

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1646

KONTROLA TOKA PROMETA

Antonio Hohnjec

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1646

KONTROLA TOKA PROMETA

Antonio Hohnjec

Zagreb, lipanj 2024.

ZAVRŠNI ZADATAK br. 1646

Pristupnik: **Antonio Hohnjec (0036538250)**
Studij: Elektrotehnika i informacijska tehnologija i Računarstvo
Modul: Računarstvo
Mentorica: prof. dr. sc. Željka Mihajlović

Zadatak: **Kontrola toka prometa**

Opis zadatka:

Proučiti osnovu infrastrukture koja čini cestovni promet. Razraditi izradu grafova na temelju zadane infrastrukture. Razraditi algoritme kontrole toka prometa na načinjenim grafovima. Vizualizirati ostvarene rezultate. Načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Koristiti grafički programski pogon Unity i programski jezik C#. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 14. lipnja 2024.

Sadržaj

Uvod	2
Popis slika.....	3
1. Opis aplikacije	4
1.1. Korišteni modeli	4
1.2. Potrebni algoritmi	5
2. Instalacija potrebne programske podrške	10
2.1. Instalacija Blender-a i dodatka za Blender	10
2.2. Instalacija RenderDoc-a	12
3. Prebacivanje i dodavanje dodatnih modela u Unity	13
3.1. Prebacivanje modela u Unity.....	13
3.2. Dodavanje potrebnih objekata	16
4. Način funkcioniranja simulacije	17
4.1. Način kretanja automobila.....	17
4.2. Određivanje težine cesta i trajanja semafora	19
5. Implementacija rada, kodovi i objašnjenja	20
Zaključak	25
Literatura	26
Sažetak.....	27
Summary.....	28

Uvod

Tijekom zadnjih nekoliko godina, mogli bismo primjetiti kako je došlo do brojnih promjena u svijetu. Jedna od najznačajnijih promjena jest nagli razvoj tehnologije. Kroz zadnjih dvadesetak godina izašlo je nekolicina stvari o kojima prije nismo uopće razmatrali da su moguće. Razvoj tehnologije, potaknuo je nadalje povećanu proizvodnju i brzinu stvaranja novih inovacija što je uvelike utjecalo na razvoj raznih tržišta pa tako i automobilske industrije.

Te sve tvrdnje, može nam potvrditi činjenica kako je u Hrvatskoj, u razdoblju od 2012. do 2022. godine broj osobnih automobila na tisuću stanovnika povećan za 44,8 posto. To je trend koji prati cijelu Europsku uniju pa i veći dio svijeta. Nadalje se postavlja pitanje gdje će i kako automobili putovati na prometnicama, kako regulirati promet da bi se izbjegle gužve.

Za ljude koji žive i rade u Zagrebu i većim mjestima, uvjeren sam kako su barem jednom zapali u gužvu na nekoj od prometnica. Sada se postavlja pitanje možemo li mi promijeniti određene prometne znakove i pravila na raskrižjima i prometnicama kako bismo ne nužno optimizirali tok prometa, već kako bismo uzeli u obzir neke od parametara o kojima se nije razmišljalo prilikom izrade prometnica i određivanja prometnih pravila na tim prometnicama.

Upravo to je cilj ovog rada; uzeti dio gradske mreže te pokušati implementirati algoritam kontrole toka koji će uzimati u obzir nekoliko ključnih parametara za raspoređivanje pravila na raskrižju te izraditi jednostavnu simulaciju koja bi nam omogućila vizualizaciju toka na tom dijelu mreže.

Rad se sastoji od učitavanja modela gradske mreže u Blender, izrade i označavanja prometnica, prebacivanje modela u Unity te zatim izrade algoritma kontrole toka i implementacija nad modelom u Unity-u. Kroz ovaj rad dati ću i opis pojedinih dijelova programskog koda te svoj zaključak o potrebi, mogućnosti i uspješnosti razvoja takvog algoritma i programa.

Popis slika

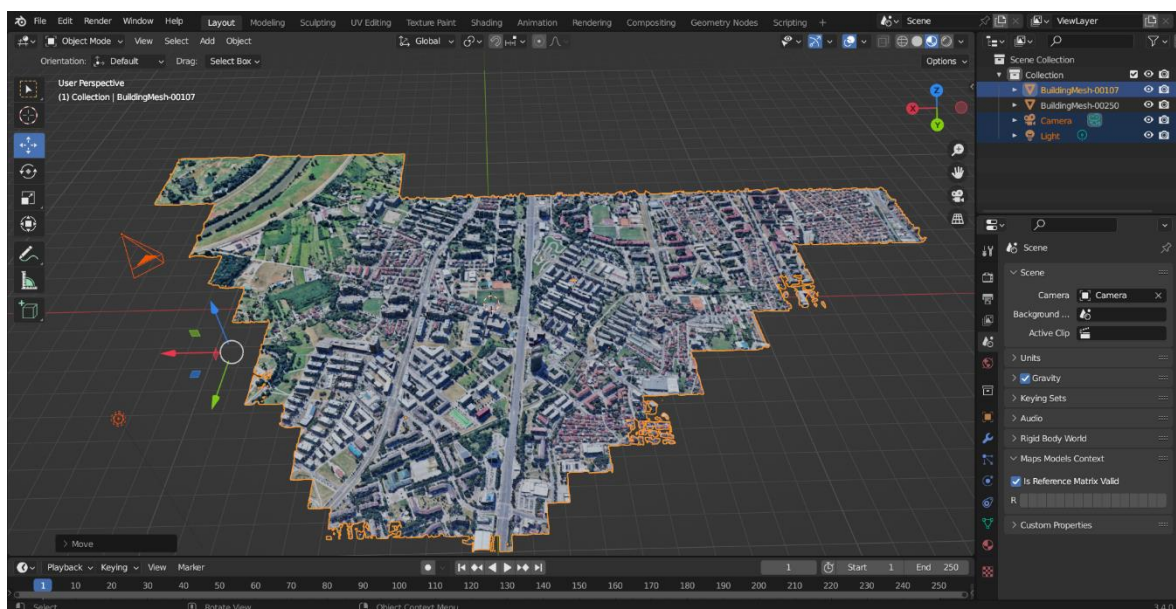
Slika 1.1. Model dijela gradske mreže u Blender-u	4
Slika 1.2 Modificiran Dijkstrin algoritam u C++ jeziku [1].....	6
Slika 1.3 Ford-Fulkerson algoritam u C++ jeziku [1]	8
Slika 1.4 DFS algoritam pretrage puta u C++ jeziku	9
Slika 2.1 Instalacija ekstenzije za Blender	10
Slika 2.2 Instalacija ispravne verzije	11
Slika 2.3 Dodavanje ekstenzije u Blender	11
Slika 2.4 Instalacija ispravne downgrade verzije RenderDoc programa.....	12
Slika 3.1 Model grada u Blender-u.....	13
Slika 3.2 Model grada u Unity bez teksture	14
Slika 3.3 Teksture modela grada	15
Slika 3.4 Model grada materijali	15
Slika 3.5 Model gradske mreže s dodatnim objektima.....	16
Slika 4.1 Automobil na početnoj točki	17
Slika 4.2 Primjer ceste sa raskrižjem.....	18
Slika 5.1 Dodavanje težine na ceste	20
Slika 5.2 Graf grada.....	21
Slika 5.3 Kod DFS pretraživanje	22
Slika 5.4 Kod SpawnPoints metode	23
Slika 5.5 Slika konačnog rada	24

1. Opis aplikacije

Kao što je već napomenuto u uvodu, aplikacija se sastoji od modela gradske mreže učitanoj putem Google maps API-ja, algoritma kontrole toka i same simulacije kretanja objekata kao vozila kroz mrežu. Kroz ovu cjelinu, proći ćemo kroz Blender i opisati postupak kojim umetnuli Google maps područje grada u sam Blender te neke mogućnosti koje smo iskoristili za daljnje oblikovanje.

