Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Mika Laurila – Tommi Laurila

Opinnäytetyö

Peliohjelmointi Javalla
Case: Space Shootah

Työn ohjaaja FL Paula Hietala
Tampere 4/2009

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma

Tekijät	Mika Laurila ja Tommi Laurila
Työn nimi	Peliohjelmointi Javalla Case: Space Shootah
Sivumäärä	48
Valmistumisaika	Huhtikuu 2009
Työn ohjaaja	Paula Hietala

TIIVISTELMÄ

Tässä opinnäytetyössä ei ollut toimeksiantajaa, sillä työ sai alkunsa haaveesta perustaa oma yritys. Opinnäytetyönä tehtiin Java-peli ja tämä dokumentti, jonka tavoite on kattaa perusteet Java-pelien tekemiseen. Tämä dokumentti on tarkoitettu kaikille niille, jotka ovat kiinnostuneet peliohjelmoinnista Javalla. Työ rajattiin niin, että se kattaa perusasiat pelipohjan luomiseen.

Työssä lähdettiin selvittämään pelipohjan rakennetta siten, että se jaettiin viiteen eri osa-alueeseen: ikkunan luomiseen, pääsilmukkaan, syötteiden hallintaan, piirtämiseen ja äänien toistamiseen. Jokainen osa-alue luotiin hyödyntämällä alan kirjallisuutta.

Teoriaosuuden jälkeen rakennettiin Java-peliä. Pelin tekeminen jaettiin osiin, jotka sisälsivät aina yhden kokonaisuuden. Pelin rakenteita tutkittiin eri näkökulmista ja menetelmiä niiden toteuttamiseen analysoitiin tarkoin, kunnes löydettiin yksinkertainen mutta toimiva ratkaisu. Joitain pelin rakenteita jouduttiin muuttamaan useaankin otteeseen, koska alkuperäinen toteutus ei ollut suorituskyvyn kannalta toivottu. Pelin rakenne kuitenkin parantui huomattavasti näiden lukuisten uudelleensuunnittelujen takia.

Työn tärkeimpänä tuloksena syntyi Java-peli, jossa käytettiin dokumentin teoriaosuudessa luotua pohjaa. Työssä kohdatut ongelmat eivät olleet pelin kannalta kriittisiä, mutta oppimisen kannalta niillä oli tärkeä rooli. Peli saatiin toimimaan, eikä suuria ongelmia ilmennyt peliä pelatessa pienien testijaksojen aikana.

Avainsanat peli, ohjelmointi, Java

TAMK University of Applied Sciences
Business Information Systems

Writers	Mika Laurila and Tommi Laurila
Thesis	Game Programming with Java Case: Space Shootah
Pages	48
Graduation time	April 2009
Thesis Supervisor	Paula Hietala

ABSTRACT

This thesis work had no client. The work started with a dream of founding our own game company. The main contributions of this thesis were a Java game and this document. The purpose of this document is to give valuable knowledge for those who are interested in game programming in general and, more importantly, for those who are interested in programming games with Java. The basic structure of the game has been defined as clearly as possible in this document.

The work began with defining the basic structure of a Java game. It was divided into five topics. The topics contained information about how to create a frame, how the main loop is built, how to draw images, how to deal with inputs and finally, how to play music. Every topic was carefully defined by examining and analyzing appropriate literature.

The actual game development began after the game canvas was created. The game was divided into different components. Each component was examined carefully and different methods were analyzed until a simple and working method was found. The process was not straight-forward and during the developmental phase, some of the game structure was modified several times because of performance issues. The game improved significantly because of these numerous redesigns.

The most important result of this work was the working Java game. It was created by using the methods covered in this document. The issues which were encountered in this project were not critical to the game itself, but they were an important part of the learning process.

Keywords game, programming, Java

Sisällysluettelo

1 Johdanto....5
1.1 Tausta....5
1.2 Opinnäytetyön tarkoitus ja tavoitteet....5
1.3 Opinnäytetyön rajaus....6
2 Pelin pohja....7
2.1 Ikkunan alustaminen....8
2.1.1 Ikkunatila....9
2.1.2 Täysnäyttötila....10
2.1.3 Passiivinen ja aktiivinen renderöinti....11
2.1.4 Tuplapuskurointi....13
2.2 Pääsilmukka....16
2.3 Syötteiden hallinta....20
2.4 Piirtäminen....22
2.5 Äänet....24
2.5.1 AudioClip....25
2.5.2 Java Sound API....25
3 Case: Space Shootah....29
3.1 Suunnitteluvaihe....29
3.2 Toteutusvaihe....31
3.2.1 Pohja....31
3.2.2 Vierivät taustat ja lentävät objektit....34
3.2.3 Pelaaja ja viholliset....36
3.2.4 Ampuminen....38
3.2.5 Törmäyksen tunnistaminen....39
3.2.6 Pelin rakenne....40
3.2.7 Lisävoimat ja käytetyt matemaattiset kaavat....42
3.2.8 Äänet....43
3.2.9 Tuloksien tallentaminen tiedostoon....43
3.3 Lopputulos....44
4 Yhteenveto....45
Lähteet....47

1 Johdanto

1.1 Tausta

Java on peliohjelmointikielenä nykyään varsin varteen otettava vaihtoehto. Aikaisemmin Java oli suhteellisen hidas kieli, mutta tämä tilanne on muuttunut viimevuosien aikana, eikä se juuri häviä C++:lle suorituskyvyssä. Vaikka C++ on vieläkin dominoiva ohjelmointikieli peliteollisuudessa, monet muut kielet kuten Java, C#, AS3 ja Python ovat nousseet yhä suositummiksi pelialalla. Java on esimerkiksi varsin suosittu kieli mobiilipeliohjelmoinnissa.

Oma peliharrastus taustalla ja haave omasta peliyrityksestä olivat suurimpia syitä tämän opinnäytetyön tekemiseen. Saimme tästä myös samalla kokemusta. Valitsimme ohjelmointikieleksi Javan, koska se on alustariippumaton ja helpompi oppia kuin C/C++ ja koska se on varsin yleinen ohjelmointikieli kouluissa. Java sisältää runsaasti valmiita kirjastoja, jotka auttavat pelin kehittämisessä. Erillisten kirjastojen lataaminen ei siis ole välttämätöntä. Javaan kuitenkin löytyy tarvittaessa hyvä määrä erilaisia kirjastoja, jotka auttavat pelien kehittämisessä.

1.2 Opinnäytetyön tarkoitus ja tavoitteet

Työn on ennen kaikkea tarkoitus olla dokumentti henkilöille, jotka ovat kiinnostuneet ohjelmoimaan pelejä Javalla tai ovat muuten vaan kiinnostuneet peliohjelmoinnista. Tarkoituksena siis ei ole aloittaa ohjelmoinnin perusteista, vaan tämän dokumentin tarkoitus on selventää, miten pelejä voidaan ohjelmoida Javalla. Tässä annetaan myös lukijalle vinkkejä, minkälaisia lähestymistapoja on olemassa ja mitä valmiita tekniikoita sekä moottoreita kannattaa kokeilla sitten, kun oma osaaminen on sillä tasolla, että kannattaa niihin ruveta tutustumaan. Tekstin ohella esitetään kuvioita ja koodiesimerkkejä lukijalle tekstin tueksi.

Tavoitteena on tuottaa sellainen paketti, joka kattaa tärkeimmät asiat liittyen peliohjelmointiin Javalla. Yksi tärkeimmistä tavoitteista on kertoa nämä seikat mahdollisimman

yksinkertaisesti ja selkeästi lukijalle, jotta hän ymmärtää lukemaansa. Koska peliteollisuus on yksi suurimmista viihteen aloista, on tavoitteena saada ihmisiä kiinnostumaan peliohjelmoinnista yleensä sekä peliohjelmoinnista Javalla.

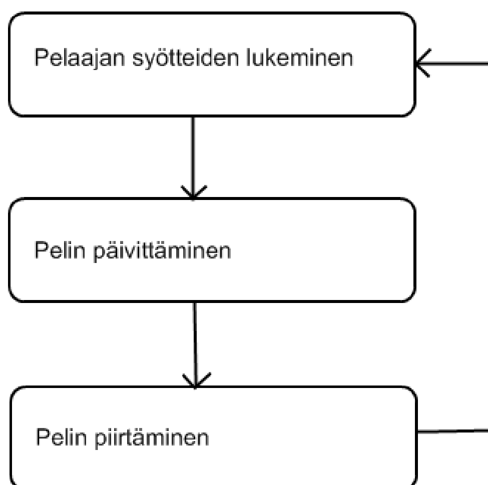
1.3 Opinnäytetyön rajaus

Tehtävä rajattiin niin, että se kattaa kaikki ne perusasiat, joita pelejä kehitettäessä tarvitaan. Tällaisia asioita ovat ympäristön alustaminen, syötteiden hallinta, grafiikan piirtäminen, pelilogiikan päivittäminen ja äänten soittaminen. Opinnäytetyöhön ei sisällytetä 3D-ohjelmointia, vaan se rajoittuu pelkästään 2D-maailmaan, ja käytämme hyväksi työssä vain Javan omia kirjastoja (Java 2D API ja Java Sound API).

2 Pelin pohja

Yleisin ongelma aloittelijoille on se, kuinka peli käytännössä rakentuu ja mihin mikäkin osa toteutetaan. Ongelmana on myös itse ikkunan alustaminen, ainakin ohjelmoijille, jotka eivät ole graafisia sovelluksia ennen tehneet tai ovat kopioineet jostain valmiista pohjasta pohjan itselleen ja muokanneet sitä. Internetistä löytyy kyllä paljon valmiita malleja, mutta parempi on sisäistää asiat itse, kuin kopioida muilta ymmärtämättä asian ydintä.

Peli rakentuu kolmesta osasta: pelin alustamisesta, itse pelistä ja pelin lopettamisesta. Pelin alustamiseen kuuluu ikkunan alustaminen, pelin resurssien lataaminen ja tarvittavien pelikomponenttien luominen. Pelin lopettamisessa on kyse pelin alustamisen päinvastaisesta toimesta eli pelin kaikkien resurssien vapauttamisesta. Itse peli onkin hieman monimutkaisempi osa. Se sisältää syötteiden tarkistukset, pelilogiikan päivittämisen ja pelin piirtämisen ruudulle. Kaikki nämä päivitetään yhden silmukan, pääsilmukan sisällä, niin kauan kunnes peli lopetetaan (Kuvio 1). On kuitenkin hyvä muistaa, että ei ole vain yhtä oikeata tapaa tehdä asioita. Pääsilmukan luomiseen on monia eri tapoja ja jokaisessa tavassa on hyvät ja huonot puolet. Tässä dokumentissa kerromme helpon tavan tehdä asioita. (Clingman 2004, 24-25.)



Kuvio 1. Yleinen kuva pelin perusrakenteesta

2.1 Ikkunan alustaminen

Ohjelmointikielestä riippumatta - poikkeuksena jotkin dynaamiset ohjelmointikielet kuten ActionScript (Flash) - ensimmäisenä tehtävänä on ikkunan alustaminen. Javassa ikkuna tai ympäristö voidaan alustaa monella tapaa. Tutuin tila näistä on ikkunatila (windowed mode). J2SE version 1.4:n myötä tuli mahdollisuus käyttää täysnäyttötilaa eli koko näyttö -tilaa (full-screen exclusive mode; FSEM). Ikkunatilalla tarkoitetaan ikkunaa, joka ei ole kooltaan kuvaruudun kokoinen. On myös mahdollista, että se on kuvaruudunkokoinen, mutta sillä on silti ikkunan kehykset ja taustalla piirretään muut sovellukset normaalisti, joten sovellus ei saa kaikkea suorituskykyä käyttöönsä. Vastaavasti täysnäyttötilalla tarkoitetaan tilaa, missä koko näyttö on varattu pelille ja muu toiminnallisuus taustalla keskeytetään. (Davison 2005.)

FSEM-tilassa saadaan paremmin näytönohjaimen suorituskyky käyttöön ja tästä syystä FSEM-tila on paljon nopeampi kuin normaali ikkunatila. Nopeampi suorituskyky on etu peleissä, mutta myös muissa graafisissa sovelluksissa. Suorituskykyä vaativia pelejä ovat esimerkiksi nopeatempoiset tasoloikkapelit, räiskintäpelit ja urheilupelit. Edellä mainituille pelityypeille suorituskyky on oltava kohdallaan tai muuten pelaaminen kärsii. Ikkunatilalle sopivia pelejä ovat korttipelit ja puzzle-pelit, joissa piirtäminen voi olla satunnaista, ja nopeudella ei ole niin väliä. Nopeuden lisäksi FSEM-tila mahdollistaa värisyvyyden (color depth tai bit depth) ja ruudun resoluution muuttamisen. FSEM-tilassa on myös mahdollista käyttää kehittynyttä graafista tekniikkaa, page-flipping-tekniikkaa, josta puhutaan myöhemmin tuplapuskuroinnin yhteydessä. (Davison 2005.)

