LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

PROCESUĀLI ĢENERĒTAS SPĒĻU PASAULES

MAĢISTRA DARBS

Autors: **Arnolds Ostrovskis**
Stud. apl. Nr. ao10061,
Darba vadītājs: profesors Dr. dat. Guntis Arnicāns

RĪGA 2018

ANOTĀCIJA

Darbā aprakstītas datorspēles un to izstrādes procesu problēmas, kas saistītas ar spēles satura izveidošanu, kā arī piedāvāts šīs problēmas risinājums – automātiski ģenerēts saturs. Šim risinājumam tika apskatītas vairākas metodes un citu autoru spēles, kurās šīs metodes tiek izmantotas. Vairākām metodēm tika piedāvātas pielāgošanas un uzlabošanas iespējas, kā arī aprakstīti šo metožu iespējamie ierobežojumi. Tika ieviesta terminoloģija un pamatprincipi, kas palīdz vispārēji saprast, kā meklēt un pielāgot automātiskās ģenerēšanas rīkus konkrētām spēlēm. Darbā aprakstītas dažas spēles, kurām nepastāv vispārēji aprakstīta automatizēta satura ģenerēšana, un kurām šī ģenerēšana tika ieviesta.

Spēles, automatizācija, satura veidošana.

ABSTRACT

Procedurally generated game worlds

The work describes computer games, problems of the development process for such games and solution for those problems in the form of automatic content generation. Various methods and games that use those methods have been researched. Adaptation and improvement possibilities as well as possible limitations were provided for many of those methods. Terminology and basic methods were proposed, that should help understand how to search and adapt automatic generation tools for certain games. After that, work describes few games, for which automatic content generation isn't generally described. For those games such generation was provided in this work.

Games, customization, content generation.

AUTOREFERĀTS

Darba laikā apskatītajā literatūra vai nu koncentrējas uz konkrētas spēles tipa ģenerēšanas aprakstu, vai arī uz vispārēju teoriju, kuru ir ļoti grūti pielietot spēlē, ņemot vērā spēles kvalitātes radītājus. Izmantojot šo literatūru, ir sarežģīti pilnvērtīgi pielietot nepieciešamās metodes jaunajās (lasītāja izdomātajās) spēlēs.

Šajā darbā tiek apskatītas gan teorētiskās zināšanās par procesuālas ģenerēšanas metodēm, gan arī konkrēti piemēri no citu autoru spēlēm. Šo avotu apvienošanas rezultātā, tika vispārinātas metodes, kuras tiek izmantotas esošajās spēlēs un piedāvāta vairāku teorētisko rīku un metožu pielāgošana konkrētiem spēļu piemēriem, kuriem pirms tam šīs metodes, vai nu nebija, aprakstītas, vai arī vispār netika izmantotas. Darba rezultātā arī tika parādīts, ka satura ģenerēšanai var izmantot vairākus matemātiskus rīkus, kuri, pirmatnēji, tika izmantoti citās nozarēs. Apvienojot šos matemātiskos rīkus ar darbā aprakstītajam ģenerēšanas pieejam, ir iespējams atrast procesuālas ģenerēšanas risinājumu gandrīz jebkurai spēlei, kas arī bija darba mērķis.

Papildus tam tika aprakstīta spēles kvalitātes teorija un noteikts, ka ģenerēšanas algoritms var ņemt vērā spēles kvalitātes īpašības, kam pēc autora zināšanām, netika pievērsta liela uzmanība citos darbos.

SATURS

APZĪMĒJUMU SARAKSTS

PĢS - Procesuāli ģenerēts saturs.

NSĢ - Nejaušo skaitļu ģenerators

Backtracking – Meklēšanas metode, kur meklēšana notiek dziļumā, izejot visas iespējamās pozīcijas, un, nonākot līdz strupceļam, meklēšana turpinās no jau apmeklētas pozīcijas, kurai ir iespēja izvēlēties alternatīvo ceļu.

IEVADS

Tradicionālajā nozīmē spēle ir strukturēta aktivitāte, ko cilvēki veic izklaides, vai sacensību pēc. Vismodernākās un iespējams visizplatītākās ir spēles, kuras tiek izveidotas un spēlētas virtuālajā vidē. Tādas spēles ir diezgan cieši saistītas ar datorzinātnēm, kas paver iespējas veikt pētījumus, kuri vērsti uz spēļu izstrādes procesu uzlabojumiem. Šobrīd viena no nozīmīgākajam problēmām šajos procesos ir spēļu resursu izstrāde. Tādi resursi varētu būt attēli, algoritmi vai piemēram trīsdimensionāli virtuālie objekti.

Pateicoties arvien jaudīgākiem datoriem, viena spēle var saturēt milzīgu spēles resursu daudzumu, kuru izveidošanai ir nepieciešams ļoti nozīmīgs cilvēkresursu skaits. Tradicionāli resursu izstrādē piedalās mākslinieki, programmētāji un citi izstrādātāji, bet pastāv arī iespēja ģenerēt šos spēļu resursus izmantojot automatizētus procesus. Diemžēl šajos automatizētos procesos ir jāiegulda nopietns pētniecisks darbs, pirms tādu procesu rezultāti paliek vērtīgāki par cilvēku veidotajiem. Tāpēc šī darba uzdevums, ir izpētīt un aprakstīt šos procesus, kam vajadzētu atvieglot to izmantošanu spēļu izstrādē.

Šādu procesu ģenerēšanas sistēmas mēdz saukt par procesuāli ģenerēta satura (turpmāk darbā PĢS) sistēmām. Šī darba laikā tiks izpētītas citu autoru vienkāršas PĢS sistēmas un paņēmieni, ka arī piedāvāti to pielāgošanas veidi konkrētam situācijām.

Par spēļu pasauli šajā darbā tiek dēvēti spēles elementi, kuri ietekmē spēles gaitu, nosaka spēles sarežģītību un vizuālo izskatu, vai citu sensoro informāciju. Tādas lietas, mākslīgais intelekts, netiks iekļauts šajā darbā, kā spēļu pasaules procesuāli ģenerēta sastāvdaļa, bet var tikt iekļauts kā palīglīdzeklis šīs pasaules izveidošanā. Spēles līmenis un spēles laukums var tikt izmantots, ka spēles pasaules sinonīms.

Darba pirmajā nodaļā lasītājs tiek iepazīstināts ar svarīgākajiem pamatjēdzieniem, kuri tiks izmantoti darba gaitā. Šo jēdzienu definējums aizgūts no citiem darbiem, bet to nozīme var tikt paplašināta vai pielāgota darba mērķu sasniegšanai.

Otrajā nodaļā tiks apskatīti citu autoru darbi PĢS jomā. Šie darbi ietvers sevī spēles, kuras izmanto PĢS un sistēmas, kuras pašas ģenerē spēles. Šo darbu apskate ļaus saprast PĢS iespējas un atrast biežāk izmantotos algoritmus.

Trešajā nodaļā būs aprakstīti daži paņēmieni un matemātiski rīki, kuru izmantošana nepieciešama PĢS sistēmu izstrādē. Pārsvarā šie rīki ļaus vispārēji saprast, kādā veidā tiek ģenerēts PĢS un palīdzes pielietot ģenerācijas procesus konkrētiem piemēriem.

Ceturtajā nodaļā tiks aprakstītas dažas autora spēles, kuras izmanto PĢS. Uzsvars tiks likts uz ģenerēšanas algoritmu, tā kvalitāti un izmantošanas iespējas citās spēlēs. Pēc autora zināšanām, nodaļā piedāvātie satura ģenerēšanas risinājumi vai nu netiek izmantoti, līdzīgās spēlēs, vai arī nav vispārīgi aprakstīti citos darbos. Šis nodaļas rezultātu iegūšanai var tikt daļēji izmantotas no otras vai ceturtās nodaļas aizgūtas metodes, bet iegūtas spēles un to PĢS sistēmas ir autora panāktie rezultāti.

1. PAMATJĒDZIENI

Darbā plaši tiks izmantoti vairāki ar PĢS saistīti jēdzieni, kuru nozīmē varētu nebūt skaidra. Šajā nodaļā lasītājs tiek iepazīstināts ar jēdzieniem un to nozīmi darba ietvaros.

1.1. Procesuāli ģenerēts saturs

Spēles saturs apzīmē jebkurus resursus, ieskaitot grafiskos un matemātiskos modeļus, spēles kodu, skaņas, stāstījumu, noteikumus un citus abstraktus un digitālus vienumus, kuri ietekmē spēles gaitu. Procesuāli ģenerēts saturs ir tāds saturs, kuru dators izveidoja patstāvīgi, vai ar cilvēka palīdzību. Visvienkāršākajā formā tas ir pilnīgi nejauši ģenerēts saturs, taču daudz lielāka uzmanība tiks vērsta algoritmiem, kuri ģenerē saturu, izmantojot noteiktu likumsakarību kopu un/vai sarežģītākus matemātiskus modeļus.

Svarīgi ir uzsvērt, ka PĢS var tikt veidots cilvēkam noteiktajā pakāpē, sadarbojoties ar datoru un algoritmiem. Tas ietver arī gadījumus, kad procesuāla ģenerācija drīzāk palīdz cilvēkam pašam veidot saturu. Skaidrības pēc tiek piedāvāti daži PĢS sistēmas piemēri atkarība no cilvēka iesaistīšanas pakāpes:

a) Algoritms pilnīgi neatkarīgi no cilvēka izveido līmeni, uzzīmējot un saliekot katru elementu spēles pasaulē.

b) Cilvēks definē dažus parametrus, ka piemēram cik koku un cik ienaidnieku viņš grib spēles pasaulē un dators izveido PĢC, ņemot vērā šos parametrus.

c) Cilvēks izveido atsevišķus spēles elementus un ievada datus par to, kādā veidā šos elementus var kombinēt. Dators savukārt izveido vairākas šo elementu kombinācijas.

d) Cilvēks saliek visus pasaules elementus, un dators aprēķina un izlabo vietas, kur cilvēks pieļāvis kļūdas balstoties uz spēles pasaules noteikumiem.

1.2. Spēles līmenis

Spēles līmenis ir spēlējama satura kopuma vienība, kurai parasti ir definēts sākums un beigas. Spēle var sastāvēt no viena vai vairākiem līmeņiem, kurus biežāk saista tikai spēles stāsts. Pārsvarā procesuāli ģenerējot saturu, tiek ģenerēts viens līmenis, kuram veiksmīgas ģenerēšanas rezultātā piemīt spēlei nepieciešamās īpašības.

Spēles līmenis var piemēram būt izolēta sala, lielajā spēles pasaule, vai arī viena krustvārdu mīkla spēlē, kur ir jārisina vairākas krustvārdu mīklas. Viens līmeņa piemērs dots *1.1.att (b)*, kuram pastāv starta pozīcija un tā iziešanas nosacījums.

1.1. att. **Spēles genotips(a) un fenotips (b)**

1.3. Genotips un Fenotips

Ģenerējot spēles pasauli, tajā jānodrošina noteikta īpašību izpilde. Piemēram ģenerējot labirintu, šim labirintam ir jāpastāv labirinta sākumam, labirinta beigām un arī ceļam starp šiem diviem punktiem. Diemžēl tādā veidā definētas īpašības, ir, diezgan sarežģītas no algoritmu viedokļa. Tieši tāpēc tie ir jāvienkāršo, izmantojot abstraktus matemātiskus modeļus.

Tāds matemātisks modelis varētu būt gan parametru vektors, gan arī piemēram neorientēts grafs, kur katra virsotne raksturo daļu no mūsu labirinta. No algoritmu puses, grafs

ir daudz vienkāršāks par labirintu, un tam jau ir vairāki izpētīti veidi, ka algoritmiski un matemātiski noteikt tā īpašības (ieskaitot ceļa atrašanu starp sākuma un beigu virsotni).

Tātad darbā pārsvarā tiks ģenerētas matemātiskas struktūras, kuru īpašības pārnesīsies uz jau gataviem līmeņiem. Tas savukārt nozīmē, ka ir jādefinē šīs matemātiskas struktūras un līmeņa attiecības. Tagelius u.c. savā darbā [13] izmanto Mākslīgajā intelektā bieži izmantotos terminus – Genotipu un Fenotipu.

Fenotips ir ģenētikā bieži izmantojams termins, kas, tulkojot no grieķu valodas, nozīmē parādīt (phainein) tipu (typos) un pēc būtības ar to apzīmē organisma fizikālās īpašības, ko cilvēks var saskatīt ar aci vai citiem māņu orgāniem. Genotips no otras puses ir informācijas kopums par šo organismu, kas parasti tiek glabāts iekš organisma DNS.

Tātad par fenotipu sauksim gatavu līmeni, un genotips apzīmēs matemātisku modeli. Starp abiem pastāvēs noteikta līdzības pakāpe, kura apzīmēs, cik lielu īpašību skaitu genotips pārnes fenotipā, par ko vēlāk tiks runāts darbā.

Genotipa un fenotipa piemērs tiek dots *1.1.att*. Šeit genotips ir grafs, kurš parāda spēles līmeņa iziešanas variantus, kamēr fenotips ir parādīts kā spēles ekrānuzņēmums, kas atbilst īstam spēlējamam līmenim.

1.5. Spēļu un līmeņu kvalitāte

Darbā svarīgu lomu spēlē tas, cik ļoti spēlei būtu jāpatīk spēlētajam, jeb cik tā ir interesanta. Šis ir diezgan subjektīvs radītājs, taču vairāki autori mēģina definēt īpašības, kas piemīt interesantām spēlēm. Jesper Juul Savā darbā [1] pieminēja 6 tādas īpašības:

1. "Definēti noteikumi". Kas nozīmē, ka spēlētājs saprot, kas jādara spēlē un šie noteikumi nemainās. PĢS sistēmu ietvaros ievērosim, ka ģenerētas pasaules seko jau esošajiem noteikumiem, ja vien šo noteikumu maiņa nav ģeneratora mērķis.

2. "Dažādi un izskaitļojami iznākumi". Pirmā daļa parastajās spēlēs varētu nozīmēt, ka spēlētājs var zaudēt, uzvarēt, vai piemēram gūt kādu punktu skaitu spēlēs beigās. PĢS šo īpašību var izpildīt, piemēram, ģenerējot līmeņus pakāpeniski, atkarībā no spēlētāja darbībām. Izskaitļošanas iznākums bieži vien ir saistīts ar sarežģītākiem matemātiskiem modeļiem, kurus var izskaitļot tikai daļēji, tādā veidā pārbaudot spēlētāja spējas un saglabājot

pārsteiguma elementu. PĢS ir svarīgi izveidot līmeņu variācijas, kur katrs līmenis ir kā mainīgais šajā matemātiskajā modelī.

3. Labi un slikti iznākumu (darbā "Valorization of the outcome"). Šī īpašība nozīmē, ka dažas spēlētāja darbības nes labākus iznākumus, un no tam dažas – sliktākus.) PĢS šai īpašībai var uzlikt papildus sarežģītības pakāpi, jo vairākas reizes izejot spēli, tas pasaules mainās, un atkārtojot vienu un to pašu darbību citā pasaulē, var rasties citi iznākumi.

4. "Spēlētāja darbietilpība", jeb sarežģītības līmenis. Tas nozīmē, kā spēlē ir svarīga spēles grūtības pakāpe, jeb precīzāk sakot: ir jāatrod spēles sarežģītība, kura nav garlaicīga, bet tajā pašā laikā nav arī pārāk grūta. PĢS šeit ievieš iespēju ģenerēt saturu ar noteiktu sarežģītības pakāpi, kas arī tiks aprakstīts vēlāk darbā.

5. "Spēlētāja piesaiste pie iznākuma". Īsumā "spēlētajam jājūtas labi, kad viņš uzvar un slikti, kad viņš zaudē". PĢS sistēmās ir vairākas iespējas panākt šo efektu, bet, iespējams, spēcīgākā no tām ir stāsta ģenerēšana, kura ļauj ģenerēt unikālu stāstu katram spēlētajam un padarīt to nedaudz personiskāku. Par stāsta ģenerēšanu tiks runāts vēlāk darbā.

6. "Apspriežama ietekme" apsver spēļu iespēju ietekmēt reālo pasauli, ka, piemēram, traumas sporta spēlēs. Šī īpašība maz ietekmē spēles, par kuram tiks runāts šeit.

2. CITU AUTORU SPĒLES AR PĢS

2.1. Ievads

Šajā nodaļā tiks apskatītas spēles, kurās jau tiek izmantotas PĢS. Nodaļas mērķi ir

a) Iepazīties ar dažam tehnoloģijām, kuras tiek izmantotas citās spēlēs

b) Piedāvāt reālas pasaules piemērus, uz kuriem varēsim atsaukties citās nodaļās

c) Apskatīt iemeslus šīm spēlēm izmanot PĢS un izvilt no tā noderīgus secinājumus.

Spēļu PĢS sistēmas šajā nodaļā būs aprakstītas diezgan augstā līmenī. Gadījumos kad tas ir svarīgi, detalizētāks sistēmas apraksts tiks piedāvāts citā, šai sistēmai veltītajā, nodaļā. Saraksta secība ir nejauša, un tikai neliela daļa ir no visām PĢS spēlēm ir ietverta. Spēles ietvertas dažādu iemeslu pēc, kuru dēļ tas interesantas šim darbam.

2.2. Publicētas Spēles

2.2.1. Elite

Elite (1984) ir labs piemērs uzdevumam, kuru bija jārisina pirmajam PĢS sistēmām, kas jau tajā laikā nebija iespējama bez plašas spēļu pasaules. Balstoties uz [2], tā bija viena no pirmajam 3D spēlēm ar plašu pasauli, kas to laiku datoriem rādija nopietnas problēmas pateicoties ļoti ierobežotai atmiņai un relatīvi lielam datu apjomam, kas nepieciešams šīs pasaules glabāšanai. Elite to atrisināja, izmantojot deterministisku PĢS sistēmu. Tas nozīmēja, ka, izmantojot ļoti vienkāršu genotipu, jeb sēklu, spēle varēja procesuāli ģenerēt un dzēst katru pasaules daļu tajā brīdī, kad tas bija nepieciešams, un, atgriežoties uz šo vietu atkal, tā būs uzģenerēta tāpat ka iepriekšējo reizi.

Spēle nav pieejama atvērta pirmkoda variantā, bet ņemot vērā to, ka šī ģenerācijas algoritma mērķis ir dinamiski veidot saturu uz vecas iekārtam, varam to uzskatīt par pietiekami ātru mūsdienu datoriem.

2.2.2. ".kkrieger"

".kkrieger" ir modernāks piemērs tām, kā PĢS sistēmas var ieekonomēt lielu atmiņas daudzumu. Šī ir vienkārša pirmās personas šaušanās spēle, kura pazīstama ar to ka tas izmērs ir 96 kilobaiti. Balstoties uz autoru prezentāciju [3], tas pārsvarā tika panākts, izmantojot procesuāli ģenerētas tekstūras. Tradicionāli tekstūrās tiek glabāta informācija par pikseļiem. ".kkrieger" gadījumā procesa soļi ir

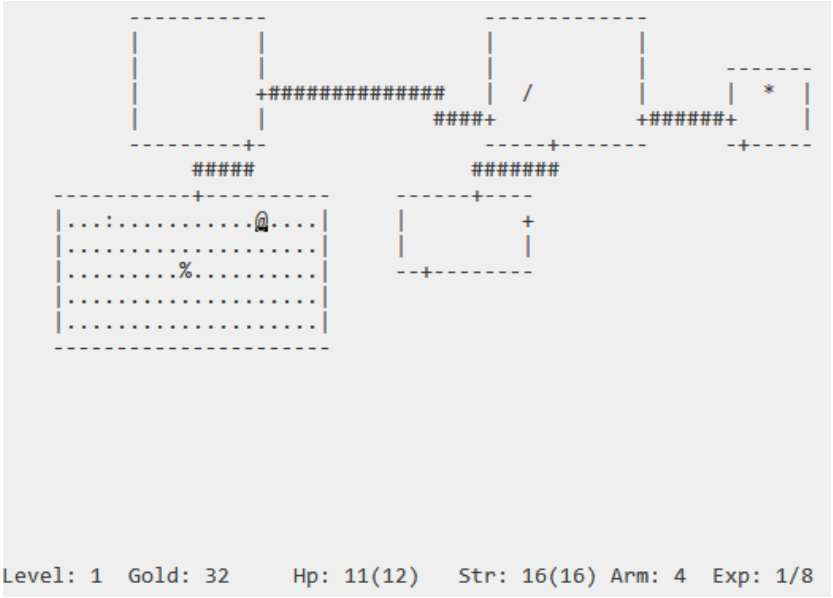
1. Procesuāli uzģenerēt triviālas tekstūras. Tādas tekstūras var piemēram būt vienkārša figūra, vai šaha rūtiņas.

2. Šai tekstūrai tiek pielietotas algoritmiskas deformācijas. Šis solis var tikt atkārtots vairākas reizes.

Šādā veidā tekstūra tiek glabāta kā vienkārša bāzes tekstūra un tas deformāciju vēsture. Ņemot vērā to, ka bāzes tekstūras izmērs ir ļoti mazs, tādā veidā var glabāt tūkstošiem tekstūru, aizņemot vien nenozīmīgu atmiņas daudzumu. Jāpiebilst, ka šādā veidā ģenerēto tekstūru izskatu ir grūti kontrolēt, bet kopumā tas ir pietiekami estētiski patīkams pateicoties simetrijai, kas raksturīga šāda veida tekstūrām.

2.2.3. Rogue

Izveidota 1983. gadā, Rogue ir viena no pazīstamākajām PĢS spēlēm, kurai sekoja vairākas citas spēles un kuras vārdā tika nosaukts spēļu žanrs Roguelike. Procesuāla ģenerācija šeit tiek izmantota, lai izveidotu spēles pasauli. Viss notiek apakšzemes cietumā, kas ir diezgan vienkārši ģenerēts, nejauši saliekot kvadrātus spēles telpā un tad savienojot šos kvadrātus, veidojot istabas un gaiteņus. Balstoties uz izpētīto literatūru, var secināt, ka šis ir visizplatītākais procesuālas ģenerācijas veids, pateicoties savai vienkāršībai, vēsturiskai nozīmei un arī praktiskam pielietojumam. Ekrānuzņēmumu no rogue spēles var redzēt 2.1.att, šobrīd tā ir pieejama Linux pakotnē bsdgames-nonfree. Tajā izmantojot simbolus tika izveidoti tuneļi, un spēlētājs atrodas @ simbola pozīcijā.



2.1. att. **Rogue ekrānuzņēmums**

2.2.4. Diablo

Diablo ir ļoti populāra spēļu sērija, ko izlaida Blizzard uzņēmums. Lielu daļu no šīs spēles aizņem pazemes izpēte, bet ietver sevī arī mijiedarbību ar nozīmīgiem tēliem augšpusē. Tieši tāpēc spēle ir sadalīta ar rokam veidotajā un procesuāli ģenerētajā daļā. Procesuāli ģenerēta pazeme ievieš variāciju spēlei, kas stipri pagarina spēles spēlējāmību, ka arī padara atkārtotu spēles iziešanu interesantāku. Līdzīgu tehniku izmantoja vairākas citas spēles, taču Diablo sērija tiek uzskatīta par sekmīgāko šajā kategorijā, tāpēc tas ir labs izpētes piemērs.

2.2.5. Dwarf Fortress

Viena no nozīmīgākajām spēlēm PĢS jomā izveidota 2002. gadā un daļēji iedvesmota no Rogue. Šī spēle ļoti intensīvi izmanto PĢS pasaules un notikumu ģenerācijai. Viena no interesantākajam tehnoloģijām ir vēstures ģenerācija. Dwarf fortress var procesuāli izveidot pasauli un tad modificēt to, simulējot laika gaitu. Šādas simulācijas ietvaros notiek tādas dabiskas parādības, ka upju pārplūšana. Šādā veidā, piemēram, vietās, kuras bieži notiek konkrēts notikums, var veidoties biomi, kas reprezentē noteiktus dzīvības nosacījumus šā bioma ietvaros.

2.2.6. Minecraft

Mūsdienās ļoti populāra spēle, ko Markus Persson izveidoja 2011. gadā. Pēc principa un PĢS tehnoloģijas ļoti līdzīga ir spēle "Terraria", kur lielāka atšķirība starp abām ir 3D un 2D spēles pasaule. Abas ir ļoti sekmīgas, un satur PĢS ka ļoti neatņemamu spēles daļu. Lielākā atšķirība šīm spēlēm no pārējam ir tāda, ka pasaule ir pilnīgi iznicināma un maināma. Pateicoties šai īpašībai, ģenerācijas process ir ļoti atvieglots. Piemēram, ja Rogue spēlē bija jāģenerē taisnstūri un gaiteņi starp šiem tiem, tad iekš "Terraria" jāģenerē tikai taisnstūri (kubi iekš Minecraft) un var ļaut lietotājam pašam veidot gaiteņus starp tiem. Abas spēles deterministiski ģenerē pasauli no sēklas, kas ļauj dalīties ar interesantām pasaulēm. (Kas uzrāda uz papildus sēklas izmantošanas veidu)

2.2.7. Infinite Mario Bros

Vēl viena Markus Persson spēle, kas labi ilustrē procesuālo ģenerāciju Platformu žanrā. Šī spēle ir ļoti līdzīga klasiskajai Nintendo spēlei "Super Mario Bros". Balstoties uz [4], šī spēle izmanto iezīmju vektoru, lai ritmiski izveidotu spēles pasauli. Vektors var ietvert sevī tādas iezīmes kā ienaidnieku skaitu, platformu skaitu u.c. Tas paver noderīgas iespējas, kuras ļauj piemēram pielāgot spēles karti tiem spēlētajiem, kuriem patīk kādas no šīm iezīmēm. Jāpiebilst, ka pēc autora personīgiem novērojumiem, spēles līmeņi ir diezgan viendabīgi, tāpēc mērķis izveidot lielu interesanta satura variāciju šeit nav pilnvērtīgi sasniegts.

