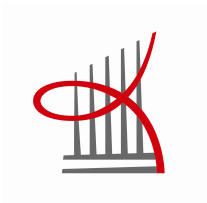


# GRAFIIKAN OHJELMOINTIA ANDROID-PELIPROJEKTISSA (CANVAS JA OPENGL ES 1.0)

Joni Rissanen

Opinnäytetyö
Helmikuu 2014
Viestintä
Vuorovaikutteinen media

TAMPEREEN AMMATTIKORKEAKOULU

Tampere University of Applied Sciences

**TIIVISTELMÄ**

Tampereen ammattikorkeakoulu
Viestintä
Vuorovaikutteinen media

Joni Rissanen:

Grafiikan ohjelmointia Android-peliprojektissa (Canvas ja OpenGL ES 1.0)

Opinnäytetyö 39 sivua

Helmikuu 2014

---

Tämä opinnäytetyö kertoo tekemästäni peliprojektista Android-käyttöjärjestelmälle, keskittyen graafisen puolen ohjelmointiin. Projekti on tehty kahdella eri tavalla; ensin tehden grafiikat Canvas-luokkaa käyttäen, sitten käyttäen OpenGL ES 1.0 -ohjelmointirajapintaa. Tässä opinnäytetyössä vertaillaan näitä kahta menetelmää aloittelevan Android-ohjelmoijan näkökulmasta.

Opinnäytetyön tarkoituksena on antaa neuvoja muille aloitteleville Android-ohjelmoijille ja valmistaa heitä mahdollisesti eteen tuleviin ongelmiin, joita itselleni on tullut vastaan projektia tehdessäni.

---

Avainsanat: ohjelmointi, java, android, canvas, opengl

**ABSTRACT**

Tampere University of Applied Sciences
Degree Programme in Media
Interactive media

Joni Rissanen:

Programming of graphics for an Android game project (Canvas and OpenGL ES 1.0)

Bachelor's thesis 39 pages

February 2014

---

This thesis is about a game project I have made for Android operating system, concentrating on programming of graphics. The project has been made in two different methods; first by executing graphics using Canvas class, and then by using OpenGL ES 1.0 application programming interface. In this thesis these methods are compared from the perspective of a novice Android-programmer.

The meaning of this thesis is to give advice to other beginner Android-programmers and make them prepared for some possibly upcoming problems, which I have run into while making this project.

---

Keywords: programming, java, android, canvas, opengl

## Sisällysluettelo

# 1 Johdanto

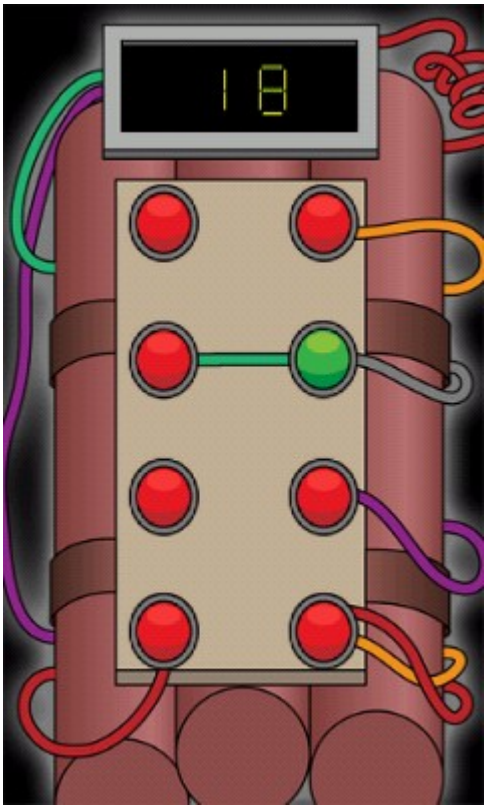
## 1.1 Projektin esittely

Kuten kaikki tietävät, herääminen aamulla voi olla varsin haastavaa joinakin päivinä. Aivot eivät toimi täydellä teholla ja jo sängystä nouseminen voi tuntua niin henkisesti kuin fyysisesti raskaalta työltä. Samoin herätyskello on oma ongelmansa, sillä vaikka se soi, ihminen voi puoliunisena – tai jopa unissaankin – painaa herätyskellon kiinni, heräämättä siihen oikeasti ollenkaan.

Keksimme toverini Riku Holopaisen kanssa näihin molempiin ongelmiin ratkaisun. Teemme yhdessä herätyskello-applikaatiota, joka varmistaa, että käyttäjä on varmasti hereillä, eikä vain paina herätyskelloa kiinni unissaan. Koska tämä tapahtuu muistipelin ansiosta, päivä alkaa heti mielekkäällä ja kevyellä aivotyöskentelyllä. Näin käyttäjä on virkeämpi heti herättyään, ja päivä saa muutenkin mukavan alun. Koska heräämisessä on kuitenkin usein kiire jonkinlaisesta kiireestä, peli ei vie juuri minuuttia kauempaa – suurin piirtein sen verran, että aivot ehtivät virkistyä, eikä käyttäjä nukahda heti uudestaan.

Edellä mainittu Holopainen, joka ei suoranaisesti liity tähän opinnäytetyöhön, on ottanut vastuulleen itse herätyskellon toiminnan ohjelmoimisen, kun taas minä puolestani hoidan pelin teon. Tämä sisältää game design -puolen, ohjelmoinnin ja grafiikoiden teon.

Tässä opinnäytetyössä keskitymme aloittelevan ohjelmoijan grafiikan ohjelmointiin ja ongelmien ratkaisemiseen. Teen saman projektin kahdella menetelmällä ja vertaan näitä keskenään. Ensimmäiseksi kokeilen yksinkertaisempaa ja aloittelijoille suositeltua tapaa, eli käyttäen Android API:n SurfaceView- ja Canvas-luokkia grafiikan esittämiseen. Toisella kerralla korvaan nämä OpenGL ES 1.0 -rajapinnalla ja sen omilla luokilla. Jälkeenpäin vertaan näitä kahta menetelmää esittäen omia ajatuksiani, ongelmia, ratkaisuja ja menetelmien vaikutusta pelin toimintaan ja suorituskykyyn.




KUVA 1. Kuvakaappaus pelistä. Luku 18 esittää jäljellä olevaa aikaa sekunteina, ja napit vaihtavat väriä järjestyksessä. Pelaajan tulee muistaa oikea järjestys ja painaa nappeja vuorollaan.

## 1.2 Miksi Android?

Android on yksi käytetyimmistä älypuhelimille tehdyistä käyttöjärjestelmistä. Muita vaihtoehtoja ovat esimerkiksi iOS, Symbian ja Windows Phone 7.

Henkilökohtaisesti Android oli projektin alussa ehdoton suosikkini jo siksi, että se on ainoa edellämainituista käyttöjärjestelmistä, jolle ohjelmoinnista minulla on alkuunkaan yhtään kokemusta: koulutukseen kuuluvassa työharjoittelussani tein Android-projekteja kuuden kuukauden ajan ja opin vähintäänkin käyttöjärjestelmän perusasiat, joskin itse pelien ohjelmoiminen jäi kovin vähäiseksi.

Käyttäjäkunniltaan Symbian ja Windows Phone eivät ole juurikaan tätä kirjoittaessani kovin suosittuja iOS:ään ja Androidiin verrattuna, mikä on varsin oleellinen seikka, mikäli haluaa ohjelmalleen mahdollisimman paljon ostajia.

Vaihtoehtoja pohdiskellessani Android meni iOS:n edelle lopullisesti sen vuoksi, että iOS:lle ohjelmointi vaatii Mac-tietokoneen. Nollabudjetti ei riittänyt moiseen, joten se jäi. Sen sijaan Androidille applikaation tekeminen on ilmaista – Windowsia käyttävä tietokone minulla oli jo, ja ohjelmointiympäristö Eclipse ja siihen ladattava Android SDK ovat ilmaisia – eikä edes ohjelman kauppaan myytäväksi laittaminen maksa paljoa. Ohjelman saaminen Google Play -ohjelmistokauppaan vaatii vain alle 20 euron (25 Yhdysvaltain dollarin) kertamaksun, joka maksetaan vain kerran per ohjelmantekijä/yritys. Täten tällä samalla maksulla voi julkaista useampia ohjelmia maksamatta niiden myyntiin saamisesta erikseen. Applen App Storessa maksu menee yksityisellä yrittäjällä siten eri tavalla, että yrittäjä maksaa reilun 70 euron (99 Yhdysvaltain dollarin) vuosittaisen maksun, mikä on selkeästi kalliimpi vaihtoehto. (iOS Developer Program, 2014) Molemmat kaupat ottavat itselleen 30% provision myytyjen ohjelmien tuloista. (Android Developers, 2014a, iOS Developer Program, 2014) Ohjelman myyminen Google Playssa maksaa siis käytännössä vain, jos ohjelma tuottaa voittoa, koska se ei vaadi kiinteää vuosimaksua, toisin kuin Applen App Store.

