Mattias Männistö

Ruutupohjainen sodan sumu 3D-strategiapelissä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Pelisovellukset

Insinöörityö

11.5.2024

Tiivistelmä

Tekijä:	Mattias Männistö
Otsikko:	Ruutupohjainen sodan sumu 3D-strategiapelissä
Sivumäärä:	43 sivua
Aika:	11.5.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaajat:	Lehtori Heini Puuska Lehtori Miikka Mäki-Uuro

Insinöörityössä tutkittiin taistelukentällä vallitsevan epävarmuuden, eli sodan sumun, merkitystä, mekaniikkoja ja visualisointia strategiapeleissä. Työssä toteutettiin Unityllä kolmiulotteiseen pelimaailmaan ruutupohjainen sodan sumu, jossa keskityttiin erityisesti suorituskykyyn ja estetiikkaan. Toteutuksesta pyrittiin tekemään myös helposti käyttöönotettava ja jatkotyöstettävä.

Toteutuksen lopputuloksena oli toimiva sodan sumu reaaliaikaiseen tai vuoropohjaiseen ylhäältä päin kuvattuun kolmiulotteiseen strategiapeliin. Toteutuksen sodan sumu toimii reaaliaikaiselle strategiapelille tyypillisellä tavalla, jossa sumulla on kolme eri tilaa. Sumu piirretään pelimaailman pintaan itse tehdyllä varjostimella. Toteutus saatiin suorituskyvyltään kohtalaiseksi, ja se toimii hyvänä pohjana sodan sumulle, mutta suurilla oliomäärillä ja pelimaailman ruudukon tarkkuudella toteutus ei toimi riittävän hyvin peleissä, jotka on tarkoitettu julkaistavaksi.

Visuaaliselta tyyliltään toteutuksesta tehtiin 2000-luvun alun 3D-strategiapelien kaltainen. Sumuun lisättiin myös mahdollisuus muuttaa visuaalista tyyliä tekstuureilla.

Avainsanat: Sodan sumu, Unity, varjostimet

Tämän opinnäytetyön alkuperä on tarkastettu Turnitin Originality Check -ohjelmalla.

Abstract

Author:	Mattias Männistö
Title:	Tile-based fog of war in a 3D strategy game
Number of Pages:	43 pages
Date:	11 May 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Game Applications
Supervisors:	Heini Puuska, Senior Lecturer
	Miikka Mäki-Uuro, Senior Lecturer

The purpose of the final year project was to study the mechanics and implementations of fog of war in strategy games. A goal was to implement a performant and aesthetic tile-based fog of war in Unity. The aim was also to make the project expandable and easily importable. The inspiration for the project was 3D strategy games from the early 2000s.

The result was a functional fog of war for a real-time or turn-based three-dimensional top-down strategy game. The project's fog of war, much like a typical real-time strategy game's fog of war, functions in three states, which are calculated in the game's logic. Then, the fog is drawn onto the game world using a custom shader. The result turned out moderately performant with a low number of units and a low grid resolution, but for games with a higher unit count the implementation needs more optimizations.

The targeted visual style of an early 2000s strategy game was achieved. The fog's visual style was also made modifiable with custom textures.

Keywords: Fog of war, Unity, shaders

Sisällys

Lyhenteet

1 Johdanto 1

2 Sodan sumu videopeleissä 1

2.1 Täydellinen ja epätäydellinen informaatio 1
2.2 Sodan sumun määritelmä 3
2.3 Sodan sumun tarve peleissä 4
2.4 Alueisiin perustuva sodan sumu 7
2.5 Alueisiin perustumaton sodan sumu 10
2.6 Visualisointitapoja 11

3 Eri tapoja toteuttaa sodan sumu 16

3.1 Ruutupohjainen etäisyyslaskenta 17
3.2 Sumukamera 17

4 Ruutupohjainen sodan sumu Unity-pelimoottorilla 19

4.1 Oliohierarkia 19
4.2 Näkyvyyden laskenta 23
4.3 Näkyvyyden datatekstuurin piirtäminen 28
4.4 Visualisointi pelimaailmaan 30
4.5 Toteutuksen yleiskäyttöisyys 39

5 Yhteenveto 40

Lähteet 44

Lyhenteet

2D	*2-dimensional*. Kaksiulotteinen. Pelimaailma toimii vain vaaka- ja pystysuunnassa.
3D	*3-dimensional*. Kolmiulotteinen. Pelimaailmassa on vaaka- ja pystysuuntien lisäksi syvyyssuunta.
4X-peli	*Explore, expand, exploit, exterminate*. Strategiapelien alaluokka, jossa tutkitaan tuntematonta ympäristöä, pyritään laajentamaan pelaajan hallitsemaa aluetta ja käyttämään alueen resursseja eduksi sekä tuhoamaan vastustajat.
CPU	*Central processing unit*. Tietokoneen yleiseen laskentaan tehty suoritin.
FoW:	*Fog of war*. Sodan sumu. Termi, jolla kuvataan sodassa vallitsevaa tiedon puutetta. Visualisoidaan videopeleissä yleensä piilottamalla pelimaailman alueita.
GPU	*Graphics processing unit*. Grafiikan laskentaan erikoistunut suoritin tietokoneissa.
LoL	*League of Legends*. Eräs taisteluareenamoninpelityylinen videopeli.
LoS:	*Line of sight*. Yksittäisen pelihahmon näkemä alue pelimaailmassa.
RBG	*Red, green, blue*. Punainen, vihreä ja sininen väri. Tietokoneen näytöllä yksittäisen pikselin eri värit saadaan aikaan yhdistelemällä pikselin RGB-värien voimakkuuksia.
RTS:	*Real-time strategy.* Reaaliaikainen strategiapeli -peligenre.

1 Johdanto

Insinöörityössä toteutetaan Unity-pelimoottorilla ruutupohjainen visualisointi taistelukentän epävarmuuden tuottamasta informaation puutteesta, eli sodan sumusta, 3D-pelimaailmaan. Unityllä tehtävä sodan sumun pelillinen toteutus tehdään reaaliaikaisen strategiapelin näkökulmasta, mutta toteutus soveltuu hyvin myös vuoropohjaiseen strategiapeliin. Toteutuksesta pyritään tekemään mahdollisimman yleiskäyttöinen, jotta sitä voisi käyttää jatkossa muissa projekteissa. Toteutusta ei rakenneta minkään olemassa olevan pelin päälle, vaan se on oma irrallinen kokonaisuutensa: toteutuksen Unity-projektiin kuuluu ainoastaan sodan sumun toteutus ja työkalut sen testaamiseen. Tavoitteena on tuottaa muutaman tiedoston kokonaisuus, jonka voisi viedä halutessaan mihin tahansa Unity-projektiin, jossa halutaan toteuttaa ruutupohjainen sodan sumu.

Jotta yleiskäyttöisyys toteutuisi, toteutuksen tulee olla mahdollisimman suorituskykyinen suurillakin oliomäärillä ja ohjelmointirajapinnan mahdollisimman yksinkertainen. Sumun visualisoinnista pyritään tekemään sellainen, että ruutupohjaisuus erottuisi mahdollisimman vähän. Visualisoinnin estetiikalla on toteutuksessa kuitenkin suorituskykyä ja käytettävyyttä pienempi prioriteetti. Visuaalisen tyylin esikuvana käytetään 2000-luvun alun kolmiulotteisten strategiapelien yleistä sodan sumun tyyliä, jossa tutkimattomat alueet piirretään mustana, tutkitut, mutta piilossa olevat alueet piirretään varjoisina ja näkyvät alueet piirretään valoisina.

2 Sodan sumu videopeleissä

2.1 Täydellinen ja epätäydellinen informaatio

Strategiapelien keskeisenä elementtinä on tehdä päätöksiä pelistä saatavilla olevan informaation perusteella. Pelit voidaan jakaa niissä esillä olevan informaation perusteella täydellisen informaation ja epätäydellisen informaation peleihin (1).

Täydellisen informaation peleissä kaikki pelin informaatio on jatkuvasti saatavilla kaikille pelaajille. Mitään informaatiota ei piiloteta muilta pelaajilta, eikä pelissä myöskään esiinny satunnaisuuteen perustuvia elementtejä, kuten noppia (1). Shakki on esimerkki täydellisen informaation pelistä. Shakissa pelilaudan tila on näkyvissä jatkuvasti molemmille pelaajille, kuten kuvassa 1 näkyy, ja molemmat pelaajat tietävät kaikki pelissä tapahtuneet aikaisemmat siirrot. Vain peräkkäisten siirtojen vuoropohjaiset pelit voivat olla täydellisen informaation pelejä, sillä jos pelaajien siirrot tapahtuvat samanaikaisesti, pelaajat eivät tiedä muiden kaikkia siirtoja (2).

Kuva 1. Shakkilaudan tila on jatkuvasti molemmille pelaajille näkyvissä (3).

Epätäydellisen informaation peleissä osa pelin informaatiosta on piilotettu. Piilotettu elementti voi olla esimerkiksi tietoa pelaajan omasta asetelmasta, jota muut pelin pelaajat eivät tiedä, tai pelin yhteistä informaatiota, jota kukaan pelaaja ei tiedä. Esimerkkinä on pokerin Texas hold ’em -pelimuoto, jossa pakassa olevat kääntämättömät kortit ovat kaikilta pelaajilta piilotettua informaatiota. Pöydälle nostetut ja käännetyt kortit ovat kaikille pelaajille yhteistä informaatiota, mutta pelaajien omat kortit, joita muut pelaajat eivät näe, ovat

piilotettua informaatiota. Kuvassa 2 on Texas hold ’em -pelin tilanne, jossa pöydällä on kolme kaikille näkyvää korttia, ja pelaajalla on kaksi vain hänelle itselleen näkyvää korttia.

Kuva 2. Texas hold ’em -pelissä osa korteista näkyy kaikille ja osa vain pelaajalle (4).

2.2 Sodan sumun määritelmä

Sodan sumu on termi, jolla kuvataan sodankäynnissä esiintyvää epävarmuutta vallitsevasta tilasta. Sotien tapahtumien kulku on tyypillisesti kaoottista ja ennalta-arvaamatonta. Etenkin tiedustelutietojen vajaavaisuus tuottaa epätäydellistä informaatiota vastustajan toimista, mutta myös esimerkiksi säätila, joukkojen kykeneväisyys toteuttaa tehtävänsä ja mekaaniset häiriöt tuottavat epävarmuustekijöitä sotasuunnitelmiin. (5.)

Laivanupotus on tunnettu epätäydellisen informaation lautapeli, joka simuloi sodan sumua. Laivanupotuksessa koko pelikenttä vastustajan puolelta on piilotettu. Pelikentästä saa tietoa kokeilemalla ampumalla satunnaisiin sijainteihin, jolloin vastustaja kertoo, osuiko ammus hänen laivaansa. Kuvassa 3 on muovinen versio laivanupotuksesta, jossa pystyseinämä estää näkemästä vastustajan puolelle. Seiniin kiinnitetään merkkejä niihin sijainteihin, joihin on itse ammuttu,

ja alas merkitään vastustajan ampumiset. Merkkien värit kertovat, osuivatko ammukset johonkin. Pystyseinän toisella puolella vastustaja merkkaa vastaavan informaation omasta suoriutumisesta omalle puolelleen.

Kuva 3. Laivanupotuslautapeli (6).

Sodan sumu tarkoittaa kaikkea informaatiota, joka sotatilanteessa jää pimentoon, mutta peleissä sodan sumulla viitataan pääsääntöisesti alueisiin perustuvaan informaation piilottamiseen, jossa osa pelimaailmasta jätetään piirtämättä pelaajalle.

2.3 Sodan sumun tarve peleissä

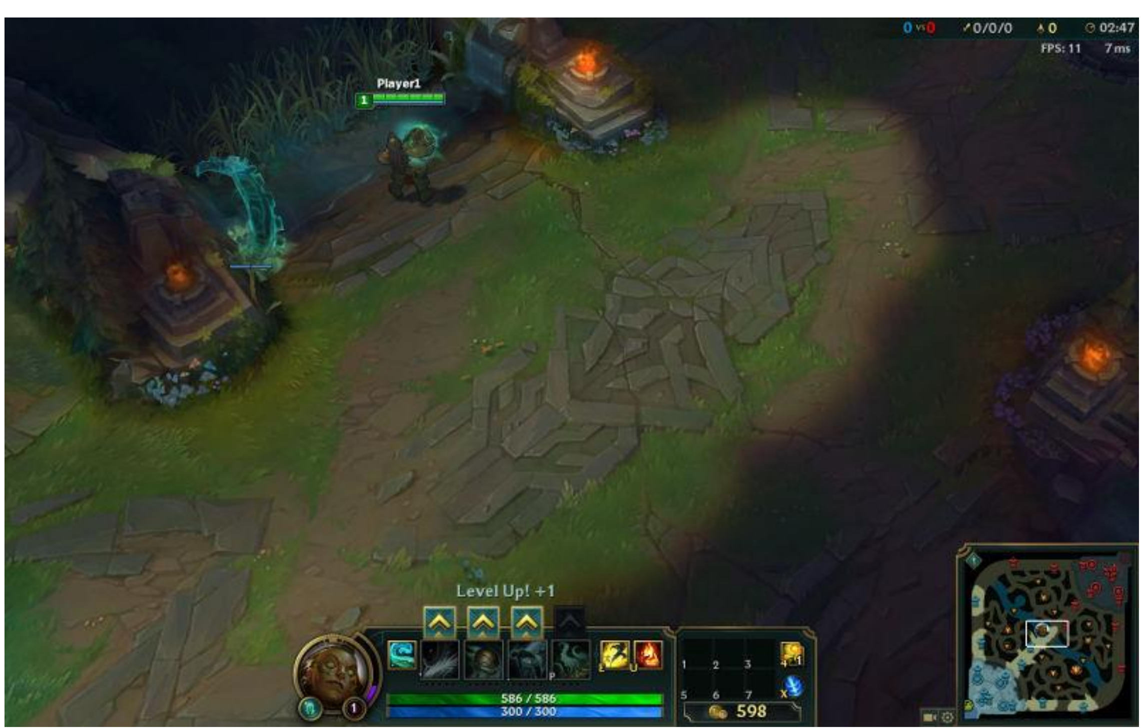
Sodan sumua esiintyy vastaavasti myös videopeleissä informaation puutteen takia. Ensimmäisen ja kolmannen persoonan taistelupeleissä informaation puutetta aiheuttaa erityisesti kameran sijainti. Näissä peleissä kamera on joko pelaajan silmien kohdalla tai pelaajan vieressä pelaajan välittömässä läheisyydessä. Pelaaja ei siis yleensä saa näköinformaatiota muualta pelimaailmasta kuin siitä alueesta, mihin hän milläkin hetkellä katsoo hahmollaan.

