Sami Kurvinen

Isometrinen pelimoottori

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

05.05.2016

Metropolia

Tekijä Otsikko Sivumäärä Aika	Sami Kurvinen Isometrinen pelimoottori 29 sivua + 2 liitettä 05.05.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja	Lehtori Miikka Mäki-Uuro

Opinnäytetyön tarkoituksena oli kehittää isometrisen pelimoottorin pohja ja tarkastella siinä käytettyjä ohjelmia ja työkaluja.

Moottorin toteutuksessa keskityttiin tekemään mahdollisimman joustava ja helposti ymmärrettävä järjestelmä. Samalla tutkittiin mahdollisuuksia yhdistää kaksi eri peligenreä yhdellä moottorilla.

Suunnitteluvaihe toteutettiin paperilla. Samalla tarkastettiin tärkeimpiä tarvittavia elementtejä ja tutkittiin mahdollisia 3D-moottoreita. Samalla tutkittiin samanlaisuuksia RTS- ja RPG-peligenrejen välillä.

Moottori ohjelmoitiin käyttämällä C#-ohjelmointikieltä ja Unity3D:n grafiikkamoottoria hyödyksi. Samalla hyödynnettiin joitain Unityn tarjoamia elementtejä, joilla projekti saataisiin nopeasti etenemään.

Avainsanat	C#, Unity3D, pelimoottori, isometrinen

Author Title Number of Pages Date	Sami Kurvinen Isometric Game Engine 29 pages + 2 appendices 05 May 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor	Miikka Mäki-Uuro, Senior Lecturer

The purpose of the thesis was to design and produce a basis for an isometric game engine, to study the process and understand the different phases of the progress and the tools used.

The aim for the game engine is to be flexible and as easy to understand as possible while searching for possibilities to unite two different game genres.

The design phase was mainly carried out on paper while looking at the most important parts of the strategic game engine and sampling different 3D engines to be used. Also the similarities between Real-time Strategy and Roleplaying Games were delved into.

The engine was programmed with the C#-programming language and Unity 3D was picked as the 3D engine to be used as a basis. Some elements provided by Unity 3D were used to get a quick start on the project.

Keywords	C#, Unity3D, game engine, isometric

Sisällys

1 Johdanto

Peliteollisuus on kasvanut suurella vauhdilla viime vuosikymmenien aikana ja on tullut huomattavaksi työllistäjäksi ohjelmoijille ja suunnittelijoille. Monenlaisia innovatiivisia pelejä on luotu yhä värikkäimmillä tavoilla. Tällä hetkellä suurin osa peliteollisuudesta on kohdistunut mobiilipelaamiseen hyvin onnistuneiden ja vielä paremmin rahaa tuottaneiden pelien, kuten Clash of Clansin ja Angry Birdsin ansiosta. Muutamat pelit ovat tarvittuun työhön ajateltuna erittäin onnistuneita ja tuottaneet hyvin paljon tuloja tekijöille.

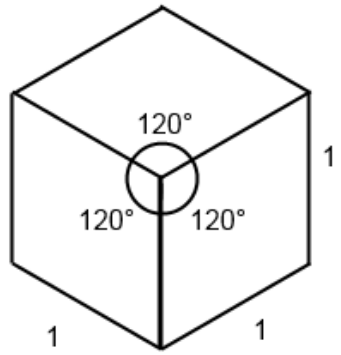
Muut alueet peliteollisuudessa ovat myös tuottaneet pelejä, osa niistä on hyvin menestyneitä. Tässä on auttanut pelin tarina, mekaniikat tai yksinkertaisesti jakamaton huomio mediassa. Strategiapelien määrä verrattuna kaikkeen muuhun on hyvin pieni, ellei kyse ole kevytstrategioista. Starcraft 2:n jälkeen on ollut monia projekteja, joista mikään ei ole päässyt yhtä hyviin tuloksiin. Monia hyviä pelejä kuten Grey Goo ja Act of Aggression on saapunut, mutta ne eivät ole silti onnistuneet saamaan sen isompaa huomiota pelien maailmassa.

Tämän projektin tarkoituksena oli tehdä isometrisellä kameralla varustettu strategiapelimoottori, joka tulee toimimaan pohjana tuleville peleille. Koodaustyö on tehty C#:lla ja Visual Studio 2015:lla. Pelimoottorin on tarkoitus valmistua mahdollisimman joustavaksi ja helposti ymmärrettäväksi. Moottorissa on tarkoitus mahdollistaa kooditiedostojen helppo vaihtaminen toiseen, selkeä modulaarinen rakenne luokkien välillä, selkeästi kommentoiduilla ja rakennetuilla koodiluokilla.

Tärkein tehtävä on saada aikaan järkevä kokonaisuus pelimoottorista ja mahdollisesti tuottaa protoversio siitä. Samalla tutkitaan mahdollisuutta laajentaa moottoria eri peligenreihin. Strategiapelejä lähimpänä genrenä voidaan pitää roolipeligenreä, joka on lähestymistavaltaan hyvin erilainen strategiapeleihin verrattuna. Niissä toimitaan osittain samalla tavalla kuin strategiapeleissä, mutta ne sisältävät erilaisuuksia monilla eri alueilla pelimekaniikoista tarinankerrontaan. Tämän takia näiden genrejen eroavaisuuksien tutkiminen moottorissa on helpompaa.

2 Strategisista ja isometrisista pelimoottoreista

Isometrinen projektio on yleisin aksonometrisen projektion tapa kuvata 3D-esine 2D-maailmassa, missä esitetään esineestä kolme eri sivua (ks. kuva 1). Kyseisessä projektiossa kaikkien akseleiden väliset kulmat ovat 120 astetta, tämän takia kaikki niiden suuntaiset viivat ovat samassa skaalassa. Aikaisemmin tämä on yksi tunnetuimmista tavoista tehdä peliin 3D-maailma 2D-moottorilla.



Kuva 1. Isometrinen projektio

Monissa aikaisemmissa pelimoottoreissa käytetty isometrinen kuvaustapa ei kuitenkaan ole täysin isometrinen vaan hyödyntää toisia kulmalukuja saavuttaakseen vähemmän sahalaitaisen tuloksen pikseligrafiikoilla. Nykyään isometrinen pelimoottori tarkoittaa monien mielestä kaikkia yläviistosta kuvattuja pelejä. Tämä muutos alkoi, kun 3D-moottorit tulivat entistä tehokkaammiksi ja 2D-moottorien suosio alkoi hiipua.

Ennen 3D-moottorien yleistymistä strategiapelit olivat yleensä kuvattu joko suoraan yläpuolelta tai isometrisesti yläviistosta. Pitemmän päälle jälkimmäinen kuvaustapa on havaittu parhaimmaksi strategiapeleihin ja pelaajien suosio on varsin huomattava. Nykyään moottoreissa hyödynnetään enemmän 3D-perspektiivejä kuin varsinaista projektiota isometrisyyden luomiseksi. Tämän takia siitä on tehty yksi tapa myydä peliä, eikä mitään syytä päivittää sanaa toiseen muotoon ole tarvittu.

Tavallisesta isometrisestä moottorista voidaan esimerkiksi nostaa Baldur's Gate, Sim City, Command & Conquer: Red Alert 2, Age of Empires 2 (ks. kuva 2), Fallout-sarjan ensimmäiset osat ja Jagged Alliance 2.

Kuva 2. Esimerkki aidosti isometrisestä pelistä: Age Of Empires 2

Nykyään lähes kaikki strategiapelit on tuotettu 3D-moottoreilla, jotka hyödyntävät tuttua ”isometristä” kamerakulmaa yhdistettynä 3D-maailman vapaaseen liikkeeseen. Silti useat mekaaniset ratkaisut, joita on hyödynnetty vanhemmissa peleissä, toimivat myös 3D-ympäristössä. Grid-pohjainen kenttäjärjestelmä helpottaa reitinhaun, kenttien ja erikoistoimintojen suunnittelusta suhteellisen helppoa, kun estetyt alueet ja laukaisimet voidaan helposti määrittää tietyille alueille [3].

2.1 Tärkeimmät osat

Monet strategiapelit ovat nousseet suureen suosioon ja joistain on tullut kulttiklassikoita. Näiden pelien suosio pohjautuu useampaan eri tekijään, mutta työssä tullaan keskittymään pelin sisäisiin tekijöihin. Suurimmat pelin suosioon vaikuttavat neljä aluetta ovat yksinkertaistettuna moottori, visuaalinen ilme, äänimaailma ja tarinaan liittyvät tekijät. Tämä projekti keskittyy enemmän mekaaniseen puoleen koko järjestelmässä, mutta muiden alueiden vaikutusta pelien suosioon ei voida kieltää.

Yleisesti otettuna ainakin kolmen eri osan on toimittava tarpeeksi hyvin, että pelistä tulee huomattava. Kulttiklassikoiden on mahdollista saada suuri kannatus heikommallakin toiminnallisuudella, mutta niiden suosio yleensä jää myös pienemmäksi. Moottori kuitenkin on se alue, joka on kaikkein tärkein pelaajien mielenkiinnon ylläpitämiseksi. Moottoriin

sisältyvät kaikki pelimekaniikat, tekoäly ja muut pelin pyörittämiseen tarvittavat osat. Pelimekaniikkoihin voidaan laskea säännöt ja mekaniikat, jotka määräävät pelin etenemisen, lopun ja yleiset vuorovaikutukset pelin ja pelaajan välillä. Pelimekaniikkojen yleisestä määritelmästä ei kuitenkaan ole päästy täyteen sopimukseen.

Moottorin on toimittava peleissä ilman isompia ongelmia, vaikka jotkin virheet ohjelmoinnissa voivat lisätä tapoja lähestyä peliä ja muuttua pelaajien ansiosta uudeksi ominaisuudeksi. Moottorin osa-alueista tietyt ovat paljon voimakkaammin esillä strategiapeleissä kuin missään muissa peligenreissä ja vaikuttavat genressään eniten pelin suosioon ja pitkäikäisyyteen.

2.1.1 Tekoäly ja reitinhaku

Strategiapeleissä tekoäly on varsin tärkeä osa moottoria. Tekoäly määrää, miten yksiköt toimivat ilman pelaajan antamia komentoja ja miten tietokone ohjaa omia joukkojaan. Tekoälyä strategiapeleissä voidaan verrata ihmisen reaktioihin taistelukentällä ja joukkojen johdossa. Jos vihollinen havaitaan, yksiköt tulevat avaamaan tulen kohteeseen ilman muuta komentoa. Jos vihollisen joukkoja havaitaan selustassa, paljon joukkoja laitetaan komentajan toimesta kyseiseen suuntaan puolustamaan. Kaikki tämä määritellään tekoälyllä.

Toinen tärkeä osa tekoälystä on joukkojen ja objektien reitinhaku. Reitinhaku on eräs osa tekoälyä, joka yleensä huomataan pelissä vasta, kun se tekee jotain väärin. Erilaiset yksiköt myös tarvitsevat omat reittinsä joilla kulkea. Maayksiköiden, ilmayksiköiden ja meriyksiköiden reitit voivat erota toisistaan hyvin paljon riippuen kenttien rakenteesta. Siellä, mistä maajoukot eivät pääse kulkemaan, ilmayksiköt todennäköisesti pääsevät helposti. Reitinhaun moitteeton toimivuus on eräs parhaimmista tavoista parantaa tekoälyn tehokkuutta.

