TANSZÉKVEZETŐ

SZAKDOLGOZAT FELADAT

György Márk Attila
Mérnökinformatikus hallgató részére

Procedurális város generálás és interaktív gyalogos szimuláció Unity engine alkalmazásával

Annak érdekében, hogy a gyors fejlesztést elősegítsék, olyan játékfejlesztő keretrendszerek jelentek meg, amelyek lehetővé teszik a fejlesztők számára, hogy a látványvilág és a programlogika tervezésére koncentráljanak, miközben a grafikus programozási feladatokat és a multiplatform támogatást a rendszer kezeli. Az ilyen keretrendszerek között megtalálható például a Unity is.

A hallgató feladata a Unity játékmotor használatával egy előre meghatározott paraméterek alapján procedurális városgeneráló algoritmus készítése, abban a városban NPC-k generálása, akik véletlenszerűen mozognak, amikkel a játékos tud kommunikálni írásban, úgy, mintha ChatGPT-vel beszélgetne. Ezen kívül a városban járművek is legyenek, amik szintén véletlenszerűen mozognak. A városban a játékos szabadon mozoghat, és újragenerálhatja azt.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be a Unity játékmotort.
- Készítsen procedurális városgeneráló algoritmus.
- Készítsen nem játékos karaktereket (NPC-ket), egyéni személyiségekkel.
- Tesztelje le az elkészült alkalmazást és értékelje a kapott eredményeket.

Tanszéki konzulens: Kárpáti Attila Ádám

Budapest, 2023. szeptember 30.

Dr. Kiss Bálint
egyetemi docens
tanszékvezető

Budapest Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

H-1117 Budapest, Magyar tudósok krt 2. I. ép. III.em.
Telefon: 463-2870. Fax: 463-2871
www.iit.bme.hu

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
IIT Tanszék

György Márk Attila (Li Jiaxiang)

UNITY VÁROS GENERÁLÁS

KONZULENS

Kárpáti Attila Ádám

BUDAPEST, 2023

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2023. 11. 21.

...…….
György Márk Attila

Összefoglaló

A mai játékfejlesztésben egyre inkább előtérbe kerülnek a játékmotorok, amelyek lehetővé teszik a fejlesztők számára, hogy kész játékfejlesztési keretrendszerekkel dolgozzanak. Ezek a keretrendszerek előre definiált elemeket tartalmaznak, amelyeket a fejlesztőknek nem kell teljesen nulláról megvalósítaniuk, és ennek köszönhetően gyorsabban fejleszthetnek játékokat.

Szakdolgozatom során az olvasó megismerkedhet a Unity játékmotor használatával és komponenseivel. Továbbá azzal, hogy hogyan lehet ennek a játékmotornak a használatával olyan játékot készíteni, amiben a felhasználó által megadott paraméterek alapján egy kockákból felépülő város generálódik.

A város procedurálisan generálódik, a bemeneti paraméterek csupán pár alapvető tulajdonságát határozzák meg a keletkező városnak, ezért ugyanazokkal a kezdeti paraméterekkel különböző városok is létre tudnak jönni. A keletkező városban gyalogosok és autók közlekednek véletlenszerűen. A játékos ebben a városban szabadon mozoghat, beszélgethet a gyalogosokkal. A felhasználó bármikor újra generálhatja a várost ugyanazon paraméterekkel, vagy akár különbözőkkel is.

Abstract

Today's game development is increasingly dominated by game engines, which allow developers to work with ready-made game development frameworks. These frameworks contain predefined elements that developers don't have to implement entirely from scratch, enabling them to develop games more quickly.

In my thesis, readers can learn about the utilization and components of the Unity game engine. Furthermore, it explains how to use this game engine to create a game in which a city, constructed from cubes, is generated based on user-specified parameters.

The city is generated procedurally, and the input parameters only determine a few fundamental properties of the generated city. Therefore, different cities can be created with the same initial parameters. Pedestrians and cars move randomly within the resulting city. Players can freely navigate the city and interact with pedestrians. Users have the ability to regenerate the city at any time using the same parameters or even different ones.

1 Bevezetés

1.1 Feladat bemutatása

A szakdolgozatom végső célja egy olyan játék elkészítése, amiben a felhasználó különböző városokban tud szabadon mozogni, nézelődni, beszélgetni a gyalogosokkal. Ezen a projekten már dolgoztam / dolgoztunk a témalabor és az önálló labor tárgyak keretein belül.

A témalabor során készítettem el a várost generáló algoritmus, ami az L-rendszer módosított változatát használva hozza létre az úthálózatot, majd minden út elemet körbevesz épületekkel. Továbbá a gyalogosok és a mozgásuk is abban a félévben lettek elkészítve, ezek **Li Jiaxiang** munkája.

Az önálló labor tárgyban valósítottam meg a gyalogosokkal való kommunikációt, erre az Inworld SDK-t használtam. Ennek a használatával különböző személyiségű karaktereket tudtam létrehozni, amelyekkel a ChatGPT-hez hasonló módon lehet kommunikálni.

A szakdolgozat során a járművek létrehozásával foglalkoztam. Ez magában foglalja, a járművek véletlenszerű kinézetét, és a mozgásukat. A járművek mozgásánál fontos feladat volt, hogy az úttest jó oldalán közlekedjenek, ne ütközzenek össze, a kereszteződésekben egy adott sorrendben haladjanak át, és a gyalogosokkal meg a játékossal az átjáróknál a valós élethez hasonlóan viselkedjenek mind a gyalogosok, mind a járművek.

1.2 Dolgozat felépítése

A dolgozat 6 fejezetre van osztva, ezekben mutatom be a játék egyes elemeinek a megvalósítását. Ezen kívül a második fejezetben található egy táblázat, amiben a dolgozatban használt kifejezések jelentései találhatóak. A dolgozatban az egyes részek megvalósításának könnyebb elmagyarázhatósága érdekében kódrészletek és képek találhatóak.

1.2.1 Unity játékmotor bemutatása

A harmadik fejezetben bemutatom a Unity játékmotor előnyeit, hátrányait, hogy miért választottam ezt, és a főbb komponenseit, amiket felhasználtam a játék megvalósításához.

1.2.2 Város generáló algoritmus

A negyedik fejezetben a városgeneráló algoritmust fejtem ki, amiben leírom, hogy az L-rendszer hogyan működik, és hogy azt miként módosítottam, hogy használható legyen város generálásra. Továbbá, hogy ezt hogyan használom az úthálózat létrehozására és hogy hogyan határozom meg egy adott épületről, hogy milyen típusú legyen.

1.2.3 NPC-k generálása és mozgása

Az ötödik a fejezetben található **Li Jiaxiang** munkája. Ebben kifejti, hogy a gyalogosok hogyan lettek véletlenszerűen legenerálva, hogyan alkotta meg a gyalogosok számára azokat a részeket, amiken tudnak sétálni és azt, hogy milyen algoritmus alapján sétálnak.

1.2.4 NPC-játékos interakció

A hatodik fejezetben azt fejtem ki, hogy a játékos miként tud interakcióba lépni a gyalogosokkal, hogy a Unity-ben hogyan lehet használni az Inworld SDK-t, és hogy miket kellett módosítanom rajta, hogy a játék stílusával összhangban legyenek a karakterek is. Ezen felül ebben a fejezetben bemutatom, hogy az InworldAI online felületén hogyan hoztam létre a különböző karaktereket, amik az elkészült játékban hozzá vannak rendelve gyalogosokhoz.

1.2.5 Járművek működése

A hetedik fejezetben azt írom le, hogy hogyan oldottam meg a járművek generálódását a mozgásukat, hogy hogyan közlekednek az út megfelelő oldalán, valamint azt, hogy a kereszteződésekben milyen sorrendben haladjanak át a gyalogosok és a járművek.

1.2.6 Eredmények

Végül a nyolcadik fejezetben található az összefoglalás az előző feladatban leírt komponensek megvalósításainak eredményeiről.

1.3 Más város generáló algoritmusok

Város generáló algoritmusokból sok verziót készítettek már, melyekhez különböző megvalósításokat használtak. Ezek mind különböző kinézetű városokat eredményeztek. Ebben a fejezetben ezek közül mutatok be párat.

1.3.1 Pszeudo végtelen városok

Stefan Greuter, Jeremy Parker, Nigel Stewart és Geoff Leach által megalkotott algoritmus volt a *pszeudo végtelen város generálás*[1]. Ez az algoritmus úgy működik, hogy a játékos, ahogy halad végig a térképen folyamatosan generálódnak az épületek procedurálisan. Az épület kinézete a térképen elhelyezkedő pozíciójából van legenerálva, így pszeudo végtelen ideig tud a felhasználó menni a térképen, anélkül, hogy két ugyanolyan épület mellett menjen el. Ennek az algoritmusnak a megvalósításához elengedhetetlen az optimalizáció.

1.3.2 Perlin-zajjal generált város

Egy másik megvalósítás Niclas Olsson és Elias Frank által lett megalkotva[2]. Ez az algoritmus a Perlin-zajt használja alapjául, amit Ken Perlin alkotott meg, eredetileg modellek textúrázására. Ők úgy közelítették meg a város legenerálását, hogy először a területet kerületekre bontották, minden kerületnek különböző a kinézete. Ezután legenerálják a fő és mellékutakat, majd az utak mellett legenerálják a különböző épületeket, úgy, hogy az adott épület az adott kerületre jellemző kinézetű legyen.

1.3.3 Egy másik L-rendszerrel generált város

Pascal Müllernek van egy *L-rendszeren*[3] alapuló megvalósítása a *város generálásra*[4]. A felhasználó ennek az algoritmusnak csak a földrajzi adatokat kell megadja, hogy hol vannak hegyek például. Ebben a megvalósításban is az úthálózat generálására van használva az L-rendszer, de ebben a megvalósításban sokkal több és komplexebb szabályok találhatóak. Ezenkívül az épületek kinézete is L-rendszerrel van legenerálva, ami alapvetően három különböző típusú lehet. Ebből adódóan a legenerált város sokkal valósághűbb, mint az általam megvalósított.

2 Jelentések

1. táblázat: Gyakran használt kifejezések jelentése

Valaminek a *Pozíciója*	Valaminek a Vector3 típusú koordinátája
Valamihez képest *Fel*	Valaminek a *pozíciója* + Vector3(0,0,1)
Valamihez képest *Le*	Valaminek a *pozíciója* + Vector3(0,0,-1)
Valamihez képest *Jobbra*	Valaminek a *pozíciója* + Vector3(1,0,0)
Valamihez képest *Balra*	Valaminek a *pozíciója* + Vector3(-1,0,0)
Út	1*1 egységnyi objektum
Úttest	2 pontot összekötő utakból felépülő egyenes
Valaminek a *szomszédja*	Valaminek a *pozíciójához* képest *fel*, *le*, *jobbra*, vagy *balra* helyezkedik el

3 Unity játékmotor

A manapság elterjedt játékmotorok közül a Unity az egyik leghíresebb az *Unreal Engine[5]* mellett. Mind a kettő használható könnyedén mind 2 és 3 dimenziós játékok elkészítésére. Mind a kettőnek megvannak az előnyei és a hátrányai. Ebben a fejezetben kifejezem, hogy mi miatt döntöttem a Unity mellett.