Ylhäältä päin kuvatuissa peleissä kamera näkee sen sijaan yleensä laajemman alueen kuin mitä pelaajan hahmo näkisi silmillään. Monesti kameraa voi myös liikuttaa täysin vapaasti, jolloin pelaajalla on mahdollisuus nähdä kaikki pelin tapahtumat. Näissä peleissä pelaajalle saatavilla olevan informaation määrää

rajoitetaan keinotekoisesti piilottamalla pelaajalta asioita, joita hän todellisuudessa voisi pelin kameran avulla nähdä. Tämä informaatio piilotetaan siten, että pelaajalle ei piirretä joitain pelimaailman alueita tai olioita.

Monissa ylhäältä päin kuvatuissa strategiapeleissä, joissa kilpaillaan muita pelaajia vastaan, on tyypillisesti tarvetta piilottaa alueita keinotekoisesti sodan sumuun. Näissä peleissä usein soditaan, ja on haluttu, että pelaajat eivät saa pelin tilanteesta kaikkea informaatiota, kuten oikean maailman sotatilanteissa. 4X-genren strategiapeleissä tutkitaan maailmaa, kerätään resursseja, laajennetaan valtakuntaa ja pyritään tuhoamaan vastustajat. 4X-genren peleissä on tyypillistä, että pelimaailma on aluksi tuntematon pelaajille, mutta sitä tutkitaan pelin edetessä. Pelialuetta tutkimalla pelaajalle selviää muun muassa missä muut pelaajat ovat, missä tärkeitä resursseja on ja miten pääsee paikasta A paikkaan B. (7.) Pelin tiedoista osa saattaa jäädä pimentoon pelaajalle, mikäli hän ei tutki maailmaa tarpeeksi aktiivisesti. Esimerkiksi vastustajien uhkataso on usein tärkeää tietoa: kuinka vahva ja minkä tyyppinen armeija vastustajalla on, ja aikooko vastustaja hyökätä.

Sodan sumua on muissakin kuin strategiapeleissä. Taisteluareenamoninpeli League of Legends (LoL) sisältää pelimaailman alueisiin perustuvan sodan sumun. LoL on kuvattu ylhäältä päin ja kameraa voi liikuttaa vapaasti. Ilman sodan sumua pelaaja siis näkisi koko pelikentän halutessaan, mutta sodan sumulla tämä on estetty: pelaaja voi nähdä vain oman ja joukkueensa hahmojen läheisyydessä olevat asiat, kuten kuva 4 osoittaa. Muu pelikenttä on sumun peitossa. (8.)



Kuva 4. League of Legends -pelin sodan sumua (8).

Sodan sumu parantaa myös pelin uudelleenpelattavuutta. Useat ylhäältä päin kuvatut strategiapelit ovat proseduraalisesti satunnaisgeneroituja, ja sodan sumu säilyttää usein pitkäänkin epävarmuutta siitä, mitä maailmassa on ja tapahtuu. Pelikerrat ovat todennäköisemmin erilaisia keskenään, sillä aina ei ole varmuutta siitä, mitä aluetta olisi järkevintä tukia ensin. Maailman tutkiminen tuo myös osaltaan kehityksen tunnetta: pelaaja aloittaa pelin tietämättä ympäröivästä maailmasta mitään, mutta tutkittuaan maailmaa pelaaja löytää asioita, joista muut eivät välttämättä ole tietoisia, ja hän voi tehdä kyseisen informaation perusteella itseään hyödyttäviä päätöksiä. Pelimaailman alueiden tutkimista voidaan pitää eräänlaisena resurssina strategiapeleissä, ja monesti maailman tutkimista myös pisteytetään (9).

Ensimmäisen persoonan Minecraft-pelissä sodan sumua esiintyy pelin karttaesineissä. Pelin maailmaa ei itsessään ole piilotettu lainkaan sodan sumuun, vaan pelaaja pystyy piirtoetäisyydestä riippuen näkemään hyvinkin kauas horisonttiin. Sumua esiintyy piirtoetäisyyden rajalla vain pelin laskentanopeuteen ja esteettisyyteen liittyvistä syistä. Pelissä pystyy luomaan kuitenkin maailman lähialueista kartan, joka on aluksi tyhjä. Sitä mukaa, kun pelaaja tutkii maailman

alueita kartta mukanaan, karttaan piirtyy automaattisesti tietoa ympäristöstä, kuten kuva 5 osoittaa.

Kuva 5. Minecraft-pelin kartta. Taustalla näkyvä sumu ei ole sodan sumua, vaan keino välttää äkillinen maailman katkeaminen piirtoetäisyyden rajalla. (10).

2.4 Alueisiin perustuva sodan sumu

Sodan sumulla piilotetaan pelimaailman alueista tietoa. Sumun visualisointiin kuuluu yleensä kolme eri tilaa, jossa alueyksikkö, eli ruutupohjaisissa peleissä ruutu, voi olla: piilotettu, tutkittu ja näkyvä. Kuvassa 6 näkyy selkeästi kaikki kolme sodan sumun tilaa tyypillisessä kaksiulotteisessa strategiapelissä. Kuvassa 6 on käytetty englanninkielisiä termejä *Unkown Area* [*sic*] (piilotettu), *Foggy Area* (sumuinen eli tutkittu) ja *Fully Visible Area* (näkyvä).

Kuva 6. Sodan sumun kolme tilaa pelissä Age of Empires II (11).

Piilotetulta alueelta ei anneta mitään informaatiota pelaajalle alueen ominaisuuksista tai tapahtumista. Se on perinteisesti kuvattu mustana alueena, mutta pelien kehittyessä on tullut myös muita visualisointitapoja. Esimerkiksi Civilization V -pelissä piilotettu alue on paksujen pilvien peitossa (12), ja Civilization VI -pelissä piilotettu alue on karttapaperia, johon ei ole piirretty vielä mitään (13).

Tutkittu alue on sellainen alue pelimaailmassa, jossa pelaaja on joskus käynyt jollain pelihahmollaan, tai muulla tavoin saanut kyseiselle alueelle hetkellisen näkyvyyden, mutta hänellä ei ole kyseiseen paikkaan tällä hetkellä näkyvyyttä. Alueen tiedot tavallaan muistetaan. Tutkituilta alueilta pelaajalle näytetään usein staattiseksi ajateltuja asioita, kuten luonnonmuodostelmia ja muiden pelaajien rakennelmia, mikäli ne ovat olleet paikalla silloin, kun alue viimeksi tutkittiin. Tutkitun tilan alueista ei yleensä saada reaaliaikaista tietoa, vaan alueet näkyvät muuttumattomina pelaajalle, vaikka alueilla tapahtuisi jotain. Tutkitun tilan alueiden tiedot päivittyvät vasta, kun pelaaja seuraavan kerran tutkii alueen uudestaan. Joitain dynaamisia, kaikille pelaajille yhteisiä asioita saatetaan näyttää reaaliajassa myös tutkituilla alueilla, kuten esimerkiksi luonnon eläinten liike. Tutkitut alueet esitetään visuaalisesti yleensä heikon näkyvyyden alueena, kuten tummana varjona tai sumuna.

Näkyvältä alueelta pelaaja saa lähes kaiken informaation reaaliaikaisena. Näkyvään alueeseen kuuluu sellaiset pelimaailman alueet, joihin pelaajan hahmoilla on näköyhteys. Näkyvä tila visualisoidaan yleensä alueena, joka on kunnolla valaistu. Näkyvän alueen informaatio ei ole välttämättä silti täydellistä. Esimerkiksi Civilization-sarjan peleissä pelaaja saattaa nähdä vastustajan kaupungin ja sotilaat, jotka puolustavat kaupunkia, mutta pelaaja ei silti tiedä, mitä kaupunki tuottaa tällä hetkellä.

Mikäli pelin maailma on sama jokaisella pelikerralla, ei välttämättä ole mielekästä piilottaa sitä täysin edes tutkimattomilta alueilta. Tällaisissa peleissä sodan sumulla on yleensä vain kaksi tilaa: tutkittu ja näkyvä, ja maailma on alusta alkaen siis vähintään tutkittu-tilassa, jossa pelimaailman maasto on näkyvissä kaikille sumun läpi.

Total War -pelisarjassa (14) pelaaja pystyy liikuttamaan ja kääntämään kameraa hyvinkin vapaasti. Sodan sumu näiden pelien taistelupelimuodossa on toteutettu siten, että jos pelaajan sotilailla ei ole suoraa näköyhteyttä vastustajan sotilasyksikköön, kyseistä sotilasyksikköä ei piirretä maailmaan lainkaan pelaajalle, joka ei sitä näe. Myös metsät ja korkea ruoho kätkevät niiden sisällä olevat sotilaat piiloon. Piilossa olevat sotilaat havaitaan vasta, kun omat sotilaat liikkuvat tarpeeksi lähelle piilossa olevia vastustajan joukkoja. Normaalitilanteessa, jossa näköesteitä ei ole, näkyvyys on useita satoja metrejä, mutta metsässä piilossa olevat hahmot tulevat vastustajapelaajan näkyviin vasta muutamien kymmenien metrien etäisyydellä. Myös vuoret ja kalliot estävät näkyvyyden: vaikka pelaaja siirtäisi kameran katsomaan vastustajan sotilasyksikköä kallion takana, sotilasyksikkö ei piirry pelaajan näkökenttään, jos siihen ei ole kellään pelaajan sotilaalla suoraa näköyhteyttä. Kyseinen piilossa oleva sotilasyksikkö pystyy silti vaikuttamaan normaalilla tavalla pelin kulkuun, esimerkiksi antamaan epäsuoraa tulta, kuten ampumaan jousipyssyillä nuolisateen kaaressa kallion yli.

Door Kickers on ylhäältä päin kuvattu kaksiulotteinen toimintapeli, jossa liikutaan rakennusten sisällä. Pelissä alueet, joihin pelaajan hahmot eivät näe, ovat sodan sumun piilottamia. (15.) Kuva 7 osoittaa, miten sodan sumu on suoraan pelaajan hahmojen näköyhteyteen perustuva: pelaaja ei huomaa lähelläkään

olevia vastustajia mikäli jokin este, kuten seinä, peittää näköyhteyden vastustajan sijaintiin.

Kuva 7. Sodan sumua Door Kickers -pelissä. Alueet, joihin pelaajan hahmot eivät näe, kuvataan sinertävinä. (15.)

2.5 Alueisiin perustumaton sodan sumu

Strategiapeleissä jokin informaatio saattaa olla muilta pelaajilta jatkuvasti piilossa. Informaatiota voidaan piilottaa myös muunkaltaisen kuin alueisiin perustuvan sodan sumun taakse. Esimerkiksi Civilization IV -pelissä on vakoilumekaniikka, *Espionage*, jolla on toteutettu sodan sumu muuhun kuin näköyhteyteen perustuvana. Espionage kuvastaa pelaajan valtakunnan tuottamaa vakoiluinformaatiota vastustajista. Suurella määrällä Espionage-pisteitä pelaaja näkee esimerkiksi vastustajan kaupungit myös siellä, mihin omilla pelihahmoilla ei normaalisti olisi näköyhteyttä. Espionage antaa myös tietoa kaupunkien tämänhetkisistä tuotannoista, mihin pelkkä näköyhteys ilman Espionage-pisteitä ei yksin riitä. (16.)

Kaikki strategiapelien pelihahmot ja yksiköt eivät aina ole saman arvoisia sodan sumun suhteen. Joillakin yksiköillä saattaa olla ominaisuus, että ne eivät näy vastustajille pelkän näköyhteyden avulla, vaan tarvitaan jokin muu lisätekijä.

Esimerkiksi Civilization-pelisarjan peleissä sukellusveneet ovat peliyksiköitä, jotka eivät näy vastustajille, vaikka vastustaja näkisikin sen ruudun, jossa sukellusvene on, vaan vastustajalla pitää olla lähistöllä jokin tietyntyyppinen sukellusveneitä näkevä yksikkö, kuten hävittäjä (17).

Ensimmäisen persoonan ammuntapeleissä voi esiintyä sodan sumua myös esimerkiksi näkymättömyyden, näkyvyyttä peittävien elementtien tai valeinformaation avulla. Team Fortress 2 -pelissä on hahmo nimeltä Spy, joka pystyy hetkellisesti muuttumaan näkymättömäksi tai naamioitumaan vastustajan pelaajaksi (18). Useissa ensimmäisen persoonan ammuntapeleissä on myös savukranaatteja, joilla voi hetkellisesti peittää näkyvyyden kaikilta pelaajilta tiettyyn pelimaailman alueeseen. Kuvassa 8 näkyy Counter-Strike 2 -pelin savukranaatin aiheuttamaa savua.

Kuva 8. Counter-Strike 2 -pelin savukranaatin savua (19).

2.6 Visualisointitapoja

Alueisiin perustuvalle sodan sumulle on lukuisia eri visualisointitapoja. Perinteisin tapa on piirtää piilotetut alueet pelimaailmasta täysin mustana, tutkittu alue tummennettuna ja näkyvä alue normaalisti. Vanhemmissa peleissä sodan sumu ei ollut niinkään sumumaista tai volymetristä, vaan pikemminkin pelimaailman

pinnan värien muokkaamista tai maailman piirtämisen rajaamista maskin avulla. Sodan sumu on piirretty suoraan pelimaailman pintaan eikä sen ylle, kuten kuvassa 9 näkyy.

