Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Rok Kogovšek

Tehnike proceduralnega modeliranja v računalniški grafiki

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Matija Marolt

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00072 / 2013
Datum: 4.4.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ROK KOGOVŠEK**

Naslov: **TEHNIKE PROCEDURALNEGA MODELIRANJA V RAČUNALNIŠKI GRAFIKI**
PROCEDURAL MODELING APPROACHES IN COMPUTER GRAPHICS

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

V diplomski nalogi preučite področje proceduralnega modeliranja v računalniški grafiki. Izberite najznačilnejše in po namenu različne pristope k proceduralnemu modeliranju, jih podrobno opišite in implementirajte v obliki demonstracijskih orodij, ki omogočajo fleksibilno širitev pravil generiranja geometrije. Podajte tudi analizo implementiranih pristopov.

Mentor:

doc. dr. Matija Marolt

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Rok Kogovšek, z vpisno številko **63100185**, sem avtor diplomskega dela z naslovom:

Tehnike proceduralnega modeliranja v računalniški grafiki

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matija Marolt,
- 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 v zbirki "Dela FRI".

V Ljubljani, dne 5. februar 2014 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Matija Maroltu, ki me je vzel pod svoje mentorstvo in mi omogočil raziskavo področja, ki sem ga tekom študija vzljubil. Zahvalil bi se tudi družini in prijateljem, ki so mi stali ob strani v času študija, predvsem ob težavah in mi poleg pomoči hkrati dajali tudi poguma in zaleta za nadaljevanje začrtane poti. Zahvalo si tudi zaslužijo za potrpežljivost ob dolgotrajni izdelavi diplomskega dela. Mateja, Franci, Eva, Žan, Štefan in Martin, hvala za vse in naj bom še nadalje v vaši dobri volji.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Fraktalna geometrija **3**

2.1 Splošno 3

2.2 Algoritmi in tehnike 6

2.2.1 MRCM 6

2.2.2 FRCM 8

2.3 Implementacija MRCM 11

2.3.1 mrcm.graphics 11

2.3.2 mrcm.instructions 12

2.3.3 mrcm.gui 12

2.3.4 Definiranje novih fraktalnih modelov 13

2.3.5 Primeri rezultatov 14

3 L-sistem **19**

3.1 Splošno 19

3.1.1 Izris in želva 21

3.1.2 Standardna gramatika 22

3.2 Razširitve 23

3.2.1 Stohastični L-sistem 23

3.2.2 Kontekstno odvisni L-sistem . . . 24
3.2.3 Parametrični L-sistem . . . 25
3.2.4 Upoštevanje okolja . . . 27
3.3 Implementacija LSPS . . . 30
3.3.1 Odstopanja od standarda . . . 30
3.3.2 lsps.graphics . . . 31
3.3.3 lsps.lsystem . . . 31
3.3.4 lsps.plants . . . 31
3.3.5 lsps.gui . . . 32
3.3.6 Definiranje novih rastlin . . . 32
3.3.7 Primeri rezultatov . . . 33

4 Gramatika oblik **39**
4.1 Splošno . . . 39
4.2 Gramatika delitve . . . 42
4.3 CGA . . . 46
4.3.1 Okolje CityEngine . . . 50
4.4 Implementacija SGS . . . 51
4.4.1 sgs.core . . . 51
4.4.2 sgs.buildingComponents . . . 52
4.4.3 sgs.bluePrints . . . 52
4.4.4 sgs.gui . . . 53
4.4.5 Definiranje novih stavb . . . 53
4.4.6 Primeri rezultatov . . . 54

5 Višinska slika **57**
5.1 Splošno . . . 57
5.2 Algoritmi višinskih slik . . . 59
5.2.1 Generiranje napak . . . 60
5.2.2 Rekurzivna razdelitev . . . 61
5.3 Implementacija HMG . . . 64
5.3.1 hmg.generators . . . 64

5.3.2 hmg.graphics 64
5.3.3 hmg.gui 64
5.3.4 Definiranje novih generatorjev površja 65
5.3.5 Primeri rezultatov 66

6 Analiza tehnik in njihovih implementacij **69**

7 Zaključek **73**

Povzetek

Proceduralno modeliranje omogoča računalniško vodeno generiranje velikih količin kvalitetnih grafičnih podob. Opravka imamo s širokim področjem, ki nima univerzalne rešitve, temveč mnogo različnih konceptov. Posledično diplomsko delo pri pregledu, analizi in implementaciji omejimo na štiri pogoste, vendar medseboj različne tehnike. Tako obravnavamo univerzalne fraktale, katerih koncept lahko vgradimo povsod, kjer imamo opravka s samopodobnostjo. Močna izbrana pristopa sta tudi gramatika oblik, ki je primerna za gradnjo modelov arhitekture z dopolnjevanjem preprostega modela v kompleksnejšega, in L-sistem, ki pri gradnji modela simulira rast in se tako predvsem uporablja za rastline. V diplomski nalogi preučimo tudi preproste, vendar učinkovite višinske slike, ki so pogoste pri modelih reliefa, vendar niso omejene nanje.

Ključne besede: računalniška grafika, proceduralno modeliranje, samopodobnost, generiranje terena, generiranje arhitekture, generiranje rastlin, fraktali, MRCM, FRCM, L-sistem, gramatika oblik, gramatika delitve, CGA, višinska slika

Abstract

Procedural modeling offers means for computer guided generation of graphical models in high quantity and quality. Since it is a wide field of research it does not have an universal solution but offers many different concepts. This thesis consequently introduces, analyses and implements just the four most common techniques with different concepts. The most universal concept are fractals, which can be used for every model with a certain degree of self-similarity. Strong techniques are also shape grammar and L-system. Shape grammar extends a simple model in steps into a more complex one and it is therefore appropriate for generating architecture. In contrast L-system simulates the model's growth, which makes it most commonly linked to plant models. The last selected concept is the simple but efficient heightmap, which is common in terrain generation but not restricted to that.

Key words: computer graphics, procedural modeling, self-similarity, generating terrain, generating architecture, generating plants, fractals, MRCM, FRCM, L-system, shape grammar, split grammar, CGA, heightmap, heightfield

Poglavje 1

Uvod

Najzamudnejše opravilo na področju računalniške grafike je gotovo modeliranje predmetov. Čeprav imamo na voljo razna grafična orodja za modeliranje kot npr. Autodesk Maya ali Blender, ki nas sprostijo iz okov modeliranja vsakega posameznega poligona, je delo še zmeraj zamudno. Zadeva postane redudantna, ko imamo problem modeliranja skupine različnih predmetov, ki so si sicer podobni v osnovah. Dandanes bomo težko našli animacijo ali simulacijo, ki bi imela mesta, gozdove, relief tal, množice ljudi ipd. modelirane za vsak element skupine posebej. Takšne probleme grafičnega modeliranja rešujemo s proceduralnim modeliranjem.

Pod termin proceduralno modeliranje spadajo vse tehnike in procedure, s katerimi računalniku postavimo pravila, po katerih računalnik sam modelira posamezne predmete. Že iz samega opisa termina je razvidno, da je to področje široke narave s stalnim potencialom širjenja. Morda je ravno to razlog za slab pregled celotnega področja. Tekom zbiranja literature se je namreč izkazalo, da proceduralno modeliranje nima neke *de facto* literature, saj takšna literatura obstaja zgolj za nekatere tehnike. Preostali pristopi, ki sem jih tekom raziskovanja odkril, so celo v znanstvenih člankih redko zastopani.

Cilj diplomske naloge je tako postal zbrati čim več pogostih tehnik proceduralnega modeliranja, izmed njih izbrati štiri najpogostejše in najbolj z

literaturo podkrepljene tehnike, jih medseboj primerjati in tudi implementirati učne primere posameznih tehnik. Med izbranimi sta najbolj obče poznani tehniki *fraktali* iz poglavja 2 in *višinske slike* iz poglavja 5. Fraktalna geometrija definira modele preko samopodobnosti, kjer model gradimo kakor skupek pomanjšanih kopij samega modela. Klasičen primer je drevo sestavljeno iz vej oz. pomanjšanih kopij, ki so prav tako skupek pomanjšanih kopij itd. Višinske slike nasprotno ne definirajo podobe modela preko grafičnih elementov, temveč preko matrike številskih vrednosti. Matrike vrednosti običajno hranimo kot slike zaradi kompaktnosti in nazorne predstave končne podobe modela. Ime tehnike izhaja iz pogoste uporabe pri modeliranju višine terena. Preostali tehniki uporabljata gramatike za gradnjo modelov. *L-sistem*, glej poglavje 3, uporablja gramatiko nizov, s katero najprej zgradi znakovni niz, ki predstavlja načrt zgradbe modela. Nato preko branja niza sestavi iz osnovnih grafičnih elementov končno grafično podobo modela. Tehnika simulira rast in je tako pogosta pri generiranju rastlinja. Drugačen pristop primeren za gradnjo arhitekture predstavlja *gramatika oblik* iz poglavja 4, ki model gradi z dopolnjevanjem obstoječega modela. Gramatika oblik prav tako ne uporablja v svoji gramatiki znakovne nize, temveč grafične elemente, in tako neposredno gradi končno grafično podobo modela.

Implementacije tehnik so sprogramirane s programskim jezikom Java 6 in grafično prikazane s programskim vmesnikom OpenGL 2.0. Za povezavo med Javo in OpenGL skrbi knjižnica LWGJL 2.8.5. Implementacije bi lahko služile kot učna osnova za nadaljne generacije. Vendar je potrebno opozoriti, da so zaradi velike širine področja predstavljene tehnike zgolj del pogostejših tehnik, ki imajo tudi dovolj literature za podporo. Poleg tega je težko predvideti aktualnost predstavljenih tehnik, ki bo odvisna predvsem od nadaljnega razvoja, saj kažejo mnoge nove tehnike velik potencial na področju proceduralnega modeliranja.

Poglavje 2

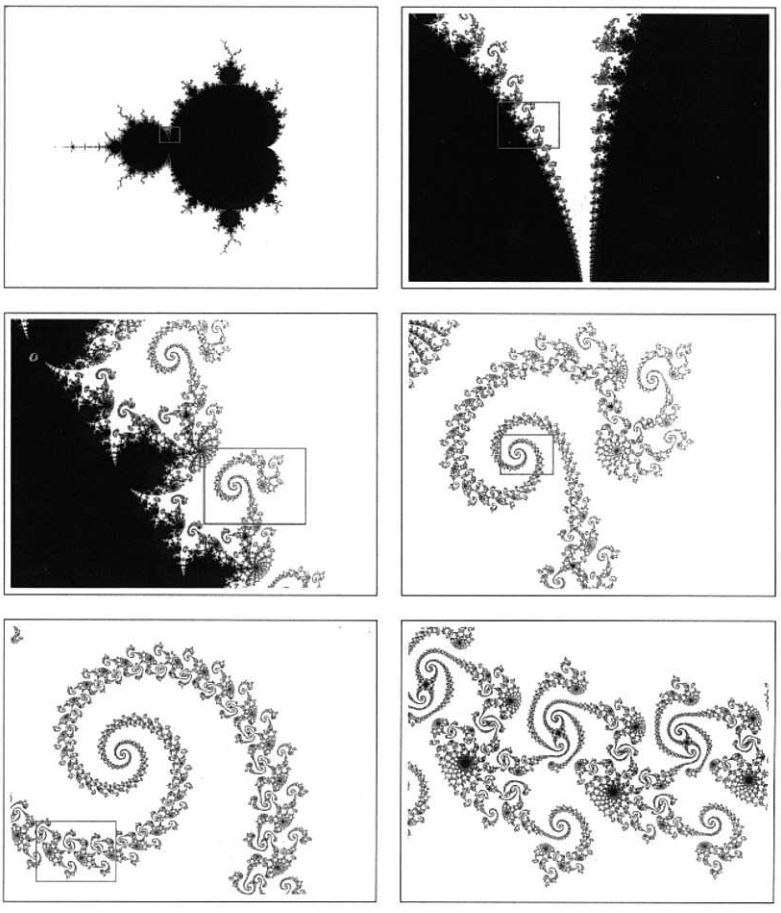
Fraktalna geometrija

2.1 Splošno

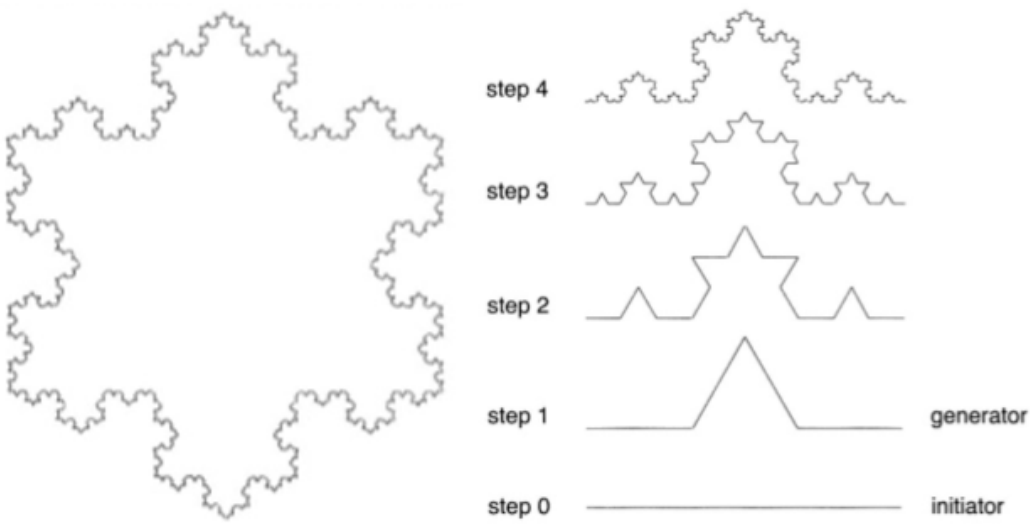
Fraktalna geometrija je razširitev klasične geometrije, ki s fraktali omogoča opise kompleksnih predmetov, kot so oblaki, rastline in gore, enako kvalitetno kot opise arhitekture. Ob fraktalih pogosto omenimo njihovega odkritelja Benoita Mandelbrota, ki jih je poimenoval po latinski besedi *fractus*, kar pomeni *razbito*. Prvi je tudi prikazal njihovo lepoto z računalniško grafiko, kjer je dokazal, kako navidezno naključni pojavi sledijo določenemu vzorcu, če jih le razbijemo na dovolj majhne dele. Matematično gledano je fraktal podmnožica geometrijskega prostora. Za nas so zanimivi, ker lahko z njimi gradimo strukture podmnožic različnih enostavnih geometrijskih prostorov[1][2].

V procedularnem modeliranju se je pod pojmom uveljavila prožnejša definicija. Tu se govori o fraktalih, kadar imamo opravka z modelom, ki ima visoko stopnjo samopodobnosti. Poenostavljeno lahko rečemo, da je model pravzaprav sestavljen iz transformiranih različic modela[3]. Samopodobnost omogoča fraktalom visoko stopnjo kompresije, saj so algoritmi izgradnje običajno tako enostavni, da zavzamejo mnogo manj prostora kot slika generiranega modela. Enostavnost algoritmov je posledica njihove rekurzivne oz. iterativne narave, kar včasih privede tudi do paradoksnih lastnosti modelov[4].

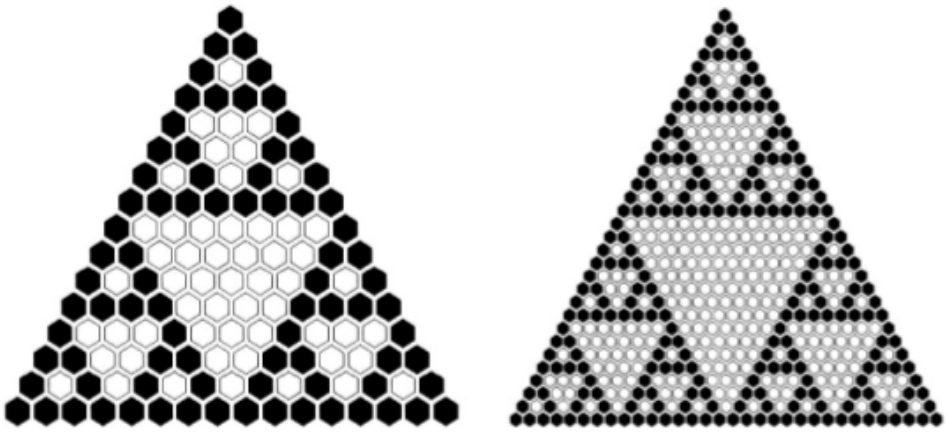
Najslavnejšo vizualizacijo fraktalov ima zagotovo Mandelbrotova množica, ki je Mandelbrota tudi pripeljala do fraktalov, ko raziskoval rekurzivne funcije s kompleksnimi števili. Klasičen primer fraktalov v geometriji je tudi Kochova krivulja in njena uporaba v Kochovi snežinki. Skozi iteracije se črte razdeli na tri dele, na kar se srednjo črto zamenja s trikotnikom, ki mu nato odstranimo spodnjo stranico. Če za osnovo uporabimo trikotnik z danim postopkom, ustvarimo Kochovo snežinko. Kochova snežinka je tudi klasični primer paradoksnih lastnosti modela, saj njena površina limitira h končni vrednosti, medtem ko obseg k neskončnosti. Fraktale lahko najdemo tudi kot številske primere npr. sodost in lihost števil v Pascalovem trikotniku[5][4].



Slika 2.1: Vizualizacija Mandelbrotove množice. Od leve proti desni in od zgoraj navzdol si sledijo povečave označenih delov prejšnje slike[4].



Slika 2.2: Kochova snežinka in Kochova krivulja[5].



Slika 2.3: Pascalov trikotnik, kjer črna predstavlja števila z liho pojavitvijo in bela števila s sodo pojavitvijo[5].

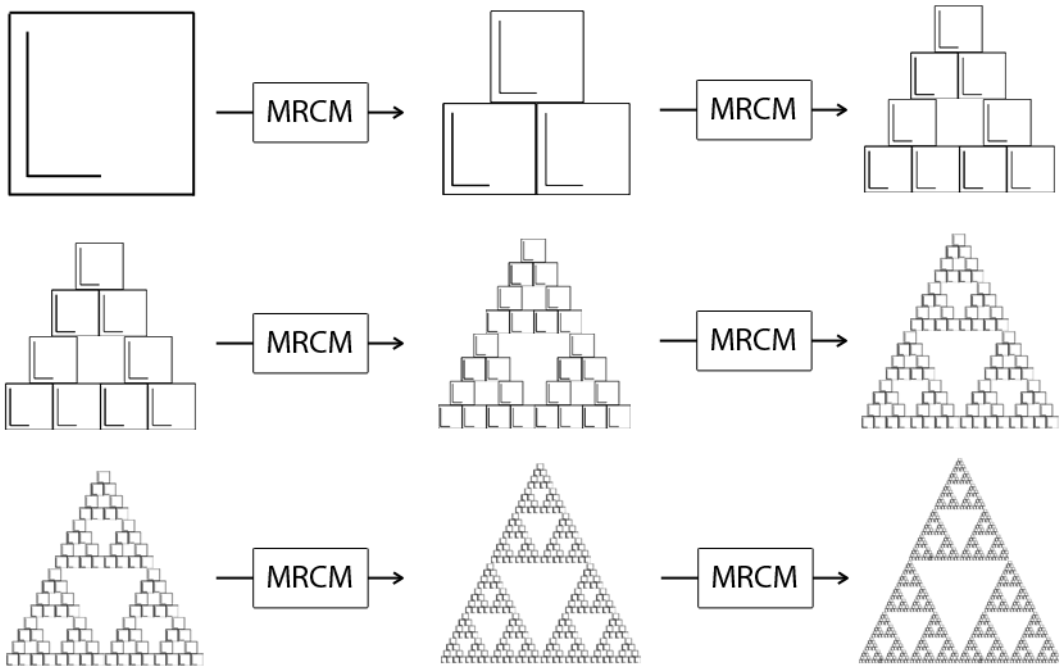
2.2 Algoritmi in tehnike

Kakor poznamo več različnih pristopov k fraktalom, poznamo tudi različne algoritme in tehnike za izdelavo le-teh. Načeloma so zaradi narave fraktalov enostavni, kratki in pogosto rekurzivni. Za bolj matematične in učne primere lahko opazujemo rekurzivne funkcije s kompleksnimi števili, ki nas pripeljejo do Mandelbrotove množice, vendar bomo zaradi preučevanja tehnik procedularnega modeliranja raje preučili pristopa, ki imata bolj praktične uporabe. Oba pristopa pripadata iterativnim funkcijskim sistemom, pri čemer se razlikujeta po deterministočnosti postopka izgradnje končne podobe fraktalnega modela. V nadaljevanju bomo tako preučili deterministični MRCM in stohastični FRCM. MRCM je pravzaprav poenostavitev FRCM-ja in imamo tako pri obeh opravka vselej z isto osnovno idejo[3][4][5].

2.2.1 MRCM

Multiple Reduction Copy Machine oz. MRCM algoritem je deterministični rekurzivni algoritem, ki simulira delovanje kopirnega stroja. Ideja algoritma je na začetku definirati začetni element in afine linearne transformacije rekurzije. Znotraj rekurzije se začetni element običajno skalira na manjšo velikost, na kar iz transformiranega elementa ustvarimo željeno število kopij. Na kopijah se nato izvedejo transformacije, ki smo jih definirali v algoritmu. Sama dolžina in zahtevnost algoritma je posledično odvisna od števila transformacij, potrebnih za fraktalni model. Pri primeru trikotnika Sierpinskega se za začetni element izbere lahko karkoli, enostaven in poučen je že kar kvadrat, pri transformacijah rekurzije pa skalirane kopije vhodnega elementa transliramo na zgornji in desni rob vhodnega elementa. Skozi ponavljanje rekurzije se podoba modela venomer izboljšuje in bliža k svoji končni podobi oz. atraktorju. Ne glede na začetni element je atraktor tu vselej isti, kar pa ni nujno za vse modele. Pri nekaterih modelih imamo lahko več različnih atraktorjev, h katerim se bližamo glede na začetno stanje modela. Če za začetni element vstavimo kar sam atraktor, se posledično podoba modela ne spremeni veliko,

saj smo začeli kar na cilju[4][5].



Slika 2.4: Izdelava trikotnika Sierpinskega z MRCM algoritmom[4].

Izdelavo trikotnika Sierpinskega smo vključno s skaliranjem omejili na tri transformacije, kar bi pri drugih fraktalnih modelih najverjetneje predstavljalo premajhno število transformacij. Na splošno velja, da lahko definiramo poljubno število transformacij, vendar ravno število in kvaliteta le-teh najbolj določata stopnjo samopodobnosti. Za lažji pregled MRCM-ja definiramo število transformacij pod oznako n in globino rekurzije z d. Običajno imajo modeli končno podobo sestavljeno iz n^d miniaturnih transformiranih različic začetnega elementa, pod pogojem, da ima vsaka kopija nad seboj izpeljano zgolj eno transformacijo. Posledično se z večanjem d-ja manjša kvaliteta kopij začetnega elementa, saj po vsakem skaliranju element zavzame vedno manjši prostor. Ta lastnost še toliko bolj približa algoritem k izvorni ideji kopirnega stroja, kjer tudi poznamo težave s kopijami kopij. Pri primeru trikotnika Sierpinskega v sliki 2.4 je to razvidno, če opazujemo črko L znotraj začetnega kvadrata, ki skozi rekurzije postaja vse manj vidna. Če bi

govorili o neskončni globini rekuzije, bi naši fraktali postali kar točke, vendar smo seveda omejeni s končno globino. Idealna podoba modela se nam pri danem algoritmu tako izmakne, vendar ni nedosegljiva. Težavo se da rešiti z razširitvijo algoritma v FRCM, kjer aproksimiramo končno podobo modela, glej poglavje 2.2.2[4].

