



ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR



SZAKDOLGOZAT

**OE-NIK
2023**

Hallgató neve:
Hallgató törzskönyvi száma:

**Erdei Barnabás
T/006736/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Erdei Barnabás**
Törzskönyvi száma: T/006736/F112904/N
Neptun kódja: 

A dolgozat címe:

Procedurálisan generált városok
Procedural city generation

Intézményi konzulens: Dr. habil. Szénási Sándor
Külső konzulens:

Beadási határidő: 2022. december 15.

A záróvizsga tárgyai: Számítógép architektúrák
Szoftvertervezés és -fejlesztés
specializáció

A feladat

Tervezzen meg és implementáljon egy olyan alkalmazást, ami képes egy város részletes modelljének véletlen generálására az előre megadott paraméterek alapján! Ehhez vizsgálja meg a hasonló fejlesztéseket, illetve az azok által használt módszereket, technikákat, eszközöket! Ezek alapján tervezze meg az elkészítendő alkalmazást, válassza ki a használandó módszereket és algoritmusokat (LSystem, Voronoi diagram, Unity motor, stb.)!

Valósítsa meg a megtervezett rendszert a választott programozási nyelv és keretrendszerek segítségével! Az alkalmazás alkalmas legyen épületek, utak és egyéb forgalmi objektumok létrehozására! A nagyobb méretű városok elfogadható idő alatti generálás érdekében biztosítsa a megfelelő skálázhatóságot! Tesztekkel igazolja a program működését, ezek alapján értékelje az elkészült munkát!

A dolgozatnak tartalmaznia kell:

- a feladat leírását,
- a meglévő rendszerek bemutatását,
- az alkalmazott eszközök, technológiák és eljárások bemutatását,
- a megvalósítandó feladat tervét,
- a tesztelés folyamatát és a teszteredményeket,
- eredmények bemutatását és értékelését,
- az architektúrában rejlő továbbfejlesztési lehetőségeket,
- a dokumentációt.



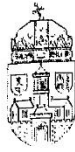
Dr. Vámosy Zoltán
intézetigazgató

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

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens



ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

HALLGATÓI NYILATKOZAT

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

Budapest, 2022. december 15.

Erdős Barnabás

hallgató aláírása



ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar
9. sz. melléklet

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Erdei Barnabás..... Neptun kód: [REDACTED] Tagozat: NIK.....
Telefon: [REDACTED] Levelezési cím (pl: lakcím): [REDACTED]

Szakdolgozat / Diplomamunka¹ címe magyarul:

Procedurálisan generált városok.....

Szakdolgozat / Diplomamunka² címe angolul:

Procedural city generation.....

Intézményi konzulens: Külső konzulens:

Dr. habil. Szénási Sándor

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	03.07.	Téma részletezése	[Signature]
2.	03.21.	Hasonló megoldások elemzése, célok, projekt méretének meghatározása	[Signature]
3.	04.11.	Megoldásra használható technikák közül a legmegfelelőbbek kiválasztása	[Signature]
4.	04.28.	Rendszerterv áttekintése	[Signature]

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplo-mamunka 2 / Diplomamunka 3 / Diplomamunka 4³ tantárgy követelményét teljesítette, beszámolóra / védésre⁴ bocsátható.

Budapest, 2022. 05. 09.....

[Signature]
.....
intézményi konzulens

¹ Megfelelő aláhúzendő!

² Megfelelő aláhúzendő!

³ Megfelelő aláhúzendő!

⁴ Megfelelő aláhúzendő!



KONZULTÁCIÓS NAPLÓ

Hallgató neve: Erdei Barnabás..... Neptun kód: [REDACTED]..... Tagozat: NIK.....
Telefon: [REDACTED]..... Levelezési cím (pl: lakcím): [REDACTED].....

Szakdolgozat / Diplomamunka¹ címe magyarul:

Procedurálisan generált városok.....

Szakdolgozat / Diplomamunka² címe angolul:

Procedural city generation.....

Intézményi konzulens: Dr. habil. Szénási Sándor..... Külső konzulens:

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	09.13.	Teendők részletezése	<i>Szénási Sándor</i>
2.	10.04.	További funkciók megoldásainak kutatása	<i>Szénási Sándor</i>
3.	11.08.	Fennakadást jelentő funkciókra alternatíva keresés	<i>Szénási Sándor</i>
4.	12.01.	Projekt értékelése	<i>Szénási Sándor</i>

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplo-mamunka 2 / Diplomamunka 3 / Diplomamunka 4³ tantárgy követelményét teljesítette, beszámolóra / védésre⁴ bocsátható.

Budapest, 2022. 12. 01.....

Szénási Sándor
.....
intézményi konzulens

¹ Megfelelő aláhúzendő!

² Megfelelő aláhúzendő!

³ Megfelelő aláhúzendő!

⁴ Megfelelő aláhúzendő!

TARTALOM

1. Bevezetés	1
1.1. Procedurális generálás lehetőségei	1
1.2. Projekt célja	2
2. Irodalomkutatás	3
2.1. Hasonló megoldások	3
2.1.1. Citygen	3
2.1.2. CityEngine	4
2.1.3. CityGen3D	4
2.1.4. CityCraft	5
3. Alkalmazott technikák	6
3.1. Adatstruktúra	6
3.1.1. Gráf	6
3.2. Úthálózat generálása	8
3.2.1. L-system	8
3.2.2. Voronoi diagram	9
3.3. Gráf javító algoritmusok	10
3.3.1. Snap algoritmus	10
3.3.2. Út metszéspontok javítása	11
3.4. Blokkok megkeresése és telekké darabolása	12
3.4.1. Blokkok keresése	12
3.4.2. Oriented bounding box subdivision	13
3.5. Épületek elhelyezése/generálása	14
3.5.1. Procedurálisan generált épületek	14
3.5.2. Felhasználó saját épület készlete	15
3.6. További dekorációs elemek elhelyezése, ötletek	16
4. Rendszerterv	17
4.1. Rétegek	17
4.1.1. Út generátor réteg	17
4.1.2. Hiba javító réteg	17
4.1.3. Blokk kiválasztó réteg	18
4.1.4. Épület réteg	18
4.1.5. Egyéb rétegek	18

4.1.6.	Konverziós réteg	18
4.2.	Felhasznált eszközök	18
4.2.1.	Programnyelv	18
4.2.2.	Megjelenítő motor	19
4.2.3.	Verzió követés	19
5.	Implementáció	20
5.1.	Projekt alapjainak lefektetése.....	20
5.1.1.	Unity projekt	20
5.1.2.	Gráf adatstruktúra	21
5.1.3.	Út generátor réteg	22
5.1.4.	Konvertáló réteg	24
5.1.5.	Proof of concept eredmények	24
5.2.	A projekt implementációja.....	25
5.2.1.	Rács adatstruktúra	25
5.2.2.	Utófeldolgozó algoritmusok.....	28
5.2.3.	Blokkok keresése.....	29
5.2.4.	Blokk telekké darabolása	31
5.2.5.	Épületek elhelyezése.....	33
5.2.6.	Megjelenítés.....	34
5.2.7.	Extra assetek	34
6.	Eredmények	35
7.	Továbbfejlesztési ötletek	38
8.	Összefoglalás	39
9.	Summary	40
	Irodalomjegyzék.....	41
	Ábrák jegyzéke.....	43

1. BEVEZETÉS

1.1. PROCEDURÁLIS GENERÁLÁS LEHETŐSÉGEI

Tartalom generálás játékokban, animált filmekben vagy szimulációkban mindig is egy hosszú, sok pénzt felemésztő folyamat. Ezek felgyorsítására véletlen tényezőket használó technikák jelentek meg. A növekvő elvárás, hogy minél részletesebb tartalom legyen készítve minél rövidebb idő alatt folyamatosan hajtja ennek a technológiának a fejlődését. Csak kézi munkával már nem lehetne kielégíteni a tartalmak mennyisége felé támasztott elvárásokat.

Példaként bemutatok pár gyakori felhasználási módot:

- Növényzetek modelljének generálása (1.1. ábra)
- Textúrák generálása (1.2. ábra)
- Tájak domborzatának elkészítése és növényzetének elhelyezése (1.3. ábra)
- Mesterséges tájak generálása, mint városok
- Játékokban felszerelés, feladatsorozatok, tereinek variációinak a növelése és elkészítése
- Animációk készítése
- Értelmes szövegek, cikkek generálása



1.1. ábra
Növény L-System-el[1]

1.2. ábra
Procedurálisan generált
textúra[2]

1.3. ábra
Procedurálisan generált
táj[3]

Fontos hozzátenni, hogy ezek technikák közel sem tökéletesek és szükséges még az eredmény kézi javítása. Továbbá nagyobb összefüggő elemek esetén ismétlődés, minta is található, vagyis nem tökéletesen véletlenszerű.

1.2. PROJEKT CÉLJA

A dolgozat procedurális generálásnak egy olyan részével foglalkozik, amely nem a természetes elemek generálásával foglalkozik, hanem egy mesterséges környezet, egy város modellezésével és elkészítésével foglalkozik. Egy várost pontosan modellezni nehéz feladat, mert számtalan olyan kicsi részletből áll mint, lámpaoszlopok, jelzőlámpák, zebrák, sorompók, metró, vasút, hidak és még lehetne sorolni további elemeket. Gyakran ezek az elemek nem is egy meghatározott szabályrendszer szerint lettek elhelyezve, aminek a hatását véletlenszerű tényezőkkel tudjuk szimulálni. Ezeken túl a városnak egy megfelelő alapot is szükséges biztosítani, ami az egész környezet vázát fogja meghatározni: az úthálózat. Az úthálózat talán a legösszetettebb része, ugyanis ez több úttípusból és mintából tevődik össze. Ezeken kívül szükség van még a telkek meghatározására és ezeken belül az épületek elhelyezésére.

Ezeknek az elemeknek a leírásához többféle procedurális generálásban használt technikát fogok elemezni és a feladatot figyelembe véve kiválasztani, hogy a végén egy hatékony és jól skálázható rendszert eredményezzen a projekt.

2. IRODALOMKUTATÁS

A rendszer tervezése előtt elemzésre kerültek a már megvalósított projektek, procedurális generálási technikák és egyéb segéd algoritmusok.

2.1. HASONLÓ MEGOLDÁSOK

Ezt a feladatot már sok projektben megoldották különböző módon és technikákkal, amik közül néhány a feladat kiírásra számára a legrelevánsabb projektek bemutatására kerülnek.

2.1.1. CITYGEN

Ez a Citygen [4] nevezetű szoftver projekt 2007-ben készült el és az akkori új hardverek által nyújtotta lehetőségeket próbálták kiaknázni úgy, hogy a jelenleg létező megoldásoknál egy kinézetre realiztikusabb úthálózatot generáljon. A problémát 3 folyamatra bontották fel:

- Fő útvonal generálás
- Másodlagos úthálózatok generálása a főútvonal által bezárt helyek között
- Épületek generálása

Az úthálózat gráf adatstruktúra segítségével kerül eltárolásra szomszédsági lista formájában, vagyis egy lista tartalmazta a csúcsokat és az ezekhez tartozó lista pedig a hozzájuk kapcsolódó éleket. Az utakat a világhoz képest adaptív módon hozták létre így megfelelően alkalmazkodott az útszakasz az adott domborzathoz. Az elemek domborzathoz igazítása többféle technikával kerül megvalósításra aszerint, hogy a domborzat meredeksége mennyire változik. Az utakat minták segítségével építették fel, aminek görbéje paraméterekkel testre szabható így képesek nem csak egyenes, hanem kanyarodó útszakaszok létrehozására is. A belső másodlagos úthálózat felépítésére L-System-et használtak. Az előre megadott szabályok segítségével lehetőséget ad különböző stílusú úthálózatok létrehozására. A várost a főutak cellákra osztják aminek a meghatározására a Minimum Cycle Basis(MCB) algoritmust használtak. Ezeket a cellákat különállóan lehet paraméterekkel testre szabni. Az út generálása után a Snap algoritmus az egymáshoz előre megadott távolságra lévő útszegmenseket kereszteződéssel egyesíti így kevesebb szakutca marad a hálózatban. Ezek után a másodlagos úthálózatból kiválasztásra kerülnek a telkek az MCB algoritmus segítségével. A kész telkeket addig darabolják, amíg egy cél méretet el nem ér mindegyik darab. Az épületek létrehozásakor 2 fő típust használtak, a belvárosi

épületek törekedtek a hely maximális kihasználására, míg a külvárosi épületek nem pontosan az utak mellett helyezkedtek el, hanem helyet hagytak kert számára is. Csak olyan épületek készülhettek el, amelyekből szabadon hozzáférhető maradt az út így a középén maradt blokkok szabadon maradtak. A kimenetben valós időben lehet navigálni és interaktívan szerkeszteni.