Yhdessä reitinhaku ja muut tekoälyn osat luovat tehokkaan tekoälyn, joka koittaa sopeutua tilanteeseen, iskeä odottamattomista kulmista ja tehdä harhautuksia. Nykyiset tekoälyt eivät kuitenkaan pääse vielä niin korkealle tasolle, ja ne luottavat lähinnä massiivisiin joukkoihin eri yksiköitä, joita ne siirtävät reitinhaun antamien reittien mukaan. Tämän takia tekoälyn tehokkuus osittain riippuu siitä, kuinka kenttien reitinhaku on rakennettu ja kuinka paljon eri mahdollisuuksia tietokoneille se antaa.

Kehittyneimmät tekoälyt eivät silti pääse yhtä korkealle ajatustasolle kuin ihmiset. Tämän takia haasteen lisäämiseksi annetaan niille yleensä lisäresursseja pelaajaan verrattuna ja muuta vastaavaa. Tekoälyn haastavuus riippuu tässä vaiheessa sen ohjelmoinnista ja siitä riippuen vaikuttaa siihen, tuntevatko pelaajat konetta vastaan pelaamisen haasteeksi vai yksinkertaisesti epäreiluksi.

2.1.2 Kontrollit ja käyttöliittymä

Toinen osa moottoria ovat kontrollit ja käyttöliittymä, jotka määrittelevät, kuinka helppoa tai vaikeaa pelin pelaaminen on, jos tekoälyä ei lasketa mukaan. Näiden avulla pelaajat suorittavat kaiken, minkä päättävät tehdä. Koska strategiapelejä on tuotettu jo useampi vuosikymmen, tietynlaiset peruskontrollit ovat olemassa, ja useat pelaajat ovat tottuneet kyseisiin kontrolleihin. Samanlaista pohjaa ei ole käyttöliittymässä, ja se vaihtelee eri pelien välillä hyvinkin paljon usein eri pelaajahahmojen taitojen mukaan.

Se miten nämä kaksi osaa on koodattu pelimoottoriin voi helposti upottaa pelin, ellei kaikki muu ole täydellisiä. Varsinkin strategiapeleissä ja roolipeleissä on hyvin tärkeää pystyä kontrolloimaan joukkojaan suurimmalla mahdollisella helppoudella, ellei kummalliselle käyttöliittymälle tai ohjaustavalle ole kunnollista selitystä tai syytä.

Käyttöliittymä on tärkeä osa pelimoottoria. Vaikka sen osa on enemmän esittää tietoja järkevämmässä valossa kuin pelkissä numeroissa, on sen kautta myös tärkeää antaa mahdollisuus ohjata yksiköitä ja rakennelmia strategiapeleissä. Pelin luonne tulee vaikuttamaan tämän takia varsin raskaasti siihen, miten ja minkälaista käyttöliittymää aletaan suunnitella.

Esimerkiksi Red Alert 2 (ks. kuva 3) -strategiapelissä hallinnoidaan rajatonta massaa sotajoukkoja. Tärkeämpää on pystyä nopeasti rakentamaan oikeanlaisia joukkoja tai rakennuksia vastustajan joukkoihin verrattuna ja iskeä oikeaan paikkaan. Lisäksi itse aktivoitavia erikoistaitoja yksiköitä ei joukoilla pahemmin ole, vaan suurin osa taisteluista perustui kivi-paperi-sakset-metodiin ja resurssipeliin. Taistelu keskittyy lähinnä suurien joukkojen siirtämisestä oikeaan paikkaan oikeaan aikaan.

Kuva 3. Red Alert 2:n käyttöliittymä [5]

Starcraft 2:n yksiköillä puolestaan on usealla paljon erilaisia taitoja, joilla on mahdollista kääntää taistelun kulku täysin. Tämän takia peliin on luotu paljon kattavampi käyttöliittymä yksiköiden kontrolloimiseen ja sen kautta helpompi pääsy yksiköiden taidoille (ks. kuva 4). Yksikköjä tuottavat rakennukset on muutettu noudattamaan samaa käyttöliittymää. Tämä luo paljon paremmin hallinnoitavan järjestelmän yksittäisille yksiköille. Armeijat Starcraft 2:ssa voivat olla hyvin suuria, mutta yksiköiden erikoistaidoilla pystytään vaihtamaan taistelun suunta helposti. Tämän takia yksittäisten yksiköiden ohjaaminen on paljon tärkeämmässä osassa Starcraft 2:ssa kuin Red Alert 2:ssa

Kuva 4. Starcraft 2: Legacy of the Void:n käyttöliittymä [6]

Tämä antaa selvää tietoa siitä, miten paljon eri osa-alueet peleissä vaikuttavat toisiinsa ja kuinka käyttöliittymän on toimittava niiden osa-alueiden ehdoilla. Käyttöliittymä on pelissä se osa, jossa suurin osa eri mekanismeista vaikuttaa näkyvimmin. Samalla käyttöliittymä voi kertoa yllättävän paljon, mihin pelissä keskitytään.

2.1.3 Toiminnallisuus ja tasapaino

Pelin ongelmaton toimivuus on tärkeää, jos minkäänlaista pelaajakuntaa haluaa pelissään ylläpitää. Jos peli kaatuu hyvin usein käynnistyessään ja päällä ollessaan, pelille voidaan odottaa ongelmia sekä pelaajamäärässä että arvosteluissa. Tämä vaikuttaa siihen, kuinka monia pelaajia peli tulee kiinnostamaan tulevaisuudessa, ja muutamat ongelmalliset PC-julkaisut viime vuosina ovat näyttäneet kuinka suureksi takaiskuksi huono julkaisuversio voi rakentua. Arkham Knight -peli on eräs huomattavimmista esimerkeistä, jolla on ollut erittäin huono PC-julkaisu.

Kuitenkin täysin ongelmatonta julkaisua ei koskaan ole ollut, ja useat pelaajat ovat tehneet tavakseen metsästää erilaisia reikiä suunnittelussa huvin tai hyödyn vuoksi. Tämä myös auttaa pelin tekijöitä tarkistamaan, missä kyseinen virhe aiheutuu ja korjaamaan kaikkein pahimmat virheet pelien toimivuudessa.

Nämä ongelmat voivat johtua monesta eri tekijästä, joita tekijät eivät ole joko ajatelleet tai yksinkertaisesti muodostavat tilanteen, jota kukaan ei pystynyt ennustamaan. Esimerkiksi Borderlands 2 on toimiva peli ilman liian suuria helposti havaittavia ongelmia tai virheitä, mutta hidas verkkonopeus voi aiheuttaa pelaajille ongelmia kenttien lataamisessa ja heittää heidät siksi aikaa lähes tyhjään kenttään (ks. kuva 5).

Kuva 5. Borderlands 2, kun kenttä ei lataudu ajoissa. Kuvasta on poistettu pelaajien nimet.

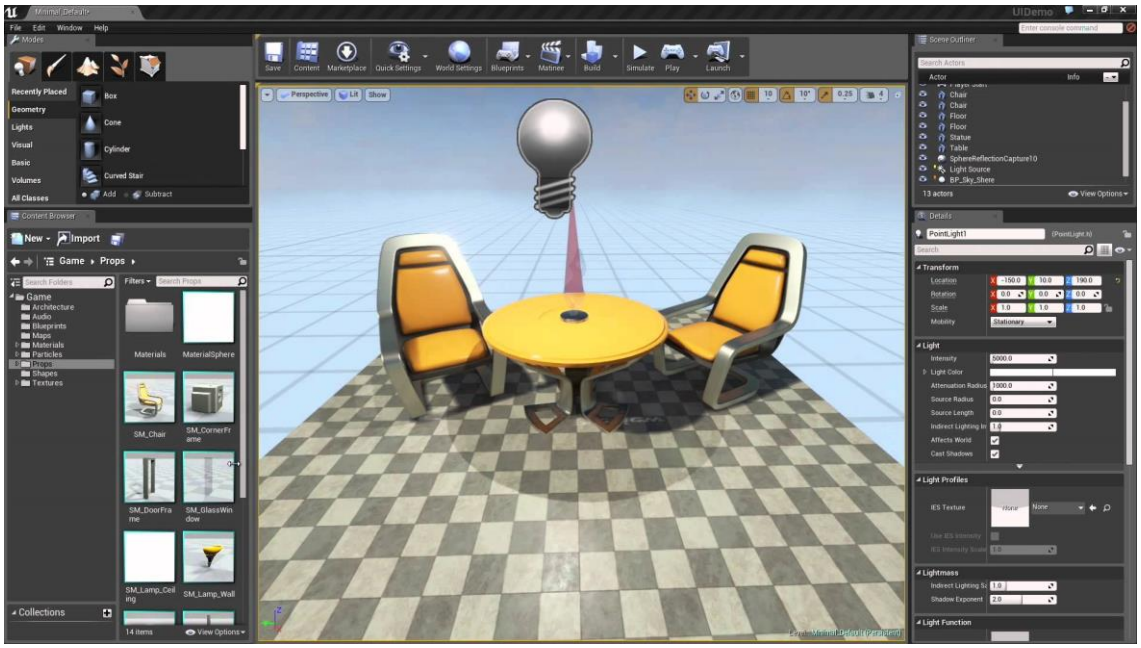
Toinen pelin pitkäikäisyyttä lisäävä asia on pelin sisäinen tasapaino eri joukkojen välillä. Tasapaino vaikuttaa jokaisessa taistelussa, ja sen takia on varsin tärkeä tekijä pelin mukavuutta ajatellen. Yksiköiden tehokkuus toisia yksiköitä vastaan on tärkeä osa pelin sisäistä tasapainoa. Tasapainon saavuttaminen on kuitenkin pidempi prosessi lähinnä tekijöiden naurettavan määrän takia ja on tämän takia hyvin hankala asia saavuttaa. Sen takia tasapainoon kohdistuvia päivityksiä tulee vanhoihinkin peleihin, kunnes jotain tasapainon suuntaista on saavutettu.

3 Moottorin suunnittelu

Suunnittelun alkuvaiheissa ymmärrettiin, ettei aikaa ole suunnitella kokonaista grafiikkamoottoria ja useampaa muuta tarvittavaa järjestelmää. Tämän takia päätettiin hyödyntää jotain olemassa olevaa grafiikka- tai pelimoottoria, jotta suoranaista moottoria päästään työstämään.