Čeprav postane podoba modela zanimivejša z večanjem d-ja, ne smemo pozabiti na n^d kopij začetnega elementa, ki jih je potrebno ustrezno transformirati in izrisati. Dilema globine postane zanimiva, ko želimo pridobiti čim lepše podobe, za katere potrebujmo večji d, kar eksponentno veča računsko delo na fraktalih. Čeprav z d-jem potrebno računsko delo hitro narašča, hitrost manjšanja velikosti fraktalov upada. Še huje je, kadar transformacije znotraj rekurzije ne zmanjšajo fraktalov za velik faktor, saj takrat potrebujemo še toliko večji d[4].

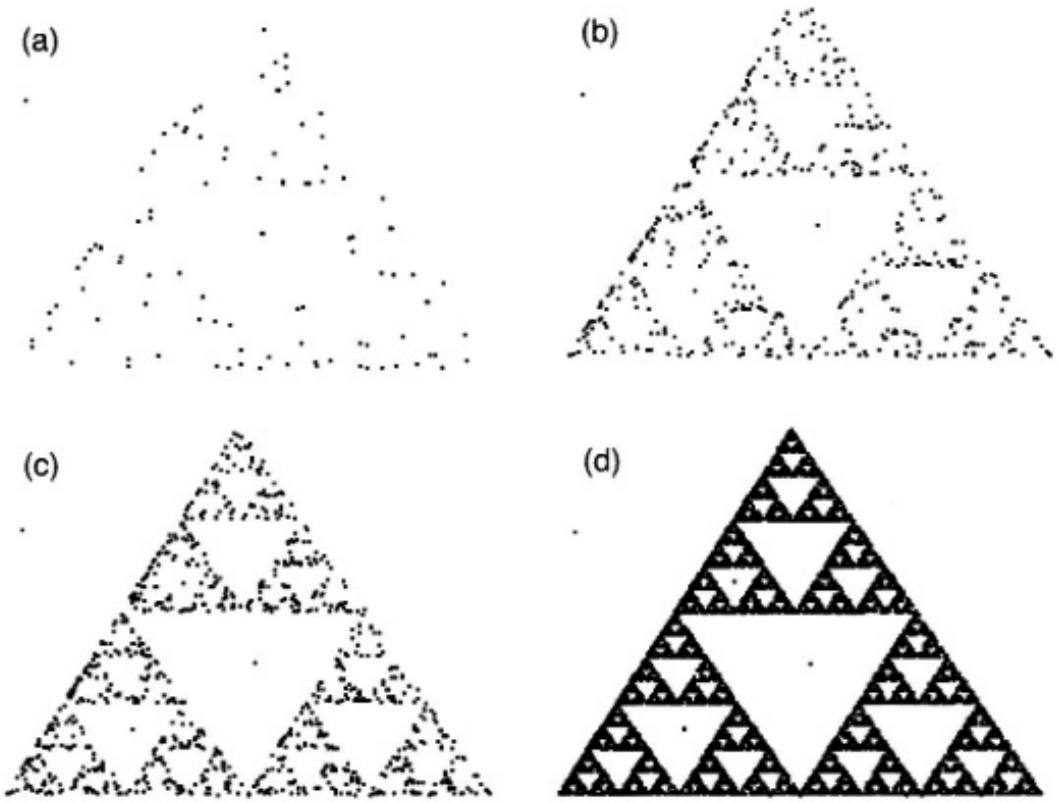
2.2.2 FRCM

Fortune Wheel Reduction Copy Machine oz. FRCM je rekurzivni algoritem, ki v MRCM vpelje naključje in igro kaosa. Namesto kopiranja celotne podobe znotraj rekurzije se tu kopira in transformira zgolj posamezno točko z le eno naključno izbrano transformacijo iz nabora transformacij oz. pravil. Algoritem nas tako privede iz začetnega elementa preko kopičenja generiranih točk do aproksimacije atraktorja, do katere se z MRCM le počasi bližamo, glej poglavje 2.2.1[4][5].

Za razumevanje algoritma je potrebno razumeti nekatere lastnosti atraktorja oz. idealne podobe fraktalnega modela[4]:

1. Atraktor je sestavljen iz množice točk, do katere se pri MRCM običajno približamo zgolj z neskončno rekurzijami.

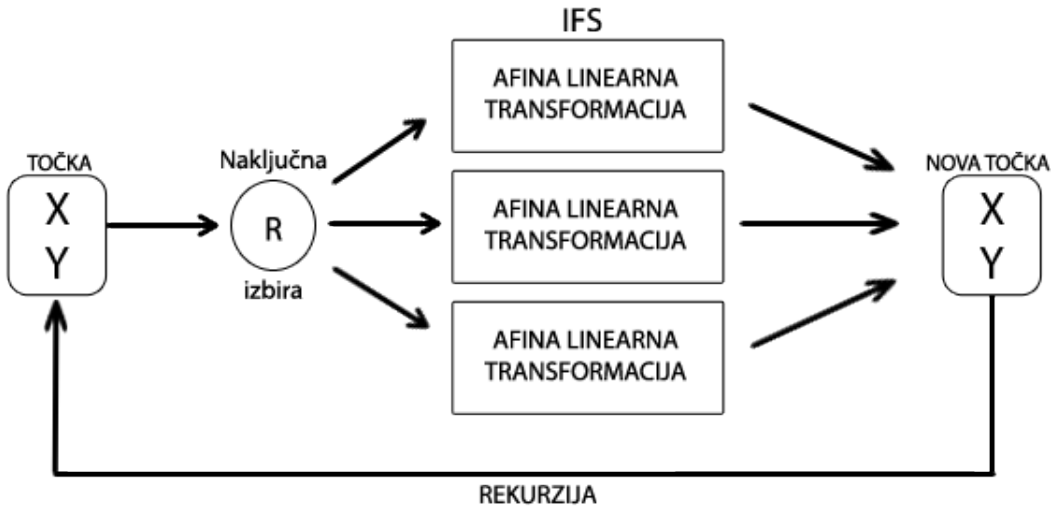
2. Če je točka p del atraktorja, potem za vse afine linearne transformacije $L_i(p)$ velja, da so tudi te del atraktorja. Lastnost znova izraža samopodobnost fraktalnega modela, saj so fraktali minimizirane kopije podobe modela.



Slika 2.5: Izdelava trikotnika Sierpinskega z FRCM algoritmom. Stanje a prikazuje podobo modela pri 100 korakih rekurzije, b pri 500, c pri 1000 in d pri 10.000 korakih. V podobah se jasno vidi tudi začetne točke, ki niso znotraj množice točk atraktorja[5].

3. $L_i(p)$ ima lahko inverzno transformacijo $L_i^{-1}(p)$. Če $L_i^{-1}(p)$ ne obstaja, potem se pri MRCM podoba s $L_i(p)$ vselej transformira v točko ali črto. V primeru obstoja $L_i^{-1}(p)$ se točka p, ki leži znotraj atraktorja, transformira znova v točko znotraj atraktorja. To ponovno izraža samopodobnost fraktalnega modela.

4. Točka p, ki leži izven množice točk atraktorja, bo po $L_i(p)$ transformaciji bližje idealni podobi kakor začetni p. Tako je vselej bolje izbrati točko $L_i(p)$ kakor točko p, če premišljeno ugibamo točko, ki bi se nahajala znotraj množice točk atraktorja.



Slika 2.6: Potek FRCM algoritma znotraj rekurzije[4].

FRCM prav tako kakor MRCM potrebuje defeniran začetni element in afine linearne transformacije. Za razliko se tu od začetnega elementa zgolj naključno izbere točka, s čimer se želimo izogniti slabo generiranim začetnim točkam. Prav tako so transformacije oz. pravila tu zgolj nabor, iz katerega se naključno izbere transformacija znotraj rekurzije. Pri pravilih je potrebno tudi definirati verjetnosti za izbiro v posamezni rekurziji, saj se z ustrezno porazdelitvijo hitreje bližamo atraktorju kakor z enakomerno porazdelitvijo verjetnosti. Znotraj rekurzije se ustvari kopija vhodne točke, ki je bodisi naključno izbrana točka začetnega elementa bodisi generirana in transformirana točka iz prešnjega koraka rekurzije. Kopijo nato transformiramo z naključno izbranim pravilom v skladu s porazdelitvijo verjetnosti in rezultat vstavimo kot vhodno točko naslednjega koraka rekurzije. Rekurzija običajno teče do generiranja nekaj tisoč točk, kar v primerjavi z MRCM ni tako zahtevno, saj se izvajajo transformacije zgolj na eni točki in ne na celotnem modelu. Začetni rezultati rekurzije pogosto ne bodo blizu atraktorja fraktalnega modela, vendar je zaradi četrte lastnosti atraktorja vsak nadaljni vstop v rekurzijo bližje cilju. Tako lahko mirno zavržemo prvih 100± generiranih točk in s tem izboljšamo končno podobo fraktalnega modela[4].

2.3 Implementacija MRCM

Pri implementaciji algoritma za generiranje fraktalov sem se odločil za MRCM algoritem in ga implementiral kot istoimenski program. Poleg preproste izgradnje algoritem krasi tudi nazorna predstavitev samopodobnosti. Implementacija je zgrajena tako, da uporabnik lahko izbere poljubni začetni element in na njem sproži poljubna navodila. Ker so navodila in začetni elementi grajeni preko dedovanja iz skupnega razreda, je ustvarjanje novih navodil oz. začetnih elementov hitro in enostavno. Za boljše delovanje in preglednost je program razdaljen na pakete *mrcm.graphics, mrcm.gui in mrcm.instructions*.

2.3.1 mrcm.graphics

Razredi znotraj paketa predstavljajo osnovne grafične elemente potrebne za prikaz fraktalov. Poglavitni razred je *MRCMModel3D*, iz katerega gradimo začetne elemente in navodila. Razred samodejno računa celotno velikost modela s pomočjo navideznega okvirja, s čimer je poskrbljeno za pravilno transformiranje modela v nadaljevanju, kakor tudi gradnika, ki hrani vhodni model. Gradnik je pravzaprav kar nova instanca razreda *MRCMModel3D*, s čimer tudi poskrbimo za povezavo med koraki. Pri izrisu se privzeto uporablja sezname za izris vsebin, saj imamo opravka s ponavljajočim se izrisom gradnika, katerega kopije eksponentno naraščajo. Izjema so začetni elementi, ki ne uporabljajo gradnikov, temveč kar zapise poligonov. Tu je potrebno za pravilno delovanje izključiti sezname s spremenljivko *USE_DISPLAYLIST*. Za dejanski izris modela poskrbi *MRCMVisualizator*, ki na začetku tudi požene izbrana navodila in začetni element. Razširitve *MRCMModel3D*, ki definirajo grafično podobo začetnih elementov, hranimo znotraj podpaketa *mrcm.graphics.baseElements*. Trenutna zbirka elementov hrani:

- *Cube* - preprosta kocka.
- *Pyramid* - štiristrana piramida, ki ima za osnovno ploskev kvadrat.
- *Rainbow_Cube* - kocka, ki ima vsako ploskev drugačne barve.

2.3.2 mrcm.instructions

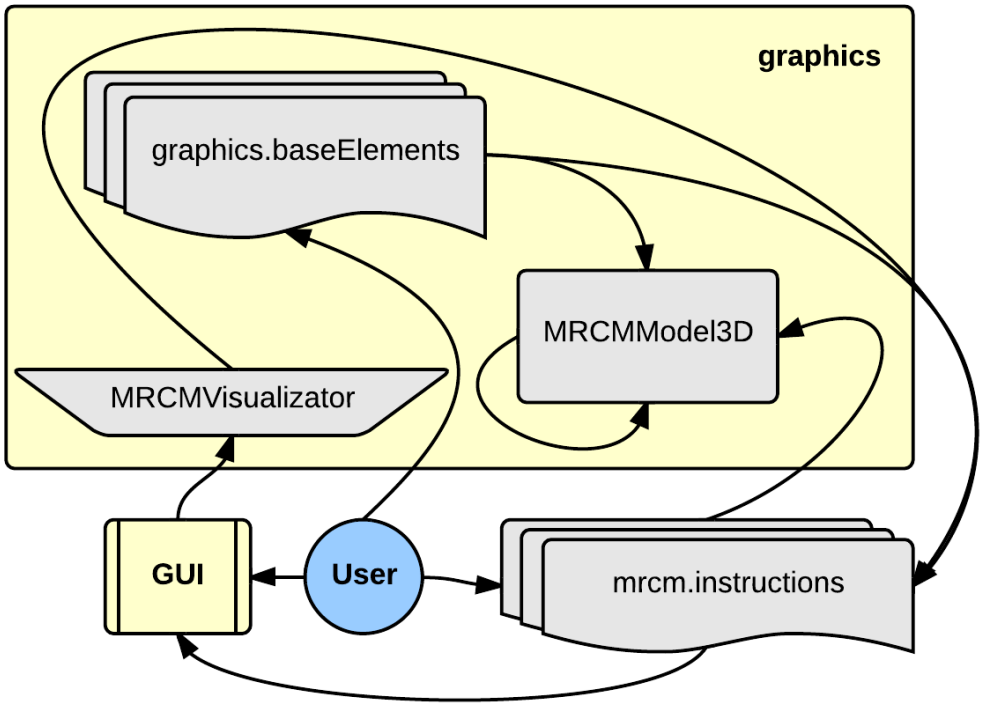
Paket vsebuje navodila, ki razširjajo razred *MRCMModel3D*. V nasprotju z začetnimi elementi navodila ne definirajo fiksnega videza, temveč transformacije gradnika *BB* - okrajšava za *BuildingBlock*, ki na začetku predstavlja izbrani začetni element, in njegove trenutke izrisa. S tem lahko tudi uspešno prikažemo model s poljubnim začetnim elementom in dokažemo, da fraktali ne glede na začetni element sčasoma dosežejo atraktor. Trenutna zbirka navodil vsebuje:

- *Corner_Cube* - skozi iteracije se vsakemu gradniku na kote njegovega okvirja pritrdijo tri za tretjino manjše kopije gradnika.
- *Koch_Cube* - skozi iteracije gradnik razdelimo na 27 enakomernih delov, od katerih zanemarimo sredinske in kotne dele. Pod sredinskimi deli razumemo del v centru gradnika in dele, katerih stranice okvirja se prekrivajo s stranicami okvirja centralnega dela. Ime navodil izhaja iz asociacije na Kochovo snežinko, saj model skozi iteracije pridobi obliko nekakšne tridimenzionalne snežinke.
- *Menger_Sponge* - tudi tu skozi iteracije gradnik razdelimo na 27 enakomernih delov, od katerih tu zanemarimo zgolj sredinske dele.
- *Sierpinski_Pyramid* - razširitev trikotnika Sierpinskega v tri dimenzije, s čimer ustvarimo piramido piramid.
- *Sierpinski_Triangle* - izriše trikotnik Sierpinskega po zgledu slike 2.4.
- *Word_Test* - izriše besedo *TEST*, kjer se posamezni gradniki besede rotirajo glede na lokacijo za bolj tridimenzionalen model.

2.3.3 mrcm.gui

Uporabniški vmesnik omogoča hitro izbiro začetnega elementa in navodil ter enostavno zamenjavo med izvajanjem. S pritiskom na gumb *Start fresh* se

zažene generiranje novega modela na začetku, kakor tudi med generiranjem obstoječega modela. Za prehajanje med koraki modela je na voljo gumb *Next iteration*.



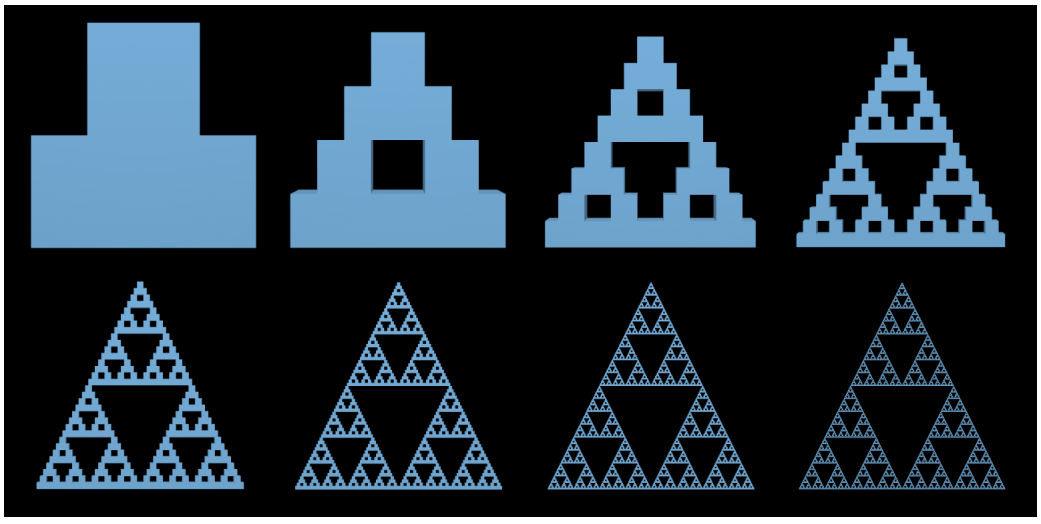
Slika 2.7: Arhitektura MRCMja.

2.3.4 Definiranje novih fraktalnih modelov

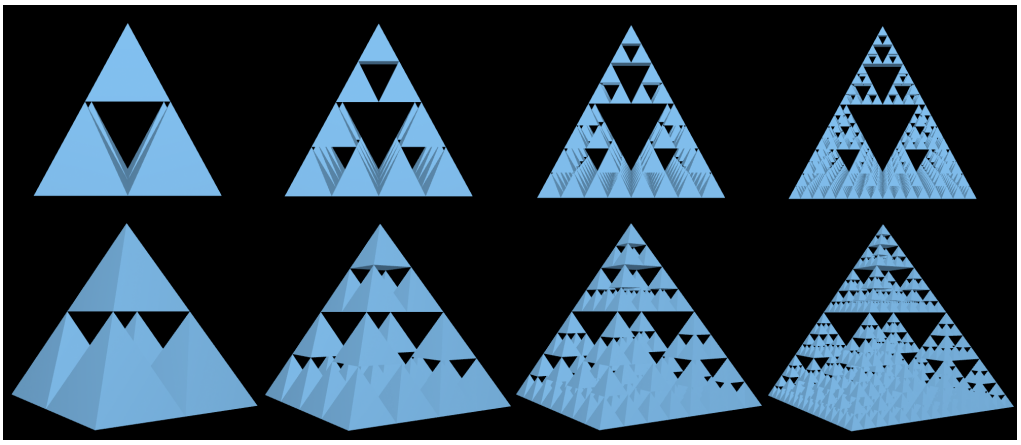
Nove začetne elemente in navodila lahko zgradimo hitro in enostavno z razširitvijo razreda *MRCMModel3D*. Ločitev med navodili in začetnim elementom se naredi zgolj z izbiro ustreznega paketa razširitve, kar omogoča tudi avtomatizirano posodabljanje uporabniškega vmesnika. V razširitvi moramo zgolj še definirati metodi *setScalingFactorInConstructor*, ki določi faktor skalacije med koraki, in *instructionsForMRCM*, ki pri začetnem elementu hrani grafično podobo, pri navodilih pa transformacije in izrise gradnika.

2.3.5 Primeri rezultatov

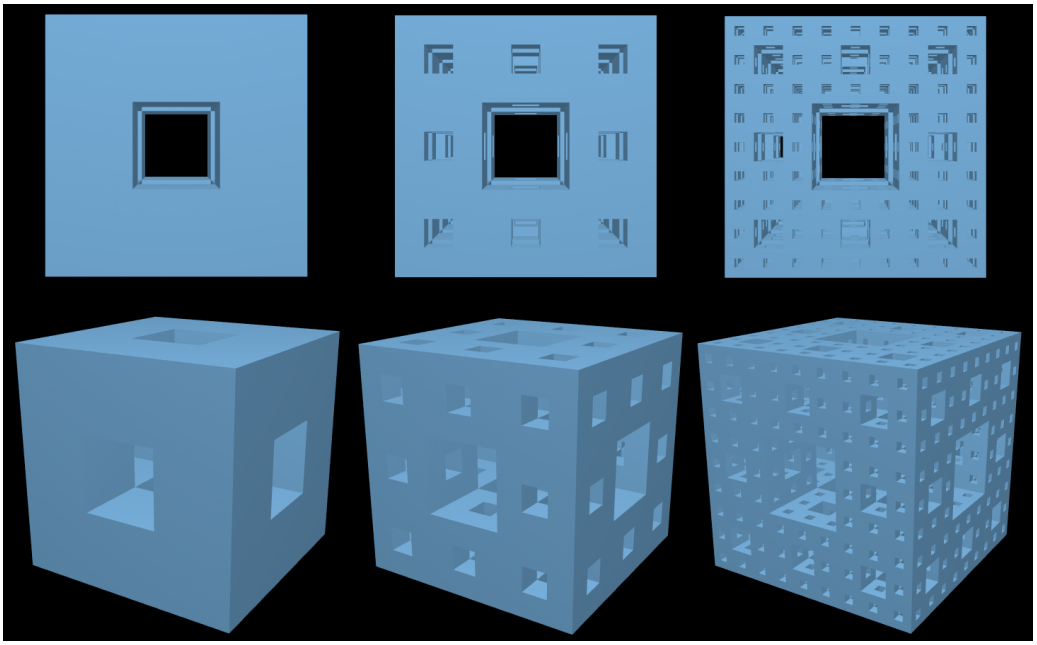
Z ločevanjem začetnih elementov in navodil pridobimo široki nabor možnih modelov, ki nam ga MRCM omogoča. Če smo natančni, nam omogoča $n = steviloNavodil * steviloZacetnihElementov$ kombinacij, ki z dovolj velikim številom korakov preide v množico atraktorjev, ki jih je toliko, kolikor je različnih navodil. Trenutni nabor implementacije nam omogoča 18 različnih kombinacij, od katerih so v nadaljevanju prikazane slike šestih. Slika 2.8 predstavlja dvodimenzionalen fraktal, ki kljub izbiri kocke za začetni element preide do atraktorja trikotniške oblike, kar dokazuje moč atraktorja. Če bi tu za začetni element izbrali piramido, se bi zgolj zmanjšalo število potrebnih korakov kot v primeru slike 2.9. Preostale slike od 2.9 do 2.13 predstavljajo različne tridimenzionalne fraktale. Ker so prikazani fraktali definirani s sprednjo ploskvijo vzporedno z zaslonom, so v slikah od 2.9 do 2.12 v dodatni vrstici prikazane iteracije fraktalov pod drugačno pozicijo pogleda za bolj tridimenzionalen občutek. Slika 2.13 moč tridimenzionalnih fraktalov še bolje odrazi s pomočjo mavrične kocke, ki z barvnimi ploskvami jasno prikaže rotacije prisotne znotraj fraktala.



