ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK	Hallgató neve:	**Kovács Krisztián**
2023.	Hallgató törzskönyvi száma:	**T/005701/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Kovács Krisztián**
Törzskönyvi száma: T/005701/FI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

Intelligens parkolóház szimulációja virtuális környezetben
Simulation of an intelligent parking garage in a virtual environment

Intézményi konzulens: Kiss Dániel
Külső konzulens:

Beadási határidő: 2022. május 15.

A záróvizsga tárgyai: Számítógép architektúrák
IoT, beágyazott rendszerek és robotika specializáció

A feladat

Az okos parkolási rendszerek fontos szerepet játszanak a városi mobilitás fejlődésében. A szabad parkolóhely keresésével töltött idő nem csak frusztrációt vált ki a sofőrökben, hanem jelenleg nagymértékű üzemanyag pazarlással is jár, így egy hatékonyan üzemelő és jól kihasznált parkolóház minden szempontból hasznos a forgalmasabb városrészek számára.
A dolgozat célja egy olyan virtuális környezetben kialakított parkolórendszer megtervezése, megvalósítása és szimulálása, amely képes optimálisan elosztani az érkező járműveket azáltal, hogy kiválasztja a számára valamilyen szempontból legalkalmasabb parkolóhelyet, majd az ideális útvonalon elvezeti oda a vendéget. A rendszer a szimuláció során legyen képes különféle virtuális szenzorok és eszközök által szolgáltatott adatokat felhasználni, továbbá legyen megoldott a virtuális vendégek érkezését egy, a rendszerhez kapcsolt külső komponens segítségével szabályozni.

A dolgozatnak tartalmaznia kell:

- a megvalósítandó feladat hátterének rövid bemutatását,
- a jelenleg elérhető hasonló célú rendszerek vagy szimulációs szoftverek ismertetését és elemzését,
- a megvalósítandó rendszer kialakításának és funkcióinak specifikációját, a megvalósításhoz szükséges lehetséges módszerek és technológiák ismertetését,
- az alkalmazás részletes rendszertervét,
- az implementáció menetének ismertetését,
- az elkészített alkalmazás tesztelési tervét, valamint a teszteredmények ismertetését.

Ph.

.....
Dr. Vámossy Zoltán
intézetigazgató

A szakdolgozat elévülésének határideje: **2024. május 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

....
külső konzulens intézményi konzulens

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022. 12. 15.....

.....

hallgató aláírása

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve:	Neptun Kód:	Tagozat:
Kovács Krisztián	[redacted]	**nappali**

Telefon:

Levelezési cím (pl.: lakcím):

[redacted]

Szakdolgozat címe magyarul:

Intelligens parkolóház szimulációja virtuális környezetben

Szakdolgozat címe angolul:

Simulation of an intelligent parking garage in a virtual environment

Intézményi konzulens:

Kiss Dániel

Külső konzulens:

Alk.	Dátum	Tartalom	Aláírás
1.	2021. 09. 29.	Tématerv egyeztetése	Kiss Dániel
2.	2021. 10. 18.	Virtuális környezet megvalósítása	Kiss Dániel
3.	2021. 11. 24.	Rendszerterv egyeztetése	Kiss Dániel
4.	2021. 12. 09.	Dolgozat szövegének korrekciója	Kiss Dániel

A hallgató a Szakdolgozat I. tantárgy követelményét teljesítette, beszámolóra bocsátható.

Kiss Dániel

Intézményi konzulens

Budapest, 2021. december 10.

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve:
Kovács Krisztián

Neptun Kód:
[redacted]

Tagozat:
nappali

Telefon:
[redacted]

Levelezési cím (pl.: lakcím):
[redacted]

Szakdolgozat címe magyarul:
Intelligens parkolóház szimulációja virtuális környezetben

Szakdolgozat címe angolul:
Simulation of an intelligent parking garage in a virtual environment

Intézményi konzulens:
Kiss Dániel

Külső konzulens:

Alk.	Dátum	Tartalom	Aláírás
1.	2022. 10. 04.	Járművek vezérlésének megvalósítása	Kiss Dániel
2.	2022. 11. 15.	Tesztelési terv	Kiss Dániel
3.	2022. 12. 09.	Tesztelési eredmények megvitatása	Kiss Dániel
4.	2022. 12. 14.	Dolgozat véglegesítése	Kiss Dániel

A hallgató a Szakdolgozat II. tantárgy követelményét teljesítette, védésre bocsátható.

Kiss Dániel
.....
Intézményi konzulens

Budapest, 2022. december 14.

TARTALOMJEGYZÉK

RÖVID TARTALMI ÖSSZEFOGLALÓ

Az okos parkolási rendszerek fontos szerepet játszanak a városi mobilitás fejlődésében. A szabad parkolóhely keresésével töltött idő nem csak frusztrációt vált ki a sofőrökben, hanem jelenleg nagymértékű üzemanyag pazarlással is jár, így egy hatékonyan üzemelő és jól kihasznált parkolóház minden szempontból hasznos a forgalmasabb városrészek számára. A dolgozat célja egy olyan virtuális környezetben kialakított parkolórendszer megtervezése, megvalósítása és szimulálása, amely képes optimálisan elosztani az érkező járműveket azáltal, hogy kiválasztja a számára valamilyen szempontból legalkalmasabb parkolóhelyet, majd az ideális útvonalon elvezeti oda a vendéget. A rendszer a szimuláció során legyen képes különféle virtuális szenzorok és eszközök által szolgáltatott adatokat felhasználni, továbbá legyen megoldott a virtuális vendégek érkezését egy, a rendszerhez kapcsolt külső komponens segítségével szabályozni.

A feladat megvalósításához szükséges megismerni a mostani parkolási helyzetet, illetve parkolási szokásainkat. Szükséges lesz megvizsgálnunk a lehetséges eszközöket, amelyeket felhasználhatunk arra a célra, hogy érzékeljük a járműforgalmat, valamint tudjuk, hogy mely parkolóhelyek szabadok, illetve melyek foglaltak. Ezen eszközök ismertetése során, már megtudunk alapozni egy rendszert, így képesek leszünk arra, hogy egy vendég számára parkoló helyett biztosítsunk, a legrövidebb út megtervezéséhez viszont megkell vizsgálnunk és összekell hasonlítanunk a manapság legyakrabban használt algoritmusokat. A fizikai részek megismerését követően, megfelelő környezetet kell választanunk a szimuláció megvalósítására, ismertetni kell a szoftvereket, illetve össze kell őket vetni funkcionalitás és használhatóság alapján. Amint a feladat elméleti síkon stabil lábakon áll, a rendszerver elengedhetetlen része lesz a megvalósításnak meg felvázolja a szimuláció működési elvét, illetve megjelenését is. Fontos megemlíteni, hogy jelen dolgozat, az ezzel megegyező című, szaktársam, Fülep Fanni által készített dolgozattal párhuzamosan, egymást kiegészítve készült. Az Ő dolgozatának eredményeképp kapott adatokat felhasználva fogjuk tudni a szimulációt megvalósítani. Ebből kifolyólag a két dokumentáció tartalmazhat megegyező részeket, és csak együtt adja vissza a teljes megvalósítást.

1. ALAPFOGALMAK

1.1. Közúti Forgalom

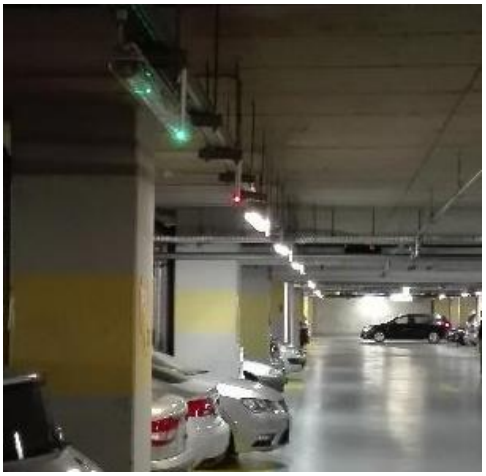
A közúti forgalom egy gyűjtő név, amely utal minden olyan járműre, amely egy megadott nyomtávon halad (út vagy autóút/pálya), és a KRESZ szabályokat betartva közlekednek, mint például a sávtartási kötelezettség, a lámpás forgalomirányításnak való elégtétel, sebességkorlátozás stb. A járművek több kategóriából állnak össze, például autóbuszok, személy gépjárművek, kamionok, kerékpárok.

1.2. Parkolás

A jogszabály „várakozás” néven hivatkozik rá, de a köznyelv szinte kizárólagosan a „parkolás” kifejezést használja, ezért az egyszerű érthetőség jegyében így fogunk utalni rá a továbbiakban. A parkolás szó alatt egy olyan jármű hosszabb ideig, egyhelyben való tartózkodásáról beszélünk, ahol a tulaj nincsen jelen.
Megvalósítás szempontjából lényeges különbség van a közterületi és a beléptető rendszerrel rendelkező parkolóhelyek (mélygarázsok, parkolóházak stb.) között. Utóbbi esetén például a rendelkezésre álló helyek száma könnyen megállapítható a kapunál kiadott jegyek száma alapján, míg közterületen ez egy ennél sokkal összetettebb feladat, amely magában foglalja valamilyen szenzoros hálózat kiépítését. Ebből kifolyólag az irodalomban két fő kutatási irány követhető nyomon: az egyik a közúti (publikus, *on-street*) parkolásra helyezi a hangsúlyt; a másik pedig, a beléptető rendszerrel rendelkező (privát, *off-street*) parkoló létesítményekre fókuszál.
Itt érdemes megemlíteni még a kültéri és beltéri parkolást is, amelyek szintén eltérő megoldásokat igényelnek. Beltéri parkolás esetén a mennyezet is alkalmazható szenzorok, valamint különböző jelzések telepítésére, például gyakran szerelnek fel foglaltságot indikáló piros/zöld színű LED-eket a parkolóhelyek fölé, ilyen található például az Újbudai Allee bevásárlóközpontban is, a 1.1. ábrán látható módon. A 1.2. táblázat összefoglalja a különböző parkolási típusokat és példákat mutat azokra.

1-1. táblázat: A parkolás típusainak összefoglalója

	On-Street	**Off-Street**
Beltéri		Parkolóházak, mélygarázsok
Kültéri	Utcai parkolás	Parkoló udvarok



1-1. ábra: Beltéri parkolás lehetőségei: foglaltságot jelző piros és zöld LED-ek (Forrás: **[1]**)

1.3. Okos városok

Az intelligens város pontos definícióját nehéz egyértelműen meghatározni, mivel a technológiák széles skálája tartozik ezen név alá. Magyarországon a hivatalos definíciót 2017-ben kormányrendeletben rögzítették. [2] Eszerint az okos város egy olyan település, amely korszerű és innovatív információtechnológiák alkalmazásával fejleszti környezetét, infrastruktúráját és szolgáltatásait. Az okos város, mint minden rendszer, különböző alrendszerekre bontható. Ezek az alrendszerek megközelítéstől függően különbözhetnek egymástól, de több szakirodalom is [3] [4] kiemeli a közlekedést, mint egyik fő alrendszert.

1.3.1. Intelligens közlekedési rendszerek

Az intelligens közlekedési rendszerek (ITS) fontos elemei az intelligens városok tervezésének és megvalósításának. Az 1994 óta hivatalosan használt fogalom mára számos más szolgáltatásra is kiterjed. Az Európai Parlament és a Tanács irányelve szerint [5] ezek olyan alkalmazások, amelyek szolgáltatásokat nyújtanak különféle közlekedési módokhoz, valamint lehetővé teszik a jobb tájékoztatást, és biztonságosabb, összehangoltabb közeledést. Feladatai közé tartozik a forgalomirányítás és forgalomszabályozás, forgalmi tájékoztatás, forgalmi ellenőrzés. Mivel a forgalomirányítás a helyváltoztatási lánc teljes egészére kihat, így a parkolás-szabályozásra is [6].

1.3.2. Okos parkolási rendszerek, PGI rendszerek

Az intelligens parkolási rendszerek (SPS) az általuk nyújtott szolgáltatások alapján osztályozhatók, és öt fő szolgáltatási kategóriát különböztethetünk meg [7]: parkolási útmutatás és tájékoztatás *(PGI Systems)*, tranzit alapú tájékoztatás *(TBI Systems)*, automatizált parkolási rendszerek *(APS)*, elektronikus parkolás és intelligens fizetési rendszerek. A parkolási tanácsadó és információs rendszerek célja, hogy támogassa a járművezetőket a szabad parkolóhely keresésében. Az első PGI rendszert az 1970-es évek elején helyezték üzembe Németországban, Aachen város területén. Többségében az első rendszerek a városközpontok parkolási lehetőségeiről nyújtottak tájékoztatást, de a későbbiekben egyre szélesebb körben alkalmazták parkolási létesítményekben is (például repülőtereken és bevásárlóközpontokban) [5].
A fentebb említett TBI rendszerek egyébként nagyon hasonlóak a PGI rendszerekhez, azzal a különbséggel, hogy a közösségi közlekedés megállóinak közelében elhelyezkedő un. P+R parkolók számára nyújt tájékoztatást, így figyelembe veszi például a tömegközlekedési menetrendeket is.

1.4. Modellezés és szimuláció

1.4.1. Szimuláció

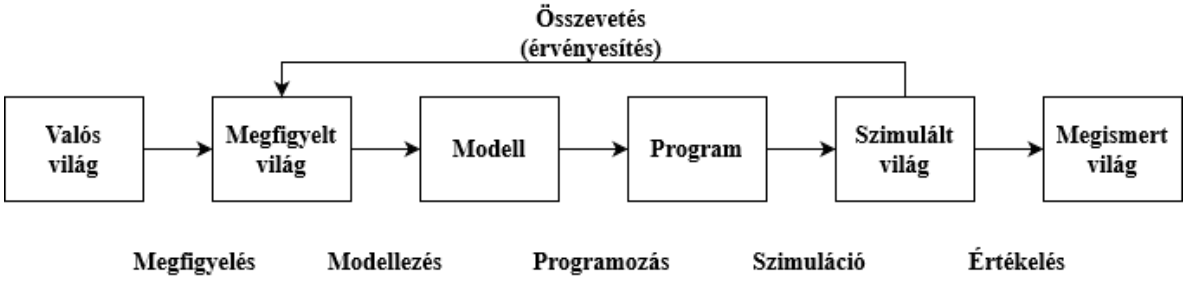
A szimulációt manapság a tudomány számos terén használják. A szimuláció legelterjedtebb jelentése nem más, mint amikor egy rendszernek vagy folyamatnak a várható vagy valós viselkedési módját számítógépes modellek segítségével vizsgálják meg. A számítógépes szimulációknak jellegzetesen sok előnye van a tradicionális „fizikai” kísérletekhez képest, olyan esetekben használják, ahol:

- Túl veszélyes lenne a kísérletet végrehajtani (mérgező/robbanó anyagok tesztelése)
- Az adott kísérletet túl drága lenne megvalósítani
- Túl sok időt venne igénybe (geológiai/botanikus kísérletek)
- Az adott kísérlet túl gyorsan menne folyamatba (robbanások/láncreakciók)
- Kisméretű elemekkel dolgozunk (atomi szint)
- Nagyméretű elemekkel dolgozunk (bolygók, vagy geológiai eróziók vizsgálata)
- Az ideális környezet nem valósítható meg (kémiai anyagok tisztasága/bármilyen elektromos rendszer)
- Az eszköz nem áll rendelkezésre
- Túl sokszor kell elvégezni
- A kísérlethez használt anyag(okból)/tárgy(akból) csak kevés vagy egy létezik
- Túl sok adattal dolgozunk, és vizsgálatuk/megértésük több évet venne igénybe

Ezekben az esetekben roppant fontos a számítógépes szimuláció, és szinte univerzálisan lehet használni a tudomány bármely területén, hogy gyors és pontos eredményeket érjünk el.

