TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

Infotehnoloogia teaduskond

Arvutisüsteemide instituut

Dmitri Fomitšjov 155221IAPB

PROTSEDUURILISELT GENEREERITUD MÄNGUMAAILM

Bakalaureusetöö

Aleksei Tepljakov
PhD
Teadustöötaja

Tallinn 2018

Autorideklaratsioon

Olen koostanud antud töö iseseisvalt. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud. Käsolevat tööd ei ole varem esitatud kaitsmisele kusagil mujal.

Autor: Dmitri Fomitšjov

20. mai 2018. a.

Annotatsioon

Töö eesmärgiks on teha algoritm, mis suudaks genereerida suvalist unikaalset mängumaailma ehk protseduuriliselt genereeritud mängumaailm lühidalt *PCG*. Töö on jaotatud kaheks osaks, esimeses osas on analüüsitud uuritavat algoritmi ja tööks vajalikke programme ning teises osas esitatakse praktiline osa. Algoritmi uurimisel selgus, miks mõned ettevõtted ei kasuta antud algoritmi ning millised on põhilised ohud antud algoritmi kasutamisel. Programmide seast tutvustatakse projektis kasutatud mängumootorit ehk programmi mängude tegemiseks ning esitatakse võrdlus teise samaväärse mängumootoriga. Praktilises osas on lahti seletatud loodud algoritmi tööpõhimõtted ning algoritmi suutlikkus probleeme lahendada. Praktilise osa lõpus on näidatud testimise tulemused ning võrreldud oodatavate tulemustega. Töö käigus osutus olulisemaks probleemiks algoritmi efektiivsemaks tegemine, järeldustes ilmnes, et tehtud algoritm on efektiivsem, kui oli projekti alguses oodatud.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 31 leheküljel, 7 peatükki, 27 joonist, 3 tabelit.

Abstract

The goal of this bachelor's degree paper is to write an algorithm which could generate unique levels, this type of algorithm is usually called procedurally generated content for short PCG. The main research is divided into two parts.

First part focuses on general information about PCG in which was determined why some companies avoid using such algorithms but at the same time there was some companies that used PCG in experimental purposes where they tried to make new and excitement games. In that part is also listed some of the PCG dangers due to which companies could lose profit or even go bankrupt. On the other hand if given PCG succeeds it can be of big profit to the company not only in making better games but in more efficient way of making new games.

The second part is focused on explaining how written algorithm works and showing how primary functions of algorithm behave in different situations. The biggest problem that algorithm will have is collisions with other objects and other minor problems. The last part of second part is testing methods and their results, about 30 tests were recorded and testing was done through whole process of making the algorithm and much more will be done in the future. Testing results show that algorithm is able to solve problems in moderate time and level generation efficiency is faster than it was predicted where averege time in seconds of generation is the size of the level divided by two, where at the beginning of testing the time to generate a level was estimated to take same amount of time in second as the size of the level. Written algorithm is very flexible and efficient due to its ability to solve problems on the fly with moderate speed.

The thesis is in estonian and contains 31 pages of text, 7 chapters, 27 figures, 3 tables.

Lühendite ja mõistete sõnastik

PCG	*Procedural content generation*, sisu protseduuriline genereerimine
Indie	*Independent* ehk Iseseisev / sõltumatu
Game engine	Mängu mootor ehk programm mängude tegemiseks
2D	Kahedimensiooniline ruum
3D	Kolmedimensiooniline ruum
Blueprints	Graafiline programmeerimiskeel
Action	Mängu tüüp ehk võistlusmäng
AAA	*Triple* a, videomängude klassifikatsiooni kõige kõrgem klass
High risk high reward	Suuremate riskidega tulevad suuremad tulemused
White box testing	Valge kasti testimine
Black box testing	Musta kasti testimine
Grey box testing	Halli kasti testimine
Basic	Programmeerimiskeel, mis kuulub lihtsamate keelte hulka
Malbolge	Programmeerimiskeel, mis kuulub raskemate keelte hulka
Blender	Programm 3D objektide tegemiseks
3D texture painting softwares	Programm 3D objektide illustreerimiseks
Drag and drop	Objekti tõmbamine ühest kohast teisse
Komponent nr1	Uue algoritmi esimene komponent
Komponent nr2	Uue algoritmi teine komponent

Sisukord

Jooniste loetelu

Tabelite loetelu

1. Sissejuhatus

Meelelahutused on peamised allikad stressi alandamiseks, kus meelelahutusi on palju erinevaid tüüpe ning iga tüüp on suunatud erinevatele vanusegruppidele [4]. Käesoleva bakalaureusetöö autor otsustas keskenduda mängu tüübile, kuna seda kasutatakse üha rohkem digitaalses maailmas, ning autor ise kasutab mänge vaba aja veetmiseks. Mängud omakord jagunevad erinevateks tüüpideks ja žanriteks [5, 6] ning iga ettevõte peab andma oma mängule hinnangu, mis peegeldaks nende mängu sisu [7, 8]. Hinnangu andmine on kohustuslik, kuna see määrab mängule vanusegrupi ning annab väga väikse ülevaate mängu sisust. Antud valdkonnas on peamiselt kaht liiki ettevõtteid: esiteks on suurettevõtted, kes teevad *AAA* mänge ehk videomängude klassifikatsiooni kõige kõrgema klassi mänge, ja teiseks *indie* ehk iseseisvad ettevõtted - nende erinevus ei seisne ainult liikmete arvus, vaid ka mängude kvaliteedis [9, 10]. Suurettevõtted on rohkem suunatud väljamõeldud või päris elust võetud loo rääkimisele, kus mängija on looga kogu aeg kaasas, näiteks hiljuti tulnud mäng *God of war* [11]. Teiseks *indie* ettevõtted, kelle mängud on madalama kvaliteediga, kui *indie* ettevõtted tahavad anda endast märku, peavad nad tegema midagi, mida pole varem tehtud - see on kaasajal väga raske. Keerukas on mingit mängu näidata uuest küljest, eksperimenteerides uute võimalustega. Üheks peamiseks allikaks eksperimenteerimisel on procedural content generation ehk protseduuriliselt genereeritud sisu, lühidalt *PCG*. *PCG* on algoritmide kogu, mis võimaldab genereerida erineva sisuga asju, näiteks suvalist 3D ehk kolmedimesioonilist objekti, suvalist mängumaailma ja palju teisi asju. *PCG* teeb mängu huvitamaks ja taas mängitavamaks, samas võimaldab mängu kiiremini ja efektiivsemalt valmistada ning mängijad ei pea uut mängu ostma, sest mängija saab sama mängu mängida mitut korda ning tunda rõõmu selle üle. Tänu sellele, et algoritm genereerib mängule sisu, ei ole ettevõttel vaja nende asjade peale aega kuulutada, vaid nad saavad keskenduda mõnele teisele probleemile. Hea *PCG* ei ole ainuke edufaktor hea mängu tegemiseks, sest on ka palju teisi faktoreid, mis mängu populaarsust mõjutavad [12].

Autori unistus on realiseerida omatehtud mängu, mis aitaks inimestel oma mõtteid teise kohta suunata ning et mängu saaks mitut korda uuesti mängida, selleks on vaja mängijale anda uus mängumaailm iga uue mängu alguses. Selle saavutamiseks tuleks teha *PCG*, mis suudaks genereerida uusi mängumaailmu ja seda ideed inspereeris päris edukas mäng *The Binding of Isaac*. Kuna autor ei tee seda projekti üksinda, vaid meeskonnas on ka üks disainer, siis *PCG* algoritm ei tohiks segada disaineri tööd, et seda saavutada, on vaja luua selline algoritm, mis oleks väga paindlik. Praeguse hinnangu põhjal ei ole tuleviku mäng *PG12* alla 12aastatele soovitatav, kuna mängus esinevad objektid ja tegevused, mis alla

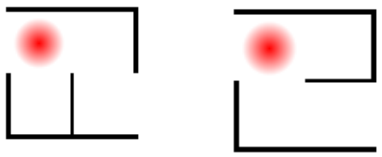
12aastatele on keelatud.

Lõputöö käigus selgitatakse, mida *PCG* endast kujutab, kus seda kasutatakse ning miks seda ei kasutata nii palju. Antakse lühiülevaade töökeskkonnast ja selle võimalustest, võrreldes mõninga teise samaväärse võimalusega. Lõpus seletatakse, kuidas loodud algoritm töötab, antakse ülevaade testimise meetoditest ning esitatakse testimise järeldused.

2. Protseduuriliselt genereeritud sisu

Procedural content generation lühidalt *PCG* on algoritmide kogum, mis võimaldab arvutil genereerida erinevaid asju. On olemas mängud, kus *PCG*d kasutatakse erinevate objektide genereerimiseks, mängumaailma genereerimiseks jne [13]. Kõige lihtsamaks näiteks võib tuua labürindi genereerimise, mis kuulub mängumaailma genereerimise hulka, antud leheküljel [14] on võimalik genereerida erineva suurusega labürinte ning peale igat genereerimist on võimalik genereeritud labürinti mängida. Kui labürindi suuruseks valida kaks korda kaks, siis algoritm ei ole suuteline rohkem kui kahte erinevat varianti pakkuda Joonis 1. Selline olukord juhtub, kui algoritmile antud parameetrid on ebapiisavad, et algoritm suudaks genereerida unikaalset mängimismaailma algoritmil, on vaja anda piisavalt parameetreid. Igal *PCG*l on olemas enda miinimum ja maksimum-parameetrite arv, mille abil algoritm on suuteline genereerima asju, kui minna alla miinimumi, ei ole algoritm suuteline genereerima talle ette antud ülesannet, mille tulemuseks on mittetöötav algoritm ning kui anda liiga palju parameetreid, siis algoritmil võib võtta väga palju aega, et jõuda lõpptulemuseni. Suuremate parameetritega algoritm hakkab kasutama väga intensiivselt arvuti ressursse, mille käigus arvuti ei jaksa antud ülesannet täita - seega on vaja leida ideaalne parameetrite arv, mille abil algoritm suudaks genereerida talle antud ülesannet võimalikult efektiivselt, kasutades võimalikult vähe arvuti ressursse.

Igavate, ühekülgsete mängude mängimisel ei tunne mängija rõõmu, isegi kui iga uue mängu alguses antakse talle uus mängumaailm, näiteks enne mainitud labürindimäng. Kui mängu lõpus mängijal on tunne, et ta raiskas aega ega saavutanud mingit tulemust, toob see kaasa soovi vahetada mäng välja, loobuda sellest, lisaks tõuseb ka mängija stressitase. Kui aga labürint oleks kolme- mõõtmeline ehk 3D [15], tõstaks see mängu kvaliteeti ja nii saaks mängu mitut korda mängida. Kui lisada veel elemente juurde, on võimalik teha tavalisest labürindimängust suur ja köitev mäng, mis hoiaks mängija huvi kõrgel ning tal oleks motivatsioon mängu jätkata.



Joonis 1. Kaks korda kaks labürindi genereerimise tulemus

PCG keerukuse tase seisneb kõnealuse mängu sisus, kus antud algoritmi kasutatakse, kui teha algoritm 2D ehk kahedimensiooniline mängumaailm, siis see on mitu korda lihtsam kui sama algoritmi teha 3D mängumaailma jaoks. *PCG*d kasutatakse kahel erineval viisil: esiteks kasutatakse *PCG*d mängu arendamise etapil, kus algoritmi ülesandeks on arendajate abistamine, andes neile uusi ideid või nende eest konkreetsete asjade tegemine, näiteks genereerides mängumaailma malli. Üheks näiteks võib tuua mängu *Hotline Miami*, kus arendamise käigus loodi algoritm, mis oli suuteline genereerima erinevaid mängumaailmu sõltuvalt ette antud parameetritest. Algoritm genereeris mängumaailma malli, kus arendajad ja disainerid tegelesid genereeritud maailmaga, muutes olemasolevat maailma ning lisades vajalikke elemente juurde [16]. Selline meetod hoiab arendajate aega kokku ning arendajad saavad keskenduda mängumaailma sisule. Teiseks *PCG*d kasutatakse mängu kestmisel ehk luuakse algoritmide kogum, mis käivitub mängija arvutis, kus algoritmi tegemine on oluliselt keerukam ja ajakulukam. Sellisel meetodil on suured riskid, kuna testijad ei ole suutelised kõiki variatsioone läbi proovima, mida algoritm pakub. Mäng, mida käesoleva töö autor üritab luua, kuulub teise tüüpi ehk *PCG* mängu kasutamisel, antud algoritmi on vaja teha nullist, kuna olemasolevad, mida saab osta või kasutada tasuta, ei ole piisavalt paindlikud kõnealuse projekti raames ning ise tehtud algoritm annab mängule unikaalsuse ja uudsuse.