3.1 Unity előnyei

Egyik nagy előnye az, hogy nagyon könnyen tanulható, az összetettebb komponensek működése is könnyen megérthető a részletes dokumentációnak köszönhetően. Mivel ez egy nagyon elterjedt játékmotor, ezért a közösség is nagyon nagy, ebből következik az, hogy az interneten nagyon sok segédanyag megtalálható hozzá, online kurzusok és videók formájában, valamint szintúgy emiatt, ha a fejlesztőnek kérdése van valamivel kapcsolatban, akkor könnyen meg tudja rá találni a választ a különböző fórumokon. A fő előnye az én szempontomból az Unreal Engine-hez képest, az, hogy a Unity az a Scriptekhez C# programozási nyelvet használ, míg az Unreal Engine C++-t. Ez azért volt fontos szempont számomra, mert a C# programozási nyelvvel sokkal régebb óta, és sokkal többet foglalkoztam, mint a C++-al. Ezen kívül még az is egy hatalmas előny, hogy a C# az egy sokkal modernebb nyelv, ezért egyszerűbb is a legtöbb szempontból.

3.2 Unity hátrányai

Az egyik nagy hátránya a Unity-nek, hogy a nagyobb számítógép teljesítmény igényű játékok optimalizálása nehézkes feladat lehet. Ez az én esetemben csak a nagy városoknál okozhat problémát, vagy ha nagyon sok gyalogos van a városban. Közepes és kis méretű városoknál, amelyekben kevesebb számú gyalogos van, ez nem probléma. Ami még ezen kívül hátrány lehet, az az, hogy a beépített UI elemekből nincs nagy választék, de ez sem igazán probléma hiszen az *Asset Store-ból[6]* megvásárolhatóak, illetve akár ingyenesen is beszerezhetőek mások által elkészített UI elemek.

3.3 Unity főbb komponensei

3.3.1 GameObject

A GameObject[7] az a Unity egyik alapvető építőeleme. Ez egy konténer, ami elhelyezkedik a világban. Erre lehet rárakni a többi különböző komponenst. A gyalogosok, autók, épületek, utak, de még a kamera és a fényforrások is GameObject-ek.

3.3.2 Prefab

Prefab-ek[8], azok olyan GameObject-ek, amik többször is felhasználhatóak. Ez azt jelenti, hogy ha van egy Scriptünk, ami elhelyez egy GameObject-et a világban, akkor át lehet neki adni egy Prefab-et, és akkor annak egy példánya jelenik meg. A Prefab-ek el vannak tárolva a projektünk fájlrendszerében, és ezekre lehet rakni komponenseket. Ha az eredeti Prefab-ben tárolt értékeket változtatjuk, akkor az összes példány, ami létrejön belőle, az új értékeket fogja használni. Az elkészült játékomban Prefab az utak, épületek, gyalogosok és az autók is.

3.3.3 Rigidbody

A Rigidbody[9] komponens szükséges ahhoz, hogy egy GameObject-re hatással legyenek a fizikai erők, mint a gravitáció például. Továbbá Rigidbody szükséges ahhoz, ha szeretnénk, hogy az adott GameObject interakcióba tudjon lépni Collider-ekkel. Ezen kívül még ez szükséges az autók létrehozásához, enélkül a Unity nem tudná felismerni, hogy az adott GameObject az egy autó.

3.3.4 Collider

A Collider-eket[10] Unity-ben arra szokás használni, hogy egy-egy objektumnak, ami elhelyezkedik a világban meghatározzuk a határait, amin belül helyezkedik el az objektum. Ez viszont nem feltétlen kell ténylegesen pont akkora legyen, mint az objektumunk, vannak esetek, amikor célszerű ezt kisebbre vagy akár nagyobbra beállítani, mint az objektumunk aktuális mérete. Minden Collider-en beállítható, egy paraméter, hogy isTrigger. Ez ara jó, hogy aminél ez az érték igazra van állítva, az nem állítja meg azokat a Rigidbody komponenst tartalmazó GameObject-eket, amik kapcsolatba lépnek vele, hanem lehet ennek köszönhetően érzékelni azt például, ha egy autó a kereszteződésbe érkezik, vagy kimegy onnan. Ha ez az érték hamis-ra van állítva,

akkor a Rigidbody-s GameObject nem tud átmenni a Collider határán (például a játékos nem esik át az úton). Több fajta Collider is létezik Unity-ben:

- BoxCollider: Egy téglatest. A játékban ezt használtam az utakhoz, és az autók testéhez.
- CapsuleCollider: Egy kapszula alakú Collider. Meg lehet határozni a magasságát és a szélességét. A játékban ezt használom a gyalogosoknál és a játékosnál.
- WheelCollider: Ez úgy jelenik meg, mint egy kör, de sok paraméter állítható rajta, például mivel ez egy jármű kerekét szimbolizálja, ezért rugózás is beállítható rajta, hogy milyen sebességgel forogjon… Ez csak akkor használható, ha a kerék amire ezt a Collider-t ráhelyezzük az egy olyan GameObject egyik komponense, aminek van Rigidbody komponense.
- MeshCollider: Ezt szokás használni azokra az objektumokra, amik a többi Collider-rel nem fedhetőek le. Ezt a játékban csak a házaknál használtam (megoldható lett volna BoxCollider-ekkel is, de az sokkal időigényesebb folyamat lett volna).

3.3.5 Script

Unity-ben a Script-ek[11] tartalmazzák a C# kódot. Ezekkel egyedi viselkedés adható meg a GameObject-eknek. A Script-ekkel tudnak kommunikálni a GameObject-ek (például meg egy GameObject egyik Script-jének egyik függvénye meg tudja hívni egy másik GameObject Scriptjének valamelyik függvényét). A játékban a viselkedések nagyrésze a Script-elésnek köszönhető: Script-el jönnek létre az utak, az épületek, Script-el van meghatározva, hogy hogyan mozogjanak a gyalogosok, az autók. A játékos mozgása is Script-el van megoldva.

3.3.6 Canvas

Unity-ben a Canvas-on[12] jelennek meg a UI elemek. Ezek közé tartozik a Menü, ami az „Esc” billentyű megnyomására jelenik meg, és amik ott találhatóak, és amikor a játékos beszélget egy gyalogossal, akkor a chat ablak is egy Canvas-on található, de a gyalogosok felett található „Press e” felirat is a UI réteg része.

3.3.7 CharacterController

Unity-ben a CharacterController[13] komponenssel lehet irányítani a karaktereket. Ezzel a komponenssel egyszerűbb irányítani a karaktereket, mint a Rigidbody komponenssel. Van egy Move metódusa, ezen keresztül lehet megadni a karakter mozgását. Az elkészült játékban ezt használom a játékos karakterének a mozgatására.

3.3.8 RayCast

RayCast-al[14] lehet egy pontból valamilyen irányba egy egyenest indítani, ami, ha átmegy egy másik GameObject-en, akkor azzal interakcióba lehet lépni akár. A játékban a RayCast-ot használom arra, hogy leellenőrizze a játék, hogy a játékos egy gyalogosnak a közelében van-e az „E” billentyű lenyomásakor, hogy beszélgetni tudjon az adott gyalogossal. Ezt használom az autóknál is, hogy amikor egy autó előtt meg van állva egy másik, akkor álljon meg.

4 A várost generáló algoritmus megvalósítása

Az algoritmus, ami az utakat és a mellettük levő épületeket legenerálja egy *YouTube videósorozat[15]* alapján készítettem el, módosításokat alkalmazva, hogy a célnak megfelelő legyen.

A játék elindulása után, vagy a generate gomb megnyomásának hatására indul el a teljes algoritmus, ami több elkülönülő lepésből épül fel: Az L-rendszer legenerálja a stringet, ami alapján az utak megjelennek a játéktérben, eredetileg az összes csak az egyenes út kinézettel, ezután azokon az utakon, amik kereszteződések, kanyarok, vagy csak út végződések, azokon átmegy az algoritmus, és kicseréli őket a megfelelő típusúra és jó irányba forgatja őket. Ezután az összes út mellé, ami mellett van szabad hely, be fog kerülni egy épület, úgy, hogy minden egyes épületre megnézi az algoritmus, hogy az az épület mekkora szabad területtel van körülvéve, erre a DFS algoritmust használja. Majd, amelyik épületeknél az jön ki, hogy kisebb területen helyezkedik el, mint 8 egység, akkor egy magas épületet helyez le oda, egyébként kertes házat rak le oda. Így kialakulnak a városon belül kertvárosi részek és belvárosi részek is. Majd ezek után az algoritmus végig megy az összes legenerált épületen, és mindegyik körül 11*11 egység területen földet helyez el, ezzel megszüntetve az esetlegesen keletkező lyukakat a városban, és körbeveszi a várost.

4.1 Az L-rendszer megvalósítása

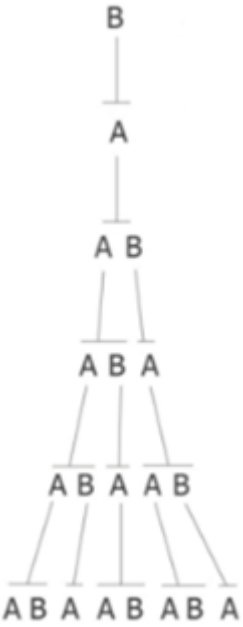
Az L-rendszer az egy formális nyelv, amit Lindenmayer Arisztid magyar biológus talált ki eredetileg a növények sejtjeinek modellezésére. Ennek az algoritmusnak számtalan megvalósítása létezik, de mindegyiknek ugyanaz az alapötlete: van egy ABC-je, amiben szerepelnek a konstansok és a változók, vannak szabályok, amik egy változót kicserélnek a szóból egy előre meghatározott (hosszabb) ABC-beli karakter sorozatra, és még az algoritmusnak meg kell adni egy kezdeti állapotot is. Ezután a szabályok alapján a generált szóra lefuttatható az algoritmus, annyiszor, amennyi meg lett adva, minden lefutás alkalmával hosszabbá téve az előző szót. Az általam megvalósított algoritmus is ilyen.

4.1.1 Az L-rendszer működése

Az algoritmus eredeti verziója, az „alga”:

- Változók: A, B
- Kezdet: A
- Szabályok: (A->AB), (B->A)

Ezekkel a beállításokkal az algoritmus n lefutás után így néz ki:



1. ábra: az "alga" algoritmus eredménye

n=0	B
n=1	A
n=2	AB
n=3	ABA
n=4	ABAAB
n=5	ABAABABA
n=6	ABAABABAABAAB
…	…

Ezt az algoritmust lehet módosítani, sok féle képen, de amit én választottam az a teknősös módszer:

4.1.2 Az L-rendszer módosítása

- Változók: F, G
- Konstansok: +, -, [,]
- Kezdet: [F]-[G]-[F]-[G]
- Szabályok:
  - (F->F[+G]) vagy (F->G[+F])
  - (G->F[-G]) vagy (G->G[-F])[1]

Amiben a karakterek jelentése:

- F: Menjen a teknős előre egy megadott távolságot.
- G: Menjen a teknős előre egy megadott távolságot.
- +: Forduljon jobbra 90º-ot a teknős.
- -: Forduljon balra 90º-ot a teknős.
- [: A teknős jelenlegi irányát és pozícióját mentse el a stack tetejére
-]: A stack tetején levő pozícióra menjen vissza a teknős, és forduljon az ott elmentett irányba, és törölje a stack tetejéről azt.

Ez a megvalósítás nagyon hasonlít a *sárkány-görbéhez[16],* de mivel a kezdeti állapot meg a szabályok is mások, mint az abban leírtak, így az algoritmus alapján generált görbe eltérően néz ki. Továbbá a megvalósított algoritmusomban van randomizálás is, ami „levág" a fából ágakat és minden változónál véletlenszerű, hogy melyik szabály lesz rá alkalmazva, ezért egy olyan megvalósítást választottam, ami hasonlít a sárkány-görbéhez, mivel abban is négyzetek szerepelnek, és egy város is nagyjából négyzetekből és téglalapokból épül fel. Ezért, ha a sárkány-görbében lévő négyzetek oldalait meghosszabbítjuk, meg párat elhagyunk, akkor a keletkező görbe az szinte ugyan az, mint az általam készített algoritmus által generált görbe.

