Tommi Lassila

Pelimoottorin suunnittelu ja toteutus

Metropolia Ammattikorkeakoulu
Insinööri (AMK)
Tietotekniikka
Insinöörityö
16.4.2012

Metropolia

Tiivistelmä

Tekijä Otsikko Sivumäärä Aika	Tommi Lassila Pelimoottorin suunnittelu ja toteutus 45 sivua + 16 liitettä 16.4.2012
Tutkinto	insinööri (AMK)
Koulutusohjelma	tietotekniikka
Suuntautumisvaihtoehto	ohjelmistotekniikka
Ohjaaja	lehtori Jukka Juslin

Videopeliteollisuus on lähivuosien aikana kasvanut yhdeksi maailman suurimmaksi viihdeteollisuuden muodoksi. Sen myötä myös pelimoottorien myynti on kasvanut.

Tämä työ selvittää pelimoottorin suunnittelun ja toteutuksen eri vaiheet ja tutkii mitkä ovat parhaimmat käytännöt ja mallit pelimoottorin toteutuksessa.

Ennen suunnittelu- ja toteutusosaa tutkittiin lyhyesti pelimoottorien historiaa, käytiin läpi yleisimpiä pelimoottoreita, minkä jälkeen selvitettiin motivaatiotekijät pelimoottorin teon taustalla.

Pelimoottorin suunnitteluosa aloitettiin määrittämällä moottorin tulevat ominaisuudet ja tekemällä niistä vaatimusmäärittely. Tämän jälkeen toteutettiin pelimoottorin toiminnallinen määrittely. Toiminnallisessa määrittelyssä suunniteltiin pelimoottorin järjestelmien toiminta alustavalla tasolla.
Toiminnallisen määrittelyn jälkeen suoritettiin tekninen määrittely- ja toteutusosa.
Teknisessä osassa suoritettiin työkalujen, komponenttien ja kirjastojen valinta, jonka jälkeen luotiin luokkakaaviot kaikille järjestelmille ja ohjelmistokomponenteille.
Toteutusosassa kirjoitettiin ohjelmisto pohjautuen teknisen sunnittelun luokkakaavioihin.

Lopuksi toteutettiin testipeli käyttäen tehtyä pelimoottoria.

Avainsanat	pelimoottori, ohjelmistotuotanto, c++

Metropolia

Abstract

Author Title Number of Pages Date	Tommi Lassila Design and development of a game engine 45 pages + 16 appendices 16 April 2012
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Development
Instructor	Jukka Juslin, Senior Lecturer

The video game industry has grown to be one of the largest entertaiment industry formats in the world. As a result, the demand for middleware software, such as game engines, has also grown. This study analyses game engine design and implementation stages, and also examines the best paractises and patterns in game engine implementation.

Before designing and implementation, the history of game engines and the most common game engines were studied and then the motivational factors behind game engine creation were examined.

Designing the game engine was started by determining the characteristics and requirements of the engine. This was followed by defining its functional specifications. In the functional specification the game engine systems operation was designed on the provisional level. After the functional design it was turn for technical specifications and coding. As for technical specifications tools and libraries were selected, which was followed by the creation of the class diagrams of the system, concluded by writing of the code.

Lastly, a test game was created by using the game engine.

Keywords	Game engine, software development, c++

Sisällys

1 Johdanto 1

2 Pelimoottorin määritelmä 2

3 Syyt pelimoottorin valmistukselle 4

4 Hyödyllisiä käytäntöjä, malleja ja työkaluja 6

4.1 Ketterät menetelmät pelimoottoria suunnitellessa 6
4.2 Visuaaliset apuvälineet ja työkalut 7
4.3 Suunnittelumallit 8
4.4 Ohjelmistotyökalut 10

5 Pelimoottorin suunnittelu ja toteutus 12

5.1 Vaatimusmäärittely 12
5.2 Toiminnallinen määrittely 15
5.2.1 Hahmonnusjärjestelmä 17
5.2.2 Syöttölaite- ja käyttöliittymäjärjestelmä 18
5.2.3 Äänentoistojärjestelmä 19
5.3 Tekninen määrittely ja toteutus 21
5.3.1 Ohjelmointikieli ja mallinnusperiaatteet 21
5.3.2 Työkalujen, komponenttien ja kirjastojen valinta 22
5.3.3 Hahmonnusjärjestelmän toteutus 23
5.3.4 Käyttöliittymäjärjestelmän toteutus 33
5.3.5 Äänentoistojärjestelmä 34
5.3.6 Ydinjärjestelmä 36

6 Testaus 40

7 Kokemukset 42

8 Yhteenveto 43

Lähteet 45

Liitteet

Liite 1. Renderer-luokka
Liite 2. SceneManager-luokka
Liite 3. Scene-luokka
Liite 4. FontManager-luokka
Liite 5. Drawable-luokka
Liite 6. Sprite-luokka
Liite 7. AnimatedSprite-luokka
Liite 8. UiComponent-luokka
Liite 9. Font-luokka
Liite 11. SoundManager-luokka
Liite 12. EventManager-luokka
Liite 13. EvenListener-luokka
Liite 14. FrameListener-luokka
Liite 15. Root-luokka
Liite 16. Config-lähdekoodi
Liite 17. Lisäkirjastot

1 Johdanto

Videopeliteollisuus on noussut yhdeksi maailman suurimmaksi viihdeteollisuuden muodoksi ja maailmanlaajuisesti se on jo ohittanut musiikki- sekä elokuvatallenteiden myynnin [1]. Videopeliteollisuuden kulta-aikana (n. 1980) myynti oli noin 5 miljardia dollaria maailmanlaajuisesti [2]. 30 vuodessa myynti on kasvanut noin 65 miljardiin dollariin [3], ja on arvioitu että se kasvaa 91 miljardiin dollariin vuoteen 2015 mennessä [4]. Yhdeksi merkittävimmäksi videopeliteollisuuden osa-alueeksi on muodostunut väliohjelmistojen (middleware) ja eritoten pelimoottorien (game-engine) myynti.

Pelien muodostuessa yhä hienoimmiksi, ja tämän myötä myös kalliimmiksi, tarvitaan kaikki mahdollinen aika itse pelin tekoon ja suunnitteluun. Ohjelmarungon tekeminen alusta asti jokaisen uuden pelin myötä on kuin keksisi pyörän aina uudelleen ja uudelleen. Tämän takia on järkevää tehdä pelimekaaniikasta mahdollisimman uudelleenkäytettävä moduuli. Kun peliä suunnitellaan, valitaan sille halutuilla ominaisuuksilla varustettu ohjelmistomoduuli, ohjelmarunko, joka hoitaa kaiken tarpeellisen grafiikan piirtämisestä fysiikan mallintamiseen. Kuten autoon, johon valitaan sille sopiva moottori, voidaan myös pelille valita sille oma ”pelimoottori”.

Tässä työssä selvitetään, miten suunnitella ja toteuttaa yksinkertainen pelimoottori. Tavoitteenani on tutkia ja selvittää mitä useimmat pelimoottorit pitävät sisällään, mitkä ovat parhaimmat käytännöt ja mitä ominaisuuksia sekä suunnitelumalleja pitäisi ottaa huomioon moottoria toteutettaessa. Sen lisäksi selvitän, miksi omaa pelimoottoria kannataa lähteä suunnittelemaan.

2 Pelimoottorin määritelmä

Pelimoottorilla (game engine) tarkoitetaan ohjelmistorunkoa, joka on tehty varta vasten videopelien luontia varten. Ohjelmistorunko useimmiten sisältää kaikki tarvitut moduulit peliä varten, kuten hahmottamisen, fysiikkamallinnuksen, tekoälyn ja audiojärjestelmän. Pelimoottoria ei kuitenkaan sekoittaa pelinteko-ohjelmiin, kuten Game Makeriin. Pelinteko-ohjelmissa ohjelmointi on koetettu jättää minimiin, kun taas pelimoottorin käyttö vaatii ohjelmointikokemusta.

Termi ”pelimoottori” nousi ensimmäisen kerran esille 1990-luvun puolivälissä, aikoina jolloin tietokonepeliyritys id Software julkaisi suositun Doom-pelisarjan ensimmäisen osan. Doom oli ensimmäisiä pelejä, jossa sen ohjelmistoarkkitehtuuri erotteli selvästi ydinohjelmiston (hahmonnus, törmäyksenhallinta ja audiojärjestelmä) ja pelillisen sisällön (grafiikka, pelikentät ja säännöt). Pelinsuunnittelijat huomasivat erottamisen arvon ja tämän myötä alkoivat lisensoida omia pelejä lisäämällä graafista sisältöä tai muuttamalla pelin sääntöjä kuitenkaan muuttamatta itse ”moottoria”. Tämä oli synty modauskulttuurille, joka toimii vieläkin faniprojektien myötä [4]. 1990-luvun loppua kohden muutamia pelejä, kuten Quake III Arena ja Unreal, oli varta vasten kehitelty uudelleenkäytettävyyttä ajatellen. Näitä pelejä pystyttiin muokkaamaan hyvinkin paljon komentosarjakielien eli skriptien avulla.[6.]

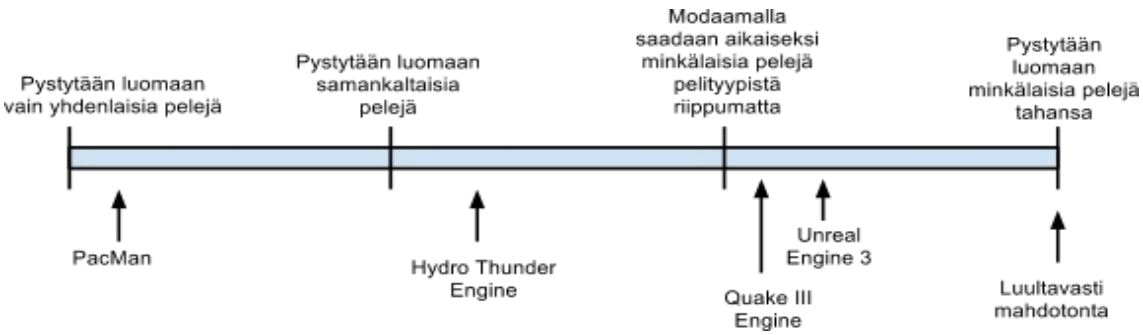
Pelimoottoriteknologian kehittymisen myötä aukesivat ovet myös jälleenmyyntimarkkinoille. Taloudellinen menestyminen ei enää riippunut vain pelien myynnistä vaan menestymään pystyi myös myymällä pelimoottoreita. Tämä taas auttoi pelinkehittäjiä, lisensoiduilla pelimoottorilla pystyttiin luomaan paremmin ja nopeammin pelejä ilman rautaista ohjelmointiosaamista.

Nykyään markkinoilla on monenlaista pelimoottoreita tarjolla:

- Torque 3D on hyvin varusteltu moottori, jolla pystytään kehittämään kaikenlaisia pelejä lajista riippumatta. Ominaisuuksiin kuuluvat mm. fysiikkamallinnus PhysX:n avulla, dynaaminen valaistus ja muokkausohjelma maailmojen luomista varten. Tuettuja alustoja ovat Wii, Xbox360, PC Windows, Mac ja Iphone. Lisenssi on maksullinen.
- Unity Engine on moottori, joka on kehitetty 3D-pelien kehittämistä silmällä pitäen. Ominaisuuksiin kuuluvat mm. dynaaminen valaistus, HDR, fysiikkamallinnus PhysX:n avulla, sisäänrakennettu reitinhakujärjestelmä ja moderni audiojärjestelmä. Tuettuja alustoja ovat Wii, Xbox 360, Playstation 3, PC Windows, Mac, Iphone, Android ja web-sovellukset. Riippuen käyttötarpeista lisenssi on joko maksullinen tai maksuton.
- C4 Engine on moottori, jolla pystyy luomaan 2D- sekä 3D-pelejä pelilajista riippumatta. Ominaisuuksiin kuuluvat mm. dynaaminen valaistus, fysiikkamallinnus, 3d-audiojärjestelmä ja vokselgrafiikkatuki. Lisenssi on maksullinen, ilmainen lisenssi on tarjolla opiskelijoille Yhdysvalloissa ja Kanadassa.
- ClanLib SDK on ilmainen ja avoimella lähdekoodilla varustettu pelimoottori, jolla pystyy tekemään 2D-pelejä. Tuettuja alustoja ovat PC Windows, Linux ja Mac. Lisenssi on ilmainen.

Ero pelin ja pelimoottorin välillä voi olla joskus hyvinkin marginaalinen. Jokin peli voi hahmontaa vain yhdenlaisia ”örkkejä” ,kun taas toisessa pelissä moottori määrittelee ”örkin” yleiset ominaisuudet, kuten materiaalin ja varjosäädökset, mutta itse renderoitava muoto piirretään ulkoisen datan mukaan. Luultavasti suurin ero pelin ja pelimoottorin välillä on moottorien arkkitehtuurisuunnittelu. Pelimoottorien arkkitehtuurisuunnittelu on useimmiten tehty mukautumaan ja toimimaan ulkoisen datan mukaan (Data-Driven Architecture). Kun pelimoottorin hahmonnusasetukset ja pelimaailma luetaan kaikki ulkoisesta datasta, saadaan moottorista monipuolisempi ja uudelleenkäytettävä. [6.]

Teoriassa pelimoottorin voi suunnitella niin että sillä pystyy tekemään kaikenlaisia pelejä kaikenlaisille eri alustoille. Tätä tosin ei ole vielä tehty eikä välttämättä pystytä edes tekemään. Niinpä useimmat pelimoottorit on suunniteltu toteuttamaan vain tietynlaisia pelejä äärimmäisen hyvin ja tehokkaasti tietynlaisille alustoille. Toki on myös moottoreita joilla pystyy toteuttamaan ammuntapelin sekä ajopelejä, mutta on turvallista sanoa ettei niillä saa yhtä hyvää jälkeä kuin moottorilla joka on varta vasten suunniteltu räiskintäpelejä taikka rallipelejä varten.



Kuva 1. Pelimoottorien uudelleenkäytettävyysasteikko. [6.]

Kuvan 1 mukaisesti voidaankin sanoa, että mitä yleiskäyttöisemmäksi moottori tehdään, sen vähemmän optimaallisempi sillä on tehdä tietynlaisia pelejä tietynlaisille alustoille muokkaamatta moottoria huomattavasti.

Nykyään pelinkehittäjät voivat lisensoida pelimoottoreja ja hyödyntää haluamansa osan moottorin komponenteista ja vaikka räätälöityjen ohjelmiskomponenttien teko on oleellisesti mukana työssä on silti moottorien käyttö ekonomisempaa kun tehdä kaikki itse alusta alkaen.[6.]