Androidin puolesta puhuu myös se, että älypuhelimien uusin tulokas, Jolla, tukee Android-ohjelmistoja, joskaan Jolla ei voi käyttää Google Playta sovelluskauppana. Sen sijaan Jollan pääyhteistyökumppani sovelluskauppana on venäläinen Yandex.Store, mutta muutkin Android-sovelluskaupat (Google Playta lukuun ottamatta) toimivat Jollassa, kuten esimerkiksi Amazon Appstore. Android-laitteilla toimivat luonnollisesti kaikki Android-sovelluskaupat. (Yle, 2014)

Mutta kuten kaikki muutkin käyttöjärjestelmät, myös Android tuo mukanaan omat ongelmansa. Pääasiassa useimpien ongelmien alku ja juuri on se, että erilaisia Android-laitteita on huomattavan paljon. Tämä aiheuttaa lisätyötä niin ohjelmoijille kuin graafikoille, kun ohjelma pitää saada toimimaan mahdollisimman monissa laitteissa, jotka saattavat toimia kukin eri tavalla. Tämä johtuu mm. käyttöjärjestelmän eri versioista, laitteistosta, ajureista ja näytön koosta. Lähtökohtaisesti olen rajannut tätä ongelmaa projektissa siten, että tekemäni ohjelma vaatii vähintään Androidin version 2.2, toimien vain kyseisessä ja uudemmissa versioissa. Tämä kuitenkin käsittää noin 99,9% tällä hetkellä käytössä olevista Android-laitteista, mikä onkin varmasti tarpeeksi. (Android Developers, 2014b)

## 1.3 Miksi Canvas ja OpenGL ES 1.0?

Koska tämä opinnäytetyö on pääasiassa suunnattu aloitteleville Android-ohjelmoijille, olen käyttänyt projektissani mahdollisimman helposti opittavia keinoja grafiikan piirtämiseen. En kuitenkaan käytä valmiita grafiikka- tai pelimoottoreita, sillä ohjelmoijan on hyvä tietää, kuinka käytetyt rajapinnat toimivat. Valmiita pelimoottoreita hyödyntämällä Canvasin ja OpenGL ES 1.0:n toiminta ei välttämättä aukene yhtä helposti, kun asiat tapahtuvat automaattisemmin. Tässä projektissa on myös huomioitava, että koska peli on mekaniikaltaan varsin yksinkertainen, valmiit pelimoottorit (kuten AndEngine, Cocos2D yms.) sisältävät paljon komponentteja, joita tämä peli ei tarvitse. Tämä tekisi pelistä tarpeettoman raskaan ja vaikuttaisi erityisesti mm. latausaikoihin. Koska tarkoitus on verrata kahta eri menetelmää, on reiluinta tehdä molemmat versiot pelistä kokonaan itse, lisäten vain tarvittavan koodin. Näin saadaan mahdollisimman puolueeton asetelma, jossa kummastakaan menetelmästä ei ole tehty tarpeettoman raskasta.

Canvasin käyttäminen on yksi helpoimpia tapoja piirtää kuvaa Android-applikaatiossa, mistä syystä se on valittu tähän projektiin. OpenGL ES sen sijaan on hieman vaativampi, mutta suorituskyvyltään selvästi Canvasia parempi (mikä tulee esille tässä opinnäytetyössä). OpenGL ES:n eri versioista 1.0 on helpoiten ymmärrettävä ja vaatii ohjelmoijalta vähiten työtä, joskin myös tarjoaa vähemmän mahdollisuuksia vaikuttaa asioihin, verrattuna OpenGL ES:n uudempiin versioihin. Tästä syystä se ei välttämättä ole kaikkein paras ratkaisu, kun katsotaan suorituskykyä ja vaikutusmahdollisuuksien määrää, mutta on kuitenkin helpoin oppia. Aloittelijan lienee siis syytä tutustua ensin versioon 1.0, jonka jälkeen – mikäli näkee sen tarpeelliseksi – siirtyä uudempiin versioihin. Tämän opinnäytetyön peliprojektin ollessa yksinkertainen 2D-peli, OpenGL ES 1.0 kattaa kaiken tarvittavan.

OpenGL ES 1.0 on myös siksi muita parempi ratkaisu, että jokainen Android-laite tukee sitä. Toisaalta OpenGL ES 2.0 toimii Androidin versiolla 2.2 (ja uudemmilla), mikä käsittää 99,9% nykyisistä Android-laitteista (helmikuussa 2014), joten tässä asiassa versioilla 1.0 ja 2.0 ei juuri ole eroa. OpenGL ES 3.0 tosin on tässä asiassa vaativampi, sillä sen käyttö tarvitsee vähintään Android-käyttöjärjestelmän version 4.3, joka on tätä

projektia tehdessäni vielä varsin uusi ja siksi yleisessä käytössä harvinainen. (Android Developers, 2014b, Android Developers, 2014c)

## 1.4 Termistöä

Säie

(engl. *thread*) Säie on luokka, joka mahdollistaa useamman koodirivin suorittamisen yhtäaikaisesti.

Tavallisesti laitteen jokainen prosessorin ydin lukee pienen pätkän yhden prosessin koodia, jonka jälkeen siirtyy seuraavaan prosessiin, josta taas seuraavaan jne. Tämä kuitenkin tapahtuu niin nopeasti, että vaikuttaa siltä, että useampi prosessi toimii yhtäaikaisesti. Käytännössä kuitenkin prosessit toimivat täten vuorotellen.

Säie mahdollistaa useamman prosessorin ytimen hyödyntämisen yhden prosessin suorittamisessa. Tämä tarkoittaa sitä, että esimerkiksi kaksi säiettä voi saada kaksi prosessorin ydintä ajamaan samaa sovellusta yhtä aikaa. Erityisesti tästä on hyötyä, mikäli halutaan parantaa ohjelman ja laitteen suorituskykyä hyödyntämällä mahdollisimman monta laitteen prosessoriydintä. Tästä on hyötyä myös ohjelman eri komponenttien synkronoinnissa, eli eri toimintojen yhdenaikaistamisessa.

Automaattinen roskienkerääjä

(engl. *garbage collector, GC*) Java-ohjelmointikieleen kuuluva automaattinen muistinhallintakomponentti. Roskienkerääjä etsii ja vapauttaa muistipaikkoja, jotka eivät enää ole käytössä. Automaattinen roskankeräys on käytössä useissa muissakin ohjelmointikielissä, esimerkiksi Pythonissa ja Rubyssa.

Roskienkerääjän vastakohta on manuaalinen muistinhallinta, joka tarkoittaa sitä, että ohjelmoijan on itse hallittava muistipaikkoja.

Käytännössä tämä vaatii ohjelmoijalta enemmän työtä, mutta tavallisesti parantaa ohjelman suorituskykyä. (Wikipedia, 2014)

Roskienkerääjä voi Android-käyttöjärjestelmässä pysäyttää ohjelman ajon pisimmillään 0,6 sekunniksi, mikä voi olla ohjelman suorituksen kannalta pitkä aika. Ohjelman tulisi toimia mahdollisimman moitteettomasti ja tasaisesti, ja puolenkin sekunnin viive voi turhauttaa ohjelman käyttäjän. Täten automaattinen roskienkerääjä ei välttämättä ole käyttäjää ajatellen paras ratkaisu. (Zechner 2011, 182)

# 2 Canvas ja OpenGL ES 1.0

Android mahdollistaa eri menetelmiä grafiikan toteuttamiseen (päivittämiseen ja piirtämiseen). Mitä peligrafiikoihin tulee, tavallisesti parhaat vaihtoehdot ovat joko Canvas-luokan tai OpenGL ES -rajapinnan käyttäminen. Projektissani käytän molempia, verraten lopulta näiden tuloksia, ongelmakohtia ja ratkaisumenetelmiä.