2.1. *PCG* arendamise etapil

PCG arendamise etapp tähendab seda, et *PCG*d kasutatakse ettevõte siseselt, kus ühe *PCG* algoritmi võib kasutusele võtta mitme erineva projekti juures. Algoritmi tegemine on ressursikulukas, mis ei ole otstarbekas paljudele ettevõtetele, kuna maalimas on peamiselt kaks ettevõttet mängude tegemiseks: ühed neist on suurettevõtted, kes teevad *AAA* mänge, ja teised *indie* ettevõtted. Suurettevõtted saavad teha kindla *PCG*, mis oleks kasulik mitmes projektis, aga *indie* ettevõttele seda tüüpi *PCG* tegemine ei too suurt kasumit, kuna need mängud, mida *indie* ettevõtted teevad, on ühekordsed ning väikse mahuga. Kuna *PCG* on suuteline genereerima lõpmatu palju asju, siis ilma mingi kontrollita algoritm kaotab oma kasulikkuse - seega algoritm peaks olema piirangutega, et saaks algoritmi kontrollida. Karmide piirangutega *PCG* muutub lihtsakoeliseks, kus algoritmil kaob võimekus genereerida unikaalseid asju, kuna *PCG* hakkab genereerima sama asja iga kord uuesti. Teisalt, kui teha liiga väikse kontrollimisega algoritm, hakkab *PCG* genereerima selliseid asju, mis pole vajalikud antud kontekstis. Hea *PCG* on selline, kus on leitud sweet spot ehk kuldne koht, kus algoritm on vabas olekus ehk teeb, mida tahab ning piirangud on ainult selleks vajalikud, et piirata seda, mida kasutaja ei soovi ega vaja lõpptulemuses [17].

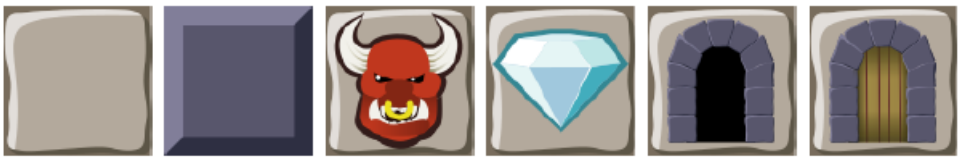
2.2. *PCG* mängu kestumisel

Teiseks *PCG*d kasutatakse mängu kestmise ajal, kus mängijale genereeritakse mängumaailm reaalajas sõltuvalt mängija tegevustest või iga uue mängu alguses teatud parameetrite järgi. Tänu sellele, et mängumaailma genereeritakse iga kord uuesti, kui hakatakse mängima, peab mängija iga kord kasutama uusi strateegiaid, mis hoiavad mängijat mängus, ilma et ta peaks uue mängu ostma. Selline lähenemine võimaldab mängu mitmeid kordi uuesti mängida, ilma et tekiks tunne, et mäng on üksluine. Kahjuks seda tüüpi *PCG*d ei kasutata nii tihti, kuna mängul on olemas oma lood ja need on tihedalt seotud mängumaailmaga, kus mängija viibib suure osa. Igakordse mängumaailma genereerimisega peaks välja mõtlema uue loo, mis on omaette raske ülesanne. Tavaliselt kasutavad seda tüüpi võimalusi *indie* ettevõtted, kuna ühe algoritmiga saab suure osa mängust valmis ega pea põhjalikult lugusid välja mõtlema, piisab ühest globaalsest loost, mis ei sõltu mängumaailmast. Sellised mängud on suunatud eelkõige mängu elementidele, mitte loo rääkimisele. Näiteks võib tuua mängu *The Binding of Isaac*, kus iga uut mängu alustades genereeritakse mängijale uus mängumaailm ja sõltumata maailmast, jääb põhilugu paika, kuna mängu lugu on väga lihtne. Selles mängus genereeritakse 2D mängumaailm, kus maailm koosneb ristkülikulistest ruumidest Joonis 2, kus iga ruum on oma suuruse ja sisuga, *PCG* algoritm selles mängus on väga paindlik, kuna mängija tegevused mõjutavad algoritmi edaspidist tööd.

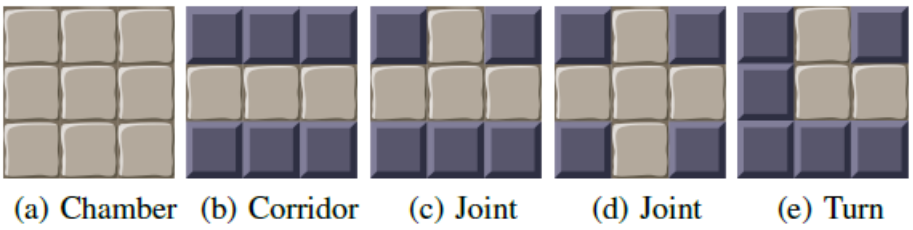
Joonis 2. Hetktõmmis mängust *The Binding of Isaac* [1]

2.2.1. 2D mängumaailmad

2D ja 3D mängumaailmade genereerimine ei erine *PCG* algoritmi struktuuri poolest, vaid algoritmi keerukusest, kus põhiliseks erinevuseks on dimensioonide arv. Mängumaailma genereerimisel *PCG* abiga kasutatakse erinevaid komponente, mille alusel algoritm hakkab mängumaailma genereerima, põhilisteks komponentideks on põranda- ja seinaobjektide kombinatsioon, mis on ette defineeritud eraldi objektidena Joonis 3. Joonisel 4 on näha viis erinevat kombinatsiooni enne mainitud objektidest, rohkem kui viite kombinatsiooni algoritmil pole vaja, kuna algoritm on suuteline antud komponente pöörama, näiteks komponent (b) on käik vasakult paremale, aga kui algoritmil oleks vaja käiku ülevalt alla, siis algoritm pöörab antud komponenti 90 kraadi ning sama saab teha teiste komponentidega, pöörates neid ette defineeritud kraadi võrra. Pööramisnurk on algoritmi sisse kirjutatud, tavaliselt 2D mängumaailmades on antud nurk 90 kraadi. Kui vaadata uuesti objekte Joonis 3, siis seal on veel lisaobjektid, mida ei ole kasutatud, need lisa- objektid lisanduvad mängumaailma peale genereerimist, lisanduvate objektide arv ja paigutus maailmas on sõltuvuses algoritmist, seotud algoritmiga.



Joonis 3. 2D mängumaailma parameetrid [2]

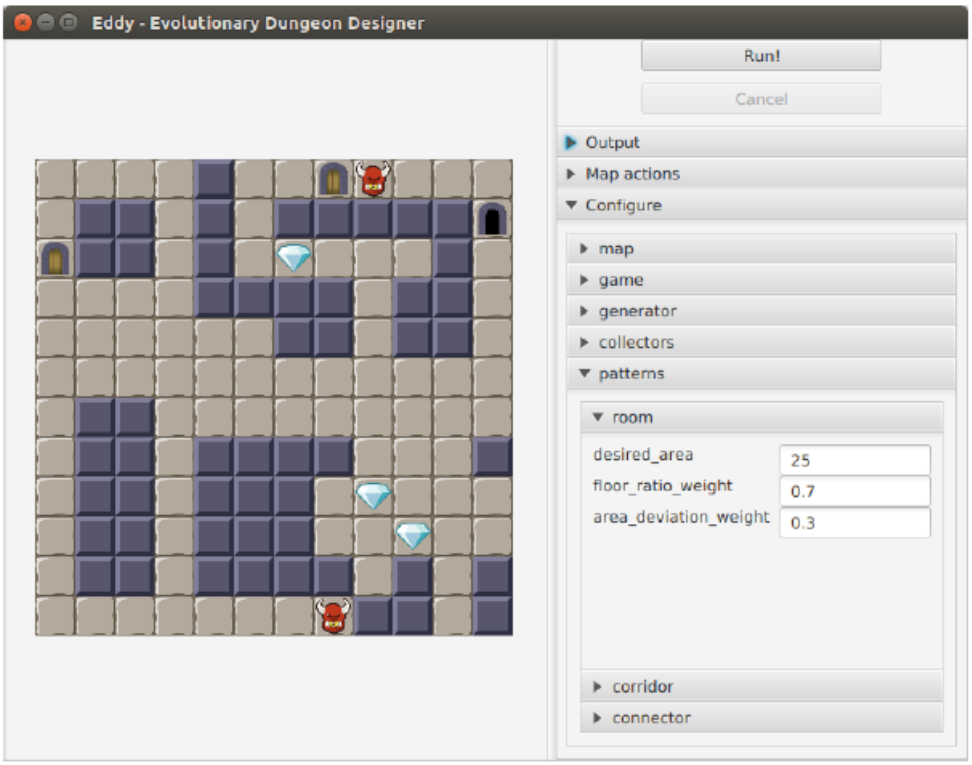


Joonis 4. 2D mängumaailma parameetrite kombinatsioon [2]

*PCG*l on olemas kasutajaliides, mida kasutatakse arendamises, kus saab seadistada erinevaid piiranguid ja parameetreid. Joonisel 5 on näha liides, kus parempoolsel ekraanil on parameetrite ning piirangute seadistus ja vasakul poolel genereeritud mängumaailma kohene tulemus. Kasutajaliides on vajalik mitte ainult välimuse pärast, vaid see on oluline ka testimise lihtsustamiseks, kus testijal on kogu aeg testitavad parameetrid silma ees, samuti ka genereeritud mängumaailm. Tänu sellele saavad nii testijad kui ka arendajad lihtsalt ja kiiresti teha mingeid parandusi suuremat vaeva nägemata. Selline kasutajaliides on kasutusel mängu arendamisel, mängijale aga näidatakse ainult lõpptulemust, kus parameetrid on ette seadistatud ning nende muutmine on võimalik ainult läbi mänguelementide, kus mängija ise uusi parameetreid ei sisesta, vaid parameetrid muutuvad mängu ajal sõltuvalt mängija tegevusest. Lihtsamate mängumaailmade genereerimisel ei kasutata dünaamilisi parameetreid, mis võimaldavad mängu ajal parameetreid muuta, vaid parameetrid on kogu aeg konstantsed, mis võivad viia samade mängumaailmade genereerimiseni ning mängija meeleolu langemiseni.

Mängumaailmade genereerimise käigus võib tekkida palju ootamatuid asju, mida *PCG* peaks olema suuteline parandama. Peamiseks probleemiks on komponentide omavahelised kattumised. Kattumisi võib lahendada erinevatel viisidel, kus antud näites [2] kasutatakse kattuvuste lahendamiseks liitumise ja lõikamise meetodit, see tähendab seda, et kui komponendil tekkib kattuvus, siis komponent lisatakse teise komponendi peale, aga see on võimalik ainult siis, kui komponentides kasutatud objektid on samasugused. Komponendi

väljumisel mängumaailma piirest lõigatakse antud komponendi osa välja, näiteks Joonis 6. Probleemideks võib olla ka lisaobjektide paigutus maailma, ilma et nad oleksid liiga tihedalt paigutatud ning objektide kättesaadavus mängijale piiratud, kui näiteks maailmas on väljapääs kohas, kuhu mängija ei saa minna, siis mäng ei ole mängitav.

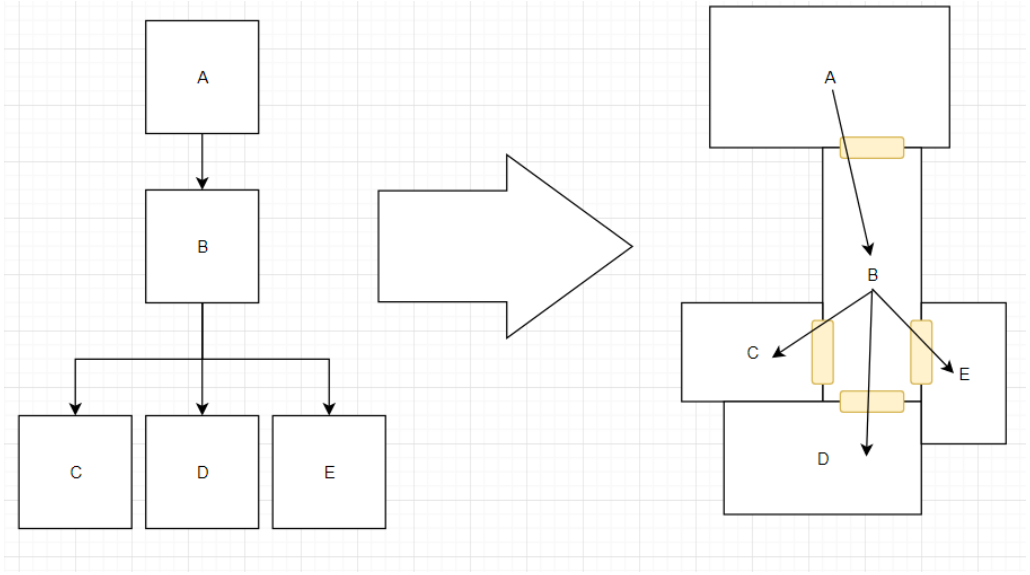


Joonis 5. 2D mängumaailma kasutaja liides [2]