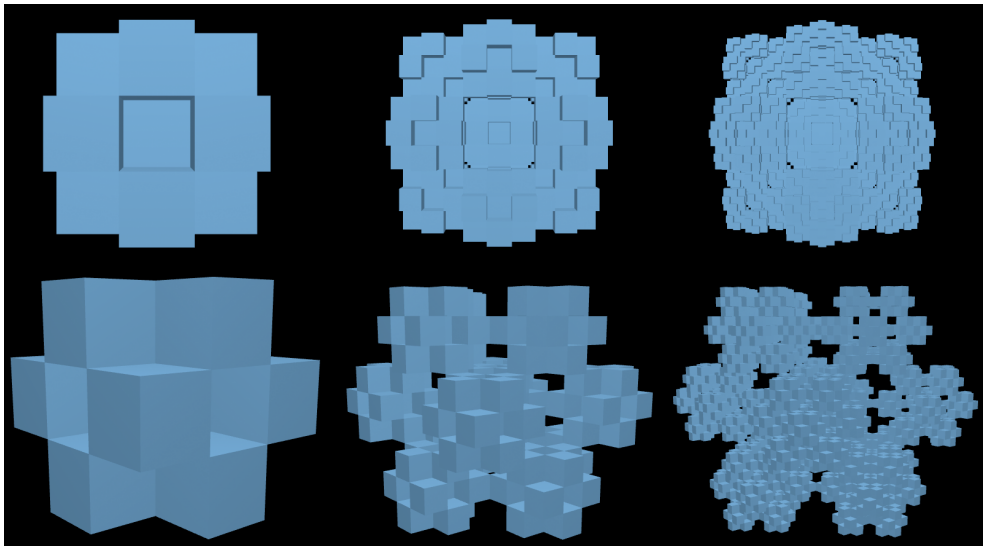
Slika 2.8: Izvedba navodil Sierpinski_Triangle s kocko za prvih osem korakov. Koraki si sledijo od leve proti desni, od zgoraj navzdol.



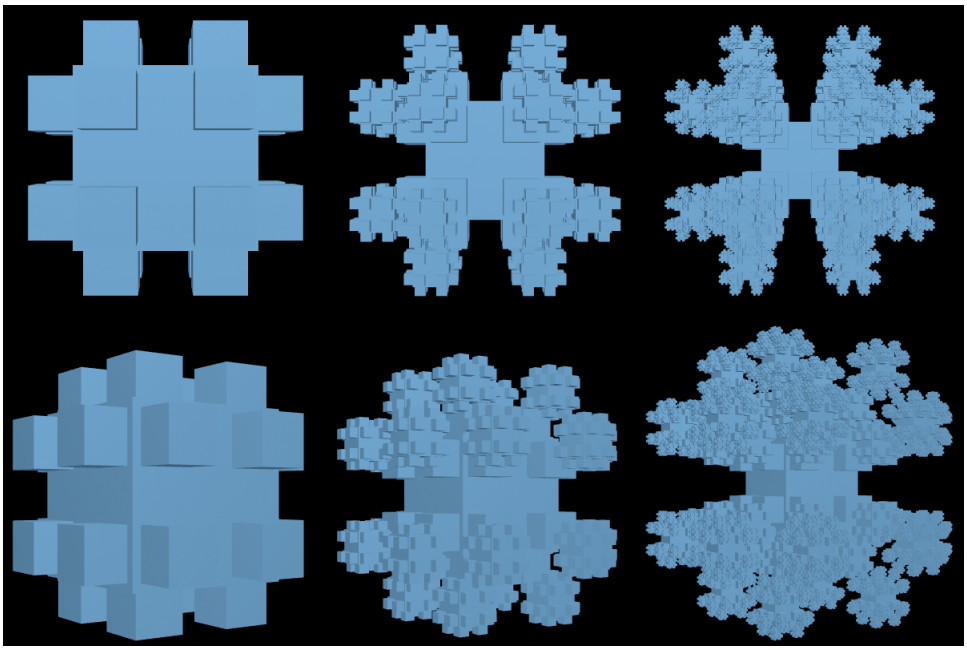
Slika 2.9: Kombinacija navodil Sierpinski_Pyramid in piramide za prve štiri korake. Koraki si sledijo od leve proti desni.



Slika 2.10: Kombinacija navodil Menger_Sponge in kocke za prve tri korake. Koraki si sledijo od leve proti desni.



Slika 2.11: Kombinacija navodil Koch_Cube in kocke za prve tri korake. Koraki si sledijo od leve proti desni.



Slika 2.12: Kombinacija navodil Corner_Cube in kocke za prve tri korake. Koraki si sledijo od leve proti desni.

Slika 2.13: Izvedba navodil Word_Test z mavrično kocko za prve tri korake. Koraki si sledijo od zgoraj navzdol, pri čemer je zadnja slika povečava črke T v tretjem koraku za bolj podroben pogled.

Poglavje 3

L-sistem

3.1 Splošno

L-sistem je okrajšava za Lindenmayer sistem, ki ga je razvil Aristid Lindenmayer za potrebe teoretičnega zapisa struktur rastlin. Sčasoma se je koncept prenesel tudi na področje računalniške grafike za simulacijo rastlin in drugih predmetov[6]. Klasična izvedba L-sistema je DOL-sistem oz. *deterministični kontekstno-neodvisni L-sistem*. Sestavljajo ga gramatika $G = \langle V, S, \omega, P \rangle$, generator nizov in generator grafike iz danega niza, kjer se običajno uporablja princip *LOGO želve*. Gramatiko G sestavljajo množica spremenljivk V, množica konstant S, začetni aksiom ω, ki definira začetni niz generatorju, in množica pravil za ustvarjanje nizov P. Generator nizov se vsako iteracijo sprehodi skozi celotni dani niz in spremenljivke po pravilih ustrezno zamenja. Pri tem posamezno spremenljivko obravnava neodvisno od preostalega niza, zato govorimo o kontekstni neodvisnosti. Sistem je deterministično, ker ob nespremenjenem začetnem aksiomu in istem številu iteracij vselej generiramo enake rezulate[7][8]. Šolski primer gramatike je gramatika za trikotnik Sierpinskega

$$G_{Sierpinski} = \langle \{A, B\}, \{+, -\}, A, \{A \to B - A - B, B \to A + B + A\} \rangle \quad (3.1)$$

kjer spremenljivki predstavljata enako črto, + in - pa rotacijo naslednje črte za 60° oz. -60°. Generator nizov tako prične prvo iteracijo z aksiomom A, ki

ga razširi v B-A-B. Skozi nadaljne iteracije(n) izdelamo sledeče nize in izrise:

n = 2

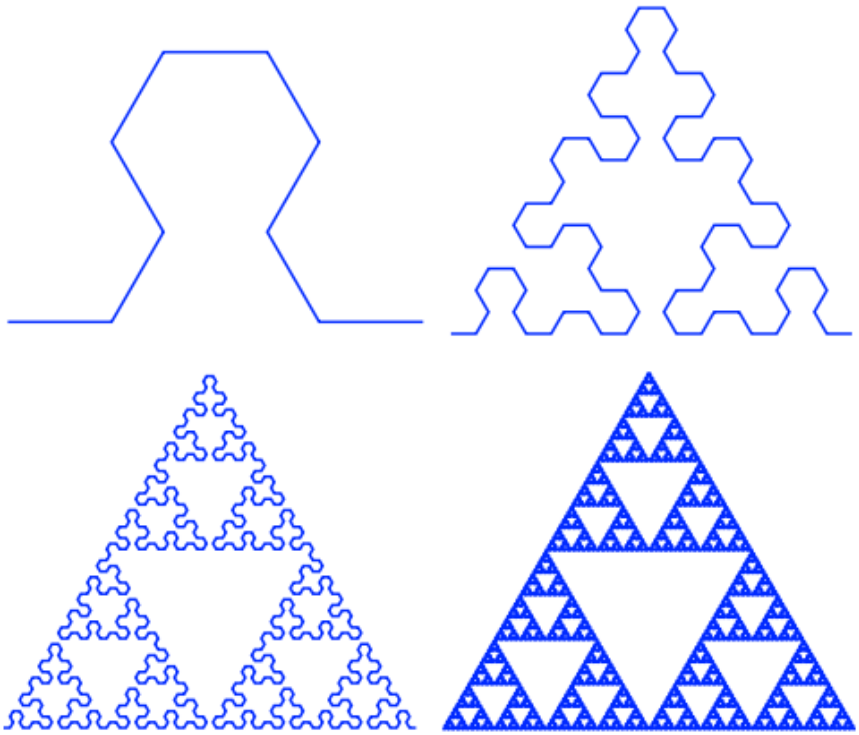
A+B+A-B-A-B-A+B+A

n = 3

B-A-B+A+B+A+B-A-B-A+B+A-B-A-B-A+B+A-B-A-B+A+B+A+B-A-B

n = 4

A-B-A+B+A+B+A-B-A-B+A+B-A-B-A-B+A+B-A-B-A+B+A+B+A-B-A+B+A+B-A-B-A-B+A+B+A-B-A+B+A+B+A-B-A+B+A+B-A-B-A-B+A+B+A-B-A-B+A+B-A-B-A-B+A+B-A-B-A+B+A+B+A-B-A



Slika 3.1: Trikotnika Sierpinskega pri n = 2, 4, 6 in 8. Izrisi si sledijo od leve proti desni, pri čemer so zaradi preglednosti skalirani na enako velikost[9].

3.1.1 Izris in želva

Največji izziv izrisa predmeta iz L-sistema, je pravilna pretvorba zaporedja spremenljivk in konstant enostavnega niza v pravilno postavljene, skalirane in rotirane kompleksnejše grafične predmete. Ravno zaradi tega problema so bili L-sistemi na začetku uporabljeni zgolj za opis topologije vejitev, nakar so geometrijske lastnosti kot dolžine črt ipd. določili po procesiranju niza. Do spremembe je prišlo, ko so v L-sistem uvedli fraktale, glej poglavje 2. To je omogočilo uporabo enostavnih grafičnih elementov kot gradnike kompleksnejšega predmeta, pri čemer se je njihova transformacija izvedla že tekom procesiranja niza. Kot dober pristop za pravilno in hitro izvedbo transformacij tekom procesiranja niza se je uveljavila *LOGO želva*[7].

Želvo si moramo predstavljati kot bitje, ki živi v matematični ravnini oz. prostoru. Sprejemati zna enostavne ukaze kot so premik naprej, izris elementa, obrat v levo itd. Za standarne ukaze želve pri L-sistemih si oglejte podpoglavje 3.1.2. Ukaz naprej tako spremeni pozicijo želve, obrat njeno usmeritev, izris elementa pa nas tipično postavi na konec izrisanega elementa. Želva mora za pravilno delovanje seveda hraniti trenutno stanje. Pozicijo želve lahko enostavno hranimo kot vektor koordinat v prostoru pod pogojem, da je želvino izhodišče tudi koordinatno izhodiče našega prostora. Za usmerjenost želve imamo na izbiro vektor kotov posamezne osi ali enotski vektor v smeri pogleda želve. Slednji pristop poenostavi premik želve, saj potrebujemo ob premiku zgolj sešteti za velikost premika skalirani enotski vektor in vektor pozicije[10]. V stanju želve se hrani tudi druge trenutne lastnosti, kot so barva in faktor skaliranja grafičnega elementa ipd. Lahko se hrani tudi referenco na stanje pred vejitvijo, kar pa lahko enostavno rešimo tudi s skladom želv. Želvo lahko razvijalec modela hitro prikroji danemu problemu, kar lahko poenostavi tako gramatiko kakor izris problema, brez večjih popravkov samega L-sistema. Ravno enostavnost koncepta in potencial enostavne razširitve glede na dani problem sta lastnosti želve, zaradi katerih je tako pogosto uporabljena v L-sistemih.

3.1.2 Standardna gramatika

Spodnji seznam simbolov se je s pomočjo knjige The Algorithmic Beauty of Plants[7] in njene pogoste citiranosti uveljavil kot standard za L-sistem. Standard se uporablja tudi v Sloveniji, kar je razvidno ob opazovanju del drugih avtorjev[11].

- F - pomeni premik želve z izrisom črte oz. drugega vnaprej definiranega osnovnega elementa. Element se izriše v skladu s podano dolžino in debelino v smeri premika želve.
- f - premik želve brez izrisa osnovnega elementa s podano dolžino in usmerjenostjo želve.
- [- začetek vejitev in posledično shranitev stanja želve.
-] - konec vejitve, želva se povrne v stanje pred vejitvijo.
- + - rotacija želve okoli Y-osi za podani kot.
- − - rotacija želve okoli Y-osi za podani kot v obratni smeri.
- & - rotacija želve okoli Z-osi za podani kot.
- ∧ - rotacija želve okoli Z-osi za podani kot v obratni smeri.
- / - rotacija želve okoli X-osi za podani kot.
- \ - rotacija želve okoli X-osi za podani kot v obratni smeri.
- | - obrat okoli Y-osi oz. rotacija okoli osi za kot 180°.
- $ - ponastavi vse rotacije želve v osnovno stanje, kar običajno pomeni nastavitev kotov na 0°, kar povrne želvo v vertikalno lego.
- ! - višina in debelina osnovnega elementa se skalirata s podanim faktorjem povečave oz. pomanjšave.

- ' - indeks izbrane barve osnovnega elementa se pomakne naprej po tabeli barv, s tem se doseže enostavno prelivanje barv.

Sama narava L-sistemov omogoča dodajanje lastnih elementov, vendar so standardni simboli v veliki večini primerov dovolj za željeni predmet.

3.2 Razširitve

L-sistem lahko prilagajamo preko želve, glej podpoglavje 3.1.1, vendar smo zaradi samih lastnosti DOL-sistema dokaj omejeni. Kot glavni pomanklji-vosti lahko imenujemo determinizem in redunantno količino osnovnih elementov. Determinizem škodi občutku realnosti izrisa, še posebej ko imamo predmet večkrat izrisan, saj ne dobimo nikakršnega občutka naključnosti, ki je tako pogost v naravi. Osnovni elementi so sicer elegantna rešitev izrisa modela, vendar se ob določenih primerih lahko začnejo po nepotrebnem kopičiti. Želeli bi npr. imeti na drevesu dolge veje, kar pri DOL-sistemu v nizu najdemo kakor FFFFF..., vendar je to redudantno, saj bomo pri izrisu izrisali isti element mnogokrat zapored, namesto da bi ga zgolj enkrat transformirali. Zaradi takšnih in drugačnih težav ter potreb se DOL-sistem uporablja zgolj kot osnova pri gradnji lastnega L-sistema, ki ga potem bodisi priredimo bodisi razširimo. V nadaljevanju so prikazane preostale pogoste oblike L-sistemov, ki so prav tako kompatibilne med seboj.

3.2.1 Stohastični L-sistem

Reševanje determinizma se prične z idejo naključnosti. Naključje lahko vstavimo v želvo ali v sam L-sistem ali kar v oboje. Naključje znotraj želve ima omejen vpliv, saj pri njej običajno zgolj spreminjamo mere, rotacije in preostale lastnosti grafičnih elementov. Sama topologija modela se ne spremeni, navkljub spremembam v geometriji. Naključnost L-sistema samega je boljša rešitev, saj z njo spremenimo tako topologijo kakor geometrijo[7].

Stohastičen L-sistem se pridobi enostavno z uvedbo množice pravil gradnje nizov, ki imajo skupno spremenljivko kot sprožilec. Pri DOL-sistemu bi se upoštevalo zgolj prvo pravilo, ostale pa bi preprosto ignorirali. Vendar moramo za vsako pravilo sedaj vedeti še verjetnosti zanj. Posledično se oblika gramatike spremeni iz $G = \langle V, S, \omega, P \rangle$ v $G = \langle V, S, \omega, P, \pi \rangle$, kjer π predstavlja množico verjetnosti za posamezna pravila v P[7]. Samo generiranje naključnosti in izbira pravila je prepuščena generatorju nizov, ki je temu ustrezno prirejen. Pri gramatiki je potrebno zgolj še paziti, da je vsota verjetnosti pravil z isto začetno spremenljivko enaka 1. Za primer uporabimo gramatiko za trikotnik Sierpinskega, glej zapis 3.1, in jo rahlo spremenimo tako, da imamo dodatno pravilo ob pojavu spremenljivke A. Poleg $A \to B - A - B$ dodamo še pravilo $A \to B$, ki naj bo manj verjetno od osnovnega. Zatorej ima manjšo verjetnost kakor drugo pravilo, hkrati pa pazimo, da njuna vsota ostane 1.

$$\begin{aligned} G_{SierpinskiRandom} = \langle \{A, B\}, \{+, -\}, A, \\ \{(A \to B, A \to B - A - B), (B \to A + B + A)\}, \{(0.3, 0.7), (1)\} \rangle \end{aligned} \tag{3.2}$$

S takim pristopom in ustreznimi verjetnostmi lahko zagotovimo, da bodo naši generirani modeli podobni med seboj, vendar stalno različni. Že gramatika 3.2 ima pri ustrezni količini iteracij neprepoznavne rezultate v primerjavi s šolskim primerom 3.1.

3.2.2 Kontekstno odvisni L-sistem

Omenili smo že, da občasno pridemo do situacij, ko želimo pravila gradnje prožiti ne samo glede na spremenljivko, temveč tudi glede na njegovo okolico oz. kontekst. Tedaj začnemo govoriti o kontekstno odvisnih L-sistemih oz. IL-sistemih. Te lahko nadalje ločimo na 1L-sisteme in 2L-sisteme, glede na opazovano okolico spremenljivke, ki jo tu imenujemo strogi prednik. Prvi preverjajo kontekst pred ali po strogem predniku, drugi opazujejo celotno okolico[7]. Vzemimo primer dolge veje iz 3.2, kjer smo rekli, da se želimo ogniti situacijam FFFFF... Recimo, da ob obstoju dovolj velikega pojava F-

jev vstavimo D, ki določa dolgo vejo oz. vstavimo C, ki zgolj zamenja F in nima pomena. Sledeča pravila produkcije nizov so možen način reševanja problema, vendar bi morala biti za optimalni način bolje definirana. Pravila so izbrana zaradi nazornosti.

$$FF < F \rightarrow D \tag{3.3}$$

$$F < CD \rightarrow C \tag{3.4}$$

$$FF < F < D \rightarrow C \tag{3.5}$$

Produkcijsko pravilo 3.3 se sproži, ko se v levi okolici od strogega prednika F nahajata 2 F-ja. Tedaj se bo strogi prednik zamenjal z D. Drugi primer pravila 1L-sistema je 3.4, kjer se opazuje desni kontekst in če je ta enak CD, se strogi prednik F zamenja s C. Primer 3.5 je pravilo 2L-sistema, kjer se strogi prednik F zamenja s C zgolj, če sta pred njim 2 F-ja in za njim D.

Iz danih primerov se lepo vidi tudi, da konteksta IL-sistema ne potrebujeta biti enake dolžine. Zato tak sistem pogosto imenujemo tudi (k,l)-sistem, pri čemer k ponazarja dolžino levega konteksta in l dolžino desnega konteksta. IL-sistem uporabljamo seveda v povezavi s kontekstno neodvisnim sistemom, saj moramo v večini primerov najprej zgenerirati kontekst, preden ga lahko preverjamo. Tedaj velja, da imajo kontekstno odvisna pravila prednost pred kontekstno neodvisnimi. Opozoriti je potrebno zgolj še na preverjanje konteksta pri vejitvah. V takih primerih je potrebno preveriti kontekste v vseh smereh vejitev, kar pomeni, da mora generator ustrezno ignorirati dele konteksta in preveriti vse veje[7].

3.2.3 Parametrični L-sistem

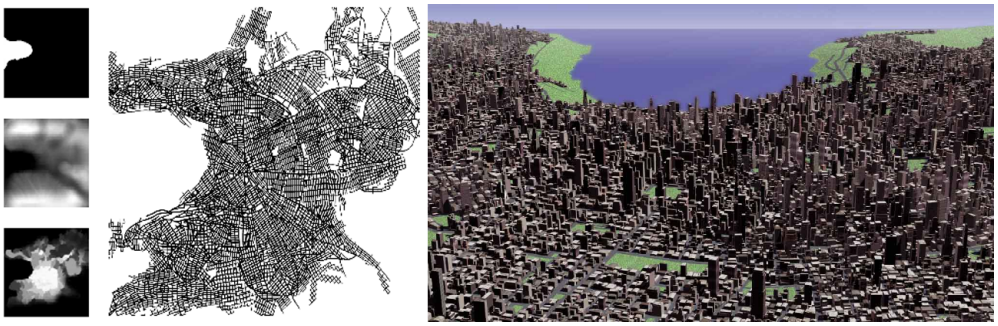
Parametrični L-sistemi dodajo v nize modelov poleg simbolov še numerične vrednosti, ki definirajo obnašanje ob izrisu. Uvedli so jih ravno zaradi redudantnega nizanja elementov, kar kvari matematično lepoto L-sistema. V nizu se poleg simbola v oklepaju zapiše numerična vrednost, ki definira trenutno stanje simbola. Nad numeričnimi vrednostmi lahko generator poljubno izvaja aritmetične operacije, če jih te pravila zahtevajo. Sama pravila so tedaj

zapisana kot *prednik(vrednost) : pogoj* $\rightarrow$ *rezultat*[7]. Poskusimo zopet rešiti problem dolgih vej iz 3.2, pri tem pa dodajmo za lepši prikaz še problem ožanje debeline vej z dolžino. Recimo, da ob vsakem branju F-ja povečamo vrednost dolžine ter ob vsakem trikratnem povečanju dolžine zmanjšamo debelino na polovico. Privzemimo, da vrednost za debelino predstavlja končno debelino in da se začetna debelina ne spreminja. Tedaj bi imeli sledeči dve pravili:

$$F(dol, deb) : dol \mod 3 == 0 \rightarrow F(dol + 1, deb/2) \tag{3.6}$$

$$F(dol, deb) : dol \mod 3 > 0 \rightarrow F(dol + 1, deb) \tag{3.7}$$

Pravili lepo rešita problem spreminjanja lastnosti veje brez kupičenja elementov. Vendar je potrebno paziti pri ustvarjanju generatorja, da bo leta znal ločiti med simboli in aritmetičnimi operatorji, saj je teh že mnogo v standardni gramatiki, glej 3.1.2. Bolj izrazit prikaz moči parametrov v L-sistemih je okolje *CityEngine* iz poglavja 4.3.1[12], ki poleg drugih tehnik v dveh kritičnih fazah uporablja ravno parametrične L-sisteme. V začetni fazi s parametričnim L-sistemom zgradi na ponujeni površini kompleksno omrežje ulic, ki se prilagaja tudi prednastavljeni gostoti prebivalstva. Ko je začetna faza končana in so parcele med ulicami razdeljene na kose, zgradi drugi parametrični L-sistem stavbe za vse parcele glede na dane podatke.



Slika 3.2: Primer mesta generiranega s CityEngine, ki vsebuje približno 26.000 stavb. Leve tri višinske slike, ki določajo kopno, relief in gostoto prebivalstva so vhodni podatki. Srednja slika je zemljevid ulic, ki je bil generiran v začetni fazi. Desna slika prikazuje na zemljevid generirane stvabe[12].

Pri parametrični L-sistemih je potrebno omeniti še funkcijske L-sisteme oz. FL-sisteme, ki razširijo koncept parametrov iz numeričnih vrednosti na objekte. Pri njih je lahko parameter numerična vrednost, klic funkcije ali referenca na neki objekt. Funkcijska narava ni omejena samo na parametre, funkcije namreč lahko dodajamo v sama pravila in kot končna vozlišča v drevesu niza[13]. Prednosti takšnega sistema so bolj programerski pogled na pravila, vgradnja že pripravljenih objektov v sam model, hranjenje vrednosti v objektih namesto seznama parametrov in še mnogo več. Kot primer lahko navedemo sledeče pravilo, ki uporablja zanko.

$$A(n) \rightarrow for(i = 0; i <= n; i++)B \tag{3.8}$$

Z uvedbo funkcij translacij, rotacij in skalacij lahko ustvarimo celo L-sistem, ki ne potrebuje želve. Potrebno je zgolj prenašati lokalni kartezični koordinatni sistem v parametrih in na njem izvajati nove funkcije. Tudi vejitve niso problem, če dodamo funkcijo, ki kopira trenutni koordinatni sistem[13].