Canvas-luokan käyttö vaatii myös SurfaceView-luokan käyttöä, joka on perinnöllinen View-luokalle, joka puolestaan käsittää pääasiassa kaiken näkyvän Android API:ssa.

OpenGL ES 1.0 sen sijaan käyttää omia luokkiaan. SurfaceViewin tilalla on GLSurfaceView, joka on perinnöllinen tavalliselle SurfaceView-luokalle. Tämän sisälle tulee vielä Renderer-rajapinta, jossa itse kuvan piirtäminen tapahtuu.

## 2.1 Teknisen puolen vertailu

Canvas on aloittelevalle pelintekijälle helppokäyttöisempi kuin mikään OpenGL ES:n nykyisistä versioista, joskin itse laitteelle raskaampi suorittaa. OpenGL ES 1.0, jota tässä projektissa käytetään, on OpenGL ES:n versioista kaikkein yksinkertaisin, joskin raskain (ollen silti kevyempi kuin Canvas). (Android Developers, 2014d)

OpenGL ES toimii C-ohjelmointikielellä, mutta sillä on Java wrapper, joka mahdollistaa rajapinnan käytön Javalla. Tämä näkyy pääasiassa siten, että vaikka Java itse ei varsinaisesti vaadi ohjelmoijalta muistinhallintaa (automaattisen roskankerääjän huolehtiessa siitä), C-pohjainen OpenGL ES vaatii. Manuaalinen muistinhallinta voikin koitua alussa ongelmaksi, mikäli ohjelmoijalla ei ole asiasta kokemusta.

OpenGL ES tukee myös 3D-grafiikkaa, toisin kuin Canvas. Itse asiassa OpenGL ES toimii joka tapauksessa 3D-ympäristössä, vaikka ohjelman grafiikat ja design olisivatkin kaksiulotteisia.

OpenGL ES mahdollistaa myös piirrettävien objektien jakamista joukkoihin (batch), mikä mahdollistaa nopeamman ja vähemmän muistia käyttävän piirtämisen. Canvasilla tämä ei ole mahdollista, vaan Canvas laittaa kaiken piirrettävän grafiikan yhteen joukkoon. Tämä asettaa (laitteesta riippuvan) muistirajan piirrettäville grafiikoille ja toimii muutenkin hitaammin. Koska kuitenkin tämän opinnäytetyön peliprojektin koko on varsin pieni, kyseisen menetelmän hyödyntäminen ei ole tässä tapauksessa tarpeellista. Mikäli projekti olisi laajempi, tämä metodi voi olla hyvinkin tärkeä, jotta peli toimisi myös pienitehoisemmilla laitteilla.

## 2.2 Säikeiden käyttö

Android-käyttöjärjestelmä käynnistää aina sovelluksen käynnistyessä UIThreadiksi nimetyn säikeen. UIThreadin pääasiallinen tarkoitus on yksinkertaisesti päivittää ohjelma ja pitää sitä yllä, sulkea ohjelma ja asettaa se tauolle tarvittaessa.

Kuitenkin, jos UIThread kuormittuu liikaa ja ns. “tukkeutuu”, käyttöjärjestelmä tulkitsee ohjelman menneen jumiin. Näin tapahtuu, mikäli säikeen suorittaminen kestää kauemmin kuin viisi sekuntia. Tällöin käyttöjärjestelmä antaa virheilmoituksen ja pakottaa sovelluksen sulkeutumaan, mikä on luonnollisesti aina ikävä asia käyttäjän kannalta. Jos siis UIThread joutuu ainoana säikeenä suorittamaan kaiken pelissä tarvittavan pelitilanteen ja grafiikan päivityksen, prosessissa voi mennä liian kauan ja ohjelma kaatuu. Tämä ongelma voidaan kiertää tekemällä uusi säie, joka ottaa tehtäväkseen ajaa pelitilanteen ja grafiikoiden päivittämisen, jättäen UIThreadille pelkästään käyttöjärjestelmän vaatimat tehtävät ohjelman ajamiseen. Näin UIThread ei pääse tukkeutumaan, ja riski ohjelman kaatumiselle pienenee huomattavasti. (Android Developers, 2014e)

Canvasin kanssa työskennellessä tarvitaan erillinen säie, mikäli ei haluta kuormittaa UIThreadia liikaa. Ohjelmoijan on tehtävä tämä säie itse, sillä Android SDK:ssa ei ole sellaista valmiina. Tavallisesti tämä tarkoittaa lisää työtä, kun säie pitää sammuttaa, keskeyttää ja jatkaa oikealla tavalla käyttöjärjestelmän vaatimusten mukaan. Jos

esimerkiksi prosessi lopettaa toimintansa, mutta säiettä ei ole sammutettu asiaankuuluvasti, voi tulla ongelmia.

OpenGL ES sen sijaan järjestää automaattisesti säikeen omassa GLSurfaceView-luokassaan, mikä helpottaa ja vähentää ohjelmoijan työtä Canvasilla työskentelyyn verrattuna.

Luonnollisesti OpenGL ES:llä tehtyyn projektiin on mahdollista lisätä omiakin säikeitä, mutta asian kanssa on oltava varovainen. Periaatteessa OpenGL ES:n voi saada toimimaan useammassa eri säikeessä, mutta eri laitteistojen ajurit eivät välttämättä tue säikeistystä, mikä puolestaan mahdollisesti aiheuttaa ongelmia. Täten, mikäli ongelmia esiintyy, niitä ei välttämättä voi korjata, koska kyse on ajureista. Helpoimmalla pääsee, kun työskentelee vain yhden tai kahden säikeen parissa. (Zechner 2011, 281)

# 3 Projektin aikana esiintyneitä ongelmia ja ratkaisuja

## 3.1 Bittikarttatiedostojen skaalaus laitteen pikselitiheyden mukaan

Koska Android-laitteita on paljon erilaisia, niiden tekniset erot luonnollisesti vaikuttavat myös ohjelmointiin. Android-laitteilla on tällä hetkellä neljä pikselitiheys-luokkaa (*low*, *medium*, *high* ja *extra high*). Kun ohjelma käynnistetään, käyttöjärjestelmä antaa ohjelmalle tiedon laitteen pikselitiheydestä, jolloin ohjelma osaa valita käyttöön oikean kokoiset kuvatiedostot. Tämän vuoksi ohjelmassa olisi hyvä olla mukana useampi erikokoinen versio kuvatiedostoista. (Android Developers, 2014f)

Käyttöjärjestelmä tekee skaalauksen automaattisesti. Jos sovelluksessa on esimerkiksi mukana vain *medium*-tason grafiikoita, ohjelmaa käytettäessä *high*-tason laitteella kuvatiedostot skaalataan 1,5-kertaisiksi alkuperäisestä. Jos kuitenkin ohjelmaan on laitettu mukaan *high*-tason grafiikat oikeaan kansioon, käyttöjärjestelmä käyttää niitä, eikä skaalaa ollenkaan.

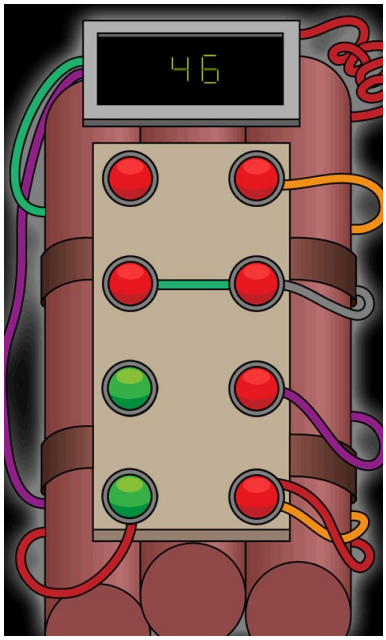
Itse ajauduin pienehköön ongelmatilanteeseen juuri tämän asian johdosta. Ohjelmaa tehdessäni käytin placeholder-grafiikoita, jotka pidin *medium*-tason grafiikoille tarkoitetussa “drawable-mdpi”-kansiossa (*mdpi = medium dots per inch*, medium-tason pikselitiheys). Kuitenkin laite, jolla ohjelmaa testasin, käytti *high*-tason näyttöä. Luodessani piirrettävää objektia olin antanut sille parametreinä mm. korkeuden ja leveyden tiettyinä lukuina bittikarttatiedoston mukaan. Käyttöjärjestelmä kuitenkin käsitteli kuvatiedostoja skaalaten ne automaattisesti, mikä vaikeutti animaation tekemistä, sillä skaalauksesta ei huomautettu missään vaiheessa, eivätkä laitteiden pikselitiheydet aina ole itsestäänselvyyksiä.

