KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittely

Antti Niemeläinen

PROTOTYYPIN JATKOKEHITYS JA DOKUMENTOINTI

Opinnäytetyö
Kesäkuu 2019

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Kesäkuu 2019
Tietojenkäsittely

Tikkarinne 9
80200 Joensuu
Puh. 013 260600

Tekijä(t)
Antti Niemeläinen

Nimeke
Prototyypin jatkokehitys ja dokumentointi

Toimeksiantaja
Process Genius Oy

Tiivistelmä

Opinnäytetyön tarkoituksena oli tuottaa Process Genius Oy:lle matemaattisen pelin prototyypin kehitys siihen asteeseen, josta heidän on mahdollista jatkokehittää sitä itse. Prototyyppi oli luotu työharjoittelun aikana. Pelin tarkoituksena on opettaa suunnistusta ja kartanlukua valitsemalla kartasta optimaalisin reitti. Vastaamalla oikein matemaattisiin kysymyksiin, pääsee pelaaja liikkumaan eteenpäin kartalla. Vaihe prototyypistä jatkokehitykseen sisältää sovellusarkkitehtuurin eheyttämistä, sen dokumentointia sekä prosessin raportointia.

Prototyypistä oli tarkoituksena eristää viisi erillistä komponenttia, joita on myös mahdollista hyödyntää tulevissa Unity-projekteissa. Komponentit tuli dokumentoida hyödyntäen UML-kaavioita sekä logiikkaa kuvaavien kaavioiden avulla.

Kehityksen aikana selvisi, että suunnitelmallisuus ja kokemus ovat tärkeitä elementtejä komponenttien luomisessa ja niiden kehityksessä. Suunnittelulle ja kehitykselle on annettava oma aikansa ja niiden liian lähekkäinen sijoitus aikataulussa voi olla haitallista niin kehitykselle kuin suunnittelulle. Pelikehityksessä komponentit ovat yleensä varsin pelikohtaisia, joten niiden hyödyntäminen muissa projekteissa ilman suuria muutoksia on hanakalaa. Lopputuotoksena projekti on sovellusarkkitehtuurillisesti eheämmässä kunnossa, jonka rajapintoja on dokumentoitu ja selkeytetty kaavioiden avulla.

Kieli	Sivuja 38
suomi	

Asiasanat

opinnäytetyö, unity, refaktorointi, UML-mallinnus

THESIS
June 2019
Degree Programme in Business information Technology

Tikkarinne 9
80200 Joensuu
Finland
Puh. 013 260600

Author (s)
Antti Niemeläinen

Title
Further development and documentation of a prototype

Commissioned by
Process Genius Oy

Abstract

The purpose of this thesis was to develop a prototype of mathematical game to the point where it is easier to develop it further. The game prototype was made during my internship at Process Genius ltd. The game's goal is to teach orienteering and map reading by selecting most optimal route from a map. The player can move on the map grid by answering mathematical questions. The phase from the game prototype to further development includes improving the design of the existing code, documenting it and reporting of the process.

The aim for the development was to extract five separate components from the game prototype which can also be utilized in other Unity projects. The Components had to be documented with UML diagrams and charts which describes the code's logic.

During the development it became clear that orderliness and experience are the most valuable elements when developing an intact software architecture. Time is needed between planning the software architecture and actual development. Without it results may be harmful for software's integrity. In the Unity development game components are usually game specifics and using them in other projects might be difficult and includes lots of component's modification. The end result of the thesis's was that project's software architecture is more intact which application programming interfaces are documented and clarified with various charts.

Language

Finnish

Pages 38

Keywords

thesis, unity, refactoring, uml

Sisältö

1 Johdanto 5
2 Lähtökohdat 7
2.1 Suunnistus 7
2.2 Toteutuksen nykytilanne 8
2.3 Refaktoroinnin ja dokumentoinnin tarkoitus 12
3 Työkalut ja niiden valinta 14
3.1 Unity 2018 14
3.2 Visual Studio 2017 15
3.3 Google Drive 15
3.4 Draw.io 16
3.5 Trello 16
3.6 Versionhallinta 17
3.7 Mitä on hyvä koodi? 18
3.8 Optimointi 18
4 Toteutus 19
4.1 Perusluokat Unityssä 19
4.2 Plugin-valintojen tarkastelu 19
4.3 GameManager 20
4.4 MathManager 22
4.5 NodeManager 27
4.6 Player ja Networking 30
4.7 MapReader 31
5 Pohdinta 34
5.1 Ongelmien kohtaaminen 34
5.2 Kuinka olisin voinut suunnitella paremmin 34
5.3 Lopputulos 35
Lähteet 37

1 Johdanto

Tässä opinnäytetyössä käsittelen Process Genius Oy:ssä tekemäni pelin prototyyppiä, jota jatkokehitän Unity 2018 -pelimoottorilla yhden askeleen lähemmäksi markkinakelpoista tuotetta Android-alustoille. Tärkeänä osana jatkokehityksen mahdollistamista on sen dokumentointi, jotta projektin jatkaminen olisi mahdollisimman helppoa ja vaivatonta. Ensimmäisenä vaiheena on prototyypin kartoittaminen, jotta tämän hetkisistä toiminnallisuuksista on selvä kuva. Toisessa vaiheessa tulee suunnitella uusi ja eheä sovellusarkkitehtuuri sekä tehdä siitä mahdollisimman modulaarinen. Tällöin komponentteja on helppo hyödyntää jatkokehityksessä sekä mahdollisissa uusissa projekteissa. Kolmannessa vaiheessa aloitetaan koodin refaktorointi, eli lähdekoodin uudelleen rakentaminen. Kolmannen vaiheen aikana kehitän kokonaisuutta pienissä osissa, jotka koostuvat PDCA-, eli plan-do-check-act -periaatteesta (All About Lean 2016). Syklin alussa pitää suunnitella, mitä tulee tekemään ja jäsentää ongelmakohdat selvästi projektinhallintaan. Jäsentelyn jälkeen alkaa toteutus. Toteutuksen valmistuttua tarkastellaan, onko ratkaisumalli paras mahdollinen. Jos totetus vaatii hiomista, aloitetaan prosessi uudestaan samasta toiminnallisuudesta, muutoin prosessin voi aloittaa uuden toiminnallisuuden kohdalta. Tämän työn tarkoituksena on siis suunnitella ja rekentaa pelinprototyypistä refaktoroinnin ja dokumentoinnin avulla eheä kokonaisuus, jota toimeksiantajan on helppo lähteä jatkamaan. Refaktorointiin kuuluu viiden komponentin eristäminen, jotta niitä pystyy mahdollisesti hyödyntämään myös tulevissa Unity-projekteissa.

Pelien kehityksessä prototyypin tekeminen on yksi tärkeimmistä pelikehityksen vaiheista, sillä siinä muutetaan abstrakti ja teoreettinen peli-idea karkeaksi, mutta testauskelpoiseksi tuotteeksi. Prototyyppivaiheessa pelin päämekaniikat olisi tärkeä pelitestata mahdollisella kohdeyleisöllä, mikä voi antaa parempaa osviittaa siitä, mitä kohdeyleisö haluaa peliltä (Dam & Siang 2019). Pelitestauksella pyritäänkin todentamaan pelin kelpoisuus niin toteuttamisen kuin rahallisen kannattavuuden suhteen. Vaikka peli voisikin olla temaattisesti kohderyhmää miellyttävä, saattavat pelin mekaniikat olla kohderyhmälle vääränlaiset. Kun pelin

prototyyppi osoittautuu tuotantokelpoiseksi ideaksi testauksen myötä, siirtyy peli esituotantoon. Tarkoitukseni onkin dokumentoida tämä siirtymävaihe sovellusarkkitehtuurin suunnittelun-, teknisen toteutuksen-, sekä mahdollisen jatkokehityksen kannalta. Dokumentaatio sovellusarkkitehtuurista tulee sisältämään muun muassa erinäisiä UML-kaavioita, joiden avulla jatkokehityksessä pystytään saamaan helposti selkeä kokonaiskuva pelistä ja sen teknisestä toteutuksesta, käyttötapauksista, rajapinnoista sekä teknisten valintojen kannattavuudesta.

Rakentamani pelin prototyyppi ei ole varsinaisesti noudattanut prototyypille piirteenomaisia ominaisuuksia kuten nopeaa ja selkeitten toiminnallisuuksien kehitystä (Games From Within 2010). Prototyypin tekeminen on ollut oppimisen matka, joka venyi vaatimusten ja ideoinnin myötä muutamaksi kuukaudeksi. Peli on kuitenkin nyt siinä pisteessä, jossa idea on ajateltu toteutuskelpoiseksi; on aika siirtyä eteen päin kehityksessä. Toiminnallisen osuuden laajuuden takia, jouduin jättämään pelitestauksen prototyypin osalta pois, mutta mikään ei estä pelitestausta jatkokehityksen ensimmäisinä askelina. Jotta kehityksen jatkaminen olisi mahdollisimman vaivatonta, on syytä tarkastella lähdekoodin rakennetta ja uudelleen käytettävyyttä. Nämä asiat harvoin täyttyvät pelien prototyypeissä ja siksi lähdekoodin refaktorointi on tärkeä askel jatkokehityksen helpottamiseksi.

Prototyypin olen rakentanut työharjoittelun aikana Process Genius Oy:llä Unity-harjoittelijana. Process Genius Oy on Joensuussa perustettu IT-alan yritys, joka tuottaa erilaisia ratkaisuja sovelluskehitystä hyödyntäen. Yrityksellä on kaksi konttoria: Joensuussa ja Helsingissä, joista Joensuussa sijaitseva on pääkonttori. Process Geniuksella on tällä hetkellä kaksi ihmistä Unity-puolella, mutta työstän prototyypin seuraavaa askelta yksin. Minulla on kuitenkin mahdollisuus saada neuvoja harjoitteluaikani vastaavalta työntekijältä Kimmo Leppäseltä, joka toimii Process Genius Oy:llä Lead Gamification Developerina eli pelillistämisen johtavana kehittäjänä.

2 Lähtökohdat

2.1 Suunnistus

Pelin pääteemana toimii suunnistus ja sen opettelu. Suunnistuksen ideana on karttaa ja kompassia hyödyntäen löytää maastosta rasteja, jotka ovat merkittynä suunnistajan karttaan. Rastit tulee käydä läpi ennalta määrätyssä järjestyksessä, jonka jälkeen palataan lähtöpisteeseen. Ensimmäisenä maaliin saapunut on voittaja. Suunnistajan täytyy lukea karttaa ja valita omien taitojen perusteella mahdollisimman nopea reitti rastien läpikäymiseksi (Suunnistusliitto 2018). Peli voi siis hyödyntää suunnistuksen kahta elementtiä, jotka ovat kartanluku ja reitinvalinta omien taitojen mukaan.

Suunnistuskartta on yleisimmin maastokartta, jonka tulee noudattaa Suunnistusliiton laatimia vaatimuksia. Näitä vaatimuksia ovat muun muassa johdonmukainen värien käyttö, maanpinnan muotojen selkeys, symbolien merkkaus, juostavuus, kartan mittakaava sekä monet värien ja symbolien asetteluvaatimukset (Suunnistusliitto 2017). Mitä enemmän näistä kartan ohjeista saa liitettyä peliin, sen todenmukaisempi ja opettavaisempi suunnistuskokemus pelatessa tulee.