4.2 Az utak generálása

Az algoritmus végigmegy a legenerált string-en, F és G karaktert találva megváltoztatja a jelenlegi pozíciót 2-vel, abba az irányba, amit jelenleg az irány változó jelöl, majd lerak az egyenes út típusból abba az irányba 2-t, ami jelenleg be van állítva (a fordulások és a mentés betöltések alapján), úgy, hogy elforgatja a jó irányba, amit az alapján tud megtenni, mivel tudja, hogy ez az út szakasz melyik irányba megy (az út szakasz generálásánál meg kell adni a kiinduló pontot, az irányt, meg a hosszt). A + és – karakterek annyit csinálnak, hogy megváltoztatják az osztályban szereplő direkción

[1] Úgy kell értelmezni ezeket a szabályokat, hogy az egy sorban szereplők közül véletlenszerűen választ a program, hogy melyik legyen megvalósítva, az aktuális változóra.

változót. [karaktert olvasva elmenti a stack tömb tetejére a jelenlegi pozíciót és irányt.] karaktert olvasva a jelenlegi pozíciót és irányt beállítja a stack tetején lévő értékekre, majd ezeket az értékeket törli onnan. Miután létrejött az összes út, akkor végigmegy egy algoritmus az összes úton, és beállítja, hogy hármas kereszteződés, négyes kereszteződés, kanyar, vagy út vége típusú legyen, amit az alapján tesz, hogy hány szomszédja van, és azok az adott úttól milyen irányba vannak:

- Ha 1 szomszédja van, akkor az egy útszakasznak a vége, tehát út vége típusú lesz, úgy forgatva, hogy abba az irányba nézzen, amerre a szomszédja van.
- Ha 2 szomszédja van, akkor lehet kanyar vagy egyenes típusú út:
    - Ha a szomszédjai ellenkező irányban vannak, akkor nem változik meg, tehát marad egyenes út típusú.
    - Ellenkező esetben pedig kanyar típusú kell legyen, és úgy kell forgatni, hogy abba a két irányba nézzen az út, amerre van az a két szomszéd.
- Ha 3 szomszédja van, akkor hármas kereszteződés típusú, és úgy kell forgatni, hogy abba a három irányba nézzen, amelyik irányokban vannak szomszédjai.
- Ha 4 szomszédja van, akkor csak ki kell cserélni négyes kereszteződés típusú útra, amit nem szükséges elforgatni, mivel ez minden irányban ugyanúgy néz ki.

4.3 Az épületek generálása

Egy algoritmus végigmegy az összes lerakott úton, és egy Dictionary-ba beleteszi azokat a pozíciókat, amik nem tartalmaznak utat, és legalább egy szomszédja út, ezeket kulcsként tárolja el, és értékként hozzárendeli, hogy az egyik szomszédos úthoz képest ez milyen irányban van. Ezután ezen a Dictionary-n végigmegy és mindegyik kulcsra megnézi, hogy mekkora területen van: DFS algoritmussal meghatározza, hogy az utak hány egységnyi területű olyan területet vesznek körbe, ami tartalmazza azt a pozíciót, aminél jelenleg tart. Majd miután ezt a számot megtudta, ha az a terület 8-nál kisebb, akkor egy magas épületet fog oda rakni, egyébként egy kertes házat:

- A magas épület: Jelenleg 2 különböző magas épület típus van, amik között a különbség az jelenleg csak a szín, de a jövőben ez kibővíthető. Egy magas épület 3 különböző Prefab-ből áll:

    - Egy darab base: ez kerül legalulra, ezen van egy ajtó, ami miatt el kell fordítani úgy ezt a Prefab-et lerakáskor, hogy út felé nézzen.
    - N darab középső rész: a felhasználó meg tudja adni, hogy ebből minimum és maximum hány darab kerülhet a base-re, de az alapértelmezett, az 1 és 100, és a minimum és a maximumot a felhasználó csak ez a két érték közöttire veheti föl. A középső részt nem forgatja el az algoritmus, mert úgy terveztem, hogy ugyanúgy nézzen ki mind a 4 irányba.
    - Egy darab tető: ez kerül rá a legfelső középső részre, és ez sincs elforgatva.
- Kertes ház: jelenleg összesen van 12 különböző, ezek közül véletlenszerűen választ egyet az algoritmus, és azt rakja oda, a jó irányba forgatva.

A leírt algoritmus jellegéből adódóan, általában úgy fog legenerálódni a város, hogy szinte mindenhol kertesház van, és egyes területeken csak magas épületek helyezkednek el, ezzel olyan hatásúvá téve azt a területet, mintha az lenne a belváros, a többi terület pedig a város kertvárosi része.

4.4 A föld generálása

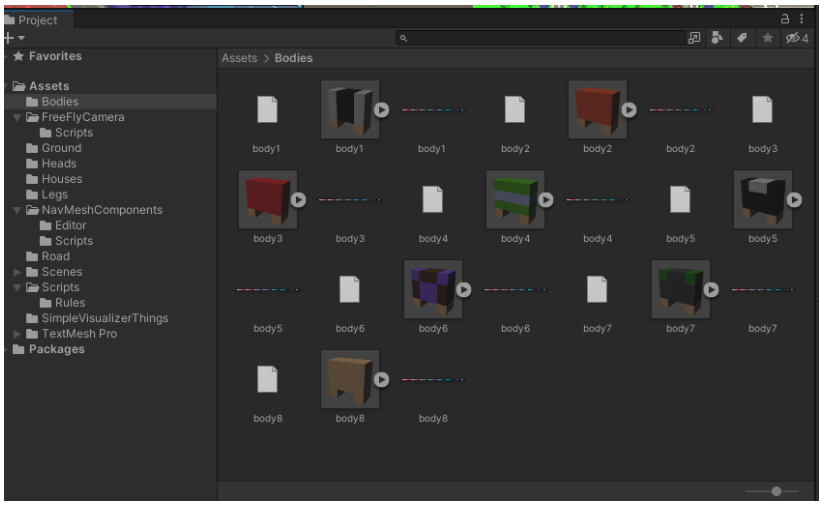
Az utakhoz és az épületekhez képest a földterületek legenerálódása sokkal egyszerűbb. Ennek az algoritmusa abból áll, hogy végigmegy az összes lerakott épületen és az aktuális épületet véve középpontul, 11*11-es területen azokra a helyekre, ahova még nem lett helyezve sem út, sem épület, oda lehelyez egy föld típusú mezőt, ami véletlenszerűen van kiválasztva a megadottak közül (jelenleg 4 fajta lehet), és véletlenszerű irányba elforgatja azt.

5 Az NPC-k (Non-playable character) generálása és mozgása

Témalabor keretében, szaktársam, **Li Jiaxiang** felelt az e fejezetben bemutatott feladatok kidolgozásáért és azok megoldásáért, ez a rész az ő Témalabor dokumentációjából lett átmásolva:

5.1 Különböző kinézetű NPC-k

5.1.1 Prefabek



2. ábra: body Prefab-ek Unity-ben

A cél különböző kinézetű NPC-k (innentől kezdve emberek) generálása volt. Ennek megvalósítására a *Magicavoxel*[17] nevezetű programot használtam. Ebben könnyen létrehozhatók és importálhatóak objektumok melyeket Unity-ben egyszerű használni. Készítettem 8 féle fejet, törzset és lábat, amely összesen 8x8x8 = 512 fajta ember generálását jelenti. Ezek Unity-be importálva Prefab-ként használhatóak.

5.1.2 Emberek összerakása

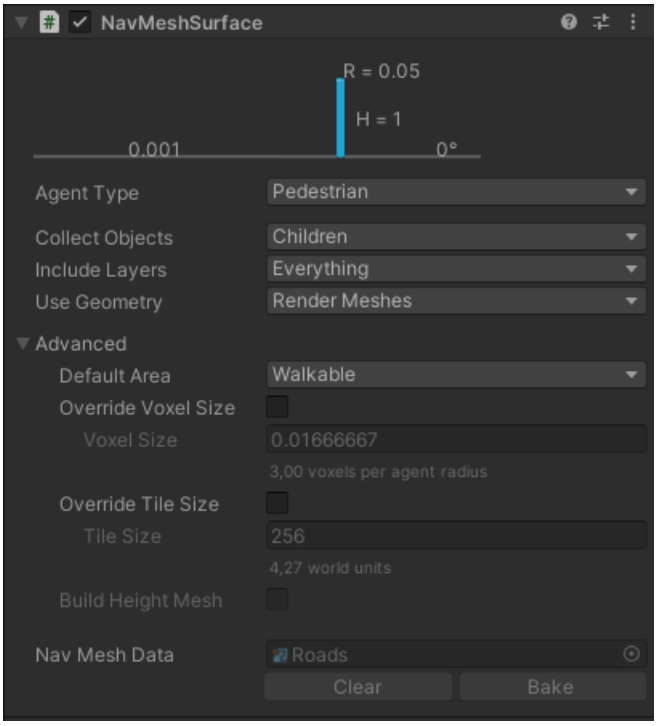
A testrészek tárolására egy külön osztályt hoztam létre: a BodyParts nevű osztályt, amely egy GameObject-ekből álló tömböt tartalmaz. Egy függvénye van ezen kívül, amely ebből a listából egy véletlenszerű elemmel tér vissza. A PedestrianSpawner osztály felelős a gyalogosok generálásáért, ami tartalmaz egy BodyParts-okat tartalmazó tömböt. Ebben a tömbben három elem van, az egyik a fejeket tartalmazó tömb, a másik a

törzseket, a harmadik pedig a lábakat tartalmazza. Egy gyalogos létrehozásakor mindhárom tömbből kiválaszt egy véletlenszerű elemet, majd ezeket létrehozza és egy közös szülő objektumhoz csatolja, így lesznek egy objektumként kezelhetőek. A generált gyalogosokat egy lista tárolja, hogy a város újra generálásakor törölni lehessen őket és újra létrehozni.

5.2 NavMeshComponents

Unity-ben van beépített navigációs rendszer, mely a UnityEngine.AI-ban található meg. A gyalogosok mozgásának hálózatához ezt választottam kiindulási alapnak. A beépített Navigation Systemnél, a valamivel bővebb és fejlettebb eszközöket tartalmazó *NavMeshComponents[18]* package-ből használtam, főként elemeket. Ezt a projektet az *AI Navigation package*[19] részeként fejlesztik és GitHub-on elérhető.