2.2.8. Spelunky

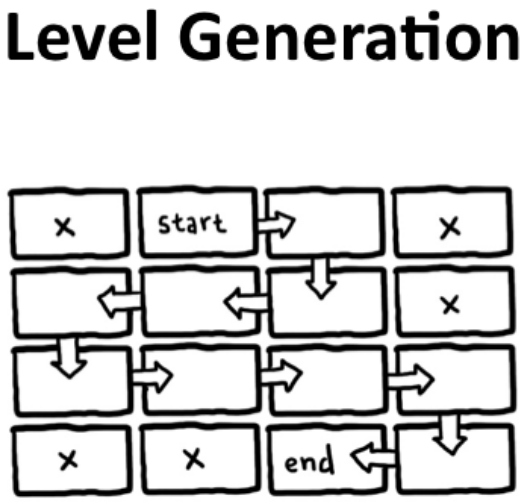
Spelunky oriģinālo versiju izlaida Derek Yu 2008. gadā. Šī ir platformera tipa spēle, kura tiek uzskatīta par ļoti labu sākuma punktu PĢS iesācējiem, pateicoties tas vienkāršajai implementācijai. Balstoties uz spēles autora rakstu [16], Spelunky PĢS sistēmu var izklastīt dažos soļos:

1. Spēles laukums sastāv no 16 istabām izklāstītam 4 x 4 režģī.

2. Ģenerācijas algoritms izvēlas vienu no augšējam rūtiņām kā sākuma pozīciju.

3. Tālāk tiek nejauši izvelēta blakus rūtiņa, un no šīs rūtiņas atkal tiek izvelēta cita blakus rūtiņa un tā līdz neesam nonākuši līdz vēlamai finiša rūtiņai.

4. Izieta ceļa laikā algoritms iezīmē, no kurienes mēs iegājām un kur mēs gājām tālāk no katras rūtiņas. Tādā veidā sadalot rūtiņas pēc parametriem “Ir izeja lejā”, “Ir izeja pa kreisi”, “Ir izeja pa labi” (un šo parametru kombinācijas). Izveidota labirinta piemēru var redzēt *2.2. att.*



2.2.att. **Spellunky labirints istabu novietojumam [16]**

5. Tālāk spēle piemeklē jau iepriekš ar rokām izveidotas istabas sagataves, kuras der noteiktai istabai ar noteiktu izeju skaitu un pozīcijām.

Pēc tam tiek veikti daži citi soļi, kurus apskatīsim vēlāk, bet pēc būtības jau tika izveidots daļēji nejaušs spēles līmenis, kas dod noteiktu variāciju. Jāpiebilst, ka šeit līdzīgi kā Minecraft ir ierobežota iespēja iznicināt sienas, jo pastāv istabas caur kuram algoritma laikā mēs neizgājām, taču šis istabas nav nepieciešamas spēles iziešanai. *2.3.att.* var redzēt ekrānuzņēmumu no spēles.

2.3.att. **Ekrānuzņēmums no spellunky [16]**

2.2.9. Borderlands

Borderlands un Borderlands 2 ir šaušanas spēle, kura izmanto PĢS sistēmu, lai ģenerētu nejaušos ieročus. Ieroči tiek ģenerēti, kombinējot vairākas savietojamas ieroču daļas. Kaut arī balstoties uz vairākām spēlētāju atsauksmēm [5], pirmajā daļā šo ieroču kombinācijas likās ļoti viendabīgas un nozīme bija tikai apmēram pieciem ieročiem, tikmēr otrajā daļa šī problēma likās daudz mazāk izteikta. Tas liecina par to, ka līdzīgu PĢS var veidot ar dažādu kvalitātes pakāpi. Tā piemēram *2.4. att*. Doti ieroči, kas atšķīrās tikai ar tekstūru.

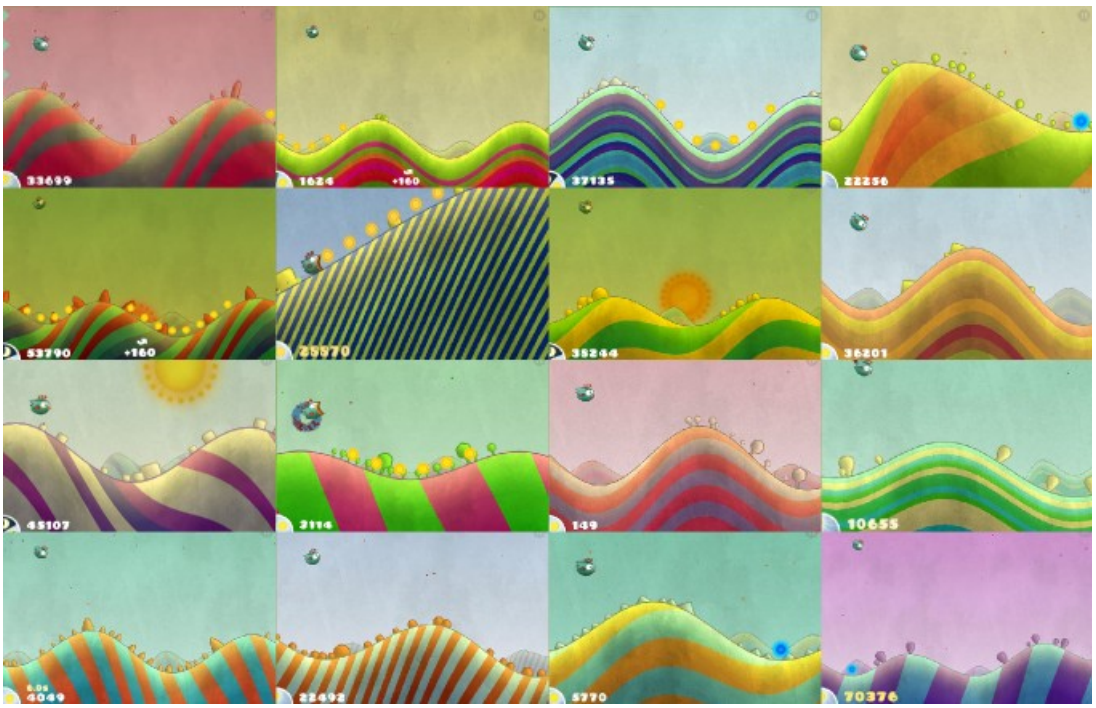
2.4. att. **Borderland viena ieroča variācijas [17]**

2.2.10. Spore

Spore ir populāra spēle par evolūciju, kur spēlētājs sāk ar vienkāršu mikroorganismu un beidz ar kosmosa tehnoloģijām. Līdzīgi ka iepriekšējās spēlēs, kosmoss šeit ir dinamiski ģenerēts, bet papildus tam, visas radības ir gan ģenerētas gan, arī iegūtas no citiem spēlētajiem, izmantojot spēles funkciju padalīties ar savām radībām, kuras savukārt arī ir kaut kādā pakāpē procesuāli ģenerētas. PĢS sistēma šeit ir arī izmantota, lai ģenerētu radību animācijas, par ko arī tiks stāstīts vēlāk darbā.

2.2.11. Tiny Wings

Tiny Wings ir populāra mobilā spēle ar vairākām kopijām mobilo spēļu tirgū, kas bieži vien norāda uz mobilas spēles kvalitāti. Līdzīgi kā vairākas citas mobilās spēles, tas process ir viendabīga vairākkārt atkārtota darbība ar ļoti mazām variācijām. Šādā situācijā Tiny Wings, balstoties uz [18] izmanto PĢS sistēmu nedaudz savādāk: tā ģenerē nejaušas, bet estētiski pilnvērtīgas tekstūras katru reizi, kad spēlētājs restartē spēli. Tas neizraisa nekādas spēlējamības izmaiņas, bet tajā pašā laikā iedod estētisku variāciju katram spēles ciklam, kas palīdz saglabāt spēlētāja interesi. Dažas no ģenerētam tekstūrām var redzēt *2.5. att.*



2.5. att. **Tiny wings ekrānuzņēmums ar dažādam tekstūrām [18]**

2.2.12. Left4Dead

Left4Dead ir 2008. gadā Valve Corporation izveidotā šaušanas spēle, kurā lielu lomus spēlē sarežģītība un noskaņojums. Balstoties uz [6] tieši tāpēc Valve izmanto PĢS, kas mēģina balansēt spēlētāja emocionālo intensitāti. Ienaidnieku starta pozīcijas un bonusa priekšmeti (tādi ka papildus ieroči un medicīniskās pakas) ir izklāstīti, spēles laikā aprēķinātās vietās, balstoties uz informāciju par spēlētāja panākumiem. Jā spēlētajam iet pārāk grūti, spēle dos vairāk bonusu un mazāk ienaidnieku. Jā spēlētajam pēc spēles mākslīgā intelekta domām paliek garlaicīgi vai pārāk viegli, tad ienaidnieku skaits palielinās un to instances tiek izveidotas mazāk paredzamās vietās. Balstoties uz [7], spēles mūzika arī ir vairāk vai mazāk procesuāli ģenerēta. Tā sastāv no vairākiem iepriekš sagatavotiem mūzikas celiņiem, kas spēles laikā tiek apvietoti, lai veidotu personīgu pieredzi katram spēlētajam balstoties uz viņa pieredzi spēlē. Bez tam spēle izmanto ar roku veidotus līmeņus, bet simulē līmeņu variācijas izmantojot tādus nejauši pozicionētus objektus, ka, piemēram, mašīnas kuras bloķē līmeņa daļas un dod dažādas izmantošanas iespējas.

2.2.13. No Man's Sky

Šī pietiekami jauna spēle ir noderīgs piemērs tāpēc, ka neskatoties uz ļoti lieliem solījumiem, tā saņēma ļoti negatīvu spēlētāju komūnas reakciju. Viens no nozīmīgajiem iemesliem tam ir tas, ka izstrādātāji solīja miljoniem unikālu sugu un planētu, taču beigās izrādījās gandrīz visas sugas ir ļoti līdzīgas viena otrai. Tas norāda uz variāciju problēmu PĢS sistēmās, kur neatlasīti rezultāti, kaut arī dod lielu objektu skaitu, bet tajā pašā laikā neuzlabo spēles spēlējamību. Parodija uz šo spēli "No Mario sky" (kas tika pārsaukta "DMCA's sky") tika vēlāk izveidota un pieejama par brīvu. Tā mēģina imitēt "No Man's Sky" tehnoloģijas 2D telpā un papildus procesuāli ģenerē spēles mūziku.

2.2.14. Yavalath

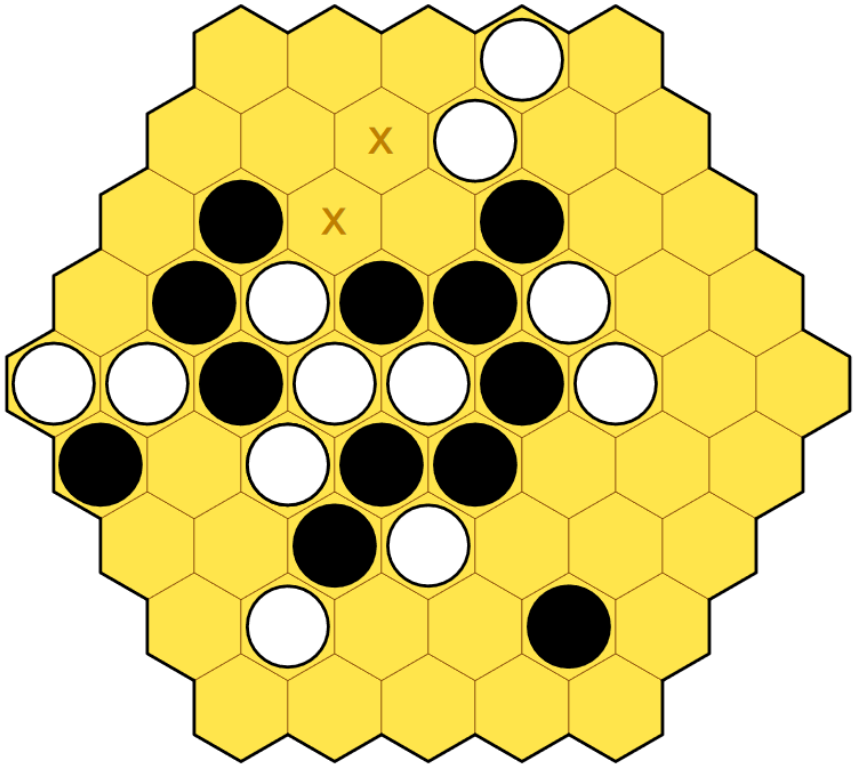
Yavalath ir procesuāli izveidota galda spēle. Tās noteikumu ģenerēja Ludi sistēma, par kuru tiks stāstīts vēlāk darbā. Yavalath noteikumi ir diezgan vienkārši

a) Spēle piedalās 2 spēlētāji, kas veic gājienus pēc kārtas, liekot savas krāsas akmeņus uz spēles rūtiņām.

b) Spēles laukums un katra spēles rūtiņa ir sešstūra formā.

c) Jā spēlētājs savāc 4 akmeņus vienā rindā, viņš uzvar. Savācot 3 vienā rindā, spēlētājs zaudē.

Šī ir pirmā izlaista spēle, kuras notiekumus uzģenerēja dators un kura vēl šobrīd ir pietiekami populāra. No sākuma parasti māc šaubas par c daļā aprakstīto noteikumu loģiskumu, taču apskatot piemēru 2.6 att, varam redzēt interesantu gājienu, kur gadījumā ja melnie uzliek "akmeni" uz x, baltie savā nākamajā gājienā nevarētu viņus apstādināt savākt 4 pēc kārtas, jo tajā vietā jau ir 2 baltie "akmeņi", un baltie zaudēs ja izliks 3 vienā rindā.



2.6. att. **Yavalath spēles stratēģisks gājiens [11]**

2.2.15. Dwarf Quest

Ir vēl viena Roguelike spēle 3D telpā. Tās lielākā īpatnība ir tāda, kā pirms kartes ģenerācijas, tiek uzģenerēts "uzdevumu grafs". Šajā grafā tiek izveidots ceļš, kas spēlētajam jāiziet, lai paveiktu līmeni. Tādā veidā viens no vienkāršākajiem piemēriem ir iziet no sākuma, tālāk paņemt atslēgu un tālāk atvērt durvis. Protams, grafa īpašības var tikt izmantotas šajā ceļā, piemēram, veidojot sazarojumus un vairākus alternatīvus ceļus starp diviem punktiem.

2.2.16. Unexplored

Relatīvi jauna spēle, ko izveidoja Joris Dormans. Tā atkal ir līdzīga "Rogue" spēlēm, bet izmanto formālās gramatikas, par kurām tiks stāstīts vēlāk darbā. Tās izteikta īpašība ir tāda, ka līmeņa ceļi ir cikliski. Būtībā tas nozīmē kā vienā spēles laikā vienā un tajā pašā vietā mums, iespējams, būs vairākkārt jāatgriežas pēc kādu noteiktu uzdevumu izpildes. J. Dormans bieži uzsver ciklisku ceļu priekšrocības, kurām arī tiks veltīta atsevišķa nodaļa darbā.

2.2.17. Citas spēles

Jāpiemin, ka PĢS sistēmas ietver sevī tas, kuras saturu ģenerē pilnīgi nejauši. Tādas spēles ir ļoti izplatītas, bet to ģenerēšanas sarežģītība ir pārāk maza, lai iekļautu šajā darbā. Tomēr dažkārt nejaušas ģenerācijas sistēmām var pielietot mazus procesuālus uzlabojumus. Piemēram, populārajā spēlē "Tetris", katrs nākamais spēles klucis tiek izvēlēts pilnīgi nejauši, taču [8] pierādīja, ka šādā veidā spēlētājs var zaudēt neatkarībā no viņa rīcības, ja viņam vienkārši nepaveiksies ar nejaušiem notikumiem. Iespējamība ir ļoti maza, bet šeit mēs varam pielietot vienkāršus procesuālas ģenerācijas likumus, kas ļaus mums izvairīties no šādas situācijas.

2.3. Sistēmas, kas veido spēļu noteikumus

2.3.1. Ievads

Iepriekšējā nodaļā tika apskatītas dažas spēles, kurās tiek izmantots PĢS. Šajās spēlēs parasti ir jau definēti noteikumi, un ģenerācija notiek pret elementiem, kurus spēlētājs parasti var sastapt, spēlējot pēc šiem noteikumiem spēles pasaulē. Tikmēr līmeņa elementu saturs nav vienīgais, kas var tikt ģenerēts. Īpaši pēdējā laikā tiek pētītas metodes, ka ģenerēt spēles noteikumus, kas ļauj izveidot spēli gandrīz pilnībā bez cilvēka iesaistīšanas.

Vismaz šobrīd tehnoloģijas vēl nav tik augsti attīstītas, lai tas spētu autonomi ģenerēt pilnas spēles, bet tas neliedz iespēju daļēji ģenerēt spēles un to noteikumus. Lai apliecinātu, ka pilnu spēļu ģenerācija ir iespējama, var apskatīt robežu piemēru: Jebkura programma sastāv no bitiem, kas nozīmē, ka, ar gandrīz neierobežotu laiku, mēs varam ģenerēt visus šo bitu variācijas ar noteiktu garuma ierobežojumu. Bet pat tad rodas problēma, ka ir vajadzīgs algoritms, kas pārbauda vai uzģenerēta bitu virkne ir spēle un vai vispār tā ir programma, kas darbosies datora vidē.

No iepriekšējiem apgalvojumiem, lielāka problēma ir iecerēta uzdevuma apjoms, jo jā piemēram bitu virkne sastāvētu no 30 tādiem bitiem, tad nebūtu nekādu problēmu izskatīt visus variantus, jo to skaits būtu 2^{30} = 1073741824, taču jau 50 variantiem, tas sagādātu grūtības, jo variantu skaits aug ļoti strauji un 2^{50} = 1.1258999e+15 varianti.

Bet pat jā spēli var izveidot izmantojot 30 bitus, ir vajadzīgs algoritms, kas šo spēli pārbauda un pasaka vai tā ir spēlējama. Ir loģiski izsecināt, ka vieglākais veids to paveikt ir

ļaut mākslīgam intelektam izspēlēt šo spēli. Šo iespēju tuvāk pēta mākslīga intelekta sistēmu nozare un piemēram Mnith u.c. savā darbā [15] apraksta to, ka šāda algoritma sekmīgai darbībai ir vajadzīgs novērtējums, kas parasti spēlēs tiek īstenots, izmantojot punktu skaitu, kas tiek piešķirts par pareizo rīcību izpildi. Šeit veidojas problēma, jo jā mēs nejauši izveidojam spēles noteikumus, tajos būs diezgan grūti , bet ne neiespējami, norādīt punktu skaitu , ko mākslīgais intelekts iegūst, veicot noteiktas darbības spēlē.

Pretēji bitu virknei, dabīgas un datoru valodas dod iespēju datoram saprast spēles uzvaras noteikumus, kas nozīmē, ka jāizmanto kāda no šādām valodām. Viena iespēja ir izmantot dabīgu valodu, taču šeit mēs atgriežamies pie problēmas, kur spēļu variantu skaits ir pārāk liels. Pat jā ģenerācijas algoritms spēj "rakstīt spēļu noteikumus" dabīgajā valodā tikai ar pareiziem teikumiem, tik un tā ir skaidrs, ka dabīgas valodās (Piemēram, latviešu) var aprakstīt jebkuru spēli, kas var eksistēt, kas atkal rāda gandrīz bezgalīgo variantu skaitu. Tāda pati problēma pastāv, ja izmantosim piemēram C++ valodu, jo kaut arī tā ir vairāk ierobežota, tāpat tā var izveidot jebkuru iespējamo programmu un spēli.

Ņemot vērā iepriekšējos argumentus, autors nav atradis nevienu īstenu noteikumu ģeneratoru, kas pietiekami neierobežoti varētu ģenerēt spēles noteikumus, taču tālāk tiks izskatīti daži darbi, kas tuvinās noteikumu ģenerācijai un tiek uzskatīta par tādu, jo pilnīga noteikumu ģenerācija neizskatās iespējama.

2.3.2. Ludi

Ludi ir PĢS sistēma, kas domāta galda spēļu ģenerēšanai. To izveidoja Cameron Browne un Frederic Maire. Balstoties uz viņu publikāciju [11], ievadā apskatītās problēmas viņi atrisināja, izmantojot ļoti ierobežotu valodu, kurā varēja definēt tikai noteikumus tikai galda spēlēm. Spēles piemērs krustiņu un nullīšu spēlei viņu valodā atrodams [11] :

```
(game Tic-Tac-Toe
   (board
    (tiling square)
    (size 3 3)
)
(win (in-a-row 3))
```

)

Atkodējot šos noteikumus, nosaka, ka spēles nosaukums ir “Tic-Tac-Toe”. Tālāk tiek definēts spēles laukums, kas šajā gadījumā ir kvadrāts (netika minēts, bet tiek uzskatīts pēc noklusējuma, ja nekas nav minēts) kas sastāv no 3x3 kvadrāta rūtiņām. Uzvaras nosacījums “win (in-a-row 3)”, apzīmē to, ka uzvarai ir nepieciešams aizņemt 3 rūtiņas pēc kārtas. Tālāk viņu darbā tika izmantotas papildus detaļu definīcijas, piemēram, tika definēti spēlētāji (šajā gadījumā krustiņi un nullītes).

Šeit rodas jautājums par to, ka šāda valoda risina ievadā izrunātas problēmas, un atbilde ir tāda, ka šīs algoritms drīzāk kombinē spēles noteikumus, nekā ģenerē tos. Pirmkārt Ludi izveidoto spēļu noteikumos vienmēr pastāv daži spēli definējoši noteikumi, ka, piemēram, spēle tiek spēlēta 2D telpā un spēlētāji veic gājienus pēc kārtas. Kopā ar dažiem citiem noteikumiem, ģenerēto spēļu skaits paliek pietiekami ierobežots, un spēles noteikumi ir drīzāk dažu līdzīgu spēļu variācijas. Piemēram, šāda sistēma varētu izveidotas krustiņu un nullīšu spēles, kur spēles laukums ir 2x2, 4x4 vai 10x10, uzvaras nosacījums varētu būt savākt 5 elementus vienā rindā, vai arī spēles laukums varētu būt trīsstūris.

Neskatoties uz šādiem ierobežojumiem, Ludi ir sekmīgi izveidojusi vismaz divas spēles, viena no kurām, “Yavalath”, tika aprakstīta jau iepriekš. “Yavalath” arī varētu aprakstīt līdzīgi krustiņu un nullīšu spēlei, kur laukums ir sešstūris, kas sadalīts rūtiņas tādā veidā, ka sešstūra mala ir 5 rūtiņas liela un uzvaras nosacījums ir aizņemt 4 rūtiņas vienā līnijā. Taču interesantākais “Yavalath” nosacījums ir tāds, ka, aizņemot 3 rūtiņas, spēlētājs zaudē. Uzvaras nosacījums tiek pārbaudīts pirmais, un jā ir savāktas 4 rūtiņas, nosacījums par 3 rūtiņām neizpildās.

Lai pārbaudītu spēles kvalitāti, mākslīgais intelekts vairākkārt izspēlēja šo spēli, lai pārbaudītu tās līdzsvaru, interesantumu un ilgumu. Detalizētāk kas tika mērīts var apskatīt [11].

Tika veikts neliels eksperiments, kur šī darba autors spēlēja “Yavalath” spēli ar dažādiem cilvēkiem. Šajā eksperimentā tika iegūti daži interesanti rezultāti:

1. Izdzirdot spēles noteikumus, ļoti bieži izskanēja viedoklis, ka uzvaras un zaudējuma noteikumi izklausās neloģiski un papildus noteikumam par zaudējumu nav nekādas jēgas.

2. Spēle izrādījās ļoti interesanta gan spēles sākumā, gan arī beigās. Neskatoties uz tik mazu noteikumu skaitu, spēles dziļums bija ļoti liels un intensitāte saglabājas visas spēles laikā.

No pirmā punkta varam izsecināt, ka cilvēks parasti neiedomātos kombinēt šos divus uzvaras/zaudējuma nosacījumus un pat jā iedomātos, pastāv liela iespēja, ka viņš tos atmestu. Tam pretī dators nevar novērtēt spēles kvalitāti neveicot strikti definētus eksperimentus,

kas ļauj ļoti objektīvi novērtēt spēli un nonākt pie secinājumiem par to, cik spēlējama tā ir. Jāpiemin, ka Ludi izmantoja algoritmu, lai izvelētos spēļu kopu, ko tas uzskatīja par interesantu, bet tas spēļu kopas novērtēšanu vēlāk veica cilvēki, tāpēc droši vien šobrīd vēl nevar pateikt, ka veiksmīgu spēli dators ģenerējis patstāvīgi.

2.3.3. Citas sistēmas

Pastāv vairākas citas sistēmas, kur tiek definēti spēles noteikumi, taču kopējā gadījumā tie ir līdzīgi Ludi sistēmai un izmanto iepriekš definēto noteikumu kopu. Dažreiz par noteikumu ģeneratoriem arī tiek uzskatītas sistēmas, kur vienkārši tiek definētas darbības, kas notiek piemēram saskaroties spēles varonim ar ienaidnieku. Šādas sistēmas pēc autora zināšanām nav izveidojušas spēles, kas tiktu sekmīgi izlaistas, līdzīgi kā to izdarīja Ludi.