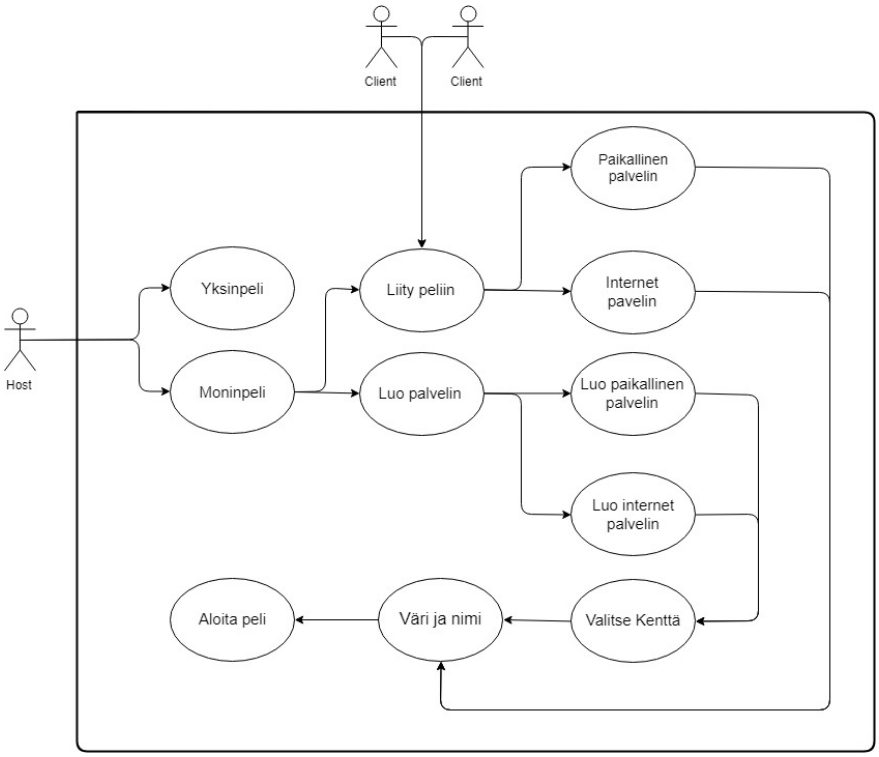
Reitinvalinnassa suunnistaja miettii reitin omien fyysisten kykyjensä perusteella ja koettaa löytää reitin, joka on itselleen optimaalisin. Pelissä ei tietenkään voi mitata suunnistajan fyysisiä kykyjä, joten reitinvalinnan tulee olla riippuvainen jostain muusta kyvystä. Tässä suunnistusteemaisessa pelissä se on matematiikka. Maasto ja sen erilaiset esteet vaikuttavat siihen, kuinka vaikeita kysymyksiä suunnistaja kohtaa matkallaan. Tiheässä metsässä on haastavampia kysymyksiä kuin tienpientareella, mutta matka saattaa olla lyhyempi metsän läpi kuljettaessa. Suunnistajan on tehtävä valinta omien matemaattisten taitojen perusteella: onko järkevämpää kulkea helppoa vai vaikeampaa reittiä, kummassa säästää enemmän aikaa?

Maastokarttaa sekä reitin valintaan käyn tarkemmin läpi niistä vastuussa olevissa MapEditor- ja MathManager-komponenteissa. Komponenttien tarkoituksena on tehdä suunnistuksesta mahdollisimman todenmukainen kokemus ja niitä tullaan hiomaan koko kehityksen ajan.

2.2 Toteutuksen nykytilanne

Tämä luku käsittää prototyypin nykytilan, eli kohdan, josta askel kohti sovellusarkkitehtuurin refaktorointia alkaa. Aloitushetkellä prototyyppi koostuu kahdesta pluginista ja scenestä, 24 scriptistä sekä sadoista pienemmistä komponenteista ja peliobjekteista, joten dokumentoimattomana sen jatkaminen ulkopuoliselle olisi hyvin työläs ja vaikea prosessi. Uudelleen rakentaminen tapahtuu peliobjekteihin liitettävissä scripteissä eli lähdekoodeissä, jotka mahdollistavat pelaajan syötteet ja peliobjektien käyttäytymisen (Unity 2019a). Uudelleen rakentamisella varmistetaan, että kuka tahansa pystyy jatkokehittämään peliä eteen päin ja että sen osia pystyy hyödyntämään mahdollisesti myös muissa projekteissa. Dokumentoimattomassa projektissa saatetaan helposti tehdä virheitä tai tehdä jo kertaalleen kirjoitettu koodi uudelleen.

Prototyyppi jakaantuu kahteen sceneen: aloitusvalikkoon ja peliin. Unityssä scenet ovat kuin pelin sisäisiä tasoja, jotka sisältävät tasolle tarpeellisia peliobjekteja ja elementtejä (Unity 2019b). Aloitusvalikossa pelaaja pystyy valitsemaan pelitavan yksin- ja moninpelin väliltä, joista vain moninpeli on toteutettu. Moninpelivalikossa on mahdollista luoda paikallinen- tai internet-serveri, johon muut käyttäjät pystyvät liittymään IP-osoitteen tai serverilistan avulla. Kun pelaajat ovat liittyneet serveriin, he pystyvät vaihtamaan omaa nimeään ja väriä. Serverin luoja eli host pystyy vaihtamaan pelikenttää, muut pelaajat eivät näe tätä valintaa. Käyttäjien ollessa tyytyväisiä nimi- ja väri-valintaan, he voivat painaa Join-painiketta ilmoittaakseen olevan valmiita aloittamaan pelin. Kun kaikki pelaajat ovat valmiita, pelin lähtölaskenta alkaa ja pelaajat siirtyvät peli-sceneen. Siirtymävaiheessa serveri ottaa vastaan pelaajatiedot ja lähettää kaikille pelaajille pelaajien värit ja nimet (Kuva 1.)



Kuva 1. Aloitusvalikon käyttötapauskaavio.

Peli-scenen kentät rakentuvat scriptable objecteista ja niiden neljästä muuttujasta (Taulukko 1). Scriptable objectit ovat itsenäisiä eikä niitä ole pakko liittää peliobjekteihin. Niiden tarkoituksena on yleesä säilöä dataa muuttujien avulla (Unity 2019c.) Näiden muuttujien pohjalta generoidaan uusia kenttiä, joita pelin jatkokehityksessä voidaan luoda lisää. Level- ja stage-kokonaislukumuuttujat tulevat määrittämään kysymysten operaattorit ja niiden vaikeusasteen. CheckPoints-struct sisältää kokonaislukutaulukon, johon merkitään rastien sijainti ruudukossa xy-koordinaateilla. Rastin sijainti muuttuu riippuen valitusta ruudukkotyylistä ja prototyypissä ruudukkoja on tällä hetkellä kahta erilaista: neli- ja kuusikulmio. Viimeisenä muuttujana on Texture2D, mihin syötetään haluttu maastokartta, jossa pelaajat suunnistavat.

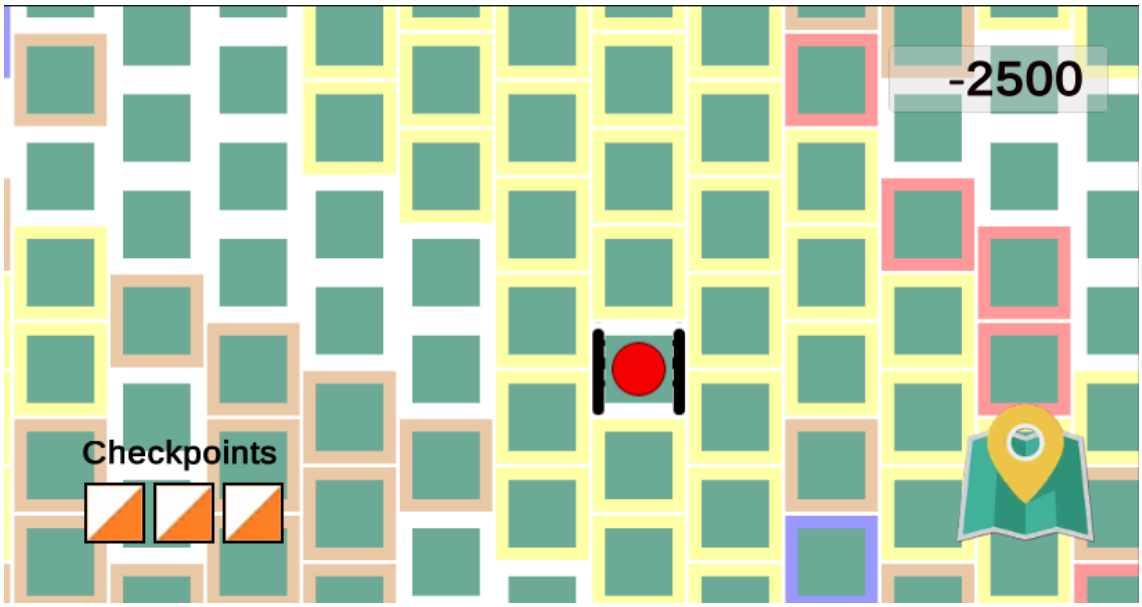
Taulukko 1. Level-luokka, josta luodaan scribtable object.

Level(level, stage)
+ int public: level
+ int public stage
+ struct public: CheckPoints
+ Texture2D public: Map

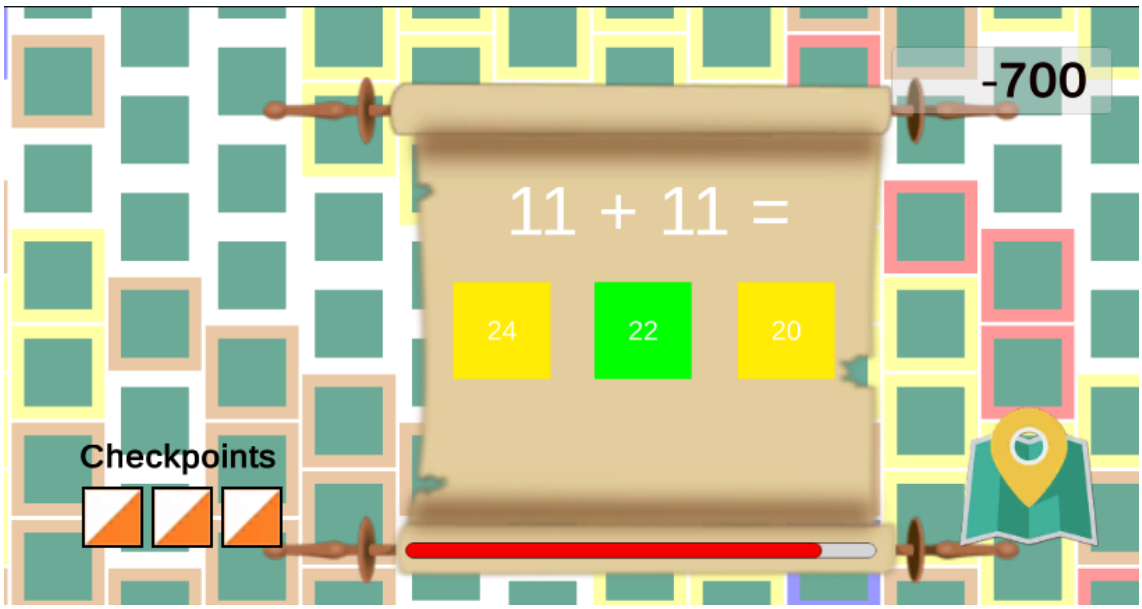
Ennen pelin alkamista pelaajille GameManager-script ottaa vastaan serverin lähettämät tiedot level-luokan scriptable objectista ja valitsee kovakoodatun ruudukkomallin, jota pelaajat eivät voi prototyyppivaiheessa valita. Kovakoodattua ruudukkoa ruvetaan rakentamaan HexGrid- tai SquareGrid-scriptissä, jotka on periytetty NodeManager-scriptistä. Ruudukko rakentuu ruuduista, joista jokainen omaa useita muuttujia ja komponentteja, kuten ruudun koko, koordinaatit ruudukossa sekä maastoa kuvaava väri. Kun ruudukko on valmis ja ruudut omaavat tarvittavat tiedot, voi peli alkaa.

Pelaajat aloittavat pelin lähinäkymästä, jossa näkyy pelaaja, maali sekä eri värisiä ruutuja (Kuva 2). Pelaajat liikkuvat ruudukolla klikkaamalla omaa pelaajahahmoa ympäröiviä ruutuja. Klikatessa ruutua pelaajaan näytölle ilmestyy matemaattinen kysymys, johon vastausvaihtoehtoja on kolmesta kuuteen riippuen maaston vaikeudesta (Kuva 3). Vastatessa oikein pelaaja jää klikkaamaansa ruutuun, kun taas väärin vastatessaan joutuu palaamaan takaisin. Ensin pelaajaan pitää kuitenkin tietää mihin suuntaan edetä, joten hänen täytyy suunnistaa käyttäen maastokarttaa (Kuva 4). Maastokartassa näkyvät pelaajien sijainti, maali sekä kaikki rastit. Tämä näkymä on saavutettu toisella kameralla, mikä renderöi kaiken paitsi ruudut, jotka eivät ole rasteja tai aloitusruutu. Pelaajan vastatessa oikein ruudussa, jossa rasti sijaitsee hänen käyttöliittymänäkymään merkataan rastin kohdalle käytymerkki. Kun pelaaja on käynyt kaikilla rasteilla ja palannut maaliin ensimmäisenä, on hän voittanut pelin. PlayerManager-script pitää yllä näitä muuttujia ja päivittää pelaajan sijaintia muille pelaajille serverin kautta.

