



Osaamista  
ja oivallusta  
tulevaisuuden  
tekemiseen

Miika Laitinen

# Tasojen ja kenttien proseduraalinen generointi videopeleissä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

31.08.2020

Tekijä(t) Otsikko	Miika Laitinen Tasojen ja kenttien proseduraalinen generointi videopeleissä
Sivumäärä Aika	30 sivua 31.08.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander Lehtori Heini Puuska
<p>Proseduraalinen generointi tarkoittaa sisällön luomista ohjelmaan ajon aikana, tässä tapauksessa kenttä- tai tasovideopeliin. Proseduraalinen generointi toteutetaan ohjelmallisesti matemaattisia algoritmeja ja sääntöjä käyttäen.</p> <p>Tässä insinööriyössä tarkastellaan kolmea eri tapaa generoida proseduraalisesti kaksiulotteisia kenttiä videopeleihin. Työssä ei ole lähdetty rakentamaan kuitenkaan kokonaista peliä kenttien testausta varten vaan generoitua sisältöä tarkastellaan teoreettisesta näkökulmasta.</p> <p>Työssä käydään hieman proseduraalisen generoinnin historiaa videopelikenttien luonnissa ja pohditaan myöhemmin, mihin suuntaan proseduraalinen generointi on menossa videopeliteollisuudessa.</p> <p>Käytännön osuus sisältää kolmen erilaisen videopelikentän proseduraalisen generoinnin toteutuksen ohjelmallisesti käyttäen Godot-pelimoottoria. Jokainen tapa käydään läpi, miten generointi on toteutettu tekstipohjaisella selostuksella ja ohjelmakoodilla. Lopputuloksia käydään läpi teoreettisesti käyttäen apuna havainnollistavia kuvia.</p> <p>Lopputuloksena syntyi tutkielma, jossa nähdään tasojen ja kenttien proseduraalisen generoinnin hyötyjä ja haittoja sekä pohditaan parannusehdotuksia olemassa oleviin toteutuksiin.</p>	
Avainsanat	Videopeli, kenttä, taso, proseduraalinen, generointi

Author(s) Title	Miika Laitinen Procedural Level Generation in Videogames
Number of Pages Date	30 pages 31 August 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Simo Silander, Senior Lecturer Heini Puuska, Senior Lecturer
<p>Procedural generation refers to the creation of content while running a program, in this case a level in a video game. The procedural generation is implemented programmatically using mathematical algorithms and rules.</p> <p>This thesis explores three different ways to procedurally generate two-dimensional levels in video games. In this Thesis, we do not create a whole game for testing, but look at the content generated from a theoretical perspective.</p> <p>The paper goes into a bit of the history of procedural generation in video game level creation and later discusses the direction in which procedural generation is going in the video game industry.</p> <p>The practical part includes the implementation of the procedural generation of three different video game level generators programmatically using the Godot game engine. Each method goes through how the generation is implemented with text-based narration and program code. The results are discussed theoretically using illustrative pictures.</p> <p>As a result, a thesis was created that looks at the benefits and drawbacks of procedural generation of levels and fields and discusses suggestions for improvement to existing implementations.</p>	
Keywords	Videogame, level, prosedural, generation

## Sisällys

1	Johdanto	1
2	Tasojen ja kenttien proseduraalisen generoinnin historiaa	2
3	Godot-pelimoottori	3
3.1	Godotin ominaisuudet	3
3.2	Godotin käyttöliittymä	4
3.3	Godot ja tasojen proseduraalinen generointi	5
4	Proseduraalisen generoinnin toteutustapoja	6
4.1	Toimijapohjainen (Agent-based)	6
4.2	Soluautomaatti (Cellular automata)	7
4.3	Fysiikkamoottoriin pohjautuva toteutus	8
5	Kaksiulotteisen tason generointi toimijapohjaisella tekniikalla	8
5.1	Tasogeneroinnin toteutus Godotilla	8
5.2	Tasogeneroinnin tulokset	13
6	Kaksiulotteisen tason generointi soluautomaatti tekniikalla	14
6.1	Tasogeneroinnin toteutus Godotilla	14
6.2	Tasogeneroinnin tulokset	21
7	Kaksiulotteisen tason generointi fysiikkamoottoritekniikalla	22
7.1	Tasogeneroinnin toteutus Godotilla	22
7.2	Tasogeneroinnin tulokset	27
8	Yhteenveto	28
	Lähteet	30

## 1 Johdanto

Tässä opinnäytetyössä on tarkoituksena tutkia erilaisia tapoja luoda videopelisiin kaksiulotteisia kenttiä generoimalla ne käyttäen hyväksi erilaisia matemaattisia algoritmeja. Työssä esitellään muutama eri toteutustapa ja ohjelmoidaan ne Godot-pelimootorilla sekä tutkitaan algoritmien tuloksia.

Tasojen proseduraalisella generoinnilla tarkoitetaan tapaa, jolla videopelisiin pystytään tietokoneen suorittamilla algoritmeilla luomaan erilaisia kenttiä ilman manuaalista ihmisen tekemää työtä. Proseduraalisen generoinnin etuina ovat satunnaisuus, pienemmät tiedostokoot ja pelin suurempi sisältömäärä. Generointi toteutetaan ohjelmoimalla peliin ajonaikainen matemaattinen algoritmi, joka tiettyjen sääntöjen perusteella toteuttaa pelikentän. [1.]

Proseduraalisen tasojen generoimisen hyödyistä tulee heti ensimmäisenä vastaan satunnaisuus. On luonnollista, että kenttä, jota pelaaja ei ole aikaisemmin nähnyt, on mielenkiintoisempaa pelattavaa kuin sellainen, joka on jo aikaisemmin tuttu eli uudelleenpelattavuus kasvaa. Tasojen generointi on myös edullisempaa tietokoneen levytilaa ajatellen kuin manuaalisesti kenttien rakentaminen. Tästä hyvänä esimerkkinä on vuonna 1984 julkaistu videopeli Elite, joka kiersi silloisten tietokoneiden tiukkoja muistirajoituksia proseduraalisella generoinnilla. [2.]

Proseduraalisella generoinnilla on toki myös varjopuolensa. Satunnaisuus itsessään saattaa ärsyttää pelaajaa, ja kenttien generointialgoritmi toimii juuri niin hyvin kuin se on peliin itseensä ohjelmoitu. Proseduraalisella generoinnilla tehtyjä kenttiä täytyy ohjelmoijan myös testata paljon ja ottaa rajatapaukset huomioon. Esimerkiksi yksi ohjelmointiongelmaksi on sellainen, että pelaaja näkee jonkin paikan kentästä, minne on mahdoton päästä, koska ohjelma ei luonut väylää sinne. Yksi mahdollinen haitta on kosmeettisuuteen liittyvä. Koska kenttien luonnin hoitaa tietokone algoritmin avulla, eikä koneella ole ihmisen visuaalista silmää, minkä pitäisi näyttää luonnolliselta pelaajalle, saattaa kaikki näyttää kentässä koneen tuottamalta. Tätä ongelmaa vastaan on olemassa kuitenkin erilaisia työkaluja, joita tässä työssä tullaan myöhemmin esittelemään. [2.]

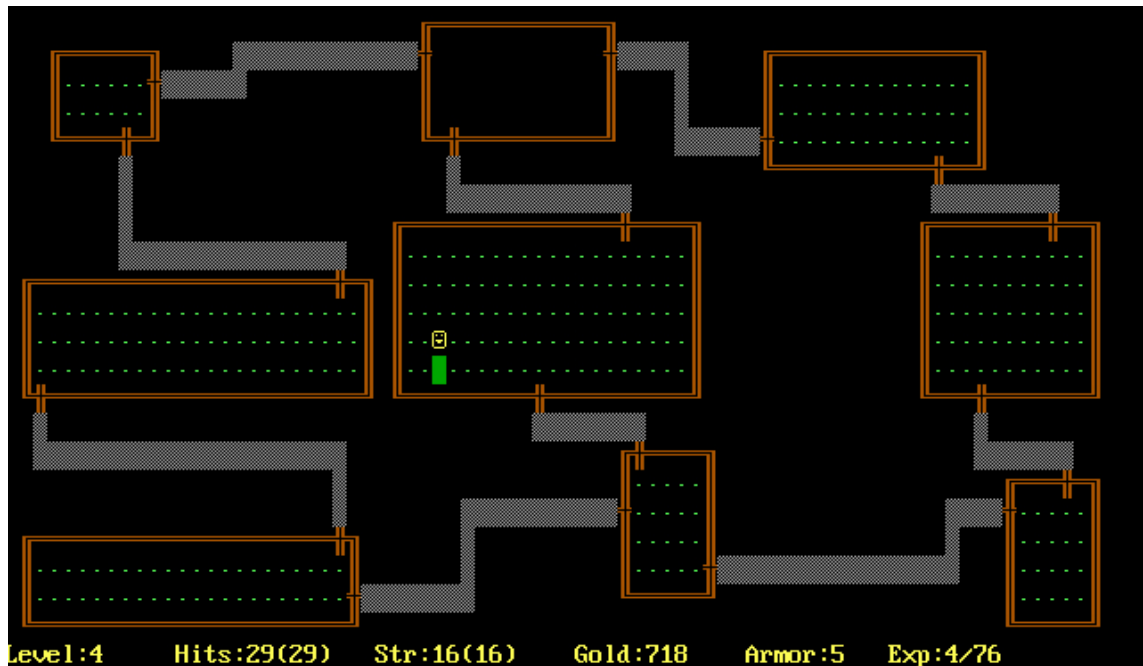
Tässä opinnäytetyössä tulee olemaan huomattava määrä kuvia, mikä johtuu aiheen vaikeasta kuvaamisesta tekstillä. Tulokset tullaan esittelemään aina yhdellä tai useammalla kuvalla ja pohtivalla tekstillä.

## 2 Tasojen ja kenttien proseduraalisen generoinnin historiaa

Tasojen proseduraalinen generointi juontaa juurensa 1978 julkaistuun videopeliin nimeltä Beneath Apple Manor. Huomattavasti tunnetumpi ja uuden videopeligenren synnyttänyt Rogue, joka julkaistiin 1980, käytti hyväkseen myös proseduraalista generointia. Itse muistan parhaiten ensimmäistä kertaa törmänneeni tasojen proseduraaliseen generointiin videopelissä Diablo (1997). Pelissä on tarkoitus tutkia luolastoja, jotka ovat täynnä hirviöitä. Oli hämmästyttävää nähdä, kuinka jokaisella pelikerralla kentät olivat erilaisia. Tuolloin myös ensimmäisen kerran huomasin, kuinka proseduraalisessa generoinnissa on omat huonot puolensa, kuten kenttien konemainen ulkoasu. [3; 4; 5.]