2.3.4. Secinājumi

Tika aprakstītas dažas problēmas, kas saistītas ar noteikumu ģenerēšanu spēlēs un kādā veidā pastāvošas sistēmas apiet šīs problēmas. Tika izveidoti divi secinājumi:

a) Spēļu ģenerēšanas sistēmas šobrīd īsti neģenerē noteikumus tradicionālajā nozīmē, bet drīzāk kombinē un evolucionē iepriekš definētas noteikumu daļas.

b) Neskatoties uz ierobežojumiem, sadarbība ar cilvēku spēlētāju palīdzību, tika izveidotas dažas ļoti sekmīgas spēles, kas tika arī publicētas.

2.4. Secinājumi

Apskatot jau izveidotas spēles un sistēmas, kas ģenerē šīs spēles, kļuva skaidrs, ka PĢS spēlēs pastāv jau pietiekami sen, bet tā izmantošanas iemesli laiku gaitā var mainīties.

Apskatītajās spēlēs un sistēmas tika atrasti vairāki paņēmieni un noteikts, kādā veidā šos paņēmienus var kombinēt, lai panāktu vēlamo rezultātu. Tika pamanīta arī tendence procesuāli ģenerēt spēles konkrētiem žanriem un pat apakšžanriem. Tā piemēram viens no populārākiem spēles žanriem ar PĢS ir spēles, kas līdzinās Rogue, jeb Roguelike.

Roguelike spēlēs pirmkārt ir diezgan zems sarežģītības līmenis PĢS sistēmu prasībām, un daļēji tas panākts pateicoties tam, ka PĢS šajā žanrā ir diezgan plaši izpētīts, kas arī ir šī darba uzdevums attiecībā uz citām spēlēm. Otrkārt Roguelike ir ļoti labi piemērots PĢS, jo viens no tā nosacījumiem ir tāds, ka progresu nevar saglabāt, un zaudējums nozīmē, ka spēle jāsāk no jauna. Ja ar šādiem nosacījumiem līmeņi nemainītos un spēlētajam būtu jāiziet viens un tas pats saturs no jauna, tas stipri pasliktinātu šo spēli padarot to garlaicīgu.

Ārpus Roguelike, ir vairāki faktori, kuri nosaka to, kā bez PĢS spēles spēlējamība ir ļoti zema, piemēram, spēlēs, kuru centrāla mehānika ir satura atklāšana būtu pietiekami garlaicīgas mūsdienu informācijas apmaiņu dēļ, kad ir iespēja uzzināt visu par spēli no citu cilvēku rakstiem, taču, procesuāli ģenerējot, šādu saturu, ir iespēja izveidot unikālu pieredzi katram spēlētajam.

3. RĪKI UN PAŅĒMIENI

3.1. Ievads

Viens no darba uzdevumiem bija paātrināt un atvieglot PĢS sistēmu izstrādi. Lai šo uzdevumu izpildītu, ir jāapskata paņēmieni, kurus atklāja citi cilvēki un kurus bieži izmanto vai arī potenciāli var izmantot PĢS sistēmās. Šādu paņēmienu skaits ir ļoti liels un ir grūti definēt, kur viens paņēmiens ir cita paņēmiena variācija un kur divi paņēmieni ir pavisam atšķirīgi, bet darbā tiks aprakstīti daži paņēmieni. Tas, kadi paņēmieni tiek iekļauti, tiek noteikts balstoties uz

a) Paņēmiena vienkāršību – PĢS sistēmas parasti ir diezgan sarežģītas pat tad, kad tiek izmantoti vienkārši paņēmieni. Neskatoties uz to, pastāv interesanti paņēmieni, kuru saprašana un izmantošana ir daudz sarežģītāka. Šādi paņēmieni ir iekļauti PĢS kategorijā, bet to aprakstīšana visdrīzāk neļaus atvieglot un paātrināt PĢS sistēmu izstrādi vienkāršām spēlēm. Tas nozīmē, ka tie nepalīdz darba mērķa īstenošanai un netiks iekļauti aprakstītos paņēmienos.

b) Paņēmienu izmantošana eksperimentā – Vēlāk darbā novitātes dēļ tiks piedāvāti eksperimenti, kuru jēga būs aprakstīt vienkāršus spēles un PĢS paņēmienus, kas vajadzīgi šo spēļu īstenošanai. Lai sagatavotu lasītāju šiem eksperimentiem, šeit tiks izskatīti paņēmieni, kuri ir identiski vai ļoti līdzīgi eksperimentos izmantotajiem.

c) Paņēmienu noderīgums spēlēs – Agrāk tika minēts, ka PĢS tiek plaši izmantots sfērās ārpus spēlēm. Piemēram, ir iespējams izveidot simetrisku tekstūru un izdrukāt to uz jakas, vai arī saplānot jaunas pilsētas ceļu un māju izvietojumu. Nodaļā aprakstītie paņēmieni būs izmantojami tieši spēļu izveidei un spēles spēlējamības uzlabošanai. Jāpiemin, ka, piemēram, tekstūru ģenerēšana Tiny Wings spēlē tomēr uzlabo spēlējamību ieviešot noteiktu variāciju atkārtotam spēļu iziešanām, bet pārsvarā tiks aprakstītas metodes, kas ietekmē spēles sarežģītību un ievieš spēlēšanas pieredzes variācijas.

Paņēmieni ir nelielā mērā gatavi risinājumu PĢS ieviešanai spēlē, bet ļoti svarīgu lomu spēlē arī PĢS rīki. Šeit rīki neapzīmē datni, kas var ģenerēt spēļu pasauli, bet gan drīzāk matemātiskus vai algoritmiskus rīkus, kuru izprašana ir vajadzīga PĢS sistēmu veidošanā.

Parasti vienā spēlē netiek izmantots tikai viens rīks, bet gan to kombinācijas, kuras dažos gadījumos definē PĢS paņēmienu.

Jāpiemin, ka aprakstītie rīki un paņēmieni bieži vien ir tikai tuvinājums, jeb matemātisks modelis tam, kas parasti varētu tikt izmantots praksē.

3.2. Rīki

3.2.1. Nejaušības ģenerators

Ka, jau iepriekš minēts, spēles kuru saturs veidots izmantojot tikai nejaušus elementus, nav pietiekami sarežģīts, lai to pētītu procesuālas ģenerācijas ietvaros. Neskatoties uz to, nejaušības ģenerators spēlē nozīmīgu lomu PĢS sistēmās. PĢS veidošanu procesu var uzskatīt par nejaušo elementu ģeneratoru, kas ierobežots ar vairākiem noteikumiem. Piemēram, PĢS sistēmas bieži vien mēģina uzģenerēt sakarīgus vārdus kādā no dabiskajām valodām, bet tajā pašā laikā ir viegli pierādīt, ka, nejauši kombinējot burtus, ir iespējams paveikt to pašu, kaut arī varbūtība iegūt vārdu, kas labi skan, ir ļoti maza, un pielietojot pat visvienkāršākos noteikums, samazinot iespējamo iznākumu skaitu, mēs varam stipri palielināt tāda vārda atrašanas varbūtību.

Savā ziņā vairāku PĢS algoritmu uzdevums ir maksimāli ierobežot visu iespējamo iznākumu kopu, un tad veikt nejaušu izlasi no šīs pašas kopas.

Turklāt nejaušo elementu ģenerators zemā līmenī tiek izmantots gandrīz jebkurā PĢS sistēmā, kur ir nepieciešama jebkāda nenoteiktība. Tāpēc ir vērts apskatīt, ka tā strādā. Dators īsti nevar uzģenerēt patiešām nejaušus elementus, tāpēc tiek izmantotas nejaušības simulācijas metodes, kas pārsvarā veido grūti paredzamas skaitļu virknes. No tiem visbiežāk izmanto divus veidus. Abiem šiem veidiem piemīt deterministiskuma īpašība, kas nozīmē, ka, izmantojot vienu un to pašu sēklu, mēs varam iegūt vienus un tos pašus rezultātus. Veidi tiek aprakstīti zemāk.

3.2.1.1. Nejaušo skaitļu ģenerators (NSĢ)

Veido skaitļu rindu. Balstoties uz [9] aprakstītiem algoritmiem, šī rinda aprēķina katru nākamo skaitli izmantojot iepriekšējos. Tādā veidā katru nākamo skaitli, ko izdod šis ģenerators, ir iespējams aprēķināt tikai aprēķinot visus iepriekšējos skaitļus, kas arī dod aprēķināto skaitļu neparedzamību. Šī īpašība arī uzlabo algoritma ātrdarbību visbiežāk

sastopamajās situācijās, jo kamēr n-tā skaitļa aprēķiniem ir jāpielieto vairāki aprēķinu cikli, nākamo skaitli no jebkura stāvokļa var aprēķināt ļoti ātri. Formulā (3.1) ir piedāvāta vienkārša NSĢ rinda, kur a, b un m ir konstantes. [9] ir pieradījums kā ar pareizi izvelētam konstantēm šāda rinda var ģenerēt visus iespējamos skaitļus no 0 līdz m.

$$y_{n+1} = (a * y_n + b) \, mod \, m \tag{3.1}$$

3.2.1.1. Jaucejalgoritms

Parasti tas izmanto noteiktu atslēgu un, pielietojot šo atslēgu ieejas vārdam, izmantojot noteiktus algoritmus, tas izejā izdod simbolu virkni. Šī simbolu virkne ir determinēta un jebkuram ieejas vārdam un atslēgas pāriem, izejas vārds būs viens un tas pats, ņemot vērā, ka tiks izmantots viens algoritms. Ir vairākas jaucejalgoritmu variācijas, bet pārsvarā lai tas iznākumu būtu sarežģīti paredzēt, to darba laiks ir salīdzinoši lēns, tāpēc tas ir retāk izmantots PĢS.

3.2.2. PĢS Sēkla

Sēkla ir neliels starta elements, kas tiek izmantots, ka atslēga procesuālas pasaules ģenerācijai un tai ir liela nozīme un plašas izmantošanas iespējas PĢS sistēmās. Sēkla parasti sastāv no viena vai vairākiem skaitļiem kuriem nepiemīt nekādas aprakstošas īpašības, un sēklas izvēle var noteikt uzģenerētas pasaules kvalitāti. Vienkāršākajā gadījumā šī sēkla tiek padota zema līmeņa nejaušo skaitļu ģeneratoram, bet pastāv arī citi izmantošanas veidi, kuri tiks pieminēti citās nodaļās. Viena no svarīgākajam sēklas īpašībām ir tas, ka jebkuram vienam un tam pašam sēklas un determinēta PĢS algoritma pārim rezultāts vienmēr ir viens un tas pats. Šī īpašība paver tādas iespējas kā

a) Uzģenerētas pasaules saglabāšana, kas nepieciešama iesāktas spēles tupināšanai.

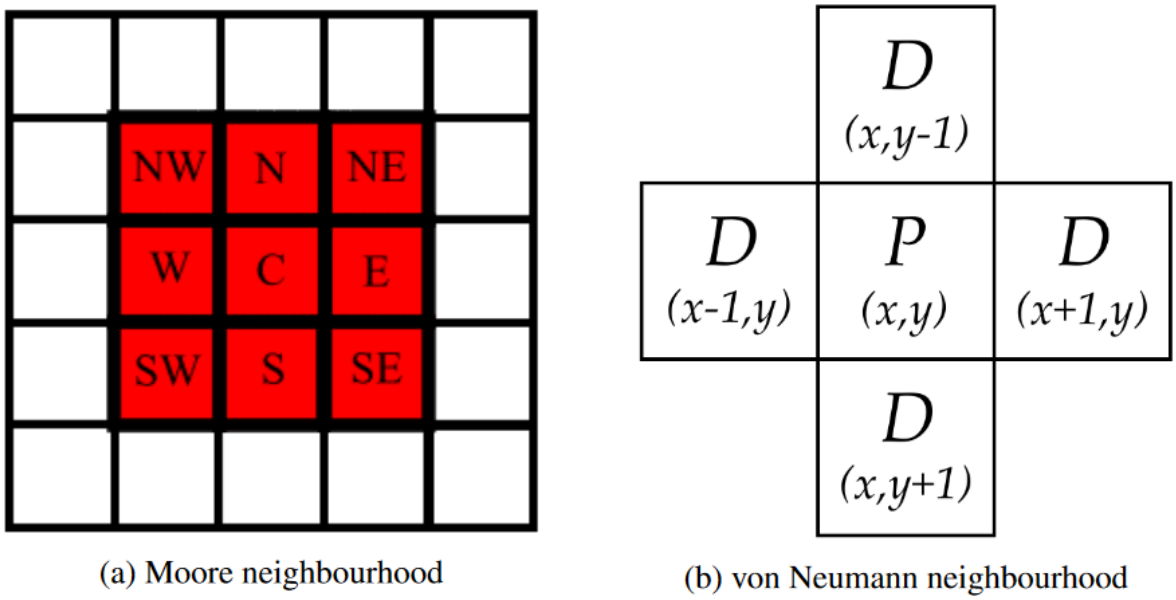
b) Dalīšanās ar kvalitatīvam vai interesantām spēles pasaulēm. Iedodot kādam citam cilvēkam mazu sēklu, šis cilvēks var uzģenerēt identisku pasauli.

c) Dinamiska pasauļu ģenerācija, kas risina ierobežotas atmiņas problēmas. Par to tiks runāts vēlāk darbā.

d) Noteiktu veidu sēklas var izmantot, lai veiktu kontrolētas izmaiņas šajās pasaulēs, ja tajās pastāv korelācijas starp sēklu un pasaules īpašībām. Tas ir izdevīgi piemēram evolucionārās PĢS sistēmās, par kurām tiks runāts vēlāk darbā.

3.2.3. Šūnu automāts

Šūnu automāts (Cellular automata) ir diskrēts skaitļošanas modelis, kas sastāv no n-dimensionāla šūnu laukuma, šūnu stāvokļu kopas un stāvokļu parejas noteikumiem, kuri parasti ir atkarīgi no šūnas kaimiņiem. Modelis sakās pirmajā solī un katrā nākamajā solī tas mainās saskaņā ar parejas likumiem. Tas tiek izmantots vairākās nozarēs arī ārpus datorzinātnēm. Jāpiemin, ka šūnu kaimiņi ir tās šūnas, kuras atrodas 1-n soļus no šīs šūnas, kur n var definēt atkarībā no automāta vajadzībām. Soļa virzienu arī var definēt, bet 2 populārākie veidi 2D automātiem ir "Moor neighbourhood", kur solis ir viena šūna vertikālajā, horizontālajā vai diagonālā virzienā un "Neumann neighbourhood", kur netiek, izmantoti diagonāli soļi. *3.1.att* var redzēt abus piemērus.



3.1. att. **Neumann un Moore kaimiņu šūnu varianti šūnu automātos [19]**

Johnson L. un citi, savā darbā, [10] parādīja, ka, izmantojot šūnu automātu, var ģenerēt bezgalīgas pazemes alas. Pēc principa tas strādā apmēram šādi:

1. Uz divu dimensiju laukuma nejaušā kārtā tiek salikti "izejami" un "neizejami" punkti.

2. Izmantojot noteiktus šūnu automāta likumus, nejaušas šūnas tiek izbīdītas grupās: tie reģioni, kur šūnu ir daudz, tiek pilnībā pārklāti ar "neizejamam" šūnām.

3. Otrais solis tiek vienlaikus pielietots arī "izejamām" šūnām.

Kontrolējot šos automāta stāvokļu likumus, tiek izveidoti "Attīrīti" tuneļi, kas strādā līdzīgi labirintiem.

Viena nozīmīga šūnu automāta īpašība ir tāda, ka tas strādā O(n) laikā, jo algoritma solis tiek pielietots katrai šūnai vienu reizi un tiek pārbaudīts katrs viņa kaimiņš, kuru vienmēr ir konstants skaits. Tas dod iespēju ļoti ātri veidot procesuāli ģenerētas pasaules. Cita Šūnu Automāta priekšrocība ir tāda, kā rezultāta īpašības pēc vairākiem soļiem ir iespējams noteiktā pakāpē paredzēt un kontrolēt balstoties uz stāvokļu parejas noteikumiem, kas ir ļoti noderīgi vairākos PĢS algoritmos.

3.2.4. Formālas gramatikas

Formālas gramatikas ir simbolu virknes konstruēšanas noteikumu kopa, kas nosaka, kādā veidā mēs varam pārveidot esošu simbolu virkni aizvietojot tas apakšvirknes ar citām simbolu virknēm. Tā piemēram mēs varam definēt noteikumus

1. A -> Ba

2. B -> BB

3. B -> b

4. B -> c

Lai izmantotu šos likumus, mēs izejam rindai no kreisās uz labo pusi, izvēlamies vienu no savietojamiem aizvietošanas likumiem un aizvietojam esošo simbolu (vai simbolu virkni) ar likumā norādīto. Izmantojot piemēra dotos noteikumus, no starta rindas "A", mēs varam iegūt tādas rindas kā "Ba", "BBBa", "bcba", u.t.t. Parasti ar lielajiem burtiem apzīmē neterminālus, jeb palīgsimbolus (tos, kurus var vēl pārveidot) un ar mazajiem – terminālus, jeb gala simbolus (tos, kurus nesastapsim likuma kreisajā pusē).

Jāpiemin, ka formālajās gramatikās izšķir deterministiskas un nedeterministiskas gramatikas, kur deterministiskas gramatikas vairāki likumi nevar saturēt vienu un to pašu simbolu virkni to kreisajā pusē (tātad iedotajā piemērā, 2., 3. un 4. noteikums nevarētu eksistēt vienā un tajā pašā gramatikā), tādā veidā pārliecinoties, ka vienu simbolu virkni var aizvietot tikai vienā veidā un jebkurš sākotnēja vārda un soļu skaita pāris izdos vienu un to pašu rindu.

PĢS sistēmās mūs biežāk interesē gramatikas ar noteiktu starta simbolu virkni, tāpēc variācijas ir iespējamas tikai nedeterministiskās gramatikās, kur varam izvelēties starp vairākiem simbolu virknes aizvietošanas variantiem un nonākt pie vairākām vārda variācijām. Pastāv vairāki veidi noteikt kādu likumu izmantosim, kad atrodam simbolu virkni, kurai pastāv vairāki šādi likumi:

a) Vienkāršākais veids ir izmantot nejaušo likumu, kas dos pilnīgi nejaušus rezultātus šis gramatikas ietvaros.

b) Cits veids ir izmantot sēklu, piemēram, katru reizi, kad jāizdara izvēle, mēs sadalām sēklu ar X un atlikumam izvelkam moduli pēc variantu skaita. Iegūto skaitli izmantojam, lai izvēlētos likumu. Kad sēkla izsmelta, vai nu atjaunot sēklu, vai izveidot jaunu izmantojot pēdējo.

Otrajam variantam ir priekšrocība, pateicoties tam, ka ir iespēja daļēji paredzēt izmaiņas rezultātā, veicot izmaiņas sēklā. Šī īpašība paliks ļoti noderīga evolucionāras atlases paņēmienā, kuru apskatīsim vēlāk darbā.

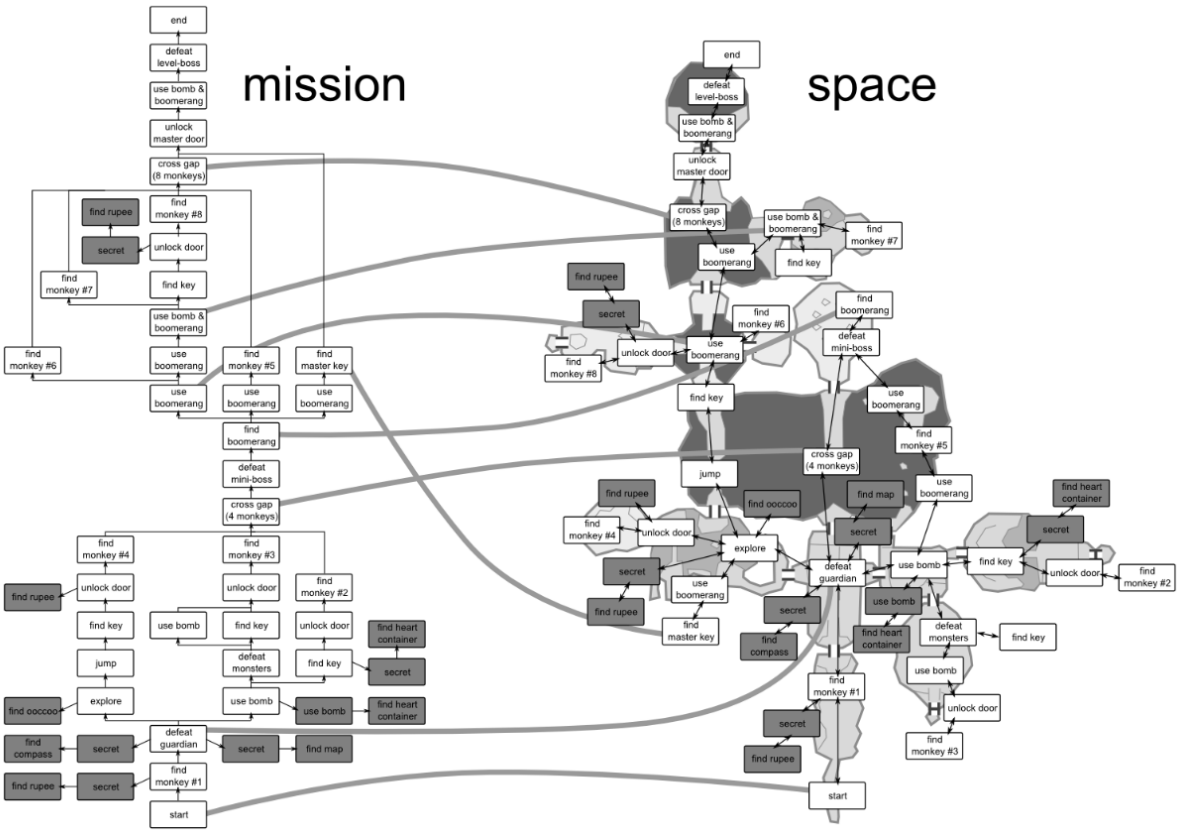
Lai uzreiz parādītu Formālo Gramatiku pielietojumu PĢS sistēmās, var izmantot vieglu piemēru: Izmantojot apskatīto piemēru, varam izveidot vienkāršu lineāru līmeni, kas sastāvēs no ienaidniekiem "b" un "c", bet vienmēr beigsies ar īpašo ienaidnieku "a".

3.2.5. Grafu gramatikas

Formālas gramatikas ir diezgan universālas un var tikt pielietotas ne tikai simbolu virknēm, bet arī dažādu dimensiju spēļu laukumiem, ģeometriskam figūrām, tekstūras deformācijām un grafiem. PĢS sistēmās īpaši noderīgi ir grafi, jo to īpašības ir pietiekami tuvas vairākām vienkāršām spēlēm. Lai to labāk parādītu, vienkāršs uzdevums abstraktai spēlei ir attēlots *3.2. att.*

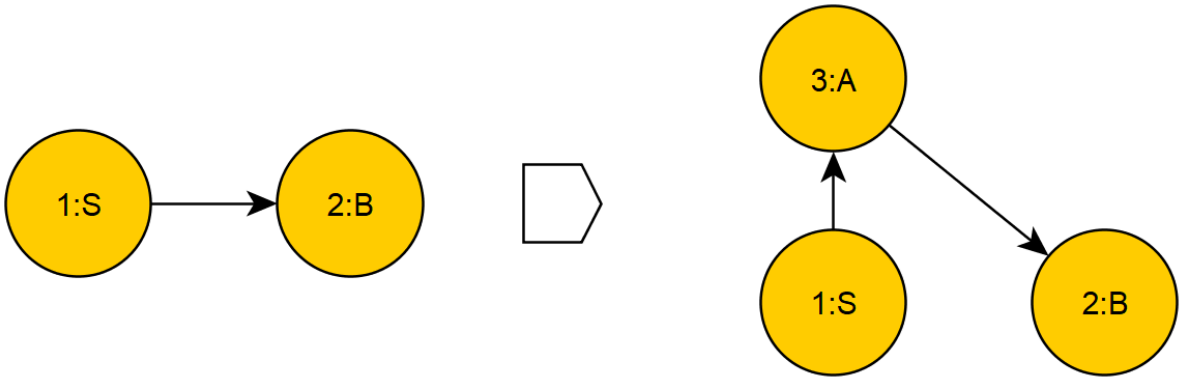
3.2. att. **Vienkāršs grafs, kas raksturo triviālu spēles līmeni**

Parasti gan reālajās spēlēs šis grafs izskatās sarežģītāk. Piemēram, J.Dormans savā darbā [12] apgalvoja, ka, pēc "Zelda" spēles līmeņu izpētes, var izšķirt spēles līmeņa misijas grafu un spēles līmeņa ģeometriju. 3.3.*att* parāda grafu sarežģītākam reālam piemēram, kur izšķir, gan uzdevuma grafu, gan arī kā tas ietekmē uzdevumus spēles ģeometrijā.



3.3.att. **Attēlojums reālas spēles līmeņa uzdevumu grafam [12]**

Grafa likumu atveidošanas un pielietojuma soļus savā darbā [12] aprakstīja J. Dormans. Likumu definīcijas piemērs redzams 3.4 att.