3 Syyt pelimoottorin valmistukselle

Syitä oman pelimoottorin tekemiseen voi olla monia syitä. Ehkä markkinoilta ei löydy vaatimuksia täyttävää pelimoottoria taikka sitten se voi olla liian kallis. Voi olla myös, että pelimoottorit ovat liian raskaasti ja monimutkaisesti toteutettuja yksinkertaisen "Space Invaders"-kopion toteutusta varten. Huippupelien pelimoottorit on hiottu

äärimmilleen; resurssit ladataan reaaliajassa pelaamisen ohella, grafiikka hahmonnetaan tasaisella 30–60 Hz:n taajuudella samalla kun fysiikkamallinnus hoitaa 250 objektin räjähdysimulaation. Kaikki ohjelmistokomponentit on suunniteltu toimimaan keskenään täydellisessä harmoniassa, jotta pelaajan tietokoneesta saataisiin käyttöön maksimaalinen laskentateho.

Yksinkertaiseen "avaruusräiskintäpeliin" tulisi kenties vain 16 kuvaa ilmaisemaan vihollisia sekä yksi ääniefekti ilmaisemaan ampumista. Tällaisessä tapauksessa huippumodernista pelimoottorista jouduttaisiin karsimaan niin paljon ominaisuuksia, että koko idea pelimoottorin käytöstä menettää merkityksen. Järkevämpää on luoda oma minimaalinen pelimoottori, joka hoitaa yksinkertaisen tehokkaasti kaikki tarvittavat toiminnot muutamien png-tiedostojen lataamisesta yksinkertaiseen MIDI-äänentoistoon.

Toinen syy voi olla pelkästään puhdas mielenkiinto sekä kokemus. Hienostuneen pelimoottorin suunnittelu ja teko on monimutkainen prosessi ja se vaati tekijältään tietämystä monelta ohjelmoinnin eri osa-alueilta:

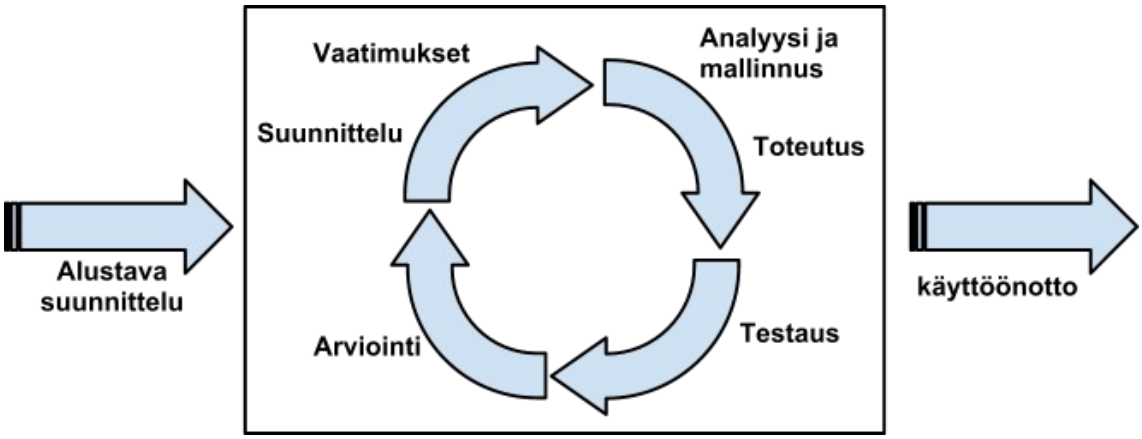
- 2D/3D-kuvamallennus
- fysiikkamallennus
- ääni- ja kuvamanipulaatio
- komentosarjakielen/skriptikielen kirjoitus
- animointi
- tekoäly
- tietokoneverkot
- muistin hallinta
- moniajo/säikeistys.

Riippuen moottorin monipuolisuudesta voi ominaisuuksien määrä kasvaa vielä tai kutistua. Sanomattakin on selvä, ettei yksinkertaiseen 2D-tasoloikkapeliin juuri tarvita verkkoyhteyttä, joten verkko-ominaisuudet voi huoleti jättää suunnittelematta pelimoottoriin.

4 Hyödyllisiä käytäntöjä, malleja ja työkaluja

4.1 Ketterät menetelmät pelimoottoria suunnitellessa

Olen huomannut omista projekteistani sen, että pelin tai pelimoottorin valmistus on dynaaminen prosessi siinä mielessä, että vaatimukset, ominaisuudet ja rajoitteet muuttuvat projektin edetessä. Ongelmia ilmenee ajan myötä eikä niitä olisi millään kyennyt ennustamaan projektin alkuvaiheissa. Jos projektin tuotantomenetelmäksi on valittu vesiputousmalli, voi ongelmien ratkominen olla vaikeaa, jopa mahdotonta riippuen siitä, kuinka pitkälle projekti on edennyt. Itse olen huomannut useimmiten, että ketterät menetelmät ovat paras malli peliä työstäessä. Iteratiivinen prosessi mahdollistaa sen, että projektin suuntaa voidaan muuttaa kesken prosessin. Ketterät menetelmät kannustavat myös modulaariseen ohjelmointiin, koska jokaisessa iteraatiosyklissä tuotetaan jotain uutta ja toimivaa.



Kuva 2. Iteratiivinen työskentelymalli.

Kuvan 2 mukaisesti jaan työskentelyni Scrumin tapaisesti pyrähdyksiin: aluksi määrittelen vaatimukset ja tavoitteet kyseiselle pyrähdyksille, jonka jälkeen suunnittelen ohjelmistokomponentit vaatimuksien pohjalta ja toteutan ne.

4.2 Visuaaliset apuvälineet ja työkalut

Olen huomannut töissäni, että piirroksien teko pelimoottorin vaatimuksia ja ominaisuuksia määritellessä on kannattavaa. Kuvat kertovat helpommin pelimoottorin visuaalisista ominaisuuksista kuin monisivuinen tekstidokumentti. Kuvitellaan tilanne, että suunnittelija on piirtänyt pelistä kuvan 3 mukaisen luonnoksen, joka kuvastaa pelin graafista sisältöä.

Kuva 3. Raptor: Call of the Shadows. [8.]

Sanonta "Kuva kertoo enemmän kuin tuhat sanaa" pätee myös tässä tapauksessa. Kuvasta pystyy esimerkiksi kertomaan, mikä on hahmonnuksen värialue ja ulottuvuus. Paljon enemmän työtä vaatisi tehdä konseptikuva pohjautuen vaatimuksiin, paitsi tietysti silloin, jos peli "tilataan" ja vaatimukset on jo ennalta määritelty. Kaikkia vaatimuksia ei tietenkään kuvien perusteella pysty tekemään, esimerkiksi minkälaisia ääniformaatteja pelimoottorin tulisi tukea.

On olemassa myös suunnittelua varten tehtyjä visuaalisia ohjelmistotuotannon työkaluja. Näitä ovat muun muassa UML ja CASE. UML on kokoelma yksinkertaisia graafisia merkintöjä, kuten laatikoita ja nuolia, joilla on määrätty merkitys. Jotkut UML-ohjelmat osaavat jopa kääntää luokkakaavion suoraan ohjelmoitavaan muotoon. Työssäni käytän UML-mallinnusta, koska henkilökohtaisesti miellän UML-kaaviot paremmin kuin tekstin.

4.3 Suunnittelumallit

Olio-ohjelmointi voi olla ajoittain hankalaa. Pitää keksiä oikeanlaiset ratkaisut tehtäville, muotoilla niistä luokat ja määritellä niille rajapinnat pitäen ne kuitenkin hallittavissa kokonaisuuksissa unohtamatta koodin uudelleenkäytettävyyttä. Isoimmissa projekteissa ratkaisujen löytäminen voi olla yllättävänkin hankalaa ja oikeaoppisen ohjelmointikäytännön ylläpitäminen hankalaa. Voi olla, että kahteen hyvin samankaltaiseen ongelmaan tehdään kaksi aivan erilaista ratkaisua. Suunnittelumallit ratkaisevat tämän ongelman esittämällä vastauksia oliopohjaisen suunnittelun yleisiin tilanteisiin ja ongelmiin.

Niin kuin minkä tahansa muun ohjelmiston suunnittelussa, suunnittelumallit ovat hyödyllisiä myös pelimoottoria suunniteltaessa. Yhtenä esimerkkinä voidaan ottaa pelimoottorin grafiikkamoduuli. Useimmiten grafiikkamoduuli on yksilö, jonka tehtävänä on piirtää kaikki tarvittavat objektit ruudulle. Graafisia objekteja sen sijaan on monta, ja niitä voidaan luoda jatkuvasti lisää tai vaihtoehtoisesti tuhota. Suunnittelumalleja hyödyntämällä voidaan löytää näille kahdelle tapaukselle oikeanlaiset luontimallit.

Luontimallia, jota grafiikkamoduuli tottelee, kutsutaan nimellä *ainokainen (Singleton)*. Ainokainen on olion luontimalli, jossa luokasta luodaan täsmälleen yksi konkreettinen ilmentymä koodiesimerkin 1 mukaisesti. Koska ainokainen pitää sisällään ainoan instanssin itsestään, on sillä äärimmäinen kontrolli siitä miten ja mistä sitä kutsutaan.

```
Class Singleton {
public:
    static Singleton* Instance();

protected:
    Singleton();
    private:
    static Singleton* mInstance;

    //Singleton.cpp

    Singleton* Singleton::mInstance = 0;

    Singleton* Singleton::Instance() {
    if(mInstance == 0) {
    mInstance = new Singleton;
    }
    return mInstance;
};
```

Koodiesimerkki 1. Ainokainen.

Tilanteelle, jossa luodaan monta (graafista) objektia, on olemassa useampi luontimalli: *Abstrakti tehdas (Abstarct Factory), rakentajarajapinta* ja *tehdasfunktio (Factory Method).* Jokaiselle mallilla on hyvät ja huonot puolensa, mutta yksinkertaisin näistä on tehdasfunktio. Tehdasfunktio on luokan luontimalli, jossa käytetään abstraktia funktiokutsua. Olion kutsumuoto on määritelty, ja luokan ilmentymän luominen on aliluokan tehtävä. Tehdasfunktiossa abstrakti luokka kapseloi konkreettisen luokan tarvitsemat tiedot ja välittää ne konkreettiselle tuotantoluokalle koodiesimerkin 2 mukaisesti.[7.]

```
//Creator.h

Class Creator {
public:
    virtual Object* Create(ObjectId Id);

    //Creator .cpp

    Object* Creator::Create ( ObjectId id) {
    if (id == MINE) return new GraphicsObject;
    if (id == YOURS) return new UiObject;
    //etc
    return 0;
}
```

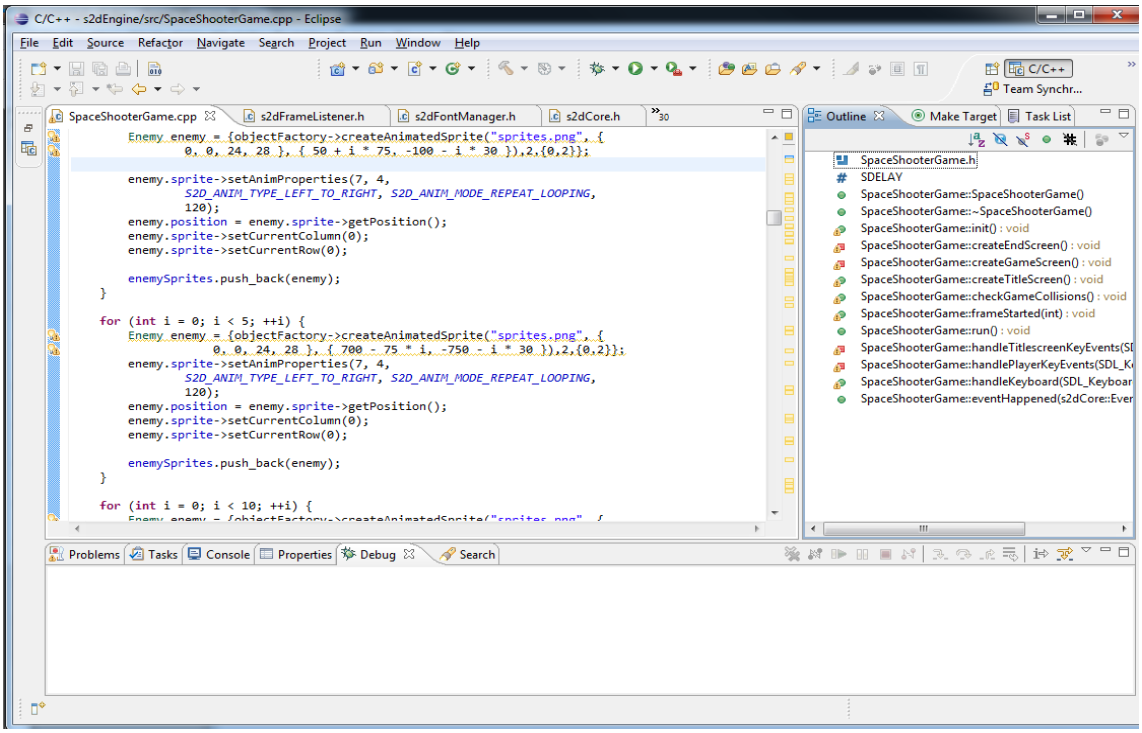
Koodiesimerkki 2. Tehdasfunktio.

4.4 Ohjelmistotyökalut

Ainoat asiat, joita ohjelmoija tarvitsee toimivan ohjelman toteuttamiseen, ovat tekstieditori ja ohjelmointikielen kääntäjän. Näillä työkaluilla työskentely on kuitenkin epäkäytännöllistä ja hidasta. Tehokkaan ohjelmoinnin takaamiseksi onkin kehitelty monenmoisia työkaluja ja ohjelmia. Olisi tyhmää olla huomioimatta ohjelmistotyökaluja myös pelimoottoria tehdessä.

Ensimmäinen valinta on aina uutta ohjelmistoprojektia aloittaessa *ohjelmointiympäristön* hankinta. Ohjelmointiympäristö on ohjelma tai joukko ohjelmia,

jolla ohjelmoija toteuttaa ohjelmistoa. Ohjelmointiympäristö ei eroa ohjelmiston luonnin kannalta juurikaan tekstieditorilla tehdystä, se kuitenkin tarjoaa apuvälineet tehokkaan ja laadukkaan koodin kirjoittamiseen. Useimmat ohjelmistoympäristöt ovat myös laajennettavissa liitännäisillä, joiden avuilla voidaan, esimerkiksi luoda valmis koodirunko tietynlaista ohjelmaa varten. Oikean ohjelmistoympäristön valinta voi lyhentää tuotantoprosessia tunneilla, joten kannattaakin uhrata aikaa sen valinnassa. Tunnetuimpia ohjelmointiympäristöjä on mm. kuvassa 4 oleva *Eclipse, Code::Blocks, Visual Studio, Netbeans.*



Kuva 4. Ohjelmointiympäristö Eclipse.

