Jonas Montonen

Proseduraalisen luolan generointityökalu

Insinööri (AMK)
Tieto- ja viestintätekniikka
Kevät 2025

KAMK • University of Applied Sciences

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Jonas Montonen

Työn nimi: Proseduraalisen luolan generointityökalu

Tutkintonimike: Tietotekniikan ja viestinnän insinööri

Asiasanat: Proceduraalinen Generaatio, Unreal Engine, Pelikehitys

Tämän opinnäytetyön tavoitteena oli kehittää työkalu, jolla toteuttaa proseduraalisesti generoituvia luolaympäristöjä peliprojektiin ”Thalassophobia”. Työn toimeksiantajana oli Varattu Valo Games. Toteutetun työkalun tarkoitus oli olla helposti käytettävä ja muokattavissa erilaisien luolaympäristöjen ja huoneistojen tekemistä varten eri artistien ja koodarien käytössä.

Ensimmäinen askel työkalun suunnittelussa oli tutkia erilaisia algoritmeja ja proseduraalisia työskentelytapoja. Tämä vaihe käytettiin valmiiden proseduraalisten tuotteiden tutkimiseen Unreal Enginen sisällä sekä sen ulkopuolella. Erityistä huomiota kiinnitettiin siihen, miten menetelmät soveltuivat kolmiulotteisten ympäristöjen luomiseen reaaliaikaisesti pelimoottorissa. Toteutuspaikkana toimi Unreal Engine 5.4, ja kehitystyö suoritettiin keväällä 2025. Aineistona käytettiin peliprojektin alkuperäisiä visuaalisia resursseja ja testauksessa hyödynnettiin myös pelimoottorin omia valmisaineistoja. Metodina toimi ohjelmointiin perustuva kokeellinen kehitystyö, jossa ratkaisuja testattiin iteratiivisesti.

Lopullinen implementointi perustui erilaisiin proseduraalisiin menetelmiin, kuten Cellular Automata, Perlin Noise ja Marching Cubes, joiden yhdistelmä mahdollisti joustavan ja realistisen luolaympäristön luomisen. Eri algoritmeista säädettiin parametreja siten, että oli mahdollista saada aikaan ahtaita käytäviä, että väljempiä tiloja. Käyttöliittymä tehtiin myös yksinkertaiseksi, jotta itse käyttö ei tuottanut ongelmia edes eiteknisille käyttäjille. Työkalua testattiin ympäristöresursseilla, joita artistit loivat peliprojektin aikana sekä valmiilla resursseilla Unreal Enginesta, ja se vastasi sille asetettuja odotuksia.

Työkalu on luotettava ja joustava lisä peliprojektille. Se säästi aikaa suunnitteluvaiheessa ja mahdollisti nopeat prototyyppien luomiset. Tämä kaikki tuo sen myötä, että projekti suoritti onnistuneesti proseduraalisen lähestymistavan kaivavien ympyröiden luomiseen. Tällä on erityiset hyödyt, kun työskentelee pienen kehitystiimin kanssa. Työkalua voidaan hyödyntää jatkossa myös muissa projekteissa pienin muutoksin.

KAMK • University of Applied Sciences

Abstract

Author(s): Jonas Montonen

Title of the Publication: Procedural Generation of a Cave Environment

Degree Title: Bachelor of Engineering, Information and Communication Technology

Keywords: Procedural Generation, Unreal Engine, Game development

The goal of this thesis was to develop a tool for procedurally generating cave environments for the game project *Thalassophobia*. The client for this work was Varattu Valo Games. The tool was designed to be user-friendly and easily customizable for the creation of various types of cave environments and room-like spaces by both artists and programmers.

The first step in designing the tool was to research different algorithms and procedural workflows. This phase involved examining existing procedural solutions both within and outside Unreal Engine. Special attention was paid to how well these methods supported the real-time creation of three-dimensional environments within a game engine. The implementation was carried out in Unreal Engine 5.4 during the spring of 2025. The materials used included the game project's original visual assets, and testing also made use of Unreal Engine's own ready-made assets. The method was an experimental development process based on programming, where solutions were tested iteratively.

The final implementation was based on a combination of procedural techniques such as Cellular Automata, Perlin Noise, and Marching Cubes. This blend enabled the creation of flexible and realistic cave environments. The parameters of the different algorithms were adjusted to allow for the generation of both narrow corridors and more open spaces. The user interface was designed to be simple, ensuring usability even for non-technical users. The tool was tested with both custom environment assets created by the project's artists and ready-made assets from Unreal Engine, and it met the expectations set for it.

The tool proved to be a reliable and flexible addition to the game project. It saved time during the design phase and enabled rapid prototyping. As a result, the project successfully implemented a procedural approach to generating tunnel-like cave structures. This approach is especially beneficial when working with a small development team. With minor modifications, the tool can also be adapted for future projects.

Sisällys

Symboliluettelo

Blueprints: Unreal Enginen visuaalinen skriptikieli, joka mahdollistaa ohjelmoinnin ilman perinteistä koodia. Käytetään proseduraalisen generoinnin toteuttamiseen tässä projektissa. (Ks. 3.1 Unreal Engine ja sen mahdollisuudet)

Cellular Automata: Algoritmi, joka käyttää matriisia ja sääntöjä luonnollisten, orgaanisten muotojen (esim. luolastojen) generointiin. Tässä työssä käytetty luolaston seinien ja lattioiden tasoittamiseen. (Ks. 2.1 Algoritmit ja niiden valinta; 5.2 Luolaston arvojen smoothaus)

Determinismi: Algoritmin ominaisuus, jossa sama siemen (engl. seed) tuottaa aina saman lopputuloksen. Hyödyllinen pelimaailmojen debuggaamisessa ja jakamisessa. (Ks. 2.1 Algoritmit ja niiden valinta)

Hilapohjainen generointi (engl. Grid-based generation): Menetelmä, jossa luolaston rakenne määritellään ruudukon avulla eri arvojen perusteella. Käytetty tässä työssä Unreal Enginessä luolastojen luomiseen. (Ks. 3.1 Unreal Engine ja sen mahdollisuudet; 4 Menetelmän valinta ja suunnittelu)

Marching Cubes: 3D-pohjainen algoritmi, joka luo pyöreitä pinnanmuotoja ja monimutkaista geometriaa luolastoihin. Käytetty tässä työssä luolaston 3D-mallin generointiin. (Ks. 4.1 Menetelmien arviointi ja valintaperusteet; 5.4 Marching Cubes -algoritmi)

Perlin-noise (Perlin-kohina): Algoritmi, joka tuottaa pehmeitä ja jatkuvia siirtymiä, soveltuu luonnollisten maisemien (esim. vuorten, jokien) generointiin. Tässä työssä käytetty luolaston pohjan randomisointiin. (Ks. 2.1 Algoritmit ja niiden valinta; 5.1 C++ koodi ja luolaston generointi)

Simplex-noise: Perlin-noisen kehittyneempi versio, joka tuottaa epälineaarisia maisemia tehokkaammin. Soveltuu maastogenerointiin tässä työssä. (Ks. 2.1 Algoritmit ja niiden valinta)

Wave Function Collapse (WFC): Sääntöihin perustuva algoritmi, joka luo hallittuja rakenteita proseduraalisessa generoinnissa. Soveltuu esimerkiksi tyrmien ja ympäristöjen luomiseen. (Ks. 2.1 Algoritmit ja niiden valinta)

1 Johdanto

Proseduraalinen generointi on ollut videopeleissä jo pitkän aikaa ja tässä opinnäytetyössä keskitytään siihen, millaisia algoritmeja generointiin liittyy ja miten niitä voidaan hyödyntää Unreal Enginen sekä pelikehityksen kanssa.

