Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Josse Hyrkkänen

Opinnäytetyö

iPhone OS-peliohjelmointi Cocos2D-sovelluskehyksen avulla

Työn ohjaaja	Petri Heliniemi
Työn tilaaja	Kyy Games Oy, yhteyshenkilönä Jani Schulze
Tampere 06/2010	
Tekijä	Josse Hyrkkänen

Työn nimi	iPhone OS-peliohjelmointi Cocos2D-sovelluskehyksen avulla
Sivumäärä	47
Valmistumisaika	Kesäkuu 2010
Työn ohjaaja	Petri Heliniemi
Työn tilaaja	Kyy Games Oy

Tiivistelmä

Tämän opinnäytetyön tavoitteena on Kyy Games Oy:lle tekemäni toimeksiannon pohjalta esitellä iPhone OS-peliohjelmointia Cocos2D-sovelluskehyksen avulla. Työssä käydään läpi keskeisiä Legends of Elendria: The Frozen Maiden-pelin sovellusosien tekoon käytettyjä työkaluja ja tekniikoita. Työssä on paljon koodiesimerkkejä, joiden avulla tämä opinnäytetyö toimii myös oppaana Cocos2D-sovelluskehystä hyödyntävään peliohjelmointiin.

iPhone OS-käyttöjärjestelmä ja sitä käyttävät laitteet ovat hieman erikoisempia alustoja pelikehityksessä, mutta julkaistujen pelien lukumäärää tutkiessa voidaan todeta, että kysyntää on paljon. iPhone OS-pelit eivät ole keskimäärin kovin laajoja, mikä tekee iPhone OS-pelikehityksestä helposti lähestyttävän ja kasvavan liiketoiminnan alan.

Objective-C:tä käytetään pääasiallisena ohjelmointikielenä iPhone OS-sovelluksissa, ja iPhone OS-sovelluskehittäjälle sen hallitseminen on tärkeää. Opinnäytetyössä esitellään nykyään Applen kehittämän Objective-C:n tärkeimpiä ominaisuuksia ja käytäntöjä, joiden avulla saadaan näkemys iPhone OS-sovelluksen ohjelmoinnista.

iPhone SDK:n luokkakirjastot eivät tarjoa kovin paljon peliohjelmointiin tarvittavia ominaisuuksia, ja tätä puutetta täydentämään käytettiin Cocos2D-sovelluskehystä. Sovelluskehys sopii hyvin 2D-pelien kehittämiseen, ja sen tuoma lisä on suuressa osassa tämän opinnäytetyön toimeksiannon toteutuksessa. Cocos2D:n käyttöön paneudutaan, mikä on ehdotonta toimeksiannon toteutuksen ratkaisujen ymmärtämiseen.

Avainsanat Objective-C, iPhone OS, Cocos2D, Peliohjelmointi

TAMK University of Applied Sciences
Business Information Systems

Writer	Josse Hyrkkänen
Thesis	iPhone OS game programming with Cocos2D framework
Pages	47
Graduation time	June 2010
Thesis Supervisor	Petri Heliniemi
Co-operating Company	Kyy Games Oy

Abstract

The main purpose of this thesis is to introduce game programming for iPhone OS with the Cocos2D-framework. The knowledge is mainly based on my thesis assignment for Kyy Games Oy company. The thesis introduces all the most important tools and techniques which were used to accomplish program components for Legends of Elendria: The Frozen Maiden-game. This thesis contains many programming samples and it can also be used as a guide for game programming with Cocos2D-framework.

iPhone OS-operating system and devices which use it are quite different from the other platforms in the game development world that we have gotten used to. By looking at the number of published games for iPhone OS, anyone can tell that there are many customers who buy those games. Usually the games for iPhone OS do not have a large amount of content, so the game development for iPhone OS is an easily approached and growing business.

Objective-C is the main programming language used in iPhone OS programs and for a iPhone OS developer it is important know how to use it. Nowadays, Objective-C is developed by Apple and its basic features and usage are introduced in this thesis. Introducing Objective-C shows what iPhone OS programming is about.

iPhone SDK's programming libraries do not support game programming well enough, and that is why Cocos2D-framework is used to fill this need. The framework can be used for 2D game development and its importance in accomplishing this thesis assignment work was great. Cocos2D is introduced, which is important in order to understand the solutions of the assignment.

Keywords Objective-C, iPhone OS, Cocos2D, Game programming

Sisällysluettelo

1 Johdanto 5
2 Applen iPhone ja iPod Touch 7
2.1 iPhone OS -käyttöjärjestelmä 8
2.2 iPhone ja iPod Touch pelilaitteena 9
3 Objective-C-ohjelmointikieli pähkinänkuoressa 11
3.1 Syntaksin perusteet 11
3.2 Tietotyypit 12
3.3 Luokat 13
3.4 Metodit 15
3.5 Aksessorimetodit yksinkertaisesti property:n avulla 17
3.6 Muistinhallintaa Objective-C:n tapaan 19
4 iPhone SDK 21
4.1 Xcode-kehitysympäristö 21
4.2 Versionhallinta 23
4.3 Instruments – sovelluksen analysoija 24
5 Cocos2D-sovelluskehys 25
5.1 Tärkeimmät luokat 26
5.1.1 CCNode 26
5.1.2 CCDirector 27
5.1.3 CCScene 28
5.1.4 CCLayer 28
5.1.5 CCSprite ja CCSpriteSheet 29
5.1.6 CCMenu ja CCMenuItem 30
5.1.7 CCAction 31
5.2 Kosketusten vastaanottaminen ja niiden käsittely 32
5.3 Cocos2D-sovelluksen rakenne 33
5.4 Cocos2D-sovelluksen luominen 34
6 Legends of Elendria: The Frozen Maiden 37
6.1 Karttatila Cocos2D:n voimin 37
6.2 QuestEngine 39
6.3 Ilmenneet ongelmat 42
7 Loppusanat ja yhteenveto 44
Lähteet 46

1 Johdanto

Tutustuin iPhone OS-pelikehitykseen suorittaessani harjoittelua Kyy Games Oy:ssä, joka on tamperelainen vuonna 2009 perustettu pelialan yritys. Kyy Games Oy valmistaa päätoimisesti iPhone-, iPod Touch- ja iPad-laitteille pelejä, joita myydään Applen ylläpitämässä App Storessa. Tämän opinnäytetyön tekemisen aikana Kyy Games Oy kehitti ensimmäistä peliään nimeltään Legends of Elendria: The Frozen Maiden, johon kehitin toimeksiantona QuestEngine-nimellä kulkevan pelin tarinan etenemistä hallinnoivan sovellusosan ja siihen liittyvän toiminnallisuuden pelin karttatilaan.

Aiempaan ohjelmointitaustaani sisältyi hyvä Java-ohjelmoinnin tuntemus ja hieman suppeampi kokemus C++-ohjelmoinnista, joten harjoittelun aikana sain hyvän mahdollisuuden tutustua itselleni täysin vieraaseen ohjelmointikieleen Objective-C:hen. C++ tuntemuksesta on paljon hyötyä Objective-C:tä opeteltaessa, sillä onhan se nimensä mukaisestikin olio-laajennus molemmille yhteiseen C-kieleen. Suuri määrä käytännön harjoittelua tuottivat tulosta, niinpä Objective-C:n kummallinen syntaksi ja omituisuudet alkoivat tuntua jo kotoisalta ja toimeksiannon toteuttaminen mahdolliselta. Harjoittelun jälkeen oli luonnollinen valinta aloittaa opinnäytteen tekeminen jo tutussa projektissa ja kiinnostavan aiheen parissa Kyy Games Oy:lle.

Työn tavoitteena on perehtyä pelisovelluskehitykseen ohjelmoijan näkökulmasta iPhone- ja iPod Touch-laitteille, aluksi selventäen, millaisia laitteet ovat, ja kuinka ne soveltuvat pelaamiseen. Ohjelmointitekniikat poikkeat monista muista ohjelmointikielistä, joten tavoitteena oli myös hankkia itselle vahva perusta Objective-C-ohjelmointikielestä ja Cocos2D-sovelluskehyksestä. Tämän opinnäytetyön kirjoitusvaiheen aikoihin julkaistu iPad oli vielä liian uusi tuttavuus otettavaksi huomioon opinnäytteessä, vaikka se kuuluukin nykyisin Kyy Games Oy:n pelien kohdelaitteistoon. Lukijalle esitellään työkalut ja tekniikat, joita tarvittiin toimeksiannon valmistamiseen, ja millaisessa roolissa ne ovat pelien kehityksessä. Lopuksi tarkastellaan toimeksiantotyötä ja minkälaisia ratkaisuja siinä käytettiin.

Olennaisin osa iPhone OS -sovelluskehityksen ymmärtämisessä on hallita Objective-C-ohjelmointi jollain tasolla, ja tässä opinnäytetyössä annetaan eväät yksinkertaisen ohjelman luomiseen. Lukijalta vaaditaan ohjelmointitaustaa, koska tarkoitus ei ole opettaa ohjelmoinnin perusteita, vaan esitellä Objective-C:n ominaispiirteet.

Objective-C:llä kirjoitetulla Cocos2D-sovelluskehyksellä oli hyvin suuri rooli toimeksiannon peliohjelmointiosuudessa. Cocos2D-sovelluskehyksellä on tarjottavana paljon, ja sen käyttömahdollisuudet tehokkaaseen hyödyntämiseen esitellään kattavasti. Syvemmin perehdyttäessä rakenne ja toimintamalli avataan, jotta lukija ymmärtää käsitteet ja logiikan sovelluskehyksen käyttöön. Cocos2D-sovelluskehys sopii erityisesti 2D-pelien kehittämiseen, ja sen tärkeimmät osat käydään läpi.

Toimeksiannon valmistamiseen tarvittavien työkalujen ja tekniikoiden tutustumisen jälkeen lukijalla on tarvittava tieto toimeksiantona toteutettujen QuestEngine- ja karttatilasovellusosien esittelyn ymmärtämiseen. QuestEnginen luomisen haasteina olivat vaatimukset täyttävän sopivan tietorakenteen luominen ja erityisesti erilaisten operaatiotapahtumien hallinnoiminen. Karttatilan rakentamisessa taas suurimmat haasteet aiheutuivat itse pelaajalle näkyvän graafisen osan hallinnoimisessa, jossa myös Cocos2D-sovelluskehystä käytetään paljon hyödyksi. Näin ollen toimeksiantotyössä lähestytään peliohjelmointia laajalta alalta ja tutustutaan lopputuotokseen kaikkine ongelmia aiheuttaneisiin vaiheisiin.

Opinnäytetyön lähdeaineistossa on pitkälti hyödynnetty internettiä, koska aiheeseen liittyvää painettua kirjallisuutta oli hyvin rajallisesti saatavilla ja lopulta ajankohtaisin tieto löytyy juuri internetlähteistä. Internetin käyttö lähdetiedon hankintaan oli osaltaan myös ainut vaihtoehto.

Applella on tarjottavanaan laaja sovelluskehittäjille suunnattu Apple Developer -sivustonsa, jossa on tarjolla kattava määrä tietoa Objective-C:stä, mutta sen valtavan tietomäärän keskellä oikeiden asioiden löytyminen voi olla hankalaa. Applen dokumentaatioista loistaa selkeästi niiden suuntaus kokeneille sovelluskehittäjille, ja aloittelevan henkilön voi olla vaikea perehtyä niihin. Materiaali on todella laadukasta ja selkeää, mutta sen massiivisuuden vuoksi käyttö voi olla hankalaa.

Cocos2D-sovelluskehys oli tämän opinnäytetyön kirjoitusvaiheessa vielä kehitysasteella, ja siihen liittyvä dokumentaatio oli vähintään yhtälailla keskeneräinen. Tärkein tietolähde löytyy sovelluskehyksen kotisivuilta, jonne uutta materiaalia lisätään pieniä määriä kerrallaan. Sivustolta ei kaikkea tarvittavaa ainakaan toistaiseksi löydy, ja toisinaan on tehtävä tutkimustyötä aina luokkakuvauksista asti.

2 Applen iPhone ja iPod Touch

Yhdysvaltalainen suuryhtiö Apple perustettiin vuonna 1976, jolloin yhtiötä kutsuttiin nimellä Apple Computer. Yhtiön nimi viittasi alkuaikojen tietokoneiden tuotantoon, joka myöhemmin muutettiin pelkäksi Appleksi tuotevalikoiman monipuolistuessa. Nykyään Apple tunnetaan monista multimedialaitteistaan, joista suosituimpia ovat mm. Macintosh-tietokoneet, iPod-musiikkisoittimet ja nykyisin ympäri maailmaa hyviin myyntilukuihin yltäneet iPhone-älypuhelimet (Apple inc. 2010).

Otetaan tarkempaan käsittelyyn iPhone ja iPod Touch, joista Apple julkaisi ensimmäiset mallit vuonna 2007. Nämä ovat myös ne laitteet, joita tarkastellaan pelilaitteina pelisovelluskehityksen näkökulmasta. iPhone ja iPod Touch ovat langattomia taskukoon multimedialaitteita, joihin on saatavilla paljon viihdesisältöä ja myös hyötykäyttöön soveltuvia ohjelmia. Suurin ero näiden kahden laitteen välillä on iPhonessa olevat puhelimen ominaisuudet ja kamera, joita voi myös käyttää hyödyksi peleissä. Pääpiirteittäin laitteet ovat kuitenkin hyvin samankaltaiset aina muotoilusta lähtien, ja eri malleissakin eroavuudet löytyvät pääosin vain muistin määrästä ja tehokkuudesta. Kuvassa 1 on esimerkki iPhone- ja iPod Touch-laitteiden muotoilusta.

Kuva 1. Vasemmalla iPod Touch ja oikealla iPhone

Applella on tapa nimetä laitteet generaation mukaan, mikä yksinkertaisuudessaan tarkoittaa, monennenko kehitysaskeleen laitemalli on kyseessä. Esimerkiksi ensimmäisen generaation iPhone on aivan ensimmäinen julkistettu iPhone-puhelinmalli ja siitä uudempi päivitetty malli kasvattaa generaation lukuarvoa aina yhdellä. Tämän opinnäytetyön kirjoitushetkellä Apple on julkistanut kolme generaatiota iPhone- ja iPod Touch -laitteista.

iPhone ja iPod Touch voidaan liittää tietokoneeseen USB-portin kautta laitteiden mukana toimitettavan johdon avulla. Tällöin iPhone- ja iPod Touch -laitteiden sisältöä, kuten musiikkia, sovelluksia ja käyttöjärjestelmän päivityksiä, päästään käsittelemään iTunes-ohjelmalla. iTunesiin on integroitu verkkokauppa, jossa kaikki Applen hyväksymä lisämateriaali myytävien ja ilmaisten tuotteiden muodossa. Mikäli verkkoyhteys on saatavilla, voi ostoksia tehdä suoraan laitteelta ja kytkentää tietokoneeseen ei näin tarvitse tehdä. Apple vaatii, että kaikki ladattava sisältö tarkistetaan ja luokitellaan heidän toimestaan jo ennen kauppapaikkaan lisäämistä. Kaikki myynnissä olevat tuotteet ovat näin ollen tarkistettu ja todettu soveliaaksi sisällöksi niin Apple yhtiölle kuin iPhone- ja iPod Touch -käyttäjille. Lisäksi myynti on keskitetty Applen järjestämälle palvelulle, jonka levitysalue on hyvin suuri. Esimerkiksi Nokialla on vastaavanlainen kauppapaikka, mutta kilpailijat ovat vielä kaukana App Storen suosiosta.