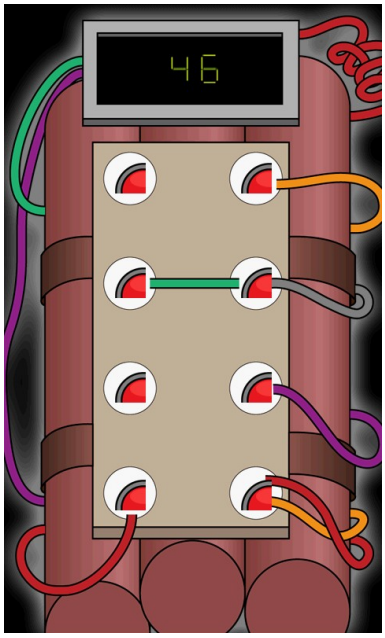
Tarkoitus oli tehdä yksinkertainen painike, jossa on kaksi framea: toisessa painike on punainen ympyrä läpinäkyvällä taustalla, kun taas toisessa framessa ympyrä on vihreä. Framet ovat vierekkäin samassa kuvatiedostossa (ks. kuva 2).

KUVA 2: Kaksi framea samassa kuvatiedostossa.

Koska kuvatiedoston koko on 80 * 40 pikseliä, ja frameja on kaksi ja ne ovat vierekkäin, asetin koodissa yhden framen leveydeksi 40 pikseliä ja korkeudeksi myös 40 pikseliä. Koska en luonnollisesti halunnut molempia frameja piirrettävän yhtä aikaa, käytin Rectangle-luokan muuttujaa (*sourceRect*), jolla hain tietyn osan kuvatiedostosta, jonka piirtäisin. Tämän lisäksi käytin Rectangle-luokan muuttujaa (*destRect*), jolla määritin piirrettävän kuvan sijainnin näytöllä.

Tämä johti ongelmaan, koska käytin tarkkaan määriteltyjä muuttujia kuvatiedoston leveydelle ja korkeudelle. *SourceRect*- ja *destRect*-muuttujien saadessa koon 40 * 40, kuvasta näkyi liian vähän; reilusti vähemmän kuin koko frame (ks. kuvat 3 ja 4).




KUVAT 3 ja 4: Vasemmanpuoleinen kuva (kuva 3) on esimerkki siitä, miltä nappien oli tarkoitus näyttää. Oikeanpuoleisessa (kuva 4) näkyy, miltä napit näyttivät.

Käyttäessäni Android SDK:n Log-toimintoa hain kuvatiedoston leveyden ja korkeuden, jolloin Android SDK:n LogCat-työkalu ilmoitti kuvan kooksi 120 * 60 pikseliä. Koska laite, jolla testasin ohjelmaa, käytti *high*-tason näyttöä ja grafiikat olivat *medium*-tason kansiossa, käyttöjärjestelmä oli automaattisesti skaalannut kuvatiedoston. Tämä johti siihen, ettei yhden framen koko enää ollutkaan 40 * 40, vaan 60 * 60 (1,5-kertainen alkuperäiseen verrattuna molemmilla akseleilla). Tämä kuitenkin oli korjattavissa korvaamalla tarkat pikseliarvot API:n valmiilla dynaamisilla muuttujilla. Vaihdoin framen leveys- ja korkeusmuuttujat hakemaan arvonsa suoraan kuvatiedostosta (käskyillä *bitmap.getWidth()* ja *bitmap.getHeight()*), minkä jälkeen kuva näkyi oikean kokoisena. Tämä johtuu siitä, että edellä mainitut käskyt eivät hae tietoja alkuperäisestä kuvatiedostosta, vaan skaalatusta versiosta.

Vaikka ongelmaan onkin varsin yksinkertainen ratkaisu, siitä huolimatta Androidille sovellusta tehdessä kannattaa mahdollisuuksien mukaan tehdä erikseen myös eri pikselitiheystasojen grafiikat, jottei käyttöjärjestelmän tarvitsisi venyttää niitä. Luonnollisesti venytetyt kuvat pikselöityvät, eivätkä kutistetutkaan kuvat näytä kovin hyviltä. Kuvankäsittelyohjelmat tekevät kuvatiedostojen skaalauksen melko pitkälti poikkeuksetta paremmin kuin Androidin skaalausmenetelmät.

Tästä opin, että Android käyttöjärjestelmänä vaatii kuvatiedostojen dynaamisen käsittelyn siten, että kuvatiedostojen koot haetaan komennoilla kuvatiedostoista, eikä aseteta tarkkaan määritettyjä numeroarvoja.

## 3.2 Animointi

Canvasilla animointi on loppujen lopuksi varsin helppoa käyttäen spritesheetiä. Otetaan esimerkiksi kuva 2, jossa on kaksi nappia – punainen ja vihreä. Koska kuvatiedostossa on kaksi framea, jotka ovat vierekkäin, saadaan helposti yhden framen leveydeksi koko kuvan leveys jaettuna kahdella. Framen korkeus on sama kuin kuvatiedoston korkeus.

Tehdään Rectangle-muuttuja *sourceRect*, jolla määritetään kumpi frame piirretään. Kun ohjelma näyttää punaisen napin, *sourceRectin* vasen laita saa arvon nolla, ollen aivan kuvan vasemmassa laidassa. Oikea laita sen sijaan on kuvan puolivälissä, ja ylä- ja alareunat ovat kuvatiedoston kanssa samat. Jos taas ohjelman halutaan piirtävän vihreä ympyrä, framen vasen laita asetetaan kuvatiedoston puoleen väliin, ja oikea reuna tiedoston oikeaan laitaan. Lopulta Rectangle-muuttuja *destRect* määrittää, mihin kuva piirretään ja minkä kokoisena.

OpenGL ES sen sijaan ei ole näin yksinkertainen käyttää animoinnissa, ja tämä tuottikin tarpeeksi ongelmia pysäyttääkseen projektin varsin pitkäksi aikaa. Löysin kuitenkin yhden ratkaisun, joka on todettu toimivaksi.

Ensiksi on otettava huomioon, että OpenGL ES:n kanssa toimiessa kuvatiedoston mittojen tulee noudattaa luvun kaksi potenssiarvoja. Toisin sanoen, kuvatiedoston korkeuden ja leveyden tulee olla jokin kahden potenssista (2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048...).

Myös tekstuurien lataaminen OpenGL ES:llä on huomattavasti työläämpää kuin Canvasilla. Huomioonotettavaa tässä on myös, ettei itselläni ollut juuri ollenkaan kokemusta muistinhallinnasta ohjelmoinnissa, joten suurimman osan ajasta minulla ei ollut aavistustakaan, mitä tein. Lopulta kuitenkin aloin hiljalleen ymmärtää tekemisiäni sen verran, että sain animaation toimimaan.

Ensin on annettava *vertex*-muuttujien arvot. Nämä määrittävät tekstuurin rajat, ollen kolmioiden kärkiä. Koska OpenGL ES käsittelee tekstuurit kolmioina, mutta kuvatiedosto koostuu neliöistä, käytin neljää kärkeä ja kahta kolmiota. Tämän jälkeen on tehtävä sama tekstuureille.

Esimerkkikoodi:

```
private float vertices[] = {
                        -0.15f, -0.1f,  0.0f,   // V1 - bottom left
                        -0.15f,  0.1f,  0.0f,   // V2 - top left
                        0.15f, -0.1f,  0.0f,    // V3 - bottom right
                        0.15f,  0.1f,  0.0f     // V4 - top right
```

```
            };
private FloatBuffer textureBuffer1;
private FloatBuffer textureBuffer2;
private float texture1[] = {
                        0.0f, 1.0f,     // top left (V2)
                        0.0f, 0.0f,     // bottom left (V1)
                        0.5f, 1.0f,     // top right (V4)
                        0.5f, 0.0f      // bottom right (V3)
            };
private float texture2[] = {
                        0.5f, 1.0f,     // top left (V2)
                        0.5f, 0.0f,     // bottom left (V1)
                        1.0f, 1.0f,     // top right (V4)
                        1.0f, 0.0f      // bottom right (V3)
            };
```

Tässä on alustettu *vertexit* (kärjet), jolloin kuvalle on annettu koko. Tämän jälkeen on luotu kaksi *FloatBuffer*-muuttujaa, yksi molemmille framelle. Sitten annetaan tekstuurien rajat *texture1* ja *texture2* -taulukoille: texture1 on vasemmanpuoleinen punainen ympyrä, ja texture2 oikeanpuoleinen, vihreä ympyrä.

