SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 253

PLANIRANJE RUTE NA VISINSKOJ KARTI

Josip Komljenović

Zagreb, lipanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 253

PLANIRANJE RUTE NA VISINSKOJ KARTI

Josip Komljenović

Zagreb, lipanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 10. ožujka 2023.

DIPLOMSKI ZADATAK br. 253

Pristupnik:	**Josip Komljenović (0036514351)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Planiranje rute na visinskoj karti**

Opis zadatka:

Proučiti optimizacijske postupke za određivanje najkraćeg puta između dvije lokacije zadane na visinskoj mapi. Posebice razmotriti problematiku postavljanja primjerice manjih tunela ili nadvožnjaka na definirani put. Implementirati neki od proučenih algoritama povezivanja lokacija. Načiniti programsku implementaciju koja omogućuje vizualizaciju, analizu i usporedbu načinjenih postupaka. Na različitim primjerima prikazati ostvarene rezultate. Načiniti ocjenu rezultata i implementiranih algoritama. Izraditi odgovarajući programski proizvod. Koristiti grafički programski pogon Unreal Engine 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: 23. lipnja 2023.

Sadržaj

Uvod

Pronalaženje najkraćeg puta između dvije točke na modelu visinske mape predstavlja jedan od optimizacijskih problema. Svoju primjenu može pronaći u raznim ljudskim djelatnostima, poput prostornog planiranja za izgradnju infrastrukture u stvarnom svijetu, industrije videoigara prilikom generiranja svijeta, modeliranja kretanja objekata u trodimenzionalnom prostoru. Neovisno o motivaciji, izračun najboljeg puta u trodimenzionalnom prostoru kojeg predstavlja visinska mapa se pokazao kao jedan od teških problema, bilo što se tiče računske složenosti izračuna, ili kvalitete dobivenog puta

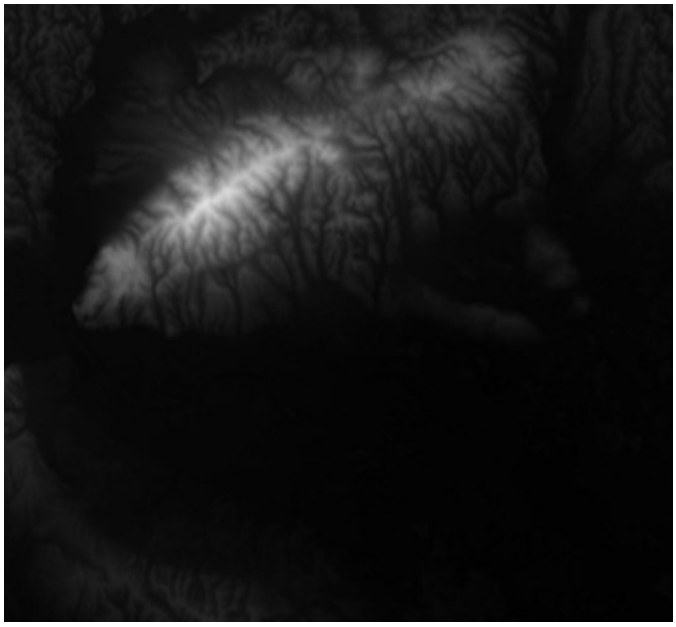
U ovom radu bit će definirani osnovni pojmovi vezani uz problem koji se rješava, pregled osnovnih algoritama koji nam mogu poslužiti u rješavanju problema, njihove prednosti i nedostatke, matematičke osnove algoritma transformacije udaljenosti na zakrivljenom prostoru, koji je jedan od najboljih postupaka za izračun optimalnog puta. Rad također prikazuje razne druge mogućnosti u izradi puta, poput dodavanja tunela ili mostova i slično, donosi analizu složenosti izračuna, kao i moguća poboljšanja za brži izračun. Rad dolazi s programskom podrškom napravljenom u *Unreal Engine*-u korištenjem programskog jezika C++.

1. Osnovni pojmovi

Za početak je potrebno definirati osnovne pojmove bitne za razumijevanje ovog rada

1.1. Visinska mapa

Visinska mapa je dvodimenzionalni prikaz karakteristika trodimenzionalnog svijeta. Kao ulaznu strukturu za visinsku mapu uzimamo dvodimenzionalnu 16-bitnu crnobijelu sliku u .png ili .raw formatu. Svaki piksel na toj slici označava jednu točku tla na modelu svijeta kojeg prikazujemo. x i y koordinate se preslikavaju direktno u x i y koordinate na modelu svijeta, uz moguće dodavanje razmaka između dvije susjedne točke, ovisno o mjerilu između visinske mape i samog svijeta koji se prikazuje. z koordinata, odnosno visina svijeta, je kodirana u boji svakog piksela. Pravilo glasi, što je piksel svjetliji (bjelji), to je visina te točke veća. Iz svega toga, zaključujemo da preslikavanje točke između modela visinske mape i prikaza svijeta glasi $(x,\ y,\ boja)_{visinska\ mapa} = (x,\ y,\ z)_{svijet}$. U Unreal Engine-u možemo kreirati novi svijet i oblikovati teren pomoću ugrađenih alata, te odabrati opciju izvoza svijeta kako bismo generirali visinsku mapu, ili možemo učitati visinsku mapu stvarnog svijeta dostupnu na internetskim servisima, ili pak našu generiranu, te na temelju nje Unreal Engine stvori tlo, a također možemo kontrolirati vrijednosti poput veličine mape ili faktora visine



Slika 1-1 Visinska mapa zagrebačkog područja (izvor: https://tangrams.github.io/heightmapper/)

1.2. Optimalan put

Prije nego što nastavimo s razmatranjem načina za izračun optimalnog puta, prvo moramo razjasniti što smatramo pod pojmom optimalnog puta. Za početak možemo reći da je optimalan put između dvije točke na modelu visinske mape onaj kojim se prođe najmanja udaljenost. I u generalnom slučaju nam to može biti dovoljno. Međutim, ako želimo podržati mogućnost postavljanja tunela ili mostova, u obzir moramo uzeti i cijenu puta. Generalno pravilo bi glasilo, dodavanjem tunela ili mostova se skraćuje duljina puta, međutim povećava se cijena puta (uz pretpostavku da je zbog tehničkih razloga teže izraditi tunel ili most te stoga on košta više od obične ceste koja ide površinom tla). Tu primjećujemo da je riječ o višekriterijskoj optimizaciji funkcije cilja. Nekad će nam biti cilj dobiti što je moguće kraći put, dok će nam nekad biti cilj dobiti najjeftiniji put, ili pak put koji zadovoljava određeni omjer duljine i cijene. Osim svega toga, nekad bismo htjeli da put koji dobijemo zadovoljava određene karakteristike. Možemo zahtijevati da put koji dobijemo ima sve zavoje manje od, primjerice, 90°, ako nam je bitna brzina kojom automobil može doći od početne do završne točke, ili da je maksimalni nagib dok se krećemo putem manji od 20° kako bi se zadovoljila moguća zakonska ograničenja. Iz svega navedenog zaključujemo da ćemo određenu pažnju prilikom izrade puta morati posvetiti i odabiru između više dobrih puteva.

2. Algoritmi za izračun optimalnog puta

Izračun optimalnog puta između dvije točke je problem kojim se računalna znanost bavi od svojih najranijih dana. Prvi algoritmi za izračun puta su osmišljeni za dvodimenzionalni svijet, koji je bio predstavljen kao labirint, ili skup prostora odvojenih određenim preprekama. U takvom svijetu je zadana pozicija objekta koji se kreće između dvije točke tog svijeta, i objekt se može kretati u nekom od zadanih smjerova. Najčešći smjerovi kretanja su pomak jednu točku gore (sjever), dolje (jug), lijevo (zapad) ili desno (istok). Osim toga moguće je zadati i mogućnost pomaka objekta u dijagonalnim smjerovima. Uzmimo za primjer svijet u kojem se objekt može pomicati u četiri prethodno navedena osnovna smjera. Objekt se nalazi na određenoj poziciji i želi se pomaknuti u određenom smjeru. Moguće su dvije situacije. U prvoj situaciji objekt se može pomaknuti u tom smjeru, te taj pomak iznosi određeni predefinirani iznos, najčešće 1, budući da se radi o pomaku u susjednu poziciju. Druga situacija je da se na toj susjednoj poziciji nalazi prepreka, te se stoga pomak u tu poziciju odbacuje. Takav svijet možemo promatrati kao graf, gdje svaki čvor predstavlja jednu moguću poziciju unutar svijeta, a bridovi predstavljaju poveznicu između susjednih pozicija. Stoga su prvi algoritmi za izračun optimalnog puta algoritmi za pretraživanje grafa. Budući da svaki čvor može imati više susjednih čvorova, algoritmi se razlikuju po tome u kojem poretku obilaze te čvorove. Od osnovnih algoritama za pretraživanje grafa ističu se algoritam za pretraživanje u širinu, i algoritam za pretraživanje u dubinu. Iako ti algoritmi mogu dati optimalan put između dvije točke, njihov problem je brzina izračuna puta. To je posebno izraženo kod algoritma za pretraživanje u širinu. On u svakom trenutku gleda sve susjedne čvorove grafa, te stoga pretražuje u krivom smjeru za pronalazak najkraćeg puta. Kod pretraživanja u dubinu situacija je nešto drugačija. Pretraživanje u dubinu nam već u prvom smjeru pretraživanja može dati najkraći put (iako je vrlo mala vjerojatnost za to), međutim put kojeg nađe algoritam pretraživanja u dubinu ne mora biti garantirano najkraći, te ako želimo dokazati da je put kojeg smo našli stvarno najkraći, potrebno je to dodatno provjeriti. Stoga su se za problem pronalaska najkraćeg puta na grafu razvili dodatni algoritmi. Prvi takav algoritam je Dijkstrin algoritam koji omogućava da prijelaze između trenutnog čvora i njemu susjednih čvorova označimo s različitim težinama, a prilikom pomaka u novi čvor, u taj čvor upisujemo dosadašnju duljinu puta. Tada za odabir novog čvora odabiremo onaj kojem je zbroj puta prijeđenog da dođemo u taj čvor, i vrijednosti puta potrebnog da dođemo u susjedni čvor najmanji. U osnovnom slučaju traženja najkraćeg puta