2.1 iPhone OS -käyttöjärjestelmä

iPhone ja iPod Touch käyttävät molemmat iPhone OS -käyttöjärjestelmää, jonka uusin julkaistu versio on 3.2 tämän opinnäytetyön kirjoitushetkellä ja samalla ensimmäinen iPad-yhteensopiva versio. Ensimmäisen generaation iPhonessa on iPhone OS -versio 1.0, kun taas hieman myöhemmin julkaistussa ensimmäisen generaation iPod Touchissa on käytössä iPhone OS versio 1.1. iPhone OS -versiopäivitykset saa helposti asennettua iTunes-ohjelmiston avulla. Päivitykset ovat ladattavissa iPhone -puhelimeen ilmaiseksi, mutta iPod Touchin omistajat saavat laitteeseensa päivitykset vain maksua vastaan.

Tällä hetkellä Applen tuoteperheessä on kolme laitetta, iPhone, iPod Touch ja iPad, joissa on iPhone OS -käyttöjärjestelmä. iPad eroaa suuren kokonsa puolesta iPhonesta ja iPod Touchista, mutta perusperiaatteet ovat samat, ja iPhone OS:n käyttö on samanlaista jokaisella laitteella. iPhone OS -käyttöjärjestelmän käyttöliittymä perustuu erilai-

siin kosketustapahtumiin, kuten painallukseen, pyyhkäisyyn, kahdella sormella tehtävään nipistysliikkeeseen jne. Käyttäjät odottavat samoja käyttöliittymä kontrolleja myös kolmannen osapuolen sovelluksilta.

Uusien iPhone OS -versioiden myötä laitteiden käyttömahdollisuuksia on kasvatettu. Ainakaan tällä hetkellä iPhone OS ei tue moniajoa, mikä on monelle käyttäjälle suuri puute, mutta tulevaisuudessa tämäkin puute voi korjaantua uuden iPhone OS -version myötä. Moniajon puute tarkoittaa sitä, että samaan aikaan ei voida suorittaa useaa erillistä sovellusta, muutamaa poikkeusta lukuun ottamatta. Esimerkiksi pelaamisen aikana puheluun vastaaminen lopettaa suorituksessa olleen pelin (Costello 2010).

2.2 iPhone ja iPod Touch pelilaitteena

iPhonea tai iPod Touchia ei alun perin suunniteltu pelilaitteiksi, eikä niitä myöhemminkään ole julistettu nimenomaan sellaisiksi. Alun perin laitteille ei voinut asentaa ulkopuolisten kehittäjien sovelluksia, mikä myöhemmin muuttui iPhone OS 2.0 julkaisun myötä (Costello 2010). Erilaisia sovelluksia alettiin julkaista, ja niiden joukossa oli myös paljon kaivattuja pelejä. Yksilöllisiä ja uniikkeja pelejä on nykyään julkaistu valtava määrä monesta eri peligenrestä, mikä nostaa iPhone OS:n potentiaaliseksi pelialustaksi langattomien multimedialaitteiden maailmassa. Valtaosa peleistä on kehitetty ainoastaan iPhone OS:lle, joka osaltaan tekee pelitarjonnasta mielenkiintoisen.

Pelilaitteina ajateltuna iPhonessa ja iPod Touchssa on puutteita, mutta myös paljon hyvää, jota voi puolestaan hyödyntää peleissä. Ensinnäkin laitteissa ei ole kiinteää näppäimistöä, vaan kaikki käyttökontrollit on toteutettavissa kosketusnäytön avulla. Kosketusnäyttö voi olla upea mahdollisuus tai valtava heikkous riippuen pelityypistä. Jos peliin halutaan erityisesti painikkeita, täytyy ne piirtää ruudulle, mikä puolestaan vie tilaa näytöltä. Painikkeiden sijoittelussa on käytettävä järkeä, niin ettei käsi ole kokoajan ruudun edessä, jos painike halutaan aktivoida.

Pelaamiskokemus muuttuu selvästi, jos verrataan pelaamista näppäimillä, kuin mitä se olisi pelkästään kosketusnäytön avulla. Painiketta painamalla pelaaja tuntee painalluksen, jota vastaavaa tunnetta ei kosketusnäytöllä saada aikaan. Näppäimistön edut tulevat helposti ilmi esimerkiksi peleissä, joissa tarvitaan ns. ristiohjainta eli jokaiselle suunnal-

le, ylös, alas, vasen, oikea, on oma painikkeensa. Nopeatempoiset ristiohjaimella pelattavat pelit toimivat kiinteällä näppäimistöllä, mutta samanlainen toteutus kosketusnäytöllä voi olla pelaajalle jopa tuskallista. Ongelma piilee siinä, ettei kosketusnäytöllä saada niin konkreettista painalluksen tunnetta aikaan kuin kiinteällä näppäimistöllä, ja tasaisella näytöllä voi tulla ohipainalluksia, vaikka sormi näyttäisi olevan täysin kohdallaan. Toinen oleellinen seikka on sormien pitäminen irti kosketusnäytöltä, kun taas näppäimistöllä sormia voi pitää valmiina napin päällä kosketusetäisyydellä.

Erillisiä näppäimiä ei välttämättä tarvita ollenkaan, vaan painallus voi kohdistua johonkin näytölle piirrettyyn objektiin. Kosketusnäyttö on elementissään, kun ruudulta valitaan tietty alue, jokin objekti on raahattava näytöllä toiseen sijaintiin tai valintoja tehdään nopeasti eripuolella ruutua. Raahauksella tarkoitetaan painalluksen aloittamista näytöllä, jonka jälkeen sormea liikutetaan näytöllä ja kosketus lopetetaan toisessa pisteessä kuin aloitettiin. Kiinteä näppäimistö on hyvin rajoittunut tällaisessa toiminnassa ja epäsymmetrisessä järjestyksessä olevat objektit ovat vaikeasti valittavissa.

Tehokkuudeltaan iPhone- ja iPod Touch -laitteet ja niiden eri generaatiot sijoittuvat mobiililaitteiden ylempään kastiin. Peleissä voidaan käyttää laitekokoon nähden näyttävää grafiikkaa kärsimättä ruudunpäivitysnopeudesta. Jos omasta pelistä halutaan jokaiselle iPhonelle ja iPod Touchille sopiva, tulee tällöin käyttää testauksen mittarina ensimmäisen generaation laitteita.

Sovelluskehityksen näkökulmasta iPhonen ja iPod Touchin merkittäviä etuja muihin mobiililaitteisiin nähden ovat muun muassa saumaton yhteensopivuus eri mallien välillä, tehokkuus ja rakenteellisten ominaisuuksien samankaltaisuus. Koska iPhone- ja iPod Touch -laitteet ovat hyvin samanlaiset, on sen tuoma etu kehitystyössä suuri. Suurimmat ongelmat pelien kehittämisestä useille matkapuhelimille johtuvat juuri niiden erilaisuudesta ja vaihtelevista ominaisuuksista.

3 Objective-C-ohjelmointikieli pähkinänkuoressa

Objective-C:n kehitystyö alkoi 1980-luvulla Brad J. Cox johtamana. Myöhemmin Apple osti Objective-C:n omistuksen ja on nykyään myös kielen pääasiallinen kehittäjä. Kuten nimi Objective-C kertoo, perustuu se C-kieleen, jota on laajennettu erillisellä oliolaajennuksella. Oliolaajennukseen on otettu pitkälti vaikutteita Smalltalk-ohjelmointikielestä, joka on yksi ensimmäisistä olio-ohjelmointikielistä. Apple käyttää myös Objective-C-ohjelmointikieltä omissa tuotteissaan, kuten Mac OS X - ja iPhone OS -käyttöjärjestelmissä (Tenon Intersystems 2010).

Valtaosa iPhone OS -luokkakirjastoista on kirjoitettu Objective-C-kielellä ja siksi sen osaaminen on tärkeää iPhone-sovelluskehittäjälle. Seuraavaksi esitellään Objective-C:n tärkeimpiä asioita ja paljon esiintyviä ominaispiirteitä.

3.1 Syntaksin perusteet

Pohjimmiltaan Objective-C:n syntaksi on kuin missä tahansa muussa ohjelmointikielessä, mutta silti sen oma persoonallinen muoto saa siihen tottumattoman ohjelmoijan ihmetyksen valtaan. Hakasulkeita käytetään runsaasti ja pistenotaatiotakin voi esiintyä siellä täällä, mutta näiden merkitys voi olla täysin toinen, mihin on kenties totuttu muita ohjelmointikieliä käyttäessä. Seuraavaksi käydään läpi perusteita syntaksin ymmärtämiseksi.

Objective-C:n tyyppinen metodikutsu erotetaan aina hakasulkeiden [ ] sisälle ja monen muun ohjelmointikielen tapaan koodirivi päätetään ;-merkillä. Mikäli ohjelmakoodissa esiintyy sisäkkäisiä hakasulkeita, on tämä merkki useasta metodikutsusta samassa lauseessa (Apple inc. 2009a, 15). Metodien suoritusjärjestys alkaa sisimmästä samaan tapaan kuin laskutoimituksissa. Yksinkertaisimmillaan metodikutsu on kaksiosainen, siinä on vastaanottajaolio ja viesti eli vastaanottajan metodi. Mikäli kutsutaan olion omaa metodia, käytetään tällöin NSObject-luokan määrittämää self-ilmentymämuuttujaa, joka viittaa olioon itseensä. Koodiesimerkissä 1 esitellään metodien kutsuminen yksinkertaisimmillaan.

```
// Yksinkertainen metodikutsu.
[olio metodinNimi];

// self viittaa itseensä eli self-olio kutsuu omaa metodia ”metodinNimi”.
[self metodinNimi];

// Sisäkkäiset metodikutsut. Ensin haetaan kutsutaan olion metodia
// ”palautaOlio2”, joka palauttaa toisen olion, jonka metodia ”metodinNimi”
// kutsutaan.
[[olio palautaOlio2] metodinNimi];
```

Koodiesimerkki 1. Perusidea Objective-C:n metodikutsuista

Mikäli metodi odottaa parametria, kirjoitetaan annettava parametri metodin nimen jälkeen. Jokainen parametri erotetaan kaksoispisteellä (;) aina edellisestä lauseen osasta ja samaa sääntöä noudatetaan, vaikka parametreja annettaisiin enemmän kuin yksi. Objective-C:ssä on mahdollista ja myös suositeltavaa kirjoittaa ennen kaksoispistettä parametria kuvaava avainsana. Koska metodin nimi on ennen ensimmäistä parametria, tulisi sen olla osaksi myös parametria kuvaava avainsana. Nämä avainsanat eivät ole pakollisia ohjelmakoodissa, mutta niiden käyttö helpottaa koodin lukemista huomattavasti. Parametrin välittäminen metodeille on esitelty Koodiesimerkissä 2.

```
// Metodille välitetään parametri merkkijono.
[olio tulostaMerkkijono:merkkijono];

// Usean parametrin välittäminen. Metodille välitetään nyt kaksi parametriä
// merkkijono ja merkkijono2. Ennen kaksoispistettä ja merkkijono2:sta on
// toista parametriä kuvaava avainsana.
[olio tulostaMerkkijono:merkkijono toinenMerkkijono:merkkijono2];
```

Koodiesimerkki 2. Metodien kutsuminen parametrien kanssa

3.2 Tietotyypit

Objective-C ohjelmoinnissa voidaan käyttää kaikkia standardeja C-kielen tietotyyppejä, joiden lisäksi käytettävissä on myös Objective-C:n omat tietotyypit. Ehtolauseissa käytetään Objective-C:n tapaan *BOOL*-tyypin *YES* ja *NO* vakio-arvoja, kun taas monessa muussa ohjelmointikielessä on totuttu käyttämään *boolean*-tyyppisiä vakioarvoja *true* ja *false*. Samoin muissa ohjelmointikielissä on totuttu käyttämään olemattomiin olioihin

null-osoittimia, jotka ovat Objective-C:ssä nimetty *nil*-osoittimiksi (Apple inc. 2009a, 113).

Luultavasti ensimmäisenä silmään pistää *id*-tietotyyppi, jonka käyttö vaikuttaa hyvin vapaalta ja moni metodi palauttaa sen tyyppisen olion. Tietotyyppinä id ei kerro siihen sijoitetusta oliosta juurikaan muuta, kuin sen olevan olio. Sillä on kaikki olion operaatiot käytettävissä, eikä mitään muuta ennen tyyppimuunnosta. Objective-C:ssä saa rakennettua helposti geneerisiä sovellusrakenteita ja ajonaikaisia valintoja dynaamisen oliotyypityksen ansiosta. Muun muassa *NSArray*-taulukkoon voidaan lisätä mitä tahansa olioita ja myöhemmin etsiä haluttu olio sen tyypin perusteella.

Selector-tyyppi eli Objective-C:ssä *SEL*, on viite metodin nimeen. Luvussa 5 esiteltävässä Cocos2D-sovelluskehyksessä sitä käytetään paljon, ja siksi on hyvä ymmärtää sen toiminnan tarkoitus. Metodi voi ottaa vastaan parametrina *SEL*-viitteen, jonka avulla tämän metodi voidaan ohjata kutsumaan haluttua metodia. Kyseessä voi siis olla mikä tahansa olio, jolla on kyseinen metodi. *Selectorit* ovat erityisen käytännöllisiä ns. *callback*-metodia kutsuttaessa jonkin tapahtuman päättyessä.

3.3 Luokat

Objective-C:n lähdekoodi suositellaan jaettavan kahteen eri tiedostoon, joita ovat .m-päätteinen toteutusosa ja .h-päätteinen esittelyosa. Nämä kaksi tiedostoa täydentävät toisiaan, ja ne ovat kuin kaksi palasta yhdestä kokonaisuudesta. Tarkoituksena on tehdä lähdekoodin käsittelystä hallittavampaa. Esittelyosasta (.h) nähdään kaikki tarvittava tieto luokan käyttöä ajatellen, mikä on toteutettuna toisessa tiedostossa eli toteutusosassa (.m). Samaan yhteyteen kuuluvat luokat voidaan kirjoittaa samoihin toteutus- ja esittelyosiin, mutta liian suureksi paisuvat tiedostot ovat hankalia käsitellä. (Apple inc. 2009a, 35).

