Sami Koivisto

Android-peliohjelmointi

Metropolia Ammattikorkeakoulu
Tutkinto Insinööri (AMK)
Koulutusohjelma Tietotekniikka
Insinöörityö Android-peliohjelmointi
Päivämäärä 15.1.2012

Metropolia

Tiivistelmä

Tekijä(t) Otsikko Sivumäärä Aika	Sami Koivisto Android-peliohjelmointi 48 sivua 15.1.2012
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Miikka Mäki-uuro Lehtori Jorma Räty

Android on viimeisen muutaman vuoden aikana kasvanut uusi käyttöjärjestelmä, joka on suunnattu pääasiassa sulautetuille laitteille kuten älypuhelimille. Tämä työ tarkastelee Androidin soveltuvuutta ohjelmistokehitykseen erityisesti peliohjelmoinnin näkökulmasta.

Työ tarkastelee erityisesti Androidista asioita, jotka eroavat siitä mihin normaali PC-puolen Java-ohjelmoija on mahdollisesti tottunut. Käydään läpi taaksepäin yhteensopivuutta rikkovia ongelmia ja miksi kannattaa käyttää Androidin omia kirjastoja Sunin Java-kirjastojen sijaan.

Peliohjelmoinnin läpikäynti aloitetaan kertaamalla perusteet sekä vertaamalla sitä muihin ohjelmoinnin osa-alueisiin. Käydään läpi kuinka pelisilmukka kannattaa mahdollisesti rakentaa peliin sekä kuinka alustus- ja latausoperaatioita kannattaa käsitellä. Perehdytään hieman Androidin tarjoamaan kahteen eri grafiikkakirjastoon: xml-pohjaiseen käyttöliittymäkirjastoon sekä alemman tason OpenGL ES -kirjastoon, joka on pelejä varten ehdottomasti parempi kirjasto. Kirjastojen lisäksi käsitellään yleisimpiä graafisia käsitteitä, kuten ruudun päivitystä, piirto-luokkia, tekstuureja ja UV-koordinaatteja. Lopuksi käsitellään muutamaa Androidin tarjoamaa äänikirjastoa ja kuinka niitä kaikkia voidaan soveltaa pelissä erillaisiin tilanteisiin.

Pohjustuksen jälkeen keskitytään hieman Android-pelin optimointiin ja syihin, miksi se saattaa kärsiä suorituskykyongelmista. Selvitetään, mitä eroa on automaatti- ja manuaalioptimoinnilla sekä esitetään kummastakin esimerkkejä. Automaattioptimoinnissa tutustutaan Zipalign- ja Proguard-työkaluihin, jotka Androidin Eclipseen lisättävä lisäpalikka tarjoaa. Näiden lisäksi tutustutaan muutamaan eri manuaalioptimoinnin menetelmään. Java puolella ehdotetaan mahdollista ratkaisua Javan roskienkerääjää vastaan. Myös grafiikka puolella käydään läpi ohjeita piirtorutiinien nopeuttamiseksi.

Lopuksi tutustutaan työtä varten tehtyyn Android-peliin. Peli on yksinkertainen 2D-avaruusreaaliaikastrategiapeli, jossa on tarkoituksena suojella omia avaruusaluksia väistelemällä kohti tulevia asteroideja. Käydään läpi pelin rakennetta sekä tutustutaan sen tärkeimpiin luokkarakenteisiin, kuten suuntapiste-järjestelmään.

Avainsanat	Android, peliohjelmointi

Metropolia

Abstract

Author(s) Title Number of Pages Date	Sami Koivisto Android game programming 48 pages 15.1.2012
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-uuro, Senior Lecturer Jorma Räty, Senior Lecturer

Android is a brand new operating system designed for embedded system like smart phones and has had some significant growth within past few years. This work examines how suited the Android operating system is for software development especially when it comes to game development.

The work examines Android related matters which differ from what Java programmers on the PC-side of things might have gotten used to. Things like issues with the backward compability and why one should use Android libraries instead of Sun Java libraries are explained.

We recap the basics of game programming and compare it to other aspects of programming. Game loop and how to properly build one in your game will get explaned along side how initialization and loading should be handled. We delve deeper in to 2 different graphics libraries supported by Android: OpenGL ES and xml-based user interface library. Graphics terms like refresh-rate, draw-classes, textures and UV-coordinates will get explained. Lastly we explain how to use the audio libraries supported by Android properly in a game.

Afterwards we consentrate on how to optimize Android games and on some reasons why they might suffer from bad performance. We find out the differences between automatic and manual optimization and show few examples from each. Automatic optimizers provided by Eclipse plugins like Zipalign and Proguard will get explained. Also few ways to manually optimize your code are explained. One of these methods is used to fight the Java Garbage Collector and the others are for optimizing your graphics rutines.

Lastly we look at the Android game which was specificly built for this work. The game is a simple 2d space real time strategy game in which the user must protect their own ships by dodging the incoming asteroids. The framework and the structure of the game is explained along side some more specific libraries like the waypoint system.

Keywords	Android, Game programming

Sisällys

1 Johdanto

Tämän työn tarkoituksena on perehtyä Android-käyttöjärjestelmään ja tarkastella sitä peliohjelmoinnin näkökulmasta. Yhtenä päätavoitteena on ollut oppia aikaisemmista Android-kehityksen aikaisista virheitä ja rakentaa niiden antamia tietoja hyväksi käyttäen parempi Android-peli ja kirjastoluokkia, jotka tukevat toteutusta. Tarkoituksena on syventyä enemmän myös Androidiin käyttöjärjestelmänä ja tarkastella sen heikkouksia ja vahvuuksia, eikä pelkästään pelikehityksen kannalta, mutta kuitenkin pelinkehitystä silmällä pitäen. Näiden asioiden lisäksi tarkastellaan kuinka hyvin Android soveltuu pelikehitykseen ja mitä Androidilla on tarjottavana pelinkehittäjälle.

Työ käsittelee myös hieman optimointiohjeistusta. Kyseisen luvun tarkoituksena on perehdyttää lukija optimointimenetelmiin, jotka osoittautuivat tehokkaiksi ja toimiviksi. Osa näistä optimointimenetelmistä on tarkoitettu pääasiassa Android-sovelluksille, mutta yleiskäyttöisempiäkin optimointeja löytyy.

Viimeisenä käydään läpi itse työtä varten tehtyä peliä ja sen ominaisuuksia. Läpikäyntiin sisältyy mm. pelin kirjastoluokkien tarkastelua ja niiden toiminnallisuuden selittämistä. Kirjastoluokista käydään myös lyhyesti läpi ratkaisuja, kuten missä sitä käytettiin ja miten se vaikutti itsepelin toimintaan.

2 Android-käyttöjärjestelmän kuvaus

Android on alunperin 2007 julkaistu Open Handset Alliancen kehittämä mobiililaitteille suunnattu käyttöjärjestelmä.

Open Handset Alliancen ja sen suurimman tukijan Googlen tarkoituksena oli kehittää yleinen open source -käyttöjärjestelmä sulautetuille järjestelmille. Erityisesti Googlen tuki on tehnyt Androidista erittäin suositun käyttöjärjestelmän varsinkin älypuhelimissa, mutta ei ainoastaan niissä. Android-käyttöjärjestelmän voi nykyään löytää monesta eri laitteesta kuten taulutietokoneista, netbookeista, televisioista, digibokseista ja autoista.

Kyseisissä laitteissa Android on vielä ainakin Suomessa harvinaisuus, mutta ei luultavasti enää muutaman vuoden päästä. [1, s. 1-2.]

2.1 Kehitys

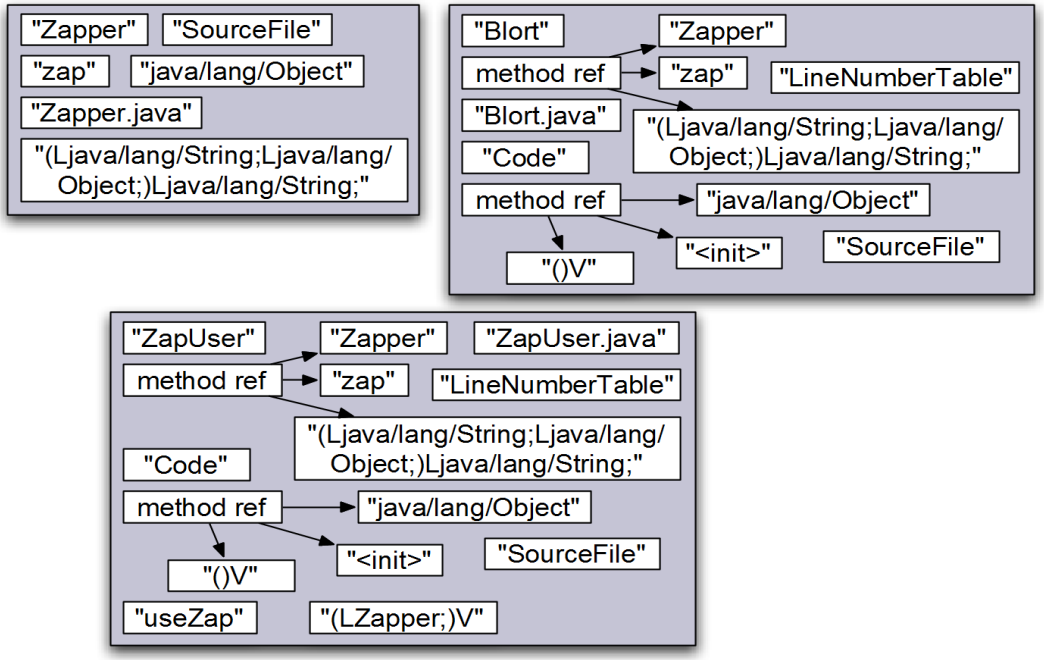
Androidin kehitys alkoi alunperin vuonna 2003 pienessä kalifornialaisessa yrityksessä nimeltä Android Inc. Yrityksen tarkoituksena oli kehittää fiksumpia mobiililaitteita, jotka olisivat enemmän perillä siitä, mitä käyttäjä siltä haluaa. Nykyään itsestäänselvyydet kuten GPS ja vapaasti muokattavissa oleva käyttöjärjestelmä olivat kehittäjien mielessä. Myöhemmin elokuussa 2005 Google osti koko Android Inc:in ja teki siitä tytäryhtiön. Näitä pieniä yksityiskohtia lukuun ottamatta Androidin kehityksestä ei tiedetä paljoa varsinkaan ennen sen lopullista julkaisua vuonna 2007. [2.]

2.2 Alustana

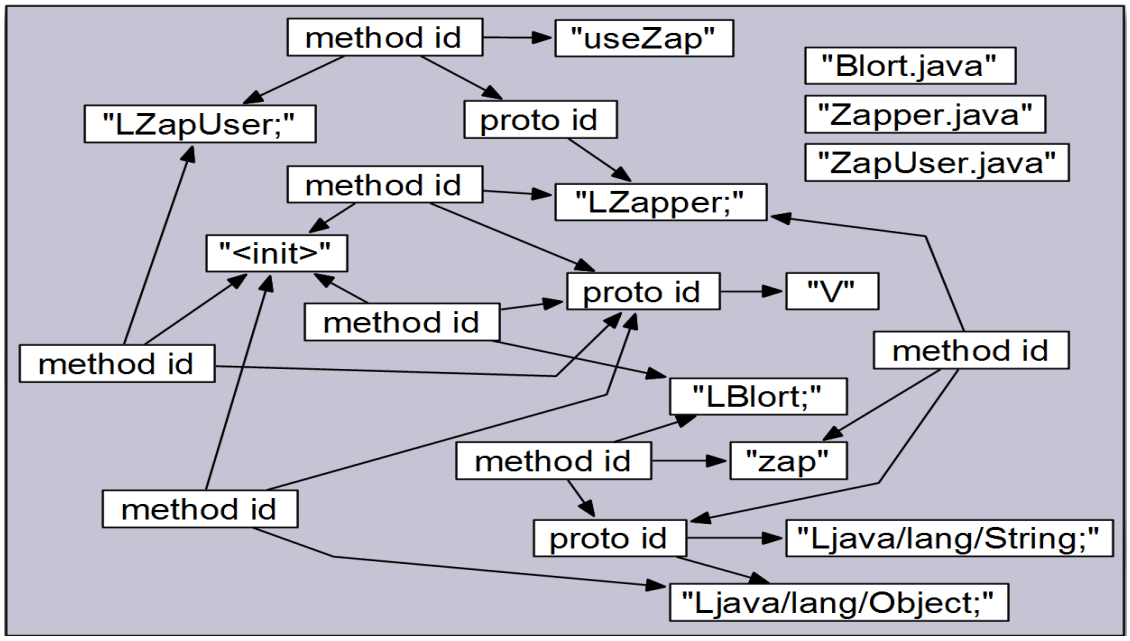
Android-käyttöjärjestelmä perustuu alunperin Linux Kernel -versioon 2.6. Kernel-versiota on sittemmin päivitetty muiden päivitysten ohella. Android tarjoaa avoimen ympäristön ohjelmistokehittäjille ja mahdollisuuden muokata suurta osaa sen koodipohjasta. Kehittäjän on mahdollista ylikirjoittaa, vaikka suuri osa Androidin standardikirjaston koodeista, jos näin näkee tarpeelliseksi.

Android käyttää Dalvik-virtuaalikonetta yhteystyössä just-in-time-kääntäjän kanssa ajaakseen käännettyä Java-koodia. Java on Android-käyttöjärjestelmän pääohjelmointikieli, ja suurin osa sen kirjastoista on kirjoitettu Javalla. Androidin Kernel-tason koodi on kuitenkin kirjoitettu C:llä [3.]. Dalvik muistuttaa virtuaalikoneena Javan normaalia virtuaalikonetta, mutta eroaa siitä muutamalla eri tavalla. Rakenteellisesti Dalvik on rekisteripohjainen virtuaalikone pinopohjaisen toteutuksen sijaan. Dalvikin dex-kääntäjä paketoi useita .class-tiedostoja yhdeksi .dex-tiedostoksi, jota se itse pystyy lukemaan, tätä ei kuitenkaan tehdä jokaisella .class-tiedostolle. Dalvik executable eli .dex-tiedosto rakentaa luokkarakenteen tiedostojen välille viittauksia, siinä missä normaali .class-tiedosto sisältää kaikki sen tarvitsemat tiedot. Tästä syystä luokat vievät vähemmän ajonaikaista likaista prosessikohtaista muistia samalla helpottaen käyttöjärjestelmää hallitsemaan muistinkäyttöään. Dex-tiedoston ja class-tiedoston eroista kertovat kuvat 1 ja 2. Dalvik muuttaa myös normaalin Java-tavukoodin vaihtoehtoiseen enemmän sille sopivampaan muotoon. Jokainen käynnissä oleva Android-sovellus saa oman ilmentymänsä Dalvikista. Eli jos yksi sovellus kaatuu, ei koko laitteen virtuaaliympäristö kaadu.

