PROSEDURAALINEN LUOLASTOGENERAATTORI

Ammattikorkeakoulututkinnon opinnäytetyö

Tietotekniikan koulutusohjelma

Hämeen ammattikorkeakoulu, kevät 2017

Anssi Koskenranta

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

TIIVISTELMÄ

RIIHIMÄKI
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

Tekijä Anssi Koskenranta **Vuosi** 2017

Työn nimi Proseduraalinen luolastogeneraattori

TIIVISTELMÄ

Tässä opinnäytetyössä tutustuttiin proseduraaliseen generoimiseen pelien kenttien luomisen näkökulmasta ja tutkittiin eri pelimoottoreiden ominaisuuksia sekä eroavaisuuksia projektiosuuden toteuttamista varten. Projektiosuudessa kehitettiin ohjelma, joka generoi sattumanvaraisen luolaston määritettyjen sääntöjen puitteissa. Työllä ei suoranaisesti ollut toimeksiantajaa, mutta se tehtiin suunnitteilla ollutta peliä varten.

Ennen projektin alkua olin kehittänyt pelejä Unity-pelimoottorilla ja C#-ohjelmointikielellä muutaman vuoden ajan. Proseduraalisesta sisällöntuottamisesta minulla ei ollut etukäteen kokemusta.

Opinnäytetyön tuloksena saatiin aikaiseksi toimiva prototyyppi proseduraalisesta luolastogeneraattorista, joka luo määritetylle alueelle satunnaisen määrän erikokoisia huoneita ja yhdistää huoneet käytävillä. Käyttöön tulevaa valmista versiota luolastogeneraattorista ei ollut edes tämän opinnäytetyön aikana mahdollista tehdä, koska lopulliset säännöt ja tarkemmat tarpeet generaattorille tulevat selville vasta, kun peli, johon generaattori luotiin, saadaan suunniteltua kokonaan.

Avainsanat Proseduraalinen generointi, peliohjelmointi, Unity

Sivut 38 s. + liitteet 3 s.

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

RIIHIMÄKI
Degree Programme in Information Technology
Software engineering

Author Anssi Koskenranta **Year** 2017

Subject of Bachelor's thesis Procedural dungeon generator

ABSTRACT

This thesis will focus on procedural generation in the context of video game level design and explore features and differences between different game engines. In the project section a program was developed, that generates arbitrary dungeon within defined ruleset. The project did not have an immediate client, but it was done for an upcoming game.

Before the start of the project, I had few years of experience in game development with game engine Unity using C# programming language. However, I did not have any experience in generating content procedurally.

As a result of the thesis was a functional prototype of the procedural dungeon generator, which creates random amount of randomized rooms inside a defined area and connects these rooms with corridors. During this project, it was not possible to create the completed version of this dungeon generator because there was no knowledge of final rulesets or precise demands for the generator until the game will be designed entirely.

Keywords Procedural generation, game programming, Unity

Pages 38 p. + appendices 3 p.

Pseudosatunnaisuus	Antaa satunnaiselta vaikuttavan tuloksen, joka ei kuitenkaan ole aidosti satunnainen. Tietokoneen luoma satunnaisuus on käytännössä aina pseudosatunnainen.
Scene	Näkymä, joka sisältää kaikki käytössä olevat objektit. Erilliset näkymät pelissä kuten päävalikko, yksittäiset kentät ja muut vastaavat voidaan rakentaa erillisiin sceneihin.
Prefab	Valmis malli jostakin objektista. Myöhemmin tämän mallin avulla valmiiksi luotu objekti voidaan kopioida scenelle. Mallia muuttamalla jo kopioidut objektit muuttuvat myös ellei niille ole määritelty yksilöllisiä arvoja.
Transform	Objektin komponentti, joka pitää sisällään sen paikan (position), rotaation (rotation) ja mittakaavan (scale).
Parent / Child	Objektilla voi olla yksi tai useampi child-objekti hierarkiassa sen sisällä, mikä perii parent-objektin transform-komponentin arvot.
Inspector	Ikkuna, josta näkee valitun objektin komponentit ja niiden arvot. Näitä arvoja voi muuttaa inspectorissa suoraan koskematta koodiin.
Sprite	Kaksiulotteinen graafinen objekti.
List	List-tyyppinen muuttuja on lista arvoista. Listaan on helppo ja nopea lisätä sekä poistaa muuttujia.
Array	Array on listaa vastaava muuttuja, jonka läpikäyminen ohjelmassa on nopeampaa kuin listan. Tietojen lisääminen ja poistaminen arraysta vaatii sen luomisen uudelleen, joka on huomattavasti raskaampi prosessi kuin listaan vastaavan muutoksen tekeminen.

SISÄLLYS

1 JOHDANTO

Opinnäytetyössäni käydään läpi, mitä proseduraalinen sisällön generoiminen tarkoittaa pääasiassa pelien näkökulmasta. Sen lisäksi tutkitaan eri pelimoottoreita ja verrataan niiden ominaisuuksia toisiinsa, jotta löydetään tämän opinnäytetyön projektille sopivin pelimoottori. Sen jälkeen paneudutaan valitun pelimoottorin tukemien ohjelmointikielien eroihin ja valitaan myös niistä sopivin projektin toteuttamiseksi.

Projektiosuuden tavoitteena on luoda ohjelma, joka pystyy automaattisesti generoimaan uuden sattumanvaraisen luolaston. Valmista ohjelmaa on tarkoitus käyttää lähitulevaisuudessa alkavassa peliprojektissa, ja sen avulla pelaajalle saadaan jokaista pelikertaa kohden uniikki kenttä tutkittavaksi.

Generoidusta luolastosta tarkoituksenani oli saada ihmisen rakentaman eikä luonnollisesti muodostuneen näköinen. Tästä syystä luon ohjelman niin, että käytävät tulevat menemään melko suoraa eivätkä tee ylimääräisiä mutkia. Tarkoituksena ei kuitenkaan ole tehdä täysin realistisen tyylistä kompleksia, koska se vaikuttaisi kentän pelattavuuteen ja mielenkiintoisuuteen negatiivisesti.

2 PROSEDURAALINEN SISÄLLÖN GENEROINTI

Proseduraalisella sisällön generoimisella tarkoitetaan sisällön luomista ohjelmallisesti käyttämällä hyväksi satunnaisuutta tai pseudo-satunnaisuutta, millä saadaan aikaiseksi lähes rajaton määrä pelattavia alueita (Wikidot 2015). Sisältöä ei luoda kuitenkaan täysin mielivaltaisesti vaan ohjelmalla on tarkat säännöt, joiden mukaan sisältöä generoidaan ja satunnaisuudella on tarkoituksena tuoda mukaan vaihtelevuutta (Doull 2008).

2.1 Proseduraalinen generointi peleissä

Pelien maailmassa käytännössä minkä tahansa sisällön luominen proseduraalisesti on mahdollista. Esimerkiksi roguelike-tyyppisessä pelissä Ultima Ratio Regum maaston ja luolastojen lisäksi myös kulttuurit, tavat, sosiaaliset tavat, rituaalit, uskonnot, konseptit ja myytit on generoitu dynaamisesti (Johnson 2016).

Hyvänä esimerkkinä proseduraalisen generoinnin ja manuaalisesti tehtyjen yksityiskohtien yhdistelmästä pidän Diablo III -peliä. Diablossa luolaston kaikki osat yksityiskohtineen on manuaalisesti käsin luotuja, mutta näistä koostuva luolasto luodaan algoritmilla. Mukaan tuodaan myös valmiiksi suunniteltuja huoneita, joissa voi sijaita jokin pelillinen tai tarinaa kuljettava tapahtuma tai tehtävä. Näin saadaan aikaiseksi luolasto, joka on jokaisella pelikerralla uniikki mutta kuitenkin sisältää paljon yksityiskohtia ja sisältöä, joita pelin tekijät ovat sinne halunneet sisällyttää.

3 PELIMOOTTORIN VALINTA

Pelimoottori on videopelien luomiseen ja kehittämiseen tarkoitettu ohjelmistokehys (software framework). Sen perimmäisenä tehtävänä on tyypillisesti antaa muun muassa seuraavat pelien tarvitsemat perustoiminnallisuudet:

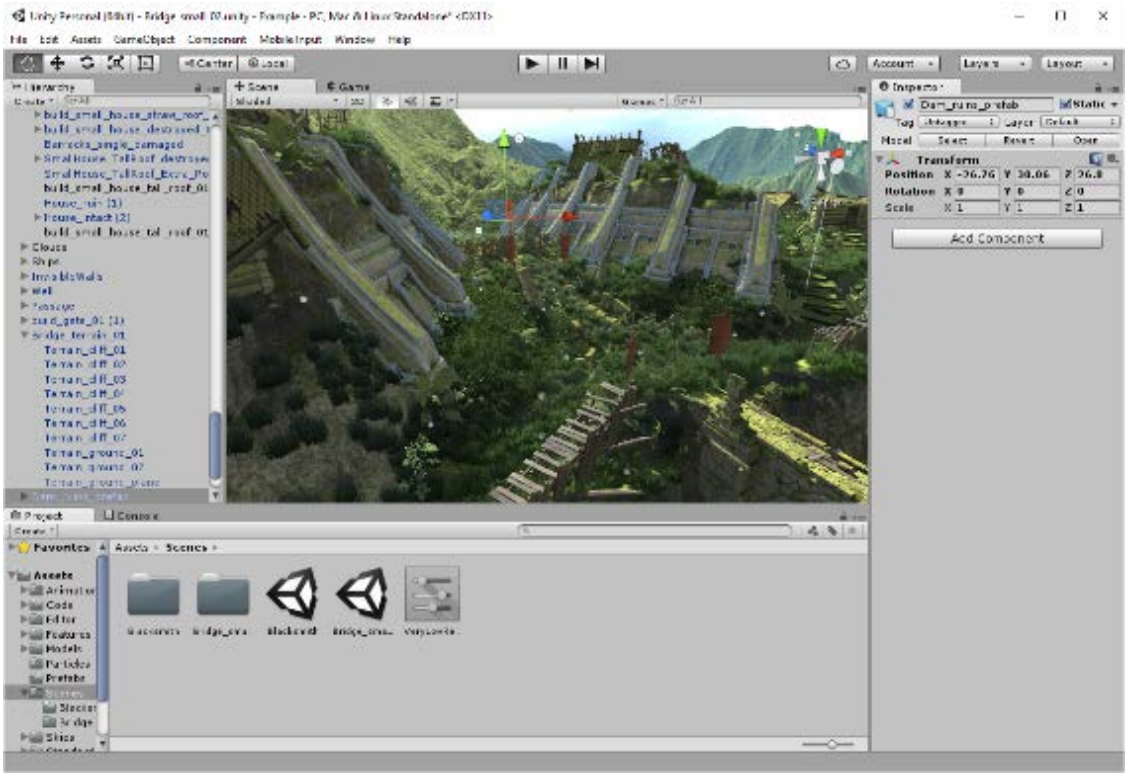
- grafiikan renderöinti
- fysiikkamoottori
- äänet
- komentosarjat (scripting)
- animaatiot
- tekoäly
- verkko-ominaisuudet
- muistinhallinta. (Enger 2013.)

Näiden lisäksi useat modernit pelimoottorit tarjoavat kehittäjälle visuaaliset kehitystyökalut, jotka mahdollistavat nopeamman kehityksen.

Hyviä vaihtoehtoja pelimoottoriksi tässä opinnäytetyöprojektissa oli neljä: Unity, Unreal Engine, CryEngine sekä kiinnostavana uutuutena helmikuussa 2016 julkaistu Lumberyard. Yksi mahdollisesti hyvä vaihtoehto olisi myös voinut olla Source 2 -pelimoottori, mutta projektin aikana sitä ei oltu vielä julkaistu kehittäjien käyttöön.

3.1 Unity

Unity on Unity Technologiesin kehittämä pelimoottori ja kehitysympäristö (Kuva 1), joka on alun perin julkaistu vuonna 2005 (Brodkin 2013). Tuettuina ohjelmointikielinä on C#, JavaScript (UnityScript) ja teknisesti vielä Boo, vaikka sen tuki on virallisesti jo lopetettu (Unity Technologies 2014).



Kuva 1. Unityn käyttöliittymä

Jo vuodesta 2009 asti on Unitysta ollut tarjolla ilmainen versio, joka nykyään tunnetaan nimellä Personal Edition, jota saa käyttää, kunnes tuottoa syntyy yli 100 000 dollaria vuodessa (Unity Technologies 2009). Sen ylittyessä tai tarvittaessa lisäominaisuuksia on tarjolla Professional Edition, josta voi maksaa 75 dollaria kuukaudessa tai 1 500 dollarin kertamaksun (Unity Technologies n.d. a). Hinnoittelu oli pitkään yksi iso syy monelle pienelle tai aloittelevalle kehittäjälle käyttää Unitya, mutta jälkeenpäin ainakin Unreal Engine ja CryEngine ovat seuranneet Unityn mallia myös tarjoten pelimoottorinsa ilmaiseksi pelinkehittäjien käyttöön.

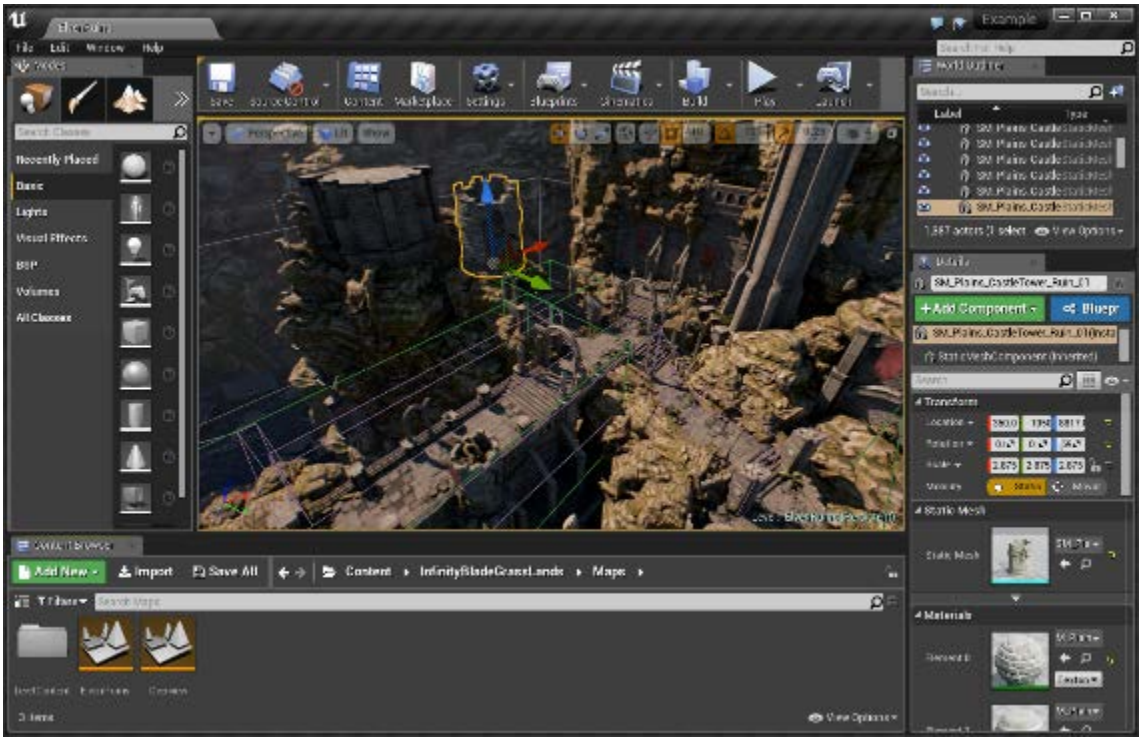
Kuva 2. Maailmanlaajuinen pelimoottorien markkinaosuus (Unity Technologies 2016)

Unity on tällä hetkellä maailmanlaajuisesti suosituin pelimoottori, kun tarkastellaan Unitya käyttävien kehittäjien tai sillä kehitettyjen pelien lukumäärää (Unity Technologies n.d. b) (Kuva 2). Yksi syy suosioon on laaja tuki 25:lle eri alustalle, joiden joukosta löytyy esimerkiksi Windows, iOS,

Android, WebGL, PlayStation 4, Xbox One sekä Wii U (Unity Technologies n.d. c). Unitya on myös yleisesti pidetty aloittelijaystävällisempänä käyttöliittymänsä ja ohjelmointikieliensä puolesta kuin ”raskaamman sarjan” kilpailijoita kuten Unreal Enginea, joka tosin on viimeisten päivitysten myötä kirinyt eroa umpeen uudistetun käyttöliittymän johdosta (Pluralsight 2015).