1.4.2. Modell

A modell a szimuláció szerves részét képezi, amely pontosan (matematika nyelvén) megfogalmazott állítások (hipotézis) és állítás-rendszereket foglal magába. A modell megalkotása során csak azokat az elemeket emeljük ki, melyek ismertek, feltételezettek és a szimuláció működéséhez feltétlenül fontosok, és ezen elemek hipotéziseit megfelelő módon kapcsolatba hozzuk egymással. Amikor a modell tervezésével és felépítésével végeznek, a tesztelés veszi kezdetét, ahol ellenőrzik, hogy a modellezni kívánt rendszer vagy elem módjára viselkedik. Ezt a folyamatot nevezzük validálásnak vagy verifikálásnak. Amennyiben a modell nem a valóságot tükrözi, úgy ebben az esetben finomítani kell, vagy az alapjaitól újra felépíteni, amely az addigi munka elvetésével jár. A modellezés folyamatának lépéseit a 1.2. ábrán láthatjuk.



1-2. ábra: A modellezés folyamata (Forrás: Saját készítés, [8] alapján)

Modell típusok [9]:

- Diszkrét Modell: Abban az esetben beszélünk diszkrét modellről, amikor az állapotváltozás nem folyamatosan, hanem (diszkrét) időintervallumokban következik be.
- Folytonos Modell: Az a modell, amely állapotváltozása minden pillanatban bekövetkezik, folyamatos modellnek hívjuk
- Determinisztikus modell: Az a modell, amelynél az eredmény egyértelműen meghatározható a program futásához megadott bemenő adatokból, determinisztikus modellnek hívjuk. Ehhez hozzá tartozik az is, hogy mindig ugyan azt az eredményt adja azonos bemeneti paraméterek és feltételek alapján, függetlenül, hogy mikor és ki futtatja a szimulációt
- Sztochasztikus modell: Ezt a szimulációt összefoglaló néven Monte Carlo-módszernek is szokták hívni. A determinisztikus modellel ellentétben itt lehetséges a véletlen számok is, melynek következményében a kimenet eltérhet, hiába az azonos bemeneti paraméterezés.

2. A TÉMA AKTUALITÁSA, JELENTŐSÉGE

A szabad parkolóhely keresése nem csak frusztrációt okoz a sofőröknek, de az üzemanyag felesleges pazarlásával növeli környezetszennyezést, valamint a bizonytalan mozgásuk miatt balesetveszélyes, illetve feltartóztatják a többi járművet [2]. Egyes tanulmányok szerint évente 700 millió eurós veszteséget okoz pusztán Franciaországban a parkolóhelyek keresésével töltött mintegy 70 millió óra. [13] A helyzetet tovább súlyosbítja a gépjárművek számának folyamatos növekedése. A KSH adatai szerint közel 4 millió személygépkocsi fut a magyar utakon, az elmúlt 20 évben több, mint másfélmillióval emelkedett [10].

Átlagosan a nagyvárosi gépjármű forgalom 30 százalékát az épp parkolóhelyet kereső járművek teszik ki. [10] Ez a probléma parkolási létesítmények esetén is fennáll: a Repülőterek Nemzetközi Tanácsa (ACI-NA) által végzett kutatás szerint az utasok jelentős késéseket szenvednek el a parkolóhelyek keresésének következtében, mivel jellemzően gépjárművel érkeznek a repülőtérre. [11]

2.1. Parkolóingatlanok piaca

A parkolóház-beruházások során kiemelkedően magas a csődbe jutott, leállított és elhalasztott projektek aránya. Budapesten a 2000 és 2015 közötti időszak folyamán összesen 30 parkoló-létesítményt létrehozó fejlesztés valósult meg. Fontos megemlíteni, hogy ennek jelentős része olyan projekt volt, ahol az építési szabályozás írta elő az ingatlan típusához kötelezően kialakítandó parkolóhelyek darabszámát (ilyenek jellemzően a bevásárlóközpontok). Ezeket a beruházásokat figyelmen kívül hagyva 15 év alatt mindössze 7 projekt valósult meg önálló fejlesztés részeként. Ehhez a számhoz viszonyítva jelentős mennyiségű, megközelítőleg 130 parkolóház és/vagy mélygarázs beruházás állt le csak Budapesten ugyanezen időszak alatt. [12]

A nehézkes fejlesztéseknek műszaki, valamint gazdasági vonatkozású okai is vannak: ilyenek például a mélygarázsok talajvíz elleni szigetelésének megoldása, valamint a hosszú megtérülési idő által okozott, külső finanszírozással kapcsolatos problémák. Utóbbihoz hozzájárul az is, hogy a szabadtéri parkolók alacsony parkolási díjai miatt a parkolóházat üzemeltető cég is kénytelen alacsonyabb árszintet alkalmazni, ami akár 35–40 éves megtérülési időt is eredményezhet.

Ugyanakkor az utóbbi évtizedek során a gépjárművek folyamatosan növekvő számából kifolyólag a parkolás kérdése a főváros közlekedésének egyik legnagyobb problémájává nőtte ki magát. A kialakult parkolási krízisből számos környezeti konfliktus is ered, többek között a városi életminőség csökkenése, a zöld területek hiánya és a városkép romlása a járdán parkoló gépkocsik miatt.

A parkolási problémák legkézenfekvőbb megoldása a saját tulajdonú gépkocsi iránti igény csökkentése a lakosság körében, például a helyi tömegközlekedés fejlesztésével, járatok

sűrítésével vagy autómegosztó szolgáltatás (car sharing) segítségével. Azonban az ilyen, szemléletformáláson alapuló megoldások hatása előreláthatólag több évtizedes távlatban lesz csak érzékelhető, a gépkocsihasználati szokások viszonylag lassú változása miatt. Ugyanakkor az is belátható, hogy a fent említett nagymértékű társadalmi feszültség csökkentése sürgető feladat, amely rövidtávú gyors fejlesztéseket is igényel.

Számos parkolásmenedzsment stratégia épít a már rendelkezésre álló parkoló létesítmények minél jobb kihasználása, hiszen aránylag gyorsan megvalósítható, továbbá gazdaságilag is igen kedvezőnek mondható megoldás, mivel nem jár új építési beruházással. Ennek több lehetséges módja van, egyrészt a kapacitás növelése által, például a létesítményben meglévő üres vagy kihasználatlan területek jobb felhasználása révén. Sajnos a tapasztalat viszont azt mutatja, hogy a járművezetők általában az utcai parkolást választják, így a létesítmények eleve nincsenek teljesen kihasználva. Következésképpen ösztönözni kell őket arra, hogy többször válasszák ezeket a létesítményeket; ennek lehetséges eszköze a PGI rendszerek alkalmazása, amivel vonzóbbá tehetőek ezek a létesítmények a gépjárművezetők számára.

3. PARKOLÓHELYEK FOGLALTSÁGÁNAK ELLENŐRZÉSE, ÉS SZENZOR LEHETŐSÉGEK

A parkolóhelyek foglaltságának monitorozása fontos szerepet játszik a vezetők támogatásában, amikor szabad parkolóhelyet keresnek egy adott parkoló részen. Ezt a monitorozási műveletet többféle képen is meglehet valósítani, attól függően, hogy hogyan szeretnénk felismerni egy parkoló foglaltságát. A leggyakoribb módszer erre a lehetőségre, különböző szenzorok használata. Ezeket a szenzorokat két nagyobb csoportba oszthatjuk be, ha általánosságról beszélünk [13]:

- Intruzív (Intrusive) Szenzorok: Ezek a szenzorok általában az utak, illetve plafonok felületébe telepített érzékelők. Ezeknek a szenzoroknak több változata lehet, az ár, környezeti feltételek, kalibráció, pontosság, hatótáv és design függvényében. A szenzorok a beépítési sajátosságuk, közvetve hozzájárulhat az adott útszakasz vagy mennyezet romlásához és csökkentik az élet tartalmát mivel telepítésükhöz megkell bontani az adott szerkezetet. Lehetséges opciók ezen eszközök közül a(z): infravörös és piezoelectric szenzorok, magnetométerek (magnetometers) és az indukciós tekercs alapú eszközök (Inductive Loop detectors).

- Nem Intruzív (Non-Intrusive) Szenzorok: Ezen szenzorok telepítése és karbantartása nagyságrendekkel egyszerűbb, mint az intruzív szenzoroké, mivel ezek a műveletek nem igénylik a telepítendő hely részleges vagy teljes megbontását. Ezekből a szenzorokból is számos változat áll rendelkezésre, ugyan azon függvények alapján, mint az intruzív változatainak. Ilyen megoldások lehetnek az optikai, ultrahangos és akusztikus szenzorok, valamint az RFID és képfeldolgozás.

3.1. Szenzorok bemutatása

A következő fejezet fontos szerepet tölt be a projekt felépítésében, illetve annak megvalósításában. Bemutatja azon szenzorok fajtáit és típusait, amelyek a leggyakrabban előfordulnak, mind parkoló helyeken (legyen az utcai, illetve parkolóház), illetve a forgalom irányítás, figyelés és analízis közben. Ez eszközök telepítése, működése, karbantartása és hátrányainak listája alapján fogjuk majd meghatározni, hogy a szimuláció elvégzéséhez melyik szenzort, vagy szenzorokat fogjuk felhasználni.

3.1.1. Infravörös Szenzor

Ezen szenzorok segítségével parkolóhelytől és emelettől függetlenül tudunk ellenőrizni egy bizonyos pontot. A szenzor egy infravörös sugár kibocsátásával dolgozik. A visszaérkező energia függvényében tudja meghatározni, hogy az adott pont szabad-e vagy sem. A jármű pozíciója mellet. pontosan meglehet mondani annak sebességét is. Hátránya ennek a megvalósításnak, hogy a környezeti hatásokra érzékeny, legyen az hó vagy esetleg köd.

3.1.2. Indukciós Tekercs Alapú Eszközök

Az indukciós (angolul Inductive Loop detectors) tekercseken alapuló szenzorok, a legelterjedtebbek a forgalom analízisben. Ezen eszközök alapját egy feltekercselt vezetékrendszer adja, melynek mérete és alakja nem számottevő a feladat ellátás során. A tekercs átlagosan a 10 és 50 kHz tartományban dolgozik, és amikor egy jármű elhalad vagy megáll a tekercs felett, képes annak önindukcióját befolyásolni, így közvetlenöl változtatja az oszcillációs frekvenciáját. A működési elve miatt, elterjedt és számos más helyen is hasznosítható, mint például forgalom nagyságának mérésére, foglaltság vizsgálatára és akár sebesség mérésére is. Hátránya, hogy a telepítési formája miatt, a forgalom által generált stresszre érzékeny, akárcsak a hőmérsékletváltozásra, ezen felül a lefedett terület minimális, így egy nagyobb méretül szakasz monitorozása több egységet is igényel.

3.1.3. Magnetométerek

A magnetométerek (angolul magnetometers) az indukciós tekercs alapú szenzorok javított változata. Előnyük közé tartozik, hogy kevésbé érzékenyek az időjárási hatásokra, ideértve a hó, köd, illetve a hőmérsékleti változásokat. A technológiai fejlődésnek köszönhetően, képesek akár vezetéknélküli módon kommunikálni, egy a közelben elhelyezett RF vevővel, adatt továbbítás érdekében. Hátrányuk viszont jelentős, mivel rengeteg modellnek a lefedett területe kicsi, és a relatíve újszerű technika miatt telepítésük és karbantartásuk költséges.

3.1.4. Piezoelectric

A piezoelectric szenzorok működése a kinematikus energián és az elektromos energián alapul. A szenzorok különböző rezgéseket mechanikai nyelnek el, ebből megállapítható egy jármű elhaladása, illetve sebessége egy adott szektorban. Hátránya, hogy az időjárási változásokra roppant érzékeny, illetve a lefedettségi terület kicsi, így több szenzort telepítése szükséges az adott helyen.

3.1.5. Optikai Szenzorok

Az optikai szenzorok (angolul optoelectronics sensors), a fényt használják alapul és arra reagálva tudják megállapítani, hogy van-e jármű az adott ponton vagy sem. Ezeknek a szenzoroknak a felépítése már valamennyivel bonyolultabb az eddig említetteknél, ugyanis ők már tartalmaznak RF adó-vevőket, magát az optikai szenzort, illetve egy vagy több mikrokontrollert. Ezeket a hardvereket előszeretettel szokták párosítani az úgynevezett Round Robin időzítési technikával, azért, hogy megfigyelhessük egy adott szakasz forgalmát. A hátránya ezeknek az érzékelőknek, hogy nem tudnak egy személygépjárművel megkülönböztetni egy gyalogostól, illetve a fényszennyezés ronthatja a teljesítményüket.

3.1.6. Ultrahangos Szenzorok

Az ultrahangos szenzorok működése hanghullámokon alapul. Az eszköz egy 25 és 50 kHz közti impulzust bocsát ki működése közben és a visszaérkezett energia mennyiség függvényében képes megállapítani azt, hogy egy adott területen van-e bármilyen akadály vagy sem. Ennek a hardvernek az társaival szemben az, hogy nagyobb területen is képes működni, így akár több egymás mellett elhelyezkedő parkolóhelyről megtudja mondani, hogy az szabad-e vagy sem, illetve telepítésük és karbantartásuk relatíve egyszerű és olcsó. A hátrányai közé tartozik az, hogy nem viseli jól a hőmérsékletváltozást így a teljesítménye romolhat.

3.1.7. RFID

Az RFID megvalósítás az egyik legbiztonságosabb és legeffektívebb alternatíva a járművek detektálására. Egy ilyen egység általában három részből áll:

- Rádió adó-vevő (Transceiver): Melynek feladata, hogy adatokat küldjön, illetve fogadjon a következő egységtől.
- Válaszjeladó (Transponder): Ez az egység kapja meg az adó-vevőtől az adatot és ő tárolja el, valamint küldi tovább azt a harmadik egység segítségével, amely az antenna.

- Antenna: Az antenna segíti a kommunikációt az adó-vevő, illetve a válaszjeladó között, így a jel távolabb elér, illetve erősebben fogható.

Az RFID előnye, hogy telepítése és üzemeltetése költséghatékony, és az előforduló hibák relatív gyorsasággal orvosolhatóak. Egyetlen hátránya, hogy ilyen esetekben egy parkolót egy gépjárműhöz rendelünk, illetve a felhasználó regisztrálásánál felmerülhetnek esetleges adatkezelési problémák a személyes adatok megadása miatt.

3.1.8. Akusztikus Szenzorok

Az akusztikus szenzorok működési elve relatíve egyszerű módszeren alapul. Az egység működéséért elsősorban egy mikrofonon alapul, amely a személygépjárművel által kibocsátott hangot érzékeli. Előnyei közé tartozik, hogy képes nagyobb területet, illetve több sávos megfigyelést is képezni, attól függően, hogy a hangot melyik irányból fogja fel a mikrofon. Hátránya közé tartozik, hogy egyes modellek rossz minőségű mikrofonnal rendelkezhetnek, a hideg hőmérsékletet nem kedvelik, illetve lassan mozgó és elektromos járművek detektálása problémás lehet, mivel ezen járművek nem generálnak megfelelő erősségű motorhangot vagy gördülési zajt.