Dalvik myös rajoittaa paljon yksi sovellus saa käyttöönsä CPU aikaa ja muistia. Muistia yksi sovellus saa noin 20 MB:tä, joka ei ole paljon. Tästä syystä varsinkin pelisovelluksilla voi olla muistin kanssa ongelmia. Tarkka määrä vaihtelee laitteesta toiseen. Tehokkaammat antavat yksittäiselle sovellukselle enemmän resursseja. Alustansa takia Dalvik ei myöskään pysty lisäämään muistin määrää kovalevy-swappauksella. [4.]



Kuva 1. Normaali .class-tiedostojen linkitysrakenne [4.].



Kuva 2. .dex-tiedoston luokkalinkityksen rakenne [4.].

Vaikka Android-sovellukset on pääasiassa tarkoitus kirjoittaa Javalla, ei tämä tarkoita, että se olisi ainoa vaihtoehto. Androidin Java niin kuin normaali Java SE 6 tarjoaa natiivitason yhteyden C- ja C++ -koodiin JNI (Java Native Interface) -rajapinnan kautta. JNI-rajapinnasta on lisää optimointiluvussa.

Käyttöjärjestelmänä Android toimii suurimmalla osalla kännyköistä hyvin ja luotettavasti sekä tarjoaa yleiset yläpuhelinominaisuudet kuten 3G:n, Wifin, multitouch-kosketusnäytön, kameran yms. Android-kirjastot tarjoavat tuen suurelle määrälle erilaisia lisälaitteita huolimatta siitä, onko niitä kyseisessä laitteessa. On siis hyvä ottaa huomioon kehityksen yhteydessä myös lisälaitteet, joita testilaitteissa ei ole.

Androidin tuetut laitteet jakavat keskenään joitain yleispiirteitä käyttöjärjestelmästä johtuen ja ne on jaettu yleisesti seuraaviin lohkoihin.

- Bootloader – alustus Boot-osion latauksen laitteen käynnistyksen yhteydessä
- Boot-osio (Boot image) – Kernel ja RAMdisk
- Järjestelmäosio (System image) – Android-käyttöjärjestelmätaso ja ohjelmistot
- Dataosio (Data image) – kaikki käyttäjädata
- Palautusosio (Recovery image) – tiedostot järjestelmänuudelleenrakennusta tai päivitystä varten.
- Radio-osio (Radio image) – Radio-pinotiedostot.

Testilaitetta hankkiessa ja ohjelmistoja kehittäessä Androidille on kuitenkin huomioitava, että useilla kännyköillä on omat ongelmansa, ja ne saattavat olla joissain tilanteissa ylipääsemättömiä ongelmia. Samanhintainen laite suurin piirtein samoin osin ei välttämättä toimi yhtä hyvin. Joitain laitekohtaisia ongelmia on onneksi helppo korjata, sillä ajavan laitteen voi aina tarkistaa ajon aikana koodiesimerkin 1 mukaisella tavalla. [1, s. 2-3.]

```
if (android.os.Build.MODEL.equals(”Nexus+one”))
{
        // Laite kohtanen koodi tähän.
}
```

Koodiesimerkki 1. Laitekohtainen koodi.

Android-käyttöjärjestelmäversiot kannattaa myös pitää mielessä, sillä versiosta toiseen hyppääminen saattaa aiheuttaa ohjelmistoissa arvaamattomia ongelmia.

2.3 Versioerot

Android-käyttöjärjestelmästä julkaistaan uusia versioita silloin tällöin. Nämä versiot yleensä sisältävät uusia ominaisuuksia, parannuksia ja bugi-korjauksia. Yleisin syy kuitenkin päivityksiin ovat uudet ajurit uusille laitteille. Tästä syystä useimmat isot päivitykset on koordinoitu uutta versiota tukevan älypuhelimen tai minkä tahansa muun laitteen julkaisun yhteyteen. Vanhat Android-laitteet eivät kuitenkaan aina ole päivitettävissä uuteen versioon, koska niitten laitteisto ei välttämättä ole täysin yhteensopiva tai se ei täytä jotain muuta vaatimusta. Käyttöjärjestelmän versio voi siis vaihdella huomattavasti kännykältä toiselle. [1, s. 15-16.]

Ohjelmistokehityksessä käyttöjärjestelmäversio on erittäin tärkeää ottaa huomioon ja asiat, kuten kehityksessä olevan ohjelman minimiversio tulisi asettaa kehitysympäristöön samantien, jotta ei käytetä vahingossa kirjastoja, joita kohdelaite ei tue. Minimiversion sekä muut versiointiin liittyvät määritellään Android-projekteissa tiedostossa nimeltä AndroidManifest.xml. AndroidManifest on Androidin oma asetustiedosto, joka määrittää sovelluksen oikeudet käyttöjärjestelmässä ja miten sovelluksen tulisi käyttäytyä tilanteissa, kuten jos puhelin alkaa soida sovelluksen ollessa päällä.

Android käsittelee versioitaan sisäisesti API-tasoina. Kukin API-taso on yksi suuri käyttöjärjestelmäpäivitys ja sisältää usein isoja parannuksia tai muutoksia. Seuraavat versiot tarkoittavat seuraavaa API-tasoa. [5.]

- Android OS 1.0 , API level 1
- Android OS 1.1 , API level 2
- Android OS 1.5 , API level 3
- Android OS 1.6 , API level 4
- Android OS 2.0 , API level 5
- Android OS 2.0.1 , API level 6
- Android OS 2.1.x , API level 7
- Android OS 2.2.x , API level 8
- Android OS 2.3.0-2 , API level 9
- Android OS 2.3.3-4 , API level 10
- Android OS 3.0.x , API level 11

- Android OS 3.1.x , API level 12
- Android OS 3.2 , API level 13.

Koodiesimerkki 2 näyttää, kuinka yksinkertaista minimiversion asettaminen on. Toinen asia, mitä kannattaa ottaa huomioon versioiden välillä, on se, että Android on taaksepäinyhteensopiva. Tämä tarkoittaa sitä, että jos sovellus on tehty oikein, se toimii myös tulevilla laitteilla ongelmitta. On kuitenkin pidettävä mielessä, että toisinpäin ei aina ole näin.

```
<uses-sdk android:minSdkVersion="4" />
```

Koodiesimerkki 2. AndroidManifest.xml:n minimiversiomääritys.

Koodiesimerkin 3 koodi on pätkä SQLite-tietokantakoodista, jossa tietokanta avataan, mutta jätetään sulkematta. Tietokanta tulisi aina sulkea käytön jälkeen, mutta se ei aiheuta ohjelmiston kaatumista. Jos se unohtuu versiossa 2.0 ja jos kyseisen koodin ajaa versiossa 1.6 ,ei mene pitkään, kun ohjelma kaatuu näin pieneen virheeseen.

```
public void update(String s){
        open().addTo(s);
}
public ScoreDatabaseAdapter open() throws SQLException {
        if(dbHelper == null){
                dbHelper = new ScoreDatabaseHelper(context);
        }
        database = dbHelper.getWritableDatabase();
        return this;
}

public void close() {
        dbHelper.close();
}
```

Koodiesimerkki 3. Mahdollinen versiokonflikti.

Ongelmat voivat kuitenkin mennä toiseenkin suuntaan. Android-version 1.5 ja 1.6 välillä tapahtui isoja muutoksia. 1.6:n mukana tuli asioita kuten ohjelman sisäiset marketostot ja resurssikäsittelyn täysmuutos, kun 1.6 oli ensimmäinen Android-versio, joka tuli moniin eri näyttökokoihin. Tämä tarkoittaa sitä, että jos ohjelma tukee minimiversioltaan jotain muuta versiota kuin 1.6, oletetaan kaikkien resurssien toimivan vanhalla

tavalla, eikä ongelmia synny. Vaikka minimiversioksi asetettaisiin versio 1.6, suurin osa 1.6-laitteista osaa käsitellä resursseja niin kuin ennenkin jollei sitä toisin AndroidManifestissa määritetä. Näin ei kuitenkaan tapahdu enää 2.0:sa ja myöhemmissä versioissa. Nämä versiot olettavat, että jos minimiversio on asetettu resurssimuutoksen jälkeen, tulee uutta käytäntöä käyttää. Tämä johtaa helposti siihen, että resursseja ei näissä tapauksissa käsitellä oikein.

Suoranaiset versiobugit on myös otettava huomioon. Vaikka Android-versio 2.2 tukee uudempaa OpenGL ES 2.0 -rajapintaa, kyseinen rajapinta kärsii pahasta VBO-bugista kyseisessä Android-versiossa. Tämä tarkoittaa sitä, että jos haluaa tukea Opengl ES 2.0:aa, tulee Android-version olla melkein 2.3. OpenGL ES 2.0 toki toimii jo 2.2:lla, mutta ilman VBO-tukea varsinkin pelit kärsivät performanssista. Tähän helppona ratkaisuna on tietenkin käyttää vanhempaa OpenGL 1.0 ES 1.0 -rajapintaa, joka ei kärsi samasta ongelmasta. [6.]

2.4 Standardikirjasto

Android sisältää erittäin laajan Googlen kehittämän standardikirjaston, sekä suurimman osan Javan omasta SE-standardikirjastosta. Nyrkkisääntönä on kuitenkin se, että jos Android tarjoaa siihen oman versionsa, käytetään sitä eikä Javan omaa. Android-versio on tehokkaampi jos vaihtoehtoja on koodiesimerkin 4 mukaisesti. JNI:n alla toimivalle natiivikoodilla on myös oma standardikirjastonsa perinteisten C/C++ -kirjastojen lisäksi.

```
System.out.println(”Hello World”);    // java versio
Log.v(”Helloworld app”, ”Hello World”); // android versio
```

Koodiesimerkki 4. Normaali Java ja Androidin logi tulostus.

Android-standardikirjasto tarjoaa järjettömän ison kasan eri ominaisuuksia, joita kehittäjät voivat käyttää. Näihin kuuluvat melkein kaikki kännykän oheislaitteet sekä asiat kuten Wifi, 3G , bluetooth ja SQLite. Standardikirjasto tarjoaa myös helpon rajapinnan OpenGL:n graafiseen kehitykseen sekä oman helppokäyttöisemmän, mutta ei niin tehokkaan natiivi-rajapintansa. Kirjasto sisältää kaiken äänikirjastoista tapahtumakäsittelyyn. Nämä perinteisemmät kirjastot ja ominaisuudet ovat varsin käyttäjäystävällisiä, eikä suureen osaan niistä tarvita suurta perehtymistä. Ei-käyttäjäystävälliset rajapinnat

kuten Activity, Intent, Service tai kaikkia näytä käyttävät ominaisuudet kuten In App purchase tarvitsevat paljon enemmän perehtymistä.

Koodiesimerkit 5 ja 6 ovat yksinkertainen esimerkki, kuinka standardikirjaston tarjoama näytönkosketustapahtumien käsittely tapahtuu. Esimerkissä 6 onTouch-funktio saa Androidilta kutsun joka kerta, kun jokin kosketustapahtuma tulee ilmi. Kehittäjällä on sitten mahdollisuus käsitellä tapahtuma, miten haluaa. Palautettava boolean arvo kertoo käyttöjärjestelmälle, käsiteltiinkö tämä tapahtuma loppuun. Oikein toteutettu tapahtuman käsittely on tärkeää varsinkin Android-peleissä, joissa tarvitaan yleensä erittäin tarkkaa kykyä hallita, mitä ikinä näytöllä tapahtuukaan.

```
import android.view.MotionEvent;
import android.view.View;

/**
 * Games input system.
 *
 * @author Sami
 *
 */
public class InputHandler {

	private static InputHandler instance;
	private boolean isPressed = false;
	private boolean isReleased = false;

	/**
	 * latest cursor location.
	 */
	public int x,y;

	private InputHandler(){

	}

	public static InputHandler instance(){
		if(instance == null) {
				instance = new InputHandler();
		}
		return instance;
	}

	public void onTouch(View v, MotionEvent event){
```

```
                x = (int) event.getRawX();
                y = (int) event.getRawY();

                switch(event.getAction()){
                case MotionEvent.ACTION_DOWN:
                                isPressed = true;
                                break;
                case MotionEvent.ACTION_UP:
                                if(isPressed) isReleased = true;
                                isPressed = false;
                                break;
                default:

                                break;
                }
        }

        public boolean isPressed(){
                return isPressed;
        }

        public boolean isReleased(){
                if(isReleased){
                                isReleased = false;
                                return true;
                }
                return false;
        }
}
```

Koodiesimerkki 5. Yksinkertainen kosketus tapahtuma käsittely.

```
public class Launcher extends Activity implements OnTouchListener {
                @Override
                public boolean onTouch(View v, MotionEvent event) {
                                InputHandler.instance().onTouch(v, event);
                                return true;
                }
}
```

Koodiesimerkki 6. Tapahtuman kuuntelija.

Androidista puuttuu perinteisten Java -kirjastojen osalta pääasiassa kaikki com.sun ja sun alkuiset kirjastot jotka ovat osa suljettua lähdekoodikantaa. Tärkeimmät näistä kirjastoista on kuitenkin kirjoitettu uudestaan jo löytyvät lähes samassa muodossa osana Androidin omia kirjastoja. Avoimeen lähdekoodiin perustuvista org.apache kirjastoista puuttuu käytännössä kaikki paitsi kirjanpito ja http osuudet. Myös avoimeen lähdekoodiin perustuvista javax -kirjastoista on mukana noin puolet. Javaxista puuttuvat osat

ovat pääasiassa laitehallintaan ja käyttöliittymiin liittyviä kirjastoja. Javaxista puuttuu myös javax.naming ja javax.annotation kirjastot mistä johtuen suurin osa isoista Java -kirjastoista kuten Hibernate tuskin tulee ikinä toimimaan Androidilla. Androidilla Javax -kirjastosta löytyy myös muutama ylimääräinen grafiikkakirjasto, joita ei normaalista JRE toteutuksesta löydy.

Suurin osa puuttuvista kirjastoista ei ole Androidissa mukana siksi, että ne eivät toimisi vaan siksi, että ne eivät veisi turhaa pinomuistia järjestelmältä. Tästä syystä useimmat puuttuvat kirjastot on mahdollista halutessa liittää projektiin, jos niille näkee tarvetta ja lähdekoodit löytää jostain.