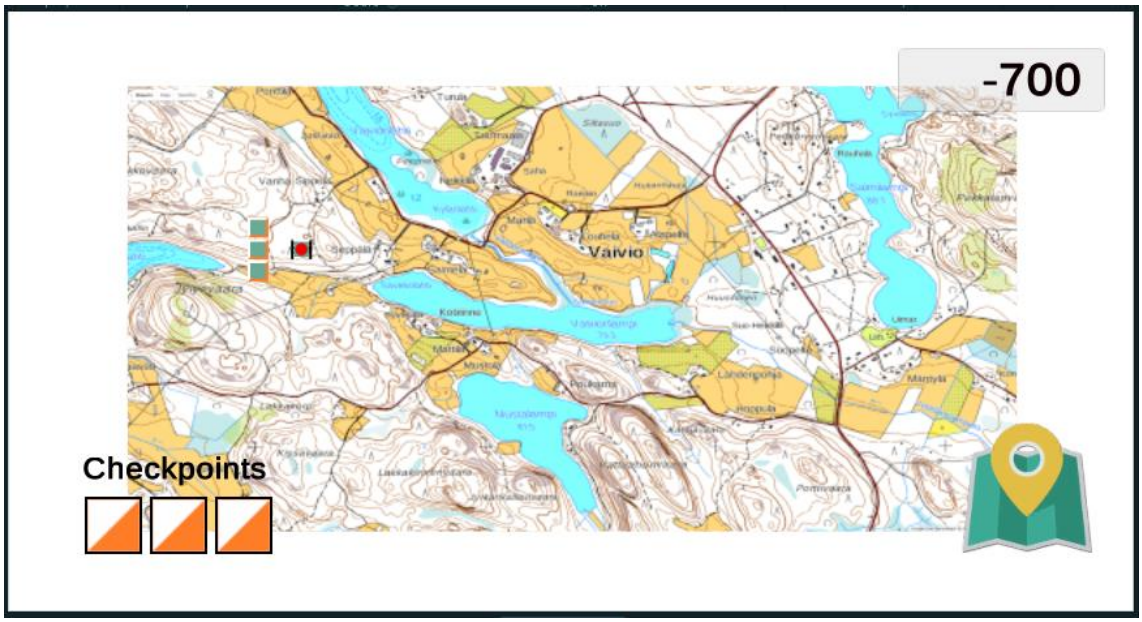
Pelin edetessä pelaajat saavat pisteitä oikeista vastauksista ja menettävät pisteitä ajan kuluessa. Pisteiden laskulla ei tällä hetkellä ole käytännön merkitystä, sillä voittajana toimii pistemäärästä huolimatta ensimmäinen maaliin päässyt pelaajaa.



Kuva 2. Peli-scene, ruudukon lähinäkymä.



Kuva 3. Peli-scene, kysymyspaneeli.



Kuva 4. Peli-scene, kartta- ja suunnistusnäkymä.

2.3 Refaktoroinnin ja dokumentoinnin tarkoitus

Refaktoroinnilla tarkoitetaan lähdekoodin uudelleen kirjoittamista sovellusarkkitehtuurin edistämiseksi. Koodin uudelleen kirjoittaminen ei tulisi saada vaikuttaa komponenttien rajapintoihin, vaan pelkästään sisäiseen rakenteeseen siten, että se palauttaa samat määritetyt asiat kuten aikaisemmin (Fowler 2017.) Rajapintojen muuttaminen muuttaisi toiminnallisuuden käyttäytymistä kokonaan. Mikään ei kuitenkaan takaa tällä hetkellä, että toiminnallisuudet pysyvät samana, kun uudelleen kirjoittaminen aloitetaan. Testien luominen tai olemassa olevien hyödyntäminen takaa sen, että haluttu alku ja lopputulos pysyvät samana (Fowler 2017). Testit helpottavat myös ilmentämään sen, missä mahdollinen ongelma koodin toteutuksessa ilmenee. Mitä kattavammat testit, sen helpompi virhekohtia on paikantaa. Tässä projektissa lähden kuitenkin tekemään yksikkötestejä vasta kehityksen lopussa, jos aikaa on vielä jäljellä. Kehitykseen on vain rajallisesti aikaa, sillä suunnittelu, dokumentaatio ja raportointi ovat aikaa vieviä prosesseja. Testejä voi jatkossa laajentaa halutessaan komponenteista luodun dokumentaation pohjalta.

Refaktoroinnin päällimmäisenä tarkoituksena on eheyttää olemassa olevaa koodia. On monia syitä sille, miksi koodia pitäisi eheyttää, kuten koodin rakennetta ei ole suunniteltu tarkasti, kokonaiskuvan puuttuminen, koodi on vaikeasti luettavaa, ohjelmassa ilmenee paljon samaa koodia, tai koodia on vaikea muokata. Syitä voi olla monia, mutta omassa tapauksessa uudelleen kirjoittamisen päällimmäinen syy on kokemattomuus ja sen tuomat ongelmat. Kokemattomana ohjelmistokehittäjänä katse ohjelman tulevaisuuteen on hyvin rajoittunut kuten pelin prototyyppiä tehdessä kävi ilmi. Vaikka kyseessä onkin vain prototyyppi, olisi sitä kirjoittaessa voinut miettiä mahdollisia ratkaisuja ja pitää katse tulevassa. Käytännön tasolla refaktoroinnin tarkoituksena on parantaa ohjelmiston arkkitehtuurillista rakennetta jäsentämällä komponentit, luokat ja muuttujat niin, että koodista tulee ymmärrettävä kokonaisuus (Fowler 2014). Prototyypissä ilmenevät toiminnallisuudet on tarkoitus säilyttää, mutta lähdekoodi, jolla toiminnallisuudet toimivat tulee kirjoittaa uudelleen tarpeen vaatiessa. Refaktorointi on iteraatioihin perustuva prosessi ja sitä tarvitaan, kun projektiin lisätään paljon toiminnallisuuksiin kohdistuvia muutoksia (Martin 2009). Lähtökohtana projektissa on eristää sekalaisesta koodista viisi komponenttia, joita pystytään mahdollisuuksien mukaan hyödyntämään jatkossa.

Tällä hetkellä komponentit ovat vahvasti riippuvaisia toisistaan epämääräisillä aksessoreilla ja muuttujilla, jotka voitaisiin eristää toisistaan, jotta komponenteista tulisi riippumattomia muuten kuin tarkoin määritellyillä rajapinnoilla. Rajapinnoista syötetään ja tulostetaan komponenttien toiminnallisuuksia kuten pelaajan liikkeet ja sijainti kartalla. Aksessorit ja metodit ovat yksi selkeimmistä refaktoroinnin kohteista, mutta aina refaktoroinnin kohteet eivät ole niin selkeitä. Useasti huonosti suunnitellut metodit saattavat sisältää jopa satoja rivejä koodia, jolloin metodin pilkkominen pienempiin osiin on tähdellistä. Komponentit on syytä suunnitella tarkoin ennen niiden refaktorointia, jotta niiden toiminnallisuudet vastaavat vaatimuksia. Prototyypin hyötynä onkin se, että kehittäjällä on selvästi mielessään se, kuinka komponenttien tulisi toimia. Tätä tietoa pystyy jäsentämään erilaisiin rakennetta ja käyttäytymistä kuvaaviin UML-kaavioihin.

UML-kaaviot ovat suunnittelussa ja kehityksessä se punainen lanka, jota kehitystiimin tulee seurata varsinkin olio-ohjelmointiin perustavassa koodissa. Se varmistaa, että jokaisella on selvä näkemys sovelluksen toiminnallisuuksista sekä teknillisestä toteutuksesta. Etenkin luokat ja rajapinnat ovat tärkeä dokumentoida selkeästi, sillä muut kehittäjät käyttävät komponentteja niiden perusteella. Dokumentointi on aikaa vievä prosessi, mutta se on edellytys, jotta kehitystiimi osaa toimia yhtenäisesti. Yksilöprojektit jäävät helposti dokumentoimattomiksi, sillä tietoa ei ole välttämätöntä jakaa kenenkään kanssa. Luokkien yhteydet, rajapinnat sekä toiminnot voivat olla selkeänä omassa mielessä, mutta dokumentointi toimii myös eräänlaisena vakuutuksena projektille, asiakkaalle sekä yrityksille. Käyttötapauksia kuvaavat kaaviot varmistavat, että toiminnallisuudet tulevat toimimaan niin kuin on toivottu. Tämä vähentää väärinkäsityksiä ja antaa selvän kuvan molemmille osapuolille siitä, kuinka toimintojen tulee toimia. Yrityksellisiä muutoksia saattaa myös tapahtua sisäisistä tai ulkoisista syistä ja työntekijät voivat mahdollisesti vaihtua. Dokumentoinnin myötä uudet kehittäjät voivat jatkaa projektia vaivattomammin eikä mahdollisia kehitykseen liittyviä virheitä tapahdu niin helposti. Hyvä ja selkeä dokumentointi ei ole vain aikaa vievä pakollinen paha, vaan se voi nostaa yrityksen kilpailukykyä ja luotettavuutta.

3 Työkalut ja niiden valinta

3.1 Unity 2018

Peli toteutetaan Unity-pelimoottorilla, jonka valinta tuli toimeksiantajan puolesta. Unityn versioon suositeltiin aina uusinta mahdollista, joten aloitin prototyypin rakentamisen Unity 2017 -pelimoottorin versiolla. Unityn päivittyessä siirryin uusimpaan Unity 2018 -versioon. Nykyisessä versiossa on ominaisuus, jonka avulla pystyy luomaan erilaisia ruudukoita haluamistaan elementeistä. Prototyypin aikana en kuitenkaan lähtenyt testaamaan uutta ominaisuutta vaan pysyttelin vielä luomassani ruudukon generointiin tarkoitetussa NodeManager-komponentissa,

vaikka ominaisuus onkin tarkoitettu erityisesti vaivattomaan prototyypin luomiseen (Duffy 2018). Jatkokehitystä ajatellen Unity 2018 voi luoda helpomman ja optimaalisemman tavan luoda ruudukoita, joten siirtymä oli kannattava tehdä. Se mahdollistaa myös helpon tavan luoda ruudukkopohjaisia karttoja ilman erillisiä maastokarttoja.

3.2 Visual Studio 2017

Visual Studion käyttäminen Unityn ohella on selvä valinta, sillä Unity tukee täysin Visual Studion käyttämistä ja sen pystyy suoraan integroimaan Unityyn. Se mahdollistaa Think smart -automaattisyötön, syntaksin korostuksen sekä monia muita hyödyllisiä ominaisuuksia, jotta koodin kirjoittaminen onnistuisi vaivattomasti (Unity Documentation 2018). Visual Studiota käytetään myös pääsääntöisenä kehitysympäristönä Karelia-ammattikorkeakoulussa, joten sen käyttäminen on entuudestaan tuttua. Visual Studio 2017 -kehitysympäristöstä on kolme erilaista versiota: Community, Professional ja Enterprise, joista itse käytän Visual Studio 2017 Community -versiota. Community on ilmainen, mutta se jättää pois maksullisten versioiden hyödyllisiä ominaisuuksia. Professional-versiolla pystyy generoimaan valmiista koodista visuaalisia malleja luokkien yhteyksistä ja rajapinnoista (Microsoft 2016). Koodin visualisointi on tärkeää, mutta siitä on enemmän hyötyä eheään koodipohjaan, jota tämä projekti ei sisällä toteuttamisen hetkellä. Tämän takia ominaisuus ei ole tarpeellinen tässä projektissa.