1.1. Korišteni modeli

Modeli u ovom projektu sastoje se od više dijelova. Prvo je model gradske mreže učitano sa Google maps-a. Zatim imamo modele cesta koji su ručno izrađeni u Blender-u kako bismo mogli izvoditi simulacije. Osim toga imamo i kontrolne točke koje predstavljaju pojedina raskrižja na pojedinim trakama ili prometnicama te ulazne i izlazne točke koje predstavljaju početak i kraj vožnje nekog od vozila. Potom, kako bismo mogli simulirati izvođenje na prometa na tim djelovima mreže imamo i modele vozila koji će u ovom slučaju za potrebe simulacije biti primitivni modeli.



Slika 1.1. Model dijela gradske mreže u Blender-u

1.2. Potrebni algoritmi

Kod izrade ovog rada potrebno je na neki način rasporediti raskrižja. Za sam raspored raskrižja, možemo postaviti dvije inačice algoritma kao dvije ideje.

Jedna ideja je da raskrižja postavljamo na temelju omjera koliko se puta određena prometna traka nađe unutar svih mogućih kombinacija raskrižja i to podijelimo sa ukupnim brojem kombinacija.

Druga ideja je napredak na prvu gdje bi se raskrižja odredila tako da odredimo najveći tok pomoću Ford-Fulkersonovog algoritma gdje bismo tada saznali po kojoj cesti se promet može najbrže kretati i njoj damo najviše vremena za zeleno na semaforu. Ponovimo taj algoritam i saznamo koja je druga najbrža pa njoj damo malo manje vremena za zeleno i tako sve dok ne odredimo sve prijelaze na raskrižju.

Prvo bismo rekli o našem osnovnom algoritmu, a zatim i nešto o Ford-Fulkersonovom algoritmu koji bi bio baza za poboljšanje kontrole toka na grafu.

Naš osnovni algoritam služi za raspoređivanje raskrižja u grafu. No to nije ništa drugo nego više semafora, tj. varijabli koje su funkcije nekoliko varijabli kao što je broj cesti koje ulaze i izlaze iz željene ceste te ukupan broj cesti. Tu se provodi raspoređivanje pripadnog vremena u ovisnosti o tome koliko kolničkih traka sudjeluje u raskrižju te raspoređivanje prikladnog vremena u odnosu na omjer prometnih traka koje u istom vremenu mogu biti aktivne te ukupnog broja prometnih traka. Daljnja razrada ovog dijela biti će pri prikazu koda u sljedećim poglavljima.

Nadalje ćemo ukratko opisati Ford-Fulkersonov algoritam, a na kraju dokumentacije objasniti zamisao i svrhu algoritma u mogućem unaprijeđenju programa.

Kroz Ford-Fulkersonov algoritam, potrebna nam je i implementacija Dijkstrinog algoritma. U nastavku ovog dijela pojasniti ćemo bazne algoritme kroz primjer implementacije u C++ programskom jeziku.

Prvo ćemo opisati dijkstrin algoritam koji smo preoblikovali za potrebe implementacije u Ford-Fulkersonov algoritam. U nastavku priložen je dio koda s modificiranim dijkstrinim algoritmom.

```
void modificirana_dijkstra(int n, vector<int>& tok,
vector<vector<int> >& graf, vector<int>& roditelj){
    vector<int> bio(n, 0); // vektor za provjeru prolaska
    for (int i = 0; i < n - 1; i++){
        int tko, koliko = -1;
        for (int j = 0; j < n; j++){
            if (!bio[j] && tok[j] > koliko){
                tko = j;
                koliko = tok[j];
            }
        }
        bio[tko] = 1;
        for (int j = 0; j < n; j++){ //provjera toka i odabir puta
            if (min(koliko, graf[tko][j]) > tok[j]){
                tok[j] = min(koliko, graf[tko][j]);
                roditelj[j] = tko;
            }
        }
    }
}
```

Slika 1.2 Modificiran Dijkstrin algoritam u C++ jeziku [\[1\]](#)

Ako poznajemo standardni Dijkstrin algoritam, onda znamo kako svrha Dijkstrinog algoritma je pronalazak najkraćeg puta na nekom grafu i to na način da zapisuje duljine putova do čvora te uvijek kao sljedeći čvor s kojim počinje je onaj na kojem je duljina puta do tog čvora najkraća.

Razlika s našim modificiranim Dijkstrinim algoritmom je da se pamti brid najlakše težine na trenutnom putu te se provjerava je li moguće preko ovog brida doći do konačnog čvora tako da tok bude najveći.

U nastavku, prikazan je dio koda za Ford Fulkerson algoritam u c++ programskom jeziku koji se veže na gore dani Dijkstrin algoritam.

Algoritam radi na način da se u *while* petlji svakim prolaskom pomoću Dijkstrinog algoritma izračunava tok, na način da se odabire brid tako da je na odabranom putu najlakši brid što teži te potom na čvorove grafa, puta koji smo odabrali oduzimamo vrijednosti toka grafa, dok na inverzne čvorove dodajemo tu količinu toka.

```

int ford_fulkerson(int n, int m, int start, int end,
vector<vector<int> >& graf){
    const int inf = 1e9;
    int networkFlow = 0;
    vector<int> roditelj(n, 0);
    vector<int> tok;

    while(1){
        tok.clear();
        tok.insert(tok.begin(), n, 0);
        tok[start]=inf;

        modificirana_dijkstra(n, tok, graf, roditelj);

        int flow = tok[end];
        if (flow == 0) break;
        networkFlow += flow;

        int tko = end;

        while( tko != start){
            graf[roditelj[tko]][tko] -= flow;
            graf[tko][roditelj[tko]] += flow;
            tko = roditelj[tko];
        }
    }
    return networkFlow;
}

```

Slika 1.3 Ford-Fulkerson algoritam u C++ jeziku [\[1\]](#)

Nadalje nam za ovaj program treba algoritam traženja puta kako bismo pokrenuli simulaciju od neke početne do krajnje točke. Za to možemo koristiti BFS, DFS, Dijkstrin algoritam za najkraći put i slično. Za jednostavnost backtrackinga puta, ovdje koristimo DFS nad grafom. Primjer DFS rekurzivnog algoritma u C++ programskom kodu imamo ovdje, gdje će u kasnijim poglavljima biti napisan kod naše implementacije u C# jeziku.

```
void DFS(int start, int end, int curr, vector<vector<int> >& graf,
vector<int> bio, vector<int>& end_path, vector<int> path){
    if (bio[curr]) return;
    bio[curr] = 1;
    if (curr == end){
        path.push(curr);
        end_path = path;
        return;
    }
    for(int i = 0; i < graf[curr].size(); i++){
        DFS(start, end, graf[curr][i], graf, bio, end_path, path);
    }
}
```

