Jan-Erik Gunnar

PELIKENTTIEN PROSEDURAALINEN GENEROINTI UNITY-PELIMOOTTORISSA

Opinnäytetyö

Tekniikan ammattikorkeakoulututkinto

Tieto- ja viestintätekniikka / Peliohjelmointi

2023

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Insinööri (AMK)
Tekijä/Tekijät	Jan-Erik Gunnar
Työn nimi	Pelikenttien proseduraalinen generointi Unity-pelimoottorissa
Toimeksiantaja	Xamk Gamelab
Vuosi	2023
Sivut	54 sivua
Työn ohjaaja(t)	Pekka Vilpponen, Teemu Saarelainen

TIIVISTELMÄ

Opinnäytetyön tavoitteena on luoda työkalu proseduraalisten kenttien luomiseksi Unity-pelimoottorissa. Tämä toteutetaan kolmivaiheisena kokonaisuutena, joka koostuu tutkimuksesta, suunnitellusta ja toteutuksesta.

Opinnäytetyön toimiantajana toimii Kaakkois-Suomen ammattikorkeakoulun (Xamk) Gamelab. Kotkan kampuksella sijaitseva Gamelab on peliohjelmoinnin insinööriopiskelijoille suunnattu oppimisympäristö. Opinnäytetyön tarkoituksena on toteuttaa proseduraalisen kenttägeneroinnin työkalu seuraten akateemisia ja eettisiä odotuksia opinnäytetyön toteutuksen ja tulosten kannalta.

Tutkimus tehdään laadullisella tutkimuksella tarkastelemalla kolmea valittua peliä, joiden toteutuksessa on käytetty proseduraalista generointia. Nämä tutkimuspelit ovat Binding of Isaac, No Man’s Sky ja Dwarf Fortress. Tutkimuksessa yritetään myös saada selville proseduraalisen generoinnin käyttämistapoja, siitä muodostuvia ongelmia sekä mahdollisia ratkaisuja ongelmiin, joita proseduraalisen generoinnin käytössä voi olla. Laadullisen tutkimuksen perusteella tarkastellaan Unity-pelimoottoria ja C#-ohjelmointikieltä, jotta voidaan muodostaa selvä käsitys työkaluista, joita hyödynnetään opinnäytetyön toteutuksessa. Tutkimusosion jälkeen rakennetaan geneerinen suunnitelma, jota hyödynnetään Unity-pelimoottorille tarkoitetun työkalun suunnittelemiseen. Tämän toteuttamiseksi luodaan ensin työkalu suunnitelmaa hyödyntämällä ja työkalun toimivuutta testataan luomalla kaksi erilaista proseduraalista generointiprojektia.

Opinnäytetyön lopputuloksena valmistui työkalu, jota käytettiin 2D- ja 3D-pelikenttien generoimiseen. Työkalu toimii hyödyntämällä generoinnin yleisiä standardeja, kuten modulaarista suunnittelua, loogisia sekä matemaattisia sääntöjä, ja kolmivaiheista proseduraalisen generoinnin käytäntöä, joka on ”generoi, tarkista ja etene”. Kolmivaiheinen toteutusmalli toimii ratkaisuna ongelmiin, joita voi muodostua proseduraalisen generoinnin käyttämisessä. Työkalu on pohja, jota voidaan hyödyntää yleisesti sellaisiin projekteihin, jotka tarvitsevat proseduraalista generointia.

Asiasanat: opinnäytetyö, Unity-pelimoottori, proseduraalinen generointi, C#, pelinkehitys

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author (authors)	Jan-Erik Gunnar
Thesis title	Procedural generation of game levels in Unity-game engine
Commissioned by	Xamk Gamelab
Time	2023
Pages	54 pages
Supervisor	Pekka Vilpponen, Teemu Saarelainen

ABSTRACT

The aim of the thesis is to create a tool for generating procedural levels in the Unity game engine. This is implemented as a three-phase process consisting of research, plan, and implementation.

The host of the thesis is the Gamelab of the South-Eastern Finland University of Applied Sciences (Xamk). Located on the Kotka campus, Gamelab is a learning environment designed for students of game programming engineering. The purpose of the thesis is to implement a procedural level generation tool following academic and ethical expectations for the execution and outcomes of the thesis.

The research is done with qualitative research by examining three selected games that have used procedural generation in their implementation. These case study games are Binding of Isaac, No Man's Sky, and Dwarf Fortress. The research also attempts to find out the ways of using procedural generation, challenges it presents, and possible solutions to the problems that may arise in the use of procedural generation. Based on the qualitative research, the Unity Game Engine and C# programming language are examined to form a clear understanding of the tools used in the implementation of the thesis. After the research section, a generic plan is built, which is utilized in designing the tool intended for the Unity Game Engine. To implement this, a tool plan is first created using the plan and the functionality of the tool is tested by creating two different procedural generation projects.

As a result of the thesis, a tool was completed that was used to generate 2D and 3D game fields. The tool works by utilizing general generation standards, such as modular design, logical and mathematical rules, and a three-phase procedural generation practice, which is “generate, validate, and progress”. The three-phase implementation model serves as a solution to problems that may arise in the use of procedural generation. The tool is a foundation that can be generally used in projects that require procedural generation.

Keywords: thesis, Unity-game engine, procedural generation, c#, game development

SISÄLLYS

LIITTEET

Liite 1. Unity-projekti, jossa on kaksi kohtausta.

1 JOHDANTO

Pelien laajentuessa on herännyt tarve kehittää pelisisältöä automaattisesti (Carpender 2011). Tähän tarkoitukseen on kehitetty useita eri tapoja automaattisen pelisisällön kehittämiseen. Yksi näistä menetelmistä on proseduraalinen generointi, joka on monivaiheinen tapa kehittää automaattisesti uutta sisältöä (Schatzeder 2021). Proseduraalisen sisällön generointikohteena voi olla pelikenttä, pelin sisäinen historia tai pelihahmot. Tällä tavalla voidaan vähentää kustannuksia ja kehitysaikaa pelinkehityksessä sekä keskittää resursseja muihin pelinkehityksen osa-alueisiin.

Opinnäytetyön tavoitteena on tutkia tarkemmin menetelmiä, joilla proseduraalista pelisisältöä kehitetään, ja mahdollisia ongelmia, joita proseduraalisesta generoinnista voi syntyä sekä sitä, miten proseduraalista generointia voidaan hyödyntää pelikenttien generointiin Unity-pelimoottorissa. Tutkimus tehdään pääasiassa tutkimalla pelejä, joiden perustoimivuus perustuu proseduraaliseen generointiin. Näistä peleistä etsitään yhtenäiset menetelmät, joilla proseduraalinen generointi on toteutettu. Tällä tavalla proseduraalisesta generoinnista löydetään yleiset standardit, joita tullaan hyödyntämään Unity-pelimoottorilla toteutettavassa proseduraalisessa generointityökalussa sekä luodaan esimerkki projektista, joka voidaan toteuttaa työkalun avulla.

Opinnäytetyön toimiantajana toimii Kaakkois-Suomen ammattikorkeakoulun (Xamk) Gamelab. Kotkan kampuksella sijaitseva Gamelab on peliohjelmoinnin insinööriopiskelijoille suunnattu oppimisympäristö. Opinnäytetyön tarkoituksena on toteuttaa proseduraalista generointia hyödyntävä työkalu, jonka toteuttamisessa seurataan akateemisia ja eettisiä odotuksia opinnäytetyön toteutuksen ja tulosten kannalta.

2 TUTKIMUKSEN PERUSTEET

Opinnäytetyön tavoitteena on tuottaa Unity-pelimoottorin sisällä toimiva generointityökalu, jonka avulla on mahdollista luoda pelikenttiä. Proseduraalisen generointityökalun toteuttaminen sisältää kuitenkin omat haasteensa aiheen rajaamisen kannalta, koska kehittäjän näkökulmasta ei ole varmaa tietoa siitä,

minkä tyyliseen peliin generointi on tarkoitettu. Tämän takia työkalu pitää olla suunniteltu helposti muokattavaksi, laajennettavaksi ja yleiskäytännölliseksi. Tämä tarkoittaa sitä, että työkalun pitää seurata hyviä yleisiä standardeja, jotka ovat odotettavissa yleiseen käyttöön tarkoitetuilta työkaluilta, mikä tuo uudet haasteet suunnittelun kannalta. Tämän takia ennen työkalun suunnittelemista opinnäytetyö aloitetaan havaintoihin perustuvalla tutkimuksella eli laadullisella tutkimuksella, jonka tavoitteena on hahmottaa yleiset käytännöt, standardit, ongelmat ja menetelmät, mitkä pitää ottaa huomioon lähestyttäessä proseduraalista generointia. Tämä tutkimus tehdään analysoimalla muutamia valittuja pelejä, joihin kuuluvat Dwarf Fortress, Binding of Isaac ja No Man’s Sky. Ne kaikki hyödyntävät proseduraalista generointia pelisisällön ja -kenttien rakentamisessa.

Laadullisessa eli kvalitatiivisessa tutkimuksessa pyritään ymmärtämään valitun aiheen laatua, ominaisuuksia ja merkityksiä (Koppa 2021). Tutkimustyypin takia tutkittavat pelit ovat valittu eri peligenreistä, jotta voidaan poistaa mahdolliset ennakkoluulot ja tuoda esille trendit, joita proseduraalisesti generoidussa sisällössä esiintyy genrestä huolimatta. Havainnot merkitään ylös niitä tehtäessä ja tutkimuksessa esitetään päähuomiot näistä havainnoista. Peleissä keskitytään sisältöön, joka käyttää proseduraalista generointia. Muu sisältö jätetään huomiomatta paitsi, jos tämä sisältö on oleellista proseduraalisen generoinnin toteuttamisen kannalta. Tämän keskittymisen rajauksen tarkoituksena on tutkimusaiheessa pitäytyminen.

Laadullisen tutkimuksen jälkeen alkaa kehittämistyö. Kehittämistyö eli toiminnallinen opinnäytetyö koostuu työkalun laadulliseen tutkimukseen perustuvasta suunnittelemisesta ja toteutuksesta. Suunnittelun tavoitteena on rakentaa selvä toimintakaavio, jonka avulla työkalun toimivuus rakennetaan. Suunnitelma tehdään ensin geneerisesti eli yleisellä tavalla. Tällöin generointiprosessi kuvaillaan mahdollisimman yleisesti ilman, että sitä kiinnitetään mihinkään pelimoottoriin tai prosessiin. Tätä seuraa tarkennettu suunnitelma, joka on rajattu Unity-pelimoottoriin. Tarkemman suunnitelman tavoitteena on rakentaa tarkka kuvaus osista, joita työkalu tarvitsee toimiakseen yhtenäisenä kokonaisuutena.

Toteutuksessa rakennetaan työkalu, jonka avulla luodaan kaksi yksinkertaista pelikenttäprojektia. Toteutuksen tavoitteena on näyttää työkalun potentiaalia sekä erilaisia käyttötarkoituksia, minkä vuoksi projektit ovat huomattavasti erilaiset. Toinen tulee käyttämään 2D- eli kaksiulotteista peliympäristöä ja toinen 3D- eli kolmiulotteista peliympäristöä. Kyseiset projektit valitaan laadullisessa tutkimuksessa tehtyjen havaintojen perusteella ja toteutetaan seuraten yleisiä havaintoja, joita on kerätty opinnäytetyön aikana.

2.1 Tutkimuskysymys ja tutkimusmenetelmät

Koska opinnäyteyön tarkoituksena on työkalun toteuttaminen, tutkimuskysymykset ja -menetelmät pitää suunnitella tavalla, jolla ne auttavat työkalun toteutuksessa. Aluksi tuotetaan analysointi, jossa pääsijaisena tutkimusmenetelmänä käytetään lähteiden hakua, niihin viittaamista ja valittujen pelien analysointia eli laadullista sisältöanalyysia (Vuori 2021). Analysoinnissa tarkastellaan ensinnäkin sitä, miten kyseiset pelit ovat hyödyntäneet proseduraalisesti generoitua sisältöä ja mitä ongelmia näissä ratkaisuissa ilmeni. Toiseksi tutkitaan niitä keinoja, joilla valitut pelit ovat välttäneet ongelmia, joita muodostuu proseduraalisen generoinnin käyttämisestä. Pelit on tarkoituksella valittu eri genreistä, jotta yhtenevät tekijät erottuisivat selkeämmin. Analyysin aikana tarkastellaan myös aiheeseen kuuluvia lähteitä, jotka keskittyvät tarkastelemaan aihetta suunnittelun kannalta. Tällä tavalla saadaan parempi yleiskuva aiheesta, mikä parantaa samalla analyysin tarkkuutta ja laatua. Analyysia hyödynnetään työkalun suunnittelussa ja toteutuksessa.

Tutkimusongelmaa muodostettaessa on tärkeää miettiä apukysymyksiä, jotka auttavat opinnäytetyötä pysymään rajatussa aiheessa (Günther & Hasanen 2021). Näitä tutkimuskysymyksiä voidaan käyttää kokoaikaisesti tutkimuksen aikana. Tämän lisäksi tutkimuksen onnistuminen voidaan mitata tarkistamalla, onko tutkimuskysymyksiin saatu vastauksia. Tämän takia tutkimuksessa yritetään saada vastaukset seuraaviin kysymyksiin, joita voidaan hyödyntää työkalun suunnitteluvaiheessa:

- Mitä proseduraalinen generointi on?
- Miten proseduraalinen generointi toimii?
- Mitkä ovat yleiset menetelmät ja standardit proseduraalisessa generoinnissa?

- Mitä mahdollisia ongelmia proseduraalisesti generoidussa sisällössä on ja miten näitä ongelmia voi välttää?
- Miten proseduraalista generointisysteemiä voi käyttää eri tarkoituksiin ja tilanteisiin?

2.2 Toteutus

Tutkimusta ei voi toteuttaa ymmärtämättä valittua aihetta, jonka takia opinnäytetyön tutkimus aloitetaan tutkimalla termiä proseduraalinen generointi ja mitä se käytännössä tarkoittaa. Tämän termin tutkimus tehdään pääasiassa tarkastelemalla termin käyttämistä akateemisissa tutkimuksissa, kirjallisuudessa ja suunnitteludokumenteissa. Tavoitteena on saada mahdollisimman tarkka ja selkeä tapa kuvailla termiä proseduraalinen generointi, jota tullaan käyttämään opinnäytetyön aikana.

Opinnäytetyön tutkimuksen toinen vaihe toteutetaan tarkastelemalla valittuja pelejä, ja niistä kerätään systemaattisesti erilaisia tapoja, joilla pelit ovat käyttäneet proseduraalisesti generoitua sisältöä. Samalla kerätään myös tietoa peliin liittyvästä mediasta. Tätä kerättyä tietoa hyödynnetään tutkimuksen rajaamisessa proseduraaliseen generointiin, sillä se voi olla vaikeasti erotettavissa muusta pelisisällöstä.

Tämän jälkeen tehdään vertailu pelin keskeisistä yhtenäisyyksistä ja selvitetään vastaukset tutkimuskysymyksiin. Jos peleistä ei löydy yhteneviä tekijöitä proseduraalisen generoinnin kannalta, niin pelejä tarkastellaan yksitellen ja aihetta käsitteleviä tutkimuskysymyksiä tarkastellaan pienemmässä mittakaavassa. Tässä tapauksessa työkalun suunnitteluvaiheessa keskitytään tapoihin, joilla voidaan toteuttaa eri lähestymistapoja samaa työkalua käyttäen. Yhtenevien tekijöiden havaitseminen olisi kuitenkin parasta pelien proseduraalisen toteutuksen kannalta, jotta havaintoja on mahdollista hyödyntää laajassa mittakaavassa.

Viimeisessä tutkimusvaiheessa toteutetaan Unity-pelimoottorin ja C#-ohjelmointikielen tarkastelu. Koska Unity-pelimoottoria tullaan käyttämään toteutuksessa, on tärkeää avata pelimoottorin toimintatapoja yleisesti. Samalla tarkasteluun on valittu C#-ohjelmointikieli, koska Unity käyttää sitä. Molemmat näistä

ovat työkaluja, joiden ymmärtäminen ja oikea käyttö ovat oleellinen osa opinnäytetyötä. Kummatkin ovat kuitenkin erittäin laajoja aiheita, joten jotta voidaan pitää opinnäytetyön skaala järkevänä, on oleellista rajata tarkastelu asioihin, joita hyödynnetään opinnäytetyössä. Tämän takia Unity-pelimoottorin tarkastelu tapahtuu laadullisen tutkimuksen jälkeen, koska laadullinen tutkimus antaa tarvittavat perustiedot suunnittelua varten. Perustietojen avulla on myös mahdollista hahmotella Unity-pelimoottorista oleelliset työkalut, joita tullaan hyödyntämään. Tämän takia analyysissa ei tulla huomioimaan Unity-pelimoottorilla tuotettuja sovelluksia vaan oleellisia perustoiminnallisuuksia ja hieman merkittävää taustatietoa.

Tutkimuksen suorittamisen jälkeen siirrytään viimeiseen vaiheeseen eli itse työkalun suunnittelemiseen. Työkalun suunnittelussa hyödynnetään havaintoja proseduraalisesta generoimisesta sekä yleisiä ohjelmoinnin lähestymismenetelmiä, joita löytyy kirjallisuudesta. Suunnittelun tavoitteena on tehdä selkeä toimintakaavio, jota voidaan käyttää toteutusta varten. Tämä toimintakaavio rakennetaan yleisestä näkökulmasta eli Unity-pelimoottoria ei oteta tässä vaiheessa huomioon millään tavalla. Tärkeintä tämän vaiheen onnistumisen kannalta on, että suunnitelma on helposti ymmärrettävissä ja yleisesti käytettävissä työympäristöstä huolimatta.

Toimintasuunnitelman valmistumisen jälkeen suunnitelma yhdistetään Unity-pelimoottoriin analysoimalla tapoja, joilla on järkevintä lähestyä työkalun eri toimintoja pelimoottorin sisällä. Tärkeintä tämän vaiheen kannalta on, että Unity-pelimoottorin erilaiset työkalut on hyvin otettu huomioon toteutuksen kannalta, mikä mahdollistaa sujuvan toteutuksen. Tärkeä huomio kuitenkin on, että työkalun pitäisi pystyä toimimaan pohjana monenlaisille eri projekteille eli se pitää suunnitella mahdollisimman avoimeksi, jotta sen käyttöönotto eri projekteja varten on mahdollisimman sujuvaa.