Kuva 9. Age of Empires II -pelin kaksiulotteista sodan sumua (20).

Pintaan piirretty sodan sumu on tyypillistä myös vanhemmissa 3D-peleissä, jossa volymetrisen sumun vaikutelmaa on yritetty tuottaa sumentamalla sumun eri tilojen välisiä rajoja. Kuvan 10 pelissä sodan sumun eri tilat sulautuvat toisiinsa esteettisemmin pienen liukuvan sumennuksen ansiosta.

Kuva 10. Civilization IV -pelin sodan sumun kolme tilaa. Oikealla näkyy muutama metsäruutu tumman varjon alueella, joka kuvastaa tutkittua, mutta ei tällä hetkellä näkyvää aluetta. (21.)

Kuvan 11 Age of Mythology -pelissä on haluttu myös vähentää sodan sumun ruutumaisuutta, vaikka pelin sodan sumu onkin ruutupohjaisesti toteutettu. Pelkän sodan sumun perusteella on usein vaikea sanoa, missä kohtaa menee ruudun raja. Pelihahmon liikkuessa ruutupohjaisuuden huomaa silti edelleen, sillä uutta aluetta tulee sodan sumusta näkyviin portaittain.

Kuva 11. Age of Mythology -pelin sodan sumua (22).

Myöhemmin sumusta on alettu tekemään enemmän sumumaista ja kolmiulotteisempaa. Sumu ei ole enää tehoste pelimaailman pinnassa, vaan jotain pinnan yläpuolella olevaa pinnan peittävää asiaa, kuten pilviä. Kuvassa 12 näkyy, miten Civilization V -pelissä sodan sumun piilottamat alueet ovat pilvien peitossa. Pilvet piirtyvät selvästi muun pelimaailman ylle, millä on haettu luonnollisempaa ja realistisempaa vaikutelmaa. Tällainen sodan sumun toteutus on hieman haastavampaa kuin suoraan pelimaailman pintaan piirretty sumu, koska korkealla olevien pilvien saatetaan tulkita peittävän eri ruutua, kuin mikä on oikeasti piilossa pelaajalta. Civilization V -pelissä asia on ratkaistu laittamalla pilvet juuri niihin sijainteihin, jotka halutaan peittää, mutta pilvet langettavat varjot muualle, millä saadaan aikaan vaikutelma, että pilvet ovat selvästi maanpinnan yläpuolella.

Kuva 12. Civilization V -pelin sodan sumun kolme tilaa. Pilvet tarkoittavat piilotettua aluetta, jota pelaaja ei ole vielä tutkinut. (12.)

Civilization VI -pelissä on palattu takaisin pintatason sodan sumuun taiteellisen tyylin vuoksi. Kuvassa 13 näkyy, miten pelissä on haluttu sodan sumun kuvastavan piirtämätöntä karttaa, johon piirretään asioita sitä mukaa, kun pelaaja tutkii maailmaa. Tässä tapauksessa on hyvin luontevaa, että sodan sumu on samassa tasossa kuin pelimaailma. Huomioitavaa on, että tyhjä karttapaperi ei paljasta pelimaailman pinnanmuodoista mitään. Maastonmuodot näytetään pelaajille vain tutkituilta ja näkyviltä alueilta.

Kuva 13. Civilization VI -pelin sodan sumun kolme tilaa. Tutkittu alue on kartalle piirretyn maaston näköistä ja piilotettu alue on tyhjää karttapaperia. (13.)

3 Eri tapoja toteuttaa sodan sumu

Pelimaailman alueisiin perustuva sodan sumu voidaan toteuttaa monella tapaa. Perinteisesti se on toteutettu ruutupohjaisena, sillä strategiapelien logiikka on tyypillisesti toiminut muutenkin jonkinlaisessa ruudukossa, joten on ollut luontevaa sitoa sodan sumu samaan ominaisuuteen. Ruuduilla ei välttämättä tarkoiteta neliöitä, vaan nykyisin yhä useammin peliruudukko koostuu heksagoniruuduista. Tässä tutkielmassa keskitytään neliöruutupohjaiseen sodan sumuun sen yksinkertaisemman toteutuksen takia, mutta samat strategiset mekanismit pätevät molempiin tapoihin toteuttaa sodan sumu.

Kaikki sodan sumun toteutukset perustuvat jollain tapaa menetelmään, jossa pelimaailmaan tai sen päälle piirretään asioita peittävä sumu, johon luodaan aukkoja niihin paikkoihin, joihin pelaajan halutaan pystyvän näkemään. Eri tekniikat poikkeavat toisistaan lähinnä siinä, miten pelaajien näkyvyysinformaatio saadaan laskettua.

3.1 Ruutupohjainen etäisyyslaskenta

Ruutuihin perustuvassa sodan sumun laskennassa ideana on se, että koko pelimaailma on jaettu ruutuihin. Pelaajien omistamien pelimaailman hahmojen oliot laskevat halutuin aikavälein kaikki ruudut, joihin kyseinen hahmo näkee. Pelaajaolio päivittää sen tiedon perusteella taulukkoa, jossa on tieto siitä, mitkä kaikki pelimaailman ruudut ovat edes jonkin pelaajan hahmon näkökentässä. Pelaajaolio pitää myös yllä tietoa siitä, mitkä kaikki ruudut ovat joskus olleet näkyvissä. Nämä informaatiot piirretään tekstuuriin, joka syötetään pelimaailman maaston varjostimelle, joka piilottaa informaation perusteella maastosta alueita.

Ruutupohjaisen toteutuksen hyötynä on sumuinformaation nopea saatavuus koodissa. Mikäli sumuinformaatiota käytetään muissakin pelin mekaniikoissa kuin pelkästään maailman näkyvyyden visualisoinnissa, ruutupohjaisesta toteutuksesta on hyötyä.

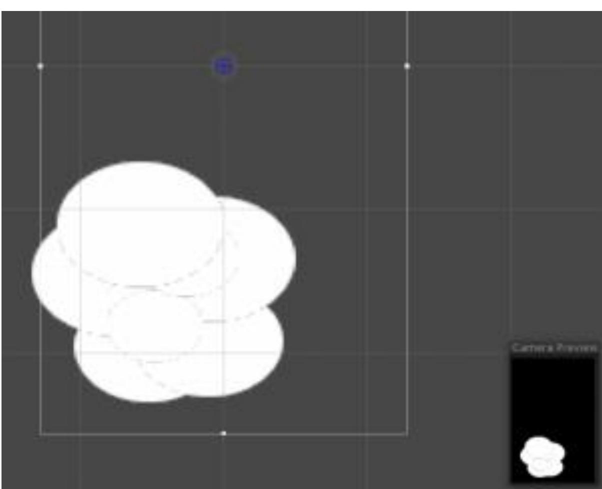
Toteutuksen haittana on se, että mitä suurempi määrä pelimaailmassa on ruutuja, sitä useampia laskutoimituksia pelaajaolio joutuu tekemään laskiessaan näkyvyysinformaatiota. Myös pelimaailman hahmojen määrä vaikuttaa suoraan laskutoimitusten määrään. Pelin kuvataajuus alkaa hyvin nopeasti heikentyä hahmomäärän ja ruudukon koon kasvaessa.

Insinöörityön toteutus tehtiin käyttäen ruutupohjaista tapaa, ja tämän toteuttaminen Unityllä käydään tarkemmin läpi luvussa 4.

3.2 Sumukamera

Ruudukon sijaan pelaajien näkyvyysinformaation luominen voidaan tehdä Unityssä käyttämällä erillistä sodan sumun näkyvyysinformaatiolle omistettua kameraa, sumukameraa. Pelimaailman hahmoille tehdään lapsiolioksi niin sanottu sumumaski, joka on hahmon näkökentän kokoinen kappale, jonka vain sumukamera pystyy näkemään. Esimerkiksi ympärilleen 15 metrin etäisyydelle näkevän hahmon sodan sumun sumumaski olisi lieriö tai pallo (2D-pelimaailmassa ympyrä), jonka säde on 15 m. Sumumaski asetetaan Unityssä omalle tasolleen

(Unityn Layer), ja sumukamera konfiguroidaan näkemään vain maailman sumumaskit. Sumukameran näkemä maailma piirretään RenderTexture-kuvatiedostoon, joka näin ollen sisältää käytännössä saman näkyvyysinformaation kuin ruutupohjaisesti lasketusta sodan sumusta piirretty tekstuuri. Tutkittu-alue saadaan aikaan tekemällä toinen vastaavanlainen kamera, jossa erona on Unityn kameran ”Don’t clear” -asetus, joka ei pyyhi RenderTextureen piirtämäänsä aiempaa kuvaa, vaan uusi kuva piirretään vanhan päälle. Tällöin tutkitut alueet jäävät kuvaan muistiin. (23.) Kuva 14 demonstroi ellipsin muotoisten sumumaskien piirtämistä RenderTextureen sumukameran näkemänä.



Kuva 14. Ellipsin muotoisia sumumaskeja sumukameran läpi kuvattuna. Oikealla alhaalla sumukameran RenderTextureen piirtämä kuva. (23.)

Etuna RenderTexturen ja sumukameran käytössä on se, että niillä saa ruutupohjaista toteutusta helpommin tarkemman sodan sumun informaation, kuin mitä ruutupohjaisessa laskennassa on mahdollista saada samalla suorituskyvyllä. Toteutus ei myöskään ole riippuvainen ruutupohjaisuudesta, mikä tekee siitä esteettisemmän. Haittana sumukamerassa on sodan sumun informaation vaikeampi saatavuus koodissa, mikäli sille on tarve. Jos halutaan selvittää, onko jokin sijainti sumun alueella vai ei, on käytettävä esimerkiksi säteensuuntausta sumukamerasta maailman tiettyyn sijaintiin ja tarkistettava, onko jonkin olion sumumaski säteen tiellä, tai luettava RenderTexturen pikseliarvoja koodissa, mikä on hidas operaatio (24).

4 Ruutupohjainen sodan sumu Unity-pelimoottorilla

Tässä työssä toteutettiin alueisiin perustuva sodan sumu Unity-pelimoottorilla. Tarkoituksena oli toteuttaa yleinen sodan sumu, jonka voisi liittää mihin tahansa Unityllä tehtävään kolmiulotteiseen reaaliaikaiseen strategiapeliin. Toteutuksen esikuvana käytettiin Age of Empires -pelien tyylistä kolmessa tilassa toimivaa sodan sumua. Toteutus tehtiin ruutupohjaisesti olettaen, että pelimaailma on neliö.

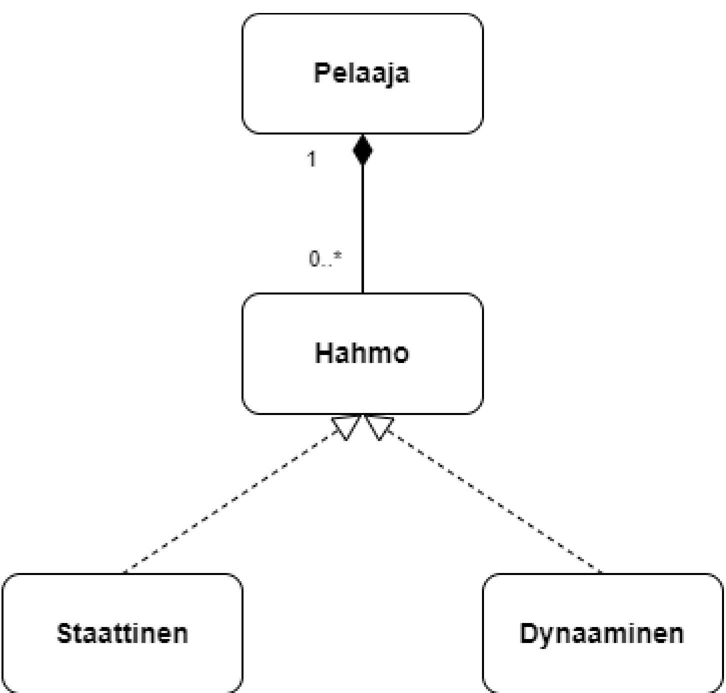
4.1 Oliohierarkia

Toteutuksessa käytettiin tyypillistä strategiapelin oliohierarkiaa. Näissä peleissä on pelaajia, toteutuksen koodissa ”Player”, joilla on hahmoja, ”Unit”. Pelaajat pystyvät näkemään ainoastaan ne asiat pelimaailmassa, jotka ovat omien hahmojen lähettyvillä. Hahmot näkevät ympärilleen joka suuntaan tietyn matkan päähän. Pelaajilla on mahdollisuus muodostaa joukkueita, ja joukkueilla on tyypillisesti yhteinen näkyvyys: pelaajat näkevät kaiken, minkä oman joukkueen muut pelaajat pystyvät näkemään.

Joissain peleissä pelaajilla on staattisia hahmoja, rakennelmia, jotka eivät liiku, mutta niillä on silti kyky nähdä ympäristöään samalla tavalla kuin pelaajan liikkuvilla hahmoilla. Ainoa ero näiden staattisten ja dynaamisten hahmojen välillä on se, että vastustajien staattiset hahmot näkyvät myös tutkituilla alueilla, mikäli hahmo on ollut olemassa silloin, kun alue viimeksi tutkittiin. Staattisista hahmoista säilyy viimeisin tieto tutkitulla alueella. Jos vastustajan staattinen hahmo tuhoutuu, mutta pelaaja ei tutki aluetta uudelleen, hahmo säilyy näkyvissä tutkitulla alueella, kunnes alue tutkitaan uudelleen.