5.2.1 NavMeshSurface



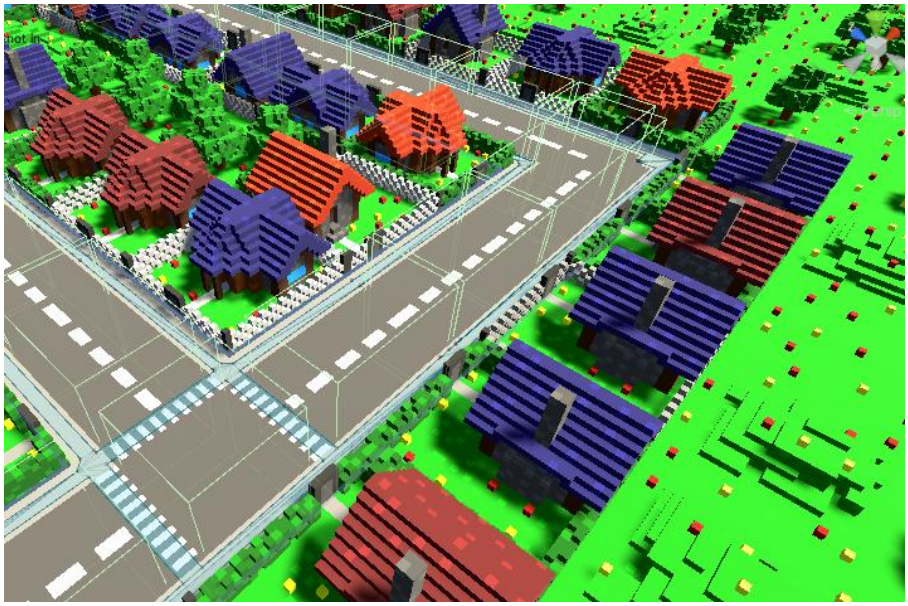
3. ábra: NavMeshSurface osztály

A *NavMesh*[20] lényege különböző típusú felületek létrehozása. Unity-ben beépítetten léteznek a Walkable, Not Walkable és Jump felületek. Az alap ötlet az utak számára való Walkable felület létrehozása volt, amelyen majd a gyalogosok tudnak közlekedni. Ilyen felületeket Bake-elni tudunk meglévő objektumokra. Itt felmerül egy probléma: a program futásidőben generálja a várost, ezért nem léteznek előre az

objektumok, amire le lehetne generálni a felületeket. Erre adott megoldást a NavMeshComponents *NavMeshSurface*[21] osztálya, amellyel futásidőben tudunk NavMesh felületeket létrehozni és kezelni. A NavMeshBaker osztály szolgál az utakra való NavMesh generálására. A program indításakor az algoritmus alapján legenrálódik a város. Az út objektumokat egy Roads nevű szülő objektum gyerekeiként hozza létre. A Roads objektum a feladata, hogy tartalmazza az összes utat, így egységesen kezelhetőek lesznek. Ezen túl egy NavMeshSurface komponenst és a NavMeshBaker scirptet tartalmaza. A NavmeshSurface komponens miatt önmagára, azaz az összes útra egyszerre lesz generálható a járható felület. A NavMeshBaker egy listát tartalmaz a felületekről, amelyekre generálni kell a NavMesh-t. Jelen esetben ez egy elemet tartalmaz, a Roads objektum saját magát helyezi el benne. A Navmesh végigmegy a listán és mindegyik felületre meghívja a BuildNavMesh függvényt. Ezáltal létrejött a felület, ami csak az utakból áll és járható.

5.2.2 NavMeshObstacles

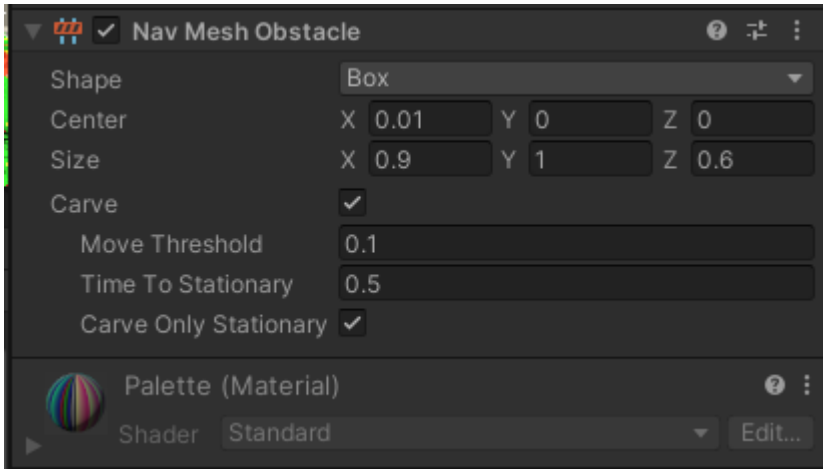
Az utaknak nem a teljes felülete lehet járható, hanem csak a járda vagy zebra része. Ennek megoldására több fajta technika is létezik, az egyik NavMeshModifierVolume, amellyel felületek típusán tudunk változtatni. Megoldás lett volna ilyen objektumok ráhelyezése az utakra és Not Walkable area-ra állítani őket.



4. ábra: NavMeshObstacle objektumok az utakon

Az általam használt technika a *NavMeshObstacle*[22]. Ezek lehelyezése tulajdonképpen akadály, amelyeken a gyalogosok nem fognak tudni átmenni és az

útvonaltervezéskor ezeket kikerülik. Ilyen objektumokkal lettek lefedve a különböző út típusok. Meg lehet adni nekik formát (téglatest, kapszula), méretet és az őt tartalmazó objektumhoz képest lévő relatív elhelyezkedését. Ezeknek a paramétereknek egyedi beállításával az összes út típusra létre lett hozva a megfelelő lefedés, ami csak a járdát és zebrát nem fedi. A kanyarodó utaknál kettő ilyen objektum is fel lett használva, hiszen egy téglatest nem tudja lefedni a kanyart, amihez komplexebb forma szükséges. Ezeket az objektumokat az utak létrehozása után rakja rá az attachObstacles függvény, amely az alapján ismeri fel az utakat, hogy hány vagy éppen milyen irányba vannak szomszédai az adott útszakasznak. A kanyarnál mind a négy esetet külön kellett kezelni, hiszen ilyenkor az Obstacle objektumokat forgatni kellett, nem úgy, mint a négyes kereszteződésnél, ahol csak középre kellett helyezni egy téglatestet.



5. ábra: NavMeshObstacle

A NavMeshObstacle-nek ezen túl van még egy tulajdonsága a Carve. Ennek a tulajdonságnak a használatával el lehet érni, hogy míg az obstacle ott van, addig kivájja a vele érintkező objektumot. Ennek használata azért fontos, mert az útvonalkeresés során így a gyalogosok nem fognak nekimenni az Obstacle-nek és csúszni a felületükön, hanem ki fogják kerülni azt. Ez minden Obstacle objektumon true-ra van állítva.

5.3 Az emberek mozgása

5.3.1 PedestrianMovement

Ez az osztály felelős a gyalogosok (Agentek) mozgásáért. Minden gyalogos (Agent) tartalmazza ezt a scriptet. Az Agentek kezdeti pozícióját random kapja, a NavMesh.CalculateTriangulation függvény segítségével. Ez a függvény a jelenlegi

NavMesh-t osztja fel sokszögekre(polygon-okra), amikből random kiválasztva egy csúcsot majd a SamplePosition függvényt használva, amely az ahhoz legközelebbi pontot adja a NavMesh-en. Ez lesz kiválasztva kezdő pozíciónak, ahova az Agent kerül. (A Warp függvénny alkalmazható erre, hogy az Agentet a megfelelő helyre tegye). A PedestrianMovement-ben ezen túl a Start és Update függvények vannak definiálva. A Start-ban a szülő objektumtól kéri le az Agent komponenst, valamint kiszámolja a triangulation-t. Az Updateben (ami frame-enként hívódik meg), megnézi, hogy az Agent-nek van-e aktív útvonala, ha nincs akkor hasonlóan a kezdő pozíció generáláshoz, generál egy másik pontot, amit beállítva célként elindul az Agent. Ezáltal folyamatosan mozogni fognak, amint elérik a céljukat egy új cél felé fognak elindulni. Ha valamiért nem sikerülne elérni a célt, az autoRepath engedélyezésével újra kalkulálja az útvonalat.

5.3.2 A* algoritmus

A Unity beépített útvonalkeresője az *A* algoritmust*[23] használja. A kezdő és végpontot közötti polygon felosztásokat nézi végig. A NavMesh tartalmazza ezeknek a polygonoknak az információit, egyes polygonok szomszédait. Ez a felosztás hasznos, hiszen a polygonok belsejében biztosan nincsen akadály. Az A* algoritmus súlyozott gráfokat használ. Célja egy kezdőpont és végpont közötti legkisebb költségű út megtalálása. Fa gráfot tárol az indulási ponttól kezdve és minden lépésben egyel tovább halad. A következő node kiválasztása az eddigi és a végponthoz eljutásig szükséges költség összege alapján adódik.

$$f(n) = g(n) + h(n)$$

1. egyenlet: A* algoritmus egyenlet

Jelen képletben az „n” a következő node, a g(n) az eddig megtett út költsége a kezdő ponttól, a h(n) pedig a becsült érték n-től a cél node-ig. Minden lépésben a legkisebb f(x) kikerül a sorból, a szomszédainak értékei ezáltal frissülnek majd ez addig folytatódik, míg a kivett pont egy cél node nem lesz, a végeredmény a legkisebb költségű út lesz. A node-ok megtartják az előző node-ot így visszavezethetőek lesznek, tehát egy útvonalat fog találni. Unityben a NavMesh polygonjain végig futtatva az algoritmust megtalálja a kedvező útvonalat.

6 NPC interakció

A játékos interakcióba tud lépni a 4-es fejezetben leírt módszerekkel létrehozott gyalogosokkal. A fejük felett van egy felirat: „Press E”. Ha a játékos egy gyalogos közelében van és nyom egy E betűt, akkor az adott gyalogos ennek hatására megáll, és megjelenik a chat felület: egy ScrollView, amibe fognak belekerülni a felhasználó által írt üzenetek, illetve az ezekre érkezett válaszok. Ez alatt van egy InputField, ahova a felhasználó írja be az üzenetet, ettől jobbra pedig egy „Bye” feliratú gomb található, ami befejezi a beszélgetést, ezzel bezárva a chat felületet, és a gyalogos tovább sétál.

6. ábra: Chat felület

6.1 NPC-k „agya”

Az NPC-kkel a beszélgetés hasonló a ChatGPT-vel való kommunikációhoz: lehet feltenni kérdést, amire válaszol a gyalogos. Viszont az általam megvalósított NPC-k nem a ChatGPT-t használják, hanem *InworldAI-ban[24]* létrehozott karaktereket. Ez abban különbözik a ChatGPT-től, hogy itt lehet létrehozni egyéni karaktereket, különböző személyiségekkel.

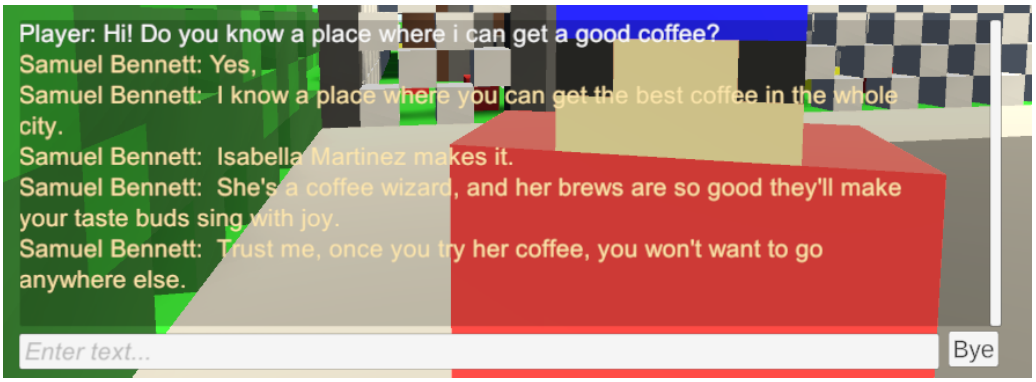
6.1.1 Inworld Scene

Ezeket a karaktereket az Inworld studio online felületén lehet létrehozni. Először létre kell hozni egy Scene-t, ez fogja tartalmazni majd a karaktereinket. A Scene-nek része egy leírás: ez alapján tudnak majd beszélni a karaktereink a városról például, ha kérdezzük őket róla. Ezenkívül lehet a Scene-hez hozzárendelni Trigger-eket, amelyek

olyan események, mint például egy földrengés. Ha egy Triggert Unityben elsütünk, akkor mindegyik karakterünk fog arról beszélni, hogy ez megtörtént.