Seuraavaksi tärkein työkalu mielestäni on versionhallintatyökalut. Versionhallinnalla tarkoitetaan tekniikkaa, joka pitää huolta tiedostojen ja lähdekoodin versioinnista eli niiden muutoksista. Onkin kriittistä versioida lähdekoodia ulkoiseen järjestelmään tulevaisuuden ongelmien varalta. Tämäkin ongelma olisi voitu välttää käyttämällä ulkopuolista versiohallintajärjestelmää. Versionhallintajärjestelmiä on monenlaisia, mutta pohjimmiltaan ne ovat hyvin samankaltaisia. Tunnetuimpia versionhallintajärjestelmiä ovat *Git , Subversion ja Mercurial.*

Hyvällä ohjelmointiympäristöllä ja versionhallintajärjestelmällä pärjää jo pitkälle. On olemassa kuitenkin vielä muutamia hyödyllisiä työkaluja, joihin kannattaa tutustua kuten, erilaisiin debuggereihin ja virheidenjäljitysohjelmiin.

5 Pelimoottorin suunnittelu ja toteutus

5.1 Vaatimusmäärittely

Pelimoottorin kehitys ei juurikaan poikkea minkä tahansa muun ohjelmiston kehityksestä, ja aina ensimmäinen vaihe on tehdä vaatimusmäärittely. Vaatimusmäärittelyn tarkoituksena on selvittää ne ohjelmistotuotteelle asetetut tavoitteet, jotka valmiin järjestelmän tulisi täyttää. Vaatimusmäärittelyllä ei kuitenkaan oteta kantaa siihen, miten vaatimukset toteutetaan. Ohjelmiston tekninen voi toiminta muuttuu sen elinkaaren aikana, siispä on järkevämpää siirtää toteutusmenetelmiin liittyvät päätökset mahdollisimman myöhäiseen vaiheeseen.

Koska pelimoottori on pohjimmiltaan kokoelma hyödyllisiä ohjelmistokomponentteja on siitä hankalaa tehdä vaatimusmäärittelyä. Siksipä kannattaa tehdä vaatimusmäärittely pohjautuen pelimoottorilla tehtyihin peleihin. Ensimmäinen askel on siis sunnitella peli.

Voi tuntua hassulta ajatukselta suunnitella peli ennen moottoria, mutta pelin suunnittelu määrittää moottorille sen rajoitukset ja vaatimukset. Yhtä lailla autonsuunnittelija suunnittelee autonsa vaatimuksien pohjalta (koko, korimalli, jne.). Ilman vaatimuksia ja rajoitteita voi lopputulos olla jotain kuvan 5 tapaista.

Kuva 5. Homer mobile.

Eritoten pelin teknisten vaatimuksien tekoon kannattaa varata aikaa ja huolellisuutta. Voi tuntua epäolennaiselta määrittää asioita, jotka eivät suoranaisesti kuulu peliin, mutta se on kannattavaa pitemmällä aikavälillä. Ilman kunnollista vaatimusten

määrittelyä, ongelmia muodostuu heti kun moottorilla tehtyjä pelejä pelataan muuallakin kuin omassa koneessa. Moottori on voitu kääntää kenties vain yhtä käyttöliittymää ajatellen ja toimimaan vain tietyllä näytönohjaimella. Jos joskus tulevaisuudessa aiotaan myydä moottorilla tehtyjä pelejä, tai sitten moottoria itse, on teknisten vaatimusten määrittely äärimmäisen tärkeää.

Tässä työssä keskitytään pelimoottorin suunnitteluun ja toteutukseen ja sen takia käytän moottorin vaatimusmäärittelyn pohjana yksinkertaistettua versiota Raptor: Call of the Shadows-pelistä.

Moottorin graafiset ominaisuudet

Pelin graafiset ominaisuudet saatiin kun, tutkittiin kuvan 4 visuaalista ulkonäköä.

- 2D-ulotteisuus
- bittikarttagrafiikka
- pelikomponentit sprite-grafiikkaa
- animointi
- käyttöliittymäkomponetteja
- pelikomponentit ovat kerroksittain.

Moottorin on kyettävä piirtämään bittikarttagrafiikkaa ja tekstiä. Bittikartat ja tekstit voivat liikkua ruudulla, ja ne voidaan piirtää kerroksittain (taustakuva alimmaisena, käyttöliittymä päälimmäisenä). Bittikarttoja voidaan myös animoida.

Moottorin on kyettävä piirtämään myös käyttöliittymäkomponetteja. Käyttölliittymäkomponentit eivät liiku muiden mukana vaan ovat staatisesti aina samalla kohdalla kuvaruutua (valikko).

Virhetilanteissa moottorin on tehtävä asianmukainen virheilmoitus ja poistuttava pääsilmukasta.

Moottorin äänitekniset ominaisuudet

Pelin äänitekniset ominaisuudet saatiin katsomalla videonäytteitä Youtube-sivustolta sekä tutkimalla pelin kotisivua.

- äänityyppi OGG
- monta eri kanavaa (päällekkäinen soitto)
- aloitus- ja pysäytystoiminto.

Moottorin on kyettävä toistamaan äänitiedostoja useilta eri kanavalta samanaikaiseksi. Moottorin täytyy tunnistaa OGG-äänityyppi sekä mahdollisesti muita äänityyppejä, kuten MP3 ja WAV. Sen lisäksi sen on kyettävä pysäyttämään ja aloittamaan äänitiedostot käyttäjän käskystä.

Käyttöliittymä

Raptor: Call of the Shadows-peli tukee monia eri syöttölaitteita, mutta tähän työhän valitsen vain muutaman.

- näppäimistötuki
- hiirituki
- näppäinyhdistelmiä (ctrl + space)
- hiireen reagoivat käyttöliittymäkomponentit.

Pelimoottorin täytyy kyetä tulkitsemaan näppäimistö- sekä hiirikomentoja. Moottorin on myöskin tuettava näppäinyhdistelmiä, kuten Ctrl + p. Näppäinkomennot tulee olla muokattavissa. Hiirtä pystyy käyttämään samanaikaisesti näppäimistön kanssa. Virhetilanteissa moottorin on tehtävä asianmukainen virheilmoitus ja poistuttava pääsilmukasta.

Tekniset vaatimukset

Tekniset ominaisuudet saatiin pelin kotisivuilta tutkimalla suositeltuja järjestelmävaatimuksia.

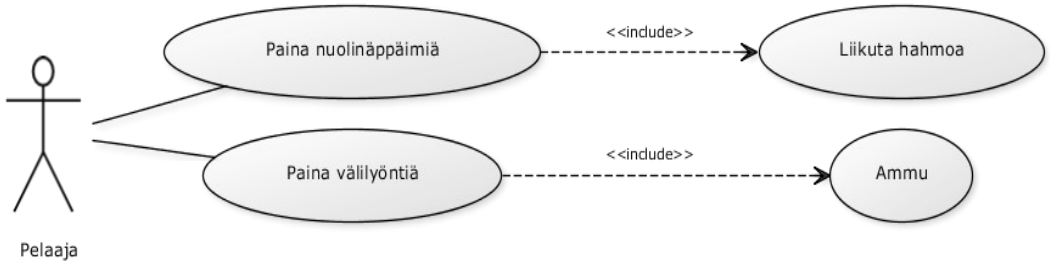
- Windows-alusta
- 2 MB muistia
- 15 MB levytilaa.

Pelimoottorin tulee toimia Windows-alustoilla ja mahdollisesti myös Linux-alustoilla. Pelimoottori ja sillä tehdyt pelit eivät vaadi tietokoneelta suurta suorituskykyä eivätkä mahdollisesti edes näytönohjainta. Pelin koon, ottamatta huomioon resurssien kokoa, ei tule olla suuri.

5.2 Toiminnallinen määrittely

Vaatimusmäärittelyn olessa tarpeeksi kattava siirryttään ohjelmistotuotannon seuraavaan vaiheeseen, toiminnalliseen määrittelyyn. Toiminnallinen määrittely kuvaa, mitä kaikkea pelimoottorilla voi tehdä sekä miten käyttäjä voi ne tehdä. Siitä syntyvä määrittelydokumentti tulee olla niin kattava, että teknisessä suunnittelussa tai sitä seuraavissa vaiheissa ei tule mitään epäselvyyksiä siitä, miten ohjelmiston tulee toimia kussakin tilanteessa. Yksinkertaiselle *Pacman*-pelimoottorille ei selvästikään kannata tehdä 500 sivuista dokumenttia, mutta mitä monimuotoisemmaksi pelimoottori tulee sen kattavammaksi kannattaa myös määrittelydokumentti tehdä.

Määrittelydokumentin tekeminen pelin vaatimusmäärittelyn pohjalta voi olla hankalaa ja se vaatiikin erityistä huolellisuutta. Erityisesti määrittelyn monimuotoisuuteen kannattaa uhrata aikaa ja huolellisuutta. Liian helposti voidaan määritellä kuvan 6 mukaisesti pelin ominaisuuksia eikä niinkään pelimoottorin.



Kuva 6. Täsmällinen käyttötapauskaavio.

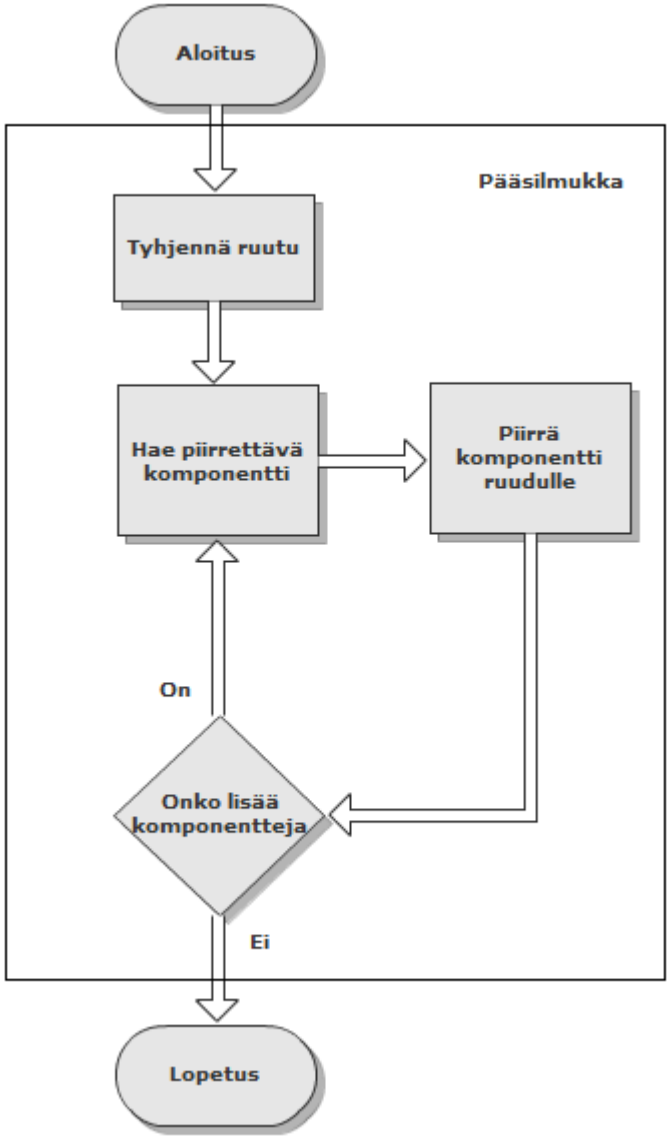
Kuvan 6 tapauksessa moottori suunnitellaan nimenomaan liikuttamaan hahmoa nuolinäppäimillä ja ampumaan välilyönnillä. Mitä sitten jos pelissä liikutetaan jotain muutakin nuolinäppäimillä? Taikka jos hahmo myös hyppää välilyönnistä? Tapaukset ovat liian yksityiskohtaisia eivätkä anna juurikaan laajentamisen varaa. Ratkaisu olisi abstrahoida tapauksia kuvan 8 mukaisesti.

Kuva 7. Geneerinen käyttötapauskaavio.

Kuvassa 8 ei oteta kantaa siihen, mikä komento näppäimistöllä annetaan, eikä siihen mihin sitä käytetään. Se voi olla pelihahmon liikuttamista varten taikka pelin sulkuoperaatio. Tapaus kuvastaakin yleistä tilannetta, jossa pelaaja antaa komennon jonka jälkeen se tulkitaan ja toteutetaan. Lähetymistapa antaa ohjelmiston toteuttajalle paljon liikkumavaraa käyttöliittymä moduulia luodessaan.

5.2.1 Hahmonnusjärjestelmä

Hahmonnusjärjestelmä on se osa moottoria, joka hoitaa bittikarttojen, tekstien ja käyttöliittymäkomponettien piirtämisen. Se pitää sisällään kaikki ne komponentit mitä tulee ajon aikana piirtää. Järjestelmä toimii kuvan 8 mukaisesti.



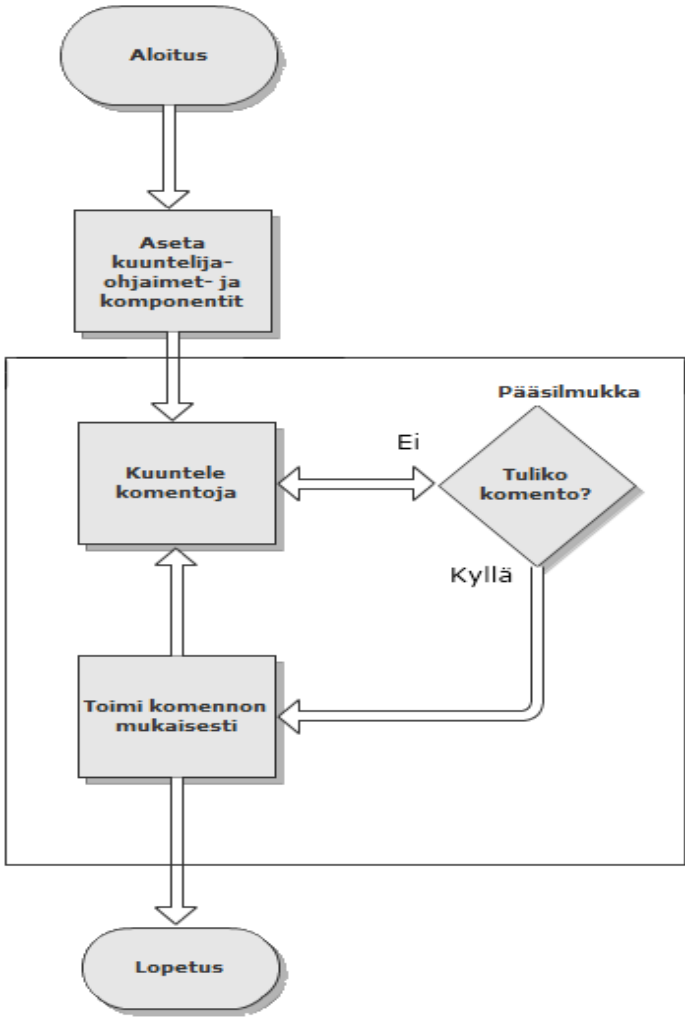
Kuva 8. Hahmonnuksen toimintakaavio.