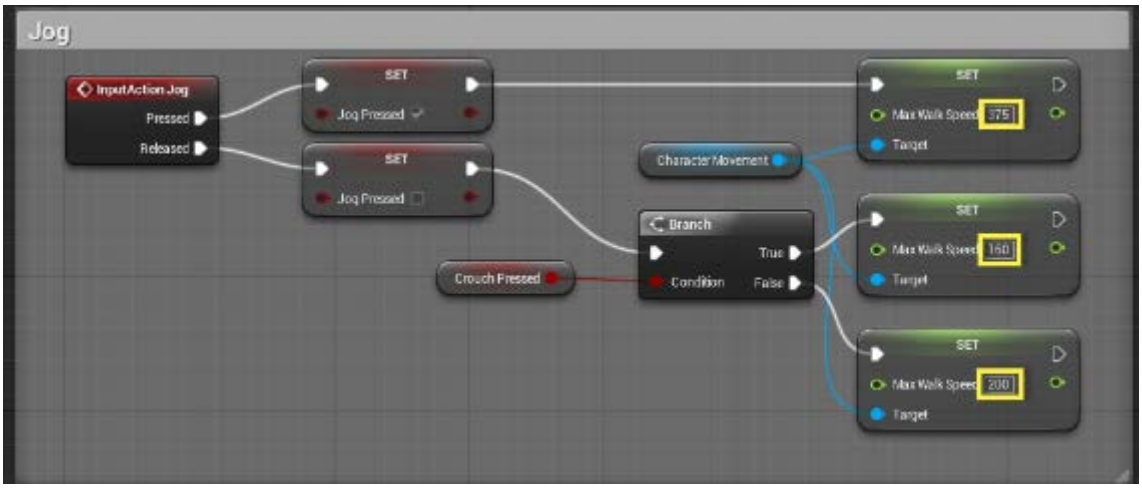
3.2 Unreal Engine

Unreal Engine on Epic Gamesin vuonna 1998 kehittämä pelimoottori (Kuva 3), joka on alun perin suunniteltu ensimmäisen persoonan räiskintäpelille Unreal (first-person shooter) (Thomsen 2010). Pelien ohjelmointi tapahtuu C++-ohjelmointikielellä ja visuaalisella Blueprint Visual Script -järjestelmällä (Epic Games, Inc. n.d.b) (Kuva 4).



Kuva 3. Unreal Enginen käyttöliittymä

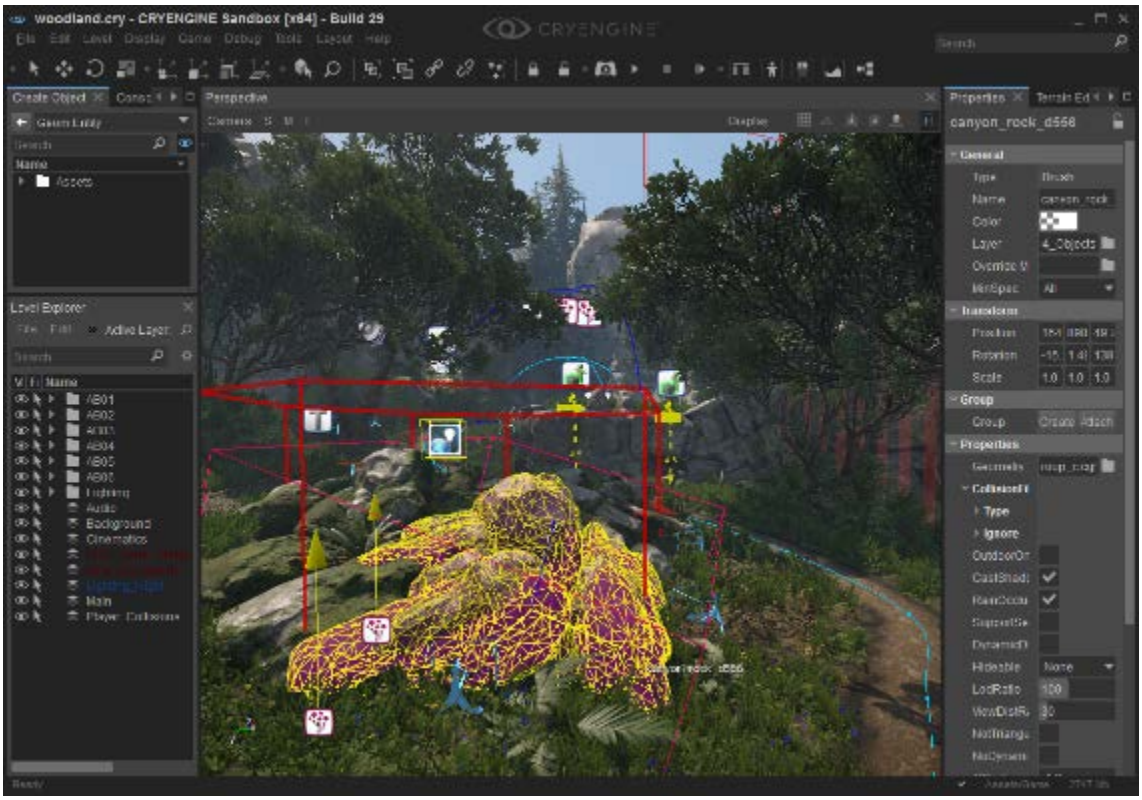
Vuonna 2014 Unreal Enginen käyttäminen mahdollistettiin myös pienemmän budjetin projekteihin 19 dollarin kuukausimaksulla. Sittemmin vuonna 2015 Epic Gamesin perustaja Tim Sweeney julkisti Unreal Enginen siirtyvän kaikille ilmaiseen muotoon. (Sweeney 2015.) Mikäli tällä pelimoottorilla tehty peli tai sovellus tuottaa bruttona yli 3 000 dollaria vuosineljänneksellä, täytyy tuotosta maksaa viisi prosenttia Epicille (Epic Games, Inc. 2016 a).



Kuva 4. Esimerkki Unreal Enginen visuaalisesta ohjelmoinnista (Epic Games, Inc. 2016)

3.3 CryEngine

CryEngine on Crytekin luoma pelimoottori, jota kehitettiin sen ensimmäisen pelin Far Cryn rinnalla ja joka julkaistiin yhdessä pelin kanssa vuonna 2004. CryEngine sisälsi niin sanotun hiekkalaatikkoeditorin, jolla oli mahdollista luoda pelimaailmaa reaaliajassa, mikä ainakin silloin oli mullistava ominaisuus. (Crytek 2016a) Ohjelmointikielinä on käytettävissä C++, Lua ja C# (Crytek 2016b).

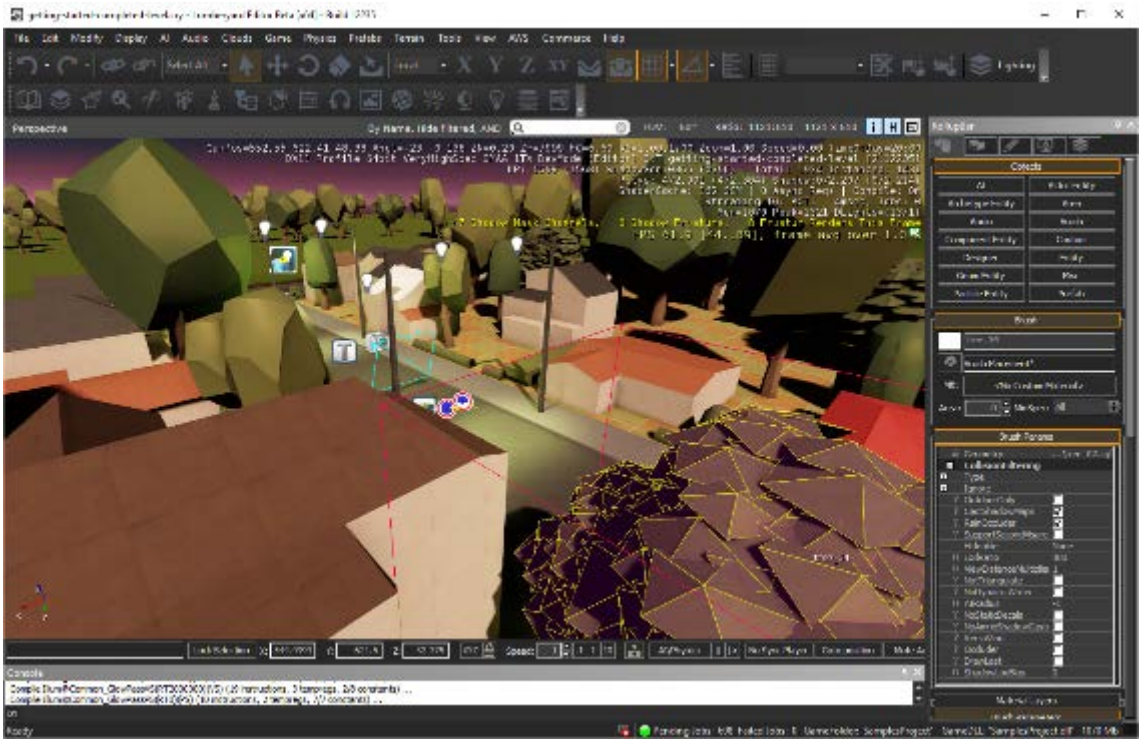


Kuva 5. CryEnginen käyttöliittymä

Alkuvuodesta 2016 Crytek julkisti CryEnginen muuttuvan ”maksa mitä haluat” -malliin (pay what you want), jolloin kehittäjän on mahdollista saada pelimoottori käyttöönsä myös täysin ilmaiseksi. Toisin kuin Unreal Engine tai Unity, CryEngine on täysin rojaltivapaa. (Crytek 2016c.) Tämän version lisäksi on myös olemassa EaaS (Engine-as-a-Service), joka maksaa 9,90 dollaria kuukaudessa, tai räätälöity yrityslisenssi vaativampiin käyttötarpeisiin (Crytek 2016d; Crytek 2016e).

3.4 Lumberyard

Vuoden 2015 aikana Amazon osti lisenssin Crytekin pelimoottoriin CryEngineen. Helmikuussa 2016 Amazon julkaisi siihen perustuvan uuden pelimoottorin, joka sai nimekseen Lumberyard (Schreler 2015; Nutt 2016) (Kuva 6). Ohjelmointi tapahtuu C++- ja Lua-kielillä ja pelejä on mahdollista kehittää Microsoft Windows -, Xbox One -, PlayStation 4-, sekä rajoitetusti iOS- ja Android-alustoille. Lumberyard on vielä beta-testivaiheessa (Amazon.com, Inc. n.d.).



Kuva 6. Lumberyardin käyttöliittymä

Lumberyard on heti julkaisustaan lähtien ollut täysin ilmaiseksi käytettävissä eikä siitä ole edes olemassa maksullista versiota. Rajoituksena on se, että mitään muuta Amazonin AWS-palveluun verrattavissa olevaa verkkopalvelua ei saa Lumberyardin kanssa käyttää. (Amazon.com, Inc. 2016.)

3.5 Pelimoottori projektilleni

Vaihtoehtoja projektille oli ennen projektin aloittamista neljä, CryEngine, Lumberyard, Unity ja Unreal Engine. Näistä vaihtoehdoista kuitenkin karsin Amazonin Lumberyardin pois jo heti alussa, koska se oli aivan uusi ja

vielä beta-vaiheessa. Näin uuden pelimoottorin kanssa olisi todennäköisesti tullut vastaan työtä hidastavia bugeja sekä dokumentaatio ja osaava yhteisö olisivat olleet todella vajavaisia.

Vertailtavaksi jäi kolme varteenotettavaa vaihtoehtoa eli CryEngine, Unity sekä Unreal Engine. Näistä jokainen oli ollut olemassa jo yli 10 vuoden ajan ja jokaiselle löytyi laaja dokumentaatio sekä isot kehittäjäyhteisöt.

CryEnginen vahvuus projektilleni oli C#-ohjelmointikielen tuki, jota itse osaan parhaiten. Henkilökohtaisesti kuitenkin pidin sen käyttöliittymää kolmikon hankalimpana ja eniten epäintuitiivisena.

Myös Unity tukee C#-kieltä ja sen lisäksi JavaScriptiä, jota osasin kohtalaisen hyvin jo valmiiksi. Unityn käyttöliittymä oli mielestäni tästä kolmikosta kaikista helppokäyttöisin ja nopeimmin sisäistettävissä. CryEngineen ja Unreal Engineen verrattuna sitä voidaan pitää pienemmän luokan pelimoottorina, joka ei pysty samaan graafiseen tasoon, mutta sillä ei ollut tämän projektin kannalta mitään merkitystä.

Unreal Engine tukee vain C++-kieltä, joka on työläämpää ja haastavampaa kuin C#. Vaihtoehtona on toki visuaalinen ohjelmointi, mutta se ei kovinkaan hyvin soveltunut käyttötarkoitukseeni. Käyttöliittymältään Unreal Engine on tullut viimeisten vuosien aikana myös helposti sisäistettäväksi.

Tämän pohdinnan jälkeen valinta oli kohtalaisen helppo ottaen myös huomioon, että olen suurimman osan tähänastisista projekteistani tehnyt Unitylla. Molemmat Unityn tukemat ohjelmointikielet sekä itse ohjelma olivat minulle jo valmiiksi tuttuja. Näiden seikkojen ansiosta pääsin heti aloittamaan projektin parissa työskentelemisen ilman uuden ohjelman tai ohjelmointikielen opettelemista.

4 OHJELMOINTIKIELEN VALINTA

Koska päädyin valitsemaan pelimoottoriksi Unityn, oli vaihtoehtoja ohjelmointikieleksi kolme, C#, JavaScript tai Boo. Näistä kuitenkin Boon virallinen tukeminen oli jo päättynyt, joten lähinnä vertailtaviksi jäivät C# ja JavaScript.

4.1 JavaScript

JavaScript on dynaaminen, korkean tason tyypittämätön ohjelmointikieli, jota ei käännetä kuten monet muut ohjelmointikielet vaan JavaScript-moottori tulkitsee sitä suoraan (Flanagan 2011, 1). Tulkittava JavaScript on jonkin verran käännettävää C#-ohjelmointikieltä hitaampi, koska tulkitseminen on yleisesti hitaampaa kuin konekieleen käännetyn ohjelman ajaminen ja kääntämisen aikana koodia voidaan optimoida tehokkaammin. Myös dynaamisuus vaikeuttaa optimointia ja hidastaa suoritusta, mutta nykypäivänä JavaScript-moottoreista on tullut niin tehokkaita, että erot jäävät näistä seikoista huolimatta monessa asiassa todella pieniksi.

Yksinkertaisimmillaan JavaScriptin dynaamisuus ja tyypittämättömyys tarkoittavat sitä, että kun luodaan uusi muuttuja, niin sille ei määritetä tyyppiä ohjelmointivaiheessa. Esimerkiksi luotaessa uusi muuttuja x, se tehdään kirjoittamalla *var x*. Avainsana *var* kertoo sen olevan muuttuja, mutta tyyppiä ei ole määritetty. JavaScript-moottori päättää ajon aikana dynaamisesti muuttujalle oikean tyypin, esimerkiksi *x = 5* määrittäisi muuttujan x tyypiksi integer eli kokonaisluku.

Tyypittämättömyys on usein aloittelevalle ohjelmoijalle helpompi käsitellä, kun muuttujan tyyppiä ei tarvitse miettiä etukäteen. Tosin oman kokemukseni mukaan monimutkaisissa ohjelmissa varsinkin virheiden etsiminen ja selvittäminen taas vaikeutuvat, koska muuttujan tyyppi voi muuttua sellaiseksi, jota ohjelmoija ei odota. Esimerkiksi jos ohjelmoija uskoo muuttujan x olevan kokonaisluku, mutta se onkin muuttunut jossain vaiheessa liukuluvuksi, niin laskutoimituksen tulos voi olla aivan eri kuin olisi pitänyt.

Näistä seikoista huolimatta JavaScript olisi myös ollut erinomainen ohjelmointikieli projektini toteuttamiseen. Varsinkin projektin alkuvaiheilla se olisi mahdollistanut C#-ohjelmointikieltä nopeamman prototyypityksen dynaamisuutensa ansiosta, joka nopeuttaa koodin kirjoittamista ja vähentää ohjelmoijan työtä. Epäilisin tosin, että se olisi tuonut jonkin verran ongelmia projektin loppuvaiheilla, jolloin ohjelma alkoi olla melkoisen monimutkainen ja isoksi osaksi työtä tulee virheiden metsästäminen koodista.