6.1.2 Inworld Common Knowledge

Common Knowledge-el lehet olyan dolgokat megadni, amiket minden NPC-tud. Én példaként létrehoztam egy olyan Common Knowledge-et, hogy az egyik karakter (Isabella Martinez) készíti a legfinomabb kávét az egész városban. Ehhez hozzárendeltem az összes karaktert és így egy beszélgetés során meg lehet kérdezni, akárkitől, hogy ki csinálja a kedvenc kávéját.



7. ábra: NPC használja a Common Knowledge-et

6.1.3 Inworld Characters

Az Inworld studio-ban a Create new character gombra kattintva tudunk létrehozni új karaktert. A felugró ablakban meg kell adni a karakter nevét és kell adni egy leírást a karakternek. Ezután választhatjuk az auto generate opciót, amivel a többi személyiségjegy automatikusan le lesz generálva, de ha egyedibb karaktert szeretnénk csinálni, akkor azt is megtehetjük, én az összes karakteremnél az utóbbit választottam.

6.1.3.1 Core Description

Itt található egy leírás a karakterünknek, a motivációi és a hibái, kihívásai. Ezek mind egyénileg megadhatóak, és a leíráson kívül ki is hagyhatók. Itt van még egy checkbox is, amit kipipálva a beszélgetés narrálva lesz: például XY felnéz a könyvéből. Ezt a funkciót kikapcsoltam mindegyik karakternél, mivel a cél az volt, hogy valamennyire élethű legyen az NPC-kkel a beszélgetés.

6.1.3.2 Identity

Itt lehet megváltoztatni a karakterünk nevét, meg lehet adni, hogy milyen névmást használ a karakter (she/her, he/him, they/them). A Role-ban adtam meg mindegyik karakternek a foglalkozását. Ki lehet választani, hogy milyen idős az adott karakter, lehet megadni neki beceneveket, hobbikat. Továbbá meg lehet adni egy Wikipedia linket, ha nem egy kitalált karaktert hozunk létre.

6.1.3.3 Personality

Itt meg lehet adni a karakterünk egyes jellemvonásait. Itt lehet megadni 1-től 9-es skálán a következő személyiségjegyek mennyire jellemzőek az adott karakterre:

- Hangulat: szomorú – vidám, harag – félelem, ellenszenv – bizalom, felkészült – meglephető
- Személyiség: negatív – pozitív, agresszív – békés, elővigyázatos – vakmerő, introvertált – extrovertált, bizonytalan - magabiztos
- Érzelmi rugalmasság: állandó – dinamikus

6.1.3.4 Facts and Knowledge

Itt lehet megadni olyan tudásokat, amiket csak az adott karakter tud, például lehet valakinek egy titka, amit egyik másik karakter tud. Továbbá hozzá is lehet rendelni a karakterhez a már létrehozott Common Knowledge-et.

6.1.3.5 Voice

Be lehet állítani a karakternek egyedi hangot: vannak előre elkészítettek, ezek közül lehet választani. Be lehet állítani a hangmagasságot, és a beszéd sebességét is. Ha beállítottuk ezeket, akkor meg lehet hallgatni, és lehet finomítani a beállításainkon kedvünk szerint. Ezt a funkciót a karaktereimnél alapértelmezetten hagytam, mivel nem terveztem használni a hang funkciót.

6.1.3.6 Dialogue Style

Itt lehet beállítani, hogy a karakterünk milyen stílusban kommunikáljon például formálisan, lazán, szarkasztikusan. Meg lehet adni egy példa párbeszédet, és majd erre fog hasonlítani a karakterrel a beszélgetés.

6.1.3.7 Goals and actions

Itt lehet megadni olyan Trigger-eket, amik hatására reagál a karakterünk úgy ahogy definiáljuk. Például be lehet állítani azt, ha interakcióba lép a játékossal, akkor legyen boldogabb.

6.1.3.8 Scenes

Itt lehet hozzáadni a karaktert egy vagy több Scene-hez.

6.2 Inworld SDK használata

Az Inworld támogatás nyújt más vidójáték motorokhoz is, például az Unreal engine-hez is, de akár lehet használni az Inworld-ös karaktereinket Minecraft-ban vagy Robloxban is. Ezekhez mind van dokumentáció, hogy hogyan lehet megvalósítani. Unity-ben. Úgy lehet használni, hogy először a Unity asset store-ból le kellett tölteni az SDK-t, majd be kellett importálni azt a projektbe. Így az Asset mappába belekerül egy új mappa: Inworld.Ai, majd ezen belül a Resources mappában az APIKeys mappában a Default Workspace Keyübe be kell állítani az api kulcsot és secret-et, amiket az Inwolrd studio-n tudunk generálni. Ezután be kell jelentkezni a fiókunkba a MenuBar-ban található Inworld gombra kattintva.

6.2.1 Inworld karakter hozzárendelése gyalogoshoz

Miután bejelentkeztünk, megtalálhatóak a karaktereink Prefab-ekként, amik már most behúzhatóak a játékba, és hozzárendelve a Player objektumot már lehet velük kommunikálni a mikrofonon és hangszórón keresztül szóban. Alapból karakter prefab-nek része egy avatár, ami animálható és a fejük felett buborékban szerepel a beszélgetés. Ezeket eltávolítottam. Van egy InworldController a Unity-s Scene-ben, ezen keresztül kommunikál az alkalmazásunk az Inworld API-val. Az InwolrdController tartalmaz az összes InworldCharacter prefab-ből egy példányt. Az InworldCharacter prefab-et akkor rendelem hozzá a gyalogoshoz, amikor a gyalogos megszületik, hasonlóan ahhoz, ahogy egy gyalogoshoz hozzá van rendelve a teste, lába, feje:

```
var body = Instantiate(parts[0].getPrefab(), new Vector3(0, 0.091f, 0),
Quaternion.Euler(0, 90, 0));
var head = Instantiate(parts[1].getPrefab(), new Vector3(0, 0.039f, 0),
Quaternion.Euler(0, 90, 0));
var legs = Instantiate(parts[2].getPrefab(), new Vector3(0, 0, 0),
Quaternion.Euler(0, 90, 0));
var ai = Instantiate(inworldCharacters[Random.Range(0,
inworldCharacters.Count)], new Vector3(0, 0, 0), Quaternion.Euler(0, 0, 0));
```

```
body.transform.parent = pedestrian.transform;
head.transform.parent = pedestrian.transform;
legs.transform.parent = pedestrian.transform;
ai.transform.parent = pedestrian.transform;
```

Ahogy látszódik mivel véletlenszerűen választunk egy InworldCharacter-t az inworldCharacter listából, ezért lesznek ismétlődések, több gyalogosnak lesz ugyanaz a személyisége. Mindegyik InworldCharacter prefab-nek van egy OnCharacterSpeaks eventje, ami akkor sül el, amikor beszél a karakter, két paramétere van: a karakter neve és az üzenet szövege. Ebben meg lehet adni, hogy ezekkel a paraméterekkel melyik GameObject-nek melyik Script-jének melyik függvénye legyen meghívva. Erre létrehoztam egy új osztályt: Chat.

6.2.2 Chat osztály

Ez az osztály felelős azért, hogy a chat ablak megjelenjen amikor a felhasználó megnyomja az E billentyűt továbbá, hogy az üzenetek megjelenjenek mind az Inworld- ös karaktertől, mind a felhasználótól. A felhasználónak és az Inworld karaktereknek az üzenetei más színűek, hogy könnyen meg lehessen azokat különböztetni. Ebben az osztályban található a SendAIResponse függvény, ez az, ami az OnCharacterSpeaks event elsülésénél hívódik meg. Ez a függvény meghív egy Send függvényt, aminek két paramétere van: az üzenet, és egy enum, ami alapján van eldöntve, hogy az üzenet milyen színű legyen. A Send függvény meghívódik akkor, amikor a felhasználó beírta az üzenetet az InputField-be és nyom egy Enter-t.

7 Járművek működése

7.1 Járművek generálása

Az összes autó ugyanúgy épül fel a játékban: van egy body komponense és 4 darab kereke. A body komponens az véletlenszerűen választóik ki 12 közül (két típus van, mind a két típusból van 6 szín). Az, hogy melyik body lesz az adott autónak kiválasztva, az a létrejöttekor derül ki.

A felhasználó kiválaszthatja, hogy maximum hány darab autót szeretne lerakni a városban. Azért nem pontosan annyi fog keletkezni, mert minden maximum annyi autó lehet, ahány út objektum van. Amikor a felhasználó rányom a GENERATE gombra (vagy a játék indulásakor) akkor miután az összes úttest objektum le lett generálva akkor megpróbál annyi autót lerakni, amennyit a felhasználó beállított maximumnak, úgy, hogy egy for ciklus addig megy ameddig el nem éri azt a számot. A for ciklusban minden iterációnál az utakat tartalmazó listából kiválaszt véletlenszerűen egyet, és annak a koordinátáját belerakja egy másik listába, ha abban a listában ez az elem még nem szerepel. Ezután ezekben a koordinátákban fogja létrehozni az összes autót.

7.2 Útvonal generálása

Az autók számára útvonal generálára két lehetséges megoldás közül kellett választanom. Az egyik megoldás az lett volna, hogy mind az 5 különböző út elemhez megalkotni a pontokat, amiken mehet az adott autó, és ebből csinálni egy irányított gráfot.

Ezt úgy lehetett volna megcsinálni, hogy empty GameObject-eket helyezünk el az egyes út elem Prefab-eken, ezek *marker*ek lennének. Ezután meg kellene határozni, hogy melyik *marker*ből melyik *marker*ekbe lehetséges eljutni. Miután mind az 5 Prefab-nél ezek meg lettek csinálva, akkor valamilyen logika szerint össze kell kötni az egymással szomszédosan elhelyezett Prefab-eket. Ez a megoldás azért nem volt a legoptimálisabb, mert feleslegesen sok extra GameObject-et kell létrehozni, viszont, ha ezt választottam volna akkor egyszerűbb lett volna a megfelelő út generálása.

A másik megoldás az, amit végül választottam, hogy egy útvonalat kiválaszt az algoritmus kezdetben, úgy, hogy azoknak úttesteknek koordinátáit tárolja el egy listában, amiken végig kell menjen, majd az alapján, hogy egymáshoz képest merre találhatóak ezek az úttestek, eltolja a cél koordinátát, hogy az autó az úttest jobb oldalán közlekedjen.

Így nem keletkeznek felesleges extra GameObject-ek, viszont ez így extra számításokhoz vezet.