3.3 Google Drive

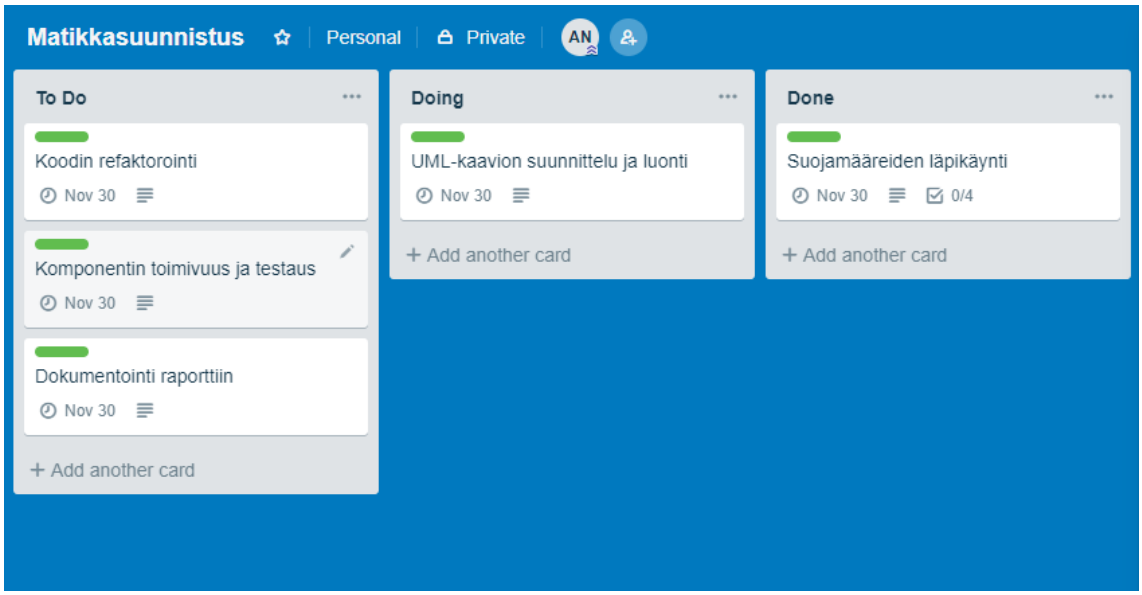
Google Drive on pilvipalvelu, joka tarjoaa 15 gigatavua tallennustilaa. Pilvessä toimivaa tallennustilaa kannattaa käyttää varmuuskopiontiin, jota hyödynnän kyseisessä tarkoituksessa. Kuviot, taulukot ja kuvakaappaukset on järkevää varmuuskopioida. Se kuuluu osaksi projektin riskienhallintaa ja madaltaa mahdollisia ulkoisia ja sisäisiä ongelmia tiedostoja kohtaan.

3.4 Draw.io

Draw.io on verkossa toimiva ja ilmainen kuvaajien piirtotyökalu. Se sisältää monenlaisia valmiita kuviomalleja, joita yleisesti käytetään graafisessa dokumentaatiossa. Tarjonnasta hyödynnän UML-mallinnuskielen kuvioita, joilla pystyy helposti kuvaamaan sovellusarkkitehtuurin toimintaa ja rakennetta visuaalisesti. Palvelu tarjoaa myös kuvaajien versionhallintaa, jos se on linkitettynä Google Driveen (Draw.io 2017.)

3.5 Trello

Trello on verkossa toimiva visuaalinen projektinhallintatyökalu, mikä auttaa jäsentämään työtehtäviä joustavasti pienempiin osiin. Sen tarkoituksena on kohdentaa työntekoa niihin yksityiskohtiin, joilla on merkitystä (Trello 2018). Työkalun avulla aion jäsentää työntekoa ketterien kehitysmenetelmien mukaan neljään sprinttiin, joissa luon komponentteja osa osalta valmiiksi. Jäsentely auttaa organisoimaan työntekoa ja antaa paremman kokonaiskuvan projektin kulusta (Kuva 5). Kouluprojekteissa olemme käyttäneet Jiraa, mutta päädyin valitsemaan Trellon projektinhallintaa varten, sillä en ole käyttänyt sitä ennen ja haluan tutustua erilaisiin projektinhallintatyökaluihin.



Kuva 5. Trello-projektinhallintatyökalun board-näkymä.

3.6 Versionhallinta

Projektin versiohallinnassa käytän itselleni tuttua Git-versionhallintaohjelmistoa, jota hyödynnän koulussa opitulla SourceTree-ohjelmalla. SourceTree mahdollistaa Gitin käyttämisen graafisella käyttöliittymällä. Graafinen käyttöliittymä helpottaa ymmärtämään versionhallintaa ja se visualisoi versioiden haarat selkeiksi janoiksi. Normaalisti Git-versionhallintaa käytetään komentorivistä erillisillä komennoilla, jolloin projektia voi olla vaikeampi hahmottaa, joten graafisten käyttöliittymien hyödyntäminen on helppo ja vaivaton ratkaisu ongelmaan. Versionhallinta tarvitsee myös tallennuspaikan eli repositoryn. Repositorynä käytän GitLab-verkkopalvelua, joka on ilmainen pienimmissä projekteissa. Monia vastaavia verkkopalveluja on tarjolla, kuten Bitbucket ja GitHub, mutta olen todennut GitLabin olevan käytettävyydeltään paras vaihtoehto. Versionhallinta ja sen käyttäminen oli oma henkilökohtainen valinta, jolla vähennän mahdollisten tiedostoihin kohdistuvien riskien määrää.

3.7 Mitä on hyvä koodi?

Ennen suurempaa suunnittelua on tärkeää hahmottaa, mitä on hyvä koodi. Tähän ei liene yhtä suoraa vastausta ja monelle aihe on subjektiivinen. Robert C. Martin antaa oman ammattimaisen näkökulman 2009 julkaistussa kirjassaan Clean Code. Martin painottaa kirjassaan, että on tärkeää seurata yhteisiä sopimuksia ja yleisiä käytänteitä, millä välttää jo valtaosan väärinymmärryksistä koodia luettaessa. Epäselvä ja huono nimeäminen on myös yksi harhaanjohtavimmista elementeistä huonossa koodissa. Tämän takia selkeä ja tarkoituksen mukainen nimeäminen on tärkeää. Suunnittellussa päätetyt nimet voivat kuitenkin kehityksen aikana menettää niille suunniteltua merkitystä, joten metodien ja muuttujien nimiä tulee tarpeen mukaan refaktoroida (Martin 2009). Suunnittellussa ja toteutuksessa tulen painottamaan tarkoituksen mukaista nimeämistä sekä yleisiä käytänteitä niin toimeksiantajan kuin Unity-kehitysyhteisön puolelta. Martinilla on paljon sanottavaa koodista ja aion referoida häntä omissa päätöksissäni kehityksen ja suunnittelun matkalla.

3.8 Optimointi

Prototyyppivaiheessa koodin optimointi ei ole aina ensimmäisellä prioriteetilla, mutta jatkokehitykseen kelpaavan pelin tulee löytää mahdollisimman optimaalisia ratkaisuja. Nämä ratkaisut syntyvät kehityksen myötä, sillä kaikkeen löytyy monia erilaisia ratkaisuja, jotkin ovat toisia optimaalisempia kuin toiset. Unity kuitenkin tarjoaa tähän vertailuun helpon ja visuaalisen Profiler-työkalun.

Profiler-työkalulla pystyy seuraamaan muun muassa koodin suorituksen kestoa, prosessorin, näytönohjaimen ja muistin käyttöä sekä pelimaailman renderöintiä ja audiota (Unity 2019d.) Oheista dataa pystyy seuraamaan aikajanalta, josta voi nähdä suorituksessa tapahtuneita piikkejä suorituskyvyssä ja näin selvittää mistä suorituskyvyn tippuminen johtuu. Tässä projektissa on hyvä pitää silmällä suorituskykyä, sillä pelikartat koostuvat mahdollisesti tuhansista peliobjekteista. Jos jokainen peliobjekti lähettää yhtä aikaa dataa voi suorituskyky tippua merkittävästi.

4 Toteutus

4.1 Perusluokat Unityssä

Dokumentoitavat komponentit ja niitä kuvaavat luokkakaaviot tulevat rakentumaan Unitylle tyypillisellä perintähierarkialla. Käytännössä se tarkoittaa, että kaikki peliobjektit Unityssä periytetään Object-luokasta, joka määrittää objektin perusmuuttujia kuten nimi, muuttamisen oikeudet sekä monia erilaisia perusmetodeja, kuten objektin instantiointi ja tuhoaminen (Unity Documentation 2018). Kaikkien luokkien ei kuitenkaan ole pakko olla periytettynä object-luokasta, jolloin luokka toimii riippumattomana peliobjekteista. Jotta objektit pystyvät omaamaan Unityn komponentteja, tulee luokan vielä periä MonoBehaviour-luokka. Komponentit mahdollistavat suuren määrän erilaisia toimintoja fysiikan mallinnuksesta objektin ulkomuotoon. Jos peliobjekti tarvitsee lähettää palvelimille tietoa, tulee peliobjekti periyttää NetworkBehaviourista MonoBehaviourin sijaan. Luokka on pääsääntöisesti samanlainen, mutta se omaa myös datan siirtomahdollisuuksia palvelimille.

4.2 Plugin-valintojen tarkastelu

Peli hyödyntää tällä hetkellä kahta eri pluginia: Lobby- ja Timing-pluginia, joista Lobby-plugin on täysin käytössä. Timing-plugin tulee käyttöön jatkokehityksen aikana. Käytän molemmista ilmaisversioita, jotka ovat saatavilla Unityn Asset Storesta ja niitä saa hyödyntää kaikissa projekteissa ehdoitta.

Lobby-plugin on jo valmiiksi rakennettu käyttöliittymä palvelimen luomiseen ja liittymiseen. Se myös mahdollistaa helpon datan siirtämisen lobby-scenestä pelisceneen. Plugin on järkevä säilyttää projektissa, sillä sen luominen itse veisi muusta kehityksestä aikaa.

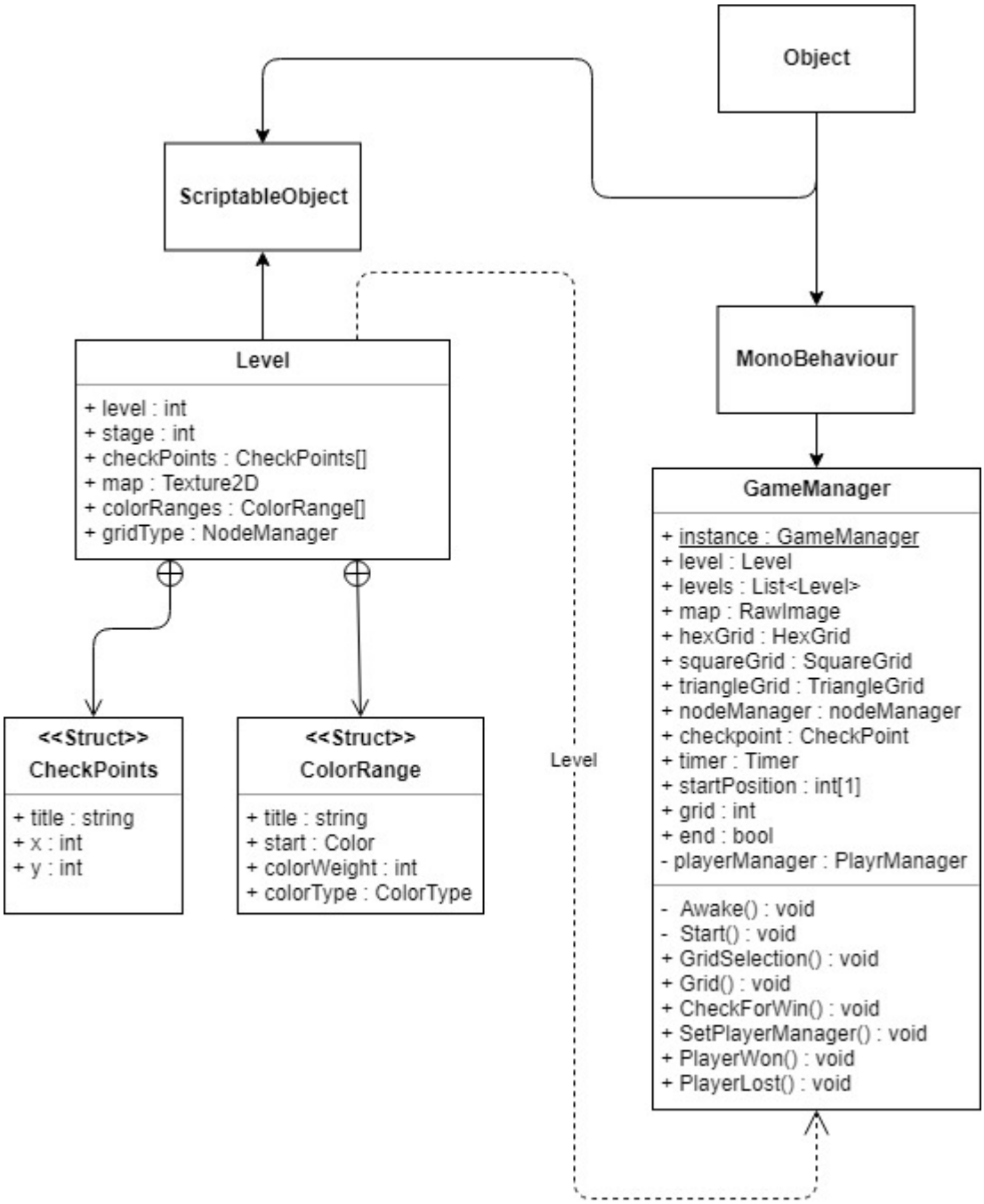
Timing-plugin korvaa Unityn oman Coroutine-metodin. Coroutine-metodien tarkoituksena on suorittaa metodin sisältö inkrementaalisesti, kunnes metodi saavuttaa tietyn pisteen. Unityn sisällyttämä Coroutine-metodi on hyvin raskas suorittaa ja monta tällaista kutsua päällekkäin saattaa viedä liikaa suorituskykyä varsinkin mobiilialustoilla. Timing-pluginin Coroutine-metodi suorittaa nämä kutsut yli kymmenen kertaa nopeammin (Trinary Software 2018.) Plugin kannattaa siis sisällyttää lähestulkoon aina Unity-projekteihin. Maksullinen versio pluginista tarjoaa perustoimintojen lisäksi monia hyödyllisiä ominaisuuksia, mutta tässä projektissa tullaan tarvitsemaan vain pluginin perusominaisuuksia.