Hyvänä ääriesimerkkinä proseduraalisessa generoinnissa voidaan nähdä avaruustutkimuspeli No Man's Sky (2016), johon pelin kehittäjät lupasivat 18 triljoonaa uniikkia tutkittavaa planeettaa. Numeerisesti tuo määrä toteutui, mutta asettien rajallinen määrä aiheutti sen, että lähes jokainen planeetta näytti samanlaiselta eikä niiden tutkiminen ollut yhtään niin mielenkiintoista, kuin kehittäjät antoivat olettaa. Pelin vastaanotto oli jopa osittain vihamielistä pelaajien keskuudessa, ja vaikka pelin teknistä toteutusta ylistettiin, sai sen heikoin lenkki, eli tasojen proseduraalinen generointi täystyrmäyksen. [6.]

Tasojen proseduraalista generointia on käytetty videopeleissä jo kymmeniä vuosia, ja tekniikan kehittyessä on yhä monimutkaisemmat algoritmit olleet mahdollisia toteuttaa. Myös tietokoneiden tehot ovat kasvaneet, jolloin pelien latausajat ovat pysyneet maltillisina.



Kuva 1. Kuvakaappaus videopelistä Rogue [4.]

### 3 Godot-pelimoottori

Godot on Juan Linietskyn ja Ariel Manzurin kehittämä ilmainen vapaan lähdekoodin pelimoottori, joka toimii Windows-, macOS- ja Linux-alustoilla. Pelejä Godotilla voi kehittää PC-, mobiili- sekä web-alustoille. Godot pyrkii olemaan kaiken kattava pelimoottori, jolla kehittäjä voi ohjelmoida videopelin alusta loppuun saakka vain käyttämällä Godotin kehitysympäristöä. Ainoastaan erilaisten assettien (grafiikat, äänet) tekeminen täytyy hoitaa muilla työkaluilla. [7.]

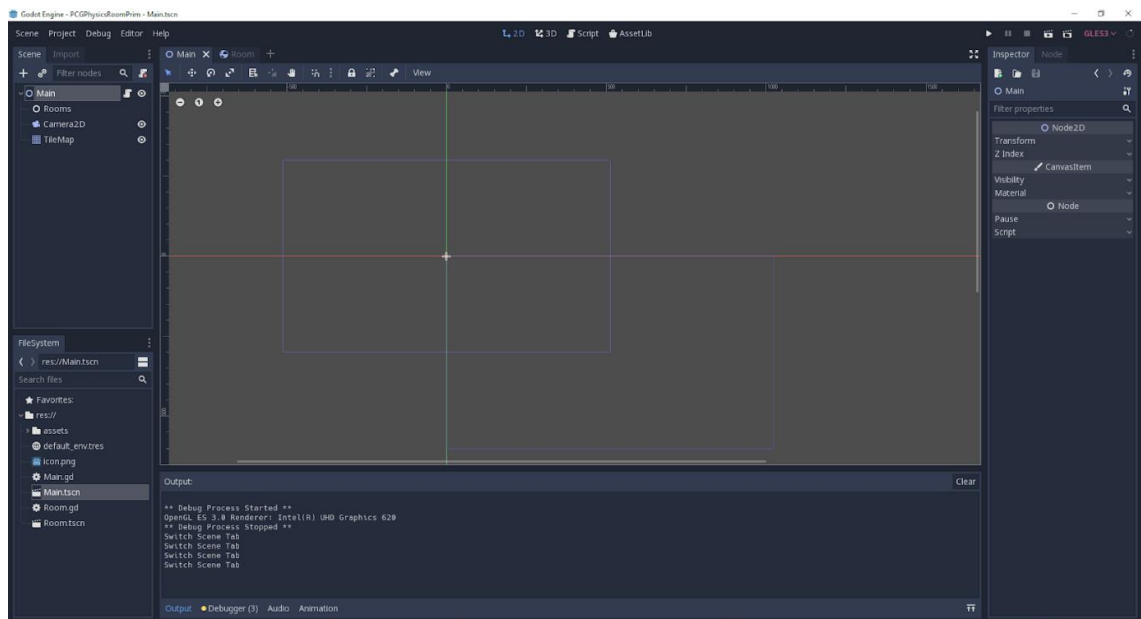
#### 3.1 Godotin ominaisuudet

Godotin ohjelmointikielenä toimii joko C++, C# tai Godotin oma skriptauskieli GDScript. GDScript on korkean tason ohjelmointikieli, joka muistuttaa paljon Python-ohjelmointikieltä. Pelimoottorin kehittäjät aluksi ajattelivat suoraan käyttävänsä jotain kolmannen osapuolen skriptikieltä, mutta tulivat siihen tulokseen, että oma kustomoitu skriptikieli toimisi parhaiten. Pelimoottori sisältää skriptieditorin automaattisella sisennyksellä, syntaksikorotuksella ja koodin täydennyksellä sekä virheidenjäljittäjän, joka pystyy käyttämään pysäytyspisteitä ja ohjelma-askellusta. Grafiikkamoottorina

Godot käyttää OpenGL ES 3.0:aa ja tulevaisuudessa pelimoottoriin tulee mahdollisesti tuki Vulkan grafiikkarajapinnalle. Pelejä pystyy kehittämään niin kaksi- kuin kolmiulotteisena. Godotissa on myös sisäänrakennettu animaatiojärjestelmä ja fysiikkamoottorina siinä toimii Bullet. [7.]

### 3.2 Godotin käyttöliittymä

Godotin käyttöliittymä näyttää melko yksinkertaiselta. Näkymässä on tähdätty minimalistisuuteen ja lisävalikot aukeavat vasta, kun jokin tietty kohta on valittuna. Vasemmalla näkyvät Scene-välilehti eli näkymä ja sen alla tiedostojärjestelmä. Keskellä näkymä näytetään sellaisena kuin se kehittäjälle näkyy eli esimerkiksi kenttäeditorina. Oikealla Inspector-välilehti näyttää valittujen objektien ominaisuuksia. Alhaalla näkyy output-kenttä, johon saa esimerkiksi virheenjäljittäjän viestit. Oikealla yläkulmassa näkyy pienellä Play-nappi, josta ohjelman suorituksen saa aloitettua.



Kuva 2. Godotin käyttöliittymä



Ohjelmoinnin kannalta mielenkiintoisin on skriptinäkö, jossa ohjelmointi tapahtuu. Näkö tulee näppärästi keskelle ruutua Scene-ruudun tilalle eikä erillisiä ohjelmia tarvitse aukaista.

```

16 ~ func _ready(): #PART1
17 ~     randomize() #PART1
18 ~     make_rooms() #PART1
19 ~
20 ~ func make_rooms(): #PART1
21 ~     for i in range(num_rooms): #PART1
22 ~         var pos = Vector2(rand_range(-hspread, hspread), 0) #PART1
23 ~         var r = Room.instance() #PART1
24 ~         var w = min_size + randi() % (max_size - min_size) #PART1
25 ~         var h = min_size + randi() % (max_size - min_size) #PART1
26 ~         r.make_room(pos, Vector2(w, h) * tile_size) #PART1
27 ~         $Rooms.add_child(r) #PART1
28 ~         #wait for movement to stop
29 ~         yield(get_tree().create_timer(1.1), 'timeout') #PART1
30 ~         #cull rooms
31 ~         var room_positions = [] #PART2
32 ~         for room in $Rooms.get_children(): #PART1
33 ~             if randf() < cull: #PART1
34 ~                 room.queue_free() #PART1
35 ~             else: #PART1
36 ~                 room.mode = RigidBody2D.MODE_STATIC #PART1
37 ~                 room_positions.append(Vector3(room.position.x, room.position.y, 0)) #PART2
38 ~
39 ~         yield(get_tree(), 'idle_frame') #PART2
40 ~         # generate a minimum spanning tree connecting the rooms
41 ~         path = find_mst(room_positions) #PART2
42 ~
43 ~ func _draw(): #PART1
44 ~     for room in $Rooms.get_children(): #PART1

```

Kuva 3. Godotin skriptieditori

### 3.3 Godot ja tasojen proseduraalinen generointi

Godot-pelimoottorissa ei ole itsessään sisäänrakennettua tasojen generointimallia eikä Godotin assettikirjaston internetsivuilta sellaista löydy vielä. Uskoisin tämän johtuvan siitä, että Godot on huomattavasti uudempi pelimoottori kuin esimerkiksi Unity, jonka assettikaupasta löytyy valmis proseduraalinen tasojen generointikehys. Godotiin on kuitenkin olemassa paljon ohjelmointikursseja, joilla pääsee helposti alkuun.

## 4 Proseduraalisen generoinnin toteutustapoja

### 4.1 Toimijapohjainen (Agent-based)

Toimijapohjainen kenttien toteutus tapahtuu käyttämällä yhtä tai useaa toimijaa, joka liikkuu kaksiulotteisella taustalla satunnaisesti suuntiin ja piirtää perässään polkua, joka määrää kentän liikkumisrajat. Toimija myös luo huoneita satunnaisesti eri kohtiin kenttää. Tämä toteutus saattaa luoda melko kaottisia kokonaisuuksia, ja huoneet saattavat generoitua myös päällekkäin. Hyvänä puolena toimijapohjaisessa toteutuksessa on, ettei kenttään synny ongelmaa, jossa pelaaja näkee jonkin alueen, mihin tietokone ei ole generoinut väylää, koska pelaaja pystyy liikkumaan juuri sinne, minne toimijakin liikkui generoinnin aikana. Toteutuksen pystyy myös kääntämään ympäri, jos haluaa sokkelomaisia kenttiä. Eli toimija luo seinät polun sijaan. Tällöin tuki pitää huomioida algoritmissa, ettei toimija luo mielivaltaisesti seiniä, jolloin saattaisi syntyä alueita, joihin pelaajalla ei ole pääsyä. [2.]



Kuva 4. Toimijapohjaisen toteutuksen idea. Punainen piste on toimija [2.]

## 4.2 Soluautomaatti (Cellular automata)

Soluautomaatti käyttää kentän generoinnissa ruudukkoa, joka täytetään satunnaisesti joko läpäisemättömällä seinällä tai tausta elementillä, jonka läpi pelaajahahmo voi liikkua. Tämän jälkeen päätetään säännöt, joilla solut käyttäytyvät naapurisoluihin nähden. Generointi tapahtuu askelissa, joita yleensä käydään monia läpi ennen lopullista tulosta. Soluautomaatti toteutus kentän generoinnissa luo luolamaisia kokonaisuuksia mitkä ovat toimijapohjaiseen toteutukseen nähden hyvin erilaisia. Ongelmana soluautomaatissa on se, että kentät ovat yleensä hyvin avoimia ja pelaaja pääsee melko helposti liikkumaan aloituspisteestään maaliin. Hyvänä puolena voidaan pitää kentän luonnollista ulkonäköä. [2.]

