Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jan Juvan

Proceduralno generiranje mestne četrti

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in matematika

Mentor: prof. dr. Branko Šter

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Raziščite pristope proceduralnega generiranja vsebine ter analizirajte algoritme za generiranje vsebine. Analizirajte strukturo urbane forme in obravnavajte njene sestavne elemente. Preglejte področje uporabe ter obravnavajte razloge zanjo. Napišite program za proceduralno generiranje mestne četrti.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Jan Juvan, z vpisno številko **63100398**, sem avtor diplomskega dela z naslovom:

Proceduralno generiranje mestne četrti

S svojim podpisom zagotavljam, da:

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

V Ljubljani, dne 5. julij 2015 Podpis avtorja:

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Proceduralno generiranje vsebine **3**

2.1 Definicija 3

2.2 Prednosti uporabe PCG 5

2.3 Pregled področja 7

2.4 Zaželene lastnosti PCG 8

3 Pristopi **11**

3.1 Preiskovalni algoritmi 14

3.2 Delitev prostora 15

3.3 Pristopi, ki temeljijo na agentih 19

3.4 Celični avtomati 23

4 Urbana forma **29**

5 Generiranje mestne četrti **33**

5.1 Definicija 33

5.2 Analiza sestavnih delov 34

5.3 Generiranje 36

6 Zaključek **53**

Povzetek

Namen diplomske naloge je pokazati prednosti proceduralnega generiranja vsebine ter pregledati trenutno stanje na tem področju.

Zmanšanje stroškov razvoja je odlična taktika za pridobitev konkurenčne prednosti pred ostalimi podjetji, ki se ukvarjajo z razvojem iger, kar lahko dosežemo z odpravitvijo potrebe po večjem številu oblikovalcev. Z uvedbo proceduralnih algoritmov za generiranje vsebine lahko dosežemo prav to. Programer lahko na ta način, ob sodelovanju z umetnikom, doseže veliko več kot večja skupina umetnikov. Ti algoritmi prav tako omogočajo posameznim umetnikom ali manjšim podjetjem ustvariti vsebinsko bogato igro in omogočajo nastanek nepričakovanih form.

Za nastanek programa, ki generira vsebino, je potrebno jasno poznavanje vsebine, v primeru tega dela - mestne četrti. Diplomska naloga obravnava to urbano formo ter analizira njene sestavne dele.

Drugi cilj diplomske naloge je ustvariti program za proceduralno generiranje mestne četrti, katerega rezultat dosega želene lastnosti za uporabo v igrah. Ustvarjeni nivo mestne četrti mora biti igralen, vizualno zadovoljiv ter ne sme dajati občutka, da je bil zgeneriran s strani proceduralnega algoritma, temveč deluje kot delo človeških rok.

Ključne besede: Proceduralno generiranje vsebine, struktura mesta, generiranje urbane forme.

Abstract

The goal of the thesis is to show the advantage of procedural content generation and to review the current situation in this field.

Reducing development costs is an excellent tactic to obtain a competitive advantage over other companies that are engaged in developing games, which can be achieved by removing the need for a larger number of designers. We can achieve that with the introduction of procedural algorithms for generating content. A programmer in cooperation with an artist can, in this way, achieve much more than a large group of artists. These algorithms also allow individual artists or small businesses to create content-rich games and allow for the creation of unexpected forms.

To create a program that generates content, one needs a clear understanding of the content being generated, in the case of this work, city quarter. The thesis deals with the urban form and analysis of its components.

The second objective of this thesis is to write a program for generating city quarters, the result of which achieves the desired characteristics for use in games. Created levels should be playable, visually appealing and not give the impression of being generated by a procedural algorithm, but rather seem to be the work of humans.

Keywords: Procedural content generation, city structure, generating urban forms.

Poglavje 1

Uvod

Po izjemno hitrem vzponu računalniške tehnologije v preteklih desetletjih so računalniške igre deležne vedno več pozornosti. Na trgu se dnevno pojavljajo novi produkti kot tudi nova podjetja, ki se borijo za pridobitev konkurenčne prednosti pred ostalimi. Ljudje so dragi in počasni, in zdi se, da jih potrebujemo vedno več. Dandanes ni nič kaj nenavadno, da komercialno igro razvija veliko število ljudi v času več let. To vodi do situacije, kjer je manj iger donosnih, kar vodi do manjših tveganj na področju razvoja in manjše raznolikosti na trgu iger. Pohitritev procesa razvoja ter odstranitev potrebe po večjem številu delovnih mest je za pridobitev prednosti na trgu nadvse primerno. Del razvoja lahko s kreiranjem ustreznega programa prepustimo računalniku in tako zmanjšamo stroške razvoja. Dodaten razlog za razvoj metod PCG je razumevanje samega snovanja vsebine. Računalniški znanstveniki radi rečejo, da procesa ne razumemo dovolj dobro, dokler ga ne prevedemo v kodo in program teče. Kreiranje programske opreme, ki generira vsebino igre, bi nam lahko pomagalo razumeti postopek, s katerim bi lahko "ročno" ustvarili vsebino in bolje razjasniti omejitve problema, s katerim se ukvarjamo.

Za nas zelo pomemben pojem je "proceduralno generiranje vsebine". Definiramo ga kot kreiranje vsebine igre z omejenim vnosom uporabnika. Ključni pojem tukaj je "vsebina", drug pomemben izraz pa je "igra". V tej nalogi bomo pregledali področje proceduralnega generiranja vsebine ter

obravnavali prednosti njegove uporabe. Prvi algoritmi za proceduralno generiranje vsebine so se pojavili v zgodnjih osemdesetih ter osnovali novo zvrst iger imenovano "rogue-like". Glavna gonilna sila za razvojem algoritmov za generiranje vsebine je bilo premagovanje omejitev velikosti pomnilnika računalnikov. Sicer deležni vedno večje uporabe, se proceduralni algoritmi za generiranje vsebine navadno uporabljajo bodisi v periferni vlogi ali pa je njihov obseg zelo omejen. Trenutne raziskave na tem področju poskušajo razširiti meje dosegljivega in povečati kvaliteto ustvarjene vsebine. V idealnem primeru bi rešitev procesa generiranja vsebine morala biti hitra, zanesljiva, nadzorljiva, ekspresivna in raznolika, vendar v praksi obstajajo nekateri kompromisi med temi lastnostmi.

V sklopu te naloge bomo ustvarili generator mestne četrti za uporabo v igrah. Generiranje mestne četrti je obsežen problem, zato bomo le-to izvedli v nivojih in obravnavali vsakega izmed njih. Za kreiranje takšnega progama je potrebno analizirati urbano formo ter njene sestavne dele. Definirati moramo prostor problema ter opredeliti željen rezultat. Z generatorjem želimo doseči določeno normo nadzora nad procesom, zato moramo paziti na potencialne parametre posameznih algoritmov, ki bodo generator sestavljali.

Poglavje 2

Proceduralno generiranje vsebine

2.1 Definicija

Najprej definirajmo pojem "proceduralno generiranje vsebine" (Procedural Content Generation) - v nadaljevanju PCG. Kreiranje vsebine igre z omejenim vnosom uporabnika je definicija, ki jo bomo uporabili [2]. Z drugimi besedami, PCG se nanaša na računalniško programsko opremo, ki lahko ustvari vsebino igre sama ali skupaj z enim ali več igralcev oz. oblikovalcev. Ključni pojem je "vsebina". V naši definiciji je vsebina večina tega, kar je vsebovano v igri: nivoji igre, zemljevidi, pravila igre, teksture, zgodbe, predmeti, problemi, glasba, itd.. V naši definiciji igralni pogon ne tretiramo kot vsebino. Prav tako pod vsebino ne štejemo računalniško vodenih agentov (Non-Player Character Artificial Intelligence - v nadaljevanju NPC AI). Razlog za to zoženje opredelitve vsebine je, da se na področju umetne inteligence v igrah izvaja več raziskav oziroma študij o uporabi AI metod za vedenje vodenih agentov, kot na področju proceduralnega generiranja vsebine. Medtem ko področje PCG velikokrat temelji na metodah umetne inteligence, jo želimo ločiti od bolj "mainstream" nalog za testiranje algoritmov umetne inteligence, kjer se najbolj pogosto uporablja z namenom, da se nauči igrati

igro. Kot vse opredelitve (razen morda tistih, ki jih najdemo v matematiki), je naša definicija PCG nekoliko arbitrarna in precej mehka na robovih. Mi bomo opredelitev PCG obravnavali kot tako, in so zavedali, da drugi ljudje opredeljujejo pojem drugače. Nekateri raje uporabljajo izraz "generativne metode".

Drugi pomemben izraz je "igra". Igre je izredno težko opredeliti (glej razpravo Wittgenstein-a [21]), zato tega tukaj ne bomo poskušali. Dovolj je reči, da z igrami mislimo takšne stvari, kot so video igre, računalniške igre, družabne igre, igre s kartami, uganke, itd.. Pomembno je, da sistem generiranja vsebine vzame v poštev zasnovo, zmožnosti in omejitve v igri, za katero ustvarja. To razlikuje PCG od tematik, kot je generativna umetnost in drugih tipov računalniške grafike, ki ne upoštevajo posebnih omejitev. Ključna zahteva nastalih vsebin je, da morajo biti igralne – za igralca mora biti omogočeno premagati ustvarjen nivo igre, se povzpeti po ustvarjenemu stopnišču, uporabiti ustvarjeno orožje, itd.. Izraza "postopkovno" in "generiranje" pomenita, da imamo opravka z računalniškimi postopki ali algoritmi, ki nekaj ustvarjajo. Metodo PCG je mogoče zagnati z računalnikom (morda s človeško pomočjo) in produkt bo proizveden. Sistem PCG je sistem, ki vključuje metodo PCG kot enega izmed njenih delov.

Da bo razprava bolj konkretna, bomo navedli nekaj stvari, za katere menimo, da so PCG:

- Programsko orodje, ki ustvarja ječe za akcijsko igro, kot je The Legend of Zelda, brez človeškega vnosa - vsakič ko zaženemo orodje se ustvari nov nivo igre.

- Sistem, ki ustvarja nova orožja za prvoosebno igro v odgovor na akcije kolektiva igralcev, tako da je orožje, ki je igralcu dano, razvita različica predhodnega.

- Program, ki ustvarja zaključeno, igralno in uravnoteženo družabno igro, morda z uporabo nekaterih obstoječih družabnih iger za izhodišče.

- Del igralnega pogona, ki hitro napolni igralni svet z vegetacijo.

- Grafično orodje, ki omogoča, da posameznik ustvari zemljevid za strateško igro, medtem ko stalno ocenjuje zasnovan zemljevid na podlagi igralnih lastnosti in predlaga izboljšave na zemljevidu za bolj uravnotežen in igralen rezultat.

Za preglednost naštejmo še nekaj stvari, ki jih ne uvrščamo pod PCG:

- Urejevalnik zemljevidov za strateško igro, ki omogoča preprosto dodajanje in odstranjevanje predmetov, brez generiranja na lastno pobudo.
- Računalniško voden agent za namizno igro.

2.2 Prednosti uporabe PCG

Zdaj ko vemo, kaj PCG predstavlja, raziščimo razloge za uporabo in razvoj takšnih metod. Izkaže se, da obstaja več pomembnih razlogov. Morda je najbolj očiten razlog za ustvarjanje vsebine odprava potrebe po oblikovalcu za kreiranje te vsebine. Ljudje so dragi in počasni, in zdi se, da jih s časom potrebujemo vse več (To velja vsaj za "AAA" igre; igre, ki se prodajajo po polni ceni po vsem svetu. Nedavni vzpon mobilnih iger je ponovno omogočil donosnost posameznim razvijalcem, vendar se tu prav tako cena razvoja s časom dviga). Od prihoda računalniških iger se število delovnih ur, ki gredo v razvoj uspešne komercialne igre, bolj ali manj stalno povečuje. Danes za igre ni nič kaj nenavadno, da jih razvija na stotine ljudi v obdobju več let. To vodi do situacije, kjer je manj iger donosnih, kar vodi do manjših tveganj na področju razvoja in manjše raznolikosti na trgu iger. Veliko višje plačanih zaposlenih potrebnih v tem procesu je oblikovalcev in umetnikov, ne programerjev. Podjetje, ki razvija igre bi lahko nadomestilo nekatere umetnike in oblikovalce z algoritmi, s čimer bi pridobilo konkurenčno prednost, saj bi igre proizvajalo hitreje in ceneje, hkrati pa ohranjalo kakovost (ta argument je podal legendarni oblikovalec iger Will Wright v svojem predavanju "The Future of Content" na Game Developers Conference leta 2005; pogovor, ki je pomagal oživiti zanimanje za proceduralno generiranje vsebine). Zmanjšanje

števila delovnih mest ni način, kako prodati PCG oblikovalcem in umetnikom. Lahko bi torej argumentirali drugače: algoritmi za generiranje vsebin, še posebej vgrajeni v inteligentna oblikovalska orodja, lahko povečajo ustvarjalnost posameznikov. To bi lahko omogočilo majhnim ekipam brez sredstev velikih podjetij in celo posameznikom, ustvarjanje vsebinsko bogatih iger, z osvoboditvijo od skrbi glede podrobnosti in težkega monotonega dela pri ohranitvi nadzora nad razvojem. Oba argumenta predpostavljata, da je tisto, kar želimo ustvariti, podobno igram, ki so na trgu danes. Prav tako bi PCG metode lahko omogočile tudi povsem nove vrste iger. Če bi imeli programsko opremo, ki bi bila sposobna ustvariti vsebino igre pri hitrosti porabe, načeloma ni razloga, da bi se igra morala končati. Za vsakogar, ki je bil kdaj razočaran nad njegovo najljubšo igro, ki nima več nivojev za raziskovanje, ugank za reševanje, itd., bi bila to nadvse zanimiva možnost. Še bolj vznemirljivo, novo ustvarjena vsebina bi lahko bila prilagojena okusu in potrebam igralca. Z združevanjem PCG z modeliranjem, na primer z izvajanjem merjenj in uporabo nevronskih mrež za modeliranje odziva akterjev na posamezne igralne elemente, bi lahko ustvarili adaptivne igre, ki si prizadevajo doseči čim večji užitek igralcev. Enake tehnike bi lahko uporabili za boljše učenje v igrah ali morda za maksimiranje zasvojljivosti iger.

