Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék

Szakdolgozat

Miskolci Egyetem

Interaktív közlekedési modell procedurális generálása és megjelenítése WebGL segítségével

Készítette:

Kása Dániel Zoltán

Programtervező informatikus BSc

Témavezető:

Piller Imre, egyetemi tanársegéd

Miskolc, 2019

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Kása Dániel Zoltán (KYDK4Z) programtervező informatikus.

A szakdolgozat tárgyköre: Szabály alapú rendszerek, számítógépi grafika, szimuláció

A szakdolgozat címe: Interaktív közlekedési modell procedurális generálása és megjelenítése WebGL segítségével

A feladat részletezése:

A dolgozat célja egy olyan interaktív alkalmazás bemutatása, amely egy város közlekedését képes szimulálni. Ehhez a város úthálózatát, az épületeket és az egyéb környezeti elemeket az elkészült szoftver procedurálisan generálja majd. A generálás során külön paraméterek jellemzik a város szerkezetének és a szimulációnak a komplexitását. Az úthálózat esetében ilyen például, hogy milyen jellegű útkereszteződések fordulhassanak elő, illetve azoknak milyen legyen a gyakorisága. Fontos szempont, hogy a szimuláció realisztikus legyen olyan tekintetben, hogy a közlekedés résztvevői szabályosan közlekedjenek. Ehhez részletesen meg kell adni azok viselkedésének matematikai modelljét, illetve bemutatni azon heurisztikákat, amellyel a szimuláció valószerűvé tehető. A program JavaScript programozási nyelven kerül megvalósításra. A grafikus megjelenítést a WebGL nevű OpenGL alapú megjelenítő környezet biztosítja majd.

Témavezető: Piller Imre (egyetemi tanársegéd)

A feladat kiadásának ideje: 2018. szeptember 28.

. . . . . .
szakfelelõs

Eredetiségi Nyilatkozat

Alulírott **Kása Dániel Zoltán**; Neptun-kód: KYDK4Z a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős programtervező informatikus szakos hallgatója ezennel büntetõjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Interaktív közlekedési modell procedurális generálása és megjelenítése WebGL segítségével* címû szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ... év ..... hó ..... nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

....

dátum témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: .....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. fejezet

Bevezetés

Szakdolgozatom témájaként egy interaktív közlekedési modell elkészítését választottam. Ez a modell egy procedurálisan legenerált városból fog állni, amelyen megfigyelhető majd a járművek közlekedése. A várost procedurálisan generált dekoratív elemekkel is ellátom majd, amely így városkép tekintetében színes megjelenítést fog adni.

Az alkalmazás WebGL segítségével internetes böngészőben fog futni. A dolgozat, valamint a program forrásfájljainak verziókezelésére a GitHub-ot használom. Maga a program implementációjában a közlekedés fizikai realisztikusságára törekszem, az objektumok mozgása valós fizikai motorral kerül kiszámolásra.

A dolgozat témaválasztását illetően nagyban befolyásolt a grafikai téma iránti érdeklődésem. Továbbá a jelenleg is nagy fejlesztés alatt álló, viszonylag friss technológia, a WebGL és az általa adott lehetőség, hogy a böngészőben futó kód közvetlen hozzáférést kap a számítógép videókártyájához szintén felkeltette érdeklődésemet. Szintén inspirált még döntésemben a tanulmányaim alatt grafikából megszerzett ismeretanyag.

Mivel Unity Engine-ben való fejlesztéssel már korábban is foglalkoztam, onnan jött az ötlet hogy mivel a Unity képes WebGL platformra buildelni, ezért ebben a motorban fogom elkészíteni a programot.

A program felhasználó általi kezelését egy külön felülettel fogom biztosítani, amelyen az egyes paraméterek állíthatóak, ezzel nyújtva a program interaktív mivoltát.

Szakdolgozatomat az előbbiek megvalósításának lépéseit tekintve 5 részre tagoltam. Az első részben utánanézek az ebben a témában készült eddigi szoftvereknek, azok működését és felépítését fogom vizsgálni. A következő részben elkészítem a várost, és a rajta történő közlekedést leíró alapvető matematikai modellt. A harmadik részben a procedurális városgenerálás algoritmusát mutatom be, először annak vázlatos működését HTML Canvas elemen szemléltetve, majd Unity Engineben a teljes működést. A következő rész a szimuláció tervezésével foglalkozik, ebben részletesebben kitérek a Unity-ben való fejlesztés lépéseire, magának a motornak a működésére. Ezen kívül természetesen részletezem a program implementációját, a szkriptekből kiemelt kódrészletek segítségével. Az utolsó részben bemutatom az elkészült szimuláció működését, szemléltetem a szimuláció egyes elemeinek változását a paraméterek változtatásakor.

A modell megalkotása során nem törekszem minden közlekedési helyzet lefedésére, melyben a valóságban megjelenő összes közlekedési elem megjelenik. A hangsúlyt inkább a városkép kialakítására helyezem, mely így egy elfogadható városképet ad, olyan elemekkel díszítve mint a park, amely fákat és szökőkutat tartalmaz, többféle épület amelyek a városban való pozíciójuktól függnek, valamint a környezetben több helyen is megjelenő fák.

2. fejezet

Témaköri áttekintés

2.1. Hasonló célú szoftverek

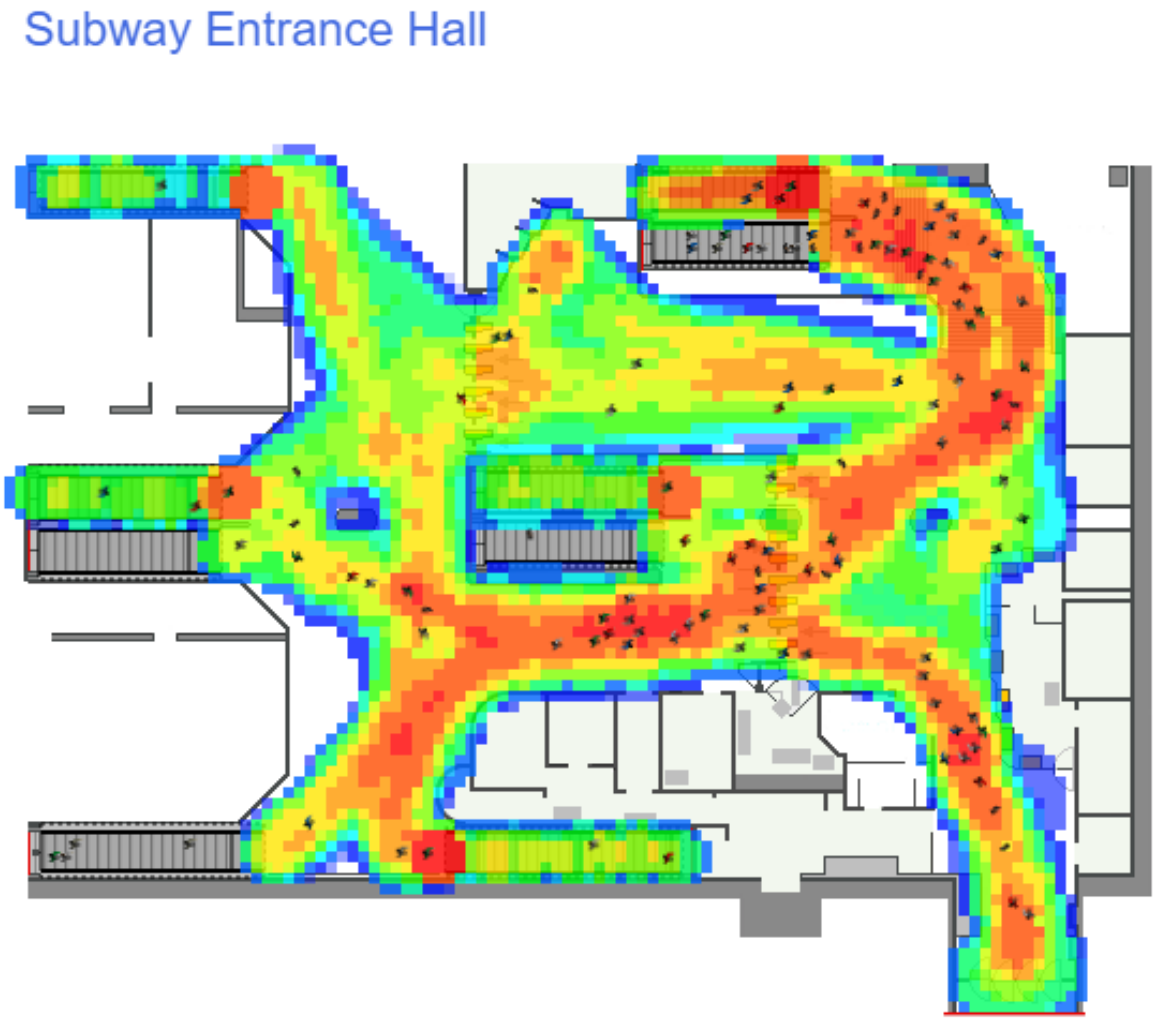
A feladatot azzal kezdtem, hogy utánanéztem milyen közlekedés modellezésére, annak szimulálására készült szoftverek készültek eddig. Ezek felépítése, valamint működése alapján fogom összeállítani a modellt, és továbbá az alapvető elvárásokat a generálást és a szimulációkat illetően.[1]

2.1.1. Road Traffic Library

Az eddigi közlekedés-szimuláló szoftverek közül kiemelném először az AnyLogic által készített Road Traffic Library-t[2]. Ezen szoftver segítségével létre lehet hozni egy úthálózatot különböző elemekből, mint például egyenes útszakasz, útkanyarulat, útkereszteződés, híd, gyalogátkelőhely, lámpás útkereszteződés, valamint buszmegállók és parkolók. A szoftver képes különböző közlekedési szabályok alkalmazására, valamint a járművek intelligens módon választják meg a haladáshoz szükséges útvonalat. Valósidejű szimuláció futtatására is van lehetőség, 2D-ben és 3D-ben is.

2.1. ábra. Kép a Road Traffic Library-ből[4]

A szimulációkat felhasználva hőtérkép is előállítható, amely megmutatja mennyire voltak forgalmasak az adott területek, így egyértelművé válik hol alakul ki könnyedén forgalmi dugó.



2.2. ábra. Hőtérkép egy metróállomás gyalogosforgalmáról az AnyLogic szimulációs szoftverrel[5]

Lehetőség van a geoinformációs rendszerekben használt vektorgrafikus formátum, azaz shapefile importálására is. Ezen fájl alapján a szoftver automatikusan előállítja az úthálózatot. A szoftver bővíthető még gyalogos- és vasútforgalomra vonatkozó könyvtárakkal is.

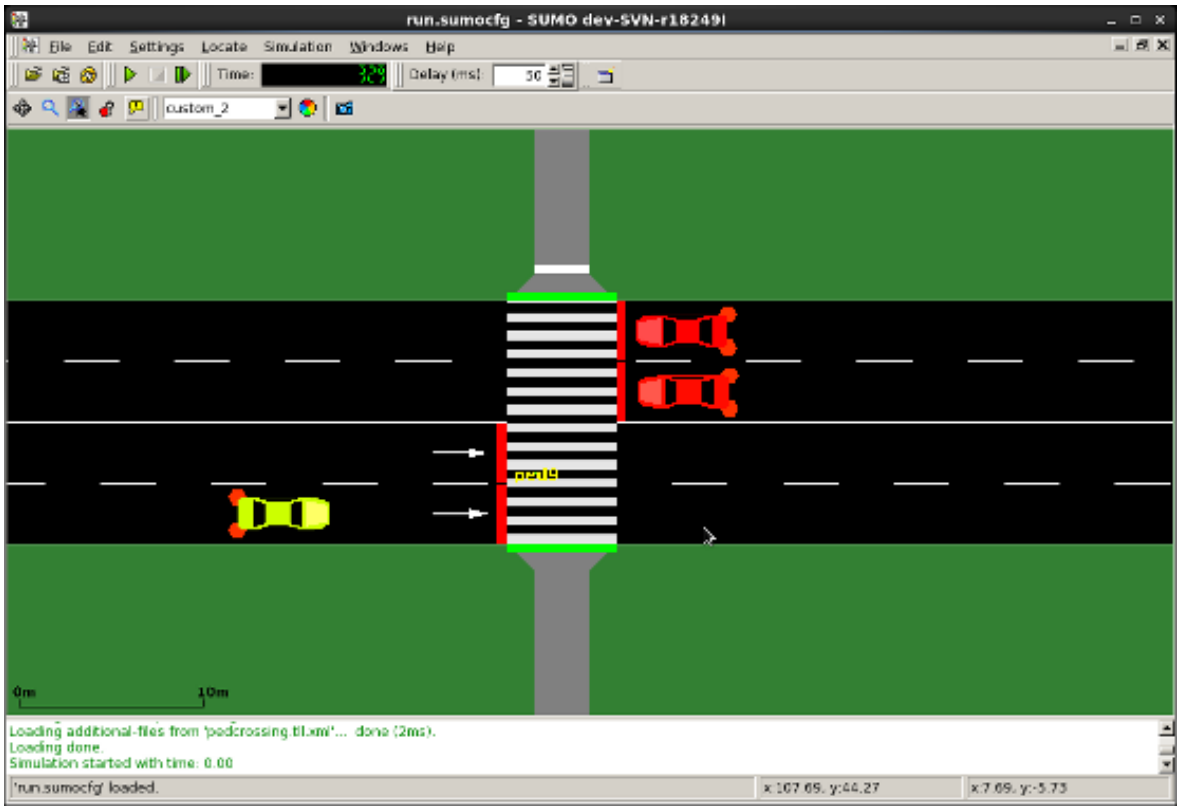
A gyalogos könyvtár segítségével szimulálható a gyalogosforgalom sűrűsége különböző környezetekben. Ilyen környezet lehet például egy buszmegálló, vagy vasútállomás területe. Az egyes épületek felépítése is befolyásolja a forgalom sűrűségét, a gyalogosok minden akadályt, ebbe beleértve egymást is, próbálnak elkerülni.

A vasút könyvtár segítségével egy pályaudvar vasútforgalma komplex módon szimulálható. A forgalmat befolyásolja a tehervonatok rakodási ideje, a karbantartás, üzleti folyamatok lefolyása, és sok más további esemény is.

2.1.2. SUMO

A SUMO, azaz Simulation of Urban MObility egy ingyenes, nyílt, közlekedés-szimuláló C++-ban írt szoftver[3]. Az első verzió 2001-ben került kiadásra, azóta már az 1.1.0-ás

verziószámnál jár. Ez a szoftver lehetőséget ad több közlekedési forma (tömegközlekedés, gyalogosok, stb) modelezzésére a személygépjárművek mellett. Továbbá nagy eszköztárral bír az olyanokhoz mint útvonaltervezés, káros-anyag kibocsátás, valamint vizuális megjelenítés. Lehetséges továbbá meglévő úthálózat importálása fájlból, valamint személyre szabott modellek használata. A SUMO nem tartalmaz semmilyen megkötést az úthálózat méretét, vagy az egyszerre szimulálható járművek számát illetően.



2.3. ábra. Kép a SUMO programból[6]

A program lehetőséget ad a szimuláció online történő kezelésére is, a TraCI, azaz Traffic Control Interface használatával. Ilyen esetben a SUMO tulajdonképpen egy szerver, amelyet egy vagy több kliensen keresztül tudunk vezérelni. TCP alapú kliens/-szerver architektúrát használ.

A közlekedési lámpák viselkedését egyénileg be lehet állítani, vagy akár előre megírt TLS programból is lehetséges importálni. Az importáláshoz számos python szkript áll rendelkezésre. Ha nem áll rendelkezésre TLS program, a SUMO képes automatikusan generálni egyet. Ilyenkor alapértelmezett értékeket rendel minden paraméterhez amelynek értéke nem volt kapcsolókkal definiálva. Ezek közé tartoznak:

- –tls.cycle.time - lámpák váltási ideje.
- –tls.yellow.time - mennyi időre sárga a lámpa, ez alapértelmezetten az utak sebességhatára alapján történik kiszámításra.
- –tls.minor-left.max-speed - az útkereszteződésben balra kanyarodást megengedő

sebességhatár (ha az út sebességhatára ezen érték felé esik, nem engedélyezett a balra kanyarodás.)

- –tls.left-green.time - ha egyszerre kap zöldet az egyenesen haladó, és a vele szemből balra kanyarodó, ezen érték idejéig a balra kanyarodó kap elsőbbséget.
- –tls.allred.time - minden zöld lámpa előtt ezen értéknek megfelelő ideig piros az összes lámpa (alapértelmezetten 0.)
- –tls.default-type - actuated értékre állítva változó hosszúságú ideig tartanak a zöld lámpák.

A szoftver alkalmas a közlekedési lámpák teljesítményének vizsgálatára, olyan útvonaltervezésre amely igyekszik a káros-anyag kibocsátást és a légszennyezést minimalizálni. A gyakorlatban történt híresebb alkalmazásai tartozik például a 2006-os labdarúgó világbajnokság-, valamint a Pápa 2005-ös Kölni látogatása során a közlekedés előrejelzése.

Maga a programcsomag tartalmazza a parancssori SUMO szoftvert, a grafikus felülettel ellátott GUISIM-et, egy úthálózat importáló NETCONVERT szoftvert, úthálózat generáló NETGEN szoftvert, egy OD2TRIPS nevű alkalmazást amely O/D mátrix alapján generál útvonalat, számos útvonal generátort, valamint egy grafikus úthálózat-szerkesztő alkalmazást.

2.2. Meglévő térkép-adatbázisok modellje

A Google Maps az egyik legelterjedtebb úthálózat-vizualizálásra alkalmas szoftver. Ez egy JavaScript alapú webalkalmazás, mely a geoinformációs adatok megjelenítése mellett valós-idejű forgalom elemzésre is képes.[7]

Ehhez az adatokat több féle forrásból állítják elő. A Base Map program keretében rengeteg különböző forrásból szereztek vektoradatokat a létező úthálózatokról. Később ennek a nevét Geo Data Upload-ra váltották, viszont a lényege ugyan az maradt. Továbbá szereznek adatokat műholdképekből, Android rendszerű okostelefonok szolgáltatásai által, valamint a nemrég bevezetett "Helyi Idegenvezetők" funkcióval, amin keresztül bárki tölthet fel adatokat.

A valós-idejű forgalomelemzéshez kezdetekben a helyi önkormányzatok által szolgáltatott adatokat használták, melyeket az egyes helyeken felszerelt szenzorok segítségével gyűjtöttek be. Viszont ezek csak a legforgalmasabb utakról adtak információt, ezért később áttértek a "crowd-source" módszerhez. Ezzel a módszerrel egy Google Maps-et futtató okostelefon GPS adatait felhasználva ad becslést egy út forgalmasságára. Lényegében azt elemzi, mennyien küldtek GPS adatot egy adott útszakaszon azonos időben.

Az utak és más objektumok kirajzolásához a böngésző leegyszerűsített vektoradatokat kap, melyek alapján felépíti az azoknak megfelelő formákat. Minden egyes objektumhoz tartoznak különböző metaadatok, mint például a sebességkorlát, haladási irány, út típusa (főút, stb), valamint egy név. Az objektumok JSON formában vannak tárolva, maga a Google Maps API pedig támogatja a GeoJSON forrásból történő adatok betöltését térképekre. Ezen információk alapján épül fel a modell amit a böngésző 2D vagy 3D formában ábrázol egy térképen.

2.3. GeoSpatial adatbázisok

A GeoSpatial adatbázisok adják az előbbi szolgáltatások hátterét. Ezek az adatbázisok kifejezetten a térbeli objektumokat leíró adatok tárolására, azok egyszerű lekérdezésére vannak optimalizálva. A leggyakoribb adatok pontok, vonalak, vagy poligonok formájában fordulnak elő.

A hagyományos adatbázisokhoz képest sokkal nagyobb teljesítményt nyújtanak ilyen területen, valamint gyakran az ilyen típusú adatok feldolgozásához bővíteni kell azok funkcionalitását. Ennek érdekében az Open Geospatial Consortium 1997-től kezdődően definiálta az ISO 19125 szabványt ezen funckionalitás megvalósításához, melynek második része leírja az SQL-ben történő megvalósítást is. Az OpenGIS szabvány ugyan magába foglalja a CORBA és az OLE/COM implementációkat is, ezek nem részei az ISO szabványnak.