3.1.9. Képfeldolgozási Módszer

A képfeldolgozási módszereken alapuló monitorozás általában egy vagy több videókamera felhasználásával valósul meg. Ezen kamerák videó jelét egy képfeldolgozó szoftverbe vezetik be, amely egy számítógépen vagy mikrokontrolleren fut. Ezen szoftverek az elmúlt évben jelentős teret nyertek mind közterületi mind magán parkolók területén, illetve számos megvalósítás megtalálható az interneten, mint például az OpenCV [14]. Általános felhasználása ennek a rendszernek a több sávos megfigyelés, rendszám olvasás és azonosítás [15], illetve közterületek figyelése. A rendszer telepítése és működtetése nem igényel nagy költségeket, viszont az Okos Parkolási Rendszerek (Smart Parking Systems) nem szeretik alkalmazni, mivel a kamera mozgása drasztikusan csökkenti a rendszer teljesítményét, így nyílt kültéri helyeken a szél jelentős gondot tud okozni, akárcsak a fény hiánya, hiszen ebben az esetben a kamerák képminősége homályos, szemcsés lesz.

3.2. Összegzés

Az előző szekcióból jól megismerhetőek a különböző szenzor típusok, illetve, hogy ezen típusok közé mely szenzor eszközök tartoznak. Egy tényleges parkoló infrastruktúra megépítésénél a nem intruzív megoldás a lehető legjobb választás. Ennek oka nem más, mint az egyszerű és költségkímélő telepítés, illetve a későbbiekben az egyszerű karbantartási folyamat. Mivel viszont a feladatunk egy szimulációs folyamat lesz, így nem teszünk különbséget a szenzorok közt, így mindkét nagyobb szenzor típus elérhető lesz a programban.

4. ÚTKERESÉSI ALGORITMUSOK

4.1. Elterjedt Algoritmusok Összehasonlítása

4.1.1. Dijkstra Algoritmus

A Dijkstra algoritmus egy közismert megvalósítása a legrövidebb út keresési problémára [16] [17]. Az algoritmus az A pontból B pontba vezető minden utat, illetve útszakaszt, valamilyen súllyal terheli, és ezután a lehető „legköltséghatékonyabb" útvonalat választja ki, de futása során minden útvonalat megvizsgál a kiindulási ponttól. Ennek következménye a rossz keresési teljesítmény és hosszú keresési idő, amely számottevően érezhető, ha a két pont közötti távolság nagy.

- Első lépés: A kezdeti pont kiválasztása
- Második lépés: Kiszámoljuk a kezdőpont és az összes kapcsolódó csomópont közötti út súlyát, majd megjelöljük melyik útvonal rendelkezik a legkisebb értékkel
- Harmadik lépés: Kiszámoljuk a kezdőpont, az előzőleg kiválasztott legkisebb értékkel rendelkező pont és az összes csomópont közötti út súlyát, majd megjelöljük melyik útvonal rendelkezik a legkisebb értékkel
- Negyedik lépés: Addig ismételjük a harmadik lépést amíg a célponthoz nem találjuk meg a legrövidebb utat

Ezen probléma miatt a Dijkstra nem megfelelő például valós idejű útvonal keresésre, így nem használható például GPS alapú navigációra, vagy merevlemezes navigációra se. Viszont megoldásként rengeteg módosított változatát találhatjuk meg az algoritmusnak a világhálón, illetve számos másik tanulmányban is.

4.1.2. Kibővített Dijkstra Algoritmus

Az eredeti algoritmusban a keresési tartomány folyamatosan növekszik ahogy a folyamat halad előre, ennek köszönhetően az ellenőrizendő csomópontok száma drasztikusan nő, minden egyes iteráció során, így növekszik a keresési idő. A bővített metódus két irányba dolgozik, mind a kezdeti mind pedig a cél végpont felől kezdi el a keresést ezen felül rendelkezik egy idő limittel, amely túllépése esetén az algoritmus újra indul egy másik specifikus régión. Ennek köszönhetően a vizsgálni kívánt régió mérete, illetve csomópontok száma drasztikusan csökken és a futási idő jelentős mértékben javul [16].

- Első lépés: A kezdetipont, illetve végpontok kiválasztása
- Második lépés: Kiszámoljuk a kezdőpont és az összes kapcsolódó csomópont közötti út súlyát, majd megjelöljük melyik útvonal rendelkezik a legkisebb értékkel. Majd kiszámoljuk a végpont és az összes kapcsolódó csomópont közötti út súlyát, majd megjelöljük melyik útvonal rendelkezik a legkisebb értékkel
- Harmadik lépés: Megjelöljük azokat a csomópontokat a vizsgált területen, amiknek a legkisebb a súlya és a második lépésben kiválasztott csomóponttal összeköttetésben állnak.
- Negyedik lépés: Addig ismételjük a harmadik lépést amíg a célpont, illetve a végpont keresése nem fedi le egymást.

A művelet elvégzése közben a futási idő növekedhet, ha a keresési tartomány túl tágra bővül, ebben az esetben ha a tartomány vagy a keresési idő átlép egy határértéket, a keresés megszakad, és egy új tartomány lesz kijelölve a keresés során [16].

4.1.3. A* Algoritmus

Az A* egy gráf bejáró és útvonal kereső algoritmus, amely gyakran használatos a számítástechnika számos területén, az optimális hatékonysága miatt [18]. Az algoritmust Peter Hart, Nils Nilsson és Bertram Raphael, a Stanford Kutatóintézet dolgozói publikálták először 1968-ban, amely az egyszerű Dijkstra algoritmus ki bővítéseként említhető, viszont annál jobb teljesítménnyel bír, mivel a kereséshez heurisztikus megvalósítást használ [19].
A program fő működési eleme egy kiértékelési függvény (1):

$$f(n) = \mathrm{g(n)} + h(n) \tag{1}$$

amely függvény útmutatást nyújt és kiválasztja a megfelelő csomópontokat az úgynevezett OPEN listában, ahol g(n) a kiindulási csomóponttól az n csomópontig tartó tényleges költség (azaz, az optimális útvonal megtalálásának tényleges költsége), h(n) a becsült, az n csomópontból a célcsomópontba vezető optimális út költsége, amely a probléma heurisztikus információitól függ [20]. Ezeknek a logikai megoldásoknak köszönhetően, az A* algoritmus nem járja be az összes lehetséges csomópontot és utat a pontok közt, és hatékonysága ebben rejlik a Dijkstrával szemben:

- A kezdeti csomópont kijelölése és a szomszédos csomópontok vagy későbbi jelöletlen csomópontok feltérképezése
- Kiszámítja az értékelő függvény értékét minden egyes lehetséges csomópontra, rendezi őket a függvény eredmény értéke szerint, majd azonosítja és megjelöli a

legkisebb értékel rendelkező csomópontot. A keresés addig nem állítható le, amíg az aktuálisan vizsgált csomópont nem lesz a célcsomópont.

- A fenti feldolgozási lépéseket kell alkalmazni, valamint tárolni a legrövidebb útvonalat, mindaddig amíg az aktuális csomópont nem a célcsomópont.

Az algoritmus megvalósításának legfontosabb szerepe a megfelelő kiértékelő algoritmus kiválasztása. Ha ezt a függvényt megfelelően választjuk ki, akkor biztosak lehetünk abban hogy, az algoritmus a lehető legrövidebb útvonalat határozza meg.

4.2. Összegzés

A fenti összehasonlításból megállapítható, hogy a Dijkstra algoritmus nem lesz megfelelő a feladat megvalósításához. A szimuláció megfelelő futása érdekében törekedni kell a lehető leggyorsabb, illetve a leghatékonyabb megvalósításra, így arra a konklúzióra jutottunk, hogy a későbbiekben kiválasztott grafikus motor AI navigációs megvalósítását fogjuk egybe dolgozni az A* algoritmussal, így elérve a legjobb eredményt.

5. GRAFIKUS ÉS JÁTÉK MOTOROK

A szimuláció megvalósításához szükségünk lesz egy szoftverrel, amely képes a bemeneti adatok feldolgozására, a modell értelmezésére, a járműveket vezérelni, illetve ezt valós időben meg is tudja jeleníteni. Az ilyen szimulációs programok megírása hosszú és komplex feladat, erre viszont megoldás lehet egy már megírt környezet letöltése vagy megvásárlása. Mivel ezen programok nem biztos, hogy tartalmazzák azokat a funkciókat, amikre a feladat megoldása során szükség lehet, ezért egy modern játék motor fogja a megfelelő környezetet megadni. A játékmotorok egy összetett, több felhasználási területen alkalmazható program csomag, amely tartalmaz alapokat, illetve eszközöket például mesterséges intelligenciához, megjelenítéshez, adatfeldolgozáshoz, szkripteléshez.

5.1. Unreal Engine

Az Unreal Engine az Epic Games által fejlesztett és karbantartott grafikus játékmotor, mely jelen pillanatban piac vezető pozíciót tölt be renderelés terén, illetve felhasználtság terén is első helyen helyezkedik el [21]. A motor első verziója az Unreal Engine 1.0 1998-ban jelent meg és az Unreal nevű játékkal debütált. A grafikus megjelenítő már abban az időben is felülmúlta a versenytársait a világítás és megjelenítési technikák terén.

A második verzió 2002-ben látta meg a napvilágot az amerikai hadsereg által fejlesztett America's Army szimulációs játékon keresztül. Az elődjéhez képest, fontos újítások kerültek a motorba, mint például a hardver gyorsított megjelenítés, ami azt jelenti, hogy a dedikált videókártyák feladata volt a megjelenítési számítások elvégzése, így levették ezt a terhet a processzorokról. Valamit fontos módosításokon ment keresztül a fizikai szimulációs modul, amelyet anno a Karma fizikai szimulációs rendszer szolgáltatott. Ennek köszönhetően akár az előző generációhoz képest, 100-szoros részletességgel lehetett modelleket, effekteket és pályákat megvalósítani és megjeleníteni a képernyőn. A második generációt követően kezdett az engine elterjedni, és a licenszelési lehetőségek miatt külső cégek is használhatták azt saját projektjük megvalósításához. Fontos megjegyezni, hogy a második generáció nem volt elérhető a nyilvánosság számára, az átlag felhasználók csak akkor tudtak hozzá férni a kódhoz, illetve a motorhoz, ha a fejlesztői környezetet az adott játékkal együtt összecsomagolták.

A harmadik generáció 2004-ben mutatták be, de csak két év múlva, 2006-ban a Gears of War nevű játékkal együtt került piacra. A motor innentől kezdve, fókuszba helyezte az objektum orientáltságot, és az úgynevezett „data-driven" magyarul adat vezérelt szkriptelési és programozási folyamatot, és modulokra osztotta a felhasználható kódot. A motor még ebben a generációban is az úgynevezett UnrealScript nevű programozási nyelvet használta, mint elődjei, és a tényleges natív C++ kódot csak akkor lehetett használni, illetve módosítani, ha az adott cég a megfelelő licenszt vásárolta meg. A harmadik generációval együtt, a cég

elkezdte fejleszteni az ingyenesen használható és bárki számára elérhető Unreal Fejlesztői Csomagot (angolul Unreal Development Kit, röviden UDK) [22], amely 2009-ben jelent meg hivatalosan, és minden eszközt tartalmazott, a motor tényleges forráskódját leszámítva, amelyet a nagy kiadók is megkaptam amikor licenszt vásároltak a motor használatához. Ennek a lépésnek köszönhetően rengeteg kezdő, illetve kis létszámú (úgynevezett „indie") fejlesztő csapat ismerhette meg közelebbről a motort, és kedvező licenszelési stratégiák miatt, akár ki is adhatták a lefejlesztett terméket. Mindezekből az következett, hogy a motor robbanásszerű térhódításba kezdett.
Az engine negyedik generációja, 2005-ben már 2 éve fejlesztés alatt volt és első bemutatása a 2012-es Game Developers Conference (GDC) rendezvényen történt meg, igaz zárt ajtók mögött, és később 2012 június 7.-én látta meg a nagyközönség. Az új grafikus motor számtalan újdonsággal érkezett:

- Egyik legnagyobb újítás a valós idejű megvilágítási rendszer volt (real-time global illumination) amelynek köszönhetően a statikus rendszer alapú világítást lecserélhették és valósághűbb fényeket érhettek el.
- A másokik jelentős újítás a C++-ra való teljes átállás és az UnrealScript teljes elhagyása.
- A harmadik az úgynevezett Blueprint rendszer bevezetése (amely az Unreal Engine 3 Kismet rendszerét cserélte le), mely egy virtuális szkriptelési rendszer, amely nem igényel tényleges kódolást, így gyorsan lehet prototípus rendszereket létrehozni és tesztelni azokat, ezzel is elmosva a határt a kódolók, designerek és a technikai művészek közt.
- A negyedik és talán legfontosabb, hogy a motor a teljes forráskóddal együtt open-source projektként jelent meg, így bárki tudja ingyenesen használni és módosítani azt bármiféle licensz nélkül. Fontos megjegyezni, hogy a motor kereskedelmi célú felhasználása licensz igényes.

A számtalan újítások és a mai napig érkező frissítések és javítások miatt, az Unreal Engine 4-dik verzióját számtalan területen felhasználják, legyen az akár filmek és reklámok készítése [23], mint például a The Mandalorian című sorozat, architektúrai munkák, jármű tervezési iparág, vagy szimulációs projektekhez, mint például önvezető autós technológiák gyorsabb fejlesztéséhez [24].

5.1.1. Unreal Engine 4 Blueprint

Az Unreal Engine 4-ben található Blueprint vizuális szkript rendszer (Blueprint Visual Scripting System) egy teljes eszköztár, amely azon a koncepción alapul, hogy egy csomópont-alapú (node-based) interfész segítségével objektumokat, program logikát, illetve játékelemeket hozzunk létre, mindezt a fő szerkesztőn belül, mindenféle előzetes kódolási tudás nélkül. Ahogy bármelyik másik hasonló rendszer, ez is az objektum orientált megvalósításon alapul, így előre ledefiniált osztályokat vagy objektumokat tudjuk használni, ami a motorban megtalálható. A rendszer rendkívüli rugalmassága és erőteljessége lehetővé teszi azt, hogy a designerek az eddig csak a programozók által elérhető eszközöket felhasználhassák. Emellett a C++ programozók képesek olyan rendszereket létrehozni, amelyeket később a designerek a saját kedvükre ki tudnak bővíteni vagy felül tudják azt írni [25].

5.1.2. Unreal Engine 4 C++

Az Unreal Engine 4-től használható C++ programozási nyelv, a C++11 azaz a C++ 2011-es revízióján alapul. Az a megszokott funkciókon felül, számos újítást, illetve motor-specifikus megvalósítás található, amely segíti a fejlesztők munkáját, ilyen például az Unreal Smart Pointer, Header Cross-Reference eszköz, illetve az Unreal Header Tool, melynek lényege egy előzetes már előre elkészített „header” fájl állományt hoz létre minden függőség implementálásával. A kódbázis már egy előre felépített keretrendszert szolgáltat számunkra, melynek köszönhetően gyorsan és hatékonyan hozhatunk létre különböző modulokat illetve osztályokat a projekthez [26].