Še en razlog za uporabo PCG je, da bi nam to pomagalo doseči večjo ustvarjalnost. Ljudje, tudi tisti s kreativno žilico, ponavadi posnemajo drug drugega kot tudi sami sebe. Algoritmični pristopi bi lahko prišli do povsem drugačne vsebine od te, ki bi jo ustvaril človek z nepričakovano a veljavno rešitvijo danega problema generiranja vsebine. Izven tematike iger je to dobro znan pojav v npr. evolucijskem razvoju. In končno, popolnoma drugačen a nič manj pomemben razlog za razvoj metod PCG, je razumeti samo snovanje vsebine. Ustvarjanje programske opreme, ki lahko kompetentno ustvarja vsebino igre, bi nam lahko pomagalo razumeti postopek, s katerim bi lahko "ročno" ustvarili vsebino in bolje razjasniti omejitve problema, s katerim se ukvarjamo. To je iterativen proces, s katerim lahko boljše metode PCG vodijo k boljšemu razumevanju procesa razvoja, ki lahko posledično pripelje do

boljših algoritmov PCG.

2.3 Pregled področja

Premagovanje omejitve velikosti pomnilnika računalnikov je bila ena od glavnih gonilnih sil za razvojem PCG tehnik. Omejene zmogljivosti domačih računalnikov, v zgodnjih osemdesetih letih z malo razpoložljivega prostora za shranjevanje vsebin igre, so prisilile oblikovalce k iskanju drugih metod za ustvarjanje in shranjevanje vsebin. Elite [7] je ena prvih iger, ki reši ta problem s shranjevanjem številke semena, ki ga uporablja za proceduralno generiranje 8 galaksij z 256 planeti z edinstvenimi lastnostmi. Še en klasičen primer zgodnje uporabe PCG je igra Rogue iz zgodnjih osemdesetih. V igri igralec raziskuje sistem ječ, nivoji igre pa se naključno generirajo vsakič, ko se začne nova igra. Tovrstnim igram pravimo "Rogue-like" igre. Avtomatsko generiranje vsebine igre pogosto prihaja s kompromisi; Rogue-like igre lahko proceduralno ustvarjo prepričljivo izkušnjo, vendar večini izmed njih (kot npr. "Dwarf Fortress" [6]) manjka vizualna privlačnost. Nedavno je bilo produceralno generiranje vsebine priča vse večji pozornosti pri komercialnih igrah. Diablo [17] je akcijska igra namišljenih vlog, ki vključuje proceduralno generiranje za ustvarjanje zemljevidov ter postavitev predmetov in pošasti. PCG je osrednji element v igri Spore [15], kjer so modeli, ki jih igralci ustvarijo, animirani s pomočjo proceduralnih animacijskih tehnik [18]. Ta prilagojena bitja se nato uporabijo za zapolnitev proceduralno ustvarjene galaksije. Civilization IV [5] je potezna strateška igra, ki omogoča edinstveno izkušnjo igranja z generiranjem naključnih nivojev. Minecraft [16] je ena od bolj priljubljenih indie iger z obsežno uporabo PCG tehnik za ustvarjanje celotnega sveta in vsebine le tega. Spelunky [1] je še ena znana rogue-like indie igra, ki uporablja PCG za samodejno generiranje variacije nivojev igre. Tiny Wings [9] je še en primer mobilne igre s proceduralnim generiranjem terena in tekstur, kar daje igri drugačen izgled z vsakim zagonom.

2.4 Zaželene lastnosti PCG

Na implementacije metod PCG lahko gledamo kot na rešitve problemov generiranja vsebine. Problem generiranja vsebine bi lahko bil ustvariti novo travo z nizko ravnjo podrobnosti, ki ne izgleda popolnoma čudno v 50 milisekundah ali morda ustvariti resnično izvirno idejo za mehaniko igre v več dneh teka programa. Zaželene in potrebne lastnosti rešitve so drugačne z vsako aplikacijo. Privzamemo lahko le, da običajno obstajajo kompromisi, na primer med hitrostjo in kakovostjo, ter raznolikostjo in zanesljivostjo.

Seznam zaželenih lastnosti PCG:

- Hitrost: Zahteve po hitrosti se močno razlikujejo; od maksimalnega časa nekaj milisekund do več dni, odvisno od tega, ali je proceduralno generiranje izvedeno v času igranja ali v času razvoja igre.

- Zanesljivost: Nekateri generatorji "slepo" generirajo vsebino, medtem ko so drugi sposobni zagotoviti, da vsebina, ki jo ustvarjajo, dejansko izpolnjuje navedene kriterije kakovosti. To je bolj pomembno za nekatere vrste vsebin kot druge, na primer ječa brez izhoda ali vhoda je katastrofalen neuspeh, medtem ko je cvet, ki zgleda malo čudno, daleč od popolnega neuspeha.

- Vodljivost: Obstaja pogosta potreba generatorjev vsebine po obvladljivosti v nekem smislu, tako da se človek ali algoritem (npr. adaptivni mehanizmi), lahko vnaprej odloči glede nekaterih vidikov vsebine, ki bo generirana. Obstaja veliko možnih razsežnosti nadzora, npr. zahtevamo lahko gladek podolgovat kamen, avtomobil, ki je večbarven, nivo igre, ki povzroči skrivnosten občutek in nagradi perfekcionista, ali majhen nabor pravil, kjer naključje ne igra nobene vloge.

- Raznolikost: Pogosto je pomembno, da lahko ustvarimo različen nabor vsebin, da bi se izognili majhnim variacijam in tako monotonemu zgledu. Na eni skrajnosti raznolikosti imamo generator, ki daje vedno znova enak rezultat z razliko obarvanosti ene ploskve na sredini nivoja,

na drugi skrajnosti pa imamo generator, ki vse komponente nivoja razporedi naključno in tako ustvari nepregleden in neigralen nivo. Merjenje raznolikosti ni trivialna tema sama po sebi, oblikovanje generatorja nivojev igre, ki ustvarja raznolike vsebine brez ogrožanja kakovosti, pa še manj.

- Pristnost: V večini primerov bi radi, da naša vsebina ni videti, kot da je bila ustvarjena s proceduralnim generatorjem vsebine.

Poglavje 3

Pristopi

Ječa v resničnem svetu je hladen, temen in grozljiv kraj, kjer so zaprti zaporniki. Ječa v računalniški igri je labirint, kjer pustolovec vstopi v eni točki, zbira zaklade, premaguje pošasti ali se jim izogiba, rešuje soljudi, pade v past in na koncu odide čez izhod na drugem mestu. To pojmovanje ječe izvira iz družabne igre namišljenih vlog "Dungeons and Dragons" in je ključna značilnost skoraj vsake računalniške igre namišljenih vlog ("RPG" – Role playing game), vključno s kultnimi igrami, kot sta seriji "Legend of Zelda" in "Final Fantasy" in nedavni uspeh, kot je "The Elder Scrolls V: Skyrim". Žanr iger "roguelike", poimenovan po igri Rogue iz leta 1980, ima proceduralno generirano ječo; serija Diablo je prav tako zelo odmevna serija iger v tej tradiciji. Zaradi tega tesnega odnosa z uspešnimi igrami in tudi zaradi edinstvenih izzivov glede nadzora v razvoju, so ječe še posebej aktivna in privlačna tema PCG.

Za namen tega dela definiramo pustolovščine in nivoje RPG igre kot labirinte, ki jih sestavljajo večinoma medsebojno povezani izzivi, nagrade in uganke tesno postavljene v času in prostoru, tako da ponudijo visoko strukturiran način napredovanja v igri [13]. Prefinjen pojem napredovanja loči ječe od drugih vrst nivojev. Čeprav je nivo ječe odprt za prosto raziskovanje (veliko bolj kot pri npr. platformnih igrah) ima to raziskovanje tesno vez z napredovanjem glede izzivov, nagrad in ugank, ki ga je predvidel razvija-

lec igre. V nasprotju z npr. platformnimi igrami nivo RPG igre spodbudi prosto raziskovanje sveta, seveda ob strogem nadzoru nad igralčevo izkušnjo, napredovanjem in taktom le-tega (za razliko od odprtih nivojev, kjer je igralec veliko bolj neodvisen). Na primer: igralci lahko svobodno izbirajo svojo pot po ječi med različnimi možnimi in pri tem nikoli ne naletijo na izzive, ki jih ni mogoče premagati pri njihovem trenutnem znanju in trenutnimi zmožnostmi. Oblikovanje ječe je tako kreiranje kompleksnega prostora igre iz želene igralne izkušnje in ne obratno.

V večini pustolovskih in RPG igrah strukturno ječe sestavlja več sob povezanih s hodniki. Medtem ko se originalno izraz "ječa" navezuje na labirint zaporniških celic, se v igrah lahko navezuje tudi na jame, sobe ali druge strukture. Poleg geometrije in topologije ječe vključujejo računalniško kontrolirane karakterje ("NPC" non-player character), dekoracije in druge objekte (npr. zaklad, orožje, hrana).

Proceduralno generiranje ječe se nanaša na generiranje topologije, geometrije in z igranjem povezanih predmetov. Tipičen postopek generiranja ječe je sestavljen iz treh elementov:

- Predstavitveni model: poenostavljena predstavitev ječe, ki zagotavlja enostaven pregled nad končno strukturo.
- Postopek za gradnjo predstavitvenega modela.
- Postopek za ustvarjanje dejanske geometrije ječe iz njenega predstavitvenega modela.

Zgoraj smo pokazali razlike ječe v primerjavi z nivoji platformne igre. Vendar pa obstajajo tudi jasne podobnosti med tema dvema vrstama nivojev. Platformne igre so zaslovele po zaslugi klasičnih iger, kot so "Super Mario Bros", "Sonic the Hedgehog" in nešteto njunih klonov, kot tudi drugih iger, ki so tam jemale navdih. Sodoben primer igre v tej tradiciji, ki vsebuje proceduralno generiranje nivojev, je igra "Spelunky". Prav tako kot ječe, nivoji platformne igre navadno vključujejo prostor za premik, stene, zaklade,

sovražnike in pasti. Vendar pa je v platformni igri igralec običajno omejen z gravitacijo: karakter se lahko premika levo ali desno in pade dol, ampak lahko običajno skoči le majhno razdaljo navzgor. Kot rezultat je prepletanje platform in vrzeli bistven element v besednjaku nivojev platformne igre.

Preglejmo različne metode za proceduralno generiranje ječe in drugih nivojev igre. Čeprav se lahko te metode zelo razlikujejo imajo eno skupno lastnost: vse so konstruktivne, proizvedejo rezultat ob zagonu. Skupno jim je velikokrat tudi to, da so hitre. Nekateri celo uspešno ustvarijo nivo v času izvajanja. Na splošno te metode zagotavljajo le omejen nadzor nad rezultatom in njegovimi lastnostmi. Stopnjo nadzora, ki jo ponuja, je danes zelo pomembna značilnost proceduralnih metod. Pod "nadzor" mislimo možnost, da lahko oblikovalec ali programer namerno usmerja procese generiranja, kot tudi raven truda potrebnega za krmiljenje. "Nadzor" določa tudi do katere mere spreminjanje lastnosti in možnosti krmiljenja daje smiseln rezultat. Ustrezen nadzor zagotavlja, da generator ustvarja konsistentne rezultate (npr. igralne nivoje igre), hkrati pa ohranja tako želene lastnosti kot tudi raznolikost.

Medtem ko PCG metode rastejo v kompleksnosti in se različne metode PCG združujejo v bolj zapletene generativne procese, so bili razviti pomembni parametri, ki pomagajo voditi generator vsebine do določenega rezultata. Kot rezultat se PCG nadzor razvija v smeri večje in naravne interakcije med oblikovalcem in PCG, z uporabo tehnik kot so deklarativno modeliranje [19][10], nadzorljivi agenti [8] in nadzor na osnovi igralnosti [14].