2.1.2. CITYENGINE

Ennek a szoftvernek [5] első verziója 2008-ban került bemutatásra és a mai napig fejlesztés alatt áll. A szoftver működése jelentősen különbözik a CityGen-től ugyanis sokkal több input adat alapján dolgozik. A folyamat megkezdése előtt a felhasználónak szükséges egy műholdas felvételtől kiválasztania a generálandó várost vagy annak egy részletét és utána lehetősége van a kép felbontását kiválasztania, nagyobb felbontású kép pontosabban generált városhoz vezethet, de több erőforrás szükséges a futás során. A szoftver az importálási folyamatot azzal segíti, hogy rengeteg széles körben használt adattípust támogat és használható például OpenStreetMap, Esri File Geodatabase stb. Miután a város elkészült azután a felhasználó szabadon beállíthat olyan paramétereket, amik a város kinézetét változtatják, mint épületek textúrái, modelljei vagy akár olyan részletek is beállíthatók, mint hogy az úton elhelyezett autók jobb vagy bal kéz szerinti közlekedésben jelenjenek meg. Az objektum modellek importálhatók más széleskörben használt modellező szoftverekből. Ezekon kívül még az út generálási szabályok is módosíthatóak. Mivel a szoftver nagyon sok felhasználói input adat alapján dolgozik ezért részletgazdag és testre szabható városok alkothatóak a segítségével, ami felhasználható akár régi történelmi városok létrehozására. Nem véletlenszerűen generál városokat, de az input adaton hasonló műveletek végez el, mint például az előző projektben bemutatott Snap algoritmus.

2.1.3. CITYGEN3D

Félreértések elkerülése érdekében kiemelem, hogy hasonlít a neve a CityGen [4] nevű szoftverhez, de a 2 projektnek csak témában van köze egymáshoz. A szoftver [6] nem egyedülálló ugyanis valójában a Unity játékmotor egyik kiegészítője, aminek első béta verziója 2018-ban, majd 2020-ban jelent meg a Unity Asset Store-ban az első stabil verziója. Működésben az előző fejezetben részletezett CityEngine-hez áll közelebb. Földrajzi adat alapján elkészül egy városnak az alapja és a felhasználó a Unity editor-ban adhat hozzá a városhoz épület és utcán található tárgyak modelljeit és ezek generálási szabályát állíthatja be. Részletes működésről szóló dokumentáció

viszont nem található erről a projektről csak a weboldalukon látható pár modulnak a működését, megvalósításukról viszont nincs szó.

2.1.4. CITYCRAFT

CityGen-hez hasonló céllal földrajzi adatok felhasználása nélkül készít el egy várost. A CityCraft [7] egyetemi projekt 2020-ban készült el. Legfontosabb technikák, amelyeket kiemelésre kerülnek a projektjükből, hogy az útgeneráláshoz a realisztikus utak érdekében egy „Agent” alapú generálást alkalmaztak. A futás során több „Agent” elindul a pályán és az egyetlen céljuk az úthálózat elkészítése egy előre definiált szabály alapján. A stratégiák, amiket használhattak előre meg voltak írva és némelyik stratégia része volt a stratégia váltás is, ami további úthálózat variációk létrehozását segítette. Továbbá képesek voltak osztódásra is vagyis egy „Agent” képes volt több „Agent”-et elindítani futása közben. Élettartamuk a stratégiájuk konfigurációjában van meghatározva. A skálázás érdekében az olyan algoritmusok miatt, amelyek igénylik a szomszédos elemek használatát gyakran például Snap algoritmus egy R-tree nevezetű adatszerkezet lett létrehozva, amely gyorsan képes visszaadni a térben szomszédos elemeket. Ezen kívül a körök keresésére is egy egyszerűbb megoldást nyújtottak az MCB algoritmusnál.

3. ALKALMAZOTT TECHNIKÁK

Ebben a fejezetben részletesen elemzésre kerülnek azok a technikák, amelyek a kutatás során több projektben is alkalmazásra kerültek. Ezek közül kerülnek kiválasztásra a projekt számára legmegfelelőbb algoritmusok miközben előnyeik és hátrányaik mérlegelésre kerülnek.

3.1. ADATSTRUKTÚRA

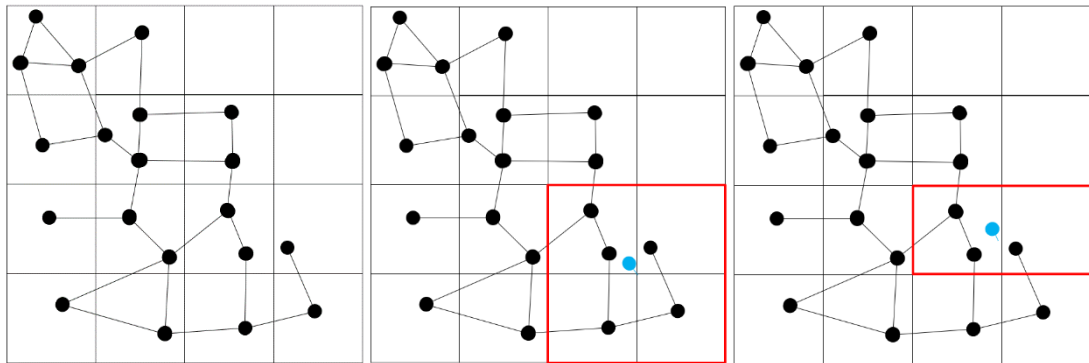
A projekt alapját képező struktúrának hatékonynak kell lennie, ugyanis az összes adatmanipuláció ezen keresztül fog történni és a teljesítmény is ettől fog függeni jelentős súlyban.

3.1.1. GRÁF

A korábban elemzett projektek döntő többségében a gráf adatstruktúra került implementálásra, mert az úthálózat csomópontjait és a hozzájuk tartozó útszakaszokat áttekinthető módon képes ábrázolni. Ennek pontosabban a szomszédsági lista ábrázolási módjával lehetett találkozni, ahol egy csúcs, ami az ebben a helyzetben az utak kereszteződésének a helyét jelöli és egy ehhez tartozó listával rendelkezik, ahol a hozzá csatlakozó útszakaszok referenciája szerepel. A projektek ennél több információval nem szolgáltak az adatstruktúrával kapcsolatban. Ez azért jelent problémát, mert a sima gráf implementáció ahogy nő úgy elkezd lassulni a növekvő elemszám miatt. Ennek következménye, hogy számos algoritmus, aminek elég lenne csak a szomszédos elemekkel dolgoznia kénytelen bejárni a teljes gráfot, hogy megtalálja a keresett elemet. Például egy adott sugárban elhelyezkedő legközelebbi elemek felderítése egy folyamatosan növekvő számítási igényű folyamat lenne, ami a skálázhatóságot ellehetetlenítené hosszú távon és ez ennek a struktúrának a legnagyobb hátránya.

Részletek hiánya miatt megtervezésre került egy skálázható gráf implementáció, ami a később tárgyalt algoritmusoknak megfelelően lett optimalizálva. Az adatstruktúra legnagyobb problémája a kezelhető méretű részegységek hiánya, ezért elsősorban ezt a problémát kellett megoldani. Ehhez ihletet egy népszerű játék a Minecraft szolgáltatott. A játék egy látszólag végtelen procedurálisan generált játékvilággal rendelkezik. A megvalósításához szükséges volt egy struktúra, amivel a világ részegységei felbontásra kerülnek. Ennek eredménye, hogy a játék futása és

betöltése jelentős mértékben felgyorsult. Ezeket a részegységeket chunk-nak [8] hívják.



3.1. ábra
Rácsos gráf

3.2. ábra
Sarokban található elem esetén
betöltött terület

3.3. ábra
Szélen található elem esetén
betöltött terület

Ehhez a rendszerhez hasonlóan rácsos szerkezet kerül kialakításra, ahol minden rács egy szabályos négyzet alakban helyezkedik el és a teljes gráfnak egy darabjával rendelkezik (3.1. ábra). Minden rács fix méretű, ami a generálás elején paraméterrel megadható kézzel, vagy automatikusan kiszámításra kerül a generálási adatokban megadott útszakasz hossza alapján így a rácsban található elemek száma közel megegyezik. Az adatstruktúra inicializáláskor a választott kezdőponton létrehoz egy rácsot és annak az összesen 8 szomszédját (átlókat beleértve) létrehozza. Ezek után, ha a hozzáadott elem kívül esne a rácson akkor lazy loading szerűen létrejön a rács, majd utána elhelyezésre kerül az új elem benne. Viszont érdemes szót ejteni a kivételekről is. A legtöbb algoritmus, amit a generált úthálózat szűrése és szépítése során felhasználásra kerülnek azoknak a helyes működéshez szüksége van az összes szomszédos elemre nem csak azokra a szomszédos elemekre, amik a rács határon belül esnek, vagyis ebben az állapotban a rácshatárok mentén hibás eredményekkel térne vissza minden hasonlóan működő algoritmus. Ennek a problémának a kiküszöbölésére a metódus, amivel a rács elemei visszaadásra kerülnek egy olyan logikával kell rendelkezzen, ami képes az algoritmus által jelenleg felhasznált elem alapján eldönteni, hogy kell-e szomszédos rácsokat ideiglenesen hozzácsatolni a jelenlegi rácshoz és ha kell akkor ezek közül melyeket szükséges. Egyszerű logikával akkor a határ menti elemek vizsgálatakor mindig hozzácsatolásra kerül minden szomszédos rács tartalma, de ez felesleges számításokkal jár. Ha sarok pontban van a kérdéses elem akkor a sarokkal szomszédos 3 rácsot szükséges csatolni (3.2. ábra), a többi esetben elég csak a megfelelő oldallal szomszédos 1 rácsot csatolni (3.3. ábra) vagy ha középen helyezkedik el akkor elég csak a jelenlegi rács elemeit feldolgozni. Ezen kívül van egy

másik jelentős kivétel is, ami akkor jelentkezik amikor élek kerülnek létrehozásra a csúcsok között. Kezelni kell a határokon túl nyúló éleket, amelyeket nem lehet a másik rács ismerete nélkül létrehozni ezért hozzáadáskor szükséges a kezdő rácsot és a cél rácsot csatolni így az összes szükséges ellenőrzés elvégezhető és létrejöhet egy új él. Legtöbb esetben használható lehetne az előző metódus, ahol már definiálásra kerültek a csatolási szabályok, de az nem kezeli azt az esetet amikor egy él átível egy rács felett teljesen, vagyis nem közvetlen szomszédról van szó ezért itt szükséges megnézni, hogy a cél elem melyik rácsban van majd azt betölteni. További optimalizálásként amikor egymáshoz csatolásra kerülnek a rácsok akkor nem egy új gráf kerül létrehozásra és egyesével másolásra kerülnek a referenciák, hanem az eredeti rácsok gráfjainak referenciái kerülnek visszaadásra egy gráf listaként így kevés iterációval gyorsan végrehajtható ez a művelet. Egyedüli hátránya, hogy erre a fogadó algoritmusnak is fel kell készülnie. Ennek a módosított gráf adatstruktúrájának a segítségével jól skálázható módon kerülnek tárolásra elemek, ahol a struktúra mérete nem fogja lassítani az utófeldolgozásban használt függvényeket, de cserébe egy olyan hátránnyal rendelkezik, hogy helytelen generálás esetén lehetséges, hogy egy rács jelentősen több adattal rendelkezik, mint egy másik, vagyis ott lassabb lesz a műveletvégzés. Megoldható lenne a kevésbé használt rácsok egyesítésével, de mivel az a felhasználásra kerülő úthálózat generálás technikákra viselkedésére nem jellemző ezért a teljesítményre nem jelent befolyást ebben a projektben.

3.2. ÚTHÁLÓZAT GENERÁLÁSA

A vizsgált projektekben döntően a következő 2 technikával lehet találkozni.