3.4. att. **Grafu gramatikas transformācijas likuma piemērs**

Aizvietošana notiek septiņos soļos balstoties uz [12] un grafiski tas tiek parādīts *3.5 att.*

1. Tiek identificēta un izvelēta virsotņu kopa, ko gribam aizvietot:

2. Izvelētas kopas šķautnes tiek sanumurētas, balstoties uz likuma kreiso pusi.

3. Izvelētajā kopā tiek noņemtas visas šķautnes.

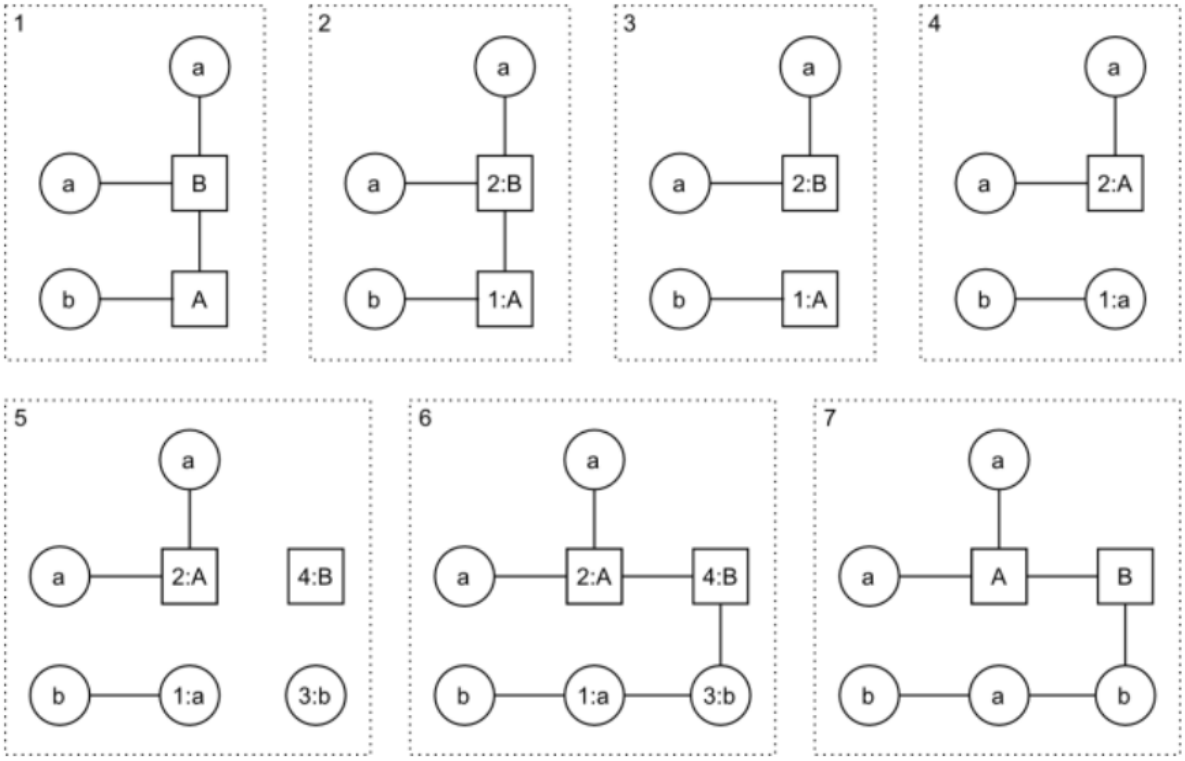
4. Izmantojot virsotņu numurus, tas tiek aizvietotas ar atbilstošam virsotnēm likuma labajā pusē.

5. Jā likuma kreisajā pusē bija papildus virsotnes, tas tiek pievienotas grafam ar attiecīgiem numuriem.

6. Izvelēta un modificēta virsotņu kopa tiek savienota ar šķautnēm, balstoties uz likuma kreiso pusi, ieskaitot virsotnes, kas savieno izvelēto grupu ar pārējo grafu.

7. Virsotņu numuri tiek noņemti.

Jāpiebilst, ka J. Dormans neizskata gadījumus, kur likums paredz virsotņu noņemšanu. Šī spēja varētu noderēt PĢS sistēmās, bet tā sarežģī grafu transformācijas, tāpēc tā tiks izskatīta eksperimentu nodaļā.



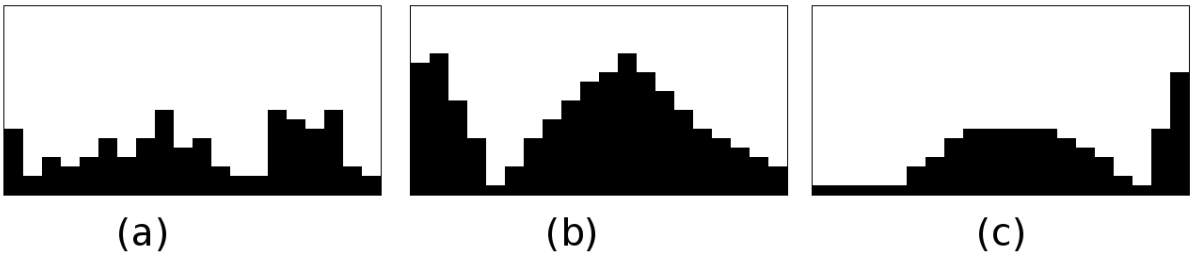
3.5. att. **Grafu gramatikas transformācijas soļi [12]**

3.2.6. Interpolācija

Interpolācija ir matemātiska metode, ar kuras palīdzību, izmantojot dotos punktus n-dimensiju telpā ņemot vērā kā n >0, mēs varam atrast papildus punktus, izmantojot interpolācijas formulu. Protams, punktu vietā var izmantot jebko, ko var izteikt ar vektoru. Visvienkāršākais piemērs ir lineāra interpolācija starp diviem dotiem punktiem divu dimensiju plaknē. Grafiski, lai atrastu visus parējos punktus, ir jānovelk taisne, kas savieno abus punktus, bet kopējā gadījumā tiek aprēķināta šis novilktas taisnes funkcija, ka parādīts formulās (3.2). Tāda formula dod iespēju jebkuram notikuma laika posmam t noteikt, kurā vietā jāatrodas punktam y, kas grib ceļot starp punktiem p_1 un p_2, kur $0 \leq t \leq 1$ ir ceļš no p_1 uz p_2.

$$y = p_1 * (t-1) + p_2 * t \tag{3.2.}$$

Interpolācija ir nepieciešama PĢS sistēmās tāpēc, ka bieži vien procesuālas pasaules mēģina pēc iespējas tuvināt dabiskām pasaulēm. Tāpēc, piemēram, jā mēs mēģinām ģenerēt zemes reljefu, ļoti nenaturāli un praktiski maz pielietojami būs ģenerēt nejaušus reljefa augstuma punktus. Daudz labāk ir ģenerēt tikai dažus augstuma punktus un noteikt visus parejos augstumus, izmantojot lineāro interpolāciju. Tādas interpolācijas rezultātā iegūsim kaut ko līdzīgu minimalistiskam kalnu zīmējumam tāpēc vēl labāk ir izmantot citas interpolācijas funkcijas, kas veidos dabiskākus reljefus. Piemēram, bikubiskas funkcijas interpolācijas rezultāts, salīdzinājumā ar iepriekš minētiem paņēmieniem, dots 3.6. att. Skaidrības pēc tiek parādīta zemas izšķirtspējas versija.



3.6. att. **Nejaušs (a), lineāri interpolēts (b) un bikubiski interpolēts reljefs (c)**

Par interpolācijas nozīmi liecina tās izmantošana arī citās jomās, kas saistītas ar lietotāja pieredzes uzlabošanu. Piemēram, visbiežāk tā tiek izmantota lietotāja saskarņu parējās interpolējot animāciju starp divām saskarnēm. Šis fakts palīdz PĢS sistēmu izstrādē, jo to jomu rezultātā tika atklātas vairākas dabīgas interpolācijas funkcijas.

3.2.7. Fraktāls

Fraktāls ir cits matemātisks termins, kas apraksta abstraktu objektu, parasti ģeometrisku. Tā galvenās īpašības ietver to, ka tas imitē dabā sastopamos objektus un var saturēt bezgalīgus, rekursīvi mazākus objektus, kuri ir vai nu simetriski līdzīgi, vai arī pilnībā vienādi ar šo lielāko objektu.

Apskatot tuvāk tā rekursīvo īpašību, varam secināt, kā fraktālu varam izteikt ar bezgalīgu koka grafu, kur katras virsotnes bērns ir kaut kādā veidā iegūts, izmantojot vecāku. Jā šādā definējumā aizvietot grafa virsotni ar grafa virsotņu kopu, tad iznāks ierobežota grafu gramatikas apakšklase ar koka struktūru un bez termināliem simboliem koka lapās, kamēr parējās koka daļās, vienmēr ir terminālas. Šī saikne dod iespēju izmantot gramatikām līdzīgas

īpašības un piemēram ģenerēt bezgalīgus līmeņus pielietojot aizvietošanu tikai tiem zariem, kurus izvelējas spēlētājs, tādejādi samazinot PĢS procesa izpildes laiku.

3.2.8. Virtuālais aģents

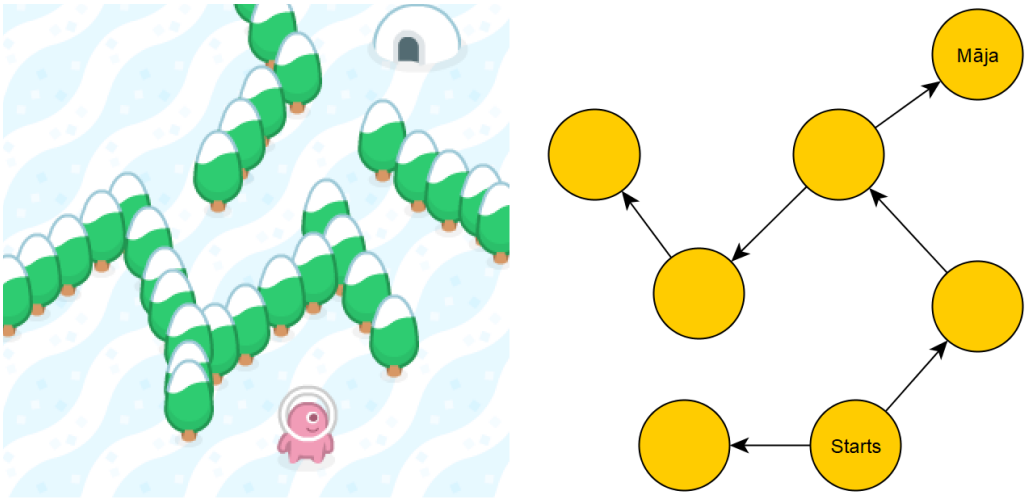
Par aģentu dēvē algoritmu, kas veic darbības virtuālajā vidē, bastoties uz noteiktiem mērķiem. PĢS gadījumā aģents var tikt izmantots, lai pārbaudītu spēles līmeņa izejamību un īpašības. Piemēram, aģenta mērķis var būt līmeņa testēšana un tas darbības rezultātā aģents pārbauda visas iespējamās darbību kombinācijas vienā spēles līmenī. Pēc šādas pārbaudes izstrādātājs var piemēram iegūt informāciju par to, vai līmenim pastāv risinājums un cik šis risinājums ir sarežģīts.

Virtuālais aģents ir ļoti vispārējs rīks, jo vienkāršākajā gadījumā tas mēģina ar rupjā spēka metodi atdarināt spēlētāja darbības katrā iespējamajā secībā. Jāpiemin, ka šāda pieeja ir diezgan neefektīva un pastāv vairāki veidi to uzlabot. Parasti tie ir saistīti ar:

a) Aģenta uzlabošanu, kura tiek pētīta mākslīgā intelekta nozarē.

b) Līmeņa genotipa vienkāršošana, kas ļauj ierobežot aģenta iespējas, ievērojami paātrinot tā darbu.

Piemēram, ja spēles mērķis ir atrast ceļu, līdz mājam, ka tas redzams *3.7.att (a)*, tad vispārējā gadījumā var izveidot aģentu, kas mēģina atrast šo ceļu nejauši ejot dažādos virzienos. No otras puses, izveidojot grafu, kas apraksta ceļošanas iespējas starp dažādiem līmeņa punktiem, ka tas parādīts *3.7.att (b)*, aģenta iespējamo virzienu skaits strauji samazinās, kas savukārt paātrina aģenta darbību.



3.7. att. **Ceļa meklēšanas spēles fenotips (a) un genotips (b)**

3.3. Paņēmieni

3.3.1. Meklēšana iespējamo variantu telpā

Viens no procesuālas ģenerācijas paņēmieniem ir izskatīt visu iespējamo variantu kopu, un sameklēt tos variantus, kuri atbilst mūsu vajadzībām. Vienkāršākajā gadījumā to var saukt par rupja spēka metodi PĢS sistēmām. Piemēram jā mēs gribam ģenerēt labu latviešu valodas vārdu, varam paņemt visus latviešu valodas burtus un kombinēt tos pēc kārtas, līdz neatrodam to kas atbilst mūsu kritērijiem. Neskatoties uz tādas metodes primitivitāti, dažos gadījumos šāda rupja spēka metode ir vienīgais veids, ka ģenerēt spēles pasauli.

Lai uzlabotu meklēšanas ātrumu, dažādas metodes tiek izmantotas, lai virzītu to pareizajā virzienā, vai nu ātri filtrējot nederīgos variantus, vai arī pārliecinoties, ka veiksmīgāki varianti tiek izskatīti pirms mazāk veiksmīgiem. Lai labāk saprastu meklēšanas problēmu, ir jādefinē daži termini, kas tiek izmantoti meklēšanas algoritmos.

Galvenais jautājums variantu meklēšanā ir "vai apskatāmais risinājums atbilst uzdotiem kritērijiem?". Šo jautājumu var arī paplašināt un vispārināt pārveidojot to par "Cik lielā mērā (vai cik tuvu) apskatāmais risinājums atbilst uzdotiem kritērijiem?". Tagad ir iespēja ne tikai izvērtēt, vai variants der, bet arī pateikt, vai viens variants ir labāks par otru. Formalizējot šo jautājumu, definēsim Atbilstības funkciju - funkcijas, kura izmantojot iedoto variantu, novērtē tā atbilstību izvirzītiem kritērijiem.

Rēķināt atbilstības funkciju un veidot modeļus gatavam spēles pasaulēm vai kartēm ir diezgan sarežģīti un neefektīvi. Tāpēc rezultāta formātam ir jādefinē vienkārša, vislabāk matemātiska, reprezentācija, jeb genotips. Sistēmā jābūt iespējai pārveidot šo reprezentāciju par spēlējamu spēles līmeni, jeb fenotipu. Šādā veidā varam droši teikt, ka, atrodot genotipu, kuram atbilstības funkcija izdod augstu rezultātu, šī genotipa atbilstošam fenotipam būs augsta atbilstība mūsu nosacījumiem.

Genotipa izvēlei ir ļoti svarīga nozīme PĢS meklēšanā, jo no tā ir atkarīga sekmīga līmeņa atrašana. Vēl genotipā izšķir tuvumu fenotipam, kas nosaka, cik viegli ir genotipu par fenotipu un cik stipri var paredzēt fenotipa īpašības, izmantojot genotipu. Piemēram, ir piedāvātas dažas fenotipa reprezentācijas to tuvuma secībā:

1. Vistuvākajā gadījumā genotips precīzi atbilst fenotipam un ir vienkārši spēles līmenis. Šajā gadījumā saglabājas visas fenotipa īpašības, genotips parasti ir ļoti tāls no abstrakta matemātiska modeļa, kas padara atbilstības funkcijas aprēķināšanu un citas algoritmiskas novērtēšanas un transformāciju darbības ļoti sarežģītas.

2. Nedaudz vienkāršojot fenotipu, mēs varam definēt objektu zonas. Piemēram, varam pateikt, ka tajā aptuvenajā vietā būs tāds ienaidnieku skaits un tāds monētu skaits. Šādas reprezentācijas genotips varētu nesaglabāt visas fenotipa īpašības, bet tas ir tuvāks modelim, kura īpašības varam izrēķināt, kas atvieglo vairāku algoritmu darbu.

3. Genotips ir vektors, kas apraksta fenotipa īpašības, piemēram, cik ienaidnieku un cik bonusu ir šajā līmenī kopumā vai arī kāds šķērslis mūs gaida līmeņa beigās. Ar šādu genotipu ir ļoti viegli strādāt, jo tas sastāv tikai no pietiekami matemātiskiem parametriem, taču šādā veidā mēs jau zaudējam iespējas kontrolēt vairākas līmeņa īpašības, kas mums varētu būt svarīgas. (Piemēram, ienaidnieku pozīcijas)

4. Visekstrēmākais genotipa atveidošanas veids ir izmantot sēklu, ko padodam ģeneratoram. Visbiežāk sēklai nav nekādu saistību ar līmeni īpašību ziņā, tāpēc tas nedod nekādu kontroli par fenotipu. No otras puses sēkla netiek vienkārši padota NSĢ bet gan izmantota līdzīgi, ka parametru vektors un varam vismaz noteikt ka mazas izmaiņas seklā dos mazas izmaiņas līmeni, kas dot nelielu kontroli meklēšanas algoritmos.

Vienu labu genotipa piemēru, balstoties uz izstrādātājas prezentāciju [14] izmanto spore, kur radības genotips ir skelets un fenotips ir šī būtne, ieskaitot ādu un muskuļus.

3.3.2. Evolucionārie algoritmi

Togelius J. un citi, savā darbā [13], piedāvāja izmantot evolucionāros algoritmus PĢS meklēšanā. Evolucionārie algoritmi jau tiek izmantoti mākslīgajā intelektā, lai atrastu mākslīga intelekta aģentus, kuru darbības ir, vai nu visvairāk pietuvinātas cilvēka darbībām, vai arī ļauj vissekmīgāk iziet spēles līmeni, izmantojot tikai spēlētajam pieejamās rīcības. Mākslīgajā intelektā pastāv vairākas līdzības ar PĢS, jo no vienas puses tiek veidots spēles līmenis un no otras puses tiek veidots aģents, kas mēģina šo līmeni iziet. Apskatot spēles no tādiem žanriem kā torņu aizsardzība (Tower defence), var izsecināt, ka dažreiz lomas mainās un aģents varētu būvēt līmeni, kas neļauj šo līmeni iziet ienaidniekiem, kuri savukārt šīs spēles ietvaros ir līmeņa sastāvdaļa. Viena svarīga līdzība ir tā, ka mākslīgiem intelektiem evolucionāros algoritmos pastāv atbilstības funkcija, kas ļauj pielietot šos algoritmus PĢS sistēmās.

Mākslīga intelekta nozarē pastāv diezgan daudzi evolucionārie algoritmi variantu atlasīšanai pēc atbilstības funkcijas kurus varam izmantot arī PĢS. Vienkāršākais no tiem

1. Izvēlas X nejaušus genotipus.

2. Katram genotipam izvelēto kopā aprēķina atbilstības funkcijas rezultātu.

3. Tiek izmesta daļa no genotipiem ar vismazāko atbilstības vērtību. Daļas izmērs atkarīgs no algoritma atlases parametriem, ko nosaka programmētājs.

4. Izmesta daļa tiek aizvietota ar visatbilstošāko (ar lielāko atbilstības funkcijas vērtību) genotipu mutācijām. Mutācijas šeit ir genotipi ar nelielam novirzēm no oriģināla genotipa. Mutētie genotipi tiek pievienoti izvēlētajai kopai, tādā veida atstājot izvēles kopā X elementus.

5. Meklēšana turpinās no 2. soļa, līdz netiek atrasts genotips ar vēlamo atbilstību, vai arī izsmelts meklēšanas soļu skaits.

Šeit lielu nozīmi spēlē mutēšanas algoritms, ja to pareizi implementē, tad evolūcijas laikā genotipu atbilstība nemitīgi augs, taču šis uzdevums nav tik triviāls. Piemēram, mēs varam meklēt līmeņus ar noteiktu ienaidnieku skaitu, un tādā gadījumā mutācijas palielina vai samazina genotipa ienaidnieku skaitu par 1. Izmantojot pareizus parametrus, ar ļoti lielu varbūtību atradīsim vēlamo līmeni. Bet šajā gadījumā genotipā ir tieši norādīts ienaidnieku

skaits. Pretējā variantā, kur genotips ir sēkla, ir ļoti sarežģīti un dažreiz pat neiespējami paredzēt kādas mutācijas jāveic, lai mutācija būtu līdzīga oriģinālam atbilstības ziņā.

3.3.3. Labirintu ģeneratori

Pietiekami bieži spēle sastāv no sākuma stāvokļa, gala mērķa stāvokļa, un vairākiem ceļiem kā šo stāvokli sasniegt. Piemēram, tas varētu būt līmenis, kur spēlētājs sāk vienā pazemes istabā un viņam jāiet cauri vairākām citam istabām, lai sasniegtu istabu ar izeju virspusē. Genotips šādam spēles līmeņa fenotipam varētu būt labirints. Labirints ir diezgan veiksmīga izvēle genotipam, pateicoties tam, ka ir atklāti un izpētīti vairāki labirinta izveidošanas algoritmi. Tā piemēram vienkāršāko no tiem (Backtracking) var aprakstīt dažos soļos:

1. Paņem sākuma virsotni.

2. Izvēlas paņemto virstoni. Jā tā nav apmeklēto kopā, pievieno to šai kopai.

3. Jā izvēlētai virsotnei nav neapmeklētu kaimiņu, turpina no 6. soļa.

4. Izvelēto virsotni ievieto stekā.

5. Izveido tuneli starp izvelēto virsotni un nejaušo kaimiņu. Paņem šo kaimiņu un pariet uz 2. soli, paņemot šo kaimiņu.

6. Ja steks nav tukšs, Izņem virsotni no tā un turpina, no 2. soļa paņemot izņemto virsotni.

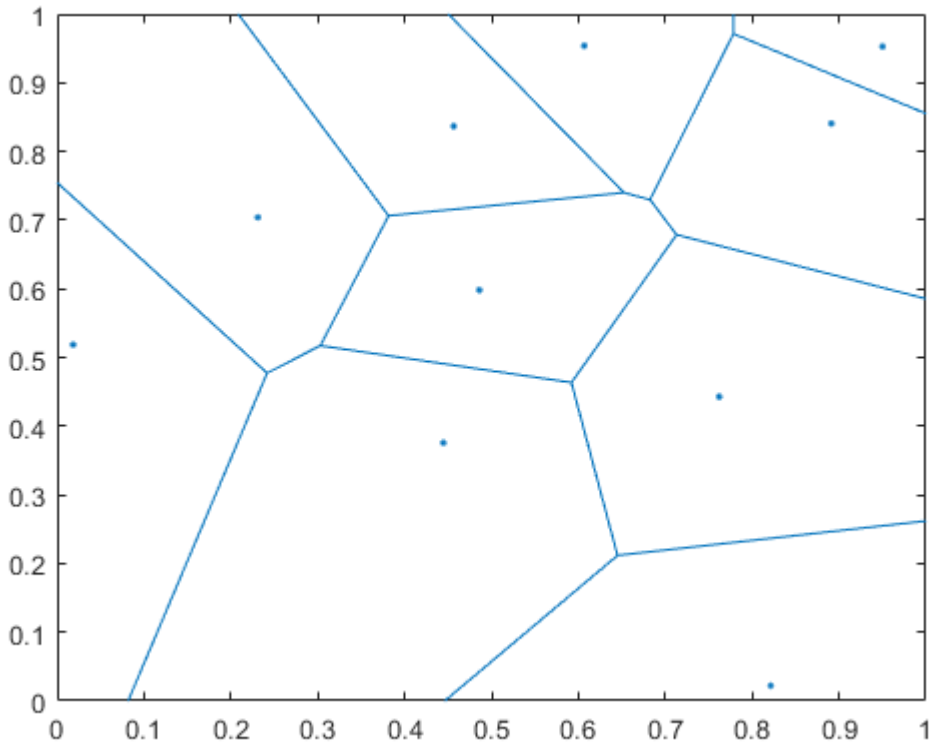
Vienkāršākajā variantā algoritms tiek pielietots divu dimensiju kvadrāta rūtiņa laukumam, taču pārsvarā labirinta veidošanas algoritmi strādā jebkurā dimensiju skaitā un, izmantojot jebkuru datu tipu, kuram varam definēt kaimiņus, kas atbilst labirintu īpašībām. Piemēram, platformēra spēlē kaimiņi ir visas rūtiņas, uz kurām varam tikt no esošas rūtiņas.

Viens vispārējs gadījums, kur varētu izmantot labirintu, ir grafs. Grafam ir pietiekam precīzi definēti kaimiņi un ka jau esam redzējuši agrāk darbā, grafu diezgan sekmīgi var izmantot spēles uzdevumu secības definēšanai.

3.3.4. Sadales algoritmi un Voroni Diagramma

PĢS spēlēs diezgan bieži tiek ģenerēti pasaules apgabali, kuri varētu nozīmēt pilsētas, biomas vai cita veida teritorijas. Viens no efektīvākajiem veidiem to paveikt ir izvietot kārtā vairākus punktus, kuri apzīmē šo teritoriju centrus, un sagrupēt visus pasaules punktus pēc tā, kuram teritorijas centram tie vistuvāk atrodas. Tādā veidā tiek izveidota Voroni Diagramma, kuras piemēru var redzēt *3.8. att.*

Voroni diagramma tiek izmantota vairākās nozarēs un, tas īpašības ir diezgan detalizēti izpētītas, kas ļauj izmantot šo diagrammu vairākiem mērķiem. Piemēram, izmantojot Delaunay triangulāciju, ir iespējams aprēķināt konkrētu vietu, kur var ievietot jaunu teritorijas punktu ar vislielāko iespējamo teritoriju pēc visu teritoriju atkārtotās izskaitļošanas. Tāpat pastāv arī algoritmi, kas ļauj pietiekami ātri konstruēt šīs diagrammas.