na dvodimenzionalnom svijetu gdje je dopušten pomak u jednom od četiri osnovna smjera Dijkstrin algoritam je jednak algoritmu za pretraživanje u širinu (budući da na takvom svijetu svaki pomak ima jednak iznos, i u svakom ciklusu pretraživanja se čvorovi pomiču samo za jedan susjedni čvor dalje od početnog. Međutim ako bi omogućili pomak u dijagonalnom smjeru, ili pogotovo ako pričamo o trodimenzionalnom svijetu, Dijkstrin algoritam pokazuje prednosti u odnosu na osnovne algoritme, zato što dijagonalni pomak nije jednak pomaku u osnovnom smjeru, a i zbog nagiba terena pomak u jednom smjeru nije jednak pomaku u drugom smjeru. Dijkstrin algoritam bi stoga mogli upotrijebiti kako bismo našli najkraći put između dvije točke na modelu visinske mape, međutim taj algoritam bi ostvario veliku prostornu i vremensku složenost za rješavanje tog problema, te nam on stoga nije pogodan. Kao nadogradnja Dijkstrinog algoritma za rješavanje problema pronalaska najkraćeg puta razvijen je algoritam A*. Taj algoritam za odabir sljedećeg čvora uz sve podatke koje koristi Dijkstrin algoritam koristi i heurističku funkciju procjene udaljenosti trenutnog čvora do ciljnog čvora. Tako algoritam prvo bira onaj čvor za koji pretpostavlja da će ga najbrže dovesti do cilja, čime je broj čvorova koje algoritam mora posjetiti značajno manji od Dijkstrinog algoritma. Kako bismo našli najkraći put za heurističku funkciju mora vrijediti svojstvo optimističnosti. Za heuristiku kažemo da je optimistična ako nikad nije veća od prave udaljenosti do cilja. Budući da je zračna udaljenost teoretski najmanja udaljenost između dvije točke, za heuristiku koristimo zračnu udaljenost jer će ona u svakoj točki biti manja ili jednaka stvarnoj udaljenosti do cilja. Algoritam A* se na dvodimenzionalnom svijetu pokazuje kao odličan algoritam i tamo nalazi široku primjenu. Međutim, na trodimenzionalnom svijetu algoritam pokazuje određene probleme. Kao prvo, riječ je o algoritmu koji radi na grafovima kao ulaznoj strukturi podataka. Zbog toga je potrebna relativno velika količina memorije za izvođenje algoritma (pogotovo na primjerima obrađenima u ovom radu, koji bi imali preko milijun čvorova). Također, prilikom izvođenja algoritma potrebno bi bilo spremati razne strukture, kao na primjer listu posjećenih čvorova, koja bi također mogla imati veliki broj članova. Osim samog problema s memorijom, kao problem prilikom izvođenja A* algoritma na trodimenzionalnom svijetu se pojavljuje i problem s heuristikom. Pitanje je na koji način odrediti pravu funkciju za heuristiku. Naime, sama brzina izvođenja A* algoritma se zasniva na kvaliteti funkcije heuristike. Ako bismo imali savršenu heuristiku (funkciju koja bi u svakoj točki uvijek vraćala točnu udaljenost do cilja), najkraći put bi odmah našli. Međutim, heuristika koju bi inače koristili (zračna udaljenost) se pokazuje kao vrlo loša heuristika za trodimenzionalni svijet, jer je pomak u svakom smjeru u svakoj točki različit (zbog visinske razlike) pa zračna udaljenost ne

pokazuje dobre rezultate. Zadržat ćemo osnovnu ideju algoritma (računanje udaljenosti od početne točke i udaljenosti od završne točke) i to izvesti na drugačiji način. Algoritam koji će biti korišten u ovom radu je algoritam transformacije udaljenosti na zakrivljenom prostoru.

2.1. Transformacija udaljenosti na zakrivljenom prostoru

Algoritam transformacije udaljenosti na zakrivljenom prostoru je algoritam prvotno zamišljen za obradu slike, međutim svoju primjenu je našao u rješavanju raznih problema. Algoritam kao ulaznu strukturu prima sliku. U našem slučaju riječ je o crno bijeloj slici visinske mape. Tu sliku možemo prikazati kao dvodimenzionalnu listu kod koje svaki piksel ima svoju poziciju (stupac i redak predstavljaju x i y vrijednost u trodimenzionalnom prostoru), a nijansa sive boje predstavlja visinu (z vrijednost), gdje bijele točke predstavljaju veću visinu, a crne manju visinu. Početnu sliku visinske mape označimo sa slovom G. Algoritam osim slike G treba i pomoćnu sliku (dvodimenzionalnu listu) F koja je jednake veličine kao i slika G. Slika F predstavlja sliku udaljenosti i nad njom će se provoditi izračuni. Na početku izvođenja algoritma sliku F podijelimo na dva dijela. Prvi dio je prostor za izračun i njega nazovemo X a drugi dio je pozadinski prostor i njega označimo s X^C. Za svaki piksel sa slike F vrijedi da pripada točno jednom skupu, ili skupu X, ili skupu X^C. Za početne vrijednosti točki slike F vrijedi sljedeće svojstvo. Ako točka pripada prostoru za izračun (skupu X), tada je njezina početna vrijednost jednaka maksimalnoj vrijednosti koju točka može poprimiti (beskonačno, ili najveća vrijednost ovisna o tipu varijable). Točke koje pripadaju pozadinskom prostoru (skupu X^C) kao početnu vrijednost imaju broj nula. Kada smo definirali početne zahtjeve algoritma, potrebno je i definirati susjedstvo. Svaki piksel može imati maksimalno 8 susjeda, po jednog lijevo, desno, gore i dolje, te po jednog u sva četiri dijagonalna smjera. Primjerice, pikselu koji se nalazi na poziciji $(7,8)$ su susjedni pikseli $(6,7)$, $(6,8)$, $(6,9)$, $(7,7)$, $(7,9)$, $(8,7)$, $(8,8)$ i $(8,9)$. Udaljenost između dva susjedna piksela označimo s $d(\overrightarrow{p_i}, \overrightarrow{p_{i+1}})$. Ovisno o varijanti algoritma s kojoj radimo postoje dvije formule za izračun udaljenosti između dvaju susjednih piksela. Prvu formulu koristimo ako su dopušteni pomaci samo u četiri osnovna smjera (gore, dolje, lijevo i desno) i ona glasi: $d(\overrightarrow{p_i}, \overrightarrow{p_{i+1}}) = |G(\overrightarrow{p_i}) - G(\overrightarrow{p_{i+1}})| + 1$, gdje je $G(\overrightarrow{p_i})$ visina trenutne točke iz koje računamo udaljenost, a $G(\overrightarrow{p_{i+1}})$ visina susjedne točke. Brojka jedan u zbroju dolazi iz činjenice da je to zračna udaljenost između dviju susjednih točaka. Ako je visinska mapa napravljena u određenom mjerilu tada je potrebno koristiti drugu brojku koja predstavlja to

mjerilo. Ako je omogućeno kretanje u dijagonalnim smjerovima tada se koristi formula za transformaciju težinskih udaljenosti koja glasi: $d(\overrightarrow{p_i}, \overrightarrow{p_{i+1}}) = \sqrt{|G(\overrightarrow{p_i}) - G(\overrightarrow{p_{i+1}})|^2 + k}$, gdje k iznosi jedan u slučaju da se susjedna ćelija nalazi u jednom od osnovnih smjerova, a dva u slučaju da je riječ o dijagonalnom smjeru (za vrijednost k ponovno vrijedi pravilo o mjerilu visinske mape i stvarnog svijeta). Sada imamo zadane sve početne vrijednosti i formule te možemo krenuti u rad algoritma. Algoritam transformacije udaljenosti na zakrivljenom prostoru provodimo dva puta, jednom za početnu točku puta, i jednom za završnu točku puta. Pod pojmom provođenja algoritma nad nekom točkom smatramo sljedeće: ta točka pripada pozadinskom prostoru (skupu X^C), a sve ostale točke pripadaju prostoru za izračun (skupu X), i za sve te točke vrijedi prethodno navedeno pravilo početnih vrijednosti. Algoritam se izvodi u dvije iteracije. U prvoj iteraciji računanje kreće od točke u gornjem lijevom kutu (piksel na poziciji $(0, 0)$) i obilaze se svi pikseli krenuvši u smjeru prema desno, a zatim jedan red dolje kada dođemo do kraja prethodnog reda, te u tom sljedećem redu opet obilazimo piksele u smjeru slijeva nadesno. Prilikom obilaska u svakom pikselu se provodi sljedeći izračun. Nova vrijednost piksela na slici F postaje jednaka minimumu između stare vrijednosti u tom pikselu i minimumu udaljenosti između tog piksela, i njemu susjednih piksela koje smo prethodno prošli, na koje dodajemo vrijednosti slike u tim susjednim pikselima. Odnosno, formula za to glasi: $F(\overrightarrow{p_c}) = \min(F(\overrightarrow{p_c}), \min_{\vec{p} \in M_1}(d(\overrightarrow{p_c}, \vec{p}) + F(\vec{p})))$, gdje je M_1 skup kojeg sačinjavaju sjeverni, zapadni, sjeverozapadni i sjeveroistočni susjed piksela $F(\overrightarrow{p_c})$, odnosno susjedi koji su već izračunati. Kada smo obavili izračun za donji desni piksel završili smo s prvom iteracijom i prelazimo na drugu. U drugoj iteraciji se provodi isti postupak, samo u drugom smjeru i s drugim susjedima. Sada krećemo od donjeg desnog piksela i obrađujemo piksele u smjeru prema lijevo, a potom prema gore. Susjedne piksele sada predstavlja skup M_2, a njega čine južni, istočni, jugoistočni i jugozapadni susjed čvora za kojeg trenutno računamo vrijednost slike. Jednom kada tako stignemo do gornjeg lijevog piksela algoritam je gotov s radom i kao rezultat dobivamo potpunu sliku udaljenosti. Rad algoritma možemo promatrati na sljedećem primjeru.