Esittelyosassa esitellään luokka, periytyminen, muuttujat, metodit, propertyt ja vakiot. Ainoastaan vakioille annetaan arvot ja luokan ilmentymämuuttujia ei alusteta. Metodeistakaan ei esitellä muuta kuin niiden kutsuosat. Esittelyosaan tehdään ulkopuolisiin luokkiin viittaavat kirjastotuonnit, jotka täydentävät käytettävien luokkien määrää. Kirjastotuonteja täytyy tehdä, jos luokassa käytetään esimerkiksi toisesta luokasta luotua

oliota ilmentymämuuttujana. Kirjastotuonnit tulisi tehdä suosituksen mukaan *#import*-esikäännöksen avulla *#includen* sijaan, mikä estää saman luokan tuonnin moneen kertaan. *#import* viittaa aina haluttavan luokan esittelyosaan, ja kyseisen luokan esittelyosa on vain yhteydessä toteutusosaan. Esittelyosaan pitäisi myös kirjoittaa metodia kuvaavat kommentit, jotta luokan käyttö olisi mahdollista ilman toteutusosan näkemistä. Koodiesimerkissä 3 on esitelty, millainen on luokan esittelyosan perusrakenne.

```
// Laajennetaan käytettävien luokkien määrää EsimerkkiLuokka:lla ja
// EsimerkkiLuokka2:lla.
#import EsimerkkiLuokka.h;
#import EsimerkkiLuokka2.h;

// Luokan esittely alkaa.
// Tämä luokka periytetään NSObject-luokasta ja saa näin käyttöönsä sen
// ominaisuudet.
@interface LuokanNimi : NSObject
{
        // Ilmentymämuuttujien esittely.
        int arvo;
        NSString* merkkijono;
}
// Metodien esittely.
-(id)init;
-(void)metodi;

// Luokan esittely päättyy.
@end
```

Koodiesimerkki 3. Luokan perusrakenne esittelyosassa

Toteutusosaan kirjoitetaan metodit ja niiden toteutus. Toteutusosa liitetään esittelyosaan jo edellä mainitulla *#import*-esikäännöksellä, joka lisätään aina toteutusosan puolelle. Metodien kutsuosat tulee olla täysin samanlaiset esittelyosan metodiesittelyjen kanssa, muuten kääntäjä tulkitsee ne eri metodeiksi. Tällöin kääntäjä antaa varoituksen, ettei luokasta välttämättä löydy käytettävää metodia, vaikka kyseistä metodia kutsuttaisiin samassa esittelyosassa. Koodiesimerkissä 4 on Koodiesimerkin 3 esittelyosaa täydentävän toteutusosan runko.

```
// Toteutusosan liitos luokan esittelyosaan.
#import ”LuokanNimi.h”

// Luokan toteutusosan määrittely alkaa.
@implementation LuokanNimi
-(id)init
{
        // Metodin toteutus.
}
-(void)metodi
{
        // Metodin toteutus.
}

// Määrittely päättyy.
@end
```

Koodiesimerkki 4. Vastaava toteutusosa Koodiesimerkki 3:n luokalle

3.4 Metodit

Olio-ohjelmoinnin mukaisesti sovellus rakentuu olioista ja niiden yksittäisistä operaatioista eli metodeista. Metodi on pieni suoritettava ohjelmalohko, jolla on tietty tehtävä. Objective-C:ssä on kahdenlaisia metodityyppejä, instanssimetodeja ja luokkametodeja. Instanssimetodi on käytettävissä vain kyseisestä luokasta luodun olion kautta. Luokkametodi taas on näkyvyydeltään avoin ja irrallinen oliosta, mikä tarkoittaa sen käytön olevan mahdollista kutsumalla metodia luokan nimen mukaan. Metodin esittelyssä instanssimetodi merkitään –(miinus) - ja luokkametodi +(plus) -merkillä. Metodit suorittavat joko toimenpiteen ja antavat palautusarvon tai tekevät vain toiminnon ilman palautusarvoa. Metodin esittelyssä *void* tarkoittaa palautusarvotonta metodia, ja kaikki muut tyypit määrittelevät palautettavan arvon tai olion tyypin.

Metodin esittely rakentuu vähintään kolmesta pakollisesta osasta. Esittelyssä ensimmäisenä luettaessa vasemmalta oikealle määritellään, onko metodi instanssimetodi vai luokkametodi, kaarisulkeiden sisään metodin palautusarvon tyyppi ja viimeiseksi metodin nimi. Edellä mainittu on metodin runko, jota jatketaan, mikäli metodin halutaan vastaan ottavan parametreja. Jokainen parametri esitellään kaksoispisteen jälkeen, jossa määritellään parametriarvon tyyppi ja nimi. Ennen kaksoispistettä suositellaan käytettävän erillistä parametrin kuvausta, joka yhden parametrin tapauksissa pitäisi sisällyttää metodin nimeen. Ideana on helpottaa kirjoitetun koodin luettavuutta, kun metodin kut-

susta nähdään heti kuvaus lähetettävästä arvosta. Koodiesimerkki 5 selventää instanssi- ja luokkametodin eron.

```
// Miinusmerkki kertoo metodin olevan instanssimetodi ja vain olion kautta
// käytettävissä.
-(float)laskeSumma:(float)luku1 luku2:(float)luku2;

// Metodin kutsu:
[olio laskeSumma:1.0 luku2:1.0];

// Edellistä metodia vastaava luokkametodi. Luokkametodi ei ole kytköksissä
// olioon, joten se ei pääse käsiksi olion ilmentymämuuttujiin.
+(float)laskeSumma:(float)luku1 luku2:(float)luku2;

// Metodin kutsu:
[Luokka laskeSumma:1.0 luku2:1.0];
```

Koodiesimerkki 5. Instanssi- ja luokkametodin rakenne ja niiden ero kutsuttaessa

Varsinaisen muodostimen puuttuessa Objective-C-kielestä, tehdään olion alustus init-metodissa. Yläluokkien alustus varmistetaan *init*-metodissa kutsumalla superoperaatiolla yläluokan *init*-metodia, joka taas jatkaa seuraavan yläluokan alustusmetodiin jne. Näin varmistetaan, että koko olion periytymishierarkia alustetaan oikeaoppisesti. Koodiesimerkissä 6 on suositeltu tapa kirjoittaa *init*-metodi, jossa täytyy aloittaa myös yläluokkien alustusketju.

```
// Muodostimen korvaava alustusmetodi.
-(id)init
{
        // [super init] aloittaa yläluokkien alustuksen, jonka onnistuessa
        // alustetaan olion muuttujat.
        if( (self=[super init]) ) {
                // Olion alustus.
        }
        return self;
}
```

Koodiesimerkki 6. Olion alustukseen käytettävä init-metodi

3.5 Aksessorimetodit yksinkertaisesti property:n avulla

Objective-C:ssä on yksinkertainen tapa luoda aksessorimetodit luokan ilmentymämuuttujille, joita kutsutaan propertyksi. Olio-ohjelmoinnin periaatteen mukaisesti ulkopuolisen toimijan tulisi käsitellä olion muuttujia aina metodin kautta, ja yleensä tämä onkin ainoa mahdollisuus olion suojauksen vuoksi (Koskimies 2000, 50). Kääntäjä luo propertyksi esitellylle muuttujalle määritetyn mukaiset aksessorimetodit. Propertyn luonti on paitsi helppoa, se säästää koodirivejä luokassa ja samalla aksessorimetodeja voidaan kutsua yksinkertaisella myös piste-notaatiolla.

Luokan ilmentymämuuttujat esitellään normaalisti luokan esittelyosassa, minkä lisäksi metodien esittelyosassa määritellään ilmentymämuuttujien propertyt. Propertyn määrittely on kolmiosainen ja se alkaa *@property*-avainsanalla, jonka jälkeen määritellään kaarisulkeiden sisällä attribuutit sekä lopuksi ilmentymämuuttujan tyyppi ja nimi. Kun property on esitelty luokan esittelyosassa, täytyy se myös osoittaa luokan toteutusoasassa *@synthesize*-avainsanalla. *@synthesize* ohjaa kääntäjän luomaan aksessorimetodit esittelyosassa sijaitsevien propertyjen mukaisesti.

Se mitä aksessorimetodeja kääntäjä luo propertysta, määritellään kahdella attribuutilla *readwrite* ja *readonly*. Oletusarvoisesti property on *readwrite*-tyyppinen, mikä käskee kääntäjän luomaan ilmentymämuuttujalle sekä asetus- että palautusmetodit. Toinen vaihtoehto on määritellä property *readonly*-tilaan, jossa ilmentymämuuttujalle tehdään vain arvon palautusmetodi. (Apple inc. 2009a, 59).

Kirjoitusoikeuden omistavan propertyn käyttäytymistä on mahdollista muokata *assign*-, *retain*- ja *copy*-attribuuteilla. Näillä vaikutetaan Objective-C tyyppisiin olio-ilmentymämuuttujiin ja niiden muistinhallinnallisiin tekijöihin. Oletuksena asetusmetodi on assign-tyyppinen eli se on yksinkertainen sijoitus ilmentymämuuttujaan ilman erityisiä toimenpiteitä. Sekä *retain* että copy vapauttavat aiemman ilmentymämuuttujan olion muistista ennen uuden asettamista. *retain* kohottaa ennen asettamista olion retainCount-arvoa yhdellä ja lisää näin sen muistinvarauksen elinikää. Oliosta tehdään kopio *copy*-attribuuttimäärityksellä. Koodiesimerkissä 7 on esitelty, kuinka ilmentymämuuttujille lisätään property-määre.

```
// Esittelyosassa luokalla on normaalisti ilmentymämuuttujia.
@interface Luokka : NSObject
{
        int maxArvo;
        NSString* teksti;
}
// Ilmentymämuuttujille lisätään property-määreet.
@property (readonly, assign) int maxArvo;
@property (readwrite, retain) NSString* teksti;

@end

// Luokan toteutusosa
@implementation Luokka

// Kääntäjälle ilmoitetaan propertyista.
@synthesize maxArvo, teksti;

@end
```

Koodiesimerkki 7. Luokan ilmentymämuuttujat määritellään propertyksi, jolloin kääntäjä luo niille aksessorimetodit.

Ohjelmakoodin luettavuus kärsii tapauksissa, joissa Objective-C:n syntaksilla kirjoitetaan monta sisäkkäistä aksessorimetodikutsua. Propertyä voidaan vaihtoehtoisesti käyttää pistenotaatiolla, mikä tekee propertyn kutsusta selkeämmän. Pistenotaatiota ei tule sotkea tavallisiin metodikutsuihin, vaan se on käytössä vain ja ainoastaan propertyn yhteydessä. Koodiesimerkissä 8 näytetään, miten propertyä kutsutaan molemmilla tavoilla.

```
// Sama operaatio Objective-C metodikutsuna ja pistenotaatiolla.
// Asetusmetodin käyttö
[olio setArvo: uusiArvo];
olio.arvo = uusiArvo;

// Palautusmetodin käyttö
int tempArvo = [olio arvo];
int tempArvo = olio.arvo;

// Useita sisäkkäisiä aksessorimetodikutsuja
[[[olio oliomuuttuja] oliomuuttuja2] setArvo: 10];
olio.oliomuuttuja.oliomuuttuja2.arvo = 10;
```

Koodiesimerkki 8. Propertyn kutsuissa on mahdollista käyttää Objective-C syntaksia tai pistenotaatiota.

3.6 Muistinhallintaa Objective-C:n tapaan

Objective-C on pohjimmiltaan C-kieli ja sen muistinhallinta on myös C-kielestä peräisin. Lähtökohtainen asetelma on, että ohjelmoija on vastuussa varaamastaan muistista. iPhone OS ei tarjoa ohjelmoijalle automaattista ”roskien kerääjää”, jollainen on mm. Mac OS X -käyttöjärjestelmässä, joten muistinhallintaan on syytä kiinnittää huomiota iPhone OS -ohjelmoinnissa muistivuotojen välttämiseksi. Muistivuodosta puhutaan silloin, kun ohjelman suorituksen aikana varatun muistipaikan osoite kadotetaan ja muistialueeseen ei päästä enää käsiksi. Muistivuodot täyttävät muistia turhaan ja voivat lopulta aiheuttaa muistin loppumisen, jolloin sovellus niin sanotusti kaatuu eli sen suorittaminen lopetetaan.

Muistinhallinnan kokonaiskuvan ymmärtämiseksi on katsottava pintaa syvemmälle. Sovelluksen käyttämä muisti jakautuu eri muistisegmenteille, joita ovat mm. stack eli pinomuisti ja heap eli kekomuisti. Paikalliset muuttujat ja metodien parametrit säilötään pinoon ja ne poistuvat heti kun ohjelmalohko päättyy. Ohjelmoijan ei tarvitse huolehtia erityisestä pinomuistin varaamisesta tai vapauttamisesta.

