Opinnäytetyö (AMK)

Tietojenkäsittelyn koulutusohjelma

Sähköisen liiketoiminnan järjestelmät

2013

Teppo Kavander

PELIOHJELMOINTI C#:LLA JA XNA:LLA

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittelyn koulutusohjelma | Sähköisen liiketoiminnan järjestelmät

Toukokuu 2013 | 55

Päivi Nygren

Teppo Kavander

PELIOHJELMOINTI C#:LLA JA XNA:LLA

Opinnäytetyön tavoitteena oli tutkia 2D-pelien ohjelmointia ja erilaisia toteutustapoja pelien yleisimpien piirteiden toteuttamiseen.

Opinnäyteyössä rakennettiin 2D-tasoloikkapeli ja sen myötä käsiteltiin peliohjelmoinnin perustekniikoita kuten tile engine –menetelmää, ruudunhallintaa ja erilaisia tapoja testata peliobjektien törmäyksiä.

Peli rakennettiin hyödyntäen Microsoftin XNA-pelikirjastoa ja Microsoft Visual Studio 2010 –kehitysympäristöä. Peli toteutettiin olio-ohjelmoinnin mukaisesti C#-ohjelmointikielellä.

ASIASANAT:

Peliohjelmointi, XNA, C#, 2D-peli

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Business Information Technology | e-Business Systems

May 2013 | 55

Päivi Nygren

Teppo Kavander

GAME PROGRAMMING WITH C# AND XNA

The goal of this thesis is to look into the 2D game programming and different ways to implement some of the most common features of games.

During the process a 2D platformer game was created. The theory also deals with game programming's basic techniques like tile engine method, screen handling and different ways to test collisions between game objects.

The game was built using Microsoft's XNA game library and Microsoft Visual Studio 2010 software. The game was implemented according to object-oriented programming style with C# programming language.

KEYWORDS:

Game programming, XNA, C#, 2D game

SISÄLTÖ

LYHENTEET JA SANASTO

AAA-peli	Yleensä suurten pelitalojen suurella budjetilla ja isolla kehittäjäjoukolla tuottama kaupallinen peli
Bittikarttagrafiikka	Toiselta nimeltään pikseligrafiikka. Yleinen tapa esittää kuvia digitaalisessa muodossa. Kuva koostuu pikseleistä.
C#	Ohjelmointikieli
DirectX	Ohjelmointirajapinta peleille Windows käyttöjärjestelmässä ohjelman ja laitteiston välille
FPS	Frames Per Second. Ruudunpäivitystiheys, eli montako kertaa ruutu päivittyy sekunnissa
Indie-peli	Itsenäisen, usein yksittäisen henkilön tai hyvin pienen ryhmän luoma peli
Isometrinen kuvakulma	Tapa esittää kolmiulotteinen objekti kaksiulotteisena kuvana, jolloin objektista pystytään näkemään kolme sivua kerrallaan
Sprite	Peleissä käytettävä bittikarttagrafiikkaan perustuva kuva
Spritesheet	Monesta spritestä koostuva isompi kuva
XNA	DirectX:n päälle luotu pelikirjasto

1 JOHDANTO

Pelit kehittyvät ja itse peliala kasvaa ja pienempien peliyritysten määrä yleistyy. Viime vuosina myös suomalaiset pelitalot ovat päässeet maailmankartalle. Pelaajana itseäni on alkanut kiinnostaa, miten pelejä oikein tehdään. Tästä syystä valitsin opinnäytetyön aiheeksi luoda esimerkkipelin C#-ohjelmointikielellä ja XNA-kirjaston avulla, jotka tarjoavat yhden tavan tutustua pelien kehittämiseen.

Opinnäytetyössä suunnittelen ja toteutan 2D-tasoloikkapelin, joka sisältää tärkeimmät perusasiat peliohjelmoinnista C#-ohjelmointikielellä XNA-kirjaston avulla toteutettuna. Tärkeimmiksi perusasioiksi määrittelen mm. ruudunhallinta ja peliobjektien törmäyksentunnistuksen. Toteutustapoja on useanlaisia ja tarkoituksena on toteuttaa toimiva peli yhdellä tapaa samalla verraten käytettyä tapaa muihin mahdollisiin toteutuskeinoihin ja löytää eri tapojen vahvuudet ja heikkoudet.

Käsittelen pelien ja peliohjelmoinnin yleisiä piirteitä luvussa 2. Esittelen mm. ruudunhallintaa ja tile engine –menetelmän ideaa ilman teknistä toteutusta. Luku 2 auttaa hahmottamaan pelien ja peliohjelmointiin liittyviä asioita yleisesti. Myöhemmissä luvuissa esittelen tekniset toteutukset näihin asioihin, jolloin keskityn itse tekniseen toteutukseen kun yleiset tiedot on jo luvussa 2 kerrottu.

Luvussa 3 esittelen XNA:ta ja luvussa 4 esittelen pelin suunnitelman ja kerron pelin ominaisuuksien määrittelyt. Luvussa 5 kerron pelin toteuttamisesta esimerkein ja vertailen omia ratkaisujani toisiin mahdollisiin ratkaisumalleihin.

2 PELIT JA PELIOHJELMOINTI

Tässä luvussa käyn läpi peleihin ja peliohjelmointiin liittyviä asioita. Läpikäytävät asiat pätevät monenlaisiin peleihin. Keskityn kuitenkin muutamissa kohdissa vain 2D-peleihin, enkä ota kantaa 3D-pelien toteutustapoihin. Jotkin käsiteltävät asiat, kuten ruudunhallinta, pätevät jokaiseen peliin.

2.1 Lajityypit

Pelit ovat usein hyvin erilaisia, mutta pitävät sisällään silti jotain yhteistä. Tästä syystä pelit voidaankin laittaa kategorioihin erilaisten ominaisuuksiensa perusteella. Pelejä voidaan lajitella pelin julkaisutyypin tai -alustan, pelin ominaisuuksien (ajopeli, roolipeli jne.), grafiikan (2D-, 3D-peli) tai jopa pelikehittäjän (Indie-, AAA-pelit jne.) taustan mukaan. Yleisimmin kuitenkin pelit eritellään juuri ominaisuuksien mukaan ja peli voi kuulua kerralla moneenkin erilaiseen lajityyppiin.

Lajityyppit voidaan rajata päälajityyppeihin ja tarkempiin lajityyppeihin. Päälajityypit ovat yleisempiä lajityypin kuvauksia ja kaksi erilaista peliä voivat kuulua samaan päälajityypiin. Tarkemmat lajityypit ovat tarkennettuja kuvauksia päälajityypistä. Päälajityyppi voi pitää sisällään monta erilaista tarkentavaa lajityyppiä. Lajityyppien, varsinkin päälajityyppien, määrä on aika vakio, mutta pelit kehittyvät ja sitä kautta on tarkempia lajityyppejä tullut lisää (Video game genres 2013). Esimerkkinä päälajityypistä on esimerkiksi ”strategia” ja sen kaksi tarkempaa lajityyppiä ”vuoropohjainen strategia” ja ”tosiaikainen strategia”.

Päälajityypit ovat:

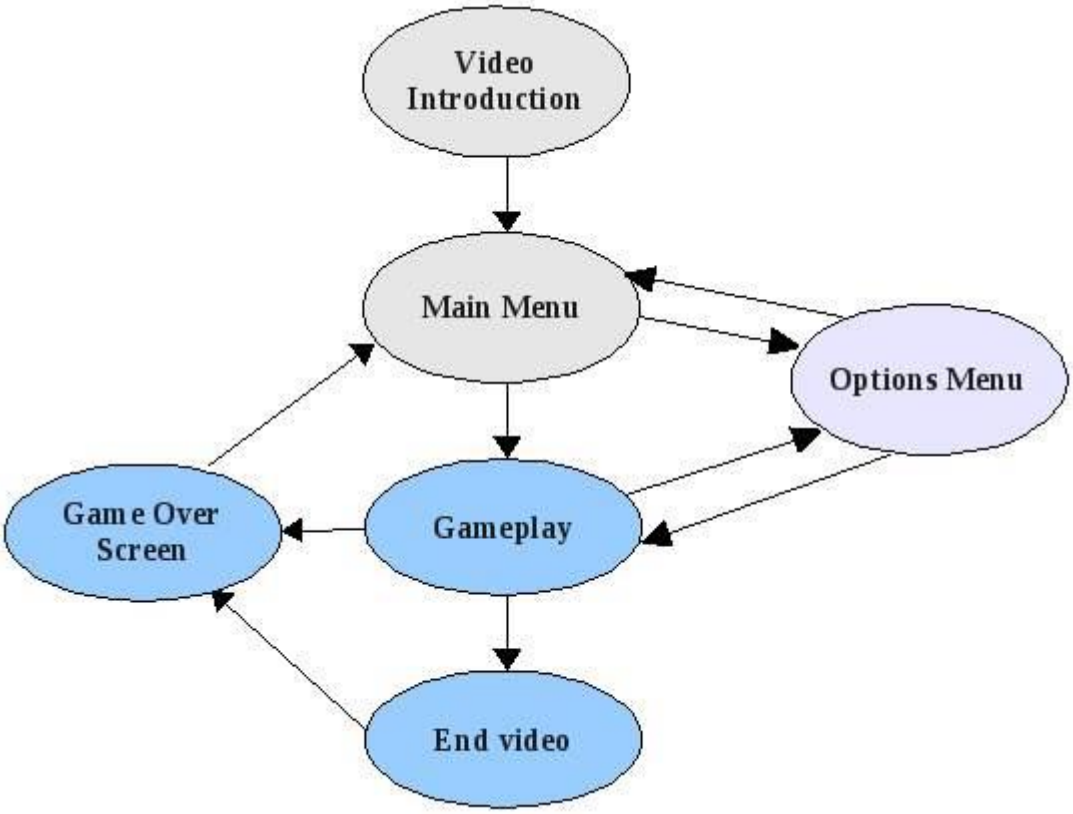
- Toiminta
- Seikkailu
- Roolipeli
- Simulaatio

- Strategia
- Muut lajityypit.

2.2 Ruudunhallinta

Pelit pitävät sisällään lajityypistä riippumatta aina ruudunhallintaa. Ruudunhallinnalla tarkoitetaan pelin eri tilojen hallintaa. Voidaan ajatella, että jokainen pääasia pelirakenteessa koostuu yhdestä ruudusta tai pelitilasta. Pelitiloille on ominaista, että tietystä pelitilasta pääsee vain tiettyihin toisiin pelitiloihin.

Kuvitellaan kuvan 1 mukainen pelirakenne, joka käynnistettäessä näyttää alussa aloitusvideon (Video Introduction) ja sen jälkeen aloitusvalikon (Main menu). Tässä on jo käytössä kaksi ruutua. Ensimmäinen ruutu on aloitusvideo. Aloitusvideon ruutu pitää sisällään kaiken aloitusvideoon liittyvän ohjelmakoodin ja sisällön. Aloitusvideon loputtua siirrytään aloitusvideo–ruudusta aloitusvalikko–ruutuun. Aloitusvalikko-ruutu pitää itselleen ominaiset asiat sisällään, esimerkiksi eri vaihtoehtoja, minne pelaaja haluaa mennä. Näitä ovat esimerkiksi uuden pelin aloittaminen, vanhan pelin lataaminen tai asetusten katsominen.



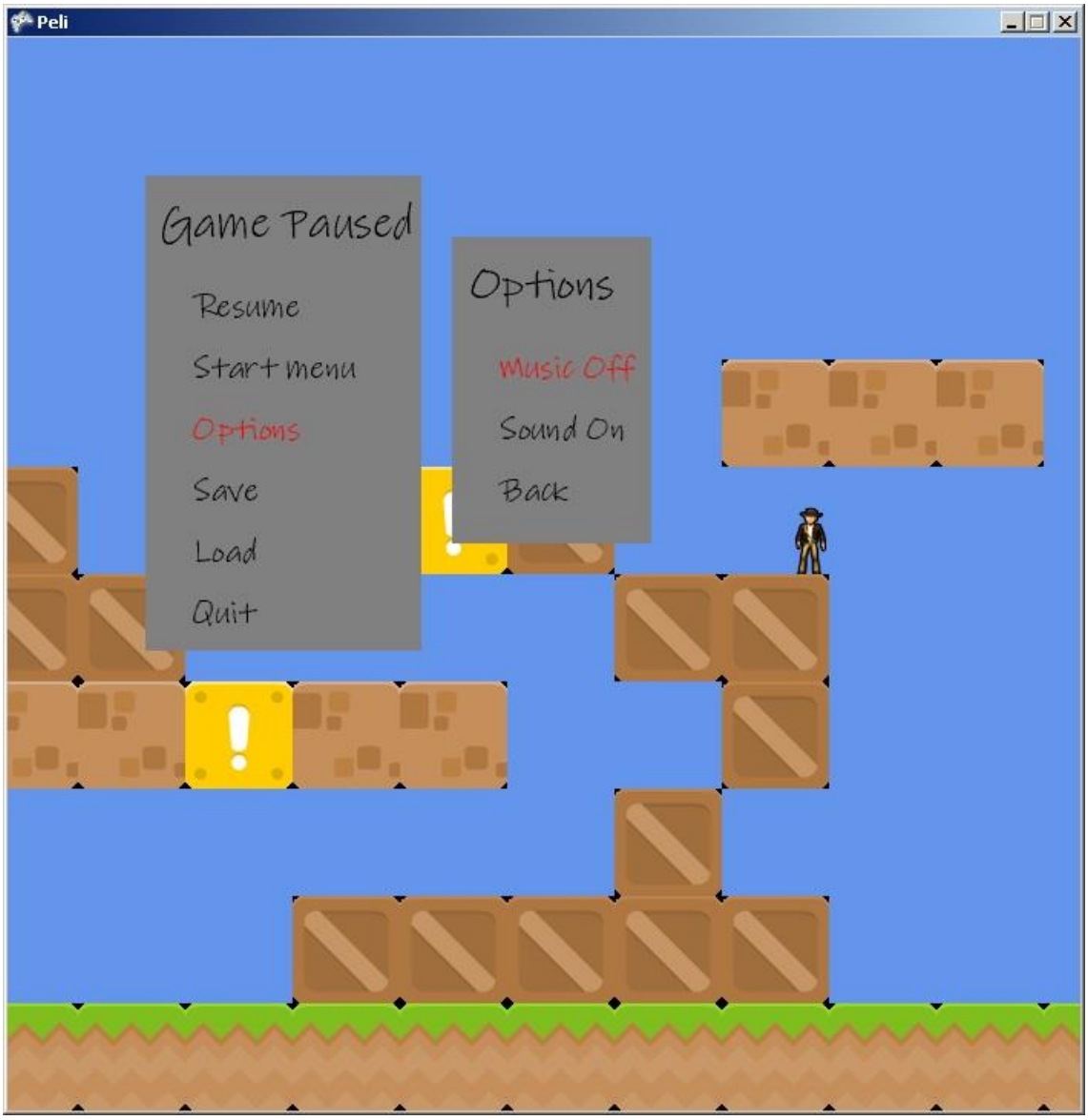
Kuva 1. Esimerkki pelitiloista ja siirtymisistä toisiin pelitiloihin (State based games 2013).