Toiseksi viimeisessä toteuttamisvaiheessa on itse työkalun toteuttaminen suunnitelman mukaan. Työkalun toteuttaminen pitäisi olla tässä vaiheessa pelkästään pohja, jonka päälle eri projekteja voi rakentaa. Toteutuksen jälkeen valitaan kaksi yksinkertaista projektia, joista toinen rakennetaan 2D- ja toinen

3D-pelimaailmaan. Tällä tavalla varmistetaan, että työkalu toimii erityyppisissä Unity-pelimoottorilla toteutettavissa peliprojekteissa.

Viimeisessä vaiheessa toteutetaan proseduraalisen kenttägeneroinnin projektit. Projektit valitaan analysoitujen pelien perusteella, mutta minimivaatimuksena on, että projektien lopputulokset ovat selkeästi toisistaan eroavat, jotta työkalun toimivuus voidaan varmentaa.

Tässä vaiheessa opinnäytetyö on valmis ja alkaa loppuanalyysi, jossa opinnäytetyö analysoidaan kokonaisuudessaan. Tämän vaiheen tärkein keskittymiskohta on tutkimusongelman ja tutkimuskysymysten tarkastelu opinnäytetyön ja työkalun kannalta. Opinnäytetyön onnistumisen kannalta on oleellista, että tutkimuskysymyksiin saadaan pätevät vastaukset, joiden vaikutukset näkyvät Unity-pelimoottorissa tuotetussa proseduraalisessa kenttägenerointisysteemissä.

2.3 Eettiset lähtökohdat

Opinnäytetöissä aiheen valinta on eettinen kysymys, jolloin pitää miettiä aiheen merkityksellisyyttä valitsemansa alan ja yhteiskunnan kannalta. Tämän lisäksi pitää varmistaa, että pystyy käsittelemään aihetta neutraalilta näkökannalta ilman, että antaisi omien ennakkoluulojen vaikuttaa tutkimustuloksiin tai tutkimukseen. Käytännössä tämä tarkoittaa, että koko tutkimuksen ajan aihetta pitää lähestyä neutraalilta ja avoimelta näkökulmalta ilman, että ennakkoluulot ja oletukset vaikuttavat tutkimukseen tai tuloksiin.

Proseduraalisen generoinnin yleisyyden vuoksi voi olettaa, että aiheesta on syntynyt yleiskäsitys. Tästä huolimatta tämä yleiskäsitys on erittäin pintapuolinen eikä vaikuta aiheen käsittelyyn negatiivisella tavalla, minkä takia aiheen neutraali käsittely säilyy eettisten odotuksien mukaisesti.

Aiheen relevanssisuuden kannalta proseduraalinen generointi esiintyy peleissä, mediassa ja ohjelmoinnissa aina vain useammin. Aiheen relevanttius on edistynyt viime vuonna tekoälyjen huomattavan kehityksen ansiosta, mikä

on mahdollistanut proseduraalisen generoinnin uuden aikakauden. Tämä tekee aiheesta erittäin ajankohtaisen ja merkityksellisen. Tämän seurauksesta on turvallista olettaa, että tutkimukset proseduraalisesta sisällöstä voivat olla hyödyllisiä yhteiskunnan kannalta ja auttavat proseduraalisten menetelmien käyttäjiä tulevaisuudessa.

Tuotettu työkalu on rajattuna Unity-pelimoottoriin, jotta se antaa opinnäytetyölle järkevän skaalan, jolloin se on toteutettavissa aikataulun mukaisesti. Tästä huolimatta työkalun suunnitteluperusteet eli standardit, jotka havaitaan tutkimuksen aikana sekä aikaansaadut tulokset ovat yleisesti hyödyllisiä proseduraalisen generoinnin käyttöönoton kannalta. Näiden perusteiden seurauksena voidaan todeta, että opinnäytetyö täyttää tutkimuksen eettiset perusvaatimukset ja lähtökohdat.

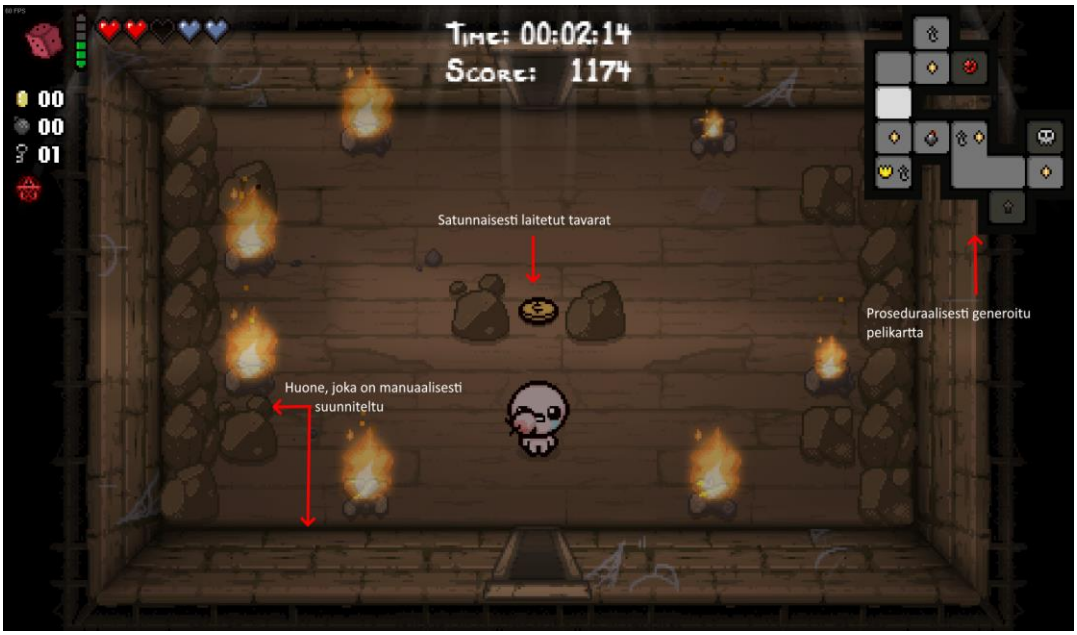
3 PROSEDURAALINEN GENEROINTI

Opinnäytetyön analyysin ja tutkimuskysymysten kannalta on tärkeä ymmärtää vastaus kysymykseen: mikä on proseduraalinen generointi? Tämä voi olla yllättävän vaikea kysymys, vaikka henkilölle olisikin muodostunut jo jonkinlainen ymmärrys kyseisestä termistä.

Pääongelmana termin määrittämisessä on epätarkkuus termin käytön kanssa, minkä takia termi itsessään on ollut tutkimusaiheena useissa tutkimuksissa kuten Tanskassa sijaitsevassa yliopistossa tehdyssä tutkimuksessa ” *What is Procedural Content Generation? Mario on the borderline”* (Togelius ym. 2011, 1). Tutkimuksen tekijät totesivat termin pääongelman olevan proseduraalisen käytön laajuus eri ammattiryhmien keskuudessa, jonka takia tarkka termin käyttö vaihtelee usein. Itse proseduraalinen luominen kuitenkin tarkoittaa tiedon tai sisällön tuottamista logiikan tai algoritmin mukaisesti sen sijaan, että ihminen loisi sisällön manuaalisesti (Sarawagi s.a).

Opinnäytetyön kannalta tarkastelemme termiä sen perusidean kannalta eli joko algoritmisen tai säännönmukaisen automaattisen datan tai sisällön generoinnin kannalta (Sarawagi s.a). Käytännössä tämä termin tarkennus sulkee

pois satunnaisesti generoidun sisällön, joka on yleensä generoitu ilman algoritmeja tai tarkempia generointisääntöjä. Toiseksi suljetaan pois pelaajien tekemä sisältö, jota voidaan käyttää proseduraalisessa generoinnissa. Näitä ovat esimerkiksi generointinumeron antaminen, generoinnin sääntöjen muokkaaminen tai proseduraalisessa generoinnissa hyödynnettävät pelaajan luomat pelinsisäiset sisällöt. Tämä johtaa luonnollisesti opinnäytetyön pääkeskittymisaiheeseen eli erilaisten pelikenttien proseduraaliseen generointiin. Pelin lähestymistavasta riippuen pelikentät tai -kartat rakennetaan usein proseduraalisesti pienemmistä valmiiksi suunnitelluista pelialueista ja pelin edetessä alueita täytetään peliobjekteilla satunnaista generointia käyttäen (kuva 1).



Kuva 1. Binding of Isaac: Rebirth -pelissä pelikartta on proseduraalisesti generoitu, mutta peli käyttää manuaalisesti tehtyjä huoneita, jotka täytetään satunnaisesti valituilla tavaroilla ja esteillä. Nämä huoneet yhdistetään proseduraalisesti pelikartaksi.

Perusongelmana proseduraalisen sisällön tutkimisessa on proseduraalisen sisällön erottaminen manuaalisesta ja satunnaisesti generoidusta sisällöstä. Ongelman voi opinnäytetyön kannalta välttää helpoiten tutkimalla valittuja tutkimuspelejä haastattelujen, artikkelien, koodianalyysien ja pelisuunnitteludokumenttien avulla. Tällöin on mahdollista välttää muun sisällön sekoittuminen proseduraaliseen sisältöön.

Pääideana on, että opinnäytetyön tutkimuksessa tarkastetaan proseduraalisen generoinnin prosessia ja lopputulosta sen sijaan, että erehdyttäisiin tutkimaan

pienempiä valmiiksi tehtyjä tai satunnaisesti valittuja osia, joita proseduraalinen generointi tyypillisesti hyödyntää osana generointiprosessia. Tällä tavalla vältetään virheellisiä tutkimustuloksia, jotka vaikuttaisivat negatiivisesti työkalun rakentamiseen.

4 PELIEN ANALYYSI

Tämä luku käsittelee proseduraalisen generoinnin analyysiä, tutkittavia pelejä ja niiden taustailmiöitä. Pelejä analysoitaessa keskitytään ainoastaan proseduraaliseen generointiin ja siihen liittyviin aineistoihin, jonka takia pelien historia, menestys ja mekaniikat jätetään analyysissa pääasiassa huomioimatta. Näistä kerrotaan ainoastaan silloin, mikäli sen mainitseminen on edellytys proseduraalisen generoinnin analyysia tai aineistoa varten. Pelien perusominaisuudet ja genre tullaan esittelemään yleisesti, jotta varmistetaan, että lukija ymmärtää pelien ja genrejen eroavaisuudet, taustan ja tällä tavalla hahmottaa proseduraalisen generoinnin käyttötavat ja käyttötapojen erot paremmin.

Pelin esittelylukua seuraa proseduraalisen generoinnin analysointia käsittelevä luku, jossa tarkastellaan pelin proseduraalista generointia käytön ja suunnittelun kannalta. Tässä osiossa ei kuitenkaan tulla vertailemaan pelien proseduraalisen generoinnin yhteneviä tekijöitä, vaan se tullaan jättämään lukuun 4.4 Yhtenäisyydet.

Yhtenäisyydet-luku nimensä mukaisesti yhdistää pelien analyysin yhtenäiseksi kokonaisuudeksi, jossa keskitytään vastaamaan tutkimuskysymyksiin valittujen pelien avulla. Luvussa pyritään löytämään tarvittavat yhtenevät tekijät johtopäätöksien tekemiseen, sillä se on oleellinen osa tutkimuksen lopullista suorittamista varten.

4.1 Binding of Isaac: Rebirth

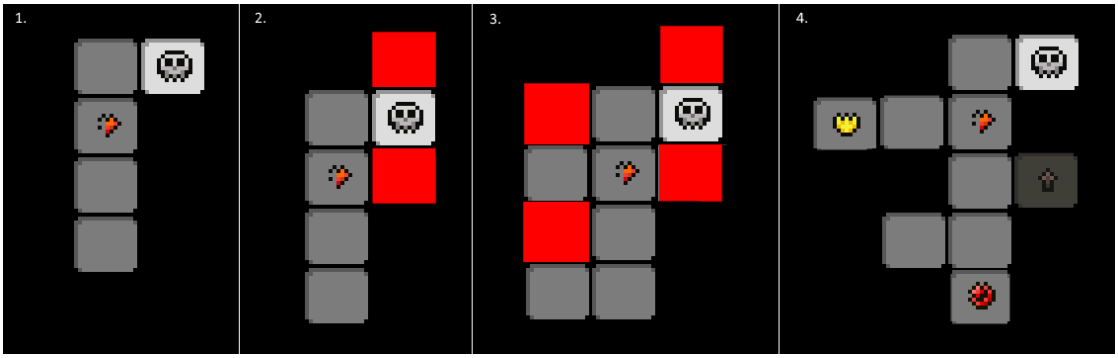
Binding of Isaac: Rebirth on roguemaiseen genreen kuuluva peli, joka julkaistiin vuonna 2014 (Nicalis, Inc.2014). Roguemainen-genressä tyypillisesti pelimaailma, kenttä tai ympäristö on proseduraalisesti generoitua ja pelaaja joutuu aloittamaan pelin alusta kuolemansa jälkeen. Roguemaiset pelit yleensä laajentuvat pelikertojen myötä joko lisäämällä uusia kenttiä tai muuntamalla

tapaa, jolla proseduraalinen generointi toimii (Stegner 2021). Binding of Isaac -pelin tapauksessa pelissä on saavutuksia, jotka avaavat uusia tavaroita, haasteita, huoneita sekä kenttiä tuleviin pelikertoihin. Nämä saavutukset joissain tapauksissa myös vaikuttavat proseduraaliseen generointiprosessiin yleensä vaikeuttamalla tai laajentamalla peliä. Pelissä liikutaan erimuotoisten huoneiden läpi, jotka täytetään tavaroilla, vihollisilla ja esteillä. Huoneet yhdistyvät yhteneväksi kokonaisuudeksi, joka on näkyvillä pelikarttana pelin oikeassa ylänurkassa (kuva 1, s. 13).

Pääosin manuaalisesti suunniteltu pelin sisältö, kuten esineet ja viholliset, on satunnaisesti aseteltu pelialueelle, mutta pelikartta on proseduraalisesti generoitu. Tämän takia pelaaja ei voi tarkalleen ennustaa oikeaa suuntaa, vaan joutuu matkustamaan labyrinttimaisten huoneiden läpi. Kiertely huoneesta toiseen jatkuu, kunnes pelaaja löytää kyseisen pelitason sisältämän päävihollishuoneen, josta päästään seuraavalle pelitasolle.

Binding of Isaac -pelissä proseduraalista generointia käytetään pelikartan luomiseen. Pelikartta luodaan yhdistämällä erimuotoisia huoneita, jotka ovat valmiiksi suunniteltuja (Rebirth Wiki 2018). Tässä analyysissä tullaan käyttämään Binding of Isaac -pelin peliversiota 1.7.7, joka julkaistiin 4.12.2021.

Prosessi, jolla kartta luodaan, on yllättävän yksinkertainen. Ensiksi päätetään pelikartan koko laskemalla huoneiden määrä. Tämä päätetään katsomalla pelaajan nykyinen peli- ja vaikeustaso. Tämän jälkeen itse pelikartan rakennus aloitetaan tekemällä polku pelaajan aloitushuoneen ja lopetushuoneen välillä. Toiseksi lisätään huoneita, jotka johtavat umpikujaan pelaajan aloitus- ja lopetushuoneen väliin. Viimeisenä asetetaan erikoishuoneita, kuten kauppoja ja haastehuoneita, umpikuja- ja sivupolkujen päihin (Rebirth Wiki 2018). Tämän prosessin voi nähdä vaiheittain kuvassa 2.



Kuva 2. Binding of Isaac -pelin proseduraalisen pelikartan rakennusvaiheet. Punaisella merkataan alueita, joihin ei voi tulla huoneita seuraavassa vaiheessa. Vaihe 4 on valmistunut pelikartta.

Jokainen generointivaihe vaikuttaa jollakin tavalla muihin vaiheisiin. Tämän lisäksi eri generointivaiheissa on virhetarkistuksia, joiden avulla vääriä tuloksia vältetään. Tällaisia ongelmatilanteita voivat olla esimerkiksi umpikuja- tai erikoishuoneiden osuminen toisiinsa tai kahden eri huoneen laittaminen samaan kohtaan. Isomman proseduraalisen generoinnin jakaminen neljään osaan mahdollistaa helpomman virheentarkistuksen ja auttaa peliä välttämään mahdollisia ongelmia, joita siinä muuten voitaisiin kohdata proseduraalista sisältöä käytettäessä.

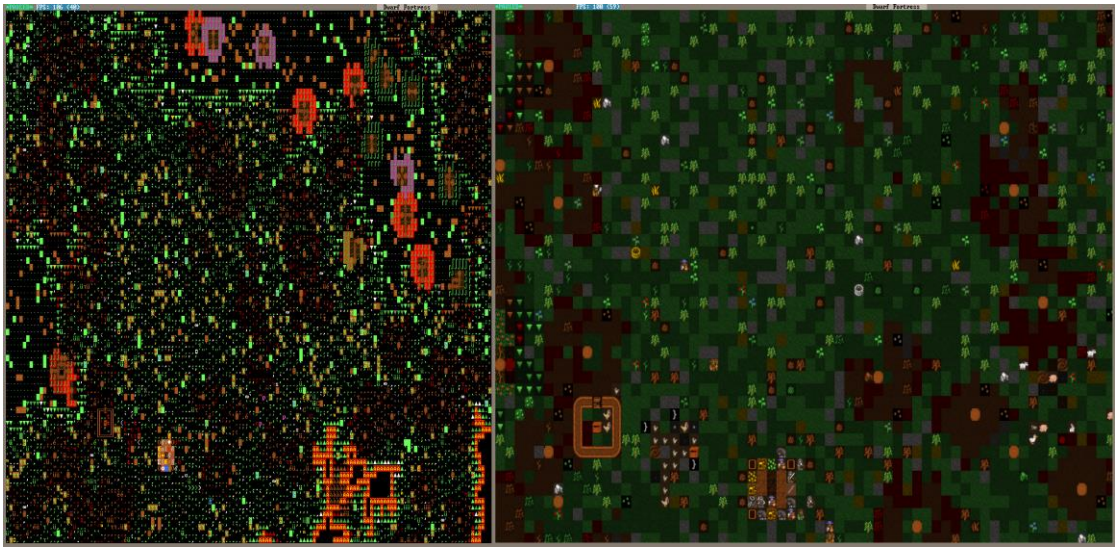
4.2 Dwarf Fortress

Dwarf Fortress on kaupunkisimulaattori-genreen kuuluva peli, joka julkaistiin vuonna 2006 (Bay 12 Games 2006). Pelissä pelaaja yrittää selvitä mahdollisimman pitkään ohjaamalla ryhmää kääpiöitä. Kääpiöiden avulla rakentuu linnoitus ja samalla sivilisaatio. Peli on hiekkalaatikkopeli eli pelillä ei ole voittotapaa, vaan sen sijaan peli jatkuu, kunnes pelaaja häviää tai kyllästyy peliin. Peli on kuitenkin tullut tunnetuksi pelin laajan simulaatioskaalan vuoksi, jossa simuloidaan muun muassa uskontoa, persoonien kehitystä, kulttuuria ja virtuaalisia ekonomiasysteemejä.

