Niko Kursu

Sokkeloiden generointi- ja muokkaustyökalu

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

8.5.2023

Tiivistelmä

Tekijä:	Niko Kursu
Otsikko:	Sokkeloiden generointi- ja muokkaustyökalu
Sivumäärä:	24 sivua + 1 liite
Aika:	8.5.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Miikka Mäki-Uuro

Insinöörityössä suunniteltiin ja ohjelmoitiin Unity-editoriin työkalu, jolla voidaan generoida sokkeloita ja muokata niitä. Projektissa noudatettiin ketterän kehityksen periaatteita ja käytettiin pelinkehitysprosessissa yleisiä työkaluja, kuten Unitya, Visual Studiota ja GitHubia.

Insinöörityön aluksi kokeiltiin muutamia erilaisia algoritmeja sokkeloiden muodostamiseen. Algoritmit ohjelmoitiin C#-ohjelmointikielellä Visual Studio 2019 -kehitysympäristössä. Kun projektissa käytettäväksi päätynyt algoritmi oli valittu, se tuotiin Unity-pelimoottoriin. Unityssa luotiin sokkeloiden 3D-malleiksi muuttamiseen tarvittavat prefab-objektit ja ohjelmoitiin tarvittavat välineet sokkeloiden muokkaamiseen.

Lopputuloksena on toimiva ja edelleen jalostamiseen sopiva, vaikkakin jatkokehitystä kaipaava työkalu.

Avainsanat: labyrintti, sokkelo, satunnaisgenerointi, kehitystyökalu

---

Abstract

Author:	Niko Kursu
Title:	Generation of Mazes and tools for editing them
Number of Pages:	24 pages + 1 appendix
Date:	8 May 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology and Communications
Professional Major:	Game Applications
Supervisor:	Miikka Mäki-Uuro, Senior Lecturer

In this project a tool for creating and editing randomized mazes for the Unity editor was implemented. Project was created using agile development methods and common tools, such as Unity, Visual Studio and GitHub.

At the beginning of the project a few different algorithms for generating mazes were tested. They were programmed using the C# -programming language and the Visual Studio 2019 -IDE. Once the algorithm used in the project was chosen, it was brought to the Unity-engine. In Unity the prefabs needed for generating 3D-objects based on the created mazes were manufactured, and the tools needed for creating and shaping the mazes were implemented.

The end result is a functional tool, which has great potential for further development.

Keywords: labyrinth, maze, random generation, development tool

Sisällys

Liitteet

Liite 1: Esimerkkejä algoritmin luomista sokkeloista

Lyhenteet

DFS: Depth-first search. Syvyyssuuntainen haku, reitinhakualgoritmi.

IDE: Integrated development environment. Kehitysympäristö.

NPC: Non-player Character. Pelimaailman hahmo, joka ei ole pelaajan kontrolloima.

VS: Visual Studio. Ohjelmointiympäristö.

1 Johdanto

Opinnäytetyössä lähdettiin kehittämään Unity-editoriin työkalua, jolla on mahdollista satunnaisgeneroida sokkeloita ja muokata niitä. Jonkinlaisia sokkeloita esiintyy lukuisissa peleissä, joten työkalulla on potentiaalia nopeuttaa monien kehittäjien työtä. Lisäksi työkalujen tekeminen kehitysprosessin nopeuttamiseksi ja helpottamiseksi on hyvin kätevä taito pelinkehittäjälle.

Satunnaisgenerointi mahdollistaa vaihtelevan pelikokemuksen luomisen esimerkiksi roguelike-peligenreä edustavissa peleissä. Myös esimerkiksi päivittäisen, viikoittaisen jne. sisällön tuottaminen jatkuvasti päivittyvään peliin helpottuu huomattavasti osittaisella automaatiolla.

Tiukan aikataulun takia tässä projektissa toteutettu työkalu on melko yksinkertainen, mutta selkeän rakenteensa ansiosta helposti laajennettavissa. Koodia kirjoitettaessa pääpaino oli tietenkin ensisijaisesti toimivuudessa, mutta myös luettavuudelle ja selkeydelle annettiin paljon painoa. Koodia on kommentoitu oleellisissa osissa kattavasti, ja muuttujien sekä metodien nimet on valittu siten, että ne jo itsessään ovat kommentteja.

Luvussa 2 kerrotaan hieman labyrinteistä ja sokkeloista sekä yleisesti että videopeleissä ja määritellään sokkelo tämän insinöörityön puitteissa. Kolmannessa luvussa käsitellään projektissa käytettyjä työkaluja ja resursseja. Lisäksi puhutaan hieman projektin työvaiheista ja sen etenemisestä. Neljännessä ja viidennessä luvussa pääpaino on insinöörityön teknisessä puolessa: esitellään sokkeloiden matemaattista taustaa, käytettyjä algoritmeja ja niiden implementaatiota. Raportissa on pyritty selittämään varsinkin lopulliseksi algoritmiksi valikoituneen ns. Depth-First Search (DFS) -algoritmin toiminta tarkasti. Viimeiset luvut keskittyvät projektin tulosten ja tulevaisuuden pohdintaan.

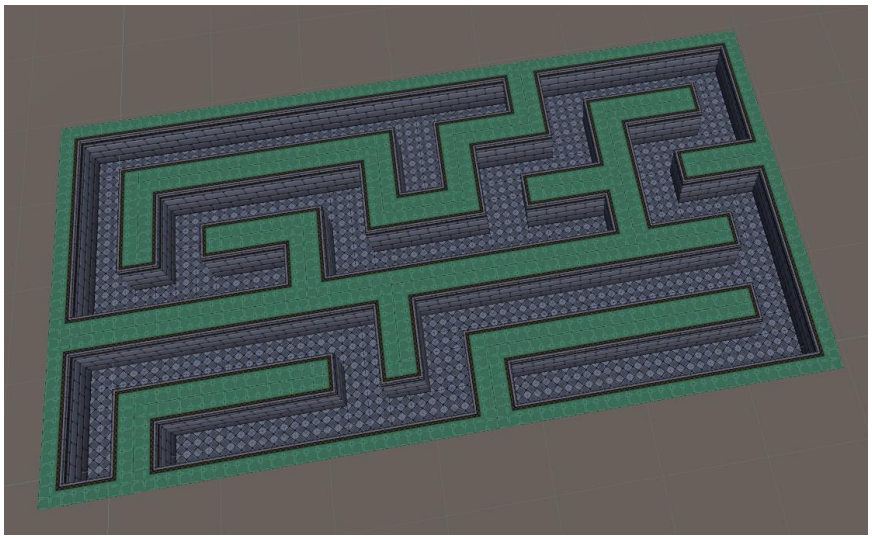
2 Labyrintit ja sokkelot

2.1 Tausta

Labyrintti on rakennus tai kuvio, jonka halki kulkee yksikäsitteinen reitti alkupisteestä loppupisteeseen. Sokkelo poikkeaa labyrintistä siinä, että se saattaa sisältää umpikujia. Tämän projektin keskeinen algoritmi luo nimenomaan sokkeloita, joiden jokaisesta pisteestä toiseen on mahdollista kulkea vain yhtä suoraa reittiä kulkien. Kuitenkin osana projektia luotu työkalu mahdollistaa oikoreittien ja sokkelon peruspalikoita isompien tilojen luomisen.

Arkipuheessa käytetään usein termiä labyrintti, vaikka tarkoitettaisiinkin sokkeloa. Tässä raportissa käytetään kuitenkin termiä labyrintti vain labyrintista puhuttaessa. Kuvassa 1 on esimerkki suomalaiseenkin historiaan kuuluvasta labyrintista, ns. jatulintarhasta. Kuvassa 2 puolestaan on esimerkki sokkelosta.

Kuva 1. Finbyn jatulintarha Nauvossa on esimerkki labyrintistä: kehän ulkoreunalta alkaa reitti keskipisteeseen, jolta ei voi poiketa (Kuva: Wikipedian käyttäjä Alphaios, CC BY-SA 3.0).



Kuva 2. Tämän projektin työkalulla luotu sokkelo. Jokaisesta pisteestä toiseen on vain yksi suora reitti.