Järjestelmän toiminta alkaa järjestelmäkomponenttien alustuksella ja piirrettävien komponenttien lataamisella, jonka jälkeen siirrytään pääsilmukan sisään.

Pääsilmukassa ensimmäisenä tehtävän on ruudun tyhjennys, jonka jälkeen suoritetaan komponenttien piirtoa.
Kaikkien komponenttien piirtämisen jälkeen on yksi hahmonnuskierros suoritettu. Tämän jälkeen aloitetaan uusi hahmonnuskierros, tai vaihtoehtoisesti siirrytään pois pääsilmukasta. Pääsilmukan loputtua piirrettävät komponentit tuhotaan ja järjestelmä lopetetaan. Jos ajon aikana järjestelmässä tapahtuu virhe, kirjataan virhe muistiin ja poistutaan pääsilmukasta.

5.2.2 Syöttölaite- ja käyttöliittymäjärjestelmä

Syöttölaite- ja käyttöliittymäjärjestelmällä tarkoitetaan kaikkea sitä millä pelaaja kommunikoi pelin kanssa, oli se sitten näppäinkomennot taikka sitten ruudulla olevan napin painallus hiirellä. Komentoja voi saada monella tapaa riippuen siitä mitä apukirjastoja käytetään. Järjestelmä toimii kuvan 9 mukaisesti.

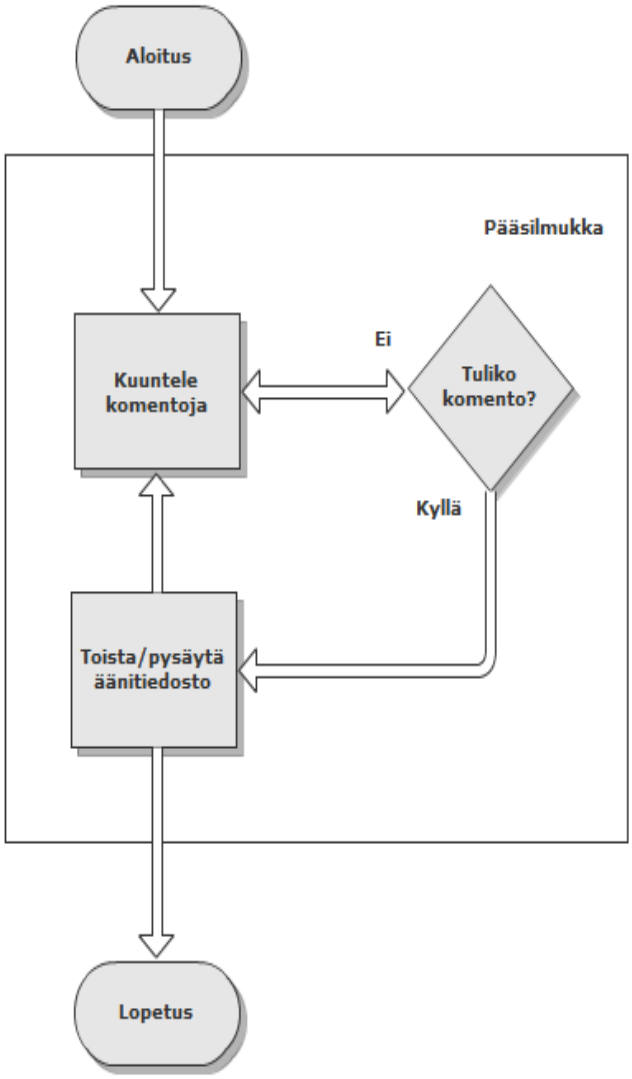


Kuva 9. Käyttöliittymäjärjestelmän toimintakaavio.

Järjestelmän toiminta alkaa järjestelmäkomponenttien alustuksella, jonka jälkeen määritellään komentokuuntelijat kaikille tarvittaville ohjaimille ja käyttöliittymäkomponenteille. Tämän jälkeen siirrytään pääsilmukkaan. Pääsilmukassa kuunnellaan komentoja kaikilta kuuntelijoilta. Kun komento on havaittu, toteutetaan komennon määrittämä toiminto. Komennon toteuttamisen jälkeen siirrytään takaisin kuunteluvaiheeseenm, paitsi jos komento oli poistumiskäsky. Tällöin poistutaan pääsilmukasta ja suoritetaan kuuntelijoiden sekä järjestelmän tuhoaminen. Jos ajon aikana käyttöliittymämoduulissa tapahtuu virhe, kirjataan virhe muistiin ja poistutaan pääsilmukasta.

5.2.3 Äänentoistojärjestelmä

Äänentoistojärjestelmä hoitaa pelin ääniteknisen osuuden. Se hoitaa äänitiedostojen latauksen sekä toistamisen. Se sisältää kaikki ladatut äänitiedostot ja toistaa ne käyttäjän haluamalla hetkellä. Järjestelmä toimii kuvan 10 mukaisesti.



Kuva 10. Äänentoistojärjestelmän toimintakaavio.

Järjestelmän toiminta alkaa järjestelmän ja apukirjastojen alustuksella ja äänitiedostojen latauksella. Tämän jälkeen siirrytään pääsilmukkaan. Pääsilmukassa kuunnellaan käyttäjän komentoja. Kun komento havaitaan, toistetaan tai pysäytetään komennon määrittämä äänitiedosto. Komennon jälkeen siirrytään takaisin kuunteluvaiheeseen, paitsi jos komento oli poistumiskäsky. Tällöin poistutaan pääsilmukasta ja suoritetaan järjestelmän tuhoaminen ja äänitiedostojen vapauttaminen muistista. Jos ajon aikana äänentoistojärjestelmässä tapahtuu virhe, kirjataan virhe muistiin ja poistutaan pääsilmukasta.

5.3 Tekninen määrittely ja toteutus

Tekninen määrittely eli arkkitehtuurisuunnittelu seuraa toiminnallisen määrittelyn jälkeen. Sen tarkoituksena on kuvata ohjelmiston tekninen arkkitehtuuri tarkasti, mutta ei kuitenkaan vielä ohjelmakoodina. Tekninen määrittely tehdään toiminnallisen määrittelyn pohjalta, ja se kuvaa ne ohjelmistokomponentit, jotka toteuttavat määritellyn vaatimat toiminnot.

Tekniseen määrittelyyn sisältyy ohjelmointikielen/kielten määrittely, ohjelmistokomponettien ja kirjastojen valinta, ohjelmistokomponenttien rakenne ja keskinäinen hierarkkia.

Tekninessä määrittelyssä kannattaa käyttää hyväksi suunnittelumalleja (design patterns) ja visuaalisia työkaluja (UML, CASE). Työkalujen ja mallien käyttö yksinkertaistaa suunnnittelutyötä ja näin lyhentää prossesin kokonaiskestoa.

5.3.1 Ohjelmointikieli ja mallinnusperiaatteet

Ohjelmointikieleksi valitsin tähän projektiin C++:n, koska kyseinen kieli on minulle entuudestaan hyvinkin tuttu. Lisäksi se on nopea ja kevyt kieli, mikä on eduksi yksinkertaista ja minimaalista pelimoottoria tehdessä. Ohjelmointialustaksi valitsin Windows 7:n, mutta teoriassa valmiin ohjelman pitäisi toimia myös Linux- ja Mac-käyttöjärjestelmissä. Toimiakseen muilla käyttöjärjestelmillä ohjelman täytyy kääntää uudelleen kyseisellä käyttöjärjestelmällä.

Toteutus on pääosin oliopohjainen ja siinä käytetään hyödyksi suunnittelumalleja. Käytän näitä, koska ne tukevat hyvin periaatetta jakaa ohjelmiston hallittaviin ja selkeisiin osiin. Osat luodaan myös mahdollisimman itsenäiseksi moduuleiksi jotka kommunikoivat keskenään tiettyjen rajapintojen kautta. Tällaisen suunnittelun takia ohjelmistoa on myös helppo laajentaa tulevaisuudessa.

En usko monirivisiin komenttiosuuksiin lähdekoodissa, joten kirjoitan koodin helposti luettavaksi ja itsestään selittäväksi. Vaikeasti ymmärrettävät kohdat tuen kommenttiosuuksilla.

5.3.2 Työkalujen, komponenttien ja kirjastojen valinta

Ohjelmointiympäristöksi valitsin Eclipsen, koska se on ilmainen ja entuudestaan tuttu. Toisin kuin Microsoft Visual Studiolla, Eclipsellä tehdyt ohjelmat eivät vaadi erinnäisten liitäntäpakettien (*redistribution package*) asenteluja toimiakseen. Tämän lisäksi käytän Subclipse-liitännäistä apuna versionhallinnassa. Versionhallintajärjestelmänä minulla on käytössä *SVN (Subversion),* joka kommunikoi ulkoisen versioarkiston kanssa. Projektissa käytän seuraavia C/C++-kirjastoja ja komponentteja:

- SDL (Simple DirectMedia Layer) on ohjelmakirjasto, joka määrittelee käyttöliittymien omat natiivitoiminnot yhdeksi yhtenäiseksi rajapinnaksi. Tämä on äärimmäisen hyödyllistä alustariippumatonta ohjelmaa tehdessä, koska rajapinta poistaa tarpeen tehdä jokaiselle käyttöjärjestelmälle omat toteutukset. SDL hoitaa rajapintansa kautta grafiikkatulostukset, äänien toistot ja syöttölaitekutsut. Vaikka SDL pystyy piirtämään, se ei kuitenkaan käytä apuna grafiikkakiihdytystä vaan kaikki piirtotyö tehdään prosessorin avulla.
- SDL Image -kirjastoa käytetään uusien kuvaformaattien lukemista varten. Tuettavavia formaatteja ovat BMP, GIF, JPEG, LBM, PCX, PNG, PNM ja TGA. SDL Image tarvitsee toimiakseen libz, libpng ja SDL kirjastot. SDL ttf on kirjasto, joka mahdollistaa TTF (TrueTypeFont)-fonttien käytön SDL-ohjelmissa.
- SDL gfx on a kirjasto, joka on tehty SDL:n päälle. SDL gfx kirjasto sekä helpottaa ja laajentaa SDL:n omia funktioita ja toimintoja. Kirjasto hoitaa esimerkiksi bittikarttojen kääntö- ja kiertotoiminnat.
- SDL mixer on kirjasto äänitiedostojen lukua ja toistoa varten. Se tukee monia tunnettuja musiikkimuotoja kuten MP3 ja OGG.

Ohjelma vaatii toimiakseen vielä muutamia dynaamisia kirjastoja, jotka ovat esitelty liitteessä 17. Nämä tulevat lisäkirjastojen mukana.

5.3.3 Hahmonnusjärjestelmän toteutus

Hahmonnusjärjestelmän ydin koostuu kolmesta luokasta: *Renderer* [liite 1], *Scene* [liite 3] ja *SceneManager* [liite 2].

Renderer on ohjelmakomponentti, joka määrittelee ohjelman näyttöasetukset ja hoitaa komponenttien piirron SDL-rajapinnan kautta. Renderer-luokka on staattinen, ja sitä voi olla vain yksi instanssi. Tämä on tehty siksi, koska näyttöasetuksien määrittely monesta eri luokasta voi aiheuttaa vikatiloja. Renderer-luokka ei omista mitään piirrettäviä komponenetteja vaan kommunikoi piirrettävien komponenttien kanssa Scene-olion kautta. Piirtototeutus on tehty koodiesimerkin 3 mukaisesti.

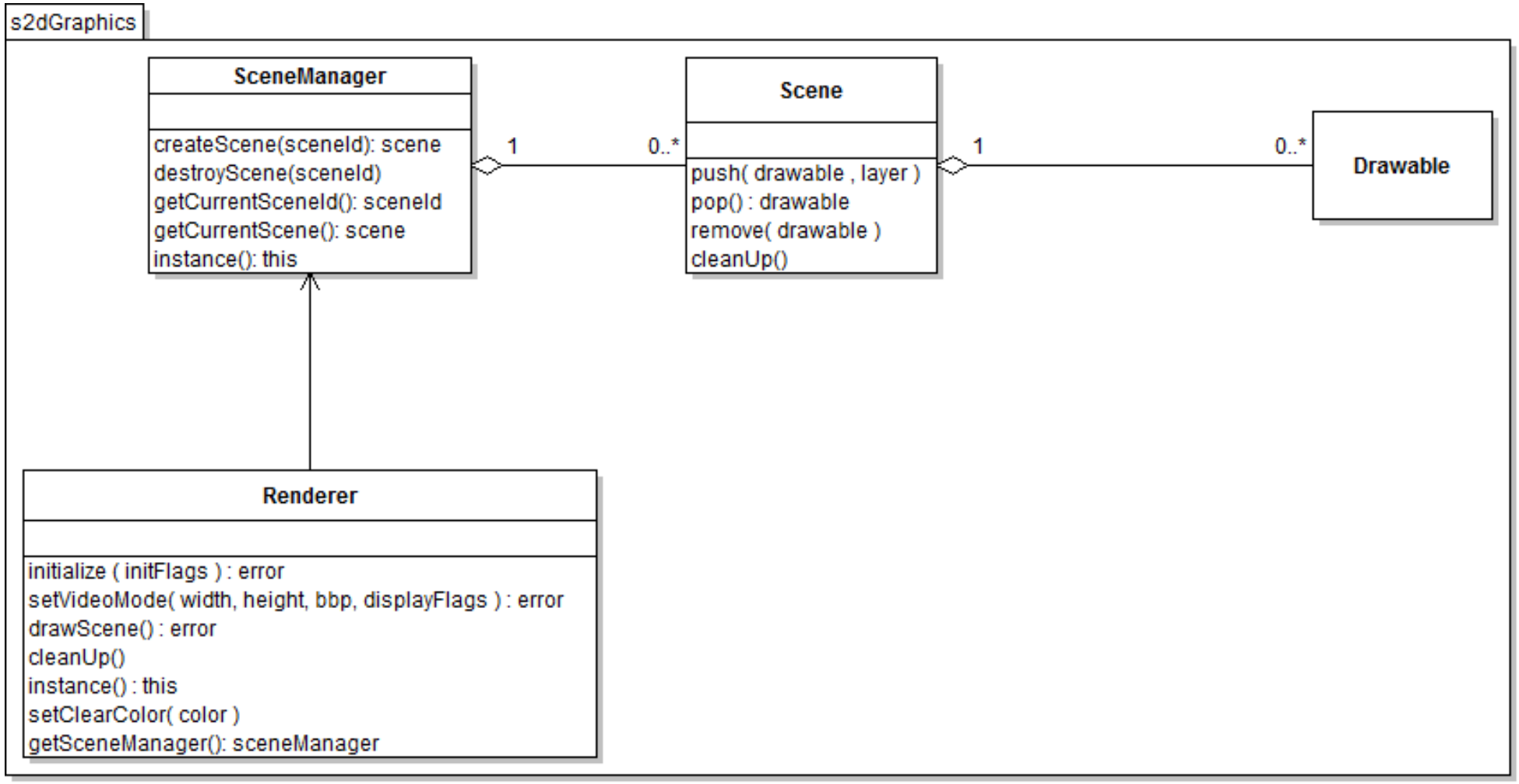
```
SceneManager *manager = SceneManager::instance();
Scene *scene = manager->getCurrentScene();
if(scene == NULL){
	//Empty scene
	return 0;
}
s2dDrawMap drawMap = scene->mDrawingQueue;

SDL_FillRect(mScreen, NULL, mClearColor); //Empty the screen
                                          // with specified color
for (int i = 0; i < 5; i++) {
	std::vector<Drawable*> vec = drawMap[i];
	if (vec.empty())
		continue;
	for (int d = 0; d < vec.size(); d++) {
		if(vec[d]->isVisible()){
			vec[d]->draw(mScreen);
		}
	}
}
SDL_Flip(mScreen); //Swap video buffers (if possible)
return 0;
```

Koodiesimerkki 3. Hahmonnusjärjestelmän piirtototeutus.