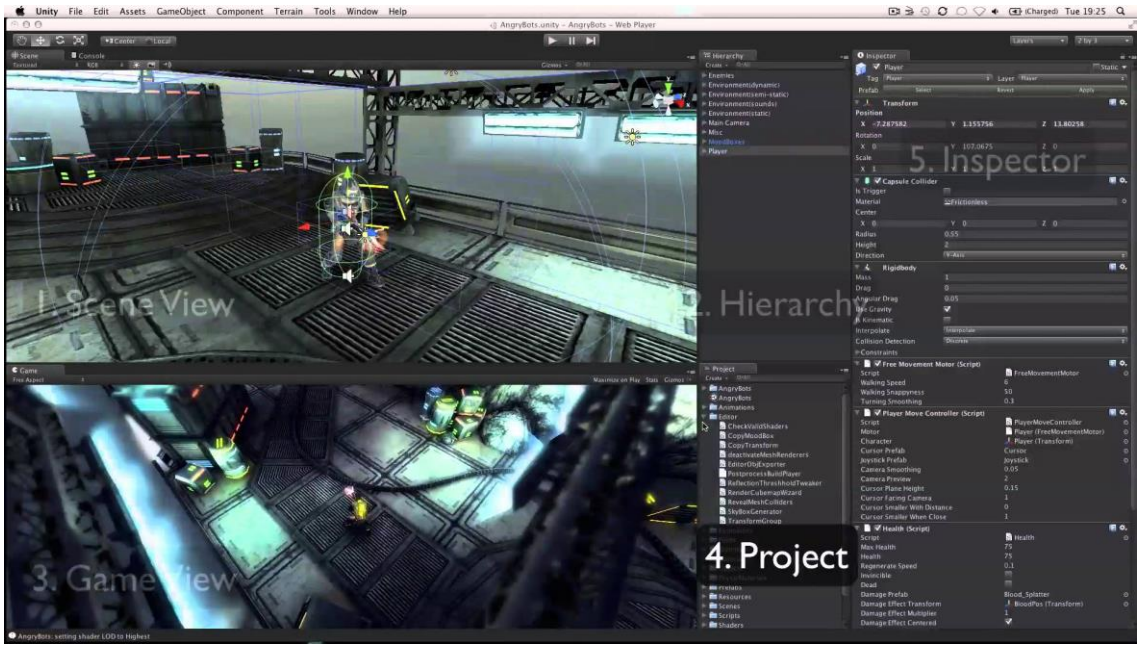
3.1 Grafiikkamoottori

Nopean tutkinnan jälkeen kaksi eri moottoria on havaittu parhaimmiksi vaihtoehdoiksi: Unreal Engine 4 (ks. kuva 6) ja Unity (ks. kuva 7). Molemmat ovat harrastekäyttöön ilmaisia, sisältävät toimivan editorin ja pystyvät hyödyntämään C#-koodia. Niillä on omat eronsa tehokkuudessa, grafiikoissa ja sisäisissä mekaniikoissa.



Kuva 6. Unreal Engine 4:n päänäkymä [7]

Vaikka moottorina UE4 on uudempi ja selkeästi graafisesti parempi [1], se myös tarkoittaa, etteivät siihen liittyvät ohjeet ja keskustelufoorumit tule sisältämään kaikkia mahdollisia ongelmatilanteita toisin kuin Unityssä, joka on jo useampia vuosia vanha ja sen takia paljon paremmin dokumentoitu [4]. Tämä aiheuttaa ongelmia, kun jokin moottorin osa muutetaan täysin. Vanhasta osasta löytyy vielä dokumentaatiota, vaikka kyseiset järjestelmät ovat vanhentuneita, tai parempia ratkaisuja ongelmiin on jo luotu.



Kuva 7. Unityn kustomoitu pääeditori, Mac-versio [8]

Unreal Enginen tarjoama ”Blueprint” visuaalinen koodausjärjestelmä vaikuttaa mielenkiintoiselta ja helpolta. Samalla kun Unityssä lähes kaiken tarvittavan hyvin vähäisellä koodaamisen määrällä, vaikka onkin suositeltua hyödyntää Unityn hienoa ohjelmointirajapintaa sen tarjoaman valmiskirjaston ja Asset Storen avulla [2]. Tämä kuitenkin taistelee varsinaisen lopputyön tarkoitusta vastaan, ja Unity tarjoaa myös helpon tavan liittää koodi haluttuun objektiin tiedostona.

Aikaisempaan Unreal Development Kit:iin verrattuna Unreal Engine 4 vaikutti lupaavalta ja helpommalta opetella. Samoin Unityllä on useita suosituksia Unityn helppokäyttöisyyden, ohjelmointirajapinnan ja hyvän yhteisön puolesta.

Kaiken tämän jälkeen on päädytty Unityyn, joka vaikuttaa parhaimmalta vaihtoehdolta hyvän dokumentoinnin, ilmaisen lisenssin ja yllättävän yksinkertaisen koodin suorittamistavan takia.

3.2 Unityn tarjoamia elementtejä

Unity tarjoaa monia erilaisia elementtejä pelinkehittäjille ja muutamia vaatimuksia. Esimerkiksi koodeja ei voida suorittaa muuten kuin olemassa olevan peliobjektin kautta.

Tätä voidaan lyhyesti kutsua ”yhden objektin vaatimukseksi”. Vaikka on mahdollista rakentaa peli kirjoittamatta riviäkään koodia Unityllä, tulee järkevä pelinkehittäjä tuottamaan paremman pelin, kun jaksaa tutustua Unityn ohjelmointirajapintaan. Koodaamalla voidaan helposti luoda erilaisia lisäjärjestelmiä, joita pelissä voidaan tarvita.

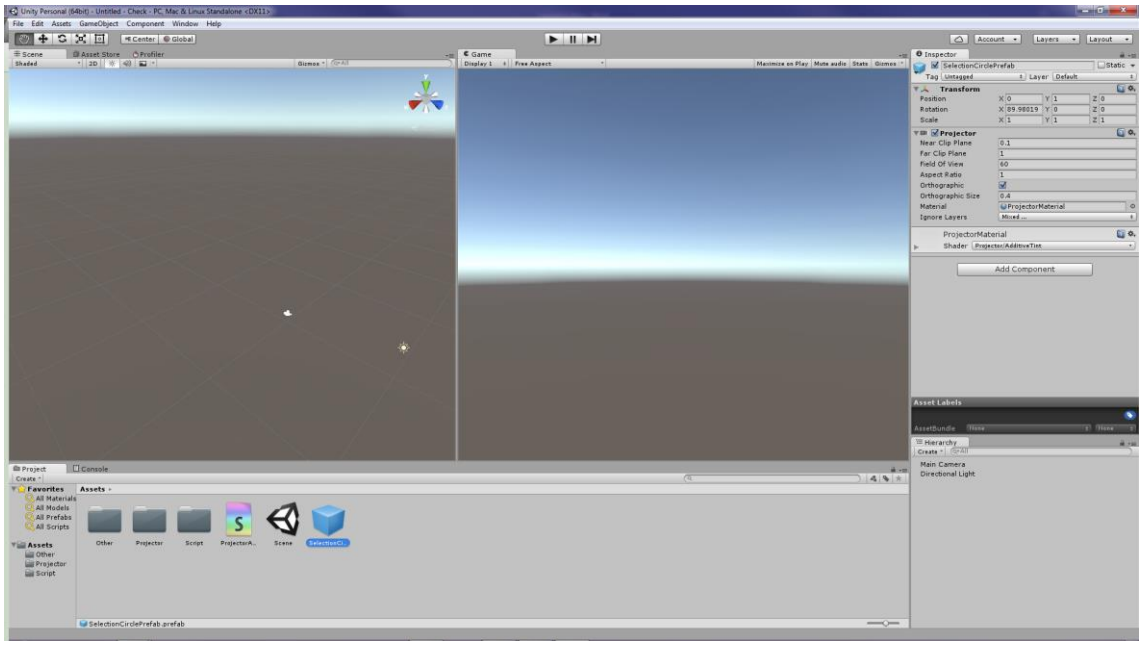
Unityn omana erikoisuutena ovat Prefab-valmiselementit, joiden avulla voidaan tallentaa valmiita yksiköitä, kameroita ja muita valmisobjekteja pelin kansioihin. Prefab-elementit voivat olla täysin valmiita, mutta niitä voi myös muokata kutsuttaessa. Prefab-elementtejä voidaan suoraan kutsua koodista ja kopioimaan niin useasti kuin tarve vaatii. Ne voidaan tallentaa pelimoottorin kansiorakenteisiin omien kansioidensa alle niiden kutsumisen helpottamiseksi.

Scene-tiedostot ovat varsinaiset kentät ja valikot, jotka on koostettu luoduista objekteista. Ne pitävät koossa kaiken kentässä tarvittavan ja niitä voidaan helposti muokata Unityn pääeditorin avulla. Vaihtaminen kentästä toiseen tehdään peliobjekteissa olevien kooditiedostojen tai komponenttien kautta.

Peliobjektit Unityssä voivat toimia lähes minä tahansa eineenä tai oliona. Se miten kyseinen objekti toimii, riippuu siihen liitetyistä komponenteista, jotka määrittelevät sen toiminnot. Koodiluokat lasketaan komponentteihin.

Skriptit Unityssä voivat sisältää useita Unityn Event-funktioita, joilla pystytään vaikuttamaan, milloin mitäkin koodissa tehdään. Ne Event-funktiot, joita tullaan koodissa tässä vaiheessa käyttämään, ovat lähinnä Start- ja Update-funktiot, joista ensimmäinen vaikuttaa nimensä mukaisesti koodin alussa ja Update moottorin jokaisella kuvan päivityksellä.

Unityn nykyinen pääeditori (ks. Kuva 8) vaikuttaa pienen tutkimisen jälkeen varsin yksinkertaiselta. Vasemmalla on ruutu/ruudut, jotka kuvaavat pelialuetta, ja niitä voidaan jakaa useampaan eri osaan. Niiden alapuolella konsoli sekä projektikansiot ja oikealla on kaikki muu tarvittava. Kaiken asioiden paikkoja voi vaihtaa kuten parhaaksi näkee. Tämä tulee auttamaan totuttautumisessa, mikä helpottaa eri osien paikantamista.



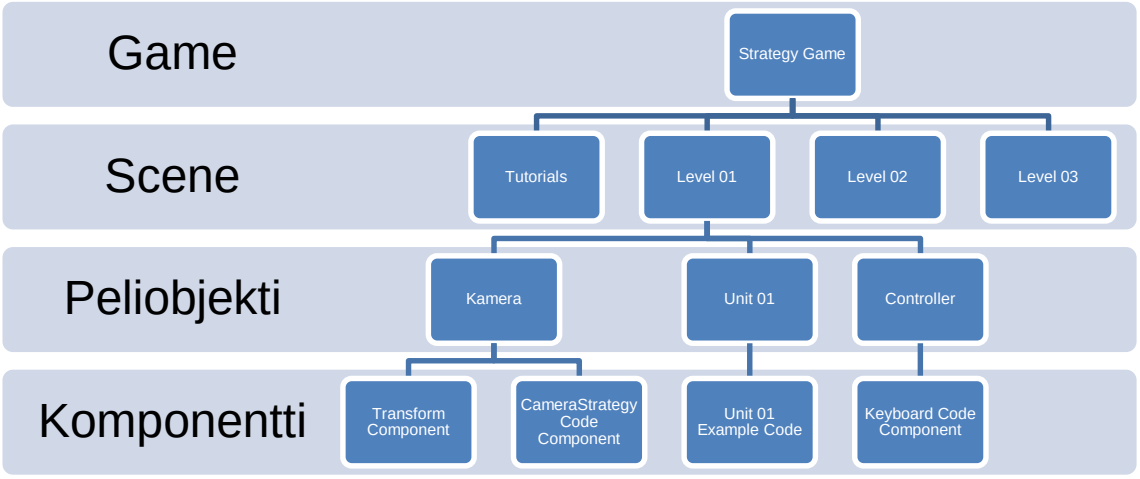
Kuva 8. Unityn päänäkymä, Windows

3.3 Rakenne

Unityn sisäinen pelin rakentaminen toimii oikeastaan kolmen peruspilarin kautta (ks. kuva 9). Nämä peruspilarit ovat Scene-tiedostot, GameObject-peliobjektit ja GameObjectien sisältämät komponentit.

