OULUN AMMATTIKORKEAKOULU

Maria Martin

PROSEDURAALISEN GENEROINNIN KÄYTTÖ PELINKEHITYKSESSÄ

PROSEDURAALISEN GENEROINNIN KÄYTTÖ PELINKEHITYKSESSÄ

Maria Martin
Opinnäytetyö
Syksy 2022
Tietojenkäsittelyn tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietojenkäsittelyn tutkinto-ohjelma

Tekijä(t): Maria Martin
Opinnäytetyön nimi: Proseduraalisen generoinnin käyttö pelinkehityksessä
Työn ohjaaja(t): Matti Viitala
Työn valmistumislukukausi ja -vuosi: Syksy 2022 Sivumäärä: 31

Opinnäytetyön tavoitteena oli tutkia proseduraalisen generoinnin käyttöä pelinkehityksessä, erityisesti kenttäsuunnittelussa, ja pohtia sen hyötyjä ja haittoja. Lisäksi sitä vertailtiin perinteiseen käsin tehtyyn kenttäsuunnitteluun. Opinnäytetyössä käytettiin esimerkkipeleinä vuonna 1980 julkaistua Rogue-peliä sekä vuonna 2011 julkaistua Jetpack Joyride -peliä, joissa molemmissa proseduraalinen generointi on keskeisessä osassa. Opinnäytetyössä havaittiin, että proseduraalinen generointi vähentää kenttäsuunnittelijan työmäärää ja antaa pelaajalle kokemuksen, joka on erilainen joka pelikerralla. Käsin tehdyt kentät ovat kuitenkin vahvimmillaan silloin, kun ne sisältävät pulmatehtäviä tai ne vievät pelin juonta eteenpäin.

Tietoperustana opinnäytetyössä käytettiin tutkimuspapereita ja artikkeleita koskien proseduraalista generointia. Pelisuunnittelun liittyvät lähteet olivat kirjallisia sekä internetlähteitä. Lisäksi ohjelmointiosuudessa käytettiin apuna tutoriaalia, johon tehtiin kuitenkin myös muutoksia.

Opinnäytetyön ohessa tehtiin myös Unity-pelimoottorilla prototyyppi pelistä, jossa proseduraalista generointia käytettiin generoimaan huoneita ja esteitä pelikentälle. Peli on nimeltään Broomstick Bash, ja se on ottanut inspiraatiota Jetpack Joyride -pelistä. Pelissä lennetään noidalla ja väistellään esteitä. Generointi on melko yksinkertaista, mutta sisältää myös rajoituksia esteiden luomisen suhteen. Opittiin kuitenkin paljon siitä, miten proseduraalinen generointi toimii, ja peliä on mahdollista laajentaa ja jatkokehittää.

Asiasanat: Pelit, Pelikehitys, Pelisuunnittelu, Peliohjelmointi

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Business Information Technology

Author(s): Maria Martin
Title of thesis: Use of procedural generation in game development
Supervisor(s): Matti Viitala
Term and year when the thesis was submitted: Autumn 2022
Number of pages: 31

The aim of this thesis was to research the use of procedural generation in game development, especially in level design as well as consider its advantages and disadvantages. It was also compared to traditional hand-made level design. Example games used in the thesis were Rogue, published in 1980, and Jetpack Joyride, published in 2011. Procedural generation is a central part of both of those games. It was found that procedural generation reduces the workload of the level designer and gives the player an experience that is different every time they play. However, handmade levels are at their strongest when they contain puzzles, or they advance the game's plot.

Research papers and articles regarding procedural generation were used as sources in the thesis. The sources related to game design were online articles and published books. In addition, I used a tutorial to help me in the programming part, but I also made changes to it.

A prototype of a game was also made using the Unity game engine. Procedural generation was used to generate rooms and obstacles on the level. The game is called Broomstick Bash and it was inspired by Jetpack Joyride. In the game the player flies as a witch and dodges obstacles. The generation is quite simple, but also has limitations regarding the creation of obstacles. I learned a lot about how procedural generation works, and it is possible to expand and further develop the game on a later date.

Keywords: Games, Game Development, Game Design, Game Programming

SISÄLLYS

1 JOHDANTO ... 6
2 PROSEDURAALINEN GENEROINTI ... 7
2.1 Proseduraalisen generoinnin määritelmä ... 7
2.2 Esimerkkipelejä ... 8
2.2.1 Rogue ... 8
2.2.2 Jetpack Joyride ... 9
2.3 Vertaus perinteiseen kenttäsuunnitteluun ... 11
2.3.1 Pelikentän määritelmä ... 11
2.3.2 Peli- ja kenttäsuunnittelun merkitys ... 11
2.3.3 Perinteinen kenttäsuunnittelu ... 13
2.3.4 Proseduraalisesti generoidut kentät ... 14
2.4 Muu käyttö ... 15
3 OMAN PELIN TOTEUTUS ... 16
3.1 Kuvaus pelistä ... 16
3.2 Unity työvälineenä ... 17
3.3 Suunnittelu ... 18
3.4 Graafikkojen rooli ... 19
4 GENEROINNIN TOTEUTUS ... 21
4.1 Huoneiden generointi ... 22
4.2 Objektien generointi ... 25
5 POHDINTA ... 28
LÄHTEET ... 30

1 JOHDANTO

Opinnäytetyö käsittelee proseduraalisen generoinnin käyttöä pelinkehityksessä. Proseduraalinen generointi tuottaa peliin sisältöä algoritmien avulla, ja se on ollut käytössä pelialalla jo alusta asti. Sisältö voi olla oikeastaan mitä vaan, esimerkiksi kenttiä, maailmoja, vihollisia ja hahmoja. Generointi hyödyntää satunnaisuutta ja luo siten uusia, erilaisia kokemuksia pelaajalle sekä myös pienentää kehittäjän työmäärää, kun kenttiä ei tarvitse tehdä käsin. Raportissa kerron tarkemmin, mitä proseduraalinen generointi on ja pohdin sen hyötyjä ja haittoja verrattuna perinteiseen kenttäsuunnitteluun. Osana opinnäytetyötä tehtiin myös 2D-mobiilipelin prototyyppi, jossa proseduraalista generointia käytettiin. Kerron siis myös pelin kehityksestä ja toteutuksesta proseduraalisen generoinnin näkökulmasta.

Pelin idea syntyi Oulun ammattikorkeakoulun GameLabissä, joka oli osa opintojani aiemmin, mutta se ei päässyt siellä kehitykseen. Minä ja yksi GameLabin käynyt visuaalisen suunnittelun opiskelija tykkäsimme kuitenkin ideasta, joten päätimme toteuttaa sen osana opinnäytetyötä. Mukaan liittyi vielä toinenkin visuaalisen suunnittelun opiskelija, eli projekti toteutettiin kolmen hengen ryhmässä. Tämä peli-idea herätti kiinnostukseni proseduraaliseen generointiin. Olin toki kuullut siitä jo aiemmin, mutta tämä projektin takia halusin myös päästä kokeilemaan sitä itse. Pelissä generoitiin vaihtuvia taustoja sekä objekteja, ja se toteutettiin Unity-pelimoottorilla.

Proseduraalinen generointi on mielestäni kiinnostavaa sen luomien mahdollisuuksien takia. Lisäksi halusin pohtia, milloin sitä kannattaa käyttää sen sijaan, että tekisi kentät käsin. Proseduraalisen generoinnin avulla voi tehdä melko laajankin pelin vähemmällä työmäärällä. Kuitenkin haittapuolena voi olla pelikenttien geneerisyys ja merkityksettömät tai pintapuoliset muutokset, jotka eivät anna pelaajalle mieleenpainuvia kokemuksia. Siksi jo suunnitteluvaiheessa on tärkeää miettiä, onko proseduraalinen generointi peliin sopivaa.