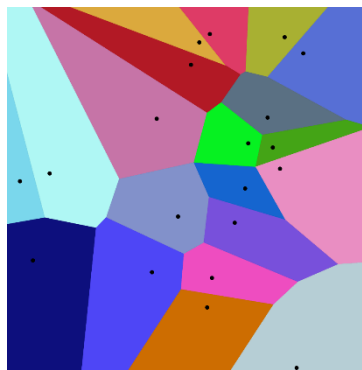
3.2.1. L-SYSTEM

Az L-system teljes nevén Lindenmayer system-et [6] Aristid Lindenmayer nevű magyar biológus fejlesztette ki. Eredetileg növény modellek készítése során került alkalmazásra és rekurzív természete miatt ahogy nő az iteráció szám annál komplexebb és exponenciálisan nagyobb növényeket eredményez. A generálás 4 paraméter segítségével történik. Az első paraméter tartalmazza a változó listát, amely tartalmazza azokat az elemeket, amik a mutáció során módosításra kerülhetnek. A másik listában pedig azok a konstansok, amelyek nem változnak a generáció során. Ezek a változók és konstansok lesznek kirajzolási utasítások. A harmadik paraméterben egy kezdő karakterlánc kerül meghatározásra, ami a generáció kezdő állapota lesz. Utolsó paraméterként a szabályok definiálására kerülnek, amik leírják, hogy a futás során melyik karaktert mivel kell helyettesíteni. Többszöri futás hatására

a karakter lánc exponenciálisan növekedni kezd és a kirajzolás során látszódik, hogy a kezdő állapotból az iterációk hatására egyre jobban szétágazik, több elemből fog állni. A kirajzolást egy „teknős” végzi, ami elkezd beolvasni az elkészült karakter sorozatot és a karakternek megfelelő előre definiált utasítást végrehajtja. Alap utasítások közé tartoznak olyanok, mint az előre haladás, forgás, pozíció mentése, pozíció visszatöltése, toll elvétele stb. Megfelelő paraméterezéssel és utasítás készlettel képes úthálózatok generálására, de mindig ugyan azt az eredményt eredményezi a fixen definiált szabályok miatt. Ennek a problémának a megoldásaképp felhasználásra kerülnek az utasításokban véletlen tényezők például változó mértékű előre menetel vagy véletlen szög méret így rendkívül nagy variációval fog rendelkezni az úthálózat, de érvénytelen utak létrehozását is okozhatja, ami utófeldolgozással szűrésre kerül. Komplexebb minták előállításához tovább lehet fejleszteni a parancsokat és nem csak egy karakterből állhat, hanem hozzá tartozó paramétereiből is így lehet az út típusok között is váltani vagy a fordulás szögét több helyen befolyásolni.

3.2.2. VORONOI DIAGRAM

A Voronoi diagram[9] segítségével generáló pontokkal partícionálásra kerül egy síkfelület, ahol egy cella úgy képződik, hogy melyik generáló ponthoz van a legközelebb a sík tartalma (3.4. ábra).



3.4. ábra
Voronoi diagram[9]

Ez a diagramm számtalan tudomány területen kerül felhasználásra például biológiai struktúrák modellezése segítségével, alkalmas repülőgépek vagy bármilyen más jármű számára a legközelebbi állomás meghatározására. Ezekon kívül felhasználásra kerül procedurális textúra generálásra is például kiszáradt föld és fa. Korábbi projekteken [10][11] a diagram felhasználásra került út generálás során, az egyik projektben tisztán csak ennek a segítségével generálódott út a másokban pedig a

CityGen projekthez hasonlóan ahol van egy fő út hálózat és az azok által bezárt részek a másodlagos utak. Itt a fő úthálózat generálásakor került használatra a Voronoi diagramm. Továbbá a távolság függvény módosításával és a generáló pontok mennyiségével befolyásolni lehet a kimenetet, amivel jobb lesz az úthálózat testreszabhatósága.

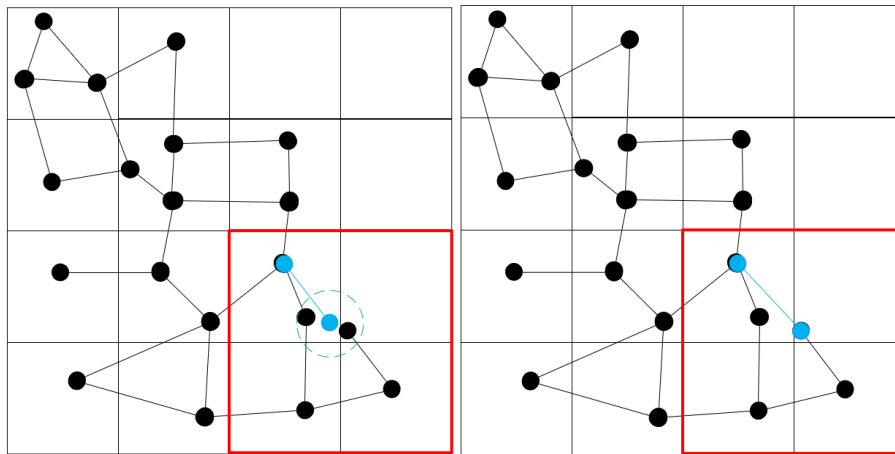
3.3. GRÁF JAVÍTÓ ALGORITMUSOK

Az úthálózat generálása közben nehezen biztosítható, hogy ténylegesen egy érvényes hálózat legyen az eredmény, de mi vehető érvényesnek? Egy olyan gráf szükséges, ahol nincs egymást metsző él ugyanis később a telkek létrehozásakor a gráfban körök kerülnek meghatározásra. A metsző élek egymást átfedő poligonokat eredményeznének és ugyan az a terület nem lehet egyszerre 2 vagy több telek része. A rendszer rugalmassága miatt bármilyen tetszőleges útfeldolgozó algoritmus használható a hibák kijavítására.

3.3.1. SNAP ALGORITMUS

Az elkészült nyers gráfban számtalan olyan alkalommal található olyan eset, ahol 2 kereszteződés feleslegesen közel kerül egymáshoz vagy akár átfedik egymást. A függvény 1 paraméterrel dolgozik, ami azt határozza meg, hogy két kereszteződés között mekkora minimális távolságnak kell lennie. Mielőtt jobban részletezésre kerül a „távolság” fogalmát szükséges tisztázni ebben a projektben. A város úthálózata egy térképhez hasonló módon kerül kezelésre így 2 dimenzió értelmezhető. A keresztezések 2 koordinátával rendelkeznek így és ebből kell a távolságot meghatározni. Ebben a térben több módon is értelmezhető a távolság, amelyek közül az egyik legnépszerűbb az Euklideszi távolság és a Manhattan távolság. Az előbbi kerül alkalmazásra egy kis módosítással, mivel elég csak az egymáshoz viszonyított távolságukat ismerni, de a pontos értéket nem ezért a gyökvonás elhagyásra kerül a gyorsabb számítás érdekében. Miután a távolság függvény tisztázva lett ezután minden csúcs a gráfban ellenőrzésre kerül, hogy mekkora távolságra van tőle az összes többi elem. Ha létezik a minimum megadott távolságnál kisebb távolságra lévő csúcs akkor a vizsgált csúcs egyesítésre kerül a legközelebb található ilyen csúccsal. Az egyesítés folyamata abból áll, hogy a vizsgált csúcs összes olyan éle átmásolásra kerül a legközelebbi csúcs szomszédsági listájába, amely még nem található meg benne. A másolás után a vizsgált elem törlésre kerül a gráfból. A teljes algoritmus végig iterálása egy költséges művelet nagy méretű gráfok esetén, de a rácsokra osztás miatt csak a

ténylegesen szomszédos elemek kerülnek ellenőrzésre így kevés iterációval képes a teljes gráfon elvégezni ezt a műveletet (3.5. ábra, 3.6. ábra).

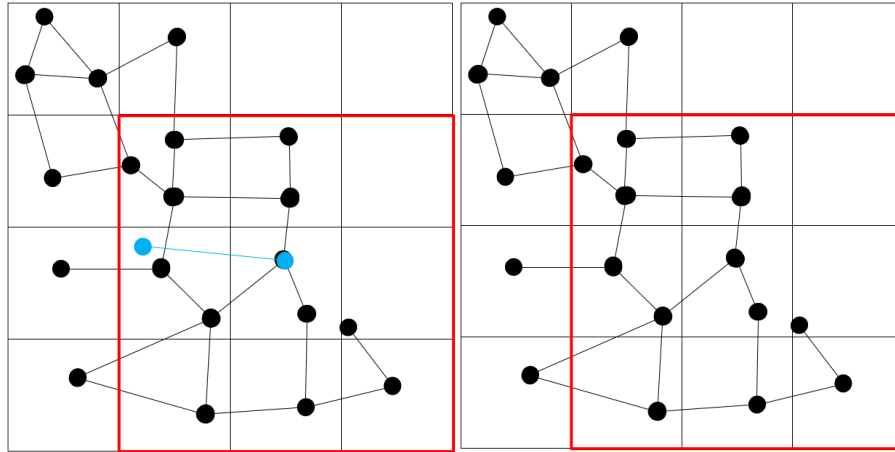


3.5. ábra
Snap algoritmus futás előtt, piros
kerettel jelölve a betöltendő rácsokat

3.6. ábra
Snap algoritmus futás után, piros
kerettel jelölve a betöltendő rácsokat

3.3.2. ÚT METSZÉSPONTOK JAVÍTÁSA

Meg kell találni azokat az éleket, amelyek egy másik élt metszenek és ki kell javítani őket, hogy a gráfban a kör kereső algoritmus helyes poligonokat adjon vissza. Ennek az első lépése az, hogy definiálni kell azt, hogy mikor metszi 2 él egymást. Erre a problémára az egyik megoldás a [12] oldalon található. Röviden az algoritmus a 2 él által alkotott 4 csúcs 3-as kombinációjának egy részének körüljárási irányának kiszámítása után 5 feltétel vizsgálata után döntést hoz, hogy 2 él metszi-e egymást. Egyetlen hibája, hogy az egymáshoz csatlakozó éleken is metszéspontot érzékel, de ez egyszerűen a végpontok ellenőrzésével szűrésre kerül. Ha detektálásra kerül egy ilyen él akkor több lehetőség is van a javításra. Az egyik opció, ami a komplexitást jelentősen növeli, ha a metszéspont helye meghatározásra kerül, majd oda egy csúcsot helyezünk el és a metsző élek 4 él felbontásaként adódnak hozzá. A másik sokkal egyszerűbb megoldással szimplán eltávolításra kerülnek ezek az élek, de itt egy olyan kivétellel kell számolni, hogy ha egy csúcsnak minden éle metszik egy másikat akkor egy elérhetetlen kereszteződéshez vezethet a szűrés. Mivel a csúcsok éleinek a száma gyorsan elérhető ezért egy szimpla kiegészítéssel ezeket a csúcsokat az éllel együtt törlésre kerülhetnek.



3.7. ábra
Hibás él futás előtt

3.8. ábra
Javított gráf

3.4. BLOKKOK MEGKERESÉSE ÉS TELEKKÉ DARABOLÁSA

Ebben a fejezetben az utak által körbezárt blokkok megkeresésének és ezeknek a feldarabolásának módszere kerül tárgyalásra.

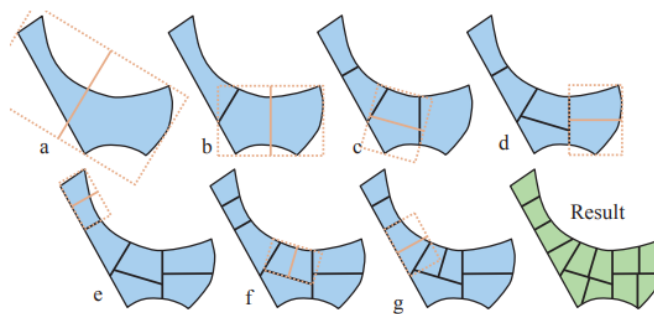
3.4.1. BLOKKOK KERESÉSE

A blokkok lesznek azok a területek, amelyeken a telkekké való darabolás történik és ezeken kerülnek elhelyezésre az épületek. A területeket a körbezáró kereszteződések segítségével határozhatóak meg. Mivel gráfok feldolgozásáról van szó ezért ez a probléma valójában a gráfban található minimális körök megkeresése. Egy hasonló projektben [13] a blokkok megfelelő meghatározása egy mélységi kereséssel történt, ahol minden csúcsban elindításra került egy egység, ami a kapcsolódó éleken való további egységek indításával addig folytatott keresést, amíg vissza nem kerül a kezdő pontba. Optimalizálás nélkül exponenciálisan nő a futási idő, így szükséges pár szabályt bevezetni. A legegyszerűbb ezek közül, ami az említett projektben felhasználásra került, hogy ha egy egység egy adott mélység után sem talál kört a gráfban akkor meg kell állítani a futását. Ez megakadályozza, hogy legrosszabb esetben az összes elemet végig ellenőrizze mielőtt eredményre jut. Továbbá, ha egy olyan csúcsra lépne rá, ami már fel lett korábban derítve és nem a kezdő pont akkor is el kell vetni a futását. Érdeemes megjegyezni, hogy a generálás indításakor a keresés mélységének maximum száma és a leghosszabb út mérete adott. Ezeket extra kikötésként felhasználva minden lépés után ellenőrizhető, hogy rendelkezésre áll-e még elég lépés, hogy a kezdő pontba vissza lehessen jutni, mert ha nem akkor szintén

szükségtelen a további keresés abban az ágban. Végrehajtás után több kört is eredményezhet a művelet. Ezek közül ki kell választani a legkisebb köröket, amelyek nem foglalnak magukban további köröket, amit az élek csúcsainak ellenőrzésével lehet kiszűrni. Az algoritmus megfelelő mélység megadása esetén nem túl költséges cserében potenciális nagyobb blokkokat kihagyhat és működéséből eredően zsákutcákkal rendelkező köröket nem képes körbejárni az eljárást.

3.4.2. ORIENTED BOUNDING BOX SUBDIVISION

A blokkok feldarabolására az egyik lehetőség az Oriented Bounding Box Subdivision [14] algoritmus alkalmazása.