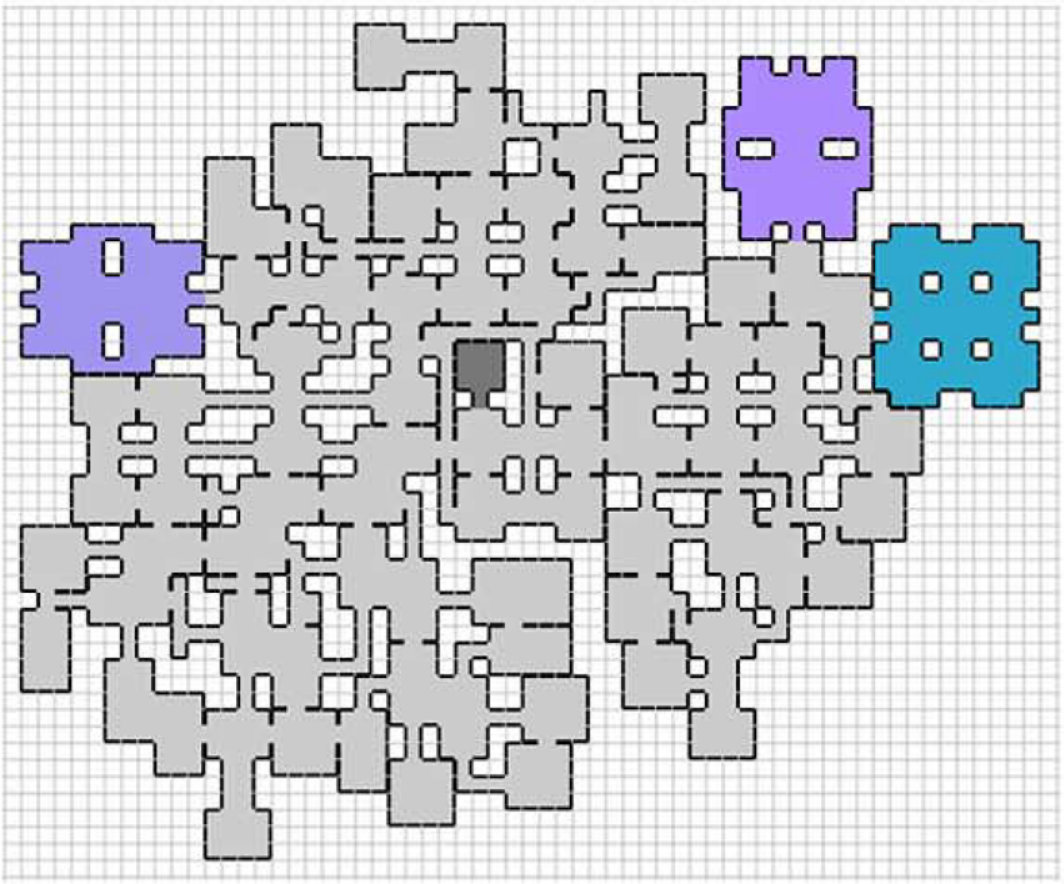
Joonis 6. 2D mängumaailma komponentide liitmine [2]

2.2.2. 3D mängumaailmad

Nagu enne mainitud, ei erine 3D mängumaailmade genereerimine 2D mängumaailmade genereerimisest, kus 3D mängumaailmadel on lisadimensioon, mis teeb *PCG* loomise ja täitmise, kasutamise raskemaks. 3D mängumaailma genereerimiseks on olemas palju erinevaid viise, näiteks alguses genereeritakse 3D graaf ja graafi genereerimise lõpus asendatakse graafi komponendid läbikäikudega, teiseks on plokkide tihe paiknemine mängumaailmas, kus plokk on sama asi, mis olid 2D mängumaailma komponendid, aga plokid on natukene keerukama struktuuriga kui 2D komponendid [3]. Antud töös pakkus huvi plokkide tihedalt paiknemise meetod, mis võimaldab geenreerida tihedat mängumaailma, kus iga plokk saaks olla erineva kuju ning sisuga. Igal plokil on olemas kindlad kohad, kus plokk saab ühenduse teiste plokkidega Joonis 7. Taolise algoritmi on varem teinud *dungeon crawler levels Valtchanov* ja *Brown* [18], tuginedes sellele ideele teha oma algoritm. Näiteks võib tuua mängu *Paranautical Activity* [19], kus on kasutusel kõnealune tehnoloogia, aga selles mängus on plokkide sisu ja mängumaailma genereerimine väga lihtsa struktuuriga, kus iga plokk on sama suurusega ja ploki sisu on ühekülgne.



Joonis 7. 3D mängumaailma plokkide omavahel ühendamine [3]



Joonis 8. 3D mängumaailma pealt vaade [3]

2.3. *PCG* ohud

PCG ei too endaga kaasa ainult positiivset, kahjuks esinevad ka mõningad negatiivsed küljed, mis mõnele ettevõttele ei ole vastuvõetavad. Peamiseks põhjuseks on ebaturvalisus, kuna automaattestid või testijad ei suuda kõiki algoritmi poolt genereeritut asju testida ning mõnel üksikul juhul võib tekkida kriitiline viga, mille pärast mängu ei saa enam mängida või väga halvas olukorras rikub mängija arvuti. Sellist riski ei taheta võtta, isegi kui see lihtsustab edaspidist tööd ja teeb mängu meeldivamaks [20]. Võib küsida, miks on vaja *PCG*d, kui seda keegi ei kasuta, siin tulevad hea näitena *indie* ettevõtted, kes võtavad selliseid riske, et kuidagi saaks konkureerida *AAA* mängudega. Kui *PCG* õnnestub, võib see mängu teha väga populaarseks ja anda ettevõttele suure kasumi, ühesõnaga võib öelda, et *PCG* on high risk high reward ehk kõrge riskiga ning kõrge kasumiga. Näiteks võib tuua mängu *No Man's Sky* Joonis 9, kus terve mäng on sõltuvuses *PCG*st ning iga asi seal on genereeritud, alustades keskkonnast ninglõpetades loomadega [21], ettevõte kulutas palju

ressursse *PCG* tegemiseks, kuid ei kulutanud piisavalt aega teistele funktsionaalsustele. Kasutades ainult *PCG*d, ei ole võimalik teha edukat mängu, nad üritasid ja ebaõnnestusid [22].

Joonis 9. Hetk tõmmis *No Man's Sky* mängust

3. Töökeskkond

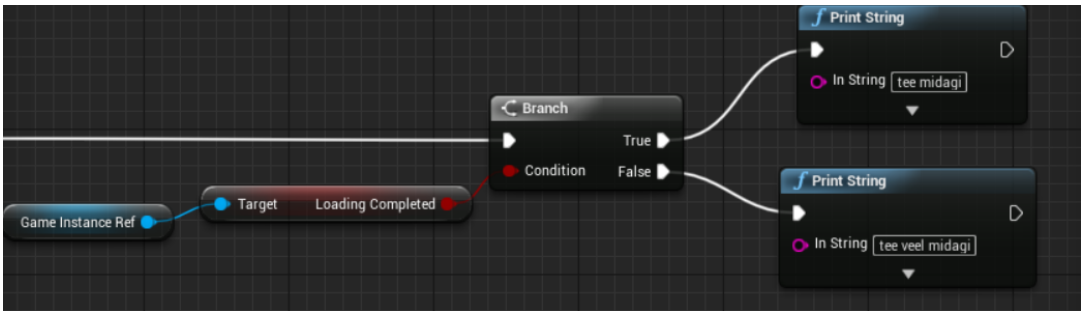
Mänge saab teha igas keeles, alustades *baisic* ja lõpetades *malbolge* [23], kuna mäng ei ole midagi rohkemat kui funktsioonide väljakutsumine ning tulemuste näitamine mängijale läbi kasutajaliidese. Praegu on olemas programmid, mis on suunatud mängude tegemiseks ning selles projektis kasutati *Unreal engine* [24]. Selliseid programme nimetatakse *game engine* ehk mängu mootor, mis on kirjutatud mingis programmeerimiskeeles ja mille juurde on tehtud kasutajaliides, et oleks lihtsam kasutada. Valisin *Unreal engine* mitte ainult sellepärast, et see on kõige populaarsem võrreldes teiste mängumootoritega [25], vaid ka huvi graafilise programeerimise tõttu, ilma et oleks vaja käsitsi koodi kirjutada ja selleks on *Unreal engine*l olemas *blueprints*. Aga ainult mängu mootorist ei ole võimalik mängu valmis teha, kuna antud mängumootor ei võimalda teha 3D objekte, heli ja muud mängule vajalikke asju, selleks on olemas teised programmid, mis ühilduvad mängu mootoriga. 3D objektide tegemiseks on eraldi programmid [26], mille projektis kasutati blenderit, seda programmi on nii autor kui ka disainer kasutanud mitmeid aastaid enne selle projektiga alustamist, seega suurt valikut ei olnud, aga ainult 3D objektide tegemisest ei ole kasu. 3D objekte oleks vaja illustreerida ja selleks on olemas oma eraldi programmid 3D texture painting softwares ehk 3D objektide illustreerimise programm [27]. Selles projektis võeti kasutusele *Substance Painter*, selle valik otsustus selle populaarsuse, kuna seda oli kasutatud mõnes *AAA* mängus, näiteks *Ghost Recon: Wildlands* ja paljud teised mängud [28]. Nimekirju programmidest, mis on vaja, et teha ilusat ja edukat mängu, on palju, aga enne loetletud programmid on selle projekti ulatuses kasutust leidnud

3.1. *Blueprints*

Blueprints võib algatajatele tunduda raske, kuna see on üks vähimatest programmeerimiskeeltest, kus ei pea käsitsi midagi kirjutama, vaid saab kasutada *drag and drop* ehk tõmba ja lase lahti meetodit. *Blueprints* kasutab muutujate jaoks erinevaid värve, nagu näiteks punane on jah/ei väärtused, sinine on objektide viitamine, lilla on teksti väljadeks ning kõige olulisem värv on valge, mis tähistab koodi voolu, kus koodi täidetakse vasakult paremale ning palju teisi värve. Võrdluseks võib tuua java programmeerimiskeelega Joonis 10, mõlemad funktsioonid on samasugused, kus *blueprintis* on parem ja lihtsam aru saada, mida antud funktsioon tee, kui tekstilises programmeerimiskeeles

Programmeerija kulutab suure osa oma tööst mitte koodi kirjutamisele, vaid sealt vigade

otsimisele ja nende parandamisele ehk debugging, kus mingi lihtsa vea leidmine võib võtta mõned tunnid,näiteks, kui mingis võrdluses on vale väärtus. *Blueprints* testimist on näha visuaalselt, kus ja millist teed pidi kood parajasti liigub Joonis 11, testimise käigus saab ka muutujate väärtusi jooksvalt vaadata, ei pea väärtusi välja trükkima kasutaja liidessesse. Nii tekstilises kui ka *blueprints* on võimalik lisada breakpoint, mis peatub koodi teatud kohas, mille järel saab uurida koodi seisundit, aga *blueprints* saab reaalajas lihtsasti jälgida koodi seisundit. Lisaks lihtsale testimisele *blueprints* saab kiiresti genereerida suurest koodi jupist väikse funktsiooni Joonis 12, mida saab edaspidi erinevates kohtades kasutada.

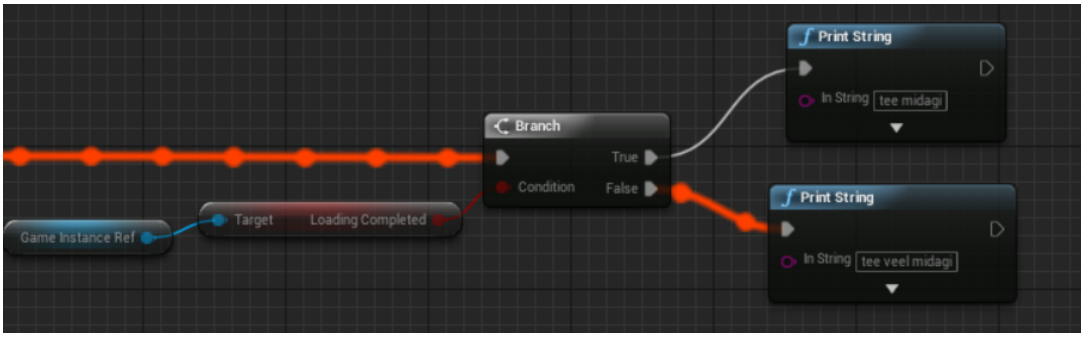


(a)