3.2.4 Upoštevanje okolja

Pogost izziv pri L-sistemih je tudi upoštevanje okolja pri gradnji modela. Klasični primeri problemov so rastline, izpostavljene vremenskim razmeram, rastline, ki rastejo ob predmetih, in vodena gradnja modela, ki ga želimo zgraditi znotraj nekih meja.

Upoštevanje vremenskih razmer kot so močan veter, težje padavine ipd. dosežemo s ponovnim procesiranjem že dobljenega modela. Osnova za to mora biti dober model, ki simulira vremenske razmere in jih sporoča preko vektorjev moči. Ko L-sistem izdela osnovni model brez vplivov vremena, se za posamezne elemente modela izračuna vektorje moči vremena zanj. Glede na vektor moči in faktor debeline, ki predstavlja uporno moč elementa, se izračuna zasuk elementa. Nakar se zasuk s transformacijskimi matrikami še realizira, vendar moramo paziti na pravilen zasuk elementov, ki sledijo zasučenemu elementu. Tako se v drugem procesiranju sekvenčno suče element za elementom, dokler niso doseženi vsi listi drevesa elementov[14].

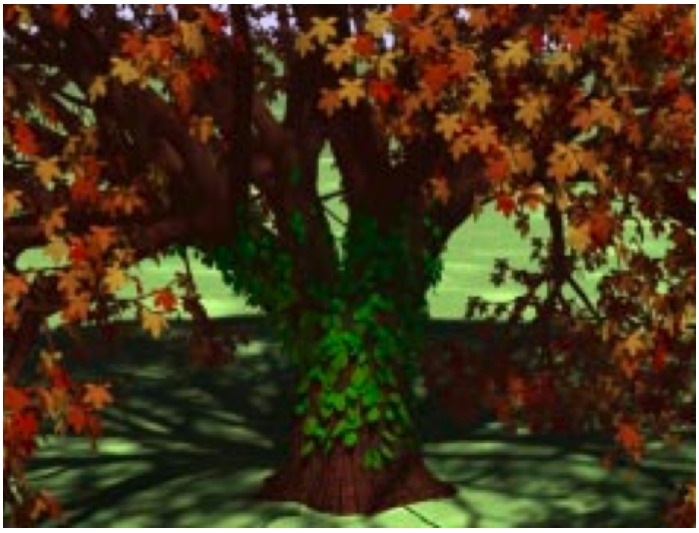
Omejevanje smeri gradnje za preostale probleme lahko dosežemo z vodiči, ki so pravzaprav zaprti liki, ki določajo mejo, preko katere model ne sme seči. Kot vodiče lahko uporabimo tudi dele drugega L-sistema in obratno, s čimer dosežemo komunikacijo med dvema samostojnima L-sistemoma. Običajno se element modela odstriže bodisi v celoti bodisi deloma, kadar je njegov center na vodiču ali preko vodiča. L-sistem ugotavlja stanje elementov s t.i. povpraševalnimi funkcijami, ki jih nudi vodič. Tipični primeri povpraševanj so: ali obstaja trk, kolikšna je razdalja do vodiča in kolikšna bo razdalja po premiku želve[15]. Komunikacija s svetom izven L-sistema je možna zgolj, če jo L-sistem začne, za kar potrebuje komunikacijski simbol znotraj gramatike, ki kliče povpraševalne funkcije. Standarden način označitve povpraševanja je ? z želenim povpaševanjem in parametri, npr. $?E(x_1, x_2, \ldots, x_n)$. Ker poteka komunikacija med svetom znotraj L-sistema in zunaj njega, imenujemo tak pristop odprti L-sistem[16]. V pravilih bi lahko povpraševanje tvorili na sledeč način.

$$?E(x) < A(v) : x == 1 \rightarrow [+(\alpha_2)F(v * r_2)?E(r_2)A(v * r_2)]\text{–}(\alpha_2) \tag{3.9}$$

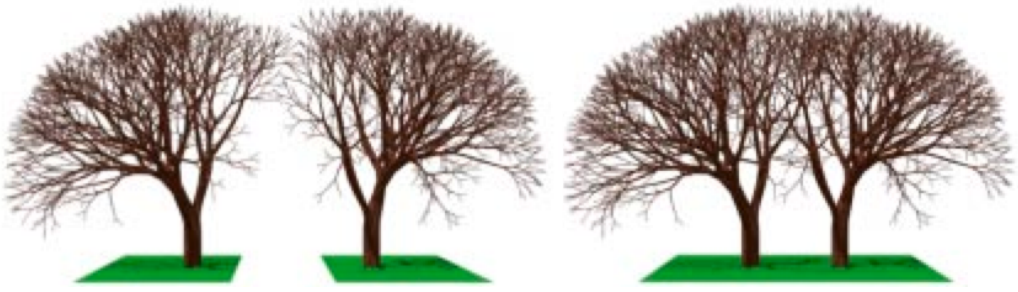
$$?E(x) \rightarrow \epsilon \tag{3.10}$$

Primer pravil s povpraševanjem je povzet iz [16] in ugotavlja, če se sme ustvariti vejitev v željeno smer, se pravi, če ne pride do trka.

Slika 3.3: Primer stiskanja modela v željeno obliko dinozavra. Leva slika predstavlja podobo vodiča, desna pa z vodičem omejeni končni model[15].



Slika 3.4: Primer L-sistema ovijalke, ki pleza po danem modelu drevesa[16].



Slika 3.5: Primer komunikacije med dvema L-sistemoma, ki ju po izgradnji postavimo v desni sliki skupaj[16].

3.3 Implementacija LSPS

Lastno implementacijo L-sistema sem omejil na sistem, prirejen za izgradnjo rastlin, od tod tudi ime *L-System Plant Simulator* oz. na kratko LSPS. LSPS omogoča enostavno dodajanje novih gramatik preko dedovanja in njihovo grafično realizacijo. LSPS sprejema tako deterministične kakor tudi stohastične gramatike, vendar je zaradi učne narave diplomske naloge sistem kontekstno-neodvisen, neparametričen in neodvisen od okolja. Dodatna funkcionalnost LSPS je simulacija gozda, ki iz danih gramatik naključno razporedi naključno generirano rastlinje, kar lepo predstavi pomen naključja za višjo raven realnosti. Delovanje je predstavljeno v nadaljevanju preko štirih paketov, ki tvorijo implementacijo: *lsps.graphics, lsps.gui, lsps.lsystem in lsps.plants*.

3.3.1 Odstopanja od standarda

Pri implementaciji standarnih simbolov, glej poglavje 3.1.2, so se pomeni simbolov rahlo spremenili. Zaradi večje priročnosti, se vsaki osi določi ustrezne rotacije s parametri *rotationOnX, rotationOnY* in *rotationOnZ*, kar poenostavi pravila gradnje in jih ravno tako naredi bolj dinamična. Kot geometrijski pomen F-ja sem definiral deblo, ki je predstavljeno kot pravilna n-kotna prizma. Število kotov prizme določimo preko grafičnega vmesnika, s čimer lahko uporabnik enostavno definira časovno zahtevnost in realnost izrisa. Preko uporabniškega vmesnika lahko uporabnik določi tudi globino iteracije, želeno gramatiko in začetne mere debla. Za deblo sem privzel, da se njegova dolžina in debelina skalirata linearno, saj je tako potrebno zgolj podati *logDecrementPercentage*. To velja za obe lastnosti debla, pri čemer ni učinek debla nič manj realen. Zaradi programskega okolja sem prav tako naletel na težavo s simbolom ', ki sem ga bil primoran spremeniti v ?, pri čemer se je pomen simbola ohranil. Pri gramatikah, ki vključujejo liste, sem le-te definiral kot nove simbole in jih geometrijsko predstavil kot prosojne deltoide, ki podobno kot debla upoštevajo stanja želve. Pravilen izris lista

je definiran že znotraj *PlantModel*, pri definiciji simbola za list je potrebno zgolj poklicati metodo *branchEnd*.

3.3.2 lsps.graphics

Vsebuje osnovne grafične elemente za prikaz gramatik, med drugim tudi razred želve Turtle. Pomembna razreda sta tudi PlantModel, ki ga morajo razširiti vsi modeli za prikaz gramatik, in PlantVisualizator, ki je pravzaprav grafični pogon za končni izris modelov. Pri grafičnih elementih sem tudi uporabil teksture, zlivanje za prosojnost, kar se vidi pri listih, in sezname za izris vsebin za hitrejši izris. Seznami so pravzaprav kar seznami seznamov, saj se je tekom razvoja razvilo veliko elementov, ki jih je bilo potrebno različno globoko gnezditi.

3.3.3 lsps.lsystem

Predstavlja srce LSPS-a, saj hrani osnovni razred za gramatike *Grammar* in razred *Generator*, ki zna generirati besede glede na vstavljeno gramatiko. Pomožni razred *Distribution* pa omogoča hiter vpis verjetnostnih distribucij pri stohastičnih gramatikah. Sistem tudi skrbi za osnovno preverjanje ponujenih gramatik. Pregleda namreč, če so aksiom in pravila v skladu s podanimi spremenljivkami in konstantami. Prav tako znotraj generiranja javi napako, če odkrije napako v verjetnostnih distribucijah.

3.3.4 lsps.plants

Sam *lsps.plants* se deli še na paket z gramatikami *lsps.plants.grammars* in paket z modeli *lsps.plants.models*. Paket z gramatikami vsebuje v implementirane gramatike, ki razširjajo razred *Grammar* in imajo svojo grafično podobo shranjeno v *lsps.plants.models*. Če se ime gramatike konča z *Random*, to nakazuje, da je bila gramatika razširjena iz deterministične v stohastično. Random razred uporablja isti razred modela kot deterministični razred, kar pomeni, da sistem pri povezavanju gramatik z geometrijami ignorira pripono

Random. Modeli grafičnega prikaza gramatik vsi razširjajo razred *PlantModel*. Trenutna zbirka gramatik je sledeča:

- *Grm* - deterministični grm.
- *GrmRandom* - stohastični grm.
- *TrikotnikSierpinskega* - trikotnik Sierpinskega po zapisu 3.1.
- *TrikotnikSierpinskegaRandom* - predlagana vpeljava naključnosti v trikotnik Sierpinskega v zapisu 3.2.
- *Willow* - gramatika za dvodimenzionalno vrbo[17].
- *WillowRandom* - vpeljava naključnosti in tridimenzionalnosti v gramatiko Willow.

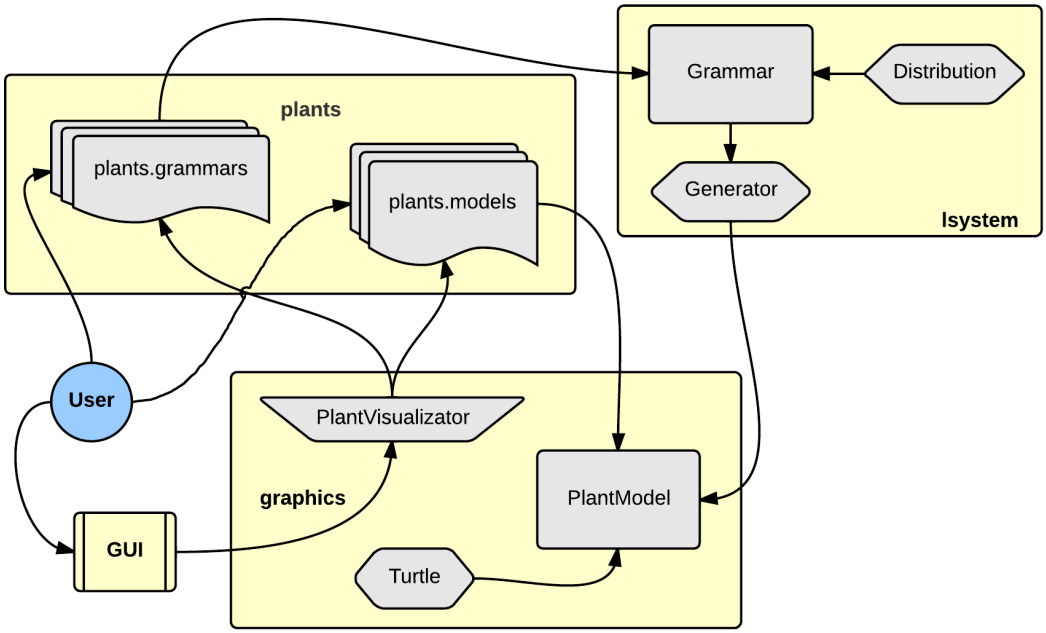
3.3.5 lsps.gui

Predstavlja uporabniški vmesnik, ki omogoča uporabniku enostavno izbiro med simuliranjem posamezne gramatike in simulacijo gozda. Simulacija posamezne gramatike omogoča izris gramatik, ki so shranjene znotraj *lsps.plants.grammars* z uporabniškimi parametri. Uporabnik lahko nastavi število iteracij, gladkost debla in začetno višino ter debelino debla. Simulacija gozda uporablja fiksno določene gramatike z naključnimi postavitvami in iteracijami.

3.3.6 Definiranje novih rastlin

Za definiranje nove rastline je zgolj potrebno razširiti razred *Grammar* tako, da v konstruktorju definiramo dodatne simbole, pravila, morebitne verjetnostne distribucije pravil in začetni aksiom. Nato je potrebno razširiti še razred *PlantModel*, kjer definiramo geometrijski pomen novih simbolov in nastavimo začetne parametre, lahko pa tudi popravimo pomene standardnih simbolov in obnašanje želve. Razširitev razreda *PlantModel* se mora imenovati tako kakor

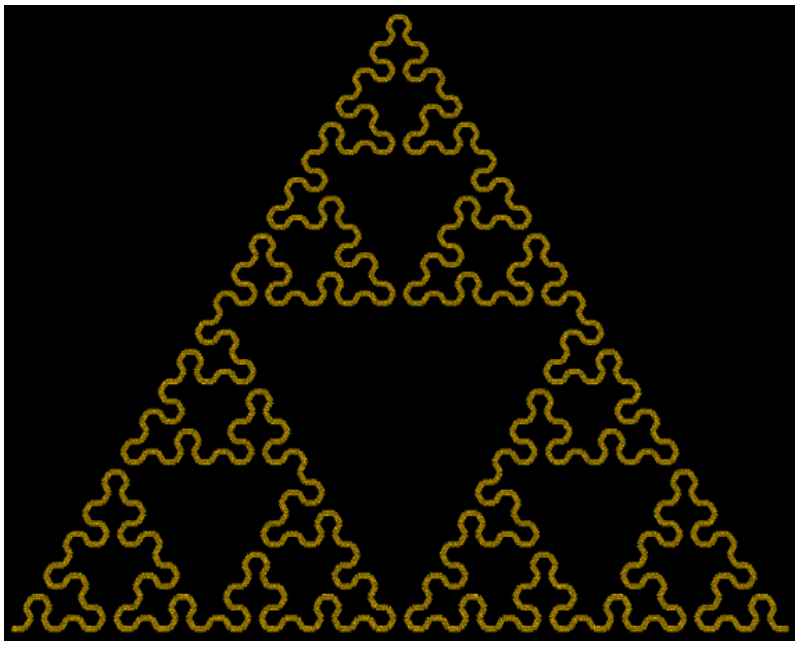
razširitev razreda *Grammar* s pripono *Model*, npr. *DrevoModel*. Omejitev poimenovanja razredov poenostavi povezavo med gramatiko in geometrijskim pomenom do te mere, da je potrebno razširitvi zgolj vstaviti v ustrezni mapi in sta ob zagonu LSPS že na voljo.



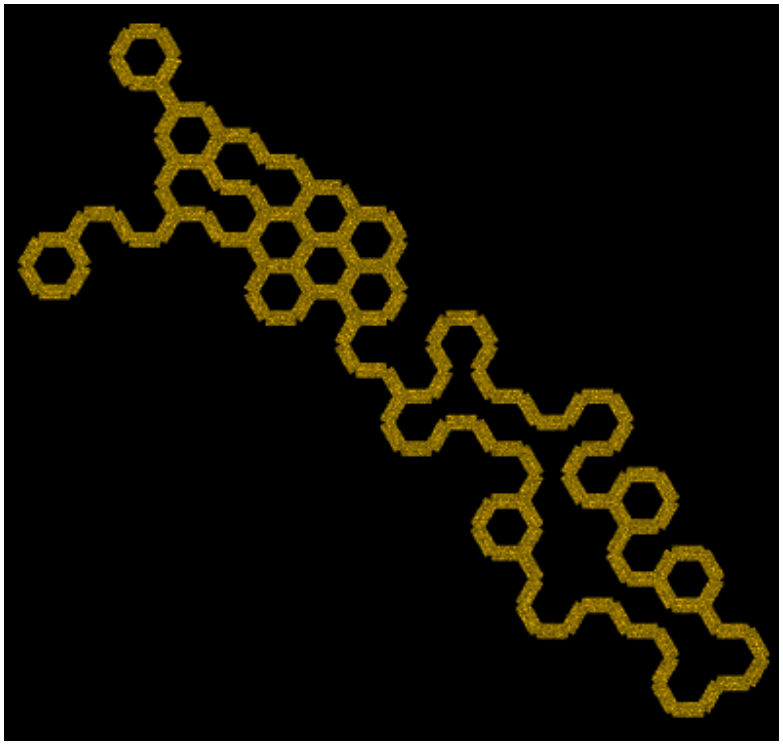
Slika 3.6: Arhitektura LSPSja.

3.3.7 Primeri rezultatov

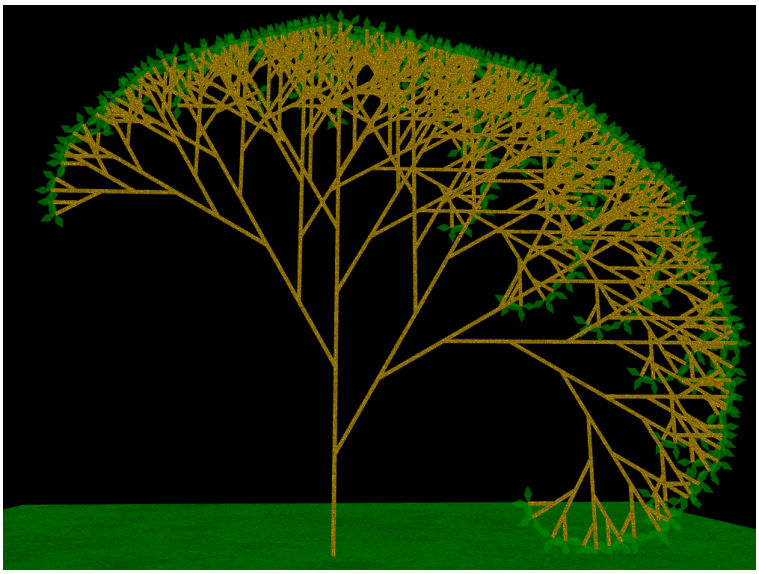
Začetna gramatika je bila za trikotnik Sierpinskega, saj predstavlja šolski primer vloge fraktalov znotraj L-sistemov. Čeprav LSPS avtomatično doda globino vrbi gramatike Willow z deblom, lahko o trodimenzionalnem rastlinju govorimo šele pri gramatikah WillowRandom, Grm in GrmRandom.



Slika 3.7: Primer gramatike TrikotnikSierpinskega s 6 iteracijami.

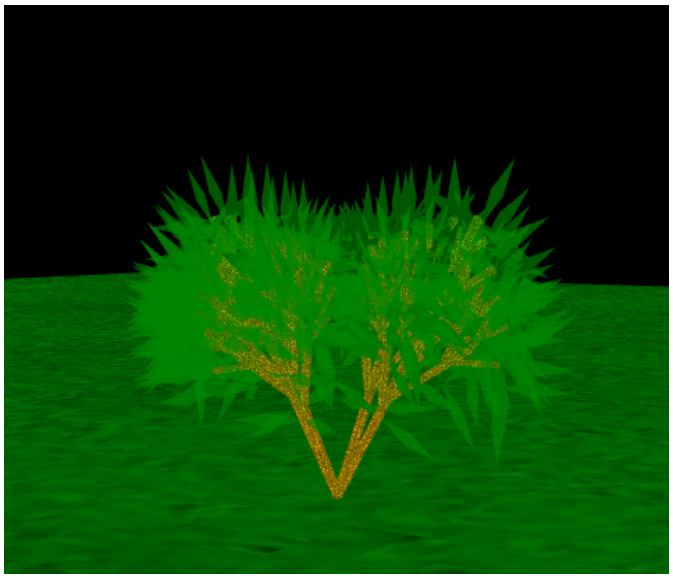


Slika 3.8: Primer gramatike TrikotnikSierpinskegaRandom s 6 iteracijami.

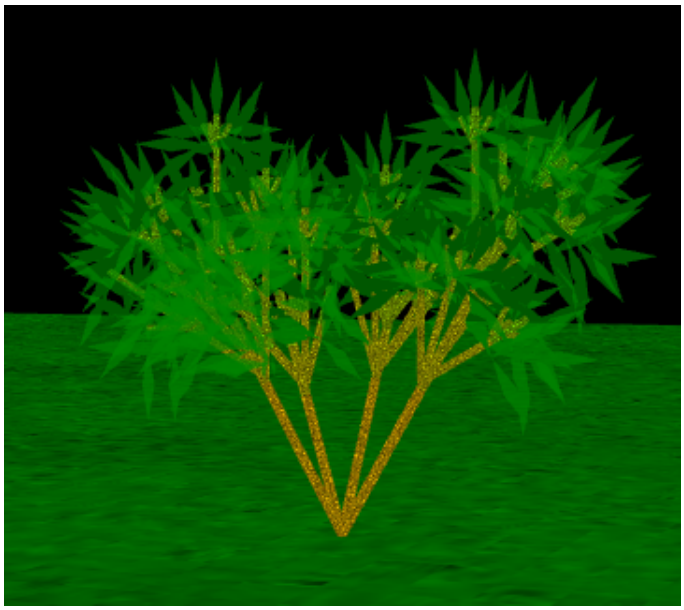


Slika 3.9: Primer gramatike Willow z 9 iteracijami.