4.2 C#

Toisin kuin JavaScript, C# on vahvasti tyypitetty ohjelmointikieli, jolla tehty ohjelma täytyy kääntää ennen kuin se voidaan ajaa. Siinä jokaisen muuttujan tyyppi määritellään heti luomisvaiheessa. Esimerkiksi kokonaislukutyyppinen muuttuja x luotaisiin kirjoittamalla *int x*, josta ensimmäinen osa *int* määrittää muuttujan tyypiksi integer eli kokonaisluku. Mikäli jossakin ohjelman suoritusvaiheessa tälle muuttujalle x yritetään antaa arvoksi esimerkiksi liukuluku, niin koodin suorittaminen pysähtyy ja se antaa ulos virheilmoituksen.

Tämä vahva tyypitys estää ohjelmoijaa hyvin pitkälle tekemästä koodillisia virheitä vahingossa ja jokaisesta muuttujasta tietää varmasti, minkä tyyppisen arvon se pitää sisällään. Tietynlaiset virheet, jotka tulkittavissa ohjelmissa tulisivat ilmi vasta ajovaiheessa, löytyvät käännettävissä kielissä jo heti kääntämisvaiheessa. Kuten jo luvussa 4.1 todettiin, käännettävät kielet ovat suorituskyvyltään tehokkaampia, mutta niiden kääntäminen hidastaa kehittämistä, kun ohjelma täytyy kääntää aina ennen kuin sitä voidaan testata.

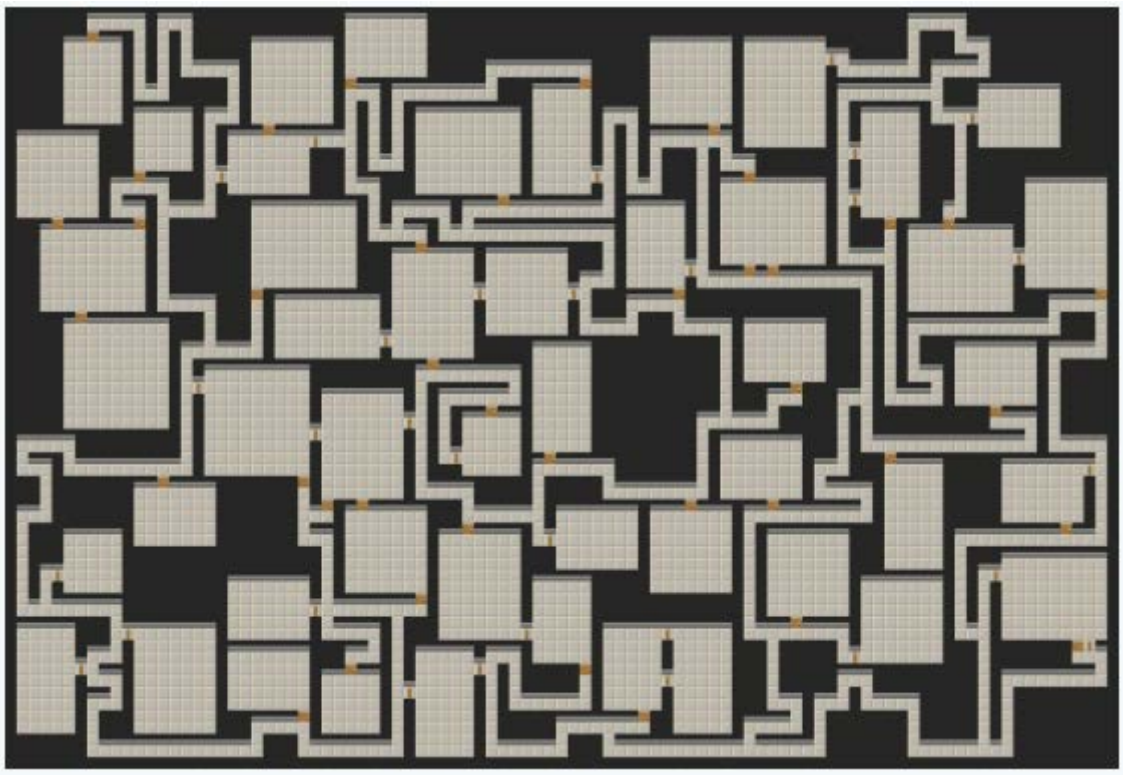
Molemmat näistä seikoista ovat myös osittain mielipidekysymyksiä ja itse pidän enemmän vahvasti tyypitetystä ja ”tiukkasääntöisestä” ohjelmointikielestä, joka suorastaan pakottaa kirjoittamaan parempaa koodia. Näiden syiden lisäksi tunsin C#-ohjelmointikielen paremmin kuin JavaScriptin, joten päätin käyttää sitä projektini toteuttamisessa.

5 PROJEKTI

Käytännön osuutena toteutin projektin, jonka tavoitteena oli saada aikaiseksi toimiva luolastogeneraattori. Lopputulosta on tarkoitus hyödyntää myöhemmin tulevaisuudessa tehtävän pelin kehityksessä, mutta se ei erityisesti vaikuttanut projektin toteutuksessa. Valmiin projektin ei ollut tarkoitus näyttää visuaalisesti juuri miltään sen ollessa vain pohja, jonka päälle pelin kentän elementit tulevat.

Ennen projektin aloittamista olin jo valmiiksi käynyt läpi erilaisia tapoja, miten muut olivat generoineet kenttiä, huoneita ja labyrintteja erilaisiin käyttötarkoituksiin. Yksi läpikäymistäni proseduraalisesti luolaston luovista peleistä oli Angband. Sen lähdekoodi on vapaasti ladattavissa ja koodi on erittäin hyvin kommentoitua, mutta C-kieli ei ollut itselleni tuttu (Angband n.d.).

Erittäin hyödylliseksi ja kiinnostavaksi totesin Bob Nystromin (2014) kirjoittaman blogitekstin, jossa hän käytti labyrintin generoimiseen käytettävää algoritmia käytävien luomiseksi. Mielestäni näin generoidusta luolastosta tuli mielenkiintoisen näköinen, mutta ei aivan sitä, mitä hain, sillä käytävät muodostavat huomattavan määrän ylimääräisiä lenkkejä (Kuva 7). Hyödynsin hänen käyttämää ideaa, jossa käytetään parittomia lukuja huoneita lisättäessä, jolla huoneet saatiin erilleen toisistaan. Sain hänen blogistaan myös idean luoda huoneet käyttämällä tiettyä määrää yrityksiä, jossa ei välitetä ollenkaan jo olemassa olevista huoneista vaan yritettiin aina vain lisätä uusi. Algoritmi tunnistaa suoritettaessa onko tämä uusi huone jo olemassa olevan huoneen päällä, jos on, niin hylätään se.

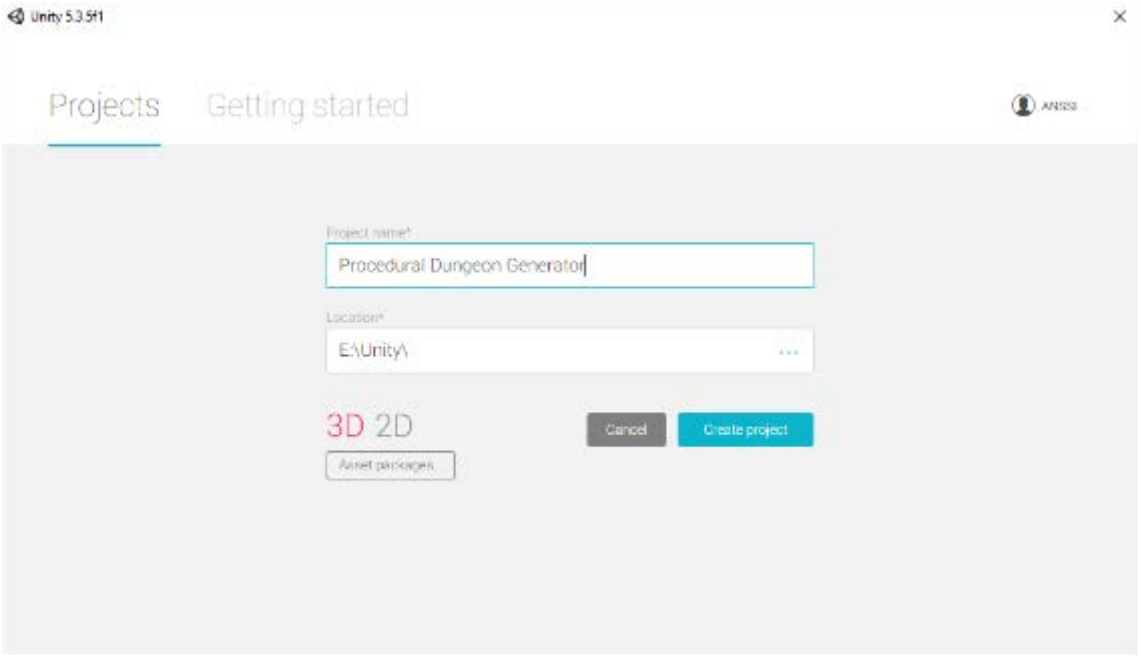


Kuva 7. Nystromin generaattorilla luotu luolasto (Nystrom, B. 2014)

Seuraavissa luvuissa kävin läpi projektin vaiheet yksi osa kerrallaan. Ennen kuin aloitin ohjelmoinnin, piti minun luoda uusi projekti Unityssa. Projektin alustuksen jälkeen aloitin ohjelmoinnin luomalla määritelmät ja muuttujat luolaston alueelle. Vasta näiden vaiheiden jälkeen luvusta 5.3 alkaen, pääsin itse generoimisen kimppuun. Loin ensin algoritmit huoneiden ja niitä yhdistävien käytävien generoimiseksi. Lopuksi lisäsin vielä algoritmit, joista ensimmäinen tarkastaa luolaston yhteneväisyyden ja kaksi muuta toimivat yhdessä generoiden käytävät luolaston erillisten osioiden välille, jos ensimmäinen algoritmi sellaisia löysi.

5.1 Unity-projektin alustus

Projektin aloituksessa ensiksi latasin ja asensin Unityn uusimman version, joka oli versio 5.3.5. Työ lähti liikkeelle luomalla uusi projekti Unityssa (Kuva 8). Annoin projektille nimen, kansion sekä valitsin sen olevan kolmiulotteinen, mikä helpotti jonkin verran tulevan luolaston hahmottamista. Tässä ikkunassa olisin myös voinut lisätä projektiin erilaisia paketteja kuten valmiita hahmoja, kameroita tai efektejä ynnä muita lisäosia. Näitä paketteja on myös mahdollista ladata lisää Unity Asset Storesta tai niitä voi tehdä vaikka itse.



Kuva 8. Uuden projektin aloitusikkuna Unityssa.

Tein uuteen projektiin minimalistisen kansiorakenteen (Kuva 9). En pitänyt kansiorakennetta tämän projektin osalta erityisen tärkeänä, koska tiedostoja ei ollut montaakaan, mutta kansiorakenne helpottaa projektin lopputuloksen hyödyntämistä tulevissa peliprojekteissa.

Kuva 9. Kansiorakenne

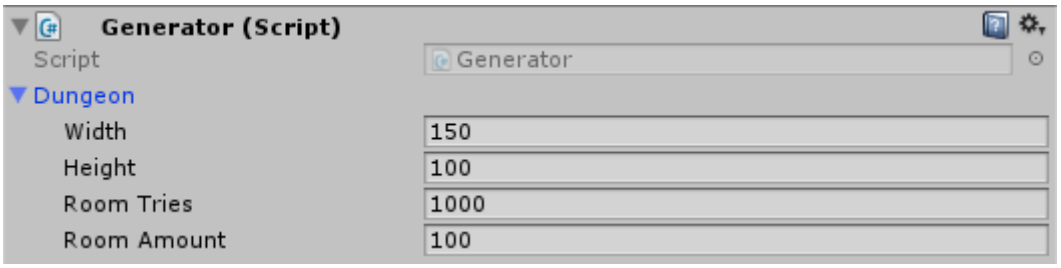
Loin valmiiksi kolme tyhjää objektia *TileFloor*, *TileRoom* ja *TileDoor*. Niiden childiksi lisäsin Sprite-objektin, kaksiulotteisen ruudun, joka tulee toimimaan lattiana. Näistä kolmesta objektista tein prefabit *Prefabs*-kansion sisälle, jotta niitä oli helpompi käyttää myöhemmin maailmaa generoidessa. Poistin ne sceneltä ja muutin vielä pelin ympäristön valon (Ambient Light) valkoiseksi eli täysin kirkkaaksi. Lopuksi tallensin scenen *DungeonGenerator*-kansion sisälle nimellä *ExampleScene*.

5.2 Projektin rajaus ja luolaston määritykset

Valmiin pelikentän oli tarkoitus olla huoneista ja käytävistä koostuva luolasto. Sen täytyi olla kooltaan rajattu, käytävien tuli olla haarautuvia ja silmukoiden muodostumisen piti myös olla mahdollista. Oli heti selvää, että tätä täytyi lähteä ratkaisemaan pienemmissä osissa tekemällä ne osa kerrallaan lisäten ominaisuuksia ja sääntöjä sekä tietenkin säätää jo tehtyä sopivammaksi. Päätin aloittaa projektin ensimmäisen osan ohjelmoimalla yksinkertaisen koodin, jolla voidaan määrittää luolaston laajuus ja sinne generoituvien huoneiden määrä tai tiheys.

```
//Dungeon.cs
[System.Serializable]
public class Dungeon
{
    public int width;
    public int height;      //area of the dungeon
    public int roomTries;   //number of tries when spawning rooms
    public int roomAmount;  //amount of rooms to be spawned
}
```

Loin luolaston asetuksille oman luokan *Dungeon*, joka tässä vaiheessa piti sisällään tiedot sen alueen koosta sekä siitä, kuinka monta kertaa huoneita yritetään luoda tai kuinka monta huonetta generoidussa luolastossa tulee olemaan yhteensä. Näistä kahdesta oli tarkoituksena jättää vain toinen lopulliseen versioon, mutta tässä vaiheessa en vielä tiennyt, kumpaa tulen lopulta käyttämään. Ennen luokan määrittämistä lisäsin attribuutin *[System.Serializable]*, joka mahdollistaa luokan sisäisten muuttujien arvojen näkymisen ja muuttamisen Unityn inspectorissa, joita voi muuttaa tulevissa projekteissa tarpeen mukaan (Kuva 10).



Kuva 10. Luokan Dungeon mukaan tehty muuttuja *dungeon* inspectorissa, jossa voidaan määrittää sille halutut arvot.

```
//Generator.cs
using UnityEngine;
using System.Collections;

public class Generator : MonoBehaviour
{
    public Dungeon dungeon = new Dungeon();
    void Awake()
    {
        //check that the area of the dungeon is odd-sized and if
not, fix it
        if (dungeon.width % 2 == 0 || dungeon.height % 2 == 0)
        {
            print("Area of the dungeon has to be odd-sized. Added
one to width and/or height.");
            if (dungeon.width % 2 == 0)
                dungeon.width++;
            if (dungeon.height % 2 == 0)
                dungeon.height++;
        }
    }
}
```

Tässä vaiheessa tekemäni generator-skripti tuli sisältämään käytännössä koko työn, erillisiä apuluokkia lukuun ottamatta. Loin juuri luodun luokan *Dungeon*-tyyppisen uuden muuttujan *dungeon*, jolle voidaan antaa arvot inspectorissa (Kuva 10). *Awake* on Unityn game loopin ensimmäisenä ajettava komponentti, joka ajetaan heti, kun skriptin instanssi ladataan. Koska ohjelmani tuli tarvitsemaan parittoman kokoisen alueen, tein heti alussa tarkastuksen ja mikäli korkeus tai leveys on jaollinen kahdella, koodi lisää sen arvoon yhden.