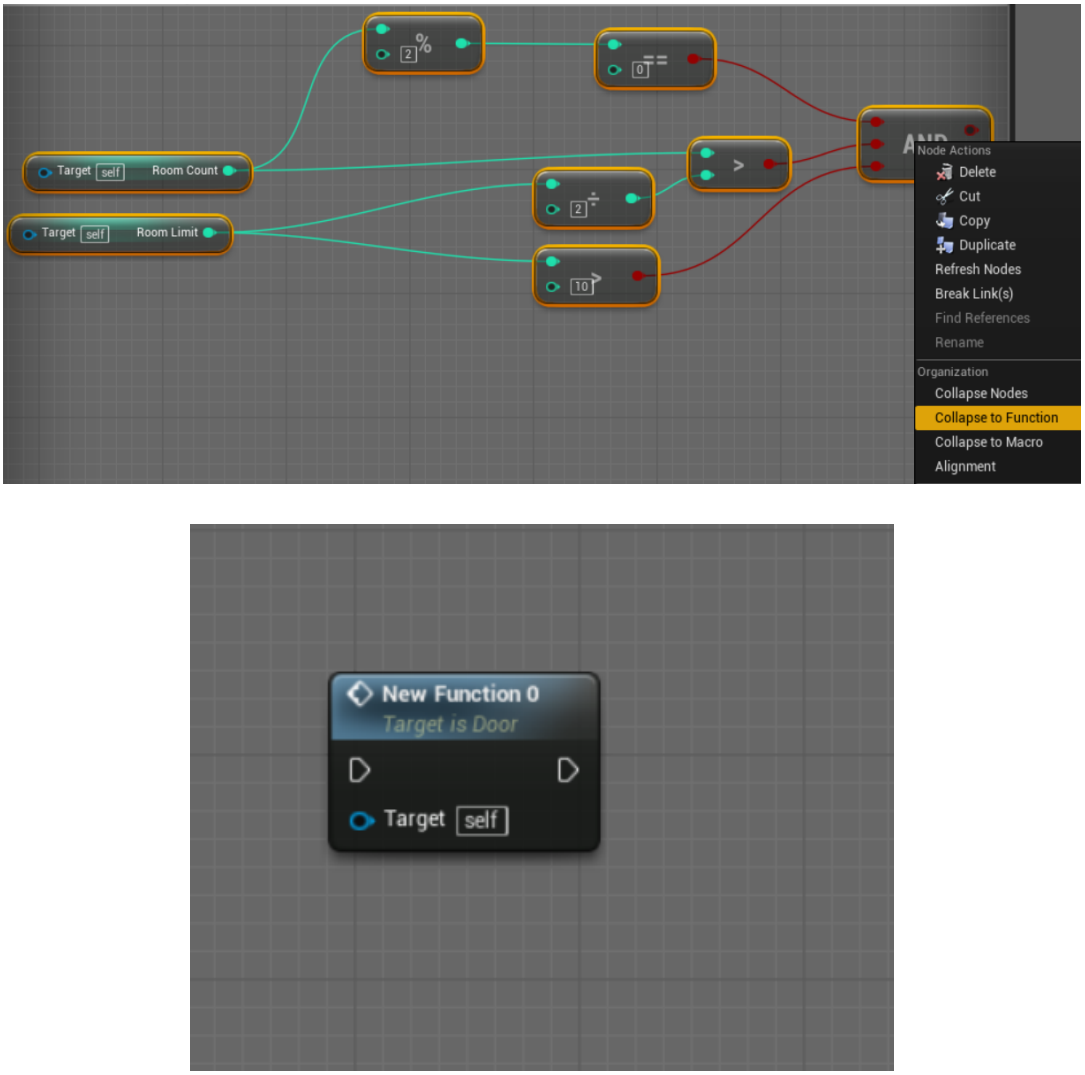
```
if (gameInstanceRef.get(loadingCompleted)){
Systeom.out.print("tee midagi");
} else {
Systeom.out.print("tee veel midagi");
}
```

(b)

Joonis 10. Kahe programmeerimis keele võrdlemine: (a) *blueprints*, (b) java



Joonis 11. Koodi testimine *blueprints*



Joonis 12. *Blueprints* koodi kompaktseks tegemine

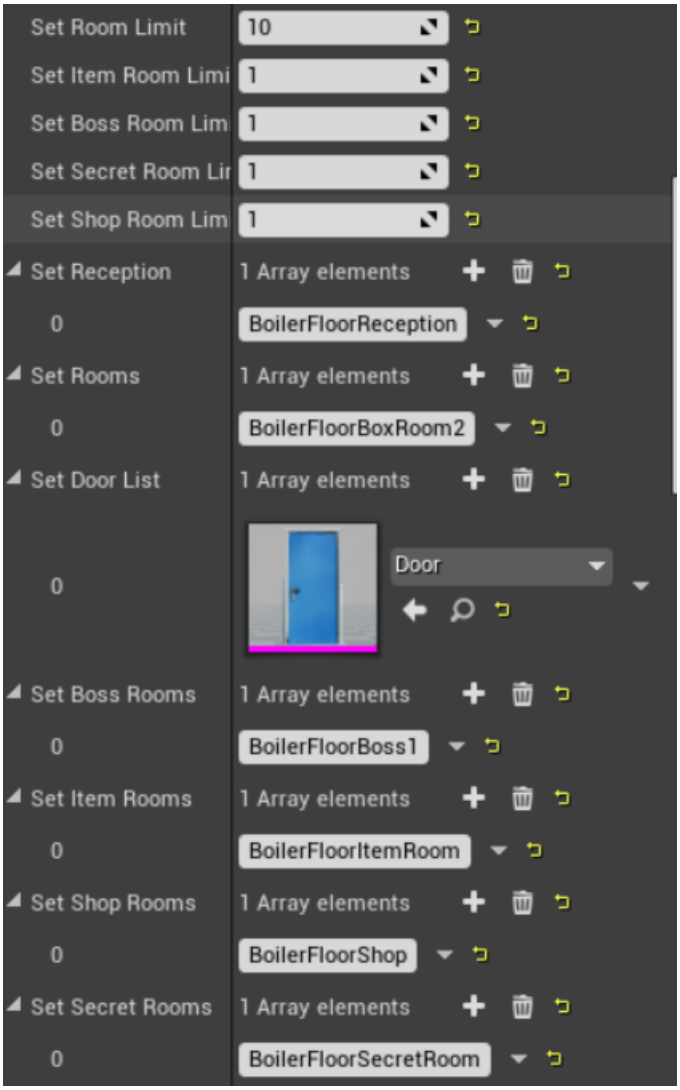
3.2. Teised mängumootorid

Unreal engine ei ole ainuke mängu mootor, mida oleks võimalik kasutada selles projektis. Kõige suurem konkurent *Unreal engine*ile on *Unity* [29], millele on samuti palju võimalusi. Nende kahe mängumootori põhilisteks erinevusteks on nende programmeerimiskeel ning sealt väljendub mängu mootori raskustase. *Unity* on lihtsam õppida kui *Unreal engine*it, kuna C++ programmeerimiskeel on raskem kui C# programmeerimiskeel, mida *Unity* kasutab [30]. Aga *Unreal engine*it teeb lihtsamaks tema graafiline programmeerimine *blueprints*, mis võib alguses tunduda raske, aga peale selle õppimist on raske sellest lahkuda, see soodustab nii koodi kirjutamist kui ka selle testimist, see on lihtsam kui käsitsi kirjutatud kood. Teisest küljest, kui on vaja internetist mingit koodi osa kopeerida,

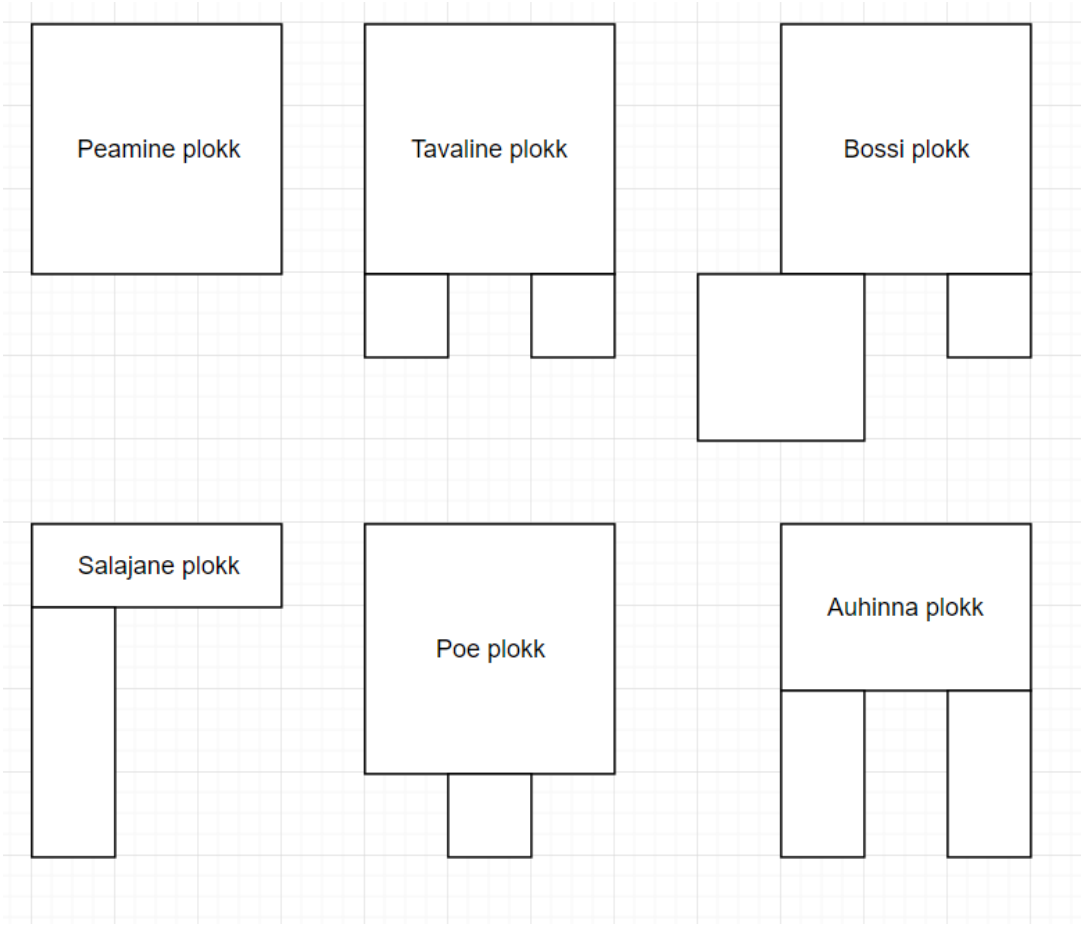
siis *blueprintis* ei ole seda võimalik teha, peab käsitsi pildilt maha tegema, mis on päris tüütu suurte funktsioonide juures, kus aga käsitsi kirjutatud koodi on võimalik internetist kopeerida ja lisada oma programmi. Maailmas on olemas ka palju teisi mängumootoreid [31], aga nende võimalused ei ole piisavalt suured, mis võimaldaks autori ambitsioone täita.

4. Töös tehtud algoritm

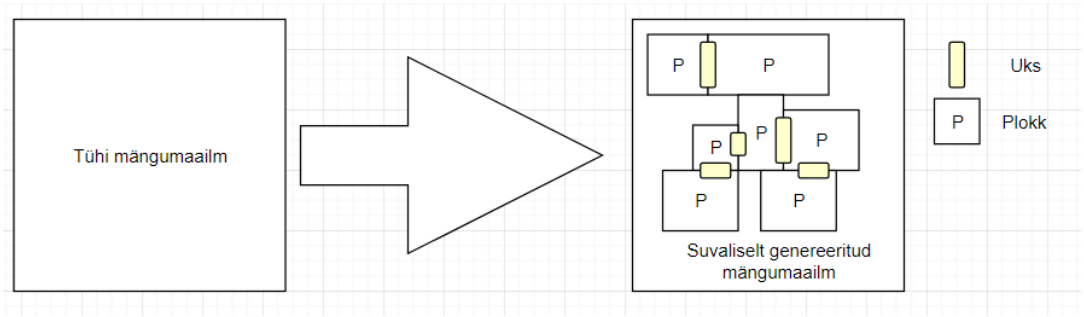
Tehtud *PCG* ülesandeks on genereerida suvalist unikaalset mängumaailma, kasutades eelnevalt määratletud parameetreid Joonisel 13, kus *set reception* ehk peamine plokk, *set rooms* ehk tavaline plokk, *set boss rooms* ehk bossi plokk, *set item rooms* ehk auhinna plokk, *set shop rooms* ehk poe plokk ja *set secret rooms* ehk salajane plokk, iga nimekiri hoiab endas kasutatavate plokkide nimesid, kust algoritm hakkab võtma plokke ja lisama mägumaailma, kasutatud plokkide struktuurid on Joonisel 14. Enne nimetatud plokid jagunevad kolmeks erinevaks tüübiks, kus peamise ploki algoritm kasutatakse ainult üks kord ning see plokk on mängija alguskohaks iga uue mängu alguses, teiseks tüübiks on tavaline plokk, mida algoritm kasutab kõige rohkem ning nendes plokkides hakkab mängija veetma suure osa mänguajast, viimaseks on bossi, auhinna, poe ja salajane plokk, mis kuuluvad spetsiaalsete plokkide hulka ja need plokkid autasustavad mängijat, kui ta leiab need mängumaailmas. *Set room limit* ehk tavalise ploki limiit, *set item room limit* ehk auhinna ploki limiit, *set boss room limit* ehk bossi ploki limiit, *set secret room limit* salajase ploki limiit, *set shop room limit* ehk poe ploki limiit. Enne mainitud väärtused sõltuvad mängumaailma lisanduvate plokkide arvust, neid limiite on vaja selle jaoks, et algoritmil oleks mingi lõpp-punkt ja suudaks oma tööd lõpetada, kui neid limiite poleks, siis algoritm ei lõpetaks kunagi oma tööd. Viimaseks on *set door list*, mis on nimekiri uste välimusest, antud nimekiri võimaldab mängu ajal ukse välimust muuta. Nende parameetritega on oodatud, et *PCG* suudab tühjast mängumaailmast genereerida täpselt talle ette antud parameetritest koosneva mängumaailma Joonisel 15.