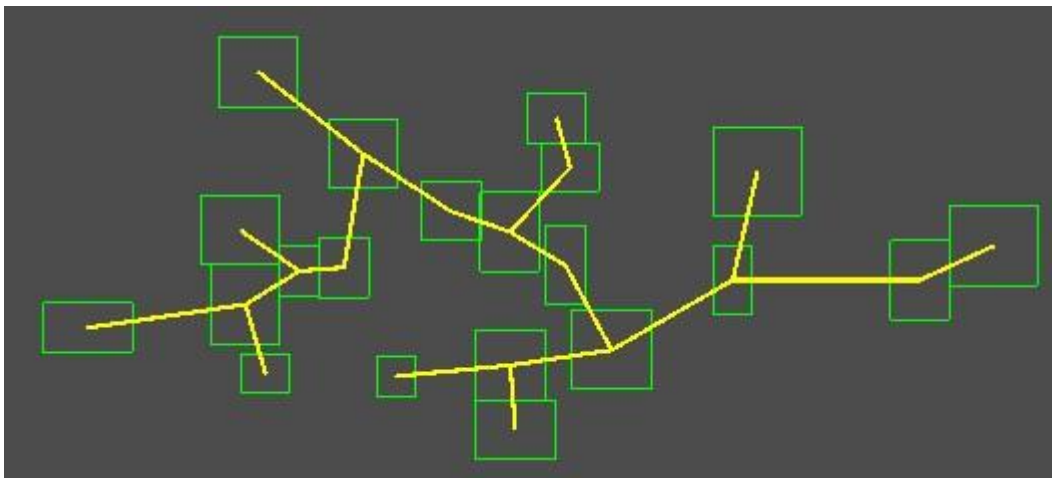
Soluautomaatti toteutukseen pystyy lisäämään käytäviä kaivavan toteutuksen, jolloin saadaan enemmän huoneista ja käytävistä koostuva toteutus. Tämä vaatii kuitenkin ohjelman koodiin lisäalgoritmin, jolloin toteutus monimutkaistuu.



Kuva 5. Soluautomaattipohjaisen toteutuksen idea. Ensimmäinen ruutu on satunnaisen generoinnin ensimmäinen vaihe [2.]

### 4.3 Fysiikkamoottoriin pohjautuva toteutus

Godotin fysiikkamoottoriin pohjautuva toteutus toimii hyväksikäyttämällä Godotin sisäänrakennettua Bullet-nimistä fysiikkamoottoria. Pelikenttä luodaan generoimalla huoneita päällekkäin kentän keskelle ja antamalla niille törmäysmallinnus, jolloin huoneet eivät pysty generoitumaan päällekkäin vaan leviävät eri suuntiin. Tämän jälkeen osa huoneista poistetaan ja sen jälkeen lisätään käytävät huoneiden välille, jotta pelaaja pystyy liikkumaan huoneesta toiseen. Jotta käytävät voidaan tehdä huoneiden välille, tarvitaan tähän Primin (Prim's algorithm) algoritmi, joka kykenee löytämään huoneiden välille reitin, jolle käytävä generoidaan. [8.]



Kuva 6. Toteutuksen idea. Primin algoritmi näkyy keltaisena viivana

## 5 Kaksiulotteisen tason generointi toimijapohjaisella tekniikalla

### 5.1 Tasogeneroinnin toteutus Godotilla

Toimijapohjainen tason generointi on esitellyistä tekniikoista yksinkertaisin ja vähiten ohjelmointia vaativa. Periaatteena on ohjelmoida siis toimija, joka kulkee kaksiulotteisella solutasolla ruutu kerrallaan satunnaiseen suuntaan ja kaivaa tietä jättäen jälkeensä polun, jota pelaaja pystyy kulkemaan. Toimija osaa myös umpikujaan ajautuessaan kulkea takaisin askelia ja etsiä seuraavan solun, jota toimija ei ole vielä kulkenut. Määritellyn kokoisen tason kaikki solut kuljettuaan toimija lopettaa toimintansa. Aikaisemmin työssä esitelty tekniikka, jossa toimija tekee huoneita, päätettiin toteuttaa tässä

yksinkertaisemmin, koska myöhemmin toisella tekniikalla toteutetussa tason generoinnissa luodaan huoneita ja käytäviä. Nyt ohjelmoitu toimija siis luo sokkelomaisen kentän, joka on käytännössä vain yksi käytävä ilman huoneita. [9.]

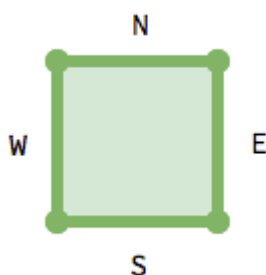
Rekursiivisen takaisin palaavan algoritmin toimintaperiaate:

1. Poimi aloitus solu ja merkitse se vierailtuksi.
2. Jos missään naapuri solussa ei ole vierailtu:
  - 2.1 Valitse satunnaisesti jokin vierailematon naapuri solu.
  - 2.2 Poista seinä solujen väliltä.
  - 2.3 Lisää tämänhetkinen solu pinnoon.
  - 2.4 Tee valitusta solusta nykyinen solu ja merkkää se vierailtuksi.
3. Jos nykyisellä solulla ei ole vierailemattomia soluja, ota ylin solu pinosta ja tee siitä nykyinen solu.
4. Toista kohdasta kaksi (2.), kunnes vierailemattomia soluja ei enää ole.

```
func check_neighbors(cell, unvisited):
    var list = []
    for n in cell_walls.keys():
        if cell + n in unvisited:
            list.append(cell + n)
    return list
```

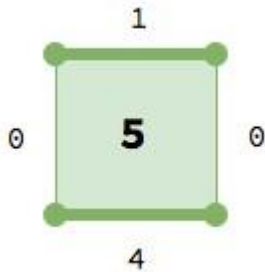
Koodiesimerkki 1. Vierailtun solun tarkistus koodissa [9.]

Seuraavaksi on tarkoitus miettiä, miten sokkelon data esitetään. Tässä käytämme Godotin TileMap nimistä nodea(solmua), joka perustuu ruudukoihin. Jokaisella solulla on neljä seinää, jotka voivat olla joko auki tai kiinni. Näitä kahta tilaa kuvaamaan voidaan käyttää yhtä bittiä, 0 kun seinä on auki ja 1 kun se on kiinni. Tätä tapaa käyttäen voidaan kaikkia solujen seinätiloja kuvata neljällä bitillä. [9.]



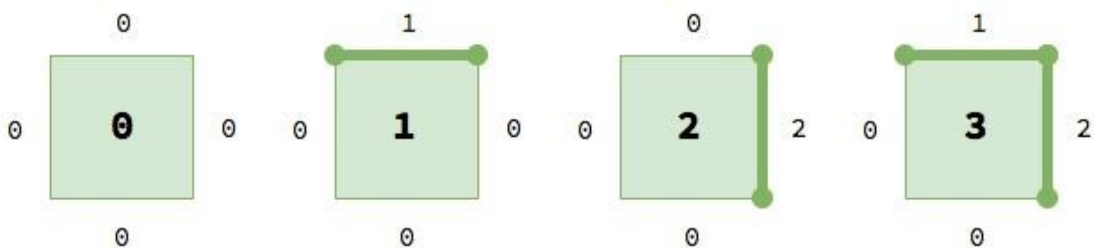
Kuva 7. Havainnollistava kuva solun seinistä [9.]

Ohjelmassa käytetään järjestystä WSEN, jossa W tarkoittaa länttä, S etelää, E itää ja N pohjoista. Nyt solujen seinämiä voidaan esittää biteillä. Esimerkiksi 0101 tarkoittaa, että solun etelä ja pohjoisseinät ovat kiinni ja itä sekä länsi ovat auki. [9.]



Kuva 8. Havainnollistava kuva seinien bittimuotoisesta esittämisestä [9.]

Nyt voimme kuvata kaikkia soluja numeroilla 0-15 eli jokaisella solulla on tunnistenumero. Tässä tavassa esittää solut on se etu, että voimme käyttää TileMap nodea(solmua) pitämään huolta seinien tiloista tunnistenumeron avulla eikä erillistä tietorakennetta tarvita tallentamaan seinien tiloja. [9.]



Kuva 9. Keskellä solua on sen tunnistenumero, ja seinien tilat esitellään bittimuodossa [9.]



Kuva 10. Ohjelmassa käytetyt ruudukot

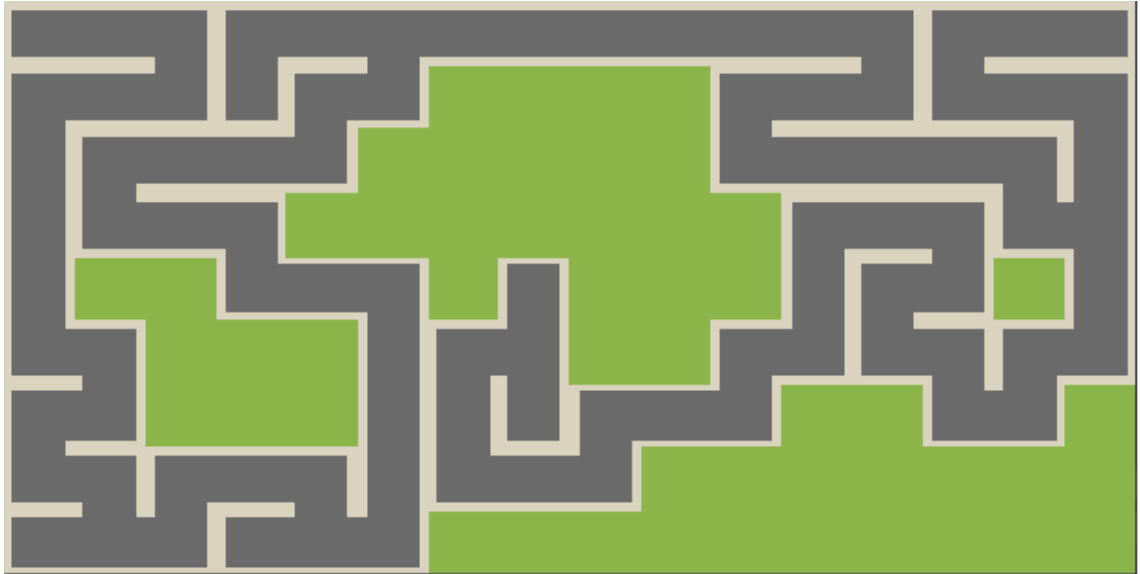
```

func make_maze():
    var unvisited = []
    var stack = []
    Map.clear()
    for x in range(width):
        for y in range(height):
            unvisited.append(Vector2(x, y))
            Map.set_cellv(Vector2(x, y), N|E|S|W)
    var current = Vector2(0, 0)
    unvisited.erase(current)
    while unvisited:
        var neighbors = check_neighbors(current, unvisited)
        if neighbors.size() > 0:
            var next = neighbors[randi() % neighbors.size()]
            stack.append(current)
            var dir = next - current
            var current_walls = Map.get_cellv(current) - cell_walls[dir]
            var next_walls = Map.get_cellv(next) - cell_walls[-dir]
            Map.set_cellv(current, current_walls)
            Map.set_cellv(next, next_walls)
            current = next
            unvisited.erase(current)
        elif stack:
            current = stack.pop_back()
    yield(get_tree(), 'idle_frame')

