

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1597

Određivanje putanje ceste na terenu

Marko Nađ

Zagreb, lipanj 2018.

Zagreb, 9. ožujka 2018.

DIPLOMSKI ZADATAK br. 1597

Pristupnik: **Marko Nad (0036479641)**
Studij: Računarstvo
Profil: Računarska znanost

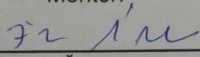
Zadatak: **Određivanje putanje ceste na terenu**

Opis zadatka:

Proučiti postupke generiranja terena i postavljanja vegetacije temeljem biotskih mapa. Proučiti postupke određivanja trase kojom će prolaziti cesta na terenu. Razraditi proučene postupke te za zadane točke povezivanja ceste, zadanu konfiguraciju terena, niz projektnih pravila i restriktivskih uvjeta ostvariti određivanje putanje ceste uz primjenu optimizacijskih algoritama. Analizirati i ocijeniti ostvarene rezultate. Diskutirati utjecaj različitih parametara. Izraditi odgovarajući programski proizvod. Koristiti programski jezik Java i grafičko programsko sučelje OpenGL. 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.

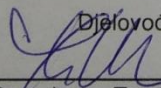
Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 29. lipnja 2018.

Mentor:



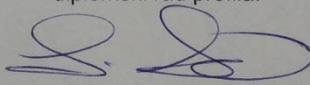
Prof. dr. sc. Željka Mihajlović

Djelovoda:



Doč. dr. sc. Tomislav Hrkać

Predsjednik odbora za
diplomski rad profila:



Prof. dr. sc. Siniša Sribljčić

Zahvaljujem mentorici prof. dr. sc. Željki Mihajlović na pruženoj pomoći, idejama i smjernicama tijekom studija. Omogućili ste mi veliku slobodu izbora i uveli me u područje koje me doista zanima.

Sadržaj

Uvod.....	1
1. Teren.....	2
1.1. Algoritam ROAM.....	3
1.2. Algoritam GeoMipMapping	7
1.3. Struktura Geometry Clipmaps.....	10
2. Pretraga prostora stanja	14
2.1 Slijepo pretraživanje	16
2.1.1 Pretraživanje u širinu BFS	16
2.1.2 Pretraga s jednolikom cijenom UCS.....	17
2.1.3 Pretraga u dubinu DFS	18
2.2 Heurističko pretraživanje	19
2.2.1 Pohlepni algoritam Greedy best-first search	19
2.2.2 Algoritam A*	20
3. Razvijeni sustav	24
3.1 Generiranje terena	24
3.1.1 Poligonalna mreža	24
3.1.2 Visinska funkcija	28
3.1.3 Tekstura terena	30
3.2 Postavljanje vegetacije.....	31
3.2 Generiranje ceste	37
3.2.1 Generiranje geometrije ceste i tunela.....	37
3.2.2 Određivanje trajektorije ceste.....	40
Zaključak	54
Literatura	55
Sažetak.....	56
Abstract	56

Uvod

Prilikom izbora trajektorije ceste potrebno je odrediti ključne zahtjeve i razmotriti niz ograničenja i kriterija. Ako nastojimo povezati dvije lokacije uz najkraće trajanje putovanja, naglasak će biti postavljen na kriterij duljine trajektorije. Ako je, s druge strane, cilj minimizirati ukupnu potrošnju goriva, preferirat će se trajektorije minimalnih visinskih oscilacija. Dvije lokacije je moguće povezati na velik broj načina i pritom je potrebno uvažiti ograničenja što problem pronalaska optimalne trajektorije čini kompleksnim.

U ovom radu je opisan sustav čiji ulaz je proizvoljan teren za kojeg se generira optimalna trajektorija automatski na temelju zadanih ograničenja i kriterija duljine, nagiba i zakrivljenosti. Prvo poglavlje opisuje postupke zapisa i prikaza terena u računalu. U drugom poglavlju opisuje se teorija iza algoritama pretrage na kojima se temelji izračun. Treće poglavlje opisuje implementaciju sustava.

1. Teren

Prikaz terena je složen problem za čije rješavanje su osmišljeni mnogi raznovrsni algoritmi. Kvalitetan sustav za prikaz terena je u stanju prikazati velike (ili čak beskonačne) terene zadržavajući pritom visoku frekvenciju osvježavanja prikaza. Takav sustav prikazuje uvjerljiv model uz dana sklopovska ograničenja i neizbježan šum u podacima, nastao zbog nesavršenosti mjernih instrumenata ili smanjene preciznosti zbog ograničenog prostora za pohranu.

Sustav za prikaz terena mora definirati reprezentaciju terena u memoriji računala. Tipično se za tu svrhu koriste visinske mape. Visinska mapa je rasterska slika čiji pikseli sadrže visinsku informaciju za pojedine točke određenog pravokutnog područja. Visinska informacija se obično pohranjuje u R ili R i G kanale slike, čime se postiže rezolucija od 2^8 , odnosno 2^{16} visinskih razina, što je dovoljno za većinu primjena. Veće preciznosti se mogu postići spremanjem dodatnih informacija u B i *alpha* kanale, a moguće je i definirati proizvoljan format podataka, međutim dva kanala su najčešće dovoljna. Kvaliteta visinske mape ovisi, osim o preciznosti, i o rezoluciji slike. Kako se visinska informacija za točke čija visina nije definirana visinskom mapom (točke „između“ piksela) mora odrediti nekom vrstom interpolacije, poželjno je da rezolucija bude što veća jer je tada preciznost veća i rezultat je bliži stvarnom stanju. Satelitske snimke s rezolucijom od trideset metara su u današnje vrijeme slobodno dostupne. Postoje i snimke s većom rezolucijom, no takve snimke su predviđene pretežito za komercijalne svrhe ili vojne simulacije.

Visinsku informaciju nije nužno predstavljati isključivo pomoću rasterskih slika. Također je moguće definirati proizvoljnu funkciju koja preslikava uređeni par koordinata u visinu. Jedno od svojstava takvog zapisa informacije jest da zauzima značajno manje memorije, a moguće je definirati i funkcije nad kontinuiranom i beskonačnom domenom. Najčešće se radi o funkcijama šuma, kao što su primjerice fraktalni šum, Perlinov šum i njegova unaprijeđena inačica, *Simplex* šum.

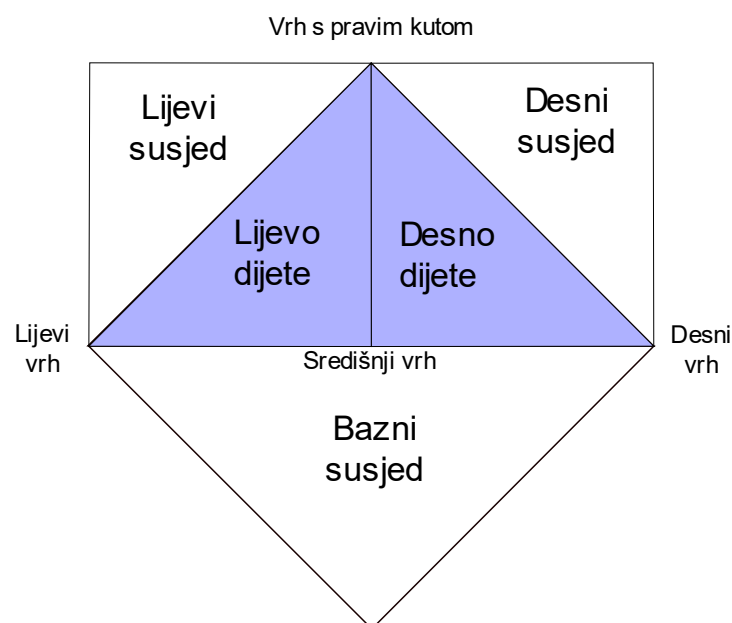
Visinska informacija se odabranim algoritmom transformira u mrežu poligona. Dobiveni model mora sadržavati što je moguće veću količinu informacije uz što manji broj poligona. Prikaz čitavog skupa podataka vrlo brzo postaje neostvariv s povećanjem skupa i stoga je potrebno na neki način doskočiti ovom problemu. Poželjno je da model bude detaljniji u blizini promatrača, a manje detaljan u daljini gdje detalji ionako nisu primjetni. Različiti algoritmi rješavaju ovaj zahtjev

na različite načine, a nekoliko odabranih algoritama s njihovim svojstvima je opisano u nastavku.

1.1. Algoritam ROAM

Real-time Optimally Adapting Meshes (ROAM) je algoritam za prikaz terena nastao u doba kada programiranje grafičkog sklopovlja nije bilo podržano. *ROAM* nastoji povećati količinu poligona na neravnijim dijelovima terena kako bi postigao vjerniji prikaz, i smanjiti na ravnijim i udaljenim dijelovima s ciljem poboljšanja performansi. Poligonalna mreža se definira iznova prije svakog kadra i šalje na GPU za prikaz.

Temeljna struktura potrebna za rad algoritma je binarno stablo trokuta (eng. *Binary Triangle Tree*). Radi se o hijerarhijskoj strukturi čiji čvorovi predstavljaju jednakokračne pravokutne trokute koji će se iscrtati. Svaki čvor stabla sadrži pet pokazivača: pokazivače na lijevo i desno dijete, lijevog i desnog susjeda te baznog susjeda. Ovi odnosi su vidljivi na sljedećoj slici.



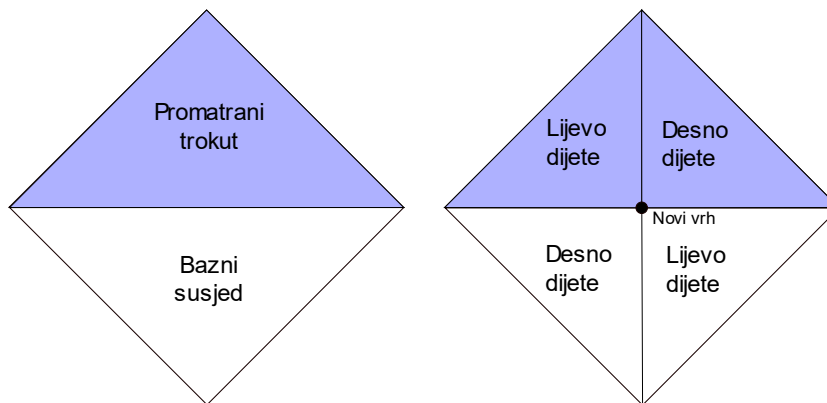
Slika 1.1. Trokut s pripadnom djecom i susjedima.

Algoritam radi na principu iterativnog rafiniranja početne mreže, a početna mreža se sastoji od samo jednog jednakokračnog pravokutnog trokuta. Taj trokut tada predstavlja neko područje terena i pridružen mu je odgovarajući dio visinske

mape. Jedan trokut je očekivano pregruba aproksimacija visinske mape, stoga će se dijeliti na nove trokute dok se ne postigne zadovoljavajuća preciznost. Trokut za koji je utvrđeno da ga je potrebno prepoloviti, dijeli se na dva nova trokuta po dužini koja spaja vrh na polovištu hipotenuze s vrhom s pravim kutom, odnosno trokut dobiva svoje lijevo i desno dijete. Inicijalni trokut (roditelj) se neće iscrtati, već samo njegova djeca.

Potrebno je napomenuti da struktura stabla ne pamti nikakve informacije o vrhovima trokuta, kao što su prostorne koordinate, normale i slično – svaki čvor stabla sadrži samo pet spomenutih pokazivača na druge čvorove, a podaci za svaki vrh se izračunavaju nakon što je čitavo stablo generirano, odnosno podaci o vrhovima su spremljeni implicitno u strukturi stabla. Primjerice, ako je početni trokut (predstavljen početnim čvorom koji nema nijedan inicijalizirani pokazivač) potrebno podijeliti samo jednom, tada nastaje binarno stablo s jednim korijenom i dva lista. Dobivena dva lista predstavljaju djecu (to su čvorovi bez inicijaliziranih pokazivača), a korijenski čvor dobiva pokazivače na djecu, i predstavlja trokut kojeg više nije potrebno iscrtati, odnosno samo listovi predstavljaju trokute koji čine konačnu mrežu. Nakon što je stablo generirano, započinje faza izračuna koordinata vrhova trokuta. U početku su poznate jedino koordinate vrhova početnog trokuta – to su koordinate u svjetskom prostoru vrhova područja kojeg taj trokut predstavlja. Stablo se potom rekurzivno obilazi sve dok se ne obrade svi listovi. Izračunaju se koordinate vrha na polovištu hipotenuze te se odgovarajuće koordinate šalju djeci. Kad se naiđe na list, trokut kojeg predstavlja se dodaje u konačnu mrežu.

Kad bi se trokuti dijelili neovisno o susjednim trokutima, na nekim mjestima u mreži bi nastajale pukotine. Pukotina nastaje kada se trokut prepolovi, a njegov bazni susjed ne. Pukotinu tada čini nepostojeći trokut čiji vrhovi su lijevi i desni vrh polaznog trokuta te novi vrh. Kako bi se pojava pukotina spriječila, prilikom dijeljenja trokuta će se zahtijevati dijeljenje i njegovog baznog susjeda, prema sljedećoj slici.

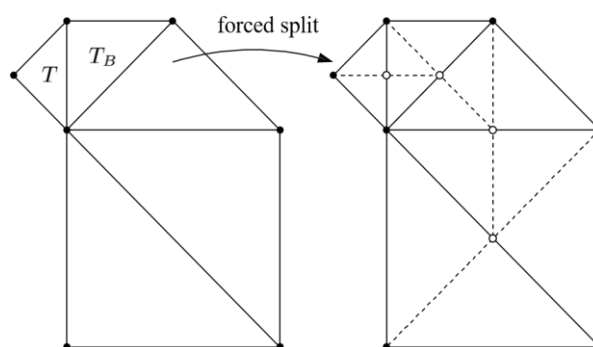


Slika 1.2. Dijeljenje baznog susjeda uslijed dijeljenja polaznog trokuta.

Ovaj odnos se naziva *dijamantom*, i predstavlja osnovni slučaj lančanog dijeljenja trokuta. Kako bi se pukotine eliminirale u potpunosti, potrebno je razriješiti tri moguća slučaja:

1. Trokuti su u odnosu *dijamant*: osnovni slučaj – podjela kao što je opisano.
2. Trokut koji se dijeli je na rubu mreže: samo taj trokut se dijeli.
3. Trokut i njegov bazni susjed nisu u odnosu *dijamant*: poziva se procedura prisilnog dijeljenja baznog susjeda (eng. *Force split*).

Prisilno dijeljenje je operacija rekurzivnog obilaska mreže dok se ne naiđe na *dijamant* ili rubni trokut. Ako izvorni trokut nije u odnosu *dijamant* sa svojim baznim susjedom, nad njime se poziva operacija dijeljenja. Ako je taj bazni susjed na rubu mreže ili u odnosu *dijamant* sa svojim baznim susjedom, izvršava se podjela prema slučaju 1, odnosno 2, a u suprotnosti se dijeljenje rekurzivno nastavlja. Na kraju rekurzije će se polazni trokut naći u odnosu *dijamant* sa svojim baznim susjedom te ih je tada moguće podijeliti bez nastanka pukotine. Jedan mogući ovakav slučaj je ilustriran na sljedećoj slici.



Slika 1.3. Prisilno dijeljenje baznog susjeda ([1]).

Odabrana metrika za donošenje odluke o dijeljenju trokuta je maksimalna geometrijska pogreška u prostoru slike. Svakom vrhu trokuta pridružena je visina koja se u općem slučaju razlikuje od stvarne visine pohranjene u visinskoj mapi. Ta razlika u visini se naziva geometrijskom pogreškom neovisnom o pogledu. Maksimalna geometrijska pogreška trokuta je maksimalna takva vrijednost među vrijednostima za sva tri vrha. U fazi pred-procesiranja ove vrijednosti se računaju za sve čvorove i pohranjuju. U slučaju promjene visinske mape, geometrijske pogreške se moraju izračunati iznova. Kada je potrebno donijeti odluku hoće li se neki trokut podijeliti ili ne, dohvaća se prethodno izračunata geometrijska pogreška te se na temelju nje i udaljenosti od promatrača određuje pogreška u prostoru slike. Ako je pogreška u prostoru slike veća od unaprijed zadane granice, trokut se dijeli.

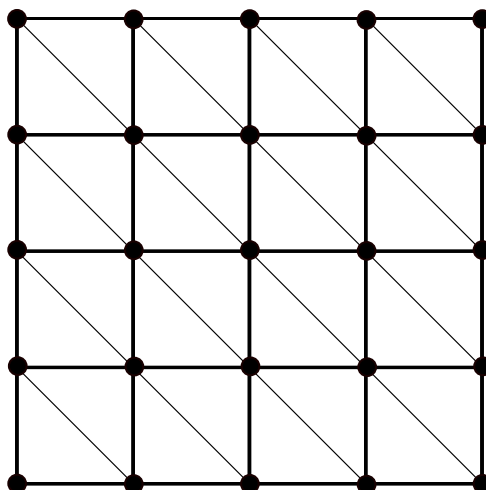
Kako bi se izbjeglo generiranje mreže iznova prije crtanja svakog kadra, *ROAM* koristi mrežu iz prethodnog kadra kao polaznu točku za novu mrežu, i definira dva prioriteta reda pomoću kojih se prethodna mreža modificira. Prvi red služi rafiniranju mreže i sadrži trokute poredane prema mjeri pogreške. Trokuti iz tog reda se dijele čime nastaje detaljnija mreža dok pogreška svih trokuta ne padne ispod zadane granične vrijednosti, ili dok mreža ne postane prevelika. Drugi prioritetni red sadrži parove trokuta koje je moguće spojiti (dijamante) i služi smanjenju broja poligona na mjestima gdje je mreža predetaljna. Pogreška dijamanta se određuje kao maksimalna pogreška njegovih trokuta, a prioritet dijamanta je obrnuto proporcionalan pogrešci, tako da se najprije spajaju oni trokuti koji će unijeti najmanju pogrešku.

Na kraju ćemo spomenuti nekoliko karakterističnih svojstava *ROAM*-a. Čitava mreža se generira na centralnom procesoru, a grafička procesorska jedinica ni na koji način ne utječe na mrežu, što čini ovaj algoritam vrlo procesorski intenzivnim. Ovaj pristup za posljedicu ima još jedan nedostatak – prije svakog kadra je potrebno prenijeti cijelu mrežu na GPU. Potrebna je faza pred-procesiranja u kojoj se generiraju binarno stablo trokuta i odgovarajuća struktura za pohranu pogrešaka trokuta koje je potrebno čuvati u memoriji za vrijeme izvođenja. Pogreška poligona proizvedene mreže je garantirano manja od zadane granične vrijednosti, a maksimalni broj poligona je moguće jednostavno podešavati.

1.2. Algoritam GeoMipMapping

Ovaj algoritam kao i mnogi njegovi prethodnici poput *ROAM*-a nastoji proizvesti prikaz terena koji je detaljan blizu promatrača, a manje detaljan u daljini. Dodatni cilj je iskoristiti mogućnosti grafičkog sklopovlja i u što većoj mjeri rasteretiti CPU. Temeljna ideja je generirati što je moguće bolju aproksimaciju na CPU u malom broju koraka i potencijalno poslati veći broj poligona nego što je potrebno uz pretpostavku da će dobitak na rasterećenju CPU neće biti poništen dodatnim opterećenjem GPU.

Reprezentacija terena koja se ovdje koristi je niz vrhova na rešetci (eng. *grid*) s horizontalnim i vertikalnim brojem vrhova oblika $2^n + 1$, za $n \geq 1$. Primjer s $n = 2$ je prikazan na sljedećoj slici.