Toteutuksessa Player- ja Unit-oliot pyrittiin pitämään mahdollisimman yksinkertaisina, jotta pelit, joihin toteutuksen sodan sumu mahdollisesti liitettäisiin, eivät vaatisi ominaisuuksia, joita kyseinen peli ei tosiasiassa tarvitsisi ilman sodan sumua. Näin ollen toteutuksen Player- ja Unit -oliosta tehtiin hyvin yleistettyjä ja sellaisia, että niillä on tyypillisen reaaliaikaisen strategiapelin vastaavien olioiden perusrakenteet. Oliot sisältävät vain toiminnallisuuden kommunikoida

sodan sumun kanssa sekä yksinkertaiset liikkumisrajapinnat, jotta sodan sumua pystyi testaamaan. Mikäli toteutuksen sodan sumua käytettäisiin jossain ulkoisessa Unity-projektissa, toteutuksen Player- ja Unit-olioita ei otettaisi mukaan, vaan ulkoisen projektin täytyisi sisältää vastaavat oliot omalla tavallaan toteutettuina. Toteutukseen tehtiin myös pelinhallitsijaolio, ”GameManager”, joka pitää kirjaa niin sanotusti aktiivisesta pelaajaoliosta, eli siitä pelaajasta, jonka näkyvyys halutaan visualisoida pelimaailmaan. Niin kuin Player- ja Unit-oliot, GameManager ei myöskään liity sodan sumun toteutukseen, vaan ainoastaan sen testaamiseen, joten näiden olioiden toiminnallisuuksista kerrotaan tässä raportissa vain tarvittava. Kuva 15 esittää tyypillisen yksinkertaisen strategiapelin oliohierarkian.



Kuva 15. Tyypillisen reaaliaikaisen strategiapelin oliohierarkia. Pelaaja omistaa hahmoja, jotka voivat olla dynaamisia tai staattisia.

Itse sodan sumun laskennasta vastaa sodan sumu -olio, ”FogOfWar”. FogOfWar-olio sisältää tiedon pelimaailman ruudukosta ja vastaa sodan sumun informaatiosta kootun tekstuurin piirtämisestä. Kyseinen tekstuuri syötetään varjostimelle piilottamaan maailman alueita.

Pelimaailman ruudukko, jonka perusteella sodan sumu lasketaan, noudattaa omaa oliohierarkiaansa. Ruudukko-olio, ”MapGrid”, sisältää leveys * pituus -määrän ruutuja, ”Tile”. Kun oletetaan, että pelimaailma on aina neliö, ruudukon koko voitaisiin ilmoittaa yhdellä kokonaisluvulla, joka ilmaisisi neliöruudukon sivun pituuden, mutta laajennettavuuden vuoksi toteutuksessa käytettiin Vector2Int-tyyppistä arvoa ruudukon koolle. Vector2Int-tyyppisessä muuttujassa pystyy määrittelemään tarpeen tullen eri arvot ruudukon leveydelle ja pituudelle. Esimerkkikoodissa 1 näkyy MapGrid-olion konstruktori, jossa luodaan pelimaailman ruudukko.

```
public class MapGrid
{
    public Vector2Int size = new Vector2Int(100, 100);
    public Tile[,] tiles;

    public MapGrid()
    {
        tiles = new Tile[size.x, size.y];

        for (int y = 0; y < size.y; y++)
        {
            for (int x = 0; x < size.x; x++)
            {
                tiles[x, y] = new Tile(x, y);
            }
        }
    }

    ~MapGrid() {}
}
```

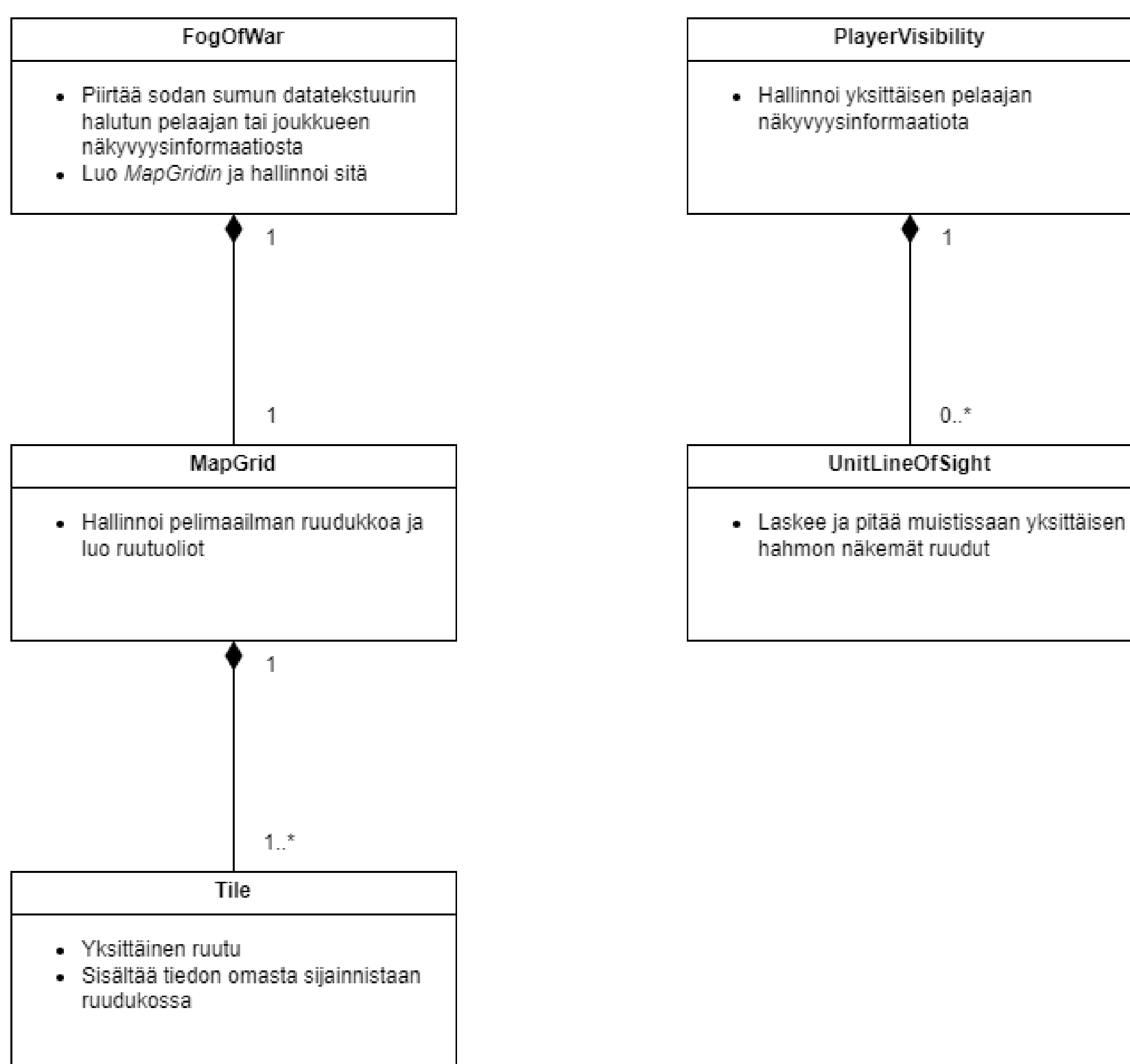
Esimerkkikoodi 1. Ruudukko-olio kokonaisuudessaan.

Ruudukon koko on oletusarvoisesti 100x100, mutta sitä voi tarpeen tullen muuttaa isommaksi tai pienemmäksi, kunhan se säilyy neliönä. Esimerkkikoodissa 2 näkyy ruutuolion yksinkertainen rakenne. Tile-olio sisältää vain x- ja y-koordinaatit, eli tiedon siitä, missä kyseinen ruutu sijaitsee ruudukossa.

```
public class Tile
{
    public Vector2Int coordinates;

    public Tile(int x, int y)
    {
        coordinates = new Vector2Int(x, y);
    }

    ~Tile() {}
}
```

Esimerkkikoodi 2. Ruutuolio sisältää vain tiedon omasta sijainnistaan ruudukossa.

Kuva 16 näyttää sodan sumun rajapinnan oliot ja niiden tärkeimmät tehtävät. FogOfWar-olio luo MapGridin, joka luo halutun määrän Tile-olioita. PlayerVisibility-olio pitää yllä yhden pelaajan näkyvyysinformaatiota, ja UnitLineOfSight laskee tarpeen tullen hahmon näkemät ruudut ja ylläpitää sitä tietoa. Kuvassa 16 näkyy myös näiden olioiden väliset riippuvuudet ja suhteelliset määrät.



Kuva 16. Sodan sumun olioiden hierarkia ja vastuualueet.

4.2 Näkyvyyden laskenta

Pelaajaoliot pitävät yllä tietoa siitä, mitkä pelimaailman ruudut pelaajan hahmot näkevät ja mitkä ruudut on joskus nähty. Jokaisella pelaajaoliolla on henkilökohtainen taulukko, johon on tallennettu jokaisen ruudun tämänhetkinen näkyvyystila pelaajalle. Pelin alkaessa kaikki pelaajan taulukon ruudut ovat aluksi piilotettu-tilassa, koodissa arvolla ”HIDDEN”. Kun pelaajan hahmo näkee jonkun ruudun, ruutu merkitään näkyväksi, ”VISIBLE”, pelaajan näkyvyystaulukkoon. Esimerkkikoodi 3 näyttää, miten jokaisessa pelin päivityskehyksessä kaikki edellisellä kehyksellä näkyväksi merkatut ruudut merkitään tutkittu-tilaan, ”EXPLORED”. Tämän jälkeen jokainen pelin pelaajaolio kysyy jokaiselta

omistamaltaan hahmolta, mitkä pelimaailman ruudut kyseinen hahmo näkee. Niiden perusteella pelaajaolio päivittää oman näkyvyystaulukkonsa.

```
enum TileVisibilities
{
    VISIBLE,
    EXPLORED,
    HIDDEN
}

void Update()
{
    // Update previously visible tiles as explored
    playerVisibility.ConvertVisibleToExplored();

    // Update currently visible tiles as visible
    foreach (Unit unit in units)
    {
        foreach(Tile tile in unit.GetVisibleTiles())
        {
            playerVisibility.tileVisibilities[tile.coordinates.x,
tile.coordinates.y] = TileVisibilities.VISIBLE;
        }
    }
}

void ConvertVisibleToExplored()
{
    for (int y = 0; y < gridSize.y; y++)
    {
        for (int x = 0; x < gridSize.x; x++)
        {
            if (tileVisibilities[x, y] == TileVisibilities.VISIBLE)
            {
                tileVisibilities[x, y] = TileVisibilities.EXPLORED;
            }
        }
    }
}
```

Esimerkkikoodi 3. Pelaajan näkyvyystaulukon päivittäminen. GridSize-muuttuja on pelaajaolion välimuistiin tallennettu tieto pelimaailman ruudukon koosta.

Jokainen pelimaailman hahmo pitää itse tallessa tietoa niistä ruuduista, jotka kyseinen hahmo näkee. Aina, kun hahmon sijainti pelimaailman horisontaalisessa xz-tasossa muuttuu siten, että hahmo siirtyy eri ruutuun, hahmo laskee uudelleen ruudut, jotka se näkee. Pelimaailma voi hyvinkin olla kumpuileva, mutta tämän insinöörityön sodan sumun toteutuksessa sumun laskentaan vaikuttaa ainoastaan hahmon horisontaalinen sijainti, eli Unityssä hahmon x- ja z-koordinaatit. On tärkeää, että hahmo laskee näkyvyysdatansa uudelleen vain silloin, kun se on tarpeellista, eikä esimerkiksi joka kehyksessä. Näin kaikkien