5.2. Unity

A Unity játék motor 2005-ben mutatta be és adta ki az 1.0 verzió számú kiadását, és az Unreal-al ellentétben nem egy játékon keresztül látta meg a napvilágot, hanem az Apple által tartott Worldwidwe Developers Conference nevű eseményen. Ez a verzió még csak macOS-X exkluzív program volt, mely később több platformos programmá nőte ki magát.
A Unity 2.0 2007-ben jelent meg és számos újítást hozott magával, mint például

- Valós idejű árnyék megjelenítés, így a dinamikusan mozgó objektumok a valóságnak megfelelő árnyékot vetettek ki
- Irányított fények, és reflektorokat imitáló fények, ezzel is tükrözve a valóságot
- Videófájlok lejátszása
- Hálózat kezelő modul, aminek segítségével online játékokat és projekteket lehet megvalósítani.

Mivel a Unity egyik építőeleme az Apple termékek támogatása, 2008-ban az App Store indulásával együtt, integrálta a legelső iPhone támogatását, hogy egy könnyen használható környezetben lehessen fejleszteni az eszközre.
A harmadik generációs kiadás 2010 szeptemberében jelent meg, és frissítette a render modult, valamint magával hozta az Android támogatást is. Ezen felül kapott egy új világítás és audió rendszert. Mivel az engine elsősorban integrálta az okostelefonokra való fejlesztés lehetőségét, hamar elterjed azon fejlesztők között, akik ezeket a platformokat célozták meg. Egy 2012 májusában tartott kérdőív alapján, a Unity volt a legelterjedtebb grafikus motor a mobil piacon és mintegy 1.3 millió fejlesztő használta.
2012 novemberében megjelent a Unity 4.0, amely animációs és renderelési újdonságot foglalt magába. Ilyenek voltak a Microsoft DirectX 11, és Adobe Flash implementáció, a Mecanim animációs rendszer, mellyel IK alapú locomotion és fizikai animációkat lehetett létrehozni, valamint egy demó mód a Linux alapú verzióról is megjelent. Az elkövetkezendő évek alatt a Unity nem kapott nagyobb frissítést, csak hibajavításokat. A következő nagyobb verziós csomag 2015-ben jelent meg és a Unity 5.0 névre hallgatott, és számtalan funkcióval jelent meg

- Egyik legnagyobb újítás a valós idejű megvilágítási rendszer volt (real-time global illumination) akár csak az Unreal Engine 4-ben.
- WebGL integráció, így az adott projektek tudnak böngészőből is futni, bármilyen extra plugin vagy modul nélkül, teljesen natívan
- Nvidia PhysX 3.3 fizikai rendszer implementációja
- valamint egy teljesen új, a nulláról újraírt audió rendszer, illetve FMOD audió eszközök integrálása

Ezen felül a motor számtalan teljesítmény béli javítást is kapott, illetve új render API-kat integráltak, mint például az AMD által használt Vulkan rendszer. Ettől a verziótól kezdve a cég elhagyta a szekvenciális elnevezést, és áttért a kiadás évét tartalmazó elnevezésekre. 2018-ban a Unity 2018 C# alapú forráskódja elérhető lett, viszont csak és kizárólag „referencia” -ként használható, mivel a licensz tiltja bármi nemű újra felhasználást vagy módosítást. Sikeressége és felhasználó barátsága, illetve a könnyen tanulható programozási nyelve miatt számtalan területen használják a mai napig, mint például filmek és reklámok készítése [27], szimuláció, építészet architektúrai munkák [28] vagy a jármű tervezési iparág.

5.2.1. Unity Programozási Nyelvek

A Unity három lehetséges opciót biztosított a fejlesztők számára, programozás terén, mely 2014-ben kettőre csökkent [29]. Az első a Unity JavaScript vagy ismertebb nevén UnityScript, a Microsoft C# és a .Net keretrendszer, illetve a Boo, amely a kis felhasználói tábor miatt 2014-ben eltávolításra került a motorból. Felhasználás, illetve lehetőségek terén mindhárom nyelv ugyan arra képes, így a használni kívánt nyelv kiválasztása teljes egészében a programozóra van bízva. A UnityScript egy JavaScript szerű nyelv, amely a kezdők számára tökéletes kiindulási pont lehet, mivel az interneten rengeteg segédlet és példa megoldás található. A nyelv a JavaScript-tel ellentétben objektum orientált megvalósításon alapul, így megtalálható minden olyan előny, amely az osztályokból, valamint öröklődésből származnak.
A második opció a már ismert és világszerte elterjed C# programozási nyelv, melyet inkább a haladó vagy professzionális programozók használnak. Az erősen típusos mivolta miatt egy nehezebben tanulható nyelv, és a UnityScript-tel ellentétben, egyik fő hátránya, hogy a típus konverziók nem automatikusan mennek végbe. Mivel a Unity tud natívan C# kódot futtatni, így ezen nyelv használatával lehet a legjobb teljesítményt elérni, és a fordítási idő is ebben az esetben a leggyorsabb.
Az utolsó használt nyelv a Boo, amely egy Python szintaktikára épülő programozási nyelv, és strukturális felépítése nagyon hasonlít A UnityScript-re. Ez a nyelv nem annyira elterjed a fejlesztők körében, így segédanyag nagyon kevés található meg az interneten hozzá [30].

5.3. Összegzés

A két motor kifejtésében látszik, hogy a programozási nyelveken, illetve a felhasználótáboron kívül jelentős eltérés igen kevés esetben található. Mindkettő rendelkezik a valós idejű megvilágítási rendszerrel, támogatja a Microsoft DirectX 11, DirectX 12, valamint a Vulkan API megvalósításokat. Így a program kiválasztása ebben az esetben teljes mértékben saját preferencia alapján történik. Az projekt megvalósításához az Unreal Engine-t választjuk, azon belül is a nemrégiben megjelent, alpha stádiumban lévő 5-ös változatát fogjuk felhasználni. A motort már az Unreal Engine 3-as (UDK) verzió óta használom, saját projektek megvalósításához, a saját képességeim fejlesztéséhez, illetve egyéb kisebb szimulációk és tesztek létrehozására.

6. RENDSZERTERV

6.1. A szimuláció rendszerterve

6.1.1. Modul Felépítés

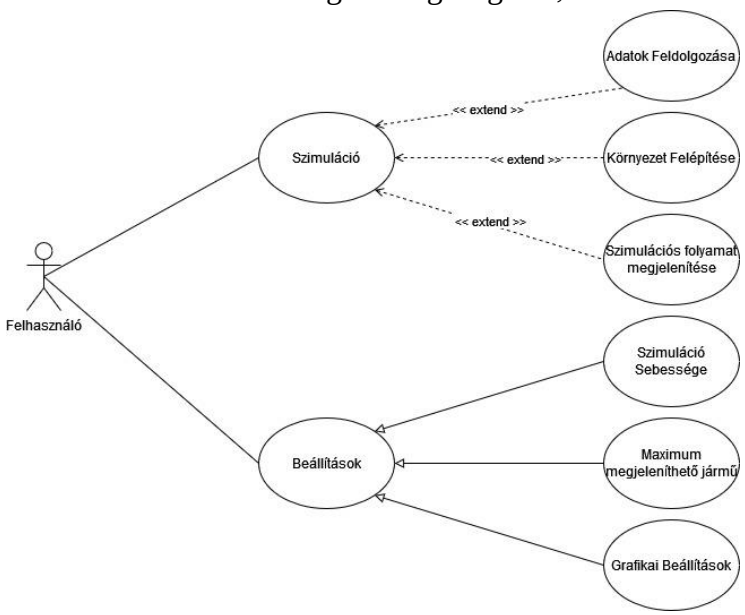
Az Unreal Engine-ben lehetőségünk van különböző modulok létrehozására, amely vegyesen tartalmazhat C++ kódot, Blueprint megvalósításokat és egyéb tartalmakat, mint képek, modellek, hangok és rengeteg más. A feladat megvalósításához ezt a modul rendszert fel fogjuk használni, hogy egy könnyen kezelhető, gyorsan tesztelhető és újra felhasználható kódbázist hozzunk létre. A kódbázis gerincét öt modul fogja alkotni:

- Adatfeldolgozási modul
- Modell kezelés modul
- Szimulációs modul
- Szenzor és szenzor háló modul
- Logikai modul

Az adatfeldolgozási modul feladata a szimulációhoz használt, kívülről érkező, előre legenerált adatokat fogja feldolgozni, és olyan formára átalakítani, amelyet majd a modell, és szimulációs modul feltud majd használni.
A modell modul feladata lesz majd a különböző viselkedési, szimulációs, érzékelő és egyéb megvalósítása, tárolása. Ebben a modulban találhatóak majd azok az osztályok, amelyek a legfontosabb szerepet játszák majd a szimuláció során, hisz ők fogják leírni azt a szabályrendszert, amely alapján a járművek követik majd egymást, reagálnak az akadályokra, a szenzorok utasításaira, illetve állapotváltozásukra, valamint, hogy milyen modell alapján fog majd maga a szimuláció futni.
A szenzor és szenzor háló modul fogja tartalmazni a felépítését, megjelenítését, illetve működési logikáját az adott intruzív vagy nem intruzív eszközöknek. Ebbe beletartozik az adott eszköz megjelenítéséhez szolgáló 3D modell, komponensek, amellyel egy járművel, illetve a szenzor hálóval kommunikálni tud, illetve egyéb logiak kód.
A logikai modul fog az útvonal keresésért, megjelenítésért és egyéb feladatokért felelni. Ő teljes egészében csak egy osztályokból álló könyvtár lesz, amely segít a szimulációs modult összekötni a többi egységgel.

6.1.2. Használati eset diagram

A szimulációi funkcióit használati eset diagram segítségével, a 6.1. ábra ismerteti:



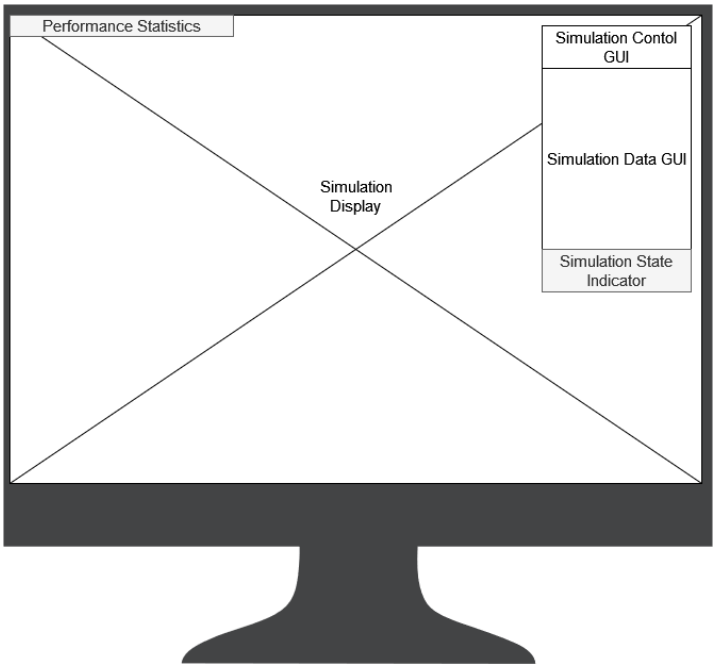
6-1. ábra: A szimuláció használati eset diagramja

A tervezés során csak egyféle szerepkörbe tartozó felhasználót azonosítottam, aki a szimulációs rendszerrel funkcionális kapcsolatba lép, ezt az aktort Felhasználó-ként jeleztem a diagramon.

A két fő használati eset a *Szimuláció* és a *Beállítások* kezelése. A *Szimuláció* során megtörténik az adatok feldolgozása, a modellek megvalósítása, illetve a szimulációs logika létrehozása. Második körben inicializálóik a környezet megjelenítése, megjelennek a járművek, a táblák, különböző grafikai elemek, mint a 3D modellek, textúrák, illetve a felhasználói interfész. Ebben a szakaszban a felhasználó tudja limitálni a GUI megjelenítését, illetve a megjelenített adatokat. Az utolsó opció a szimulációs folyamat vezérlése, illetve megjelenítése. A felhasználó képes lesz szüneteltetni, illetve teljes egészében megállítani a szimulációt.

6.1.3. Szimulációs felület drótvázas terve

Bár hivatalosan nem tekinthető UML diagramnak, de érdemes figyelembe venni az szimuláció kinézetét és viselkedését is tervezés során. Az előbb ismertetett funkciók alapján a következő úgynevezett drótvázas tervet (wireframe) készítettem el a szimulációs felületről, amely a 6.2 ábrán látható:



6-2. ábra: A szimuláció főképernyőjének drótvázas terve

A képernyőtervek szerint a főképernyőn található lesz maga a szimulációs folyamat menete. A bal felső sarokban lehet majd tájékozódni a szimuláció teljesítménye lesz majd látható, ami memória, processzor, valamint GPU használtságot fog mutatni. A jobb oldalon lesznek elhelyezve a szimuláció specifikus interfészek, mint a szimuláció irányítására szolgáló vezérlő felület, a szimulációról szóló adatok megjelenítése, mint például az út keresési algoritmus futási ideje, az adott járműre vonatkozó adatok, illetve a kiválasztott parkóló hely információi. Az utolsó egység pedig egy státusz indikátor, amely hiba esetén jelzést ad a felhasználó számára.

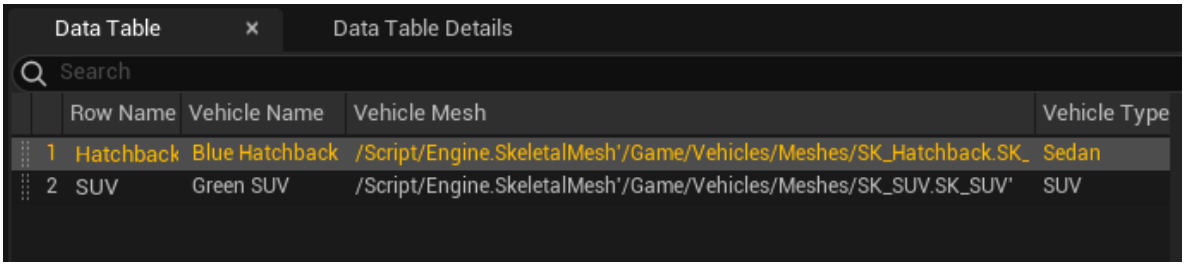
A szimuláció beállítások képernyőn (6.2. ábra) továbbra is látszik a szimulációs folyamat képe, ezzel azonnali visszajelzést adva a felhasználónak az aktuális konfiguráció megerősítéséről. Ezen képernyőn keresztül fogjuk tudni beállítani a szimulációsban megjelenítendő, illetve irányítandó járművek maximális számát, illetve különböző grafikai beállításokat érhetünk el innen, mint például textura, 3D modellek vagy akár a világítás vagy árnyékok részletességének konfigurálása. Mivel ezen menük megjelenítése és stílusa komplex feladat és a projekt előrehaladásával folyamatosan fejlődi, tényleges drótvázas terv megalkotása ebben a fázisban még nem lehetséges.