Unreal Engine -pelimoottori on kehittynyt huomattavasti viime vuosina. Alun perin sitä käytettiin yksityisessä pelikehityksessä, mutta nykyään se on nykyään laajinten käytetty 3D-pelimoottori, josta hyötyvät niin peliteollisuus kuin elokuvateollisuus. Epic games on myös uusimmissa versioissa lisännyt proseduraalisia työkaluja, joita hyödynnetään tässä opinnäytetyössä.

Työn lopullisena tavoitteena oli kehittää työkalu, jolla saadaan aikaan vedenalaisia luolaympäristöjä peliin nimeltä Thalassophobia. Työkalu kehitettiin yritykselle Varattu Valo Games, joka käyttää työkalua hyödykseen yhdessä pelin kentistä, jossa pelin ympäristö sijoittuu vedenalaisen luolaston tutkimiseen.

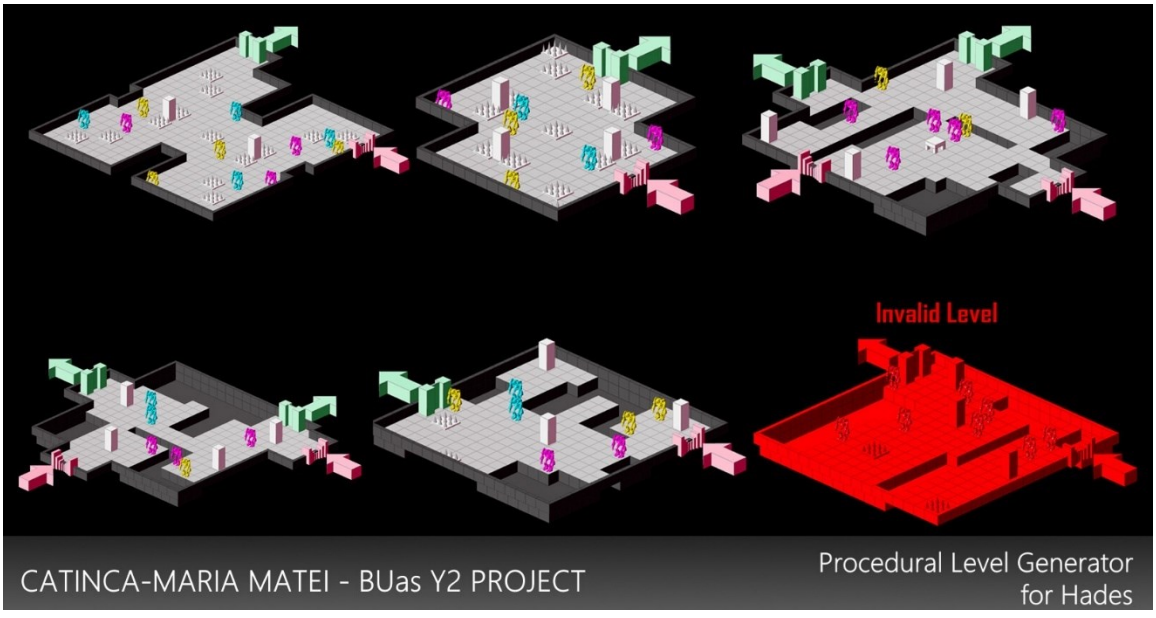
2 Proseduraalinen generointi

Proseduraalinen generointi tarkoittaa sisällön luomista algoritmien ja satunnaisuuden avulla. Tällä tyylillä voidaan luoda valtavia pelimaailmoja, ympäristöjä ja resursseja ilman, että jokaista yksityiskohtaa tarvitsee tehdä käsin. Tämä lähestymistapa on lähinnä suosittu pelikehityksessä. [1]

Peleissä proseduraalista generointia käytetään esimerkiksi maailmojen tai vankiluolastojen luomiseen. Tunnettuja esimerkkejä generoinnista ovat Minecraft, No Man's Sky ja Valheim, jotka käyttävät generointialgoritmeja maailmojen luomiseen (Kuva 1). Lisäksi myös roguelike-peleissä, kuten *The Binding of Isaac* (Kuva 2) tai *Hades* (Kuva 3), generoidut kentät saavat jokaisen pelikerran tuntumaan erilaiselta.

KUVA 1. Esimerkkipelejä proseduraalisessa generoinnissa

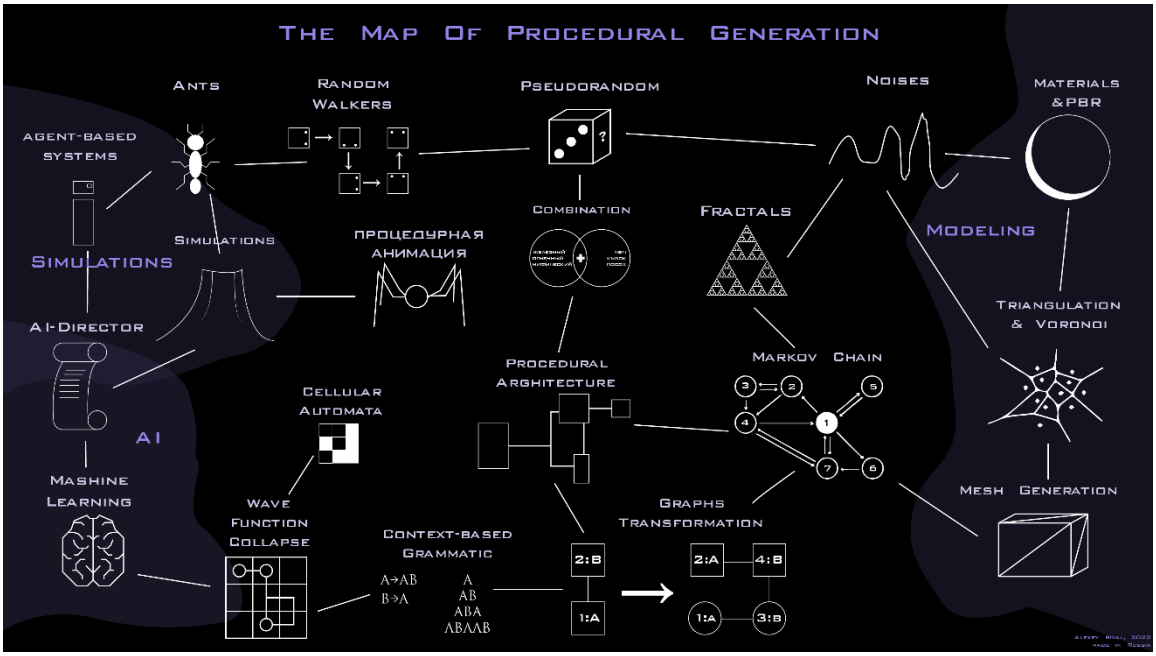
KUVA 2. The Binding of Isaacin vankityrmän generaatio. Kuva visualisoi huoneistojen sijoittelua ja miten ympäristö rakentuu vihreän aloitusalueen ympärille.