Ruudunhallinnan perusidea on se, että peli pilkotaan tiettyihin palasiin, eli ruutuihin, jolloin jokainen ruutu pitää sisällään vain itseään koskevat toiminnot. Ruudunhallinnalla pyritään saamaan ohjelmakoodi loogiseen, järjestettyyn ja helposti hallittavaan muotoon (State based games 2013).

Pelitiloihin liitetään yleensä kaksi ominaisuutta: näkyvyys ja aktiivisuus. Kun tietty pelitila nähdään (esim. aloitusvalikko) niin se on myös aktiivinen eli se ottaa vastaan pelaajan syötteitä. Useasti on näin, mutta on myös mahdollista, että pelitila näkyy, mutta ei ole aktiivinen. Tällainen tilanne tulee vastaan vaikka pelin taukovalikossa.

Kuvassa 2 nähdään tässä opinnäytetyössä toteutetun pelin ruudunhallintaa. Kuvassa on näkyvissä 3 ruutua: Game Paused -ruutu, Options -ruutu ja

taustalla oleva peliruutu. Näistä ainoastaan Options –ruutu on sekä näkyvä että aktiivinen. Options –ruudun vierellä näkyvä Game Paused –ruutu ja peliruutu näkyvät, mutta eivät ole aktiivisia. Erilaisia pelin tiloja on mahdollista pitää useampia samanaikaisesti voimassa.

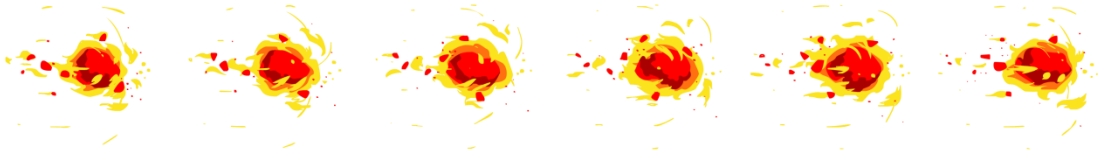


Kuva 2. Esimerkki pelitilojen ominaisuuksien näkyvyys ja aktiivisuus käytöstä.

2.3 2D-kuvan piirtäminen ruudulle

2D-peleissä erilaiset piirrettävät pelisisällöt koostuvat spriteistä. Sprite-grafiikka perustuu pikseligrafiikkaan, joka muodostaa pelihahmot, tavarat ja pelin muun graafisen sisällön. Sprite on aina suorakulmion muotoinen, mutta se voi sisältää joko kokonaan tai osittain läpinäkyviä pikseleitä. Peleissä spritet on kerätty yleensä isompiin kuvatiedostoihin, joita kutsutaan spritesheetiksi. Spritesheetin käyttö helpottaa pelihahmojen animoimista ja ohjelmoimista, kun spritet sijaitsevat yhdessä ja samassa spritesheetissä. Spritesheet vähentää myös pelin laitteistovaatimuksia muistin suhteen (What is a sprite sheet 2013).

Kuvassa 3 näkyy pieni spritesheet, jossa yksittäistä tulipalloa kutsutaan spriteksi. 2D-grafiikalla toteutetussa pelissä animointi tapahtuu kuvan 3 tapaisen spritesheetin avulla. Kuvassa näkyy kuusi hieman toisistaan eroavaa tulipalloa. Niiden animointi toteutetaan piirtämällä ensin ensimmäinen sprite vasemmalta, sen jälkeen samaan kohtaan nopeasti vasemmalta katsoen seuraava sprite jne. Tällä tavalla saadaan vaikutelma animaatiosta, kun spritet piirretään vuorotellen hyvin lyhyessä ajassa.



Kuva 3. Spritesheet ja tulipallon animaatio (2d-sprite animation tutorial 2013).

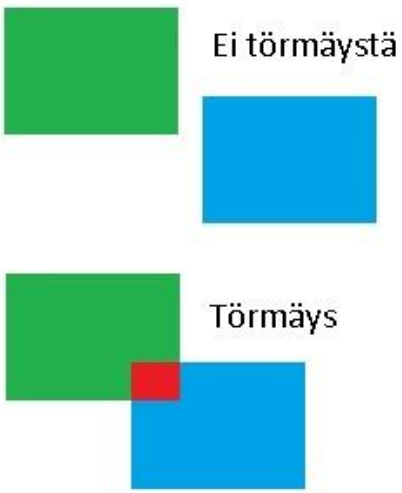
2.4 Törmäystesti

Jokaiselle pelille oleellinen asia on törmäystesti (collision test). Törmäystestin tehtävä on tarkistaa eri peliobjektien törmäykset. Törmäystestit voidaan luokitella kolmeen erilaiseen testiin, kun tarkoituksena

on tutkia aktiivisten peliobjektien keskinäisiä törmäyksiä. Nämä kolme testiä ovat: bounding box -, circle - ja per pixel collision -tekniikat. Kaikki ovat erilaisia ja riippuen tilanteesta jokin kolmesta törmäystestistä on parhaiten soveltuvin. Yhtä ja parhainta tekniikkaa, jota voisi joka kohdassa käyttää, ei ole.

2.4.1 Bounding box collision

Ensimmäinen ja helpoin tapa testata kahden objektin mahdollista törmäystä on bounding box collision -menetelmä. Bounding box collision -menetelmä liittyy tapaan, miten kuva piirretään ruudulle. Sprite luokkaa luodessa esille nousi se, että sprite on suorakulmainen kuva, tai osa isommasta kuvasta, joka piirretään ruudulle. Nämä osat koostuvat pikseleistä, osa läpinäkyvistä ja osa näkyvistä. Bound-box collision -menetelmässä on tarkoituksena tarkastaa, menevätkö pelissä olevien kahden eri sprite -objektin suorakulmat päällekkäin kuten kuvassa 4 on havainnollistettu. Bounding box collision –menetelmässä ei törmäystesti ota kantaa siihen, ottavatko vain näkyvät pikselit yhteen vaan törmäys tapahtuu jo silloin, kun läpinäkyvätkin pikselit ovat päällekkäin (Bounding box collision 2013).



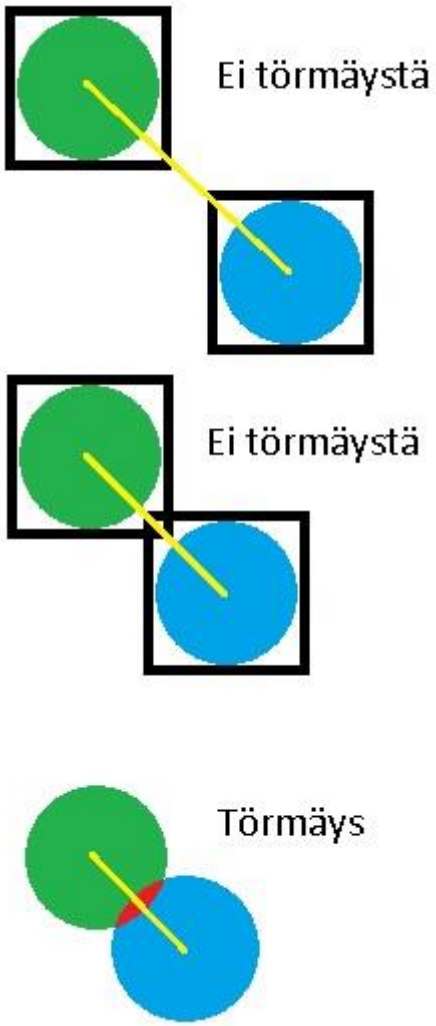
Kuva 4. Bounding box collision.

Bounging box collision –menetelmää sovelletaan tilanteissa, jossa molemmat törmäykseen osallistuvat objektit ovat joko kokonaan tai osittain sovitettavissa suorakulmion muotoon.

2.4.2 Circle collision

Joskus objekti on muodoltaan sellainen, ettei sen törmäystä pysty bounding box collision –menetelmän avulla kätevästi tarkastamaan, eikä pikselin tarkalle törmäystestille ole tarvetta. Tällainen tilanne syntyy vaikka kahden pallon törmäystä tarkastaessa. Ideana on tällöin se, että pyöreän objektin sädettä verrataan törmäävän pyöreän objektin säteeseen. Jos näiden kahden objektin säteet yhteenlaskettuina on suurempi lukuarvo kuin objektien keskipisteiden välinen etäisyys, syntyy törmäys (Circle collision 2013).

Kuvassa 5 keskimmäisiä palloja katsoessa huomaa, että bounding box collision –tekniikalla saataisiin törmäys aikaiseksi, koska kahden spriten reunat (mustat viivat) menevät päällekkäin, vaikka itse pallot eivät toisiaan kosketa. Vasta kuvan 5 alimmassa kohdassa (spriten reunat poistettu esimerkistä selvyyden vuoksi) tapahtuu törmäys, kun kahden pallon keskipisteiden etäisyys (keltainen viiva) on pienempi kuin pallojen säteiden summa.

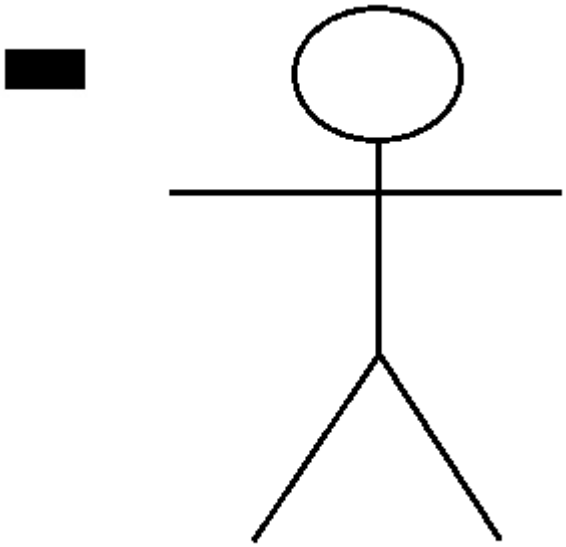


Kuva 5. Circle collision.

2.4.3 Per pixel collision

Raskaimpana, mutta kaikista tarkimpana törmäystä testaavista menetelmistä on per pixel collision -menetelmä. Per pixel collision -menetelmä tulee pakolliseksi, kun kuvia aletaan skaalaamaan erikokoisiksi, tai kuva kierretään tai kuva ei ole pyöreän tai suorakulmion muotoinen objekti. Kuvassa 6 näkyvä panos ja tikku-ukko ovat kaksi törmäävää objektia. Ainoastaan panos on suorakulmion muotoinen eikä koko tikku-

ukkoa pysty millään saamaan joko ympyrän tai suorakulmion muotoiseksi, joten kahta edellä mainittua törmäystestiä ei voi tässä soveltaa.



Kuva 6. Sprite tikku-ukosta ja panoksesta.

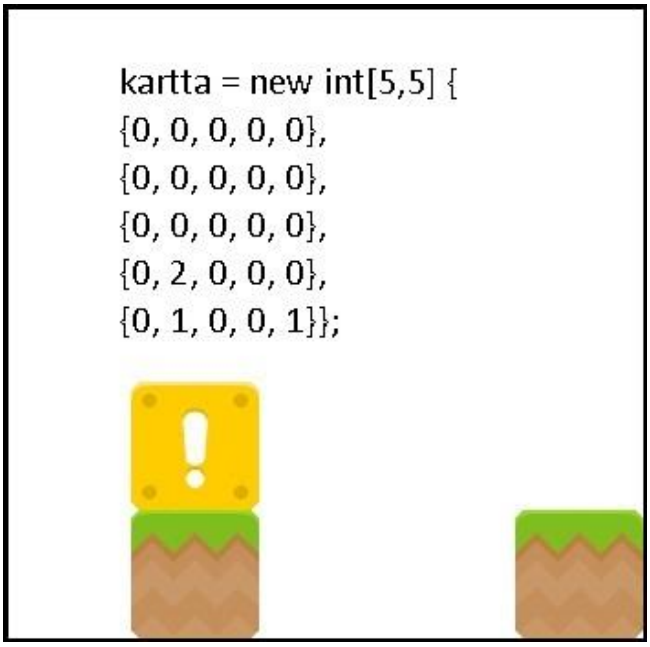
Per-pixel collision on raskas tapa testata törmäys, mutta sen avulla saadaan täydellinen törmäystesti luotua. Kun tämä testi tehdään oikein, se osaa ottaa huomioon läpinäkyvät pikselit, skaalatut ja kierretyt kuvat. Per-pixel collision –menetelmä käy läpi objektin kuvan kaikki pikselit, täysin läpinäkyvistä täysin näkyviin pikseleihin, ja vertaa niiden koordinaatteja toisen objektin pikselien koordinaatteihin. Tarkistus osaa ottaa yksittäisen pikselin koordinaatit ja tiedot jopa skaalatusta ja kierretystä kuvasta. Tämä on mahdollista, kun käytetään matriisilaskentaa (Coll detection overview 2013).