2.2 Esimerkkejä sokkeloista peleissä

Sokkeloita on esiintynyt videopeleissä jo niinkin varhain kuin vuonna 1959 (Mouse in the Maze, MIT, TX-0 mainframe). Hieman tunnetumpia esimerkkejä sokkelopeleistä ovat mm. Berzerk (1980, Stern, Arcade), Gauntlet (1985, Atari Games, Arcade) sekä jonkin verran tuoreempana esimerkkinä Legend of Grimrock (2012, Almost Human Games, iOS, Microsoft Windows, Linux, MacOS). Lukemattomissa nykyaikaisemmissa peleissä esiintyy myös sokkeloita, joko kenttinä tai osana niitä. Myös puhtaita sokkelopelejä löytyy, mutta genrenä se ei ole kovinkaan suosittu.

Tässä projektissa luotu työkalu on tarkoitettu luomaan sokkeloita, joiden pohjalta voi lähteä rakentamaan kenttiä. Työkalu siis ei luo valmiita pelejä, vaan vain komponentteja kenttiin.

3 Käytetyt työkalut, resurssit ja projektin eteneminen

Insinöörityössä käytettiin pitkälti tuttuja, turvallisia ja hyväksi havaittuja työkaluja. Projektia varten hankittuja resursseja muokattiin itse, omiin tarkoitusperiin sopiviksi. Työn toteutus eteni enimmäkseen mutkattomasti.

3.1 Visual Studio 2019 -ohjelmointiympäristö

Projektissa käytettiin ohjelmointiympäristönä Visual Studio 2019:ta. Visual Studio on Unityn kanssa työskennellessä luonnollinen, tuttu ja turvallinen valinta, koska se on Unityn oletus-IDE. Ensimmäinen, satunnaistettuun Primin algoritmiin perustuva sokkelogeneraattori kirjoitettiin Visual Studiolla konsolisovellukseksi. Jo tässä vaiheessa oli tiedossa, että projekti tullaan viemään Unityyn, joten sekä VS että C# olivat selkeästi parhaat valinnat käytettäviksi IDE:ksi ja ohjelmointikieleksi.

3.2 Unity 2020.3.18f1 -pelimoottori

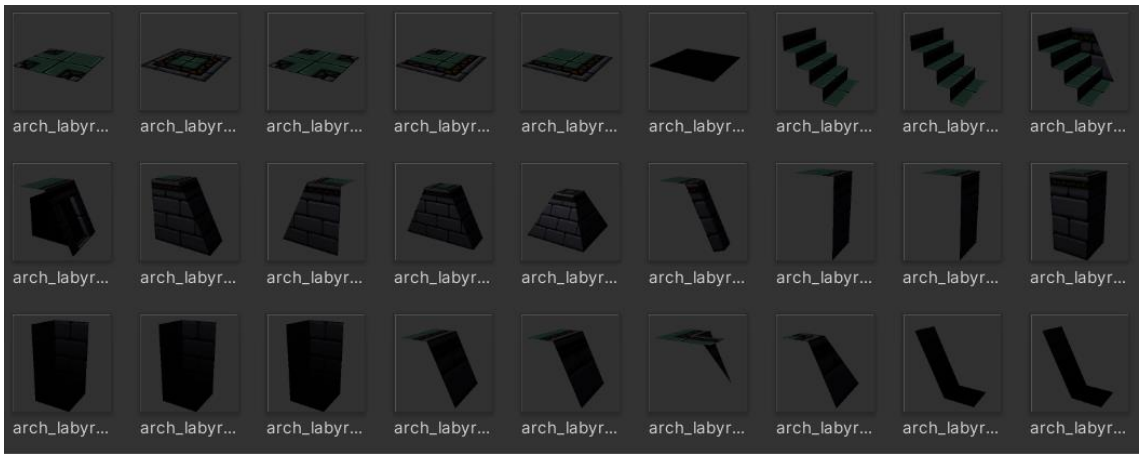
Pelimoottoriksi valikoitui Unity sekä tuttuutensa että sisältämiensä, projektin vaatimien ominaisuuksien vuoksi. Tässä projektin vaatimilla ominaisuuksilla tarkoitetaan mahdollisuutta luoda omia työkaluja kehitysympäristöön sekä mahdollisuutta hankkia ja käyttää kolmannen osapuolen tuottamia 3D-objekteja.

3.3 GitHub- ja GitHub DeskTop -versionhallintatyökalut

Versionhallintaan projektissa käytettiin GitHubia ja sen työpöytäsovellusta. Myös BitBucket vastaavine rajapintoineen oli harkinnassa, mutta valinta kohdistui tällä kertaa GitHubiin.

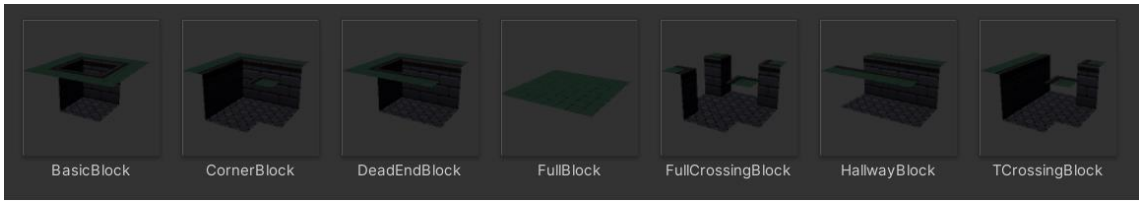
3.4 Retro Dungeons: The Labyrinth 2.2 -resurssipaketti

Sokkelot luotiin Unityssa käyttämällä Unity Asset Storesta hankittua Retro Dungeons -pakettia, joka sisälsi 3D-objekteja ja niihin liitettäviä materiaaleja. Paketti hankittiin verkko-osoitteesta https://assetstore.unity.com/packages/3d/environments/dungeons/retro-dungeons-the-labyrinth-157479. Kuvassa 3 on muutamia Retro Dungeons -paketin sisältämiä prefab-objekteja.



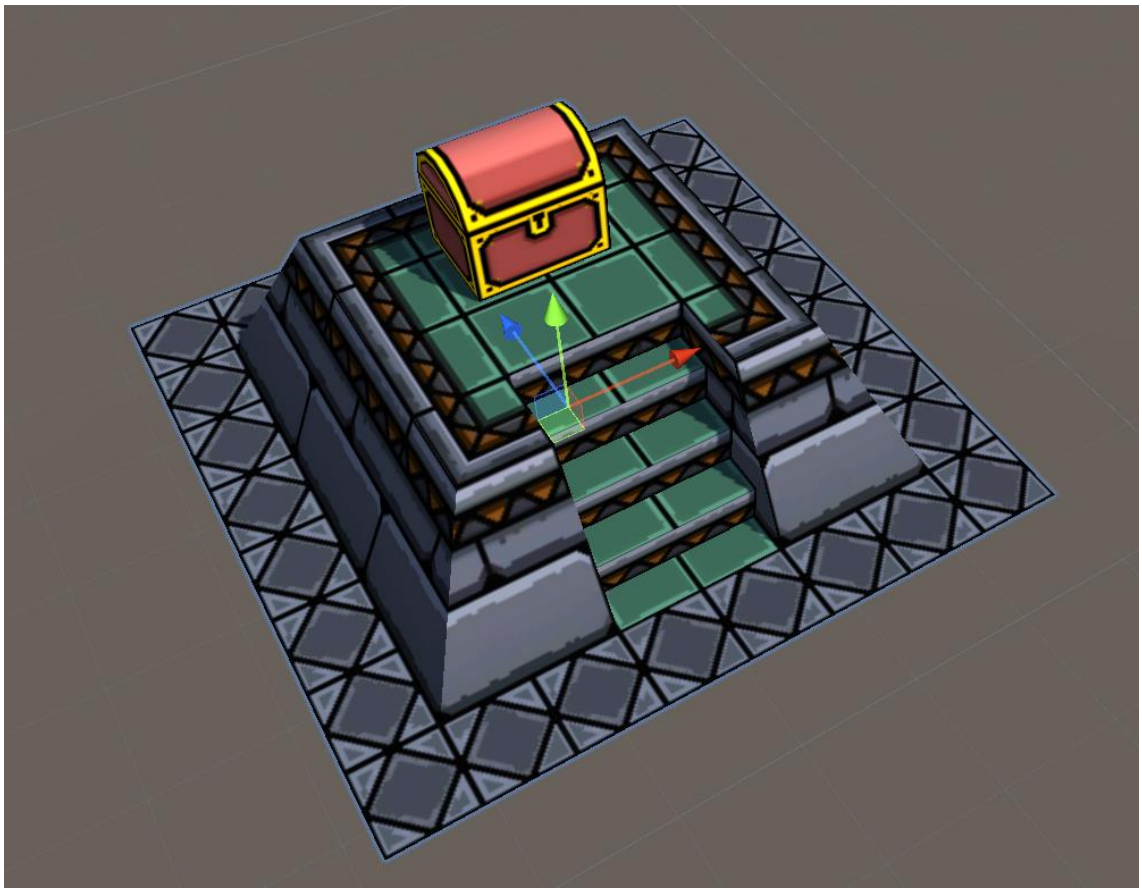
Kuva 3. Osa projektissa käytetyistä kolmannen osapuolen luomista resurs-seista.