Pelikartta on jaettu ruutuihin, joissa voi olla samanaikaisesti useita tavaroita tai hahmoja. Tämän lisäksi pelialueessa on useita korkeuskerroksia, jotka on merkitty z-kerroksiksi. Jokainen z-kerros kuvastaa kahdeksan metrin korkeutta ja yhdellä pelikartalla voi olla maksimissaan 600 z-kerrosta, joista ainakin 15 jätetään vapaaksi tyhjälle taivastilalle (DF-Wiki 2014a). Kaikki nämä tasot on

simuloitu samalla tavalla, minkä takia pelaaja pystyy rakentamaan jättimäisiä maanalaisia tunnelisysteemejä tai pilvenpiirtäjiä.

Simulaation laajuudella on kuitenkin omat haittapuolensa. Ensinnäkin peli on erittäin vaikea oppia uuden pelaajan kannalta sen laajuuden vuoksi. Systeemit eivät ole helposti ymmärrettäviä ja usein hävitessä on vaikeaa ymmärtää, mitkä valinnat aiheuttivat pelin loppumisen. Tämän lisäksi pelin grafiikka on 1990–2000-luvun tasoa. Grafiikkataso on pidetty erittäin yksinkertaisena helpottaakseen laitevaatimuksia simulaation tarkkuuden jo valmiiksi aiheuttamien korkeiden vaatimusten vuoksi (Hall 2014). Pelaajat ovat kuitenkin tehneet grafiikkapaketteja, jotka muuttavat pelin ulkonäköä pelaamisen helpottamiseksi (Kuva 3).



Kuva 3. Pelinäkymä Dwarf Fortress -pelistä. Vasemmalla puolella on normaali pelinäkymä ja oikealla puolella on sama pelitilanne Phoebus-grafiikkapaketin kanssa.

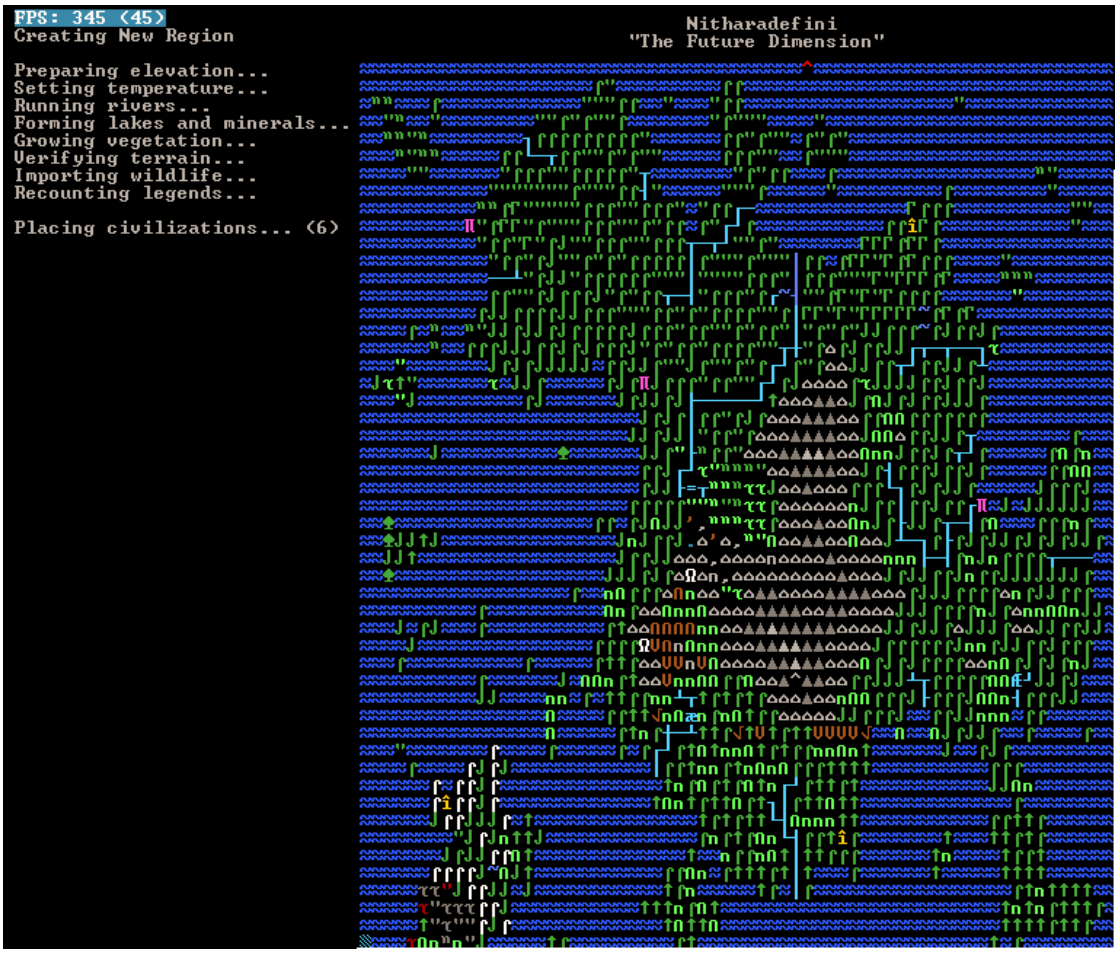
Pelin proseduraalisen generoinnin skaala antaa pelaajille loputtoman määrän eri pelitilanteita ja maailmoja. Tämän takia peli on ottanut oman vahvan asemansa simulaatiopelien genrestä.

Pelin käyttämä proseduraalisen generoinnin laajuus on ongelmallista tutkimuksen kannalta, minkä takia rajaamme analyysin käsittelemään pelkästään pelimaailman generointia. Tämä rajaaminen pitää tutkimuksen kiinni laajemmassa kokonaisuudessa sen sijaan, että tutkimuksessa käytettäisiin aikaa pienempiin proseduraalisiin generointisysteemeihin, jotka perustuvat samoihin

pohjasysteemeihin, joita maailman generoinnissa käytetään. Tässä analyysissä tullaan käyttämään Dwarf Fortress -peliversiota 0.43.03, joka julkaistiin 22.5.2016

Pelissä generoidaan kokonainen maailma pelaajan generointiasetusten perusteella. Dwarf Fortress -pelin pelimaailma on ruutupohjainen, jossa jokainen pelimaailma on pienimmillään 17 x 17 ruutua ja suurimmillaan 257 x 257 ruutua kooltaan. Jokainen pelimaailman ruuduista myös itsenään sisältää 20 x 20-ruudukon, josta pelaaja valitsee aloituspaikansa.

Dwarf Fortress -pelin proseduraalinen generointi ei kuitenkaan generoi pelimaailmaa yhdellä kerralla, vaan generointiprosessi on jaettu yhdeksään eri vaiheeseen. Näiden vaiheiden lisäksi on viimeinen kymmenes vaihe, jossa generoidaan pelimaailmalle historiaa. Historian generointi voi myös pienellä tapaa vaikuttaa pelattavan pelimaailman ulkonäköön (kuva 4).



Kuva 4. Dwarf Fortressin maailman generointivaiheet ovat selkeästi esillä pelaajalle. Viimeisimmässä vaiheessa ollaan laittamassa sivilisaatioita maailmaan.

Pelaaja voi myös muokata kaikkia proseduraalisen generoinnin vaiheita asettamalla omia asetuksia maailman generointisääntöihin. Peliä varten on tehty suuri määrä virheentarkistuksia jokaisen generointivaiheen välille ja jos jokin pelaajan vaatimuksista ei toimi, siitä ilmoitetaan heti. Esimerkki varoitusviestistä: ”Ei pystytä laittamaan tulivuoria, koska maailmassa ei ole tarpeeksi vuoria.” Koska proseduraalisen generoinnin prosessit etenevät vaiheittain, myöhemmät vaiheet yleensä muokkaavat aiempien vaiheiden ruutuja loogisella tavalla: metsät tarvitsevat vettä lähistölle, aavikot syntyvät kuiville alueille sekä aiempi esimerkki, jossa tulivuoret yrittävät korvata vuoriston huippuja, mikäli näitä huippuja löytyy pelikartalta.

Viimeisenä asetetaan sivilisaatiot, joita voidaan käyttää pelaamiseen. Jokaisella eri sivilisaatiotyypillä on omat vaatimukset ruudun tyypistä. Sen takia on mahdollista muodostua tilanne, jossa sivilisaatiota ei pystytä asettamaan. Tällöin pelaaja saa varoitusviestin, jossa sanotaan maailman olevan niin sanotusti ”kuollut maailma”, jossa pelin pääpelimuotoa eli linnoitusmuotoa ei pysty pelaamaan.

Dwarf Fortress on leikannut jättimäisen proseduraalisen generointityön useisiin eri osiin. Se mahdollistaa helpon virheen tarkistuksen ja tiedon antamisen pelaajalle. Prosessin laajuutta ei juurikaan pysty ymmärtämään ennen kuin alkaa tarkastelemaan pelissä olevaa ”edistyneet maailmangenerointiasetukset” -kohtaa, jolloin voi laskemalla huomata, että näitä asetuksia on yli 70 (DF-Wiki 2014b). Tästä huolimatta pelaaja saa jokaisesta generointimuokkauksesta selvän palautteen pelivaroitusten kautta. Sen ansiosta pelaaja ei koskaan törmää tilanteeseen, jossa hän jäisi ihmettelemään, minkä takia pelimaailma ei generoidu odotuksen mukaisesti.

4.3 No Man’s Sky

No Man’s Sky on avaruusseikkailupeli, joka julkaistiin vuonna 2016 (Hello Games 2016). Pelissä pelaaja aloittaa satunnaisesti valitulla planeetalla ja pelaajan pitää mennä planeetalta toiselle parantaakseen avaruusalustaansa eri planeetoilta löydetyillä resursseilla matkalla kohti galaksin keskipistettä.

Peli ei kuulosta uniikilta, kun sen pääideasta kuulee lyhyen selostuksen. Se, mikä tekee pelistä uniikin, on proseduraalisen generoinnin käyttäminen. Pelin galaksit, olennot ja avaruusalukset ovat kaikki tehty proseduraalista generointia käyttäen. Tämä oli isona osana pelin mainostuskampanjaa ja lainaten pelin teknistä suunnittelijaa Sean Murrayta: ”Pelin täysi läpikäyminen vaatisi 584 miljardin vuoden peliajan” (Kharpal 2016). Pelin mainostus luonnollisesti keskittyi generoituihin ympäristöihin ja eläimiin (kuva 5).

Kuva 5. Yksi mainostuksessa käytetyistä kuvista, joka näyttää mahdollisia ympäristöjä ja eläimiä, joita voi proseduraalisesti generoida No Man’s Sky -pelissä (Hello Games 2016).

Pelin laajuutta on toisin sanoen mahdotonta edes kuvitella ja peli saikin suurta huomiota juuri näin mittavassa skaalassa onnistumisen takia. Tästä huolimatta peli sai paljon kritiikkiä julkaisussaan, koska useat pelaajat totesivat nopeasti, että pelin mekaniikat olivat toistuvia eikä generointi pystynytkään kantamaan peliä odotetulla tavalla (Kuchera 2016).

No Man’s Skyn peliskaalan takia on erittäin vaikeaa alkaa tutkimaan proseduraalista generointia kokonaisuudessaan, minkä takia analyysi keskittyy pelin olentojen ja planeettojen generointiin. Tällä tavalla pelin proseduraalisen generoinnin käytöstä saadaan hyvä yleiskuva ja tutkimuksen skaala pysyy järkevänä. Tutkimuksessa käytetään No Man’s Sky -pelin julkaisuversiota eli 1.03:sta, joka julkaistiin 12.8.2016.

No Man’s Skyn planeetan luonti toimii vaiheittain. Ensin planeetan koko määritetään satunnaisesti, jonka jälkeen lisätään ympäristö planeetan tyypin mukaisesti ja sitten lisätään proseduraalisesti generoitu kasvisto ja eläimistö. Viimeisenä asetetaan resurssit, joita pelaaja käyttää aluksensa päivittämiseen ja liikkumiseen. Edellisten vaiheiden tulokset vaikuttavat seuraavien vaiheiden tuloksiin ja systeemi toimii suurimmalta osalta hyvin, mutta pelaajat huomasivat erittäin paljon toistoa planeetan maastossa ja ympäristössä pelin koosta huolimatta. Tämä voi tarkoittaa, että generoinnin säännöt ovat liian tiukat tai generointi on liian suppea (GDC 2015).

Eläimien generointi sen sijaan on jaettu kuuteen eri vaiheeseen. Ensimmäiseksi valitaan eläimen pohja, minkä jälkeen pohjaan lisätään yksityiskohdat, kuten sarvet, korvat tai nenät. Seuraavaksi lisätään kerrostuma, johon kuuluu mahdolliset ihot, höyhenet ja suomut. Samanaikaisesti luun skaalaus aloitetaan, minkä myötä mallin perusluuston kokoa muutetaan. Tämä prosessi pienentää tai paisuttaa mallia. Tämän jälkeen lisätään käyttäytymislogiikka ja viimeiseksi vaiheet käydään uudestaan läpi, jotta voidaan vähentää äärimmäisiä piirteitä. (GDC 2015.) Prosessi kuulostaa järkevältä ja sisältää virheentarkastelun, kuten planeettojen ympäristökin, mutta syntyvät olennot saattavat olla outoja (kuva 6).

Kuva 6. No Man’s Skyn proseduraalisen olentoluonnin outoja tuloksia, jotka eivät näytä loogisilta tai sovi ulkonääöltään pelimaailmaan (Crowbcat 2016).

Mistä tämä ongelma sitten johtuu? Virheentarkistus ja generointi toimivat parhaiten, kun voidaan asettaa selkeitä sääntöjä, jotka ovat loogisia, matemaattisia tai mekaanisia. Ulkonäkö ei kuitenkaan kuulu näihin, minkä takia tarkan

generointiprosessin luominen on erittäin vaikeaa. Tämän lisäksi planeettojen generointi toimii täydellisesti, vaikka tulokset olivatkin usein samankaltaisia. Tästä voidaan todeta, että proseduraalista generointia kannattaa käyttää harkitusti, mikäli generointiprosessia ei pystytä määrittämään järkevällä tavalla. Proseduraalinen generointi toisin sanoen toimii parhaiten silloin, kun generoidaan mekaanisiin, loogisiin tai matematiikkaan perustuvia asioita, joissa on mahdollista määrittää säännöt tarkasti (Carpenter 2011).

4.4 Yhtenäisyydet

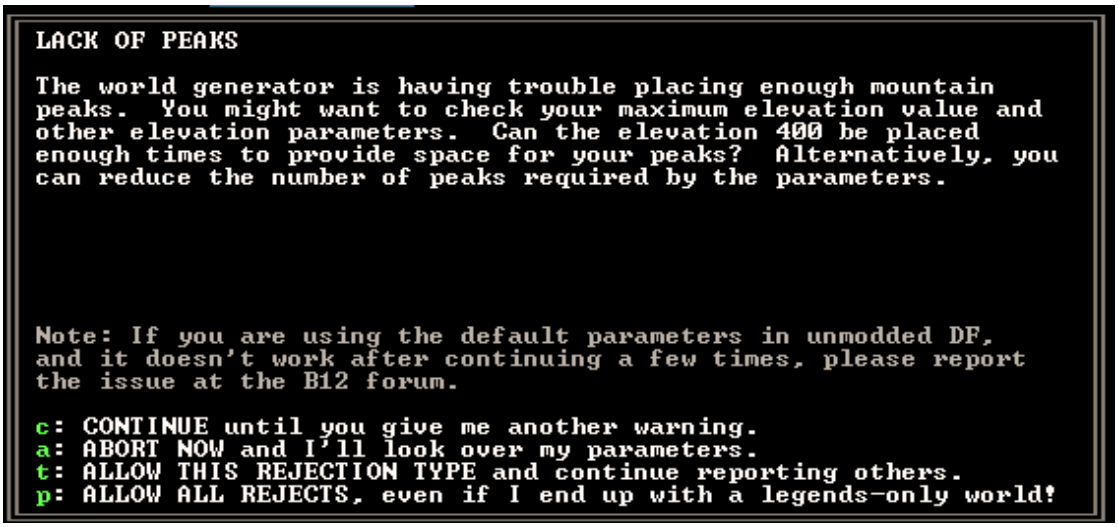
Aiheen kannalta ensioletus oli, että yhtenäisyyksien löytyminen olisi vaikeaa, minkä takia suunnitteluvaiheessa varauduttiin käsittelemään aiheita pienemmässä skaalassa. Tämä oletus on kuitenkin todistettu vääräksi ja pelkkä ensisilmäys yksittäisiin tutkimuksiin osoittaa yhtenäisyyksiä.

Ensimmäisenä kuitenkin kannattaa käydä läpi yhtenäisyydet tutkimuskysymyksen kannalta. Kysymykset ovat ”Mitä mahdollisia ongelmia proseduraalisesti generoidussa sisällössä on?” ja ”Miten näitä ongelmia voi välttää?”. Ensisilmäyksellä vaikuttaisi, että vain No Man’s Sky -pelin proseduraalisessa generoinnissa oli ongelmia, mutta todellisuudessa muut valitut tutkimuspelit vain välttävät vastaavat ongelmat virhetarkistuksen tai tarkkaan määritellyn ja suunnitellun generointiprosessin avulla.