Slika 1.4 DFS algoritam pretrage puta u C++ jeziku

2. Instalacija potrebne programske podrške

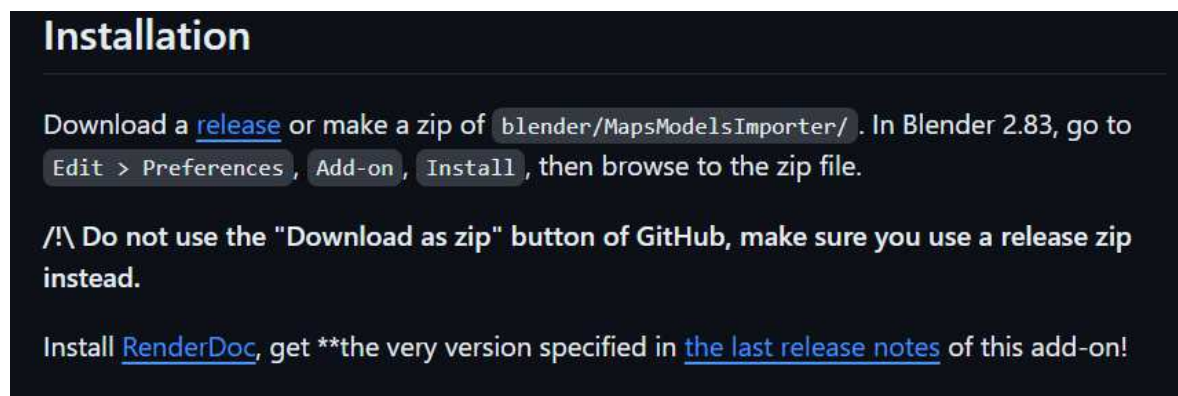
Za instalaciju osim nekih poznatih nam alata potrebno je i instalirati neke posebne biblioteke korištene u ovom projektu. Kroz sljedećih par podcjelina opisati ću što je sve potrebno instalirati te neke napomene vezane za instalaciju.

Za potrebe ovog projekta, nije moguće instalirati najnoviju verziju alata već nam je potrebno uzeti u obzir verzije alata koje su usklađene sa dodatnim ekstenzijama i bibliotekama koje koristimo.

2.1. Instalacija Blender-a i dodatka za Blender

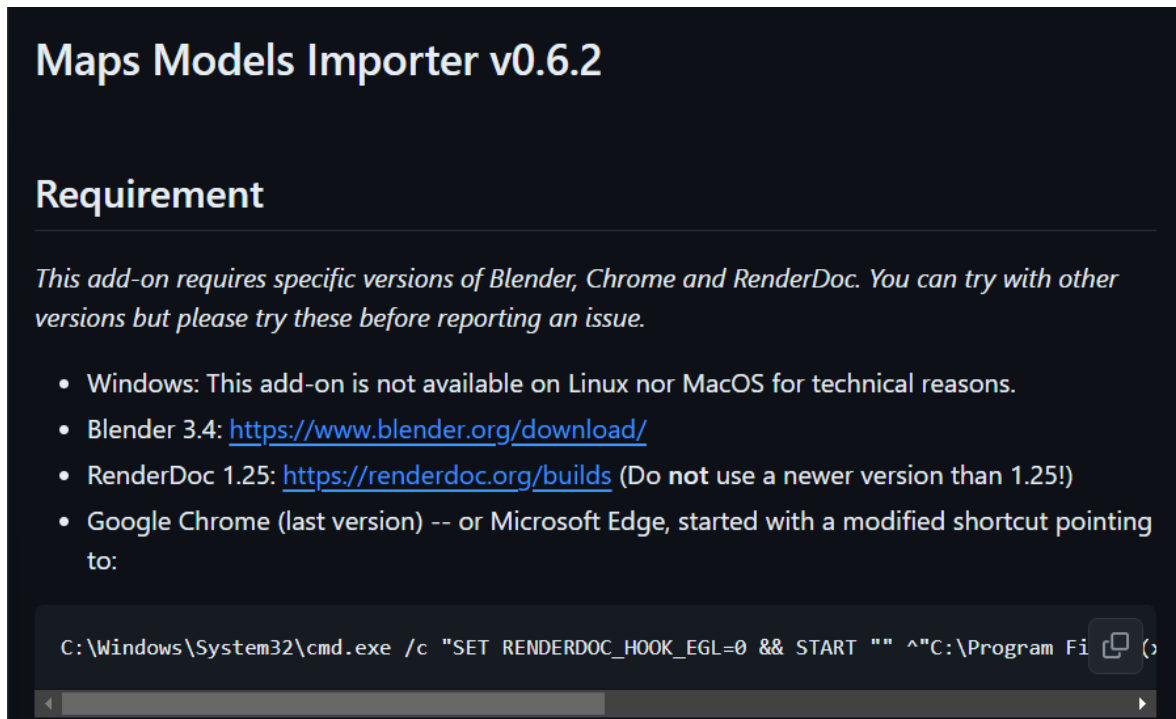
Kao što je navedeno, nećemo koristiti najnoviju verziju Blendera već ćemo koristiti verziju 3.4. Za instalaciju prethodnih verzija Blendera, idemo na sljedeći link: <https://www.blender.org/download/previous-versions/> , pritisnemo „Download Any Blender“ i ondje odaberemo „Blender 3.4“ te potom za odaberemo verziju 3.4.0 za jedan od ponuđenih operacijskih sustava koje imamo. U našem slučaju to je windows. Pri završetku skidanja instalatora, pokrenemo ga te zadržimo sve standardne opcije.

Po završetku instalacije, potrebno je instalirati zasebnu ekstenziju sa githuba <https://github.com/eliemichel/MapsModelsImporter/tree/master>. Ondje idemo dolje i kliknemo na release.



Slika 2.1 Instalacija ekstenzije za Blender

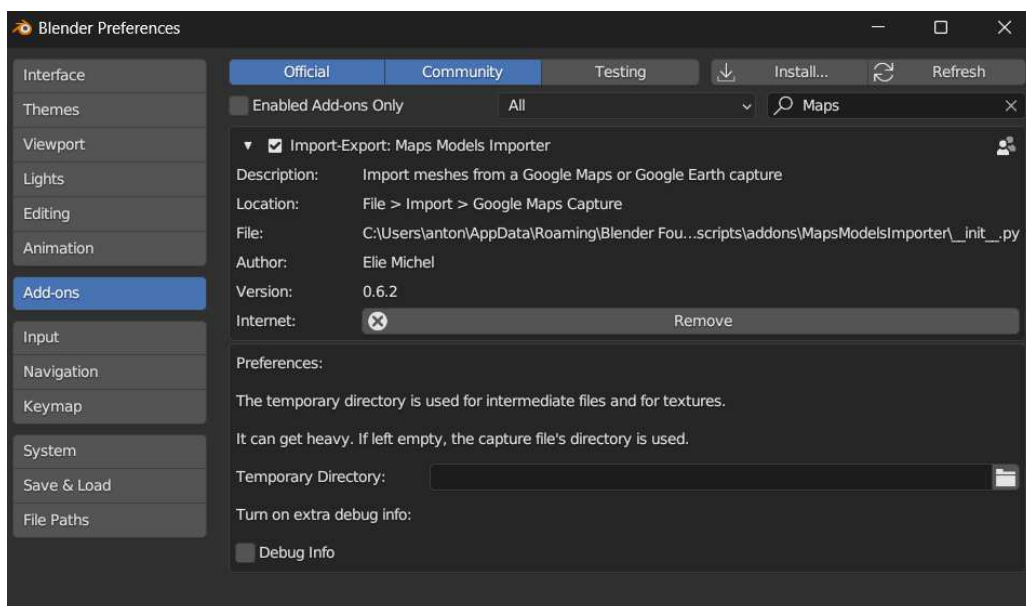
Nadalje, ponovno idemo dolje dok ne vidimo sljedeći zaslone.