Paketin tarjoamista palikoista rakennettiin suurempia, tämän projektin sokkeloiden yhden solun kokoisia prefabeja. Sokkelotaulukon yhden solun koko Unityyn tuotuna on 4 * 4 pelimoottorin yksikköä. Kuva 4 esittää kaikki perustason sokkelon luomiseen tarvittavat prefabit.



Kuva 4. Prefab-objektit, joilla on mahdollista luoda kaikki DFS-algoritmin kehittämät sokkelot.

Lisäksi kokeilumielessä luotiin muutamia erikoispaloja, kuten esimerkiksi isompia huoneita ja aarresaleja. Kuvassa 5 esimerkkinä aarrehuone.



Kuva 5. Esimerkki erikoishuoneesta.

3.5 Kehitysprosessi

Projektilla oli huomattavan tiukka aikataulu, mutta aiempi kokemus reitinhakualgoritmeista nopeutti alkuun pääsyä. Ensimmäinen toimiva sokkeloalgoritmi syntyikin jo viikossa, ja vaikka se hylättiin, siitä saatu kokemus siirtyi hyvin luonnollisesti seuraavaan versioon. Seuraava, DFS-pohjainen algoritmi saatiin kirjoitettua nopeasti sekä kauniiksi että toimivaksi.

Kun sokkeloiden tietorakenne Cell-olioita sisältävänä kaksiulotteisena taulukkona oli päätetty, tarvittiin vielä keino muuntaa sokkelot taulukoista peliobjekteiksi. Koska tämän prosessin voi mieltää siten, että taulukon pohjalta ikään kuin tulostetaan prefabeja pelimoottoriin, nimettiin siitä vastaava metodi MazePrinteriksi. Metodi ottaa luodun taulukon ja sen sisältämien Cell-alkioiden

ominaisuuksien perusteella valitsee sekä tulostettavan prefab-objektin että sen sijainnin ja rotaation pelimaailmassa.

Prefab-objektien luominen vei oman aikansa, mutta aiempi ammatillinen kokemus teki prosessista sujuvan. Välttämättömien objektien lisäksi rakennettiin muutamia erikoispalikoita testausta varten.

Versionhallinta ei mennyt ensimmäisellä yrityksellä optimaalisesti; projektiin unohdettiin luoda asianmukainen git-ignore-tiedosto. Oikeanlainen tiedosto löytyi verkko-osoitteesta https://github.com/github/gitignore/blob/main/Unity.gitignore [2023]. Unohdus aiheutti lieviä vaikeuksia tarkkailla projektiin tehtyjä muutoksia ja lisäsi jonkin verran työmäärää. Tämä kuitenkin havaittiin jo varhain ja saatiin korjattua. Lisäksi koska projektin parissa työskenteli vain yksi koodari, voitiin vahvasti luottaa tehtyjen muutosten olevan linjassa projektin yleisen suunnan kanssa. On kuitenkin aiheellista noudattaa hyviä käytäntöjä, jotta vältetään huonojen menettelytapojen sisäistäminen, ja siksi asia korjattiin.

Itse Unity-editoriin kehitetyn työkalun kanssa liikuttiin jokseenkin tuntematto-mammilla työsaroilla, eikä heikohko dokumentaatio auttanut asiaa. Työkalusta saatiin kuitenkin tehtyä toimiva.

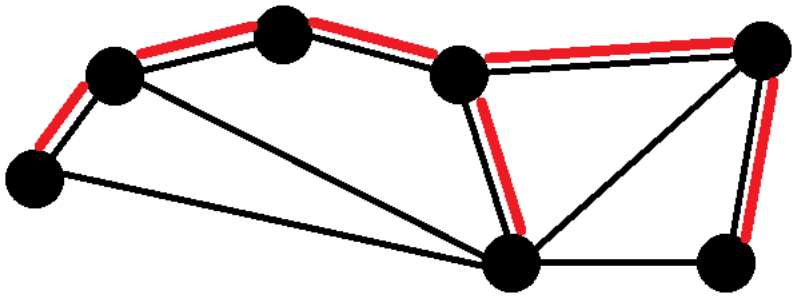
4 Sokkelon luominen

Tässä luvussa kerrotaan projektin teknisestä puolesta hieman tarkemmin, ja varsinkin DFS-algoritmin toiminta kuvaillaan hyvinkin tarkasti. Myös algoritmin implementaatio on pyritty selittämään vähintään riittävällä tarkkuudella. Koodiesimerkit on kommentoitu siten, että lukijan on helppo ymmärtää, mitä niissä tehdään.

4.1 Matemaattinen tausta

Tässä projektissa sokkeloita käsiteltiin verkko- eli graafiteorian pohjalta. Sokkelot voidaan ajatella painottamattomina graafeina, joissa solut ovat taulukon vastaavia soluja ja särmät niiden välistä puuttuvia seiniä.

Voidaan huomata, että jokainen tässä projektissa luotu sokkelo on ns. virittävä puu. Virittävä puu tarkoittaa graafin G syklitöntä aligraafia T, joka sisältää kaikki graafin G solmut (Hella 2017). Kuvassa 6 mustalla on piirretty esimerkki graafista G. Se koostuu seitsemästä solmusta ja niiden välisistä kymmenestä särmästä. Punaisella on piirretty esimerkki virittävästä puusta. Virittävä puu sisältää kaikki G:n solmut, mutta vain sen verran särmiä, kuin on välttämätöntä kaikkien solmujen lisäämiseksi graafiin T (Hella 2017).



Kuva 6. Mustalla piirretty graafi G ja punaisella piirretty sen virittävä puu T.

4.2 Satunnaistettu Primin algoritmi

Ensimmäinen projektia varten luotu algoritmi perustui satunnaistettuun Primin algoritmiin. Algoritmi on seuraavanlainen:

- Aloitetaan ruudukolla, joka on täynnä seiniä.

- Valitaan solu, merkitään se osaksi sokkeloa. Lisätään solun seinät listaan seinistä.