2.5 Erot muihin kehitysalustoihin

Android eroaa kehitysalustana PC:stä lähinnä siinä, että se on paljon rajoittuneempi. Vaikka Android tarjoaa erittäin hyvät kirjastot ja kehitysympäristön, se on silti sulautettu käyttöjärjestelmä ja tuo mukanaan kaikki siihen liittyvät ongelmat. Rajoitteet kuten rajattu muisti ja suorituskyky tappavat suurimman osan toiveet hienoista peleistä nopeasti. Tämä ei tietenkään tarkoita, että näitten ongelmien yli ei voisi päästä. Se on kuitenkin suuri aloituskynnys. Siinä missä PC:llä pelikehittäjät saavat vapaat kädet tehdä ja käyttää resursseja, miten haluavat, samaa ei voi sanoa Androidista.

Jo pelkkä Android-asennuspaketin maksimikoko, joka oli viime tarkistuksella 16 mb, ei ole kovin suuri ja tulee pelissä, jossa on paljon resursseja, erittäin nopeasti täyteen. Paketin koko ongelmaan on onneksi lähitulevaisuudessa tulossa korjaus, jossa koko nousee aina 4 GB:een asti.

Suurin ongelma Androidille kehittäessä kuitenkin useimmiten on kynnys hankkia pätevä Android-puhelin. Melkein missä vain löytyy PC, jolla pelikehitys onnistuu, mutta Android tarjoaa kännykättömille vain emulaattorin, joka ikävä kyllä vaatii teho-PC:n, jos sitä haluaa käyttää Android-kehitykseen, varsinkin jos kyseessä on pelikehitys.

3 Peliohjelmoinnin perusteet ja rakenne Androidilla

Luku 3 käsittelee yleisesti, mitä on peliohjelmointi ja miten se eroaa muista ohjelmoinnin haaroista. Samalla se käsittelee hyviä yleiskäytäntöjä. Näiden asioiden lisäksi syvennytään hieman Android-kohtaisiin asioihin, jotka kannattaa pitää mielessä, kun ryhtyy omaa projektia kasaamaan Android-alustalle.

3.1 Mitä on peliohjelmointi?

Yksinkertaisimmillaan peliohjelmointi on pelisovellusten ohjelmointia. Peliohjelmointia pidetään yleensä yhtenä ohjelmoinnin erikoisosa-alueena samalla tavalla kuten esim. tietokanta- tai tietoturvaohjelmointia. Verrattuna muihin ohjelmoinnin osa-alueisiin pelit ovat erittäin monimutkaisia ohjelmia ja saattavat olla suurelle osalle vaikeimpia ohjelmia, joita he ovat ikinä kirjoittaneet. Toki kaikki on suhteellista. Tekstin käsittelyohjelma on varmasti monimutkaisempi kuin perinteinen matopeli. [7, s. 11.]

Siinä missä suurin osa muista erikoistumisosa-alueista keskittyy yhteen, peliohjelmointi ammentaa hyvin montelta osa-alueelta. Peliohjelmointi yhdistelee logiikkaa, matematiikkaa, grafiikkaa, ääniä, tietokantoja ja tilannetapahtumien käsittelyä. Vaikka yksi henkilö tuskin osaa kaikkia osa-alueita, on ne silti jonkun projektissa osattava.

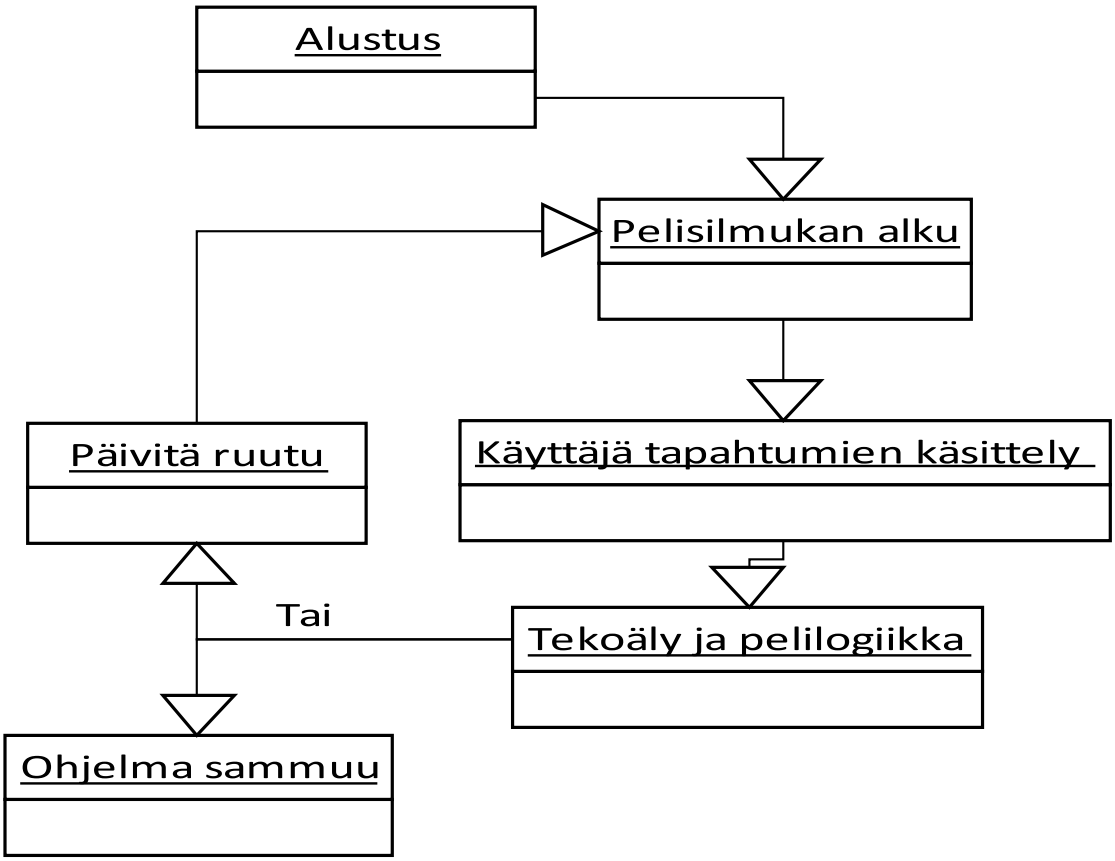
3.2 Perusteet

Kun peliohjelmoinnin aloittaa ensimmäisen kerran, on hyvä pyrkiä irrottautumaan joistain perinteisempien ohjelmien tavoista ja opetella asiat yksi kerrallaan. Suurin haaste aloittavalle peliohjelmoijalle on oppia uusi tapa ohjelmoida, joka tukee enemmän reaaliaikaohjelmia kuin tapahtumiin heräävä ohjelmia, joihin he ovat aikaisemmin tottuneet.

Pelien loogisen rakenteen tutkiminen on yleensä hyvä aloituspiste ennen kuin aloittaa oman pelin kirjoittamisen. Muiden ohjelmista saa hyvin ohjeistusta, miten asiat yleensä kannattaisi tehdä, jotta peliä ei tule suunniteltua tai koodattua huonosti. [7, s. 12-13.]

Pelisilmukka on peleissä yleisesti käytössä oleva rakennemenetelmä, missä pelin tila päivittyy jatkuvasti pyöriessään silmukassa päivittäen samaa tai samantapaista koodia, kunnes ohjelma sammuu. Pelisilmukan päätarkoituksena on kuitenkin pitää päivitys tasaisena riippumatta siitä, onko käyttäjätapahtumia tehty. Itse olen tottunut käyttämään

kuvan 1 esimerkin mukaista pelilooppia. Ideana on pitää pelilooppi pyörimässä niin kauan, kuin peli on päällä ja poistua siitä vasta, kun pelin pitää sammua. Alustusosiolla yleensä tarkoitetaan kaikkea, mitä tarvitsee tehdä ennen kuin siirrytään lopullisesti peli-looppiin. Tämä yleensä sisältää asiat kuten olioalustukset, tekstuurien lataukset ja muut pidempään kestävät operaatiot. Alustukset tehdään perinteisimmin näyttämällä latausruutu ja hoitamalla latausoperaatiot taustalla tai lataamalla asiat hitaammin samalla, kun jotain vähemmän raskasta on ruudulla, kuten aloitusvalikko tai vastaava.



Kuva 3. Yksinkertaistettu pelilooppi.

Peliloopin aloituspiste tapahtuu joka kierroksen alussa, jolloin tarkastellaan yleensä pelin senhetkinen tila, joka sitten vaikuttaa kyseisen kierroksen toteutukseen. Tämä yleisesti toteutetaan switch case -rakenteella, ja se sisältää yleensä tilat kuten peli käynnissä ja peli pysäytetty. Tämä näkyy koodiesimerkissä 7.

```
private void step(){
                long duration = (time = System.currentTimeMillis()) - laststep;
                laststep = time;
                if(duration > 100){
                                duration = 100;
                }
```

```
                delta = duration/16.0f;

                switch(state){
                case GAME:
                                game_step();
                                break;
                case PAUSED:
                                pause_step();
                                break;
                case INIT:
                                if(m_renderer.isInitialized())init_step();
                                break;
                }
}
```

Koodiesimerkki 7. Peliloopin tilarakenne.

Loppu peliloopin sisällöstä riippuu paljolti siitä, missä tilassa peli on, mutta pääasiallisesti se kulkee seuraavasti:

- Tarkastetaan tapahtumat, eli onko esim. jotain näppäintä painettu.
- Käyttäydytään tapahtuman perusteella ja päivitetään pelilogiikka.
- Päivitetään pelinäkymä tai poistutaan pelistä.

Monimutkaisuudestaan huolimatta pelit harvemmin eroavat perusrakenteestaan harvoin. Siksi vaikka peli on kuinka yksinkertainen tahansa siinä on aina opittavaa, jopa kokeneet peliohjelmoijat palauttavat rakenteita mieleensä muutaman vuoden välein rakentamalla jonkin pienen pelin kuten tetriksen tai matopelin.

3.3 Grafiikka ja sen rajoitteet

Grafiikka on osa peliohjelmointia huolimatta siitä onko peli iso tai pieni. Tässä luvussa on tarkoitus tarkastella yleisesti pelien grafiikkaa ja siten hieman perehtyä Androidin kahteen eri grafiikkarajapintaan.

Peligrafiikan päätarkoitus on kertoa käyttäjälle, mitä pelissä tapahtuu. Pelinäkymä päivittyy yleisesti yhden peliloopin laskentaosioiden päätteeksi. Ruudun päivitys tapahtuu yleensä 30-60 kertaa sekunnissa riippuen paljolti siitä, kuinka kauan yhteen loogiseen päivitykseen ja ruudun piirtämiseen kuluu aikaa [7, s. 11]. Ruudun päivitys tulisi olla ainakin yli 20 kertaa sekunnissa, jotta voidaan puhua suhteellisen sujuvasta ruudun päivityksestä.

Akkukäyttöisillä laitteilla, kuten Android-kännyköillä kannattaa kuitenkin välttää ylipäivittämästä ruutua, jos vain on mahdollista, sillä näyttö on usein se laite, joka vaatii eniten sähköä. Ruudun jatkuva päivittäminen nopeuttaa akun kulutusta.

Viimeinen peliloopissa tapahtuva asia eli ruudunpäivitys on rakenteeltaan tärkeä osa kaikkia pelejä. Tästä syystä ruudunpäivitys tulisi miettiä ja suunnitella niin, että rakenne on juuri sopiva, eikä aiheuta ongelmia tai hidasta kehitystä. Yleinen hyvä rakenteellinen ratkaisu on luoda jokaisesta ruudulle piirrettävästä elementistä oma piirtoluokka, joka toteuttaa piirron halutulla tavalla. Näitä piirtoluokkia käsittelee yleensä yksi pääpiirtofunktio, joka iteroi läpi kaikki piirrettävät elementit koodiesimerkin 8 mukaisesti.

```
@Override
public void onDrawFrame(GL10 gl) {
              gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
              gl.glLoadIdentity();

              synchronized(this){
                            for(Drawable var : drawables){
                                          var.draw(gl);
                            }
              }
}
```

Koodiesimerkki 8. Perinteinen piirto-olioiden iterointi.

Piirto-olioita on yleensä kahta eri tyyppiä. Efektioliot, jotka eivät ole kiinni missään sitä määrittävässä oliossa vaan toteuttavat ruudulla tapahtuvia efektejä pelkästään niille annettujen tietojen varassa. Esimerkiksi tämän tapainen piirto-olio voi olla sellainen, jonka tarkoituksena on piirtää pelissä liikesuuntanuolet tai räjähdysefektin. Kummatkin näistä tarvitsevat vain sijainnin ja voivat sen jälkeen toimia itsenäisesti.

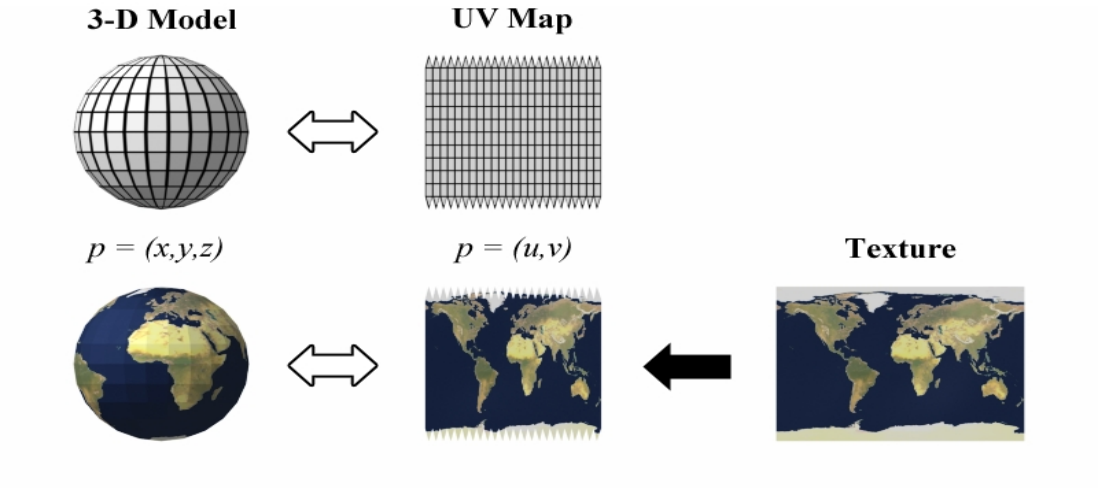
Toisentyyppinen olio on sellainen, joka on kiinni jossain sen piirtoparametreja määrittävässä oliossa. Näin tehdään, koska on hyvän käytännön vastaista sekoittaa piirto- ja logiikkakoodeja samoihin luokkiin. Tämä tekee koodista myös paljon siistimmän. Tämäntyyppiset piirto-oliot käsittelevät yleensä isäntäoliotaan kahdella eri tavalla: joko pitämällä vain yhden isäntäolion per piirto-olio tai listaamalla ison kasan isäntäolioita ja iteroimalla kaikki läpi piirron yhteydessä. Jos listaus on mahdollista, tämä tulisi olla ensimmäinen ratkaisu joka ainoa kerta. Listaus soveltuu erityisesti, jos pitää piirtää suuri määrä samantapaisia objekteja. Listaustapa on myös paljon tehokkaampi tapa piirtää. On myös hyvä pitää mielessä, jos piirto- ja logiikkarutiinit toimivat eri säikeissä. Silloin operaatiot pitää synkronoida, jotta data ei varmasti muutu piirron yhteydessä.