Slika 1.4. Mreža bloka terena veličine 5 x 5 vrhova.

Razmak između susjednih vrhova je jednak za sve vrhove i x i z koordinate su određene unaprijed te se ne mijenjaju tijekom izvođenja. y – koordinata, koja predstavlja visinu, se određuje prilikom učitavanja terena iz visinske mape dimenzija jednakih dimenzijama rešetke s vrhovima. Za dovoljno male terene dovoljan je i jedan blok konstruiran na ovaj način. Veće terene nije prikladno prikazivati pomoću samo jednog bloka, jer u tom slučaju teren bi na nekim dijelovima bio predetaljan, a na drugim pregrub. Zbog tog razloga teren se prikazuje pomoću većeg broja takvih blokova koji dijele rubne vrhove. Prednosti ovakve reprezentacije su jednostavnost implementacije i mogućnost učinkovitog generiranja blokova s nižom razinom

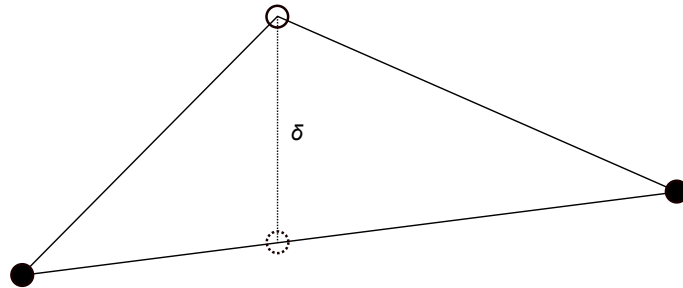
detalja koristeći već postojeću detaljnu mrežu, kao što će biti opisano u nastavku. Jedan od nedostataka je što se, zbog rasporeda vrhova na rešetci, ovakvim modelom ne može opisati terene koji imaju primjerice špilje ili druge strukture za koje je potrebno više od jedne visine za danu točku rešetke. Također, kako susjedni blokovi dijele rubne vrhove, bit će ih potrebno transformirati dvaput, međutim to ne predstavlja velik problem jer se ovako čitav blok može iscrtati jednim zahtjevom za iscrtavanje (eng. *draw call*).

Većina blokova terena neće biti vidljiva u svakom trenutku, stoga bi bilo poželjno slati na GPU samo one blokove koje je stvarno potrebno prikazati. Nepotrebni blokovi se uklanjaju pomoću strukture četverostabla (eng. *quadtree*). Čvorovi stabla su trodimenzionalne obujmice (eng. *bounding box*). Svaki čvor sadrži obujmicu unutar koje se nalaze obujmice njegove djece, a listovi strukture sadržavaju obujmice blokova terena. Prilikom određivanja vidljivih blokova (blok je vidljiv ako se barem djelomično nalazi unutar volumena pogleda) stablo se obilazi od korijena te se rekurzivno provjeravaju presijecanja volumena pogleda s obujmicom u trenutnom čvoru. Ako nema presjeka, nijedan od blokova sigurno nije vidljiv te se čitavo podstablo odbacuje. U suprotnosti, barem jedan blok će biti vidljiv te se provjera nastavlja rekurzivno sve do listova. Na ovaj način velik broj poligona je odbačen prije slanja na GPU. Ostatak poligona će biti odbačen na GPU, čime se broj izračuna ponešto povećava, ali u mnogo manjoj mjeri u odnosu na rasterećenje procesora koji je u ovakvom sustavu usko grlo (eng. *bottleneck*).

Samo odbacivanje nepotrebnih blokova nije dovoljno za postizanje prihvatljivih frekvencija osvježavanja kod velikih terena. Ovaj algoritam rješava taj problem korištenjem *Geo-Mip-Mapa*, postupkom za prikaz blokova terena u odgovarajućoj razini detalja analognim postupku *Mip-Map* kod preslikavanja tekstura. Svaki blok terena će biti predstavljen pomoću više takvih mapa. Mapa razine 0 predstavlja polazni blok u punoj rezoluciji. Svaka sljedeća mapa se dobiva smanjenjem rezolucije prethodne uklanjanjem vrhova svakog drugog stupca i retka rešetke. Kako je polazna mapa dimenzija $2^n + 1$, sljedeća mapa će nakon ove operacije biti dimenzija istog oblika, $2^{n-1} + 1$. Zbog ovakvog odabira dimenzija vrhovi se mogu uklanjati jednoliko sve do najgrublje razine koja ima samo 4 vrha u kutovima polazne mape.

Po uzoru na *Mip-Mape*, i ovaj algoritam odabire manje detaljnu mapu kada smanjenje detalja ne unosi preveliku pogrešku prikaza. Ova pogreška se određuje

slično kao kod *ROAM*-a – koristi se pogreška u prostoru slike. Uklanjanjem vrha unosi se pogreška δ (geometrijska pogreška neovisna o pogledu). Iznos pogreške je udaljenost uklonjenog vrha od njegove projekcije na novonastalu dužinu koja povezuje njegove susjede, prema sljedećoj slici.

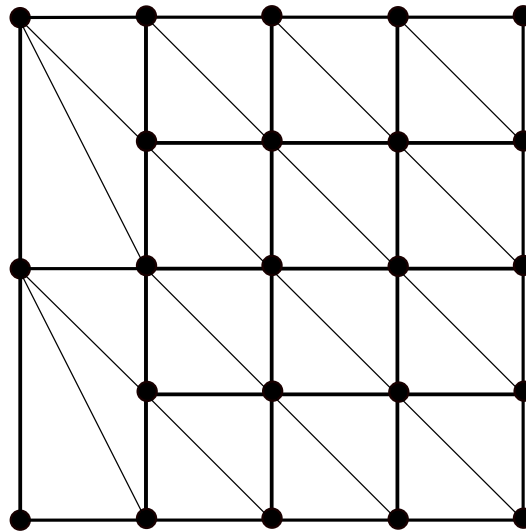


Slika 1.5. Geometrijska pogreška neovisna o pogledu (isprekidana linija).

Uočljivost ove promjene ovisi o relativnom položaju kamere u odnosu na blok i udaljenosti bloka od kamere, d . Na temelju tih informacija izračunava se pogreška u prostoru slike, ϵ . Ako je maksimalna pogreška svih vrhova manje detaljne mape, ϵ_{max} , manja od zadanog praga, τ , prijelaz na tu mapu će biti prihvatljiv.

Izračun ϵ_{max} za svaki blok i svaki kadar može biti procesorski prezahtjevan, a tu vrijednost nije moguće unaprijed izračunati. Kako bi se doskočilo ovom problemu, uvodi se pojednostavljenje u kojem se pretpostavlja da se kamera nalazi uvijek paralelno s blokovima terena, odnosno pretpostavlja se najgori slučaj (pogreška je najuočljivija kada je kamera paralelna s terenom). Uz poznate vrijednosti τ i δ_{max} bloka, sada je moguće unaprijed odrediti udaljenost na kojoj pogreška ϵ_{max} postaje neprihvatljivo velika i predstavlja minimalnu udaljenost na kojoj je dozvoljeno prikazivati taj blok. Prilikom određivanja prikladne razine sada više nije potrebno izračunavati pogrešku u prostoru slike ni za jedan vrh, već je dovoljno usporediti udaljenost bloka od kamere s minimalnom dozvoljenom udaljenošću za svaku od mapa. Za svaki blok se tada odabire najgrublja dozvoljena mapa. Uvođenjem ovakvog pojednostavljenja u određenoj mjeri se broj poligona dodatno povećava (zbog pretpostavke najgoreg slučaja), ali u skladu s osnovnom idejom *GeoMipMappinga*, opterećenje je prebačeno s CPU na GPU.

Preostaje još riješiti problem pukotina koje se javljaju između blokova različitih razina detalja. Jedno moguće rješenje je dodavanje rubnih vrhova manje detaljnoj mapi, no to bi značilo da je potrebno generirati dodatni blok terena i pohraniti ga u memoriju. *GeoMipMapping*, umjesto dodavanja vrhova, uklanja vrhove s rubova detaljnijih blokova. Da bi se to postiglo, dovoljno je koristiti drugačiji skup indeksa vrhova uz polaznu mrežu, bez potrebe za generiranjem nove mreže. Ovaj slučaj je prikazan na sljedećoj slici.



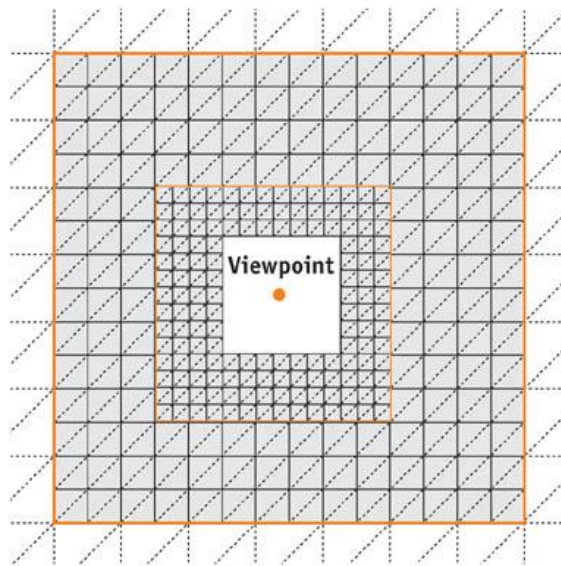
Slika 1.6. Prilagođeni blok koji s lijeva graniči s manje detaljnim blokom (nije na slici).

Zaključno, *GeoMipMapping* je algoritam koji iskorištava mogućnosti grafičkog sklopovlja čime rasterećuje centralni procesor. Mreža se generira samo jednom, u fazi pred-procesiranja, i ne mora se slati GPU prije svakog kadra kao što je to bio slučaj kod *ROAM*-a. Za rješavanje problema pukotina nije potrebno generirati „zakrpe“, već je dovoljno koristiti modificirani skup indeksa za vrhove.

1.3. Struktura Geometry Clipmaps

Geometry Clipmaps je struktura za implementaciju razina detalja koju su izvorno osmislili autori Losasso i Hoppe, a ovdje će biti opisana njegova inačica temeljena na GPU autora Asirvatham i Hoppe. Jedan od glavnih ciljeva algoritma je prebacivanje što je moguće veće količine izračuna na GPU.

Teren je u ovom postupku predstavljen nizom koncentričnih pravokutnih prstena, a u središtu se nalazi pravokutnik. Na sljedećoj slici su vidljiva dva prstena ove strukture.



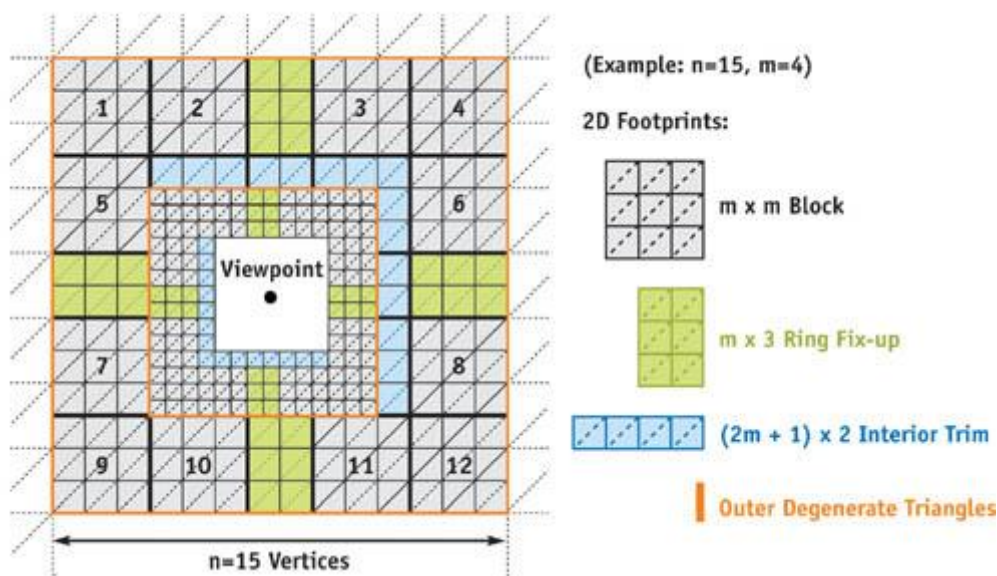
Slika 1.7. Izgled strukture *Geometry Clipmaps* ([4]).

Dijelovi mreže se pomiču tako da kamera uvijek bude u središtu. Svi prsteni se sastoje od jednakog broja vrhova, i zbog toga vrhovi prstena bližih promatraču imaju manje razmake, odnosno prikazuju bliže dijelove terena u višoj razini detalja, a udaljeniji prsteni analogno imaju nižu razinu detalja.

Kao i kod postupka *GeoMipMapping*, informacija o geometriji je razdvojena na dva dijela: x i y – koordinate (koje ovdje predstavljaju širinu i dubinu) su spremljene kao konstantni podaci vrhova, a z koordinata (visina) je spremljena u teksturi. Svaka razina (svaki prsten) ima svoju teksturu dimenzija $n \times n$ koja se ažurira svaki put kada se promatrač pomakne.

Kako bi se omogućilo pomicanje prstena u odnosu na promatrača, dva uvjeta moraju biti zadovoljena. Prvo, dimenzija n mora biti neparan broj, jer se na taj način dobivaju prsteni koji nisu savršeno centrirani, prema prethodnoj slici. S obzirom da prsteni nisu savršeno centrirani, moguće je postići privid pomicanja prstena zamjenom šire i uže stranice. Kako je grafičko sklopovlje optimizirano za rad s teksturama dimenzija koje su potencija broja 2, prikladno je odabrati n oblika $2^k - 1$, čime jedan redak i stupac teksture ostaju neiskorišteni. Drugi uvjet je postojanje

dodatnih dopunskih mreža kojima se omogućuje spomenuto zamjenjivanje šire i uže stranice. Izgled strukture s dopunskim mrežama je prikazan na sljedećoj slici.



Slika 1.8. Struktura *Geometry Clipmaps* sa svim tipovima mreža ([4]).

Ako se promatrač s prethodne slike pomakne prema gore i desno, unutarnji prsten će se također pomaknuti u istom smjeru i doći na mjesto dopunske mreže vanjskog prstena (na slici je označena plavom bojom). Time će nastati praznina s donje i lijeve strane unutarnjeg prstena, a na njeno mjesto će se postaviti nova dopunska mreža.

Na prethodnoj slici je također vidljivo da se prsteni ne sastoje od samo jedne mreže, već više njih. Na taj način je omogućeno uklanjanje mreža izvan volumena pogleda. Naivni pristup prikaza ovakve mreže bi prije svakog kadra generirao svaki prsten na temelju visinske mape i položaja kamere i slao GPU na iscrtavanje. Ovaj pristup bi bio neučinkovit, kao što je pokazano na primjeru ROAM-a. Zbog ponavljajućih uzoraka unutar pojedinih prstena, i međusobno strukturno identičnih prstena, moguće je značajno ubrzati iscrtavanje. Umjesto alociranja po jednog spremnika vrhova za svaku mrežu u strukturi, dovoljno je alocirati po jedan spremnik vrhova za svaki *tip* mreže i koristiti ga za prikaz svih mreža istog tipa. Primjerice, kvadratni blokovi prstena na prethodnoj slici (označeni brojevima 1 – 12) se prikazuju pomoću jednog spremnika s 9 vrhova. Točne koordinate se izračunavaju u procesoru vrhova za svaki blok. S obzirom da je struktura svih prstena identična,

isti spremnici vrhova se koriste za sve prstene, uz slanje prikladnog parametra skale procesoru vrhova.

Ako je n oblika $2^k - 1$, kvadratni blokovi će biti dimenzija $m = (n + 1) / 4$. Svaka stranica prstena se može sastojati od najviše četiri takva kvadratna bloka, zbog čega preostaje šupljina dimenzija $m \times 2$. Svaka stranica prstena ima jednu takvu šupljinu, i zbog toga se alocira jedan spremnik vrhova za dopunske mreže svih stranica svih prstena (na prethodnoj slici te su mreže označene zelenom bojom). Za šupljinu oblika slova „L“ širine jednog pravokutnika s unutarnje strane prstena se alociraju četiri spremnika vrhova i koriste za sve prstene. Četiri spremnika su potrebna jer pomak unutarnjeg prstena može biti u jednom od četiri smjera u odnosu na vanjski prsten (gore – lijevo, gore – desno, dolje – lijevo, dolje – desno) i jer se na taj način izbjegava rotacija koja bi bila potrebna u slučaju samo jednog spremnika. Posljednji tip mreže je niz trokuta s vanjske strane prstena koji popunjavaju pukotine između prstena različitih razina detalja.

Procesor vrhova ima dodatni zadatak uklanjanja efekta iskakanja (eng. *popping*), do kojeg dolazi uslijed naglih promjena u geometriji mreže. Na vanjskim rubovima prstena se definira prijelazno područje, u kojem se vrhovima dodjeljuje interpolirana visina detaljnije i manje detaljne mape. Interpolacijski faktor poprima vrijednost 0 na unutarnjem rubu prijelaznog područja i izvan njega, a na vanjskom rubu vrijednost 1. Između dva ruba faktor linearno raste.

Algoritam *GeometryClipmaps* se razlikuje od prethodnih algoritama primarno po tome što su gotovo sve operacije preseljene na GPU. Spremnici vrhova se generiraju samo jednom i pohranjuju u memoriju GPU-a, što ima minimalne memorijske zahtjeve i eliminira potrebu za slanjem geometrije na GPU prije svakog kadra. S obzirom da se geometrija određuje u procesoru vrhova, jednostavno je implementirati postupak uklanjanja efekta iskakanja. Približno konstantna veličina poligona u prostoru slike je postignuta konstantnom gustoćom vrhova pojedinih razina i smanjenjem gustoće udaljenih razina.

2. Pretraga prostora stanja

Pretraga prostora stanja (eng. *State Space Search*) je metoda iz područja umjetne inteligencije za pronalazak rješenja problema do kojeg se može doći uzastopnim obilaskom konfiguracija, odnosno stanja. Tipičan primjer problema koji se rješava takvim algoritmima je primjerice problem određivanja najkraćeg puta između dvije točke na prometnoj mreži.

Pretraga započinje početnim stanjem, a cilj je pronaći barem jedno stanje koje ima unaprijed zadano traženo svojstvo. U primjeru određivanja najkraćeg puta početno stanje je lokacija početne točke, a jedino ciljno stanje je završna točka. U svakom stanju moguće je poduzeti jednu od niza akcija koje su definirane za to stanje; svaka od akcija će rezultirati prelaskom u drugačije stanje. Konačno rješenje problema je slijed akcija kojim se iz početnog stanja dolazi u jedno od ciljnih stanja. Problem ovog tipa se, prema tome, sastoji od sljedećih komponenata:

1. **Skup stanja S .** Sadrži sva stanja u kojima algoritam pretrage može biti u nekom trenutku.
2. **Početno stanje s_0 .** Prvo stanje u pretrazi.
3. **Skup ciljnih stanja G .** Skup stanja čijim pronalaskom je problem pretrage riješen. Moguće je da problem nema rješenja, i u tom slučaju je G prazan skup.
4. **Funkcija sljedbenika *successor*.** Funkcija koja za svako stanje određuje skup stanja u koja se iz njega može prijeći.
5. **Funkcija cilja *goal*.** Ispitni predikat koji za svako stanje određuje je li ciljno ili nije.

