Mikko Kuhno

3D-grafiikan optimointi mobiilialustalle Unity-ympäristössä

Tietotekniikan pro gradu -tutkielma

23. marraskuuta 2016

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Mikko Kuhno

Yhteystiedot: +358409105675, mikko.kuhno@gmail.com

Ohjaajat: Jarno Kansanaho ja Tuomo Rossi

Työn nimi: 3D-grafiikan optimointi mobiilialustalle Unity-ympäristössä

Title in English: Optimizing 3D-graphics to mobile using Unity

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Pelit ja 3D-grafiikka

Sivumäärä: 75+3

Tiivistelmä: Mobiilimarkkinoilta löytyy hyvin laaja kirjo erilaisia mobiilipelejä. Mobiililaitteet ovat laajimmalle levinnyt tietokonemuoto. Viimevuosina mobiililaitteiden graafiset ominaisuudet ovat nousseet sellaiselle tasolle, että niillä voidaan renderöidä upeita 3D-ympäristöjä reaaliajassa. Silti mobiililaitteet vaativat optimointia sulavaan peligrafiikan laskemiseen.

Tämä pro gradu tutkielma paneutuu 3D-mobiiligrafiikan optimointiin keskittyen Unity-pelimoottoriin. Teoriaosuudessa käydään läpi 3D-grafiikan luomisen peruskäytänteitä siirtyen Unityn käyttämään OpenGL ES liukuhihnaan ja sen optimointimahdollisuuksiin. Käytännön osuudessa testataan kolmioiden, valaistuksen, sekä varjostimien vaikutusta mobiililaitteiden ruudunpäivitysnopeuksiin. Optimointimenetelmät implementoidaan Endless Tea Studiosin Gravitoid mobiilipeliin.

Avainsanat: Unity, Mobiili, 3D-grafiikka, Optimointi, Liukuhihna, OpenGL, OpenGL ES, Gravitoid

Abstract: Mobile markets are swarming with different kinds of games. Mobile devices are the most widely spread personal computer type in the world. In recent years the graphical processing unit in these devices has come to such level that you can render astonishing 3D-environments on these handheld machines. All though powerful and small, they are not as well suited for realtime rendering as normal desktop computers. This is why mobile game

development requires optimization to work fluently in handheld devices.

This thesis dives into the world of mobile graphic optimizing on certain development applications. The theoretical chapter will focus on explaining the rendering pipeline on Unity and OpenGL ES and the different optimization methods they offer. Practical part will go through list of effective ways to optimize 3D-scenes on a mobile device. Practical test environment include vertex optimization, lighting optimization and shader optimization. Basis for optimization methods is a mobile game named Gravitoid. Gravitoid is 2.5D physics platformer game that utilizes multiple 3D models and lighting.

Keywords: Unity, Mobile platforms, 3D graphics, Optimizing, Pipeline, OpenGL, OpenGL ES, Gravitoid

Termiluettelo

API	Application Programming Interface. Ohjelmointirajapinta, jonka mukaan eri ohjelmat voivat keskustella keskenään.
Blender	3D-mallinnus ja editoriohjelma.
Deferred Rendering	Jaksotettu renderöinti.
Dekoodaus	Datan muuntaminen takaisin alkuperäiseen muotoonsa.
Enkoodaus	Lähtödatan muuntaminen tiiviimpään muotoon.
Forward rendering	Suora renderöinti.
FPS	Frames Per Second. Ruudunpäivitysnopeus.
Gravitoid	Endless Tea Studiosin esikoispeli.
HLSL	High Level Shading Language. Korkean tason varjostinkieli.
Inertia-avaruus	Inertial Space. Kuvastaa fysikkamoottorin käyttämää objektiavaruuden ja maailman avaruuden välistä koordinaatistoa.
Kameran avaruus	Camera Space. Kuvastaa kameran paikallista koordinaatistoa.
Kolmiosarja	Kolmiot, jotka lasketaan grafiikkakortissa sarjamaisena ryhmänä.
Kolmioverkko	3D-maailmassa sijaitseva kolmioista muodostettu verkko.
Kolmioviuhka	Kolmiot, jotka lasketaan grafiikkakortissa viuhkamaisena ryhmänä.
Komentojonokutsu	Batch Call. Materiaaliin sidottu kappaleen piirtokutsu.
Kulmapiste	Monikulmion kulman määrittävä piste.
Kulmapisteiden jakaminen	Vertex Split. Yhden kulmapisteen jakaminen kahdeksi tai useammaksi kulmapisteeksi.
Kulmapisteiden yhdistäminen	Vertex Welding. Kahden tai useamman kulmapisteen yhdistäminen yhdeksi kulmapisteeksi.
Kyhmytyskartta	Bumpmap Bittikartta, jonka avulla esitetään pinnan epämuodostumia.
Liukuhihna	Pipeline. Prosessointielementtien sarja, jossa edeltävän elementin tuotos on seuraavan elementin syöte.
Läpivientikutsu	Set Pass Call. Renderöinnissä tapahtuva alatason kutsu, joka

	määrittää mitä varjostinläpivientiä käytetään.
Maailman avaruus	World Space. Kuvastaa pelimaailman koko koordinaatistoa.
Mip kartta	Mipmap. Esilaskettu progressiivisesti pienempiresoluutioinen versio tekstuuritiedostosta.
Normaalikartta	Normal map. Kyhmytyskarttaa monipuolisempi pinnan epämuodostumisen esittämiseen luotu bittikartta.
NPOT	Non Power of Two. Tekstuuritiedostoissa esiintyvä kuvan koko, joka ei ole kahden potensseja.
NURBS	Non-uniform rational basis spline. Matemaattisesti laskettava 3D-pinta, jonka tarkkuus pysyy vakiona kaavamaisen esityksen ansiosta.
Näkymäfrustrum	Kameranäkymän alue, joka rajaa ruudulle piirtyvät kappaleet.
Näkymäkartta	Scenegraph. Tietokantavisualisointi graafisten kappaleiden keskinäisistä suhteista.
Objektiavaruus	Object Space. Kuvastaa objektin paikallista koordinaatistoa.
OpenGL ES	Open Graphics Library Embedded System. Ohjelmointirajapinta, joka erikoistuu tietokonegrafiikan tuottamiseen.
Ortograafinen kamera	Kameraprojektio ilman syvyysvääristymää.
Parallaksikartta	Parallax map. Pinnan korkeuserojen illuusion luomiseen tehty bittikartta.
Perspektiivikamera	Kameraprojektio syvyysvääristymällä.
Pistepilvi	Suuresta kulmapistemäärästä koostuva "pilvi."Pistepilvistä muodostetaan kolmioverkkoja.
Polygoni	Kolmesta tai useammasta kulmapisteestä muodostuva monikulmio.
Profiler	Unityn profilointityökalu, jolla mitataan ajettavan pelin suorituskykyä halutussa laitteistossa.
Proseduraalinen	Tietokoneella generoitu.
Renderöinti	Prosessi, missä lähtödatasta muodostetaan 2D-kuva.
Reunan poisto	Edge Collapse. Reunajanan poistaminen yhdistämällä janan päätepisteet yhdeksi pisteeksi.

Siirtymäkartta	Displacement map. Bittikartta 3D-objektin pinnanmuotojen muokkaamiseen.
Skene	Scene. Unityn kehitysnäkymä, joka sisältää yhden pelikentän ja siihen liittyvät tiedot.
SoC	System on a Chip. Yhdelle piirilevylle integroitu tietokoneen rakenne.
Tarkkuustaso	LOD eli Level of Detail. Kappaleen monimuotoisuuden määrittävä säännöstö.
Tason erotus	Detach Face. Samojen kärkipisteiden jakavien kolmioiden muuntaminen käyttämään erillisiä kärkipisteitä.
Tekseli	Texel. Kuvatekstuurin pikseli.
Tekstuuriatlas	Suuri tekstuuritiedosto, joka sisältää usean eri objektin tekstuurit.
Tekstuurikartta	Texture map. Bittikartta, joka sisältää 3D-objektin pinnalle heijastettavan kuvan.
UI	Käyttöliittymä. Sisältää menuelementit sekä painikkeet.
Unity	Monialustainen pelimoottori videopelien tekemiseen.
Valaistuskartta	Lightmap. Bittikartta, joka sisältää 3D-objektin pinnalle heijastettavat ennalta lasketut valaistukset.
Vertex Lit	Verteksivalaistus.

Kuviot

Taulukot

Sisältö

1 Johdanto

3D-grafiikkaa on esiintynyt eri medioissa jo kohta 40 vuoden ajan. Ensimmäiset kolmiulotteiset esitykset ovat olleet karkeita esityksiä, joiden tulevaisuutta on kritisoitiin ja epäiltiin. Ajan saatossa 3D-grafiikasta on muodostunut kuitenkin erottamaton osa viihdekulttuuria niin elokuva-, mainos- kuin pelituotannoissakin. Jokainen 3D-tuotannon osa-alue asettaa visualisoinnille erilaiset vaatimukset ja haasteet. Elokuvapuolella yhtä ruudulla näkyvää kuvaa voidaan renderöidä jopa päiviä tehokkailla 3D-laskemiseen kustomoiduilla tietokoneilla. Peliteollisuudessa haasteeksi muodostuu peligrafiikan reaaliaikainen renderöiminen. Kuva pitää saada ruudulle millisekunneissa tuntien sijaan. Tällöin elokuvamaisesta laadusta on tingittävä. Reaaliaikainen renderöinti vaatii muusta renderöimisestä poikkeavia laskuoperaatioita pitääkseen ruudunpäivitysnopeuden sulavana. Pelimarkkinoiden kukoistus alkoi 90-luvulla ja on tähän päivään mennessä muodostunut suurimmaksi viihdeteollisuuden muodoksi. Mobiilipelien suosio nousi Applen vuonna 2007 julkaiseman iPhonen myötä ja kattaa tällä hetkellä noin kolmasosan koko pelimarkkinoista. **[01]**; **[02]**

Mobiilimarkkinoilla on lukuisia erilaisia pelejä, joista yhä useampi käyttää visuaalisesti monipuolisia 3D-objekteja. Kuitenkin mobiilialusta on PC- tai konsoligrafiikkaan verrattuna hyvin rajoittunut kehitysympäristö. Pienen kokonsa ja akkukestonsa vuoksi mobiililaitteissa joudutaan tekemään suuriakin kompromisseja visuaalisen ilmeen kanssa. Tehorajoitteet tulee ottaa huomioon jo kehitysvaiheessa, sillä runsaat ja monipuoliset efektit ovat laitteistolle hyvin raskaita. PC- ja konsolilaitteissa on yleensä laskutehoa niin paljon, ettei optimointi ole kriittistä kuin suurissa AAA-peleissä. Mobiilipuolella optimointia on hyvä tehdä jo yksinkertaisemmissakin peleissä rajoitettujen laitteistotehojen vuoksi.

Mobiilipelien kehitys lähti puhelinten mukana tulleista pienistä peleistä, ja ala on kasvanut lyhyessä ajassa miljardibisnekseksi. Nykypäivänä niin Androidin Google Playn kun Applen App Storen myyntilistojen kärkinimien taustalla on miljoonabudjetteja omaavia pelitaloja, kuten esimerkiksi Rovio, Supercell, King, jne. Suurin osa markkinoille tulevista peleistä on kuitenkin pienempiä, muutaman ihmisen yrityksiä. Esimerkiksi Koreassa suuri osa tarjonnasta tulee pieniltä,4 muutaman hengen pelifirmoilta. **[03]** Siksi pelin kehitysvaiheessa harva tiimi lähtee tekemään kokonaan omaa pelimoottoria pelilleen. Yleensä käyttöön otetaan

jonkin toisen firman tarjoama pelimoottori ja rakennetaan peli sen päälle. Pelimoottorimarkkinoilla Unity on ollut pitkään hallitsevassa asemassa sen monipuolisen laitetuen ja käyttäjäystävällisen käyttöliittymän ansiosta. **[04]** Erillinen pelimoottori on pelin optimoinnin kannalta ongelmallinen ratkaisu, sillä useasti silloin ei kehittäjillä ole pääsyä pelimoottorin sisäiseen mekaniikkaan.

Tämä pro gradu tutkielma paneutuu Unityllä tehdyn 3D-pelin optimointiin mobiilialustalle soveltuvaksi. Työssä käydään läpi liukuhihna Unity-pelimoottorista Andoid-laitteiden käyttämään OpenGL ES-grafiikkarajapintaan, sekä sen tarjoamat optimointimahdollisuudet. Teoriaosuudessa käydään läpi 3D-grafiikan muodostaminen mobiililaitteen näytölle OpenGL ES liukuhihnaa myöten. Ensiksi käydään läpi 3D-objektin renderöiminen, tarkkuuden taso (Level Of Detail), sekä pintamateriaali ja keskitytään mobiililaitteiden rajoitteisiin pelialustana. Tämän jälkeen esitellään liukuhihna Unity-pelimoottorista OpenGL ES grafiikkarajapintaan ja listataan eri tavat optimoida tätä liukuhihnaa.

Luvussa kolme käydään läpi esiteltyjen optimointimenetelmien työmäärää ja tehokkuutta. Listatuista metodeista valitaan muutama ja tehdään niistä koetilanteet mobiilialustalle. Testattavaksi tulee kolmioiden määrä, valaistuspolun vaikutus, ja varjostinohjelman vaikutus ruudunpäivitysnopeuteen. Testauskappaleessa testataan näiden optimointimenetelmien toimivuutta, ja vaikutusta visuaaliseen ulkonäköön. Luvussa neljä käydään läpi saadut hyödyt ja haitat sekä tehdään niiden pohjalta johtopäätökset käytetyistä metodeista.

1.1 Työkalut

Pelien optimointia on mahdollista tehdä usealla eri tyylillä. Eri optimointimenetelmät vaikuttavat eri kohtiin pelin toiminnassa. Jotkut optimointimenetelmät vähentävät kolmioiden määrää näkymässä vähentäen grafiikan prosessointiyksikön kuormitusta. Toiset voivat hoitaa pelimekaniikallisia toimintoja paremmin vähentäen CPU:n rasitusta. Tässä työssä keskitytään Unityn tarjoamiin optimointimahdollisuuksiin ja käydään läpi, miten 3D-malleja voidaan optimoida. Optimoinnin testaus tehdään Unityn Profiler-työkalun avulla. Osa optimoinnista tehdään Blender työkalulla. Pääasiallisina työvälineinä toimivat siis Unity ja Blender. Näiden kahden työkalun avulla saadaan suoritettua lähes kaikki tarvittava optimointi. Vie-

lä muutama vuosi sitten mobiililaitteiden suorituskyvyn testaamiseen ei juurikaan ollut erinäisiä ohjelmia. **[05]** Myöhemmin testiohjelmia on ilmestynyt käytettäväksi ja esimerkiksi pelimoottorit tarjoavat testaamiseen omia sisäänrakennettuja testiohjelmia. Unityn Profiler valittiin siksi, että se tarjoaa kolmansien osapuolten työkaluja runsaammin dataa käsiteltäväksi.

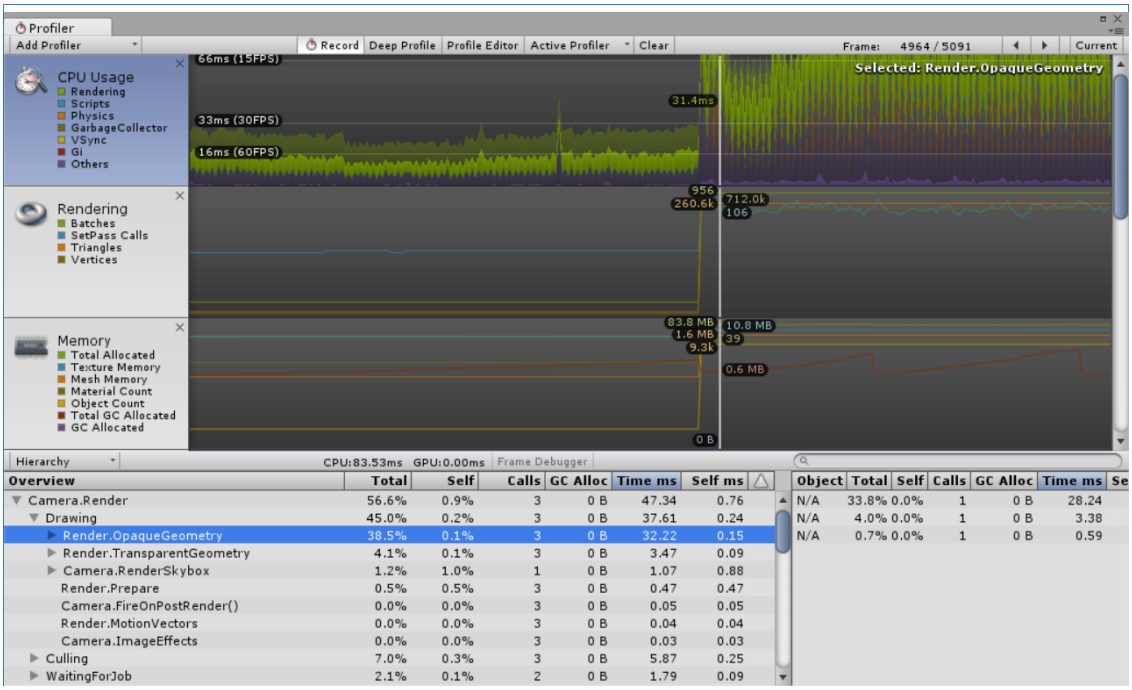
Unity on Unity Technologiesin kehittämä alustariippumaton pelimoottori. Se käyttää eri grafiikkakirjastoja päätealustasta riippuen. Mobiililaitteilla rajapintana toimii OpenGL ES. Ensimmäinen versio Unitystä julkaistiin 8. kesäkuuta 2005. Sittemmin Unity on saanut viisi suurta päivitystä, ja uusin versionumero on 5.4.0.

Käytännön vaiheessa optimointityön tuloksia tarkastellaan Unity Profilerin avulla. Profiler on Unityyn sisäänrakennettu statistiikan keräystyökalu. Profilerin avulla on mahdollista tarkastella pelin suorituskykyä 300:n yksittäisen ruudun aikaikkunan verran. Profilointinäkymässä voi tarkastella eri osa-alueiden vaikutusta suorituskykyyn. Osa-alueita ovat seuraavat:

- CPU:n profilointi
- Renderöinnin profilointi
- Muistin profilointi
- Audion profilointi
- Fysiikan profilointi
- GPU:n profilointi

Tehokas optimointi tulee aloittaa eri pullonkaulojen kartoituksesta ja niiden optimoimisesta. **[06]** Optimoinnin jälkeen peliskenaarion pitäisi toimia hieman sulavammin jonkin toisen asian aiheuttaessa pullonkaulan performanssiin. Tässä työssä varsinainen pelimekaniikan optimointi jätetään taustalle ja keskitytään graafisten ominaisuuksien optimointiin. Lähempään tarkasteluun tulee CPU:n, muistin sekä renderöinnnin profilointi. GPU:n profilointia ei voida tehdä, sillä mobiilitestilaitteissa ei ole tarvittavia grafiikkakortin ominaisuuksia datan saamiseksi. Testauskeneissä grafiikan tuottaminen kuormittaa CPU:n toimintaa keskimäärin yli 60 %, aiheuttaen näin suurimman tarpeen optimoinnille.

3D-mallien optimointi on kohtuullisen rajoitettua Unityssä. Siksi niiden muokkaamiseen käytetään Blender säätiön ylläpitämää, avoimeen lähdekoodiin perustuvaa mallinnusohjel-



Kuvio 1. Unity Profilerin näkymää. Keskivaiheilla oleva hyppy johtuu skenen vaihdoksesta

maa, Blenderiä. Blender on etenkin indie-kehittäjien piirissä suosittu mallinnusohjelma monipuolisten ominaisuuksiensa ja ilmaisen lisenssinsä vuoksi. Ensimmäinen versio Blenderistä julkaistiin jo vuonna 2002. Blender oli aluksi maksullinen, mutta sen luoman yhtiön mennessä konkurssiin sen lähdekoodin lisenssi muutettiin avoimeksi. Blenderin kehitys jatkui pääosin vapaaehtoisvoimin nostaen sen suosiota vuosi vuodelta. Nykyään Blenderiä ylläpitää suuren vapaaehtoisyhteisön lisäksi kaksi täysipäiväistä ja kaksi osa-aikaista työntekijää. **[07]** Blender sisältää 3D-mallinnusominaisuuksiensa lisäksi lukuisia muita ominaisuuksia, kuten:

- Useiden geometristen primitiivien tuki
- Blender renderer ja Cycles rendered
- Blender pelimoottori
- Kolmansien osapuolten renderöintimoottorien tuki
- Simulaatiotyökaluja (nesteet, pehmeät materiaalit)
- Avainkehystetty animaatio
- Partikkelijärjestelmä, jossa on tuki partikkelipohjaiselle hiusten generoinnille
- Python-skriptaustyökalu pelilogiikan, automaation ja eri tiedostojen tuontiin/vientiin

- Video- ja audioeditointityökalut
- Tekstuurien piirtotyökalut, proseduraaliset tekstuurit
- Kamera- ja objektiseuranta

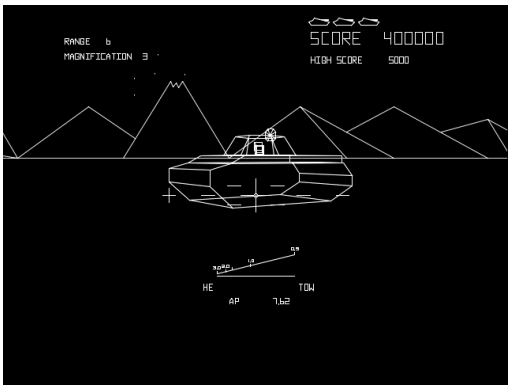
2 3D-peligrafiikka mobiililaitteissa

Kolmiulotteista tietokonegeneroitua grafiikkaa on käytetty nyt jo vuosikymmeniä kuvastamaan erilaisia tilanteita. Nykyään 3D-materiaaliin törmää arkipäiväisessä elämässä päivittäin. Peleissä 3D-grafiikan laskemiseen kohdistuu muusta mediasta poikkeava haaste: 3D-kuva pitää saada piirtymään ruudulle lähes reaaliajassa. Elokuva- ja mainosteollisuudessa yhden kuvan renderöimiseen voidaan käyttää tunteja tai jopa päiviä. Peleissä tai muissa reaaliaikaisissa simulaatioissa kuva tulisi päivittää yli kolmekymmentä kertaa sekuntia kohden, jolloin sulava illuusio liikkeestä muodostuu. Peligrafiikasta vielä haasteellisemman tekee mobiililaitteisiin suunnattu grafiikka, sillä nopean ruudunpäivityksen lisäksi kohdelaitteen laskennalliset tehot eivät ole varsinaiseen pelaamiseen erikoistuneiden konsolien tai kotitietokoneiden luokkaa. Laskuoperaatiot pitää saada laskettua kämmenen kokoiselle laitteelle pienellä virrankulutuksella. Tämä kappale käy läpi ensin hieman 3D-peligrafiikan historiaa, josta siirrytään käsittelemään yleisimmät peligrafiikan osa-alueet, kuten pelinäkymä, pintamateriaalit ja tarkkuustasot (Level Of Detail). Lopuksi esitellään mobiililaite pelialustana ja käydään läpi Unity - OpenGL ES:n liukuhihna.

2.1 3D-peligrafiikan historia

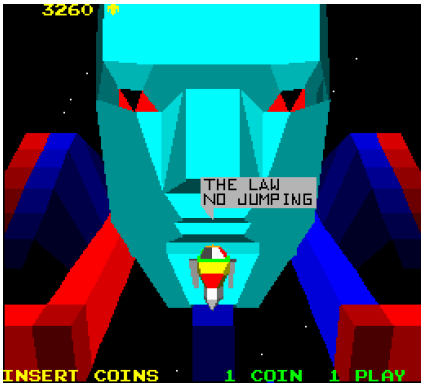
Tietokonepelien historia ulottuu jopa 1950-luvun loppupuolelle, jolloin kiistellysti ensimmäinen tietokonepeli Tennis for Two kehitettiin. **[08]** On vaikea sanoa, mikä peli oli varsinainen ensimmäinen 3D-peli, sillä sen määrittäminen, mikä on oikeasti 3D-esitystä, riippuu tarkastelijan näkökulmasta. Ennen "todellisen"3D-kuvannan esiintuloa pelimarkkinoilla vallitsi pitkään niin sanottu 2.5-D ilmiö. Kyseessä on keino huijata ihmissilmää kolmiulotteisesta vaikutelmasta rasterikuvien skaalauksella, sekä parallaksitasojen avulla. Vuonna 1980 julkaistu Battlezone piirsi pelikuvan vektoreina ruudulle ja näin loi kolmiulotteisen ympäristön, missä kaikki pelinäkymän kappaleet esitettiin rautalankaversioina. Pelimaailmaa pyöritti siis oikea 3D-pelimoottori, ja esimerkiksi Mark J. P. Wolf kategorisoi Battlezonen ensimmäiseksi oikeaa 3D-ympäristöä käyttäväksi kolikkopeliksi. **[08]** Värimaailma oli mustavalkoinen, mikä muutettiin vihreäksi ja punaiseksi ruutuun liimattavalla värikalvoilla. Myöhemmin Battlezone julkaistiin muun muassa Atari 2600:lle. Battlezonen julkaisun jälkeen

joulukuussa 1980 Yhdysvaltain armeija lähestyi Ataria idealla, jossa Battlezone muutettaisiin simulaatioksi samoihin aikoihin muodostetulle rynnäkköpanssarivaunuosastolle. Peliin tehtiin lukuisia muutoksia, kuten uudet kontrollit, uusia ajoneuvoja, sekä aseita. **[09]**; **[10]**



Kuvio 2. Pelikuvaa Battlezone pelistä. (Kuva: Wikipedia **[11]**)

Atarin vuonna 1983 julkaisema I, Robot oli ensimmäinen peli, joka hyödynsi interaktiivista ja varjostettua 3D-grafiikkaa. Peli oli graafisilta ominaisuuksiltaan vuosia aikaansa edellä, eikä samantasoista peligrafiikkaa nähty seuraavan kerran ennen kuin vasta vuosia myöhemmin pelikonsolien yleistymisen myötä. Edistyksellisestä grafiikastaan huolimatta I, Robot ei ollut kaupallinen menestys, ja kyseisiä kolikkopelikoneita valmistettiin vain 75 -1000 kappaletta. **[12]**



Kuvio 3. Pelikuvaa I, Robot-pelistä (Kuva: Daniel Rehn **[13]**)

Battlezone ja I, Robot olivat molemmat kolikkopelikoneita, ja myöhemmin 1980-luvulla elettiin kolikkopelihallien kulta-aikaa. 1990-luvun loppupuolella pelihallipelien kultakausi kääntyi kuitenkin laskuun konsoleiden yleistymisen myötä. **[14]** Ensimmäinen kaupallinen

pelikonsoli oli vuonna 1972 julkaistu Magnavox Odyssey. Se sisälsi analogisen kontrollerin, vaihdettavat pelikasetit sekä valopistoolin. **[14]** Odysseytä seurasivat lukuisat muut konsolit. Osa markkinoille tulleista konsoleista oli floppeja. Toiset konsolit, kuten Atari 2600, Nintendo Entertainment System ja Sega Genesis/Mega Drive ovat kulttimaineessa vielä tänäkin päivänä. Pelikonsolipuolella 3D-grafiikka ilmeni vasta konsolien neljännen sukupolven myötä, kun pelikonsoleihin saatiin tarpeeksi laskentatehoa 3D-mallien sulavaan laskentaan.