Yhden piirto-olion piirtäminen voi tapahtua esimerkiksi koodiesimerkki 9:n mukaisella tavalla.

```
/**
 * Preset draw function called by the renderer object.
 *
 * @param gl - OpenGL 1.0 context.
 */
public final void draw(GL10 gl){
                if(!active) return;
                preDraw(gl);
                onDraw(gl);
                postDraw(gl);
}

protected void preDraw(GL10 gl){
                gl.glPushMatrix();
                gl.glTranslatef(host.x, host.y, 0);
                TextureManager.bindTexture(gl, texture);
}

protected void onDraw(GL10 gl){
                gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, safe().getTexture(KEY));
                gl.glVertexPointer(3, GL10.GL_FLOAT, 0, safe().getVertex(KEY));
                gl.glRotatef(host.angle, 0, 0, 1);
                gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, safe().getSize(KEY)/3);

                gl.glColor4f(0, 0, 1, 1);
                gl.glVertexPointer(3, GL10.GL_FLOAT, 0, safe().getVertex(DIRECTION));
                gl.glDrawArrays(GL10.GL_LINES, 0, safe().getSize(DIRECTION)/3);
                gl.glColor4f(1, 1, 1, 1);
}

protected void postDraw(GL10 gl){
                gl.glPopMatrix();
}
```

Koodiesimerkki 9. Piirto-olioiden toteutukset.

Teksturointi on tietokonegrafiikan termi, jolla tarkoitetaan johonkin muotoon piirrettyä kuvaa. On tämä muoto sitten 2d tai 3d. Tekstuurit luetaan yleensä suoraan sisään kuva tiedostosta ja piirretään johonkin muotoon ruudulle. Tekstuureita voidaan käyttää monella tapaa saadakseen aikaan haluttu efekti tai näkymä käyttäjälle. Animaatioiden tai kuvakokonaisuuksien rakentaminen monesta eri tekstuurista on varsin yleistä, eikä tekstuurien muokkaaminen jälkikäteen koodissa ennen piirtoa ole harvinaisuus [8.]. Tekstuureita ei ole tarvetta aina piirtää kokonaisuudessaan, vaan tarvittu osa voidaan valita UV-koordinaateilla. UV-kartoitus on tapa, jolla 2d-kuva voidaan kartoittaa 3d-mallin päälle kiinnittäen määrätyt kuvan kohdat määrättyyn 3d-mallin kohtaan esimerkiksi kuvan 2 mukaisella tavalla. Näitä kartoitettuja pisteitä kutsutaan UV-koordinaateiksi.



Kuva 4. UV-kartoitettu pallo.

3.3.1 Android-käyttöliittymäkirjasto

Android tarjoaa kehittäjille käyttöliittymäkomponenteille tarkoitetun 2d-grafiikkakirjastoapin. Tämä graafinen API tarjoaa kehittäjälle suurimman osan perinteisistä piirtofunktioista, joka mahdollistaa tekstuurien piirtämisen ruudulle ilman suurempaa vaivaa juuri sinne, minne kehittäjä ne haluaa. Tämä grafiikka-API on pääasiassa tarkoitettu normaaleja käyttöliittymäkomponentteja varten. Tästä syystä sitä myös käytetään suurimmassa osassa Android-ohjelmistoja. Käyttöliittymäorientaatio näkyy myös siinä, että Androidin käyttöliittymäkirjastolla tehtyjä käyttöliittymiä voidaan kirjoittaa suoraan xml-muotoon joko käsin tai Eclipsen Android-plugin-työkalun avustuksella.

Kuinka tälläinen käyttöliittymiä varten tehty grafiikkakirjasto sopii pelikehitykseen? Ei hyvin. Kirjasto soveltuu toki hyvin yksinkertaisten pelien kehitykseen, mutta helposti tulee vastaan tämän kirjaston suurimmat ongelmat: se ei ole laitteistokiihdytetty eikä se tarjoa hirveästi ominaisuuksia. Tämä tarkoittaa sitä, että Android-laitteen suorituskyvyn rajat tulevat nopeasti vastaan eikä sillä voi tehdä monimutkaisia graafisia manipulaatioita. Kirjasto kuitenkin soveltuu hyvin siihen, mihin se on tehty: peleistä käyttöliittymiin. Kirjastolla voi esimerkiksi tehdä pelille aloitus- ja pysäytysvalikot ja taas käyttää monimutkaisempaa OpenGL-kirjastoa itse pelitoteutukseen. Jos pelinkehitys ja OpenGL ovat kuitenkin kummatkin uusia asioita, Androidin käyttöliittymäkirjaston käyttö saattaa helpottaa kehitystä, jos performanssista ei tule ongelma. [9.]

3.3.2 OpenGL

OpenGL (**Open G**raphics **L**ibrary) on alustariippumaton graafinen 2d- ja 3d-rajapinta [10.], joka tarjoaa hyvin alhaisen tason yhteyden piirtää ja muokata grafiikkaa. Laitteistokiihdytyksen ja alatason yhteytensä takia OpenGL on huomattavasti tehokkaampi ja monipuolisempi API kuin Androidin käyttöliittymiin tarkoitettu 2d-kirjasto, myös käyttöliittymissä. Kaikista eduista huolimatta tämä tietenkin tarkoittaa sitä, että OpenGL:n käyttö on huomattavasti vaikeampaa ja vie enemmän aikaa kuin yleisempi ja helppokäyttöisempi Android-käyttöliittymäkirjasto.

OpenGL:stä on muutama eri versio sekä sivuhaara erilaisille alustoille. Android käyttää OpenGL:n sulautetuille järjestelmille tarkoitettua OpenGL ES-haaraa. OpenGL ES (**OpenGL** for **E**mbedded **S**ystems) eroaa päähaarasta hyvin vähän ja sisältää lähes kaikki samat ominaisuudet kuin päähaara muutamaa poikkeusta lukuun ottamatta. Jos jompikumpi haaroista sujuu, ei toisen kanssa pitäisi tulla ongelmia.

Android tukee OpenGL ES:n kolmea eri versiota: ES1.0:aa, ES1.1:tä ja ES2.0:aa. ES1.0 on kaikista versioista tuetuin ja sisältää aika pitkälti kaikki ominaisuudet, joita moni pelinkehittäjä Androidilla tarvitsee. ES1.0 vastaa suurin piirtein normaali OpenGL:n versiota 1.3. ES1.1 on tuettu kaikissa Android-puhelimissa, joiden versio on 1.6 tai uudempi. Tämä tarkoittaa käytännössä kaikkia puhelimia, jotka on valmistettu viimeisen muutaman vuoden sisällä ja noin 99 % kaikista Android-puhelimista [11]. ES1.1 eroaa 1.0:sta hyvin vähän, mutta tuo muutaman tärkeän ominaisuuden tuen, joka suuresti tehostaa piirtoa varsinkin peleissä. Näistä ominaisuuksista tärkein on paranneltu VBO (Vertex Buffered Object) -tuki. VBO:t ovat OpenGL:ssä käytettyjä pistekoordinaattilistoja, joiden tarkoituksena on nopeuttaa pisteiden piirtoa laitteen ruudulle. VBO:n luonnista kertoo koodiesimerkki 10. Kannattaa myös huomioida, että Androidin OpenGL-rajapinta pyörii omassa säikeessään riippumatta muista tekijöistä. Tästä syystä pelisilmukka tulee eroamaan hieman aikaisemmin esitellystä ja vaatii enemmän huolta, että piirto ja laskenta eivät tapahdu samalla hetkellä.

```
FloatBuffer vertexBuffer;

float r = 10;
float vertices[] = {
               -r, -r, 0,
                r, -r, 0,
               -r,  r, 0,
                r,  r, 0
```

```
};

ByteBuffer bb = ByteBuffer.allocateDirect(4 * vertices.length);
bb.order(ByteOrder.nativeOrder());
vertexBuffer = bb.asFloatBuffer();
vertexBuffer.put(vertices);
vertexBuffer.position(0);
```

Koodiesimerkki 10. VBO:n luonti.

ES2.0 eroaa kahdesta versioedeltäjästään paljon ja lisää samalla paljon uusia ominaisuuksia. Versio ES2.0:n merkittävimpänä ominaisuutena on ohjelmoitava shader-tuki, joka mahdollistaa paljon monipuolisemman erikoisefektien käytön ilman, että täytyy turvautua vartavasten luotuihin tekstuureihin, jotka vievät näytönohjaimen muistia. ES2.0 myös heittää suurimman osan vanhemmista OpenGL-käytännöistä päälaelleen, minkä takia osa aikaisemmasta osaamisesta menee hukkaan. Kaikkien muutosten lisäksi ES2.0 on huomattavasti vaikeakäyttöisempi kuin edeltäjänsä, mutta ehdottomasti sen arvoinen, jos sen kaipaamille ominaisuuksille on tarvetta. Android-versio 2.2 ja uudemmat tukevat ES2.0:aa, mutta versio 2.2:n VBO-tuki on rikki, joten suositeltu versio on 2.3 ja eteenpäin. Jos vanhemman tyylin OpenGL on tuttu ja ES2.0-ominaisuuksille ei ole tarvetta, kannattaa luultavasti pidättäytyä vanhassa ja tutussa. [6.]

3.4 Äänikirjastot

Androidin standardikirjasto tarjoaa 2 eri äänikirjastoa kehittäjille, jotka soveltuvat hyvin pelin kehitykseen. Ulkoisille kirjastoille ei ole tästä syystä juuri tarvetta.

Ensimmäinen näistä kirjastoista on MediaPlayer. Kuten nimikin kertoo, tämä mediasoitinrajapinta tarjoaa tuen niin musiikin kuin videon toistoon ja nauhoittamiseen. Pelikäytössä MediaPlayer soveltuu erityisen hyvin taustamusiikin ja muiden pidempien äänitiedostojen soittamiseen. Koska MediaPlayer ei lataa koko tiedostoa muistiin vaan streamaa niitä pyydettäessä, se ei sovellu erityisen hyvin nopeasti muuttuviin tilanteisiin, joissa esimerkiksi musiikin sijainti tai kokotiedosto muuttuu jatkuvasti lennosta. [12.]

Toinen näistä kirjastoista on SoundPool. SoundPool on toteutettu täyttämään juuri sen aukon, minkä MediaPlayer jättää auki: nopeat ja lyhyet äänitiedostot. SoundPooliin voi ladata kasan pieniä klippejä, jotka voidaan soittaa erittäin nopeasti soittopyynnöllä. Huonona puolena SoundPool ei huoli isoja äänitiedostoja ja aiheuttaa helposti ongelmia kehittäjälle. Jos klippi on liian iso, se pitäisi silti soittaa tarpeeksi nopeasti tilanteen tullessa [13].

Koska kumpikin kirjasto toimii itsenäisesti, paras vaihtoehto pelinkehitykseen on käyttää kumpaakin tarpeen mukaan. Kannattaa myös pitää mielessä, että alustuksia ja klippejä ladatessa alkuperäinen lataus saattaa olla suhteellisen raskas operaatio. Se on kannattavinta tehdä alustusoperaatioiden yhteydessä.

4 Android-alustan optimointi

Optimointi tarkoittaa koodiin suorituskyvyn parantamista ohjelman toimivuuden parantamiseksi. Pelit ovat yleensä erittäin suorituskykyvetoisia ohjelmia. Pelien tarvitsee usein pyöriä erittäin tasaisella päivitysnopeudella ja toteuttaa tuhansia operaatioita monta kertaa sekunnissa ilman, että käyttäjä huomaa viivettä päivityksessä. Tästä johtuen optimointi on erityisen tärkeää ottaa pelikehityksessä huomioon jo aikaisessa vaiheessa ja pitää mielessä, vaikka sille ei juuri kyseisellä hetkellä olisikaan tarvetta. Android-pelien optimointi ei eroa merkittävästi muun tyyppisten pelien optimoinnista. Kuten useimmilla muillakin alustoilla Android-optimointi on täysin kiinni siitä, mitä kannattaa optimoida ja mitä resursseja voidaan käyttää, jotta ohjelma toimisi jossain toisessa kohdassa paremmin. Tämä resurssien paikasta toiseen siirto on yleisin tapa optimoida koodia. Yleensä myös se on helpoin tapa.

Optimointi jaetaan yleensä kahteen leiriin: automaattiin ja manuaalioptimointiin. Automaattioptimointia tapahtuu yleensä koodin kääntäjän toimesta tai JIT (**just-in-time compilation**) -sovellusten toimesta eikä siihen yleisesti ole paljon kehittäjällä vaikuttamista [14.].

Manuaalioptimointi on kehittäjän kannalta yleensä tärkeämpää, sillä siihen voi itse vaikuttaa enemmän. Manuaalioptimoinnilla tarkoitetaan yleensä jotain menetelmää tai tapaa muokata koodia niin, että se toimii paremmin. Yksinkertaisimmillaan manuaalioptimointi voi olla huonon koodin toteuttamista paremmalla tavalla. Tässä luvussa käydään läpi ainakin kaksi manuaalioptimoinnin menetelmää, jotka ovat JNI ja Javan roskienkerääjän parempi hallinnointi.

4.1 Suoraan tuetut optimointisovellukset

Android SDK tarjoaa kaksi eri automaattioptimointisovellusta kehittäjille, jotka toimivat hyvin yhdessä ja tarjoavat myös ominaisuuksia optimoinnin ulkopuolelta.

4.1.1 Zipalign

Zipalign on automaattinen Android .apk -paketin optimointijärjestelmä, joka sisältyy Eclipsen ADT-liitentään. Zipalign ajetaan automaattisesti, kun Eclipsen ADT-liitennän export wizard ajetaan ja sille välitetään paketin private-salausavain.

Zipalignin päätarkoitus on asetella kaikki .apk-paketin sisältämä raaka binääridata 4 tavurajoitteiseksi, jotta se on ajon aikana helpompi ja nopeampi lukea ohjelmaan sisään. Pääasiassa tämä tarkoittaa sitä, että ohjelma käyttää vähemmän muistia ja käynnistyy nopeammin. Parhaimpana puolena Zipalignissa on se, että siinä ei ole haittapuolia. [15.]

4.1.2 Proguard

Proguard on suoraan Androidin ADT Eclipse -liitännän sisältämä obfuskointi-ohjelma [16]. Vaikka obfuskointi ei ole optimointia, Proguard-automaatti optimoi, mitä pystyy ja poistaa turhaa koodia obfuskoinnin aikana. Toisin kuin Zipalign, Proguard ei ole automaattisesti mukana Eclipsen paketin rakennuksessa, mutta sen voi asetustiedostoja muokkaamalla saada tapahtumaan automaattisesti paketin rakennuksen yhteydessä.