Käytettiinpä ikkunatilaa tai FSEM-tilaa, vaatii se toimiakseen luokan, joka periytyy java.awt.Window-luokasta. Window-tyypin luokkia ovat esimerkiksi java.awt.Frame tai java.swing.JFrame. Käytämme tässä opinnäytetyössä esimerkeissä JFramea, koska se on keveä (lightweight) komponentti. Tämä tarkoittaa sitä, että se on kokonaan Javalla tehty, mikä vastaavasti varmistaa sen, että sovellus toimii samalla tavalla alustasta riippumatta (Fletcher 2001).

2.1.1 Ikkunatila

Ikkunatilan alustaminen vaatii siis luokan, joka periytyy JFrame-luokasta tai jolla on kyseinen objekti tietojäsenenään. On makuasia kummalla tekniikalla sen tekee. Tässä opinnäytetyössä käytämme periyttämistä. Ihan ensimmäiseksi on hyvä määritellä tiettyjä arvoja JFramelle.

Ikkunaan vaikuttavia metodeita ovat muun muassa setResizable(...), jonka avulla voidaan estää ikkunan koon muuttaminen; setVisible(...), jonka avulla saadaan kehys näkymään ja setPreferredSize(...), jonka avulla voidaan määritellä kehyksen koko. On hyvin tyypillistä lisätä myös ikkunan sulkemiselle oletustoiminto. Toimintoa kutsutaan silloin, jos ikkuna suljetaan painamalla rastia (jos kyseessä on esimerkiksi Windows) ikkunan kehyksestä. Tähän tarkoitukseen JFramesta löytyy setDefaultCloseOperation(...)-metodi. Metodi ottaa parametrinaan kokonaisluvun ja käyttökelpoiset kokonaisluvut löytyvät WindowConstants-luokasta. Edellä mainittujen lisäksi alustusmetodi sisältää setIgnoreRepaint(...)-metodin, joka selitetään myöhemmin aktiivisen renderöinnin yhteydessä. (Lesson: Using.... 1997.) Alla on ohjelmakoodiesimerkki (Koodiesimerkki 1) ikkunatilan alustamisesta.

```
public void initWindow() {
    // Asettaa ikkunalla suosituskoon 640 x 480.
    setPreferredSize(new Dimension(640, 480));

    // Pistää suosituskoon käyttöön.
    pack();

    // Kytkee pois päältä koon muuttamisen
    setResizable(false);

    // Jättää huomioimatta järjestelmän lähettämät
    // piirtokutsut.
    setIgnoreRepaint(true);

    // Pistää ikkunan näkyviin
    setVisible(true);

    // Asettaa ikkunalle oletustoiminnon, kun rastia
    // painetaan.
    setDefaultCloseOperation(WindowConstans.EXIT_ON_CLOSE);
}
```
Koodiesimerkki 1. Ikkunatilan alustusmetodi

2.1.2 Täysnäyttötila

Täysnäyttötilan eli FSEM-tilan alustaminen ei eroa merkittävästi ikkunatilan alustamisesta. Ainoa iso ero on siinä, että sen alustamiseen tarvitaan java.awt-paketista löytyviä GraphicsEnvironment- ja GraphicsDevice-objekteja. GraphicsEnvironment-objektin avulla haetaan GraphicsDevice-objekti. GraphicsDevice-objektilla voidaan tarkistaa, tukeeko alusta täysnäyttötilaa vai ei. Jos kyseinen alusta tukee FSEM-tilaa, sen käyttöön ottaminen tapahtuu kutsumalla GraphicsDevice- objektin setFullScreenWindow(...)-metodia, jonka parametrina välitetään Window-tyypin objekti eli esimerkiksi tässä tapauksessa JFrame. (Martak 2001.)

Resoluutio on mahdollista muuttaa kutsumalla GraphicsDevice-objektin metodia setDisplayMode(...). Metodin parametrina välitetään DisplayMode-objekti, joka ottaa parametriksi vastaan resoluution leveyden ja korkeuden ja sen lisäksi bittisyvyyden ja virkistystaajuuden. Parametri hyväksyy yleisesti tunnetut bittisyvyydet kuten 8 bittiä, 16 bittiä, 32 bittiä ja sekä vaihtoehtoisesti DisplayMode.BITH_DEPTH_MULTI, jos näyttö tukee kaikkia edellä mainittuja bittisyvyyksiä. Virkistystaajuus kannattaa asettaa arvoksi DisplayMode.REFRESH_UNKNOWN, jos ei ole varma näytön virkistystaajuuksista. Väärä virkistystaajuus voi vahingoittaa näyttöä, ainakin kuvaputkinäyttöä. Näytön tuetut resoluutiot, bittisyvyydet ja virkistystaajuudet voidaan hakea GraphicsDevice-objektilta kutsumalla getDisplayModes()-metodia. Metodi palauttaa DisplayMode-objektitaulun, joten sen avulla on helppo asettaa näyttö haluttuun tilaan. (Martak 2001.) Alla on Koodiesimerkki 2, kuinka alustettaisiin näyttö tiettyyn resoluutioon.

```
// GraphicsDevice-objekti, jota käytetään FSEM-tilan
// alustamiseen ja FSEM-tilasta poistumiseen.
private GraphicsDevice graphicsDevice;

public void initFSEM() {
    // Hakee GraphicsEnvironment-objektin.
    GraphicsEnvironment ge =
        GraphicsEnvironment.getLocalGraphicsEnvironment();

    // Hakee oletus näyttölaitteen eli GraphicsDevice-objektin.
    graphicsDevice = ge.getDefaultScreenDevice();

    // Poistaa ikkunan koon muuttamisen
    setResizable(false);

    // Poistaa ikkunalta kehykset
    setUndecorated(true);

    // Jättää huomioimatta järjestelmän lähettämät piirtokutsut.
    setIgnoreRepaint(true);

    // Tarkistaa tukeeko näyttö täysnäyttötilaa.
    if(graphicsDevice.isFullScreenSupported()) {
        // Asettaa FSEM-tilan.
        graphicsDevice.setFullScreenWindow(this);

        // Vaihtaa resoluution, bittisyvyyden ja
        // virkistystaajuuden näytölle.
        graphicsDevice.setDisplayMode(
            new DisplayMode(640, 480,
            DisplayMode.BIT_DEPTH_MULTI,
            DisplayMode.REFRESH_RATE_UNKNOWN));
        }
}
```
Koodiesimerkki 2. Täysnäyttötilan alustusmetodi

2.1.3 Passiivinen ja aktiivinen renderöinti

Javassa voidaan tehdä renderöinti eli piirtäminen ruudulle kahdella tapaa: joko aktiivisesti tai passiivisesti. Passiivisella renderöinnillä (passive rendering) tarkoitetaan sitä, että GUI-sovelluksessa (Graphics User Interface) käyttöjärjestelmä lähettää viestin Swing-komponentille, joka vastaavasti voi kutsua paint(...)-metodia (Koodiesimerkki 3). Metodi paint(...) sisältää konkreettiset piirtokutsut, mutta metodia ei koskaan kutsuta suoraan. Käytännössä tämä tarkoittaa sitä, että paint(...)-metodia voidaan kutsua käyttöjärjestelmästä silloin, kun ikkunan kokoa muutetaan, ikkunaan napautetaan hiirellä tai repaint()-metodia kutsutaan ohjelmakoodissa. Hyvin yleinen tapa on kutsua repaint()-

metodia pääsilmukassa. Metodin repaint() kutsuminen pääsilmukassa ei kuitenkaan takaa sitä, että sovellus piirtäisi odotetulla tavalla. Metodi repaint() kutsuu paint(...)-metodia, kuten jo edellä tuli ilmi, ja tämä prosessi menee järjestelmän kautta. Järjestelmän kautta tapahtuva piirtäminen rasittaa itse renderöintiprosessia ja tulos ei välttämättä ole suorituskyvyn kannalta odotettu. Renderöinti on satunnaista, eikä kehittyneen graafisen, page-flipping -tekniikan, käyttö ei ole mahdollista. Parempi vaihtoehto on siis käyttää aktiivisista renderöintiä – ainakin peleissä. (Martak 2001.)

```
// Järjestelmä kutsuu tätä metodia tarvittaessa.
@Override
public void paint(Graphics g) {
    // Piirtäminen
    g.drawString("Hello World", 50, 50);
}
```
Koodiesimerkki 3. Passiivinen renderöinti

Aktiivinen renderöinti (active rendering) tarkoittaa sitä, että ohjelmassa kutsutaan renderöintimetodia (Koodiesimerkki 4). Tämän takia setIgnoreRepaint(...)-metodia on syytä kutsua ja asettaa sen arvoksi true, jotta järjestelmä ei vahingossakaan kutsu paint(...)-metodia. Metodi sulkee kaikki järjestelmän lähettämät piirtokutsut kokonaan. Renderöinti voidaan tehdä suoraan while-silmukan sisällä. Aktiivinen renderöinti on varsin hyödyllinen sovelluksissa, jotka tarvitsevat nopeutta ja täydellistä kontrollointia, kuten esimerkiksi peleissä. Passiivisen ja aktiivisen renderöinnin samanaikainen käyttö on mahdollista, mutta tämä voi aiheuttaa erilaisia ongelmia, kuten sovelluksessa tapahtuvia niin kutsuttuja lukkoja (deadlock). (Martak 2001.)

Lukkoja voi syntyä silloin, kun kaksi tai useampi säie ei saa suoritusaikaa. Tässä tapauksessa säikeet voivat olla pääsilmukan oma säie ja Swing-komponentin niin kutsuttu Event Dispatch -säie. Pääsilmukassa kutsutaan itse tehtyä piirtometodia ja Swing-komponentti kutsuu omaa paint(...)-metodia. Säikeet yrittävät päästä samoihin resursseihin käsiksi ja näin ollen odottavat ikuisesti vuoroaan. Tilanteesta syntyy lukko. (Lesson: Concurrency: Deadlock 2006.)

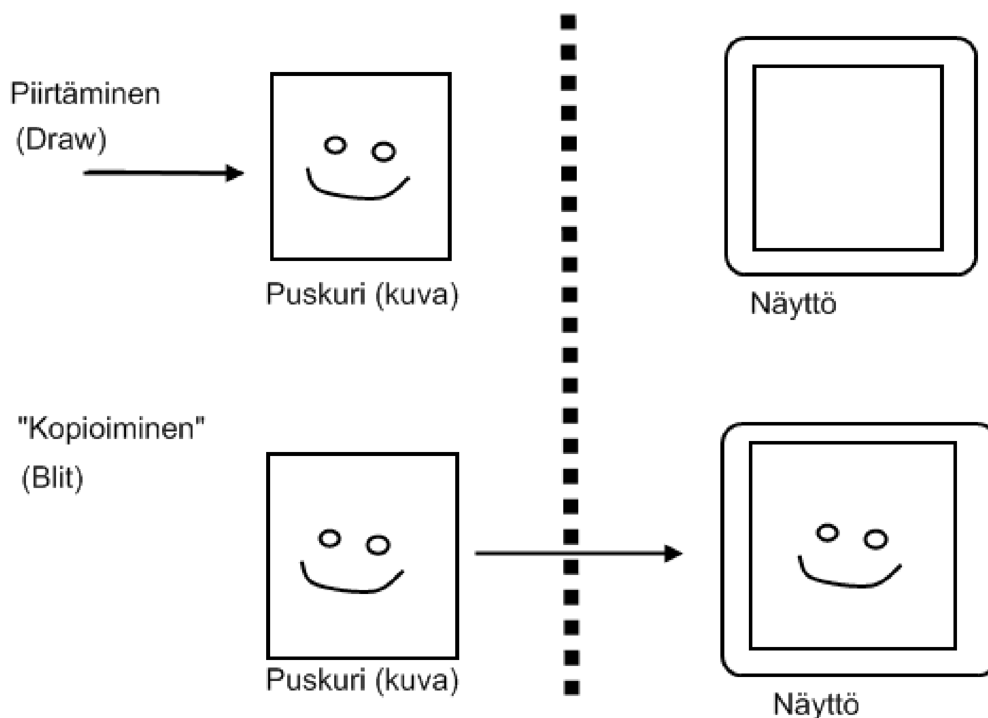
```
// Piirtää silmukan sisällä.
while(true) {
    Graphics g = getGraphics();
    g.drawString("Hello World", 50, 50);
    g.dispose();
}
```

Koodiesimerkki 4. Aktiivinen renderöinti

2.1.4 Tuplapuskurointi

Tuplapuskuroinnin (double-buffering) ja page-flipping-tekniikoiden käyttö on yleinen tapa estää ruudun välkkyminen (Brackeen 1996). Tuplapuskurointi ei välttämättä paranna suorituskykyä, vaan voi jopa heikentää sitä. Tärkeämpää kuin numerot suorituskykyä mitattaessa, on visuaalinen kokemus itse pelistä (Martak 2001). Ilman tuplapuskurointia peli voi näyttää monin verroin huonommin tehdyltä.