5.3 Huoneiden luonti

Aioin ensin tehdä myös huoneille oman luokan, mutta totesin tyypin *rect* pitävän sisällään tarvittavat tiedot eli koordinaatit x ja y sekä koon eli korkeuden ja leveyden.

```
public List<Rect> rooms = new List<Rect>();
public Transform tileRoom;

void Awake()
{
    //...
```

```
    AddRooms();
    SpawnFloor();
}

void AddRooms()
{
    for(var i = 0; i < dungeon.roomAmount; i++)
    {
        //Random odd-sized rooms
        int width = Random.Range(2, 4) * 2 + 1;
        int height = Random.Range(2, 4) * 2 + 1;

        //Random position inside dungeon area
        int x = Random.Range(0, Mathf.FloorToInt((dungeon.width -
width) / 2)) * 2 + 1;
        int y = Random.Range(0, Mathf.FloorToInt((dungeon.height -
height) / 2)) * 2 + 1;

        //Create new rectangle with just randomized variables for
size and position.
        Rect room = new Rect(x, y, width, height);
        //Push the new room into list "rooms"
        rooms.Add(room);
    }
}
```

Lisäsin *Awake*-metodin loppuun kohdat, jotka kutsuvat metodeja *AddRooms* ja *SpawnFloor*, joka tehtiin seuraavaksi. Ensin luomani *AddRooms*-metodi piti sisällään silmukan, jossa luolaston alueelle luodaan huoneet. Algoritmi aloittaa arpomalla huoneille koot käyttämällä satunnaisia lukuja määritellyiltä väleiltä, jotka sitten kerrotaan kahdella ja sen arvoon lisätään vielä yksi, jolloin huoneen koko tulee aina olemaan pariton. Tässä vaiheessa tämä oli vain yksinkertainen käyttämäni tapa toteuttaa haluttu lopputulos. Se oli siinä suhteessa huono systeemi, että kirjoitin arvot suoraan koodiin eikä niitä voi helposti muuttaa myöhemmin esimerkiksi inspectorista.

Huoneen lisäämisen jälkeen algoritmi arpoo huoneille paikat luolaston alueen sisältä. Paikan arpomisessa algoritmi ottaa huomioon myös huoneen koon. Käytin tässä samaa laskutoimitusta kuin huoneen kokoa arvottaessa, millä sain huoneet parittomiin ruutuihin. Lopuksi tein *rect*-tyyppisen muuttujan *room*, jonka määritin sisällään nämä arvotut tiedot, ja se lisättiin alussa luotuun listaan *rooms*.

```
void SpawnFloor()
{
    //Loop through all "room" rectangles in list "rooms"
    foreach (Rect room in rooms)
    {
        //Loop through all tiles that there is inside the room
        for(int x = Mathf.FloorToInt(room.x); x < Mathf.Floor-
ToInt(room.x + room.width); x++)
        {
            for (int y = Mathf.FloorToInt(room.y); y < Mathf.Floor-
ToInt(room.y + room.height); y++)
            {
                //Instantiate new tiles, make "TilesForRooms" their
parent and rename them to make debuggin easier
```

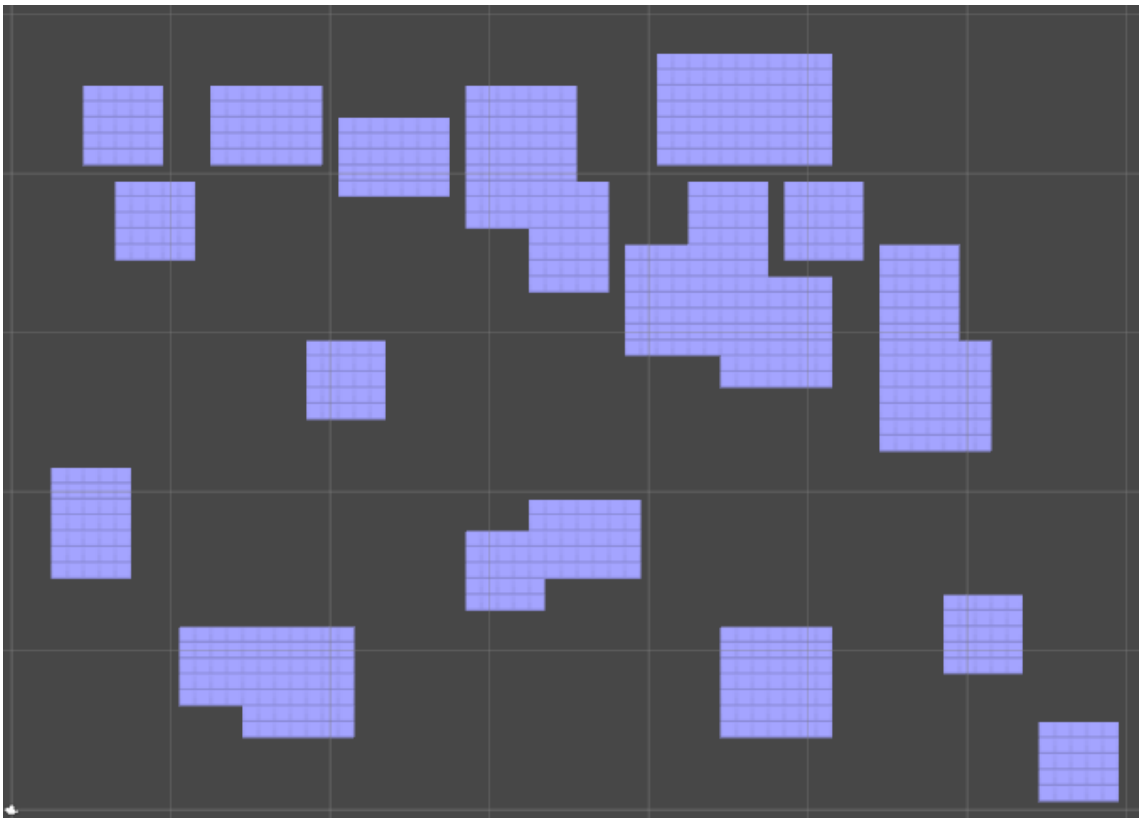
```
                Transform newTile = Instantiate(tileRoom, new Vec-
tor3(x, 0, y), Quaternion.identity) as Transform;
                newTile.parent = GameObject.Find("TilesFor-
Rooms").transform;
                newTile.name = "TileRoom (" + x + ", " + y + ")";
            }
        }
    }
}
```

Seuraavaksi tein *SpawnFloor*-metodin, jonka tarkoituksena oli lisätä scenelle aikaisemmin luotuja ruutuja lattiaksi. Tässä vaiheessa loin algorimin, joka luo huoneille lattian. Ohjelman myöhemmässä vaiheessa tarkoituksenani oli lisätä myös käytävien sekä ovien ruudut tässä metodissa. Luomani algoritmi sisälsi kolme sisäkkäistä for-silmukkaa ja kun algoritmi suoritetaan se käy läpi seuraavat asiat:

- ensimmäinen käy läpi kaikki huoneet listalta *rooms*,
- toinen käy huoneen laattojen x-koordinaatit läpi ja
- kolmas käy huoneen laattojen y-koordinaatit läpi.

Jokaisella kierroksella algoritmi luo uuden ruudun (*newTile*), jonka paikkana on toisen ja kolmannen for-silmukan muuttujat. Lattioiden parentiksi määritin annettavaksi scenelle valmiiksi lisätty objekti *TilesForRooms* ja uudelle laatalle nimeksi prefabin nimi sekä sen koordinaatit. Tässä vaiheessa algoritmi ei vielä tarkasta, onko koordinaatit jo käytössä eli ruutuja voi tulla useampi päällekkäin.

En pitänyt kuvassa 11 näkyvästä lopputuloksesta huoneiden luomisessa, koska huoneet olivat oudon mallisia. Tällä tekniikalla koodin luomat huoneet olisivat voineet myös olla vain yhdessä kasassa, kun koodissa ei ollut huoneen paikalle mitään rajoituksia. Seuraavaksi päätin kokeilla muuttaa koodia niin, että se tarkastaa huoneita luotaessa, ettei niiden alle jää toista huonetta vaan kaikki huoneet generoituvat erilleen.



Kuva 11. Generaattorin luomat huoneet kun niiden paikkaa tai päällekkäisyyttä ei ole rajoitettu.

Saadakseni huoneet erilleen piti generaattorin koodia muuttaa huoneiden lisäämisen osalta niin, että se tarkastaa, onko uusi huone jonkin aikaisemmin luodun huoneen kanssa päällekkäin. Tämän lisäksi myös huoneiden määrän tilalle otin käyttöön yritysten määrän, koska huoneet edelleen lisätään täysin satunnaisesti ja olisi teoriassa mahdollista, että koko koodi jumiutuisi sen yrittäessä loputtomasti lisätä huonetta vanhojen päälle.

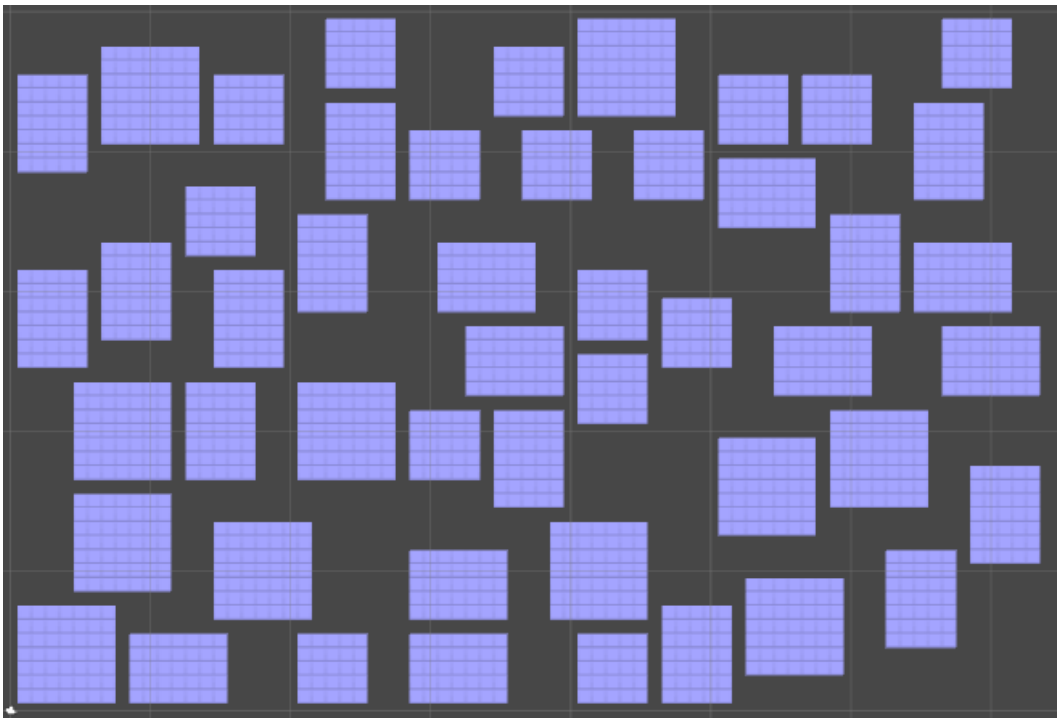
```
void AddRooms()
{
    for(var i = 0; i < dungeon.roomTries; i++)
    {
        //Random odd-sized rooms
        int width = Random.Range(2, 4) * 2 + 1;
        int height = Random.Range(2, 4) * 2 + 1;

        //Random position inside dungeon area
        int x = Random.Range(0, Mathf.FloorToInt((dungeon.width -
width) / 2)) * 2 + 1;
        int y = Random.Range(0, Mathf.FloorToInt((dungeon.height -
height) / 2)) * 2 + 1;

        //Create new rectangle with just randomized variables for
size and position.
        Rect room = new Rect(x, y, width, height);

        //Check if new room overlaps with any other room
        bool overlaps = false;
        foreach(Rect other in rooms)
        {
            if (room.Overlaps(other))
            {
```

```
                overlaps = true;
                break;
            }
        }

        //If the new room didn't overlap with any other, push it
into list "rooms"
        if (!overlaps)
        {
            rooms.Add(room);
        }
    }
}
```

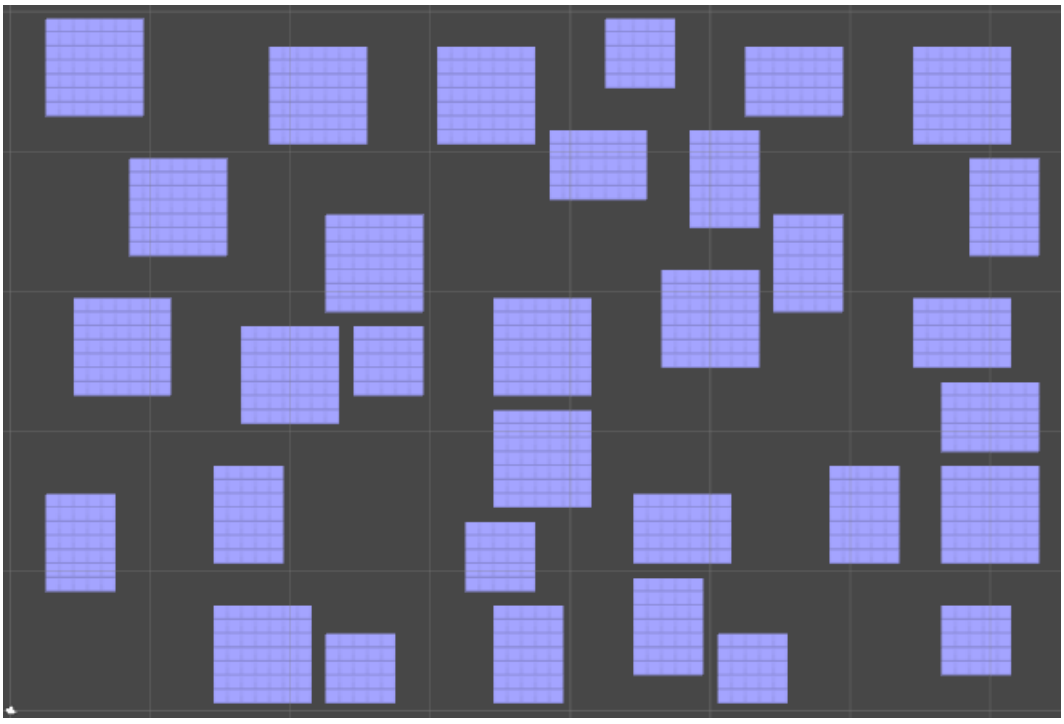
Vaihdoin koodin alussa *dungeon.roomAmount*-muuttujan tilalle muuttujan *dungeon.roomTries*, jolloin sen arvoksi määritelty yritysten määrä toimii for-silmukan ajettavien kierrosten lukumääränä. Huoneiden paikan ja koon arpomiseen sekä itse *room*-muuttujan luomiseen tehdyt koodit jätin ennalleen. Uuden osan aloitin tämän jälkeen kohdassa jossa luotiin boolean-tyyppinen muuttuja *overlaps*, jonka oletusarvoksi määritin *false*.

Ajattaessa algoritmi tekee tarkastuksen käyttämällä foreach-silmukkaa, joka käy läpi kaikki jo aikaisemmin *rooms*-listaan lisätyt huoneet. Itse tarkastuksen algoritmi suorittaa kutsumalla rect-tyypin Overlaps-metodia, jolle annetaan verrattavaksi rect-tyyppinen muuttuja *other*. Mikäli algoritmi löytää päällekkäisyyksiä, muuttaa se muuttujan *overlaps* arvoksi true ja keskeyttää silmukan break-komennolla, koska on turha tarkastaa muita päällekkäisyyksiä, kun jo yksi riittää huoneen hylkäämiselle. Lopuksi määritin uuden huoneen lisättäväksi *rooms*-listaan, mikäli päällekäisyyksiä ei löytynyt.

Tällä koodin uudella versiolla lopputulos oli mielestäni huomattavasti parempi ja järkevämmän näköinen (Kuva 12). 1 000 yrityksellä huoneita tulee vielä turhan tiiviisti, mutta sen muuttaminen on helppoa suoraa inspectorissa ja 100 yrityksellä huoneiden tiheys näytti aika sopivalta (Kuva 13). En ollut vielä täysin tyytyväinen lopputulokseen vaan halusin myös muun mallisia huoneita, mutta tämä sai tässä vaiheessa projektia kelvata. Aioin vielä palata projektin loppuvaiheilla tämän asian pariin.



Kuva 12. Generaattorin luomat huoneet kun päällekkäisyydet on estettynä ja kaikki huoneet ovat erillään toisistaan.



Kuva 13. Kuvassa 12 on generaattorille annettu yritysten määräksi 1 000 ja tässä muuten aivan samalla koodilla luodut huoneet, mutta vain 100 yrityksellä.