Perusasetuksillaan Proguard saattaa kuitenkin olla varsin epävakaa optimoinneissaan. Tästä syystä asetustiedostoa kannattaa muokata sen verran, että Proguard käsittelee koodista vain ne kohdat, jossa ei synny luokkayhteyksissä epäselvyyksiä. Erityisesti ulkoiset kirjastot on hyvä jättää Proguard-käsittelyn ulkopuolelle, vaikka ne olisikin mahdollista käydä läpi, jos ei muuten niin ongelmien ja virheiden välttämiseksi.

Koodiesimerkki 11:n mukainen Proguardin asetustiedosto estää Androidin omien rajapintaluokkien obfuskoinnin sekä estää tilanteet, joissa kirjastokoodi saattaisi aiheuttaa

ongelmia. Esimerkiksi estetään natiivi c/c++:aan viittaavien metodien obfuskoinnin, jotta yhteys natiivitason kirjastoihin säilyy.

```
-optimizationpasses 5
-dontusemixedcaseclassnames
-dontskipnonpubliclibraryclasses
-dontpreverify
-verbose
-optimizations !code/simplification/arithmetic,!field/*,!class/merging/*

-keep public class * extends android.app.Activity
-keep public class * extends android.app.Application
-keep public class * extends android.app.Service
-keep public class * extends android.content.BroadcastReceiver
-keep public class * extends android.content.ContentProvider
-keep public class * extends android.app.backup.BackupAgentHelper
-keep public class * extends android.preference.Preference
-keep public class com.android.vending.licensing.ILicensingService

-keepclasseswithmembernames class * {
    native <methods>;
}

-keepclasseswithmembernames class * {
    public <init>(android.content.Context, android.util.AttributeSet);
}

-keepclasseswithmembernames class * {
    public <init>(android.content.Context, android.util.AttributeSet, int);
}

-keepclassmembers enum * {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}

-keep class * implements android.os.Parcelable {
  public static final android.os.Parcelable$Creator *;
}
```

Koodiesimerkki 11. Proguardin asetustiedosto proguard.cfg

4.2 Oman projektin optimointi

Omassa projektissa optimointi kannattaa keskittää mahdollisiin ongelmakohtiin koodissa, eikä mikro-optimoida jokaista pienintäkin kohtaa. Laitteesta riippuen näitä ongelmakohtia voi olla useampiakin. Esimerkkinä yhdestä tällaisestä ongelmasta, mikä useimmille Android-kehittäjille tulee vastaan, on Javan roskienkerääjä.

4.2.1 Java GC

Java Garbage Collector (GC) tai roskienkerääjä on Java-ohjelmointikielen käyttämä automaattinen muistinhallintajärjestelmä, jonka tarkoituksena on tarkkailla, mitä olioitten varaamaa muistia tulisi vapauttaa? Perinteinen Java-roskienkerääjä eroaa Androidin

vastaavasta lähinnä rakenteellisesti johtuen Androidin omasta Dalvik-virtuaalikone arkkitehtuurista. Siinä missä normaali PC-ympäristön Java-virtuaalikoneesta on käynnissä vain yksi instanssi, on taas Androidin Dalvik-virtuaalikoneesta käynnissä jokaista sovellusta varten oma instanssinsa. Jokainen näistä Dalvik-virtuaalikoneista sisältää oman keon, jota sen oma roskienkerääjä tarkastelee ja siivoaa. [4.]

Normaalissa PC-ympäristössä Javan roskienkerääjä toimii ongelmitta, eikä aiheuta mitään ongelmia. Samaa ei voi aina sanoa, kun kehitetään Androidille johtuen lähinnä paljon rajoitetummasta resurssimääristä. Androidin käyttämä Java-roskienkerääjä toimii ongelmitta, eikä vaadi yleensä huomiota tai käytön optimointia, kun olioita ei luoda uusia suuria määriä eikä niillä ole vaihtuvuutta. Jos vaihtuvuutta esiintyy, kannattaa pyrkiä uudelleen käyttämään olioita mahdollisimman paljon. Näin vältetään roskienkerääjän aiheuttamat satunnaiset suorituskyvyn pudotukset.

Ratkaisu roskienkerääjään on Object Pool-suunnittelumalli eli olioiden varastointi [17]. Käytännössä tämä tarkoittaa sitä, että käytöstä poistuneet olioita ei jätetä roskienkerääjän kerättäviksi vaan varastoidaan tehdasluokkaan, kunnes kyseisen tyyppisestä oliosta tarvitaan uusi ilmentymä. Kun uudelle olioilmentymälle on tarvetta asetetaan vanhaan varastoituun olioon uuden tarvittavan olion tiedot ja annetaan se takaisin käyttöön. Tällä tavalla estetään roskienkerääjän turha läpikäynti ja uusien olioiden luonti, joilla se on kallis operaatio. Jokaista oliota ei kuitenkaan kannata varastoida, sillä jos niitä käytetään aniharvoin tai niiden luonti ei ole raskasta, saattaa se johtaa vain suorituskyvyn menetykseen.

Koodiesimerkit 12 ja 13 esittelevät yhden luokan Olio-varastointitoteutuksen. Vec2f on vektoriluokka, joka on jatkuvassa käytössä eri puolilla ohjelmaa. Tästä syystä oli järkevää varastoida. Esimerkin 13 VectorFactory-luokka lisää luokkakohtaista lisätoteutusta jo aiemmin määriteltyyn yliluokkaan BaseFactory. VectorFactory-luokka määrittää itsensä myös singleton-toteutukseksi, joten siihen on helppo päästä käsiksi tarpeen vaatiessa, kuten esimerkki 14 kuvastaa.

```
/**
 * Sub-class for ObjectFactory.
 *
 * Main purpose is to handle any sort of object it is given.
 * Recycle and reuse those objects.
 *
 * @param <T> type of objects this class should handle.
```

```
 */
public class BaseFactory<T> {

		/**
		 * Big list of objects that can be reused.
		 */
		private List<T> m_free_objects = new ArrayList<T>();

		/**
		 * Reuse an object.
		 *
		 * @return object to use. or null if none is available.
		 */
		public T assign(){
			if(!m_free_objects.isEmpty()){
				T m = m_free_objects.get(0);
				m_free_objects.remove(m);
				return m;
			} else {
				return null;
			}
		}

		/**
		 * Set the given object for reuse.
		 *
		 * @param f
		 */
		public void free(T f){
			m_free_objects.add(f);
		}

		/**
		 * Returns the current stack of free objects.
		 *
		 * @return
		 */
		public List<T> stack(){
			return m_free_objects;
		}
}
```

Koodiesimerkki 12. Yhtä oliotyyppiä käsittävän objectpool-luokan rakenne.

```
public class VectorFactory extends BaseFactory<Vec2f>{

		private static VectorFactory m_instance;

		private VectorFactory(){

		}

		public static VectorFactory instance(){
			if(m_instance == null){
				m_instance = new VectorFactory();
			}
			return m_instance;
		}

		public Vec2f assign(){
			Vec2f v = super.assign();
			if(v == null){
				v = new Vec2f();
			}
			v.x = 0;
			v.y = 0;
			return v;
		}

		public Vec2f assign(float x, float y){
			Vec2f v = super.assign();
			if(v == null){
```

```
                                        v = new Vec2f();
                        }
                        v.x = x;
                        v.y = y;
                        return v;
                }

                /**
                 * Wipes this singleton clean.
                 */
                public void clear(){
                        stack().clear();
                        m_instance = null;
                }
}
```

Koodiesimerkki 13. Esimerkkiä 12 toteuttava luokka.

```
Vec2f first = m_vectors.assign();
Vec2f second = m_vectors.assign();
for(ShipTemplate friendlies : ships){
    for(ShipTemplate baddies : enemies){
        /*
         * Simple bounding box check.
         * faster then the conclusive check performed afterwards
         * if this test is passed.
         */
        if(friendlies.x + friendlies.shieldRadius < baddies.x - baddies.shieldRadius
                || friendlies.x - friendlies.shieldRadius > baddies.x + baddies.shieldRadius
                || friendlies.y + friendlies.shieldRadius < baddies.y - baddies.shieldRadius
                || friendlies.y - friendlies.shieldRadius > baddies.y + baddies.shieldRadius
        ) continue;

        first.x = friendlies.x;
        first.y = friendlies.y;
        second.x = baddies.x;
        second.y = baddies.y;
        first.subs(second);
        float dist = first.length();
        if(dist < friendlies.shieldRadius+baddies.shieldRadius){
            first.normalize();
            if(friendlies.collision(baddies,first)){
                baddies.redirect(GameRenderer.scrWidth);

                if(friendlies.dieOnCollision){
                    friendlies.path.finished();
                    if(active != null && active.equals(friendlies)){
                        active = null;
                        m_renderer.selector.setActive(false);
                    }
                }
            }
        }
    }
}
VectorFactory.instance().free(first);
VectorFactory.instance().free(second);
```

Koodiesimerkki 14. Objectpooling-käyttöesimerkki.

Oliovarastointia on mahdollista käyttää myös rajoittuneen Androidin ulkopuolella muissa Java-ohjelmissa, mutta ei ole suositeltavaa, sillä haittapuolena se lisää yleistä muistin käyttöä johtuen pelkästään siitä, että ei-käytössä olevia olioita saattaa kertyä helposti enemmän kuin niitä tarvitaan. Samainen tilanne voi johtaa myös muistivuotoihin.

4.2.2 JNI

JNI tai Java Native Interface on Java rajapinta, jolla voidaan käsitellä kirjastoja, jotka on kirjoitettu muilla ohjelmointikielillä kuten c:llä, c++:lla tai assemblerilla. JNI on hyvä tapa uudelleen käyttää jo aikaisemmin kirjoitettuja c tai c++ -kirjastoja ilman, että niitä kirjoitetaan uudestaan Javalla. Tämä mahdollistaa myös kirjastojen ja ominaisuuksien käytön, jotka eivät ole tuettuja tai mahdollisia Javassa.

Lisämahdollisuuksien lisäksi JNI soveltuu hyvin optimoimaan laskennallisesti raskaita operaatioita, koska ne voidaan ajaa suorituskykyisemmässä c- tai c++ -koodissa. On hyvä kuitenkin huomioida, että JNI:n läpi tehdyt natiivikutsut ovat n. 5 kertaa raskaampia kuin normaalit Java-funktiokutsut, jotka helposti tappavat suurimman osan saavutetusta hyödystä. Tästä syystä JNI:tä tulee aina käyttää harkiten. Erinomainen käyttötarkoitus Android-pelissä JNI:lle voisi olla esimerkiksi raskas fysiikan laskenta. Hyvänä puolena JNI:n alle ajettavalla koodilla on oma kekonsa, joka mahdollistaa suuremman muistimäärän käytön Androidilla, jossa kullakin sovelluksella on Dalvikissa määritetty maksimimuistin määrä. Suositeltavaa on kuitenkin, että muistialueita jaetaan tarpeen mukaan, jotta vältytään turhilla JNI-rajapinnan läpi kulkevilta funktiokutsuilta.

Androidilla käytettynä JNI-kirjastojen kääntäminen vaatii Linux-käyttöjärjestelmän, joka saattaa vaikeuttaa kehitystä.

4.3 Graafinen optimointi

Graafinen optimointi on usein yksi vaikeimmista optimoinnin osa-alueista, sillä se vaatii monesti paljon suurempaa asian osaamista kuin alkuperäisessä toteutuksessa. Tässä luvussa keskitytään pääasiassa muutamaan Androidin alla toimivaan OpenGL-optimointitapaan. Optimointi keskittyy pääasiassa OpenGL:ään siksi, että Androidin käyttöliittymäkirjaston komponentteja on melkein mahdotonta optimoida siirtymättä OpenGL:ään.

Helpoin tapa on noudattaa jo Peliohjelmointi-luvun grafiikkaa ja sen rajoitteita kappaleen ohjeita listoittamalla piirto-olioita niin, että yleinen OpenGL-kutsujen määrä pysyy mahdollisimman pienenä. Samalla pyritään varastoimaan ja käyttämään VBO:ita vii-

saasti, eikä luoda uutta joka piirtoa varten. VBO-varastoinnista kannattaa katsoa luvusta 5.

Jos kehitysalustana käytettävä Android-puhelin tukee OpenGL ES versiota 1.1, voidaan VBO:n käyttöä tehostaa entisestään lataamalla tarvittavat VBO:t pysyvästi videomuistiin. Täten nopeutetaan niiden suoritusten aikaista hakua. Haittapuolena videomuistiin lataaminen kasvattaa useasti yleisiä latausaikoja.

Koodiesimerkki 15 lataa sille määritetyn FloatBuffer-olioon rakennetun VBO:n ja lisää sen videomuistiin ja palauttaa siihen viittaavan int-muuttuja-arvon samalla tavalla kuin tekstuureja ladattaessa. Kuten tekstuurien yhteydessä myös videomuistiin ladatut VBO:t tulee vapauttaa jossain pisteessä. Videomuistiin ladatut VBO:t noudattavat samoja piirteitä kuin tekstuurien käsittely videomuistissa, eli ne täytyy joka näytön tilamuutoksen välillä poistaa vanhat ja ladata uudestaan. Koodiesimerkki 16 kuvaa koodia, joka poistaa kaikki m_vertexpointer -listan sisältämät VBO-viittaukset.

```
/**
 * Turn build FloatBuffer vertex coordinate list into a GPU memory pointer.
 *
 * @param gl - OpenGL 1.0 context.
 * @param key - key string to later get the vertex pointer. if set to null the pointer is not
saved.
 * @param vertexBuffer - should contain the coordinates for the VBO.
 * @return built vertex VBO pointer.
 */
public final int toVertexPointer(GL10 gl, String key, FloatBuffer vertexBuffer){
    if(gl instanceof GL11){
        // this thing is completely untested. so... continue here ;>
        GL11 gl11 = (GL11) gl;
        int vertexBufferIndex = 0;
        int[] buffer = new int[1];
        gl11.glGenBuffers(1, buffer, 0);
        vertexBufferIndex = buffer[0];
        gl11.glBindBuffer(GL11.GL_ARRAY_BUFFER, vertexBufferIndex);
        gl11.glBufferData(GL11.GL_ARRAY_BUFFER
                          , vertexBuffer.capacity() * 4
                          , vertexBuffer
                          , GL11.GL_STATIC_DRAW);

        if(key != null){
            m_vertexpointers.put(key, vertexBufferIndex);
        }

        return vertexBufferIndex;
    } else {
        return 0;
    }
}
```

Koodiesimerkki 15. VBO:n tallentaminen videomuistiin.

```
public final void deletePointers(GL10 gl){
    if(gl instanceof GL11){
```

```
        GL11 gl11 = (GL11) gl;

        if(!m_vertexpointers.isEmpty()){

            for(Integer i : m_vertexpointers.values()){
                int[] array = new int [1];
                array[0] = i;
                gl11.glDeleteBuffers(1, array, 0);
            }
        }
        m_vertexpointers.clear();
    }
}
```

Koodiesimerkki 16. VBO:n vapauttaminen videomuistista.