Tuplapuskurointi tarkoittaa sitä, että piirretään ensimmäiseksi puskuriin eli kuvaan (offscreen image tai back buffer) ja sen jälkeen kuva kopioidaan näytölle (Kuvio 2). Tuplapuskuroinnin yksi muoto on page-flipping-tekniikka. Page-flipping eroaa hieman tuplapuskuroinnista. Sen sijaan, että ”kopioidaan” kuva näytölle, vaihdetaan vain videoosoittimien paikkoja. FSEM-tila, aktiiviinen renderöinti ja laitteiston tuki (tarpeeksi muistia näytönohjaimessa) ovat edellytyksiä page-flipping -tekniikan käytölle. (Martak 2001.)



Kuvio 2. Tuplapuskurointi

Puskuri voidaan luoda Javassa käyttämällä java.awt.image.BufferStrategy-luokkaa. Luokka käyttää puskurinaan java.awt.image.VolatileImagea. VolatileImage on laitekiihdytetty. Laitekiihdytyksellä tarkoitetaan sitä, että jokin muu kuin CPU (Central Processing Unit eli prosessori) hoitaa renderöinnin ja yleensä se on laitteesta löytyvä näytönohjain (Wikipedia: Hardware... 2008). Näytönohjaimen avulla saadaan paljon enemmän suorituskykyä renderöintiin.

Tuplapuskuroinnin alustaminen on Javassa helppoa, jos käyttää BufferStrategyä puskuristrategiana. JFrame-luokasta löytyy valmiiksi metodi createBufferStrategy(...). Metodin parametrina tulee puskurien lukumäärä. Tarvittaessa voidaan hakea laitteiston rajoitukset kutsumalla java.awt.GraphicsConfiguration-luokan getBufferCapabilities()-metodia. Tämä voidaan tehdä vasta, kun JFrame on luotu. On siis hyvin mahdollista antaa puskurien määräksi esimerkiksi kolme, jos vain laitteisto tukee sitä. Jotta renderöinti onnistuu käyttäen BufferStrategya, on hyvä antaa BufferStrategy myös luokan muuttujaksi. Koodiesimerkistä (Koodiesimerkki 5) huomataan puskuristrategian alustamisen helppous.

```
// Luokan muuttuja
private BufferStrategy bufferStrategy;

public void initBufferStrategy() {

    // Luo uuden puskuristrategian.
    // Parametrina tulee puskurien lukumäärä.
    createBufferStrategy(2);

    // Hakee juuri luodun BufferStrategyn
    // luokan muuttujaan.
    bufferStrategy = getBufferStrategy();
}
```

Koodiesimerkki 5. BufferStrategyn alustaminen.

Käytämme siis luotua BufferStrategy-objektia puskuristrategiana ja käytämme sitä tietysti myös renderöintiin. BufferStrategyn käyttö on hyvin yksinkertaista. Haetaan ensimmäiseksi BufferStrategy-objektilta Graphics-objekti ja muutetaan se Java 2D API:n Graphics2D-objektiksi. Sen jälkeen kutsutaan piirtometodia, jonka parametrina välitetään juuri haettu Graphics2D-objekti. Piirtäminen tapahtuu tässä vaiheessa takapuskuriin. Sen jälkeen on hyvä vapauttaa Graphics-objekti kutsumalla dispose()-metodia, koska sitä ei tarvita enää sillä silmukan kierroksella. Kun kaikki edellä mainittu on tehty, on aika näyttää puskuri ruudulla eli kutsua BufferStrategyn show()-metodia (Koodiesimerkki 6).

```
private void render() {
    // Graphics2D-objekti
    Graphics2D g = null;

    try {
         // Hakee Graphics-objektin BufferStrategylta ja
         // muuttaa sen Graphics2D-objektiksi.
         g = (Graphics2D) bufferStrategy.getDrawGraphics();

         // Piirtää puskuriin.
         draw(g);
     }
     finally {
         // Vapauttaa Graphics2D-objektin, kun sitä
         // ei enää tarvita.
         g.dispose();
     }

     // Näyttää puskurin näytölle.
     bufferStrategy.show();
}
```
Koodiesimerkki 6. Renderöinti BufferStrategyllä (Martak 2001)

2.2 Pääsilmukka

Pelin perusrakenteeseen kuuluu while-silmukka. Silmukan voi tehdä ilman säiettä (thread) tai säikeen kanssa. Säikeiden ansiosta voidaan tehdä monta asiaa rinnakkain. Tämä tarkoittaa sitä, että esimerkiksi yhdessä säikeessä piirretään latausruutua ja toisessa säikeessä ladataan resursseja, kun ilman säikeitä pystyttäisiin tekemään vain toista asiaa kerralla. Ensimmäiseksi siis piirrettäisiin latausruutu ja sitten vasta ladattaisiin resurssit. Jälkimmäinen tapa on hitaampi ja latausruutuun on mahdotonta tehdä minkäänlaista animaatiota. Käytämme tässä opinnäytetyössä säikeistettyä silmukkaa.

Java tukee siis säikeitä ja ohjelmoijan näkökulmasta sovelluksessa on aina vähintään yksi pääsäie. Pääsäikeen lisäksi voidaan luoda useampia säikeitä joko käyttäen abstraktia java.lang.Thread-luokkaa tai toteuttamalla java.lang.Runnable-rajapinta (Koodiesimerkki 7). Kummassakin tapauksessa luokan pitää toteuttaa run()-metodi, jota säie kutsuu sen käynnistyessä. (Lesson: Concurrency... 2006.)

```
// Esimerkki luokasta, joka toteuttaa säikeen
// rajapinnan (Runnable) kautta.
public class TestThread implements Runnable {

    private Thread gameThread = null;

    public void startGame() {
        if(gameThread == null) {
            gameThread = new Thread(this);
            gameThread.start();
        }
    }

    @Override
    public void run() {
        while(true) {
            update();
            render();
        }
    }

    public static void main(String[] args) {
        new TestThread().startGame();
    }
}
```

Koodiesimerkki 7. Säie-esimerkki käyttäen Runnable-rajapintaa

Javassa ei ole moniperintää, joten rajapinta antaa tietynlaista vapautta ja vaihtoehtoja säikeen toteuttamiseen abstraktiin luokkaan nähden, koska tällöin voidaan pääluokka (tai pohja) periyttää jostain muusta luokasta kuin Thread-luokasta. Alla on koodiesimerkki (Koodiesimerkki 8) abstraktin Thread-luokan toteutuksesta, josta huomaamme, että se ei paljon eroa Runnable-rajapinnan toteutuksesta.

```
// Esimerkki luokasta, joka toteuttaa säikeen
// käyttäen abstraktia luokkaa Thread.
public class TestThread extends Thread {

    @Override
    public void run() {

        while(true) {
            update();
            render();
        }
    }

    public static void main(String[] args) {
        new TestThread().start();
    }
}
```

Koodiesimerkki 8. Säie-esimerkki käyttäen abstraktia Thread-luokkaa

Pelin pää- eli while-silmukkaan tulee, kuten jo aikaisemmin mainittiin, pelin päivitys- ja piirtämismetodit. Sen lisäksi silmukka kannattaa jollain tapaa hidastaa, ettei konetta ylikuormiteta turhaan. Annetaan JVM:lle (Java Virtual Machine) aikaa tehdä työnsä eli esimerkiksi kerätä roskat (garbage collection). Annetaan myös CPU:lle aikaa tehdä muita tehtäviä ja estetään joidenkin piirtotehtävien ylihyppääminen. Thread.sleep(...)-metodi (Koodiesimerkki 9) on yksi tapa toteuttaa silmukan hidastaminen. Se ei ole kovin tarkka, joten siihen ei täysin kannata luottaa. Metodin sleep(...) sijaan Javasta löytyy valmiita metodeita, joiden avulla ajastaminen onnistuu paremmin. (Davison 2005.)

```
@Override
public void run() {
    // Päivitää peliä 60 kuvaa sekunnissa.
    // Jakamalla 1000 / 60 saadaan aika, kuinka kauan
    // joudutaan nukkumaan.
    int sleepTime = 1000 / 60;

    while(gameIsRunning) {
        update();
        render();

        // Nukutaan hieman.
        try {
            Thread.sleep(sleepTime);
        } catch(InterruptedException e) {}
    }
}
```
Koodiesimerkki 9. Thread.sleep()-metodin käyttö

Javasta löytyy System.getCurretTimeMillis()-metodi, joka palauttaa ajan millisekunteina. Metodi getCurrentTimeMillis() ei ole kovin tarkka, eikä sitä nykyään suositella käytettäväksi, koska tilanne muuttui 1.5 version myötä. Javan mukana tuli paljon tarkempi ajastin, System.nanoTime(). Näiden metodien toimintaperiaatteet ovat samat. Kummallakin metodilla on mahdollista laskea yhteen silmukan kierrokseen käytetty aika (Koodiesimerkki 10). Sen avulla voidaan sleep(...)-metodin epätarkuutta korjailla, mutta tässä opinnäytetyössä siihen ei perehdytä sen tarkemmin.

```
long startTime = System.nanoTime();
// Jotain aika vievää tänne, esimerkiki
// update() ja render()

// Palauttaa käytetyn ajan.
long usedTime = Systen.nanoTime() - startTime;
```
Koodiesimerkki 10. Esimerkki ajastimien käytöstä

Pelin nopeus kerrotaan yleensä FPS:nä (frame per second) eli kehysnopeutena. Perusperiaate on se, että mitä suurempi FPS, sitä sulavammin peli pyörii. Sulavan liikkeen aikaansaamiseksi 60 FPS:ää on yleisesti pidetty hyvänä rajana. Osa markkinoilla olevista peleistä pyörii kuitenkin 30 FPS:llä tai jopa alle tuon, joten kovin orjallisesti tuota rajaa ei pidä ottaa. FPS tarkoittaa sitä, kuinka monta kuvaa piirretään sekunnissa näytölle. Se voidaan määritellä esimerkiksi sleep(...)-metodin avulla sijoittamalla sopivan suuruinen millisekuntimäärä parametriksi. Kun 1000 jaetaan halutulla FPS:llä, saadaan milli-

sekunnit, jotka ilmaisevat kuinka kauan pitää nukkua aina yhden silmukan aikana. Nukkumisaikaa voidaan säädellä edellä mainittujen getCurrentTimeMillis()- tai nanoTime()-metodien avulla.

2.3 Syötteiden hallinta

Javassa näppäinkomennot käsitellään käyttämällä hyödyksi API:sta löytyvää rajapintaa java.awt.event.KeyListener. Tämä voidaan tehdä suoraan implementoimalla KeyListener mihin tahansa luokkaan (Koodiesimerkki 11). Luokan pitää toteuttaa tietyt metodit, jotta se menee kääntäjästä läpi. Nämä metodit ovat keyPressed(...), keyReleased(...) ja keyTyped(...), joista kaksi ensimmäiseksi mainittua ovat tärkeimmät. Metodin keyPressed(...) avulla voidaan tarkistaa, mikä nappi on juuri sillä hetkellä pohjassa. Vastaavasti keyReleased(...)-metodilla voidaan tarkistaa, mikä nappi on juuri vapautettu. Luokka on myös rekisteröitävä johonkin komponenttiin, jotta näppäinkomentoja kuunneltaisiin. Tämä tehdään kutsumalla addKeyListener(...)-metodia. Tässä tapauksessa komponentti, johon kuuntelija (KeyListener) rekisteröidään, on JFrame. (Java™ Platform... 2006.)

```
// JFrame toteuttaa KeyListener-rajapinnan.
public class TestFrame extends JFrame implements KeyListener {

    public TestFrame() {
        // Alustetaan JFrame.
        // ...

        // Lisää tämän luokan JFrame-komponentin
        // kuuntelulistalle.
        addKeyListener(this);
    }

    @Override
    public void keyPressed(KeyEvent e) {}

    @Override
    public void keyReleased(KeyEvent e) {}

    @Override
    public void keyTyped(KeyEvent e) {}
}
```
Koodiesimerkki 11. KeyListener-rajapinnan käyttö

KeyListenerin lisäksi Java APIsta löytyy myös java.awt.event.KeyAdapter, joka on vastaavasti abstrakti luokka. Tämä mahdollistaa samat asiat kuin KeyListener, mutta lisäksi voi itse päättää, mitkä edellä mainituista metodeista haluaa toteuttaa. Haittapuolena on se, että KeyAdapter-luokkaa ei voida toteuttaa luokkaan, joka periytyy jo jostain toisesta luokasta. Sen toteuttamiseen vaaditaan erillinen luokka, joka sitten tuodaan tavalla tai toisella JFrame-komponentin rekisteröitäväksi. Alla on koodiesimerkki KeyAdapterin käytöstä (Koodiesimerkki 12). (Java™ Platform... 2006.)

```
// Luokka periytyy KeyAdapter-luokasta.
public class OwnListener extends KeyAdapter {

    @Override
    public void keyPressed(KeyEvent e) {
    }
}

public class TestFrame extends JFrame {
    // Asettaa kuuntelija-objektin luokan muuttujaksi.
    private OwnListener ownListener = new OwnListener();

    public TestFrame() {
        // Alustetaan JFrame.
        // ...

        // Rekisteröi kuuntelijan JFrame-komponentin
        // kuuntelulistalle.
        addKeyListener(ownListener);
    }
}
```

Koodiesimerkki 12. Esimerkki KeyAdapter-luokan käytöstä

Näppäinten kuunteleminen tapahtuu keyPressed(...)-, keyReleased(...)- ja keyTyped(...)-metodien sisällä. Metodit välittävät parametrinaan KeyEvent-objektin. Tämän objektin kautta voidaan kysyä, onko jokin näppäin pohjassa vai ei. KeyEvent-luokka sisältää julkiset ja staattiset int-tyypin muuttujat jokaiselle näppäimelle. Kaikkien kolmen metodin toimintaperiaate on sama. Seuraava koodiesimerkki havainnollistaa hieman tarkemmin, miten näppäinten painalluksia luetaan (Koodiesimerkki 13).