Pomoću ovih komponenata definira se *prostor stanja*, graf čiji čvorovi su stanja, a bridovi prijelazi određeni mogućim akcijama za svako od stanja. Graf je usmjeren jer u općem slučaju nije moguće izvršiti prijelaz u prethodno stanje. Ako se stanjima definiraju cijene prijelaza u nova stanja, radi se o težinskom usmjerenom grafu. Obilaskom grafa nastaje *stablo pretrage* – novi usmjereni graf čiji čvorovi su također stanja, ali u kojem je moguće postojanje većeg broja čvorova s istim stanjem i u kojem nema ciklusa. Stablo pretrage predstavlja sve moguće slijedove akcija za sva stanja, a jedan takav slijed koji povezuje korijen (početno stanje) s nekim od ciljnih stanja predstavlja rješenje problema. Bitno je napomenuti kako stablo pretrage može biti beskonačno premda je skup stanja konačan. To se

dogada u slučaju da prostor stanja ima ciklus jer tada je moguće konstruirati beskonačan slijed koji obilazi čvorove tog ciklusa. Nepraktično je, ili u slučaju beskonačnog stabla, nemoguće u memoriji računala predstavljati stablo pretrage zasebnom strukturom, već je ono definirano implicitno.

Prije definicije općenitog algoritma pretrage, potrebno je spomenuti još nekoliko termina iz područja pretrage:

1. **Proširivanje čvorova (eng. *expanding*).** Proširivanje čvora podrazumijeva generiranje čvorova sa stanjima u koja se može prijeći iz stanja u trenutnom čvoru.
2. **Otvoreni čvorovi.** Čvorovi koji su generirani, ali u trenutnom koraku pretrage još uvijek nisu prošireni.
3. **Zatvoreni čvorovi.** Čvorovi koji su generirani i prošireni.
4. **Strategija pretraživanja.** Redoslijed kojim se otvoreni čvorovi odabiru za proširivanje.

Pretraga se odvija odabirom otvorenog čvora prema strategiji pretraživanja i evaluacijom ispitnog predikata *goal* za stanje u tom čvoru. Ako se radi o ciljnom stanju, algoritam vraća pronađeno ciljno stanje, a u suprotnosti proširuje čvor, čvorove sljedbenike dodaje u skup otvorenih čvorova i nastavlja od prvog koraka. Pseudo-kod općenitog algoritma pretrage je prikazan u nastavku.

```
search(s0) {  
    openList = [node(s0)]  
    while not empty openList {  
        currentNode = openList.remove()  
        if goal(currentNode.state) {  
            return currentNode  
        }  
        for succ in successor(currentNode.state) {  
            insert(openList, succ)  
        }  
    }  
    return NO_SOLUTION  
}
```

Isječak koda 2.1. Općeniti algoritam pretrage.

Metoda dodavanja čvorova u listu otvorenih čvorova izravno definira strategiju pretraživanja. Različiti algoritmi pretrage se razlikuju prvenstveno po strategiji pretraživanja. O njoj izravno ovise sva bitna svojstva algoritama:

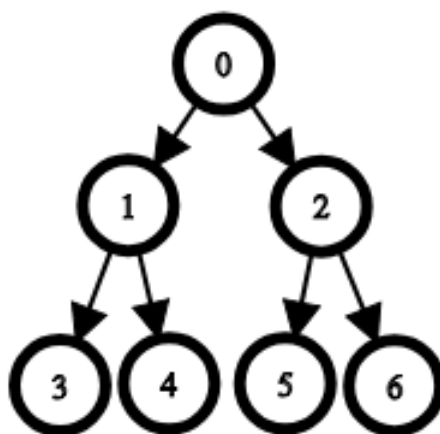
- **potpunost** – pronalazak rješenja kada ono postoji
- **optimalnost** – pronađeno rješenje je rješenje s najmanjom cijenom
- **vremenska složenost** – broj otvorenih čvorova
- **prostorna složenost** – broj pohranjenih čvorova

Algoritmi pretrage se s obzirom na strategiju pretraživanja dijele na neinformirane i informirane algoritme. Neinformirani algoritmi nemaju nikakvu informaciju o lokaciji rješenja, i oni slijepo pretražuju prostor stanja. Informirani algoritmi na temelju poznatih informacija o rješenju koriste različite heuristike kako bi ubrzali pretragu. U nastavku su opisani karakteristični algoritmi obje kategorije.

2.1 Slijepo pretraživanje

2.1.1 Pretraživanje u širinu BFS

Pretraživanje u širinu (eng. *Breadth-first search* – *BFS*) je jednostavna strategija pretrage koja obilazi stablo pretrage razinu po razinu. Potporna struktura ovog algoritma je obična lista, a strategija pretraživanja u širinu je implementirana dodavanjem otvorenih čvorova na kraj liste. Koraci algoritma će biti prikazani na primjeru stabla pretraživanja sa sljedeće slike.



Slika 2.1. Stablo pretrage.

Na početku je otvoren jedini čvor prve razine, čvor 0. Njegovim proširivanjem se na kraj liste redom dodaju čvorovi druge razine, čvorovi 1 i 2. Proširivanjem čvora 1 se dodaju čvorovi 3 i 4, a proširivanjem čvora 2 se dodaju čvorovi 5 i 6, čime je posjećena treća i posljednja razina.

BFS garantirano pronalazi rješenje ako ono postoji jer obilazi sve razine stabla pretraživanja. Premda stablo pretrage može imati beskonačnu dubinu, rješenje, ako postoji, će se nalaziti na konačnoj dubini. Ako je faktor grananja čvorova konačan, sve razine do one u kojoj je rješenje imaju konačan broj elemenata, stoga ih je moguće sve obići. Ako rješenje ne postoji, u slučaju konačnog stabla algoritam staje nakon što su svi čvorovi posjećeni, a u slučaju beskonačnog stabla algoritam nikad neće stati.

BFS je također i optimalan, odnosno pronalazi rješenje u najmanjem broju koraka. S obzirom da se pretražuje razinu po razinu počevši od prve razine, nije moguće pronaći rješenje na većoj dubini prije optimalnog rješenja. U slučaju većeg broja optimalnih rješenja otkriveno će biti samo prvo, ovisno o redoslijedu kojeg generira funkcija sljedbenika *successor*.

Vremenska složenost algoritma je $O(b^{d+1})$, pri čemu je b faktor grananja (eng. *branching factor*), a d redni broj razine s optimalnim rješenjem (numeracija počinje s nulom). Složenost je $O(b^{d+1})$ jer je rješenje u najgorem slučaju na zadnjem mjestu razine d , pa je potrebno otvoriti sve čvorove razine $d + 1$, osim sljedbenika ciljnog čvora. Prostorna složenost je $O(b^{d+1})$ jer se svih $O(b^{d+1})$ otvorenih čvorova pohranjuje u memoriji.

BFS je zbog svoje eksponencijalne vremenske i prostorne složenosti prikladan samo za manje probleme, odnosno one probleme čije rješenje je na maloj dubini u stablu pretrage. *BFS* je neprikladan za probleme s velikim faktorom grananja. Također, algoritam pronalazi rješenje u najmanjem broju koraka, a ne rješenje s najmanjom cijenom, stoga ga se ne može primjenjivati na probleme koji definiraju cijene prijelaza između stanja.

2.1.2 Pretraga s jednolikom cijenom UCS

Pretraga s jednolikom cijenom (eng. *Uniform-cost search – UCS*) rješava nedostatak *BFS*-a uzimanjem cijene prijelaza u obzir. Optimalnim rješenjem se ne smatra nužno ono rješenje do kojeg se dolazi u najmanjem broju koraka, već rješenje do kojeg se dolazi s najmanjom ukupnom cijenom svih prijelaza od

početnog stanja. Potporna struktura algoritma *UCS* je prioritetni red, a čvorovi su sortirani po ukupnoj cijeni.

UCS je potpun i optimalan ako je cijena prijelaza pozitivna jer najprije posjećuje čvorove s manjom cijenom. Ako postoje prijelazi s cijenom manjom ili jednakom nuli, algoritam može zaglaviti u beskonačnoj petlji.

Prostorna i vremenska složenost su također $O(b^d + 1)$. Najveća moguća dubina optimalnog rješenja s cijenom C u problemu s najmanjom cijenom prijelaza ϵ je $\text{floor}(C / \epsilon)$, stoga se složenost određuje pomoću tih cijena.

UCS je, kao i *BFS*, prikladan za manje probleme. Za razliku od *BFS*-a, *UCS* se može koristiti kod problema čiji prijelazi između stanja imaju definirane pozitivne cijene.

2.1.3 Pretraga u dubinu DFS

Kod pretrage u dubinu (eng. *Depth-first search* – *DFS*) najprije se proširuje najdublji otvoreni čvor u stablu pretrage. Strategija pretrage je izvedena dodavanjem novih čvorova na početak liste otvorenih čvorova. Za razliku od algoritama *BFS* i *UCS*, *DFS* nije potpun ni optimalan. Stablo pretrage se ne obilazi razinu po razinu, stoga je moguće da će algoritam pronaći rješenje na većoj dubini od optimalnog rješenja. Ako je stablo pretrage beskonačno, moguće je da algoritam zapne u beskonačnoj petlji. Vremenska složenost algoritma je $O(b^m)$, gdje je m maksimalna dubina stabla. Prostorna složenost je $O(b * m)$. *DFS* je zbog svoje prostorne složenosti prikladan kod problema čije stablo pretrage nije beskonačno i u slučaju da nije potrebno pronaći optimalno rješenje.

Osnovni algoritam se može koristiti i uz beskonačna stabla ako je dubina rješenja unaprijed poznata. Algoritam se modificira na način da se pretraživanje zaustavlja ako dubina prijeđe dubinu rješenja, i nastavlja se s čvorovima na prethodnoj razini. Vremenska složenost tada postaje $O(b^d)$, a prostorna $O(b * d)$.

Ako dubina nije poznata unaprijed, a stablo pretrage je beskonačno, još uvijek je moguće koristiti pretragu u dubinu uz drugačiju modifikaciju osnovnog algoritma. Pretražuje se uz ograničenu dubinu, kao što je opisano, počevši s dubinom 1 i iterativno povećavajući dubinu. To znači da će se čvorovi na plićim razinama posjećivati veći broj puta, međutim ta strategija nije neučinkovita jer se većina čvorova nalazi na dubljim razinama. Vremenska složenost je $O(b^d)$, a prostorna $O(b * d)$. Kako ovaj algoritam obilazi stablo pretrage razinu po razinu,

potpun je i optimalan, i ima manju složenost od algoritma BFS. Ova modifikacija je prikladna za probleme s velikim faktorom grananja kojima dubina rješenja nije poznata unaprijed.

2.2 Heurističko pretraživanje

Kod nekih problema poznate su dodatne informacije o rješenju. Algoritmi slijepe pretrage ne koriste te informacije te se postavlja pitanje je li moguće ubrzati pretragu ugradnjom dodatnih informacija u algoritme. Razvijen je niz heurističkih algoritama koji uvažavaju dodatne informacije na različite načine uz različite složenosti i kriterije optimalnosti.

Prije opisa ove klase algoritama potrebno je definirati novi pojam na kojemu se ovi algoritmi temelje – *heuristiku*. Heuristika je funkcija koja za svako od stanja na temelju poznatih informacija o rješenju daje procjenu o ukupnoj cijeni svih prijelaza između tog stanja i cilja. U ciljnom stanju, prema tome, heuristička funkcija poprima vrijednost 0. Ideja heurističkih algoritama je, umjesto samo stvarne cijene prijelaza do stanja, koristiti vrijednost dobivenu odabranom kombinacijom stvarne cijene i heurističke procjene.

2.2.1 Pohlepni algoritam Greedy best-first search

Pohlepni algoritam procjenjuje udaljenost do ciljnog stanja samo na temelju heuristike, potpuno zanemarujući stvarnu cijenu stanja. U svakom koraku se odabire otvoreni čvor s najmanjim iznosom heurističke funkcije. Ova strategija pretrage je izvedena pomoću prioritetskog reda kao strukture za otvorene čvorove, kao što je to slučaj kod algoritma *UCS*. Razlika u odnosu na *UCS* je kriterij sortiranja čvorova u redu – pohlepni algoritam sortira čvorove po vrijednosti heurističke funkcije, dok ih *UCS* sortira po ukupnoj cijeni prijelaza. Učinkovitost pretrage je u ovom slučaju potpuno ovisna o kvaliteti heuristike, zato što se odabiru akcije za koje je procijenjeno da su najbolje, a ne one koje to stvarno jesu. Ako je za neki problem heuristika prikladno odabrana, pretraga pohlepnim algoritmom može završiti u vrlo malom broju koraka. U slučaju neprikladne heuristike učinkovitost pretrage može postati i niža od učinkovitosti pretrage slijepim algoritmima. Prilikom odabira heurističke funkcije potrebno je voditi računa i o njenoj složenosti s obzirom da je njome potrebno evaluirati svaki otvoreni čvor. Čak i ako algoritam koristi savršenu heuristiku (onu koja za svako stanje daje točnu procjenu cijene prijelaza do optimalnog rješenja), još uvijek je moguće da će učinkovitost pretrage biti niža od

one slijepih algoritama ako joj je složenost previsoka. Primjer takve heuristike je funkcija koja za svako stanje određuje procjenu koristeći primjerice *BFS*. Takva heuristika doista jest savršena (jer je *BFS* optimalan), međutim složenost takve pretrage bi očito bila neisplativo visoka zbog eksponencijalne složenosti *BFS*-a.

Opisani algoritam nije potpun, a zbog toga ni optimalan, jer ne pretražuje stablo pretrage razinu po razinu. Vremenska i prostorna složenost su veće od složenosti *BFS*-a – $O(b^m)$, ali bez obzira na to pretraga će biti učinkovitija ako se za konkretan problem konstruira kvalitetna heuristika.

Ako se odbaci pohranjivanje generiranih čvorova u memoriji, dobiva se inačica algoritma s manjom složenošću – *Hill-climbing search*. U svakom koraku se odabire onaj čvor čiji je iznos heurističke funkcije najmanji, a ostali čvorovi se odbacuju. Prostorna složenost je zbog toga $O(1)$, a vremenska $O(m)$. Modifikacija je upotrebljiva kod problema s malim brojem lokalnih optimuma. Problem lokalnih optimuma se može ublažiti višestrukim pokretanjem algoritma uz različita početna stanja, ali taj pristup je primjenjiv samo na optimizacijske probleme kod kojih početno stanje nije zadano.

2.2.2 Algoritam A*

Algoritam *A** u pretrazi koristi sve dostupne informacije – stvarnu cijenu čvora i heurističku procjenu. Algoritam ima nekoliko varijanti, a u nastavku će biti opisana varijanta korištena u implementacijskom dijelu ovog rada. Pseudo-kod algoritma je prikazan u nastavku.


```

search(s0) {
    open = [node(s0)] // priority queue; nodes are sorted by
                        // the sum of total cost and heuristic
                        // cost

    closed = []

    while open is not empty {
        current = open.removeFirst()
        if goal(currentNode.state) {
            return currentNode
        }

        for succ in successor(currentNode.state) {
            if succ' in open and succ'.cost < succ.cost {
                continue
            }
            if succ' in closed and succ'.cost < succ.cost {
                continue
            }

            open.remove(succ')
            open.insert(succ)
        }
    }

    return NO_SOLUTION
}

```

`succ'` is an existing node that holds the same state as the `succ` node

Isječak koda 2.1. Pseudo-kod algoritma A*.

Algoritam A* u svakom koraku odabire onaj otvoreni čvor čija je suma ukupne cijene i heurističke procjene najmanja. Na taj način se među stanjima s jednakom

cijenom preferiraju ona za koja je procijenjeno da su bliže ciljnom stanju. Algoritam *UCS*, koji razmatra samo cijenu čvorova, u općem slučaju može odabrati čvor udaljeniji od cilja prije bližeg čvora, odnosno A^* uklanjanjem heuristike degenerira u *UCS*.

U prethodnim pod-poglavljima opisani su slijepi i optimalni algoritmi te jedan heuristički, ali neoptimalan. Algoritam A^* je nastao kao odgovor na potrebu za kombinacijom ovih svojstava – konstruirati algoritam koji je optimalan i učinkovito pretražuje prostor stanja koristeći heuristike. Na primjeru pohlepnog algoritma je pokazano da mogućnost pronalaska optimalnog rješenja ovisi o izboru heuristike. To je slučaj i kod algoritma A^* – samo određena klasa heuristika će dati optimalno rješenje. Da bi pretraga rezultirala optimalnim rješenjem, korištena heuristika mora biti *dopustiva* (eng. *admissible*). Dopustiva heuristika je heuristika koja ne precjenjuje cijenu do ciljnog stanja za svako stanje. Trivijalan primjer dopustive heuristike je heuristika koja za svako stanje vraća vrijednost 0 – kao što je pokazano, A^* tada postaje ekvivalentan algoritmu *UCS* (koji je optimalan). Ako heuristika za neko stanje na putu do optimalnog rješenja precijeni udaljenost, moguće je da će pretraga zaobići taj slijed, odnosno algoritam više nije optimalan.

Algoritam A^* implementira dodatnu optimizaciju koristeći listu posjećenih stanja. Posjećena stanja se pamte te ako se naiđe na već posjećeno stanje, uspoređuju se cijene postojećeg i novog čvora s tim stanjem. Ako je cijena ranije otvorenog čvora manja, nema potrebe ponovno proširivati čvor s tim stanjem jer cijena slijeda do rješenja preko tog čvora može biti samo veća od cijene slijeda s postojećim čvorom. U nekim slučajevima je moguće izbjeći pamćenje posjećenih stanja ako je za problem moguće definirati heuristiku koja ima svojstvo *konzistentnosti*. Konzistentna heuristika je ona heuristika koja producira monotono rastući niz zbrojeva ukupne cijene i heurističke procjene u svim sljedovima, odnosno ona za koju vrijedi sljedeća relacija:

$$f(n_2) \geq f(n_1),$$

pri čemu je f zbroj ukupne i heurističke cijene, a n_2 su svi čvorovi sljedbenici čvora n_1 . Uz konzistentnu heuristiku najjeftiniji čvor za neko stanje će biti generiran prvi pa sve ostale čvorove s tim stanjem nije potrebno ni razmatrati. Vremenska i prostorna složenost ovise o izboru heuristike. Uz konzistentnu heuristiku, složenosti su $O(\min(b^d, b * |S|))$, pri čemu je $|S|$ veličina skupa stanja. Komponenta $b * |S|$

potječe od pamćenja posjećenih stanja – u najgorem slučaju svako stanje će biti posjećeno jednom, a pri tome će biti generirano b čvorova za svako stanje.

A^* je algoritam izbora pri rješavanju problema kod kojih postoje dostupne informacije o rješenju jer pomoću tih informacija odbacuje velike dijelove stabla pretrage.

3. Razvijeni sustav

U sklopu rada razvijen je sustav čije su temeljne dvije komponente generator krajolika i komponenta za optimizaciju trajektorije ceste. Temeljne komponente se sastoje od niza drugih komponenata koje su nadomjestive. Odlučeno je da će sustav biti podijeljen na radni okvir i klijentski kod pri čemu klijentski kod odabire i konfigurira komponente radnog okvira prema potrebi.

U implementaciji je korišten jezik *Java* zbog svoje robusnosti te grafičko programsko sučelje *OpenGL*. Biblioteka *LWJGL* (*Lightweight Java Game Library*) je upotrijebljena kao tanak omotač prema *OpenGL*-ovom API-ju. Sjenčari (eng. *shaders*) su pisani u jeziku *GLSL*.