Unityssä kenttä rakennetaan Scene-tiedostoon, joka tulee sisältämään kaiken kyseisessä kentässä käytettävän. Jokainen asia ja esine Scene-tiedoston sisällä on peliobjekti (GameObject), jonka omat komponenttiosat määräävät sen toiminnan.

Komponentteja objekteissa voi olla useita erilaisia komponentteja. Komponentit sisältävät kyseiselle objektille tarkoitetun toiminnallisuuden. Yksi peliobjekti voi sisältää useita kymmeniä eri komponentteja fysiikkaan, sijaintiin tai muihin tärkeisiin alueisiin liittyen. Varsinkin koodiin liittyviä komponentteja on useasti monia.



Kuva 9. Unityn kenttärakenteesta yksinkertaistettuna

Pelimoottorin rakenne tullaan jakamaan eri osa-alueiden kesken ja yleishyödylliset koodiluokan pätkät siirretään Utility-tiedostoihin. Tärkeimmät osat ovat kontrollit, käyttöliittymä ja tekoäly. Toiminnallisuuden täydellisyys ja tasapaino on mahdollista tehdä kunnolla vasta, kun kaikki muu on valmista.

Jokaisella luokalla tulee olemaan oma virtuaaliset kantaluokkansa. Ne tulevat sisältämään tarvittavat toiminnot, jotka ovat kaikille kyseisen luokan olioille samat. Niissä tosin voidaan käyttää aliluokan attribuutteja tarvittaessa. Tarkoituksena on tehdä koodiluokista helposti vaihdettavia ja muokattavia selkeällä pohjarakenteella.

Kunhan tietyt piirteet, jotka on havaittu tarvittaviksi, on toteutettu aliluokassa sitä voi käyttää moottorilla. Vaikkakin tietyt asiat, jotka havaitaan tarvittaviksi, jättäisikin koodaamatta, tulee suurimpiin kohtiin olemaan pohjakoodi, joka periytyy aliluokalle, ellei sitä ole päällekirjoitettu kyseisessä aliluokassa.

3.3.1 Koodikirjastot

Muita koodikirjastoja, kuin Unityn omaa ohjelmointirajapintaa, tuskin tullaan käyttämään ennen pelimoottorin kokonaista valmistumista, jolloin voidaan ryhtyä tarkastelemaan mahdollisuuksia koodin optimoimiseen. Tämä on lähinnä helppoa objektien käsittelyä ja moottorin tehon kasvattamista varten.

Työn tarkoituksena on tuottaa mahdollisimman suuri osa koodista itse. Tämän avulla koodin siirtäminen muille moottoreille ei tule niin raskaaksi työksi. Lopulta, kunhan moottori on valmis, voidaan tutkia mahdollisuuksia optimoida järjestelmää Unity-version valmistumisen loppuvaiheissa.

MonoBehaviour-luokka toimii kantaluokkana kaikille Unityn koodiluokille ja sisältää useita erilaisia Event-funktioita. Tämän takia vain Utility-kooditiedostot eivät sisällä kantaluokkanaan MonoBehaviouria. Muilla se ajetaan kantaluokkien kantaluokkana hämmennyksen vähentämiseksi. MonoBehaviourin hyödyllisyys yksinkertaisesti on liian suuri sen huomiotta jättämiseksi.

3.3.2 Järjestys

Kun pelimoottoria aloitetaan koodaamaan, on tärkeintä aloittaa tekemällä yksinkertainen pohjasuunnitelma siitä, mitä osia moottoriin koodataan ja mitkä ovat tärkeimmät osat pelimoottorin alkuvaiheessa. Tällä saadaan aikaan selkeä alkupiste, johon keskittyä koodatessa, ja se myös helpottaa projektin aikatauluttamista, vaikka kyseistä projektia koodattaisiinkin työpäivän jälkeen.

Helpoin aloituspiste moottorissa on yksinkertaisten virtuaalisten luokkien koodaamisessa, jotka tulevat toimimaan pohjana halutuille osille ja edetä siitä kamerakontrolleihin, yksiköiden liikkeisiin ja käyttöliittymään. Tästä edettäisiin tekoälyn ja karttatoimintojen kautta reitinhakuun ja muihin alkuvaiheessa vähemmän tärkeisiin osiin. Näihin loppuvaiheen töihin kuuluvat mm. animaatiot ja mahdollinen fysiikkamoottorin hyödyntäminen, johon Unity antaa oman osansa Event-funktioilla ja erikoisemmilla metodikutsuilla.

Kannattaa ottaa huomioon, että tietyt koodausalueet vaativat useampaa toiminnallisuutta eri koodialueilla. Esimerkiksi reittihakua ei voi kunnolla koodata, jos kentän rakenteesta ja yksiköiden luonteesta ei vielä ole tehty päätöksiä.

3.3.3 Modulaarinen ohjelmointi

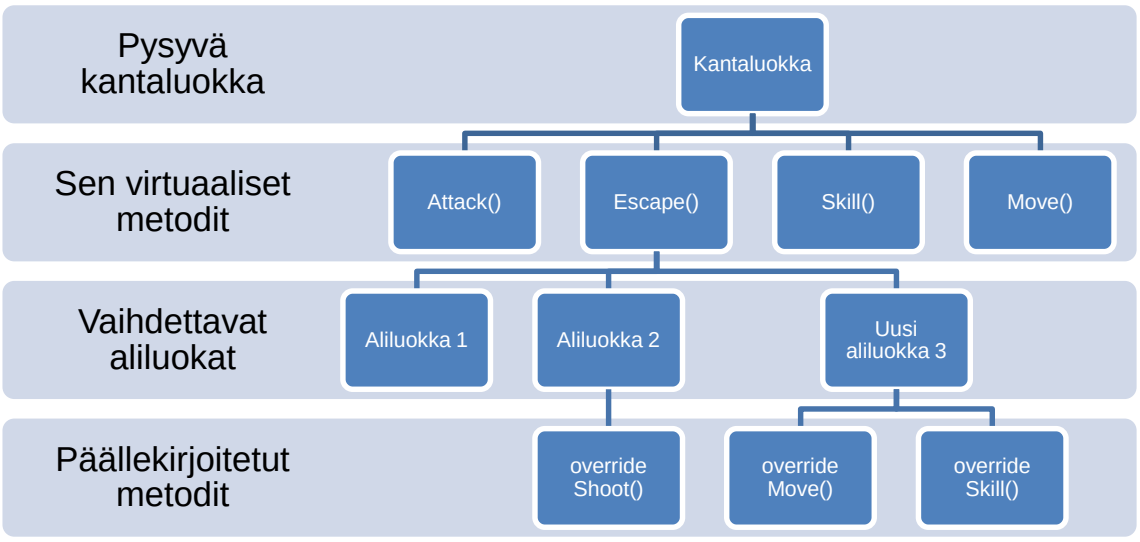
Alusta alkaen modulaarisuus koodissa on ollut yksi suurimmista kohteista, joka on haluttu toteuttaa tässä moottorissa. Kooditiedostojen helppo vaihtaminen objektissa on yksi

askel, mutta Event-funktioiden sisällä oleva koodi jaetaan omiin metodeihinsa. Tällä tavoin koodia on helpompaa käydä läpi ja vaihtaa niitä alueita, joita halutaan.

Kantaluokissa olevat virtuaaliset metodit on suurimmaksi osaksi kirjoitettu toiminaan kyseisen metodin mukaisesti ilman ulkoisia vaikuttajia. Esimerkiksi yksikön kantaluokassa on yksinkertainen liikkumismetodi, joka toimii aliluokkien metodina liikkua niin kauan kuin sitä ei aliluokassa päällekirjoiteta. Muuttujat, joita metodi hyödyntää, otetaan aina aliluokasta, ellei sitä ole aliluokassa määritelty. Tämä helpottaa koodikomponenttien vaihtamista peliobjekteissa, vähentää turhaa koodia ja hyödyntää valmista koodia kantaluokissa.

Yksinkertaistettuna esimerkkinä kantaluokan virtuaalimetodeista Escape-metodia käyttävät kaikki aliluokat. Osa aliluokista päällekirjoittaa toisia metodeja, kuten Shoot tai Move. Silti kaikki aliluokat voivat vaihtaa paikkaa keskenään eri objekteissa ilman mitään tarvittavia muutoksia (ks. kuva 10).

Tätä voidaan hyödyntää useassa alueessa strategiapeliä. Yksiköitä ”päivitettäessä” pelissä voidaan yksikölle antaa eri muuttujat kestävyyteen tai suoraan vaihtaa tapa liikkua tai ampua. Muutoksen suuruudesta voidaan päätellä, kuinka suuria muutoksia yksikön koodiluokkaan tarvitaan, jotta se toimii moitteettomasti. Koko yksikkö voidaan myös muuttaa täysin, jolloin yksikölle on helpompaa luoda uusi aliluokka.



Kuva 10. Yksinkertaistettu kuvaus luokkien ja metodien vaihdettavuudesta keskenään.

Metodit koitetaan pitää tiettyyn pisteeseen asti lyhyinä ja sisältämään yhden tietyn toiminnon. Tähän kuitenkin tehdään poikkeuksia esimerkiksi kontrolleissa, koska niiden toiminnat voivat tarvita useamman eri muuttujan liikuttamista edestakaisin. On arveltu, että muuttujien pitäminen samalla alueella on varsin tärkeä osa suoralukuisuutta ja helppokäyttöisyyttä.

Kyseessä ei ole vain tapa antaa pelikehittäjille helpompi tapa toteuttaa omiaan toimivalla koodipohjalla, vaan myös pelin ulkopuolisille ohjelmoijille ja suunnittelijoille mahdollisuus luoda helposti jotain uutta peliin. Tietenkin on mahdollista käyttää vapaata pohjaa huijaamiseen pelissä. Tämän takia on tärkeää saada osittainen tarkistusalgoritmi moottorin versiolle tai yleiselle kokonaisuudelle.