Pelikonsolien myötä 3D-pelit alkoivat yleistyä kiihtyvällä tahdilla. Konsolien neljännen generaation ja kotitietokoneiden laskentatehon kasvaessa markkinoille tuli ensiksi pelitaloista tunnettujen pelien konsoliversioita ja myöhemmin eksklusiivisesti tietyille laitteille tehtyjä pelejä. Huomionarvoisia 3D-pelejä on muun muassa:

- Zarch (1987)
- Starglider (1988)
- Hard Drivin' (1989)
- Alpha Waves (1990)
- Catacomb 3D (1991)
- Ultima Underworld: The Stygian Abyss (1992)

Yleiseksi piirtotyyliksi polygongrafiikka tuli 1990- luvun keskivaiheilla. Kolmiulotteinen sisältö alkoi yleistyä peligrafiikassa konsoleiden, kuten Playstationin, Sega Saturnin, sekä Nintendo Ultra 64:n myötä. Tuolloin markkinoilla kukoistivat isometriset 2.5D-pelit, kuten Zaxxon (1982), SimCity (1989) jne. Useita 3D-kiihdytettyjä osia hyödyntäviä pelejä sanottiin myös 2.5D-peleiksi niiden käyttämien kaksiulotteisten kenttäobjektien tai esirenderöityjen taustojen vuoksi. Tällaisia 2.5D-pelejä ovat muun muassa FPS-pelit Wolfenstein 3-D ja Doom, sekä esirenderöidyt 3D-pelit, kuten Myst ja Gadget. 2.5D pelit antoivat kuvan kolmiulotteisuudesta ilman varsinaista 3D-kiihdytintä. Näin säästettiin laskentatehoja varsinaiseen peliin pitäen visuaalinen ulkonäkö kuitenkin modernina. PC-puolella kolmiulotteisen pelimaailman nosti julkisuuteen ID softwaren kehittämä Wolfenstein 3D, vaikka sen kolmiulotteinen ympäristö oli hyvinkin rajattua. Kenttien objektit ja viholliset olivat kameraan suunnattuja kaksiulotteisia spritejä, ja pelin aseet pelimaailman päälle esipiirrettyjä sprite-animaatioita. Pelimaailma salli ainoastaan 90 asteen kulmia eikä yhtään korkeuseroja. Lukuisat näistä ongelmista korjattiin ID Softwaren seuraavaan peliin, Doomiin. Doomissa

pelimaailma sisälsi monimuotoisia kulmia, korkeuseroja, sekä esilaskettuja valaistuksia. Julkaisunsa jälkeen Doom ja sen jatko-osat muokkasivat pelikulttuuria merkittävästi. Doom oli ilmestymisensä jälkeen suositumpi ohjelmisto kuin esimerkiksi käyttöjärjestelmä Windows - 95. Windowsin kehittäjä Bill Gates suunnittelikin ostavansa ID softwaren ja teki cameoesiintymisen Doomissa nostaakseen Windowsin myyntilukuja. **[15]** Ensimmäinen varsinaisesti natiivi 3D-peli oli ID softwaren kehittämä Quake. Muista peleistä tuttu objektien esittäminen kaksiulotteisina spriteinä korvattiin natiiveilla 3D-objekteilla. Quake käytti myös ensimmäisenä pelinä erillistä grafiikkakiihdytintä 3D-kuvan luonnissa.

Vuosien saatossa 3D-pelien määrä on kasvanut räjähdysmäisesti, ja nyt on mahdollista tehdä oma kolmiulotteista materiaalia sisältävä peli hyvinkin lyhyessä ajassa. **[16]** Kolmiulotteista materiaalia on mahdollista saada joko ilmaiseksi tai rahaa vastaan. Pelin tekemiseen erikseen tarkoitetuilla ohjelmilla, kuten Unreal Engine tai Unity on mahdollista kasata peliprototyyppi ilman riviäkään omaa koodia. Varjopuolena pelien nopealle kehitykselle on tullut erilaisten valmiskomponenttien osista kasatut peliraakileet, joita myydään valmiina peleinä. Pelien pelaaminen siirtyi mobiilille jo Nokian aikoihin matopelin muodossa. Sittemmin mobiilipelaaminen on kasvanut älypuhelinten läpimurron myötä yhdeksi suurimmista pelimarkkinoista. **[17]** 3D-grafiikka yleistyi suhteellisen nopeasti mobiililaitteissa laskentatehojen kasvamisen myötä. Nyt käsikokoinen laite pystyy pyörittämään näyttävän näköisiä 3D-pelimaailmoja kohtuullisella virrankulutuksella. Mobiilialustoille luotiin aivan oma grafiikkarajapintansa, OpenGL ES. Tämän grafiikkakirjaston päällimmäinen tarkoitus on tarjota mahdollisimman paljon samanlaisia grafiikkaoperaatioita kuin OpenGL API (Application Programming Interface). OpenGL ja OpenGL ES API:sta kerrotaan lisää luvussa 2.5.

Strategy ennustaa pelimarkkinoiden kokonaistulojen kasvavan tasaisesti vuosikymmenen loppuun saakka. Tämä tarkoittaisi suurinpiirtein 93,18 miljardin dollarin tuloja vuonna 2019 **[17]** Näistä kokonaismarkkinoista noin kolmasosan (34 mrd) ennustetaan olevan mobiilipelejä. Mobiilimarkkinoiden ehdottomin vahvuus on mobiililaitteiden suuri määrä. Kehittyneiden grafiikkakirjastojen ja kokoonsa nähden tehokkaiden grafiikkaprosessorien myötä myös 3D-pelit voivat näyttää laadukkailta myös mobiilialustoilla.

2.2 Pelinäkymä

Pelinäkymä käsittää kaiken sen, mitä pelin aikana ruudulla näkyy. Se sisältää yleensä jonkinlaisia käyttöliittymäkomponentteja sekä itse pelinäkymän, missä peliä pelataan. Kolmiulotteisessa pelimaailmassa pelinäkymää voidaan käsitellä monella eri tasolla. Esimerkiksi pelinäkymässä voi olla yksi tai useampi 3D-kappale. Tämä peliobjekti voi sisältää useampia lapsiobjekteja. Esimerkiksi pelinäkymässä oleva auto on yksi oma peliobjektinsa. Se sisältää kuitenkin lapsiobjekteja, kuten renkaat, ratin, moottorin jne. Tällaista 3D-objektien jaottelua kutsutaan näkymäkartaksi (scene graph), ja sen avulla voidaan esittää objekteja pelinäkymässä loogisesti. **[18]**

Peliobjektit koostuvat kolmioverkoista (mesh). Kolmioverkot vuorostaan koostuvat kolmioista (triangle), ja kolmiot koostuvat kulmapisteistä (vertex) sekä niiden välille muodostuvista viivoista (line). Jukka Räbinä kuvaa väitöskirjassaan 3D-datan muodostuvan soluista, jotka muodostavat lopuksi solukomplekseja. **[19]** Nollasolu on yksi ainut piste avaruudessa. Yksi solu on taas kahden pisteen välille differentioituva jana. Kaksisolu muodostuu, kun kahden yhdensuuntaisen janan välille muodostetaan loputon määrä janan kopioita. Janoista muodostuu näin taso. Samaa logiikkaa jatkamalla kolmisolu on kahden samansuuntaisen tason väliin muodostuva volyymi. Solujen avulla saadaan neliömäisiä rakennelmia, mutta rakennetta voi vielä yksinkertaistaa hieman luomalla simpleksejä. Yksinkertaisin simpleksirakenne on 0-simpleksi, joka on vain piste avaruudessa. 1-simpleksissä lisätään uusi piste ja vedetään niiden välille jana. 2-simpleksissä lisätään janan sijasta kolmas piste avaruuteen, joka yhdistyy kaikkiin aiemmin luotuihin pisteisiin. Saadaan aikaan kolmio. Neljäs lisättävä piste muodostaa neljästä kolmiosta koostuvan nelitahokkaan. Pelimoottoreissa kappaleet esitetään yleensä 2-simpleksisessä muodossa, eli kolmiossa. OpenGL määrittää pisteet ja janat 2D/3D-entiteeteiksi. Pisteiden ja janojen kärkipisteet käsitellään todellisina pisteinä 3D-avaruudessa, mutta kun ne projisoidaan kameralle niillä on ennalta määritetty pisteen koko ja viivan paksuus pikseleinä. Kun kuvaa lähennetään, venyy pisteiden väli, mutta paksuus pysyy ennalta määritettynä tehden niistä osaksi 2D-entiteettejä. **[18]**

2.2.1 3D-lähtödata ja sen luominen

Jokainen ruudulla näkyvä 3D-objekti vaatii lähtödatan, josta kyseinen objekti voidaan laskea ruudulle. Data on käytännössä vain tiedosto täynnä kappaleen kulmapisteiden koordinaatteja, tekstuuritietoja jne. Eri ohjelmat voivat käyttää eri datatyyppejä 3D-objektien laskemiseen, mutta yleisiä standardeja on muun muassa Wavefront (.obj), Collada (.dae), 3D Studio (.3ds), FBX (.fbx) jne. Unity hyväksyy oletusarvoisesti kaikkia edellä mainittuja tiedostomuotoja. Unity-pelimoottoriin voi tuoda myös konversion kautta Max-, Maya-, Blender-, Cinema4D-, Lightwave- ja Cheetah3D-tiedostoja. **[20]** Konversion kautta tuomisen etuna on nopea mallien iterointi pelimaailmaan, sillä uutta iteraatiota ei tarvitse manuaalisesti tuoda Unityyn, vaan editointiohjelmassa tallentaminen päivittää Unityn 3D-mallin automaattisesti. Optimointinäkökulmasta konversion kautta tuodut 3D-mallit ovat käännettyjä huonompia niiden sisältäen paljon turhaa tietoa, mikä hidastaa päivittämistä. Unityn lisäosakaupasta (Asset Store) on mahdollista hankkia myös muutamia kolmannen osapuolen konversiolisäosia erikoisempien tiedostomuotojen tuomiseksi Unityyn. Alla on esimerkki Wavefrontin tavasta tallentaa yksinkertaisen kuution data.

```
# Blender v2.76 (sub 0) OBJ File: ''
# www.blender.org
mtllib kuutio.mtl
o CUBEEEE_Cube
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -0.999999
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vn 0.000000 -1.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 1.000000 0.000000 0.000000
vn -0.000000 -0.000000 1.000000
vn -1.000000 -0.000000 -0.000000
vn 0.000000 0.000000 -1.000000
```

```
usemtl CubeMaterial
s off
f 1//1 2//1 3//1 4//1
f 5//2 8//2 7//2 6//2
f 1//3 5//3 6//3 2//3
f 2//4 6//4 7//4 3//4
f 3//5 7//5 8//5 4//5
f 5//6 1//6 4//6 8//6
```

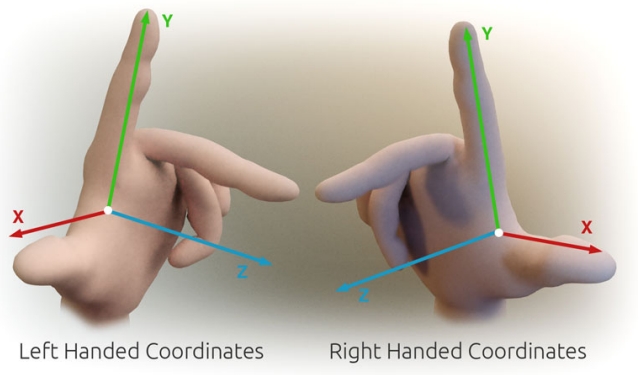
3D-dataa on mahdollista luoda usealla eri tavalla. **[21]** Eri käyttökohteisiin käy erilaiset luontitavat. 3D-dataa voidaan muodostaa muun muassa seuraavilla eri menetelmillä:

- 3D datan kirjoittaminen suoraan tekstimuotoon.
- Jonkin muun datan muokkaaminen 3D:ksi.
- Ohjelmallinen 3D-datan kirjoittaminen (proseduraalinen mallinnus.)
- Mallinnusohjelmissa muodostettava 3D-data.
- Kuvista muodostettu 3D-data.
- 3D-skannereista tuotava data.

Eri menetelmiä on mahdollista yhdistää 3D-mallien luomisessa. Esimerkiksi J. C. Carr ym. käyvät vuonna 2001 julkaistussa artikkelissa läpi pistepilvidatan muuntamisen kolmioverkoksi Radial Basis Functions menetelmällä. **[22]** Hankitusta pistepilvidatasta muodostetaan kolmioverkko. Kolmioverkkoja joudutaan yleensä optimoimaan, mikäli niitä halutaan pyörittää reaaliaikaisesti etenkin mobiililaitteissa. Optimoinnissa pistepilvestä muodostettua kolmioverkkoa uudelleenlasketaan optimoidusti tai sen pohjalta tehdään uusi optimoitu 3D-malli. Esimerkiksi rakennusten seinien siirtäminen pelimoottoriin saattaa vaatia perinteistä tekniikkaa, jolla polygonverkon päälle luodaan rautalankamalli. Luotuun rautalankamalliin muotoillaan tarkemmat muodot, kuten oviaukot ja ikkuna-aukot. **[23]** 3D-Mallista tehdään yleensä tarkka versio sekä yksinkertaistettu versio. Tarkasta mallista voidaan tehdä normaalimappaus (normal mapping), joka liitetään myöhemmin optimoidun mallin päälle. Skannauksen yhteydessä otetuista kuvista tehdään tekstuuritiedot. 3D-objekti pyritään optimoimaan mahdollisimman hyvin, ettei kolmioiden määrä nouse peliruudulla liikaa.

2.2.2 Pelinäkymä

Pelit voidaan jakaa kaksiulotteisiin ja kolmiulotteisiin peleihin pelinäkymästä riippuen. Kaksiulotteisten pelien peliavaruutena toimii X- ja Y-koordinaattien määräämä avaruus, missä X-akseli on horisontaali- ja Y-akseli on vertikaaliakseli. Tämä on helppo visualisoida muodostamalla peukalon ja etusormen väliin suorakulma. Tällöin peukalo voi olla X-akseli ja etusormi Y-akseli. Kolmas ulottuvuus tuo mukaan Z-akselin, joka esimerkiksi OpenGL rajapinnassa määrittää syvyysakselin. **[24]** Joissain muissa ohjelmissa, kuten esimerkiksi Autodesk 3DS ja Blender, syvyysakselina toimii X-akseli. Syvyysakselin määrityksellä on merkitystä, sillä toisin kuin 2D-koordinaatistossa, 3D-koordinaatistoja on kaksi eri tyyppiä. OpenGL:n käyttämä koordinaatisto on niin sanottu vasemman käden koordinaatisto, ja Blenderin käyttämä koordinaatisto on oikean käden koordinaatisto. **[24]**; **[25]** Käsiesimerkkiä jatkaen; jos oikean ja vasemman käden peukalolla ja etusormella muodostaa suorakulman, niin käsien asentoa muuttamalla ja peilaamalla voidaan oikealla kädellä matkia kaikkia vasemman käden kaikkia koordinaatteja. Jos taas molempien käsien peukalo- ja etusormikoordinaatteihin lisätään keskisormella saatava syvyyskoordinaatti, huomataan, ettei vasemman käden koordinaatistoa voi enää matkia oikealla kädellä. Kuvio 4 havainnollistaa vasemman ja oikean käden koordinaatistot.



Kuvio 4. Vasemman ja oikean käden koordinaatistot (Kuva: Wikipedia **[26]**)

2.2.3 Koordinaattiavaruudet

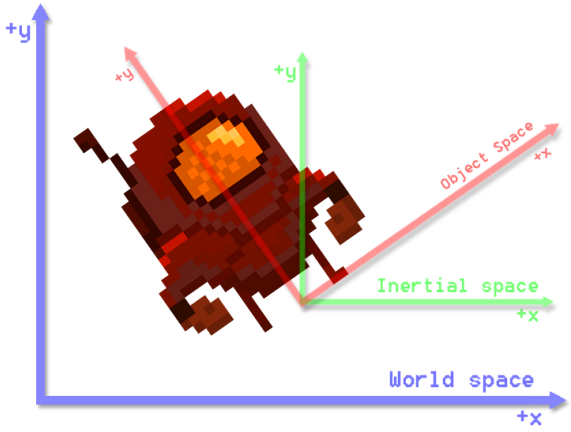
Pelikuvassa näkyvät kappaleet sijaitsevat peliavaruudessa. Kaksiulotteisessa pelissä objektit sijaitsevat kaksiulotteisessa koordinaattiavaruudessa ja kolmiulotteisessa pelissä kolmiulot-

teisessa avaruudessa. Yleensä pelkkä yksi koordinaattiavaruus ei kuitenkaan riitä kuvaamaan kaikkea pelilogiikkaa, vaan tarvitaan useita eri koordinaattiavaruuksia toimivan pelinäkymän kuvaamiseen. Näitä koordinaatistoja ovat seuraavat:

- Maailman avaruus (World space)
- Objektiavaruus (Object space)
- Kameran avaruus (Camera space)
- Inertia-avaruus (Inertial space)

Eri koordinaattiavaruuksia käytetään siksi, että ne toimivat omissa näkökulmissaan yhtä ainutta koordinaattiavaruutta paremmin. **[24]** Otetaan esimerkiksi vaikka rallipelissä olevan auton esittäminen. Ralliauto liikkuu pelimaailmassa olevalla radalla. Auton lokaatio esitetään maailman avaruudessa samaan tapaan, kuin sen liikkumista seurattaisiin kartalta. Jos auto suistuu pelissä tieltä ojaan, muuttuu auton rotaatio fysiikkamoottorin määräämällä tavalla siten, että se kääntyy esimerkiksi katolleen. Fysiikkamoottorit käyttävät lentoratojen laskemisessa apuna inertia-avaruutta, joka pitää kirjaa paikallisen avaruuden ja maailman avaruuden välisistä toiminnoista. Auton kääntyessä ojaan sen vasen takarengas vaurioituu. Tämä vaurio tulkitaan auton paikallisessa koordinaatistossa. Maailman avaruuden näkökulmasta tarkasteltaessa ralliauton vasen rengas on liian erikoistunut käsite kontrolloitavaksi maailman koordinaateilla. Mikäli kamera sijaitsee auton ohjaamossa, välittyy auton kääntyminen katolleen pelaajalle siten, että pelikuva on nyt väärinpäin. Kamera sijaitsee omassa kameraavaruudessa, joka kääntyi tässä tapauksessa auton mukana maailman avaruuteen nähden ylösalaisin. Kuvio 5 esittää koordinaattiavaruuksia kaksiulotteisessa maailmassa. Unityn koordinaattiakselit ilmoitetaan vasemman käden koordinaateilla, jossa Y-akseli on syvyysakseli. Silloin esimerkiksi kappaleen koordinaateissa (3,5,3) keskimmäinen arvo kertoo kohteen sijainnin syvyysarvon. **[27]**

Maailman avaruus on pelimaailman universaali koordinaatisto, johon kaikki pelimaailman objektit sijoitetaan. Se on suurin tarvittava koordinaatisto. Maailman avaruudessa on origin tai world zero, jonka mukaan muut kappaleet sijoitetaan pelimaailmaan. **[27]** Maailman avaruuden koordinaatiston avulla pidetään kirjaa muun muassa eri objektien lokaatiosta ja rotaaatiosta, kameran lokaatiosta ja rotaatiosta, maaston muodoista sekä reitin hakemisesta. Myös Unityssä jokainen ruudulle vedetty tai hierarkianäkymässä luotu objekti katsotaan



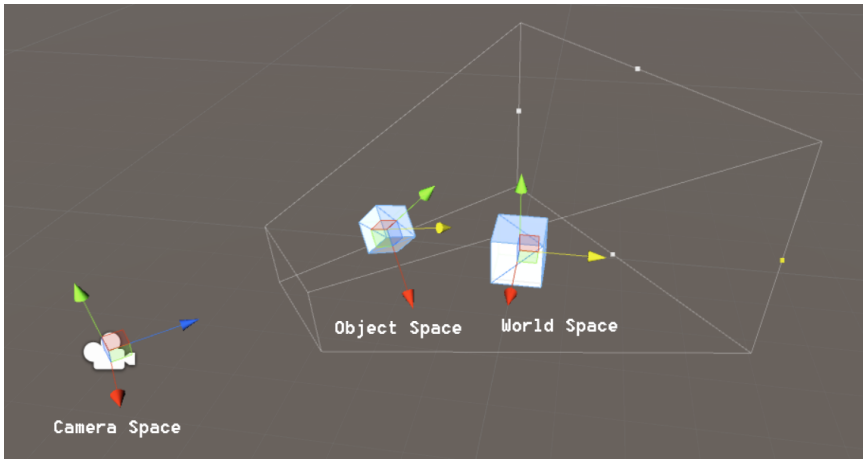
Kuvio 5. Koordinaattiakselit kaksiulotteisessa maailmassa

ensisijaisesti maailman avaruuden mukaan oikeaan paikkaan, rotaatioon sekä skaalaukseen.

Jokainen objekti sijaitsee jossain kohtaa maailman avaruutta, mutta jokaisella objektilla on myös oma paikallinen koordinaatistonsa, objekti-avaruus. Objekti-avaruuden avulla peliobjektien yksityiskohtaisempia toimintoja on helpompi kuvata. Jos jokin objekti muuttuu maailman avaruudessa, niin sen muutokset siirtyvät suoraan objektin paikalliseen avaruuteen. Kuitenkin paikallisessa avaruudessa voidaan tehdä muutoksia maailman avaruudesta välittämättä. Esimerkiksi edellä mainitun rallipelin auton renkaat kannattaa kuvata objekti-avaruuden näkökulmasta maailman avaruuden sijaan. Jos renkaiden pyörimistä tulisi laskea maailman avaruuden koordinaatiston kautta, olisi niiden paikkoja työlästä laskea. Kun renkaiden muutokset hoidetaan niiden paikallisessa objekti-avaruudessa, voidaan niitä pyörittää esimerkiksi vain oman akselinsa ympäri. Kun auto liikkuu kartalla, vaikuttaa liikkuminen maailman avaruudessa automaattisesti myös renkaiden sijaintiin, kulmaan ja skaalaukseen. Paikallisella avaruudella voidaan selvittää myös muita kysymyksiä, kuten onko jokin objekti niin lähellä, että sen kanssa voi olla vuorovaikutuksessa, tai missä suunnassa haluttu kohde on. **[25]** Unity hoitaa maailman ja objektin avaruuden hierarkianäkymällä. Mikäli jokin kappale on toisen kappaleen lapsiobjektina, kuuluu se silloin sen objektiavaruuteen. Lapsiobjekteja voidaan liikuttaa myös erikseen, mutta kaikki mitä tapahtuu sen vanhemmalle, heijastuu myös lapsiobjekteihin.

Kaikki pelimaailman kappaleet kuvataan ruudulle virtuaalisen kameran avulla. Tällä kameralla on myös oma avaruutensa nimeltä kamera-avaruus. Kamera-avaruus käsittelee kameran lokaatiota sekä suuntaa. **[25]** Kappaleet kuvautuvat pelimaailmasta ruudulle lukuisten eri vaiheiden kautta. Kolmiulotteisesta koordinaatistostaan huolimatta kamera-avaruutta käsitellään kaksiulotteisena avaruutena, sillä kameran piirtämä kuva on kuitenkin kaksiulotteinen mukaelma pelimaailmasta. Vaikka ruudulle piirtyvä kuva on kaksiulotteinen, tarvitaan kameran Z-akselia piirrettävien kohteiden syvyystarkasteluun. Jokainen piirrettävä pikseli saa syvyysarvon. Jos jonkin objektin takana olevaa kohdetta pyritään piirtämään, huomataan kyseisen pikselin kohdalla olevan jo dataa. Tällöin tarkastellaan kameran Z-arvoa ja kameraa lähempänä oleva piste kirjoitetaan taulukkoon. Unityssä syvyystarkastelu hoidetaan liukuhihnalla verteksivarjostimen ja pikselivarjostimen välissä. Samalla tehdään taakse tai eteenpäin osoittavien kolmioiden poisto-operaatiot. **[28]** Unityssä pelikuva voi muodostua useammasta kuin yhdestä kamerasta. Usean kameran avulla voidaan luoda erilaisia efektejä tai näkymiä pelikuvaan. Kameroiden kuvat voidaan piirtää joko koko ruudulle tai vain osalle pelikuvaa. Mahdollisia käyttökohteita usealle kameralle on esimerkiksi autopeleissä taustapeili, seikkailu- tai strategiapeleissä pieni kartta, tai vaikka räiskintäpeleissä aseen kiikari. Tutkielman empiirisessä osassa testattavana oleva Gravitoid-mobiilipeli käyttää pelimaailmassa kolmea kameraa. Ensimmäinen kamera kuvaa etualalla olevan pelitilanteen, toinen kamera kuvaa taustalle jäävät 3D-objektit, ja viimeinen kamera hoitaa tausta-avaruuden ja todella kauas jäävät objektit. Toissin kuin annetuissa esimerkkitapauksissa kaikkien kolmen kameran kuvat siirretään koko pelikuvaan päällekkäin. Kolmen kameran käyttö todettiin hyödylliseksi niiden antaman lisäefektin vuoksi. Etualalle jäävä pelikuva voidaan selkeyttää yhdelle kameralle ja taustagrafiikkaa voi muokata lennosta ilman, että varsinainen pelitilanteen visuaalinen ilme kärsii muokkauksista.