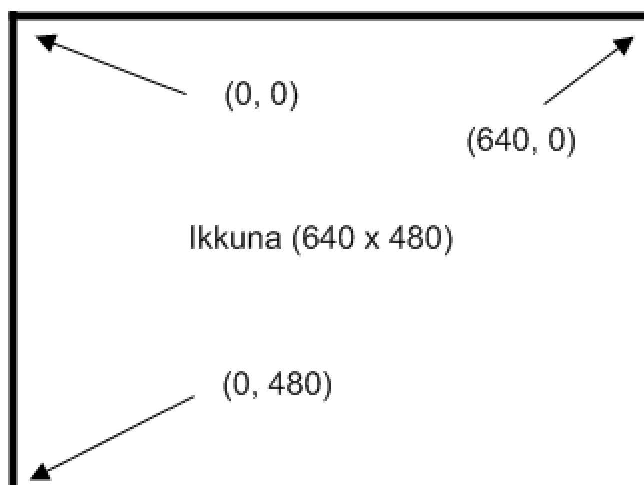
```
@Override
public void keyPressed(KeyEvent e) {
    // Hakee painetun näppäimen arvon.
    int keyCode = e.getKeyCode();

    // Tulostaa tekstin, jos painetun näppäimen
    // arvo on sama kuin Space-näppäimen.
    if(keyCode == KeyEvent.VK_SPACE) {
        System.out.println("Space-näppäin on painettu");
    }
}
```

Koodiesimerkki 13. Esimerkki keyPressed(...)-metodin käytöstä

2.4 Piirtäminen

Javassa voidaan piirtää kuvia, primitiivejä ja merkkijonoja. Javassa piirtäminen tapahtuu Graphics2D-objektin kautta. Vaihtoehtoisesti on myös mahdollista piirtää Graphics-objektin kautta. Erona noilla kahdella on se, että Graphics2D on paljon uudempaa tekniikkaa kuin Graphics ja se kuuluu varsinaisesti Java 2D APIn piiriin. Graphics2D periytyy Graphics-luokasta, mutta se on kauttaaltaan paljon yhtenäisempi ja sillä voi piirtää paljon monipuolisemmin, kuin Graphics-objektilla.

Javassa koordinaatiston origo (0,0) on oletusarvoisesti asetettu vasempaan yläkulmaan. X on suurempi kuin nolla, kun mennään oikealle. Y on suurempi kuin nolla, kun mennään alas koordinaatistossa (Kuvio 3). Origo on mahdollista myös vaihtaa kutsumalla Graphics2D-objektin translate(...)-metodia.



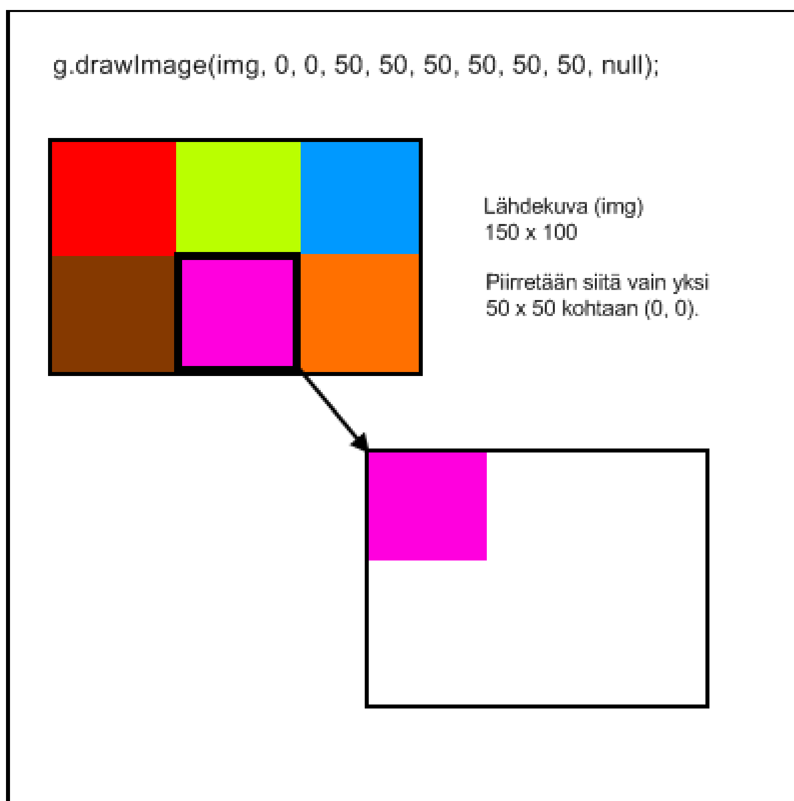
Kuvio 3. Koordinaatisto

Java tukee kuvia kuten PNG, GIF, JPEG, BMP ja WBMP, mutta on myös mahdollista tehdä tuki muille kuvatyypeille. Kuvat on ensin ladattava muistiin, jotta niiden piirtäminen onnistuu. Javasta löytyy paljon erilaisia metodeita, joilla voidaan ladata kuvia. Oikean tai toimivan lataustekniikan löytäminen onkin yksi niistä asioista, jotka joskus aiheuttavat hankaluuksia, koska niitä on Javassa todella paljon. Yksi ehkä käytetyin, helpoin ja nopein tekniikka on käyttää javax.imageio.ImageIO-luokkaa (Koodiesimerkki 14) (Hester 2007). Tämä tekniikka toimii myös, jos kuvia haetaan JAR-paketin sisältä.

```
BufferedImage image = null;

try {
    // Hakee kuvan kalle.png suoraan juuresta.
    URL url = getClass().getResource("kalle.png");
    image = ImageIO.read(url);
}
catch(IOException e) {
    e.printStackTrace();
}
catch(IllegalArgumentException e) {
    e.printStackTrace();
}
```

Koodiesimerkki 14. Kuvan lataus.

Kuvien piirtäminen Javassa tapahtuu kutsumalla Graphics2D-objektista löytyviä drawImage(...)-metodeita ja kuvat voivat olla Image-, BufferdImage- tai VolatileImage-tyyppisiä kuvia. Osalla drawImage(...)-metodeista on mahdollista valita lähdekuvasta vain osa, jonka haluaa piirtää ruudulle (Kuvio 4). Metodi on myös erityisen kätevä animaatioiden sekä tiilien piirtämiseen, kun kaikki animaatiot ja tiilet ovat yhdessä kuvatiedostossa. Tiilellä (tile) tarkoitetaan pientä suorakulmaista kuvaa, joka on osa jotain suurempaa kokonaisuutta. Tiiliä käyttämällä voidaan rakentaa esimerkiksi erilaisia kenttiä. Skaalaus ja kuvan kääntely on myös mahdollista kyseisellä metodilla. Skaalauksen voi myös tehdä Graphics2D-objektista löytyvällä scale(...)-metodilla.



Kuvio 4. drawImage(...)-metodi

Kuvien lisäksi on siis myös mahdollista piirtää primitiivejä ja merkkijonoja. Graphics2D:stä löytyy paljon erilaisia primitiivien piirtämiseen tarkoitettuja metodeja kuten fillRect(...), drawRect(...) ja drawLine(...). Omanlaisia muotoja voidaan piirtää kutsumalla metodia drawShape(...). Metodin fillRect(...) avulla yleensä maalataan tausta tietyllä värillä, jottei liikkuvista kuvista tai animaaitoista jää pikseleitä taustalle. Javasta löytyy kirjasimen valitsemiseen java.awt.Font-luokka, jonka avulla voidaan määritellä käytettävä kirjasin, tyyli ja koko. On kuitenkin suositeltavaa käyttää itse kuvista tehtyjä kirjasimia, jo persoonallisemman tyylin vuoksi. Merkkijonojen piirtäminen tapahtuu drawString(...)-metodia käyttäen.

2.5 Äänet

Äänet ovat hyvin tärkeä osa peliä. Usein luodaan tunnelma äänien ja musiikkien avulla. Tässä alaluvussa kuvataan hyvin lyhyesti, mitä vaihtoehtoja Javasta löytyy ja esitetään yksinkertainen esimerkki kustakin tekniikasta.

2.5.1 AudioClip

Javassa on jo versiosta 1.0 lähtien ollut mukana java.applet.AudioClip, joka on hyvin yksinkertainen rajapinta äänien soittamiseen. Vaikka se on yksinkertainen, sen vahvuutena on lukuisten eri formaattien tukeminen ja monen äänen soittaminen samanaikaisesti. Kun se on hyvin yksinkertainen, siitä puuttuu myös joitakin tärkeitä elementtejä, joita peleissä saatetaan tarvita. Esimerkiksi ei ole mahdollista saada selville, koska äänitiedosto on loppunut, ja ajonaikaisten muutoksien tekeminen ole mahdollista äänitiedostoihin (kuten äänen voimakkuuden säätäminen). Äänitiedoston lataaminen ja sen soittaminen, on kuitenkin hyvin yksinkertainen toimenpide (Koodiesimerkki 15). (Davison 2005.)

```
URL url = getClass().getResource("sound.wav");
AudioClip clip = Applet.newAudioClip(url);
clip.play();
```
Koodiesimerkki 15. AudioClipin käyttö.

2.5.2 Java Sound API

Java 1.3 esiteltiin kuitenkin paljon AudioClipiä kehittyneempi API, Java Sound API. APIn avulla voidaan esimerkiksi soittaa, nauhoittaa, tallentaa ja muokata ääniä. Java Sound API tukee esimerkiksi MIDI- ja WAV-musiikkitiedostoja. Javasta löytyy paketit javax.sound.midi ja javax.sound.sampled. Paketti javax.sound.midi tarjoaa rajapinnat MIDIen lukemiseen, ja paketti javax.sound.sampled tarjoaa vastaavasti rajapinnat esimerkiksi WAV-tiedostojen lukemiseen. Javasta löytyy myös spi-paketit midi- ja sampled-pakettien alta, joiden avulla voidaan tehdä tuki muille ääniformaateille.

MIDI

MIDI-tiedostojen soittamiseen tarvitaan javax.sound.midi.Sequencer ja javax.sound.midi.Sequence. Edellä mainittujen lisäksi on hyvä implementoida javax.sound.midi.MetaEventListener, jonka avulla voidaan kuunnella erilaisia tapahtumia kuten äänitiedoston loppumista. MetaEventListener toimii samalla tavalla kuin esimerkiksi KeyEventListener. Luokka toteuttaa tietyn metodin, joka tässä tapauksessa on meta(...)-metodi, jonka parametrina välitetään MetaMessage-objekti. MetaMessage-objektilta voidaan kysyä kaikenlaisia tapahtumia. Yksinkertaisin sovellus MIDIn soittamiseen näkyy alla olevassa koodiesimerkissä (Koodiesimerkki 16).

```
try {
    // Hakee Sequencer-objektin.
    Sequencer sequencer = MidiSystem.getSequencer();

    // Hakee Sequence-objektin eli äänitiedoston.
    Sequence sequence = MidiSystem.getSequence(
    this.getClass().getResource("music.mid"));

    // Avaa Sequencerin.
    sequencer.open();

    // Asettaa äänitiedoston Sequenceriin.
    // Tämän jälkeen soittaminen on mahdollista.
    sequencer.setSequence(sequence);

    // Soitetaan MIDI-äänitiedosto kutsumalla start()-metodia.
    sequencer.start();

} catch(MidiUnavailableException e) {
} catch(InvalidMidiDataException e) {
} catch(IOException e) {}
```

Koodiesimerkki 16. MIDI-tiedoston soittaminen

Vastaavasti javax.sound.sampled tarjoaa enemmän vaihtoehtoja. Sampled-paketista löytyy luokat javax.sound.sampled.Clip ja javax.sound.sampled.SourceDataLine, joita voidaan käyttää äänten kuuntelemiseen.

