Santeri Hienonen

2D-kenttäeditorin kehittäminen Unity-pelimoottorille

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

26.11.2017

Metropolia

Tekijä Otsikko Sivumäärä Aika	Santeri Hienonen 2D-kenttäeditorin kehittäminen Unity-pelimoottorille 40 sivua + 1 liite 26.11.2017
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tietotekniikka
Ammatillinen pääaine	Ohjelmistotekniikka
Ohjaajat	Lehtori Miikka Mäki-Uuro

Työn tavoitteena oli kehittää työkalu 2D-pelikenttien luomiseen Unity-pelimoottorille. Tarkoituksena oli rakentaa työkalu olemassa olevan peliprojektin tarpeisiin, ja samalla perehtyä työkalun toiminnallisuuden taustalla olevaan teoriaan ja käsitteisiin, kuten myös Unityn editorin laajentamiseen.

Työssä tutkittiin marching squares -algoritmin soveltuvuutta 2D-kenttien rakentamiseen, sekä sitä, miten lineaarista interpolointia voitaisiin hyödyntää sen tuottamien muotojen parantamiseen. Tämän jälkeen tutkittiin, miten algoritmin luomaa kolmiointia voitaisiin optimoida Delaunay-kolmiointia hyväksikäyttäen. Sopivien algoritmien löydyttyä työkalu toteutettiin käytännön tasolla, ja pohdittiin, miten työkalua voitaisiin hyödyntää paloista koostuvien pelitasojen rakentamisessa tai mahdollisessa tasojen automaattisessa generoinnissa. Työssä tarkasteltiin myös lopullisen toteutuksen järkevyyttä ja kehityskohteita.

Työkalun toiminnallisuutta käsiteltiin työssä yleisellä tasolla syventymättä tarkemmin sen koodiin tai luokkarakenteeseen, kun taas toteutuksen Unity-pelimoottoriin liittyviä yksityiskohtia tarkasteltiin hieman seikkaperäisemmällä tasolla.

Työn tuloksena saatiin aikaan toimiva kenttäeditori, jota on tarkoitus käyttää jatkossa projektissa, jota varten työkalua lähdettiin alun perin kehittämään.

Avainsanat	Unity, mesh, marching squares, Delaunay-kolmiointi

Author	Santeri Hienonen
Title	Developing 2D Level Construction Tool for Unity Game Engine
Number of Pages	40 pages + 1 appendix
Date	26 November 2017
Degree	Bachelor of Engineering
Degree Programme	Information Engineering
Professional Major	Software Engineering
Instructors	Miikka Mäki-Uuro, Senior Lecturer

The goal of the thesis was to develop a tool for building 2D game levels in the Unity game engine. The purpose was to build a tool for the needs of an ongoing game development project, and at the same time focus on the theory and principles behind the functionality of the tool, as well as extending the functionality of the Unity editor. The idea for the thesis was to follow the development process, as well as the operation process of the finished tool, from the beginning to the end.

In the thesis it was studied how suitable the marching squares algorithm would be for 2D game level construction, and how linear interpolation could be combined with marching squares to produce better shapes. After deemed suitable, it was explored how the triangulation produced with the marching squares algorithm could be optimized with Delaunay triangulation. After finding suitable algorithms to apply in the project, they were used in the actual implementation of the tool. Under consideration was also how the tool could be used in the process of constructing finished levels into the game out of level pieces, and possibly in the process of generating the levels automatically. Also, the reasonability of the way the tool was implemented, and the features that could be improved in the tool, were evaluated.

The inner workings of the tool are presented in a general level of detail without going further into the code or class structure implementations, whereas the implementation details considering the Unity game engine are introduced in greater detail.

As end result the study produced a working level editor, which is planned to be used in the future in the project for which the tool was built for.

Keywords	Unity, mesh, marching squares, Delaunay triangulation

Sisällys

Lyhenteet ja käsitteet

Collider	Unity-komponentti, jolla muodostetaan törmäyspintoja fysiikkalaskentaa varten.
Delaunay-kolmiointi	Muodostaa kolmioinnin samassa tasossa sijaitsevalle diskreetille pistejoukolle.
Kolmioverkko	Graafinen vektoriverkko, joka muodostuu kolmioista, joilla on yhteisiä kulmapisteitä ja sivuja toistensa kanssa.
Marching squares	Tietokonegrafiikka-algoritmi, jolla voidaan luoda ääriviivat kaksiulotteiselle skalaarikentälle.
Mesh	Yksinkertaisista konvekseista monikulmioista koostuva muoto, joiden joukossa kolmioverkko on yksi erikoistapaus.
Polygoni	Monikulmio.
Unity	Pelimoottori, jolla kehitetyn pelin voi helposti kääntää useille alustoille.
Unity editori	Graafinen käyttöliittymä pelinkehitykseen.
Unity Scene	Unityn tiedostomuoto, johon pelimaailma tallennetaan. Yksittäistä Scene-tiedostoa voidaan ajatella yhtenä pelin tasona.
Unity Serialization	Automaattinen prosessi, jossa datarakenteet ja peliobjektien tilat muunnetaan formaattiin, jonka Unity voi tallentaa ja rakentaa uudelleen myöhemmin.
Unity unit	Unityn sisäinen mittayksikkö, jonka vakioarvo on 1 unit = 1 metri.

1 Johdanto

Insinöörityön tavoitteena oli kehittää työkalu Unity-pelimoottorille kaksiulotteisten kenttien nopeaan rakentamiseen. Työn tarkoituksena oli selvittää, mitä tällainen kenttägenerointi vaatii ja miten se olisi mahdollista toteuttaa pelimoottorin ja sen editorin tarjoamissa puitteissa. Aiheesta pyrittiin työssä muodostamaan kokonaisuus käsittelemällä sekä käytännön toteutusta että taustalla olevia teoreettisia käsitteitä, joiden varaan toteutus rakentuu.

Aihe valittiin johtuen kenttätyökalun tarpeesta olemassa olevaan peliprojektiin. Työkalun luonnin tarkoituksena oli suoraviivaistaa pelin kentänluontiprosessia ja nopeuttaa erilaisten kenttäsuunnitelmien iterointia luomalla Unityn editoriin toiminnallisuutta, jota siinä ei valmiiksi ollut. Työn toivottiin avaavan myös jossain määrin Unityn editorin sisäistä toimintaa ja sitä, mitä sen laajentaminen käytännössä vaatii.

Kenttätyökalun luonnin tavoitteena oli osaltaan siirtää projektin kenttäsuunnittelun ideaa kokonaisten tasojen käsin luomisesta kohti paloista koostuvien tasojen luontia. Tällöin hyväksi havaittuja ideoita pystyttäisiin kierrättämään tasosuunnittelussa helposti uudelleen pienillä muokkauksilla. Toivon mukaan näin vältettäisiin samalla täysin automatisoidun tasojen generoinnin aiheuttamat ongelmat pelisuunnittelun kannalta. Tässä tapauksessa olisi myös tärkeää pitää huoli siitä, että tasot eivät kuitenkaan tuntuisi pelaajasta samojen asioiden kierrättämiseltä. Aluksi tasoja koottaisiin paloista käsin. Konseptin toimiessa voitaisiin tulevaisuudessa mahdollisesti siirtyä täysin automatisoituun paloista kasattavien tasojen luontiin, joka tarjoaisi peliin teoriassa loputtoman määrän pelattavaa.

Työn luvussa 2 käsitellään työkalulta vaadittuja ominaisuuksia, sekä lähtökohtia, joista työkalua lähdettiin rakentamaan. Luvussa 3 vuorostaan käydään läpi työkalun perustoiminnallisuus. Neljännessä luvussa käsitellään kolmannessa luvussa luodun kenttäobjektin kolmioinnin optimointi. Luvussa 5 seurataan käyttäjän palautteen matka painalluksesta lopulliseen pelissä käytettävään muotoon. Luvussa 6 käsitellään muut työkaluun perustoiminnallisuuden lisäksi lisätyt ominaisuudet. Seitsemännessä luvussa spekuloidaan, miten työkalulla luotuja kentän osia voitaisiin liittää toisiinsa ja miten niitä voitaisiin mahdollisesti käyttää kokonaisten tasojen generoimiseen. Luvussa 8 tarkastellaan kriittisesti työssä havaittuja ongelmia ja työkalun kehityskohteita, ja

arvioidaan myös luodun työkalun käyttökelpoisuutta. Yhdeksännessä luvussa tehdään loppuyhteenveto aiemmissa luvuissa käsitellyistä aiheista.

2 Työn lähtökohdat

Tässä luvussa käsitellään lähtökohtia, joista työkalua lähdettiin rakentamaan. Aluksi luvussa pyritään antamaan yleiskuva Unity-pelimoottorista, käydään läpi työkalulta vaaditut ominaisuudet ja tutkitaan, millaisia valmiita vaihtoehtoja työkalulta vaaditun toiminnallisuuden täyttämiseksi olisi ollut tarjolla. Luvussa kuvataan myös uuden työkalun pohjana käytetyn vanhan työkalun toiminnallisuus, ja tutkitaan, miten Unityn editorin käyttöliittymän alkuperäistä komponenttinäkymää on mahdollista muokata ja laajentaa.

2.1 Unity-pelimoottori

Unity on pelimoottori, jonka tärkeimpiä ominaisuuksia on sen tuki käytännössä kaikille suurimmille julkaisualustoille; esimerkiksi Windows, Linux, macOS, iOS, Android, PlayStation 4, Xbox One ja Nintendo Switch ovat kaikki tuettuja. Indie-pelinkehittäjien kannalta Unitystä tekee mielenkiintoisen myös sen hinnoittelumalli. Pelimoottorin perusversio on käyttäjälle täysin ilmainen, jos käyttäjän tai yrityksen vuosittaiset tulot ovat alle 100 000 $. Unity tarjoaa pelinkehitykseen graafisen käyttöliittymän, eli Unity editorin, joka mahdollistaa käyttäjälle ”drag and drop” -toiminnallisuuden pelimaailman luomiseen. Unity tarjoaa myös käyttäjän kannalta helposti lähestyttävät työkalut esimerkiksi grafiikkojen ja äänien käsittelyyn ja pelimaailman fysiikat on mahdollista toteuttaa Unityn omalla fysiikkamoottorilla. Editorin toiminnallisuutta on myös mahdollista laajentaa Unity Asset Storen kautta, jossa muut käyttäjät voivat jakaa tai myydä itse tekemiään editorilaajennuksia tai pelisisältöä. Ohjelmointikielenä Unity käyttää C#-ohjelmointikieltä, joka on elokuusta 2017 eteenpäin ainoa pelimoottorin virallisesti tukema ohjelmointikieli. [1; 2.]

2.2 Työkalulta vaadittu toiminnallisuus

Peliprojektissa, johon työkalu on tarkoitettu käytettäväksi, oli jo valmiiksi olemassa oleva kenttätyökalu, jonka käytettävyys ei kuitenkaan ollut projektin tarpeisiin tyydyttävää.

Kenttien rakennus ja muokkaus oli työkalulla yksinkertaisesti liian työlästä ja aikaavievää. Uudesta työkalusta pyrittiin tekemään mahdollisimman nopea ja vaivaton käyttää. Tavoitteena oli saada työkalu toimimaan piirto-ohjelman tapaan: käyttäjä voi ”piirtää” valmista kenttää ilman viivettä sekä Unityn Scene- että Play-näkymissä. Lisäksi käyttäjä voi lisätä peliobjekteja kenttään pikanäppäimiä käyttäen tehtyjen muutosten tallentuessa kenttään käytetystä näkymästä riippumatta (normaalisti Unityn Play-näkymässä tehdyt muutokset eivät tallennu varsinaiseen, Scene-näkymässä näkyvään kenttään). Koska lähtötiedot kyseisentyylisistä ongelmista olivat työtä aloitettaessa vähäiset, työn toteutuksen tapa, ja ylipäätään se, onko kuvatun kaltainen toteutus mahdollinen, olivat täysin avoimia kysymyksiä. Työn lähtökohdaksi muodostuikin vastauksien etsiminen seuraaviin kysymyksiin:

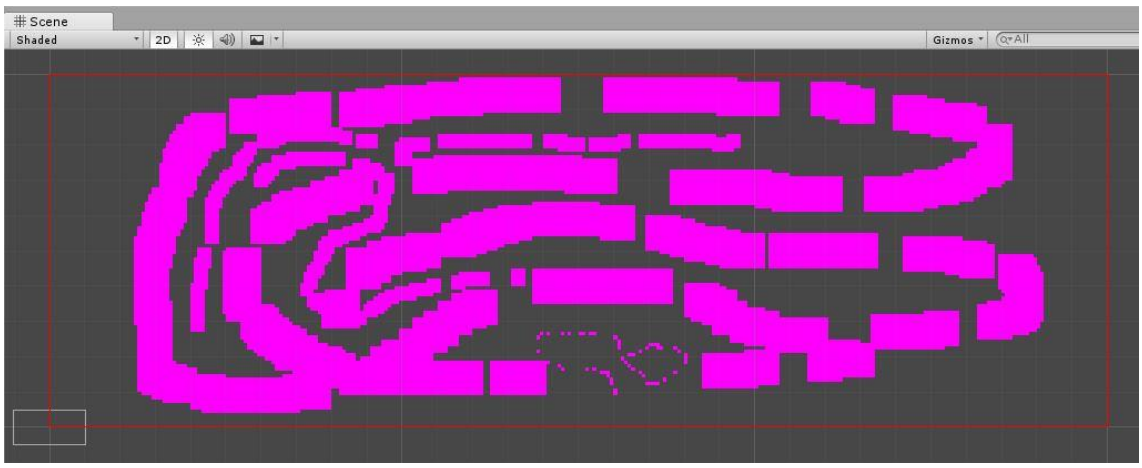
1. Miten Unityn editorin toiminnallisuutta voidaan laajentaa?
2. Onko valmista kenttää mahdollista generoida reaaliajassa?
3. Voiko Unityn Play-näkymästä tallentaa tietoa Scene-näkymään, ja jos voi, mitä tietoa voidaan tallentaa?

2.3 Valmiit vaihtoehdot

Koska työkalu vaikutti jo lähtökohtaisesti suhteellisen työläältä ongelmalta, aloitettiin työnteko kartoittamalla, olisiko jo valmiita ratkaisuja vastaavan tyylisiin ongelmiin mahdollista löytää Unityn Asset Storesta kohtuullista korvausta vastaan. Asset Storesta löytyikin nopeasti useita kentänrakennustyökaluja, mutta suurin osa näistä oli kehitetty 3D-kenttien rakentamiseen. Kaikki 3D-työkalut päätettiinkin rajata pois, koska vaikka ne vakuuttivat käytännössä poikkeuksetta ominaisuuksillaan, niin valtaosa näistä ominaisuuksista olisi turhia 2D-käytössä, eikä näistä ominaisuuksista haluttu myöskään turhaan maksaa. Löydetyistä 2D-vaihtoehdoista lupaavimmat tarkasteluun otetut työkalut olivat Ferr2D Terrain Tool [3] ja 2D Terrain Editor [4]. Mutta koska edellä mainittujen työkalujen (tai minkään muunkaan Asset Storesta löydetyn kenttätyökalun) käyttötapa ei vastannut luvussa 2.2 mainittuja vaatimuksia, päätettiin työkalu ainakin pyrkiä toteuttamaan itse. Hyvänä puolena oman työkalun kehittämisessä pidettiin myös sitä, että jos työkalun kanssa törmätään yhteensopivuus- tai muihin ongelmiin, niin ei olla ulkopuolisen kehittäjän mahdollisen käyttäjätuen varassa, vaan mahdolliset ongelmat voidaan paikantaa ja korjata itse.

2.4 Vanhan toteutuksen kuvaus

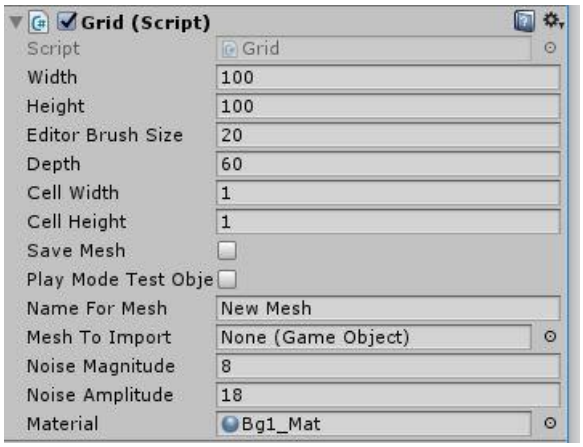
Vähäisistä lähtötiedoista johtuen työlle valittiin aloituspisteeksi projektista löytynyt vanha kenttätyökalu. Työkalu otti vastaan käyttäjän hiiren klikkauksen sijainnin ja ylläpiti sen perusteella bittikarttaa, jossa yhtä Unityn unit-yksikköä vastasi yksi bitti. Jos käyttäjä klikkasi hiiren oikeaa näppäintä, bitti sai koordinaatissa arvon 1 (lisäys), ja taas vasenta näppäintä klikkaamalla arvon 0 (poisto). Lisätessä työkalu piirsi palautteena kyseiseen koordinaattiin teksturoimattoman neliön muotoisen meshin (mesh = kolmioverkko, tästä eteenpäin käytetään nimitystä mesh) ja poistaessa tietysti poisti neliön koordinaatista (kuva 1). Viimeistellyn, pelissä käyttökelpoisen kentän sai työkalulla aikaan Finalize-näppäintä painamalla. Toteutuksen ongelmia olivat satunnainen kaatuilu, raskaus, kentän hankala muokattavuus ja Finalize-toimenpiteen kesto, josta oli tarkoitus päästä eroon reaaliaikaistamalla Finalizessä tapahtuva toiminnallisuus. Käyttäjäpalautteesta muodostuva bittikartta vaikutti kuitenkin hyvältä myös uuden toteutuksen tarpeisiin, ja vanhan työkalun editorikäyttöliittymä antoi pohjan Unityn editorin muokkaamiseen.