Pogledali bomo več družin proceduralnih tehnik. Zaradi enostavnosti bomo vsako izmed teh tehnik predstavili v okviru enotnega tipa, bodisi ječe ali nivo platformne igre. Prva družina so metode, ki temeljijo na preiskovanju. Sledi družina algoritmov delitve prostora. Dana sta dva različna primera, kako se lahko ječe ustvari z delitvijo prostora. Osnovna ideja je, da rekurzivno razdeliti razpoložljivi prostor v kose in jih nato poveže v ječo. Naslednja družina algoritmov so celični avtomati, ki se izkažejo za enostavno in hitro sredstvo pri ustvarjanju struktur, kot so jame. In končno še metode,

ki temeljijo na agentih.

3.1 Preiskovalni algoritmi

Obstaja veliko različnih pristopov k ustvarjanju vsebine za igre. Pristopi, ki temeljijo na iskanju so bili v zadnjih letih intenzivno raziskani. Pri teh je uporabljen evolucionaren algoriem ali kakšno drugo stohastično iskanje oz. optimizacija za iskanje vsebine z želenimi lastnostmi po prostoru rešitev. Osnovna metafora je razvoj kot proces iskanja. V prostoru rešitev mora obstajati primerna rešitev in če iteriramo dovolj dolgo in izvajamo majhne spremembe, pri čemer zavrnemo škodljive ter ohranimo korektne, bomo eventualno prispeli do želene rešitve. Ta metafora je bila uporabljena za opis procesa razvoja v mnogih različnih disciplinah: na primer, Will Wright (oblikovalec SimCity in The Sims) je v svojem govoru na Game Developers Conference 2004, opisal proces razvoja igre kot iskanje. Bistveni elementi pristopa, ki temelji na iskanju, so naslednji:

Iskalni algoritem

To je "motor" metode, ki temelji na iskanju. Pogosto relativno preprosti evoluacijski algoritmi delujejo dovolj dobro, čeprav včasih obstajajo znatne koristi za uporabo bolj sofisticiranih algoritmov, ki vzamejo omejitve v račun, ali pa so specializirani za generiranje specifičnega dela vsebine.

Predstavitev vsebine

To je predstavitev artefaktov, ki jih želimo ustvariti, npr. nivoje igre, probleme ali morda krilate mačke. Predstavitev vsebine je lahko karkoli, od nabora realnih števil do niza črk. Prikaz vsebine opredeljuje (in s tem tudi omejuje), katere vsebine se lahko ustvarijo in določi, ali je mogoče učinkovito iskanje.

Evaluacijske funkcije

Funkcija vrednotenja je funkcija od artefaktov (individualni del vse-

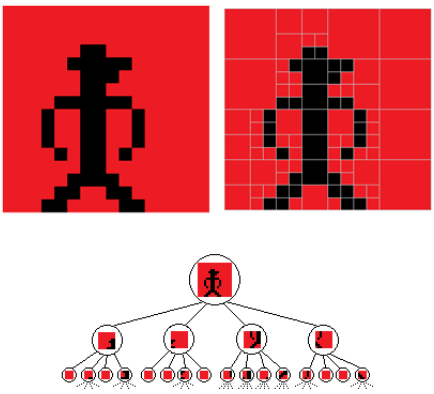
bine) do števila, ki označuje kakovost artefakta. Rezultat evaluacijske funkcije lahko kaže na npr. igralnost nivoja igre ali npr. estetsko privlačnost krilate muce. Izdelava evaluacijske funkcije, ki zanesljivo meri kakovosti artefaktov, je pogosto ena izmed najtežjih nalog pri razvoju PCG metod, ki temeljijo na iskanju.

3.2 Delitev prostora

Pri delitvi prostora delimo dvodimenzionalen ali tridimenzionalen prostor v disjunktne podprostore tako, da vsaka točka leži natanko v enem izmed njih. Podprostore imenujemo tudi celice. Algoritmi za delitev prostora pogosto delujejo hierarhično: vsaka celica v delitvi je nadalje deljena z uporabo istega rekurzivnega algoritma. To omogoča, da se celice razporedijo v delitveno drevo. Poleg tega takšna drevesna struktura podatkov omogoča hitre geometrijske poizvedbe glede katere koli točke v prostoru; to naredi drevesa delitve prostora posebej pomembne za računalniško grafiko, ki omogočajo, na primer učinkovito zaznavanje trčenja.

Najbolj priljubljena metoda za delitev prostora je binarna delitev prostora (BSP), ki rekurzivno ločuje prostor na dve podmnožici. Skozi binarno delitev prostora, lahko prostor predstavimo z binarnim drevesom, imenovanim BSP drevo. Različne variante BSP izberejo drugačne načine delitve prostora na podlagi podobnih pravil. Takšni algoritmi vključujejo štiriška in osmiška drevesa: štiriško drevo razdeli dvodimenzionalen prostor v štiri kvadrante, osmiško drevo pa razdeli tridimenzionalen prostor na osem oktantov. Mi bomo uporabili štiriško drevo na dvodimenzionalni sliki kot najenostavnejši primer, čeprav se uporabljajo enaka načela za osmiška drevesa na tridimenzionalnem prostoru za druge vrste shranjenih podatkov. Štiriško drevo z globino n lahko hrani vsako binarno sliko $2n$ x $2n$ pikslov, skupno število vozlišč drevesa pa je odvisno od strukture slike. Koren vozlišča predstavlja celotno sliko in njeni štirje otroci predstavljajo zgornji levi, zgornji desni, spodnji levi in spodnji desni kvadrante slike. Če imajo piksli v vsakem kvadrantu različne

barve, se ta kvadrant razdeli. Postopek se rekurzivno ponavlja dokler vsak list drevesa ne vsebuje le pike iste barve (glej sliko 3.1).

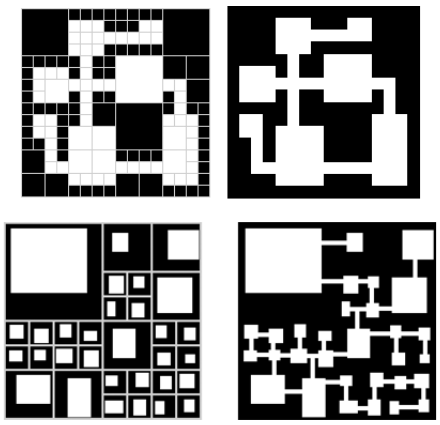


Slika 3.1: Primer štiriškega drevesa binarne slike.

Ko so algoritmi za deljenje prostora uporabljeni v 2D ali 3D grafiki, je njihov namen navadno, da predstavljajo obstoječe elemente, kot so poligoni ali piksli, namesto da ustvarijo nove. Kljub temu je delitev prostora na disjunktne podmnožice primerna rešitev za ustvarjanje sob v ječi, ali na splošno, ločenih območij v nivoju igre. Generiranje ječe preko BSP sledi pristopu, kjer algoritem deluje kot "vsevidni" arhitekt namesto kot "slepi" kopač, kot je pogost z algoritmi, ki temeljijo na agentih. Celotno območje ječe predstavlja koren BSP drevesa in je rekurzivno deljeno, dokler ni izpolnjen zaključni pogoj (kot npr. najmanjše velikosti za sobe). BSP algoritem zagotavlja, da se nobeni dve sobi ne prekrivata in omogoča zelo strukturiran videz ječe.

Kako tesno generativni algoritmi sledijo načelom "tradicionalnih" delitvenih algoritmov vpliva na izgled ustvarjene ječe. Na primer, ječo je mogoče ustvariti z štiriškim drevesom, ki kvadrante izbira naključno, jih deli in ko enkrat konča, vsakemu kvadrantu dodeli vrednost 0 (prazno) ali 1 (soba), pri čemer pazimo, da so vsi prostori povezani. Tak pristop ustvarja zelo simetrične, "kvadratne" ječe. Poleg tega lahko načelo, da mora kvadrant vsebovati en sam element (ali piksle enotne barve, pri primeru slik) sprostimo za namene generiranja ječe; če vsak list vsebuje eno sobo, lahko pa

ima tudi praznih področja, to omogoča sobe različnih velikosti, dokler so njihove mere manjše od mej kvadranta lista.



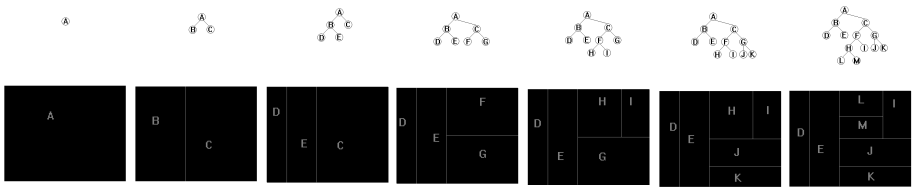
Slika 3.2: Zgornja slika prikazuje ječo ustvarjeno s pomočjo štiriškega drevesa, kjer vsaka celica sestoji iz popolnoma praznega prostora (črna) ali sobe (bela).

Ti prostori se nato lahko povežejo drug z drugim, z uporabo procesov, ki temeljijo na naključju ali naboru pravil. Kljub temu dodatku bodo ječe še vedno verjetno imele precej strukturiran videz.

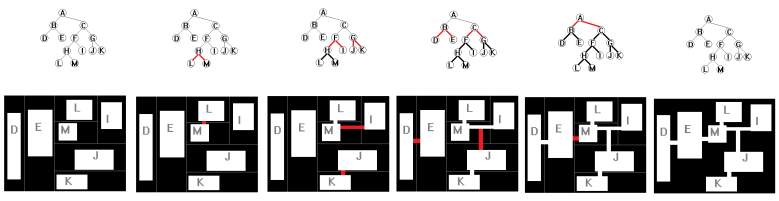
Poglejmo še bolj stohastični pristop, ki ohlapno temelji na BSP tehnikah. Imamo območje za našo ječo s širino w in višino h, ki je shranjena v korenu BSP drevesa. Prostor lahko razdelimo vzdolž navpične ali vodoravne črte in posledično za posamezne celice ni potrebno, da so enake velikosti. Drevo generiramo tako, da v vsaki iteraciji naključno izberemo vozel in delimo celico po naključno izbrani vodoravni ali navpični črti. Vozel ne delimo dalje, če presega najmanjšo dovoljeno velikost (za ta primer vzemimo minimalno širino w / 4 in minimalno višino h / 4). Na koncu vsaka celica vsebuje eno sobo; vogali vsakega prostora so izbrani stohastično tako, da prostor leži znotraj particije in ima sprejemljivo velikost (torej ni premajhen). Ko je drevo ustvarjeno, dodamo hodnike s povezovanjem otrok istega starša med seboj. Spodaj je podana psevdo-koda algoritma. Slik 3.3 in 3.4 prikazujeta tak postopek generiranja ječe.

Postopek generiranja ječe:

1. Začnemo s celotnim območjem ječe (koren BSP drevesa)
2. Razdelimo območje vzdolž vodoravne ali navpične črte
3. Izberi eno od dveh novonastalih celic
4. Če je ta celica večja od minimalne sprejemljive velikost: gremo na korak 2 (trenutna celica je nov prostor, ki bo deljen)
5. Izberemo drugo celico in gremo na korak 4
6. Za vsako celico:
7. Ustvarimo prostor znotraj celice z naključno izbiro dveh točk (zgoraj levo in spodaj desno) znotraj svojih meja
8. Začenši z najnižjimi sloji, povežemo sobe istega starša s hodniki
9. Ponavljamo 9, dokler otroci korenskega vozlišča niso povezani

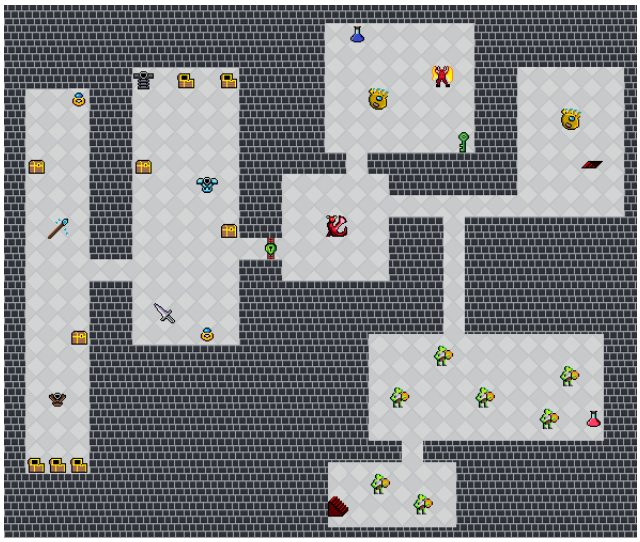


Slika 3.3: Stohastična delitev območja ječe.



Slika 3.4: Porazdelitev sob in hodnikov za ječo iz slike 3.3.