5.4 Käytävien luonti

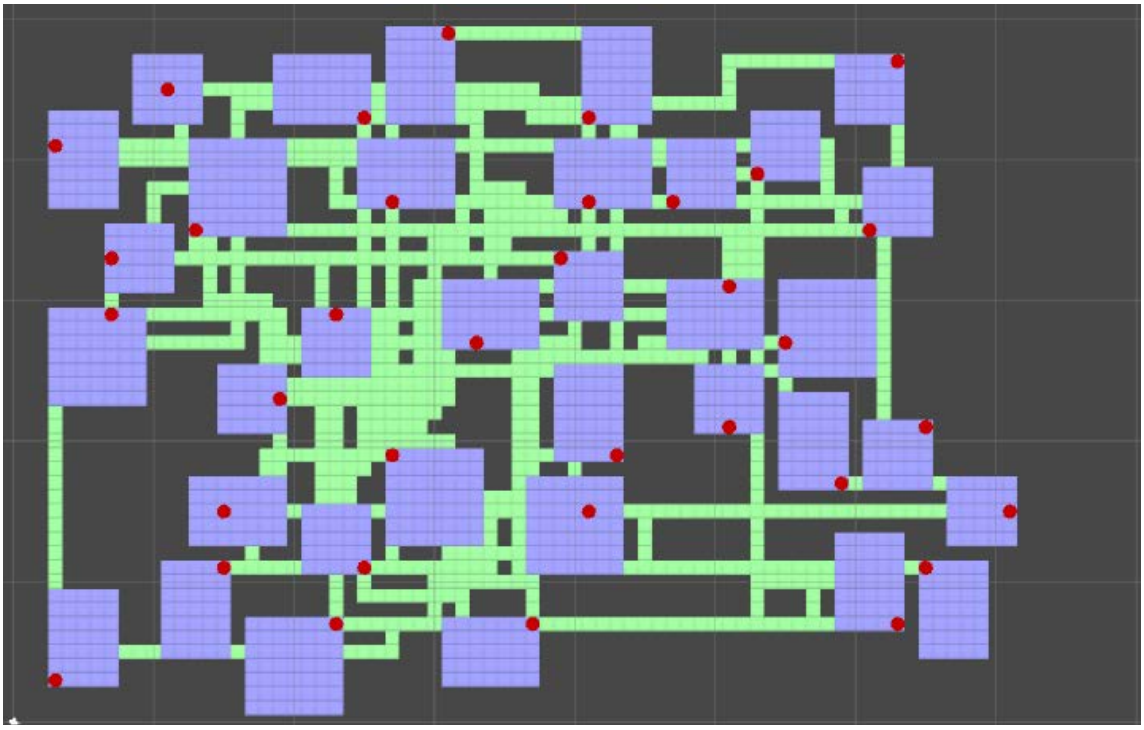
Kun huoneet oli saatu generoitua, piti seuraavaksi saada aikaiseksi niitä yhdistävät käytävät, mikä oli huomattavasti haastavampi tehtävä. Yksinkertaisin ideani lähteä toteuttamaan käytäviä oli tehdä niitä kahden huoneen välille niin kauan, kunnes kaikki huoneet on käyty läpi. Koska algoritmin generoimat huoneet on lisätty satunnaisiin paikkoihin kentälle, niin käytävät vaeltaisivat huoneiden läpi ja toistensa päältä tai ylitse.

```
public List<Vector2> corridors = new List<Vector2>();
public Transform tileCorridor;
public bool[,] floor;
public List<Vector2> positionsInsideRooms = new List<Vector2>();

void Awake()
{
    //...

    floor = new bool[dungeon.width + 1, dungeon.height + 1];

    AddRooms();
    AddCorridors();
    SpawnFloor();
}

void AddCorridors()
{
    foreach (Rect room in rooms)
    {
        //Get random position inside room that is odd number
        int xPos = Random.Range(Mathf.FloorToInt(room.xMin / 2),
Mathf.FloorToInt(room.xMax / 2)) * 2 + 1;
        int yPos = Random.Range(Mathf.FloorToInt(room.yMin / 2),
Mathf.FloorToInt(room.yMax / 2)) * 2 + 1;

        //Add the position to list for later use
        positionsInsideRooms.Add(new Vector2(xPos, yPos));
    }
}
```

Lisäsin koodin alkuun neljä uutta julkista muuttujaa, Vector2-tyyppisen *corridors*-lista, Transform-tyyppisen muuttujan *tileCorridor*, kaksiulotteisen boolean-tyyppisen arrayn *floor* sekä listan *positionsInsideRooms*. Inspectorissa myös raahasin *tileCorridor*-muuttujalle arvoksi aikaisemmin luotu *TileCorridor*-prefab. Lisäsin *Awake*-metodin sisälle alustus juuri luodulle arraylle *floor*, jossa sen kooksi määritellään luolaston leveys ja korkeus plus yksi, jolloin saadaan jokaiselle ruudulle oma solu. Lisäsin *Awake*-metodiin myös kutsun seuraavaksi luotuun *AddCorridors*-metodiin ennen lattioiden lisäämistä.

Loin uuden metodin *AddCorridors*, jonka sisällä olevassa silmukassa algoritmi valitsee satunnaiset paikat huoneista, joista käytävät alkavat ja mihin ne loppuvat. Seuraavaksi tein foreach-silmukan, joka käy kaikki *rooms*-listassa olevat huoneet läpi. Silmukan sisällä algoritmi arpoo huoneesta satunnainen ruudun, jonka paikka on pariton. Erottelin aikaisemmin huoneet käyttämällä parittomuutta, mutta sama tekniikka toimii myös käytäville. Ilman tätä rajoitusta käytävät olisivat voineet kulkea aivan vierekkäin luoden

erittäin sekavan ja huonon lopputuloksen (Kuva 14). Lopuksi vielä lisäsin nämä arvotut paikat alussa luotuun listaan myöhempää käyttöä varten.



Kuva 14. Generoidut käytävät, kun niitä ei oltu millään tavalla rajoitettu.

```
void AddCorridors()
{
    //...

    //All positions added to list, now we just make corridors from
position 0 to 1, 1 to 2, 2 to 3 and so on...
    for (int i = 0; i < positionsInsideRooms.Count; i++)
    {
        Vector2 startPos = positionsInsideRooms[i];
        Vector2 endPos;
        if (i < positionsInsideRooms.Count - 1)
            endPos = positionsInsideRooms[i + 1];
        else
            endPos = positionsInsideRooms[0];

        //Get difference in position for distance
        int xDist = Mathf.Abs(Mathf.RoundToInt(startPos.x - end-
Pos.x));
        int yDist = Mathf.Abs(Mathf.RoundToInt(startPos.y - end-
Pos.y));

        //Get direction where corridor will be built
        int xDir = (startPos.x < endPos.x) ? 1 : -1;
        int yDir = (startPos.y < endPos.y) ? 1 : -1;

        Vector2 curPos = startPos;
        int loopCount = 0;
        bool directionX = true;
        while (xDist > 0 || yDist > 0)
        {
            //Prevent infinite loops (paranoia)
```

```
            if (loopCount++ > dungeon.width + dungeon.height)
                break;

            switch (directionX)
            {
                case true:
                    if (xDist < 1)
                        goto case false;
                    curPos.x += xDir;
                    corridors.Add(curPos);
                    curPos.x += xDir;
                    xDist -= 2;
                    break;
                case false:
                    if (yDist < 1)
                        goto case true;
                    curPos.y += yDir;
                    corridors.Add(curPos);
                    curPos.y += yDir;
                    yDist -= 2;
                    break;
            }
            corridors.Add(curPos);

            if (Random.Range(0f, 1f) < 0.15f && (xDist > 0 && yDist
> 0))
                directionX = !directionX;
        }
    }
}
```

Olin nyt tehnyt algoritmin, joka arpoo huoneille satunnaiset paikat *positionsInsideRooms*-listalle, joten seuraavaksi oli aika käyttää niitä ja luoda käytäviä vaeltamaan niiden välille. Aloitin luomalla for-silmukan, jossa yhdistetään ensimmäinen paikka toiseen, toinen kolmanteen ja niin edelleen, kunnes lopuksi viimeinen huone yhdistetään takaisin ensimmäiseen. Kun algoritmi on käynyt koko listan lävitse, on kaikki huoneet yhdistettyinä toisiinsa. Koska algoritmi lisää huoneet satunnaisessa järjestyksessä, tulevat käytävät menemään huoneiden ja toisen käytävien läpi luoden useita mahdollisia reittejä.

Seuraavaksi loin kaksi Vector2-tyyppistä muuttujaa *startPos* sekä *endPos*, joiden välille käytävä tällä silmukan kierroksella tehdään. Laskin muuttujille *xDist* ja *yDist* näiden paikkojen etäisyydet molemmille akseleille erikseen käyttämällä. Sen jälkeen hain suunnan, johon se tulee käytävää luomaan vertailemalla molempien x ja y -koordinaatteja. Määritin ja alustin vielä seuraavat kolme uutta muuttujaa ennen seuraavaa vaihetta:

- *curPos*, jolla ”kuljetaan” käytävän reitti läpi,
- *loopCount*, jonka tarkoitus on ainoastaan estää loputon silmukka ja
- *directionX*, jonka arvo on true, kun liikutaan x-akselilla.

Käytävän luomiseen käytin while-silmukkaa, jota ajetaan niin kauan, kunnes muuttujien *xDist* ja *yDist* arvot tippuvat nollaan eli ollaan loppupisteessä. Heti silmukan alkuun lisäsin laskurin, joka lisää *loopCount*-muuttujan arvoon yhden jokaisella kierroksella, ja mikäli sen arvo nousee yli pi-

simmän mahdollisen matkan luolaston kulmasta kulmaan, niin silmukka rikotaan. Tätä ei kuitenkaan pitäisi koskaan tapahtua, mikäli ohjelma toimii oikein.

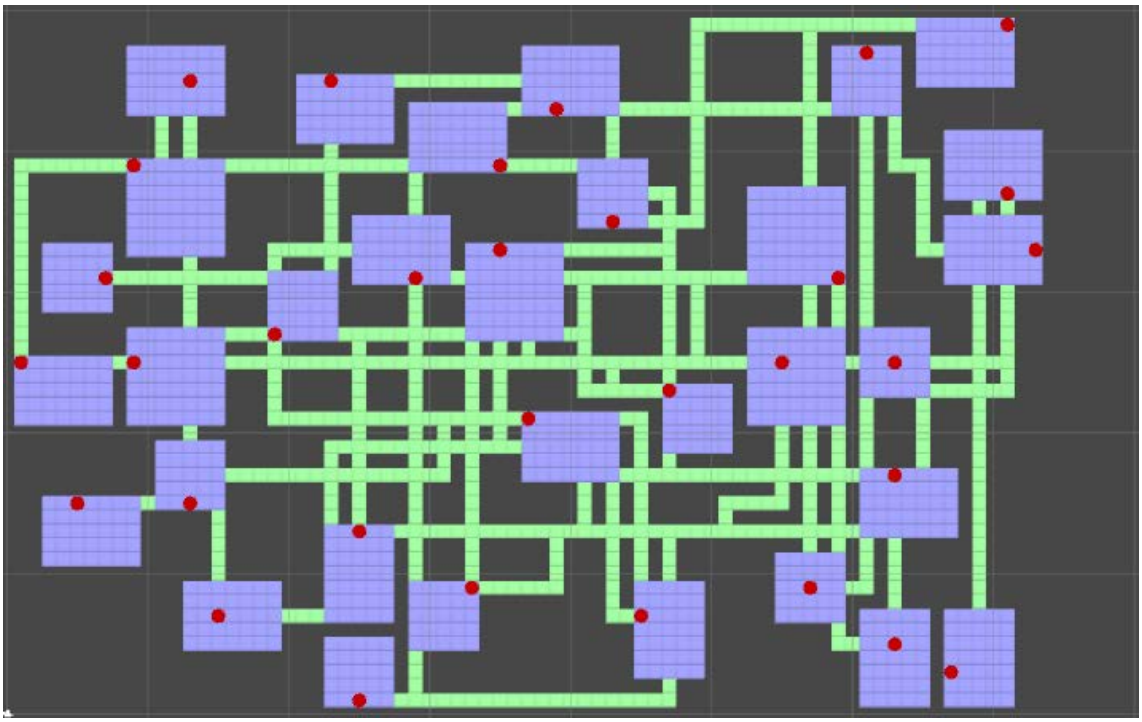
Liikkumisen logiikkana käytin switch-case-valintarakennetta, joka toimii *directionX*-muuttujan arvon perusteella. Molempien case-haarojen alkuun lisäsin ehdon, joka toteutuu, kun *xDist*- tai *yDist* -muuttujan arvo on jo nolla. Muussa tapauksessa algoritmi liikkuu *xDir*- tai *yDir*-muuttujan vastaiseen suuntaan ensin kerran ja tallentaa tämän aivan alussa luotuun *corridors*-listaan ja heti perään algoritmi liikkuu vielä toisen kerran, jotta se pysyy parittomissa ruuduissa. Tämän jälkeen algoritmi vähentää matkaa mittaavasta muuttujasta kaksi. Lopuksi vielä lisäsin *corridors*-listaan jälkimmäisen liikkeen jälkeisen paikan ja loin ehdon suunnan vaihtamiselle, joka suoritetaan, mikäli arvottu luku väliltä 0.0 ja 1.0 on alle 0.15.

```
void SpawnFloor()
{
    //Loop through all "room" rectangles in list "rooms"
    foreach (Rect room in rooms)
    {
        //Loop through all tiles that there is inside the room
        for (int x = Mathf.FloorToInt(room.x); x < Mathf.Floor-
ToInt(room.x + room.width); x++)
        {
            for (int y = Mathf.FloorToInt(room.y); y < Mathf.Floor-
ToInt(room.y + room.height); y++)
            {
                //...
                newTile.name = "TileRoom (" + x + ", " + y + ")";
                floor[x, y] = true;
            }
        }
    }

    foreach (Vector2 corridorTile in corridors)
    {
        int x = Mathf.RoundToInt(corridorTile.x);
        int y = Mathf.RoundToInt(corridorTile.y);

        if (floor[x, y] == true)
            continue;

        Transform newTile = Instantiate(tileCorridor, new Vector3(x,
0, y), Quaternion.identity) as Transform;
        newTile.parent = GameObject.Find("TilesForCorridors").trans-
form;
        newTile.name = "TileCorridor (" + x + ", " + y + ")";
        floor[x, y] = true;
    }
}
```

SpawnFloor-metodissa huoneiden lattioiden lisäämiselle ei tullut mitään muuta muutosta kuin sisimmäisen for-silmukan lopussa muutetaan *floor*-arrayn arvoksi true jokaista ruutua vastaavalta kohdalta. Käytäviä varten luodaan foreach-silmukka, joka käy kaikki käytäville aikaisemmin lisätyt paikat läpi. Otetaan aluksi talteen ruudun x- ja y-koordinaatit ja sen jälkeen tarkastetaan, onko siinä kohdassa jo valmiiksi ruutu, mikäli on, niin jatketaan silmukassa seuraavaan ruutuun. Muussa tapauksessa luodaan uusi

ruutu vastaavasti kuin aikaisemmin huoneille ja lisätään uusi ruutu *floors*-listaan.

Edellä tehtyjen muutosten jälkeen käytävät olivat jo selkeitä käytäviä toisin kuin kuvassa 15 ja luolasto alkoi näyttää paljon paremmalta, mutta käytäviä oli vielä aivan liikaa (Kuva 15). Ratkaisuksi tähän keksin kaksi vaihtoehtoa: joko yritän yhdistellä näitä käytäviä toisiinsa ja lopetan käytävän piirtämisen, kun se yhdistyy muihin käytäviin tai alun perinkin vedän käytävän huoneesta käytävään, mikäli sopiva käytävä löytyy tarpeeksi läheltä. Molemmilla tavoilla voi tulla ongelma, jota aikaisemmin ei ollut, eli luolasto voi jakautua kahteen tai useampaan erilliseen osaan. Sen välttämiseksi pitäisi myös saada kirjoitettua algoritmi, joka tarkastaa kaikkien huoneiden olevan kiinni yhdessä osassa ja tarvittaessa lisää käytäviä yhdistämään näitä osia.