3.1 Generiranje terena

3.1.1 Poligonalna mreža

Prilikom dizajniranja algoritma određeni su nužni zahtjevi i poželjna svojstva po uzoru na algoritme opisane u drugom poglavlju. Temeljne niti vodilje su ovdje bile malo opterećenje centralnog procesora, obavljanje što je moguće veće količine posla u fazi pred-procesiranja, postojanje razina detalja i jednostavnost implementacije. Dobiveni teren mora biti dovoljno velik da prikaže područje na kojem će biti generirana tipična dionica ceste (nekoliko desetaka kilometara). Nisu potrebni tereni veličine kontinenata (iako će ih biti moguće postići uz smanjenu kvalitetu), stoga je odlučeno što je moguće više izračuna prebaciti u fazu pred-procesiranja na račun povećanog zauzeća memorije. Ideja je unaprijed pohraniti blokove terena u memoriju grafičke procesorske jedinice i tijekom izvođenja samo slati zahtjeve za prikazom odabranog podskupa blokova. Posljedica ovoga će biti nisko opterećenje i CPU i GPU, što je primarni cilj, na račun velikog zauzeća video memorije koja će postati ograničavajući faktor, ali neće predstavljati problem za ovu primjenu.

Struktura poligonalne mreže je ista kao kod algoritma *GeoMipMapping* – vrhovi su raspoređeni na ekvidistantnoj rešetci koja se proteže x i z – koordinatama, kao na slici 1.4. Jedna mreža predstavlja određeni dio terena – *blok* terena – a cjelokupni teren se sastoji od većeg broja blokova. Dimenzije rešetke i gustoća vrhova su podesivi parametri, i o njihovoj kombinaciji će ovisiti kvaliteta prikaza. Radi jednostavnosti svi blokovi imaju jednake dimenzije. Definiranjem različitih gustoća vrhova za isto područje terena dobivaju se blokovi u različitim razinama

detalja. Broj razina i njihovu detaljnost određuje korisnik. Ako su zatražene dvije razine detalja, svaki blok će se generirati dvaput – jednom u nižoj, i jednom u višoj razini detalja.

Svaki blok se sprema u zaseban spremnik spremnika vrhova (eng. *Vertex Array Object* - VAO). Vrhovi bloka su smješteni u spremnik vrhova (eng. *Vertex Buffer Object* - VBO). Vrhovi su spremljeni po recima te se koriste indeksi za iscrtavanje čime se izbjegava potreba za ponavljanjem vrhova. Prilikom generiranja vrhova generiraju se i normale u vrhovima i smještaju u novi VBO istog VAO. Normale se dobivaju uzorkovanjem visinske funkcije i aproksimacijom metodom konačnih diferencija. Derivacije po x i z varijablama su dobivene pomoću centralnih diferencija:

$$\begin{aligned}\nabla[x]f(x, z) &= \lim_{h \rightarrow 0} \frac{f(x + h, z) - f(x - h, z)}{2h} \\ \nabla[z]f(x, z) &= \lim_{h \rightarrow 0} \frac{f(x, z + h) - f(x, z - h)}{2h}\end{aligned}$$

Normalni vektor u točki (x, z) je tada:

$$\left[\lim_{h \rightarrow 0} \frac{f(x + h, z) - f(x - h, z)}{2h}, \quad -1, \quad \lim_{h \rightarrow 0} \frac{f(x, z + h) - f(x, z - h)}{2h} \right]$$

Uvrštavanjem konkretnog h i nakon skaliranja s $-2h$ dobiva se:

$$[f(x - h, z) - f(x + h, z), \quad 2h, \quad f(x, z - h) - f(x, z + h)]$$

Ovaj oblik je efikasniji za izračun jer se izbjegavaju dva dijeljenja po normali. Za izračun svake normale potrebno je uzorkovati visinsku funkciju četiri puta.

U sljedećem koraku se generiraju teksturne u i v - koordinate. Koordinate su dobivene jednostavnim skaliranjem na temelju dimenzija prostora kojeg jedna tekstura predstavlja. U posljednji spremnik se zapisuju težinski utjecaji korištenih tekstura. Izračun tih utjecaja je temeljen na biomsjoj mapi koja će biti objašnjena nešto kasnije. Ovdje ćemo samo spomenuti da suma svih težina mora biti 1 jer se u suprotnom javljaju artefakti pretamnih ili presvijetlih tekstura.

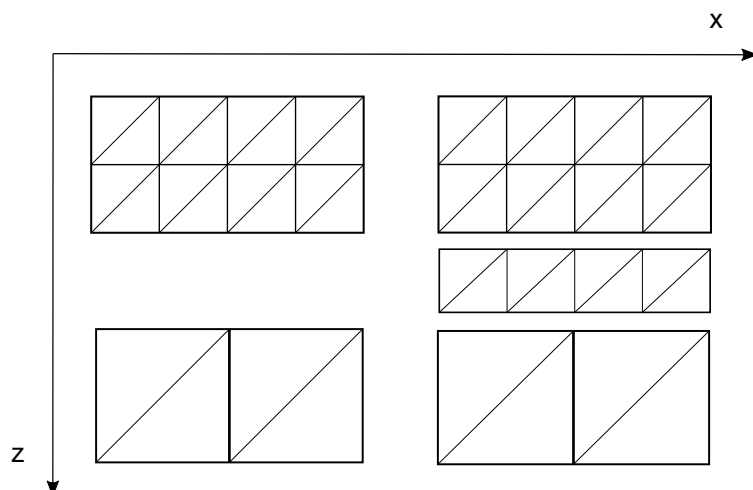
U fazi predprocesiranja se generiraju svi blokovi u svim razinama detalja. Kako je konstrukcija pojedinog bloka neovisna o svim ostalim blokovima, ovaj proces je paraleliziran. Blokovi se generiraju u nizu pozadinskih dretvi (eng. *daemon threads*) i dodaju u zajedničku višedretveno sigurnu strukturu. Zadatak dodjele poslova pozadinskim dretvama je povjeren zasebnoj komponenti koja ima pristup

generiranim blokovima i koja za vrijeme izvođenja određuje blokove koji će se prikazati. Blokovi se generiraju počevši od najniže razine detalja prema višim razinama jer generiranje manje detaljnih blokova traje kraće pa ih je moguće ranije i pokazati. Zbog tog razloga faza pred-procesiranja u ovom pristupu nije u potpunosti ekvivalentna fazama pred-procesiranja ranije opisanih algoritama (sve je pripremljeno prije bilo kakvog prikaza). Ovdje je omogućen prikaz *trenutačno generiranih elemenata* dok generiranje još uvijek traje.

Odabir razina detalja je temeljen na ideji približno konstantne veličine poligona u prostoru slike, po uzoru na algoritam *Geometry Clipmaps*. Razina detalja ovisi o udaljenosti promatrača od bloka, a pragovi udaljenosti su korisnički definirani. Prije prikaza svakog kadra potrebno je odrediti skup prikladnih blokova i poslati tu informaciju GPU. Potrebno je napomenuti kako se, nakon što su blokovi generirani, više ni u jednom trenutku ne šalju ili modificiraju spremnici vrhova na GPU, već se šalje samo niz *OpenGL*-ovih identifikatora blokova. Ovaj proces je vrlo procesorski nezahtjevan. Dodatno ubrzanje je postignuto uvođenjem tolerancije na manje pomake i keširanjem prethodno prikazanih blokova. Ako se promatrač u vremenu proteklom između dva prikaza pomaknuo za udaljenost manju od zadanog praga, ponovno se prikazuje isti skup blokova kao u prethodnom kadru. U primjenama gdje su najčešći mali i kontinuirani pomaci promatrača procesorski dio posla se tako većinom svodi na usporedbu dviju udaljenosti.

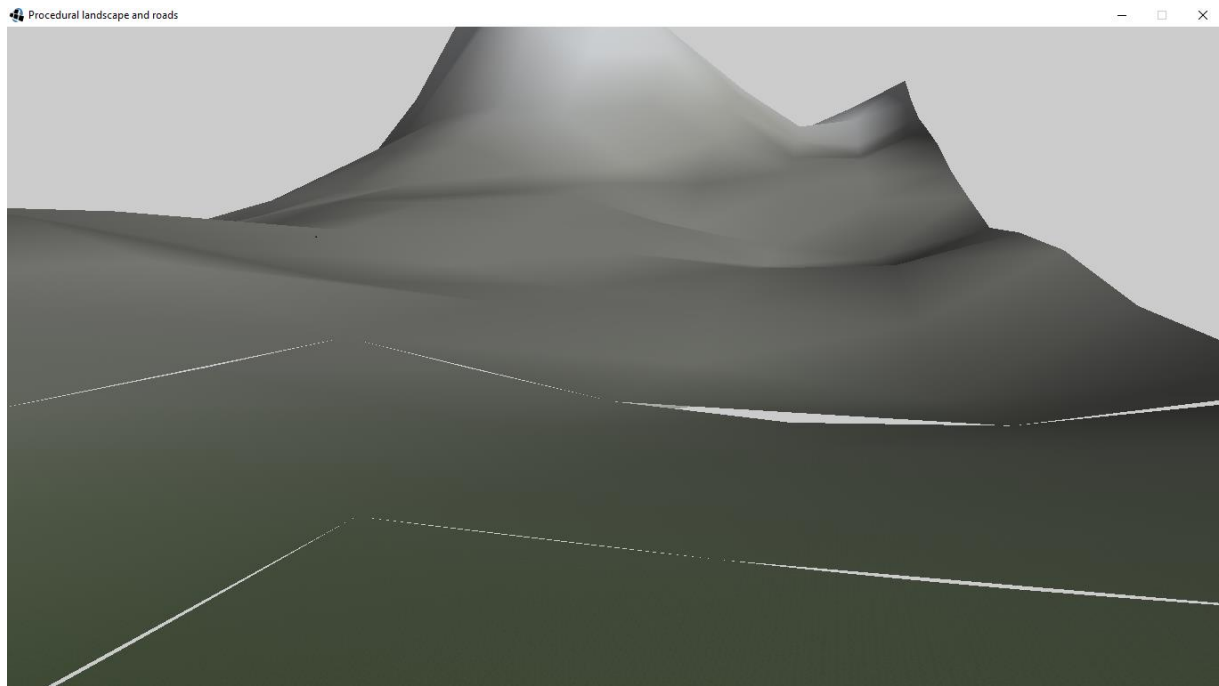
Preostaje još riješiti problem pukotina između blokova s različitim razinama detalja. Pukotine su otklonjene generiranjem zakrpa između blokova. Generiranje zakrpa počinje obilaskom blokova od gornjeg lijevog po recima do donjeg desnog. Svakom bloku se generiraju zakrpe koje će u konačnici biti smještene dolje i desno u odnosu na svaki blok (nije potrebno za svaki blok generirati zakrpe sa sve četiri strane jer bi se tada sve zakrpe generirale dvaput). Za svaka dva susjedna bloka i L razina detalja potrebno je generirati $L * (L - 1)$ zakrpa (jer se zakrpe ne generiraju ako su blokovi iste razine detalja).

Na sljedećoj slici je prikazan pogled na dva bloka s različitim razinama detalja u smjeru negativne y - osi (odozgo). Na slici je također dodan i nepostojeći razmak između blokova kako bi bilo lakše objasniti generiranje zakrpa (u stvarnosti su rubovi blokova preklapljeni).

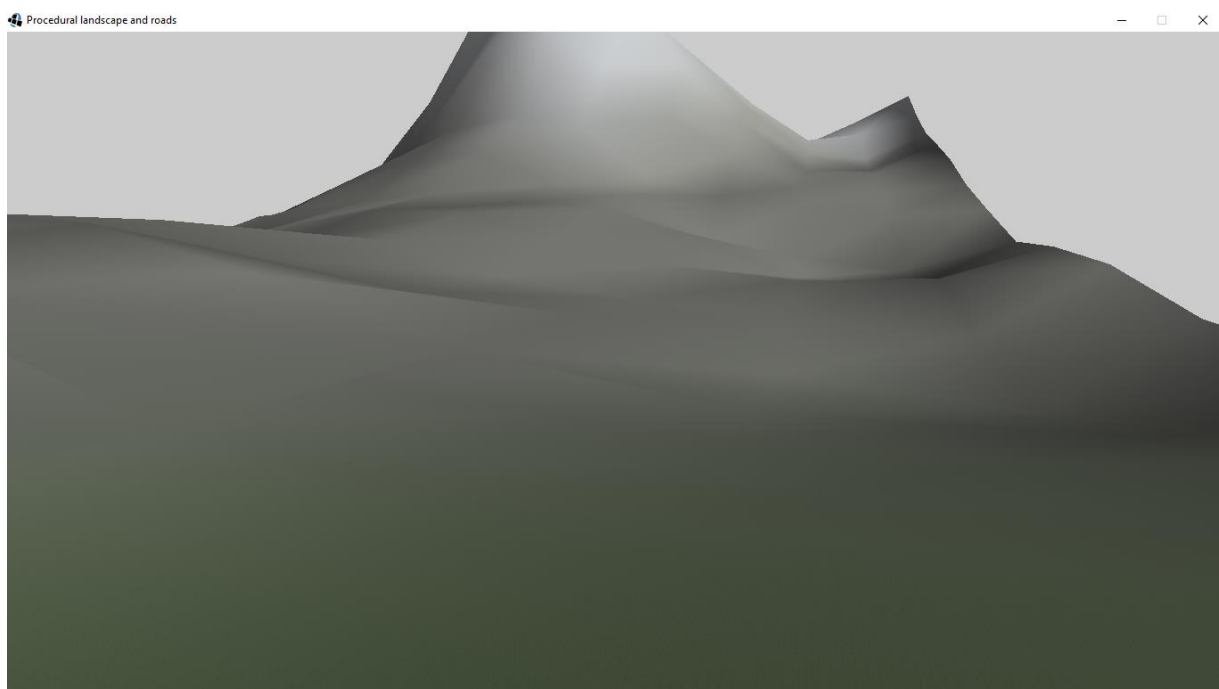


Slika 3.1. Blokovi s različitim razinama detalja prije (lijevo) i nakon generiranja zakrpe (desno).

Zakrpa će se sastojati od dva niza vrhova. Broj vrhova u svakom nizu će biti jednak broju vrhova na rubu detaljnijeg bloka. Niz vrhova stranice koja graniči s detaljnijim blokom se dobije kopiranjem ruba detaljnijeg bloka. Drugi niz se u početku popuni vrhovima s ruba manje detaljnog bloka, a zatim se iterativno dodaju novi vrhovi između svaka dva susjedna vrha. Ovaj postupak se ponavlja sve dok novi niz ima manje vrhova od prvog niza. Visina (y – koordinata) vrhova koji se umeću između postojećih vrhova se ne uzorkuje iz visinske mape, već se određuje kao aritmetička sredina visina dva vrha između kojih se vrh umeće. Na taj način se novi vrhovi smještaju na katete trokuta na rubu bloka čime se u potpunosti uklanjaju pukotine. Usporedba prikaza sa zakrpama i bez njih je prikazana na sljedeće dvije slike.



Slika 3.2. Izgled terena bez generiranih zakrpa (razlika u detaljnosti blokova različitih razina je povećana kako bi artefakt bio uočljiviji).



Slika 3.2. Teren uz generiranje zakrpa.

3.1.2 Visinska funkcija

Komponenta za generiranje terena očekuje visinsku funkciju kao ulaz. Visinska funkcija implementira preslikavanje uređenih parova koordinata (x, z) u visinske vrijednosti terena. Očekuje se da funkcija bude definirana nad

kontinuiranom domenom, što je zadovoljeno za *Simplex* šum korišten u sustavu. Podržani su i tereni predstavljeni visinskim mapama, a kontinuitet domene je postignut bilinearnom interpolacijom vrijednosti u mapi.

U slučaju *Simplex* šuma, za svaku točku (x, z) funkcija šuma generira vrijednost iz intervala $[-1, 1]$, pri čemu vrijednosti bliskih koordinata imaju svojstvo sličnosti. Prije uporabe dobivene vrijednosti, ona se skalira na interval $[0, 1]$.

Jednostruko uzorkovanje šuma rezultira terenom koji djelomično nalikuje na teren kakav bi mogao postojati u stvarnosti, ali također se javlja i problem prevelike glatkoće. Stvarni tereni imaju nekoliko razina neravnina. Primjerice, prva razina može predstavljati generalni izgled terena – raspored planina i dolina. Visina na prvoj razini se mijenja s niskom frekvencijom i visokim amplitudama. Na sljedećoj razini mogu se nalaziti brežuljci i udubine na terenu, kao što su ponikve i slično. Visine na ovoj razini se mijenjaju s manjom amplitudom i većom frekvencijom. Svaka sljedeća razina predstavlja sve veću razinu detalja s nižim amplitudama i većim frekvencijama.

Visina terena je izvedena koristeći ovu pretpostavku o postojanju opisanih razina. Za potrebe izračuna frekvencija definira se bazna frekvencija – frekvencija prve razine, faktor povećanja frekvencije za sljedeće razine i broj razina neravnina – *oktava*. Za izračun visine koriste se početna amplituda te faktor smanjenja amplitude za svaku sljedeću razinu – *hrapavost*. Pseudo-kod ovog izračuna je prikazan u nastavku.

```
totalNoise = 0
aplitudesSum = 0
for i = 0 to octaves
    f = baseModifier * frequencyIncreaseFactori
    amplitude = roughnessi //roughness is lesser than 1
    aplitudesSum += amplitude
    totalNoise += simplex(x * f, z * f) * amplitude
totalNoise /= aplitudesSum
```

Isječak koda 3.1. Pseudo-kod izračuna visine.

Nakon izvođenja ovog isječka koda dobiva se vrijednost također iz intervala [0, 1], ali s većim brojem razina neravnina. Prije skaliranja na stvarni raspon visina, nad dobivenom vrijednošću se još provodi transformacija koja će znatno povećati visoke vrijednosti, i umjereno smanjiti niske vrijednosti. Na taj način se povećava varijacija visine u visokim predjelima, i smanjuje u niskim predjelima. Transformacija je sljedeća:

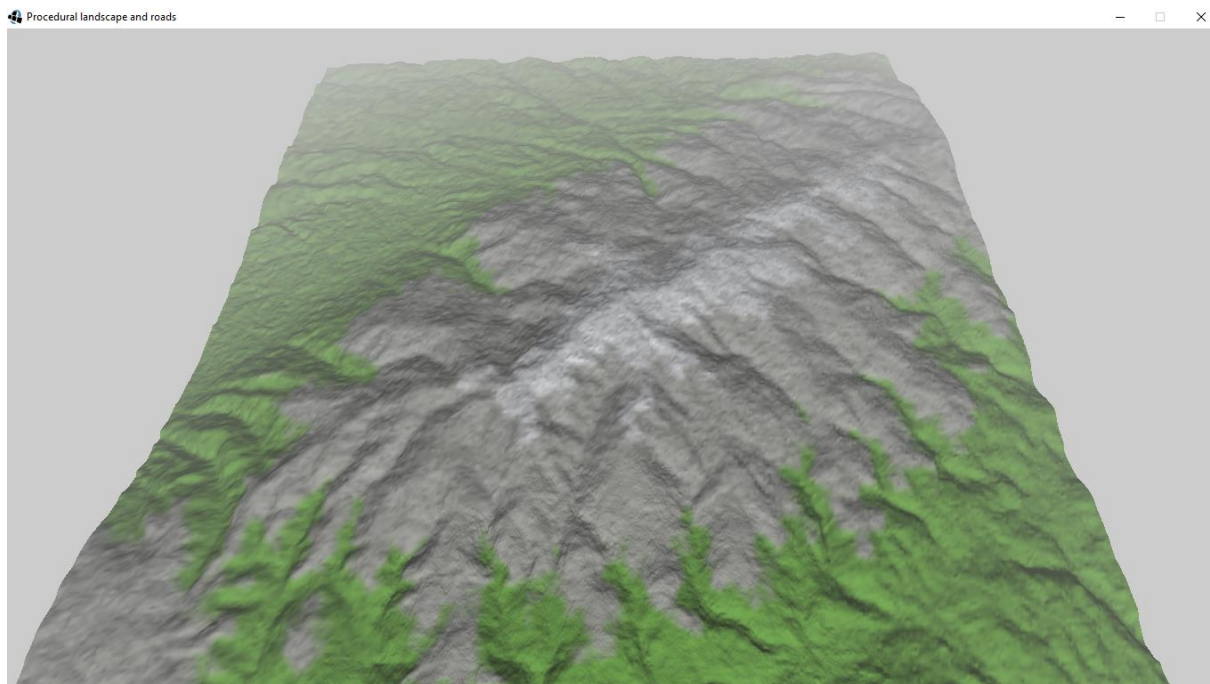
```
totalNoise = (totalNoise + threshold)variation
```

`threshold` definira kojem dijelu raspona [0, 1] će se iznos povećati, a kojem smanjiti, a `variation` definira iznos te promjene.