Čeprav je bila binarna delitev prostora tukaj primarno uporabljena za generiranje sob brez medsebojnega prekrivanja, je treba opozoriti, da se hierarhija BSP drevesa lahko uporablja za druge aspekte generiranja ječe. Prejšnji primer na sliki 3.4 je pokazal, kako se povezanost sob lahko določi z BSP drevesom: uporaba hodnikov za povezave sob istih staršev zmanjšuje možnost prekrivanja ali sekanja hodnikov. Poleg tega lahko vozlišča drevesa, ki niso listi, predstavljajo skupek sob z istimi lastnostmi; na primer, del ječe lahko vsebuje močnejše pošasti, ali pošasti, ki so bolj odporne na magijo. Poleg podobnih lastnosti, imajo lahko te skupine sob le en vhod do preostalih delov ječe; To omogoča kontrolo igralčevega napredovanja po ječi kot tudi tematsko obarvane in ločene dele ječe. Primer enostavne ječe vidimo na sliki 3.5.



Slika 3.5: Primer ječe iz slike 3.4, z uporabo particij za namene tematskega deljenja sob glede na njihovo vsebino.

3.3 Pristopi, ki temeljijo na agentih

Pristopi za generiranje ječe, ki temeljijo na agentih običajno sestojijo iz posameznega agenta, ki koplje tunele in ustvarja sobe v zaporedju. V nasprotju z algoritmi delitve prostora je za pristope, ki temeljijo na agentih, bolj verje-

tno, da ustvarijo organske in morda kaotične ječe namesto lepo organiziranih in strukturiranih ječ. Videz ječe je v veliki meri odvisen od obnašanja agenta: agent z visoko stopnjo stohastičnosti bo ustvaril zelo kaotične ječe, medtem ko se lahko agent z vpogledom naprej izogne sekanju hodnikov ali sob. Opozoriti je treba, da je vpliv parametrov obnašanja agenta na videz ustvarjene ječe težko uganiti brez obsežnega testiranja; tako so pristopi, ki temeljijo na agentih veliko bolj nepredvidljivi od pristopov delitve prostora. Poleg tega ni zagotovila, da bodo pristopi, ki temeljijo na agentih, ustvarili ječo brez prekrivanja sob med seboj, ali ječo, ki se ne razteza le v kotu prostora namesto po celoti. Naslednji odstavki bodo prikazali uporabo pristopov na osnovi agentov pri generiranju ječe.

Obstaja neskončno načinov vedenja za agenta pri ustvarjanju ječe, kar posledično povzroči ogromno različnih rezultatov. Najprej bomo v vpogled vzeli visoko stohastično, "slepo" metodo. Agent začne v neki točki ječe in naključna smer je izbrana (gor, dol, levo ali desno). Agent začne kopati v tej smeri in vsaka izkopana ploščica ječe se nadomesti s ploščico hodnika. Po izkopu prve je verjetnost 5%, da bo agent spremenil smer (izbira nove, naključne smeri) in še verjetnost 5% da bo agent ustvaril sobo naključne velikost (v tem primeru velikosti med 3 in 7 v obeh dimenzijah). Za vsako izkopano ploščico, ko se agent premika v isti smeri, verjetnost za spreminjanje smeri raste za 5%. Prav tako za vsako izkopano ploščico, kjer agent ne ustvari sobe, verjetnost, da agent postavi sobo raste za 5%. Ko agent enkrat spremeni smer, je verjetnost za spremembo smeri spet na 0%. Prav tako, ko agent doda sobo, verjetnost da agent doda sobo pade na 0%. Slika 3.6 prikazuje primer poteka algoritma, spodaj pa je navedena tudi psevdo-koda za ta algoritem.

Potek algoritma:

1. Določimo možnost spreminjanja smeri $Pc = 5$
2. Določimo možnost dodajanja sobe $Pr = 5$
3. Postavimo agenta na ploščico ječe in izberemo naključno smer

4. Kopamo v tej smeri
5. Naključno izberemo število Nc med 0 in 100
6. Če $Nc < Pc$: naključno spremenimo smer agenta in nastavimo $Pc = 0$
7. Sicer: nastavite $Pc = Pc + 5$
8. Naključno izberemo število Nr med 0 in 100
9. Če $Nr < Pr$: ustvarimo sobo naključne širine in višine, med 3 in 7, okoli trenutne pozicije agenta in nastavimo $Pr = 0$
10. Sicer nastavimo $Pr = Pr + 5$
11. Če ječa ni dovolj velika se vrnemo na korak 4

Slika 3.6: Kratek prikaz stohastičnega, slepega agenta.

Da bi se izognili pomanjkanju nadzora iz predhodnega stohastičnega pristopa, ki lahko povzroči prekrivanja sob in slepe ulice, je lahko agent malo bolj obveščen o celotnem videzu ječe in ima vpogled naprej (ali bi dodana soba povzročila prekirivanje sob ali hodnikov). Prav tako ni potrebe po poskusu spreminjanja smeri v vsakem koraku, da bi se izognili čudnim hodnikom.

V drugem primeru si bomo pogledali primer manj stohastičnega agenta z vpogledom naprej. Tako kot prej, agent začne na naključno izbranem mestu v ječi. Agent preveri ali bi dodajanje sobe v trenutnem položaju povzročilo sekanje z obstoječimi sobami. Če vse možne velikosti kreirane sobe povzročijo sekanje z obstoječimi, agent izbere smer in kopa tam, kjer hodnik ne povzroči sekanja z sobami. Algoritem se ustavi, ko se agent ustavi na lokaciji, kjer ni prostora ne za hodnik, ne za sobo, brez povzročitve sekanja z obstoječimi elementi. Slika 3.7 prikazuje primer teka algoritma, spodaj pa je dana psevdokoda za ta algoritem.

1. Postavimo agenta na ploščico ječe
2. Izberemo pomožne spremenljivke $Fr = 0$ in $Fc = 0$
3. Za vse možne velikosti sob, kjer potencialni prostor ne seka obstoječih sob:
4. Postavimo sobo, nastavimo $Fr = 1$ ter izstopimo iz zanke
5. Za vse možne hodnike v vse smeri, ki ne sekajo obstoječih sob:
6. Dodamo hodnik, nastavimo $Fc = 1$ ter izstopimo iz zanke
7. Če je $Fr = 1$ ali $Fc = 1$ se vrnemo na korak 2

Slika 3.7: Kratek prikaz obveščenega agenta z vpogledom naprej.

Potrebno je opozoriti, da primera s slepim in obveščenim agentom prikazujeta zelo naivna, preprosta pristopa. Sliki 3.6 in 3.7 prikazujeta v veliki meri najslabši možni scenarij z rezultatom, kjer pride bodisi do prekrivanja sob ali predčasne zaključitve generiranja. Medtem ko kompleksnejše dopolnitve algoritmov preprečijo mnoge od teh problemov še vedno ostaja dejstvo, da je težko predvideti takšne težave brez obsežnega preizkušanja oz. testiranja. To je lahko zaželena lastnost saj lahko nekontroliran algoritem ustvari bolj organične in realne jame in zmanjša predvidljivost ječe za igralca, lahko pa ustvari tudi nivoje, ki niso igralni ali zanimivi. Prav tako imajo v primerih, prikazanih zgoraj, agentovi parametri velik vpliv na igralnost ter vizualno vrednost v ustvarjeni ječi in spreminjanje teh parametrov za boljši rezultat ni trivialna naloga.

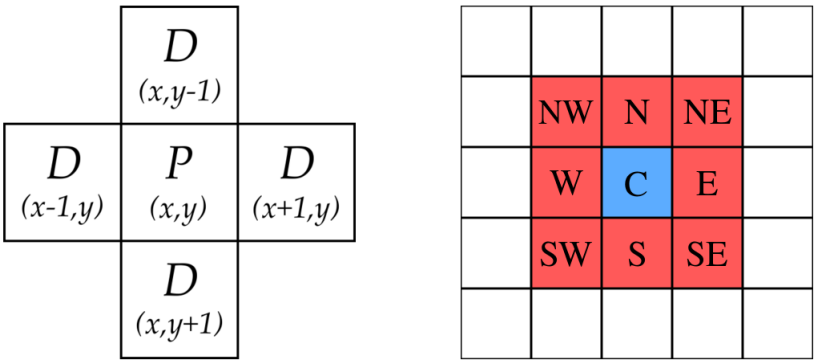
3.4 Celični avtomati

Celični avtomat (cellular automaton) je diskreten računski model. Celični avtomati so obširno raziskana tema v računalniški znanosti, fiziki in celo nekaterih vejah biologije, kot so modeli rasti, razvoja, fizičnih pojavov, itd. Medtem ko so bili celični avtomati deležni številnih publikacij, je osnovni koncept pravzaprav zelo preprost in ga je mogoče razložiti v nekaj točkah in sliki ali dveh.

Celični avtomat sestoji iz n-dimenzionalne mreže, nabora stanj in nabora tranzicijskih pravil. Večina celičnih avtomatov je bodisi enodimenzionalnih (vektorji) ali dvodimenzionalnih (matrike). Vsaka celica je lahko v enem od več stanj. V najpreprostejšem primeru so lahko celice v dveh stanjih. Porazdelitev stanj celic na začetku poskusa (v času t_0) je začetno stanje celičnega avtomata. Od takrat naprej avtomat teče v diskretnih korakih, ki temeljijo na danem naboru pravil. Ob vsakem času t vsaki celici določimo novo stanje, ki temelji na stanju celice in stanju vseh celic v njeni soseščini v času $t - 1$.

Soseska določa, katere celice okrog določene celice C vplivajo na prihodnost stanja celice C. Za enodimenzionalne celične avtomate je soseska določena s svojo velikostjo, tj. koliko celic na levo ali desno se soseska razteza. Za dvodimenzionalne avtomate sta najbolj pogosta tipa sosesk: Moore-ova soseska in Von Neumann-ova soseska. Obe soseski imata lahko velikost ena ali več. Moore-ova soseska je kvadrata: soseska velikosti ena je sestavljena iz osmih celic okoli celice C, vključno z tistimi, ki si z C delijo le kot. Von Neumann-ova soseska je kot križ s centrom v C: soseska velikosti 1 je sestavljena iz štirih celic okoli C; zgoraj, spodaj, levo in desno (glej sliko 3.8).

Število možnih konfiguracij soseščine je enako številu stanj celice na število celic v soseščini. Ta številka lahko hitro postane velika; primer dvodimenzionalnega avtomata z Moore-ovo sosesko velikosti dve ima $2^{25} = 33554432$ konfiguracij. Za majhne soseske običajno opredelimo prehodna pravila kot tabelo, kjer je vsaka možna konfiguracija soseske povezana z enim prihodnjim stanjem, pri velikih soseskah pa prehodna pravila običajno temeljijo



Slika 3.8: Dve vrsti sosesk za celični avtomat. Von Neumann-ova na levi in Moore-ova na desni.

na razmerju celic, ki so v vsakem od stanj.

Celični avtomati so zelo raznoliki in za več vrst se izkaže, da so turingovi. Pojavile so se ideje, da bi lahko ti avtomati oblikovali podlago za nov način razumevanja narave skozi modeliranje "od spodaj navzgor" (bottom-up modelling) [20]. Seveda pa se bomo v tem delu večinoma ukvarjali s tem, kako jih je mogoče uporabiti za proceduralno generiranje vsebine.

V objavi iz leta 2010, Johnson et al. opisuje sistem za generiranje neskončnega sistema jam z uporabo celičnih avtomatov [11]. Motivacija za tem je bila ustvariti neskončno igro plazenja po jamah, z okoljem, ki se razprostira v nedogled in neopazno v vse smeri. Dodatna omejitev pri tem je, da morajo jame izgledati organsko, brez ravnih robov in ostrih kotov. Medij za shranjevanje resnično neskončne jame ne obstaja, zato mora biti vsebina ustvarjena v času izvajanja, medtem ko igralci raziskujejo nova področja. Igra prikazuje okolje zaslon za zaslonom, kar ustvari časovno okno nekaj sto milisekund za kreiranje novega prostora.

Ta metoda uporablja naslednje štiri parametre za nadzor procesa generiranja:

- Odstotek skalnih celic (nedostopno območje)

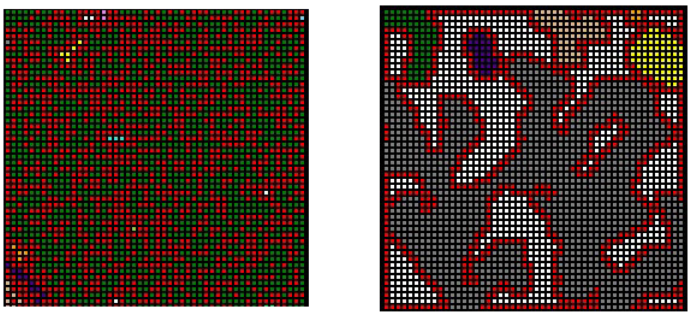
- Število iteracij celičnega avtomata
- Mejna vrednost soseske, za rojstvo skale ($T = 5$)
- Število celic v soseski

  Prostor predstavlja mreža, kjer lahko vsaka celica v enem izmed dveh stanj: prazno ali skala. Sprva je mreža prazna. Generiranje posamezne sobe deluje tako:

- Mreža je naključno posuta s skalami, za vsako celico obstaja verjetnost r (npr. 0,5), da je skala. Rezultat tega je pretežno enakomerna porazdelitev skal v prostoru.
- Celični avtomat na mreži iterira n (npr. 2) korakov. Edino pravilo za ta celični avtomat je, da se celica spremeni v skalo v naslednjem koraku, če je vsaj T (npr. 5) njenih sosednih celic skal, sicer se bo spremenila v prazen prostor.
- Zaradi estetskih razlogov so skalne celice, ki mejijo na prazen prostor označene kot celice stene, ki so funkcionalno enake kot skalne celice, vendar z drugačnim izgledom.