XNA:ssa on erilaisia matriiseja, joista jokainen mittaa tiettyä arvoa kuten kuvan kääntökulmaa tai vaikka skaalausta. Matriisien vertailua varten on olemassa yksi kokonainen matriisityyppi, johon voidaan sisällyttää kaikki pienemmän matriisikokonaisuudet, ja näin verrata yhtä isoa matriisia toiseen isoon matriisiin niin, että kaikki oleelliset tiedot ovat niissä tallennettuna.

Per-pixel collision -menetelmä on raskas ja vaatiikin erityistä huomiota, kun sen toteuttaa. Bounding box collision - ja circle collision -testit voi suorittaa jokaiselle objektille jokaisella pelin päivityskierroksella, mutta per-pixel collisionin –tekniikan kanssa näin ei voi tehdä. Per-pixel collision -menetelmää varten käytetään toista kahdesta edellämainittusta törmäystestistä ensin testaamaan, ovatko tietyt objektit (esimerkiksi ammus ja vihollinen) niin lähellä toisiaan, että testin tulos näyttää positiivista. Kun jompikumpi törmäystesti palauttaa positiivisen tuloksen, eli niiden mukaan törmäys tapahtuu, käynnistetään per-pixel collision -menetelmä tarkastamaan, kohtaavatko kahden objektin näkyvät pikselit toisensa.

2.5 Tile engine -tekniikka

Pelimaailma toteutetaan usein 2D-peleissä lajityypistä riippumatta yleisesti käytetyn tekniikan, tile engine -menetelmän, avulla. Tässä on ideana luoda pelimaailma, joka muodostuu pienistä numeroiduista osioista, soluista, joissa jokaisella numerolla on oma viittauksensa tiettyyn kuvaan. Tile-engine tekniikka on ollut jo kauan tunnettu hieman monipuolisempaa peligrafiikkaa tukevista pelikonsoleista lähtien (Tile engine 2013). Kuvassa 7 on esimerkki tile engine –tekniikasta.

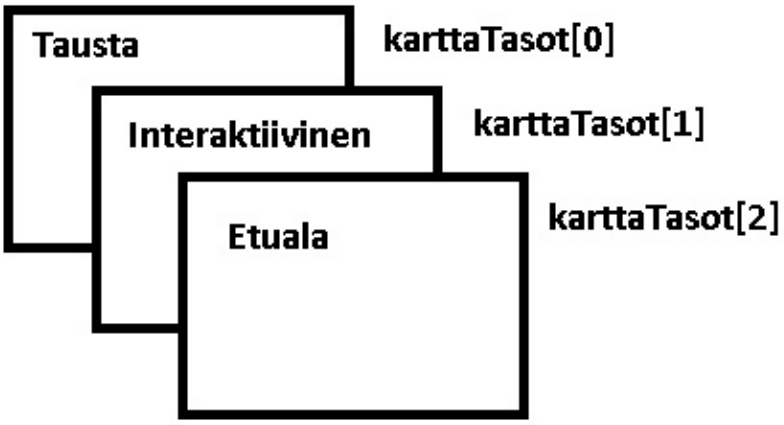


Kuva 7. Esimerkkikuva tile engine –tekniikasta.

Tile engine –menetelmä luo kartan, joka koostuu soluista. Solut ovat ohjelmakoodissa numerokoodattuja alueita. Kuvassa 7 näkyy muuttuja ”kartta”, joka on kaksiulotteinen taulukko. Taulukko pitää sisällään arvoja, joista jokaisella arvolla on oma graafinen vastikkeensa. Kuvan 7 taulukon mukaisesti arvo 0 vastaa ”ei mitään”, eli ohjelma ei piirrä arvon 0 kohdalle mitään. Kuvan 7 esimerkissä arvoa 1 vastaavaksi kuvaksi on asetettu kuva maakappaleesta, jossa on hieman ruohoa ja arvon 2 graafinen vastakappale on keltainen laatikko jossa on huutomerkki. Menetelmä piirtää ruudulle jokaista arvoa vastaavan kuvan. Arvoja vastaavat kuvat on koottu yhteen suurempaan kuvatiedostoon ja näitä pienempiä kuvia kutsutaan nimellä tile.

Tile engine –menetelmä voi tukea muitakin muotoja kuin suorakulmion muotoisia tile-elementtejä, esimerkiksi kuusikulmion muotoisia tilejä. Myös isometrisestä kuvakulmasta toteutettu 2D-peli on mahdollista luoda tile engine menetelmällä. Muitakin mahdollisia muotoja on, mutta kolme edellä mainittua tapaa ovat yleisimmät. (Tile engine 2013).

Tile engine –menetelmä pystyy pitämään sisällään myös erilaisia pelaajalle näkymättömiä sääntöjä peliobjekteille. Tämän voi toteuttaa kuvan 8 mukaisella monitasoisella tile engine –menetelmällä.



Kuva 8. Esimerkki monitasoisen tile engine -menetelmän ideasta.

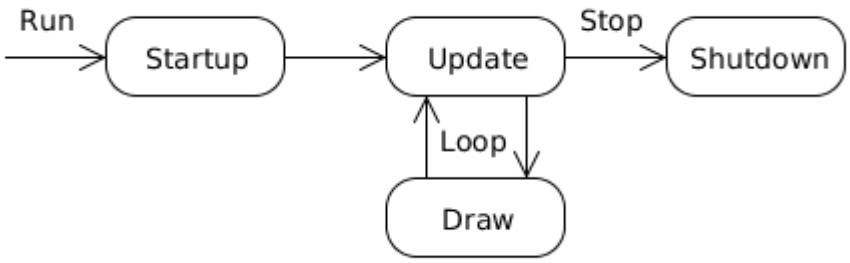
Monitasoisella tile engine –menetelmällä voidaan yhdelle solulle antaa monta eri arvoa. Solu voi esimerkiksi sisältää taustaspriten, joka piirretään solun taustalle, interaktiivisen osion, jossa on pelaajaan tai muihin pelin objekteihin vaikuttava sääntö, joka on kuitenkin pelaajalle näkymätön tai näkyvä, sekä muita tasoja (Jaegers 2010, 294-296).

2.6 Pelin runko

2.6.1 Yleistä

Peleillä on aina tietynlainen runko, jossa pelin logiikka etenee. Peleillä on yksinkertaisimmillaan alku, loppu, päivitys- ja piirtokohdat (Game engine design the game loop 2013). Riippuen muutamasta asiasta, kuten esimerkiksi pelin lajityypistä tai käytetystä pelikirjastosta, voi rakenne

vaihdella hieman. Kuvassa 9 nähdään yksinkertainen malli yleisestä pelirungosta.



Kuva 9. Pelin eteneminen alusta loppuun (game engine design the game loop 2013).

Kuvan 9 pelirunko sisältää neljä kohtaa: Startup, Update, Draw ja Shutdown. Kuvassa on osoitettu nuolilla pelin eteneminen. Kun peli käynnistetään, peli lataa grafiikat, äänet ja alustaa pelin objekteille oletusarvot kohdassa Startup. Kohdassa shutdown vapautetaan pelin tiedot koneen muistista.

2.6.2 Update- ja Draw -osiot

Pelirungon tärkein kohta on Update- ja Draw–kierto. Kun peli on käynnistetty ja peli on ladannut kaikki tarvittavat sisältönsä (äänet, kuvat yms) ja alustanut tiedot, se jää kiertämään ikuisesti kehää, kunnes peli lopetetaan. Kierto etenee Update-kohdasta Draw-kohtaan ja sieltä takaisin Update-kohtaan. Toisin kuin tavallisimmissa hyötyohjelmissa, esimerkiksi Microsoft Wordissä, pelin Update- ja Draw-osiot osiot suoritetaan riippumatta siitä, saako peli pelaajalta syötteen, kuten näppäimen painalluksen, kun hyötyohjelmissa ohjelma etenee vasta tiettyjen tapahtumien (esimerkiksi näppäimistön painallus) toteuduttua. (Game programming 2013).

Update-kohdassa toimii pelin logiikka. Siellä otetaan talteen pelaajan syötteet, tarkistetaan törmäykset ja päivitetään muut peliobjektit kuten vihollisten liikkeet ja tekoälyn reaktiot. Myös mahdolliset ääniefektit suoritetaan Update-osiossa.

Kun Update-osio on suoritettu kokonaan läpi, peli siirtyy Draw-osioon. Draw-osiossa peli piirtää ruudulle kaikki tarvittavat peliobjektit. Kun objektit on piirretty ruudulle, peli siirtyy Update-osioon takaisin.

Update- ja Draw-osiot ovat tarkoituksella erotettuina toisistaan. Esimerkiksi jos peli on hyvin iso ja raskas ja tietokone, jolla peliä pelataan, ei ole tarpeeksi tehokas, voi olla, että Draw-osio alkaa jäämään jälkeen Update-osiosta ja pelin FPS-arvo (Frames Per Second) laskee. FPS:llä tarkoitetaan sitä, montako kertaa sekunnissa peli piirtää kuvan ruudulle, eli montako kertaa Draw-osio suoritetaan sekunnissa. Jos Update-osio suoritetaan sekunnissa 60 kertaa, ei Draw-osio välttämättä pysty samaan. Tämän voi ratkaista ohittamalla joka toinen Draw-osio, jolloin pelilogiikka (Update-osio) päivittyy kaksi kertaa nopeammin kuin kuvien piirto ruudulle (Draw-osio). Näin peli ei vaadi tietokoneelta yhtä paljon ja kumpikaan osio ei niin herkästi jää jälkeen (Understanding the game loop basix 2013).

3 XNA

XNA on .NET -ohjelmointikielien kanssa yhteensopiva kirjasto, joka on toteutettu DirectX kirjaston päälle ja jonka tarkoituksena on alusta lähtien ollut antaa pelikehittäjille helppo tapa kehittää pelejä. XNA:n ensimmäinen versio julkaistiin vuonna 2006. Uusin vakaa versio XNA 4.0 julkaistiin vuonna 2011 (Microsoft XNA 2013). Yleensä myös XNA-pelikehitykseen käytetään joko ilmaista Microsoftin Visual Studiota tai sen maksullisia versioita. Ilmainen XNA ja Visual Studion ilmainen versio Express ovat Microsoftin sivuilta vapaasti ladattavissa. XNA:lla on Visual Studioon omia moduuleja, joilla voidaan nopeasti suunnata peli joko Windows Phone -alustalle tai Windows- ja XBOX360 -alustoille.

XNA on lyhyessä ajassa noussut suosituksi työkaluksi toteuttaa pelejä eri alustoille. Se tukee sekä 2D- että 3D-grafiikkaa. Vaikka XNA tukeekin .NET –ohjelmointikieliä, niin virallinen tuki on kuitenkin vain C#- ja Visual Basic.NET –kielille. C#-ohjelmointikieli on näistä noussut suosituimmaksi. (How to learn XNA 2013).

XNA on hyvin käyttäjäystävällinen, sillä se on hyvin dokumentoitu, siihen on paljon oppaita ja se on suunniteltu hyvin helppokäyttöiseksi. XNA-kirjasto onkin saavuttanut suosiotaan varsinkin peliohjelmoinnin harrastajissa ja Indie –pelejä tekevien keskuudessa. XNA:lla on toteutettu muutamia hyvinkin tunnettuja ja palkintoja tai kunniamainintoja keränneitä pelejä, kuten esimerkiksi peli Bastion (Super Giant Games 2013).

XNA on suunniteltu Microsoft Visual Studio -ympäristöllä kehitettäväksi. Visual Studio pystyy generoimaan muutamat luokkatyypit automaattisesti XNA:ta varten. Visual Studion avulla pystyy myös tekemään erilaisia testejä, kuten kuinka paljon peli käyttää muistia ja prosessoria. Visual Studion avulla oman pelin saa myös muutettua asennustiedostoksi, joka pitää sisällään pelin ja sen tarvitsemat komponentit.