Usean kameran käytössä ongelmaksi muodostuu se, minkä kameran tiedot tulee piirtää päällimmäiseksi. Tällöin apuna käytetään piirtotasoja (layer). Piirtotasoissa pienimmän luvun saanut kappale piirretään muiden päälle. Kuvio 6 havainnollistaa eri koordinaattiakseleita. Kuviossa vasemmalla on kameran paikallisen koordinaatiston suunta sekä näkymäfrustrum, keskellä pienemmän kuution paikallinen objekti-avaruus, ja oikealla isompi kuutio maailman avaruuden koordinaatistossa.



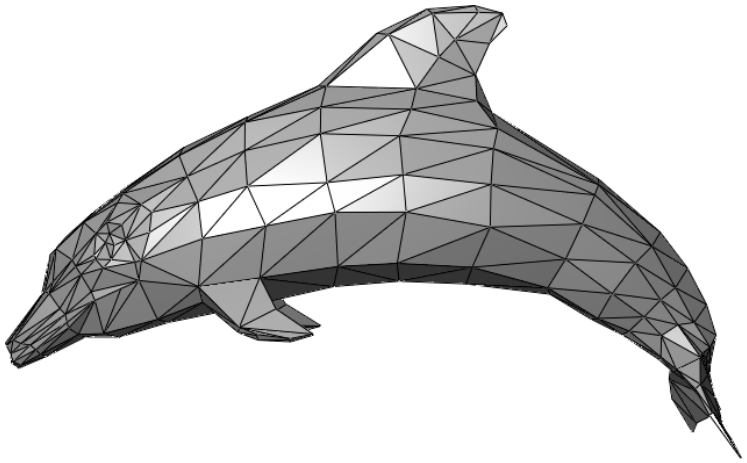
Kuvio 6. Eri koordinaatistoja Unity-ympäristössä.

Viimeisenä koodrinaattiavaruutena on inertia-avaruus. Se sijoittuu maailman avaruuden ja objekti-avaruuden puoliväliin. **[25]** Inertia-avaruuden alkupiste on sama kuin objekti-avaruuden alkupiste. Koordinaatiston akselit taas ovat jatkuvasti yhdensuuntaiset maailman avaruuden akselien kanssa. Tällöin saadaan sopiva puoliväli objektin käyttäytymiselle maailman koordinaattiavaruudessa. Esimerkkinä edellä mainittu rallipelin auto, joka liikkuu pelimaailmassa. Mikäli autoa käännetään vasemmalle, on helpompi verrata auton objekti-avaruuden kulmaa maailman avaruuden kanssa yhdensuuntaiseen inertia-avaruuteen, joka on valmiiksi objekti-avaruuden origossa. Ilman inertia-avaruutta auton lokaatio pitäisi siirtää maailman avaruuden origoon, laskea käännettävä kulma ja siirtää auto takaisin alkuperäiseen lokaatioon.

2.2.4 3D-data pelinäkymään

Pelimoottorit laskevat 3D-datan ruudulle kolmiomuodossa. 3D-grafiikkaa voidaan laskea muunkinlaisilla primitiiveillä, kuten kolmiosarjoilla tai NURBS-pinnoilla. Kolmiointi on silti yleisin tapa käsitellä dataa sen ollessa yksinkertaisin esitysmuoto tasosta. Unity tukee kolmioitua tai nelikulmioitua 3D-dataa. **[29]** Kolmiota käytetään siksi, että kolmion kulmapisteiden välille on aina mahdollista muodostaa taso. Luodulle tasolle määritetään myös suuntanormaali, joka kertoo miten päin luotu taso on. Mikäli kolmio ei osoita kameraan päin, ei sitä tarvitse piirtää. Näin voidaan puolittaa piirtotyö, sillä kolmiosta lasketaan aina vain kameraan osoittava puoli. Tätä kutsutaan taakse osoittavien monikulmioiden poistoksi (back-face

culling). Taakse osoittavien kolmioiden poisto on tehokas tapa optimoida pelinäkymää ja se on yleisesti pelimoottoreissa automaattisesti päällä. Unity pelimoottorissa taakse osoittavien kolmioiden poistoa voi hallita varjostinkoodilla. Kuvio 7 havainnollistaa nelikulmioilla luodun delfiinin kolmiointioperaation jälkeen.

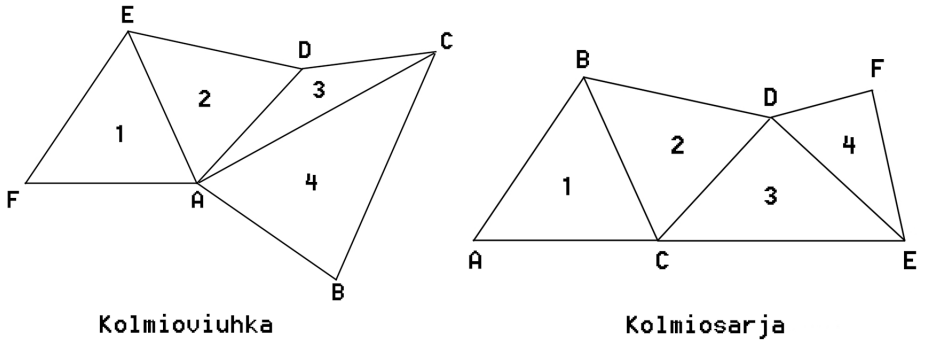


Kuvio 7. Kolmioitu delfiinin 3D-malli. (Kuva: Wikipedia **[30]**)

Kolmioiden tasoille lasketaan normaalit, jotta tiedetään miten päin kolmio näkyy pelimaailmassa. Pintanormaaleja voidaan tallentaa joko kolmioiden pinnalle, kärkipisteille tai molemmille. **[25]** Jos kulmapisteiden normaaleja ei ole määritelty, ei voida tietää onko kulmapisteiden välinen taso oikeaan suuntaan, eikä kappaleelle voida laskea pyöristyksiä oikein. Tämän lisäksi normaalitietoja tarvitaan törmäyksen tunnistuksessa, taakse osoittavien monikulmioiden poistossa sekä partikkeleiden kimmoittamisen simuloimisessa. Jotkut tiedostomuodot sisältävät normaalitiedot valmiina, mutta joissain tapauksissa normaalit lasketaan jälkikäteen. Kolmion kärkipisteet voivat kiertää joko myötä- tai vastapäivään sovelluksesta riippuen. Normaalien orientaatio voidaan laskea valitsemalla kappaleen jokin normaaliton kolmio, selvittämällä sen naapurikolmiot ja kääntämällä niiden orientaatio tarvittaessa oikeinpäin. Tämän jälkeen etsitään mahdolliset pehmennysryhmät (smoothing groups) ja lasketaan kärkipisteille normaalit. Näin kappale ei tarvitse loputonta määrää kolmiota pyöreyden esittämiseen, vaan verteksinormaalien avulla pyöreys voidaan aikaansaada pienelläkin kolmiomäärällä. **[31]**

Grafiikkakortit osaavat laskea muutamia monikulmioita kolmioita tehokkaammin. Yleisimmät laskutyötä nopeuttavat ratkaisut ovat kolmioviuhka ja kolmiosarja. Niiden edut yksittäis-

ten kolmioiden laskemisen verrattuna ovat kolmioiden naapurien yhtenäiset kärkipisteet. Esimerkiksi kolmiosarjassa kaksi vierekkäistä kolmiota jakavat kaksi kärkipistettä keskenään, jolloin niiden koordinaatit tarvitsee laskea vain kerran. Kuviossa 8 on neljän kolmion muodostama kolmiosarja sekä kolmioviuhka. Mikäli kolmiosarja lasketaan pelkkinä kolmioina, pitää sarjasta tallentaa muistiin kärkipisteet ABC, BDC, DEC sekä DFE. Sarjana tallennettuna samasta monikulmiosta tarvitsee tallentaa vain kuusi kärkipistettä muodossa ABCDEF. Grafiikkakortti purkaa tämän sarjan samaksi kolmiosarjaksi, kuin erikseen tallennetut kolmiot. Kolmioviuhka käyttää samaa periaatetta sillä erotuksella, että jokainen kolmio jakaa yhden kärkipisteen muiden kolmioiden kanssa. Kuten kuviosta 8 voi havaita, viuhkan jokainen kolmio jakaa pisteen A muiden kanssa sekä yhden toisen pisteen naapurinsa kanssa.



Kuvio 8. Kolmioviuhka ja kolmiosarja.

Kolmiota monimuotoisempi primitiivi on monikulmio. Editointivaiheessa 3D-kappaleet esitetään yleensä monikulmioina, kuten nelikulmioina. Monikulmio, eli polygoni, koostuu kolmesta tai useammasta eri kulmapisteestä, joiden välille muodostetaan taso. Monikulmioista on mallinnusvaiheessa suuri etu, sillä ruudulla on vähemmän turhaa tietoa ja editoiminen on selkeämpää. Monikulmiot luokitellaan kahteen eri pääryhmään: yksinkertaisiin monikulmioihin sekä monimutkaisiin monikulmioihin. Yksinkertaisten monikulmioiden kulmapisteiden välille voidaan piirtää yhtenäinen pinta, kun taas monimutkaisten monikulmioiden tasossa esiintyy reikiä. **[25]** Koska monimutkaisia monikulmioita on vaikea esittää järkevästi, muutetaan niiden rakennetta hienovaraisesti. Monimutkaisen tason reiät muutetaan monikulmion koveraksi reunaksi, jonka yhdyskäytävän molemmat reunat ovat samassa paikassa. Tällöin reunasaumaa ei näy monikulmiosta, mutta sen kaikki reunapisteet muodostavat monikulmion reunat. Monikulmion useasta kärkipisteestä johtuen monikulmio voi leikata

oman tasonsa kanssa. Tämä on ei-toivottu tilanne ja aiheuttaa tason kanssa ongelmia. Editointivaiheessa kannattaa varmistaa, ettei malleista löydy tällaisia päällekkäisyyksiä. Yksinkertaiset monikulmiot voidaan jakaa kuperiin ja koveriin malleihin. Koverissa malleissa on vähintään yksi "lommo", eli kärkipiste jonka kulma on yli 180 astetta. Koverat monikulmiot pystyy jakamaan kuperiksi monikulmioiksi tarkistamalla kärkipisteiden kulmat ja tarvittaessa leikkaamaan koverasta kärkipisteestä lähtien monikulmio kahdeksi pienemmäksi monikulmioksi. Viime kädessä päädytään kolmioihin. Monikulmiomalleja on selkeämpi käsitellä editoitaessa, sillä niiden kanssa toimittaessa mallin topologiasta saa paremmin selvää. Editointiohjelmissa on valmiina yleensä vähintään taso- ja ympyräpolygonit. Grafiikkaa piirrettäessä monikulmiot hajotetaan juuri kolmioiksi, kolmioviuhkaksi tai kolmiosarjaksi.

Monikulmioita yhdistettäessä saadaan aikaan monikulmioverkko. Käytännössä kaikki pelinäkymässä olevat kappaleet muodostuvat kolmioverkoista. Selkeä esimerkki kolmioverkoista on esimerkiksi pelimaailman maasto. Maasto voi sisältää suurenkin määrän kolmioita, jolloin sen reaaliaikainen piirtäminen on raskasta. Grafiikkaliukuhihnalla monikulmioverkkoja pyritään muuttamaan kolmiosarjaksi tai kolmioviuhkoiksi, jolloin ne voidaan prosessoida kolmioita tehokkaammin. **[32]** Monikulmioverkolle on mahdollista tehdä muutamia toimenpiteitä käyttötarkoituksesta riippuen. Näitä toimenpiteitä on:

- Kulmapisteiden yhdistäminen (Vertex welding)
- Kulmapisteiden jakaminen (vertex split)
- Tason erotus (detach face)
- Reunan poisto (edge collapse)

Kaikkia edellä mainittuja operaatioita tarvitaan kolmioverkon tarkkuustason muutoksissa, jolloin tiheästäkin kolmioverkkomallista saadaan pelimoottoreihin soveltuva versio ilman huomattavaa topologian muuntumista. Kolmioverkon reaaliaikainen optimointi on hyvin yleinen tapa optimoida pelimaailmaa. Kulmapisteiden yhdistämisessä laajan kulman kulmapisteitä yhdistetään yleensä keskenään, jolloin reunanmuotoja edustavat matalan kulman kärkipisteet pysyvät tarkkoina tasaisen pinnan vähentäessä kolmioita.

Monikulmioista ja monikulmioverkoista saadaan muodostettua kappaleita (object). Kappale voi muodostua joko yhdestä monikulmioverkosta, muutamasta kolmiosta tai monimut-

kaisemmista yhdistelmistä. Kappale on yleensä jokin tunnistettava objekti näkymässä. Esimerkiksi kaupunkisimulaatiossa yksi auto voi olla yksi kappale. Pelinäkymässä kappale voi käsittää myös lapsikappaleita, jotka perivät vanhempansa objektiavaruuden.

Valot sekä objektit sijoittuvat maailmassa kameran näkymään. Näkymä on se rajattu aluem jonka kautta pelimaailmaa katsotaan. Kameralla on maailmassa oma lokaatio, skaalaus sekä orientaatio. Kamera määrää katselukulman ja perspektiivivääristymän piirrettävälle näkymälle. Kameran näkymästä käytetään kahta erilaista variaatiota: ortografista sekä perspektiivistä kameraa. Ortografinen kamera piirtää kuvan ilman perspektiivivääristymää, niin sanotusti kappaleen oikeilla mitoilla. Perspektiivikamera ottaa huomioon etäisyydestä johtuvan perspektiivin vääristymän ja muokkaa kuvaa sen mukaisesti. Ortografinen kamera soveltuu hyvin esimerkiksi arkkitehtisuunnitteluun, pohjapiirrustuksiin ja leikkauskuviin, kun taas perspektiivikameralla simuloidaan oikean kameran toimintoja pyrkien realistisen näköiseen kuvantamiseen. Perspektiivikamerassa on mukana näkymäfrustrum, joka leikkaa kuvasta liian lähellä ja liian kaukana olevat kappaleet pois. Liian lähellä olevat kappaleet voivat haitata haluttua näkymää, ja liian kaukana olevat kohteet käyttävät turhaan grafiikkaprosessorien tehoja. Kaukana olevien kappaleiden kolmiomääriä pyritään myös pienentämään tarkkuustasoilla, minkä seurauksena kolmioiden määrä kameranäkymässä pysyy vakiona läheltä kauas katsottaessa. Unityssä 2D-pelit käyttävät pääosin ortograafista kameraa 3D-ympäristössä, missä syvyysakseli on lukittu. 3D-peleissä käytetään perspektiivikameraa. **[27]**

Ruudulla näkyvä pelikuva prosessoidaan lukuisten erilaisten vaiheiden kautta grafiikkarajapinnasta riippuen. Otetaan esimerkiksi OpenGL ES-grafiikkaliukuhna. OpenGL ES on erityisesti mobiililaitteille suunnattu API. 3D-Kuvan piirtämiseksi käydään grafiikkaliukuhihnalla läpi seuraavat vaiheet: verteksivarjostus, primitiivien kasaus, näkymän leikkaus, rasterointi, pikselivarjostus, fragmenttikohtaiset operaatiot ja kehyspuskuri. Varjostimet (shader) ovat ohjelmoitavia tasoja, joissa lasketaan 3D-datan eri vaiheita annetuilla parametreillä. Verteksivarjostuksessa lakeminen keskittyy kulmapisteiden laskemiseen, pikselivarjostin taas laskee kolmioista rasteroitavan tason pikselien väriarvoja. **[31]** OpenGL ES:n käyttämästä grafiikkaliukuhihnasta käyttäjän muokattavissa ovat ainoastaan verteksidata, väriarvot, verteksivarjostin ja pikselivarjostin. Tarvittaessa joitain operaatioita voi jättää välistä pois. Esimerkiksi yksinkertaisten varjostusten esittämiseen riittää pelkkä verteksivarjos-

timen hyödyntäminen. **[33]** Unity käyttää mobiililaitteille käännetyissä peleissä OpenGL ES-grafiikkaliukuhihnaa. Unityn varjostimet ovat editointivaiheessa korkean tason varjostinkielellä, joka käännetään peliä käännettäessä OpenGL ES muotoon. OpenGL ES ja Unity - OpenGL ES liukuhihna käydään läpi tarkemmin luvuissa 2.5 ja 2.6.

2.3 Pintamateriaali ja LOD

3D-kuvanta sisältää kolmioverkon lisäksi yleensä pintamateriaalin. Yhdenväriset varjostetut kappaleet näyttävät vahamaisilta esityksiltä luonnollisen näköisen objektin sijaan. Tietysti mallissa voi värjätä kolmioita eriväriksiksi muodostaen kolmioverkolla yksityiskohtia, mutta tämä tekniikka vaatii todella paljon kolmioita halutun vaikutuksen aikaansaamiseksi. Tehokkaampi tekniikka on kääriä kappaleen pinnalle tekstuurikuva, joka esittää yksityiskohtaisempia pinnanmuotoja ja värejä. **[31]** Tekstuuri on siis bittikartta, joka heijastetaan haluttuun kohtaan objektia. Objektin kolmiot sijoitetaan johonkin kohtaan tätä bittikarttaa, ja sen rajaamat kuvapikselit, eli tekselit, parametrisoidaan kolmioiden pinnalle. Tekstuurin voi ympäröidä objektin pinnalle useammalla erilaisella tyylillä, ja mallinnusohjelmat kuten Autodesk 3DS, Maya ja Blender tarjoavat näistä vähintään taso-, kuutio-, pallo- ja UV-tekniikat. Tekstuuritiedostot lasketaan kaksiulotteisessa koordinaatistossa, jonka akseleita kuvataan U- ja V-kirjaimin. UV-koordinaatteja käytetään siksi, etteivät ne sekoittuisi 3D-maailman X-,Y-, ja Z-arvoihin. **[25]** Kun kappaleen kolmioille on tiedossa niiden tekstuurin UV-sijainti, voidaan se parametrisoida oikein kolmion pintaan. Tämän jälkeen näkymä projisoidaan kameran avulla kaksiulotteiseksi esitykseksi.

Tekstuuritiedostojen lisäksi kappaleille asetetaan materiaali. Pintamateriaali määrittää sen, miten kappale heijastaa valoa (ambient, diffuusi, spekulaari). Peliobjekteilla ei tarvitse välttämättä olla tekstuuritietoja, mutta ilman materiaalia ei kappaletta voi kuvantaa. Mikäli materiaalia ei erikseen määritellä, käytetään ohjelman tai pelimoottorin oletusmateriaalia heijastusten laskemiseen. Unity pelimoottorissa jokaisella objektilla on pakko olla vähintään yksi materiaali. Materiaaliin on Unityssä liitetty myös varjostinohjelma, joka määrittää kappaleen piirtotyylin. **[34]** Tekstuureja on mahdollista lisätä useampia saman kappaleen pinnalle. Pelkkä pintakuviotekstuuri on vain yksi tapa lisätä kappaleen yksityiskohtaisuutta. Erilaisia tekstuurityyppejä ovat muun muassa:

- Tekstuurikartat (Texture map)
- Valaistuskartat (Lightmap)
- Kyhmytyskartat (Bump map)
- Normaalikartat (Normal map)
- Parallaksikartat (Parallax map)
- Siirtymäkartat (Displacement map)

Valaistuskartta sisältää nimensä mukaisesti tekstuuritiedostossa ympäröivän valaistuksen objektille. Haluttuun objektiin vaikuttavien valojen valaistumäärät lasketaan objektin pinnalle sijoitettavaan valaistuskarttaan. Kun valaistuskartta lasketaan objektille etukäteen ja sijoitetaan tekstuuritiedostoon, ei sitä tarvitse laskea jatkuvasti reaaliajassa. Näin vältytään dynaamisten valaistusten aiheuttamilta mahdollisilta valaistusartefakteilta. Lisäksi vähätehoisissa laitteissa laskentatehoa säästyy muuhun toimintaan. **[35]** Valaistuskartat rajoittuvat yleensä vain staattisiin objekteihin ja kenttärakenteisiin.

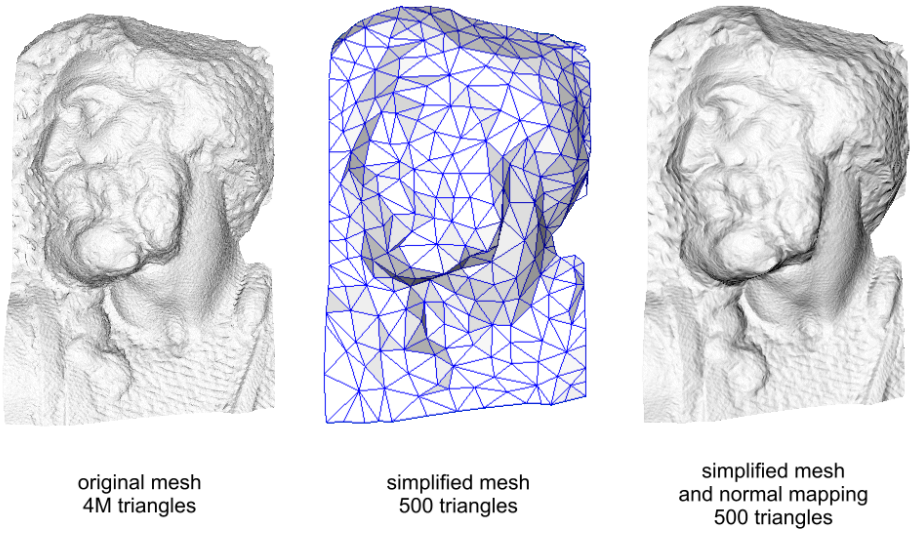
Kyhmytyskartoitus on tekniikka, jolla objektin tasaiselle pinnalle luodaan varjostuksen avulla illuusio epätasaisuuksista. Kyhmytyskartat muodostuvat harmaan eri sävyistä, joissa harmaasävyasteikon vaaleat arvot katsotaan pinnasta nouseviksi kohdiksi ja tummat laskeviksi. Toisin kuin siirtymäkartoissa, kyhmytyskartta ei muokkaa objektin muotoja, joten esimerkiksi kyhmytetyn kappaleen siluetti on silti sileä. Kyhmytyskartat eivät myöskään osaa esittää hyvin esimerkiksi vinosta tulevaa valoa, joten sen luoma illuusio ei ole aivan täydellinen kaikissa katselukulmissa. Kyhmytyskartat ovat tehokkaita pinnan pienien epätasaisuuksien ilmentämisessä, sillä ne vievät vähän tilaa ja toimivat hyvin orgaanisten muotojen kanssa. **[36]**

Kyhmytyskartasta on myöhemmin kehitetty suurempaa väriskaalaa käyttävä normaalikartta (normal map). Kyhmytyskartan tavoin normaalikartta muuttaa pinnan valon ja varjon käyttäytymistä kartan mukaan luoden illuusion pinnan epätasaisuudesta. Kyhmytyskartasta poiketen normaalikartta käyttää koko RGB-väriskaalaa muotojen esittämiseen. Tämä luo kyhmytyskarttaan nähden sen edun, että pinnan epätasaisuuksilla on väriarvoista saatu normaali. Varjopuolena tässä käytänteessä on se, että normaalikartta on sidottuna kappaleen pintaan. Kyhmytyskartta ei ole sidonnainen tasoon ja se voi vääristää minkä tahansa pinnan normaaleja. **[36]** RGB-arvot siirretään suoraan XYZ-koordinaateiksi. Poikkeuksena on normaa-

likartan Z-arvo, sillä siitä ei tarvitse näyttää kuin arvot nollasta -1:een, koska geometriaa, joka osoittaa katsojasta ulospäin, ei näytetä. Normaalikartan värikoordinaatit siirtyvät XYZ-akseliin seuraavasti:

```
X: -1  +1 : Red: 0 - 255
Y: -1  +1 : Green: 0 - 255
Z: 0  -1: Blue: 128 - 255
```

Esimerkiksi suoraan kappaleesta ulospäin osoittava piste normaalikartassa saisi arvon XYZ (0,0,-1). RGB-koordinaateiksi muunnettuna sama olisi (128, 128, 255). Peliteollisuudessa normaalikartoitus on yleistynyt lähes rutiininomaiseksi tavaksi lisätä kappaleiden tarkkuutta. Työmäärältään normaalikartoitus on tosin kohtuullisen vaativaa, sillä kappaleesta joko tehdään korkean resoluution malli, josta normaalikartta muodostetaan, tai normaalikartoitus luodaan kuvankäsittelyohjelmassa suoraan tekstuuritiedostoksi. Tämän jälkeen malli joko optimoidaan tai tehdään kokonaan uudestaan mahdollisimman matalaresoluutioiseksi, jotta se veisi mahdollisimman vähän laskentatehoja. Normaalikartoitus vaatii myös ylimääräistä tekstuurimuistia bittikartoille. Kuvio 9 esittää normaalikartoituksen tehoa kolmioverkon optimoinnissa.



Kuvio 9. Esimerkki normaalikartan tehokkuudesta. (Kuva: Wikipedia **[38]**)

Parallaksikartta (Parallax occlusion map) on normaalikartan ja kyhmytyskartan tapainen tyyli lisätä pinnanmuotoja objektiin. Sen erikoisuus piilee tekstuuritiedoston UV-koordinaattien

siirtämisessä syvyysvaikutelman aikaansaamiseksi. Parallaksikartassa harmaasävykartan normaalitaso on 1.0, ja alin mahdollinen taso ilmaistaan arvolla 0.0. Kartan monimuotoisuus saadaan laskettua, kun yhdistetään kartan korkeusarvot katselukulmaan. Näin voidaan laskea, mitkä pikselit jäävät korkeuskartassa näkyviin. Laskut suoritetaan yksinkertaistetulla säteenseurannalla pikselivarjostimessa. Parallaksikartta ei kuitenkaan sisällä varjojen muodostamista. Parallaksikarttaa käytetään siksi usein joko kyhmytyskartan tai normaalikartan kanssa yhdessä, jolloin valon ja varjon laskeminen tulee jommasta kummasta ja tekstuurimuunnos parallaksikartasta. **[39]** Unity käytti parallaksisiirtymien laskemiseen omia Parallax diffuse- ja Parallax specular-varjostimia, kunnes nämä ominaisuudet yhdistettiin oletusvarjostimeen korkeuskarttakohdaksi. Kappaleelle voi siis normaalikartan lisäksi lisätä korkeuskartan joka toimii parallaksikartan tavoin. Parallaksivääristymävarjostimet ovat kuitenkin Unityn tarjoamista varjostimista raskaimpia käyttää, joten niiden käyttöä suositellaan vältettävän aina kun mahdollista.