3.9. ábra
Oriented bounding box subdivision[14]

Az algoritmus addig darabol egy adott blokkot, amíg egy felhasználó által meghatározott terület alatt nem lesz a mérete, vagyis ez is egy szintén rekurzív módon működő algoritmus (3.9. ábra). Elsőnek ki kell számolni a blokk legkisebb határoló négyszögét amit angolul „bounding box”-nak neveznek. A négyszög 4 csúcsának a koordinátája egyszerűen kiszámolható a blokk csúcsai koordinátájának a minimum és maximum értékeinek felhasználásával. Ezután a négyszög elfelezésre kerül egy olyan egyenessel, amely párhuzamos a rövidebb élével. Konvex esetben a blokk 2 élét fogja metszeni az egyenes. Ennél az esetnél a blokkok a metszéspontokat felezővonalra szerint a csúcsok 2 csoportba tartoznak, ami a 2 elvágott rész lesz és ezekhez még hozzá kell adni a 2 metszéspontot és kész a 2 blokk. Ekkor ellenőrizni kell a területét, hogy kell-e tovább darabolni. Mivel egy szabálytalan sokszög lesz a művelet eredménye ezért a terület számítása bonyolult feladat így a jelenlegi határoló négyszög területe alapján fog dönteni az algoritmus, mert nincs szükség akkora pontosságra. Ha viszont több mint 2 metszéspontunk van akkor a jelenleg darabolt rész egy konkáv alakzat. A felező vonal kezdő pozíciójától a metszéspontoktól számolt távolság segítségével sorba lehet távolság szerint rendezni a metszéspontokat. Erre azért van

szükség, mert az algoritmus így tudni fogja, hogy mikor lépett be a blokkba és mikor lépett ki és meghatározható a metszett élék segítségével, hogy a metszéspontok mely darabhoz fognak tartozni. A csoportosítás után ebben az esetben $N/2$ alakzat lehet a kimenet metszéspontok számától függően. A feldarabolt telkek létrehozása után a létrejövő új élék felhasználhatóak keskeny utcák, járdák generálására egy adott blokkon belül. Az algoritmus egyik legköltségesebb része a legkisebb bounding box meghatározása, mert poligon orientációtól függően nagyobb vagy kisebb területű négyzetet határoz meg. Ez közelíthető az adott poligon többszöri középpontos forgatás és újbóli négyzet kiszámítással, de precizitása és költsége függ a forgatásban használt mértékének a finomságától.

Az algoritmus átdolgozása után, ha a poligon leghosszabb élnek a felezőpontja kerül felhasználásra a felező vonal kiszámításakor, akkor a bounding box kiszámítása elhagyható. A kiinduló él felező pontján merőlegesen kivetített vonallal meghatározható egy azt metsző másik él a poligonban és azon generálás változatossága miatt egy véletlen érték kerül kiválasztásra, ami a felezővonal végét fogja jelenteni. További előnye még ennek a megoldásnak, hogy ha a kivetített vonal legközelebbi metsző éle kerül visszaadásra, akkor a futás mindig 2 poligont fog eredményezni és nem kell figyelni a konkáv és konvex okozta nehézségekre.

3.5. ÉPÜLETEK ELHELYEZÉSE/GENERÁLÁSA

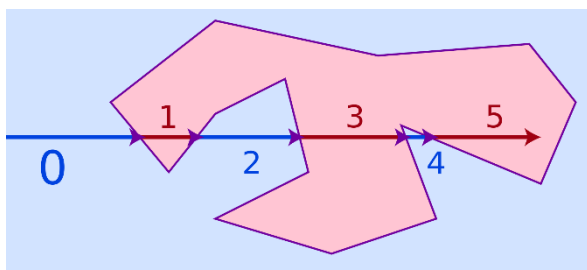
Ennél a résznél már felhasználásra kerül a megjelenítésre használt 3D motor által rendelkezésre álló lehetőségek közül néhány metódus például objektum létrehozása, méretének változtatása és a motor által támogatott 3D modellek importálása egy köztes konverziós réteg felhasználása segítségével.

3.5.1. PROCEDURÁLISAN GENERÁLT ÉPÜLETEK

Ha a felhasználó nem rendelkezik saját épület modellekkel akkor a [15]-ben részletezett projekt technikájához hasonlóan felhasználható a projektben útgeneráláshoz használt L-System más parancs készlettel. A parancs készlet a legegyszerűbb esetben csak 2 parancsból áll. Az első parancs a jelenlegi test magasságát fogja növelni. A másik parancs pedig az előző test határai felhasználásával létrehoz egy új objektumot, majd a méretének skálázását fogja befolyásolni. Ezzel a parancs készlettel egyszerű, változatos stílusú épületeket generálhatóak. Ha az első parancs nagy értékek közül választhat akkor felhőkarcoló szerű épületek jelennek meg. Az L-System tovább bonyolítható komplexebb épületek generálása érdekében.

3.5.2. FELHASZNÁLÓ SAJÁT ÉPÜLET KÉSZLETE

A népszerűbb 3D motorok támogatják a leggyakrabban használt modell formátumokat így csak az a feladat hárul a projektre, hogy a modell határainak és a telek határainak segítségével meg határozzunk, hogy elfér-e az adott modell a telekben egy véletlenszerűen meghatározott pozícióban.



3.10. ábra
Raycasting algoritmus X tengely irányban[16]

A véletlenszerű pontot az adott poligon bounding box határán belül kerül generálásra, de mivel a kiszámított négyzet nagyobb, mint az adott telek ezért szükséges a generált pont vizsgálata. Az ellenőrzésre a Raycasting algoritmus [16] jól használható, amely a kiszámolt bounding box felhasználásával egy X tengellyel párhuzamosan kivetített egyenes egyenletét számolja ki a határ kezdetétől a határ végéig amire a vizsgálandó pont illeszkedik. Az egyenes által metszett élék metszéspontjait ki kell számolni és egy számláló segítségével szakaszokra kerül felbontásra aszerint, hogy hányszor lépett be és ki a telekből az egyenessel. Mivel az egyenes a telken kívül kezdődik ezért belátható, hogy a számlálónak mindig páros értéke lesz, ha a telken kívül található a szakasz és páratlan értéke, ha belül. Ha a vizsgált pont egy olyan szakaszon található, amely telken kívül helyezkedik el akkor az adott pont eldobásra kerül és új kerül generálásra.

Egy megtalált poligonon belüli pontra ezután ráhelyezhető egy épület modell, ha elfér. Ennek vizsgálatára elsőként egy kör meghatározása történik, aminek a sugara az épület modell leghosszabb oldala. Következő lépésben adott ponthoz legközelebbi él kerül meghatározásra. Ennek az élnek a távolsága összehasonlításra kerül az épület méretét meghatározó sugárral és ha ez a távolság nagyobb akkor az épület elfér ezen a ponton a poligonban. Ellenkező esetben kisebb épület modell keresésére van lehetőség az importáltak közül. Ha nem fér el egy épület sem, akkor a pontot eldobjuk és ez esetben is új generálására kerül sor.

3.6. TOVÁBBI DEKORÁCIÓS ELEMEL ELHELVEZÉSE, ÖTLETEK

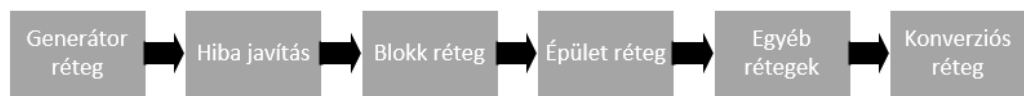
Az elkészült telkek változatossága érdekében egy elkészült telkek besorolható egy előre meghatározott kategóriába például lakónegyed, bolt, park stb., ami a rajta generálásra kerülő épületek típusát és mennyiségét befolyásolja. Például park esetén egy olyan listából kerülhetnek épületek kiválasztásra, amelyek növényeket tartalmaznak.

4. RENDSZERTERV

Ebben a fejezetben a szoftver architektúrájáról lesz szó. Elsősorban a szoftvert alkotó rétegek, majd a projekt megvalósítására használt segéd szoftverek kerülnek bemutatásra.

4.1. RÉTEGEK

Mivel a projektet szinte a végtelenségig lehetne bővíteni további javító, dekoráló algoritmusokkal ezért fontos szempont, hogy egy könnyen bővíthető szoftvert eredményezzen. Továbbá felsorolásra kerül a projektben megvalósításra kerülő rétegek listája (4.1. ábra).



4.1. ábra
Rétegek vázlata

4.1.1. ÚT GENERÁTOR RÉTEG

Ez a réteg tartalmazza azokat az algoritmusokat, amelyek a város alapját szolgáló úthálózat generálásáért felelősek. A rétegben több fajta generátor osztály is szerepelhet, hogy bővítés után később futáskor választani lehessen város stílusok között. Bemenete a generátor algoritmustól függ, mert a generátorok testreszabhatósága nem azonos. Kimenatként a korábban az adatszerkezeteknél említett gráf implementáció kerül felhasználásra, amivel a szoftver további részei dolgozni fognak.

4.1.2. HIBA JAVÍTÓ RÉTEG

Az elkészült úthálózat gráf ezután ebben a rétegben szereplő hibajavító algoritmusokkal kerül feldolgozásra. Az algoritmusoktól függően itt is szükséges lehet bemeneti paraméter például Snap algoritmus esetében a távolságra. Erre a rétegre azért van szükség, mert a telek kiválasztás helyes működése érdekében garantálni kell az úthálózat hiba mentességét.

4.1.3. BLOKK KIVÁLASZTÓ RÉTEG

Ebben a rétegben a szűrt gráfból a város blokkjainak kiválasztása, feldarabolása és kategorizálása történik. Kimenatként egy listát kapunk a telkekről.

4.1.4. ÉPÜLET RÉTEG

A rendelkezésre álló telkek listájának segítségével itt kerülnek elhelyezésre az épületek. Ez tartalmazza az épület generálási logikákat és a felhasználó saját modelljei esetén annak elhelyezését a logikáját.

4.1.5. EGYÉB RÉTEGEK

Miután minden lényegesebb réteg végez azután lehet elhelyezni a dekoratív rétegeket, amelyek a város kisebb részleteit töltik ki. Ez a réteg a jövőbeli bővítési opciókat teszi lehetővé.

4.1.6. KONVERZIÓS RÉTEG

A kód újra felhasználhatóság érdekében az eddigi rétegek nem tartalmaznak keretrendszerre irányuló függőséget és ezért egy olyan rétegre van szükség, ami az általunk generált adatokat képes olyan objektumokká alakítani, amit az aktuális megjelenítő motor képes értelmezni. Unity 3D játékmotor esetén GameObject-ek létrehozása történik az adatokból és ezt már képes térben megjeleníteni. Ez a folyamat valamennyi overhead-el jár, de a teljes város generálásához szükséges számítási igényhez képest elhanyagolható.

4.2. FELHASZNÁLT ESZKÖZÖK

A fejezetben a projekt megvalósítása közben használt külső eszközök bemutatása.

4.2.1. PROGRAMNYELV

Kiválasztásakor szerepet játszott a megjelenítésre használt 3D motor által használt szkript nyelv. Választásom a C#-ra esett, mert a projektben felhasználásra kerülő játékmotort ennek a nyelvnek a segítségével lehet szkriptelni, de nem befolyásolja ez a választás semmilyen formában a használt algoritmusok működését.

4.2.2. MEGJELENÍTŐ MOTOR

A projekt során nem kerül tárgyalásra a megjelenítő motor megírása ezért választani kellett egy már implementált, megfelelő teljesítménnyel rendelkező motort. Az egyedüli szempont, ami fontos, hogy gond nélkül képes legyen kezelni több tízezer objektumot is egyszerre, mert a város sok objektumból fog állni és ennek a megjelenítését valós időben el kell tudnia végezni. Ezek közül a két talán legnépszerűbb szabadon elérhető Unity és Unreal Engine között kellett döntést hozni. Mindkettő jól tud kezelni sok objektumot és rengeteg beállítással rendelkeznek, amivel a megjelenítés optimalizálására lehetőséget ad. C# és Unity egyszerűbb használata miatt erre esett a választás, de ezen kívül bármilyen más jó teljesítményű megjelenítő motor alkalmas lenne. Nem lett volna muszáj 3D motort választani, mert korábbi projektek bemutatták, hogy 2D-ben is megjeleníthető egy generált város, de megjelenítés szempontjából látványosabb cserébe viszont jóval több teljesítményre van szüksége.