A szabvány többek között a következő műveleteket definiálja:

- Térbeli számítások, azaz objektumok közti távolság, vonalak hossza, stb
- Térbeli predikátumok, objektumok közötti kapcsolatokat leíró logikai lekérdezések
- Geometriai konstruktorok, alakzatot leíró adatok alapján új geometriai elem létrehozása
- Egy tetszőleges objektumhoz tartozó információk lekérdezése

3. fejezet

Matematikai modell részletezése

3.1. Modell specifikálása

A közlekedésre irányuló modell fő tulajdonságai közé kell hogy tartozzon az elegendő információ tárolása ahhoz, hogy alapvetően tudjanak a szimulált járművek rajta közlekedni. Ezt rengeteg féle képpen meg lehet valósítani, sokféle eleme lehet egy adott térképnek, akár minden egyes előforduló úttípusra, szabályra, táblára tekinthetünk úgy mint a modell egyedi része. Ez a megközelítés viszont lehetségesen túlbonyolítaná a modellt, ezáltal a procedurális generálás nem adna annyira elfogadható úthálózatot, lehetségesen életszerűtlen helyzetek épülhetnek fel a sok elem megléte, azok elhelyeződése miatt. Sokkal inkább célszerű a modell részeként tekinteni a főbb építőelemeket, melyek elegendőek magához az úthálózat generálásához, valamint néhány alapvető szabályt megvalósító elemet, mint például a lámpás útkereszteződés, az egyirányú út, valamint a többsávos út.

3.2. A modellben használt elemek

Matematikailag a modell egy gráfból áll, melynek csomópontjai jelölik az útkereszteződéseket, vagy két egyenes útszakasz között az összekötő részt. Az élek jelölik magát az útszakaszokat, ezeknek az éleknek pedig számos tulajdonságai lehetnek amivel különbséget teszek több elem között. Az élek lehetnek irányítottak is, a modellben ez fogja jelölni az egyes elemeken a haladási útvonalat.

3.2.1. Egyenes út

Alapvető egyenes útszakasz, ennek lehet több paramétere is. Ennek az elemnek elsősorban tartalmaznia kell az úton létező sávok számát. Az elem leírásához szükség van annak kezdő és végpontjára. További információt ad a modell többi részének, ha tartalmazza egy megközelítőleges égtájnak megfelelő irányt. Ebből következtetni lehet arra ha például kanyar következik, és ha igen akkor milyen irányban, csak arra van szükség rá hogy megnézzük milyen módon változik az utak hozzávetőleges iránya.

Alapvetően az út olyan szélességű, hogy elfér egymás mellett két jármű, ezért ha több sáv van csak arra kell figyelni hogy megfelelő helyen közlekedjenek az adott járművek. A gráfon való pozicíójának behatárolásához a kezdő és végpont az ő általa

összekötött két csomópontot jelenti.

3.2.2. Egyirányú út

Hasonló az egyenes úthoz, viszont egyértelműen meg kell határozni a két végpont közül melyik a kiindulási, melyik a célpont. Hasonlóan az egyenes szakaszhoz, azzal a különbséggel, hogy ez az út kizárólag 1 sávos.

A gráfot tekintve itt a két összekötött csomópont között egy irányított él lesz, melynek kezdő és végpontjaként egyértelműen el kell jelölni a két pontot.

3.2.3. Kanyar

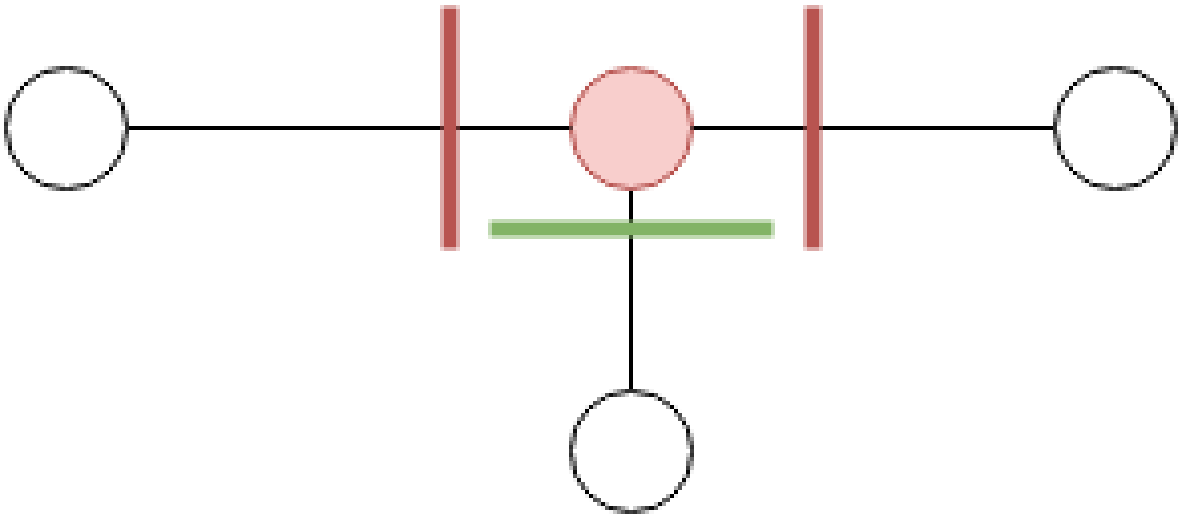
Vehető tulajdonképpen külön elemnek, de lényegében csak két egymást követő egyenes szakasz, melyek az őket összekötő elem mentén nem egyenes irányban folytatódnak. A kanyar iránya adódik az általa összekötött két egyenes útszakasz iránya által.

A gráfon kanyarnak tekinthető az olyan csomópont, melyből csak két él indul ki, ha ez a két él iránya egymástól eltérő, és nem ellentétes.

3.2.4. Útkereszteződés

Három vagy több útszakasz találkozásánál használatos elem. Tulajdonságai közé tartozhat az hogy tartalmaz-e jelzőlámpát az útkereszteződés, vagy sem. Az útkereszteződés jellemezhető annak középpontjával, valamint az azt érintő utak halmazával.

A gráfon ez azt jelenti, hogy amennyiben egy csomópontból legalább 3 él indul ki, az egy útkereszteződés.



3.1. ábra. Háromágú útkereszteződés lámpákkal

3.2.5. Személygépjármű

Legalapvetőbb közlekedési jármű. Az autókról nyilván kell tartani azok jelenlegi pozícióját, mely abból adódik hogy éppen melyik élen haladnak, melyik csomópont felé, és

attól milyen távolságra vannak. Minden jármű véletlenszerűen kiválasztott útvonalat kap, amin végig kell haladjon. Ezek az útvonalak a teljes gráfon egy-egy részgráfok. A jármű jellemzői közé tartozik annak sebessége, valamint a következő csomópontot elérve a várható haladási iránya, tehát előre, balra, vagy jobbra megy-e tovább.

Tudnia kell a vele egy úton közlekedő járművekől is, azok pozíciójától függ a viselkedése. A járművek kezdetben véletlenszerű helyen kezdenek, majd ha sikeresen eljutottak a célpontjukhoz eltűnnek a modelből (leparkolt kocsinak tekintve). Ezen útvonal kijelölése egy véletlenszerűen kiválasztott csomóponttól indul, ahonnan véletlenszerű irányba indulunk el, majd a következő csomópontot elérve szintén új irányt választunk addig, ameddig vagy zsákutcába nem ér az autó, vagy eléri a paraméterként maximálisan kijelölt úthosszat. Szintén a paraméterezést tekintve ilyenkor új jármű jelenik meg ha jelenleg kevesebb van mint a megadott maximális érték.

3.2.6. Autóbusz

Kissé különlegesebb járműtípus, vonatkozik rá néhány további szabály. Többek között elsőbbsége van megállóból elindulva, valamint ha lehet próbál jobbra tartani,többsávos úton használhatja a külső sávot is egyenes haladásra és balra kanyarodásra. A buszmegállók között megkötött útvonalon kell haladnia, mindegyiknél pedig meg kell állnia. Ha elérte az útvonal végét megfordul, és visszafelé halad végig rajta. A szimuláció teljes ideje alatt az útvonalat követi, kezdetben az útvonal végéről elejétől indul.

Ez az útvonal a gráfot tekintve egy legmélyebb szinten elhelyezkedő véletlenszerűen választott csomópont, valamint a tőle legtávolabb lévő csomópont közötti utat jelenti. Kijelölése könnyen megoldható ezen két csomópont szüleit végigkövetve a középen lévő kiindulási pontig.

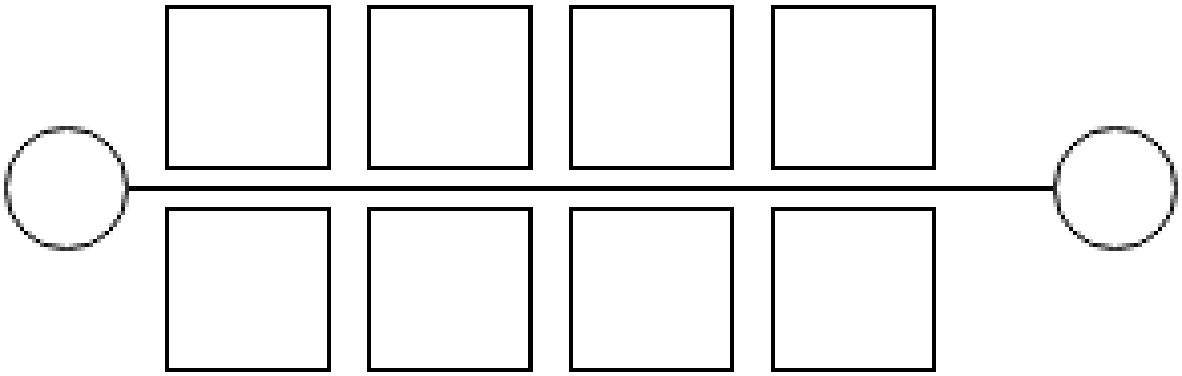
3.2.7. Buszmegálló

Autóbusznak elhelyezett megállási pont. Két csomópont közötti él felezőpontján helyezkedik el, ezért leírásához elegendő az adott élt megadni. Szükséges információ a buszmegállók közötti minimum távolság, azaz hány csomópontnak kell lennie legalább két buszmegálló között. Ezt a számot véletlenszerűen fogom meghatározni minden buszmegálló elhelyezése után újra számítva.

Ezen elem egy olyan csomópont, amelyből nem indul él, és nem is tart belé él.

3.2.8. Épületek

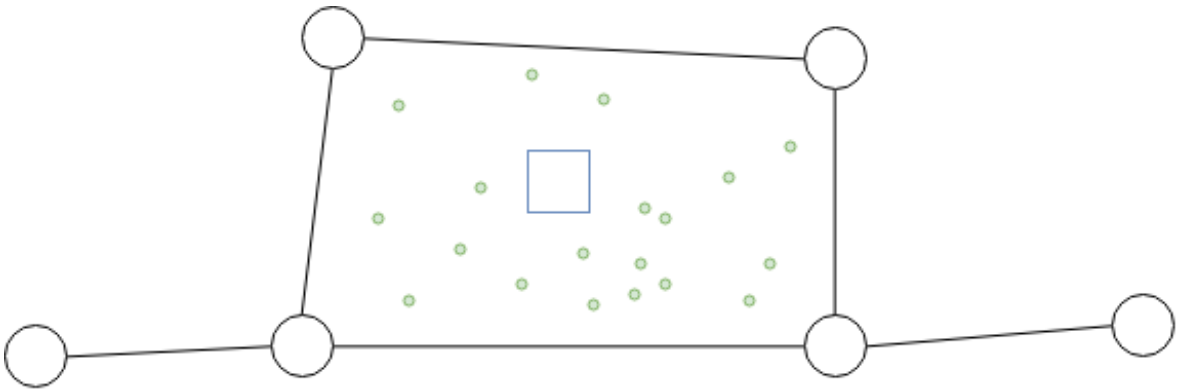
A gráf létrejötte után kijelölendő elemek. Az élek két oldalán helyezkednek el a házak, amelyek az alacsonyabb szinten lévő csomópontok között blokkházak, magasabb szinteknél pedig kertesházak. Az út két oldalán lévő elhelyezkedés szimmetrikus. A blokkok magassága változó.



3.2. ábra. Út két oldalán lévő épületek

3.2.9. Park

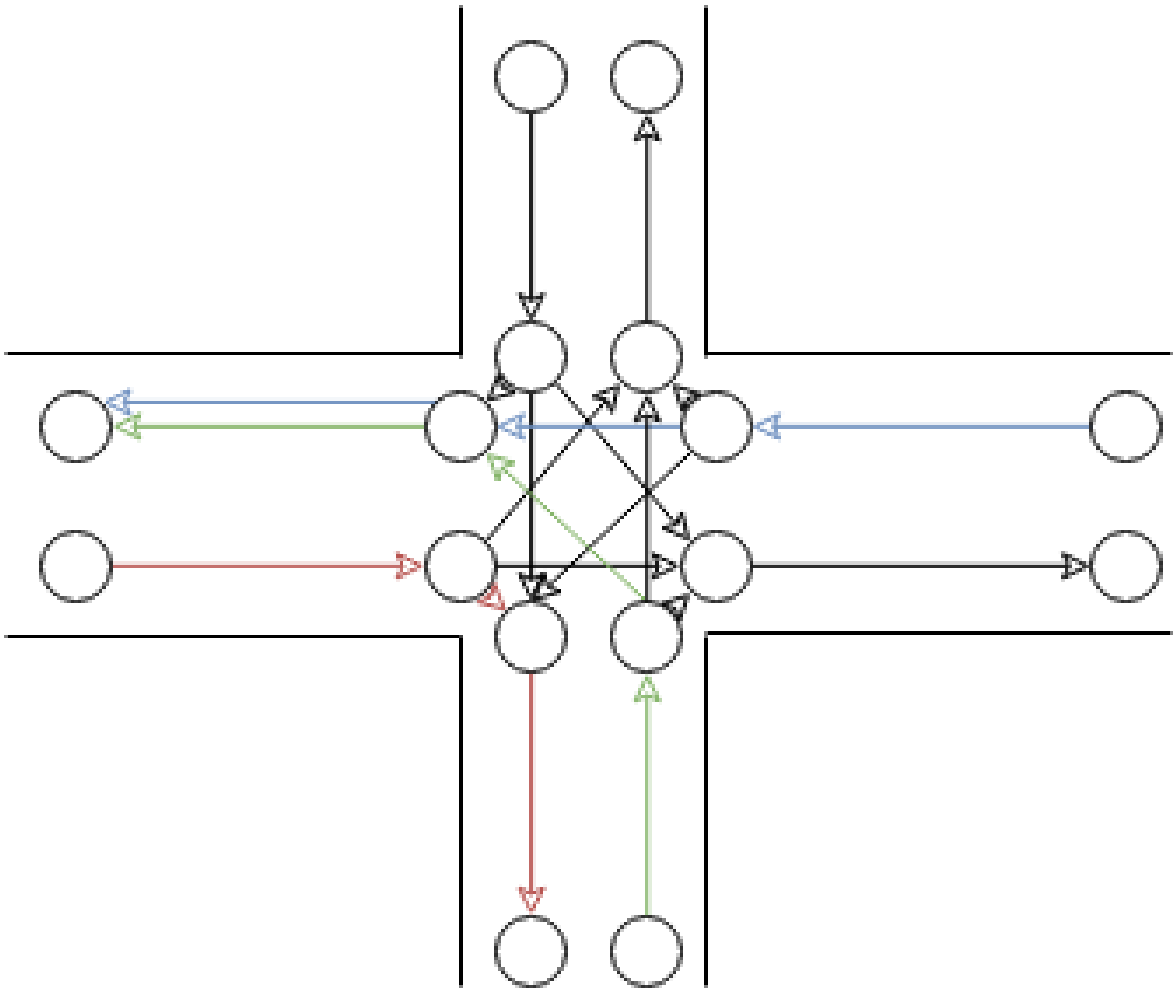
Szintén a gráf létrejötte után kerül elhelyezésre, kizárólag az utakkal teljesen közbezárt területeken. Az adott területen belülre fák kerülnek, valamint egy szökőkút.



3.3. ábra. Elkerített részen lévő park, kék négyzettel jelölve a szökőkutat, zöld pontokkal a fákat

3.2.10. Útvonal

Ez az egy járműhőz tartozó egyedi haladási vonala az úthálózaton. Nem ugyan az, mint az útvonal éleinek kezdő és végpontja, ugyanis itt egyértelműen ki kell jelölni annak a sávnak a pontjait, amelyikben a jármű haladni fog. Útkereszteződésben ez segít a helyes irányba való kanyarodásban.



3.4. ábra. Egy útkereszteződésbeli útvonalakat kijelölő pontok. Példa útvonal balra kanyarodáshoz (zöld), jobbra kanyarodáshoz (piros), egyenes haladáshoz (kék).

3.2.11. Közlekedési szabályok

Néhány alapvető közlekedési szabály amelyet a járműveknek követniük kell, általában attól az elemtől függ ahol haladnak, valamint attól amelyhez tartanak

- Egyirányú útra csakis a kijelölt útirányban hajthatnak rá a járművek
- Ha az út több sávot tartalmaz, a külső sáv csak jobbra kanyarodásra alkalmas, a belső sáv balra kanyarodásra, illetve egyenes haladásra
- Ha az útmenti buszmegállóból elindulni készül a busz, a többi járműnek elsőbbséget kell neki adnia
- Jelzőlámpával ellátott útkereszteződésbe nem hajthat be ha a lámpa piros, ha aközben vált hogy benne van akkor még áthaladhat

3.3. A modell kritériumai

3.3.1. Generálási szempont

Magának az úthálózatnak a generálására vonatkozóan a következők a célok:

- Viszonylag kevés elemből álljon össze a generált úthálózat
- Ne legyen életszerűtlen az összeállított úthálózat
- Tartalmazza az alapvető közlekedési helyzetek szimulálására szükséges elemeket
- Vizuálisan emlékeztessen egy átlagos városra
- Bizonyos mértékig paraméterezhető legyen, főleg a méreteket érintve

Elemek száma

Az említett viszonylag kevés elem, vagy pontosabban kevés különbözű típusú elem azt jelentené, hogy ne legyenek túl specifikusak az elemek, hanem akár könnyedén egyik elem bővítésével elő lehessen állítani egy másikat. Ilyen például a modellben az egyenes szakasz, melyből kis változtatással előáll az egyirányú út.

Életszerű úthálózat

Röviden annyit tesz, hogy nem tartalmaz olyan területeket ahol össze van szűkítve kis területen rengeteg út. Tehát legyen minimális hely kihagyva az utak között az épületeknek. Életszerűtlennek tekinthető továbbá az is, ha egy útkereszteződésből rengeteg út indul ki, azaz 5-6 avagy annál több. Ezért a modellben az útkereszteződés maximálisan 4 utat köthet össze.

Közlekedési helyzetek

Legyen benne jelzőlámpával ellátott útkereszteződés, többsávos út, egyirányú út, valamint buszmegálló. Ezek az elemek szükségesek minden esetben, hogy a szimuláción meg lehessen jeleníteni a lámpák működését, a többsávos utak eseten a járművek sávválasztását és tartását, egyirányú út irányának a betartását, és buszmegállónál a busz megállását, neki járó elsőbbség megadását.

Átlagos város képe

Átlagos városra hasonlít akkor, ha a központtól kifelé haladva ritkul az úthálózat, valamint a többsávos utak az előreláthatólag forgalmasabb helyeken vannak elhelyezve. Ennek érdekében a modellben elhelyezett csomópontok átlagosan kevesebb élt indítanak magukból, minél távolabb vannak a gráf kiindulási pontjától, azaz a középponttól.

Paraméterezés

Ki lehessen jelölni hogy a generálás meddig tartson, azaz hány csomópontot terjesszen ki.

3.3.2. Szimulációs szempont

A már legenerált úthálózaton a szimulációra tekintve ezek a célok:

- A járművek kövessék a modellben definiált alapvető szabályokat
- A szimuláció bizonyos mértékben paraméterezhető legyen, főleg a járművek számát illetően
- Reagáljanak egymásra a járművek
- Ne legyen életszerűtlen a járművek viselkedése
- Teljesítmény szempontjából elfogadható legyen egy adott méretig

Szabályok

A már előbbiekben leírt szabályoknak megfelelően közlekedjenek a járművek, haladási útukat azok alapján válasszák meg.

Paraméterezés

Meg lehessen adni mennyi jármű legyen maximálisan egyidőben az úthálózaton, külön értve a buszokat ettől. Továbbá az is megadható legyen milyen gyakorisággal jelenik meg személygépjármű, és mekkora lehessen a neki kijelölt útvonal maximális hossza csomópontokban mérve.

Reagálás

Ha egy jármű közelít egy másik felé kezdjen lassítani. Ez a lassítás függjön a távolság mértékétől, ha elég közel kerül hozzá teljesen álljon meg. Csak akkor induljon el egy jármű, ha haladási útját nem állja el másik jármű. Közúti baleset, azaz ütközés esetén a két érintett jármű kis idő után tűnjön el, ezzel szemléltetve az elszámolás, elvontatás folyamatát.