Vaikka normaalikartan ja kyhmytyskartan oikea käyttö luo tehokkaasti lisää tarkkuutta objektille, on olemassa vieläkin tehokkaampi tapa lisätä mallin tarkkuutta: kun pelkän pinnan varjostuksen muokkaaminen ei tuota haluttua tulosta, käytetään apuna siirtymäkarttaa. Siirtymäkartta muokkaa kappaleen kolmioverkon kolmioiden lokaatiota kolmioverkon paikallisen normaalin suuntaan annetun korkeuskartan mukaan. Korkeuskartta voi olla tekstuuri, korkeuskartta tai proseduraalisesti luotu tekstuuritiedosto. Siirtymäkartan mustaa väriä ei muuteta ja valkoista muutetaan määritetyn maksimiarvon verran. Kaikki harmaan sävyt sijoittuvat näiden kahden ääriarvon väliin. Siirtymäkartan huono puoli normaalikarttaan ja kyhmytyskarttaan verrattuna on varsinaisen kolmioverkon muokkaamista. Hyvänlaatuisen siirtymäkarttaefektin aikaansaamiseksi tarvitaan kuitenkin suuri määrä mikromonikulmioita hyvän tuloksen aikaansaamiseksi. Tällainen käytäntö on reaaliaikaisessa renderöimisessä huono ratkaisu. **[37]** Kolmioverkko tesseloidaan yleensä peleissä dynaamisesti eri tarkkuustasoille siten, että kameran läheisyydessä kolmioverkon tiheys on suurempi, kuin kauempana esiintyvä geometria. Siirtymäkarttaa käytetään usein proseduraalisten mallien ja maaston kuvaamiseen.

Tarkka maaston kuvaus tai yksityiskohtia täynnä olevat peliobjektit käyttävät luonnollisesti paljon laitteen laskentatehoja pelinäkymän piirtämisessä. Mikäli peliobjekteja on useam-

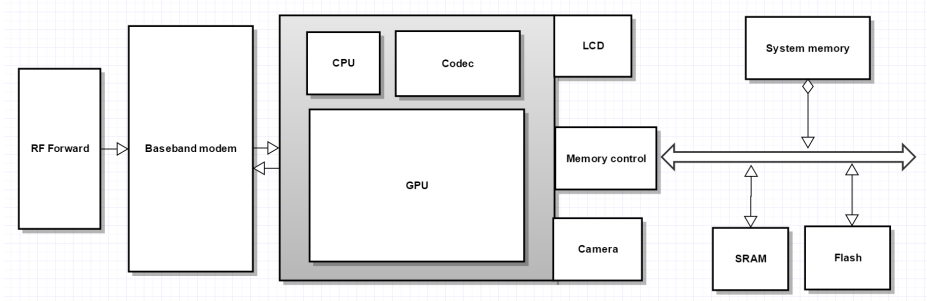
pia ja näkymä kattaa suuren palan maastoa, käy näkymän renderöiminen todella raskaaksi. Jotta näkymää voisi pyörittää reaaliaikaisesti, pitää kolmioverkon tarkkuutta optimoida. Tätä varten on kehitetty eri tarkkuustasoja (Level of detail), joilla objekteja ja pelimaisemaa voidaan kuvantaa. Tarkkuustasot määrittelevät näkymässä olevien objektien muotojen monimutkaisuutta verrattuna sen etäisyyteen kamerasta. Mitä kauempana piirrettävä objekti on näkymässä, sitä vähemmän se tarvitsee pikkutarkkoja yksityiskohtia. Lentosimulaattorit ja avoimen maailman pelit käyttävät liukuvaa tarkkuudentason tekniikkaa maaston kuvantamiseen. Kameraa lähellä oleva maa piirretään tiheämmällä kolmioverkolla, kuin kaukana näkyvä maasto. Kolmioverkkojen muokkaus tapahtuu yleensä joko reunojen luhistamisella (edge collapse) tai kulmapisteitä poistamalla. **[40]** Tieto maastosta tallennetaan nelipuujärjestelmään tai kasipuujärjestelmään.

Tarkkuustasoja käytetään paljon myös objektien kanssa. Objektien kanssa kulmapisteiden poistaminen tai reunojen luhistaminen voi tosin tuoda ei-haluttuja tuloksia. Tällöin usein turvaudutaan manuaaliseen tarkkuustasojen määrittämiseen. Samasta objektista tehdään haluttu määrä eri tarkkuustasolla olevia versioita. Nämä tallennetaan listaan, ja etäisyyden perusteella vaihdetaan malli joko tarkempaan tai epätarkempaan versioon samasta mallista. Tämä aiheuttaa kuitenkin siirtymävaiheessa särähdyksen, kun objekti vaihdetaan yhdestä toiseen. Efektin minimoimiseksi on tehty erilaisia pikselivarjostimessa ajettavia sulautusefektejä.

2.4 Mobiili 3D-grafiikka-alustana

Mobiililaitteiden yleistymisen myötä mobiilipelimarkkinat lähtivät räjähdysmäiseen nousuun nostaen suomalaisiakin pelitaloja, kuten Supercell, Rovio ja Fingersoft maailmankuuluiksi. Mobiililaitteiden laskentatehot ovat kasvaneet vuosi vuodelta ja nyt uusimmat laitteet tukevat jo 4K resoluutioita ja esimerkiksi DirectX 11 rajapintaa. Mobiililaitteiden suunnittelu on kuitenkin haastava prosessi, sillä kaikki mobiililaitteiden tarjoamat ominaisuudet pitää puristaa taskukokoiseen kannettavaan laitteeseen. Tämä asettaa erityisiä haasteita erityisesti virrankulutukseen, laskentaoperaatioiden määrään ja lämmöntuotantoon. Mobiililaitteiden suunnittelussa on päädytty SoC (System on a Chip) tyyppiseen ratkaisuun. SoC lähestymistavassa koko laitteen arkkitehtuuri suunnitellaan yhdeksi piirikokonaisuudeksi. Kun kaikki eri komponentit "sulautetaan"yhdeksi kokonaisuudeksi, voidaan sen koko, virrankulutus,

lämmöntuotanto jne laskea tiettyjen vaatimusten mukaan. SoC ratkaisu ilmenee esimerkiksi muisteissa, jotka jaetaan grafiikkaprosessorin ja laskentaprosessorin kesken. **[31]** Erilaisia modulaarisia ratkaisuja, kuten Googlen Project Ara tai PuzzlePhone, on kehitelty. Yhteensopivuusongelmien ja spesifien rajapintojen takia modulaariratkaisut ovat toistaiseksi pysyneet vielä konseptitasolla. Suuret älypuhelinvalmistajat, kuten Apple, Samsung ja Asus luottavat yhden piirilevyn suunnitteluratkaisuun. Kuvio 10 esittää laatikkodiagrammina yksinkertaistetun SoC arkkitehtuurin. Videoiden ja musiikin enkoodaukselle ja dekoodaukselle on omat piirinsä, mutta kaikki laitteen osa-alueet jakavat saman muistin.



Kuvio 10. System on a chip arkkitehtuuri **[31]**

Varsinaista laskentatehoa taskukokoiseen laitteeseen on mahdotonta lisätä loputtomiin. Transistorikokojen lähestyessä minimikokoja niidenkin määrää alkaa olla vaikea kasvattaa. Suurimmat hyödyt saadaan tehokkaasta optimoinnista. Optimointi otetaan huomioon jo mobiililaitetta suunniteltaessa ja se jatkuu sovellusten suunnitteluun asti. Eniten käytetyille toiminnoille suunnitellaan omat laskupiirit. Esimerkiksi äänen toistaminen voidaan hoitaa mobiililaitteen prosessorin kautta. Prosessorin käyttäminen on kuitenkin virrrankulutuksen näkökulmasta kallista, joten parempi keino on käyttää erillistä äänen käsittelyyn suunniteltua äänipiiriä. Sama periaate pätee muun muassa videotiedostojen enkoodaukseen ja dekoodaukseen, sekä kameran kuvan prosessoimiseen. Nämä vaiheet on mahdollista laskea prosessorilla, mutta se on sillä työlästä ja siksi kuluttaa paljon virtaa. Erillinen videoiden enkoodaukseen ja dekoodaukseen suunniteltu piiri hoitaa vaaditun työn tehokkaammin vähemmällä virrankulutuksella.

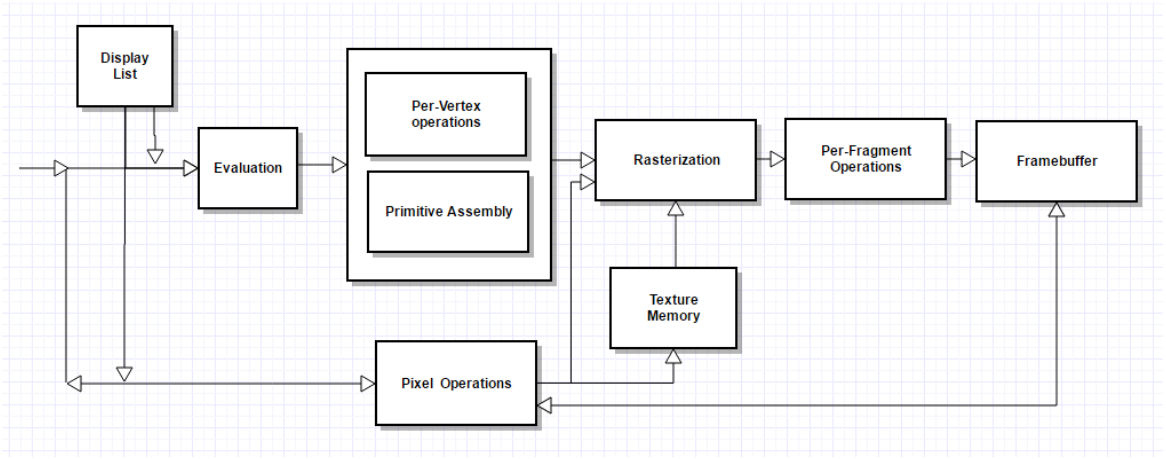
Ruudulle renderöitävästä 2D/3D-materiaalista vastaa erillinen grafiikkapiiri. Uusimpien mobiililaitteiden näytön resoluutio on 4K, mikä asettaa grafiikkapiirille kovia paineita. 4K resoluution sulava ruudunpäivitys vaatii pöytäkoneiltakin huomattavia tehoja. Nyt sama las-

kentateho pitää pakata taskukokoiseen laitteeseen. Unity pääsi vuonna 2013 tekemässään teknologiademossa sulavaan 30 FPS (frames per second) nopeuteen iPad 4 tablettitietokoneella. **[41]** Mobiililaitteen näytön resoluutio on 2048 * 1536 pikseliä. Geometria operaatiot vaativat paljon laskemista ja renderöinti vaatii laskemisen lisäksi paljon dataa. Erilaisia optimointimahdollisuuksia kuitenkin löytyy grafiikkaliukuhihnan eri vaiheista. Esimerkiksi laskuoperaatioissa voidaan käyttää liukulukujen (floating point) sijaan kiintopiste (fixed point) arvoja. Näiden tarkkuus ei vastaa floating pointtia, mutta ne voidaan laskea nopeammin pienemmällä virtamäärällä, mikä on mobiililaitteissa suuri etu. Pieni laskennallinen epätarkkuus on todettu vähäiseksi hinnaksi virransäästöön verrattuna. Muita optimoinnin paikkoja on esimerkiksi pikselien syvyysarvojen määrittämisessä. Normaali OpenGL liukuhihna suorittaa syvyystarkastelun varjostuksen loppupäässä. **[18]** Tämä tarkoittaa turhaa pikselinvarjostusta niille pikselelille, jotka jäävät toisten fragmentin taakse piiloon. Mikäli syvyystarkastelu suoritetaan välittömästi interpolaatiovaiheen jälkeen, taustalle jääviä pikseleitä ei missään vaiheessa edes lasketa. Näin voidaan säästää lukuisia ylimääräisiä laskuoperaatioita varsinaiseen näkyvän datan laskemiseen.

Vuonna 2013 Unity Technologies testasi mobiililaitteiden suorituskykyä teknologiademon avulla. Testien pääasiallinen tarkoitus oli kartoittaa mobiililaitteiden tehokkuutta, sekä uusien visuaalisten ratkaisujen toimivuutta. Laitteistopohjana toimi PowerVR-5xx, Tegra4, Adreno-3x0, sekä Mali-T6xx. Nämä laitteet kykenivät piirtämään ruudulle keskimäärin 30 ruudun sekuntinopeudella 250 000 staattista verteksiä, sekä 80 000 animoitua verteksiä **[41]**

2.5 OpenGL ES

OpenGL ES (Embedded System) on OpenGL tuoteperheeseen kuuluva erityisesti mobiililaitteita varten suunniteltu Application Programming Interface (API). Se eroaa PC puolella käytettävästä OpenGL apista karsituilla ominaisuuksilla ja vähän virtaa kuluttavilla laskuoperaatioillaan. OpenGL ES esiteltiin ensimmäisen kerran vuonna 2003. Seuraavana vuonna implementoitiin ajuripäivitys ja versionumero kasvoi 1.1:een.Tällöin varjostinliukuhihna toimi vielä Fixed Function tavalla, eikä sitä suuremmin päässyt kustomoimaan. Kuvio 11 kuvastaa OpenGL version 1.5 grafiikkaliukuhihnan laatikkodiagrammin. **[42]**

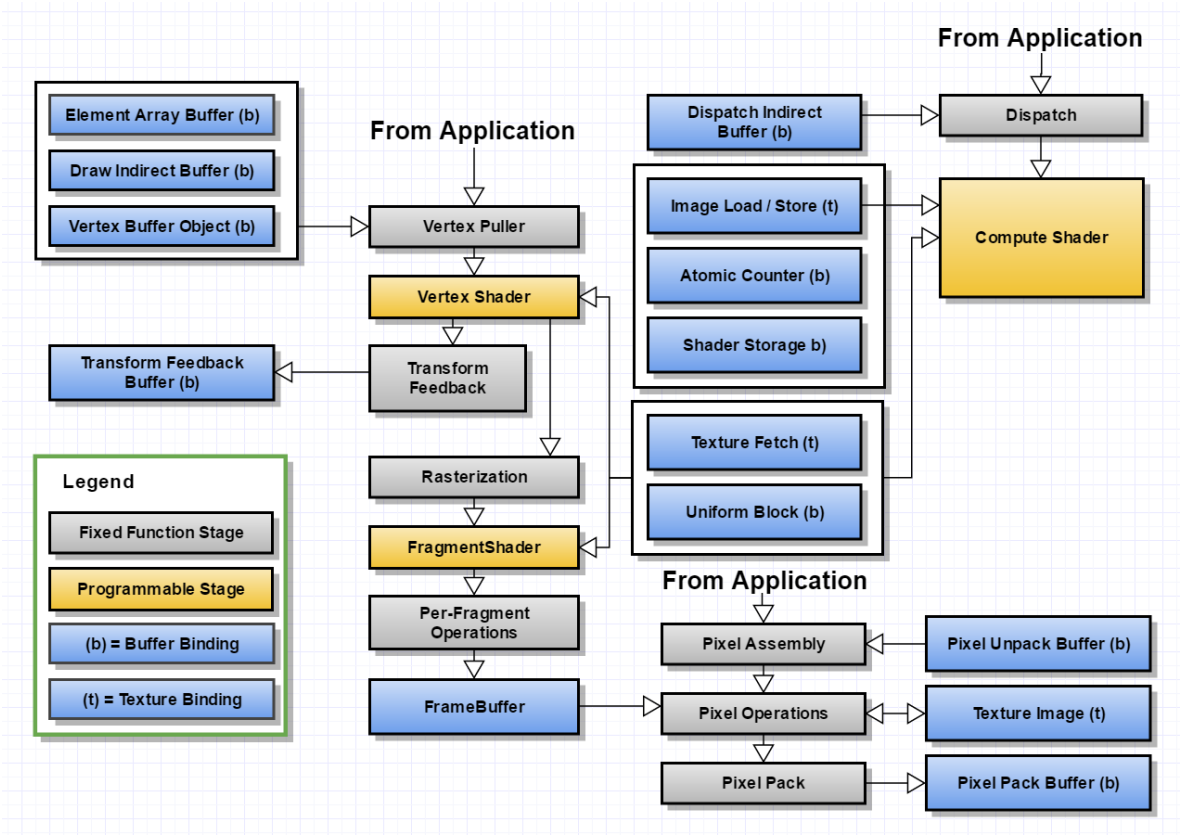


Kuvio 11. OpenGL 1.5 version laatikkodiagrammi **[42]**

Vuonna 2007 julkaistiin OpenGL ES versio 2.0. Se toi mukanaan ohjelmoitavat kulmapiste- ja fragmenttivarjostimet. Samana vuonna OpenGL ES:tä haarautettiin selaimelle tarkoitettu WebGL. Seuraava suuri virstanpylväs oli vuoden 2012 päivitys 3.0:aan. Päivitys toi mukanaan 32 bittiset kokonaisluvut, sekä float arvot, NPOT (Non-Power-Of-Two) tekstuurit, 3D-syvyystekstuurit, tekstuurilistat, sekä usean renderöintikohteiden tekniikan (multiple render targets). **[43]** Samasta versiosta haarautettiin myös selaimille tarkoitettu WebGL 2.0. Kaksi vuotta myöhemmin vuonna 2014 ajuripäivityksen myötä mukaan implementoitiin laskennalliset varjostimet (compute shaders). Nykyisin suurin osa OpenGL ominaisuuksista toimii Embedded Systemissä suoraan, mutta joitain operaatioita on karsittu pois. Kuvio 12 kuvaa OpenGL ES version 3.1 laatikkodiagrammin. Kuvioita 11. ja 12. vertaamalla voi nähdä millaisia uudistuksia OpenGL ES:n on tehty vuosien saatossa. **[33]**

Suurimpia muutoksia OpenGL ja OpenGL ES:n välillä on komentojen kerryttämisen poistaminen näyttölistasta jälkiprosessointia varten, sekä grafiikkaliukuhihnan ensimmäisen asteen käyrien ja pintageometrian approksimoinnin poistaminen. **[44]** OpenGL ES ohjelmat toimivat myös OpenGL alustalla. OpenGL tuoteperheen omistaa voittoa tavoittelematon konsortio Khronos Group.

Khronos group on kehittänyt OpenGL tuoteperheen rinnalle uutta API:a nimeltä Vulkan. Sitä on rakennettu AMD:n lahjoittaman Mantle renderöintiapplikaation pohjalta ja sen on tarkoitus muodostaa universaali standardi mikä toimisi samalla tavalla kaikissa päätelaitteissa.



Kuvio 12. OpenGL ES version 3.1 laatikkodiagrammi. **[33]**

Vulkanin tulevaisuudesta ollaan vielä epävarmoja. Jotkut lähteet väittävät, että Vulkan tulee elämään rinnakkaiseloa OpenGL rajapinnan kanssa tarjoten tehokkaamman, mutta samalla haasteellisemman rajapinnan. **[43]** Toiset lähteet ennustavat OpenGL rajapinnan hiljaista hiipumista pois käytöstä. Vielä on liian aikaista spekuloida mitä Vulkan aiheuttaa grafiikkarajapiireille, mutta oletettavasti ainakin muutaman vuoden verran Vulkan ja OpenGL tulevat elämään rinnakkain. Rinnakkaiselon aikana tarkastellaan rajapintojen suosiota ja harkitaan OpenGL tuoteperheen syrjäyttämistä. Vulkanin tarjoamia etuja on muun muassa yksinkertaistetummat ajurit, resurssinhallinta aplikaatiokoodissa, komentobufferit jne. **[45]** Useat prosessit, mitkä OpenGL ja OpenGL ES käsittelee korkean tason ajuriabstraktiossa siirtyy Vulkanissa applikaatiopuolelle. Tällöin ne ovat käyttäjän kustomoitavissa, eivätkä vie ajuri rajapinnassa tilaa. Vulkanin yhdistetty API rajapinta toimii niin pöytäkoneissa ja konsoleissa kuin myös mobiililaitteissa, jolloin yksi versio tulee toimimaan kaikkialla. Vulkanin ensimmäinen versio julkaistiin 16.02.2016, eikä kovin moni peli vielä tue sitä. Unity muok-

kaa tällä hetkellä mobiilisovellukset vielä OpenGL ES muotoon. Tämä liukuhihna käydään tarkemmin läpi luvussa 2.6 Unity - OpenGL ES pipeline.

Unity 5.4 tukee kolmea erilaista OpenGL ES versiota. OpenGL ES 2.0, OpenGL ES 3.0, sekä OpenGL ES 3.1. Aiemmin mukana ollut OpenGL ES 1.1 on jätetty pois sen vanhentuneen kirjaston vuoksi. Version 1.1 poistaminen valinnoista oli isohko askel, sillä samalla kumottiin takautuva yhteensopivuus vanhempiin laitteisiin Fixed Function renderöintiliukuhihnan poistamisen myötä. Staattisen liukuhihnan korvasi täysin ohjelmoitava liukuhihna (Fully Programmable Pipeline). Uutuutena 2.0 toi myös yhteensopivuuden työpöytäkäytössä olevaan OpenGL 4.1:n ja siitä uudempiin versioihin. FPP, eli täysin ohjelmoitava liukuhihna tarkoittaa sitä että liukuhihnan tiettyjä työvaiheita on mahdollista muokata lennossa. **[46]** Varjostimen muutettavat kohdat näkyvät kuviossa 12. keltaisella. Muokattavia kohtia on siis verteksivarjostin (vertex shader), fragmenttivarjostin (fragment shader) sekä laskennallinen varjostin (compute shader). Ohjelmoitavan liukuhihnan ansiosta Unityllä voidaan kirjoittaa täysin omaan käyttöön kustomoituja varjostimia, mikä helpottaa laskentaprosessia, kun varjostintyö voidaan hoitaa yhdellä ainoalla syklillä.

2.6 Unity - OpenGL ES liukuhihna

Unity hyödyntää 3D-grafiikan varjostinoperaatioiden laskemiseen varianttia korkean tason varjostinkielestä (High Level Shading Language). Tätä kieli kulkee nimellä Shader Lab. **[47]** Varsinainen varjostinkoodi kirjoitetaan *CGPROGRAM* pätkien sisään. Korkean tason varjostinkieli on käyttäjäystävällinen versio grafiikkakortin ominaisuuksista. Sen avulla on mahdollista kirjoittaa varjostimia ilman erillistä grafiikkakortin toimintaan sijoittuvaa tietoutta. Varjostinkoodi käännetään ja optimoidaan tukemaan tarvittavaa grafiikkarajapintaa. Unity osaa tehdä tämän kaiken automaattisesti, joten mikäli haluaa jotain nopeasti valmiiksi tai testattavaksi, ei varjostimiin tarvitse sen enempää perehtyä. Varjostimet toimivat käyttäjälle työkaluina, joita voi lisätä skeneen tietämättä sen enempää sen varsinaista toiminnallisuutta. Jokainen objekti pelissä voi käyttää joko samaa varjostinta, tai vastaavasti jokaisella objektilla voi olla oma tietynlaiseen lopputulokseen räätälöity varjostin. Unityssä varjostin on kiinteästi yhteydessä kappaleen materiaaliin. **[44]**

Mikäli yksikään Unityn tarjoamista varjostimista ei tee haluttuja toimintoja, voi varjostimen kirjoittamisen tehdä myös itse. Unityssä on käytössä pintavarjostin (Surface shader), kulmapistevarjostin (Vertex shader), pikselivarjostin (Fragment shader), sekä kiinteän toiminnan varjostin (Fixed function shader). Pintavarjostin on tarkoitettu pääsääntöisesti valaistuksen ja varjostuksen laskemiseen. Se toimii korkeamman abstraktiotason kanssakäymisellä Unityn valaistusliukuhihnan kanssa. Useimmat näistä tukee automaattisesti suoraa, sekä jaksotettua valaistuspolkua. Mikäli varjostimen päätoiminen tarkoitus on muokata 3D-mallin topologiaa, tulee käyttää kulmapiste- ja fragmenttivarjostintyyppiä. Nimensä mukaisesti muokattavissa on silloin kappaleen topologia, sekä pintakuviot. Varjostimessa kappaleiden laskenta tapahtuu kahdessa eri vaiheessa, kolmioiden muokkaamisvaiheessa sekä fragmenttien laskemisvaiheessa. Viimeinen varjostin on kiinteän toiminnallisuuden varjostin. Se on vanhentuneella teknologialla toimiva varjostin, eikä tarjoa tukea sen eri osa-alueiden muokkaamiseen. Kiinteän toiminnallisuuden varjostin oli käytössä OpenGl ES 1.0 versiossa mutta nyttemmin suositellaan käyttämään jotain muuta varjostinta kappaleiden renderöimiseen. **[47]**

Varjostimen kokoaminen tapahtuu osissa ja on mahdollista, että ajettavassa pelissä ei koskaan tarvita kaikkia varjostimen ominaisuuksia. Varjostin sisältää siis lukuisia eri variantteja, joita voidaan hakea yksitellen tarpeen mukaan käyttöön. Varjostimen kääntäminen osissa nopeuttaa grafiikan prosessoimista, kun mahdollisesti todella massiivisia varjostimia ei tarvitse kääntää kokonaisuudessaan laitteille. **[48]**

Unityn varjostimien kääntäminen tapahtuu taustaprosessilla nimeltä UnityShaderCompiler. Tämä prosessi käynnistyy Unityssä vasta siinä vaiheessa, kun varjostimia pitää kääntää. Useampia prosesseja voi käynnistää rinnakkain toimivaksi, mutta jokainen prosessi tarvitsee oman CPU ytimen toimiakseen. Varjostimen kaikki variantit tallennetaan Unityssä kansioon Library/ShaderCache ja täysin identtiset varjostimet käyttävät niiden viimeksi käännettyjä versioita. Jos varjostimia muuttaa paljon, täyttyy tämä kansio turhasta datasta. Unityn dokumentaatiossa suositellaan varjostinkansion tiedostojen poistamista vähän väliä, sillä silloin vähennetään turhan datan varastointia. Pahimmillaan kansion datan poistaminen aiheuttaa vain varjostinvarianttien uudelleenkääntämisen. **[49]**

Varjostinohjelmat ovat kiinteästi yhteydessä Unityn tarjoamiin renderöintipolkuihin. Polkuja on kolme erilaista ja niitä voidaan vaihtaa lennosta riippuen päätelaitteen suorituskyvystä ja

asetetuista oletuspoluista riippuen. Renderöintipolut ovat verteksivalaistus (Vertex Lit), suora valaistus (Forward rendering), sekä jaksotettu valaistus (Deferred rendering).