Ezután a következő lépés az útvonal kereső algoritmus kitalálása volt. Az egyik lehetőség az lett volna, hogy az úthálózatból kiválaszt az algoritmus véletlenszerűen két különböző elemet, majd a kettő között a *Dijkstra-algoritmussal[25], A* algoritmussal[23]*, vagy egy másik útvonal kereső algoritmussal megtalálni a két pont között egy lehetséges utat. Ez a megoldás azért nem volt megfelelő, mert nagyon sok számítással járnak, és ha nagyon sok autó van egyszerre, akkor sokkal erőforrás igényesebb így meghatározni az útvonalat. Ezen kívül ez még azért is lett volna nem a megfelelő megoldás, mert akkor az autók határozottan mennének arra a pontra, ahova menniük kell, ami valósághű, viszont én inkább azt szerettem volna, hogy azt a hatást keltsék a járművek, hogy nem ismerik a várost, nem tudják mi hol van, hova szeretnének menni, ezért választottam a következő megoldást:

Az útvonal az autó számára az egy Vector3 lista, amiben a koordináták vannak, amiken az autó át kell menjen, ez a lista lesz a *path*. Ennek generálódásához kell egy kezdeti pont (ez a 7.1-es alfejezetben keletkező lista elemei). A *path*-ba belekerül a kezdőpont, majd a többi elem ezzel a for ciklussal kerül bele a *path* listába:

```
for (int i=0; i<pathLength; i++)
{
     if (i == 0)
     {
          var possibleNexts = getPossibleNexts(start, start, roads);
          path.Add(possibleNexts[Random.Range(0, possibleNexts.Count)]);
     }
     else
     {
          var possibleNexts = getPossibleNexts(path[i-1], path[i], roads);
          path.Add(possibleNexts[Random.Range(0, possibleNexts.Count)]);
     }
}
```

Ebben a for ciklusban a pathLength változó az amilyen hosszú legyen a generálandó lista. Az i = 0 elemet azért kell külön kezelni, mert a getPossibleNexts függvénynek paraméterként kell a lista két utolsó eleme, de i = 0 esetében még csak 1 elemű a lista. A getPossibleNexts függvény azokat a koordinátákat adja vissza, amik az utolsó elemnek a listában azok a szomszédjai, amik nem a lista utolsó előtti eleme (így nem fog út közben visszafordulni az autó). Viszont, ha nincs olyan szomszédja annak az útnak, ami nem az előtte lévő elem, az azt jelenti, hogy ez egy zsákutca. Ezen a ponton két lehetőségünk lenne:

- Ez lesz az utolsó elem a listában: Ez azért nem szép megoldás, mert akkor így eléggé rövid útvonalak is lehetnek.
- A másik megoldás az az, hogy ebben az esetben visszafordítjuk az autót. Ezt a megoldást választottam, mert így minden esetben a megadott hosszúságú út lesz generálva, de így meg kellett oldani, hogy itt vissza tudjon fordulni az autó.

Így le lett generálva a lista, de ez még így nem lesz jó, mivel ebben az utaknak a közepe van koordinátákként, ami nem megfelelő, mert az autók az úttest jobb oldalán kell közlekedjenek. Ennek a javításához először egy for ciklussal végig kell menni a *path* listán és kell csinálni belőle egy másik listát. Ebben a listában irányok lesznek, amik megmondják, hogy az koordinátából milyen irányba található a következő koordináta. Ezt úgy számolja ki, hogy a következő koordinátából kivonja az aktuálist. Ezzel a következő koordináták közül kapjuk valamelyiket: (1,0,0), (-1,0,0), (0,0,1), (0,0,-1). Ezekből a koordinátákból egyértelműen meghatározhatóak az irányok.

Az irányokat tartalmazó listából és a koordinátákat tartalmazó listából már ki lehet számolni, hogy hova kell legyen eltolva a koordináta. Az *i*-edik koordinátának az eltolását ki lehet számolni az *i*-edik és az *i-1*-edik irányból:

- Összesen 16 kombináció lehet ezekből (az *i*-edik és az *i-1*-edik elem lehet fel, le, jobbra, balra)
- Ha az *i*-edik irány ugyanaz, mint az *i-1*-edik, akkor az azt jelenti, hogy nem kell forduljon az autó. Mivel tudjuk, hogy melyik irányba megy az autó, így a megfelelő koordinátát könnyű eltolni: például, ha felfele megy, akkor az X koordinátának kell növelni az értékét, ezzel eltolva jobbra egy keveset, pont annyit, hogy jobb oldalon menjen az autó. Ugyanígy a többi 3 iránynál is.
- Ha az *i*-edik irány pont az ellenkezője az *i-1*-ediknek (fel-le, jobbra-balra), akkor az azt jelenti, hogy ott vissza kell fordulni. Ebben az egy esetben viszont nem csak el kell tolni az adott koordinátát, hanem ki kell cserélni 3 koordinátára, hogy megfelelő ívben kanyarodjon. Ha például az *i*-edik elem a felfele (tehát fentről érkezik az autó és felfele fog menni), akkor a három koordináta, ami lesz az *i*-edik helyett, az az *i*-edik eltolva balra,

lefele, jobbra, ebben a sorrendben. Ennél is ugyanígy kell kiszámolni a többi irányra is.

- Jobb kanyar, ha a következőek közül valamelyik teljesül: *i-1* = fel, *i* = jobbra vagy *i-1* = le, *i* = balra vagy *i-1* = jobbra, *i* = le vagy *i-1* = balra, *i* = fel. Ennél abba az irányba kell eltolni, amelyik az *i*-edik elem, tehát ha az egy fel, akkor felfele kell eltolni. Eredetileg ezt nem csak abba az irányba toltam el, hanem jobbra is, de ez azért nem volt megfelelő, mert úgy túlságosan levágta a kanyart, és ráhajtott a járdára.
- Ezek után már csak 4 lehetséges eset maradt: *i-1* = fel, *i* = balra vagy *i-1* = le, *i* = jobbra vagy *i-1* = jobbra, *i* = fel vagy *i-1* = balra, *i* = le. Ezek a bal kanyarok. Itt, ha az *i*-edik elem a felfele, akkor a koordinátát azt jobbra és le kell eltolni.

Így most már a megfelelő koordináták vannak a listában. Már csak Instantiate-elni kell a kezdőpontot az autót, és át kell adni neki a *path*-ot, hogy végig tudjon menni rajta.

8. ábra: generált path koordináták

9. ábra: generált path koordináták, menetiránnyal

7.3 Járművek mozgása

A járművek megfelelő mozgásának megvalósítása két lépésből áll: először meg kell oldani, hogy irányítható legyen az autó, másodszor pedig, hogy önállóan tudjon mozogni az autó a megfelelő irányba. Ezekre a Youtube-on találtam megoldásokat, amiket átalakítottam, hogy megfelelő legyen. Az elsőt egy *videó[26]* alapján a másikat pedig egy *videósorozat[27]* alapján oldottam meg. Ezeket a következő két Script-ben valósítottam meg:

7.3.1 CarMovement

Ebben az osztályban van egy Vector2 típusú változó: *movement*. Ennek a változónak az értékeit adja meg a CarAI script. Az X komponense ennek a változónak az, hogy mennyire kell elforduljon az autó. Ezt az értéket a CarAI -1 és 1 között állítja be. Ezt át kell alakítani, hogy a -1 legyen a balra kanyarodás, a +1 a jobbra kanyarodás. Ezt meg lehet azzal oldani, hogy a maximum kanyarodási szöggel beszorozzuk. Ez álltalában 30º szokott lenni, de az elkészült játékban 40º-ot használok, hogy az autók képesek legyenek egy ívű megfordulást csinálni. A *movement* Y értéke 0 és 1 közötti (de lehetne kisebb, mint 0, de a CarAI minimum 0-s értéket állít be). Ez az érték határozza meg a sebességét az autónak.

Az autó mozgatása alapvetően 3 lépésből áll: autó elmozgatása: HandleMotor, autó kanyarodása: HandleSteering és kerekek animálása: HandleWheels. Ez a három függvény minden autónál meghívódik a FixedUpdate függvényén belül (másodpercenként 50x frissül).

7.3.1.1 HandleMotor

A WheelCollider-nek van motorTorque értéke, amit be lehet állítani a *movement*.y * *power*-re. A *power* az egy konstans, ami a motor erősségének felel meg. A motorTorque érték beállításával lehet meghatározni, hogy milyen gyorsan forogjanak a kerekek. Ez a forgás viszont a kerekeknek nem fogja változtatni a kinézetét, ez csak az adott WheelCollider-t forgatja meg. Ezt csak a két első keréknél változtatom, ezzel első kerék meghajtású lesz az összes autó. Ugyanitt kezelem azt is, hogy az autó álljon meg. Ezt úgy, hogy amikor a *movement*.y 0 vagy annál kisebb, akkor a WheelCollider brakeTorque értékét 1000-re állítom be. Ezt azért kell csinálni, mivel egyébként az autók csak motorféket használnának, és nem állnának meg időben a kereszteződések előtt.

7.3.1.2 HandleSteering

Itt állítom be, hogy a WheelCollider a jó irányba nézzen a *movement*.x érték alapján. Ezt is csak az első két keréknél kell megcsinálni, mivel a valóságban is az autóknak csak az első kerekei forognak a kanyarodás irányába. Ebben a függvényben a WheelCollider steerAngle értékét kell csak állítani.

7.3.1.3 HandleWheels

A három függvény közül ez az egyetlen, ami valamivel komplikáltabb, de ez se annyira összetett: ebben a függvényben mind a négy keréknek a WheelCollider *position* és *rotation* értékek alapján kell beállítani a kerekek kinézetét. Ezt úgy oldottam meg, hogy a WheelCollider-nek van egy olyan metódusa, hogy *GetWorldPose* ami pont a szükséges értékeket adja vissza. Ennek a függvénynek a használatával könnyen beállíthatóak a szükséges értékek.

7.3.2 CarAI

Ez az a Script, ami irányítja az autót. A SetPath függvény állítja be rajta a már kiszámolt *path*-ot. Kiszámolja, hogy eredetileg milyen irányba kell nézzen az autó amikor létrejön, és ezt be is állítja rajta. Ezt a *transform.InverseTransformPoint* függvénnyel csinálja. Ez a függvény visszaad egy Vector3 változót, amiben a *path* lista második elemének a relatív pozíciója van a Scriptet tartalmazó GameObject pozíciójához képest (ez az első pont a *path* listában, ami egyben az a pont, ahol létrejön az autó). Ebből az irány kiszámítható az Atan2 függvénnyel. Ami ennek a függvények az eredménye, arra kell beállítani az autónak a *transform.rotation* értékét.

A Script Update függvényében három függvény kerül meghívásra minden alkalommal: CheckIfArrived, Drive, CheckCollision. Ezek a függvények a következőkért felelősek:

7.3.2.1 CheckIfArrived

Ez hívódik meg először. Ha az autó nincs megállva, akkor megnézi, hogy az autó jelenlegi pozíciója és a *path* jelenlegi eleme között a távolság az kisebb-e, mint a konstans érték, ami be van állítva, ha nem kisebb, akkor nem csinál ez a függvény semmit, ellenkező esetben a *path* következő elemét állítja be célnak. Ha nem létezik a következő elem (a path lista végére ért), akkor Destroy-olja az autót.

7.3.2.2 Drive

Ez az a függvény, ami állítja a CarMovement-ben szereplő movement értékeit. Ha a *stop* (ez akkor igaz, ha előtte van egy autó, vagy ha kereszteződésben várakoznia kell) értéke az autónak igaz, akkor 0,0-ra állítja be a movement-et. Egyébként ugyanúgy meghatározza az irányt, amiben van a következő pont, mint a kezdeti lépésben (a *transform.InverseTransformPoint* és *Atan2* függvényekkel). Ha ez az érték túl nagy vagy túl kicsi (ha a következő pont nem az autó előtt van), akkor a *movement.x* értéket -1 vagy +1-re állítja be, egyébként pedig az érték az -1 és +1 közötti lesz. Ha a *stop* értéke hamis, akkor a *movement.y* értékét 1-re állítja be.