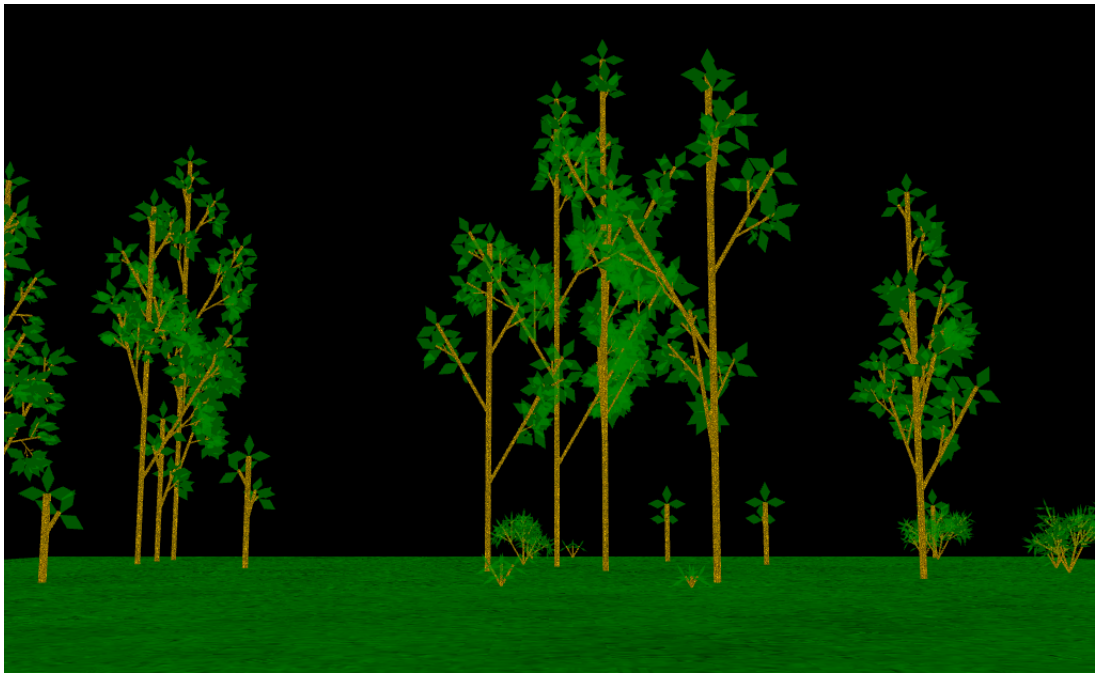
Slika 3.10: Primer gramatike WillowRandom z 9 iteracijami.



Slika 3.11: Primer gramatike Grm s 6 iteracijami.



Slika 3.12: Primer gramatike GrmRandom s 6 iteracijami pri enakih parametrih kot primer gramatike Grm.



Slika 3.13: Primer simulacije gozda.

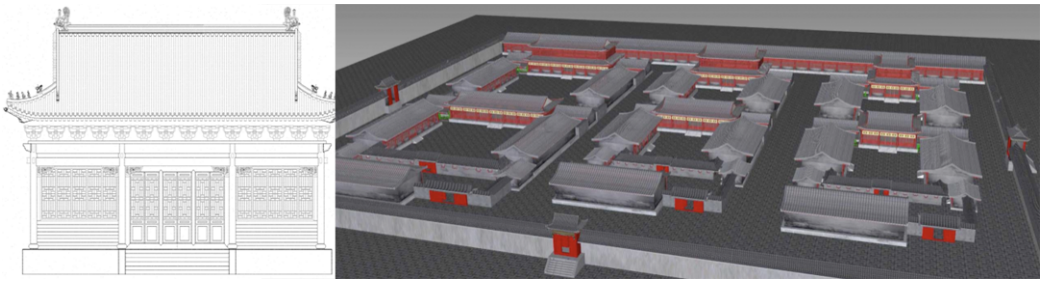
Poglavje 4

Gramatika oblik

4.1 Splošno

Proceduralno modeliranje z *gramatiko oblik* (ang. Shape Grammar - v nadaljevanju krajšajmo s SG) nas zaradi opravka z gramatikami lahko spominja na L-sistem, vendar gre za različna pristopa. Pri L-sistemu generiramo z gramatiko znakovne nize, iz katerih nato gradimo geometrijsko podobo, medtem ko SG že v gramatiki vsebuje geometrijske elemente. Z geometrijskimi elementi, ki jih imenujemo oblike, namreč tvorimo samo abecedo gramatike in tako skozi iteracije generiranja neposredno gradimo končno podobo. Idejo SG sta uvedla George Stiny in James Gips pod motom "oblikovanje je računanje", ko sta raziskovala vpeljavo logike v vizualni izgled. Od njunih začetnih treh gramatik se je področje uporabe SG močno razširilo in ustvarilo razne podgramatike, ki so prirejene svojim področjem uporabe. Primeri področij so analiza slik, izdelava grafov, reševanje inženirskih problemov, kot npr. razporeditev napeljave v letalu, ter računalniška grafika, kjer je pogosto uporabljena v sklopu modeliranja arhitekture in skulptur[18][19][20].

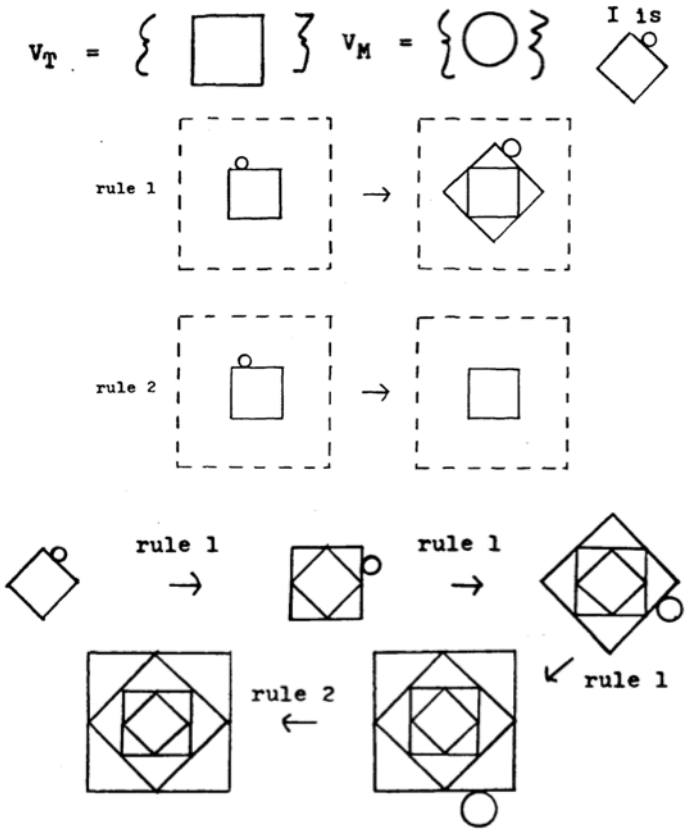
Po definiciji $SG = \langle V_t, V_m, R, I \rangle$ so gradniki SG-ja množici oblik V_t in V_m, množica parov R in začetna oblika I. Znotraj V_t definiramo terminalne oblike oz. končne elemente, medtem ko pri V_m zapišemo neterminalne oblike oz. markerje. Ker markerji predstavljajo oblike, ki prožijo nadaljno generiranje



Slika 4.1: Primer uporabe SG za generiranje tradicionalne kitajske arhitekture s prepoznavanjem pravil gradnje glede na nekaj stranskih risov. Levo je primer stranskega risa, uporabljenega pri projektu kot vir, desno je izhodni model mesta. Generirane stavbe sledijo zgolj pridobljenim pravilom in niso tridimenzionalne kopije stranskih risov[21].

in terminalne oblike predstavljajo fiksne elemente gramatike, je za pravilno delovanje pomembna disjunktnost abecednih množic V_t in V_m. Pravila za generiranje oblik so zapisana kot oblikovna pravila (u,v) znotraj množice R, kar beremo kakor $u \rightarrow v$, pri čemer u in v predstavljata neko obliko, sestavljeno poljubno iz enega ali več elementov V_t in V_m. Široki nabor možnih kombinacij elementov abecednih množic privede do novih množic V_t^+, V_m^+,V_t^* in V_m^*. Množica V_t^+ vsebuje vse kombincije oblik iz V_t, podobno vsebuje V_m^+ kombinacije elementov V_m. Če množicam dodamo V_t^+ in V_m^+ še prazno obliko oz. element brez vsebine, ustvarimo množici V_t^* in V_m^*. Oblikovno pravilo (u, v) ima vselej u oblikovan kot kombinacijo elementa V_t^* in elementa V_m^+ ter v kot kombinacijo elementa V_t^* in elementa V_m^*. Podobno zgradimo začetno obliko I kot kombinacijo elementa V_t^* in elementa V_m^*, pri čemer I vsebuje vsaj kakšen element enak nekemu u-ju znotraj oblikovnih pravil (u, v) množice R. Kombinacija oblik ni zgolj sopomenka za unijo oblik, saj lahko tudi opravimo razne transformacije na oblikah, kot npr. zrcaljenje, transliranje in rotiranje. Vendar smo zgolj s kombiniranjem oblik omejeni s transformacijami in začetno abecedo, ki jo moramo tako povečati oz. definirati nove transformacije, če želimo razširiti razpon končnega izbora. Alternativa dodajanju redudantnih oblik in transformacij v SG so družine oblik, kjer obliki lahko določimo di-

menzije po podanih navodilih. Družina oblik je definirana s parametrično obliko, ki ima za koordinate točk podane spremenljivke. Član družine je vsaka oblika pridobljena z vstavitvijo konkretnih podatkov v spremenljivke parametrične oblike. Družino oblik lahko vidimo kot poenostavitev transformiranja parametrične oblike, pri čemer imamo poleg klasičnih transformacij na voljo tudi izkrivljanje oblike. Tako lahko parametrični pravokotnik ročno zrcalimo, skaliramo, rotiramo, hkrati pa ga lahko deformiramo v kvadrat ali kakšen drug lik[18][22].



Slika 4.2: Primer enostavne unimarker SG in poteka generiranja[18].

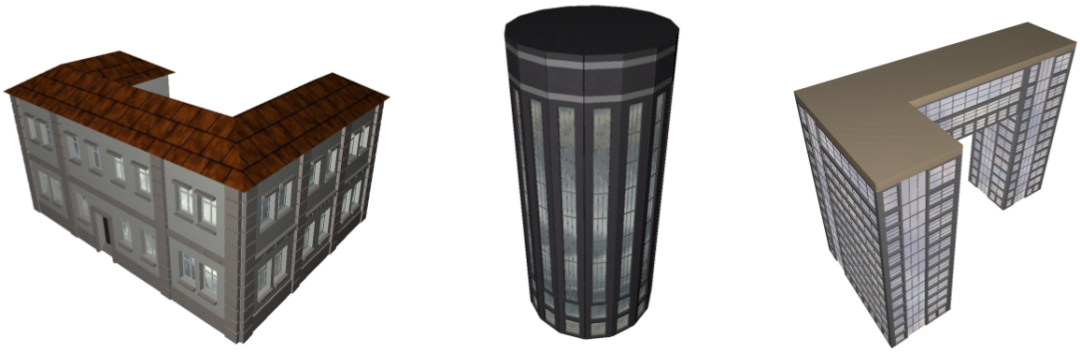
Potek generiranja končnih oblik SG je podoben drugim gramatikam. Znotraj iteracije generiranja se ob uporabi nekega pravila $u \rightarrow v$, poišče v podobi element, ki mu geometrijsko ustreza u tako po končnih elementih kakor tudi po markerjih. Izdelati hitre in uspešne algoritme za iskanje ujemajoče se

podoblike je lahko sila težavno opravilo, še posebej pri tridimenzionalnem prostoru. Kot alternativa se SG rahlo posploši v t.i. gramatiko množic(ang. set grammar). Pri posplošitvi se oblike prikaže kakor objekte in tako ni težav pri iskanju ujemanja v podoblikah. Ko se ugotovi transformacija za u, da ta sovpada z najdenim elementom naše podobe, se enaka transformacija izvede tudi na v. Transformirani v nato zamenja najdeni element na podobi. Postopek izvajanja pravila se lahko izvaja *sekvenčno* ali *paralelno*. Običajno se uporablja sekvenčni SG, kjer se v vsaki iteraciji generiranja pravilo uporabi zgolj na enem ustreznem elementu podobe. Alternativa je paralelni SG, ki poišče znotraj iteracije vse ustrezne elemente podobe in na vseh izvede pravilo. Gramatiko lahko poleg izvajanja pravil omejimo tudi s pravili o elementih pravil. Tako poznamo *neizbrisljivo* in *unimarker SG*. Pri neizbrisljivi SG se omejimo s končnimi elementi levega dela pravila, ki morajo imeti svoj identični element v desnem delu pravila. Z omejitvijo dosežemo, da bo terminalni element ob vključitvi fiksno ostal na svojem mestu. Idejo se lahko razširi še naprej v unimarker SG, ki doda omejitev točno enega markerja v levem delu in kvečjemu enega markerja v desnem delu pravila. Unimarker nam tako omogoči, da imamo vselej opravka z zgolj enim markerjem, kar lahko poenostavi gradnjo gramatik za modele. Slika 4.2 prikazuje enostaven dvodimenzionalen primerek unimarker SG, kjer abecedo sestavljata kvadrat in krog, zadnji je tudi marker gramatike. Jasno prikazuje tudi potek generiranja podobe, kjer vsaka uporaba pravila predstavlja svojo iteracijo[18][23].

4.2 Gramatika delitve

Enostavna in učinkovita podgramatika SG primerna za modeliranje arhitekture je *gramatika delitve*(ang. Split grammar). Gramatika spada med SG, ki so poenostavljene v gramatiko množic. Za modeliranje tridimenzionalnih stavb je to še toliko bolj primerno, saj se posvetimo predvsem gramatiki sami in ne toliko algoritmom za hitro generiranje oblik, kamor spada tudi iskanje podoblik. Prav tako gramatika uspešno rešuje problem naključnosti

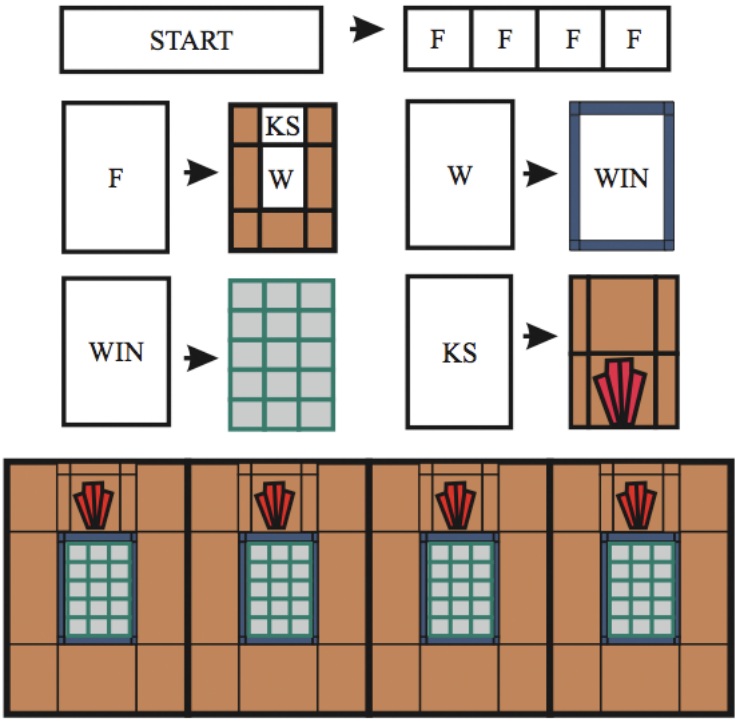
SG pri arhitekturi, ki izhaja iz načina izbire pravil. Večjo naključnost namreč dosežemo z večjim naborom pravil in samih kombinacij, vendar niso vse situacije primerne za naključno pravilo. Hitro se nam lahko zgodi, da bomo imeli v drugem nadstropju stavbe vrata in v prtljičju streho. V drugih gramatikah bi bila alternativa za takšno katastrofo ročna določitev izbora pravil tekom procesa izdelave. Gramatika delitve ima določene omejitve, ki omogočajo ravnovesje med naključnostjo in pravilno avtomatično izbiro pravil. Omejitve povzročijo tudi rahlo determinističnost v postopku izgradnje, v smislu zaporedja obiskovanja podoblik, dobljenih preko delitve[23].



Slika 4.3: Primer končnih podob stavbe pri gramatiki delitve, kjer so bili različni začetni elementi in nastavljeni atributi vzrok tolikšni razliki[23].

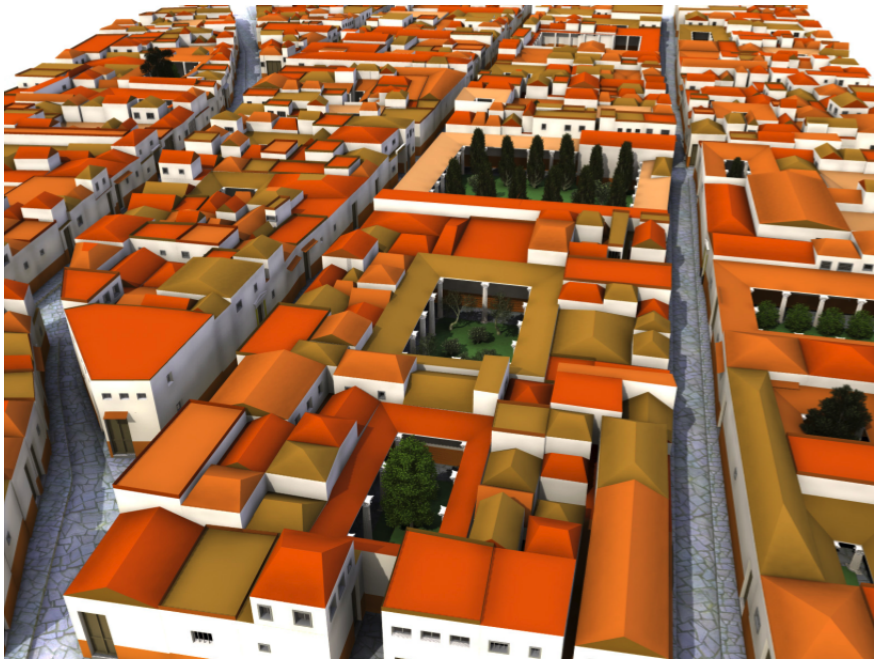
Prva omejitev gramatike delitve je definicija objekta oz. oblike, ki je tu parametrična, označena in hrani določene atribute. Obliko definiramo z *osnovno obliko* $b = \langle s, P, V \rangle$ s konveksnim in zaprtim geometrijskim elementom s, ki je centriran v izhodišču, množico koordinat P in simbolom V. Množica P vsebuje koordinate presekališč stranic oz. ploskev pri treh dimenzijah s koordinatnimi osmi. Simbol V predstavlja povezavo med obliko v abecedi in njeno grafično podobo. Klasični tridimenzionalni primeri osnovnih oblik so kvadri, cilindri in prizme. Elemente abecede tako definiramo kot transformirane osnovne oblike, pri čemer opravimo afine transformacije na geometrijskem elementu s in koordinatah v množici P. Tudi sama ideja kombiniranja oblik je drugačna od splošnega SG. Tu v iteracijah ne govo-

rimo o generiranju nove kombinacije oblik, temveč o novi delitvi oblike ali o zamenjavi osnovnega elementa znotraj oblike. Delitev oblike pomeni dekompozicijo vhodne oblike na več osnovnih oblik abecede znotraj istega prostora po navodilih oblikovnega pravila. Pri pravilu delitve $u \rightarrow v$ predstavlja u povezano podmnožico abecede, oblika v pa vsebuje iste elemete kot u, z izjemo točno enega. Tu se delitev izvede na elementu v u, ki ga ne najdemo več v v. Princip delitve oblike jasno prikazuje slika 4.4 z delitvijo pravokotnika v pročelje hiše. Delitev je morda zavajajoč izraz, saj daje občutek enakomerne porazdelitve, čeprav za to pri dekompoziciji elementa ni potrebe. Pri sliki pročelja je delitev razdelila začetni pravokotnih enakomerno, kar je zgolj posledica izbire parametrov ter uporabljenega pravila, kar se jasno vidi v nadalnjih pravilih delitev. Primer delitve v tridimenzionalnem prostoru, bi bila razdelitev kvadra na $n \times m \times k$ kvadrov znotraj istega prostora[23].



Slika 4.4: Primer delitve pročelja stavbe, kjer simboli znotraj oblik označujejo markerje in barvne oblike terminalne elemente[23].

Poleg pravila delitve poznamo pri gramatiki delitve znotraj oblikovnih pravil še pravilo pretvorbe. Pravilo pretvorbe $u \rightarrow v$ pri u vsebuje poleg drugih elementov osnovno obliko, ki se pri v zamenja z drugo osnovno obliko, medtem ko se preostali elementi ohranijo. Pravilo zamenjave je koristno pri elementih stavbe kot so streha, okno in steber. Edina omejitev pravila pretvorbe je, da mora biti nova osnovna oblika v v znotraj prostora, ki zajema staro osnovno obliko v u. Ravno poraba prostora je opaznejša razlika med pravilom delitve in pravilom pretvorbe. Pravilo delitve zajema namreč enak prostor v vhodni obliki kakor v izhodni, medtem ko pravilo pretvorbe omogoča, da izhodna oblika zajema manjši prostor kakor vhodna. Jasen primer je zamenjava kvadra za prizmo za potrebe strehe, kjer se del začetne prostornine izgubi[23].



Slika 4.5: Rekonstrukcija Pompejev z gramatiko delitve. Osnovno obliko stavb so določili preko temeljev, nato gramatika opravi svoje[24].

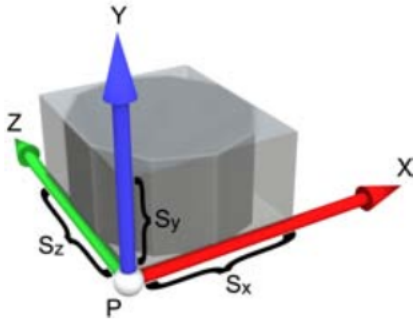
4.3 CGA

Computer Generated Architecture oz. *Computer Graphics Architecture* ali na kratko CGA je naslednja stopnja v razvoju SG in gramatike delitve. Medtem ko gramatika delitve predvsem definira koncept ideje, nam CGA že ponudi standardno obliko gramatike za modeliranje arhitekture. CGA z novimi koncepti v kombinaciji z uveljavljenim okoljem *CityEngine* omogoča modeliranje mnogo kompleksnejših stavb že direktno preko gramatike. Objavljenih je mnogo primerov uporabe CGA za različne tipe stavb, od zgodovinskih hiš vse do nebotičnikov. Slika 4.9 prikazuje rekonstrukcijo Rima s CGA in okoljem CityEngine. Razen nekaterih znamenitosti, rastlinja in vzeptin je celotna rekonstrukcija izvedena preko CGA. Znamenitosti kot kolosej in hipodrom so seveda ročno modelirani zaradi svoje unikatnosti. V primerjavi s sliko 4.5, kjer je so Pompeji generirani preko implementacije gramatike delitve, je razlika v kompleksnosti stavb očitna. Na sliki se tudi vidi ena večjih izboljšav CGA in sicer zaznavanje prostornine med posameznimi oblikami. Sama gramatika delitve ne zna komunicirati med oblikami in tako se lahko pri kompleksnejših zgradbah hitro zgodi kakšno prekrivanje oblik, npr. streha prizidka prekrije okno nadstropja[25][26][27].

Slika 4.6: Rekonstrukcija Rima s CGA in CityEngine okoljem[27].

Oblika je pri CGA definirana kot $shape = \langle s, G, D \rangle$, kjer s predstavlja simbol iz abecede gramatike, G je sama geometrija oz. geometrični atributi in D je skupek numeričnih atributov. Najpomembnejši numerični atributi

so pozicija P, ortogonalni vektorji X, Y in Z, ki opisujejo koordinatni sistem oblike in vektor velikosti S. Našteti numerični atributi definirajo okvir oblike (ang. scope), znotraj katerega se nahaja oblika. Začetni okvir se dodeli že začetni obliki, na kar se skozi iteracije dodaja posamezni okvir vsaki na novo ustvarjeni obliki. Izračun okvirjev se zgodi avtomatično in na posamezni okvir lahko vplivamo zgolj z ukazi za okvir[26][25].



