Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Juš Lozej

Generiranje svetlobi prilagojenih dreves z uporabo genetskih algoritmov

diplomsko delo

univerzitetni študijski program prve stopnje
računalnistvo in informatika

Mentor: prof. dr. Marko Bajec

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:
Generiranje svetlobi prilagojenih dreves z uporabo genetskih algoritmov

Tematika naloge:

V okviru diplomske naloge izdelajte algoritem za generiranje dreves. Pri tem upoštevajte, da obstaja vir svetlobe, kateremu se rast dreves prilagaja (kot bi se v naravi). Algoritem implementirajte s spletno aplikacijo, ki bo omogočala nastavljanje parametrov gradnje ter ponujala ustrezno vizualizacijo dogajanja. Uporabnik naj ima možnost spremljanja podatkov, ki so ključni pri gradnji drevesa. Problematiko poskusite na posameznem in več drevesih.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Juš Lozej sem avtor diplomskega dela z naslovom:

Generiranje svetlobi prilagojenih dreves z uporabo genetskih algoritmov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Marka Bajca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 25. avgusta 2015 Podpis avtorja:

Zahvaljujem se družini, ki me je v času študija vedno podpirala in spodbujala. Prav tako bi se rad zahvalil vsem članom kluba, ki so stresnost študija izjemno zmanjšali. Posebna zahvala gre tudi mentorju prof. dr. Marku Bajcu, ki je vstopil kot mentor, ko nisem mislil, da bom mentorja dobil.

Vsem mojim.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
GA	genetic algorithm	genetski algoritem
FF	fitness function	funkcija uspešnosti
SUS	stochastic universal sampling	univerzalno stohastično vzorčenje
RWS	roulette-wheel selection	izbor z ruletnim kolesmo
WebGL	web graphics library	spletna grafična knjižnica
HTML	HyperText Markup Language	jezik za označevanje besedila

Povzetek

V sklopu tega diplomskega dela je bila izdelana spletna aplikacija za samodejno generiranje svetlobi prilagojenih dreves. Za prilagajanje dreves svetlobi uporablja genetske algoritme. Za proceduralno generiranje dreves uporablja knjižnico procTree, ki predstavlja primer parametričnega modela za generiranje dreves. V prvem delu diplomskega dela bo predstavljena teorija, na podlagi katere je nastala končna aplikacija. Ta vsebuje: teorijo o generiranju dreves in teorijo o genetskih algoritmih. Temu sledi poglavje, ki opisuje implementacijo aplikacije. Ta poda način uporabe, opiše postopek generiranja dreves ter prilagajanja teh z uporabo genetskih algoritmov. Na kratko je opisana tudi izdelava uporabniškega vmesnika, paralelizacije in prikazovalnika. Zadnje poglavje prikazuje rezultate, dobljene z aplikacijo. Poglavje predstavi nekaj splošnih primerov generiranih dreves, opiše vpliv parametrov na generiranje dreves, prikaže interakcijo med drevesi in na koncu predstavi še nekaj težave ter možnih nadgradenj aplikacije.

Ključne besede: proceduralno generiranje dreves, genetski algoritmi, spletna aplikacija, HTML5, WebGL.

Abstract

As a part of this bachelor's theses a web application for automatic generation of light adapted trees was made. Genetic algorithms are used to adapt the trees towards the light source. The library ProcTree, which is an example of a parametric model for generating trees, is used to generate the trees. The first part of the theses contains the theory used in the creation of the application. The chapter consists of the following topics: tree generation theory and genetic algorithm theory. This chapter is followed by descriptions of the implementation of the application. Specifically the chapter talks about the usage of the application, it describes the process of tree generation and how this generation is influenced by genetic algorithms. It also describes the creation of the user interface and the renderer, and how the application is parallelized. The last chapter shows the results of the application. First some basic result are shown, than the effects of some parameters on the generation are explained, after that some interaction between trees are shown. The theses is concluded by a few problems and possible improvements of the application.

Keywords: Procedural generation of trees, genetic algorithms, web aplication, HTML5, WebGL.

Poglavje 1

Uvod

V moderni računalniški grafiki je pogosta uporaba vnaprej izračunanih vsebin postala precej popularna. Ta pristop je viden predvsem v industriji iger, ki zahteva izrisovanje čim bolj realističnih scen v pravem času. Novejši pogoni, namenjeni izdelavi iger, zapisujejo svetlobo, izračunano z sledenjem žarkov, v barvne teksture modelov. To nato omogoča prikazovalniku izrisovanje scene v pravem času z izjemno realističnimi rezultati.

Tak pristop smo želeli uporabiti za generiranje oblike dreves. Pri standardnem pristopu bi nek grafični oblikovalec izdelal nabor generiranih dreves in jih nato, na nek določen način, postavil v sceno. Za velike scene je ta pristop zelo zamuden in dolgotrajen. Nabor dreves, ki jih je oblikovalec sprva izdelal, tudi ni nujno najbolj ustrezen. Zato bi bilo dobro imeti aplikacijo, ki bi lahko generirala velik nabor dreves in le-te nato prilagodila okolju. Pri tem smo predpostavili, da izdelava dreves ni časovno omejena, saj je pomemben le končni rezultat.

Aplikacija predpostavlja, da je drevo prilagojeno okolju, ko ima čim več listov neposredno izpostavljenih svetlobi. Prilagajanje poteka z uporabo genetskih algoritmov. Genetski algoritmi nam nudijo neizčrpno preiskovanje domene možnih dreves. Drevesa nudijo veliko raznolikost v svoji obliki in je zato temu primerno tudi iskalni prostor velik. Genetski algoritmi so tudi zelo adaptivni in relativno robustni, zato se zdijo kot dober način preiskovanja

takšnih prostorov.

Poglavje 2

Teorija

V tem poglavju je opisano teoretsko ozadje uporabljenih metod. Opisane so tudi nekatere metode, ki niso bile uporabljene v končni različici, vendar so bile v fazi načrtovanja diplomskega dela mišljene kot možne alternative končni implementaciji. Opisanih je nekaj metod generiranja dreves in splošna teorija genetskih algoritmov.

2.1 Generiranje dreves

Za generiranje dreves obstaja veliko metod. Te metode se ponavadi razdelijo na dve osnovni komponenti: generiranje oblike/skeleta drevesa in izdelavo geometrijske mreže drevesa. V nadaljevanju bodo opisane metode generiranja dreves z uporabo Lindemayerjevih sistemov (L-sistemov), kolonializacije prostora ter splošnih parametričnih modelov.

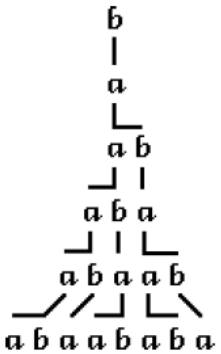
2.1.1 L-sistemi

L-sistemi ali Lindenmayerjevi sistemi so prepisovalni sistem namenjen opisovanju oblik fraktalnih objektov. Imenovani so po svojem izumitelju Astridu Lindenmayerju. L-sistem vsebuje abecedo simbolov, nabor pravil, ki razširjajo vsak simbol v daljše nize, začetni niz, imenovan "aksiom", ki deluje kot inicializacija sistema, ter način izrisovanja dobljenih nizov simbolov v geometrijske

oblike. Lindenmayer jih je razvil za opisovanje obnašanja rastlinskih celic in njihovega procesa rasti med razvojem rastline.

DOL-sistem

DOL-sistemi predstavljajo najpreprostejšo obliko L-sistemov. So deterministični in se pri generiranju ne ozirajo na okoliščine (angl. context-free). Delovanje DOL-sistemov najlažje opišemo s primerom.
Imamo abecedo sestavljeno iz dveh znakov: a in b. Vsak znak je povezan s svojim prepisovalnim pravilom. Pravila nam povejo, s katerim znakom ali nizom znakov prepisati izvorni znak. Znak a ima pravilo $a \rightarrow ab$, torej a zamenjamo z nizom ab, znak b pa ima pravilo $b \rightarrow a$, ki nam pove, da znak b prepišemo z znakom a. Proces prepisovanja se začne z začetnim nizom imenovanim aksiom. V našem primeru bo to b. V prvem koraku prepisovanja se znak b prepiše z znakom a. V naslednjem koraku se nato a prepiše z nizom ab. Potem se znaka a in b hkrati prepišeta v ab in a, tako da dobimo nov niz aba. Nato se ta niz prepiše v $abaab$. To potem nadaljujemo poljubno število iteracij. Primer je prikazan na sliki 2.1.

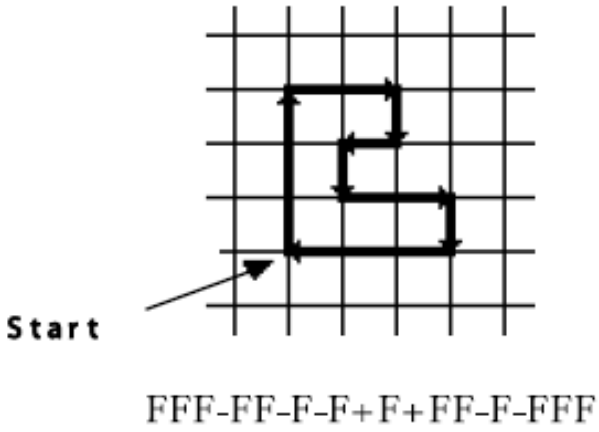


Slika 2.1: Primer prepisovanja DOL-sistema [1]

Iz primera lahko razberemo, da vsak DOL-sistem vsebuje nabor znakov, označenih z V, nabor prepisovalnih pravil, označenih s P, ter aksiom, označenega z ω.

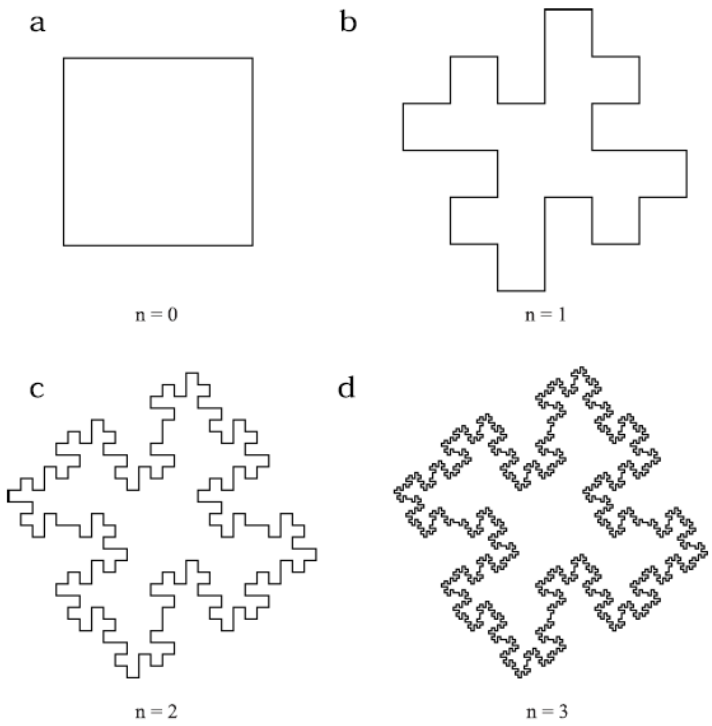
Uporaba L-sistemov v grafiki

Abecedo L-sistemov lahko razumemo kot ukaze za vodenje risanja. Prepisovanje nam nudi metodo rekurzivnega risanja enakih nizov in zato je zelo uporabno za risanje fraktalov. V ta namen definiramo abecedo:
F - Premakni pisalo naprej za eno enoto, pri tem za sabo riši črto.
f - Premakni pisalo naprej in za sabo ne riši črte.
$-$ - Spremeni smer pisala za kot δ v levo smer.
$+$ - spremeni smer pisala za kot δ v desno smer.
Kot δ je privzeto nastavljen na 90 stopinj. Glede na zgornja pravila bi se niz $FFF-FF-F-F+F+FF-FFF$ izrisal v naslednjo sliko:



Slika 2.2: Interpretacija niza [1]

Vzemimo za primer L sistem z aksiomom ω: $F-F-F-F$ ter pravilom p: $F \rightarrow F-F+F+FF-F-F+F$. V ničti iteraciji prepisovanja dobimo le z aksiomom definiran kvadrat (**a**). S prvo iteracijo (**b**) prepisovanja se vsaka stranica kvadrata nadomesti z zgoraj definiranim pravilom, torej vsak F v aksiomu se zamenja z nizom v pravilu. Z nadaljnjim prepisovanjem (**c/d**) se vsak F v prepisovalnem pravilu zopet prepiše z definiranim pravilom.