6-3. ábra: A szimuláció beállítás képernyőjének drótvázas terve

7. A MEGVALÓSÍTÁS LEÍRÁSA

7.1. Adattáblák és kezelésük

A feladat megvalósítása során, számos olyan részt, illetve komponenst találunk, amelyek több fajta paraméterezéssel rendelkeznek, viszont alapjaiban megegyeznek. Mivel rengeteg ismétlődő elem található ezért azon alapelv, hogy mindegyik elem egyedileg megvalósításra kerüljön, nem kivitelezhető, hisz ebben az esetben számos fájlt kellene kezelnünk, amely a projekt növekedése során problémát okozna. Ilyen esetek például a járművek meghatározása, illetve azon boltok és üzlethelységek leírása, amely a szimuláció elengedhetetlen része. Annak érdekében, hogy a kezelhetőséget le egyszerűsítsük, és az esetleges projekt növekedés bekövetkeztével a felhasználás gördülékenyebb legyen, az Unreal Engine úgynevezett „Data Table” megoldását használtuk fel. A rendszert legegyszerűbben egy Excel fájlhoz tudjuk hasonlítani mely sorokat, oszlopokat, illetve cellákat tartalmaz.



7-1. ábra: A járműveket tartalmazó adattábla

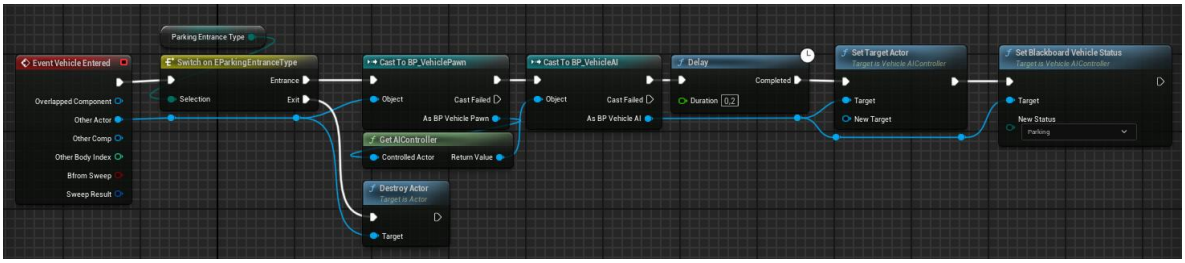
Ezeknek a tulajdonságainak köszönhetően egyszerűen és gyorsan tudjuk kezelni a projekt egyes részeit még akkor is, ha nagy mennyiségű adattal dolgozunk. Egyik ilyen terület, ahol adattáblák felhasználásra kerültek az a járművek megvalósításánál történt. Az összes vizuális (használt modell, fizikai információk) illetve szimulációs infó (Jármű típusa, megnevezése) megtalálható egy adott járműről. Így a járművek megvalósítása során csak egyetlen osztályra illetve példányra van szükségünk, és a rendszer véletlenszerűen kiválasztja a megfelelő adatokat a jármű „spawn”-olása során. A rendszert felhasználtuk még a gyalogos bejáratok megvalósítása során, mely megmondja, hogy az adott bejáratnál mely üzletekbe tudunk eljutni a leggyorsabban. Ennek a funkciónak a parkolóhely kiválasztás folyamatában van fontos szerepe, hiszen ezekkel az információkkal tudjuk majd a megfelelő parkolóhelyet kiválasztani számára.

7.2. A világban használt Aktor-ok

A következő fejezetben azon komponensek kerülnek bemutatásra, amely a szimuláció gerincét alkotják. Ezek az aktor-ok felelnek például a parkolók foglaltság jelzésének kezelésére, a megfelelő parkolóhely kiválasztására, vagy a járművek irányítását végzik.

7.2.1. Autós Be – és Kijárati aktor-ok

A személygépjárművek be-, illetve kihaladására szolgáló aktor egy fontos szereplő a szimuláció során, hiszen enélkül az osztály nélkül nem lehetne mérni, hogy mennyi autó érkezett, illetve hagyta el az épületet, hova szeretnének eljutni a vásárlók, mennyi időt töltött bent egy gépjármű. A fejlesztés során fontos szempont volt, hogy ezen osztály metódusai, illetve tulajdonágai akár C++ kód módosítása nélkül is lehetséges legyen, így a legtöbb megvalósított kódrészlet felülírható a vizuális szkriptelési felületen is. Az osztály legfontosabb elemei a bejárat típusa, amely megmondja, hogy be- vagy kijáratról beszélünk, a vezérelt „spawn" pont (mely egy későbbi szakaszban bővebben kifejtésre kerül), amelynek beállítási lehetősége bizonyos kondíciókhoz van kötve, illetve kettő darab esemény, amely egy jármű beérkezését, illetve távozását jelzi az adott aktor környezetében.



7-2. ábra: A személygépjárművek beérkezésének, illetve távozásának kezelése Blueprint felületen.

7.2.2. Gyalogos bejárati aktor-ok

A bejárati aktor egy vizuális megjelenítésre és a parkolóhelyek számontartására létrehozott osztály. Az ő feladatuk az, hogy az előző szekcióban ismertetett adattáblák alapján eltárolják, hogy egy vásárló mely üzletbe juthat el, melyek azok a parkolóhelyek, amely a bejárat közelében, vagy egy bizonyos körön belül helyezkednek el, valamint különböző, a parkolóhely aktor megvalósításának megkönnyebbítésére szolgáló metódusokkal is rendelkeznek. Akárcsak a személygépjármű bejáratot megvalósító osztály, itt is fontos volt, hogy a logikai funkcionalitást akár kódolás nélkül is elvégezhessük, a gyorsabb fejlesztés és tesztelések érdekében. Vizuális megjelenítés szempontjából a rendszer két komponenssel rendelkezik:

- Egy StaticMeshComponent – Amely a világban található vizuális reprezentációt képviseli.
- Illetve egy BoxCaputeComponent – Amellyel a bejárathoz tartozó területet tudjuk meghatározni, illetve azt, hogy mely parkolóhelyek esnek az adott bejárat felügyelete alá.

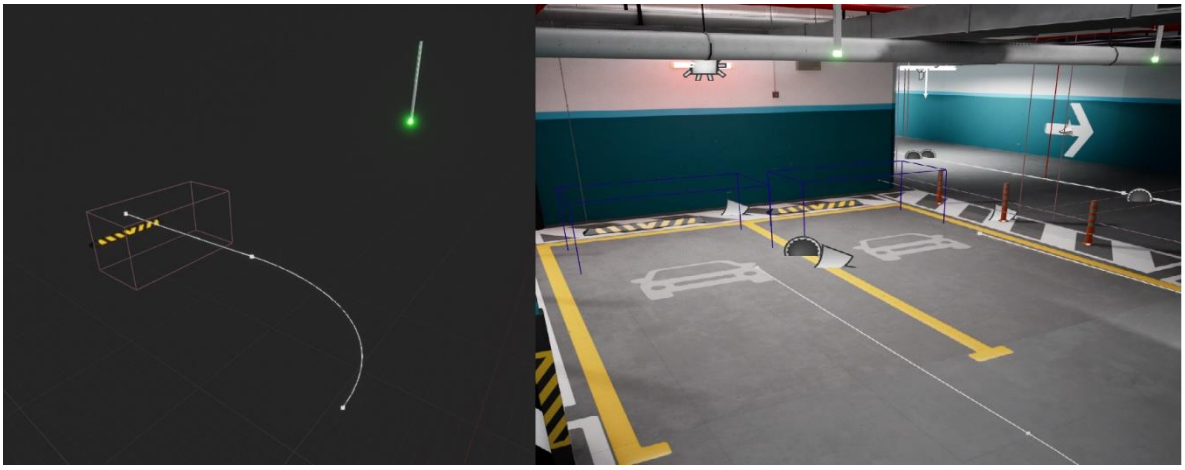
A BoxCaputeComponent-nek köszönhetően, amikor a rendszer inicializálja az aktort, vagy azt valós időben a pályán áthelyezzük akkor képesek vagyunk meghatározni, hogy ezen a területen belül található-e „ParkingSpace" típusú aktor, és ha igen, akkor azt felvesszük a

bejárathoz tartozó listába, tárolás céljával, illetve, hogy számon tudjuk tartani, hogy az adott bejárathoz mely parkolók használhatóak fel, egy esetleges keresés lefutásakor.

7.2.3. Parkolóhely aktor-ok

A járműveken kívül az egyik legmeghatározóbb vizuális, illetve logikai komponens a rendszerben maguk a parkolóhelyek. Ennek az aktor-nak rengeteg szerepe van a szimuláció során, illetve számtalan paraméterrel és funkcionalitással is rendelkezik. Az osztály öt különböző al-komponenst tartalmaz:

- Egy StaticMeshComponent típusú változó, amely képes statikus modellek megjelenítésére. Ezt a feladat során a földön található sárga-fekete parkolóhely végét jelző 3D-s modell megjelenítésére használtuk. Mivel ez a része a parkolóhelynek bárhol állítható, a modell tetszőlegesen cserélhető, vagy akár üresen is hagyható, így semmi nem jelenik meg ebben az esetben.
- Egy újabb StaticMeshComponent típusú változó, ez a kiválasztott szenzornak a fizikai reprezentációját jeleníti meg, mely alapesetben egy zöld-piros fény dióda, melynek színe a parkoló foglaltság állapotához képest változik.
- Egy BoxCollision komponenst, amely egy, a szimuláció futtatása alatt nem látható, viszont rengeteg szempontból fontos egység. Ő egy úgynevezett bounding box, amely meghatározza a parkoló hely kívánt méretét, és több metódus segítségével eseményeket fűzhetünk hozzá. Ezek az események alapjaiban C++-ban lettek megírva, de azért, hogy a későbbiekben gyorsabban és egyszerűbben lehessen tesztelni és fejleszteni, interfészen keresztül tudnak kommunikálni a Blueprint rendszerrel, így akár felülírhatjuk vagy kibővíthetjük őket úgy, hogy az ős megvalósítás érintetlen marad. A szimuláció során kettő eseményt használtunk, amikor a bounding box határába valami beleér, illetve azt elhagyja valamilyen aktor.
- Rendelkezik az aktor egy DecalComponent típusú kiegészítéssel is, amely csak vizuális, esztétikai megjelenítést hajt végre. A parkolónak megszabhatjuk, hogy az milyen típusú legyen (értsd ez alatt: mozgássérült, elektromos autóknak fenntartott hely, családi parkoló, stb…). Ez a típusmeghatározás magában a parkoló kiválasztásban szerepet játszik, viszont ezen érték alapján tudjuk azt is meghatározni, hogy a földön található felfestés milyen piktogramot jelenítsen meg, erre szolgál a decal component.
- Az utolsó elem egy SplineComponent, amely megmondja az oda parkolni kívánó járműnek, hogy mi lenne a legideálisabb ív, amellyel a parkolót meg tudja közelíteni és oda be tud állni. Ez a komponens minden egyes parkoló esetében egyedileg beállítható kézzel, illetve forgatható attól függően, hogy menetirány szerinti jobb- vagy balkezes parkolóról beszélünk.



7-3. ábra: A parkoló aktor a vizuális szkript felületen, illetve a világban elhelyezve.

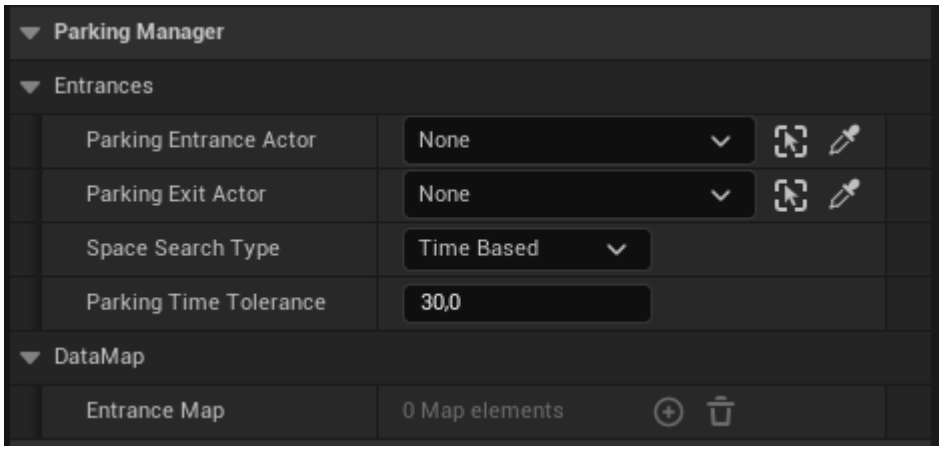
Ezek a parkolóhelyek a világban szabadkézzel vagy erre a projektre kifejlesztett osztállyal is elhelyezhetőek, illetve egyenként személyre szabhatóak különböző paneleken keresztül.

7.2.4. Jármű „Spawn” aktor-ok

Ahhoz, hogy a szimuláció megvalósuljon, szükségünk van arra, hogy különböző járművek jelenjenek meg a pályán. Erre a problémára a legegyszerűbb megoldás egy jármű „spawn” aktor, amely automatikusan véletlenszerű időközönként, vagy külső kérésre azonnal létrehoz egy új járművet, amely a szimuláció során felhasználható. Tulajdonságai ezen objektumnak meglehetősen egyszerűek és lényegre törőek. A legelső és legfontosabb egy úgynevezett „VehicleDataTable”, amely már a korábban megemlített adattáblát tartalmazza, így tudjuk, hogy milyen autók „elkészítése” lehetséges a rendszer által. A „VehicleBaseClass” változó egy osztályreferencia, amely megadja, hogy milyen objektum lesz az alapja a járműnek, milyen fizikai tulajdonságokkal rendelkezik, milyen hajtáslánc található a járműben, a hajtott kerekek számát, illetve a jármű tömegét is megszabja. A legutolsó paraméter egy logikai változó, amely a „bIsActive” névre hallgat, ezen változó segítségével tudjuk be-, illetve kikapcsolt állapotba hozni a „spawn” pontot, melynek segítségével elkerülhető az, hogy a rendszer a teljes inicializálás előtt elkezdjen járműveket megjeleníteni a pályán, ezzel befolyásolva a szimuláció eredményét. Ebből az aktorból csak egyetlenegy található meg a pályán minden időben, és referencia alapon hozzá van kötve a parkolás irányító aktorhoz (ParkingManager), amely a járművek lekérését végzi.

7.2.5. Parkolás irányító aktor-ok

A szimuláció gerincét alapul adó parkolás irányító aktor (ParkingManager), a lehető legkritikusabb pontja a projektnek. Az ő feladata a jármű „spawn” pont irányítása és a járművek lehívása, tárolja a parkolóházban található összes bejáratot, illetve kijáratot, felel a járművek megfelelő parkolóhelyének kiválasztásáért a megadott módszer szerint és vezérli a UI rendszert. Megvalósítása vegyesen, C++ kódban, illetve Blueprint szkriptelés segítségével történt, de ahogy azt már több elemnél is említettem, ezek a megvalósítások könnyen felülírhatóak, új funkciók megvalósítása érdekében, anélkül, hogy kódolni kellene azt. Az osztály több tulajdonsága tetszés szerint paraméterezhető, amely a szimuláció viselkedését, illetve kimenetét tudja befolyásolni.



7-4. ábra: A parkoló manager aktor beállítási lehetőségei a szimuláció befolyásolásához.

A szimuláció indítását követően, a ParkingManager elsődleges feladata egy olyan kulcs-értékpárt tároló lista elkészítése (FString – AentranceManager referencia), amely eltárolja egyedi kulcsként az adott üzlet nevét, illetve a hozzá tartozó bejáratot. Ezáltal egy ismétlődések nélküli, könnyen kereshető adathalmazt kapunk. Ezután a járművek „spawn” logikája következik. Arra, hogy a rendszer járművet hozzon létre két lehetőségünk van:

- Véletlenszerű – A járművek véletlenszerűen a felhasználó által megadott időintervallumon belül jönnek létre.
- Külső esemény hatására – Ez abban az esetben hajtódik végre, amikor a szimuláció a predikció miatt kell, hogy létrehozzon járművet.

