legfontosabb komponense az EventTrigger, ami lehetővé teszi, hogy a kontroller kurzor különböző eseményeire feliratkozunk. Ilyen események például a pointer belépése vagy kilépése a rúnából, a érintőfelület lenyomása, vagy a „drag” esemény, ami a kontroller mozgatását jelenti úgy, hogy a gomb közben le van nyomva. Ezeknek az eseménykezelőknek a segítségével implementáltam a rajzolást. Ha a kurzort a rúnára visszük (PointerEnter esemény) és megnyomjuk az érintőfelületet (PointerDown esemény), akkor megjelenik a szikra (Sparkl) és megkezdődik a rajzolás. Ezen eseményekre a RuneController osztály pointerEnter és startDrawing metódusait iratkoztattam fel. Az utóbbi az, ami elkéri a mutató pontos helyét a paraméterként kapott általános BaseEventData PointerEventData-vá kasztolásával, és megpéldányosítja a Sparkle objektumot arra a helyre. A PointerDrag eseményre a feliratkoztatott drawTheRune függvény fut le minden alkalommal, amikor rajzolás közben mozgatjuk a kontrollert, ez a startDrawing-hoz hasonlóan elkéri a mutató aktuális helyzetét és ez alapján frissíti a szikra pozícióját. Az érintőfelület elengedésének, vagy a rúnáról való letérés hatására a szikra eltűnik és a rajzolás abbamarad. Az utóbbi esemény a pointerExit metódus kezeli. A rajzolás közben rúnáról való letérés hibának számít és ezt ebben a függvényben kezeljük, egy számláló növelésével. Hiba esetén az AdaptED keretrendszert is értesíteni kell, ami az AdaptEDConnector osztályom sendRuneFaultEvent metódusának meghívásával történik. Ennek a függvénynek két paramétert adok át, a az aktuális hibaszámláló állását és a rúna típusát, hogy ezek a keretrendszer felületén megjelenített grafikonon is látszódjanak.

Az ellenőrzőpontok kezeléséta CheckPointController osztály végzi, ami a Sparkle egy ellenőrzőponton való áthaladásakor növel egy számlálót a RuneController-ben és kikapcsolja a collidert, amin épp áthaladt, hogy azt később már ne számolja újra.

A RuneEnd objektumon áthúzva a kurzort meghívódik a rúnarajzolást kiértékelő evaluateRunePerformance függvény a RuneController osztályban. Ez a metódus, megvizsgálja az ellenőrződön és hibaszámláló állását és ezek alapján eldönti, hogy a rúna rajzolása sikeres vagy sikertelen volt. Az első esetben az onRuneSuccess ,míg a másodikban az onRuneFail metódust hívja meg. Ezeknek a függvényeknek a hatása függ a játék aktuális állapotától ( GameManager kapcsolóitól). Ha a rúnarajzolás abban a fázisban történt amikor még csak az erdőben kellett megkeresni őket, akkor siker esetén egy felirat jelenik meg, ami gratulál a sikeres teljesítéshez, viszont, ha a gyakorlás során rajzoljuk végig megfelelően a rúnát, akkor már egy varázsgömböt kapunk, amit eldobhatunk. Sikertelen végigrajzolás esetén mindkét esetben egy „próbáld újra” felirat jelenik meg, és a rúna állapota (számlálók, ellenőrzőpontok) visszaáll a kezdetibe. A hibákat és kihagyott ellenőrzőpontokat számon tartó számlálókat a végső harc közben a varázsgömb sebzésének kiszámítására használom. Minél több volt a hiba annál kevesebbet sebeznek. A harc közben, már nincs olyan, hogy sikertelen végigrajzolás, Az onRuneFail ezen ága tovább hív az onRuneSucces-be.