23	25	26	26	28
24	25	27	28	29
24	26	27	29	30
26	27	29	29	30
27	29	30	31	31

Slika 2-1 Dio visinske mape s pripadajućim vrijednostima visine

7	8	9	10	11
9	10	10	11	13
10	12			

Slika 2-2 Dio slike udaljenosti za pripadnu visinsku mapu

Na slici Slika 2-1 se nalazi dio visinske mape G na kojoj ćemo pokazati rad algoritma. Na slici Slika 2-2 se nalazi dio slike udaljenosti F koji je izračunat za tu visinsku mapu. Riječ je o prvoj iteraciji algoritma u kojoj je postupak krenuo od gornjeg lijevog kuta. Prazni pikseli na slici udaljenosti F su oni pikseli koji još nisu izračunati u algoritmu i njihova vrijednost u tom trenutku je beskonačno. Algoritam trenutno računa vrijednost za središnji piksel na tim slikama $(2, 2)$.

7	8	9	10	11
9	10	10	11	13
10	12			

Slika 2-3 Piksel koji računamo i njemu susjedni pikseli koji pripadaju skupu M_1

Algoritam će prvo izračunati udaljenosti između tog piksela i njemu susjednih piksela koji pripadaju skupu M_1 i te vrijednosti iznose:

$$d_{nw} + F_{nw} = \sqrt{|27 - 25|^2 + 2} + 10 \approx 12.45$$

$$d_n + F_n = \sqrt{|27 - 27|^2 + 1} + 10 = 11$$

$$d_{ne} + F_{ne} = \sqrt{|27 - 28|^2 + 2} + 11 \approx 12.73$$

$$d_{nw} + F_{nw} = \sqrt{|27 - 26|^2 + 1} + 12 \approx 13.41$$

Početna vrijednost za središnju točku je bila beskonačno, stoga uzimamo minimum udaljenosti do susjednih točaka. Najkraća udaljenost je do sjevernog susjeda i ona iznosi 11 te stoga tu vrijednost upisujemo u središnji piksel

7	8	9	10	11
9	10	10	11	13
10	12	11		

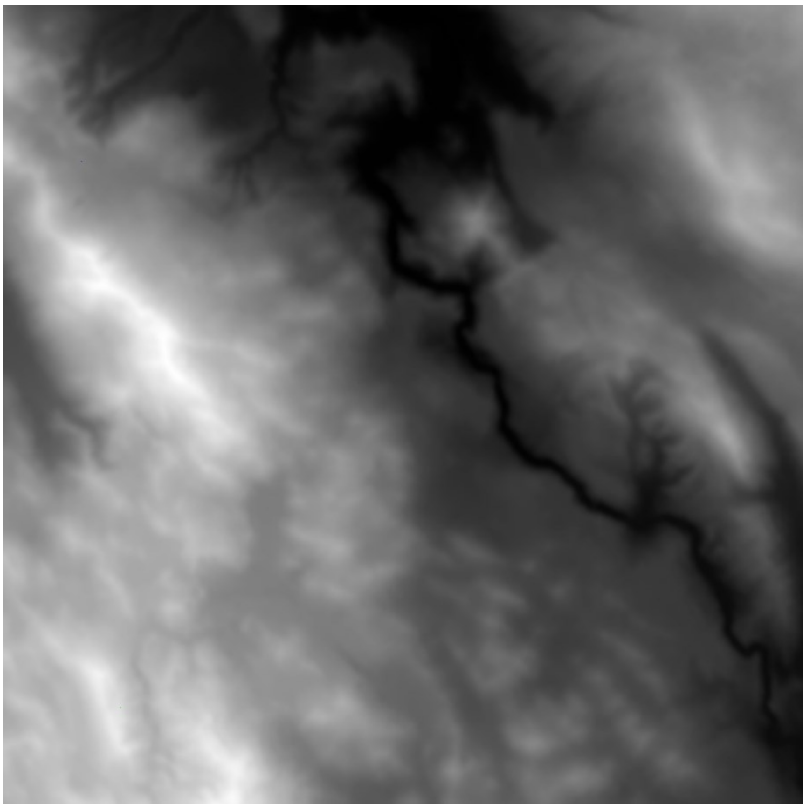
Slika 2-4 Rezultat nakon izvođenja algoritma za piksel na poziciji $(2, 2)$

Algoritam se zatim nastavlja pomakom na desni piksel gdje se provodi isti postupak.

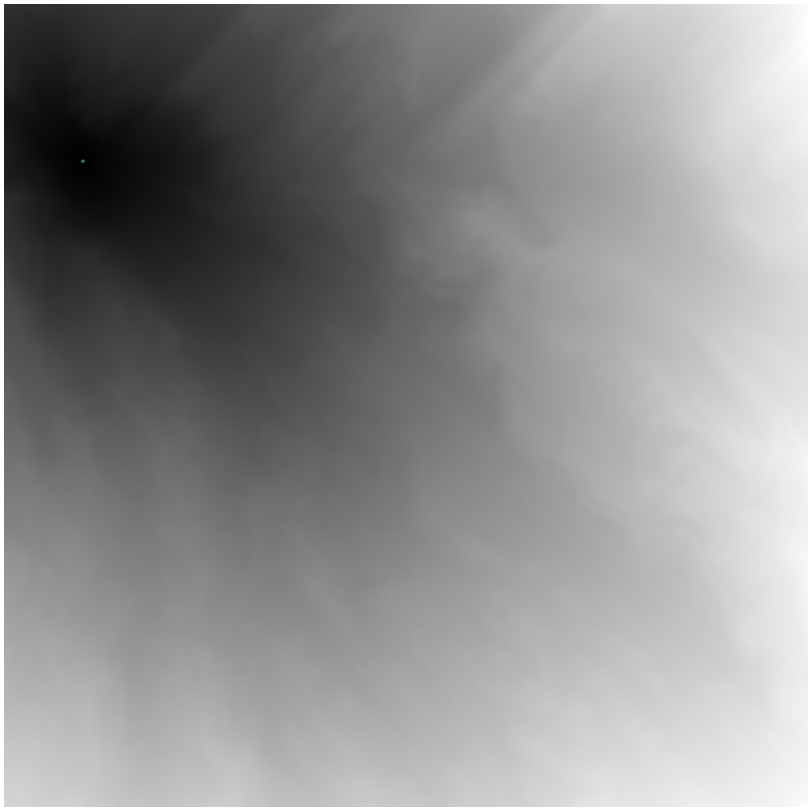
2.2. Korištenje algoritma transformacije udaljenosti na zakrivljenom prostoru za izračun najkraćeg puta

Kao što je prethodno rečeno, algoritam transformacije udaljenosti na zakrivljenom prostoru se može koristiti za izračun najkraćeg puta, budući da je to algoritam za obradu slike, a visinsku mapu možemo predstaviti sa slikom. Kako bismo dobili optimalan put, algoritam je potrebno izvesti dva puta. U prvom izvođenju algoritma početna točka puta (točka u kojoj se objekt trenutno nalazi) pripada pozadinskom prostoru (skupu X^C) dok sve ostale točke pripadaju prostoru za izračun (skupu X). U drugom izvođenju algoritma završna točka puta (točka u koju objekt želi stići) se nalazi u pozadinskom prostoru, a ponovno za sve ostale točke vrijedi da se nalaze u skupu X. Nakon što smo izveli algoritam za početnu i završnu točku sve što nam preostaje je zbrojiti te dvije slike. Zbroj tih slika provodimo kao zbroj

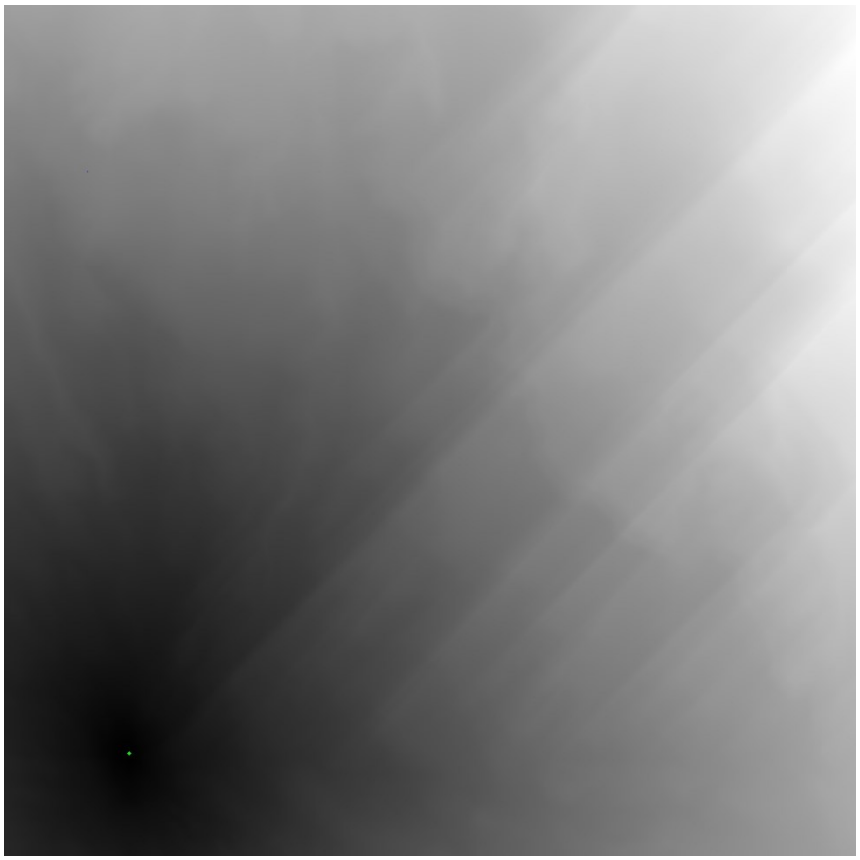
dviju matrica, primjerice vrijednost zbrojene slike na pikselu $(2, 2)$ jednaka je zbroju slike za početnu točku na pikselu $(2, 2)$ i slike za završnu točku na pikselu $(2, 2)$. Jednom kada imamo završnu sliku možemo jednostavno dobiti konačni put, tako što krenuvši od početne točke spajamo točke koje imaju najmanju vrijednost na konačnoj slici. Zbrajanje vrijednosti na početnoj i završnoj slici možemo shvatiti kao poveznicu s algoritmom A*. U algoritmu A* svaka točka je imala vrijednost trenutno prijeđenog puta zbrojenu s vrijednošću heuristike. U algoritmu transformacije udaljenosti na zakrivljenom prostoru za generiranje najkraćeg puta svaka točka ima vrijednost slike udaljenosti za početnu točku (predstavlja prijeđeni put) zbrojenu sa slikom udaljenosti za završnu točku (predstavlja heuristiku). Prednost algoritma transformacije udaljenosti je što pomoću njega možemo na relativno jednostavan način dobiti vrlo dobru heuristiku. Uz to, algoritam transformacije udaljenosti ima i razne druge prednosti. Primjerice, ako je neka točka česta početna ili završna točka, za tu točku možemo u memoriju spremiti vrijednost slike udaljenosti i koristiti ju više puta, zato što bez obzira koja je druga krajnja točka puta, slika udaljenosti je jednaka. Druga prednost je što se u ovom algoritmu ne moramo ograničiti na najkraći put između dvije točke, nego je broj krajnjih točaka proizvoljan. Ako bismo željeli izračunati najkraći put između početne točke, koju ćemo označiti s A, i dvije završne točke, označene s B i C, sve što nam je potrebno je prilikom izračuna slike udaljenosti za krajnju točku u pozadinski prostor (skup X^C) dodamo obje točke (B i C). Trajanje izvođenja izračuna slike udaljenosti za jednu ili više točki je jednako. Nakon što smo proveli algoritam kao rezultat dobivamo najkraći put između točaka A i B, ili između točaka A i C, ovisno o tome koji put je kraći. To možemo primijeniti na bilo koji broj točaka, jedini uvjet je da sve krajnje točke možemo podijeliti u sva skupa (početne i završne točke puta). Ako bi problem traženja najkraćeg puta gledali kao problem teorije grafova, algoritam transformacije udaljenosti nam omogućava da nađemo najkraći brid na bipartitnom grafu. Algoritam transformacije udaljenosti je također pogodan za određene modifikacije koje bi dovele do bržeg izvođenja, no o tome će biti više govora u idućim poglavljima.