Joonis 13. Kasutaja liides arendamise keskkonnas parameetrite määramiseks



Joonis 14. Töös kasutatud plokkide struktuur

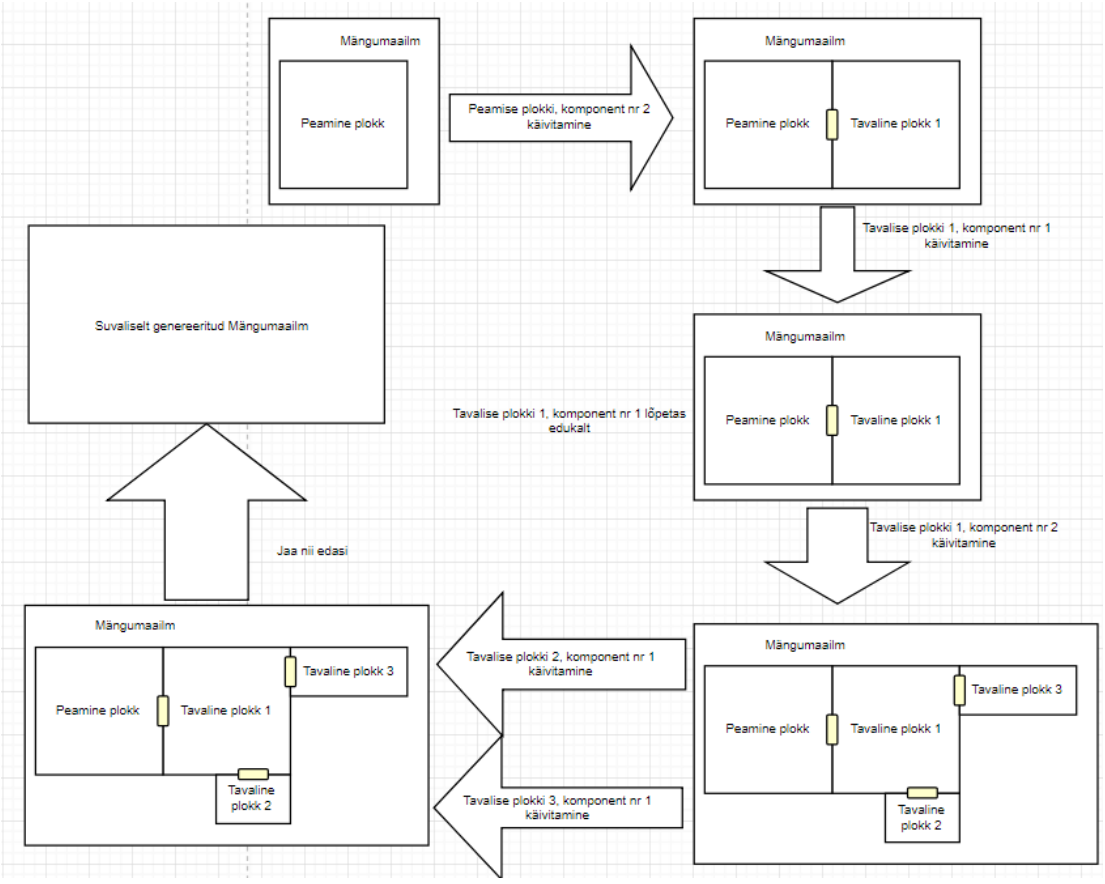


Joonis 15. *PCG* poolt oodatav tulemus

Tehtud *PCG* algoritm on hajutatud kõikide plokkide vahel, kus igal plokil on oma algoritm ning ploki algoritm on jaotatud kaheks komponendiks, esimene komponent vastutab ploki ümbruse eest ehk kontrollib kattuvusi edaspidi komponent nr1 ning teine komponent vastutab uue ploki ilmumise eest mängumaailma, edaspidi komponent nr2. Igal plokil võib olla üks komponent nr1 ning kuni kolm komponenti nr2, komponendid on suutelised kommunikeeruma nii ploki siseselt kui ka eelmise ploki komponentidega, andes teada

tekkinud probleemidest. *PCG* on olemas üks erandjuht, mis asub peamisel plokil, millel sisaldub ainult komponent nr2. Põhiliseks kommunikatsioonimeetodiks on teatamine kattuvuste kohta, mängija plokist sees ja väljas viibimisest ning ploki edukast ilmumisest mängumaailma. Kõikidel komponentidel on olemas ligipääs mängu olekule, kus hoitakse vajalikku informatsiooni mängumaailma kohta, mängu olek sisaldab endas lisatud plokkide arvu ning millisesse tüüpi lisatud plokid kuuluvad.

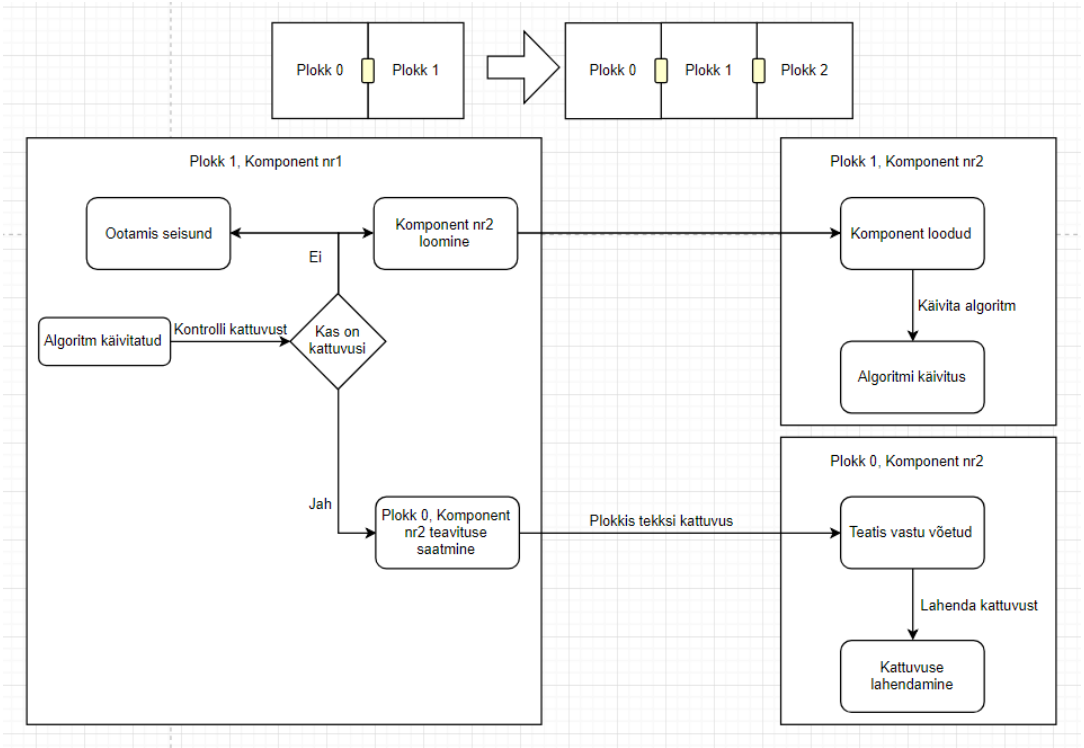
Iga uue mängu alguses tekkib ahelreaktsioon plokkide algoritmidest Joonis 16, kus peale iga õnnestunud ploki lisamist mängumaailma luuakse ja käivitatakse uus algoritm ning see toimub niikaua, kuni jõutakse antud piirangutesse. Mängumaailmas võib töötada samaaegselt kuni kahe plokki algoritm, see võimaldab mängumaailma genereerimist kiirendada, aga kui lubada rohkem kui kahe ploki algoritmi, siis see võtab arvutilt liiga palju ressursse, mis teeb algoritmi katki ning algoritm ei ole suuteline enam töötama. Iga komponent luuakse ja käivitatakse erinevates olukordades, kus komponent nr1 luuakse samal ajal, kui lisatakse uus plokk mängumaailma ning selle komponendid käivitatakse kohe peale lisamist. Komponendi nr2 loomine ja käivitamine toimub läbi komponent nr1, kus komponent nr1 lisab ja käivitab kuni kolm komponenti nr2.



Joonis 16. Algoritmi ahelreaktsioon

4.1. Plokkide mängumaailmasse lisamine

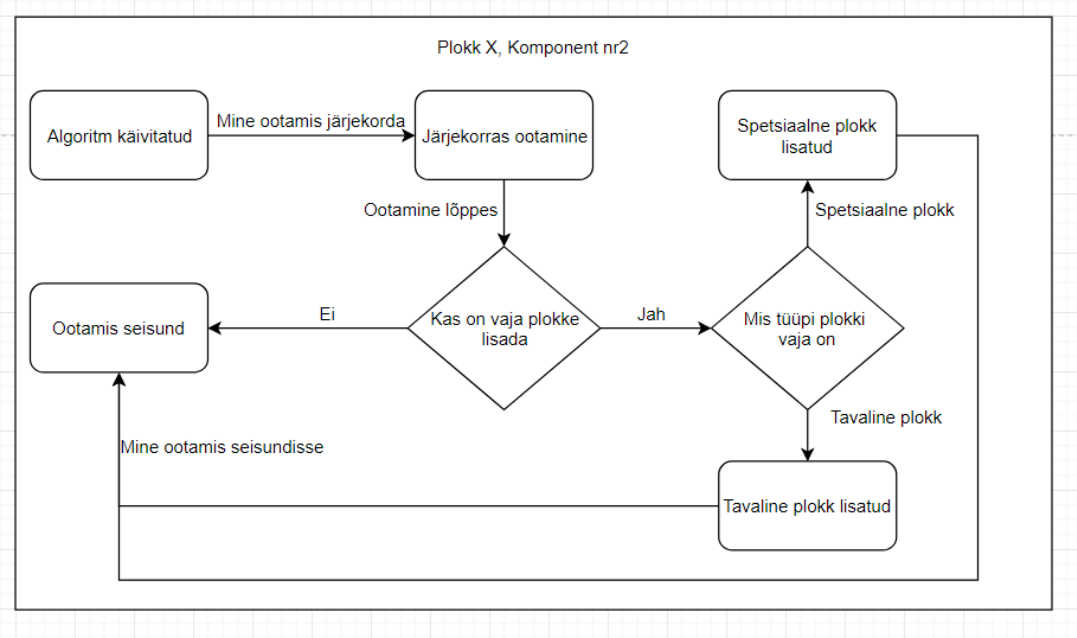
Mängumaailmas on seisund selline, kus plokk 0 ehk peamine plokk lõpetas oma algoritmi ning lisas mängumaailma plokk 1. Kohe peale plokk 1 lisamist mängumaailma tema algoritm käivitub. Esimesena läheb tööle komponent nr1, mis kontrollib ploki kattuvust tema ümbrusega, funktsioon, mida käivitatakse kattuvuste kontrolliks, käivitatakse kaks korda. Funktsiooni käivitamine kahel korral on vajalik selleks, et tagada kindlus kattuvuste tekkimisel, näiteks arendamise etapil tekkisid olukorrad, kus kattuvust ei registreeritud, aga pidi olema registreeritud, kui aga samad funktsiooni käivitati kaks korda, siis kattuvusi registreeriti igakordselt. Funktsoon ei registreerinud kattuvusi esimesel korral, kuna mängumaailmas on korraga töös mitu algoritmi, mis jagavad arvuti ressursse, mille tõttu võib juhtuda, et funktsiooni kattuvus jääb registreerimata. Kui kattumisi ei tuvastata, siis komponent nr1 loob ja käivitab teise algoritmi ning samal ajal läheb ootamisseisundisse. Kui kattuvus oli registreeritud, siis komponent nr1 annab sellest teada eelmise ploki komponendile nr2, kus lahendatakse tekkinud probleem Joonis 17, kattuvuste lahendamisest on seletatud peatükis 4.3.



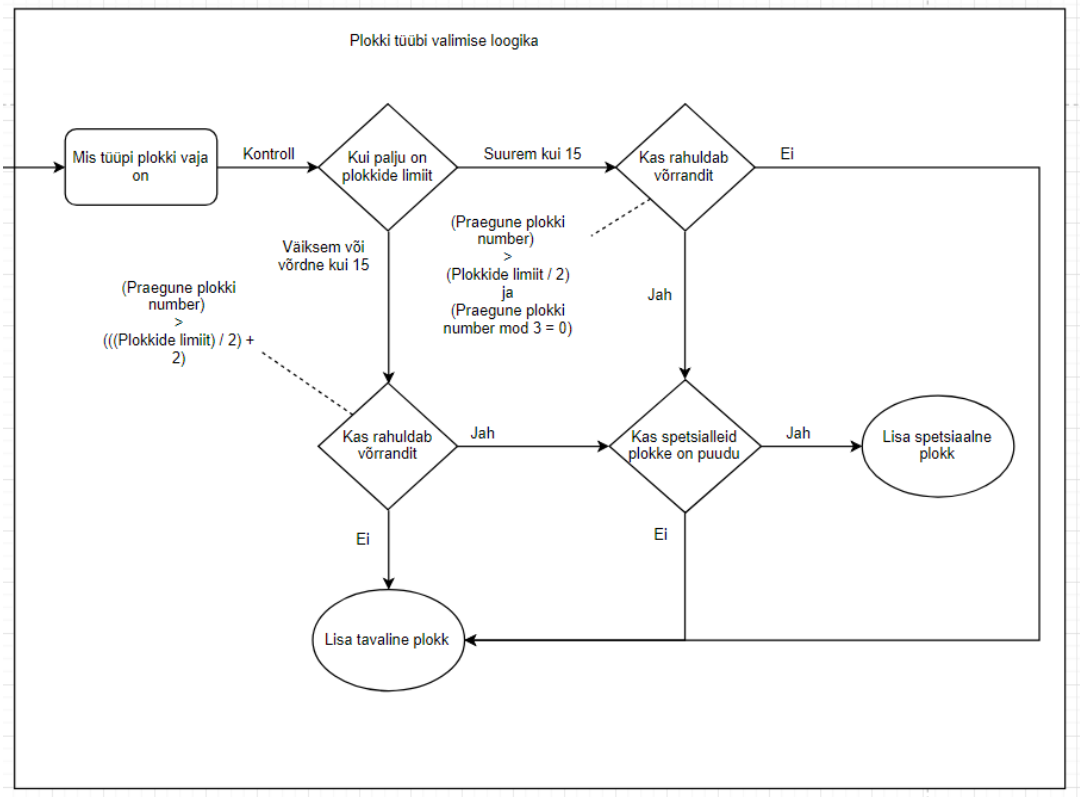
Joonis 17. Plokkide lisamine mängumaailmasse