3.8.att. **Voroni diagrammas piemērs [21]**

3.3.5. Izklāšana

Var izmantot gadījumos, kad spēles karti var sadalīt vairākos reģionos ar identisku formu, kur šos gabalus noteiktajā pakāpē var mainīt vietām neizjaucot kartes spēlējamību. Izklāšana nozīmē, ka mums pastāv rūtiņa, kas ir kartes elements. Šai rūtiņai ir vairākas

variācijas un viss, kas vajadzīgs kartes ģenerēšanai ir salikt šis rūtiņas pareizajā kartībā. Atkarībā no spēles rūtiņu kartībai var nebūt nozīme un karte tiek izveidota, pilnīgi nejauši saliekot šīs rūtiņas. Dažreiz rūtiņām ir definēti nosacījumi, tādi, ka, piemēram, rūtiņai A no labās puses var būt tikai rūtiņa B vai C.

Šī ir, iespējams, vieglākā PĢS metode, ar kuru ieteicams sākt jebkuram, kas grib iepazīties ar procesuālo ģenerēšanu. Spelunky izmanto šo pieeju apvienojot to ar vienkāršam aizvietošanas gramatikām. Par šis metodes vienkāršību liecina tas, ka galda spēle Catan izmanto to rūtiņu kāršu maisīšanas un izklāstīšanas veidā.

3.3.6. Plānošana

Viens no svarīgajiem satura veidiem spēlēs ir stāsts. Bez stāsta parasti ir diezgan sarežģīti radīt nepieciešamo atmosfēru un sajūtu par spēles realitāti. Līdzīgi spēles noteikumu ģenerēšanai, kura tika aprakstīta agrāk darbā, stāsta ģenerēšanas algoritmi šobrīd ir pietiekami līdzīgi drīzāk notikumu kombinēšanai pareizajā secībā. Lai šos notikumus sakārtotu šajā pareizajā secība, varam izmantot kādu no plānošanas algoritmiem, kuri tiek plaši izmantoti mākslīga intelekta sistēmās. Šāds algoritms tā vienkāršākajā gadījumā izmanto rupjā spēka metodi un izveido grafu ar visiem iespējamiem notikumiem, kur katrs notikums modificē aģenta stāvokli un aģenta mērķis ir savākt stāvokļu modifikācijas līdz tā stāvoklis sakrīt ar vēlamo.

Stāsta ģenerēšanas gadījumā mums ir vajadzīgi divi stāvokļi

1. Mērķa stāvoklis – Piemēram jā mēs gribam ģenerēt stāstu, kur spēlētājs kļūst par varoni, tad ir jādefinē stāvoklis “spēlētājs ir varonis”. Īstajā spēlē tas varētu nozīmēt, ka spēlētāja varoņdarbu skaits ir lielāks par 10 un viņa ļauno darbu skaits ir mazāks par 3.

2. Sākuma stāvoklis – Izmantojot tikko minēto piemēru, varam definēt stāvokli, no kūra stāsts tiks iesākts. Ļoti vienkāršajā gadījumā šis stāvoklis varētu būt spēlētājs bez varoņdarbiem un ļauniem darbiem, bet kuram ir palicis dzīvot 11 dienas, un viņš var paveikt tikai vienu varoņdarbu vai ļauno darbu dienā.

Augstāk aprakstītājs piemērs ir ļoti vienkāršs stāsta ģenerēšanai, izmantojot plānošanu, tā soļus var viegli aprakstīt sekojoši:

1. Izveidojam grafu ar sākuma virsotni, ka sākuma stāvokli.

2. No virsotnes izveidojam trīs šķautnes, tas attiecīgi nozīmē varoņdarbu, ļauno darbu un nekā nedarīšanu. Šis solis tiek rekursīvi pielietots ar maksimālo dziļumu 10.

3. Katra virsotne ir parametru "atlikušo dienu skaits", "varoņdarbu skaits" un "ļauno darbu skaits" trijnieks.

Uz šo brīdi tika izveidots grafs, kurā ir visas iespējamas šo trijnieku variācijas, ir izpildītas un jā tas nav pretrunā ar nosacījumiem, tad dažas no iegūtam virsotnēm atbilst mērķa stāvoklim, un ceļš uz šo stāvokli izveidos stāstu, kas sastāvēs no notikumiem "Spēlētājs paveica ļauno darbu", "Spēlētājs paveica varoņdarbu" u.t.t. Protams šis ir ļoti triviāls piemērs un to uzlabot var definējot vairākus ļauno un labo darbu aprakstus, bet plānošanas princips būtībā paliek tāds pats citiem paņēmieniem.

Minētajā gadījumā esam aprakstījuši stāstu, kur jau ir definētas varoņa iespējas, taču lielākam stāstu daudzumam ir iespējams ģenerēt arī notikumus.

3.4.Secinājumi

Starp apskatītajiem rīkiem un paņēmieniem tika izskatīt tikai vienkāršākie pēc autora domām noderīgākie. Neskatoties uz to, šo rīku skaits ir pietiekami liels un daudzi no tiem var tikt izmantoti dažāda tipa satura ģenerēšanai.

Īstu secinājumu no šiem rīkiem gan varēs veidot, apskatot to pielietojumu konkrētās spēlēs.

4. PIELIETOJUMI

4.1. Ievads

Iepriekšējās nodaļās tika izskatītas esošas spēles un sistēmas, kas ģenerē to noteikumus. Tāpat tika izskatīti rīki un paņēmieni, kas tiek izmantoti, lai izveidotu tādas PĢS sistēmas. Tas nodaļas deva īsu ieskatu, kur konkrēti šīs sistēmas var tikt izmantotas un kas vispārēji varētu tikt paveikts, tas izmantojot, bet netika minēti vispārēji gadījumi, kur tie tiek izmantoti un detalizēti apskatīts, kas var tikt paveikts, tas izmantojot. Par pēdējo vairāk tiks stāstīts nākamajā nodaļā, kamēr šis nodaļas mērķis ir piedāvāt PĢS vispārējas izmantošanas kategorijas.

4.2. Pielietojumu sfēras

4.2.1. Līmeņu variācijas

Visbiežāk minēta PĢS priekšrocība ir, ka pasaules var ģenerēt no jauna katru reizi, kad spēlētājs sāk jaunu spēli. Formāli var pateikt, kā PĢS sistēma var ģenerēt līmeņa variācijas. Tas nozīmē to, ka līmeņu atkārtotā iziešana nebūs tīk garlaicīga. Spēlēs, kurās stāvokļa saglabāšana iespējama tas ir īpaši svarīgi. Nejaušas līmeņu variācijas ir arī ļoti svarīgas izstrādātajiem, kuri spēles taisa mazās komandās, jo ir ļoti grūti novērtēt līmeni, ko pats esi taisījis. PĢS gadījumā līmeņus daļēji taisīja dators, kas daļēji risina šo problēmu.

4.2.2. Līmeņu neierobežots izmērs

Cita svarīga PĢS īpašība ir tāda, ka, izmantojot pietiekami ātrus algoritmus, dators var ģenerēt lielākus līmeņus, nekā spēlētāji varētu izstaigāt. Tas veido līmeņa neierobežotības ilūziju. Pretēji tam cilvēks parasti nevarētu izveidot tāda izmēra pasaules, darot to ar rokām.

4.2.3. Palīdzība cilvēkam

Ka jau tika minēts, līmeņa ģenerēšanai nav obligāti jābūt 100% datora paveiktai un cilvēks varētu strādāt kopā ar datoru. Piemēram, dators var ģenerēt labirintu, un cilvēks var salikt objektus, kas ierobežos spēlētāja kustību, ļaujot viņam ceļot tikai labirinta ietvaros. Tas pats var notikt otrādi: Cilvēks definē ceļu, kurā spēlētajam būtu jāstaigā, un dators saliek dekorācijas un ienaidniekus gar šo ceļu.

Nedaudz savādāka pieeja ir definēt likumus, kurus līmenim jāievēro, un izveidot algoritmu, kurš pārliecinās par šo likumu ievērošanu. Tādi likumi varētu piemēram būt noteikta līmeņa sarežģītība, vai spēlētāja iepazīšana ar kādu konkrētu mehāniku. Izmantojot šo algoritmu, cilvēks var strādāt līmeņu redaktorā un jebkurā brīdī viņš var uzspiest pogu, procesuāli uzģenerēs un piedāvās lietotajam līmeni, kas pietuvināts jau izveidotajam, bet ar modifikācijām kuras nodrošina uzdoto likumu ievērošanu.

4.2.4. Dinamiskā līmeņu ģenerēšana

Kā jau tika minēts, agrāk, kad datoru atmiņa bija ļoti ierobežota, spēlēs PĢS sistēmas tika izmantotas, lai ģenerētu pasaules daļas, kuras mums šobrīd vajadzīgas. Mūsdienās pat ja operatīvas atmiņas nepietiek visas līmeņa glabāšanai, ir iespēja ielādēt līmeņa datus no cietā diska. Diemžēl datu apmaiņa starp šiem diviem mēdijiem nav tik liela, savukārt datu apmaiņa starp procesoru un operatīvo atmiņu ir daudz, straujāka, kas ļauj mums izmantot deterministiksus PĢS algoritmus un aprēķināt līmeņa daļas dinamiski, kas pie pareizi izveidotiem algoritmiem dod krietni lielāku ātrumu.

4.2.5. Cits skatupunkts

Īpaši atjautības spēlēs lielu nozīmi spēle izaicinājums nestandarta pieejai šī līmeņa risināšanai un, ņemot vērā to, ka datora līmeņu būvēšanas process stipri atšķiras no cilvēka, tas dod iespēju veidot iepriekš neredzētus līmeņus. Labs apliecinājums tam ir Yavalath galda spēle, kur bieži vien rodas doma par to, ka papildus noteikumam nav nekādas jēgas, bet tas pievieno spēlei papildus sarežģītības dimensiju, par ko liecina spēles popularitāte.

4.2.6. Izmantošana ārpus spēlēm

Nenovirzoties no spēļu tēmas vēl joprojām, var definēt dažas PĢS iespējas ārpus spēļu nozares. Tas saistīts ar to, ka spēles bieži ir noteiktā pakāpē modelētas reālas situācijas un jo īpaši reālas spēles. Piemēram, ja spēlē izmantojot konkrētus noteikumus tiek ģenerēta pilsēta vai mazākas platības arhitektūras izvietojums, tad ir vispārējā gadījumā ir iespējas izmantot līdzīgus algoritmus īstas arhitektūras ģenerēšanai.

Šajā gadījumā tiek izmantota PĢS īpašība, kura paredz sekošanu noteiktam likumu un sakarību kopumam un, definējot šādas sakarības atbilstoši reālas pasaules likumiem, ir iespējams pielāgot spēlēm domātos algoritmus tādā veidā kā tie palīdzēs arī ārpus tam spēlēm. Tas, protams, attiecas gan uz spēles ģeneratoru pārnešanu reālajā pasaulē, gan arī šo ģeneratoru pārnešanu uz citām spēlēm.

4.3. Secinājumi

Īsi tika apskatīti PĢS pielietojumi un izsecināts, ka tie ir diezgan universāli savā darbībā gan spēļu ietvaros, gan arī ārpus tam. To vispārīgums attiecas arī uz to sadarbības pakāpi ar cilvēku, jo PĢS nenozīmē saturu, kas pilnībā izveidots datora autonomās darbības rezultātā, bet gan arī gadījumus, kur datoram ir palīga loma tradicionālajā līmeņu izveides procesā. Šāda veida palīdzība var noderēt arī ārpus spēlēm, kur līmeņa vieta, piemēram, tiek izveidots arhitektūras plāns, kuram jāseko konkrētiem noteikumiem.

5. EKSPERIMENTI

Eksperimenta uzdevums ir palīdzēt izpildīt visa darba uzdevumu izpētīt un atvieglot PĢS sistēmu izveidi spēlēm. Ņemot vērā nosacījumus, ka darba eksperimentiem jābūt atkārtojamiem un faktu, ka viens no labākajiem veidiem ir atvieglot jebkādu uzdevumu ir pievest vispārējus un uzskatāmus piemērus, tad eksperimentā tiks aprakstītas dažas vispārējas spēles un PĢS sistēmu ievešana šajās spēlēs.

Ņemot vērā iepriekšējas nodaļas, tādas spēles, kurām ir vairāki eksistējoši piemēri ar PĢS sistēmām netiks apskatītas, jo tas neveicina darba uzdevumu sakarā ar to, ka efektīva PĢS sistēmu ieviešana šajās spēlēs jau ir pietiekami izpētīta. (Tas attiecas piemēram uz Roguelike spēlēm)

5.2. Vispārēja Izdzīvošanas spēle

5.1.1. Spēles apraksts

Viens spēles žanrs ar pietiekami specifisku, bet bieži izmantojamu mehāniku ir izdzīvošanas un šaušanas spēle, kuras noteikumus var vispārēji aprakstīt sekojoši:

1. Spēlētāja avatārs atrodas līmeņa centrā un nekustās. Vienīga darbība ir šaut lodes jebkurā virzienā 360 grādu leņķī.

2. No dažādiem punktiem uz spēlētāju (kartes centru) nāk ienaidnieki.

3. Lodes ietekmē ienaidniekus definētā veidā, piemēram, iznīcina.

4. Spēles mērķis ir neļaut ienaidniekiem tikt uz spēles centru.

Spēles elementu apraksts:

Spēles laukums: Ņemot vēra to, ka visas kustības notiek no vai uz centru, paliek skaidrs, ka laukuma koordinātu aprakstam ir izdevīgāk, izmanto Eiklīda telpu.

Spēlētāja avatārs: Var neieverot, jo tā vienīgas īpašības ir neļaut ienaidniekiem tikt uz centru un no centra lodes sāk savu ceļu. Tātad, vienīgais, ietekmējoša īpašība ir spēlētāja atrašanas vieta, kas ir centrs (0, 0).

Lodes: Lodes uzsāk savu ceļu no kartes centra (0,0) un turpina savu ceļu, nemainot savu leņķi Eiklīda koordinātu sistēmā. Tātad vektorā (θ, r), θ paliek konstants un r nemitīgi palielinās.

Ienaidnieki: Ienaidnieki uzsāk savu ceļu kāda punktā (θ, r) un kustās tieši uz centru, kas Eiklīda koordinātu sistēmā nozīmē nemainīgu leņķi. Atšķirība no lodēm, to rādiuss nemitīgi samazinās, līdz tas sasniedz 0, kas nozīmē, ka tas ir centrā.

5.1.2. Spēles pamatjēdzieni un īpašības

Spēles galvenais notikums, ir lodes saskaršanas ar ienaidnieku. Vienkāršības pēc sauksim šo notikumu par sadursmi.

Sadursmes rezultātā var veikt vairākas darbības, bet šoreiz pieņemsim, ka, saduroties ienaidniekam un lodei, tie abi tiek iznicināti.

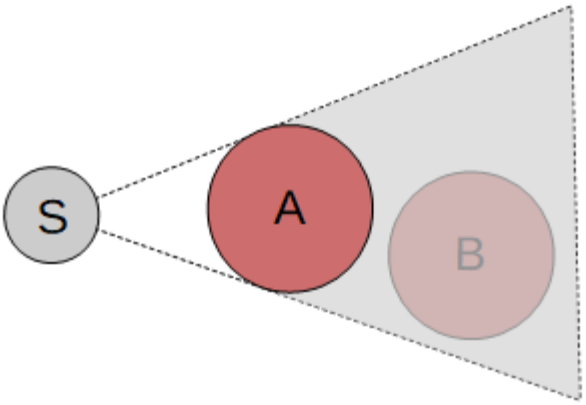
Šoreiz PĢS sistēmas mērķis ir uzlabot spēles kvalitāti, kas pēc iepriekšējas definīcijas citu īpašību starpā ietver sevī dažādus izskaitļojamus iznākumus un regulējamu spēles sarežģītību. Tā kā pēc spēles apraksta, vienīgā spēlētāja izvēle ir šaut izvēlētajā leņķī, tad iznākumus var kontrolēt, izvēloties nākamo ienaidnieku, kuru gribam trāpīt. Tas nozīmē, ka labs iznākums spēlē rodas no pareizajam ienaidnieku izvēles secībām, kamēr slikts iznākums rodas tad, ja ienaidnieku secība izvelēta nepareizi. Savukārt sarežģītība tiek kontrolēta, nosakot pareizas izvēles sarežģītību un izvēlētas secības garumu.

No tā izriet, ka svarīgi ir kontrolēt 2 īpašības:

1.) Kurus ienaidniekus var izvēlēties.

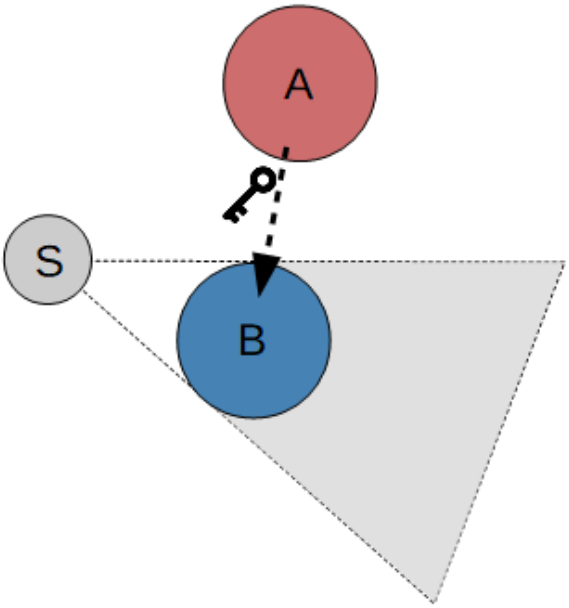
2.) Kādā secībā ir visizdevīgāk izvēlēties šos ienaidniekus.

Pirmo īpašību ir viegli kontrolēt piemēram aizsedzot vienu ienaidnieku kopu, izmantojot citu ienaidnieku kopu. Piemēram, *5.1.att* riezdams, ka, ņemot vērā spēlētāja pozīciju S, neiznicinot ienaidnieku A, nevar iznicināt ienaidnieku B un jebkuru citu ienaidnieku, kas atrodas pelēkajā laukumā.



5.1. att. **Ienaidnieks A aizsedz ienaidnieku B**

Otras īpašības kontrolēšanai izmantosim paņēmienu, kuru Joris Dormans piedāvāja savā prezentācijā [20] - Slēdži un atslēgas. Mūsu gadījumā atslēga ir ienaidnieks, kuru iznicinot, viens slēdzis pārvēršas par ienaidnieku. *5.2.att* ilustrēts slēdža un ienaidnieka izvietojums. Tajā ienaidnieks B (slēdzis) skaitās neiznicināms, līdz tiks iznicināts ienaidnieks A. Tas nozīmē, ka ienaidnieks A noteikti tiks iznicināts pirms B un jebkura ienaidnieka pelēkajā zonā.



5.2. att. **Ienaidnieks B ir neiznicināms, kamēr A nav iznicināts**

Papildus tam tiks izmantoti bonusa ienaidnieki. Sadursme ar bonusa ienaidnieku atvieglo visu līmeni, izmantojot noteiktas spējas. Šobrīd nav svarīgi, kādās spējas dod bonusa ienaidnieks, bet izmantosim to spēles mērķa definēšanai:

a) Spēles mērķis ir iznicināt visus ienaidniekus.

b) Spēles efektīvākā stratēģija ir iznicināt to ienaidnieku, kas pietuvina spēlētāju pie vistuvākā bonusa ienaidnieka.

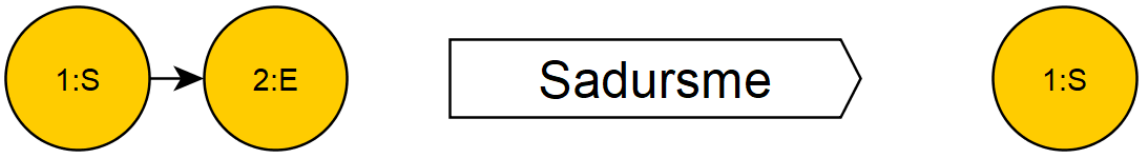
Punkts "b" ir tieši saistīts ar spēles "dažādiem izskaitļojamam nobeigumiem", ņemot vērā to, ka izskaitļot šos nobeigumus būs aizvien sarežģītāk, varam secināt, ka šis noteikums ļaus mums regulēt spēles sarežģītību.

5.1.3. Genotipa definēšana

Spēles līmeņa genotipam tiks izmantots vienvirziena grafs un grafa gramatikas. Lai definētu grafa gramatiku, noskaidrosim, kādā veidā grafs atveidos spēles līmeņa īpašības:

a) Grafā būs viena spēlētāja virsotne S. Pārejas virsotnes reprezentēs ienaidniekus un tiks apzīmētas ar E (un L situācijās, kad ienaidnieks ir slēdzis).

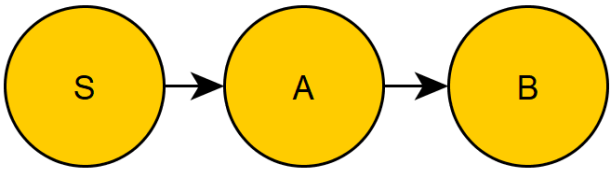
b) Ienaidnieka iznicināšana ir grafa transformācija, kur tiek noņemta viena E virsotne. *5.3.att* ir parādīta vienkārša sadursmes transformācija, kurā jāņem vērā, ka visas virsotnes "2:E" šķautnes pariet uz virsotni "S".



5.3.att **Transformācija vienkāršam sadursmes gadījumam**

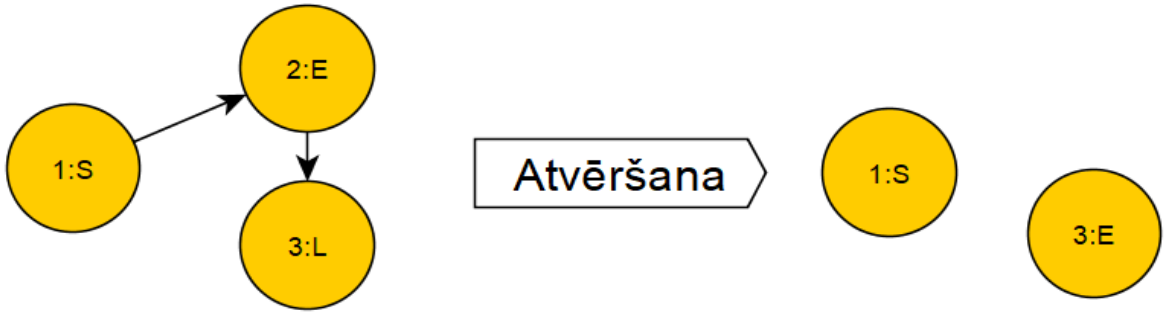
c) Kā redzams *5.3.att,* sadursme var notikt tikai ar ienaidniekiem, kuriem virsotne grafā tiek savienoti ar spēlētāja virsotni S.

d) Ņemot vērā īpašību c, vienas virsotnes aizsegšana var tikt apzīmēta, noliekot tai priekšā citu virsotni. Tā piemēram *5.1.att* redzamā aizsegšana grafā izskatīsies, tā kā tas parādīts *5.4. att*.