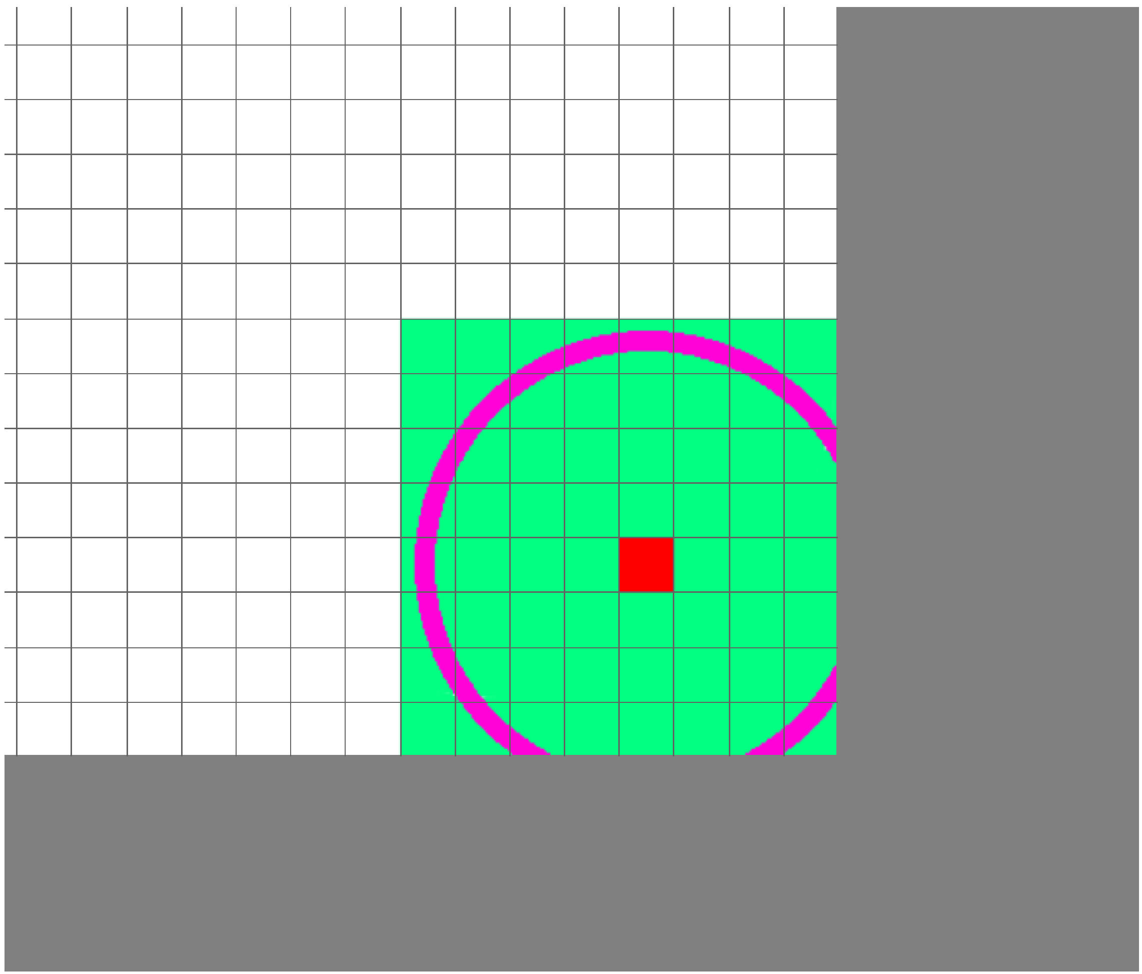
hahmojen yhteinen laskentakuorma jakaantuu tasaisemmin eri kehyksille, sillä on hyvin epätodennäköistä, että kaikki hahmot vaihtaisivat ruutuaan samalla kehyksellä. Laskennan alussa hahmon näkemien ruutujen lista nollataan, jotta se ei sisällä vanhentunutta näkyvyysinformaatiota. Sen jälkeen lasketaan, mitkä ruudut ovat hahmon näköetäisyyden – englanniksi line of sight, koodissa ”LoS” – sisällä. Ruutujen näkyvyyteen ei tässä toteutuksessa vaikuta mikään muu tekijä kuin etäisyys hahmosta ruutuun. Esteet ja muut suoran näkölinjan estävät tekijät eivät vähennä näkyvyyttä, ja esimerkiksi kukkulan päällä oleminen ei lisää näkyvyyttä alamäkeen. Esimerkkikoodissa 4 lasketaan yksinkertaistetulla tavalla hahmon näkemät ruudut. Hahmo käy läpi kaikki pelimaailman ruudut, ja laskee oman etäisyytensä niihin.

```
List<Tile> visibleTiles = new List<Tile>();

void CalculateVisibleTiles()
{
    // Clear previously visible tiles from list
    visibleTiles.Clear();

    // Unit’s current position in the world’s xz-plane
    Vector2 ownCoordinates = Get2DPosition();

    // Calculate the Unit’s distance to each Tile
    for (int y = 0; y < mapGrid.tiles.GetLength(1); y++)
    {
        for (int x = 0; x < mapGrid.tiles.GetLength(0); x++)
        {
            Tile tile = mapGrid.tiles[x, y];

            // Add Tile to visible tiles list if it’s within LoS range
            if (Vector2.Distance(ownCoordinates, tile.coordinates) <=
LOSRange)
            {
                visibleTiles.Add(tile);
            }
        }
    }
}
```

Esimerkkikoodi 4. Hahmon näkemien ruutujen laskenta ilman suorituskyvyn optimointia.

Suorituskyvyn kannalta on olennaista, että hahmo ei tarkasta turhaan pelimaailman jokaista ruutua, vaan ainoastaan ne ruudut, jotka ovat vaaka- tai pystysuunnassa enimmillään hahmon LoS:n etäisyydellä, kuten kuvassa 17 havainnollistetaan. Tämän etäisyyden sisällä olevilta ruuduilta tarkastetaan vielä tarkka etäisyys, ja mikäli se on pienempi tai yhtä suuri kuin hahmon LoS, ruutu

merkitään hahmolle näkyväksi listaan. Lista sisältää lopulta kaikki ruudut, jotka hahmo näkee. Loput ruudut oletetaan hahmolle näkymättömiksi.



Kuva 17. Havainnollistava kuva laskennan optimoinnista, jossa on merkattu punaisella hahmon tämänhetkinen ruutu, liilalla hahmon LoS, ja vihreällä ne ruudut, jotka ovat maksimissaan LoS:n etäisyydellä hahmosta. Valkoisella merkattuja pelimaailman ruutuja ei tarvitse käsitellä. Harmaa alue on pelimaailman ulkopuolista aluetta.

Esimerkkikoodissa 5 ruutujen näkyvyyslaskenta toteutetaan edellistä esimerkkiä optimoidummin laskien vain ne ruudut, joilla on vähintäänkin jokin mahdollisuus olla hahmon näkökentän sisällä. Tässä tapauksessa pitää olla tarkkana, että tarkastellaan vain pelimaailman sisältämiä ruutuja, eikä vahingossa yritetä käsitellä ruutuja pelimaailman ruudukon ulkopuolelta.

```
public void CalculateVisibleTiles()
{
    // Clear previously visible tiles from list
    visibleTiles.Clear();

    // Calculate the square of LoSRange
    float sqrLOS = LOSRange * LOSRange;

    // Calculate the Unit’s distance to each Tile within reasonable
distance
    for (
        int y = Mathf.Max(0, Mathf.FloorToInt(currentTile.y -
LOSRange));
        y < Mathf.Min(mapSize.y, Mathf.CeilToInt(currentTile.y + 1 +
LOSRange));
        y++
    )
    {
        for (
            int x = Mathf.Max(0, Mathf.FloorToInt(currentTile.x -
LOSRange));
            x < Mathf.Min(mapSize.x, Mathf.CeilToInt(currentTile.x + 1
+ LOSRange));
            x++
        )
        {
            // Add Tile to visible tiles list if it’s within LoS range
            Tile tile = mapGrid.tiles[x, y];
            if ((tile.coordinates - currentTile).sqrMagnitude <= sqr-
LOS)
            {
                visibleTiles.Add(tile);
            }
        }
    }
}
```

Esimerkkikoodi 5. Hahmon näkemien ruutujen laskenta optimoidulla tavalla, jossa lasketaan vain ne ruudut, jotka ovat maksimissaan hahmon LoS:n etäisyydellä vaaka- tai pystysuunnassa ruudukossa.

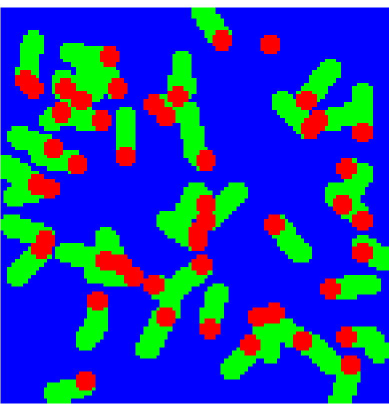
Esimerkkikoodissa 5 etäisyyksiä on vertailtu käyttäen vektorin sqrMagnitude-attribuuttia sen sijaan, että olisi käytetty helpommin luettavaa Vector2.Distance-metodia. Unityn ohjesivuston mukaan tällä on positiivinen vaikutus suorituskykyyn (25), mutta erään käytännön testin mukaan tällä optimoinnilla ei olisi vaikutusta (26). Vector2.Distance-metodi palauttaa kahden sijainnin välisen tarkan etäisyyden, mutta tässä tapauksessa tarkan etäisyyden laskeminen ei ole tärkeää, vaan se, onko kahden sijainnin välinen etäisyys pienempi kuin LoS. Etäisyyden ja LoS:n vertailuun riittää vektorin sqrMagnitude-attribuutin käyttäminen, joka palauttaa vektorin pituuden neliön. Koska sekä tarkasteltavien sijaintien välisen vektorin pituus että LoS ovat positiivisia reaalilukuja, vektorin pituuden

neliön vertaaminen LoS:n neliöön kertoo, onko sijaintien välinen etäisyys pienempi kuin LoS (27).

4.3 Näkyvyyden datatekstuurin piirtäminen

Kun pelaajaolio on koonnut kaikilta hahmoiltaan tiedon niiden näkemistä ruuduista, voidaan kyseinen data syöttää tekstuuriin, jonka perusteella lopulta maailmasta piilotetaan asioita. Mikäli on tarvetta usean pelaajan yhteiselle näkyvyysdatalle, kuten joukkueissa, näkyvyysdata kootaan kaikilta joukkueen pelaajilta. Paremman näkyvyyden datalla on aina suurempi prioriteetti kuin huonomman näkyvyyden datalla, eli jos joukkueessa yksikin pelaaja näkee tietyn ruudun, joukkueen yhteisellä näkyvyydellä kaikki joukkueen pelaajat näkevät kyseisen ruudun. Kaikki pelin pelaajaoliot päivittävät joka kehyksellä oman näkyvyysdatansa ja pitävät sen muistissaan, mutta tekstuuriin piirretään ainoastaan se data, joka halutaan visuaalisesti näyttää pelimaailmassa. Toteutuksessa GameManager-olio vastaa siitä, että oikean pelaajan näkyvyysdata syötetään FogOfWar-oliolle, joka piirtää datan tekstuuriin.

FogOfWar-olion luoman tekstuurin jokainen pikseli, eli tekseli (kuviointialkio), kuvastaa yhtä pelimaailman ruutua. Jokainen tekseli väritetään sen mukaan, mikä näkyvyys sillä pelaajalla, jonka näkyvyys halutaan visualisoida, on kyseiseen ruutuun. Näkyvissä olevien ruutujen tekselit ovat punaisia, tutkittujen ruutujen vihreitä, ja piilotettujen sinisiä. Tämä yksinkertaisti varjostimen kirjoittamista, kun voitiin suoraan hyödyntää tekstuurin tekselin RGB-värikanavia piirrettävän pikselin lopullisen värin muodostamisessa. Kuvassa 18 näkyy sodan sumun datatekstuuri tietyllä hetkellä. Pelin hahmot ovat liikkuneet jonkin verran paljastaen piilotetulta alueelta pelimaailmaa.



Kuva 18. Sodan sumun datasta piirretty tekstuuri, jossa punaiset alueet ovat hahmojen tämänhetkisen näkyvyyden alueita, vihreät tutkittua aluetta ja sininen piilotettua aluetta.

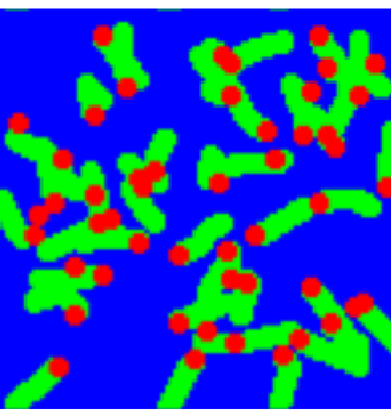
Kun tekstuuri asetettiin käyttämään bilineaarista suodatusta pistemäisen suodatuksen sijaan esimerkkikoodin 6 kaltaisesti, tekstuuria saatiin sumennettua hieman, mikä tuotti esteettisemmän lopputuloksen.

```
fowTexture = new Texture2D(mapGrid.size.x, mapGrid.size.y, TextureFor-
mat.RGBA32, 0, false);
fowTexture.filterMode = FilterMode.Bilinear;
```

Esimerkkikoodi 6. Tekstuurin luonti bilineaarisella suodatuksella. Tekstuurin pikselimäärä on sama, kuin sodan sumun laskennassa käytettävän ruudukon koko.

Bilineaarista interpolaatiota käyttävässä suodatuksessa kuvan pikselin väriarvoon vaikuttaa neljän lähimmän viereisen pikselin väriarvot painotettuina (28), kun taas pistemäisessä suodatuksessa pikselin väriarvo säilyy alkuperäisenä. Tekstuurin sumennus ei sotke sodan sumun dataa, sillä data säilyy aina tarkkana pelaajaolioiden taulukoissa. Sodan sumun tekstuuria ei koskaan lueta koodissa, vaan sen tehtävä on ainoastaan visualisoida sodan sumun informaatio pelimaailmaan. Siksi tekstuuria voidaan manipuloida esteettisemmäksi, kunhan se säilyttää sodan sumun informaation riittävän tarkasti.

Kuvassa 19 näkyy sumennettu sodan sumun datatekstuuri tietyllä hetkellä.



Kuva 19. Sodan sumun datatekstuuri bilineaarisella suodatuksella, joka sumentaa datan parempaa visualisointia varten verrattuna kuvan 18 pistemäisellä suodatuksella luotuun dataan.

Esimerkkikoodissa 7 päivitetään sodan sumun datatekstuurin pikselien värit sodan sumun datan perusteella. Metodiin syötetään argumenttina sen pelaajan näkyvyysinformaatio, joka halutaan visualisoida pelimaailmaan.

```
void UpdateFoWTexture(TileVisibilities[,] tileVisibilities)
{
    for (int y = 0; y < mapGridSize.y; y++)
    {
        for (int x = 0; x < mapGridSize.x; x++)
        {
            fowTexture.SetPixel(x, y, FoWColors[(int)tileVisibili-
ties[x, y]]);
        }
    }

    fowTexture.Apply();
}

Color[] FoWColors = new Color[] {
    Color.red,
    Color.green,
    Color.blue
};
```

Esimerkkikoodi 7. Sodan sumun datatekstuurin piirtäminen näkyvyysinformaation perusteella.

4.4 Visualisointi pelimaailmaan

Päivitetty sodan sumun datatekstuuri viedään joka kuvakehyksessä pelimaailman maaston ja halutessa myös kaikkien pelimaailman hahmojen materiaalien

varjostimelle. Tämä toteutettiin syöttämällä tieto materiaalien MaterialPropertyBlockeihin. Toteutuksessa GameManager-olio vastasi tiedon syöttämisestä pelimaailman olioille, kuten esimerkkikoodissa 8 on tehty.

```
void UpdateSceneMaterials()
{
    Renderer[] renderers = Object.FindObjectsOfType<Renderer>();

    foreach(Renderer renderer in renderers)
    {
        // Set the fog of war texture property in the property block
        MaterialPropertyBlock propertyBlock = new MaterialProperty-
Block();
        propertyBlock.SetTexture("_FogOfWarData", _fogOfWar.GetFoWTex-
ture());
        propertyBlock.SetFloat("_WorldSize", _planeSize.x);
        propertyBlock.SetFloat("_BlurRadius", 1f);
        renderer.SetPropertyBlock(propertyBlock);
    }
}
```

Esimerkkikoodi 8. Sodan sumun datatekstuurin syöttäminen pelimaailman olioiden materiaalien varjostimille.