KUVA 3. Hades-pelin proseduraalinen generointi. Kuva visualisoi miten huoneistot rakentuvat uuteen huoneeseen siirryttäessä ja mitkä huoneistot eivät mene algoritmin läpi.

Proseduraalisen generoinnin vahvuuksia ovat sen helppo skaalautuvuus ja kyky tuottaa monipuolista ja loputonta sisältöä ilman valtavaa työmäärää. Tämä tekee siitä loistavan työkalun avoimen maailman peleihin ja pienemmille kehitystiimeille. Toisaalta haasteita riittää: generoidun sisällön laatua voi olla vaikea hallita, ja jos randomisoinnit eivät ole tarpeeksi monipuolisia, pelaaja saattaa huomata toistuvuutta. [1]

Algoritmien kehitys on myös edennyt reilusti eteenpäin. Nykyisin hyödynnetään paljon esimerkiksi Perlin-noisea, Cellular Automataa ja Marching Cubesia, joilla syntyy realistisia ja monimutkaisia ympäristöjä. Kehittyneimmät menetelmät osaavat jopa mukauttaa sisältöä pelaajan toimintaan ja pelin kontekstiin, mikä tekee pelikokemuksesta vielä astetta dynaamisemman (Kuva 4). [1]

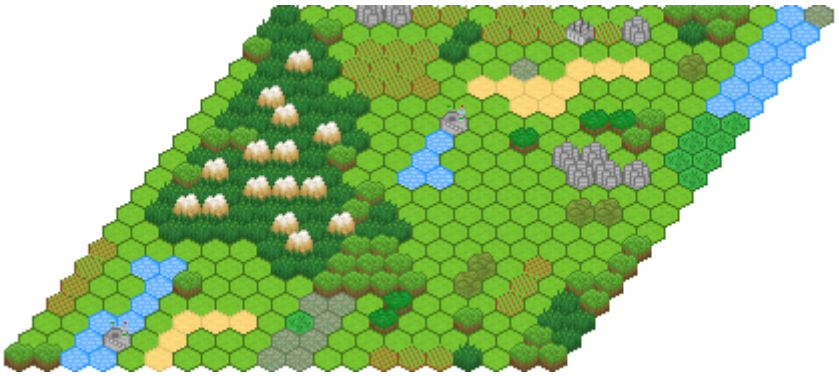


KUVA 4. Erilaisia generaatiosysteemejä

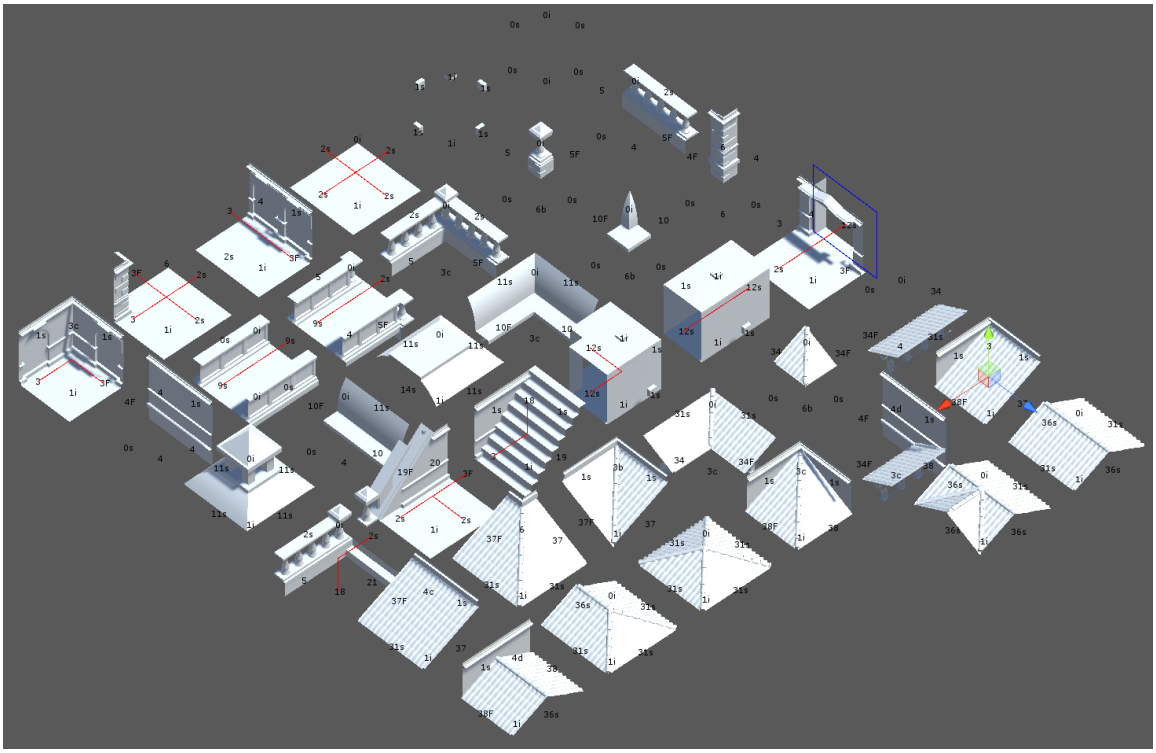
2.1 Algoritmit ja niiden valinta

Proseduraalisen generoinnin ydin ovat algoritmit, jotka määräävät, minkälaista sisältöä generoidaan. Algoritmien valinta ja käyttö ovat tärkein osa, sillä eri algoritmeilla on erilaisia vahvuuksia ja heikkouksia. Esimerkiksi perinteiset numerogeneraattorit voivat tuottaa simppeliä ja nopeaa sisältöä, kun taas Perlin-noise toimii paremmin luonnollisten ja monimutkaisten ympäristöjen luomiseen. [1]

Valinnassa tärkeää on myös tehokkuus ja skaalautuvuus. Yksinkertaisilla tyyleillä voidaan tuottaa nopeasti suuria määriä dataa, mutta niillä ei monesti saada tarpeeksi tarkkuutta. Vaihtoehtoisesti L-systeemit tai Wave Function Collapse (WFC) avaavat sääntöihin perustuvaa generointia, mikä tuottaa hallitumpia rakenteita valmiilla paloilla [2]. Näin saadaan aikaan esimerkiksi paloista rakennettuja ympäristöjä (Kuva 5) (Kuva 6) [3].