7.3.2.3 CheckCollision

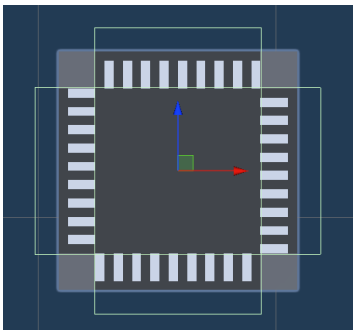
Ez függvény a *Phisics.RayCast* függvénnyel egy sugarat indít az autó orrából abba az irányba, amelyik irányba néz az autó. Ha ez egy másik autó objektumot érint, akkor megállítja az autót. Ennek köszönhetően nem ütik el egymást az autók, ha még mielőtt az autó beérne a kereszteződésbe már várakozik az autó sávjában egy másik autó.

7.4 Kereszteződés

A kereszteződést két lépésben valósítottam meg: először csak az autókat vettem figyelembe a kereszteződéseknél, majd miután ez megfelelően működött egy kicsit módosítva a kódon meg lett oldva, hogy a gyalogosok és az autók is megálljanak, és a helyes sorrendben menjenek át a kereszteződésen.

7.4.1 Autók a kereszteződésben

A kereszteződés megvalósításához az intersect3 és intersect4 Prefab-eket kellett módosítanom. Ahhoz, hogy a kereszteződésben lévő autókat érzékelni lehessen mind a kettőhöz hozzá kellett adni 2 BoxCollider-t:



10. ábra: Intersect4 BoxCollider-ek

A 11. ábrán a zöld körvonalas téglalapok az új BoxCollider-ek. Ugyanezek a BoxCollider-ek lettek az intersect3-nál is használva. Azért kell, hogy túl nyúljon a Prefab szélein egy másik út elemre, mivel, ha nem lógna túl, akkor az autóknak egyből meg kellene állniuk, ami kevésbé lenne élethű.

Ezután a SmartIntersection Script-ben oldottam meg a kereszteződés logikáját. A kereszteződésben lévő autókat egy Queue-ban tárolom. Ez egy speciális lista, aminek két függvényét használom: *Enqueue* és *Dequeue*. Az előbbi a Queue végére rakja be az új elemet, a *Dequeue* pedig kiveszi az elejéről az első elemet. Így a kereszteződés az egy FIFO (First In First Out) lesz. Ezen kívül külön eltárolom azt az autót, ami jelenleg mozog a kereszteződésben, ez lesz a *currentCar*. Ez nem lesz eleme a queue-nak.

Unity-ben van olyan függvény, hogy onTriggerEnter és onTriggerExit. Ezeket felhasználva lehet érzékelni, ha egy autó érkezik vagy távozik a kereszteződésből. Ezeknek a függvényeknek van egy bemenő paraméterük (ez az a GameObject, ami bement vagy kiment a kereszteződésből).

Az onTriggerEnter függvényben először megnézi az algoritmus, hogy autó-e. Ezt úgy oldottam meg, hogy Unity-ben a GameObject-eknek be lehet állítani *tag*-et, ezért az autó Prefab-eken ezt beállítottam „Car” -ra, és, ha egy GameObject a kereszteződésbe ér (lehet ez a játékos vagy gyalogos is), akkor megnézi, hogy a *tag*, ami rajta be van állítva az a „Car” -e, és ha igen, akkor egy autó ért a kereszteződésbe. Ha az autó, ami a kereszteződésbe érkezik az még nem szerepel a *queue*-ban (ez azért fontos, mert az autók nem csak egy komponensből állnak, ezért, ha a kereszteződésbe érnek, akkor mivel a szülőn van beállítva a *tag*, ezért a többi is megörököli ezt a *tag*-et, és minden komponens bekerülne a *queue*-ba) és ha ez nem a jelenleg a kereszteződésen áthaladó autó, akkor berakja a *queue*-ba, és az autónak a stop értékét igazra állítja, ezzel megállítva azt.

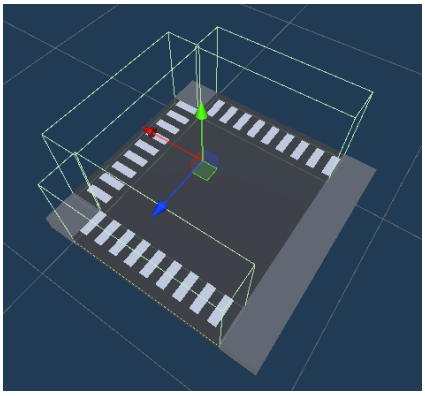
Az Update függvény megnézi, hogy a *currentCar* az *null*-e (ha null, akkor azt jelenti, hogy jelenleg egy autó sincs a kereszteződésben), ha az, akkor a *queue* elejéről kiveszi az első autót, és ez lesz a *currentCar*. Ezután ennek az autónak a stop értékét hamisra állítja.

Az onTriggerExit ugyanúgy megnézi, hogy autó-e a kimenő GameObject, mint ahogy az onTriggerEnter nézte meg. Ha autó, akkor a *currentCar* értékét beállítja *null*-ra, és így a következő update-ben a *queue* lején lévő autó el fog tudni indulni.

7.4.2 Gyalogosok a kereszteződésben

A gyalogosok, játékosok és autók találkozását a kereszteződésekben úgy terveztem, hogy ha egy gyalogos érkezik a kereszteződésbe, akkor, ha épp van egy autó már a kereszteződésben, akkor azt várja meg, ameddig elhagyja, majd induljon el. Ameddig gyalogosok vannak a kereszteződésben, addig egy autó se hajthat be a kereszteződésbe. Ha a játékos tartózkodik a gyalogos átkelőhelyen, akkor a gyalogosok szabadon mozoghatnak, viszont az autók ne hajtsanak be a kereszteződésbe.

Ahhoz, hogy érzékelni lehessen, hogy egy gyalogos a kereszteződésbe érkezett, létrehoztam az intersect3 és intersect4 típusú utakon egy-egy új empty GameObject-et. Ezekhez hozzáadtam annyi BoxCollider-t, amennyi gyalogos átkelő található az adott út típuson:



11. ábra: gyalogos átkelőhely BoxCollider-ek

A gyalogosok kezeléséhez létre kellett hoznom egy külön Script-et: SmartCrosswalk. Ennél is az onTriggerEnter és az onTriggerExit függvényeket kellett használni a gyalogosok (és a játékos) érzékelésére, de itt az autók észlelésénél használt *tag*-es megoldás helyett sokkal optimálisabban határozom meg a belépő és kilépő GameObject-ek típusát: a belépő és kilépő GameObject-nek lekérem a PedestrianMovement komponensét, ezt eltárolom a *pedestrian* változóban, és a PlayerMotor komponensét is lekérem (Mivel PedestrianMovement komponense csak a gyalogosoknak van, PlayerMotor komponense pedig csak a játékosnak van, ezért a kettő közül legalább az egyik *null*, és ebből meghatározható, hogy gyalogos, játékos vagy más lépett-e a gyalogos átkelőhelyre). Ez a megoldás itt azért jobb, mint a *tag*-es az autók esetében, mert így változóban el tudom tárolni a PedestrianMovement-et, amiben már meg lettek valósítva a gyalogost megállító és elindító függvények a gyalogos és játékos interakciója részben. Ezeket a függvényeket itt újra fel lehet használni.

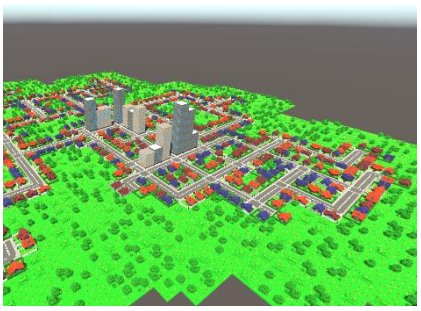
Az onTriggerEnter-nél amikor egy gyalogos érkezik a gyalogos átkelőhelyre, akkor megáll a gyalogos, és a PedestrianMovement komponensét eltárolom egy listában, hogy majd el lehessen indítani egyszerűen az összes gyalogost egy for loop segítségével.

A SmartIntersection Script-ben létre kellett hozni egy változót: *pedestrianWiting*. Ennek az értékét a SmartCrosswalk Script beállítja igazra, ha egy gyalogos vagy játékos belépett az átkelőhelyre, ha az utolsó gyalogos is kilépett, és a játékos sincs benn, akkor hamisra állítja. Ezt a változót fel kell használni a SmartIntersection Update függvényében: akkor veszi az első autót a *queue*-ból, ha a pedestrianWaiting értéke hamis, vagyis nem várakozik egy gyalogos se. Ugyanebben a függvényben, ha a *queue* üres és a *pedestrianWaiting* igaz, akkor az összes gyalogos, aki várakozik az elindul.

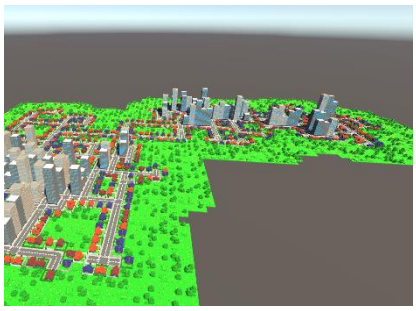
Az onTriggerExit-ben, ha a játékos az a GameObject az, amelyik kilépett, akkor elindulnak az autók, ha gyalogos lépett ki, akkor eltávolítja a gyalogosok listájából ezt a gyalogost, és ha nincs több gyalogos ezután a listában, és a játékos sincs a gyalogos átkelőhelyen, akkor elindulhatnak az autók.

8 Eredmények

8.1 Város generálás eredmények

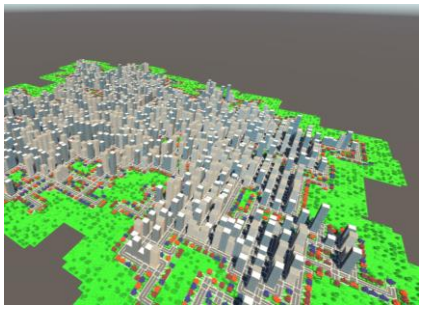


12. ábra: L-rendszer 7-es mélység, randomize bekapcsolva



13. ábra: L-rendszer 10-es mélység, randomize bekapcsolva

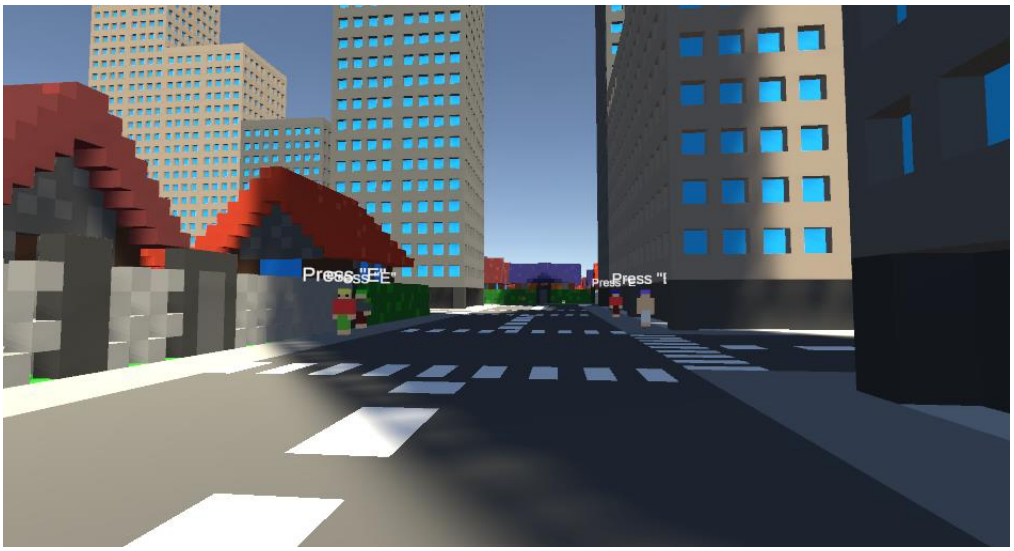
14. ábra: L-rendszer 7-es mélység, randomize kikapcsolva



15. ábra: L-rendszer 10-es mélység, randomize kikapcsolva

Jól látszódik, hogy amikor a randomize ki volt kapcsolva, akkor sokkal nagyobb kiterjedésű város lett legenerálva. Továbbá amikor nagyobb mélységig lett bejárva az algoritmus, akkor is nagyobb kiterjedésű lett a város. A 13. ábrán szereplő város úgy néz ki, mint ami még csak most kezd modernizálódni pár darab irodaépülettel a központjában. A 14. ábrán lévő város olyan látszatot kelt, mintha két város egybe nőtt volna, és ahol találkoznak, ott kertvárosi rész alakult volna ki. A 15. ábrán a város szerintem a legkevésbé élethű, mivel nagyrészt kertváros van, és teljesen véletlenszerűen vannak irodaépületek, amik nem igazán illenek így össze. Végül a 16. ábrán szereplő a legnagyobb, és ebből kifolyólag a legtovább tartott ennek a legenerálása, de ez hasonlít a legjobban szerintem egy valós modern városra.