Slika 2.2 Instalacija ispravne verzije

Tada odaberemo na Assets i skinemo datoteku MapsModelsImporter-v0.6.2.zip.

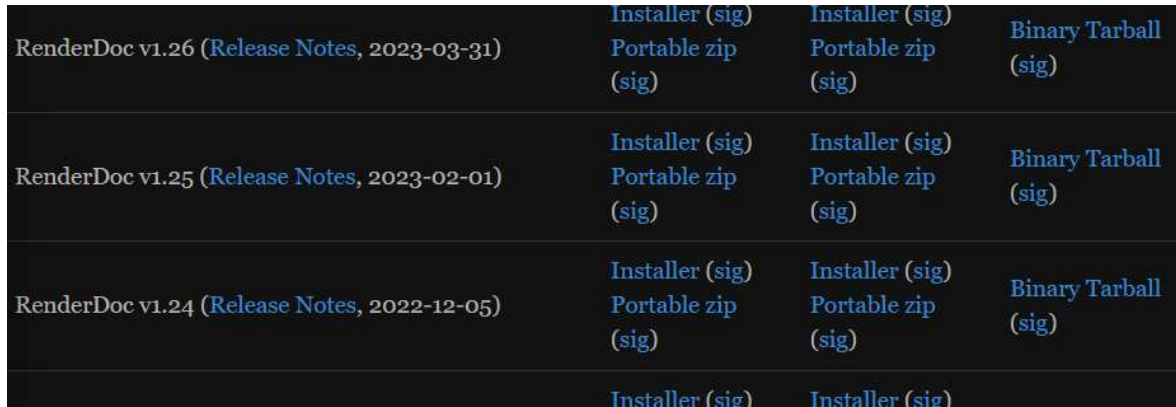
Po završetku instalacije otvorimo Blender i u projektu idemo na Edit → Preferences → Add-ons → Install → Odaberemo skinutu datoteku → Install Add-on. Nakon što smo to učinili idemo na Preferences i pretražimo taj add-on i označimo kućicu.



Slika 2.3 Dodavanje ekstenzije u Blender

2.2. Instalacija RenderDoc-a

Za RenderDoc, potrebno je instalirati verziju RenderDoc v1.25. Za instalaciju idemo na <https://renderdoc.org/builds#stable> , idemo prema dolje dok ne dođemo do verzije 1.25 kao u prikazu prema slici.



RenderDoc v1.26 (Release Notes, 2023-03-31)	Installer (sig)	Installer (sig)	Binary Tarball (sig)
RenderDoc v1.25 (Release Notes, 2023-02-01)	Portable zip (sig)	Portable zip (sig)	Binary Tarball (sig)
RenderDoc v1.24 (Release Notes, 2022-12-05)	Installer (sig)	Installer (sig)	Binary Tarball (sig)
	Portable zip (sig)	Portable zip (sig)	Binary Tarball (sig)
	Installer (sig)	Installer (sig)	

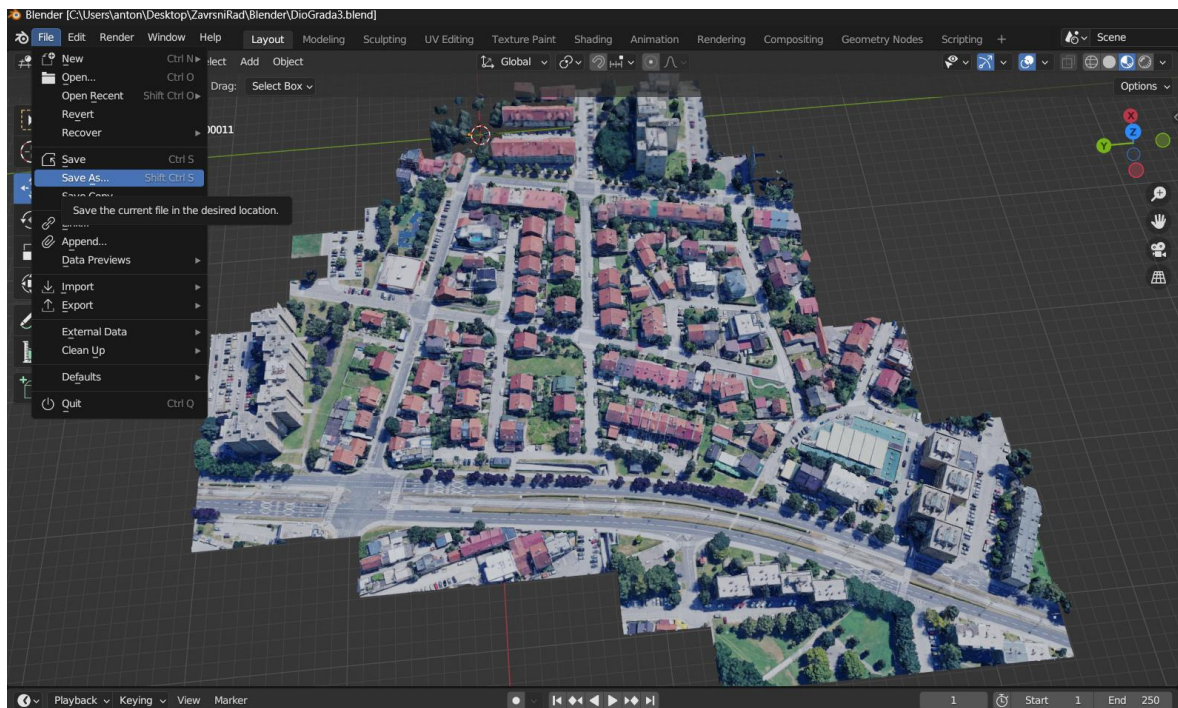
Slika 2.4 Instalacija ispravne downgrade verzije RenderDoc programa

Zatim odaberemo jedan od ponuđenih operacijskih sustava te skinemo installer. Po završetku otvori nam se installer gdje dalje pritisnemo dalje i standardnu instalaciju.

3. Prebacivanje i dodavanje dodatnih modela u Unity

Samo prebacivanje modela grada iz Blendera u Unity sastoji se od nekoliko koraka. Iako način na koji je odrađeno potpuno prebacivanje modela i tekstura nije najefikasniji, jednostavan je i dovoljan za potrebe ovog rada.