- Niin kauan, kuin lista seinistä sisältää seiniä,

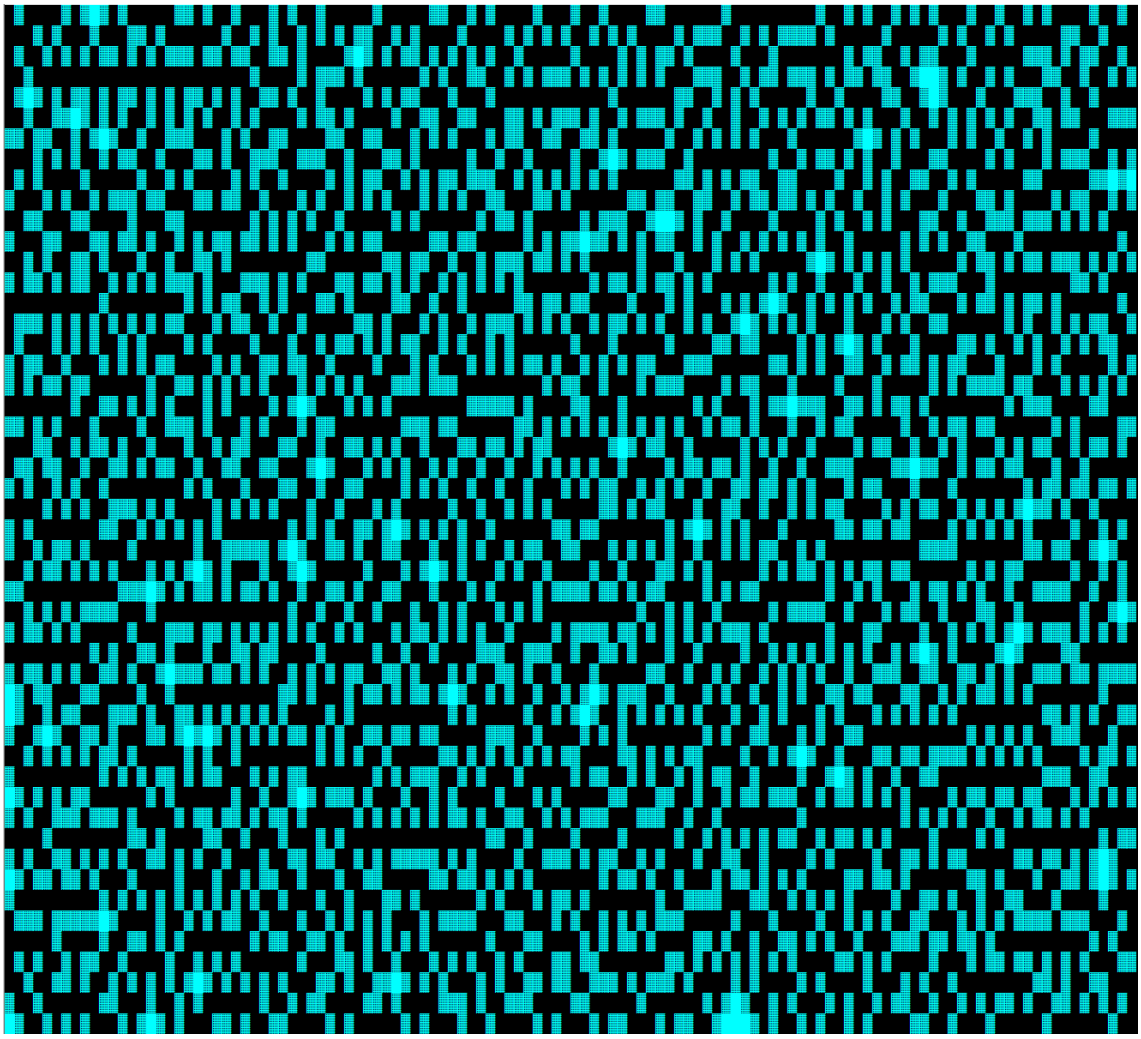
    - valitaan satunnainen seinä listasta. Jos vain toinen soluista, jotka seinä erottaa on jo tutkittu,

        - muutetaan seinä läpikulkureitiksi ja merkitään se solu, jota ei vielä ollut tutkittu tutkituksi

        - lisätään ympäröivät seinät listaan

    - poistetaan seinä listasta.

Algoritmi saatiin ohjelmoitua toimivaksi, joskin sitä testattiin vain konsolilla eikä tuotu Unityyn jatkokehitystä varten. Algoritmi päätettiin hylätä, sillä sen luomat sokkelot olivat ”rumia”: ne sisälsivät useita töksähtäviä umpikujia, kuten kuvasta 7 saattaa huomata.



Kuva 7. Satunnaistetulla Primin algoritmilla luotu sokkelo.

4.3 DFS-algoritmi

Seuraavaksi kokeiltiin Depth-First Searchiin eli DFS:iin perustuvaa algoritmia.

DFS-algoritmin toiminta:

- Valitaan satunnainen solu.
- Merkitään solu tutkituksi ja tehdään lista sen naapureista. Jokaiselle naapurille, aloittaen satunnaisesti valitusta naapurista:

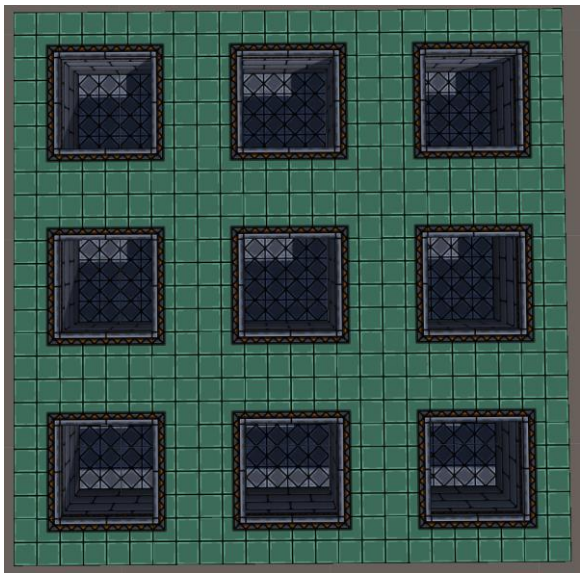
- Jos kyseistä naapuria ei ole vielä tutkittu, poistetaan seinä solun ja naapurin välistä ja tehdään rekursio käyttäen naapuria uutena valittuna soluna.

(Rosetta Code 2023.)

Tämän algoritmin luomat sokkelot olivat sekä visuaalisesti että kenttäsuunnittelua silmällä pitäen huomattavasti miellyttävämpiä, ks. kuva 2 sivulla 3.

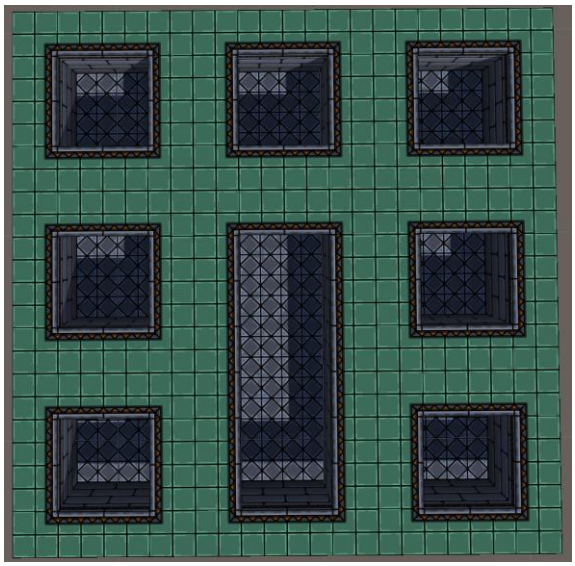
DFS-algoritmin eteneminen

Alussa luodaan taulukko, jonka kaikki solut ovat vielä alkutilassaan, eli toisin sanoen niillä on vielä kaikki seinänsä, ks. kuva 8. Taulukon indeksointi alkaa vasemmasta alakulmasta indeksistä (0, 0).



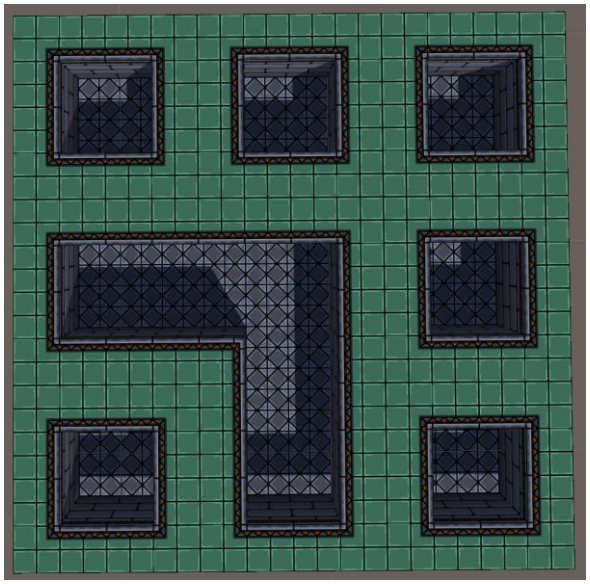
Kuva 8. Sokkelon alkutila.