Suorituskyvyn optimoinnin kannalta voisi olla järkevämpää pitää välimuistissa tallessa kaikkien pelimaailman olioiden Renderer-komponentit, jotta niitä ei tarvitsisi hakea joka kehys uudelleen Object.FindObjectsOfType-metodilla, jonka käyttö on hidasta (29). GameManagerin toteutus ei kuitenkaan kuulu tämän insinöörityön itse sodan sumun toteutukseen, vaan ainoastaan sen testaamiseen, joten todettiin, että asialla ei ole merkitystä sodan sumun suorituskyvyn kannalta testauksen ulkopuolella.

Varjostin on grafiikan piirtämisketjussa ohjelma, joka ajetaan grafiikkasuorittimella. Varjostin sisältää ohjeita, jotka suoritetaan jokaiselle kuvan pikselille (30). Unityssä on sisäänrakennettuja varjostimia, mutta varjostimia voi kirjoittaa myös itse käyttäen Unityn ShaderLab-kieltä, tai vaihtoehtoisesti käyttää visuaalista Shader Graph -työkalua tietyntyyppisissä grafiikkaliukuhihnoissa (31).

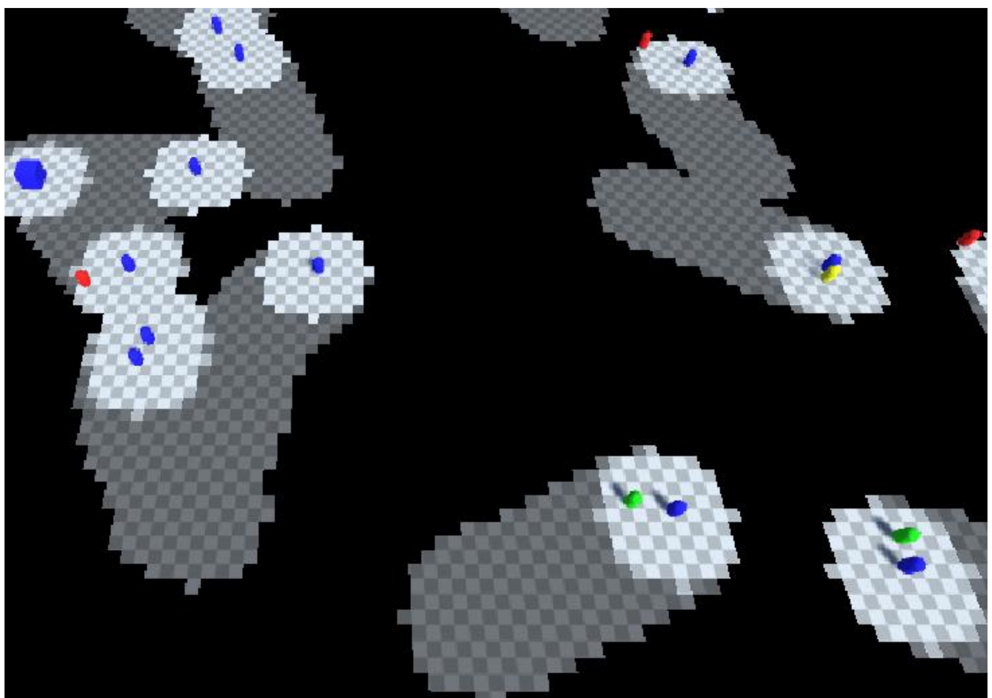
Työn toteutuksessa pelimaailman olioiden materiaalit käyttävät itsekirjoitettua Standard Surface Shader -varjostinta. Standard Surface Shadereissa on etuna se, että niissä ei tarvitse itse huolehtia esimerkiksi valojen ja varjojen toteutuksesta, vaan niillä on helppoa käsitellä kappaleiden pintoja häiritsemättä muuta

graafista laskentaa (32). Esimerkkikoodissa 9 lasketaan varjostimessa materiaalin ulostuloväri sodan sumun perusteella. Sumun datatekstuurin tekselin punaisen värin arvolla kerrotaan materiaalin albedo-tekstuurin tekselin väri, ja vihreällä arvolla kerrotaan tummennettu albedo-tekstuurin väri. Piilotetut alueet muodostavat mustan värin, koska mustilla alueilla sumun datatekstuurin R- ja G-kanavien arvot ovat nollia.

```
void surf (Input IN, inout SurfaceOutput o)
{
    fixed4 albedoColor = tex2D(_MainTex, IN.uv_MainTex);
    fixed4 fogColor = tex2D(_FogOfWarTex, IN.worldPos.xz /
_WorldSize);

    o.Albedo =
        fogColor.r * albedoColor.rgb
        + fogColor.g * (albedoColor.rgb / 2);
}
```

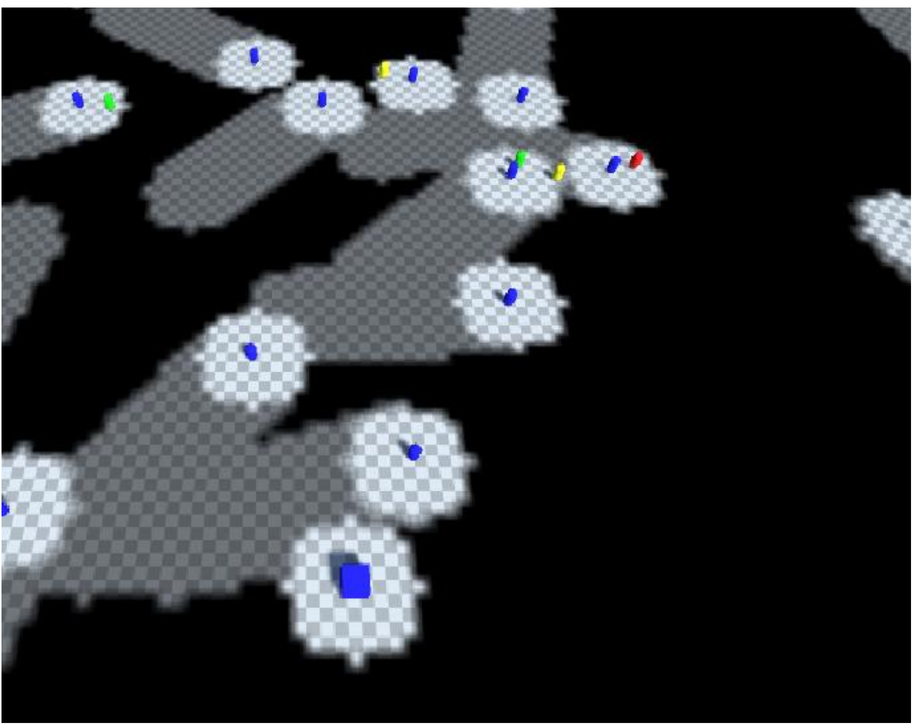
Esimerkkikoodi 9. Varjostimen ulostulovärin laskenta.

Kuvassa 20 näkyy, miten yksittäinen pelimaailman ruutu on selkeästi jossain sodan sumun kolmesta tilasta (näkyvä, tutkittu tai piilotettu), kun sodan sumun datatekstuuria ei ole sumennettu bilineaarisella suodatuksella.



Kuva 20. Sodan sumu maailmassa tekstuurin pistemäisellä suodatuksella.

Kuvassa 21 on lopputulos siitä, kun datatekstuurissa käytetään bilineaarista suodatusta, ennen kuin se annetaan varjostimelle. Ruutujen rajat ovat pehmentyneet, mutta ruutumaisuuden erottaa silti selkeästi.



Kuva 21. Sodan sumu bilineaarisella tekstuurin suodatuksella.

Vaikka tekstuuri toimikin bilineaarisella suodatuksella, joka sumensi tekstuuria, se tuotti pelimaailmassa silti melko rosoista sumua. Jotta sodan sumusta sai vielä esteettisempää, tekstuuria sumennettiin varjostimessa vielä lisää käyttäen Gaussin sumennusta. Gaussin sumennus saatiin aikaan yhdistämällä käsiteltävän pikselin arvoon sen viereisten pikseleiden arvoja oikeassa suhteessa, jokseenkin samaan tapaan kuin bilineaarisessa suodatuksessa, mutta hieman eri säännöillä. Bilineaarista sumennusta on käsitelty luvussa 4.3. Kuvassa 22 näkyy sumentamaton esimerkkikuva ja sama kuva Gaussin sumennuksella. Kuvien vieressä on pikseleiden sumennuksessa käytettävät matriisit.

Operation	Kernel ω	Image result g(x,y)
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

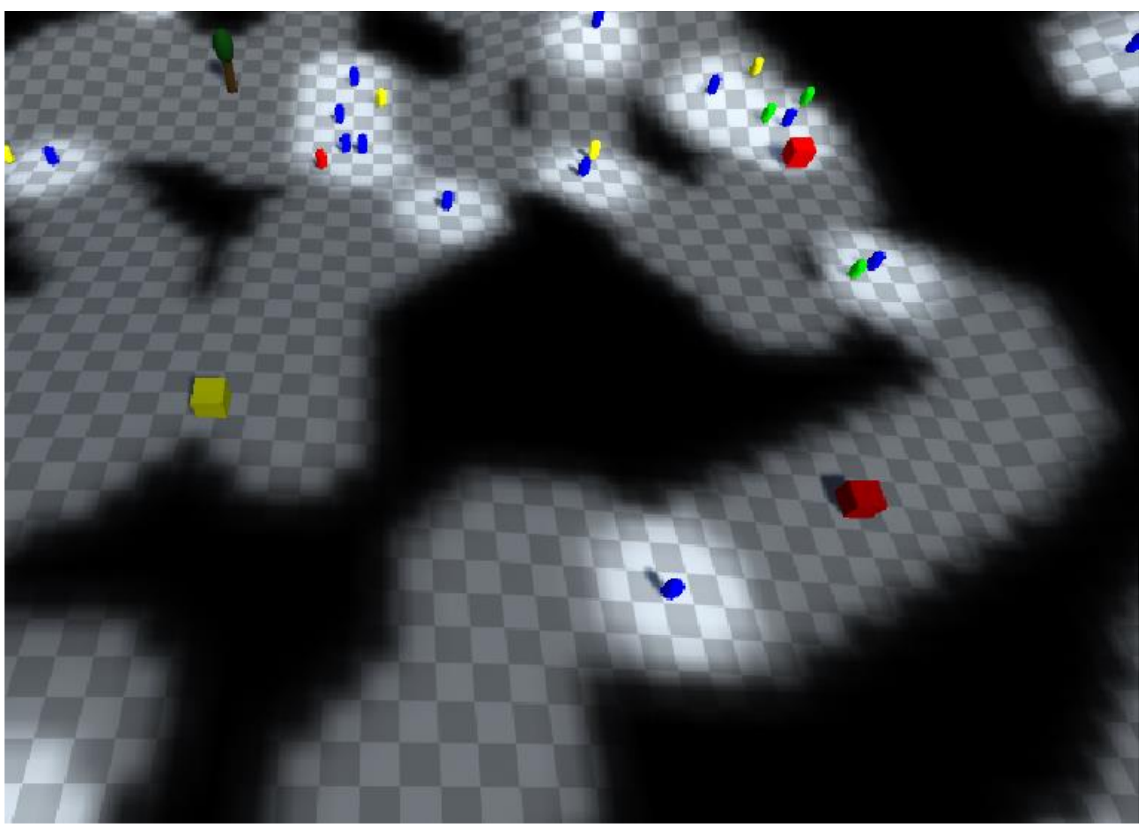
Kuva 22. Approksimaatio Gaussin sumennuksen pikseleiden näytteenottojen painotusarvoista (33).

Kun varjostimessa lasketaan kuviointialkion väri, voidaan sodan sumun datatekstuurista ottaa näytteitä viereisistä pikseleistä. Esimerkkikoodi 10 näyttää, miten Gaussin sumennuksen matriisin painotuksilla viereisten pikseleiden värejä käytetään lopullisen värin muodostamisessa.

```
// Gaussian blur
// =============

// https://en.wikipedia.org/wiki/Kernel_(image_processing)
// Note: Manually unrolled for-loops to avoid compiler confusion

// The temporary additive color from sampling
fixed4 tempFogSampleColor = 0;

// Corner pixels, weight 1
tempFogSampleColor += tex2D(_FogOfWarData, (worldPos + float2(1, 1)) *
_FogOfWarData_TexelSize.xy);
tempFogSampleColor += tex2D(_FogOfWarData, (worldPos + float2(1, -1))
* _FogOfWarData_TexelSize.xy);
tempFogSampleColor += tex2D(_FogOfWarData, (worldPos + float2(-1, -1))
* _FogOfWarData_TexelSize.xy);
tempFogSampleColor += tex2D(_FogOfWarData, (worldPos + float2(-1, 1))
* _FogOfWarData_TexelSize.xy);

// Middle pixels, weight 2
tempFogSampleColor += 2 * tex2D(_FogOfWarData, (worldPos + float2(0,
1)) * _FogOfWarData_TexelSize.xy);
tempFogSampleColor += 2 * tex2D(_FogOfWarData, (worldPos + float2(1,
0)) * _FogOfWarData_TexelSize.xy);
tempFogSampleColor += 2 * tex2D(_FogOfWarData, (worldPos + float2(0, -
1)) * _FogOfWarData_TexelSize.xy);
tempFogSampleColor += 2 * tex2D(_FogOfWarData, (worldPos + float2(-1,
0)) * _FogOfWarData_TexelSize.xy);

// Center pixel, weight 4
tempFogSampleColor += 4 * tex2D(_FogOfWarData, worldPos * _FogOf-
WarData_TexelSize.xy);

// Final fog data color: color from combined samples divided by sample
count
fixed4 fogData = tempFogSampleColor / 16;
```

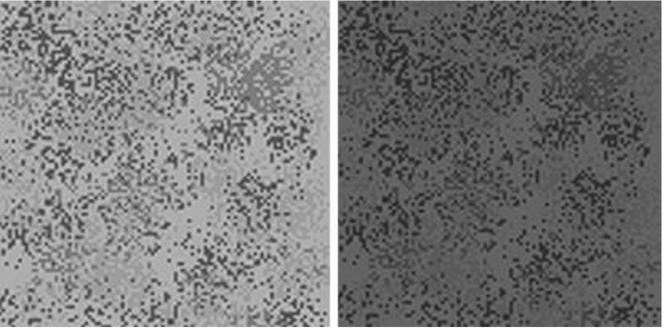
Esimerkkikoodi 10. Sodan sumun datatekstuurin Gaussin sumennus varjostimessa.