Kuva 15. Generoidut käytävät kun ne on rajoitettu kulkemaan parittomilla ruuduilla. Kuvassa ylimääräiset punaiset pisteet näyttävät *AddCorridors*-metodissa arvotut pisteet *positions*-listalta.

```
public bool[,] floorRoom;
public bool[,] floorCorridor;

void Awake()
{
    //...
    floorRoom = new bool[dungeon.width + 1, dungeon.height + 1];
    floorCorridor = new bool[dungeon.width + 1, dungeon.height + 1];

    AddRooms();
    SpawnFloor();
    AddCorridors();
}

//Spawns floor for rooms
void SpawnFloor()
```

```
{
    //Loop through all "room" rectangles in list "rooms"
    foreach (Rect room in rooms)
    {
        //Loop through all tiles that there is inside the room
        for(int x = Mathf.FloorToInt(room.x); x < Mathf.Floor-
ToInt(room.x + room.width); x++)
        {
            for (int y = Mathf.FloorToInt(room.y); y < Mathf.Floor-
ToInt(room.y + room.height); y++)
            {
                //Instantiate new tiles, make "TilesForRooms" their
parent and rename them to make debuggin easier
                Transform newTile = Instantiate(tileRoom, new
Vector3(x, 0, y), Quaternion.identity) as Transform;
                newTile.parent = GameObject.Find
("TilesForRooms").transform;
                newTile.name = "TileRoom (" + x + ", " + y + ")";
                floor[x, y] = true;
                floorRoom[x, y] = true;
            }
        }
    }
}

//Returns true if on top of other corridor
bool AddCorridorTile(Vector2 pos)
{
    bool collidedWithCorridor = false;
    int x = Mathf.RoundToInt(pos.x);
    int y = Mathf.RoundToInt(pos.y);

    if (floorCorridor[x, y] == true)
    {
        collidedWithCorridor = true;
    }
    else if (floor[x, y] == false)
    {
        corridors.Add(pos);
        Transform newTile = Instantiate(tileCorridor, new Vector3(x,
0, y), Quaternion.identity) as Transform;
        newTile.parent = GameObject.Find
("TilesForCorridors").transform;
        newTile.name = "TileCorridor (" + x + ", " + y + ")";
        floor[x, y] = true;
        floorCorridor[x, y] = true;
    }
    return collidedWithCorridor;
}
```

Muutin koodia lisäämällä heti alkuun kaksi uutta julkista muuttujaa *floorRoom* ja *floorCorridor*, jotka ovat alustettuina identtisiä aikaisemmin luodun *floor*-muuttujan kanssa. Nämä tulevat myös toimimaan samalla tavalla, mutta ensimmäinen pitää sisällään vain huoneiden ruudut ja jälkimmäinen käytävien ruudut, kun aikasempi *floor* sisältää kaikki ruudut. Alustetin myös nämä muuttujat *Awake*-metodissa samanlaisiksi ja vaihdoin vielä metodien kutsumisen järjestystä niin, että metodia *AddCorridors* kutsutaan viimeisenä, jotta arraysta *floor* löytyy jo tiedot huoneiden ruuduista.

Jatkoin poistamalla *SpawnFloor*-metodista jälkimmäinen foreach-silmukan, jolla aikaisemmin luotiin lattia käytäville ja tein sitä varten uuden metodin *AddCorridorTile*. Tämä metodi lisää käytävien lattiat yksi ruutu kerrallaan samalla, kun käytävää luodaan, ja yhtäaikaisesti myös tarkistaa, ollaanko jo jonkin aikaisemman käytävän päällä. Lisäsin myös sisimmäisen silmukan loppuun kohdan, jossa muutetaan *floorRoom* arrayn arvoksi ruudun kohdalta true.

Loin uuden metodin *AddCorridorTile*, joka palauttaa boolean-tyyppisen arvon ja ottaa vastaan Vector2-koordinaatit, jotka tallennetaan muuttujaan *pos*. Tein ja alustin metodin alussa muuttujan *collidedWithCorridor*, joka on tyyppiä boolean ja annoin sille aloitusarvoksi false sekä kokonaislukutyyppiset muuttujat *x* ja *y*, joihin pyöristin metodille lähetetyn *pos*-muuttujan vastaavat arvot.

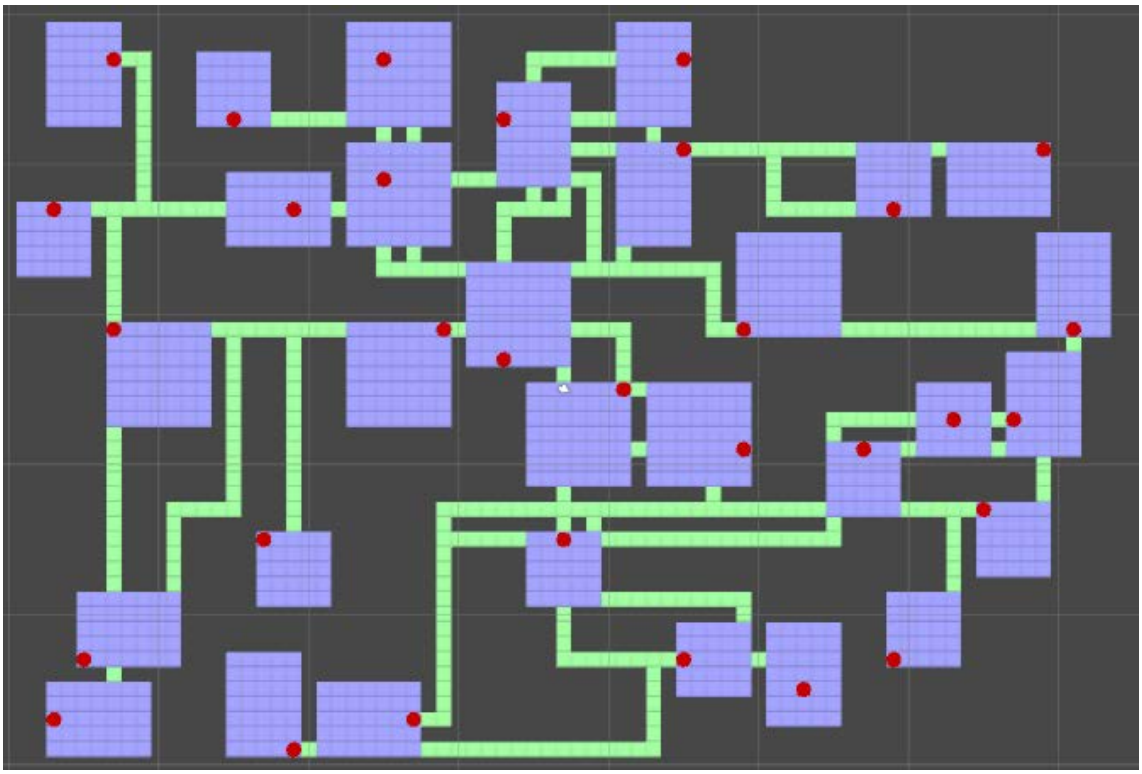
Tarkastin, onko nykyinen paikka jo jonkin aikaisemman käytävän ruudun kohdalla arraysta *floorCorridor*. Mikäli nykyinen paikka oli jo käytössä, määritin booleanin *collidedWithCorridor* arvoksi true. Muussa tapauksessa tarkistin, oliko nykyinen ruutu jo muuten käytössä, eli tässä tapauksessa ollaanko jonkin huoneen alueella. Mikäli ruutu oli täysin tyhjä, lisättiin se *corridors*-listaan. Myös tähän lisäsin vastaavan koodin kuin mitä aikaisemmin käytin *SpawnFloor*-metodissa foreach-silmukan sisällä, missä luodaan uusi ruutu, määritetään sen parentiksi objekti *TilesForCorridors*, annetaan sille nimeksi ”TileCorridor (*x*, *y*)”, jonka x sekä y tulevat muuttujien arvoista ja muutetaan myös *floor*- sekä *floorCorridor*-array:n arvoksi true näiltä koordinaateilta. Viimeiseksi koodi vielä palauttaa *collidedWithCorridors*-booleanin arvo.

```
void AddCorridors()
{
    //...
        while (xDist > 0 || yDist > 0)
        {
            //Prevent infinite loops (paranoia)
            if (loopCount++ > dungeon.width * dungeon.height)
                break;

            switch (directionX)
            {
                case true:
                    if (xDist < 1 && yDist > 0)
                        goto case false;
                    curPos.x += xDir;
                    AddCorridorTile(curPos);
                    curPos.x += xDir;
                    xDist -= 2;
                    break;
                case false:
                    if (yDist < 1 && xDist > 0)
                        goto case true;
                    curPos.y += yDir;
                    AddCorridorTile(curPos);
                    curPos.y += yDir;
                    yDist -= 2;
                    break;
            }
            if (AddCorridorTile(curPos))
```

```
            {
                break;
            }

            if (Random.value < 0.15f)
                directionX = !directionX;
        }
    }
}
```

AddCorridors-metodi tarvitsi vain muutaman pienen muutoksen. Kun aikaisemmin jokaisen liikkeen jälkeen lisättiin *corridors*-listaan uusi paikka, niin nyt se lähetään juuri luodulle *AddCorridorTile*-metodille. Ensimmäisen ruudun aikana ei palautetulla booleanilla tehdä mitään, mutta jälkimmäisen kohdalla sen arvo tarkastetaan. Mikäli palautetun booleanin arvo on true, niin lopetetaan käytävän piirtäminen eli breakataan ulos while-silmukasta. Vaihdoin myös aikaisemmin käytössä olleen Random.Range-metodin tilalle yksinkertaisemman staattisen muuttujan Random.value, joka palauttaa tarvittavan liukuluvun väliltä 0.0 ja 1.0.

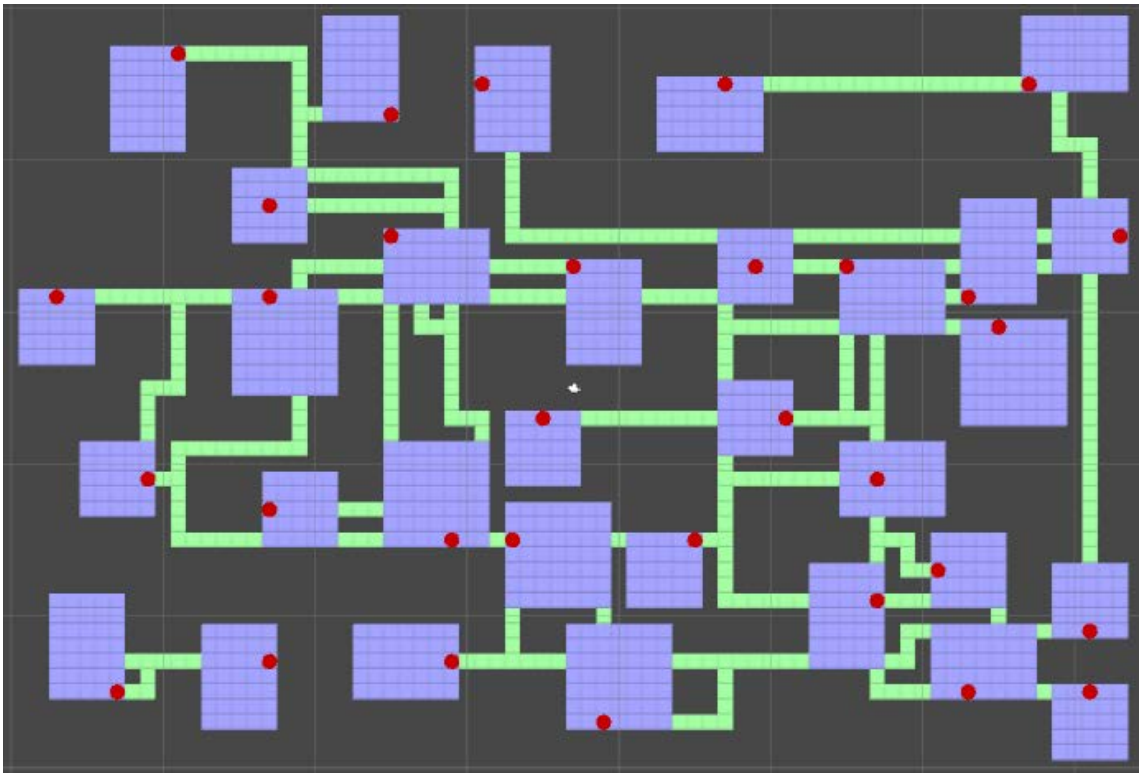
Näillä muutoksilla uusi luolasto alkoi olla erittäin lähellä sitä, mitä lähdin alun perinkin hakemaan (Kuva 16). Suurimpaan osaan huoneista on useampia reittejä ja ne muodostavat käytävien kanssa useita mahdollisia reittejä pelaajalle. Käytävät kulkevat kohtuullisen loogisesti kohti seuraavaa huonetta kuten voisi olettaa ihmisen rakentamasta verkostosta. Enää itseäni ei erityisemmin häirinnyt muu kuin muutamat aivan rinnakkain kulkevat käytävät, joille pitäisi vielä jotain saada tehtyä.



Kuva 16. Käytävät kulkevat edelleen parittomilla ruuduilla, mutta nyt ne pysähtyvät kohdatessaan toisen käytävän, joka rajoittaa niiden määrää huomattavasti.

Muutamalla kymmenellä generoinnilla en saanut aikaiseksi luolastoa, joka olisi jakautunut kahteen tai useampaan erilliseen osaan, mutta ainakin oman ajatuksen juoksun mukaan se oli tässä vaiheessa täysin mahdollista, joskin melko epätodennäköistä.

Yritin kymmeniä kertoja saada generoitua luolastoa, jossa jokin osa olisi muusta luolastosta erillään eikä se kertaakaan tapahtunut. Tiesin sen olevan kuitenkin mahdollista ja lopulta, kun jatkoin projektia eteenpäin, se viimein myöskin tapahtui (Kuva 17). Irrallisten osien yhdistämiseksi muuhun luolastoon täytyi minun vielä tehdä tarkastus, joka käy koko luolaston läpi ja tarkastaa sen olevan yhdessä osassa ja lisää tarvittaessa käytävän erillisten osien välille.



Kuva 17. Vasempaan alakulmaan on generoitunut kaksi huonetta, jotka ovat erillään muusta luolastosta. Oikeanpuoleisesta huoneesta on ensin lähtenyt käytävä vasemmalle ja sen jälkeen sieltä uusi käytävä jonnekin muualle, mutta se törmäsi edelliseen käytävään ja pysähtyi siihen. Ohjelman pitäisi tunnistaa, kun tällainen tapahtuu, ja luoda uusi käytävä, joka yhdistää irralleen jääneet osat muuhun luolastoon.

5.5 Irrallisen osan löytäminen

Ensimmäinen ongelma oli luoda algoritmi, joka osaisi tunnistaa, kun luolaston kaikki osat eivät ole kiinni toisissaan. Ongelman ratkaisemiseksi päätin luoda flood fill -algoritmin ideaan perustuvan algoritmin, joka valitsee yhden ruudun ja lähtee käymään koko luolastoa läpi ruutu kerrallaan kulkien aina viereiseen ruutuun. Kun algoritmi ei enää löydä vapaita ruutuja, voidaan tarkistaa, onko kaikki ruudut luolastossa käyty lävitse ja mikäli ei,

niin tiedetään, että jossain on irrallinen osa. Tämä algoritmi toistetaan, kunnes luolaston kaikki ruudut on käyty läpi ja näin on löydetty kaikki erilliset osat.

Algoritmi, joka hakee kaikki luolaston alueet, on sen verran pitkähkö, että yritän selvyyden vuoksi jakaa sen muutamaan osaan ja käydä nämä osat yksittäin läpi.

```
public int region;
public List<bool[,]> regions = new List<bool[,]>();
public List<List<int>> regionIndex = new List<List<int>>();
public bool[,] solvedTiles;
public List<Vector2Int> unsolvedDirs = new List<Vector2Int>();

void Awake()
{
    //...
    GameObject.Find("Main Camera").transform.position = new Vector3
(dungeon.width / 2f, 50f, dungeon.height / 2f);

    region = 0;
    floor = new bool[dungeon.width + 1, dungeon.height + 1];
    floorRoom = new bool[dungeon.width + 1, dungeon.height + 1];
    floorCorridor = new bool[dungeon.width + 1, dungeon.height + 1];
    solvedTiles = new bool[dungeon.width + 1, dungeon.height + 1];

    AddRooms();
    SpawnFloor();
    AddCorridors();

    CheckConnections();
}
```

Ensimmäiseksi loin valmiiksi seuraavat uudet muuttujat:

- Kokonaisluku-tyyppinen muuttuja *region*, joka toimii yksinkertaisesti erillisten alueiden järjestysnumerona.
- Lista *regions*, joka pitää sisällään kaksiulotteisen boolean-tyyppisen arrayn, mihin merkitään erillisten alueiden sisältämät ruudut.
- Lista *regionIndex*, jonka sisällä on lista kokonaisluvuista ja tämä sisällä oleva lista toimii järjestyslukuna eri alueiden huoneille.
- Kaksiulotteinen boolean-tyyppinen array solvedTiles, johon merkitään luolastoa tarkastaessa jokainen läpikäyty ruutu.
- Vector2Int-tyyppinen lista unsolvedDirs, johon lisätään ohitetut ruudut, kun luolastoa käydään lävitse ja mihin voidaan palata, kun aikaisemmin valittu suunta on käyty loppuun.