Kuva 1. Vanhalla kenttätyökalulla luotua meshiä ennen Finalize-vaihetta

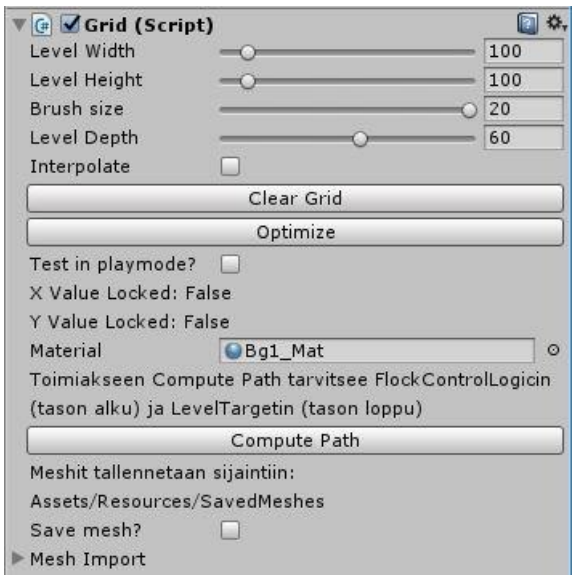
2.5 Unityn editorin käyttöliittymän muokkaus

Unityn editorin käyttöliittymän muokkaaminen osoittautui nopeasti helpoksi. Ilman käyttäjän tekemiä muokkauksia Unity näyttää editorissa käyttäjälle kaikki tarkastellulle luokalle määritellyt julkiset muuttujat, jotka täyttävät Unityn serialization-ominaisuuden vaatimukset (kuva 2).



Kuva 2. Normaali editorinäkymä ennen käyttäjän muokkauksia

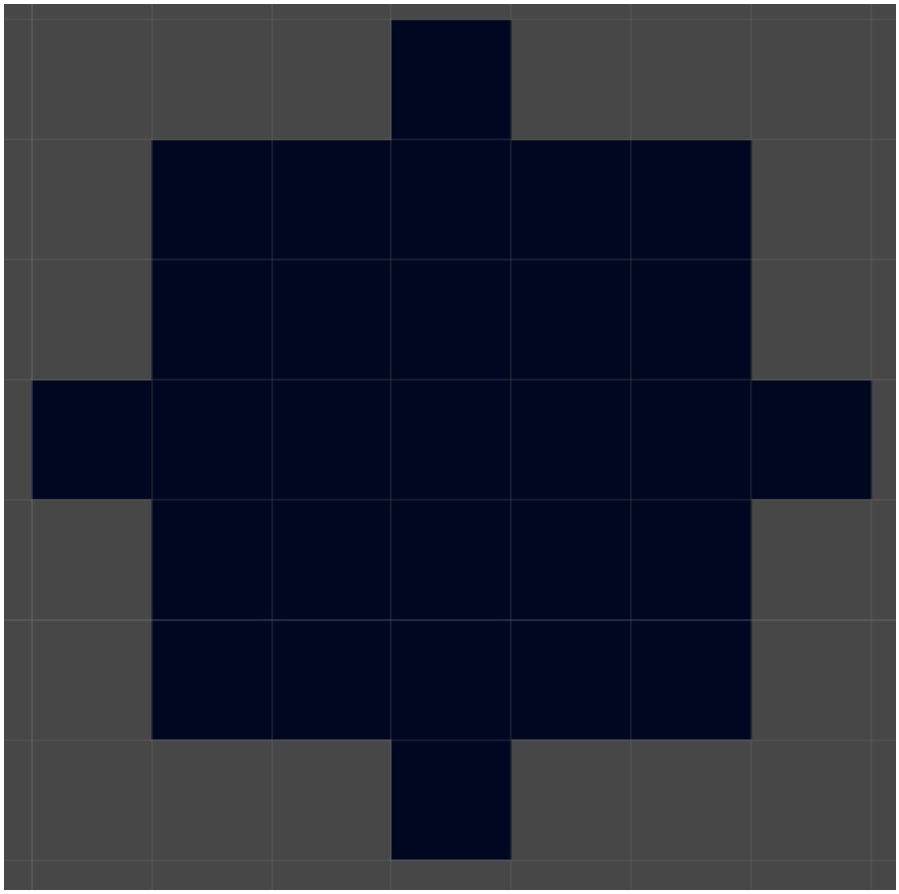
Käyttäjä voi kuitenkin halutessaan kirjoittaa jokaiselle luokalle oman editoriluokan, joka laitetaan perimään Unityn Editor-kantaluokka. Sille asetetaan kohteeksi käyttäjän itse tekemä luokka, jolloin editorissa kohdeluokkaa tarkasteltaessa vakionäkymä ylikirjoitetaan käyttäjän määrittämällä näkymällä, ja kaikki kohdeluokan julkiset muuttujat ovat vapaasti muokattavissa (kuva 3). Käyttäjä voi myös kaapata käyttäjäsyötteen hallinnan editoriluokassa tutkimalla senhetkistä Event-oliota (sisältää viimeisimmän käyttäjäpalautteen) editorinäkymän ollessa auki, jolloin editorin normaali toiminnallisuus esimerkiksi hiiren vasemmalle klikkaukselle voidaan korvata käyttäjän määrittelemällä toiminnallisuudella.



Kuva 3. Muokattu editorinäkymä samalle luokalle kuin kuvassa 2.

3 Toteutusperiaatteet

Lähtötilanteen pohjalta lähdettiin siis rakentamaan toteutusta, jossa käyttökelpoista meshiä voidaan rakentaa lennossa. Jotta mesh olisi suoraan käyttökelpoinen myös pelin tasona, sille tulee generoida meshin kulmapisteiden perusteella reaaliajassa myös törmäyspinta (collider) fysiikkojen törmäystarkistuksiin. Kuten edellisessä luvussa kuvattiin, pohjana ollut vanha editorityökalu rakensi meshiä reaaliajassa vain neliöistä, mutta jos meshin olisi tarkoitus olla suoraan käyttökelpoista, työkalun tulisi pystyä tuottamaan pyöreitä ja kaarevia muotoja tarkemmin kuin mitä neliöistä koostetulla meshillä pystytään approksimoimaan (kuva 4).



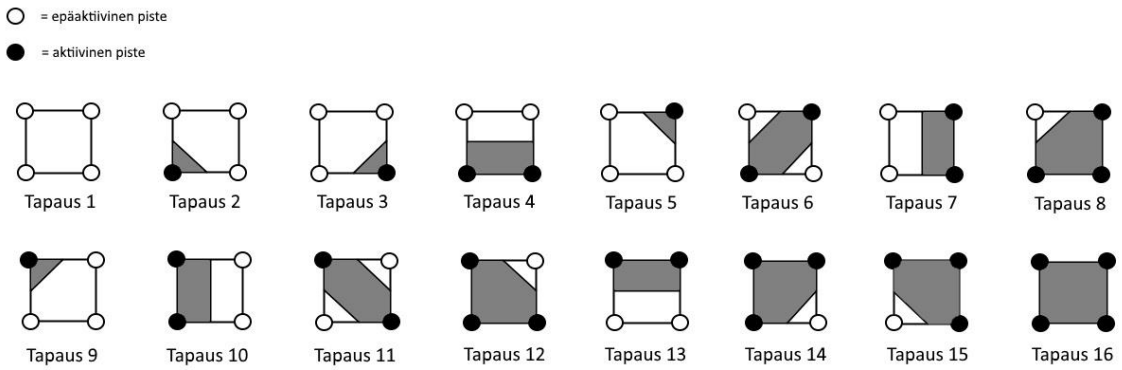
Kuva 4. Neliöistä approksimoitu ympyrä, kun ympyrän halkaisija on 3

Neliömesheissä on kuitenkin se hyvä puoli, että niitä pystytään yhdistelemään helposti. Jos esimerkiksi yhtä bittikartan pistettä vastaa yksi neliö, ja jos neliön koko on yksi, sijoittuu sen jokainen sivu aina kahden vierekkäisen bittikartan pisteen puoleen väliin. Tällöin aktivoidulle pisteelle on helppo tehdä kolmiointi ja kolmiointia on helppo ylläpitää, koska päällekkäisyyksiä ei voi syntyä. Kolmiografiikalla vuorostaan voidaan helposti

piirtää ympyrälle tarpeeksi tarkka esitys, mutta jos ajatellaan tämän työkalun todennäköistä käyttötilannetta, jossa käyttäjä piirtää olemassa olevan ympyrän päälle toisen ympyrän, jouduttaisiin lopputuloksena tuleva muoto kolmioimaan monimutkaisesti uudelleen. Muodostuvan polygonin (eli monikulmion) reunapisteistä olisi myös vaikeaa pitää kirjaa, koska ympyrän reunapisteet voisivat sijaita täysin mielivaltaisissa koordinaateissa, verrattuna siihen, että neliöistä approksimoituun ympyrään lisättäisiin vain neliö niihin ympyrän säteen sisällä oleviin koordinaatteihin, joissa ei vielä ole neliötä. Tarvitaan siis paremmin ja monipuolisemmin kaarevia muotoja approksimoiva muoto kuin neliö, mutta samanaikaisesti muotoja tulee pystyä myös helposti ja saumattomasti yhdistelemään toisiinsa.

3.1 Marching squares (marssivat neliöt)

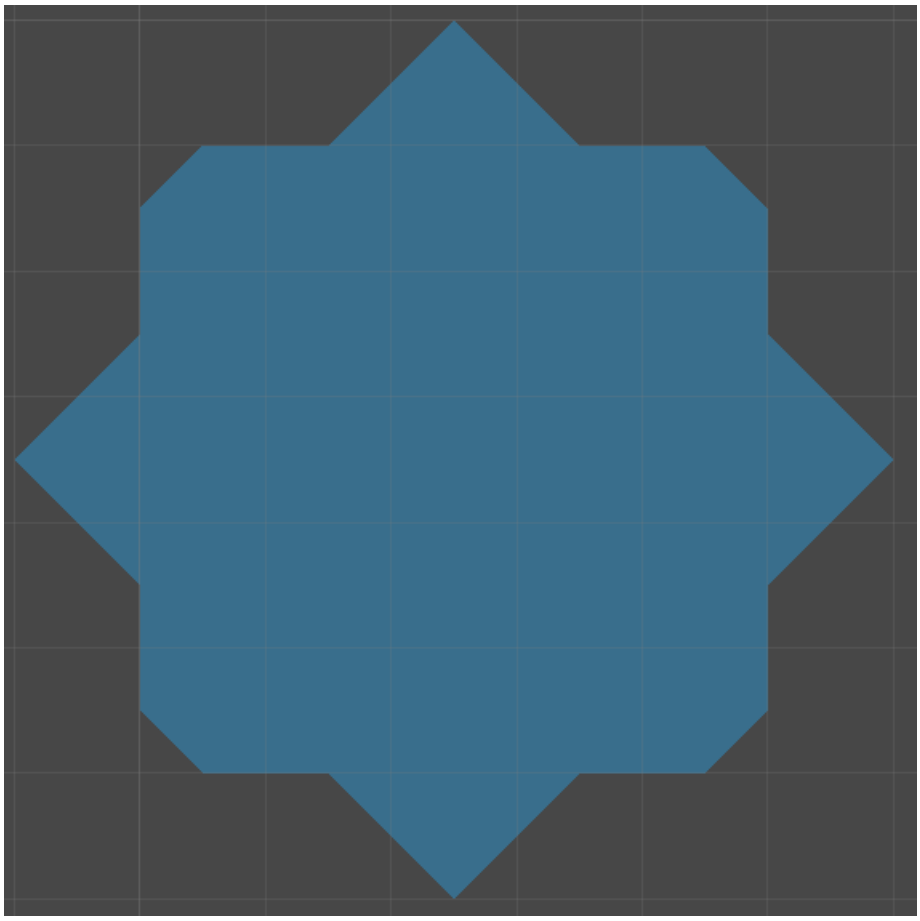
Ratkaisutavaksi yksinkertaiseen, mutta samalla myös tarpeeksi monipuoliseen, kolmiointiin löytyi tutkimisen jälkeen marching squares -algoritmi [5], jossa jokainen koordinaatiston yksittäinen piste on pikselin sijasta kulmapiste neliölle, joista koostuvaa verkkoa käsitellään meshinä. Kun koordinaatiston yksittäinen piste on neliön kulmapiste ja jokainen koordinaatiston yksittäinen piste on erikseen aktivoitavissa, on neliöllä siis neljä muuttujaa, jotka voivat saada arvon 0 tai 1, eli siis 16 erilaista kombinaatiota. Neliö voi muodostaa siis konfiguraationsa perusteella 16 erilaista muotoa. Jokainen marching squares -neliö koostuu kahdeksasta pisteestä, sillä jokaisen kahden kulmapisteen välillä on välipiste, joka on käytössä, jos vain toinen siihen yhteydessä olevista kulmapisteistä on aktiivinen. Näin neljällä muuttuvalla arvolla saadaan aikaan 16 erilaista polygonia (kuva 5).



Kuva 5. Kaikki mahdolliset marching squares -neliökonfiguraatiot

Kuten kuvasta 5 voidaan todeta, tapaukset 6 ja 11 ovat moniselitteisiä. Vastakkaisissa kulmissa olevat aktiiviset pisteet voidaan toteutustavasta riippuen yhdistää joko yhdeksi polygoniksi tai niistä voidaan muodostaa kaksi erillistä polygonia. Tässä työssä ne päätettiin yhdistää yhdeksi polygoniksi. Kuvasta 5 on myös nähtävissä, kuinka helppoa marching squares -algoritmin toteuttaminen käytännössä on: jokaisen neliön vasen alakulma on vähiten merkitsevä bitti ja vasen yläkulma eniten merkitsevä bitti. Ennen piirtämistä käydään läpi jokaisen neliön bittiarvo ja päätetään sen perusteella, mitä neliön pisteitä kolmioinnissa käytetään ja millainen kolmiointi niiden välille muodostetaan.

Koska jokainen meshin kulma ei ole enää 90 asteen kulma, vaan 45 asteen tai 135 asteen kulma, on luonnollisesti mahdollista saada jo silminnähden parempi approksimaatio ympyrän muodosta (kuva 6). Kuvasta 5 voidaan myös nähdä, että marching squaresin perustapaus, jos aktivoitu piste ei ole piirtoalueen reunalla, on salmiakkikuvio, sillä jokainen piste joka ei ole piirtoalueen rajalla on samanaikaisesti vasen alareuna, oikea alareuna, vasen yläkulma sekä oikea yläkulma jollekin ruudukon neliölle. Tämä tarkoittaa myös sitä, että marching squares ei pysty perustoteutuksessaan tuottamaan 90 asteen kulmia, missä kulman määrittävät sivut ovat x- ja y-akselin suuntaisia.

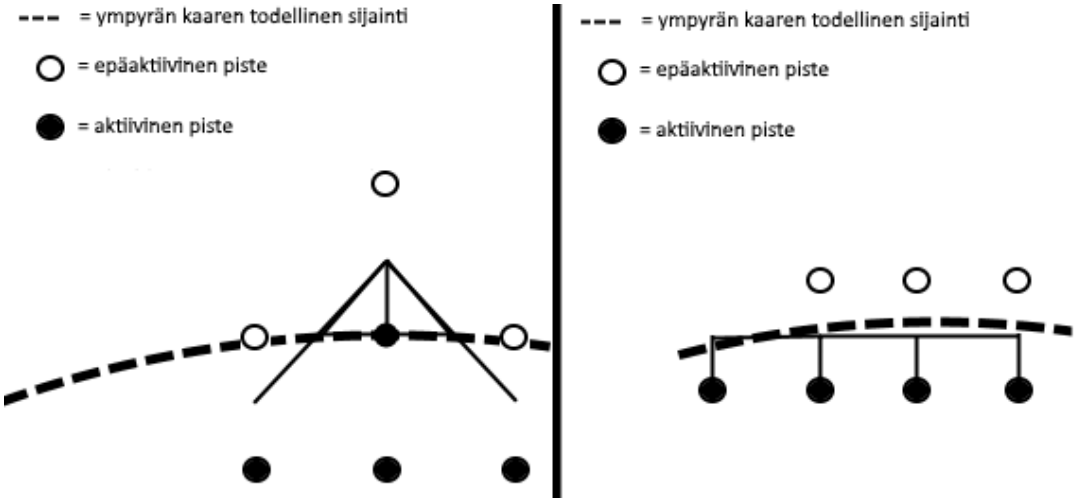


Kuva 6. Marching squares -approksimoitu ympyrä, kun ympyrän halkaisija on 3

Saman tien voidaan jo kuitenkin sanoa, että muoto ei ole tarpeeksi hyvä, jos tavoitteena on saada aikaan kaarevia muotoja. Tähän ongelmaan marching squares tarjoaa kuitenkin myös ratkaisumahdollisuuden. Käyttäjä siis aktivoi ruudukosta hiiren painalluksella pisteitä, jotka vastaavat neliöruudukon kulmapisteitä. Näiden pisteiden tulee sijaita koko ajan ruudukon koon mukaan asetetuissa vakiosijainneissa. Mutta näiden pisteiden lisäksi käytettävissä on nyt jokaisen kahden kulmapisteen välissä olevat välipisteet, jotka eivät ole käyttäjän suoran kontrollin alaisina, vaan niiden aktiivisuus riippuu niihin yhteydessä olevien kulmapisteiden aktiivisuudesta. Niiden ei siis tarvitse sijaita missään tietyssä vakiosijainnissa, kunhan ne sijaitsevat jossain niihin yhteydessä olevien kulmapisteiden väliin muodostuvalla suoralla [6]. Näitä pisteitä voidaan siis liikutella vapaasti kulmapisteiden määrittämällä sivulla.