4.2.3. VERZIÓ KÖVETÉS

A projekt forráskódjának a kezelésére érdemes használni egy verzió követő szoftvert és szervert. Számtalan előnnyel jár például kód biztonságos tárolása távoli szerveren. Továbbá lehetővé teszi a kód korábbi verzióinak megtekintését és visszaállítását, ha a fejlesztés során új hibák jelennek meg egy módosításnál. Ezek a szoftverek a csapatmunkát is segítik, de ez a projekt önállóan kerül kidolgozásra szóval nem befolyásolta ez a szempont a döntést. Az egyik legnépszerűbb lehetőség a Git verziókövető és GitHub szerver használata, amire végül a választás esett korábbi tapasztalatok miatt és a szolgáltatás méret korlátozása sem okoz problémát.

5. IMPLEMENTÁCIÓ

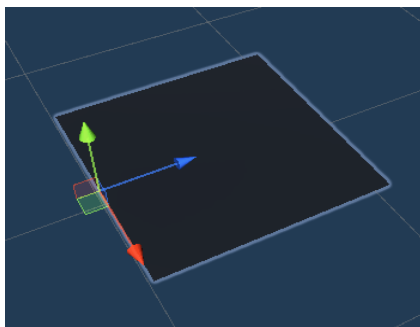
A fejezet a projekt implementációjának folyamatát részletezi és annak kihívásaival foglalkozik.

5.1. PROJEKT ALAPJAINAK LEFEKTETÉSE

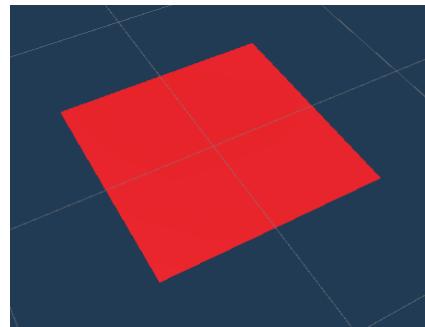
A minimum projekt létrehozása, ami egy már kezdetleges eredménnyel rendelkezik és a megvalósíthatóságot teszteli, avagy „proof of concept”.

5.1.1. UNITY PROJEKT

Az implementáció nulladik lépése a Unity projekt elkészítésével kezdődött, ahol egy teljesen üres projekt kerül létrehozásra. Ezek után a tervben részletezett rétegeknek megfelelően mappaszerkezet jött létre, amivel egy könnyen áttekinthető és bővíthető szoftver a cél. Még szükség van egy pár nagyon alap út (5.1. ábra) és kereszteződés modell (5.2. ábra) létrehozására, hogy a fejlesztés közben vizuálisan is lehetőség legyen a történések követésére, mert a generáció hibáit sokkal egyszerűbb ránézésre kiszűrni ennél a projektnél és ezekre reagálni javító algoritmusokkal. A teknőssel való kirajzolás miatt az út esetében az origó pont az út kezdeti pontján került meghatározásra így a méretének skálázása közben nincs szükség az elemet újra pozícionálni. Középpontos elhelyezés esetén egy méret skálázás nem csak a kívánt előre irányba történne, hanem a másik vége is egyszerre nőne az elemnek. A renderelés gyorsítása érdekében lapok kerülnek felhasználásra, amik nem rendelkeznek „vastagsággal” és csak a szükséges látható oldala kerül kirenderelésre. Az elemek színe csak a tesztelés miatt eltérő, hogy könnyebben lehessen azonosítani az elemet hibák keresése közben.



5.1. ábra
Egyszerű út lap



5.2. ábra
Egyszerű kereszteződés lap

Egyelőre a megjelenítés beállításának minősége nincs finom hangolva, mert generálás után fog kiderülni, hogy szükséges-e a teljesítmény érdekében módosítani olyan beállításokon, mint árnyék rajzolás pontossága, objektum renderelés távolság stb.

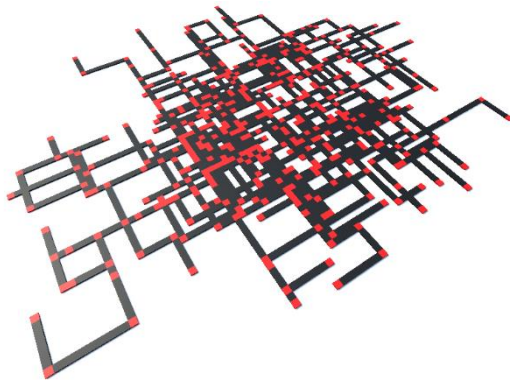
5.1.2. GRÁF ADATSTRUKTÚRA

Egy gráf adattípus létrehozásához szükség van az élek és a csúcsok definíciójára. Ezeknek a létrehozásakor ügyelni kellett arra, hogy később ugyan ezeket az adattípusokat szükséges újra felhasználni a konverziós rétegben ezért generic típusal lesz lehetőség a létrehozásukra így bármilyen adattípussal létrehozható a gráf és elemei. A generic típusnak sok előnye van kód újra felhasználhatóság miatt, de így bármilyen típusnál biztosítanunk kell az algoritmusok helyes működését. Ennek érdekében a generikus típusra egy interfész megkötés kerül elhelyezésre, amivel megkötésre kerül, hogy a gráfban használandó objektum elhelyezhető a térben 3 dimenziós koordináták segítségével. Ez azért szükséges, mert ebben az egyedi gráf implementációban az elemek egyenlősége a pozíciója segítségével kerül meghatározásra és nem megengedhető az, hogy több csúcs ugyan abban a pontban helyezkedjen el, mert a feladat szempontjából nem értelmezhető. A gráfot alkotó másik elem az él az a saját magát alkotó kezdő és vég csúcs referenciájával fog rendelkezni, amivel pozíciója, elforgatása és hossza pontosan meghatározható. Ezen kívül még tartalmazni fogja az egyenes egyenletét így további műveletek során nem szükséges ennek a kiszámolása.

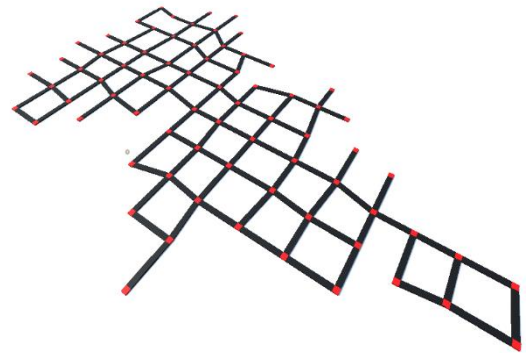
Egy gráf adott rácsa szomszédsági lista segítségével kerül eltárolásra, aminek a megvalósítására a C#-ban implementált Dictionary adattípus lett felhasználva. Ebben a kulcs az a csúcs lesz és ehhez fog tartozni egy HashSet halmaz implementáció, ami az éleket tartalmazza. Segítségével a gráfban hatékonyan kiválasztható egy adott csúcs megfelelő éle. A HashSet helyes működéséhez szükséges az él adattípusnak egy hatékony Hashcode és Equals implementáció is, amikben az élet meghatározó 2 csúcs kerül felhasználásra. A gráf még tartalmaz segéd metódusokat, amikkel csúcs és él hozható létre, törölhető és kereshető. Már létező elemek hozzáadását a Dictionary és HashSet a helyes Hashcode implementáció miatt meggátolja így hatékonyan kerülnek kiszűrésre.

Két elem egyenlőségének vizsgálatakor az elemek térbeli pozíciója kerül felhasználásra, de a térbeli elhelyezés miatt lebegőpontos számok segítségével kerülnek eltárolásra a pozíciók és ezek közvetlen összehasonlítása nem lehetséges a

tud értelmezni, mint előre menetel, fordulás, pozíció mentése, betöltése, toll felemelése és még további utasítások létrehozására is van lehetőség. A létrehozott teknős osztály implementálja ezeket a műveleteket. Példányosításkor egy kezdőpontot igényel, ahol a rajzolást fogja elkezdni. A teknős tárolja a jelenlegi és az előző csúcsot annak érdekében, hogy az útszakaszokat be tudja kötni a hálózatba. Az előre menetel függvénnyel a jelenlegi pontból elindul előre és egy új csúcsot hoz létre, majd beköti az új élet a 2 csúcs közé. A jelenlegi irányának a szögével szögfüggvényekkel ki tudjuk számolni, hogy hol lesz az új csúcs pozíciója. Az elhelyezés előtt ellenőrizni kell a jó minőségű generáció érdekében (snapping), hogy van-e közel másik csúcs ehhez a pozícióhoz, ha van akkor az újat törölni kell és a talált pontra kell ugrania a teknősnek, ahova az élet beköti. A távolságot négyzetes számítás segítségével kiszámolásra kerül az összes pontra és összehasonlításra kerül, hogy a legközelebb az benne van-e egy megadott sugárban.



5.4. ábra
Snap nélkül



5.5. ábra
Snap használatával

A fordulás függvény a teknős jelenlegi irányának a szögét módosítja. Futás után egy adott szövegből mindig változatlan lenne az eredmény, de véletlenszerű tényezők bevezetésre kerülnek az előre menetel hosszára és a fordulás szögére. A generáció így már változatos eredményekkel rendelkezik. Az utolsó fontos utasítás az a teknős mentő függvénye, ami a jelenlegi állapotát menti el a teknősnek egy változóba, hogy betöltéskor rendelkezésre álljon.

A teknős osztályt egy osztályon keresztül kerül vezérlésre. A vezérléshez az L-System-et futtatja egy általunk megadott iteráció számmal, majd egy ciklussal történik az eredmény beolvasása. Az eredményben lévő karakterek egy utasítást fognak jelölni:

- F: előre menetel
- +: jobbra kanyarodás

- -: balra kanyarodás
- [: pozíció mentése
-]: pozíció betöltése



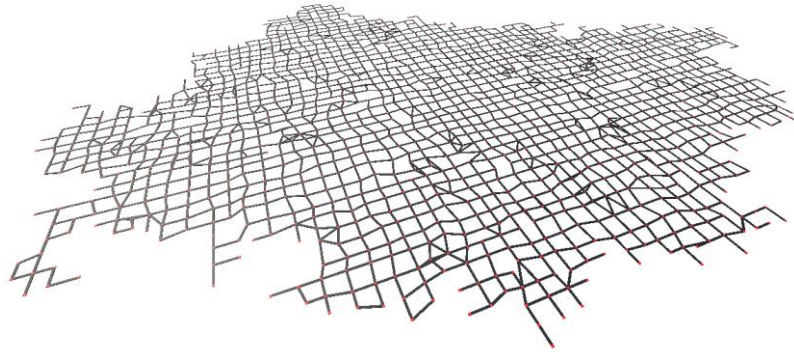
5.6. ábra
Példa az általam használt kimenet formátumára

5.1.4. KONVERTÁLÓ RÉTEG

A réteg a gráf elemeit konvertálja a Unity által használt GameObject formátumba és visszarendezi egy gráf struktúrába. A gráf minden elemén végig iterálva létrehoz egy GameObject-et, amik hozzáadásra kerülnek a konvertált gráfban. Az elem konverziója során egy köztes út építő osztály felhasználásra kerül. Ebben kerül definiálásra a folyamat, hogy csúcsok és élek esetén milyen modellt kell példányosítani egy adott elemből. Az út építő osztályban a felhasználó szabadon helyezhet el kereszteződés és út modelleket felhasználásra. Ezek a hozzáadott elemek már meg is jelennek a képernyőn.

5.1.5. PROOF OF CONCEPT EREDMÉNYEK

Ebben az állapotban egy egyszerűbb úthálózatot egész nagy méretben képes legenerálni a szoftver, de teljesítménye és minősége több szempontból is kifogásolható. Elsősorban mivel a prototípusban a sima gráf adattípus került felhasználásra ezért ahogy a tervben is előre volt látva, a generálás ideje exponenciálisan nő az elemek számának növelésével. Másik probléma, hogy egymást keresztező útszakaszokkal lehet találkozni néhol, ami a telkekbe szervezés algoritmusát megteveszti. Ezeknek ellenére is a prototípus viszonylag jó eredményeket adott így jó alapot fog képezni a projekt megvalósításra az architektúra.



5.7. ábra
Egy nagyobb iteráció számmal készített hálózat több mint 1 perc alatt

5.2. A PROJEKT IMPLEMENTÁCIÓJA

A proof of concept kódbázisának a rendezése és refaktorálása után lehetőség van a további összetett funkciók megvalósítására.

5.2.1. RÁCS ADATSTRUKTÚRA

Tervezés közben a gráf struktúra tűnt a legegyszerűbben implementálhatónak és leggyakrabban használatnak, de gyorsan lelassul sok elem esetén. Ennek a problémának a megoldása érdekében a terv további részében említett rácsos szerkezettel kibővített gráf implementálása következik. A kódban létrehozásra került a Map struktúra, ami egy Dictionary-t tartalmaz. Kulcsként egy 3 dimenziós koordinátát használ és értéke egy hozzátartozó gráf adatstruktúra. A Map létrehozásakor egy rács méret megadása szükséges, ami meghatározza a rács méretét. Így például egy 20-as rács méret esetén a Dictionary-ban a (0,0) pozícióban lévő rács egy olyan gráfot tartalmaz, ami a (19,19)-es négyzet alakú területen tartózkodó összes elemet tartalmazza, vagyis ez a koordináta a rács bal felső sarkát jelöli. A struktúra helyes működéséhez implementálni kell az összes műveletet, amit egy gráf is támogat. A műveletek jóval több logikát tartalmaznak ezért segéd függvényekre is szükség lesz. Elsősorban meg kell tudni határozni, hogy egy elem a koordinátája alapján melyik rács része. Nem bonyolult feladat ugyanis a rács méret és a Math.Floor függvény segítségével vissza lehet kerekíteni a legközelebbi „egész” rács értékre. Így például 20-as méret esetén (34, 12) esetén a (20, 0)-ás rács koordinátáját fogja visszaadni.