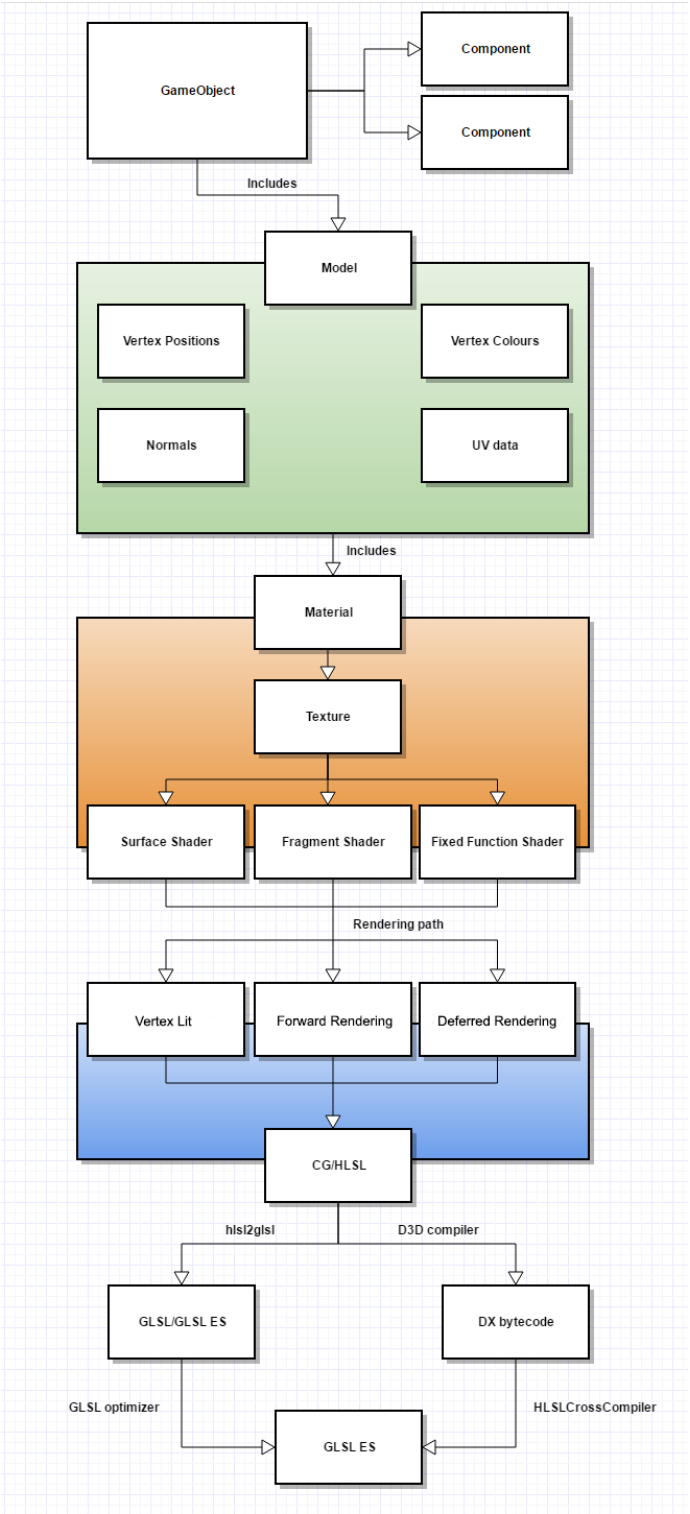
Verteksivalaistus on näistä poluista yksinkertaisin, eikä se tue esimerkiksi reaaliaikaista varjojen laskentaa ollenkaan. Verteksivalaistus laskee valaistuksen määrän yhdellä läpiviennillä (pass) kaikille valoille. Laskenta tapahtuu verteksikohtaisesti, joten pikselikohtaista valaistusta ei tueta. Edukkaana puolena suorassa valaistuslaskennassa on mahdollinen yhden kuvan tuotto yhdellä läpiviennillä. Tämä vaatii tosin valaistuksen karsimisen yhteen universaaliin valoon ja loppujen valojen esilaskemiseen (bake). Mikäli valoja on enemmän, nostaa jokainen valo renderöintiä yhdellä läpiviennillä erillisen valaistuslaskennan puuttuessa. **[50]**

Oletusarvoisesti Unity käyttää suoraa valaistuspolkua. Valaistus voidaan laskea yhdessä tai useammassa läpiviennissä riippuen täysin valoista, jotka vaikuttavat piirrettävään objektiin. Pikselivalojen määrää voi muokata asetuksista vastaamaan haluttua visuaalista ulkoasua. Pikselivalaistus on kuitenkin laskennallisesti monimutkaisempaa verteksivalaistuksen laskemiseen verrattuna, joten esimerkiksi mobiililaitteiden kanssa sen käyttöä tulee harkita. Pikselivalaistuksen lisäksi voidaan laskea neljä pistevalaistusta verteksikohtaisesti. Valoille voi määrittää missä tärkeysjärjestyksessä valot lasketaan. Tärkeät valonlähteet lasketaan aina pikselikohtaisesti, toissijaiset verteksikohtaisesti. **[52]**

Viimeisin renderöintipolku on jaksotettu valaistuspolku. Se on myös laskennallisesti raskain valaistuspolku, sillä kaikki valaistukset lasketaan pikselikohtaisesti, eikä valonlähteiden lukumäärää ole rajoitettu. Deferred renderöintipolussa renderöinti tapahtuu kahdella läpiviennillä. Ensimmäinen on G-buffer läpivienti, missä peliobjektit renderöidään tuottamaan ruutuavaruuden puskurit diffuusivalaistukselle, spekulääriva laistukselle, siloituksille, maailman avaruuden normaaleille, emissiolle sekä syvyydelle. Saatuja puskureita käytetään toisessa läpiviennissä valaistuksen laskentaan emissiopuskuriin. **[52]**

Varjostinfunktioiden ja renderöintipolun selvittyä data on valmis vietäväksi päätelaitteelle. Korkean tason varjostinkieli pitää vielä kääntää halutussa laitteessa toimivaan muotoon. Unity käyttää kahta eri polkua muuttaessaan korkean tason varjostimet OpenGL ES muotoon. Vanhemmassa menetelmässä High Level Shading Language (HLSL) muutetaan GLSL/GLSL ES muotoon käyttäen hlsl2glsl kääntäjää. Käännetty GLSL/GLSL ES ei kuitenkaan ole opti-

moitu, joten tämän jälkeen ajetaan GLSL optimizer, joka poistaa generoiduista varjostimista turhat rivit, ylimääräiset kopiointioperaatiot, koodin suppeuttamisen (code folding) jne... Tämä menetelmä ei kuitenkaan tue varjostinmalli (Shader Model) 3.0:aa uudempia malleja. Varjostinmallit ovat komponentteja, jotka auttavat CPU:n lähettämän grafiikan renderöimisessä grafiikkaprosessorilla. Varjostinmallit sisältävät pikselivarjostimen, sekä kulmapistevarjostimen. **[31]** Varjostinmalli 3.0:n rajoite tarkoittaa, ettei myöskään DX9:ää uudempia ominaisuuksia voida hyödyntää. Jotta voitaisiin kääntää varjostimia, jotka tukisivat DX11 asti olevia varjostimia, tarvitaan toisenlainen liukuhihna. Tämä liukuhihna muuttaa korkean tason varjostinkielen DX bytekoodiksi Microsoftin omalla D3D-kääntäjällä. Tämän jälkeen DX bytekoodi käännetään GLSL/GLSL ES muotoon kustomoidulla HLSL CrossCompilerilla. Koko liukuhihna on karkeasti kuvattuna kuviossa 13. **[49]**



Kuvio 13. Unity HLSL - OpenGL ES liukuhihna.

3 Case Gravitoid

Käytännön työnä tässä Pro gradu tutkielmassa pyritään optimoimaan Endless Tea Studiosin esikoispeliä Gravitoidia. Gravitoid on mobiilialustalle julkaistava fysiikkapuzzlepeli. Pelissä ohjataan yksin avaruuteen eksynyttä pelaajaa maaliin gravitaatiokenttiä hyödyntäen. Hyppyjen määrää ei ole pelissä rajoitettu, mutta niiden määrästä pidetään kirjaa ja jotkut pelin tehtävät vaativat tietyn hyppymäärän alittamisen. Pelaajalle tuodaan ongelmanratkontaelementtejä erilaisten peliobjektien avulla. Optimointikysymykset otettiin huomioon jo pelin kehitysvaiheissa, mutta optimointia on silti mahdollista suorittaa lukuisissa eri kohteissa. Testattavat tilanteet irroitetaan Gravitoid pelistä siten, ettei skriptaukset vaikuta ruudunpäivitysnopeuteen. Näin voidaan keskittyä graafisten ominaisuuksien mittaamiseen ja optimointiin. Testit jaetaan kolmeen kategoriaan, jotka jakaantuvat erilaisiin testiskeneihin. Ensimmäinen testikategoria on kolmioiden määrän testaus. Toinen testitilanne testaa valojen vaikutusta ruudunpäivitysnopeuteen ja viimesessä testitilanteessa testataan varjostimien vaikutusta ruudunpäivitysnopeuteen. Mittaustulokset analysoidaan Unityn Profiler työkalulla ja saaduista tuloksista tehdään tilastolliset johtopäätökset.

3.1 Gravitoid

Gravitoid on Endless Tea Studiosin ensimmäinen julkaistava mobiilipeli. Gravitoidissa yksinäinen astronautti pyrkii gravitaatiota hyödyntämällä hyppimään madonreikään ja sitä kautta seuraavaan kenttään. Gravitoidin kehittäminen alkoi Jyväskylän GameLabissa syksyllä 2015 ja on kestänyt syksyyn 2016 asti. Pelin kehitystiimi koostui aluksi kahdeksasta henkilöstä: kaksi koodaria, kolme graafikkoa, kenttäsuunnittelija, äänisuunnittelija sekä manageri. Sittemmin ryhmän lukumäärää pudotettiin neljän hengen tiiviiseen työryhmään. Gravitoidin pohjimmainen idea on tarjota pelaajalle fysiikkapainotteisia puzzle-elementtejä kenttien läpäisyn haasteeksi. Peli tarjoaa tällä hetkellä yli kolmekymmentä pelikenttää tutkittavaksi, sekä useita eri avaruuspukuja pelaajahahmolle. Pelaajalla on käytössään kaksi eri poweruppia helpottamaan kenttien läpäisyä: Rakettirepulla pelaaja voi vaihtaa lennosta suuntaa ja painovoimakoukulla voi pysyä planeettojen kiertoradalla niiden gravitaatiokentän ulkopuolella. Pelaajaa vastassa on myös lukuisat erityyppiset planeetat, sekä teleportaatioma-

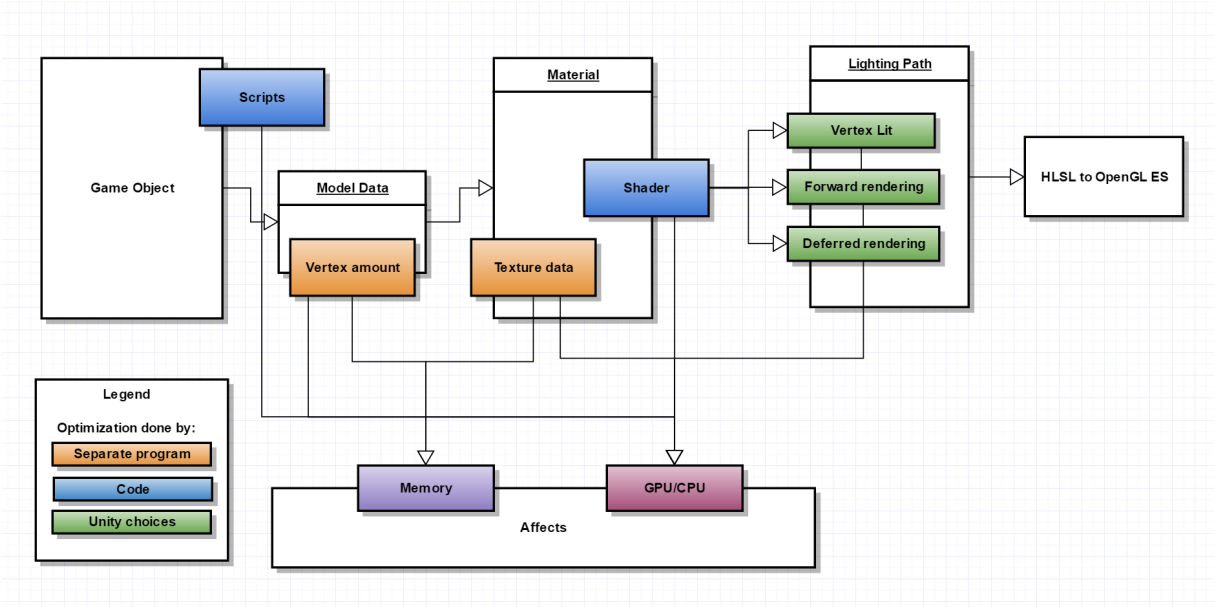
donreiät. Jokainen elementti tuo uudenlaisen haasteen kenttien läpäisyyn ja haastaa pelaajaa harkitsemaan hyppyreittejä. Gravitoidin graafinen ulkoasu on leikkisä vähäpolygoninen 3D ja hahmoa lukuunottamatta kenttien objektit ovat 3D-objekteja. 3D-kappaleet tuovat tarvetta optimoinnille, sillä vaikka mobiililaitteiden 3D-prosessointikyky on kasvanut tasaista tahtia, asettaa se silti rajoitukset sille mitä kaikkea ruudulla voidaan näyttää.

3.2 Mittausmenetelmät

Mittaukset suoritetaan erikseen mittausta varten tehdyillä Unity-skeneillä. Skenet on rakennettu siten, että se mittaa varjostimen ja valaistuspolkujen eri osa-alueita hallitussa ympäristössä. Testitilanteet pilkotaan eri skeneihin, jolloin saadaan helpommin luotettavaa dataa juuri halutusta testikohteesta. Ruudun vasemmassa laidassa näkyy juokseva ruudunpäivitys, kymmenen sekunnin keskivertoruudunpäivitysnopeus, sekä suurin ja pienin ruudunpäivitys testitilanteesta. Ylhäällä keskellä näkyy otsikkona testissä olevan skenen nimi. Ruudun oikeassa ja vasemmassa alanurkassa näkyvät nuolet seuraavaan ja edeltävään testitilanteeseen. Kuvio 22 kuvastaa testitilanteiden ulkonäköä. Ensimmäisenä testataan kolmioiden määrän ja instanssoinnin vaikutusta suorituskykyyn. Tämän jälkeen katsotaan valaistuspolkujen vaikutusta. Valaistuksissa varjojen heijastaminen jätetään pois, sillä Gravitoidiin ei tule varjostuksia. Polygonmäärien ja valaistuspolun testaamisen jälkeen testataan varsinaisia varjostinohjelmia keskenään. Testeissä käydään läpi Unityn tarjoaman mobiilivarjostimen ja oletusvarjostimen toimintaa, sekä erikseen Gravitoidia varten tehtyjen kustomvarjostimen ja taustavarjostimen toimintaa. Gravitoid ei sisällä juuri yhtään tekstuuritietoja, joten niiden optimointia käsitellään ilman käytännön testejä. Unityssä on mahdollista asettaa eri kameroille eri valaistuspolkuja. Valaistuspolut on käyty läpi luvussa 2.6 Unity - OpenGL ES liukuhihna. Testiskeneissä pelikuva lasketaan kolmen eri kameran avulla. Pääkamera kuvaa pelitilannetta ruudun etuosassa. Taustalle jäävät kappaleet piirretään kahden kameran turvin mistä toisen tehtävä on piirtää pelkkä taustataivas ja toisen taustaobjektien piirtäminen. Kaikki kamerat käyttävät jokaisessa testitilanteessa samaa valaistuspolkua.

Optimointimahdollisuudet sijaitsevat grafiikkaliukuhihnan eri vaiheissa ja vaikuttavat mobiililaitteen eri osa-alueiden toimintaan. Esimerkiksi valaistuksen laskeminen kuormittaa lähes yksinomaa grafiikkaprosessoria, kun taas suuret tekstuurikoot kuormittavat tämän lisäksi

paljon muistia. Jotkut optimointikohteet on hoidettava optimoiduilla skripteillä tai varjostimilla ja jotkut optimointikohteet taas vaativat ulkopuolisen ohjelman. Kuvio 14 esittää yksinkertaistetun version grafiikkaliukuhihnan toiminnasta optimointikohteiden näkökulmasta. Ohjelmoitavissa olevat optimointikohteet rajoittuvat kappaleeseen kiinnitettyjen skriptien ja varjostimen muokkaamiseen. Varjostimissa ohjelmoitavat kohdat näkyvät kuviossa 12. Kohdat ovat verteksivarjostin, fragmenttivarjostin, sekä laskennallinen varjostin, jotka käyttävät valittua valaistuspolkua kappaleiden esittämisessä. Kolmioiden ja tekstuuridatan optimointi tapahtuu erillisissä ohjelmissa. Unityn sisällä 3D-dataa voi optimoida tarkkuustasoilla ja tekstuureilta Mip kartoituksen avulla (mipmapping).



Kuvio 14. Optimointiriippuvuusdiagrammi Unity - OpenGL ES liukuhihnasta

Testidata kerätään Unity Profiler työkalun avulla. **[53]** Profiler toimii siten, että testipeli käännetään ja ajetaan testilaitteen ollessa kiinni tietokoneessa USB:lla. Ennen kääntämistä valitaan kääntäjän asetuksista kehitysversio sekä Profilerin automaattinen yhdistäminen päälle. Tämän jälkeen testiskenet käännetään suoraan puhelimeen. Kääntämisen valmistuessa APK käynnistyy automaattisesti testilaitteessa ja Profiler aukenee Unityn editorinäkymään. Editorissa toimiva Profiler ei tallenna testidataa, vaan poistaa kaiken mikä ei mahdu aikajanalle. Janalle tallentuu 300 otantaa. **[06]** Unity ei tarjoa sisäänrakennettua mahdollisuutta lukea tallennettua dataa, joten testitilanteessa data luetaan suoraan profilerista. Profiler käyttää luonnollisesti resursseja tietojen siirtämiseen ja esittämiseen, joten se tulee

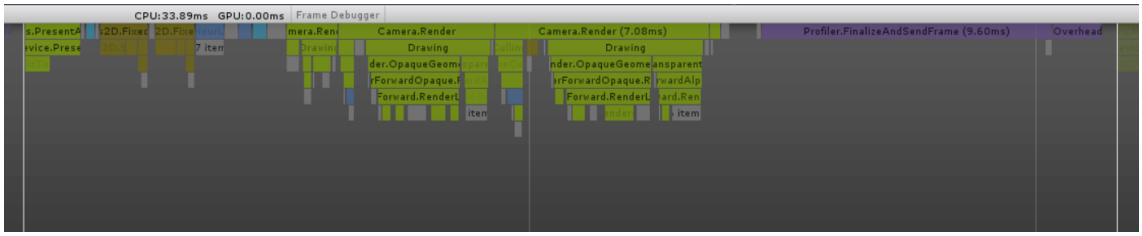
huomioida testituloksia tarkasteltaessa. Tietokoneella testattaessa resurssikustannukset ovat pienemmät testilaitteiston suuremman tehon myötä. Mobiililaitetta testattaessa resurssikustannus näkyy huomattavasti selkeämmin. Profilerin aiheuttaman kustannuksen selvittämiseksi jokaisen testiskenen vasemmassa ylälaidassa on ruudunpäivitysnopeutta tarkkailevat arvot. Ylin arvo ilmoittaa senhetkisen ruudunpäivitysnopeuden ja toiseksi ylin mittaa kymmenen sekunnin keskiarvon ruudunpäivitysnopeudesta. Alimmat kaksi mittaavat maksimi- ja minimilukuja ruudunpäivitysnopeuksista. Profilerin ollessa pois päältä, testilaitteiden keskimääräiseksi ruudunpäivitysnopeudeksi instanssoimattomalla asteroidikentällä tuli 29 ruutua sekunnissa, mikä on vielä ihan pelattava ruudunpäivitys mobiililaitteelta. Kun Profiler laitetaan päälle, tippui keskimääräinen ruudunpäivitysnopeus 21.4. ruutuun sekunnissa. Profiler tiputtaa mobiililaitteella testattaessa siis noin 30 % laskentatehoa pois, mikä on kohtuullisen paljon. Tämä tulee huomioida lopputuloksia tarkastellessa.

Profiler tarjoaa hyvin paljon erilaista tarkasteltavaa dataa aina muistin käytöstä eri prosessorisäikeiden aikaleimoihin. Koska suurin osa Profilerin datasta on testitilannetta katsoen toissijaista, valitaan kirjattavaksi muutama tärkeä kohde ja tarkastellaan testiympäristöä niiden avulla. Tallennettavaksi testidataksi valittiin seuraavat kohdat:

- Ruudun piirtoaika: ms
- Kameran piirtoaika: ms
- Ruudunpäivitysnopeus: fps
- Läpivientikutsut (Set Pass Calls): kpl
- Piirtokutsujen määrä: kpl
- Kolmioiden määrä: kpl
- Kärkipisteiden määrä: kpl
- Käytetyn muistin määrä: Mb
- Skenen objektien määrä: kpl

Ruudun piirtoaika tarkoittaa sitä aikaa, mikä mitattavalla ruudulla kestää piirtyä näytölle. Kameran piirtoaika tarkoittaa skenessä olevien kolmen kameran kuvan laskemiseen mennyttä aikaa. Kameran piirtoaika on skenestä riippuen noin 12 - 60 % koko ruudun piirtoajasta. Muuta laskemista skenessä on esimerkiksi fysiikoiden laskeminen, grafiikoiden synkkaus ja lähettäminen laitteen näytölle piirrettäväksi, käytöspäivitykset (behaviour update) ja

niin edelleen. Kuvion 15. kuvakaappaus kertoo testitilanteessa Profilerin aikajanalla näkyvän CPU:n yhden kuvan piirtämisessä käytetyt resurssit ja niiden kestot. Kuvasta huomaa mitä kaikkea kukin kolmesta kamerasta renderöi ja kauanko renderöinneissä menee. Vaikka kuvan tilanteessa kameroiden piirtämiseen ei mennyt kuin 13,93 millisekuntia, niin lopullinen piirtoaika oli 36 millisekuntia. Kuvan finalisointiin meni 9,60 millisekuntia ja loput 12,47 millisekuntia meni muihin erinäisiin tehtäviin, kuten fysiikkaan, partikkeleiden päivittämiseen, käyttäytymispäivityksiin jne.



Kuvio 15. Esimerkkikaappaus Profilerin aikajanasta.

Ruudunpäivitysnopeus on sen hetkinen arvio, montako ruutua piirtyy sekunnissa kyseisellä piirtoajalla. Läpivientikutsut (Set Pass Call) kertovat montako kertaa renderöitäessä pitää sitoa uusi varjostin renderöintiin. Tämä eroaa komentojonokutsuista (Batch Calls) sillä, että yksi komentojonokutsu on johonkin materiaaliin sidottu piirtokutsu. Unity-pelimoottori pyrkii kasaamaan useiden kappaleiden renderöimisen yhdeksi kappaleeksi muistiin, jolloin niitä ei tarvitse hakea useasti. Silloin samaa materiaalia käyttävät kappaleet voidaan hoitaa yhdellä komentojonokutsulla. Materiaali voi sisältää useita läpivientejä, jolloin tarvitaan läpivientikutsu. Läpivientikutsut taas sitovat uuden varjostimen käyttöön. **[54]** Piirtokutsu tehdään siinä vaiheessa, kun jokin kappale pitää piirtää ruudulle. Silloin CPU tekee piirtokutsun grafiikkaprosessorille, joka laskee kappaleen ruudulle. Kolmioiden määrä tarkoittaa skenessä olevien 3D-objektien kolmioiden yhteenlaskettua määrää. Kärkipisteet ovat edellä mainittujen kolmioiden kärkipisteiden määrä. Käytetyn muistin määrällä ja skenen objektien määrät ovat yksiselitteisiä.

Kirjattavista tiedoista staattisia arvoja ovat skenekohtaiset objektit, kolmioiden sekä kärkipisteiden määrä. Vähäistä vaihtelua ilmenee läpivientikutsuissa, sekä piirtokutsuissa. Muuttuvia arvoja ovat kameroiden piirtoaika, yhden ruudun renderöintiaika, ruudunpäivitysnopeus, sekä käytetyn muistin yhteenlaskettu määrä. Ruudun piirtoajan pystyy Profilerissa tarkastele-

maan kamerakohtaisesti, missä huomataan, mikä kamera käyttää eniten aikaa omien osiensa piirtämiseen. Kameroiden erillistä ruudunpiirtoaikaa ei testitilanteessa kirjata erikseen, vaan kaikkien kameroiden käyttämä yhteispiirtotyö merkitään, jolloin ne voidaan vähentää tarvittaessa koko ruudun piirtoajasta. Tällöin voidaan tarkastella paljonko aikaa menee varsinaisen grafiikan piirtämiseen ja paljonko jää muuhun laskennalliseen toimintaan, kuten synkkaukseen, animaatioiden tilojen tarkasteluun, fysiikoiden laskemiseen jne. Testidataa tulee kaksikymmentä otantaa per testattava mobiililaite ja skene. Ennen mittausten aloittamista skene pyörii testilaitteessa noin 10 minuuttia. Otantaa varten Profilerin tallentaminen pysäytetään ja kaksikymmentä ruutua valitaan Profilerin näyttämästä kolmestasadasta ruudusta. Koska testitilanteen skenet ovat hyvin passiivisia, ei ruudunpäivitysnopeudessa oletettavasti tapahdu suuria muutoksia ja näin testiotannasta saadaan kohtuullisen tasaista.