5.4. att. **Virsotne B nav sasniedzama**

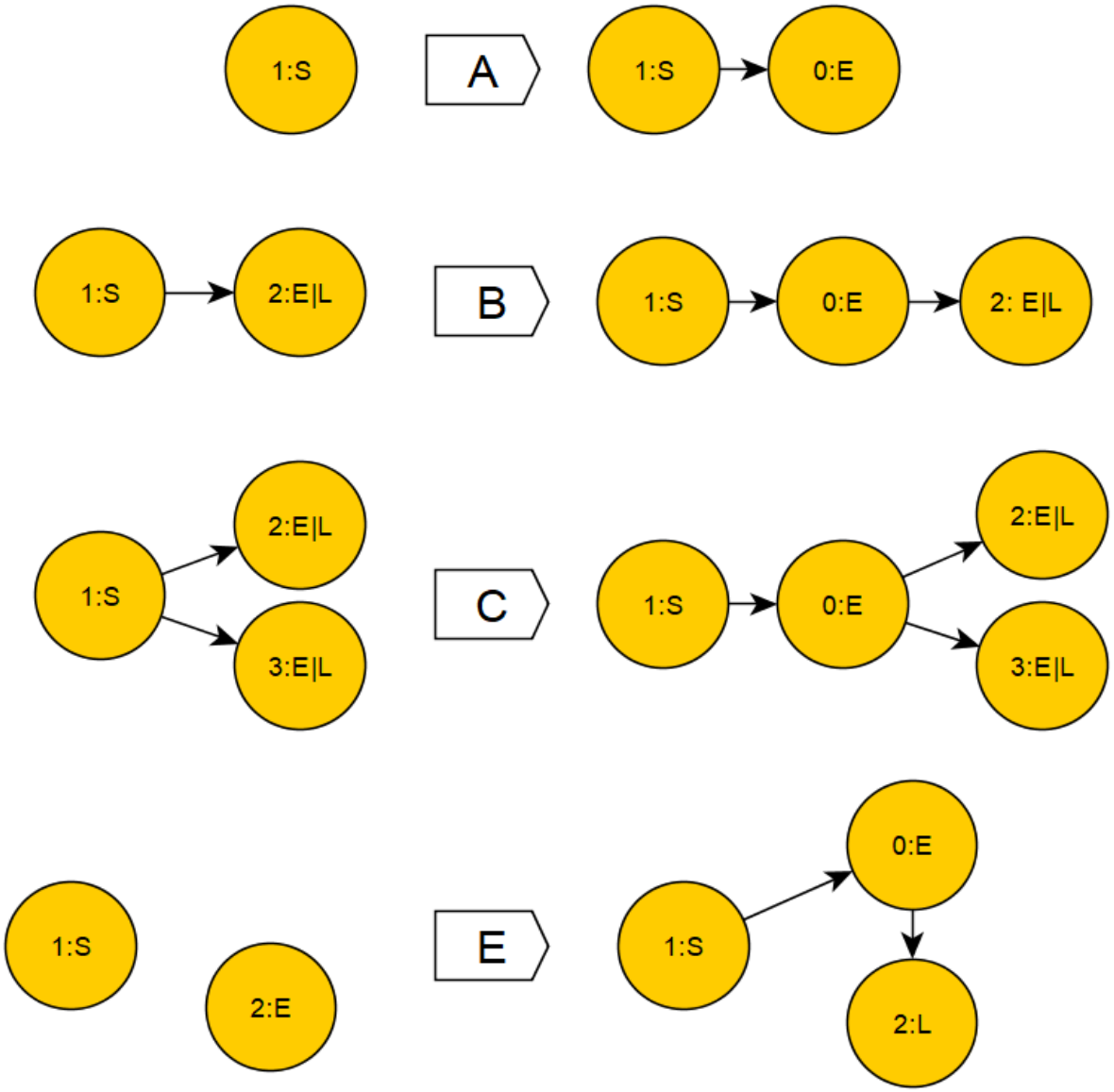
e) Iepriekš minētā atslēga var tikt īstenota, izmantojot transformāciju, kura ir redzama *5.5. att.* Šajā attēlā ar L apzīmē slēdzi, kuru nevar iznicināt ar lodēm, bet transformācijas laikā tā pārtop parastajā ienaidniekā, rādot "atvēršanas" efektu.



5.5.att. **Atvēršanas efekts**

f) Viens ienaidnieks var aizsegt vairākus citus ienaidniekus, tāpēc grafā viena virsotne var iziet uz vairākām citam virsotnēm.

Ņemot vērā visas šīs īpašības, tika izveidotas grafa transformācijas, kuru rezultāts vienmēr ir spēlējams grafs. Grafs sākas ar starta virsotni S, un tā transformācijas attēlotas *5.6. att*. "E|L" nozīmē, ka var tikt izmantots gan E gan L un labajā pusē tas paliek nemainīgs.



5.5.att. **Grafa transformācijas līmeņa ģenerēšanai**

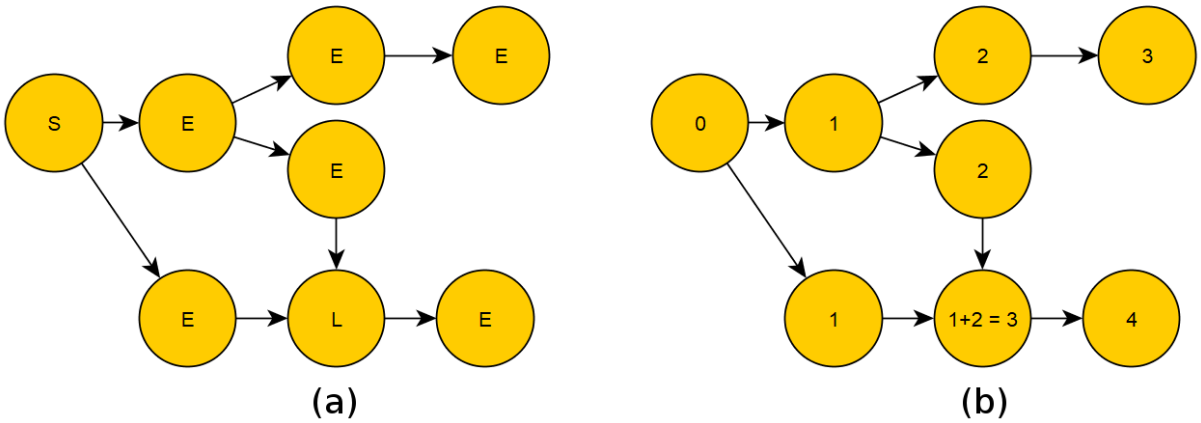
3 pirmās transformācijas (A, B, C) tika iegūtas otrādi apgriežot sadursmes transformāciju parastajā gadījumā ienaidniekiem, kuri aizsedz 0 (A), vienu (B), vai divus (C) citus ienaidniekus. Tas pats iespējams arī vairākiem aizsedzamiem ienaidniekiem, bet šobrīd apskatīsim tikai vienkāršus gadījumus. Transformācija D ir iegūta, otrādi apgriežot atvēršanas transformāciju. Tātad visas transformācijas ir veiktas otrādi apgriežot sadursmes efektus, kas pierāda to, ka pielietojot attiecīgus sadursmes efektus atgriezeniskā secībā, vienmēr ir iespējams iziet šo līmeni.

5.1.4. Līmeņa sarežģītība

Iepriekš tika minēts, ka līmeņa kvalitāti nosaka tas, cik līdzsvarota ir tā sarežģītība, un sarežģītības kontrolēšanai tika piedāvāti bonusa ienaidnieki, kuri pazemināsturamāko līmeņa sarežģītību. Tajā pašā laikā noteicam, ka visizdevīgākais ceļš ir vienmēr tiekties pēc tuvākā bonusa.

Tāpēc lai kontrolētu grafa sarežģītību:

1. Uzbūvējam nejaušu grafu (*5.6.att a*) un būvēšanas laikā iezīmējam katrai virsotnei tas dziļumu (*5.6.att b*). Gadījumos, kur vienā virsotnē ieiet vairākas citas (slēdža gadījums), to dziļumi summējas.



5.6.att. **Līmeņa grafa piemērs (a) un tā katras virsotnes dziļums**

2. Tiek izvēlēta virsotne ar pietiekami mazu dziļumu, un noņemtas visas virsotnes starp S un izvēlēto virsotni izmantojot sadursmes un atvēršanas transformācijas. Izvelētā virsotne pārtop par bonusa virsotni un visas turpmākās transformācijas tiek veiktas ar sarežģītības atlaidi.

3. Tiek nobloķētas visas virsotnes, kuru dziļumi ir mazāki par tikko izvēlētas virsotnes dziļumu. Nobloķētas virsotnes nevar izvēlēties par jaunajiem bonusa punktiem, bet uz tam var veikt transformācijas.

4. Tiek atkārtoti izskaitļoti virsotņu dziļumi.

5. Tiek aprēķināts sarežģītības līmenis, balstoties uz izdarītām transformācijām un palikušo virsotņu skaitu (ņemot vērā transformāciju sarežģītības atlaidi). Ja sarežģītība ir pietiekama, tad līmenis ir gatavs. Ja tas nav pietiekams, tad atkārtojam soļus no 2 punkta.

Pateicoties trešajam punktam, bonusi tiek sasniegti secībā no vistuvākā uz tālāko. Tātad izdevīgākais ceļš līmeņa spēlēšanai ir tāds pats, ka algoritmā aprakstītais (transformāciju secībā) un sarežģītības līmenis tiem abiem sakrīt, kas arī dod vēlamo sarežģītību.

Jāpiemin, ka dažreiz virsotnes var beigties pirms vēlamais sarežģītības līmenis ir sasniegts. Tas nozīmē, ka sākumā ir jāaprēķina minimālais grafa garums un cik maza dziļuma virsotnes ir jāizvēlas pārvēršanai bonusa virsotnēs.

5.1.5. Ātrdarbība

Izmantosim iepriekšējo nodaļu, lai saprastu algoritma sarežģītību. Algoritma sarežģītība tiek mērīta attiecībā pret virsotņu skaitu n.

1. Pirmkārt, lai izveidotu nejaušu grafu, ir vajadzīgas n-1 nejaušas transformācijas (viena transformācija vienmēr izveido vienu virsotni un mēs sākam ar vienu starta virsotni).

2. Lai aprēķinātu grafa virsotņu dziļumus, ir nepieciešams iziet visas grafa virsotnes, kas aizņem n soļus.

3. Līmeņa sarežģītības noteikšanai jāveic viena transformācija uz katru virsotni, kas ir n soļi. Katru reizi, kad tiek pievienota bonusa virsotne, ir jāpārrēķina virsotņu dziļums. Sliktākajā gadījumā bonusa virsotņu skaits nepārsniedz n, tāpēc tas nevar aizņemt vairāk par n^2 soļu.

Tātad sliktākajā gadījumā grafs tiek izveidots $O((n-1)+n+n+n^2) = O(n^2)$ laikā. Ņemot vērā, ka virsotņu skaits nav liels, tad šo algoritmu var pielietot spēles laikā.

5.1.6. Fenotipa apraksts

Soļi, kas nepieciešami fenotipa izveidošanai izmantojot iepriekš izveidoto grafu:

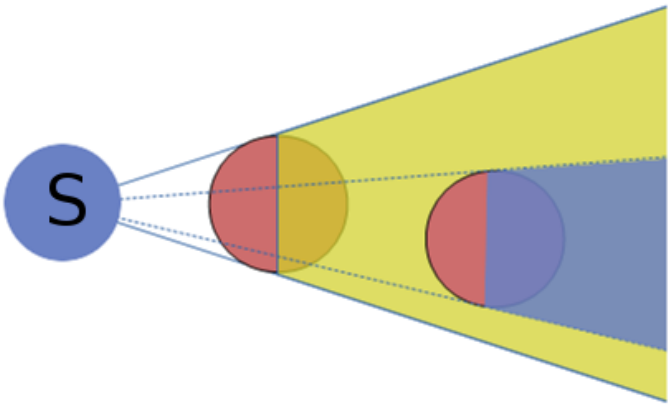
1. Tiek paņemta virsotne S, kurai tiek izveidots spēlētājs laukuma centrā. Ar šo virsotni tiek turpināts 2. solis.

2. Paņemtajai virsotnei tiek aprēķināts laukums, ko tā aizsedz. Starta virsotnei tas ir viss spēles laukums, kamēr pārejās virsotnes aizsedz laukumu kā redzams *5.1.att.*

3. Aizsegtajā laukā tiek izvietoti ienaidnieki attiecīgi paņemtas virsotnes izejošām virsotnēm un, paņemot katru no tiem, tiek rekursīvi pielietots šis algoritms no 2. soļa.

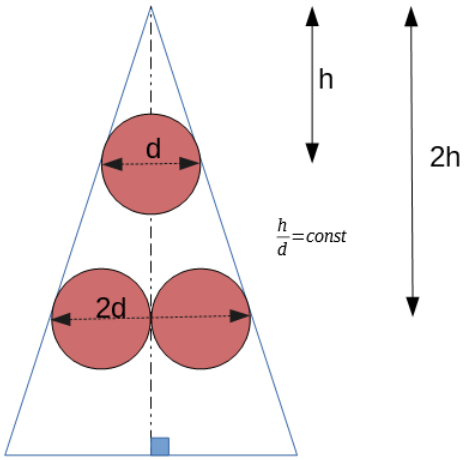
4. Tiek aizvietoti attiecīgo atslēgu, slēdžu un bonusa attēli.

Rekursīvas laukuma noteikšanas piemēru var redzēt *5.7.att*, kur ar dzelteno laukumu parādīts pirmās virsotnes aizsegtais laukums, kamēr zilajā laukumā ir otras virsotnes aizsegtais laukums.



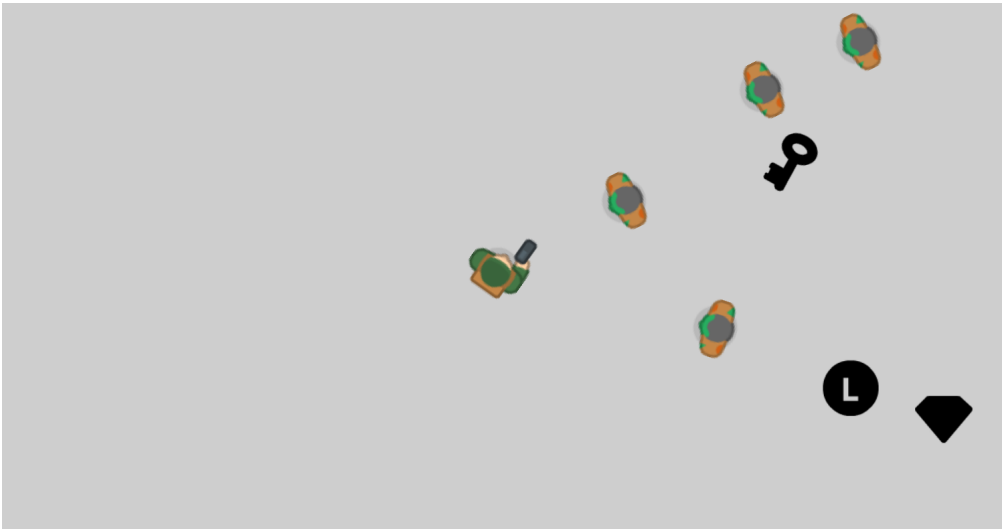
5.7.att. **Divu secīgu virsotņu aizsegtie laukumi**

lai aprēķinātu minimālo attālumu, kurā aizsegtajā laukumā var vienlaikus ievietot 2 ienaidniekus, izmantojam līdzīgo trīsstūru īpašību. Šī īpašība paredz to augstuma un pamata garuma attiecības saglabāšanos. Balstoties uz *5.8.att*, tas dod iespēju aprēķināt minimālo attālumu (h), kurā var ievietot 2 ienaidniekus.



5.8.att. **Minimālais attālums starp ienaidnieku un 2 tā aizsēdamajiem ienaidniekiem.**

Tādā veidā tiek izveidots spēlējams fenotips. Fenotipa piemēru, kas izveidots no *5.6.att* redzamā genotipa, var redzēt *5.9.att*. Tajā centrā ir spēlētājs, L apzīmē slēdzi un dimanta figūra ir bonusa ienaidnieks.



5.9.att. **Aprakstītās spēles ekrānuzņēmums**

5.1.7. Variācijas

Piedāvātajās metodēs netika izmantoti pārāk specifiski elementi, kas dod iespēju izmantot šo ģenerēšanas metodi citām spēlēm. Piemēram, zvejošanas spēle, kur, šaušanas vietā, tiek pievilkti ienaidnieki (jeb zivis, priekšmeti), var tikt izveidota neieviešot nekādas izmaiņas genotipā. Tāpat līdzīgu metodi var izmantot spēlē, kur tiek izmantota Dekarta telpa, ar tas koordinātu sistēmu. Vienīga atšķirība šeit ir tas, ka leņķa vietā šeit tiek izmantota x koordināte, un rādiuss ir y koordināte.

5.2. Spēle ar labirintu

Agrāk tika apskatīta Spellunky spēle un pateikts cik viegla ir šis spēles ģenerēšana, kas padara to par labu piemēru šim darbam. Šajā apakšnodaļā tiks apskatīts vispārīgs gadījums, kur apvienojot vienkāršas PĢS metodes, var izveidot spēlējamus līmeņus ar vairākām variācijām.

5.2.1. Spēles apraksts

1. Spēlētājs kontrolē avatāru, kas var pārvietoties pa labi, pa kreisi, lēkt un mijiedarboties ar spēles līmeņa objektiem.

2. Spēlēs mērķis ir tikt līdz noteiktam punktam, kuru turpmāk sauksim par finišu. Tajā punktā varētu būt izeja, vai priekšmets, kuru spēlētājs varētu savākt.

3. Spēlē darbojas gravitācijas likumi.

4. Spēlē ir vairāki objekti, kas var vai nu traucēt, vai arī palīdzēt tikt līdz finiša punktam.

5.1.2. Spēles pamatjēdzieni un īpašības

Spēle sastāv no rūtiņām, kuru izmērs sakrīt ar spēlētāja izmēru un katra objekta izmēru. Spēles līmeņa sākumā tajā katrai rūtiņai ir noteikts tips:

1. Brīva rūtiņa – rūtiņa, pa kuru var brīvi pārvietoties.

2. Bloķētā rūtiņa – rūtiņa, pa kuru nav iespējami pārvietoties, bet ir iespējams uz tās stāvēt.

3. Ienaidnieka rūtiņa – rūtiņa, kurai pieskaroties spēlētājs zaudē.

4. Batuts – rūtiņa, kura ļauj veikt augstu lēcienu.

5. Kāpnes – rūtiņa, kura ļauj pārvietoties vertikālajā virzienā.

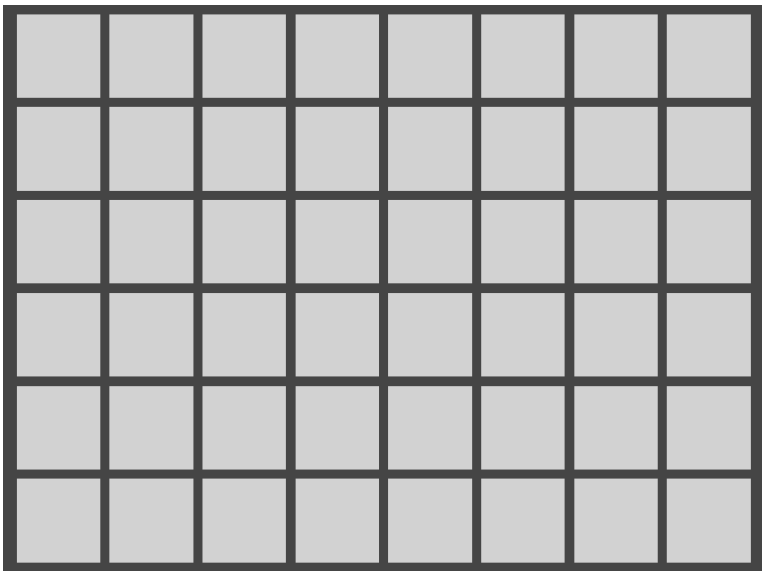
6. starta rūtiņa – rūtiņa kurā iesāk ceļu spēlētājs.

5.2.3. Genotipa apraksts

Šajā spēlē kombinēsim dažādus PĢS veidus. Kombinācija notiek secīgi izpildot dažādus algoritmus, un katra iepriekšēja soļa rezultātā sadalīsim līmeni daļās, kuras varam ģenerēt, izmantojot nākamo algoritmu.

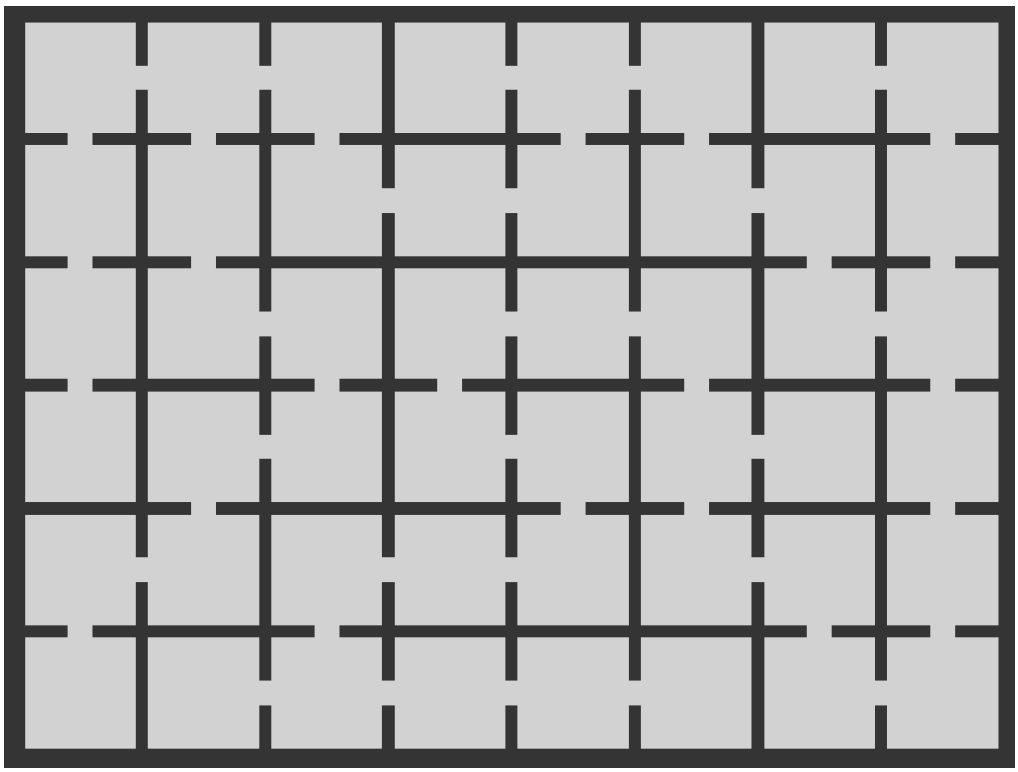
Pirmais solis šāda spēles līmeņa ģenerēšanā ir izveidot labirintu, kurā ir iespēja iziet no starta rūtiņas uz beigu rūtiņu. Jā, mēs taisītu šo labirintu no spēles mazākajam iespējamajām rūtiņām, tad spēles sastāvētu tikai no labirinta un tai nebūtu nekāda papildus spēlējamība (tai skaitā nevarētu izvietot papildus objektus, tādus kā ienaidniekus). Tieši tāpēc labirints tiek veidots, izmantojot lielākas rūtiņas, kuras sauksim par istabām.

Tātad lai izveidotu labirintu, vispirms izveidojam rūtiņu laukumu, kur katra rūtiņa apzīmē istabu. Piemēram 8x6 istabu laukumam parādīts *5.10. att*.



5.10.att. **Istabu izkārtojums 8x6 rūtiņās**

Pēc tam, izmantojot jebkuru labirinta algoritmu (piemērā izmantots iepriekš aprakstītais "Backtracking" algoritms), šīs istabas tiek savienotas ar nelielu "tuneli". Tādas savienošanas rezultātu var redzēt *5.11.att*. Šeit var redzēt, ka tika izveidots labirints, kur no katras rūtiņas var nonākt līdz jebkurai citai.



5.11.att. **Rūtiņu labirints**

Izveidotajā labirintā varam atsevišķi izpētīt katru rūtiņu un secināt, ka tas var sadalīt kategorijās, atkarībā no tā, kuros virzienos šīm rūtiņām ir pieejamas izejas. Saskaitot tas, iegūsim 15 rūtiņu variācijas. Nākamajā solī šis rūtiņas ir jāaizvieto ar iepriekš izveidotajam istabām. Tā paveikšanai pastāv vairākas stratēģijas:

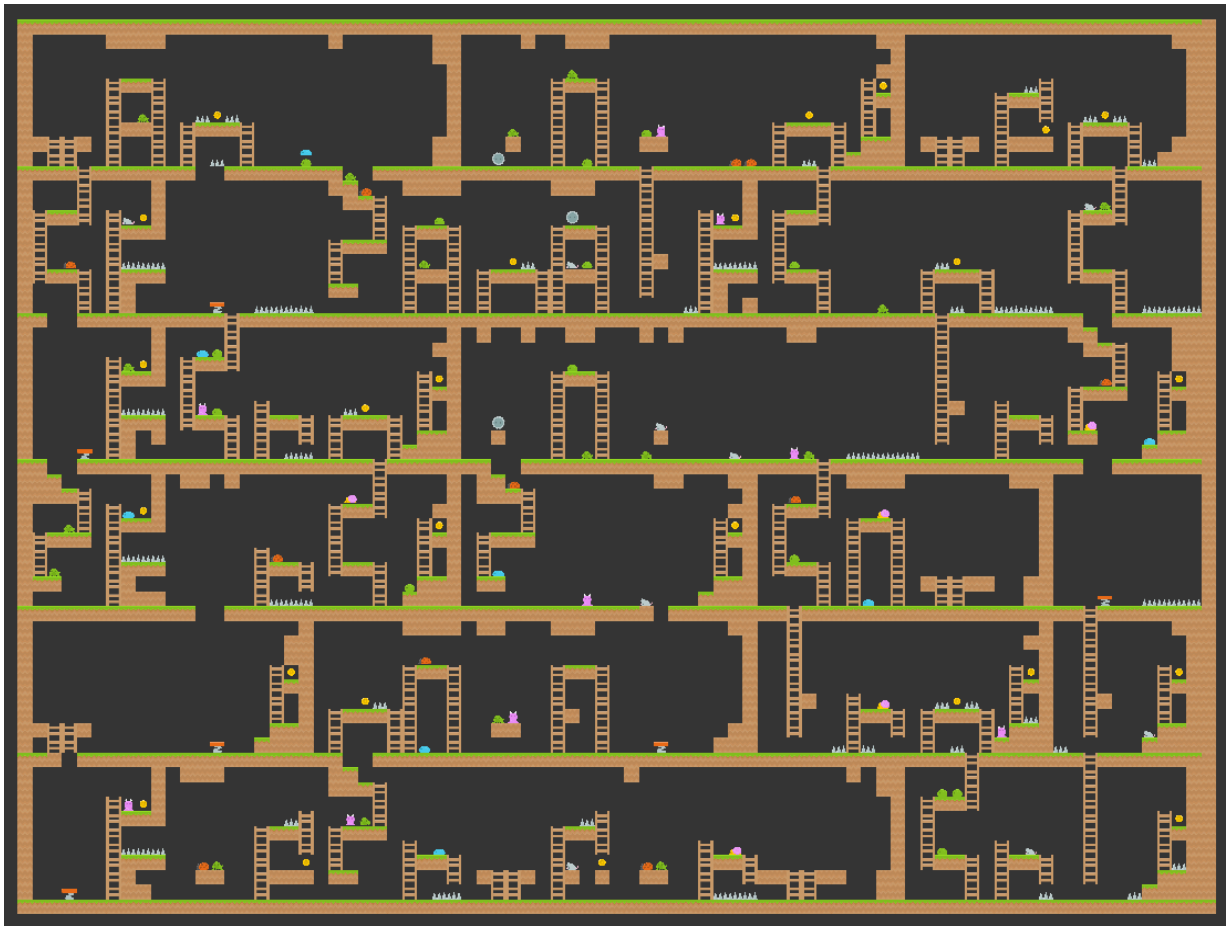
a) Līdzīgi Spellunky spēlei, istabu krājumā var atrast istabu ar attiecīgai rūtiņai sakrītošu izeju skaitu. Šāds veids ir algoritmiski ļoti vienkāršs un dod augstas kvalitātes istabas, taču izveidot vairākas istabas priekš katras rūtiņu variācijas ir pietiekami liels darbs, tāpēc ir vērts apskatīt sarežģītākas, bet mazāk laikietilpīgas alternatīvas.

b) Tā kā katrs tunelis atrodas starp divām rūtiņām, tad apakšējo un kreiso istabas malu vienmēr var turēt atvērtu. Tādā veidā par izejas aizvēršanu vienmēr rūpēsies labā un apakšējā rūtiņa. Šāds veids samazina iespējamo istabu variāciju skaitu līdz 4 variācijām, kas gandrīz 4 reizes samazina līmeņu veidošanas darbietilpību. Protams, šādam variantam apakšējā un kreisā mala vienmēr izskatīsies nedabiski, jo tiek kontrolēta tikai labā un augšējā mala.

c) Istabām var izveidot universālas variācijas, kur katra mala ir atvērta. Tādā gadījumā, lai izveidotu slēgtas/atvērtas izejas, tiek izveidotas tuneļu variācijas. Tas dod iespēju veidot variācijas vienam istabas veidam, vienam horizontālam tuneļa veidam un vienam vertikālajam

tuneļa veidam. Kopumā šeit variantu skaits ir pietiekami līdzīgs b variantam un to izskats varētu labāk iederēties dažādos spēļu stilos.