Havaintojen perusteella ongelmia proseduraalisen generoinnin käyttämisestä tulee selkeästi silloin, kun halutaan tuottaa monimutkaista sisältöä proseduraalisen generoinnin avulla, jossa tulosta ei pysty helposti rajaamaan. Täysin loogisissa tai matemaattisten sisällön tapauksissa, kuten pelikartoissa, ympäristön tai huoneiden rakenteissa, pystytään virhetarkistukset määrittämään käyttämällä matematiikkaa ja logiikkaa. Tällä tavalla pelaaja ei koskaan törmää ongelmatuotoksiin. Ongelmat syntyvät kuitenkin sisällöstä, jota ei voida yksinkertaisesti tarkastamaan matematiikan tai logiikan avulla. No Man’s Skyn tapauksessa proseduraalista generointia käytettiin pelinsisäisten hahmojen ja grafiikkojen luomiseen. Tietokoneelle ei ole kuitenkaan luotu mitään tarkistusprosessia, jolla varmistettaisiin, että nämä proseduraalisesti tehdyt hahmot

näyttäisivät pelaajan silmissä loogisilta (Kuchera 2016). Tällaisten tarkastuksien tekeminen rajoittamatta mahdollisia pelihahmoja liikaa on erittäin vaikeaa. Tämän takia on parasta rajoittaa proseduraalinen generointi tilanteisiin, jossa pystytään luomaan selkeitä generointisääntöjä.

Kaikissa peleissä esiintyi yhtenäinen virheentarkistus, jolla vahvistettiin, että pelin pelattavuus ei kärsi proseduraalisesta sisällöstä. Tämä tarkistus oli luultavasti selkein pelissä Dwarf Fortress. Pelissä annettiin selkeä viesti maailman generointiongelmista, minkä avulla pelaaja voi muokata generointiasetuksia (kuva 7).



Kuva 7. Dwarf Fortress -pelin maailman generoinnin virheilmoitus. Ilmoitus antaa tarkat tiedot kyseisen vaiheen muodostamista ongelmista ja ehdotuksia virheiden välttämiseksi.

Proseduraalisen generoinnin toteutuksen kannalta isot generointiprosessit leikattiin pienempiin osiin toimivampaa toteutusta varten. Tämä prosessin hajauttaminen näkyi kaikissa tutkituissa peleissä. Tämä pienempiin osiin leikkaaminen on selkeästi tehty tarkistuksen helpottamiseksi ja ongelmatilanteiden välttämiseksi.

Tämän perusteella voidaan todeta, että oleellinen osa proseduraalisen generoinnin toteuttamista on generointien hajauttaminen pieniksi aihekohtaisiksi vaiheiksi. Työprosessi voidaan nähdä kolmivaiheisena. Ensiksi generoidaan haluttu asia. Toiseksi tehdään virhetarkistus, joka on tarkkaan suunniteltu generoidun aiheen, prosessin ja käytön kannalta. Viimeisenä virhetarkistuksen

tuloksen perusteella aloitetaan joko alusta tai jatketaan seuraavaan vaiheeseen. Tämän jälkeen voidaan aloittaa uusi generointityö, jossa käytetään aiempaa työtä pohjana. Tallentamalla aiemmat vaiheet on mahdollista palata niihin, mikä vähentää suoristusvaatimuksia tietokoneen kannalta ja auttaa mahdollisissa ongelmatilanteissa.

Lyhyesti sanottuna prosessi on ”generoi, tarkista ja etene”. Tämä proseduraalisen generoinnin suunnittelumalli vaikuttaa olevan valmiiksi käytettävissä mihin tahansa proseduraalista generointia vaativaan aiheeseen. Ainoa haaste toteutuksen kannalta on tarpeeksi laajan virhetarkistuksen rakentaminen jokaista generointivaihetta varten.

5 TYÖKALUT

Tämä luku esittelee työkalut, ominaisuudet ja tekniikat, joiden käyttäminen on oleellinen osa kehittämistyön suunnittelemista ja toteutusta. Analyysi tapahtuu arvioimalla aihetta yleisesti, mutta pääosin keskittymällä Unity-pelimoottorin osiin, jotka ovat relevansseja kehitystyön toteuttamisen ja suunnittelun kannalta. Analyysi tulee ensin keskittymään Unity-pelimoottoriin, jonka jälkeen seuraa C#-ohjelmointikielen käyttö Unity-pelimoottorissa. Analyysin tavoitteena on mahdollistaa vaivaton ja tehokas työnkulku suunnittelu- ja toteutusluvuissa. Opinnäytetyössä käytetään Unity-pelimoottorista versiota 2021.3.16f1, joka julkaistiin vuonna 2021.

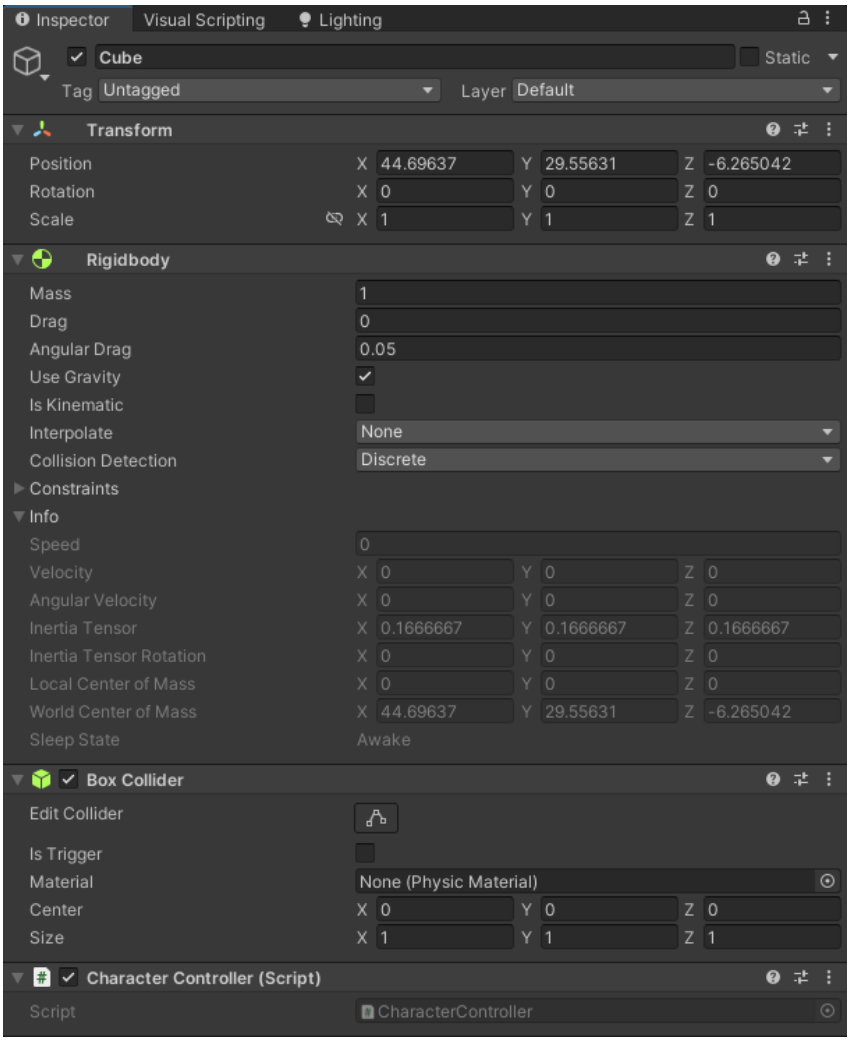
Ohjelmointianalyysissa tarkastellaan proseduraalista generointia ohjelmoinnin kannalta ja sitä, mitä Unity-pelimoottori tarjoaa C#-ohjelmointikielellä opinnäytetyöhön. Ohjelmointi voi olla erittäin vaikea aihe analyysin ja kuvailun kannalta, minkä takia luku keskittyy antamaan perusidean eri toiminnoista ja selventämään, miten niitä voidaan hyödyntää yksityiskohtaisen analyysin sijaan. Tavoitteena on siis antaa lukijalle ymmärrys opinnäytetyössä käytettävästä ohjelmoinnista ja syistä, jotka johtivat näiden tekniikoiden valitsemiseen.

5.1 Unity-pelimoottori

Unity-pelimoottori julkaistiin ensimmäistä kertaa vuonna 2005 Applen kehittäjäkonferenssissa. Unityn tavoitteena oli mahdollistaa pelinkehitys saatavaksi

kaikille pelinkehittäjillä varallisuudesta huolimatta. Unityn globaalien viestintätoimintojen johtaja Marcos Sanchez sanoi, että yhtiön tavoitteena oli aina pelinkehityksen demokratisoituminen (Axon 2016). Tämä lähestymistapa selkeästi onnistui yhtiön kannalta, sillä Unity-pelimoottori kehittyi yhdeksi suosituimmista pelimoottoreista seuraavan vuosikymmenen aikana, mikä näkyi selvästi käyttäjätilastoissa sekä mediassa (Takahashi 2022).

Unity-pelimoottorin kehitysympäristö perustuu peliobjekteihin, jotka rakentuvat komponenteista (ECS FOR UNITY s.a.). Komponenttien avulla voidaan hallita peliobjektien käyttäytymistä hyödyntämällä joko omia koodeja eli skriptejä, joita pystyy luomaan C#-ohjelmointikielellä tai käyttämällä Unity-pelimoottorin valmiiksi rakennettuja osia, kuten fysiikka-, animointi- tai äänisysteemiä (Coding in C# in Unity for beginners s.a.). Jotkut komponentit ovat automaattisesti kaikilla peliobjekteilla kuten Transform-komponentti, joka sisältää peliobjektin paikka-, rotaatio- ja skaalatiedot.



Kuva 8. Inspector-työkalu, jolla pystytään tarkastelemaan, muokkaamaan ja muuttamaan peliobjektien komponentteja.

Näitä komponentteja voi lisätä tai manipuloida käyttäen Inspector-paneelia, joka näkyy kuvassa 8. Manipulointimahdollisuudet luodaan skriptin sisällä laittamalla eri komponenttien muuttujien arvot avoimeksi, jolloin Inspector voi löytää nämä muuttujat ja muokata arvoja Inspector-ruudun kautta. Muuttujat on ohjelmoinnissa käytetty termi, joka tarkoittaa tietoa tai arvoa, jota ohjelma käsittelee. Tämä mahdollistaa skriptien toteuttamisen pelisuunnittelijoita varten, mikä tarjoaa sujuvan työnkulun eri projekteja varten.

Komponentit perustuvat ajatukseen siitä, että jokaisesta halutusta toiminnosta voidaan tehdä oma komponenttinsa (Coding in C# in Unity for beginners s.a.). Esimerkiksi pelaajahahmo voisi rakentua neljästä komponentista, jotka olisivat Unity-pelimoottorin sisältämät Transform-, Rigidbody- ja Box Collider -komponentit sekä ohjelmoijan tekemä Character Controller, joka hallitsisi pelihahmoa (kuva 8). Näistä komponenteista Unityn Transform sisältää hahmon paikan, Rigidbody mahdollistaa fysiikkasysteemin käytön ja Box Colliderin avulla hahmo voi tunnistaa törmäykset, joita Rigidbody hyödyntää fysiikkalaskuissa (Unity User Manual: Important Classes s.a.). Komponenttisysteemin etuna on se, että komponentti pitää tehdä vain kerran, minkä jälkeen sen voi liittää mihin tahansa peliobjektiin. Tämä lisää aina saman käyttäytymistavan.

Komponentit voivat myös sisältää viittauksia muihin komponentteihin tai muihin peliobjekteihin (Coding in C# in Unity for beginners s.a.). Tämä mahdollistaa eri komponenttien yhteistyön ja tiedon välityksen. Tätä tiedonvälitystä varten on myös tehty Unity-pelimoottorin ohjelmoitavat objektit, jotka ovat globaalisia objekteja. Globaalisuuden ansiosta niihin voidaan viitata mistä tahansa ja jokainen ohjelmoitava objekti on uniikki eli kaikki viittaukset osoittavat samaan objektiin ja samaan dataan, jota objekti sisältää. Tämä mahdollistaa tiedon välittämisen järjestelmällisesti eri peliobjektien välillä, mikä myös välttää tietyt virhetilanteet, joita voi muodostua, jos viitattu peliobjekti ei olekaan samassa kohtauksessa.

Kohtaukset ovat Unity-pelimoottorissa oleva järjestelmä, joka mahdollistaa pelin tai ohjelman pilkkomisen pienempiin osiin, kuten esimerkiksi pelitasoihin

(Unity User Manual: Scenes s.a.). Kaikki kohtaukset sisältävät koordinaattijärjestelmän, jota peliobjektien Transform-komponentti käyttää laittaakseen peliobjektit oikeille asemille kohtauksessa. Kohtaukset yleensä täytetään peliobjekteilla halutulla tavalla, minkä jälkeen kohtaus on käytettävissä. Kohtauksen pystyy sen valmistumisen jälkeen halutessaan lataamaan käyttämällä kohtausmanageria, jonka avulla useat kohtaukset voidaan yhdistää yhtenäiseksi ohjelmaksi kohtauksen lataustoimintoa käyttämällä. Nämä kohtaukset voidaan tallentaa esimerkiksi listan sisälle, mikä mahdollistaa helpon lataamisen. Lista ohjelmoinnissa tarkoittaa tietorakennetta, joka sisältää useita saman tapaisia arvoja tai objekteja. Lista mahdollistaa asioiden helpon käsitellyn ja järjestelyn. Listoja usein käytetään silmukoissa, jotka toistavat samoja käskyjä haluttuun pisteeseen asti.

5.2 C#-ohjelmointikieli

C#-ohjelmointikieli on moderni objekteihin eli tiedon ja toimintojen kapselointiin perustuva kieli, joka toimii pohjana Unity-pelimoottorin komponenttisysteemille. C#-ohjelmointikielen esimerkkiominaisuuksiin kuuluu poistuneiden objektien vapauttaminen muistista eli automaattinen roskienkeruu, tyypin turvallisuus eli tiukat muuttujatyyppisäännöt, jotka välttävät tyyppivirheitä ohjelman käynnin aikana, sekä asynkroninen ohjelmointi eli useiden operaatioiden samanaikainen suorittaminen (Microsoft 2022a). Näiden ominaisuuksien lisäksi C# on suunniteltu helposti ymmärrettäväksi ohjelmointikieleksi, minkä takia se on ihanteellinen pelinkehitystä varten.

Unity, usean muun vastaavan ohjelman tapaan, tarjoaa omia ”kirjastoja” C#-ohjelmointikielelle (Unity User Manual: Overview s.a.). Kirjastot ohjelmoinnissa tarkoittavat esirakennettua funktioita eli eri koodipätkiä, joihin voidaan viitata ja käyttää uudelleen eri ohjelmissa. Kirjaston tavoitteena on tarjota yhtenäiset toimintaperiaatteet ohjelmiston kehittäjille ja nopeuttaa kehitysaikaa, minkä lisäksi samalla pystytään välttämään koodin toistamista. Unity-pelimoottorin kirjastot luonnollisesti tarjoavat perustoiminnot, joilla Unity-pelimoottorin komponenttisysteemi yhdistetään C#-ohjelmointikieleen. Kuva 9 antaa tarkemman vaikutelman kirjastojen toiminnasta.

```
using UnityEngine;
using UnityEngine.UI;
using System;
```

Kuva 9. Kirjastoviittaus C#-skriptin sisällä. Kirjastoviittaukset mahdollistavat kirjastojen sisällä olevien toimintojen käyttämisen. Valkoisella merkatut viittaukset tarkoittavat, että viittausta käytetään koodin sisällä, minkä takia viittausta ei voida poistaa.

Otetaan kirjastosta esimerkiksi kaksi kirjastoviittausta, jotka ovat System ja UnityEngine. System on C#-peruskirjasto, joka sisältää perusominaisuuksia kuten virheenkäsittelyä, matemaattisia toimintoja ja muistihallintaa (Kuva 9). Esimerkiksi $Math.Sqrt(numero)$ on automaattinen funktio, joka laskee neliöluvun annetusta numerosta. Funktiot tai menetelmät ovat yksinkertaisesti tapoja käsitellä tietoa määritetyllä tavalla. UnityEngine on tärkein kirjasto Unity-pelimoottorissa ja sisältää fysiikka-, audio-, grafiikka- ja Monobehavior-luokan sekä peliobjekti- ja komponenttisysteemit (Unity User Manual: Overview s.a.). UnityEngine sisältää lisäksi monia muita toimintoja, joiden käyttäminen on oleellinen osa ohjelman kehittämistä Unity-pelimoottorissa. Esimerkiksi $void\ Start(){}$ on funktio, jota Monobehavioria käyttävät peliobjektit käsittelevät automaattisesti ensimmäistä kertaa, kun peliobjekti käynnistyy. Unity-komponentin pystyy luomaan C#-kielellä perimällä Monobehavior-luokan.

```
public class RoomRecovery : Recovery<GameObject>
```

Kuva 10. Perintä luokan sisällä tapahtuu käyttämällä merkkiä ”:”, jota seuraa perittävän luokan nimi. Kuvan esimerkissä luokka RoomRecovery perii Recovery-luokan toiminnot.

Perintä ohjelmoinnissa tarkoittaa luokan perustoiminnallisuuksien kopioimista toiseen luokkaan, joka tällä tavalla pystyy käyttämään ja muokkaamaan niitä (Microsoft 2022b). Kun komponentti on luotu, siihen voidaan helposti viitata C#-kielessä viittaamalla komponentin nimeen. Jos luokan nimi on ”Pelaaja”, tehdään viittaus Pelaajaan ja lisätään paikallinen nimi perään. Viittausten tekeminen eri muuttujiin, luokkiin ja toimintoihin on oleellinen osa ohjelmointityötä ja mahdollistaa eri koodien yhteistyön.