A rúna rajzolása közben a szikra vonalat húz maga után, hogy látszódjon a rajzolás útvonala. Ezt a LineDrawer osztály végzi. Ez induláskor létrehoz egy LineRenderer komponenst a rúnán. A PointerDrag eseményre feliratkozva az addPointOnDrag metódusban a mutató aktuális helyzetét hozzáadom egy listához, amiből végül a LineRenderer segítségével vonalat rajzolok. Mivel az esemény a rajzolás megkezdése után(„drag” alatt) minden képkockánál elsül, ezért a vonal mindig az aktuális állapotot mutatja.

A rajzolás során akadt egy kis problémám a PointerDrag eseménnyel, mégpedig az, hogy ez csak akkor lépett működésbe, ha egy bizonyos távolságnál tovább húztam már a mutatót. Ez azt eredményezte, hogy a gomb lenyomásakor létrehozott szikra csak egy idő után kezdte el követni a kurzort. Ezt úgy oldottam meg, hogy a GVR API kontrollert kezelő script-jében (GvrPointerInputModuleImpl) megkerestem a „drag” eseményért felelős kódrészletet és átírtam az esemény első elsüléséhez szükséges küszöb méretét, így megoldódott a probléma.

Ezzel a megoldással csak akkor lehet probléma, ha az API-t frissítem, mert ugyebár az átírt script is frissül, de mivel az is verzió-követve van ezért ez könnyen karbantartható.

## Varázslás

A varázslás a rúna sikeres végigrajzolásakor megjelenő varázsgömb eldobása. A varázslást a FireBallController kezeli. A varázsgolyó elhajítása tényleges dobómozdulatra történik meg. Ezt úgy valósítottam meg, hogy a gömb mozgását a kontroller giroszkópjának állapotától tettem függővé. A giroszkóp a tengelyek körül szögsebességet adja vissza radiánban mérve egy háromdimenziós vektor formájában (GvrControllerInput.Gyro.x) . Azt, hogy csak megfelelő lendülettel, karsebességgel induljon meg a dobás azt az x tengely körül szögsebesség mérésével érem el. Ha a szögsebessége meghaladja a 3.14 radiánt másodpercenként, tehát a fél fordulat per másodpercet bebillentek egy kapcsolót és elmentem ezt az időpillanatot. Ha ez a kapcsoló ilyen állapotban marad elég ideig, tehát ha a kontroller szögsebessége nem megy 3.14 radián per másodperc alá adott ideig, akkor megtörténik a dobás. Ezt az elmentett időpillanat és az aktuális idő különbségéből számolom.

Ha a felül leírt kritériumok teljesülnek akkor a giroszkóp aktuális értékének megfelelő méretű (minél nagyobb a szögsebesség annál nagyobb) kezdeti erőt adok a gömbhöz (akár csak valódi dobásnál).

## A végső harc

A végső harc során részben a már bemutatott elemek jelennek meg például a rúnarajzolás, a varázslás vagy a dialógusok, részben pedig új elemek jelennek meg, mint az Ogre támadása vagy a játékos kitérése, elugrása.

Az ogréhez való visszatéréskor megjelenik egy dialógus, ami felvezeti a harcot. Ezután az Ogre sokszorosára nő és megkezdődik a harc. Ezt a WizzardController enrage corutin-ja végzi. A corutine egy olyan függvény, ami a C#-os yield-return nyelvi elemeket kihasználva, olyan működést valósít meg, hogy megszakítja a függvény végrehajtását és a következő képkocka kiszámításakor folytatja. Erre azért van szükség, mert egy ciklus futása különben egy képkocka alatt történne meg és nem lenne meg a folyamatos hatás. Az enrage függvény egy ciklusban folyamatosan skálázza fel az Ogre méretét.