XNA:ta on kritisoitu vahvasti siitä, että se tukee vain tiettyjä käyttöjärjestelmiä. Virallisesti asia onkin näin, mutta tähän on tarjonnut

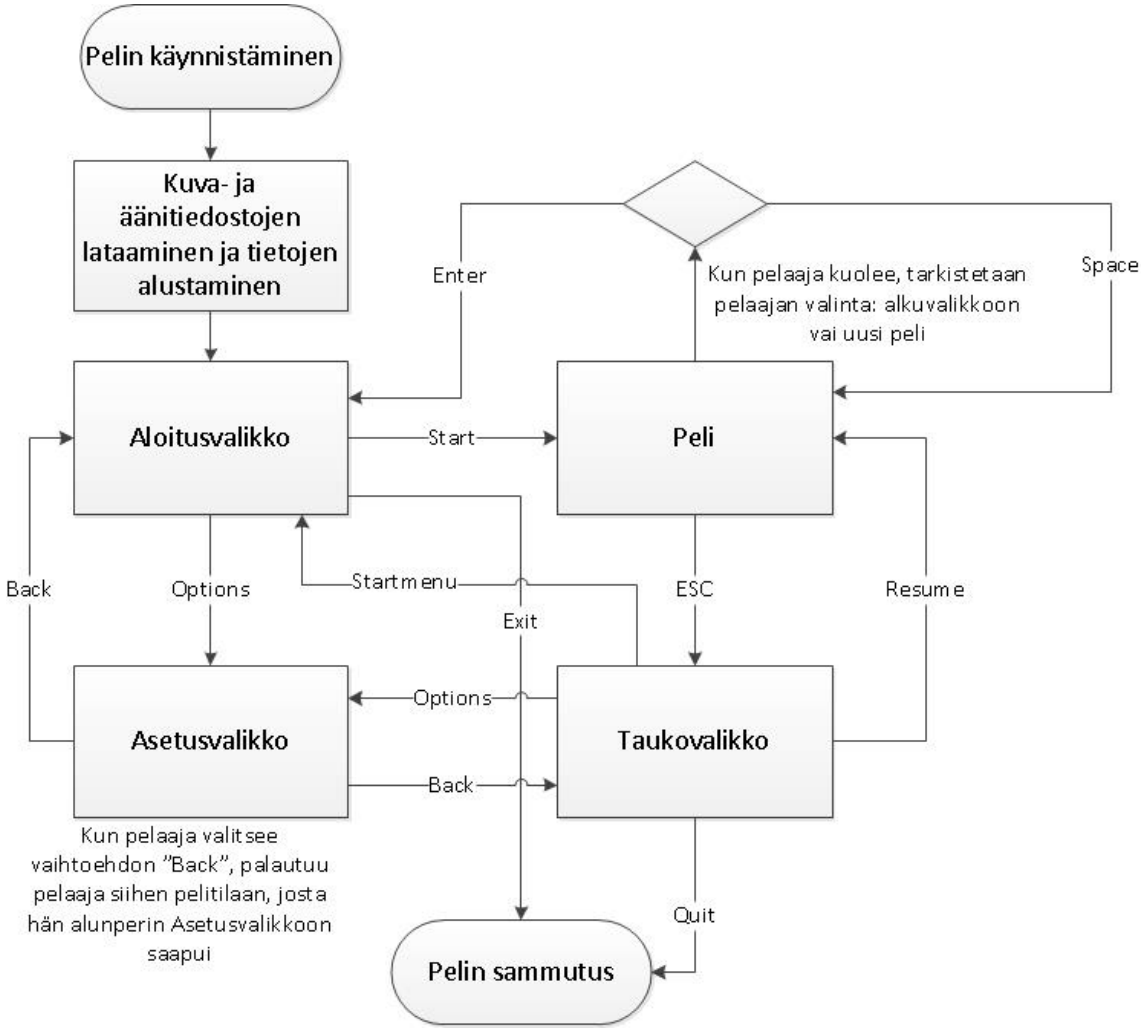
ratkaisun vapaalla lähdekoodilla toteutettu MonoGame (Monogame 2013). MonoGame:n tarkoitus on kääntää XNA:lla toteutetut pelit muihin käyttöjärjestelmiin, kuten Linux -, Mac OS X -, iOS – ja Android-käyttöjärjestelmiin.

4 ESIMERKKIPELIN SUUNNITTELU JA MÄÄRITTELY

Toteutettavan esimerkkinä toimivan pelin lajityypiksi valitsin tasoloikkatyylisen 2D-grafiikkaan perustuvan pelin. Päädyin tähän, koska tällaiseen peliin pystyy sisällyttämään monta eri osa-aluetta 2D-peliohjelmoinnista. Pelin avulla päästään tutustumaan tile engine –menetelmää hyödyntävään toteutukseen ja peli tulee sisältämään bounding box collision - ja per pixel collision -menetelmät. Kutsun toteutettavaa peliä nimellä Esimerkkipeli.

Esimerkkipelissä pyrin kokoamaan paljon erilaisia (2D-)peliohjelmointiin liittyviä asioita, joita opinnäytetyössä on jo aiemmin esitelty. Esimerkkipelin on tarkoitus toimia esimerkkinä, miten pelin voi yhdellä tavalla toteuttaa. Samalla myös pohditaan, miksi mikäkin ratkaisu on toteutettu niin kuin se on toteutettu.

Kaaviossa 1 on havainnollistettu esimerkkipelin erilaisten pelitilojen kulku. Esimerkiksi Aloitusvalikko-ruudusta saadaan avattua ruudulle Asetusvalikko valitsemalla vaihtoehto Options. Pelitiloja vaihdetaan, käynnistetään ja suljetaan valitsemalla tietyn niminen vaihtoehto, kuten äskettäisessä esimerkissä valittiin vaihtoehto Options. Osa ruuduista aktivoidaan myös suoraan näppäimenpainalluksilla (ESC-, Space- ja Enter-näppäimet), jos ollaan tietyssä ruudussa. Esimerkiksi jos Peli-ruudussa, jossa itse peli toimii, painetaan ESC-näppäintä, laittaa se pelin tauolle ja avaa Taukovalikon. Vastaavasti Enter- ja Space-näppäimillä on omat toimintansa.



Kaavio 1. Pelin tilakaavio.

Esimerkkipelin ominaisuudet ovat:

- Ikkunoitu, ei skaalautuva peliruutu resoluutiolla 700 x 700 pikseliä
- Pelissä on kerrallaan vain yksi pelitila aktiivisena
- Pelialueen korkeus on sama kuin ruudunkorkeus, pelin kamera seuraa vain vaakatasossa tapahtuvaa liikettä ja seuraa pelaajaa sen mukaan
- Pelissä on 2 erilaista vihollistyyppiä, molemmilla omanlaisensa tapa liikkua
- Pelaaja tuhoutuu koskiessaan viholliseen, viholliset tuhoutuvat pelaajan ampumaan panokseen
- Panoksia ruudulla maksimissaan vain 3 kappaletta
- Pelaaja voi liikkua vasemmalle ja oikealle sekä hyppiä

- Pelissä on 2 erilaista karttaa
- Pelissä on kaksi erilailla pelaajan liikkeisiin reagoivaa laattaa
- Pelissä on mahdollisuus laittaa peli tauolle
- Peliä pelataan ja valikkoja ohjataan vain näppäimistöllä.

Esimerkkipeli pyritään toteuttamaan olio-ohjelmoinnin sääntöjä noudattaen. Esimerkkipelin grafiikat ja äänet hankitaan internetistä ja näiden lisenssien tulee olla sellaiset, että niitä saa vapaasti käyttää ja lisenssien ehtoja pyritään kunnioittamaan. Lisenssien mahdolliset vaatimat maininnat listataan pelin alkuvalikkoon. Esimerkkipeliä ei ole tarkoitus missään vaiheessa julkaista vaan sen tehtävä on toimia tässä opinnäytetyössä esimerkkitapauksena.

5 ESIMERKKIPELIN OHJELMOINTI

5.1 Yleistä

Tässä luvussa esitellään Esimerkkipelin tärkeimmät kohdat. Käyn läpi, miksi olen päättänyt toteuttaa eri osat tietyllä tavalla ja vertailen valintaani muihin mahdollisiin toteutusvaihtoehtoihin. Pyrin löytämään hyödylliset ominaisuudet valitsemiltani toteutustavoilta. Poimin lähdekoodista aiheen mukaan ne kriittisimmät kohdat esille. Mainitsen, mitä luokkia toteutin ja mitä luokkaa käytän jokaisessa kohdassa. Luokkien sisältämä ohjelmakoodi esitetään tilanteen mukaan siltä osin kuin se on oleellista. Tavoitteenani on saada esitettyä pelin kannalta oleellisten kohtien toimintatapa vaihtoehtoineen.

Esimerkkipelin toteutin Microsoftin kehittämillä työkaluilla kuten XNA-pelikirjastolla ja Microsoft Visual Studio 2010 –kehitysympäristöllä C#-ohjelmointikielellä. Peli on suunniteltu ja toteutettu olio-ohjelmoinnin mukaisesti.

5.2 Ruudunhallinta

Esimerkkipelin ohjelmoimisen aloitin luomalla pelin ruudunhallintajärjestelmän. Ruudunhallintajärjestelmästä on hyvä aloittaa, koska kaikki muut pelin osaset rakennetaan sen sisälle. Ruudunhallintaan on olemassa monta erilaista tapaa, mutta kaksi erityistä tapaa on noussut esille erilaisia oppaita tutkiessa. Ensimmäinen tapa (kuva 10) on kaikkein yksinkertaisin, nopein ja helpoin tapa toteuttaa. Siinä on ideana, että eri pelitilat on eroteltu toisistaan valintalausein ja pelitilaan sidottu ohjelmakoodi on sijoitettu valitun valintalauseen sisälle. Näin ollen riippuen siitä, minkä pelitilan halutaan olevan voimassa, peli suorittaa vain tietyn osan ohjelmakoodia. Näin voidaan esimerkiksi laittaa kaikki päävalikkoa koskevat ohjelmapalaset yhden valintalauseen sisälle ja kun ehto on tosi, suoritetaan sen sisällä olevat ohjelmalauseet. Näitä valintalauseita luodaan kaksi

kappaletta. Ensimmäinen sijoitetaan Update- ja toinen Draw-osioon. Näin peli tietää, mitä päivitystä seurata ja mitä piirtää ruudulle.

```
//1 = päävalikko, 2 = peli
if (screen == 1)
{
        //päävalikkoa koskeva ohjelmakoodi tulee tänne
}
else if (screen == 2)
{
        //peliä koskeva ohjelmakoodi tulee tänne
}
```

Kuva 10. Ruudunhallinta valintalausein.

Kuvan 10 mallin mukaan toteutettuna kaikki ohjelmalauseet ovat samassa tilassa, useimmiten XNA:n generoiman Update-metodin sisällä eriteltyinä valintalauseilla. Tämä tapa toimii pienissä peleissä, joissa pelitiloja on muutama. Kuvan 10 tavalla ei kuitenkaan saa Esimerkkipelin vaatimaa taukovalikkoa toteutettua eikä tapa oikein ole olio-ohjelmoinnin mukainen. Vaikkei opinnäytetyössä tehtävä peli ole iso tai monimutkainen, toteutetaan sen ruudunhallinta kuitenkin hieman erilaisella mutta paremmin käyttöön taipuvan ratkaisun avulla, jotta peliin saadaan helposti toteutettua pelin keskeyttävä taukovalikko.

Toinen ja myös Esimerkkipelissä käytetty toteutustapa noudattaa olio-ohjelmoinnin tyyliä. XNA:ssa on valmiina DrawableGameComponent-luokka ja se on oleellisessa osassa tässä tapauksessa. DrawableGameComponent-luokka pitää sisällään omat Update- ja Draw-metodit täsmälleen samanlaisia kuin ne on XNA:n itse generoiman Game1-luokan (pelin pääluokka) sisälläkin. DrawableGameComponent-luokkaa voidaa pitää samanlaisena luokkana kuin Game1-luokka on. Jos Game1-luokkaa kutsuu pelin pääluokaksi, niin DrawablegameComponent on komponentti, joka suoritetaan Game1-luokan jälkeen automaattisesti.

Luokka sisältää myös kaksi hyödyllistä ominaisuutta, enabled ja visible. Näiden käyttöön tutustutaan pian myöhemmin, mutta tämän luokan ja kahden edellä mainittujen arvojen käyttö mahdollistaa sen, että monta erilaista komponenttia voi olla samaan aikaan joko käytössä ja näkyvissä, tai pelkästään näkyvissä tai käytössä, joskin jälkimmäiselle ei yksinään helposti käyttöä keksi. Näiden avulla on mahdollista pilkkoa ohjelmakoodia pienempiin kokonaisuuksiin ja määrätä, mikä osa on milloinkin käytössä.

Koska pelissä on useampi peliruutu, luon luokan Screen, joka pitää sisällään jokaiselle peliruudulle yhtenäiset metodit ja muuttujat. Kaikki peliruudut tulevat periytymään tästä luokasta. Luokka Screen taas periytyy luokasta DrawableGameComponent.

Esimerkkipelin ruudunhallinta sijaitsee pääluokassa Game1.cs. Pääluokkaan luodaan jokaisesta peliruudusta oma muuttujansa, sekä yksi muuttuja taltioimaan nykyisen aktiivisen ruudun (Kuva 11).

```
Screen activeScreen; //aktiivinen ruutu
StartScreen startScreen; //aloitusvalikko
GameScreen gameScreen; //peliruutu
OptionsScreen optionsScreen; //asetusvalikko
PauseScreen pauseScreen; //taukovalikko
```

Kuva 11. Muuttujien luonti.

Luodut muuttujat alustetaan pelin käynnistyessä (kuva 12) ja samalla niille luovutetaan niiden tarvitsemat fontit ja grafiikat. Kun ruudut on luotu, ne kaikki piilotetaan samantien Hide–metodilla, lisätään ne Game1-luokan komponentteihin ja siirretään haluttu ruutu (Esimerkkipelin tapauksessa startScreen eli aloituvalikko) muuttujaan activeScreen ja määritetään se näkyväksi ja aktiiviseksi Show–metodilla (Kuva 12). Ruutujen lisääminen Game1-luokan komponentteihin mahdollistaa sen, että peli osaa itse automaattisesti käydä kaikkien komponenttien Update- ja Draw-metodit läpi,

riippuen siitä, ovatko ruutujen visible (näkyvyys) ja enabled (aktiivisuus) ominaisuudet arvoiltaan tosi. Tällä tavoin esimerkiksi aloitusvalikkoa koskevat ohjelmalauseet voidaan ohjelmoida luokan StartScreen Update-metodiin ja ohjelman lähdekoodista tulee helpommin hallittava.

```
//luodaan asetusvalikko
optionsScreen = new OptionsScreen(
this,
spriteBatch,
Content.Load<SpriteFont>("Fonts/Lindsey16"),
Content.Load<SpriteFont>("Fonts/Lindsey22"),
new Vector2(300,100),
whiteRectangle);
Components.Add(optionsScreen);
optionsScreen.Hide();

activeScreen = startScreen; //aloitusruudusta aktiivinen ruutu
activeScreen.Show(); // ruutu laitetaan näkyväksi ja aktiiviseksi
```

Kuva 12. Asetusvalikon luonti ja aktiivisen valikon määrittely.