```

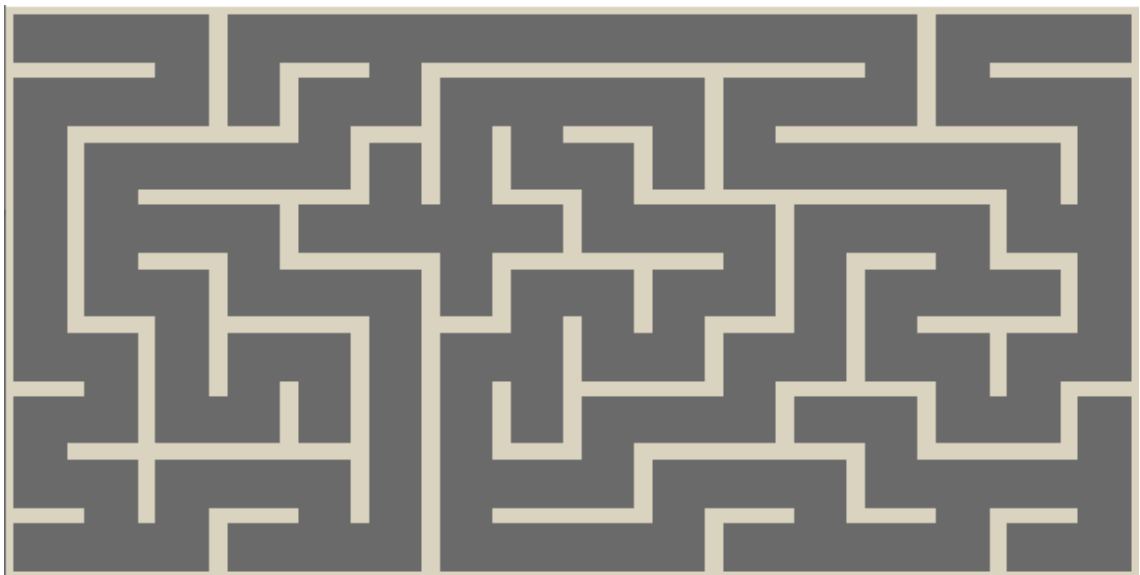
Koodiesimerkki 2. Sokkelon luontifunktio [9.]

Sokkelon luonti funktio perustuu rekursiiviseen takaisin askeltavaan algoritmiin. Funktiossa käytetään hyväksi jo aikaisemmin työssä esiteltyä `check_neighbors`-funktioita, joka tarkastaa solun naapurien vierailtu-tilan. Koodin viimeinen rivi ei ole osa logiikkaa vaan on tarkoitettu viivyttämään ruudunpäivitystä, jolloin ohjelmoija voi itse nähdä, kuinka sokkelo rakentuu solu kerrallaan. Ilman tätä riviä sokkelo vain ilmestyy suoraan kuvaruudulle, kun ohjelma suoritetaan Godotissa.



Kuva 11. Toimija työssään generoimassa sokkeloa

Kun ohjelman suorittaa, alkaa se toimijan avulla generoimaan sokkeloa. Tason generoinnin valmistuttua toimijan algoritmin suoritus lopetetaan, ja valmis sokkelo jää kuvaruudulle.

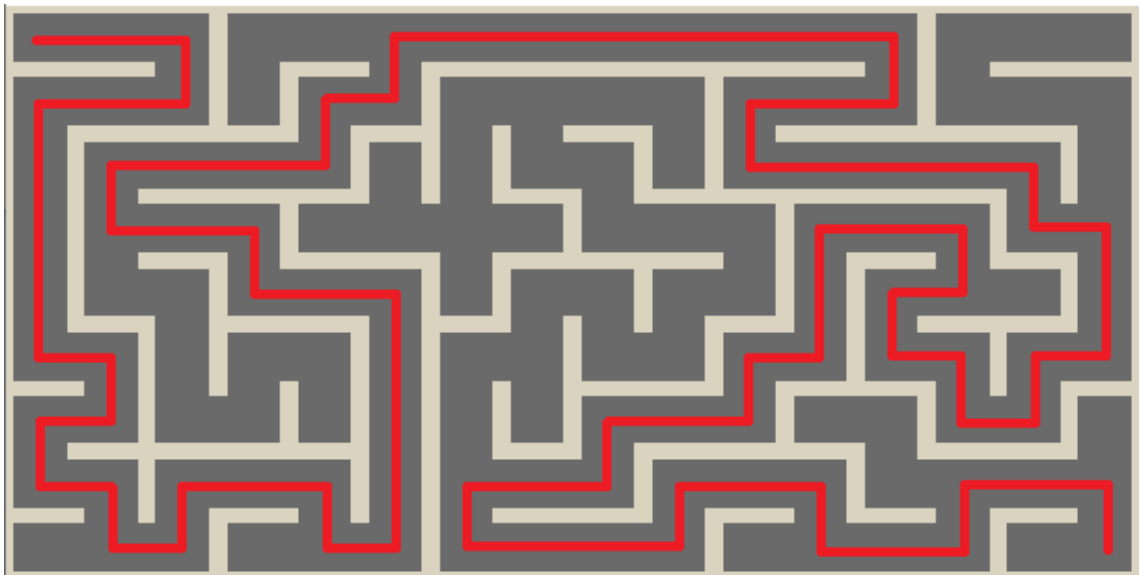


Kuva 12. Valmis sokkelo



## 5.2 Tasogeneroinnin tulokset

Kuten kuvasta 13, näemme saa tasogeneraattori melko pätevän näköisen sokkelon aikaan. Hyvänä asiana generoidussa tasossa voidaan pitää sitä, että sokkelossa ei ole paikkoja, mihin pelaaja ei pääsisi, vaan kaikki käytävät ovat pelaajan tutkittavissa. Kannattaa kuitenkin huomata, että generoitu sokkelo on melko pieni kuitenkin kooltaan, jos määrittelemme, että pelaajahahmo olisi yhden solun kokoinen. Algoritmiin ei ole lisätty sellaista osaa, joka määritteli, missä pelaajahahmon aloitus- ja päätepiste sijaitsevat, mutta sen toteutus ei olisi vaikeaa. Olisi luonnollista tehdä nämä pisteet mahdollisimman kauas toisistaan, esimerkiksi aloitus voisi olla vasemmassa yläreunassa ja päätepiste oikeassa alareunassa.



Kuva 13. Sokkelon ratkaisu. Aloituspiste on vasemmassa ylänurkassa ja päätepiste oikeassa alnurkassa

Kun katsomme sokkelon ratkaisua, näyttää siltä, että tasogenerointi onnistui melko hyvin. Sokkelon ratkaisu ei ole liian pitkä, mutta se ei ole myöskään liian lyhyt. Huonona puolena sokkelosta voisi kuitenkin sanoa, että keskellä kenttää on melko pitkä polku, joka johtaa pelaajan umpikujaan. Umpikujaa pystyisi kuitenkin mahdollisesti käyttämään hyödyksi lisäämällä sinne esimerkiksi jonkin esineen, joka auttaisi pelaajaa, jolloin pelkän umpikujan löytäminen ei turhauttaisi pelaajaa. Oletamme siis, että pelaajan näkökenttää rajoitetaan jotenkin, ettei pelaaja itse tiedä olevansa menossa kohti umpikujaa. Toinen mahdollisuus olisi muokata päätepiste sijaitsemaan keskellä kenttää sijaitsevaan umpikujaan, jolloin kentän ratkaisun pituus ei hirveästi lyhenisi, ja se voisi

olla hyvä harhautus, jos pelaaja olettaisi, että koska aloituspiste on vasemmassa ylänurkassa, on päätepiste suurella todennäköisyydellä oikeassa alanurkassa.

Kokeilin myös luoda useamman kentän peräkkäin ja tutkin, minkälaisia sokkeloita algoritmi saa aikaiseksi. Joskus algoritmi loi todella lyhyitä ratkaisuja, jos pysyisimme edellisen aloitus- ja päätepisteen määritelmässä. Tällöin syntyi myös erittäin pitkiä käytäviä, joista monet johtivat umpikujaan. Algoritmi loi kuitenkin satunnaisesti erittäin hyvän pituisia kenttiä. Jos kentän kokoa lähtee muuttamaan isommaksi alkaa yksi iso ongelma nostaa päätään. Sokkelot muuttuvat erittäin vaikeiksi navigoida, ja tällainen on harvoin suotavaa. Sokkelot kannattaa siis pitää sopivan kokoisina. Vaihtoehtoisesti generointialgoritmia voisi parantaa antamalla sille säännöksi, että sokkelon ratkaisu täytyy löytyä vaikkapa tietyn askelmäärän jälkeen. Tällöin voisimme luopua määritelmästä, että aloitus ja päätepiste on aina kentän jossain nurkassa. Yksi parannusmahdollisuus voisi olla sellainen, että tehdään aloitus- ja päätepisteen välille enemmän kuin yksi ratkaisu. Tällä hetkellä algoritmi generoi vain ja ainoastaan yhden ratkaisun sokkelolle.

Suoraan tällaisenaan käytettävänä kentän generointialgoritmi ei ehkä olisi kauhean hyvä vaan sitä pitäisi hioa erilaisilla säännöillä ja testata vielä oikean pelin avulla. Pohjana algoritmi on kuitenkin kohtalaisen toimiva ja siihen saa lisättyä omia sääntöjä melko vaivatta, koska koodi ei ole hirveän monimutkaista eikä sitä ole valtavia määriä.

## **6 Kaksiulotteisen tason generointi soluautomaatti tekniikalla**

### **6.1 Tasogeneroinnin toteutus Godotilla**

Soluautomaattipohjainen kentän generointi on hieman monimutkaisempi toteuttaa. Periaatteena tekniikassa on käyttää soluautomaattimallia, jonka tutkiminen on lähtenyt 1940-luvulla jo liikkeelle. Seuraavassa luvussa käydään hieman läpi soluautomaatin toimintaperiaatetta.