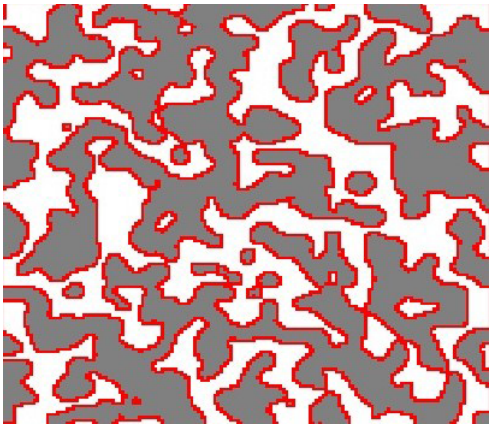
Ta preprost postopek ustvari presenetljivo realističen sistem jam. Slika 3.9 prikazuje primerjavo med začetnim stanjem avtomata in rezultatom po več iteracijah celičnega avtomata.

Medtem ko metoda generira en prostor, igra zahteva številne med sabo povezane prostore. Ustvarjena soba morda nima nobenih odprtin izven pravokotnika, ki jo omejuje, se pravi ni nikakršnega zagotovila, da bosta sosednji sobi povezani. Zato med generiranjem prostora sočasno generiramo tudi najbližje sosede. Če povezava med sosednjima sobama ne obstaja ustvarimo tunel med njima, na mestu, kjer sta si najbližje. Na prosotru vseh sob in tunelov nato izvedemo dodatne iteracije celičnega avtomata s čimer se izognemo ravnim robovom. Slika 3.10 kaže rezultat tega procesa v obliki devetih



Slika 3.9: Generiranje jame: Primerjava med rezultatom avtomata in naključnim začetnim stanjem ($r = 0{,}5$ v obeh primerih); Parametri celičnega avtomata: $n = 4$, $M = 1$, $T = 5$. Celice zidu so predstavljene z rdečo, celice skale pa z belo barvo.

sob, ki so neopazno povezane. Ta proces generiranja je zelo hiter in lahko ustvari vseh devet sob v manj kot milisekundi na sodobnem računalniku.



Slika 3.10: Generiranje jame: Sistem jam ustvarjen z celičnim avtomatom. Celice zidu so predstavljene z rdečo, celice skale pa z belo barvo. Siva območja predstavljajo tla. ($M = 2$; $T = 13$, $n = 4$; $r = 50\%$)

Če povzamemo, ta metoda uporablja naslednje štiri parametre za nadzor procesa generiranja jame:

- Odstotek celic skal (nedostopno območje)
- Število iteracij celičnega avtomata
- Mejna vrednost soseske, za rojstvo skale ($T = 5$)
- Število celic v soseski

Zaključimo lahko, da je število parametrov majhno in dejstvo, da so relativno intuitivni, je prednost pristopov kot je Johnson-ov. Vendar pa je to tudi ena od slabosti pristopa: za oblikovalce kot programerje ni enostavno v celoti razumeli vpliv, ki ga ima en sam parameter na proces generiranja, saj vsak parameter vpliva na več značilnosti rezultata. Nemogoče je ustvariti sistem jam, ki dosega specifične zahteve, kot so število sob z določeno mero povezanosti. Zato so igralne karakteristike nekoliko odtujene od kontrolnih parametrov. Vsako konkretno povezavo med metodo generiranja in igralnimi karakteristikami bi spet dosegli le z obsežnim testiranjem.

Poglavje 4

Urbana forma

Osredotočimo se na rezultat, ki ga želimo doseči. Ker je cilj tega dela generirati mesto si oglejmo samo mesto in njegovo strukturo. Najprej si oglejmo definicijo urbane forme.

Urbano formo lahko opredelimo kot prostorski vzorec človekovih dejavnosti v določenem trenutku. Urbana forma je izraz, ki opisuje fizične elemente v mestu. Nanaša se na ureditev, funkcijo in estetsko kvaliteto stavb in ulic, ki prekrivajo prostor mesta. Na splošno lahko urbane forme klasificiramo na različne načine. Tukaj se bomo osredotočili na tri ključne karakteristike: gostota, povezanost in dostopnost. Te karakteristike niso neodvisne druga od druge čeprav merijo različne lastnosti urbane forme, in vsaka vpliva na razvoj urbanih form drugače.

Urbana gostota

Urbana gostota je merilo enote interesa (npr. zaposlovanja ali poseljevanja) na enoto površine (na primer, blok, soseska, mesto) Obstajajo številna merila gostote, tri najbolj pogosta so: gostota prebivalstva (tj. št. prebivalcev na enoto površine), gostota pozidane površine (tj. površina stavbnih zemljišč na enoto površine), in gostoto zaposlovanja (tj. število delovnih mest na enoto površine).

Povezanost

Povezanost se nanaša na gostoto ulic in njihovo obliko. Pogosta merila

povezanosti vključujejo gostoto križišč, velikost blokov prostora med ulicami ali število križišč na kilometer [12]. Kjer je povezanost ulic visoka (velika gostota križišč z manjšimi bloki, ki omogočajo pogoste spremembe v smeri, pri potovanju skozi mesto) obstaja običajno pozitivna korelacija z izbiro hoje kot načina potovanja in posledično prihaja do manj onesnaževanja.

Dostopnost

Dostopnost lahko opredelimo kot dostop do zaposlitve, stanovanja, storitev in krajev v mestu [3] [4]. To lahko razumemo kot kombinacijo razdalje in časa potovanja. Pogosta merila dostopnosti vključujejo centraliziranost prebivalstva, dostopnost delovnih mest z javnimi prevoznimi sredstvi in oddaljenost od centra mesta.

Seveda pa mesta ne nastanejo iz danes na jutri temveč se večajo in spreminjajo skozi čas. Na sam razvoj vpliva vrsta faktorjev. Mesto prav tako ne živi samo zase temveč ga naseljuje množica ljudi in posledično ga gradijo različni interesi. Rast mesta skozi čas, ker je vodena s strani meščanov kot celote, upošteva zgoraj navedene lastnosti in se navadno izboljšuje na podlagi zgoraj omenjenih meril, vendar zaradi lokalnih interesov, človeške umrljivosti in kompromisov, ki nastajajo zaradi tega, končen rezultat do neke mere odstopa od maksimalnega potenciala, največje efektivosti. Urbanih form je ogromno, saj definicija dopušča veliko mero variranja.

Generiranje urbane forme po celotni zalogi vrednosti njenih lastnosti in variacij pri razvoju skozi čas predstavlja ogromen problem. Ker generiramo mesto za uporabo v igrah je potrebno opozoriti, da bi visoka mera efektivnosti strukture mesta škodovala izgledu (raziskovanje mesta v obliki šahovnice ni precej zanimivo). Naš cilj je večja razvejanost ulic, manj simetrična forma in bolj organski videz. V tem diplomskem delu se bomo osredotočili na manjšo, omejeno podmnožico urbanih form. Generirali bomo večinsko rezidentsko mestno četrt (z visoko mero gostote pozidane površine), ki je del starega mesta. (npr. del starega obmorskega mesta z ozkimi prepletajočimi se ulicami in gosto zgručenimi stavbami). Generator mora biti z vsakim zagonom

sposoben generirati naključno mestno četrt, neenostavne in človeku zanimive strukture, ki ne delujejo kot rezultat proceduralnega algoritma temveč kot delo človeških rok.

Poglavje 5

Generiranje mestne četrti

5.1 Definicija

Cilj tega diplomskega dela je generirati rezidentno mestno četrt. Vprašati se moramo, kaj je mestna četrt in kakšna je njena struktura. Za namene tega dela bomo mestno četrt definirali kot skupek stavb, poseljenih z sobami in hodniki, zgručanih in prepletenih z ulicami v omejenem prostoru. Ključni deli naše četrti so naslednji elementi:

Prostor

Tridimenzionalna ali dvodimenzionalna predstavitev prostora, ki vsebuje informacije o sestavnih elementih (stavbe z sobami, ulice, zid), ki lahko služi kot prikaz rezultata enega zagona programa. V našem primeru bo to več slojev dvodimenzionalnih mrež celic (pritličje, prvo nadstropje, itd.).

Zid

Meja prostora v katerem bodo postavljene stavbe. Prostor, ki ga omejuje je lahko enostaven (pravokotnik, kvader) ali pa ima kompleksnejšo obliko, vsekakor pa mora biti povezan. Ker generiramo četrt za uporabo v igrah, je zid ločnica, ki omejuje prostor premikanja igralca. V našem primeru bo zid določen z vrsto celic v pravem stanju. Te celice

v vsaki od mrež ločujejo celice stavb in ulic od nedostopnih celic.

Stavbe

Informacija o prostoru, ki ga stavba pokriva ter informacije o elementih, ki ga polnijo (sobe, hodniki). V našem primeru bo stavba določena z množico celic v stanju sob ali hodnika.

Ulice

Kot smo omenili so ulice četrti med seboj prepletene poti, ki ovijajo stavbe in jih povezujejo med seboj. V našem primeru bodo ulice določene z vrsto celic v določenem stanju.

Sobe, Hodniki

Sestavni deli stavb, ki jo popolnoma pokrivajo. V našem primeru bodo sobe in hodniki določeni z vrsto celic v pravem stanju. Ker želimo ločiti sobe in hodnike med sabo, ter želimo informacijo, katere sobe in hodniki formirajo določeno stavbo, ima vsaka soba ter hodnik svoje stanje.

Zapis ki ga bomo uporabili, bo definiral prostor iskanja oz. rasti za naše algoritme. Predstavimo ga z tridimenzionalno matriko MxAxB, kjer je M število nadstropij, A širina ter B višina. Vsaka celica matrike vsebuje vrednost, ki predstavlja stanje. (0 - nedostopen prostor, 1 - zid, 2 - ulica, 3 - soba1, 4 - soba2, 5 - hodnik1, itd.) Poleg matrike za popolen prikaz potrebujemo tudi listo stanj celic, ki tvorijo stavbo. (Stavba1 - 3, Stavba2 - 3,4, itd.).

5.2 Analiza sestavnih delov

Sočasno generiranje celotnega prostora ter vseh elementov v prostoru pri pogoju da bo nivo igralen, z vsakim zagonom uspešno ustvarjen ter vizualno zadovoljiv je praktično nemogoč problem. Seveda pa lahko problem razbijemo na manjše obvladljive probleme ter izvedemo proceduralno generiranje

po nivojih. Začeli bomo z praznim prostorom ter z vsakim nivojem generiranja dobili nov prostor, ki ga bomo preiskovali oz. po njem rastli (naslednji nivo generiranja). Zaradi odvisnosti med posameznimi nivoji moramo zaporedje generiranja previdno določiti. Vidimo, da obstajajo odvisnosti med oblikami stavb, ulic in obliko prostora, ki ga omejuje zid, ter med oblikami sob, hodnikov in oblikami stavb. Poglejmo si kaj pričakujemo od oblik posameznih elementov.

Prostor stavb in ulic, ki ga omejuje zid

Za ta prostor ne moremo reči prav veliko. Variacij je ogromno, edina omejitev je, da mora biti povezan. Lahko bi zahtevali neko mero konveksnosti. Za samo obliko, preden je poseljena z stavbami, ne moremo reči ali je korektna ali nekorektna zato lahko generiranje zidu uvrstimo v končne nivoje generiranja celotne četrti.

Stavbe

Za razliko od prostora, ki ga omejuje zid, imamo za obliko stavbe več norm in pričakovanj, kar močno zmanjša število korektnih variacij. Da obliko prepoznamo kot stavbo morajo bit dimenzije le-te v določenih mejah, prostor mora biti povezan, sama oblika pa ne sme imeti hodniku podobnih izrastkov (serije celic z natanko dvema sosednima celicama enakem stanju). Ker delamo na dvodimenzionalni mreži si lahko obliko stavbe predstavljamo kot pravokotnik kateremu odrežemo več manjših pravokotnikov. Minimalno in maksimalno število odrezanih pravokotnikov sta prav tako nastavljiva parametra pri generiranju. Generiranje obilke stavbe predstavlja lažji in rešljiv problem v primerjavi z generianjem prostora, ki ga omejuje zid, vendar je, zaradi še vedno široke množice variacij, netrivialen. Ker je oblika stavbe v odvisnosti z obliko prostora, ki ga omejuje zid, kot tudi z oblikami sob, ki jo polnijo, je generiranje le-te primerno izvesti v začetnih nivojih generiranja četrti.