Slika 2.3: Primer uporabe L-sistemov za izrisovanje fraktalov. Število n nam pove kolikokrat se je znak F prepisal. [1]

Zgoraj navedeno abecedo lahko uporabljamo samo za risanje dvodimenzionalnih fraktalov. V ta namen razširimo abecedo z dodatnimi znaki za rotacijo. Pisalu priredimo lokalni kartezični koordinatni sistem **XYZ**, pri katerem os **Z** kaže naprej, os **Y** gor, os **X** pa na stran. Novo abecedo sestavljajo znaki:

$+$ - rotacija okoli osi **Y** v levo smeri

$-$ - rotacija okoli osi **Y** osi v desno smer

& - rotacija dol okoli osi **X**

$\wedge$ - rotacija gor okoli osi **X**

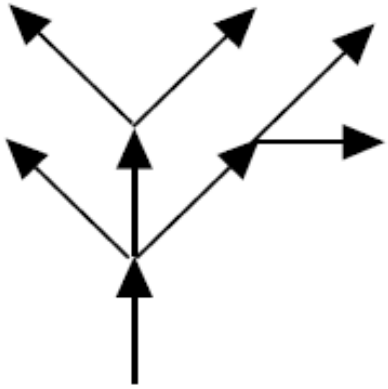
$\backslash$ - obrat levo okoli osi **Z**

/ - obrat desno okoli osi **Z**

$|$ - obrat nazaj

OL-sistemi

Da bomo z L-sistemi lahko generirali drevesa, bomo krivuljo morali vejiti. To rešujejo OL-sistemi. OL-sistemi vsebujejo dodatne znake za vejenje. Ti so označeni z oglatimi oklepaji. Z njimi si zapomnimo trenutno stanje pisala, torej njegov položaj in rotacijo. Znak [porine trenutno stanje na sklad, znak] pa ga iz sklada povleče. Torej, ko je v nizu znak [, si bo sistem zapomnil položaj pisala in nam tako omogočal risanje veje. Znak] označuje zaključek veje in vrne pisalo na njen izvor. To nam nato omogoča risanje drugih vej. Spodaj navedeni primer 2.4 prikazuje drevo, dobljeno iz niza $F[+F][-F[-F]F]F[+F][-F]$.



Slika 2.4: Primer uporabe znakov [in] za vejeneje. [1]

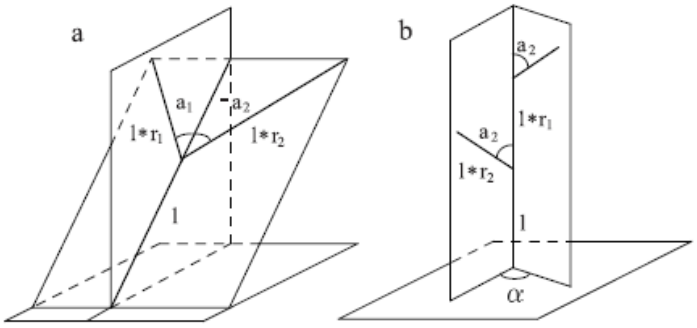
Nadaljnje razširitve

Obstajajo še nadaljnje razširitve L-sistemov. Do sedaj smo obravnavali deterministične L-sisteme, torej enak nabor pravil, in aksiom bi vedno izrisal enako drevo. V ta namen so uvedli stohastična pravila. Pravilom, ki si delijo enak znak, so dodali več izidov, vsakega s svojo verjetnostjo. Pri klicu pravila se proporcionalno verjetnosti izbere enega iz možnih izidov. Tako kljub enakemu naboru pravil lahko dobi različne oblike dreves.

Dodali so tudi pravila, ki imajo, glede na to iz katerega pravila izhajajo, drugačne izide. Takšna pravila se imenujejo pravila, občutljiva na kontekst (angl. context sensitive).
Da bi lahko isti nabor pravil lažje spreminjali, so dodali tudi parametrična pravila. Ta pravila jemljejo kot vhod določene parametre, ki vplivajo na njihov izid. Omogočajo nam, da z L-sistemi implementiramo določene parametrične modele za generiranje dreves.

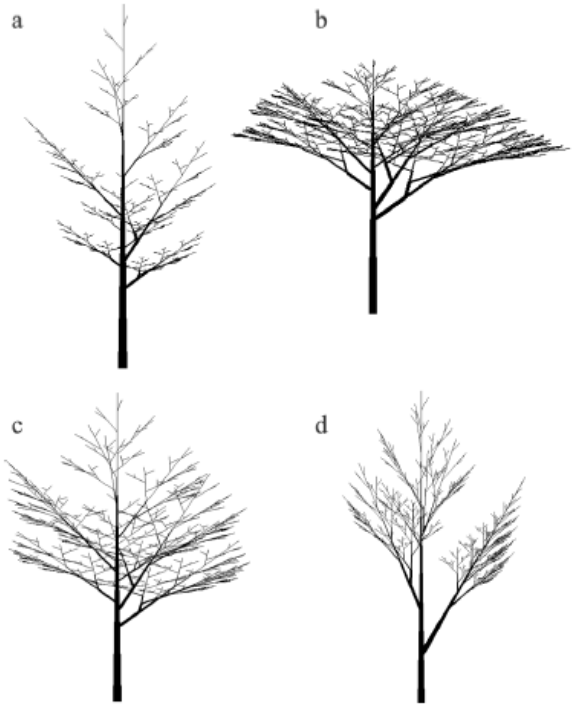
Hondin model generiranja struktur dreves

Hondin model je eden izmed prvih preprostih modelov, namenjenih generiranju dreves. Honda je za svoj model predpostavil [1]:
• Segmenti drevesa so ravni.
• Materni segment se veji na dva otroka sočasno.
• Dolžina otrok se krajša s konstantno hitrostjo $r1$ in $r2$ glede na materni segment.
• Materna in otroška segmenta ležita na enaki vejni ravnini. Otroka imata konstantna vejna kota $a1$ in $a2$ glede na materni segment.
• Vejna ravnina je fiksna glede na smer gravitacije tako, da je čim bližje horizontalni ravnini. Izjema so veje, ki se vejijo direktno iz debla drevesa. Te imajo konstanten kot α.



Slika 2.5: Značilnosti vej po Hondinem modelu. [1]

S spreminjanjem teh parametrov lahko dobimo velik nabor drevesu podobnih oblik. Osnovni Hondin model so z nadaljnjimi implementacijami še izboljšali. Ena izmed izboljšav so stohastična pravila in kontekstu občutljiva pravila.




```
n = 10
#define r1 0.9    /* contraction ratio for the trunk */
#define r2 0.6    /* contraction ratio for branches */
#define a0 45     /* branching angle from the trunk */
#define a2 45     /* branching angle for lateral axes */
#define d 137.5   /* divergence angle */
#define wr 0.707  /* width decrease rate */

ω  : A(1,10)
p1: A(l,w) : * → !(w)F(l)[&(a0)B(l*r2,w*wr)]/(d)A(l*r1,w*wr)
p2: B(l,w) : * → !(w)F(l)[-(a2)$C(l*r2,w*wr)]C(l*r1,w*wr)
p3: C(l,w) : * → !(w)F(l)[+(a2)$B(l*r2,w*wr)]B(l*r1,w*wr)
```

Slika 2.6: Primer dreves generiranih z hondinem modelu skupaj z pravili za njihovo generacijo. [1]

2.1.2 Generiranje dreves z kolonializacijo prostora

Pri tem pristopu generiramo drevo z zapolnjevanjem prostora. Torej za razliko od L-sistema pri tem modelu generiramo veje tako, da določimo, kje se bodo le-te končale. Za generiranje samih vej uporablja pristop celičnih avtomatov. Izdelava drevesa poteka po naslednjih korakih (slika 2.7):

a. Znotraj prej določenega volumna, naključno ali po določenem pravilu, generiramo nabor izčrpovalnih točk (angl. attrition points). Te točke bodo ob koncu algoritma predstavljale konice vej generiranega drevesa.

b/c. Z uporabo kolonializacije prostora iterativno generiramo stikajoče se celice, tako da z njimi dosežemo vse izčrpovalne točke ali pa je doseženo želeno število iteracij. Te celice predstavljajo skelet našega drevesa

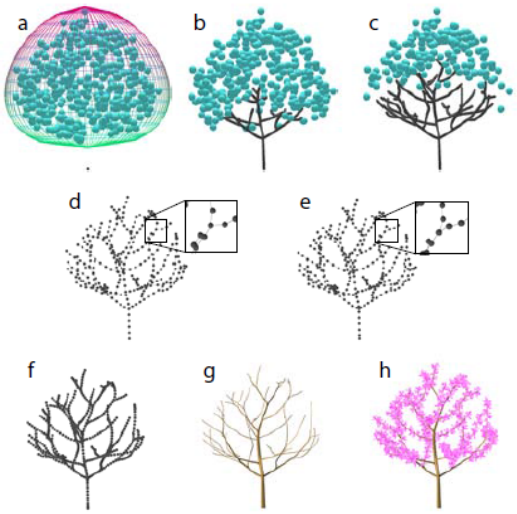
d. Odstranimo nepotrebne vmesne celice in tako dobimo ravne segmente.

e. Preostale celice premaknemo vzporedno proti osnovnemu segmentu, iz katerega so se vejili. S tem dosežemo manjše vejne kote in doprinesemo k videzu dreves.

f. Z uporabo razdelitvenega (angl. subdevision) algoritma dobljen skelet še dodatno zgladimo.

g. Skeletu izdelamo geometrijsko mrežo. Debelina vej se od konic do korenin širi.

h. Dobljenemu drevesu na konicah dodamo liste.