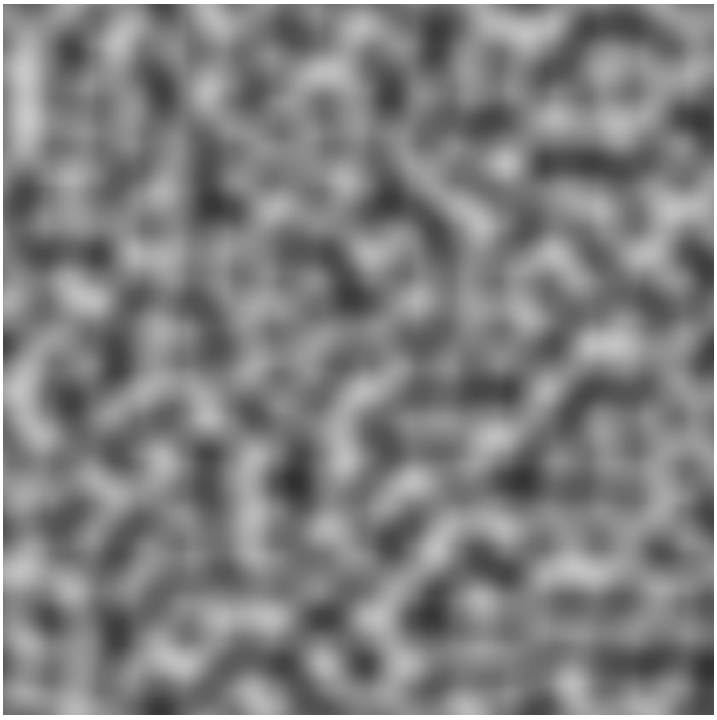


KUVA 5. Wave function collapse -ympäristö

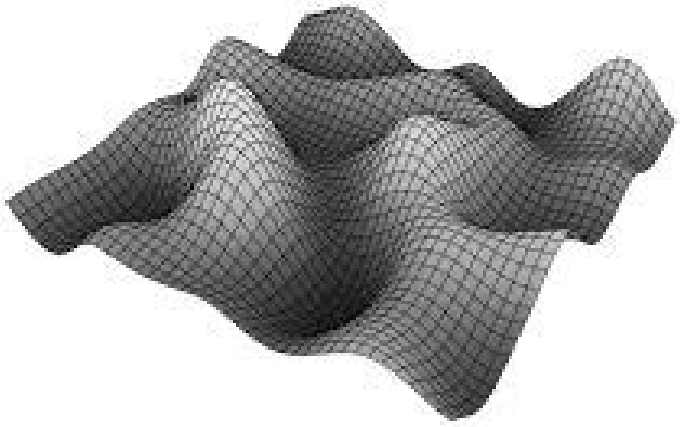


KUVA 6. Wave function collapse -vankityrmä

Maastojen ja ympäristöjen luomisessa suosittuja työkaluja ovat Perlin- ja Simplex-noise, joilla voi tehdä epälineaarisia maisemia, kuten vuoristoja tai jokia (Kuva 7) (Kuva 8) [4]. Voronoi-diagrammit taas toimivat hyvin karttojen aluejakoihin tai luonnollisten elinympäristöjen simulointiin [5]. Cellular Automata sopii erityisesti luolastoihin, koska se luo luonnollisen näköisiä, toisiinsa liittyviä tiloja [6]. Pelikehityksessä on tärkeää, että algoritmi tukee toistettavuutta eli determinismiä. Silloin sama siemen tuottaa aina saman tuloksen, mikä helpottaa esimerkiksi pelimaailmojen testaamista tai pelaajien välistä maailmojen jakamista [2].



KUVA 7. 2D-esimerkki Perlin noisesta



KUVA 8. 3D-esimerkki Perlin noisesta

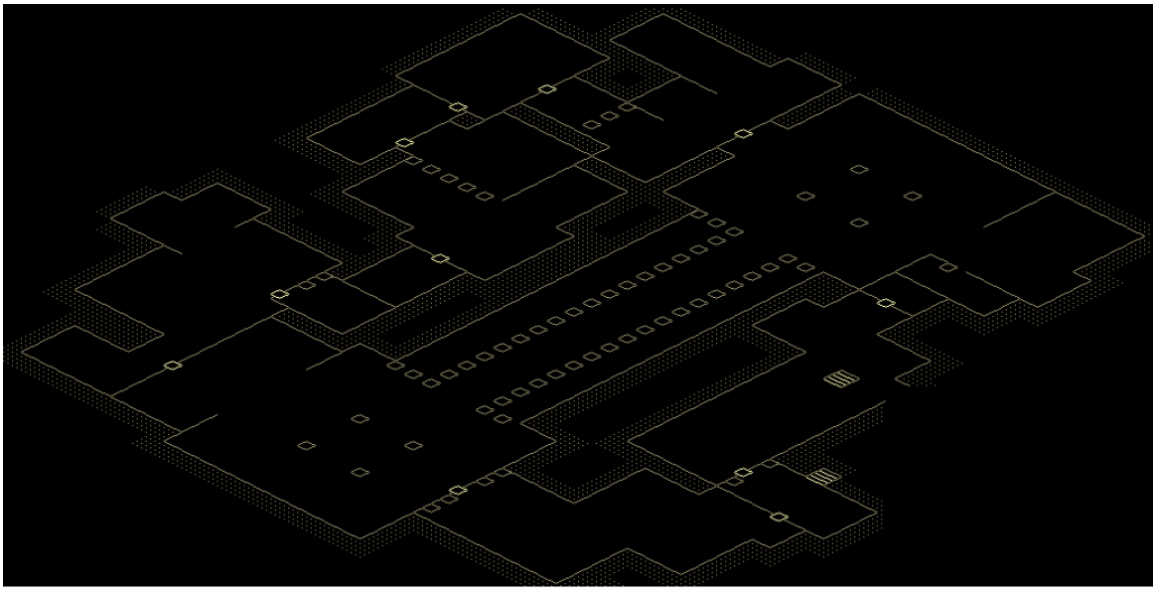
Lopulta algoritmin valinta riippuu pelin tarpeista. Halutaanko täysin satunnainen maailma, joka yllättää joka kerta, vai enemmän hallittua mutta proseduraalisesti luotua sisältöä? Näihin kysymyksiin vastaaminen ohjaa oikean menetelmän valintaan ja varmistaa, että lopputulos on sekä visuaalisesti näyttävä että teknisesti toimiva.

2.2 Käyttötyylit ja sovellukset

Proseduraalisia ympäristöjä hyödynnetään erityisesti roguelike- ja avoimen maailman peleissä, joissa dynaamiset ympäristöt pitävät pelin kiinnostavana [7]. Generointiin voi käyttää erilaisia tekniikoita riippuen siitä, millaista tunnelmaa ja kokemusta haetaan.

Täysproseduraalinen generointi luo kaiken alusta asti algoritmien varassa ilman valmiita palikoita. Se sopii hyvin peleihin, joissa ympäristö muuttuu jatkuvasti, kuten Minecraftin luolajärjestelmässä, jossa maan alla avautuu satunnaisia muotoja ja rakenteita [8]. Cellular Automata on tässä tyylissä yleinen valinta, koska se jäljittelee luonnollisia, orgaanisia muotoja [6]. Perlin- ja Simplex-kohina taas tuovat realismia maaston muotoihin ja syvyyseroihin [4].

Puoliproseduraalinen generointi yhdistää käsin tehdyt osat ja satunnaisuuden. Esimerkiksi Spelunky-pelissä luolastot rakennetaan valmiista huonepohjista, jotka järjestellään sitten sattumanvaraisesti, jotta pelikokemus pysyy tarpeeksi hallittuna [9]. Sääntöpohjainen generointi taas asettaa tarkat raamit luolastojen rakenteille. Esimerkiksi jokaisesta osasta pitää päästä ulos, huoneet yhdistetään käytävillä tai tietyt alueet varataan tietyille haasteille. Diablon kaltaiset ARPG-pelit käyttävät tätä tyyliä luodakseen loogisia mutta vaihtelevia tasoja (Kuva 9) [7].