Scene on luokka, joka sisältää kaikki piirrettävät komponentit ja niiden ominaisuudet, eli voisi kuvailla sen olevan ”kohtaus”, jonka Renderer-olio piirtää. Scene-luokka säilöö komponenttinsa piirtojärjestyksen perusteella niin, että esimerkiksi taustakuvat piirretään ensin ja käyttöliittymäkomponentit viimeiseksi. Järjestyksen määrittää moottorin käyttäjä komponentteja luodessaan. Piirrettävät komponentit voi halutessaan nimetä omalla tunnisteella, mutta se ei ole pakollista, koska luokka osaa luoda omat tunnisteet.

Scene-luokkia luodaan ja tuhotaan SceneManager-luokan kautta, ja se myös määrittää, minkä Scene olion Renderer-olio tulee piirtämään seuraavaksi. SceneManger tallentaa kaikki luodut Scene-oliot säiliöön, josta niitä voidaan tarvittaessa hakea oikeaa *sceneId*-tunnistetta vastaan. Tästä syystä SceneManager-luokka on staattinen, ja sitä voidaan luoda vain yksi instanssi.

Hahmonnusjärjestelmän luokkakaavio on kuvassa 11.



Kuva 11. Hahmonnusjärjestelmä.

Erilaisia piirrettäviä komponentteja moottorissa on neljä: *Sprite* [liite 6], *AnimatedSprite* [liite 7], *UiComponent* [liite 8] ja *Text* [liite 9]. Käyttäjän on mahdollista tehdä myös oma piirrettävä komponentti, mutta sen täytyy periä *Drawable*-luokka [liite 5], taikka jonkun sen aliluokista ja antaa sille luonnissa tarvittavat parametrit *(mId,* mType).

Drawable-luokka on kaikkien piirtokomponenttien aliluokka, jonka kautta Renderer luokka suorittaa piirtämisen. Drawable-luokka on geneerinen piirtoluokka joka sisältää osoitteen piirrettävästä bittikartasta sekä kaikki tarvittavat perustoiminnot. Näiden lisäksi oliolla on yksilöllinen tunniste *mId,* jonka avulla pystytään erottelemaan piirrettävät komponentit toisistaan. Drawable-luokka ei kuitenkaan piirrä mitään vaan se määrittää yhteisen rajapinnan piirtofunktiota varten.

Sprite-luokka on laajennetuilla piirto-ominaisuuksilla varustettu piirtoluokka, jonka ominaisuuksiin kuuluu esimerkiksi kuvan koon muuttaminen ajon aikana. Jotta alkuperäisellä piirrettävälle bittikartalle ei tapahtuisi mitään, on sillä oma kopio siitä jota olio voi muokata mielensä mukaan. Olion tuhouduttua myös kopiokuva tuhotaan. Sprite-luokka toteuttaa myös bittikartan piirtämisen, eli se toteuttaa Drawable-luokan *Draw*-funktion. Draw funktio on toteutettu koodiesimerkin 4 mukaisesti.

```
SDL_Rect destRect = mDestination; //Copy from the original draw des-
tination

if (mModifiedSurface == NULL) {    //If bitmap surface was null, draw
pink                                        rectangle
	SDL_FillRect(dest, &destRect, 0xff00ff);
}
else if (SDL_BlitSurface(mModifiedSurface, &mModifiedSource, dest,
&destRect) == -1) {
	SDL_FillRect(dest, &destRect, 0xff00ff);
}
```

Koodiesimerkki 4. Draw-funktion toteutus.

AnimatedSprite on Sprite-luokasta peritty aliluokka, jossa on perustoimintojen lisäksi animaatio-ominaisuuksia. Animointia varten luokalle tarvitsee määritellä animaatioasetukset, tahdistus sekä animointiruutujen rivi- ja sarakemäärä. Piirtomenetelmänsä AnimatedSprite perii Sprite-luokalta

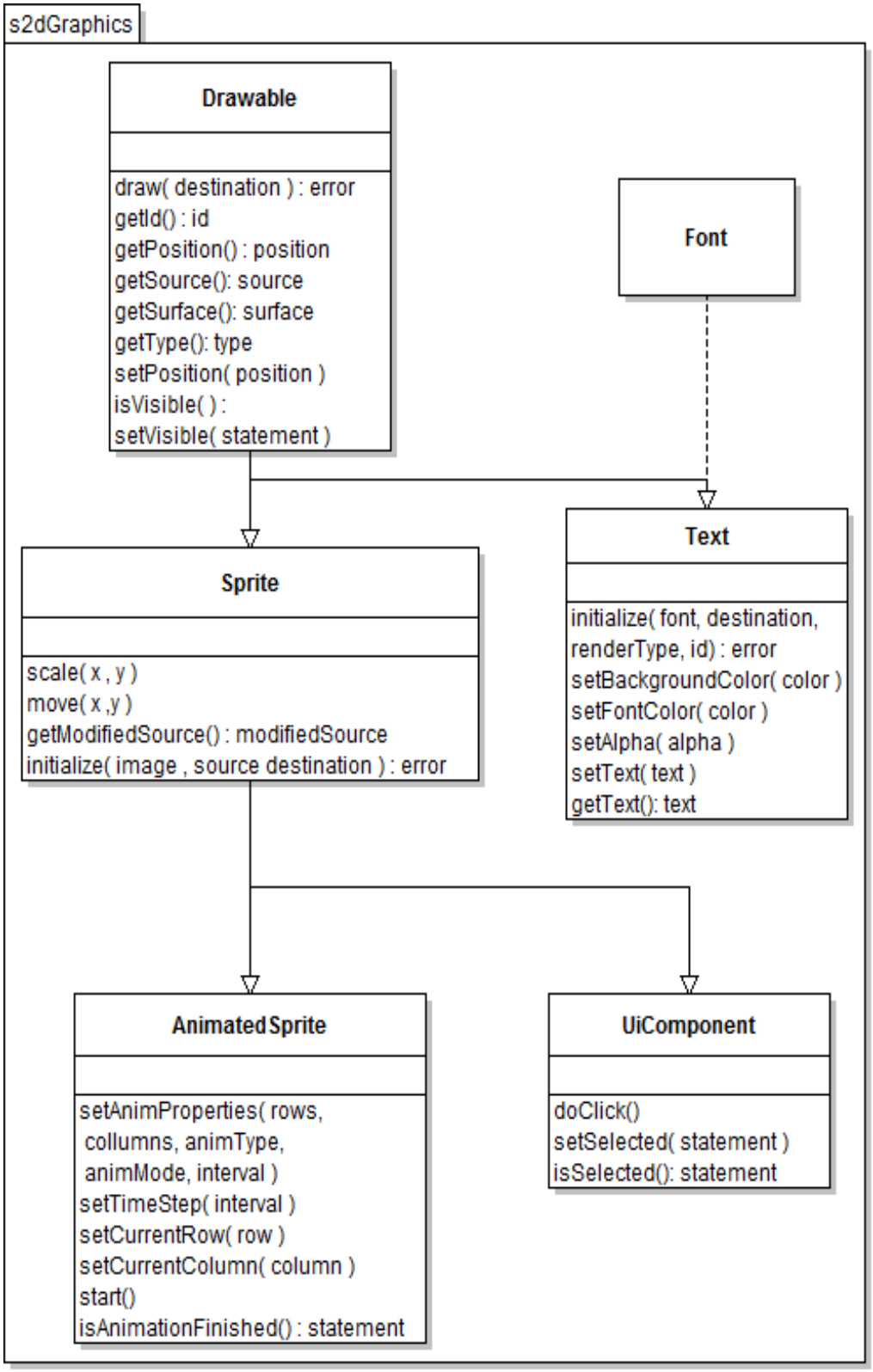
UiComponent on käyttöliittymäkomponentti, joka perii piirto-ominaisuutensa Sprite-luokasta. Luokka toimii kuten Sprite luokka, mutta se kuuntelee käyttäjän hiirikomentoja. Kun käyttäjä napsauttaa komponenttia hiirellä, lähettää se tästä viestin *EventListener*-oliolle. Jotta olio voisi toimia, täytyy sille antaa osoitin *ActionListener*-olioon. Piirtomenetelmänsä UiComponent perii Sprite-luokalta

Text-luokka on tekstikirjoitusluokka, joka pystyy kirjoittamaan TTF-fontteja. Fonttien lataus ja kirjoitus suoritetaan SDL ttf-kirjaston kautta. Fontit ja sen ominaisuudet ovat tallennettu *Font* [liite 16] olioon, josta Text-oliolla on osoitin. SDL ttf-luokka luo tekstit aina itsenäisiksi bittikartoiksi, mikä tarkoittaa myös sitä, ettei tekstiä pysty muokkaamaan ilman bittikartan uudelleenluontia. Text luokka hoitaa tämän ongelman koodiesimerkin 5 mukaisesti.

Piirtokomponenttien luokkakaaviot löytyvät kuvasta 12.

```
mText = text;


if (mFont == NULL) {
	mSurface = NULL;
	return;
}
if(mSurface != NULL) { //free previous bitmap surface
	SDL_FreeSurface(mSurface);
}
switch (mRenderType) {
	case S2D_TEXT_BLENDED: {
		mSurface = TTF_RenderText_Blended(mFont, mText,
							mFontColor);
		break;
	}
	case S2D_TEXT_SOLID: {
		mSurface = TTF_RenderText_Solid(mFont, mText,
							mFontColor);
		break;
	}
}
```

Koodiesimerkki 5. Tekstin muokkaus.



Kuva 12. Piirtokomponentit.

Piirrettävien komponenttien sekä fontit luodaan omilla tehdasluokilla. Font olioiden luontia varten on tehty *FontManager*-luokka [liite 4]. Se lataa ja hallitsee kaikkia ohjelmassa olevia TTF-fontteja niin, että mahdollisimman moni Text-olio käyttää samaa Font-oliota. Tämä vähentää resurssien tuhlausta. Koska luokka sisältää kaikki ohjelmassa olevat Font-oliot, on luokka sen takia staattinen ja siitä voi luoda vain yhden instanssin.

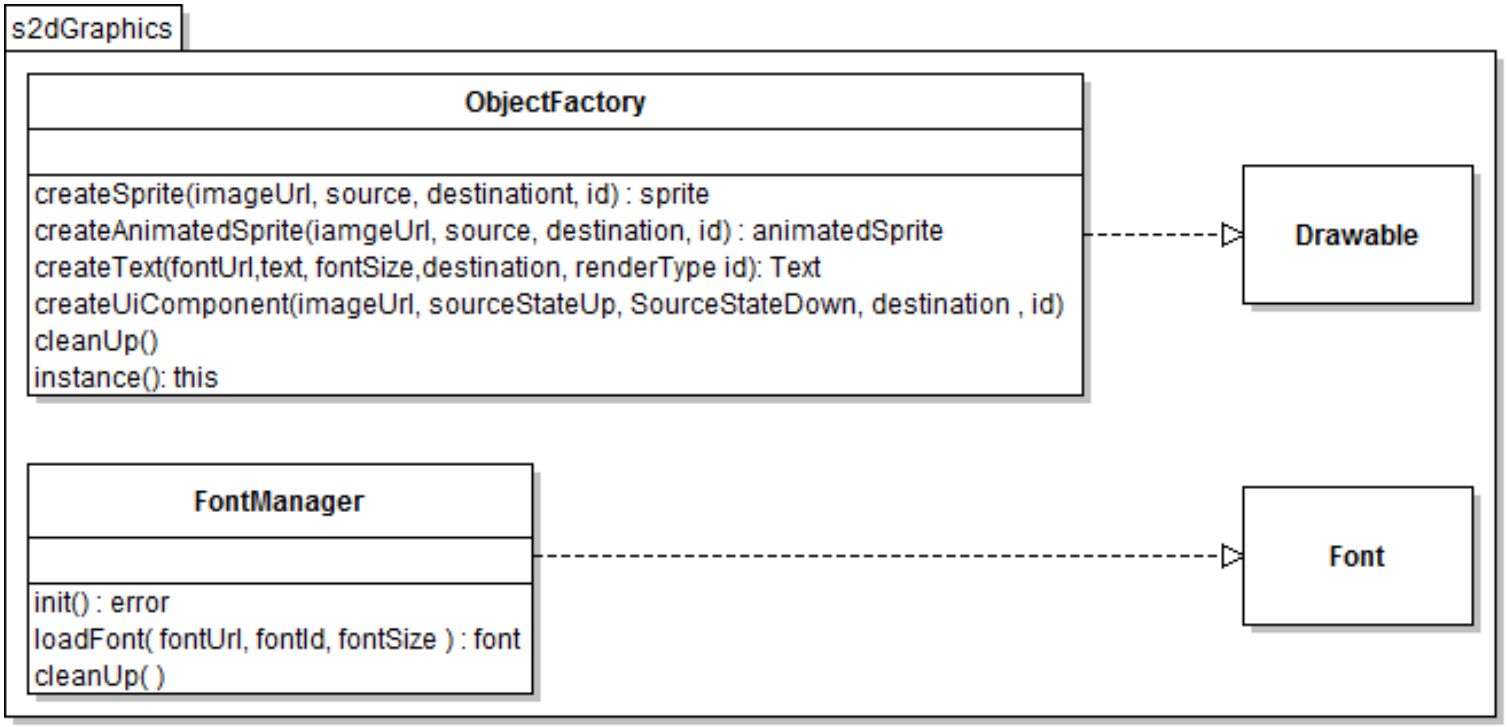
Kaikki Drawable-luokasta perityt luokat tehdään *ObjectFactory*-oliossa [Liite 10]. ObjectFactory-luokassa on tehdasfunktiot kaikille piirrettäville luokille, esimerkkinä Sprite-olion luonti joka on esitetty koodiesimerkissä 6.

```
std::string retId;
if (id == 0) { //No id defined. Create own.
	char* buffer = new char[255];
	std::string str(imgURL);
	str += "Sprite";
	sprintf(buffer, "%d", mCreatedObjects);
	str += buffer;
	retId = str.c_str();
	mCreatedObjects++;
	delete buffer;
} else {
	retId = id;
}

SDL_Surface *img;
img = loadImage(imgURL);
Sprite *ret = new Sprite();
ret->initialize(img, src, dest, retId.c_str());
return ret;
```

Koodiesimerkki 6. Sprite-olion luonti.