Slika 4.7: Primer okvirja oblike[26].

Proces generiranja ima v primerjavi s prejšnimi variacijami SG zgolj dve bistveni razliki, in sicer aktivnost oblik ter prioriteta pravil. Do sedaj sta se obliki u in v stalno zamenjali po izvedbi oblikovnega pravila $u \rightarrow v$. Pri CGA se uvede aktivnost oblike, ki odloča o vidnosti oblike v podobi. Aktivnost in neaktivnost oblike nam omogoča poizvedovanje po celotni hiearhiji oblike in ne samo po trenutni podobi. Uvedba hierahije nam odpre nove možnosti za manipulacijo oblike, med drugim tudi interaktivno urejanje. Zaradi uvedbe aktivnosti se postopek izvajanja rahlo spremeni. Namesto zamenjave oblik u in v se oblika u označi kot neaktivna in se preko nje postavi aktivno obliko v. Prav tako se pri iskanju oblik, ki ustrezajo u, omejimo na aktivne oblike. Pri iskanju oblik nam CGA rešuje tudi problem nadziranja raznolikosti gradnje. Uvede se prioritete pravil za določanje vrstnega reda glede na podrobnosti, ki jih prikazuje oblika. Iskanje se nato izvede s preiskovanjem v širino, pri čemer imajo prednost pravila z višjo prioriteto. Prioritete pravil s tem omogočijo, da raznolikost gradnje poteka bolj nadzorovano od nizke do visoke stopnje podrobnosti[26].

CGA na novo definira tudi oblikovna pravila in poleg njih tudi uvede CGA ukaze, ki omogočajo krajši zapis pravil in nove možnosti. Nova oblika oblikovnega pravila

$$id : predecessor : cond \rightarrow successor : prob \tag{4.1}$$

ima h klasični obliki $u \rightarrow v$ dodane še indetifikacijsko številko *id*, pogoj *cond* ter verjetnost izbire pravila *prob*, pri čemer sta pogoj in verjetnost opcijske narave oz. relativna na vrsto modela. Sledijo praktični primeri pravil in ukazov:

$$1 : facade(h) : h \geq 9 \rightarrow floor(h/3)floor(h/3)floor(h/3) : 0.9 \tag{4.2}$$

$$2 : A \rightarrow [T(0,0,6)S(8,10,18)I(\text{"}cube\text{"})]T(6,0,0)S(7,13,18)I(\text{"}cube\text{"}) \tag{4.3}$$

$$3 : facade \rightarrow Subdiv(\text{"}Y\text{"}, 3.5, 0.3, 3r, 3)\{floor|ledge|floor|floor\} \tag{4.4}$$

$$4 : floor \rightarrow Repeat(\text{"}X\text{"}, 2)\{B\} \tag{4.5}$$

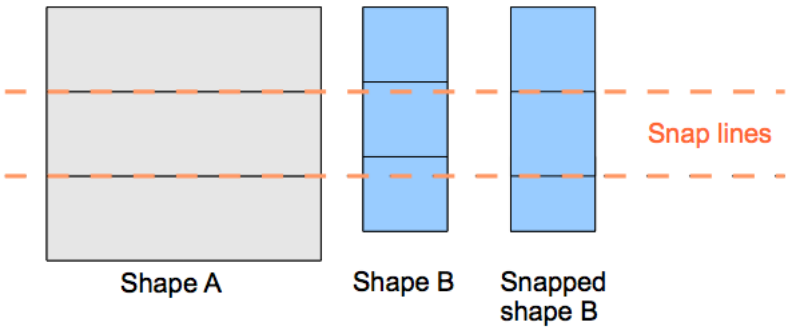
$$5 : a \rightarrow Comp(\text{"}faces\text{"})\{A\} \tag{4.6}$$

Primer pravila 4.2 je klasično pravilo, ki postavi na položaj fasade tri nadstropja z višino, ki je enaka tretjini višine fasade h, če je $h \geq 9$. Prav tako ima pravilo verjetnost izbora 90%. Naslednje pravilo 4.3 uporablja ukaze, ki se izvajajo na okvirju oblike. Podobno kakor pri L-sistemih se tu najdejo ukazi za skalacijo S, rotacijo R, translacijo T, [za začetek novega okvirja na skladu okvirjev in] za skok na prejšnji okvir sklada. Za dodajanje novih instanc geometrijskih primitivov se uporblja ukaz I("tip"). Pravilo 4.4 je delitveno pravilo, katerega grafično podobo smo spoznali že pri gramatiki delitve. Ukaz Subdiv("axis", splitSizes){newElements} nam razdeli izbrano obliko po smeri izbrane osi glede na podane numerične vrednosti in elemente. Numerične vrednosti so lahko tudi relativne, te označimo s priveskom r. Ob izvajanju pravila se relativne vrednosti izračunajo preko formule

$$newAbs_i = r_i * (ScopeS_{axis} - \sum abs_j) / \sum r_i \tag{4.7}$$

kjer $ScopeS_{axis}$ označuje ustrezno komponento vektorja velikosti za okvir, r_i relativne vrednosti zapisane pred r in abs_j prednastavljene absolutne vrednosti. Pri primeru 4.5 najdemo CGA ukaz Repeat("axis", factor){shape}, ki omogoči hitro razdelitev v mrežo oblik. Ukaz bo tekom osi *axis* ponavljajoče delil vhodno obliko v mrežo oblik *shape*, katerim se velikosti ustrezno popravljajo sproti. Delitev v mrežo bo ponovil za $\lceil ScopeS_{axis}/factor \rceil$. Zadnji primer 4.6 prikazuje ukaz $Comp(type, param)\{A|B|...|Z\}$, ki izvaja delitve na oblikah, ki imajo manj kot 3 dimenzije. Dani primer nam na vsako ploskev oblike doda obliko A. Lahko tudi uporabimo parameter *param*, ki omogoča razna dodatna navodila. Takšen primer je Comp("edge", 3){B}, ki novo obliko B poravna s tretjim ogliščem trenutne oblike[26].

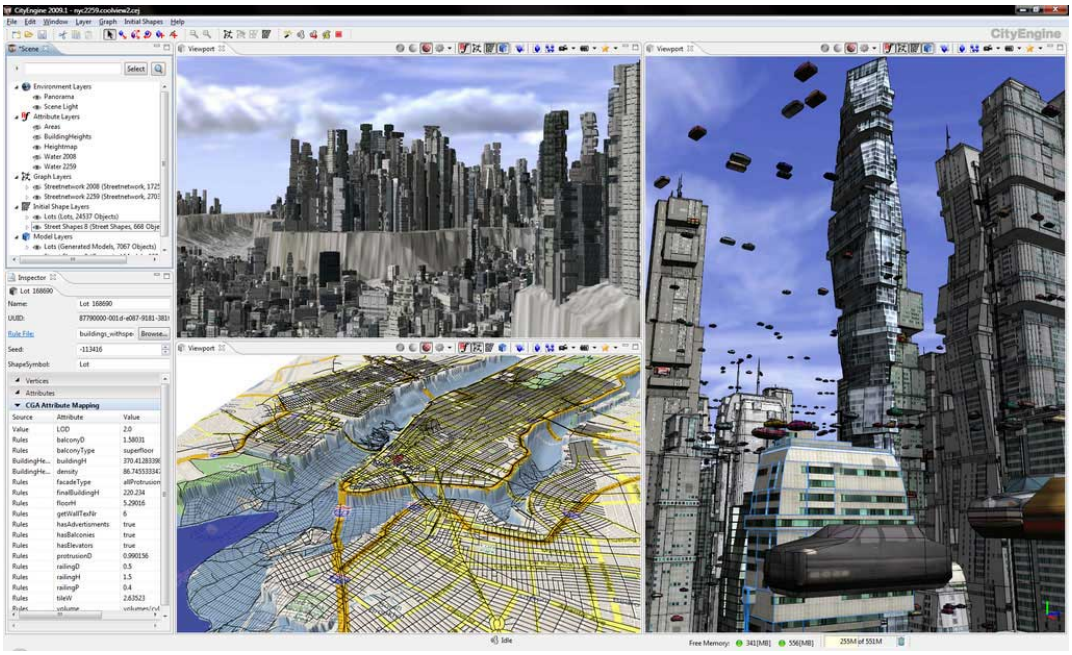
Gradnja stavb pri CGA sledi enostavni ideji, ki omogoča veliko fleksibilnost. Uporablja se zgolj osnovna geometrijska telesa, ki se jih na modelu po želji postavlja preko ukazov za okvir. Čeprav sledimo preprosti ideji, naletimo na problema prekrivanja in poravnave različnih teles. Problem prekrivanja se rešuje s povpraševanji o morebitnih trkih. Če povpraševanje naleti na trk, ko želimo postaviti npr. okno, se okno enostavno zavrže in nadomesti s steno. Problem poravnavanja različnih teles se rešuje z uporabo prijemalnih črt, ki predstavljajo podaljšave črt iz že postavljenih oblik. Prijemalne črte prožimo sami znotraj gramatike z ukazom Snap("axises") pred postavitvijo nove oblike. Vsi nadaljni ukazi split in repeat bodo oblike poravnali z danimi prijemalnimi črtami[25][26].



Slika 4.8: Primer uporabe prijemalnih črt pri poravnavi oblike B[25].

4.3.1 Okolje CityEngine

CityEngine je komercialno orodje, ki omogoča uporabniku hitro generiranje mest po lastnih željah preko grafičnega vmesnika. Popularen je predvsem za rekonstrukcije in razne arhitekturne predstavitve. Okolje je programirano v C++ in za osnovo zgradb uporablja CGA gramatiko. Ker omogoča dodajanje lastne CGA gramatike je še toliko bolj mikavno orodje pri izdelavi lastnih CGA modelov, saj smo tedaj popolnoma prepuščeni sami gramatiki in nam ni potrebno pretirano skrbeti za prikaz. Popularnost kombinacije CGA in CityEngine je vidna tudi iz znanstvenih člankov, kjer ob iskanju proceduralnega modeliranja stavb naletimo najpogosteje ravno na to kombinacijo[28].



Slika 4.9: Grafični vmesnik okolja CityEngine

4.4 Implementacija SGS

Implementacija predstavlja nekakšen vmesen korak med gramatiko delitve in CGA. Pravzaprav smo zgolj razširili sekvenčno gramatiko delitve z nekaterimi novostmi CGA-ja, kot sta koncepta okvirja oblike in aktivnost oblik, kar tehniko močno poenostavi za implementacijo. Temu primerno je implementacija poimenovana *Split Grammar Simulator* oz. z okrajšavo SGS. SGS tvorijo sledeči programski paketi: *sgs.core, sgs.buildingComponents, sgs.bluePrints in sgs.gui.*

4.4.1 sgs.core

Paket predstavlja jedro SGS-ja, saj vsebuje vse grafične elemente in ogrodja, potrebna za delovanje. Grafični pogon za izris modelov je *SGSVisualizator*, ki zgolj prikazuje produkt načrtov, ki razširajo razred *BluePrint*. *BluePrint* in *SGSModel3D* predstavljata pravo jedro programa. Medtem ko *BluePrint* hrani pravila delitve in pretvorbe ter dodatne lastnosti oblik, kot npr. barve, nosi *SGSModel3D* geometrijski pomen oblik. *SGSModel3D* se razširi v *BasicShape*, ki hrani fiksno definirane osnovne oblike, in v *BuildingComponent*, ki predstavlja obliko, grajeno iz enega ali več *BasicShape* oz. *BuildingComponent* oblik in transformacij, izvedenih na njih. Osnovne oblike se od preostalih oblik razlikujejo tudi po terminalnosti, saj so privzeto lahko le terminalne oblike in v hiearhiji ne morejo imeti otrok. Razred, vreden razlage, je tudi *Distribution*, ki hrani spodnjo in zgornjo mejo verjetnosti pravila v načrtih. Ker osnovne oblike predstavljajo geometrijsko jedro modela, razširitve razreda *BasicShape* hranimo v podpaketu *sgs.core.basicShapes*. Priloženo zbirko osnovnih oblik sestavljajo:

- *Box* - preprosta kocka.
- *Cylinder* - preprost približek valja sestavljen iz trikotnikov, ki sestavljajo osnovni ploskvi, in pravokotnikov, ki gradijo plašč.
- *Prism* - štiristrana prizma, ki ima plašč sestavljen iz trapezov.

- *Pyramid* - štiristrana piramida, ki ima za osnovno ploskev kvadrat.
- *TriangularPrism* - štiristrana prizma, ki ima plašč sestavljen iz dveh trikotnikov in dveh kvadratov.

4.4.2 sgs.buildingComponents

Predstavlja zbirko razširitev razreda *BuildingComponent*:

- *Door* - preprosta vrata s kvadrasto kljuko.
- *Facade* - omogoča razdelitev zidu ali kakšnega drugega elementa na več podenot. Osnovni razred ima več razširitev, ki povečajo število podenot oz. spremenijo njihove simbolne označbe.
- *ChurchBell*, *Factory4Roof*, *Floor*, *SimpleHouse*, *SimpleFactory*, *SimpleSkyscraper* in *Tower* - predstavljajo razdelitve pripravljene za določene načrte.
- *InitialBasicShape* - omogoča postavitev osnovne oblike, ki bo posredno imela otroka. Razred pravzaprav shrani v sebi podano osnovno obliko in prekopira numerične podatke potrebne za pravilno delovanje. Uporablja se predvsem kot začetni ovoj pri začetni osnovni obliki.
- *Wall* - preprosti ozek zid.
- *Window* - preprosto okno, sestavljeno iz okna in štirih robnih zidov.

4.4.3 sgs.bluePrints

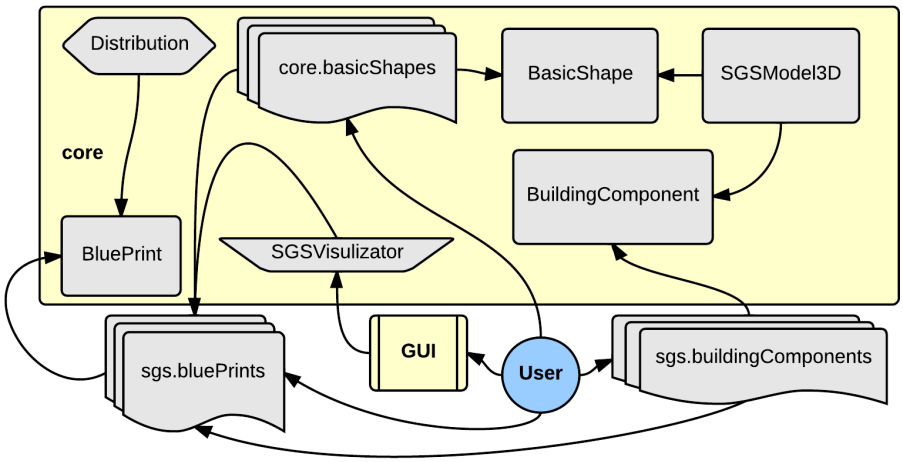
Hrani konkretne načrte za arhitekturo oz. razširitve razreda *BluePrint*:

- *Church* - preprosta cerkev z zvonikom, kjer se vrata postavijo naključno.
- *Factory* - preprosta tovarna z deljeno streho, dimnikom in naključno postavljenimi vrati.

- *House* - preprosta hiša, ki nudi naključno postavitev oken in vrat.
- *Skyscraper* - preprost nebotičnik, ki nudi naključno izbiro strehe in postavitve vrat v pritličju.

4.4.4 sgs.gui

Paket gradi uporabniški vmesnik, ki omogoča hitro izbiro načrta in njegovega takojšnega oz. postopnega razvoja. Ob uporabi možnosti *Show Wireframe* se načrt razvija z označenimi robovi uporabljenih oblik, kar omogoča jasno viden potek iskanja v globino po oblikah in uporabe pravil na neterminalnih oblikah. Gumb *Start fresh* zažene novo instanco modela, ki zamenja morebitni stari model. Razvijanje modela postopoma dosežemo z gumbom *Next iteration*, razvoj v trenutku pa z gumbom *Finish model* .



Slika 4.10: Arhitektura SGSja.

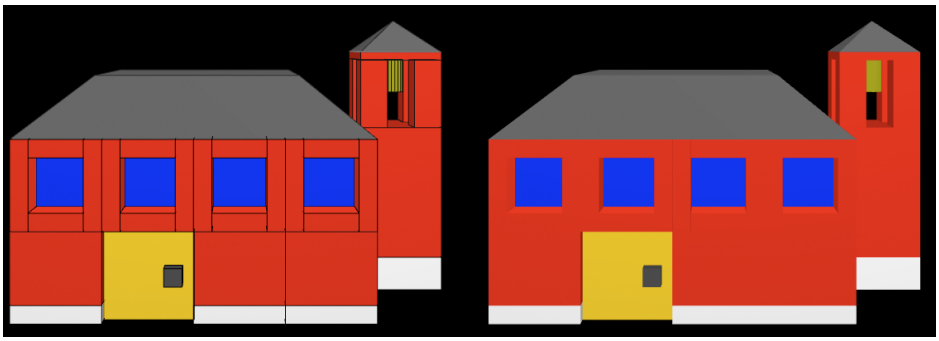
4.4.5 Definiranje novih stavb

Nov načrt se lahko izdela že z uporabo podanih oblik, če te zadoščajo. Potrebno je zgolj razširiti razred *BluePrint*, ki ima že vgrajeno preverjanje elementov pravil in abecede oblik. Za dodajanje pravil, elementov abecede in začetne oblike so na voljo tudi razne metode. Ob potrebi lahko tudi

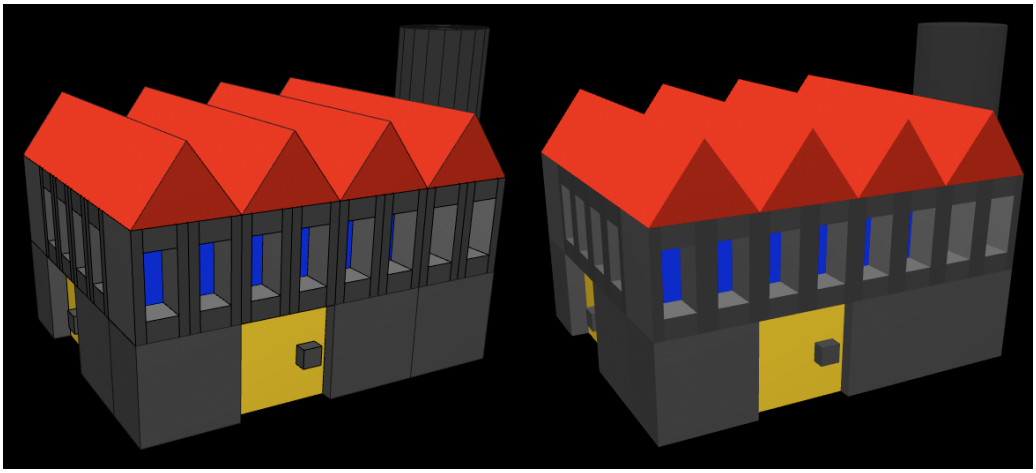
razširimo razreda *BasicShape* oz. *BuildingComponent*, vendar ne smemo pozabiti na implementacijo metode *getFirstNonterminalElementDefinedInImplementation* pri razširitvi razreda *BuildingComponent*. Metoda namreč vrača morebitne neterminalne oblike definirane v razredu, saj jo uporablja preiskovanje v globino. Kadar preiskovanje v globino odkrije neterminalno obliko, ki nima podanega pravila delitve ali pretvorbe, obliko označi kot terminalno in ponovno prične iskanje. Prireditev v terminalno obliko predstavlja rahlo poenostavitev gramatike delitve, vendar jo lahko razumemo kakor implicitno pravilo pretvorbe. Priporočeno je slediti tudi načelu gramatike delitve, da obliko definiramo kot centrirano v koordinatnem izhodišču, čeprav uporabljamo okvirje oblike. Priporočilo namreč poenostavi gradnjo oblike in težje pride do nepredvidljivih situacij.

4.4.6 Primeri rezultatov

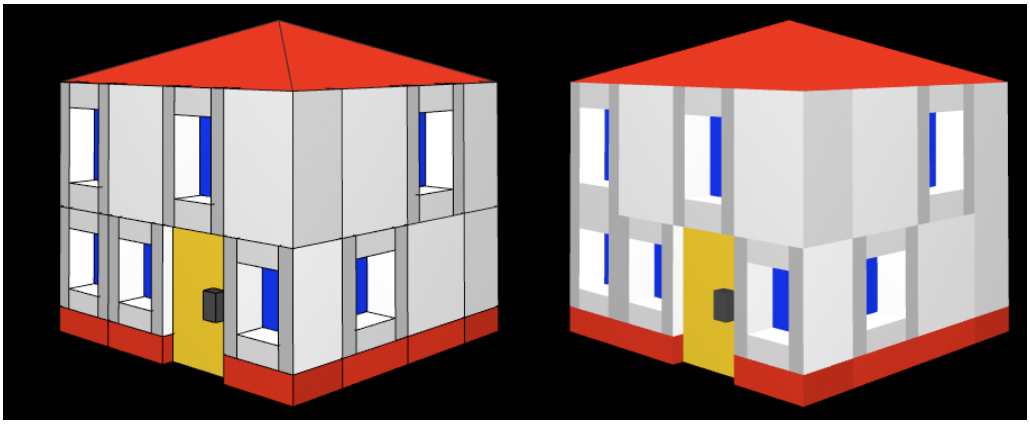
Vsi načrti, prisotni v implementaciji, so stohastični, vendar se razlikujejo po številu naključno določenih elementov. Pri načrtih *Church* in *Factory* je naključnost omejena zgolj na postavitev vrat in zidov v pritljičju. Načrt *Skyscraper* temu doda še naključno izbiro strehe med prizmo in piramido, načrt *House* pa naključnost razširi na okna in s tem uvede povsem naključni videz pročelja hiše. Primeri končnih modelov za posamezne načrte so vidni v slikah od 4.11 do 4.14, kjer je povsod začetna oblika kvader različnih dimenzij.