KUVA 9. Diablo 1 -pelin vankityrmän generointi

Tyylejä voi myös yhdistellä. Dead Cells yhdistää puoliproseduraalista ja sääntöpohjaista otetta: alueet rakennetaan satunnaisista moduuleista, mutta eteneminen ja vaikeustaso noudattavat tiettyjä sääntöjä (Kuva 10) [10].

Luolien generointityyli riippuu pelin luonteesta, immersion tasosta ja pelaajalle tarjotusta haasteesta. Dynaaminen ympäristö voi parantaa uudelleenpelattavuutta, mutta täysin satunnainen rakenne voi joskus pilata kokemuksen – siksi proseduraalisten pelien suunnittelu vaatii järkeä ja harkintaa.

KUVA 10. Dead cells -esimerkki osittaista proseduraalisesta generoinnista

3 Unreal engine ja sen mahdollisuudet

Proseduraalinen generointi vaatii tehokkaita ja joustavia työkaluja. Oikeiden välineiden valinta on tärkeää, sillä se myös vaikuttaa sekä prosessin sujuvuuteen että lopullisen pelimaailman laatuun [11]. Tässä luvussa tutustutaan siihen, millaisia työkaluja opinnäytetyössä käytettiin.

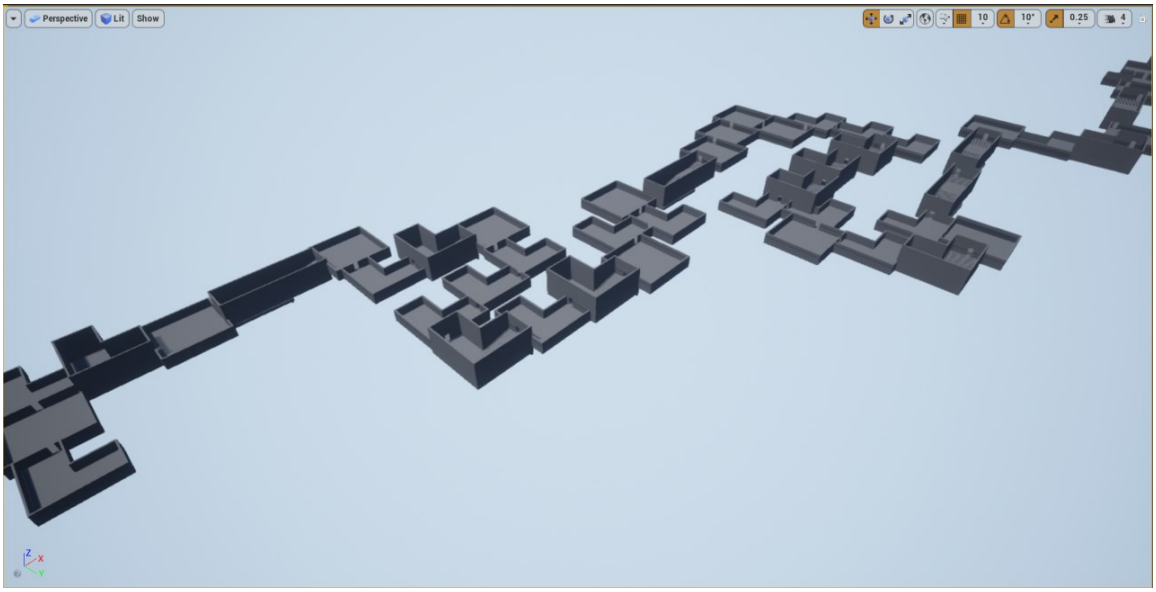
Unreal Engine on yksi pelialan parhaista pelimoottoreista, ja sen monipuoliset ominaisuudet tekevät siitä täydellisen alustan proseduraaliseen generointiin [12]. Graafiset työkalut, integroidut järjestelmät ja vahva tuki proseduraaliselle sisällölle mahdollistavat monimutkaisten ja elävien pelimaailmojen luomisen. Tässä osiossa katsotaan tarkemmin, miten Unreal Engineä hyödynnettiin tässä projektissa ja mitä se tarjosi vedenalaisten luolaympäristöjen generointiin.

Yksi Unrealin isoimmista valteista on Procedural Content Generation (PCG), joka antaa mahdollisuuden luoda dynaamisia ympäristöjä ilman, että jokaista osaa tarvitsee asetella käsin (Kuva 11) [13]. PCG-työkalulla voi määritellä säännöt ja rajat, joiden puitteissa luolastot ja muut ympäristöt syntyvät. Tämä säästää aikaa ja pitää pelimaailman tuoreena ja vaihtelevana.

KUVA 11. Unreal engine 5.5 -PCG

Toinen tärkeä työkalu on Unrealin skriptikieli Blueprints, joka tarjoaa visuaalisen tavan ohjelmoida ilman, että tarvitsee kirjoittaa perinteistä koodia. Uusimmissa unrealin versioissa Blueprint-kieltä voidaan käyttää suurimpaan osaan asioista, nykyisten päivitysten seurauksena [14]. Aikaisemmin blueprinttien käyttö hidasti koodin kääntymistä ja aiheutti muita pieniä ongelmia.

Tässä projektissa Unrealia käytettiin luolien generointiin hilapohjaisella (engl. grid-based) lähestymistavalla. Algoritmit määrittelivät luolastojen muodot eri arvojen perusteella, mikä varmisti, että lopputulos ei ollut liian samanlaista (Kuva 12) [15].



KUVA 12. Proseduraalinen vankityrmä Unreal Enginessä

Cellular Automata -algoritmi tuli myös käyttöön: sen avulla luolien muotoja hiottiin ja tasapainotettiin, jotta ne olivat pelattavia eivätkä liian kaoottisia [6]. Unrealin työkalujen ansiosta proseduraalinen generointi hoitui tehokkaasti ja skaalautuvasti. Lopputuloksena syntyi isoja, monipuolisia luolaympäristöjä ilman, että jokaista osaa tarvitsi itse rakentaa.

4 Menetelmän valinta ja suunnittelu

Luolien proseduraaliseen generointiin on monia tapoja, mutta tässä projektissa keskittyminen oli dynaamisissa ja vaihtelevissa ympäristöissä. Tässä luvussa käydään läpi eri tekniikoiden kykyä luoda monipuolisia ja pelattavia luolaympäristöjä ja valitsen opinnäytetyöhön sopivimmat algoritmit [11].

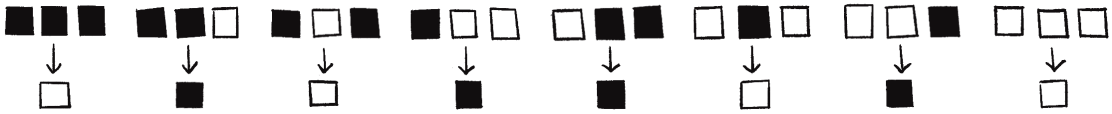
Tarkastelin erityisesti Cellular Automataa, Perlin-kohinaa, Marching Cubesia ja hila-pohjaista generointia, jotka sopivat parhaiten tähän aiheeseen (Kuva 13).