3.1.3 Tekstura terena

Teren je teksturiran u skladu s biomskom mapom – komponentom sustava koja je zadužena za određivanje klimatskih prilika na svim koordinatama terena. Biomska mapa određuje klimatske prilike koristeći informaciju o visini te o *vlazi* – vrijednosti koja u određenoj mjeri utječe na raspored tekstura terena i na raspored vegetacije, što će biti objašnjeno kasnije. Vlaga je proizvoljno definirana vrijednost koja se temelji na šumu, i koja sprječava pojavu strogih granica u vrsti teksture na zadanim visinama. Osim varijacije teksture pomoću vrijednosti vlage, teksture prijelaznih područja se interpoliraju trapezoidnim težinskim funkcijama čime se eliminiraju grubi prijelazi. Nakon interpolacije suma svih težinskih funkcija iznosi 1 i tako se izbjegavaju spomenuti artefakti pretamnih ili presvijetlih tekstura.

Na sljedećoj slici je prikazan teren generiran postupcima opisanim u ovom poglavlju. Teren na slici predstavlja područje Medvednice dimenzija 12 x 12 kilometara dobiven iz visinske mape rezolucije 1081 x 1081 piksel.



Slika 3.3. Područje Medvednice dimenzija 12 x 12 kilometara.

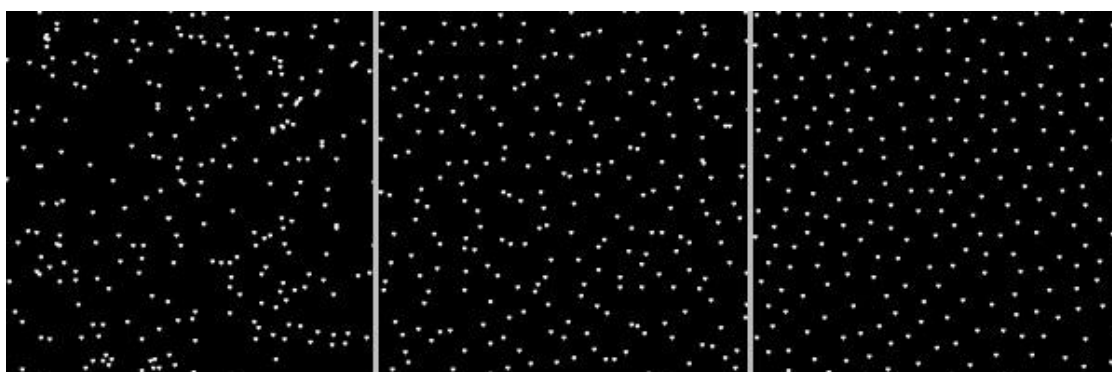
3.2 Postavljanje vegetacije

Biomska mapa korištena u sustavu podržava dva glavna bioma – područja umjerene klime s travom i listopadnim drvećem te snježna područja sa zimzelenim drvećem. Prijelazna područja sadrže mješavinu ove dvije vrste drveća. Svako stablo je predstavljeno s dva modela – jedan model koji predstavlja stablo u visokoj razini detalja i jedan koji je spoj dvaju pravokutnika kojima je pridružena tekstura sa slikom odgovarajućeg stabla. Sustav podržava rad i s proizvoljno velikim brojem razina detalja, ali u slučaju većeg broja razina potrebno je izraditi i veći broj modela objekata.



Slika 3.4. Scena s objektima s različitim razinama detalja.

Stabla su raspoređena pomoću algoritma uzorkovanja Poissonovih diskova. Ovaj algoritam je prikladan za određivanje lokacija stabala jer producira točke koje nikad nisu međusobno udaljene manje od zadane minimalne udaljenosti ili više od maksimalne udaljenosti te su raspoređene jednoliko. Algoritmi raspoređivanja temeljeni na potpunoj slučajnosti ili slučajnom rasporedu unutar rešetke nemaju ova svojstva. Usporedba rezultata ovih algoritama je vidljiva na sljedećoj slici.



Slika 3.5. Usporedba rezultata triju algoritama: potpuna slučajnost (lijevo), slučajni raspored unutar rešetke (sredina) i uzorkovanje Poissonovih diskova (desno).

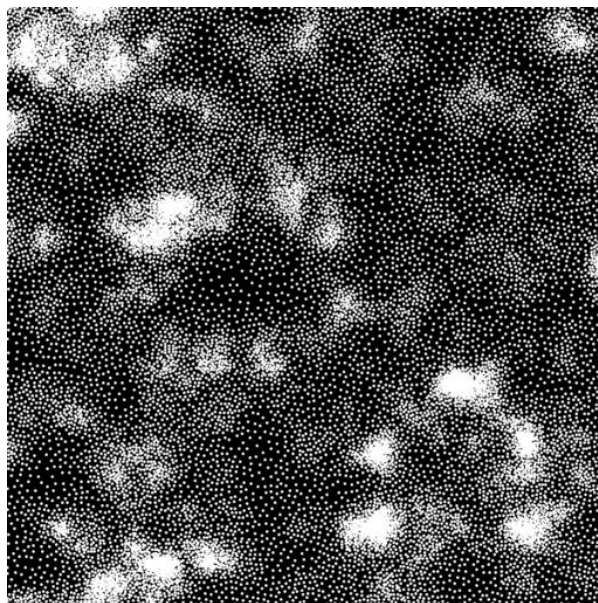
Algoritam započinje slučajnim odabirom lokacije početne točke koju dodaje u listu točaka za obradu i listu rezultatnih točaka. Za svaku točku iz liste za obradu

nastoji se generirati do k točaka na kružnici sa središtem u trenutnoj točki. Ako nova točka nije preblizu nekoj od postojećih, dodaje se u listu rezultata i listu za obradu čime je omogućeno daljnje generiranje. Za provjeru bliskosti točaka koristi se rešetka (eng. *grid*). Algoritam staje kad više nije moguće generirati nove točke ili je dosegnut zadani maksimalan broj točaka. Pseudo-kod algoritma je prikazan u nastavku.

1. Stvori rešetku takvu da se u njoj može naći najviše jedna točka. Ako je minimalna tražena udaljenost među točkama r , tada je duljina stranice svake ćelije rešetke jednaka $r / 2^{0.5}$.
2. Slučajno odaberi početnu točku i dodaj ju u listu rezultata, listu za obradu i u rešetku.
3. Dok lista za obradu nije prazna:
 - 3.1 Slučajno odaberi točku T_o iz liste za obradu.
 - 3.2 Generiraj k točaka na kružnici sa središtem u odabranoj točki i radijusom iz intervala $[r, 2r]$.
 - 3.3 Za svaku točku provjeri je li preblizu nekoj od postojećih pomoću rešetke – ako je, odbaci je, inače dodaj točku u listu za obradu, listu rezultata i rešetku.
 - 3.4 Ukloni točku T_o iz liste za obradu.
4. Vрати listu rezultata.

Isječak koda 3.2. Pseudo-kod algoritma uzorkovanja Poissonovih diskova.

Rezultat ovakvog uzorkovanja izgleda kao na desnom dijelu slike 3.5. Ovakav raspored je prilično uvjerljiv, ali moguće je postići i realističniji rezultat modifikacijom algoritma na način da iznos udaljenosti r nije fiksna, već se mijenja u ovisnosti o lokaciji. Time se postiže raspored koji se sastoji od većeg broja grupacija točaka, i izgleda kao na sljedećoj slici.



Slika 3.6. Rezultat prilagođenog uzorkovanja.

U originalni algoritam su uvedene sljedeće izmjene:

1. Algoritam prima funkciju f koja za predane koordinate (x, z) daje iznos r .
2. Veličina ćelije je $r_{max} / 2^{0.5}$, pri čemu je r_{max} maksimalni iznos kojeg f može proizvesti.
3. Svaka ćelija više ne čuva jednu točku, već listu točaka. To je potrebno zato što stvarna vrijednost r sada može biti i manja od r_{max} .
4. Funkcija za određivanje bliskosti iterira po svim točkama liste svake od obližnjih ćelija.

Funkcija za određivanje iznosa r definirana je pomoću biomske mape, koja nudi funkcionalnost određivanja gustoće vegetacije za svaku koordinatu terena. Gustoća vegetacije ovisi o strmini lokacije i o vlazi (koja je definirana kao funkcija šuma s kodomenom $[0, 1]$). Preciznije, gustoća vegetacije je određena kao umnožak iznosa vlage i kosinusa kuta kojeg normala terena zatvara s xz ravninom. Kako bi se dodatno istaknula pojava grupacija, određeno je da funkcija f vraća kvadrat prethodne vrijednosti.

Nakon definiranja lokacija stabala, određuje se koja vrsta stabla će se naći na kojoj lokaciji. Zadatak ove odluke također je dodijeljen mapi bioma, koja određuje preferiranu vrstu stabla za predane (x, z) koordinate na temelju visine, vlage i proizvoljno definiranog šuma pomoću kojeg se eliminira problem strogih granica između vrsta stabala, slično kao kod teksturiranja terena.

Zbog prirode ovog algoritma nije ga moguće paralelizirati u potpunosti (lokacije budućih točaka nisu neovisne o lokacijama postojećih točaka). Ubrzanje se može dobiti primjerice podjelom područja uzorkovanja na blokove, i na svakom bloku odrediti točke zasebnim izvođenjem algoritma uzorkovanja. Nedostatak ovog pristupa mogućnost da se točke na rubovima blokova generiraju međusobno preblizu. Uz pomoć zajedničke višedretveno sigurne strukture za pohranu rezultata ovaj problem se može ublažiti, ali tada se javljaju problemi s nedeterminističnošću rezultata jer konačan raspored ovisi o rasporedu izvođenja dretvi zaduženih za pojedine blokove, što ovisno o primjeni može i ne mora predstavljati problem.

Zbog opisanih problema, odlučeno je da će određivanje rasporeda i vrsta stabala biti izvedeno tehnikom *protočne strukture* (eng. *pipelining*). Umjesto generiranja cjelokupnog skupa točaka, zatim određivanja prikladne vrste stabla za svaku točku i konačno prikaza svih stabala, ideja protočne strukture je podijeliti sve ove faze na manje dijelove koji se mogu izvoditi paralelno. Za ostvarenje ubrzanja koriste se tri dretve sa sljedećim zadacima:

1. Dretva koja generira točke uzorkovanjem Poissonovih diskova. Nakon što se generira određeni broj točaka, točke se dodaju u blokirajući red (red u kojeg se elementi mogu dodavati i uzimati sigurno iz različitih dretvi; dretva koja pokušava uzeti element iz praznog reda ili dodati element u pun red je blokirana dok se stanje u redu ne promijeni).
2. Dretva koja uzima točke iz reda i pridružuje im odgovarajuću vrstu stabla na temelju biomske mape. Točke s pridruženim vrstama stabla se dodaju u novi red.
3. Glavna dretva zadužena za prikaz. Ovisno o lokaciji kamere, uzima odgovarajući skup točaka s pridruženim vrstama stabala iz reda prethodne dretve i prikazuje ih. Ako u nekom trenutku za trenutačnu lokaciju promatrača još nisu generirane sve točke, prikazuju se samo do tada generirane točke za tu lokaciju.

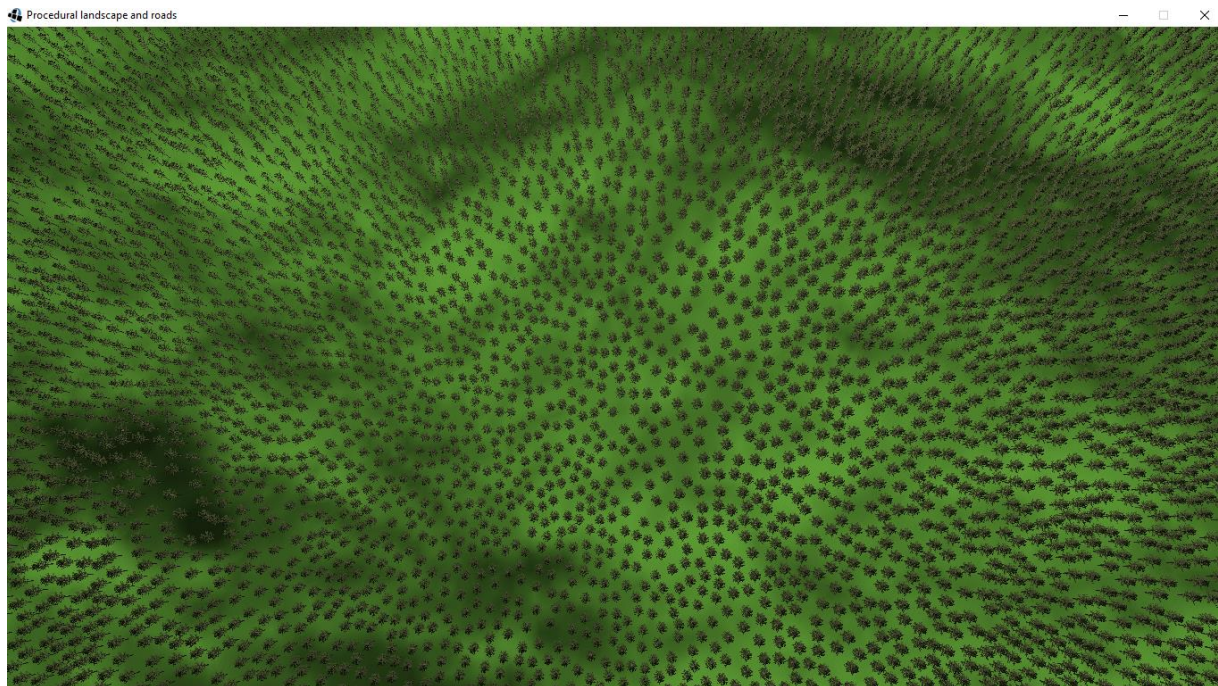
Ubrzanje dobiveno na ovaj način je značajno, a algoritam je i dalje determinističan. Ako trajanje generiranja točaka označimo s T_1 , a trajanje pridruživanja vrsta stabala točkama s T_2 , ukupno vrijeme koje je potrebno čekati prije nego što se stabla mogu prikazati iznosi $T_1 + T_2$. Ako se koristi protočna struktura, ovi poslovi su podijeljeni u faze. U jednoj fazi generiranja dretva zadužena za uzorkovanje generira n točaka, a u jednoj fazi pridruživanja dretva zadužena za

pridruživanje vrsta točkama to učini također za n točaka. Ovakvih faza ima $\text{ceiling}(N / n)$, pri čemu se u zadnjoj fazi obrađuje manje od n točaka ako N / n nije cijeli broj. Na početku obrade radi samo dretva za uzorkovanje (protočna struktura se puni), a na kraju samo dretva za pridruživanje (protočna struktura se prazni). Ako trajanje jedne faze uzorkovanja označimo s t_1 , a trajanje jedne faze pridruživanja s t_2 , ukupno trajanje će iznositi:

$$t_1 + (\text{ceiling}(N / n) - 2) * \max(t_1, t_2) + t_2$$

Sa smanjenjem n t_1 i t_2 se smanjuju pa iznos ukupnog trajanja teži ka $\max(T_1, T_2)$. Naizgled je logično odabrati minimalni mogući n (1), međutim potrebno je uzeti u obzir i vrijeme potrebno za komunikaciju između dretvi i njihovo blokiranje, koje će biti to veće što je više faza obrade. Zbog tog razloga je prikladno odabrati ipak nešto veći n (pokazalo se da je za ovu primjenu prikladan n 1000). Osim što je ukupno trajanje kraće, moguće je korištenje i djelomično generiranih rezultata. Lančano gašenje pozadinskih dretvi po završetku obrade je ostvareno tehnikom *Thread Poisoning*.

U glavnoj programskoj petlji potrebno je odrediti prikladnu razinu detalja svakog objekta u svakom kadru. Ovaj zadatak je dodijeljen komponenti koja je inicijalizirana s uređenim parovima (*lokacija, vrsta stabla*) i koja za svaki kadar prima lokaciju kamere i vraća listu modela koje je potrebno prikazati (zapravo se vraća mapa čiji ključevi su mreže (eng. *mesh*), a vrijednosti liste lokacija za tu mrežu kako bi se svi modeli s istim mrežama mogli iscrtati uzastopno, čime se minimizira broj promjena stanja *OpenGL-a*). Ta komponenta ima informaciju o rasponima udaljenosti koji su pridruženi modelima u različitim razinama detalja za sve objekte. Kod svakog poziva se prolazi kroz rešetku koja je definirana prilikom inicijalizacije – za svaku ćeliju se najprije odredi za koje modele je uopće moguće da se nađu u njoj (na temelju definiranih raspona) i obilaze se mape za svaki model koji se može naći u toj ćeliji, dohvaća se točna lokacija, određuje razina detalja te se rezultat dodaje u listu rezultata. Raspored stabala dobiven opisanom metodom je prikazan na sljedećoj slici.



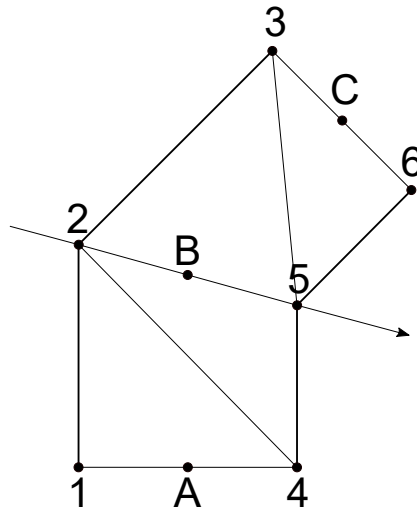
Slika 3.7. Raspored stabala s varirajućom gustoćom.

3.2 Generiranje ceste

U ovom poglavlju je opisan način generiranja ceste za zadani teren. Prvo pod-poglavlje opisuje algoritam generiranja trodimenzionalnih modela cesta i tunela. Algoritam dobiva niz točaka kroz koje cesta ili tunel moraju proći te na temelju njih postavlja cestu i tunele na teren. U drugom pod-poglavlju je opisano kako se određuje niz takvih točaka koji rezultira trajektorijom optimalnom s obzirom na zadane kriterije.