Olioiden luonnin yhteydessä ObjectFactory suorittaa myös bittikarttojen lataukset, lukuun ottamatta Text-olion luontia, jossa fontin lataus hoidetaan FontManager-luokassa. ObjectFactory-luokka pitää sisällään kaikki ohjelmassa olevat bittikartat, ja tämän takia se on staattinen ja sitä voidaan luoda vain yksi instanssi. Luokka myös

hallitsee bittikarttoja niin, että samoja bittikarttoja voidaan käyttää monien eri piirrettävien komponenttien kesken. Jos käyttäjä haluaa luoda tehdä omia piirrettäviä komponentteja, kannattaa hänen luoda oma, ObjectFactorystä peritty tehdasluokka komponenttien luontia varten. Tehdasluokkien luokkakaaviot löytyvät kuvasta 13.



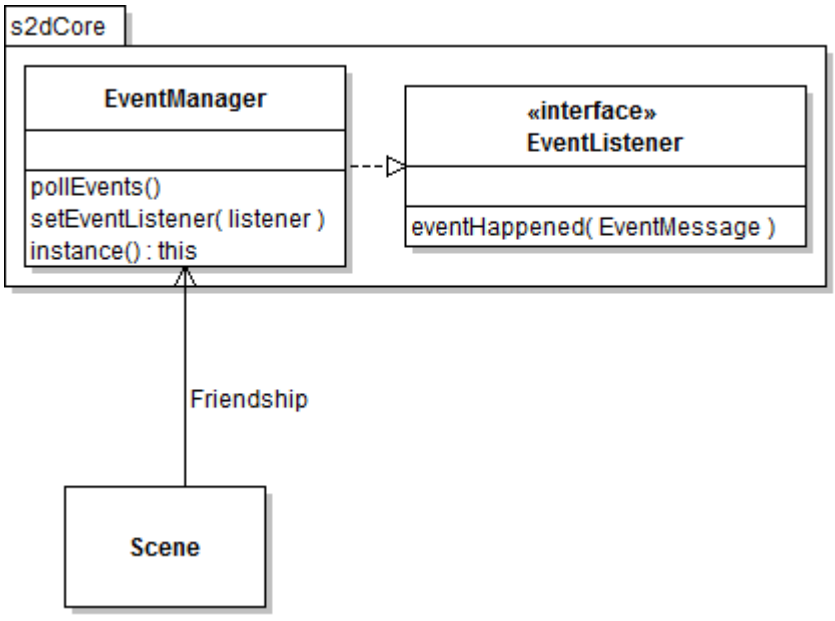
Kuva 13. Tehdasluokat piirtokomponenteille ja fonteille.

5.3.4 Käyttöliittymäjärjestelmän toteutus

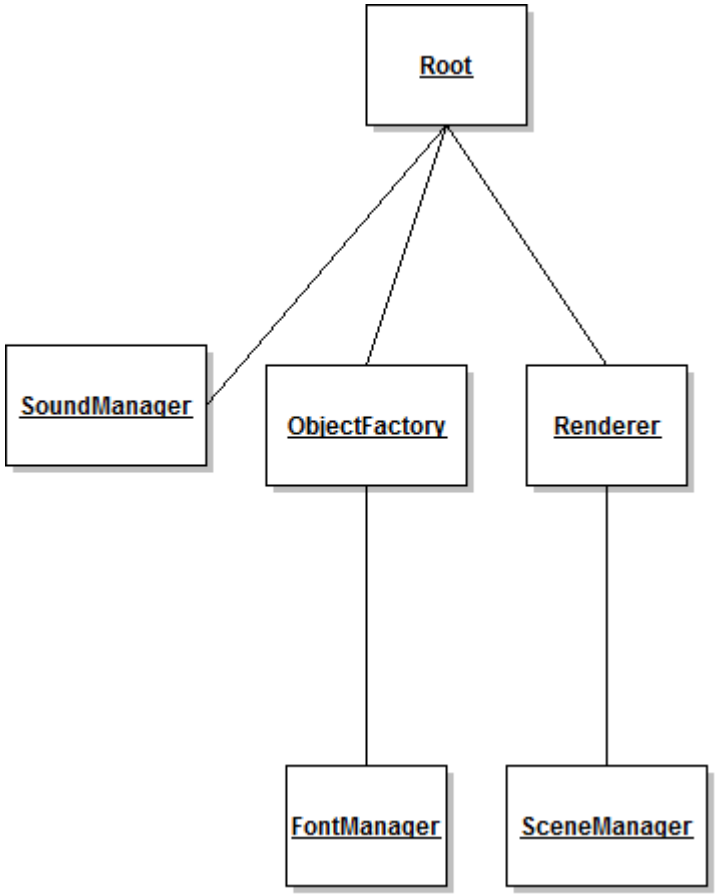
Käyttöliittymä koostuu kahdesta luokasta, *EventManager* [liite 12] sekä *EventListener* [liite 13].

EventManager on luokka, joka kuuntelee SDL-rajapinnan kautta käyttäjän näppäimistö- ja hiirikomentoja. Näiden lisäksi se myös kuuntelee kaikkia käyttöliittymäkomponetteja (UiComponent) ystäväluokka Scenen kautta. Komennon saatua pakataan se viestiksi (*EventMessage* [liite 16]) ja lähetetään eteenpäin EvenListenerlle. EvenManagerilla voi olla vain yksi EventListener-rajapinta, ja tämän takia se staattinen ja siitä voi luoda vain yhden instanssin.

EvenListener on rajapinta, jonka kautta käyttäjä saa viestin komennoista. Jotta EventListener toimisi, täytyy käyttäjän periä luokasta oma kuuntelija ja antaa se EventManagerille *setListener*-funktion kautta. Tämän lisäksi sen pitää toteuttaa *eventHappened*-funktio. Käyttöliittymäluokkien luokkakaaviot löytyvät kuvasta 14.



Kuva 14. Käyttöliittymäjärjestelmä.



Kuva 15. Moottorin pääluokkien hierarkia.

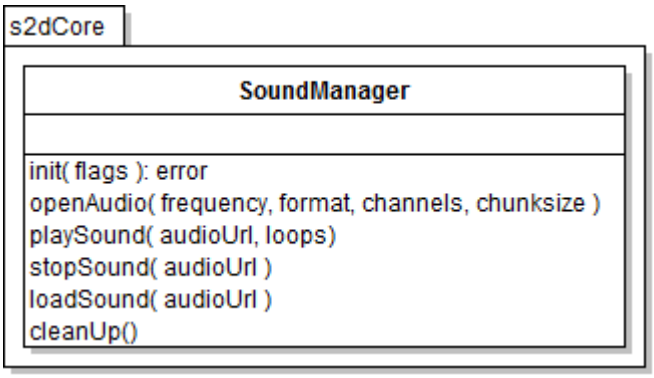
5.3.5 Äänentoistojärjestelmä

Äänentoistoa hallitsee luokka nimeltä *SoundManager* [liite 11]. Se on luokka, joka määrittää pelin ääniasetukset sekä toistaa tarvittavat äänitiedostot. Se sisältää kaikki äänitiedostot, jotka pelissä tulee olemaan ja tämän takia se staattinen ja siitä voi luoda vain yhden instanssin. Äänitiedostojen luku ja toisto on toteutettu SDL mixer -kirjaston avulla ja niin, ettei samoja äänitiedostoja tallenneta useampaan kertaan. Äänitiedostojen lataus on toteutettu koodiesimerkin 7 mukaisesti.

```
void SoundManager::loadSound(const char *url) {
	Mix_Chunk *music = NULL;
	if (mSoundMap.find(url) == mSoundMap.end()) {
		music = Mix_LoadWAV(url);
		if (music == NULL) {
			std::cout << "Error: could not load music "
					<< url << std::endl;
			std::cout << Mix_GetError() << std::endl;
		} else {
			s2dSoundData data = {music , -1};
			mSoundMap[url] = data;
		}
	}
}
```

Koodiesimerkki 7: Äänitiedostojen lataus

Äänentoistojärjestelmän luokkakaavio löytyy kuvasta 15.



Kuva 16. Äänentoistojärjestelmä.

5.3.6 Ydinjärjestelmä

Ydinjärjestelmällä tarkoitetaan sitä osaa moottorissa joka hallitsee kaikkia muita moottorin osia. Tällaista järjestelmämuotoa kutsutaan myös nimellä julkisivu (*facade*).

Root [liite 15] on julkisivu-luokka, joka tarjoaa kuvan 16 mukaisesti yhtenäisen rajapinnan alijärjestelmäluokkien rajapintojen joukolle ja niiden keskeisille toiminnoille. Tämä ratkaisu vähentää alijärjestelmien riippuvuuksia ja niiden välistä kommunikaatiota. Luokkien väärinkäyttöriski vähentyy myös huomattavasti, kun kaikki alijärjestelmät täytyy luoda Root-olion kautta. Tämän takia Root-oliolla on ystävyyssuhde kaikkien alijärjestelmien kanssa.

Käyttäjä pääsee aliluokkiin käsiksi vain Root-olion kautta ja esimerkiksi näyttöasetusten määrittely ja äänitiedostojen lataus täytyy hoitaa koodiesimerkin 8 mukaisesti.

```
Renderer *renderer = root->getRenderer();
renderer->setVideoMode(800, 600, 32, SDL_HWSURFACE);

SoundManager *soundManager = root->getSoundManager();
soundManager->loadSound("shoot.ogg");
soundManager->loadSound("hit.ogg");
soundManager->loadSound("explosion.ogg");
```

Koodiesimerkki 8. Root olion käyttö.

Käyttäjää helpotukseksi Root luokalla on alijärjestelmien luontia varten tehty *bigInit*-funktio, joka luo kaikki alijärjestelämät perusasetuksilla. Vastaavanlainen funktio on myös alijärjestelmien tuhoamista varten nimeltä *bigClean.*

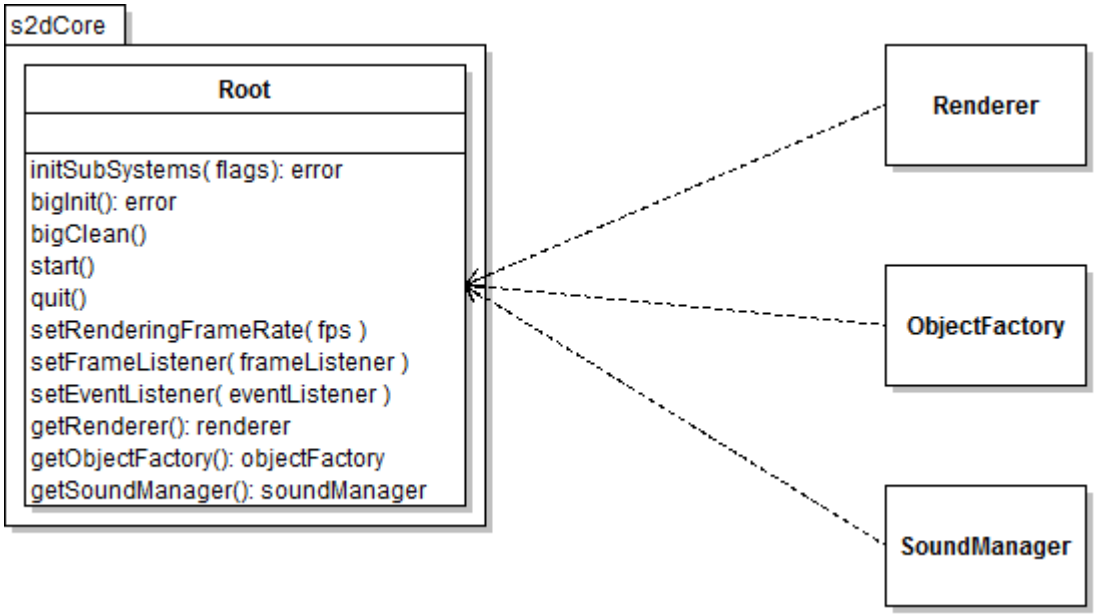
Kun kaikki halutut alijärjestelmät on luotu, voidaan siirtää Root, olio ajotilaan kutsumalla *start*-funktio. Olio kiertää pääsilmukkaa niin kauan kunnes, käyttäjä lopettaa sen. Silmukan sisällä Renderer olio suorittaa piirtotoiminnot jonka jälkeen EventManager olio kuuntelee käyttäjän komentoja koodiesimerkin 9 mukaisesti.

```
void Root::start() {

	mState = S2D_STATE_RUNNING;
	int elapsedTime = 0;
	int previousTime = SDL_GetTicks();

	Renderer *mRenderer = getRenderer();

	while (mState == S2D_STATE_RUNNING) {
		if(mFrameRate == -1){
			mFrameListener->frameStarted(elapsedTime);
			previousTime = SDL_GetTicks();
			mRenderer->drawAll();
			mFrameListener->frameEnded(elapsedTime +
					SDL_GetTicks() - previousTime);
		}
		else {
			if(elapsedTime > (1.f / (float) ((mFrameRate)) * 1000)) {
				mFrameListener->frameStarted(elapsedTime);
				previousTime = SDL_GetTicks();
				mRenderer->drawAll();
				mFrameListener->frameEnded(
					elapsedTime + SDL_GetTicks() -
						previousTime);
			}
		}

		EventManager::instance()->pollEvents();
		elapsedTime = SDL_GetTicks() - previousTime;
		SDL_Delay(1); //Delaying for other running processes
	}
}
```

Koodiesimerkki 9. Pääsilmukka.

Root-luokka hallitsee myös ruudunpäivitystä, ja sen pystyy määrittelemään setRenderingFrameRate-funktion kautta. Halutessa ruudunpäivitystahdin voi jättää säätämättä, mutta tämä ei ole suositeltavaa mahdollisten ylikuumenemistilanteiden varalta.

Jotta käyttäjä pystyisi tekemään toimintoja, kuten luomaan ja tuhoamaan komponentteja, täytyy käyttäjän päästä pääsilmukan sisälle ilman lähdekoodin muokkausta. Tämä voidaan tehdä EventListener-olion kautta, mutta parempi vaihtoehto on käyttää *FrameListener*-kuuntelijaoliota [liite 14]. Root olio-lähettää FrameListener-oliolle viestin ennen piirtotoimintija sekä piirtotoimintojen jälkeen. Perimällä FrameListener olion-ja toteuttamalla joko *frameStarted*- taikka *frameEnded*-funktiot, pystyy käyttäjä suorittamaan omia toimintoja ruudunpäivityksien välillä. Ydinjärjestelmän luokkakaavio löytyy kuvasta 17.



Kuva 17. Ydinjärjestelmä.