KUVA 13. Esimerkki proseduraalisesta luolasta

4.1 Menetelmien arviointi ja valintaperusteet

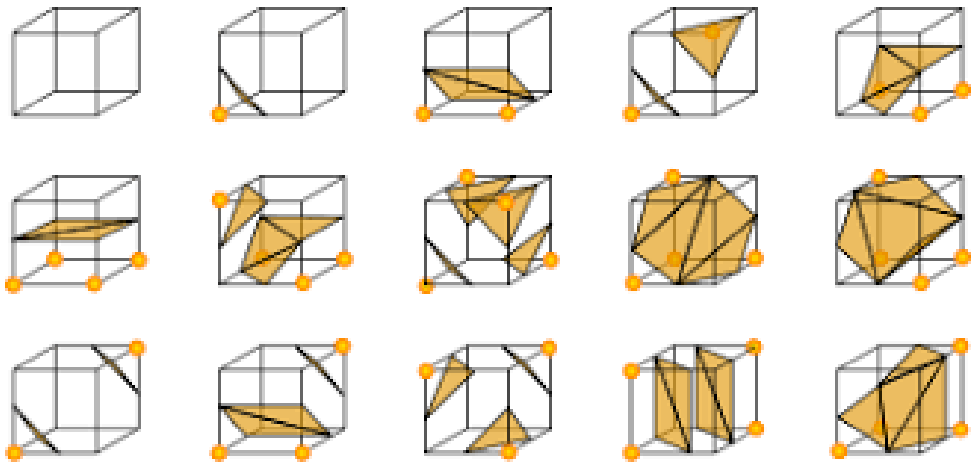
Ensimmäisenä tarkastelussa oli Perlin-kohina, joka luo pehmeitä ja jatkuvia siirtymiä luolastoihin. Se tuottaa visuaalisesti kivoja ja saumattomia ympäristöjä pseudosatunnaisten arvojen avulla [1]. Perlin-kohinan vahvuus on sujuvissa maisemissa, mutta pelattavuuden kannalta se ei yksin riitä, sillä ympäristöt voivat jäädä liian tasaisiksi tai hankaliksi navigoida. Siksi sitä yleensä yhdistetään muihin tekniikoihin, jotta lopputulos on tarpeeksi monipuolinen ja toimiva.

Seuraavaksi Cellular Automata, joka on loistava luomaan luonnollisia ja hieman sekavia luolastoja. Se toimii matriisipohjaisesti, ja solut kehittyvät tiettyjen sääntöjen mukaan (Kuva 14) [6]. Tuloksena on orgaanisia muotoja, jotka muistuttavat aitoja luolia. Tämä tekniikka sopii mainiosti Perlin-kohinan pariksi, sillä se tuo lisää satunnaisuutta ja elävyyttä.



KUVA 14. Cellular Automatan toiminta datan käsittelyssä

Lopuksi Marching Cubes on 3D-pohjainen lähestymistapa, joka mahdollistaa pyöreiden pinnanmuotojen luomisen luolastoon [16]. Se on erityisen hyvä luodessa monimutkaisia 3D-ympäristöjä, mutta vaatii enemmän laskentatehoa, mikä rajoittaa sen käyttöä suuremmissa pelimaailmoissa. Eli marching Cubes on erinomainen, kun tarvitaan yksityiskohtaisia ja realistisia maastomalleja, mutta se saattaa olla laskennallisesti liian raskas suurempiin ympäristöihin.



KUVA 15. Marching Cubes -toiminta ja miten algoritmi rakentaa paloja ympäristöön datan perusteella.

4.2 Lopullinen valinta

Lopullinen käyttötyylien valinta keskittyi eri algoritmien sekoitukseen. Ensimmäisenä käytetään Perlin noisea antamaan randomisoitu pohja luolaston generoinnille, minkä jälkeen Cellular Automata -algoritmia käytetään datan paranteluun, jotta saadaan aikaan luonnollisemman oloinen luolasto. Lopuksi käytettiin Marching Cubes -algoritmia itse proceduraalisen meshin luomiseksi (Kuva 16).

```
// Generate noise for cave structure
float RoomNoiseValue = FMath::PerlinNoise3D(NoiseInput

// Smooth the map using Cellular Automata rules
for (int i = 0; i < SmoothIterations; i++)
{
    SmoothMap();
}

// Generate the mesh based on map data using Marching
GenerateMeshWithMarchingCubes();
```

KUVA 16. Valitut algoritmit

5 Toteutusprosessi

Tässä osiossa käsitellään luolaston generointiin käytettyjä tyylejä ja täytäntöönpanoja, kuten Perlin-noisen datageneroitia. Lopullinen tavoite oli joka vaiheessa saada generoitua käyttäjäystävällinen systeemi, jolla saadaan iteroitua monta versiota vaihtamalla parametrejä systeemistä.

Luolaston generointi perustuu kolmeen päävaiheeseen: Perlin-noisen käytöstä luolaston alustamiseen, cellular automatan smoothaukseen, sekä lopuksi Marching Cubes -algoritmin avulla geometrian luomiseksi. Näiden vaiheiden avulla luodaan monipuolisia ja vaihtelevia luolastomaisemia, joissa on myös luonnollisia piirteitä.

5.1 C++ -koodi ja luolaston generointi

Aluksi luolaston data alustetaan käyttämällä Perlin-noisea, joka antaa randomisoidut arvot luolaston generointia varten. Se siis määrittää, mitkä alueet ovat maa-alueita ja mitkä jäävät tyhjiksi. Alla oleva koodi käyttää luotua 3D-perlin-funktiota ja antaa erilaiset arvot ympäristölle (Kuva 17).

```
// Initialize the map with Perlin noise
for (int x = 0; x < MapWidth; x++)

{
    for (int y = 0; y < MapHeight; y++)
    {
        for (int z = 0; z < MapDepth; z++)
        {
            // Create a solid "seafloor" layer at a specific height
            int SeafloorHeight = MapDepth; // You can adjust this value as needed
            if (z >= SeafloorHeight)
            {
                Map[x + y * MapWidth + z * MapWidth * MapHeight] = true;
                continue;
            }

            // Apply the seed to the noise function
            FVector NoiseInput = FVector(x, y, z) * NoiseFrequency + Seed;

            // Generate noise for cave structure
            float RoomNoiseValue = FMath::PerlinNoise3D(NoiseInput) * NoiseAmplitude;

            // Additional noise for noodle-like tunnels
            FVector TunnelNoiseInput = FVector(x, y, z) * NoiseFrequency * 2.5f + Seed;
            float TunnelNoiseValue = FMath::PerlinNoise3D(TunnelNoiseInput) * NoiseAmplitude * 0.4f;

            // Combine the noise layers to determine if this voxel is filled
            float CombinedNoiseValue = RoomNoiseValue - TunnelNoiseValue;

            if (CombinedNoiseValue < FillPercentage)
            {
                Map[x + y * MapWidth + z * MapWidth * MapHeight] = true;
            }
        }
    }
}
```