4.3 GameManager

GameManager on pelille tärkein komponentti, jota yleisimmin käytetään singleton-mallina. Singleton-mallin tarkoituksena on varmistaa, että sovelluksessa on vain yksi staattinen ilmentymä kyseisestä luokasta. Kun ilmentymään liittää pelille tarpeelliset oliot, pystyy näitä olioita kutsumaan ilmentymän kautta (Source Making 2018). GameManagerit ovat yleensä uniikkeja omille peleilleen, joten niiden uudelleen käytättämistä on käytännössä turhaa miettiä, elleivät pelin toiminnallisuudet ja komponentit ole samanlaisia seuraavassa projektissa, kuten jatko-osa pelille. On kuitenkin tärkeää, että dokumentaatiosta käy ilmi se, mitä GameManager tarvitsee toimiakseen. GameManager saattaa olla hyvin eläväinen luokka läpi pelin kehityksen, sillä pelille yleisesti tarpeelliset ominaisuudet voivat muuttua sen aikana. Alussa kannattaa keskittyä peliä ylläpitäviin ominaisuuksiin, jotka ilmenevät useasti. Tässä pelissä sellaisia ominaisuuksia ovat muun muassa tasojen ja kenttien valinta, ruudukkomallit, pelaaja sekä pelin tilannetta kuvaavat voitto- ja häviöehdot.

Prototyyppivaiheessa oleva GameManager ei kuitenkaan ota vastaan näitä kaikkia ominaisuuksia, vaan ne johdetaan suoraan komponentista komponenttiin, jolloin riippuvuussuhteiden verkko kasvaa suureksi. GameManagerin refaktoroinnin tarkoituksena on ohjata nämä riippuvuudet GameManagerin kautta, jotta muilla komponenteilla olisi selkeämmät rajapinnat.

Refaktorointia kuvaavassa GameManager-luokkakaaviossa käydään läpi Level-luokan ja GameManagerin välistä riippuvuussuhdetta (Kuvio 1). Luokkien välillä on silloin riippuvuussuhde, kun toisen luokan toiminnallisuus on riippuvainen toisesta luokasta, eikä se mahdollisesti toimisi ilman tätä riippuvuussuhdetta (UML-diagrams 2018). GameManager on riippuvainen Level-luokasta, sillä se tarvitsee luokasta luodun ilmentymän muuttujia, joita se jakaa eteenpäin pelin muille komponenteille. GameManager tarvitsee myös muita public-suojamääreen omaavia luokkia, mutta level-riippuvuussuhteella dokumentaatiossa halutaan painottaa sitä, että kentät ja niiden parametrit luodaan vielä manuaalisesti ja ne pitää lisätä levels-listaan.



Kuvio 1. GameManager- ja Level-luokan välinen riippuvuussuhde luokkakaaviossa refaktoroinnin jälkeen.

4.4 MathManager

Prototyyppivaiheessa MathManager tuottaa matemaattisia kysymyksiä, mutta niiden generoimiseen ei ole annettu erillisiä ehtoja. Uudelleen kirjoittamisen tarkoituksena on tarkentaa näitä ehtoja niin, että se kuvastaa suunnistukselle olennaista reitin valintaa.

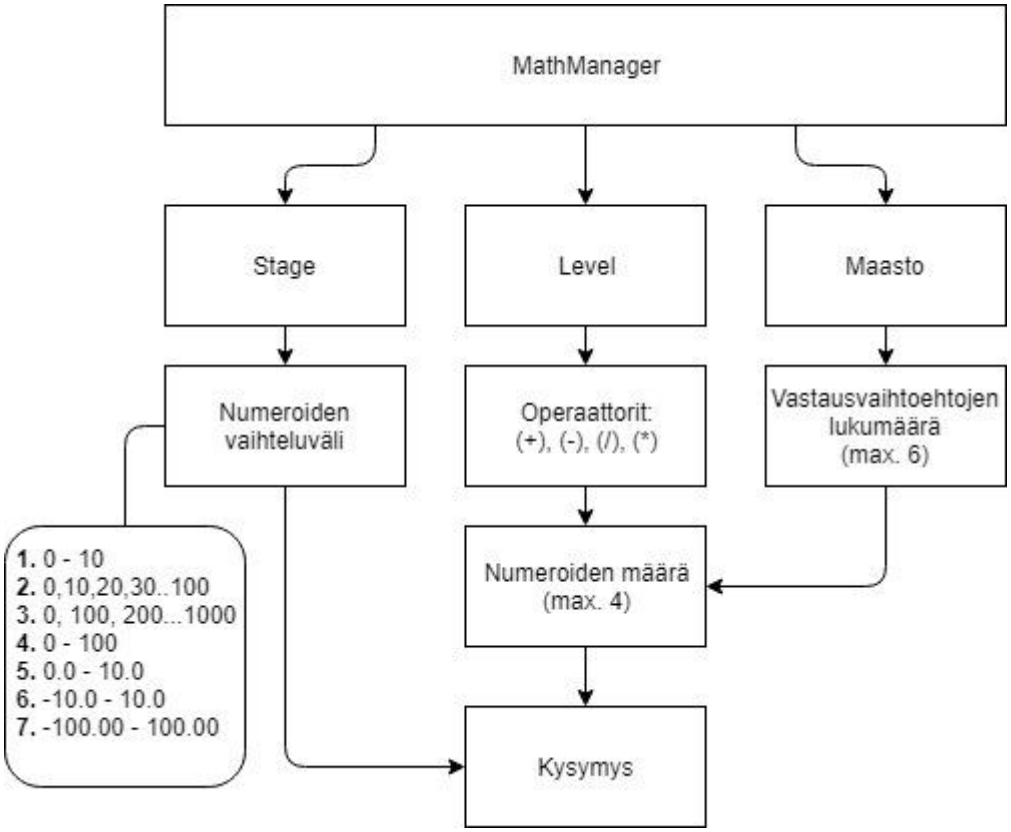
Ennen MathManagerin suurempaa suunnittelua eteen tulee kysymys: onko tämän pelin tarkoitus myös opettaa matematiikkaa suunnistuksen ohella? Kysymysten generoiminen ei ole teknisesti haastava toteuttaa, mutta niiden generoiminen ymmärtäen matematiikan didaktiikan vaatii jo kasvatustieteiden ammattilaisia.

Matemaatikko ja matemaattisten opetuspelien edelläkävijä Keith Devlin käy läpi 11 periaatetta ideaalista oppimisympäristöstä matematiikan oppimiseen, joissa hän painottaa ympäristön ja oppimisen suhdetta toisiinsa ja kuinka ne vaikuttavat oppimiseen. Pelissä on samoja piirteitä, joita Devlin käy läpi, kuten stimuloiva ympäristö, jossa oikealla vastauksella on positiivisia seurauksia, mutta pelin nopea kilpailutyylinen tapa pelata ei ole välttämättä paras mahdollinen opetuskeino (Devlin 2011). Suunnistusteemainen matematiikkapeli ei myöskään perustu tosielämän tilanteisiin, joten pelissä opittua saattaa olla vaikea soveltaa jatkossa. Pelin pääkohtana toimii suunnistus, joten matemaattisten kysymysten tulee enemmänkin olla helpohkoa aivojumppaa jo matematiikasta ymmärtävälle, kuin ohjastavaa opetusta uudelle oppijalle.

Kysymysten tulee kuitenkin antaa samalla tavalla haastetta kuin suunnistuksessa reitin valinta. Minkälainen kysymys on vaikea ja kenelle? Peliä on mahdoton lähteä yksilöimään, joten joitakin rajoja on vedettävä matemaattisten perustaitojen hallinnasta. Opetussuunnitelman mukaan perus aritmetiikkaa opetetaan ensimmäisellä ja toisella vuosiluokalla ja kolmannesta kuudenteen siirrytään opettelemaan algebraa, jossa jonkin nimittäjä voi olla tuntematon (Opetushallitus 2014.) Opetussuunnitelmaa voi hyvin käyttää ohjenuorana kysymysten haasteellisuudesta aloittaen ensin perusoperaatioista ja siirtyen lopulta muuttujan arvon selvittämiseen.

Kysymysten generoiminen tulee huomiomaan kolme eri muuttujaa: level, stage ja kartassa ilmenevä maasto. Komponenttiin kuuluvat myös Timer- ja Score-scriptit, joiden tehtävänä on pitää kirjaa vastausajasta ja pisteistä (Kuvio 2). Level määrittää sen, kuinka monesta luvusta ja operaattorista kysymys tulee muodostumaan; mitä suurempi level, sen suurempi varianssi. Stage vaikuttaa numeroiden vaihteluväliin samalla tavalla. Vaihteluväli muuttuu laajemmaksi mitä korkeampi stage on kyseessä. Kartan maasto vaikuttaa monivalintakysymyksien

vastausvaihtoehtojen lukumäärään. Kysymykset tulevat koostumaan näistä elementeistä ja ne vaihtuvat joka pelikerralla.

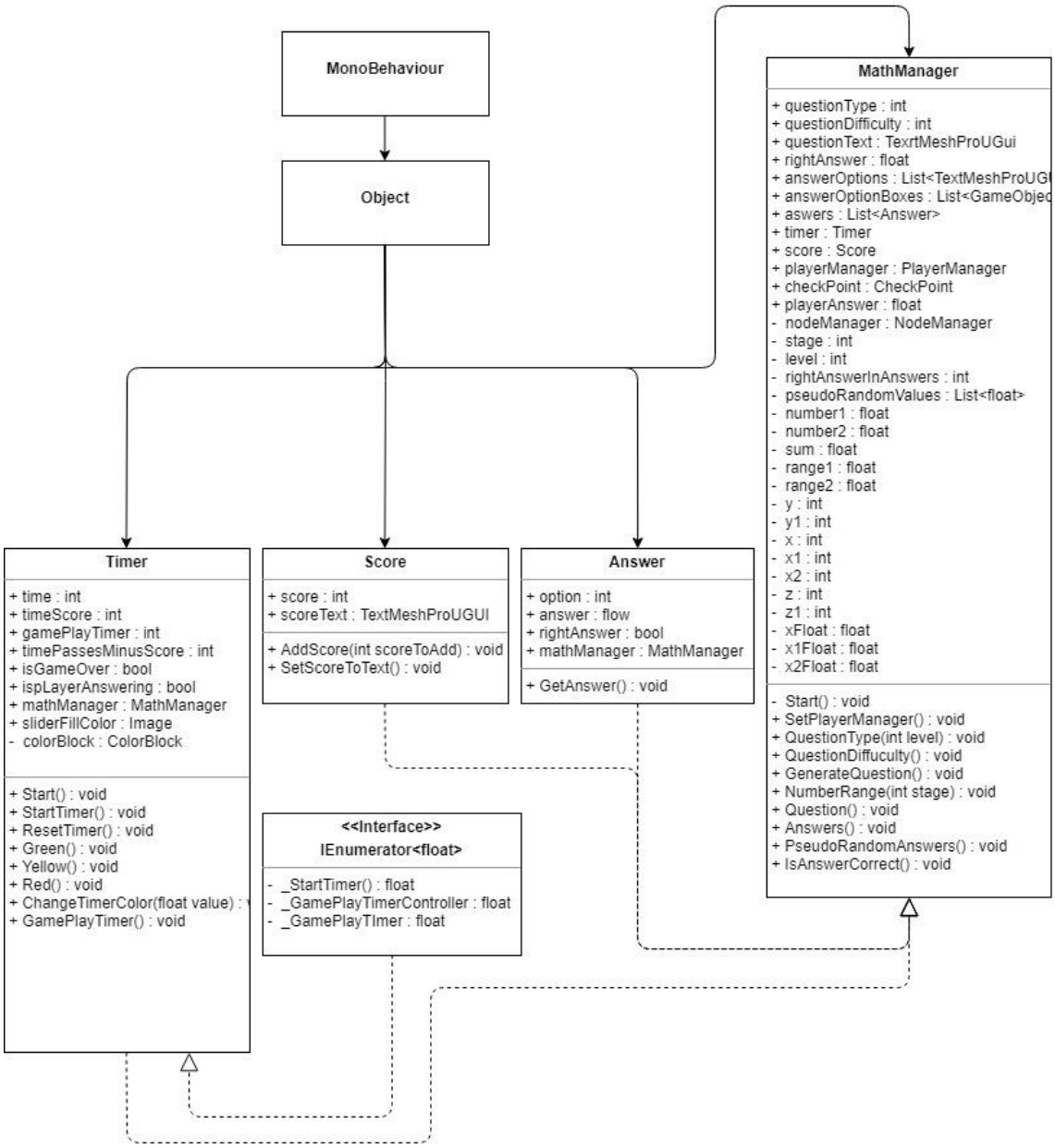


Kuvio 2. Levelin, stagen ja maaston vaikutus kysymysten generointiin.