4.1.1. Plokki tüübi valik

Komponent nr2 algoritmi käivitumisel läheb algoritm ootamisjärjekorda. Järjekord on vajalik selleks, et tagada ainult kahe algoritmi samaaegne töötamine mängumaailmas, täpsemalt on järjekorras ootamist kirjeldatud peatükis 4.2. Peale ootamist algoritm kontrollib mängu seisundit, vaadates, kas on vaja lisada plokke mängumaailma, kui ei ole, siis läheb ootamis- seisundisse, mis erineb järjekorra ootamisest. Kui mängumaailma on vaja lisada plokke, siis algoritm küsib uuesti mängu seisundit ja vaatab, mis tüüpi plokki on vaja järgmiseks lisada Joonis 18. Ploki tüübi valimine käib läbi kahe võrdluse: esiteks kontrollitakse mängumaailma ploki limiiti, kui plokkide limiit on väiksem või võrdne 15ga, siis tehakse järgmine võrratus: (praegune ploki number) > ((plokkide limiit / 2) + 2), selles võrratuses kontrollitakse, et praegune plokki number, mida tahetakse, oleks üle poole vajaliku plokkide arvust. Kui aga plokkide limiit on suurem kui 15, siis tehakse: (praegune ploki number) > (ploki limiit / 2) ja ((praegune ploki number) mod 3 = 0), selles võrratuses kontrollitakse praeguse ploki numbrit, mida tahetakse, et oleks üle poole vajaliku plokkide arvust ja et praeguse ploki number jaguks kolmega ehk jääk liige oleks null. Kolmega jagamine on vajalik, et teha spetsiaalsete plokkide lisamist mängumaailma natukene hõredamaks, mille tulemusel spetsiaalseid plokke lisatakse iga kolmanda plokina. Kui võrratused tagastasid positiivse väärtuse, tehakse viimane kontroll, mille järel otsustatakse ploki tüüp Joonis 19.



Joonis 18. Plokki tüübi valik

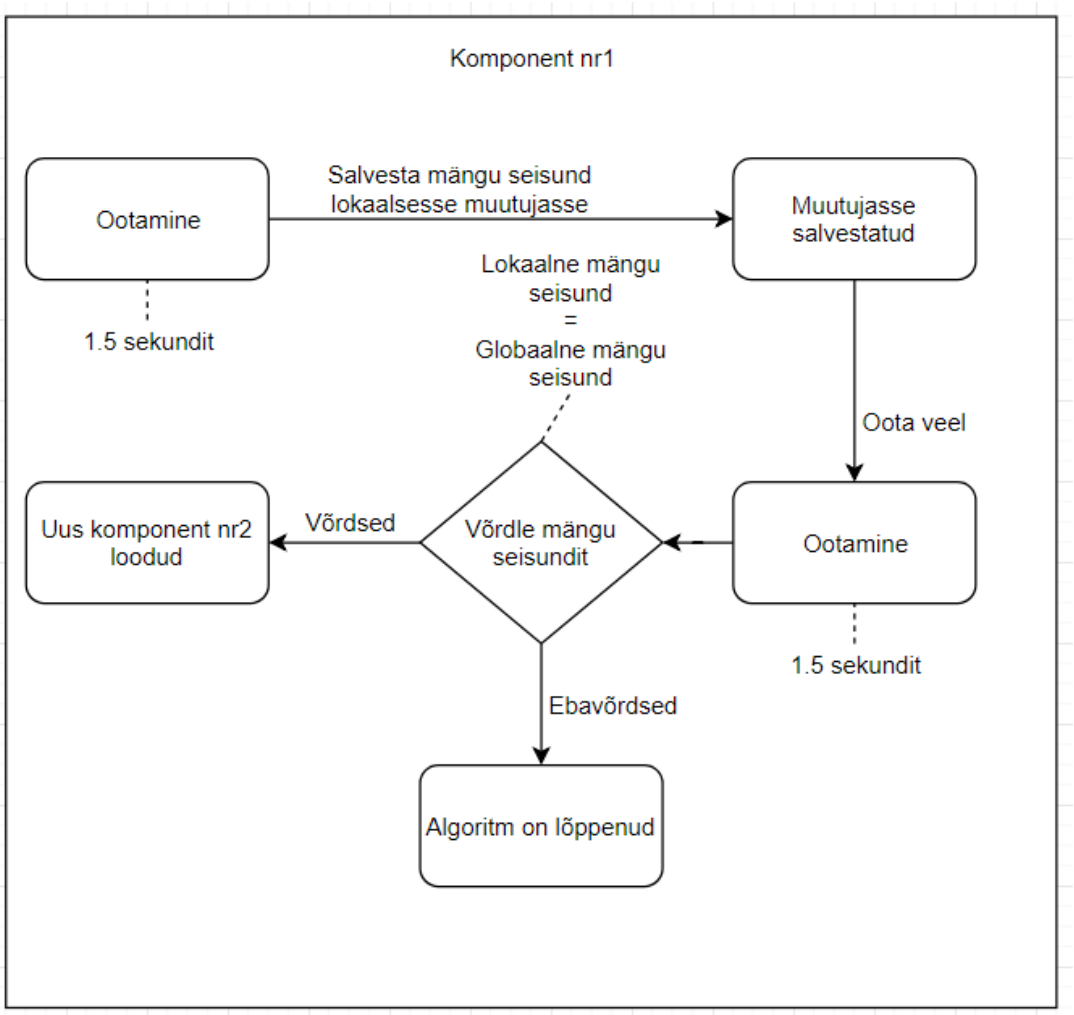


Joonis 19. Plokki tüübi valimise loogika

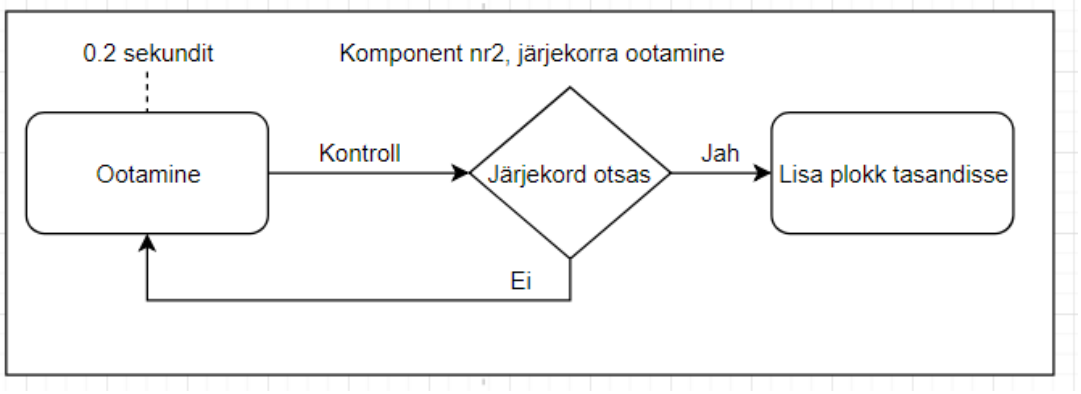
4.2. Algoritm ootamis seisund

PCG algoritmil on kokku kolm erinevat ootamisseisundit: komponendil nr1 on üks seisund ja komponendil nr2 on kaks erinevat seisundit. Komponent nr1 ootamisseisund on vajalik selle jaoks, et vältida *PCG* algoritmi seiskumist, kuna mõnikord juhtub, et ploki lisamine mängumaailma on võimatu, aga plokkide limiit ei ole täis - seega varasemad ploki algoritmid, mis on ootamisseisundis, peavad tekitama uue komponendi, et *PCG* saaks edasi töötada. Esimese komponendi ootamine võtab kokku kolm sekundit, kus esimese poole algoritm lihtsalt ootab ning peale 1,5 sekundit algoritm salvestab globaalse mängu seisundi lokaalsesse muutujasse, kus peale 1,5 sekundit ta võrdleb lokaalset ja globaalset seisundit ja kui nad on samad, siis see tähendab, et mängumaailma ei tekkinud uusi plokke ja on vaja lisakomponente Joonis 20, kui aga nad on erinevad, siis algoritm lõpetab oma töö. Teisel komponendil on vaja kaht erinevat ootamisseisundit: üks on vajalik järjekorra jälgimiseks ja teine genereerimise lõpetamise jälgimiseks. Nagu enne mainitud, suudab *PCG* mitut algoritmi samaaegselt jooksutada ja et tagada, et rohkem kui kaks algoritmi ei töötaks samaaegselt, on vaja hoida järjekorda Joonis 21, kus algoritm käib küsimas

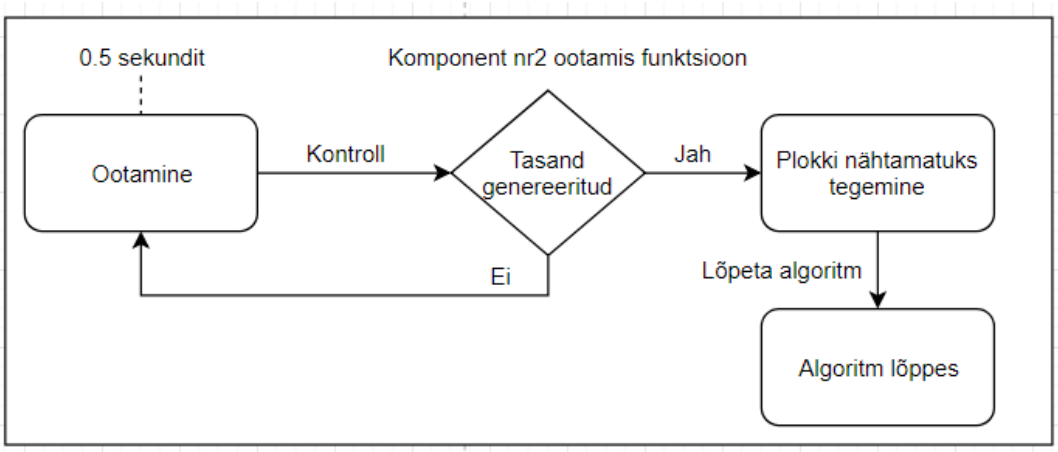
järjekorra seisundit iga 0,2 sekundi tagant. Teiseks on ootamine kuni mängumaailm on genereeritud, peale mida tehakse kõik plokid nähtamatuks Joonis 22, sellel on mängu seisundi küsimine iga poole sekundi tagant, mille lõpus algoritm lõpetab oma tegevuse. Nende kahe ootamisseisundite vahe on 0,3 sekundit, kuna algoritmil on olulisem pigem lisada mängumaailma plokke kui et neid nähtamatuks teha.