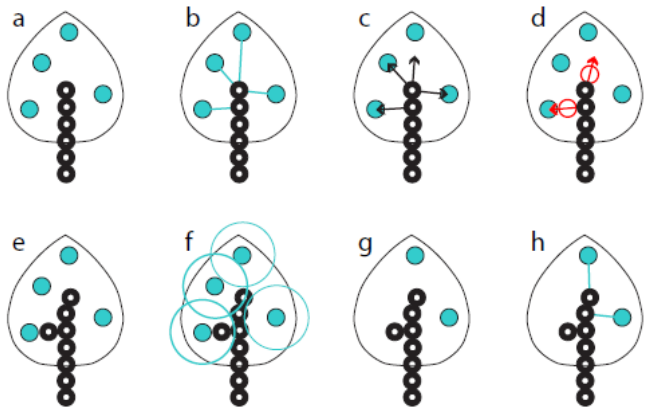
Slika 2.7: Postopek generiranja drevesa z kolonializacijo prostora. [2]

Obliko dreves lahko še dodatno polepšamo tako, da postopoma dodajamo nove izčrpovalne točke (slika 2.8).

Slika 2.8: Drevo generirano z dodajanjem dodatnih izčrpovalnih točk [2]

Kolonializacija prostora

Osnova delovanja modela je algoritem kolonializacije prostora. Algoritem, ilustriran na spodnji sliki 2.9, deluje po naslednjem postopku:
a. Imamo nabor izčrpovalnih točk ter nekaj že prej generiranih celic.
b. Za vsako izčrpovalno točko najdemo njej najbližjo celico.
c. Določimo normirane vektorje od celice do izčrpovalnih točk.
d. Določimo smer nove celice kot normirano vsoto vektorjev.
e. Naredimo nove celice.
f. Preverimo, ali so nove celice v določeni bližini izčrpovalnih točk.
g. Če so, jih odstranimo.
h. Algoritem ponavljamo, dokler ne odstranimo vseh izčrpovalnih točk.



Slika 2.9: Delovanje algoritma kolonializacije prostora [2]

2.1.3 Parametrični modeli

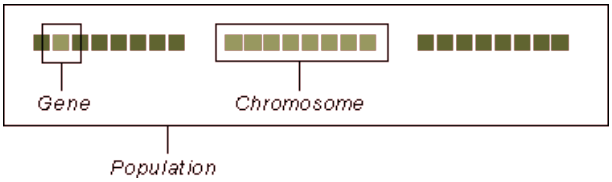
Parametrični modeli uporabljajo nabor parametrov za določanje oblike drevesa. Nekatere takšne modele je možno implementirati z uporabo L-sistemov. Primer takšnega je Hondin model. Zaradi fraktalne narave L-sistemi niso zmožni generirati dovolj različnih načinov vejenja dreves in se zato z njimi generirana drevesa omejijo na majhen nabor osnovnih dreves. Težava je tudi v veliki zahtevnostjo uporabe L-sistemov, saj zahtevajo natančno znanje nji-

hove gramatike ter problema, ki ga želimo z njimi rešiti. Parametrični modeli omogočajo izdelavo velikega nabora različnih tipov dreves brez dejanskega razumevanja načina vejenja dreves.
Obstaja veliko primerov parametričnih modelov namenjenih generiranju dreves. V sklopu diplomskega dela smo predelali model Jasona Weberja in Jospeha Penna[4] ter model Jorgena Nystada[3]. Na koncu smo se odločili za že izdelano implementacijo procTree, saj uporablja podobne principe kot prej navedena modela. Knjižnica procTree je bolje opisana v poglavju 3. Skozi raziskavo modelov smo opazili, da se novejši modeli ne osredotočijo tako na generiranje oblike drevesa, kot se osredotočajo na generiranje geometrijske mreže dreves. Na primer, Nystadov model izdela geometrijsko mrežo, ki je posebej prilagojena teselaciji.

2.2 Genetski algoritmi

Genetski algoritmi, v nadaljevanju označeni s kratico GA, so prilagodljivi hevristični iskalni algoritmi. Spadajo v skupino po naravi navdihnjenih algoritmov, imenovanih evolucijski algoritmi. Delujejo na osnovi evolucijskih idej naravnega izbora in genetike. Uporablja se jih za reševanje optimizacijskih problemov s pomočjo psevdonaključnega preiskovanja prostora. Kljub svoji naključni osnovi, algoritem ne preiskuje prostora naključno, saj uporablja mero uspešnosti prejšnjih generacij za vodenje nadaljnjega preiskovanja prostora. GA so bili zasnovani tako, da simulirajo naravno okolje, v katerem bi nato potekala evolucija glede na osnove, ki jih je prvič predstavil Charles Darwin v delu *O nastanku vrst z delovanjem naravnega izbora ali ohranjanje prednostnih vrst v boju za preživetje*. Algoritmi se predvsem naslanjajo na princip medsebojnega tekmovanja vrst in iz tega sledeče nadvlade ene nad drugo. GA uporabljajo osnove naravnega izbora nad posamezniki zaporednih generacij populacije z namenom, da dosežejo najboljše možne posameznike. Nabor vseh posameznikov predstavlja populacijo, katere velikost se skozi potek GA ne spreminja. Vsak posameznik je predstavljen s kromosomom, ki

je lahko zaporedje števil ali bitov, niz znakov, permutacija, binarno drevo in podobno. Kromosomi posameznikov predstavljajo točko v iskalnem prostoru našega problema in hkrati tudi njegovo rešitev. Kromosom je nato sestavljen iz manjših delov, imenovanih geni 2.10, torej v primeru kromosoma, ki je zaporedje števil, bi bili geni posamezna števila znotraj zaporedja. Vsak posameznik, torej njegov kromosom, predstavlja točko v iskalnem prostoru in hkrati tudi rešitev našega optimizacijskega problema.



Slika 2.10: Grafična predstavitev populacije, kromosoma in gena [10]

Uspešnost teh rešitev nato izmerimo s funkcijo uspešnosti, v nadaljevanju označene z kratico FF (angl. Fitness Function), ki kot parameter sprejme posameznika, oziroma njegov kromosom, in nam pove, kako dobro ta posameznik rešuje naš problem. Funkcija v kontekstu evolucije predstavlja posameznikovo zmožnost tekmovati z ostalimi posamezniki populacije. Da bodo GA delali pravilno, predpostavimo, da bodo posamezniki z visoko vrednostjo FF proizvedli posameznike, katerih vrednost FF bo večja od staršev. Pri tem upoštevamo, da se bodo posamezniki z visokimi vrednostmi FF tudi večkrat razmnoževali kot tisti z manjšimi vrednostmi FF in s tem izboljšali prihodnje generacije. Cilj GA je torej najti posameznika z največjo možno vrednostjo FF. To doseže z izmenjavanjem genov posameznikov glede na vrednost FF.

2.2.1 Algoritem

GA skozi potek delovanja ne spreminja velikosti populacije. Za vsakega posameznika znotraj te populacije hranimo njihov kromosom in vrednost FF. Prvo generacijo inicializiramo naključno, torej vsak posameznik dobi naključno generiran kromosom. Posameznikom nato izmerimo vrednost FF.

Glede na vrednost FF nato posameznike razmnožimo. Posamezniki z visokimi vrednostmi FF dobijo več priložnosti za razmnoževanje kot posamezniki z nizkimi vrednostmi FF. Kromosome staršev križamo in s tem dobimo kromosome potomcev. Potomci, ki nastanejo kot rezultat križanja, nosijo karakteristike staršev (mešanico genov staršev). Ker pri razmnoževanju dobimo nove posameznike, moramo velikost populacije prilagoditi prejšnji velikosti. To najlažje naredimo tako, da se znebimo najslabših posameznikov (najnižje vrednosti FF). S tem omogočamo, da se zaporedne generacije izboljšujejo. Da se populacija ne bi ujela v lokalni maksimum FF, posameznike tudi mutiramo. Mutiramo jih tako, da jim naključno spremenimo neke gene. Ta postopek nato ponavljamo za neko fiksno število generacij ali pa dokler ne dobimo posameznika, ki je dovolj dober.
Delovanje algoritma lahko opišemo z naslednjo psevdo kodo:

```
1. Nakljucno inicializiraj populacijo(t)
2. Doloci vrednosti funkcije uspesnosti populacije(t)
3. Ponavljaj
    1. Izberi starse, za krizanje, iz populacije(t)
    2. Z izbranimi starsi izvedi krizanje in s tem izdelaj populaci
    3. Populacijo(t+1) mutiramo
    4. Doloci vrednosti funkcije uspesnosti populacije(t+1)
4. dokler najboljsi posameznik ni dovolj dober
```

2.2.2 Inicializacija

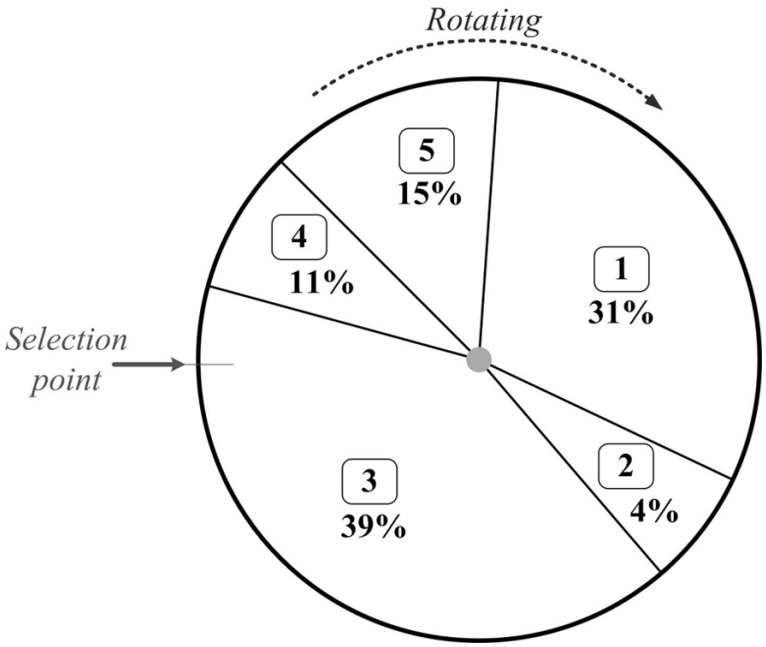
Kot smo omenili prej, populacijo inicializiramo naključno, ponavadi omejeno na določeno območje. Pri tem prilagodimo velikost populacije kompleksnosti iskalne domene.

2.2.3 Izbor staršev

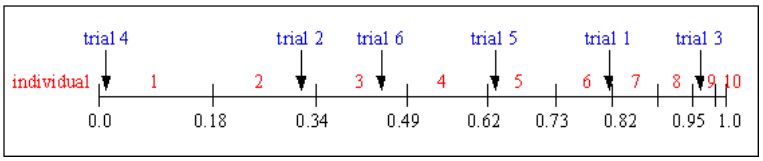
Starše za razmnoževanje se izbere na podlagi vrednosti FF. Posamezniki z visokimi vrednostmi bodo imeli torej večjo verjetnost biti izbrani za starše. Najbolj enostaven način je, da posameznike uredimo glede na vrednost FF in izberemo N najboljših. Parameter N je število staršev. Težava pri tem je, da število izbranih staršev ni proporcionalno glede na vrednost FF, ampak samo glede na zaporedno mesto v populaciji. Metodi, ki to upoštevata, sta izbor z roletnim kolesom in univerzalno stohastično vzorčenje.