3.2 Lineaarinen interpolointi

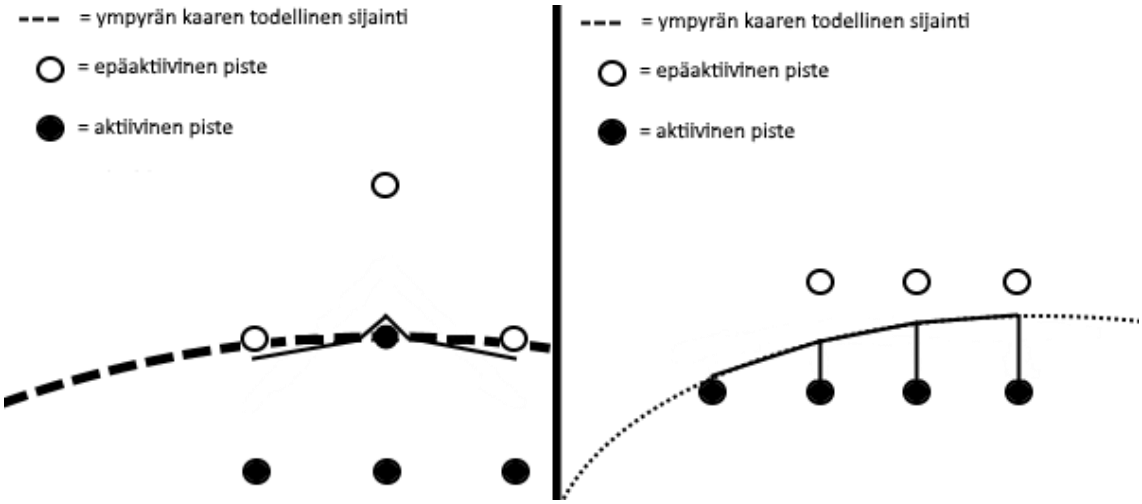
Jos pyritään saamaan aikaan ympyränkaltaista muotoa, otetaan keskipisteeksi piste, jonka käyttäjä on aktivoinut hiirellä, ja tutkitaan sen ympäriltä alue, jonka käyttäjä on määrittänyt ympyrän säteeksi. Jos kulmapiste on ympyrän sisällä tai ympyrän kaarella, piste aktivoidaan, ja jos taas kulmapiste on ympyrän kehän ulkopuolella, sitä ei käsitellä. Tiedetään siis, että ympyrän kehä kulkee todellisuudessa joko aktivoidun pisteen läpi tai aktivoidun pisteen ja sen viereisen epäaktiivisen pisteen muodostaman suoran läpi. Marching squares -perustoteutuksessa välipisteet ovat aina neliön kahden kulmapisteen puolessavälissä, jolloin jos oletetaan, että aktivoitu piste oli ympyrän kehällä, marching squares sijoittaa sen muodostaman muodon huipun puolen ruudun päähän ympyrän todellisesta kehäpisteestä (kuva 7).



Kuva 7. Marching squaresin muodostama raja verrattuna ympyrän kaaren todelliseen sijaintiin kahdessa eri tapauksessa ympyrää approksimoitaessa.

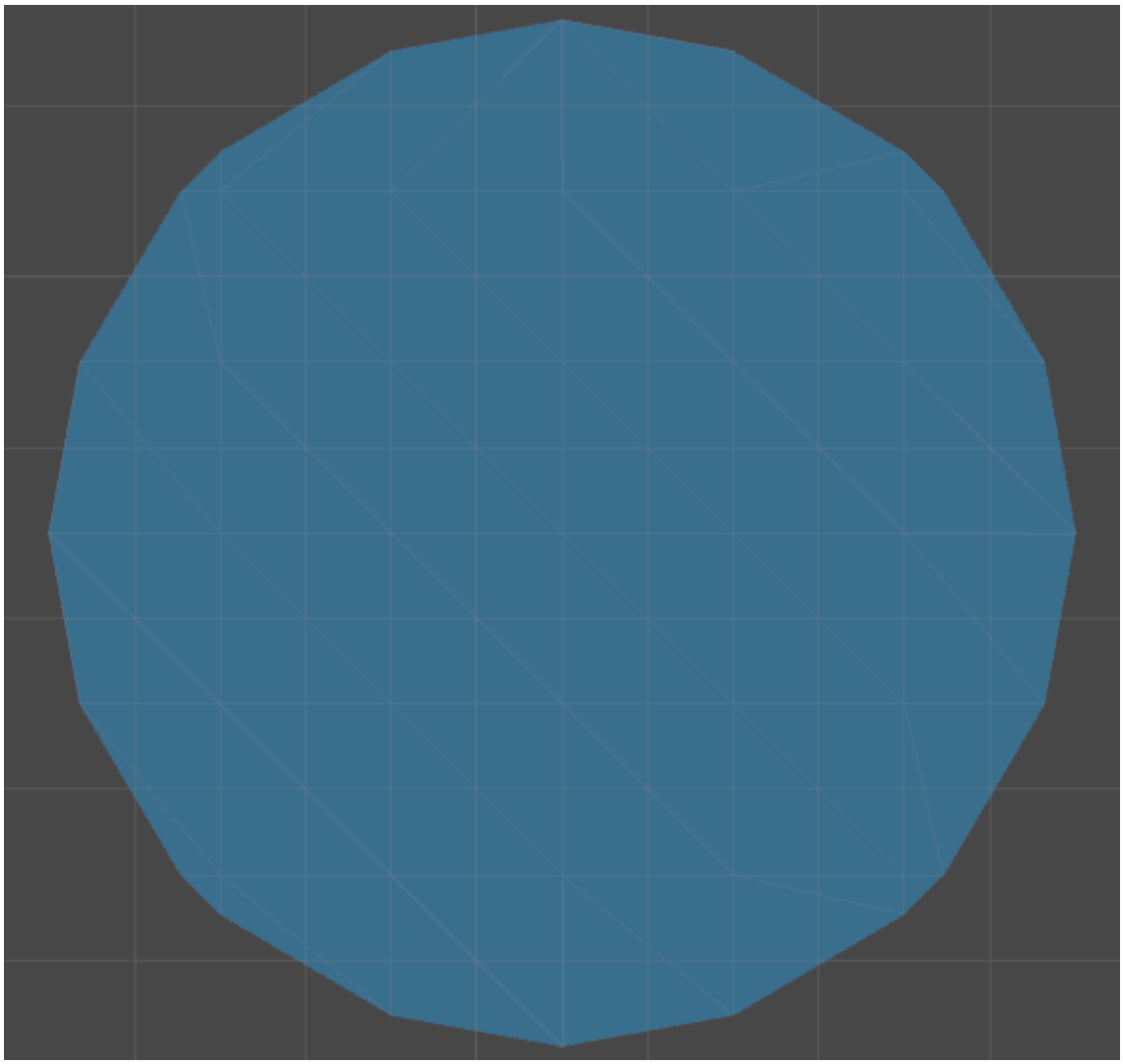
Ratkaisuna ongelmaan voidaan kuitenkin liikutella välipisteitä. Kun välipisteet sijoitetaan mahdollisimman lähelle ympyrän kaaren todellisia kehäpisteitä, saadaan huomattavasti tarkempi ympyräapproksimaatio (kuva 8). Jokaisen aktivoidun kulmapisteen kohdalla siis tutkitaan, onko kyseisen kulmapisteen vieressä epäaktiivisia kulmapisteitä, ja jos on, lisätään aktivoitu kulmapiste lineaarisen interpolaation alkupisteeksi ja sen viereinen epäaktiivinen kulmapiste päätepisteeksi. Tämän jälkeen interpoloidaan lineaarisesti aktiivisen pisteen ja sen viereisen epäaktiivisen pisteen välillä niin kauan, kunnes niiden välisen pisteen etäisyys keskipisteestä ylittää ympyrän kehän etäisyyden keskipisteestä, eli ympyrän säteen, ja sijoitetaan välipiste edelliseen ympyrän sisällä olleeseen koordinaattiin. Käytännössä interpolointialueelle siis lisätään jokainen piste, jonka

painallus aktivoi, ja myös kaikki epäaktiiviset pisteet, jotka ovat aktiivisten pisteiden vieressä, jotta jokaista aktiivista pistettä kohden on olemassa piste, jota kohti näiden kahden pisteen välisen pisteen sijaintia voi interpoloida.



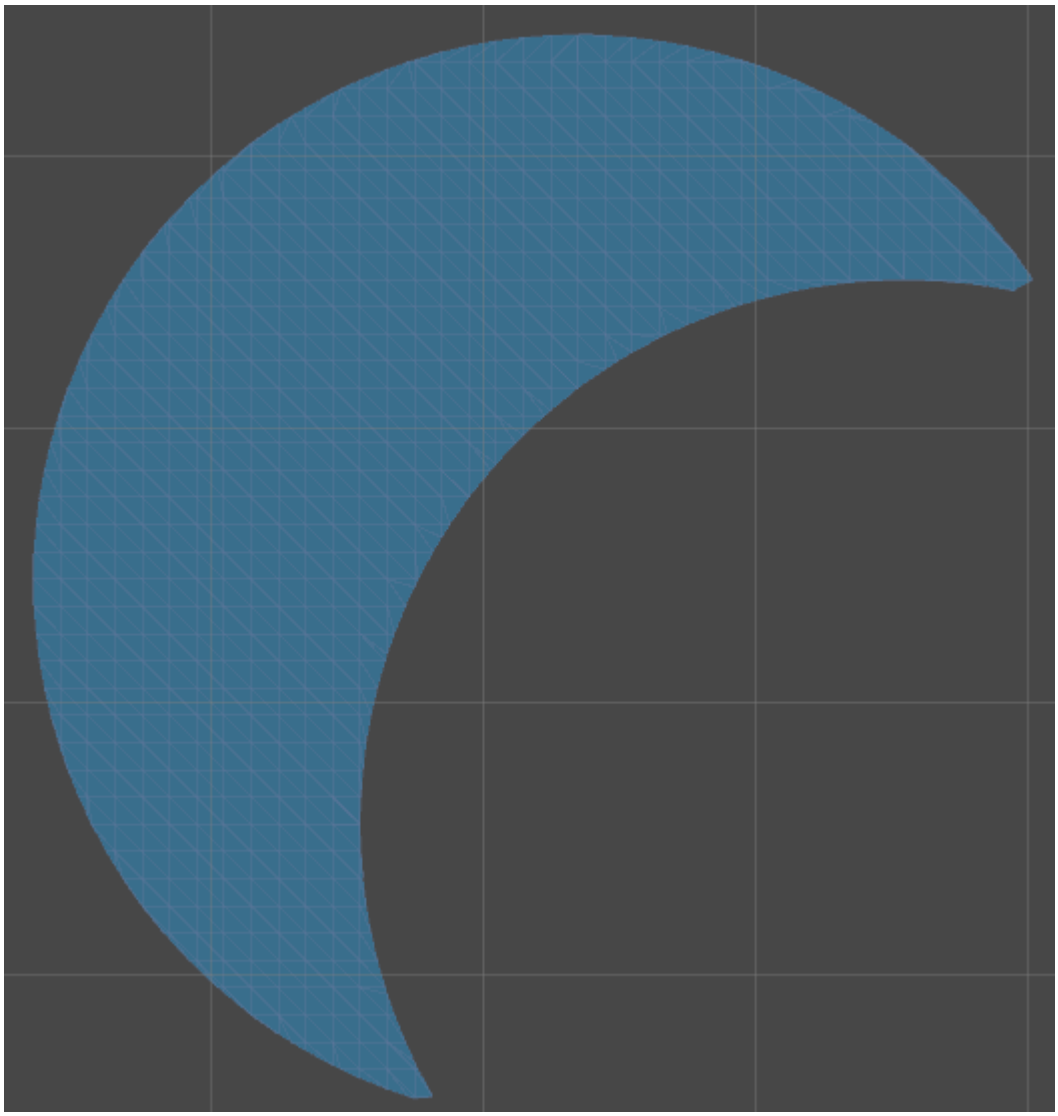
Kuva 8. Marching squaresin muodostama raja verrattuna ympyrän kaaren todelliseen sijaintiin kahdessa eri tapauksessa ympyrää approksimoitaessa, kun välipisteen sijainti on interpoloitu

Interpoloinnin tarkkuus verrattuna ympyrän todelliseen kehäpisteeseen riippuu interpolointiaskeleen pituudesta. Lyhyemmällä interpolointiaskeleella päästään tarkempaan lopputulokseen, mutta interpolointi vie kauemmin verrattuna pidempään interpolointiaskeleeseen. Käytännössä toteutuksella päästään siis yksittäisen ympyrän tapauksessa tarpeeksi hyvään approksimaatioon, jos vain interpolaatioaskel on tarpeeksi pieni ja ruudukon kulmapisteiden määrä tarpeeksi suuri (kuva 9).



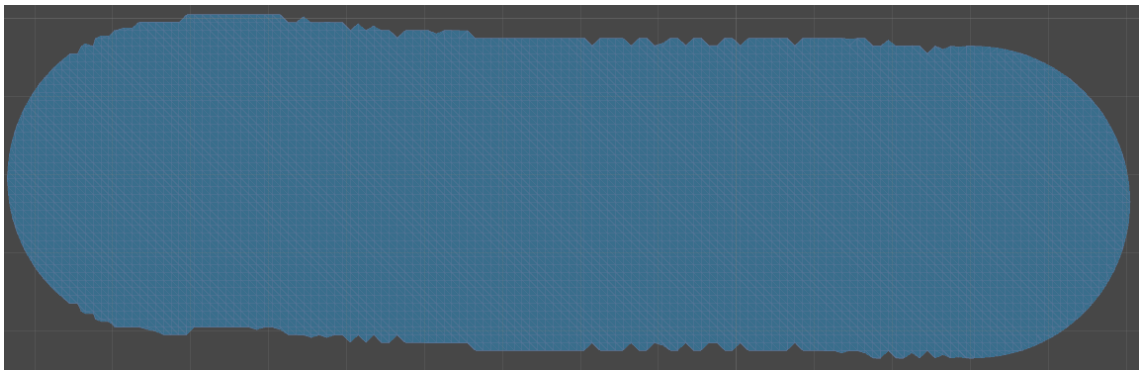
Kuva 9. Marching squares -approksimoitu ympyrä välipisteiden interpoloinnilla, kun ympyrän halkaisija on 3

Entä jos käyttäjä haluaa muokata aikaisempaa muotoa poistamalla siitä alueen? Poistaessa interpoloidaan päinvastaisesti, eli ympyrän keskipisteen arvo vaihdetaan epäaktiiviseksi, jos se ei ole jo valmiiksi epäaktiivinen, ja kaikki kulmapisteet ympyrän säteen alueella asetetaan epäaktiivisiksi. Jos ympyrän säteen sisällä olevan epäaktiivisen pisteen vieressä on aktiivinen kulmapiste, interpoloidaan lineaarisesti epäaktiivisen ja aktiivisen kulmapisteen välisen välipisteen sijaintia näiden pisteiden välillä aivan normaalisti ja asetetaan välipisteen arvo mahdollisimman lähelle poistetun alueen muodostaman ympyrän sädettä (kuva 10).



Kuva 10. Interpoloitu ympyrä, jonka alueelta on poistettu interpoloitu ympyrä

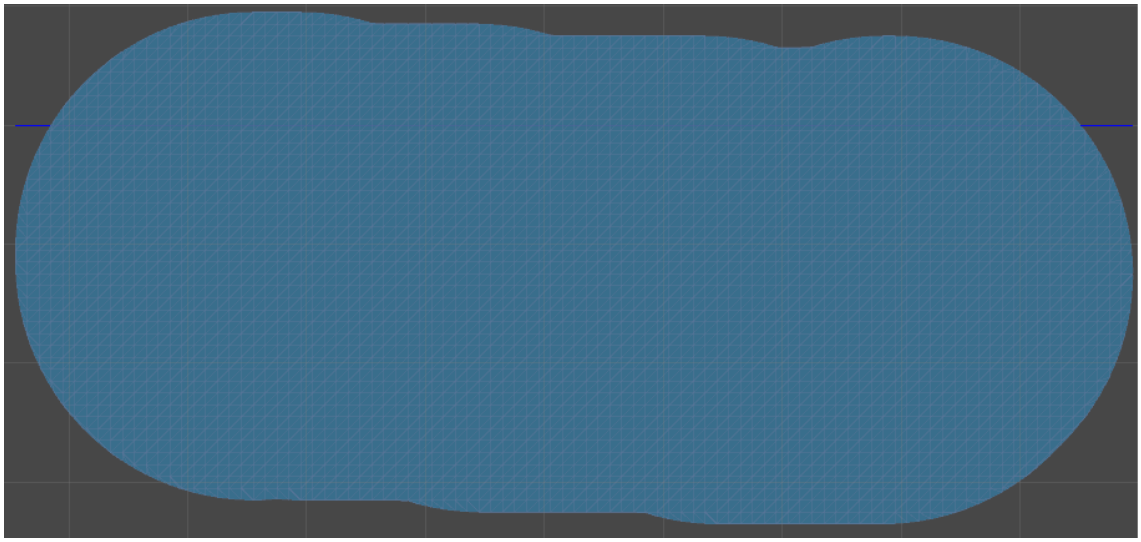
Vaikka aikaan saadaan jo tarpeeksi pyöreäreunainen mesh yhdellä painalluksella, tämä ei vielä vastaa realistista käyttötilannetta työkalulle. On tärkeää, että painiketta pohjassa pitämällä saa vedettyä tarpeeksi laadukasta meshiä suoraan ruudulle, jotta tasoa voi rakentaa suoraan reaaliajassa. Meshin reunojen laatu ei ole kuitenkaan vielä tällä toteutuksella tarpeeksi hyvällä tasolla, jotta se täyttäisi työkalulle asetetut vaatimukset (kuva 11).