3.4 RTS:n ja RPG:n vaihdannaisuus

Moottorilla halutaan myös todistaa, kuinka helposti on mahdollista vaihtaa pelin pelimuoto tai peligenre, kunhan aiheet ovat riittävän lähellä toisiaan. Pääpiirteittäin molemmat strategiapelit toimivat samalla tavalla kuin roolipelit. Varsinkin pelin kontrollitasolla on helppo havaita yhteneväisyyksiä. Ohjataan yksiköitä, taistellaan vihollista vastaan ja suoritetaan tehtäviä. Suurin osa eroista näiden pelien välillä on roolipelien raskaampi suhtautuminen hahmoihin ja niiden sisäisiin rakenteisiin, erilaisuudet käyttöliittymässä ja strategiapelien tarve tehokkaampaan tekoälyyn.

Tarkemmin roolipeleissä, esimerkiksi Baldur's Gatessa (ks. kuva 11), yksiköiden varusteet ovat paljon tarkemmassa tarkastelussa kuin strategiapelissä, jossa voidaan sanoa joukkojen yksinkertaisesti kantavan perusvarusteita. Roolipeleissä taistelut jakautuvat enemmän taktisten kohtauksien puolelle kuin suurempaan strategiaan. Näissä kohtauksissa yksiköiden yksittäiset taidot, varusteet ja sijainti voivat olla ero elämän ja kuoleman välillä.

Kuva 11. Baldur's Gatessa pelaaja ohjastaa kuuden tai alle kuuden henkilön ryhmää fantasiamaailmassa.

Suurin todennäköisin muutos tulee käyttöliittymään. Käyttöliittymää on muokattava tarkemmin pelin omia tarkoitusperiä varten. Varsinkin mahdollisuus taitojen pikaiseen käyttämiseen on tärkeää, kuten on niiden helppo vaihtaminen toisiin taitoihin. Vaikka suurin osa roolipeleistä on joko vuoropohjaisia pelejä tai reaaliaikaisia, joissa voidaan yleensä keskeyttää peli siirtojen tekemiseksi ja tarkempien toimintojen suorittamiseksi ilman isompaa kiirettä.

Strategiapeleissä, esimerkiksi Command & Conquer 3:ssa (ks. kuva 12), voi yksiköiden sijainti vaikuttaa yllättävän paljon, mutta ei täysin samalla marginaalilla kuin roolipeleissä. Yhden yksikön tuhoutuminen ei vaikuta strategiapelissä yhtä voimakkaasti kuin roolipelissä. Pelkkä joukkojen koko merkkaa suurta erilaisuutta. Kun roolipelissä ohjastat alle tusinaa henkilöä, strategiapelissä on ohjauksesi alla monia kymmeniä eri joukkoa, joiden sisäinen joukkovahvuus voi olla sadoissa. Yhden sotilaan kuolema, ellei kyseinen ole erikoisyksikkö, ei vaikuta peliin isollakaan tavalla. Kaiken lisäksi rakennukset ovat strategiapelien oma erikoisuutensa roolipeleihin verrattuna, mutta niitä voidaan myös ajatella yksikköinä. Ne vain eivät pysty liikkumaan ja tuottavat tarvittavia resursseja tai yksiköitä.

Kuva 12. Command & Conquer 3:ssa joukot ovat massiivisia, ja yksittäiset yksiköt harvemmin tekevät suurta vaikutusta.

Muutoin näiden kahden genren välinen ero on lähinnä eri numeroarvojen ympärillä, joilla tarkastellaan tehdyn vahingon määrää vihollisyksikköön tai rakennelmaan. Tähän asiaan kuitenkin pystytään lähinnä keskittymään optimointivaiheessa, kun kaikki moottorin osat toimivat oikein.

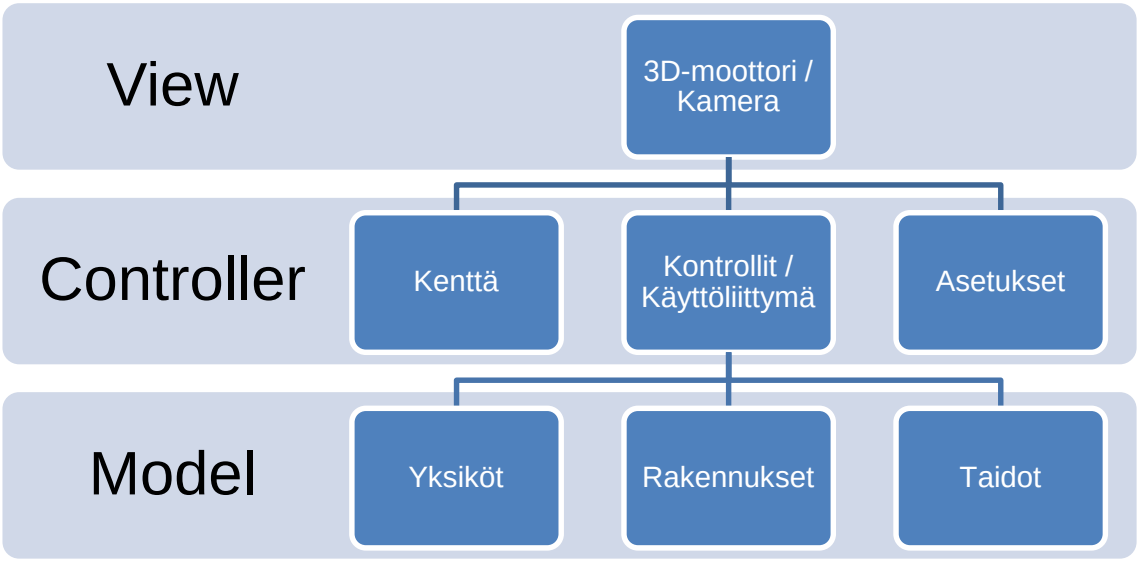
Metodien ylikuormittaminen helpottaa erilaisten olioiden luomista ja ryhmittämistä ilman, että kutsuihin eri kontrolliluokissa tarvitsee tehdä isompia muutoksia eri metodeja kutsuttaessa. Esimerkiksi liikkumiskomennon saadessaan kaikki valitut yksiköt liikkuvat annettuun pisteeseen välittämättä erilaisuuksista kyseisessä metodissa eri yksiköiden välillä. Isoin erilaisuus strategia- ja roolipelin välillä on pelaaja ja tekoälyhahmojen yksinkertaisuus. Jos strategiapelissä päivitetään yksikköä, tiedetään, mitä osaa yksikössä kasvatetaan tai lasketaan. Roolipelissä asia ei toimi niin yksinkertaisesti.

Vaikka osia voidaankin vaihtaa yllättävän helposti, vaatii jokainen komponentti joitain edellytyksiä. Ne voivat olla yllättävän turhia roolipelissä toisin kuin strategiapelissä. Varsinainen vaihdannaisuus saadaan vasta, kun moottori on saatu optimoitua ja on selvitetty, mitä voi käyttää kummallakin puolella moottoria.

4 Toteutetusta moottorista

Mitä pidempään tätä työtä on tehty, sen pidemmälle on käynyt selväksi, kuinka hankalaa on ohjelmoida pelimoottoria tyhjän päälle. Vaikka projektissa on käytetty Unityn moottoria ja editoria, menee koodia kirjoittaessa pitkään erillisten ongelmien selvittämisessä ja parhaimpien toimivien lähestymistapojen etsimisessä. Työtä aloitettaessa Unity, johon en ollut kerennyt tutustumaan kovinkaan paljon, vaikutti helpoimmalta vaihtoehdolta varsinaista koodityöskentelyä varten ja sitä se on ollut Visual Studion avulla.

Pelimoottorissa osia on aivan liikaa, jotta sitä voi ajatella puhtaana MVC-mallin projektina, mutta osat toimivat suurimmassa osassa samalla tavalla (ks. kuva 13). Kaikki, mitä tehdään kameran osoittaman ruudun kautta, liikkuu kontrolleja ja käyttöliittymää tarkkailevien luokkien kautta vaikuttamaan muuttujia ja suorittamaan metodeja yksiköistä ja muiden kentällä olevien objektien koodiluokista.



Kuva 13. Yksinkertaistettu tapa ajatella moottoria

Jokaisella alueella on oma kansionsa. Kontrollereina toimivat näppäimistöä ja hiirtä seuraavat koodikomponentit sekä kentän ja pelin asetuksiin vaikuttavat koodikomponentit. Koska kaikki kooditiedostot on oltava kiinnitettynä objektiin, luodaan kenttään tyhjä olio, johon liitetään kaikki tarvittavat koodikomponentit. Tällä tavalla saadaan aikaan yksittäinen objekti, jota kutsumalla päästään käsiksi kaikkiin ympäristöllisiin kooditiedostoihin, jos tiettyjä asioita esimerkiksi kentän koossa tarvitsee muuttaa.

4.1 Moottoriin implementoidut toiminnallisuudet

Projektissa on tällä hetkellä ohjelmoitu kameran liikkeitä hallinnoivia luokkia, hiiren toiminnollisuuksia ja yksikköjen toiminnollisuuksiin liittyviä luokkia. Myös jonkin verran tutkimusta on tehty Unityn käyttöliittymään liittyvän koodin osalta. Muuten on lähinnä tutkittu mitä eri alueita olisi tärkeintä priorisoida ja kirjoitettu alustavia kantaluokkia eri alueille.

Kamera toimii sekä hiiri- että näppäimistö ohjauksella. Näppäimistössä peli tunnistaa nuolinäppäimet ja WASD-näppäinten painallukset ja liikuttaa kameraa vastaavasti kiihtyvällä vauhdilla, kunnes saavuttaa maksiminopeuden. Mahdollista on myös kameran pyörittäminen kiintopisteen ja oman akselin ympäri. Zoom-toiminnallisuus on lisätty hiiren rullaan ja hiiren liikkuessa lähelle näytön reunaa tulee kamera liikkumaan sen kyseisen reunan suuntaan.

Hiirellä pystyy valitsemaan ja liikuttamaan yksiköitä, joihin on tällä hetkellä ohjelmoitu liikettä varten arvot ja erilaisia tarvittavia muuttujia. Drag & Hold -valintatapa on myös ohjelmoituna hiiren toimintoihin. Valmiit metodit toimivat odotetusti hiiren ja näppäimistön koodiluokissa, mutta ovat vielä varsin keskeneräisiä.

Yksiköiden kantaluokka sisältää perusliikkeet ja ampuu kaikkia vihollisjoukkueessa olevia yksiköitä automaattisesti. Aliluokilla on onnistuneesti testattu muuttujien ja metodien toiminnollisuuksien vaihdoksia päällekirjoitettaessa eri metodeja ja muuttujia.

Käyttöliittymään luodaan 5 eri tekstilaatikkoa, mutta niiden sijainnit ja graafinen olomuoto eivät ole kovinkaan näyttävät. Käyttöliittymän sijaintiin liittyy hyvin monia eri tekijöitä joiden ratkaisemisessa voi aikaa kulua yllättävän paljon. Kannattaa huomioida, että Unityllä on tällä hetkellä kaksi eri tapaa koodin kautta luoda käyttöliittymä, mutta uudempaa suositellaan sen helpomman toiminnallisuuden takia. Kaikki käyttöliittymän paneeliosat laitetaan Canvas-objektille ja yhdistetään pääkameraan. Vaikka osittainen ratkaisu aikaisemmin minikartan näyttämiseksi kentästä oli jo olemassa, ei se toiminut kunnolla uudemmassa käyttöliittymän versiossa.