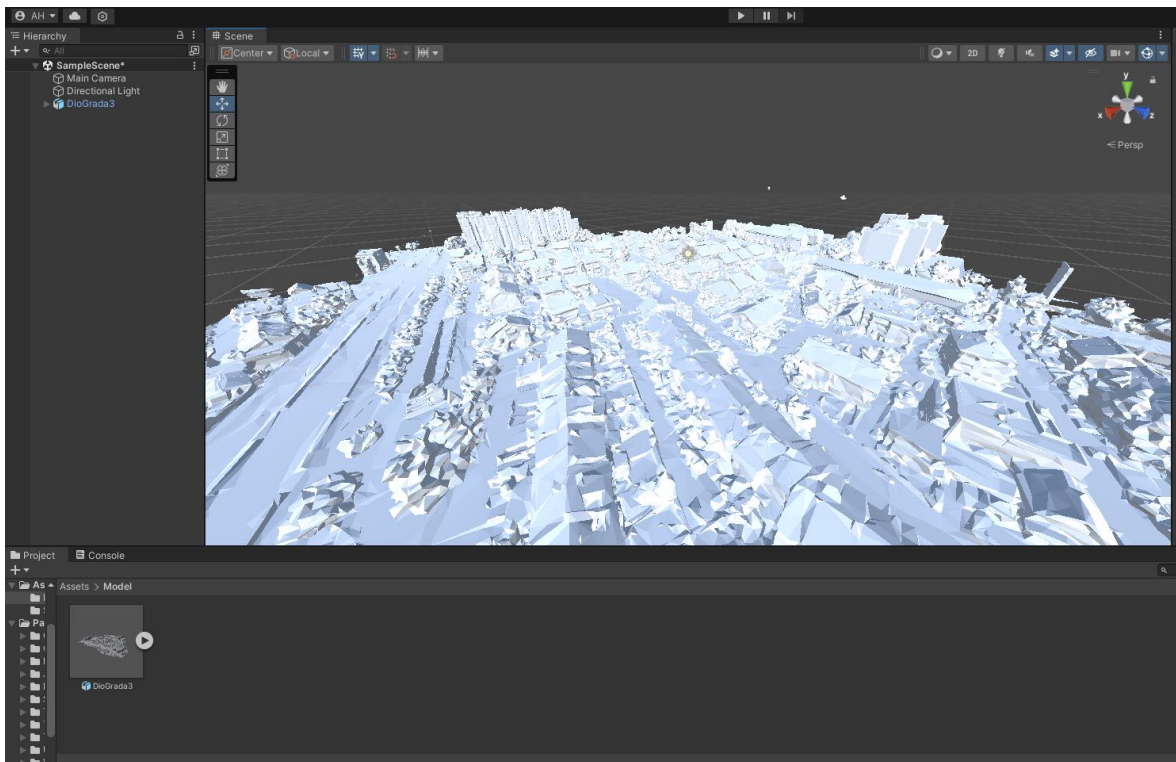
3.1. Prebacivanje modela u Unity



Slika 3.1 Model grada u Blender-u

Za samo prebacivanje modela grada u Unity dovoljno je spremiti datoteku u neki od direktorija iz Unity assets te nadalje povući model na radni dio Unity programa.

Kada to učinimo, u Unity programu izgleda ovako:



Slika 3.2 Model grada u Unity bez teksture

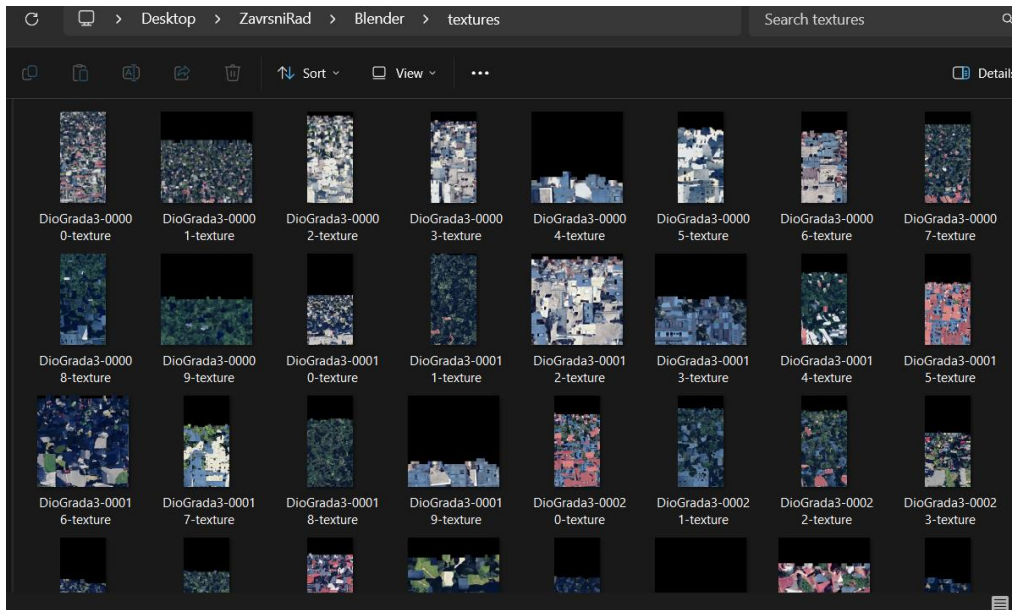
Sada je potrebno prebaciti teksture u Unity. To izvedemo tako da prvo u blenderu generiramo teksture koje onda umetnemo u novi direktorij unutar Unity Assets.

To učinimo tako da u Blender programu, odaberemo File → External Data → Pack Resources te bi trebalo u obavijesti Blendera pisati kako nema što zapakirati. To znači da je sve ispravno te tada ide sljedeći postupak:

File → External Data → Unpack Resources

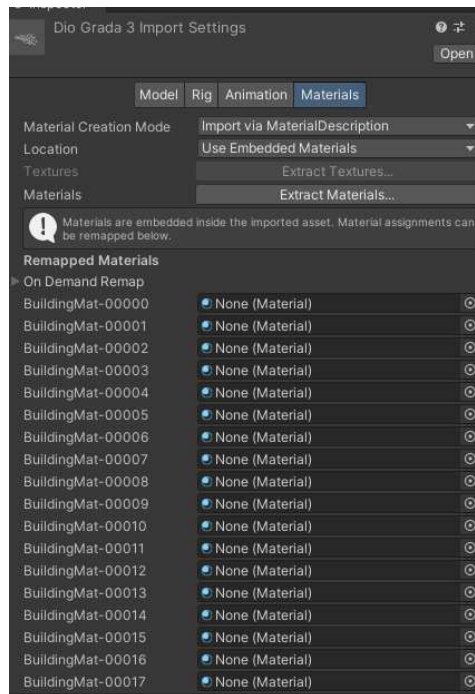
File → External Data → Pack Resources

Nakon što se to odradilo možemo otići u project direktorij Blender projekta i vidjeti kako nam je stvorilo direktorij textures s našim teksturama.



Slika 3.3 Teksture modela grada

Ono što je malo vremenski zahtjevnije je sljedeći dio. Naime mi ovdje imamo 300 odvojenih tekstura koje prebacujemo u Unity, a na taj isti način, Unity ima 300 material komponenti koje moramo popuniti s pripadajućim teksturama. U ovom slučaju, prebacivanje tekstura i spajanje na model je obavljeno ručno. Upravo zato nije ovo nije najefikasnije rješenje.