OpenGL ES 1.1 -toteutus eroaa jossain määrin normaalista 1.0-toteutuksesta. Koodiesimerkki 17 näyttää toteutuksen, joka toteuttaa saman piirron eri tavalla riippuen OpenGL-versiosta. Hyvänä puolena 1.1-toteutuksen käyttö jossakin ongelmakohdassa ei edellytä toteutuksen muutosta aikaisemmissa 1.0-toteutuksissa. Kannattaa kuitenkin olla huolellinen käytettäessä toteutuksia sekaisin, että ongelmakohtia ei synny.

```
protected void onDraw(GL10 gl){
    /* if OpenGL ES 1.1 is supported. */
    if(gl instanceof GL11){
        GL11 gl11 = (GL11) gl;

        gl11.glBindBuffer(GL11.GL_ARRAY_BUFFER
               , GPUMemoryManager.instance().getTexturePointer(KEY));
        gl11.glTexCoordPointer(2, GL11.GL_FLOAT, 0, 0);

        gl11.glBindBuffer(GL11.GL_ARRAY_BUFFER
               , GPUMemoryManager.instance().getVertexPointer(KEY));
        gl11.glVertexPointer(3, GL11.GL_FLOAT, 0, 0);
        gl11.glRotatef(host.angle, 0, 0, 1);
        gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 0, safe().getSize(KEY)/3);
    } else {
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, safe().getTextureBuffer(KEY));
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, safe().getVertex(KEY));
        gl.glRotatef(host.angle, 0, 0, 1);
        gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, safe().getSize(KEY)/3);
    }
}
```

Koodiesimerkki 17. OpenGL ES 1.1 - tuettu piirtototeutus.

Jos OpenGL ES 1.1 ei ole tuettu tai tarvitaan optimointia lisää, voidaan optimoida myös tekstuurien käsittelyä. Tekstuurin latausta on hyvin vaikea optimoida ja siihen on tässä turha ryhtyä. Tekstuurien käytön parantaminen ja itse tekstuuritiedostojen parantaminen on helpompaa ja kannattavampaa.

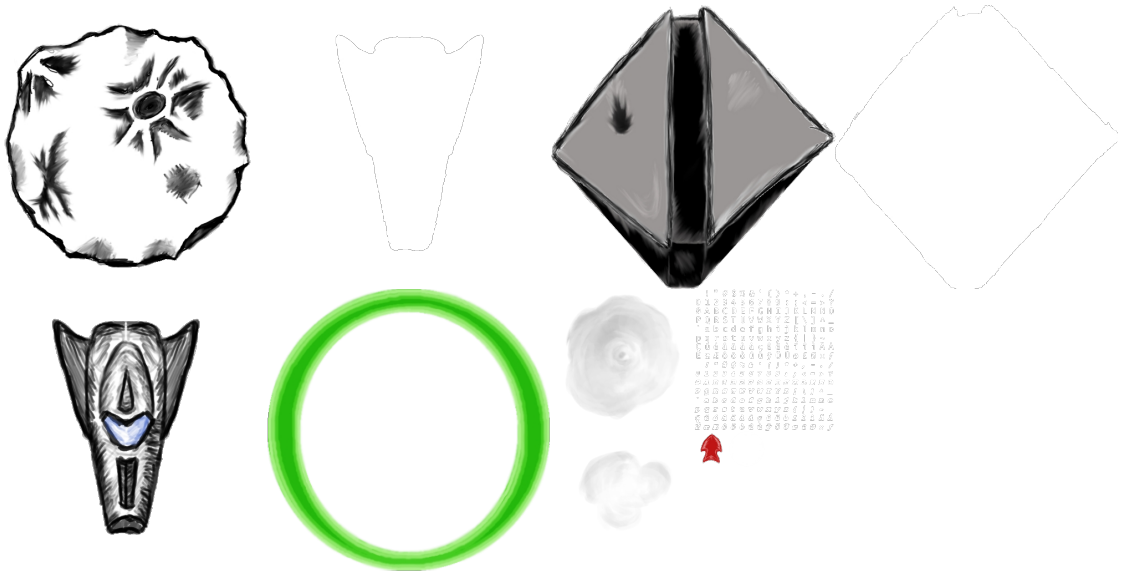
Helpoin tekstuurioptimointimenetelmä on välttää vaihtamasta videomuistissa viitattua tekstuuria. Tämä tarkoitattaa käytännössä sitä, että tarkastellaan uuteen piirtorutiiniin

siirryttäessä, mikä kyseinen osoitettu tekstuuri on. Jos se on sama kuin nykyinen, ei yritetä vaihtaa tekstuuriosoitinta. Tästä on esimerkki koodiesimerkissä 18.

```
/**
 * Binds the texture if security check deems it necessary.
 *
 * @param gl - OpenGL 1.0 context.
 * @param texture - texture ID returned by getTextureId.
 */
public static void bindTexture(GL10 gl, int texture) {
              if(latestBind != texture){
                            gl.glBindTexture(GL10.GL_TEXTURE_2D, texture);
                            latestBind = texture;
              }
}
```

Koodiesimerkki 18. Tekstuuriviittauksen vaihtotarkastelu.

Koska koodiesimerkki 18:n mukainen käsittely estää vain saman tekstuurin jatkuvan asettamisen, tarvitaan yleiseen tekstuurin vaihteluun toinen optimointimenetelmä. Paras tapa välttää tekstuurivaihdot on yksinkertaisesti olla tekemättä niitä ja yhdistää suurin osa tekstuureista yhdeksi isoksi tekstuuriksi, johon voidaan suoraan viitata piirron aikana ilman, että syntyy tarvetta vaihtaa sitä. Tällaisia tekstuuriryppäitä kutsutaan yleensä tekstuuriatlaksiksi, ja ne ovat erityisesti peleissä yleisessä käytössä helposti toteutettavana graafisena optimointimenetelmänä. Tekstuuriatlaksen voi joko rakentaa käsin kuvankäsittelyohjelmalla tai käyttää jotain jo verkossa pyörivaa ilmaisohjelmaa (kuva 5) [18].



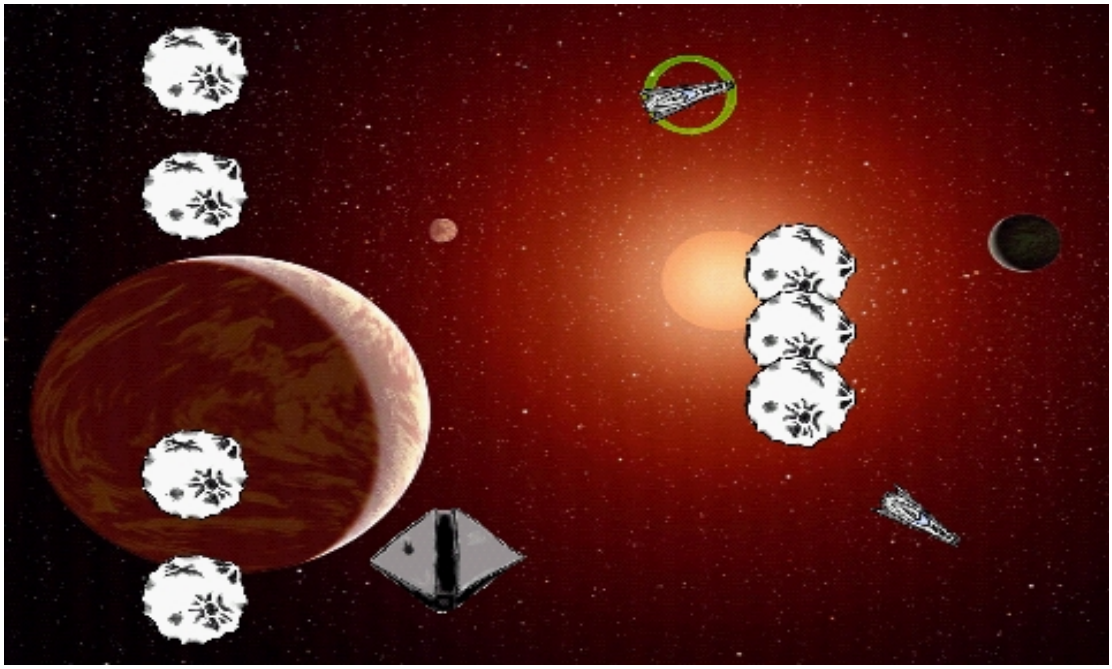
Kuva 5. Tekstuuriatlas

Yksi tekstuuriatlas voi käsitellä kymmeniä tai jopa satoja yksittäisiä tekstuureja ja laskea tekstuurivaihtojen määrää noin yhtä suurella määrällä. Atlaksesta voidaan sitten hakea oikea piirrettävä tekstuuri normaaleilla UV-koordinaateilla.

5 Space Strategy -pelin toteutus

Tämän luvun tarkoituksena on perehtyä tätä insinöörityötä varten toteutettuun esimerkkipeliprojektiin, joka on saanut työnimen Space Strategy [19.]. Tarkoituksena oli tehdä hyvin kännykän näytölle soveltuva reaaliaikaisesti toimiva strategiapeli, joka käyttäjän olisi helppo oppia, mutta samalla toteutettavissa muutamassa kuukaudessa. Toisena päätavoitteena oli koodatessa pitää yleiskäyttöiset kirjastoluokat erossa itse pelitoteutuksesta ja mahdollisesti käyttää niitä myös tulevissa Android-peliprojekteissa ilman suurempia muutoksia.

Pelissä on tarkoituksena pitää omat avaruusalukset hengissä samalla, kun väistellään eri puolelta ruutua lähestyviä asteroideja kuvan 6 mukaisesti. Käyttäjä häviää heti, kun kaikki omat avaruusalukset on tuhottu. Käyttäjä taas voittaa tason, jos kaikki asteroidit ovat lentäneet ruudun ulkopuolelle. Jos avaruusalus kuitenkin osuu asteroidiin, tuhoutuu se ja saa asteroidin lentämään pois radalta tehden siitä näin varsin arvaamattoman.



Kuva 6. Kuva pelitilanteesta.

Kaikki pelin 3 eri tasoa on rakennettu käyttäen xml-tiedostoja, jotka sitten käydään läpi koodissa ja luodaan sitä vastaava taso. Käytännössä tämä tarkoittaa, että lähes kuka tahansa voi muokata tasoista sellaisia kuin haluaa. Koodiesimerkki 19 esittää tason 1 xml-tiedostoa ja kertoo kaikki tason tarvitsemat tiedot käyttäjän omista aluksista, asteroideihin ja näytettäviin teksteihin.

```
<xml>
	<title>
		<name value="Level 1" />
		<difficulty value="Easy" />
	</title>
	<friendly>
		<ship>
				<type value="scout" />
				<attributes x="40%" y="45%" angle="0" />
		</ship>
		<ship>
				<type value="scout" />
				<attributes x="50%" y="45%" angle="0" />
		</ship>
		<ship>
				<type value="starbase" />
				<attributes x="60%" y="80%" angle="0" />
		</ship>
	</friendly>

<queue type="text" extra="Welcome to Space Strategy!" from_x="10%" from_y="8%" duration="5000" />
<queue type="text" extra="Click any of the ships to activate them." from_x="10%" from_y="8%" duration="5000"/>
<queue type="text" extra="When active you can draw waypoints for ships to move." from_x="10%" from_y="8%" du-
ration="5000" />
<queue type="text" extra="Protect your ships and avoid the asteroids." from_x="10%" from_y="8%" dura-
tion="5000" />
<queue type="text" extra="" from_x="10%" from_y="8%" duration="2000" />
<queue type="text" extra="Wave 1" from_x="10%" from_y="8%" duration="2000" />
```

```
<queue type="enemy" extra="asteroid" from_x="-10%" from_y="40%" to_x="110%" to_y="40%" joined="true" />
<queue type="enemy" extra="asteroid" from_x="-10%" from_y="50%" to_x="110%" to_y="50%" joined="true" />
<queue type="enemy" extra="asteroid" from_x="-10%" from_y="60%" to_x="110%" to_y="60%" joined="true" />
<queue type="text" extra="" from_x="10%" from_y="8%" duration="5000" />
<queue type="enemy" extra="asteroid" from_x="110%" from_y="10%" to_x="-10%" to_y="10%" joined="true" />
<queue type="enemy" extra="asteroid" from_x="110%" from_y="30%" to_x="-10%" to_y="30%" joined="true" />
<queue type="enemy" extra="asteroid" from_x="110%" from_y="50%" to_x="-10%" to_y="50%" joined="true" />
<queue type="text" extra="" from_x="10%" from_y="8%" duration="5000" />
<queue type="enemy" extra="asteroid" from_x="110%" from_y="50%" to_x="-10%" to_y="50%" joined="true" />
<queue type="enemy" extra="asteroid" from_x="110%" from_y="10%" to_x="-10%" to_y="10%" joined="true" />
<queue type="enemy" extra="asteroid" from_x="110%" from_y="90%" to_x="-10%" to_y="90%" joined="true" />
<queue type="text" extra="" from_x="10%" from_y="8%" duration="15000" />
<queue type="text" extra="Wave 2" from_x="10%" from_y="8%" duration="2000" />
<queue type="enemy" extra="asteroid" from_x="110%" from_y="40%" to_x="-10%" to_y="40%" joined="true" />
<queue type="enemy" extra="asteroid" from_x="110%" from_y="50%" to_x="-10%" to_y="50%" joined="true" />
<queue type="enemy" extra="asteroid" from_x="110%" from_y="60%" to_x="-10%" to_y="60%" joined="true" />
<queue type="text" extra="" from_x="10%" from_y="8%" duration="2000" />
<queue type="enemy" extra="asteroid" from_x="-10%" from_y="10%" to_x="110%" to_y="10%" joined="true" />
<queue type="enemy" extra="asteroid" from_x="-10%" from_y="30%" to_x="110%" to_y="25%" joined="true" />
<queue type="enemy" extra="asteroid" from_x="-10%" from_y="70%" to_x="110%" to_y="75%" joined="true" />
<queue type="enemy" extra="asteroid" from_x="-10%" from_y="90%" to_x="110%" to_y="90%" joined="true" />
<queue type="end" extra="Level clear | GameOver" from_x="10%" from_y="8%" duration="5000" />
</xml>
```

Koodiesimerkki 19. Tason 1 xml-tiedosto.