4.2 Osien vaihdettavuus

Kaikki koodiin liittyvät luokat tehdään eri tiedostoihin samaa pohjakaavaa käyttämällä ja tiedostojen vaihtaminen keskenään pystytään suorittamaan helposti. Objektiin lisätyn koodiluokan nimi voidaan helposti vaihtaa Unityn omasta editorista objektiin liitetystä koodikomponentista. Tilanteen niin vaatiessa objekteille voidaan lisätä eri kooditiedostoja myös koodiluokkien itsensä kautta, mutta se vaatii tarkempaa suunnittelua.

Tietenkin koodin toimivuus riippuu kantaluokan riippuvuuksien mukaisesta koodaamisesta, mutta se on ohjelmoijien itse toteutettava ja testattava. Jokainen virtuaalinen kantaluokka sisältää omat yksinkertaiset versionsa tarvittavien metodien rakenteesta, paitsi erikoistoiminnoista. Erikoistoiminnot, esimerkiksi yksiköillä, on mahdotonta määritellä kantaluokassa muina kuin abstrakteina metodeina. Erikoistoimintojen varsinaiset metodit sijaitsevat vain aliluokissa. Käyttöliittymään tai yksikköpohjaan on tarkoitus koodata metodi, joka tarkastaa onko erikoistoimintoja päällekirjoitettu yksiköissä ja luo vastaavan näppäimen käyttöliittymään.

Käyttöliittymä tullaan viimeistelemään vasta loppuvaiheessa ja sen on myös oltava joustavin kaikista osista, varsinkin strategiapeli/roolipeli vaihdannaisuutta miettiessä. Parhaiten toimivan järjestelmän pystyy kuitenkin rakentamaan optimointivaiheessa, kun tiedetään kaikki suurimmat tekijät rakenteiden välillä.

Koodiluokkien ja niissä olevien osien vaihdettavuus ei vielä tässä vaiheessa ole kunnolla näkyvissä. Tällä moottorilla valmistettavat pelit tulevat hyödyntämään Prefab-objekteja paljon enemmän niiden yksinkertaisen käytön ja muokattavuuden takia. Kuitenkin huomattavaa on, että kooditiedostojen kantaluokka vaikuttaa hyvin voimakkaasti siihen, miten kyseistä objektia hallinnoidaan.

Lisäksi ylikuormitetut funktiot kantaluokissa voivat sisältää kaiken, mitä vaihdannaisissa tarvitaan. Täten ne vähentävät koodaustyön määrää, kun suunnitellaan uusia yksiköitä ja kontrollimallin päivityksiä pelin eri osia varten.

4.3 Ratkaisuja koodissa

Koodi on tarkoitettu helppolukuiseksi useammalla eri tavalla. Esimerkiksi ennen kaikkia funktioita kommentoidaan, mitä varten kyseinen funktio on, ja myöhemmässä vaiheessa myös, miten niitä hyödynnetään. Koodin suoralukuisuus on tärkeää pelimoottorille, jos sitä aiotaan käyttää myöhemmässä vaiheessa johonkin toiseen projektiin. Pienetkin muutokset koodissa voivat aiheuttaa uusia isompia virheitä, jos koodaaja ei tiedä, mitä tekee, tai on unohtanut jonkin osan koodistaan. Näiden virheiden korjaamiseen tulee menemään työtunteja yhä enemmän, jos koodin alkuperäinen kirjottaja on vaihtunut toiseen, jolla ei ole mitään suoranaista tietoa koodista ja siitä, miten koodin on tarkoitus toimia.

Kaikki muuttujat, paitsi väliaikaiset, nimetään englanniksi helposti ymmärrettävillä nimillä tai nimiyhdistelmillä. Sen merkitystä koodissa ei voi vähätellä, ja se vaikuttaa varsin paljon eri järjestelmien rakentamisessa. Tärkeää on varoa, ettei koodiin kirjoiteta jo valmiiksi käytettyjä nimiä samaan tiedostoon, ellei sitä pystytä perustelemaan. Varsinkin koodikirjastojen, ohjelmointirajapintojen ja kantaluokkien nimettyjä muuttujia kannattaa varoa, koska niiden käyttäminen voi aiheuttaa odottamattomia seurauksia.

Suurin osa koodista ja metodeista suoritetaan tällä hetkellä Update-metodin sisällä. Kyseinen metodi suoritetaan jokaisen ruudun aikana kerran. Kaikki koodi sen sisältä koitetaan sijoittaa erillisiin metodeihin myöhempää käyttöä ja helpompaa ymmärtämistä varten.

Projektissa on käytetty tähän mennessä muutamaa erilaista tapaa yksinkertaistaa koodirakennetta ja samalla koittanut luoda koodista mahdollisimman helposti vaihdettavaa. Nämä kuitenkin toimivat kunnolla vain, jos kyseinen koodaaja jaksaa lukea hieman, mitä on muokkaamassa. Metodien ylikuormittaminen, selkeä koodikäsiala, raskas kommentointi ja selkeä nimeämistapa muuttujille ovat lyhyesti sanottuna tärkeimmät kohteet tämän moottorin koodissa.

4.3.1 Yksi kutsu, toimiva kokonaisuus

Unity toimii todella hyvin yhden kutsun pohjalla. Kun objekti, jossa on liitettynä kooditiedosto, luodaan kentälle, se automaattisesti aloittaa suorittamaan skriptejä ja koodiluo-

kissa sijaitsevia Event-funktioita ja niiden kautta koodiluokan muita metodeja. Tämä toiminta on täysin automaattista, kunhan käytetty koodiluokka perii Unityn MonoBehaviour-kantaluokan. Tämä helpottaa huomattavasti erilaisten toimintojen suorittamista oikeassa järjestyksessä.

Tärkeä osa tätä ajatusta on metodien ja luokkien ylikuormittaminen. Kaikki saman alueen järjestelmät, esimerkiksi yksiköt, käyttävät samaa kantaluokkaa. Ne toimivat samoilla metodeilla, mutta erilaisuudet kantaluokasta ajetaan ylikuormitettujen tai päällekirjoitettujen metodien kautta.

Samalla saman kantaluokan omistavat objektien koodiluokat hyödyntävät virtuaalikantaluokassa olevia funktioita ja päällekirjoittavat funktiot, jotka vaihtuvat kantaluokasta. Tällä tavalla moottori voi käyttää samaa kutsua ohjatakseen useaa erilaista objektia, joilla on sama kantaluokka. Tämä tekee koodista suoralukuista ja, ylikuormittamalla alkuperäisen funktion, hyvin helposti vaihdettavaksi. Jos kooditiedostoista tulee liian pitkiä, voidaan luoda tukea antavia tiedostoja tietyille ylikuormitetuille toiminnoille, jotta yhden rivin löytämiseksi ei tarvitse käydä läpi kymmentä tuhatta riviä koodia. Tämä helpottaa koodin läpikäymistä, jos ei tiedä täysin, mitä suoranaisesti etsii.

4.3.2 Helppo ymmärrettävyys

Ilman kommentteja jokainen rivi on käännettävä yksi kerrallaan ja varsinkin kooditiedoston ollessa pala koottua kaaosta on koodin kääntäminen hyvinkin haastavaa. Tämän takia selkeä koodaustapa ja kommentointi on hyvin tärkeää, jotta muutkin kuin vain koodaaja itse ymmärtää helposti, mitä koodilla on ajettu takaa.

Kameran kantaluokan koodaus sisältää parhaan erimerkin luokkien rakenteesta moottorissa (ks. kuva 14). Siitä voidaan suoraan nähdä, mitä tarvitaan varsinaisen kamera-koodiluokan aikaansaamiseksi. Kameran asetukset tulevat vaihtumaan eri peligenrejen välillä, ja alustavasti kantaluokka pysyy lähes puhtaasti abstraktina. Myöhemmässä vaiheessa sille saatetaan tuottaa virtuaaliset metodit yleiskäyttöä varten.

```
abstract public class CameraTemplate : MonoBehaviour{

    // variables that need to be
    public float cameraSpeed = 15;
    public float cameraZoomSpeed = 10;

    //Abstract method to create a focuspoint for the camera
    public abstract GameObject CreateFocusPoint();

    //Abstract methods
    public abstract void CameraKeyPressed ();

    //Abstract methods that move the camera
    public abstract Vector3 CameraMovement (int k);
    public abstract Vector3 CameraRotation ();
    public abstract Vector3 CameraOrbit ();
    public abstract Vector3 CameraZoom (float scroll);
```

Kuva 14. Kameran kantaluokka

Muuttujat ja koodi yleensäkin on kirjoitettu englanniksi käyttäen yksinkertaista sanastoa. Vaikka tämä luokka, kuten kaikki muutkin, tulee muuttumaan tulevaisuudessa, se sisältää kaiken tarvittavan.

CameraStrategy-lapsiluokka päällekirjoittaa kantaluokan metodit ja suorittaa kaikki tarkistukset LateUpdate Event -funktion aikana, joka suoritetaan viimeisenä kaikista Event-funktioista. Sen aikana lasketaan yleinen kameran liikkumissuunta, nopeus sekä mahdolliset variaatiot kuten pyöriminen ja zoom. Mutta keskitymme nyt käymään läpi CameraMovement-metodin läpikäyntiä (ks. kuva 15).

```
//Keys to move camera Arrow keys or WASD keys, Call CameraMovement() to create movement vector
//Add created vector to the Final movement vector
if (Input.GetAxis ("Horizontal") != 0 || Input.GetAxis ("Vertical") != 0 ){
    finalMove += CameraMovement (0);
}
if (Input.GetKey (KeyCode.W) || Input.GetKey (KeyCode.S) || Input.GetKey (KeyCode.A) || Input.GetKey (KeyCode.D)) {
    finalMove += CameraMovement (1);
}
```

Kuva 15. CameraStrategy aliluokan LateUpdate Event -funktion osa, tarkistaa jos kameraa liikutetaan ja luo vektorin lopullista suunnan laskentaa varten

Aluksi kommenteissa selitetään, mitä metodi tekee ja mitä se saa vastaukseksi tai vaikuttaa. Tässä tapauksessa tarkastellaan sekä nuolinäppäimien että WASD-näppäimien painalluksia. Tällä saadaan selville, minne kameran on tarkoitus liikkua neljässä ilmansuunnassa. Molemmat ovat yksittäisen jos-lauseen takana, että molempien tulokset saadaan eriteltyä ja laskettua yhteen helposti.