Életszerű viselkedés

A járművek tartsák az út vonalát, ne térjenek át a szembe sávba. Gyorsítás és lassítás viszonylag realisztikusan történjen, ne pedig hirtelen. Kanyarodás előtt igyekezzenek lelassítani, kanyart pedig végig lassan bevenni.

Teljesítmény

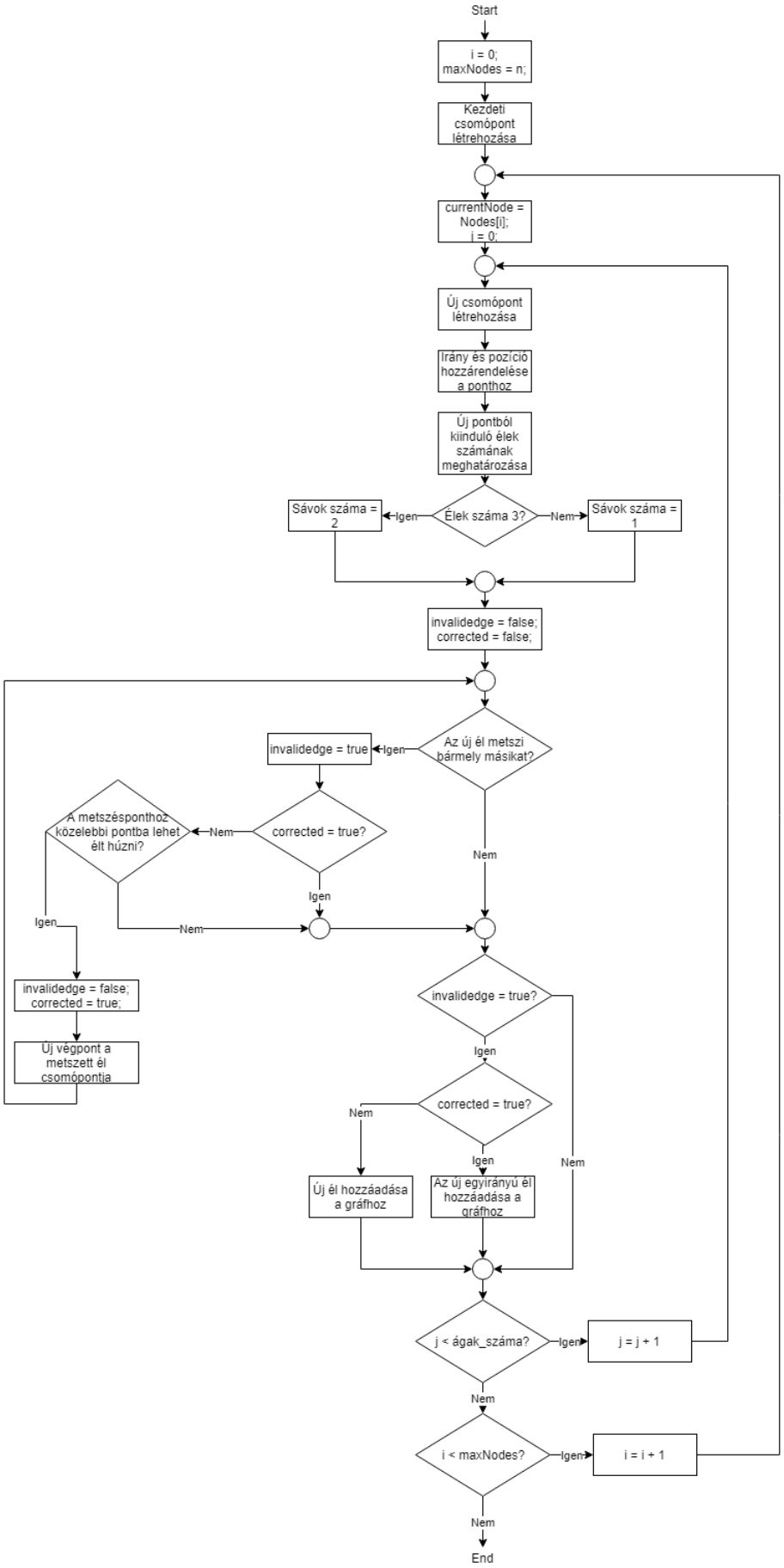
A szimuláció mérete mind az úthálózat csomópontjait, mind az egyszerre megjelenő autók számát értem. Abban az esetben ha ezek az értékek nem kimagaslóan nagyok, a szimulációnak lényegesebb teljesítményromlás nélkül kell futnia.

4. fejezet

Generálás

4.1. Úthálózat Generálása

Maga a generáló algoritmus megalkotásakor első lépésben a nagyvonalakban leírt elvárt működést vizsgáltam. A hálózatnak középponttól kifelé haladva ritkulnia kell, ezzel közelítve egy valódi várost. A paraméterezéssel kapcsolatos elvárásokat könnyen meg lehet valósítani. Egy gráfként képzelhető el a megalkotott úthálózat, melynek csomópontjai jelölik az egyes útelemek elejét, élei pedig azoknak tartományát. A gráf 0. szintjét jelöltem el a hálózat középpontjának, innen mindig 4 irányba indulnak élek. Mivel az egy csomópontból kiinduló maximális élek számának 4-et választottam, ez garantálja hogy a középpontnak kijelölt ponttól minden irányba nagyjából egyenletes mértékben terjedjen az úthálózat. Minden további csomópont generálódásakor kapja meg annak értékét, hogy mennyi irányba indulnak belőle élek.



4.1. ábra. A generálás lépéseinek egyszerűsített folyamatábrája

4.1.1. Generáló algoritmus

A csomópontok generálása a következőképpen történik:

1. A kiválasztott csomópont kap egy véletlenszerű irányt, észak, dél, kelet, vagy nyugat értékében
2. Ha a kapott érték megegyezik azzal az iránnyal amerre a csomópont szülője van, vagy már indult arra belőle él, újat kap
3. A kiválasztott iránynak megfelelő véletlen generált X és Y koordinátákat kap
4. A kapott koordinátákból alkotott csomópont felkerül a gráfra
5. Ha a jelenlegi csomópontból még kell éleket indítani, akkor kezdődik előről

Az előbbiekben említett X és Y koordinátát kissé pontosítanám. Ha a csomópontot nyugat vagy kelet irányba generáljuk, az X koordináta értéke a jelenlegi pont X koordinátája, hozzáadva egy előre definiált értékek közötti véletlen szám. Az Y koordináta ilyenkor egy minimális eltérés az út fekvésében, hasonlóan generálódik, de lényegesen alacsonabb véletlen számot kap. Ez tulajdonképpen a valóságban is látható "tökéletlenségekre" vezethető vissza, ugyanis a legtöbb esetben ott sem teljesen merőleges egymásra kettő út. Az észak és dél irányba induló pontok esetén ugyan ez az eljárás, csak a két koordináta szerepe felcserélődik. Fontos viszont az is, hogy a középponttól távolodva átlagosan kevesebb elágazása legyen egy csomópontnak, ezért ehhez felhasználható annak gráfbeli szintje. Minden csomópontról eltárolódik annak szintje (szülő szintje + 1), ami befolyásolja az elágazások számát. Egy másik probléma az, hogy az így generált élek gyakran metszhetik egymást, amely az aluljárók és a hídak hiányában itt nem megengedett, ezért utolsó lépésnek ellenőrizni kell hogy az új él metszi-e bármely meglévő élek közül akár az egyiket is, és ha igen akkor nem kerülhet fel a gráfra. Az él viszont hozzáadható az eredetileg kijelölt csomóponthoz legközelebbi csomóponttal történő helyettesítéssel, feltéve ha így nem történik létező út metszése. A buszmegállók elhelyezésével kapcsolatban a következő feltételeket adtam:

- A legcélszerűbb a hálózat két, egymástól távol lévő, a gráf mélyebb szintjein elhelyezkedő pont közötti út létrehozása
- Az útvonalnak érintenie kell a középpontot, a gráf 0. szintjét
- Lehető legkevesebbszer érintse többször ugyan azt a csomópontot

Ennek érdekében először kijelölöm az út egyik végét, mint megállót. Ezek után keresek egy utat a gráf 0. szintjéhez, melyen legfeljebb minden második csomópontot kijelölöm megállónak. A középpont elérése után kijelölöm a második végpontot, mely a középponttól az eddigi iránynak ellentétesen, a gráf mélyebb szintjei közül kell lennie. Az ehhez a középpontból vezető úton, az utóbbihoz hasonló eljárással kijelölök megállókat. Az utak sávszámának meghatározására a gráfbeli mélységüket használom. A magasabb szinten lévő élek nagyobb valószínűséggel lesznek többsávosak. Ez a generálás végén választódik ki. Az egyirányú utak kijelölése is a folyamat legvégén történik. Ennek az oka magából az utak jellegéből adódik, ugyanis egyirányú út nem végződhet zsákutcában, és nem is zárhatja el a hálózat egy részét a többi elől. Éppen ezért egy olyan részgráfot kell találni amely Hamilton-kört alkot, amelyen egy részgráfot már nyugodtan egyirányúvá lehet tenni.

4.1.2. Az egyirányú út

Egy könnyű módszer annak megoldására, hogy mely utak lehetnek egyirányúak az lenne, ha a generálás során egymást metszett utakból kialakult éleket jelölöm el egyirányúnak. Ezek az élek ugyanis biztos hogy egy létező Hamilton-út részei, mivel az eredetileg fagráfnak megfelelő úthálózat bármely két csomópontja közé húzott él Hamilton-utat ad.

4.2. Alapelemek megjelenítése HTML Canvas segítségével

4.2.1. Fejlesztői környezet felállítása

Az algoritmus működését, eredményét egy HTML Canvas objektumon szemléltetem. Magát a kódot JavaScript-ben írtam, annak ECMAScript 6-os verziójában. Fejlesztői környezetnek A JetBrains által készített WebStormot használtam. Először is létrehoztam egy HTML fájlt, ahol a body tag-en belül elhelyeztem a canvas elemet. Erre az elemre JavaScriptből a HTML-ben megadott id-je alapján lehet hivatkozni. Ezután létrehoztam egy JavaScript fájlt, melyben egy (document).ready() szintaktikában elhelyezett függvényhívással indítom a generálást. A (document).ready() a jQuery függvénykönyvtárnak része. Azért szükséges ebben az esetben, mert ameddig a html fájl nem töltött be teljesen, a script nem futtatható mert a canvas elemre nem létezne semmilyen referencia. Ez a részlet biztosítja hogy csak akkor kezdődjön a script futtatása, ha a HTML documentum teljesen betöltött. A js fájlban globálisan eltárolom egy változóba a canvasre mutató referenciát, majd egy másik váltózóba lekérem ennek a kontextusát.

4.2.2. Létrehozott osztályok

Node

A gráfon egy csomópontot leíró objektum. Egy konstruktort tartalmaz, amely beállítja az osztály 6 adattagjának értéket. Ezek az adattagok a következőek:

- x: A csomópont X koordinátája
- y: A csomópont Y koordinátája
- branches: Hányfelé kell tovább ágaztatni a csomópontot
- level: A gráf hanyadik szintjén helyezkedik el a csomópont
- connectedFrom: Milyen irányból lett kiterjesztve a csomópont
- parentNodeIndex: A gráfban lévő csomópontokat tartalmazó vektoron belül ezen csomópont szülőjének indexe

Edge

A gráfon egy él leírására szolgáló objektum. Egy konstruktort tartalmaz, amely inicializálja 4 adattagját. Ezek az alábbiak:

- from: Az a csomópont, ahonnan az él kezdődik
- to: Az a csomópont, amiben az él végződik
- lanes: Az él által jelölt úton hány sáv van
- oneway: Logikai változó, az út egyirányú-e vagy sem

Graph

Magát a gráfot leíró objektum, mely tartalmaz 2 függvényt annak canvas-on történő kirajzolására. Paraméter nélküli konstruktora 2 adattagját inicializálja üres értékkel. Adattagjai és függvényei a következők:

- Nodes: A csomópontokat tartalmazó vektor
- Edges: Az éleket tartalmazó vektor
- drawAllNodes: Függvény, mely végig iterál a Nodes vektoron, és mindegyik elemére meghívja a kirajzoló függvényt
- drawAllEdges: Függvény, végig iterál az Edges vektoron, egyirányú út esetén nyilat rajzol, ellenkező esetben egyenes vonalat

4.2.3. Segédfüggvények

intersects

Ez a függvény arra szolgál, hogy kiszámítsa két él metszi-e egymást. Ehhez két paramétert vár, edge1 és edge2 néven. Visszatérési értéke logikai típusú. Működésében a két szakaszból egyeneseket képez, majd ezek alapján az X = v1 + lambda*d1, X = v2 + gamma *d2 egyenletrendszer mátrixának kiszámolja a determinánsát. Ha ez 0, a két szakasz nem metszi egymást. Ellenkező esetben megnézi az egyenletrendszerben szereplő gamma és lambda értékeket, azaz a két egyenes mentén az irányvektor hányszorosát kell venni hogy elérkezzünk a ponthoz. Ha ez nem 0 és 1 közé esik, a metszéspont nincs a szakaszon belül.

```
function intersects(edge1,edge2) {
    let det, gamma, lambda;
    det = (edge1.to.x - edge1.from.x) * (edge2.to.y - edge2.from.y) -
    (edge2.to.x - edge2.from.x) * (edge1.to.y - edge1.from.y);
    if (det === 0) {
        return false;
    } else {
        lambda = ((edge2.to.y - edge2.from.y) * (edge2.to.x - edge1.from.x)
         + (edge2.from.x - edge2.to.x) * (edge2.to.y - edge1.from.y)) / det;
        gamma = ((edge1.from.y - edge1.to.y) * (edge2.to.x - edge1.from.x)
```

```
            + (edge1.to.x − edge1.from.x) * (edge2.to.y − edge1.from.y)) / det;
        return (0 < lambda && lambda < 1) && (0 < gamma && gamma < 1);
    }
}
```

distance

Két csomópont közötti távolság kiszámítására szolgáló függvény. Ehhez a két vektor által meghatározott szakasz hosszát számolja ki pitagorasz tétellel. Ezzel az eredménnyel tér vissza.

maxBranchesReached

Ezzel a függvénnyel megmondható hogy egy adott csomópontba már lett-e 4 él húzva. A függvény paramétereiként megkapja a keresett csomópontot, a gráfot, és a jelenleg kiterjesztés alatt álló csomópont gráfbeli indexét. Amennyiben a gráfban megtalált csomópont branches értéke kisebb mint 3, akkor növeli 1-el és hamis értékkel tér vissza, azaz még nem volt 4 él húzva. Ellenkező esetben igaz értéket ad.

getBranchCount

Ez a függvény egy egész számot kap paraméterként, mely egy csomópont gráfbeli mélységére utal. Ez alapján a szám szerint visszaadja hogy mennyi irányba ágazzon el az adott csomópont. Első szinten 80% az esélye hogy 4 irányban ágazik el, ez szintenként 20%-al csökken. Ha nem 4 irányban ágazik el, véletlenszerűen generált számot ad vissza 0 és 2 között, mivel a csomópont szülőjéhez vezető élt ilyenkor nem számoljuk.

findMaxLevelNodes

A gráf legmélyebb szintjén lévő csomópontok megtalálására szolgáló függvény. Paraméterként megkapja a gráfot. Első lépésben beállítja a max értéket a gráf első csomópontjának szintjére, és inicializálja a maximális mélységben elhelyezkedő csomópontok vektorát. Ezután maximum kereséssel beállítja max értéket. Majd a gráf Nodes vektorán végig iterálva hozzáadja az ezen max értéknek megfelelő mélységű csomópontokat az eredmény vektorhoz. Végül ezzel a vektorral tér vissza.

DrawBStops

Buszmegálló rajzolására használt függvény. Egy csomópontot kap paraméterként. A kontextust felhasználva felrajzolja a canvasra egy zöld négyzet formájában. Visszatérési értéke nincs.

makeBusStops

Ez a függvény szolgál a buszmegállók elhelyezésére. Egyelőre csak a hozzávetőleges helyüket jelöli, nem az él közepére teszi a helyüket, hanem a csomópontokra. Paraméterként megkapja a gráfot. Először meghívja a findMaxLevelNodes függvényt, ebből a kapott értéket eltárolja egy változóba.

Kijelöl az útvonal kezdetének az előbb kapott vektorból egy véletlenszerű elemet. Ezt felrajzolja a DrawBStops függvénnyel. Ezután végig iterál a maximum mélységű tömbökön, maximum keresést végezve a kiindulási ponttól mért távolságot illetően. Ehhez a distance függvényt használja. Ha megvan a végpont azt is felrajzolja. Ezután elindul a kiindulási ponttól, sorra véve a csomópontok szüleit. Minden buszmegálló kirajzolása után kap egy véletlen értéket, amely megmondja hogy hány csomópont után kell rajzolnia mégegyet. Ha eljutott a gráf kezdőpontjáig megcsinálja ugyan ezt az út végpontjától is. Visszatérési értéke nincs.

drawCanvasNode

Egy paraméterként megkapott csomópont koordinátáinak megfelelő helyre négyzetet rajzol a canvasre.

drawCanvasEdge

Egy élt kap meg paraméterként, melynek kezdő és végpontjai között egyenes vonalat húz a canvasre.

4.2.4. A generáló függvény

A generálás kódjának fő része ebben a függvényben van. Ez egy paraméter nélküli függvény, visszatérési érték nélkül. A következőkben vázolom a működését.

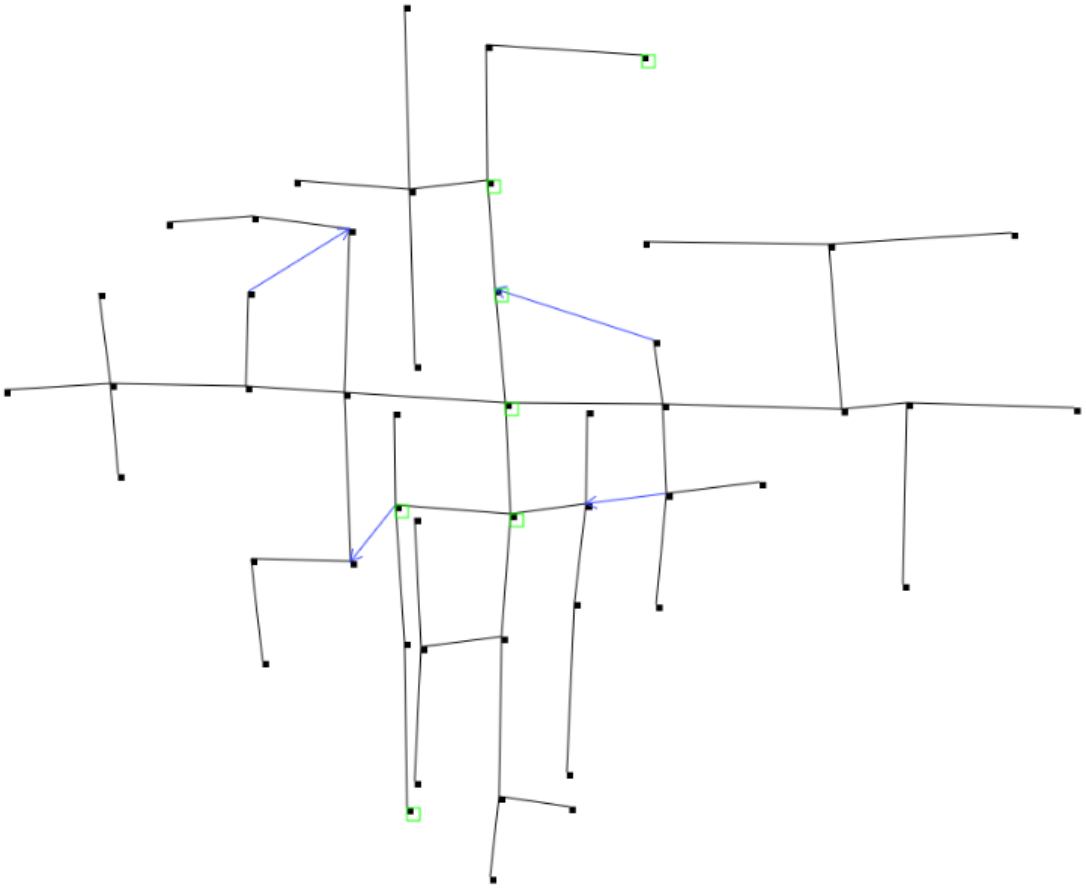
Először inicializálja a gráfot az osztály példányosításával, majd kijelöli a kezdő csomópontot, annak paramétereit beállítva fix értékekre, tehát 4 elágazású lesz, az 1000,500 koordinátákon.