Sobe

Veliko bolj določljive oblike. V primerjavi s stavbami so te oblike bolj

enostavne. Predstavljamo si jih lahko na enak način kot pri stavbah, maksimalno število odrezanih pravokotnikov pa je manjše, kot tudi same dimenzije sobe in tako je število variacij manjše kot pri sobah. Ker je oblika sobe v odvisnosti z obliko stavbe in je od vseh elementov ena bolj prepoznavnih oblik, generiranje le teh uvrstimo v začetne nivoje generiranja četrti.

Hodniki, ulice

Zelo, mogoče najbolj, prepoznavne oblike zaradi unikatnih lastnosti. Množica variacij je ogromna, omejitve pa manjše kot pri prostoru, ki ga omejuje zid. Generiranje ulic in hodnikov predčasno bi bilo zelo težko, če ne nemogoče. Ulice se lahko prekrivajo, hodniki pa v manjših stavbah včasih niso potrebni. Ker so močno odvisni od oblike sob in stavb je primerno generiranje teh uvrstiti v končne nivoje generiranja četrti.

5.3 Generiranje

Pri generiranju mestne četrti bomo sledili naslednjem zaporedju:

1. Generiranje sob
2. Generiranje stavb
3. Zapolnitev stavb sobami ter dodajanje hodnikov
4. Gručenje stavb v gosto formacijo z dodajanjem ulic
5. Definiranje zidu, ki omejuje četrt

5.3.1 Generiranje sob

Oblika sobe, kot smo omenili, je precej prepoznavna oblika. Če si obliko sobe predstavljamo kot pravokotnik kateremu odrežemo več manjših, lahko zanjo postavimo par omejitev pri generiranju. Oblika sobe bo prepoznavna za

število odrezanih pravokotnikov med minimalno 0 in maksimalno 3. Oblike, ki prekoračijo omejitve imajo precej kompleksno obliko, ki sicer morda vizualno deluje lepo vendar preveč odstopa od naših pričakovanj. Poleg števila izrezanih pravokotnikov so v primerjavi s stavbami manjše tudi same dimenzije sobe. Rečemo lahko, da delamo z enostavnimi oblikami, katerih variacij ne bo prav veliko. Upoštevati moramo seveda simetrijo in rotacije, da se izognemu ponavljanju oblik. Najenostavnejši način za generiranje teh enostavnih oblik je, da različnim začetnim pravokotnikom, primernih dimenzij, odrežemo manjše pravokotnike v enem, dveh ali treh od njegovih kotov. Glej sliko 5.1. Lahko bi rezali tudi v četrtem kotu ali celo po robovih vendar se bomo temu, zaradi predhodno navedenih omejitev pri pričakovanjih, izognili. Rezultat je velika množica primernih in enostavnih kot tudi v večini vizualno zanimivih sob. Za zagotovilo korektnosti oblik sob uporabimo na koncu še evaluacijske funkcije, s katerimi odstranimo nedovoljene oblike na podlagi nabora pravil.

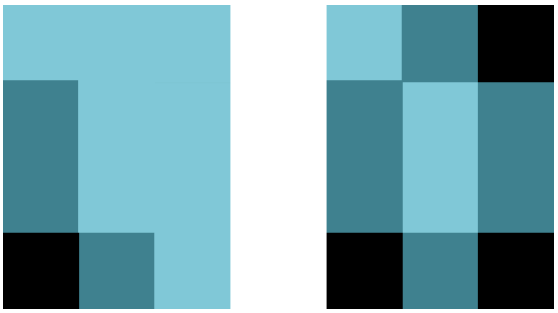
Psevdo koda generiranja oblik sob:

1. Za vsako dimenzijo prostora med minimalno in maksimalno:
2. Za vsako število odrezanih pravokotnikov med 0 in 3:
3. Za vsako dimenzijo odrezanih pravokotnikov med minimalno in maksimalno:
4. Za vse možne razporeditve odrezanih pravokotnikov po kotih prvotnega pravokotnika:
5. Začetnemu pravokotniku odrežemo manjše pravokotnike in ga shranimo v množico rešitev

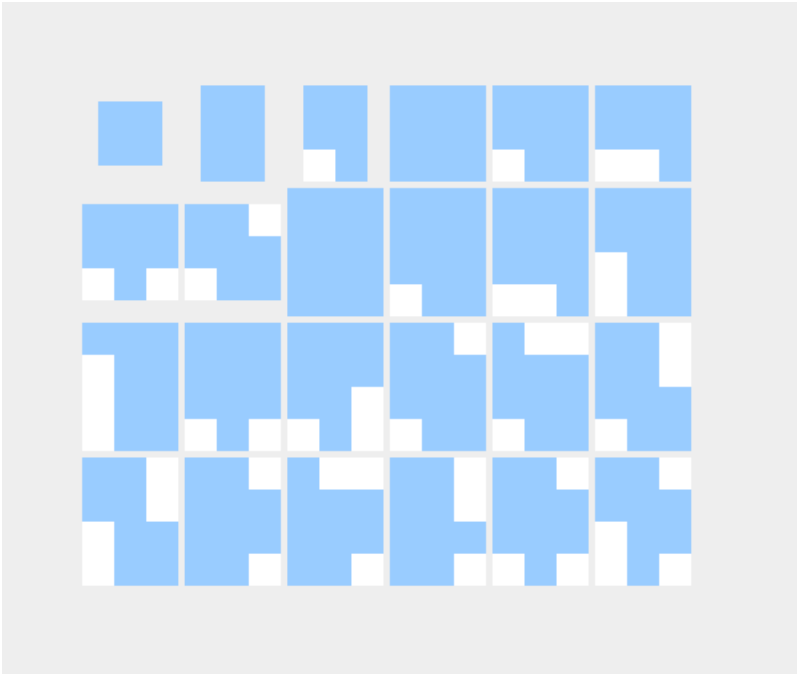
Algoritem za generiranje oblik sob uporablja naslednje parametre za nadzor procesa generiranja:

- Maksimalne in minimalne dimenzije prostora
- Maksimalno in minimalno število odrezanih pravokotnikov

- Maksimalne in minimalne dimenzije odrezanih pravokotnikov
- Dovoljene pozicije odrezanih pravokotnikov
- Evaluacijske funkcije



Slika 5.1: Prikaz rezanja pravokotnika dimenzij 3x4. Slika 1 prikazuje rezanje posameznega pravokotnika, Slika 2 pa rezanje treh. Črne celice označujejo kot v katerem režemo, temno modre pa prostor, ki ga izrezan pravokotnik lahko pokriva.



Slika 5.2: Del množice generiranih oblik sob.

5.3.2 Generiranje stavb

Stavbe so veliko večjih dimenzij kot sobe, saj jih le-te polnilo, zato pričakujemo, da bodo stavbe približno tri do petkrat večje od sob, kar bi pri istih omejitvah pripeljalo do le malo večje generirane množice oblik. Temu pa ni tako, saj obliko omejujejo drugačna pravila. V primerjavi s sobami je oblika stavbe lahko bolj kompleksna. Če si obliko predstavljamo kot smo si jo pri sobah, lahko zanjo postavimo nove omejitve. Medtem, ko smo sobam pustili najbolj enostavne oblike, sedaj tega nočemo. Poleg tega pa nas ne omejuje maksimalna mera kompleksnosti oblike zato lahko omejitev maksimalnega števila odrezanih pravokotnikov odstranimo. Popolnoma pravokotna oblika stavbe je dolgočasna, minimalno število odrezanih pravokotnikov zato moramo povečati. Rečemo lahko, da želimo bolj organski izgled kot pri sobah.

Vprašanje, ki sledi je, kako doseči željen rezultat. Problema bi se lahko lotili z enakim pristopom, kot pri generiranju sob vendar problem zaradi manjših omejitev pri generiranju postane obširnejši, sam rezultat algoritma pa so preenostavne, nezanimive oblike. Ker želimo bolj organski izgled lahko problem poskusimo rešiti z celičnimi avtomati, saj so nadvse primerni za generiranje bolj organskih oblik kot je npr. sistem jam. Glej sliko 5.3.

Slika 5.3: Rezultat celičnega avtomata na mreži 60x60 po sedmih iteracijah.

Algoritem celičnega avtomata izvedemo na prostoru stavbi primernih dimenzij, s čimer ustvarimo posamezno sobo organske oblike. Seveda pa nad

algoritmom želimo malce več nadzora. Prav tako kot pri celotni četrti lahko na podlagi istih lastnosti ocenjujemo obliko stavbe. (Čas potovanja, razdalje med sobami, ipd.) Rezultat, ki ga želimo je povezan prostor, do neke mere centraliziran okoli neke točke. (S tem iz množice rešitev odstranimo npr. podolgovate ozke stavbe) Kot nov faktor nadzora, dodamo algoritmu poleg nabora parametorv, ki jih potrebuje celični avtomat, filtre rasti. Te si lahko predstavljamo kot dvodimenzionalno mrežno dodatnih cen rojstva nove polne celice na določeni poziciji v prostoru. Da lahko zagotovimo, da bo rezultat algoritma korektne oblike dodamo še testiranje z evaluacijskimi funkcijami, ki v primeru izrojene oblike, le-to zavržejo.

Psevdo koda iteracije celičnega avtomata z filtri rasti:

1. Za vsako celico (na poziciji [i][j]):
2. Če je število polnih celic v soseski $>$ ($T = 4$) + vrednost filtra na poziciji [i][j].
3. Celica postane polna.
4. Sicer:
5. Celica postane prazna.

Algoritem za generiranje oblik stavb uporablja naslednje parametre za nadzor procesa generiranja:

- Maksimalna in minimalna dimenzija prostora
- Odstotek skalnih celic (nedostopno območje)
- Število iteracij celičnega avtomata
- Mejna vrednost soseske za rojstvo skale ($T = 5$)
- Število celic v soseski
- Filtri rasti
- Evaluacijske funkcije

3	3	2	2	3	3
3	2	2	2	2	3
2	2	1	1	2	2
2	1	1	1	1	2
2	1	0	0	1	2
2	1	0	0	1	2
2	1	0	0	1	2
2	1	0	0	1	2
2	1	1	1	1	2
2	2	1	1	2	2
3	2	2	2	2	3
3	3	2	2	3	3

Slika 5.4: Primer filtra rasti.

Slika 5.5: Rezultat našega algoritma. 81 zaporednih generiranj oblik stavb.

5.3.3 Zapolnitev stavb z sobami

Generiranje stavb je skoraj končano. Sedaj moramo generirane oblike stavb zapolniti z ustvarjeno množico sob. Problem, ki ga moramo rešiti je problem

pokrivanja ploščic. Rešimo ga lahko z preiskovalnim algoritmom, katerega množica potencialnih rešitev je skupina sob postavljenih v stavbo. Jedro algoritma je rekurzivna metoda, ki hrani informacijo o zapolnitvi prostora v določeni iteraciji. Metoda naključno izbere ploščico, v našem primeru obliko naključne sobe, jo postavi na določeno mesto v prostor stavbe ter kliče samo sebe (nova iteracija). Končni pogoj oziroma pogoj pri katerem se iteriranje ustavi, je skupina evaluacijskih funkcij. Te funkcije preverjajo, če je prostor stavbe zapolnjen in ali obstaja v prostoru stavbe zaključen prostor, premajhen za obstoječe ploščice, kar zagotavlja konec iteranja. Za večji nadzor nad procesom generiranja dodamo ploščicam, s katerimi zapolnjujemo, informacijo o številu uporab, s čimer lahko odstranimo rezultate, kjer je stavba zapolnjena s premalo unikatnimi oblikami sob. Za večjo verjetnost uspešne zapolnitve lahko evaluacijski funkciji, ki preverja ali je soba zapolnjena, dodamo parameter, ki dopušča določeno število nezapolnjenih celic. Te celice lahko algoritmi, ki bodo enkrat polnili sobo s predmeti, prepoznajo kot npr. shrambo ali pa npr. možnost za dodajanje skrivnih prostorov, kar doda novo dimenzijo v igralčevi izkušnji. Zaradi obsežnega prostora potencialnih rešitev preiskovalnega algoritma in ker ne moremo zagotoviti, da za dano stavbo obstaja rešitev, moramo biti pozorni na čas teka algoritma. Preiskovanje prostora rešitev za posamezno stavbo lahko časovno omejimo. V primeru, da rešitve za dan prostor ne najdemo ali se dan čas izteče, mora algoritem zaprositi za novo obliko stavbe, ter poskusiti ponovno. Za algoritem pri naših parametrih se izkaže, da je število problemov (stavb z naborom sob) pri katerih rešitev ne obstaja, izredno majhna zato algoritem zelo hitro generira zapolnjeno stavbo. Kratek tek algoritma na poti do rešitve je sicer priročna lastnost, za nas pa ni najpomembnejša, saj ne generiramo v času teka igre, ki bo rezultat prikazovala.