Varsinainen CameraMovement-metodi (ks. kuva 16) on hieman kevyemmin kommentoitu kuin Update-funktiossa olevat metodikutsut. Tärkeintä on selittää, mitä milloinkin tapahtuu ja mihin tietty osa metodia on tarkoitettu.

```
// Check wasd or arrow keys forinput and move as such
public override Vector3 CameraMovement(int k){

    //Arrow keys axis variables
    float mHorizontal;
    float mVertical;

    Vector3 move = new Vector3 (0, 0, 0);
    //Arrow Keys
    if (k == 0) {
        //Get axis input from arrow keys
        mHorizontal = Input.GetAxis ("Horizontal");
        mVertical = Input.GetAxis ("Vertical");

        //Add to return Vector
        move = new Vector3 (mHorizontal, mVertical, 0.0f);

    } //WASD keys
    else {
        if (Input.GetKey (KeyCode.W))
            move += (transform.forward * cameraVelocity * Time.deltaTime);
        if (Input.GetKey (KeyCode.A))
            move += (transform.right * cameraVelocity * Time.deltaTime);
        if (Input.GetKey (KeyCode.D))
            move += (-transform.right * cameraVelocity * Time.deltaTime);
        if (Input.GetKey (KeyCode.S))
            move += (-transform.forward * cameraVelocity * Time.deltaTime);
    }


    //return final vector
    return move;
}
```

Kuva 16. CameraStrategy-luokan päällekirjoitettu CameraMovement-metodi

Koodiluokkien rakenne perustuu yleiseen luokkarakenteeseen, johon suurin osa koodaajista on tottunut. Aluksi määritellään tarvittavat muuttujat, jos näin kantaluokassa ei vielä ole tehty tai niitä tarvitaan vain lyhytaikaisesti. Annetaan niille arvot Start-metodissa, joka käydään läpi kerran objektin ilmestyessä. Tämän jälkeen Update- ja LateUpdate-metodit käyvät läpi omat toimintonsa ja tällä hetkellä projektissa kaikki työ keskittyy tälle alueelle ja sen sisältämiin metodeihin. Kaiken lopuksi tulevat uudet metodit ja päällekirjoitetut kantaluokan metodit. Niiden kutsut suoritetaan Update-metodien sisällä jokaisen ruudunpäivityksen aikana ja sisältävät kaiken työn, jota luokissa tuotetaan.

Ainoat erilaiset koodiluokat ovat Utility-luokat, jotka sisältävät yleiskäyttöisiä metodeja eivätkä noudata yleistä koodipohjaa tai omista MonoBehaviouria kantaluokkana. Utility-luokkien sisältämiä metodeja kutsutaan lähinnä muista luokista eikä sen takia sisällä mitään Event-funktioita. Liitteenä opinnäytetyössä on kyseiset kameran koodiluokat kokonaisuudessaan parempaa tarkastelua varten (liite 1 ja liite 2).

5 Pohdintaa

Tämän projektin aikana on onnistuttu tutustumaan Unityyn varsin hyvin. Vaikka muutamia osia ohjelmasta ei ole avautunut kunnolla, projektissa on loppuvaiheessa pystytty paljon paremmin hyödyntämään Unityn tarjoamat elementit. Projektin aikatauluttaminen on voitu toteuttaa paremmin, mutta eri ongelmilla on onnistuttu sotkemaan ne useaan kertaan.

5.1 Unityn toiminnallisuus pelinkehitysohjelmana ja 3D-moottorina

Unity on hyvin monipuolinen ohjelma ja on huomattavaa, että sitä päivitetään edelleen yhä tehokkaammaksi. Pitkäikäisyytensä vuoksi se on myös saanut hyvin tehokkaan yhteisön taaksensa, joka auttaa mielellään uusia ohjelmoijia ja pelintekijöitä Unityn mielenkiintoisemmista ominaisuuksista.

Projekti ei ole kirjoitusvaiheessa vielä päässyt animaatiovaiheeseen, joten moottorin tehokkuudesta ei voi suoranaisesti omasta kokemuksesta kertoa mitään. Sen tekevät tuhannet pelit, jotka Unityllä on tehty. Vaikka osalla tuotetuista peleistä on tiettyjä fysiikkaan liittyviä ongelmia ja Unityn 3D-moottori ei ole yhtä uusi ja tehokas kuin UE4, on Unity edelleen toimiva ja varsin hyvin optimoitu moottori, kunhan ohjelmoidun pelin sisällä ei ole mitään isompia ongelmia kuten muistivuotoja.

Unity tunnetaan useista yksinkertaisilla grafiikoilla tuotetuista peleistä, mutta tiettyjä suurempia graafisia töitä on pakko ihailla (ks. kuva 17). Unreal Engine 4:n tasolle Unity tuskin pääsee, mutta vanhaksi moottoriksi se tekee hyvin hienoa jälkeä. Suurin tekijä graafisessa loistavuudessa on se, kuinka paljon aikaa grafiikkaan tullaan aikaa käyttämään. Tässä varsinkin Unityn Asset Store voi vaikuttaa varsin paljon, koska se tarjoaa ilmaiseksi ja rahaa vastaan isojakin 3D-grafiikkapaketteja.

Kuva 17. Unityn DirectX11-kilpailun kunniamaininnan ansainnut teos. [9]

Yksinkertaisesti sanottuna Unity on helppokäyttöinen ohjelma, jonka taustalla on hyvin taipuva 3D-moottori. Toisin kuin Unreal Engine 4:ssä, kun isompia muutoksia tulee moottoriin, ei Unitystä tehdä seuraavaa kaupallista versiota vaan päivitetään vanha versio uuteen. Kyseiset päivitykset on myös mahdollista estää, jos haluaa varmistaa, ettei mikään muutos ohjelmassa sekoita projektia.

Unityn sisäinen ohjelmointirajapinta sisältää kaikki peruselementit, joita pelin tekemiseen voidaan tarvita. Koodiluokkien kautta voidaan luoda yksinkertaisia elementtejä, joille voi säätää omat asetukset heti luomisen yhteydessä, ja kooditiedostot voidaan liittää heti luomisprosessin jälkeen, tosin vain olemassa olevista kooditiedostoista.

5.2 Toteutus

Vaikka projekti muutamien tekijöiden takia muuttuikin hieman haastavaksi ajallisesti, on projekti onnistunut varsin hyvin ja tulee jatkumaan sivuprojektina opintojen jälkeenkin. Olemassa olevat kamera ja näppäinkontrollit toimivat juuri niin kuin pitääkin. Ainoastaan pientä optimointia kameran liikkeiden nopeudessa tarvitaan. Kontrollien ja käyttöliittymän yhdistäminen tulee olemaan seuraava olennainen osa ja näppäinyhdistelmien selventäminen ja muokattavuus.

Yksiköiden toimivuus on perustasolla. Niillä on tärkeimmät arvot selvillä: liikkuvat komennon saatuaan, ja ne pystyvät hyökkäämään vihollisen kimppuun, kunhan sellainen määritellään. Vaikka tulevaisuudessa tehdään aliluokat useammalle erilaisille yksikölle, tulee suurin osa niiden suunnittelusta pelien tekijöillä. Tämä lähinnä jatkuvien väärinkäytösten takia eri materiaalien välillä tiettyjen indie-pelien ”tekijöiden” toimesta.

Eri alueiden priorisointi ja koodaaminen olivat tärkeä osa projektissa ja hieman paremmalla aikataululla olisin hyvin todennäköisesti saanut enemmän aikaan. Yksinkertaisesti sanottuna, jos olisin tuntenut Unityn perusteellisemmin kuin sen, että se hyödyntää C#-pohjaisia kooditiedostoja hyvin tehokkaasti ja on yllättävän helppokäyttöinen. Pienen alkukankeuden jälkeen ohjelma tuli selkeytymään varsin nopeasti, vaikka päätökseni koodata kaikki, mitä koodin kautta pystyi, oli hieman hitaampaa tehdä, kuin se olisi ollut varsinaisen kenttäeditorin kautta.

5.3 Tulevaisuudessa toteutettavaa

Projektissa on hyvin paljon tekemistä ennen kuin sitä voi suoranaisesti kutsua moottoriksi, mutta priorisoituina tekijöinä on käyttöliittymän viimeistely, jonkinlaisen tuotantorakennuksen tai kohteen koodaaminen. Siitä voidaan siirtyä reitinhakuun ja tekoälyyn.

Varsinaisen selkeän rakennejärjestelmän luominen projektiin on varsin tärkeä osa projektia. Tämän avustuksella kooditiedostoista saadaan yhteneväisellä tavalla merkatut ja täten helpommin ymmärrettävät ilman isompia hankaluuksia, miten kaikki toimii.

Vaikka projektiin on tarkoitus tehdä osittainen yksinkertainen animaatiojärjestelmä, on tärkeämpänä kohteena tuottaa toimiva moottori, jolloin animaatio ja grafiikkapuoli tulevat näyttämään yksinkertaiselta. Tämän moottorin avustuksella pelejä tekevät itse saavat tuottaa 3D-mallit ja animaatiot haluamalleen tasolle.

Kaikki muu hoidetaan myöhemmässä vaiheessa samalla, kun tutkaillaan koodia RTS/RPG vaihdannaisuutta varten.

5.4 Lyhyesti työstä kokonaisuutena

Opinnäytetyössä on tutkittu pelimoottorin suunnittelua ja rakentamista alun suunnittelusta toteutukseen, vaikka aika ei riittäisikään työn täydelliseen valmistumiseen. Samalla on pohdittu mahdollisuutta toteuttaa siitä tarpeeksi joustava useammalle eri peligenrelle ilman liian suurta eroa koodissa. Tässä apuna on käytetty Unityä ja sen tarjoamaa 3D-moottoria. Ohjelmointiprosessi on kirjoitettu C#:lla.

Moottoria on suunniteltu täysin tyhjältä pöydältä, joka on ollut mielenkiintoinen prosessi alusta loppuun. Kaikki ideat on koostettu järkevään pakettiin, vaikka se on ottanut oman aikansa. Yksinkertaisuus on nostettu tärkeimmäksi tehtäväksi näiden suunnitelmien myötä ja sillä on hyvin paljon parannettu työnopeutta. Aloituspisteenä kameran ohjelmointi on ollut parhain mahdollinen uudelle Unityn käyttäjälle. Tällä tavalla on saatu konkreettista työtä aikaan nopeasti.

Työn aikana on opittu käyttämään ja paremmin hyödyntämään Unityä ja sen elementtejä. Ison projektin suunnitteleminen on aiheuttanut alussa ongelmia, mutta lopulta suunnitelmat on saatu aikaiseksi parin uudelleenkirjoituksen jälkeen. Unity on varsin helppokäyttöinen ohjelma, jonka 3D-moottoria on pystytty alkuvaikeuksien jälkeen helposti ohjaamaan koodin kautta. Varsinkin Unityn tarjoamia elementtejä on tutkittu mielenkiinnolla ja niitä on hyödynnetty projektin edetessä.

Koodaamisen aikana on tullut vastaan useita tuttuja ongelmia eri ratkaisujen välillä, vaikka joitain poikkeuksiakin on ollut. Isoimmat ongelmat on sisältyneet Unityn vanhempien versioiden koodattavasta käyttöliittymärajapinnasta, joka on aikaisempana vuotena vaihdettu toiseen tehokkaampaan ja yksinkertaisempaan versioon. Sen takia käyttöliittymä on kirjoitettu kaksi kertaa uudestaan, kunnes vanha versio on huomattu vaihtaa uudempaan. Tämä on vaikuttanut varsin paljon aikatauluun.