Tähän mennessä asiat olivat vielä yksinkertaisia, kunnes näitä buffereita piti käyttää itse koodissa:

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

// TEXTURE 1
byteBuffer = ByteBuffer.allocateDirect(texture1.length * 4);
byteBuffer.order(ByteOrder.nativeOrder());
textureBuffer1 = byteBuffer.asFloatBuffer();
```

```
textureBuffer1.put(texture1);
textureBuffer1.position(0);
// TEXTURE 2
byteBuffer = ByteBuffer.allocateDirect(texture2.length * 4);
byteBuffer.order(ByteOrder.nativeOrder());
textureBuffer2 = byteBuffer.asFloatBuffer();
textureBuffer2.put(texture2);
textureBuffer2.position(0);
```

*ByteBufferit* varaavat tarvittavan määrän laitteen muistista, ja asettavat sekä *vertexit*, että tekstuurit sinne.

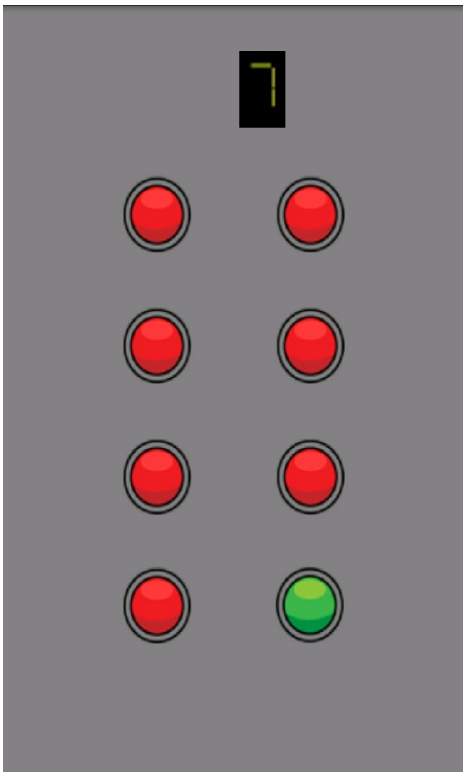
Mikä tässä pääasiallisesti itselleni aiheutti ongelman, oli pätevien tutoriaalien puute. Itse muistinkäsittelystä minulla ei ollut mitään kokemusta, joten apua piti etsiä internetistä. Suurin osa löytämistäni OpenGL ES -tutoriaaleista käsittelee 3D-grafiikkaa, ja 2D-grafiikkaan liittyvät ohjeet puolestaan opettavat piirtämään vain yhden kuvan ilman animaatiota. Minun piti siis itse koittaa ymmärtää *ByteBuffereiden* toimintaa ja soveltaa erilaisia tulkintojani. Lopulta päädyin tähän tulokseen, jossa molemmat tekstuurit (eli framet) ladataan erikseen omiin *FloatBuffereihinsa*. Loppujen lopuksi ratkaisu on toimiva, joskin muistinhallintaan perehtymättömänä on mahdoton sanoa, onko metodi aivan optimaalisin mahdollinen. Kuitenkin ottaen huomioon ohjelman käynnistyksen nopeus OpenGL ES -versiossa Canvas-versioon nähden, käyttämäni metodi on tavallisesti ainakin Canvasia nopeampi (lisää kohdassa 4.1 – Ohjelman latausaika).

## 3.3 Sovellus toimii emulaattorissa, muttei laitteessa

Yksi tärkeimmistä asioista, mitä tulee sovelluksen testaamiseen, on ettei emulaattoriin saa luottaa liikaa. Vaikka Android SDK:n emulaattorit ovatkin varsin toimivia ja emuloivat käyttöjärjestelmää hyvin, laitteistoa ja ajureita se ei kuitenkaan emuloi aina

edes välttävästi. Tästä syystä projektin edetessä on syytä tarkistaa sovelluksen toiminta välillä myös oikealla Android-laitteella – mitä useammalla eri laitteella, sen parempi.

Saatuani pelin valmiiksi OpenGL ES:ää käyttäen, se toimi täydellisesti emulaattorissa. Kun sitten asensin ohjelman myös omalle Samsung Galaxy S 2:lleni (Android-käyttöjärjestelmän versio 2.3.5), grafiikat pettivät täysin. Vain osa objekteista oli näkyvissä: painikkeet ja oikeanpuoleinen digitaalinumero, joka näyttää yksittäiset sekunnit jäljellä olevasta ajasta. Taustakuva ja vasemmanpuoleinen digitaalinumero (joka näyttää kymmenet sekunnit) puuttuivat, ja tilalla oli vain tummanharmaata väriä (ks. kuva 5). Sama tapahtui Asus Nexus 7 -tabletilla (jossa Androidin versio 4.4.2).



KUVA 5: Kuvakaappaus Galaxy S 2:n ruudusta. Vaikka aikaa on jäljellä 17 sekuntia, vain numero 7 näkyy, luvun 1 ollessa piilossa. Luku on kahtena erillisenä objektina, joista tässä tapauksessa vain toinen on näkyvissä. Myös taustakuva puuttuu (vrt. kuvat 1 ja 3).

Ongelmasta mutkikkaan teki erityisesti se, että jokaisen objektin piirtäminen oli ohjelmoitu niin monilta osin samankaltaiseksi kuin mahdollista: tekstuurien lataus, animointi ja kuvien esitys on kukin tehty periaatteessa samoista koodinpätkistä.

Merkille pantavaa on erityisesti se, että molemmat numerot laitetaan näkyville jopa samassa funktiossa, joka on käytännössä sama koodi kahdesti vain pienillä muutoksilla:

```
private int tens;       // Time left (left number)
private int ones;       // Time left (right number)

// DRAW
public void draw(GL10 gl) {
/** DRAW TENS */
// bind the previously generated texture, according to correct number
if (tens == 0) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures0[0]); }
else if (tens == 1) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures1[0]); }
else if (tens == 2) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures2[0]); }
else if (tens == 3) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures3[0]); }
else if (tens == 4) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures4[0]); }
else if (tens == 5) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures5[0]); }
else if (tens == 6) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures6[0]); }
else if (tens == 7) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures7[0]); }
else if (tens == 8) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures8[0]); }
else { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures9[0]); }

// Enable client states (must be disabled after drawing!)
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

// Set the face rotation
gl.glFrontFace(GL10.GL_CW);

// Some devices might require this
gl.glEnable(GL10.GL_TEXTURE_2D);

// Point to our vertex buffer
gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertexBuffer);
if (tens == 0) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, textureBuffer0); }
else if (tens == 1) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer1); }
else if (tens == 2) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer2); }
```

```
else if (tens == 3) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer3); }
else if (tens == 4) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer4); }
else if (tens == 5) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer5); }
else if (tens == 6) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer6); }
else if (tens == 7) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer7); }
else if (tens == 8) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer8); }
else { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, textureBuffer9); }

// Move to right position (x, y, z)
gl.glTranslatef(tensPosition.x, tensPosition.y, 0.0f);

// Draw the vertices as triangle strip
gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, vertices.length / 3);

// Disable the client state before leaving
gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

// RESET translateF
gl.glLoadIdentity();