Izbor z roletnim kolesom

Izbor z roletnim kolesom, v nadaljevanju označen z kratico RWS (roulette-wheel selection), znan tudi kot izbor, proporcionalen glede na vrednost uspešnosti (fitness proportionate selection), je metoda izbiranja staršev, ki izbira posameznike proporcionalno glede na vrednost FF. Starše izbira z naključnim vzorčenjem prostora, ki vsebuje normiran nabor vrednosti FF posameznikov populacije. Ta prostor dobimo tako, da vrednosti FF uredimo po velikosti in jih normiramo tako, da je seštevek vseh enak 1. Nato na intervalu od 0 do 1 izberemo naključno število. Starša izberemo glede na to, v kateri interval to število nato pade 2.11. To nato ponavljamo, dokler ne dobimo želenega števila staršev 2.12.



Slika 2.11: Primer vzorca RWS. [5]



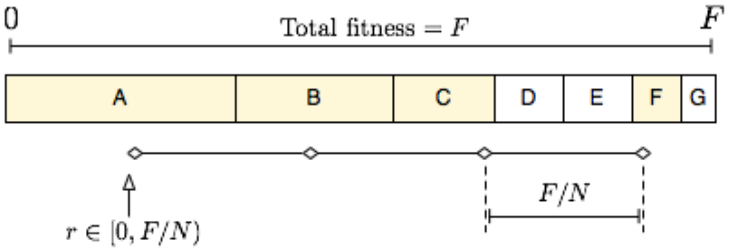
Slika 2.12: Primer celotnega izbora RWS. [6]

Algoritem izbira starše proporcionalno glede na vrednosti FF. Težava se pojavi, če en posameznik zavzema večji del celotnega intervala populacije, saj se lahko zgodi, da bodo ostali posamezniki zelo redko izbrani.

Univerzalno stohastično vzorčenje

Univerzalno stohastično vzorčenje, v nadaljevanju označeno z kratico SUS (angl. Stochastic Universal Sampling), želi popraviti težave, ki so se pojavile pri algoritmu RWS. Vzorčni prostor pripravi na enak način kot pri RWS.

Težavo RWS reši z enakomernim vzorčenjem prostora od naključne točke naprej. To da možnost tudi šibkejšim posameznikom, da so izbrani za razmnoževanje. Algoritem najprej izbere naključno točko, na intervalu od 0 do 1/N, kjer bo začel vzorčenje. Ta točka predstavlja prvi vzorec. Nato naredi še N-1 vzorcev, vsak zamaknjen za 1/N od prejšnjega. Starše nato izberemo glede na to, v kateri interval tej vzorci padejo 2.13.



Slika 2.13: Primer SUS. [7]

Elitizem

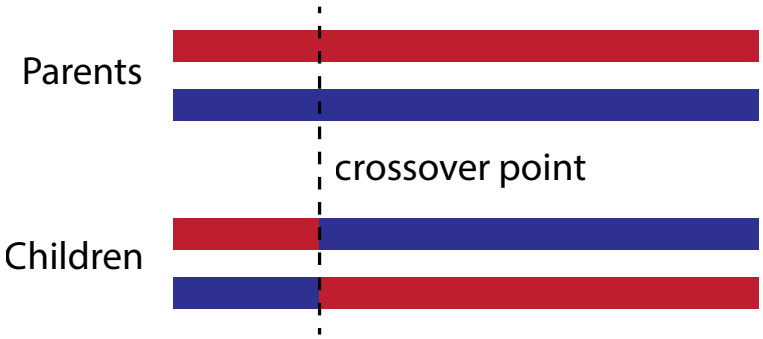
Elitizem (v kontekstu GA) pomeni, da vsaki zaporedni generaciji dodamo nekaj najboljših posameznikov prejšnje generacije. Elitizem izboljša iskanje v primerih, ko zaporedne generacije ne pomenijo vedno boljših posameznikov. Iskanje se izboljša, saj so najboljši kromosomi vedno prisotni. Tako se lahko njihovi geni vedno križajo. Elitizem nudi pospešitev iskanja. Kritike elitizma so predvsem v tem, da ni v skladu s principi naravne selekcije, katere GA simulirajo.

2.2.4 Genetski operatorji

Ko imamo izbrane vse starše, moramo njihove kromosome izmenjati. To naredimo z uporabo genetskih operatorjev: Križanja in mutacije.

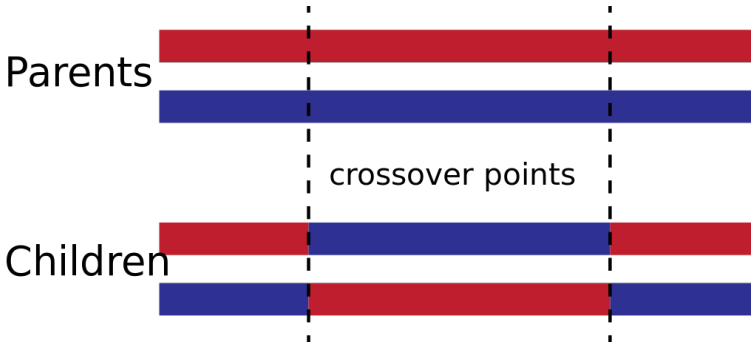
Križanje

Križanje je postopek, ki združi kromosome dveh ali več staršev v kromosom otroka. Križanje mora biti prilagojeno glede na tip kromosoma. Pri kromosomu v obliki permutacije moramo torej paziti, da križanje ne bo vstavilo števil, ki so že v kromosomu. Pri kromosomu v obliki drevesa naredimo križanje tako, da izmenjamo veje dreves in tako naprej. V nadaljevanju bo opisanih nekaj metod za križanje kromosomov v obliki vektorja števil, saj tej predstavljajo najbolj pogosti tipi kromosomov.
Najbolj enostaven način križanja je križanje po eni točki 2.14. Najprej določimo eno točko, ki razdeljuje kromosom na dva dela. Pri tem načinu križanja dobimo dva otroka. Prvi otrok dobi levi del kromosoma prvega starša in desni del drugega starša, drugi pa obratno.



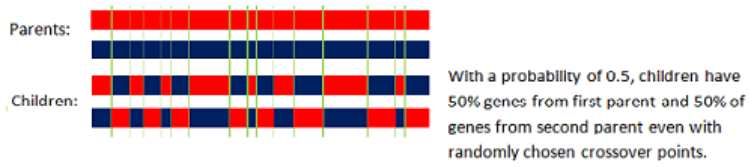
Slika 2.14: Primer križanja po eni točki. [9]

Najlažja izboljšava tega je, da dodamo več točk in za vsako točko izmenjujemo, čigave gene otrok prejme. Ti algoritmi se poimenujejo glede na število točk, ki jih uporabimo. Torej, če imamo dve točki, se križanje imenovalo križanje po dveh točkah 2.15.



Slika 2.15: Primer križanja po dveh točkah. [9]

Težava pri takem načinu križanja je, da se geni izmenjujejo vedno na isti način. To težavo rešuje z enakomernim križanjem 2.16. Pri enakomernem križanju uporabljamo fiksno mešalno verjetnost, ki nam pove ali se bo gen izmenjal. Na intervalu od 0 do 1 za vsak gen generiramo naključno število. Če je le-to večje od mešalne verjetnosti, se gena izmenjata, če je manjše se ne. Torej se bo z mešalno verjetnostjo 0.5 izmenjalo okoli 50% genov .



Slika 2.16: Primer enakomernega križanja. [9]

Za križanje lahko uporabimo tudi več staršev in s tem izboljšamo tudi kakovost dobljenega kromosoma. To najlažje naredimo tako, da za vsak gen še naključno izberemo starše, katerih gene bomo izmenjali.

Mutacije

Genetski algoritmi lahko ostanejo v lokalnem maksimumu, čemu se lahko izognemo z mutacijo. Po križanju je dobro dobljene otroke še mutirati. To pomeni, da naključno spremenimo določene gene v kromosomu. To lahko

naredimo fiksno, tako da se isti geni vedno spreminjajo, ali pa naključno, z naključnim izbiranjem genov za mutacijo. Ker se gen naključno spremeni, lahko dobimo posameznika, ki se ne nahaja v enaki okolici iskalne domene kot ostali posamezniki populacije. Z nadaljnjim izvajanjem GA se lahko izkaže, da se ti posamezniki nahajajo bližje globalnemu maksimumu kot kromosomi pred mutacijo.

2.2.5 Prekinitev

Kako vedeti kdaj algoritem prekiniti? Najbolj osnovna metoda prekinitve je, da algoritem prekinemo po nekem fiksnem številu generacij. Pri preprostejših problemih algoritem hitro konvergira proti neki rešitvi, pri zahtevnejših pa ne. Zato je dobro slediti spreminjanju najboljšega kromosoma. Če se ta po nekaj generacijah ne spremeni, lahko algoritem prekinemo. V nekaterih primerih želimo dobiti le dovolj dobro rešitev, kar naredimo s pragom najboljšega posameznika. Uporabnik nastavi prag vrednosti FF, in če najboljši posameznik preseže ta prag, potem lahko algoritem prekinemo. Za GA lahko predpostavimo, da se bliža rešitvi, ko je razlika kromosomov dovolj majhna, zato lahko algoritem prekinimo tudi, če je kumulativna razlika kromosomov dovolj majhna. Pri tem je dobro razliko vsakega kromosoma obtežiti z normirano vrednostjo FF.

2.2.6 Prednosti in slabost

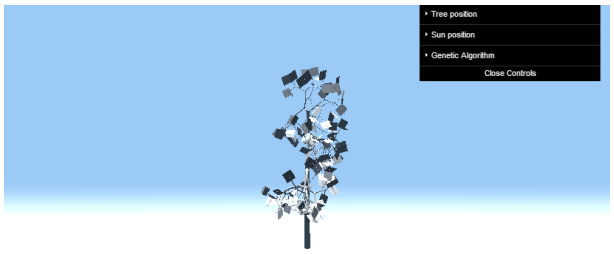
Prednosti GA so, da so bolj robustni in manj prostorsko zahtevni kot drugi podobni iskalni algoritmi (na primer linearno programiranje, iskanje v globino, iskanje v širino ...). Nudijo tudi prednosti pri optimizaciji visoko prostorskih, večmodelnih in visoko dimenzionalnih problemov, saj prostor ne preiskujejo izčrpno. Težava GA je, da se ujamejo v lokalne maksimume. Za reševanje visoko dimenzionalnih večmodelnih problemov potrebujemo enako kompleksno FF. Te funkcije so ponavadi zelo časovno zahtevne, zato za zaporedno računanje le-teh porabimo veliko časa. Težava je tudi, da se s kom-

pleksnostjo problema veča tudi iskalna domena in to zahteva tudi večje populacije, ki pa zahtevajo temu primerno večjo število izračunov FF. Prav tako jih ne moremo uporabljati na dinamičnih podatkih, saj bi se s podatki spreminjala tudi iskalna domena in s tem tudi maksimumi FF.