Kuva 23 näyttää pelimaailmassa Gaussin sumennuksen tuottaman lopputuloksen, jossa ruutujen rajat ovat hälventyneet huomattavasti.



Kuva 23. Sodan sumu Gaussin sumennuksella ja bilineaarisella tekstuurin suodatuksella.

Sumennuksen jälkeen sumu näytti jo melko esteettiseltä ja alkuperäisen tavoitteen kaltaiselta. Sumusta saatiin kuitenkin vielä esteettisempää, kun sumulle ja piilotetulle alueelle lisättiin omat kuvan 24 kaltaiset tekstuurit.



Kuva 24. Piilotetun alueen, eli tässä tapauksessa pilvien, kuvatekstuuri vasemmalla, ja sumun kuvatekstuuri oikealla.

Nämä tekstuurit laitettiin myös liikkumaan omaan tahtiin, millä saatiin aikaan tuulivaikutteisen ja volymetrisen usvan tuntua. Tekstuurit saatiin liikkumaan ottamalla aika mukaan näytteenottoon. Koodin selkeyttämiseksi piilotettu alue nimettiin pilviksi, ”cloud”, ja tutkittu alue sumuksi, ”fog”. Liikkuvaa tekstuuria käytettäessä on huomioitava, että kuvatekstuuri toistuu vaaka- ja pystysuunnassa. Kuten kuvassa 24 näkyy, tässä toteutuksessa käytettiin tekstuureja, jotka eivät toistuneet esteettisesti, vaan toistuvan tekstuurin raja näkyi selkeästi, mutta demonstraatio- ja testausnäkökulmasta selkeällä rajalla ei ollut merkitystä. Esimerkkikoodissa 11 sumun ja pilvien kuvatekstuureista otetaan näytteitä ajan funktiona, millä piilotetun alueen ja sumun tekstuureihin saatiin liike.

```
fixed4 fogColor = tex2D(_FogTex, IN.uv_FogTex + _Time.y * _FogScroll-
Speed.xy * scrollMultiplier);
fixed4 cloudColor = tex2D(_CloudTex, IN.uv_CloudTex + _Time.y *
_CloudScrollSpeed.xy * scrollMultiplier);
```

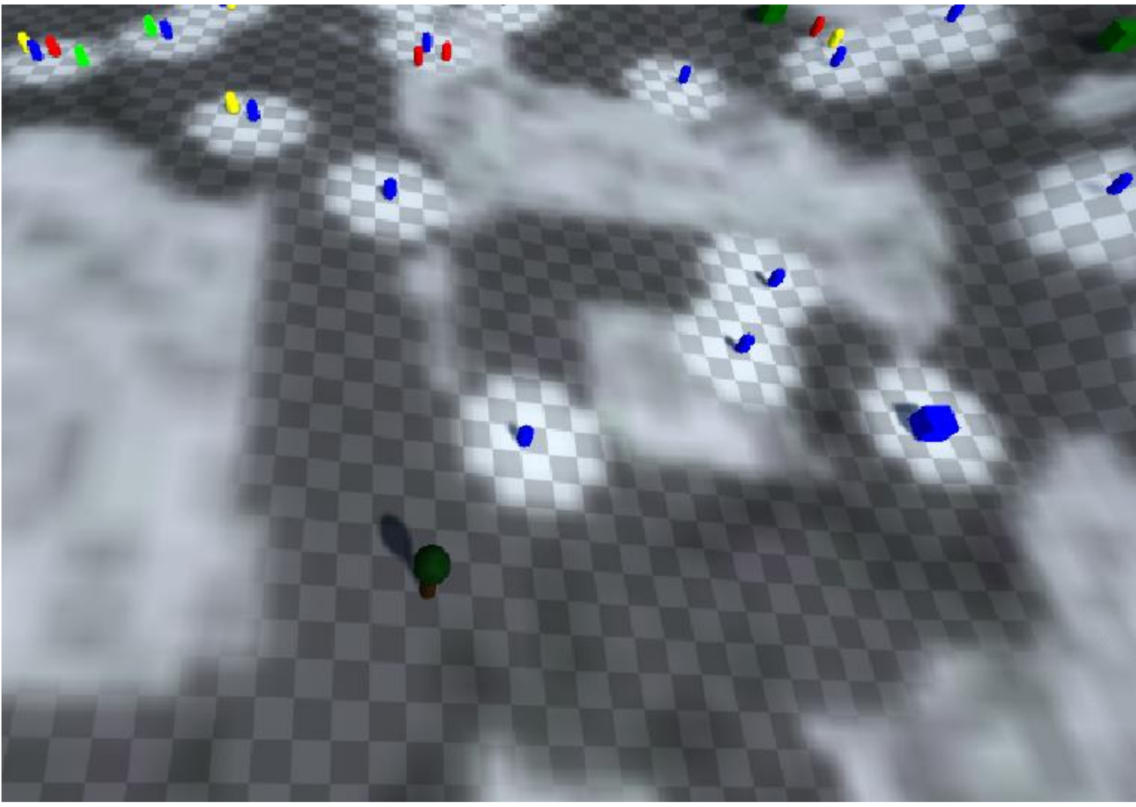
Esimerkkikoodi 11. Sumun ja piilotetun alueen näytteenotto ajan funktiona.

Tämän jälkeen lopullinen kuviointialkion väri saatiin yhdistämällä kaikki sumun kuvatekstuurit ja materiaalin alkuperäinen väri sodan sumun datatekstuurin perusteella, kuten esimerkkikoodissa 12 tehdään.

```
// Final color: Add all fog of war textures together based on fogData
o.Albedo =
    (
        textureColor.rgb * fogData.r
        + textureColor.rgb * fogColor.rgb * fogData.g
        + cloudColor.rgb * fogData.b
    ) * _Tint.rgb;
```

Esimerkkikoodi 12. Lopullisen tekselin värin laskeminen. Mukaan on tuotu vielä värisävy, ”Tint”, jolla voidaan värjätä materiaali halutulla värillä.

Kuva 25 näyttää lopullisen tuloksen pelimaailmassa. Tässä versiossa on yhdistettynä sodan sumun datatekstuurin bilineaarinen sumennus ja Gaussin sumennus sekä liikkuvat sumu- ja pilvitekstuurit tutkituille ja piilotetuille alueille.



Kuva 25. Lopullinen sumennettu sodan sumu liikkuvine tekstuureineen.

Toteutuksen sodan sumu on piirretty suoraan pelimaailman ja sen kappaleiden pintaan. Tämän vuoksi muun muassa kolmiulotteiset maastonmuodot saattavat jopa piilotetuilla alueilla paljastaa informaatiota enemmän kuin olisi tarkoitus, sillä alueiden siluetit saattavat erottua muuta maastoa vasten kameran kuvakulmasta riippuen.

Vastustajien hahmojen piilottaminen tutkitulta ja piilotetulta alueelta on muun logiikan vastuulla: pelkkä sodan sumun varjostin ei poista niitä pelimaailmasta millään lailla. Koska toteutuksessa pelimaailman hahmot käyttävät samaa sodan sumun materiaalia kuin pelimaailman maasto, ne saavat samat sumun teksturoidut vaikutukset pintaansa, mutta se ei poista niiden näkyvyyttä maailmasta. Pelimaailman hahmojen ei välttämättä tarvitse edes käyttää sodan sumun varjostinta materiaaleissaan, mutta tässä toteutuksessa kaikki pelimaailman oliot käyttävät samaa varjostinta. Mikäli halutaan piilottaa hahmoja pelimaailman piilotetuilta ja tutkituilta alueilta, hahmot pitää piilottaa asettamalla niiden piirrettävä olio epäaktiiviseksi silloin, kun ne ovat ruudussa, jota pelaaja ei

aktiivisesti näe. Esimerkkikoodi 13 näyttää, miten toteutuksessa pelaajaolio suorittaa kaikkien niiden hahmojen piilottamisen, jotka eivät ole pelaajan näkemien ruutujen alueella. SetRenderVisibility on hahmo-olion metodi, jota pelaaja-olio kutsuu tarvittaessa, mutta selkeyden vuoksi kyseinen metodi on kirjoitettu samaan esimerkkikoodiin.

```
void Update()
{
    foreach (Unit unit in gameManager.allUnits)
    {
        // Case: own team units
        if (unit.GetTeam() == GetTeam())
        {
            unit.SetRenderVisibility(TileVisibilities.VISIBLE);
        }
        // Case: enemy team units
        else
        {
            Vector2Int unitPosition = unit.GetOccupiedTileCoordi-
nates();
            unit.SetRenderVisibility(tileVisibilities[unitPosition.x,
unitPosition.y]);
        }
    }
}

void SetRenderVisibility(TileVisibilities visibility)
{
    switch (visibility)
    {
        case TileVisibilities.VISIBLE:
            mesh.SetActive(true);
            break;
        case TileVisibilities.EXPLORED:
            mesh.SetActive(_canBeSeenOnExplored);
            break;
        case TileVisibilities.HIDDEN:
            mesh.SetActive(false);
            break;
    }
}
```

Esimerkkikoodi 13. Ylempänä Player-olion Update-metodin koodia, jossa kaikkien pelimaailman hahmojen piirto asetetaan päälle tai pois niiden sijainnin perusteella. Koodi ajetaan vain sillä pelaajalla, jonka näkyvyys halutaan visualisoida maailmassa. Alempana SetRenderVisibility on Unit-olion metodi.

4.5 Toteutuksen yleiskäyttöisyys

Toteutuksesta pyrittiin tekemään mahdollisimman yleiskäyttöinen. Koodia suunniteltaessa pyrittiin välttämään lopputulosta, jossa sodan sumun toteutusta

käyttävässä projektissa joutuisi tekemään suuria kompromisseja, jotta sodan sumu olisi yhteensopiva projektin kanssa. Tämä yritettiin ratkaista eriyttämällä sodan sumun laskenta ja sen testaustyökalut omiin kokonaisuuksiinsa. GameManager-, Player- ja Unit-oliot ovat vain ja ainoastaan sodan sumun testaamista varten luotuja olioita, jotka yrittävät toteuttaa mahdollisimman yleisiä reaaliaikaisen strategiapelin olioiden toiminnallisuuksia. Näiden olioiden oletetaan olevan jo valmiiksi toteutettu sellaisessa projektissa, johon tämä sodan sumu mahdollisesti halutaan liittää.

Toteutuksen sodan sumun oliot, FogOfWar, MapGrid, Tile, PlayerVisibility ja UnitLineOfSight sekä varjostin FogOfWar.shader ovat ne tiedostot, jotka on suunniteltu vietäväksi ulkoiseen projektiin, jossa sodan sumua halutaan käyttää. Nämä oliot sisältävät julkisia metodeja, joita kutsumalla sodan sumua pystyy käyttämään ulkoisessa projektissa. On kuitenkin useita eri sodan sumun mekaniikkoja, joita ei pystytä toteuttamaan pelkästään näiden olioiden ominaisuuksilla, vaan mekaniikkojen toteutukseen tarvitaan pelaaja- ja hahmo-olioiden logiikkaa. Ulkoisen projektin vastuulle jää hyvinkin paljon pelimaailman olioiden käsittelystä, kuten niiden Renderer-komponenttien materiaalien muokkaaminen ja piirrettävien kappaleiden piilottaminen alueilta, joihin pelaajalla tai joukkueella ei ole näkyvyyttä.

Itse sodan sumun toteutuksen oliot kirjoitettiin C#-kielellä pyrkien käyttämään mahdollisimman vähän Unityn omia kirjastoja. Muutamia Unityn kirjastoja kuitenkin käytettiin, sillä niiden datatyypit ja metodit koettiin erittäin hyödyllisiksi olioiden toteutuksissa. Ottaen huomioon, että toteutus on suunniteltu käytettäväksi nimenomaan Unity-projektissa, Unityn kirjastojen käyttämistä ei koettu huonoksi ratkaisuksi.

5 Yhteenveto

Insinöörityössä tutkittiin sodan sumun merkitystä strategiapeleissä ja toteutettiin Unity-pelimoottorilla vanhanaikainen ruutupohjainen sodan sumu 3D-pelin pohjaan. Työn lopputuloksena oli tavoitteiden mukainen sodan sumun perusrakenne reaaliaikaiseen strategiapeliin.

Pelimekaanisesti työn toteutus oli hyvin yksinkertainen, ja se täytti vain sodan sumun vähimmäisvaatimuksen. Useiden nykyaikaisten pelien sodan sumuissa maastonmuodot ja muut esteet vaikuttavat hahmojen näkökenttiin. Tämän insinöörityön tavoitteisiin ei kuulunut mekaniikoiltaan monimutkainen sodan sumu, jossa esteet rajoittavat hahmojen näkyvyyttä. Sen kaltainen sodan sumun toteutus on kuitenkin pelattavuudeltaan strategisesti monipuolisempi, ja siksi se olisi ollut hyvä lisä myös tähän insinöörityöhön. Etenkin sellainen mekaniikka, jossa korkeilla pelimaailman alueilla olevilla hahmoilla on suurempi näkökenttä alamäkeen, olisi todennäköisesti ollut melko yksinkertaista toteuttaa työhön. Kyseinen mekaniikka on työn esikuvana käytetyssä kaksiulotteisessa reaaliaikaisissa strategiapelissä Age of Empiresissa (34). Projektista puuttuu myös toteutus sille, että tutkitulla alueella olevat vastustajien staattiset hahmot eivät päivittyisi pelaajalle reaaliajassa.