Objective-C:ssä muistinhallintaan liittyy neljä metodia: *alloc*, *release*, *retain* ja *autorelease*, jotka on määritelty NSObject-luokassa. Heap-muistista varataan oliolle *alloc*-metodilla ja vastaavasti vapautetaan *release*-metodilla. *retain*-metodin tarkoitus on jatkaa olion elinikää suurentamalla olion retainCount-ilmentymämuuttujan arvoa. Jos tutkitaan olioiden luomis- ja vapautusmekaniikkaa hieman lähemmin, huomaamme, että NSObject-luokassa on määritelty oliolle retainCount-muuttuja, joka määrittää, milloin olio vapautetaan muistista. Kun retainCount saavuttaa arvon 0, vapautetaan olion muistinvaraus. *autorelease* on hieman erikoisempi tapaus. Kun olio saa *autorelease* kutsun, lisätään sen osoitin niin kutsuttuun autorelease-altaaseen, johon asetetut oliot vapautetaan määrittelemättömästi myöhemmin. Käytännössä vastuu olion vapauttamisesta siirretään pois itseltä ja samalla kasvatetaan olion elinkaarta. Koodiesimerkissä 9 esitellään, miten muistinvaraus käytännössä toimii.

```
-(void)metodi:(int)arvo
{ // Ohjelmalohko alkaa.
	// NSString-osoitin ja olio tallentuvat pinomuistiin.
	NSString* teksti1 = @"teksti1";

	// teksti2, teksti3 ja teksti4 olioiden osoitin tallentuu
	// stack-muistiin ja olio heap-muistiin.
	NSString* teksti2 = [[NSString alloc] init];
	NSString* teksti3 = [[NSString alloc] init];
	NSString* teksti4 = [[NSString alloc] init] autorelease];

	if (YES) { // Esimerkin vuoksi ehto on aina tosi.
		// testi-muuttuja esitellään ja alustetaan arvo parametrilla.
		int testi = arvo;
	}
	// testi-muuttujaan ei enää voida viitata, koska se on poistunu stack-
	// muistista heti lohkon päättyessä.

	// teksti3 muistinvaraus poistetaan heap-muistista ja osoitin
	// nollataan varmuuden vuoksi.
	[teksti3 release];
	teksti3 = nil;
} // Metodin ohjelmalohko päättyy ja teksti2 aiheuttaa muistivuodon, koska sen
// osoitin katoaa ohjelmalohkon päättyessä ja sen muistinvaraus jää voimaan.
```

Koodiesimerkki 9. Muistinvarausten pysyvyys ohjelmalohkojen sisällä

Juuri ennen olion vapauttamista muistista suoritetaan roskien keräämisoperaatio, eli järjestelmä kutsuu *dealloc*-metodia, jossa tulisi viimeistään vapauttaa kaikki olion omat muistinvaraukset. Vapautettaessa oliota järjestelmä tutkii, löytyykö oliolta *dealloc*-metodia periytymishierarkian viimeisestä aliluokasta lähtien, ja siirtyen NSObject-luokkaan asti, mikäli *dealloc*-metodia ei ole yliajettu muissa luokissa. Jotta kaikkien yläluokkienkin *dealloc*-metodeja kutsuttaisiin, käytetään *super*-kutsua seuraavaan yläluokkaan. Koodiesimerkissä 10 on yksinkertainen malli *dealloc*-metodin yliajoon.

```
-(void)dealloc
{
	// Luokalla voisi olla "teksti" nsstring-ilmentymämuuttuja, joka
	// täytyy vapauttaa isäntäolion tuhoutuessa.
	[teksti release];
	// Jatketaan dealloc-kutsuketjua seuraavaan yläluokkaan.
	[super dealloc];
}
```

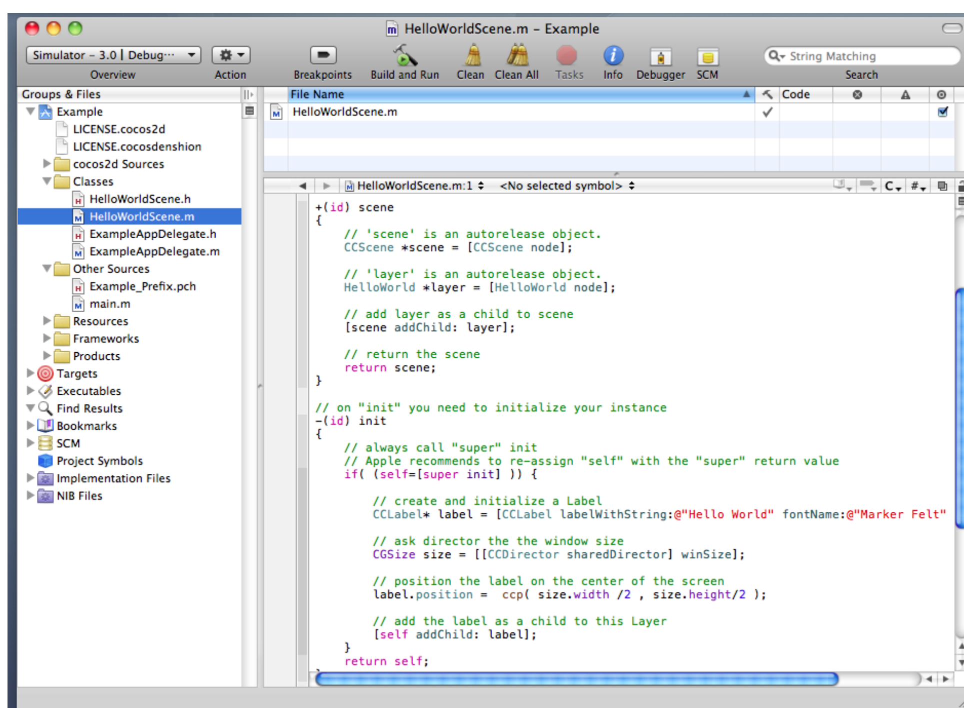
Koodiesimerkki 10. Olion muistinvarausten putsaaminen yliajetussa dealloc-metodissa

4 iPhone SDK

iPhone SDK, joka on lyhenne sanoista Software Development Kit, sisältää kaikki tarvittavat peruslähdekoodikirjastot, Xcode-kehitysympäristön, kääntäjän, dokumentaatiota ja niin edelleen, jota käytetään iPhone OS -sovelluksen kehittämiseen. SDK:n voi ladata ilmaiseksi Applen iPhone Dev Center -sivustolta sinne rekisteröitymisen jälkeen. SDK on käytössä vain Machintosh-käyttöjärjestelmälle, joten iPhone OS -kehittäjän on hankittava tietokoneensa Applelta. Oman sovelluksen julkaisu App Storessa onnistuu vain liittymällä 99$ maksavaan kehittäjäohjelmaan. iPhone SDK sisältää paljon hyödyllisiä työkaluja, joista tärkeimmät esitellään seuraavaksi.

4.1 Xcode-kehitysympäristö

iPhone OS -sovelluskehittäjän kehitystyökalu on nimeltään Xcode, jonka päänäkymä on esitelty Kuvassa 2. Näppärän editorin lisäksi Xcodessa on integroituna kääntäjä, testauslaitteiden hallinta, oikeaa laitetta simuloiva emulaattori (Kuva 3) ja paljon muuta hyödyllistä (Apple inc. 2009b). Xcoden käyttö on helppoa, ja ilmaiseksi kehitystyökaluksi se on varsin laadukas kokonaisuus.



Kuva 2. Päänäkymä Xcode-kehitysympäristöstä

Kuva 3. iPhone-emulaattorilla voidaan nopeasti testata sovellusta

Kääntäjä ymmärtää kaikkia C-, C++- ja Objective-C-ohjelmointikieliä, ja tietyillä toimenpiteillä näitä kaikkia voidaan käyttää samassa ohjelmassa. C ja Objective-C toimivat yhdessä ilman erityisiä temppuja ja Objective-C-ohjelmakoodin seassa näkee monesti käytettävän C-kielen kutsuja. Kun Objective-C:tä käytetään yhdessä C++:n kanssa, puhutaan tällöin Objective-C++:sta. Lähdekoodin toteutusosat nimetään tällöin .mm-päätteellä.

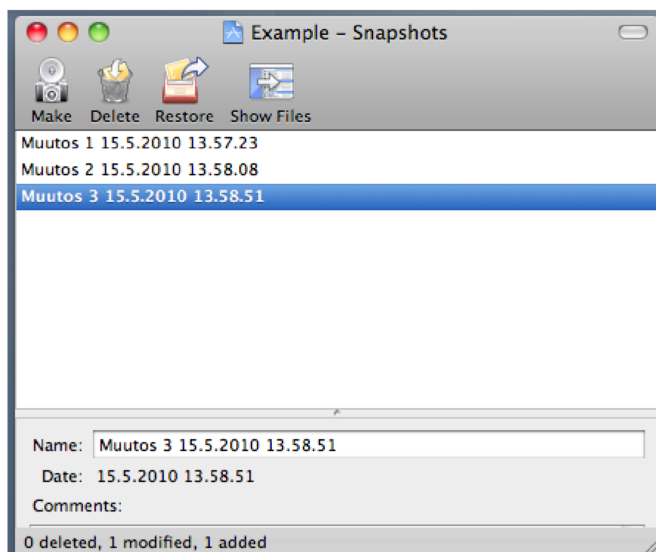
Projektin tiedostot näyttävät jakaantuvan Xcode-näkymässä eri kansioihin, mutta todellisuudessa tällaista kansiolajittelua ei ole. Kaikki uudet tiedostot, kuten kuvat ja luokat on nimettävä projektikohtaisesti uniikilla nimellä. Uusien resurssitiedostojen lisäämisen yhteydessä on muistettava tehdä kyseisestä tiedostosta kopio projektia varten, muuten käytettävään tiedostoon tehdään vain viite projektihakemiston ulkopuolelle. Uuden tiedoston lisäysvaiheessa on mahdollista rastittaa resurssitiedoston kopioiminen projektiin.

Xcoden vahvuuksia on ehdottomasti sen yhteistyö iPhone OS -laitteiden kanssa. Sovellus voidaan kääntää ja suorittaa suoraan laitteessa, milloin aikaa ei tarvitse käsin tehtävään asennustyöhön. Sovelluksen ajon aikana laite lähettää Xcoden konsoliin kaikki ohjelmoidut tulostukset, joita on helppo seurata ja ne eivät häviä, vaikka sovellus kaatuisi. Samoin myös debuggaus-toiminnot on täysin käytettävissä. Sovelluksen testaaminen laitteessa on näin tehty helpoksi ja lähes yhtä nopeaksi kuin emulaattoria käytettäessä

4.2 Versionhallinta

Versionhallinta on lähes välttämätön lisä ohjelmointityössä. Lyhyesti kuvailtuna versionhallinta on aikakone, joka pitää kirjaa muutoksista ja mahdollistaa edellisiin muutoksiin palaamisen. Pienessäkin projektissa versionhallinnan hyöty on suuri, ja sen merkitys kasvaa projektin laajuuden ja työskentelevien henkilöiden lukumäärän mukaan. Versionhallinta hoitaa projektin muutosten liittämisen ja jakamisen, mikä helpottaa ja nopeuttaa projektin parissa työskentelyä.

Xcodessa itsessään on pienimuotoinen versionhallintajärjestelmä nimeltään Snapshot (Kuva 4). Snapshotilla tallennetaan projektin nykytilanne, ja myöhemmin projektin edetessä sen tila pystytään palauttamaan mihin tahansa Snapshotilla otettuun tilanteeseen. Snapshot on hyödyllinen yhden sovelluskehittäjän projekteissa, mutta useamman henkilön tiimin tarpeisiin se ei sovellu.



Kuva 4. Yksinkertainen versiointi onnistuu Snapshotilla

Markkinoilla on tarjolla useita niin kaupallisia kuin avoimen lähdekoodin versionhallintajärjestelmiä. Molemmista kategorioista löytyy varteenotettavia vaihtoehtoja valittavaksi versionhallintajärjestelmäksi. Xcodessa on valmis integroitu tuki Perforce-, CVS- ja Subversion-versionhallintajärjestelmien käyttöön.

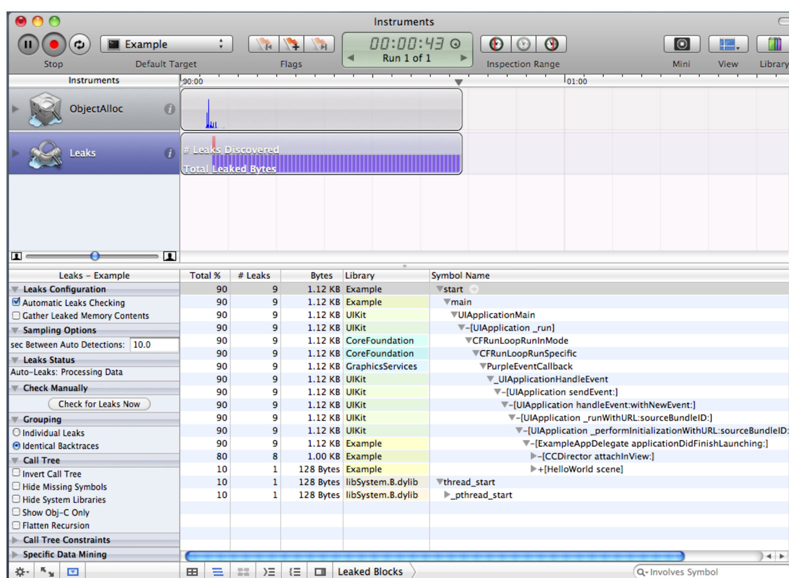
Ulkoista versionhallintaa käytettäessä on muistettava siirtää projektin build-hakemisto pois projektihakemiston alta niin, ettei se kuulu versioitavien tiedostojen joukkoon.

Muutoin sovelluksen käännöksen tuotos build-hakemistosta aiheuttaa jatkuvasti konflikteja versionhallinnassa ja tuottaa tarpeettomia hidasteita versionhallinnan käyttöön.

4.3 Instruments – sovelluksen analysoija

Muistivuodot ovat monen ohjelmoijan riesana ja niiden löytäminen varsinkin omasta ohjelmakoodista on hankalaa. Applen Xcode-kehitystyökalupaketissa on tähän ongelmaan helpotus Instruments-sovelluksen muodossa. Instrumentsin periaate on yksinkertainen; se tutkii suoritettavan sovelluksen muistin varauksia ja merkitsee aikajanalle muistivuodon syntymishetken. Tämän jälkeen kaikki muistivuodot kirjautuvat listaan, josta voidaan tutkia metodikutsu kerrallaan ja näin paikantaa muistivuodon aiheuttava ohjelmakoodi. Instrumentsilla voidaan ajaa myös muita hyödyllisiä testejä, kuten suorituskyvyn analysointia.

Instruments toimii saumattomasti niin emulaattorin kuin oikean laitteen kanssa, tosin testaus kannattaa aina suorittaa oikealla laitteella todenmukaisimman tuloksen saamiseksi. Muistin käyttöä analysoidaan reaaliajassa, mikä paljastaa myös muistivuotojen syntymiseen johtavat tilanteet. Muistivuodon syntymiseen tarvitaan monesti tietynlainen tapahtumien ketju. Sovellusta voi näin ollen testata kokeilemalla erilaisia tilanteita, ja Instruments hoitaa muistivuotojen tarkkailun. Kuvassa 5 nähdään, että Instruments on löytänyt muistivuodon, joka on merkitty punaisella viivalla Leaks-osioon.



Kuva 5. Instruments jäljittää helposti muistivuodot ajettavasta sovelluksesta

5 Cocos2D-sovelluskehys

iPhone OS:n tarjoamat luokkakirjastot sisältävät tarvittavat työkalut yksinkertaisten pelien tekemiseen, mutta peliohjelmointia varten on suositeltavaa ottaa avuksi jokin peliohjelmointiin tarkoitettu sovelluskehys tai hyödyntää olemassa olevaa OpenGL ES -tukea. OpenGL ES:n käyttäminen on suoraan sanottuna vaikeaa ja sen opetteleminen voi olla monelle liian haastavaa. OpenGL ES:n tehokasta käyttöä voidaan hyödyntää Cocos2D-sovelluskehyksellä. Cocos2D tarjoaa helpommin lähestyttävän Objective-C:llä kirjoitetun luokkakirjaston, joka pohjimmiltaan käyttää tehokkaasti OpenGL ES:ää. Alun perin Python:lla kirjoitettu Cocos2D-sovelluskehys on käännetty iPhone OS -yhteensopivaksi ja se on täysin ilmainen avoimen lähdekoodin sovelluskehys (Cocos2D 2010a).