Sitten valitaan satunnainen solu taulukosta, tässä tapauksessa solu indeksistä (1, 0). Se merkitään tarkastetuksi, ja valitaan sen satunnainen naapuri. Tässä tapauksessa valinta osui soluun (1, 1). Poistetaan seinä solujen välistä, ks. kuva 9.



Kuva 9. Ensimmäinen seinä on poistettu.

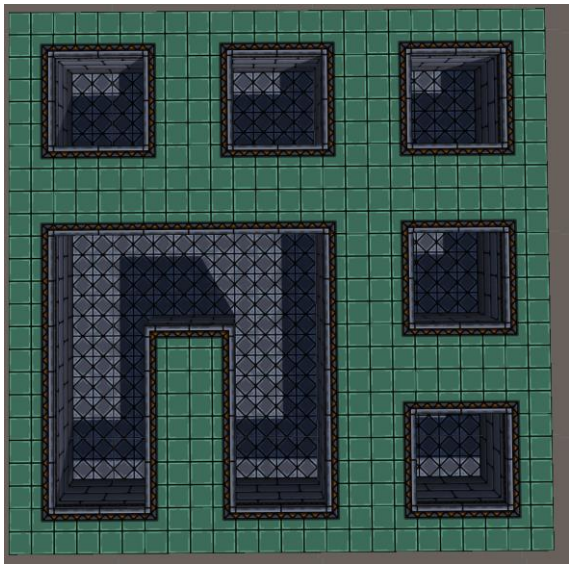
Seuraavaksi valitaan satunnainen solun (1, 1) naapuri, jota vielä ei olla tarkastettu, tässä tapauksessa solu (0, 1). Merkitään solu tarkistetuksi ja poistetaan seinä. Ks. kuva 10.



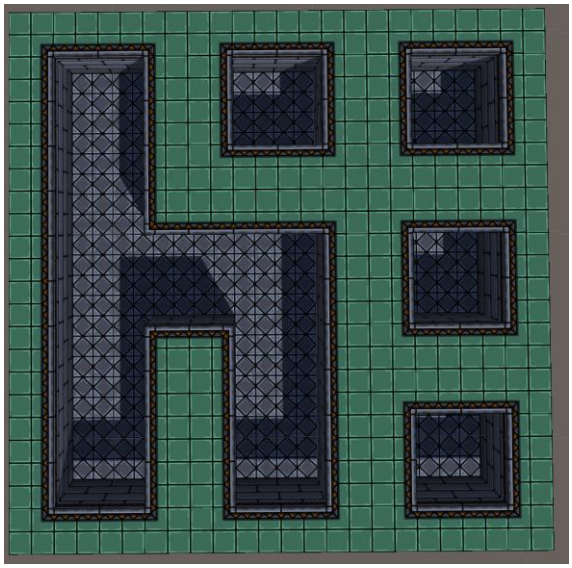
Kuva 10. Toinen seinä on poistettu.

Jatketaan kuten aiemminkin, ja päädytään soluun (0, 0). Nyt voidaan kuvasta 11 huomata, että tällä solulla ei enää ole tutkimattomia naapureita. Rekursio

palaa soluun (0,1), jonka ainoa tutkimaton naapuri on solu (0,2), joten algoritmi jatkaa polun kaiverrustaan sinne, kuva 12.

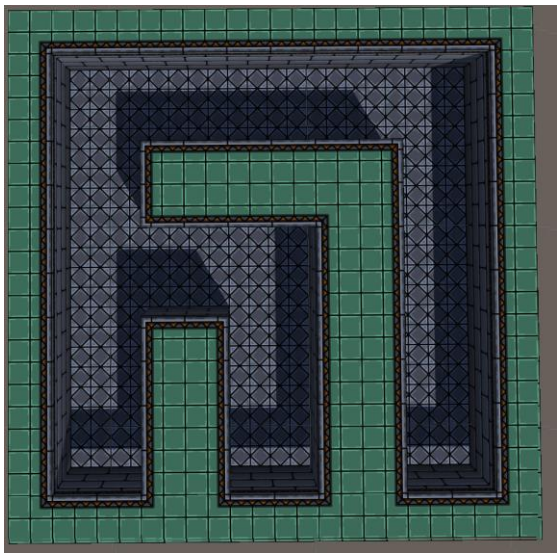


Kuva 11. Kolmas seinä on poistettu ja on saavuttu umpikujaan.



Kuva 12. Algoritmi on jatkanut etenemistään.

Nyt on saavuttu tilanteeseen, jossa algoritmilla on jäljellä vain yksi reitti. Loppu-tulos kuvassa 13.



Kuva 13. Luotu sokkelo.

4.4 Sokkelon generointiin käytetty koodi

Sokkelot luodaan aluksi Cell-olioita sisältävänä kaksiulotteisena taulukkona [Unity Documentation 2023]. Esimerkkikoodissa 1 on Cell-luokan konstruktori:

```
public Cell()
    {
        xPos = 0;
        zPos = 0;
        hasBeenChecked = false;
        neighbours = new Cell[4] { null, null, null, null };
        //in the beginning the cell has all of its walls
        walls = new bool[4] { true, true, true, true };
        surroundingWalls = 4;


    }
```

Esimerkkikoodi 1. Cell-luokan konstruktori.

Solulla on siis sijainnit x-akselilla ja z-akselilla, tieto siitä, onko solu jo tarkastettu, tieto solun seinien lukumäärästä ja taulukot sekä sen naapurisoluista että sen seinistä. Taulukko naapureista, ”neighbours”, sisältää tiedon sekä naapurin olemassaolosta, että sen sijainnista. Taulukko seinistä, ”walls”, toimii vastaavalla tavalla mutta sillä erotuksella, että solulla on oletuksena kaikki neljä seinää. Molemmissa tapauksissa taulukon indeksi nolla tarkoittaa sijaintia solun

vasemmalla reunalla, indeksi yksi yläreunalla, kaksi oikealla reunalla ja kolme alareunalla. Tämän ominaisuuden kätevyys paljastuu, kun taulukkoa siirretään Unityyn 3D-objektiksi muunnettavaksi.

DFS:n olennainen koodi on esitetty esimerkkikoodeissa 2 ja 3.

```
public Cell[,] CreateMaze(int givenWidth, int givenHeight)
    {
        //start with all the walls
        maze = CreateMazeWithOnlyWalls(givenWidth, givenHeight);

        //Start at a random cell
        int randX = Random.Range(0, givenWidth);
        int randY = Random.Range(0, givenHeight);
        Cell cell = maze[randX, randY];

        MakePaths(cell);

        return maze;
    }
```

Esimerkkikoodi 2. Luodaan taulukko, jonka solut ovat vielä käsittelemättä.

Ensimmäisenä siis luodaan annetun kokoinen taulukko, jonka kaikilla soluilla on kaikki seinät. Sitten taulukosta valitaan satunnainen solu, ja sitä alkupisteenä käyttäen kutsutaan rekursiivista MakePaths-metodia. MakePaths-metodin implementaatio on esimerkkikoodissa 3.

```
    private void MakePaths(Cell cell)
    {
        //allowdNeighbourNumbers consists of the neighbours the cell has. 0 =
neighbour to the left, 1 = above
        //2 = to the right and 3 is below
        List<int> allowedNeighbourNumbers = new List<int>();

        //how many neighbours the cell has
        int amountOfNeighbours = 0;

        //mark this cell as checked
        cell.hasBeenChecked = true;

        //set the cells neighbours
        SetNeighbours(cell);

        //check how many neighbours and add the neighbours to the list
        for (int i = 0; i < 4; i++)
        {
            if (cell.neighbours[i] != null)
            {
                allowedNeighbourNumbers.Add(i);
```

```
            amountOfNeighbours++;
        }
    }

    //For each neighbor, starting with a randomly selected neighbor:
    //If that neighbor hasn't been visited,
    //remove the wall between this cell and that neighbor,
    //and then recurse with that neighbor as the current cell.

    //for all the neighbours,
    for (int i = 0; i < amountOfNeighbours; i++)
    {
        //choose random neighbour, and remove it from the list
        int randomIndex = Random.Range(0, amountOfNeighbours - i);
        int checkThisNeighbour = allowedNeighbourNumbers[randomIndex];
        allowedNeighbourNumbers.RemoveAt(randomIndex);

        //if not checked yet
        if (!cell.neighbours[checkThisNeighbour].hasBeenChecked)
        {
            //remove the wall between the cells
            RemoveWallBetween(cell, cell.neighbours[checkThisNeighbour]);
            //and recurse with this neighbour
            MakePaths(cell.neighbours[checkThisNeighbour]);
        }
    }
```

Esimerkkikoodi 3. MakePaths-metodi.