5.1 Rakenne

Pelin toteutus käyttää 2 eri Android-aktiviteettia toimiakseen. Aktiviteetit ovat Androidin omia sisäisiä aliohjelmia, joista Android-ohjelma koostuu. Näistä aktiviteeteistä yksi on varattu aloitusmenulle, josta pelaaja voi valita haluamansa tason. Toinen aktiviteetti vastaa itse pelistä ja ottaa käynnistyksen yhteydessä vastaan tiedoston, josta tason tiedot tulisi lukea.

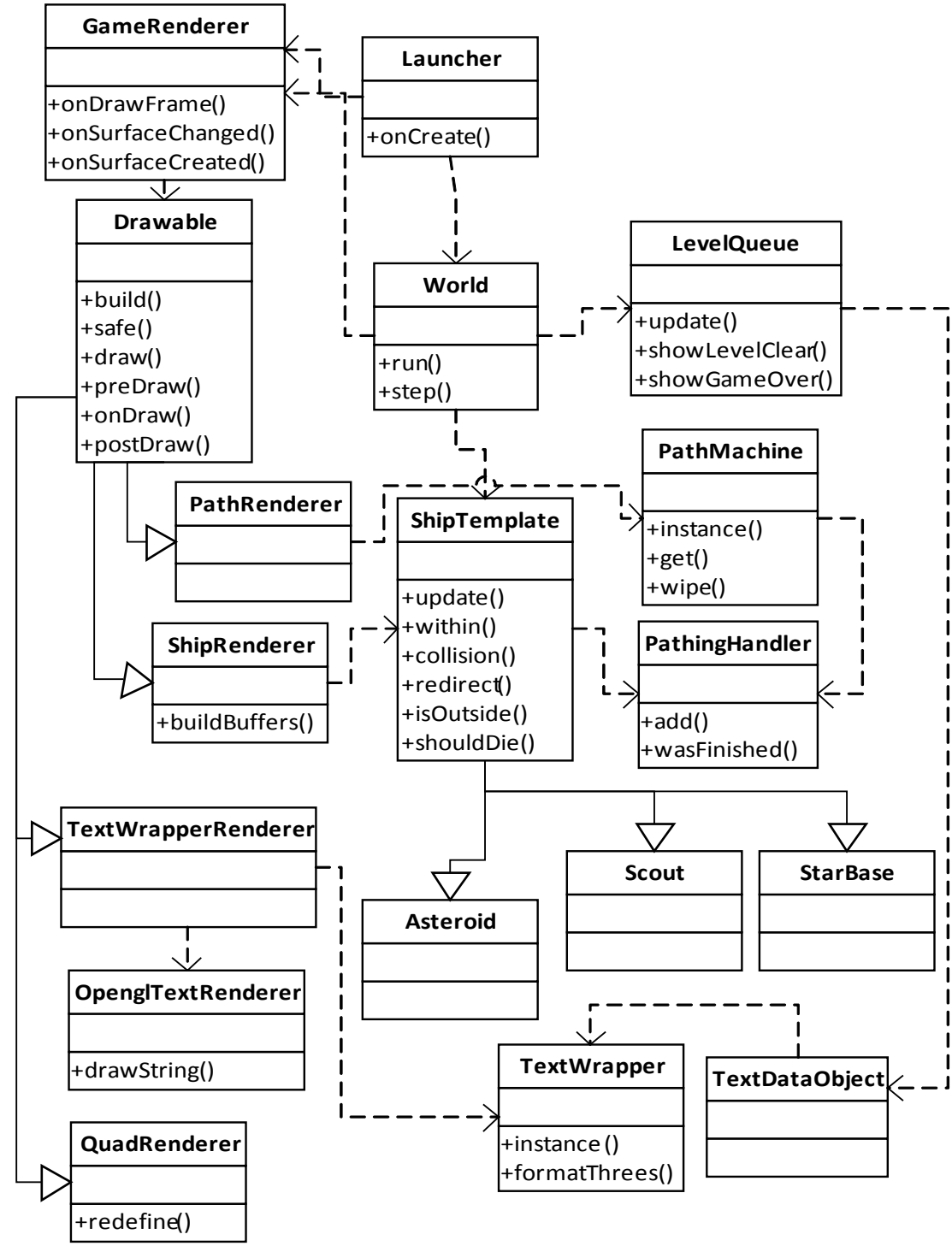
Käytössä oleva aloitusmenutoteutus on kaikessa yksinkertaisuudessaan MenuHandler-luokan alla. Pääasiassa aloitusmenu ja tason valinta koostuvat pelkästään Androidin natiivitason tarjoamasta UI xml -työkalun tuotoksista. Aloitusmenu käynnistyy varsin nopeasti riippumatta, millä kännykällä se ajetaan johtuen juuri siitä, että se ei vaadi paljon ladattavaa materiaalia, mikä on yleisesti hyvä asia. Ainoa tällainen ladattava materiaali on tasojen xml:istä parsitut lähdetiedot tason valintanapeille. Jos tai kun pelin aloitusmenu alkaa pursuta ja sen latausaika kasvaa, johtaa tämä usein myös hitaaseen reagointiin käyttöliittymäpuolella. Tämä on kuitenkin helposti korjattavissa asettamalla ns. tyhjä aktiviteetti ennen menuun siirtymistä. Näin nopeutetaan käynnistysikonin painamisreagointia.

Rakenne monimutkaistuu houmattavasti siirryttäessä itse pelin rakenteeseen. Pelin alkupiste on aina tässä toteutuksessa Launcher-luokassa, jonka tarkoituksena on tehdä alustusoperaatiot, kuten OpenGL-näkymän ja pelitason alustaminen. Viimeisenä operaationa Launcher-luokka käynnistää peliloopin luomalla World-luokan säikeen. Kaiken

tämän jälkeen Launcher-luokan tehtäväksi jää vain kosketus- ja näppäintapahtumien välittäminen niitä hallinnoiville luokille.

Launcher-luokan käynnistämä World-luokassa toimiva pelilooppi toimii täsmälleen kuin aikaisemmin peliohjelmointiluvussa ehdotettiin. Käytännössä tämä tarkoittaa sitä, että siinä tarkastellaan pelin kosketustapahtumat ja miten ne vaikuttavat eri ShipTemplate-luokan olioihin, joista koostuvat kaikki pelissä liikkuvat pelaajan omat alukset kuin vihollisasteroiditkin. Tarkasteluiden jälkeen ShipTemplate-olioiden tila päivitetään, jonka jälkeen päivitetty ruutu piirretään kännykän näytölle, kun pyydetään GLSurfaceView-luokan ilmentymää päivittämään ruutu. Kyseinen ruudun päivitys toimii Android-kirjastojen alla omassa säikeessään, ja lopulta se päätyy GameRenderer-luokan onDrawFrame-funktioon, jossa kaikki pelin sisältämä OpenGL-toteutus piirretään ruudulle.

Kuva 7 sisältää luokkakaavion suurimmasta osasta luokista, jotka ovat osa projektin pelitoteutusta. Se ei kuitenkaan sisällä ns. erikoisluokkia, kuten partikkelimoottoria ja muita singleton-luokkia, joilla ei ole suoranaista vaikutusta moottoriin. Niiden merkitys on visuaalinen, jotta kaaviota olisi helpompi lukea.



Kuva 7. Pelin luokkarakenne.

5.2 Liikepistejärjestelmä

Yksi tärkeimmistä pelin ominaisuuksista on sen käyttämä liikepistejärjestelmä. Se listaa ja käsittelee liikepisteet ShipTemplate-olioille, jotta ne voivat liikkua täsmälleen näiden pisteiden välillä. Järjestelmä liikuttaa kaikkia pelissä olevia avaruusaluksia ja asteroideja

niille määrättyjen liikepisteiden perusteella. Nämä liikepisteet voivat olla ennaltamäärättyjä, kuten asteroideilla, tai käyttäjän asettamia, kuten avaruusaluksilla. Päätarkoituksena järjestelmää suuniteltaessa oli antaa käyttäjälle täysi kyky ohjastaa avaruusaluksia juuri niin kuin hän haluaisi, mikä johti liikepisteratkaisuun, jossa liikeradat voivat olla niin monimutkaisia tai yksinkertaisia kuin haluaa.

ShipTemplate-olio kuvaa mitä tahansa pelissä liikkuvaa esinettä avaruusaluksista asteroideihin. Se sisältää samalla tiedon omasta waypoint-listastaan wrapperi-luokka PathingHandlerin välityksellä sekä määrittää, kuinka kyseinen olio liikkuu kunkin pisteen välillä. Tämä ShipTemplate-luokan update-toteutus on nähtävillä koodiesimerkistä 22, joka sijaitsee luvun 5 kappaleessa 4.1.

PathingHandler-luokka sisältää tiedon listasta pisteitä, jotka on kuvattu käyttäen vektoriluokka Vec2f:ää. PathingHandlerin päätarkoituksena on varastoida pisteet ja kitkeä turhat pisteet pois, kun niitä ollaan lisäämässä siihen. Turhiin pisteisiin kuuluu esim. liian lähellä oleva piste edelliseen lisättyyn pisteeseen, joka ainoastaan kasvattaisi listan kokoa ja muistinkäyttöä turhaan.

Koodiesimerkki 20 esittää, kuinka rutiinin tarkistus hylkää kaikki pisteet, jotka ovat MIN_RADIUS-muuttujan määrittämällä etäisyydellä edellisestä pisteestä. Se myös poistaa vanhoja pisteitä listasta, jos listan koko käy liian suureksi ja uusia pisteitä tulisi silti lisätä.

```
public void add(int x, int y){
		if(path.isEmpty()){
				path.add(m_factory.assign(x,y));
		} else {
				Vec2f last = path.get(path.size()-1);
				if((x > last.x + MIN_RADIUS || x < last.x - MIN_RADIUS)
				&& (y > last.y + MIN_RADIUS || y < last.y - MIN_RADIUS))
				{
						if(path.size() >= MAX_SIZE){
								m_factory.free(path.get(0));
								path.remove(0);
						}
						path.add(m_factory.assign(x,y));
				}
		}
}
```

Koodiesimerkki 20. PathingHandlerin pisteen lisäys ja tarkistusrutiini.

Alimpana tasona on PathMachine-luokka. Tämän singleton-luokan ainoa tarkoitus on varastoida kaikki sillä hetkellä käytössä olevat PathingHandler-oliot. Aina kun uusi Pat-

hingHandler-olio luodaan, saa PathMachine-luokka sen automaattisesti. Samoin aina kun roskienkerääjä tuhoaa PathingHandler-olion, se poistetaan PathMachine-luokasta. Ainoa muu tarkoitus PathMachine-luokalla varastoinnin lisäksi on välittää kaikki liikepisteet piirrettäväksi, jos näin halutaan tehdä.

5.3 Drawable-piirtorajapinta

Drawable-luokka on oma piirtorajapintatoteutukseni ja eroaa täysin Androidin samannimisestä luokasta. Drawable-luokan tarkoituksena on tarjota helppo rajapinta uusien piirto-olioiden luontiin. Tähän sisältyy myös VBO:n luonti ja varastointiolioiden ylläpitäminen. Piirtorajapintaa kuvaa hyvin koodiesimerkki 21, missä vain Drawable-luokan draw-funktion toteutus on oikeasti osa Drawable-luokkaa eikä ole ylikirjoitettavissa. Loput kutsuttavat preDraw, onDraw ja postDraw ovat Drawable-luokassa vain raakatoteutuksina, eli niillä on yksinkertainen toteutus, joka kuitenkin aliluokassa kannattaa ylikirjoittaa.

```
public static final BufferFactory build() {
	if(m_factory == null){
		m_factory = new BufferFactory();
	}
	return m_factory;
}

public static final BufferSafe safe() {
	if(m_safe == null){
		m_safe = new BufferSafe();
	}
	return m_safe;
}

/**
 * When using preset draw this function is called right before.
 *
 * override to add your own functionality.
 *
 * @param gl - OpenGL 1.0 context.
 */
protected void preDraw(GL10 gl){

}

/**
 * Main draw function.
 * Called after preDraw and just before postDraw.
 *
 * Following preset functionality:
 *     - Draws the last vertex and texture buffers returned using Triangle strip.
 *
 * override to add your own functionality.
 *
 * @param gl - OpenGL 1.0 context.
 */
protected void onDraw(GL10 gl){
```

```
		if(m_safe != null && m_safe.getVertex(m_safe.latest_texture) != null){
			if(m_safe.getTextureBuffer(m_safe.latest_texture) != null){
					gl.glTexCoordPointer(2
					, GL10.GL_FLOAT
					, 0
					, m_safe.getTextureBuffer(m_safe.latest_vertex));
			}
		gl.glVertexPointer(3, GL10.GL_FLOAT, 0, m_safe.getVertex(m_safe.latest_vertex));
		gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP
			, 0
			, m_safe.getSize(m_safe.latest_vertex)/3);
		}
}

/**
 * When using preset draw this function is called right after.
 *
 * override to add your own functionality.
 *
 * @param gl - OpenGL 1.0 context.
 */
protected void postDraw(GL10 gl){

}

/**
 * Preset draw function called by the renderer object.
 *
 * @param gl - OpenGL 1.0 context.
 */
public final void draw(GL10 gl){
		if(!active) return;
		preDraw(gl);
		onDraw(gl);
		postDraw(gl);
}
```

Koodiesimerkki 21. Tärkeimmät Drawable-luokan toteutukset.

Drawable-luokan valmis onDraw-toteutus piirtää yksinkertaisen esineen ruudulle. Esineeksi oletetaan edellinen määritetty esine. Luokan build- ja safe-funktiot antavat singleton-tyyppisen viittauksen BufferFactory- ja BufferSafe-luokkiin, jotka käsitellään tarkemmin luvun 4 kappaleessa 3.

5.4 Oman projektin aloittaminen

Tässä luvussa käsitellään, mitä vaaditaan oman Android-peliprojektin aloittamiseen ja pystyttämiseen. Luvussa käydään lyhyesti asiat, jotka tulisi osata, sekä asiat, jotka olisi hyvä osata, mutta ei välttämätöntä.

5.4.1 Tarvittava osaaminen

Lähtökohtaisesti, ennen kuin aloittaa edes Android-pelikehityksen, tulisi osata Java-koodia ainakin siinä määrin, että oliopohjainen ohjelmointi sujuu Javalla. Myös grafiikka on tärkeä osa pelejä. Siksi jokin Androidin tukema graafinen rajapinta kannattaisi olla

tuttu. Nämä kaksi asiaa ovat ainoat asiat, jotka on pakko tietää. Muut oppii huoletta tarpeen vaatiessa.

Kahden edellä mainitun asian lisäksi olisi hyvä tietää hieman vektorimatematiikasta ja pelinkehityksestä. Pelinkehityksestä voi lukea hieman Peliohjelmointi-luvusta. Vektorimatematiikan käytöstä on koodiesimerkki 22.