Visuaaliselta osalta tavoitteisiin päästiin. Sodan sumusta tuli kohtuullisen esteettinen ja sumun visuaalinen tyyli saavutti sen tavoitteen, johon alun perin pyrittiin. Toteutus ei itsessään ole välttämättä riittävän hyvä sellaisenaan julkaistavaksi strategiapelissä nykyajan vaatimuksiin nähden, mutta se on hyvä lähtökohta sodan sumun toteutukselle, jota on helppo jatkaa haluamallaan tavalla. Sodan sumun visuaalisesta tyylistä olisi saanut nykyaikaisemman käyttämällä sumun varjostimessa parallaksikartoitusta. Parallaksikartoituksella tuotetaan näennäistä syvyysvaikutelmaa kappaleen pintaan, millä olisi voinut saada sodan sumun ilmentymään selvästi pelimaailman pinnan yläpuolella, kuten Civilization V -pelin pilvistä koostuva sodan sumu (12). Parallaksikartoituksen opetteluun ei kuitenkaan jäänyt tarpeeksi aikaa, ja se ei kuulunut alkuperäiseen tavoitteeseen, joten sen toteutuksesta luovuttiin.

Suorituskyvyn suhteen toteutuksessa on hyvinkin paljon parannettavaa. Ruutujen laskennassa tapahtuu suorituskyvyn optimoinneista huolimatta edelleen paljon turhaa, kuten esimerkiksi se, että pelaajaolio laskee joka kehys uudelleen näkyvyytensä, eikä silloin, kun se muuttuu. Vaikka pelaajaolio ei kuulukaan itse sodan sumun toteutukseen, vaan sen testaamiseen, on sodan sumu silti toteutettu sillä oletuksella, että pelin pelaajaolio laskee joka kehys näkyvyytensä uudelleen. Lisäksi pelaajan useat hahmot saattavat nähdä samat pelimaailman

ruudut, jolloin niitä ei tarvitsisi käsitellä moneen kertaan, mutta tässä toteutuksessa sellaista tarkistusta ei tehdä. Mitä suuremmat näkökentät hahmoilla on, sitä todennäköisemmin päällekkäisiä ruutuja käydään läpi, jolloin tapahtuu turhaa laskentaa.

Koodin muuttujien datatyypit ja olioiden rajapinnat pyrittiin tekemään mahdollisimman järkeviksi sekä luettavuuden että suorituskyvyn kannalta, mutta näiden kahden välillä piti tehdä monesti kompromisseja. Insinöörityötä tehtäessä kokeiltiin muun muassa erilaisia datatyyppejä näkyvyysdatalle. Toteutuksessa yksittäisen pelaajan näkyvyysdata on tallennettu pelaajalle itselleen näkyvyystiloja sisältävään kaksiulotteisen taulukkoon. Taulukko sisältää siis vain yhden pelaajan näkyvyysdatan, mikä hankaloittaa useiden pelaajien näkyvyysdatan yhdistämistä. Jos data olisi tallennettu näkyvyystilan sijaan bittimaskeina, joissa yksittäinen bitti kertoo yksittäisen pelaajan näkyvyyden tiettyyn pelimaailman ruutuun, kaikkien pelaajien näkyvyysdatat olisivat voineet olla yhdessä taulukossa. Tällöin useiden pelaajien näkyvyysdatojen visualisointi helpottuisi huomattavasti, kun olisi vain yksi taulukko, jonka data pitää lukea tekstuurin muodostamiseksi. Työssä ei kuitenkaan saatu bittimaskiratkaisua toimimaan suorituskykyisesti, joten siitä luovuttiin. On kuitenkin hyvin todennäköistä, että suunnittelemalla paremmin muuttujien datatyypit ja olioiden rajapinnat suorituskykyä voitaisiin parantaa huomattavasti.

Toteutuksesta puuttuu myös suorituskyvyn kannalta hyvinkin olennainen asia, nimittäin monisäikeistäminen. Insinöörityötä tehdessä pyrittiin toteuttamaan monisäikeistäminen näkyvyyden laskentaan, jossa jokainen pelaaja teettäisi näkyvyyslaskelmansa samanaikaisesti rinnakkain. Tämä voisi parantaa suorituskykyä huomattavasti, sillä nyt toteutuksessa kaikki laskenta tapahtuu peräkkäin, mikä kasvattaa yhden kehyksen laskenta-aikaa. Monisäikeistämistä ei kuitenkaan saatu toimimaan insinöörityötä tehtäessä. Monisäikeistämisestä on sitä enemmän hyötyä, mitä useamman pelaajan näkyvyys lasketaan samalla laitteella. Peleissä tekoälyä vastaan tämä on tyypillistä. Sen sijaan moninpeleissä, joissa useampi ihmispelaaja pelaa kukin omalla laitteellaan, laskentaa voidaan hajauttaa usealle eri laitteelle. Jokainen pelaaja voisi laskea vain oman näkyvyytensä omalla laitteellaan, mikä helpottaisi jokaisen pelaajan laitteen

laskentakuormaa huomattavasti isojen pelaajamäärien tapauksissa. Laskentaa voisi myös mahdollisesti siirtää CPU:lta GPU:lle käyttämällä Unityn compute shader -varjostimia, mikä voisi parantaa suorituskykyä.

Työn sisältämien kooditiedostojen yleiskäyttöisyyttä on vaikea arvioida kokeilematta niitä erilaisissa peliprojekteissa. Koodi on kuitenkin kirjoitettu melko puhtaalla C#-kielellä vähäisellä määrällä riippuvuuksia ulkoisiin kirjastoihin. Koodi on luettavuudeltaan koodin englannin kielen takia yleiskäyttöistä, sillä iso osa pelikehityksestä toteutetaan englanniksi. Koodi on myös pääsääntöisesti hyvin yksinkertaista, mikä helpottaa sen ymmärtämistä. Huono puoli toteutuksen koodissa on se, että iso osa sodan sumun toimintalogiikasta odotetaan tapahtuvan pelin pelaaja- ja hahmo-olioissa, jotka eivät varsinaisesti kuulu tämän insinöörityön julkisesti jaettavaksi tarkoitettuun toteutukseen. Työn koodi on vain rajapinta sodan sumun datan käsittelyyn, mutta se, mitä datalla tekee, on ulkoisen projektin vastuulla. Ideaalitilanteessa työn kooditiedostot olisi pakattu Unity-paketiksi, jonka voisi ladata Unityn pakettienhallinnan avulla. Tämä ei kuitenkaan kuulunut alkuperäiseen tavoitteeseen, ja aikarajoitusten vuoksi siitä luovuttiin.

Lähteet

1 How do you compare and contrast dynamic games with perfect information and imperfect information? Verkkoaineisto. LinkedIn. <https://www.linkedin.com/advice/1/how-do-you-compare-contrast-dynamic-games-perfect-information>. Luettu 6.3.2024.

2 Ross, Don. 2023. Game Theory. Verkkoaineisto. Stanford Encyclopedia of Philosophy. <https://plato.stanford.edu/entries/game-theory/>. Julkaistu 25.1.1997. Päivitetty 3.9.2023. Luettu 21.4.2024.

3 Chessboard. 2024. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Chessboard>. Päivitetty 4.5.2024. Luettu 8.5.2024.

4 Walker, Michael. 2022. The Pre-Flop Mistakes to Avoid in Texas Hold'em. Verkkoaineisto. WinningBetsUSA. <https://winningbetsusa.com/online-casinos/the-pre-flop-mistakes-to-avoid-in-texas-holdem/>. Julkaistu 21.12.2022. Luettu 21.4.2024.

5 Puumalainen, Simo. 2015. Sodan sumu ja kitka sodankäynnin yleisissä periaatteissa. Pro gradu. Maanpuolustuskorkeakoulu. Doria-tietokanta.

6 Lautapelit ja älypelit. Verkkoaineisto. <https://hobbyhall.fi/fi/lapset/lelut-yli-3-vuotiaille-lapsille/lautapelit-ja-alypelit/f/all/laivanupotus>. Luettu 6.3.2024.

7 Goldring, Alex. 2019. Fog of War. Verkkoaineisto. Medium. <https://medium.com/@travnick/fog-of-war-282c8335a355>. 7.6.2019. Luettu 22.2.2024.

8 Jung, Jaewon. 2016. A Story of Fog and War. Verkkoaineisto. Riot Games. <https://technology.riotgames.com/news/story-fog-and-war>. 13.5.2016. Luettu 16.2.2024.

9 Score. 2024. Verkkoaineisto. Fandom. <https://ageofempires.fandom.com/wiki/Score>. Päivitetty 24.2.2024. Luettu 5.3.2024.

10 Hunt, Cale. 2018. Beginner's guide to maps in Minecraft: Windows 10 and Xbox One. Verkkoaineisto. Windows Central. <https://www.windowscentral.com/beginners-guide-maps-minecraft-windows-10-edition>. Päivitetty 30.7.2018. Luettu 6.3.2024.

11 Huge5000RTSFan. 2021. Verkkoaineisto. Age of Empires Forums. <https://forums.ageofempires.com/t/how-to-design-up-to-date-fog-of-war-by-aoe4/114904>. 23.1.2021. Luettu 2.3.2024.

12 VainApocalypse. 2016. Fog of War thoughts. Verkkoaineisto. Civfanatics. <https://forums.civfanatics.com/threads/fog-of-war-thoughts.599971/>. 17.10.2016. Luettu 23.2.2024.

13 Abent, Eric. 2022. Civilization 6 Tips: Succeeding In The Early Game. Verkkoaineisto. SlashGear. <https://www.slashgear.com/780923/civilization-6-tips-succeeding-in-the-early-game/>. Päivitetty 25.2.2022. Luettu 23.2.2024.

14 Total War Saga: Troy. 2020. Videopeli.

15 Door Kickers. Verkkoaineisto. Nintendo. <https://ec.nintendo.com/NZ/en/titles/70010000031012>. Katsottu 17.3.2024.

16 Kaitzilla. 2018. The complete guide to espionage. Verkkoaineisto. Civfanatics. <https://forums.civfanatics.com/threads/the-complete-guide-to-espionage.638613/>. 19.11.2018. Luettu 20.2.2024.

17 Submarine (Civ 6). Verkkoaineisto. Fandom. <https://civilization.fandom.com/wiki/Submarine_(Civ6)>. Luettu 20.2.2024.

18 Spy. 2024. Verkkoaineisto. Team Fortress 2 Wiki. <https://wiki.teamfortress.com/wiki/Spy>. Päivitetty 11.1.2024. Luettu 20.2.2024.

19 Bhatti, Fariha. 2023. CS2 smokes – how to use new responsive Counter-Strike 2 smoke grenades. Verkkoaineisto. PCGamesN. <https://www.pcgamesn.com/counter-strike-2/smokes>. 2023. Luettu 14.4.2024.

20 Prieto, José Antonio. FOW-research. Verkkoaineisto. GitHub. <https://petermcp.github.io/FOW-research/>. Luettu 28.3.2024.

21 MacIntosh, Iain. The Civilization IV Project: Episode 1. Verkkoaineisto. TheSetPieces. <https://thesetpieces.com/gaming/civilization-iv-project-episode-1/>. Luettu 23.2.2024.

22 Abramov, Alex. 2017. Hack Age of Mythology: turning off the fog of war. Verkkoaineisto. Sudonull. <https://sudonull.com/post/69715-Hack-Age-of-Mythology-turning-off-the-fog-of-war>. 21.6.2017. Luettu 23.2.2024.

23 Arielsan. 2018. Implementing Fog of War for RTS games in Unity ½. Verkkoaineisto. Gemserk. <https://blog.gemserk.com/2018/08/27/implementing-fog-of-war-for-rts-games-in-unity-1-2/>. 27.8.2018. Luettu 15.2.2024.

24 Bgolus. 2019. Verkkoaineisto. Unity Forum. <https://forum.unity.com/threads/rendertexture-to-texture2d-too-slow.693850/>. 12.6.2019. Luettu 18.2.2024.

25 Vector3.sqrMagnitude. Verkkoaineisto. Unity. <https://docs.unity3d.com/ScriptReference/Vector3-sqrMagnitude.html>. Luettu 24.3.2024.

26 Tarodev. 2022. Unity Code Optimization - Do you know them all? Verkkoaineisto. YouTube. <https://www.youtube.com/watch?v=Xd4UhJufTx4>. 13.3.2022. Katsottu 24.3.2024.

27 Inequalities. Verkkoaineisto. Purdue University. <https://www.math.purdue.edu/academic/files/courses/2007fall/MA301/MA301Ch2.pdf>. Katsottu 21.4.2024.

28 Nett, Jesse. Bilinear Interpolation. Verkkoaineisto. Portland State University. <https://web.pdx.edu/~jduh/courses/geog493f09/Students/W6_Bilinear%20Interpolation.pdf>. Luettu 15.4.2024.

29 Object.FindObjectsOfType. Verkkoaineisto. Unity Documentation. <https://docs.unity3d.com/ScriptReference/Object.FindObjectsOfType.html>. Luettu 24.3.2024.

30 Vivo, Patricio Gonzalez. 2015. The Book of Shaders. Verkkoaineisto. <https://thebookofshaders.com/01/>. Luettu 15.4.2024.

31 Using Shader Graph. 2024. Verkkoaineisto. Unity Documentation. <https://docs.unity3d.com/Manual/shader-graph.html>. Julkaistu 13.4.2024. Luettu 15.4.2024.

32 Writing Surface Shaders. 2024. Verkkoaineisto. Unity Documentation. <https://docs.unity3d.com/Manual/SL-SurfaceShaders.html>. Julkaistu 13.4.2024. Luettu 15.4.2024.

33 Kernel (image processing). 2023. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Kernel_(image_processing)>. Päivitetty 17.10.2023. Luettu 23.3.2024.

34 Age of Empires. 1997. Videopeli.