3.2.1 Generiranje geometrije ceste i tunela

Algoritam za generiranje poligonalne mreže ceste očekuje listu točaka kroz koje cesta mora proći. Točke se interpoliraju krivuljom Catmull – Roma uz zadanu rezoluciju čime se dobiva znatno veći broj točaka na krivulji. Svake dvije tako dobivene susjedne točke činit će jedan segment ceste. Svaki od segmenata je trapez (koji se sastoji od dva trokuta) kao na sljedećoj slici.



Slika 3.8. Geometrija ceste.

Na slici su vidljiva dva segmenta ceste – jedan između točaka A i B, i jedan između točaka B i C. Zadatak algoritma je odrediti pozicije vrhova 2 i 5 za svaki segment uz zadržavanje konstantne širine (preostali vrhovi sa slike su određeni u prethodnim ili budućim iteracijama). Da bi širina ceste bila konstantna, očito je da će stranica između vrhova 1 i 2 morati biti nešto duža od stranice između vrhova 4 i 5. Najprije se određuje vektor simetrale kuta između vektora v_{AB} i v_{BC} . Za svaki od ova dva vektora se određuje vektor koji pokazuje „desno“, a on je jednak vektorskom umnošku pripadnog vektora i vektora koji pokazuje u pozitivnom smjeru y – osi. Vektori se normaliziraju i zbroje te se rezultat također normalizira i na taj način se dobiva vektor simetrale. Potom se određuje kut kojeg zatvaraju vektori v_{BA} i v_{BC} . Pomoću tog kuta određuje se udaljenost točke B i vrha 5 te točke B i vrha 2 izrazom $0.5 * \text{širina} / \sin(\text{kut} / 2)$. Konačno, položaj vrha 5 se dobiva zbrojem vektora iz ishodišta do točke B i vektora simetrale kuta skaliranim s izračunatom udaljenošću. Položaj vrha 2 se dobiva zbrajanjem vektora iz ishodišta do vrha B s negiranim vektorom simetrale skaliranim istom udaljenošću.

Postupak određivanja normala je nešto kraći. Kao i kod određivanja vrhova, za vektore v_{AB} i v_{BC} potrebni su vektori koji pokazuju „desno“. Za svaki od vektora se također određuje i vektor u smjeru „gore“ – vektor normale na površinu. On se dobiva vektorskim umnoškom pripadnog vektora s njegovim vektorom „desno“ i normalizacijom. Ove dvije normale se potom zbrajaju i rezultat se normalizira, čime se dobivaju normale u vrhovima 2 i 5.

Na kraju je još potrebno odrediti teksturne koordinate. u – koordinate svih vrhova na lijevoj strani ceste će biti 0, a na desnoj 1. v – koordinata se mijenja s duljinom ceste i jednaka je za svaka dva susjedna vrha (primjerice, vrhovi 2 i 5 će imati istu v -koordinatu). Za svaku točku krivulje (A, B, C itd.) određuje se njena udaljenost od početka ceste uzastopnim zbrajanjem duljina segmenata. Dobivena vrijednost se dijeli „duljinom texture“ – vrijednošću koja predstavlja duljinu dijela ceste kojem ta tekstura pripada te se dobiva v – koordinata. v – koordinata će u jednom trenutku postati veća od 1, što je u redu jer je određeno da ceste imaju ponavljajuće texture.

Svaka od ovih vrijednosti zajedno s indeksima vrhova se sprema u zaseban spremnik vrhova *VBO* koji su potom pridruženi jednom spremniku takvih spremnika *VAO*.

Kako bi se ovako generiran model ceste mogao postaviti na teren, potrebno je djelomično prilagoditi visinu terena na dijelovima ispod ceste. Bez ovog koraka središnja krivulja ceste bi bila smještena točno na terenu, a njeni rubovi bi se u općem slučaju mogli naći ispod površine terena ili previše iznad površine. Teren se modificira izmjenom visinske funkcije, a ne poligonalne mreže terena, čime se omogućava naknadno generiranje modela terena u različitim razinama detalja. Visinska funkcija predstavlja stvarni teren, a trodimenzionalni model samo njegovu poligonalnu reprezentaciju. Teren poprima visinu središnje linije ceste na dijelovima ispod ceste i uskom području oko ceste, a visina dijelova u blizini ceste se interpolira s visinom terena. Postavljanje visine vrhova terena na visinu ceste u uskom području oko ceste je nužno kako bi se spriječilo probijanje ceste poligonima terena. To se događa ako je vrh terena pokraj ceste na visini većoj od središnje razine ceste. Minimalna širina područja s kojom neće biti takvog probijanja jest $r * \sqrt{2}$ pri čemu je r udaljenost dva susjedna vrha najdetaljnije mreže terena.



Slika 3.9. Model ceste.

Generiranje modela tunela se temelji na istim principima kao i generiranje modela ceste, stoga neće biti detaljnije objašnjeno. Samo ćemo spomenuti da se prilikom postavljanja tunela kroz uzvisinu područja ispred oba ulaza u tunel *iskapaju*, odnosno visina terena se smanjuje u određenom rasponu.

3.2.2 Određivanje trajektorije ceste

Zadatak određivanja trajektorije ceste dodijeljen je drugoj temeljnoj komponenti sustava. Ta komponenta kao ulaz dobiva visinsku funkciju, početnu i konačnu točku ceste te niz kriterijskih funkcija i ograničenja. Na temelju tih parametara pronalazi se trajektorija koja povezuje početnu i konačnu točku ne kršeći pritom nijedno od ograničenja.

Za rješavanje ovog problema korišteni su algoritmi pretrage opisani u prethodnom poglavlju. Temeljni algoritam u ovom dijelu rada je A^* zbog svojih svojstava optimalnosti i mogućnosti korištenja heuristika, i rad komponente sustava će biti objašnjen na primjeru tog algoritma. Pored algoritma A^* implementiran je i algoritam *Greedy best-first search*, koji će poslužiti za usporedbu s A^* . Nijedan od algoritama slijepe pretrage nije odabran za implementaciju zbog inferiornijih svojstava u odnosu na A^* .

Prilikom određivanja optimalne trajektorije kvaliteta trajektorije se određuje na temelju niza kriterija. Broj kriterija koje je moguće ugraditi u algoritam je

proizvoljan, a implementacija uvažava tri kriterija: duljinu trajektorije, nagib i zakrivljenost. Važnost ovih svojstava se određuje kriterijskim funkcijama, a suma iznosa svih funkcija daje cijenu rješenja – vrijednost koja govori u kojoj mjeri generirana trajektorija odgovara traženim kriterijima. Kriterijske funkcije su definirane na način da nesklad između trajektorije i odgovarajućeg kriterija producira veću cijenu, odnosno kvalitetnom trajektorijom se smatra ona čija je ukupna cijena niska. Ovakva formulacija je prikladna jer se pronalazak najniže cijene tada može obaviti algoritmima kojima je to temeljna osobitost, kao što je A^* .

Idealna trajektorija se sastoji od beskonačnog i kontinuiranog niza točaka koje su raspoređene u skladu sa zadanim kriterijima i ograničenjima. S obzirom da je zadatak algoritma generirati niz točaka i da se trajektorija postepeno gradi dodavanjem novih točaka u postojeću trajektoriju, očito je da je nemoguće postići beskonačan i kontinuiran niz. Da bi se problem mogao predstaviti grafom, potrebno je koristiti određene aproksimacije. Beskonačna i kontinuirana domena (xz ravnina) se diskretizira pomoću rešetke, a točke rešetke predstavljaju točke nove domene. Ovakva domena je prikladna za izvođenje algoritama traženja najkraćeg puta, a s obzirom da je potrebno pronaći trajektoriju na terenu konačnih dimenzija, u implementaciji je korišteno i ograničenje domene.

Premda je teren trodimenzionalan, i trajektorija koju je potrebno generirati u konačnici je trodimenzionalna, prikladno je koristiti dvodimenzionalnu domenu za pretragu. Ovim pristupom se otklanja problem beskonačnog faktora grananja nastalog zbog beskonačno velikog broja mogućnosti za visinu terena – besmisleno je generirati beskonačan broj točaka i potom tražiti jednu koja leži na terenu ako je ta informacija unaprijed poznata (potrebno je napomenuti da broj mogućnosti ipak nije beskonačan, već onoliko velik koliko rezolucija korištene visinske funkcije to dozvoljava – taj broj je i dalje prevelik za praktičnu primjenu). Traženje najkraćeg puta se na ovaj način svodi na traženje puta u dvodimenzionalnoj mreži uz mjeru udaljenosti temeljenu na euklidskoj udaljenosti, nagibu i zakrivljenosti. Udaljenost dvije točke rešetke (cijena) se izračunava zbrajanjem tri komponente koje potječu od odgovarajuće tri kriterijske funkcije:

- 1. Funkcija udaljenosti.** Za dvije točke mreže određuju se pripadne točke terena pomoću visinske funkcije te se određuje njihova euklidska udaljenost, d . Iznos funkcije udaljenosti se potom određuje umnoškom s korisnički zadanim faktorom.

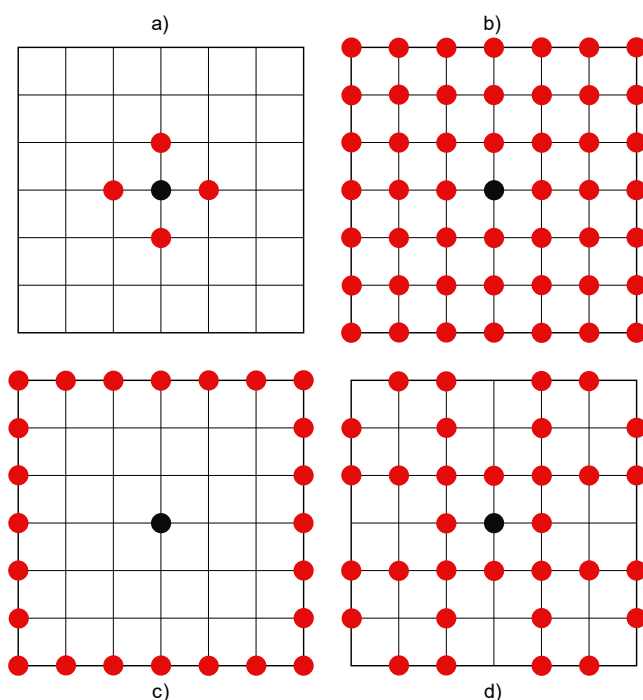
- 2. Funkcija nagiba.** Nagib s se određuje arkus sinusom omjera visinske razlike i euklidske udaljenosti dviju točaka terena. Ako je nagib veći od korisnički zadanog ograničenja, funkcija daje vrijednost pozitivne beskonačnosti. U ostalim slučajevima vrijednost s se transformira odabranim polinomom, množi faktorom d te se množi odabranim faktorom. Ove parametre zadaje korisnik i pomoću njih definira kriterije nagiba prikladne za promatrani teren i namjenu ceste.
- 3. Funkcija zakrivljenosti.** Zakrivljenost c je definirana kao kut kojeg zatvara dužina između trenutne i prethodne točke s dužinom između trenutne i buduće točke. Kao što je to slučaj i s funkcijom nagiba, zakrivljenost se također transformira odabranom funkcijom i množi odabranim faktorom čime se dobiva konačna cijena zakrivljenosti. Također, ako je zakrivljenost veća od maksimalne dopuštene zakrivljenosti, cijena će iznositi beskonačno, što će za posljedicu imati odbacivanje promatrane trajektorije.

Stanja u formulaciji problema će biti točke rešetke. Rješenje je tada čvor koji sadrži ciljno stanje (čvorovi sadrže reference na čvorove-roditelje i pomoću njih se rekonstruira cjelokupna putanja). Zbog nesavršenosti prikaza realnih brojeva u računalu, nije prikladno stanja definirati kao točke s realnim koordinatama. S takvom definicijom postoji mogućnost da ista točka bude zapisana na više načina, zbog čega bi algoritam radio sa znatno većim brojem stanja. Taj problem se rješava ograničenjem da točke imaju cjelobrojne koordinate i uvođenjem funkcije koja preslikava točke s cjelobrojnim koordinatama u točke s realnim koordinatama. Prijelaz između stanja je dodatak nove točke u trajektoriju. Cijena prijelaza je vrijednost dobivena na temelju kriterijskih funkcija kako je opisano.

Uz ovako definiranu funkciju cijene problem je moguće riješiti pomoću nekih od algoritama slijepe pretrage, primjerice pomoću algoritma UCS. U ovom konkretnom problemu ipak su poznate dodatne informacije o rješenju, stoga ih je moguće ugraditi u pretragu u obliku heuristike i povišiti učinkovitost pretrage. Dodatna informacija koja će ovdje biti od koristi je udaljenost krajnje točke trajektorije od ciljne točke. Očekuje se da će pretraga biti brža ako se najprije razmatraju one točke koje su bliže cilju, stoga je heuristika definirana kao euklidska udaljenost između promatrane točke i ciljne točke. Ova heuristika nije zahtjevna za

izračun, i osigurava optimalnost rješenja svojim svojstvom dopustivosti: nije moguće da heuristika precijeni udaljenost jer uvijek daje udaljenost točaka na ravnoj liniji.

Preostaje još definirati funkciju susjedstva *successor* o kojoj će najviše ovisiti kvaliteta rješenja i trajanje izračuna. Nekoliko funkcija susjedstva je ilustrirano na sljedećoj slici.



Slika 3.10. Skupovi sljedećih točaka za različite funkcije susjedstva. Jednostavno susjedstvo (a), susjedstvo *all* (b), susjedstvo *farthest* (c) i susjedstvo *nearest unique* (d).

Funkcija susjedstva mora biti definirana tako da osigurava dohvatljivost ciljnog stanja iz početnog stanja. Najjednostavnija funkcija koja omogućava dohvat rješenja u općem slučaju generira četiri susjedne točke, kao na slici 3.10. a). Iz trenutačne točke trajektorije na taj način je moguće prijeći u jednu od te četiri točke (neke točke će ipak biti odbačene zbog ograničenja nagiba i zakrivljenosti, ovisno o trenutačnoj situaciji). Ako je susjedstvo premalo, javljaju se artefakti smjera – postaje očito da dijelovi ceste mogu biti usmjereni jedino u smjerovima istok – zapad i sjever – jug. Problem ograničenog broja smjerova se rješava povećanjem susjedstva. Primjerice, u implementaciji je podržana mogućnost definiranja veličine kvadrata sa središtem u trenutačnoj točki iz kojeg se uzimaju susjedne točke (funkcija susjedstva *all*), prema slici 3.10. b).

Broj smjerova koje je moguće definirati na ovaj način je znatno veći, ali s povećanjem susjedstva i faktor grananja se također povećava. Na istoj slici je moguće uočiti da neke točke daju iste smjerove kao druge točke susjedstva. Primjerice, ako središnjoj točki dodijelimo koordinate (0, 0), točke s koordinatama (1, 1) i (2, 2) će rezultirati istim smjerovima. Kako bi se faktor grananja smanjio, a broj mogućih smjerova zadržao, u susjedstvo se uključuju samo točke s jedinstvenim smjerovima. Ovo je moguće učiniti na više načina, a implementacija podržava dva. Prva mogućnost je zadržati samo točke na rubu kvadrata unutar kojeg se definira susjedstvo, prema slici 3.10. c) (susjedstvo *farthest*).

Druga podržana mogućnost je zadržati najbliže točke s jedinstvenim smjerom, prema slici 3.10. d) (susjedstvo *nearest unique*). Točke ovog susjedstva se učinkovito identificiraju koristeći činjenicu da im je najveći zajednički nazivnik apsolutnih iznosa koordinata jednak 1, a ta provjera se izvršava pomoću Euklidovog algoritma:

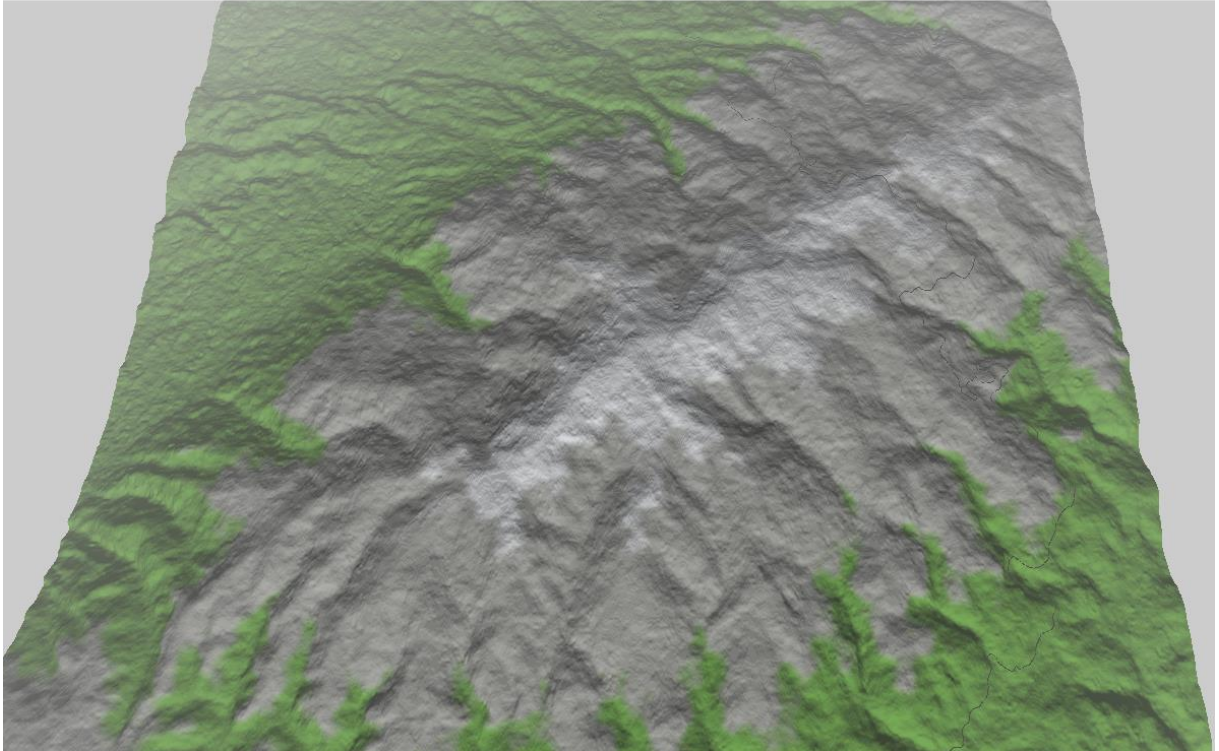
```
// greatest common divisor
int gcd(a, b) {
    return b == 0 ? a : gcd(b, a % b)
}
```

Isječak koda 3.3. Euklidov algoritam.