2 PROSEDURAALINEN GENEROINTI

2.1 Proseduraalisen generoinnin määritelmä

Yksinkertaisesti ilmaistuna proseduraalinen generointi tarkoittaa pelisisällön luontia algoritmia käyttäen (Togelius ym. 2011). Kyseinen sisältö voi olla esimerkiksi peliin liittyviä kenttiä, hahmoja, karttoja, aseita, tekstuureja ja musiikkia. Pelattavuuden kannalta on tärkeää, että sisältöä tuottava ohjelma ottaa huomioon pelin suunnittelun, rajoitukset sekä mahdollisuudet – jokainen generoitu kenttä on pystyttävä suorittamaan läpi ja jokaista asetta on pystyttävä käyttämään. (Shaker, Togelius & Nelson 2016, 1–2.)

Proseduraalista generointia on kuitenkin vaikea määritellä tarkasti, ja voikin olla helpompi määritellä mitä se ei ole, ja mitä se voi olla. Togelius ym. (2011) mukaan se ei ole pelaajien luomaa sisältöä. Monet offline-pelit, eli pelit, joita pelataan ilman nettiyhteyttä, sisältävät kenttäeditorin, karttaeditorin, hahmoeditorin tai jonkin muun työkalun, jonka avulla pelaaja voi luoda omaa sisältöä. Sisältö ei kuitenkaan ole proseduraalisesti generoitua, koska sen on luonut ihminen. Myöskään online-strategiapeleissä, kuten esimerkiksi Civilization-pelissä, pelaajan rakentama kaupunki ei täytä määritelmää, vaikka pelaaja ei voikaan kontrolloida esimerkiksi kaupungin kasvunopeutta. Pelaajan toiminta, kuten kaupungin rakentaminen tiettyyn paikkaan, voi olla osa proseduraalista generointia, mutta pelaajan tarkoituksellisesti luoma sisältö ei itsessään sitä ole. Peli voi kuitenkin generoida reaktioita kaupungin rakentamiseen.

Satunnaisuus ja adaptiivisuus ovat asioita, jotka voivat olla proseduraalista generointia. Satunnaisuus on usein osa generointia, sillä se auttaa luomaan sisältöä, joka ei ole ennakoitavissa. Sillä on kuitenkin rajansa, sillä peli ei voi generoida sekalaisia elementtejä pelikentälle ilman minkäänlaisia rajoituksia, koska sellaista peliä olisi mahdotonta pelata. Adaptiivisuus taas tarkoittaa sitä, että peli mukautuu ja luo uutta sisältöä sen mukaan, mitä pelaaja on pelissä tehnyt. Se voi esimerkiksi säätää pelin vaikeustasoa riippuen siitä, kuinka taitava pelaaja on. Satunnaisuus ja adaptiivisuus eivät kuitenkaan aina tarkoita, että peli käyttää proseduraalista generointia, eivätkä ne myöskään ole vaatimuksia sille. (Sama.)

2.2 Esimerkkipelejä

2.2.1 Rogue

Rogue (1980) on yksi vanhimmista peleistä, jossa on käytetty proseduraalista generointia kenttien luomiseen (Shaker ym. 2016). Pelin tavoitteena on löytää tie luolaston läpi, taistella hirviöitä vastaan ja löytää aarteita, jotka auttavat pelaajaa. Mikäli hahmo kuolee, peli alkaa alusta. Peli generoi uudet kentät uusilla vihollisilla, aarteilla ja huoneilla aina, kun uusi peli alkoi. Rogue oli aikanaan erittäin vaikutusvaltainen ja inspiroi useita pelejä sekä genren *roguelike*. (Wikipedia 2021.)

Rogue loi kenttiä niin, että niissä oli huoneita, käytäviä, vihollisia ja aarteita, kuten esimerkiksi aseita tai taikaesineitä. Huoneet sijoittuivat 3x3-kokoiseen ruudukkoon ja yhdistyivät käytävillä, ja yhdessä huoneessa täytyi olla portaat seuraavaan kenttään. (Wikipedia 2021.)

Kuvio 1 sisältää kuvankaappauksen Rogue-pelistä Manleyn (2013) videolta. Erikokoiset huoneet yhdistyvät #-merkeillä kuvatuilla käytävillä, ja portaat seuraavalle tasolle merkataan %-merkillä. Pelihahmoa kuvaa @-merkki ja käärmettä S-kirjain. Manley kertoo myös, että Roguen tekijät halusivat pelin olevan pelattavissa myös heille itselleen, mikä ei olisi ollut mahdollista tai hauskaa, jos kaikki kentät olisi suunniteltu käsin. Proseduraalinen generointi pitää siis pelin kiinnostavana. Löydettävät aarteetkaan eivät ole tunnistettavissa ennen kuin niitä käyttää. Niiden ominaisuudet ovat myös satunnaisgeneroituja, jolloin ne saattavat olla joskus myös haitallisia ja esimerkiksi tehdä vihollisista voimakkaampia tai nopeampia. (Manley 2013.)

KUVIO 1. Kuvakaappaus Rogue-pelistä (Manley 2013)

2.2.2 Jetpack Joyride

Jetpack Joyride (2011) on Halfbrick Studiosin Androidille julkaisema peli, jossa pelihahmo lentää rakettirepulla esteitä väistellen. Pelissä on vain yksi kenttä, joka on kuitenkin joka kerralla erilainen ja vaihtelee myös pelaamisen aikana. Lentäessä taustat vaihtuvat sekä kolikoita ja esteitä ilmestyy kentälle satunnaisessa järjestyksessä. Silloin tällöin peli luo myös ”power up”-esineitä, jotka keräämällä hahmo saa hetkellisesti uusia kykyjä. Power upit kestävät siihen asti, kunnes pelaaja osuu johonkin esteeseen.

Muscat (2012) kertoo, kuinka pelin kehitysvaiheessa testaajien palaute nosti esille kaksi ongelmaa. Peli oli liian intensiivinen ja tylsä, minkä takia pelaajat kyllästyivät nopeasti. Testipelaajat kokivat pelin liian vaikeaksi, koska olivat jatkuvasti yhden osuman päässä kuolemasta. Ratkaisuksi keksittiin menopeli-power upit, jotka sekä vähentävät intensiivisyyttä että tarjoavat pelaajalle jotain uutta tekemistä. Menopelit muuttavat hahmon ohjausta. Esimerkiksi moottoripyörä ajaakin maata pitkin ja painovoimapuku päällä hahmo juoksee vuorotellen katossa ja maassa. Lisäksi menopeli antaa pelaajalle tavallaan yhden lisäelämän, sillä esteeseen osuessaan pelaaja menetti vain menopelinsä, mutta pystyi silti jatkamaan peliä.

Peli ei generoi esteitä ilmestymään niin, ettei niiden ohi voisi päästä, koska silloin pelaamisesta tulisi mahdotonta. Se ei myöskään generoi pelkkiä esteitä, vaan tasapainottelee pelaajalle mieluisten asioiden ja haasteiden välillä. Kolikot ilmestyvät kentälle eräänlaisissa rykelmissä, ja menopelit ilmestyvät tietyin väliajoin ja vain yksi kerrallaan. Pelaajan poimiessa menopelin myös ruudulla jo olevat esteet tuhoutuvat ja peli generoi hetken kuluttua kaksi kolikkorykelmää peräkkäin ennen kuin esteet jatkuvat (Muscat 2012.) Sama tapahtuu, kun menopeli menetetään. Pelaajan kannalta olisi ärsyttävää, jos hahmo osuisi esteeseen käytännössä heti menopelin saamisen jälkeen, koska sen väistäminen voi olla liian vaikeaa. Se tekisi menopelin poimimisesta rankaisevaa.