Cocos2D on tarkoitettu 2D-pelien tai interaktiivisten sovellusten tekoon. Sen mukana on integroituna paljon hyödyllisiä komponentteja, kuten CocosDenshion äänien toistamiseen tarkoitettu kirjasto, Box2D- ja Chipmunk-sovelluksessa painovoimaa simuloivat fysiikkamoottorit jne. Ilmaiseksi sovelluskehykseksi sillä on paljon tarjottavanaan. Se ei ole kuitenkaan kaiken kattava paketti. Jos esimerkiksi halutaan tehdä 3D-pelejä iPhonelle, ei Cocos2D:n käyttö ole järkevää. Cocos2D:n mukana ei tule mitään graafista työkalua, jolla saisi esimerkiksi Flash-editorin tapaan aseteltua graafiset elementit paikoilleen, vaan käyttö on täysin ohjelmointilähtökohtainen.

Cocos2D:stä on harmittavan vähän oppaita, joiden avulla aloittelija voisi opetella Cocos2D:n käyttöä. Cocos2D:n internetsivuilta voi ladata kehitystiimin luoman projektin, joka on tarkoitettu sovelluskehyksen testaamiseen, mutta sitä voi myös hyödyntää esimerkkinä Cocos2D:n käyttämiseen. Testit kattavat kaikki sovelluskehyksen ominaisuudet ja lähdekoodia pääsee tutkimaan täysin avoimesti. Cocos2D:n internetsivut ovat pääasiallisin lähde tiedonhankintaan ja erityisesti siellä olevan foorumin voisi nostaa tärkeäksi tietolähteeksi. Jos mieltä askarruttaa jokin kysymys Cocos2D:n käytöstä, on sitä luultavasti joku muu ehtinyt jo kysyä foorumilla, ja aktiivisen yhteisön tai kehitystiimin ansiosta on ratkaisu kehitetty kysymykseen.

Cocos2D:tä käytettäessä sovelluksen pohjana käytetään samaa UIWindow-oliota, kuten normaalin iPhone OS -sovelluksen kanssa. Cocos2D:n oma sovelluksen ohjauskomponentti liitetään UIWindow-olioon, jonka jälkeen käytetään sovelluskehyksen omaa ra-

kennetta sovelluksen hallinnassa. Kaikki ominaisuudet, kuten UIKit-kirjasto, eivät välttämättä ole yhteensopivia Cocos2D:n kanssa, mikä rajaa käytettävien komponenttien käyttömahdollisuuksia.

5.1 Tärkeimmät luokat

Cocos2D-sovelluskehys sisältää laajan kirjon erilaisia luokkia yleisistä erikoistuneempiin käyttötarpeisiin. Sovelluskehyksen kehittäjät muuttivat luokkien nimeämistapaa 0.9-versiosta alkaen niin, että luokan nimi aloitetaan CC-lyhenteellä. Lyhenteellä viitataan yksinkertaisesti Cocos2D:hen. Internetistä saattaa löytyä esimerkkejä, joissa on käytetty vanhempaa Cocos2D:n kehitysversiota, ja niistä puuttuvat uuden nimeämistavan mukaiset CC-lyhenteet. Nämä esimerkit ovat yleisesti ottaen aivan käyttökelpoisia, kun vain lisää CC-lyhenteen olion tyyppeihin. Seuraavaksi esitellään Cocos2D:n luokkia, joita käytetään peliohjelmoinnissa paljon suoraan tai sitten omiin tarpeisiin muokattuna periyttämisen avulla (Ben 2010).

5.1.1 CCNode

CCNode on perusluokka, josta lähes kaikki Cocos2D:n piirtoon liittyvät luokat pohjimmiltaan periytyvät, eli se on perusluokka samaan tapaan kuin NSObject-luokka. CCNode sisältää perustiedot kuten ankkuripisteen, sijainnin, koon jne. Jos mistä tahansa Cocos2D -sovelluskehyksen luokasta voi luoda olion, jota pystyy kääntämään, skaalaamaan tai siirtämään, periytyy se CCNodesta (Ben 2010.)

CCNode sisältää tärkeän parent- ja child-tietorakenteen, mikä on hyvin olennaisessa osassa Cocos2D:tä käytettäessä. Jokainen CCNodesta periytyvä luokka voi olla toisen CCNodesta periytyvän luokan parent tai child. Parent-olio omistaa kaikki childiksi asetetut oliot ja kun parent hävitetään, sovelluskehys poistaa myös kaikki child-oliot. Samaan tapaan child-oliot ovat aina samassa suhteessa parent-olioon ja ne reagoivat parent-olion tapahtumiin. Cocos2D perustuu pitkälti tämän hierarkian käyttöön. Jokainen child-olio saa niin kutsutun z-arvon, mikä tarkoittaa olion syvyystasoa samassa childryhmässä. Mitä suurempi z-arvo childilla on, sitä ylempänä se on piirtojärjestyksessä, toisin sanoen suurempi z-arvon omaava piirretään pienemmän päälle.

CCNodella on schedule-metodi, jolla peliohjelmoinnissa tarvittavan silmukan saa luotua helposti. Schedule-metodille välitetään *selector*-tyyppinen viite oman luokan metodiin, joka tämän jälkeen toimii callback-metodina, eli se ottaa vastaan schedulen kutsut. Oletuksena scheduler tekee callback-kutsun jokaisen frame-päivityksen aikana, jonka pitäisi olla normaalisti 60 kertaa sekunnissa. Kutsutiheyttä voi myös muuttaa haluamalleen nopeudelle.

5.1.2 CCDirector

CCDirector on koko sovelluksen esityksen ohjaaja. CCDirectorilla hallitaan muun muassa sovelluksen ajoa, kuten sen pysäyttämistä, minkä jälkeen voidaan jatkaa sovelluksen suorittamista samasta tilanteesta (Ben 2010). CCDirectorin tärkeimpiin tehtäviin kuuluu myös tilojen eli CCScene-olioiden hallinnoiminen. CCScenejä vaihdetaan aina CCDirectoria kutsumalla ja tähän on tarjolla kaksi vaihtoehtoista tapaa. CCDirector tarjoaa vaihtoehtoisia efektejä tilojen vaihtumisen välillä, jos tilasiirtymään halutaan näyttävyyttä, ja monesti tällaiset pienet yksityiskohdat tuovat peleihin kaivattua eloa.

Ensimmäinen CCScene-olio asetetaan ajoon CCDirectorin *runWithScene*-metodilla, minkä jälkeen tiloja vaihdetaan, joko *replaceScene*- tai *pushScene*-metodilla. Suositeltavampi tapa on korvata ajossa oleva CCScene toisella, jolloin kutsutaan CCDirectorin *replaceScene*-metodia, joka saa parametrikseen uuden CCScenen. Vanha CCScene poistetaan kokonaan muistista, joten samaan tilaan ei voida enää palata ilman ulkopuolista tilan tallennusmekaniikkaa. Etuna on muistin vapauttaminen aina tilasta toiseen siirryttäessä ja suorituksessa olevien tilojen lukumäärän pysyminen minimaalisena.

pushScene-metodilla CCScene-olioita lisätään CCDirectorin stack-jonoon, jossa viimeisin lisätty CCScene piirretään ruudulle. Metodi ei siis poista vanhaa CCSceneä, vaan se jää muistiin odottamaan tähän tilaan paluuta. *pushScene*-metodin vastametodi *popScene* poistaa yhden CCScenen käänteisessä järjestyksessä stack-jonoon lisäämiseen nähden, eli viimeisin CCScene poistetaan ja siirrytään sitä edeltävään CCSceneen. Koodiesimerkki 11 esittelee käytännössä, miten CCScenen vaihto suoritetaan ohjelmakoodissa.

```
// Ensimmäinen CCScene asetetaan runWithScene-metodilla.
CCScene* scene1 = [OmaScene1 node];
[[CCDirector sharedDirector] runWithScene: scene1];

// scene1 poistetaan ja tilalle asetetaan scene2.
CCScene* scene2 = [OmaScene2 node];
[[CCDirector sharedDirector] replaceScene: scene2];

// scene3 ja scene4 lisätään stack-jonoon, josta scene4 on viimeisenä lisätty
// ja näin ollen näkyväksi tilaksi.
CCScene* scene3 = [OmaScene3 node];
CCScene* scene4 = [OmaScene4 node];
[[CCDirector sharedDirector] pushScene: scene3];
[[CCDirector sharedDirector] pushScene: scene4];

// scene4 poistetaan stack-jonosta, jolloin jonossa seuraavana oleva scene3
// vaihtuu nykyiseksi tilaksi.
[[CCDirector sharedDirector] popScene];
```

Koodiesimerkki 11. CCDirectorin tilavaihtojen käyttö

5.1.3 CCScene

Cocos2D-sovelluksen tilakokonaisuudet rakennetaan yhden CCScene-olion alaisuuteen, ja näitä tiloja vaihdetaan CCDirectorin avulla. Sovelluksissa on useita tiloja, esimerkiksi peleissä on omat tilansa päävalikolle ja itse pelille, joita varten tehdään oma CCScene-olio. Cocos2D-sovelluskehys piirtää ruudulle kaikki CCScenen alaisuudessa olevat child-hierarkiarakenteeseen lisätyt CCNode-oliot, joilla on graafisia ominaisuuksia.

5.1.4 CCLayer

CCLayerin pääasiallinen tehtävä on lisätä sovelluksen tilaosien hallittavuutta. CCLayerillä on CCScenelle lisättävä taso, jonka tarkoitus on ryhmitellä samaan kategoriaan liittyvät asiat. Child-oliot käyttävät Cocos2D-rakenteen mukaisesti aina parent-olion ominaisuuksia, kuten koordinaatistoa. CCLayerillä luodut tasot helpottavat monesta oliosta koostuvan kokonaisuuden liikuttelua ja hallintaa.

Samalle CCLayerille lisättyjen child-olioiden piirtojärjestystä voi muuttaa keskenään, mutta kahden tai useamman CCLayerin child-olioiden järjesteleminen keskenään ei ole mahdollista. Samaan kokonaisuuteen sisältyvät elementit on pidettävä samalla

CCLayerillä, jotta ei tulisi tilanteita, jossa piirtojärjestyksen vaihtaminen on mahdoton toteuttaa halutulla tavalla liian usean CCLayerin käytön vuoksi.

5.1.5 CCSprite ja CCSpriteSheet

CCSpriteä käytetään, kun ruudulle halutaan piirtää kuva tai useasta kuvasta koostuva animaatio. Piirto perustuu pikseligrafiikkaan, joka sallii osittain tai kokonaan läpinäkyvien kuvien käytön. Oletusarvoisesti Cocos2D tukee png-, bmp-, tiff-, jpg-, ja gif-kuvatiedostoja, joita käytetään CCSpriten luomiseen.

CCSprite-olion luomiseksi on useampi vaihtoehto, ja tilanteen mukaan on valittava niistä sopivin. Helpoin tapa on luoda CCSprite käyttämällä *spriteWithFile*-alustusmetodia, jolle annetaan parametrina kuvatiedoston nimi. Koodiesimerkkissä 12 CCSprite luodaan suoraan kuvatiedoston nimellä. Luonnin jälkeen CCSprite lisätään child-hierarkiaan esimerkiksi CCLayerille, jonka jälkeen sovelluskehys piirtää sen ruudulle. Tämä tapa sopii käytettäväksi pienissä sovelluksissa ja sovelluksen tiloissa, joissa suorituskyvyn merkitys ei ole suuri. Jokainen CCSprite tuottaa oman piirtokutsun sovelluskehyksessä, joten suuri joukko CCSpritejä alkaa hidastaa sovellusta erillisten piirtokutsujen vuoksi (Cocos2D 2010b). Koodiesimerkissä 12 on yksinkertainen tapa luoda CCSprite-olio, käyttämällä lähteenä kuvatiedostoa.

```
// Yksinkertainen tapa luoda CCSprite-olio tiedoston nimeä käyttäen.
// CCSprite lisätään luonnin jälkeen esimerkiksi CCScene-olion child:ksi,
// jolloin se piirretään ruudulle.
CCSprite* spriteOlio = [CCSprite spriteWithFile:@”testImage.png”];
[layerOlio addChild:spriteOlio z:10];
```

Koodiesimerkki 12. CCSprite-olion luominen kuvatiedostosta

CCSpriteSheet on tarkoitettu optimoimaan CCSpriten käyttöä. CCSpriteSheet luodaan yhdestä kuvatiedostosta esimerkiksi spriteSheetWithFile-metodilla, joka saa parametrina kuvatiedoston nimen. Kuvatiedosto voi sisältää montaa pientä kuvaa, eli siitä voidaan pilkkoa suorakulmion muotoisia palasia, joita käytetään CCSpriten luomiseen. Tällöin käytettävä tekstuuri luodaan muistiin vain kerran ja CCSpriten käytöstä tulee taloudellisempaa. CCSpriteSheetiä käytettäessä luodaan CCSprite-olio *spriteWithSpriteSheet*-metodilla, jolle välitetään parametrina käytettävä CCSpriteSheet-olio ja

CGRect-tyyppinen suorakulmioalue kuvasta, josta CCSprite halutaan luoda. CGRect sisältää suorakulmaisen alueen x- ja y-koordinaatit sekä sivujen korkeuden ja leveyden. CCSpriteä ei tällöin lisätä esimerkiksi CCLayerille, vaan se asetetaan CCSpriteSheetin childiksi ja CCSpriteSheetin täytyy vielä olla esimerkiksi CCLayerin childina. Koodiesimerkissä 13 on malli CCSpriten käytöstä CCSpriteSheetin kanssa.

```
// CCSpriteSheet:llä optimoidaan usean CCSpriten käyttöä.
CCSpriteSheet* spriteSheetOlio = [CCSpriteSheet spriteSheetWithFile:
                              @”testImage.png”];

// Luodaan 50 CCSprite-oliota käyttäen CCSpriteSheet:n tekstuuria.
for (int i=0; i<50; i++) {
        // spriteSheetOlion tekstuurista käytetään samaa osaa, eli x:0 y:0
        // sijainnista, leveys:50 korkeus:50 suuruista aluetta.
        CCSprite* spriteOlio = [CCSprite spriteWithSpriteSheet:spriteSheetOlio
                                          rect:CGRectMake(0,0,50,50)];
        // CCSprite lisätään spriteSheetOlion omistukseen.
        [spriteSheetOlio addChild:spriteOlio z:10];
}

// Jotta CCSprite:t saisivat piirtokutsun, täytyy CCSprite:t omistava
// CCSpriteSheet lisätä CCScene-tilan child-hierarkiassa.
[layerOlio addChild: spriteSheetOlio z:10];
```

Koodiesimerkki 13. CCSpriten käyttäminen CCSpriteSheetin kanssa

CCSpriteSheetin optimointi perustuu siihen, että tekstuuri on valmiina muistissa, ja CCSpriteSheet tuottaa vain yhden suuren piirtokutsun sovelluskehykselle. Koska kaikki CCSprite-oliot lisätään yhden CCSpriteSheet-olion omistukseen, ei niitä voida jakaa esimerkiksi usealle eri CCLayerille, mikä voi olla rajoittava tekijä CCSpriteSheetin käyttöön.