Soluautomaatin ajatus on se, että se koostuu monista soluista, jotka vaihtavat tilaansa valittujen sääntöjen mukaan. Ne ovat pistetty useimmiten säännöllisen muotoiseen hilaan, joka on monesti 2-ulotteinen suorakulmainen ruudukko. Usein solun tilanvaihdos riippuu naapureista, mutta se voi riippua myös solun menneisyydestäkin. Soluautomaateissa syntyy kohtalaisen mutkikkaita, usein liikkuvia, muuttuvia ja itseään toistavia kuvioita.

Esimerkiksi Conwayn Life-peli eli "elämän peli" muistuttaa itseään pelaavaa jätkänshakkia. Peli on nollan pelaajan peli, jota tavallisimmin tietokone pelaa itseksensä. Life-pelin pohjana on tavallisesti suorakulmainen ruudukko, hila, jonka ruudut ovat soluja, jotka voivat olla joko eläviä tai kuolleita.

Monesti elävää solua merkitään mustalla, kuollutta valkealla. Värit valitaan usein siten, että näytön tai tietokonetulosteen normaalia taustaväriä lähellä oleva väri merkitsee kuollutta solua.

Soluautomaattia jäljittelevälle systeemille voidaan syöttää alussa vaikkapa satunnaista kohinaa, jossa kuolleita ja eläviä soluja on vierä vieressä satunnaisesti arvottuna. Silloin lähtökohta on mustavalkea sekamelska.

Life-pelissä oleva solu muuttuu ajan mukana seuraavasti:

Elävä solu, jolla on alle 2 elävää naapuria, kuolee yksinäisyyteen

Elävä solu, jolla on yli 3 naapuria, kuolee liikakansoitukseen

Elävä solu, jolla on 2-3 naapuria, säilyy hengissä

Kuollut solu, jolla on tarkoin kolme elävää solua, alkaa elää.

Tällöin sanotaan, että Conwayn Lifellä on 23/3 sääntö. 23 tarkoittaa, että 2 tai 3 naapurisolua pitää solun hengissä, ja 3 sitä, että 3 naapurisolua synnyttää uuden solun. Kun tietokone pelaa Life-peliä, mustavalkea sekamelska häviää, ja näytölle ilmestyy monenlaisia ja monen kokoisia ruuduista koostuvia, ajan mukana muuttuvia kuvioita. Life-pelin tyyppisiä soluautomaatteja on laadittu monia. Melko monimutkainen 012345678/3-sääntö tuottaa tikapuumaisia, virtapiiriä muistuttavia kuvioita. [10.]

Soluautomaattikentän generoinnissa käydään läpi monta solujen sukupolvia, jolloin saadaan muodostettua luolamainen kenttä. Luolasta saa hieman erilaisen muuttamalla ohjelmassa olevia muuttujia eri arvoihin. Tässä työssä käytetty toteutus on ehkä aavistuksen hienompi kuin perinteinen soluautomaattiratkaisu, sillä ohjelma osaa myös kaivaa tunneleita luolastoon, jos muuttajat määritetään sillä tavalla, ettei yhtä isoa luolastoa synny, vaan luolasto jakautuu pienempiin huoneen tapaisiin osiin. Jos tunnelin kaivausta ei toteutettaisi, saattaisi käydä niin, että pelaaja näkee huoneita, mihin ei kuitenkaan ole pääsyä.

```
func generate():
    clear()
    fill_roof()
    random_ground()
    dig_caves()
    get_caves()
    connect_caves()
```

Koodiesimerkki 3. Käytetyt funktiot luolan generoinnissa [11.]

- Clear-funktio putsaa kentän.
- Fill\_roof-funktio täyttää kentän läpäisemättömillä seinillä.
- Random\_ground-funktio asettaa satunnaisesti läpäistäviä soluja kentälle.
- Dig\_caves-funktio kaivaa luolastoja perustuen solujen naapureihin.
- Get\_caves-funktio ottaa luolaston huoneet talteen listaan.
- Connect\_caves-funktio yhdistää luolaston huoneet tarpeen vaatiessa.

```
func fill_roof():
    for x in range(0, map_w):
        for y in range(0, map_h):
            set_cell(x, y, Tiles.ROOF)
```

```
func random_ground():
    for x in range(1, map_w-1):
        for y in range(1, map_h-1):
            if Util.chance(ground_chance):
                set_cell(x, y, Tiles.GROUND)
```

```
func dig_caves():
    randomize()
    for i in range(iterations):
        var x = floor(rand_range(1, map_w-1))
        var y = floor(rand_range(1, map_h-1))
        if check_nearby(x,y) > neighbors:
            set_cell(x, y, Tiles.ROOF)
        elif check_nearby(x,y) < neighbors:
            set_cell(x, y, Tiles.GROUND)
```

```
func get_caves():
    caves = []
    for x in range (0, map_w):
        for y in range (0, map_h):
            if get_cell(x, y) == Tiles.GROUND:
                flood_fill(x,y)
    for cave in caves:
        for tile in cave:
            set_cellv(tile, Tiles.GROUND)
```

Koodiesimerkki 4. Funktioiden toteutuksia koodina [11.]

Yksi mielenkiintoisimpia algoritmeja ohjelmassa on käytävien luominen luolaston huoneiden välille. Koodissa on rivejä kuitenkin sen verran monta, että on parempi käydä koodia läpi pala kerrallaan.

Ensimmäisenä esitellään tunnelin luonti funktio, joka saa parametreinä kaksi pistettä ja itse generoidun luolan.

```
func create_tunnel(point1, point2, cave):
    randomize()
    var max_steps = 500
    var steps = 0
    var drunk_x = point2[0]
    var drunk_y = point2[1]
```

...

Koodiesimerkki 5. Käytävän luonti funktion määrittely [11.]

Seuraavaksi käydään läpi edellisessä funktiossa määritelty määrä askelia.

```
...
while steps < max_steps and !cave.has(Vector2(drunk_x, drunk_y)):
    steps += 1
```

...

Koodiesimerkki 6. Käydään läpi askeleet [11.]

Sitten määritellään vakiopainoarvot muuttujille n, s, e, w. N tarkoittaa north eli pohjoinen. S on south eli etelä. E on east eli itä ja w on west eli länsi.

```
...
var n = 1.0
    var s = 1.0
    var e = 1.0
    var w = 1.0
    var weight = 1
```

...

Koodiesimerkki 7. Määritellään vakiopainoarvot muuttujille [11.]

Seuraavana painotetaan satunnaiset reunalähdöt luolaston huoneiden reunalta.

```
...
if drunk_x < point1.x:
    e += weight
elif drunk_x > point1.x:
    w += weight
if drunk_y < point1.y:
    s += weight
elif drunk_y > point1.y:
    n += weight
...
```

Koodiesimerkki 8. Painotetaan satunnaiset reunalähdöt [11.]

Sitten normalisoidaan todennäköisyys välille nolla ja yksi. Otetaan siis jokaisen ilmansuunnan keskiarvo ja valitaan suunta, johon käytävää lähdetään kaivamaan.

```
...
var total = n + s + e + w
    n /= total
    s /= total
    e /= total
    w /= total
...
```

Koodiesimerkki 9. Normalisoidaan todennäköisyys [11.]

```
...
var dx
var dy
var choice = randf()

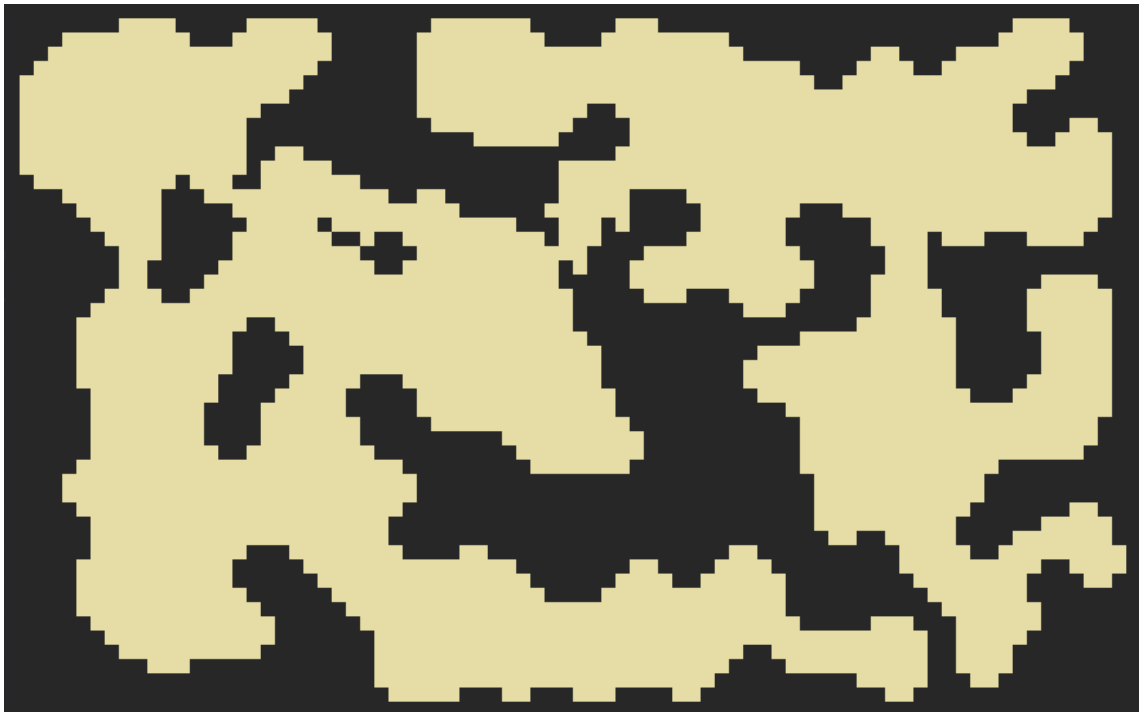
if 0 <= choice and choice < n:
    dx = 0
    dy = -1
elif n <= choice and choice < (n+s):
    dx = 0
    dy = 1
elif (n+s) <= choice and choice < (n+s+e):
    dx = 1
    dy = 0
else:
    dx = -1
    dy = 0
...
```

Koodiesimerkki 10. Valitaan suunta [11.]

Lopuksi tehdään vielä tarkistus, ettei käytävää lähdetä kaivamaan kentän rajojen yli. Luolastoon siis jää näin ulkoreunalle aina läpipääsemätön seinä.

```
...
if (2 < drunk_x + dx and drunk_x + dx < map_w-2) and \
    (2 < drunk_y + dy and drunk_y + dy < map_h-2):
    drunk_x += dx
    drunk_y += dy
    if get_cell(drunk_x, drunk_y) == Tiles.ROOF:
        set_cell(drunk_x, drunk_y, Tiles.GROUND)
        set_cell(drunk_x+1, drunk_y, Tiles.GROUND)
        set_cell(drunk_x+1, drunk_y+1, Tiles.GROUND)
```

Koodiesimerkki 11. Varmistetaan, ettei kentän rajojen yli mennä [11.]