Kuviot 2 ja 3 ovat kuvakaappauksia Jetpack Joyride -pelistä. Kuviossa 2 on scifi-tyylinen metalliseinäinen laboratoriotausta sekä laseresteitä, joihin osuessa peli päättyy. Kuviossa 3 taas on jono kolikoita pelaajalle kerättäväksi, ja tausta on vaihtunut luolastomaiseen. Taustoista voi havaita toistuvuutta, esimerkiksi luolaston kiviseinän kuvio on toistuvaa, joskin sitä välillä rikotaan hehkuvilla aukoilla. Samoin metalliseinän ovi, venttiili sekä keltainen kuvio toistuvat.

KUVIO 2. Kuvakaappaus Jetpack Joyride -pelistä

KUVIO 3. Kuvakaappaus Jetpack Joyride -pelistä

2.3 Vertaus perinteiseen kenttäsuunnitteluun

2.3.1 Pelikentän määritelmä

Pelikenttä tai taso voidaan määritellä eri tavoin pelistä ja ihmisestä riippuen. Se voi tarkoittaa tiettyä ympäristöä tai paikkaa pelimaailmassa, missä pelaaminen tapahtuu. Se voi myös tarkoittaa tapaa jakaa pelin fyysinen tila tiettyjen pelikokemusten mukaan tai merkitä pelaajan etenemistä peleissä, joissa pelattavuus toistaa itseään. Joskus kenttiä kutsutaan myös esimerkiksi kartoiksi tai maailmoiksi. (Rogers 2014, 209–211.) Tässä raportissa keskitytään enimmäkseen ensimmäiseen määritelmään, eli ympäristöön tai paikkaan, missä pelaaminen tapahtuu.

Pelikentällä on kuitenkin aina jonkinlainen pelialue, jossa pelaaja on vuorovaikutuksessa pelin kanssa. Se tarkoittaa sellaista paikkaa, jossa pelaajan on toteutettava jokin haaste tai tavoite tai tehdä jokin tietty asia päästäkseen eteenpäin. Pelialueen ei välttämättä tarvitse olla rajattu, vaan pelistä riippuen sieltä voi olla mahdollista poistua ja palata takaisin myöhemmin. (Wilson 2018.) Näin voi olla esimerkiksi avoimen maailman peleissä.

2.3.2 Peli- ja kenttäsuunnittelun merkitys

Pelisuunnittelu on tärkeä vaihe pelin tuotannossa, sillä ilman sitä koko pelistä ei tulisi mitään. Se on kuin piirustus, jonka avulla peli toteutetaan. Pelin muoto, säännöt, juoni ja muut ominaisuudet

ovat kaikki osa suunnittelua. Niiden lisäksi hyvä suunnittelu auttaa luomaan hyvän pelikokemuksen, välttelemään virheitä tuotannon aikana sekä minimoimaan riskejä. (Manninen 2007, 28–32.)

Kenttäsuunnittelu on osa pelisuunnittelua, ja erityisesti pienissä pelistudioissa sekä peli- että kenttäsuunnittelija ovat usein sama henkilö. Isoissa studioissa näissä rooleissa voi olla useitakin henkilöitä, joista jokainen keskittyy johonkin tiettyyn osuuteen, kuten vaikka taistelumekaniikkaan tai hahmoihin. Kenttäsuunnittelija päättää kenttien teeman, tarkoituksen ja tavoitteen sekä rakentaa kentät niin, että ne ovat ratkaistavissa tietyllä tavalla. Teemalla tarkoitetaan kentän visuaalista ilmettä ja ympäristöä sekä niihin liittyviä mahdollisia erilaisia mekaniikkoja tai vihollisia. Esimerkiksi talviteemaisessa kentässä pelihahmo voi liukua liikkuessaan jään yli, ja avaruusasemalle sijoittuvassa kentässä taistellaan avaruusolentoja vastaan. (Rogers 2014, 212.)

Kenttien tarkoitus ja tavoite kulkevat usein käsi kädessä. Rogersin (2014, 223–225) mukaan pelikentillä on usein yksi neljästä tavoitteesta:

1. pakene tai selviä
2. tutki
3. opeta
4. moraalinen viesti.

Pakene tai selviä -tavoitteessa pelaajan on selvittävä hengissä vaikeassa tilanteessa, jolloin pelattavuus on yleensä nopeasti etenevää ja toiminnallista. Tutkimistavoitteessa pelaaja etenee yleensä rauhallisemmin pelimaailmaa tutkien, ja pelin tarina voi avautua ympäristöstä löytyvien vihjeiden perusteella. Opetustavoite on yleensä opettavaisissa peleissä, mutta myös viihdepelit voivat opettaa esimerkiksi historiaa. Pelissä voi olla myös jokin moraalinen viesti, tai se voi myös haastaa pelaajan omaa moraalia esimerkiksi pakottamalla valitsemaan kahden eri vaihtoehdon välillä ja kohtaamalla seuraukset myöhemmin.

Suunnittelijan pitää vielä muistaa kentän tarkoitus, eli syy, miksi kenttä on ylipäätään olemassa. Se voi esimerkiksi opettaa uuden mekaniikan tai laajentaa vanhaa, tai ylipäätään vain opettaa, miten peliä pelataan. Näiden asioiden miettiminen on pelisuunnittelijalle tärkeää, sillä se auttaa rakentamaan kentän niin, että pelaaja saavuttaa kyseiset tavoitteet. (Rogers 2014, 225.) Suunnitteluvaiheessa myös päätetään, kuinka kentät toteutetaan tuotantovaiheessa – käsin vai algoritmilla, eli proseduraalisesti generoimalla. On toki myös mahdollista toteuttaa vain osa kentästä tai kentistä

proseduraalisesti. Esimerkiksi kentän rakenne voi pysyä samana, mutta viholliset tai aarteet generoituvat aina uusiksi, tai luolaston kymmenestä kentästä kentät 5 ja 10 ovat aina samat, mutta muut vaihtelevat. Täten perinteinen kenttäsuunnittelu sekä proseduraalisesti generoidut kentät eivät sulje toisiaan pois, vaan molempia voidaan käyttää samassa pelissä.

2.3.3 Perinteinen kenttäsuunnittelu

Perinteisessä kenttäsuunnittelussa kentät rakennetaan ihmisen toimesta esimerkiksi kenttäeditorin avulla. Käsintehtyjen kenttien vahvuuksia ovat muun muassa kenttien laatu, merkityksellisyys ja tarinankerronta. Erityisesti se auttaa luomaan tiettyjä hetkiä pelissä, kuten olennaisia osia tarinaa, erityisiä haasteita pelaajalle tai vaikkapa erityisen merkittäviä esineitä tai vihollisia. Näitä proseduraalinen generointi harvoin pystyy tekemään, sillä mikäli algoritmi on niin tarkka ja rajattu, että lopputulos on aina sama, se ei eroa juuri mitenkään ihmisen tekemästä kentästä. Tällöin on järkevämpää luoda kentät käsin suunnittelijan haluaman pelattavuuden tai hetken ympärille. (Johnson 2017.)

Perinteinen kenttäsuunnittelu sopii erityisesti sellaisiin peleihin, joissa tarina, pelimaailma ja pelaajan immersio ovat tärkeässä osassa tai joissa ratkotaan tehtäviä tai ongelmia. Pulmapelien kenttien on oltava ratkaistavissa tietyssä määrässä liikkeitä ja tiettyjä mekaanikoita käyttäen, mikä tekee niiden generoimisesta erittäin haasteellista. Mitä useampi askel ratkaisuun sisältyy, sitä monimutkaisempaa pulmakentän rakentaminen on, ja algoritmin täytyisi käydä läpi kaikki mahdolliset vaihtoehdot ja sitten generoida kenttä, joka ei ole liian yksinkertaisesti ratkaistavissa. Käsin tehdyt pulmakentät ovat usein myös tehty nimenomaan pelaajan nähtäväksi: pelaaja näkee tietyt elementit tai ominaisuudet ennen toisia, tai jonkin komponentin tarkoituksen ymmärtää vasta silloin, kun pelaaja on käyttänyt aikaa jonkin toisen ominaisuuden parissa. Algoritmin siis täytyisi jollain tavalla osata arvioida, millä tavalla pelaaja näkee ja kokee kentän ja siinä sijaitsevat elementit. (Johnson 2017.)