Kuva 11. Useita peräkkäin liitettyjä interpoloituja pyöreitä meshejä

Kuten kuvasta 11 voidaan nähdä, meshejä peräkkäin liitettäessä sen reunalle muodostuu sahalaitakuviota. Ainoastaan meshin molemmat ääripäät näyttävät siltä, miltä niiden haluttaisiin näyttävän. Ongelmaksi muodostuu nimenomaan tämänhetkinen interpolaatiototeutus. Lisäysoperaatiolla ei voida deaktivoida kulmapisteitä, mutta aktiivisen ja epäaktiivisen kulmapisteen välisen välipisteen interpoloitu arvo voi muuttua ympyröitä peräkkäin aseteltaessa. Jos esimerkiksi oletetaan, että ympyrän huippukohta asettuu ensimmäisellä painalluksella aktiivisen ja epäaktiivisen kulmapisteen välille muodostuvalle suoralle xy, interpoloidaan suoralla xy oleva välipiste ympyrän huippukohtaan. Jos seuraavalla päivityksellä käyttäjä lisää ympyrän viereen toisen ympyrän samalla säteellä ja samalla ympyrän keskipisteen y-koordinaatilla, mutta siirtäen ympyrän keskipistettä yhden yksikön verran oikealle, on mahdollista, että ympyrän kaari läpäisee taas suoran xy. Koska ympyrän huippukohta ei kuitenkaan ole enää suoralla xy keskipisteen sijainnin muuttumisen johdosta, suoralla xy oleva välipiste interpoloidaan y-suunnassa alemmaksi kuin mitä se edellisellä päivityksellä oli. Dataa alkuperäisestä painalluksesta siis menetetään, eikä ensimmäisellä painalluksella piirretty alue näytä uuden painalluksen jälkeen enää samalta, miltä se alun perin näytti. Tämä on käyttäjän kannalta epäintuitiivista, eikä se myöskään näytä hyvältä, joten on tärkeää, että piirrettyjen ympyröiden rajat säilytetään, jos ne sijoittuvat meshin ulkokehälle. Tähän ratkaisu on lopulta yksinkertainen. Jokaista uutta suoralle xy sijoittuvaa interpoloitua välipisteen sijaintia verrataan välipisteen nykyiseen sijaintiin. Jos välipiste uudella sijainnilla laajentaa meshin pinta-alaa, uusi sijainti sijoitetaan vanhan tilalle, ja jos uusi sijainti taas pienentää pinta-alaa, välipiste pidetään vanhassa sijainnissaan. Käytännössä tämä tarkistetaan katsomalla, kummalla puolella välipistettä sen viereinen aktiivinen kulmapiste sijaitsee. Jos välipiste on aktiivisen pisteen vasemmalla puolella, uusi pisteen sijainti sijoitetaan vanhan sijainnin tilalle, jos sen x-arvo on pienempi. Jos välipiste on vuorostaan oikealla puolella, tutkitaan, onko uudella

pisteen sijainnilla suurempi x-koordinaatti. Jos välipiste on aktiivisen pisteen yläpuolella, sijoitus tapahtuu, jos uuden sijainnin y-koordinaatti on suurempi, ja jos alapuolella, sijoitetaan, jos uuden sijainnin y-koordinaatti on pienempi. Tällä tavalla saadaan säilytettyä kaikilla annetuilla ympyrän keskipisteillä ympyrän ääriarvot, jolloin lopputuloksena saatava mesh on ulkonäöltään lähempänä sitä, mitä useilta peräkkäin asetelluilta ympyröiltä voisi intuitiivisesti odottaa (kuva 12).

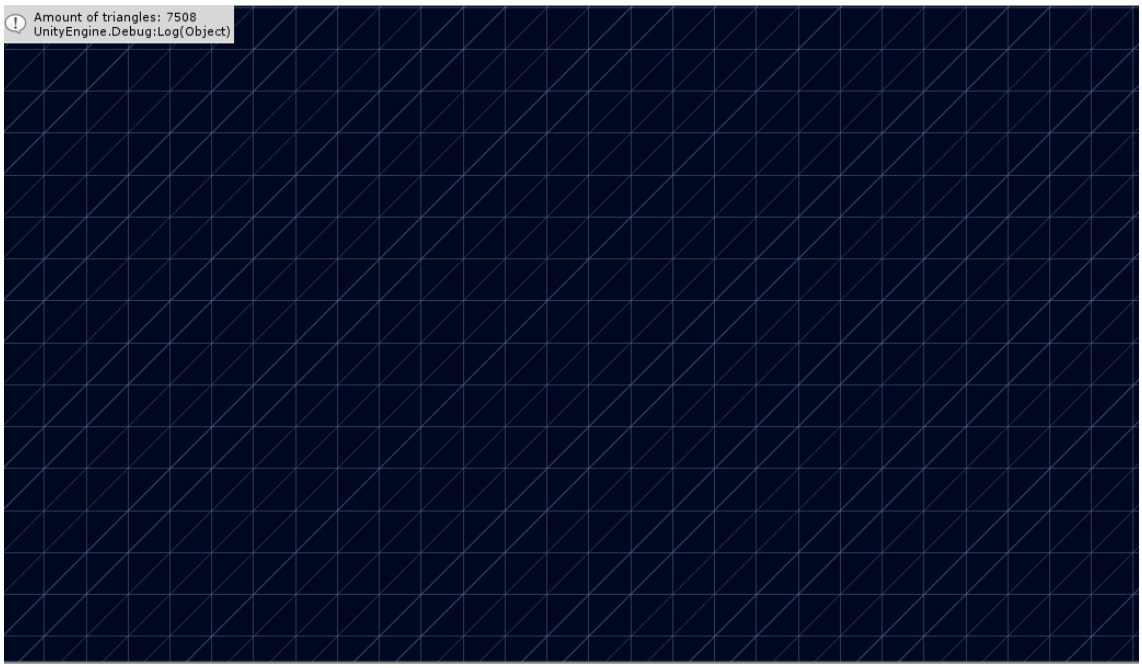


Kuva 12. Kun jokaisen peräkkäisen ympyrän ääriarvot saadaan säilytettyä, lopputulos on lähempänä sitä mitä työkalulta halutaan.

Kuten kuvasta 12 voidaan nähdä, toteutuksella saa näiden vaiheiden jälkeen tuotettua reaaliajassa miellyttävän pyöreää meshiä, jonka reuna etenee johdonmukaisesti. Lopputuloksena saatavassa meshissä on kuitenkin nähtävissä selkeä porrastus, joka johtuu siitä, että ympyrän keskipiste voi tällä erää sijoittua vain marching squares -kulmapisteisiin, jotka sijoittuvat kokonaislukukoordinaatteihin piirtoalueen sisällä. Meshin laatu todettiin kuitenkin ainakin tällä erää tarpeeksi hyväksi.

Seuraava ongelma, johon tarvittiin vastauksia, oli kuitenkin välittömästi nähtävissä. Työkalulla oli tarkoituksena luoda kenttiä peliin, joka on suunnattu mobiilialustalle. Unityn dokumentaatio suosittelee, että mobiilipeleissä tulisi tähdätä korkeintaan 100 000 pisteeseen tarpeeksi hyvän suorituskyvyn takaamiseksi [7], eli on edullista suorituskyvyn kannalta, että kenttä rakentuu mahdollisimman pienestä määrästä pisteitä. Naiivi marching squares -toteutus kuitenkin muodostaa kolmioinnin jokaiselle marching squares -ruudulle (kuva 13), jolloin lopputuloksena syntyvässä meshissä on suuri määrä tarpeettomia kolmioita ja pisteitä, kun samaan muotoon voitaisiin päästä myös paljon

kevyemmällä kolmioinnilla, joka vain yhdistäisi marching squaresilla aikaansaadun polygonin reunapisteet toisiinsa.



Kuva 13. Marching squares -algoritmin eräälle polygonille muodostamaa kolmiointia.

Seuraavaksi työssä lähdettiinkin tutkimaan, miten meshin muodostavien pisteiden (ja sitä kautta myös kolmioiden) määrää voitaisiin vähentää, ja sitä, olisiko se mahdollista tehdä reaaliajassa.

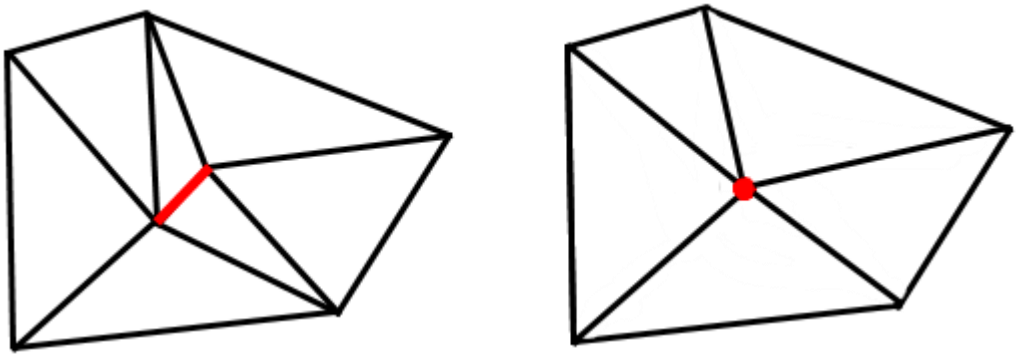
4 Meshin optimointi

Meshien optimoimista lähdettiin työn seuraavassa vaiheessa tutkimaan ilman mitään aiempaa kokemusta käsiteltävästä aiheesta. Optimointivaihetta varten lähdettiin aluksi etsimään algoritmia, jolla marching squares -algoritmin tuottamaa kolmiointia voitaisiin optimoida. Luvun aliluvuissa käsitellään algoritmit, joilla meshin optimoiminen pyrittiin työkaluun toteuttamaan.

4.1 Kokeilu 1: Edge collapse

Ensimmäinen löytynyt varteenotettava vaihtoehto lyhyen tutkimisen jälkeen oli edge collapse -algoritmi [8].

Algoritmin perusajatus vaikutti helposti ymmärrettävältä ja näytti etukäteen olevan myös suhteellisen nopeasti toteutettavissa. Edge collapse -algoritmi ottaa iteratiivisesti aina lyhimmän sivun, joka on polygonin muodon sisällä (sivun kumpikaan päätepiste ei saa siis olla polygonin kulmapiste, jotta polygonin muoto säilyy samana), ja yhdistää sivun päätypisteet yhteen ”romautettavan” sivun puoleenväliin, ja yhdistää myös kaikki päätypisteisiin yhdistyneet sivut tähän uuteen pisteeseen, jonka jälkeen molemmat ”romautetun” sivun päätypisteet voidaan poistaa pistejoukosta (kuva 14).

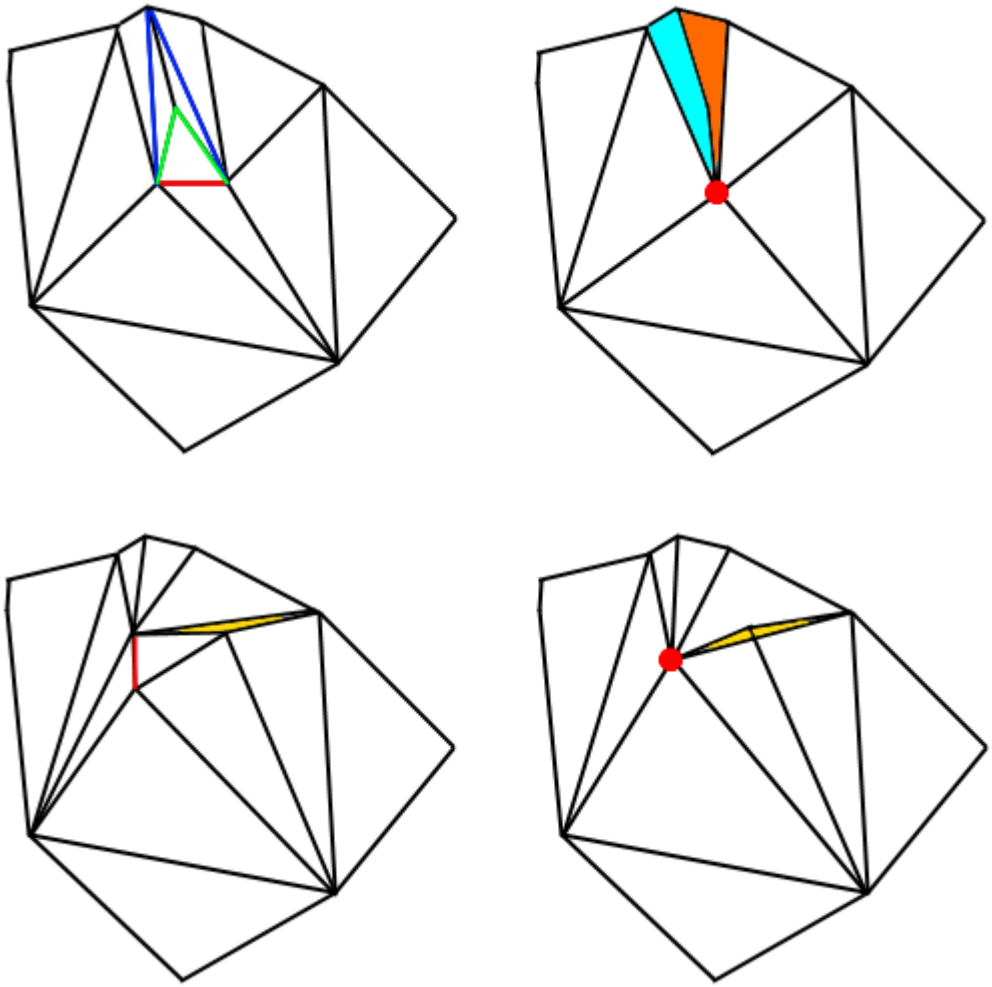


Kuva 14. Erittäin yksinkertainen edge collapse -tapaus mielivaltaisesti kolmioidulle polygonille, joka demonstroi pisteiden poistoa ja sivujen yhdistämistä.

Sivun päätypisteiden data voidaan yhdistää yhteen pisteeseen kuitenkin vain, jos tietyt ehdot täyttyvät [9]. Ensinnäkin, molemmin puolin ”romautettavaa” sivua voi olla vain yksi yhdistettävä sivupari kerrallaan (eli käytännössä vain yksi kolmio voidaan poistaa molemmilta puolilta käsiteltävää sivua yhden edge collapse -iteraation aikana). Jos yhdistettäviä sivupareja per puoli on useampia, kolmioinnin eheys voi vaarantua, ja sivuista on mahdollista muodostua monikulmioita, jotka eivät ole kolmioita. Toiseksi, kun kaikki käsiteltävän sivun päätepisteisiin yhdistyvät sivut yhdistetään uuteen pisteeseen, joka sijoittuu käsiteltävän sivun puoleenväliin, yksikään niistä kolmioista, jotka nämä sivut muodostavat, ei saa pyörähtää ympäri uudella päätepisteen sijainnilla. Jos näin pääsisi käymään, polygonin kolmiointiin voisi syntyä päällekkäisiä kolmioita, ja jälleen kerran olisi mahdollista muodostua monikulmioita, jotka eivät ole kolmioita, ja kolmioinnin eheys voisi näin vaarantua (kuva 15).

Ensimmäinen ongelma voitiin ratkaista tutkimalla, kuinka moni meshin kolmioista sisältää viittauksen tarkasteluun valittuun sivuun. Jos sivun sisältäviä kolmioita on enemmän kuin kaksi, edge collapse -operaatiota ei voida suorittaa tarkastellulle kolmiolle, ja käsittelyyn tulee etsiä meshin seuraavaksi lyhin sivu. Jälkimmäinen ongelma

voitiin taas tarkistaa tutkimalla kolmion tason normaalia vanhalla kulmapisteen sijainnilla ja uudella kulmapisteen sijainnilla. Jos kolmion tason suunta on muuttunut uudella kulmapisteen sijainnilla verrattuna vanhaan kulmapisteeseen, edge collapse -operaatiota ei voida suorittaa tarkasteltavalle sivulle ja tarkasteluun tulee jälleen etsiä meshin seuraavaksi lyhin sivu.



Kuva 15. Tilanteet, jossa edge collapse -operaatiota lyhyimmälle sivulle ei saa tehdä. Ensimmäisessä kuvaparissa näkyy tilanne, jossa kaksi sivuparia poistetaan saman operaation aikana. Toisessa kuvaparissa keltaisella väritetty kolmio pyörähtää ympäri, kun poistettavan sivun molempiin päätypisteisiin yhdistetyt sivut yhdistetään uuteen pisteeseen.

Hyvänä puolena edge collapse -optimointi toimi välittömästi ja antoi myös hyviä tuloksia: optimoidut kolmioinnit olivat laadukkaita ja kolmioiden määrä oli kolmioitavaan muotoon nähden sitä, mitä saattoi olettaa. Huonona puolena heti ensimmäisestä kokeilusta lähtien oli selvää, ettei algoritmi sopisi tämän työn käyttötarkoitukseen. Kuten algoritmin kuvauksestakin saattoi päätellä, se on voimakkaasti rekursiivinen, ja rekursiotasojen määrä kasvaa nopeasti datasetin kasvaessa. Koska marching squares -algoritmilla

muodostettu mesh sisältää suuren määrän tarkasteltavia sivuja, joista suurin osa on meshin muodon muodostamiseksi turhia, kasvaa datasetin ja siten myös rekursiotasojen määrä todella suureksi erittäin nopeasti tässä työssä kuvatun työkalun käyttötapauksessa, mikä käytännössä tarkoittaa siis sitä, että algoritmi optimoi meshin todella hitaasti. Algoritmia käytännössä toteutettaessa rekursio pitikin purkaa, jottei kutsupino ylivuoda suorituksen aikana, mutta vaikka tällä tavalla toteutuksesta saatiinkin teknisesti toimiva, kasvoi käsiteltävien sivujen määrä nopeasti niin suureksi, ettei algoritmia voinut pitää käyttötarkoitukseen tarpeeksi suorituskykyisenä.

Edge collapse -algoritmi osoittautuikin niin hitaaksi, että meshien optimointia varten päätettiin etsiä jokin toinen lähestymistapa ja algoritmi. Siihen käytetty aika ei tuntunut kuitenkaan täysin hukkaan heitetyltä. Ensinnäkin algoritmin toteutti suhteellisen nopeasti. Se oli myös toimiva ja mielenkiintoinen konsepti meshin sivujen optimointiin liittyen, ja lopulta se ohjasi ajattelua marching squaresilla muodostetun meshin optimoimisesta kohti seuraavaa optimointimenetelmää.