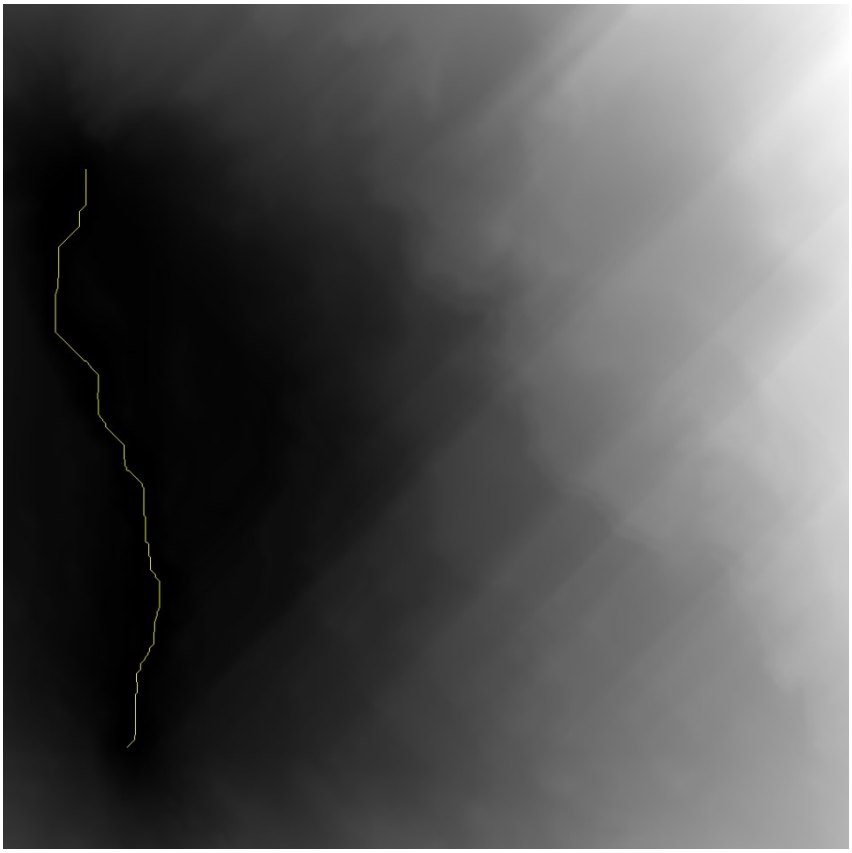
Slika 2-5 Primjer visinske mape



Slika 2-6 Slika udaljenosti za početnu točku



Slika 2-7 Slika udaljenosti za završnu točku

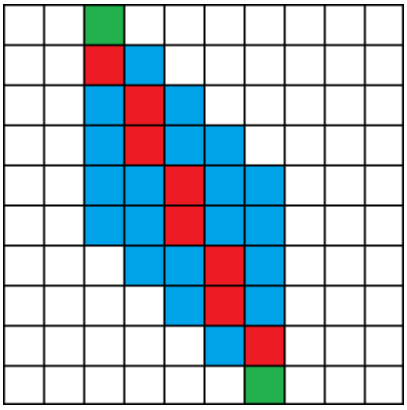


Slika 2-8 Najkraći put između početne i završne točke

3. Dodatne opcije za najkraći put

3.1. Kretanje u svim smjerovima

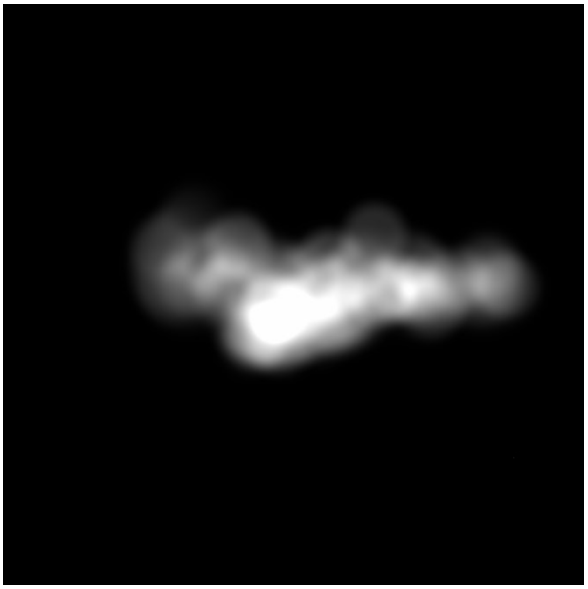
Prethodno navedeni algoritmi (od algoritama za pretraživanje grafova do algoritma transformacije udaljenosti) su algoritmi koji se kreću u osam osnovnih smjerova, što dolazi iz definicije susjedstva. Problem kod tog pristupa je što to može dovesti do više puteva koji su jednake duljine za algoritam, dok su u stvarnosti različitih duljina. To posebno dolazi do izražaja na ravnim površinama. Problem možemo prikazati na sljedećem primjeru:



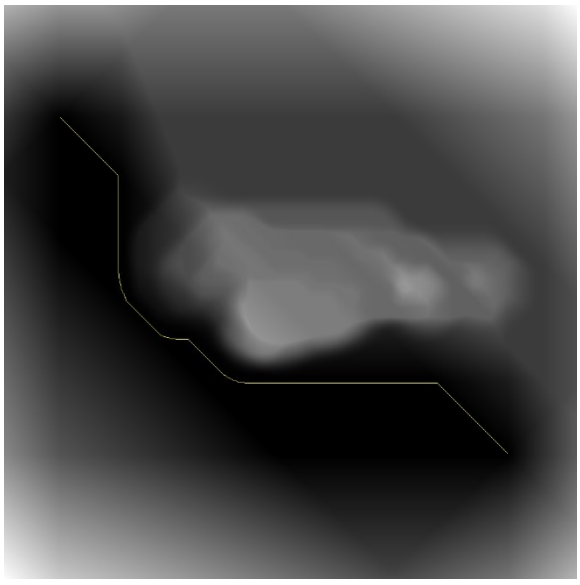
Slika 3-1 Problem kretanja u svim smjerovima

Slika prikazuje visinsku mapu koja je ravna ploha. Zelenom bojom su označene krajnje točke među kojima želimo putovati. Točke označene plavom ili crvenom bojom prikazuju točke kojima treba putovati kako bi konačni put bio jednak ako je dopušteno kretanje samo u osnovnih osam smjerova. Svaki od tih puteva radi jednake pomake, pet pomaka u smjeru dolje, i četiri pomaka u smjeru dolje desno. Iako, samo jedan od tih puteva, ako se možemo gibati u svim smjerovima, će biti najkraći (na slici označen s crvenom bojom). Taj problem možemo riješiti uz modifikaciju načina na koji pronalazimo najkraći put na slici udaljenosti. Na slici udaljenosti svi pikseli koji su označeni plavom bojom bi imali jednaku vrijednost (najmanju vrijednost na slici udaljenosti). Tada krenuvši od početnog piksela označimo sve piksele u kojima se možemo pomaknuti samo u jedan susjedni piksel, a da pripadaju skupu piksela s najkraćom udaljenosti. Ti pikseli moraju pripadati najkraćem putu, a praznine između tih piksela popunimo s dužinama. Prilikom popunjavanja s dužinama moguće su dvije situacije. Dužina čitavim svojim putem prolazi područjem najmanjih vrijednosti na slici udaljenosti i tada tu dužinu dodajemo kao dio puta. Ako dužina prolazi dijelom kroz područje koje se ne nalazi u skupu najmanjih udaljenosti, tada dužinu treba pomicati sve dok

se ne odredi točka u kojoj će dužina u potpunosti pripadati prostoru najkraćih udaljenosti, kada se nastavlja postupak dok se ne dođe do cilja. Čitav proces će biti prikazan na sljedećem primjeru.



Slika 3-2 Visinska mapa koja prikazuje planinu okruženu ravnim dijelom

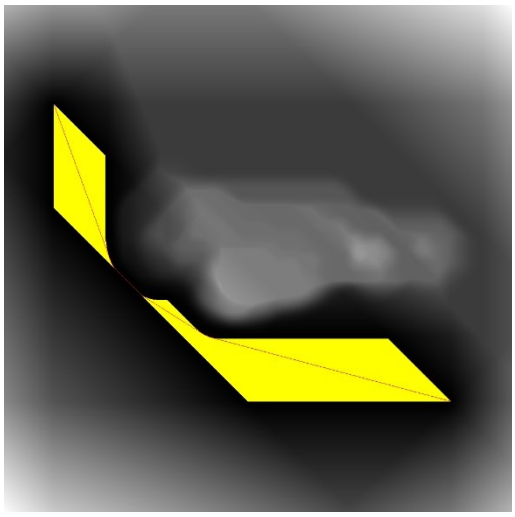


Slika 3-3 Najkraći put ako je dopušteno gibanje samo u osnovnim smjerovima

Na ovom primjeru vidimo kako put ispravno zaobilazi prepreke, međutim na ravnim dijelovima bira put koji nema optimalnu udaljenost. Kako bismo to popravili moramo pogledati područje u kojem je vrijednost slike udaljenosti jednako početnoj, odnosno završnoj točki puta.