5.1.6 CCMenu ja CCMenuItem

Valikkojen teko onnistuu helposti CCMenu- ja CCMenuItem-luokan olioilla. CCMenuItem-olioista luodaan varsinaiset painikkeet, jotka liitetään CCMenu-olioon. CCMenu hallitsee CCMenuItem-olioita ja esimerkiksi CCMenua liikutettaessa, liikkuvat myös kaikki CCMenuItem-oliot samassa suhteessa. Ainoastaa CCMenu lisätään child-elementiksi CCLayer- tai CCScene-oliolle, mikä CCSpriten tapaan lisää CCMenun piirrettäväksi ruudulle.

CCMenuItem-olion voi luoda helposti yksittäisestä kuvasta tai kuvista käyttämällä CCMenuItemImage-luokan alustusmetodeita. CCMenuItemillä on kolme tilaa: perustila, painike on painettu pohjaan ja painikkeen käyttö on estetty, joista estetty tila on valinnainen. Alustusmetodin valinnan mukaan kaikki tai tarvittavat CCMenuItemin tilat alustetaan kuvatiedostoilla. CCMenuItem voidaan alustuksessa linkittää haluttuun metodiin välittämällä kohdeolio ja selector-tyyppinen viite metodiin. Sovelluskehys kutsuu painiketta painaessa haluttua metodia, jossa sen toiminnot halutaan käsitellä. Koodiesimerkissä 14 on malli yhden painikkeen valikosta, jota voisi helposti laajentaa lisäämällä CCMenuItem-olioita esimerkin tapaan.

```
// Valikon painikkeen alustusmetodi, jolla lisätään painikkeen kaikkiin
// tiloihin oma kuva.
CCMenuItem* painike = [CCMenuItemImage
                         itemFromNormalImage:@”painike-perustila.png”
                         selectedImage:@”painike-painettu.png”
                         disabledImage:@”painike-estetty.png”
                         target:kohdeOlio
                         selector:@selector(painikettaPainettu:)];

// CCMenu:n alustus yksinkertaisesti listaamalla CCMenuItem-oliot. Listan
// oliot erotetaan toisistaan pilkulla ja lista päätetään nil-osoittimella.
CCMenu* valikko = [CCMenu menuWithItems: painike, nil];

// Valikko asetetaan child-hierarkiaan, jolla painikkeet piirretään ruudulle.
[layerOlio addChild:valikko z:10];
```

Koodiesimerkki 14. Valikon luominen CCMenu- ja CCMenuItem-olioilla

5.1.7 CCAction

CCAction on yliluokka toiminnoille, joita voi antaa CCNode-olioiden suoritettavaksi. Cocos2D:ssä on useita valmiita toimintoja, joilla yleensä vaikutetaan CCNode-olioiden attribuutteihin, kuten sijaintiin. Suurin osa perustoimintojen suorituksesta tapahtuu annettavan ajan puitteissa omassa säikeessään, joten toimintoja voi asettaa useita samaan aikaan. Lähes kaikilla toiminnoilla on reverse-metodi, joka tekee luodusta toiminnosta käänteisen toiminnon, eli jos nämä kaksi toimintoa ajetaan peräjälkeen, on silloin kohde CCNode-olio lähtötilassa.

CCAction-toiminnoissa on myös joukko aputoimintoja, joilla saadaan perustoimintojen suoritusta muutettua. Yksi tällainen toiminto on CCSequence, joka tekee muista toimin-

noista jonon ja käynnistää ne yksi kerrallaan. CCCallFunc-toiminto kutsuu haluttua metodia ja se on hyödyllinen vain CCSequencen kanssa. Sijoittamalla CCCallFunc-toiminnon CCSequence-jonon viimeiseksi, voidaan välittää tieto eteenpäin toimintojen päättymisestä. Koodiesimerkissä 15 luodaan useita erityyppisiä CCAction-olioita, jotka ensin järjestetään jonoksi CCSequence-oliolla, ja lopulta CCSprite-olio ajaa jonoon asetetut toiminnot.

```
// Luodaan CCSprite-olio, joka suorittaa actionit.
CCSprite* sprite = [CCSprite spriteWithFile:@”testImage.png”];
//
id actionMoveTo = [CCMoveTo actionWithDuration: 1.0f position: ccp(10,10)];
id actionReverse = [actionMoveTo reverse];
id callFunc = [CCCallFunc actionWithTarget:self selector:@selector(method:)];
id sequence = [CCSequence actions:actionMoveTo, actionReverse, callFunc, nil];
[sprite runAction: sequence];
```

Koodiesimerkki 15. Actionien luominen ja käyttö CCNode-oliolla

5.2 Kosketusten vastaanottaminen ja niiden käsittely

Kosketusnäytön ollessa ainut käyttäjän syötteitä vastaanottava käyttöliittymä, on sen merkitys iPhone OS -sovelluksissa suuri. Cocos2D-sovelluskehyksessä on oma mekaniikka, jolla olio saadaan vastaanottamaan kosketusnäytön tapahtumia. Haluttu olio lisätään CCTouchDispatcherille, joka hallinnoi kosketustapahtumia ja välittää niitä eteenpäin. CCTouchDispatcheriä käyttävän olion täytyy toteuttaa *CCTargetedTouchDelegate*-protokolla. Käytössä on neljä erilaista metodia, joista jokaista kutsutaan erilaisen kosketustapahtuman yhteydessä. Näistä metodeista on toteutettava vähintään kosketuksen alkamisen vastaanottava *ccTouchBegan*-metodi, joka palauttaa *BOOL*-arvon CCTouchDispatcherille merkiksi siitä, että kosketukseen halutaan reagoida, ja jatkossa saman kosketustapahtuman kutsut lähetetään samalle oliolle. (Cocos2D. 2010b) Koodiesimerkissä 16 on kaikki neljä *callback*-metodia, joilla näytön kosketuksia voidaan vastaanottaa.

```
// Alustetaan CCTouchDispatcher ja lisätään sille olio, joka ottaa näytön
// kosketuksia vastaan.
[[CCTouchDispatcher sharedDispatcher] addTargetedDelegate: olio priority:0
swallowsTouches:YES];

// Kosketus alkaa.
- (BOOL)ccTouchBegan:(UITouch *)touch withEvent:(UIEvent *)event
{
        // Kerrotaan CCTouchDispatcherille, että tämä olio ottaa vastaan tämän
        // kosketustapahtuman callback-kutsut.
        return YES;
}

// Kosketus piste muuttuu.
- (void)ccTouchMoved:(UITouch *)touch withEvent:(UIEvent *)event
{
        //
        CGPoint location = [touch locationInView: [touch view]];
        CGPoint convertedLocation = [[CCDirector sharedDirector]
                                       convertToGL:location];

        // spriteOliota siirretään kosketuspisteeseen.
        [spriteOlio setPosition: kosketusPiste];
}

// Kosketus päättyy.
- (void)ccTouchEnded:(UITouch *)touch withEvent:(UIEvent *)event
{ }

// Kosketustapahtuma peruutetaan.
- (void)ccTouchCancelled:(UITouch *)touch withEvent:(UIEvent *)event
{ }
```

Koodiesimerkki 16. Kosketustapahtumia käsittelevät metodit

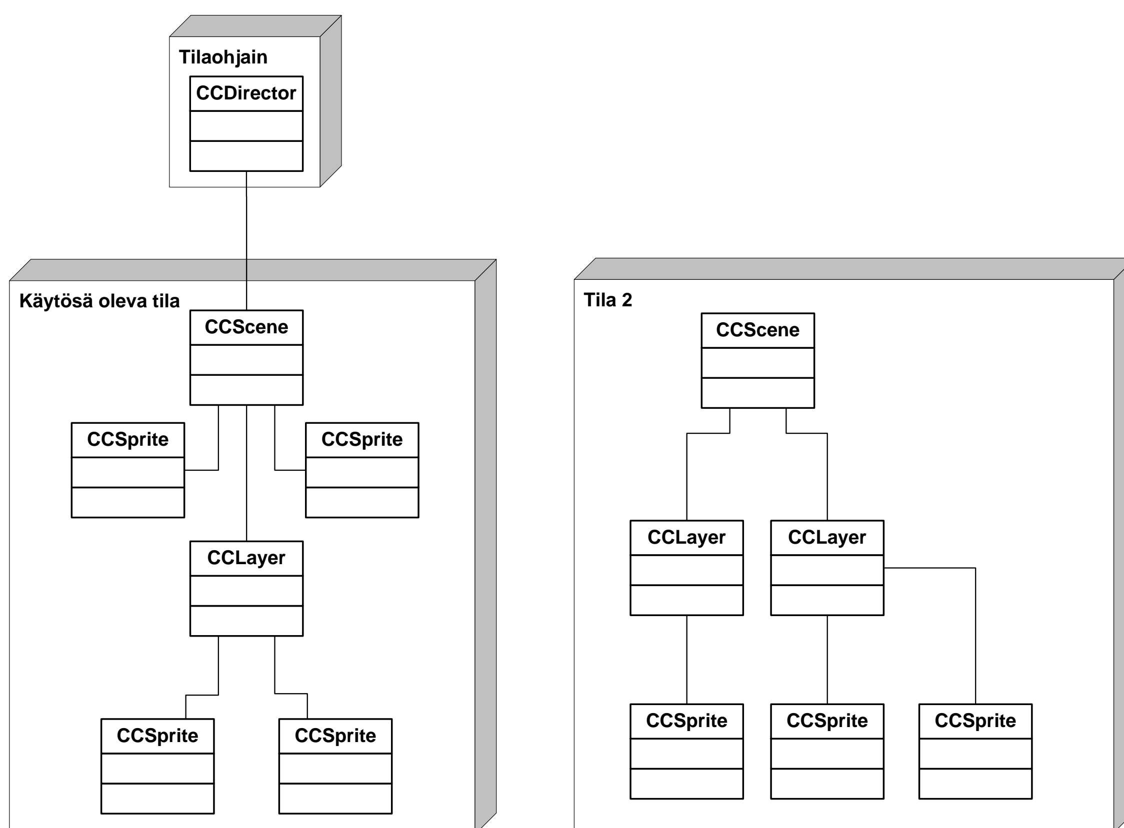
5.3 Cocos2D-sovelluksen rakenne

Koko sovelluksen ohjaamisesta on vastuussa CCDirector. CCDirector sijoittuu Cocos2D-sovelluksen arkkitehtuurin huipulle, missä se päättää pienempien itsenäisten kokonaisuuksien eli CCScene-olioiden esitysvuoroista. CCScene-olioita vuorotellaan tai lisätään CCDirectorin esitysjonon mukaisessa järjestyksessä.

CCScene on paketti, jossa kaikki tilan graafinen esitys toteutetaan. Normaalisti CCScenelle lisätään yksi tai useampi CCLayer child-elementeiksi, joilla tilakokonaisuutta jaetaan hallittavuuden kannalta omiin kokonaisuuksiin. CCSceneä voidaan käyttää samaan tapaan kuin CCLayeriä, joten sovellus on mahdollista toteuttaa kokonaan CCScenen

päälle ilman CCLayer-tasoja, mutta hallittavuuden kannalta CCScene:llä tulisi olla ainakin yksi CCLayer graafista esitystä varten.

Sovelluksen graafinen esitys luodaan pitkälti CCSprite-olioilla. Samaan kategoriaan kuuluvat CCSprite-oliot ryhmitellään CCScene- ja CCLayer-tasoille, joiden child-hierarkian hyödyntäminen mahdollisesti helpottaa tasolle lisättyjen olioiden käsittelyä. Kuvassa 6 on yksinkertainen malli sovelluksen tiloista.



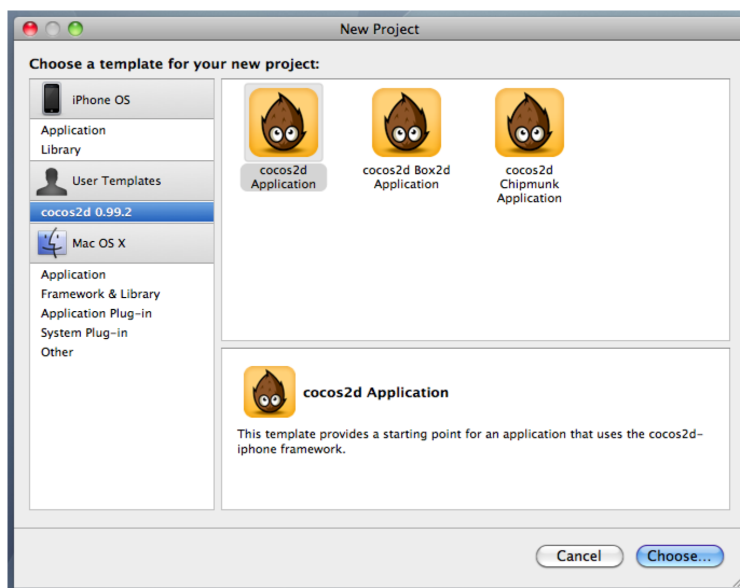
Kuva 6. Cocos2D-sovellus rakentuu tiloista, joiden ajoa CCDirector ohjaa.

5.4 Cocos2D-sovelluksen luominen

Cocos2D-sovelluskehys on täysin ilmainen, ja uusimmat lähdekooditiedostot voi ladata sen kotisivulta http://www.cocos2d-iphone.org tai hakemalla ne suoraan projektin versionhallinnasta. Projektin mukana tulee paljon tiedostoja itse sovelluskehyksen lisäksi, mistä suuri osa on tekstimuotoista dokumentaatiota tai lisenssitietoja. Joukossa on myös Xcode-projekti, joka sisältää Cocos2D:tä varten kirjoitettuja testejä. Testejä voi vapaasti

tutkia lähdekoodista ja kääntää ne ajettavaksi sovellukseksi. Testitapauksista saa näin ollen hyvää esimerkkiä Cocos2D:n käytöstä ja mihin kaikkeen se kykenee.

Xcodeen voidaan lisätä Cocos2D-mallipohja eli template, joka lisää mahdollisuuden luoda Cocos2D:n projekteja tai luokkia, joissa on valmiina Cocos2D-elementit. Cocos2D-projektitiedostojen joukossa on install_template.sh, joka on komentoriviltä ajettava skripti, jolla saa mallit ajettua helposti Xcodeen. Kuvassa 7 on uuden projektin aloitusvaihe, josta voi valita suoraan Cocos2D-projektin, kun templaten on asennettu.



Kuva 7. Cocos2D templaten asennuksen jälkeen on helppo luoda Cocos2D-projekti.

iPhone OS näyttää asennetut sovellukset listattuna, jossa on sovellusikoni ja nimi. Oletuksena sovelluksen nimi on projektille annettu nimi. Sovellusikoni on projektissa nimella *icon.png*, jonka voi yksinkertaisesti korvata toisella 57x57-kokoisella kuvalla. Sovelluksen käynnistyksen aikana ruudulle piirretään *default.png*-kuvatiedosto, jonka voi myös vaihtaa helposti projektissa korvaamalla (Zirkle, Hogue 2010, 3).