A komponens futása a jármű létrehozásától függően halad tovább. Amennyiben a jármű külső esemény alapján lett létrehozva, akkor ő már rendelkezik az összes szükséges szimulációs adattal, ideértve a bent tölteni kívánt időt, illetve az úti célt. Viszont véletlenszerű létrehozásnál, ezeket az értékeket nekünk kell biztosítani, ezt az AI kontroller fogja elvégezni számunkra, amit később bővebben megismerünk. Ezután a járművek

különböző eseményekre kerülnek fel, mellyel például mérhetjük az átlagos sebességét, a megtett utat, a belépéstől a parkolóhelyig eltelt időt, a tényleges bent töltött időt, illetve a parkolótól a kijáratig eltelt időt. Amint ezen folyamatok lefutottak, a járművek elkezdenek mozogni a legelső bejárat felé, az AI kontroller és a Behavior Tree segítségével. Amikor a gépkocsi megérkezik az egyik irányított bejárathoz, akkor a rendszer lekéri, hogy melyik üzletbe szeretne eljutni. Ekkor az aktor a már előzőleg létrehozott listából megkeresi a célpontot, és lekéri, hogy melyik bejárathoz tartozik, amikor ez megtörtént, a bejáratot reprezentáló osztálytól egy metódus segítségével átveszi az összes szabad, még használatlan parkolókat, amelyet felhasznál a legoptimálisabb parkolóhely kiválasztására. A kiválasztásra a jelen implementáció szerint kettő darab lehetőségünk van, de a fejlesztési irányoknak köszönhetően ez bármikor bővíthető a felhasználó által, akár C++ kódban, akár a vizuális szkriptelési felületen. A két jelenlegi megvalósítás:

- First Free – Amely egy egyszerű feltételes keresés, a legelső üres parkolóhelyet adja vissza a rendszernek, és oda küldi tovább a személygépkocsit.
- Time Based – A rendszer a bent tölteni kívánt idő alapján határozza meg, hogy a jármű melyik parkolóhelyet kapja meg. Így azokat a látogatókat, akik kevesebb időt töltenek bent, próbálja a lehető legközelebb helyezni a bejárathoz, ezzel gyorsítva a parkolóhelyek cseréjét. Lehetséges egy „Parking Time Tolerance” nevű változó segítségével beállítani azt, hogy mikor mondja azt a rendszer, hogy az adott jármű prioritást élvez ebből a szempontból és érdemesebb számára egy közeli hely kiválasztása, vagy egy messzebb lévő parkolót kapjon.

Amikor a járművek a parkolót elhagyják, akkor a Parking Manager szolgáltat számukra egy olyan kijáratot, amely a jelen pozíciójuktól könnyen meg tudnak közelíteni. Amikor ezt a pontot elérték, értesítik erről a vezérlőt, ami befejezi a különböző adatok mérését, majd ezeket exportálja egy CSV fájlba, és amikor az írás sikeresen megtörtént, megsemmisíti a járművet, ezzel felszabadítva a memória tartományt, illetve a CPU időt. A szimuláció végeztével az osztály minden eddigi cache-elt adatot is ebbe a kimeneti fájlba menti el.

7.3. Járművek AI vezérlése

Ahhoz, hogy a járműveket irányítani tudjuk, szükséges valamilyen intelligencia, amely képes a megfelelő útvonalat meghatározni A, illetve B pont között. Erre a célra az Unreal-ben található AI keretrendszert használtuk fel, melyet kibővítettünk a számunkra szükséges funkcionalitásokkal és változókkal.

7.3.1. Unreal Blackboard Komponens

Az első része a megvalósításnak az úgynevezett AI Blackboard komponens, amely nem más, mint egy kulcs-érték párból álló konténer. Ebben a fájlban adhatjuk meg azokat a változókat, amelyet a járműnek ismernie kell a szimuláció futása során, illetve ezen változók módosításával érhetünk el különböző viselkedési és végrehajtási rutinokat. A helyes megvalósítás kritikus fontosságú, hiszen később erre az osztályra épül az AI vezérlés megvalósítása. A szimuláció során a járműnek szüksége van a következő paraméterekre:

- egy referenciára, ami saját magát tárolja, ez az úgynevezett SelfActor objektum típusú változó
- egy SimulationStatus enum alapú változóra, amely a jármű jelenlegi státuszát tárolja, amely a viselkedés kiválasztásában jelent segítséget
- egy ParkingTime, lebegő pontos változó, amely megmondja, hogy a jármű mennyi ideje parkol az adott területen
- egy ParkingSpace nevű változó, amely az általunk készített, „ParkingSpace” típusú változó. A szimuláció alatt a jármű erre a helyre szeretne eljutni, illetve erről a helyről szeretne távozni majd a parkolóházból. Ennek a helynek az ismerete elengedhetetlen az útvonal alkotáshoz.
- illetve egy TargetActor nevű változóra, amelyet a szimuláció különböző helyzetekben használ. Ez a változó lehet az előttünk haladó jármű, a parkoló be, illetve kijárata, vagy más a navigációt szolgáló segéd aktor.

Amikor a megfelelő objektumok deklarálásra kerültek a Blackboard komponensben, akkor felhasználhatjuk őket mind C++, mind a Blueprint környezetben, illetve a Behavior Tree keretrendszerben, amely ezekből az adatokból képes különböző viselkedési formát választani a jármű számára.

```
TargetActorKeyName = FName("TargetActor");
ParkingTimeKeyName = FName("ParkingTime");
ParkingSpaceLocationKeyName = FName("ParkingSpace");
SimulationStatusKeyName = FName("SimulationStatus");
```

7-5. ábra: Az AI által használt kulcs-érték párok C++ megvalósításban.

7.3.2. Unreal Behavior Tree Komponens

Az Unreal Engine-ben található egy flexibilis és robosztus AI keretrendszer, melynek a része a Behavior Tree is. Ennek segítségével gyorsan és egyszerűen lehet tesztelni, és fejleszteni különböző AI rendszereket. A fa balról jobba, illetve fentről lefelé hajtja végre az utasításokat, és különböző kondíciók, illetve változók értékei alapján hozza meg a döntést, hogy melyik ágat futtassa le az adott pillanatban. A rendszer tartalmaz úgynevezett node-

okat, melyek felhasználhatóak, de a feladat sajátossága miatt, szükséges volt ezen lista kibővítése az általunk használni kívánt funkciókkal. Az AI gráf erősen függ az úgynevezett „Task Node”-októl (feladatok). Ezek azok a különböző logikai megvalósítások, amelyek arra szolgálnak, hogy a szimuláció zökkenőmentesen működjön. Ahhoz, hogy elérjük a kívánt eredményt, öt darab egyedi „Task” node került megvalósításra, melyek különböző feladatot hajtanak végre. Ebből a legelső és a legfontosabb az úgynevezett „BTTASK_MoveVehicle”, amely a jármű mozgásáért felel. Ez a feladat egy FVector3 típusú paramétert kap változónak, amely az úti cél koordinátája a világban. Ez a megvalósítás tudja a jármű pontos elhelyezkedését, illetve irányát minden egyes Tick-ben (frissítési időszakban), és ezen adatokat a feladat meghívásakor átadja az útvonal tervező algoritmusnak, mely megadja nekünk, hogy hogyan jutunk el a kívánt pontig. Ezt az útvonal eredményt, az AI keretrendszerben megtalálható MoveTo() metódus megkapja, illetve az UVehicleMovementComponent is, amely a mozgásért felel. Ezt a megvalósítást a szimuláció és a jármű AI futási ideje alatt számtalan alkalommal használjuk, mint például a bejárati sorompóhoz való mozgás, a sorompótól a parkolóhelyig való mozgás vagy a parkológarázs elhagyása során is. A következő két megvalósítás viszonylag egyszerű, a „BTTask_FindEntrance”, illetve „BTTASK_FindExit” feladatoknak a lényege, hogy a be- és kijáratot lekérjék a ParkingManager aktor-tól, így a referenciáknak köszönhetően megkapható ezeknek a elhelyezkedési koordinátái. A negyedik osztály is egy egyszerűbb megvalósítás, melynek feladata az, hogy az AI szimulációnak a státuszát változtatni tudjuk. Ez a task node, bemenetként megkapja az új szimulációs státuszt, és a referenciaként tárolt járműnek beállítja azt, így ahol például az éppen érkező autónak a státuszát „Moving To Entrance” (Bejárat megközelítése) módról „Moving To Sparking Space” (Parkolóhely megközelítése) módra tudjuk váltani, ezzel akár befolyásolva a jármű viselkedését.

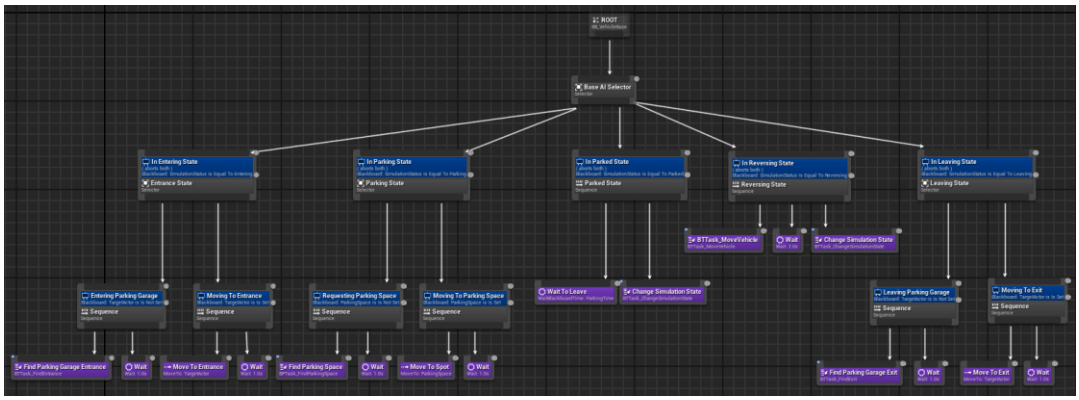
A jármű viselkedése alapvetően öt fő ágból áll, amelyek különböző alágakra bontakoznak ki, a meglévő kondíciók alapján. Az öt főág balról jobbra haladva a következő:

- Entrance State – Melynek fő követelménye, hogy a járműnek a státusza „Entering” állapotban legyen. Ez az eset akkor következik be, amikor a jármű be szeretne lépni a parkolóházba. Az ág képes önmagának és az alárendelt ágainak a futásának a megszakítására abban az esetben, ha ez a státusz ezen idő alatt megváltozna. Ezután két ág található, amelyből az első az „Entering Parking Garage”, ahol a fent említett „BTTask_FindEntrance” feladat megadja a jármű számára a bejárat koordinátáját, és a Blackboard-ban található „TargetActor” értékét null-ról a bejáratra állítja, majd ezt követően lefut a „BTTASK_MoveVehicle” és megkezdődik a jármű mozgása, és állapota „Parking”-ba kerül.
- Parking State – Ez a státusz akkor lép életbe, amikor a jármű elért a bejárai sorompóhoz, és parkolni szeretne. Ebben az esetben is két ág található a főág alatt, a

bal oldali megkéri a rendszert, hogy biztosítson neki egy parkolóhelyet, ezt a ParkingManager aktor végzi el, és a visszatérési értéke a rendszer szerinti legoptimálisabb parkoló hely, majd a Blackboard-on a ParkingSpace kulcsot beállítja annak. A jobb oldali főág a parkolóhely felé történő mozgást végzi, melynek alapját már az előző részben megismertük. Fontos, hogy ez a rész csak akkor fog lefutni, amikor a jármű státusza „Parking" és a ParkingSpace kulcs értéke nem null. Amikor a jármű elérte az úti célját, státusza „Parked"-ra változik.

- Parked State – Ez az állapot a legegyszerűbb, amely a fában található. Alágakkal nem rendelkezik, csak két darab „task node"-dal, melynek egyike egy úgynevezett „Wait Node" mely szimulálja a jármű parkolóban történő várakozását. Amikor ez az időzítő letelt, a jármű állapota „Reversing" módba kerül.
- Reversing State – A jármű a ParkingSpace aktor-ban található SplineComponent segítségével kitolat a parkoló helyről, a mozgás befejeztével vár két másodpercet majd státuszt változtat „Reversing" státusból „Leaving"-re.
- Leaving State – Ez a státusz megfelel az „Entrance State"-nek. Megkeresi a kijáratot majd a TargetActor kulcsnak ezt átadja. Amikor ez megtörtént a jármű mozgása megkezdődik, és elhagyja a parkolóházat.

Ahogy a szimuláció halad, illetve a jármű állapota változik, úgy futnak le a különböző ágak a viselkedésfában. A fa kellő részletessége miatt, a járművek képesek rengeteg szituációra reagálni, illetve megoldást találni arra.



7-6. ábra: Az AI és a szimuláció által használt viselkedési fa, amely reszponzív a szimuláció változásaira.

7.3.3. Unreal AI Vezérlő és A* megvalósítás

Ahhoz, hogy a fent említett Behavior Tree komponensünket használni tudjuk, létre kell hoznunk egy úgynevezett „AI Controller" osztályt, amely képes annak futtatására. Ez az osztály nem csak a viselkedési fa futtatásáért felelős, hanem ő tartalmazza a különböző komponenseket is, amelyek a járművek működéséhez elengedhetetlenek. Ilyen például az

„UAIPerceptionComponent” komponens, amely a jármű látásáért felelős, meghatározva, hogy mekkora a látószög, a maximális látott távolság, illetve a jármű mekkora távolságtól kezdve „felejti el”, hogy látott valamit. Ezeken felül tartalmazza még a szaktársam által már bemutatott „UVehicleAvoidanceComponent” komponenst, amely az ütközések megelőzésére szolgál, illetve a „UVehicleFollowingComponent” amely a követési logikáért felel. Az osztály még tartalmaz olyan metódusokat, amelyek segítségével akár Blueprint-en, akár a viselkedés fán keresztül tudjuk befolyásolni egy jármű állapotát. Ezek a metódusok a fent említett kulcs-érték párokat módosítják.

Ahhoz, hogy a Behavior Tree-vel való irányítás megvalósítsuk, szükséges, hogy felülírjuk az ős osztályban megtalálható „void OnPossess(APawn* InPawn)” metódust, melynek lényege, hogy a világban megtalálható egy virtuális reprezentáció fölött (ebben az esetben a személygépjármű) átvegye az irányítást. Abban az esetben, ha a bemeneti változó (az irányítani kívánt jármű) érvényes (nem null pointer és nem áll szemétgyűjtés előtt) akkor lehetőségünk van a beépített „BlackboardComponent”, illetve „BehaviorComponent” beállítására. Ezekben az esetekben meg kell adnunk az általunk elkészített, és használni kívánt fájlokat, majd ezek után a „BehaviorComponent”-ben található beépített metódus segítségével, a void StartTree(UBehaviorTree& Asset) referenciával elindítjuk a fa működését.