Poglavje 3

Implementacija

Na podlagi zgoraj napisane teorije smo izdelali aplikacijo, ki generira drevesa, prilagojena izvoru svetlobe. Aplikacija je spletna, izdelana v programskem jeziku JavaScript. Prikaz je narejen z uporabo knjižnice Three.js, ki omogoča lažjo uporabo vmesnika WebGL. Za generiranje dreves uporabljamo knjižnico ProcTree, ki uporablja parametrični model za generiranje dreves. Genetski algoritmi so bili izdelani po lastni implementaciji. Z uporabo spletnih delavcev HTML5 (angl. HTML5 web worker) smo implementirali vzporedno izvajanje izrisovanja scene in GA. Uporabniški vmesnik je izdelan z uporabo knjižnice dat.gui.js, ki omogoča neposredno manipulacijo spremenljivk ter neposreden klic funkcij.



Slika 3.1: Videz aplikacije

Aplikacije je dostopna na spletnem portalu gitHub: `https://github.com/jus390/Genetic-Trees`

3.1 Uporaba aplikacije

Aplikacija deluje na vseh brskalnikih, razen Internet Explorerju, saj ne podpira spletnih delavcev HTML5. Testirali smo jo na brskalnikih Google Chrome in Mozilla Firefox. Aplikacija ima prikazovalnik, ki se razteza čez celotno spletno stran. Prikaz se prilagaja velikosti okna. Z levim klikom na prikazovalnik lahko kamero vrtimo okoli središča scene. Z miškinim koleščkom in srednjim klikom se lahko približujemo/oddaljujemo od središča scene. V zgornjem desnem kotu se nahaja uporabniški vmesnik, ki uporabniku omogoča spreminjanje položaja sonca in drevesa ter parametrov genetskega algoritma. Ko nastavimo vse parametre, lahko zaženemo optimizacijo z gumbom z oznako *Start*, v razdelku *GeneticAlgorithm*. Z bližnjico h lahko skrijemo/razkrijemo uporabniški vmesnik, z bližnjico r pa lahko kadarkoli generiramo naključno drevo. Za boljše sledenje programu lahko s tipko $F12$ odpremo konzolo, kjer se bo prikazovala natančnejša sled programa. Pri tem moramo paziti, da jo odpremo preden začnemo izvajati optimizacijo.

3.2 Generiranje dreves

Drevesa se generirajo z uporabo modificirane knjižnice ProcTree. Avtor knjižnice je uporabnik supereggbert s spletnega repositorija GitHub[12]. Model, katerega knjižnica uporablja, je parametričen. Uporablja naslednje prametre:

• **seed**: Izvorno seme naključnega generatorja, ki vpliva za naključno spreminjanje dreves z enakim naborom parametrov.

• **levels**: Stopnja rekurzivnega deljenja vej. Pove nam, kolikokrat se bo drevo vejilo.

• **vMultiplier**: Koeficient, s katerim se pomnoži teksturna koordinata **V**.

Vpliva na velikost teksture na drevesu.

• **twigScale**: Velikost listov.

• **initalBranchLength**: Osnovna velikost vej. Velikost vej generiranih v prvem koraku vejenja, ki ne pripadajo deblu.

• **lengthFalloffFactor**: Faktor, po katerem se veje krajšajo z nadaljnjim vejenjem.

• **lengthFalloffPower**: Eksponent, s katerim se veje krajšajo z nadaljnjim vejenjem.

• **clumpMax/clumpMin**: Parametra določata interval, ki pove, kako skupaj bodo veje na konicah. Torej pri visoki vrednosti *clump* bodo veje na konicah zelo blizu, pri nizkih vrednosti pa narazen.

• **branchFactor**: Faktor, ki vpliva na to, kako zelo veje pri vejenju upoštevajo smer osnovne veje.

• **dropAmount**: Faktor, ki vpliva na spust veja. Večja kot je stopnja vejenja, večji ima vpliv. Torej na veje, ki so bližje deblu po rasti ima manjši vpliv.

• **growAmount**: Podobno kot *dropAmount*, le da ta vpliva na dvig vej z zaporedno generacijo. *GrowAmount* je za razliko od *dropAmount* normaliziran glede na parameter *levels*.

• **sweepAmount**: Parameter nagne veje v smeri osi **X**, glede na trenutno stopnjo vejenja.

• **sweepAmount2**: Enako kot sweepAmount, le v smeri osi **Z**.

• **maxRadius**: Radij debla.

• **climbRate**: Parameter vpliva na vertikalno rast debla. Pove nam, koliko so dolgi vsi segmenti debla, razen prvega.

• **trunkKink**: Parameter zamakne konce segmentov debla v pozitivni in negativni smeri osi **X** izmenično, glede na stopnjo vejenja. Torej, če bo v eni stopnji zamaknilo v pozitivno smer, potem bo po naslednjem vejenju zamaknilo v negativno smer.

• **treeSteps**: Pove, kolikokrat se veji deblo.

• **taperRate**: Parameter krajša osnovno dolžino vej relativno na stopnjo

vejenja debla. Torej, večji kot je parameter, bolj bo drevo ozko pri vrhu.
• **radiusFalloffRate**: Parameter vpliva na manjšanje radija vej z zaporednimi vejenjem.
• **twistRate**: Parameter vpliva na razporeditev vej z debla. Če je parameter nič, bodo vse veje na isti strani debla. Večji kot je parameter, bolj se zaporedne veje vrtijo okoli debla. Veje se torej generirajo v obliki vijačnice okoli debla.
• **trunkLength**: Velikost osnovnega debla pred vejenjem.

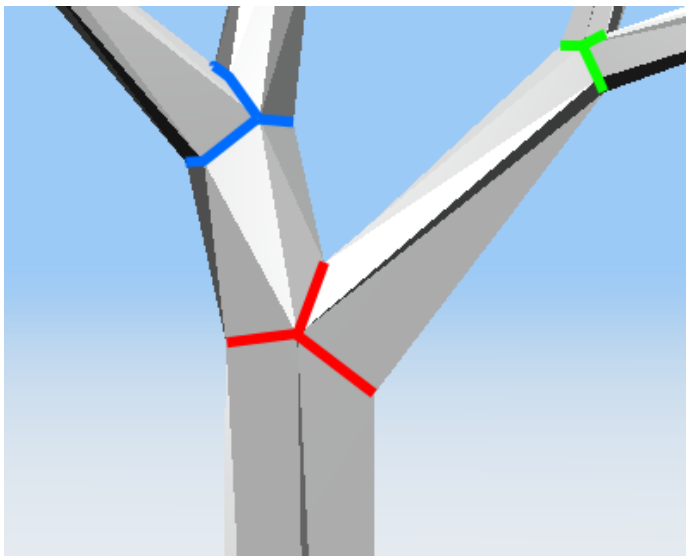
Knjižnica definira objekt tipa *Tree*, ki vsebuje koordinate točk, teksturni koordinati **UV**, kompozicijo poligonov (katere točke sestavljajo poligone) in normale poligonov. Generiranje poteka po naslednjem postopku:
1. Generiranje oblike drevesa.
2. Generiranje točk geometrijske mreže vej in dela.
3. Generiranje listov
4. Izdelava poligonov debla in vej iz prej generiranih točk.
Knjižnica je bila izdelana za uporabo knjižice GLGE, zato je bilo potrebno prilagoditi formate točk v format, ki ga uporablja knjižnica three.js.

3.2.1 Generiranje oblike

Oblika drevesa je vsebovana v objektu tipa *Branch*. Vsak objekt vsebuje podatke enega segmenta debla/vej. Nosi podatke o izvoru segmenta, smeri rasti, dolžini in kazalce na materinski segment ter oba otroška segmenta.
Generiranje oblike poteka znotraj funkcije *split*. Funkcija izdela simetrično binarno drevo med seboj povezanih objektov *Branch*. Funkcija prvo generira obliko materinskega segmenta. Nato, če stopnja rekurzije ni presegla števila dovoljenih vejenj, se rekurzivno kliče na otroških segmentih. Znotraj delovanja se glede na parametre drevesa nastavijo parametri trenutnega segmenta in lokacijo ter smer otrokov. Proces se nato ponavlja, dokler ne doseže želenega števila vejenj.

3.2.2 Generiranje geometrijske mreže

Izdelava geometrijske mreže poteka v dveh funkcijah *createForks* in *doFaces*. Funkcija *createForks* izdela točke geometrije, funkcija *doFaces* pa te točke poveže v poligone. Funkcija *createForks* najprej naredi osnovni obroč točk okoli izvora debla, pri tem upošteva parameter *segments*, ki nam pove, koliko točk bo obroč vseboval. Geometrijo vejitev naredi v obliki treh, na koncu spojenih pol-obročev (glej sliko 3.2). Torej, glede na smer starša in smer otrok naredi polovice obročev, ki si delijo dve točki na robovih. Ti trije pol-obroči nato sestavijo tri ločene obroče. Vsak obroč predstavlja izvor oziroma, v primeru starševske veje, konec segmenta. Da pri generiranju poligonov ne bo prišlo do torzije znotraj topologije segmenta, so indeksi teh točk urejeni glede na hčerinske segmente. Funkcija z vsakim zaporednim vejenjem zmanjša radij vej glede na parameter $radiusFalloffRate$. Lokacije točk se nato shranijo v tabelo.



Slika 3.2: Oblika geometrije vejenj.

Izdelane točke se nato povežejo v poligone. Točke zaporednih obročev se povežejo tako, da se povežejo točke z istim indeksom v tabeli točk obročev.

Nato se točke prvega obroča povežejo s točkami drugega obroča, tako da se indeks točk drugega obroča zamakne za 1. S tem dobimo trikotnike. Med generiranjem nastavlja tudi teksturne koordinate posameznih poligonov. Indeksi povezanih točk se nato shranijo v tabelo.
Na koncu se generirajo še listi. Listi se generirajo na zadnjih segmentih drevesa. Vsak list vsebuje dvojne poligone, vsak z normalo obrnjeno v nasprotno smer, tako da so vidni na obeh straneh. Teksturne koordinate se na drugi strani prezrcalijo.

3.3 Genetski algoritmi

Postopek GA v aplikaciji predstavlja funkcija *GAOptimize* in njena razširitev *GAOptimize*2. Funkcija ima naslednje parametre:
• **treeLoc**: Lokacija drevesa, katerega želimo optimizirati.
• **maxIter**: Maksimalno število generacij.
• **numOfInstances**: Velikost populacije.
• **numOfChild**: Število otrok.
• **sameBestIter**: Število generacij, po katerem se funkcija prekine, če se najboljši posameznik ni spremenil.
• **useBest**: Stikalo, ki nam pove, če se najboljši posameznik prejšnje generacije prenese v naslednjo.
• **minFitnessTerm**: Če uspešnost najboljšega posameznika preseže $minFitnessTerm$, se funkcija prekine.
• **mutationRate**: Delež populacije, ki bo po križanju mutirala.
• **useFacing**: Ali pri oceni FF upoštevamo smer listov.
• **useTrunk**: Ali pri oceni FF upoštevamo poligone debla in vej.