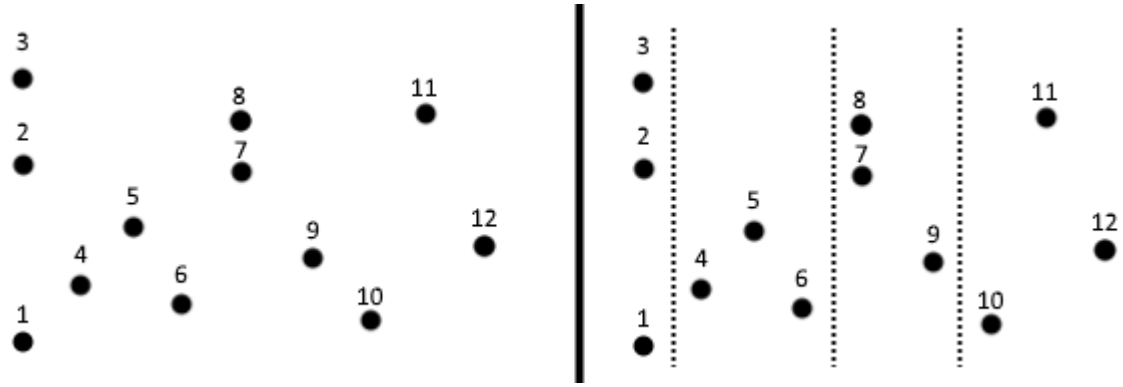
4.2 Kokeilu 2: Delaunay-kolmiointi

Edge collapse -kokeilu pakotti ajattelemaan hieman tarkemmin marching squares -algoritmin luonnetta ja sitä, miten meshin optimointi saataisiin suoritettua järkevämmin. Miksi alkaa optimoimaan olemassa olevaa erittäin epäoptimaalista kolmiointia, jossa kaikki polygonin muodon sisällä olevat pisteet poistetaan kumminkin optimoinnin jossain vaiheessa? Marching squares -algoritmin muodostaman meshin ulkopisteet ovat kuitenkin helposti saatavilla, joten miksi ei käytettäisi vain näitä pisteitä meshin muodon määrittämiseen, jonka jälkeen kolmiointi muodostettaisiin kyseiselle muodolle. Seuraava tutkimisen kohde olikin siis löytää algoritmi, joka muodostaisi kolmioinnin polygonille, jonka kulmapisteet ovat tiedossa.

Käytännössä vaihtoehtoja löytyi kaksi: ear clipping -algoritmi [10; 11.] ja Delaunay-kolmiointi [12], joista valittiin jälkimmäinen sen paremman suorituskyvyn vuoksi. Delaunay-kolmioinnin periaate on, että samalla tasolla sijaitsevalle pistejoukolle P on olemassa Delaunay-kolmiointi, jossa kun jokaisen kolmioinnin kolmion kärkipisteiden kautta piirretään ympyrä, niin yksikään joukon P sisältämä piste ei sijaitse yhdenkään tällaisen ympyrän määrittämän alueen sisällä. Delaunay-kolmiointi on mahdollista toteuttaa useilla eri tavoilla, joista tarkasteluun valittiin divide and conquer (jaa ja valloita)

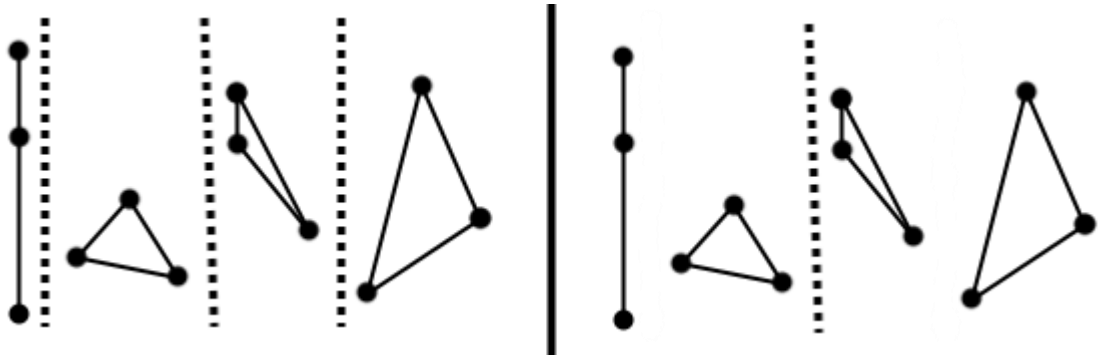
-algoritmi, koska sen on todistettu olevan nopein tapa muodostaa Delaunay-kolmiointi polygonille. [13; 14.]

Perusmuodossaan divide and conquer -algoritmi [15] laskee Delaunay-kolmioinnin pienimmälle konveksille joukolle, joka sisältää kaikki pisteet sille syötteenä annetussa pistejoukossa. Syötteenä annetaan pistejoukko, joka on järjestettynä ensin x-koordinaatin, ja sen jälkeen y-koordinaatin mukaiseen järjestykseen. Tämän jälkeen pistejoukko jaetaan niin kauan pienempiin osiin, kunnes jäljellä on enää osapistejoukkoja, jotka sisältävät korkeintaan kolme pistettä (kuva 16).



Kuva 16. Divide and conquer -algoritmin ensimmäinen vaihe: jako osajoukkoihin.

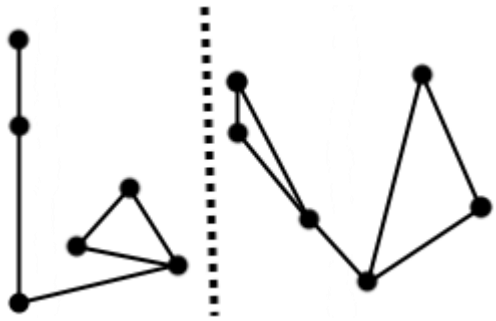
Luodut osajoukot voidaan välittömästi kolmioida kolmen pisteen tapauksissa tai niiden sisältämien pisteiden välille voidaan vetää suora kolmen x- tai y-suunnassa peräkkäisen pisteen tai kahden pisteen tapauksissa. Tämän vaiheen jälkeen jokaisella iteraatiolla kaksi vierekkäistä osapistejoukkoa yhdistetään toisiinsa (kuva 17).



Kuva 17. Edellisessä vaiheessa luotujen osajoukkojen pisteiden yhdistäminen ja uudet osajoukot.

Osajoukkoja yhdistäessä vasemmanpuoleisen osajoukon sivut, joita kutsutaan tässä VV-sivuiksi, yhdistetään oikeanpuoleisen osajoukon sivuihin, joita tässä kutsutaan nimellä OO-sivut. Osajoukkoja yhdistäviä uusia sivuja kutsutaan vuorostaan VO-sivuiksi. Osajoukkoja yhdistettäessä on mahdollista, että VV- tai OO-sivuja joudutaan poistamaan, mutta uusia VV- tai OO-sivuja ei luoda ikinä osajoukkoja yhdistettäessä. Osajoukkojen yhdistämiseen tarvittavat operaatiot ovat siis VO-sivujen luonti ja VV- ja OO-sivujen poisto.

Ensimmäinen vaihe osajoukkoja yhdistettäessä on VO-kantasivun luonti. VO-kantasivu muodostetaan yhdistettävien osajoukkojen sellaisten y-arvoltaan pienimpien pisteiden välille, joiden välille muodostuva sivu ei leikkaa minkään VV- tai OO-sivun kanssa (kuva 18).



Kuva 18. Osajoukkojen yhdistämisen ensimmäinen vaihe, osajoukkojen välille lisätään VO-kantasivut

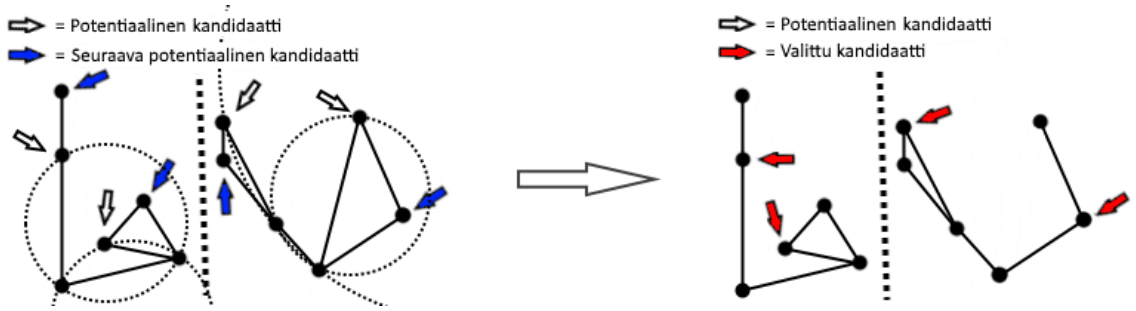
Seuraavaksi määritetään VO-kantasivuun yhdistyvä seuraava VO-sivu (eli liikumme alhaalta ylöspäin). Uuden sivun yksi päätepiste on siis jompikumpi VO-sivun päätepisteistä, ja tästä riippuen uuden sivun toinen päätepiste on joko vasemman- tai oikeanpuoleisessa osajoukossa. Valitsemmekin seuraavaksi molemmista osajoukoista parhaan mahdollisen kandidaatin toiseksi uuden sivun päätepisteeksi ja vertaamme tämän jälkeen, että kumpi kandidaateista on sopivampi, jolloin saamme selville uuden sivun molemmat päätepisteet.

Ensimmäinen potentiaalinen kandidaatti on VO-kantasivun päätepisteeseen VV- tai OO-sivulla yhdistyvä piste, joka muodostaa pienimmän myötäpäiväisen kulman VO-kantasivun kanssa oikean osajoukon tapauksessa, tai vastapäiväisen kulman vasemman osajoukon tapauksessa. Seuraava potentiaalinen kandidaatti on piste, joka

muodostaa toiseksi pienimmän kulman kantasivun kanssa. Potentiaalisen kandidaatin sopivuudelle on kaksi kriteeriä:

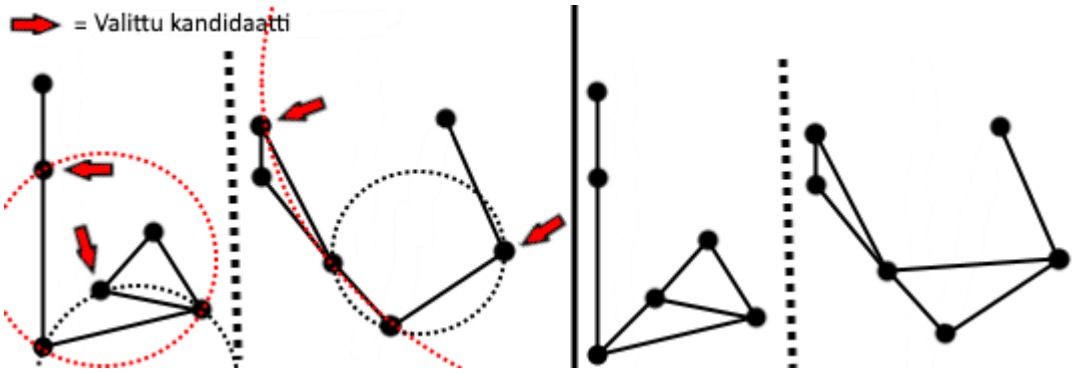
- Sivun, jonka päätepiste potentiaalinen kandidaatti on, ja VO-kantasivun välinen kulma tulee olla alle 180 astetta.
- Ympyrä, jonka kehäpisteitä VO-kantasivun molemmat päätepisteet ja potentiaalinen kandidaatti ovat, ei saa sisältää seuraavaa potentiaalista kandidaattia sen rajaaman alueen sisällä.

Jos potentiaalinen kandidaatti täyttää molemmat näistä kriteereistä, siitä tulee kyseisen puolen osajoukon kandidaatti uuden VO-sivun toiseksi päätepisteeksi. Jos ensimmäinen kriteeri ei täyty, tutkitun puolen osajoukolle ei valita kandidaattia kyseisellä iteraatiolla. Jos ensimmäinen kriteeri täyttyy, mutta toinen ei, niin VV- tai OO-sivu, joka yhdistää potentiaalisen kandidaatin VO-kantasivun päätepisteeseen poistetaan. Tämän jälkeen prosessia toistetaan, kunnes lopullinen kandidaatti löydetään tai kunnes potentiaalisia kandidaatteja ei enää ole jäljellä (kuva 19).



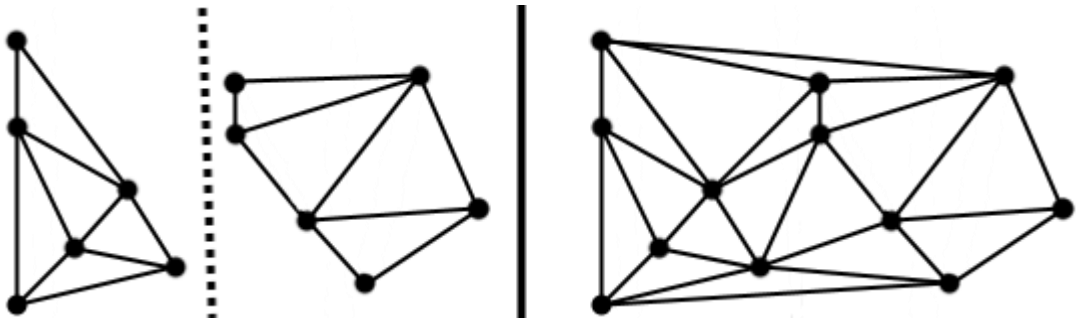
Kuva 19. Kandidaattien valinta

Jos kummankaan puolen osajoukosta ei löydy kandidaattia, osajoukkojen yhdistäminen on suoritettu. Jos taas vain toisesta osajoukosta löydetään kandidaatti, valitaan se suoraan uuden VO-sivun toiseksi päätepisteeksi. Jos molemmista osajoukoista vuorostaan löydetään potentiaalinen kandidaatti, vertaillaan, kumman osajoukon kandidaatti on parempi uuden VO-sivun toiseksi päätepisteeksi. Vertailu toimii samalla logiikalla kuin osajoukkojen sisäisten kandidaattien valinta: jos ympyrä, jonka kehäpisteitä VO-kantasivun molemmat päätepisteet ja kandidaatti ovat, sisältää toisen osajoukon valitun kandidaatin, valitaan toisen osajoukon kandidaatti lopulliseksi kandidaatiksi uuden VO-sivun toiseksi päätepisteeksi (kuva 20).



Kuva 20. Lopullisen kandidaatin valinta ja uudet VO-sivut

Tämän jälkeen prosessia toistetaan jokaiselle uudelle VO-sivulle, kunnes kandidaatteja ei enää löydy (kuva 21).



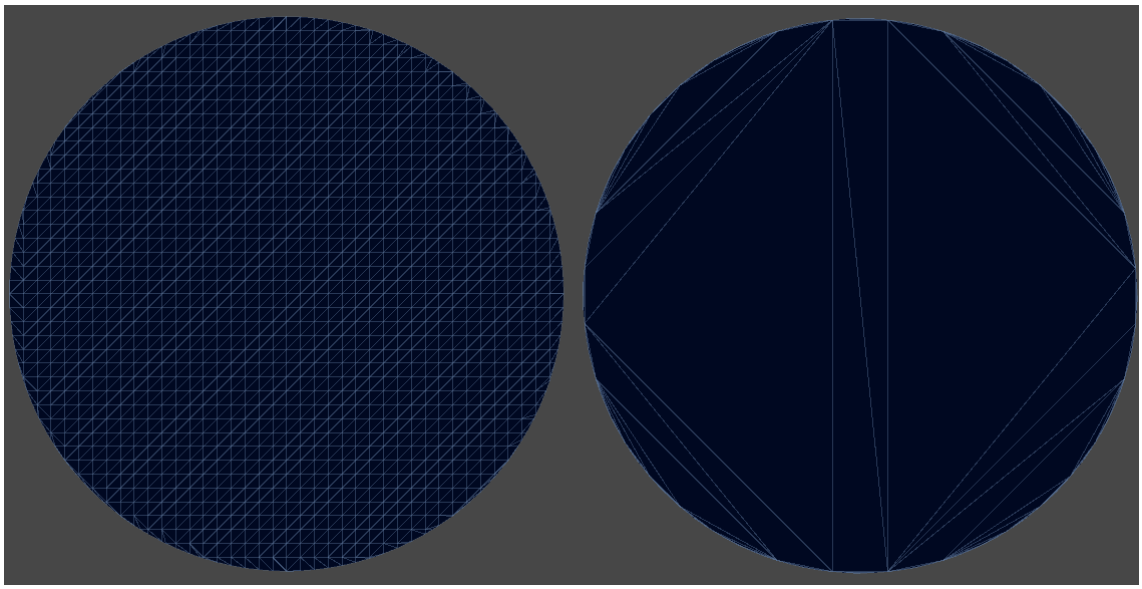
Kuva 21. Osajoukot yhdistettynä ja esimerkkipistejoukon lopullinen kolmiointi.

Koska perusmuodossaan Delaunay-kolmiointi tekee kolmioinnin vain konveksille ulkomuodolle, niin jos haluamme, että työkalulla pystytään muodostamaan myös konkaaveja polygoneja, joiden jokainen sisäkulma ei tarvitse olla alle 180-astetta, joudumme lisäämään kolmiointialgoritmiin rajoitteita. Tätä kutsutaan rajoitetuksi Delaunay-kolmioinniksi. Rajoitetulla Delaunay-kolmiointialgoritmilla luotu kolmiointi voi sisältää Delaunay-kolmion määritelmän täyttäviä kolmioita, mutta se ei takaa, että kaikki kolmioinnin sisältämät kolmiot täyttäisivät Delaunay-kolmion määritelmän. Rajoite määrittää käytännössä, että kahden määritetyn pisteen välillä tulee olla sivu. Tämän jälkeen rajoitesivun tieltä poistetaan kaikki sen kanssa leikkaavat Delaunay-kolmioinnin luomat sivut. Toimenpide muodostaa polygonin sisään ”rajoitepolygonin”. Rajoitesivun luonnin jälkeen rajoitepolygoni kolmioidaan geneerisesti uudelleen rajoitesivun kanssa. Mikään rajoitepolygonin sisään jäävä kolmio ei enää tämän jälkeen täytä puhtaan Delaunay-kolmioinnin vaatimuksia. Polygonin konkaavit kulmat käsitellään kolmioinnissa käytännössä samalla tavalla kuin polygonin sisällä olevat reiät: ne

käsitellään polygonin sisäisinä rajoitepolygoneina. Kun rajoitepolygonin rajoitesivut on luotu, kaikki rajoitepolygonin muut sivut tuhotaan ”tartuttamalla ne kolmioita syövällä viruksella” [16], jolloin jäljelle jää vain haluttu rajoitettu muoto.