Perintää voi myös hyödyntää omissa luokissa, ja onkin olemassa useita eri tapoja hyödyntää perinnän perusideaa eri tekniikkoja hyödyntäen. Esimerkkinä tällaisesta vaihtoehdosta perinnälle on ”Interface” eli rajapinta tai Abstrakti-luokka. Alkuperäisessä luokassa määritetään funktiot, joita sitä hyödyntävien

muiden luokkien pitää käyttää. Tämä mahdollistaa yhteiset toiminnot rajapintaa hyödyntävien luokkien välillä, mikä tekee yhteisistä viittauksista luokkia hyödyntämällä mahdolliseksi. Rajapintoja ja abstrakteja luokkia voi ajatella sopimukseksi, joka vaatii tiettyjen toimintojen lisäämisen ja auttaa tällä tavalla ohjelmoijaa ohjelman yhtenäisten toimintojen rakentamisessa. Tämän lisäksi se auttaa välttämään ongelmatilanteita, joita voi syntyä ohjelmoinnissa eri lähestymistyylien takia.

```
public class CharacterController : MonoBehaviour
{

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        if (Input.GetKeyUp(KeyCode.D))
        {
            this.transform.Translate(new Vector2(1, 0));
        }
    }
}
```

Kuva 11. Monobehavior-luokka mahdollistaa hahmon liikkumisen Update-toiminnon sisällä. Kuvan tapauksessa ohjelma odottaa, että pelaaja nostaa ylös näppäimen D ennen kuin hahmoa liikutetaan.

CharacterController eli hahmonohjaus on laajempi esimerkki komponentista, joka mahdollistaa hahmon liikkeen nuolinäppäimillä (kuva 11). Update-funktio tapahtuu jokaisen kehyksen eli jokaisen kuvan piirtämisen jälkeen, minkä takia Update-funktiota voidaan käyttää pelaajan näppäinten tarkistukseen. Pelaajan painettua D-näppäintä toteutetaan hahmon liikutus, jossa hahmoa liikutetaan käyttäen Transform-komponentin koordinaatistosysteemiä yhden yksikön oikealle X-akselissa. Komponenttien lisäksi Unity tarjoaa muita toimintoja, joita voidaan hyödyntää C#-ohjelmointikieltä hyödyntäen, kuten aiemmin mainitut ohjelmoitavat objektit.

```
public abstract class Rule<T> : ScriptableObject
{

    public abstract bool Check(List<T> lists);
}
```

```
[CreateAssetMenu(fileName = "MinRoomCount", menuName = "Rules/MinimumRoomCount")]
Unity Script | 0 references
public class MinRoomCount : Rule<GameObject>
{
    [SerializeField] private int MinRooms=5;
    2 references
    public override bool Check(List<GameObject> rooms)
    {
        if (rooms.Count < MinRooms)
        {
            return false;
        }
        return true;
    }
}
```

Kuva 12. Ohjelmoitava objektikoodi, jossa toteutetaan ohjelmoitavan objektin abstraktiluokka ”Rule”, ja jonka toteutus tapahtuu ”MinRoomCount”- luokan sisällä. CreateAssetmenu-komento luo menun, jonka kautta ohjelmoitava objekti voidaan luoda.

Ohjelmoitava objekti perustuu perintään samalla tavalla kuin Monobehavior-luokan perintä. Objektin käyttäytyminen kuitenkin alkaa erota sen viittauksien ja käytön kanssa. Koska ohjelmoitavaa objektia ei kiinnitetä kohtaukseen tai peliobjekteihin, se ei itsenäisesti koskaan käynnisty. Tämän sijaan objektiin pitää viitata Monobehavior-luokkaa käyttävässä komponentissa, minkä avulla ohjelmoitavaa objektia voidaan hyödyntää (Unity User Manual: ScriptableObject s.a.).

Isoimpana etuna on yhtenäisen tiedon helppo jakaminen viittauksen kautta. Tämän lisäksi, koska ohjelmoitavia objekteja ei liitetä kohtaukseen, ne ovat aina saatavina toisin kuin yksittäiset komponentit. Toinen ero on se, että ohjelmoitava objekti pitää luoda ennen käyttöä toisin kuin komponentit, joita voidaan vapaasti liittää peliobjekteihin.

Ohjelmoitavan objektin luomisprosessi on yksinkertainen. Ensiksi tehdään koodi, joka perii ScriptableObjectin, johon laitetaan halutut menetelmät ja viittaukset. Tämän lisäksi objekti tarvitsee tavan, jolla se luodaan. Yleisimmin tämä toteutetaan CreateAssetMenu-käskyn avulla, mikä lisää ohjelmoitavan objektin haluttuun menuun (kuva 12). Tämän jälkeen mennään Unity-pelimoottorin päänäkymään ja luodaan ohjelmoitava objekti, joka voidaan liittää

haluttuihin komponentteihin. Tärkeä huomio on se, että on mahdollista tehdä useita eri ohjelmoitavia objekteja samaa pohjaa käyttäen. Esimerkiksi jos ohjelmoitava objekti nimeltä ”auto”, jossa on muuttuja nopeus, voidaan tehdä auto1 ohjelmoitava objekti, jonka nopeus on 5 ja sitten auto2, jonka nopeus on 10. Tämän takia on tärkeää harkita tarkkaan, milloin kannattaa käyttää ohjelmoitavia objekteja, koska muuten on mahdollista, että projekti hukkuu eri ohjelmoitavien objektien määrään, mikäli ohjelmoitavia objekteja käytetään väärissä töissä.

Globaalisuus kuitenkin tuo omat haittansa, kuten ohjelmoitavien objektien muokkaamisen komponenttien kautta, koska jokainen ohjelmoitava objekti on ainutlaatuinen ja globaalisti viitattuna sen muokkaaminen voi helposti johtaa odottamattomaan käyttäytymiseen ja virhetilanteisiin. Tämä voi olla seurausta siitä, jos ohjelmoitavaa objektia muokataan yhden peliobjektin kautta, mutta toinenkin peliobjekti käyttää sitä ilman, että muokkauksien vaikutusta on otettu huomioon.

6 SUUNNITTELU

Tämä luku käsittelee kehittämistyön suunnittelua. Suunnittelua lähestytään kahdessa eri vaiheessa, joita seuraa suunnitelman analyysi ja pohdinta.

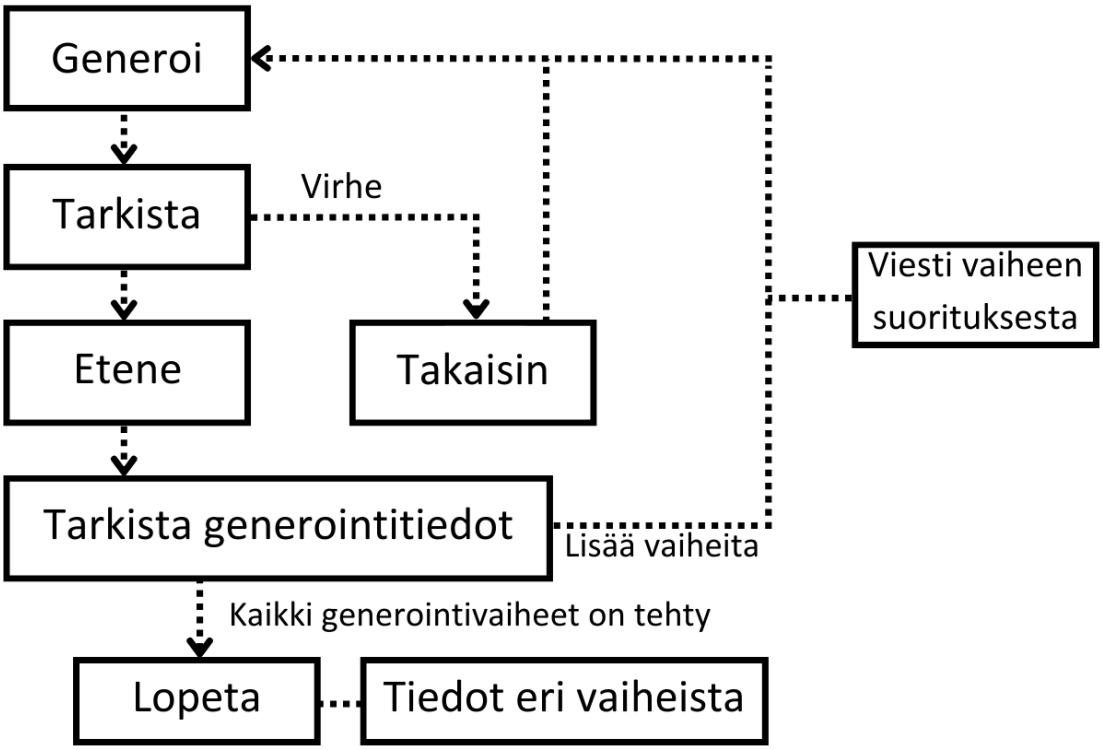
Ensimmäinen vaihe on toimintakaavion tekeminen. Toimintakaavion tarkoituksena on ohjelman toimintojen hahmotteleminen yleisesti eli se, miten ohjelma toimii vaihe vaiheelta. Tärkeä huomio toimintakaavion suunnittelun kannalta on se, että toimintokaavion tarkoituksena on antaa geneerinen toimintamalli ohjelmalle. Täten tässä vaiheessa ei pitäisi olla viittauksia Unity-pelimoottoriin tai sen osiin. Tämä lähestymistapa mahdollistaa toimintakaavion suunnittelemisen neutraalilta kannalta, minkä ansiosta kehittämistyön voi periaatteessa toteuttaa missä tahansa ympäristössä, jossa on mahdollista toteuttaa kehittämistyön kaikki toimintavaiheet. Toimintakaavion tekeminen perustuu pääosin laadullisessa tutkimuksessa kerättyihin tuloksiin, jotka ovat osoittaneet täyttävänsä yleiset standardit proseduraalisen generoinnin suunnittelun kannalta.

Toimintakaavion rakentamisen jälkeen seuraa luku, jossa toimintakaavion perusteella hahmotellaan suunnitelma siitä, miten toimintakaavion malli toimii Unity-pelimoottorin sisällä. Tässä tullaan yhdistämään toimintakaavion suunnitelma Työkalut-luvussa kerättyihin tietoihin. Lopputuloksena pitäisi olla selvä kuvaus toimintamallin eri osien toiminnasta Unity-pelimoottorin sisällä, mikä mahdollistaa kehittämistyön toteutuksen.

Suunnitelman analyysissa ja pohdinnassa tarkastellaan tehtyä suunnitelmaa kriittisellä silmällä sekä yritetään muodostaa kuvaa suunnitelman tarjoamista mahdollisuuksista. Samalla on mahdollista analysoida, onko opinnäytetyön tutkimuskysymyksiin saatu alustavat vastaukset, jotta kehittämistyön toteuttamiseen jatkaminen on mahdollista.

6.1 Geneerinen suunnitelma

Kerättyjen tietojen perusteella on oleellista, että työkalu on modulaarisesti toteutettu. Nämä modulaariset pienet osat pitää pystyä helposti yhdistämään toisiin modulaarisiin osiin ja virhetarkastusprosessiin, jonka avulla voidaan palata aiempaan vaiheeseen. Käyttäjälle pitää pystyä antamaan selvät viestit vaiheiden onnistumisen tai epäonnistumisen kannalta, jotta työkalun hyödyntäminen yksinkertaistuu. Näiden tietojen perusteella malli toimii kuvan 13 mukaisesti.



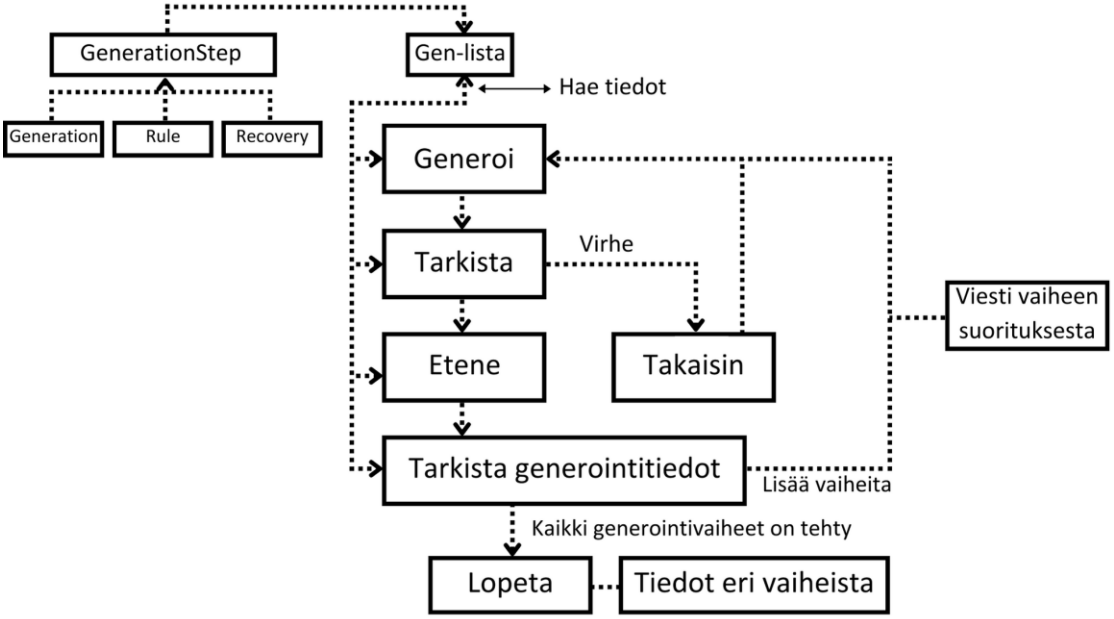
Kuva 13. Geneerinen generointimalli

Prosessissa on yksi generoija, jonka päätehtävänä on yhdistää generoinnin eri vaiheosat yhtenäiseksi prosessiksi. Itse generointi tapahtuu edelleen käyttämällä aiemmin esiteltyä mallia ”generoi, tarkista ja etene”, jonka avulla pystymään varmistamaan, että lopputulos pysyy sääntöjen mukaisena. Säännöt, generointi ja tarkistus pitäisi pystyä yhdistämään jollakin tavalla yhdeksi vaiheeksi, joka mahdollistaa helpon palikkamaisen generointirakentamisen. Jokaisen generointivaiheen jälkeen sääntöä käytetään tuloksen vertaamisen. Suunnitelman tekeminen on selkeää aiemmin tehdyn tutkimuksen ansioista ja tämän vaiheen suunnitelmaa pystytään soveltamaan yleisesti missä tahansa generointityössä. Opinnäytetyön skaalan takia suunnitelmaa tullaan soveltamaan vain yhdessä pelimoottorissa eli aiemmin mainitussa Unity-pelimoottorissa.

6.2 Suunnitelman käyttö Unity-pelimoottorissa

Toimintamallin muuttaminen Unity-pelimoottorissa toimivaan prosessiin on yksinkertainen tehtävä, kun tiedossa ovat jo tärkeimmät työkalut, joita tullaan käyttämään toteutuksessa. Unity-pelimoottorin kannalta prosessi on parhainta ajatella sen mukaan, miten toimintamallin pystyy toteuttamaan eri komponent-

tien ja systeemien kannalta. Tämän toteutuksen kannalta hyödynnetään komponenttisysteemiä, C#-ohjelmointikielen rajapintoja ja luokkia, ohjelmoitavia objekteja sekä Inspector-työkalua.



Kuva 14. Generointimalli Unity-pelimoottorissa.

Proseduraalinen generointi tapahtuu täysin peliobjektin ”procedural generator” sisällä, joka ottaa Generator-nimisen C#-kielellä rakennetun komponentin ajamaan prosessia. Luokan sisällä on lista ”Gen”, joka ottaa sisäänsä GenerationStep-luokan jäseniä (kuva 15).

```
public class GenerationStep<T>
{
    public string stepTitle;
    public Generation<T> generationStage;
    public Rule<T> ruleCheck;
    public Recovery<T> restore;
}
```

Kuva 15. GenerationStep-luokka koostuu kolmesta ohjelmoitavasta objektista ja tekstikentästä, johon laitetaan askeleen otsikko.

GenerationStep-luokka koostuu kolmesta ohjelmoitavasta objektista, jotka rakennetaan abstraktia luokkaa käyttäen. Näiden objektien nimet ovat ”Generation”, ”Rule” ja ”Recovery”. Ideana on, että tehtyämme ensin generointivaiheen tarkistetaan generoinnin tulos ”Rule”-jäsentä käyttäen, jonka jälkeen aiempi vaihe palautetaan, jos sääntöä rikotaan. Tämä toteutustyyli myös mah-

dollistaa sääntövaiheen jättämisen tyhjäksi, mikäli generointivaiheen lopputuloksella ei ole väliä. Luokka sisältää lisäksi muuttuja-stringiin kuuluvan jäsenen otsikko. String-muuttuja tarkoittaa yksinkertaisuudessaan muuttujaa, joka sisältää tekstikentän. Otsikkoa käytetään virheilmoitusten ja vaiheiden suorittamisen ilmoittamiseen.

```
public abstract class Recovery<T> : ScriptableObject
{
    2 references
    public abstract List<T> Recover(List<T> currentList, List<GameObject> listcopy);
    2 references
    public abstract List<T> Record(List<T> currentListy);

}
public abstract class Generation<T> : ScriptableObject
{
    17 references
    public abstract List<T> Generate(List<T> lists);
}
public abstract class Rule<T> : ScriptableObject
{
    2 references
    public abstract bool Check(List<T> lists);
}
```

Kuva 16. Abstraktit luokat, joilla GenerationStep-luokan jäsenet toteutetaan. Jokainen luokista perii ohjelmoitavan objektin, jolloin tulevat luokat ovat myös ohjelmoitavia objekteja.

Toteutus abstraktia luokkaa käyttämällä tapahtuu, koska toteutuksilla pitää olla yhteinen nimi, jonka avulla pystytään viittaamaan toteutuksen toimintoihin. Nämä abstraktit luokat peritään ohjelmoitavassa objektiluokassa, mikä mahdollistaa proseduraalisen generoinnin toteutuksen pienissä osissa, kuten suunnitelmassa tavoiteltiin.

Luokkien menetelmät ovat generointi, tarkistus, tallenna ja palauta. Generointimenetelmä ottaa itselleen listan ja palauttaa uuden listan prosessin suorittamisen jälkeen. Sääntöluokan tarkista-menetelmä ottaa myös listan, mutta palauttaa yksinkertaisemman bool-muuttujan, joka sisältää vastauksen ”totta” tai ”väärin”. Tallenna-menetelmä nimensä mukaisesti tallentaa nykyisen tilanteen mahdollista palautusta varten. Palauta-menetelmä ottaa myös listan ja palauttaa aiemman vaiheen, jos on tarvetta.