Ezután egy for ciklussal végig iterál a csomópontokat tároló vektoron, minden csomóponton kezdetben megadja a connectedFrom értéket, tehát hogy az adott csomópont melyik irányból lett kiterjesztve.

Ha ez megvan, újabb ciklus kezdődik, mely a csomópont elágazásait terjeszti ki egyesével. Ezen belül megkapja minden iterációban a kiterjesztés irányát, az új csomópont elágazásainak számát a getBranchCount függvénnyel, valamint az új élen lévő sávok száma is kiszámítódik. Ilyenkor már létezik az új él és csomópont mint objektum, következő lépésben megnézi hogy az új él metszi-e bármely meglévő élt. Ez egy szimpla iteráció az éleket tartalmazó vektoron, az intersects függvény segítségével eldönti hogy történt-e metszés.

Amennyiben volt metszés, de az élt be lehetett húzni a metszett élnek a metszésponthoz közelebb álló csomópontjához, az új él megmarad, és egyirányú lesz, a csomópontot pedig elvetjük. Ha nem lehetett behúzni, vagy még így is történt metszés, akkor mindkettőt elvetjük. Ezen vizsgálat után az új él és csomópont amennyiben valamelyik is alkalmas maradt, hozzáadásra kerülnek a gráfbeli vektorhoz.

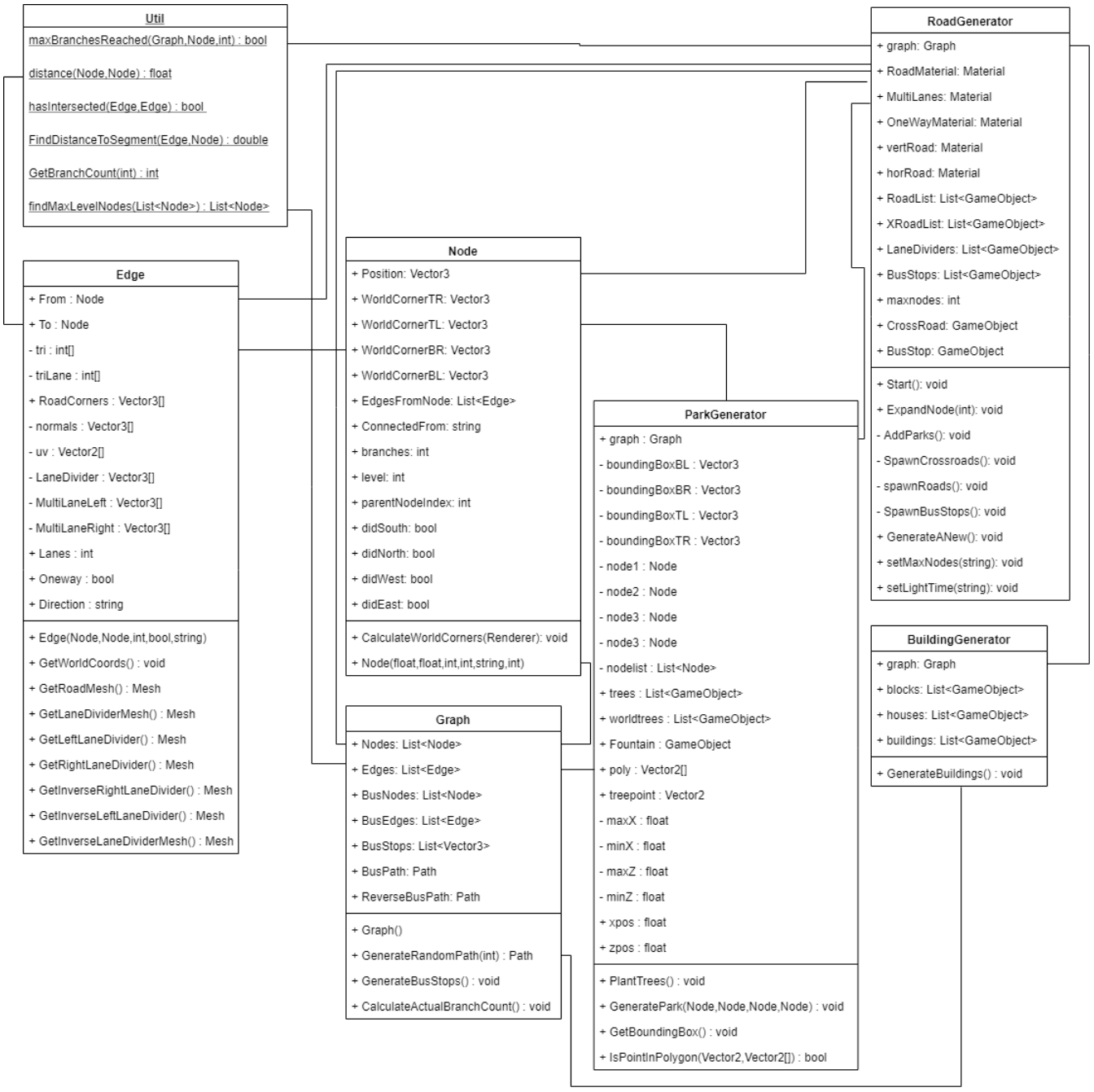
Miután a kellő mennyiségű csomópont ki lett terjesztve, a függvény meghívja a gráfon belüli drawAllNodes és drawAllEdges függvényeket, majd végül a makeBusStops függvényt. Ezzel kész a generálás, az elemek felkerültek a canvasre.



4.2. ábra. Kép a generáló algoritmus eddigi eredményéről

4.3. Az algoritmus implementálása Unity-ben

Mivel a szimulációt a Unity Enginel fogom elkészíteni, első lépésben az előbbi JavaScript implementációt ki kell bővíteni. Ez főleg a kirajzoló függvények átírását fogja jelenteni, amelyben 2D canvas helyett most már 3D-s megjelenítésben kell gondolkozni, de ezen kívül a C# programnyelvre való áttérés miatt lehetnek még minimális változtatások.



4.3. ábra. A generálásban résztvevő osztályok UML osztálydiagramja

4.3.1. Osztályok

Util

Létrehoztam egy Util nevű statikus osztályt, melynek fő célja az lenne hogy külön osztályba összegyűjti az összes használt matematikai Segédfüggvényt. Ebbe az osztályba került bele a maxBranchesReached, a distance, az intersects mostmár hasIntersected néven, a getBranchCount, valamint a findMaxLevelNodes függvények.

Továbbá bővült egy újab fügvénnyel, a FindDistanceToSegment-el. Ez a függvény

paraméterként egy élt és egy csomópontot vár. Visszatérési értékként a csomópont és a szakasz közötti legrövidebb távolságot adja.

Node

Ez az osztály az eredeti implementációhoz nagy mértékben hasonló. Egy lényeges különbség, hogy a csomópont pozícióját egy Vector3 típusú adattagban tárolom. Ez a típus 3 float típusú koordinátát tárol, az x,y, és z koordinátákat.

Hozzáadtam valamint 5 új adattagot, amelyek közül 4 a csomóponthoz tartozó objektum négy sarkát jelőlik. Ezek az adattagok Vector3 típusúak, kiszámításukra az osztályon belül elhelyeztem egy CalculateWorldCorners függvényt, melynek paraméterül meg kell adni a létező csomópont objektum Renderer komponensét.

Az ötödik új adattag egy Edge lista, mely a csomópontból kiinduló éleket tartalmazza, nem beleértve az ide érkezőket.

A CalculateWorldCorners függvény a következő képpen használja fel a renderert: a renderer objektum képes visszaadni a hozzá tartozó 3D mesht behatároló téglatestet. Ennek a téglatestnek lekérdezhető a maximum és minimum értéke, mely rendszerint a jobb felső és a bal alsó sarkát adja vissza. Ezen két pont tudatában kiszámítható a maradék két sarok is, a koordináták vegyes felhasználásával.

```
public void CalculateWorldCorners (Renderer renderer)
        {
            WorldCornerTR = renderer.bounds.max;
            WorldCornerTL = new Vector3(renderer.bounds.min.x,
            0.1f, renderer.bounds.max.z);
            WorldCornerBL = renderer.bounds.min;
            WorldCornerBR = new Vector3(renderer.bounds.max.x,
            0.1f, renderer.bounds.min.z);
        }
```

Edge

Ez az osztály esett át a legnagyobb átalakításon, ugyanis az utat ábrázoló mesht dinamikusan kell kirajzolni, amelyhez először el kell végezni néhány számítást.

A konstruktora változatlan maradt, viszont az osztály rengeteg adattaggal bővült. Először is a RoadCorners Vector3 típusú tömbbel, amely az útszakasz négy sarkának koordinátáit tárolja. A mesh megalkotásához szükséges a poligont alkotó háromszögek megadása, erre létrehoztam egy egész típusú tri tömböt.

Szintén hasonlóan, az úton jelen lévő felezővonalak, sávelválasztó vonalak jelölésére is szükség van egy objektumra. A felezővonal sarkait a LaneDivider tömbbe, a bal és jobb oldali sávelválasztó sarkait pedig a MultiLaneLeft és MultiLaneRight Vector3 típusú tömbben tárolom. Ezeknek a válaszvonalaknak külön háromszög tömböt is deklaráltam triLane néven.

A meshez szükséges még normálvektorokat tartalmazó, illetve uv tömböket is bevezettem.

Az osztály új metódusokkal bővült, ezek közül első a GetWorldCoords void típusú paraméter nélküli metódus. Ez a metódus annak függvényében, hogy a Direction

adattagban milyen irány van megadva, kiszámolja az általa összekötött csomópontok sarkai alapján az útszakasz sarkainak koordinátáit.

```
case "north":
                    RoadCorners[0] = From.WorldCornerTL;
                    RoadCorners[1] = From.WorldCornerTR;
                    RoadCorners[2] = To.WorldCornerBL;
                    RoadCorners[3] = To.WorldCornerBR;
                    LaneDivider[0] = ((RoadCorners[0] +
                    RoadCorners[1]) / 2) + Vector3.up * 5;
                    LaneDivider[1] = ((RoadCorners[0] +
                    RoadCorners[1]) / 2) + Vector3.up * 10;
                    LaneDivider[2] = ((RoadCorners[2] +
                     RoadCorners[3]) / 2) + Vector3.up * 5;
                    LaneDivider[3] = ((RoadCorners[2] +
                    RoadCorners[3]) / 2) + Vector3.up * 10;
                    if (Lanes == 2)
                    {
                        MultiLaneLeft[0] = ((RoadCorners[0] +
                        Vector3.up * 5) + LaneDivider[0]) / 2;
                        MultiLaneLeft[1] = ((RoadCorners[0] +
                        Vector3.up * 10) + LaneDivider[1]) / 2;
                        MultiLaneLeft[2] = ((RoadCorners[2] +
                        Vector3.up * 5) + LaneDivider[2]) / 2;
                        MultiLaneLeft[3] = ((RoadCorners[2] +
                        Vector3.up * 10) + LaneDivider[3]) / 2;
                        MultiLaneRight[0] = ((RoadCorners[1] +
                        Vector3.up * 5) + LaneDivider[0]) / 2;
                        MultiLaneRight[1] = ((RoadCorners[1] +
                        Vector3.up * 10) + LaneDivider[1]) / 2;
                        MultiLaneRight[2] = ((RoadCorners[3] +
                        Vector3.up * 5) + LaneDivider[2]) / 2;
                        MultiLaneRight[3] = ((RoadCorners[3] +
                        Vector3.up * 10) + LaneDivider[3]) / 2;
                    }
                    break;
```

Példa esetként, ha észak felé tart az út, akkor a kiindulási pont bal és jobb felső, a célpont bal és jobb alsó koordinátáira lesz szükség.

Ha az út két sávos, ezen sarkok alapján kiszámolja a felező- és sávelválasztó vonalak sarkait. Ezek megjelenésével nem kell foglalkozni, csak jelző értékkel rendelkeznek a járművek számára, ezért az útszakasz felett foglalnak helyet, egész pontosan 5 egységnyi értékkel felette.

Ezek után beállítja a tri és triLane értékeit, azaz hogy a háromszögek megalkotásához milyen sorrendbe járja be az adott pontokat.

A GetRoadMesh metódus maga az utat reprezentáló Mesh objektummal tér vissza. Paraméter nélküli metódus. Első lépésben meghívja a GetWorldCoords függvényt, ezzel beállítva az út sarkainak értéket. Majd létrehoz egy új Mesh objektumot, melynek csúcsait beállítja az előbb kiszámolt pontokra, háromszögek értéket pedig a tri tömbre.

A normálvektorok itt nem kapnak nagy jelentőséget, Z tengely szerinti negatív egyenes értéket adtam nekik. Az uv koordinátákat szimplán a mesh négy sarkának jelöltem el.

```
public Mesh GetRoadMesh()
        {
            GetWorldCoords();
            Mesh mesh = new Mesh();
            mesh.vertices = RoadCorners;
            mesh.triangles = tri;
            normals[0] = -Vector3.forward;
            normals[1] = -Vector3.forward;
            normals[2] = -Vector3.forward;
            normals[3] = -Vector3.forward;
            mesh.normals = normals;
            uv[0] = new Vector2(0, 0);
            uv[1] = new Vector2(1, 0);
            uv[2] = new Vector2(0, 1);
            uv[3] = new Vector2(1, 1);
            mesh.uv = uv;
            return mesh;
        }
```

A GetLaneDividerMesh, GetLeftLaneDivider, valamint a GetRightLaneDivider metódusok az előbbivel hasonló módon funkcionálnak, csak a csúcsok halmaza változik.

Ám ezekkel a vonalakkal kapcsolatban szükség volt létrehoznom három Inverse metódust is, a GetInverseLeftLaneDivider, GetInverseRightLaneDivider, és a GetInverseLaneDividerMesh függvényeket. Erre az a magyarázat, hogy ezek alapvetően síkok, azaz a hátlapjaik nem láthatóak.

Bár a Unity tartalmaz megoldást ütközők konvexé alakítására, ez ebben az esetben nem mindig működik, mert gyakran a válaszvonal négy csúcsai koplanáris pontok. Ezért a következő megoldást választottam: a létrehozott síkokkal megegyező pozícióban létrehozom a síkok ellentétes irányba néző változatát. Ezt a feladatot látják el ezek az inverse metódusok, melyek az adott háromszög tömböt visszafelé járják be.

Graph

A gráfot leíró osztály bővült néhány új adattaggal.

Külön Node és Edge listában tartalmazza mostmár a buszok útvonalát leíró csomópontokat és éleket. A buszmegállok koordinátái tárolására egy Vector3 típusú listát is bevezettem.

Továbbá ebben az osztályban tárolom a buszok számára kirendelt útvonalat is, melyet egy Path nevű osztállyal írok le. Ezt a későbbiekben részletezném a szimulációval kapcsolatban.

Az osztály konstruktora üres listával inicializálja az összes adattagot.

Egy új metódus került az osztályba, a GenerateRandomPath. Ennek egy egész szám típusú paramétert kell megadni, ez lesz az útvonal maximális hossza. A metódus visszatérési értéke Path típusú, ez fogja tartalmazni az útvonal adatait.

Működésében egy Node és egy Edge listában tárolja el az útvonal által bejárt csomópontokat és éleket. Először választ egy véletlenszerű csomópontot, majd a csomó-

pontból kiinduló élek közül választ egyet, és az él túlsó végén lévő csomóponthoz megy tovább. Ezt addig folytatja, amíg el nem éri a megadott maximális úthosszat, vagy ameddig már nem lehet tovább menni egy adott csomópontból. Végül a Node és Edge listát felhasználva visszadja a Path objektumot.

```
while(currentsize < length && nextnode.EdgesFromNode.Count > 1)
            {
                Edge nextedge = nextnode.
                EdgesFromNode[Random.Range(0, nextnode.EdgesFromNode.
                Count)];
                if(nextedge.From != previousNode && nextedge.To !=
                previousNode)
                {
                    previousNode = nextnode;
                    nextnode = nextnode == nextedge.From ? nextedge.To :
                    nextedge.From;
                    edges.Add(nextedge);
                    nodes.Add(nextnode);
                    currentsize++;
                }

            }
```

A GenerateBusStops metódusban történt pár változtatás. Ezt is úgy írtam meg, hogy a Path objektumot használja, ezért a metódus az osztályon belüli két busz útvonalat leíró adattag értékét állítja be.

Működésében nagyjából azonos, egy fontos változtatással, miszerint a megállók elhelyezése közben bejárt csomópontokat és éleket eltárolja listában. Az útvonal vége felőli bejárt utat megfordítás után hozzáadom a kiindulás felőli úthoz, ez adja a busz teljes útvonalát.

Ezt az útvonalat megfordítom teljes egészében, így megkapom a buszok útvonalát ellentétes irányban. Mindkettő útvonal tárolódik a végén.

```
nodesfromEnd.RemoveAt(nodesfromEnd.Count - 1);
nodesfromEnd.Reverse();
BusNodes.AddRange(nodesfromEnd);
edgesfromEnd.Reverse();
BusEdges.AddRange(edgesfromEnd);
this.ReverseBusPath = new Path(BusNodes, BusEdges);
BusEdges.Reverse();
BusNodes.Reverse();
this.BusPath = new Path(BusNodes, BusEdges);
```

A CalculataActualBranchCount metódusra azért volt szükség, mert generálás után a csomópont branches adattagja nem minden esetben reflektálja a hozzá tartozó tényleges élek számát, például ha az egyik kiterjesztése érvénytelen volt. Ez a metódus megszámolja ténylegesen hány él indul és/vagy érkezik az adott csomópontba, és ez alapján beállítja a branches értékét.

```
public void CalculateActualBranchCount()
```

```
        {
            foreach (Node n in Nodes)
            {
                n.branches = 0;
                foreach(Edge e in Edges)
                {
                    if((e.To.Position == n.Position) ||
                    (e.From.Position == n.Position && !e.Oneway))
                    {
                        n.branches++;
                    }
                }
            }
        }
```

BuildingGenerator

Ez az osztály felelős az utak menti épületek elhelyezéséért. Adattagjai közé tartozik az úthálózatot reprezentáló Graph típusú graph, a kertes- és blokkházak sablonjait tartalmazó blocks és houses GameObject típusú lista, valamint a legenerált épületeket tartalmazó buildings lista.

Egyetlen metódusa van, a GenerateBuildings paraméter nélküli, void visszatérésű metódus. Ebben a metódusban történik a generálás több lépésben. Első lépésben megnézi, van-e legalább 30 csomópont a gráfban. Hogyha igen, akkor a gráf azon szintjein, amelyek nagyobbak, mint a csomópontszám 1/25-e, blokkházakat helyez el, ahol kisebb ott kertesházakat. Hogyha kevesebb mint 30 csomópontból áll a gráf, akkor mindenhova kertesházat fog elhelyezni. Az épületek pozíciójának kijelölésére kiszámolja az adott élet reprezentáló vektort, annak vég- és kezdőpontjának különbségével. Ezután kiszámol egy ezen vektorra merőleges vektort, mely az él vektorának és az y tengelyt reprezentáló vektornak vektoriális szorzata. Ennek kiszámítására tartalmaz a Vector3 struktúra egy Cross nevű metódust. Az út vektorát 9 részre osztom fel, ezen részek hosszát egy buildingslot nevű vektor tárolja. Ezen adatok kiszámítása után egy for cikluson belül elhelyezi az épületeket. A ciklusváltozó kezdeti értéke megegyezik a buildingslot vektorral, amely minden iteráció után a buildingslot értékével nő. Kilépési feltételként adtam azt, hogy a ciklusváltozóban tárolt vektor hosszának nagyobbnak kell lennie, mint az élt jelölő vektor hosszánál 20 egységgel kisebb vektor. Ezzel elkerülöm azt, hogy útkereszteződéshez túl közel kerüljön egy épület. Mielőtt elhelyezne egy épületet, megnézi hogy az adott területen megadott területen belül létezik-e valamilyen más objektumhoz tartozó Collider komponens. Ez annak érdekében kell, hogy ne kerüljön fa épületen belülre, illetve ne legyen két épület túl közel egymáshoz.