Clip

Clip on lähes samanlainen kuin AudioClip, joka aikaisemmin jo mainittiinkin. Clip-objekti pystyy soittamaan äänitiedoston, joka on melko pieni. Pienellä tässä tapauksessa tarkoitetaan alle 2-5MB kokoluokan äänitiedostoa. Suurempien äänitiedostojen soittamiseen kannattaa käyttää DataSourceLine-objektia, josta kerrotaan myöhemmin. Clip on siis objekti, joka ladataan valmiiksi muistiin ennen sen soittamista. Tästä syystä äänen pituus on helppo saada selville, mikä vastaavasti helpottaa aloituskohdan säätämistä ja kappaleen uudelleen toistamista (looppaamista). Alla on esimerkki yksinkertaisesta Clip-soittimesta (Koodiesimerkki 17). (Davison 2005.)

```
try {
    // Avaa äänitiedoston virtana.
    AudioInputStream stream =
    AudioSystem.getAudioInputStream(
        getClass().getResource("music.wav"));

    // Hakee formaatin.
    AudioFormat format = stream.getFormat();

    // Kerää tiedot DataLine luomiseen.
    DataLine.Info info =
        new DataLine.Info(Clip.class, format);

    // Luo tyhjän Clipin käyttäen DataLine tietoja hyväksi.
    Clip clip = (Clip) AudioSystem.getLine(info);

    // Alkaa kuuntelemaan Clipin tapahtumia.
    clip.addLineListener(this);

    // Avaa virran Clippina. Tiedosto on nyt soitettavissa.
    clip.open(stream);

    // Sulkee virran.
    stream.close();

    // Asettaa Clipin aloittamaan alusta.
    clip.setFramePosition(0);

    // Soittaa Clipin.
    clip.start();

} catch (UnsupportedAudioFileException e) {
} catch (IOException e) {
} catch (LineUnavailableException e) {}
```

Koodiesimerkki 17. Clipin avulla äänitiedoston soittaminen

DataSourceLine

DataSourceLine eroaa Clip:stä siten, että se pystyy käsittelemään isojakin tiedostoja. DataSourceLine on niin sanottu puskuroitu virta. Puskuriin virtaa koko ajan dataa, joten äänitiedostoa ei tarvitse ladata kokonaan valmiiksi muistiin. DataSourceLinen alustaminen on verrattavissa Clipin alustamiseen. Clipin sijaan luodaan DataSourceLine-objekti. DataSourceLine-luokan objektia käytettäessä joudutaan lukemaan ja soittamaan vain osan äänestä kerrallaan käyttämällä esimerkiksi while-silmukkaa (Koodiesimerkki 18). Seuraava koodiesimerkki on otettu suoraan Java Sound Programmers Guidesta (Java Sound... 2000).

```java
// Aloitetaan.
line.start();

// Luetaan osa kerralla.
while (total < totalToRead && !stopped) {
    numBytesRead = stream.read(myData, 0, numBytesRead);

    if (numBytesRead == -1)
        break;

    total += numBytesRead;
    line.write(myData, 0, numBytesRead);
}

// Luetaan loppuun, pysäytetään ja suljetaan.
line.drain();
line.stop();
line.close();
```

Koodiesimerkki 18. DataSourceLinen äänidatan lukeminen (Java Sound... 2000)

3 Case: Space Shootah

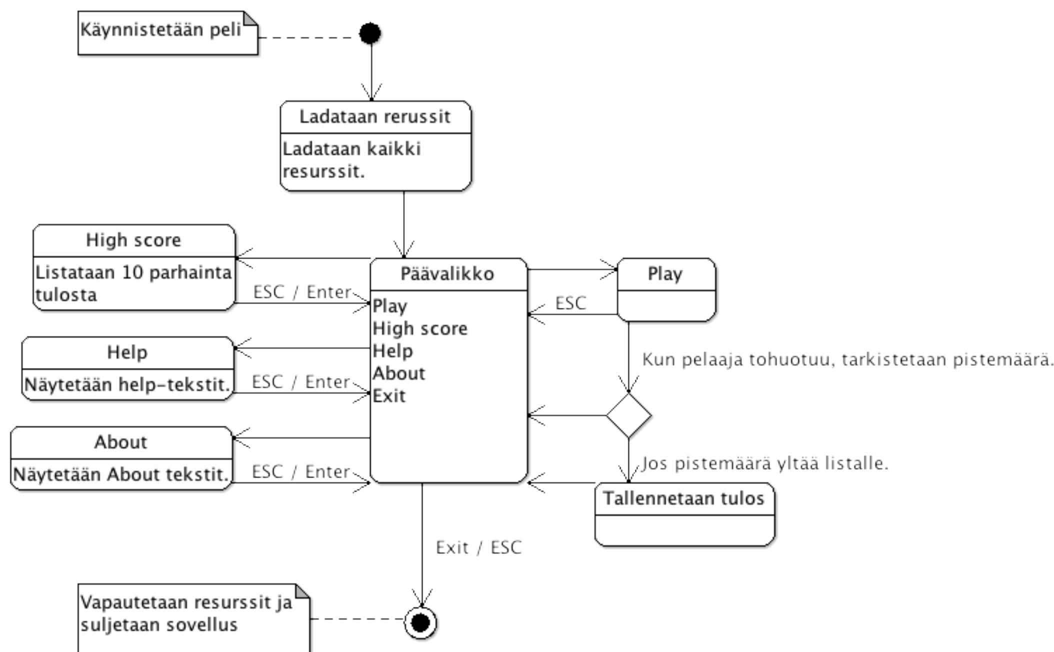
Tarkoituksenamme oli rakentaa yksinkertainen, täysin toimiva avaruusräiskintäpeli, joka pohjautuu jo tässä opinnäytetyössä esitettyyn pohjaan. Sen lisäksi tässä luvussa kuvaamme myös muutamia tekniikoita, joita muissakin peleissä voidaan käyttää, kuten esimerkiksi vierivät taustat.

Valitsimme avaruusräiskintäpelin aiheeksemme, koska se on hyvin yksinkertainen toteuttaa, mutta toisaalta sen voi tehdä hyvinkin monimutkaisesti. Pong-pelin ohella se on hyvin tyypillinen peli, jonka voi kehittää ensimmäiseksi peliksi. Se ei myöskään vaadi matemaattista osaamista, joka voi olla hyvinkin monelle kompastuskivi esimerkiksi tasoloikkapelin tekemisessä.

3.1 Suunnitteluvaihe

Pyrimme rajaamaan pelin suunnitteluvaiheessa mahdollisimman huolellisesti, jotta epäselvyyksiä ei tulisi kesken kehitysvaiheen. Näin voi tapahtua ja tapahtuukin monissa harrastusprojekteissa. Näin kävi myös meille, kun lähdimme alunperin suunnittelemaan roolipeliä. Pelistä tuli liian laaja ja me lisäsimme ominaisuuksia koko ajan lisää ja lisää, jolloin emme saaneet peliä koskaan valmiiksi. Ongelmana onkin hyvin monesti peliohjelmoijaharrastelijoilla se, että pelin kehittäminen jää puolitiehen. On siis hyvä lähteä suunnittelemaan alussa hyvinkin pientä ja huolellisesti rajattua peliä.

Vaikka kaaviot ja suunnitelmat tulevat muuttumaan yleensä projektin edetessä, suunnittelimme silti pelin tilakaavion hyvin korkealla tasolla etukäteen osaksi helpottaaksemme ohjelmointityötä, mutta osaksi myös hahmottaaksemme, kuinka paljon tiloja pelissä kokonaisuudessaan tulee olemaan. Seuraava kuvio esittää pelin tilakaaviota (Kuvio 5).



Kuvio 5. Pelin tilakaavio

Tilakaavion lisäksi teimme pienen listan siitä, mitä peli tulisi sisältämään ja mitä ei. Lista auttoi hahmottamaan, mitä piti vielä tehdä projektiin kussakin kohtaa. Listaa voisi kutsua myös pelin vaatimusmäärittelyksi (Kuvio 6).

- Peli on 640 pikseliä leveä ja 480 pikseliä korkea eikä peli ole skaalautuva.
- Peli tukee ikkuna- ja FSEM-tiloja.
- Pelissä on tasoja (layer), joista vähintään kaksi on vieriviä tasoja, kaksi sprite-tasoa, objektitaso ja hud-taso.
- Pelissä on yksi ase, jonka saa päivitettyä.
- Pelaajalla on käytettävissä kilpi, joka kuluu, kun pelaaja ottaa iskua vastaan.
- Peli käyttää ääniefektejä, kun pelaaja kerää power-uppeja, ampuu tai tuhoutuu. Äänitiedostot ovat WAV-tiedostoja.
- Pelaaja voi liikkua kahdeksaan eri suuntaan.
- Viholliset syntyvät erilaisissa muodoissa.
- Pelissä on valikot: about, help ja highscore.
- Pelissä ei ole pysähdys-tilaa (Pause).

Kuvio 6. Vaatimusmäärittely.

3.2 Toteutusvaihe

3.2.1 Pohja

Aluksi teimme pohjan pelille ja loimme tarvittavat luokat, joita käytetään globaalisti eri luokissa kuten Config-luokka. Config-luokka sisältää vain pysyviä, julkisia ja staattisia muuttujia. Niitä käytetään eri paikoissa eri tarkoituksiin. Esimerkiksi pelin FPS on määritelty siellä yhtenä muuttujana. Teimme pelille ensimmäiseksi tyhjän JFrame-ikkunan, jota sitten päivitettiin projektin edetessä. Loimme sinne tuen kummallekin näyttötilalle. JFramen lisäksi ohjelmoimme joitakin perusluokkia, joita käytetään eri yhteyksissä kuten Sprite-, AnimatedSprite- ja Font-luokat.

Sprite-luokalla on vain yksi tietojäsen ja se on BufferedImage. Erona BufferedImageen on se, että Sprite-luokka sisältää valmiiksi erilaisia piirtometodeita, joita hyödynnetään piirrettäessä kuvia. AnimatedSprite vastaavasti esittää animoitua Spriteä eli animoitua kuvaa. Se periytyy Sprite-luokasta ja sen tärkeimpiä tietojäseniä ovat lähdesuorakulmiot (source rectangles), kehysnopeus (FPS) ja tieto siitä onko animaatio kiertänyt kaikki kuvat läpi. Lähdesuorakulmioita hyödynnämme animoinnissa siten, että niiden avulla piirretään vain tietty alue kuvasta (Katso kappale 2.4). Kun animaatio etenee, vaihdetaan vain paikkaa lähdekuvasta.

AnimatedSprite-luokka toteuttaa Cloneable-rajapinnan, jotta AnimatedSprite-objekteja pystytään kloonaamaan. Tämä oli vättämätöntä, koska muuten sama animaatio pyörisi kaikilla objekteilla. Jos kaksi vihollista tuhoutuisi lähes samanaikaisesti, jälkimmäinen animaatio aloittaisi siitä mihin edellisen vihollisen animaatio jäi. Kloonauksen avulla saimme kaikille vihollisille omat tuhoutumisanimaatiot.

Font-luokka käyttää hyväksi Spriteä. Sprite eli kuva sisältää kaikki kirjaimet, numerot ja joitain erikoismerkkejä. Ideana on tallentaa java.util.Hashtable-listaan jokaista merkkiä vastaava suorakulmio ja käyttää niitä piirtämiseen. Javasta löytyy valmiiksi luokka java.awt.Rectangle, joka ottaa vastaan sijainnin ja koon. Niiden tietojen avulla voidaan piirtää lähdekuvasta eli Spritestä tietty osa ruudulle, ja tässä tapauksessa se on jokin merkki.

Seuraava kuva havainnollistaa, kuinka lähdesuorakulmioiden avulla piirrretään lähdekuvasta vain osa ruudulle (Kuvio 7). Perusperiaate on lähes sama kuin animaatioiden piirtämisessä. Erona merkkien piirtämiseen on se, että kohdesijainti pysyy samana.

Kuvio 7. Havainnollistava kuva kirjainten ja numeroiden piirtämisestä lähdekuvasta

Omien kirjasinten käyttö ei suinkaan ole pakollista, koska Javasta löytyy jo valmiiksi Graphics2D-luokasta metodi drawString(...), joka hoitaa saman asian valmiiksi valitulla järjestelmäkirjasimella. Pelin teeman ja persoonallisemman ilmeen takia on kuitenkin hyvä opetella käyttämään omia kirjasimia.