Tällä tavalla listaa pyörittämällä voidaan helposti hoitaa kaikki generointivaiheet ohjelmoitavien objektien sisällä yleishyödyllisellä tavalla. Koska vaiheet

on toteutettu ohjelmoitavien objektien sisällä, on myös mahdollista, että vaiheet voidaan suorittaa ilman listan käyttöä. Tämä vaikuttaa pieneltä edulta, mutta koska on mahdotonta ennustaa mihin tarkoitukseen generointia tehdään, voi olla, ettei lista sovellu generointityöhön. Näissä tapauksissa luokkien välillä pyöritetään tyhjää listaa, jota ei käytetä ja generoinnin tallennusmenetelmä tehdään ohjelmoitavien objektien sisällä.

Prosessin vaiheiden toteutus ja tarkistus tapahtuu yksinkertaisen silmukan sisällä. Ennen jokaista silmukkaa tallennetaan nykyinen generointiprosessi, jotta on mahdollista palata siihen, mikäli sääntöä on rikottu. Silmukan sisäinen prosessi näyttää seuraavalta: tallenna prosessi, generoi, tarkista sääntö, tarkista tulos sääntöä vastaan ja riko silmukka, jos sääntöä ei rikota. Mikäli sääntö rikkoutuu silmukassa, palataan takaisin alkuun, jossa ladataan alkuperäinen tilanne käyttämällä väliaikaista listaa tai muuta menetelmää käyttäen. Tämä palautuspiste luodaan jokaisen vaiheen alussa ja siihen tallennetaan onnistunut tilanne. Mikäli prosessi epäonnistuu liian monta kertaa, generointi lopetetaan ja käyttäjälle annetaan virheviesti, jossa kerrotaan missä generointivaiheessa epäonnistuttiin. Tällä tavalla on helppoa löytää ongelmavaiheita, joissa on joko liian tiukat säännöt tai huono generointiprosessi. Vaiheen onnistuessa annetaan samalla tavalla käyttäjälle viesti konsoli-ikkunan kautta, jossa mainitaan generointivaihe ja kuinka monta kertaa se epäonnistui ennen onnistumista. Tämä mahdollistaa vaikeiden generointivaiheiden helpon tunnistamisen ja testauksen.

Tällä tavalla koko prosessi rakentuu pienistä osista ja toteutus myös mahdollistaa loputtaman laajennuksen ja muokkaamisen eri tilanteita varten. Kentän toteutuksen kannalta opinnäytetyön toteutus tulee kuitenkin keskittymään pienempään mittakaavan.

7 KEHITTÄMISTYÖ

Laadullinen tutkimus ja työkalujen analyysi edesauttoivat selkeän suunnitelman tekemisessä. Vaikka suunnitelma on selkeä, on kuitenkin tärkeää analysoida suunnitelman aiheuttamat haasteet ja ongelmatilanteet, joista suurin osa

johtuu proseduraalisen generoinnin luonteesta. Kun proseduraalisen generoijan osia rakennetaan käyttäen GenerationStep-luokkaa, on tärkeää muodostaa selkeät tarkistussäännöt, jotka kattavat halutun tuloksen. Myös generointivaiheen muodostaminen rajallisella tavalla on tärkeä vaihe toteutusta. Tällä pystytään välttämään ongelmatilanteita, joista esimerkiksi No Man’s Sky -peli kärsi.

Koska prosessia voidaan laajentaa loputtomasti, on tärkeää rajata kehittämistyö järkevästi, ettei skaala ylitä opinnäytetyön rajoja. Tämän takia kehittämistyön tavoitteena tulee olemaan kaksi yksinkertaista kenttägenerointiprosessia. Ensimmäisessä generoidaan Binding Of Isaac -pelin tyylinen labyrintti 3D-pelimaailmassa ja toisessa rakennetaan Dwarf Fortressista inspiroitu pelimaailma, mutta hieman yksinkertaistettuna 2D-pelimaailmassa.

7.1 Toteutus

Kehittämistyötoteutus tapahtui pienin muutoksin. Ensiksi luotiin abstraktiluokat Sääntö-, Palautus- ja Generointi-luokat. Näiden luokkien perijät yhdistettiin GenerationStep-luokassa yhtenäiseksi prosessiksi, jonka avulla eri sääntöjä, palautuksia ja generointeja voidaan helposti yhdistää.

Tämän jälkeen alkoi itse generointiprosessin tekeminen Generaattori-luokan sisällä, joka perii Monobehavior-luokan. Luokka toimii suhteellisen yksinkertaisella tavalla. Luokka ottaa generointilistan, joka rakentuu GenerationStep-luokan jäsenistä, ja prosessin alettua kaikki listan jäsenet mennään läpi yksi kerrallaan. Prosessin alettua luodaan lista, jota muokataan prosessin aikana. Listan tekemisen jälkeen kutsutaan GenerationStep-luokan Generointi-menetelmää, joka on ohjelmoitavassa objektissa. Menetelmä säilyttää kaiken tiedon, jota generointivaiheessa tehdään.

Generoinnin viimeistelyn jälkeen tarkistetaan sääntö ja jos sääntöä on rikottu, aloitetaan aiemman tuloksen palautus tai alkutilanteen palautus. Muussa tapauksessa jatketaan eteenpäin ja annetaan Console-työkalun kautta viesti, jossa kerrotaan vaiheen onnistumisen suorittaminen sekä se, kuinka monta

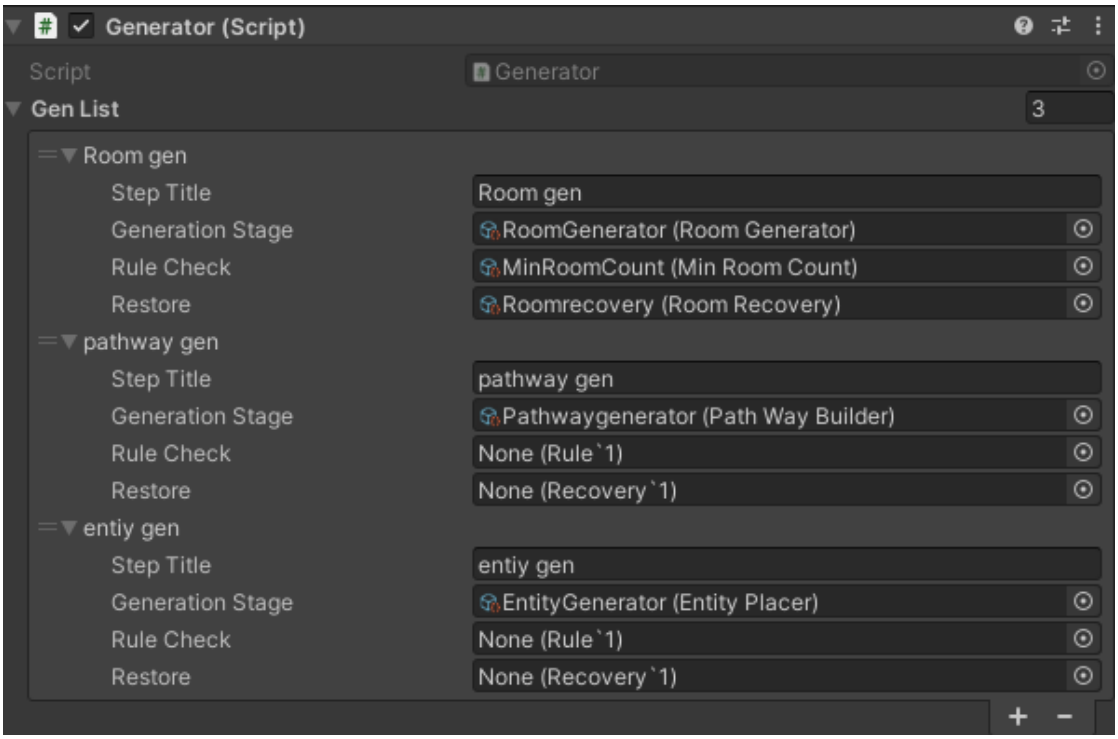
kertaa vaihetta yritettiin. Näiden tietojen tarkoituksena on antaa käyttäjille parempi ymmärrys generointivaiheiden toimivuudesta ja tällä tavalla tehdään mahdolliseksi helppo ongelmankorjaus. Generoinnissa käytettävät grafiikat ovat kaikki Unity-pelimoottorin sisällä tuotettuja grafiikkoja ja materioita. Vaiheen suorittamisen jälkeen tallennetaan nykyinen vaihe, jotta palautusluokka voi hyödyntää sitä, jos sääntöä rikotaan seuraavassa vaiheessa. Vaiheessa voidaan myös käyttää vaihtoehtoista lähestymistapaa, mikäli lista ei sovellu generointiin.

```
public void Generate()
{
    if (genList.Count == 0)
    {
        Debug.Log("genlist is empty in " + gameObject.name);
        return;
    }
    List<GameObject> currentLists = new List<GameObject>();
    bool _rulePassed;
    for (int i = 0; i < genList.Count; i++)
    {
        int _attemptCount = 0;
        _rulePassed = false;

        do
        {
            List<GameObject> currentListsCopy = new List<GameObject>();
            _attemptCount++;
            if (genList[i].restore != null)
            {
                // Store the current state of the currentLists before running the generation step to avoid posible problems...
                currentListsCopy = genList[i].restore.Record(currentLists);
            }

            //generate according to current step
            if (genList[i].generationStage != null)
            {
                Debug.Log(currentLists.Count + " " + genList[i].stepTitle);

                currentLists = genList[i].generationStage.Generate(currentLists);
            }
            //check if generation breaks rule
            if (genList[i].ruleCheck != null)
            {
                _rulePassed = genList[i].ruleCheck.Check(currentLists);
            }
            else
            {
                _rulePassed = true;
            }
            //restore earlier state if rule has been broken and repeat loop
            if (!_rulePassed && genList[i].restore!=null)
            {
                currentLists=genList[i].restore.Recover(currentLists,currentListsCopy);
            }

        } while (!_rulePassed && _attemptCount < 10);

        if (_rulePassed == false)
        {
            Debug.Log("failed at " + genList[i].stepTitle);
            return;
        }

        Debug.Log(genList[i].stepTitle + " completed. Current Stage: " + i + " passed attempt count: " + _attemptCount);
    }
}
```

Kuva 17. Generator-luokka kokonaisuudessaan. Luokka pyörittää listaa eri generaatiovaiheiden läpi, kunnes ne kaikki ovat suoritettu.

Eri generointiprojektit tarvitsevat luonnollisesti omat apuluokkansa generoinnin hyödyntämistä varten. 3D-generoinnissä tavoitteena oli tehdä eri kokoisia huoneita, yhdistää ne ja täyttää huoneet tavaroilla sekä vihollisilla. Tämän takia

prosessissa käytettiin apuluokkaa ”Room” sekä ”Teleport” ja ohjelmoitavat objektit, joista prosessit rakennettiin, olivat ”EntityPlacer”, ”RoomCount”, ”PathWayGenerator” ja ”RoomGenerator”.



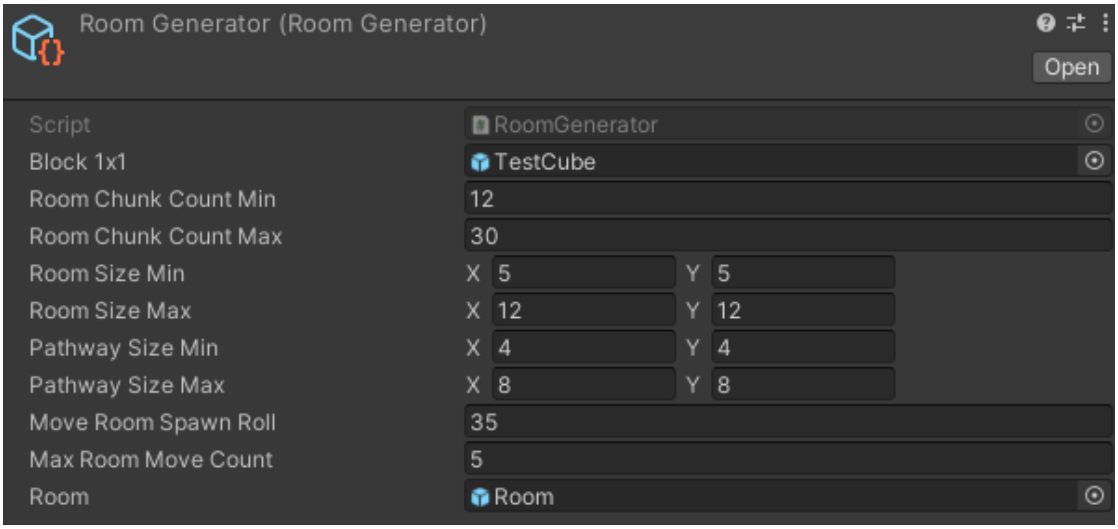
Kuva 18. Generaattorityökalu, jonka generointivaiheet on täytetty tarvittavilla ohjelmoitavilla objekteilla

Room-luokka on esine, joka asetetaan jokaiselle generoitavalle huoneelle. Room-luokan tarkoituksena on säilyttää kaiken asiaankuuluvan tiedon, jota hyödynnetään generointiprosessissa. Luokan seurauksena Generator-luokassa pyöritetään huoneista listaa, joka mahdollistaa nopeamman muokkaamisen ja palauttamisen verrattuna isompaan listaan, jossa on kaikki tieto käytettävistä objekteista.

```
public class Room : MonoBehaviour
{
    public Vector2Int min = new Vector2Int(int.MaxValue,int.MaxValue);
    public Vector2Int max = new Vector2Int(int.MinValue,int.MinValue);
    public List<GameObject> roomWalls = new List<GameObject>();
    public List<GameObject> roomCorners = new List<GameObject>();
    public List<GameObject> entities = new List<GameObject>();
    public Teleport wayPoints;

}
```

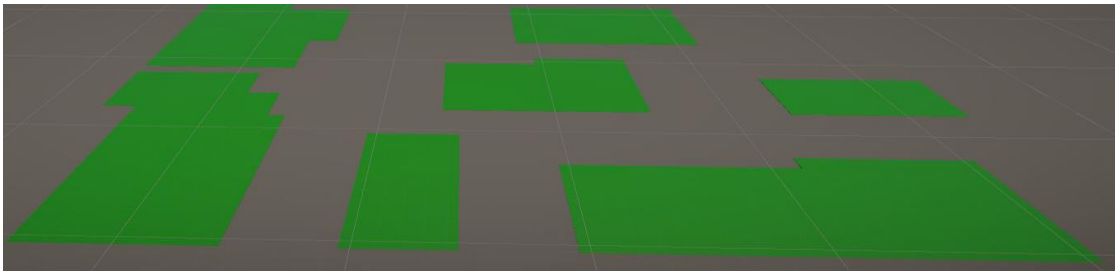
Kuva 19. Room-luokka, joka toimii apuluokkana generointiprosessille. Luokassa on kaikki tarvittavat muuttujat eri vaiheiden suorittamista varten.

Ensimmäisessä generointivaiheessa generoidaan huoneet. Generointivaiheen muokkaus tapahtuu täysin Inspector-työkalun sisällä, mikä mahdollistaa erilaisen lähestymistavan käyttäjiä varten. Luokkaa varten on tehty kahdeksan eri asetusta, joilla prosessia voi muokata (kuva 20).



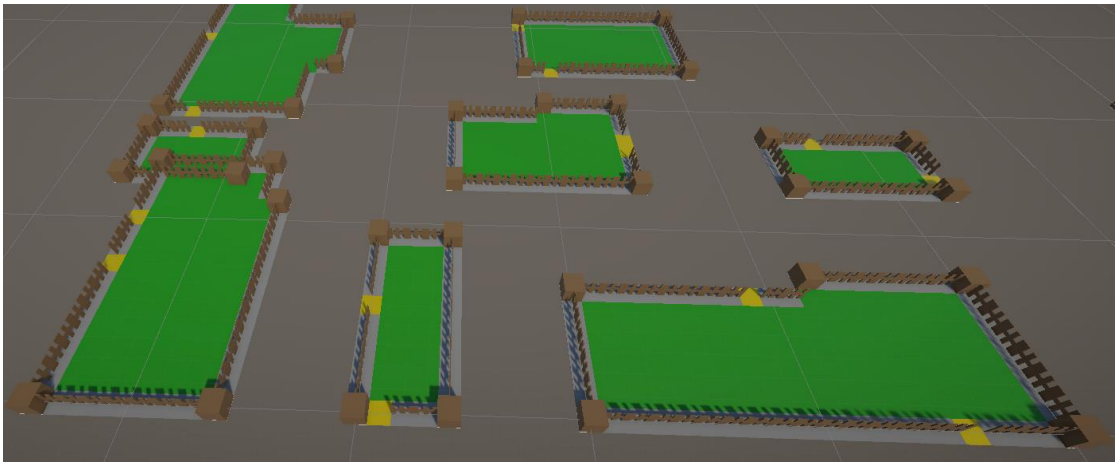
Kuva 20. RoomGenerator on ohjelmoitava objekti, jonka sisällä on generointivaiheen eri asetukset ja viittaukset.

Huoneiden generoinnissa generoidaan ensin satunnaisen kokonaisia neliskulmaisia huoneita, jotka rakennetaan 1 x 1 x 1 -kokoisista palikoista kuten ”Testi-neliö”, jota käytettiin testauksessa. Tämän jälkeen huoneet, jotka koskettavat tai ovat päällekkäin, yhdistetään yhtenäisiksi huoneisiksi. Huonepalikat voidaan tarkoituksella jättää generointisääntöjen kautta pidemmälle etäisyydelle toisistaan, minkä avulla saataisiin pelkästään pieniä neliskulmaisia huoneita. Asetuksia muokkaamalla prosessissa saadaan tuotettua variaatiota lyhyestä generointiprosessista huolimatta. Tämä on ainoa generointivaihe huonegeneroinnin kannalta, jossa hyödynnetään sääntöä tarkistaakseen minimihuonemäärä. Muissa vaiheessa säännön lisääminen ei ollut tarpeellista, koska generointivaiheissa ei ole haitallisia tuloksia.