Slika 3-4 Područje jednake vrijednosti na slici udaljenosti

Na slici vidimo dva mnogokuta povezana s pravcem. Na središnjem dijelu (pravcu) najkraći put mora prolaziti tim točkama i njih dodamo u konačni put. Kako bismo dobili najkraći put u području mnogokuta, sve što je potrebno je spojiti završne točke. Kod prvog mnogokuta (gore lijevo) to možemo napraviti jednostavno, budući da pravac koji spaja početnu točku puta s početnom točkom središnjeg pravca prolazi čitavim svojim dijelom kroz žuto područje (područje najmanje vrijednosti na slici udaljenosti). Drugi mnogokut je složeniji te za njega ne možemo s jednom dužinom spojiti krajnje točke, te je stoga potrebno dužinu podijeliti na dvije dužine koje ćemo onda pomicati dok god se obje svim svojim točkama ne budu nalazile na području najmanjih vrijednosti na slici udaljenosti. Konačni rezultat je prikazan na sljedećoj slici

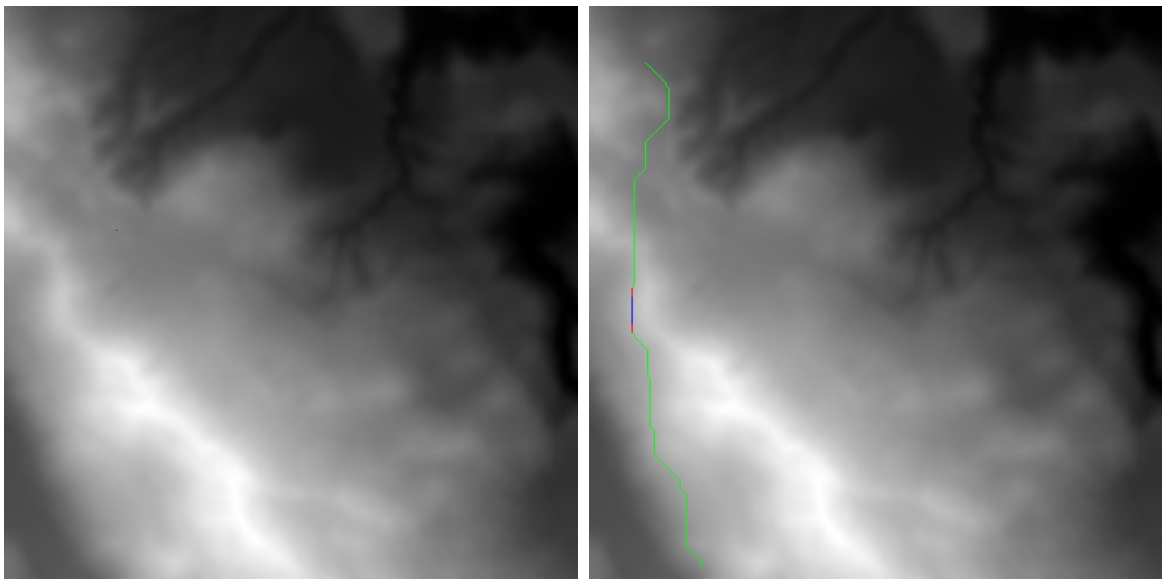


Slika 3-5 Najkraći put označen crvenom bojom ako se možemo gibati u svim smjerovima

3.2. Dodavanje tunela i mostova

Teoretski optimalan put između dvije točke u trodimenzionalnom prostoru bi se u potpunosti nalazio na pravcu koji prolazi kroz te dvije točke. Međutim to na modelu visinske mape nije moguće postići, zbog naše limitiranosti kretanja površinom. Ako bismo nekako mogli promijeniti površinu koja nam je dostupna za kretanje mogli bismo skratiti potreban put, a ta modifikacija se može ostvariti izgradnjom tunela ili mostova, što će biti obrađeno u ovome poglavlju. Izgradnja tunela ili mostova je skup postupak, stoga se u optimalnost puta uključuje faktor cijene. Cilj u ovome radu će biti pokušati smanjiti duljinu puta bez povećanja cijene (jer u slučaju da nam cijena nije bitna mogli bismo povući jednu dugačku kombinaciju mostova i tunela kako bi konačni put bio pravac između početne i završne točke). Kako bismo skratili put, prvo je potrebno razmotriti što taj put uopće čini dužim. Kako bi se izbjegle prepreke na površini put se produljuje na dva načina: dodavanjem zavoja (na xy ravnini) i promjenom nadmorske visine (u z smjeru). Dodavanje optimalnog tunela ili mosta je složen izračun, stoga ima smisla fokusirati se na ta dva načina produljenja puta. Promjena nadmorske visine na optimalnome putu se događa ako se između točaka koje se spajaju nalazi određena kotlina (ili planina), a njezin obilazak bi značio značajno produljenje puta, te prolazak kroz/preko nje daje kraći put. Kako bismo dobili najbolje mjesto za izradu tunela/mosta, potrebno je pronaći ekstreme puta (točke puta s najmanjom ili najvećom nadmorskom visinom). Zatim oko te točke odaberemo dijelove puta na kojima nadmorska visina konstantno raste, ako je ekstrem minimum, ili pada, ako je ekstrem maksimum. Jednom kada smo to odredili, odredili smo dio puta koji je dobar kandidat za izgradnju tunela ili mosta. Nakon toga odaberemo jednu točku koja se nalazi prije ekstrema, i jednu točku koja se nalazi poslije ekstrema. Izračunamo cijenu puta koji nastane izgradnjom tunela/mosta i usporedimo s cijenom bez njihove izgradnje. Ako je cijena manja, krajnje točke tunela/mosta dodatno udaljimo od točke ekstrema, a ako je cijena veća, krajnje točke približimo točki ekstrema. Prilikom odabira točaka zbog brzine izvođenja možemo se poslužiti raznim postupcima optimizacije, poput binarnog pretraživanja. Postupak provodimo sve dok nam se ne izjednače cijene gradnje s i bez tunela, ili dok ne dođemo do krajnjih točaka. Nakon toga imamo točke tunela/mosta koje će maksimalno skratiti duljinu puta uz uvjet da cijena izgradnje puta ostane jednaka. Ovaj postupak možemo promijeniti i tako da zahtijevamo da krajnje točke tunela/mosta budu na jednakoj nadmorskoj visini (što je posebno prisutno u slučaju mosta). Tada izračun mosta mijenjamo tako da odredimo prvu točku s jedne strane, a drugu točku onda dobijemo kao točku na drugoj strani koja ima

jednaku nadmorsku visinu. Nakon što smo i tu odredili visinu ostatak postupka je jednak. Postupak možemo prikazati na visinskoj mapi iz prvog poglavlja