Razširitev GAOptimize2 je namenjena optimizaciji drevesa, pri tem pa upošteva še drugo drevo. Ta funkcija ima še dodatne parametre:
• **tree2**: Objekt tipa $Tree$ drugega drevesa.
• **tree2Loc**: Lokacija drugega drevesa.

Obe funkciji vrneta objekt tipa *Tree*, ki predstavlja najboljšega posameznika.

3.3.1 Kromosom

Kromosom drevesa je v vsakem drevesu shranjen v objektu *gene*. Vsebuje nabor parametrov drevesa. Objekt *gene* je potreben, saj so parametri znotraj knjižnice shranjeni statično, kar pomeni, da si iste parametre delijo vsi posamezniki tipa *Tree*. Kromosom kljub temu ne vsebuje vseh parametrov dreves, ampak samo tiste, ki vplivajo na rast drevesa. Ti so (v oklepaju so navedeni intervali teh parametrov): **seed** (0-4000), **initialBranchLength** (0.5-0.8), **lengthFalloffFactor** (0.5-0.8), **lengthFalloffPower** (0.3-0.7), **clumpMax** (0.4-0.5), **clumpMin** (clumpMax-(0.0-0.4)), **branchFactor**(2.0-4.0), **dropAmount** (-0.3-0.3), **growAmount** (-0.5-1.0), **sweepAmmout** (-0.15-0.15), **sweepAmmout2** (-0.15-0.15), **climbRate** (0.05-1.0), **trunkKink**(0.0-0.3), **taperRate** (0.7-1.0), **radiusFalloffRate** (0.74-0.79), **twistRate**(0.0-10.0), **trunkLength** (1.5-2.1).
Ostali parametri imajo pa fiksne vrednosti: **segnemts** (6), **levels** (5), **vMultiplier** (1.16), **twigScale**(0.22), **maxRadius** (0.111), **treeSteps** (4).
Naključno drevo se generira tako, da se tej parametri inicializirajo na navedenih intevalih, iz teh parametrov se potem inicializira objekt tipa *Tree*, kateremu se nato doda objekt *gene*.

3.3.2 Funkcija uspešnosti

Najpomembnejša komponenta GA je funkcija uspešnosti. V našem primeru želimo z njo oblikovati drevo tako, da bodo veje prilagojene kotu svetlobe. Za FF smo predpostavili, da bo drevo prilagojeno, če bo čim več listov neposredno izpostavljeno svetlobi. Torej listov ne smejo zakrivati drugi objekti. Zato smo definirali metodo, ki prešteje, koliko poligonov listov je neposredno izpostavljenih žarkom sonca. Določanje, ali je poligon neposredno izpostavljen soncu, bomo izvedli s preverjanjem, ali kakšen drugi poligon preseka žarek, preden ta pride do lista. Presečišče s poligonom določimo tako, da

najprej najdemo presečišče žarka z ravnino, na kateri ta poligon leži, nato pa še preverimo, ali se to presečišče nahaja znotraj poligona. Žarek je definiran z enačbo:

$$P = P_0 + tV \tag{3.1}$$

Pri tem je P točka na žarku, P_0 je izvor žarka, V je smer žarka v obliki vektorja, t pa faktor, ki nam pove kako daleč od izvora v smeri V leži točka. Enačba ravnine je:

$$P * N + d = 0 \tag{3.2}$$

Pri tem je P točka na ravnini, N je normala ravnine, d nam pa pove, kje v smeri normale se nahaja ravnina. Torej, če poznamo eno točko na ravnini ter normalo ravnine, velja $-d = P * N$. Če želimo dobiti presečišče žarka z ravnino, mora veljati, da točka leži na žarku in na ravnini. Zato enačimo spremenljivko P žarka s spremenljivko P ravnine. Dobimo naslednjo enačbo:

$$(P_0 + tV) * N + d = 0 \tag{3.3}$$

Iz te funkcije izpostavimo t:

$$t = -(P_0 * N + d)/(V * N) \tag{3.4}$$

Dobljeni t nato vstavimo v enačbo žarka(3.1) in dobimo presečišče žarka z ravnino. Da izvemo, ali neka ravnina leži med poligonom lista, ki ga preverjamo, in soncem, izračunamo spremenljivko t lista ter spremenljivko t poligonove ravnine. Če je t ravnine manjši od t lista, potem presečišče leži pred našim listom. Sedaj moramo preveriti, ali to presečišče dejansko leži na poligonu. Najprej lahko preverimo, če se presečišče nahaja dovolj blizu središča poligona. To naredimo tako, da izračunamo evklidsko razdaljo med središčem poligona in presečiščem. Če je razdalja večja od razdalje do najbolj oddaljenega oglišča, lahko presečišče zavržemo. Če se pa nahaja dovolj blizu, potem preverimo še, ali leži znotraj poligona. To preverimo tako, da izračunamo vektorje od presečišča do vseh oglišč, nato izračunamo ostre kote med njimi. Če se koti seštejejo v 360 stopinj, se presečišče nahaja v poligonu. Funkcijo uspešnosti v aplikaciji predstavlja funkcija $treeLightFitness$ in

njeno razširitev *treesLightFitness*. Funkcija kot parameter sprejme objekt tipa *Tree*, lokacijo dreves, ter dve stikali, ki nam povesta, ali pri izračunu prekrivanja upoštevamo tudi poligone debla ter vej (*useTrunk*) in ali vpliva usmerjenost listov na rezultat funkcije (*useFacing*). V inicializaciji predpostavi, da so vsi poligoni nezakriti in jim tako dodeli vrednost 1 ali, če je označeno stikalo *useFacing*, absolutno vrednost skalarnega produkta smeri žarka in normale poligona. Ti žarki izvirajo iz lokacije sonca usmerjeni so proti proti sredini poligona lista, ki ga preverjamo. Nato za vsak poligon listov preveri, ali ga zakriva kateri drug poligon. To naredi tako, da gre za vsak poligon lista čez vse ostale poligone liste, in če je označeno stikalo *useTrunk*, tudi čez vse poligone debla. Če en poligon zakriva list ali če je njegovo središče nahaja pod koordinato 0, se mu dodeli vrednost uspešnosti 0 in zanka, ki gre čez ostale poligone listov, se prekine. Če je označeno stikalo *useTrunk* in vrednost uspešnosti tega lista ni enaka nič, potem funkcija izvede še drugo zanko, ki gre čez poligone debla in vej. Vrednosti uspešnosti se med delovanjem shranijo v tabelo. Na koncu funkcija sešteje vse vrednosti tabele in to vrne kot uspešnost drevesa.
Razširitev *treesLightFitness*, vsebuje še parametre za dodatno drevo. Torej objekt *Tree* drugega drevesa, ter lokacijo drugega drevesa. Pri optimizaciji pa kot oviro prvemu drevesu upošteva tudi geometrijo drugega. Torej so dodane zanke za poligone listov ter debela drugega drevesa.

3.3.3 Izbor

Za izbor staršev aplikacija uporablja univerzalno stohastično vzorčenje. Izbor je implementiran v obliki funkcije, imenovane *stohasticSelection*. Funkcija kot parameter sprejme tabelo vrednosti FF ter število staršev, ki mora biti sodo število. Algoritem deluje po zgoraj opisanem postopku (poglavje 2.2.3), vrača pa tabelo indeksov posameznikov izbranih za razmnoževanje. Katera starša se bosta razmnožila izberemo tako, da izberemo prvega posameznika v tabeli ter naslednjega posameznika, ki nima enakega indeksa v tabeli staršev ali enake vrednosti FF. Ta posameznika nato križamo, njuna indeksa pa

odstranimo iz tabele. To nato ponavljamo, dokler tabela indeksov ni prazna.

3.3.4 Križanje

Implementirali smo dve različni metodi križanja. Prva je standardno univerzalno križanje (glej poglavje 2.2.4). Torej, za vsak gen generiramo naključno število ter, če to preseže mešalno verjetnost se ta gen izmenja. Ta način križanje je implementirano kot funkcija *crossOver*, ki kot parameter sprejme starševski drevesi, vrne pa dva otroka. Druga metoda križanja križa starše tako, da vzame povprečje vrednosti genov obeh staršev. Tudi ta funkcija kot parameter sprejme oba starša, vendar za razliko od prve funkcije pa ta vrne le enega otroka.
Testirali smo obe funkciji in obe najdeta ustrezni drevesi. Katera metoda se izkaže kot boljša oziroma hitrejše najde rešitev, nismo mogli določiti. Aplikacija zato uporablja prvo metodo, ki je genetsko bolj pravilna.

3.3.5 Mutacija

Funkcija *GAOptimize* ima parameter *mutationRate*. Parameter nam pove, kolikšen delež otrok bo mutiral. Za vsakega novega otroka se generira naključno realno število na intervalu med 0 in 1, ki nam pove, ali bo otrok mutiral. Če je to število manjše od parametra *mutationRate*, potem otrok mutira. Mutacijo izvaja funkcija *mutateTree*, ki kot parametre sprejme posameznika, ter parameter *mutationAmount*, ki nam pove, kolikšen del genov se bo spremenil. V funkciji se zopet (za vsak gen) izračuna naključno število, in če je manjši od *mutationAmount*, se genu dodeli naključno vrednost. Funkcija nato vrne spremenjenega otroka. Parameter števila otrok je ponavadi manjši od velikosti populacije, zato funkcija po mutaciji generira toliko naključnih dreves, da obdrži velikost populacije.

3.3.6 Prekinitev delovanja

Na prekinitev funkcije *GAOptimize* vpliva več dejavnikov. Prvi je, če se doseže maksimalno število generacij, ki je označeno z parametrom $maxIter$. Funkcija se prekine tudi, če se najboljši posameznik ne spremeni več kot $sameBestIter$ generacij. Tretji način pa je, če vrednost FF najboljšega posameznika preseže $minFitnessTerm$. Parameter $minFitnesTerm$ je privzeto nastavljen na vrednost 0, kar pomeni, da se na ta način funkcija ne bo prekinila.

3.3.7 Interakcija dreves

Dve drevesi generiramo tako, da za vsako drevo posebej zaženemo GA. Najprej je na vrsti drevo, ki je bližje soncu, nato pride na vrsto še drugo drevo, z prej generiranim drevesom kot oviro.

3.4 Prikazovanje rezultatov

Za prikazovanje rezultatov aplikacija uporablja prikazovalnik, izdelan s knjižnico Three.js, ki nam omogoča lažjo uporabo vmesnika WebGL. WebGL je zasnovan na osnovi OpenGL ES ($forembededsystems$) 2.0 in za izrisovanje uporablja element HTML5 Canvas. Omogoča nam uporabo grafične procesne enote (GPU) na spletnih straneh. Aplikacija uporablja Three.js-ov WebGlRenderer.

3.5 Uporabniški vmesnik

Za izdelavo uporabniškega vmesnika uporablja aplikacija knjižnico $dat.gui.js$. Knjižnica omogoča neposredno manipulacijo spremenljivk, podpira mape (angl. folder) in omogoča dogodke ob spremembah posameznih elementov. Programsko lahko animiramo tudi tranzicije parametrov, ki so potem vizualizirane na samem vmesniku.