Rácsok létrehozásakor Lazy loading-hoz hasonlóan csak akkor kerül létrehozásra egy rács, ha egy elem beillesztése során a pozíciója helyén még nem létezik rács. A létrehozás egyszerű, mert csak egy új bejegyzést kell létrehozni a

Dictionary-ban a megfelelő koordinátával. Egy csúcs beszúrása során ellenőrizni kell, hogy létezik-e a rács, ahova beillesztésre kerül a csúcs. Ha nem létezik akkor létrehozásra kerül beszúrás előtt és a rács koordinátához tartozó gráf csúcs hozzáadás metódusát meghívásra kerül. Él hozzáadásakor a 2 végén lévő csúcs helyét kell megtalálni. Ennek megtalálására egy segéd metódusra van szükség, amivel a gráf implementációja fog bővülni. A metódussal képes a gráf egy meglévő gráfot hozzáadni magához. Ez azért szükséges, mert ha nem ugyan abban a rácsban lenne az él 2 vége akkor nem tudná a gráf ellenőrizni, hogy létezik-e már az él így elutasítaná az elem hozzáadását, mert már a másik végének a referenciáját sem találná meg. A hozzáadás művelete maga egyszerű, mert csak a hívó gráfhoz hozzáadódik a másik gráf összes eleme és így az él hozzáadása elvégezhető a szokásos módon.

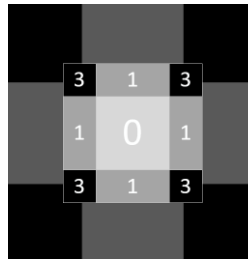
Ezek után implementálni kell a bonyolultabb logikával rendelkező segéd függvényeket is, amik a nagy részét fogják végezni az optimalizációnak. A későbbi feldolgozó függvényeknek lehetséges, hogy nem csak a jelenlegi rács adatai szükségesek, hanem a szomszédosoké is. Erre egy triviális megoldás lenne az, hogy ha ilyen metódussal van dolgunk akkor visszaadásra kerül az összes szomszédja is a lekéréskor, de sosem fog dolgozni a mind a 9 szomszédos ráccsal egyszerre így sok felesleges iterációt jelentene. Helyette egy olyan rendszer kerül implementálásra, ami a jelenlegi kérdéses elem pozíciója alapján eldöntheti, hogy szükséges-e a több rácsot csatolni. Ez jelentősen felgyorsítja a legközelebbi elem megkeresését, ami az úthálózat építésekor minden lépésben használatra kerül. Bemenetnek felhasználja az elem pozícióját és a sugarat, amin belül keressük a legközelebbi elemet. Egy segéd metódus létrehozása szükséges, ami rács határokat ellenőrzi, hogy közel van-e megadott pozíció valamelyikhez.

```
private bool[] FillBorderList(Vec3 vertexPos, float snapRange)
{
    bool[] borderList = new bool[4];
    borderList[0] = Mod((int)vertexPos.X, m.this.ChunkSize) < snapRange; // bottom side
    borderList[1] = Mod((int)vertexPos.Z, m.this.ChunkSize) < snapRange; // right side
    borderList[2] = Mod((int)vertexPos.X, m.this.ChunkSize) > (this.ChunkSize - snapRange); // top side
    borderList[3] = Mod((int)vertexPos.Z, m.this.ChunkSize) > (this.ChunkSize - snapRange); // left side
}
```

5.8. ábra
Szomszédok ellenőrzése elem pozíció és sugár segítségével

Az eredmény egy bool tömb, aminek az értékei jelzik, hogy melyik rács határ oldalhoz van közel a kérdéses elem. Érdemes megemlíteni, hogy a C# beépített modulo függvénye nem tér vissza a célnak megfelelő értékekkel negatív számok esetén ezért egy helyes implementáció keresése szükséges. Az implementáció [17]-ben megoldott módon készült el. Miután képes meghatározni, hogy melyik határhoz van közel a

vizsgált elem így elég csak az azzal szomszédos rácokat egyesíteni. Fontos figyelembe venni, hogy ha egy rács sarkában található az elem akkor több szomszédos rács egyesítése is szükséges.



5.9. ábra
Egyesíteni szükséges rácok száma elem pozíció alapján.

Ennek a módszernek köszönhetően legrosszabb esetben is harmad annyi ráccsal dolgozik egy például leközelebbi elem keresése algoritmus.

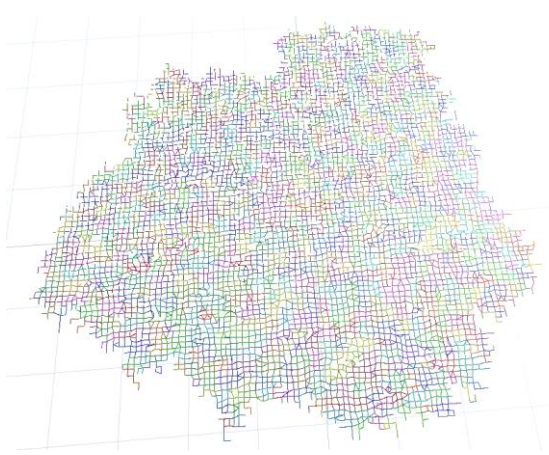
Továbbá a konvertáló réteg számára biztosítani kell 2 módszert, ami a teljes rácsszerkezet által tárolt csúcsokkal és élekkel visszatér, mert konvertáláskor szükséges végig iterálni az összes tárolt elemen.

Az adatszerkezet használatához minden módszer rendelkezésre áll és ilyenkor merülhet fel a kérdés, hogy mennyivel javult a teljesítmény az eredeti struktúrához képest? Az összehasonlításban a normál gráf került összevetésre a rácisos szerkezetű verzió több optimalizáció utáni eredményével. A futási időt a prototípusban használt L-System 6-os és 7-es iteráció számú futtatásával került tesztelésre.

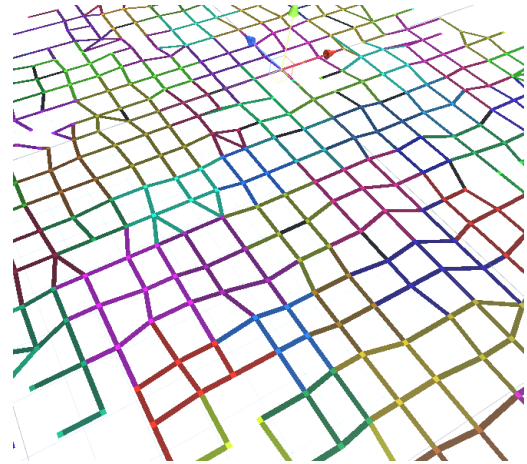
Adatstruktúra	Futási idő 6-os iterációval	Futási idő 7-os iterációval
Normál gráf	0:46	15:37
Map adattípus (Version 1)	4:01	-
Map adattípus (Version 2)	0:39	3:44
Map adattípus (Version 3)	0:20	2:11
Map adattípus (Version 4)	0:09	1:12
Map adattípus (Version 5)	0:06	0:45
Map adattípus (Version 6)	0:04	0:38

Az első elkészült verzió nem várt eredménnyel futott, de refaktorálás és felesleges iterációkat okozó hibák eltávolítása után jelentős gyorsulás tapasztalható.

Az így generált úthálózat eredményéről a (5.10. ábra) és (5.11. ábra) tanúsít. Az egyszínű elemek jelölik az ugyan abban a rácsban tartózkodó elemeket.



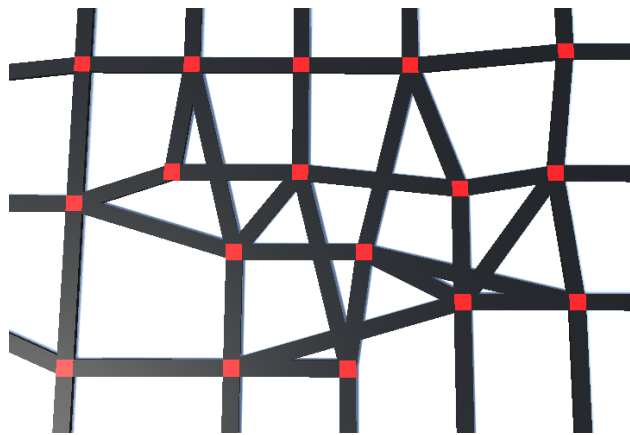
5.10. ábra
Úthálózat ~20000 objektummal



5.11. ábra
Rácsok ábrázolása színnel

5.2.2. UTÓFELDOLGOZÓ ALGORITMUSOK

A legnagyobb problémát az utak kereszteződése okozta (5.12. ábra) a prototípusban és ezeknek a kiszűrésére egy javító réteget kell létrehozni, ami a generált gráfot feldolgozza.



5.12. ábra
Hibás útszakaszok a prototípusban

A szakaszok kijavítása előtt detektálásra kerülnek ezeket a hibák egy algoritmus segítségével, aminek a működése a 3.3.2 fejezetben részletezésre került. A megvalósítás nem járt kihívással, ugyanis már használatra kész állapotban rendelkezésre állt az eljárás, de okozott nem előre látott problémákat az eredmény. A

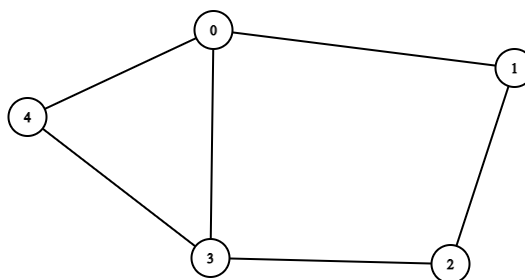
megoldásban a metsző élek eltávolításra kerülnek az úthálózat feleslegesen sűrű részeinek javítása érdekében. Ennek következménye kivételes esetekben, hogy egy adott csúcs csak ilyen éllel rendelkezett ezért az összes eltávolításra került vagyis egy elérhetetlen kereszteződés keletkezett ahova nem vezetett út. A probléma megoldása triviális, mert van lehetőség végig iterálni az összes csúcson hatékonyan és csak ellenőrizni, hogy rendelkezik-e legalább 1 éllel. Ha nincs hozzátartozó él akkor eltávolításra kerül.

5.2.3. BLOKKOK KERESÉSE

Első lépésként az alkalmazott technikák között lévő mélyégi keresés megvalósítása történik meg. Egy potenciális blokk egy csúcsokból és élekből alkotott zárt sokszög területének kerül definiálásra a feladatban. A keresés indításakor szükséges megadni egy maximális mélység szintet és egy kezdő pontot. Az eljárás minden csúcson elindul a gráfban és annak az élein halad tovább a következő csúcs felé rekurzív módon mindig feljegyezve a jelenlegi megtett utat egy előzmény listába. A keresés a következő feltételek teljesülése után szakad félbe:

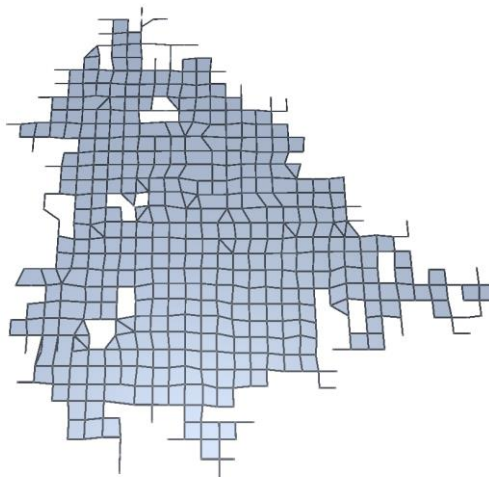
1. Már tartalmazza az előzmény lista a jelenlegi vizsgált csúcsot
2. Elérte a megadott maximális mélységi szintet a keresés
3. Ha a kezdő csúcs és jelenlegi vizsgált csúcs távolsága meghaladja a maximális út hossza és hátralevő mélységi szintek számának szorzatát
4. Visszaért a kezdő csúcsra (Ez esetben a kezdő csúcs nem adódik hozzá az előzmény listához még egyszer.)