A kertesházak elhelyezéséért felelős blokk a következő, ezzel működésben megegyezik a blokkházaké is, csupán a vektorok méretében van különbség:

```
Vector3 theRoad = e.To.Position — e.From.Position;
Vector3 buildingslot = theRoad / 10f;
Vector3 side = Vector3.Cross(theRoad, Vector3.up).normalized;
for (Vector3 offset = buildingslot; offset.magnitude < theRoad.
magnitude — 20f; offset += buildingslot)
```

```
{
    if (!Physics.CheckBox(e.From.Position + offset + side * 14f +
    Vector3.up * 6f, new Vector3(5f, 5f, 5f)))
    {
        buildings.Add(Instantiate(houses[Random.Range(0, 2)], e.
        From.Position + offset + side * 14f, Quaternion.identity));
    }
    if (!Physics.CheckBox(e.From.Position + offset - side * 14f +
    Vector3.up * 6f, new Vector3(5f, 5f, 5f)))
    {
        buildings.Add(Instantiate(houses[Random.Range(0, 2)], e.
        From.Position + offset - side * 14f, Quaternion.identity));
    }
```

ParkGenerator

A ParkGenerator osztály tartalmazza a fák és a parkok generálásáért felelős metódusokat. Adattagjai közé tartozik a gráf, a park négy sarkát jelölő Node típusú csúcspontok, valamint Vector3 típusú koordináták, ezen négy pont tárolására alkalmas Node lista, valamint a koordináták tárolására alkalmas Vector2 tömb, a generált fák tárolásáért felelős GameObject típusú worldtrees tömb, és végül a fák sablonjait tartalmazó trees GameObject típusú lista.

Az osztály PlantTrees metódusa véletlenszerű pozíciókban fákat helyez el, a gráfot befoglaló négyzet területén belül. Első lépésben kiszámolja ezen négyzet négy sarkát, linq függvények segítségével. Ezután a gráf csomópontszámának ötszörösének megfelelő mennyiségű fát helyez el.

```
public void PlantTrees()
    {
        float minx = graph.Nodes.OrderByDescending(
        x => x.Position.x).LastOrDefault().Position.x;
        float maxx = graph.Nodes.OrderByDescending(
        x => x.Position.x).FirstOrDefault().Position.x;
        float minz = graph.Nodes.OrderByDescending(
        x => x.Position.z).LastOrDefault().Position.z;
        float maxz = graph.Nodes.OrderByDescending(
        x => x.Position.z).FirstOrDefault().Position.z;
        for (int i = 0; i < 5 * graph.Nodes.Count; i++)
        {
            xpos = Random.Range(minx, maxx);
            zpos = Random.Range(minz, maxz);
            treepoint = new Vector2(xpos, zpos);
            if (!Physics.CheckBox(new Vector3(treepoint.x, 6f,
            treepoint.y), new Vector3(5f, 5f, 5f)))
            {
                worldtrees.Add(Instantiate(trees[Random.Range(0, 3)],
                new Vector3(treepoint.x, 0.1f, treepoint.y),
                Quaternion.identity));
```

```
            }
        }
    }
```

A GetBoundingBox függvény szerepe a kapott csomópontok alapján meghatározni hogy azok közül melyik pont a park melyik sarkát jelenti. Így megkapjuk a bal alsó és felső, valamint jobb alsó és felső sarkát a parknak.

```
private void GetBoundingBox()
    {
        maxX = nodelist.OrderByDescending(
        x => x.Position.x).FirstOrDefault().Position.x;
        minX = nodelist.OrderByDescending(
        x => x.Position.x).LastOrDefault().Position.x;
        maxZ = nodelist.OrderByDescending(
        x => x.Position.z).FirstOrDefault().Position.z;
        minZ = nodelist.OrderByDescending(
        x => x.Position.z).LastOrDefault().Position.z;
        boundingBoxBL = new Vector3(minX, 0.1f, minZ);
        boundingBoxBR = new Vector3(maxX, 0.1f, minZ);
        boundingBoxTL = new Vector3(minX, 0.1f, maxZ);
        boundingBoxTR = new Vector3(maxX, 0.1f, maxZ);
    }
```

Az IsPointInPolygon függvény két paramétert kap, egy Vector2 típusú pontot, valamint egy Vector2 tömböt amely tartalmazza egy poligon pontjait. Visszatérési értéke igaz, hogyha a pont benne van a poligonban, hamis hogyha nincs. Ennek meghatározására megnézi a poligon összes élét jellemző szakaszra, hogy a pont ezen szakasz végpontjai közé esik-e, valamint ha igen, akkor a szakasz megfelelő oldalán van-e.

```
public bool IsPointInPolygon(Vector2 point, Vector2[] polygon)
    {
        int polygonLength = polygon.Length, i = 0;
        bool inside = false;
        float pointX = point.x, pointY = point.y;
        float startX, startY, endX, endY;
        Vector2 endPoint = polygon[polygonLength - 1];
        endX = endPoint.x;
        endY = endPoint.y;
        while (i < polygonLength)
        {
            startX = endX; startY = endY;
            endPoint = polygon[i++];
            endX = endPoint.x; endY = endPoint.y;
            //
            inside ^= (endY > pointY ^ startY > pointY)
                      && ((pointX - endX) < (pointY - endY) *
                      (startX - endX) / (startY - endY));
        }
        return inside;
```

```
    }
```

A parkok generálását a GeneratePark függvény végzi. Paraméterként megkapja a parkot leíró négy csomópontot. Ezen négy csomópont alapján meghatározza a park négy sarkát a GetBoundingBox függvénnyel. Miután ez megvan, egy for ciklusban elhelyez 100 darab fát a park területén belül véletlenszerű pozícióban. Az objektumok elhelyezése előtt megnézi, hogy ütközne-e bármely más objektummal, hogyha igen akkor új pozíciót kap. Végül hasonló módszerrel elhelyez egy darab szökőkutat is a parkban.

```
do
        {
            xpos = Random.Range(minX, maxX);
            zpos = Random.Range(minZ, maxZ);
            treepoint = new Vector2(xpos, zpos);
        } while (!IsPointInPolygon(treepoint, poly) || Physics.
        CheckBox(new Vector3(treepoint.x, 6f, treepoint.y),
        new Vector3(5f, 5f, 5f)));
        worldtrees.Add(Instantiate(Fountain, new Vector3(treepoint.x,
        0.1f, treepoint.y), Quaternion.identity));
```

RoadGenerator

Többek között ebbe az osztályba került a fő generáló függvény, valamint a kirajzolásért felelő függvények.

Adattagjai közé tartozik maga a gráf, mint Graph típus, külön Material típus három féle útra (egy sávos, két sávos, egyirányú), négy GameObject lista, melyek az útszakaszok, az útkereszteződések, a választóvonalak, és a buszmegállók tárolására vannak, valamint egy sablon objektum elhelyezésére alkalmas CrossRoad és BusStop GameObject típusú adattagok.

Az osztályon belüli Start metódus az ezen osztály komponensként tartalmazó objektum létrejötte utáni legelső képkockában fut le, pontosan egyszer.

Ebben a metódusban először elhelyezem a kezdeti csomópontot, majd ameddig kevesebb csomópont létezik mint a megadott maximum, az ExpandNode metódussal kiterjesztem a soron következő csomópontot.

Ha ez megtörtént, újra számoltatom a csomópontok elágazásainak számát a gráfon belüli CalculataActualBranchCount metódussal, majd meghívom a SpawnCrossroads, spawnRoads, a gráfon belüli GenerateBusStops, majd a SpawnBusStops metódusokat a csomópontok, útszakaszok, és a buszmegállók kirajzolásához. Ha ez megtörtént, beállítom a BuildingGenerator és a ParkGenerator komponensek graph értékét az ebben az osztályban eltárolt gráfra, majd végül meghívom ezeknek épület és fa generáló metódusait, valamint ezen osztályban a parkok elhelyezéséért felelős AddParks metódusát. Utolsó lépésben a talajt jelölő sík méretét a csomópontok számának függvényében beállítom a megfelelő értékre.

```
void Start()
    {
        graph.Nodes.Add(new Node(-500, -500, 4, 0, "none", 0));
        for(int i = 0; graph.Nodes.Count < maxnodes; i++)
        {
```

```
            ExpandNode(i);
        }
        graph.CalculateActualBranchCount();
        SpawnCrossroads();
        spawnRoads();
        graph.GenerateBusStops();
        SpawnBusStops();
        gameObject.GetComponent<BuildingGenerator>().graph = this.graph;
        gameObject.GetComponent<BuildingGenerator>().GenerateBuildings();
        gameObject.GetComponent<ParkGenerator>().graph = this.graph;
        gameObject.GetComponent<ParkGenerator>().PlantTrees();
        AddParks();
        GameObject.FindGameObjectWithTag("Terrain").
        transform.localScale = new Vector3(1000 + 2*maxnodes, 1f,1000 +
        2*maxnodes);
    }
```

Az ExpandNode metódusba került bele a generáló algoritmus fő része, ez egy visszatérési érték nélküli metódus, mely egy egész számot vár paraméterként. Ez az egész szám annak a csomópontnak az indexét jelöli, amelyet ki fog terjeszteni. Funkcionalitásában nem történt változás.

A SpawnCrossRoads metódus felelős az útkereszteződések kirajzolásáért. Visszatérési érték nélküli, paraméter nélküli metódus. Az Instantiate metódussal létrehozza egy megadott pontban, az eltárolt sablon útkereszteződés egy másolatát. Ezután kiszámolja a sarkait a CalculateWorldCorners metódussal. Végül a jelzőlámpa irányításáért felelős CrossRoadController szkriptnek átadja a csomópontot.

Az AddParks metódus void típusú és paraméter nélküli. Feladata megtalálni azokat a négy csomópontból álló részeket a gráfban, melyek kört alkotnak, majd ezeket átadni a ParkGenerator osztály generáló metódusának. Végignézi a gráf összes élét, ha talál egyirányút az garantálja hogy egy kör része. Szintén elkezdi nézni az élek listáját előlről, ha talál egy élt ami csatlakozik a talált egyirányú él egyik végpontjához, és nem egyezik meg vele, az lesz a kör második éle. Ezután újra elkezdi nézni a gráf éleit, ha talál egyet amelyik csatlakozik az egyirányú él valamelyik végpontjához, és nem egyezik meg sem az egyirányú, sem a már talált második éllel, ez lesz a kör harmadik éle. Végül keres egy negyedik élt, amely két végpontja megegyezik a második és harmadik él végpontjai közül valamelyikkel, és az eddigi élek közül egyikkel sem egyezik meg. Ezt megtalálva ezen él két végpontja, valamint az egyirányú él két végpontja lesznek a park csúcsai.

Az utolsó él meghatározása:

```
foreach (Edge edg in graph.Edges)
{
    if (edg != ed && edg != edge && edg != e && (edg.From == ed.From ||
    edg.From == ed.To || edg.To == ed.From || edg.To == ed.To) &&
    (edg.From == e.From || edg.From == e.To || edg.To == e.From ||
    edg.To == e.To))
    {
        node3 = edg.From;
        node4 = edg.To;
        gameObject.GetComponent<ParkGenerator>().GeneratePark(node1,
```

```
        node2, node3, node4);
    }
}
```

A spawnRoads void visszatérésű, paraméter nélküli függvény dinamikusan képez mesheket az Edge listában tárolt élek adataiból. A materiált beállítja annak függvényében, hogy az út egyirányú, két sávú, vagy egy sávú. A választóvonalak renderer komponensét kikapcsolja, mivel azok megjelenítésére nincs szükség. Példa egy választóvonal létrehozására:

```
LaneDividers.Add(GameObject.CreatePrimitive(PrimitiveType.Quad));
LaneDividers[LaneDividers.Count - 1].GetComponent<MeshFilter>().mesh =
edge.GetLaneDividerMesh();
LaneDividers[LaneDividers.Count - 1].GetComponent<MeshCollider>().
sharedMesh = LaneDividers[LaneDividers.Count - 1].
GetComponent<MeshFilter>().sharedMesh;
LaneDividers[LaneDividers.Count - 1].GetComponent<Renderer>().
enabled = false;
LaneDividers[LaneDividers.Count - 1].tag = "LaneDivider";
```

A SpawnBusStops metódus void típusú, és paraméter nélküli. Az Instantiate metódus segítségével létrehozza a sablonként eltárolt megálló objektum másolatát a buszmegállók pozíciót tartalmazó listának megfelelő helyeken.

```
private void SpawnBusStops()
    {
        foreach(Vector3 pos in graph.BusStops)
        {
            BusStops.Add(Instantiate(BusStop, pos, Quaternion.identity));
        }
    }
```

A GenerateANew metódus a felhasználói felület kezelését látja el. A Generate Anew gombra kattintva lefut ez a metódus, mely törli az összes létrehozott objektumot, a gráf tartalmát üríti, majd meghívja a start metódust amellyel egy új úthálózatot generál.

Szintén a UI-hoz kapcsolódó két metódus a setMaxNodes, mely a beírt értékre beállítja a maximálisan generálandó csomópontokat. Valamint a setLightTime, amely átadja a jelzőlámpák szkriptjének a váltáshoz használandó időtartamot.

4.3.2. A Scene létrehozása

Ahhoz hogy a megírt metódusokat használni lehessen, hozzá kell rendelni valamilyen GameObject objektumhoz. Ezeket az objektumokat lehet dinamikusan is létrehozni C# szkripten belül, de minden esetben szükség van legalább egy GameObjectet létrehozni a Unity editorjában, enélkül nem lehet szkriptet futtatni.

A GameObjecthez rendelt szkriptek az objektum létrejöttekor meghívják a Start metódusokat, majd minden egyes képkockafrissítés után az Update metódust.

Az alap scene tartalmaz egy Main Camera objektumot, valamint egy Directional Light fényforrást. Létrehoztam ezek mellé egy üres GameObjectet, melyhez komponensként hozzárendeltem a RoadGenerator forrásfájlját. Ezt az objektumot elneveztem

Manager-nek, ezt fogom használni a szimuláció kezelésére, ezen keresztül hozok létre, illetve törlök objektumokat. A Managerhez hozzárendelt szkript a Scene betöltődésekor most már meghívja a Start metódust, mely végrehajtja a generálás lépéseit.

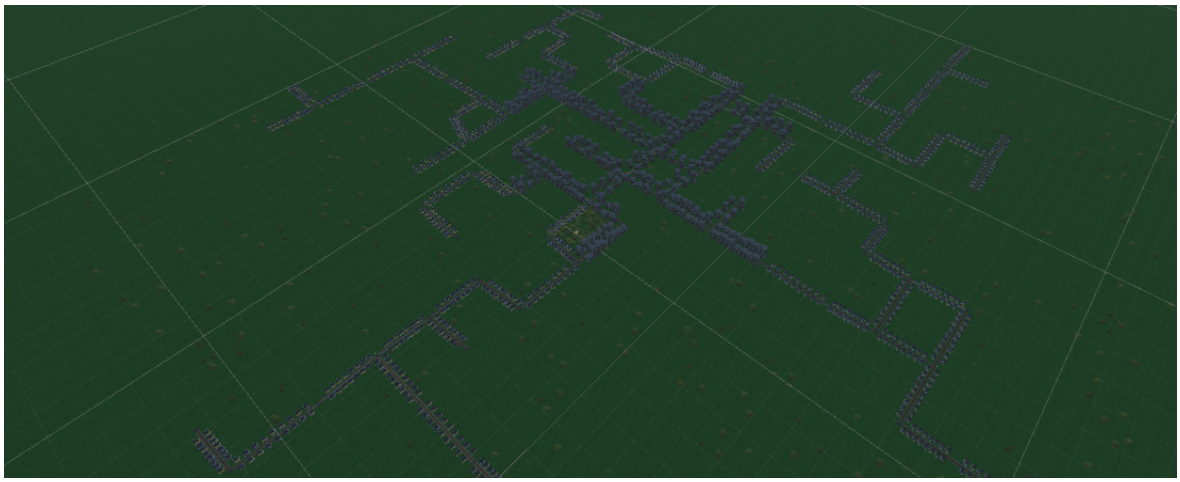
Viszont ehhez még szükség van a szkript adattagjainak feltöltésére. Amint írtam, ez az osztály tartalmaz 3 Material típust, és 2 GameObject típust amelyet az editorból fel kell tölteni.

Először is elkészítettem a materialokat, felhasználva egy alap útburkolat textúrát. Ezt a textúrát kissé átszerkesztve előállítottam a 3 úttípushoz szükséges materialokat, majd hozzárendeltem őket a szkripthez.

Ezek után létrehoztam egy új objektumot a scene-ben, mely egy egyszerű négyzet alakú síklap. Ezen síklapot fogom felhasználni a csomópont jelölésére alkalmas sablonként. Létrehoztam ennek az objektumnak négy gyerekelemét, ezek a négy oldalán lévő jelzőlámpáknak felelnek meg. Ezeket az egyszerűség kedvéért egy alap kapszula alakzattal reprezentálom. Elheyeztem rajta egy sötétszürke, útra emlékeztető materialt. Ezt az objektumot a projekt erőforrásai közé helyezve létrejön róla egy sablon, ezt már hozzá lehet rendelni a generáló szkripthez.

A buszmegállók sablonját is hasonlóan készítettem el, egy szimpla kapszula alakzattal jelölöm. Ehhez az objektumhoz adtam egy új BoxCollider komponenst, mellyel a megálló által behatárolt területet tudom eljelölni. Ennek a komponensnek az isTrigger tulajdonságát engedélyezve lehet a fizikai motor igénybe vétele nélkül ellenőrizni azt, hogy mely objektumok léptek a területére. Ezzel készen léve hozzáadtam a sablont a szkripthez.

A működést kipróbálva az alábbi eredményhez jutottam:



4.4. ábra. Kép a generáló algoritmus eredményéről Unityben

5. fejezet

Tervezés, implementáció

5.1. A WebGL-ről

A WebGL, azaz Web Graphics Library, egy 2D és 3D grafikus renderelésre képes JavaScript alapú API, amely lehetőséget ad interaktív grafika megjelenítésére a webböngészőkben, bármilyen más harmadik féltől származó bővítmény használata nélkül. Képes teljes mértékben hasznát venni a hardveres gyorsításnak, a képfeldolgozási utasításokat a GPU-val hajtatja végre.

Első verzióját 2011 március 3.-án adta ki a Khronos Group, azóta is ők fejlesztik. A Khronos Group emelett még sok más grafikai alkalmazásprogramozási felületet készít, melyek közül a két legismertebb talán az OpenGL és a Vulkan.

A WebGL az OpenGL ES-en alapszik, mely amint a nevéből is adódik, beágyazott rendszerekhez készült. Széles körben használatos főleg mobiltelefonok, tableteken, és más hordozható készülékeken. A WebGL első verziója az OpenGL ES 2.0-án alapult, azóta már kiadták a 2.0-ás verziót, amely az OpenGL ES 3.0-t vette alapul.

Az 1.0-ás verziót már szinte az összes modern webböngésző támogatja, valamint a HTML5 szabványnak is része. A 2.0-ás verziót még sok böngésző csak részlegesen támogatja, néhány pedig, mint például a Microsoft Edge, egyáltalán nem.

A WebGL API nagyon alacsony szintű természete miatt nem célszerű natívan programozni. Az alap alakzatok megjelenítése is rengeteg időt venne igénybe ilyen megvalósítással. Emiatt is készült rengeteg függvénykönyvtár hozzá, amelyek lényegesen megkönnyítik a fejlesztési folyamatot, magasszintű felületet biztosítanak egyszerű alakzatok megjelenítésére, textúra beolvasására, UV map előállításához, projekciós és modelview mátrixok kiszámolásához.

Az egyik ilyen híres függvénykönyvtár a Three.js. A Three.js nagyon bő funkcionalitással rendelkezik, egyszerűen létrehozható vele scene, renderer, kamera. Képes irányított és pontszerű fényforrások kezelésére. Megkönnyíti a materialok használatát, animációk létrehozását, és még sok más hasznos funkciót tartalmaz.

A Three.js-ben először is egy scene-t kell létrehoznunk, ebben az objektumban fog történni a kirajzolás. Ezután jön a renderer létrehozása, majd egy kamera objektum hozzáadása a scenehez. Ha ezek megvannak, a meglévő scenehez hozzá lehet adni a kívánt objektumokat, azaz fényforrásokat, irányítást megvalósító objektumokat, valamint mesheket. A mesheket a Three.js-ben lehetőség van JSON objektumokból betölteni, majd dinamikusan hozzájuk rendelni materialt. Az animate függvényben meghívva a renderer kirajzoló függvényét frissül a megjelenés.

5.2. Unity

A Unity Engine egy, a Unity Technologies által fejlesztett játékmotor. Elsőként az Apple Worldwide Developers Conference nevű rendezvényén mutatták be 2005-ben. Célja kezdetektől a játékfejlesztés folyamatának megkönnyítése. Ennek érdekében az új fejlesztők számára könnyen tanulhatóvá alakították ki a működését, sok "Drag and Drop" funkció található benne. Másik szempontból viszont több platformra is egyszerű vele fejleszteni, mára már több mint 25 különböző platformot támogat, ezzel megkönnyítve a cross-platform alkalmazások fejlesztését. Maga a motor elsősorban bár játékmotornak lett kialakítva, de sok más területen is használják, beleértve a filmipart, autóipart, és az építőipart.