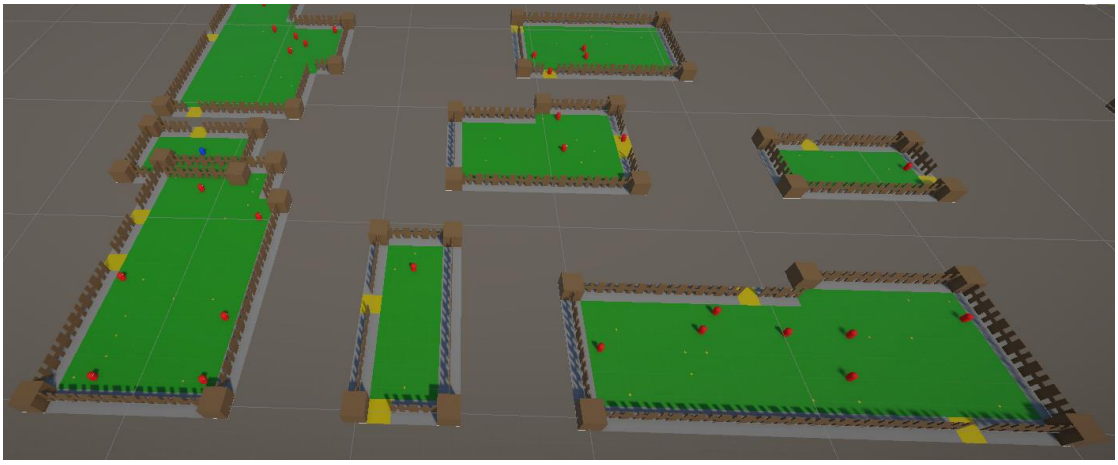
Kuva 21. Generoinnin ensimmäinen vaihe. Huoneet generoidaan ensiksi neliskulmaisiksi palikoiksi ja toisiaan lähellä olevat palikat yhdistetään yksittäisiksi huoneiksi.

Toisessa vaiheessa huoneille merkitään selvät rajat ja huoneet yhdistetään toisiin huoneisiin. Jokaisen huoneen rajoista valitaan kaksi kohtaa, joista tehdään kulkuväyliä huoneiden välillä ja nämä on merkattu keltaisella värillä (kuva 22). Kulkuväylät yhdistetään toisiinsa huoneluokissa Teleport-luokkaa käyttäen tarkistamalla samalla, että kaikkiin huoneisiin pääsee.



Kuva 22. Generointivaiheessa 2 huoneille laitetaan rajat ja tapa kulkea eri huoneisiin. Jokaisessa huoneessa keltainen palikka on yhdistetty toisen huoneen keltaiseen palikkaan. Tällä tavalla huoneet yhdistetään yhtenäiseksi kokonaisuudeksi, jossa pelaaja voi liikkua.

Viimeisessä vaiheessa huoneet täytetään vihollisilla ja tavaroilla, tässä tapauksessa kolikoilla ja vihollishahmolla. Prosessissa lasketaan pelattavan huoneen koko ja asetetaan vihollisia suhteessa 1:10 jokaista 1 x 1 olevaa palikkaa kohden ja kolikoita asetetaan samalla tavalla 1:5. Viimeiseksi selvitetään pienin huone. Tästä huoneesta otetaan pois kaikki viholliset ja kolikot, minkä jälkeen pelaajahahmo laitetaan huoneeseen. Tässä vaiheessa olisi mahdollista lisätä pelimekaniikat, jolloin lopputuotteena olisi toimiva peli.

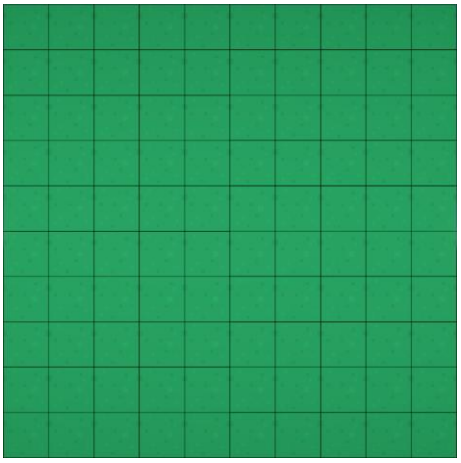


Kuva 23. Generointivaiheessa 3 huoneet täytetään keltaisilla pienillä kolikoilla ja punaisilla vihollisilla. Lopuksi pienin huone valitaan pelaajan aloitushuoneeksi. Pelaaja on sininen hahmo vasemmalla olevassa pienessä huoneessa.

Toisessa generointiprojektissa on kaksi apuluokkaa ”Grid” sekä ”GridObject” ja ohjelmoitavat objektit ”GridCreator”, ”MountainGenerator”, ”RiverGenerator”, ”VillageGenerator” sekä ”FarmGenerator”. Tavoitteena on tehdä samantapainen pelimaailma kuten Dwarf Fortress -pelissä tai muussa vastaavassa 2D-ruudukkopohjaisessa pelissä, minkä takia Grid-luokan tarkoituksena on tiedon säilyttäminen ja helppo muokkaaminen. Tämän takia luokka sisältää korkeus- ja pituustiedot, joita generointi käyttää ja näiden tietojen pohjalta rakennetaan [korkeus kertaa pituus] pohjainen ruudukko, joka täytetään GridObject-luokkaan kuuluvilla esineillä. Nämä esineet sisältävät taas ruututyypin, objektityypin ja kuvan, jota ruutu käyttää. Jokainen generointivaihe hyödyntää aiempia vaiheita, minkä takia sääntöihin perustuvaa tarkistusta ei käytetä, koska haitallisia tuloksia ei ole olemassa.

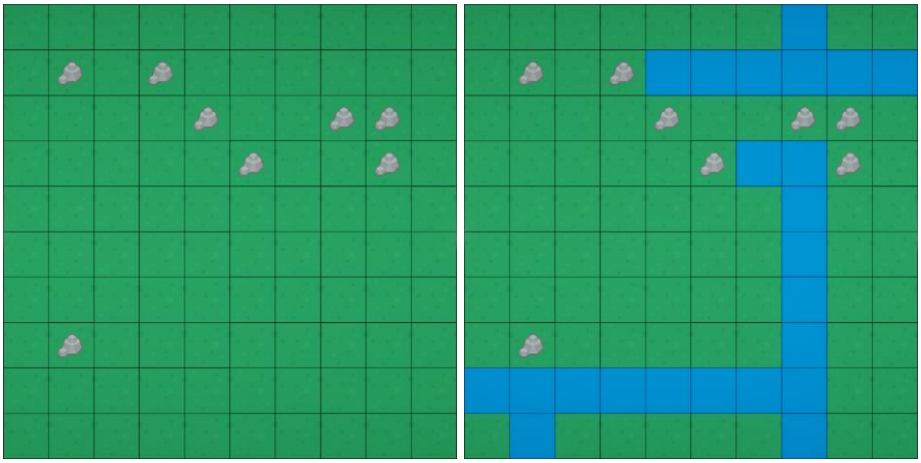
Kuva 24. Kenney-yhtiön CC0-lisensillä tuotetut grafiikat, joita hyödynnetään tässä projektissa (Kenney 2016).

Tässä generoinnissa hyödynnettiin KenneYn tuottamaan ”Mediaval RTS” -grafiikkapakettia, jota KenneY-yhtiö jakaa Creative Commons CC0 -lisenssillä, mikä tarkoittaa, että grafiikat ovat julkisessa omistuksessa eli vapaasti käytettävissä. Projektin kannalta valitut grafiikat tulevat esittämään ensimmäinen jokea, toinen ruohoa, kolmas vuorta, neljäs kylää ja viimeinen peltoa (kuva 24).



Kuva 25. Generoinnin ensimmäinen vaihe, jossa rakennetaan ruudukko ja täytetään se halutulla aloitus ruututyypillä

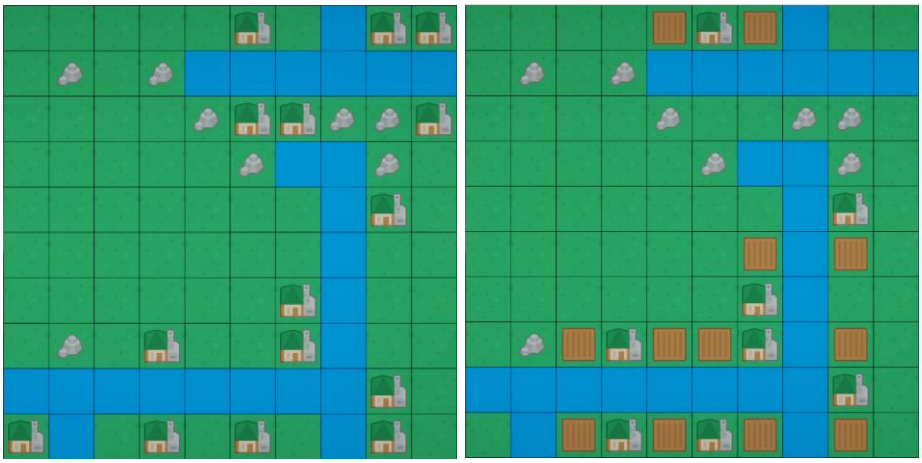
Ensimmäisessä vaiheessa ruudukko luodaan käyttäen haluttuja korkeus- ja pituusarvoja. Koko ruudukko täytetään tässä vaiheessa yhdellä annetulla kuvalla, kuten valitulla ruoho grafiikalla (kuva 24). Tämän jälkeen ruudukko käydään läpi kerran ja kaikkiin Ruudukko-objektien ruututyyppeihin merkataan, onko ruudun paikka normaali, kulma vai pääty. Näitä tietoja käytetään tulevissa generointivaiheissa.



Kuva 26. Vasemmalla puolella generoinnin toinen vaihe, jossa asetetaan vuoria pelikartalle. Oikealla puolella on kolmas vaihe, jossa generoidaan joet. Joet jakaantuvat ja kääntyvät törmätessä vuoriin.

Toisessa vaiheessa lisätään vuoria. Vuoria lisätään valitsemalla haluttu määrä satunnaisia ruutuja, joista vuoret laajenevat vieressä oleviin ruutuihin annettujen rajoitusten perusteella. Tähän käytetään Mountaingenerator-ohjelmoitavan objektin vuorilaajennus-asetuksen minimi- ja maksimiarvoja. Vuorien asetusten jälkeen tarkistetaan, muutetaanko joku vuorista vuoriryhmäksi. Vuoriryhmissä ympäröiviä ruutuja muutetaan satunnaisesti vuoriksi.

Kolmannessa vaiheessa generoidaan jokia. Joet generoidaan valitsemalla satunnainen päädyssä oleva ruutu, josta joki aloittaa generoinnin vastakkaiseen päätyyn asti. Tämän jälkeen päätetään maksimiruutumäärä jokea kohti. Jos joen seuraavassa ruudussa on vuori, niin joki jakaantuu kahteen ja jakaantuneet joet aloittavat matkan vastakkaisiin suuntiin. Joen generointi jatkuu, kunnes joki saavuttaa päädyn tai ruutumäärä jokea kohden loppuu. Vaihe voidaan toistaa useampaan kertaan, mutta joet yhdistyvät, mikäli ne törmäävät toisiinsa ja seuraavan joen generointi alkaa.



Kuva 27. Vasemmalla puolella on generoinnin neljäs vaihe, jossa asetetaan kyliä jokien viereisiin ruutuihin. Oikealla puolella on viides vaihe, jossa generoidaan pellot ja poistetaan kylät, joiden viereisissä ruuduissa ei ole tarpeeksi peltoja.

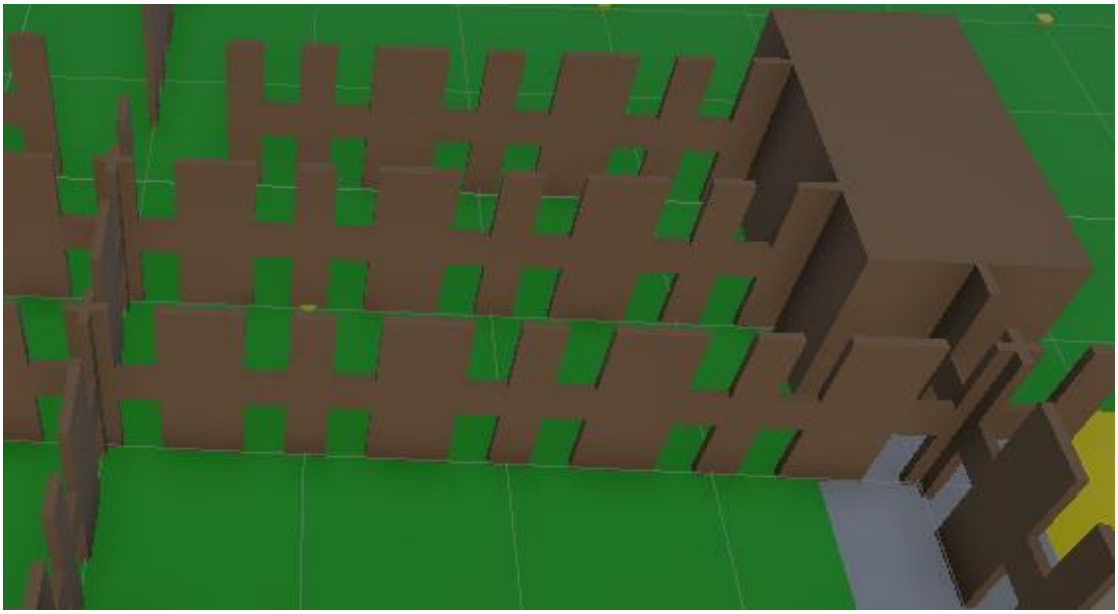
Kylägeneroinnissa valitaan satunnaisia ruutuja, jotka ovat ruohikkoa ja sijaitsevat jokiruutujen vieressä. Kylägenerointi lopetetaan, kun haluttu määrä kyliä on tehty tai, kun ruudukossa ei enää ole vaatimukset täyttäviä ruutuja. Tämän jälkeen jatketaan peltogenerointiin, jossa ensiksi arvotaan tarvittava peltomäärä jokaista kylää kohti, minkä jälkeen tarkistetaan mahdolliset peltopaikat jokaiselle kylälle. Jos talolla ei ole tarpeeksi mahdollisia peltopaikkoja, se poistetaan. Tämän jälkeen pellot asetetaan jokiruutujen ja kylien viereen. Jokaisella kylällä on omat peltonsa, joten jos jonkin kylän viereiset vapaat peltoruudut tippuvat alle tarvittavan peltoruutumäärän, kylä poistetaan. Poistetut kylät korvataan normaalilla ruohoruudulla ja niitä voidaan käyttää muiden kylien peltogeneroinnissa, jos viereisessä ruudussa on kylä.

Tässä vaiheessa generointi on tuottanut ruudukon halutuilla säännöillä ja siihen voidaan helposti yhdistää haluttavat pelimekaniikat. Mahdollinen vaihtelu

lopputulosten kanssa näkyy helposti muokkaamalla vähänkin generointiasetuksia ja ruudukkoa voidaan helposti saada haluamakseen pienellä työpanoksella.

7.2 Toteutuksen analyysi

Toteutuksen suorittamisen jälkeen on tärkeää analysoida saatuja projekteja kriittisillä silmillä ja harkita parannuskeinoja. Tässä tapauksessa molemmat generointiprosessit antoivat oman opetuksensa. Huonegenerointi oli ensimmäinen projekti, joka toteutettiin prosessia käyttäen ja näytti nopeasti apuluokkien tarpeen prosessin sujuvassa suorituksessa. Tämän lisäksi se osoitti proseduraalisen generoinnin haasteet, koska kaikki generoinnit eivät sujuneet aivan oletetulla tavalla, kuten huoneiden yhdistymisvaiheessa (kuva 28).



Kuva 28. Jokaisella huoneella pitäisi olla aitoja vain ruudun reunoissa, mutta huoneen määrittelyvirhe aiheuttaa useita aitakerroksia huoneen sisällä.

Ongelma tässä tilanteessa johtuu huoneiden yhdistämistavasta. Huoneet ovat neliskanttisia, minkä takia huoneiden kokoa on helppo verrata keskenään, mutta huoneiden yhdistymisen jälkeen huoneet muuttuvat suorakulmaisiksi monikulmioksi, jota ei ole otettu huomioon yhdistymisprosessissa. Jos sama huone yhdistyy kaksi tai useampaa kertaa on mahdollista, että toinen huone yhdistys epäonnistuu, minkä seurauksena on huoneita huoneiden sisällä, mikä ei ole haluttu lopputulos.

Ruudukkogeneroinnissa näkyy listan käytön rajoitukset. Koska generointi tapahtui lähes täysin ruudukon sisällä, listan pyörittäminen oli hyödytöntä toteutuksen kannalta. Sen sijaan jokaisessa vaiheessa piti käyttää erillistä palautusmenetelmää. Tämä hidastaa generointiprosessin tekemistä ja pakottaa suunnittelemaan erillisiä prosesseja palautuksen tekemiseen, mikä ei ole ideaalista. Projektin tyylin takia ruudukosta piti tehdä kopio vaiheiden lopussa, mikä käytännössä tarkoittaa muutamia satoja ylimääräisiä peliobjekteja, joita pitää käsitellä ja poistaa prosessin aikana. Parempi tapa olisi luultavasti muokata Generator-luokkaa siten, että se käyttäisi ruudukkoa listan sijaan. Tämä näyttää sen, että proseduraalinen generointi on helpompi toteuttaa, kun se suunnitellaan projektikohtaiseksi. Tämä tarkoittaa sitä, että Generator-luokkaa on parasta käyttää pohjana ja tehdä pienempiä muokkauksia saadakseen tehokkaimman mahdollisen tuloksen.

8 JOHTOPÄÄTÖKSET

Opinnäytetyön lopputuloksena oli yleishyödyllinen työkalu pelikenttien generoimiseen Unity-pelimoottorissa. Työkalu muodostui hyödyntäen laadullisessa tutkimuksessa saatuja standardeja, joita käytettiin toimintasuunnitelman tekemisessä.

Laadullisen tutkimuksen aikana saatiin selville, että proseduraalinen generointi toimii parhaiten hyödyntämällä modulaarista kehittämistä sekä pienempien vaiheiden lopputulosten tarkastelemista tarkasti määritettyjen sääntöjen perusteella, joiden tulosten myötä jatketaan seuraavaan generointivaiheeseen tai palautetaan aiempi generointitulos.

Tämän perusteella luotiin suunnitelma, jossa työkalun osat hajautettiin selviin osiin, jotka sitten toteutettiin Unity-pelimoottorissa. Lopussa prosessia käytettiin toteuttaakseen kaksi erilaista kenttägenerointiprojektia, jotka todistivat, että työkalu toimii proseduraalisien pelikenttien generoimisiin. Samalla todistettiin laadullisessa tutkimuksessa tehdyt havainnot niitä hyödyntämällä.