Game1-luokan Update-metodissa valvotaan pelaajan näppäimistösyötteitä, verrataan niitä aktiivisena olevaan ruutuun ja aktiivisen ruudun sisältöön. Valintalauseella tarkistetaan, missä kohdassa (0-2) osoitin on ollut aloitusvalikossa ja suoritetaan sen mukainen toimenpide (Kuva 13). Esimerkiksi kun valitaan aloitusvalikosta ”Start”, on sen osoittimen numero 0 ja peli luo uuden pelin ja asettaa aktiiviseksi ikkunaksi peliruudun.

```
if (activeScreen == startScreen)
        {
            if (InputHandler.ButtonPressedAndReleased(Keys.Enter))
            {
                switch (startScreen.Selected)
                {
                    case 0:
                        //start – aloitetaan uusi peli
                        gameScreen.CreateNewGame(1); //luodaan uusi peli
                        activeScreen.Hide(); //visible ja active = false
                        activeScreen = gameScreen; //vaihdetaan aktiiviruutu
                        activeScreen.Show(); //visible ja active = true
                        break;
                    case 1:
                        //options - asetusvalikko
                        activeScreen.Enabled = false; //jätetään aloitusvalikko
                      //näkyviin mutta lopetetaan sen päivittäminen
                        activeScreen = optionsScreen;
                        activeScreen.Show();
                        break;
                    case 2:
                        //Exit – lopetetaan peli
                        this.Exit();
                        break;
                }
            }
        }
```

Kuva 13. Ruudunhallinta pääluokan Update-osiossa.

Toteutus toimii, mutta siinä on silti pieniä ongelmia. Koska kaikki kuvan 13 mukaiset ruudunhallintatoimet joudutaan ohjelmoimaan Game1-luokan sisälle, tulee ruudunhallinnasta hankala hallittava, jos ruutujen määrä on suuri.

5.3 Kamera

Esimerkkipelin kamera toteutetaan staattisena Camera –luokkana. Se sisältää runsaasti erilaisia metodeja, joiden tehtävänä on palauttaa erilaisia tietoja kameraan ja pelimaailmaan liittyen. Kuvassa 14 on esiteltynä Camera-luokan oleellisimmat esimerkkipeliä koskevat ominaisuudet ja metodit.

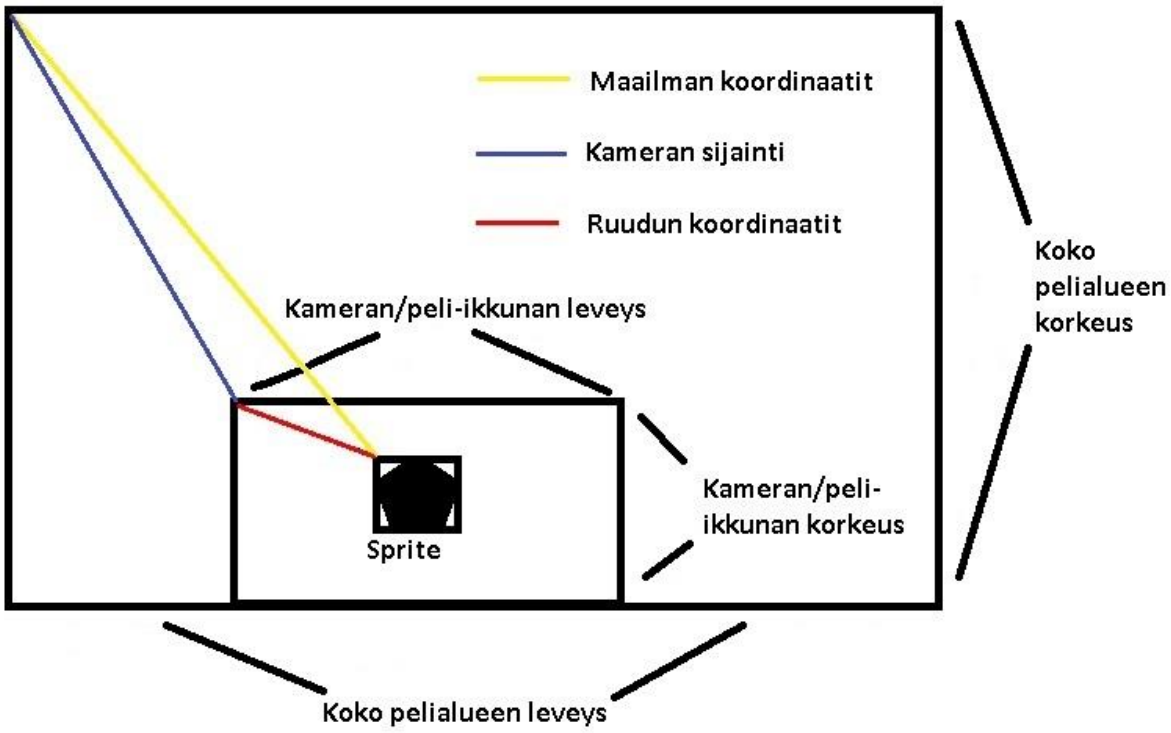
```
public static Vector2 Position
{
    get { return position; }
    set
    {
        //rajaa kameran sijainnin niin, ettei kamera poistu pelialueelta
        position = new Vector2(
            MathHelper.Clamp(value.X,0,worldWidth - screenWidth),
            MathHelper.Clamp(value.Y,0,worldHeight - screenHeight));
    }
}
public static bool ObjectIsVisible(Rectangle bounds)
{
    //palauttaa Tosi, jos bounds on kameran näköalueella
    return CameraScreen.Intersects(bounds);
}

public static Vector2 TransformWorldToScreen(Vector2 point)
{
    //muuttaa mailman koordinaatit ruudun koordinaattiin
    return point - position;
}

public static Rectangle TransformWorldToScreen(Rectangle rectangle)
{
    //muuttaa mailman koordinaatit ruudun koordinaattiin
    return new Rectangle(
        rectangle.Left - (int)position.X,
        rectangle.Top - (int)position.Y,
        rectangle.Width,
        rectangle.Height);
}
```

Kuva 14. Camera –luokan tärkeimmät metodit ja ominaisuudet.

Ominaisuuden Position (kuva 14), joka hallinnoi kameran sijaintia, valitsin esiteltäväksi sen hyödyntämän, XNA-kirjastossa valmiina olevan, MathHelper luokan metodin Clamp takia. Kyseisen metodin avulla voidaan kätevästi tarkistaa muuttujalle position ominaisuuden kautta annettu arvo. Jos arvo on negatiivinen, se muutetaan arvoksi 0 ja jos arvo on isompi kuin ruudun leveys tai pituus vähennettynä kameran leveydellä tai pituudella, antaa se arvoksi ruudun leveyden tai pituuden vähennettynä kameran leveydellä tai pituudella. Näin mahdollistetaan se, ettei kameran kumpikaan reuna poistu pelialueelta, vaan se pysyy sille rajatulla alueella.

Metodi ObjectIsVisible (Kuva 14) pitää sisällään bounding box collision – menetelmän, vaikkei se suoranaisesti törmäystä testaakaan. Metodin tarkoituksena on tarkistaa, onko parametrinä annettu suorakulmio kameran näköpiirissä. Tätä metodia hyödynnetään myöhemmin Esimerkkipelin vihollisten aktivoimisen yhteydessä, jossa viholliset aktivoidaan vasta, kun pelaaja liikkuu tarpeeksi lähelle.

Metodin TransformWorldToScreen tarkoituksena on muuttaa peliobjektien sijainnit ruudulle piirrettävään muotoon. Kuvassa 15 on esimerkki kuvitellun pelin pelialueen ja kameran suhteista.



Kuva 15. Koordinaattien hahmotus.

Kuva 15 selittää TransformToWorld-metodin toimintaa. XNA:ssa peliruudulle piirrettäessä jokaiselle piirrettävälle objektille pitää antaa aina sijainti X/Y – koordinaatistossa. XNA:ssa X/Y koordinaatisto toimii eri tavoin kuin

tavanomaisesti matematiikassa. XNA:ssa ylhäällä vasemmalla oleva ruudun yläreuna on arvoltaan 0,0 ja X-akseli kasvaa oikealle mentäessä ja Y-akseli alaspäin mentäessä.

Peliobjektien koordinaatit säilytetään koko pelimaailman koordinaatteihin verrattuna, jolloin osa peliobjekteista on kameran ulkopuolella. Jotta tiedetään peliobjektin sijainti kameraan nähden, pitää peliobjektin sijainnista vähentää kameran sijainti. Jäljelle jäävä arvo kertoo peliobjektin sijainnin kameraan nähden. Kameran koordinaatti on aina pelimaailman koordinaattien mukaan. Kaikki piirrettävät objektit ajetaan aina tämän metodin läpi, jotta piirtäminen ruudulle onnistuu oikein. Metodia myös hyödynnetään tile engine –menetelmässä laskemaan, mitkä solut ovat ruudulla näkyvissä. Näin voidaan laskea, mitkä solut voidaan jättää piirtämättä ja saada aikaan kevyemmin toimiva tile engine –ratkaisu.

5.4 Tile engine –tekniikka

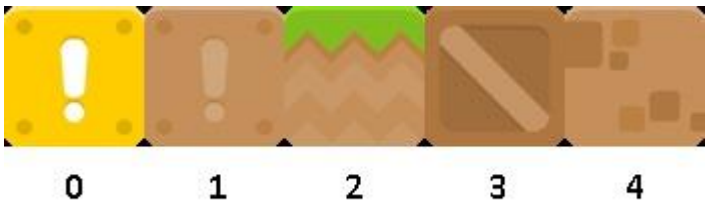
5.4.1 Tile engine –tekniikan toteutus

Tile engine –tekniikan toteutin luomalla yhden staattisen TileMap –luokan. TileMap –luokan voisi toteuttaa monella eri tapaa, mutta aion pysytellä yksinkertaisessa, yksitasoisessa mallissa. Toteutus ei tule tukemaan erillistä editoria, koska Esimerkkipeli sisältää kaksi karttaa, eikä täten ole järkevää tehdä turhaa työtä luomalla editoria. En aio esitellä vertailtavan tile engine -menetelmän editoria, mutta esittelen erot oman toteutukseni ja editoria tukevan tile engine –menetelmän väliltä.

TileMap –luokka Esimerkkipelissä pitää sisällään pelin molemmat kartat ja niiden luominen on esitelty kuvassa 16. Kuvassa 17 näkyy kartan arvoille oma graafinen vastineensa.

```
static public void InitializeTileMap(int mapID)
 {
    if (mapID == 1)
    {
        mapWidth = 30; //kartan leveys solujen lkm
        mapHeight = 10; //kartan korkeus solujen lkm
        //kartta, jossa jokaisella numerolla on oma vastikkeensa
        map = new int[10, 30] {
        { 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7 },
        { 7,7,7,7,9,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7 },
        { 7,7,7,9,7,7,7,7,7,8,7,0,7,7,7,7,9,7,7,7,7,7,7,7,7,4,7,7,7,7 },
        ...
        { 7,5,7,3,3,3,3,3,7,8,7,3,7,8,7,3,7,7,7,3,3,3,7,7,7,7,7,7,6,7 },
        { 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2 }};

    }
    else
    {
        mapWidth = 20;
        mapHeight = 10;

        map = new int[10,20] {
        { 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 },
        { 7, 7, 7, 7, 7, 7, 7, 7, 4, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 },
        { 7, 7, 7, 7, 7, 7, 9, 7, 4, 7, 7, 7, 7, 7, 7, 7, 0, 7, 0, 7 },
        ...
        { 7, 7, 5, 3, 4, 7, 7, 7, 7, 7, 7, 7, 3, 7, 7, 7, 7, 6, 7, 7 },
        { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 }};
    }

    mapId = mapID; //laitetaan kartan ID muistiin

    Camera.WorldWidth = tileWidht * mapWidth; //annetaan kameralle tiedot
    Camera.WorldHeight = tileHeight * mapHeight; //pelialueen koosta

    Camera.Position = new Vector2(0, 0); //asetetaan kamera kohdilleen
}
```

Kuva 16. TileMap –luokan kartan alustusmetodi.



Kuva 17. Kartan spritesheet.

Kuvassa 16 on TileMap–luokan InitializeTileMap-metodi, joka luo annetun parametrin perusteella pelikartan ja antaa Camera–luokalle tarvittavat tiedot pelimaailman koosta. Kuvan 16 kartta (kaksiulotteinen taulukko map) pitää sisällään tiedot solujen arvoista. Kartta määrittää solujen sisälle tiedot, mikä tile piirretään mihinkin kohtaan (arvot 0-4) (Kuva 17), pelaajan aloituskohdan (arvo 5), tavoitteen, jonne pelaajan on määrä päästä (numero 6), tyhjän solun tunnuksen (arvo 7) ja vihollisten aloituspaikat (arvot 8 ja 9). TileMap –luokalla on erilliset metodit palauttamaan vihollisten ja pelaajan aloituspaikat, ja niitä kutsutaan luokan ulkopuolelta pelaajaa ja vihollisia alustaessa. Tällä menetelmällä on helppoa lisätä, poistaa tai muuttaa vihollisten tai pelaajan sijaintia vaihtamalla vain solun arvoa.