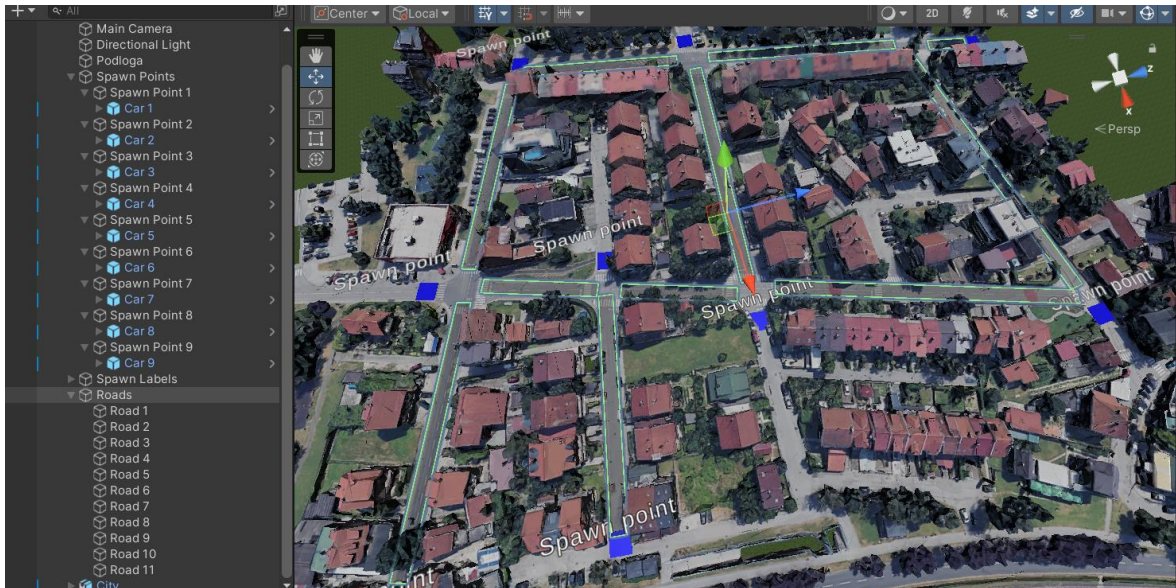


Slika 3.4 Model grada materijali

3.2. Dodavanje potrebnih objekata

U Unity osim modela grada, nalaze se i neki drugi dodatni objekti.

To su: ceste, okviri za početne i krajnje točke kretanja vozila, model vozila te podloga kao okvir za boju prostora izvan dijela učitane gradske mreže.



Slika 3.5 Model gradske mreže s dodatnim objektima

4. Način funkcioniranja simulacije

Zbog jednostavnosti i mogućnosti kolizija i sličnih stvari, svaka početna točka sadrži na sebi vozilo. Pri pokretanju simulacije, vozilo iz nasumično odabrane točke počinje se kretati do nasumično odabranog cilja. Cilj je jedan od preostalih početnih točaka.



Slika 4.1 Automobil na početnoj točki

4.1. Način kretanja automobila

Samo kretanje automobila odnosi se na kretanje po dodatno postavljnim cestama koje sam namjerno ostavio lagano zatamnjene kako bi se lakše prepoznao koncept simulacije. Automobil se po cesti kreće od jednog kraja ceste do drugog, gdje kada je došao do kraja predstavlja da je došao do raskrižja.

Na raskrižju automobil stoji u ovisnosti o tome kamo želi nastaviti te kakav je semafor. Zbog jednostavnosti pretpostavka je da je na svakom raskrižju semafor, tj. neko sinkronizirano propuštanje i zaustavljanje prometa.



Slika 4.2 Primjer ceste sa raskrižjem

Možemo na ovom primjeru detaljnije objasniti. Označenu cestu imamo kao cesta 1. U programu imamo zapisane sve prijelaze s jedne zatamnjene ceste na drugu ili na neku od označenih točka koje također predstavljaju skretanje i cestu, samo sa težinom nula. U datom trenutku, dolaskom na kraj ceste, automobil skreće u jednu od navedenih cesta. Napomena je da nije nužno da automobil može skrenuti na sve ceste u raskrižju, tj. ima područja na kojima iako vizualno ima ceste, vozilo nemože skrenuti na nju.

4.2. Određivanje težine cesta i trajanja semafora

U ovome poglavlju, definirati ću osnovni način definiranja težine cesta i trajanja semafora, gdje ću se u sljedećim poglavljima dotaknuti same izvedbe.

Moramo prvo definirati neke od parametara.

1. Definiramo kako su sve ceste jednake važnosti te su ograničenja jednaka na svim cestama te iznose 50km/h.
2. Definiramo da svaki smjer skretanja u raskrižju ima svoju prometnu traku, pa time nema čekanja na druge smjerove iako je zeleno.
3. Na cesti nema zaustavljanja i skretanja vozila sve do kraja ceste, tj. raskrižja.
4. Težine cesta odnose se kao vrijeme potrebno da se prođe kroz cijelu dionicu ceste.
5. Trajanje semafora ovisi o omjeru i broju cesta koje prolaze kroz određeni smjer.

Nadalje samu definiciju težine cesta definiram kao duljina ceste / ograničenje brzine * koeficijent kojim stavljamo u omjer duljinu objekta s duljinom ceste u prirodi što ispada kao vrijeme potrebno da se prođe cesta od početka do kraja. Raskrižje je također definirano vremenom te u sekundama, čime se tada može uspostavljati ukupna duljina puta. Ukupna duljina puta je tada suma težina prijeđenih cesta + suma prijeđenih raskrižja, što je ekvivalentno trajanju simulacije.

5. Implementacija rada, kodovi i objašnjenja

Prvo što nam je potrebno jest postaviti težine cesta. Kao što je to navedeno, kako bi nekakvo mjerenje što više realno, koristimo vrijeme potrebno da automobil pređe cijelu dionicu ceste uz ograničenje od 50 km/h kao težine cesta.

Formula za izračun težine glasi:

$$Težina = \frac{\text{duljina ceste}}{\text{ograničenje brzine}} * k + 5$$

Objašnjenje formule je vrlo jednostavno. Duljina ceste je duljina koja se nalazi u Unity alatu, ograničenje brzine je već navedeno ograničenje od 50km/h pretvoreno u m/s, k je koeficijent skaliranja gdje sam uzeo u omjer duljinu u Unity alatu te duljinu ceste izmjerenu na alatu Google Maps.

U nastavku slijedi implementacija u kojoj stavljamo težine u listu za daljnju obradu.

```
void Setup_weights(GameObject gameObject, int i)
{
    GameObject roads = gameObject.transform.GetChild(i).gameObject;

    int n = roads.transform.childCount;

    for (int j = 0; j < n; j++)
    {
        GameObject road = roads.transform.GetChild(j).gameObject;

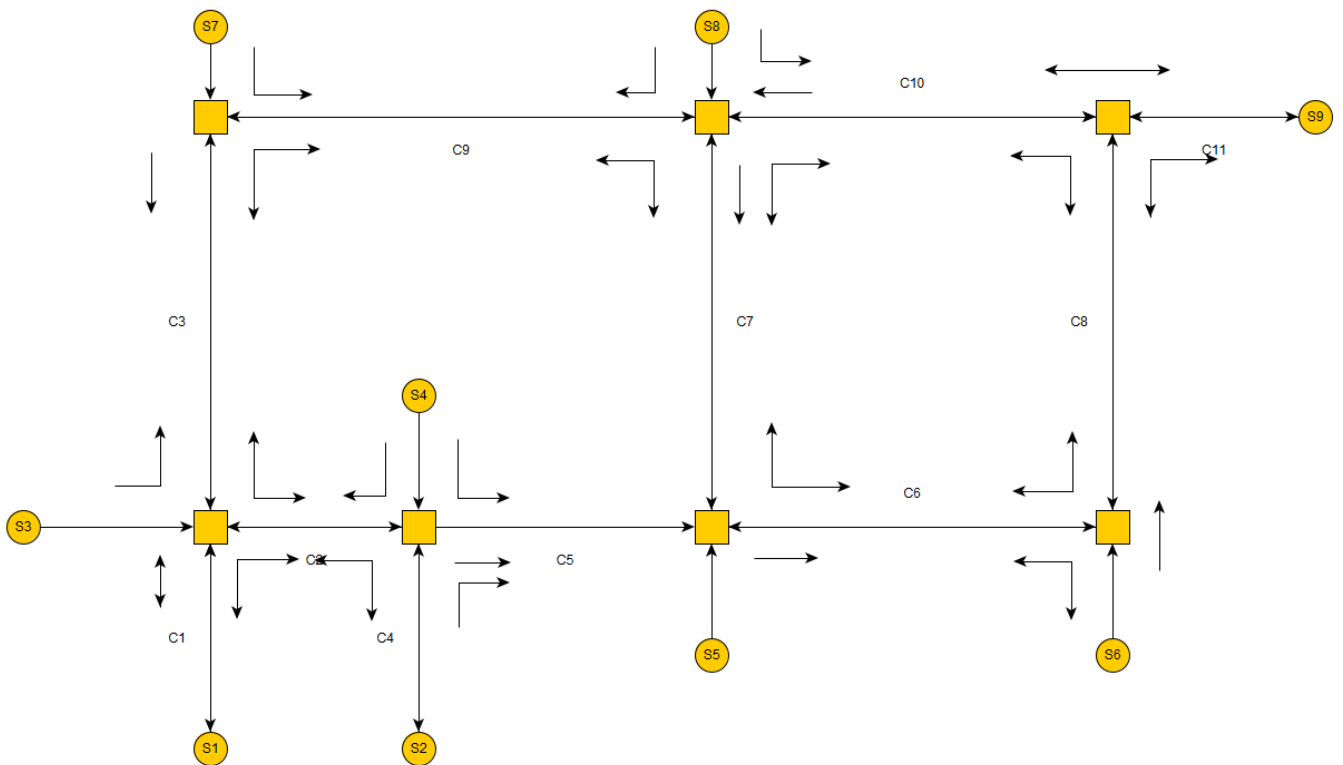
        float duljina_ceste = Math.Max(road.transform.localScale.x,
                                       road.transform.localScale.z);

        float k = 10f * 0.8f;
        float ogranicenje = 500f / 36f;
        float duljina_puta = duljina_ceste * k;
        float tezina_ceste = duljina_ceste / k / ogranicenje + 5;

        road_weights.Add(tezina_ceste);
    }
}
```