Saliekot istabas izmantojot vienu no aprakstītajam stratēģijām, var panākt to pašu labirintu, bet ar dabiskāku izskatu un iespējam pievienot dažādus šķēršļus un citus objektus, kuri uzlabos spēli. Piemēram, izmantojot b variantu no *5.11.att* redzamā labirinta var izveidot *5.12.att* redzamo istabu labirintu. Šajā labirinta spēlētajam ir iespējas pārvietoties starp tām istabām, starp kurām šī pārvietošanas ir iespējama arī *5.11.att* redzamajā rūtiņu labirintā.



5.12.att. **Istabu labirints un spēles ekrānuzņēmums**

Tālāk, nosakot starta un finiša istabu, tiek izveidots spēlējams līmenis, bet to var uzlabot.

a) Vieglākais veids to paveikt ir ieviest varbūtiskas rūtiņas istabu krājumā. Varbūtiskās rūtiņas ir istabas sastāvdaļa, kuru pastāvēšana neietekmē līmeņa spēlējamību. Tas nozīmē, ka ieliekot istabas variantu labirinta rūtiņā, ir iespēja iekļaut un neiekļaut šīs daļas. Šī iekļaušana

ietekmēs tikai istabas izskatu, bet spēlētajam tas dos iespaidu par lielāku istabu variāciju spēlē.

b) Vēl viens viegls uzlabošanas veids ir nejauši salikt ienaidniekus. Ienaidniekiem ir objekti, kuri negatīvi ietekmē spēlētāju, bet kurus ir iespējams iznicināt. Tas nozīmē, ka nav svarīgi, kur šie objekti ir izvietoti, jo jā tie bloķē ceļu līdz finišam, tad pastāv iespēja tos noņemt no šī ceļa.

c) Cits veids veikt uzlabojumus ir atrast līmenī noteiktas rūtiņu kombinācijas, kuras var aizvietot ar citām rūtiņu kombinācijas, neietekmējot līmeņa iziešanas gaitu. Šādas rūtiņu kombinācijas parasti varētu būt šķēršļi, kurus jāiziet, lai tiktu no viena punkta uz citu.

Izmantojot aprakstīto secību, ir iespējams izveidot spēlējamus līmeņus un ļoti labi noslēpt to, ka līmenis tika ģenerēts, kombinējot tikai dažas istabu variācijas.

5.2.4. Līmeņa sarežģītība

Līmeņa sarežģītību ir iespējams kontrolēt katrā līmeņa būvēšanas brīdī.

1. Nosakot lielāku rūtiņu laukumu istabām, tiek potenciāli palielināta maksimālā iespējamā līmeņa sarežģītība.

2. Būvējot labirintu, var kontrolēt garākā ceļa garumu, kas dod iespēju ielikt starta un finiša istabas šī ceļa pretējos galos. Jo lielāks šis ceļš, jo lielāka sarežģītība.

3. Istabu krājumā ir iespējams definēt katra istabas varianta sarežģītību un tad izmantot šo sarežģītību, lai noteiktu kuru istabu likt labirinta rūtiņās.

4. Var likt vairāk vai mazāk ienaidnieku. Jā spēlē pastāv bonusa priekšmeti, tad arī tie var regulēt līmeņa sarežģītību.

5. Iepriekšējā apakšnodaļā minētie šķēršļi arī var tikt sagrupēti to sarežģītības pakāpē un likti līmenī, balstoties uz šo pakāpi.

Kopumā tas dod ļoti labu kontroli par līmeņa sarežģītību un noderīgi ir tas, ka pārsvarā pirmie soļi stipri nobīda sarežģītību, kamēr pēdējie soļi ļauj veikt mazas un precīzas nobīdes spēles līmenī.

5.2.5. Ātrdarbība

Algoritma ātrdarbību šajā spēlē visvienkāršāk rēķināt attiecībā pret istabu skaitu.

1. Labirinta izveidošana pēc iepriekš aprakstītā algoritma var tikt paveikta O(n) laikā.

2. Istabas var tikt izvēlētas nejauši no konstanta variāciju skaita, tāpēc var pieņemt, ka tas strādā O(1) laikā.

3. Ienaidniekus un neatkarīgos objektus var likt atsevišķi katrai istabai. Ņemot vērā konstantu ienaidnieku un objektu skaitu katrā istabā, šī darbība ir paveicama O(n) laikā.

Kopumā līmeņa ģenerēšana aizņem O(n+1+n) = O(n) laiku un, ņemot vērā to, ka istabu skaits vienmēr ir ļoti mazs, šis algoritms strādā ļoti ātri.

5.2.6. Fenotipa apraksts

Fenotips intuitīvi tuvs jau aprakstītajam genotipam, tāpēc lai to izveidotu, ir nepieciešami tikai daži soļi:

1. Rūtiņām un objektiem tiek pievienotas to fiziskās īpašības. Piemēram, ienaidniekiem pievieno gravitācijas kodu un sienām definē kolīzijas robežas.

2. Starta istabā tiek ievietots spēlētājs, un finiša istabā pievieno durvis, vai kādu citu priekšmetu, kas simbolizē izeju.

3. Rūtiņām un objektiem tiek ielikti to attiecīgie attēli. Piemēram, *5.12.att* ir jau parādīts līmeņa fenotips, nevis genotips.

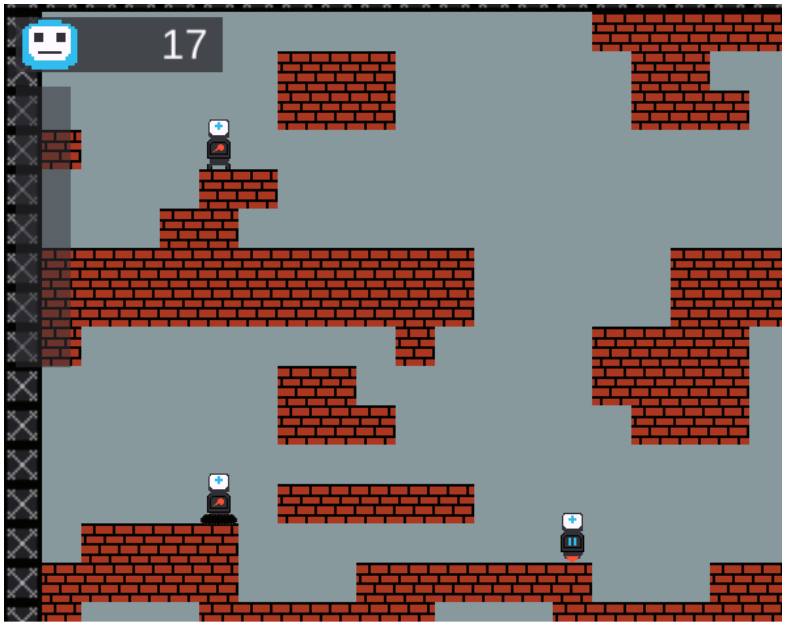
5.2.7. Variācijas

Šis ir ļoti vienkāršs un plaši pielāgojams PĢS sistēmas paveids. Tā variācijas ietver:

1. Noņemot gravitāciju, var veidot tādus līmeņus ar skatu no augšas. Šādi spēle līdzināsies kādai no "Roguelike" spēlēm.

2. Izveidojot nosacījumu, ka bloķētās rūtiņas (sienas) var iznicināt, var izlaist labirinta ģenerēšanas soli un likt istabas pilnībā nejauši. Tādā veidā var arī veidot spēles, kuru galvenais mērķis ir iznicināt pēc iespējas vairāk rūtiņu. Tādas spēles piemērs ir autora izveidotā "Hack a bot base" spēle, kura redzama *5.13.att*. Attēlā redzamas 4 istabas, kur katrā

no tām sākumā bija robots, bet viens no tiem tika uzspridzināts, iznicinot grīdas rūtiņas, un mākslīgi veidojot izeju uz zemāku līmeni.



5.13.att. **"Hack a bot base" spēle**[1]

5.3. Atjautības spēle

Iepriekš minētas spēles bija iespējams pārvērst diezgan vienkāršos matemātiskos modeļos, kuriem līmeņu veidošana balstījās uz šo modeļu īpašībām. Pārsvarā šajās spēlēs fenotips stipri atšķiras no genotipa un ļāva noslēpt vienkāršus līmeņa atrisinājumus, izmantojot ierobežotu laiku, vai nepilnu informāciju.

Pastāv arī spēles, kurās nav iespējams atrast tādu genotipu, kas vienlaikus saglabās fenotipa īpašības un tajā pašā laikā ļaus ātri ģenerēt šāda genotipa variācijas. Parasti tādas ir atjautības spēles, jo tas ir pietiekami abstraktas un visa informācija tajās ir dota no sākuma.

Tieši tāpēc šajā nodaļa tiks apskatīta atjautības spēle, kuras ģeneratoru varēs pielāgot jebkurai līdzīgai spēlei balstoties uz genotipu, kas gandrīz nevienkāršo tas fenotipu.

1 https://fancyhat.itch.io/hack-a-bot-base

5.3.1. Spēles apraksts

Spēle ir iespējama kustība četros virzienos (pa labi/kresi, uz augšu/leju). Tā sastāvēs no rūtiņu laukuma, kur katrai rūtiņai var būt viens no 5 sākuma stāvokļiem:

1. Tukša rūtiņa – rūtiņa, caur kuru neapstājoties var iziet spēlētājs.

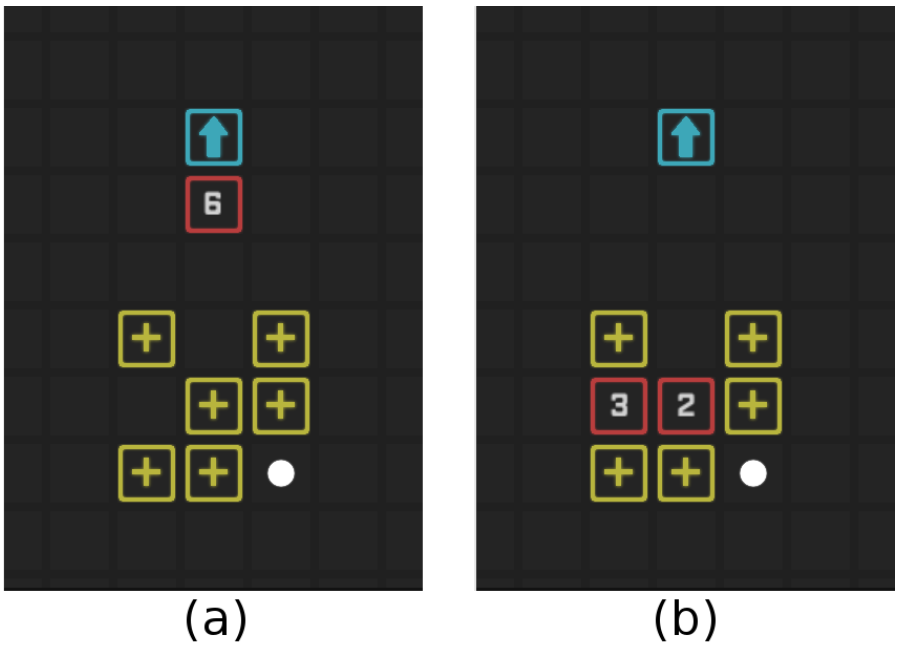
2. Spēlētāja sākuma rūtiņa – rūtiņa, no kuras spēlētājs sāk savu ceļu.

3. Finiša rūtiņa – rūtiņa, kura ir jāsasniedz spēlētajam, lai izietu līmeni.

4. Pluss rūtiņa – rūtiņa, kas palielina punktu skaitu par 1 un pārvēršas par tukšu rūtiņu pēc tam, kad spēlētājs to sasniedz.

5. Durvju rūtiņa – rūtiņa, kurai ir noteikts ieejas punktu skaits. Sasniedzot šo rūtiņu, tiek pārbaudīts spēlētāja punktu skaits. Ja punktu skaits ir mazāks par rūtiņas ieejas punktu skaitu, tad spēlētājs zaudē. Pretējā gadījumā rūtiņa pārvēršas par tukšu rūtiņu un spēlētāja punktu skaits tiek anulēts.

Sākot kustību jebkurā virzienā, spēlētājs kustas šajā virzienā, līdz tas sasniedz jebkuru netukšu rūtiņu. Piemēram, daži vienkārši spēles līmeņi var izskatīties, tā kā tas redzams *5.14.att.* Attēlā baltais aplis ir spēlētājs, bulta ir izeja un rūtiņa ar ciparu ir durvis, kur cipars apzīmē nepieciešamo punktu skaitu.



5.14.att. **Atjautības spēles ekrānuzņēmumi**[2]

2 https://fancyhat.itch.io/plustaka

5.3.2. Spēles pamatjēdzieni un īpašības

Spēlē ir nedaudz līdzīga labirinta spēlēm, jo tajā pēc būtības ir jāatrod pareizais ceļš, jeb pareizā gājienu secība. Lielākā atšķirība šeit ir saistīta ar to, ka spēles laukums mainās ar katru gājienu, tāpēc genotipā nevar vienkārši pielietot jau aprakstīto labirinta ģenerēšanas algoritmu. Vēl viens ierobežojums ir saistīts ar to, ka šajā spēlē nepastāv sienas (pastāv iespēja ieviest durvis ar pārāk lielu punktu skaitu un uzskatīt to par sienu, bet šoreiz uzskatīsim, ka visām durvīm jāatrodas risinājuma ceļā), tāpēc nepastāv arī iespēja ierobežot spēlētāja kustību jebkurā virzienā.

Apskatot *5.14.att (a)* redzamo līmeni, var pamanīt, ka līmeņa atrisināšanai ir jāiziet pilnīgi visas plusa rūtiņas. Ņemot vērā spēles nosacījumu, ka katra rūtiņa pazūd, tad var secināt, ka katra rūtiņa tiks izieta tikai vienreiz. Šie divi nosacījumi sakrīt ar Hamiltona ciklu atrisināšanas problēmu, kurai nav polinomiāla laika atrisinājumu. Protams spēles kustība ir ierobežotas tikai četros virzienos, kas stipri atvieglo līmeņa atrisināšanu, bet tas parāda, ka spēlei nepastāv viegls matemātisks risinājums.

5.3.3. Genotipa apraksts

Kā tika minēts, genotips šai spēlei ir ļoti tuvs fenotipam, ko var redzēt *5.14.att*. Viss spēles līmenis tiek raksturots ar rūtiņām un spēlētāja starta pozīciju. Spēlējama līmeņa izveidošana ir diezgan vienkārša, izmantojot rūtiņu likšanu atgriezeniskajā secībā.

Tālāk aprakstītajā algoritmā kaimiņu rūtiņu definēsim, ka rūtiņu, kurai attiecībā pret apskatāmo rūtiņu vienlaikus izpildās sekojošie 2 nosacījumi:

a) Tā ir tukša rūtiņa, kurai ar apskatāmo rūtiņu sakrīt vai nu x, vai arī y koordināte.

b) Starp šo un apskatāmo rūtiņu pastāv tikai tukšas rūtiņas.

Ņemot vērā šo definīciju, tālāk aprakstīts līmeņa ģenerācijas algoritms:

1. Tiek izveidota durvju rinda. Šī rinda sastāv no veseliem skaitļiem, kuri simbolizē durvju ieejas punktu skaitu. (Rinda var tikt izveidota nejauši, vai būt definēta iepriekš.)

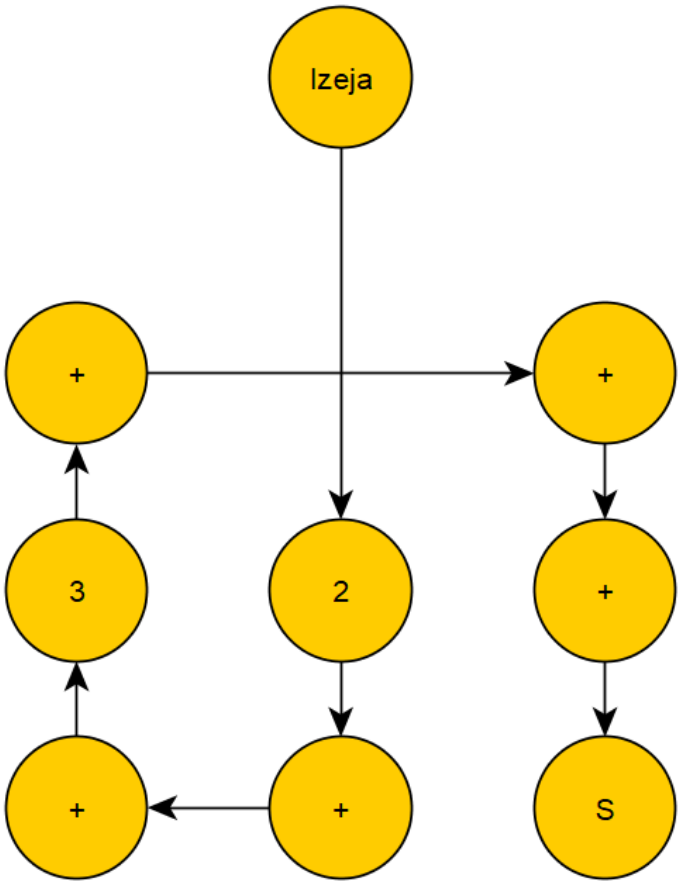
2. Tiek izvelēta finiša rūtiņa un no tās sākts ceļš.

3. No izvelētās rūtiņas nejaušajā kaimiņu rūtiņā tiek ievietota un izvēlēta durvju rūtiņa. Durvju rūtiņa tiek ievietota izņemot pirmo skaitli no durvju rindas, un izmantojot to, ka durvju ieejas punktu skaitu. Nākamais solis tiek atkārtots x reižu, kur x ir tikko izmantotais skaitlis.

4. No izvelētas rūtiņas tiek izvelēta jauna kaimiņa rūtiņa ar nosacījumu, ka šī rūtiņa nav beigu rūtiņas kaimiņš. Izvēlētajā rūtiņā tiek ielika plusa rūtiņa.

5. Jā durvju rindā vel palikuši skaitļi, tad algoritms turpina darbu ar trešo soli. Pretējā gadījumā zem rūtiņu lakuma tiek izveidota papildus rūtiņu rinda un šajā rindā, zem piedējas izvēlētas rūtiņas, tiek ielikta spēlētāja starta rūtiņa.

Šāda algoritma darbības secības attēlojumu var redzēt *5.15.att*, kur durvju rinda ir {2, 3} un līmenis atbilst *5.14.att (b)* redzamajam. Līmeņa atrisinājuma ceļš ir pretējs līmeņa būvēšanas ceļam.



5.15.att. **Līmeņa būvēšanas secība**

Aprakstīto algoritmu var uzlabot vairākos veidos, izmantojot papildus pārbaudes, bet šoreiz uzskatīsim, ka algoritmu optimizēt vairs nav iespējams, jo tas dod iespēju viegli pielāgot to citām līdzīgam spēlēm.

Izmantojot šo aprakstīto algoritmu, tika izveidots spēlējams līmenis, taču šajā līmeni var pastāvēt nepilnības, kas saistītas ar līmeņa sarežģītību. Ģenerēšanas laikā nepārliecinājāmies par to, ka spēlētājs nevar nogriezt ceļu un kaut kādā brīdī atvērt nepareizās durvis, tajā pašā laikā izejot līmeni daudz ātrāk.

Lai atrisinātu šo problēmu, līmenī var palaist aģentu, kas pēc līmeņa ģenerācijas iziet to visos iespējamos veidos un pārliecinās, ka neviens no šiem veidiem nav mums nevēlams. Jā aģents atrod ceļa saīsinājumu, tad tiek ģenerēts cits līmenis.

5.3.4. Līmeņa sarežģītība

Līmeņa sarežģītība tiek kontrolēta izmantojot durvju rindu, ņemot vērā sekojošas īpašības:

a) Gājienu skaits = durvju rindas elementu skaits + durvju rindas elementu summa.

b) Lielāks durvju skaits un garāks ceļš līdz durvīm palielina līmeņa sarežģītību.

Sarežģītības novērtēšanu pēc līmeņa izveidošanas var veikt, izmantojot aģentu, kas pārbauda ceļu:

a) Jo vairāki virzieni izmantoti aģenta atklātajā labākajā risinājumā, jo tas risinājums sarežģītāks.

b) Jo dziļāk aģents aiziet no pareizā ceļa, pirms nonāk strupceļā, jo ilgāk šo ceļu būs jāmeklē spēlētajam.

5.3.5. Ātrdarbība

Ātrdarbību var novērtēt attiecībā pret gājienu skaitu n. Kā tika minēts genotipa aprakstā, procesu var sadalīt divās daļās:

1. Tiek ģenerēts nejaušs līmenis. Šo darbību var paveikt n soļos, jo katrā solī tiek ģenerēta viena rūtiņa, kas iznākumā dod vienu gājienu.

2. Tiek pārbaudīti visi šī līmeņa iziešanas varianti. Šī darbība rāda ātrdarbības problēmas, jo katru nākamo gājienu sliktākajā gadījumā var iziet 4 dažādos veidos (attiecīgi 4 virzienos). Tas rada 4^n algoritma darbības, kas dod $O(a^n)$ izpildes laiku.

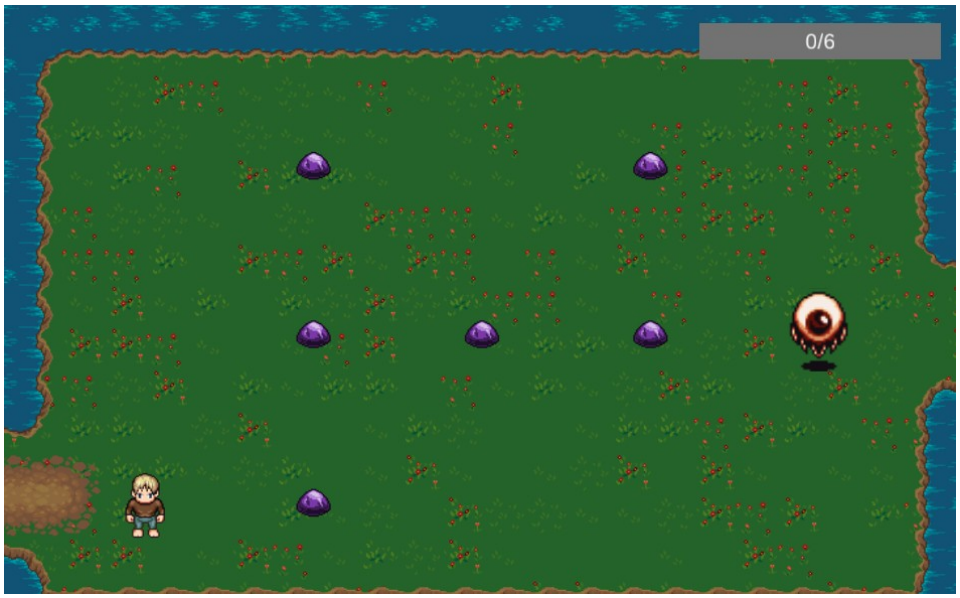
Kopā sanāk $O(n+a^n) = O(a^n)$ izpildes laiks, kas sarežģītākos gadījumos neļauj ģenerēt kartes spēles laikā, taču tas nenozīmē, ka šo PĢS sistēmu nevar izmantot. Līmeņus var ģenerēt iepriekš uz izstrādātāja datora un spēlē iekļaut jau gatavos līmeņus. Līmeņus var izvēlēties sekojošos veidos:

a) Izstrādātājs var vairākas dienas ģenerēt līmeņus ar vajadzīgo sarežģītību un tad ar rokām izvēlēties tos, kuri viņam vairāk patikas.

b) Var pilnveidot aģentu un izmantot evolucionāros algoritmus, kas tika aprakstīti iepriekš. Atbilstības funkcija varētu balstīties uz aģenta spēles iziešanas testiem, un mutācijas būtu viena nejauša kaimiņa izvēles maiņa ģenerācijas procesā.