Perinteisen kenttäsuunnittelun yhtenä heikkoutena voisi mainita sen hitauden ja kalleuden. Ihmistyöntekijä on kallis, ja mitä suurempi peli, sitä enemmän työntekijöitä tarvitaan. Samoin myös pelinkehitykseen vaadittava aika pitenee, sillä hyvin suunnitellut kentät vievät aikaa. (Shaker ym. 2016.)

Peli on myös uusi vain ensimmäisellä pelikerralla, eikä pelaajaa välttämättä kiinnosta pelata peliä toista kertaa. Pelaaja voi myös oppia kenttien ratkaisut ja strategiat ulkoa, jolloin pelistä voi tulla pelaajalle liian helppo. (Johnson 2017.)

2.3.4 Proseduraalisesti generoidut kentät

Proseduraaliset kentät luodaan algoritmin avulla. Algoritmit voivat luoda useita erilaisia kenttiä nopeasti vaatimusten mukaan. Tämä nopeuttaa pelinkehitystä sekä tekee pelistä laajemman pienemmällä työmäärällä. Se myös lisää pelin uudelleenpelattavuutta, sillä peli generoi uusia kenttiä joka pelikerralle. Tämän takia proseduraalisesti generoituja pelejä ei voi varsinaisesti ikinä pelata kokonaan läpi. (Johnson 2017.)

Algoritmipohjainen kenttäsuunnittelu on myös havaittu hyväksi keinoksi luoda peliin haastetta, mahdollisesti siksi, että sitä käytetään usein roguelike-peleissä. Pelaaja ei tiedä, mitä seuraavaksi tapahtuu tai minkälainen kenttä on seuraavaksi edessä, eikä täten voi oppia strategiaa tai aarteiden paikkoja ulkoa. Pelaajan täytyy onnistua reagoimaan oikein heti ensimmäisellä kerralla aina, kun pelissä tapahtuu jotain. (Sama.)

Proseduraalisesti generoitujen kenttien yhtenä heikkoutena onkin kenttien geneerisyys. Kentät näyttävät erilaisilta, mutta todellisuudessa erot voivat olla hyvin pinnallisia, sillä generoiduilla kentillä ei yleensä ole minkäänlaista merkittävää tarkoitusta tai edistymisen tunnetta. Ne ikään kuin koostuvat palasista, jotka järjestellään uudelleen satunnaisella tavalla. Lopulta uusi kenttä on vain yksi kenttä muiden joukossa, eikä sisällä varsinaisesti mitään erityistä mieleenpainuvaa kokemusta pelaajalle. Täten generoitua kenttää voi harvoin käyttää pelisuunnittelijan taitojen todisteena tai näytteenä suunnittelun innovaatiosta. (Togelius ym. 2013.)

Proseduraalinen generointi sopiikin hyvin peleihin, jotka eivät ole niin vahvasti tarinasta riippuvaisia. Se on myös hyvä peleihin, joissa pelaajan halutaan tutkivan laajaa pelimaailmaa, tai endless runner -tyylisiin peleihin, joissa pitää esimerkiksi päästä mahdollisimman pitkälle.

2.4 Muu käyttö

Proseduraalisen generoinnin käyttömahdollisuudet eivät tietenkään rajoitu vain kenttägenerointiin. Kenttien lisäksi esimerkiksi maaston tai jopa kokonaisten pelimaailmojen luonti onkin yleisimpiä tapoja käyttää proseduraalista generointia. Maasto onkin erityisen yleinen, sillä melkein kaikissa kolmiulotteisissa peleissä on jonkinlaista maata, jonka päällä pelaaja kävelee tai ajaa. Maastossa on myös yleensä ainakin jonkin verran korkeusvaihteluita sekä kasvillisuutta, pelin tarpeiden mukaan. Maastolla voi myös olla mekaaninen tarkoitus pelkän visuaalisuuden lisäksi, esimerkiksi vuoristoa voi käyttää rajaamaan pelialuetta tai estämään pelaajaa näkemästä tietylle alueelle. Nämä tarkoitukset on otettava huomioon myös generointiskriptissä. Maaston generointiin on monia erilaisia metodeja ja tekniikoita, jotka valitaan pelin vaatimusten mukaan. (Shaker ym. 2016.)

Spore-peli käytti proseduraalista generointia hahmojen luontiin. Pelaaja pystyi rakentamaan erinäköisiä olentoja haluamistaan palasista, minkä jälkeen olennon käyttäytymismalli generoitiin sen fyysisten erityspiirteiden mukaan. Esimerkiksi jos pelaaja halusi olentonsa tulevan toimeen muiden olentojen kanssa, hän saattoi valita enemmän sosiaalisia osia. Kehittäjien ei siis tarvinnut luoda tuhansia 3D-malleja eri hahmovaihtoehdoista, vaan ne generoitiin pelaajan luoman ”reseptin” perusteella. Sen avulla myös tiedostokoot pystyttiin pitämään pieninä. Spore käytti proseduraalista generointia myös musiikissa. Siitä tuli iloisempaa, kun pelaaja lisäsi hahmolleen sosiaalisen osan, ja uhkaavampaa, kun aggressiivinen osa valittiin. (Naone 2008.)

3 OMAN PELIN TOTEUTUS

3.1 Kuvaus pelistä

Peli, työnimeltään *Broomstick Bash*, kehitetään Android-alustalle, ja se on saanut inspiraatiota aiemmin esimerkeissä mainitusta Jetpack Joyridestä. Peli kertoo noidasta, joka on menettänyt voimansa, ja hänen täytyy kerätä maagista energiaa lentämällä saadakseen voimansa takaisin. Lentäminen on kuitenkin vaarallista, ja noidan täytyy väistellä esteitä ja vihollisia. Tavoitteena on lentää mahdollisimman pitkälle, ja peli loppuukin silloin, kun noita osuu johonkin esteeseen.

Peliä pelataan 2D-sivunäkymässä, ja lentäminen tapahtuu näyttöä naputtelemalla. Noita liikkuu jatkuvasti eteenpäin ja lentää ylöspäin, kun pelaaja naputtelee näyttöä, ja alas silloin, kun pelaaja ei koske näyttöön. Pelissä on vain yksi proseduraalisesti generoitu kenttä, joka on joka kerralla erilainen. Sen tavoitteena on Rogersin neljästä tavoitteesta ensimmäinen, eli pakene tai selviä. Vaikka pelissä ei varsinaisesti paeta mistään, on mahdollisimman pitkälle lentäminen eräänlaista selviytymistä.

Kuvio 4 on kuvakaappaus pelin prototyypistä. Noita liikkuu automaattisesti eteenpäin kiihtyvällä nopeudella, ja pelaaja ohjaa noidan lentokorkeutta. Ympyrässä olevat hehkuvat oliot ovat maagista energiaa, jota pelaaja kerää. Kerätty määrä näkyy laskurissa vasemmassa yläkulmassa. Sininen lintu on este, ja peli päättyy noidan osuessa siihen.

KUVIO 4. Kuvakaappaus Broomstick Bash -pelistä

3.2 Unity työvälineenä

Projektin tärkein työväline on Unity-pelimoottori, joka on erittäin suosittu pelinkehitysohjelma. Se valittiin projektiin, sillä sen perusversio on saatavilla ilmaiseksi, se on jo tuttu ohjelma ryhmälle sekä siksi, että sille on saatavilla paljon tutoriaaleja ja ohjeita.