MakePaths siis kutsutaan annetulle solulle, ja merkitään kyseinen solu vierailluksi. Lisäksi määritellään, mitkä naapurit solulla on. Mikäli solu sijaitsee taulukon nurkassa, on sillä vain kaksi naapuria, ja mikäli se sijaitsee reunalla muttei nurkassa, on sillä vain kolme naapuria. Lista ”allowedNeighbourNumbers” koostuu numeroista nollasta kolmeen. Mikäli listassa on numero nolla, se tarkoittaa sitä, että solulla on naapuri vasemmalla puolellaan, numero yksi tarkoittaa naapuria yläpuolella, numero kaksi naapuria oikealla reunalla ja numero kolme alareunalla. Täten esimerkiksi vasemmassa ylänurkassa sijaitsevan solun naapurilista koostuu numeroista kaksi ja kolme.

Näistä naapureista valitaan satunnaisesti yksi, poistetaan se naapurilistasta, ja mikäli sitä ei vielä ole tutkittu, poistetaan seinä alkuperäisen ja tutkittavan solun välistä. Tämän jälkeen kutsutaan MakePaths uudelleen, tutkittava solu annettuna parametrina. Tätä rekursiota toistetaan, kunnes tutkimattomia soluja ei enää ole jäljellä, ts. kaikki naapurilistat ovat tyhjiä.

On syytä huomata, että tämä koodi luo vasta taulukon Cell-tyyppisistä olioista. Jotta sokkelo voidaan luoda Unityssa vaaditaan myös metodia, joka muuntaa taulukon 3D-objekteiksi ja luo ne pelimoottoriin. Esimerkkikoodissa 4 on esimerkki siitä, kuinka umpikuja eli solu, jolla on kolme seinää, käsitellään koodissa.

```
case 3:


    //deadend
    if (givenMaze[i, j].walls[0] == false)
    {
        q = faceLeftQ;
    }
    if (givenMaze[i, j].walls[1] == false)
    {
        q = faceUpQ;
    }
    if (givenMaze[i, j].walls[2] == false)
    {
        q = faceRightQ;
    }
    if (givenMaze[i, j].walls[3] == false)
    {
        q = faceDownQ;
    }

    GameObject.Instantiate(deadEndPiece, new Vector3(i * 8f,
    0, j * 8f), q, mazeGO.transform);

    break;
```

Esimerkkikoodi 4. Umpikuja-palan asennon ja sijainnin määrittäminen.

Metodissa tutkitaan annetun taulukon solut läpi, ja sen mukaan, kuinka monta seinää ja missä sijainneissa ne sillä ovat, määritetään käytettävä prefab ja sen asento. Jos siis esimerkiksi solulla on kaksi seinää, tiedetään sen olevan joko suora käytävä tai nurkka. Edelleen tutkimalla seinien sijainnit saadaan selville, onko kyseessä käytävä vai nurkka ja missä asennossa solu sokkelossa on.

5 Sokkelon muokkaaminen

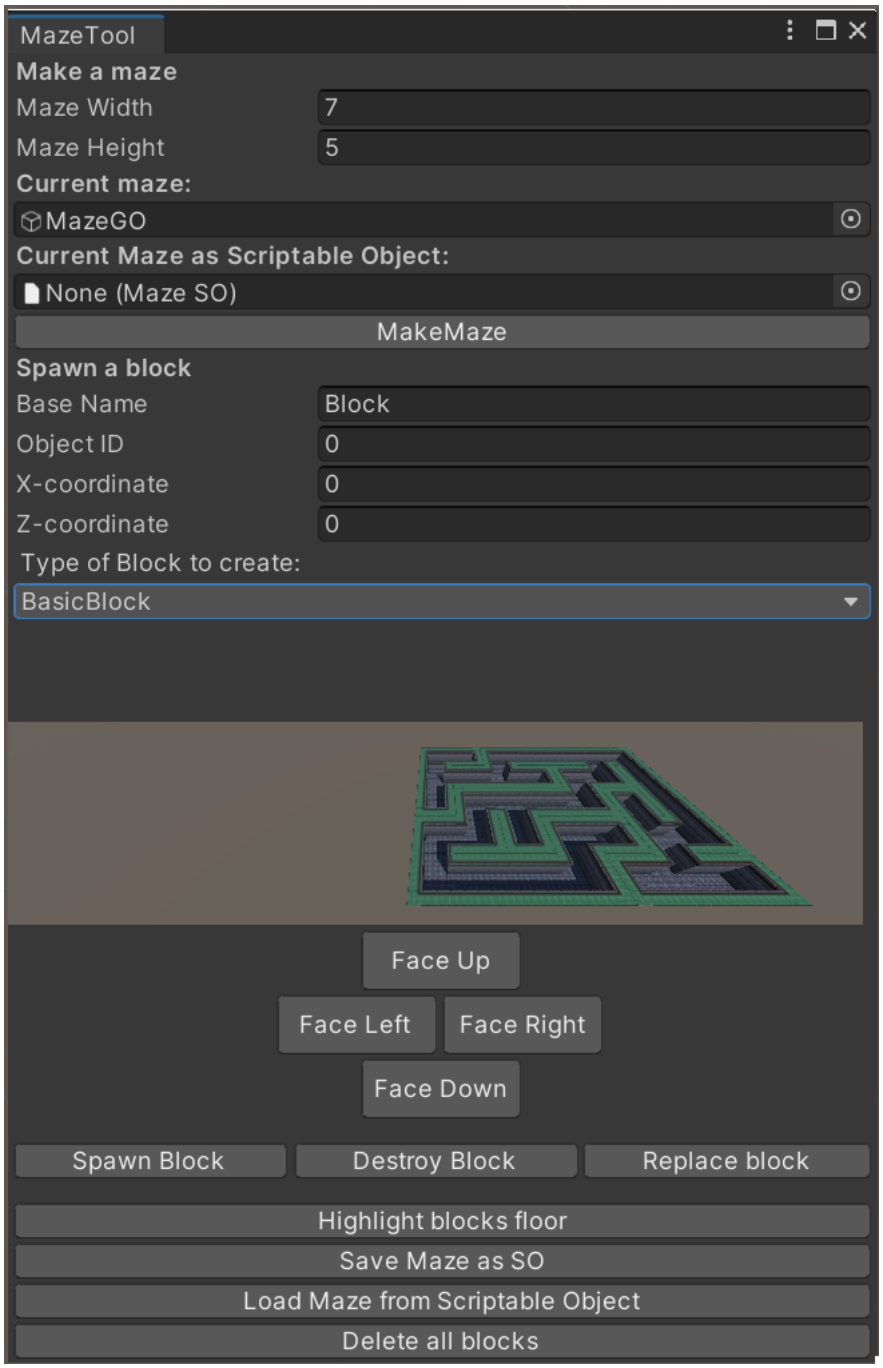
5.1 Luotu työkalu

Unity tarjoaa mahdollisuuden omien työkalujen lisäämiseksi editoriin [Unity Documentation 2023], ja tässä on ominaisuutta hyödynnetty; projektin tarkoitus oli nimenomaan luoda työkalu kehittäjille.

Kun sokkelo on luotu, sitä halutaan muokata. Tätä varten tehtiin Unityyn työkalu, ”MazeTool”. Työkalulla voidaan luoda annetun kokoinen sokkelo, tuhota koko sokkelo, muokata sokkelon yksittäisiä soluja ja tallentaa sokkelo myöhempää työskentelyä varten. Lisäksi on mahdollista rakentaa sokkeloita solu kerrallaan.