Ensimmäisenä uudelleenkirjoittamisen kohteena MathManagerissa on suojamääreiden tarkastelu ja metodien pilkkominen pienempiin osiin luokan selkeyttämiseksi. Metodien pilkkominen tulee tehdä harkiten ja miettiä mitkä kokonaisuudet kuuluvat yhteen, ja jotta pilkkomisesta on hyötyä, on metodien nimet mietittävä mahdollisimman kuvaaviksi (Fowler 2017, 90). Kysymysten generointiin tarvitaan monia erilaisia muuttujia erilaisine tietotyyppeineen. Prototyypissä olin sijoittanut kaikki muuttujat heti luokan määrittelyn jälkeen, ja koska muuttujia käytetään pelkästään laskukaavoissa, niiden nimeäminen on myös haastavaa. Tällaisissa hankalissa tilanteissa on hyvä lähteä tarkastelemaan sitä, kuinka vastaavia ongelmia hoidetaan alalla. Unity-ohjelmoinnissa on ollut tapana sijoittaa muuttujat lähelle metodeja, joissa niitä hyödynnetään ja saavat näin paremman kontekstin (Leppänen 2018). Tämän tiedon varjolla lähdin asettamaan muuttujia niiden metodien läheisyyteen, joissa niitä käytetään. Tarkastelin myös, tarvitaanko muuttujaa muissa luokissa vai pelkästään MathManagerissa ja muunsin

niiden suojamääreitä tarvittaessa. Suojamääreitä on kolmea erilaista: public, private ja protected. Private ja protected -suojamääreiden tarkoituksena on eristää sellaisten muuttujien käytettävyyttä, joita ei ole tarkoitus käyttää luokan ulkopuolella. Public-suojamääreen tarkoituksena on taas näkyä myös muille luokille (Ohjelmointiputka 2011.) Public-suojamääreelliset metodit toimivatkin yleisesti luokkien rajapintoina.

Suunnittelemani MathManager-luokkakaavio on hyvin sekava muuttujien paljouden takia eikä siitä jää lukijalle paljoa käteen, mutta luokkakaavioita tarkastellessa on hyvä kiinnittää huomiota edellä mainittuihin public-, private- ja protected-suojamääreisiin (Kuvio 3). Luokkakaaviossa suojamääreitä esitetään (-), (#) ja (+) -merkeillä, joista (+) -merkki tarkoittaa public-suojamäärettä, (-) -merkki private-suojamäärettä ja (#) -merkki protected-suojamäärettä (Uml-diagrams 2018). Näistä vain public-suojamääre näkyy luokan ulkopuolelle. Muuttujia ja metodeja suunnitellessa onkin tärkeä miettiä, missä niitä käytetään ja tarvitsevatko muut luokat käyttöoikeuksia. Muuttujia ja metodeja eristämällä komponenttien rajapinnat selkiytyvät.



Kuvio 3. MathManager-komponentin luokkakaavio refaktoroinnin jälkeen.

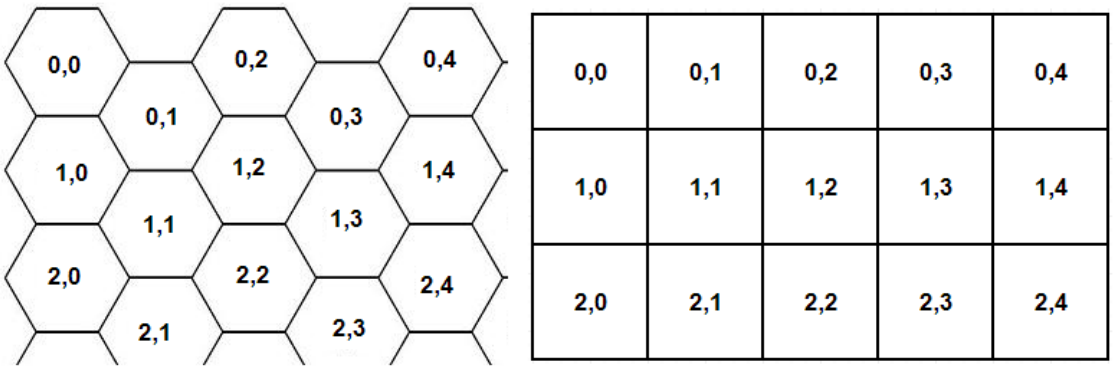
Kehityksen alussa kysymykset perustuvat aritmetiikkaan ja algebraan, mutta jatkokehityksen kannalta on hyvä miettiä myös sitä, kuinka muita matematiikan osa-alueita kuten geometria ja sanallisit kysymykset pystyy lisäämään peliin.

Lopputuloksena MathManagerista tuli suurehko komponentti, jonka toiminnallisuuksia on vaikeahko muuttaa ilman logiikan muuttamista. Komponentti generoi satunnaisia kysymyksiä kolmen eri muuttujan perusteella ja kysymysten luominen vaatii paljon numeroiden pyörittelyä riippuen näistä kolmesta muuttujasta.

Luokkana se kuitenkin selkeytyi metodien pilkkomisen ja suojamääreiden määrittämisen myötä. Ilman uudelleen kirjoittamista jatkokehitys olisi ollut melkein mahdotonta.

4.5 NodeManager

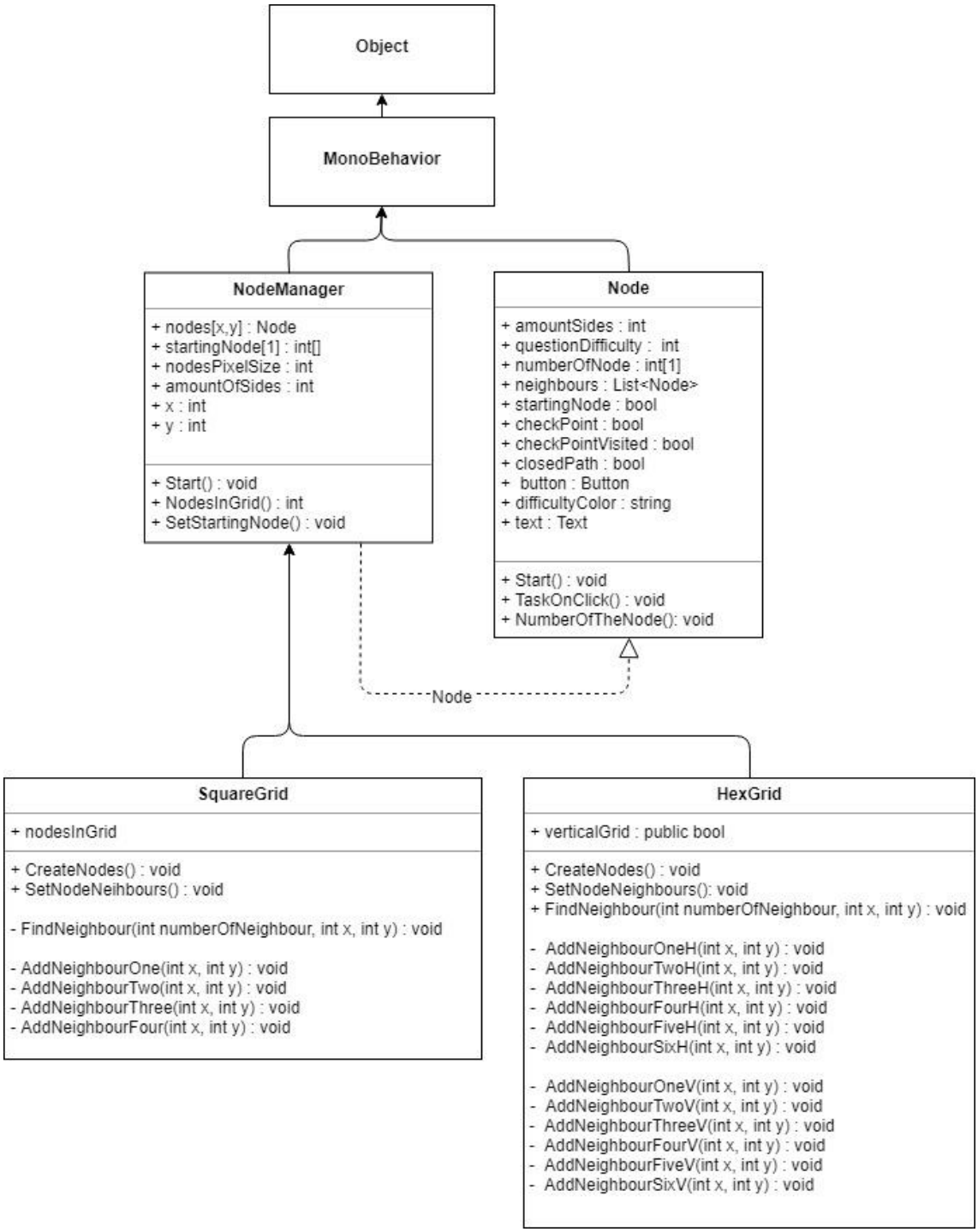
NodeManager-komponentin tarkoituksena on generoida ruudukoita Node-luokan peliobjekteista. Node-luokka sisältää muuttujia, jotka kuvastavat ruudun muotoa sekä omaa- ja mahdollisten naapurien sijaintia ruudukossa (Kuvio 5). NodeManager periytetään SquareGrid- ja HexGrid-luokkiin, joissa neli- ja kuusikulmioista koostuvat ruudukot rakentuvat. Rakennusvaiheessa ruudukon osat saavat sijainnin, koordinaatit sekä naapurit. Neli- ja kuusikulmioiden koordinaatit eroavat hieman toisistaan, mutta perusajatuksena on, että molempien origo sijaitsee ruudukon vasemmassa yläkulmassa (Kuvio 4). Normaalisti voisi ajatella, että origon tulisi sijaita vasemmassa alakulmassa ja koordinaatit kasvaisivat oikeaan yläkulmaan mentäessä, niin kuin prototyyppivaiheessa olinkin luonut. Refaktoroitaessa komponenttia päätin kuitenkin vaihtaa koordinaattien asettelua tutustuessani suosituimpiin Google Maps ja OpenStreetMap -karttapalveluihin. Molemmat palvelut luovat ruuduista koostuvat kartat aloittaen vasemmasta yläkulmasta ja numeroiden kasvavan oikeaan alakulmaan (Liedman 2014). Ruudukon koordinaatiston numeroinnilla ei ole projektille käytännön vaikutusta, mutta suosituimpia standardeja on hyvä mukailla, jotta kokonaisuus on nykypäivän standardien mukainen.



Kuvio 4. Kuusi- ja nelikulmio ruudukon koordinaatit refaktoroinnin jälkeen.