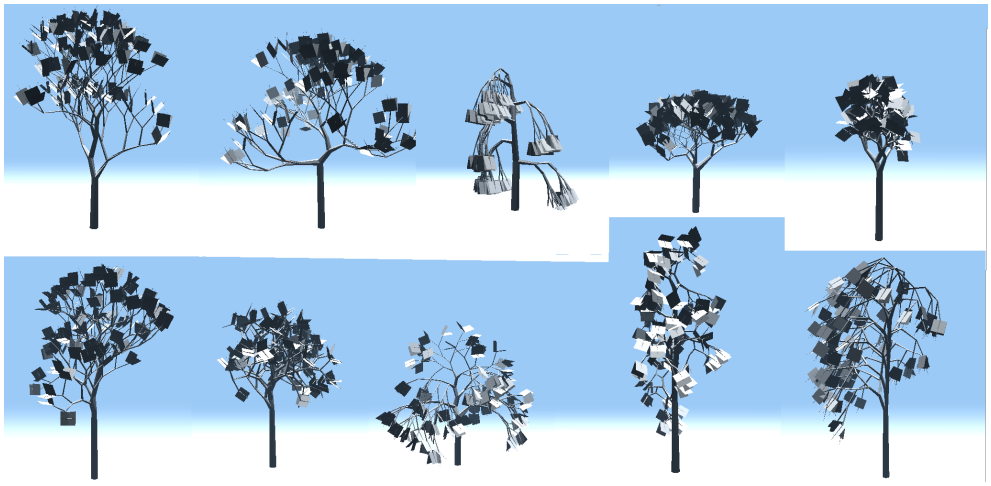
3.6 Paralelizacija

Paralelizacija programa je izvedena z uporabo spletnih delavcev HTML5. Spletni delavci omogočajo izvajanje daljših izsekov kode, brez da bi se prekinilo delovanje glavnega programa. Komunikacija med delavci in glavnim programom poteka preko sporočil. Sporočila pri prejemniku sprožijo dogodek, v katerem se nato sporočilo procesira. Sporočila so lahko katerega koli podatkovnega tipa in nimajo omejene velikosti. Aplikacija uporablja spletne delavce za izvajanje GA v ozadju. Tako lahko uporabnik še vedno premika kamero, izris pa se kljub temu, da poteka optimizacija drevesa, ne prekine. Glavni program delavcu pošilja spremembe lokacije luči ter, ko želi izvesti GA, vse potrebne parametre. Glavni program od delavca prejme najboljšega posameznika. Ko se ta posameznik znotraj delovanja spremeni, delavec pošlje sporočilo. Enako sporočilo nato pošlje tudi ob koncu izvajanja.

Poglavje 4

Rezultati

Model za generiranje dreves lahko generira velik nabor različnih dreves (slika 4.1), ki nam omogoča preiskovanje velike iskalne domene. Iskalna domena se izkaže za zelo kaotično. Funkcija uspešnosti ima veliko lokalnih maksimumov, zato je zelo majhna verjetnost, da bomo dobili enako drevo dvakrat zaporedoma.

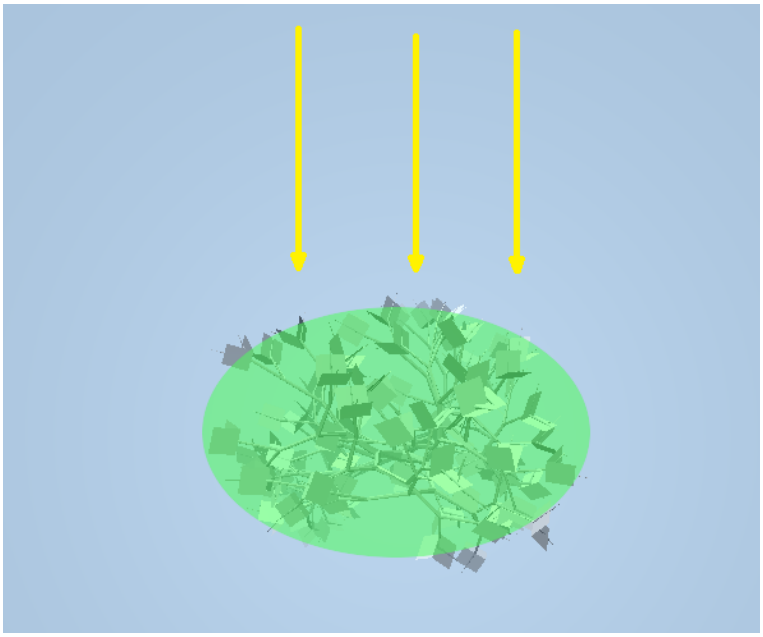


Slika 4.1: Nabor naključno generiranih dreves.

Drevesa, ki jih prejmemo kot rezultat genetskih algoritmov, v primerjavi z naključnimi drevesi kažejo izboljšano obliko, oziroma obliko, bolje prilagojeno vpadnem kotu svetlobe (slika 4.2).

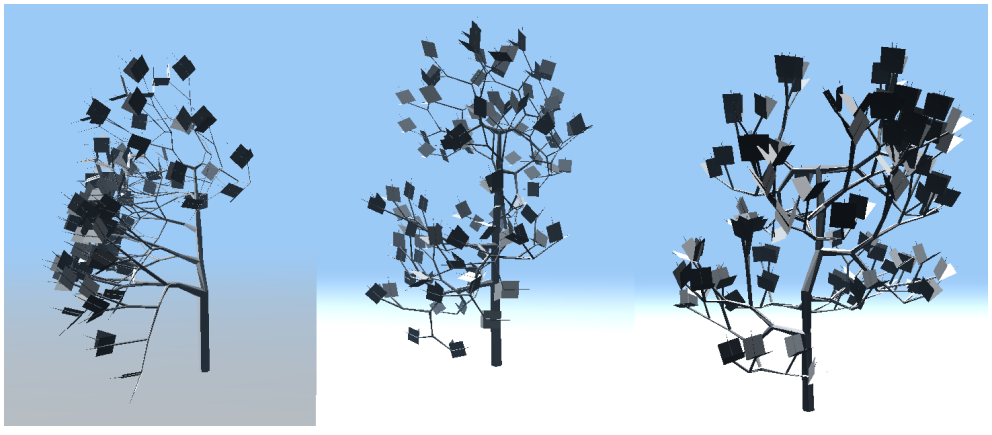
Slika 4.2: Primerjava med naključnim drevesom (levo) in drevesom dobljenim z GA z virom svetlobe nad njim (desno)

Drevesa, dobljena z genetskimi algoritmi, rastejo tako, da bodo listi bolje izpostavljeni svetlobi. Ponavadi se oblika drevesa razširi v smer pravokotno na smer svetlobe (slika 4.3). Slika prikazuje spremembo oblike glede na izvor svetlobe. Rumene puščice prikazujejo smer svetlobe, zelena elipsa pa grobo opiše obliko drevesa. Iz slike je razvidno, da je oblika drevesa razširjena v smeri, pravokotni na smer svetlobe.



Slika 4.3: Vpliv svetlobe na obliko drevesa.

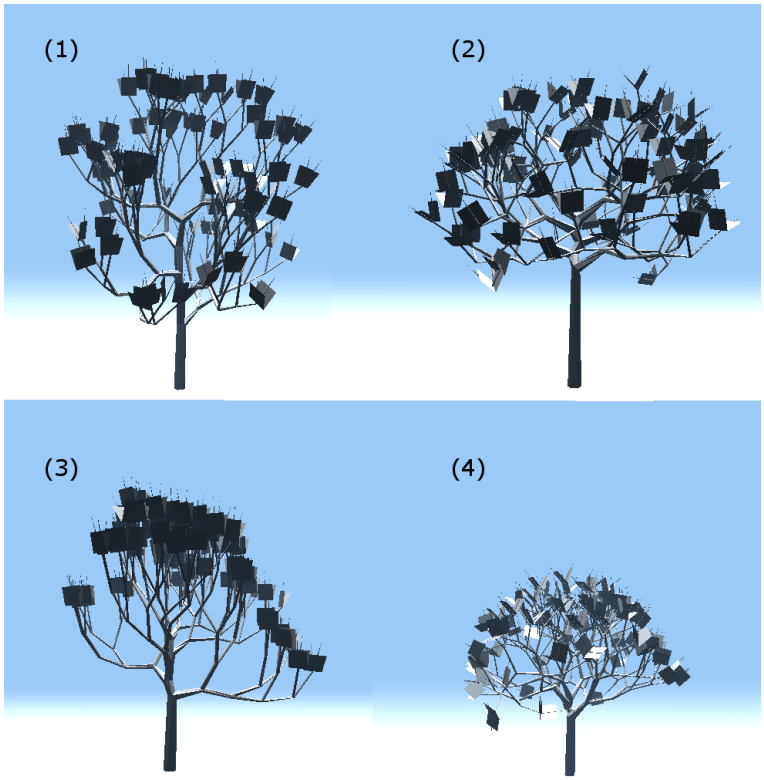
Spodnja slika (4.4) prikazuje zaporedno generirana drevesa pod različnimi koti svetlobe. Listi se ponavadi "naberejo"na strani drevesa, ki je direktno izpostavljena svetlobi.



Slika 4.4: Vpliv kota svetlobe na distribucijo listov. Levi ima svetlobo na svoji levi strani, na srednjega sije diagonalno z leve, na desnega pa iz vrha.

4.1 Vpliv parametrov FF na videz drevesa

Funkcija uspešnosti uporablja dve stikali: *useTrunk* in *useFacing*. Stikalo *useTrunk* povzroči, da funkcija pri izračunu trkov svetlobe z geometrijo upošteva tudi geometrijo debla. Stikalo *useFacing* pa povzroči, da se listi obrnejo pravokotno na smer svetlobe. Torej svetloba mora na njih padati čim bolj pod pravim kotom (Za bolj natančen opis preberi poglavje 3.3.2). Spodnje slike (4.5) prikazujejo vpliv teh parametrov na drevesa pod istimi pogoji.