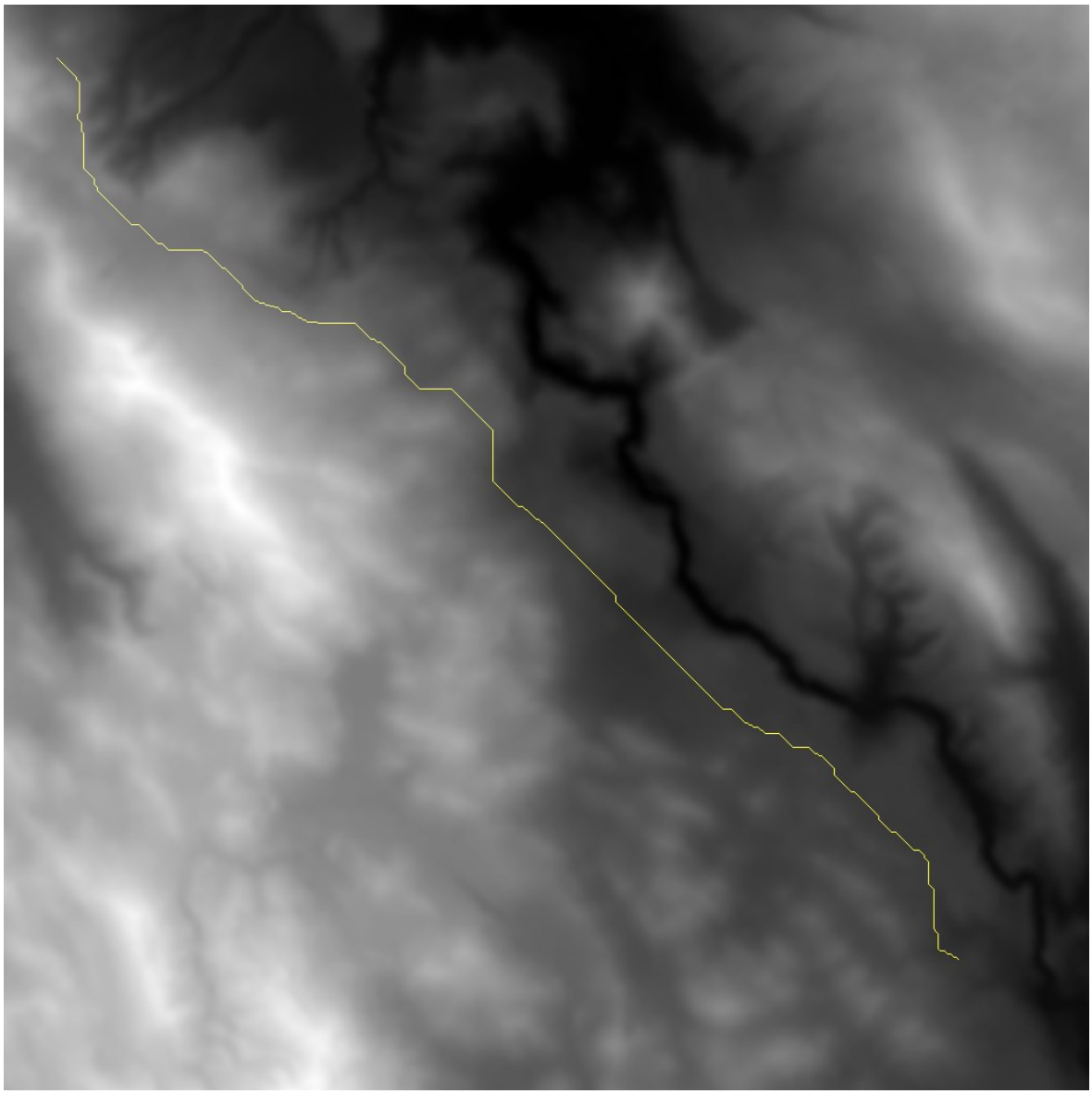


Slika 3-6 Put s tunelom na najvišem području

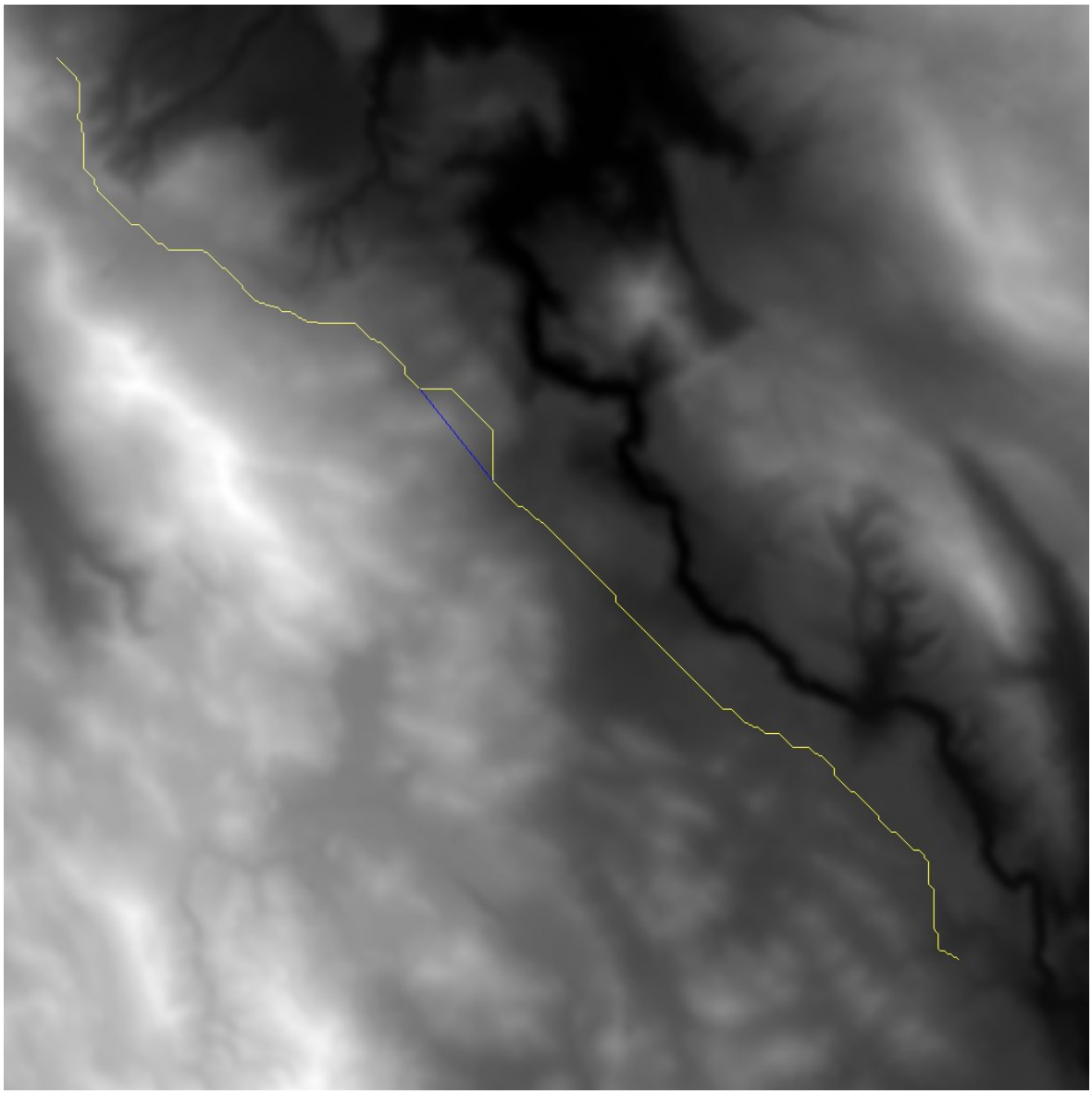
Kako bismo našli optimalan tunel prvo je bilo potrebno odrediti najbolji put pomoću algoritma transformacije udaljenosti na zakrivljenom prostoru i taj put je predstavljen sa zelenim linijama na desnoj slici. Zatim je bilo potrebno na tom putu odrediti točku koja je na najvećoj visini na tom putu. Nakon što smo to odredili bilo je potrebno u osnovnim smjerovima pogledati dio puta oko najviše točke na kojem se cesta spušta i taj dio je označen s crvenom bojom. Zatim je na tom dijelu provedena potraga za optimalnim tunelom s obzirom na duljinu i cijenu i kao rezultat je dobivena plava dužina. Tunel koji bi bio dulji od toga bi bio preskup, a tunel koji bi bio kraći nam ne bi dovoljno skratio put. U implementaciji je ovaj postupak napravljen za najvišu točku na putu (globalni ekstrem) no postupak se može napraviti i nad lokalnim ekstremima kako bismo dobili mjesta pogodna za izgradnju tunela.

Drugi način zbog kojeg se put produljuje je prisutnost zavoja. To je posebno izraženo ako se put sastoji od dionica koje imaju dugačke ravne dijelove, a uzastopni zavoji skreću u istom smjeru (lijevo ili desno). Uzimajući u obzir tu informaciju u izračunu tunela koristimo sljedeći algoritam. Generiramo segmente puta na sljedeći način. Početnu točku uzmemo kao početak prvoga segmenta. Zatim se krećemo do prvog zavoja i u njemu odredimo smjer skretanja na temelju kojeg segment označimo kao lijevi ili desni. Postupak nastavljamo do sljedećeg zavoja gdje ponovno radimo provjeru smjera. Ako je smjer skretanja isti kao prethodni zavoj nastavljamo s postupkom, a ako je smjer skretanja različit odredili smo završnu točku segmenta. Traženje sljedećeg segmenta nastavljamo na mjestu posljednjeg

zavoja prethodnog segmenta, i postupak provodimo sve dok ne dođemo do posljednje točke puta. Nakon što smo završili s izračunom segmenata izračunamo duljinu puta duž svakog segmenta i duljinu puta koji bi spajao krajnje točke segmenta (što predstavlja duljinu puta ako postoji tunel/most). Izračunamo razliku između te dvije vrijednosti i na mjestu gdje je ta razlika najveća je najbolje mjesto za izgradnju tunela.



Slika 3-7 Najkraći put na modelu visinske mape bez korištenja tunela



Slika 3-8 Najkraći put na modelu visinske mape uz korištenje tunela (označenog plavom bojom)

4. Implementacija

Za potrebe ovog rada načinjena je i implementacija u *Unreal Engine*-u koristeći programski jezik C++. Prvi korak je učitati model visinske mape u svijet. Jednom kada je visinska mapa učitana (ili ručno) stvorena *Unreal Engine* stvara površinu koja predstavlja tlo svijeta, a sastoji se od niza međusobno povezanih trokuta. U ovome radu su korišteni i ručno stvorene površine, i površine iz stvarnog svijeta. Nakon što je visinska mapa učitana može se dodati materijal tla, koji određuje izgled površine. Za potrebe ovog rada korišten je materijal koji je definiran s osnovnom bojom koja iznosi (0, 121, 0) i bez dodatnih mogućnosti kako bi dobili najbolje performanse pri iscrtavanju svijeta.

Za sam izračun puta bilo je potrebno napraviti vlastiti razred koji će sadržavati sve potrebne funkcionalnosti. Prvo je bilo potrebno izračunati sliku visinske mape za slučaj da je svijet ručno stvoren. Za to je potrebno transformirati koordinate svijeta u koordinate slike. Budući da su u radu korišteni svjetovi veličine 114200x114200 kojima je centar u ishodištu, a slika visinske mape je veličine 1024x1024, nad svijetom je trebalo izvršiti transformaciju koordinata pomoću formule $(-57100 \cdot 111 \cdot x, -57100 \cdot 111 \cdot y)$. Za izračunavanje visine terena u točki se koristi algoritam praćenja linije. Funkcija za praćenje linije dolazi ugrađena u *Unreal Engine* i poziva se sa

```
world->LineTraceSingleByObjectType(
    OUT hitResult,
    startLocation,
    endLocation,
CollisionObjectQueryParams(ECollisionChannel::ECC_WorldStatic
),
    FCollisionQueryParams());
```

Gdje su startLocation i endLocation varijable koje definiraju liniju okomitu na xy ravninu. Nakon što imamo sliku visinske mape imamo sve potrebno da bismo izračunali sliku udaljenosti. Na početku inicijaliziramo početne vrijednosti slike udaljenosti na način objašnjen u prethodnim poglavljima

```
for (int32 y = 0; y < imageSize; y++) {
    for (int32 x = 0; x < imageSize; x++) {
        int32 currentPixelIndex = ((y * imageSize) + x);
        if (x != startLocation.X || y != startLocation.Y)
image[currentPixelIndex] = FLT_MAX;
```

```
        else image[currentPixelIndex] = 0;
    }
}
```