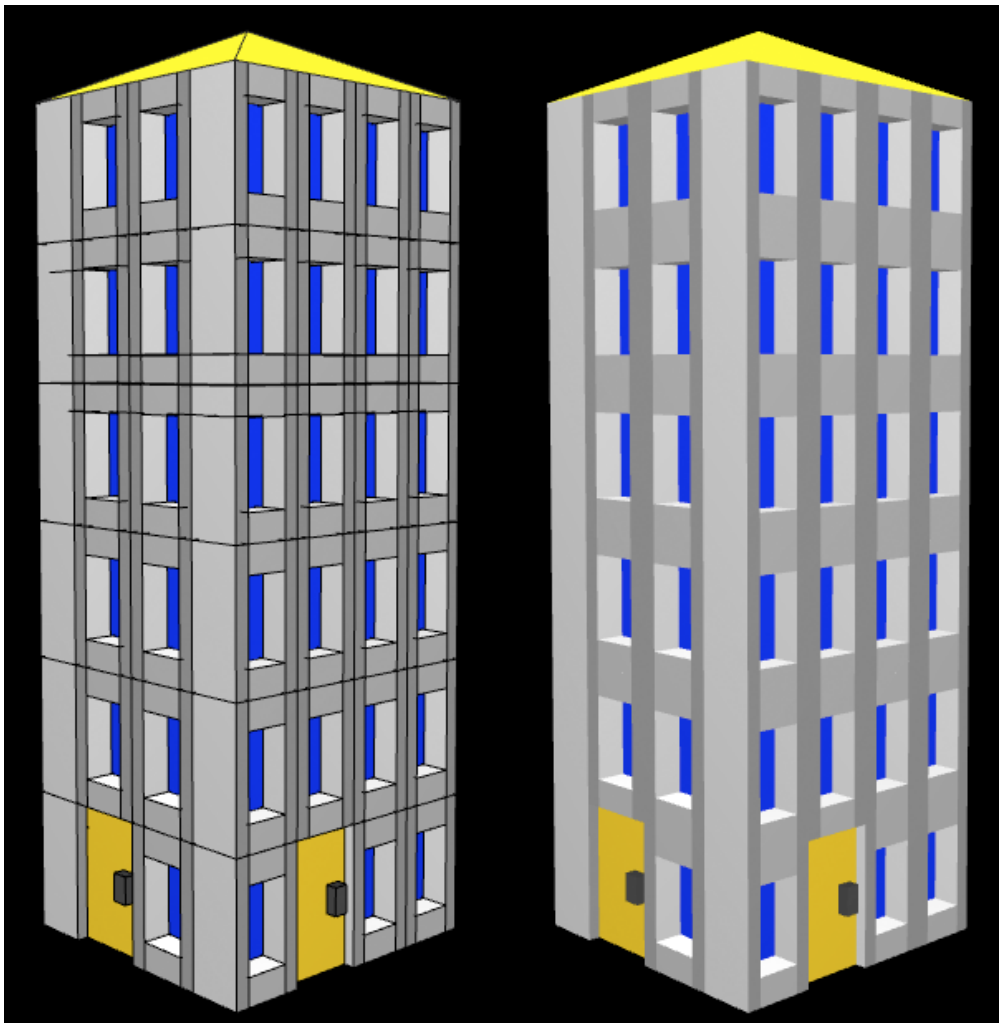
Slika 4.11: Primer generiranega modela cerkve iz načrta Church. Leva slika ima vključene robove oblik, desna jih ima izključene.



Slika 4.12: Primer generiranega modela tovarne iz načrta Factory. Leva slika ima vključene robove oblik, desna jih ima izključene.



Slika 4.13: Primer generiranega modela hiše iz načrta House. Leva slika ima vključene robove oblik, desna jih ima izključene.



Slika 4.14: Primer generiranega modela nebotičnika iz načrta Skyscraper. Leva slika ima vključene robove oblik, desna jih ima izključene.

Poglavje 5

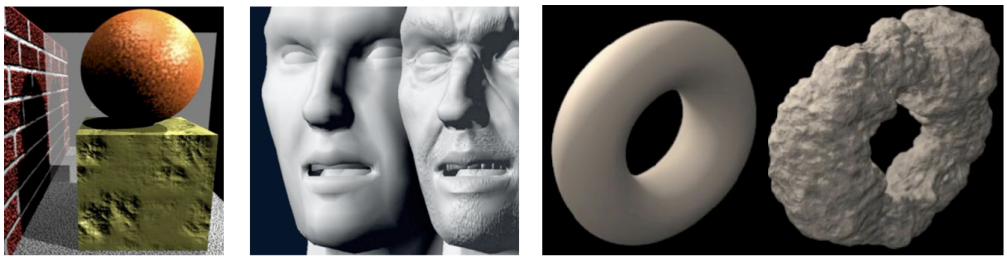
Višinska slika

5.1 Splošno

Višinska slika(ang. heightmap oz. heightfield) je rasterska slika, ki v svojih pikslih hrani določene podatke. Najpogosteje hranimo posamezne višine modeliranega reliefa, hranimo pa lahko poljubne podatke glede na naše potrebe in znotraj omejitev slike. Slika 5.2 je tipičen primer višinske slike reliefa[29].

Slika 5.1: Normalizirana 8-bitna višinska slika reliefa Zemlje, vključno z vodo in ledom[29].

Lepota sivinske slike je v intuitivnosti pomena odtenkov na preprosto in hitro zgrajeni sliki. Črna barva predstavlja minimalno vrednost, kar je v primeru slike 5.2 višina morske gladine, bela barva pa določa maksimalno vrednost, kar je v primeru slike 5.2 nadmorska višina najvišjih gorstev. Vmesni odtenki sivine postajajo svetlejši z večanjem shranjene vrednosti. Področja višinske slike so predvsem povezna s terenom, saj v primerjavi s poligonskimi mrežami dosežemo mmnogo manjšo porabo spomina pri visoki stopnji podrobnosti. Pogosto jih tudi najdemo pri prikazovanju in sporočanju informacij. Lep primer vpliva informacij prikazuje slika 3.2, kjer posredovane informacije o lokacijah kopnega, razgibanosti reliefa in gostoti poselitve določajo omejitve L-sistema. Posredovanje informacij preko višinskih slik znotraj računalniške grafike najde tudi dom pri teksturah modelov, saj lahko z njimi izvedemo razne efekte na modelih. Primeri t.i. teksturnih efektov, ki uporabljajo višinske slike, so lepljenje izboklin(ang. bump mapping), lepljenje normal (ang. normal mapping) in lepljenje odmika(ang. displacement mapping). Pri lepljenju izboklin z višinsko sliko vplivamo na izračun osvetlitve modela, s čimer končna podoba pridobi na hrapovosti oz. na izbolkinah. Poenostavitev lepljenja izbolkin je lepljenje normal, kjer v RGB komponentah višinske slike hranimo nove XYZ normale modela, ki zamenjajo stare normale modela. Lepljenje normal omogoča kotnrolo gladkosti izven modela samega. Najbolj ekstremno izmed trojice je lepljenje odmika, ki z višinsko sliko spremeni dejansko geometrijo modela[29][30].

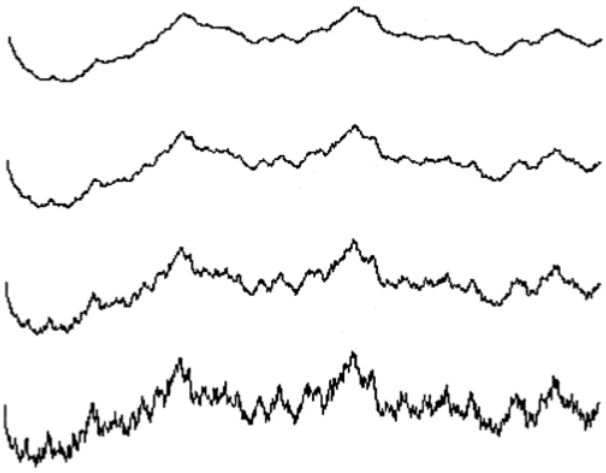


Slika 5.2: Primeri uporabe teksturnih efektov, od leve si sledijo lepljenje izbolkin, lepljenje normal in lepljenje odmika[30].

Višinske slike običajno hranimo kot slike običajnih 8-bitnih formatov. To pomeni, da smo omejeni z RGB standardom glede razpona shranjenih vrednosti. Če shranimo vrednost v zgolj en kanal piksla, imamo na voljo le 256 vrednosti, kar je pri nekaterih podatkih izredno premajhna vrednost in tako preko normalizacije izgubimo vrednost podatkov. Pri uporabi vseh treh RGB kanalov za shranjevanje vrednosti imamo na voljo razpon velikosti $256^3 = 16.777.216$ in če temu dodamo še alfa kanal, pridobimo razpon velikosti $256^4 = 4.294.967.296$. Za karkoli več potrebujemo izbrati format slike z več biti[29].

5.2 Algoritmi višinskih slik

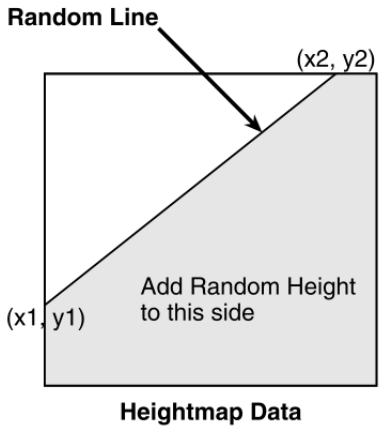
Preprosti pristop k izdelavi višinskih slik je uporaba zbranih realnih podatkov in njihova pretvorba v višinsko sliko. Sliko lahko izdelamo tudi ročno, brez pravih podatkov v programih za urejanje slik kot sta Slikar ali Photoshop. Z vidika proceduralnega modeliranja nas veliko bolj zanimajo algoritmi, ki gradijo naključne višinske slike z visoko stopnjo realnosti. Močna skupina algoritmov za generiranje višinskih slik se imenuje *Fractal Terrain Modelling*, ki generirajo višinske slike reliefov. Njihovi začetki segajo do Benoita Mandelbrota, ko je opazoval grafe fraktalnih Brownovih gibanj v odvisnosti od časa, glej sliko 5.5. Opazil je podobnost v izgledu grafov in zračni liniji gorske verige, iz česar je sklepal, da bi s prehodom v dve dimenziji prispel do aproksimacije gorskega reliefa v naravi. Od takrat dalje večina algoritmov za izdelavo reliefa uporablja fraktalno Brownovo gibanje kot temelj za razvoj novih idej, čeprav ni nobenega znanega razloga, zakaj ima matematični fenomen tako podobnost z gorstvi. Ravno rahla podobnost z gorstvi je največji problem algoritmov, saj ne generirajo gorstev, ki bi upoštevali odtoke vode in s tem tudi erozije. Takšne modele je običajno potrebno še nadalje obdelati s filtri, ki simulirajo erozijo[31][32][33].



Slika 5.3: Grafi fraktalnih Brownovih gibanj v odvisnosti od časa.[34].

5.2.1 Generiranje napak

Ideja algoritmov je naključno generiranje "napak" znotraj ravnine višinske slike. Razlikujejo se tako po izpeljavi kakor naključnosti. Osnovni primer algoritma je *Fault formation*, kjer v iteracijah z naključno črto razdelimo ravnino in enemu delu prištejemo naključno višino.



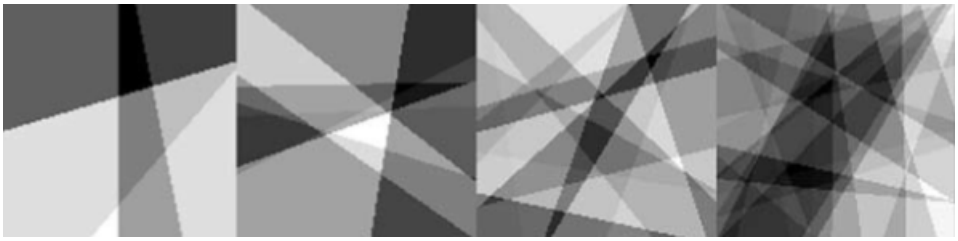
Slika 5.4: Prvi korak Fault formation algoritma.[35].

Če dopustimo tekom iteracij povsem naključno višino, potlej bomo vselej kreirali kaotične reliefe. Rešitev je linearno nižanje meje maksimalno generi-

rane višine. Že preprosta formula

$$mejaV_i = maxV - ((maxV - minV) * i)/vsehIteracij \tag{5.1}$$

nudi dobre rezultate, še posebej ob filtriranju za erozijo. Poleg predstavljenega pristopa poznamo npr. *Poisson faulting*, kjer se uporablja normalna porazdelitev za naključnost, in lahko uporabimo algoritem tudi na sferi, s čimer je primeren za generiranje planetoidov[33][35].



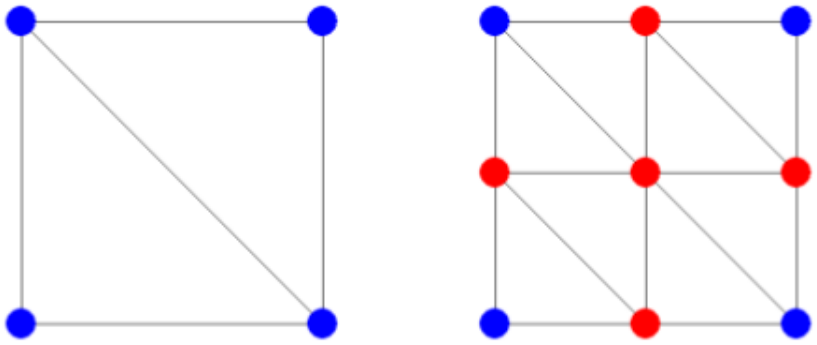
Slika 5.5: Primeri višinskih slik generiranih s Fault formation pri 4, 8, 6 in 32 iteracijah.[35].

5.2.2 Rekurzivna razdelitev

Algoritmi z rekurzivno razdelitvijo skozi iteracije dele reliefa razdelijo na več manjših delov. Vsak algoritem se prične s štirimi robnimi višinami, ki tvorijo kvadrat. Kvadrat se v rekurziji razdeli na štiri nove kvadrate, katerih vrednosti oglišč izračunamo iz oglišč prejšnjega kvadrata s prištevkom naključnega odklona. Naključni odklon niža nivo deterministočnosti naklona, vendar mora biti njegovo območje proporcionalno s stranico trenutnega kvadrata, drugače lahko nastanejo na reliefu nenaravni vrhovi v obliki špic[31].

Triangular-Edge Subdivision

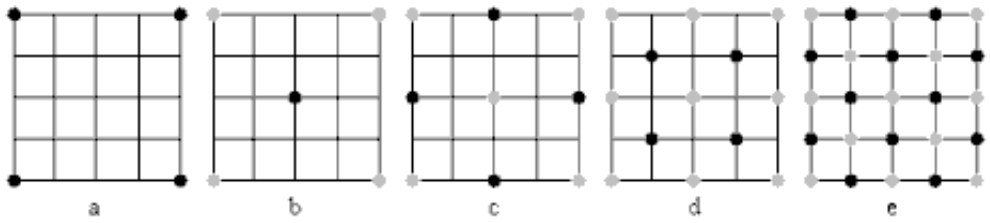
Algoritem predstavlja najpreprostejšo metodo razdeliteve, kjer kvadrat najprej razdelimo na dva trikotnika. V razpolovišče vsake stranice trikotnika se nato shrani naključna vrednost, kar privede do petih novih točk. Nove in stare točke ustrezno tvorijo nove kvadrate, na katerih se postopek ponovi. Slabost pristopa je neupoštevanje vrednosti starih oglišč[31].



Slika 5.6: Prikaz delovanja algoritma Triangular-Edge Subdivision. Modre točke predstavljajo stare točke, rdeče pa generirane točke[31].

Diamond-Square Subdivision

Ime algoritma izvira iz dvostopenjskega poteka, kjer oglišča izmenično povezujemo v kvadrate in diamante oz. kare. Namesto računanja vrednosti v razpoloviščih stranic likov, tu računamo vrednosti središča lika s povprečjem oglišč in prištevkom naključnega odklona. Znotraj algoritma je potrebno tudi rešiti dilemo robnih diamantov, katerim manjka eno oglišče. Za manjkajoče oglišče lahko bodisi interpoliramo sosednji oglišči bodisi uporabimo vrednost na nasprotni strani mreže[31][36].



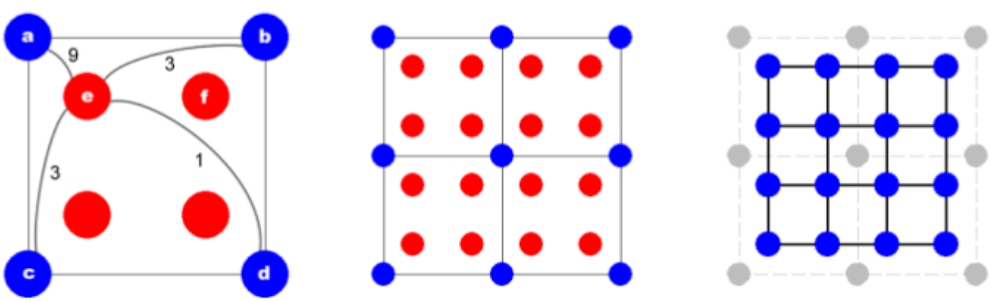
Slika 5.7: Prikaz delovanja algoritma Diamond-square Subdivision. Črne točke predstavljajo generirane točke, sive pa stare točke. Iteracije a, c in e združijo točke v kvadrate, iterciji b in d pa v diamante[31].

Square-Square Subdivision

Tu razdelimo kvadrat na polovico manjši kvadrat, ki je centriran znotraj izvornega kvadrata. Posledično postavitev novega kvadrata postavi nova oglišča na razpolovišče razdalje med ogliščem in središčem starega kvadrata. Vrednosti novih oglišč se izračunajo kot uteženo povprečje starih s prištevkom naključnega odklona. Primer izračuna oglišča e pri sliki 5.8 bi bil lahko

$$e = (9a + 3b + 3c + 1d)/16 + random \tag{5.2}$$

Za pravilno delovanje algoritma je potrebno začeti z mrežo devetih oglišč, drugače ne obdržimo kvadratne mreže. Za razliko od prejšnjih algoritmov razdelitve tu stara oglišča zavržemo in nova mreža zamenja staro. Algoritem nudi mnogo bolj gladke reliefe kakor prejšnji[31].



Slika 5.8: Prikaz delovanja algoritma Square-Square Subdivision. Modre točke predstavljajo začetne točke v mreži, rdeče so generirane točke in sive predstavljajo zavrženo točke[31].

5.3 Implementacija HMG

HeightMap Generator oz. skrajšano HMG predstavlja implementacijo predstavljenih algoritmov za generiranje višinskih slik. HMG hkrati zgenerira višinsko sliko in jo prevede v tridimenzionalni prostor. Njegovo delovanje je razdeljeno v troje paketov: *hmg.generators, hmg.graphics in hmg.gui.*

5.3.1 hmg.generators

Predstavlja jedrni paket, ki vsebuje tako osnovo za definicijo generatorja razred *Generator* kakor tudi njegove razširitve v implementacije željenih algoritmov. Generatorji omogočajo shranjevanje višinskih slik v slikovnih datotekah in glajenje rezultatov s pomočjo erozijskega filtra. Erozijski filter višine ponovno izračuna kot vsoto utežene začetne višine in nazadnje izračunane vrednosti, kar povzroči gladkejšo podobo višinske slike.

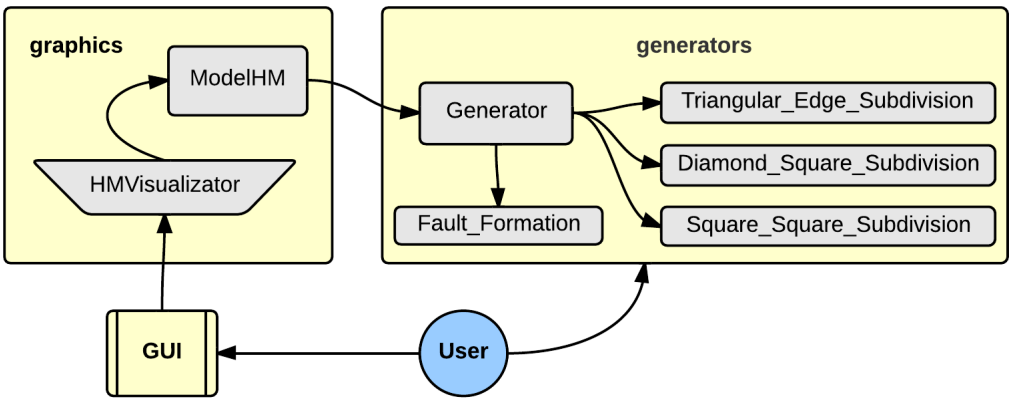
5.3.2 hmg.graphics

Grafična predstavitev višinske slike v tridimenzionalnem prostoru se izvede v razredu *ModelHM*, ki prebere višinsko sliko iz datoteke in dobljene višine ustrezno združi v sosednje trikotnike. Dejanski prikaz predstavitve izvede *HMVisualizator.*

5.3.3 hmg.gui

Zaradi različnih pristopov generatorjev je uporabniški vmesnik razdeljen v tri okna. Prvo okno omogoča izbiro metode generiranja, ki nato proži naslednje okno. Videz vmesnega okna se prilagodi vhodnim potrebam generatorjev in omogoča naključno generiranje vhodnih podatkov za generator. Ob posredovanju vhodnih podatkov se odpre zadnji del uporabniškega vmesnika, ki je zopet skupen za vse generatorje. Zadnje okno predstavlja krmilnik generatorja, ki vsebuje tudi dvodimenzionalno predstavitev višinske slike. S pritiskom na gumb *Start fresh to* nam generator iz vhodnih podatkov ge-

nerira višinsko sliko po številu iteracijah, podanih v krmilniku. Ta pristop generiranja je še posebej primeren pri *Fault_Formation*, kjer se poskuša linearno omejiti pribitke višine, za kar pa potrebujemo skupno število iteracij. Ob pritisku gumba *Add iteration* se višinski sliki doda nova iteracija, kar se pri *Fault_Formation* izvede brez omejitve naključnega višinskega pribitka. Zadnji gumb *Save Heightmap* nam omogoča shraniti trenutno višinsko sliko kot slikovno datoteko na poljubno mesto. Krmilnik tudi omogoča aktivacijo erozijskega filtra preko možnosti *Filter Heightmap*, nakar se uporabniku prikaže drsnik za nadzor moči filtra. Številka v kotu krmilnika predstavlja velikost višinske slike.



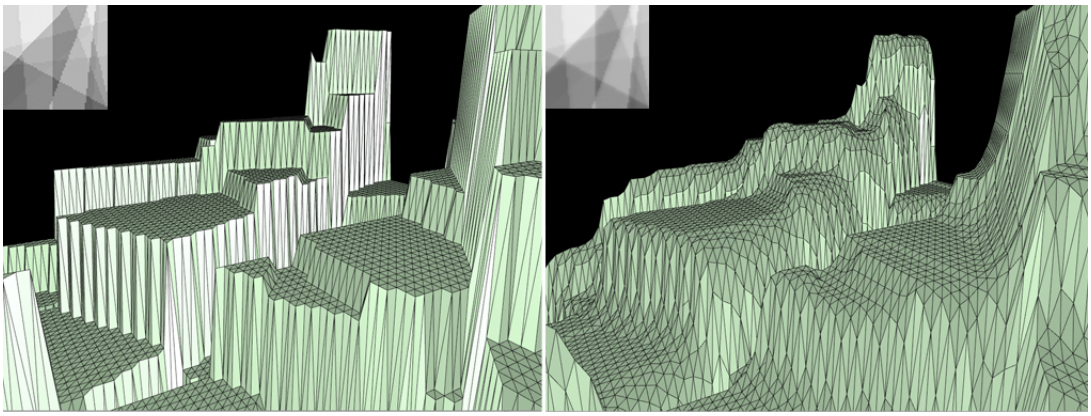
Slika 5.9: Arhitektura HMGja.