Unity on monialustainen pelimoottori, mikä tarkoittaa, että sillä voi kehittää pelejä sekä mobiili- että konsoli- ja tietokonealustoille. Yhteensä Unityllä voi tehdä pelejä jopa 25:lle eri alustalle, ja sen takia onkin mahdollista kehittää peli sekä Android- että iOS-puhelimille samaan aikaan alustojen eroista huolimatta. Unity tunnetaankin pääasiallisena mobiili- ja indiepelien kehitysympäristönä. Siihen vaikuttaa tietysti myös, että Unity on ilmainen pienimuotoisille projekteille, mikäli kehittäjä ei tienaa pelillään yli 100 000 dollaria vuodessa. Tämän takia myös aloittelijan on helppo kokeilla pelin tekemistä vaikka vain harrastuksena ilman rahallisia sitoutumisia. Unityllä on myös mahdollista tehdä 2D-, 3D-, VR- ja AR-pelejä, eli se ei rajoitu vain tiettyyn tyyppiin. Se on kehittäjien kannalta hyvä asia, sillä heidän ei tarvitse opetella käyttämään uudenlaista pelimoottoria esimerkiksi VR-pelin kehitystä varten. (Dealessandri 2020.)

Unity on myös erittäin aloittelijaystävällinen, ja sen avulla voi kehittää pelejä jopa hyvin vähällä koodauksella. Pelimoottorin yhteisö on myös vahva, ja niin aloittelevat kuin kokeneetkin kehittäjät voivat kysellä apua ja vinkkejä. Unity Asset Storea ei ole myöskään syytä unohtaa – se on paikka,

josta löytyy lukuisia erilaisia assetteja, eli esimerkiksi valmiiksi tehtyjä hahmoja, animaatioita, tekstuureja, ääniä ja grafiikoita joko ilmaisina tai maksullisina. Niitä käyttämällä pelkkä yksi koodarikin voi saada aikaan näyttävän pelin, vaikkei osaisikaan itse tehdä grafiikkaa tai animaatioita. Käyttäjät voivat myös itse ladata tekemiään assetteja myyntiin. (Arnia Software 2020.)

Pelimoottorina Unityllä on sisäänrakennettuja ominaisuuksia, jotka helpottavat pelin tekemistä, kuten vaikka fysiikat, törmäyksen tunnistaminen, valaistus ja miten valo heijastuu erilaisista objekteista (Sinicki 2021). Täten käyttäjän ei tarvitse itse koodata esimerkiksi painovoimaa, vaan pelihahmolle voi vain laittaa Rigidbody-komponentin, jolloin hahmo putoaa alas, mikäli sen alla ei ole mitään. Samalle hahmolle voi myös laittaa törmäyskomponentin, jolloin hahmon liike esimerkiksi pysähtyy, kun se törmää johonkin toiseen objektiin, jolla on myös törmäyskomponentti. Myöskään valon heijastumista ei tarvitse itse koodata, sillä Unitylle voi vain sanoa, että yksi objekti heijastaa valoa ja toinen ei.

Unityllä on toki myös heikkoutensa. Se ei esimerkiksi ole kovin hyvä erittäin suurille projekteille, sillä se ei välttämättä anna muokata tarvittavia asioita tai optimoida asioita tarpeeksi syvällisesti. Käytännössä siis samat asiat, mitkä tekevät Unitystä helpon käyttää, ovat myös rajoittavia tekijöitä. (Dealessandri 2020.)

3.3 Suunnittelu

Peli toteutettiin Unity-pelimoottorin avulla kolmen opiskelijan ryhmässä, johon sisältyy kaksi graafikkoa ja yksi ohjelmoija. Proseduraalinen generointi oli kaikille uutta, joten ryhmälle olikin erityisen tärkeää miettiä yhdessä, mikä olisi paras tapa toteuttaa peli halutulla tavalla. Päätöksiin vaikutti myös, että tuotantoaikaa oli vain kymmenisen kuukautta ja sitä tehtiin osa-aikaisesti muiden opintojen ohessa, ja ryhmässä oli vain yksi ohjelmoija. Näiden asioiden takia oli parempi valita yksinkertaisempia ratkaisuja.

Jo alkuvaiheessa tiedettiin, että tuotannossa tehdään vain pelin prototyyppi, jossa on vain muutama suunnitelluista mekaanikoista. Tärkeimpänä niistä oli tietysti proseduraalinen generointi. Peli suunniteltiin kuitenkin paljon laajemmaksi kuin prototyyppiversio, jotta sen kehitystä voi jatkaa myöhemmin. Esimerkiksi täydessä versiossa noidan kissa pystyisi taistelemaan vihollisia vastaan, mutta koska se ei ole osa ydinpeliä, se ei ole mukana prototyypissä.

Proseduraaliseen generointiin sisältyy siis pelikentän luonti. Kentän taustan täytyy vaihtua vähän väliä, sekä kentälle täytyy ilmestyä sekä kerättävää maagista energiaa että satunnaisia esteitä satunnaisiin paikkoihin. On kuitenkin tärkeää huomioida muun muassa se, etteivät esteet voi ilmestyä kentälle niin, ettei niiden ohi pääse. Päätettiin, että yhdenlainen tausta toimii yhtenä kokonaisuutena, eli huoneena, johon liitetään myös esteet ja kerättävä energia. Käytännössä generointia tapahtuu kahdesti: ensin generoidaan huone ja sitten huoneeseen sisältöä.

3.4 Graafikkojen rooli

Graafikot ovat nimensä mukaisesti vastuussa pelin grafiikoista, eli siitä, miltä peli näyttää. Graafikot suunnittelivat pelin värimaailman sekä hahmon ulkonäön yhdessä, ja sen jälkeen jakoivat tehtävänsä keskenään. Yhtenäisen tyylin löytäminen oli graafikoille tärkeää, jotta molemmat voivat luoda peliin sopivia kuvia. Tarvittavia grafiikoita oli tietysti monenlaisia, eli taustoja, nappeja, pelihahmo, objekteja sekä animaatioita. Animaatioista graafikot tekivät sprite sheetin. Se on tiedosto, joka sisältää yhden tai useamman animaation kuvat vierekkäin työskentelyä helpottaen ja tilaa säästäen. Graafikot tallensivat kaikki tiedostot Unity-projektiin niiden valmistuessa, jotta ne olivat helposti kaikkien saatavilla.

Proseduraalisen generoinnin suhteen graafikoilla olikin vain yksi asia mietittävänään, eli miten pelikentän taustat voidaan toteuttaa niin, että vaihtuvuus on sulava tai se sopii peliin muulla tavalla. Tausta ei voi vaihtua esimerkiksi metsäisestä maisemasta kaupunkinäkymään äkillisesti ilman minkäänlaista siirtymää. Lisäksi oli huomioitava, että taustojen järjestys on satunnainen, eli seuraava tausta voi olla millainen tahansa. Tämä ongelma ratkaistiin lisäämällä taustojen alku- ja loppupäähän sumua. Käytännössä graafikot tekivät taustoille erilliset päätypalaset, joissa oli yhteen sulautuvaa sumua, ja ne kiinnitettiin Unityssä taustakuvaan. Se auttoi eri taustoja sulautumaan paremmin yhteen, eikä generoinnissa tarvitse ottaa huomioon, minkälainen maisema on seuraavaksi tulossa, vaan se saa olla täysin satunnainen.

Kuvio 5 kuvaa taustojen vaihtumiskohtaa. Violetti kimalteleva sumu peittää eri taustojen välisen sauman ja tekee siten siirtymästä luonnollisemman näköisen. Sumu on kiinni molempien huoneiden päätypaloissa.

KUVIO 5. Kuvakaappaus taustojen vaihtumiskohdasta

4 GENEROINNIN TOTEUTUS

Broomstick Bashin generointiskriptin lähtökohtana käytettiin Placzekin (2018) tutoriaalia, joka ohjeistaa kehittämään Jetpack Joyriden kaltaisen pelin. Se oli helppo valinta peliä varten, sillä Broomstick Bashin tärkeimmät elementit, kuten pelattavuus ja proseduraalinen generointi, ovat samankaltaisia Jetpack Joyriden kanssa. Tutoriaalissa oli hyviä ohjeita myös muihin osuuksiin, mutta generointiskripti oli tärkeimmässä roolissa.