Työkalua varten alettiin aluksi luomaan omaa toteutusta Delaunay-kolmioinnista divide and conquer -algoritmin perustoiminnallisuuden kuvauksen pohjalta, mutta kun algoritmin ymmärtämiseen ja toteutukseen oli käytetty pari päivää ilman, että toiminnallisuus edes normaalille konkaaville tapaukselle oli valmiina, päätettiin oman toteutuksen kehittäminen keskeyttää, jotta työkalu saataisiin toimintakuntoon järkevässä ajassa (optimointi oli kuitenkin vain yksi osa työkalun kokonaistoiminnallisuudesta). Ratkaisuna työkaluun otettiin käyttöön Triangle.NET-kirjasto [17], joka muodostaa rajoitettuja Delaunay-kolmiointeja samassa tasossa sijaitsevalle pistejoukolle. Ainoa käytännön ongelma Triangle.NET-kirjaston käyttöönottamisessa Unityn kanssa oli se, että Triangle.NET on kirjoitettu .NET-versiota 4 käyttäen, kun taas Unity tukee vakioasetuksissaan .NET-versiota 3.5, joka tarkoittaa yhteensopivuusongelmia kirjastoa käyttöönotettaessa. Kirjoitushetkellä uusin Unityn versiopäivitys 2017.1 toi kuitenkin mukanaan kokeellisen tuen .NET-versiolle 4.6, joka pitää asettaa päälle asetuksista sijainnista Build Settings > Player Settings > Other Settings > Configuration > Scripting Runtime Version. Muutoksen jälkeen Triangle.NET-kirjaston voi tuoda Unityyn ilman yhteensopivuusongelmia.

Triangle.NET-kirjastolla sai aikaan laadukkaita kolmiointeja (kuva 22) ja kirjaston kolmiointialgoritmi (kirjasto käyttää edellä kuvattua divide and conquer -algoritmia vakioasetuksena, vaikka siihen on sisällytetty myös muitakin kolmiointialgoritmeja) oli nopea ja tehokas. Kirjasto kuitenkin muodostaa kolmioinnit yksittäisille polygoneille, ja kirjaston kolmiointimetodi tarvitsee syötteenään polygonin reunapisteet järjestettynä myötäpäivään. Työn seuraavassa vaiheessa tarvitsikin selvittää, miten marching squares -algoritmilla luodun pistedatan voisi erotella erillisiksi polygoneiksi ennen kolmiointialgoritmille vientiä ja että miten polygonin muodon määrittävät reunapisteet voisi järjestää myötäpäiväiseen järjestykseen.



Kuva 22. Vasemmalla alkuperäinen marching squares-kolmiointi, oikealla samalle muodolle tehty rajoitettu Delaunay-kolmiointi. Vasemmanpuoleisessa kuviossa on 2676 kolmiota ja oikeanpuoleisessa vuorostaan 162, eli kolmioiden määrä saatiin optimoinnilla pudotettua noin kuudestoistaosaan alkuperäisestä.

5 Pistedatan käsittely

Kirjaston käyttöönoton jälkeen työkalulla pystyttiin tuottamaan optimoituja kolmiointeja, ja kolmioinnit saatiin muodostettua suhteellisen nopeasti. Samaan aikaan aiemmasta tavoitteesta kolmioinnin reaaliaikaisuuteen jouduttiin kuitenkin käytännössä luopumaan, sillä ennen kolmiointialgoritmille vientiä meshin sisältämät polygonit tulee erotella toisistaan (täytyi olla mahdollista, että työkalulla luotu kenttä sisältää useamman kuin yhden polygonin). Tämän lisäksi jokaisen yksittäisen polygonin kulmapisteet tulee järjestää myötäpäivään, jotta kolmiointialgoritmi toimii.

Vaikka kolmiointialgoritmi muodostaisikin kolmioinnin polygoneille tarpeeksi nopeasti (varsinkin, jos kolmiointi voitaisiin rajoittaa vain osiin jotka tarvitsevat uudelleenkolmiointia, eli käyttäjän painalluksen vaikutusalueelle), tuomittiin pistedatalle tehtävä esikäsittely joka tapauksessa niin raskaaksi, että reaaliaikaisuudesta oli parempi ainakin tässä vaiheessa luopua. Myöskään työskentelyn sujuvuuden kannalta ajatus siitä, että koko kenttä piirrettäisiin valmiiksi ennen optimointia, ei tuntunut mahdottomalta ajatukselta, varsinkin kun kentälle tuli lisätä myös törmäyspinnat, joiden lisääminen oli ajatuksena yksinkertaisinta optimoinnin loppuvaiheessa valmiiksi piirrettyyn tasoon.

Pistedatan käsittelemiseksi työkalun tarvitsi reaaliajassa käydä läpi seuraavat vaiheet:

- Käyttäjän painalluksen talteenotto.
- Marching squares -kulmapisteiden aktivointi ja välipisteiden interpolointi.
- Marching squares -neliöiden konfiguraatioiden tutkiminen ja kolmiointi.

Kun haluttu mesh oli saatu luotua, käytiin viimeisteltäessä meshiä lopulliseksi pelin tasoksi läpi vuorostaan seuraavat vaiheet:

- Polygonien erottelu.
- Polygonin muodon sisällä olevien pisteiden poisto pistedatasta ja suorien etsintä.
- Polygonin reunapisteiden järjestäminen myötäpäiväiseen järjestykseen Triangle.net-kirjaston kolmiointimetodia varten.
- Kolmiointi.
- Lopullisen kenttäobjektin luonti ja törmäyspintojen rakentaminen tasoon.

Käytännössä ”putkesta”, jonka käyttäjän painallus käy läpi ennen päätymistään valmiiseen tasoon, tuli siis seuraavanlainen:

1. Käyttäjän painallus piirtoalueen sisällä olevassa liukulukukoordinaatissa pyöristetään alaspäin lähimpään kokonaislukukoordinaattiin. Koska todettiin, että ainoa työkaluun tarvittava muoto on ympyrä, muodostuu käyttäjäpalautteesta aina ympyräapproksimaatio käyttäjän määrittämällä säteellä, jonka keskipiste sijaitsee käyttäjän painalluksen määrittämässä kokonaislukukoordinaatissa. Tämän jälkeen kaikki ympyrän säteen sisällä sijaitsevat kulmapisteet asetetaan aktiivisiksi, jos ne eivät jo olleet aktiivisia, tai vaihtoehtoisesti poisto-operaation tapauksessa ympyrän säteen sisällä sijaitsevat kulmapisteet asetetaan epäaktiivisiksi.

2. Kohdassa 1 määritetty ympyrän säteen alue merkitään interpolaatioalueeksi, joka varmistaa sen, että interpolointi suoritetaan vain painalluksen vaikutusalueella sijaitseville pisteille. Interpolaatioalueen sisällä olevilta kulmapisteiltä tutkitaan, onko niiden vieressä epäaktiivisia kulmapisteitä. Jos on, niin aktiivisen ja epäaktiivisen kulmapisteen välisen välipisteen sijaintia interpoloidaan niin kauan, kunnes sen etäisyys ympyrän keskipisteestä ylittää ympyrän kehän. Tämän jälkeen välipiste asetetaan edelliseen sijaintiin, joka oli vielä ympyrän kehän sisällä, jolloin välipiste sijaitsee niin lähellä ympyrän todellista kehäpistettä, kuin interpolaatioaskeleen määrittämissä rajoissa on vain mahdollista.

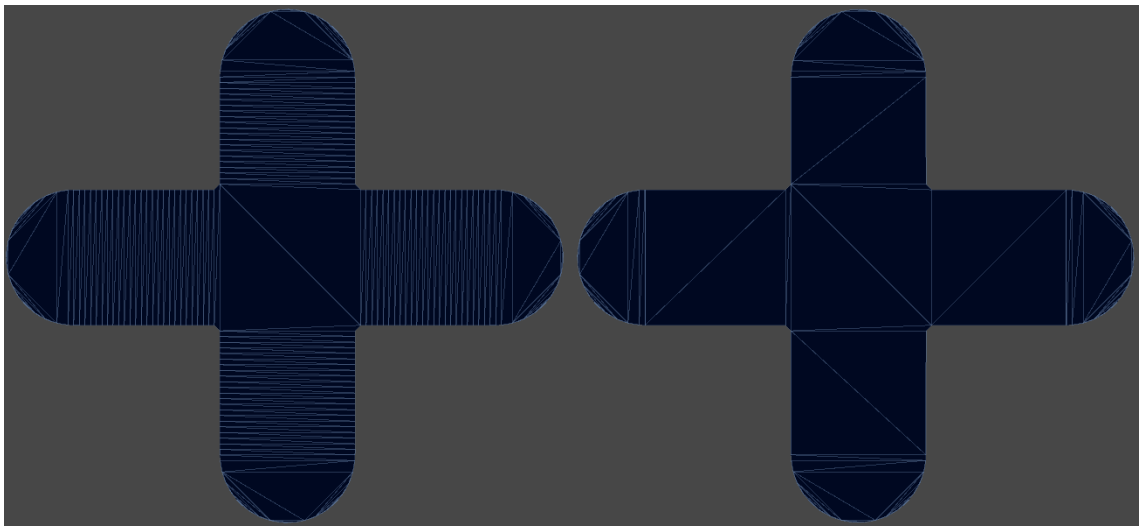
3. Jokaisen painalluksen jälkeen pistedata kolmioidaan marching squares -algoritmilla, joka antaa tarkan visuaalisen presentaation käyttäjän luoman meshin ulkomuodosta käyttäjälle. Vaiheita 1-3 jatketaan, kunnes käyttäjä haluaa viimeistellä tason painaen viimeistelynäppäintä.

4. Viimeistelynäppäimen painalluksen jälkeen pistedata annetaan yhtenä listana polygonien erottelumetodille, joka erottelee polygonit toisistaan, ja antaa palautteena listan listoja, yksittäisen listan sisältäessä kaikki yhteen polygoniin sisältyvät pisteet. Metodin toiminta on yksinkertainen. Se ottaa tutkittavakseen yksittäisen aktiivisen marching squares -kulmapisteen ja kaikki sen pääilmansuunnissa sijaitsevat aktiiviset naapurikulmapisteet, jonka jälkeen se tutkii, kuuluuko jokin näistä pisteistä jo olemassa olevaan polygoniin. Jos jokin naapuripiste ei ole aktiivinen, tarkastelujoukkoon lisätään aktiivisen ja epäaktiivisen pisteen välinen interpoloitu välipiste. Jos mikään pisteistä ei kuulu mihinkään olemassa olevaan polygoniin, luovat ne yhdessä uuden polygonin. Jos joku tai useampi pisteistä kuuluu jo olemassa olevaan polygoniin, sijoitetaan kaikki tarkastelujoukon pisteistä tähän polygoniin. Jos piste vuorostaan yhdistää useampia olemassa olevia polygoneja, valitaan näistä ensimmäisenä luotu (polygoni, jolla on polygonitaulukossa pienin indeksi) ja liitetään kaikki tarkastelujoukon ja muiden löydettyjen polygonien pisteet valittuun polygoniin.

5. Polygonien erottelun jälkeen listat polygonien sisältämistä pisteistä annetaan metodille, joka karsii pistejoukosta kaikki pisteet, jotka eivät ole polygonin reunapisteitä. Marching squaresilla luodulla pistedatalla reunapisteiden erottelu on helppoa, jos piste on kahden kulmapisteen välinen välipiste, se sijaitsee polygonin reunalla. Kaikki marching squares -kulmapisteet voidaan puolestaan poistaa pistejoukosta. Ainoa tapaus, missä kulmapiste on polygonin reunapiste, on silloin, jos aktiivinen piste sijaitsee määritellyn piirtoalueen rajalla. Kaikki piirtoalueen rajalla sijaitsevat aktiiviset kulmapisteet sisällytetään siis myös polygonin reunapistedataan. Tässä vaiheessa pistejoukkoa voidaan kuitenkin karsia vielä enemmänkin.

    5.1. Jos esimerkiksi kolme peräkkäistä kulmapistettä sijaitsee pistealueen rajalla, ne muodostavat yhdessä suoran. Suoran ilmaisemiseksi ei kuitenkaan tietenkään tarvitse kuin kaksi pistettä, joten keskimmäinen näistä pisteistä voidaan poistaa. Onkin optimoinnin kannalta tarpeellista poistaa piirtoalueen reunoilta kaikki tarpeettomat pisteet kahden päätepisteen välillä, jolloin kolmioinnissa ei tarvitse

ottaa tarpeettomia pisteitä huomioon ja säästetään kolmioiden määrässä. Työkalun suorittama suorien linjojen etsintä ja tarpeettomien välipisteiden poisto on kuvattu tarkemmin liitteessä 1. Lopputuloksena polygonin reunoilta saadaan karsittua kaikki muodon ilmaisemisen kannalta tarpeettomat välipisteet (kuva 23).



Kuva 23. Saman polygonin kolmiointi ilman suorien linjojen tarkistusta ja suorien linjojen tarkistuksen kanssa.

6. Kun pistejoukossa on jäljellä enää polygonin muodon ilmaisemiseen tarvittavat kulmapisteet, ne järjestetään myötäpäiväiseen järjestykseen. Aloituspisteeksi valitaan polygonin piste, jolla on pienin x-koordinaatti. Koska kohdassa 5 tutkittiin suoran sivun määrittävät pisteparit, tutkitaan jokaiselle pisteelle, onko sille määritetty pisteparia. Jos pistepari löytyy, liitetään piste suoraan pariinsa. Jos tutkittavalle pisteelle ei löydy paria, myötäpäiväinen järjestys ratkotaan yksinkertaisena kauppamatkustajan ongelmana: jokaiselle pisteelle etsitään lähin naapuri, joka valitaan seuraavaksi tutkittavaksi pisteeksi. Tällöin tulee totta kai varmistaa, että algoritmi alkaa käymään muotoa läpi myötäpäiväisesti. Tämä pyrittiin varmistamaan kolmella muuttujalla. Kuvion neljä ensimmäistä pistettä tulee valita niin, että seuraavalla pisteellä on korkeampi tai yhtä korkea y-koordinaatti kuin edellisellä, jonka jälkeen jokaiselle pisteelle etsitään vain normaalisti lähin naapuri. Kuitenkin, jos näiden ensimmäisen neljän pisteen aikana saavutetaan polygonin korkein y-koordinaatti, aletaan kyseisestä pisteestä eteenpäin tutkia vain lähintä naapuria. Toinen erikoistapaus on, jos neljän ensimmäisen pisteen aikana jollekin tutkittavista pisteistä löydetään pistepari (eli välille on määritetty suora sivu), jolloin pisteparin

päätepisteestä eteenpäin tutkitaan normaalisti lähintä naapuria. Vaikka on selvää, että toteutus on matemaattisesti vähintäänkin epävakaalla pohjalla, niin se antoi kuitenkin lähes kaikilla kokeiluilla toivotun lopputuloksen, koska mesh luodaan käytännössä ympyröistä, jolloin pienimmän x-koordinaatin omaava piste on muodon puolivälissä, jolloin sillä on lähes kaikissa tapauksissa pisteitä suuremmalla y-koordinaatilla johon siirtyä. Jos reuna ei ole pyöreä, pisteeseen on käytännössä aina siirrytty pisteparin päätepisteenä, joka on otettu huomioon pistepariehdossa. Lähimmän naapurin käyttäminen muodostuu kuitenkin ongelmaksi kapeiden terävien muotojen kohdalla, jolloin kulma saattaa leikkautua pois lähimpään naapuripisteeseen yhdistettäessä (kuva 24). Totesimme kuitenkin, ettei työkalun tarvitse mallintaa tällaisia muotoja, joten puutetta ei lähdetty korjaamaan.

Kuva 24. Tilanne, jossa lähin naapuri-algoritmi ei tuota toivottua tulosta. Vasemmalla on marching squaresilla luotu alkuperäinen mesh, oikealla taas mesh optimoinnin jälkeen. Sirpin yläreunassa on nähtävissä, kuinka algoritmi yhdistää epätoivotut pisteet toisiinsa.

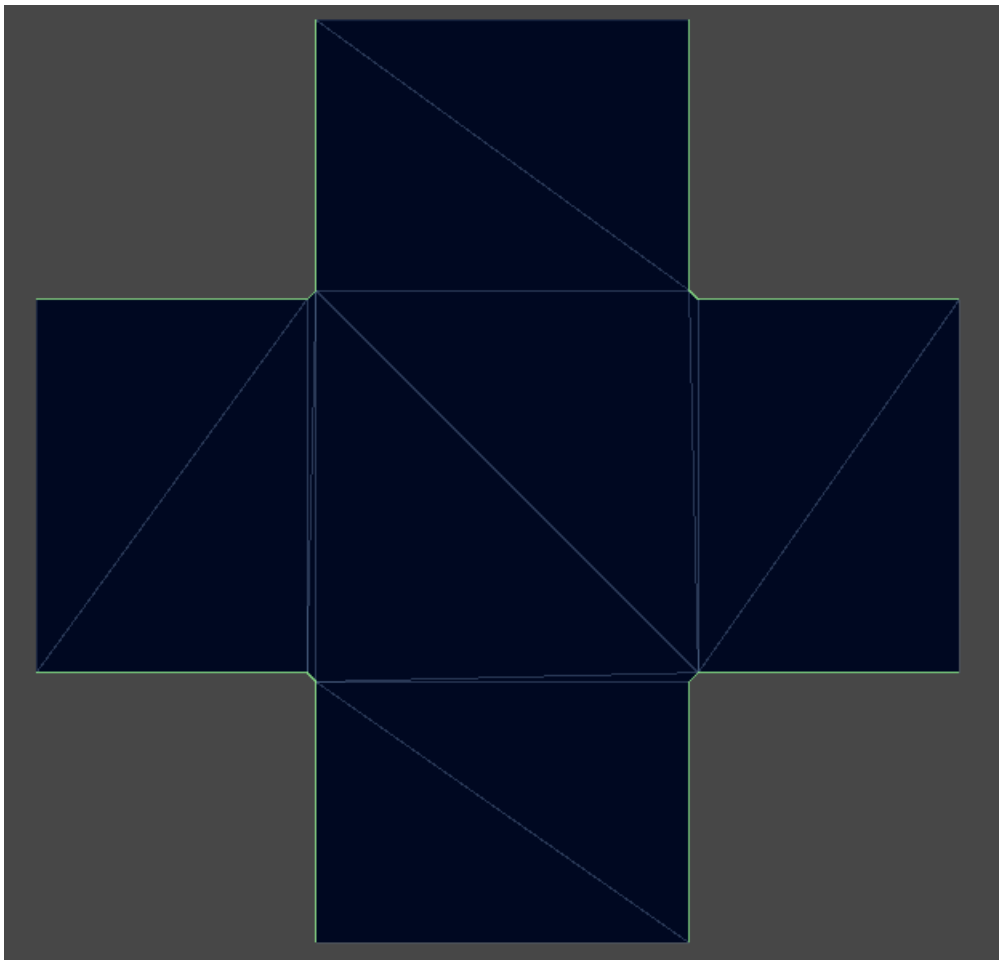
7. Kun polygonin pisteet on saatu järjestettyä myötäpäiväiseen järjestykseen, pisteet annetaan Triangle.NET-kirjaston kolmiointimetodin käsiteltäviksi, josta palautteena saadaan lopullisen kolmioinnin sisältämät pisteet ja näihin pisteisiin viittaavat kolmiot listoina.