Jokainen iPhone OS -sovellus kutsuu käynnistyessään *applicationDidFinishLaunching*-metodia, joka on (projektin nimi) + AppDelegate-luokassa. Tämän metodin sisältöä muokkaamalla otetaan sovelluksen kontrolli haltuun, ja ensimmäinen CCScene-olio lisätään CCDirectorille ajoon. Koodiesimerkissä 17 on Cocos2D-templatea käyttämällä luodun projektin *AppDelegate*-luokka, johon sovelluksen ensimmäinen kutsu lähetetään. *applicationDidFinishLaunching*-metodiin lisätään tarvittaessa omien sovellus-

komponenttien alustukset, mutta vähintään metodissa lisätään ensimmäinen suoritettava tila CCDirectorille.

```
// Sovellus käynnistymisen jälkeen tätä metodia kutsutaan.
// Cocos2D-template valmiiksi kaikki liitokset iPhone OS-komponentteihin,
// josta on helppo lähteä rakentamaan omaa sovellusta.
- (void) applicationDidFinishLaunching:(UIApplication*)application
{
        // Init the window
        window = [[UIWindow alloc] initWithFrame:
                                [[UIScreen mainScreen] bounds]];
        // cocos2d will inherit these values
        [window setUserInteractionEnabled:YES];
        [window setMultipleTouchEnabled:YES];

        // Try to use CADisplayLink director
        // if it fails (SDK < 3.1) use the default director
        if( ! [CCDirector setDirectorType:CCDirectorTypeDisplayLink] )
                [CCDirector setDirectorType:CCDirectorTypeDefault];

        // Use RGBA_8888 buffers
        // Default is: RGB_565 buffers
        [[CCDirector sharedDirector] setPixelFormat:kPixelFormatRGBA8888];

        // Create a depth buffer of 16 bits
        // Enable it if you are going to use 3D transitions or 3d objects
//      [[CCDirector sharedDirector] setDepthBufferFormat:kDepthBuffer16];

        // Default texture format for PNG/BMP/TIFF/JPEG/GIF images
        // It can be RGBA8888, RGBA4444, RGB5_A1, RGB565
        // You can change anytime.
        [CCTexture2D setDefaultAlphaPixelFormat:
                        kTexture2DPixelFormat_RGBA8888];

        // before creating any layer, set the landscape mode
        [[CCDirector sharedDirector] setDeviceOrientation:
                                CCDeviceOrientationLandscapeLeft];
        [[CCDirector sharedDirector] setAnimationInterval:1.0/60];
        [[CCDirector sharedDirector] setDisplayFPS:YES];

        // create an openGL view inside a window
        [[CCDirector sharedDirector] attachInView:window];
        [window makeKeyAndVisible];

        // Seuraavaksi lisätään ensimmäinen ajoon lisättävä tila.
        [[CCDirector sharedDirector] runWithScene: [HelloWorld scene]];
}
```

Koodiesimerkki 17. Sovelluksen käynnistävä *applicationDidFinishLaunching*-metodi, johon Cocos2D:n template tekee kaikki sovelluskehyksen tarvitsemat lisäykset.

6 Legends of Elendria: The Frozen Maiden

Tämän opinnäytetyön toimeksianto on tehty ensimmäiseen Kyy Games Oy:n kehittämään peliin *Legends of Elendria: The Frozen Maiden*. Pelissä on tarkoitus puolustaa omaa linnoitusta sijoittamalla pelialueelle torneja, jotka ampuvat tietä pitkin hyökkäävää vihollisjoukkoa. Mekaniikka on pitkälti niin kutsutun *Tower defence* -pelityypin mukainen, mutta peliin on lisätty paljon roolipelielementtejä, kuten tarinan ja erikoiskykyjen muodossa, joten genre ei ole niin yksiselitteinen. Liittyessäni projektiin, kehitystyö oli siinä vaiheessa, että tarinaa ei ollut pelissä sisällytettynä millään tasolla, ja tästä syntyi myös tämän opinnäytetyön aihe.

Toimeksiannon voi jakaa kaksiosaiseksi kokonaisuudeksi. Tärkeintä oli luoda mekaniikka, jolla saataisiin tarinaa kuljetettua keskitetyn hallintakomponentin avulla, joka myöhemmin nimettiin QuestEngineksi. QuestEngine on tiiviissä yhteistyössä pelin karttatilan kanssa, jossa tarinan kerronta pääosin tapahtuu. Karttatila käsittää yksittäisen Cocos2D-sovelluskehystä käyttävän tilakokonaisuuden, jossa luodaan ja hallitaan kaikki käyttöliittymään ja graafiseen esitykseen liittyvät asiat. Toimeksiannon tehtävä oli näin ollen hyvin monipuolinen ja mielenkiintoinen, jossa täytyi kehittää oma tietorakennemalli ja käyttää Cocos2D-sovelluskehystä.

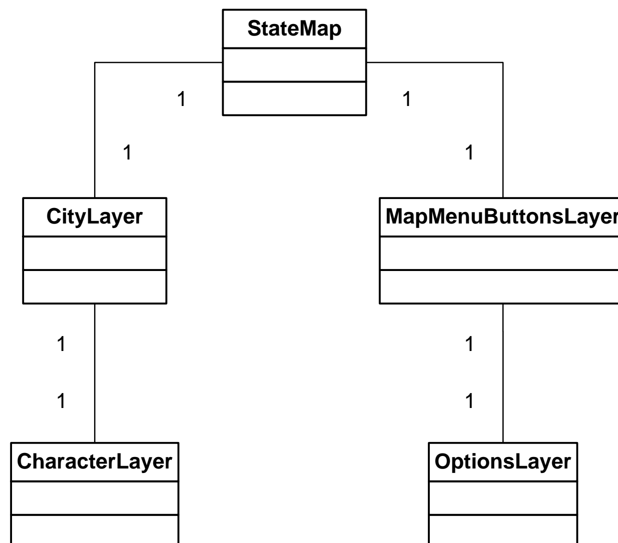
6.1 Karttatila Cocos2D:n voimin

Karttatilan eli StateMapin ohjelmointi liittyi vahvasti toimeksiantoon, vaikka suoranaisesti sitä ei määritelty lähtökohtaisesti toteutettavaksi. Pelin tilat ovat Cocos2D-sovelluksen mukaisesti jaettu CCScene-olioihin, jollainen karttatila myös on. Karttatilan ja QuestEnginen välinen yhteistyö on niin tiivistä, että molempien osa-alueiden kehittäminen samaan aikaan oli kehitysprosessin kannalta paras ratkaisu.

Pelin maailma on kuvattu 2D-grafiikalla viistosti ylhäältä päin. Karttatilassa pelaaja voi valita seuraavan suoritettavan tehtävän, joita avautuu eri sijainteihin pelin edetessä. Pelin alkaessa kaupunkeja on vain yksi kappale, jonka jälkeen tehtäviä suoritettaessa uusia kaupunkeja ja sivutehtäviä avautuu kartalle, ja lopulta pelaajalla on tutkittavanaan laaja kaupunkiverkosto koko valtakunnasta. Hahmo liikkuu kartalla paikasta toiseen yksin-

kertaisen reittihaun mukaisia teitä pitkin, kun pelaajan painallus kohdistuu johonkin kaupunkiin. Toinen tärkeä tehtävä eli tarinan kuljettaminen eteenpäin tapahtuu myös karttatilassa, sen päälle aukeavissa ikkunoissa. Keskustelut ovat kytköksissä aina tiettyyn tehtävään, joka aktivoituu pelihahmon saapuessa aktiiviseen kaupunkiin.

Karttatila rakentuu Cocos2D:n tilamallin mukaisesti CCScene-olion ympärille ja tilaan siirrytään, kun tämä CCScene-olio asetetaan CCDirectorille. Tilasiirtymät käsitellään kaikki erillisen tilakoneen kautta keskitetysti, mutta sen käsitteleminen ei kuulu tämän opinnäytetyön piiriin. StateMap-oliolla on omistuksessaan eli childiksi lisätyt CCLayer-olioit CityLayer ja MapMenuButtonsLayer, jotka sisältävät oman hallintakokonaisuutensa. Rakennetta jaetaan eri haaroihin niiden käytön mukaan. Kartalla ja kartan päälle sijoitetuilla valikkopainikkeilla ei ole juurikaan yhteisiä toimintoja, joten ne on syytä erottaa kumpikin omaksi CCLayer-tasoksi. Tällöin näyttöä suuremman kartan liikuttelu ei vaikuta valikkojen sijaintiin. Kuvassa 8 on karttatilan yksinkertaistettu rakennemalli, miten karttatilan kokonaisuudet on omiin hallittaviin osiin.



Kuva 8. Karttatilan jako pienempiin osiin CCLayer-olioilla

CityLayer on kokonaisuus, jossa lisätään CCSprite-tyyppiä olevat taustakuva, kaupungit, tiet, erikoistehosteet ja lisäksi oma CCLayer-tyyppinen CharacterLayer piirtotaso kartalla liikkuvalle hahmolle. CityLayer on ylin hallitseva taso heti CCScene-tyyppisen StateMap-olion jälkeen, mikä tekee kartan liikuttelusta helppoa. Karttaan kohdistuvat käyttäjän kosketukset otetaan vastaan ja käsitellään CityLayerillä. Vaihtoehtoisia tapah-

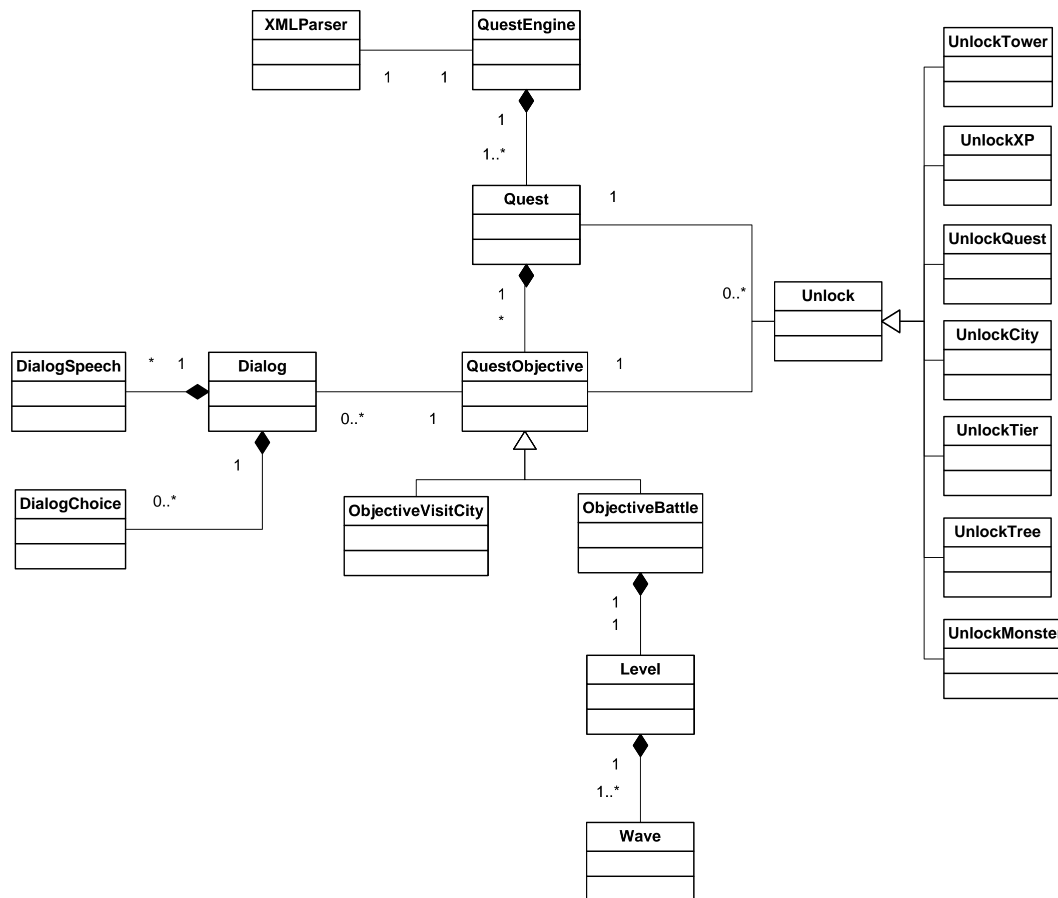
tumia ovat jonkin kaupungin painaminen tai kartan liikuttaminen pyyhkäisyliikkeellä. CityLayeriä liikuttamalla, liikkuvat mukana myös kaikki child-hierarkiassa alempana olevat oliot. Cocos2D:n apu on tässä valtava, sillä kaikkia kartalle sijoiteltujen olioiden sijaintia ei tarvitse päivittää omassa koodissa, vaan Cocos2D hoitaa tässä tapauksessa suurimman työn.

Valikot luodaan ja hallitaan MapMenuButtonsLayerillä tai sen omistukseen lisätyillä CCLayer-tasoilla. Valikot ovat aina ylimpänä karttatilan piirtojärjestyksessä. Koska karttatilassa on kaksi CCLayer-tasohaaraa itse kartalle ja valikoille, on tämän piirtojärjestyksen määrittäminen helppoa vaihtamalla MapMenuButtonsLayerin z-arvo suuremmaksi kuin CityLayerillä on. Tällöin MapMenuButtonsLayer ja sen child-oliot ovat aina CityLayeriä ylempänä.

6.2 QuestEngine

Pelin kulkua ohjaa ns. QuestEngine, jonka avulla ylläpidetään erinäisiä tietoja pelaajalle annettavista tehtävistä eli Questistä. Pelaajan on suoritettava päätehtäviä edetäkseen pelissä ja niiden lisäksi on myös mahdollista suorittaa sivutehtäviä, jotka ovat yksittäisiä pelimaailmaa täydentäviä palasia ja antavat pelaajalle ylimääräisiä palkintoja. Jokainen Quest on yksi kokonaisuus, jota voidaan pilkkoa pienentyviin kokonaisuuksiin, mitä alemmas kuljetaan tietorakennepuussa. QuestEngine on tämän rakenteen huipulla ja vastaa oikean tiedon hakemisesta ja muuttamisesta.

QuestEnginen omistaa MetaGame-olio, jonka julkisen näkyvyyden ansiosta siihen päästään käsiksi useista eri tarvittavista sovellusalueista. MetaGame-olio alustetaan sovelluksen käynnistyessä, milloin myös kaikki sen omistamat oliot QuestEngine mukaan lukien alustetaan käyttövalmiiksi. MetaGamen tarkoituksena on olla kanavana yleisesti tarvittaviin sovelluksen osiin ja se on hyvin korkealla olioiden omistusrakenteessa. Kuvassa 9 on QuestEnginen puurakenne kokonaisuudessaan.