Kezdetben egy Boo nevű programozási nyelvet használt a különböző szkriptekhez, de ezt később felváltotta a JavaScripten alapuló UnityScript nevű nyelv. Mára viszont a UnityScriptnek is csak részleges támogatottsága van. Ez annyit tesz, hogy nem lehetséges többé ilyen szkripteket létrehozni a motor újabb verzióiban, de a régebbi projekteket amelyek ilyen típusú szkripteket tartalmaznak még tudja kezelni. A használatos nyelv helyette a C#, melynek támogatottsága előreláthatólag a fejlesztők nyilatkozatai szerint a jövőben csak bővülni fog. Maga a motor futási időben C++ kódot futtat, ebbe ágyazódnak be wrapperekkel a más nyelven megírt szkriptek.

Licenszét tekintve a szoftvernek három kiadása van: az ingyenes Personal, és a fizetős Pro valamint Plus verziók.Ezek között sok különbség nincsen, a fizetős kiadások főbb újdonságait a fejlesztők által nyújtott plusz szolgáltatások teszik ki, mint például az online tárhely, vevőszolgálatnál előnyben részesülés, oktató tanfolyamok, stb. Viszont léteznek korlátozások az egyes kiadások felhasználására. Az ingyenes változatot csakis személyes használatra, vagy évi $100,000 bevételt meg nem haladó üzleti célra lehet felhasználni. A Plus verziónál a megengedett bevételi korlát $200,000, míg a Pro verzió korlátlanul használható.

Amint említettem sok platformot támogat, ezek közé tartozik a WebGL is. A továbbiakban ezen platform sajátosságait részletezném.

5.2.1. Unity WebGL

A WebGL magasabb szintű absztrakciós keretrendszerei közé tartozik néhány játékmotor is. Ezek közül a két legismertebb az Unreal Engine és a Unity Engine. Ezek WebGL-beli teljesítménye bár nem egyezik meg az olyan grafikus APIk-éval mint a Direct3D, Vulkan, vagy OpenGL, mégis egy jelentős megszorítást küszöböl ki azzal, hogy teljesen közvetlen hozzáférést biztosít a számítógép grafikus kártyájához.

Implementáció szempontjából nincs különbség a platformok között Unityben, a sajátosságok a fordításkor jönnek elő. WebGL build esetén a kimeneti fájlok közé tartozik egy HTML fájl amelybe be van ágyazva a WebGL applikáció, egy TemplateData mappa amely tartalmazza a weboldal és a lejátszó megjelenését leíró fájlokat, valamint egy Build mappa, amely tartalmazza a WebGL tartalom betöltéséhez szükséges JavaScript fájlokat, valamint a projekt által használt összetevőket, tömörített unityweb állományokban.

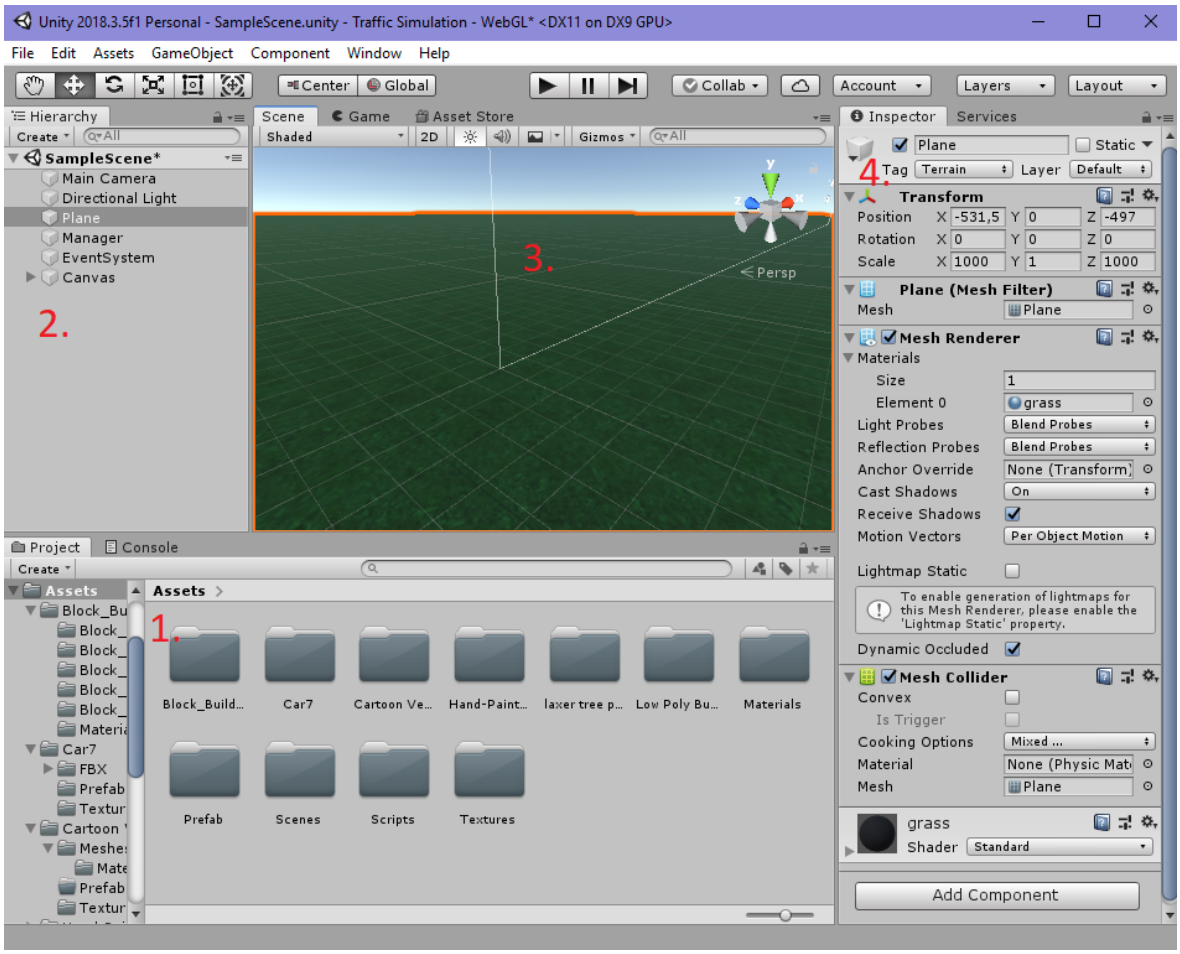
Mivel a WebGL csakis JavaScript kódot tud futtatni, a Unity pedig C# szkripteket használ, futási időben pedig C++ kódot futtat, szükséges a build folyamán a kód átkonvertálása. Ez több lépésben történik, először is a .NET kódot egy IL2CPP, azaz Intermediate Language To C++ nevű technológiával átalakítja C++ forráskód-

dá. Ha ez megtörtént, a létrejött C++ kód az Emscripten nevű compiler segítségével átfordul JavaScript kóddá. Az így létrejött forrás már futtatható egy weboldal WebGL lejátszójában.

5.2.2. Unity Editor

A program működésének kialakításában a szkripteken kívül a másik nagy szerepet az editor tölti be. Az editorból elérhető a projekthez hozzáadott összes erőforrás, azaz a textúrák, materialok, meshek, szkriptek, valamint az összes létrehozott scene. Az editor beépítve tatalmazza a Unity Asset Store-t, ahonnan le lehet tölteni fizetős és ingyenes komponenseket, legyen az szkript, mesh, vagy bármilyen más építőelem. A projekthez innen szereztem be az autók, a buszok, az épületek, a fák, és a szökőkutak modelljét.

Az editorban egy scene megnyitása után megjelenik annak jelenlegi állapota. Ez azt jelenti, hogy kilistázza a benne elhelyezett GameObject objektumokat, valamint megjeleníti azokat a világban. A scene 3D-s megjelenítőjében szabadon mozgatható bármely objektum, egyet kiválasztva pedig megnyílik a rá vonatkozó inspector. Az inspector kilistázza a kiválasztott GameObject összes komponensét, valamint azoknak paramétereit. Ezeket a paramétereket innen meg lehet változtatni, melynek hatása egyből meg is jelenítődik. Az objektumoknak kötelezően lennie kell Transform komponensének, mely a világban lévő pozíciójukat írja le x, y, és z koordinátákkal, valamint szintén ezen három tengelyen való forgatásukat és skálázásukat. Az inspectorban lehetséges hozzáadni objektumokhoz új komponenst, mely lehet beépített elem, vagy egy megírt C# szkript. A scene-en belül lehetséges az objektumok között hierarchia kialakítása is, azaz egy objektum lehet más objektumoknak a szülője. Az objektumokat el lehet látni tag-ekkel, melyek segítségével szkripteken belül egyszerűen beazonosíthatóvá válnak.



5.1. ábra. Az editor részei megszámozva: 1: Projekt erőforrásai 2: A Scene összes objektuma 3: A Scene objektumainak világon belüli reprezentációja 4: Inspector

A projekt erőforrásaihoz lehetséges hozzáadni sablon GameObject objektumokat is, azaz a scene-en belül létrehozott egyik objektum jelenlegi állapotát mentjük le, az összes komponensével, azoknak jelenlegi állapotaival együtt. Az így lementett erőforrás szkriptek segítségével dinamikusan hozzáadható a scene-hez futási időben. A szkripteken belüli publikus adattagok inicializálhatóak az editorból, az inspector segítségével. Ezen adattagok bármilyen típust felvehetnek, beleértve minden különböző típusú erőforrást. Ezzel lehetséges referenciát átadni a szkripteknek egy adott GameObjectről, materialról, vagy bármilyen scene-en belüli objektum komponenséről.

5.3. Szimuláció tervezése

A szimuláció felépítését tekintve a következők mondhatóak el. Először is felhasználja a már legenerált úthálózatot, azon a járművek mozgásának alapjaként szolgál egy véletlenszerű részgráf, amely a bejárandó útvonalat reprezentálja. A járművek mozgásához felhasználom a Unity fizikai motorját, amely kvázi realisztikusan képes szimulálni a kerekek mozgásából, azok ívéből a jármű haladási irányát, sebességét, illetve a fékezés szimulálásához felhasználható eszközt nyújt, mellyel ténylegesen csak a kerekek forgásának sebességét csökkentem, ami közvetetten csökkenti a jármű sebességét.

5.3.1. Szenzorok

A járműveknek a környezettel és egymással való interakciója érdekében úgynevezett szenzorokat használok. Ez nem más, mint a fizikai motor egy raycast függvénye, mely egy pontból kiindulva a megadott irányvektornak megfelelő irányba, megadott hosszúságú sugarat bocsát ki. Ennek lehetséges lekérdezni az első, más objektum ütköző komponensével történt metszéspontját. Visszaadja valamint azt is, hogy ezen metszéspontnál melyik másik objektummal találkozik. Az észlelt objektum meghatározására annak tag-jét használom fel. A következő szabályokat alkalmaztam az esetek lekezelésére:

- Ha egy jármű szenzora másik járművel találkozik, akkor a két jármű távolságának fordított arányában korlátozza a maximális sebességet.
- Ha egy jármű szenzora útkereszteződést észlel, és a jármű kanyarodni fog, vagy pedig kanyar következik, akkor maximális sebességét korlátozza az eredeti harmadára.
- Ha egy jármű szenzora olyan buszmegállót észlel, amiben éppen busz tartózkodik, álljon meg és engedje el a buszt.
- Ha egy jármű szenzora olyan útkereszteződést észlel, ahol az ő oldaláról éppen piros a jelzőlámpa, akkor álljon meg.
- Ha egy jármű szenzora választóvonalat észlel a jármű bal szélétől jobbra, és a jármű nincsen éppen útkereszteződésben, akkor kanyarodjon enyhén jobbra.
- Ha egy jármű szenzora többsávos út sávelválasztóját észleli a jármű bal szélétől jobbra, és a jármű nincs éppen útkereszteződésben, és a következő lehetőségnél jobbra fog fordulni, akkor enyhén kanyarodjon jobbra.

5.3.2. Szükséges objektumok

A szimulációban résztvevő járművek példányosításához létrehoztam néhány sablon objektumot, azaz prefab-et. Ezek a személygépjármű és az autóbusz objektumok. Az alap objektumokat a Unity Asset Store-ból szereztem be, ezek alapjában véve tartalmazták a mesht, egyesével elkülönítve a négy kereket. Az alap objektum felépítése a következő képpen néz ki: Van egy főobjektum, mely gyerekelemként tartalmazza a karosszéria mesh-ét, valamint egy Wheels objektumot. Ez a Wheels objektum rendelkezik négy gyerekelemmel, melyek rendszerint a négy keréknek felelnek meg. A busz és az autó felépítése ilyen szempontból megegyezik, a hozzáadott komponensek is megegyezőek lesznek. A kettő közötti különbséget a hozzárendelt szkript fogja majd adni.

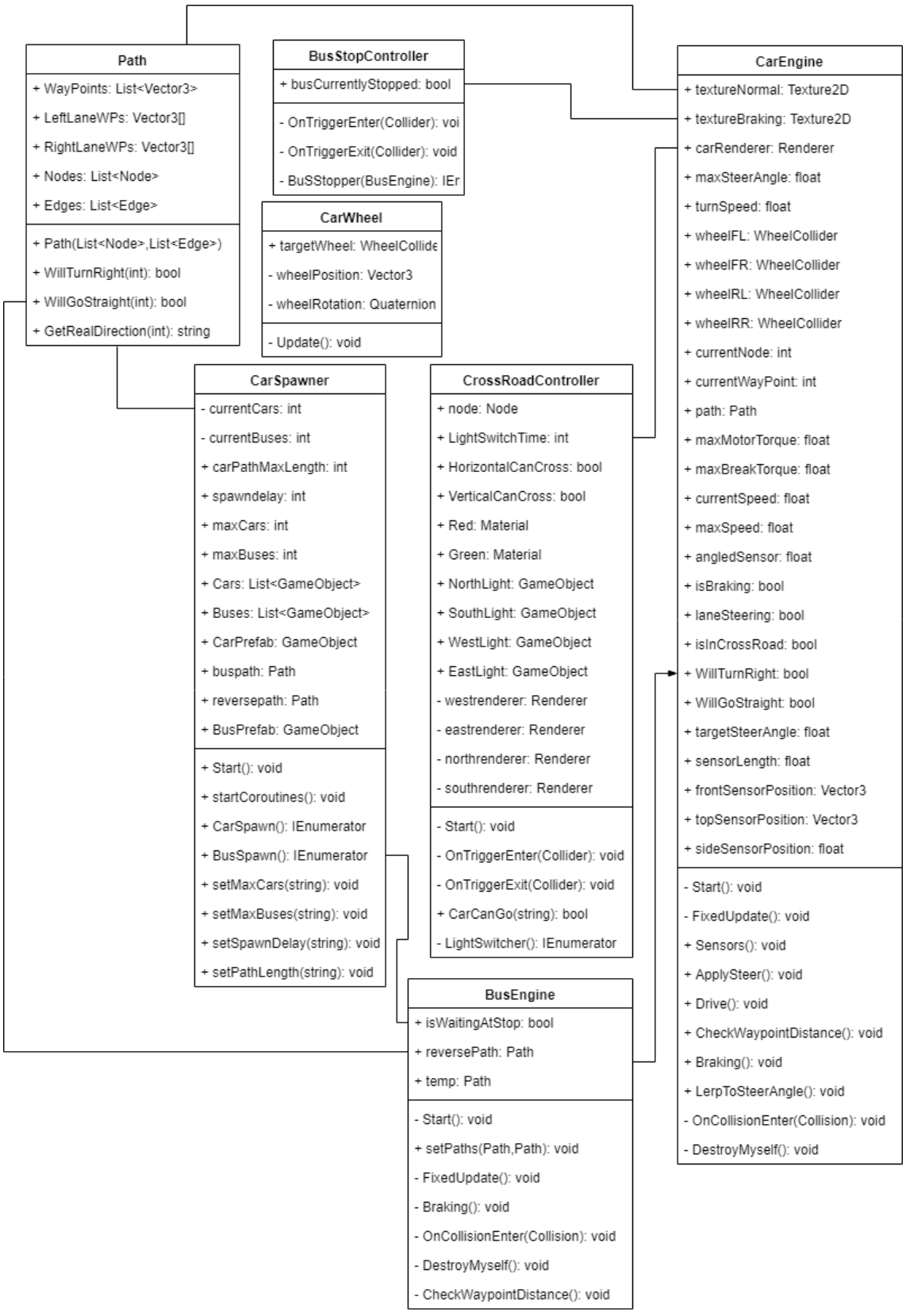
Komponensek hozzáadása

Kezdésként a főobjektumhoz hozzáadtam egy Rigidbody komponenst. Ezen komponens által lehet befolyásolni az objektum működését a fizikai motoron belül, szükséges az ütközések vizsgálatához is. Ezen Rigidbody komponensen belül beállítottam az autó tömegét 1200 egységre, a buszét pedig 1400 egységre.

Következő lépésben szükséges a járművek kerekeihez WheelCollider komponenseket rendelni. Ezen komponens számolja ki a fizikai motoron keresztül a kerekek fordulatszámát, csúszását, ennek a komponensnek a manipulálásával lehet gázt és féket, valamint a kormányzást szimulálni. A szkriptben szükség lesz erre a komponensre való hivatkozásra, ezért nem a meglévő kerekekhez adom hozzá, hanem létrehozok a kerék objektumokból egy másolatot. Ezekből a másolatokból törlök minden komponenst, és hozzáadok egy WheelCollidert. Ennek a sugarát beállítom a megfelelő mértékre mindkét járműnél, hogy lefedje a kerék meshét. Az így létrehozott objektum hivatkozható a szkriptben anélkül, hogy egy teljes GameObject-et kéne hozzárendelni, mivel ezek az objektumok csak egyetlen komponensből állnak a Transformon kívül.

A főobjektumba elhelyeztem rendszerint egy CarEngine és egy BusEngine szkriptet. A BusEngine a CarEngine leszármazottja, minimális módon fog eltérni az autó szkriptjétől. Végül a karosszériát reprezentáló objektumhoz hozzáadtam egy mesh collider komponenst, mely az ütközések vizsgálatára, a szenzorok működéséhez szükséges.

5.3.3. Szimulációs szkriptek



5.2. ábra. A szimulációval foglalkozó szkriptek UML osztálydiagramja

Path

Ez az osztály adja vissza a járműveknek az egyedi útvonalaikat. Adattagjai közé tartozik egy WayPoints nevű Vector3 lista, amely a célpontok koordinátáinak a listája, egy LeftLaneWPs és egy RightLaneWPs Vector3 tömb, amelyek a többsávos utak esetén tárolják a két a belső és külső sávra vonatkozó célpontokat. Tartalmaz továbbrá egy Node és egy Edge típusú listát, amely a gráf csomópontjait és éleit tartalmazza.

Két csomópont között a haladási irány kiszámítására írtam egy GetRealDirection függvényt. Ez a függvény paraméterként megkapja a következő csomópont indexét, ami hogyha megegyezik a jelenlegi él célpontjával, akkor visszaadja az él haladási irányát, ellenkező esetben annak ellentetjét adja vissza.

```
public string GetRealDirection(int idx)
        {
            if(Nodes[idx] == Edges[idx].To)
            {
                return Edges[idx].Direction;
            }
            else
            {
                switch (Edges[idx].Direction)
                {
                    case "north":
                        return "south";
                    case "south":
                        return "north";
                    case "west":
                        return "east";
                    case "east":
                        return "west";
                    default:
                        return null;
                }
            }
        }
```

Szintén írtam két függvényt annak meghatározására, hogy egy adott csomópont elérése után a jármű jobbra fog-e kanyarodni, illetve egyenesen megy-e tovább. Ezek a WillTurnRight és a WillGoStraight metódusok. Ezek működésükben a GetRealDirection metódust használják fel a jelenlegi és a következő csomópontra, majd a kettő kapott értéket vizsgálva meghatározzák hogy kanyarodik-e, illetve egyenes megy-e tovább.

```
public bool WillTurnRight(int idx)
        {
            if(idx < Nodes.Count - 2)
            {
                string currentlyGoing = GetRealDirection(idx);
                string willGo = GetRealDirection(idx + 1);
                if ((currentlyGoing == "north" && willGo == "east") ||
                (currentlyGoing == "east" && willGo == "south") ||
```

```
                (currentlyGoing == "south" && willGo == "west") ||
            (currentlyGoing == "west" && willGo == "north"))
                {
                    return true;
                }
            }

            return false;
        }
```

A másik metódus működésében megegyezik a fentivel, azzal különbséggel hogy a két érték egyenlőségének esetén ad igaz értéket, ellenkező esetben hamisat.