KUVA 17. Perlin noise -funktio

5.2 Luolaston arvojen pehmentäminen

Kun perusarvot on luotu, käytetään seuraavaksi arvojen tasoittamiseen tarkoitettua algoritmia parantamaan luolaston ulkonäköä ja toiminnallisuutta. Tähän käytetään Cellular Automataa, joka muuttaa ympäristöä, siten että luolastojen seinät ja lattiat saavat pehmeämmän ja realistisemman muodon. Tämä myös varmistaa, että luolastossa ei ole liian teräviä kulmia tai epätasaisuuksia (Kuva 18).

```cpp
// Function to apply Cellular Automata smoothing
void ANoiceBasedCaves::SmoothMap()
{
    TArray<bool> NewMap = Map;  // Copy current map

    for (int x = 0; x < MapWidth; x++)
    {
        for (int y = 0; y < MapHeight; y++)
        {
            for (int z = 0; z < MapDepth; z++)
            {
                // Ensure outer walls remain solid
                if (x == 0 || x == MapWidth - 1 ||
                    y == 0 || y == MapHeight - 1 ||
                    z == 0 || z == MapDepth - 1)
                {
                    NewMap[x + y * MapWidth + z * MapWidth * MapHeight] = true;
                    continue;  // Skip smoothing for outer walls
                }

                int NeighborWallCount = GetNeighborWallCount(x, y, z);

                // Noise-based smoothing influence
                float RoomNoiseValue = FMath::PerlinNoise3D(FVector(x, y, z) * NoiseFrequency) * NoiseAmplitude;
                float TunnelNoiseValue = FMath::PerlinNoise3D(FVector(x, y, z) * (NoiseFrequency * 2.5f)) * (NoiseAmplitude * 0.4f);

                float NoiseBias = FMath::Clamp(RoomNoiseValue + TunnelNoiseValue, -1.0f, 1.0f);

                // Adjust thresholds dynamically based on noise
                int WallThreshold = 4 + (NoiseBias > 0 ? 1 : -1);

                if (NeighborWallCount > WallThreshold)
                {
                    NewMap[x + y * MapWidth + z * MapWidth * MapHeight] = true;
                }
                else if (NeighborWallCount < WallThreshold)
                {
                    NewMap[x + y * MapWidth + z * MapWidth * MapHeight] = false;
                }
            }
        }
    }

    Map = NewMap;  // Apply the smoothed map
}
```

KUVA 18. Cellular automatan käyttö

5.3 Luolaston sisäänkäynnit ja tunnelit

Erityisesti sisäänkäyntien luominen luolastoon on tärkeää, jotta pelaajat löytävät helposti reittejä luolastoon. Tämä saavutetaan lisäämällä luolaston yläosaan erillinen vaihe, joka poistaa osan luolasta ja luo paremman sisäänkäyntialueen, joka sitten muovautuu muuhun ympäristöön (Kuva 19).

```
// Carve out entrances at these points
for (FVector Entrance : EntrancePoints)
{
    int EntranceRadius = 5;  // Radius of the entrance

    for (int x = Entrance.X - EntranceRadius; x <= Entrance.X + EntranceRadius; x++)
    {
        for (int y = Entrance.Y - EntranceRadius; y <= Entrance.Y + EntranceRadius; y++)
        {
            for (int z = Entrance.Z - EntranceRadius; z < MapDepth - 18; z++)  // Carve all the way to the top
            {
                // Ensure we're within bounds and carve out space
                if (x >= 0 && x < MapWidth && y >= 0 && y < MapHeight && z >= 0 && z < MapDepth)
                {
                    Map[x + y * MapWidth + z * MapWidth * MapHeight] = false;
                }
            }
        }
    }
```

KUVA 19. Luolaston päällisosan muokkaus

5.4 Marching Cubes -algoritmi

Lopuksi luolaston oikean geometrian luomiseen käytetään Marching Cubes -algoritmia, joka on tehokas tapa muuntaa luolan rakenne kolmiulotteiseksi geometriaksi. Tämä algoritmi tarkastelee kaikkia alueita luolaston sisällä ja määrittää, milloin ja miten luolassa pitäisi olla seinä tai käytävä. Tämä luo järkevämpiä geometrian muotoja kuin perinteinen kuutioiden järjestäminen (Kuva 20) (Kuva 21).

```
for (int z = 0; z < MapDepth - 1; z++)
{
    // Get the cube configuration based on the voxel data
    int CubeIndex = 0;
    if (Map[x + y * MapWidth + z * MapWidth * MapHeight]) CubeIndex |= 1;
    if (Map[(x + 1) + y * MapWidth + z * MapWidth * MapHeight]) CubeIndex |= 2;
    if (Map[(x + 1) + (y + 1) * MapWidth + z * MapWidth * MapHeight]) CubeIndex |= 4;
    if (Map[x + (y + 1) * MapWidth + z * MapWidth * MapHeight]) CubeIndex |= 8;
    if (Map[x + y * MapWidth + (z + 1) * MapWidth * MapHeight]) CubeIndex |= 16;
    if (Map[(x + 1) + y * MapWidth + (z + 1) * MapWidth * MapHeight]) CubeIndex |= 32;
    if (Map[(x + 1) + (y + 1) * MapWidth + (z + 1) * MapWidth * MapHeight]) CubeIndex |= 64;
    if (Map[x + (y + 1) * MapWidth + (z + 1) * MapWidth * MapHeight]) CubeIndex |= 128;

    // Check if the cube is entirely inside or outside the surface
    if (CubeIndex == 0 || CubeIndex == 255) continue;
```

KUVA 20. Pieni osa Marching Cubes -funktiosta

```
// Generate the vertices for the cube using interpolation
FVector Verts[12];
if (EdgeTable[CubeIndex] & 1) Verts[0] = VertexInterpolation(FVector(x, y, z)
if (EdgeTable[CubeIndex] & 2) Verts[1] = VertexInterpolation(FVector(x + 1, y
if (EdgeTable[CubeIndex] & 4) Verts[2] = VertexInterpolation(FVector(x + 1, y
if (EdgeTable[CubeIndex] & 8) Verts[3] = VertexInterpolation(FVector(x, y + 1
if (EdgeTable[CubeIndex] & 16) Verts[4] = VertexInterpolation(FVector(x, y, z
if (EdgeTable[CubeIndex] & 32) Verts[5] = VertexInterpolation(FVector(x + 1, 
if (EdgeTable[CubeIndex] & 64) Verts[6] = VertexInterpolation(FVector(x + 1, 
if (EdgeTable[CubeIndex] & 128) Verts[7] = VertexInterpolation(FVector(x, y +
if (EdgeTable[CubeIndex] & 256) Verts[8] = VertexInterpolation(FVector(x, y, 
if (EdgeTable[CubeIndex] & 512) Verts[9] = VertexInterpolation(FVector(x + 1,
if (EdgeTable[CubeIndex] & 1024) Verts[10] = VertexInterpolation(FVector(x +
if (EdgeTable[CubeIndex] & 2048) Verts[11] = VertexInterpolation(FVector(x, y
```

KUVA 21. Sääntö osa Marching Cubes funktiota

Tässä vaiheessa algoritmi käyttää luotua karttaa (engl. Map), jossa on data luolan rakenteeseen, joka luo siitä 3D-mallin, joka sitten näkyy Unreal Enginen 3D-maailmassa.