/** DRAW ONES */
if (ones == 0) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures0[0]); }
else if (ones == 1) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures1[0]); }
else if (ones == 2) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures2[0]); }
else if (ones == 3) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures3[0]); }
else if (ones == 4) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures4[0]); }
else if (ones == 5) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures5[0]); }
else if (ones == 6) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures6[0]); }
else if (ones == 7) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures7[0]); }
else if (ones == 8) { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures8[0]); }
else { gl.glBindTexture(GL10.GL_TEXTURE_2D, textures9[0]); }

gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
```

```
gl.glFrontFace(GL10.GL_CW);

gl.glEnable(GL10.GL_TEXTURE_2D);

gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertexBuffer);
if (ones == 0) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, textureBuffer0); }
else if (ones == 1) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer1); }
else if (ones == 2) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer2); }
else if (ones == 3) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer3); }
else if (ones == 4) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer4); }
else if (ones == 5) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer5); }
else if (ones == 6) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer6); }
else if (ones == 7) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer7); }
else if (ones == 8) { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0,
            textureBuffer8); }
else { gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, textureBuffer9); }

gl.glTranslatef(onesPosition.x, onesPosition.y, 0.0f);

gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, vertices.length / 3);

gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

gl.glLoadIdentity();
}
```

Koodi on molemmissa periaatteessa sama; vain sijainti ja valittava numero vaihtuvat (muuttujat *ones* ja *tens*). Silti vain jälkimmäisenä toteutettu numero näkyy ruudulla.

Internetistä löytyy muutamia ohjeita, jotka neuvovat korjaamaan tämänkaltaisia ongelmia. Ilmeisesti muistamisen arvoista on, että kuvatiedostojen koot on oltava luvun

kaksi potenssin mukaisia (kuten mainittu aiemmin), mutta koska olin jo tehnyt niin, se ei ratkaissut ongelmaa. (Stack Overflow, 2014)

Toinen merkittävä asia on myös se, että toimiessa kaksiulotteisten grafiikoiden kanssa, on syytä pitää objektien z-akselin sijainti arvossa 0.0f (*float*-muuttuja). Ylläolevan koodin riveissä

```
gl.glTranslatef(tensPosition.x, tensPosition.y, 0.0f);
```

ja

```
gl.glTranslatef(onesPosition.x, onesPosition.y, 0.0f);
```

objektien sijainniksi määritetäänkin z-akselilla nimenomaan 0,0f.

Ongelman lähde paljastui *MyGLSurfaceView*-luokan *onDrawFrame*-funktiosta, jossa määritetään "kameran" sijainti samankaltaisella koodirivillä.

```
// DRAW
// clear Screen and Depth Buffer
gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
// Reset the Modelview Matrix
gl.glLoadIdentity();
// Drawing
gl.glTranslatef(0.0f, 0.0f, -5.0f); // move 5 units INTO the screen, is
                                       the same as moving the camera 5
                                       units away
bg.draw(gl);                        // Draw background
clockNumbers.draw(gl);              // Draw clock numbers
for (int i = 0; i < _lights.size(); i++) {      // Draw lights/buttons
            BombLight b = _lights.get(i);
            b.draw(gl);
}
```

Kommentin “move 5 units INTO the screen...” perusteella olen lainannut kyseisen koodinpätkän samalta internetsivustolta, jonka avulla alunperin olin opetellut peliohjelmointia Androidille, ja joka perustuu Mario Zechnerin kirjaan “Beginning Android Games”. Kyseisessä tutoriaalissa neuvotaan siirtämään kameraa viisi yksikköä pois päin piirretyistä kuvioista. (Against the Grain, 2014) Täten, vaikka itse objektit ovatkin z-akselin sijainniltaan kohdassa 0.0, *onDrawFrame*-funktion *glTranfslatef*-komento vaikuttaa kaikkiin sen jälkeen tuleviin *glTranslatef*-komentoihin. Tässä tapauksessa siis taustakuvan (*“bg”*), numeroiden (*“clockNumbers”*) ja nappien (*“BombLight”*-luokka) *draw*-funktioissa kutsuttuihin komentoihin. Kyseisen *glTranslatef*-komennon arvot nollautuvat vasta *glLoadIdentity*-komennolla, joka edellisessä koodiesimerkissä on ennen *glTranslatef*-komentoa.

Vaihdoin *glTranslatef*-komennon arvoiksi “0.0f, 0.0f, 0.0f” aikaisemman “0.0f, 0.0f, -5.0f” sijaan. Nyt peli toimii kuten pitääkin ja kaikki objektit näkyvät.

Tämän ongelman myötä opin, ettei tutoriaaleihin ja emulaattoriin kannata luottaa täysin. Vaikka molemmat koodit toimivatkin emulaattorissa, eivät ne silti toimi kaikissa (jos yhdessäkään) laitteessa.

Toimiessa OpenGL ES:n ja 2D-grafiikoiden kanssa, on syytä muistaa pitää aina kaikki objektit (ml. kamera) z-akselilla sijainnissa 0.0. Muuten voi tulla hyvinkin merkillisiä ongelmia. Tällaisessa tilanteessa ongelmanratkaisusta erityisen vaikean tekee se, ettei ongelma vaikuta juurikaan loogiselta, koska täysin samalla tavalla tehdyistä objekteista osa toimii ja osa ei.

# 4 Suorituskyky

## 4.1 Ohjelman latausaika

Sovelluksen latausajan voi mitata mm. siten, että laittaa – projektista riippuen joko *SurfaceView*- tai *Renderer*-luokan – ensimmäisenä käynnistyvän funktion alkuun komennon, joka hakee laitteen sen hetkisen ajan millisekunnin tarkkuudella. Toinen aikatarkistus laitetaan samaisen funktion loppuun ja verrataan näiden aikojen eroa. On myös syytä asettaa kolmas aikatarkistus sen funktion loppuun, jossa kuva piirretään, jotta tiedetään ohjelman käynnistyksen ja ensimmäisen kuvan piirtämisen välinen ero.

Canvasilla työskennellessä ensimmäisenä ajettava funktio on *SurfaceView*-luokan rakentajafunktio. Rendererin kanssa kyseinen funktio taas on *Renderer*-rajapinnan *onSurfaceCreated*-funktio.

Koodiesimerkki (Canvas):

```
public class SurfaceViewBomb extends SurfaceView
            implements SurfaceHolder.Callback, SensorEventListener {

// ...
// Class variables (hidden in this example as unnecessary)
// ...

/** test loading time variables */
private long loadTimeStart = 0;                 // when loading begins
private long loadTimeStop = 0;                  // when loading is done
private long loadTimeDelta = 0;                 // subtraction

public SurfaceViewBomb(Context context) {
            super(context);
```

```
        /** test loading time start */
        // Time when loading begins
        loadTimeStart = System.currentTimeMillis();

        // ...
        // Everything that is loaded, initialized or otherwise handled at
        // the start of the game (hidden in this example as unnecessary)
        // ...

        /** test loading time stop */
        // Time when loading is done
        loadTimeStop = System.currentTimeMillis();
        // Subtraction
        loadTimeDelta = (loadTimeStop - loadTimeStart);
        // Log results
        Log.d("", "TIME STARTED: " + loadTimeStart);
        Log.d("", "TIME NOW:     " + loadTimeStop);
        Log.d("", "TIME DELTA:   " + loadTimeDelta);
}
```

Kyseinen koodi hakee luokan käynnistyksen ajankohdan (*loadTimeStart*), jonka jälkeen ladataan ja alustetaan kaikki pelissä tarvittava. Tämän jälkeen haetaan uusi aika (*loadTimeStop*) ja lasketaan näiden aikojen erotus (*loadTimeDelta*).

OpenGL ES:llä testi menee käytännössä samalla tavalla, joskin tarkistus tapahtuu Rendererin *onSurfaceCreated*-funktiossa:

```
public class MyRenderer implements GLSurfaceView.Renderer {

// ...
// Class variables (hidden in this example as unnecessary)
// ...

/** test loading time variables */
private long loadTimeStart = 0;
```

```
private long loadTimeStop = 0;
private long loadTimeDelta = 0;

public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            /** test loading time start */
            // Time when loading begins
            loadTimeStart = System.currentTimeMillis();

            // ...
            // Everything that is loaded, initialized or otherwise handled at
            // the start of the game (hidden in this example as unnecessary)
            // ...

            /** test loading time stop */
            // Time when loading is done
            loadTimeStop = System.currentTimeMillis();
            // Subtraction
            loadTimeDelta = (loadTimeStop - loadTimeStart);
            // Log results
            Log.d("", "TIME STARTED: " + loadTimeStart);
            Log.d("", "TIME NOW:     " + loadTimeStop);
            Log.d("", "TIME DELTA:   " + loadTimeDelta);
}
```

Periaatteessa pelin ei kuitenkaan käytännössä voida sanoa latautuneen ennen kuin peli alkaa päivittää itseään (eli näyttää käyttäjälle näkyvää kuvaa). Täten latausaikaa laskiessa on oleellista tietää myös kuinka kauan tämän ensimmäisen pelitilanteen päivityksen ja kuvanpiirron tekemisessä kestää.

Tämä on yksinkertaista testata laittamalla kertaluontoinen testi sen funktion loppuun, jossa piirtäminen tapahtuu. Canvasin kanssa toimiessa tein itse *SurfaceView*-luokkaan *render*-funktion, joka kutsutaan omatekemässäni säikeessä, jonka tehtävänä on sekä päivittää pelitilanne että piirtää jokainen frame. OpenGL ES:n *Renderer* toimii omalla valmiilla säikeellään, joka automaattisesti kutsuu *onDraw*-funktiota joka kerran, kun pelitilanne päivitetään.

Esimerkki Canvasille tehdystä testistä:

```
// As class variables
private long loadTimeDraw = 0;                  // time after first draw
private Boolean loadTimeDrawTaken = false;      // true after first draw,
                                                   so doesn't test that more
                                                   than once

public void render (Canvas canvas) {

            // ...
            // draw everything (hidden in this example as unnecessary)
            // ...