TileMap–luokan tulee pystyä havaitsemaan, mihin soluun annettu koordinaatti sijoittuu. Ratkaisu on jakaa annettu X-koordinaatti tile-elementin leveydellä (Esimerkkipelissä kartta-tilen koko on 70x70 pikseliä) ja laskemalla mikä on jäljelle jäävä kokonaisluku. Sama suoritetaan jakamalla Y-koordinaatti tilen korkeudella, jolloin saadaan erilliset X ja Y kokonaislukuarvot. Näiden arvojen avulla voidaan saada selville solun sijainti map–taulukossa. Esimerkiksi sijainti X = 60 ja Y = 80 antavat arvot 0 ja 1, koska X-koordinaatti jaettuna tilen leveydellä 70 on 0. Vastaavasti Y-koordinaatin avulla saadaan 80 / 70 = 1. Koordinaatti (60,80) sijoittuu siis kartan soluun map[1,0]. Muut TileMap–luokan metodit pystyvät tällä tiedolla (map[1,0]) tekemään muita suorituksia, kuten hakemaan tiedon, mikä on solun arvo sekä muuttamaan sen ja testaamaan onko se pelaajan kulkua estävä.

TileMap-luokka ei sisällä Update-metodia, koska kaikki sen tiedot ovat staattisia ja muuttumattomia. Tarpeelliset muuttuvat tiedot muutetaan metodien avulla, jotka kutsutaan luokan ulkopuolelta erikseen tarvittaessa. Luokalla on kuitenkin Draw-metodi piirtämään sen osan karttaa, joka on ruudulla näkyvissä (Kuva 18).

```
static public void Draw(SpriteBatch spriteBatch)
 {
            int tileID;

            int startingIndex = GetSquareAtPixelX((int)Camera.Position.X);
            int endingIndex = GetSquareAtPixelX((int)Camera.Position.X +
            Camera.ScreenWidth);
    //tarkistus mistä kohtaa aloitetaan
    for (int y = 0; y < mapHeight; y++)
    {
        for (int x = startingIndex; x <= endingIndex; x++)
        {
            if ((x >= 0) && (x < mapWidth))
            {
                tileID = map[y, x];
                if (tileID <= 4)
                spriteBatch.Draw(spriteSheet,
                Camera.TransformWorldToScreen(GetPosition(y, x)),
                GetSourceRectangle(tileID), Color.White);
                else if (tileID == 6)
                {
              spriteBatch.Draw(whiteRectangle,
                Camera.TransformWorldToScreen(GetPosition(y, x)), Color.Red);
                }
            }
        }
    }
}
```

Kuva 18. TileMap –luokan Draw osio.

Vaikka kyseessä ei ole iso peli, on silti hyvä tapa pyrkiä hieman optimoimaan pelin ohjelmalauseita. Alussa olevat muuttujat startingIndex ja endingIndex laskevat Camera –luokan sijainnin ja leveyden perusteella sen osan karttaa, joka sijoittuu näkyvän peli-ikkunan ruutuun. Ideana on piirtää vain se osa karttaa, joka oikeasti tulee peliruudulle näkyviin ja jättää peli-ikkunan ulkopuolelle jäävä osa piirtämättä. Draw osio laskee alku- ja loppupisteet vain X-akselille. Y-akselia ei tarvita laskemaan, koska molempien Esimerkkipelin karttojen korkeus on 10 solua ja peli-ikkunan korkeus on muuttumaton 700 pikseliä. Yhden tilen korkeus oli 70 pikseliä, joten 10 solua kertaa 70 pikseliä on sama kuin peliruudun korkeus. Kamera ei siis liiku Esimerkkipelissä ylös eikä alas.

5.4.2 Toteutusten vertailu

Otan oman tile engine –toteutukseni rinnalle vertailun kohteeksi monitasoisen tile engine –toteutuksen (Jaegers 2013b). Perusidea on kirjan toteutuksessa sama, eli kartta on ohjelmakoodissa edelleen kaksiulotteinen taulukko. Suurin ero on taulukon solujen sisällössä. Omassa tile engine –tekniikassani käytin solun sisältöinä arvoja 0-9, joilla jokaisella on oma merkityksensä. Kirjan mallissa taas solujen sisältö koostuu luokan MapSquare olioista. Kyseinen luokka on suunniteltu niin, että se pitää sisällään yksiulotteisen taulukon, jossa kirjan esimerkin mukaan on kolme solua: tausta-, interaktiivinen- ja etuala–solut. Tämän lisäksi luokka MapSquare, joka siis vastaa yhtä solua kartalla, voi sisältää tekstityyppisen arvon. Tällä tekstityyppisellä arvolla voidaan antaa soluille tietty merkitys, esimerkiksi toteuttaa niin, että jos solun arvon merkitys on tietynlainen, ei siitä voi tietokoneen ohjaama vihollinen kulkea läpi. Tämän menetelmän avulla voidaan luoda vihollisille tietyt rajatut alueet, joilla ne voivat liikkua. Omassa pelissäni toteutin vihollisen liikkumisen kartalla eri tavalla, josta kerron seuraavassa alaluvussa enemmän.

Palataan MapSquare –luokan yksiulotteiseen taulukkoon. Aiemmin esittelin luvussa 2 kuvan 8 avulla moniulotteisen tile engine –menetelmän toteutusta ja se on juuri samanlainen, mitä Jaegersin toteutuksessa on tehty. MapSquare –luokan yksiulotteisen taulukon solut pitävät sisällään yhden numeroarvon, ja vastaavasti tietty arvo merkitsee tiettyä asiaa. Suurena erona tässä menetelmässä verrattuna omaani on se, että jokaisella taulukon solulla on omanlaisensa päivitystoimenpide. Taulukon taustasolun tiedot ovat kaikki tausta-tile-elementtejä, niihin ei voi törmätä, eikä niillä ole erillistä toimintaa, ne vain luovat pelille taustan. Interaktiivinen osio on se, mitä koko oma toteutukseni on. Se pitää sisällään kartta-tilet ja niiden sijainnin. Tämän lisäksi se myös sisältää interaktiiviset objektit, kuten esimerkiksi oven tai laatikon, jolla on jokin toiminnallinen merkitys. Taulukon viimeinen solu, etualasolu, on toiminnaltaan samanlainen kuin taustasolu. Ainoa erotus siinä

on se, että taustasolu piirretään alimmaiseksi ja etualasolu etummaiseksi. Interaktiivinen solu jää näiden kahden väliin.

Jaegersin toteutus on jokaisella osa-alueella paremmin toteutettu kuin itse toteuttamani ratkaisu. Siinä on kuitenkin yksi asia, mikä oli ratkaisevaa kun mietin miten toteutan tile engine –menetelmän Esimerkkipelissä. Tämä kriittinen syy on Jaegersin mallin monimutkainen luonne ja teoriassa mahdottomuus luoda pientäkään karttaa ilman erillistä karttaeditoria. Kartta olisi mahdollista luoda ilman editoria, mutta toteutustapa ilman editoria vaatisi sen, että jokaisen solun olion joutuisi käsin luomaan erikseen, ja jokaiselle oliolle antamaan metodilla tietyt arvot. Omassa Esimerkkipelin toteutuksessa solut koostuvat pelkistä numeroista, eivätkä olioista, joten työmäärä on käsin karttaa luotaessa paljon pienempi. Esimerkkipelissä on myös suunniteltu olevan vain kaksi karttaa, joten kahta karttaa varten luotava kokonainen editori ei olisi ollut tarkoituksenmukaista. Jaegersin toteutuksessa on erona myös se, että siinä toteutettu editori tallentaa kartan tiedostoksi ja peli lataa sen tarvittaessa tiedostosta. Tämän ansiosta karttoja ei tarvitse käsin kirjoittaa karttoja käsittelevän luokan sisälle, vaan ne ovat siististi erossa pelin lähdekoodista.

5.5 Piirtäminen ja animointi

5.5.1 Toteutus

Tile engine –menetelmä pitää sisällään oman piirtämistekniikkansa, mutta se koskee vain tile engine –menetelmää koskevia asioita, kuten solujen sisältöjä. Kaikille muille peliobjekteille kuten pelaajalle, vihollisille ja ammuksille pitää luoda oma yhteinen toteutuksensa.

Esimerkkipelissä peliobjekteilla on hyvin vähän animaatiota. Pelaajalla ja vihollisilla on vain liikkumisanimaatiot ja panoksilla ei ole animaatiota lainkaan. Pelaajan pelihahmolla on myös yksi animoimaton sprite, jossa pelihahmo on liikkumaton ja katsoo kohti pelaajaa. Esimerkkipelissä

viholliset ovat koko ajan liikkeessä kun ne ovat ruudulla. Tiivistettynä siis tarvitaan toteutustapa jolla saadaan viholliset toistamaan liikkumisanimaatiota jatkuvasti ja toteutustapa, jolla pelaaja liikkuu tai on paikallaan ja joka myös tukee animoimattoman panoksen piirtämistä. Ratkaisu tähän on Sprite-luokka.

Sprite-luokka on itse luomani luokka, joka pitää sisällään paljon tietoja. Päätin säilöä kaikki peliobjekteja koskevat tiedot Sprite-luokan sisälle, koska luokka tarvitsee monet tiedot mm. Draw-metodissa. Sprite-luokan sisältämät tärkeimmät animaatioon ja piirtämiseen liittyvät muuttujat on esitelty kuvassa 19.

```
private Texture2D texture;  //kuva
private Vector2 worldPosition;  //sijainti pelimaailmassa
private bool animate; //animoidaanko kuvaa
private float scale = 1.0f; //skaalaus
private float rotation = 0.0f; //kiertokulma
private Color tint = Color.White; //värjäys
private float frameTime; //kuinka kauan yksi frame on ruudulla
private float elapsedFrameTime; //kauanko frame on ollut ruudulla
private int currentFrame; //nykyinen frame
private int frames; //framejen määrä yhteensä
```

Kuva 19. Sprite-luokan muuttujat.

Kuvassa 19 on esillä kaikki tärkeimmät animaatioon ja kuvan ruudulle piirtämiseen liittyvät muuttujat. Sprite-luokka pitää sisällään kuvatiedoston, sijainnin, kuvan skaalauksen, kiertämisen ja värityksen sekä tiedot animaatiota varten.

Sprite-luokalla on myös oma Update-metodi. Update-metodi laskee nykyisen framen aikaa ruudulla ja tarvittaessa piirtää seuraavan framen. Kuvassa 20 näkyy Update-metodin kohta, jossa animaatiota suoritetaan.

```
//jos objekti on animoitu
    if (animate)
    {
        //lisätään nykyisen framen ruudullaoloaikaa
            elapsedFrameTime +=
            (float)gameTime.ElapsedGameTime.TotalSeconds;
        //jos frame on ollut tarpeeksi kauan ruudulla
        if (elapsedFrameTime >= frameTime)
        {
            //siirrytään seuraavaan frameen
            currentFrame++;
            //tarkistetaan, onko viimeinen frame menossa
            //ja jos on, aloitetaan animaatio alusta
            if (currentFrame == frames - 1)
            {
                currentFrame = 0;
            }
            //nollataan framen ruudullaoloaikaa laskeva muuttuja
            elapsedFrameTime = 0.0f;
        }
    }
    else
    {
            //jos objekti ei ole animoitu (panos, pelaajan paikallaan oleminen)
        //niin poistetaan kuvan efektit (ei käännettyä kuvaa) ja
        //muutetaan
        //nykyinen frame oikeaksi (panos = 0, pelaaja = viimeinen frame)
        spriteEffects = SpriteEffects.None;
        currentFrame = frames - 1;
    }
```

Kuva 20. Sprite-luokan animaation päivitys.

Animaation päivitys toimii kuvan 20 mukaisesti niin, että ensin tarkistetaan, onko objekti animoitu (pelaajan tai vihollisen liikkumisanimaatio) vai ei (panos tai pelaajan seisomiskuva). Jos objekti on animoitu, lisätään muuttujaan elapsedFrameTime tämän ja edellisen päivityksen välinen aikaero. Kun muuttujan arvo on suurempi kuin frameTime-muuttujan, joka määrittää kauanko yksi frame on ruudulla, päivittää ohjelma uuden framen eli kasvattaa muuttujan currentFrame arvoa ja tarkistaa, ettei edellinen frame ollut viimeinen ja nollaa elapsedFrameTime muuttujan. Näin saataisiin esimerkiksi aiemmin luvussa 2.3 kuvassa 3 esitetty tulipallo animoitua jatkuvasti. Toteutustapa toimii, kun jokaisen spritesheetin kuvat on laitettu jonoksi, jolloin kuvan korkeus ei muutu missään kohtaan vaan

ainoastaan framen alkamiskohta muuttuu. Alkamiskohta lasketaan muuttujan currentFrame ja yksittäisen framen leveyden mukaan, joka on kaikissa Esimerkkipelin frameissa sama.