8.1 Tutkimuskysymysten pohdinnat

Vaikka työkalu toimisikin odotetulla tavalla, tärkeä osa opinnäytetyön onnistumisen mittaamista on tutkimuskysymyksiin vastaaminen. Vastausten muodostuksessa on tärkeää muistaa, että vastaukset eivät ole absoluuttisia totuuksia, vaan pohdintoja asetettuihin kysymyksiin. Tämän takia on mahdollista, että kysymysten pohdinta johtaa erilaisiin tuloksiin, jos kysymykset kysytään uudelleen toisenlaisessa tutkimuksessa. Opinnäytetyön tutkimuskysymykset olivat seuraavanlaiset:

- Mitä proseduraalinen generointi on?
- Miten proseduraalinen generointi toimii?
- Mitkä ovat yleiset menetelmät ja standardit proseduraalisessa generoinnissa?
- Mitä mahdollisia ongelmia proseduraalisesti generoidussa sisällössä on ja miten näitä ongelmia voi välttää?
- Miten proseduraalista generointisysteemiä voi käyttää eri tarkoituksiin ja tilanteisiin?

Ensimmäiseen kysymykseen saatiin tarvittavat tiedot luvussa 3 Proseduraalinen generointi, jossa todettiin, että proseduraalinen generointi on säännönmukaista tiedon generoimista, jota hyödynnettiin laadullisen tutkimuksen tekemisessä.

Toinen kysymys on hieman monimutkaisempi, mutta kysymykseen saadaan vastaus analysoimalla ensimmäisen kysymyksen vastausta ja ottamalla huomioon laadullisen tutkimuksen tulokset luvussa 4.4 Yhtenäisyydet. Proseduraalinen generointi toimii generoimalla uutta tietoa tai sisältöä seuraamalla ohjelmoijan tai käyttäjän määrittelemiä sääntöjä.

Kolmannen kysymyksen tulos saadaan analysoimalla laadullista tutkimusta. Proseduraalinen generointi jaetaan pienempiin osiin, mitä kutsutaan modulaariseksi suunnitteluksi. Näiden pienempien generointivaiheiden lopputuloksia verrataan määritettyihin sääntöihin, minkä avulla vaihe joko aloitetaan uudelleen tai se vaikuttaa seuraavaan generointivaiheeseen. Säännöt taas toimivat parhaiten, jos ne seuraavat selkeitä matemaattisia tai loogisia määritelmiä, joka ei anna tietokoneella mahdollisuutta tehdä haitallista lopputulosta. Tämä sääntö vahvistui myös työkalun toteutuksessa, kun huoneiden yhdistämisessä tulos ei ollut haluttu, koska sääntöä ei ollut määritelty oikein.

Toiseksi viimeistä kysymystä lähestyttiin jo hieman kolmannen kysymyksen pohdinnassa. Ongelmat muodostuvat, kun generoitavalle asialle ei ole tehty selkeitä loogisia tai matemaattisia sääntöjä tai jos nämä säännöt on jollakin tavalla muodostettu väärin. Ongelmilta vältytään parhaiten generointivaiheiden ja sääntöjen tarkalla suunnitelulla, minkä avulla virheitä voidaan välttää.

Viimeistä kysymystä käsiteltiin kehittämistyössä, kun työkalu toteutettiin. Koska proseduraalinen generointi on tiedon tai sisällön generoimista, sen käytöllä ei ole todellisia rajoja. Tämän takia työkalu näytti mahdollisuuksien monipuolisuutta generoimalla ensiksi pelikentän huoneista 3D-peliympäristöstä, minkä jälkeen sitä käytettiin koordinaatistoon perustuvan 2D-pelikartan generoimiseen. Tässäkin on tärkeää huomioida, että kehitysten lopputulos riippuu täysin generointivaiheiden ja sääntöjen laadusta, minkä takia tarkka suunnittelu on tärkein vaihe proseduraalisen generoinnin käyttöönottoa varten.

Kaikkiin tutkimuskysymyksiin on saatu tarvittava pohdinta. Tämän perusteella on mahdollista rakentaa pätevät vastaukset. Nämä pohdinnat antavat selvän kuvan aiheesta ja sen tarjoamista mahdollisuuksista sekä ratkaisuja ongelmatilanteisiin, joita uudet käyttäjät voivat kokea ottaessaan proseduraalista generointia käyttöön ensimmäistä kertaa. Tämän takia on helppoa todeta, että opinnäytetyö on onnistunut tavoitteissaan.

8.2 Jatkokehitysehdotukset

Opinnäytetyön, projektien ja tutkimusten suorittamisen jälkeen on tärkeää tarkastella mahdollisia jatkotutkimuksia, joita voidaan johtaa nykyisestä tutkimuksesta, aiheesta tai tuloksista. Tällä tavalla edistetään tulevia tutkimuksia ja voidaan harkita valittua tutkimusaihetta toiselta näkökulmalta. Tämän opinnäytetyön aihe, pelikenttien proseduraalinen generointi, on alusta alkaen ollut erittäin laaja tutkimusaihe, vaikka tässä opinnäytetyössä se onkin rajattu Unity-pelimoottoriin. Tästä aiheesta voi tehdä lukuisia tutkimuksia, minkä takia aiheen rajaaminen on tärkeä osa tutkimuksen suunnittelua. Jatkokehityksen suunnittelun kannalta on tärkeää huomioida se, mitä tutkimuksen aikana on saavutettu ja voiko näistä saavutuksista johtaa uusia tutkimusaiheita.

Koska opinnäytetyönaikana saatiin tuotettua systeemi, jota voidaan käyttää pelikenttien proseduraaliseen generointiin, yksi luonnollisista tutkimusaiheista tai projekteista olisi pelin tuottaminen luotua systeemiä käyttäen. Toinen vaihtoehto olisi systeemin laajempi toteutus. Vaikka systeemi tehtiin pelikenttiä varten, sen prosessi voidaan helposti laajentaa muihin proseduraalista generointia hyödyntäviin tarkoituksiin. Viimeinen jatkokehitysehdotus olisi tekoälyn käyttäminen proseduraalisessa generointiprosessissa. Tekoäly on ottanut suuraskelia viime vuosien aikana koneoppimisen kehityksen myötä, minkä vuoksi aiheiden yhdistäminen olisikin ajankohtaista ja voisi johtaa uusien tekniikoiden löytämiseen proseduraalisen generoinnin kannalta

9 POHDINTA

Opinnäytetyön lopputuloksena syntyi proseduraalinen generointityökalu, joka hyödynsi opinnäytetyössä tehtyä laadullista tutkimusta ja tämä paljasti proseduraalisessa generoinnissa käytetyt standardit. Osana analyysia keskityttiin myös ymmärtämään sitä, miten voidaan välttää proseduraalisen generoinnin käyttämisestä muodostuvia ongelmatilanteita. Näitä löydettyjä tietoja hyödynnetiin työkalun suunnitelman tekemisessä, jonka takia Unity-pelimoottorissa työskentely oli sujuva prosessi ja työkalu tuotti haluttuja proseduraalisesti generoituja pelikenttiä. Tämä oli opinnäytetyön tavoitteena, joka on selvästi onnistunut ja sen yhteydessä on muodostettu selkeä toimintamalli proseduraalisen generoinnin suunnittelemista ja toteuttamista varten. Tästä huolimatta johtopäätösten tekeminen ei ole vain onnistumisen tarkastamista vaan tulosten opetusten sisältämistä, joten on tärkeää, että opinnäytetyötä tarkastellaan kriittisellä silmällä.

Opinnäytetyön aiheen valinnassa huomioitiin eettisyys ja aiheen merkityksellisyys alan kannalta. Tämän lisäksi analyysi ja johtopäätökset ovat luonnollisesti muodostuneet opinnäytetyön aikana analysoitavasta materiaalista ja tehdystä työstä ilman, että ennakkoluulot tai odotukset ovat vaikuttaneet tuloksiin. Tämän takia on vaivatonta todeta, että työn akateeminen luotettavuus ja eettiset olettamat ovat säilyneet koko opinnäytetyön ajan.

Proseduraaliseen generointiin liittyvä teoria, joka käytiin pääasiassa läpi luvussa nimeltään 3 Proseduraalinen generointi, oli oleellinen osa opinnäytetyötä. Tärkein asia oli proseduraalinen generointi -termin yhdistäminen selkeästi ymmärrettävään konseptiin, jonka ympärille opinnäytetyö voitiin rakentaa. Tämän lisäksi tutkittavien pelien materiaalin analysoinnin tärkeys tuli selkeästi esille tutkimuksen aikana, mikä mahdollisti tarkan suunnitelman tekemisen, jota pystyttiin hyödyntämään lopputulosten saamiseksi.

Teoria kuitenkin teki vain pintakosketuksen proseduraaliseen generointiin, joten teoria olisi voinut olla mahdollisesti laajempi. Ongelmana kuitenkin on aiheen laajuus, jonka takia aihe pitää pystyä rajaamaan järkevään mittakaavaan. Tämän takia useita generoitiin liittyviä aiheita ei ole mahdollista käsitellä yhdessä opinnäytetyössä. Tuotettua lopputulosta pystytään kuitenkin helposti laajentamaan sisältämällä siihen muita proseduraalisen generoinnin aihepiiriin kuuluvia asioita, kuten jatkokehitysehdotuksissa käsiteltiin. Tuotettu suunnitelma ja työkalu eivät kuitenkaan ole rajoitettu pelkästään pelikenttien generointiin, vaan ne voidaan todennäköisesti erittäin pienellä vaivalla ottaa käyttöön myös muissa proseduraalisissa generointitöissä. Tämän takia on parasta ajatella tätä opinnäytetyötä alustavana oppaana proseduraaliseen generointiin ja sen ymmärtämiseen.

LÄHTEET

Axon, S. 2016. Unity 10 vuotta: Parempaa tai huonompaa - pelien kehittäminen ei ole koskaan ollut helpompaa. Ars Technica. Verkkodokumentti. Päivitetty 27.9.2016. Saatavissa: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/ [viitattu 26.2.2023].

Bay12Games. 2006. Dwarf Fortress. Videopeli. Silverdale, WA: Bay12Games. Saatavilla: https://www.bay12games.com/dwarves/ [viitattu 3.2.2023].

Binding of Isaac: Rebirth Wiki. 2018. Level Generation. WWW-dokumentti. Päivitetty 1.4.2023. Saatavissa: https://bindingofisaacrebirth.fandom.com/wiki/Level_Generation [viitattu 3.1.2023].

Carpenter, E. 2011. Procedural Generation of Large Scale Gameworlds. Dublinin yliopisto, Trinity College. Tietojenkäsittelytieteen maisterin tutkielma. PDF-dokumentti. Saatavissa: https://www.scss.tcd.ie/publications/theses/diss/2011/TCD-SCSS-DISSERTATION-2011-056.pdf [viitattu 1.1.2023].

New disappointment discovered : No Man's Sky (2016). 2016. Crowbcat. Videoleike. Päivitetty 16.8.2016. Saatavilla: https://www.youtube.com/watch?v=A8P2CZg3sJQ [viitattu 21.1.2023].

No Man's Sky: How I Learned to Love Procedural Art. 2015. GDC. Videoleike. päivitetty 21.7.2015. Saatavilla: https://www.youtube.com/watch?v=vcEA41eBOGs [viitattu 16.1.2023].

Günther, K. & Hasanen, K. 2021. Laadullisen tutkimuksen verkkokäsikirja: Tutkimuksen suunnittelu. Tietoarkisto. WWW-dokumentti. Päivitetty 2021. Saatavissa: https://www.fsd.tuni.fi/fi/palvelut/menetelmaopetus/kvali/laadullisen-tutkimuksen-prosessi/tutkimuksen-suunnittelu/ [viitattu 30.1.2023].

Haas, J.K. 2014. A History of the Unity Game Engine. Worcester Polytechnic Institute. Insinööritieteiden tiedekunta. Tutkimus. PDF-dokumentti. Saatavissa: https://digital.wpi.edu/show/tx31qh96p [viitattu 18.3.2023].

Hall, C. 2014. Dwarf Fortress will crush your CPU because creating history is hard. Polygon. WWW-dokumentti. Päivitetty 23.7.2014. Saatavissa: https://www.polygon.com/2014/7/23/5926447/dwarf-fortress-will-crush-your-cpu-because-creating-history-is-hard [viitattu 9.1.2023].

Hello Games. 2016. No Man's Sky. Videopeli. Guildford, Yhdistynyt Kuningaskunta: Hello Games.

Jyväskylän yliopiston Koppa. 2021. Laadullinen tutkimus. WWW-dokumentti. Päivitetty 28.10.2021. Saatavissa: https://koppa.jyu.fi/avoimet/hum/menetelmapolkuja/menetelmapolku/tutkimusstrategiat/laadullinen-tutkimus [viitattu 1.1.2023].

Mediaval-rts. 2016. Kenney. CC-0. PNG-tiedosto. Päivitetty 10.8.2016. Saatavissa: https://www.kenney.nl/assets/medieval-rts [viitattu 12.4.2023].

Kharpal, A. 2016. 'No Man's Sky': Would you play a game that takes 584 billion years to explore? CNBC. WWW-dokumentti. Päivitetty 11.8.2016. Saatavissa: https://www.cnbc.com/2016/08/10/no-mans-sky-release-would-you-play-a-game-that-takes-584-billion-years-to-explore.html [viitattu 10.1.2023].

Kuchera, B. 2016. No Man's Sky was a PR disaster wrapped in huge sales. Polygon. WWW-dokumentti. Päivitetty 16.9.2016. Saatavissa: https://www.polygon.com/2016/9/16/12929618/no-mans-sky-disaster-lies-lessons-learned [viitattu 11.1.2023].

McWhertor, M. 2014. Entinen EA:n toimitusjohtaja John Riccitiello nyt Unityn johdossa. Polygon. Verkkodokumentti. Päivitetty 22.10.2014. Saatavissa: https://www.polygon.com/2014/10/22/7039683/electronic-arts-john-riccitiello-unity-ceo [viitattu 8.3.2023].

Microsoft. 2022. A tour of the C# language. WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ [viitattu 24.3.2023].

Microsoft. 2022. Inheritance in C# and .NET. WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ [viitattu 12.3.2023].

Nicalis, Inc. 2011. The Binding of Isaac. Videopeli. Santa Ana, California: Nicalis, Inc. [viitattu 30.1.2023].

Sarawagi, S. s.a. Procedural Generation – A Comprehensive Guide Put in Simple Words. Scaleyourapp. WWW-dokumentti. Saatavissa: https://scaleyourapp.com/procedural-generation-a-comprehensive-guide-in-simple-words/ [viitattu 13.2.2023].

Schatzeder, D. 2021. The Logic of Procedural Generation. Schatzeder. WWW-dokumentti. Päivitetty 19.3.2021. Saatavissa: https://schatzeder.medium.com/the-logic-of-procedural-generation-3043368a6a06 [viitattu 16.2.2023].

Stegner, B. 2021. What Are Roguelike and Roguelite Video Games? MUO. WWW-dokumentti. Päivitetty 24.10.2021. Saatavissa: https://www.makeuseof.com/what-are-roguelike-and-roguelite-video-games/ [viitattu 28.1.2023].

Takahashi, D. 2022. Unity-raportti: Unitylla tehtyjen pelien määrä kasvoi 93 % vuonna 2021. GamesBeat. Verkkodokumentti. Päivitetty 16.3.2022. Saatavissa: https://venturebeat.com/games/unity-report-number-of-games-made-with-unity-grew-93-in-2021/ [viitattu 6.3.2023].

DF2014:Advanced world generation. 2014. The Dwarf Fortress Wiki. WWW-dokumentti. Päivitetty 21.1.2023. Saatavissa: http://dwarffortresswiki.org/index.php/DF2014:Advanced_world_generation [viitattu 9.1.2023].

DF2014:Z-level. 2014. The Dwarf Fortress Wiki. WWW-dokumentti. Päivitetty 20.12.2022. Saatavissa: http://dwarffortresswiki.org/index.php/DF2014:Z-level [viitattu 6.1.2023].

Togelius, J., Kastbjerg, E., Schedl, D.C. & Yannakakis, G. 2011. What is Procedural Content Generation? Mario on the borderline. IT University of Copenhagen. PDF-dokumentti. Saatavissa: http://dx.doi.org/10.1145/2000919.2000922 [viitattu 2.2.2023].

Scripting API. 2021. Unity. Verkkodokumentti. Päivitetty 5.5.2023. Saatavissa: https://docs.unity3d.com/ScriptReference/index.html [viitattu 14.4.2023].

Coding in C# in Unity for beginners. s.a. Unity. Verkkodokumentti. Päivitetty 5.5.2023. Saatavissa: https://unity.com/how-to/learning-c-sharp-unity-beginners [viitattu 15.3.2023].

ECS FOR UNITY. s.a. Unity. Verkkodokumentti. Saatavissa: https://unity.com/ecs [viitattu 15.3.2023].

Unity User Manual: Important Classes. s.a. Unity. Verkkodokumentti. Päivitetty 5.5.2023. Saatavissa: https://docs.unity3d.com/Manual/ScriptingImportantClasses.html [viitattu 16.3.2023].

Unity User Manual: Scenes. s.a. Unity. Verkkodokumentti. Päivitetty 5.5.2023. Saatavissa: https://docs.unity3d.com/Manual/CreatingScenes.html [viitattu 16.3.2023].

Unity User Manual: Overview of .NET in Unity. s.a. Unity. Verkkodokumentti. Päivitetty 5.5.2023. Saatavissa: https://docs.unity3d.com/Manual/overview-of-dot-net-in-unity.html [viitattu 16.3.2023].

Unity User Manual: ScriptableObject. Unity. s.a. Unity. Verkkodokumentti. Päivitetty 5.5.2023. Saatavissa: https://docs.unity3d.com/Manual/overview-of-dot-net-in-unity.html [viitattu 17.3.2023].

Vuori, J. 2021. Laadullinen sisällönanalyysi. Laadullisen tutkimuksen verkkokäsikirja: Laadullinen sisällönanalyysi. WWW-dokumentti. Saatavissa: https://www.fsd.tuni.fi/fi/palvelut/menetelmaopetus/kvali/analyysitavan-valinta-ja-yleiset-analyysitavat/laadullinen-sisallonanalyysi/ [11.5.2023].

KUVALUETTELO