Koodiesimerkissä 22 on ShipTemplate -olion päivitysrutiini, jossa olio seuraa sille määritettyjä reittipisteitä. Olion kulma ja suunta lasketaan käyttämällä vektorimatematiikkaa ja muutetaan olion xy-sijaintia näiden suhteen. Suurin osa matematiikasta on piilotettu Vec2f-vektoriluokan alle, mikä helpottaa piilottamalla suuren osan laskennasta ja tekemällä koodista siistimpää.

```
private Vec2f pathingAssist = new Vec2f();
private Vec2f zeroAngle = new Vec2f(0,1);


/**
 * updates this shiptemplate object.
 *
 * @param delta
 */
public void update(float delta){
		if(!path.getPath().isEmpty()){
				hasStopped = false;
				pathingAssist.x = path.getPath().get(0).x-x;
				pathingAssist.y = path.getPath().get(0).y-y;
				calculateAngle();
				float n = 1/pathingAssist.length();
				float nx = pathingAssist.x*n;
				float ny = pathingAssist.y*n;
				x += velocity*nx*delta;
				y += velocity*ny*delta;

				if(pathingAssist.length() < deadZone){
						path.getPath().remove(0);
						if(dieOnEmptyPath && path.getPath().isEmpty()){
								dead = true;
						}
				}
		} else {
				hasStopped = true;
		}
}

/**
 * Calculates the current angle for the set pathingAssist
 *
 * @param p
 */
private void calculateAngle(){
		if(ignoreAngle) return;
		angle = zeroAngle.angle(pathingAssist);
		if(pathingAssist.x >= 0) angle = (float) (Math.PI*2-angle);
		angle *= 180/Math.PI;
}
```

Koodiesimerkki 22. Reittipistejärjestelmän toteutus.

5.4.2 Projektissa huomioitavat asiat

Kuten mitä tahansa projektia aloittaessa, kannattaa oman pelin tavoite rajata mahdollisimman tarkasti ja pohtia ennen koodaamisen aloittamista, kuinka kukin suunniteltu ominaisuus kannattaisi toteuttaa. Hyvä nyrkkisääntö varsinkin aloittevalle peliohjelmoijalle on se, että jos halutun ominaisuuden suunnittelussa ei selviä, kuinka ominaisuus tulisi toteuttaa, kannattaa se jättää pois kokonaan. Sama pätee ominaisuuksiin, jotka eivät toimi odotetusti tai eivät ole yhtä hyviä kuin alunperin oletettiin. Tällaiset ominaisuudet on hyvä jättää pois samantien, kun ne ilmenevät. Tästä syystä pelin koodikanta muuttuu suuressa osin moneen otteeseen pelinkehityksen aikana.

Yleiskehityksellisten asioiden lisäksi on hyvä pitää mielessä jatkuvasti, mille kohdelaitteille peliä halutaan kehittää. Jos tarkoituksena on tehdä Android-markettiin tai vastaavaan palveluun myyntiin menevä tuote, on hyvä unohtaa emulaattorikehitys hyvin nopeasti ja siirtyä oikeisiin testilaitteisiin. Samoin jos peli on graafisesti näyttävä ja/tai laskennallisesti raskas, se saattaa johtaa siihen, että uudetkin puhelimet saattavat olla liian heikkoja pyörittämään peliä, vaikka käyttöjärjestelmäversio olisikin oikea. Kannattaa myös huomioida, että suuresta mallimäärästään johtuen Android-puhelimien komponentteja on valtava määrä eikä kannata olettaa niiden olevan yhtä hyvä- tai huonolaatuisia kuin testipuhelimen.

Kehitysympäristöllä ei suuremmilla osin ole merkitystä Android-kehityksessä. Windows, Linux ja OSX toimivat kukin ihan mainiosti kehitysalustoina ja ovat hyvin tuettuja. Ainoa asia, mikä tulee vaikuttamaan kehitysympäristön valintaan oman mieltymyksen lisäksi, on se, halutaanko ohjelmassa käyttää natiivikoodia. Jos halutaan käyttää natiivikoodia, helpoimmalla pääsee, kun valitsee minkä tahansa Linux-distribuution. Syy tähän on se, että jotta Androidilla toimivaa natiivikoodia voidaan kääntää, tarvitaan Androidia vastaava käyttäjärjestelmä. Koodin kääntäminen onnistuu myös muilla käyttöjärjestelmillä, mutta se vaatii paljon enemmän asettelua ja mahdollisesti joitain kolmannen osapuolen työkaluja.

Kehitykseen voi käyttää mitä tahansa ANT-yhteensopivaa työkalua tai helpoimpana vaihtoehtona Eclipseä [20], jolle löytyy suoraan Googlen tarjoama liitäntä Android-kehitykseen.

5.4.3 Android-projektin pystyttäminen Eclipse-kehitysympäristöön

Android-projektin pystyttäminen Windows- tai Linux-ympäristöön ei ole kovinkaan vaikeaa, ja prosessi on lyhyesti dokumentoitu Android-kehittäjäsivustolle [21]. Lyhyesti Android-kehitys vaatii käyttöjärjestelmälle sopivan Android-sdk-paketin [22], eclipseen ADT-lisäpalikan [23] ja näiden kahden osan yhdistämisen.

Android-sdk-paketti ei vaadi erillistä asentamista yhdelläkään käyttöjärjestelmällä ja täten se on helppo ottaa käyttöön. Windowsilla on kuitenkin suositeltavaa käyttää Googlen tarjoamaa valmista .exe-binääripakettia, sillä ilman sitä Windows rekistereitä tarvitsevat manuaalista muokkausta. Ennen kuin Android-sdk-paketti otetaan käyttöön, se kannattaa vielä päivittää. Päivitys tapahtuu vaivattomasti ainaki sdk:n sisällä olevalla sdk-hallintatyökalulla.

Eclipse-asennus on hieman monimutkaisempi, ja osa siitä tarvitsee uusia joka kerta, kun uusi Eclipse-työtila luodaan. Ensimmäisenä kannattaa asentaa lähes kaikki Eclipseen tarjolla olevat liitännät osoitteesta ssl.google.com/android/eclipse/. Ihan jokaista liitäntää ei ole tarvetta asentaa, jotta kehitys voi alkaa mutta se on suositeltavaa. Erityisesti DDMS on tärkeä liitäntä, joka tarjoaa kehityksen aikaista diagnostiikkaa. Toimiakseen kaikki Eclipse-liitännät tarvitsee kuitenkin yhdistää Android-sdk-pakettiin. Sdk-paketin yhdistäminen on Eclipsessä työtila-kohtaista.

5.4.4 Peliprojektin aloittaminen

Kuten jo peliohjelmointiluvussa mainittiin, kannattaa peliprojekti aloittaa tutustumalla siihen, mistä peliprojekti yleensä koostuu ja miten nämä voitaisiin soveltaa juuri omaan peliin. Toteutus-luvusta voi tarkastella yhtä tällaista peliprojektia.

Kun alkututustumiset on tehty, kannattaa suunnitella peli ainakin seuraavat asiat mielessä pitien:

- Minkälaista peliä ollaan tekemässä ja mitä se pitää sisällään?
- Miten suunnitellut toiminnot ajateltiin toteuttaa?
- Yleisen rakenteen alustava suunnitelma.
- Alustava työnjako.

Ensimmäinen kohta on projektiin eniten vaikuttava, sillä on hyvin erilaista toteuttaa Tetris verrattuna tasoloikkaan. Toisen kohdan eli ominaisuuksien toteutustavan ei tarvitse olla kaikilta kohdin täysin tiedossa vielä tässä vaiheessa, mutta jos suunnittelun yhteydessä ei selviä, miten asiat tulisi ainakin yleispiirteiltään toteuttaa, on ominaisuus pelkkä toive. Kolmas kohta eli rakenne on erityisen tärkeää, jos työtä ei tehdä yksin. Hyvin suunniteltu ja sitä myöten toteutettu rakenne helpottaa työtä tulevaisuudessa. Neljäs ja viimeinen kohta eli työnjako on hyvä hoitaa ajoissa. Varsinkin monimutkaisten ominaisuuksien kanssa on parempi, että sitä toteuttavat alusta loppuun samat henkilöt, jotta koodi pysyy selkeänä ja myös tulevaisuudessa helposti muunneltavana.

Pelit ovat yleensä hyvin monimutkaisia ja varsinkin loppua kohden helposti erittäin sekavia, kun mukaan aletaan lisätä niin sanottuja kosmeettisia muunnoksia jo silloin, kun pelimoottori on valmis tai hyvin valmis, mutta siitä puuttuu vielä valmiin pelin hohto. Tämän takia kannattaa pitää koodi siistinä mahdollisimman pitkään ja tarpeen vaatiessa kirjoittaa se moneen otteeseen uusiksi yhtenäisyyden ja siisteyden säilyttämiseksi. Jos kaikki muu epäonnistuu, aina voi kokeilla, mitä tapahtuu.

6 Yhteenveto

Vasta 2008 julkaistu Android on käyttöjärjestelmänä ja alustana kasvanut merkittävästi viimeisen muutaman vuoden aikana. Ongelmat, kuten paikoitellen huonosti toimiva versioyhteensopivuus, eivät ole olleet esteenä. Kirjastovirheet ovat harvinaisia, mutta sitäkin turhauttavimpia. Onneksi suurin osa kirjastovirheistä on helposti itse korjattavissa tai kierrettävissä. Androidin valmiskirjastot ovat laajat ja tarjoavat hyvät työkalut pelikehitykseen. Google tarjoaa myös useita valmistyökaluja Android-kehitykseen, jotka ovat osittain jopa paremmat kuin PC-puolen Javassa.

Suurin ongelma tällä hetkellä Android-kehitykselle on eri kohdelaitteiden valtava määrä. Kohdelaitteiden vaihtelevat Android-versiot ja kokoonpanot aiheuttavat helposti ohjelmavirheitä tai jopa laitekohtaisia virheitä. Onneksi tarjolla olevat kirjastot mahdollistavat laitekohtaisen tarkastelun ja koodin ajon.

Pelit ovat usein niin monimutkaisia ohjelmia, että yleinen suorituskyky on helposti ongelma varsinkin Androidin tapaisissa sulautetuissa järjestelmissä. Androidin Java-arkkitehtuuri ei tässä mielessä paranna asiaa yhtään ja tästä syystä Android kärsii helposti performanssiongelmista. Erityisesti halvemmat laitteet vaativat useasti raskasta optimointia toimiakseen sujuvasti. Riippuen pelistä optimointi kannattaa keskittää prosessoriin, muistiin tai näytönohjainoptimointeihin, sillä optimoinnit ovat useasti resurssivaihtokauppoja. Varsinkin näytönohjain jää helposti jälkeen näyttävimmissä peleissä. Se tarkoittaa sitä, että piirto ei tapahdu tarpeeksi nopeasti tai muistia ei ole tarpeeksi pitämään tekstuureja muistissa pitkiä aikoja tai jopa ollenkaan.

Ongelmista huolimatta Android on hyvä käyttöjärjestelmävalinta pelikehitystä varten. Käyttöjärjestelmää parannellaan jatkuvasti uusilla päivityksillä ja korjauksilla, jotka korjaavat edellä mainittuja ongelmia ja parantavat jo hyvin laajoja standardikirjastoja. Jos mobiilikehitys kiinnostaa, on Androidia vaikea olla suosittelematta pelikehitykseen.

Lähteet

1: James Steele, Nelson To, The Android Developer's Cookbook, 2011.
2: , Android (operating system), Verkkodokumentti <http://en.wikipedia.org/wiki/Android_%28operating_system%29> 2011, Luettu 28.9.2011.
3: Tim Bray, What Android Is, Verkkodokumentti <http://www.tbray.org/ongoing/When/201x/2010/11/14/What-Android-Is> 2010, Luettu 28.9.2011.
4: Bornstein Dan, Dalvik VM Internals, Verkkodokumentti <http://sites.google.com/site/io/dalvik-vm-internals> 2008, Luettu 3.12.2011.
5: Google, Android API Levels, Verkkodokumentti <http://developer.android.com/guide/appendix/api-levels.html> 2011, Luettu 28.9.2011.
6: Google, 3D with OpenGL, Verkkodokumentti <http://developer.android.com/guide/topics/graphics/opengl.html> , Luettu 4.10.2011.
7: André LaMothe, Tricks of the 3d Game Programming Gurus, 2003.
8: , Texture Mapping, Verkkodokumentti <http://www.happy-werner.de/howtos/isw/parts/3d/chapter_2/chapter_2_texture_mapping.pdf> , Luettu 12.3.2011.
9: Google, 2D Graphics, Verkkodokumentti <http://developer.android.com/guide/topics/graphics/2d-graphics.html> , Luettu 3.10.2011.
10: Khronos group, OpenGL Overview, Verkkodokumentti <http://www.opengl.org/about/overview/> , Luettu 14.1.2012.
11: Google, Platform Versions, Verkkodokumentti <http://developer.android.com/resources/dashboard/platform-versions.html> , Luettu 3.10.2011.
12: Google, Media, Verkkodokumentti <http://developer.android.com/guide/topics/media/index.html> , Luettu 4.10.2011.
13: Google, SoundPool, Verkkodokumentti <http://developer.android.com/reference/android/media/SoundPool.html> , Luettu 4.10.2011.
14: Wikipedia, Just-in-time compilation, Verkkodokumentti <http://en.wikipedia.org/wiki/Just-in-time_compilation> , Luettu 29.10.2011.
15: Google, zipalign, Verkkodokumentti <http://developer.android.com/guide/developing/tools/zipalign.html> 2011, Luettu 1.11.2011.
16: , Obfuscated code, Verkkodokumentti <http://en.wikipedia.org/wiki/Obfuscated_code> 2011, Luettu 1.11.2011.
17: Object Oriented Design, Object Pool, Verkkodokumentti <http://www.oodesign.com/object-pool-pattern.html> , Luettu 1.11.2011.
18: Ivan-Assen Ivanov, Practical Texture Atlases, Verkkodokumentti <http://www.gamasutra.com/view/feature/2530/practical_texture_atlases.php> 2006, Luettu 15.11.2011.
19: Sami Koivisto, Space Strategy Repository, Verkkodokumentti <https://github.com/xill/and_sstrat> 2012, Luettu 14.1.2012.
20: Eclipse Foundation, Eclipse IDE, Verkkodokumentti <http://www.eclipse.org/> , Luettu 22.10.2011.
21: Google, ADT Plugin for Eclipse, Verkkodokumentti <http://developer.android.com/sdk/eclipse-adt.html> , Luettu 22.10.2011.
22: Google, Download the Android SDK, Verkkodokumentti <http://developer.android.com/sdk/index.html> , Luettu 22.10.2011.
23: Google, Android Developer Tools, Verkkodokumentti <http://developer.android.com/guide/developing/tools/adt.html> , Luettu 22.10.2011.