Lisäsin *Awake*-metodiin koodinpätkän, joka yksinkertaisesti siirtää kameran keskelle luolastoa. Seuraavaksi asetin muuttujan *region* aloitusarvoksi 0 ja alustin arrayn *solvedTiles* vastaamaan aikaisemmin alustettuja arrayta eli sisältämään solun luolaston jokaiselle ruudulle. Muut uudet muuttujat alustin jo heti niiden luontivaiheessa. Lopuksi vielä lisäsin loppuun kohdan, jossa kutsutaan myöhemmin luotavaa *CheckConnections*-metodia.

```
//Find first tile that isn't checked yet by CheckConnections()
Vector2Int GetFirstTile()
{
    Vector2Int firstTile = new Vector2Int();

    for(int y = 0; y < floor.GetLength(1); y++)
    {
        for (int x = 0; x < floor.GetLength(0); x++)
        {
            if(floor[x, y] && !solvedTiles[x, y])
            {
                firstTile = new Vector2Int(x, y);
                goto End; //Little ugly but works
            }
        }
    }

End:
    return firstTile;
}
```

Ennen kuin pystyin luomaan metodin *CheckConnections*, tarvitsin sille avuksi toisen metodin, joka etsii ruudun, josta lähdetään tarkastuksessa liikkeelle. *GetFirstTile*-metodi palauttaa *Vector2Int*-tyyppisen muuttujan, joka on ensimmäisen löydetyn ruudun paikka, jota ei ole vielä tarkastettu *CheckConnections*-metodissa.

Seuraavaksi loin uuden metodin *CheckConnections* (Liite 1), jonka alussa *regionIndex*-listaan lisätään uusi lista vuorossa olevalle alueelle ja luodaan sekä alustetaan seuraavat uudet muuttujat:

- Boolean-tyyppinen muuttuja *runAgain*, jonka arvo on oletuksena false. Jos luolastossa on vielä tarkastamattomia erillisiä osia, muutetaan *runAgain*-muuttujan arvoksi true.
- Vector2Into-tyyppinen muuttuja *pos*, jonka arvoksi tulee GetFirstTile-metodin palauttama arvo.
- Kokonaisluku *loopCount*, jota käytetään kuten aikaisemminkin estämään loputtomia silmukoita, mikäli koodissa on virhe.
- Lista *possibleDirs* Vector2Int-tyyppisistä muuttujista, johon lisätään aina väliaikaisesti kaikki suunnat, joihin seuraavalla silmukan kierroksella voidaan liikkua.

Toteutin tarkastuksen päättymättömällä while-silmukalla, josta breakataan ulos tiettyjen ehtojen täyttyessä. Silmukan alkuun lisäsin myös aikaisemminkin käytetyn tarkistuksen, joka lopettaa silmukan, mikäli se ei ole ohjelmointivirheen takia päättynyt. Vielä ennen varsinaista ehjoten tarkastelua tyhjennetään possibleDirs-lista, jotta se on tyhjä silmukan jokaisella kierroksella.

Ensimmäisenä koodi tarkastaa, löytyykö tarkastettava koordinaatti arraysta *floorCorridor*, eli selvitetään, onko kyseisessä kohdassa käytävän vai huoneen ruutu. Näiden jälkeen tarkastetaan, onko muuttujan *runAgain* arvo true. Mikäli se on, niin kasvatatetaan *region*-muuttujan arvoa yhdellä ja kutsutaan *CheckConnections*-metodia uudelleen.

5.5.1 Käytävän ruutu

Kun kyseessä on käytävän ruutu, merkitään se ensiksi *solvedTiles*-arrayhyn tarkistetuksi. Tämän jälkeen algoritmi tarkistaa tarkastettavana olevan ruudun ylä- ja alapuolella sekä sivuilla olevat koordinaatit. Jos tarkastettavasta kohdasta löytyy merkintä *floor*-arrayssa, mutta ei sitä ei ole merkitty *solvedTiles*-arrayhyn, niin uusi tarkastamaton ruutu on löydetty ja se lisätään *possibleDirs*-listaan.

Seuraavaksi algoritmi käy läpi *unsolvedDirs*-listan tarkastaen, löytyykö siitä jo edellisessä kohdassa löydettyjä tarkastamattomia ruutuja. Mikäli vastaavuus löytyy, se poistetaan *possibleDirs*-listasta.

Lopuksi määritetään seuraavaksi vuorossa oleva ruutu seuraavin ehdoin:

- Jos *possibleDirs*-listalla on vain yksi arvo, siitä tulee seuraava ruutu.
- Jos *possibleDirs*-listalla on useampi kuin yksi arvo, ensimmäisestä tulee seuraava ruutu ja loput lisätään *unsolvedDirs*-listalle.
- Jos *possibleDirs*-lista on tyhjä ja *unsolvedDirs*-listalla on vähintään yksi arvo, niin *unsolvedDirs*-listan ensimmäisestä arvosta tulee seuraava ruutu ja se poistetaan listalta.
- Jos mikään edellisistä ehdoista ei toteudu, on algoritmi käynyt läpi kaikki aluuen ruudut. Tässä tapauksessa tarkastetaan *GetFirstTile*-metodilla, onko luolastossa tarkastamatonta irrallista osaa, ja jos on, merkitään muuttujalle *runAgain* arvoksi true. Kummassakin tapauksessa lopuksi breakataan ulos while-silmukasta.

5.5.2 Huoneen ruutu

Kun kyseessä on huoneen ruutu, algoritmi luo ja alustaa uuden Rect-tyyppisen muuttujan *curRoom*. Seuraavaksi algoritmi käy *rooms*-listan huoneet läpi for-silmukalla tarkastaen, minkä sisällä nykyinen *pos*-koordinaatti sijaitsee. Kun algoritmi on löytänyt huoneen, se määritetään *curRoom*-muuttujan arvoksi ja lisätään huoneen järjestysluku *regionIndex*-listan *region*-kohdassa olevaan listaan. Järjestysluku saadaan for-silmukan kierrosluvusta. Koska huone on jo löydetty, koodi breakkaa ulos for-silmukasta, jolloin säästytään turhilta tarkastuksilta.

Huoneen ollessa tiedossa algoritmi hakee siitä int-tyyppisiin muuttujiin *xMin*, *xMax*, *yMin* ja *yMax* talteen reunien maksimi- ja minimiarvot sekä x- että y-akseleilla. Tämän jälkeen algoritmi looppaa huoneen ruudut läpi x-akselilla tarkastaen listoilta *floorCorridor* sekä *solvedTiles*, löytyykö huoneen ylä- tai alapuolelta näistä koordinaateista käytävänruutua, jota ei ole vielä tarkastettu. Löydetyt ja tarkastamattomat ruudut lisätään *unsolvedDirs*-listaan. Seuraavaksi vastaava prosessi käydään läpi y-akselin suuntaisesti tarkastaen sivuilla olevia koordinaatteja.

Tarkastusten jälkeen algoritmi looppaa huoneen kaikki ruudut läpi merkiten ne *solvedTiles*-listaan tarkastetuiksi. Lopuksi algoritmi tarkastaa, onko *unsolvedDirs*-listalla vähintään yksi arvo, ja jos on, niin *unsolvedDirs*-listan ensimmäisestä arvosta tulee seuraava ruutu ja se poistetaan listalta. Muussa tapauksessa on algoritmi käynyt läpi kaikki alueen ruudut ja tarkastetaan

GetFirstTile-medotilla, onko luolastossa tarkastamatonta irrallista osaa. Mikäli se löytyy, merkitään *runAgain*-muuttujan arvoksi true ja breakataan ulos while-silmukasta, tai jos tarkastamatonta aluetta ei löytynyt, niin vain breakataan ulos.

5.6 Irrallisten alueiden yhdistäminen

Luvussa 5.5 käsiteltiin algoritmi, joka etsii luolastosta erilliset alueet ja listaa niiden sisältämät huoneet *regionIndex*-listaan. Seuraavana tehtävänä oli saada luotua toinen algoritmi, joka puolestaan yhdistää nämä erilliset alueet käytävällä.

Ideani tämän toteuttamiseksi oli perin yksinkertainen. Algoritmissa on kaksi osaa, joista ensimmäinen käy läpi erillisten alueiden huoneet, laskee niiden välisen etäisyyden ja vertaa sitä lyhimpään saatuun etäisyyteen. Kun se on käynyt kaikki mahdolliset yhdistelmät läpi, niin tuloksena on kaksi huonetta eri alueilta, joiden välimatka on kaikista lyhin. Tämän jälkeen algoritmin toinen osa luo käytävän näiden kahden huoneen välille yhdistäen erilliset alueet. Molempien osioiden koodi toistuu, kunnes kaikki ensimmäisestä alueesta irrallaan olevat alueet on käyty lävitse.

```
void Awake()
{
    //...
    if (region > 0)
        ConnectRegions();
}
```

Awake-metodiin ei tarvita kuin pieni lisäys, joka tarkistaa, onko *region*-muuttujan arvo enemmän kuin nolla eli onko alueita enemmän kuin yksi. Mikäli arvo on yksi tai enemmän, niin kutsutaan metodia *ConnectRegions*.

```
void ConnectRegions()
{
    List<Vector2> positions = new List<Vector2>();
    List<Vector2> tempPos = new List<Vector2>();
    float shortestDistance = float.MaxValue;
    for(int regionID = 1; regionID < regionIndex.Count; regionID++)
    {
        for(int cellOfFirstRegion = 0; cellOfFirstRegion < regionIn-
dex[0].Count; cellOfFirstRegion++)
        {
            for(int cellOfNRegion = 0; cellOfNRegion < regionIn-
dex[regionID].Count; cellOfNRegion++)
            {
                float distance = Vector2.Distance(positionsIn-
sideRooms[regionIndex[0][cellOfFirstRegion]], positionsIn-
sideRooms[regionIndex[regionID][cellOfNRegion]]);
                if (distance < shortestDistance)
                {
                    shortestDistance = distance;
                    tempPos.Clear();
                    tempPos.Add(positionsInsideRooms[regionIn-
dex[0][cellOfFirstRegion]]);
                    tempPos.Add(positionsInsideRooms[regionIndex[re-
gionID][cellOfNRegion]]);
                }
```

```
                }
            }
            positions.AddRange(tempPos);
        }

        for (int i = 0; i < positions.Count; i += 2)
                GenerateConnectingCorridor(new Vector2Int(positions[i]),
new Vector2Int(positions[i + 1]));
}
```

Loin uuden metodin *ConnectRegions*, joka hoitaa erillisten alueiden yhdistämisen. Aluksi loin ja alustin kaksi Vector2-tyyppistä listaa *positions* ja *tempPos* sekä float-tyyppisen muuttujan *shortestDistance*, jonka arvoksi määritin float-tyypin maksimiarvon.

Seuraavaksi lisäsin kolme sisäkkäistä for-silmukkaa. Niistä ensimmäinen käy läpi *regionIndex*-listan irralliset alueet jättäen väliin ensimmäisen, jota pidetään pääalueena, johon muut alueet yhdistetään. Toinen silmukka käy läpi ensimmäisen eli pääalueen huoneet. Kolmas, viimeinen silmukka, käy puolestaan läpi ensimmäisen silmukan mukaan vuorossa olevan irrallisen alueen huoneet.

Silmukoiden sisällä luodaan uusi muuttuja *distance*, joka on tyyppiä float. Muuttujan *distance* arvoksi lasketaan kahden koordinaatin välinen matka käyttämällä Vector2.Distance-metodia. Huoneille on aikaisemmin luvussa 5.4 määritelty koordinaatit, joista käytävät lähtevät liikkeelle, ja ne lisättiin myös *positionsInsideRooms*-listalle.

Koodi ottaa ensin *regionIndex*-listalta listan indeksistä 0. Tältä listalta koodi hakee arvon indeksistä, joka vastaa muuttujan *cellOfFirstRegion*-arvoa, joka on toisen for-silmukan kierrosluku. Tuloksena on ensimmäisen alueen n:s (nth) solun indeksi numero, jota käyttämällä saadaan *positionsInsideRooms*-listalta Vector2.Distance-metodille ensimmäinen koordinaatti. Toinen koordinaatti saadaan samalla tavalla, mutta ensimmäisen alueen sijaan käytetään ensimmäisen for-silmukan mukaista aluetta ja *cellOfNRegion*-muuttujan mukaista arvoa, joka puolestaan on kolmannen for-silmukan kierrosluku.

Kun muuttujan *distance* arvo on saatu selville, tarkistetaan, onko se pienempi kuin *shortestDistance*-muuttujan arvo. Jos on,

- määritellään muuttujan *shortestDistance* arvoksi muuttujan *distance* arvo,
- tyhjennetään lista *tempPos*,
- lisätään *tempPos*-listalle koordinaatit, joita käytettiin matkan laskemisessa.

Ensimmäisen silmukan lopussa, kun kaksi sisempää on ajettu läpi, lisätään *positions*-listalle *tempPos*-listan sisältämät arvot. Tämä siis tehdään jokaiselle ensimmäisen silmukan läpikäymälle alueelle, kun kaikki huoneyhdistelmät on käyty läpi.

ConnectRegions-metodin lopussa ajetaan vielä yksi for-silmukka, joka käy lävitse joka toisen solun listalta *positions*. Jokaisella kierroksella kutsutaan

GenerateConnectingCorridor-metodia, joka käsitellään seuraavassa luvussa. Sille annetaan ensimmäiseksi arvoksi *positions*-listalta for-silmukan kierrosta vastaavan parillisen solun arvo ja toiseksi arvoksi ensimmäistä arvoa seuraava parittoman solun arvo.

```
public void GenerateConnectingCorridor(Vector2Int startPos, Vec-
tor2Int endPos)
{
    print("New connection between points: " + startPos.getPosS-
tring() + endPos.getPosString());

    //Get difference in position for distance
    int xDist = Mathf.Abs(Mathf.RoundToInt(startPos.x - endPos.x));
    int yDist = Mathf.Abs(Mathf.RoundToInt(startPos.y - endPos.y));

    //Get direction where corridor will be built
    int xDir = (startPos.x < endPos.x) ? 1 : -1;
    int yDir = (startPos.y < endPos.y) ? 1 : -1;

    Vector2 curPos = startPos.getAsVector2();
    int loopCount = 0;
    bool directionX = true;
    while (xDist > 0 || yDist > 0)
    {
        //Prevent infinite loops (paranoia)
        if (loopCount++ > dungeon.width * dungeon.height)
            break;

        switch (directionX)
        {
            case true:
                if (xDist < 1 && yDist > 0)
                    goto case false;
                curPos.x += xDir;
                AddCorridorTile(curPos);
                curPos.x += xDir;
                xDist -= 2;
                break;
            case false:
                if (yDist < 1 && xDist > 0)
                    goto case true;
                curPos.y += yDir;
                AddCorridorTile(curPos);
                curPos.y += yDir;
                yDist -= 2;
                break;
        }
        AddCorridorTile(curPos);

        if (Random.value < 0.15f)
            directionX = !directionX;
    }
}
```

Seuraavaksi loin edellisessä luvussa kutsutun *GenerateConnectingCorridor*-metodin. Tämä metodi toimii teknisesti samalla tavalla kuin luvussa 5.4 luotu *AddCorridors*-metodi, mutta sisältää kaksi pientä muutosta. Ensimmäinen muutos on se, että koordinaatit, joiden välille käytävä luodaan, annetaan metodia kutsuttaessa, kun *AddCorridors*-metodissa koordinaatit haettiin for-silmukalla *positionsInsideRooms*-listalta. Toinen muutos on se, ettei lopussa välitetä siitä, mitä *AddCorridorTile*-metodi palauttaa.

6 POHDINTA

Kun alun perin lähdin suunnittelemaan proseduraalisen luolaston generoimista, tiesin sen olevan haastava aihe, vaikka se voi helposti kuulostaa paljon todellisuutta yksinkertaisemmalta. Tehdään vain kasa huoneita eri paikkoihin ja yhdistellään ne käytävillä, helppo homma siis. Todellisuudessa yksinkertaiselta kuulostava tehtävä muodostui nopeasti monimutkaisemmaksi ja haasteellisuus yllätti projektin aikana minutkin.

Perimmäisenä tavoitteena projektissa oli luoda luolasto, joka jossain määrin vastaa ihmisen rakentamaa kompleksia. Mielestäni onnistuin tavoitteessa melko hyvin, sillä käytävät kulkevat suhteellisen suoria reittejä eivätkä kiemurtele matkalla turhaan, kuten luonnollisessa luolastossa voisi tapahtua. Luolaston kokoa voi helposti muuttaa suoraan inspectorissa, ja muuttamalla huoneiden luomisyritysten määrää pystyy vaikuttamaan luolaston tiheyteen. Näiden ominaisuuksien perusteella voin sanoa onnistuneeni projektini päätavoitteessa.