Kada imamo početnu sliku ulazimo u algoritam implementiran na sljedeći način

```
for (int32 y = 0; y < imageSize; y++) {
    for (int32 x = 0; x < imageSize; x++) {
        int32 currentPixelIndex = ((y * imageSize) + x);
        float selfValue = OriginalImage[currentPixelIndex];
        float northWest = FLT_MAX; float north = FLT_MAX;
float northEast = FLT_MAX; float west = FLT_MAX;
        if (x > 0 && y > 0) northWest = abs(OriginalImage[(y
- 1) * imageSize + (x - 1)] - selfValue) + diagonal_move +
image[(y - 1) * imageSize + (x - 1)];
        if (y > 0) north = abs(OriginalImage[(y - 1) *
imageSize + x] - selfValue) + 111 + image[(y - 1) * imageSize
+ x];
        if (y > 0 && x < 1023)  northEast =
abs(OriginalImage[(y - 1) * imageSize + (x + 1)] - selfValue)
+ diagonal_move + image[(y - 1) * imageSize + (x + 1)];
        if (x > 0) west = abs(OriginalImage[y * imageSize +
(x - 1)] - selfValue) + 111 + image[y * imageSize + (x - 1)];
        image[curPixelIndex] =
FMath::Min(image[currentPixelIndex], FMath::Min(northWest,
FMath::Min(north, FMath::Min(northEast, west))));
    }
}
for (int32 y = imageSize - 1; y >= 0; y--) {
    for (int32 x = imageSize - 1; x >= 0; x--) {
        int32 currentPixelIndex = ((y * imageSize) + x);
        float selfValue = OriginalImage[currentPixelIndex];
        float southWest = FLT_MAX; float south = FLT_MAX;
float southEast = FLT_MAX; float east = FLT_MAX;
        if (x < 1023 && y < 1023) southEast =
abs(OriginalImage[(y + 1) * imageSize + (x + 1)] - selfValue)
+ diagonal_move + image[(y + 1) * imageSize + (x + 1)];
        if (y < 1023) south = abs(OriginalImage[(y + 1) *
imageSize + x] - selfValue) + 111 + image[(y + 1) * imageSize
+ x];
```

```
        if (y < 1023 && x > 0) southWest =
abs(OriginalImage[(y + 1) * imageSize + (x - 1)] - selfValue)
+ diagonal_move + image[(y + 1) * imageSize + (x - 1)];
        if (x < 1023) east = abs(OriginalImage[y * imageSize
+ (x + 1)] - selfValue) + 111 + image[y * imageSize + (x +
1)];
        image[curPixelIndex] =
FMath::Min(image[currentPixelIndex], FMath::Min(southEast,
FMath::Min(south, FMath::Min(southWest, east))));
    }
}
```

Iz slike udaljenosti put generiramo na sljedeći način

```
int pos_x = startPosition.X; int pos_y = startPosition.Y;
positions.Add(FVector2D(pos_x, pos_y));
FVector2D directions[8]{
    {1, 0}, {1, 1}, {0, 1}, {-1, 1}, {-1, 0}, {-1, -1}, {0, -
1}, {1, -1}
};
int32 curPixelIndex = ((pos_y * imageSize) + pos_x);
float min_value = FLT_MAX;
int direction = 0;
for (int i = 0; i < 8; i++) {
    int32 otherPixelIndex = curPixelIndex + directions[i].Y *
imageSize + directions[i].X;
    if (image[otherPixelIndex] < min_value) {
        min_value = image[otherPixelIndex];
        direction = i;
    }
}
pos_x += directions[direction].X;
pos_y += directions[direction].Y;
positions.Add(FVector2D(pos_x, pos_y));
while (pos_x != endPosition.X || pos_y != endPosition.Y) {
    min_value = FLT_MAX;
    curPixelIndex = ((pos_y * imageSize) + pos_x);
    int min_direction = direction;
    for (int i = -1; i < 2; i++) {
        int dir = (direction + i) % 8;
        if (dir == -1) dir = 7;
```

```
            int32 otherPixelIndex = curPixelIndex +
    directions[dir].Y * imageSize + directions[dir].X;
            if (i == 0) {
                if (image[otherPixelIndex] < min_value) {
                    min_value = image[otherPixelIndex];
                    min_direction = dir;
                }
            }
            else {
                if (image[otherPixelIndex] + turnCoefficient <
    min_value) {
                    min_value = image[otherPixelIndex] +
    turnCoefficient;
                    min_direction = dir;
                }
            }
        }
        if (min_direction != direction) {
            turns.Add(FVector(pos_x, pos_y, min_direction));
        }
        direction = min_direction;
        pos_x += directions[direction].X;
        pos_y += directions[direction].Y;
        positions.Add(FVector2D(pos_x, pos_y));
    }
```

Krenuvši od početne točke puta tražimo susjednu točku koja ima najmanju vrijednost na slici udaljenosti i nakon toga u tom smjeru nastavljamo postupak. Pod tim smjerom smatramo nastavak puta ravno ili skretanje za 45° u nekom smjeru. Ako za dolazak do najmanje vrijednosti treba napraviti zavoj dodajemo ga u listu zavoja, a u slučaju da više susjeda ima jednaku vrijednost odabiremo onog koji ide ravno, budući da za skretanje vrijednosti udaljenosti dodajemo pozitivni koeficijent.

Izračun tunela i mostova je ostvaren pomoću sljedećih funkcija

```
    for (FVector2D vec : positions) {
        float pointHeight = this->getMapHeightAtPoint(FVector2D(-
    57100 + 111 * vec.X, -57100 + 111 * vec.Y));
        if (pointHeight > maxHeight) {
            highestPoint.Set(vec.X, vec.Y);
            maxHeight = pointHeight;
        }
```

```
}
pointsAroundHighest.Add(highestPoint);
int direction = 0;
FVector2D nextPoint = FVector2D(highestPoint.X,
highestPoint.Y);
for (int i = 0; i < 8; i++) {
    nextPoint.Set(highestPoint.X + directions[i].X,
highestPoint.Y + directions[i].Y);
    if (positions.Contains(nextPoint) && this-
>getMapHeightAtPoint(FVector2D(-57100 + 111 * nextPoint.X, -
57100 + 111 * nextPoint.Y)) < this-
>getMapHeightAtPoint(FVector2D(-57100 + 111 * highestPoint.X,
-57100 + 111 * highestPoint.Y)) + 1) {
        pointsAroundHighest.Add(FVector2D(nextPoint.X,
nextPoint.Y));
        while (true) {
            FVector2D anotherPoint = FVector2D(nextPoint.X +
directions[i].X, nextPoint.Y + directions[i].Y);
            if (positions.Contains(anotherPoint) && this-
>getMapHeightAtPoint(FVector2D(-57100 + 111 * anotherPoint.X,
-57100 + 111 * anotherPoint.Y)) <= this-
>getMapHeightAtPoint(FVector2D(-57100 + 111 * nextPoint.X, -
57100 + 111 * nextPoint.Y)) + 1) {
                nextPoint = anotherPoint;

pointsAroundHighest.Add(FVector2D(nextPoint.X, nextPoint.Y));
            } else break;
        }
    }
}
float lengthWithoutTunel = 0.0;
for (FVector2D vec : pointsAroundHighest) {
    float heightDiffence = getMapHeightAtPoint(FVector2D(-
57100 + 111 * vec.X, -57100 + 111 * vec.Y)) -
getMapHeightAtPoint(FVector2D(-57100 + 111 * lastVector.X, -
57100 + 111 * lastVector.Y));
    float length = sqrt(12321 + heightDiffence *
heightDiffence);
    lengthWithoutTunel += length;
}
float lengthWithTunel = 0.0;
```

```
FVector2D startPoint = FVector2D(pointsAroundHighest[0].X,
pointsAroundHighest[0].Y);
FVector2D endPoint =
FVector2D(pointsAroundHighest[pointsAroundHighest.Num() -
1].X, pointsAroundHighest[pointsAroundHighest.Num() - 1].Y);
float heightDiff = getMapHeightAtPoint(FVector2D(-57100 + 111
* startPoint.X, -57100 + 111 * startPoint.Y)) -
getMapHeightAtPoint(FVector2D(-57100 + 111 * endPoint.X, -
57100 + 111 * endPoint.Y));
lengthWithTunel = sqrt((111 * abs(endPoint.Y - startPoint.Y))
* (111 * abs(endPoint.Y - startPoint.Y)) + heightDiff *
heightDiff);
if (lengthWithoutTunel < lengthWithTunel * tunelPrice)
tunelPoints = pointsAroundHighest;
```