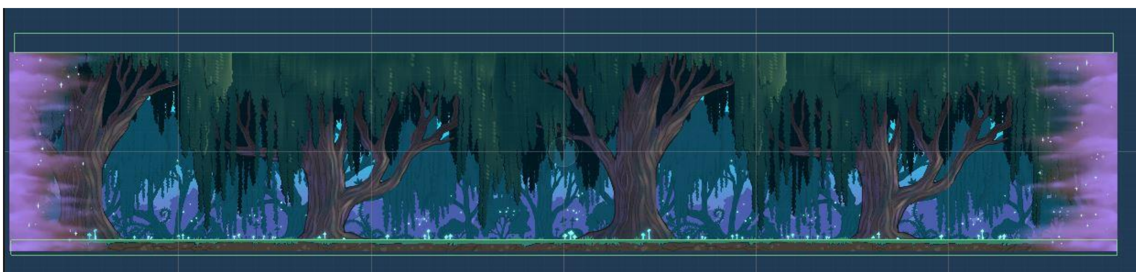
Pelissä generoidaan käytännössä vain kahta asiaa: huoneita ja objekteja eli kerättävää energiaa tai esteitä. Huoneen generoinnissa käytettiin apuna Placzekin tutoriaalia. Alkuvaiheessa siihen ei ollut tarvetta tehdä muutoksia, sillä olimme päätyneet jo suunnitteluvaiheessa mahdollisimman yksinkertaiseen ratkaisuun, johon tutoriaalin koodi riitti hyvin. Myöhemmin kuitenkin totesimme, että emme halua saman huoneen toistuvan peräkkäin, eli uuden huoneen tulisi aina olla eri kuin edellisen, ja se vaati myös koodimuutoksia. Yksi huone siis koostuu yleensä yhdenlaisesta taustasta sekä taustaan sopivista sumupaloista, mutta teimme myös variaatioita ja eri pituisia versioita huoneesta. Huoneet tallennettiin prefabeiksi, eli valmiiksi kokonaisuuksiksi, ja lisättiin sitten huonelistaan. Prefabit ovat Unityn menetelmä tallentaa peliobjekti sekä kaikki siihen liittyvät asetukset, skriptit ja komponentit yhdeksi tiedostoksi, joka on helposti uudelleenkäytettävissä. Se on myös helposti muokattavissa, ja muokkaukset siirtyvät automaattisesti kaikkialle, missä kyseistä prefabia on käytetty. (Unity Technologies 2022b.)

Kuvio 6 kuvaa varhaista vaihetta pelin kehityksessä. Tausta saatiin valmiiksi nopeasti, mutta pelihahmoa eli noitaa kuvaa vain valkoinen pallo, ja maaginen energia on rykelmä sinisiä palloja. Protografiikoita käytettiin, jotta ohjelmoija voi aloittaa oman osuutensa ja kirjoittaa koodia jo valmiiksi. Virallisten grafiikoiden teko vie aikaa, eikä ohjelmoijan kannata jäädä niitä odottelemaan, sillä protografiikoiden vaihto lopullisiin ei ole iso toimenpide. Pallorykelmä tehtiin käsin ja tallennettiin prefabiksi, joka sitten tallennettiin objektilistaan, eli sitä ei generoitu suoraan rykelmäksi. Pelin lopulliset energiat tehtiin myös samalla tavalla, eli niistä tehtiin erilaisia ryhmiä erilaisissa muodoissa, mitkä sitten tallennettiin prefabeiksi ja lisättiin listaan.

KUVIO 6. Kuvakaappaus pelistä tuotannon alkuvaiheessa

4.1 Huoneiden generointi

Kuvio 7 on esimerkki yhdestä huoneesta. Tässä huoneessa tausta koostuu kolmesta palasta, eli yhdestä keskiosasta ja kahdesta sumupalasta, jotka sulautuvat saumattomasti keskiosaan. Yläreunassa ja alareunassa maantasolla on vihreällä rajatut näkymättömät laatikot, jotka merkkaavat törmäysalueita, joista pelihahmo ei pääse läpi. Alareunan törmäysalue, eli tässä projektissa lattia, toimii myös apuna huoneen leveyttä mitattaessa, sillä sen on koko huoneen levyinen yhtenäinen osa. Huoneille myös lisättiin tagi, eli Unityn sisäinen tunniste, jota käytetään myöhemmin generointiskriptissä tunnistamiseen.



KUVIO 7. Esimerkki huoneesta

Huoneen generointiskripti tunnistaa, kun huoneen reuna on lähellä laskemalla huoneen leveyden lattian avulla ja vertaamalla sitä pelaajan sijaintiin. Reunan lähestyessä peli valitsee huonelistasta satunnaisen huoneen ja liittää sen senhetkisen huoneen perään. Vastaavasti edellinen huone tuhotaan, kun pelaaja on tarpeeksi kaukana siitä.

Kuvioiden 8 ja 9 sisältämissä koodikatkelmissa tarkistetaan ensin, onko uudelle huoneelle tarvetta. Funktiota kutsutaan alirutiinin avulla 0,25 sekunnin välein. Alirutiinit ovat Unityn metodeja, jotka pystyvät jakamaan tehtävänsä useamman kuvaruudun päivityksen ajaksi, kun muut funktiot tekevät tehtävänsä yhden kuvaruudun päivityksen aikana (Unity Technologies 2022a). Ensimmäinen kuvio käsittelee muuttujien määrittelyä, ja siinä kirjataan ylös muun muassa pelaajan sijainti sekä kauimmaisen huoneen koordinaatti. Lisäksi huoneen lisäyspiste ja poistopiste määritellään suhteessa pelaajan sijaintiin ja näytön leveyteen. AddRoomX-muuttuja siis merkkaa pistettä, jonka kohdalla huone täytyy luoda, jos sellaista ei siinä ole.

```
private void GenerateRoomIfRequired()
{
    //creates list of rooms that need to be removed
    List<GameObject> roomsToRemove = new List<GameObject>();
    //flag if more rooms need to be added
    bool addRooms = true;
    //saves player position
    float playerX = transform.position.x;
    //if no room exists after this point then room needs to be added
    float addRoomX = playerX + screenWidthInPoints;
    //stores the point where level currently ends
    float farthestRoomEndX = 0;
    //point after which room should be removed
    float removeRoomX = playerX - screenWidthInPoints;
```

KUVIO 8. Koodikatkelma huoneen generointiskriptistä.

Seuraavassa koodikatkelmassa funktio käy läpi jokaisen huoneen huonelistassa. Huoneen pituus mitataan lattian avulla, jonka jälkeen lasketaan huoneen alku- ja päättymispiste, eli roomStartX ja roomEndX. Sen jälkeen tarkistetaan, onko huoneen alkupiste suurempi kuin piste, jonka kohdalla uusi huone täytyy generoida. Samalla tavalla tarkistetaan, onko huoneen päättymispiste pienempi kuin se, jonka jälkeen huone täytyy poistaa. Myös viimeisimmän huoneen päättymispiste tallennetaan, sillä sitä käytetään myöhemmin uuden huoneen lisäysfunktiossa merkitsemään paikkaa, johon uusi huone sijoitetaan. Lopuksi suoritetaan vielä vanhojen huoneiden poisto ja kutsutaan addRoom-funktiota, jos uusi huone on tarpeellinen.

```
foreach (var room in currentRooms)
{
  //enumerates currentRooms
  float roomWidth = room.transform.Find("floor").localScale.x;
  float roomStartX = room.transform.position.x - (roomWidth * 0.5f);
  float roomEndX = roomStartX + roomWidth;
  //if there is a room after addRoomX point then room doesn't need to be added
  if (roomStartX > addRoomX)
  {
    addRooms = false;
  }
  //if the room ends to the left of removeRoomX point then it needs to be removed
  if (roomEndX < removeRoomX)
  {
    roomsToRemove.Add(room);
  }
  //finds rightmost point of the level
  farthestRoomEndX = Mathf.Max(farthestRoomEndX, roomEndX);
}
//removes rooms marked for removal
foreach (var room in roomsToRemove)
{
  currentRooms.Remove(room);
  Destroy(room);
}
//new room needs to be added
if (addRooms)
{
  AddRoom(farthestRoomEndX);
}
```

KUVIO 9. Koodikatkelma huoneen generointiskriptistä.