Ezen felül az AI részhez tartozik az útkeresés megvalósítása is. Az Unreal Engine ugyan tartalmaz beépített útkeresési algoritmust és navigáció támogatási rendszert, de mi úgy döntöttünk, hogy ezt felülírva, az A* algoritmust fogjuk használni. Fontos leszögezni, hogy a motor rendelkezik az alapvető ős, illetve interfész osztályokkal (C++ template structure), ahhoz, hogy ezt meg tudjuk tenni, de tényleges implementáció nem található, így ez a mi feladatunk. Ahhoz, hogy a rendszert használni tudjuk, létre kell hoznunk két osztályt, amely az „APathFindingComponent” illetve az „ANavigationData” leszármazottja. A fent említett osztályok közül, az első feladata a különböző szomszédossági esetek vizsgálata, az adott út súlyozott értékének kiszámítása, illetve egyéb segédfunkciók megvalósítása:

- float GetHeuristicCost() – Ez egy megközelítőleges súlyozott érték az A, illetve B pont közötti eljutásra. Erre számtalan logikai megoldás létezik, a feladat megoldása során az úgynevezett „Manhattan Distance” alapján számolt érték lett visszaadva.
- bool IsTraversalAllowed() – Megadja, hogy lehetséges-e az útvonal létrehozás az A, illetve B pont között. Ez a metódus figyelembe veszi a lehetséges akadályokat, mint például az úgynevezett „Blocking Volume”-okat, melyek bizonyos helyek lezárására használtunk fel a projekt során, illetve a „Direction Volume”-okat mellyel meghatároztuk a lehetséges haladási irányokat.

- int32 GetNeighbourCount() – Megadja hogy a paraméterként kapott referencia „node”, hány közvetlen szomszéddal rendelkezik.
- FNodeRef GetNeighbour() – Visszaadja a bemeneti index alapján lekért szomszédot a szomszédossági listából.

Ezen funkcionalitások implementálása meglehetősen egyszerű, hisz alapjait a tanulmányaim során megismerhettem különböző kurzusokon keresztül. Amikor ez az osztály elkészült, megvalósításra került a navigációs adatkezelő, illetve feldolgozó osztály. Ez felelős azért, hogy a garázsban elhelyezett úgynevezett „NavMesh Volume” szegmenseit, illetve adatait fel tudjuk dolgozni, és azokat felhasználhassuk az A* megvalósítással. Az „ANavigationData” osztály leszármazotja felel azért, hogy a pályát pontokra bontsa, majd megkeresse a legrövidebb utat, a korábban bemutatott metódusok segítségével. Két metódust kell implementálni a feladat elvégzéséhez:

- bool ProjectPoint() – Feladata, hogy a bejárható területet kisebb pontokra, úgynevezett „node”-okra bontsa, és azokat különböző tulajdonságokkal ellássa, mint például a magasság értéke, megközelíthetőség.
- static FPathFindingResult FindPath() – A létrehozott pontok felhasználásával, illetve a korábban implementált egyéb funkciók segítségével visszaadja a lehető legrövidebb utat, amellyel eljuthatunk A pontból B pontba.

7.4. UI és Interakció megvalósítása

A következő szekció röviden összefoglalja a UI, illetve a hozzáhasznált komponensek és interfészek megvalósítását és implementálását, amelynek köszönhetően különböző adatok nyerhetőek ki egy adott járműről, illetve a szimuláció folyamatát befolyásolhatjuk.

7.4.1. Interakció Interfész és aktor komponens

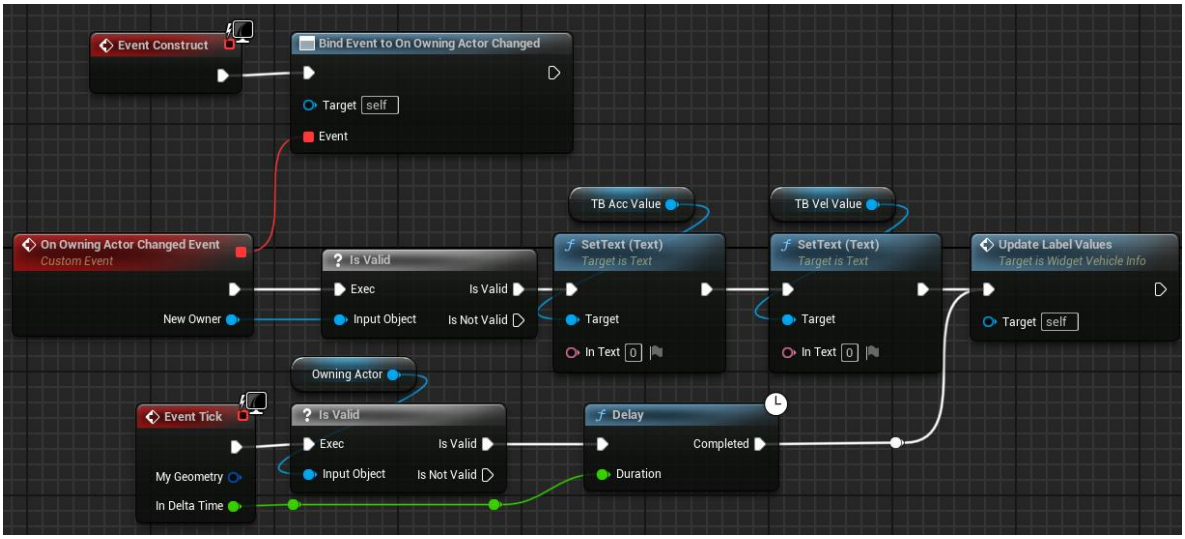
Az interakció része a szimulációnak egy viszonylag egyszerű és gyors megoldás, amit személy szerint számos saját projektben is használtam már. A megvalósítás első lépése egy interfész, amely az „IParkingUsableInterface” nevet viseli és három metódus implementálását írja elő (fontos megjegyezni, hogy ezek a metódusok, mint rengeteg másik, könnyen felülírható a vizuális szkript felületen), és alapja C++ kódban van meghatározva:

- void OnInteracted(APawn* InstigatorPawn) – A metódus akkor fut le amikor a felhasználó rákattint egy gépjárműre, paraméterként az interakciót kezdeményező objektumok kapja meg.
- void OnBeginFocus() – A funkció vizuális visszajelzést ad a kiválasztott járműről. Amikor a felhasználó a kurzort egy adott járműre helyezi, de még nem kattintott rá,

a rendszer egy fehér körvonallal kijelöli számára a modellt. Ezt a kiemelés egy úgynevezett Post Process Shader segítségével valósítható meg.

- OnEndFocus() – A fent említett kiemelést tünteti el a járműről.

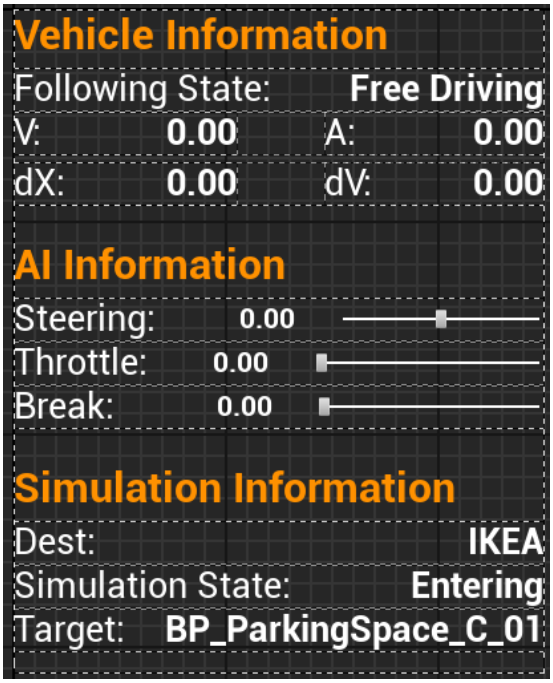
Ehhez a megvalósításhoz tartozik még három osztály, az egyik az úgynevezett „UVehicleWidgetComponent”, amely egy aktor komponens, illetve egy „UVehicleWidget”, amely egy UI elem típus, amire különböző grafikákat és funkciókat tudunk kötni. Az „UVehicleWidget” osztály tartalmaz egy eseményt, amely akkor fut le, amikor a kiválasztott jármű és annak adatai megváltoztak, egy metódust, amivel a kiválasztott járművet megváltoztathatjuk, illetve egy privát mezőt, ami a jelen pillanatban kiválasztott járművet tárolja. A „UVehicleWidgetComponent” osztály csak a világban való megjelenítésért felel, és beállításai könnyedén elérhetőek a Blueprint felületen. Az utolsó osztály a „UParkingInteractionComp” mely a kiválasztás logikát végzi, ellenőrzi, hogy a felhasználó ténylegesen egy járműre kattintott, illetve értesíti a megfelelő UI elemeket ezen változásról. A háttérben meghúzódó logika egy egyszerű „line trace detect” megvalósításon alapul. A line trace használata és részletes leírása a szaktársam Fülep Fanni, megegyező című szakdolgozatában található. Amikor a felhasználó kattint, akkor egy „line trace”-t vetítünk ki a nézőpont és a kurzor között, majd amikor ez a vonat megszakad, akkor ellenőrizzük, hogy a szakadást egy olyan aktor okozta-e, amely implementálja a fent említett „IParkingUsableInterface” interfészt. Amennyiben ez igaz, akkor meghívjuk ezen az aktor-on az OnInteracted() metódust, amely lefuttatja a megvalósított logikát, majd utasítást ad ki a UI felé, hogy adatváltozás történt. Ekkor a „UVehicleWidget” osztályban lefut a „SetOwningActor()” metódus, illetve a már előzőleg tárgyalt esemény, és a változások megjelennek a felhasználói felületen.



7-7. ábra: Az interfész és az általunk megírt esemény implementálása egy “UVehicleWidget” osztályban.

7.4.2. UI és Interakció Interfész megvalósítása

A UI elem a már fent említett „UVehicleWidget” osztályon alapul. Mivel ez az objektum tartalmazza a jelen pillanatban kiválasztott és megfigyelt járművet, és tárolja azt egy referencia értékben, könnyen kinyerhetőek az általunk fontosnak ítélt adatok. Ezen kívül elérhetőek a szimuláció állapotát leíró adatok is. A UI felület dizájn felépítése után, egy a már szakmából ismert módon, úgynevezett „binding”-ok segítségével kerülnek az értékek a megfelelő helyre. Ezen értékek változásakor a rendszer automatikusan frissíti őket, amely a grafikai elem újrarajzolását eredményezi és ezzel a legfrissebb információt nyújtja a felhasználó számára.



7-8. ábra: A UI elem végleges dizájnja.

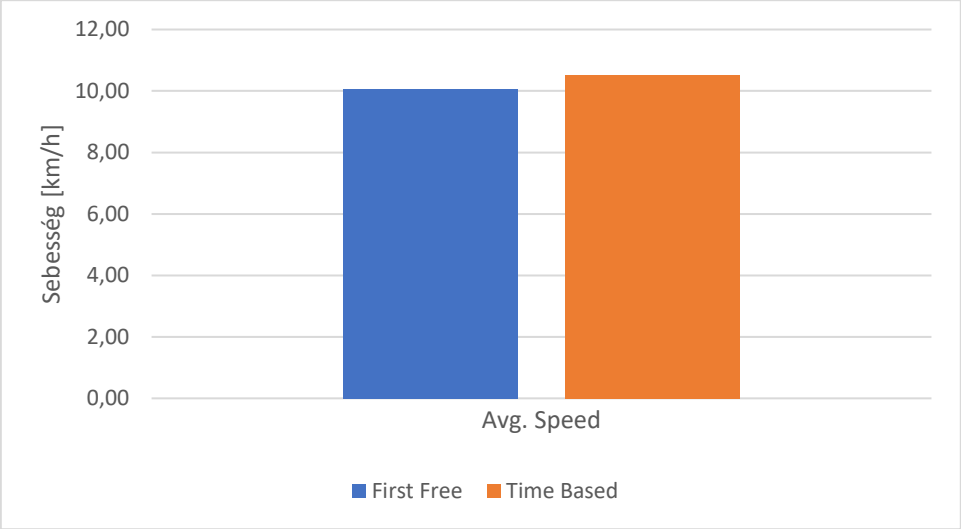
8. TESZTELÉS

A tesztelés során a két eddig meglévő parkolóhely kiválasztását hasonlítottam össze, azokból az adatokból, amelyet a program futása végén a rendszer a lemezre ír. Ezen kívül az útkeresési algoritmusok hatékonyságát vettem figyelembe.

8.1. Megvalósított parkolóhely kiválasztás metódusainak tesztelése

A kinyert adatokból több grafikon is készült, mellyel szemléltethető az egyes algoritmusok működése, erősségei, illetve gyengeségei. A tesztek mindkét esetben, harminc darab személygépjárművet szimuláltak, melyek értékei a külső, predikciós rendszerből kapták meg úti céljukat, illetve egyéb tulajdonságaikat is. Három értéket vettem alapul, amely a valóságban is tükrözheti egy ilyen rendszer teljesítményét:

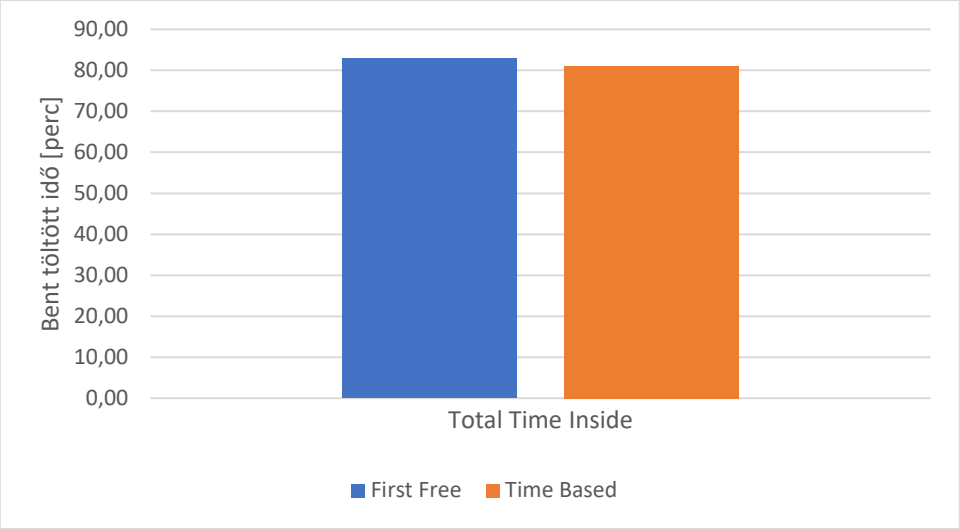
- Az átlagos sebesség, amellyel a gépjármű vezetők közlekednek. Ennek fontos szerepe lehet a mindennapokban, hisz abban az esetben, ha a rendszer jól működik, akkor nem alakul ki torlódás a gépjárművek között, nő a maximális sebesség, illetve gyorsabban elérhető a vásárló célpontja. Mint ahogy ez a lenti grafikonon is látszik, a két megvalósítás között érdemi, ténylegesen érezhető különbség nem tapasztalható.



8-1. ábra: Az átlagos sebesség különbség, a két megvalósított megoldás közt.