5.3.6. Fenotips

Kā jau tika minēts, fenotips ir ļoti pietuvināts genotipam, bet ir iespējas aizvietot abstraktas rūtiņas ar sarežģītākiem attēliem un pievienot uzmanību novērsošus elementus, kā tas redzams *5.16.att*.



5.18.att. "**NsiT Adventure" ekrānuzņēmums**[3]

3 https://fancyhat.itch.io/nsit-adventure

5.3.6. Variācijas

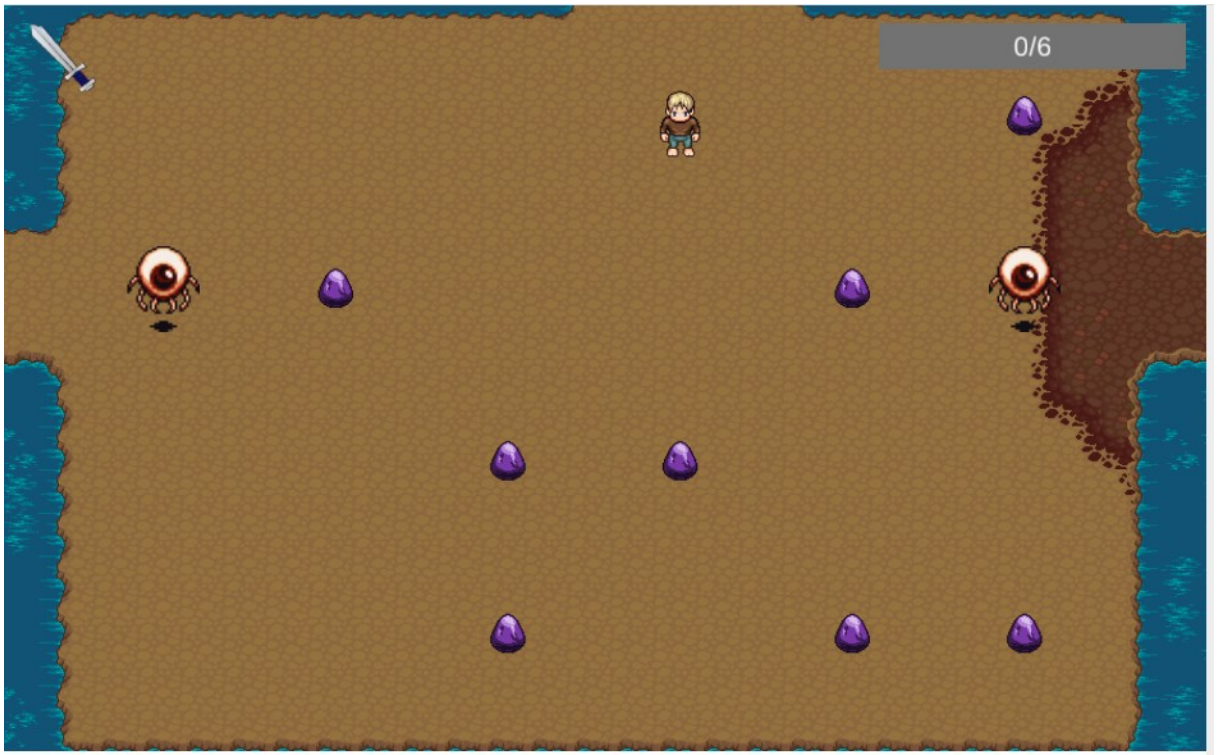
Pirmkārt aprakstītajam algoritmam ir ieviešamas tādas variācijas, ka, piemēram, vairākas izejas un papildus priekšmeti, kas palīdz iziet līmeni. Tādas īpašības, piemēram, var redzēt *5.18.att* redzamajai spēlei.

Tā kā tika izmantota pieeja, kur vispirms tiek ģenerēts līmenis un tad izmantots aģents, kas pārbaudītu līmeņa kvalitāti, tad var secināt, ka šo pieeju var izmantot jebkurai spēlei ar šiem nosacījumiem:

a) Spēlē ir galīgs iziešanas variantu skaits.

b) Spēlei ir iespējams ģenerēt nejaušu līmeni.

Šie nosacījumi izpildās vairākām loģikas un atjautības spēlēm, kas dod algoritmam plašas izmantošanas iespējas.



5.18.att. **Divas finiša rūtiņas**

REZULTĀTI UN SECINĀJUMI

Darbā definēts uzdevums bija saistīts ar PĢS sistēmu veidošanas atvieglošanu izstrādātajiem. Lai šo uzdevumu paveiktu, iesākumā tika definēti un aprakstīti daži pamatjēdzieni, kuri ļāva lasītajam saprast procesuālas ģenerēšanas šķēršļus.

Tālāk darbā aprakstītas spēles deva iespēju iepazīties ar konkrētiem piemēriem, kur PĢS jau tiek izmantots. Ņemot vērā to, ka lielāka daļa no šiem piemēriem ir pietiekami veiksmīgas spēles, tas deva iespēju saprast, kur PĢS ir efektīvi pielietojams.

Rīkos un paņēmienos autors izvēlējās vienkāršākos un plašāk pielietojamākos rīkus, kurus konkrētiem piemēriem ir iespējams pielietot praksē. Iepriekš aprakstītās nodaļas palīdzēja augstā līmenī aprakstītiem teorētiskiem rīkiem piedāvāt vienkāršu paskaidrojumu ar konkrētiem izmantošanas piemēriem, kas lasīšanas laikā ļauj saprast šo rīku un paņēmienu pielietojumu iecerētajos projektos.

Eksperimentu nodaļā autors aprakstīja konkrētas autora spēles, kuras, vienlaicīgi, ir arī ļoti pielāgojamas citiem spēļu žanriem. Katras spēles apraksts tika veikts līdzīgā formā, kura tika balstīta uz iepriekš definētiem svarīgiem jēdzieniem un kvalitātes pazīmēm. Tas ļauj apskatīt autora apsvērumus, kuri veikti veidojot PĢS sistēmu katrai spēlei un pielietot līdzīgus apsvērumus veidojot savu spēli.

Darbā laikā netika pārklāts ļoti plašs spēļu klāsts, sakarā ar to, ka katras PĢS sistēmas veidošana prasa pietiekami lielu laiku, kas gan liecina par šī darba nepieciešamību, gan arī par tā pilnveidošanas iespējam.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Jesper Juul: "The Game, the Player, the World: Looking for a Heart of Gameness". In Level Up: Digital Games Research Conference Proceedings, edited by Marinka Copier and Joost Raessens, 30-45. Utrecht: Utrecht University, 2003. Pieejams: http://www.jesperjuul.net/text/gameplayerworld/

2. "Games by Frontier – Elite" Arhivēts (2010). Pieejams: https://web.archive.org/web/20100127094607/http://frontier.co.uk/games/elite

3. Thomas M., Christoph M.:".kkrieger content creation in 96kb". Assembly Seminars (2004)

4. Noor S., Georgios Y., Julian T.: “Towards Automatic Personalized Content Generation for Platform Games”. (2010) Pieejams: https://www.aaai.org/ocs/index.php/AIIDE/AIIDE10/%20paper/viewFile/2135/2546

5. Jared Newman: "7 Things I Learned After 15 Hours with Borderlands 2" (2012). Pieejams: http://techland.time.com/2012/09/19/7-things-i-learned-after-15-hours-in-borderlands-2/

6. Booth M.: "The AI systems of Left 4 Dead" (AIIDE) (2009) Pieejams: http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf

7. Mike M., Tim L.: “Left 4 Dead ingame Developer Commentary” (2008) Pieejams: http://left4dead.wikia.com/wiki/Developer_Commentary_(Left_4_Dead)

8. Burgiel, H. (1997). How to lose at Tetris. The Mathematical Gazette, 81(491), 194-200.

9. Hellekalek, P.:"Good random number generators are (not so) easy to find" Mathematics and Computers in Simulation 46 (1998) 485-505. Pieejams: http://random.mat.sbg.ac.at/results/peter/A19final.pdf

10. Johnson, L., Yannakakis, G.N., Togelius, J.: "Cellular automata for real-time generation of infinite cave levels" (2010) Pieejams: https://www.itu.dk/~yannakakis/a7-Johnson.pdf

11. Browne, C., Maire, F.: "Evolutionary Game Design" IEEE Transactions on Computational Intelligence and AI in Games (2010) Pieejams: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.421.3450&rep=rep1&type=pdf

12. Dormans, Joris. “Adventures in level design: generating missions and spaces for action adventure games.” (2010). Pieejams: https://pdfs.semanticscholar.org/5716/8efaa56e7ee7742444a56c683e77738146cb.pdf

13. Togelius, J., Yannakakis, G.N., Stanley, K.O., Browne, C.: Search-based procedural content generation. (2010) Pieejams: https://www.itu.dk/~yannakakis/SB-PCG.pdf

14. Kate Compton: "Practical Procedural Generation for Everyone" Game Developer conference (2017) Pieejams: https://youtu.be/WumyfLEa6bU

15. Mnih, V., Kavukcuoglu, K., Silver D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: "Playing Atari with Deep Reinforcement Learning" NIPS (2013) Pieejams: https://arxiv.org/pdf/1312.5602v1.pdf

16. Derek Yu: "The Full Spelunky on Spelunky" Pieejams: http://makegames.tumblr.com/post/4061040007/the-full-spelunky-on-spelunky

17. "Weapons by prefix (Borderlands)" Pieejams: http://borderlands.wikia.com/wiki/Weapons_by_prefix_(Borderlands)

18. "How to Choose Colours Procedurally" Pieejams: http://devmag.org.za/2012/07/29/how-to-choose-colours-procedurally-algorithms/

19. Noor Shaker, Julian Togelius, and Mark J. Nelson: Procedural Content Generation in Games: A Textbook and an Overview of Current Research. (Trešā nodaļa) Springer (2016). Pieejams: http://pcgbook.com/wp-content/uploads/chapter03.pdf

20. Joris Dormans: "Everything you need to know about level design in 20x15 seconds" Ignite Amsterdam 5 (2011). Pieejams: https://youtu.be/0B59Er2BQp0

21. “Voronoi diagram” Pieejams:https://se.mathworks.com/help/matlab/ref/voronoi.html

PIELIKUMI

1. Pielikums

Plustaka un NsiT Adventure līmeņu generātors

```
import itertools
import random
import json

class MapSolution:
	def __init__(self, moves, points, boxes, checkpoints, w, h):
		self.moves = moves[::-1]
		self.points = [self.offset(p) for p in points[1:-1] if p not in boxes]
		self.spawn = self.offset(points[-1])
		self.boxes = list(zip(boxes, checkpoints))
		self.exit = points[0]
		self.width = w
		self.height = h
		tiles = {}
		tiles.update({p:0 for p in self.points})
		tiles.update({p[0]:p[1] for p in self.boxes})
		tiles.update({self.exit:-1})
		self.tiles = tiles
		self.valid = True
		self.box_count = len(boxes)

	def offset(self, point):
		return (point[0], point[1]) #(point[0], point[1]+1)

	def drawMap(self, delim='|'):
		map_text = []
		for y in range(self.height):
			map_line = []
			for x in range(self.width):
				map_line += ["o"]
			map_text += [map_line]
		for x, y in self.points:
			map_text[y][x] = "+"
		for (x, y), n in self.boxes:
			map_text[y][x] = str(n)
		map_text[self.exit[1]][self.exit[0]] = "e"
		map_text[self.spawn[1]][self.spawn[0]] = "s"

		return delim.join([''.join(line) for line in map_text])

	def getJSON(self):
```

```
                data = { 'moves':''.join(self.moves), 'spawn_x':self.spawn[0], 'spawn_y':self.spawn[1],
'map':self.drawMap()}
                return json.dumps(data)

        def moveTraverser(self, moves, taken_tiles, direction):
                for move in moves:
                        if move in taken_tiles:
                                return move
                return None


        def getNodes(self, current_pos, taken_tiles):
                yield self.moveTraverser(((x, current_pos[1]) for x in range(current_pos[0]-1, 0-1,
-1)), taken_tiles, 'd')
                yield self.moveTraverser(((x, current_pos[1]) for x in range(current_pos[0]+1,
self.width)), taken_tiles, 'a')
                yield self.moveTraverser(((current_pos[0], y) for y in range(current_pos[1]-1, 0-1,
-1)), taken_tiles, 's')
                yield self.moveTraverser(((current_pos[0], y) for y in range(current_pos[1]+1,
self.height)), taken_tiles, 'w')

        def checkMap(self):
                self.check(set(self.tiles.keys()), self.spawn, 0, 0)
                return self.valid

        def check(self, tiles, current_pos, boxcount, points):
                if not self.valid or len(tiles) == 0:
                        return
                for node in self.getNodes(current_pos, tiles):
                        if node is None:
                                continue
                        value = self.tiles[node]
                        if value == 0:
                                self.check(tiles-set([node]), node, boxcount, points+1)
                        elif value == -1:
                                self.valid = False
                        elif points >= value and  boxcount+1 < self.box_count:
                                        self.check(tiles-set([node]), node, boxcount+1, 0)



class MapGenerator:
        def __init__(self, checkpoints):
                self.w = 3
                self.h = 6
                self.checkpoints = checkpoints
                self.exits = [(0,0),(1,0),(2,0)]
                self.can_end = lambda p: p[1] == self.h-1
                self.forbidden_tiles = []
                self.ACCEPT_RATE = 1000 #100*(2**(checkpoints[0]-4))
                self.SOLUTIONS = []
                self.ACCEPTED =  0


        def indexPos(self, index):
```

```
            return (index%self.w, index/self.w)

        def moveTraverser(self, moves, taken_tiles, direction, check_forbidden):
            for move in moves:
                if move in taken_tiles:
                    break
                elif not check_forbidden or move not in self.forbidden_tiles:
                    yield (move, direction)

        @staticmethod
        def moveDiversity(moves):
            return len(set(moves))

        @staticmethod
        def linearMoves(moves):
            linear = 0
            prev_move = 'n'
            for move in moves:
                if move == prev_move:
                    linear += 1
                prev_move = move
            return linear

        def possibleMoves(self, current_pos, taken_tiles, check_forbidden = True):
            left_moves = self.moveTraverser(((x, current_pos[1]) for x in range(current_pos[0]-1,
0-1, -1)), taken_tiles, 'd', check_forbidden)
            right_moves = self.moveTraverser(((x, current_pos[1]) for x in
range(current_pos[0]+1, self.w)), taken_tiles, 'a', check_forbidden)
            up_moves = self.moveTraverser(((current_pos[0], y) for y in range(current_pos[1]-1,
0-1, -1)), taken_tiles, 's', check_forbidden)
            down_moves = self.moveTraverser(((current_pos[0], y) for y in
range(current_pos[1]+1, self.h)), taken_tiles, 'w', check_forbidden)

            return itertools.chain(left_moves, right_moves, up_moves, down_moves)

        def generate(self):
            for start in self.exits:
                for first_box, _ in self.possibleMoves(start, []):
                    self.forbidden_tiles = [m[0] for m in self.possibleMoves(start,
[first_box])]
                    self.makeMove([], [start, first_box], [first_box], first_box,
[self.checkpoints[0]], 0)
                    self.forbidden_tiles = []


        def makeMove(self, moves, tiles, boxes, current_pos, checkpoints, depth):
            if depth < checkpoints[-1]:
                for move, direction in self.possibleMoves(current_pos, tiles):
                    #print "{2}: {0} => {1}".format(current_pos, move, depth)
                    self.makeMove(moves+[direction], tiles+[move], boxes, move,
checkpoints, depth+1)
            elif len(checkpoints) < len(self.checkpoints):
                for move, direction in self.possibleMoves(current_pos, tiles, False):
                    #print "{2}: {0} => {1}".format(current_pos, move, depth)
```

```
                                        self.makeMove(moves+[direction], tiles+[move], boxes+[move],
move, checkpoints+[self.checkpoints[len(checkpoints)]], 0)
                        else:
                                #print "{2}: {0} => {1}".format(current_pos,
set(self.possibleMoves(current_pos, tiles)), depth)
                                home = [(m, d) for m, d in self.possibleMoves(current_pos, tiles) if
self.can_end(m)]
                                if len(home) > 0:
                                        final_moves = moves+[home[0][1]]
                                        final_points = tiles+[home[0][0]]
                                        solution = MapSolution(final_moves, final_points, boxes,
checkpoints, self.w, self.h)
                                        #solution.drawMap("\n")
                                        if solution.checkMap() and MapGenerator.moveDiversity(moves) > 3
and MapGenerator.linearMoves(moves) < 2:
                                                self.ACCEPTED+= 1
                                        #if MapGenerator.moveDiversity(moves) > 3 and
MapGenerator.linearMoves(moves) < 2:
                                                #if random.randint(0, self.ACCEPT_RATE) == 0:
                                                self.SOLUTIONS += [solution.getJSON()]


def main():
        accepted = {}
        for i in range(4, 10):
                gen = MapGenerator([i])
                gen.generate()
                random.shuffle(gen.SOLUTIONS)
                for solution in gen.SOLUTIONS[:5]:
                        print (solution)

                accepted[i] = gen.ACCEPTED

        print ("Maps Generated")
        for key in accepted:
                print ("{}: {}".format(key, accepted[key]))

def pair_main():
        accepted = {}
        for t in [(2,3), (3,2), (3,3), (4,3), (4,4)]:
                gen = MapGenerator(list(t))
                gen.generate()
                random.shuffle(gen.SOLUTIONS)
                for solution in gen.SOLUTIONS[:8]:
                        print (solution)

                accepted[t] = gen.ACCEPTED

        print ("Maps Generated")
        for key in accepted:
                print ("{}: {}".format(key, accepted[key]))


#main()
pair_main()
```

2. Pielikums

Platformera spēles līmeņu ģenerātors

```
public class PlatformGenerator : MonoBehaviour {

	const int CellSize = 10;

	public Button btnNext;

	public GameObject roomPrefab;
	public GameObject bridgePrefab;

	public int width = 6;
	public int height = 4;

	public int seed = 626340;

	GameObject[] rooms;
	List<GameObject> bridges;
	BtRectangle maze;

	RoomManager roomManager;

	int step = 1;

	public void OnButtonNext() {
		switch(step) {
		case 1:
			StartCoroutine (MazeStep1 ());
			break;
		case 2:
			StartCoroutine (MazeStep2 ());
			break;
		case 3:
			StartCoroutine (MazeStep3 ());
			break;
		case 4:
			StartCoroutine (MazeStep4 ());
			break;

		}
	}

	void Awake() {
		roomManager = GameObject.Find
("RoomManager").GetComponent<RoomManager> ();
	}

	GameObject AddRoom(int index, Vector3 pos) {
```

```
                return rooms [index] = Instantiate<GameObject> (roomPrefab, transform.position +
pos*CellSize, Quaternion.identity, transform);
        }

        IEnumerator MazeStep1() {
                btnNext.enabled = false;
                Random.InitState(seed);

                foreach (Transform t in transform) {
                        Destroy (t.gameObject);
                }
                roomManager.ClearRooms ();

                bridges = new List<GameObject> ();
                rooms = new GameObject[width * height];
                AddRoom (0, transform.position);
                maze = new BtRectangle (width, height);
                yield return maze.Generate (DrawRoom, 0f);
                step = 2;
                btnNext.enabled = true;
                yield return new WaitForSeconds (1);
                //yield return MazeStep2 ();
                //yield return MazeStep3 ();
                //yield return MazeStep4 ();
        }

        IEnumerator MazeStep2() {
                btnNext.enabled = false;

                foreach (GameObject g in bridges) {
                        Destroy (g);
                }

                yield return new WaitForSeconds (0.2f);

                for (int x = -CellSize/2; x <= CellSize * width - CellSize/2; x++) {
                        roomManager.DrawTile (2, new Vector3(x, -CellSize/2));
                }

                for (int y = -CellSize/2 +1; y <= CellSize * height - CellSize/2; y++) {
                        roomManager.DrawTile (4, new Vector3(-CellSize/2, y));
                        roomManager.DrawTile (4, new Vector3(CellSize * width-CellSize/2, y));
                }

                yield return new WaitForSeconds (0.1f);

                for (int i = 0; i < rooms.Length; i++) {
                        rooms [i].GetComponent<RoomScript> ().PlaceObjects (maze.GetCell(i));
                        yield return new WaitForSeconds (0.1f);
                }
                step = 3;
                btnNext.enabled = true;
        }

        IEnumerator MazeStep3() {
```

```
		btnNext.enabled = false;

		foreach (GameObject g in GameObject.FindGameObjectsWithTag("Token")) {
			g.GetComponent<IGrammarToken> ().RandomResolve ();
			yield return null;
		}
		step = 4;
		btnNext.enabled = true;
	}

	IEnumerator MazeStep4() {
		btnNext.enabled = false;

		foreach (GameObject g in GameObject.FindGameObjectsWithTag("Token")) {
			g.GetComponent<IGrammarToken> ().RandomResolve ();
			yield return null;
		}
		//step = 1;
		//btnNext.enabled = true;
	}

	void DrawRoom(Int2 pos1, Int2 pos2, int index) {
		Vector2 v1 = transform.position + new Vector3 (pos1.x, pos1.y,0);
		Vector2 v2 = transform.position + new Vector3 (pos2.x, pos2.y,0);

		GameObject r = AddRoom (index, new Vector3 (pos2.x, pos2.y,0));
		GameObject bridge = Instantiate<GameObject> (bridgePrefab, (v1 + v2) *CellSize/2,
transform.rotation);
		bridge.transform.SetParent (r.transform, true);
	}
}


public class Cell {
	static int unvisited = 0;
	bool visited;
	public bool Visited {
		get{return visited; }
		set{
			if (value != visited) {
				if (value == true) {
					unvisited--;
				} else {
					unvisited++;
				}
				visited = value;
			}
		}
	}

	public bool rightOpen;
	public bool topOpen;

	public Cell () {
		this.visited = false;
```

```
                this.rightOpen = false;
                this.topOpen = false;
                unvisited++;
        }

        public static bool AllVisited() {
                return unvisited == 0;
        }
}

public class Int2 {
        public int x;
        public int y;

        public Int2 (int x, int y)
        {
                this.x = x;
                this.y = y;
        }
}


public abstract class BackTrackGenerator {

        protected int width;
        protected int height;

        Cell[] cells;
        int current;

        protected abstract Int2 GetPos (int index);
        protected abstract int GetIdx (int x, int y);
        protected abstract IEnumerable<int> Neighbours (int index);

        public delegate void DrawFunction(Int2 cur, Int2 next, int index);

        void DrawNext(int next, DrawFunction drawFunction) {
                Int2 pos1 = GetPos (current);
                Int2 pos2 = GetPos (next);

                if (pos2.x > pos1.x) cells[current].rightOpen = true;
                if (pos1.x > pos2.x) cells[next].rightOpen = true;
                if (pos2.y > pos1.y) cells[current].topOpen = true;
                if (pos1.y > pos2.y) cells[next].topOpen = true;

                drawFunction (pos1, pos2, next);
        }

        public BackTrackGenerator (int width, int height) {
                this.width = width;
                this.height = height;
        }

        void Visit(int index) {
                current = index;
```

```
		cells [index].Visited = true;
	}


	public IEnumerator Generate(DrawFunction drawFunction, float delay = 0.1f) {
		cells = new Cell[width*height];
		for (int i = 0; i < cells.Length;i++) {
			cells [i] = new Cell ();
		}
		Stack<int> stack = new Stack<int> ();

		Visit (0);
		while (!Cell.AllVisited ()) {
			List<int> nodes = new List<int> ();
			foreach(int i in Neighbours(current)) {
				if (!cells [i].Visited) {
					nodes.Add (i);
				}
			}
			if (nodes.Count > 0) {
				int next = nodes [Random.Range (0, nodes.Count)];
				stack.Push (current);
				//remove wall current->next
				DrawNext(next, drawFunction);
				yield return new WaitForSeconds(delay);
				Visit (next);
			} else if (stack.Count > 0) {
				current = stack.Pop ();
			}
			//yield return new WaitForSeconds(delay);
		}
	}

	public Cell GetCell(int index) {
		return cells [index];
	}

}
```

Maģistra darbs " **Procesuāli ģenerētas spēļu pasaules**" izstrādāts LU Datorikas fakultātē.

Darba teksta galīgā versija izgatavota 18.01.2018.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____

(Autora paraksts un datums)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par **p i e m ē r o t u / n e p i e m ē r o t u** (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: _____

(Vadītāja paraksts un datums)

Darbs iesniegts **maģistratūras sekretariātā** _____.

(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā. Studiju metodiķe: _____.

(Metodiķes paraksts)

Recenzents: _____

(Akad.amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____

(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____

(Sekretāra paraksts)