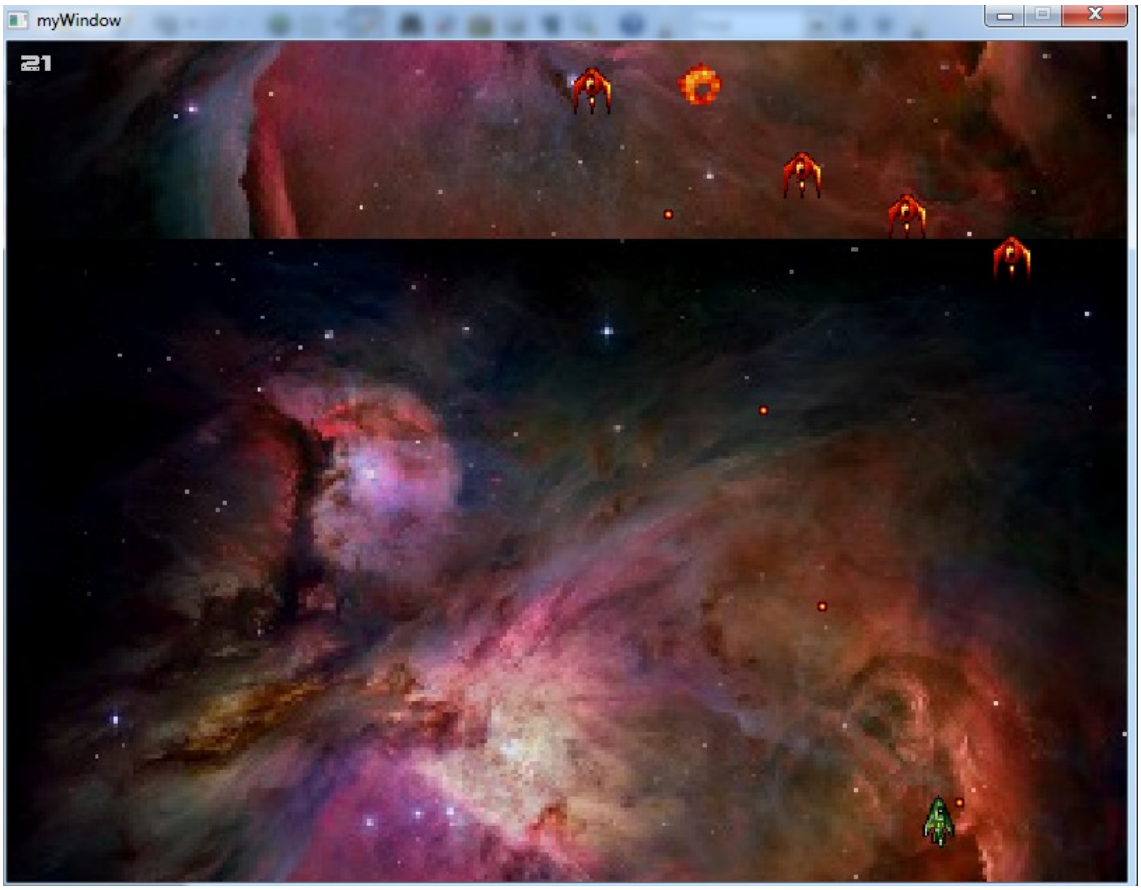
6 Testaus

Pelimoottori testattiin tekemällä peli pohjautuen vaatimusmäärittelyssä esiin tulleeseen pelisuunnitelmaan. Graafisen samankaltaisuuden lisäksi testissä käytettiin kaikkia pelimoottorin osa-alueita. Tämä takaa komponenttien toimivuuden, mutta myös täyttää vaatimusmäärittelyssä tehdyt vaatimukset.

Peli on ylhäältäpäin kuvattu avaruusammuskelupeli. Se koostuu kahdesta osasta: aloitusvalikosta [kuva 18] ja pelitilasta [kuva 19]. Aloitusvalikossa on pelin nimi (”Space Mutant”) sekä valinnat ”Start” ja ”Quit”. Valintanuolta siirretään nuolinäppäimistä haluttuun kohtaan ja Enter-näppäimestä valitaan kyseinen toiminto. Pelin ensimmäinen tila ei näppäinkomentojen lisäksi sisällä juuri toimintoja.

Kuva 18. Testipelin aloitusvalikko.

Pelitila on kahdesta tilasta toimintapitoisin. Pelaaja ohjastaa vihreää alusta nuolinäppäimillä samalla kun kuva tähtisumusta rullaa hitaasti alaspäin. Space-näppäimestä saadaan käyttöön aluksen tykit minkä tuloksena on näyttävä punainen sarjatulitus. Hetkeä myöhemmin ensimmäiset viholliset ilmestyvät ruudun ylälaidalta hienossa rivissä. Avaruuden sankari kiitää läpi avaruuden ampuen samalla galaksin hirvityksiä saaden aikaiseksi näyttäviä räjähdyksiä. Muutaman vihollisaallon jälkeen taso loppuu ja kuvaruutuun ilmestyy teksti tason läpäisystä.



Kuva 19. Testipelin taso 1.

Pelin kesto ei ollut muutamaa minuuttia pitempi, mutta riittävän pitkä esittämään pelimoottorin kyvyt. Kaikki piirrettävät komponentit ladattiin onnistuneesti peliin, ja ne näkyivät selvästi. Pelimoottori hallitsi hyvin yli 30 animoitua piirtokomponettia eivätkä uusien komponenttien luonti kesken pelin aiheuttanut ongelmia. Kontrollit olivat tarkat,

joskin peli hieman takerteli, kun nuolinäppäimiä painettiin samanaikaisesti (tämä tosin ei ollut pelimoottorin syytä vaan pelin toteutuksen). Äänentoisto ei missään vaiheessa takerrellut, ja monien äänitiedostojen toisto sujui mutkattomasti.

7 Kokemukset

Minua on aina kiehtonut tekniikka pelien takana, joten pelimoottorin teko tuntui luonnolliselta valinnalta insinöörityöaihetta valittaessa. Tiesin aihealueen olevan valtaisa; pelkästään pelimoottorin graafisesta järjestelmästä saisi aikaiseksi oman projektin. Koetin ratkaista ongelman tekemällä pelimoottorista mahdollisimman minimalistisen ja teknisesti yksinkertaisen. Ratkaisuista huolimatta aihe osoittautui liian laajaksi eikä tehdystä pelimoottorista tullut ihan niin kattavaa kuin olisin halunnut.

Alkuperäisen suunnitelman mukaan moottorissa olisi ollut reaaliaikanen törmäystenhallinta. Moottori olisi joka kierros tarkastanut jokaisen piirrettävän komponentin sijainnin ja laskenut mahdolliset kontaktipisteet. Käyttäjän määrittämien asetuksien mukaan moottori olisi joko hylännyt nämä tapahtumat tai lähettänyt niistä viestin kuuntelijaoliolle. Ongelmaksi nousi moottorin tapa hallita ja ylläpitää piirrettäviä komponentteja. Törmäyslaskentojen aikana olisi komponentit pitänyt käydä hakemassa yksitellen piirtolistasta. Tämä käytäntö olisi ollut liian hidas, ja se olisi hajoittanut liikaa järjestelmäarkkitehtuuria. Käyttöliittymäkomponenttien tapauksessa jouduin turvautumaan huonoihin käytäntöihin ja törmäyshallinnan lisääminen olisi huonontanut tilannetta entisestään.

Grafiikkajärjestelmän teko sen sijaan onnistui yllättävän hyvin. Olen tyytyväinen siitä, miten helposti uusia komponentteja pystyy luomaan. Sen lisäksi piirto-operaatio on äärimmäisen nopea esimerkiksi, 400 animoidun komponentin samanaikainen piirto ei tunnu tuottavan mitään ongelmia. Ainoa osa, mikä tuntuu kuluttavan laskentatehoa, on piirtokomponenttien skaalaus, joten ei ole suositeltavaa skaalata komponentteja lennossa, ainakaan kovin suuriksi.

Käyttöliittymäjärjestelmälle olisin toivonut enemmän monimuotoisuutta komentojen saralta. Ainoat komennot, jotka pelimoottori tulkitsee ovat hiiri ja näppäinkomennot, ja ne se saa SDL Event -oliolta. Näiden lisäksi se myös tulkitsee

käyttöliittymäkomponenttien komennot, mutta pohjimmiltaan nekin ovat peritty SDL Event -luokasta. Komennot lähetetään viesteinä käyttäjälle ja käyttäjän tehtäväksi jää purkaa viestistä saatu SDL Event-olio. Olisinkin halunnut, että käyttäjä pystyy määrittelemään, mitä näppäimiä ja komponentteja kuunnella.

8 Yhteenveto

Työssä selvitettiin, miten suunnitella ja toteuttaa yksinkertainen pelimoottori. Sen lisäksi tutkittiin ja selvitettiin, mitä useimmat pelimoottorit sisältävät ja mitkä ovat parhaimmat käytännöt moottoria toteuttaessa. Moottoria lähdettiin toteuttamaan puhtaalta pöydältä, ja toteutukseen kului noin 2–3 kuukautta. Tukena pelimoottorin teossa oli runsas ohjelmointitietämys sekä kokemus pelien toteutuksesta.

Moottorin toteutus onnistui hyvin, ja sen tuloksena syntynyt peli toimi moitteetta. Pelin luominen moottorilla onnistui näppärästi ja helposti, ja sen koko koodiriveissä oli lähes 500. Pelin kirjoittaminen kesti noin viisi tuntia. Peliä tutkittaessa voidaan päätellä moottorin täyttävän kaikki sille määrätyt vaatimukset.

Pelimoottorin teknilliset valinnat onnistuivat myös hyvin. SDL-pohjaiset kirjastot toimivat hyvin ja kääntyivät helposti myös Linux-alustoilla. Kirjastojen kääntäminen tuotti ajottain ongelmia varsinkin Windows-alustalla.

Suurimmaksi ongelmiksi nousi kuitenkin projektissa aiheen laajuus ja toteutusajan vähyys. Moottorista jouduttiin karsimaan ominaisuuksia, jotta se valmistuisi ajallaan. Tuotantoprosessia helpotti huomattavasti hyvät ohjelmointikäytännöt, ja esimerkiksi suunnittelumallit osoittautuivat korvaamattomiksi.

Toiseksi ongelmaksi nousi kokemattomuus isojen järjestelmien hallinnasta. Tietämättömyyteni arkkitehtuurisuunnittelusta pitkitti tuotantoprosessia jo entisestään. Kenties olisin saanut monimuotoisemman pelimoottorin aikaisemmaksi, jos olisin paneutunut arkkitehtuurisuunnitteluun paremmin.

Vaikka pelimoottori on hyvin yksinkertainen, on siinä kuitenkin paljon potentiaalia. Oliopohjainen ohjelmoiminen ja hyvät käytännöt tekivät moottorista helposti laajennettavan ja muokattavan, ja uskonkin sen olevan kovassa käytössä tämän työn jälkeen. Pelinsuunnittelijana minulla on varastossa iso liuta tekemättömiä pelejä, ja uskonkin tämän moottorin olevan oiva työkalu pelien teossa.

Lähteet

1 Video Game Sales Wiki. Videopeliteollisuus. Verkkodokumentti. <http://vgsales.wikia.com/wiki/Video_game_industry>. Luettu 16.3.2012.

2 Polson, Ken. 3.2012. Chronology of Arcade Video Games. Verkkodokumentti <http://vidgame.info/arcade/> Luettu 16.3.2012.

3 Reuters. 6.6.2011. Factbox: A look at the $65 billion video games industry. Verkkodokumentti <http://uk.reuters.com/article/2011/06/06/us-videogames-factbox-idUKTRE75552I20110606>. Luettu 16.3.2012.

4 Global Industry Analysts, Inc. 6.2009. Video games - a global strategic business report. Verkkodokumentti <http://www.strategyr.com/Video_Games_Market_Report.asp>. Luettu 16.3.2012.

5 Laukkanen, Tero. Modding scenes. 2005. Verkkodokumentti <http://tampub.uta.fi/tup/951-44-6448-6.pdf>. Luettu 16.3.2012.

6 Gregory, Jason. 7.10.2009. Game Engine Architecture. Luettu 26.3.2012.

7 Gamma, Helm, Johnson & Vlissides. 1995. Design Patterns. Luettu 26.3.2012.

8 Apogee Software, Ltd. 1994. Raptor: Call of the Shadows. Verkkodokumentti. <http://www.3drealms.com/raptor/index.html>. Luettu 7.5.2012.

Renderer-luokka

```
/*
 * s2dRenderer.h
 *
 *  Created on: 3.3.2012
 *      Author: Tommi Lassila
 */

#ifndef S2DRENDERER_H_
#define S2DRENDERER_H_

#include <core/s2dConfig.h>
#include <graphics/s2dSceneManager.h>
#include <graphics/drawable/s2dDrawable.h>


using namespace std;

namespace s2dGraphics {

class Renderer{
public:

	virtual ~Renderer();

	int setVideoMode(int width, int height, int bpp, Uint32 displayFlags);

	int drawAll();

	void cleanUp();
```

```
        static Renderer* instance();

        void setClearColor(Uint32 clearColor);
        void setWindowCaption(const char *title, const char *icon);

        SceneManager* getSceneManager();

protected:
        Renderer();

private:
        SDL_Surface *mScreen;
        Uint32 mClearColor;

        static Renderer* mInstance;

};

} /* namespace s2dGraphics */
#endif /* S2DRENDERER_H_ */
```

SceneManager-luokka

```
/*
 * s2dSceneManager.h
 *
 *  Created on: 18.3.2012
 *      Author: Tommi Lassila
 */

#ifndef S2DSCENEMANAGER_H_
#define S2DSCENEMANAGER_H_

#include <graphics/s2dScene.h>
#include <core/s2dConfig.h>

namespace s2dCore{ class EventManager; }

namespace s2dGraphics {

typedef std::map< const char * ,Scene*> s2dSceneMap;

class SceneManager {
public:

	virtual ~SceneManager();

	Scene* createScene(const char *sceneId);
	void destroyScene(const char *sceneId);
	Scene* getScene(const char *sceneId);

	const char* getCurrentSceneId();
	Scene* getCurrentScene();

	void setCurrentScene(const char *sceneId = NULL);
```

```
protected:
	SceneManager();
	static SceneManager* instance();
	void cleanUp();

private:
	static SceneManager* mInstance;
	const char *mCurrentSceneId;
	s2dSceneMap mSceneMap;

	friend class Renderer;
	friend class s2dCore::EventManager;
};

} /* namespace s2dGraphics */
#endif /* S2DSCENEMANAGER_H_ */
```

Scene-luokka

```
/*
 * s2dScene.h
 *
 *  Created on: 17.3.2012
 *      Author: Tommi Lassila
 */
#ifndef S2DSCENE_H_
#define S2DSCENE_H_
#include <core/s2dConfig.h>
#include <graphics/drawable/s2dDrawable.h>

using namespace std;
namespace s2dCore { class EventManager;}
namespace s2dGraphics {

typedef std::map< int ,std::vector<Drawable*> > s2dDrawMap;

class Scene {
public:
	Scene();
	virtual ~Scene();

	void push(Drawable *d, int layer);
	Drawable* pop(std::string id);
	void remove(Drawable *d);
	void cleanUp();

private:
	friend class Renderer;
	friend class s2dCore::EventManager;
	s2dDrawMap mDrawingQueue;
};
} /* namespace s2dGraphics */
```

```
#endif /* S2DSCENE_H_ */
```