Joonis 20. Komponent nr1 ootamis funktsioon



Joonis 21. Komponent nr2 järjekorra ootamine

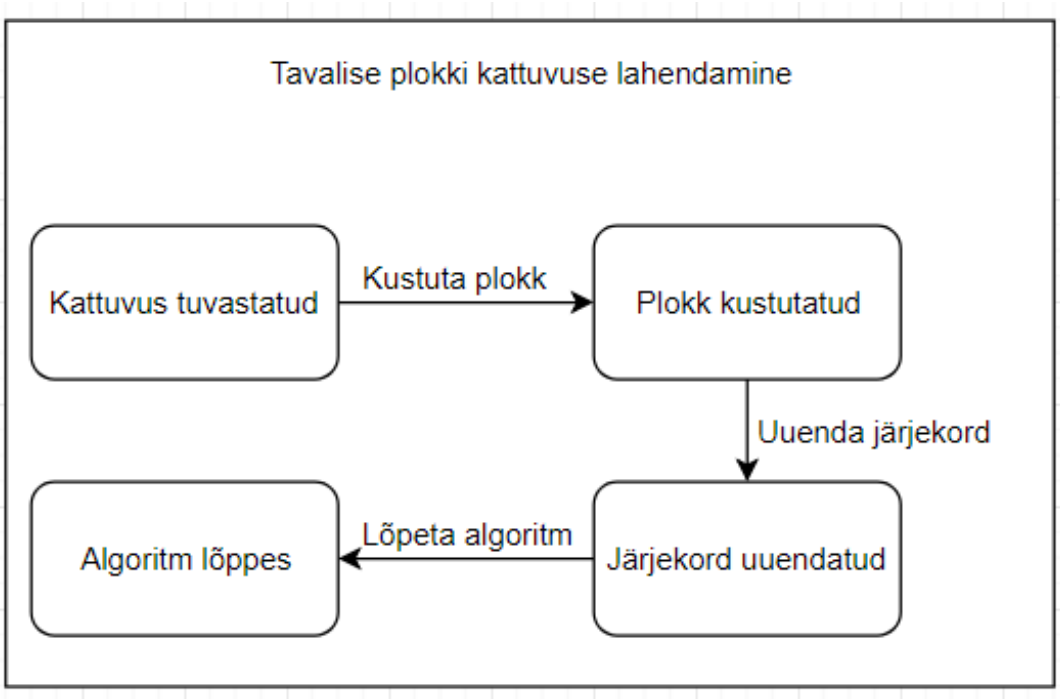


Joonis 22. Komponent nr2 ootamis funktsioon

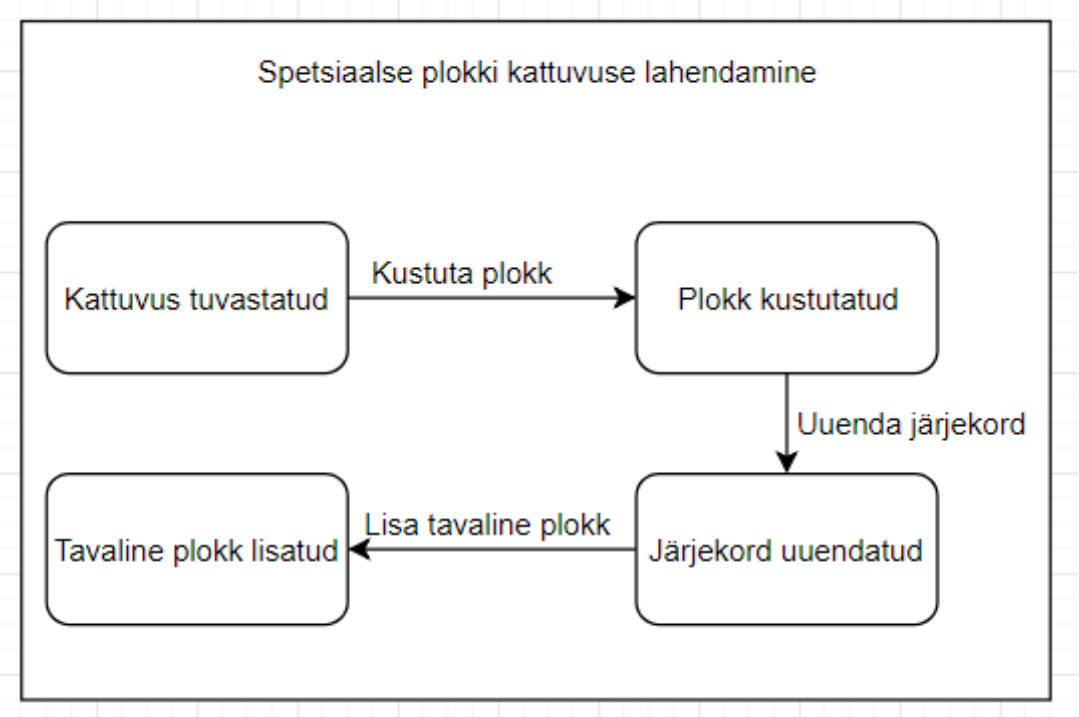
4.3. Algoritmi suutlikus probleeme lahendada

PCG peamiseks probleemiks on kattuvuste lahendamine, mis tekib iga kolmanda ploki lisamisega mängumaailma ning nende lahendamine peaks võtma võimalikult vähe aega. Kattuvuste lahendamiseks on loodud kaks funktsiooni üks vastutab tavalise ploki kattuvuste eest ja teine spetsiaalsete plokkide kattuvuste eest. Kui tuleb kattuvus tavalise ploki tõttu, siis algoritm lihtsalt kustutab selle ploki ja uuendab järjekorda Joonis 23. Spetsiaalse ploki kattuvuse juures algoritm üritab lisada samasse kohta tavalist plokki Joonis 24. Arendamise etapil viidi läbi testid, kus püüti teha nii, et algoritm üritaks samasse kohta lisada mitut plokki, mille tulemuseks oli see, et algoritm muutus väga aeglaseks ning kulutas väga palju arvuti ressursse - seega praeguses versioonis tehakse ainult korduv lisamine spetsiaalsete plokkide puhul. Komponentidevaheline suhtlemine on piiratud 2sekundilise ajaga, kui selle aja jooksul komponendid ei anna teada oma olekust, siis seda loetakse kui probleemiks

ning käitutakse nii, nagu oleks tekkinud kattuvus..



Joonis 23. Tavalise plokki kattuvuse lahendamine



Joonis 24. Spetsiaalse plokki kattuvuste lahendamine

5. Testimine

Peamised testimise meetodid on *white box testing* ehk valge kasti testimine, *black box testing* ehk musta kasti testimine ja *grey box testing* ehk halli kasti testimine. Valge kasti testimisel on testijal kood kogu aeg kättesaadav, tänu sellele testija saab näha, millal ja kus kohas mingit funktsioone kasutatakse. Musta kasti testimisel ei ole testijal koodile ligipääsu, tavaliselt viiakse selliseid teste läbi testijate poolt, kes mängivad mängu, nagu see oleks valmis, kirjutades üles, kus kohas ja millises olukorras tekkis mingi viga. Halli kasti testimine on valge ja musta kasti kombinatsioon [32, 33]. Selles töös kasutati valge kasti testimise meetodit, kuna testimist viidi läbi kogu töö protsessi ajal.

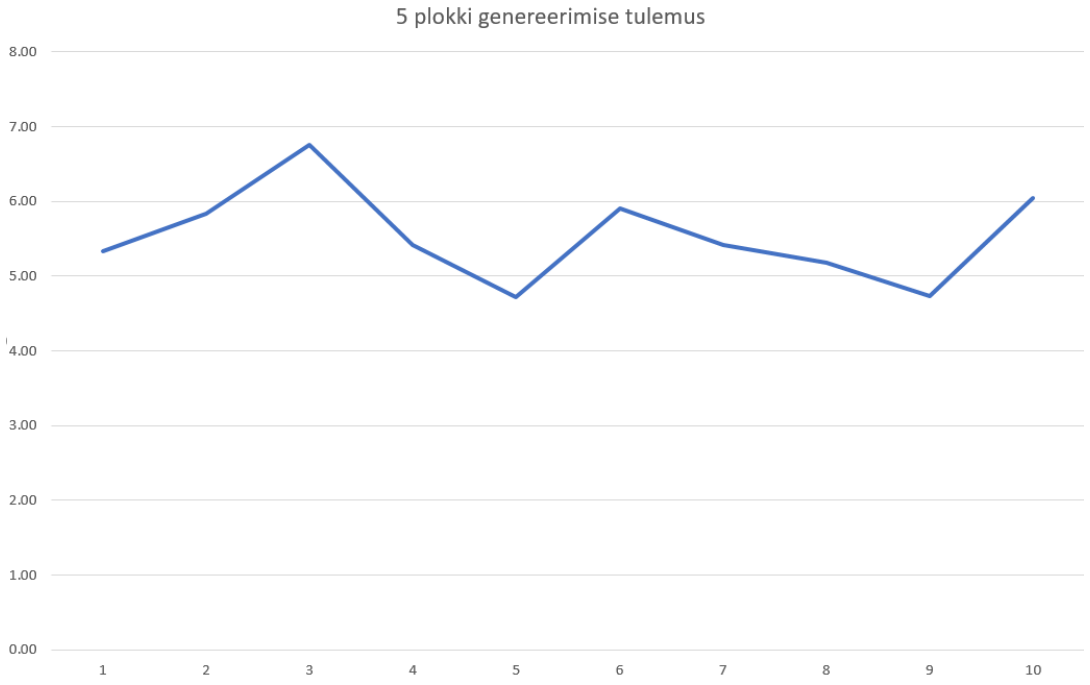
Saab tuua 30 konkreetset testi, kus 10 testi oli tehtud 5 ploki piiranguga, 10 testi oli tehtud 10 ploki piiranguga ja 10 testi 100 ploki piiranguga, lõpptulemuses oli oodatud plokkide koguarv mängumaailmas võrdne ploki piirangu arvuga pluss neli spetsiaalset plokki ehk 5 ploki piiranguga peaks olema 9 plokki mängumaailmas, 10 ploki puhul 14 plokki ning 100 ploki puhul 104 plokki. Mängumaailma genereerimise ajal oli eeldatud, et ühe ploki lisamine mängumaailma peaks võtma umbes ühe sekundi [34].

5.1. 5 plokki testimise tulemused

Testimise tulemuseks tuli välja, et 5 ploki piiranguga *PCG* võttis keskmiselt 5.53 sekundit mängumaailma genereerimiseks ning algoritm ei suutnut 30% korral pidada piirangut, kuna mõningatel juhtudel plokid ilmusid mängumaailma nii, et algoritm ei suutnut spetsiaalseid plokke lisada, seega ta pidi ühe või kaks tavalist plokki lisama, et saaks spetsiaalseid plokke ilmutada mängumaailma. Täpsemad tulemused on näha järgmises tabelis 1 ja graafiliselt joonisel 25. Keskmiselt võttis ühe ploki lisamine mängumaailma algoritmile aega 1.1 sekundit, mis oli oodatud tulemusest aeglasem 0.1 sekundit.

Katse	Aeg	Plokke
1	5.33	10
2	5.84	10
3	6.75	9
4	5.41	9
5	4.72	9
6	5.90	9
7	5.42	11
8	5.18	9
9	4.74	9
10	6.05	9

Tabel 1. 5 plokki testimise tulemused



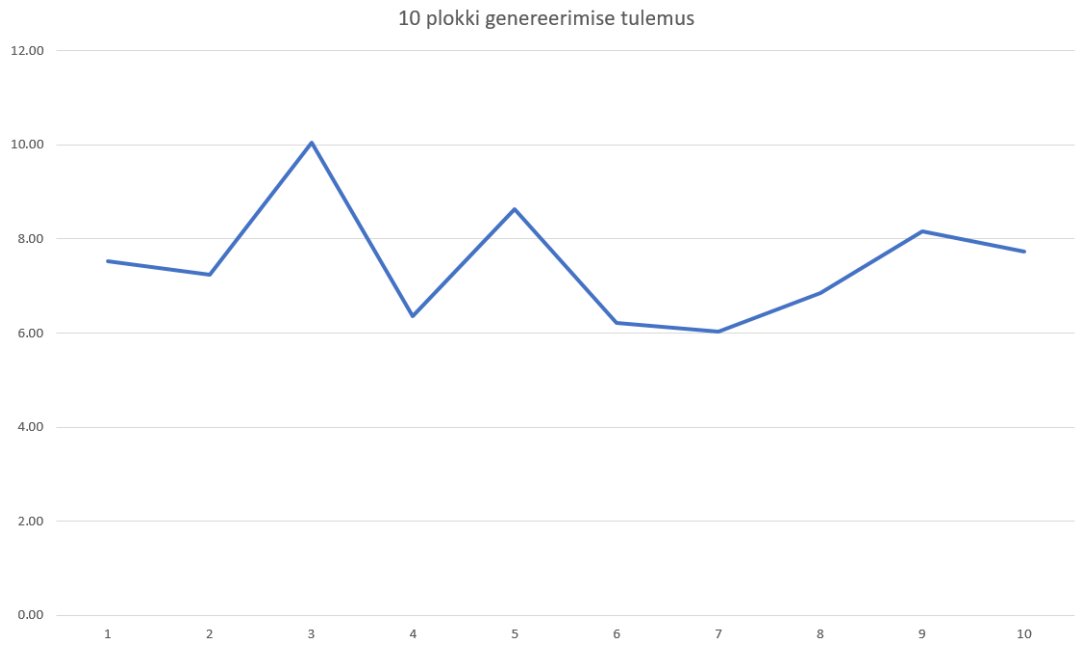
Joonis 25. 5 plokki genereerimise aja tulemuse graafik

5.2. 10 plokki testimise tulemused

10 ploki piiranguga *PCG* suutis keskmiselt genereerida mängumaailma 7.48 sekundiga ning ainult ühel korral algoritm ei suutnut pidada piiranguid, tekkinud probleem oli samasugune, mis 5 ploki genereerimisel. Täpsemad tulemused on tabelis 2 ja graafiliselt joonisel 26. Keskmiselt võttis ühe ploki lisamine mängumaailma algoritmile aega umbes 0.75 sekundid, mis oli oodatavast tulemusest 0.15 sekundit kiirem.