Az útvonalat a konstruktor adja vissza, mely megkapja az érintett csomópontok és élek listáját. Először is megnézi hogy a kapott út tartalmaz-e éleket, vagy pedig csak egy pontból áll. Ha valós útvonal, akkor a kijelöli a kezdőpontot az első él irányának függvényében. Példaként ha dél felé tart az él, akkor a kezdőpont az első csomópont középpontja és bal felső csúcsa közötti felezőpont. Ha az él iránya dél felé tart, de kiindulási pontja a következő csomópont, akkor a kezdőpont a csomópont középpontja és jobb alsó csúcsa közötti felezőpont lesz.

```
case "south":
    if (nodes[1] == edges[0].From)
    {
        WayPoints[0] = (nodes[0].WorldCornerBR +
        WayPoints[0]) / 2;
    }
    else
    {
        WayPoints[0] = (nodes[0].WorldCornerTL +
         WayPoints[0]) / 2;
    }
    break;
```

Ezek után végigiterál az útvonal további pontjain, ezzel megegyező módszerrel kiszámolja a célpontokat. Ha egy adott élen a sávok száma 2, akkor kiszámolja a RightLaneWPs és LeftLaneWPs adott indexű elemeit, majd a haladási iránynak megfelelően cseréli az adott célpontot a következő módon:

```
if (edges[i - 1].Lanes == 2)
{
    LeftLaneWPs[i] = (point +
    WayPoints[WayPoints.Count - 1]) / 2;
    RightLaneWPs[i] = (nodes[i].
    WorldCornerBR + WayPoints[WayPoints.Count - 1]) / 2;
}
if (WillTurnRight(i-1))
{
    if(edges[i - 1].Lanes == 2)
    {
        WayPoints[WayPoints.Count - 1] = RightLaneWPs[i];
```

```
    }
    point = (nodes[i].WorldCornerBR + nodes[i].
    WorldCornerTR) / 2;
    WayPoints.Add((nodes[i].WorldCornerBR + point) / 2);
}
else if(!WillGoStraight(i-1))
{
    if (edges[i - 1].Lanes == 2)
    {
    WayPoints[WayPoints.Count - 1] = LeftLaneWPs[i];
    }
    point = (nodes[i].WorldCornerBL + nodes[i].
    WorldCornerTL) / 2;
    WayPoints.Add((nodes[i].WorldCornerTL + point) / 2);
}
else if(edges[i - 1].Lanes == 2)
{
    WayPoints[WayPoints.Count - 1] = LeftLaneWPs[i];
}
```

A konstruktor lefutása után az osztály WayPoints adattagja már tartalmazza az útvonal bejárásához szükséges összes pontot.

CarSpawner

Ez a szkript felelős a buszok és az autók létrehozásáért. Tartalmazza a következő publikus adattagokat, amelyeket a felhasználói felületről futás közben lehet módosítani:

- carPathMaxLength: Egy autó útjának maximális hossza, csomópontokban mérve.
- spawndelay: Két autó létrehozása közötti időtartam.
- maxCars: Az egyszerre maximálisan létezhető autók száma.
- maxBuses: Az egyszerre maximálisan létezhető buszok száma.

Szüksége van magára a busz és autó objektumokra, ezek GameObject típusúak és az editorból rendelem hozzá őket. Az éppen létező autókat és buszokat külön GameObject típusú listában tartja nyilván. Működése kettő korutinon alapszik, az egyik az autókat, a másik a buszokat hozza létre.

```
IEnumerator CarSpawn()
{
    while (true)
    {
        yield return new WaitForSeconds(spawndelay);
        if(currentCars < maxCars)
        {
            Cars.Add(Instantiate(CarPrefab));
            Debug.Log("car created");
            Cars.Last().GetComponent<CarEngine>().path =
```

```
            gameObject.GetComponent<RoadGenerator>().graph.
            GenerateRandomPath(carPathMaxLength);
            currentCars++;
        }
        for (int i = 0; i < Cars.Count; i++)
        {
            if (Cars[i] == null)
            {
                Cars.Remove(Cars[i]);
                currentCars--;
            }
        }
    }
}
```

A korutinokat IEnumerator típusú függvényekként kell létrehozni. Ezen korutinok indulásuk után külön szálon futnak. Sajátosságok még az, hogy lehet őket várakoztatni, amelyre a „yield return new WaitForSeconds()” szintaktika ad lehetőséget. A fenti korutin az autók létrehozásáért felelős. Egy végtelen ciklust futtat, melynek minden iterációja elején vár a spawndelay-ben megadott ideig. Ezután, ha kevesebb autó van eddig mint a megengedett, akkor hozzáad egy újat a listához, majd a létrejött autó CarEngine szkript komponensében beállítja az útvonalat, melyet RoadGenerator komponens gráfja fog generálni, hossza pedig függ a megadott változótól.

Mivel az autók céljuk elérésekor törlik magukat, a listából is el kell távolítani. Ez egy egyszerű null check, végigiterál a Cars listán és törli a null referenciájú elemeket. A korutinok elindítása egy függvényben történik, mely a komponens betöltődésekor a Start-ban hívódik:

```
public void startCoroutines()
        {
            StartCoroutine(CarSpawn());
            StartCoroutine(BusSpawn());
        }
```

A buszokért felelős korutin nagymértékben megegyezik az autókéval, annyi különbséggel hogy fix 10 másodperces késleltetés van, valamint azonos busz útvonalat kap mindegyik. Ezt a szkriptet a Manager objektum komponenseként adom hozzá a scenehez.

CarEngine

A CarEngine szkript felel az autók összes mozgásáért. Az editorban inicializált adattagjai a következőek:

- TextureNormal: Alap állapotban a jármű textúrája.
- TextureBraking: Fékezés közben a jármű textúrája.
- Négy WheelCollider objektum.
- carRenderer: Az autó meshrenderer komponense.

Az osztályon belül nyilván van tartva annak maximális kormányzási íve, maximális kanyarodási sebessége, a bejárandó út, az út hanyadik csomópontjánál jár, az úton hanyadik kijelölt célpontnál tart, a motor maximális forgatónyomatéka, a fék maximális ereje, a jármű jelenlegi sebessége, a jármű maximális sebessége, az hogy a jármű jelenleg fékez-e, sávot vált-e, útkereszteződésben van-e, jobbra fog-e kanyarodni, egyenesen fog-e menni, valamint a jelenlegi kormányzás íve. Definiálva van továbbá a szenzorok pozíciója, ezek a jármű elején vannak bal oldalt, középen és jobb oldalt, valamint a jármű fölött szintén ugyan ilyen elrendezésben, hozzátéve még egyet, amelyik a járműtől jobbra néz.

A szkript start metódusa a következő:

```
private void Start()
    {
        transform.position = path.WayPoints[0];
        if(path.WayPoints.Count > 1)
        {
            transform?.LookAt(path.WayPoints[1]);
        }
    }
```

Beállítja a jármű pozícióját az első célpontra, és ha az út nem csak egyetlen csomópontból áll (azaz tényleges útvonal), elforgatja a járműt a következő célpontja felé. Az objektum animálására az Update metódus helyett a FixedUpdate-et használom. Ez azért előnyös, mert a fizikai számítások amelyek a képkockaszámtól függenek, a sima Update metódusban helytelenül működnének. A FixedUpdate ezzel a fizikai motor frekvenciáját használja. A metódus az alábbiakat tartalmazza:

```
private void FixedUpdate()
    {
        Sensors();
        ApplySteer();
        Drive();
        CheckWaypointDistance();
        Braking();
        LerpToSteerAngle();
    }
```

Ebből az első, a Sensors metódus a szenzorokkal kapcsolatos számításokat végzi el. Megnézi ütközött-e valamelyik valamilyen objektummal, ha igen akkor elvégzi a hozzá tartozó műveleteket. Egy példa a sávelválasztó észlelésére:

```
if (Physics.Raycast(sensorStartPos, Quaternion.AngleAxis(-angledSensor,
         transform.up) * transform.forward, out hit, 3))
        {
            if (hit.collider.CompareTag("MultiLaneDivider") &&
            !WillTurnRight)
            {
                if (!isInCrossRoad)
                {
                    Debug.DrawLine(sensorStartPos, hit.point);
```

```
                    targetSteerAngle = -5f;
                    laneSteering = true;
                }

            }
        }
```

A metódus ezen része az autó bal oldala felé vetít sugarat, melynek hossza 3 egység. A sensorStartPos egy előre beállított vektor a szenzorok számára, a középső első szenzor helyét jelenti, a többi ehhez képest relatívan helyezkedik el. A második paraméter az y tengely körüli forgatást jelenti -90 fokkal (az angledSensor értéke). Ha a sugár ütközött, akkor az első feltétel teljesül, eltárolódik a hit változóba az érintett ütköző. A következő lépésben az érintett objektum tag-jét vizsgálom. Ha megegyezik a sávelválasztó tag-jével, valamint az autó nem akar jobbra kanyarodni, és nincs is útkereszteződésben, akkor balra 5 fokban kezd el kormányozni, és a sávváltást jelző változó igazra vált. Ez a változó a Sensors metódus elején mindig hamis értéket kap.

A következő metódus, az ApplySteer az amelyik kiszámolja hogy az autónak milyen irányba és mennyit kell kormányoznia. Ha az autó éppen sávot vált és nincs útkereszteződésben, a metódus nem csinál semmit. Ha ezek nem teljesülnek, megkapja a következő célponthoz tartó relatív vektort, amelynek hossza alapján kiszámolja a kormányzás kívánt szögét a következő képpen:

```
Vector3 relativeVector = transform.InverseTransformPoint(path.
WayPoints[currentWayPoint]);
float newSteer = (relativeVector.x / relativeVector.magnitude)*
maxSteerAngle;
targetSteerAngle = newSteer;
```

A következő metódus a Drive, amely a jármű kerekeinek forgatásáért, valamint a fékezésért felelős. Első lépésben kiszámolja a jelenlegi sebességet a kerék kerületének és forgásának függvényében. Ezután ha gyorsabban megy, mint a maximális sebesség akkor fékez, ha lassabban akkor gázt ad, egyébként pedig nem ad gázt, és nem fékez.

```
public void Drive()
    {
        currentSpeed = 2 * Mathf.PI * wheelFL.radius *
        wheelFL.rpm * 60 / 1000;
        if(currentSpeed < maxSpeed)
        {
            isBraking = false;
            wheelFL.motorTorque = maxMotorTorque;
            wheelFR.motorTorque = maxMotorTorque;
        }
        else if(currentSpeed > maxSpeed)
        {
            wheelFL.motorTorque = 0;
            wheelFR.motorTorque = 0;
            isBraking = true;
        }
        else
```

```
        {
            isBraking = false;
            wheelFL.motorTorque = 0;
            wheelFR.motorTorque = 0;
        }

    }
```

A következő metódus megnézi a jármű távolságát a következő célpontjától. Ezt a beépített Vector3.Distance függvénnyel teszi. Ha 2.5 egységnyi távolságon belülre ért, akkor megnézi ez-e az utolsó célpont, ha igen akkor törli saját magát. Ellenkező esetben, ha nem az utolsó előtti csomópontnál jár, akkor kiszámítja hogy a következő csomópontnál milyen irányba fog tovább menni. Ezt a Path osztály függvényeivel teszi. Ezek után inkrementálja a currentWayPoint változót, mely a soron következő célpont indexét jelöli. Ha nincs útkereszteződésben a célpont elérésekor, akkor a csomópontok indexét is növeli.

Fontos elkülöníteni itt a csomópont és a célpont fogalmát. Ahogyan az a modellben is definiálva lett, a csomópont az úthálózat szakaszainak összekötő részét jelöli, ami akár lehet útkereszteződés. A célpontok viszont a járművek részére kijelölt útvonal fix pontjai, amelyek egyértelműen kijelölik hogy az út melyik részén kell haladniuk. Ebből adódik, hogy egy útkereszteződésben bár csak egy csomópont van, két célponton halad át benne a jármű. Az elsőt még az útkereszteződésbe való belépés előtt éri el, ilyenkor növekszik a csomópont indexe. A második célpontot, amely azt jelöli hol kell kiérnie az útkereszteződésből, akkor éri el amikor még benne van. Ezért a második esetben nem kell inkrementálni a csomópontok indexét.

Amennyiben a jármű még nem érte el a következő pontot, megnézi a metódus, hogy a jármű 40 egységnyi távolságon belül van-e a következő célponthoz. Ha igen, és nem egyenesen akar továbbhaladni, és jelenleg nem fékez, akkor a maximális sebességet 3 egységre állítja. Ellenkező esetben, ha a jármű nem fékez, a maximális sebességet 8 egységre állítja.

```
private void CheckWaypointDistance()
    {
        if (Vector3.Distance(transform.position,
        path.WayPoints[currentWayPoint]) < 2.5f)
        {
            WillTurnRight = false;
            if(currentNode == path.Nodes.Count - 1)
            {
                Destroy(gameObject);
                return;
            }
            if(currentNode < path.Nodes.Count - 2)
            {
                WillTurnRight = path.WillTurnRight(currentNode);
                WillGoStraight = path.WillGoStraight(currentNode);
            }
            currentWayPoint++;
            if (!isInCrossRoad)
```

```
            {
                currentNode++;
            }
        }else if(Mathf.Abs(Vector3.Distance(transform.position,
        path.WayPoints[currentWayPoint])) < 40.0f && !WillGoStraight &&
        !isBraking)
        {
            maxSpeed = 3f;
        }
        else if (!isBraking)
        {
            maxSpeed = 8f;
        }
    }
```

A soron következő Braking metódus felel a jármű fékezéséért. Ha a jármű éppen fékezni akar, kicseréli a textúrát arra, ahol világítanak a féklámpák, majd beállítja minden kerék fékerejét a megadott maximális értékre. Ha éppen nem fékez, visszaállítja a textúrát, és leveszi a fékerőt a kerekekről.

```
private void Braking()
    {
        if (isBraking)
        {
            carRenderer.material.mainTexture = textureBraking;
            wheelRL.brakeTorque = maxBreakTorque;
            wheelRR.brakeTorque = maxBreakTorque;
            wheelFL.brakeTorque = maxBreakTorque;
            wheelFR.brakeTorque = maxBreakTorque;
        }
        else
        {
            carRenderer.material.mainTexture = textureNormal;
            wheelFL.brakeTorque = 0;
            wheelFR.brakeTorque = 0;
            wheelRL.brakeTorque = 0;
            wheelRR.brakeTorque = 0;
        }
    }
```

Az utolsó metódus a LerpToSteerAngle. Ennek a feladata az, hogy a kerekek ne hirtelen mozduljanak el célirányba kormányzás közben, hanem fokozatosan, a turnSpeed változó által megadott gyorsasággal forduljanak. Lényegében nem más, mint egy lineáris interpoláció a jelenlegi kerékív és a kívánt kerékív között, turnSpeed függvényében.

```
public void LerpToSteerAngle()
    {
        wheelFL.steerAngle = Mathf.Lerp(wheelFL.steerAngle,
        targetSteerAngle, Time.deltaTime * turnSpeed);
```

```
        wheelFR.steerAngle = Mathf.Lerp(wheelFR.steerAngle,
        targetSteerAngle, Time.deltaTime * turnSpeed);
    }
```

Az osztályban van még egy OnCollisionEnter metódus is, ez az ütközésekkor keletkező események alapértelmezett lekezelője, ezt kell felüldefiniálni a működés megadásához. Jelen esetben, ha egy Car tag-el ellátott objektummal ütközött, akkor 5 másodperc elteltével törli magát az autó.

BusEngine

Ez a szkript a CarEngine leszármazottja. Új adattagként szerepel benne az isWaitingAtStop, mely azt jelzi éppen megállóban áll-e,valamint egy reversePath, amely a busz útvonalának ellentétes irányban lévő változatát tárolja. Lényegesebb változtatás csak a CheckWaypointDistance-ben történt. Itt ha elérte útvonalának az utolsó pontját, megkapja új útvonalként az ellentétes irányú utat, és az indexek visszaállnak kezdeti értékükre.

```
if (currentNode == path.Nodes.Count)
            {
                temp = path;
                path = reversePath;
                reversePath = temp;
                currentNode = 1;
                currentWayPoint = 1;
                transform.position = path.WayPoints[0];
                transform.LookAt(path.WayPoints[1]);
                return;
            }
```

CarWheel

Ennek a szkriptnek a feladata a mesheken lévő kerekek forgatása, a WheelCollidereknek megfelelően. Editorban hozzárendeltem komponensként a járművek kerekeihez, adattagként megadtam nekik az adott kerék WheelCollider-jének a referenciáját. Az Update metódusban beállítja a kerekek pozícióját és forgatását a collidereknek megfelelően.

```
    void Update()
    {
        targetWheel.GetWorldPose(out wheelPosition,out wheelRotation);
        transform.position = wheelPosition;
        transform.rotation = wheelRotation;
    }
```

BusStopController

A buszmegálló objektumokhoz komponensként hozzárendelt szkript. Eltárolja hogy jelenleg áll-e területén busz. Ha a buszmegálló objektum triggerjébe belelép egy másik

objektum, egy eseményt lő el, amelyet a szkript OnTriggerEnter metódusa kezel le. Ebben a szkriptben megnézi, hogy a belépő objektum Bus tag-el van-e ellátva, ha igen akkor eltárolja ennek tényét, valamint a busz BusEngine komponensében beállítja as isWaitingAtStop értékét igazra. Ezek után indít egy korutint, mely 5 másodperc elteltével továbbengedi a buszt.

```
void OnTriggerEnter(Collider other)
    {

        if (other.CompareTag("Bus"))
        {
            busCurrentlyStopped = true;
            other.gameObject.GetComponentInParent<BusEngine>().
            isWaitingAtStop = true;
            StartCoroutine(BuSStopper(other.gameObject.
            GetComponentInParent<BusEngine>()));
            Debug.Log("Stopped a bus");
        }
    }
```

Az OnTriggerExit event kezelő csak egyszerűen beállítja a busCurrentlyStopped értékét hamisra.

CrossRoadController

Ez a szkript az útkereszteződés objektumoknak egyik komponense. Az editorban megadtam neki az útkereszteződés körüli lámpa objektumok referenciáit, materialokat a zöld és piros színű lámpákhoz. Tartalmaz továbbá egy adattagot, mely megadja hány másodpercenként váltanak át a lámpák. A Start metódusban megnézi hány felé ágazik el a csomópont. Ha több mint kétfelé, akkor eltárolja a lámpáinak Renderer komponenseit, és elindítja a lámpák váltásáért felelős LightSwitcher korutint. Ellenkező esetben törli a lámpa objektumokat, és átengedi minden irányból az összes járművet. A LightSwitcher metódus az alábbi módon néz ki:

```
IEnumerator LightSwitcher()
    {
        while (true)
        {
            westrenderer.material = Green;
            eastrenderer.material = Green;
            northrenderer.material = Red;
            southrenderer.material = Red;
            HorizontalCanCross = true;
            yield return new WaitForSeconds(LightSwitchTime);
            westrenderer.material = Red;
            eastrenderer.material = Red;
            HorizontalCanCross = false;
            yield return new WaitForSeconds(5);
            northrenderer.material = Green;
            southrenderer.material = Green;
```

```
            VerticalCanCross = true;
            yield return new WaitForSeconds(LightSwitchTime);
            northrenderer.material = Red;
            southrenderer.material = Red;
            VerticalCanCross = false;
            yield return new WaitForSeconds(5);
        }
    }
```

Váltásoknál 5 másodpercig vár, addig mindkét irányba pirosat jelez, majd a LightSwitchTime-nak megfelelő ideig az egyik irányt átengedi, a másikat nem. A megszerzett renderer komponenseken keresztül ezt a materialok kicserélésével is szemlélteti.

Azt, hogy egy adott autó átmehet-e, a CarCanGo metódus adja vissza. Ezt a metódust a CarEngine hívja meg, amikor egyik szenzora útkereszteződést észlel. A függvény megkapja milyen irányból jön az autó, és ennek függvényében visszaad egy igaz vagy hamis értéket.

```
public bool CarCanGo(string direction)
    {
        if((HorizontalCanCross && (direction == "west"
        || direction == "east")) || (VerticalCanCross &&
        (direction == "north" || direction == "south")))
        {
            return true;
        }
        return false;
    }
```

A szkript tartalmaz továbbá két esemény lekezelőt, OnTriggerEnter és OnTriggerExit néven. Ezek arra szolgálnak, hogy a járművek szenzorainak hosszát lecsökkentsék amíg útkereszteződésen belül vannak, hogy ne zavarjon be közlekedésükbe útkereszteződésen kívüli objektum. Amint kiértek ez visszaáll az eredeti értékre.