Kuvassa 21 näkyy ominaisuus GetSourceRectangle, joka laskee ja palauttaa suorakulmion, joka rajaa spritesheetistä halutun piirrettävän alueen, yhden framen.

```
private Rectangle GetSourceRectangle
{
    get { return new Rectangle(Width * currentFrame, 0, Width, Height); }
}
```

Kuva 21. Halutun framen laskeminen

Kuvan 21 toiminta on hyvä hahmottaa tulipallokuvan avulla. Tulipallokuvan koko on 567x94 pikseliä, mutta esimerkkiä helpottaaksemme voidaan kuvitella että kuva 3 koko muutettaisiin 600x100 pikselin kokoiseksi. Tulipallolla on yhteensä kuusi framea ja jokainen on keskenään samankokoinen ja ne ovat kaikki samalla tasolla yhdessä rivissä. Tästä saadaan tietää se, että yhden framen leveys on 600 / 6 eli 100 pikseliä, ja koska kaikki framet ovat rivissä, on yhden framen korkeus 100 pikseliä. On hyvä muistaa, että XNA:ssa on omanlaisensa koordinaattijärjestelmä eli origo on ylhäällä vasemmalla, X-akseli kasvaa oikealle ja Y-akseli kasvaa alaspäin ja että sprite irroitetaan spritesheetistä aina suorakulmion muodossa. Ensimmäinen frame on numeroltaan 0, seuraava 1 jne. Sprite-luokassa muuttuja currentFrame sisälsi tietoa, kuinka mones frame on menossa ja sitä hyödynnetään kuvan 21 tapauksessa. Ensimmäinen frame alkaa koordinaatista 0,0 ja on leveydeltään ja korkeudeltaan 100 pikseliä. Toinen frame alkaa koordinaatista 100,0 ja on leveydeltään ja korkeudeltaan edelleen 100 pikseliä. Koordinaatti, josta frame alkaa, lasketaan kertomalla framen leveys muuttujan currentFrame arvolla.

Sprite-luokalla on myös oma Draw-metodi, joka ottaa vastaan parametrina SpriteBatch-olion. SpriteBatch-luokka on XNA-kirjaston oma luokka ja se hoitaa kaiken piirtämisen ruudulle. SpriteBatch toimii niin, että kun se avataan Draw-metodissa Begin-metodilla, siihen voidaan sisällyttää monta eri piirtokäskyä. Kun kaikki halutut piirtokäskyt on siihen sisällytetty, se suljetaan End-metodilla. End-metodi ohjaa SpriteBatch-olion luovuttamaan sisältönsä näytönohjaimelle, joka piirtää kaiken kerralla ruudulle. SpriteBatch siis kerää kaikki piirtokäskyt itsellensä ja lähettää ne lopuksi näytönohjaimelle.

5.5.2 Toteutusten vertailu

Tekemäni Sprite-luokka toimii hyvin Esimerkkipelin puitteissa. Ongelmana siinä on se, että yhteen luokkaan on pyritty sisällyttämään liian paljon asioita. Koska Esimerkkipelissä käytetyt kuvat ovat kaikki yhden rivin spritesheetejä, ei Esimerkkipeliin tarvinnut kuvien 20 ja 21 mallia monimutkaisempaa toteutustapaa luoda. Tämä keino ei tällaisenaan kuitenkaan toimi hyvin, kun peliobjekti sisältää monta erilaista animaatiota ja kun halutaan luoda enemmän automaattista toimintaa.

Tähän on ratkaisuna eritellä kuvan piirtämis- ja animaatiotoiminnot eri luokkiin (Jaegers 2013a). Ideana on tehdä erillinen luokka animaatiolle, joka pitää sisällään yhden tietyn animaation kaikki framet. Tämä luokka pitäisi sisällään samat tiedot, joita omassa Sprite-luokassanikin oli animaatioita varten, eli nykyisen framen, framen ruudullaoloajan ja muut animaatioon liittyvät tiedot. Tämän lisäksi tässä mallissa on luokkaan laitettu tekstimuuttuja, joka pitää sisällään seuraavan animaation nimen.

Tällä saavutetaan se, että jos esimerkiksi yhdessä spritesheetissä on monta animaatiota pelaajalle, kuten liikkuminen, paikallaan oleminen, hyppääminen, ryömiminen, ampuminen ja kuoleminen, niin nämä animaatiot voidaan eritellä ja luoda jokaiselle oma olionsa. Jokainen näistä olioista tietää vain oman animaationsa. Tekstimuuttuja on sitä varten, että jos

tiedetään jonkin animaation tulevan aina tietyn animaation jälkeen, voidaan se määrittää tapahtumaan automaattisesti. Jos pelissä vaikka olisi sääntö, että aina kun pelaaja on ampunut kerran, pelaajan täytyy seistä paikallaan, voidaan ampumisanimaation tekstimuuttujaan antaa arvosti paikallaan seisomisanimaatio-olion nimi, jolloin aina ampumisen jälkeen pelaajan hahmo aloittaa tämän animaation.

Näitä animaatio-olioita päivittää ja hallitsee toinen ylempi luokka, joka pitää sisällään eri peliobjektin kaikki eri animaatio-oliot. Tämä hallitsijaluokka sisältää myös itse kuvatiedoston, josta animaatio-oliot ottavat omat animaationsa ja sisältää myös objektin sijainnin.

Tämä malli sisältää myös kolmannen luokan, joka pitää sisällään edellä esiteltyä animaatioita hallitsevaa luokkaa. Tämä luokka pitää sisällään tiedot törmäystestejä varten sekä se hallinnoi edellistä esiteltyä luokkaa.

Oma Sprite-luokkani on perusidealtaan samanlainen kuin äsken esitelty malliratkaisu. Animaation päivityksen perusidea on täsmälleen samanlainen. Edellinen ratkaisu noudattaa hyvin olio-ohjelmoinnin tyylejä pilkkomalla eri asiat pieniin osasiin ja kokoamalla niistä kokonaisen ratkaisun. Se taipuu moneen eri tilanteeseen. Oma ratkaisuni on siinä mielessä huonompi, ettei se juuri sovellu suuremman ja erilaisen pelin toteutukseen sellaisenaan vaan se vaatisi paljon muutoksia. Edellinen malli taas on idealtaan sopeutuvaisempi moniin erilaisiin peleihin eikä sitä juuri tarvitsisi muokata.

5.6 Törmäystestit

Esimerkkipeli sisältää erilaisten objektien välisiä törmäyksiä. Näitä törmäyksiä ovat pelaajan, panosten ja vihollisten törmäämiset ympäröivään pelimaailmaan ja pelaajan ja vihollisten ja panosten keskinäiset törmäykset. Jokainen törmäystesti toteutetaan bounding box collision –menetelmää hyödyntäen. XNA:ssa bounding box collision –menetelmä on toteutettu kuvassa 22 näkyvän metodin Intersects avulla.

```
public bool BoundingBoxCollision(Rectangle collisionRectangle)
{
return bulletCharacter.CollisionRectangle.Intersects(collisionRectangle);
}
```

Kuva 22. Bounding box collision XNA:ssa.

Kuvan 22 bound box collision -törmäystesti on pelin panosta esittävän luokan, Bullet-luokan, sisällä oleva törmäystesti. Se ottaa parametrinä vihollista ympäröivän suorakulmaisen kulmion tiedot ja vertaa niitä metodin Intersects avulla.

Intersects-metodi on XNA-kirjastossa olevan Rectangle-luokan metodi. Se laskee annetun suorakulmion kulmien koordinaatit, vertaa niiden koordinaatteja omiinsa ja tutkii, onko joku koordinaateista sen itsensä sisällä. Jos on, se palauttaa arvon tosi.

Se ei kuitenkaan yksinään riitä. Kuvassa 23 nähdään lepakon ja pelaajan olevan selvästi toisista erillään, mutta bounding box collision –menetelmä väittää siinä sattuneen peliobjektien törmäyksen.

Kuva 23. Kahden objektin törmäys bounding box collision –menetelmällä.

Vaikka bounding box collision –menetelmä näyttää väärin, on sillä silti merkittävä osa. Koska bounding box collision –menetelmä on kevyt, käytetään sitä testaamaan milloin kaksi objektia ovat hyvin lähellä toisiaan.

Kuvassa 23 näkyy, että kaksi objektia on todellakin hyvin lähellä toisiaan, ja törmäystesti väittää törmäyksen sattuneen. Tästä on hyvä jatkaa raskaammalla ja tarkemmalla per pixel collision –menetelmällä.

Esimerkkipelissä toteutettu per pixel collision –menetelmä on mahdollimman yksinkertainen. Esimerkkipeli on suunniteltu niin, ettei kuvia ole kierretty, eikä skaalattu, joten per pixel collision –törmäystesti voidaan suorittaa ilman matriiseja. Oma toteutukseni laskee törmäyksen kahden eri kuvan pikseleiden ja niiden läpinäkyvyyden mukaan.

Jotta kahden objektin spriten pikseleitä voidaan verrata, ne täytyy ensin laskea ja hakea oikeasta kohtaa spritesheetiä. Kuvan 24 mukainen getColors-metodi luo taulukon, joka on kooltaan niin suuri, että siihen mahtuu kaikki nykyisen framen pikselit. GetColors-metodi hyväksikäyttää GetSourceRectangle-ominaisuutta, joka palauttaa nykyisen framen spritesheetistä. Tällä tavoin getColors-metodi osaa laskea oikeat pikselit muuttujaan c. GetColors-metodi sisältyy Sprite-luokkaan.

```
public Color[] getColors()
 {
    Color[] c = new Color[Width*Height];
    texture.GetData(0, GetSourceRectangle, c, 0, c.Length);
    return c;
 }
```

Kuva 24. Väritaulukon laskeminen.

Kuvan 24 metodia hyödynnetään per pixel collision –törmäystestissä. Kuvassa 25 nähdään Esimerkkipelissä toteutettu per pixel collision –menetelmä, joka on sijoitettuna GameScreen-luokan sisälle.

```
//värit a1 ja b1, sijainnit a2 ja b2, leveydet a3 ja b3, korkeudet a4 ja b4
private Boolean PixelCollision(Color[] a1, Color[] b1, Vector2 a2,
Vector2 b2, int a3, int b3, int a4, int b4)
{
    //lasketaan alue, jossa törmäys on
    int x1 = Math.Max((int)a2.X, (int)b2.X);
    int x2 = Math.Min((int)a2.X + a3, (int)b2.X + b3);
    int y1 = Math.Max((int)a2.Y, (int)b2.Y);
    int y2 = Math.Min((int)a2.Y + a4, (int)b2.Y + b4);

    for (int y = y1; y < y2; ++y)
    {
        for (int x = x1; x < x2; ++x)
        {
            //käydään pikseli kerrallaan läpi
            int i = (x - (int)a2.X) + (y - (int)a2.Y) * a4;
            Color colora = a1[i];
            Color colorb = b1[(x - (int)b2.X) + (y - (int)b2.Y) * b4];
                //jos arvo A on eri kuin 0, tapahtuu törmäys. jos A = 0, on pikseli
                läpinäkyvä
            if (colora.A != 0 && colorb.A != 0)
            {
                return true; //törmäys
            }
        }
    }
    return false; //ei törmäystä
}
```

Kuva 25. Per pixel collision –törmäystesti.

Kuvan 25 törmäystesti kutsutaan sen jälkeen, kun sitä ennen suoritettu bounding box collision –törmäystesti on todennut törmäyksen tapahtuneen eli kun peli on kuvan 23 mukaisessa tilanteessa. Kuvan 25 PixelCollision metodi ottaa parametreinä kahden törmäävän objektin tiedot: väritaulukot, jotka luodaan kuvan 24 metodissa, sijainnin, leveyden ja korkeuden. Metodi laskee alueen, jossa törmäys on tapahtunut ja käy sen jälkeen jokaisen pikselin kerrallaan läpi verraten niiden A arvoa. A arvo sisältää tiedot pikselin läpinäkyvyydestä, jossa 0 on täysin näkyvä ja jokainen muu arvo enemmän tai vähemmän läpinäkyvä. Esimerkkipelin spriteissä ei ole muita kuin joko täysin näkyviä tai täysin läpinäkyviä pikseleitä, joten metodin ei tarvitse tietää osittain läpinäkyvistä pikseleistä mitään.

Kuvassa 26 nähdään pelitilanne, jossa peli on edennyt vähän matkaa kuvan 23 tapahtumasta.

Kuva 26. Kahden objektin törmäys per pixel collision –menetelmällä.

Kuvassa 26 nähdään, että lepakon siipi ja pelaajan pää osuvat yhteen eli niiden näkyvät pikselit ovat päällekkäin. Kuvien 23 ja 26 tilanteissa on selvä ero bounding box collision – ja per pixel collision –menetelmien lopputuloksissa. Ilman bounding box collision –menetelmää ei kuitenkaan voisi käyttää per pixel collision –menetelmää, koska ohjelma muuttuisi liian raskaaksi, vaikka objektien, joiden törmääminen tulisi tarkastaa, lukumäärä onkin varsin pieni.

Oman per pixel collision –menetelmän toteutus voisi olla tehokkaampi. Sitä ei ole optimoitu lainkaan. Sen sijaan, että käytäisiin läpi jokainen pikseli, voitaisiin esimerkiksi käydä läpi vain joka toinen, kolmas tai neljäs pikseli. Tehokkuutta karsii myös se, että omassa toteutuksena väritaulukot luodaan aina uudestaan. Ne olisi voinut esimerkiksi laskea spritesheetin latausvaiheessa ja säilöä erillisiin muuttujiin, joista ne voisi tarvittaessa hakea.

5.7 Pelaaja ja viholliset

Esimerkkipelissä pelaajan toteutus on aika riippuvainen Sprite-luokan toiminnoista. Sprite-luokan sisällän on piirtämisen ja törmäystestin lisäksi myös hahmon liikkumista ja olotilaa säätelevät metodit. Olotilalla tarkoitan

sitä, onko pelaajan hahmo maassa kävelemässä vai onko hahmo juuri hypännyt ja on parhaillaan ilmassa. Pelaaja on toteutettu yksinkertaisella Player-luokalla.