Joitain huomioita on selvitetty eli peligenrejen väliltä, vaikka sovellusmahdollisuudet ovat olleet vähäiset pelimoottorin alkuvaiheessa. Projektin loppuvaiheilla on selvitetty, että varsinkin strategiapelien ja roolipelien yhdistäminen voidaan mahdollistaa muuttamalla yksiköiden erilaisuus taitojen monipuolisuuteen. Strategia- ja roolipelien yhdistelmiä on tuotettu aikaisemminkin, muttei kovinkaan laajalla skaalalla.

Tehtyyn työhön ollaan täysin tyytyväisiä ja sen kehitystyötä tullaan jatkamaan.

Lähteet

1 Yleistä tietoa Unitystä. Wikipedia. Viimeksi muokattu 24.10.2015. Verkkodokumentti. https://fi.wikipedia.org/wiki/Unity_%28pelimoottori%29. Luettu 13.08.2015.

2 Yleistä tietoa Unreal Enginestä 4:stä. Wikipedia. Viimeksi muokattu 31.03.2016. Verkkodokumentti. https://en.wikipedia.org/wiki/Unreal_Engine#Unreal_Engine_4. Luettu 13.08.2015.

3 Isometristen maailmojen luomisesta. Game Development Envato Tuts+. Verkkosivu. Viimeksi muokattu 04.04.2016. http://gamedevelopment.tutsplus.com/tutorials/creating-isometric-worlds-a-primer-for-game-developers--gamedev-6511. Luettu 28.07.2015.

4 Unityn ohjelmointirajapinta. Unity Documentation. Verkkosivu. Viimeksi muokattu 26.03.2016. http://docs.unity3d.com/ScriptReference/. Luettu jatkuvasti.

5 Uutinen: C&C: Red Alert 2:n siirtymisestä ilmaisjakeluun, tanskaksi. IGN. Verkkosivu. Viimeksi muokattu 29.03.2016. http://dk.ign.com/spilkultur/69290/news/fa-command-conquer-red-alert-2-gratis. Luettu 12.01.2016.

6 Starcraft 2:een liittyvä kokoelmavideo tilanteista turnauksista. DotaGeeks. Verkkosivu. (http://dotageeks.com/starcraft-ii-best-moments/) Viimeksi muokattu 31.05.2015. Varsinainen kuvan osoite: http://dotageeks.com/wp-content/uploads/2015/05/starcraft-2-epic.jpg. Lisätty 22.03.2016.

7 Unreal Enginen virallinen Youtube-kanava. Kuva UE4:n esittelyvideosta. Youtube. Verkkosivu. Viimeksi muokattu 25.04.2016. https://www.youtube.com/user/UnrealDevelopmentKit/playlists. Varsinainen kuvan osoite: https://i.ytimg.com/vi/QMsFxzYzFJ8/maxresdefault.jpg. Lisätty 22.03.2016.

8 Unityn virallinen Youtube-kanava. Kuva Unityn käyttöliittymävideosta. Youtube. Verkkosivu. Viimeksi muokattu 25.04.2016. https://www.youtube.com/channel/UCG08EqOAXJk_YXPDsAvReSg. Varsinaisen kuvan osoite: https://i.ytimg.com/vi/5cPYpI6_yLs/maxresdefault.jpg. Lisätty 22.03.2016.

9 DirectX 11 kilpailun (http://blogs.unity3d.com/2013/02/01/directx-11-competition/). Kunniamaininnan ansainnut teos. http://blogs.unity3d.com/wp-content/uploads/2013/02/itscalledlight21.jpg. Lisätty 02.04.2016.

Muut kuvat ovat tuotettu peleistä itse, tehty Windows Wordin SmartArt:lla tai piirretty koneella.

CameraTemplate -kantaluokka

```
using UnityEngine;
using System.Collections;

abstract public class CameraTemplate : MonoBehaviour{

    // variables that need to be
    public float cameraSpeed = 15;
    public float cameraZoomSpeed = 10;

    //Abstract method to create a focuspoint for the camera
    public abstract GameObject CreateFocusPoint();

    //Abstract methods
    public abstract void CameraKeyPressed ();

    //Abstract methods that move the camera
    public abstract Vector3 CameraMovement (int k);
    public abstract Vector3 CameraRotation ();
    public abstract Vector3 CameraOrbit ();
    public abstract Vector3 CameraZoom (float scroll);

}
```

CameraStrategy -aliluokka

```
using UnityEngine;
using System.Collections;

public class CameraStrategy : CameraTemplate {
    // Call variable for the camera component
    private Camera cam;
    // Camera Position
    public Vector3 camPos;
    // Call variable for focus point object
    public GameObject FocusPoint;

    // Maximum movement speed, Velocity & Zoom speed
    public float camMaxSpeed = 2.0f;
    public float cameraVelocity = 0.5f;
    public int zoomSpeed = 5;

    //Mouse boundary for mouse movement
    public int boundary = 15;

    //Variables for screen width & length
    public int width;
    public int height;

    //Boundaries for camera
    public float Min_Z = -500f;
    public float Min_X = -500f;
    public float Min_Y = -500f;
    public float Max_Z = 500f;
    public float Max_X = 500f;
    public float Max_Y = 500f;

    //Variable for counted movement vector
    Vector3 finalMove = new Vector3(0,0,0);

    // Use this for initialization
    void Start () {

            width = Screen.width;
            height = Screen.height;

            FocusPoint = CreateFocusPoint ();

            cam = gameObject.GetComponent<Camera> ();
            cam.orthographic = true;

            cam.backgroundColor = Color.green;
            cam.enabled = true;
            FocusPoint.transform.parent = cam.transform;

            if (cam == null)
                        Debug.Log("Camera Null");



            }
```

```
// Update is called once per frame
void Update () {


}

// Last Update of the frame
void LateUpdate () {
    // Get camera position
    camPos = cam.transform.position;

    //Mouse boundary movement
    finalMove += CameraMovementMouseBounds();

    //Zoom
    float scroll = Input.GetAxis("Mouse ScrollWheel");
    if (scroll != 0)
        CameraZoom (scroll);

    //Keys to move camera Arrow keys or WASD keys, Call CameraMovement() to cre-
ate movement vector
    //Add created vector to the Final movement vector
    if (Input.GetAxis ("Horizontal") != 0 || Input.GetAxis ("Vertical") != 0 )
            {
            finalMove += CameraMovement (0);
            }
    if (Input.GetKey (KeyCode.W) || Input.GetKey (KeyCode.S) || Input.GetKey
    (KeyCode.A) || Input.GetKey (KeyCode.D))
            {
            finalMove += CameraMovement (1);
            }

    //Rotation
    if (Input.GetKey (KeyCode.Q) || Input.GetKey (KeyCode.E)) {
            CameraRotation();
            }

    //Orbit
    if (Input.GetKey(KeyCode.Z) || Input.GetKey(KeyCode.C))
    {
        CameraOrbit();
    }

    //Exec Final Move
    Vector3 MoveIt = Vector3.zero;

    //Assign movement speed boundaries for the camera
    MoveIt = new Vector3 (
            Mathf.Clamp (finalMove.x, -camMaxSpeed, camMaxSpeed),
           Mathf.Clamp (finalMove.y, -camMaxSpeed, camMaxSpeed),
            Mathf.Clamp (finalMove.z, -camMaxSpeed, camMaxSpeed));

    //Debug for Vector MoveIt, Not used due to restraining order
    //Debug.Log ("1: " + finalMove + ", 2: " + MoveIt);

    //Remove possible unintended z-axis movement
    transform.Translate(new Vector3(MoveIt.x, MoveIt.y,0));
```

```
//The actual movement and clamping camera to boundary
cam.transform.position = new Vector3(
        Mathf.Clamp(cam.transform.position.x, Min_X, Max_X),
        Mathf.Clamp(cam.transform.position.y, Min_Y, Max_Y),
        Mathf.Clamp(cam.transform.position.z, Min_Z, Max_Z));

//Zero the movement vector
        finalMove = Vector3.zero;
}

//For saved camera locations, unused
        public override void CameraKeyPressed(){}

// Check wasd or arrow keys forinput and move as such
        public override Vector3 CameraMovement(int k){

//Arrow keys axis variables
float mHorizontal;
float mVertical;

Vector3 move = new Vector3 (0, 0, 0);
//Arrow Keys
if (k == 0) {
//Get axis input from arrow keys
        mHorizontal = Input.GetAxis ("Horizontal");
        mVertical = Input.GetAxis ("Vertical");

        //Add to return Vector
        move = new Vector3 (mHorizontal, mVertical, 0.0f);
        } //WASD keys
 else {
        if (Input.GetKey (KeyCode.W))
            move += (transform.forward * cameraVelocity * Time.deltaTime);
        if (Input.GetKey (KeyCode.A))
            move += (transform.right * cameraVelocity * Time.deltaTime);
        if (Input.GetKey (KeyCode.D))
            move += (-transform.right * cameraVelocity * Time.deltaTime);
        if (Input.GetKey (KeyCode.S))
            move += (-transform.forward * cameraVelocity * Time.deltaTime);
                }


 //return final vector
        return move;
}

public Vector3 CameraMovementMouseBounds()
{
    Vector3 movement = new Vector3(0,0,0);
    Debug.Log(" " + boundary);
    if (Input.mousePosition.x > width - boundary)
    {
       movement += transform.right * Time.deltaTime * (25 * cameraVelocity);
    }

    if (Input.mousePosition.x < boundary)
    {
```

```
            movement += -transform.right * Time.deltaTime * (25 * cameraVelocity);
        }

        if (Input.mousePosition.y > height - boundary)
        {
         movement += -transform.forward * Time.deltaTime * (25 * cameraVelocity);
        }

        if (Input.mousePosition.y < boundary)
        {
         movement += transform.forward * Time.deltaTime * (25 * cameraVelocity);
        }

        return movement;
    }
    //Camera Rotattion Method, returns movement vector only activated if q key or
e key was pressed
    public override Vector3 CameraRotation(){
    //Temp move vector
    Vector3 move = new Vector3 (0, 0, 0);

    //Check if key pressed was Q otherwise it was E, Add movement to vector
    if (Input.GetKey (KeyCode.Q))
            cam.transform.RotateAround (FocusPoint.transform.position, Vec-
tor3.down, 200 * cameraVelocity * Time.deltaTime);
    else
            cam.transform.RotateAround (FocusPoint.transform.position, Vec-
tor3.up, 200 * cameraVelocity * Time.deltaTime);

    // Return movement vector
    return move;
    }
    public override Vector3 CameraOrbit(){
            Vector3 move = new Vector3 (0, 0, 0);
    if (Input.GetKey(KeyCode.Z))
        cam.transform.RotateAround(cam.transform.position, Vector3.down, 200 *
cameraVelocity * Time.deltaTime);
    else
         cam.transform.RotateAround(cam.transform.position, Vector3.up, 200 *
cameraVelocity * Time.deltaTime);
    return move;
    }

    public override Vector3 CameraZoom(float scroll){
            cam.orthographicSize += (scroll * 5);

            return new Vector3 (0, 0, 0);
    }

    public override GameObject CreateFocusPoint ()
    {
            GameObject Focus = new GameObject ();
            Focus.transform.position = Vector3.zero;
            return Focus;
    }


}
```