            /** test loading time after first frame */
            // Check boolean, so doesn't make this check more than once
            if (!loadTimeDrawTaken) {
                        // Get time after first frame
                        loadTimeDraw = System.currentTimeMillis();
                        // Subtraction
                        loadTimeDelta = loadTimeDraw - loadTimeStart;
                        // Log result
                        Log.d("", "TIME DRAW:    " + loadTimeDelta);
                        // Boolean set to true, so doesn't check more than
                        once
                        loadTimeDrawTaken = true;
            }
}
```

OpenGL ES:n kanssa logiikka on pitkälti sama, joten sitä ei tarvinne kirjoittaa erikseen. Vain funktioiden nimet vaihtuvat.

Tein nämä testit Android SDK:n emulaattorilla käyttäen kahta eri käyttöjärjestelmän versiota (2.2 ja 4.0), kuten myös kolmella eri Android-laitteella: Samsung Galaxy S 2, Samsung Galaxy S Plus ja Asus Nexus 7.

Käynnistin sovelluksen kullakin laitteella viisi kertaa, otin tulokset ylös ja laskin niiden keskiarvot. Lopputulokset näkyvät seuraavassa taulukossa:

| Laite, käyttöjärjestelmän versio ja projekti | | Latauksen alusta latauksen loppuun (rakentaja / onSurfaceCreated) (millisekunnit) | Latauksen alusta 1. framen valmiiksi piirtämiseen (millisekunnit) | Latauksen ja 1. framen erotus (eli käytännössä ensimmäisen framen päivitys- ja piirtonopeus) (millisekunnit) |
|---|---|---|---|---|
| Emulaattori (2.2) | Canvas | 448,2 | 594,8 | 146,6 |
| | OpenGL ES | 336,8 | 372,8 | 36,0 |
| Emulaattori (4.0) | Canvas | 964,8 | 1172,6 | 207,8 |
| | OpenGL ES | 1211,2 | 1263,2 | 52,0 |
| Samsung Galaxy S 2 (2.3.5) | Canvas | 240,8 | 309,6 | 68,8 |
| | OpenGL ES | 195,4 | 199,0 | 3,6 |
| Samsung Galaxy S Plus (2.3.5) | Canvas | 121,6 | 211,0 | 89,4 |
| | OpenGL ES | 80,4 | 140,4 | 60,0 |
| Asus Nexus 7 (4.4.2) | Canvas | 232,4 | 408,4 | 176,0 |
| | OpenGL ES | 279,0 | 290,0 | 11,0 |

TAULUKKO 1: Sovelluksen latausajat eri alustoilla.

Kuten tuloksista voidaan huomata, OpenGL ES:llä tehty projekti latautuu Canvasiin nähden huomattavasti nopeammin Galaxy S 2:lla, Galaxy S Plus:lla ja emulaattorilla, jossa on Androidin versio 2.2. Kuitenkin ajettaessa käyttöjärjestelmän uudemmilla versioilla (4.0 ja 4.4.2) Canvas lataa ohjelman nopeammin. Erot ovat kuitenkin pääsääntöisesti pieniä, mutta on otettava huomioon, että tämä on varsin pieni ja kevyt peli. Suuremmassa projektissa erot olisivat todennäköisesti selkeämmät.

Tässä on syytä muistaa, etteivät emulaattorit ole kovinkaan luotettavia antamaan kunnollisia testituloksia. Android 4.0 on raskaampi kuin 2.2, mutta emulaattoreiden

laskentateho on käytännössä sama. Täten itse käyttöjärjestelmä hidastaa enemmän emulaattoria, jossa on uudempi ja raskaampi käyttöjärjestelmä. Tämä ei kuitenkaan vaikuta 4.0-version emulaattorin omiin tuloksiin, jossa nähtävästi OpenGL ES on hieman Canvasia hitaampi latautumaan.

Tulosten perusteella on ilmiselvää, että jokaisessa tapauksessa latauksen ja ensimmäisen piirretyn framen välisen ajan erotus on merkittävästi pienempi OpenGL ES:llä – erityisesti Galaxy S 2:lla, jossa ero on peräti 19-kertainen. Käytännössä tämä tarkoittaa sitä, että pelitilanteen piirtämisen päivitysnopeus on OpenGL ES:ssa nopeampi. Tästä lisää seuraavassa kappaleessa.

## 4.2 Päivitysnopeus

Ruudun päivityksen nopeuden testaaminen ei sekään ole monimutkaista. Tapa, jolla itse tämän tein, vaatii vain kaksi luokkamuuttujaa ja yksinkertaisen testin.

Esimerkkikoodi Canvasilla:

```
// As class variables
/** test fps */
private long fpsTestStart = 0;      // Time when testing starts, reset when
                                       second passes
private int fpsCount = 0;           // Counts frames, zeroes when second
                                       passes

public void render (Canvas canvas) {

            // ...
            // draw everything
            // ...