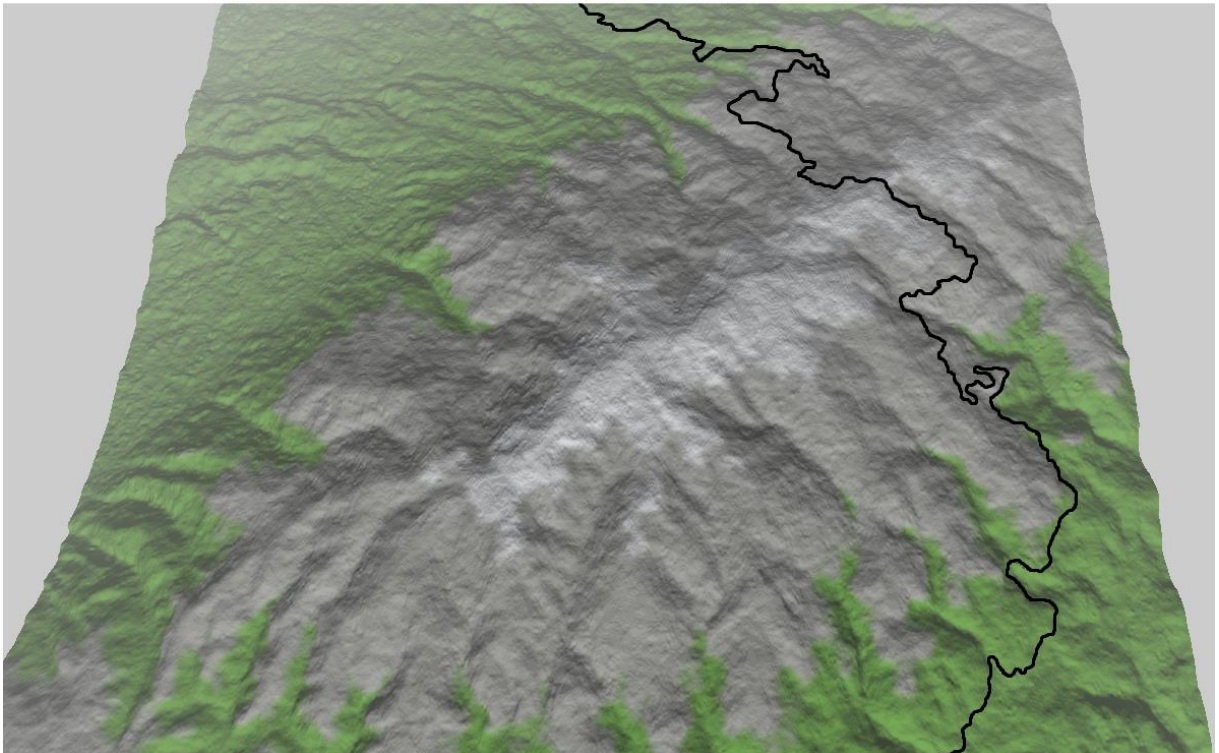
U nastavku će biti prikazan rad algoritma A^* na ranije prikazanom terenu Medvednice veličine 12 x 12 kilometara. Rezolucija rešetke s točkama je 15 metara, zbog čega je broj stanja jednak 640000. Komponente cijene su definirane kriterijskim funkcijama kako slijedi:

- udaljenost: $c_d = d$
- nagib: $c_s = 80 * d * s^2$
- zakrivljenost: $c_c = 10 * c^3$

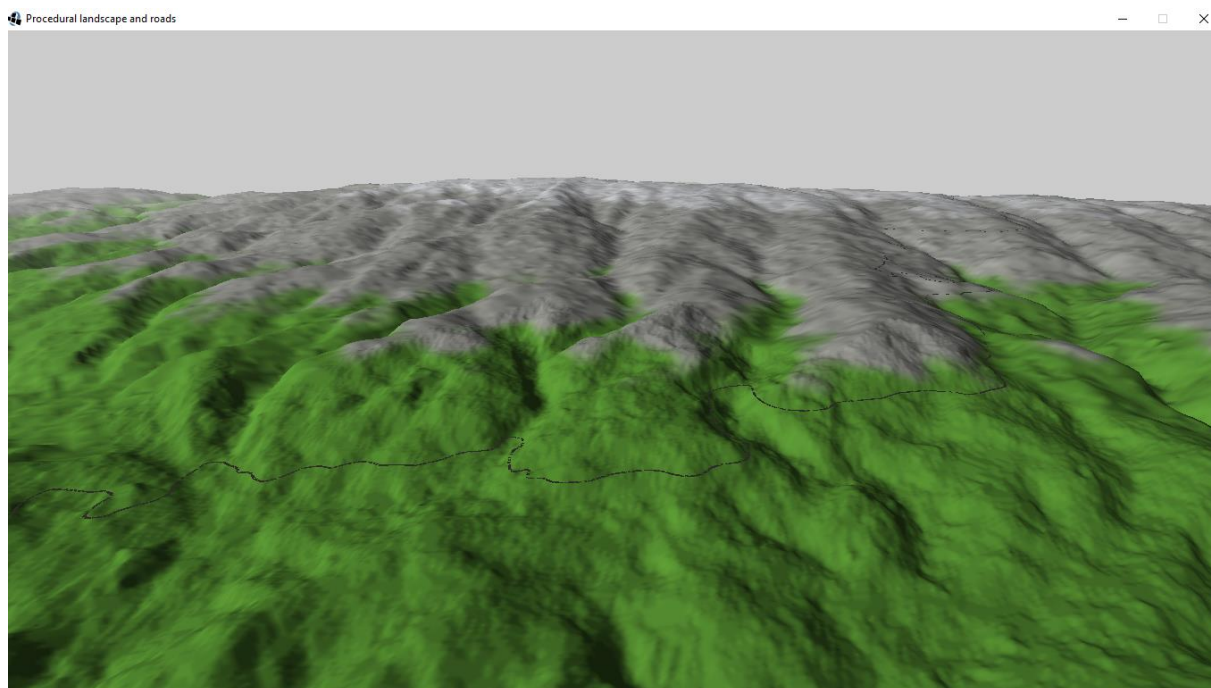
Ukupna cijena jednaka je zbroju $c_d + c_s + c_c$. Maksimalan dozvoljeni nagib je 9%. Maksimalna zakrivljenost je 1.75 radijana, odnosno 100 stupnjeva. Najveća udaljenost od točke čiji susjedi se razmatraju je 3, a funkcija susjedstva je *farthest*. Izračun trajektorije algoritmom A^* na računalu s procesorom *Intel Core i7 – 2670QM*, 2.2 Ghz i 4 Gb RAM-a traje 73.16 sekundi, a cijena trajektorije iznosi 34322 apstraktne jedinice. Rezultat je prikazan na sljedećem nizu slika.



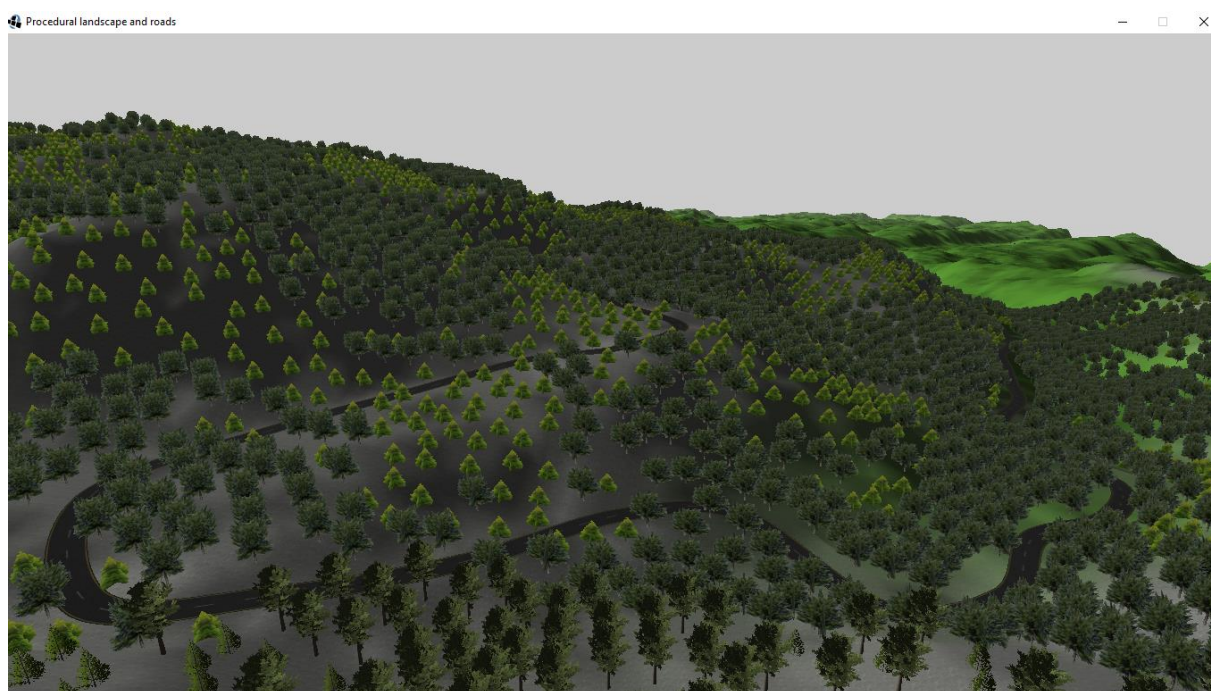
Slika 3.11. Konačna trajektorija ceste (cesta je istaknuta na slici 3.12., stoga ju se može koristiti kao referencu jer je cesta prikazana na velikoj udaljenosti od kamere).



Slika 3.12. Teren s istaknutom trajektorijom ceste.



Slika 3.13. Približen prikaz – trajektorija se prilagođava obliku terena.

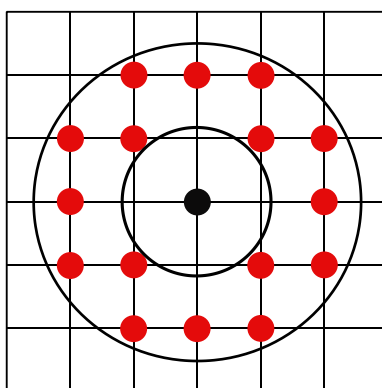


Slika 3.14. Dio trajektorije uz prikazanu vegetaciju.



Slika 3.15. Cesta iz perspektive vozača.

Implementacija podržava i mogućnost generiranja tunela. Funkcija susjedstva se modificira na način da osim bliskog skupa točaka koje služe generiranju novih odsječaka ceste generira i skup točaka na većoj udaljenosti pomoću kojih će se definirati krajnje točke tunela. Točke se mogu birati u krugu zadanog radijusa, a u implementaciji se točke odabiru iz kružnog vijenca čiji unutarnji radijus predstavlja minimalnu, a vanjski maksimalnu duljinu tunela. Točke se odabiru na taj način jer se u suprotnosti faktor grananja povećava u prevelikoj mjeri. Točke koje generira ovako modificirana funkcija susjedstva su vidljive na sljedećoj slici.

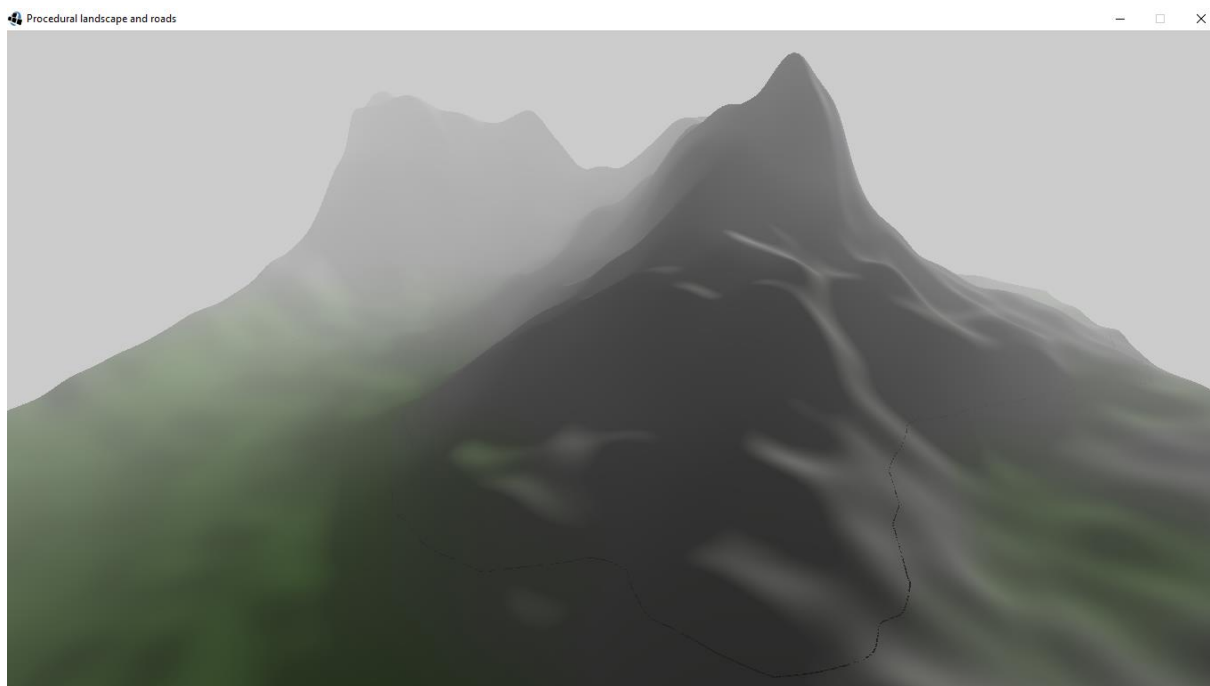


Slika 3.16. Rezultat funkcije susjedstva za tunele.

Ako je potrebno generirati tunele s velikim rasponom podržanih duljina, ova modifikacija neće biti dovoljna za smanjenje složenosti. Implementacija podržava dodatno ograničenje broja susjednih točaka stohastičkim uzorkovanjem točaka iz kružnog vijenca sa zadanim radijusima. Uzorkovanje je ostvareno Fisher – Yatesovom metodom koja generira nepristrane permutacije potrebnog broja elemenata i sprema ih u izvornu listu. Permutacija se predstavlja *proxy* objektom kojemu je potporna struktura ta lista čime se izbjegava nepotrebna alokacija memorije. Ovdje je važno napomenuti da algoritam koji koristi ovu modifikaciju više nije optimalan, čak ni potpun. Osim toga, pretraga je i nedeterministična. Pokazuje se da algoritam s ovom modifikacijom ipak uspijeva pronaći kvalitetna rješenja uz razumno velik broj dopuštenih točaka.

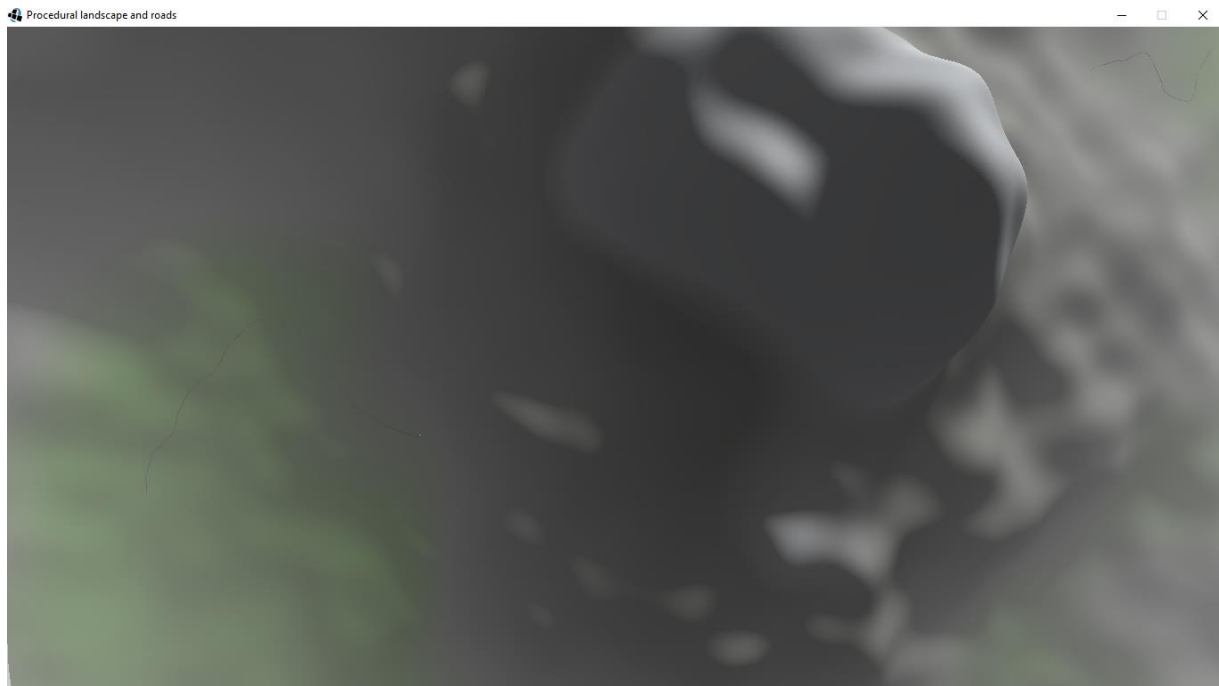
Ako je generiranje tunela podržano, potrebno je izmijeniti i funkciju cijene. Prvi uvjet koji mora biti ispunjen jest da funkcija generira vrijednost beskonačno ako između trenutne i promatrane točke nije moguće generirati tunel. Ta situacija će nastati ukoliko se između točaka ne nalazi uzvisina. Postojanje uzvisine se provjerava višestrukim uzorkovanjem visinske mape – ako je iznos uzorkovane visine veći za zadani prag od visine odgovarajuće točke na zamišljenom pravcu koji povezuje trenutačnu i promatranu točku za sve uzorke, tada se zaključuje da se radi o uzvisini. Ukoliko se otkrije postojanje uzvisine između točaka, komponente cijene koje se odnose na duljinu, nagib i zakrivljenost se izračunavaju na potpuno isti način kao što je to slučaj kod ceste, samo s drugim parametrima koji određuju preferenciju prema pojedinim kriterijima.

U nastavku je prikazan primjer dizajniranja trajektorije na drugačijem terenu. Teren je generiran sintetički uporabom *Simplex* šuma, i sadrži planinu koja dijeli teren na dvije nizine. Krajnje točke trajektorije su postavljene u ove dvije nizine, tako da ih je moguće povezati jedino pomoću strmih cesta ako se ne dopuštaju tuneli. Teren je dimenzija 10 x 35 kilometara, rezolucija mreže je 100 metara, funkcija susjedstva *nearest unique*, maksimalan nagib ceste je 15%, tuneli nisu omogućeni, a ostali parametri su jednaki kao u prethodnom primjeru. Trajanje izračuna je 2.29 sekundi, a cijena trajektorije 37036 apstraktnih jedinica. Rezultat je prikazan na sljedećoj slici.