Kuva 14. Luolasto, joka on selvästi jakautunut osiin

Muutetaan ohjelman muutamaa muuttujaa, jotta saadaan erilainen luolasto aikaiseksi.

```
var map_w    = 80
var map_h    = 50
var iterations = 150000
var neighbors = 4
var ground_chance = 58
var min_cave_size = 80
```

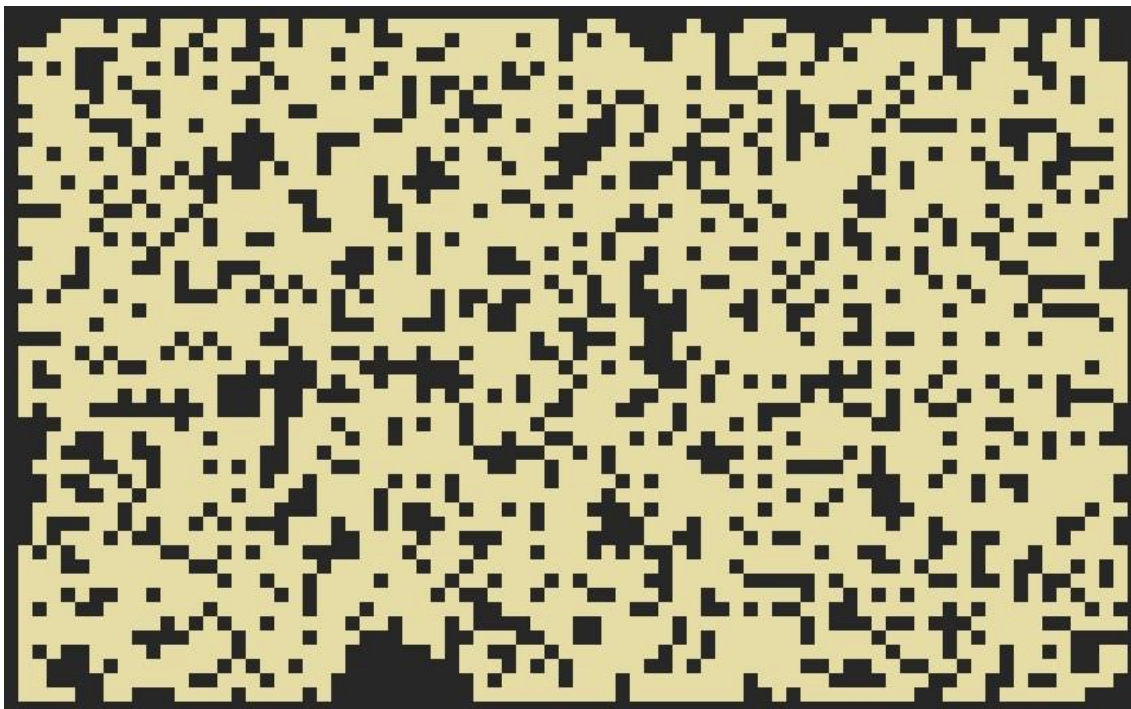
Koodiesimerkki 12. Muuttujat, joilla pystyy vaikuttamaan luolaston muotoon [11.]



Kuva 15. Luolasto, joka on aukeampi

Kuvissa 15 ja 16 nähdään, että vaihtamalla paria muuttujaa ohjelman koodissa saadaan todella erilaisia lopputuloksia. Muuttujien vaihdolla saa myös muutettua tason kokoa, mutta tässä kannattaa olla maltillinen, sillä liian ison kentän generointi saattaa kestää aivan liian kauan. Jos iteraatioita pienennetään liikaa, tuottaa algoritmi lähinnä solujen sekamelskan (kuva 17).





Kuva 16. Soluautomaatti on käynyt liian vähän solujen sukupolvia läpi

## 6.2 Tasogeneroinnin tulokset

Soluautomaatti toimii hyvin, jos halutaan generoida luolamaisia kenttiä videopeliin. Pelikokemuksen kannalta luolastot toimivat hyvin tietyissä skenaarioissa ja toisissa heikommin. Jos ajatellaan, että luolamainen kenttä generoitaisiin peliin, jossa tarkoituksena olisi päästä aloituspisteestä päätepisteeseen, on luolamainen kenttä tähän tarkoitukseen hyvä. Pelaaja voi lähteä moneen eri suuntaan avonaisessa luolastossa, ja jos aloitus- sekä päätepiste on arvottu vaikkapa minimietäisyyspohjaisesti, ei ongelmaa liian lyhyestä tai helposta kentästä synny.

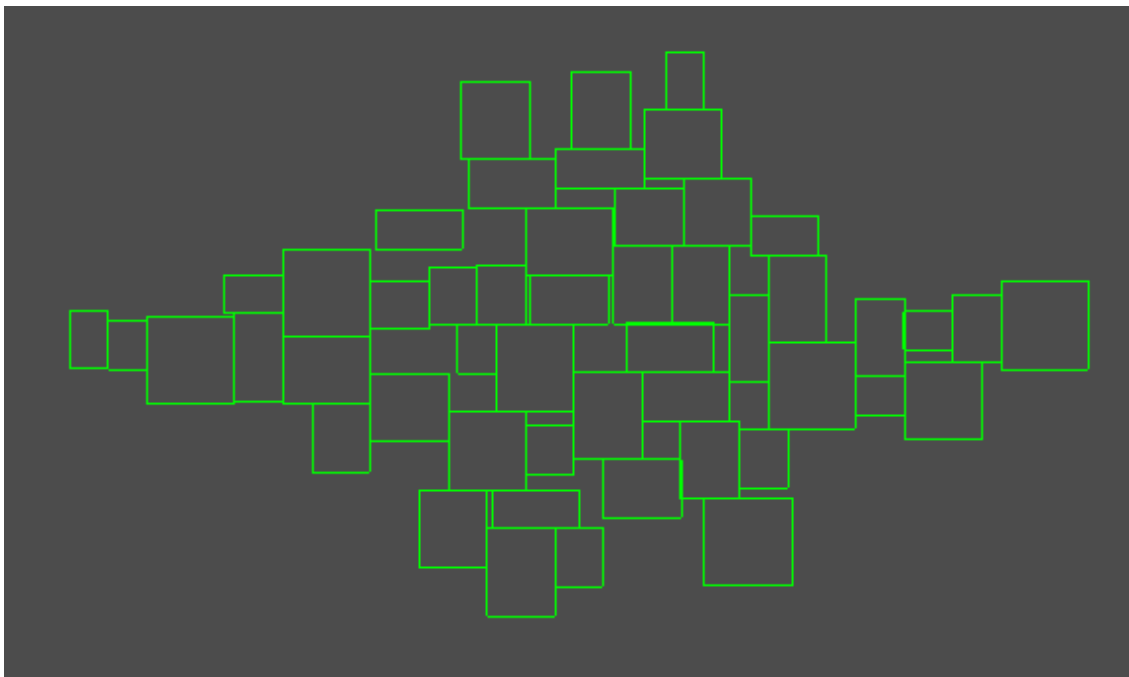
Jos haluaa tavoitella suoraviivaisempaa pelikokemusta, on ehkä parempi käyttää vähemmän aukeaa luolastoa, jossa on enemmän sokkelomaista pohjaa. Tällöin pelaaja tosin tulee törmäämään umpikujiin samalla tavalla kuin toimijapohjaisessa toteutuksessa. Hyvänä puolena voidaan kuitenkin nähdä se, että jos pelaajahahmo on jälleen yhden solun kokoinen objekti, on liikkumavaraa soluautomaattitoteutuksessa huomattavasti enemmän. Tällöin pelaaja voi vaikkapa kiertää jotain esteitä pelissä, mikä ei olisi mahdollista yhden solun paksuisissa käytäväsokkeloissa. Avoimen luolaston

kohdalla myös kentän isompi koko ei aiheuta niin nopeasti pelaajalla hirveää päänvaivaa, koska päätepiesteeseen on mahdollisuus navigoida useampaa reittiä pitkin. Avoimessa luolastossa voisi olla hyvä myös rajoittaa pelaajan näkökenttää, jolloin peli vaatisi sen, että pelaaja tutkii luolastoa enemmän kuin vain sen verran, että tämä pyrkii aktiivisesti kohti päätepiestettä. Soluautomaattitoteutuksessa toimisi myös erinomaisesti “minimiaskel aloitus- ja päätepiesteen välille” -ratkaisu.

## 7 Kaksiulotteisen tason generointi fysiikkamoottoritekniikalla

### 7.1 Tasogeneroinnin toteutus Godotilla

Fysiikkamoottoritoteutus on melko yksinkertainen toteutukseltaan, ja se monimutkaistuu huomattavasti, kun siihen lisätään Primin algoritmilla huoneiden välille käytävät. Toteutuksessa tehdään kasa huoneita käyttäen Godotin sisäänrakennettua fysiikkamoottoria nimeltään Bullet. Tämän jälkeen huoneille tehdään törmäysmalli, joka estää huoneita generoitumasta päällekkäin, jolloin fysiikkamoottori levittää huoneet satunnaisesti ympäriinsä. Ohjelmassa pystyy määrittelemään huoneiden minimi- ja maksimikoon. Ohjelmassa on myös määritelty huoneille vaakatason leviämismuuttuja, joka levittää huoneita enemmän vaaka kuin pystytasossa. Tällä on tarkoituksena se, että pelaaja mahdollisesti aloittaa pelin joko kentän vasemmasta tai oikeasta laidasta. Lopputulos on kuvan 18 mukainen. [12.]



Kuva 17. Ensimmäinen vaihe fysiikkamoottoritoteutuksessa

Huoneet saadaan tehtyä näppärästi käyttäen Godotin sisäänrakennettua piirto-ominaisuutta, jolla saadaan tehtyä suorakulmion muotoisia huoneita.

```
extends RigidBody2D
var size
func make_room(_pos, _size):
    position = _pos
    size = _size
    var s = RectangleShape2D.new()
    s.custom_solver_bias = 0.75
    s.extents = size
    $CollisionShape2D.shape = s
```

Koodiesimerkki 13. Funktio, jolla saadaan piirrettyä huoneet [12.]