5.3.4 Definiranje novih generatorjev površja

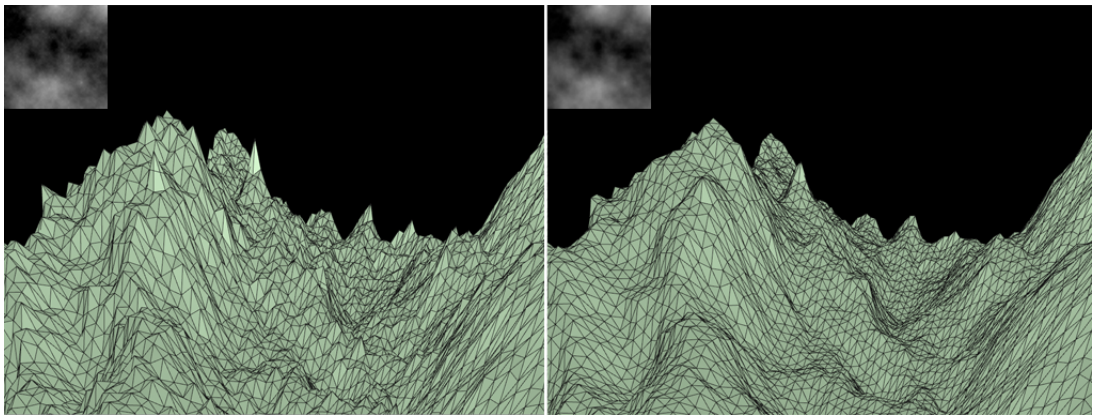
Uporabnik lahko implementira poljubni algoritem za generiranje višinskih slik, vendar ga mora implementirati kot razširitev razreda *Generator* znotraj paketa *hmg.generators*. Za uporabo novega generatorja v uporabniškem vmesniku, je potrebno prirediti srednje okno za nastavitev vhodnih podatkov generatorja. Okno je definirano v razredu *UserInput*, kjer je potrebno najmanj prirediti konstruktor, če so elementi vmesnika ustrezni.

5.3.5 Primeri rezultatov

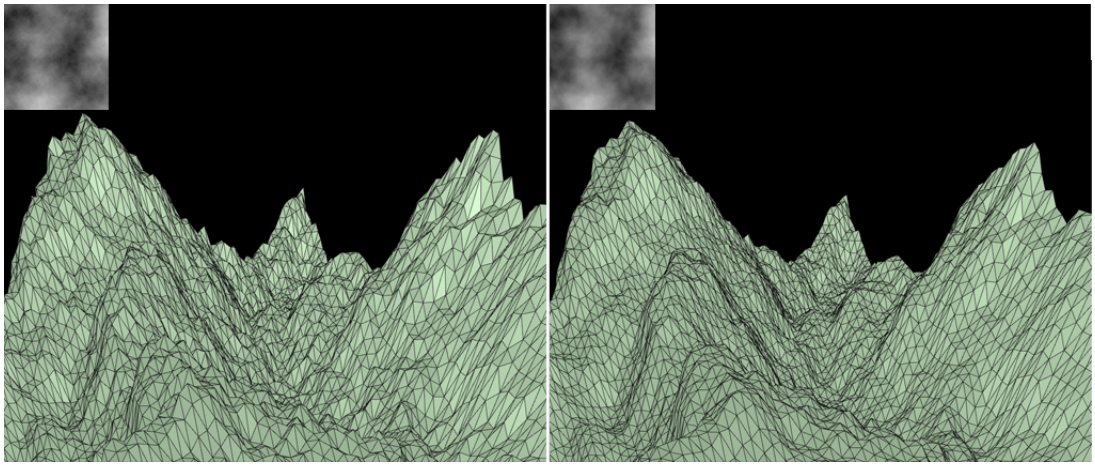
Sledeče slike od 5.10 do 5.13 prikazujejo višinske slike, grajene z vsemi štirimi predstavljenimi algoritmi, njihove tridimenzionalne predstavitve in z erozijskim filtrom popravljene podobe. Rezultati posameznih algoritmov imajo nekatere tipične lastnosti, ki so razvidne že iz slik. *Fault_Formation* tvori terasaste reliefe kot v sliki 5.10 pri fiksni velikosti, s čimer mislimo dolžino x širino pikslov. Za koničaste gorske vrhove je primeren *Diamond_Square_Subdivision*, medtem ko *Square_Square_Subdivision* bolje generira gladke gorske verige, primerjaj sliki 5.11 in 5.12. Za razliko od preostalih algoritmov nam naključni *Triangualar_Edge_Subdivision* vrača zgolj kaotične reliefe, glej sliko 5.13. Skupna lastnost višinskih slik rekurzivnih razdelitev je hitra rast v številu točk, s čimer se veča zahtevnost grafične podobe reliefa. Samo gladkost terena lahko poleg izbire algoritma urejamo tudi z erozijskim filtrom, ki mu lahko določimo moč na intervalu [0,100], kjer 0 pomeni neobstoj erozije in 100 poravnavo terena v ravnino.



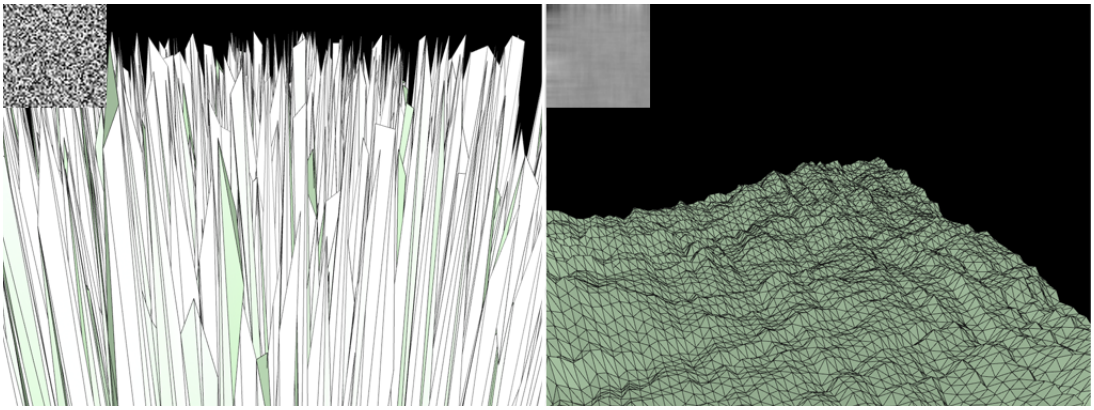
Slika 5.10: Terasasti relief v levi sliki je tipična podoba višinske slike, ki je bila generirana s Fault_Formation. Višinska slika ima velikosti 65x65 in 12 iteracij. Desna slika je levo višinsko sliko spustila skozi filter erozije z močjo 40%.



Slika 5.11: Diamond_Square_Subdivision generira običajno gorsko verigo s koničastimi vrhovi kot v levi sliki . Višinska slika ima velikosti 65x65 in 6 iteracij. Gladko podobo desne slike dosežemo že s 30% filtom erozije.



Slika 5.12: Leva slika prikazuje tipičen rezultat Square_Square_Subdivision-a z dokaj gladko gorsko verigo. Višinska slika ima velikosti 66x66 in 6 iteracij. Zaradi visoke gladkosti rezutata algoritma je v desni sliki uporabljen zgolj 20% filter erozije.



Slika 5.13: Kaotičnost višinskih slik generiranih s Triangualar_Edge_Subdivision je očitna že v dvodimenzionalnem prostoru, kjer dobimo občutek šuma, še bolj pa pride do izraza v tridimenzionalnem prostoru, kar je oboje razvidno v levi sliki. Višinska slika ima velikosti 65x65 in 6 iteracij. Zaradi kaotične podobe višinske slike je za smiselno gladko podobo v desni sliki potreben kar 90% erozijski filter.

Poglavje 6

Analiza tehnik in njihovih implementacij

Ob primerjavi predstavljenih tehnik hitro ugotovimo, da se tehnike močno razlikujejo že na konceptualni ravni. Posledično skupinska analiza tehnik ne more določiti relativne kvalitete posameznih tehnik, saj so bile tehnike ustvarjane za dotične probleme modeliranja v računalniški grafiki. Tako lahko s fraktali modeliramo samopodobnost, kar je pri proceduralnem modeliranju ključnega pomena za poenostavitev modela in se posledično fraktali mnogokrat uporabljajo v povezavi drugimi tehnikami. Nasprotno višinske slike predstavljajo zapis modelov, ki so definirani z matriko vrednosti. Višinske slike tako redkeje srečamo izven modeliranja reliefa, zapisa omejitev kot npr. gostota poselitve in teksturnih efektov. Po širini možnosti uporabe tehnik bi tako morali gramatiko oblik in L-sisteme postaviti na sredino med fraktale in višinske slike. Z gramatiko oblik lahko namreč enostavno opišemo predmete kot so stavbe, kjer z opisom pričnemo pri osnovnih geometrijskih telesih in opise tekom korakov dopolnjujemo s podrobnostmi sestave. L-sistemi nasprotno gradijo model z njegovo rastjo iz začetnega elementa, kar je idealno za simulacijo rastlinja, prometne infrastrukture, rečne struge, ipd.

Zanimivo dejstvo predstavljenih tehnik je njihova starost. Če namreč izpustimo višinske slike, katerih definicija je že dolgo poznana na področju

kartografije, so bile preostale tehnike kljub različnim konceptom uradno definirane v kraktem obdobju sedmih let 70. let prejšnjega stoletja. Poleg koncepta samega je najverjetneje ravno dolga doba 40-ih let razlog, da so se pristopi utrdili na področju proceduralnega modeliranja in se pri tem uspešno nadalje razvijali. Začetne definicije tehnik so bile predstavljene po sledečem vrstnem redu: 1968 L-sistemi, 1972 gramatika oblik in 1975 fraktali. Presenetljivo je dejstvo, da so kljub velikega pomena samopodobnosti v proceduralnem modeliranju bili fraktali zadnja definirana tehnika[7][18][2].

Vprašanje za programerja, ki bi bilo vredno analize, je zagotovo zahtevnost implementacije posameznih tehnik. Sledeča analiza je narejena na podlagi priloženih implementacij, količini njihovih sestavnih elementov in številu porabljenih ur za njihovo implementacijo. Najpreprostejša tehnika so zagotovo višinske slike. Sam tridimenzionalen prikazovalnik višinske slike ni zahtevan, potrebno je zgolj pravilno generirati mrežo trikotnikov iz podane slike. Ravno tako ni velik izziv implementacija algoritmov za generiranje višinskih slik. Za rahlo zahtevnešjega se izkaže algoritem MRCM, ki mora skozi rekurzijo omogočati uporabo modela definiranega v prejšnjem koraku in uporabniku omogočiti metode, s katerimi enostavo definira pravila za nove fraktale. Mnogo zahtevnejša implementacija je program SGS, kjer je potrebno implementirati enostaven sistem oblik, njihovega pomnenja za pravilen izris, iskanje neterminalnih oblik v modelu in uporabniku enostaven zapis novih načrtov z abecedo, pravili in začetno obliko. Poskrbeti je potrebno tudi za upoštevanje pravila prostornine, ki zahteva od pravil zamenjave in pretvorbe, da novi element ne presega prostornine starega elementa. Pri implementacijah MRCM in SGS se kot dober koncept izkaže okvir oblike, ki definira kvader, ki zajema model. Okvir oblike tako omogoči hitro računanje in uspešno transformiranje modela. Ravno problem neznane velikosti modela srečamo pri L-sistemih oz. pri implementaciji LSPS. Izmed štirih implementacij se je ravno LSPS izkazal za najzahtevnešjo. Potrebno je implementirati osnovno gramatiko in ji dodati možnost poljubne uporabniške razširitve, generator nizov, ki uspešno uporablja osnovno gramatiko in njene

razširitve, želvo kot pretvornik nizov v njihovo grafično podobo in še sam grafični prikaz elementov in končnega modela.

Zahtevnost implementacij lahko tudi merimo v številu grafičnih elementov modela potrebnih za izris, kar močno vpliva na časovno zahtevnost tehnik oz. njihovih implementacij. Implementacija, ki ji število elementov za izris narašča najpočasneje, je SGS. Pri SGS namreč stare elemente "zavržemo", izrišejo se zgolj končni elementi. Čeprav izris s kopičenjem elementov pri SGS ne trpi, se povečuje čas preiskovanja v globino, saj hierarhija oblik, ki poenostavi hranitev oblik in njihovih pozicij, z vsakim novim elementom veča gobino. Pri algoritmih rekurzivne razdelitve višinskih slik končna velikost modela za izris narašča hitreje, saj ima vsak naslednik približno štirikratno velikost prednika. Hkrati najdemo znotraj implementacije HMG tudi algoritem *generiranje napak*, ki ohranja začetno velikost višinske slike. Novo stopnjo naraščanja zahtevnosti izrisa prinašata LSPS in MRCM. Razlog za veliko količino elementov znotraj LSPS je v sami implementaciji, ki poleg stohastičnih pravil ne implementira drugega kakor osnovno verzijo L-sistema. Glavna slabost osnovnegaL-sistema je redudantno kopičenje elementov za podaljševanje dela modela kot npr. podaljševanje debla s kopičenjem majhnih kosov debla. Rešitev problema bi bila implementacija parametričnega L-sistema. Pri MRCM količina elementov narašča eksponentno kot $steviloKopijGradnika^n$, kjer n predstavlja število iteracij. Problem izhaja iz dejstva, da se miniaturne verzije fraktalov še zmeraj računajo kot telesa in ne kakor točke, čeprav so bila skalirana na nivo točk.

Glavni cilj proceduralnega modeliranja je ustvariti sistem, ki končnemu uporabniku z minimalno količino vnešenega uporabniškega dela izdela model, ki bo čim bližji uporabniški željeni podobi. Analiza temelji na konceptni ideji in priloženih implementacijah. Implementacija, ki od uporabnik zahteva najmanj poznavanja implementacije same je gotovo LSPS. L-sistemi namreč uporabljajo gramatiko z nizi, ki ima uveljavljeno standardno gramatiko. Standardna gramatika je definirana tako, da za večino modelov ne potrebujemo razširjati gramatike. To pomeni za uporabnika pri kreiranju

novih modelov zgolj vnos pravil za ustvarjanje nizov in morebitnih dodatnih simbolov abecede. Konceptualno še enostavnejši pristop ima MRCM, kjer mora uporabnik zgolj definirati lokacije izrisa znotraj navodil, vendar mora sam poskrbeti za pravilne vrednosti v transformacijah. Začetnik gotovo najlažje predvidi končno podobo modela pri SGS, saj je velikost omejena na velikost začetne oblike in ker se pravila najlažje gradi po vrstnem redu korakov. Nasprotno LSPS začetniku preko znakovnih nizov ne nudi neke jasne podobe do testiranja, prav tako potrebuje uporabnik nekaj izkušenj s fraktali za uspešno kreiranje novih fraktalnih modelov v MRCM brez težav. Čeprav se SGS izkaže za najbolj predvidljivo implementacijo, zahteva od uporabnika skupno delo potrebno za LSPS in MRCM. Običajno je potrebno zgraditi nekaj novih oblik, saj imamo opravka z gramatiko oblik, kar predstavlja delo enakovredno delu za MRCM. Delo enakovredno delu LSPS je označevanje oblik s simboli in tvorjenje pravil z danimi simboli. HMG je predvsem implementacija algoritmov za ustvarjanje višinskih slik. Uporabnik mora ob želji po novi metodi generiranja sam izvesti celotno implementacijo algoritma z razširitvijo glavnega razreda, da bo metoda na voljo znotraj uporabniškega vmesnika. Prav to predstavlja največjo oviro za preprostost tehnike, saj želimo, da uporabnik ne potrebuje poglobljenega poznavanja implementacije za pravilno uporabo in nadgrajevanje le-te.

Poglavje 7

Zaključek

Cilj diplomske naloge je bil zbrati čim več pogostih tehnik proceduralnega modeliranja. Tekom raziskovanja se izkazalo, da imamo opravka z izredno širokim področjem, ki je slabo podkrepljeno s konkretno literaturo in ki se stalno razvija, tudi v nepredvidljive smeri kot je npr. uporaba genetskega programiranja. Posledica narave področja je bilo skrčenje predstavljenih tehnik na štiri pogoste tehnike, ki nosijo različne koncepte in nudijo konkretnejšo literaturo za raziskovanje. Ogledali smo si *fraktale*, *L-sisteme*, *gramatiko oblik* in *višinske slike*.

Znotraj pregledov smo si podrobneje ogledali ideje za koncepti, dobre in slabe strani tehnik, njihove razširitve in področja uporabe. Zgrajene so bile tudi preproste implementacije kot učni primeri za posamezno tehniko, ki običajnemu uporabniku omogočajo, da z malo truda ustvari nove modele. Izvedena je bila tudi krajša skupinska analiza štirih tehnik, ki je delno grajena na samih implementacijah. Skupinsko analizo je potrebno vzeti zgolj kot okvirno mnenje, saj je z oziranjem na implementacije pristranska. Gotovo bi lahko kakšno težavo implementacije rešili drugače in bi bilo s tem končno mnenje o tehniki boljše. Konkretna skupinska analiza tudi ni mogoča zaradi konceptualne razlike v tehnikah, kar je posledica gradnja tehnik za reševanje različnih problemov modeliranja. Čeprav se vsaka tehnika izkaže zgolj na svojem področju, pridejo tehnike do pravega sijaja šele ob skupni uporabi. Lep

primer sinergije tehnik je okolje *CityEngine*, glej poglavje 4.3.1, ki omogoča uporabo vseh štirih tehnik za gradnjo mest.

Tekom raziskovanja literature so bile odkrite marsikatere tehnike, ki bi lahko zanimale bralca. Primeri takih tehnik, ki imajo na voljo nekaj začetne literature, so *modeliranje z genetskim programiranjem*[37], *Example-based Model Synthesis*[38][39][40] in *Generative modeling*[41].

Literatura

[1] M. F. Barnsley, *Fractals everywhere*. DoverPublications. com, 2012.

[2] D. Allan and B. Dalke, "Fractals," in *Braun Awards Publication, 2011 14th Braun Awards*. Saginaw Valley State University - College of Science, Engineering and Technology, 2011.

[3] D. S. Ebert, "Advanced modeling techniques for computer graphics," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 153–156, 1996.

[4] G. W. Flake, *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptations*. The MIT Press, 1998.

[5] H. O. Peitgen, H. Jürgens, and D. Saupe, *Chaos and fractals: new frontiers of science*. Springer, 2004.

[6] R. Kogovšek, *L-sistemi za simulacijo rastlin v 3D prostoru*, 2013, seminarska naloga pri predmetu Računalniška grafika in tehnologija iger, FRI, Univerza v Ljubljani.

[7] P. Prusinkiewicz, A. Lindenmayer, J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. de Boer, and L. Mercer, *The algorithmic beauty of plants*. Springer-Verlag New York, 1990, vol. 2, no. 6.

[8] M. Marolt, "Proceduralno modeliranje," 2013, zapiski predavanj pri predmetu Računalniška grafika in tehnologija iger, FRI, Univerza v Ljubljani.

[9] (2013, Julij) L-system. Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/L-system

[10] H. Abelson and A. A. Di Sessa, *Turtle geometry: The computer as a medium for exploring mathematics*. the MIT Press, 1986.

[11] M. Žitnik, *L-sistemi za fraktalno generiranje 3D objektov*, 2011, seminarska naloga pri predmetu Računalniška grafika, FRI, Univerza v Ljubljani.

[12] Y. I. Parish and P. Müller, "Procedural modeling of cities," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 2001, pp. 301–308.

[13] J.-E. Marvie, J. Perret, and K. Bouatouch, "The fl-system: a functional l-system for procedural geometric modeling," *The Visual Computer*, vol. 21, no. 5, pp. 329–339, 2005.

[14] M. Favorskaya, A. Zotin, and A. Chunina, "Procedural modeling of broad-leaved trees under weather conditions in 3d virtual reality," in *Intelligent Interactive Multimedia Systems and Services*. Springer, 2011, pp. 51–59.

[15] B. Beneš, O. Št'ava, R. Měch, and G. Miller, "Guided procedural modeling," in *Computer graphics forum*, vol. 30, no. 2. Wiley Online Library, 2011, pp. 325–334.

[16] P. Prusinkiewicz, P. Federl, R. Karwowski, and R. Mech, "L-systems and beyond," *Course notes from SIGGRAPH*, 2003.

[17] (2013, Januar) Lindenmayer fractals - generating fractals - examples. The University of British Columbia. [Online]. Available: http://www.math.ubc.ca/ cass/courses/m308-03b/projects-03b/skinner/ex-generating.htm

[18] J. Gips, "Shape grammars and their uses," Ph.D. dissertation, Stanford University Palo Alto, CA, 1974.

[19] M. Ozkar and S. Kotsopoulos, "Introduction to shape grammars," 2008, zapiski predavanj Shape grammar, School of Architecture and Planning, MIT.

[20] W. W. H. Been, "Engineering shape grammars," *Formal engineering design synthesis*, p. 65, 2001.

[21] F. Hou, Y. Qi, and H. Qin, "Drawing-based procedural modeling of chinese architectures," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 1, pp. 30–42, 2012.

[22] G. Stiny, "Introduction to shape and shape grammars," *Environment and planning B*, vol. 7, no. 3, pp. 343–351, 1980.

[23] P. Wonka, M. Wimmer, F. X. Sillion, W. Ribarsky *et al.*, "Instant architecture," *ACM Transactions on Graphics*, vol. 22, no. 4, pp. 669–677, 2003.

[24] P. Müller, T. Vereenooghe, A. Ulmer, and L. Van Gool, "Automatic reconstruction of roman housing architecture," *Recording, modeling and visualization of cultural heritage*, pp. 287–298, 2005.

[25] M. Lipp, "Interactive computer generated architecture," Ph.D. dissertation, Citeseer, 2007.

[26] P. Mueller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," *Acm transactions on graphics*, vol. 25, no. 3, pp. 614–623, 2006.

[27] K. Dylla, B. Frischer, P. Mueller, A. Ulmer, and S. Haegler, "Rome reborn 2.0: A case study of virtual city reconstruction using procedural modeling techniques," *Computer Graphics World*, vol. 16, p. 25, 2008.

[28] J. Halatsch, A. Kunze, and G. Schmitt, "Using shape grammars for master planning," in *Design Computing and Cognition'08*. Springer, 2008, pp. 655–673.

[29] (2013, November) Heightmap. Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Heightmap

[30] M. Marolt, "Teksturni efekti," 2013, zapiski predavanj pri predmetu Računalniška grafika in tehnologija iger, FRI, Univerza v Ljubljani.

[31] G. Macklem, "Computer landscape generation and smoothing," *Masters Comprehensive Exam*, 2003.

[32] F. K. Musgrave, "Methods for realistic landscape imaging," *Yale University, New Haven, CT*, 1993.

[33] F. K. Musgrave, C. E. Kolb, and R. S. Mace, "The synthesis and rendering of eroded fractal terrains," in *ACM SIGGRAPH Computer Graphics*, vol. 23, no. 3. ACM, 1989, pp. 41–50.

[34] D. S. Ebert, *Texturing & modeling [electronic resource]: a procedural approach*. Morgan Kaufmann, 2003.

[35] T. Polack, *Focus on 3D terrain programming*. CengageBrain. com, 2002.

[36] (2013, November) Generating random fractal terrain. Game Programmer. [Online]. Available: http://www.gameprogrammer.com/fractal.html

[37] T. Schnier and J. S. Gero, "Learning genetic representations as alternative to hand-coded shape grammars," in *Artificial Intelligence in Design'96*. Springer, 1996, pp. 39–57.

[38] P. Merrell and D. Manocha, "Constraint-based model synthesis," in *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*. ACM, 2009, pp. 101–111.

[39] P. Merrell, "Example-based model synthesis," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*. ACM, 2007, pp. 105–112.

[40] E. Yeguas, R. Muñoz-Salinas, and R. Medina-Carnicer, "Example-based procedural modelling by geometric constraint solving," *Multimedia Tools and Applications*, vol. 60, no. 1, pp. 1–30, 2012.

[41] S. Havemann and D. W. Fellner, "Generative mesh modeling." Ph.D. dissertation, University of Braunschweig-Institute of Technology, 2005.