Yksittäisen solun muuttaminen vaikuttaa myös sitä ympäröiviin soluihin: jos muutetaan umpikuja käytäväksi, ei riitä vain poistaa muutetusta solusta yhtä seinää vaan myös naapurisolulta on poistettava vastaava seinä. Tämä hoidetaan työkalun koodissa.

Kuvassa 14 näkyy työkalun ulkoasu editorissa.



Kuva 14. MazeTool editorissa.

5.2 Työkalun implementointi

Insinöörityössä tehdyn työkalun koodi on tyypillistä Unityn työkalujen luomiseen käytettyä koodia, jossa itse työkalun olennaisin osuus sijaitsee OnGUI-metodissa. Koodia on tähän raporttiin lisättäväksi liikaa, mutta esimerkkikoodissa 5

on näytteenä kuitenkin palanen metodista, joka päivittää solua ympäröivät solut sen muokkaamisen jälkeen.

```
private void CheckWalls(Cell cell, Cell neighbourCell)
    {
        //if cell to the left of its neighbour
        if (cell.xPos < neighbourCell.xPos)
        {
            //has a wall on the right edge
            if (cell.walls[2])
            {
                //but the neighbour doesnt have a wall on its left edge
                if (!neighbourCell.walls[0])
                {
                    neighbourCell.walls[0] = true;
                    neighbourCell.surroundingWalls++;
                }
            }
            //doesnt have a wall on the right edge
            else
            {
                //but the neighbour does have a wall on its left edge
                if (neighbourCell.walls[0])
                {
                    neighbourCell.walls[0] = false;
                    neighbourCell.surroundingWalls--;
                }
            }
        }
```

Esimerkkikoodi 5. Viereisen solun seinien päivittäminen.

5.3 Sokkelon tallentaminen

Jatkotyöskentelyä varten sokkelo on voitava tallentaa. Helpoin tapa on yksinkertaisesti muuntaa luotu sokkelo prefab-objektiksi. Työkalulla voidaan myös luoda Scriptable Object sokkelosta [Unity Documentation 2023]. Jatkokehityksen kannalta on vielä syytä tarkasti pohtia, mitä ominaisuuksia oikeastaan halutaan tallentaa ja mikä loppujen lopuksi on järkevin tapa tallentamisen toteuttamiseen.

6 Tulokset ja jatkokehitysideat

Projektissa saavutettiin keskeisimmät tavoitteet. Luotiin pelinkehittäjille työkalu, joka kykenee sekä satunnaisgeneroimaan sokkeloita että muokkaamaan niitä. Eritoten sokkelon luomiseen tehty algoritmi on joustava, jatkokehitykseen

mainiosti sopiva palanen koodia. Koodin modulaarisuus ja helppolukuisuus tekee siitä sekä helposti ymmärrettävää että helposti työstettävää.

Yksi keskeisistä tavoitteista oli luoda selkeää koodia. Aiempi ammatillinen kokemus pelinkehityksestä oli synnyttänyt vahvan tahdon kirjoittaa koodia, jota myös kehittäjä, joka koodia itse ei ole luonut, pystyy sekä lukemaan että muokkaamaan. Jossain vaiheessa tulee vastaan raja, kuinka paljon monimutkaisia asioita pystyy yksinkertaistamaan, mutta sekä selkeästi nimetyt metodit, muuttujat ja luokat että asianmukainen kommentointi kuitenkin tekevät koodista ymmärrettävää. Lisäksi koodin jakaminen useisiin metodeihin ja luokkiin auttaa sekä sen muokkaamisessa että luettavuudessa.

Luodut sokkelot ovat visuaalisesti miellyttäviä ja vaikuttavat pelinkehityksen kannalta olevan jo itsessään, ilman kummempaa muokkausta, hyvin käyttökelpoisia kenttiä.

6.1 Reitinhaku

Jatkossa projektiin olisi aiheellista lisätä työkalu, jolla voidaan määrittää kentän aloituspiste ja maali. Lisäksi työkalu, joka visualisoisi reitin aloituksen ja maalin välille, voisi olla hyödyllinen, varsinkin suuria sokkeloita ajatellen. Jos esimerkiksi alku ja loppu arvotaan, tuntuisi pelillisesti epätarkoituksenmukaiselta, mikäli ne osuisivat vierekkäisiin soluihin niin, että kuljettava reitti olisi vain yhden solun mittainen. Muutenkin kenttää muokatessa ei työkalu nykyisellään mitenkään estä umpinaisten alueiden muodostamista. Alkuperäiset, puhtaasti satunnaisgeneroidut sokkelot tosin sisältävät aina reitin sokkelon jokaisesta pisteestä jokaiseen muuhun pisteeseen. Mutta mikäli reitti alusta loppuun sokkeloa muokattaessa katkeaa, ei työkalu tästä varoita. Voitaneen kuitenkin pitää melkoisen huonona kenttäsuunnittelutyöskentelynä tapaa, jossa tuotoksia ei lainkaan testata.

6.2 Valaistus

Valaistuksen automaattinen luominen kenttiin olisi helppo tapa lisätä tehtyjen sokkeloiden visuaalista miellyttävyyttä. Lisäksi valaistuksella voisi olla pelaajan kannalta hyödyllisiä ominaisuuksia: esimerkiksi vihreän sävyinen valo johdattaa aarteen luokse, punainen valo varoittaa tulevasta taistelusta jne.

6.3 NPC:t

Sokkeloihin voisi asettaa vihollisia automaattisesti, ja tietenkin myös työkalun kautta. Sokkeloissa voisi olla vaikkapa tietyille vihollistyypeille omia huoneitaan, pomohuoneita, aarteenvahteja jne. Myös muut NPC:t (non-player character) kuin viholliset, kuten esimerkiksi kauppiaat, oppaat jne, voitaisiin samaan tapaan lisätä sokkeloihin.

6.4 Erikoissolut ja tilat

Pelkkä sokkelo saattaa pelinä helposti olla tylsä. Tämän vuoksi projektiin lisättiin myös muutama useammasta solusta koostuva huone. Kuitenkaan isompien tilojen lisäämistä automaattisesti sokkeloon ei ehditty toteuttaa, mutta työkalun kautta manuaalisesti se solu kerrallaan rakentamalla onnistuu.

7 Yhteenveto

Insinöörityössä pyrittiin luomaan pelinkehittäjille työkalu, jolla on mahdollista nopeasti ja tehokkaasti luoda sokkeloita. Sokkelolla tarkoitetaan rakennetta, jonka jokaisesta pisteestä toiseen on mahdollista kulkea vain yhtä reittiä pitkin, tosin tässä projektissa luotu työkalu mahdollistaa sokkeloiden muokkaamisen myös siten, että sokkeloon voidaan luoda oikoreittejä, isompia huoneita ja jopa täysin muusta sokkelosta eristettyjä alueita. Lisäksi on syytä huomata, että vaikka termiä labyrintti käytetään usein puhekielessä tarkoittamaan sokkeloa, todellisessa labyrintissa on olemassa vain yksi yksikäsitteinen reitti alusta loppuun. Tämä

projekti ei siis luo labyrintteja, vaan sokkeloita, ja edelleen näitä sokkeloita voidaan muokata niin, että ne eivät enää ole edes todellisia sokkeloita.

Projektin parissa työskentely vahvisti uskoa omaan ohjelmointitaitoon. Toki sekä opintojen että muun ohjelmointikokemuksen ohessa on tullut paljon käsiteltyä taulukoita, mutta tässä yhteydessä saatiin taulukot sisältämään kompaktissa muodossa yllättävän paljon dataa yllättävän vähällä koodilla. Mitään monimutkaisia rakenteita tai ratkaisuja ei projektin koodi sisällä, ja voisikin sanoa kehitystyössä noudatetun vahvasti ns. KISS-periaatetta, eli ”Keep It Simple, Stupid.” Kehitystyön ohessa testattiin koodia jatkuvasti, ja lopputuloksesta tuli jopa yllättävän virheettömästi toimiva. Lisäksi algoritmi toimii sangen nopeasti.

Sokkelogenerointi, projektin molempien algoritmien tapauksessa, perustui reitinhakualgoritmien soveltamiseen. Oli mielenkiintoista huomata, kuinka erilaisia sokkeloita algoritmit loivat.