Kuvioiden 10 ja 11 koodikatkelmat kuvaavat huoneen lisäysosuutta koodissa. Ensin kirjataan ylös, mikä edellinen huone oli, jonka jälkeen huonelista suodatetaan tunnisteen perusteella niin, että vain edellisestä huoneesta eroavalla tunnisteella merkityt huoneet säilyvät. Tämä lisättiin alkuperäiseen tutoriaalikoodiin, jotta vältyttäisiin saman taustan toistumiselta. Esimerkiksi jos edellinen huone oli metsätaustainen, koodi suodattaa mahdollisista huoneista pois kaikki metsäksi merkityt huoneet.

```
void AddRoom(float farthestRoomEndX)
{
  GameObject lastRoom = currentRooms[currentRooms.Count - 1];
  GameObject[] filteredRooms = availableRooms.Where(room => !lastRoom.CompareTag(room.tag)).ToArray();
```

KUVIO 10. Koodikatkelma huoneen lisäyksestä.

Sen jälkeen koodi valitsee suodatetusta listasta satunnaisen huoneen ja laskee sekä sen leveyden että keskipisteen käyttäen lattiaksi nimettyä törmäysaluetta. Niiden avulla huone sijoitetaan oikealle paikalleen. Lopuksi huone vielä lisätään tämänhetkisten huoneiden listaan.

```
//picks random index of the room prefab
int randomRoomIndex = Random.Range(0, filteredRooms.Length);
//creates a room object using the index chosen
GameObject room = Instantiate(filteredRooms[randomRoomIndex]);
//get the size of the floor which = width
float roomWidth = room.transform.Find("floor").localScale.x;
//calculates where center should be
float roomCenter = farthestRoomEndX + roomWidth * 0.5f;
//sets position of the room
room.transform.position = new Vector3(roomCenter, 0, 0);
//adds room to the list of current rooms
currentRooms.Add(room);
```

KUVIO 11. Koodikatkelma huoneen lisäyksestä.

4.2 Objektien generointi

Objekteihin sisältyvät sekä kerättävä maaginen energia että kaikki esteet, ja niitä generoidaan saman skriptin avulla. Huoneiden tapaan myös objekteista tehtiin valmiit prefabit ja ne tallennettiin objektilistaan. Generointiskripti luo uuden objektin, kun viimeisin objekti on tulossa näkyviin ja sijoittaa sen satunnaisen välimatkan päähän edellisestä. Mahdollista välimatkaa on tietysti rajoitettu, ettei koodi sijoittaisi sitä liian kauas ja aiheuttaisi tilannetta, jossa pelaajalla ei olisi mitään tekemistä. Tarkoitus on luoda variaatiota, etteivät objektit ilmestyisi tasaisin väliajoin, koska se myös kävisi pelaajalle tylsäksi ja ennalta arvattavaksi. Myös tässä osiossa käytettiin Placzekin tutoriaalia, mutta tällä kertaa se vaati hieman muutoksia. Tutoriaalin generointiskripti loi uusia objekteja satunnaisesti ilman mitään rajoituksia. Tämä johti siihen, että peli saattoi generoida esimerkiksi liian monta esteobjektia peräkkäin, mikä voi saada pelaajan kyllästymään tai ärsyyntymään. Sen takia koodiin lisättiinkin jonkin verran rajoituksia, joiden tarkoitus on pitää generoidut objektit vaihtuvina ja yllättävinä sekä vähentää liiallista toistuvuutta.

Kuvio 12 kuvaa esimerkkejä käytetyistä objekteista. Ensimmäinen objekti on energiarykelmä, joista jokainen on erikseen kerättävä. Kaksi muuta ovat esteobjekteja. Lintu lentää noitaa kohti, mutta punainen pallo pysyy paikallaan. Pelaajan törmätessä niihin peli päättyy.

KUVIO 12. *Esimerkkejä objekteista*

Peliin sisältyy pelihahmon lisäksi kaksi objektityyppiä: energiaa ja este- eli vihollisobjekteja. Generointia rajattiin niin, että vain maksimissaan kolme saman tyypin objektia voi ilmestyä kentälle peräkkäin. Tämä tarkoittaa, että esimerkiksi kolmen energiarykelmän jälkeinen neljäs objekti on aina vihollistyyppiä. Toiseksi rajoitukseksi lisättiin se, ettei sama vihollisobjekti voi ilmestyä kahta kertaa peräkkäin. Esimerkiksi kristalliesteen jälkeen toinen samanlainen kristalli ei voi ilmestyä kentälle, mutta paholaislintu voi.

Kuten huoneiden generoinnissa, myös objektien generointi tunnistaa ensin, onko uudelle objektille tarvetta. Tunnistusfunktio toimii samalla tavalla kuin huoneenkin, joten sitä ei ole tarvetta toistaa, mutta objektin lisäysfunktiossa on eroavaisuuksia aiemmin mainittujen rajoitusten takia.

Kuvioiden 13–15 sisältämät koodikatkelmat kuvaavat lisäysfunktiota. Se ottaa vastaan tiedon viimeisimmän objektin sijainnista, jota käytetään myöhemmin uuden sijainnin määrittelyyn. Ensiksi luodaan kopio objektilistasta, jottei sieltä vahingossa poisteta mitään pysyvästi. Sen jälkeen rivillä 108 tehdään ensimmäinen tarkistus. SameTagSpawnCount on muuttuja, johon on tallennettu, montako kertaa saman tyypin objekti on generoitu kentälle, ja maxSameSpawned on suurin sallittu määrä, joka on määritelty aiemmin kolmeksi. Tarkistuksen sisällä käytettävä lastSpawnedTag pitää muistissaan viimeksi luodun objektin tunnisteen. Tarkistuksen mennessä läpi objektilistasta suodatetaan pois ne, joilla on sama tunniste kuin viimeksi luodulla objektilla, ja sen jälkeen riveillä 113–114 listalta valitaan satunnaisesti seuraava objekti.

```
104 void AddObject(float lastObjectX)
105 {
106     List<GameObject> filteredObjects = new List<GameObject>(availableObjects);
107     // Compares the amount of object types spawned so far to the maximum of same type allowed
108     if (sameTagSpawnCount == maxSameSpawned)
109     {
110         filteredObjects = filteredObjects.Where(go => !go.CompareTag(lastSpawnedTag)).ToList();
111     }
112     // generates random index to select a random object from array
113     int randomIndex = Random.Range(0, filteredObjects.Count);
114     GameObject chosenObject = filteredObjects[randomIndex];
```

KUVIO 13. Koodikatkelma objektin generoinnista

Seuraavaksi kuvion 14 koodikatkelmassa rivillä 115 tarkistetaan, onko valittu objekti vihollinen sekä onko se sama kuin edellinen. Kaikki tällä hetkellä pelikentällä olevat objektit löytyvät objects-listasta, ja tarkistus tapahtuu nimen perusteella. Jos uusi objekti on sama, se poistetaan filteredObjects-listalta ja valitaan uusi satunnainen objekti.

```
115     // If the next object is an enemy, compares it to the previous object and rerolls if they're the same
116     if (objects.Count > 0 && chosenObject.CompareTag("Enemy") && chosenObject.name == objects.Last().name)
117     {
118         filteredObjects.RemoveAt(randomIndex);
119         randomIndex = Random.Range(0, filteredObjects.Count);
120     }
121     // creates instance of the randomly selected object
122     GameObject obj = Instantiate(filteredObjects[randomIndex]);
123     obj.name = chosenObject.name;
```