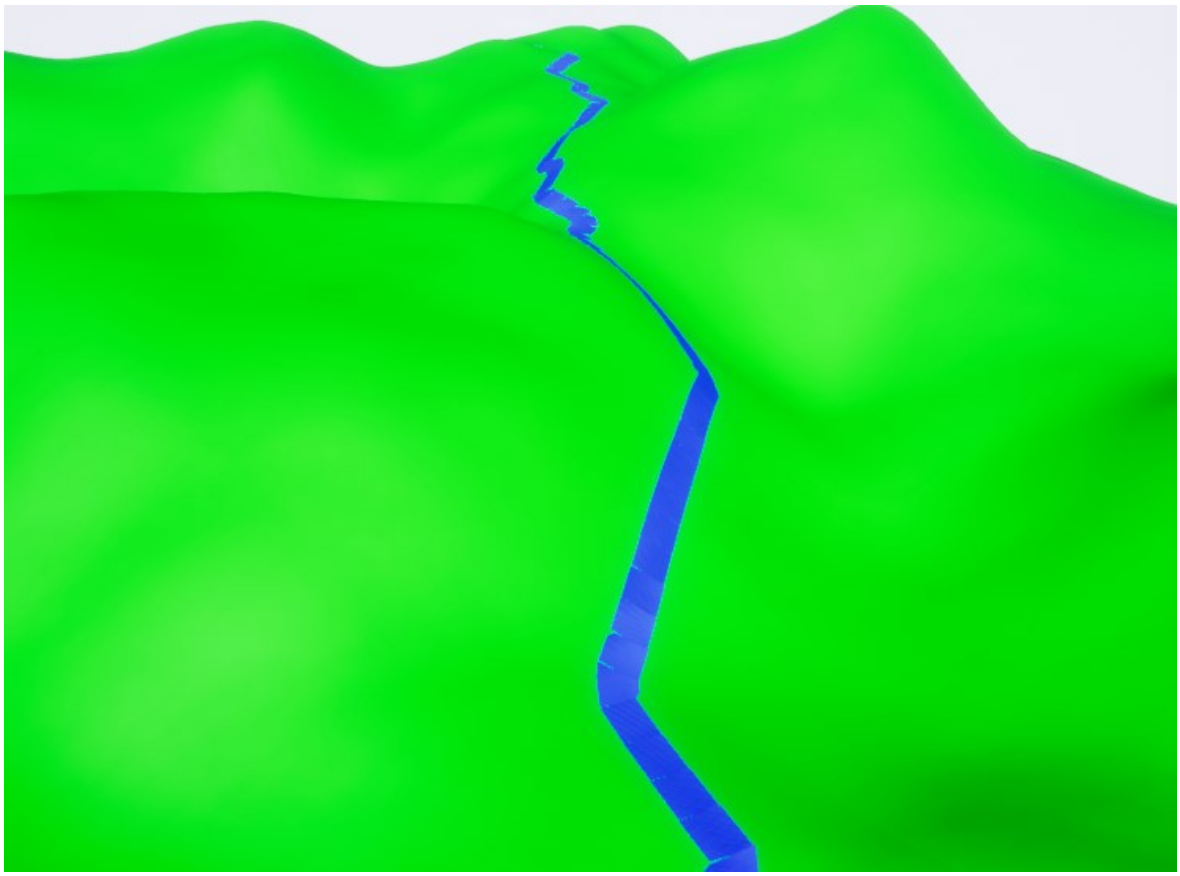
Algoritam najprije nađe najvišu točku (globalni ekstrem), zatim nađe sve susjede koji pripadaju putu i vrijednost visine im pada. Kada je pronađena lista točaka za te točke se usporedi duljina puta bez i s tunelom i ako je tunel isplativ stvorimo ga. Drugi način izračuna tunela prikazuje sljedeći kod

```
FVector firstTurn = turns[0];
int firstDirection = firstTurn.Z;
int directionChange = 0;
int lastDirectionChange = 0;
int lastDirection = firstDirection;
FVector beginningOfTurn = FVector(firstTurn.X, firstTurn.Y,
firstTurn.Z);
TArray<TPair<FVector, FVector>> endPoints;
for (int i = 1; i < turns.Num(); i++) {
    directionChange = lastDirection - turns[i].Z;
    if (lastDirectionChange != directionChange) {
        lastDirectionChange = directionChange;
        endPoints.Add(TPair<FVector,
FVector>(FVector(beginningOfTurn.X, beginningOfTurn.Y,
beginningOfTurn.Z), FVector(turns[i].X, turns[i].Y,
turns[i].Z)));
        beginningOfTurn = FVector(turns[i - 1].X, turns[i -
1].Y, turns[i - 1].Z);
        i--;
    }
    lastDirection = turns[i].Z;
}
```

```
TPair<FVector2D, FVector2D> bestPair;
float bestValue = 0;
for (TPair<FVector, FVector> pair : endPoints) {
    FVector2D s = FVector2D(pair.Key.X, pair.Key.Y);
    FVector2D e = FVector2D(pair.Value.X, pair.Value.Y);
    float lengthWithoutTunel =
calculatePathDistanceBetweenTwoPoints(s, e);
    float heightDiff = getMapHeightAtPoint(FVector2D(-57100 +
111 * s.X, -57100 + 111 * s.Y)) -
getMapHeightAtPoint(FVector2D(-57100 + 111 * e.X, -57100 +
111 * e.Y));
    float lengthWithTunel = sqrt((111 * abs(e.Y - s.Y)) *
(111 * abs(e.Y - s.Y)) + heightDiff * heightDiff);
    float tunelValue = lengthWithoutTunel - lengthWithTunel;
    if (pathValue > bestValue) {
        bestValue = pathValue;
        bestPair = TPair<FVector2D, FVector2D>(FVector2D(s.X,
s.Y), FVector2D(e.X, e.Y));
    }
}
turnTunnel = getPointsBetween(bestPair.Key, bestPair.Value);
```

Prvi korak je izračun segmenata puta u kojima se ide ravno, ili skreće samo u jednom smjeru. Zatim za te segmente izračunamo duljine puta bez i s tunelom i odaberemo onaj tunel s kojim se put najviše skraćuje.

Vizualizacija puta je obavljena pomoću ugrađene komponente *Unreal Engine*-a USplineComponent. Ona predstavlja točke definirane Bézierovom krivuljom koja nam može poslužiti za vizualizaciju puta. Kako bismo iscrtali put koristimo mrežu poligona koja se sastoji od osam trokuta postavljenih u ravninu. Za prikazivanje je također potrebno definirati materijal koji je kao i površina visinske mape definiran samo s osnovnom bojom. Krivulja prolazi kroz točke puta, za vrijednosti točke ponovno koristimo algoritam praćenja linije, ali ovog puta osim same vrijednosti lokacije sudara, koristimo i kut, kako bismo postavili nagib iscrtavanja.



Slika 4-1 Vizualizacija najkraćeg puta na modelu visinske mape

5. Analiza složenosti algoritma i moguća poboljšanja

Algoritam je testiran nad simulacijama površine veličine 114200x114200 piksela. Za izračun trajanja puta je bilo potrebno do nekoliko sekundi, i to uključuje i puteve koji prolaze od jednog do drugoga kuta mape. Što se tiče prostorne složenosti, za izračun puta je potrebna lista visina visinske mape, te dvije pomoćne liste koje su jednake veličine kao i lista visinske mape. Prostorna složenost je konstantna, bez obzira na karakteristike konačnoga puta. Pronalazak tunela za najvišu točku puta nema veliki utjecaj na performanse, a za pronalazak tunela između zavoja je potrebno nekoliko sekundi za složenije primjere.

Ako bismo željeli poboljšati performanse algoritma određene dijelove koda možemo izvoditi paralelno. Primjerice, izračun slike udaljenosti za početnu i završnu točku su potpuno neovisni postupci, te stoga njih možemo izvoditi u isto vrijeme. Također, za svaku točku izračun između te točke i svakog od njezinih susjeda možemo provoditi paralelno. Zbroj slika udaljenosti za početnu i završnu točku predstavlja matrično zbrajanje te se stoga on može provesti na procesoru koji podržava izvođenje jedne instrukcije nad više podataka, poput grafičkog procesora. Osim toga, prilikom izračuna slike udaljenosti za svaku točku na početku dok nismo došli do početne točke sve vrijednosti su postavljene na $+\infty$, te će stoga i minimumi iznositi $+\infty$ zbog čega izračun za taj dio slike ne moramo ni provoditi. To posebno dolazi do izražaja ako nam se početna točka nalazi u donjem desnom kutu slike, kada prvu iteraciju algoritma možemo uvelike skratiti. Kada računamo sliku udaljenosti za početnu točku, i na toj slici izračunamo vrijednost udaljenosti za završnu točku puta, nakon toga svaku vrijednost koja je veća od vrijednosti udaljenosti za završnu točku možemo ignorirati, budući da ćemo birati minimume, a ako je vrijednost veća od vrijednosti za završnu točku (koja bi na konačnoj slici udaljenosti bila 0 + vrijednost za tu točku na početnoj slici) na konačnoj slici bismo dobili veću vrijednost koju ne bismo uzimali u obzir za konačni put. To može skratiti trajanje računanja puta za kratke puteve. Osim toga, ako koristimo opcije skraćivanja puta pomoću mostova i tunela u memoriju možemo spremiti određene vrijednosti puta koje onda kasnije ne bismo trebali ponovno računati.

Zaključak

Pronalaženje najkraćeg puta je jedan od čestih optimizacijskih problema. Ovaj rad je prikazao rješavanje tog problema u trodimenzionalnom prostoru na modelu visinske mape. Kroz rad su prikazane teoretske postavke raznih algoritama i njihove dobre i loše karakteristike. Rad donosi i pregled proširenja koja se mogu načiniti na putu, poput dodavanja tunela ili mostova. Uz rad je načinjena i osnovna implementacija prikazanih algoritama koristeći *Unreal Engine* i programski jezik C++, te je prikazana vremenska i prostorna složenost algoritama uz savjete o poboljšanju implementacije. Izračun optimalnog puta na modelu visinske mape je i dalje otvoreni problem na kojem je moguće ostvariti poboljšanja, naročito u smislu brzine izračuna, te dodavanja tunela ili mostova na putu.

Literatura

[1] Ikonen, L., Toivanen, P. *Shortest Route on Height Map Using Gray-Level Distance Transforms*. Discrete Geometry for Computer Imagery, Napulj, (2003.), str. 308-316

[2] Saha, P. K., Wehrli, F. W., Gomberg, B. R. *Fuzzy Distance Transform: Theory, Algorithms and Applications*. Computer Vision and Image Understanding 86, 3 (2002.) str. 171-190.

[3] Toivanen, P. *New geodesic distance transforms for gray-scale images*. Pattern Recognition Letters 17, 5 (1996.) str. 437-450.

[4] Wichmann, D. R., Wuensche, B. *Automated route finding on digital terrains* (2004.)

[5] Epic Games, Unreal Engine 5.2 Documentation, Poveznica: https://docs.unrealengine.com/5.2/en-US/; Pristupljeno: 13. svibnja 2023.

[6] Komljenović, J. *Prikaz proširenih krivulja na modelu visinske mape*. Završni rad, Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva, 2021.

Sažetak

Rad prikazuje rješavanje problema izračuna optimalnog puta na modelu visinske mape. U radu su obrađeni razni algoritmi za taj problem, od čega je detaljno opisan algoritam transformacije udaljenosti na modelu visinske mape. Također je prikazano kako mogućnost izgradnje tunela i mostova može utjecati na konačni izračun optimalnog puta. Prikazani su i različiti načini vizualizacije optimalnog puta. Uz rad dolazi i programska potpora napravljena u *Unreal Engine*-u u programskom jeziku C++ kao i analiza prostorne i vremenske složenosti postupka.

Ključne riječi: optimalni put, optimiranje, visinska mapa, planiranje rute

Summary

The paper presents the solution to the problem of calculating the optimal path on the heightmap model. It deals with various algorithms for this problem, of which the distance transformation algorithm based on the height map model is described in detail. It is also shown how the possibility of building tunnels and bridges can affect the final calculation of the optimal route. Different ways of visualizing the optimal path are also shown. The work is accompanied by software support made in Unreal Engine in the C++ programming language, as well as an analysis of the spatial and temporal complexity of the procedure.

Keywords: optimal path, optimization, height map, route planning

Privitak

Uz ovaj rad dolazi i programski kod koji je dostupan na sljedećem linku: https://github.com/MightyJosip/DiplomskiRad. Za pokretanje programskog koda potrebno je na računalu imati instaliran *Unreal Engine* (rad je testiran na verziji 4.26)