6. fejezet

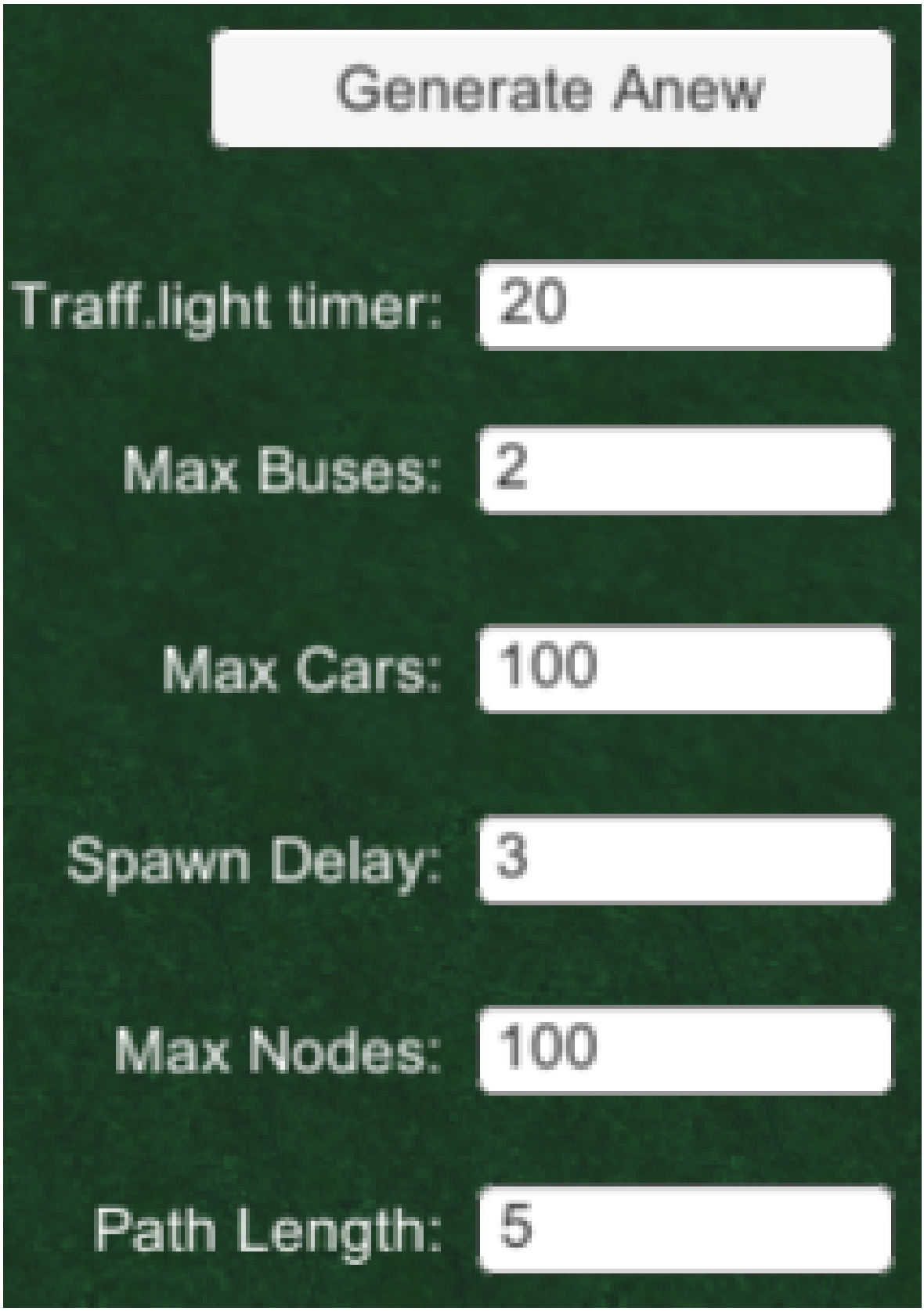
Szimulációk

6.1. A szimuláció alakulása paraméterezés függvényében

A paraméterezést az alkalmazáson belül megjelenő felületről lehet állítani. A következő értékek változtathatóak:

- A közlekedési lámpák váltakozásainak ideje, másodpercben megadva
- A szimulációban egyidőben maximálisan résztvevő autóbuszok száma
- A szimulációban egyidőben maximálisan résztvevő személygépjárművek száma
- Két személygépjármű létrehozása közötti eltelt idő, másodpercben mérve
- A város úthálózadának csomópontszáma
- Egy személygépjármű útjának maximális hossza, csomópontokban mérve

Ezen értékek közül a gráf csomópontszámán kívül mindegyik változtatható a város újra generálása nélkül. Az újra generálást egy Generate Anew gombbal lehet elvégezni a felületen.



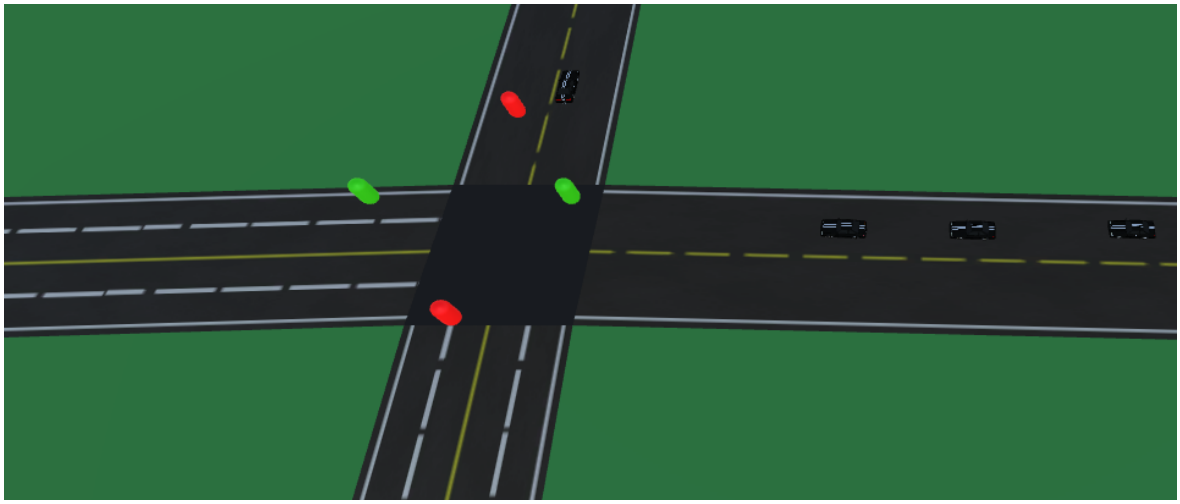
6.1. ábra. A paraméterek beállítási felülete, azok alapértelmezett értékeivel

Az alapértelmezett paraméterekkel csekély forgalom jön létre egy kis méretű vá-

rosban. Maga a város képe nagyjából tükrözi egy ilyen méretű város lehetséges elrendezését, az épületek típusai a központ felé bérházak, a peremeknél kertesházak. A szimuláció futása közben a kamerát a W,A,S,D gombokkal lehet mozgatni, valamint az egér görgőjével közelíteni és távolítani.

6.1.1. Forgalmi szituációk

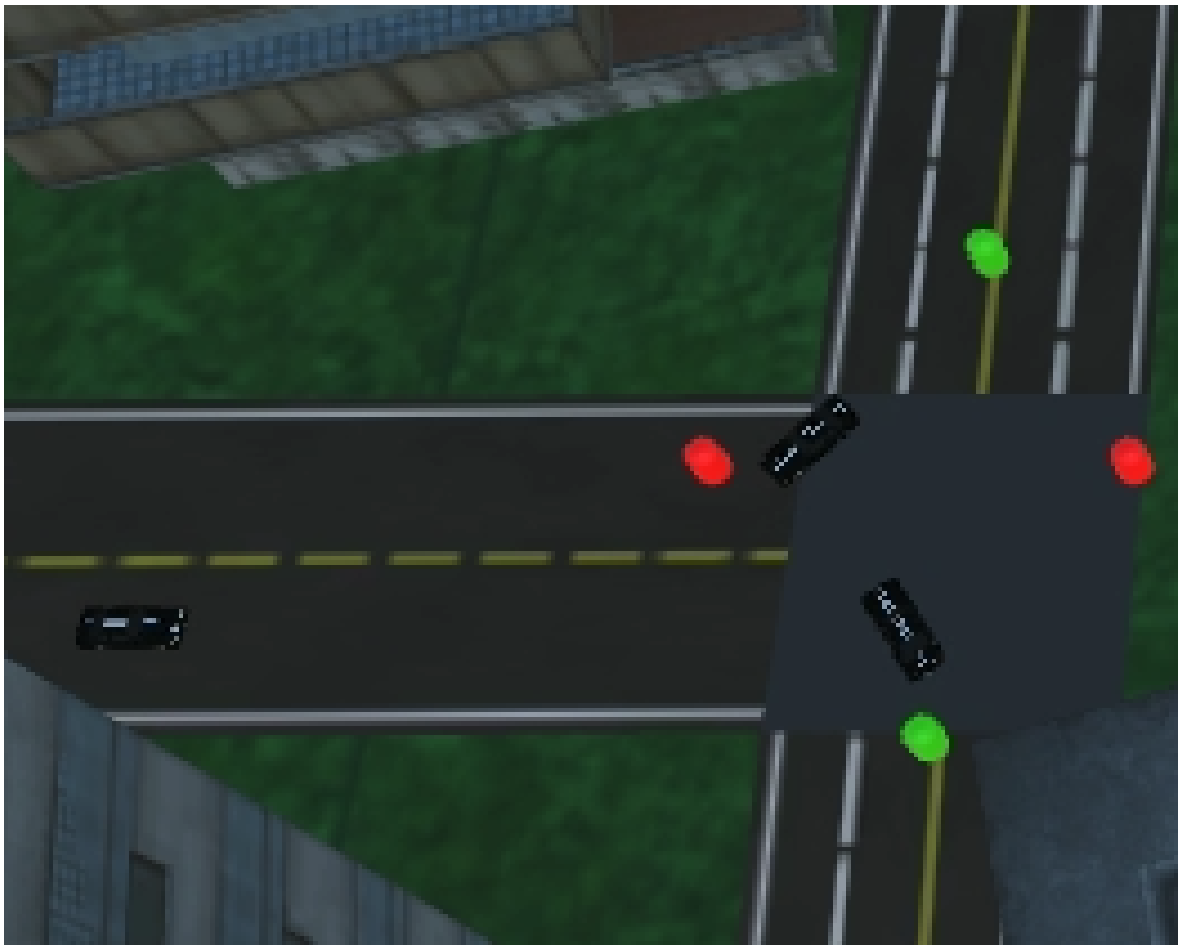
A közlekedési szabályok is megfigyelhetőek az adott objektumoknál, az alábbi képeken ezt szemléltetem.



6.2. ábra. Útkereszteződésnél várakozó autók éppen elindulnak

6.3. ábra. Többsávos úton a felülről párhuzamosan érkező autók balra és jobbra kanyarodnak, velük egyidőben szemben jövő autó pedig balra

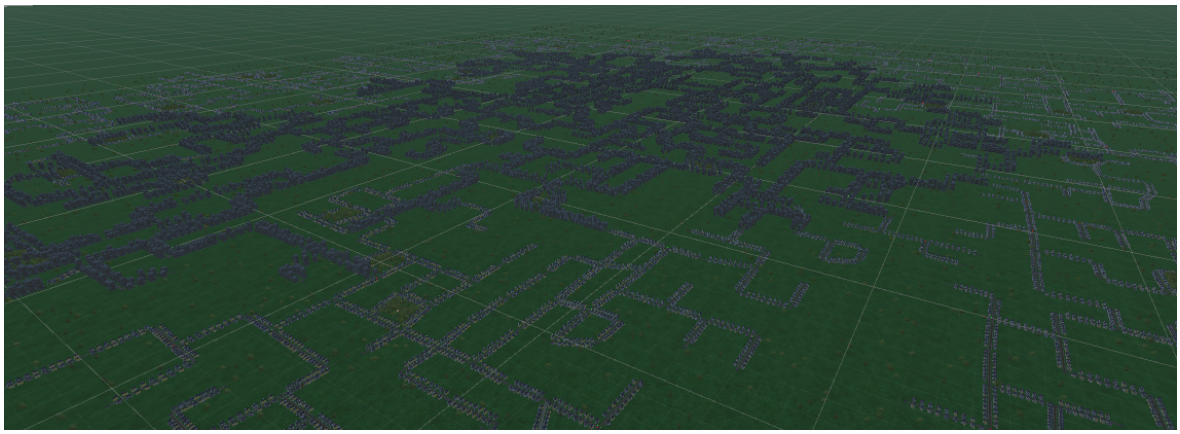
6.4. ábra. Autó az előtte lévő autóbusz elindulását várja



6.5. ábra. Balra nagy ívben kanyarodó autó elengedi maga előtt a jobbra kis ívben kanyarodót

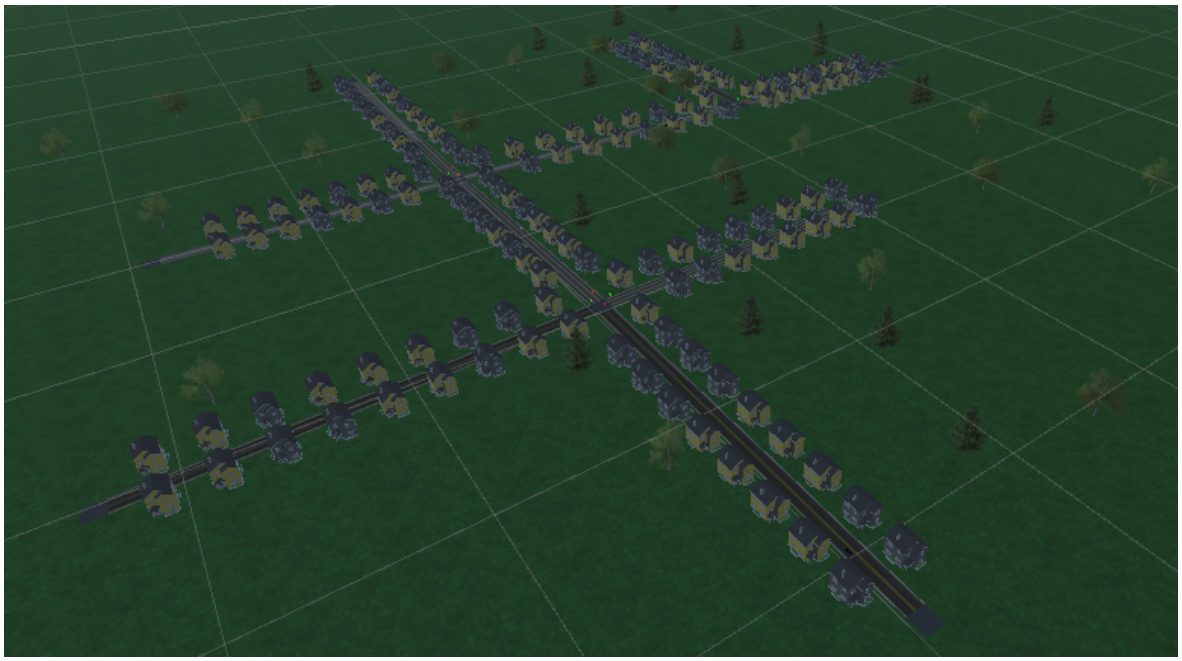
6.1.2. Úthálózat mérete

A generálás méretét tekintve nagyobb városoknál is elvártan működik. A legnagyobb méret amivel teszteltem, az egy 2000 csomópontból álló város. Ezt körülbelül 3 másodperc alatt generálja le egy átlagos számítógépen.



6.6. ábra. Egy kétezer csomópontból álló város képe

Kipróbáltam valamint egy kisebb, 10 csomópontból álló községet is, megjelenésében és funkcionalitásában ez is megfelelő.



6.7. ábra. Tíz csomópontból álló község

6.1.3. A forgalom sűrűsége az úthálózat méretének függvényében

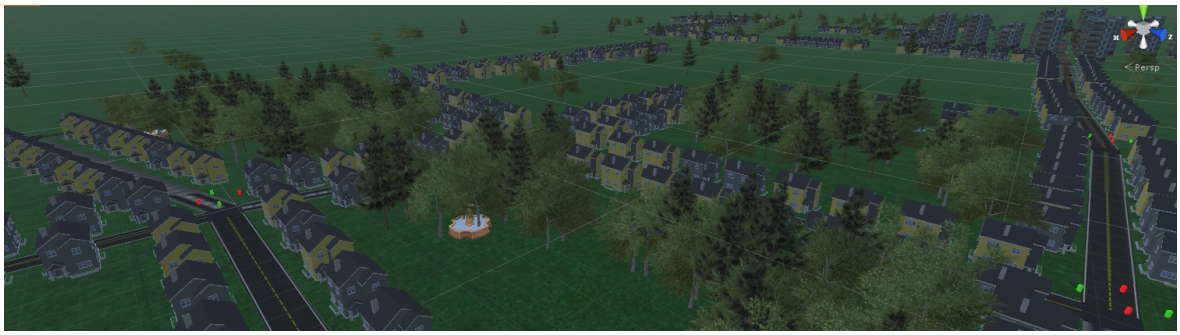
A minimális egy másodperces késleltetés miatt az autók létrehozását illetően könnyebbnek találtam a forgalom vizsgálatát egy kisebb, nagyjából 20 csomópontból álló úthálózaton.

Itt észrevehető hogy átlagos forgalom akkor kezd el kialakulni, ha a ötször annyi mennyiségű jármű van az utakon, mint amennyi csomópontból áll az úthálózat. Ez a

paraméterezés még nem okoz forgalmi dugót. A jelzőlámpák váltási idejét felnövelve 40 másodpercre minimális lassulás észlelhető, de ezt ellensúlyozza az, hogy kevesebbszer kell megállnia egy autónak, ha sokáig egyenesen halad.

6.8. ábra. Nemrég váltott jelzőlámpa előtt várakozik három autó (20 csomópont 100 autóval)

A parkok is viszonylag rendesen generálódnak, főleg a nagyobb méretű városokban figyelhető ez meg.



6.9. ábra. Parkok egymás mellett egy város szélén

7. fejezet

Összegzés

A feladat teljesítését természetesen az első részben leírt szoftverek vizsgálatával kezdtem, melyből megtudtam pár információt az ilyen jellegű szoftverek működéséről. Az ott leírtak összefoglalják, hogy egy átlagos ilyen célú szoftver milyen felépítéssel bír, általánosan milyen funkcióik vannak, valamint az általuk kezelt objektumok milyen tulajdonságokból épülnek fel.

A modell megalkotásánál definiált kritériumokat teljesítettem, az elkészült programban megjelennek az ott leírt elemek, forgalmi szituációk, azok működése a leírtak szerint történik.

A generálás folyamatát a harmadik részben dokumentáltam. Az algoritmus eredményéből látszik, hogy alkalmas a forgalmi helyzetek megjelenítésére, az átlagos közlekedésre. Városkép szempontjából elfogadható eredményt ad, az épületek és más díszítőelemek logikusan helyezkednek el a város területén.

A szimuláció tervezésével foglalkozó fejezetben részletesen dokumentáltam az egyes szkriptek működését, valamint a Unity Engine-en belüli implementációt. Az egyes algoritmusokat kódpéldákkal is szemléltettem. Szimuláció közben látható ennek az implementációnak az eredménye, a járművek mozgása a Unity fizikai motorján keresztül történik.

Az utolsó részben megvizsgáltam a paraméterezéssel kapcsolatban felállított követelmények teljesülését. A program a megalkotott felületen keresztül működésében befolyásolható, a különböző paraméterezések eredménye pedig szemmel látható.

Összefoglalva, a program feladatként kiírt célokat, bár közlekedésre vonatkozóan nem mélyül bele a rengeteg úthálózati elembe, való életben létező szabályokba, hanem ehelyett csak az egyszerű elemeket kezeli. Viszont a megalkotott szoftver nagyon könnyen bővíthető, a modellre építve egyszerűen lehet hozzáadni elemfajtákat ha az implementációt tekintjük.

Irodalomjegyzék

[1] https://www.researchgate.net/publication/228966705_A_Review_of_Traffic_Simulation_Software

[2] https://www.anylogic.com/road-traffic/

[3] https://www.dlr.de/ts/en/desktopdefault.aspx/tabid-9883/16931_read-41000/

[4] https://www.researchgate.net/publication/316700661_GIS_three-dimensional_Modelling_with_geo-informatics_techniques

[5] https://cloud.anylogic.com/model/6322b634-2cc5-4d40-90cf-3bb44c825ea7?mode=SETTINGS&tab=GENERAL

[6] https://sumo.dlr.de/wiki/Tutorials/TraCIPedCrossing

[7] https://developers.google.com/maps/documentation/

CD-melléklet tartalma

A dolgozat PDF változatát a `dolgozat.pdf` fájlban találjuk.

A dolgozat LaTeX segítségével készült. A forrásfájlok a `dolgozat` jegyzékben találhatók.

...