6 Tulokset ja analyysi

Tässä luvussa käsitellään työn lopputuloksia sekä analysoidaan tavoitteiden onnistumista. Tulokset perustuvat käytännön testaukseen sekä projektin aikana kerättyyn palautteeseen. Analyysissä arvioidaan suorituskykyä, luotujen ympäristöjen laatua sekä käytännöllisyyttä. Lisäksi pohditaan mahdollisia parannuskohtia ja jatkokehitystarpeita, joita työkalun jatkokäyttö vaatii.

6.1 Toteutuksen arviointi ja kokemukset

Toteutuksen arvioinnissa keskitytään siihen, kuinka hyvin työkalu täyttää sille asetetut pelilliset vaatimukset. Työn tavoitteena oli luoda monipuolisia ja mielenkiintoisia luolastoja, jotka tukevat peliprojektin tarpeita.

Generointiprosessin nopeus: Testausten perusteella luolaston generointi onnistui järkevässä ajassa, vaikka hyvin monimutkaisten luolastojen luominen vaati huomattavasti enemmän resursseja. Suorituskyvyn parantaminen hienosäätämällä käytettyjä algoritmeja voisi olla tarpeen, mikäli työkalua haluttaisiin käyttää laajemmissa alueissa.

Luotujen luolaympäristöjen monimuotoisuus: Perlin-noise sekä arvojen pehmennys antoi luolastoille kiinnostavan muodon ja vaihtelevaa maastoa, joka tuntui luonnolliselta. Eri alueilla luolastot saatiin tuntumaan erilaisilta, mikä loi mielenkiintoista peliympäristöä. Kuitenkin joidenkin alueiden yksityiskohdat olisivat voineet olla tarkempia, erityisesti sisäänkäyntien ja kapeiden käytävien osalta.

6.2 Jatkokehitys ja pohdinta

Projektin aikana opin käytännössä, kuinka monimutkaisten algoritmien yhdistäminen vaikuttaa pelimaailman rakenteeseen ja pelikokemukseen. Opin myös, kuinka tärkeää on tehdä työkalusta helposti käytettävä, jotta se tukee koko pelitiimiä – eikä vain ohjelmoijia. Lisäksi opin iteratiivisen kehittämisen ja palautteen merkityksen ohjelmistokehityksessä.

Proseduraalisen työkalun kehittäminen oli mielenkiintoinen ja opettavainen prosessi. Kuitenkin testauksessa nousi esiin joitain parannuskohtia ja jatkokehitystarpeita, jotka voisivat viedä työkalua entistä pidemmälle ja parantaa sen toimivuutta käytännössä.

Suorituskyvyn optimointi oli pahin ongelma, jossa parannuksia voitaisiin tehdä. Erityisesti suurilla pelialueilla luolastojen generointi hidastaa pelin käynnistymistä ja huonontaa siten käyttäjä kokemusta. Tämänhetkinen iteraatio vie noin 10 sekuntia tai enemmän generoinnissa.

Toinen kehityskohde on kasviston ja esineiden lisääminen, jotka antaisivat luolastoille elävyyttä ja monimuotoisuutta. Tällä hetkellä työkalu keskittyy pääasiassa luolaston luomiseen, mutta kasvien ja muiden ympäristön generointi on osa, joka pitää myöhemmin lisätä systeemiin.

Kaiken kaikkiaan työkalun kehittäminen eteni hyvin, mutta se jää vielä avoimeksi jatkokehitykselle. Proseduraalinen generointi tulee varmasti olemaan osa tulevia projektejani, ja uskon, että kehitystyötä voidaan jatkaa entistä isommilla alueilla.

Lähteet

[1] Smith, J. (2024). Procedural Generation: A Primer for Game Devs. Game Developer. Saatavilla 18.3.2025 https://www.gamedeveloper.com/design/procedural-generation-a-primer-for-game-devs

[2] Shaker, N., Togelius, J., & Nelson, M. J. (2016). Procedural Content Generation in Games. Springer. Saatavilla 18.3.2025 https://link.springer.com/book/10.1007/978-3-319-42716-4

[3] Kleineberg, M. (n.d.). Wave Function Collapse. GitHub repository. Saatavilla 18.3.2025 https://github.com/mxgmn/WaveFunctionCollapse

[4] Miller, G. S. P. (1986). The Definition and Rendering of Terrain Maps. ACM SIGGRAPH Computer Graphics. Saatavilla 18.3.2025 https://dl.acm.org/doi/10.1145/15922.15910

[5] Gurney, J. (2017). Procedural Generation with Voronoi Diagrams. Game Developer (ent. Gamasutra). Saatavilla 18.3.2025 https://www.gamedeveloper.com/programming/procedural-generation-with-voronoi-diagrams

[6] Johnson, L., Yannakakis, G. N., & Togelius, J. (2010). Cellular Automata for Real-time Generation of Infinite Cave Levels. Proceedings of the 2010 Workshop on Procedural Content Generation in Games. Saatavilla 18.3.2025 https://dl.acm.org/doi/10.1145/1814256.1814266

[7] Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). Procedural Content Generation for Games: A Survey. ACM Transactions on Multimedia Computing, Communications, and Applications. Saatavilla 18.3.2025 https://dl.acm.org/doi/10.1145/2422956.2422957

[8] Minecraft Wiki. (n.d.). Generated Structures. Saatavilla 18.3.2025 https://minecraft.fandom.com/wiki/Generated_structures

[9] Yu, D. (n.d.). Spelunky's Procedural Principles. Game Developer (ent. Gamasutra). Saatavilla 18.3.2025 https://www.gamedeveloper.com/design/spelunky-s-procedural-principles

[10] Motion Twin. (n.d.). Dead Cells: Procedural Generation Insights. GDC Vault (esim. konferenssiesitys). Saatavilla 18.3.2025 https://www.gdcvault.com/play/1025768/Dead-Cells-Procedural-Generation

[11] Epic Games. (n.d.). Unreal Engine Overview. Saatavilla 18.3.2025 https://www.unrealengine.com/

[12] Unreal Engine. (n.d.). Unreal Engine Documentation. Saatavilla 18.3.2025 https://docs.unrealengine.com/5.5/en-US/

[13] Unreal Engine. (n.d.). Procedural Content Generation Framework. Unreal Engine Documentation. Saatavilla 18.3.2025 https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation-overview

[14] Unreal Engine. (n.d.). Blueprints Visual Scripting. Unreal Engine Documentation. Saatavilla 18.3.2025 https://docs.unrealengine.com/5.5/en-US/blueprints-visual-scripting-in-unreal-engine/

[15] Tran, T. (2020). Procedural Dungeons in Unreal Engine. Game Developer (ent. Gamasutra). Saatavilla 18.3.2025 https://www.gamedeveloper.com/programming/procedural-dungeons-in-unreal-engine

[16] Lorensen, W. E., & Cline, H. E. (1987). Marching Cubes: A High Resolution 3D Surface Construction Algorithm. ACM SIGGRAPH Computer Graphics. Saatavilla 18.3.2025 https://dl.acm.org/doi/10.1145/37401.37422