Slika 5.1 Dodavanje težine na ceste

Zatim je potrebno povezati raskrižja prijelaze s ceste na cestu, što ćemo učiniti s matricom susjedstva koju nećemo ovdje prikazivati kao kod već samo kao graf.



Slika 5.2 Graf grada

Kružići predstavljaju točke stvaranja, kvadratići predstavljaju raskrižja, a crte između njih predstavljaju u kojem smjeru se smiju kretati.

Dalje što bismo željeli pokazati je traženje puta od početne do krajnje točke. Kao što smo prije spomenuli, točke odabiremo nasumično te se ne može dogoditi da početak i kraj budu iste točke. U nastavku slijedi kod za DFS pretragu.

```
int DFS(int start_road, int end_road, int curr_road, ref List<int>
solution, List<int> road, List<int> bio, int zadnje_raskrizje)
{
    if (bio[curr_road] == 1)
    {
        return -1;
    }
    bio[curr_road] = 1;
    road.Add(curr_road);

    if (curr_road == end_road)
    {
        solution = new List<int>(road);
        return 1;
    }

    List<Tuple<int, int> > new_roads = new List<Tuple<int,
int>>(road_to_road[curr_road]);

    new_roads.Sort(delegate (Tuple<int, int> x1, Tuple<int, int> x2)
    {
        if (road_weights[x1.Item1 - 1] > road_weights[x2.Item1 - 1])
        {
            return -1;
        }

        return 1;
    });

    for (int i = 0; i < new_roads.Count; i++)
    {
        if (zadnje_raskrizje == new_roads[i].Item2) continue;

        int nadeno = DFS(start_road, end_road, new_roads[i].Item1 - 1,
ref solution, new List<int>(road), new List<int>(bio),
new_roads[i].Item2);
        if (nadeno == 1)
        {
            return 1;
        }
    }

    return -1;
}
```

Slika 5.3 Kod DFS pretraživanje

Ovdje obavljamo standardnu DFS pretragu samo su susjedi na koje pozivamo DFS sortirani uzlazno što je karakteristika Greedy algoritma, a uz to stavljeno je pamćenje s kojeg raskrižja je automobil došao tako da se ne može vraćati po istome putu.

Jedan od važnijih djelova koda također je i metoda `SpawnPoints` koja odabire početne i krajnje točke simulacije, pripadne ceste, poziva DFS koji traži putanju od početka do cilja te služi za početni prikaz automobila koji se kreće po putu.

```

GameObject spawn_points =
gameObject.transform.GetChild(i).gameObject;

int spawn_count = spawn_points.transform.childCount;
int start_point = rnd.Next(1, spawn_count + 1);
print(start_point);
GameObject spawn_point_start =
spawn_points.transform.GetChild(start_point - 1).gameObject;
GameObject car =
spawn_point_start.transform.GetChild(0).gameObject;

int start_road_tmp=rnd.Next(0,spawn_to_roads[start_point-1]
.Count);
pocetna_cesta = spawn_to_roads[start_point - 1][start_road_tmp];

car.SetActive(true);
int random_vrijednost = 0;
do{
    random_vrijednost = rnd.Next(3);
}while(end_tocke[random_vrijednost] == start_point);

List<int> bio = new List<int>();

for (int j = 0;j < road_weights.Count; j++) bio.Add(0);

for (int j = 0; j < spawn_to_roads[start_point - 1].Count; j++) {
    int nadeno = DFS(spawn_to_roads[start_point - 1][j] - 1,
        spawn_to_roads[end_tocke[random_vrijednost] - 1][0] - 1,
        spawn_to_roads[start_point - 1][j] - 1, ref putanja, new
List<int>(), new List<int>(bio), -1);
    if (nadeno == 1) break;
}

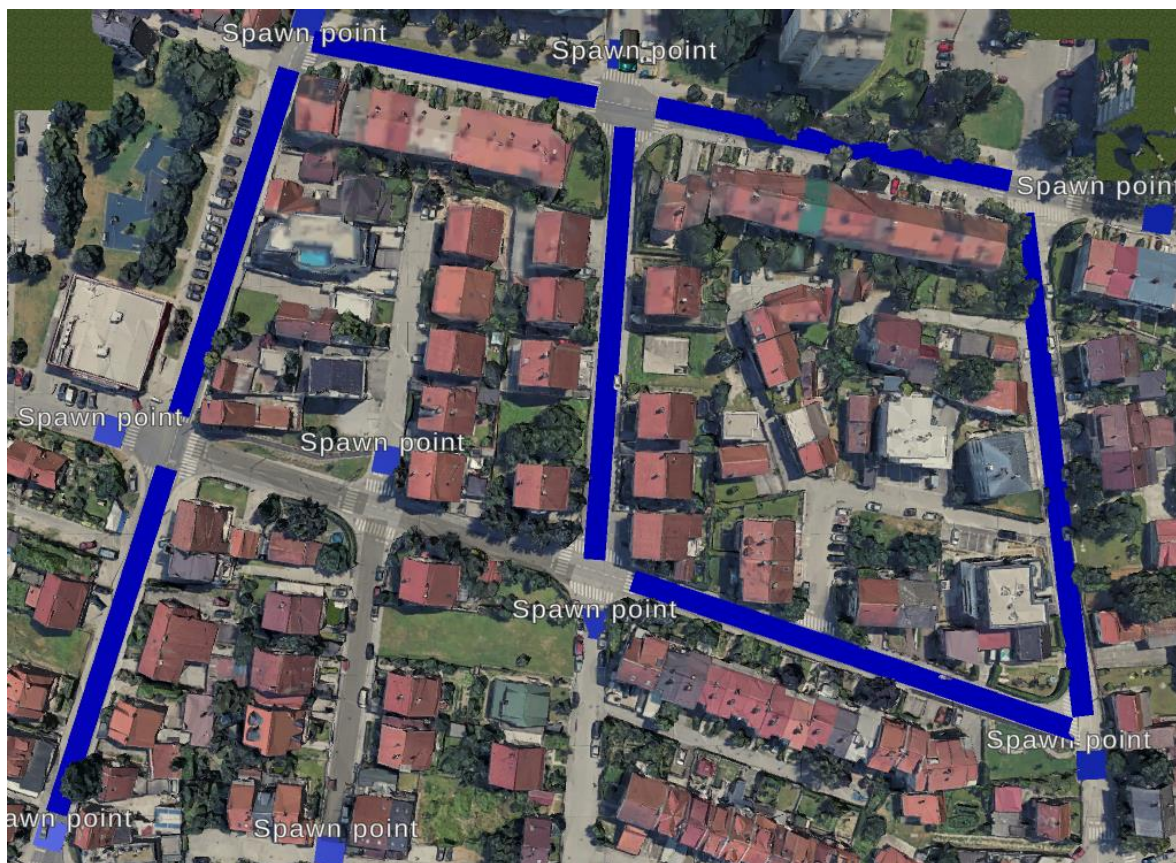
for (int j = 0; j < putanja.Count; j++) {
    GameObject roads = gameObject.transform.GetChild((i+ 1) %
2).gameObject;

    for (int k = 0; k < roads.transform.childCount; k++){
        if (putanja[j] == k){
            MeshRenderer renderer =
roads.transform.GetChild(k).GetComponent<MeshRenderer>();
            renderer.material.SetColor("_Color", Color.blue);
        }
    }
}
}