Psevdo koda rekurzivne metode: (Metoda drži informacijo o zapolnitvi prostora v določeni iteraciji)

1. Evaluacijske funkcije preverijo ali je končni pogoj izpolnjen

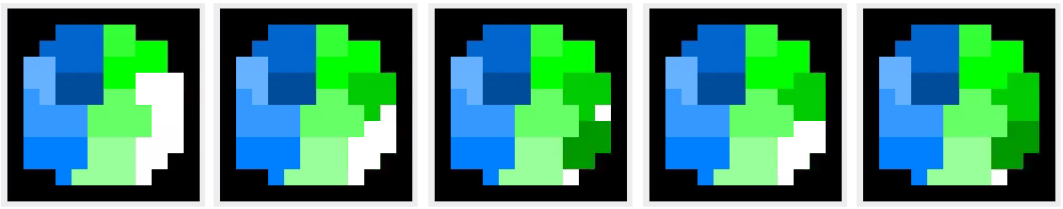
2. Za vsako izmed ploščic (oblike sob), katere število uporab je manjše od

omejitve:

3. Za vsako izmed rotacij ploščice:

4. V prostor na določeno mesto položimo ploščico in povečamo njen števec uporab

5. Rekurzivni klic metode z novo zapolnjenim prostorom

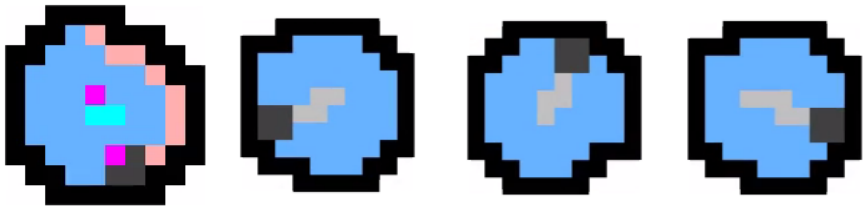


Slika 5.6: Reševanje problema polaganja ploščic. Slika prikazuje primer poteka preiskovanja.

Slika 5.7: Primer nenaravno porazdeljenih sob znotraj večje stavbe.

Hitro opazimo, da pri stavbah večjih dimenzij dobimo rezultat v katerem je razporeditev sob zaradi pomankanja hodnika precej nenaravna, velikokrat pa se v sredini stavbe pojavi soba, ki meji na večje število sosednjih sob kot ostale sobe. Veliko bolj korekten rezultat bi dobili, če bi dana soba ustrezala pričakovani obliki hodnika. Reševanje problema s pokrivanjem ploščic sob, katerim bi sedaj dodali vse možne oblike hodnika, je precej "slep" pristop,

zato bi bilo primerno poiskati alternativo. V tem delu bomo problem rešili z algoritmom, ki prepozna robne celice stavbe, tam ustvari predsobo, nato oceni lokacije celic, kjer bi stala osrednja, prej omenjena "hodnik" soba, in nato preiskuje po prostoru možnih povezav v obliki hodnika, med tema skupinama celic. Glej sliko 5.8. Algoritem nato v stavbi z dodanim hodnikom reši problem pokrivanja ploščic in ga tako napolni z sobami.



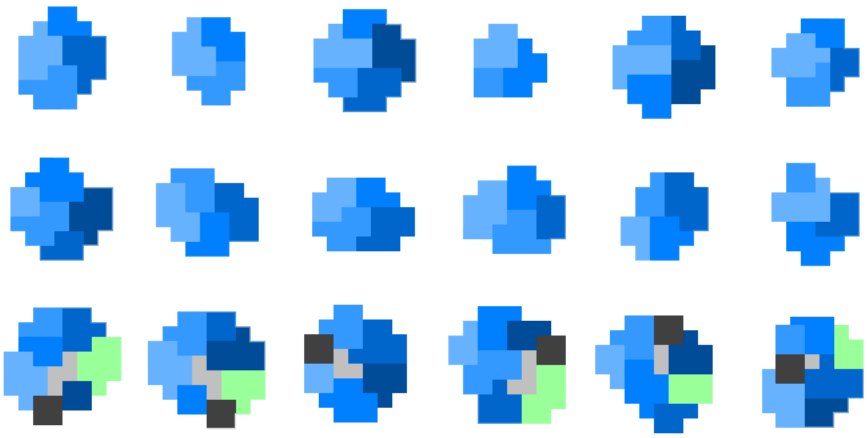
Slika 5.8: Slika 1 prikazuje osnovo za generiranje hodnikov in predsob. Roza obarvane celice označujejo robne celice, kjer lahko postavimo predsobo. Sivo obarvane označujejo predsobo, svetlo modre označujejo celice sredinske "hodnik" sobe, vijolične pa dve izbrani celici, ki jih bo generiran hodnik povezoval. Slike 2, 3 in 4 prikazujejo uspešno generiran hodnik.

Algoritem potrebuje:

- Množico ploščic (množica oblik sob)
- Prostor, ki ga zapolnjuje (oblika stavbe)

Algoritem za polnjenje stavb s sobami uporablja naslednje elemente za nadzor procesa:

- Minimalna dimenzija stavbe, ki ji dodamo hodnik
- Evaluacijske funkcije za preverjanje končnega pogoja
- Maksimalno število uporab posamezne sobe
- Maksimalno število nezapolnjenih celic



Slika 5.9: Razultat algoritma za pokrivanje ploščic. S sobami napolnjena stavba.

5.3.4 Gručenje stavb v gosto formacijo

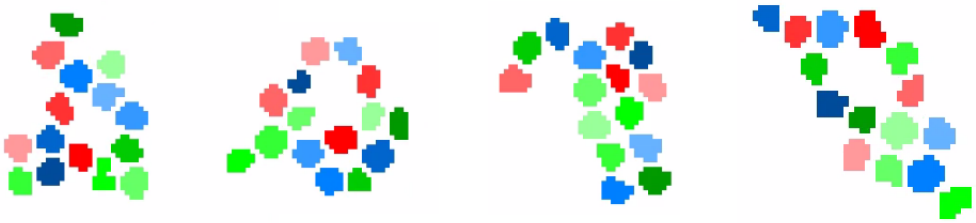
Do sedaj smo ustvarili množico, z sobami napolnjenih sob, ki jih moramo sedaj urediti v formacijo mestne četrti. Lahko bi najprej generirali obliko mestne četrti in nato reševali problem polaganja ploščic (tokrat oblik generiranih stavb) vendar takšen pristop zaradi veliko večjega prostora potencialnih rešitev ni optimalen. Ob analizi sestavnih delov meste četrti smo ugotovili, da je bolje stavbe najprej gručiti z dodajanjem ulic in kasneje definirati zid, ki omejuje prostor mestne četrti. Že pri analizi hodnikov v stavbah smo ugotovili, da bi bilo generiranje ulic vnaprej precej zamudno, za razliko od hodnikov pa so ulice še veliko manj omejene z pričakovano predstavo (saj so razširitve ulic, oz. prostora med stavbami nekaj veliko bolj naravnega) zato bomo problem ulic rešili drugače. Pri gručenju stavb lahko upoštevamo pravilo, da se stavbi ne smeta stikati, in tako ustvarimo prostor med njima. Tako ustvarjen prostor tretiramo kot prostor ulic. Nastali problem bi lahko rešili z reševanjem problema gručenja krogov (“Circle Packing Porblem”), kjer množico krogov gručimo v čimbolj gosto formacijo. Dotični problem je močno raziskovana tema geometrije in zanj obstaja vrsta algoritmov, ki ga

rešujejo. Osnovna oblika algoritma je prikazana v psevdo kodi spodaj.

Psevdo koda algoritma za gručenje krogov:

1. Začnemo z naključno porazdeljenimi krogi v prostoru.
2. Optimiziramo te pozicije tako, da se krogi med seboj ne prekrivajo, z večanjem razdalje med krogi, ki se prekrivajo in manjšanjem razdalje med ostalimi.
3. Izvedemo izboljševalno akcijo (npr. zamenjamo pozicije dveh krogov pri katerih pričakujemo, da bo prišlo do izboljšave).
4. Vrnemo se v korak 2. Iteracija se zaključi ko dosežemo maksimalno dovoljeno število iteracij oz. algoritem ne najde več izboljševalne akcije.

Problem je iz matematičnega vidika zelo obširen in iskanje izboljševalne akcije je težavna naloga. Poskusimo izvesti dani algoritem, brez iskanja izboljševalne akcije. Glej sliko 5.10.



Slika 5.10: Rezultat algoritma za gručenje krogov brez izboljševalnih akcij.

Hitro vidimo, da je takšna izvedba algoritma pomankljiva. Brez kompleksne izboljševalne akcije algoritem povzroči prevelike anomalije v strukturi četrti. Lahko bi poskusili poiskati primerno izboljševalno akcijo, vendar seveda obstajajo alternativne poti. Opozoriti je potrebno, da v našem primeru

delamo s šahovnico celic in je tako zaloga vrednosti za lokacije stavb diskretna in ne zvezna. Pri konstruiranju algoritma je primerno, da to prednost vzamemo v poštev. Poleg tega je velika pomankljivost pri reševanju problema gručenja krogov dejstvo, da z uporabo očrtanega kroga izgubimo informacijo o obliki črte (zidu), ki stavbo omejuje. Problem bomo v tem delu rešili z rastočim algoritmom, pri katerem začnemo z praznim prostorom v katerega sredino, postavimo naključno stavbo. Algoritem nadaljuje tako, da v prostor doda novo naključno stavbo. Postavi jo nad že postavljeno formacijo stavb, na vse pozicije, v katerih bi pri premiku navzdol le-ta trčila z ostalimi, in jo nato pomika navzdol v zadnjo pozicijo, kjer se stavbe še ne stikajo, kar tretiramo kot novo formacijo. Tako formiramo ulico med stavbama. Postopek se ponovi za vse štiri smeri (gor, dol, levo in desno) in nato množico rešitev ocenijo evaluacijske funkcije (velikost očrtanega pravokotnika, velikost očrtanega kroga, seštevek razdalj med celicami vseh sob, ipd.). Po izbiri najboljše rešitve se postopek ponavlja dokler v prostor ne dodamo vseh stavb. Tako formiramo gosto formacijo stavb med katerimi se prepletajo ulice. Postopek bi lahko izboljšali z nenaključnim, intelegentnim izbiranjem zaporedja dodanih sob. Psevdo koda algoritma je spodaj.

Psevdo koda algoritma diskretne rasti:

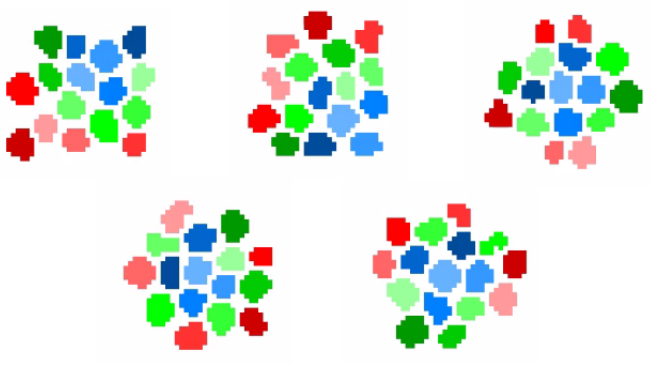
1. V center prostora postavimo prvo stavbo

2. V prostor dodamo novo stavbo na vse pozicije okoli obstoječe formacije tako, da je nova stavba za 1 celico odmaknjena od formacije

3. Z evaluacijskimi funkcijami ocenimo vse nastale pozicije ter izberemo najugodnejšo

4. Nastalo formacijo pomaknemo na sredino prostora

5. Če nismo dodali vseh stavb se vrnemo v korak 2



Slika 5.11: Rezukltat algoritma diskretne rasti.

5.3.5 Definiranje zidu

Po vseh problemih, ki smo jih rešili do sedaj, bi moralo biti definiranje zidu, ki omejuje četrt, preprosta naloga. V prostoru imamo postavljene stavbe okoli katerih moramo "zarisati" zid. Problema se lotimo z zavedanjem, da so stavbe zgručane z razmikom ene celice. Najenostavnejša rešitev bi bil najmanjši očrtan krog ali pravokotnik, vendar je ta za naše namene neprimerna. Malce kompleksnejša rešitev bi dosegli z žarki, s katerimi bi "svetili" po vseh stolpcih mreže prostora od zgoraj in spodaj, ter po vseh vrsticah iz leve in desne. Žarek se ustavi, ko trči ob stavbo. Celice, ki jih žarek obsvetli bi tretirali kot nedostopen prostor. Vse celice nedostopnega prostora, ki mejijo na celice stavb pa bi tako formirale zid. Problem tega pristopa je, da lahko žarek prodre globoko v formacijo stavb in povzroči anomalijo (škodi povezljivosti, razdvoji četrt, ipd.). Lahko bi poiskali nabor pravil za žarke, ki bi se takšnim anomalijam izognil, vendar obstajajo bolj optimalni pristopi. V analizi celičnih avtomatov in pri njihovi uporabi pri generiranju sob smo ugotovili, da se pri pravih pogojih forma polnih celic širi. Glej sliko 5.12.