8. Kun vaiheet 4-7 on saatu suoritettua kaikille polygoneille, luodaan viimeisessä vaiheessa varsinainen Unityn kenttäobjekti ja lisätään sille tarvittavat törmäyspinnat.

Törmäyspinnat luotiin tässä työkalussa käyttäen Unityn EdgeCollider2D-komponenttia, joka luo törmäyspintasivun jokaisen sille syötteenä annetun pisteen välille. Törmäyspinnat rakennettiin käyttäen vaiheessa 6 luotua myötäpäiväisesti järjestettyä pistetaulukkoa. Jokaisen yksittäisen polygonin ääriviivalle luodaan oma törmäyspinta. Koska törmäyspintoja haluttiin optimoida niin, ettei piirtoalueen ulkorajalla sijaitseville sivuille luoda turhaa törmäyspintaa (sivut sijaitsevat tason ulkopuolella, joten törmäystarkistusta näille pinnoille ei tarvita), voi myös yksittäisellä polygonilla olla useampi kuin yksi törmäyspinta. Polygonien törmäyspintojen luonti toteutettiin siis seuraavilla ehdoilla:

- Jokaista polygonia kohti luodaan uusi EdgeCollider2D-komponentti.
- Jos tarkasteltava piste ei ole piirtoalueen reunalla, se lisätään tällä hetkellä käsittelyssä olevaan collider-komponenttiin.
- Jos tarkasteltava piste on piirtoalueen reunalla, mutta sitä edellinen piste ei ole piirtoalueen reunalla (käytännössä piste siis päättää käsittelyssä olevan colliderin), tarkasteltava piste lisätään tällä hetkellä käsittelyssä olevaan collider-komponenttiin.
- Jos tarkasteltava piste on piirtoalueen reunalla ja sitä edeltävä piste oli myös piirtoalueen reunalla, mutta seuraava piste ei ole piirtoalueen reunalla (käytännössä piste siis aloittaa uuden colliderin), lisätään uusi collider-komponentti, johon piste lisätään, jos käsittelyssä olevaan komponenttiin on jo lisätty pisteitä (tällä vältetään mahdolliset tyhjät collider-komponentit).
- Jos kaksi peräkkäistä pistettä sijaitsee piirtoalueen reunalla ja ne määrittävät sivun, joka ei kulje piirtoalueen reunaa pitkin, lisätään kyseiset pisteet uuteen collider-komponenttiin, jos käsittelyssä olevaan komponenttiin on jo lisätty pisteitä (käytännössä pisteiden määrittämä sivu muodostaa yhden kokonaisen colliderin).

Kun törmäyspinnat on saatu luotua, kolmiointialgoritmin muodostama piste- ja kolmiodata asetetaan kenttäobjektin Mesh-komponentille. Näiden vaiheiden jälkeen Unityyn on luotu kenttäobjekti, jota voidaan kokeilla Unityn pelitilassa (kuva 25).



Kuva 25. Unityyn luotu kenttäobjekti törmäyspintojen kanssa.

6 Editorin muut ominaisuudet

Työkalulla pystyttiin siis tässä vaiheessa luomaan Unityssä toimiva kenttäobjekti törmäyspintojen kanssa. Työkaluun lisättiin kuitenkin vielä muutama ominaisuus toiminnallisuuden täydentämiseksi ja kenttien rakentamisen helpottamiseksi.

Jotta meshien luonti helpottuisi, varsinkin suorien linjojen muodostamisen osalta, editoriin lisättiin mahdollisuus lukita koordinaatti tiettyyn sijaintiin x- tai y-suunnassa. Kun käyttäjä painaa shift-näppäimen pohjaan, y-koordinaatti lukitaan senhetkiseen hiiren sijaintiin y-akselilla, ja kaikissa lisäys- ja poisto-operaatioissa käytetään hiiren todellisen y-sijainnin sijasta lukitsemishetkellä tallennettua y-sijaintia niin kauan, kun shift pidetään pohjassa. Control-näppäin vuorostaan lukitsee vastaavasti x-koordinaatin.

Koska Unityn Scene-tiedostoa tallennettaessa työkalulla luotu mesh ei samalla tallentunut Scene-tiedostoon, tarvittiin työkaluun tallennusmekanismi, jotta aiemmin luotua meshiä voidaan palata muokkaamaan myöhemmin. Aluksi mesh yritettiin tallentaa liittämällä se peliobjektiin, joka tallennettiin projektin tiedostoihin prefabina. Peliobjektiin sisältyvä mesh ei kuitenkaan ainakaan tätä työtä tehdessä tehtyjen kokeilujen perusteella tallennu peliobjektin mukana. Tämän sijaan, kun pelkän meshin tallensi yksittäisenä .asset-tiedostona, niin kaikki meshin sisältämä data tallentui oikein. Tallennus toteutettiinkin niin, että tallennus loi kaksi tiedostoa:

- .prefab-tiedoston, joka on käytännössä Unityn peliobjekti, johon on lisätty komponenttina meshin tilan uudelleenluomiseen tarvittavan datan säilytysluokka. Luokka sisältää kaiken serializable-datan, mitä tarvitaan, jotta mesh voidaan rakentaa työkaluun uudelleen. Luokka sisältää myös polun meshiin eli kenttäeditorissa samaan aikaan tallennettuun .asset-tiedostoon.
- .asset-tiedoston, joka sisältää pelkästään luodun meshin. Oikea mesh-tiedosto ladataan kenttäeditoriin säilytysluokasta löytyvän viittauksen perusteella.

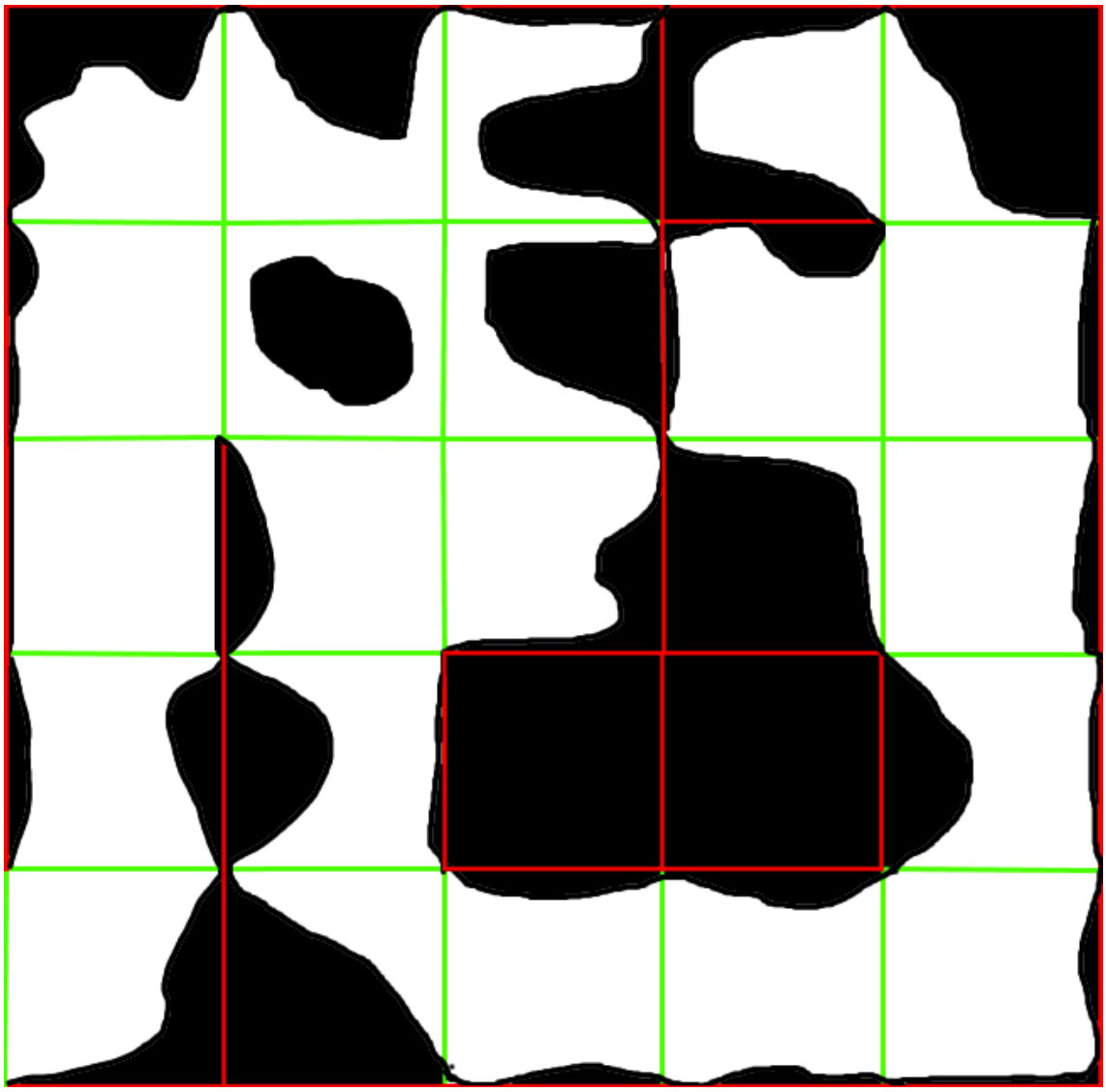
Yksi haluttu ominaisuus oli peliobjektien lisääminen pikanäppäimillä editorinäkymään, jotta kenttien luomisesta saataisiin mahdollisimman nopeaa ja yksinkertaista. Jotta pikanäppäimet olisivat editorinäkymässä käyttäjän muokattavissa, päätettiin pikanäppäimet toteuttaa <char, GameObject> -pareina, jolloin chariksi määritellään haluttu näppäimistön näppäin, ja GameObjectiksi asetetaan peliobjekti, joka kyseisellä näppäimellä halutaan luoda. C#-listaluokka Dictionary ei ole kuitenkaan Unity serializable, jonka takia sitä ei voida esittää editorinäkymässä. Tästä syystä listaa varten määritettiin oma struct-tyyppi KeyShortcut, joka sisältää charin ja GameObjectin. KeyShortcut määritettiin kentällä System.Serializable, jolloin se on esitettävissä Unityn editorissa, jos se on määritetty julkisena. Tämän jälkeen pikanäppäimet voitiin esittää editorissa listana KeyShortcutteja.

Seuraavaksi pikanäppäinten jatkuvaan kuunteluun tarvittiin jokin keino. Käyttäjäpalaute pystytään kaappaamaan muokatussa editoriluokassa, kuten jo aiemmin todettiin. Koska Unity kuitenkin tuhoaa editoriluokan instanssin aina, kun editorinäkymässä vaihdetaan johonkin toiseen ikkunaan, ei tämä soveltunut suoraan kovin hyvin pikanäppäinten kuunteluun: näppäimiä kuunneltiin ainoastaan silloin, kun kyseinen editori-ikkuna oli valittuna. Unityn delegate-toiminnallisuudella metodikutsu on kuitenkin mahdollista delegoida Unityn main loopin yhteyteen, eli metodi voidaan kutsua samalla, kun Unity käy läpi sisäisiä metodikutsujaan, jos Unityn luokka vain tarjoaa mahdollisuuden metodikutsun tilaamiseen. Unityn sisäisten kutsujen suoritusjärjestyksestä johtuen

senhetkinen Event-olio, joka sisältää käyttäjäpalautteen, on kuitenkin instantioitu vain, jos sitä käsitellään piirto-operaatioihin liittyvien kutsujen yhteydessä. Unityn Scene-näkymässä mahdollisuuden metodikutsun tilaamiselle tarjoaa SceneView-luokan OnSceneGUI-metodi, jolloin delegaatille annettua metodia kutsutaan aina OnSceneGUI-metodikutsun yhteydessä. Koska editoriluokasta ei ole välttämättä aina olemassa instanssia silloin, kun pikanäppäimiä halutaan tutkia, irrotetaan kutsuttava metodi luokan instanssista tekemällä siitä staattinen. Staattiselle metodille tilataan callback aina muokatun editorinäkymän target-luokan luonnin yhteydessä ja metodi irrotetaan delegaatista aina target-luokan instanssin poiston yhteydessä. Kun todetaan, että käyttäjän määrittämää pikanäppäintä on painettu, instantioidaan näppäintä vastaava peliobjekti hiiren senhetkiseen sijaintiin. Näin pikanäppäimiä voidaan kuunnella jatkuvasti Scene-näkymässä, mutta Play-näkymässä pikanäppäimet eivät tällä toteutuksella toimi, sillä OnSceneGUI-metodia kutsutaan vain Scene-näkymässä. Play-näkymässä toimivia pikanäppäimiä ei vielä toteutettu tähän työkaluun, mutta helpointa olisi todennäköisesti vain luoda oma pikanäppäimien kuunteluskripti Play-näkymälle. Koska Play-näkymässä tehdyt muutokset eivät tallennu Scene-näkymään, ja täten varsinaiseen pelimaailmaan, voisi Play-näkymässä lisätyt objektit vain tallentaa taulukkoon, josta ne Play-näkymästä poistuttaessa lisätään Scene-näkymään, jolloin muutokset saadaan säilymään.

7 Kenttien luonti

Työkalun alkuperäisenä käyttötarkoituksena oli kentän osien luominen, joita yhdistelemällä peliin voitaisiin rakentaa kokonaisia tasoja. Miten työkalulla luoduista kenttäpaloista kentät sitten lopulta luotaisiin? Ensimmäinen mieleen tullut ajatus, joka vaikutti sopivan yksinkertaiselta tasojen luonnin automatisointia varten, oli se, että kenttäpalat yhdistettäisiin toisiinsa yksinkertaisesti palojen avoimien sivujen perusteella. Kaksi avointa palan sivua yhdistyy suoraan toisiinsa palojen ollessa vakiokokoisia, jolloin palat sopisivat aina saumattomasti yhteen. Jokaisessa pelaajan reitillä olevassa kenttäpalassa olisi siis vähintään yksi avoin sivu. Vaihtoehtoisesti kaikki sivut voisivat olla avoimia, jolloin pala on osa avointa aluetta kentässä (kuva 26).

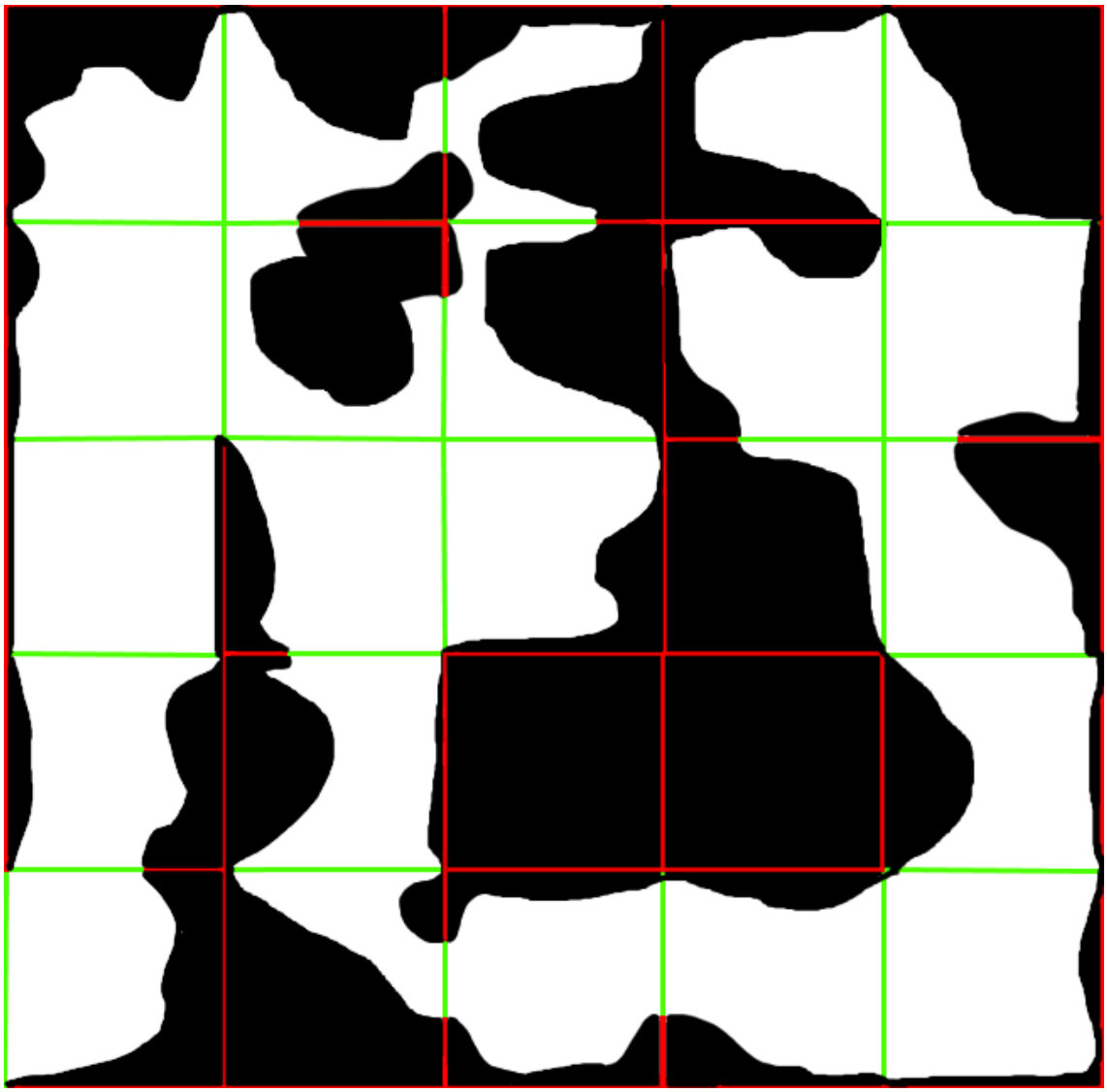


Kuva 26. Karkeasti piirretty illustraatio siitä, millaisia tasoja ja muotoja avoimien sivujen perusteella yhdistettävistä paloista voisi saada aikaan.