Kuva 9. QuestEnginen puurakenne, josta pelin kulkuun liittyvä tieto poimitaan

QuestEngine alustaa kaikki Questit pelin käynnistyessä XML-tiedostosta, jonka jälkeen ne säilytetään saatavilla muistissa lähes koko sovelluksen suorituksen aikana. Pelaamisen aikana kaikki questeihin tehtävät muutokset käsitellään QuestEnginen kautta. Pelitilanne tallennetaan tiedostoon sovelluksen sulkeutuessa NSCoding-protokollaa hyödyntäen ja samaan tilaan palataan sovelluksen käynnistyessä seuraavan kerran. XML-tiedostoa käytetään vain uuden pelin alustusta varten ja vanha pelitilanne ladataan aina omasta tiedostosta.

XML:n etuna on sen käsiteltävyys irrallaan projektista. Kaikki pelin tehtäviin ja tarinaan liittyvä on kirjattu XML-muotoon, jota voi muokata myös ohjelmointia osaamaton henkilö. XML-tiedoston koko kasvaa helposti suureksi ja sen lukeminen suorituksen aikana lisää paljon suorittimen kuormitusta hetkellisesti ja näkyvästi. Tiedostosta lukua ei kannata käyttää jokaisessa kelvollisessa paikassa sen suuren kuormituksen vuoksi, vaan sille on varattava sopiva ajankohta, jossa sovelluksen hidastumisesta ei ole haittaa.

QuestEnginen oliot alustetaan XML-tiedostosta heti pelin käynnistyessä muiden resurssien lataamisen yhteydessä, jolloin sovellus on erillisessä lataustilassa ja suorituksessa näkyvät kuormituspiikit eivät haittaa pelaajan pelikokemusta.

XML-tiedosto luetaan hyödyntämällä valmista NSXMLParser-komponenttia Foundation.frameworkistä. Tämän valmiin komponentin lisäksi tarvitaan oma toteutusosa, missä otetaan vastaan NSXMLParser-olion lähettämät callback-tyyppiset metodikutsut. Lopullinen toiminta suoritetaan itse kirjoitetulla XMLParser-luokan oliolla, joka luo ja alustaa muita olioita XML:stä luettavan sisällön mukaisesti. XMLParser-luokka periytetään tavallisesti NSObject-luokasta ja sille kirjoitetaan tarvittavat callback-metodit, joita NSXMLParser kutsuu XML-tiedoston lukuvaiheessa. Kaikkia callback-metodeja ei suinkaan tarvitse toteuttaa, vain tarvittavat riittää hyvin. Pelissä käytettiin kahta callback-metodia, joita kutsutaan, kun aloitus- ja lopetus-tag löydetään. Arvot ovat syötettynä tagin attribuuteiksi, minkä todettiin paremmaksi tavaksi kuin arvojen sijoittamisena aloitus ja lopetus tagin sisälle. Koodiesimerkissä 18 on metodit, joita XML:n parsija kutsuu, kun se löytää aloitus- ja lopetus-tagin.

```
// XML-parsija kutsuu tätä metodia, kun aloitus-tag löydetään.
-(void)parser:(NSXMLParser*)parser
        didStartElement:(NSString*)elementName
        namespaceURI:(NSString*)namespaceURI
        qualifiedName:(NSString*)qName
        attributes:(NSDictionary*)attributeDict
{

}

// Vastaavasti tätä metodia kutsutaan, kun jokin lopetus-tag löydetään.
-(void)parser:(NSXMLParser*)parser
        didEndElement:(NSString*)elementName
        namespaceURI:(NSString*)namespaceURI
        qualifiedName:(NSString*)qName
{

}
```

Koodiesimerkki 18. XML:n tieto käsitellään parsijan *parser*-metodeilla

XML-tiedosto sisältää siis lähes kaiken Questeihin liittyvät alustusarvot ja tekstit. Jokainen Quest on oma kokonaisuutensa. XML:ssä vaaditaan olevan ainakin yksi quest, jolle on määritelty vähintään yksi tehtävä eli objective ja taistelutehtävän taistelun alustustiedot, muut ovat vaihtoehtoisesti lisättävissä ja sovellus tutkii dynaamisesti, mitkä

tiedot ovat olemassa ja toimii sen mukaisesti. Jokaisella Questillä voi olla uusia pelissä avattavia palkintoja, x-määrä suoritettavia tehtäviä ja dialogeja tehtävän alkaessa ja päättyessä.

Lopulta XML-tiedoston sisältämä tieto näyttelee hyvin suurta osaa QuestEnginen toiminnassa. Pelin tehtävät ja juoni on täysin mahdollista muuttaa vaihtamalla XML-tiedostoa. XML:stä suoritettavan alustuksen jälkeen kontrolli on täysin QuestEnginellä. Käytännössä karttatilassa tapahtuvat toiminnot johtavat siihen, että QuestEngineltä kysytään tietoja tai sitä pyydetään suorittamaan muutoksia Quest-olioihin. Tyypillisesti QuestEngine tarkistaa, onko Quest suoritettu onnistuneesti, mikä karttatilassa näytetään pelaajalle.

6.3 Ilmenneet ongelmat

Kuten olettaa voi, ei ongelmilta voitu välttyä tässäkään projektissa. Cocos2D:n käytön kokemattomuuden ja vaihtuvien vaatimusten vuoksi jo kirjoitettua koodia täytyi muuttaa pariin otteeseen tavoitteeseen pääsemiseksi. Totuus on yhä, että perusteellisella ja ennakoivalla suunnittelulla päästään toteutusvaiheessa helpommalla. Säännöllisen testauksenkaan hyötyä ei ole syytä aliarvioida yhtään.

Uusien muutosten myötä tuote käy läpi perusteellisen testauksen, jonka suorittaa testaukseen valikoitu henkilö. On tärkeää, ettei kukaan tee lopullista testausta omista muutoksistaan, vaan siihen valikoidaan joku kehitystiimin ulkopuolelta. Omalle työlleen on tapana sokeutua, jonka vuoksi ulkopuolisen henkilön mielipide on tärkeä. Testaus on myös suoritettava sillä laitteella, jolla lopullinen käyttäjä sitä käyttää, eli tässä tapauksessa iPhonella tai iPod Touchilla, eikä saa luottaa emulaattorin antavaan kuvaan toimivuudesta. Laitteella testatessa nähdään se, millainen on sovelluksen todellinen suoritusnopeus ja tilanteet, joissa sovellus kaatuu.

Ensimmäisenä haasteena oli luoda XML-tiedosto, joka kattaisin kaiken tarvittavan tiedon ja sellaisessa muodossa, jotta Quest-oliot saataisiin alustettua oikein. Ensimmäinen kehitysversio otettiin käyttöön melko varhaisessa vaiheessa, jolloin kaikkea tietoa ei vielä ollut XML-tiedostoon sisällytetty. Ongelmat huomattiin, kun pelin uusien ominaisuuksien merkitseminen XML-tiedostoon olisi tuonut kohtuuttoman suuren ohjelmointi-

työn tiedoston lukuvaiheeseen. XML täytyy suunnitella niin, että olioita luodessa ne voidaan helposti liittää oikealle omistajalleen. Toimimattoman rakenteen huomaa siitä, että XML:n eri osissa joudutaan käyttämään tunnisteita, kuten yksilöllistä id-lukua kokonaisuuksien yhdistämiseen.

Tyypillinen peliohjelmoinnin ongelma liittyy suorituskykyyn. Mitä pidemmälle projekti eteni, sitä selkeämmin suorituskyvyssä alkoi näkyä heikentymistä pelin karttatilassa. iPhone ja iPod Touch suoriutuvat hyvin taustalla tapahtuvista matemaattisista toiminnoista, mutta ajattelemattomasti toteutettu graafisten olioiden käyttö heikentää suorituskykyä helposti. Ongelman ydin kehittyi liian monen erillisen CCSprite-olion käytöstä. Tilanteen ratkaisi CCSprite-olioiden kutsujen optimointi CCSpriteSheetin avulla. Tällöin Cocos2D tekee yhden suuren kutsun piirtokomponenteille monen yksittäisen kutsun sijaan. Karttatilan ruudunpäivitysongelmat keskittyivät pelin loppuosaan, jossa kaupunkeja ja teitä voi olla samaan aikaan näkyvillä useita. Erityisesti tiet, jotka toteutetaan pienistä ketjutetuista kuvista, tuottavat suuren määrän piirtokutsuja, jos niitä ei optimoida CCSpriteSheetin avulla.

Ongelmallisin tapaus lienee pelitilanteen tallennus. Pelaajan kannalta ihanteellisin tilanne on pelitilanteen pysyminen täysin samassa tilassa riippumatta yllättävistä häiriötekijöistä. Tässä täytyy muistaa, että moniajon puuttuessa peli tapetaan esimerkiksi puheluun vastattaessa. Mobiilipeleissä käyttäjältä ei pitäisi kysyä pelitilanteen tallentamista, vaan yleisesti sen oletetaan tapahtuvan ilman erillistä toimenpidettä. QuestEnginen puolella tilan tallennus on vielä helppoa, mutta karttatilassa omissa säikeissä suoritettavat tapahtumat eli Actiont tekivät täysin saman tilaan palaamisen hankalaksi. Nopeat tapahtumaketjut jouduttiin näin ollen aloittamaan alusta, mikäli peli keskeytyi odottamattomasti.

7 Loppusanat ja yhteenveto

Lähtökohtana toimeksianto ja aihe olivat alusta lähtien hyvin mielenkiintoiset. Alustana iPhone eroaa paljon muista alustoista, joille olen sovelluksia tehnyt, mikä vain lisäsi uutuuden viehätyksellään mielenkiintoa. Kokemus Objective-C-ohjelmoinnista ja iPhone-laitealustasta olivat tämän työn aloitusvaiheessa hyvin minimaaliset. Vain opinnäytetyötä edeltävä harjoittelujakso oli työskentelyn pohjana. Opeteltavaa oli paljon, mutta kuitenkin lähtökohta tuntui sopivan haastavalta.

Säännöllinen työskentely *Legends of Elendria: The Frozen Maiden* -projektin parissa on kehittänyt ohjelmointitaitojani valtavasti, mikä on selkeästi säännöllisen käytännön harjoittelun ansiota. QuestEnginen ja karttatilan valmistumisen jälkeen voi löytää nykyisellä tietotasolla huonoja ratkaisuja, mutta lopullinen kokonaisuus on täysin toimiva ja ajaa täysin sen tehtävän, joka sille asetettiin.

QuestEnginen toteutukseen olen tyytyväinen ja siihen, miten hyvin suunnittelemani tietorakenne lopulta toimi. Tietorakenne on looginen ja puumaisesta rakenteesta saa kaiken tarvittavan tiedon pienissä osissa. Rakenteesta muodostui lopulta melko laaja, joten joissakin tapauksissa tiedon onkiminen voi olla hankalaa, mutta ei suinkaan mahdotonta.

Karttatilan kehitystä vaivasi sen päämääräämättömyys. Pienistä lisäyksistä alkoi kehkeytyä etenemistä hidastavia esteitä, joita ratkaistiin helpoilla ratkaisuilla, mikä toi myöhemmin uusia etenemistä hidastavia esteitä. Samaa toimintoa jouduttiin paikkaamaan muutamaan kertaan uusiksi, mutta lopulta korjaukset olivat pieniä, ja eivät vaikuttaneet perustan suunnitteluun. Pääosin karttatilan grafiikan piirtoa jouduttiin loppupuolella optimoimaan.

Toimeksiannon myötä syntyneet sovellusosat QuestEngine ja karttatila olivat vielä keskeneräisiä valmiiseen peliin. Jatkoin työntekijänä Kyy Gamesin opinnäytetyön toimeksiannon jälkeen, joten pääsin jatkamaan aloittamaani työtä ja viemään sen lopulliseen pisteeseensä. Toimeksiantotyötä voi tuskin suoraan hyödyntää muissa projekteissa, sillä se on varsin tarkasti määritelty yhden projektin mukaiseksi, mutta siinä on myös onnistuneita asioita, joita voidaan soveltaa muihin projekteihin.

Uskon, että työni tarjoaa hyvän tietopohjan uudelle iPhone OS-sovelluskehittäjälle. Objective-C-ohjelmointikieli on työssä hyvin tärkeässä osassa, ja sitä tutkiessani, sain hyvin kehitettyä omaa tietotasoa ja osaamistani. Kokonaisuudessaan Objective-C on tietysti hyvin monimuotoinen ja sisältää paljon hienouksia, mutta perustietous ja ominaispiirteet ovat tässä työssä hyvin kiteytettynä. Aloittelevalle iPhone OS -ohjelmoijalle tämä opinnäytetyö tarjoaa lähtökohdan uuden asian opiskelussa. Syventävää tietoutta täytyy osata myöhemmin hankkia itse, mutta perustiedon pohjalta on huomattavasti helpompi lähteä sitä rakentamaan.

Cocos2D-sovelluskehys tarjoaa tällä hetkellä helposti lähestyttävän paketin pelikehittäjälle. Sovelluskehykseen tuodaan vielä uusia ominaisuuksia hienoja ominaisuuksia, mutta sitä voi jo tällä hetkellä käyttää laajankin pelin tekemiseen. Opinnäytetyössä paneudutaan myös Cocos2D:n tärkeimpiin osa-alueisiin, joiden lisäksi on paljon muita hyödyllisiä ominaisuuksia. Kun perusteet ovat hallussa, voi Cocos2D:n käyttöön perehtyä tarkemmin.

Lähteet

Apple inc. 2009a. The Objective-C 2.0 Programming Language.
[online] [viitattu 15.02.2009]
http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf

Apple inc. 2009b. Tools for iPhone OS Development.
[online] [viitattu 12.02.2010]
http://developer.apple.com/iphone/library/referencelibrary/GettingStarted/URL_Tools_for_iPhone_OS_Development/index.html

Apple inc. 2010. Apple Reports Second Quarter Results.
[online] [viitattu 25.04.2010]
http://www.apple.com/pr/library/2010/04/20results.html

Ben. 2010. Key Classes in Cocos2D.
[online] [viitattu 04.03.2010]
http://www.cocos2dandme.com/

Cocos2D. 2010a. About.
[online] [viitattu 14.04.2010]
http://www.cocos2d-iphone.org/about

Cocos2D. 2010b. Programming Guide.
[online] [viitattu 10.03.2010]
http://www.cocos2d-iphone.org/wiki/doku.php/prog_guide:index

Koskimies, Kai 2000. Oliokirja.
Helsinki: Satku.

Tenon Intersystems. 2010. Objective-C.
[online] [viitattu 09.05.2010]
http://www.tenon.com/products/codebuilder/Objective-C.shtml

Costello, Sam. 2010. iPhone OS History.
[online] [viitattu 15.04.2010]
http://ipod.about.com/od/iphonesoftwareterms/a/firmw_history.htm

Zirkle Paul, Hogue Joe. 2010. IPhone Game Development: Developing 2D & 3D Games in Objective-C.
Sebastopol, CA: O'Reilly Media, Inc.