Testilaitteina toimii OnePlus 2 älypuhelin, sekä Samsung Galaxy Note 10.1 tablettitietokone. Molemmat laitteet edustavat markkinoilla olevien mobiililaitteiden tehokkaampaa päätä eikä varsinaista budjettilaitetta ole testitilanteessa ollenkaan. Taulukko 1 esittää One Plus 2:n sekä Samsung Galaxy Noten laitteistojen eroavaisuudet.

Specs	OnePlus 2	Samsung Galaxy Note 10.1
Piirisarja:	Qualcomm MSM8994 Snapdragon 810	Qualcomm Snapdragon 800
Suoritin:	Octa-core	Quad-core 2.3 GHz Krait 400
Ytimet:	4x1.56 GHz Cortex-A53	4x 2.3 GHz
	4x1.82 GHz Cortex-A57	
GPU:	Adreno 430	Adreno 330
Muisti:	4GB RAM	3GP RAM
Tallennustila:	64GB	16GB
Resoluutio:	1080 x 1920	2560 x 1600
Android versio:	6.0.1 (Marshmallow)	5.1.1 (Lollipop)

Taulukko 1. OnePlus 2:n sekä Samsung Galaxy Noten laitteistotiedot

3.2.1 Polygonien määrän testaus

Polygonien määrän testiskeneissä näkyy normaalin pelimaailman taustalla planeetta ja asteroidikenttä. Asteroidikenttä on kolmion määrän testauksessa keskeisin osa ja se on tuotu suoraan Blender ohjelmasta yhtenä omana kappaleenaan. Kolmion määrää testataan kolmella eri skenellä. Ensimmäinen skene testaa optimoimatonta asteroidikenttää. Siinä siemenasteroideista muodostettu asteroidikenttä muutetaan yhdeksi ainoaksi kappaleeksi ja tuodaan ilman erillistä optimointia Unity-pelimoottoriin. Yhtenä kappaleena tuotuna asteroidikenttä käyttää turhaan testilaitteiden muistia saman asteroidin kopioilla. Toinen skene on optimoitu versio asteroidikentästä, jossa kolmioiden määrä on pudotettu puoleen. Viimeisessä skenessä asrteroidikentän asteroidit on instanssoitu siten, että kahdesta siemenasteroidista muodostetaan optimoimattoman asteroidikentän tyyppinen asteroidikenttä kopioimalla siemenasteroidit Unityssä. Tällöin muistiin tarvitsee varata ainoastaan kahden siemenasteroidin tiedot, lokaatiot, rotaatiot ja skaalaukset kopioille. Instanssoinnin pitäisi pienentää muistin käyttöä.

3.2.2 Valaistuspolun valinta

Yksi laskennallisesti työläimmistä prosesseista varjostimesta riippuen on reaaliaikainen valojen laskeminen pelinäkymään. Mikäli käytetään verteksivalaistuspolkua usean eri valonlähteen kanssa voi sen yksinkertaisemman varjostimen tuoma hyöty valua täysin hukkaan valaistuslskennan tehdessä jokaisen valonlähteen kohdalla uuden läpiviennin. Testattavaksi otetaan Unityn tarjoamat kolme eri valaistuspolkua: verteksivalaistuspolku, suora valaistuspolku sekä jaksotettu valaistuspolku. Jokainen valaistuspolku testataan omassa yksittäisessä testiskenessä. Jokainen skene on identtinen valaistuspolkua lukuunottamatta. verteksivalaistuspolkua ei suositella käytettävän enää uusien pelien tekemisessä sen vanhentuneen ja suoraviivaisemman valaistuksen laskemisensa takia, mutta se pidetään mukana hyvänä vertailukohtana visuaalisten erojen määrästä. Valaistuspolkuja testattaessa jokainen näkymän kolmesta kamerasta muutetaan käyttämään kyseistä valaistuspolkua. Näin voidaan varmistaa, ettei esimerkiksi pelikameran valaistuspolku sotke testitulosta sen käyttäessä erilaista menetelmää valaistuksen laskemisessa.

Testiskeneissä ruudulla näkyy avaruusaluksen kappale, jota valaisee pääosin kaksi 35:stä kei-

lamaisesta valonlähteestä koostuvaa valotelinettä. Näiden lisäksi kappaletta valaisee kuusi efektityyppistä valoa, joista neljä on keilamaisia ja kaksi pistemäisiä valonlähteitä. Viimeisenä valona toimii yksi kohdistettu valo. Kohdistetun valon tarkoitus on valaista kaikki muut ympärillä olevat kappaleet luoden yleisen ambientvalaistuksen pelinäkymälle.

3.2.3 Varjostimien testaus

Testattavia varjostimia on neljä erilaista. Jokainen varjostin testataan omassa testiympäristössä valaistuspolkujen testauksen tyyppisesti. Testiskenessä hallitsevana osana on kolmioiden testauksessa aikaansaatu asteroidikentän optimoitu 3D-malli. Jokainen testiskenen objekti on muutettu käyttämään testattavaa varjostinohjelmaa yhdessä suoran valaistuspolun kanssa. Suora valaistuspolku valittiin siksi, että se tulee olemaan lopullisessakin tuotoksessa käytettävä valaistuspolku ja se antoi testiympäristön valoilla parhaan visuaalisen tuloksen. Testiympäristön valaistuksesta vastaa viisi erilaista valoa: Pääsuuntavalo valaisee pelaajahahmon planeetan, efektivalo antaa väriefektin pelaajan asuttamalle planeetalle. Asteroidikentän ja planeetan valaistuksesta vastaa kolmen keilavalon ryhmä. Päällisvalo vastaa yleisestä asteroidien valosta, aurinkoefektivalo antaa auringon suunnasta tulevan valoefektin ja kenttäefektivalo valaisee asteroidikentän etuosaa tuoden esille heijastuksen vaikutusta valitussa varjostimessa.

Ensimmäinen testattava varjostin on Unityn tarjoama oletusvarjostin. Oletusvarjostin on mukana antamassa referenssiä suorituskyvystä ilman minkäänlaista optimointia. Seuraavaksi testataan Unityn tarjoamaa mobiililaitteille optimoitua varjostinta mobiilivarjostin skenellä. Mobiilivarjostin on huomattavasti kevyempi, kuin oletusvarjostimena toimiva varjostin. Mobiilivarjostimen avulla saadaan tietoa siitä, miten paljon suorituskyky kasvaa Unityn tarjoamien työkalujen avulla. Mobiilivarjostin on todella lyhyeksi kirjoitettu ja sisältää vain tekstuuritietojen siirtämisen kappaleiden pinnalle. Tästä johtuen etuosassa olevalle planeetalle ja tausta-asteroideille piti tehdä oma yhden värin tekstuurit, jotka piirretään kappaleiden pinnalle. Tekstuuritiedostot ovat kokoa 10 * 10 pikseliä ja sisältävät vain halutun värin. Mikäli tarvittavia värejä olisi enemmän, kannattaisi väreistä koostaa tekstuuriatlas mistä halutut värit haettaisiin. Testitilanteessa yhden lisätekstuurin hakeminen pelaajan asteroidille aiheuttaa yhden ainoan lisäkutsun, joka ei ole mainittava kuormitus. Jos eri tekstuureja käyttäviä

planeettoja olisi enemmän, olisi tekstuuriatlaksen tekeminen järkevää. Mobiilivarjostimen huono puoli on juuri äärimmilleen karsittu varjostinkoodi. Koska varjostimelle voi antaa ainoastaa tekstuuritiedoston, pitää jokaisella peliobjektilla olla omat tekstuurit. Normaalisti tämä ei tulisi olemaan ongelma, sillä 3D-mallit teksturoidaan lähes kaikissa peleissä. Gravitoidin tapauksessa taas jokaiselle kappaleelle pitää tehdä tekstuurit testitilannetta varten, mikä on työlästä. Mikäli jonkin kappaleen väri ei ole haluttu, pitää tekstuurit tehdä erikseen. Tämä aiheuttaisi ylimääräistä työtä.

Kolmas testattava varjostin on varsinaiselle pelinäkymälle tehty mobiiliystävällinen varjostin. Tämä kustomoitu varjostin on myös ensimmäinen erikseen Gravitoidia varten tehty varjostin. Kustomoitua varjostinta suunniteltaessa tavoitteena oli tehdä siitä mahdollisimman kevyt pitäen kuitenkin tarvittavat ominaisuudet mukana. Varjostimen ohjelmoinnin erikoisin ratkaisu oli CG kirjaston ohittaminen kokonaan. Varjostin esitellään tarkemmin luvussa 3.3 Optimointi. Viimeinen testattava varjostin on taustakappaleilla käytössä oleva varjostin. Se eroaa kenttäobjekteissa käytettävästä varjostimesta siten, että se käyttää CG kirjastoa varjostusten laskemisessa ja värjää esitettävän kappaleen reunoja tarvittaessa halutun sävyiseksi. Reunavärjäyksen ansiosta kappaleet näyttävät sulautuvan taustaan paremmin, jolloin pelaajan huomio pysyy selkeämmin etualalla olevissa peliobjekteissa. Taustavarjostin on oletettavasti kustomoitua varjostinta ja Unityn mobiilivarjostinta raskaampi käyttää. Kuitenkin reunavalaistuksen tuoma visuaalinen lisä painaa vaakakupissa menetettyä ruudunpäivitystä enemmän. Taustavarjostimen oletetaan toimivan vähintään Unityn standardivarjostinta paremmin.

3.3 Optimointi

Optimointia voi lähestyä monelta eri kantilta. Järkevin tapa on mitata pelin toimintaa jollain työkalulla, tunnistaa pullonkauloja aiheuttavat tapahtumat ja optimoida niiden toimintaa. Joskus pullonkaulat voivat olla grafiikan laskemisesta johtuvia ja joskus esimerkiksi pelimekaniikkaan liittyvät toiminnat aiheuttavat ongelmia. Esimerkiksi Gravitoidissa yksi pullonkaulaa aiheuttava pelimekaniikkaan liittyvä kohta oli lentorataa ennustavan janan laskeminen. Hypyt lasketaan Unityn omalla fysiikkamoottorilla eikä siinä ole esilaskentaa, joten samaa dataa ei ole mahdollista käyttää lentoradan näyttämiseen ja varsinaiseen hyppyyn.

Lentorata pitää siis laskea useaan otteeseen ensin hyppyvektorin esittämiseksi ja myöhemmin pelaajan varsinaisen hypyn laskemiseksi. Lentoradan esilaskenta oli raskasta ja siksi se tiputti ruudunpäivitysnopeutta huomattavasti. Uuden lähestymistavan kautta ongelma saatiin poistettua ja hyppyvektori voidaan laskea nyt ilman halvaannuttavaa ruudunpäivitysnopeuden putoamista. Joskus pullonkaulat ovat graafisessaa esittämisessä. Liian monen objektin pyörittäminen etenkin monipuolisen valaistuksen kanssa asettaa rasitusta grafiikkapiirille. Tällöin grafiikkapuolen optimointi tulee tärkeäksi osaksi pelikehitystä. Grafiikan optimoinnissa hyviä käytänteitä löytyy monta. Oheinen lista on Unityn laatima optimointikohteiden tarkastuslista. Listasta löytyvät oikeastaan kaikki järkevät optimointikohteet grafiikan saralta: **[55]**

- Pidä kolmioiden määrä alle 200 000 per piirrettävä ruutu.
- Mikäli käytetään valmiita varjostimia, ne kannattaa valita mobiilikategoriasta niiden ollessa nopeampia ja valmiiksi optimoituja.
- Pyri pitämään skenekohtainen materiaalien määrä alhaisena, jolloin lataamisajat pysyisivät pienenä.
- Tekstuuriatlaksen käyttö vähentää piirtokutsujen määrää huomattavasti.
- Laita staattisille kappaleille "static"arvo editorissa, jolloin ne piirretään vain kerran.
- Käytä pikselivalaistusta vain tarvittaessa, sillä se syö paljon resursseja.
- Vältä dynaamisen valaistuksen käyttöä.
- Käytä kompressoituja tekstuureja ja pidä huoli, että niiden koko on 2:ssa potensseissa. (Unity osaa käyttää muitakin kokoja, mutta silloin tekstuurimuistia jää käyttämättä.)
- Vältä sumuefektin käyttöä.
- Katso mitä peittoon jäävän geometrian poiston (occlusion culling) avulla voi tehdä. Sillä voi vähentää näkyvää geometriaa ja vähentää kutsujen määrää huomattavasti.
- Piirrä kaukaiset objektit suoraan taivaskarttoihin (skybox).
- Käytä pikselivarjostimia tai tekstuurin yhdistelijöitä joilla yhdistää useita tekstuureja usean läpiviennin (multi-pass) lähestymisen sijaan.
- Käytä puolen tarkkuuden muutujia (half precision variable) aina kun mahdollista.
- Minimoi monimutkaisten matemaattisten operaatioiden kuten kerto, cosini jne. käyttämistä pikselivarjostuksessa.
- Minimoi fragmenttikohtaisten tekstuurien määrää.

3.3.1 Kolmioiden määrän vähentäminen

Peliskeneä on mahdollista optimoida monin eri tavoin. Yleisimpiä optimointikohteita kolmiulotteisessa pelissä on juuri kolmioiden määrä, vaikka nykyään mobiililaitteilla voidaan renderöidä jo kohtuullisen yksityiskohtaisia 3D-malleja. Kolmioiden määrän pudottaminen objekteista tuo mahdollisuuden käyttää useampaa samaa objektia pelinäkymässä ilman laitteen liiallista rasittamista. Otetaan esimerkiksi Gravitoidin Breaking Planet malli. Breaking Planet on epävakaan näköinen planeetta, joka hajoaa pienen viiveen jälkeen pelaajan laskeuduttua sen pinnalle. Breaking Planet mallinnettiin ensin ehjänä planeettana, joka hajotettiin Blenderin solun halkaisija työkalulla (Cell Fracture). Solun halkaisija algoritmi pilkkoo 3D-mallin annetuilla parametreillä pienempiin palasiin. Planeetan kuoren alle haluttiin toisen värinen materiaali, joten se määritettiin solun halkaisijan ominaisuuksista erilliseksi materiaaliksi. Kappaleen pinta ja sisäiset tasot jaettiin siis kokonaan eri kappaleiksi. Tämä aiheuttaa kappaleiden reunapisteiden tuplaantumisen. Kappaleen pilkkomista hoitava algoritmi ei myöskään katso yhtään kärkipisteiden sijaintia pilkkoessaan planeettaa, joten mallia tarkasteltaessa kappaleiden reunoilta löytyy mitättömän pieniä kolmioita. Nämä ylimääräiset kolmiot eivät näy kaukaa katsottuna, mutta raskauttavat mallia. Planeetan erotetut pinnat yhdistämällä ja turhat kärkipisteet poistamalla 3D-mallista saatiin noin 30 % kevyempi lähes identtisellä ulkonäöllä. Planeettojen manuaalinen läpikäyminen vie aikaa ja vaivaa, mutta se on tehokas tapa optimoida kolmioiden määrää pelissä. Joissain kentissä hajoavia planeettoja voi olla useita samaan aikaan näytöllä, jolloin ylimääräinen 30 % säästö kolmioissa kertaantuu nopeasti hyvinkin suureksi luvuksi.

Testitilannetta varten tehtiin asteroidikentän 3D-malli Blenderissä. Optimoimattomaan skeneen asteroidimallin jokaisessa asteroidissa oli 320 kolmiota. Tämä kopioituna asteroidikentäksi muodostaa noin 260 000 kolmiota. Kopioidut asteroidit muokattiin yhdeksi ainoaksi objektiksi ja vietiin Unityyn. Tällöin se käyttää turhaan kolmioverkkomuistia sen sisältäen vain kopioita kahdesta siemenasteroidista. Optimoituun asteroidikenttään käytettiin Blenderin Decimate toimintoa. Toiminnon avulla asteroidikentän kolmiot vähennettiin noin puoleen alkuperäisestä. Läheltä katsottuna kolmioiden vähentäminen vaikutti hieman asteroidikentän ulkonäköön, mutta tarkasteltaessa oikealta etäisyydeltä eroa ei huomaa. Tämä tarkasteluetäisyydestä johtuva yksityiskohtien määrä on tärkeää testata kolmioden määrää vähennettäessä,

sillä oikein tehtynä malleista voi saada yli puolet kolmioista pois ilman visuaalisen ulkoasun kärsimistä.

3.3.2 Varjostimet

Varjostinpuolella Unitysta ei löytynyt Gravitoidin tarkoitukseen sopivaa varjostinta. Unityn mukana tulee monta erilaista mobiilille tehtyä varjostinta, mutta kaikissa näissä oli ongelmana varjostimen muokkausmahdollisuudet. Koska mobiililaitteella toimivasta varjostimesta pitää tehdä mahdollisimman kevyt ja tehokas, oli esimerkiksi erillisen väriarvon lisääminen otettu kokonaan pois käytöstä. Tämä muodostui Gravitoidin tapauksessa ongelmalliseksi, sillä melkein kaikki objektit käyttävät pintamateriaalinaan vain väriarvoa, sekä muutamia heijastusparametrejä. Suunnitteluvaiheessa käytössä on ollut Unityn oletusvarjostin, joka tarjoaa hyvin laajan kirjon eri muokkaussominaisuuksia. Tämän perusvarjostimen koko on 319 riviä ja se sisältää monta sellaista ominaisuutta mitä Gravitoidissa ei tulla tarvitsemaan. Varsinaista peliä testatessa Unityn oletusvarjostin on toiminut moitteettomasti kappaleiden renderöimisessä. Unityn varjostimet toimivat datalähtöisesti (data-driven) eli varjostinkoodista käydään vaan muokatut osa-alueet läpi. Tällöin tyhjäksi jätetyistä kohdista ei tule ylimääräisiä kutsuja. **[56]** Jatkossa se tullaan kuitenkin korvaamaan myöhemmin esiteltävällä ja testattavalla kustomoidulla varjostimella.

Unity tarjoaa myös mobiilioptimoituja varjostimia. Erilaisia mobiilivarjostimia on 12 kappaletta eri käyttötarkoituksiin. Normaaleille objekteille käytössä on joko mobiili diffuusi (Mobile Diffuse) tai mobiili kyhmytetty diffuusi (Mobile Bumped Diffuse) varjostimet. Nämä varjostimet ovat huomattavasti oletusvarjostinta kevyempiä mobiili diffuusi varjostimen ollessa 32 riviä ja mobiilin kyhmytetyn diffuusin ollessa 37 riviä pitkiä. Esimerkiksi kyhmytetyn diffuusivarjostimen heijastuksen laskeminen hoidetaan seuraavasti:

```
void surf (Input IN, inout SurfaceOutput o) {
fixed4 c = tex2D(_MainTex, IN.uv_MainTex);
o.Albedo = c.rgb;
o.Alpha = c.a;
o.Normal = UnpackNormal(tex2D(_BumpMap, IN.uv_MainTex));
}
```

Ylläolevasta funktiosta voi huomata, että varjostimen väriarvot tuodaan tässä tapauksessa *_MainTex* tekstuurikartasta, eikä pelkän väriarvon syöttöä ole. Gravitoidin tapauksessa edellämainittu menetelmä on huono ratkaisu lähes jokaisen peliobjektin käyttäessä varjostukseen pelkkää väriarvoa tekstuurien sijaan. Varjostukset lasketaan kappaleen pinnalle ilman normaalikarttoja, joten niidenkin pitäminen mukana on turhaa. Ratkaisuna on tehdä oma varjostinohjelma, mikä sisältää pelkät väri- ja heijastusarvot. Yleensä standardityyppinen Shader Lab syntaksinen varjostin käyttää pintakuvien laskemiseen HLSL:n sukulaiskieltä Cg:tä. Cg kirjoitetaan Shader Lab syntaksin sisälle seuraavasti:

```
CGPROGRAM
//Cg koodi
ENDCG
```

Unityn Cg syntaksi käyttää usein operaatioiden tekemiseen tekstuuritiedostojen hakua, vaikkei varsinaista tekstuuria olisikaan. Varjostin on kuitenkin mahdollista kirjoittaa siten, että Cg syntaksin ohittaa kokonaan. Alla oleva varjostin karsii Unityn tarjoamia graafisia ominaisuuksia pois sisältäen ainoastaan oleellisimmat ominaisuudet.

```
Shader "Custom/StandardShader" {
Properties {
_Color ("Color", Color) = (1,1,1,1)
_Emission ("Emissive Color", Color) = (0,0,0,0)
_Shininess ("Shininess",Range(0.00, 1)) = 0.7
_SpecColor("Spec Color", Color)=(1,1,1,1)
}
SubShader {
Pass{
Material{
Diffuse [_Color]
Ambient [_Color]
Specular [_SpecColor]
Emission [_Emission]
Shininess [_Shininess]
}
Lighting On
SeparateSpecular On
```

```
}
}
FallBack "Diffuse"
}
```

Ylläolevasta varjostimesta on karsittu Cg osio kokonaan pois ja kappaleiden valaistus lasketaan yhdellä ainoalla läpiviennillä. **[47]** Kappaleelle annetaan väri, hehkun väri, kiilto sekä kiillon väri. Mikäli hehkua ei kappaleessa tarvitse, voi sen värin jättää mustaksi ja silloin kappale ei hohda ollenkaan. Tämäkin on turha tarkistus siinä vaiheessa mikäli hehkua ei kappaleelle tarvitse, mutta mikäli sen arvoa ei muuta, se poistetaan käännettäessä varjostimesta, jolloin sitä ei ajon aikana tarvitse tarkastaa. Valaistus lasketaan kiinteällä valaistuslaskennalla (fixed function) verteksikohtaisesti. Kun jokaiseen erityistapaukseen tehdään juuri omaa työtään vastaava varjostin, varmistetaan, ettei piirtovaiheessa tarvitse käydä ylimääräisiä tarkastuksia läpi ja näin prosessi nopeutuu. Mikäli kappaleeseen liitetyltä varjostimelta halutaan jotain erikoisominaisuuksia valon tai varjon käsittelyyn, kannattaa siitä tehdä kokonaan uusi varjostinohjelma ja käyttää sitä. Tällöin varjostuksia laskiessa ei tarvitse pysähtyä pohtimaan ylimääräisiä ehtoja, jotka vievät turhaan prosessointiaikaa. Ylläoleva varjostin tulee toimimaan Gravitoidissa kustomvarjostimena sen hyvin yksinkertaistetun varjostinfunktionsa takia. Mahdollisesti muuhun piirtämiseen tästä kustomvarjostimesta ei ole sen käyttäessä yksinkertaistetumpaa valaistuslaskentaa muokattavien sijaan. Kuitenkin testeistä saatu visuaalinen ilme on ollut Gravitoidiin juuri oikeanlainen, miksi varjostin sopii siihen loistavasti. Kuvio 16 esittää varjostimista saadut visuaaliset erot. Kustomvarjostin on siinä vasemmalta katsottuna kolmas.

Väriarvot olisi mahdollista siirtää tekstuurien avulla objekteille yhdellä väriatlastekstuurilla. Tekstuuriatlas toimii siten, että jokainen 3D-objekti viedään takaisin editointiohjelmaan ja UV kartoitetaan käyttämään tekstuuriatlaksen haluttua kohtaa. Tämä on hyvin työläs prosessi ja tekee myöhemmässä vaiheessa värien muuttamisesta työlästä. Unityn oman mobiilivarjostimen testiympäristö vaatii jokaiselle kappaleelle tekstuuritiedostot. Testitilanteessa tarvittavia tekstuurivärejä kappaleilla oli vain kaksi, joten tekstuuriatlaksen tekeminen jätettiin pois.

Toisena erikoisvarjostimena toimii taustaplaneetoilla ja asteroideilla oleva varjostin. Lisävarjostimen teko todettiin tarpeelliseksi sen sisältämän reunahehkun takia. Reunahehku on kappaleen reunoille laskettava värimuunnos, jonka ansiosta taustakappaleet saadaan sulautumaan taka-alalle. Saman tyylisen efektin voisi saada aikaiseksi käyttämällä sumua, mutta se on laskennallisesti raskaampaa eikä sen käyttöä siksi suositella mobiilisovelluksissa. Varjostin löytyy liitteestä A. koodit. Reunavalaistusvarjostin käyttää Cg-osiota ja tarvitsee näin tekstuuritietojen hakemisen. Lisä elementtinä on *_RimColor* sekä *_RimPower* arvot. Ensimmäinen on editorissa annettava väriarvo reunoille ja toisella liukuvalla arvolla voidaan säätää kuinka kaltevalle pinnalle väriä lisätään. Kaltevuuskulma lasketaan pinnan normaalitiedoista kameran katselukulmaan suhteutettuna.

Jokainen varjostin antaa aavistuksen erilaisen visuaalisen ilmeen kappaleille. Kuvio 16 demonstroi eri varjostimien vaikutuksen testitilanteen taustaplaneetalle. Varjostimet ovat vasemmalta oikealle: Unityn oletusvarjostin, Unityn mobiilivarjostin, kustomoitu varjostin sekä taustavarjostin. Tausta-asteroideille tarkoitettu reunavalaistuksen sisältävä varjostin esitetään hyvin hillityn reunavalaistuksen kanssa. Mobiilivarjostimessa ja kustomoidussa varjostimessa huomataan planeetan päällisen kiillon puuttuvan. Kustomivarjostimeen sitä on mahdollista lisätä lisäämällä *_Shininess* arvoa ja muuttamalla *_SpecColor* väriä kirkkaammaksi.