### A hordódobás

A harc megkezdésekor meghívódik a BarrelSpawner osztály startBarrelThrowing metódusa, ami elindítja a spawnBarrels nevű corutine-t. Ez a függvény az alany nyugalmától függő (ezt a NeuroSky headset-től kérdezem le) időközönként létrehoz egy hordót, és elindítja a játékos felé. A nyugalom értékének beállítása az Update függvényben kapott helyet, így az folyamatosan frissül. Az alábbi kódrészlet szemlélteti a működését.

void Update () {

int meditation = AdaptEDConnector.Meditation;

spawnTime = 40000 / (meditation \* meditation);

if (spawnTime > 18)

spawnTime = 18;

}

private IEnumerator spawnBarrels()

{

yield return new WaitForSeconds(7.0f);

while (gameManager.OgreAlive && ( gameManager.Phase1

|| gameManager.Phase2))

{

GameObject barrel = Instantiate(barrelPrefab,

barrelSpawn.position,

Quaternion.identity);

Destroy(barrel, 3.0f);

FindObjectOfType<AudioManager>().playSound("barrelthrow");

yield return new WaitForSeconds(spawnTime);

}

}

### Az elugrás

A játékos felé dobott hordók sebzést okoznak, és ha túl sok találja el, akkor meghal és újra kell kezdeni a harcot. Ennek elkerülése érdekében el lehet ugrani előle az érintőfelület segítségévével. Mivel alapvetően az API nem tartalmaz „swipe” esemény, ezt nekem kellett implementálni, méghozzá olyan módon, hogy érintéskor eltárolom az érintés pozícióját, és az elengedés pillanatában összevetem ezt az aktuális koordinátákkal és ebből számítom ki, hogy történt-e swipe-olás, illetve, ha igen, akkor emlyik irányba. Ezeket a számításokat a VRPlayerDash osztály alábbi metódusa végzi.

void Update () {

if (GvrControllerInput.TouchDown)

{

initialTouchPos = GvrControllerInput.TouchPos

}

if (GvrControllerInput.TouchUp)

{

float deltaPos = initialTouchPos.x -

GvrControllerInput.TouchPos.x;

if (deltaPos > 0.3)

{

if (dashPos == DashPos.Mid)

{

fightStandPos = transform.position -

vrCamera.transform.right\*3;

dashPos--;

}

else if(dashPos == DashPos.Right)

{

fightStandPos = initialFightPos;

dashPos--;

}

}

else if (deltaPos < -0.3){

. . .

Ugyan ez a másik irány esetén is.

}

if (gameManager.Phase1 || gameManager.Phase2)

{

transform.position = Vector3.Lerp(

transform.position, fightStandPos, 0.15f);

}

## Általános funkciók

### Kapcsolat az AdaptED keretrendszerrel

### Audió

A megfelelő hangeffektek lejátszását rendkívül fontosnak találtam, mert ezek nagyban növelhetik a környezet és az élmény valóságosságát. A hangok lejátszását az AudioManager osztály végzi.

Ez úgy épül fel, hogy tartalmaz egy Sound tömböt. A Sound osztály tárolja egy hangeffektnek, minden olyan tulajdonságát, amit állítani szeretnék. Pl.: egy azonosító név, maga a hang, magassága, hangerő vagy, hogy ha vége újrakezdődjön-e. Az AudioManager osztály biztosít különböző metódusokat, szolgáltatásokat a hangok globális, vagy lokális, adott objektumhoz csatolt lejátszásához, illetve a lejátszás állapotának lekérdezéséhez. Ezek a metódusok csak a Sound objektum nevét várják, és utána kikeresik azt a tárolt tömbben. Példának a késleltetett lejátszás metódus az alábbi módon működik (A késleltetés mértékét nem másodpercben, hanem a mintavételezési frekvenciában kell megadni. A dokumentáció alapján, így egy másodperc 44100 Hz) :

public void playSoundWithDelay(string name,float secs)

{

Sound sound = Array.Find(sounds,

s => s.name.Equals(name));

if (sound != null)

{

ulong delay = (ulong)(secs \* 44100);

sound.source.Play( delay);

}

}

### Újrakezdés

# Önálló munka értékelése, mérések, eredmények bemutatása

# Összefoglaló