Merkkijononen piirtämiseen on monia erilaisia tapoja. Seuraavassa koodiesimerkissä on yksi tapa, jolla toteutimme sen (Koodiesimerkki 19). Metodin ideana on lukea merkkijonon jokainen merkki kerralla ja merkin avulla etsiä listasta oikea suorakulmio eli sijainti ja koko. Sen jälkeen kun oikea suorakulmio on löytynyt listasta, piirretään sen avulla kyseinen alue lähdekuvasta ruudulle määrättyyn paikkaan.

```
public void drawString(Graphics2D g, String str, float x, float y,
int lettersPerRow) {

    // Muutetaan String-objekti char-taulukoksi.
    char[] c = str.toCharArray();
    int rows = 0;
    float xOffset = 0.0f;

    // Luetaan kaikki merkit taulukosta
    // ja haetaan merkin sijainti
    // lähdekuvasta. Tämän jälkeen piirretään merkki ruudulle.
    for (int i = 0; i < c.length; i++) {
      char ch = c[i];

      if (i % (lettersPerRow + 1) == 0) {
            rows++;
            xOffset = i * spaceBetweenLetters + fontWidth;
      }

      Rectangle f = chars.get(ch);
      fontSheet.drawScaledImage(g, f.x, f.y, f.width,
          f.height,
          x + (spaceBetweenLetters * i) - xOffset,
          y + rows * spaceBetweenRows,
          f.width, f.height);
    }
}
```

Koodiesimerkki 19. Merkkijonojen piirtäminen

Javassa syötteiden käsitteleminen ei ole kovin tehokasta, koska se tapahtuu järjestelmän kautta. Normaalisti sulavan liikkeen aikaan saaminen ja monen napin painaminen yhtä aikaa on monen ylimääräisen muuttujan takana, koska jos vaihdat suoraan objektin nopeutta tai sijaintia keyPressed(...)-metodissa, objekti lähtee liikkumaan hieman nykivästi. Tämä johtuu siitä, kun käsky menee järjestelmän kautta, niin sen nopeus ei välttämättä vastaa pelin päivitysnopeutta. Yksi tapa kiertää tämä, on käyttää esimerkiksi boolean-tyyppisiä muuttujia. Päivitysmetodissa tarkistetaan boolean-muuttujan arvo ja muutetaan objektin nopeutta sen mukaan. Toinen vaihtoehto on käyttää omaa syötteiden hallintaa, johon me myös päädyimme. Toteutuksessamme käsittelemme suoraan syötteet pääsilmukan sisällä, jolloin ei tarvitse odottaa järjestelmän kautta kulkevaa tiedonsiirtoa. Jotta kaikki tämä olisi mahdollista, teimme KeyInputListener-luokan, joka periytyy abstraktista KeyAdapter-luokasta. Toteutimme sinne keyPressed(...)- ja keyReleased(...)-metodit, joiden avulla käsittelemme boolean-tyyppistä taulukkoa. Luokan yksinkertaisuus näkyy seuraavassa koodiesimerkissä (Koodiesimerkki 20).

```
public class KeyInputHandler extends KeyAdapter {

    // Taulukko, johon kunkin näppäimen tila tallennetaan.
    private boolean[] keyArray = new boolean[256];

    // Edellisen painetun näppäimen arvo.
    private int oldKeyCode = 0;

    // Metodi tarkistaa onko jokin näppäin painettu
    // pohjaan.
    public boolean isHoldDown(int key) {
        return keyArray[key];
    }

    // Tarkistaa onko jokin näppäin painettu.
    // Edellinen näppäin ei saa olla sama kuin uusi painettu.
    public boolean isPressed(int key) {
        if (keyArray[key] && oldKeyCode != key) {
            oldKeyCode = key;
            return true;
        }
        return false;
    }

    @Override
    public void keyPressed(KeyEvent e) {
        // Jokaisella näppäimellä on uniikki numero.
        // Kyseisen numeroa vastaavan taulukon alkion
        // arvo muutetaan arvoksi ”true”.
        keyArray[e.getKeyCode()] = true;
    }

    @Override
    public void keyReleased(KeyEvent e) {
        // Sama kuin keyPressed(...)-metodissa, mutta
        // arvo asetetaan ”false”:ksi ja palautetaan oldKeyCode
        // arvoon 0.
        keyArray[e.getKeyCode()] = false;
        oldKeyCode = 0;
    }
}
```

Koodiesimerkki 20. KeyInputHandler-luokka

3.2.2 Vierivät taustat ja lentävät objektit

Teimme peliin pari taustaa, jotka vierivät tietyllä nopeudella. Taustat ovat kooltaan 640 x 480 pikseliä. Yksi vaihtoehto olisi ollut tehdä taustat osista (tiilistä), mutta koska resoluutio ei ollut suuri, teimme taustat kokonaisista kuvista. Isot kuvat vievät enemmän muistia, mutta nykyään koneet ovat paljon tehokkaampia ja samanlaisia ongelmia muis-

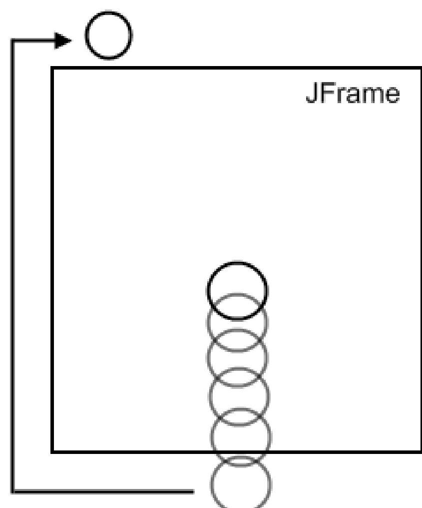
tin kanssa ei tule – ainakaan niin helposti. Tiiliä käyttämällä piirretään samoja kohtia kuvasta useita eri paikkaan eli tarvitaan huomattavasti vähemmän grafiikkaa ruudun täyttämiseen. Taustoja varten loimme Background-nimisen luokan, joka toteuttaa logiikan kuvan vierimiseen.

Teimme taustojen vierimisen hyvin yksinkertaisesti. Tausta piirretään ruudulle ja sitä liikutetaan ylhäältä alas. Heti taustan liikkuessa piirretään toinen kuva piirtymään ylhäältä alas siten, että se seuraa edellisen kuvan perässä. Seuraava koodiesimerkki havainnollistaa idean (Koodiesimerkki 21).

```
// Piirtäminen
if(velocity.y > 0){

    if(position.y < Config.SCREEN_HEIGHT) {
        sprite.draw(g, position.x, position.y);
    }

    sprite.draw(g, position.x,
        position.y - sprite.getHeight());
}

// Päivittäminen
position.x += velocity.x;
position.y += velocity.y;

if(velocity.y > 0) {
    position.y %= sprite.getHeight();
}
```

Koodiesimerkki 21. Vierivä tausta

Vierivien taustojen lisäksi teimme kaksi tasoa, joihin sijoitimme liikkumaan erilaisia kuvia. Niiden luomiseen teimme luokat BackgroundSprites ja FlyingSprite. BackgroundSprites-luokan tarkoitus on vain päivittää ja hallita FlyingSprite-objekteja eli käytännössä se sisältää listan FlyingSprite-objekteista. Kun FlyingSprite siirtyy ruudusta ulos, se alustetaan ruudun yläkulmaan satunnaiseen paikkaan (Kuvio 8).



Kuvio 8. Lentävän objektin lentoreitti

3.2.3 Pelaaja ja viholliset

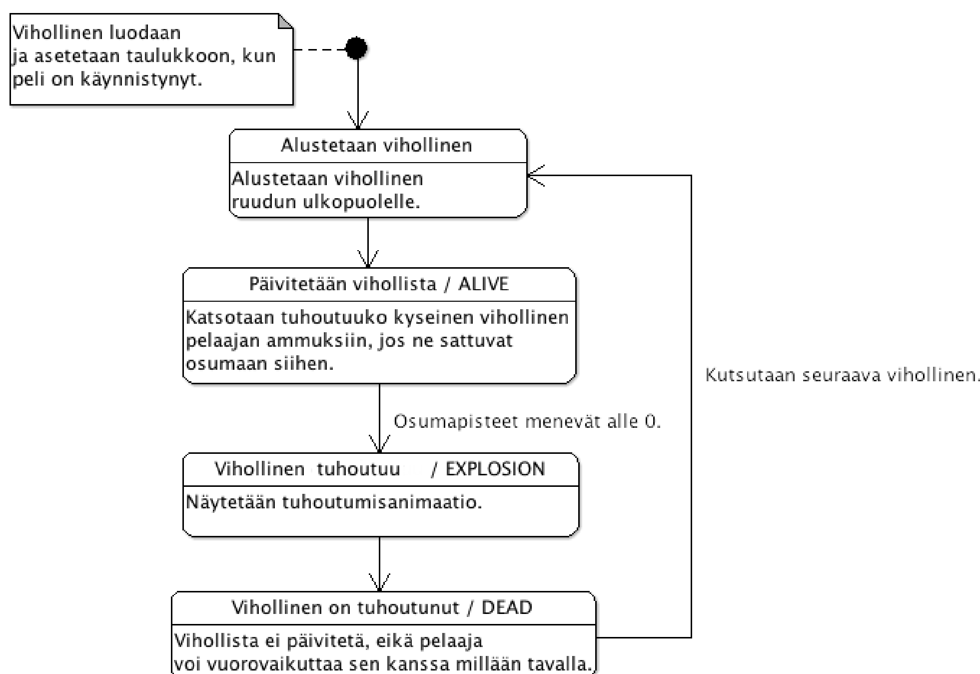
Pelaaja ja viholliset rakensimme periyttämällä ne suoraan Entity-luokasta. Entity-luokan lisäksi viholliset toteuttavat myös Enemy-rajapinnan. Alunperin Entity-luokka oli sama kuin pelaaja. Muutimme luokan myöhemmin abstraktiksi, jotta sitä pystyttiin hyödyntämään myös vihollisten luomisessa. Tämä muutos helpotti ohjelmointia, koska ei tarvinnut ohjelmoida samoja asioita moneen kertaan. Entity-luokka sisältää esimerkiksi tilat, jotka entiteetti voi saada. Tilat ovat ”ALIVE”, ”EXPLOSION” ja ”DEAD”. Luokka sisältää logiikan sille, kuinka alukset liikkuvat ja reagoivat erilaisten tilojen muutoksiin. Aseet, sijainti, nopeus ja törmäyksiin tarkoitettu suorakulmio ovat myös tärkeimpiä Entity-luokan tietojäseniä. Entity-luokan tarkoitus on olla valmis pohja pelaajalle sekä viholliselle.

Enemy-rajapinta sisältää muutaman abstraktin metodin, jotka vihollisen pitää toteuttaa kuten pisteiden palauttamisen, piirtometodin ja päivitysmetodin. Vaikka Entity-luokka sisältää piirto- ja päivitysmetodit, oli hyvä tehdä erikseen Enemy-rajapinta, koska tällöin voidaan sijoittaa kaikki viholliset samaan listaan ja lukijan on helpompi lukea ohjelmakoodia. Välttämätön Enemy-rajapinta ei kuitenkaan ole.

Pelaaja siis periytyy Entity-luokasta ja sillä on Entity-luokan lisäksi omia ominaisuuksia kuten kilpi, jota viholliset eivät omaa. Kilven toiminta, pelaajan pisteet ja kontrollit ovat myös pelaajan omia ominaisuuksia. Vastaavasti vihollisilla on omat logiikkansa siihen, miten ne liikkuvat ja ampuvat. Kaikille yhteistä on alustusmetodin uudelleen määrittely.

Kyseinen alustusmetodi on kätevä erilaisissa tilanteissa, kuten uusien vihollisten aktivoimisessa.

Vihollisia ei todellisuudessa luoda missään vaiheessa kesken peliä. Kaikki viholliset luodaan, kun peli ladataan. Tämä takaa sen, että JVM:n roskien kerääjä (garbage collector) ei tule ylimääräistä työtä. Tälläinen rasitus on helposti huomattavissa, sillä peli pätkisi ajoittain. Käytännössä aktivoiminen tässä tarkoittaa sitä, että pelissä on taulukko vihollisista, joilla on tietynlainen tila. Taulukko käydään ohjelmakoodissa lävitse ja samalla katsotaan jokaisen taulukon alkion eli vihollisen tila. Jos sen tila on ”DEAD” , alustetaan se tiettyyn paikkaan ja asetetaan tilaksi ”ALIVE”. Kun taas vihollinen tuhotaan, tila vaihdetaan arvoksi ”EXPLOSION” ja tämä tilan vaihtuminen tapahtuu sitten, kun vihollisen osumapisteet menevät alle arvon nolla. ”EXPLOSION” tilan aktivoiduttua näytetään aluksen räjähdys ja sen jälkeen vihollisen tila vaihdetaan arvoksi ”DEAD”, jolloin sitä ei näy ruudulla, eikä pelaaja pysty olemaan vuorovaikutuksessa siihen millään tavalla. Seuraava tilakaavio havainnollistaa asian (Kuvio 9).