FontManager-luokka

```
/*
 * s2dFontManager.h
 *
 *  Created on: Mar 27, 2012
 *      Author: Tommi Lassila
 */

#ifndef S2DFONTMANAGER_H_
#define S2DFONTMANAGER_H_

#include <core/s2dConfig.h>

namespace s2dCore { class Root;}
namespace s2dGraphics {
typedef std::map<const char*, Font*> s2dFontMap;

class FontManager {
protected:
	virtual ~FontManager();
	static FontManager* instance();
	int init();
	Font* loadFont(const char *fileUrl, int fontSize);
	void cleanUp();
	FontManager();

private:
	static FontManager* mInstance;
	s2dFontMap mFontMap;
	friend class ObjectFactory;
	friend class s2dCore::Root;
};
```

```
} /* namespace s2dGraphics */
#endif /* S2DFONTMANAGER_H_ */
```

Drawable-luokka

```
/*
 * sd2Drawable.h
 *
 *  Created on: 3.3.2012
 *      Author: Tommi Lassila
 */

#ifndef SD2DRAWABLE_H_
#define SD2DRAWABLE_H_

#include <core/s2dConfig.h>

namespace s2dGraphics {

class Drawable {
public:

	virtual ~Drawable(){};
	virtual void draw(SDL_Surface *dest) = 0;

	std::string getId() const;

	SDL_Point getPosition() const;
	SDL_Rect getSource() const;
	SDL_Surface *getSurface() const;

	eDrawableType getType() const;
	void setPosition(SDL_Point destination);

	bool isVisible() const;
	void setVisible(bool visible);
```

```
protected:
        Drawable();
        std::string mId;
```

```
        eDrawableType mType;

        SDL_Surface *mSurface;
        SDL_Rect mDestination;
        SDL_Rect mSource;

        bool mVisible;
};

}
 /* namespace s2dGraphics */
#endif /* SD2DRAWABLE_H_ */
```

Sprite-luokka

```
/*
 * s2dSprite.h
 *
 *  Created on: 6.3.2012
 *      Author: Tommi Lassila
 */

#ifndef S2DSPRITE_H_
#define S2DSPRITE_H_

#include <core/s2dConfig.h>
#include <graphics/drawable/s2dDrawable.h>

namespace s2dGraphics {

class Sprite : public Drawable {
public:
	virtual ~Sprite();
	virtual void draw(SDL_Surface *dest);
	virtual void initialize(SDL_Surface *image, SDL_Rect src, SDL_Point
				dest, const char* id);

	void scale(double x, double y);
	void move(int x, int y);
	SDL_Rect getModifiedSource() const;

protected:
	Sprite();
	SDL_Rect mModifiedSource;
	SDL_Surface* mModifiedSurface;
	SDL_Point mScale;
	friend class ObjectFactory;
};
```

```
} /* namespace s2dGraphics */
#endif /* S2DSPRITE_H_ */
```

AnimatedSprite-luokka

```
* AnimatedSprite.h
 *
 *  Created on: 6.3.2012
 *      Author: Tommi Lassila
 */

#ifndef ANIMATEDSPRITE_H_
#define ANIMATEDSPRITE_H_

#include "s2dSprite.h"

namespace s2dGraphics {

enum eAnimationMode {
	S2D_ANIM_MODE_REPEAT = 0,
	S2D_ANIM_MODE_REPEAT_LOOPING,
	S2D_ANIM_MODE_ONCE
};

enum eAnimationType {
	S2D_ANIM_TYPE_LEFT_TO_RIGHT = 0, S2D_ANIM_TYPE_RIGHT_TO_LEFT,
S2D_ANIM_TYPE_IDLE
};

struct AnimationProperties{
	int rows;
	int columns;
	int currentRow;
	int currentColumn;
	int animationInterval;

	eAnimationType type;
```

```
        eAnimationMode mode;
    };
```

```
class AnimatedSprite: public s2dGraphics::Sprite {
public:

	virtual ~AnimatedSprite();

	virtual void draw(SDL_Surface *dest);
	virtual void initialize(SDL_Surface *image, SDL_Rect src, SDL_Point
dest, const char* id);

	void setAnimProperties(int rows, int columns, eAnimationType type =
			S2D_ANIM_TYPE_LEFT_TO_RIGHT, eAnimationMode mode =
			S2D_ANIM_MODE_REPEAT, int animInterval = 0);
	void setTimeStep(Uint32 interval);
	void start();

	void setCurrentRow(int row);
	void setCurrentColumn(int column);

	bool isAnimationFinished();

protected:
	AnimatedSprite();
private:

	Uint32 static timeStepCallback(Uint32 interval, void *param);
	AnimationProperties *mAnimationProperties;
	SDL_TimerID mTimerId;

	friend class ObjectFactory;

};

} /* namespace s2dGraphics */
#endif /* ANIMATEDSPRITE_H_ */
```

UiComponent-luokka

```
/*
 * s2dUiComponent.h
 *
 *  Created on: Mar 30, 2012
 *      Author: root
 */

#ifndef S2DUICOMPONENT_H_
#define S2DUICOMPONENT_H_

#include <graphics/drawable/s2dSprite.h>
#include <core/s2dConfig.h>
#include <core/s2dEventListener.h>

namespace s2dGraphics {

using namespace s2dEvent;
using namespace s2dCore;

class UiComponent: public s2dGraphics::Sprite {
public:
	virtual ~UiComponent();
	void doClick();

    virtual void initialize(SDL_Surface *image, SDL_Rect srcStateUp, SDL_Rect
srcStateDown, SDL_Point dest, const char* id);

    void setSelected(bool state);
    bool isSelected();

protected:
	UiComponent();
```

```
private:
	bool mState;
```

```
        SDL_Rect mSourceStateDown;
        SDL_Rect mSourceStateUp;
        friend class ObjectFactory;

};

} /* namespace s2dGraphics */
#endif /* S2DUICOMPONENT_H_ */
```

Font-luokka

```
/*
 * s2dFont.h
 *
 *  Created on: Mar 27, 2012
 *      Author: Tommi Lassila
 */

#ifndef S2DFONT_H_
#define S2DFONT_H_

#include <graphics/drawable/s2dDrawable.h>
#include <core/s2dConfig.h>


namespace s2dGraphics {

class Text : public Drawable {

public:

	virtual ~Text();

	virtual void initialize(Font* font, SDL_Point dest, eRenderType type,
const char *id);

	void setFontColor(Uint8 red, Uint8 blue, Uint8 green);
	void setBackgroundColor(Uint8 red, Uint8 blue, Uint8 green);
	void setText(const char *text);

	const char *getText() const;

	virtual void draw(SDL_Surface *dest);
```

```
protected:
        Text();
```

```
private:

	Font *mFont;
	const char *mText;
	SDL_Color mFontColor;
	SDL_Color mBackgroundColor;
	eRenderType mRenderType;

	friend class ObjectFactory;
};
}

/* namespace s2dGraphics */
#endif /* S2DFONT_H_ */
```

ObjectFactory-luokka

```
/*
 * ObjectFactory.h
 *
 *  Created on: 7.3.2012
 *      Author: Tommi Lassila
 */

#ifndef GRAPHICSCONTENTFACTORY_H_
#define CONTENTFACTORY_H_

#include <graphics/drawable/s2dSprite.h>
#include <graphics/drawable/s2dAnimatedSprite.h>
#include <graphics/drawable/s2dText.h>
#include <graphics/drawable/s2dUiComponent.h>

#include <core/s2dConfig.h>
#include <graphics/s2dFontManager.h>

namespace s2dCore { class Root; }

namespace s2dGraphics {

typedef std::map< const char* ,SDL_Surface*> s2dSurfaceMap;

class ObjectFactory {
public:

	virtual ~ObjectFactory();

	Sprite* createSprite(const char* imgURL, SDL_Rect src, SDL_Point dest,
				const char *id = 0);
	AnimatedSprite* createAnimatedSprite(const char* imgURL, SDL_Rect src,
				SDL_Point dest, const char *id = 0);
```

```
Text* createText(const char* fontUrl, const char* text, int
                 fontSize,SDL_Point  dest,
                 eRenderType type, const char* id = 0);
```

```
        UiComponent* createUiComponent(const char* imgURL,
                                       SDL_Rect srcStateUp,
                                       SDL_Rect srcStateDown,
                                       SDL_Point dest, const char *id = 0);
        void cleanUp();

protected:
        ObjectFactory();
        static ObjectFactory* instance();

private:

        static ObjectFactory* mInstance;
        s2dSurfaceMap mSurfaceMap;
        SDL_Surface* loadImage(const char *imgURL);

        int mCreatedObjects;

        friend class s2dCore::Root;
};
} /* namespace s2dGraphics */
#endif /* GRAPHICSCONTENTFACTORY_H_ */
```

SoundManager-luokka

```
/*
 * s2dSoundManager.h
 *
 *  Created on: 5.4.2012
 *      Author: Tomppa
 */

#ifndef S2DSOUNDMANAGER_H_
#define S2DSOUNDMANAGER_H_

#include <SDL_mixer.h>
#include <core/s2dConfig.h>

namespace s2dCore {

struct s2dSoundData
{
	Mix_Chunk* data;
	int channel;
};

typedef std::map< const char* ,s2dSoundData> s2dSoundMap;

class SoundManager {
public:

	virtual ~SoundManager();

	int init(int flags);
	int openAudio(int frequency = MIX_DEFAULT_FREQUENCY, Uint16 format =
MIX_DEFAULT_FORMAT, int channels = 2, int chunksize = 1024);
	int playSound(const char* url, int loops = -1);
```

```
        int stopSound(const char* url);
        void loadSound(const char *url);
        void cleanUp();

protected:
        SoundManager();

        static SoundManager* instance();

private:
        static SoundManager *mInstance;
        s2dSoundMap mSoundMap;

        friend class Root;
};

} /* namespace s2dCore */
#endif /* S2DSOUNDMANAGER_H_ */
```

EventManager-luokka

```
/*
 * s2dEventManager.h
 *
 *  Created on: Apr 2, 2012
 *      Author: root
 */

#ifndef S2DEVENTMANAGER_H_
#define S2DEVENTMANAGER_H_

#include <core/s2dConfig.h>
#include <core/s2dEventListener.h>
#include <core/SDL_collide.h>
#include <graphics/drawable/s2dUiComponent.h>
#include <graphics/s2dSceneManager.h>

namespace s2dCore {

using namespace std;
using namespace s2dGraphics;

class EventManager {

protected:
	virtual ~EventManager();
	static EventManager* instance();

	void pollEvents();
	void setEventListener(EventListener* eventListener);

	EventManager();
private:
```

```
void pollUiComponents();

EventListener *mEventListener;
```

```
        static EventManager *mInstance;

        friend class Root;

};

} /* namespace s2dCore */
#endif /* S2DCOLLISIONMANAGER_H_ */
```

EvenListener-luokka

```
/*
 * s2dEventListener.h
 *
 *  Created on: Mar 29, 2012
 *      Author: Tommi Lassila
 */

#ifndef S2DEVENTLISTENER_H_
#define S2DEVENTLISTENER_H_

#include <SDL.h>
#include <core/s2dConfig.h>

namespace s2dCore
{

using namespace s2dEvent;

class EventListener
{
public:
	virtual void eventHappened(EventMessage eventMessage){};
	virtual ~EventListener(){};
};

}

#endif /* S2DEVENTLISTENER_H_ */
```

FrameListener-luokka

```
/*
 * s2dFrameListener.h
 *
 *  Created on: Mar 29, 2012
 *      Author: Tommi Lassila
 */

#ifndef S2DFRAMELISTENER_H_
#define S2DFRAMELISTENER_H_

namespace s2dCore
{

class FrameListener {
public:
	virtual ~FrameListener(){};

	virtual void frameStarted(int elapsedTime){};
	virtual void frameEnded(int elapsedTime){};

};
}



#endif /* S2DFRAMELISTENER_H_ */
```

Root-luokka

```
/*
 * Root.h
 *
 *  Created on: 6.3.2012
 *      Author: Tommi Lassila
 */

#ifndef ROOT_H_
#define ROOT_H_

#include <graphics/s2dObjectFactory.h>
#include <graphics/s2dRenderer.h>
#include <core/s2dFrameListener.h>
#include <core/s2dEventManager.h>
#include <core/s2dSoundManager.h>

namespace s2dCore {

using namespace s2dGraphics;
using namespace s2dEvent;

class Root {
public:
	virtual ~Root();

	int initSubsystems(Uint32 initFlags);

	int bigInit();
	void bigClean();

	void setRenderingFrameRate(int frameRate);
```

```
void start();
void quit();
Renderer *getRenderer();
```

```
        ObjectFactory* getObjectFactory();
        SoundManager* getSoundManager();

        static Root* instance();

        void setFrameListener(FrameListener* frameListener);
        void setEventListener(EventListener* eventListener);

protected:
        Root();

private:
        int mFrameRate;
        FrameListener *mFrameListener;
        eEngineState mState;
        static Root* mInstance;
        void pollEvents();

};
} /* namespace s2dGraphics */
#endif /* ROOT_H_ */
```

Config-lähdekoodi

```
/*
 * s2dConfig.h
 *
 *  Created on: 3.3.2012
 *      Author: Tommi Lassila
 */

#include <vector>
#include <map>
#include <iostream>

#include <SDL_image.h>
#include <SDL.h>
#include <SDL_ttf.h>
#include <SDL_rotozoom.h>

#ifndef S2DCONFIG_H_
#define S2DCONFIG_H_

typedef TTF_Font Font;

enum eEngineState{
	S2D_STATE_ERROR = -1,
	S2D_STATE_RUNNING = 0,
	S2D_STATE_QUIT,
	S2D_STATE_IDLE,
};
enum eRenderType{
	S2D_TEXT_SOLID = 0,
	S2D_TEXT_BLENDED
};
```

```
enum eDrawableType{
	S2D_TYPE_SPRITE = 0,
	S2D_TYPE_ANIMATED_SPRITE,
```

```
	S2D_TYPE_TEXT,
	S2D_TYPE_UI_COMPONENT
};

struct SDL_Point{
	float x,y;
};

namespace s2dEvent
{
	enum eEventType{
		S2D_EVENT_TYPE_INPUT = 0,
		S2D_EVENT_TYPE_UI,
	};

	struct EventMessage
	{
		void* event;
		eEventType eventType;
	};
}

#endif /* S2DCONFIG_H_ */
```

Lisäkirjastot

SDL.dll
SDL_ttf.dll
SDL_mixer.dll
SDL_image.dll
libwebp-2.dll
libvorbisfile-3.dll
libvorbis-0.dll
libtiff-5.dll
libSDL_gfx-13.dll
libpng15-15.dll
libogg-0.dll
libjpeg-8.dll
libfreetype-6.dll