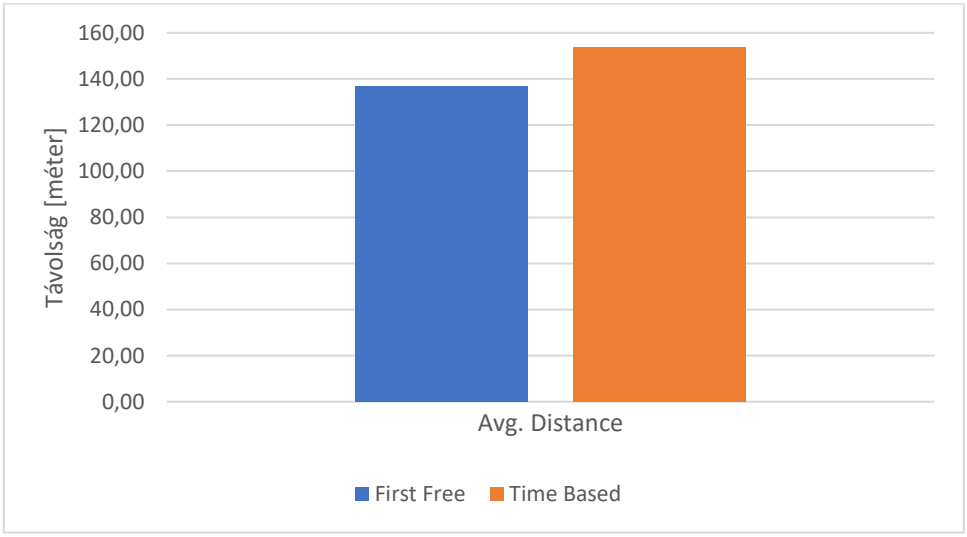
- A teljes bent töltött átlagos időszak, mely értéket az autó behajtásától egészen a kijárat átlépéséig számoljuk, melynek mértékegysége a perc. Ezen érték tud minket arról tájékoztatni, hogy egy jármű átlagosan mennyi időt tölt a parkolóház területén. Itt látható, hogy az úgynevezett „Time Based" megvalósítás, mely a parkolóhely kiválasztásánál figyelembe veszi azt is, hogy mennyi időt szeretne a vásárló bent tölteni, majdnem két perces előnyt nyújt a vásárlók számára. Ez az idő egy átlagos felhasználó számára igaz nem tűnhet soknak, viszont a parkolóház szempontjából jelentős lehet, hiszen, ha vevőnként átlagosan két percet sikerül rövidíteni a bent

töltött időből, akkor sokkal gyorsabban lehet a vásárlókat cserélni, ezzel elősegítve a rendszer eredményességét, a felhasználók elégedettségét, illetve az üzletek profit maximalizálását.



8-2. ábra: Az átlagos teljes idő különbség, a két megvalósított megoldás közt, amit egy jármű bent töltött.

- Az utolsó összehasonlítási szempont az az átlagosan megtett teljes út, mely szintén a parkolóház bejáratától, annak kijáratáig mért a program. Ezen értéket rengeteg külső elem tudja befolyásolni, mint például a szabad parkolók és az üzlethez vezető bejárat elhelyezkedése, illetve a parkolóházból kivezető kijáratok pozíciója. Ahogy az a lenti grafikonon is látható, ebben az esetben a „Time Based" metódus körülbelül 30 méter extra távolságot jelent átlagosan a vezetők számára. Ennek oka a fent említett változók miatt lehetséges.



8-3. ábra: Az átlagos teljes megtett távolság, a két megvalósított megoldás közt, amit egy jármű tett meg bent.

8.2. Az útkeresési megvalósítások összehasonlítása

Az útkeresési algoritmusok között a megvalósításon kívül számottevő különbség nem található, illetve nem érzékelhető a program futása közben sem. Mind a beépített megoldás, mind az A* képes a feladat elvégzésére, körülbelül azonos futási idővel, illetve memória használattal, melyek egyetlen befolyásoló tényezője az egyszere a szimulációban lévő gépjárművek száma. Az alábbi táblázatban megtalálható a beépített, illetve az A* megvalósítás közti minimális különbség, abban az esetben, ha a rendszer 30 járművel dolgozik egyidejűen:

8-1. táblázat: Az útkeresési algoritmusok teljesítménybeli összehasonlítása.

30 darab autó esetén, egy járműre	**Unreal Engine 5 Recast NavMesh**	**Custom A* Pathfinding**
Memória használat	0.211 MB	0.226 MB
CPU Idő	0.013 %	0.011 %
Futási Idő	0.0489 s	0.0415 s

Ezek a tesztek az általam használt eszközön lettek elvégezve, a beépített Unreal Profiler program segítségéven, A hardware egy 8 magos 16 szálas @ 4.2 Ghz órajelen futó processzorral, illetve 16 GB DDR4 @ 3600 Mhz RAM-mal rendelkező számítógép volt. Ahogy az látszik, számottevő eltérés nem található, így bármelyik megoldást is választjuk, biztosak lehetünk abban, hogy számunkra megfelelő és jó eredményt kapunk vissza.

9. TARTALMI ÖSSZEFOGLALÓ

A feladat során ismertettem a mostani parkolási rendszereket, illetve parkolási szokásokat. A különböző típusú, illetve fajtájú szenzorok összehasonlítása után arra a konklúzióra jutottam, hogy a feladat megoldásához nem fogok speciálisan egy típust kiválasztani, így a szimulációban mind az intruzív, mind a nem intruzív szenzor típusok megtalálhatóak lesznek.

Az útkeresés problémájára számos közismert algoritmust hasonlítottam össze, melyben ismertettem az előnyeiket, illetve a velük járó hátrányokat. A Dijkstra algoritmus a mai napig egy alapköve az ilyen feladatok megoldásának, de az A* a teljesítménybéli, illetve a futási időben történő előnye miatt a projekt elkészítéséhez ezt használtam fel.

Ezután a rendszerterv megvalósítása közben körvonalazódtak azon elemek, illetve technológiák, melyek felhasználásra kerültek a projekt megvalósítása során. Ilyen pont volt például az Unreal Engine 5 kiválasztása, mely alapja a szimulációnak.

A szimuláció megvalósításához létrehoztam számos különböző komponenst, amelyek nagyban megkönnyítették a munkámat. Ezek számos formát öltöttek a fejlesztés során, legyen az adattáblák kezelése, a járművek virtuális megjelenítése vagy akár a felhasználó által használt grafikus interfész. Munkám során jobban belátást nyertem a motorban megtalálható AI rendszerbe, melynek robosztus felépítését saját osztályokkal kiegészítve beépítettem a feladatba. Ennek köszönhetően egy reszponzív, gyors és könnyen átlátható vezérlés jött létre a járművek mozgatására, amely az A* útkeresési algoritmust használja arra, hogy a jármű eljusson céljához.

Ahhoz, hogy a rendszer értelmet nyerjen, implementáltam több fajta lehetséges parkolóhely kiválasztási metódust, melyek figyelembe veszik azt, hogy a járművezető hova szeretne eljutni, illetve, hogy körülbelül mennyi időt szeretne bent tölteni. Ezen információk, illetve a parkolóház telitettsége alapján, a rendszer képes megmondani, hogy a felhasználó melyik parkolóba kerüljön.

A grafikus felhasználói felület elengedhetetlen része volt a projektnek, hisz enélkül képtelenek lettünk volna a szimulációs adatok megjelenítésére. Ezen megvalósítás egy általam már korábban használt és bevált módszert alkalmaz, melynek eseményvezérelt természete lévén könnyedén a projekthez lehettet igazítani.

A tesztelés során a szimuláció különböző részei kerültek megfigyelés alá, mint például a beépített útkeresési algoritmus és az A* közötti különbségek teljesítmény szempontjából, illetve a megvalósított parkolóhely keresési metódusok különböző eredményei.

10. Abstract

As part of the assignment, I described the current parking systems and parking habits. After comparing the different kinds of sensors, I concluded that I will not choose a specific type for this task, so that both intrusive and non-intrusive sensor types will be included in the simulation.

For the path finding problem, I have compared several well-known algorithms, describing their advantages and disadvantages. Dijkstra's algorithm is a staple for solving such problems to date, but due to A*'s advantage in performance and runtime, I have chosen to use it for this project.

Then, as the system design was implemented, the elements and technologies that were used to implement the project were outlined. One such point was the selection of Unreal Engine 5 as the basis for the simulation.

I created several different components to implement the simulation, which made my work much easier. These took many forms during the development, whether it was the management of data tables, the virtual representation of vehicles or even the graphical interface used by the user. During my work, I gained more insight into the AI framework in the engine, with its robust architecture and its own classes. This resulted in a responsive, fast and straightforward approach to control the movement of vehicles, using the A* path finding algorithm to get the vehicle to its destination.

To achieve a system that is comprehensible, I have implemented several types of possible parking space selection methods that consider where the driver would like to arrive and approximately how much time he or she would like to spend inside. Based on this information, and the occupancy of the parking garage, the system can tell the user which parking space to use.

The graphical user interface was an essential part of the project, without it we would have been unable to display the simulation data. This implementation uses a method that I have used and tested before, which, due to its event-driven nature, can be easily adapted to the project.

During the testing, different parts of the simulation were observed, such as the differences in performance between the built-in path finding algorithm and A*, as well as the different results of the implemented parking space search methods.

11. IRODALOMJEGYZÉK

[1] A. Tóth, „A Budapesti P+R Parkolás Informatikai Támogatása Applikáció Fejlesztésével”, Budapesti Műszaki és Gazdaságtudományi Egyetem, Budapest, 2016.

[2] „56/2017. (III. 20.) Korm. rendelet egyes kormányrendeleteknek az »okos város«, »okos város módszertan« fogalom meghatározásával összefüggő módosításáról”.

[3] R. Giffinger, C. Fertner, H. Kramar, R. Kalasek, N. Milanović, és E. Meijers, *Smart cities - Ranking of European medium-sized cities*. 2007.

[4] S. Dirks és M. Keeling, „A vision of smarter cities”, IBM Global Business Services, GBE03227-USEN-04, 2009.

[5] *Az Európai Parlament és a Tanács 2010/40/EU irányelve az intelligens közlekedési rendszereknek a közúti közlekedés területén történő kiépítésére, valamint a más közlekedési módokhoz való kapcsolódására vonatkozó keretről*. Strassburg, 2010.

[6] C. Csiszár és Z. P. Sándor, *Közlekedési informatika*. BME Tanárképző Központ, 2014.

[7] S. Shaheen, „Smart Parking Management Field Test: A Bay Area Rapid Transit (BART) District Parking Demonstration”, 2005, Elérés: okt. 08, 2021. [Online]. Elérhető: https://escholarship.org/uc/item/6d58554x

[8] T. Daiki, G. Siegler, P. Szlávi, és L. Zsakó, *Modellezés és szimuláció*. Budapest: Eötvös Loránd Tudományegyetem Informatikai Kar, 2012. Elérés: dec. 08, 2021. [Online]. Elérhető: https://dtk.tankonyvtar.hu/handle/123456789/3937?show=full

[9] G. Geda, *Modellezés és szimuláció az oktatásban*. Kempelen Farkas Hallgatói Információs Központ, 2011.

[10] D. Shoup, „Cruising for parking”, *Transp. Policy*, köt. 13, o. 479–486, febr. 2006, doi: 10.1016/j.tranpol.2006.05.005.

[11] D. A. Burdette, „An Investigation of Advanced Parking Information Systems at Airports”, *Compend. Grad. Stud. Pap. Adv. Surf. Transp. Syst. 1999*, o. 79–112, 1999.

[12] Erő Z., Bardóczi S., Germán T., Ekés A., Sipos Z., és Fehérvári Z., „Újlipótváros parkolásfejlesztési tanulmány”. 2015. Elérés: nov. 24, 2021. [Online]. Elérhető: https://www.kozszolgaltato.bp13.hu/letoltheto-dokumentumok/

[13] P. R. Verma és M. Verma, „Techniques for Smart & innovative parking, critical observations and Future Directions: A review”, in *2015 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, Kumaracoil, India, dec. 2015, o. 431–437. doi: 10.1109/ICCICCT.2015.7475317.

[14] „OpenCV 2.4.13.7 documentation”. https://docs.opencv.org/2.4.13.7/ (elérés dec. 03, 2021).

[15] K. M. Sajjad, „Automatic License Plate Recognition using Python and OpenCV”, o. 5.

[16] M. Noto és H. Sato, „A method for the shortest path search by extended Dijkstra algorithm”, in *SMC 2000 Conference Proceedings. 2000 IEEE International Conference on Systems, Man and Cybernetics. „Cybernetics Evolving to Systems, Humans, Organizations, and their Complex Interactions” (Cat. No.00CH37166)*, Nashville, TN, USA, 2000, köt. 3, o. 2316–2320. doi: 10.1109/ICSMC.2000.886462.

[17] Y. Deng, Y. Chen, Y. Zhang, és S. Mahadevan, „Fuzzy Dijkstra algorithm for shortest path problem under uncertain environment”, *Appl. Soft Comput.*, köt. 12, sz. 3, o. 1231–1237, márc. 2012, doi: 10.1016/j.asoc.2011.11.011.
[18] S. J. Russell és N. Peter, *Artificial Intelligence: A Modern Approach*, 4th Edition. University of California at Berkeley: Pearson, 2000. [Online]. Elérhető: https://www.pearson.com/us/higher-education/program/Russell-Artificial-Intelligence-A-Modern-Approach-4th-Edition/PGM1263338.html
[19] P. Hart, N. Nilsson, és B. Raphael, „A Formal Basis for the Heuristic Determination of Minimum Cost Paths”, *IEEE Trans. Syst. Sci. Cybern.*, köt. 4, sz. 2, o. 100–107, 1968, doi: 10.1109/TSSC.1968.300136.
[20] J. Yao, C. Lin, X. Xie, A. J. Wang, és C.-C. Hung, „Path Planning for Virtual Human Motion Using Improved A* Star Algorithm”, in *2010 Seventh International Conference on Information Technology: New Generations*, Las Vegas, NV, USA, 2010, o. 1154–1158. doi: 10.1109/ITNG.2010.53.
[21] A. Šmíd, „Comparison of Unity and Unreal Engine”, Czech Technical University in Prague, 2017.
[22] „UE3 & UDK | UDN WebHome”. https://docs.unrealengine.com/udk/Three/WebHome.html (elérés dec. 06, 2021).
[23] „Unreal Engine Powers Film & Television Production”, *Unreal Engine*. https://www.unrealengine.com/en-US/solutions/film-television (elérés dec. 05, 2021).
[24] „Unreal Engine is an Advanced Training and Simulation Software Platform”, *Unreal Engine*. https://www.unrealengine.com/en-US/solutions/simulation (elérés dec. 05, 2021).
[25] „Unreal Engine 4 Blueprint Visual Scripting”. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/ (elérés dec. 06, 2021).
[26] „Introduction to C++ Programming in UE4”. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/IntroductionToCPP/ (elérés dec. 10, 2021).
[27] U. Technologies, „CGI film, & cinematic trailers in real-time | Animation software | Unity”. https://unity.com/solutions/film-animation-cinematics (elérés dec. 06, 2021).
[28] U. Technologies, „Architecture, Engineering & Construction | Unity”. https://unity.com/solutions/architecture-engineering-construction (elérés dec. 06, 2021).
[29] U. Technologies, „Unity - Manual: Scripting”. https://docs.unity3d.com/Manual/ScriptingSection.html (elérés dec. 06, 2021).
[30] J. K. Haas, „A History of the Unity Game Engine”, Worcester Polytechnic Institute, Worcester, E-project-030614-143124, márc. 2014.

12. ÁBRAJEGYZÉK

13. TÁBLÁZATJEGYZÉK