```

Slika 5.4 Kod `SpawnPoints` metode

Konačno, izgled programa koji radi možemo vidjeti na slici.



Slika 5.5 Slika konačnog rada

Naime ovdje možemo vidjeti automobil na Spawn point koji se nalazi na vrhu u sredini. Plave linije predstavljaju put kojim se automobil kreće. U prikazanom slučaju automobil se kreće ravno do srednje Spawn point točke. Tada on radi krug sve do početnog raskrižja gdje nastavlja do Spawn point točke koja se nalazi na vrhu lijevo. Automobil nastavlja ići skroz ravno sve do najdonjnje lijeve točke koja joj je bila cilj.

Za napomenuti je da se ovdje vidi kako put sigurno nije najkraći jer imamo jedan ciklus, te također možemo vidjeti da kao što sam i prije napomenuo automobil ne može izvoditi polukružno okretanje na ni jednoj cesti. U našoj izvedbi programa, vrlo lagano bismo mogli dozvoliti na nekim raskrižjima bolje rečeno cestama polukružno okretanje.

Zaključak

U današnje vrijeme, s porastom prometa i velikim porastom broja osobnih, teretnih i putničkih vozila, dolazi do stvaranje sve većih prometnih zastoja i čekanja. Tu se dovodi u pitanje kako te zastoje spriječiti ili ih barem umanjiti. Jedno od rješenja je da se projektiraju nove prometnice, povećava broj traka, izvodi regulacija količine automobila na cesti i slično. Ta rješenja iako su efektivna, teško su izvediva uz manjak prostora za gradnju prometnica te zakonske regulative za neke drastične regulacije.

Ono što možemo izvesti jest pokušati regulirati promet na način da iako imamo povećani broj automobila da se regulacijom prometa, prometnih znakova i semafora, unutar zakonskih regulativa, ubrza tok prometa.

Taj problem je izuzetno složen kako zbog same problematike raspoređivanja prometa, tako zbog brojnih faktora koje moramo uračunati u te izračune. Vjeruje se kako je to problem koji je rješiv uporabom napredne umjetne inteligencije koja bi mogla uračunati te čimbenike te optimizirati raspored prometa.

Iako se zbog same problematike to čini najbolji pristup, smatram kako niti jedan problem ne bismo smjeli sagledati iz samo jednog kuta. Ideja kojom sam pristupao ovome radu je da se pokuša potaknuti na sagledanje tog problema iz kuta koji bi pristupio problemu na način da sagledamo gradsku mrežu kao niz manjih cjelina nad kojima bi problem optimizacije bio lakše rješiv te da bi se tada cjeline povezivale uz dodatne algoritme kako bi se dobio neki optimalan raspored većeg dijela gradske mreže.

U ovom radu, sam raspored raskrižja je odrađen jednostavno u usporedbi sa stvarnom složenosti problema, no smatram kako je ovo dobar uvod u pristupanje cjelini i načinu rješavanja problema optimizacije toka prometa.

Literatura

- [1] Napredno programiranje i algoritmi u C-u i C++-u, Domagoj Kusalić, 5. nepromijenjeno izdanje, Zagreb, 2014.

Sažetak

U ovome radu, dane su informacije o idejama, načinu izrade rješenja, implementaciji algoritama nad nekim realnim modelima i podacima te upoznavanje sa učitavanjem realnih modela iz svijeta kroz programske alate RenderDoc, Blender i Unity.

Ovim radom nisu pojašnjene metode i načini korištenja navedenih alata izvan potreba za izradu ovog rada, no dane informacije i postupci mogu poslužiti u izradi sličnih radova.

U rad krećemo sa idejom, postupcima učitavanja modela, algoritmima koji se koriste te koji mogu biti nadogradnja rada te sam postupak daljnjeg izrada i načina rada simulacije.

Ovaj rad ne optimizira kontrolu toka nad grafom ili dijelom gradske mreže, ali daje dobar uvid u problematiku koja se javlja pri pokušaju rješavanja tog problema, a i problematiku simulacije istog.

Kroz rad možemo vidjeti brojne prilike za poboljšanje dijelova programa. Od kvalitete grafike, do povećanja pristupnih i krajnjih točaka, uvođenja novih faktora u raskrižja te poboljšane simulacije programa koja je problem za sebe. Tako se potiče čitatelja da prati ovaj rad, dođe do rješenja na brži način te da krene rješavati problem po problem, možda i nešto novo osmisli.

Svaki rad treba svrhu. To je inovacija nečeg novog, razmatranje nečeg postojećeg kako bismo došli do novih saznanja ili poticaja za daljnje unaprijeđenje i nadogradnju područja.

Summary

In this work, we are given information about ideas, methods of creating and implementing algorithms on real life models and data. Also, we are met with methods of loading real life objects to objects that we can model through tools like RenderDoc, Blender and Unity.

Here, the methods of using tools that were mentioned are not described to the details as it is not in the context of this work, but the information and methods from this work can be easily translated for creation of similar works.

On the start of this work we begin with ideas, methods of loading models and algorithms that could be used in upgrading this work as well as upgrading the simulation algorithm.

This work does not optimize traffic control on graphs or in this way city network, but it gives inside look at problematics of solving traffic control optimization as well as problem of simulating that traffic.

There are countless opportunities to upgrade parts of this work. From quality of graphics, to expanding map and number of entry points and better simulation of program. The reader is encouraged to read this work carefully, to create similar work faster and efficiently and to solve some of the problems that they find in this work, so they could create something entirely new.

Every work needs purpose. It is the innovation of something new, consideration of things that exist so we could get to new knowledges or to encourage others for further improvement and development of area of traffic control optimization.