Katse	Aeg	Plokke
1	7.53	14
2	7.24	14
3	10.04	14
4	6.35	14
5	8.64	14
6	6.22	14
7	6.03	14
8	6.86	14
9	8.17	15
10	7.74	14

Tabel 2. 10 plokki testimise tulemused



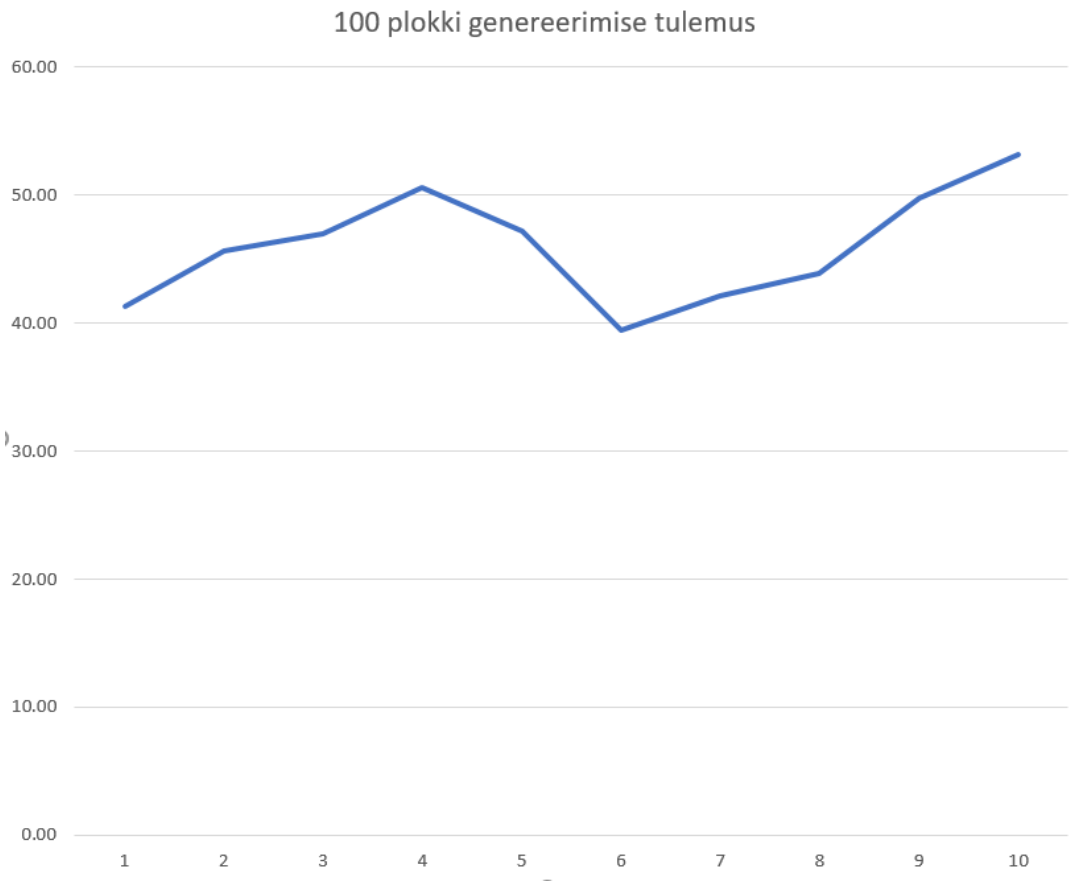
Joonis 26. 10 plokki genereerimise aja tulemuse graafik

5.3. 100 plokki testimise tulemused

Viimaseks testiks kasutati 100 ploki piirangut, kus mängumaailma genereerimine võttis keskmiselt 46 sekundit ning algoritm ei läinud piirangutest välja. Täpsemad tulemused on tabelis 3 ja graafiliselt joonisel 27. Algoritm ei läinud üle piirangutest, kuna algoritmil oli päris palju ruumi, et lisada spetsiaalseid plokke, ilma et oleks vaja lisada tavalisi plokke. Keskmiselt võttis ühe ploki lisamine aega 0.46 sekundit, mis on 0.54 sekundit kiirem oodatavast tulemusest.

Katse	Aeg	Plokke
1	41.27	104
2	45.65	104
3	46.99	104
4	50.56	104
5	47.16	104
6	39.43	104
7	42.09	104
8	43.88	104
9	49.80	104
10	53.19	104

Tabel 3. 100 plokki testimise tulemused



Joonis 27. 100 plokki genereerimise aja tulemuse graafik

6. Järeldused

Testimise tulemuste järgi võib öelda, et *PCG* suudab genereerida unikaalseid mängumaailmu minimaalsete parameetritega, aga selle saavutamiseks peab mõningatel üksikolukordadel piirangutest välja minema, mis võib tunduda halb, aga tehniliselt on see hea, kuna algoritm ei pea uuesti oma tööd alustama, vaid lisab ühe või kaks lisaplokki, et tema algoritm saaks lõppeda. Kui aga *PCG* jälgiks neid piiranguid rangelt, siis mängumaailma genereerimine väiksemate parameetritega võtaks rohkem aega, mida ei tohi lubada. Need piirangud on vajalikud selleks, et algoritm saaks oma tööd lõpetada teatud mahus, mitte sundida algoritmi tegema nii, nagu on kirjas, *PCG* peab olema vaba loominguga ilma suurte piiranguteta. Nagu varem mainitud, kui anda algoritmile vähe sisendparameetreid, ei ole algoritm suuteline oma tööd lõpetama, kus selles töös tehtud *PCG* suudab minimaalsete parameetritega genereerida unikaalseid mängumaailmu, aga kaotab genereerimise aja ja täpsuse, kuna algoritm peab lisasamme tegema, et tagada eeldatav tulemus. Kui aga tehtud *PCG*le anda piisavalt parameetreid, siis algoritm mitte ainult ei suuda genereerida rohkem erinevaid mängumaailmu, vaid teeb seda efektiivsemalt. Minimaalsed parameetrid tehtud *PCG*le on viie ploki piiramine, hea on 10 ja enama ploki piiranguga, kus iga ploki tekitamine mängumaailma on umbes pool sekundit, mis on oodatavast tulemusest kaks korda parem.

7. Kokkuvõte

Käesoleva lõputöö eesmärgiks on luua *PCG*, mis oleks suuteline genereerima unikaalset mängumaailma, mis oleks piisavalt paindlik ning efektiivne. Töö on jaotatud neljaks osaks, kaks esimest osa on analüüsitavad ja viimased kaks osa on suunatud algoritmi tööprotsessi seletamiseks-kirjeldamiseks ning algoritmi testimiseks.

Protseduurilise genereerimise sisust ehk *PCG* analüüsist selgus, et on erinevaid *PCG*sid ning igal tüübil on oma positiivsed ja negatiivsed küljed. Lisaks selgus, miks suur-ettevõtted kasutavad *PCG*d suures osas mängu arendamise etapil, kus aga *indie* ehk iseseisvad ettevõtted eksperimenteerivad uute võimalustega kasutades *PCG*d.

Töökeskkonna osas on töös välja toodud kasutatavad programmid ning programmi valiku põhjused. Lisaks oli lühidalt võrreldud tavalise progammeerimise viisi ehk tekstilist programmeerimist ja töökeskkonnas olevat graafilist programmeerimist. Töökeskkonna lõpus on võrreldud kasutatavat programmi lähtuvalt konkurendi võimalustest.

Viimases kahes osas on välja toodud oodatavad tulemused loodud *PCG*lt ning selle tegelik tulemus, samas on selgitatud, kuidas töötavad algoritmi peamised funktsioonid ning kuidas algoritm suudab probleeme lahendada. Testimise osas selgus, et tehtud algoritmi tulemused olid paremad kui oodatud tulemused, testimisel toodi välja 30 erinevat testi, kus kasutati kolme erinevat piirangut.

Antud töö käigus loodud-koostatud *PCG* on suuteline genereerima suvalist unikaalset mängumaailma, kasutades erinevaid plokke, mille sisu võib olla erinev - see ei sega plokkide kujundamist. Algoritm on suuteline parandama jooksvalt tekkinud vigu ning teatama sellest teistele plokkidele. Tehtud algoritm on üks osa suurest projektist, mille arendamine jätkub tulevikus.

Kasutatud kirjandus

[1] F. H. Edmund McMillen. (2011) The binding of isaac. [Last access: 28.04.2018]. [Online]. Available: https://store.steampowered.com/app/113200/The_Binding_of_Isaac/

[2] A. Baldwin, S. Dahlskog, J. M. Font, and J. Holmberg, "Mixed-initiative procedural generation of dungeons using game design patterns," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, aug 2017.

[3] R. van der Linden, R. Lopes, and R. Bidarra, "Procedural generation of dungeons," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 1, pp. 78–89, mar 2014.

[4] L. Grace. (2005) Game type and game genre. [Last access: 28.04.2018]. [Online]. Available: http://www.lgrace.com/documents/Game_types_and_genres.pdf

[5] Editor. (2017) The different types of entertainment. [Last access: 28.04.2018]. [Online]. Available: http://www.prtt.org/types-of-entertainment/

[6] M. Team. (2017) The major differences between 'indie' and 'aaa' video games. [Last access: 28.04.2018]. [Online]. Available: https://www.windowscentral.com/indie-vs-aaa-which-type-game-you

[7] PEGI. (2017) Pegi age ratings. [Last access: 10.05.2018]. [Online]. Available: https://pegi.info/

[8] E. S. Association. (2018) Esrb entertainment software rating board. [Last access: 12.05.2018]. [Online]. Available: http://www.esrb.org/

[9] B. LOWRY. (2017) Complete list of game genres. [Last access: 28.04.2018]. [Online]. Available: https://www.windowscentral.com/indie-vs-aaa-which-type-game-you

[10] R. Mahesa. (2017) The difference between working in indie and aaa game development. [Last access: 28.04.2018]. [Online]. Available: https://hub.packtpub.com/difference-between-working-indie-and-aaa-game-development/

[11] S. S. M. Studio. (2018) God of war. [Last access: 12.05.2018]. [Online]. Available: https://www.playstation.com/en-us/games/god-of-war-ps4/

[12] K. T. Kama, “Indie mÃngu loomise efektiivseima teekonna analÃÃs,” Master’s thesis, tallinna tehnikaÃŒlikool, 2017.

[13] D. S. Julian Togelius, Emil Kastbjerg and G. N. Yannakakis. (2011) What is procedural content generation? [Last access: 03.05.2018]. [Online]. Available: http://julian.togelius.com/Togelius2011What.pdf

[14] D. Pettifor. (2014) Maze generator. [Last access: 02.05.2018]. [Online]. Available: https://www3.nd.edu/~dpettifo/software/maze/index.html

[15] CrazyLegs66. (2017) 3d maze escape (version 2.16). [Last access: 03.05.2018]. [Online]. Available: https://scratch.mit.edu/projects/14777574/

[16] J. A. Brown, B. Lutfullin, and P. Oreshin, “Procedural content generation of level layouts for hotline miami,” in *2017 9th Computer Science and Electronic Engineering (CEEC)*. IEEE, sep 2017.

[17] C. CHAPPLE. (2014) How procedural generation is being used to develop exciting new games. [Last access: 07.05.2018]. [Online]. Available: https://www.mcvuk.com/development/how-procedural-generation-is-being-used-to-develop-exciting-new-games

[18] V. Valtchanov and J. A. Brown, “Evolving dungeon crawler levels with relative placement,” in *Proceedings of the Fifth International C∗ConferenceonComputerScienceandSoftwareEngineering − C3S2E12. ACMPress*, 2012.

[19] D. Distribution. (2013) Paranautical activity. [Last access: 02.05.2018]. [Online]. Available: https://store.steampowered.com/app/250580/Paranautical_Activity_Deluxe_Atonement_Edition/

[20] T. A. Tanya X. Short. (2017) Procedural generation in game design. [Last access: 29.04.2018]. [Online]. Available: https://books.google.ee/books?id=-ZcnDwAAQBAJ&pg=PT23&lpg=PT23&dq=why+aaa+games+dont+use+PCG&source=bl&ots=3uNGyq0FPO&sig=BKLXdWbyKNHtFE1jIjMsjXPFl_o&hl=en&sa=X&ved=0ahUKEwjC6v-L3d_aAhWFA5oKHaZgAhsQ6AEIaDAH#v=onepage&q&f=false

[21] M. HART. (2016) No manâs sky has more discovered species than earth. [Last access: 29.04.2018]. [Online]. Available: https://nerdist.com/no-mans-sky-has-more-discovered-species-than-earth/

[22] ALASTAIR. (2016) The rise and fall of no manâs sky. [Last access: 29.04.2018]. [Online]. Available: https://www.gamespew.com/2016/10/the-rise-and-fall-of-no-mans-sky/

[23] N. Yakimov. (2017) What is a list of programming languages ordered from easiest to hardest to learn? [Last access: 02.05.2018]. [Online]. Available: https://www.quora.com/What-is-a-list-of-programming-languages-ordered-from-easiest-to-hardest-to-learn

[24] E. Games. (2018) Unreal engine. [Last access: 02.05.2018]. [Online]. Available: https://www.unrealengine.com/en-US/what-is-unreal-engine-4

[25] L. FRAM. (2018) Best game engines for indie game development. [Last access: 02.05.2018]. [Online]. Available: https://blog.g2crowd.com/blog/game-engine/best-game-engines-indie-game-development/

[26] All3DP. (2018) 30 best 3d design/3d modeling software tools (15 are free). [Last access: 02.05.2018]. [Online]. Available: https://all3dp.com/1/best-free-3d-modeling-software-3d-cad-3d-design-software/

[27] Slant. (2018) What are the best 3d texture painting softwares? [Last access: 02.05.2018]. [Online]. Available: https://www.slant.co/topics/8643/~3d-texture-painting-softwares

[28] Allegorithmic. (2017) The best of substance 2017: Games. [Last access: 02.05.2018]. [Online]. Available: https://www.allegorithmic.com/blog/best-substance-2017-games

[29] D. Helgason. (2018) Unity. [Last access: 30.04.2018]. [Online]. Available: https://www.upwork.com/hiring/development/c-sharp-vs-c-plus-plus/

[30] J. MARSH. (2016) C vs. c++: Which language is right for your software project? [Last access: 30.04.2018]. [Online]. Available: https://www.upwork.com/hiring/development/c-sharp-vs-c-plus-plus/

[31] H. ElHady. (2018) Top game engines in 2018. [Last access: 30.04.2018]. [Online]. Available: https://blog.instabug.com/2017/12/game-engines/

[32] C. WODEHOUSE. (2017) White-box testing vs. black-box testing. [Last access: 01.05.2018]. [Online]. Available: https://www.upwork.com/hiring/development/white-box-testing-vs-black-box-testing/

[33] T. S. T. S. P. Ltd. (2017) Gray box vs black box vs white box testing: Briefly explained difference. [Last access: 01.05.2018]. [Online]. Available: https://www.utest.com/articles/gray-box-vs-black-box-vs-white-box-testing-briefly-explained-difference

[34] D. Fomitsjov. (2018) Pcg genereerimise testimise hetktommised. [Last access: 01.05.2018]. [Online]. Available: https://imgur.com/a/Oer7Fra