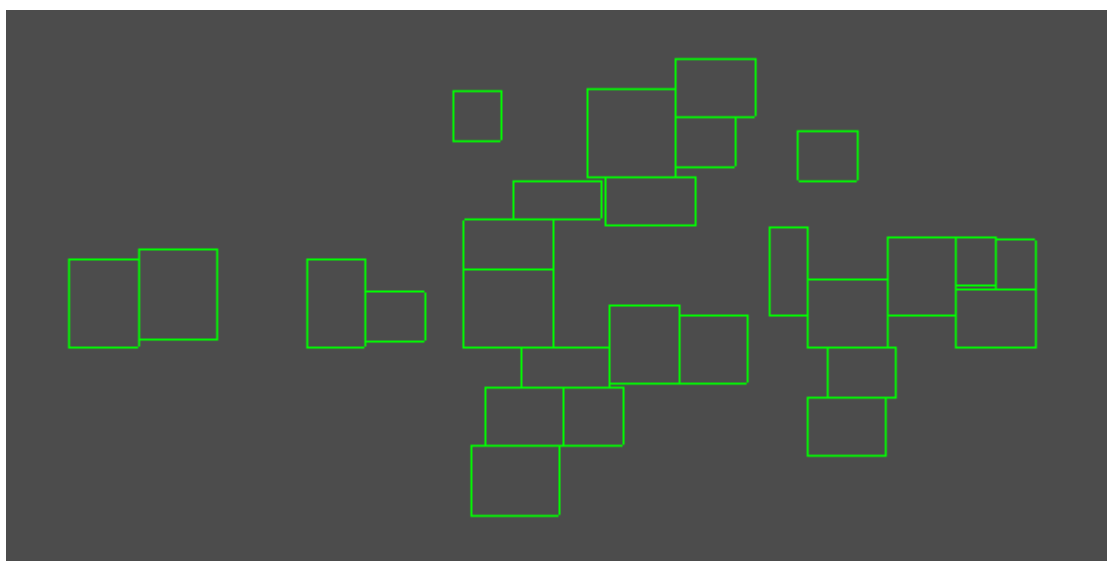
Tämän jälkeen otetaan huoneista noin puolet pois, jolloin saadaan paremman näköinen tulos. Huoneiden poiston määrää pystyy muuttamaan, jos haluaa vaikkapa tehdä sadan huoneen kentän ja poistaa näistä huoneista neljäkymmentä, saadaan kuudenkymmenen huoneen kenttä. Esimerkissä on tehty viidenkymmenen huoneen kenttä ja poistettu huoneista puolet. Huoneen poisto ohjelman koodissa tapahtuu antamalla huoneelle tietty mahdollisuus prosentteina tuhoutua.

```

func make_rooms():
    for i in range(num_rooms):
        var pos = Vector2(rand_range(-hspread, hspread), 0)
        var r = Room.instance()
        var w = min_size + randi() % (max_size - min_size)
        var h = min_size + randi() % (max_size - min_size)
        r.make_room(pos, Vector2(w, h) * tile_size)
        $Rooms.add_child(r)
    yield(get_tree().create_timer(1.1), 'timeout')
    var room_positions = []
    for room in $Rooms.get_children():
        if randf() < cull:
            room.queue_free()
        else:
            room.mode = RigidBody2D.MODE_STATIC
            room_positions.append(Vector3(room.position.x, room.position.y, 0))
    yield(get_tree(), 'idle_frame')
    path = find_mst(room_positions)

```

Koodiesimerkki 14. Huoneiden luontifunktio [12.]



Kuva 18. Generoitu kenttä, jossa huoneista on poistettu puolet

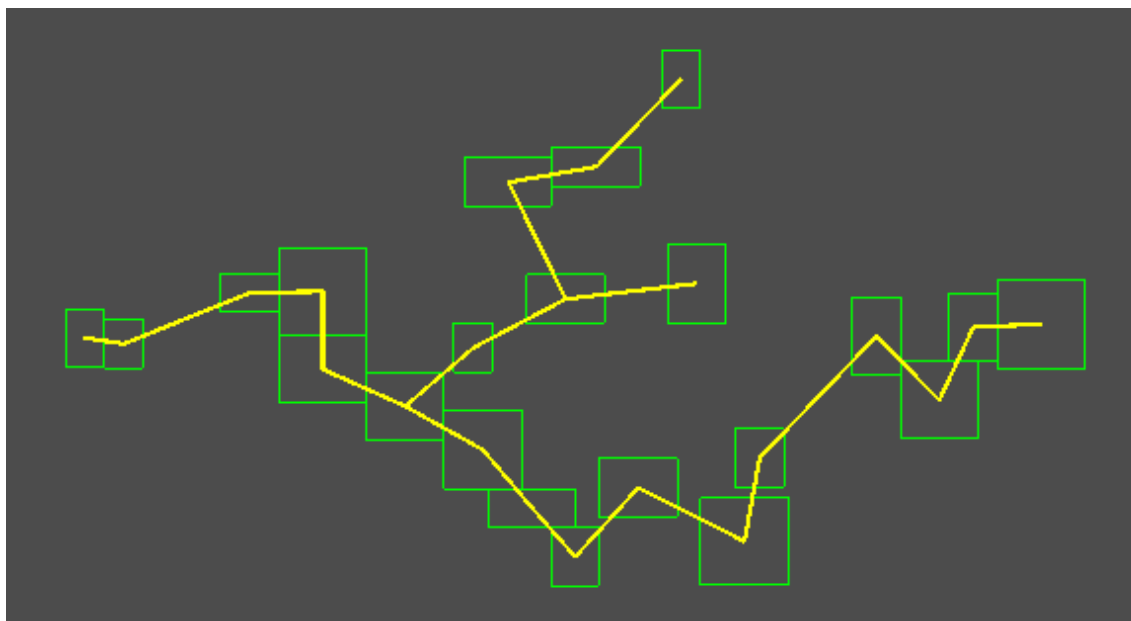
Kun huoneista on puolet poistettu, toteutetaan Primin algoritmi, jolla huoneet saadaan yhdistettyä toisiinsa. Lyhyesti ilmaistuna Primin algoritmi etsii pienimmän virittävän puun, jonka avulla huoneiden yhteys löydetään. On olemassa muitakin algoritmeja, jotka pystyvät löytämään virittävän puun, esimerkiksi Kruskalin algoritmi ja Borůvkan algoritmi. [13; 14; 15.]

```

func find_mst(nodes):
    var path = AStar.new()
    path.add_point(path.get_available_point_id(), nodes.pop_front())
    while nodes:
        var min_dist = INF
        var min_p = null
        var p = null
        for p1 in path.get_points():
            p1 = path.get_point_position(p1)
            for p2 in nodes:
                if p1.distance_to(p2) < min_dist:
                    min_dist = p1.distance_to(p2)
                    min_p = p2
                    p = p1
        var n = path.get_available_point_id()
        path.add_point(n, min_p)
        path.connect_points(path.get_closest_point(p), n)
        nodes.erase(min_p)
    return path

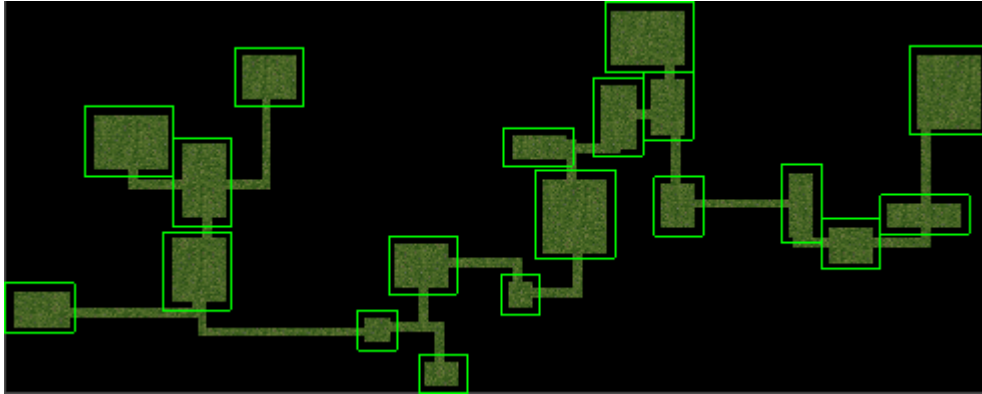
```

Koodiesimerkki 15. Primin algoritmi ohjelmoinnillisesti toteutettuna [12.]



Kuva 19. Primin algoritmi toteutettuna keltaisena viivana huoneiden välille

Lopuksi toteutetaan itse huoneiden yhdistäminen käytävillä. Ohjelmaan on määritelty, että käytävä on noin kolmen solun paksuinen suora kaistale, jonka ilmansuuntina voi olla ylös, alas, vasen tai oikea. Vinottain kulkevia käytäviä ei generoida.



Kuva 20. Huoneet yhdistettynä käytävillä

```
func carve_path(pos1, pos2):
    var x_diff = sign(pos2.x - pos1.x)
    var y_diff = sign(pos2.y - pos1.y)
    if x_diff == 0: x_diff = pow(-1.0, randi() % 2)
    if y_diff == 0: y_diff = pow(-1.0, randi() % 2)
    var x_y = pos1
    var y_x = pos2
    if (randi() % 2) > 0:
        x_y = pos2
        y_x = pos1
    for x in range(pos1.x, pos2.x, x_diff):
        Map.set_cell(x, x_y.y, 0)
        Map.set_cell(x, x_y.y + y_diff, 0)
    for y in range(pos1.y, pos2.y, y_diff):
        Map.set_cell(y_x.x, y, 0)
        Map.set_cell(y_x.x + x_diff, y, 0)
```

Koodiesimerkki 16. Käytävien luontiskripti [12.]

## 7.2 Tasogeneroinnin tulokset

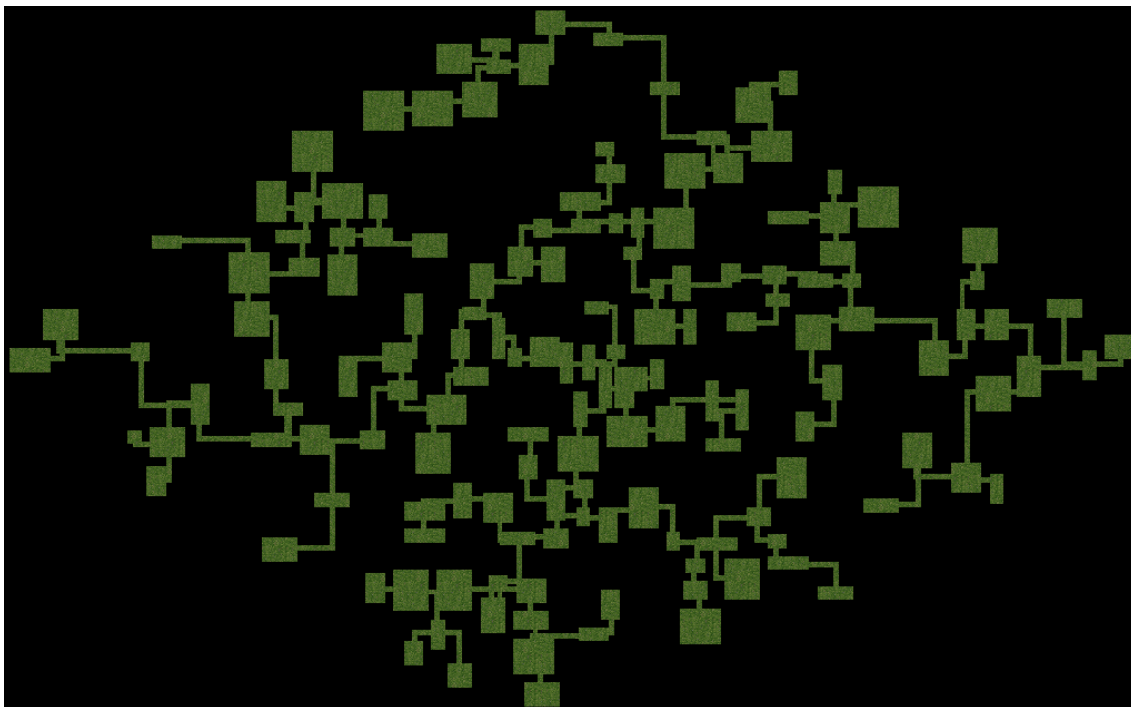
Fysiikkamoottoritoteutuksella saa lopulta ihan mallikkaan näköisiä kenttiä aikaiseksi. Kentät toki on jälleen hyvä pitää kohtuullisen kokoisena, koska liian isoa kenttää olisi todella hankala navigoida ja pelin kulku olisi vaikeampi toteuttaa. Algoritmi generoi valitettavasti paljon umpikujia, kun määritellään pelaajan aloittavan vasemman laidan huoneesta ja päätepisteen sijaitsevan oikean laidan huoneessa. Pienemmän kokoisilla kentillä tätä ongelmaa pystyy hieman kiertämään.