Player-luokan sisällä on kuvan 27 mukainen Update-metodi, joka tunnistaa pelaajan näppäimistösyötteet ja käskyttää Sprite-luokan oliota niiden perusteella.

```
public void Update(GameTime gameTime)
 {   //jos pelaaja on elossa
    if (alive)
    {  //lisätään edellisestä ruudunpäivityksestä kulunut aika muuttujaan
       float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
            //tarkistetaan käyttäjän syötteet
       if (InputHandler.ButtonPressed(Keys.W))
       {
          character.Jump(); //kokeile hypätä
       }
       if (InputHandler.ButtonPressed(Keys.D))
       {
          character.Effects = SpriteEffects.FlipHorizontally;
          character.Animate = true; //käännä kuva ja animaoi
          character.Movement = new Vector2(90 * elapsed, 0); //liike
       }
       else if (InputHandler.ButtonPressed(Keys.A))
       {
          character.Effects = SpriteEffects.None;
          character.Animate = true;
          character.Movement = new Vector2(-90 * elapsed, 0);
       }
       else
       {
          character.Animate = false; //liikkumaton
       }
            //character on Sprite-luokan olio ja se päivitetään
       character.Update(gameTime);
            //päivitetään kamera
       if (Camera.CameraScreen.Right - character.WorldPosition.X < 200)
       {
          Camera.Position += new Vector2(90 * elapsed, 0);
       }
                        else if (character.WorldPosition.X –
            Camera.CameraScreen.Left < 200)
       {
          Camera.Position += new Vector2(-90 * elapsed, 0);
       }
    }
 }
```

Kuva 27. Player-luokan Update-metodi.

Kuvan 27 toiminnot ovat aika yksiselitteiset. Ensin tarkistetaan näppäinsyötteet ja käskytetään Sprite-luokan oliota character halutuilla tavoilla, jonka jälkeen päivitetään Sprite-olio. Tämän jälkeen päivitetään Camera-luokka vertaamalla pelaajan sijaintia peli-ikkunan reunoihin nähden. Jos pelaaja on liian lähellä reunaa, siirtyy kamera pelaajan suuntaan. Näin pelaaja ei voi koskaan poistua peli-ikkunasta sivuttaissuunnassa.

Player-luokalla on kuvan 28 mukainen Draw-metodi. Sen ainoa tarkoitus on antaa parametrinä saanu SpriteBatch-luokan olio Player-luokan Sprite-luokan olion Draw-metodille.

```
public void Draw(SpriteBatch spriteBatch)
 {
    character.Draw(spriteBatch);
 }
```

Kuva 28. Player-luokan Draw-metodi.

Esimerkkipelin viholliset on luotu hieman erilaisella tavalla. Esimerkkipelissä on tarkoitus olla kaksi erilaista vihollista erilaisilla liikeradoilla, joten loin peliin lentävän ja kävelevän vihollistyypin. Lentävä vihollinen lentää aloituspisteestään tietyn matkan vasemmalle. Se kääntyy takaisin kohti aloituspistettään, kun se joko törmää johonkin tai se on lentänyt tietyn pitkän matkan. Lentävä vihollinen lentää edestakaisin ja lentämällä siihen ei painovoima vaikuta, eli se pysyy aina samalla korkeudella. Kävelevä vihollinen taas on painovoimalle altis ja se jatkaa matkaa vasemmalle kunnes se törmää seinään ja vaihtaa suuntaa. Molemmat vihollistyypit ovat passiivisia, kunnes pelaaja on tarpeeksi lähellä, jolloin ne aktivoituvat. Kerran aktivoitunut vihollinen ei enää passivoidu.

Loin peliin rajapinnan Enemy, koska molemmat vihollistyypit sisältävät samannimisiä, mutta toiminnoiltaan erilaisia metodeja ja koska viholliset on kerätty GameScreen-luokassa yhteen taulukkomuuttujaan. Olioita ei saa liitettyä taulukkoon, jos ne ovat eri luokan olioita, mutta koska molemmat vihollistyypit, FlyingEnemy- ja WalkingEnemy-luokat, periytyvät yhteisestä Enemy-rajapinnasta, ne voidaan sisällyttää samaan taulukkoon Enemy-tyyppisinä olioinstansseina.

Molempien vihollisten Update-metodit ovat keskenään hyvin samanlaisia. Kuvassa 29 näkyy lentävän vihollisen Update-metodi.

```
public void Update(GameTime gameTime, Vectoc2 playerPosition)
 {
    float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
    if (playerPosition.X + 500 - character.WorldPosition.X >= 0 ||
        character.CollisionRectangle.Intersects(Camera.CameraScreen))
       {
              active = true; character.Animate = true;
       }
    if (alive && active)
    {
        character.Movement = new Vector2(xMovement * elapsed, 0);

        character.Update(gameTime);

           if (character.XMovementZero || character.WorldPosition.X <
           turnpoint.X || character.WorldPosition.X > spawnpoint.X)
        {
            xMovement *= -1;
            if (character.Effects == SpriteEffects.FlipHorizontally)
                character.Effects = SpriteEffects.None;
            else
                character.Effects = SpriteEffects.FlipHorizontally;
        }

    }
 }
```

Kuva 29. FlyingEnemy-luokan Update-metodi.

Kävelevän vihollisen Update-metodi on samanlainen, ainoana erona se, että se ei Sprite-luokan olion päivityksen jälkeen tarkista kuin sen, onko

xMovementZero arvoltaan tosi. xMovementZero arvo kertoo, onko vihollinen törmännyt seinään ja muuttanut liikkumisnopeuden nollaksi, jolloin tiedetään muuttaa aiempi nopeus päinvastaiseksi ja näin vaihtaa kulkusuuntaa. Update-metodin alussa oleva valintalause vertaa parametrinä annetun pelaajan sijaintia vihollisen omaan sijaintiin ja liian lähellä oleva pelaaja muuttaa vihollisen aktiiviseksi. Update-metodi noudattaa samaa kaavaa kuin Player-luokan Update-metodi, sillä erolla etteivät viholliset tarkista pelaajan syötteitä eikä vihollisten liike vaikuta Camera-luokkaan. Vihollisten Draw-metodi on samanlainen kuin pelaajan Draw-metodi.

5.8 Muut toiminnot

Esimerkkipelissä sekä äänet että näppäimistönsyötettä tarkkailevat luokat on toteutettu staattisina. Ääniä täytyy pystyä toistamaan joka paikasta ja samoin syötteitä tarkkaillaan monessa paikassa, joten olioiden luominen näistä luokista monessa paikkaa olisi työlästä. Pelin äänet ladataan kerralla pelin pääluokan Game1.cs loadContent-metodissa ja saman pääluokan Update-metodi vastaa näppäimistön syötteitä tarkkailevan luokan tietojen päivityksestä. Kummallakaan luokalla ei ole omaa Update- tai Draw-metodia.

6 YHTEENVETO

Tapoja toteuttaa peli on paljon, eikä pelien kehittämiseen ole olemassa yhtä ja oikeaa tapaa. Tässä opinnäytetyössä tutustuin olio-ohjelmointi mallin mukaiseen toteutukseen ja erilaisiin keinoihin toteuttaa pelin osa-alueet olio-ohjelmoinnin mukaisesti. Kävin läpi monenlaiset osa-alueet peliohjelmoinnista kuten ruudunhallinnan, kuvan piirtämisen ja animoinnin, tile engine –menetelmän ja erilaiset tavat testata törmäyksiä. Jokaisen edellä mainituista voi toteuttaa erilaisilla tavoilla ja se, mikä ratkaisee, on käyttötarkoitus.

Käyttötarkoitus nousikin muutamissa kohdissa esille, varsinkin törmäystestejä verrattaessa. Jokaisen testin tarkoituksena on testata samaa asiaa, eli objektien keskinäistä törmäämistä. Kuitenkin jokainen testi toteuttaa testauksen eri tavalla ja täten jokaiselle testille on oma aikansa ja paikkansa, jolloin niitä tulee käyttää.

Yleisemmin ajatellen myös ruudunhallinassa, tile engine –menetelmässä ja muissa tässä opinnäytetyössä esitellyissä kohdissa on kyse erilaisista käyttötarkoituksista sekä siitä, miten paljon olio-ohjelmointia sovelletaan. Hyvin pitkälle suunniteltu ja toteutettu asia, esimerkiksi tile engine –menetelmä, pystyy sulautumaan ja toimimaan sellaisenaan ilman muutoksia monessakin eri projektissa.

Esimerkkipelissä käytetty tile engine –menetelmän toteuttamistapa oli parempi Esimerkkipelin kannalta, kuin pidemmälle olio-ohjelmoinnissa viety vertausmalli, sillä kaikki sen toiminnot oli suunniteltu juuri Esimerkkipeliä varten. Esimerkkipeli toimi tämän opinnäytetyön havainnollistavana esimerkkinä pelistä, sen toteutuksesta ja osa-alueiden vertailemisesta. Esimerkkipelin, ja sen osien kuten tile engine –menetelmän ja kuvan animoinnin käyttötarkoitus oli tarkkaan rajattu tähän opinnäytetyöhön.

Olio-ohjelmoinnin mukaisesti pidemmälle viety ja pienempiin osiin pilkotun toteutuksella on tosin omat hyvät puolensa. Jako pienempiin kokonaisuuksiin parantaa yksittäisten toimintojen hahmottamista ja niiden

muuttaminen yleensä heijastuu varsin vähäisesti muihin pelin osien toimintoihin.

Suuri merkitys on myös uudelleenkäytettävyydellä. Esimerkiksi aiemmin esitellyt kuvan piirtämiseen ja animoimiseen liittyvät luokat ovat tästä hyvä esimerkki. Esimerkkipelissä käytetty toteutustapa ei ole kovin uudelleenkäytettävä, toisin kuin sille vertailukohdaksi esitelty malli, joka oli pilkottu pienempiin ja useampiin osiin kuin oma toteutukseni. Oma toteutukseni oli toteutettu niin, että se pystyi käsittelemään tietynmallisia kuvia toisin kuin vertailussa esitelty malli pystyi käsittelemään monenlaisia kuvia.

Pyrin käsittelemään peliohjelmointia koskevia asioita ja vertailemaan niiden erilaisia toteutustapoja. Ongelmia oli itse pelin ohjelmoimisessa, sillä peliä ei oltu alussa suunniteltu ihan niin tarkasti kuin olisi ehkä pitänyt. Tämä johti joissain tapauksissa tiettyjen ominaisuuksien uudelleensuunnittelemiseen, kun uusi ominaisuus ei taipunutkaan jo aiemmin toteutettujen asioiden malliin. Lopputulos oli kuitenkin täysin toimiva peli.

Opinnäytetyö toimi itselleni hyvänä oppimisprojektina. Opin lisää yleisesti peliohjelmoinnista ja myös XNA-pelikirjastosta ja ohjelmoinnista noin yleisesti. Mainitsemieni suunnitteluvirheiden ansiosta opin myös paremmin pelien ja ohjelmien olio-ohjelmointia, niin suunnittelemista kuin toteuttamistakin ajatellen.

LÄHTEET

2d-sprite animation tutorial. Darian Brown. Viitattu 01.04.2013.
http://www.darian-brown.com/2d-sprite-animation-tutorial/

Bounding box collision. Allegro wiki. Viitattu 08.05.2013.
http://wiki.allegro.cc/index.php?title=Bounding_Box

Circle collision. Computer games programming. Viitattu 08.05.2013.
http://cgp.wikidot.com/circle-to-circle-collision-detection

Coll detection overview. Riemers XNA tutorials. Viitattu 08.05.2013.
http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series2D/Coll_Detection_Overview.php

Game engine design the game loop. Entropy Interactive. Viitattu 01.04.2013.
http://entropyinteractive.com/2011/02/game-engine-design-the-game

Game programming. Wikipedia. Viitattu 01.04.2013.
http://en.wikipedia.org/wiki/Game_programming

How to learn XNA. Active tuts+. Viitattu 01.04.2013.
http://gamedev.tutsplus.com/articles/how-to-learn/how-to-learn-xna/

Jaegers, K. 2013a Expanded Sprite Engine Tutorial. Viitattu 4.1.2013.
http://www.xnaresources.com/default.asp?page=Tutorial:SpriteEngine:1

Jaegers, K. 2013b New tile engine tutorial series. Viitattu 4.1.2013.
http://www.xnaresources.com/default.asp?page=Tutorial:TileEngineSeries:1

Jaegers, K. 2010. XNA 4.0 Game Development by Example Beginner's Guide. 1. Birmingham UK: Packt Publishing Ltd.

Microsoft XNA. Wikipedia. Viitattu 01.04.2013.
http://en.wikipedia.org/wiki/Microsoft_XNA

Monogame. Viitattu 10.02.2013. http://www.monogame.net

State based games. Coke and code. Viitattu 10.02.2013.
http://slick.cokeandcode.com/wiki/doku.php?id=state_based_games

Super Giant Games. Super Giant Games. Viitattu 01.04.2013.
http://supergiantgames.com/?p=1286

Tile engine. Wikipedia. Viitattu 01.04.2013.
http://en.wikipedia.org/wiki/Tile_engine

Understanding the game loop - basix. Active tuts+. Viitattu 01.04.2013.
http://active.tutsplus.com/tutorials/games/understanding-the-game-loop-basix/

Video game genres. Wikipedia. Viitattu 10.02.2013.
http://en.wikipedia.org/wiki/Video_game_genres

What is a sprite sheet. Code and web. Viitattu 10.02.2013.
http://www.codeandweb.com/what-is-a-sprite-sheet/