Kuvaa 26 katsoessa lähestymistavassa on kuitenkin välittömästi nähtävissä omat ongelmansa. Kahden rinnakkain liitetyn suljetun sivun liitoskulmista voi tulla todella kapeita, jos samasta kulmasta lähtee avoin sivu, tai varsinkin jos kulmasta lähtee kaksi avointa sivua. Samoin palojen muotoihin tulee tietty rajoittava säännöllisyys, jos avoimen sivun tulee olla auki reunasta reunaan, jolloin muotoihin tulee aina kapea kohta palojen saumakohtiin.

Esitettyjä ongelmia on kuitenkin mahdollista korjata palaideaa laajentamalla. Jos yksittäinen sivu jaettaisiin esimerkiksi kolmeen osaan, saataisiin yhdelle sivulle 8 erilaista yhdistämistapaa, jolloin paloja voitaisiin liittää selvästi monipuolisemmin toisiinsa. Palojen muotoilun kanssa olisi myös selvästi vähemmän rajoitteita, kun palojen

sisältämä muoto voisi loppua kokonaisen sivun reunan sijasta myös mihin tahansa sivujen välipisteistä (kuva 27).



Kuva 27. Karkeasti piirretty illustraatio siitä, millaisia tasoja ja muotoja voitaisiin muodostaa, jos yhdistettävät sivut jaettaisiin kolmeen osaan.

Vaikka kuvaa 26 ja kuvaa 27 vertailtaessa on selvää, että jälkimmäisen tarjoama lopputulos on parempi, lisääntyy myös tarvittavan työn määrä merkittävästi kuvassa 27 esitetyllä palojen liittämistekniikalla. Jos alun perin esitetyllä kokonaisten sivujen liittämistekniikalla saadaan 2^4 = 16 erilaista palatyyppiä, niitä olisi palan jokaisen sivun kolmeen osaan jakamisen jälkeen 2^12 = 4096. Kentän muotojen monipuolisuuden ylläpitämiseksi tarvittavien erilaisten palavarianttien määrä muodostuisikin nopeasti niin suureksi, että ajatuksen toimivuutta pitää vähintäänkin harkita hyvin tarkasti.

Palojen liittämiseen käytettävä tekniikka, ja ylipäätään se, miten paloja tullaan hyödyntämään, on projektissa vielä harkinnan alaisena. On mahdollista, että kenttien luonnin automatisoinnin ideasta luovutaan täysin, jolloin palojen yhdistely ja kenttien kokoaminen, tehtäisiin kokonaan käsin. Tällöin myöskään palojen koolla tai yhdistymispisteillä ei olisi niinkään merkitystä, koska palat voitaisiin palakohtaisesti sovittaa toisiinsa. Jos paloja ei saataisi yhdistettyä näin järkevästi, voitaisiin kenttätyökalulla maalata palojen välille sopiva siirtymäpala. On myös täysin mahdollista, että jos pala-ajattelua ei vain saada toimimaan kenttäsuunnittelussa, kentät koostettaisiin normaaliin tapaan suoraan yhtenä palana.

8 Kehityskohteet ja kriittinen tarkastelu

Koska toteutuksen reaaliaikaisuudesta jouduttiin lopulta luopumaan, on hieman kyseenalaista, tarvitseeko myöskään optimoimatonta meshiä luoda ollenkaan reaaliajassa. Nykyisessä toteutuksessa luodusta marching squares -meshistä ei kuitenkaan käytetä kuin polygonin reunapisteet, joten meshin jatkuva reaaliaikainen generoiminen voitaisiin ennen tason viimeistelyvaihetta korvata bittikartalla, josta etsitään lopullisessa meshissä käytettävät kulmapisteet. Tällä lähestymistavalla piirtokäyttöliittymästä saataisiin todennäköisesti suorituskyvyn kannalta selvästi kevyempi. Tässä tapauksessa pisteiden loppukäsittely kuitenkin vaatisi todennäköisesti vuorostaan enemmän työtä, koska interpolointi pitäisi siirtää loppukäsittelyvaiheeseen ja kulmikkaiden reunojen pyöristämiseen pitäisi todennäköisesti käyttää jonkinlaista Bézier-käyriä hyödyntävää interpolaatiototeutusta. Lineaarinen interpolaatio toimisi myös tällaisessa toteutuksessa, jos taustalle lisäisi samanlaisen datarakenteen kuin nykyisessä toteutuksessa. Interpoloinnin tulos vain näkyisi käyttäjälle vasta viimeistelyvaiheen jälkeen, eikä reaaliajassa kuten tämänhetkisessä toteutuksessa. Tämänhetkinen meshin jatkuva generointi puolustaakin paikkaansa siinä mielessä, että käyttäjä näkee lopputuloksen jo piirtovaiheessa, ja toteutus on kuitenkin samalla tarpeeksi suorituskykyinen datamäärille, joilla sitä käytetään.

Luvussa 3 mainittu rajoite siitä, että käyttäjän asettaman ympyrän keskipiste sijaitsee aina jossain kokonaislukukoordinaatissa rajoittaa kuvassa 12 näkyvällä tavalla vapaalla kädellä vedetyn meshin reunan pehmeyttä, jolloin tarpeeksi läheltä tarkasteltaessa sen reunaan syntyy selvä porrastus. Olisikin edullista reunan pehmeyden kannalta, jos käyttäjä voisi sijoittaa ympyrän liukulukukoordinaattiin, mutta ongelmaksi muodostuu

marching squares -algoritmi: jos ympyrän keskipiste ei sijoitu kokonaislukukoordinaattiin, niin ympyrän huippukohta sijoittuu sivulle, jonka molemmilla puolilla on epäaktiivinen kulmapiste. Tällöin välipiste ei ole myöskään aktiivinen eli ympyrän huipuksi muodostuu suora sivu edellisen ja seuraavan välipisteen välillä, toisin sanoen ympyrän huipun muoto menetetään. Ongelma on vaikea ratkaista, käytännössä marching squaresia pitäisi jollain tavalla laajentaa. Toisaalta on myös hieman kyseenalaista, muodostuuko tästä edes visuaalista ongelmaa, koska mittakaava on melko pieni. Voi olla, ettei ympyrän ääriarvojen häviäminen olisi edes ongelma. Liukulukukeskipisteitä pitäneekin siis kokeilla toteutuksessa jossain vaiheessa.

Alkuperäisestä tavoitteesta, että tasoa voitaisiin luoda reaaliaikaisesti, luovuttiin raskaan pisteiden käsittelyn vuoksi. Mutta olisiko tasoa ollut kuitenkin mahdollista luoda reaaliajassa? Jatkuvan tasonluonnin toteuttaminen vaatisi ensinnäkin sitä, että kolmiointi tulisi reaaliaikaisesti kohdistaa vain jokaisen painalluksen vaikutusalueella olleeseen alueeseen, koko polygonia ei voitaisi millään kolmioida jokaisen painalluksen jälkeen kokonaan uudestaan. Jokaisella painalluksella pitäisi myös tehdä kaikki polygoneja koskevat tarkistukset ennen kolmiointia: yhdistettiinkö vanhoja polygoneja, lisättiinkö kokonaan uusi polygoni, jos polygoneja yhdistettiin, niin mitkä polygonit yhdistettiin. Poisto-operaation yhteydessä: poistettiinko polygoneja tai jakautuiko yksi polygoni painalluksella useampaan polygoniin? Tämän jälkeen pisteet tulisi ennen kolmiointia vielä järjestellä myötäpäiväiseen järjestykseen ennen kolmiointialgoritmille antamista. On suoraan sanottuna vaikea nähdä, että tällaista toteutusta saisi millään toimimaan reaaliajassa. Tason rakennus reaaliajassa voisi onnistua, mutta toteutuksen pitäisi mahdollistaa huomattavasti kevyempi pisteiden käsittely. Mahdollisesti käyttäjä lisäisi kentästä ainoastaan kulmapisteet ja rakentaisi vain yhtä polygonia kerrallaan.

Unityyn on Unityn Asset Storesta saatavilla erilaisia maksullisia kenttäeditoreja, joilla voitaisiin päästä vastaavaan lopputulokseen. Syy, miksi tätä toteutusta kuitenkin lähdettiin työstämään, on se, että kaikki näistä editoreista oli toteutettu teknisesti jokseenkin samalla tavalla: Bézier-käyriä hyödyntävä interpolaatio, jossa kenttä muodostetaan interpolaation päätepisteistä ja niiden välille muodostuvalla kaarella sijaitsevista pisteistä. Työhön haluttiin ottaa kuitenkin lähtökohdaksi piirtotyökalumainen toteutus, mutta jälkeenpäin voi miettiä, oliko tämä edes järkevä ajatus. Esimerkiksi toteutus, joka hyödyntää kolmannen asteen Bézier-käyriä, vaikuttaa ajatuksena elegantilta toteutukselta, jossa polygonin reunapisteiden kontrollointi olisi huomattavasti helpompaa. Toteutuksessa määritettäisiin piste, joka jää polygonin reunalle, ja tälle kaksi

kontrollipistettä. Kahden reunapisteen määrittämisen jälkeen muodostetaan Bézier-käyrä ensimmäisen reunapisteen ja toisen reunapisteen välille, käyttäen kontrollipisteinä ensimmäisen pisteen jälkimmäistä kontrollipistettä ja jälkimmäisen pisteen ensimmäistä kontrollipistettä. Tätä työtä voisikin siis ajatella jokseenkin epäonnistuneena konseptikokeiluna. Henkilökohtaisesti näkisin, että kuvan 27 kaltaisia muotoja olisi vaivattomampaa muodostaa reunapisteitä lisäten ja Bézier-käyriä hyväksikäyttäen kuin pyrkiä piirtämään niitä piirtotyökalumaisesti.

9 Yhteenveto

Työn päällimmäisenä tavoitteena oli kehittää työkalu Unity-pelimoottorille pelikenttien rakentamista varten. Tässä tavoitteessa onnistuttiin, ja saatiin aikaan toimiva toteutus, jolla kenttiä on mahdollista tehdä peliin. Alkuperäisenä ajatuksena ollutta kentän generointia paloista ei kuitenkaan työn puitteissa ehditty käymään läpi kuin ajatuksen tasolla.

Työkalun alkuperäiseen toimintakuvaukseen olennaisena osana kuuluneesta pelitason reaaliaikaisesta rakennuksesta jouduttiin luopumaan, mutta kenttiä oli silti mahdollista rakentaa työkalulla tarpeeksi nopeasti. Toteutusperiaatteet todettiin toimiviksi, työkaluun saatiin toimiva kolmioinnin optimointi, ja työkalun suorittama pisteiden käsittely saatiin toimimaan luotettavasti.

Työ opetti paljon Unityn editorin sisäisestä toiminnasta ja sen muokkaamisesta, sekä kolmiointialgoritmeista ja meshien käsittelystä yleisesti. Koska työkalulla luotujen kenttämeshien laatua pidettiin projektin tarkoituksiin hyvänä, on työssä aikaansaatua työkalua tarkoitus käyttää pelin kenttien luomiseen myös tulevaisuudessa

Lähteet

1 Wikipedia. Unity (game engine). Verkkoaineisto. <https://en.wikipedia.org/wiki/Unity_(game_engine)>. Luettu 13.11.2017.

2 Fine, Richard 2017. UnityScript's long ride off into the sunset. Verkkoaineisto. <https://blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/>. 14.8.2017. Luettu 13.11.2017.

3 Ferr2D Terrain Tool. Verkkoaineisto. <https://www.assetstore.unity3d.com/en/#!/content/11653>. Luettu 13.11.2017.

4 2D Terrain Editor. Verkkoaineisto. <https://www.assetstore.unity3d.com/en/#!/content/15761>. Luettu 13.11.2017.

5 Wikipedia. Marching Squares. Verkkoaineisto. <https://en.wikipedia.org/wiki/Marching_squares>. Luettu 14.11.2017.

6 Wong, Jamie 2014. Metaballs and Marching Squares. Verkkoaineisto. <http://jamie-wong.com/2014/08/19/metaballs-and-marching-squares/>. 19.8.2014. Luettu 14.11.2017.

7 Unity Manual. Optimizing graphics performance. Verkkoaineisto. <https://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html>. Luettu 15.11.2017.

8 Berry, Harry 2013. How to simplify a marching squares mesh? Verkkoaineisto. <https://stackoverflow.com/questions/17896447/how-to-simplify-a-marching-squares-mesh>. 29.7.2013. Luettu 15.11.2017.

9 Schertler, Nico 2014. Mesh Simplification: Edge Collapse Conditions. Verkkoaineisto. <https://stackoverflow.com/questions/27049163/mesh-simplification-edge-collapse-conditions>. 7.9.2015. Luettu 15.11.2017.

10 Wikipedia. Polygon triangulation. Verkkoaineisto. <https://en.wikipedia.org/wiki/Polygon_triangulation>. Luettu 25.11.2017.

11 Wikipedia. Two ears theorem. Verkkoaineisto. <https://en.wikipedia.org/wiki/Two_ears_theorem>. Luettu 25.11.2017.

12 Wikipedia. Delaunay triangulation. Verkkoaineisto. <https://en.wikipedia.org/wiki/Delaunay_triangulation>. Luettu 28.10.2017.

13 Su & Drysdale 1996. A Comparison of Sequential Delaunay Triangulation Algorithms. Verkkoaineisto.

<http://www.cs.berkeley.edu/~jrs/meshpapers/SuDrysdale.pdf>. 1.4.1996. Luettu 28.10.2017.

14 Shewchuk, Jonathan Richard 1996. Triangulation Algorithms and Data Structures. Verkkoaineisto. <http://www.cs.cmu.edu/~quake/tripaper/triangle2.html>. 12.8.1996. Luettu 28.10.2017.

15 Peterson, Samuel. COMPUTING CONSTRAINED DELAUNAY TRIANGULATIONS. Verkkoaineisto. <http://www.geom.uiuc.edu/~samuelp/del_project.html>. Luettu 28.10.2017.

16 ConstraintMesher.cs. Verkkoaineisto. <https://triangle.codeplex.com/SourceControl/latest#Triangle.NET/Triangle/Meshing/ConstraintMesher.cs>. Luettu 28.10.2017.

17 Woltering, Christian 2013. Triangle.NET. Verkkodokumentti. <https://triangle.codeplex.com/>. 18.6.2013. Luettu 28.10.2017.

Työkalun suorittaman suorien etsinnän tarkempi kuvaus

Mahdollisia suoria piirtoalueen rajalla voidaan tutkia siten, että kun löydetään kulmapiste, se merkitään mahdollisesti muodostuvan sivun toiseksi päätepisteeksi. Tämän jälkeen voidaan edetä reunaa, jolla valittu piste sijaitsee, niin kauan, kunnes reunalta löydetään ensimmäinen epäaktiivinen reunapiste. Kun epäaktiivinen piste on löydetty, voidaan valita sitä edellinen aktiivinen piste muodostuvan sivun päätepisteeksi, jos pisteen, josta tutkiminen aloitettiin, jälkeen löydettiin ainakin kaksi aktiivista pistettä. Tämän jälkeen muodostetun sivun molemmat päätepisteet merkitään pistepariksi, jonka jälkeen kaikki samalla sivulla sijaitsevat tarpeettomat pisteet voidaan poistaa pistejoukosta päätepisteiden väliltä. Sama periaate pätee myös pisteisiin, jotka eivät sijaitse piirtoalueen reunalla. Jos kahden vierekkäisen marching squares -neliön konfiguraatio (ks. kuva 5) on sama, on mahdollista, että ne muodostavat suoran sivun. Tämä voidaan tarkistaa tutkimalla neliöiden välipisteiden sijainteja. Jos on mahdollista, että konfiguraatio muodostaa suoran sivun vaaka- tai vaakadiagonaalisuunnassa (tapaus 4 tai tapaus 13), tutkitaan, onko vierekkäisten neliöiden välipisteiden muodostaman sivun kulma suhteessa positiivisen x-akselin suuntaiseen sivuun sama. Jos kulma on sama, eli vertailtujen neliöiden välipisteet sijaitsevat samalla suoralla, voidaan pistejoukosta poistaa välipiste, joka on ensimmäisen neliön oikean- ja toisen neliön vasemmanpuoleisen sivun välipiste, ja muodostaa pistepari ensimmäisen neliön vasemman- ja oikeanpuoleisen neliön oikeanpuoleisen sivun välipisteiden välille. Jos taas on mahdollista, että konfiguraatio muodostaa suoran sivun pysty- tai pystydiagonaalisuunnassa (tapaus 7 tai tapaus 10), tehdään vastaava kulmien vertailu suhteessa positiivisen y-akselin suuntaiseen sivuun. Neliöiden konfiguraatioita vertaillaan tällä tavoin, kunnes löydetään neliö, jonka konfiguraatio ei ole sama, jolloin lopullinen suoran muodostava pistepari merkitään x-akselin tapauksessa ensimmäisen vertailuun valitun neliön vasemmanpuoleisen sivun ja viimeisen vertaillun neliön oikeanpuoleisen sivun välipisteiden välille, ja y-akselin tapauksessa ensimmäisen vertailuun valitun neliön alimman sivun ja viimeisen vertaillun neliön ylimmän sivun välipisteiden välille. Koska pisteparit tulee myös sijoittaa myötäpäiväiseen järjestykseen, jotta niiden käyttäminen onnistuisi pisteitä myötäpäiväiseen järjestykseen asetettaessa, tutkitaan ennen pisteparien asetusta kummasta pisteestä kumpaan pisteeseen sivua tulisi kulkea, jotta polygoni käydään läpi myötäpäiväisesti. Tästä riippuen pistepareja asetettaessa aloituspisteeksi voidaan asettaa joko piste, josta tutkiminen aloitettiin, tai

piste joka tutkimalla löydettiin. Loppupiste asetetaan vastaavalla tavalla muodostuvan sivun läpikäyntisuunnan mukaisesti.