Kokonaisuutena opinnäytetyöni projektiosuus toteutui suunnitellussa aikataulussa, kun karsin muutamia alun perin suunnittelemiani ominaisuuksia ja ne jäivät odottamaan jatkokehitystä, josta kerron seuraavassa luvussa. Projektin eri osuuksien arviot ja toteutuneet työmäärät vaihtelivat aika reilusti laidasta toiseen, mikä oli oletettavaa, koska aihe on itselleni täysin uusi. Opinnäytetyön kirjoittaminen taas osaltaan venyi reilusti vuoden 2017 keväälle asti suunnitellun kesän 2016 sijaan. Viivästyminen tosin johtui osittain ulkoisista tekijöistä.

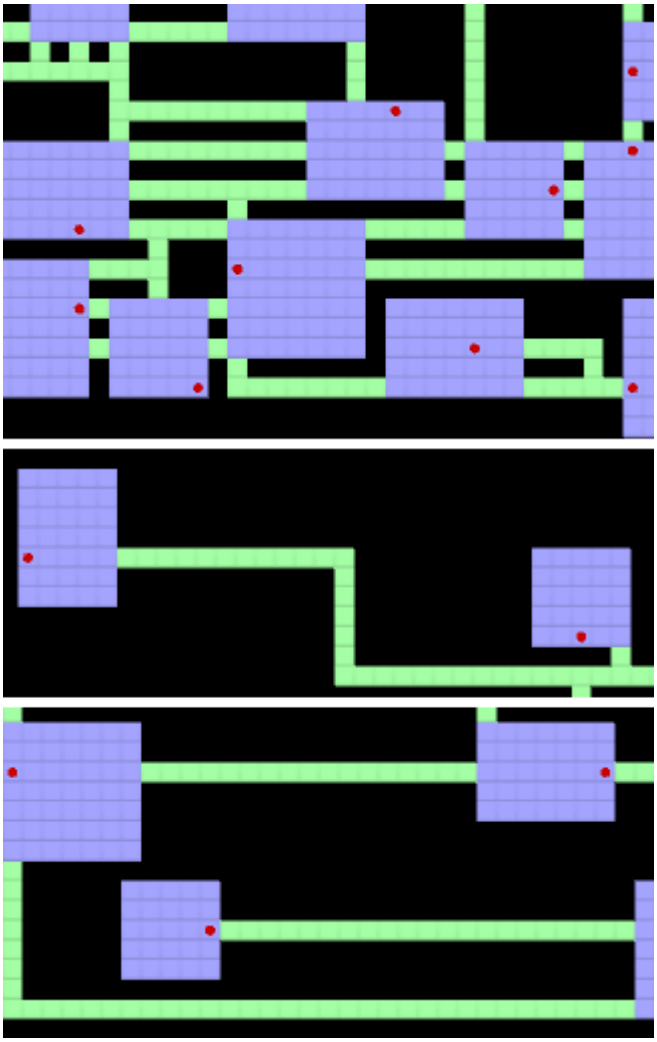
Projektin alussa tutustuminen siihen, miten ylipäätänsä sisältöä luodaan proseduraalisesti, vei arvellun kuukauden sijasta kaksi. Sitten taas arvioin ensimmäisen koodinpätkän luomisen tarvitsevan parin viikon verran aikaa, jolloin siitä olisi tullut muutama eri versio, mutta olinkin tyytyväinen heti ensimmäiseen versioon, joka valmistui yhdessä illassa. Laajemmalti projektin eri osioiden arvioiminen ajallisesti on lähes mahdotonta, sillä esimerkiksi koodin, joka luo käytävät, ehdin tehdä projektin aikana useamman kerran. Muutin koodin vähän jokaista osaa projektin kuluessa eikä sen yksittäisiä osia ja niiden tekemistä ja muuttamista oikein edes pysty erottelemaan.

6.1 Jatkokehitys

Projekti on pohja, jota voi lähteä jalostamaan myöhempää käyttöä varten jossakin pelissä, kuten oli tarkoituskin. Olisin halunnut saada määriteltyä vielä tarkempia ehtoja käytäville. Kuvassa 18 on esiteltynä useasti generoituvia käytäviä, jotka haluaisin karsia pois tai muokata mielekkäämmiksi pelattavuuden kannalta.

- Ylimmässä kohdassa käytäviä tulee pienelle alueelle liian tiheästi, välillä jopa kolme rinnakkain yhdistäen samat huoneet.
- Keskimmäisessä kohdassa on luolaston laidalle generoitunut yksittäinen huone, jonne johtaa suhteellisen pitkä käytävä, jota olisi pelissä melko tympeää juosta edestakaisin.

– Alimmassa kohdassa on kyseessä myös pitkä käytävä, joka tosin ei ole niin epämiellyttävä, koska se ei johda umpikujaan.



Kuva 18. Kolme tilannetta, jossa käytävät generoituivat ei-halutulla tavalla.

Käytävien hienosäätämisen lisäksi projektia voisi myös jatkaa lisäämällä erimuotoisia huoneita, mikä todennäköisesti vaatisi, että osa koodista kirjoitettaisiin uudelleen. Erimuotoiset huoneet toisivat heti huomattavasti mielenkiintoisemman lopputuloksen. Tätä hieman kokeilin nopeasti antamalla huoneiden ilmestyä toisten huoneiden päälle, mutta ongelmaksi koitui hyvin nopeasti sen hallitsemattomuus. Teoriassa huoneiden lisääminen vapaasti myös toistensa päälle olisi mahdollistanut jollakin satunnaisuudella luolaston, jossa kaikki huoneet ovat kiinni toisissaan. Muutenkin ilman rajoituksia huoneista tuli liian erikoisen muotoisia, mikä ei näyttänyt enää hyvältä. Nyt jälkikäteen pohdittuna olisin voinut tehdä käsin muutamia erilaisia huoneita ja käyttää muuten käyttämääni tekniikkaa, jossa tarkastettiin, onko huoneen alla jo aikaisempaa huonetta.

Koska luolasto tullaan generoimaan pelin latausvaiheessa, ei sen optimointi ollut erityisen tärkeä seikka. Tavoitteellisesti sen pitäisi kuitenkin olla suhteellisen tehokas, ettei pelin latausruutu tulisi kestämään liian pitkään. Mittasin projektin lopuksi, kuinka kauan yhden luolaston generoiminen kestää millisekunneissa. Käytin siihen .NET:stä löytyvää Stopwatch-luokkaa, ja

tulos oli luokkaa 100-120 millisekuntia. Tämä itseasiassa yllätti positiivisesti, sillä oletin sen olevan ainakin nelin- tai viisinkertainen. Kuitenkin ennen tämän projektin käyttämistä julkaistavassa pelissä tekisin muutamia nopeita ja yksinkertaisia optimointeja. Latausaikaan tulee pelissä kuitenkin myös useita muita muuttujia ja jokainen pienempikin optimointi nopeuttaa pelin latautumista.

6.2 Haasteet

Projektin alussa yritin käyttää hyväkseni vanhempien roguelike-pelien lähdekoodeja, joita on julkaistu kaikkien saataville. Niistä ei kuitenkaan ollut niin paljoa hyötyä kuin olisin voinut toivoa. Näiden vanhempien roguelike-pelien koodi on kirjoitettu ohjelmointikielillä, joita en ole opiskellut, niiden koodia ei ollut kommentoitu tarpeeksi kattavasti ja koodin rinnalle ei ollut tarjolla tekstiä, joka avaisi ohjelmoijan ajatusmaailmaa ja ideointiprosessia.

Kun puolivahingossa löysin Bob Nystromin blogin ”Rooms and Mazes: A Procedural Dungeon Generator”, pääsin melko nopeasti vauhtiin käyttämällä hänen kuvailemaa ideaa, jossa huoneet luodaan käyttämällä yrityskertoja. Hän myös käytti Dart-ohjelmointikieltä, joka oli syntaksiltaan todella lähellä käyttämääni C#-ohjelmointikieltä, mikä helpotti koodin tulkinnassa.

LÄHTEET

Amazon.com, Inc. FAQ. Viitattu 19.4.2016.
https://aws.amazon.com/lumberyard/faq/

Andgband. Viitattu 16.6.2016.
http://rephial.org/

Brodkin, J. 2013. How Unity3D Became a Game-Development Beast. Viitattu 20.4.2016.
http://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/

Crytek. CryENGINE 1. Viitattu 19.4.2016.
http://www.crytek.com/cryengine/cryengine1/overview

Crytek. CRYENGINE Programming. Viitattu 23.2.2017.
http://docs.cryengine.com/display/CEPROG/CRYENGINE+Programming

Crytek. 2016. Crytek Unveils All-New CRYENGINE V and Community-Centered “Pay What You Want” Model. Viitattu 19.4.2016.
http://www.crytek.com/news/crytek-unveils-all-new-cryengine-v-and-community-centered--pay-what-you-want--model

Crytek. 2014. Crytek announces its CRYENGINE-as-a-Service Program. Viitattu 19.4.2016.
https://www.cryengine.com/news/crytek-announces-its-cryengine-as-a-service-program

Crytek. Enterprice Licensing Solutions. Viitattu 19.4.2016.
https://www.cryengine.com/get-cryengine/full-license

Doull, A. 2008. The Death of the Level Designer: Procedural Content Generation in Games. Viitattu 27.3.2016.
http://roguelikedeveloper.blogspot.fi/2008/01/death-of-level-designer-procedural.html

Enger, M. 2013. Game Engines: How do they work? Viitattu 23.2.2017.
http://www.giantbomb.com/profile/michaelenger/blog/game-engines-how-do-they-work/101529/

Epic Games, Inc. Frequently asked questions (FAQ). Viitattu 19.4.2016.
https://www.unrealengine.com/faq

Epic Games, Inc. Programming Guide. Viitattu 23.2.2017.
https://docs.unrealengine.com/latest/INT/Programming/

Flanagan, D. 2011. JavaScript: The Definitive Guide. Sebastopol: O’Reilly Media, Inc.

Johnson, M. 2016. Generation Next, Part 1: How Games Can Benefit From Procedurally Generated Lore. Viitattu 23.2.2017.
https://www.rockpapershotgun.com/2016/07/22/future-of-procedural-generation-1/

Nutt, C. 2016. Amazon launches new, free, high-quality game engine: Lumberyard. Viitattu 19.4.2016.
http://www.gamasutra.com/view/news/265425/Amazon_launches_new_free_highquality_game_engine_Lumberyard.php

Nystrom, B. 2014. Rooms and Mazes: A Procedural Dungeon Generator. Viitattu 17.6.2016.
http://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/

Pluralsight. 2015. Unity, Source 2, Unreal Engine 4, or CryENGINE - Which Game Engine Should I Choose? Viitattu 23.2.2017.
https://www.pluralsight.com/blog/film-games/unity-udk-cryengine-game-engine-choose

Schreler, J. 2015. Sources: Amazon Spent Big Bucks On Crytek's Engine. Viitattu 19.4.2016.
http://kotaku.com/sources-amazon-spent-big-bucks-on-cryteks-engine-1696008878

Sweeney, T. 2015. If you love something, set it free. Viitattu 19.4.2016.
https://www.unrealengine.com/blog/ue4-is-free

Thomsen, M. 2010. History of the Unreal Engine. Viitattu 23.2.2017.
http://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine

Unity Technologies. 2014. Documentation, Unity scripting languages and you. Viitattu 29.4.2016.
http://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/

Unity Technologies. 2009. Unity 2.6 released and now free! Viitattu 26.4.2016.
https://unity3d.com/company/public-relations/news/unity2.6-press

Unity Technologies. n.d. Pricing. Viitattu 19.4.2016.
https://store.unity3d.com/products/pricing

Unity Technologies. n.d. Company Facts. Viitattu 30.3.2016.
https://unity3d.com/public-relations

Unity Technologies. n.d. Multiplatform. Viitattu 19.4.2016.
https://unity3d.com/unity/multiplatform

Wikidot. Procedural Content Generation. Viitattu 26.3.2016.
http://pcg.wikidot.com/

KUVALÄHTEET

Epic Games, Inc. Setting Up Character Movement in Blueprints. Viitattu 20.4.2016.
https://docs.unrealengine.com/latest/INT/Gameplay/HowTo/CharacterMovement/Blueprints/Setup_2/index.html

Nystrom, B. 2014. Rooms and Mazes: A Procedural Dungeon Generator. Viitattu 17.6.2016.
http://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/

Unity. Company Facts. Viitattu 30.3.2016.
https://unity3d.com/public-relations

Liite 1

CHECKCONNECTIONS-METODI

```
void CheckConnections()
{
    bool runAgain = false;
    regions.Add(new bool[dungeon.width, dungeon.height]);
    regionIndex.Add(new List<int>());

    Vector2Int pos = GetFirstTile();

    int loopCount = 0;
    List<Vector2Int> possibleDirs = new List<Vector2Int>();
    while (true)
    {
        //Prevent infinite loops (paranoia)
        if (loopCount++ > dungeon.width * dungeon.height) break;

        possibleDirs.Clear();

        if(floorCorridor[pos.x, pos.y])
        {
            solvedTiles[pos.x, pos.y] = true;
            regions[region][pos.x, pos.y] = true;

            if (floor[pos.x + 1, pos.y] && !solvedTiles[pos.x + 1,
pos.y])
                possibleDirs.Add(new Vector2Int(pos.x + 1, pos.y));
            if (floor[pos.x, pos.y + 1] && !solvedTiles[pos.x, pos.y
+ 1])
                possibleDirs.Add(new Vector2Int(pos.x, pos.y + 1));
            if (floor[pos.x - 1, pos.y] && !solvedTiles[pos.x - 1,
pos.y])
                possibleDirs.Add(new Vector2Int(pos.x - 1, pos.y));
            if (floor[pos.x, pos.y - 1] && !solvedTiles[pos.x, pos.y
- 1])
                possibleDirs.Add(new Vector2Int(pos.x, pos.y - 1));

            foreach(Vector2Int unsolvedDir in unsolvedDirs)
            {
                for(int i = 0; i < possibleDirs.Count; i++)
                {
                    if(unsolvedDir.x == possibleDirs[i].x && un-
solvedDir.y == possibleDirs[i].y)
                    {
                        possibleDirs.RemoveAt(i);
                    }
                }
            }

            if(possibleDirs.Count == 1)
            {
                pos = possibleDirs[0];
            }
            else if(possibleDirs.Count > 1)
            {
                pos = possibleDirs[0];
                possibleDirs.RemoveAt(0);
                unsolvedDirs.AddRange(possibleDirs);
            }
            else if(possibleDirs.Count == 0 && unsolvedDirs.Count >
0)
```

```
                {
                    pos = unsolvedDirs[0];
                    unsolvedDirs.RemoveAt(0);
                }
                else
                {
                    if (GetFirstTile().x > 0)
                        runAgain = true;
                    break;
                }
            }
            else
            {
                Rect curRoom = new Rect();

                for(int i = 0; i < rooms.Count; i++)
                {
                    if(rooms[i].Contains(new Vector2(pos.x, pos.y)))
                    {
                        curRoom = rooms[i];
                        regionIndex[region].Add(i);
                        break;
                    }
                }

                //Find and list all not visited corridors leaving this
room
                int xMin = Mathf.RoundToInt(curRoom.xMin);
                int xMax = Mathf.RoundToInt(curRoom.xMax);
                int yMin = Mathf.RoundToInt(curRoom.yMin);
                int yMax = Mathf.RoundToInt(curRoom.yMax);

                for (int x = xMin; x < xMax; x++)
                {
                    if (floorCorridor[x, yMin - 1] && !solvedTiles[x,
yMin - 1])
                        unsolvedDirs.Add(new Vector2Int(x, yMin - 1));
                    if (floorCorridor[x, yMax] && !solvedTiles[x, yMax])
                        unsolvedDirs.Add(new Vector2Int(x, yMax));
                }

                for (int y = yMin; y < yMax; y++)
                {
                    if (floorCorridor[xMin - 1, y] && !solvedTiles[xMin
- 1, y])
                        unsolvedDirs.Add(new Vector2Int(xMin - 1, y));
                    if (floorCorridor[xMax, y] && !solvedTiles[xMax, y])
                        unsolvedDirs.Add(new Vector2Int(xMax, y));
                }

                //Add all tiles in room to solvedTiles
                for(int x = xMin; x < xMax; x++)
                {
                    for(int y = yMin; y < yMax; y++)
                    {
                        solvedTiles[x, y] = true;
                        regions[region][x, y] = true;
                    }
                }

                if(unsolvedDirs.Count > 0)
                {
                    pos = unsolvedDirs[0];
                    unsolvedDirs.RemoveAt(0);
```

```
                    }
                    else
                    {
                        if (GetFirstTile().x > 0)
                            runAgain = true;
                        break;
                    }
                }
            }

            if(runAgain)
            {
                region++;
                CheckConnections();
            }
        }
```