A feltételek közül elméletileg csak az 1. és 4. szükséges, de az elfogadható futási idő érdekében érdemes hozzávenni a maradék feltételeket is, amikkel a felesleges rekurzív hívások száma jelentősen lecsökken. Fontos hozzátenni, hogy a maximális mélység száma határozza meg, hogy hány csúcsból állhat maximum egy blokk. Ennek növelése nagyobb telkek létrehozását engedi meg, de a futási idő exponenciálisan nő vele. A keresés befejezésekor az előzmény lista hozzáadódik egy közös listához, ahol az összes megtalált kör tárolódik.



5.13. ábra
Körön belül található kisebb kör

A blokkok még szűrésre szorulnak ugyanis találhatóak olyan gráf körök, amikben van egy kisebb kör, amiről egy példa a (5.13. ábra)-n látható. A megtalált kör a csúcsok szerinti azonosítókkal 01234 sorozattal lenne jelölve. Mielőtt egy kört a blokk listába elhelyezhetünk szükséges ellenőrizni, hogy a körön belül a sorozatban jelölt csúcsok között nincs másik él, ami ugrást biztosít a sorozat egy másik tagja felé út közben levágva a csúcsok egy részét. Ennek megoldása közben felhasználható a keresés azon jellemzője, hogy mindig óramutatóval megegyező irányba fogja gyűjteni a sokszöget alkotó csúcsokat. Végig iterálható a poligon olyan módon, hogy kiválasztásra kerül a jelenlegi csúcs mögött és előtt elhelyezkedő csúcs. Ezek után kiválasztásra kerül a jelenlegi csúcs azon élei, amelyek a sokszöget alkotó csúcs valamelyike felé mutat. Ha ezek az élek között van olyan, ami nem a korábban definiált kiválasztott előző és következő csúcs felé mutat, akkor egy kisebb kört tartalmazó blokkot jelent és ez a blokk eldobásra kerül.



5.14. ábra
Megtalált blokkok szürkével 5-ös mélységi szint mellett

A kiszűrt blokkok ezek után kirajzolhatóak a generált úthálózaton, amiről minta (5.14. ábra)-n található.

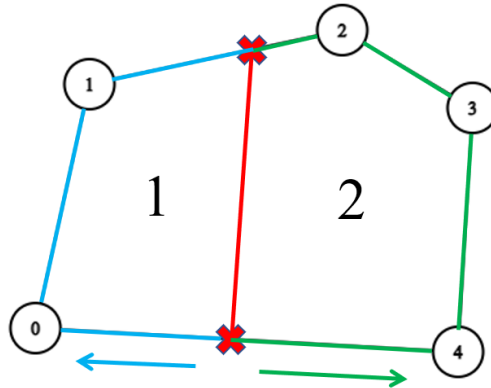
5.2.4. BLOKK TELEKKÉ DARABOLÁSA

Az eredményül kapott blokkok feldolgozása következik, ahol elkészül a város telekrendszere. A 3.4.2-es fejezetben részletezett algoritmus kerül megvalósításra az extra módosításokkal az utolsó bekezdésből. Az egyenes egyenletekkel való számolás miatt egy extra matematikai könyvtár [18] kerül hozzáadásra a projekthez, ami itt egyenletrendszerek megoldására kerül felhasználásra. Az algoritmus egy cél telek méretet, poligont vár és a következő lépésekkel foglalható össze:

1. Leghosszabb blokk él kiválasztása
2. Kiválasztott él középpontjától merőleges irányba sugár kivetítése
3. Sugár által metszett legközelebbi él kiválasztása
4. Metszett él felezőpontjának adott sugarában véletlen pont kijelölése
5. Kivetített sugár kezdőpontjából és véletlen pontból új csúcs alkotása és összekötése éllel, ami a felező élet fogja definiálni
6. Él két oldalán levő pontok szét válogatása 2 új poligonba
 - 6.1. A leghosszabb él, amin a felező él kezdő pontja található annak a kiválasztása
 - 6.2. A kiválasztott éllel kezdve óramutatóval megegyező irányba végig iterálás a poligon többi élén és az érintett csúcsok feljegyzése új poligonba
 - 6.3. Ha a felező él által metszett cél él megtalálásra kerül, akkor a felező él vég csúcsának hozzá adása az új poligonhoz
 - 6.4. A 2. új poligon létrehozásához 6.1-től művelet megismétlése, de 6.2-ben óramutatóval ellenkező irányba történik az iteráció az éleken
7. Ha a 2 poligon valamelyikének területe nagyobb a futás előtt meghatározott cél értéknél, akkor 1. ponttól kezdve a műveletek végrehajtása az adott poligonon

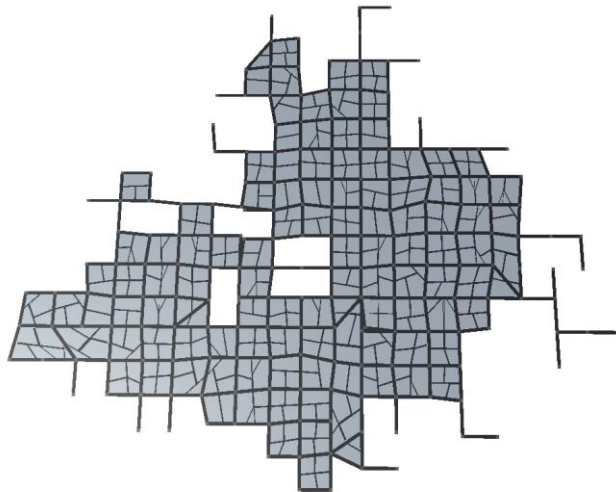
Pár pont ezek közül magyarázatra szorul. A 3. pontban a sugár által metszett él meghatározható a kiválasztott él normál vektorának egyenes egyenlete és a többi él által alkotott egyenes egyenlete által létrehozott egyenletrendszerek megoldásával. A

4. pontban a véletlenszerű pont a változatos telkek érdekében kerültek felhasználásra. A 7. pontban a területszámítás szabálytalan sokszögek esetében nagy kihívást jelent, de nem szükséges tudni a pontos terület méretét ezért csak egy nagyon pontatlan becslés kerül felhasználásra. Folyamata a középponttól és a többi csúctól számított távolságok átlaga, ami a darabolás közben elégségesnek bizonyult. Végző lépésként a poligonokat alkotó éleket fel kell használni a kis utcák létrehozására, aminek a generálása megegyezik a normál úthálózat kirajzolásának menetével, csak keskenyebb utakkal.



5.15. ábra
Elkészült 2 poligon darabolás után

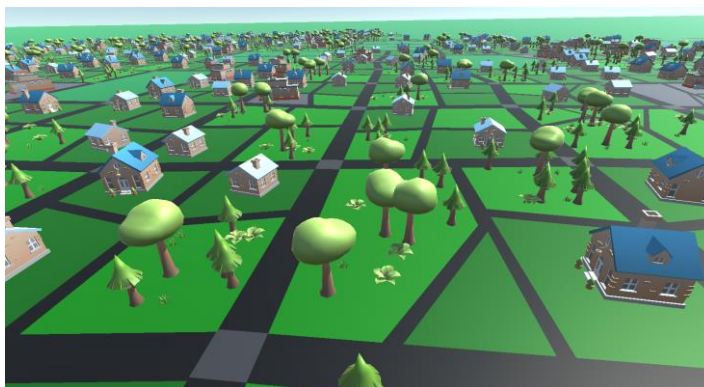
Az algoritmus működését (5.15. ábra) mutatja be, ahol a felező vonal 2 kezdő csúcsa és az alkotó él vörössel van jelölve. A 2 irányú él iteráció útvonalát a különböző színnel kiemelt poligon határ mutatja be. A kimenet látható egy kisebb hálózat esetén a (5.16. ábra)-án.



5.16. ábra
Blokk darabolás eredménye élesben

5.2.5. ÉPÜLETEK ELHELYEZÉSE

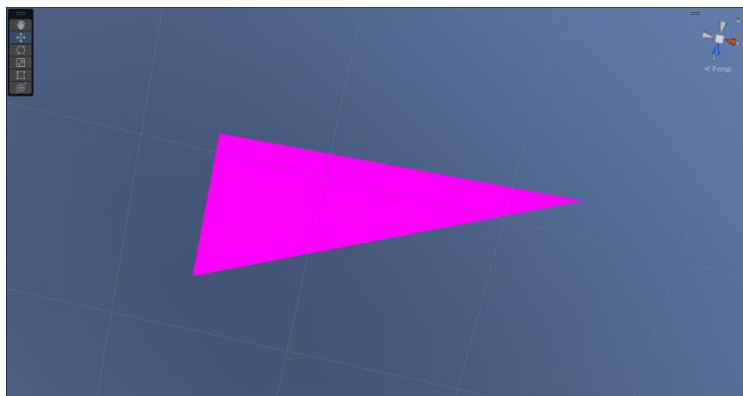
A változatosság érdekében az elkészült telkeket be kell klasszifikálni egy típusba. Ezekből a típusokból bármennyit lehetséges definiálni és a telek objektumokkal való feltöltésének stratégiája változik a típustól. A projektben definiálásra került a lakó terület, piactér és park típus. Minden típus az egyszerű megkülönböztetés érdekében más színű alapterülettel rendelkezik. A felhasználó minden típushoz hozzáadhat saját épület modelleket és az algoritmus ezeknek a méreteivel dolgozik a továbbiakban. Egy épület területét egy egyszerű sugár jellemez, amely az általa elfoglalt területet közelíti. A modell pozíciója a telekben véletlenszerűen kerül elhelyezésre, de elhelyezés előtt több szempont ellenőrzése is szükséges. Mivel a véletlen pont generálásakor a bounding box határokat veszi figyelembe, ezért szükséges ellenőrizni, hogy a kiválasztott pont a telken belül található-e. A 3.5.2-ben részletezett Raycasting algoritmus gyorsan eredményt ad a választott ponton áthaladó sugár egyenes egyenletének és a poligon egyenes egyenletének vizsgálata után kapott metszéspontok segítségével. Az egyenes szakaszokra bomlik aszerint, hogy hányszor lépett be és ki a poligonból. Ha páratlan szakasz részen helyezkedik el a véletlen pont akkor telken belül található. Más esetben pedig addig kerül választásra új pont, amíg nem a poligonon belül található. Ezután meg kell határozni a ponthoz legközelebbi egyenes távolságát, amit a [19]-ban található formula pontosan meghatároz. Ha a távolság kisebb, mint a felhasználó épület listájából véletlen kiválasztott épület mérete, akkor egy új, kisebb épület kerül kiválasztásra. Előfordulhat, hogy nem fér el semelyik épület sem ezért ilyenkor új véletlen pont generálása történik meg. Végtelen ciklusok elkerülése érdekében egy megadott számú sikertelen próbálkozás után az algoritmus kihagyja a jelenlegi telket a generálásból. Az elhelyezett épületek a legközelebbi telek él függvényében elforgatásra kerülnek részben véletlen érték segítségével.



5.17. ábra
Parkokkal, lakóházakkal és boltokkal telített város részlet

5.2.6. MEGJELENÍTÉS

Az épületekkel feltöltött város a teljesítményt jelentősen lerontotta ezért pár beállításon szükséges volt módosítani és render hibák is rontották a látvány világot. Az árnyékok kikapcsolása olyan objektumokon, mint az utak és kereszteződések sok gyorsítással jártak. Egy nagyobb hiba, hogy a telek kirajzoláskor némely mesh fejjel lefele rajzolódott ki, mert mesh készítéskor a Unity motor az óramutatóval megegyező irányban várja el a poligonok megadását és ez a telek daraboláskor nem teljesült minden esetben.



5.18. ábra
Föld alá néző poligon

A problémával már sokan találkoztak ezért kevés keresés után [20]-en lévő forrás segítségével gyorsan egy jól működő megoldással sikerült előállni.

5.2.7. EXTRA ASSETEK

A város feltöltésekor az épületek külső forrásból [21] kerültek kiválasztásra elsősorban idő spórolás miatt. A fekete út textúrák helyett rajzolásra került egy új nagyon egyszerű út textúra.



5.19. ábra
Új út textúra

6. EREDMÉNYEK

A projekt kimenete vizuális leginkább ezért futási időn és megjelenésen kívül nem igazán mérhető a projekt eredménye. A futási idő vizsgálatának jóságát az eltelt idő alatt generált várost alkotó objektumok számával lehet jellemezni, ami tartalmazza az utakat, kereszteződéseket, telkeket, épületeket, tárgyakat és növényeket. A mérések 3 futás átlagát ábrázolják különböző méretű L-system iteráció méret mellett:

Generált objektumok száma	Út generálás ideje	Blokk kiválasztásosnak, darabolásának ideje	Épületek elhelyezésének ideje
~13000	0:00.83	0:00.43	0:00.62
~40000	0:04.67	0:01	0:01
~75000~	0:30	0:02	0:02
~210000	3:40	0:05	0:08

A futási időben észrevehető, hogy az út generálásának ideje exponenciálisan nő, ami már 200 ezer objektum körül több mint 3 percet vesz igénybe, aminek a fejlesztésével érdemes lehet foglalkozni további fejlesztés során.