Komponenttia pystyy hyödyntämään missä tahansa projektissa, jossa ruudukon luontia tarvitaan, eikä se ole riippuvainen muista komponenteista. Ruutuihin ja niiden muuttujiin pystyy vaikuttamaan Node-luokasta ja ulkonäköön peliobjektista, johon scritpti on kiinnitettynä. Tällä hetkellä luokka sisältää suunnistukselle olennaisia muuttujien tietoja, mutta tiedot pystytään korvaamaan uudessa projektissa olennaisella datalla. Ruudukko luodaan CreateNodes-metodin avulla ja vaikka ruudukko luodaan vasta kutsun yhteydessä, siitä pystyy luomaan prefabin. Prefabien avulla peliobjekteja pystyy tallentamaan siten, että ne pitävät sisällään kaikki tarvittavat rakennuselementit (Unity 2019e). Tällöin ruudukkoa ei tarvitse generoida aina käynnistyksessä, ja ruutuihin pystyy tekemään manuaalisesti muutoksia.

Refaktorointi NodeManagerissa kohdistuu siihen, kuinka ruudukko luodaan Node-peliobjekteista ja miten luotua ruudukkoa hyödynnetään pelissä. Komponentissa on myös hyvä tarkastella muuttujien ja metodien suojamääreiden asettamista niin, että ne selkeyttävät ja määrittävät rajapintoja. Ruudukkojen luominen tapahtui suuressa metodissa, jota lähdin pilkkomaan pienempiin kokonaisuuksiin: CreateNodes, SetNodeNeighbours -metodeihin. GameManager kutsuu public-suojamääreen omaavia metodeja luodakseen ruudukon. Nämä metodit kutsuvat AddNeighbour- ja FindNeighbour-metodeja, joiden avulla ruudukon ruudut ovat tietoisia ympärillä olevista ruuduista (Kuva 10.)



Kuvio 5. NodeManager-komponentin luokkakaavio refaktoroinnin jälkeen.

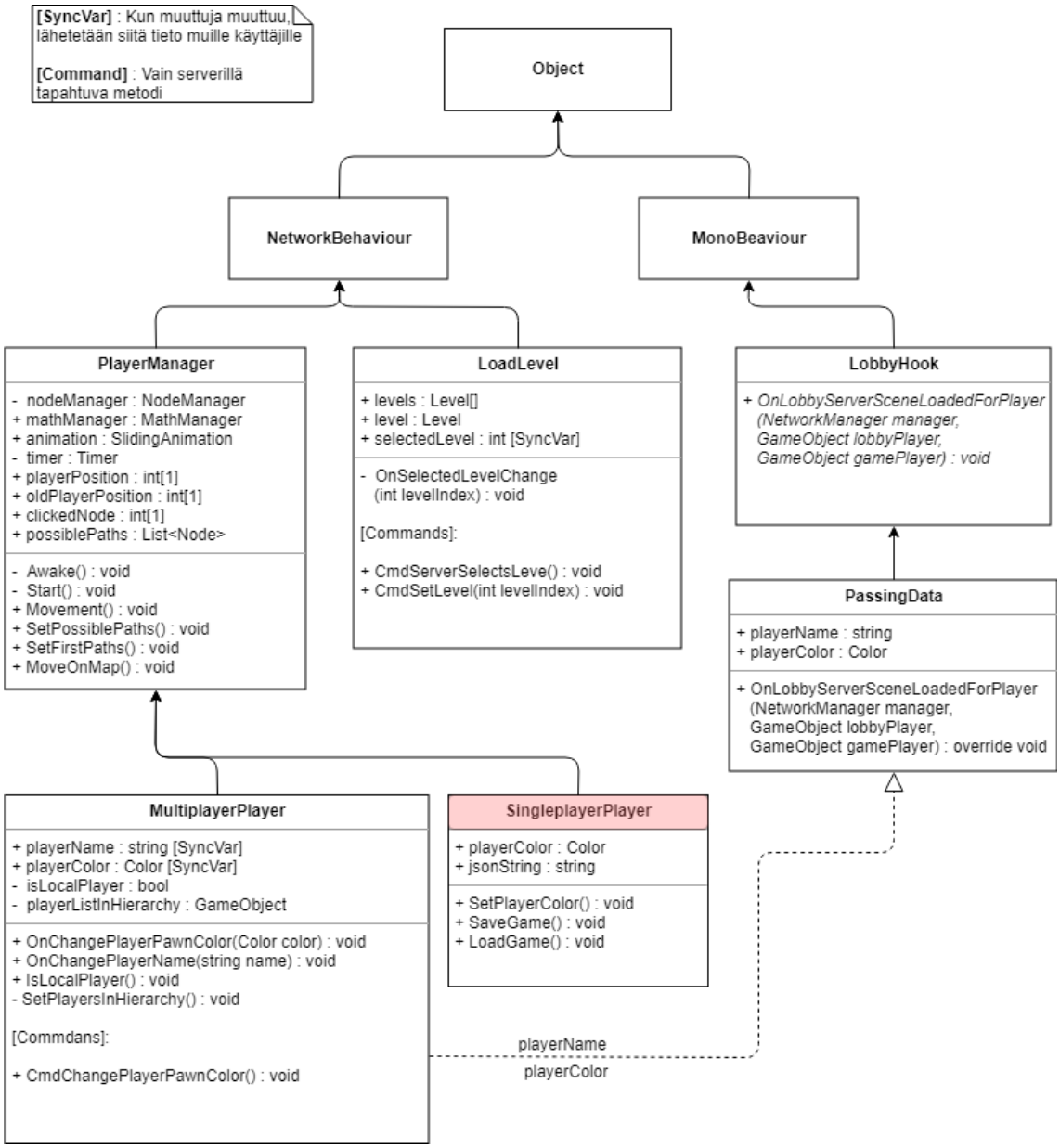
Refaktoroinnin jälkeen komponenttia pystyy helposti laajentamaan erilaisten ruudukoiden luontiin periyttämällä NodeManager-luokan ja luomalla ruutujen sijaintitietoja koskevat metodit. Kolmio on kuitenkin neli- ja kuusikulmion lisäksi ainoat säännölliset monikulmiot, joiden avulla pystyy luomaan ruudukkoja ilman ylimää-

räisiä aukkoja (Morphingtiling 2018). NodeManager ei kuitenkaan rajoita tekemästä aukottomia ruudukkoja, mutta ne tulee ottaa huomioon ruudun muodossa ja sijainnissa.

4.6 Player ja Networking

Pelaaja-komponentin tarkoituksena on olla tietoinen pelimaailmasta, joka tässä tapauksessa on karttakuvan pohjalta luotu ruudukko. Tietämällä missä ruudussa pelaajakomponentti on, pystyy komponentti myös tietämään, mihin se pystyy liikkumaan. Komponentti toimii pääosin PlayerManager-scriptin kautta, mikä pitää kirjaa näistä tiedoista. Moninpelin myötä ei kuittenkaan riitä, että vain paikallinen pelaaja tietää, missä hän sijaitsee kartalla, vaan tämän tiedon on liikuttava myös muille pelaajille (Kuvio 6.) UNet on Unityn tarjoama ratkaisu netin kautta toimivan moninpelin luomiseksi. Lähdin rakentamaan kyseisen palvelun avulla tiedonsiirtoa palvelimen kautta muille pelaajille. Refaktorointi keskittyi UNetin käyttöönottoon ja sen sovittamiseen PlayerManageriin niin, ettei sen läsnäolo vaikuta komponentin toimintoihin. Kehityksen loppuvaiheessa sain kuitenkin tietää, että UNet on vanhentunut. Palvelimet tulevat toimimaan seuraavan kolmen vuoden ajan eikä sen käyttöä enää suositella (House 2018a). PlayerManager ei kuitenkaan ole riippuvainen pelkästään Unetin käytöstä, ja se pystyy hyödyntämään Unityn seuraavaa Connected Games -ratkaisua (House 2018b). Tälle vaihdokselle kehityksessä ei ollut enää aikaa, mutta muutos on kannattava ratkaisu pelin tulevaisuutta ajatellen.

Yksinpeliominaisuuksia ilman palvelimen käyttöä ei tässä projektissa toteuteta, mutta luokkakaavio käy läpi tarpeellisia metodeja, joita yksinpelissä toimivan pelaajakomponentin pitäisi omata (Kuvio 6.) Yksinpeli ei ole muuten moninpelia erilaisempi, ellei yksinpeliin kehitetä jatkossa muunlaisia toiminnallisuuksia.



Kuvio 6. Player ja Networking -komponentin luokkakaavio.

4.7 MapReader

MapReader oli keskeneräisin komponentti prototyyppivaiheessa, joten komponentti ei niinkään vaadi uudelleen kirjoittamista vaan sen suoraa kehitystä. Maastokarttaa analysoivan komponentin on tarkoituksena lukea maastokartan värejä, joiden perusteella se osaa antaa värin pelissä olevan ruudukon ruuduille. Suunnistuksessa käytettävät kartat ja niiden värit vaihtelevat riippuen karttojen valmistajasta, mutta yhdenmukaisuuteen on kuitenkin pyrittävä (Suunnistusliitto 2017.)

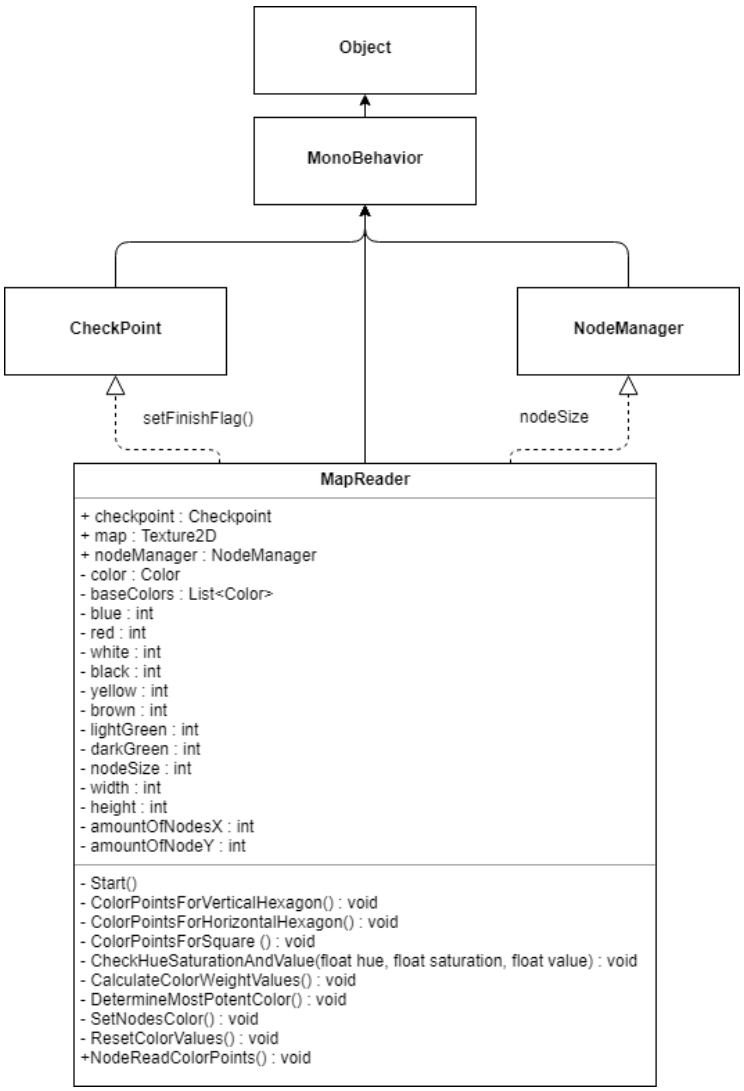
Suurimmassa osissa kartoista ilmenee seuraavia värejä: sininen, punainen, valkoinen, musta, keltainen, ruskea ja vihreä, josta on myös olemassa vaaleampaa ja tummempaa sävyä.

Värit kuvastavat suunnistettavan alueen maastoa, esteitä sekä kuljettavuutta. Sininen kuvastaa vesialuetta, keltainen avoimia alueita, vihreä kasvillisuutta ja ruskea maanpinnan muotoja. Mustaa ja harmaata taas käytetään muihin suunnistuksessa merkittäviin elementteihin, kuten polut, kivet, rakenteet tai jyrkänteet (Suunnistusliitto 2017.) Suunnistusliiton standardivärien lisäksi käytän punaista, valkoista ja tumman- ja vaaleanvihreää, joista punainen kuvastaa katuja, valkoinen helposti kuljettavaa tasaista maastoa ja vihreän sävyt helposti ja vaikeasti kuljettavaa metsää. Näiden maastoa kuvaavien värien perusteella lähden jatkokehittämään kartanlukukomponenttia ja nämä kyseiset värit ilmenevät myös Maastokartat.fi -sivustolla, josta maastokarttojen käyttäminen on ilmaista (Maastokartat.fi 2019). Kartanlukukomponenttiin voi lisätä rajapinnan muustakin karttapalvelusta, mutta tässä vaiheessa käytän pelkästään Maastokartat-sivuston kuvia. Kuvat liitetään GameManager-komponentissa olevaan Level -scriptable objectiin ja tällä tavoin eri kentissä voi olla eri karttoja.