Kuvio 16. Varjostimien visuaaliset erot

3.3.3 Tekstuurit

Tekstuuritiedostojen kanssa saa olla tarkkana etenkin mobiililaitteille kehiteltäessä. Huolimattomalla teksturoinnilla saa helposti hitaasti lataavan ja tökkivän version. Yleensä hidastumisen syynä on näkyvyyteensä nähden aivan liian suuret tekstuurit, epämääräisen kokoiset

tekstuurikuvat sekä lukuisten erillisten tekstuuritiedostojen hausta johtuva läpivientikutsujen määrä. Jokainen kappaleen erillinen tekstuuri pitää ennen ruudulle piirtämistä hakea laitteen muistista. Mikäli jollain objektilla on esimerkiksi 20 erillistä tekstuuria, tarvitaan jokaisen hakemiseen erillinen kutsu. Useat pienet tekstuurit kannattaa yhdistää yhdeksi isoksi tekstuuriatlakseksi. Tekstuuriatlas on yksi iso tekstuuritiedosto, joka sisältää lukuisia pienempiä tekstuureita. Näin jokainen objekti jakaa saman tekstuurin eri osia jolloin lukuisten pienten tekstuurien lataamisen sijaan tarvitsee ladata vain yksi iso tekstuuri. **[57]** Tekstuurien siirtäminen manuaalisesti tekstuuriatlakseen voi olla kuitenkin työlästä. Atlaksen tekemistä voi myös automatisoida joko Unityn lisäosakaupasta löytyvillä lisäosilla tai erillisen mallinnusohjelman omilla työkaluilla.

Tekstuurikoko on toinen tärkeä osa-alue pelinäkymää optimoitaessa. Vaikka tekstuurikoko on kasvanut myös mobiililaitteilla ja epäsäännöllisen kokoiset tekstuurit piirtyvät normaalisti, on suositeltavaa suunnitella tekstuuritarkkuus ja koko lopulliseen tuotokseen sopivaksi. Ideaalitilanteessa pelinäkymän jokaista piirrettävää pikseliä kohti on kartoitettu yksi tekseli. **[57]** Todellisuudessa tällaiseen päästään harvoin. Teksturoinnissa kannattaa kiinnittää huomiota yhtenäiseen yleisilmeeseen tarkkojen tekstuurien sijaan. On miellyttävämpi katsoa hieman epätarkkaa mutta yhtenäistä näkymää, kun törmätä todella tarkkojen tekstuurien ja epätarkkojen tekstuurien ristiriitaiseen maisemaan.

Mip kartat (mipmap) ovat hyvä ja nopea tapa lisätä sulavuutta peliskenaarioon. Ne ovat progressiivisesti pienenevä versio samasta tekstuurista. Pienempiä versioita käytetään, mikäli piirrettävä kohde on kaukana kamerasta. Vaikka ylimääräiset mip kartat pitää tallentaa tekstuurimuistiin ja näin varata 33 % enemmän tekstuurimuistia, on niiden tuoma performanssihyöty haittoja suurempi. Koskela Timo ym. painottaa teoksessaan maksimitekstuurikoon selvittämisen tärkeyttä. **[58]** Liian suuret tekstuuritiedostot raskauttavat renderöntiprosessia ja mip karttojen ollessa päällä on mahdollista ettei suurimpia tekstuurikokoja edes käytetä peliskenaarioissa.

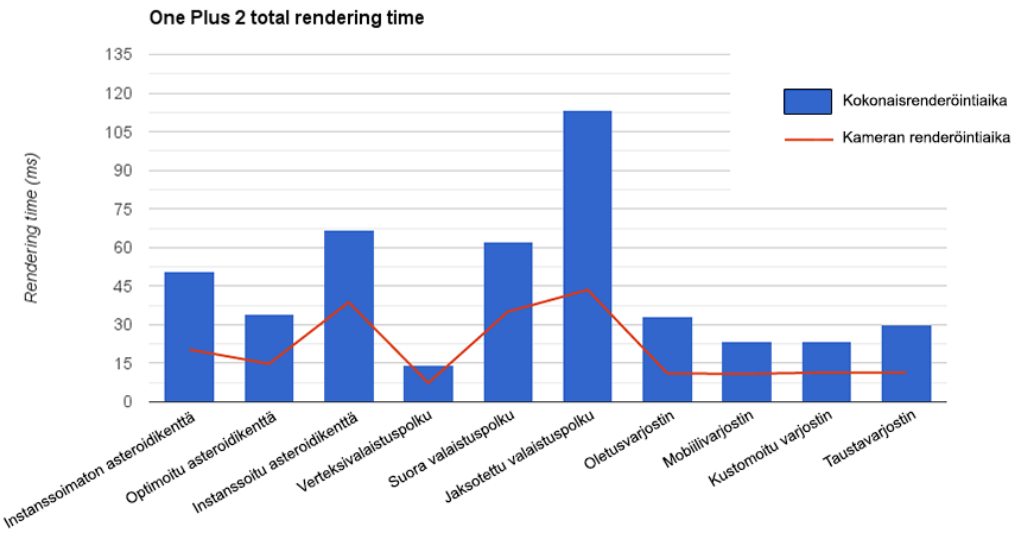
3.4 Tulosten analysointi

Tulokset analysoidaan kahden testilaitteen testidatasta. Analysointi jaetaan kolmeen aliosastoon. Ensimmäisessä osiossa käydään läpi kolmioiden määrän vaikutus testilaitteiden suorituskykyyn. Aliluvussa tarkastellaan kolmen ensimmäisen testiskenen tuloksia niiden käsitellessä samaa testitilannetta. Testiskenet ovat instanssoimaton asteroidikenttä, optimoitu asteroidikenttä, sekä instanssoitu asteroidikenttä. Toisessa osiossa tarkastellaan valaistuksen vaikuttamista laitteiden suorituskykyyn. Tässäkin on kolme testiskeneä valaistuspolkujen mukaan: verteksivalaistus, suora valaistus, sekä jaksotettu valaistus testitapaukset. Viimeinen kappale käsittää neljän eri varjostinskenen testitilanteet. Testattavana on Unityn tarjoama oletus- ja mobiilivarjostin, sekä aliluvussa 3.3.2. esitellyt kustomoitu varjostin ja taustavarjostin. Pääfokus testitilanteissa on laitteistojen saavuttamissa ruudunpäivitysnopeuksissa. Tarkastelussa on myös ruudun keskimääräinen piirtoaika ja sitä verrataan kameroiden käyttämään kuvan piirtoaikaan.

3.4.1 Kolmioiden määrän testaustulokset

Ensimmäiset testiskenet käsittelevät kolmioiden määrän testausta. Molempien laitteiden testit pyörivät kymmenen minuuttia ennen tietojen kirjaamista tasaisen rasituksen varmistamiseksi. Kuviot 17 ja 18 kuvastavat ruutujen keskimääräistä piirtoaikaa. Kuvioissa oleva punainen viiva kuvastaa kameroiden ruudunpäivitysaikaa, mikä on osa kokonaisruudunpäivitysaikaa. Kuviot 19 ja 20 esittää keskimääräiset ruudunpäivitysnopeudet eri skeneissä virhearvoineen. Kolmioiden määrää testaavat skenet ovat kuvioissa vasemmalta katsottuna ensimmäiset kolme pylvästä. Seuraavat kolme tarkastelee valaistuksen vaikutusta ja viimeiset neljä varjostimien vaikutusta suorituskykyyn. Kuvioiden lisäksi Taulukko 2. näyttää päätelaitteiden skenekohtaiset ruudunpäivitysnopeudet, laitteistojen keskiarvollisen ruudunpäivitysnopeuden sekä niiden erot testilaitteiden välillä.

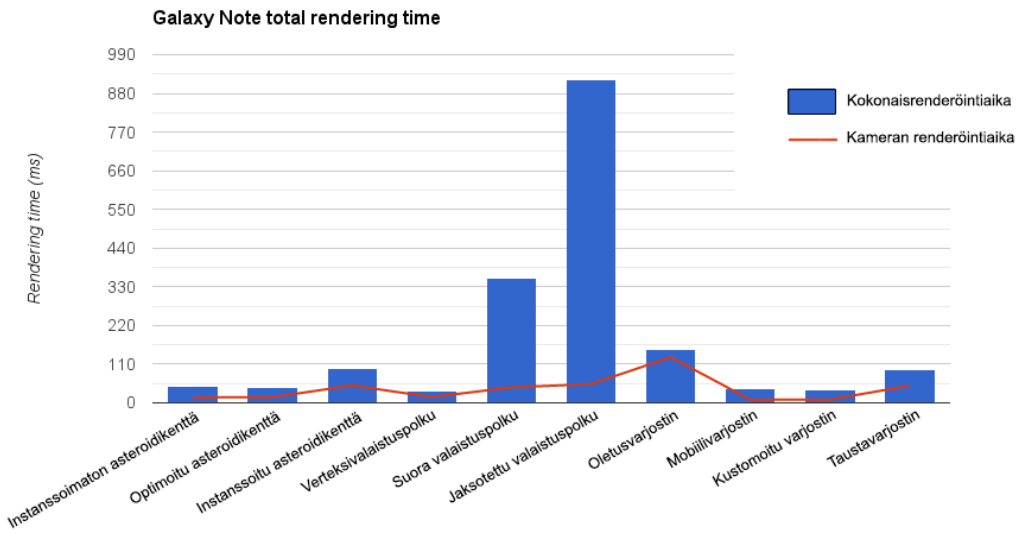
Odotusten mukaan instanssoimaton asteroidikenttä pyöri heikommin optimoituun asteroidikenttään verraten. Instanssoimattoman asteroidikenttä skenen ruudunpäivitysnopeus oli älypuhelimella keskimäärin 20,6 ruutua sekunnissa ja tablettitietokoneella 22,2 ruutua sekunnissa. Instanssoimaton asteroidikenttä oli ainoa testiskene, joka pyöri tablettitietokoneella



Kuvio 17. Ruutujen piirtoajan keskiarvot One Plus 2 älypuhelimella.

suuremmalla ruudunpäivitysnopeudella, kuin älypuhelimessa. Molemmissa testitilanteissa 3D-malleja oli testilaitteen muistissa 54,6 megatavua. Koska tablettitietokone sai suuremman ruudunpäivitysnopeuden, oli sen keskihajonta myös suurempi. Tämä selittyy osittain sillä, että mitä vähemmän aikaa ruudun piirtämiseen menee, sitä suuremmin ruudunpiirtoajan vaihtelut vaikuttavat ruudunpäivitysnopeuteen. Tablettitietokoneessa on kuitenkin kautta linjan suurempi hajonta ruudunpäivitysnopeudessa. Molempien laitteiden keskiarvoruudunpäivitysnopeus oli 21,4 ruutua sekunnissa ja laitteiden välinen FPS ero oli vain 1,6. Optimoitu asteroidikenttä pyöri älypuhelimessa keskimäärin 32,2:n ruudun sekuntinopeudella, mikä 56 % nopeampi. Tablettitietokoneessa ruudunpäivitysnopeuden nousu oli vain 1,9 ruutua sekunnissa, mikä on kolmioiden määrän vähentämiseen verrattuna todella vähän. Instanssoimattomassa testiskenessä kolmioita oli kuitenkin 255 000 kappaletta verraten optimoidun skenen 125 000 kolmioon. Kolmioiden määrällä ei siis ollut testitilanteissa suurikaan merkitys suorituskykyyn, vaan graafiset rasitteet tulevat jostain muualta.

Unityssä instanssoidun asteroidikentän suorituskyky oli yllätys, sillä se laski ruudunpäivitysnopeuden odotettua alemmaksi. Instanssoinnin pitäisi nostaa testiskenen ruudunpäivitysnopeutta pienemmän muistinkäytönsä vuoksi. Instanssoitu asteroidikenttä toimi kuitenkin oletuksia huonommin, vaikka skenen muistin käyttö olikin huomattavasti pienempi. 3D-

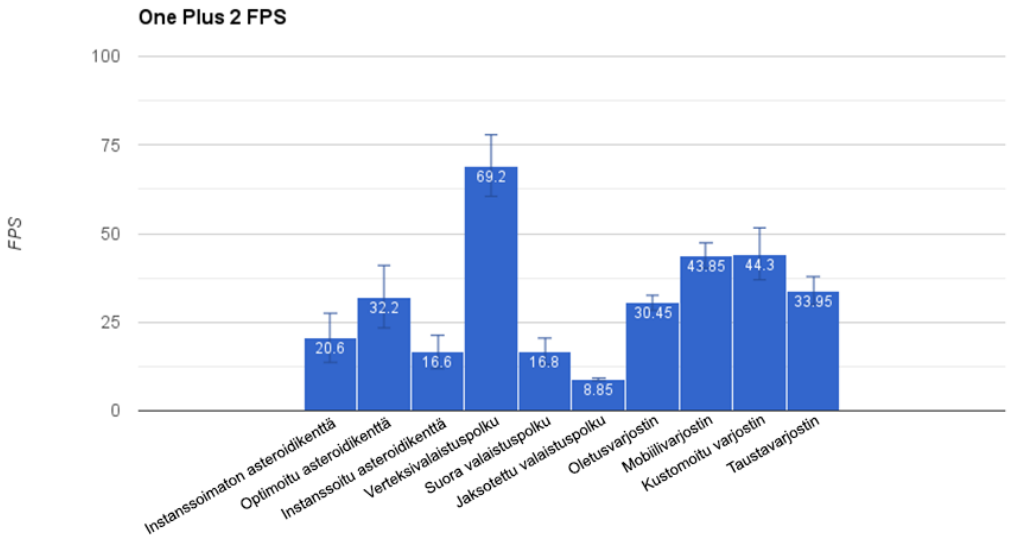


Kuvio 18. Ruutujen piirtoajan keskiarvot Samsung Galaxy Note tabletilla.

kappaleita oli instanssoimattomassa skenessä 54,6 megatavua ja instanssoidussa vain 1,7 megatavua. Kokonaismuistinkin käyttö putosi instanssoidussa skenessä 34 megatavua. Instanssoidun asteroidikenttäskenen keskimääräinen ruudunpäivitysnopeus oli älypuhelimessa 16,6 ruutua sekunnissa ja tablettitietokoneessa 10,7 ruutua sekunnissa. Syynä on huomattavasti noussut piirtokutsujen sekä uudelleenpiirtokutsujen määrät, jotka rasittavat CPU:ta. Älypuhelimessa instanssoimattoman ja optimoidun asteroidikentän testauksessa kameroiden piirtoaika oli noin 40 % koko ruudun piirtoajasta. Tablettitietokoneessa puolestaan 32 %:n ja 35 %:n välillä. Tablettitietokoneella oli älypuhelimeen verrattuna enemmän vaikeuksia valaistusten kanssa, mikä näkyy vielä selkeämmin valaistusta käsittelevässä luvussa. Instanssoidussa asteroidiskenessä älypuhelimen kameroiden piirtoaika oli 58 % kokonaispiirtoajasta. Vastaava aika oli tablettitietokoneella 51 %. Grafiikkapiirin uudelleenkutsut ja ylimääräiset läpivientikutsut tapahtuvat kameroiden piirtotoimintojen sisällä.

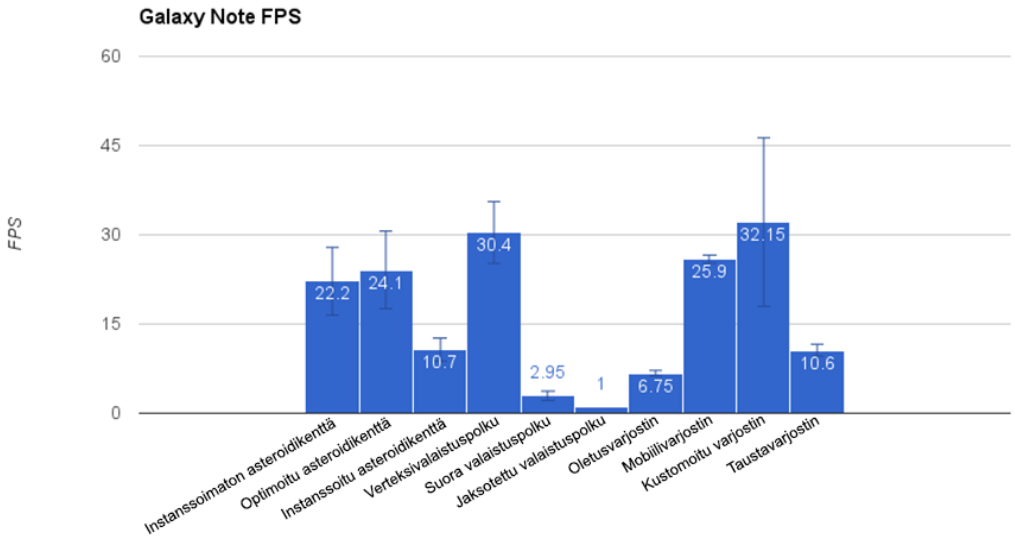
3.4.2 Valaistuksen testaustulokset

Valaistuspolkuja testattaessa verteksivalaistuspolun ruudunpäivitysnopeus on huomattavasti suurempi verrattuna suoraan ja jaksotettuun valaistuspolkuun. Tämä tulos oli odotusten mukainen ja johtuu siitä, ettei koko valaistuspolku tue kaikkia testissä käytettyjä valoja. Osa



Kuvio 19. One Plus 2 älypuhelimella saadut FPS keskiarvot.

valojen laskemisesta sivuutetaan siis kokonaan, mikä näkyy myös lopputuloksessa. Jos vertaa liitteessä olevan kuvion 22. valaistustilanteiden kuvia, voi niiden visuaalisesta ulkoasusta todeta verteksivalaistuspolun olevan paljon yksinkertaisemman näköinen muihin verrattuna. Testitilanteissa erikoista oli älypuhelimen ja tablettitietokoneen suuret erot ruudunpäivitysnopeudessa. Osa on selitettävissä tablettitietokoneen suuremmalla näytön resoluutiolla, mutta vanhemman kannan grafiikanprosessointiyksikkö on myös vastuussa pudonneesta ruudunpäivitysnopeudesta. Verteksivalaistuspolussa keskimääräinen ruudunpäivitysnopeus oli älypuhelimessa jopa 69,2 ruutua sekunnissa. Se on neljä kertaa nopeampi suoran valaistuspolun 16,8:n ruutuun sekunnissa ja melkein kahdeksan kertaa nopeampi jaksotetun valaistuspolun keskimääräiseen 8,85 ruutuun sekunnissa. Tablettitietokoneessa eroavaisuudet olivat hieman hillitympiä, vaikkakin syy siihen on suoran ja jaksotetun valaistuspolkujen todella hitaassa ruudunpäivitysnopeudessa. Verteksivalaistuksen testiskenaario pyöri tablettitietokoneessa 30,4 ruudun sekuntinopeudella, kun taas suoran valaistuspolun skenessä ruudunpäivitysnopeus oli 2,95 ruutua sekunnissa ja jaksotetun valaistuspolun testissä vain 1 ruutu sekunnissa. Verteksivalaistuspolun ruudunpäivitysnopeus erosi suuresti älypuhelimen ja tablettitietokoneen välillä. Ruudunpäivitysnopeuden ero oli jopa 38,8 ruutua sekunnissa testilaitteiden keskivertopäivitysnopeudesta. Valaistuspolkujen näin huono ruudunpäivitysnopeus tuli testatessa yllätyksenä. Älypuhelimessa verteksivalaistuspolussa kameroiden piirtoon meni 51



Kuvio 20. Samsung Galaxy Note tabletilla saadut FPS keskiarvot.

% koko ruudun piirtoajasta. Tablettitietokoneessa vastaava lukema oli 48 %.

Unity suosittelee käyttämään suoraa valaistuspolkua mobiilipeleissä. Syynä on verteksivalaistuspolun huono visuaalinen ulosanti ja jaksotetun valaistuksen raskaampi varjostinoperaatio, mikä laski ruudunpäivitysnopeuden myös älypuhelimessa todella alas (8,85 FPS). Tämän lisäksi jaksotetun valaistuspolun testiskenessä näkyy selkeästi valaistuksesta johtuvat artefaktit mitä suorassa valaistuspolussa ei näy. Valaistusartefaktit vaihtuivat myös testilaitteesta riippuen. Älypuhelimessa ongelmia oli lähinnä valojen leikkaamisessa. Tablettitietokone värjäsi tämän lisäksi koko valaistuskohteen sinertäväksi. PC laitteella skene toimi sulavalla 57,9 FPS nopeudella ilman minkäänlaisia visuaalisia anomalioita. Kuviossa 21 näkyy, älypuhelimen ja tablettitietokoneen eriävät valaistusartefaktit. Mikäli mobiililaitteille harkitsee jaksotettua valaistuspolkua, joutuu visuaalista ilmettä koeajamaan jatkuvasti päätelaitteella. Skenen jatkuva kääntäminen mobiiliaitteeseen on aikaa vievää ja laitteistosta riippuvat erot lopputuloksessa vaatii useamman laitteen käyttämistä testauksessa.

Testitilanteissa päätelaitteet suoriutuivat myös suoran valaistuspolun kanssa huonosti. Älypuhelin käytti keskimäärin 61,8 millisekuntia yhden ruudun piirtämiseen. Tästä ajasta 56 % oli kameroiden käyttöön käytettyä aikaa. Tablettitietokone suoriutui älypuhelinta huonommin keskimääräisen ruudun piirtoajan ollessa 355,3 millisekuntia. Ruutujen piirtoajasta

Kuvio 21. Deferred Rendering valaistuspolun artefaktit. Vasemmalla One Plus 2

vain 12,3 % oli kuitenkin kameroiden käyttämää piirtoaikaa. Valaistuksen laskeminen lukeutuu Profilerissa Graphics.PresentAndSync osion alle. Valitettavasti Profiler ei anna tarkempaa tietoa valaistuksen laskentaan liittyvistä toimista. Jaksotetun valaistuspolun testiskenessä älypuhelin käytti keskimäärin 113.2 millisekuntia yhden ruudun piirtämiseen. Kameroiden piirtäminen käytti 38 % koko ruudun piirtoajasta. Tablettitietokone kulutti vastaavasti keskimäärin 918,8 millisekuntia yhden ruudun piirtämiseen tiputtaen ruudunpäivitysnopeuden hyvin lähelle yhtä kuvaa sekuntia kohden. Tästä ajasta kameroiden piirtoon meni keskimäärin vain 53 ms, mikä vastaa vain 5.8 %:a kokonaispiirtoajasta. Suoran ja jaksotetun valaistuspolun testitilanteiden huono ruudunpäivitysnopeus johtui siis liian monen valon aiheuttamasta grafiikkaprosessorin ylikuormittumisesta. Keskimääräinen ruudunpäivitysnopeus laitteiden välillä oli jaksotetulla valaistuspolulla vain 4,925 ruutua sekunnissa. Valaistuksen laskentaan käytetty muisti oli jokaisessa skenessä kohtuullisen pieni 78 - 80 Mb välillä. Reaaliaikaisesti laskettavia valoja pitäisi karsia todella rajusti, mikäli kyseisen skenen ajattelisi siirtää julkaistavaan versioon. Monen valonlähteen tilanteissa esilaskettu valaistus tulee todella tärkeäksi tekijäksi, sillä staattisissa kohteissa valaistuksen esilaskeminen tiputtaa reaaliaikaisen valojen laskemisen pois nopeuttaen laskuprosesseja huomattavasti. Tekstuurikarttoihin lasketut valot lisäävät tekstuurimuistin määrää, mutta yhdessäkään testiskenessä muistinkäyttö ei ole vielä tullut esteeksi.

3.4.3 Varjostimien testaustulokset

Varjostimissa testitulokset olivat älypuhelimessa tasaisia kautta rinnan. Tablettitietokoneen ruudunpäivitys toimi hyvin mobiilivarjostimen ja kustomoidun varjostimen kanssa, mutta huonosti Unityn oletusvarjostimella sekä taustavarjostimella. Älypuhelinta testatessa vaikutti siltä, että varjostimesta riippumatta testattavan skenen ruudunpäivitysnopeus oli kiitettävä, eikä oletusvarjostimen vaihtaminen ole välttämätön toimenpide mobiilioptimomisessa. Ruudunpäivitysnopeudet olivat hyväksyttäviä Unityn oletusvarjostimen tarjotessa 30,45 FPS ja mobiilivarjostimen 43,85 FPS. Kustomiotu varjostin pyöri 44,3 ruutua sekunnissa ja taustavarjostin 33,95 ruutua sekunnissa. Tablettitietokone pudotti kuitenkin oletusvarjostimen ruudunpäivitysnopeuden niin alas, ettei sitä voitaisi hyväksyä julkaistavaan peliin. Oletusvarjostimen kanssa ruudunpäivitysnopeus oli vain 6,75 ruutua sekunnissa. Mobiilivarjostin toimi jo paremmin tarjoten 25,9 ruutua sekunnissa ja kustomvarjostin toimi jopa 32,15 ruutua sekunnissa ollen jo mobiililaitteelle kiitettävä ruudunpäivitysnopeus. Taustavarjostin tiputti tulosta taas laskien päivitysnopeuden vain 10,6 ruutuun sekunnissa. Varjostimien testauksessa ilmeni muita testitilanteita suuremmat laitteistojen väliset suorituserot. Ruudunpäivitysnopeuden erot laitteiden välillä olivat 12,2 - 23,7 ruutua sekunnissa. Normaalisti laitteiden väliset eroavaisuudet olivat alle 10 ruutua sekunnissa. Oletusvarjostimen ruudunpäivitysnopeus oli älypuhelimessa pahimmillaan jopa 23,7 ruutua sekunnissa nopeampi. Suurin laitteistokohtainen ero tuli kuitenkin verteksivalaistuspolussa sen ollessa jopa 38,8 ruutua sekunnissa.

Ruudunpäivitysnopeuden hitautta voidaan selittää ainakin läpivientikutsujen määrällä, sekä piirtokutsujen määrällä. Piirtokutsujen määrä tippui mobiilivarjostimessa ja kustomvarjostimessa 69:stä kutsusta 36:een mobiilivarjostimessa ja 33:een kustomvarjostimessa. Varjostimia testatessa ilmeni Profilerin toimintaan kohdistuva mielenkiintoinen havainto. Kustomoidun varjostimen testiskene näytti Profilerissa kolmioiden määräksi vain 242 500 kolmiota ja mobiilivarjostin 243 300 kolmiota. Oletusvarjostimessa ja taustavarjostimessa kolmioiden määräksi rekisteröityi 965 400 kolmiota. Kaikki neljä varjostinskeneä ovat täysin identtisiä toistensa kanssa. Ainoastaan 3D-mallien materiaalit vaihdettiin testattavan varjostimen sisältävään materiaaliin. Syy on mahdollisesti valaistuksen laskennassa, sillä esimerkiksi oletusvarjostimella testattaessa efektivalojen poistaminen tiputtaa kolmioiden määrän kustomivarjostimessa näkyvään 242 500 kolmioon. Vastaavasti kustomvarjostimessa efektivalojen

poistaminen ei tiputtanut kolmioiden määrää ollenkaan. Syynä tähän on mahdollisesti kustomivarjostimen ja mobiilivarjostimen karsittu valaistuslaskenta, missä Profiler ei lue mahdollisesti kaikkia tietoja operaatioista. Kolmioiden määrän vaihtelu vaatisi lisätutkimusta. Taulukko 2 esittää vielä testilaitteiden ruudunpäivitysnopeudet, sekä laitteistojen keskivertoruudunpäivitysnopeuden ja erotuksen eri skeneissä.