KUVIO 14. Koodikatkelma objektin generoinnista

Seuraavaksi kuvion 15 koodikatkelmassa määritellään uuden objektin sijainti. X- ja Y -koordinaattien minimi- ja maksimiarvot on alustettu etukäteen, ja niiden väliltä valitaan satunnainen sijaintipaikka, johon uusi objekti generoidaan. Sen jälkeen se lisätään objects-listaan ja tarkistetaan, oliko sen tunniste sama kuin edellisen. SameTagSpawnCount-muuttuja nollataan, mikäli tunnisteet eivät olleet samoja.

```
124     // sets the position
125     float objectPositionX = lastObjectX + Random.Range(objectsMinDistance, objectsMaxDistance);
126     float randomY = Random.Range(objectsMinY, objectsMaxY);
127     obj.transform.position = new Vector3(objectPositionX, randomY, 0);
128     // adds the created object to list for tracking and removal
129     objects.Add(obj);
130     if (!string.IsNullOrEmpty(lastSpawnedTag) && !obj.CompareTag(lastSpawnedTag))
131     {
132         sameTagSpawnCount = 0;
133     }
134     sameTagSpawnCount++;
135     lastSpawnedTag = obj.tag;
```

KUVIO 15. Koodikatkelma objektin generoinnista

5 POHDINTA

Projektissa saatiin aikaiseksi suhteellisen yksinkertainen, mutta toimiva prototyyppi pelistä. Sen sisältämä proseduraalinen generointi on alkeellinen, mutta tarkoitukseen sopiva. Tavoitteena olikin vain oppia aiheesta opinnäytetyötä varten sekä tietysti kokeilla, miten proseduraalinen generointi ylipäätään toimii, ja näihin tavoitteisiin päästiin. Toteutus sujui yllättävänkin helposti Placzekin tutoriaalin avulla, ja sen pohjalta oli myös helppo tehdä muutoksia. Missään vaiheessa ei tullut mitään suurempia ongelmia, mutta kaksi ohjelmoijaa olisi ollut projektille hyödyksi, sillä nyt kaikki tehtävät grafiikoita lukuun ottamatta kasaantuivat yhdelle henkilölle. Se vaikeutti tekemistä ja viivästytti aikatauluja. Esimerkiksi loitsumekaniikkaa ei yksinkertaisesti ehditty tehdä aikataulun puitteissa.

Pelissä ja generoinnissa on kuitenkin todella paljon potentiaalia jatkokehitystä varten. Pelimaailman taustoja olisi mahdollista generoida niin, että ne koostuisivat pienemmistä palasista, jotka järjestäytyisivät satunnaisesti yhdeksi taustaksi aina kun uusi tausta generoituu. Näin esimerkiksi metsätaustasta voitaisiin saada lukuisia eri versioita. Nyt taustat koostuvat lähinnä yhdestä tai kahdesta isosta palasta sekä alku- ja loppupaloista.

Taustagenerointia voisi laajentaa myös niin, että seuraava tausta tiedettäisiin jo edellisen taustan generoinnin aikana, jolloin viimeinen pala olisi yhteensopiva siirtymäpala. Tämä tosin vaatisi jo graafikoiltakin enemmän työtä, sillä kaikille mahdollisille taustoille pitäisi olla sopivat siirtymäpalat. Lopputulos olisi kuitenkin luonnollisemman ja sulavamman näköinen kuin pelissä käytetty taikasumu.

Objektien generointia voisi muokata ainakin niin, että tietynlaisia objekteja ilmestyy vain silloin, kun jokin tietty tausta on aktiivisena. Esimerkiksi metsätaustan aikana peli generoisi vain metsään sopivia objekteja tai metsäisiä versioita jo olemassa olevista objekteista. Graafikon toki täytyisi myös tehdä eri taustojen mukaisia variantteja.

Peliä oli suunniteltu laajemmaksi kuin mitä prototyyppiin toteutettiin. Tuhottavat viholliset ovat yksi ominaisuus, joka olisi mahdollista toteuttaa. Noidan kissa pääsisi taistelemaan vihollisten kanssa, ja objektigenerointi keskeytyisi siksi aikaa, kunnes vihollinen on tuhottu. Myös viholliset generoitaisiin peliin niin, että niiden välillä kuluisi aina tietty aika ja että ne vaikeutuisivat mitä kauemmin kentällä on selviydytty.

Joka tapauksessa jatkokehitysmahdollisuuksia on paljon, ja peli onkin suunniteltu jo melkein läpikotaisin. Proseduraalisen generoinnin pohja on jo olemassa, ja sen päälle pystyy kyllä rakentamaan myös uusia ominaisuuksia.

LÄHTEET

Arnia Software 2020. What Makes Unity So Popular in Game Development? Hakupäivä 7.5.2022. https://www.arnia.com/what-makes-unity-so-popular-in-game-development/.

Dealessandri, Marie 2020. What is the best game engine: is Unity right for you? Hakupäivä 7.5.2022. https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you.

Johnson, Mark R 2017. Integrating Procedural and Handmade Level Design. Hakupäivä 5.6.2022. https://www.academia.edu/31504413/Integrating_Procedural_and_Handmade_Level_Design.

Manley, Scott 2013. Rogue - The Original Roguelike. Hakupäivä 22.11.2022. https://www.youtube.com/watch?v=vxF1osPkplA.

Manninen, Tony 2007. Pelisuunnittelijan käsikirja – Ideasta eteenpäin. Tallinna: Kustannus Oy Rajalla.

Muscat, Luke 2012. Depth in Simplicity: The Making of Jetpack Joyride. GDCVault. Hakupäivä 27.11.2022. https://www.gdcvault.com/play/1015527/Depth-in-Simplicity-The-Making.

Naone, Erica 2008. Creating Creatures. MIT Technology Review. Hakupäivä 21.11.2022. https://www.technologyreview.com/2008/06/17/127401/creating-creatures/.

Placzek, Mark 2018. How to Make a Game Like Jetpack Joyride in Unity 2D. Hakupäivä 6.9.2022. https://www.raywenderlich.com/5458-how-to-make-a-game-like-jetpack-joyride-in-unity-2d-part-1.

Rogers, Scott 2014. Level Up! The Guide to Great Video Game Design. Chichester: Wiley.

Shaker, Noor, Togelius, Julian & Nelson, Mark J 2016. Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Cham: Springer.

Sinicki, Adam 2021. What is Unity? Everything you need to know. Hakupäivä 7.5.2022. https://www.androidauthority.com/what-is-unity-1131558/.

Togelius, Julian, Champandard, Alex J, Lanzi, Pier Luca, Mateas, Michael, Paiva, Ana, Preuss, Mike & Stanley, Kenneth O 2013. Procedural content generation: goals, challenges and actionable steps. Hakupäivä 5.6.2022. https://www.researchgate.net/publication/285878527_Procedural_content_generation_goals_challenges_and_actionable_steps.

Togelius, Julian, Kastbjerg, Emil, Schedl, David & Yannakakis, Georgios N 2011. What is Procedural Content Generation? Mario on the borderline. Hakupäivä 27.2.2022. https://www.researchgate.net/publication/228620622_What_is_Procedural_Content_Generation_Mario_on_the_borderline.

Unity Technologies 2022a. Unity Manual – Coroutines. Hakupäivä 8.11.2022. https://docs.unity3d.com/Manual/Coroutines.html.

Unity Technologies 2022b. Unity Manual – Prefabs. Hakupäivä 20.9.2022. https://docs.unity3d.com/Manual/Prefabs.html.

Wikipedia 2021. Rogue (video game). Hakupäivä 12.3.2022. https://en.wikipedia.org/wiki/Rogue_(video_game).

Wilson, Jonathon 2018. Level Design: Understanding a level. Hakupäivä 27.11.2022. https://www.gamedeveloper.com/design/level-design-understanding-a-level.