Kuvio 9. Vihollisen tilakaavio

Peli siis koostuu pelaajasta, vihollisista ja power-upeista, joista kerromme myöhemmin lisää. Pelaajasta on yksi instanssi koko pelissä toisin kuin vihollisia, joita on aluksi alustettu useita jokaista vihollistyyppiä. Peliruutu on jaettu osiin siten, että vihollisia mahtuu ruudulle tietty määrä. Ohjelmakoodissa tämä tarkoittaa taulukkoa. Taulukkoa käytetään hyväksi, kun uusia vihollisia alustetaan ruudulle. Tämä myös auttaa erilaisten muotojen luomisessa. Taulukko on ohjelmakoodissa kaksiulotteinen (Koodiesimerkki 22). Viholliset on numeroitu id:n perusteella. Taulukon arvot vastaavat vihollisten id:tä. Kun uusia vihollisia alustetaan ruudulle, käydään läpi taulukon alkiot ja sen mukaan alustetaan tietynlaisia vihollisia id:n perusteella.

```
public static final int[][] PATTERNS = {
        {0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0,
         1, 0, 0, 0, 0, 0, 1,
         0, 2, 2, 2, 2, 2, 0},

          ...
};
```

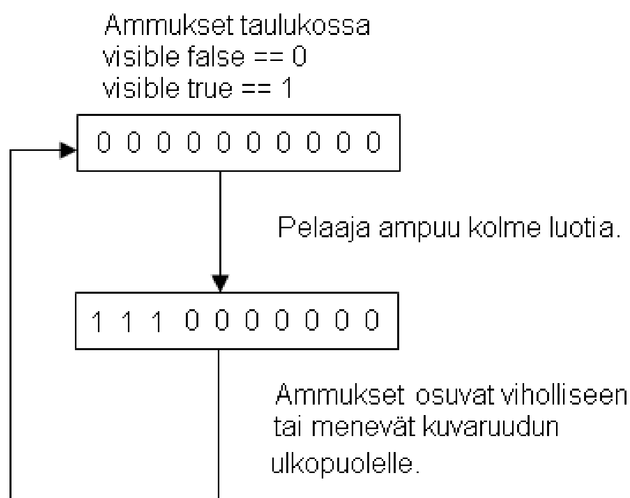
Koodiesimerkki 22. Vihollisten alustamiseen käytetty kaksiulotteinen taulukko

3.2.4 Ampuminen

Ampumisen toteutimme tekemällä abstraktin luokan Weapon, josta sitten periytimme erilaisia räätälöityjä aseita vihollisille sekä pelaajalle. Abstraktiin luokkaan keräsimme kaikille aseille yhteiset ominaisuudet. Yhteisiä ominaisuuksia ovat esimerkiksi lista ammuksista. Lista sisältää Bullet-objekteja. Bullet esittää yhtä ammus-objektia, jotta lukuisten ammuksien käsitteleminen olisi helpompaa.

Bullet-luokan tyypillisiä ominaisuuksia ovat sijainti, törmäykseen tarkoitettu suorakulmio, tehokkuus ja näkyvyys. Jos näkyvyys on tosi, ammus päivitetään ja näytetään ruudulla. Jos näkyvyys on epätosi, ammusta ei päivitetä eikä piirretä ja tällöin se tulkitaan jo käytetyksi ammukseksi tai ammukseksi, joka on vielä varastossa. Tässä vaiheessa on hyvä huomata, että ammuksissa on käytetty lähes samanlaista tekniikkaa kuin vihollisten käsittelyssä. Aluksi emme tehneet toteutusta edellä mainitulla tavalla, vaan loimme luodit ajon aikana. Luotien luominen ajon aikana tuotti paljon rasitusta garbage collectorille. Yksittäinen luoti ei vienyt paljon muistia eikä sen siivoaminen muistista ollut silmin havaittavissa, mutta useiden luotien luominen lyhyellä aikavälillä aiheutti

pelissä ajoittaista pätkimistä. Seuraava kuvio (Kuvio 10) selventää hieman ammusten logiikkaa.



Kuvio 10. Ammusten logiikka

3.2.5 Törmäyksen tunnistaminen

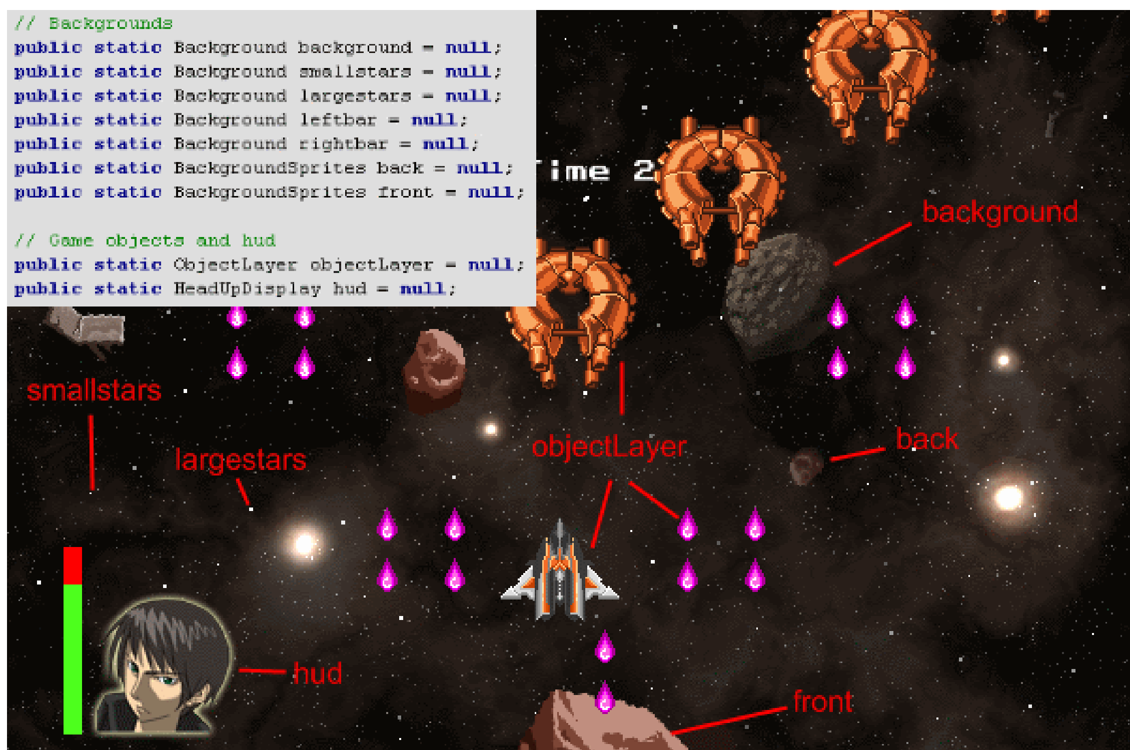
Törmäyksien tunnistaminen ja varsinkin niihin reagointi ovat varmasti yksi vaikeimmista, mutta tärkeimmistä asioista, koska ilman niitä on mahdotonta rakentaa toimivaa peliä. Nykyään löytyy ilmaisia fysiikkamoottoreita, joiden avulla voidaan hoitaa törmäykset helposti ilman matemaattista osaamista. Törmäyksen tunnistaminen tämänkaltaisessa pelissä on kuitenkin hyvin yksinkertaista. Monimutkaisia törmäyksentunnistamis-algoritmeja ei tarvita, vaan voidaan käyttää Javan java.awt.Rectangle-luokasta löytyvää intersects(...)-metodia. Metodi palauttaa boolean-tyyppisen arvon sen mukaan, leikkaavatko suorakulmiot toisensa. Metodi on hyvin yksinkertainen toteuttaa myös itse (Koodiesimerkki 22). Pääasiassa käytämme tätä törmäyksen tunnistamista vihollisten, pelaajan, power-uppien ja ammuksien tunnistamisessa. Katsotaan törmääkö esimerkiksi vihollinen ammukseen ja tehdään sen mukaan erilaisia asioita. Pelaajalla on erikseen myös törmäyksen tunnistukset ruudun reunoihin.

```
public boolean collideWith(float x, float y,
    float width, float height) {

    if (this.x < (x + width) &&
        this.y < (y + height) &&
        (this.x + this.width) > x &&
        (this.y + this.height) > y) {
      return true;
    }

    return false;
}
```

Koodiesimerkki 22. Törmäyksen tunnistaminen.

3.2.6 Pelin rakenne

Aluksi suunnittelimme käyttävämme listaa ikkunoiden hallintaan. Yksi ikkuna olisi sisältänyt yhden tason ja yksi taso olisi ollut esimerkiksi vierivä tausta, liikkuvat tausta-objektit -taso tai itse päätaso (ObjectLayer), jossa kaikki pelaajan ja vihollisten toiminta suoritetaan. Toteutusvaiheessa kuitenkin huomasimme, että ns. ikkunointi oli liian hankala systeemi ilman kunnollista perehtymistä. Ikkunat tallennettiin listaan ja listaan tehdyt muutokset kesken for-silmukan aiheuttivat paljon virheitä, joten päädyimme siihen tulokseen, että poistetaan koko systeemi ja sen sijaan käytetään suoraan tasoja ilman listan apua. Tasojen käyttö poisti paljon virheidentarkistuksia, mutta samalla lisäsi ohjelmakoodia. Tällä varmistettiin kuitenkin se, että peli ei kaadu kesken pelaamisen. Tasot projektissa ovat luokkia. Alla on havainnollistava kuva tasoista (Kuvio 11).



Kuvio 11. Tasot

Rakensimme pelin kahteen osaan: itse peliin ja valikoihin. Valikkoina toteutimme high score, about ja help. Alunperin ne olivat erikseen ikkunoissa. Oli paljon helpompi ohjelmoida kaikki samaan luokkaan. Itse peli sisältää joukon tasoja ja yhden tilan. Suurin osa tasoista on taustatasoja, mutta niiden lisäksi peli sisältää HUD:n (Head Up Display) ja objektitason (objectLayer). Taso objectLayer on pelin ydin, jossa kaikki pelin erilaiset toiminnot kuten törmäyksien tunnistamiset, aluksien liikuttaminen ja pisteiden lasku toteutetaan. Valikot sisältävät lähes kaikki samat tasot kuin itse pelikin lukuunottamatta HUD- ja objectLayer-tasoa.

Ylin luokka ennen JFrame-ikkunaa on GameManager, joka toimii pelin ja valikoiden hallinnoijana. GameManagerin päätehtävänä on ladata pelin käynnistyessä pelin resurssit (joista puhumme seuraavassa kappaleessa tarkemmin), piirtäminen ja aktiivisena olevan tilan päivittäminen. GameManager huolehtii myös tilojen välillä tapahtuvasta siirtymäefektistä. Efekti käynnistetään aina, kun tila vaihtuu. GameManager sisältää useita tiloja. Menu ja Gameplay ovat niistä tärkeimmät. Muut tilat liittyvät siirtymäefektin eri vaiheisiin. Tilat on rakennettu käyttäen Enum-tietorakennetta hyväksi. Toinen tila koskee valikoita ja toinen itse peliä. Valikko sisältää alavalikoita, joista mainittakoon Highscore-, About- ja Help-valikot. Alavalikkojen tilat on myöskin Game-

Managerin tilojen tapaan rakennettu Enum-rakenteella. Gameplay ei sisällä minkäänlaisia tiloja, vaikka yksi perustiloista olisi voinut olla tauko-tila (Pause), jota emme kuitenkaan alkuperäisen suunnitelman mukaan toteuttaneet.

Resurssien lataaminen tapahtuu Content- ja ContentLoader-luokkien avulla. Content-luokka sisältää kaikki suurimmat objektit staattisina tietojäseninään (esim. edellä mainitut tasot). ContentLoader vastaavasti sisältää metodit, joilla luodaan objektit Content-luokan tietojäsenistä. Käytännössä se tarkoittaa sitä, että kaikki resurssit, kuten äänitiedostot ja kuvat, ladataan niissä metodeissa. ContentLoader-luokan metodeita kutsutaan GameManager-luokan konstruktorissa. Koska peli ei ole resurssien puolesta kovin suuri, teimme resurssien latauksen ilman minkäänlaista latausmittaria. Resurssien lataaminen tapahtuu, kun peli käynnistetään.