A megjelenés egy kisebb képgalériával lehet bemutatni.



6.1. ábra
~200000 objektum messziről



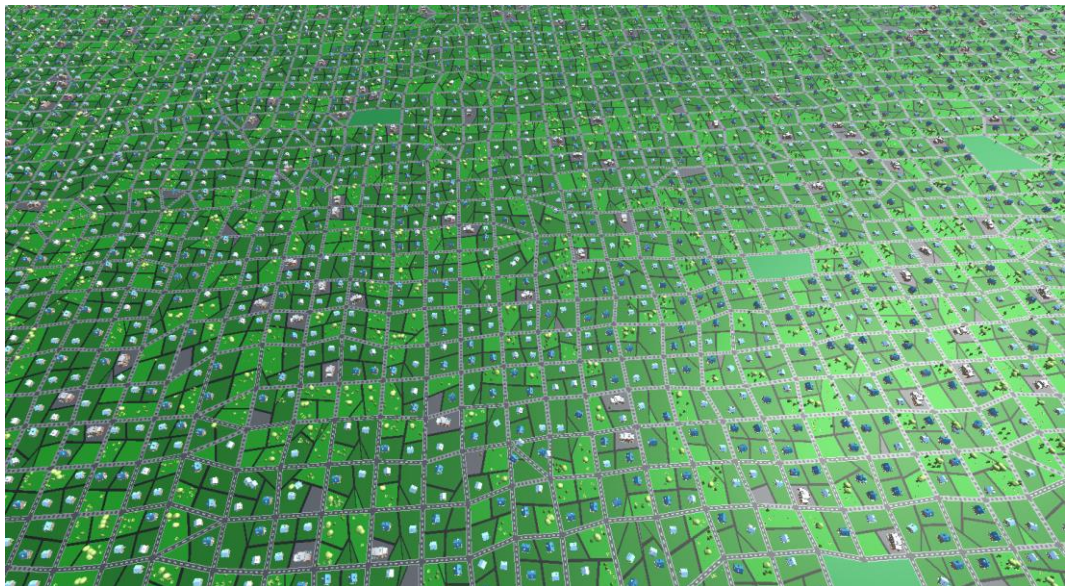
6.2. ábra
Közelített telek nézet



6.3. ábra
Közelített telek nézet



6.4. ábra
Közelített telek nézet



6.5. ábra
Távolabbi telek nézet

7. TOVÁBBFEJLESZTÉSI ÖTLETEK

A projekt sajátossága miatt szinte csak a képzelet szab határt mikor a továbbfejlesztési lehetőségekről van szó. Leglátványosabb ezek közül a domborzat hozzáadása lenne véleményem szerint. Változó felszínnel szóba jöhet a folyók létrehozása, hegyes környékek, kanyonban elhelyezkedő városok és hidak használata is lehetséges lenne.

Úthálózattal kapcsolatban lehetne még több típusú út több generálási logikával összevegyítve. A keresztezések kiegészítése útburkolati jelzésekkel, táblákkal, jelzőlámpákkal, körforgalommal stb. Az utak nem csak egyenesből állnak, hanem görbék felhasználása is lehetséges lenne, amivel 3 dimenzióban magasságban is változhatna. A hálózat kiegészíthető lehet villamossal, metróval, autóbusz állomásokkal. Út menti tárgyak is hozzáadhatóak mint villanypózna, kerítés.

Telkeknél elsősorban a méretük variációja lehetne komolyabb fejlesztés, mert jelenleg átlagban mind hasonlít és több telek egyesítésével lehetséges lenne nagyobb helyet elfoglaló épületek elhelyezésére például repterek, gyárak. Feltöltésükkel kapcsolatban hatékonyabb logikával lehetne feltölteni épületekkel ezzel nagyobb népsűrűség lenne lehetséges. Telkeket körülvevő kerítés, villanypóznák átívelő vezetékkel, tavak stb. Az épületek szobáinak generálása is megvalósítható.

Maga a város jelenleg ránézésre teljesen élettelen ezért az úthálózatot alkotó gráfot könnyedén fel lehetne használni véletlenszerű útvonalak generálására. Ezeken az útvonalokon NPC karakterek sétálhatnának, autókkal közlekedhetnének, tömegközlekedést szimulálhatnának.

Minden fejlesztéssel több részlet jelenne meg a városban, de a teljesítmény drasztikus csökkenését okozná ezért okosabb távoli rács betöltéssel is lehetséges lenne kísérletezni.

Viszont nem csak generálás minőségén lehet dolgozni, hanem a felhasználói interfészen is, mert jelenleg a Unity Editoron keresztül van csak lehetőség paramétereket bevinni a szoftverbe, ami közel sem felhasználó barát.

8. ÖSSZEFOGLALÁS

A szakdolgozatban egy procedurálisan generált város lett megvalósítva elsősorban játékokban pálya tartalomgyártás felgyorsítására. Első sorban tárgyalásra kerül a technika használati lehetőségei. Majd a lehetőségek kiaknázása érdekében már megvalósított projektek után kutatva elemzésre kerültek a technika kreatív felhasználásai. Az elemzett megoldások technikáinak felbontása olyan részproblémák szempontjából, mint:

- Adatok tárolási módszere
- Úthálózat generációja
- Utak által bezárt blokkok kiválasztása
- Blokkok feldarabolása építési telekké
- Telkek épületekkel és egyéb tárgyakkal való populációja
- Generált épület modellek és felhasználó által importált modellek használata

Ezek után a megvalósított rendszer felépítése kerül ismertetésre és annak bővítési lehetőségei kerülnek kiemelésre. Következő fejezetben a rendszer fejlesztési lépéseit részletesen összefoglaló dokumentáció található, ami részletesen kifejti az adott részegység feladatának megvalósítási lépéseit és kihívásait.

Végezetül a megvalósított rendszer eredményeinek ismertetése teljesítmény futási idő elemzésével és kép galérián keresztül.

9. SUMMARY

In this thesis, a procedurally generated city was implemented to accelerate the production of content in computer games. First, the potential uses of the technique are discussed. Then creative uses of these techniques are analysed by looking at projects that have already been implemented to use this potential. The techniques of the analysed solutions are broken down in terms of sub-problems such as:

- Data storage methods
- Road network generation
- Selection of blocks closed by road paths
- Blocks segmentation into building sites
- Population of plots with buildings and other objects
- Use of computer generated city models and user imported ones

After the analysis, the architecture of the implemented software and its extension options are described. The next chapter provides a detailed summary of the development steps of the system, explaining in detail about implementation steps and challenges of the particular sub-tasks.

Finally, the results of the implemented system are presented through performance runtime analysis and with image gallery.

IRODALOMJEGYZÉK

- [1] L-system wikipedia:
(<https://en.wikipedia.org/wiki/L-system>), utoljára megtekintve: 2022.04.23.
- [2] Procedural texture:
(https://en.wikipedia.org/wiki/Procedural_texture), utoljára megtekintve:
2022.04.30.
- [3] Procedural generation:
(https://en.wikipedia.org/wiki/Procedural_generation), utoljára megtekintve:
2022.04.30.
- [4] George, K., Hugh, M.: Citygen: An Interactive System for Procedural City Generation,
(http://www.citygen.net/files/citygen_gdtw07.pdf), utoljára megtekintve:
2022.04.24.
- [5] CityEngine hivatalos weboldal:
(<https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview>), utoljára megtekintve: 2022.04.24.
- [6] CityEngine3D hivatalos weboldal:
(<https://www.citygen3d.com/>), utoljára megtekintve: 2022.04.24.
- [7] Theodor, A., Marcus, A., Alexander, A., Jacob, E., Anton, H., Viktor, T.: Procedural Generation of Modern 3D Cities Bachelor's thesis:
(https://odr.chalmers.se/bitstream/20.500.12380/301969/1/2002%20proceduralgenerationof3dcities_122370000000017720_605621_main-2.pdf), utoljára megtekintve:
2022.04.30.
- [8] Minecraft chunk wiki weboldal:
(<https://minecraft.fandom.com/wiki/Chunk>), utoljára megtekintve: 2022.04.24.
- [9] Voronoi diagram wikipedia:
(https://en.wikipedia.org/wiki/Voronoi_diagram#Farthest-point_Voronoi_diagram),
utoljára megtekintve: 2022.04.25.
- [10] City procedural generation Voronoi approach:
(<http://gamedevindie.com/city-procedural-generation-voronoi-approach/>), utoljára megtekintve: 2022.04.25.

- [11] Procedural city generation: (<https://pjreddie.com/projects/procedural-city-generation/>), utoljára megtekintve: 2022.04.25.
- [12] How to check if two given line segments intersect? – Geeks for Geeks: (<https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/>), utoljára megtekintve: 2022.04.20.
- [13] Großer Beleg, Martin Petrasch: Prozedurale Städtegenerierung mit Hilfe von L-Systemen: (https://www.inf.tu-dresden.de/content/institutes/smt/cg/results/minorthesis/mpetrasch/files/Beleg_MPetrasch.pdf), utoljára megtekintve: 2022.12.01.
- [14] Procedural Generation of Parcels in Urban Modeling: (<https://www.cs.purdue.edu/cgylab/papers/aliaga/eg2012.pdf>), utoljára megtekintve: 2022.04.29.
- [15] Procedural Modeling of Cities: (https://cgl.ethz.ch/Downloads/Publications/Papers/2001/p_Par01.pdf), utoljára megtekintve: 2022.04.29.
- [16] Point in polygon wikipedia: (https://en.wikipedia.org/wiki/Point_in_polygon), utoljára megtekintve: 2022.04.29.
- [17] Modulo implementation stackoverflow: (<https://stackoverflow.com/questions/1082917/mod-of-negative-number-is-melting-my-brain>), utoljára megtekintve: 2022.05.14.
- [18] Math.NET Numerics könyvtár: (<https://numerics.mathdotnet.com/>), utoljára megtekintve: 2022.11.02.
- [19] Távolság pont és egyenes között formula: (<https://brilliant.org/wiki/distance-between-point-and-line/>), utoljára megtekintve: 2022.11.05.
- [20] Poligon csúcs sorrend irányának ellenőrzése: (<https://stackoverflow.com/questions/1165647/how-to-determine-if-a-list-of-polygon-points-are-in-clockwise-order>), utoljára megtekintve: 2022.11.10.
- [21] Épület modellek: (<https://assetstore.unity.com/packages/3d/environments/landscapes/low-poly-cartoon-mini-pack-free-227405>), utoljára megtekintve: 2022.11.15.

ÁBRÁK JEGYZÉKE

1.1. ábra Növény L-System-el[1]	1
1.2. ábra Procedurálisan generált textúra[2]	1
1.3. ábra Procedurálisan generált táj[3]	1
3.1. ábra Rácsos gráf.....	7
3.2. ábra Sarokban található elem esetén betöltött terület	7
3.3. ábra Szélen található elem esetén betöltött terület.....	7
3.4. ábra Voronoi diagram[9]	9
3.5. ábra Snap algoritmus futás előtt, piros kerettel jelölve a betöltendő rácsokat.....	11
3.6. ábra Snap algoritmus futás után, piros kerettel jelölve a betöltendő rácsokat.....	11
3.7. ábra Hibás él futás előtt	12
3.8. ábra Javított gráf	12
3.9. ábra Oriented bounding box subdivision[14]	13
3.10. ábra Raycasting algoritmus X tengely irányban[16]	15
4.1. ábra Rétegek vázlata	17
5.1. ábra Egyszerű út lap.....	20
5.2. ábra Egyszerű kereszteződés lap.....	20
5.3. ábra L-System példa[1].....	22
5.4. ábra Snap nélkül.....	23
5.5. ábra Snap használatával	23
5.6. ábra Példa az általam használt kimenet formátumára.....	24
5.7. ábra Egy nagyobb iteráció számmal készített hálózat több mint 1 perc alatt.....	25
5.8. ábra Szomszédok ellenőrzése elem pozíció és sugár segítségével	26
5.9. ábra Egyesíteni szükséges rácsook száma elem pozíció alapján.	27
5.10. ábra Úthálózat ~20000 objektummal.....	28
5.11. ábra Rácsook ábrázolása színnel.....	28
5.12. ábra Hibás útszakaszok a prototípusban	28
5.13. ábra Körön belül található kisebb kör.....	30
5.14. ábra Megtalált blokkok szürkével 5-ös mélységi szint mellett.....	30
5.15. ábra Elkészült 2 poligon darabolás után	32
5.16. ábra Blokk darabolás eredménye élesben.....	32

5.17. ábra Parkokkal, lakóházakkal és boltokkal telített város részlet	33
5.18. ábra Föld alá néző poligon.....	34
5.19. ábra Új út textúra	34
6.1. ábra ~200000 objektum messziről.....	35
6.2. ábra Közelített telek nézet.....	36
6.3. ábra Közelített telek nézet.....	36
6.4. ábra Közelített telek nézet.....	37
6.5. ábra Távolabbi telek nézet	37