Skene:	K.A. OnePlus 2	K.A. Galaxy Note	Laitteiden K.A. FPS	Erotus
Instanssoimaton:	20,6	22,2	21,4	1,6
Optimoitu:	32,2	24,1	28,15	8,1
Instanssoitu:	16,6	10,7	13,65	5,9
Verteksivalaistus:	69,2	30,4	49,8	38,8
Suora valaistus:	16,8	2,95	9,875	13,85
Jaksotettu valaistus:	8,85	1	4,925	7,85
Oletusvarjostin:	30,45	6,75	18,6	23,7
Mobiilivarjostin:	43,85	25,9	34,875	17,95
Kustomoitu varjostin:	44,3	32,15	38,225	12,15
Taustavarjostin:	33,95	10,6	22,275	23,35

Taulukko 2. Testilaitteiden keskiarvoFPS, sekä erotus

3.5 Havaintoja ja suosituksia

Testituloksia tarkasteltaessa osa oletuksista osui kohdalleen, osassa tuli eriäviä tuloksia ja osassa ilmeni odottamattomia lopputuloksia. Kolmioiden määrällä oli testiversioissa suhteellisen vähän vaikutusta lopputulokseen. Kolmioiden määrään verrattuna tärkeämpää on keskittyä valaistuksen määrän ja laatuun sekä erillisesti hallittavien kappaleiden määrän. Mitä enemmän dataa joudutaan siirtämään CPU:n ja GPU:n välillä, sitä raskaammaksi kyseinen skene tulee. Unityn oletusvarjostimet tekevät suhteellisen hyvää työtä itsessäänkin mutta jos osaamista on, kannattaa peliin tehdä omat varjostimet. Varjostin tarvitsee kuitenkin kirjoittaa vain kerran ja sen jälkeen sen vaikutukset siirtyvät kaikkiin sitä käyttäviin kappaleisiin. Varjostimia suunniteltaessa kannattaa ottaa huomioon pelin visuaalinen ulkoasu, sillä se voi ra-

joittaa vaihtoehtoja. Esimerkiksi testitilanteessa olevan kustomvarjostimen visuaalinen ulosanti on juuri haluttu Gravitoid mobiilisovellukseen, mutta toisenlaiseen peliin se ei mahdollisesti kävisi tekstuuritiedostojen käsittelyn puuttuessa kokonaan. Myös kustomvarjostimen tarjoama valaistuspolku jättää joitain visuaalisia ominaisuuksia pois käytöstä, mitkä voivat haitata tyylillisesti eriävissä pelissä. Valmiita varjostimia ja varjostimien kustomointiratkaisuja löytyy muun muassa Unityn lisäosakaupasta. **[59]**

3D-mallien optimointia on mahdollista automatisoida, mutta silloin optimointi voi vaikuttaa lopputulokseen muuntamalla mallien topologiaa ei halutulla tavalla. Manuaalinen kappaleiden optimointi on työlästä, mutta tuottaa taatusti halutun visuaalisen lopputuloksen. Optimoinnin tulisikin tapahtua luonnollisena jatkumona 3D-mallien iteroinnissa lopulliseen ulkonäköönsä. 3D-mallien optimoinnin automatisoinnissa Unity tarjoaa muun muassa tarkkuustasojen muokkaustoiminnot, sekä tasokohtaiset kolmioiden poistotoiminnot (per-layer culling). Mikäli pelin 3D-ympäristössä on staattisia kappaleita, kannattaa ne ehdottomasti merkitä Unityssä staattisiksi. Normaalikarttojen ja teksturoinnin avulla peliobjekteista voidaan tehdä näyttäviä pienemmällä kolmiomäärällä. Piirtotasojen avulla näkymäfrustrumin ulkopuolelle jäävät kappaleet voidaan poistaa käytöstä vähentäen piirtokutsujen määrää.

Valaistuksen testaaminen ja esilaskenta skeneihin on testien pohjalta erityisen tärkeää sen ollessa erityisesti mobiililaitteissa todella raskasta laskea reaaliajassa. Valaistuksen optimointi havaittiin testitilanteiden avulla todella tärkeäksi osa-alueeksi. Vaikka verteksivalaistus oli ruudunpäivitysnopeudeltaan toimivin, ei sen visuaalinen ulkoasu pärjännyt kilpailussa muille valaistuspoluille. verteksivalaistuspolkua kannattaa harkita ainoastaan siinä vaiheessa, jos aikoo kehittää 3D-pelejä vanhemmalle mobiililaitteistolle tai jos sen tarjoama nimenomainen visuaalinen ulkoasu on haluttu. Suoralla valaistuspolulla saadaan myös verteksivalaistuksen tyylinen ulkoasu aikaiseksi ilman sen rajoituksia, joten sen käyttäminen on suositeltavaa. Jaksotettua valaistuspolkua voi suositella ainoastaan PC- tai konsolijulkaisuihin laskutehojen ollessa tarpeeksi suuret sen sulavaan laskemiseen. Testitilanteiden yllättävät artefaktit kertovat, miten erilailla eri mobiililaitteet ymmärtävät valaistuspolun käsittelyn.

Tekstuureita ei testitilanteessa testattu niiden puuttuessa Gravitoidista kokonaan. Tekstuurit voivat aiheuttaa pullonkauloja pelin suorituskykyyn, sillä ne voivat viedä hyvinkin paljon muistia ja etenkin yksittäisten tekstuuritiedostojen kanssa aiheuttaa ylimääräisiä piirtokut-

suja. Teksturoinnissa tärkeää on pitää yhtenäinen visuaalinen ilme esimerkiksi tekstuurin tarkkuuksien kanssa.

4 Johtopäätökset

Tässä pro gradu työssä testattiin eri optimointimenetelmiä mobiililaitteille suunnitellun pelin optimoimiseksi. Kolmessa eri testiskenaariossa selvitettiin kolmioiden, valaistuksen ja varjostimien vaikutus mobiililaitteen suorituskykyyn. Teoriaosuudessa käytiin läpi 3D-mallien muodostamiseen liittyvät olennaiset seikat, sekä renderöintiliukuhihna Unity-pelimoottorista mobiililaitteissa toimivaan OpenGL ES grafiikkarajapintaan. Testitilanteissa kolmioiden määrän vähentämisellä oli oletettua pienempi vaikutus suorituskykyyn. Unityssä instanssointi aiheutti ruudunpäivitysnopeuden tippumisen pelimoottorin käsitellessä jokaista instanssoitua peliobjektia yksi kerrallaan. Staattiset kappaleet kannattaa mahdollisuuksien mukaan tuoda yhtenä isompana 3D- tiedostona pelimoottorissa kopioimisen sijaan. Mikäli pelikappaleille tulee joitain dynaamisia ominaisuuksia, on niiden instanssointi Unity-pelimoottorissa tärkeää. Tarkkuustasojen muokkaustoiminnot kannattaa hyödyntää eri kappaleille ja tasokohtaisten kolmioiden poistotoiminnot on hyvä hyödyntää monimuotoisissa ympäristöissä. Kolmioiden määrää pystyy vähentämään kappaleissa tehokkaasti normaalikartoilla sekä tekstuuritiedostoilla.

Valaistusten testauksessa testattiin Unityn tarjoamat kolme valaistuspolkua. Valojen lisäämisen todettiin olevan helposti mobiililaitteiden toimintakykyä halvaannuttava työvaihe. Esirenderöidyt valokartat ovat lähes pakollista mobiililaitteille kehiteltäessä. Unityssä oletusarvoisesti päällä oleva suora valaistuspolku kannattaa pitää mobiililaitteissa päällä verteksivalaistuspolun tarjotessa visuaalisesti heikkotasoisempaa jälkeä ja jaksotetun valaistuspolun tarjotessa erilaisia valaistusartefakteja mobiililaitteesta riippuen. Valaistuksen esilasekenta ja valonlähteiden määrän rajoittaminen ovat ensisijaisia työvaiheita mobiililaitteille kehitettäessä.

Varjostimien vaihtoa suositellaan vähintään valmiiksi tarjottuihin mobiilivarjostimiin. Varjostimien tehokkuutta on helppo testata sillä kappaleiden materiaaleja on helppo vaihtaa keskenään. Mikäli kehitystyössä ei ole resursseja tehdä omia varjostimia, tarjoaa Unity-mobiilivarjostimet valmiina ja ne ovat hyvin optimoituja. Mobiilivarjostimien visuaalinen tarjonta on rajattua, mutta sen tuoman ruudunpäivitysnopeuden kasvu on sen arvoista. Omaa varjostinta kannattaa harkita siinä vaiheessa, mikäli haluaa optimoitua tehokkuutta visuaa-

lisilla lisäominaisuuksilla. Valmiita varjostimia löytää Unityn lisäosakaupasta, sekä internetistä.

Tekstuurien optimoinnin testaaminen ohitettiin tässä työssä Gravitoid mobiilipelin käyttäessä pääosin teksturoimattomia 3D-malleja. Tekstuurien optimointia käytiin kuitenkin pintapuolisesti läpi lähdemateriaalien turvin. Tekstuurien optimoinnissa huomio kannattaa kiinnittää tekstuurien kokoon ja määrään skenessä. Huonompi tekstuurien resoluutio toimii paremmin jos se on yhtäläistä muun ruudulla näkyvän materiaalin kanssa. Tekstuureita tehtäessä tekstuurien koko kannattaa pitää kahden potensseissa sillä vaikka Unity tukee muunkin kokoisia tekstuureja, se maksimoi tekstuurimuistin käytön mahdollistaen suuremmat määrät tekstuureja tekstuurimuistiin. Tekstuurien määrää kannattaa vähentää luomalla tekstuuriatlaksia. Näin vältetään turhia piirtokutsuja CPU:n ja GPU:n välillä. Philip Rideout neuvoo kirjassaan "iPhone 3D"testaamaan tekstuurien vaikutusta suorituskykyyn poistamalla ajon aikana tekstuurit käytöstä. **[60]** Tekstuurit saa Unityssä pois päältä joko skriptin avulla tai manuaalisesti testitilanteessa poistamalla tekstuuritiedot kappaleiden materiaaleista. Lukuisten eri materiaalitiedostojen kanssa manuaalinen tekstuuritiedostojen poistaminen voi olla työlästä. Tekstuurien optimointi on prosessi mikä jatkuu koko pelinkehityksen elinkaaren ajan.

Gravitoid mobiilisovellukseen tullaan implementoimaan testitilanteissa testatut varjostimet, sekä testeissä olevat asteroidikenttien muokatut versiot. Valaistustestauksesta saatiin hyviä käytänteitä eri kenttien valaistuksen suunnitteluun siten, että se ei raskauta päätelaitteita suunnattomasti. Jatkotutkimuksen tarpeita ilmeni muun muassa jaksotetun valaistuspolun testauksessa ilmenneistä valoartefakteista, sekä varjostimia testatessa ilmenneistä kolmion määrien vaihteluista.

Lähteet

[01] Mäyrä Frans, Mobile Games, John Wiley & Sons, Inc, (2015)

[02] Newzoo, Global Report: Us And China Take Half Of $113BN Games Market In 2018, https://newzoo.com/insights/articles/us-and-china-take-half-of-113bn-games-market-in-2018/, (2013), urldate: 17.11.2016

[03] Seok-Woo Shin, Sang-Hyeon Park, Yang-Woo Park, Seung-Il Moon, The Improvement plans of the Mobile Game Industry based on Mobile Web 2.0, SERSC, (2013)

[04] Unity, Unity public-relations: Thirty-four percent of top games are made with Unity, https://unity3d.com/public-relations, (2016), urldate: 17.11.2016

[05] Antochi Iosif, Juurlink Ben, Vassiliadis Stamatis, Liuha Petri, GraalBench: A 3D Graphics Benchmark Suite for Mobile Phones, ACM, (2004)

[06] Dickinson Chris, Unity 5 Game Optimization, Packt Publishing, (2015)

[07] Blender, History of Blender, https://www.blender.org/foundation/history/, (2013), urldate: 17.11.2016

[08] Mark J. P. Wolf, The Video Game Explosion: A History from PONG to Playstation and Beyond, ABC-CLIO, (2008)

[09] Gaming History, Bradley Trainer, http://www.arcade-history.com/?n=bradley-trainer&page=detail&id=330, (2013), urldate: 17.11.2016

[10] Serious Game Classification, The Bradley Trainer, http://serious.gameclassification.com/EN/games/14320-The-Bradley-Trainer/index.html, (2016), urldate: 17.11.2016

[11] Wikipedia, Battlezone (1980 video game), https://en.wikipedia.org/wiki/Battlezone_(1980_video_g (2016), urldate: 17.11.2016

[12] Brian Penzone, I, Robot Official Registry, https://web.archive.org/web/20080211120900/home.colu (2008), urldate: 17.11.2016

[13] Flickr, I, Robot (1983 / Dave Theurer / arcade), https://www.flickr.com/photos/daniel-rehn/8665131939, (2013), urldate: 17.11.2016

[14] Rogers Scott, Level Up!, Wiley, (2014)

[15] Kushner David, Masters Of Doom, Random House, (2004)

[16] Daniel Valente De Macedo, Maria Andreia Formico Rodrigues, Experiences with Ra-

pid Mobile Game Development Using Unity Engine, Computers in Entertainment, (2011)

[17] PwC, Video games Key insights at a glance, PwC, (2015)

[18] Pulli Kari, Aarnio Tomi, Miettinen Ville, Roimela Kimmo, Vaarala Jani, Mobile 3D Graphics with OpenGL ES and M3G, Elsevier Inc., (2008)

[19] Jukka Räbinä, On a Numerical Solution of the Maxwell Equations by Discrete Exterior Calculus, University of Jyväskylä, (2014)

[20] Unity Manual 5.4-V, 3D formats, https://docs.unity3d.com/Manual/3D-formats.html, (2016), urldate: 17.11.2016

[21] Tomas Akenine-Möller Eric Haines, Real-Time Rendering Second Edition, A K Peters, Ltd., (2002)

[22] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, T. R. Evans, Reconstruction and Representation of 3D Objects with Radial Basis Functions, ACM, (2001)

[23] Merlo Alessandro, Dalcò Luca, Fantini Filippo, Game engine for Cultural Heritage, IEEE, (2012)

[24] Mike Smithwick, Mayank Verma, Pro OpenGL ES for Android, Apress, (2012)

[25] Fletcher Dunn, Ian Parberry, 3D Math Primer for Graphics and Game Development, Wordware Publishing, Inc., (2002)

[26] Wikipedia, 3D Cartesian coordinate handedness, https://en.wikipedia.org/wiki/Cartesian_coordinate_system#/media/File:3D_Cartesian_Coodinate_H (2013), urldate: 17.11.2016

[27] Goldstone, Will, Unity game development essentials, Packt Pub, (2009)

[28] Unity Manual 5.4-V, ShaderLab: Culling & Depth Testing, https://docs.unity3d.com/Manual/SL-CullAndDepth.html, (2016), urldate: 17.11.2016

[29] Unity Manual 5.4-V, Meshes, https://docs.unity3d.com/Manual/class-Mesh.html, (2016), urldate: 17.11.2016

[30] Wikipedia, Example of a triangle mesh representing a dolphin., https://en.wikipedia.org/wiki/Triangle_mesh#/media/File:Dolphin_triangle_mesh.png, (2007), urldate: 17.11.2016

[31] Woo Jeong-Ho, Sohn Ju-Ho, Nam Byeong-Gyu, Yoo Hoi-Jun, Mobile 3D Graphics SoC:From Algorithm to Chip, Wiley-IEEE Press, (2010)

[32] Hugues Hoppe, Optimization of mesh locality for transparent vertex caching, Addison-Wesley Publishing Co., (1999)

[33] Jon Leech, OpenGL ES Version 3.1, Khronos group Inc., (2015)

[34] Unity Manual 5.4-V, Creating and Using Materials, https://docs.unity3d.com/Manual/Materials.html, (2016), urldate: 17.11.2016

[35] Luksch Christian, Tobler Robert F., Habel Ralf, Schwarzler Michael, Wimmer Michael, Fast Light-Map Computation with Virtual Polygon Lights, ACM, (2013)

[36] Mikkelsen, Morten, Simulation of Wrinkled Surfaces Revisited, (2008)

[37] Policarpo Fabio, Oliveira Manuel M., Comba Joao L. D., Real-Time Relief Mapping on Arbitrary Polygonal Surfaces, ACM, (2005)

[38] Wikipedia, Normal mapping used to re-detail simplified meshes., https://en.wikipedia.org/wiki/Normal_mapping#/media/File:Normal_map_example.png, (2006), urldate: 17.11.2016

[39] Inami Masahiko, Yanagida Yasuyuki, Tachi Susumu, Detailed shape representation with parallax mapping, ICAT, (2001)

[40] Mark Duchaineau, MurrayWolinsky, David E. Sigeti, Mark C. Millery, Charles Aldrich, Mark B. Mineev-Weinstein, ROAMing Terrain: Real-time Optimally Adapting Meshes, Los Alamos National Laboratory, (1997)

[41] Renaldas Zioma, Ole Ciliox, Unity: The Chase - Pushing the limits of Modern Mobile GPU, Unity Technologies, (2013)

[42] Segal Mark, Akeley Kurt, The OpenGL Graphics System: A Specification (Version 1.5), Silicon Graphics, Inc, (2003)

[43] Bailey Mike, What educators need to know about where openGL is and where it is going, Consortium for Computing Sciences in Colleges, (2015)

[44] David Blythe, Aaftab Munshi, OpenGL ES Common/Common-Lite Profile Specification, Khronos group Inc., (2008)

[45] Khronos Group, Vulkan Overview, Khronos Group, (2016)

[46] Wes McDermott, Creating 3D Game Art for the IPhone with Unity: Featuring Modo and Blender Pipelines, Taylor and Francis, (2010)

[47] Unity Manual 5.4-V, ShaderLab Syntax, https://docs.unity3d.com/Manual/SL-Shader.html, (2016), urldate: 17.11.2016

[48] Unity Manual 5.4-V, Shader assets, https://docs.unity3d.com/Manual/class-Shader.html, (2016), urldate: 17.11.2016

[49] Unity blogs, Bringing DirectX 11 features to mobile in Unity 5.1, https://blogs.unity3d.com/2015/05/26/dx11-features-on-mobile/, (2015), urldate: 17.11.2016

[50] Unity Manual 5.4-V, Vertex Lit Rendering Path Details, https://docs.unity3d.com/Manual/RenderTech-VertexLit.html, (2016), urldate: 17.11.2016

[51] Unity Manual 5.4-V, Forward Rendering Path Details, https://docs.unity3d.com/Manual/RenderTech-ForwardRendering.html, (2016), urldate: 17.11.2016

[52] Unity Manual 5.4-V, Deferred shading rendering path, https://docs.unity3d.com/Manual/RenderTech-DeferredShading.html, (2016), urldate: 17.11.2016

[53] Unity Manual 5.4-V, Profiler window, https://docs.unity3d.com/Manual/Profiler.html, (2016), urldate: 17.11.2016

[54] Unity Manual 5.4-V, Rendering Statistics Window, https://docs.unity3d.com/Manual/RenderingStatistics.html, (2016), urldate: 17.11.2016

[55] Unity Manual 5.4-V, Optimizing graphics performance, https://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html, (2016), urldate: 17.11.2016

[56] Unity Manual 5.4-V, Standard Shader: Content and Context, https://docs.unity3d.com/Manual/StandardShaderContextAndContent.html, (2016), urldate: 17.11.2016

[57] Beets Kristof, Gustavsson Mikael, Olsson Erik, ShaderX7: Advanced Rendering Techniques, Cengage Learning, (2009)

[58] Vatjus-Anttila Jarkko M. Koskela Timo, Hickey Seamus, Effect of 3D Content Simplification on Mobile Device Energy Consumption, ACM, (2013)

[59] Unity, Asset Store, https://www.assetstore.unity3d.com, (2016), urldate: 17.11.2016

[60] Rideout Philip, iPhone 3D Programming Developing Graphical Applications with OpenGL ES, O'Reilly Media, (2010)

Liitteet

A Koodit

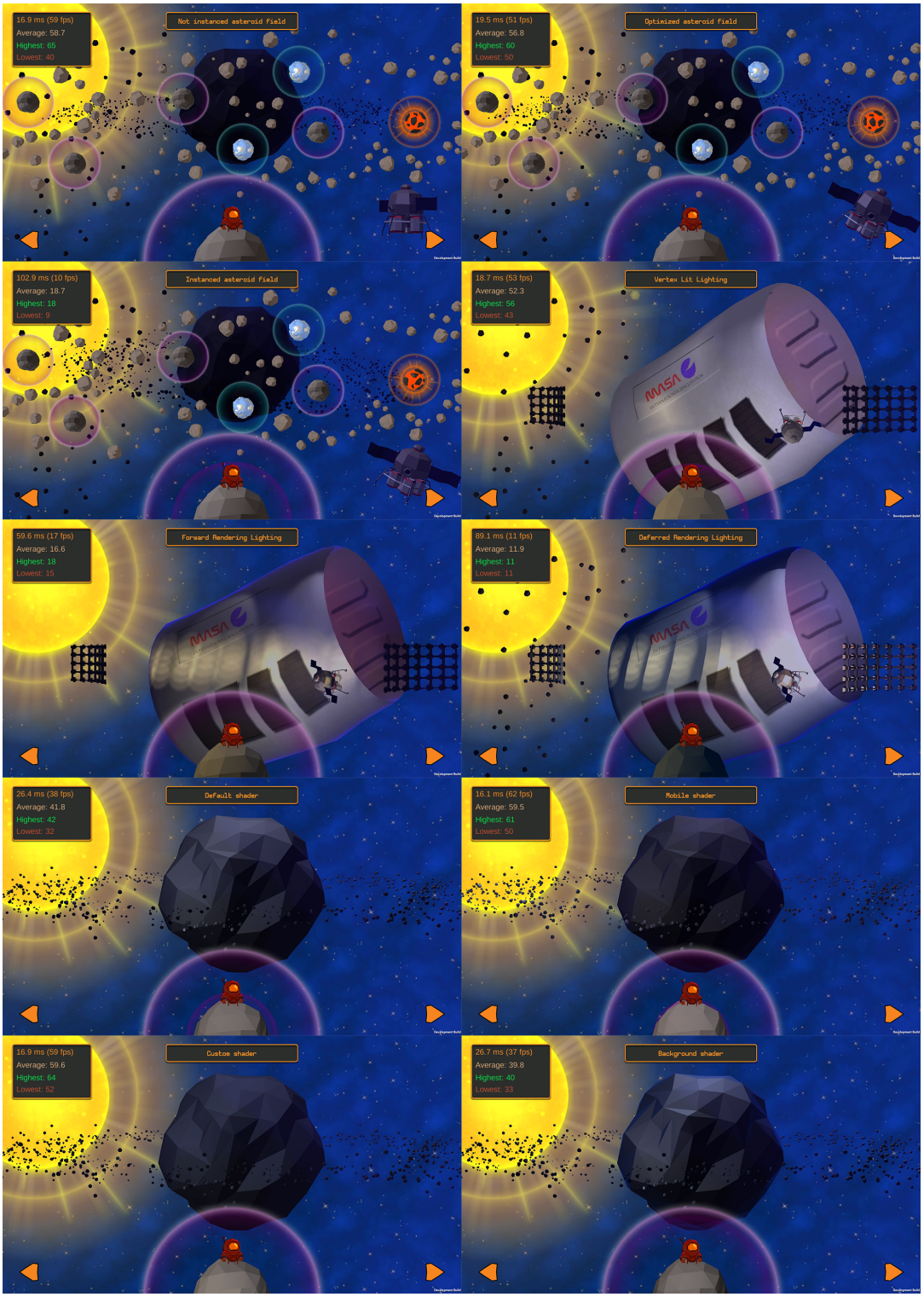
Alla oleva koodi on reunavalaistuksen laskeva varjostin.

```
Shader "Custom/MopiiliSheideri" {
Properties {
_Color ("Color", Color) = (1,1,1,1)
_MainTex ("Albedo (RGB)", 2D) = "white" {}
_Glossiness ("Smoothness", Range(0,1)) = 0.5
_Metallic ("Metallic", Range(0,1)) = 0.0

//Added Rim lighting
_RimColor ("Rim Color", Color) = (1, 1, 1, 1)
_RimPower ("Rim Power", Range(0.5, 9.0)) = 3.0
}
SubShader {
Tags { "RenderType"="Opaque" }
LOD 200

CGPROGRAM
// Physically based Standard lighting model, and enable shadows on
// all light types
#pragma surface surf Standard fullforwardshadows

// Use shader model 3.0
#pragma target 3.0

sampler2D _MainTex;
struct Input {
float2 uv_MainTex;

//Rim Lighting viewDirection
float3 viewDir;
```

```
};
// Using half and float, because its more efficient and uses less power to
// calculate on mobile
half _Glossiness;
half _Metallic;
float4 _Color;

half _RimPower;
float4 _RimColor;

void surf (Input IN, inout SurfaceOutputStandard o) {
// Albedo comes from a texture tinted by color
float4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
o.Albedo = c.rgb;

//RimColor from variables
half rim = 1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));
o.Emission = _RimColor.rgb * pow (rim, _RimPower);

// Metallic and smoothness come from slider variables
o.Metallic = _Metallic;
o.Smoothness = _Glossiness;
o.Alpha = c.a;
}
ENDCG
}
FallBack "Diffuse"
}
```

B Kuvakaappaukset

Ohessa kuva kaikista testiskeneistä One Plus 2 mobiililaitteella. Huomionarvoista on vasemman yläreunan ruudunpäivitysnopeudet Profilerin ollessa pois päältä



Kuvio 22. Kuvakaappaukset kaikista testiskeneistä