Slika 5.12: Primer rasti forme skozi iteracije celičnega avtomata. Stopnje iteriranja so prikazane z leve proti desni. Prikazan primer je pod vplivom filtra rasti.

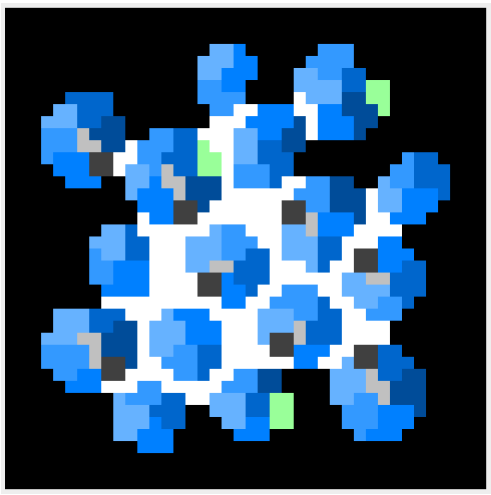
To lastnost lahko tukaj uporabimo v naš prid in tako zelo efektivno rešimo problem. Lotimo se tako, da stavb ustvarimo okvir polnih celic okoli množice stavb, ter pustimo, da celični avtomat iterira dokler polne celice ne objamejo formacije stavb. Nabor pravil celičnega avtomata nastavimo tako, da se formacija širi v vse smeri razen v tunele širine dva ali manj. Glej sliko 5.13.

Slika 5.13: Prikaz maksimalne rasti forme z iteriranjem celičnega avtomata, v odprtine širine 1 in 2. Modro obarvane celice predstavljajo stavbe, črno obarvane nedosegljiv prostor (to so celice forme, ki raste), sivo obarvane pa ulice.

Psevdo koda algoritma:

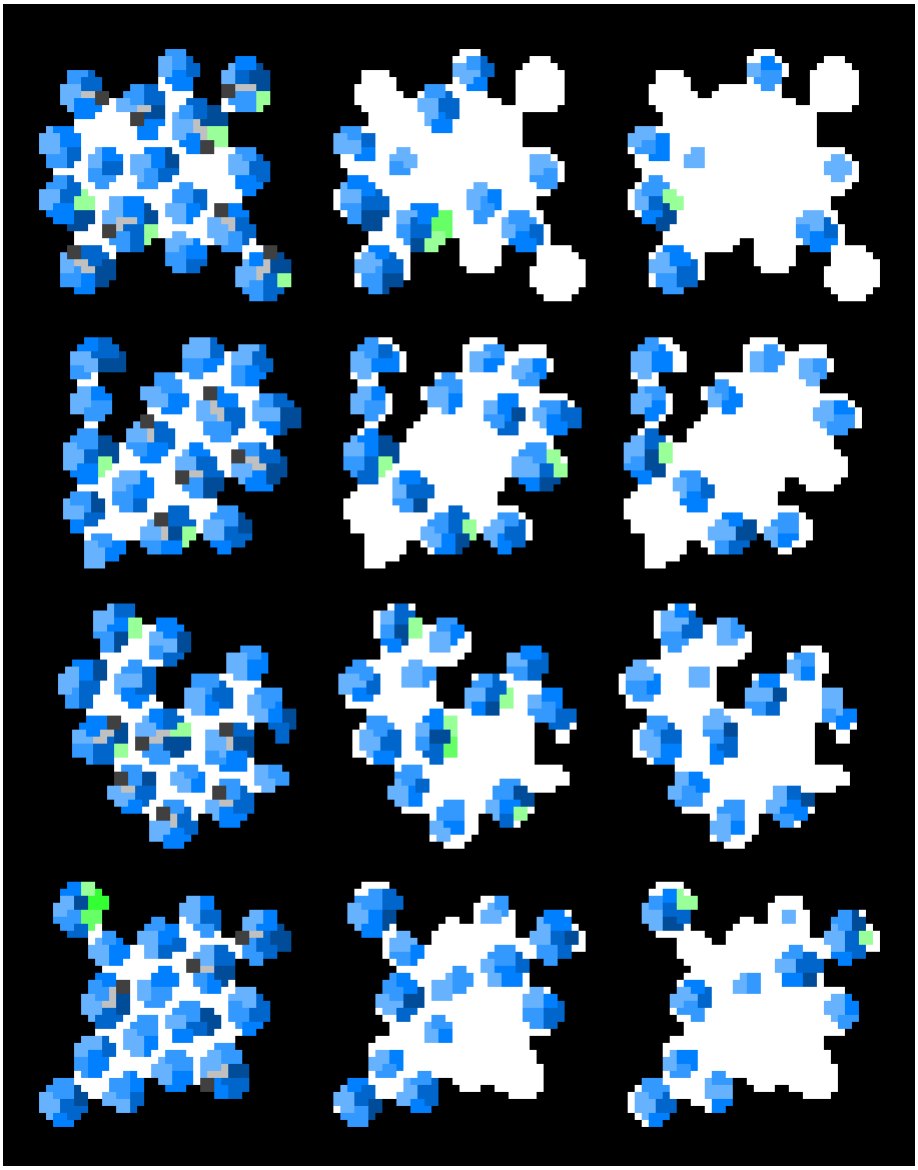
1. Formaciji stavb očrtamo pravokotnik.
2. Pravokotnik povečamo za 3 v vse smeri.
3. Prostor izven pravokotnika napolnimo z polnimi celicami (nedostopen prostor).

4. Celični avtomat iterira dokler nova iteracija še dela spremembe.

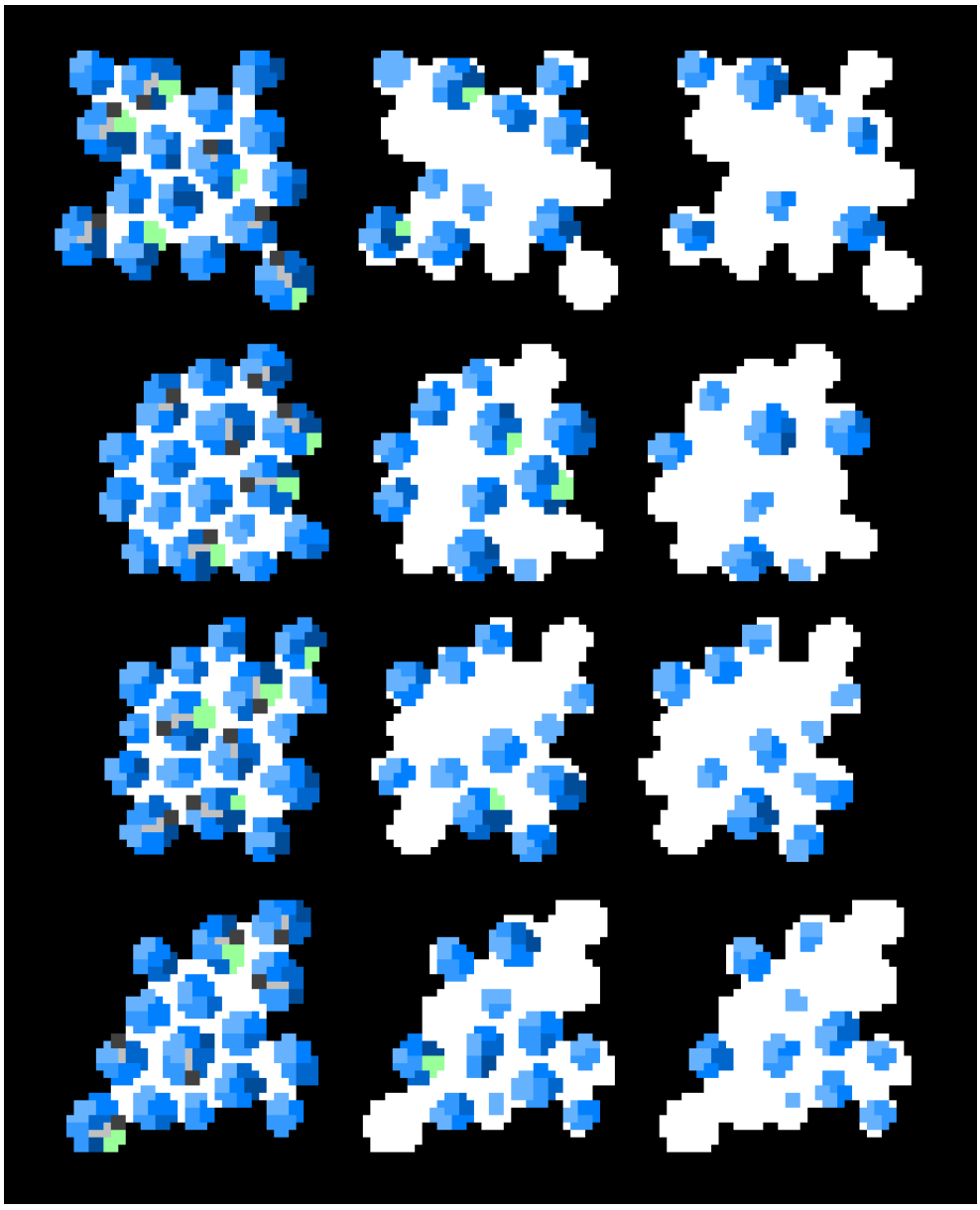


Slika 5.14: Uspešno generirano območje nedosegljivih celic, ki omejujejo prostor premikanja.

Dobljena rešitev je rezultat celotnega algoritma generiranja mestne četrti. Dosegli smo visoko stopnjo pozidane površine, visoko stopnjo povezanosti ter organski izgled. Množica možnih rešitev je ogromna, saj rezultati med sabo močno varirajo, definirali pa smo številne parametre s katerimi lahko proces generiranja nadziramo in spreminjamo. Kljub temu ustvarjena četrt ne daje občutka, da je bila zgrajena s pomočjo proceduralnega algoritma, njen videz pa je zadovoljiv.



Slika 5.15: Končen rezultat celotnega algoritma za generiranje arhitekture mestnih četrti. Vsaka vrstica prikazuje, pritličje (levo), prvo (sredina) in drugo (desno) nadstropje četrti.



Slika 5.16: Končen rezultat celotnega algoritma za generiranje arhitekture mestnih četrti. Vsaka vrstica prikazuje pritličje (levo), prvo (sredina) in drugo (desno) nadstropje četrti.

Poglavje 6

Zaključek

Pokazali smo, da je proceduralno generiranje vsebin izvedljiv pristop. Če analiziramo sestavne dele forme in razbijemo problem v manjše kose, ga lahko primerno priredimo našim zmožnostim in razpoložljivim sredstvom. Vsekakor lahko na ta način ob sodelovanju z umetnikom dosežemo veliko več, kot bi lahko v tem času ustvaril umetnik sam.

Literatura

[1] Illiger A. Tiny wings, 2011.

[2] Yu D., Hull A. Spelunky, 2009.

[3] Togelius J., Kastbjerg E., Schedl D. and Yannakakis. What is procedural content generation?: Mario on the borderline.

[4] Ingram D.R. The concept of accessibility: A search for an operational form. *Regional Studies*, 5(2):101–107, 1971.

[5] Ingram D.R. Public transit and job access in chicago. *Transportation Engineering Journal, American Society of Civil Engineers*, 98:351–366, 1972.

[6] Firaxis Games. Civilization iv, 2005.

[7] Veivendi Universal Games. The hobbit: The prelude to the lord of the rings, 2003.

[8] Braben D., Bell I. Elite, 1984.

[9] Doran J., Parberry I. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):111 —- 119, 2010.

[10] Lawrence J., Yannakakis G.N., Togelius J. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 10:1–10:4, New York, NY, USA, 2010. ACM.

[11] Liapis A., Yannakakis G.N., Togelius J. Sentient sketchbook: Computer-aided game level authoring. *Proceedings of ACM Conference on Foundations of Digital Games*, 2013.

[12] Cervero R., Kockelman K. Travel demand and the 3ds: Density, diversity and design. *Transportation Research Part D: Transport and Environment*, 2(3):199 – 219, 1997.

[13] Maxis. Spore, 2008.

[14] Mojang. Minecraft, 2011.

[15] Blizzard North. Diablo, 1997.

[16] Pcg wiki: Procedural content generation wiki. `URLhttp://pcg.wikidot.com/`.

[17] Linden R., Lopes R., Bidarra R.. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78 – 89, mar 2014. doi: 10.1109/TCIAIG.2013.2290371.

[18] Lopes R., Tutenel T., Bidarra R. Using gameplay semantics to procedurally generate player-matching game worlds. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, pages 3:1–3:8, New York, NY, USA, 2012. ACM.

[19] Smelik R.M., Tutenel T., Kraker K.J., Bidarra R. A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics*, 35:352–363, 2011.

[20] Was J., Sirakoulis G.C., Bandini S. *Cellular Automata: 11th International Conference on Cellular Automata for Research and Industry, ACRI 2014, Krakow, Poland, September 22-25, 2014, Proceedings*. Lecture Notes in Computer Science / Theoretical Computer Science and General Issues. Springer International Publishing, 2014.

[21] L. Wittgenstein. *Philosophical Investigations*. Blackwell, 1953.