8.2 Gyalogosok generálása eredmények



16. ábra: különböző kinézetű gyalogosok közlekednek, fejük felett a „ Press „E” ” felirattal

17. ábra: egy városhoz generált NavMesh

Egy generált városnak a NavMesh felületei láthatóak (Walkable - kék), ezen túl az utakon elhelyezett Obstacle-ből kialakult hálózat. Sikeresen le lett fedve az utaknál a nem járható rész és jól látszik kékkel az összes járda és zebra, amik járhatóak. A beépített útvonaltervező A* algoritmust használva hatékony az útvonalkeresés. A NavMeshComponents package is jól kezelhető, azonban a CalculateTriangulation költséges művelet, így sok gyalogosnál több időt vesz igénybe az útvonal kiszámítása és pontok keresése.

8.3 Gyalogosokkal kommunikáció eredmények

18. ábra: Samuel Bennett-el beszélgetés

19. ábra: Victor Sinclair-rel beszélgetés

Amint látható a két NPC ugyanarra a kérdésre teljesen más választ adott, Samuel inkább arról beszélt, hogy milyen napja van: hogy ivott egy nagyon rossz kávét, míg Victor a hobbijairól beszélt.

8.4 Autók mozgása eredmények

20. ábra: Autók és gyalogosok közlekednek a városban

Képpel nem lehet rendesen szemléltetni, hogy az autók miként mozognak, viszont látszódik, hogy különböző kinézetű autók jöttek létre. Minden autó az úttest jobb oldalán közlekedik. A jobb felső sarokban lévő kereszteződésben várakoznak egymásra az autók. Továbbá látszódik egy olyan kereszteződés is, amelyben az autók várják, hogy a gyalogosok elhagyják a gyalogos átkelőhelyeket, hogy ők is mehessenek. Így az elvárásoknak megfelelően közlekednek az autók.

9 Továbbfejlesztési lehetőségek

9.1 Több fajta épület

Lehetne belerakni több fajta magas épületet, meg kertes házat. Ezen a kettőn kívül még több épület típust is megvalósítható lenne. Lehetne olyan épületeket belerakni, amik nem 1*1-es méretűek olyanokat, amikből van megadva minimum vagy maximum darabszám, amennyi kell legyen a városban. Olyan épületekkel is kibővíthető a játék amiknek van valamilyen céljuk, például egy bevásárlóközpontot, ahol megállnak a gyalogosok és az autók, bemennek, majd kijönnek és hazamennek.

9.2 Különleges járművek

Lehetnének különböző járművek, például: tűzoltó, mentő, rendőr, akik miatt a többi autó félre kell álljon. Lehetne busz is, ami több helyen megáll és szállnak fel és le róla gyalogosok.

Ezen kívül a már létrejött autókba beleülhetne a játékos valamilyen animációval, vagy valahogyan meg idézhetne a játékos akármilyen autót, amiket tud irányítani. Esetleg más típusú járműveket is lehetne létrehozni, például repülőket, helikoptereket.

Az autók működésén is lehetne fejleszteni, egy valósághű KRESZ rendszer megalkotásával. Például, hogy a kereszteződésbe bemehessen egyszerre két autó, amiknek az útvonala nem metszik egymáséit. A jelenlegi FIFO helyett lehetne megvalósítani jobbkéz szabályt, esetleg közlekedési lámpákat. Ezt utána tovább lehetne fejleszteni, hogy a közlekedési lámpák úgy váltakozzanak, hogy a dugó minél kisebb legyen, ezzel egy okos várost alkotva.

9.3 Valósághűbb játékélmény

Jelenleg a város síkban generálódik le, de megvalósítható lehetne, hogy esetleg a Perlin-zajt használva legyen domborzat is beépítve a város generálásba. Vagy felhasználó által rajzolt domborzat is valamilyen formában implementálható lehetne.

Lehetne állatokat, idővel változó növényzetet, napszakokat, esetleg évszakokat belerakni, amikben a város kinézete is megváltozik, benne az ég is, a gyalogosok más ruhákban járhatnának.

Továbbá hangokkal is valósághűbbé lehetne tenni a várost. Az autóknak motorhangot adni, a gyalogosoknak valamilyen egyszerű *Sims*[28] típusú nem emberi nyelvet, de annak hangzót. A gyalogos és játékos sétálásának is lehetne hangja. Ezen kívül környezeti hangot is lehetne, például néha szél hang, esőben eső hang.

9.4 Összes NPC különböző karakterrel

Jelenleg csak 9 különböző karaktert hoztam létre, és ezek vannak véletlenszerűen hozzárendelve a gyalogosokhoz, de megoldható lehetne, hogy minden gyalogosnak teljesen egyedi legyen a személyisége. Mélyebb személyiségeket is ki lehetne találni a személyeknek, mindegyiknek összetett hátteret lehetne írni, a közös tudást lehetne bővíteni, és esetleg a gyalogosok mozoghatnának a személyiségüknek megfelelően, például egy vidám gyalogos gyorsabban sétáljon, mint egy szomorú.

9.5 Különleges események

Lehetnének a városban véletlenszerűen különböző események: például az egyik épület kigyullad, ezért egy tűzoltóautó odamegy a lehető legrövidebb úton, kerülgetve a többi autót, úgy, hogy azok félre állnak, és amikor megérkezik a tűzoltó a helyszínre akkor tűzoltó NPC-k szállnak ki és eloltják a tüzet és utána elmennek. Vagy ezt tovább fejlesztve lehetne az, hogy egy teljes története legyen a játéknak, vagy esetleg több különálló véletlenszerű története.

Irodalomjegyzék

[1] Stefan Greuter, Jeremy Parker, Nigel Stewart, Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities.

[2] Niclas Olsson, Elias Frank. Procedura city generation using perlin noise: http://www.diva-portal.org/smash/get/diva2:1119094/FULLTEXT02.pdf

[3] L-system: https://en.wikipedia.org/wiki/L-system

[4] Pascal Müller. Procedural modelling of cities. SIGGRAPH01

[5] Unreal Engine: https://www.unrealengine.com/en-US

[6] Asset Store: https://assetstore.unity.com/

[7] GameObject: https://docs.unity3d.com/ScriptReference/GameObject.html

[8] Prefabs: https://docs.unity3d.com/Manual/Prefabs.html

[9] Rigidbody: https://docs.unity3d.com/ScriptReference/Rigidbody.html

[10] Collider: https://docs.unity3d.com/ScriptReference/Collider.html

[11] Scripting: https://docs.unity3d.com/Manual/ScriptingSection.html

[12] Canvas: https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html

[13] CharacterContrtoller: https://docs.unity3d.com/ScriptReference/CharacterController.html

[14] Phisics.Raycast: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

[15] Procedural Town: https://www.youtube.com/playlist?list=PLcRSafycjWFcbaI8Dzab9sTy5cAQzLHoy

[16] Dragon curve: https://en.wikipedia.org/wiki/Dragon_curve

[17] MagicaVoxel: https://ephtracy.github.io/

[18] NavMeshComponents: https://github.com/Unity-Technologies/NavMeshComponents

[19] AI Navigation package: https://docs.unity3d.com/Packages/com.unity.ai.navigation@2.0/manual/index.html

[20] NavMesh: https://docs.unity3d.com/ScriptReference/AI.NavMesh.html

[21] NavMesh Surface: https://docs.unity3d.com/560/Documentation/Manual/class-NavMeshSurface.html

[22] NavMesh Obstacle: https://docs.unity3d.com/530/Documentation/Manual/class-NavMeshObstacle.html

[23] A* algorithm: https://en.wikipedia.org/wiki/A*_search_algorithm

[24] InworldAI: https://www.inworld.ai/

[25] Dijkstra-algoritmus: https://hu.wikipedia.org/wiki/Dijkstra-algoritmus

[26] Simple Car Controller in Unity Tutorial: https://www.youtube.com/watch?v=Z4HA8zJhGEk&t=579s

[27] Traffic system in Unity 2020: https://www.youtube.com/playlist?list=PLcRSafycjWFdDou6OTCmGbRZ9lwLXjuMO

[28] The Sims: https://www.ea.com/games/the-sims

Függelék

Felhasznált programok

A szakdolgozatom elkészítése során a következő alkalmazásokat használtam:

- Unity: A játékmotor, aminek a használatával lett elkészítve. Az alkalmazás elkészítéséhez a Unity 2021.3.22f1-es verzióját használtam.
- Microsoft Visual Studio: A Unity-ben használt C# script-ek elkészítésére használtam.
- MagicaVoxel: A játékban szereplő modellek ennek a segítségével lettek megrajzolva.
- Microsoft Word: A szakdolgozat megírására használtam.

Felhasználói instrukciók

A játék indulásakor egyből legenerálódik egy város, utakkal, épületekkel, gyalogosokkal és autókkal. Az induláskor mindig ugyanazokkal a kezdeti paraméterekkel generálódik, de ugyanúgy véletlenszerűen, mint az újra generálásnál. A játékos a „W”, „A”, „S”, „D” billentyűkkel tud mozogni a városban, „Szóköz” billentyűvel tud ugrani, a „Shift” billentyű lenyomva tartásával tud sprintelni. Gyalogos közelében az „E” billentyű lenyomásával tud elkezdeni a játékos egy beszélgetést az adott gyalogossal. „Esc” billentyűt lenyomva jelenik meg a menü, amiben tud a játékos változtatni annak a városnak paraméterein, ami a GENERATE gomb lenyomásával jön létre. A RESUME gombot lenyomva tűnik el a menü. A QUIT gombbal lehet kilépni a játékból.

2. táblázat: Várossal kapcsolatos paraméterek jelentése

Paraméter neve	Paraméter jelentése	Minimum-Maximum érték
LSystem Depth	Az L-rendszer a város generálásakor milyen mélységig legyen bejárva (város mérete).	6 – 10, egész
Maximum Building Height	A felhőkarcolók maximális magassága.	1 – 100, egész
Minimum Building Height	A felhőkarcolól minimum magassága.	1 – 100, egész
Ignore Chance	Ekkora eséllyel hagy el az algoritmus utakat a városból.	0 – 1, lebegőpontos szám
Number Of Pedestrians	Gyalogosok száma	0 – 75, egész
Max Speed Of Pedestrians	Gyalogosok maximális sebessége	0.1 – 1, lebegőpontos szám
Max Number Of Cars	Autók maximális száma a városban.	1 – 1000, egész
Randomize	Hagyjon-e el véletlenszerűen utakat a városból az algoritmus.	Igaz - Hamis