Kartanlukukomponetti lukee karttakuvan pikseleitä ruuduista ja käy läpi koko pelin ruudukon. Ruudussa komponentti laskee, mitä värejä ruudulla on ja laskee värien painoarvojen mukaan sen, mikä ruudulla oleva väri on dominoivin (Kuvio 7.) Kehityksen edetessä jouduin muuttamaan pikseleiden lukua ja värien painoarvoa optimointisyistä. Ennen muutosta kartanlukukomponetti luki kaikki pikselit, joka vie liikaa aikaa isoilta kartoilta. Tämän takia lähdin vähentämään pikseleiden lukemista koko ruudusta kahteen poikkileikkaavaan janaan ruudun keskeltä leveys- ja pituussuunnassa. Tämän ratkaisun avulla aikaa kuluu vähemmän, mutta ruutua ei lueta niin tarkasti, jonka seurauksena ruudun väri ei välttämättä kuvaa tarkasti kartan maastoa. Kompensoidakseni tätä lähdin arvioimaan kartan värien painoarvoa uudestaan. Asetin vedelle suurimman painoarvon, sillä se on suurin vaikuttaja maastossa. Käytännössä, jos luetusta ruudusta kolmasosa on sinistä ja kaksi kolmasosaa vihreää, komponentti olettaa, että vesieste on suurempi tekijä kuin metsikkö, joten pelissä ruutu lasketaan vesiesteeksi.

MapReader-komponentti onnistui kaikista komponenteista parhaiten, sillä se oli viimeinen komponentti, jota lähdin kehittämään ja näin ollen olin omannut paljon

tietoutta siitä, kuinka komponenttia kannattaa lähteä rakentamaan. Komponentti tekee melko spesifisen asian, joten sen muita mahdollisia käyttötarkoituksia voi olla melko vähän. Komponenttia pystyy kuitenkin hyödyntämään, jos projektissa tarvitaan lukea kuvan värejä ja asettamaan luetun datan esimerkiksi väriruuduk-kona. Värien painoarvoa voi muuttaa sen mukaan, mitä värejä kuvasta halutaan painottaa.



Kuvio 7. MapReader-komponentin luokkakaavio refaktoroinnin jälkeen.

5 Pohdinta

5.1 Ongelmien kohtaaminen

Kokemattomana kehittäjän kohtasin niin suuria kuin pieniä ongelmia kehityksen aikana. Suunnitelmat eivät kulkeneet aina käsi kädessä kehityksen kanssa ja osa suunnitelluista ratkaisuista olivat käyttökelvottomia, epäselviä tai eivät palvelleet tarpeeksi hyvin komponentin käyttötarkoitusta. Suurin ongelmista liittyi kehityksen tarkoitukseen sekä motivaatioon.

Kehityksen edetessä ajatus komponenttien uudelleen käytettävyydestä alkoi miettyttämään. Käyttötarkoitukset luoduille komponenteille ovat hyvin kapeat ja niiden rajapinnat spesifisiä. Motivaatio komponenttien uudelleen käytettävyydestä hiipui kehityksen aikana, mutta en antanut sen haitata päämäärää luoda pelille eheämpää ja dokumentoitua sovellusarkkitehtuuria.

Toisena ongelmana kohtasin ajan käytön. Kehityksen ohella sain työpaikan backend -kehittäjänä, mikä sekoitti opinnäytetyöhön paneutumista. Mitään kiirettä opinnäytetyöllä ei ollut omasta tai toimeksiantajan puolelta, mutta kehityksen, dokumentoinnin ja raportoinnin aikana se toi paljon ylimääräistä stressiä elämään. Stressin myötä kehityksen aikataulua oli venytettävä, mikä taas pirstaloitti opinnäytetyön tekemisen yhteen päivään viikosta.

En kuitenkaan joutunut olemaan stressin ja motivaation puutteen kanssa yksin. Tukena toimi työyhteisö, koulu sekä perhe. Olen myös saanut ymmärrystä, apua ja tukea monesta eri suunnasta, jos sitä olen tarvinnut. Tuen avulla olen saanut lisää voimaa jatkaa kehitystä.

5.2 Kuinka olisin voinut suunnitella paremmin

Yksi ongelmiin johtavista asioista oli kokemattomuus sovelluskehittäjänä. Tämä toiminnallinen opinnäytetyö on ollut minulle suurin kehitysprojekti, jota olen päässyt tekemään. Suunnittelu ja kehitys ovat välillä tuntuneet huojuvalta korttitalolta, mutta ajan edetessä olen myös oppinut pitämään näistä huterista langoista kiinni

sekä vahvistamaan niitä uusilla tiedoilla ja taidoilla. Epäonnistumisen ymmärtäminen vaatii ymmärryksen siitä, mitä olisi pitänyt tehdä toisin. Tällaisia asioita olivat muun muassa ajan käytön hallinta, työkalujen valinta ja niiden vaihtaminen sekä tarkemmat määrittelyt komponenttien jatkokäyttöä varten.

Sovellusarkkitehtuurin suunnittelu ja dokumentointi kulkivat liian lähekkäin toisiaan. Refaktoroitavat komponentit olisi pitänyt määritellä tarkemmin, jotta selkeä dokumentointi olisi ollut helpompaa. Kaavioiden dokumentointityökaluina käyttämäni Draw.io ja Visual Paradigm osoittautuivat molemmat hyvin kömpelöiksi. Kun kaavioon tuli muutoksia, yhden muuttujan muuttaminen luokkakaaviosta vaati, että koko luokan muuttujat piti kirjoittaa uudestaan käsin, sillä kopioiminen ja liittäminen rikkoi tekstin formaatin lukukelvottomaksi. Kaavioiden tulee olla peruspiirteiltään helppolukuisia ja selkeitä kokonaisuuksia, sillä yhdenmukaisuus kuvaajissa lisää niiden luettavuutta huomattavasti. Toinen kehittäjä kehityksen rinnalla olisi myös tuonut varmuutta ja motivaatiota kehitykseen, eikä teknisten ongelmien takia olisi tarvinnut painia yksin. Tiimin panos ja sen vaikutus on tullut selväksi uudessa työssäni back-end -kehittäjänä.

5.3 Lopputulos

Lopputuotoksena pelin sovellusarkkitehtuurista tuli eheämpi ja selkeämpi kokonaisuus, jonka rajapintoja ja toiminnallisuuksia on kuvattu kuvaavilla kaaviolla. Tämän johdosta toimeksiantajan on helpompi lähteä jatkokehittämään peliä eteenpäin haluamaansa suuntaan. Pelin komponentit tulevat varmasti muuttumaan jatkokehityksen myötä, mutta ilman dokumentaatiota ja refaktorointia projektin jatkaminen olisi ollut vaikeampaa kuin projektin uudelleen käynnistäminen.

Testien luomiselle kehityksen lopussa ei jäänyt aikaa, joten niiden luominen jää jatkokehittäjille. Komponenttien muuttumisen myötä testienkin on muututtava, joten liian aikaisessa vaiheessa testien kirjoittaminen ja niiden korjaaminen voi viedä kehitykseltä aikaa.

Komponenttien hyödyntäminen jatkossa voi kuitenkin olla vaikeaa, sillä komponentteihin muodostui melko spesifiset rajapinnat ja osat omaavat vanhentuneita Unityn tarjoamia ratkaisuja, jotka pitäisi korvata uusimmalla versiolla. Pelejä

luotaessa syntyy harvoin komponentteja, joita pystyisi hyödyntämään muissa projekteissa. Vain hyvin samankaltaiset pelit ja projektit pystyvät käyttämään luotuja komponentteja pienillä muutoksilla niin, että se on vielä kustannustehokasta. Pelejä ei kannata lähteä rakentamaan komponenttien uudelleen käytettävyyttä mielessä pitäen, vaan peliä pitää luoda pelin omilla ehdoilla: mitä peli tarvitsee ollakseen helppo kehittää ja mukava pelata? Uusia projekteja luotaessa voidaan miettiä, pystyisikö vanhojen projektien komponentteja hyödyntämään tässä projektissa vai ei. Pelejä kehittäessä kannattaakin suunnata katse nykyhetkeen ja lähitulevaisuuteen, eikä miettiä mahdollisia tulevia projekteja.

Sovelluksen lisäksi sain paljon kokemusta eheän sovellusarkkihtuurin luomisesta ja dokumentaatiosta, mikä tulee auttamaan minua sovelluskehittäjän uralla.

Lähteet

All About Lean. 2016. The Key to Lean – Plan, Do, Check, Act! https://www.allaboutlean.com/pdca/. 20.11.2019.

Benson, D. 2017. Part 1: Working with Files. Draw.io Online. https://support.draw.io/display/DO/Part+1%3A+Working+with+Files. 22.11.2018.

Dam, R & Siang, T. 2019. Stage 4 in the Design Thinking Process: Prototype. Interaction Design Foundation. https://www.interaction-design.org/literature/article/stage-4-in-the-design-thinking-process-prototype. 13.5.2019.

Devlin, K. 2011. Mathematics Education for a New Era. Video Games as a Medium for Learning. Nattick, Massachusetts: A K Peters, Ltd. 5-10 s.

Duffy, S. 2018. Introduction to the new Unity 2D tilemap system. Raywenderlich. https://www.raywenderlich.com/23-introduction-to-the-new-unity-2d-tilemap-system. 21.11.2018.

Fowler, M. 2014. Refactoring: Improving the Design of Existing Code Massachusetts, Courier in Westford. 9, 17-18, 90 s.

House, B. 2018a. Unity. Evolving multiplayer games beyond UNet. https://blogs.unity3d.com/2018/08/02/evolving-multiplayer-games-beyond-unet/?_ga=2.7117553.2914498.1557900598-1938539523.1509183832. 15.5.2019.

House, B. 2018b. Unity. https://blogs.unity3d.com/2018/09/12/multiplayer-connected-games-first-steps-forward/. 15.5.2019.

Leppänen, K. 2018. Henkilökohtainen tiedonanto. 23.11.2018.

Liedman, P. 2014. Liedman. The Hitchhacker's Guide To Tiled Maps. http://www.liedman.net/tiled-maps/. 13.5.2019.

Llopis, N. 2010. Prototyping: You're (Probably) Doing It Wrong. Games From Within. http://gamesfromwithin.com/prototyping-youre-probably-doing-it-wrong. 6.11.2018.

Maanmittauslaitos. 2019. Maastokartat.fi. https://www.maastokartat.fi/. 8.3.2019.

Martin, R. 2009. Clean Code. Person Education, Inc. Massachusetts.

Metalbolix. 2011. C++-ohjelmointi: Osa 9 - Luokkien perusteet.

Ohjelmointiputka. 2018. C++-ohjelmointi: Osa 9 - Luokkien perusteet. https://www.ohjelmointiputka.net/oppaat/opas.php?tunnus=cpp_ohj_09. 21.11.2018.

Morphingtiling. 2018. Regular and semi-regular tilings. https://morphingtiling.wordpress.com/2010/12/27/regular-and-semi-regular-tilings/. 23.11.2018.

Opetushallitus. 2014. Perusopetuksen Opetussuunnitelman Perusteet. Helsinki: Next Print Oy. 128-129, 234-239 s.

Sourcemaking. 2018. Singleton Design Pattern. https://sourcemaking.com/design_patterns/singleton. 13.11.2018.

Suunnistusliitto. 2019. Suunnistus. https://www.suunnistusliitto.fi/liitto/suunnistuksen-lajimuodot/suunnistus. 1.3.2019.

Trinary Software. 2018. What is More Effective Coroutines? trinary.tech/category/mec/. 26.11.2018.
Uml-diagrams. 2018. Visibility in UML. https://www.uml-diagrams.org/visibility.html. 2019.4.12.
Unity. 2019a. Scripting. https://docs.unity3d.com/Manual/ScriptingSection.html. 16.5.2019.
Unity. 2019b. Scenes. https://docs.unity3d.com/Manual/CreatingScenes.html. 16.5.2019.
Unity. 2019c. Introduction to Scriptable Objects. https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/scriptable-objects 28.11.2018.
Unity. 2019d. Profiler overview. https://docs.unity3d.com/Manual/Profiler.html 13.5.2019.
Unity. 2019e. Prefabs. https://docs.unity3d.com/Manual/Prefabs.html. 20.5.2019.