3.2.7 Lisävoimat ja käytetyt matemaattiset kaavat

Lisävoimien (power-up), kuten suojan ja ammuksien päivityksien, luomiseen teimme graafiset ikonit ja Powerup-luokan, jolta löytyy ominaisuutena Sprite, id-numero, sijainti, näkyvyys, nopeus ja suunta. Kaikki pelin lisävoimat käyttävät samaa luokkaa, koska ne eivät eroa toisistaan mitenkään muuten kuin ikonin ja id-numeron avulla. Lisävoimien logiikan toteuttamisessa käytämme sen id-numeroa.

Käytämme hyväksi pelissä muutamaa matemaattista kaavaa useassakin kohdassa. Lineaarinen interpolointi on yksi näistä funktioista (Koodiesimerkki 24). Sillä tuotetaan todella sulavaa liikettä. Käytämme sitä siirtymäefektissä ja teksteissä, jotka tulevat ennen pelin alkua ja pelin loputtua. Se on todella kätevä funktio, jolle on paljon käyttökohteita. Toinen metodi, jota käytettiin, oli clamp(...)-metodi eli suomeksi puristin-metodi (Koodiesimerkki 24). Sen ideana on rajata arvo tietylle välille.

```
// Lineaarinen interpolointi
public static float lerp(float t, float a, float b) {
    return a + t * (b-a);
}

// Rajaa arvon tietylle välille.
public static float clamp(float value, float min, float max) {
    if (value > max) return max;
    if (value < min) return min;

    return value;
}
```

Koodiesimerkki 24. Lineaarinen interpolointi - ja clamp-metodit

3.2.8 Äänet

Päädyimme käyttämään pelissä ääniefekteinä pieniä WAV-tiedostoja. Minkäänlaista taustamusiikkia emme käyttäneet, koska Java ei tue MP3- tai OGG-tiedostoja suoraan. Käyttämämme taustamusiikki WAV-formaattina olisi ollut liian suuri. WAV-tiedostojen soittamiseen teimme ClipPlayer-luokan. Valitsimme Clip-objektien käytön DataSourceLinen sijaan, koska käyttämämme WAV-tiedostot ovat kooltaan pieniä, joten ne mahtuvat hyvin valmiiksi ladattuna muistiin. ClipPlayer on käytännössä luokka, jossa on tietojäsenenä Clip-objekti ja silmukoiden lukumäärä eli kuinka monta kertaa kappale pyöritetään alusta loppuun. Sen toteutus ei eroa juurikaan siitä, mitä tämän dokumentin äänet-kappaleessa (katso 3.2.8) esitettiin. ClipPlayerin start()- ja stop()-metodeita kutsutaan tarpeen vaatiessa: esimerkiksi jos vihollinen tuhoutuu, pelaaja ottaa lisävoiman tai pelaaja ampuu.

3.2.9 Tuloksien tallentaminen tiedostoon

Teimme pelin pisteiden tallennukseen Stats-nimisen luokan. Pisteet tallennetaan taulukkoon. Taulukko koostuu kymmenestä parhaasta pistemäärästä. Tallennus tapahtuu, kun siirrytään pelistä takaisin valikoihin. Pistemäärät tallennetaan binäärimuodossa tiedostoon. Pistetaulukko ladataan pelin käynnistyessä ja piirretään Highscore-valikossa.

3.3 Lopputulos

Lopputuotoksena syntyi avaruusräiskintäpeli. Peli toimii niinkuin pitää, mutta sen rakenne jäi hiukan sekavaksi. Luokkakaaviota siitä oli hyvin hankala tehdä, koska jotkut luokat eivät olleet tarkasti määriteltyä ja suunnitteltuja. Luokkien suunnittelu etukäteen olisi ollut kuitenkin hieman hankalaa, koska se olisi vaatinut enemmän aikaa ja paneutumista. Aikaa ei ollut, ja toisekseen pelin rakenne olisi varmasti muuttunut huolimatta siitä, oltaisiinko luokkakaavio tehty vai ei. Aika ei olisi myöskään riittänyt sen päivittämiseen. Pääasia on kuitenkin se, että peli toimii. Minkäänlaista valmiiksi suunniteltua kehystä tai moottoria emme lähteneet tekemään. Sen aika tulee myöhemmin kokemuksen ja ajan myötä. Tärkeimmät luokat kuitenkin selitimme tässä dokumentissa melko korkealla tasolla, joten jonkinlaisen kuvan pelin rakenteesta toivottovasti pystyimme välittämään.

4 Yhteenveto

Opinnäytetyön tuloksena syntyivät tämä dokumentti ja Java-peli itsellemme harjoitukseksi. Dokumentin alussa tutkimme, kuinka Javassa luodaan pohja pelille, koska se on hyvin monelle Javasta kiinnostuneille peliohjelmointia harrastaville epäselvää. Tämän ongelmakohdan pyrimme mahdollisimman selkeästi selventämään. Pyrimme tekemään kokonaisuudesta tiiviin paketin, joka kuitenkin kattaisi kaikki tarpeelliset osa-alueet, joita pelinkehittämisessä Java-alustalla tarvittaisiin. Näihin osa-alueisiin kuuluivat ikkunan luominen, grafiikan piirtäminen ruudulle, syötteiden prosessointi, pääsilmukan luominen ja äänien toistaminen. Loppujen lopuksi dokumentista tuli mielestämme ytimekäs paketti ja ohjeiden avulla on helppo päästä alkuun. Joitain ratkaisemattomia ongelmia pelin pohjaan jäi, kuten Mac OS X:llä äänien toistaminen ei tuntunut toimivan dokumentissa esitetyllä tavalla.

Opinnäytetyön toinen tuotos, ja se meille henkilökohtaisesti tärkeämpi oli itse Java-peli. Työprosessi oli odotettua pitkäkestoisempi, mutta saimme sen lähes valmiiksi. Pelejä voi aina kehittää eteenpäin, harvoin ne ovat täysin valmiita. Meiltä jäi muutama ominaisuus toteuttamatta peliin. Peli toimii, mutta sen lopullinen toteutus ei kuitenkaan meitä tyydyttänyt. Jotkin ratkaisut jäivät kaivelemaan ja epäloogisuuksia jäi luokkien rakenteisiin. Kehitettävää jäi paljon, vaikka kokemusta on jo jonkun verran takana.

Työn kirjallinen osuus jäi osittain puutteelliseksi, koska teimme pelin rakenteesta liian mutkikkaan kirjallisesti ilmaistavaksi. Teimme pelissä myös yhden perinteisen virheen, kun lisäsimme ominaisuuksia kesken pelin kehittämisen. Kaikki ominaisuudet, jotka alunperin suunnittelimme paperille, olisivat riittäneet hyvin. Toisekseen kirjoittaminen pelin työosuudesta oli vaikeampaa kuin alunperin olimme osanneet odottaa. Oli vaikea päättää, mitä asioita kannattaisi ottaa huomioon ja mitä ei. Jos kaiken olisimme kirjoittaneet ja näyttäneet, dokumentista olisi tullut liian suuri, eikä siitä silloin olisi tullut tarpeeksi ytimekäs paketti. Lukijan kannalta tämä tulos on harmittava, mutta pääasia on kuitenkin se, että annoimme ideoita pelin kehittämiseen.

Javan omista kirjastoista jäi mieleemme joitakin asioita. Java 2D API oli hyvä, mutta ei täydellinen. Sen suorituskyky vaihteli alustalta alustalle. Esimerkiksi Mac OS X:ssä suorituskyky ilman minkäänlaisia säätöjä oli kehnonlainen. Java 2D API kuitenkin riittää monenlaiseen tarpeeseen, mutta jos haluaa tehdä pelistään aidosti alustariippumatto-

man ja suorituskyvyltään erinomaisen, kannattaa kokeilla esim. JOGL:ää tai LWJGL:ää. JOGL on OpenGL-binding Javalle ja vastaavasti LWJGL on valmis Java-pelikirjasto. LWJGL sisältää valmiiksi OpenGL:n, OpenAL:n ja tuen erilaisille peliohjaimille. OpenGL on hyvin yleisesti käytetty rajapinta graafisiin toimintoihin, ja vastaavasti OpenAL on rajapinta ääniohjelmointiin.

Javan äänipuoli tuntui hyvin bugiselta ja näin jälkeenpäin voimme perustellusti sanoa, että Java Sound APIin ei kannata koskea. Sen sijaan kannattaa katse kääntää mieluummin Java OpenAL:n (JOAL) suuntaan. OpenAL on alustariippumaton 3D-ääni-API, joka on tarkoitettu pääasiassa peleille. Esimerkiksi edellä mainittu pelikirjasto LWJGL sisältää valmiiksi OpenAL:n.

Tässä opinnäytetyössä emme perehtyneet fysiikkaan, vaikka fysiikka on nykyään yksi tärkeimpiä elementtejä peleissä. Kunnollisen fysiikkamoottorin luominen vaatii kuitenkin matemaattista osaamista ja aikaa. Jos törmäyksien tunnistamisiin matemaattiset lahjat eivät taivu, kannattaa tutustua valmiisiin kirjastoihin. Phys2D, JBox2D ja JBullet ovat kaikki hyviä ehdokkaita.

Kokonaisuudessaan opinnäytetyön tekeminen oli kuitenkin meille positiivinen kokemus. Opimme projektin edetessä paljon erilaisia pelin toteutukseen liittyviä asioita, mm. miten joitain asioita kannattaa tehdä ja mitkä vastaavasti olivat huonoja ratkaisuja pelin rakenteen kannalta. Opimme tekemään yksinkertaisemmin syötteiden hallinnan taulukon avulla, minkä alunperin teimme listalla. Opimme pieniä kikkoja, joilla saimme peliin sulavuutta. Yksi tärkeimpiä oivalluksia oli myös se, että silmukassa ei kannata luoda tai tuhota objekteja jatkuvasti, vaan kannattaa luoda ne hyvissä ajoin ja käyttää niitä tehokkaasti hyväksi jatkossa. Opimme myös sen, että pelin suunnitteludokumentti on hyvin tärkeä osa pelin suunnittelun kannalta.

Lopuksi voimme antaa vielä yhden neuvon lukijalle: Älä lähde tekemään pelimoottoria, vaan tee itse peliä. On suurempi todennäköisyys, että peli jää kesken, jos mietit vain pelimoottoria. Pelimoottori kehittyy samalla, kun teet peliä. Huomaat peliprojekteissa, mitkä asiat eivät toimineet, niin kuin niiden piti toimia ja mitkä taas toimivat. Kannattaa kerätä vanhoista peliprojekteista luokkia talteen ja kehittää niitä eteenpäin. Myös aivan hyvä vaihtoehto on käyttää valmista pelimottooria.

Lähteet

Brackeen, David 1996. 256-Color VGA Programming in C: Double-Buffering, Page-flipping, & Unchained Mode [online] [viitattu 12.01.2009] http://www.brackeen.com/vga/unchain.html

Clingman, Dustin 2004. Practical Java Game Programming. [online] [viitattu 14.10.2008] [salasana]. http://site.ebrary.com/lib/tamperepoly

Davison, Andrew 2005. Killer Game Programming in Java. O'Reilly. [online viitattu 5.11.2008]. ISBN 0-596-00730-2. http://fivedots.coe.psu.ac.th/~ad/jg/

Fletcher, Josh 2001. AWT vs Swing. [online] [viitattu 24.10]. http://dn.codegear.com/article/26970

Hester, Josiah 2007. Ultimate Java Image Manipulation. [online] [viitattu 01.01.2009]. http://www.javalobby.org/articles/ultimate-image/

Lesson: Concurrency: Deadlock 2006. [online] [viitattu 28.01.2009]. http://java.sun.com/docs/books/tutorial/essential/concurrency/deadlock.html

Lesson: Concurrency: Defining and Starting a Thread 2006. [online] [viitattu 20.12.2008]. http://java.sun.com/docs/books/tutorial/essential/concurrency/runthread.html

Lesson: Using Swing Components: How to Make Frames (Main Windows) 1997. [online päivitetty 30.04.2007] [viitattu 01.10.2008]. http://java.sun.com/docs/books/tutorial/uiswing/components/frame.html

Java™ Platform, Standard Edition 6 API Specification: KeyListener 2006. [online] [viitattu 09.11.2008]. http://java.sun.com/javase/6/docs/api

Java Sound Programmer Guide: Chapter 4: Playing Back Audio: Setting Up the SourceDataLine for Playback 2000. [online] [pdf] [viitattu 28.01.2009]. http://java.sun.com/j2se/1.4/pdf/sound-dev-guide-1.4.2.pdf

Martak, Michael 2001, Lesson: Full-Screen Exclusive Mode API. [online] [päivitetty 04.03.2002] [viitattu 03.10.2008]. http://java.sun.com/docs/books/tutorial/extra/fullscreen/index.html

Wikipedia: Hardware Acceleration 2008. [online] [viitattu 2.01.2009]. http://en.wikipedia.org/wiki/Hardware_acceleration