Algoritmi ei välitä siitä, meneekö käytävät päällekkäin, mikä saattaa joskus tuottaa hieman oudon näköisiä käytävämuodostelmia, jos huoneita on monia vierekkäin. Tämä saattaa myös aiheuttaa sen, että pelaaja pääsee johonkin huoneeseen kahdesta eri kohdasta, mikä ei ole pelin kulun kannalta välttämättä suotavaa. Kun generoi todella ison kentän, 150 huonetta (kuva 21), alkaa algoritmi hieman valitettavasti murentua. Huoneet saattavat generoitua päällekkäin ja käytävät mennä huoneiden päältä. Lopputulos on melkoinen sekamelska ja näyttää todella konemaiselta ulkoasullisesti virheineen.

Fysiikkamoottoritoteutuksessa on myös yksi todella huono puoli ja se liittyy käytävien pituuteen. Koska huoneet joskus leviävät todella paljon, saattaa tämä tuottaa turhan pitkiä käytäviä, jotka rikkovat pelin jouhevuuatta, kun pelaaja liikkuu huoneesta toiseen.

Hyvänä puolena toteutuksessa voi pitää sen helppoa muokattavuutta, kun säätää algoritmin käyttämiä muuttujia. Tarpeeksi kauan kun etsii oikeita muuttujien arvoja, saa varmasti parempia generoituja kenttiä.

Tämä toteutus voisi kuitenkin toimia hyvänä pohjana pelin tason generoinnille, mutta se vaatisi pieniä korjauksia. Yksi parannus voisi olla, että joihinkin huoneisiin on kaksi käytävää yhden sijaan. Tällöin pelaaja ei ajautuisi niin helposti umpikujiiin. Toinen parannus voisi olla huoneiden liika leviämisen estäminen, jolloin myös käytävien pituus olisi maltillisempi. Lisäksi algoritmiin voisi lisätä umpikujiiin johtavien polkujen lyhentäminen tai vaikka niiden kokonaan estäminen. Tällöin toki kentän suoraviivaisuus lisääntyisi, eikä pelissä enää olisi tutkimista lainkaan. Suoraviivaisuus voi silti olla suotavaa, jos pelin tempo haluttaisiin pitää nopeana.



Kuva 21. 150 huoneen kenttä. Pelattavuus todennäköisesti kärsii tämän kokoisen kentän kanssa

## 8 Yhteenveto

Tasogenerointi algoritmit ovat mielenkiintoisia ja oli yllättävää, kuinka paljon matemaattisia malleja pystyy hyödyntämään proseduraalisen generoinnin ohjelmoinnissa. Tämän työn tehtyäni aloin ajattelemaan matematiikkaa ja algoritmeja uudessa valossa niiden näppäryyden vuoksi ohjelmoinnissa.

Erilaisia kenttien generointialgoritmeja on olemassa kymmeniä ja näistä on vielä olemassa erilaisia variaatioita yhtä monta. Tässä työssä esitellyt algoritmit ovat melko yleisiä paitsi fysiikkamoottoritoteutus, jossa tarvitaan pelimoottorin sisäänrakennettua fysiikkamallinnusta. Kentän generointialgoritmeja on tutkittu paljon ennen tämän työn syntymää, ja jos aihe kiinnostaa, kannattaa lukea tässä työssä aikaisemmin viitattuun tutkielmaan, jonka tekivät kuusi Göteborgin yliopiston oppilasta. [2.]

Internet on pullollaan erilaisia kentän generoimiskursseja, mutta näistä valitettavan moni on kuitenkin jossain määrin vajaita. Mielestäni tässä työssä parhaiten oikean videopelin kenttägeneraattoreina toimisi agenttipohjainen ja soluautomaattiratkaisut ilman sen



suurempaa hiomista. Fysiikkamoottoriratkaisu on melko keskeneräinen ja sitä pitäisi paljon viilata vielä, että sitä pystyisi oikeassa videopelissä käyttämään. Videopelin kehitystä ajatellen on tärkeää miettiä, minkälaisia kenttiä peliinsä haluaa tehdä pelattavuutta tukien.

Tämän päivän videopelit käyttävät erittäin paljon proseduraalista generointia niin kenttien kuin pelissä olevien esineiden luomiseen. Nämä algoritmit saattavat olla erittäin monimutkaisia, joissa otetaan huomioon valtavat määrät erilaisia muuttujia pelaajahahmon tasosta kentän kokoon. [2.]

Kannattaa kuitenkin ottaa huomioon, että proseduraalinen generointi ei välttämättä ole aina paras ratkaisu, kun kehitetään videopeleihin kenttiä. Joskus on pelin logiikan ja pelattavuuden kannalta huomattavasti suositeltavampaa tehdä kenttiä sekä esineitä manuaalisesti. Toisin sanoen proseduraalista generointia ei missään nimessä kannata ajatella sopivana oikotienä, jos haluaa hyvän pelin kehittää. [2.]

Uskoisin, että tulevaisuudessa proseduraalista generointia hyödynnetään mahdollisesti nykyistä enemmän ja esimerkiksi koneoppimisen sekä tekoälyn lisääminen algoritmeihin toisi parannuksia ja uusia mahdollisuuksia proseduraaliseen generointiin.

## Lähteet

- 1 Procedural generation. Verkkoaineisto. Wikipedia.  
<[https://en.wikipedia.org/wiki/Procedural\\_generation](https://en.wikipedia.org/wiki/Procedural_generation)> Luettu 27.9.2019.
- 2 An Exploration of Procedural Content Generation for Top-Down Level Design Bachelor's thesis in Computer Science and Engineering. Verkkoaineisto.  
<<https://pdfs.semanticscholar.org/56c9/e151acc37b720778e61a35713e58c80b05a5.pdf>> Luettu 2.10.2019.
- 3 Beneath Apple Manor. Verkkoaineisto. Wikipedia.  
<[https://en.wikipedia.org/wiki/Beneath\\_Apple\\_Manor](https://en.wikipedia.org/wiki/Beneath_Apple_Manor)> Luettu 27.9.2019.
- 4 Rogue. Verkkoaineisto. Wikipedia.  
<[https://en.wikipedia.org/wiki/Rogue\\_\(video\\_game\)](https://en.wikipedia.org/wiki/Rogue_(video_game))> Luettu 27.9.2019.
- 5 Diablo. Verkkoaineisto. Wikipedia.  
<[https://en.wikipedia.org/wiki/Diablo\\_\(video\\_game\)](https://en.wikipedia.org/wiki/Diablo_(video_game))> Luettu 27.9.2019.
- 6 No Man's Sky. Verkkoaineisto. Wikipedia.  
<[https://en.wikipedia.org/wiki/No\\_Man%27s\\_Sky](https://en.wikipedia.org/wiki/No_Man%27s_Sky)> Luettu 21.10.2019.
- 7 Godot. Verkkoaineisto. Wikipedia.  
<[https://en.wikipedia.org/wiki/Godot\\_\(game\\_engine\)](https://en.wikipedia.org/wiki/Godot_(game_engine))> Luettu 4.10.2019.
- 8 Prim's algorithm. Verkkoaineisto. Wikipedia.  
<[https://en.wikipedia.org/wiki/Prim%27s\\_algorithm](https://en.wikipedia.org/wiki/Prim%27s_algorithm)> Luettu 7.10.2019.
- 9 Kidscancode.org Procedural Generation in Godot. Verkkoaineisto. Chris Bradfield. <[http://kidscancode.org/blog/2018/08/godot3\\_procgen1/](http://kidscancode.org/blog/2018/08/godot3_procgen1/)> Luettu 7.10.2019.
- 10 Soluautomaatti. Verkkoaineisto. Wikipedia.  
<<https://fi.wikipedia.org/wiki/Soluautomaatti>> Luettu 27.9.2019.
- 11 Procedural generation with Godot: Creating caves with Cellular Automata. Verkkoaineisto. Abitawake.com.  
<<https://abitawake.com/news/articles/procedural-generation-with-godot-creating-caves-with-cellular-automata>> Luettu 4.10.2019.
- 12 Kidscancode.org Procedural Generation in Godot. Verkkoaineisto. Chris Bradfield. <[http://kidscancode.org/blog/2018/12/godot3\\_procgen6/](http://kidscancode.org/blog/2018/12/godot3_procgen6/)> <[http://kidscancode.org/blog/2018/12/godot3\\_procgen7/](http://kidscancode.org/blog/2018/12/godot3_procgen7/)> <[http://kidscancode.org/blog/2018/12/godot3\\_procgen8/](http://kidscancode.org/blog/2018/12/godot3_procgen8/)> Luettu 7.10.2019.
- 13 Spanning tree. Verkkoaineisto. Wikipedia.  
<[https://en.wikipedia.org/wiki/Spanning\\_tree](https://en.wikipedia.org/wiki/Spanning_tree)> Luettu 7.10.2019.
- 14 Kruskal's algorithm. Verkkoaineisto. Wikipedia.  
<[https://en.wikipedia.org/wiki/Kruskal%27s\\_algorithm](https://en.wikipedia.org/wiki/Kruskal%27s_algorithm)> Luettu 7.10.2019.

- 15 Borůvka's algorithm. Verkoaineisto. Wikipedia.  
<[https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s\\_algorithm](https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm)> Luettu  
7.10.2019.