Jos ja kun työkaluun lisätään reitinhaku sokkelon eri solujen välille, voisi siihen käyttää reitinhakuun tehokkaampaa algoritmia, esimerkiksi A*-algoritmia. Erityyppinen reitinhaku alkaa jo olemaan melkoisen tuttua, joten sen toteuttaminen varsinkin näin yksinkertaisessa tapauksessa tuskin tuottaisi vaikeuksia.

Matematiikan osa-alueena graafiteoria on sekä erilainen että kiehtova alue. Tämän projektin koodaaminen toimi varmasti hyvänä alkuharjoitteena teorian muuttamisessa koodiksi. Vaikka tässä yhteydessä ei graafiteoriaan kovinkaan syvälle tarvinnut paneutua, algoritmien pohja kuitenkin on suoraan sovelletussa graafiteoriassa.

Lähteet

Hella, Lauri. 2017. Johdatus graafiteoriaan. Luentomateriaali. Tampereen yliopisto, Luonnontieteiden tiedekunta.

Retro Dungeons: The Labyrinth. Verkkoaineisto. Zerin Labs. <https://assetstore.unity.com/packages/3d/environments/dungeons/retro-dungeons-the-labyrinth-157479>. Luettu 1.3.2023.

Unity Documentation. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/ScriptReference>. Luettu 1.3.2023.

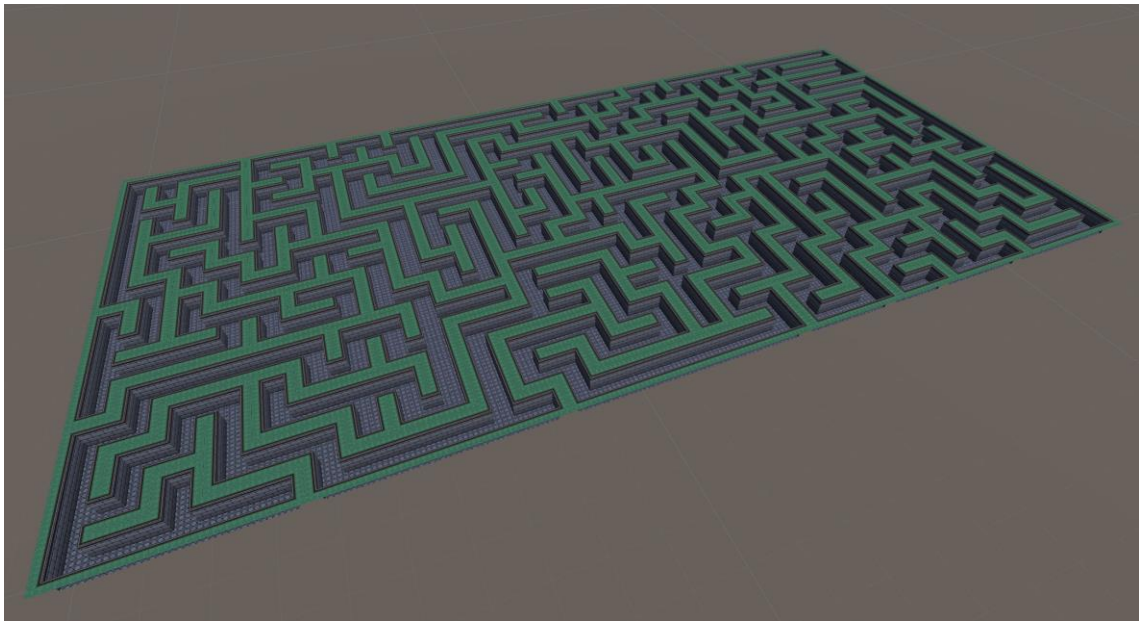
GitHub. Verkkoaineisto. GitHub, Inc. <https://github.com/>. Luettu 1.3.2023.

Github/gitignore. Verkkoaineisto. GitHub, Inc. <https://github.com/github/gitignore/blob/main/Unity.gitignore>. Luettu 1.3.2023.

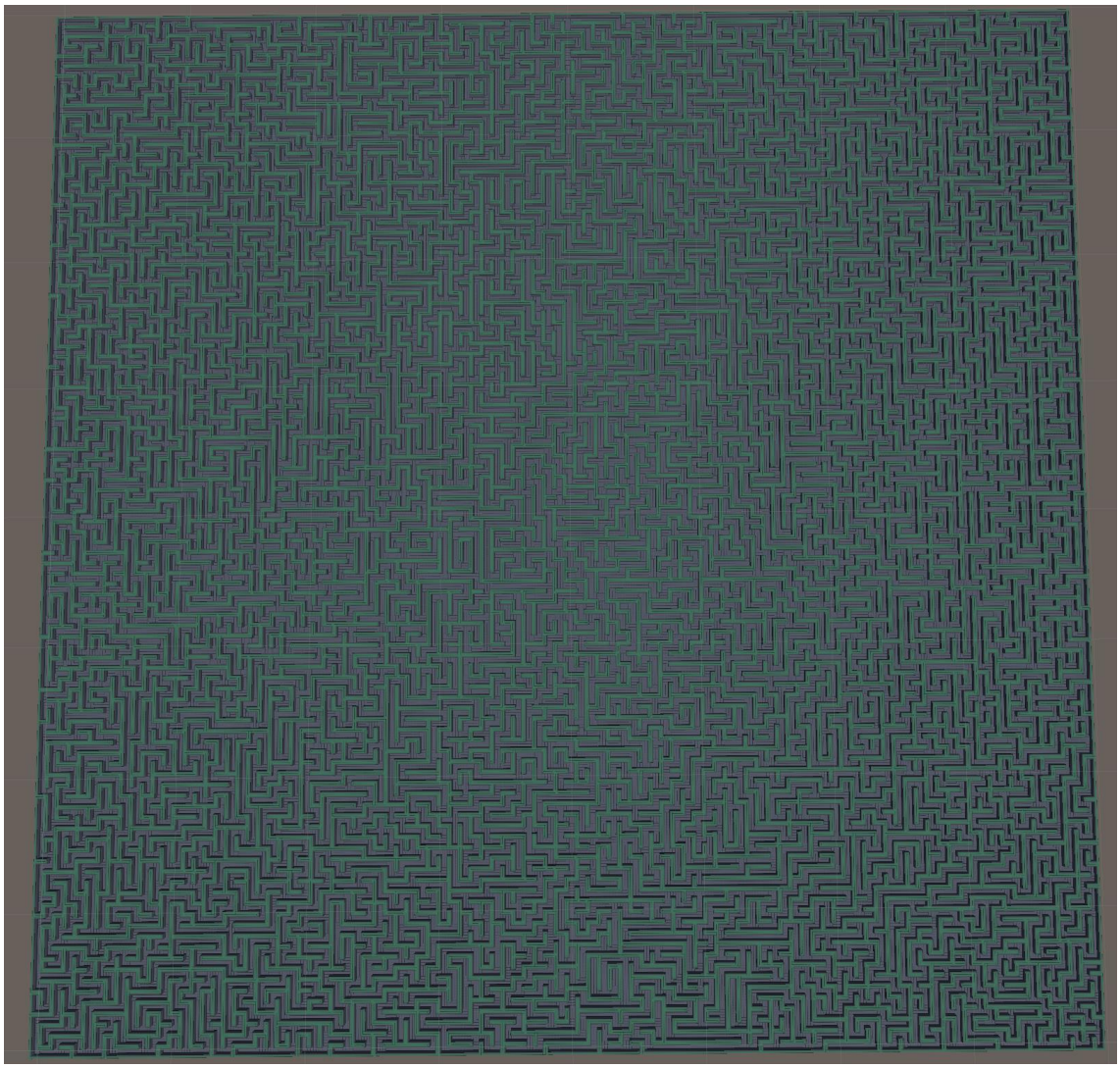
Maze Generation. Verkkoaineisto. Rosetta Code. <https://rosettacode.org/wiki/Maze_generation>. Luettu 1.3.2023.

Liitteet

Esimerkkejä algoritmin luomista sokkeloista



Kuva 15: 30*15-kokoinen sokkelo



Kuva 16: 100*100-kokoinen sokkelo