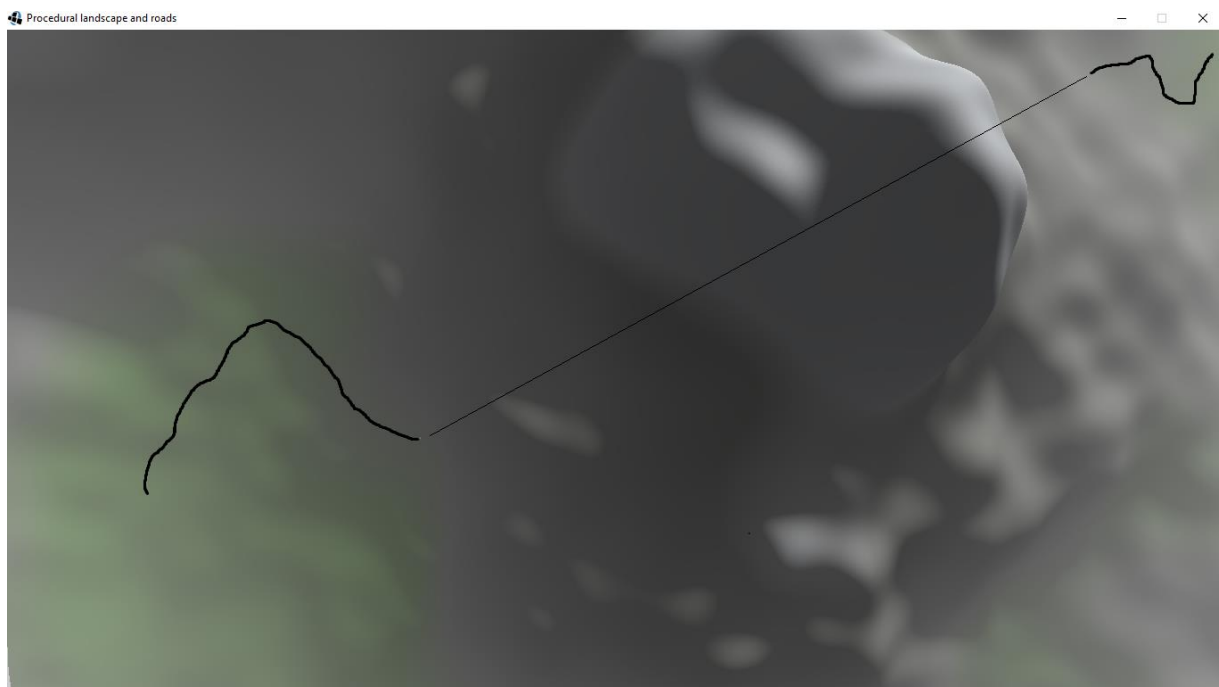


Slika 3.17. Cesta zbog ograničenja nagiba obilazi planinu.

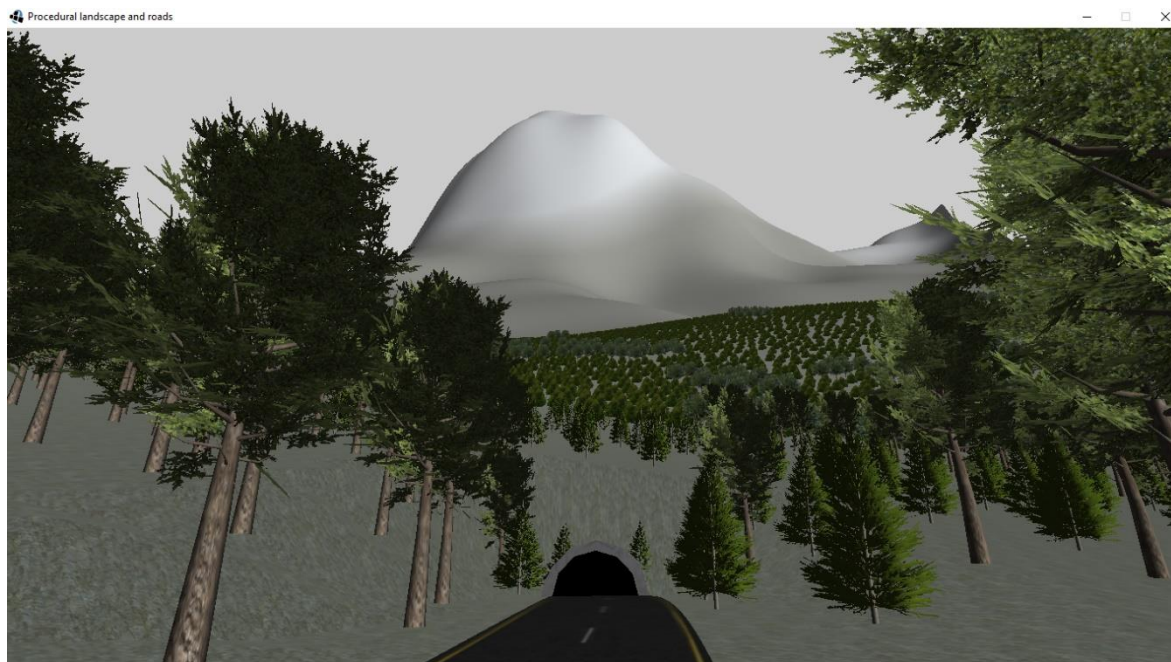
Na ovom terenu nije moguće postići manje strmu cestu. Postavljanjem ograničenja nagiba na 10% algoritam ne uspijeva pronaći rješenje. Ako se omogući postavljanje tunela, algoritam uspijeva pronaći trajektoriju s maksimalnim nagibom 10%. Komponente cijene tunela su definirane na način da ukupna cijena tunela bude veća od cijene ceste iste duljine, čime se preferiraju ceste, a tuneli se generiraju isključivo ako više nije moguće nastaviti trajektoriju cestom. Dozvoljena duljina tunela iznosi između 8 i 10 kilometara (iznosi su podešeni za konkretan teren), a broj mogućih točaka za tunele nije ograničen. Za pronalazak rješenja s tunelom ukupne cijene 21610 jedinica potrošene su 23 minute i 25.82 sekunde. Rješenje je prikazano na sljedeće tri slike.



Slika 3.18. Trajektorija s jednim tunelom (trajektorija je istaknuta na slici 3.19., stoga ju se može koristiti kao referencu jer je trajektorija prikazana na velikoj udaljenosti od kamere).

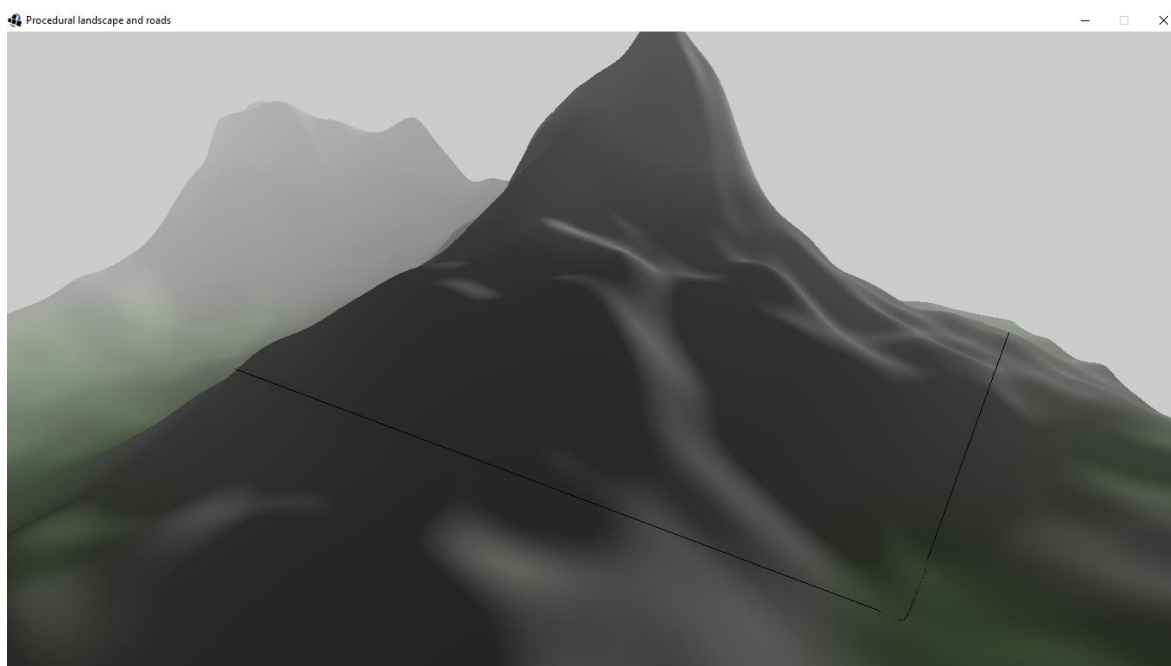


Slika 3.19. Trajektorija s označenim cestama (deblje linije) i tunelom (tanka linija).



Slika 3.20. Tunel iz perspektive vozača.

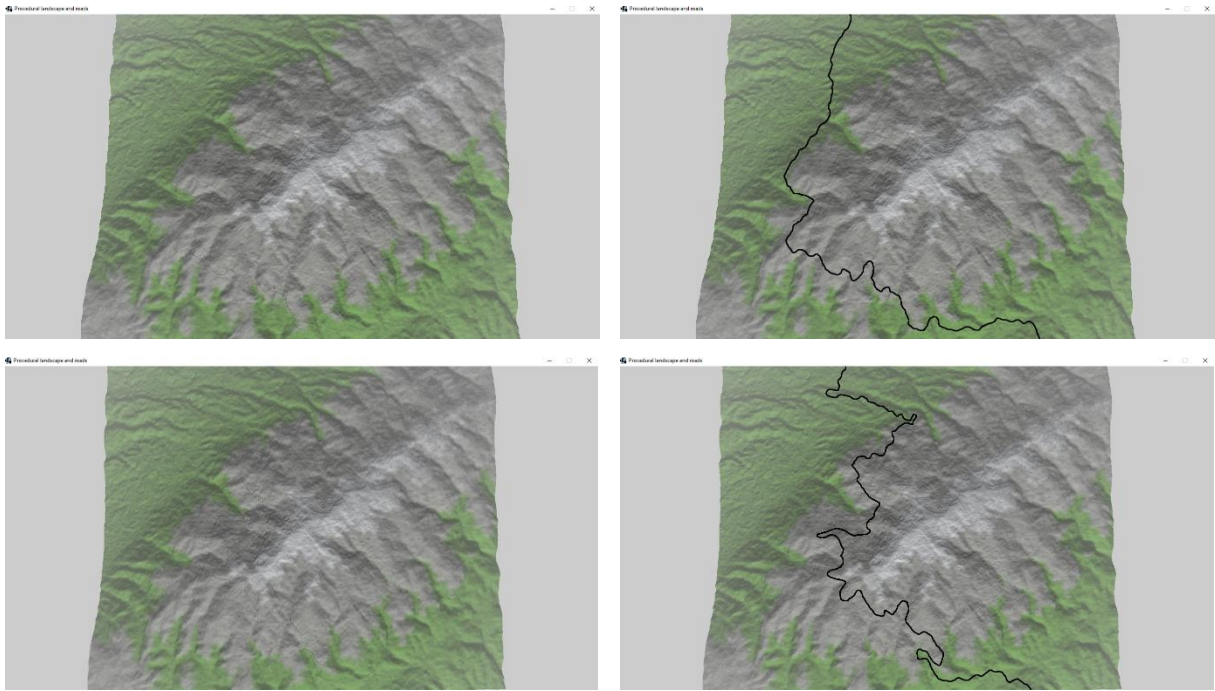
Dobiveno rješenje je optimalno, ali njegov izračun je dugotrajan. Ako se uvede opisana modifikacija ograničenja broja mogućih točaka za tunele na 100, dobiva se rješenje kvalitete približne kvaliteti optimalnog rješenja u više od 70 puta kraćem roku. Dobivena trajektorija sadrži dva tunela, cijena joj je 22713 jedinica, a za izračun su utrošene 19.82 sekunde. Rješenje je prikazano na sljedećoj slici.



Slika 3.21. Rješenje s dva tunela. Crnim linijama su naznačene putanje tunela.

Algoritam *Greedy best-first search* je pokrenut s oba prikazana terena uz iste parametre koji su dani algoritmu A^* , ali rješenje nije pronađeno unutar prvih 30 minuta, stoga je algoritam zaustavljen. Rješenje očito postoji u oba slučaja, ali pohlepni algoritam ga ne pronalazi zbog prirode problema i samog algoritma. Algoritam „slijepo“ prati heuristiku, odnosno pretpostavlja da heuristika u većini slučajeva dovodi do stanja bližeg rješenju. Za određene terene to je doista i ispunjeno – primjerice, na pretežito ravnom terenu bez strmih uspona bilo koja susjedna točka (izuzev onih za čiji dohvat cesta mora naglo promijeniti kut) je validan izbor sljedećeg stanja. Pohlepni algoritam u takvom slučaju može generirati trajektoriju u onoliko koraka koliko je minimalno potrebno da se prijeđe iz početnog u konačno stanje. Ako se na tom putu nađe određeni mali broj strmih uspona, pretraga će se djelomično usporiti (jer se za svaki prestrm uspon pretraga mora vratiti na prethodne čvorove), ali još uvijek će pronalaziti rješenje u razumnom vremenu. Ako se, međutim, teren sastoji gotovo isključivo od strmih uspona s malo ravnih dijelova, i ako je prostor stanja k tome relativno velik kao što je to slučaj u opisanim primjerima, pretraga prečesto nailazi na slijepe sljedove stanja i velik udio vremena se troši na njihovo istraživanje (algoritam zapinje u lokalnim optimumima). Ovaj problem bi se mogao ublažiti pomoću informiranije heuristike koja bi primjerice bila u mogućnosti detektirati blizinu strmih uspona i modificirati procjenu u skladu s tom informacijom. Time bi se i povisila složenost izračuna heurističke procjene o čemu je također potrebno voditi računa. Pohlepni algoritam je zbog ovih razloga prikladniji za rješavanje problema s malim brojem lokalnih optimuma kod kojih je moguće definirati jednostavnu heuristiku, ili prostor stanja nije prevelik. Ako je potrebno pronaći bilo kakvo rješenje u vrlo malom broju koraka, pohlepni algoritam je najbolji izbor.

Kako bi se pohlepni algoritam mogao usporediti s algoritmom A^* , rezolucija mreže na terenu Medvednice je smanjena s 15 na 50 metara te su tako pokrenuta oba algoritma. A^* pronalazi rješenje u 2.694 sekunde, a pohlepni algoritam u više od dvadeset puta kraćem roku, 0.126 sekundi. Kvaliteta rješenja je zbog toga očekivano viša u slučaju A^* s cijenom 26645 jedinica, za razliku od rješenja pohlepnog algoritma s cijenom 39014 jedinica. Rezultantne trajektorije su prikazane u nastavku.



Slike 3.22 – 3.25. Trajektorija A^* (gore lijevo), istaknuta trajektorija A^* (gore desno), trajektorija pohlepnog algoritma (dolje lijevo) i istaknuta trajektorija pohlepnog algoritma (dolje desno).

Na slici s putanjom dobivenom pomoću A^* se može uočiti razlika u odnosu na putanju dobivenu istim algoritmom uz veću rezoluciju. Radi se o tome da zbog manje rezolucije algoritam ne uspijeva pronaći putanju s finijim zavojima koji su potrebni kako bi cesta mogla prijeći uzvisinu na njenom najstrmijem dijelu uz zadana ograničenja nagiba. Premda ovo rješenje ima prividno nižu cijenu od rješenja dobivenog uz veću rezoluciju, ne može se zaključiti da mu je kvaliteta doista viša upravo zbog različitih parametara izvođenja. Cijene rješenja je smisleno uspoređivati (u ovoj konkretnoj formulaciji problema) jedino za rješenja dobivena uz iste parametre.

Zaključak

Ne postoji *najbolja* metoda za prikaz krajolika. Prikladnost metode ovisi o onome što se nastoji postići te je to primarni razlog njihove brojnosti. Prije odabira je potrebno razmotriti kriterije koji moraju biti ispunjeni i karakteristike različitih pristupa – mogućnost izvođenja u realnom vremenu, preciznost prikaza, opterećenje centralnog procesora ili grafičke kartice, memorijsko zauzeće. Prikladnost odabira se također i mijenja s vremenom – stariji algoritmi prilagođeni tadašnjim mogućnostima hardvera u današnje vrijeme su pretežito neupotrebljivi. Na primjeru implementiranog sustava je pokazano kako je moguće razdvojiti karakteristike na bitne i nebitne za konkretnu primjenu i u potpunosti prilagoditi algoritam samo bitnim kriterijima. Za ovaj konkretan sustav memorijsko zauzeće i trajanje faze pred-procesiranja su proglašeni nebitnim stavkama te se na njima nije štedjelo. To je omogućilo bolje ispunjenje bitnih kriterija – minimalnu komunikaciju CPU i GPU, zanemarivo opterećenje CPU i postojanje razina detalja. S današnjim hardverom još uvijek je nemoguće ostvariti sve komponente – potrebno je izdvojiti skup najbitnijih za određenu primjenu.

Dizajniranje trajektorije ceste na kompleksnim terenima je zahtjevan zadatak zbog ogromnog broja mogućnosti izbora. Pokazalo se da predstavljanje problema grafom i obilazak tog grafa algoritmima pretrage daje relevantne rezultate ostvarive u razumnom vremenu. Opisano je kako uvažiti ograničenja i kako definirati niz kriterija koji će voditi pretragu. Sustav procjenjuje kvalitetu trajektorije na temelju tri kriterija – duljine, nagiba i zakrivljenosti – kojima se određuje važnost pomoću niza parametara. Ako je primjerice niži nagib preferiran u odnosu na duljinu, moguće je postići takvu trajektoriju smanjenjem važnosti duljine i povećanjem važnosti nagiba pomoću parametara. Na primjeru implementiranog sustava je pokazano kako ograničenja i kriteriji koji se mogu ugraditi u algoritam mogu biti potpuno proizvoljni – ako je, primjerice, poželjno konstruirati trajektoriju koja maksimalno pokriva populacijska središta na putu između početne i krajnje točke, dovoljno je dodati kriterijsku funkciju koja mijenja cijenu trajektorije na temelju podataka o gustoći populacije i definirati proizvoljan niz parametara za određivanje preferencije te pokrivenosti.

Literatura

- [1] Duchaineau M., Wolinsky M., Sigeti D. E., Miller M. C., Aldrich C., Mineev-Weinstein M. B. *ROAMing Terrain: Real-time Optimally Adapting Meshes*. VIS97 IEEE Visualization '97 Conference, Phoenix USA, 1997, 81-88.
- [2] de Boer W. H. *Fast Terrain Rendering Using Geometrical MipMapping*. 2000. https://flipcode.com/archives/article_geomipmaps.pdf, pristup 1.5.2018.
- [3] Losasso F., Hoppe H. *Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids*. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004), 2004, 769–776.
- [4] Asirvatham A., Hoppe H. *Terrain Rendering Using GPU-Based Geometry Clipmaps*. GPU Gems 2, 2005, 27–45.
- [5] Russel S., Norvig P. *Artificial Intelligence: A Modern Approach*. Third Edition. Prentice Hall. 1995.
- [6] Galin E., Peytavie A., Maréchal N., Guérin E., *Procedural Generation of Roads*, Comput. Graph. Forum (Eurographics) 29(2), 2010, 429–438.
- [7] Martek C. *Procedural generation of road networks for large virtual environments*. Thesis. Rochester Institute of Technology. 2012.

ODREĐIVANJE PUTANJE CESTE NA TERENU

Sažetak

Rad opisuje metodu određivanja optimalne trajektorije ceste koja povezuje početnu i konačnu točku na proizvoljnom terenu na temelju niza kriterija i ograničenja. Prvo poglavlje se bavi postupcima zapisa i prikaza terena u računalu. U drugom poglavlju su opisani algoritmi pretrage na kojima je izračun temeljen. Treće poglavlje opisuje razvijeni sustav i dobivene rezultate.

Ključne riječi: računalna grafika, algoritmi pretrage, višekriterijska optimizacija, teren, razine detalja, proceduralno generiranje.

ROAD TRAJECTORY CALCULATION FOR ARBITRARY TERRAINS

Abstract

This paper describes a method of determining the optimal trajectory of the road that links the initial and final point on arbitrary terrains based on a series of criteria and constraints. The first chapter deals with various terrain data representations and visualization in the computer. The second chapter describes the search algorithms on which the calculation is based. The third chapter describes a developed system and the results obtained.

Keywords: computer graphics, search algorithms, multi-criteria optimization, terrain, level of detail, procedural generation.