            /** test fps */
```

```
            // Add 1 to frame count
            fpsCount++;
            // Get current time
            long currentTime = System.currentTimeMillis();
            // Check if second has passed
            if (currentTime - fpsTestStart >= 1000) {
                        // If second has passed, log result (frame count)
                        Log.d("", "*** FPS: " + fpsCount);
                        // Reset variables
                        fpsTestStart = currentTime;
                        fpsCount = 0;
            }
}
```

Tässä koodiesimerkissä lasketaan muuttujan *fpsCount* avulla piirrettyjen framejen määrä, ja kun sekunti on kulunut, framejen määrä ilmoitetaan *Log*-toiminnolla, asetetaan uuden sekunnin alkamisaika ja nollataan *fpsCount*-laskuri.

Ajoin sovelluksen samoilla laitteilla kuin edellisessä testissä, otin ylös kymmenen ensimmäisen sekunnin päivitysnopeus-arvot ja laskin niistä keskiarvon. Tulokset seuraavassa taulukossa:

| Laite, käyttöjärjestelmän versio ja projekti | | Päivitysnopeus (framea sekunnissa) |
|---|---|---|
| Emulaattori (2.2) | Canvas | 10,0 |
| | OpenGL ES | 32,3 |
| Emulaattori (4.0) | Canvas | 16,1 |
| | OpenGL ES | 29,7 |
| Samsung Galaxy S 2 (2.3.5) | Canvas | 21,4 |
| | OpenGL ES | 60,3 |
| Samsung Galaxy S Plus (2.3.5) | Canvas | 28,5 |
| | OpenGL ES | 58,5 |
| Asus Nexus 7 (4.4.2) | Canvas | 9,9 |
| | OpenGL ES | 60,3 |

TAULUKKO 2: Sovelluksen päivitysnopeus eri alustoilla.

Tuloksista näkee, että jokaisessa tapauksessa OpenGL ES:n päivitysnopeus on selkeästi Canvasia parempi.

Huomionarvoista on, että tässä peliprojektissa sekä Canvas että OpenGL ES rajoittavat framejen määrän 60:een sekunnissa (Canvas-versiossa tämä on määritelty manuaalisesti, kun taas OpenGL ES:ssä rajoitin on automaattinen). Canvasin kanssa tosin yksikään testilaitteista ei pääse edes sinne asti, joten rajoittimella ei tässä tapauksessa ole merkitystä. OpenGL ES:llä sen sijaan Galaxy S 2 ja Nexus 7 jopa ylittivät rajoittimen yhdellä tai kahdella framella silloin tällöin (minkä vuoksi näiden keskiarvo on hieman yli 60,0). Rajoittimen idea on pääasiassa säästää laitteen akkua, estäen peliä toimimasta nopeammin kuin on tarpeen. Näin pieni ylitys ei kuitenkaan ole merkittävä haitta.

Periaatteessa, mikäli laskemme aikaisemmassa testissä ilmenneet latauksen loppuun ajamisen ja ensimmäisen framen piirtämisen väliset aikaerot (joka on siis Galaxy S 2:lla 3,6 millisekuntia), voimme laskea tällä laitteella teoreettiseksi päivitysnopeudeksi noin 277 framea sekunnissa. Saman laskukaavan mukaan Nexus 7 pääsisi myös teoreettiselta päivitysnopeudeltaan noin 90 frameen sekunnissa, joka sekin ylittää säikeen asettaman 60:n rajan. Tähän ei kuitenkaan ole tarvetta, sillä 60 framea sekunnissa on varsin riittävä mille tahansa peliprojektille, emmekä halua kuluttaa laitteen akkua turhaan.

## 4.3 Muistin käyttö

Myös muistin käyttö on oleellinen asia älypuhelinsovelluksissa. Halvemmissa malleissa ei välttämättä ole paljoakaan keskusmuistia, joten sovellus ei saisi viedä sitä liikaa. Mikäli keskusmuistia kulutetaan, laitteen toiminta voi hidastua ja pahimmassa tapauksessa ohjelma voi kaatua.

Eclipsessä on DDMS-näkymä (*Dalvik Debug Monitor Server*), jonka avulla voi seurata mm. sovelluksen muistinkäyttöä. Tämän yksinkertaisen työkalun avulla testasin kuinka paljon käyttöjärjestelmä varaa muistia sovellukselle (*heap size*), ja kuinka paljon sitä on lopulta käytössä (*allocated memory*).

Tämänkin testin toteutin kahdella emulaattorilla ja kolmella laitteella. Annoin pelin toimia noin kymmenen sekuntia, jonka jälkeen otin tulokset ylös:

| Laite, käyttöjärjestelmän versio ja projekti | | Sovellukselle varattu muisti (*heap size*) (MB) | Käytetty muisti (*Allocated memory*) (MB) | Käytetty muisti / varattu muisti (%) | Objektien määrä |
|---|---|---|---|---|---|
| Emulaattori (2.2) | Canvas | 2,754 | 2,267 | 82,3 | 46 707 |
| | OpenGL ES | 2,754 | 2,290 | 83,1 | 47 304 |
| Emulaattori (4.0) | Canvas | 8,633 | 7,111 | 82,4 | 42 561 |
| | OpenGL ES | 7,633 | 5,713 | 74,8 | 42 644 |
| Samsung Galaxy S 2 (2.3.5) | Canvas | 5,254 | 2,711 | 51,6 | 51 860 |
| | OpenGL ES | 5,254 | 2,722 | 51,8 | 52 223 |
| Samsung Galaxy S Plus (2.3.5) | Canvas | 5,254 | 2,745 | 52,2 | 52 644 |
| | OpenGL ES | 5,254 | 2,760 | 52,5 | 53 061 |
| Asus Nexus 7 (4.4.2) | Canvas | 18,770 | 14,251 | 75,9 | 47 666 |
| | OpenGL ES | 9,137 | 7,636 | 83,6 | 47 967 |

TAULUKKO 3: Muistin käyttö.

Eroja ei juuri ole, paitsi Androidin uudemmilla versioilla (4.0 ja 4.4.2). Emulaattorissa (4.0) eroa on tosin vain yksi megatavu, eikä se täten ole merkittävä. Nexus 7:llä sen sijaan Canvas käyttää jopa kaksi kertaa niin paljon muistia kuin OpenGL ES.

Galaxy S 2:lla ja Galaxy S Plus:lla testatessa näkyy, että käytettyä muistia on vain reilut 50 % niin paljon kuin varattua muistia. Ilmiö kuitenkin näkyy sekä Canvasissa, että OpenGL ES:ssä, joten vertailussa näillä kahdella ei juuri ole eroa tässä asiassa.

Vaikkei asiaa näykään taulukossa, Canvasia käyttävässä sovelluksessa objektien määrä vaihtelee kaikilla laitteilla lähes jatkuvasti sovelluksen ollessa käynnissä. OpenGL ES:llä määrä vaihtelee ensimmäisten muutaman sekunnin ajan, mutta tasaantuu myöhemmin. Canvasillakaan määrän vaihtelu ei ole suurta, vain noin 1-5 objektia

kerrallaan. Mutta vaikka tämä ei vaikutakaan muistin käyttöön merkittävästi, muutosten tiheys kertoo automaattisen roskienkerääjän toimivan aktiivisemmin Canvasilla kuin OpenGL ES:llä, hidastaen ohjelman toimivuutta prosessorin laskentatehon puolella.

# 5 Yhteenveto

Vaikka OpenGL ES 1.0 onkin aloittelevalle ohjelmoijalle haastavampi kuin Canvas, menee se selkeästi edelle ohjelman toimivuudessa.

Omalla kohdallani suurimmat haasteet tulivat OpenGL ES:n muistinhallinnan kanssa, mutta asiaa projektin myötä opittuani kaikki alkoi hiljalleen vaikuttaa enemmän ja enemmän ymmärrettävältä. Luonnollisesti kaikkea ei opi yhden projektin aikana, mutta melko yksinkertaisia esiintyneet ongelmat lopulta olivat. Kaikissa niistä ei tosin löytynyt juurikaan logiikkaa, mikä vaikeutti ratkaisujen oivaltamista. Lopulta kuitenkin – kiitos pääasiassa internetin – ongelmat ratkesivat ja projekti saatiin onnelliseen päätökseen.

Canvasin kanssa ongelmia ei juurikaan esiintynyt, sillä sen käytöstä minulla oli hieman kokemusta jo aiemmin. Canvas on muutenkin huomattavasti yksinkertaisempi käyttää, sillä se vaatii vähemmän komentoja; kuvien hallinta aina animaation toiminnasta piirtämiseen vaatii huomattavasti vähemmän koodia. Canvasin heikkona puolena kuitenkin on selkeä ero ohjelman päivitys- ja latausnopeudessa, mikä onkin merkittävämpää itse ohjelman käyttäjän, eli asiakkaan, kannalta.

Yhdistäessämme herätyskelloprojektin ja tämän pelin, aiomme käyttää OpenGL ES -versiota pelistä nimenomaan sen toiminnan sujuvuuden ansiosta. Mikä on parasta käyttäjälle, on parasta myös itse ohjelmalle.

# Lähdeluettelo


**KIRJALÄHTEET**

Zechner, M. 2011. Beginning Android Games. New York: Apress Media LLC

**VERKKOLÄHTEET**

Against the Grain. 2014. OpenGL ES Android – Displaying Graphical Elements (Primitives). Käyty 10.1.2014. http://obviam.net/index.php/opengl-es-android-displaying-graphical-elements-primitives/

Android Developers. 2014a. Get Started with Publishing. Käyty 18.2.2014. http://developer.android.com/distribute/googleplay/publish/register.html

Android Developers. 2014b. Dashboards. Käyty 14.2.2014. http://developer.android.com/about/dashboards/index.html

Android Developers. 2014c. OpenGL ES. Käyty 14.2.2014. http://developer.android.com/guide/topics/graphics/opengl.html

Android Developers. 2014d. OpenGL ES. Käyty 10.1.2014. http://developer.android.com/guide/topics/graphics/opengl.html#choosing-version

Android Developers. 2014e. Processes and Threads. Käyty 10.1.2014. http://developer.android.com/guide/components/processes-and-threads.html#Threads

Android Developers. 2014f. Supporting Multiple Screens. Käyty 14.1.2014. http://developer.android.com/guide/practices/screens_support.html

iOS Developer Program. 2014. Distribute your App. Käyty 18.2.2014. https://developer.apple.com/programs/ios/distribute.html

Stack Overflow. 2014. glDrawElements crashing on devices, working well on emulator. Käyty 18.2.2014.
http://stackoverflow.com/questions/8570465/gldrawelements-crashing-on-devices-working-well-on-emulator/8606990#8606990

Wikipedia. 2014. Automaattinen roskienkeräys. Käyty 13.1.2014.
http://fi.wikipedia.org/wiki/Automaattinen_roskienker%C3%A4ys

Yle. 2014. Jolla yhteistyöhön venäläisen Yandexin kanssa. Käyty 3.1.2014.
http://yle.fi/uutiset/jolla_yhteistyohon_venalaisen_yandexin_kanssa/6932240