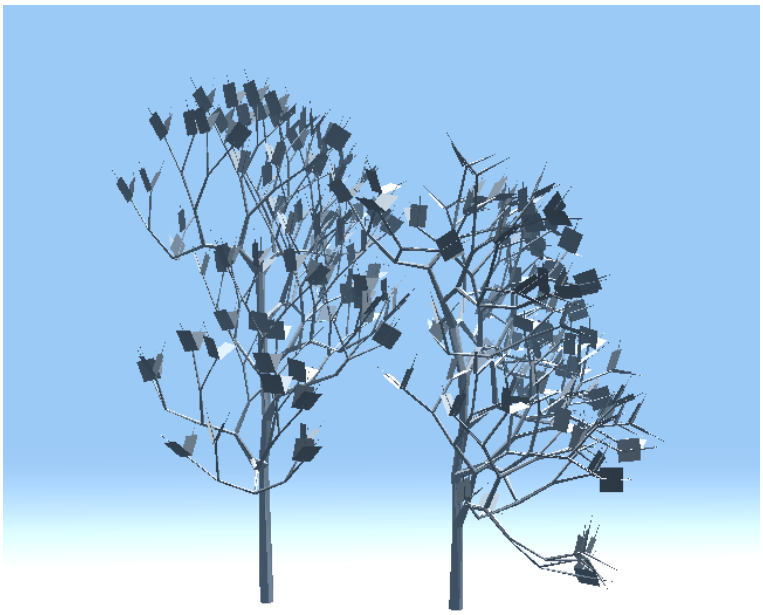
Slika 4.5: Drevesa so obsijana z vrha. (1) brez stikal, (2) *useFacing*, (3) *useTrunk*, (4) *useFacing* in *useTrunk*

Pri (1) je razvidno, da trk z deblom ni upoštevan, saj ima dobljeno drevo liste blizu deblu. Pri (2) je uporabljeno stikalo *useFacing*, zato se spremeni

distribucija listov tako, da je oblika bolj okrogla. Listi na tem drevesu niso nabrani samo na strani kjer je izvor svetlobe. Pri (3) se listi naberejo na vrhu drevesa, saj jih tako ne zakrivajo drugi listi ter deblo. Pri (4) imamo kombinacijo prejšnjih oblik, listi so na vrhu in oblika je bolj kroglasta.

4.2 Interakcija med drevesi

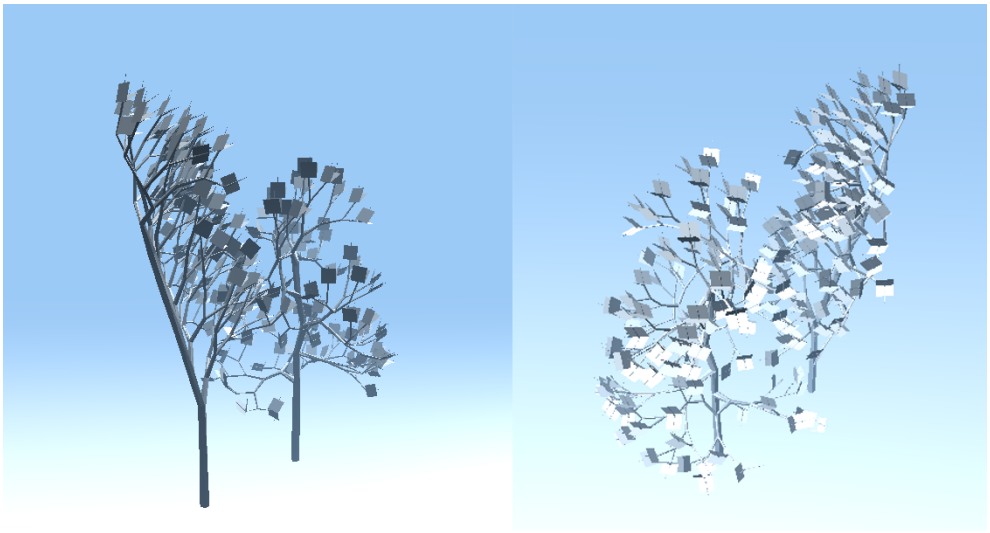
Največja moč genetskih algoritmov se razkrije pri generacij dveh dreves, ki ovirata ena drugo. Tukaj se res izkaže adaptivna narava GA, saj isti algoritem, samo z dodatnimi ovirami, pravilno prilagodi generirano drevo.



Slika 4.6: Primer interakcije dreves.

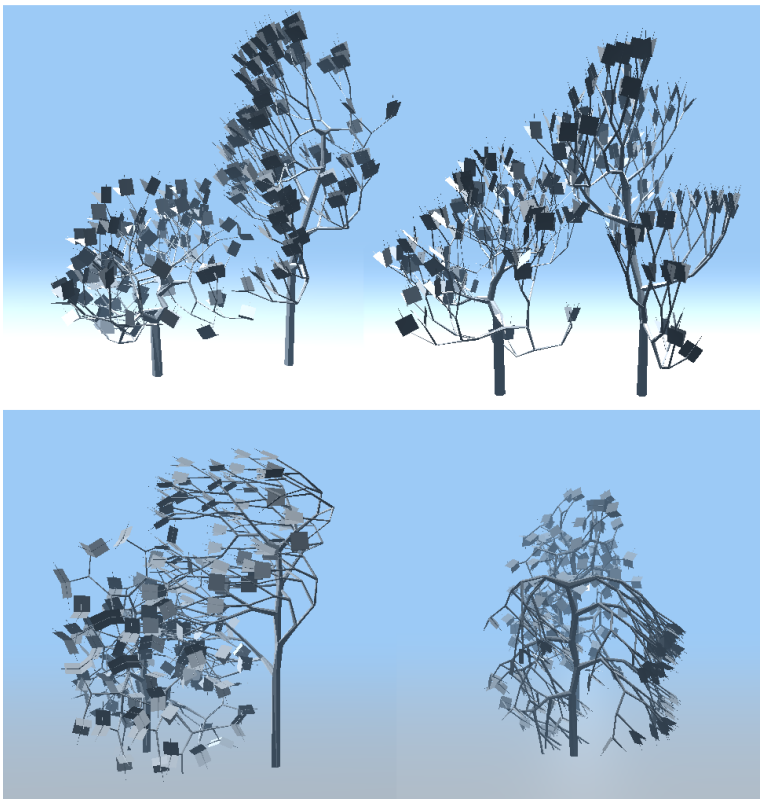
Če pri optimizaciji upoštevamo tudi drugo drevo, bomo dobili drevo, ki raste tako, da njegovih listov drugo drevo ne zakriva. Kot primer si oglejmo sliko 4.6. Na njej je bilo naprej generirano desno drevo, sonce pa se nahaja v zgornjem desnem kotu slike. Drugo drevo je zato zraslo višje kot prvo in

tako izpostavilo večje število svojih listov svetlobi.
Spodnja slika (4.7) prikazuje še en zanimiv rezultat interakcije. Drugo generirano drevo postavi veje na stran tako, da jih ne zakrije sprednje drevo. Zadnje drevo pri tem tudi upogne deblo.



Slika 4.7: Primer interakcije dreves pri katerem drevo ukrivi deblo na stran.

Spodnje slike (4.8) prikazujejo še nekaj primerov interakcije dreves.



Slika 4.8: Dodatni primeri interakcije. Zgornja sliki sta osvetljeni iz vrha (leva rahlo iz diagonale), spodnji pa iz strani. Posebej je zanimiv spodnji desni primer, kjer sta drevesi slikani s strani drugega drevesa, kjer se je zadnje drevo razvejilo v obe strani in tako izpostavilo liste svetlobi.

4.3 Težave in možne nadgradnje

Največja težava algoritma je počasno ocenjevanje uspešnosti dreves. Algoritem funkcije bi lahko pospešili z dodatno paralelizacijo ocenjevanja. Pri tem bi bilo dobro to izvajati na grafično enoti, saj ima večjo vzporedno storilnost. Postopek bi lahko tudi izboljšali z vpeljavo hierarhije očrtanih teles(angl. Bounding volume hierarchy), s pomočjo katere bi prej vedeli, s katerimi poligoni bo žarek dejansko interagiral in nam tako ne bi bil potreben prehod čez vse poligone.

Program bi lahko razširil s podporo več kot dveh dreves in bi tako lahko z njim generirali cele gozdove. Skladno s tem bi bilo dobro dodati tudi izvoz generiranih scen. Interakcijo med drevesi bi lahko tudi pospešili tako, da bi drevesom izdelali preprostejše konveksne ovojnice in trke računali z njimi. Dobro bi bilo razmisliti tudi o drugačnem modelu osvetljevanja, saj trenutni model predpostavi, da je vir svetlobe statičen.

Poglavje 5

Zaključek

Izdelana aplikacija prikazuje, da je pristop prilagajanja dreves na okolico z uporabo genetskih algoritmov uspešen. Drevesa, dobljena z optimizacijo, rastejo tako, da so veje obrnjene proti svetlobi. Pri tem upoštevajo druga drevesa in se glede na njih prilagodijo.

Genetski algoritmi se izkažejo kot pravilna odločitev za reševanja tako visoko dimenzionalnega problema, katerega bi težko reševali s standardnimi iskalnimi algoritmi. Genetski algoritmi so tudi za reševanje tega problema zanimivi, saj najdejo veliko različnih načinov, kako rešiti isti problem in zato v dveh zaporednih iteracijah ne dobimo dveh enakih dreves. Z manjšo razširitvijo aplikacije bi z njo lahko generirali cele gozdove. Opazili smo, da je največja težava aplikacije počasno izvajanje ocene uspešnosti dreves, zato bi z nadaljnjo optimizacijo ocenjevalne funkcije rezultate lahko generirali veliko hitreje.

Torej, če povzamemo, pristop generiranja dreves z genetskimi algoritmi nudi adaptiven in precej robusten način samodejnega generiranja svetlobi prilagojenih dreves.

Literatura

[1] P. Prusinkiewicz, A. Lindenmayer. "The Algorithmic Beauty of Plants",*Springer-Verlag New York, Inc.*, 1996, ch. 1–2.

[2] A. Runions, B. Lane, P. Prusinkiewicz. "Modeling Trees with a Space Colonization Algorithm",*Eurographics Workshop on Natural Phenomena*, 2007.

[3] J. Nystad. "Parametric Generation of Polygonal Tree Models for Rendering on Tessellation-Enabled Hardware",*Norwegian University of Science and Technology Department of Computer and Information Science*, 2010.

[4] J. Weber, J. Penn. "Creation and Rendering of Realistic Trees",*US Army Research Lab*, 1995.

[5] Roulette-wheel selection. [Online]. Dosegljivo: `http://opticalengineering.spiedigitallibrary.org/mobile/article.aspx?articleid=1157793`. [Dostopano 18. 8. 2015].

[6] Roulette-wheel selection. [Online]. Dosegljivo: `http://www.geatbx.com/docu/algindex-02.html`. [Dostopano 18. 8. 2015].

[7] Stochastic universal sampling. [Online]. Dosegljivo: `https://en.wikipedia.org/wiki/Stochastic_universal_sampling`. [Dostopano 18. 8. 2015].

[8] Genetic algorithm. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Genetic_algorithm. [Dostopano 18. 8. 2015].

[9] CrossOver. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm). [Dostopano 18. 8. 2015].

[10] Introducition to genetic algorithms. [Online]. Dosegljivo: http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html. [Dostopano 18. 8. 2015].

[11] Ray casting. [Online]. Dosegljivo: https://www.cs.princeton.edu/courses/archive/fall00/cs426/lectures/raycast/raycast.pdf. [Dostopano 19. 7. 2015].

[12] Knjižnica procTree. [Online]. Dosegljivo: https://github.com/supereggbert/proctree.js/. [Dostopano 19. 7. 2015].

[13] Three.js. [Online]. Dosegljivo: http://threejs.org/. [Dostopano 18. 8. 2015].

[14] Introducition to HTML 5 Web Workers. [Online]. Dosegljivo: http://cggallant.blogspot.com/2010/08/introduction-to-html-5-web-workers.html. [Dostopano 18. 8. 2015].